

序

真的是不经意的工作机会，才使得我有从事 Spring 方面的项目架构和开发经验，不只是 Spring，还有 Hibernate 和 Tapestry 也在其中。在 Java Web 框架中，成熟的 Tapestry 现在是前端开发领域中的一枝独秀。Hibernate 成为了 O/R Mapping 领域事实上的标准，尤其是它对 EJB 3.0 的巨大贡献。Spring 更不用说了，它使得开发者和具体 J2EE 平台技术处于“松耦合”的状态。它们的强强联手，使得开发软件产品过程确实是一种快乐，发自开发者内心的快乐。快乐是无限的，您也许不想错过。

Tapestry 推崇 OO 方式开发 Web 前端。它将 HTTP 底层 API 技术屏蔽起来，尤其是开发者再也不用直接面对 JSP 和 Servlet 了。Hibernate 使得 Java 开发者能够高效地对 RDBMS 进行 CRUD 操作，而不管各种 RDBMS 所运行的平台、版本如何。尤其是开发者再也不用直接面对 JDBC、SQL 语句了，至少大部分场合是如此。Spring 是各种 Web 框架的黏合剂，无论是在 Open Source 领域，还是在非 Open Source 领域。当然，Tapestry 和 Hibernate 也在 Spring 的操控范围内。架构级的 Spring 在主导着整个 J2EE 社区开发 Web 应用的方向。因此，Spring 是本书讨论的重点，但是 Tapestry、Hibernate 也将在本书的视野中。

Spring，中文含义暂且理解为“春天”。春天，意味着万物复苏，而 Spring 将担当此任。

Spring IoC，借助于依赖注入设计模式，使得开发者不用理会对象自身的生命周期及其关系，而且能够改善开发者对模式的使用。请记住，管理单个的对象本身不是件难事，难就难在处于 Team 中的各个对象，因为它们之间的关系往往是千变万化的。这正如开发者在不同团队中所处的不同角色一样。借助于 Spring IoC，不仅能够使得应用中对象的关系更加清晰、一致，而且还使得一切对象可控。最为重要的一点是，对象本身的生命周期及对象之间的关系不用再让开发者费神了。

Spring AOP，借助于 Spring 实现的拦截器，开发者能够实现以声明方式使用企业级服务，比如安全性服务、事务服务。请记住，AOP 能够合理地补充 OOP 技术，Spring AOP 合理地补充了 Spring IoC 容器。没有 Spring IoC 的 Spring AOP 是不完善的，没有 Spring AOP 的 Spring IoC 是不健壮的。借助于 Spring AOP，开发者能够高效地使用 J2EE 企业服务。声明式、基于元数据访问企业级服务，这些都是 Spring AOP 的操控范围。

Spring 服务抽象，借助于各种 J2EE API 抽象，使得开发者能够一致地使用 J2EE 技术，而不管具体是使用什么 J2EE API。借助于 Spring 服务抽象，使得应用代码大大减少，请记住“更少的代码、更少的 Bug”的原则。另外，通过研究 Spring 服务抽象，使得开发者能够快速掌握各种 J2EE API 的核心内容，因为 Spring 抽象服务是架构在 J2EE API 基础之上的。请记住，Template 方法实现是 Spring 服务抽象中至关重要的开发利器。缺少 Template 方法实现，会使 Spring 操作 JNDI、Transaction、JMS、Hibernate 等企业级服务显得生涩。

Spring IoC + Spring AOP + Spring 服务抽象，一起形成了 Spring。这样一个有机的整体，使得构建轻量级的 J2EE 架构成为可能，而且事实证明，非常有效。请记住，Team Work 往往比单打独斗更具威力。如果剥离 IoC、AOP 或服务抽象中的任何一者，则 Spring 必然受伤不轻。正因为 Spring 提供了这样一套完整的利器，才使得它备受开发者推崇，这也是它

同其他类似 Web 框架的区别所在。我们向来不相信有“银弹”，即使它存在，在它未到来之前，赶紧选择 Spring 吧。否则一旦“银弹”到来，您也可能措手不及！

如果您还没有接触过 Spring，则作为一名 J2EE 开发者，不熟悉 Spring，有点勉为其难。因为，架构级的 Spring 框架已成为主流，它已经冲入了 J2EE 的核心，并在引领整个 J2EE 开发、架构的方向。现在正是学习并使用 Spring 开发企业级 Java 应用的时机。如果您还在观望的话，即使您不购买这本图书，作者还是希望您能够去使用 Spring，从而体验 Spring 深层次的开发经验。基于我们的架构、开发经验，Spring 富有内涵。随着您对 Spring 的日益使用，您会慢慢喜欢上它。

本书总共分成 3 部分。第一部分，重点阐述 Spring 的架构。这部分内容循序渐进带领开发者进入 Spring 中。主要在于阐述 Spring IoC 和 Spring AOP。第二部分，重点阐述 Spring 的使用。这部分内容从简化 Java/J2EE 的角度出发，从 J2EE 平台各个技术层面分析，并给出大量的研究实例，对 Spring 提供的 API 进行阐述。主要在于阐述 Spring 对 J2EE API 提供的服务抽象。第三部分，重点阐述 Spring 高级专题。这部分内容重点对视图技术进行了研究，因为对于开发 Web 应用而言，前端界面的开发往往工作量很大。因此，使用合理的视图技术开发 Web 应用对于项目的成功与否很关键。另外，Web 应用的安全性往往也是企业应用中最为重要的需求之一，而用于 Spring 的 Acegi 安全框架很好地解决了这个问题，这也是第三部分重点研究的内容之一。

在架构软件、开发软件过程中，架构师（设计人员、开发者）往往会低估很多不重要的开发环节，最终导致了软件系统发布的延期。这其中，往往会有很多未预期的因素，比如人员变动、对新技术不熟悉、对产品需求的把握不准确，导致产品的延期。对于写书而言，也存在类似的因素。作者在编著《精通 Spring》这本书之前，也低估了各种因素而导致图书交付日期的延迟。但是，这些现在都不是问题，因为我们（与此图书相关的角色）都克服了所有的困难。父母对全家的悉心照顾，爱妻的支持，乖巧听话的、还未满周岁的女儿，出版社为保证图书质量而提出的、富有成效的修改建议，还有那些关心此书的所有朋友、读者都是保证本书能够顺利地并高质量地出版的重要前提。在将近半年的时间中，他们都默默为此书付出了很多、很多。作者很感谢他们，没有他们的付出，就不可能有此书的诞生。当然，也感谢 Spring 开发团队的努力。请记住，热情、激情、友情、亲情是我们生活、工作的动力。

将此书献给所有的 Java、J2EE 开发者。

J2EE 架构师 罗时飞

2005 年 3 月于广州

前 言

关于本套丛书

从来没有任何事物像互联网那样，对人类的活动产生如此深刻的影响，无论是政府、企业，以及个人，莫不如此。与此同时，IT 产业也正面临着一场变革——传统应用向基于 Internet/Web 的服务模式转化。

翻开历史，我们可以看到互联网的形成和发展就是以分布性、开放性和平台无关性为基础的，这是 Internet 与生俱有的属性。随着互联网应用的发展，又引入了诸如 RPC/COM/CORBA 等技术，但这些技术在实际应用中，又存在着很多不足和局限。它们的特定协议也难以通过防火墙，因而不适于在 Web 上应用开发。为了进一步开发基于 Web 的应用，相继出现了 Sun 公司的 Sun ONE (Open Net Environment 开放网络环境) 和 Microsoft 公司的 .NET 两大 Web 服务技术体系。其中，Sun ONE 以 Java 技术为核心，更接近或者满足于互联网在智能化 Web 服务上对分布性、开放性和平台无关性的要求，同时其在健壮性、安全性、组件化等方面也更为成熟稳定，获得了众多 IT 厂商和产品的支持，是目前惟一在市场上得到了广泛应用的技术体系。

Sun ONE 体系结构以 Java 语言为核心，包括 J2SE/J2EE/J2ME，并基于一系列开放和流行标准、技术及协议。要特别指出的是，Sun ONE 体系结构本身作为开放式体系结构，在得到 IBM/BEA/Oracle/Sybase 等这些 IT 巨擘支持的同时，更得到了互联网上 Open Source 社区的青睐。我们很容易地从网上免费获得和使用包括 Java 集成开发环境、Java 数据库，甚至是中间件 (Application Server) 服务器等产品，以及它们的源代码。这对于加速国内中小企业的信息化建设和自有知识产权产品开发、提高企业应用和软件行业的整体水平，无疑是一次难得的机会。

综观国内的技术发展，广大的 Java 程序开发人员及正在转向 Java 体系进行开发的技术人员虽然已面临这一令人激动和鼓舞的转型期，却苦于没有足够的相关资料和文献，尤其对国内的最新 Java 技术动态和技术现状知之甚少，而图书市场上 Java 的书籍尽管汗牛充栋，但精品罕见，能反映出 J2EE 及 Sun ONE 的框架全貌的书籍更是奇缺。

电子工业出版社计算机图书研发部为进一步推动国内 Java 技术的应用与发展，不失时机地推出了《开发专家之 Sun ONE》系列丛书。

本套丛书以 Sun ONE 整体架构为基础，全面体现了 Sun ONE 的技术核心——Java 的应用开发。丛书从各个角度深入 Java 应用开发的各个层面，涵盖了 Java 技术的所有重要思想和实践，体现了最新的 Java 技术进展和动态，大幅度提升读者的理论和应用水平。同时，丛书重点突出实用性。书中引入了大量的行业应用范例，使读者不仅能快速掌握开发技能，而且对于开发者进行综合系统分析也有所裨益。

关于本书

自从 J2EE 平台诞生，它就专注于企业级 Java 市场。这其中，出现了各种不同的新技

术方向、新基础框架，以及新技术社区。Open Source 社区在推动 J2EE 发展方面起了不可替代的作用。开发者都知道，该社区开发的产品紧跟最新的技术规范、针对 J2EE 规范本身，以及基于 J2EE 的应用中的缺陷能够快速给出最佳解决方案。最重要的一点是 Open Source 是开放的，相应产品的升级不需要应用开发者维护和升级。新的观点、技术框架在经过业界的使用和磨炼后，Open Source 产品便可以为企企业级 Java 应用提供基础支撑作用。

另外，开发者通过使用 Open Source 产品，可以获得最佳实践和业界的丰富经验，也不需要锁定在商用产品上，从而为开发者提供了最有保障的机制。

Spring 作为解决开发者和 J2EE 平台技术间的使用隔阂起到了很重要的作用。直接借助于 J2EE API 开发应用，使得开发者将大部分精力花费在同资源交互的处理上，而对实际应用系统的开发却不多，这使得开发 J2EE 效率低下的问题暴露无疑。开发者都知道，J2EE 平台技术解决了传统两层架构中的问题，即不存在平台服务。但是，这种服务的使用模式对于开发者而言，现有的 J2EE API 的复杂性使得开发者很为难。因此，Spring 试图消除上述隔阂。作为架构级的框架，Spring 是很出色的。无论开发者是否打算使用 Spring 开发 Java、J2EE 应用，研究 Spring 都是有必要的。

其一，Spring 将现有 J2EE 开发中存在的问题都摆出来了，这给开发者指引了正确的方向，从而避免犯错误。

其二，Spring 将这些问题或者部分解决，或者一直在解决中。借助于 Spring，开发者能够快速掌握使用 J2EE 技术的最佳实践。至少，开发者不会再误用 J2EE 技术了。

其三，通过研究 Spring，能够看到整个 Java/J2EE 的发展趋势，处于开发中的 J2EE 5.0 印证了这一点。因此，在某种程度上，Spring 是未来使用 J2EE 技术的缩影。

其四，Spring 将各种业界广泛使用的技术集成进来，以便于开发者使用。Spring 将各种 Open Source 和非 Open Source 技术、框架集成在一起。这也体现出 Spring 的通用性、易用性。

借助于 Spring 架构企业级 Java 应用，确实能够为开发者提供有益的帮助。随着对 Spring 使用的日益深入，开发者就越会体会到 Spring 为开发者架构、开发 Web 应用考虑得多么周到、细致。Spring 本身就是为开发者而来的，比如 Spring IoC、Spring AOP、Spring J2EE 服务抽象、Spring Acegi 等这些都是为开发者准备的开发利器。

优美的架构（比如，合理进行分层、各层又是有机的整体）、一致的开发模式（比如，借助于为 J2EE 服务提供的模板方法操作资源）、富有生机的 Spring 开发者社区（比如，新版本的发布周期非常短），这就是 Spring。

我们的联系方式如下：

咨询电话：(010) 68134545 68131648

电子邮件：support@fecit.com.cn

服务网址：<http://www.fecit.com.cn> <http://www.fecit.net>

通用网址：计算机图书、飞思、飞思教育、飞思科技、FECIT

飞思科技产品研发中心

目 录

第一部分 Spring 架构分析

第 1 章	Spring 启程.....	3
1.1	背景知识	3
1.2	运行 Spring 实例应用.....	3
1.2.1	实例 1: example1	4
1.2.2	实例 2: example2.....	7
1.2.3	实例 3: example3.....	8
1.2.4	实例 4: example4.....	9
1.3	Spring I/O 实用类	12
1.4	小结	13
第 2 章	安装和构建 Spring.....	15
2.1	获得二进制文件	15
2.2	基于源代码构建 Spring.....	17
2.2.1	基于 CVS 访问以获得 源代码	17
2.2.2	构建 Spring 框架.....	20
2.2.3	重要 Ant 任务	25
2.3	安装 Spring.....	27
2.4	小结	28
第 3 章	控制反转 (Spring IoC)	29
3.1	IoC 背景知识	29
3.2	Spring IoC.....	30
3.2.1	BeanFactory	30
3.2.2	ApplicationContext.....	39
3.3	IoC 其他内容	43
3.3.1	发布并监听事件	43
3.3.2	自定义 JavaBean 属性 编辑器	46
3.4	小结	48

第 4 章	面向方面编程 (Spring AOP)	49
4.1	AOP 及 Spring AOP	
	背景知识	49
4.2	Spring AOP 装备	51
4.2.1	Before 装备	52
4.2.2	After 装备.....	55
4.2.3	Throws 装备.....	58
4.2.4	Around 装备.....	61
4.3	ProxyFactoryBean	65
4.4	对象池	68
4.5	小结	71
第 5 章	深入 Spring 架构	73
5.1	架构概述	73
5.2	Spring 具体构件	74
5.2.1	Spring 上下文	74
5.2.2	Spring Web	75
5.2.3	Spring 数据访问 对象 (DAO)	76
5.2.4	Spring ORM	78
5.2.5	Spring Web MVC 框架	78
5.3	综合实例分析	78
5.3.1	实例概述	80
5.3.2	安装和配置 example11.....	83
5.3.3	架构分析	88
5.4	小结	92

第二部分 Spring 应用开发

第 6 章	命名服务——JNDI	97
6.1	背景	97
6.2	Spring 对 JNDI 提供的支持....	98

6.2.1	JndiObjectFactoryBean	99
6.2.2	JndiObjectTargetSource	102
6.2.3	JndiTemplate	105
6.2.4	JndiCallback	109
6.3	小结	110
第 7 章	事务服务——JTA	111
7.1	背景	111
7.2	Spring 对事务管理提供的支持	112
7.2.1	PlatformTransactionManager	113
7.2.2	声明式事务	117
7.2.3	编程式事务	133
7.3	小结	136
第 8 章	消息服务——JMS	137
8.1	背景	137
8.2	Spring 对 JMS 提供的支持	138
8.2.1	JmsTemplate	139
8.2.2	事务管理	164
8.3	小结	165
第 9 章	邮件服务——JavaMail	167
9.1	背景	167
9.2	Spring 对 JavaMail 提供的支持	167
9.2.1	使用 CosMailSenderImpl	168
9.2.2	使用 JavaMailSenderImpl	170
9.3	小结	172
第 10 章	企业 Bean 服务——EJB	173
10.1	背景	173
10.2	Spring 对 EJB 提供的支持	173
10.2.1	开发 EJB	176

10.2.2	访问 EJB	187
10.3	小结	189
第 11 章	持久化服务——DAO、JDBC、ORM	191
11.1	背景	191
11.2	Spring 对 DAO 提供的支持	192
11.3	Spring 对 JDBC 提供的支持	193
11.3.1	JdbcTemplate	193
11.3.2	DataSourceTransactionManager	200
11.3.3	连接数据库的方式	200
11.3.4	将 JDBC 操作建模为 Java 对象	201
11.4	Spring 对 ORM 提供的支持	206
11.4.1	Hibernate 介绍	207
11.4.2	Hibernate 集成支持	216
11.5	小结	224
第 12 章	任务调度服务——Quartz、Timer	225
12.1	背景	225
12.2	Spring 对 Quartz 提供的支持	225
12.2.1	QuartzJobBean 和 JobDetailBean 的使用	228
12.2.2	MethodInvokingJobDetailFactoryBean 的使用	233
12.3	Spring 对 Timer 提供的支持	238
12.3.1	ScheduledTimerTask 的使用	239
12.3.2	MethodInvokingTimerTaskFactoryBean 的使用	243

12.4	小结	247
第 13 章	远程服务	249
13.1	背景	249
13.2	Spring 对远程服务提供的支持	251
13.2.1	RMI 使能服务	251
13.2.2	Hessian 使能服务	259
13.2.3	Burlap 使能服务	267
13.2.4	HTTP Invoker 使能服务	273
13.3	Spring 对 Web 服务提供的支持	280
13.4	小结	291
 第三部分 Spring 高级主题		
第 14 章	视图技术集成	295
14.1	Spring Web MVC	296
14.1.1	配置 DispatcherServlet	297
14.1.2	开发及配置 Controller	298
14.1.3	配置 ViewResolver	300
14.1.4	配置 HandlerMapping	302
14.2	Struts	303
14.2.1	Spring JPetStore 的 ApplicationContext 集成方式	304
14.2.2	Spring 提供的集成方式	306
14.3	Tapestry	309
14.4	JSF	309
14.5	JSP 和 JSTL	309
14.6	Velocity 和 FreeMarker	310
14.7	XSLT	311
14.8	Tiles	311

14.9	JasperReports	312
14.10	文档视图	313
14.11	小结	313
第 15 章	Tapestry 集成	315
15.1	Tapestry 介绍	315
15.2	Page 和组件模板	318
15.3	创建 Tapestry 组件	320
15.4	Tapestry 校验子系统	320
15.5	管理服务器端状态	327
15.6	配置 Tapestry 应用	328
15.7	与 Spring 集成	329
15.8	小结	332
第 16 章	JSF 集成	333
16.1	Web 前端开发的趋势	333
16.2	JSF 介绍	334
16.3	Spring 和 JSF-Spring 提供的 JSF 集成	336
16.4	example29 实例研究	337
16.4.1	部署及使用	338
16.4.2	开发过程	343
16.4.3	Spring 提供的 JSF 集成能力	355
16.4.4	JSF-Spring 项目提供的 JSF 集成能力	355
16.5	小结	357
第 17 章	用于 Spring 的 Acegi 安全框架	359
17.1	Acegi 介绍	359
17.2	Acegi 架构及使用	362
17.2.1	构建 contacts 应用	362
17.2.2	Acegi 架构综述	370
17.2.3	Web 资源的认证	372
17.2.4	Web 资源的授权	377
17.2.5	配置 Acegi Servlet 过滤器	378
17.2.6	方法级的认证和授权	388

17.3 其他内容	389	A.4 代码使用	411
17.3.1 实现密码的加密 处理	391	附录 B spring-beans.dtd 的内容 模型	413
17.3.2 缓存用户信息	393	B.1 beans 节点	413
17.4 小结	394	B.2 bean 节点	414
附录 A 实例代码安装	395	B.3 constructor-arg 节点	417
A.1 代码说明	395	B.4 property 节点	419
A.2 钟情 JBoss	395	B.5 lookup-method 节点	419
A.3 工具下载与安装	396	B.6 replaced-method 节点	420
A.3.1 Spring IDE	396	附录 C 参考资料	421
A.3.2 Tapestry Spindle	400	后记	425
A.3.3 JBoss IDE	406		
A.3.4 Hibernate Synchronzier	411		

开发专家之

Sun ONE

第一部分 Spring 架构分析

本部分内容将带领开发者逐渐进入到架构级的 Spring 框架中。如果您还未接触过 Spring，则务必阅读第 1 章内容。因为，它会带您进入 Spring 殿堂。如果需要对 Spring 框架的安装和构建过程进行系统性研究，则不要错过第 2 章内容。第 3、4 章分别对 Spring IoC 容器、Spring AOP 实现进行了展示，它们是 Spring 框架的核心。在有了这些基础知识后，我们将进一步深入到 Spring 的架构中，即第 5 章内容。上述所有内容都是围绕 Spring 的架构展开的，而且本书的后续内容会持续深入到 Spring 的架构中。

从 Spring 的源代码到 Spring 的安装、构建过程，再到 Spring 提供的 IoC 和 AOP，这些内容都在此进行了深入的阐释。

当然，开发者在阅读第二、三部分时，可以翻阅这部分内容，从而能够加深对 Spring 架构的掌握程度。

第 1 章 Spring 启程

本章内容将给出若干个基于 Spring 开发的相关实例，使得开发者能够了解 Spring 基础知识，从而为深入了解 Spring 其他内容奠定基础。

全书实例代码的安装请开发者参考“附录 A 实例代码安装”。

1.1 背景知识

作为 Open Source 框架，Spring 是很令人振奋的。Spring 从一开始，直到现在，而且包括以后，将持续获得更多开发者的青睐，因为它是优秀的、是为开发者设计的实用框架。现存的 Open Source 框架很多，但是属于 Java/J2EE 架构级的优秀框架不多，包括现在流行的 Struts、Tapestry 等¹都不是。研究 Spring 的过程，是很有意义的一件事情，因为它将业界的各种开发经验（包括 Open Source 的和非 Open Source 领域的）融入到其中。

Spring 提供的控制反转（Inversion of Control, IoC²）和面向方面编程（Aspect-Oriented Programming, AOP³）插件式架构降低了应用组件之间的依赖性。借助于 XML 定义文件，开发者能够在运行时连接不同的应用组件。这对于单元测试特别有用，特别是那些需要针对不同客户实施不同配置的应用而言。

目前，存在的依赖注入类型有 3 种：基于设值（setter-based）方法、基于构建器（constructor-based）以及基于接口（interface-based）注入。Spring IoC 支持前两种，即借助于 Spring 开发者可以通过构建器，或者设值方法创建对象，并对对象的状态进行管理。

其中，依赖注入是 Spring 框架的基础。Spring 在基于依赖注入的基础之上，同时还提供了其他大量的功能，比如 Spring MVC 框架、事务管理框架、DAO 支持、支持主流的 O/R Mapping 工具、支持各种标准 J2EE 组件技术的集成（JMS/JavaMail/EJB/Web 服务等）、集成各种视图（Web 视图和非 Web 视图）技术。这也是使用 Spring IoC 容器优于其他 IoC 容器的理由。

1.2 运行 Spring 实例应用

为阐述清楚 Spring IoC 框架帮助开发者完成了哪些工作，本章给出的运行实例起到了很重要的作用，即开发者通过代码能够很清楚地看到 Spring 的具体内容。

¹ 开发者可以参考网站：<http://www.waferproject.org>，即 Web 应用框架项目。

² 控制反转（IoC），又称之为依赖注入（Dependency Injection, DI）。本书约定：它们的含义一致。至于有关 IoC 和 DI 的更详细解释，开发者可以参考网址：<http://www.martinfowler.com/articles/injection.html>。同时，本书第 3 章将深入研究 Spring IoC。

³ 本书第 4 章将深入研究 Spring AOP。

1.2.1 实例 1: example1

example1（所有的源代码位于 example1 项目中）并没有使用 Spring 框架。本章将通过一系列重构步骤，并最终借助于 Spring 提供的 IoC 框架来实现实例应用。接下来，给出大体步骤（详细内容，请参考 example1 项目）。

首先，开发者需要实现 FileHelloStr.java 类。

```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;

import java.util.Properties;

/**
 * 基于文件形式，读取 HelloWorld 所需的字符串
 *
 * @author luoshifei
 */
public class FileHelloStr {
    protected static final Log log = LogFactory.getLog(FileHelloStr.class);
    private String propfilename;

    public FileHelloStr(String propfilename) {
        this.propfilename = propfilename;
    }

    public String getContent() {
        String helloworld = "";

        try {
            Properties properties = new Properties();
            InputStream is = getClass().getClassLoader().getResourceAsStream(
                propfilename);
            properties.load(is);
            is.close();
            helloworld = properties.getProperty("helloworld");
        } catch (FileNotFoundException ex) {
            log.error(ex);
        } catch (IOException ex) {
            log.error(ex);
        }
    }
}
```

```
    }  
  
    return helloworld;  
}  
}
```

上述类实现了对传入的、名为 **propfilename** 的属性文件的读取工作，并能够打印出相关异常信息。

其次，开发者还需要实现 **HelloWorld.java** 类。

```
package com.openv.spring;  
  
import org.apache.commons.logging.Log;  
import org.apache.commons.logging.LogFactory;  
  
/**  
 * 获得 HelloWorld 字符串  
 *  
 * @author luoshifei  
 */  
public class HelloWorld {  
    protected static final Log log = LogFactory.getLog(HelloWorld.class);  
  
    public String getContent() {  
        FileHelloStr fhStr = new FileHelloStr("helloworld.properties");  
        String helloworld = fhStr.getContent();  
  
        return helloworld;  
    }  
}
```

上述类调用了 **FileHelloStr**。

第三步，开发者需要实现 **HelloWorldClient.java** 类，从而实现对 **HelloWorld** 类的调用。

```
package com.openv.spring;  
  
import org.apache.commons.logging.Log;  
import org.apache.commons.logging.LogFactory;  
  
/**  
 * HelloWorld 客户应用  
 *  
 * @author luoshifei  
 */  
public class HelloWorldClient {  
    protected static final Log log = LogFactory.getLog(HelloWorldClient.class);
```

```
public static void main(String[] args) {  
    HelloWorld hw = new HelloWorld();  
    log.info(hw.getContent());  
}  
}
```

第四步，开发者还需要编写 `HelloWorld.properties` 文件，其内容如下。

```
helloworld = "Hello World!"
```

最后，运行 `example1` 应用。通过两种方式可以运行 `HelloWorldClient` 客户应用。其一，通过 Eclipse IDE；其二，通过 Ant `build.xml` 文件的 `run` 任务。实例输出信息如下：

```
Buildfile: D:\workspace\example1\build.xml  
compile:  
run:  
[java] 2004-12-12 14:44:43 com.openv.spring.HelloWorldClient main  
[java] 信息: "Hello World!"  
BUILD SUCCESSFUL  
Total time: 2 seconds
```

其中，`example1` 使用到的 Ant `build.xml` 文件内容如下。

```
<?xml version="1.0"?>  
<project name="example1" default="run" basedir=".">  
    <path id="lib">  
  
        <fileset dir="../example1lib/jakarta-commons">  
            <include name="commons-logging.jar"/>  
        </fileset>  
  
    </path>  
  
    <target name="run" depends="compile" description="Run HelloWorldClient">  
  
        <java classname="com.openv.spring.HelloWorldClient" fork="yes">  
            <classpath refid="lib"/>  
            <classpath path="classes"/>  
        </java>  
  
    </target>  
  
    <target name="compile">  
  
        <mkdir dir="classes"/>  
  
        <javac destdir="classes" source="1.5" target="1.5"  
            deprecation="false" optimize="false" failonerror="true">  
            <src path="src"/>  
            <classpath refid="lib"/>  
        </javac>  
    </target>  
</project>
```

```

<copy todir="classes">
  <fileset dir="src">
    <include name="helloworld.properties"/>
  </fileset>
</copy>
</target>
</project>

```

通过上述过程，开发者可以看出：**HelloWorld** 明显依赖于 **FileHelloStr**。如果开发者需要通过其他途径获得“**Hello World!**”信息，则需要重构现有的 **FileHelloStr** 类，即通过更通用的 **HelloStr** 接口形式给出。一种较好的实现方式是将创建 **FileHelloStr** 对象的职责委派给 **HelloWorldClient** 客户。**example2** 实现了这种需求。

1.2.2 实例 2: example2

example2 所有的源代码位于 **example2** 项目中。为实现上述需求，开发者需要对 **example1** 完成如下几方面的重构（请开发者特别注意粗体内容）。

首先，创建 **HelloStr** 接口。

```

package com.openv.spring;

/**
 * HelloStr 接口
 *
 * @author luoshifei
 */
public interface HelloStr {
    public String getContent();
}

```

其次，声明 **FileHelloStr** 实现了 **HelloStr** 接口。

```
public class FileHelloStr implements HelloStr
```

FileHelloStr.java 的其他内容同 **example1**。

第三，重构 **HelloWorld** 类。

```

package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * 获得 HelloWorld 字符串
 *
 * @author luoshifei
 */
public class HelloWorld {
    protected static final Log log = LogFactory.getLog(HelloWorld.class);
    private HelloStr hStr;
}

```

```
public HelloWorld(HelloStr hStr) {
    this.hStr = hStr;
}

public String getContent() {
    return hStr.getContent();
}
}
```

最后，开发者需要重构 HelloWorldClient 客户应用。

```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * HelloWorld 客户应用
 *
 * @author luoshifei
 */
public class HelloWorldClient {
    protected static final Log log = LogFactory.getLog(HelloWorldClient.class);

    public static void main(String[] args) {
        FileHelloStr fhStr = new FileHelloStr("helloworld.properties");
        HelloWorld hw = new HelloWorld(fhStr);
        log.info(hw.getContent());
    }
}
```

至此，开发者可以运行客户应用了。具体输出结果同 example1，这里不再给出。

开发者应该注意到：HelloWorld 不再操作 FileHelloStr 了，而是对 FileHelloStr 实现的 HelloStr 接口进行操作。从而，现在的 HelloWorld 更加通用了。但是，现有的应用还是存在一个问题，即开发者必须在其实现的客户代码中创建 FileHelloStr 对象，并连接到 HelloWorld 中。因此，开发者需要借助于工厂类，以注入 HelloWorld 和 FileHelloStr 的依赖性。这正是 example3 需要解决的问题。

1.2.3 实例 3: example3

example3 所有的源代码位于 example3 项目中。为实现上述需求，开发者需要对 example2 完成如下几方面的重构（请开发者特别注意粗体内容）。

首先，创建 HelloWorldFactory.java 工厂。

```
package com.openv.spring;

/**
```

```
* 注入 HelloWorld 和 HelloStr 依赖性
*
* @author luoshifei
*/
public class HelloWorldFactory {
    public static HelloWorld getFileHelloWorld() {
        HelloStr hStr = new FileHelloStr("helloworld.properties");
        HelloWorld hw = new HelloWorld(hStr);

        return hw;
    }
}
```

其次，重构客户代码。

```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * HelloWorld 客户应用
 *
 * @author luoshifei
 */
public class HelloWorldClient {
    protected static final Log log = LogFactory.getLog(HelloWorldClient.class);

    public static void main(String[] args) {
        HelloWorld hw = HelloWorldFactory.getFileHelloWorld();
        log.info(hw.getContent());
    }
}
```

最后，运行客户代码，结果同上。

其中，HelloWorldFactory 负责创建和集成客户应用所需的对象。至此，开发者终于借助于依赖注入（HelloWorldFactory 类）实现了反转控制。

注意

上述 example1、example2、example3 都没有使用到 Spring。借助于 Spring 提供的核心工厂模式，开发者能够消除手工编写工厂类的需要。基于 Spring 的 example4 解决了这个问题，即将创建对象的工作交由 Spring 负责，从而消除了对工厂类、方法的需要。

1.2.4 实例 4：example4

对于开发过大型应用系统的开发者而言，他们都熟知：应用系统越大，相应的工厂类越多。一般情况下，工厂类都是简单的、仅提供静态方法和变量的单实例。它们将创建对

象，并将这些对象绑定在一起。显而易见，这将存在大量的重复代码。

Spring 框架最基本的一项功能就是，充当创建对象的工厂。其具体工作步骤如下：

- 读取并分析 Spring 配置文件（比如，基于 XML 文件格式）。
- 通过 Java 反射机制，创建并集成上述配置文件中定义的对象。
- 将创建的对象传回给开发者的应用代码。因此，开发者不用编写工厂类，其前提是需要使用 Spring 框架。

至于重构 example3，即开发 example4（所有的源代码位于 example4 项目中）的步骤如下。

首先，编写 Spring 配置文件，即 appcontext.xml（开发者也可以基于其他格式编写 Spring 配置文件，本书其他部分将讨论到相关内容）。

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

    <bean name="fileHelloWorld"
        class="com.openv.spring.HelloWorld">
        <constructor-arg>
            <ref bean="fileHello"/>
        </constructor-arg>
    </bean>

    <bean name="fileHello"
        class="com.openv.spring.FileHelloStr">
        <constructor-arg>
            <value>helloworld.properties</value>
        </constructor-arg>
    </bean>

</beans>
```

上述 Spring 配置文件的根元素为 beans，它含有一个或多个 bean 元素。其中，bean 元素用于描述应用代码中的 JavaBean 对象。

通过 name 属性能够惟一标识某 JavaBean。比如，在上述代码中，通过传入“fileHelloWorld”字符串能够访问到 HelloWorld 对象。另外，class 属性能够确定待实例化的类，比如上述代码中的“class=com.openv.spring.FileHelloStr”。当 Spring 创建 JavaBean 实例时，bean 的 constructor-arg 子元素值将传入到其构建器中，比如上述代码中的 helloworld.properties。如果构建器存在多个参数，则开发者可以使用 index 属性（或者 type 属性）指定对应的参数取值。

通过 ref 元素能够引用 Spring 配置文件中的其他已定义的 JavaBean。比如，上述代码中的 fileHelloWorld 引用了 fileHello。为将取值传递给构建器（或者设置方法），开发者需要使用 value 元素。其中，Spring 能够将 value 元素取值转换为相应的 Java 类型。

其次，重构 HelloWorldClient 客户代码。


```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

/**
 * HelloWorld 客户应用
 *
 * @author luoshifei
 */
public class HelloWorldClient {
    protected static final Log log = LogFactory.getLog(HelloWorldClient.class);

    public HelloWorldClient() {
        Resource resource = new ClassPathResource("appcontext.xml");
        BeanFactory factory = new XmlBeanFactory(resource);
        HelloWorld hw = (HelloWorld) factory.getBean("fileHelloWorld");
        log.info(hw.getContent());
    }

    public static void main(String[] args) {
        new HelloWorldClient();
    }
}
```

最后，运行 HelloWorldClient 客户应用。

Buildfile: D:\workspace\example4\build.xml

compile:

run:

```
[java] 2004-12-12 15:08:45 org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
[java] 信息: Loading XML bean definitions from class path resource
[appcontext.xml]
[java] 2004-12-12 15:08:46
org.springframework.beans.factory.support.AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'fileHelloWorld'
[java] 2004-12-12 15:08:46
org.springframework.beans.factory.support.AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'fileHello'
[java] 2004-12-12 15:08:46 org.springframework.beans.factory.support.
```

```
AbstractAutowireCapableBeanFactory autowireConstructor
[java] 信息: Bean 'fileHello' instantiated via constructor [public
com.openv.spring.FileHelloStr(java.lang.String)]
[java] 2004-12-12 15:08:46 org.springframework.beans.factory.support.
AbstractAutowireCapableBeanFactory autowireConstructor
[java] 信息: Bean 'fileHelloWorld' instantiated via constructor [public
com.openv.spring.HelloWorld(com.openv.spring.HelloStr)]
[java] 2004-12-12 15:08:46 com.openv.spring.HelloWorldClient <init>
[java] 信息: "Hello World!"
BUILD SUCCESSFUL
Total time: 1 second
```

开发者应该注意到, 上述输出结果同前面 3 个实例有所不同, 即除了应用本身的输出信息外, 还包含了 Spring 的输出信息。

通过输出信息, 可以获悉: 创建了两个单实例的 HelloWorld 和 FileHelloStr, 即 Spring 默认时仅创建单实例的 JavaBean, 通过 Spring 配置文件中 bean 元素的 singleton 属性能够控制创建 Java 实例的方式。

至于本实例中的 HelloWorldClient 客户应用的具体解释, 本书将在第 3 章给出。

1.3 Spring I/O 实用类

开发者在 example4 中使用了 ClassPathResource 类装载 appcontext.xml 文件。Spring 在整个框架中提供了 org.springframework.core.io 包, 供方便装载相关资源使用。当然, 无论是 XML, 还是文件, 还是 URL, core.io 包都提供了很好的支持。图 1-1 展示了其中的主要类。

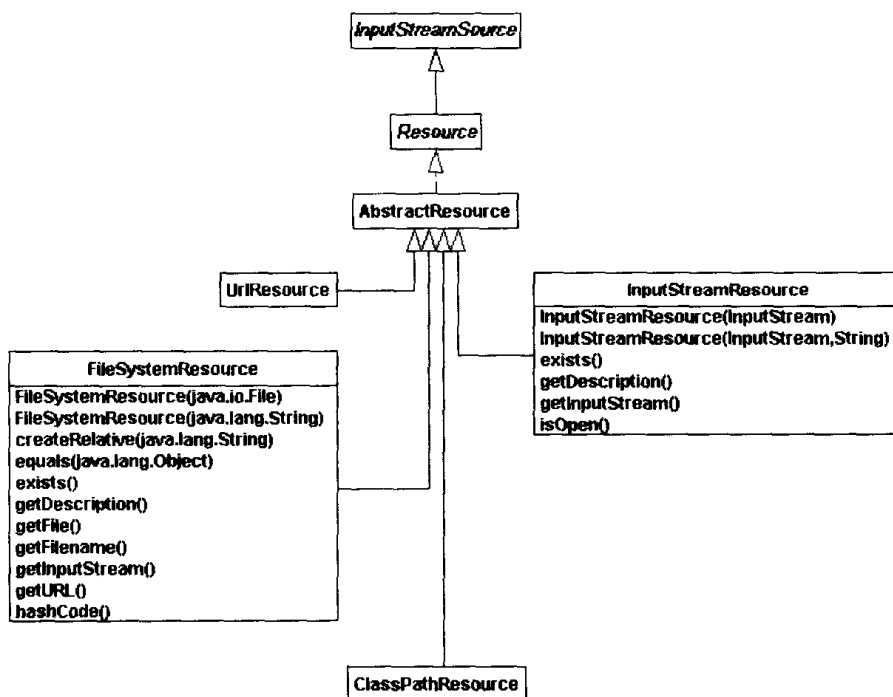


图 1-1 core.io 包所含类的类图

其中，提供了 Resource 接口的不同实现。比如，基于 URL 的 `UrlResource`、基于输入流的 `InputStreamResource`、基于文件系统的 `FileSystemResource`、基于应用 classpath 的 `ClassPathResource`。从图 1-1 可以看出，开发者可以从不同位置、以不同方式装载 Spring 配置文件。这对于基于 Spring 或非 Spring 的应用的单元测试、集成测试而言，特别有意义。

Resource 接口的具体内容见图 1-2 所示。

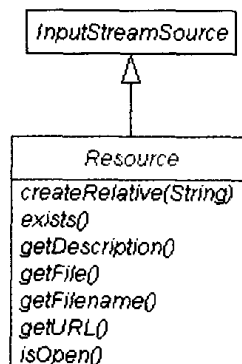


图 1-2 Resource 类图

1.4 小 结

本章通过 4 个实例，研究了 Spring 提供的 IoC 初步内容，Spring 提供的功能远不止这些。至此，开发者应该对 Spring 有了初步认识，并且能够根据本章内容，开发一些简单的 Java 应用。

另外，本章还介绍了 `org.springframework.core.io` 包。它为装载 Spring 配置文件提供了最直接的支持。借助于 `core.io`，开发者能够顺利地完成应用的单元测试和集成测试。

在深入 Spring 开发 Java/J2EE 应用之前，开发者应该掌握 Spring 的安装和构建。这正是第 2 章的研究内容。

第 2 章 安装和构建 Spring

Spring, 是企业级应用提供“一站式”服务的框架。

正如开发者所知, 在提供优秀的开发工具和采用先进的软件技术前提下, 开发出良好的软件产品也是不容易完成的任务。尽管 J2EE 平台承诺, 开发 Java/J2EE 应用很简洁、高效, 但实际情况不是如此。因为, 开发过程往往难于控制进度、开发效率低下, 而且很多开发者并没有真正用好 J2EE 组件技术。

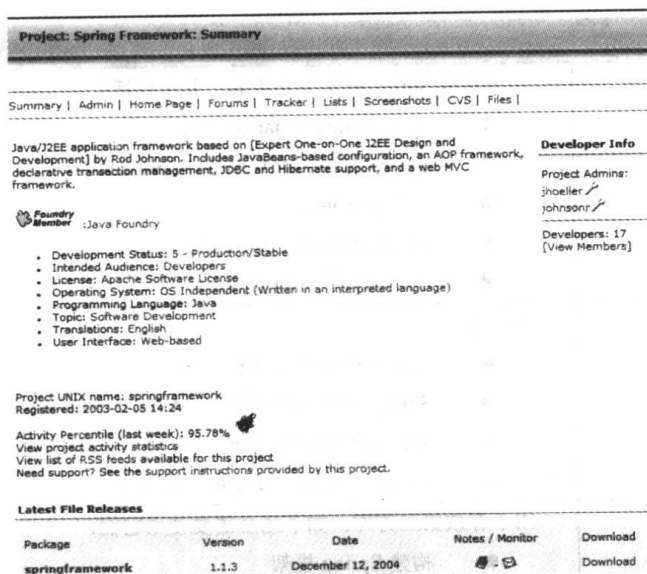
Spring 确信: J2EE 应该易于使用; 针对接口编程, 而不是类; 最大化使用 JavaBean, 以配置应用; OO 设计是最为重要的, 甚至比 J2EE 更重要; 不要过度使用受查异常 (Checked Exception); 利于测试, 无论是单元测试, 还是集成测试。这些内容, Spring 都表现得很优秀, 而且在持续改进。

Spring 同时提供了二进制发布版和相应的源代码。开发者通过 SourceForge 网站 (<http://www.sourceforge.net/projects/springframework/>) 能够获得它们。借助于源代码, 开发者能够更好地理解 Spring 架构, 而且利于应用的调试。当然, 开发者也可以构建 Spring 的自定义版本, 从而能够满足自身的业务需求和架构需求。

本章内容将带领开发者研究 Spring 框架的安装和构建过程。

2.1 获得二进制文件

通过 SourceForge Spring 项目入口, 开发者能够获得 Spring 发布版的持续更新版本。具体网址位于: <http://www.sourceforge.net/projects/springframework>, 如图 2-1 所示。



Project: Spring Framework: Summary

Summary | Admin | Home Page | Forums | Tracker | Lists | Screenshots | CVS | Files |

Java/J2EE application framework based on [Expert One-on-One J2EE Design and Development] by Rod Johnson. Includes JavaBeans-based configuration, an AOP framework, declarative transaction management, JDBC and Hibernate support, and a web MVC framework.

Developer Info

Project Admins:
jhoeller ✓
johnsonr ✓

Developers: 17
[View Members]

Project Details:

- Development Status: 5 - Production/Stable
- Intended Audience: Developers
- License: Apache Software License
- Operating System: OS Independent (Written in an interpreted language)
- Programming Language: Java
- Topic: Software Development
- Translations: English
- User Interface: Web-based

Project UNIX name: springframework
Registered: 2003-02-05 14:24

Activity Percentile (last week): 95.78%
View project activity statistics
View list of RSS feeds available for this project
Need support? See the support instructions provided by this project.

Latest File Releases


Package	Version	Date	Notes / Monitor	Download
springframework	1.1.3	December 12, 2004		Download

图 2-1 SourceForge Spring 项目

Spring 二进制发布版同时提供两种版本。一种是 Spring 二进制发布版本本身，而不提供其依赖的第三方库，比如 `spring-framework-1.1.3.zip`。另一种是包含了其依赖的第三方库，比如 `spring-framework-1.1.3-with-dependencies.zip`。本书建议开发者直接下载第二种发布版。

发布版目录结构

开发者在下载完成 Spring 后，通过解开发布版 zip 文件，能够浏览到如图 2-2 所示目录结构。



图 2-2 带第三方库的 Spring 发布版目录结构

其中的主要目录所包含的内容如表 2-1 所示。

表 2-1 目录内容

目 录	内 容
dist	用于存放 Spring 框架发布内容。其中包括： <ul style="list-style-type: none">● 构成 Spring 的 jar 文件集合● Spring 框架 JSP 标签库 (<code>spring.tld</code>)● <code>spring-beans.dtd</code>● FreeMarker 宏 (<code>spring.ftl</code>)● Velocity 宏 (<code>spring.vm</code>)
docs	用于存放： <ul style="list-style-type: none">● Spring 框架 API 文档 (<code>api</code>)● Spring MVC 框架教程 (<code>MVC-step-by-step</code>)● Spring Reference 文档 (HTML 和 PDF 版本)● Spring 框架标签库文档 (<code>taglib</code>)
lib	用于存放第三方库。如下场合需要使用它们： <ul style="list-style-type: none">● 构建 Spring 框架● 运行实例应用
mock	存放用于 (单元) 测试的 Mock 类

目 录	内 容
samples	用于存放实例应用, 其中包括了用于 Web 应用的模板, 比如 webapp-typical
src	Spring 框架源代码
test	Spring 框架测试代码

开发者可能已经注意到, Spring 框架二进制发布版中也包含了用于构建目的的 Spring 框架源代码。请开发者注意, 这并不完整。这同 2.2 节待介绍的基于 CVS 访问获得 Spring 框架的所有源代码及相关说明文件有所区别, 比如 Spring 框架二进制发布版中并没有包含如何构建 Spring Reference 文档的 readme.txt 文件 (位于 \docs\reference 目录下)。因此, 本书将基于 2.2 节给出的内容来研究 Spring 的构建过程。

2.2 基于源代码构建 Spring

现如今, 大量的 Open Source 项目都是基于 CVS 管理的, 而 Eclipse IDE 对 CVS 的支持相当完美, 因此采用基于 Eclipse IDE 开发 Open Source 项目成了很自然的事情, 比如 JBoss 应用服务器项目、Spring 框架项目。

在开发者下载 Spring 框架源代码后, 可以发现 Spring 框架项目本身就是一个基于 Eclipse 的 Java Project。

2.2.1 基于 CVS 访问以获得源代码

为获得 Spring 框架源代码, 开发者需要完成如下几个步骤。

步骤

(1) 在 Eclipse IDE 中切换到 CVS Repository Exploring 方面 (Perspective), 如图 2-3 所示。

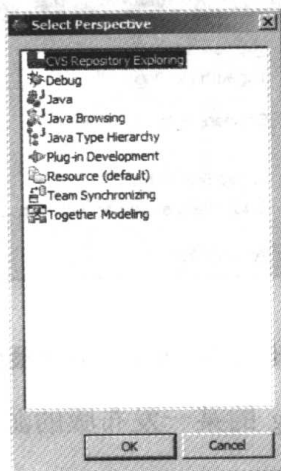


图 2-3 CVS Repository Exploring 方面

(2) 在 CVS Repositories 中新建存储库位置, 并输入图 2-4 中的 Spring CVS 信息。

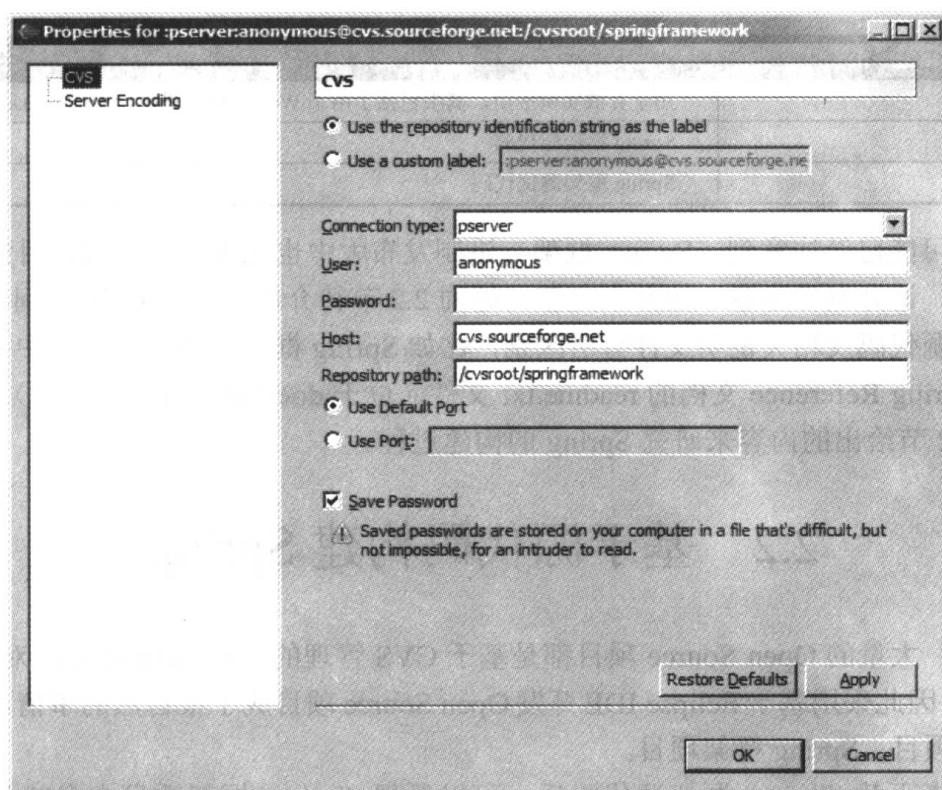


图 2-4 Spring CVS 信息

(3) 将 Spring 框架项目检出 (Check Out), 如图 2-5 所示。请注意, 这是 Spring 框架源代码的最新版。

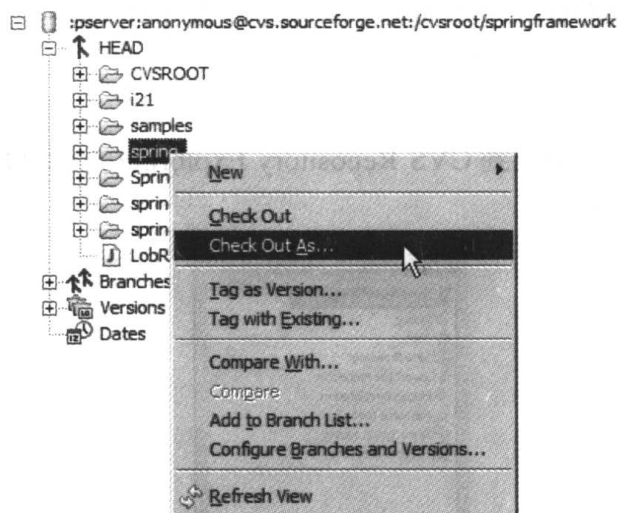


图 2-5 检出 Spring 框架项目 (最新版本)

当然, 开发者如果只是对 Spring 的某一发布版的源代码感兴趣, 则可以通过如图 2-6 所示获得。

然后, 在弹出的对话框中, 单击 “Finish” 按钮, 如图 2-7 所示。

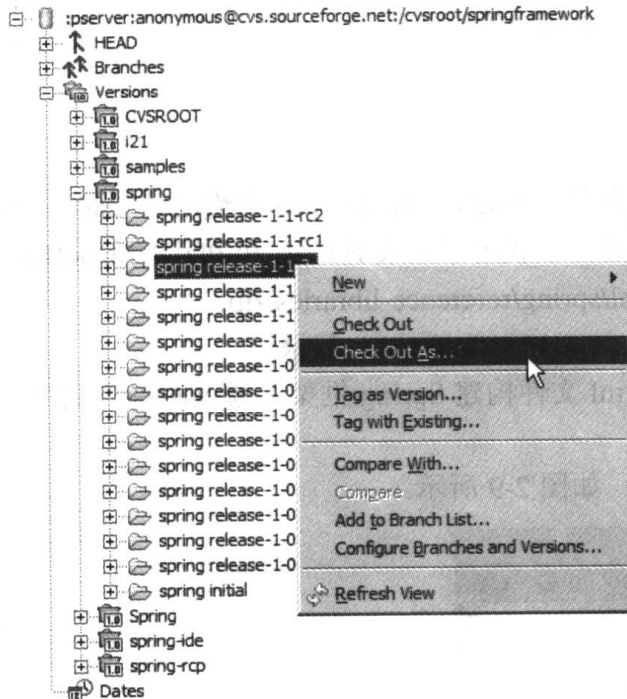


图 2-6 检出 Spring 框架项目（某一版本）

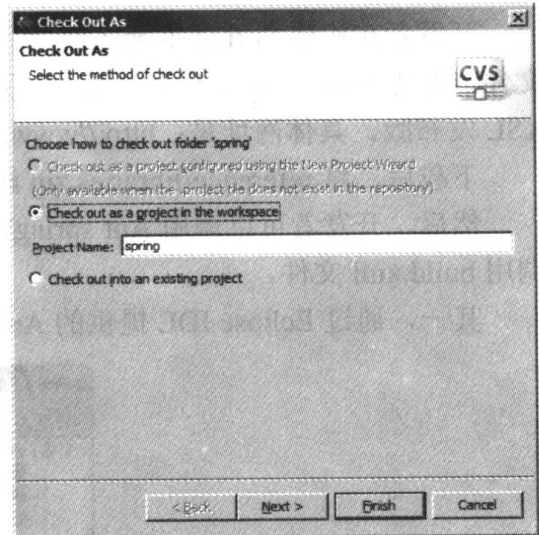


图 2-7 确定 Spring 框架项目的检出方式

(4) 开发者可以在 Java perspective 的 Package Explorer 视图中浏览到 spring project 的组成情况，如图 2-8 所示。

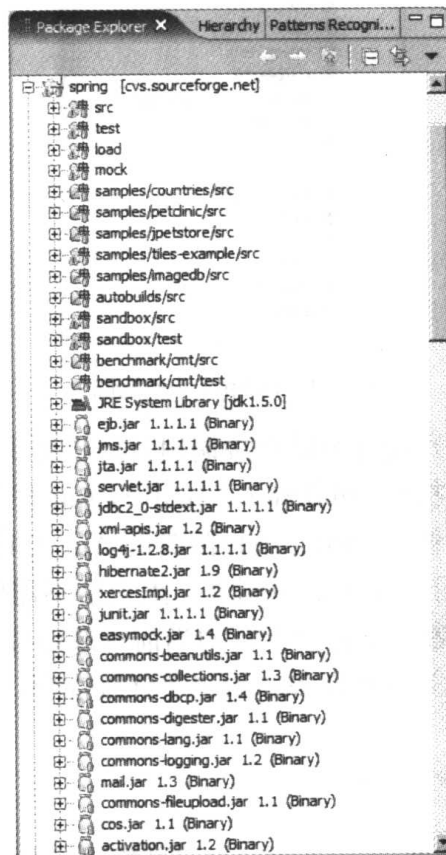


图 2-8 spring project 的组成情况

通过上述过程，开发者已经完成了 Spring 源代码的检出。

2.2.2 构建 Spring 框架

开发者在构建 Spring 框架之前，需要认真阅读 `spring\docs\reference` 中给出的 `readme.txt` 文件。通过 `readme.txt` 文件可以获悉，开发者需要下载用于生成 HTML 和 PDF 的 DocBook XSL 发布版。具体网址是：<http://www.jteam.nl/spring/reference-libraries.zip>。

下载后，将 `reference-libraries.zip` 解压到 `spring\docs\reference` 目录。

然后，开发者可以调用 Ant `spring/build.xml` 文件构建 Spring 框架。通过两种方式可以调用 `build.xml` 文件。

其一，通过 Eclipse IDE 提供的 Ant 视图，如图 2-9 所示。

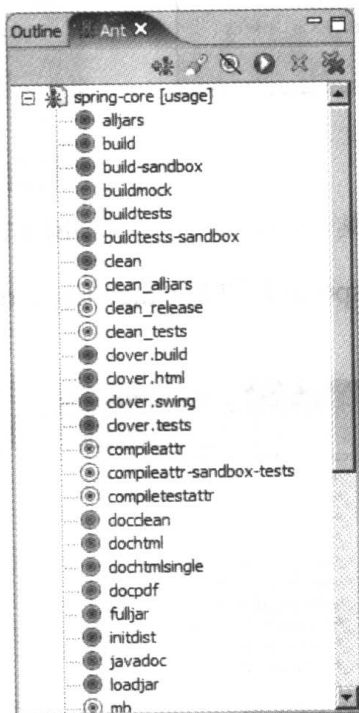


图 2-9 Eclipse Ant 视图

其二，通过命令行方式。开发者可以直接调用目标机器上安装的 Ant 工具（注意，必须是 1.6.2 及其以上版本），以执行 `build.xml` 中给出的 Ant 任务（或目标）。当然，也可以调用 `build.bat` 批处理文件，即使用 `spring` 项目附带的 Ant 工具（位于 `spring\lib` 目录下）。本文构建 Spring 框架，就是基于 `build.bat` 构建的，因为这不需开发者再去安装 Ant 工具。

具体构建过程如下（开发者需要将命令行的当前目录定位到 `d:\workspace\spring`，然后输入 `build release`）。本书在相关 Ant 任务中给出了详细注释，使得开发者能够更好地理解 Spring 的构建过程。

```
D:\workspace\spring>build release
```

```
//设置 JDK，并将 Ant、JUnit、clover.jar 库放置在构建 Spring 发布版的 classpath 中
```

```
D:\workspace\spring>D:\jdk1.5.0\bin/java -cp
```

```
lib/ant/ant.jar;lib/ant/ant-launcher.jar;lib/ant/ant-junit.jar;lib/junit/junit.jar;lib/clover/clover.jar;
D:\jdk1.5.
0/lib/tools.jar org.apache.tools.ant.Main release
Buildfile: build.xml

compileattr:
[attribute-compiler] Generated attribute information for 1 classes. Ignored 0 classes.

//编译源代码

build:
[mkdir] Created dir: D:\workspace\spring\target\classes
[mkdir] Created dir: D:\workspace\spring\target\classes\META-INF
[javac] Compiling 874 source files to D:\workspace\spring\target\classes
[javac] Note: * uses or overrides a deprecated API.
[javac] Note: Recompile with -Xlint:deprecation for details.
[rmic] RMI Compiling 1 class to D:\workspace\spring\target\classes
[copy] Copying 4 files to D:\workspace\spring\target\classes
[copy] Copying 1 file to D:\workspace\spring\target\classes\META-INF

//创建用于存放待构建产出物的目录

initdist:
[mkdir] Created dir: D:\workspace\spring\dist

//根据 Spring 模块划分, 而构建各个 Spring jar 包

modulejars:
[jar] Building jar: D:\workspace\spring\dist\spring-core.jar
[jar] Building jar: D:\workspace\spring\dist\spring-aop.jar
[jar] Building jar: D:\workspace\spring\dist\spring-context.jar
[jar] Building jar: D:\workspace\spring\dist\spring-dao.jar
[jar] Building jar: D:\workspace\spring\dist\spring-orm.jar
[jar] Building jar: D:\workspace\spring\dist\spring-web.jar
[jar] Building jar: D:\workspace\spring\dist\spring-webmvc.jar

//构建单个 Spring jar 包, 即将上述所有 Spring jar 包打包成单个 spring.jar 文件

fulljar:
[jar] Building jar: D:\workspace\spring\dist\spring.jar

//编译 Mock 类

buildmock:
[mkdir] Created dir: D:\workspace\spring\target\mock-classes
```

```
[javac] Compiling 17 source files to
D:\workspace\spring\target\mock-classes
```

```
[javac] Note: * uses or overrides a deprecated API.
[javac] Note: Recompile with -Xlint:deprecation for details.
```

mockjar:

```
[jar] Building jar: D:\workspace\spring\dist\spring-mock.jar
```

//创建 Spring 框架源代码 zip 文件。另外，还拷贝其他 4 个文件到 dist 目录中

srczip:

```
[zip] Building zip: D:\workspace\spring\dist\spring-src.zip
[copy] Copying 1 file to D:\workspace\spring\dist
[copy] Copying 1 file to D:\workspace\spring\dist
[copy] Copying 1 file to D:\workspace\spring\dist
[copy] Copying 1 file to D:\workspace\spring\dist
```

alljars:

//创建 Spring 框架 API 文档

javadoc:

```
[mkdir] Created dir: D:\workspace\spring\docs\api
[javadoc] Generating Javadoc
[javadoc] Javadoc execution
[javadoc] Loading source files for package org.springframework.aop...
[javadoc] Loading source files for package
org.springframework.aop.framework...
[javadoc] Loading source files for package
org.springframework.aop.framework.adapter...
[javadoc] Loading source files for package
org.springframework.aop.framework.autoproxy...
[javadoc] Loading source files for package
org.springframework.aop.framework.autoproxy.metadata...
[javadoc] Loading source files for package
org.springframework.aop.framework.autoproxy.target...
[javadoc] Loading source files for package
org.springframework.aop.interceptor...
[javadoc] Loading source files for package
org.springframework.aop.support...
[javadoc] Loading source files for package org.springframework.aop.target...
[javadoc] Loading source files for package org.springframework.beans...
[javadoc] Loading source files for package
org.springframework.beans.factory.....
```

.....

```
[javadoc] Loading source files for package org.springframework.beans.factory.access...
[javadoc] Loading source files for package org.springframework.beans.factory.config...
[javadoc] Loading source files for package org.springframework.beans.factory.support...
[javadoc] Loading source files for package org.springframework.beans.factory.xml...
[javadoc] Loading source files for package org.springframework.beans.propertyeditors...
[javadoc] Loading source files for package org.springframework.beans.support...
[javadoc] Loading source files for package org.springframework.cache.ehcache...
[javadoc] Loading source files for package org.springframework.context...
```

.....

```
[javadoc] Building index for all the packages and classes...
[javadoc] Building index for all classes...
[javadoc] Generating D:\workspace\spring\docs\api\stylesheet.css...
[javadoc] 30 warnings
```

preparedocs:

//创建HTML文档

dohtml:

```
[java] Writing preface.html for preface(preface)
[java] Writing introduction.html for chapter(introduction)
[java] Writing background.html for chapter(background)
[java] Writing beans.html for chapter(beans)
[java] Writing validation.html for chapter(validation)
[java] Writing aop.html for chapter(aop)
[java] Writing ch06.html for chapter
[java] Writing transaction.html for chapter(transaction)
[java] Writing metadata.html for chapter(metadata)
[java] Writing dao.html for chapter(dao)
[java] Writing jdbc.html for chapter(jdbc)
[java] Writing orm.html for chapter(orm)
[java] Writing mvc.html for chapter(mvc)
[java] Writing view.html for chapter(view)
[java] Writing ch14.html for chapter
[java] Writing ejb.html for chapter(ejb)
[java] Writing remoting.html for chapter(remoting)
[java] Writing mail.html for chapter(mail)
[java] Writing scheduling.html for chapter(scheduling)
[java] Writing springbeansdtd.html for appendix(springbeansdtd)
[java] Writing index.html for book
```

dohtmlsingle:

```
//创建 PDF 文档
```

```
docpdf:
```

```
[java] Making portrait pages on A4 paper (210mmx297mm)
[java] [INFO] Using com.icl.saxon.aelfred.SAXDriver as SAX2 Parser
[java] [INFO] FOP 0.20.5rc
[java] [INFO] Using com.icl.saxon.aelfred.SAXDriver as SAX2 Parser
[java] [INFO] building formatting object tree
[java] [INFO] setting up fonts
[java] [INFO] [1]
[java] [INFO] [2]
```

```
.....
```

```
[java] [INFO] [27]
[java] [INFO] [28]
[java] [INFO] [29]
[java] [INFO] area contents overflows area
```

```
.....
```

```
[java] [INFO] [194]
[java] [INFO] [195]
[java] [INFO] Parsing of document complete, stopping renderer
[delete] Deleting: D:\workspace\spring\docs\reference\pdf\
docbook_fop.tmp
```

```
refdoc:
```

```
//构建 Spring 框架二进制发布版
```

```
release:
```

```
[mkdir] Created dir: D:\workspace\spring\target\release
[zip] Building zip:
D:\workspace\spring\target\release\spring-framework-1.1.3.zip
[zip] Building zip:
D:\workspace\spring\target\release\spring-framework-1.1.3-with-dependenci
es.zip
```

```
//成功构建 Spring
```

```
BUILD SUCCESSFUL
```

```
Total time: 6 minutes 23 seconds
```

通过上述过程，开发者完成了 Spring 框架的构建。请开发者注意，上述给出的构建日志并不是完整的，只是给出了最重要的部分。

2.2.3 重要 Ant 任务

尽管 Spring 框架已经构建完成，但是其中的一些细节还是值得开发者去研究的。比如，通过研究 Ant Spring build.xml 文档，开发者可以获得如下几方面的重要内容。

- Spring 本身是如何构建的
- Spring 构建过程中有哪些地方值得开发者借鉴、学习的
- 如何分发构建后的 Spring

开发者可以带上这些问题，来深入了解 build.xml。本章接下来给出一些重要的 Ant 任务，其中涉及到单元测试、测试代码质量、文档生成。

1. 单元测试

为测试 Spring 框架，开发者可以调用 Ant tests 任务。其中，在执行 tests 任务过程中，Ant 工具将使用到 junit.jar 文件（默认时，使用 JUnit 3.8.1 发布版提供的），即在执行 tests 任务中的 <junit> 任务时。如果开发者是使用自己安装的 Ant 工具，则还需要将 junit.jar 文件拷贝到 Ant 安装目录下的 lib 目录中。

在测试 Spring 框架的同时，如果开发者还需要查看 JUnit 单元测试结果，则可以去 target\test-reports 目录查看。但是，test-reports 提供的是基于 XML 格式的测试结果，这并不利于开发者查看。在此，本章介绍 build.xml 提供的 testsummary 任务，即其中调用了 <junitreport> 任务，以格式化基于 XML 格式的测试结果。图 2-10 给出了 testsummary 任务执行的输出结果（位于 target\test-summary 目录）。

Unit Test Results

Designed for use with JUnit and Ant.

Summary

Tests	Failures	Errors	Success rate	Time
1921	2	2	99.79%	254.182

Note: failures are anticipated and checked for with assertions while errors are unanticipated.

Packages

Name	Tests	Errors	Failures	Time(s)
org.springframework.aop.framework	210	2	2	17.425
org.springframework.aop.framework.adapter	8	0	0	1.943
org.springframework.aop.framework.autoproxy	21	0	0	5.538
org.springframework.aop.framework.autoproxy.metadata	26	0	0	5.098
org.springframework.aop.interceptor	4	0	0	2.233
org.springframework.aop.support	59	0	0	8.363
org.springframework.aop.target	15	0	0	6.910
org.springframework.beans	77	0	0	9.884
org.springframework.beans.factory	55	0	0	4.698
org.springframework.beans.factory.access	12	0	0	3.745
org.springframework.beans.factory.config	40	0	0	6.460
org.springframework.beans.factory.xml	130	0	0	8.342
org.springframework.beans.propertyeditors	13	0	0	0.481
org.springframework.beans.support	2	0	0	0.771

图 2-10 基于 HTML 的 Spring 框架单元测试结果

2. 测试代码质量

在一定程度上，单元测试能够保证代码质量。但是，单元测试的质量如何保证呢？

通常，开发项目都存在大量的单元测试代码。开发者通过执行 JUnit 测试代码后，能够建立起对目标代码的自信心。当单元测试代码很多时，很难保证开发团队中各个开发者的所有单元测试代码的质量，比如待测试的代码覆盖度如何。因此，Spring 框架使用了 Clover¹。Clover（注意，它并不是 Open Source 的，但如果您是 Open Source 开发者，则可以免费申请到用于 Open Source 的 Clover。）能够保证单元测试的质量。在此，本章并不打算给出大量篇幅介绍 Clover，开发者可以参考试用版下载网址，以获得 Clover 的更详尽介绍。

让我们再次回到 Ant build.xml 吧！build.xml 提供了如下几个 Clover 相关的 Ant 任务。

- clover.build
- clover.tests
- clover.swing
- clover.html

开发者从上述名字就能够看出它们的各自功能。本章分别给出在执行 Ant clover.swing（基于 Swing UI）和 clover.html（基于 HTML）任务后的输出结果，如图 2-11 和 2-12 所示。

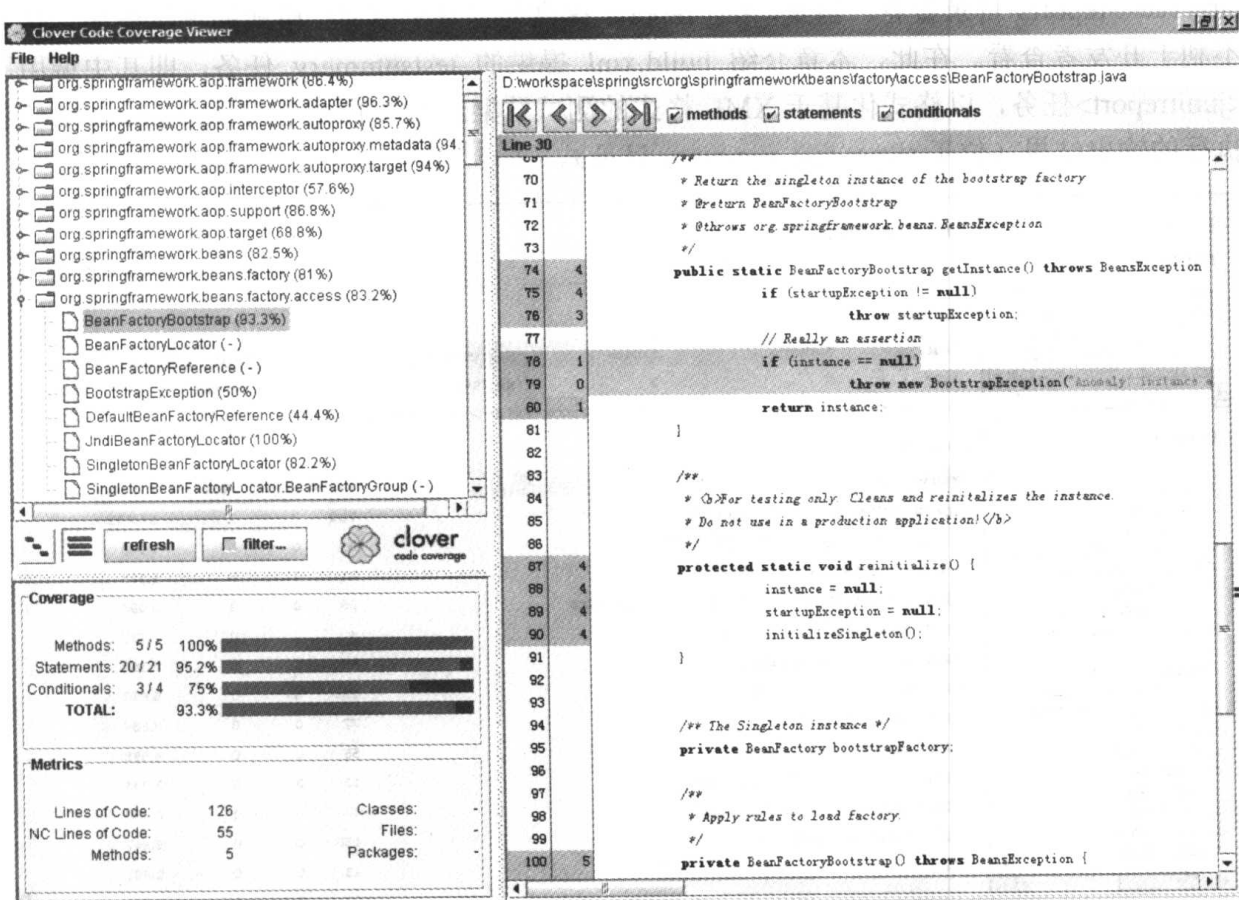


图 2-11 clover.swing 执行结果

¹ 试用版下载网址位于：<http://www.cenqua.com/clover/>。

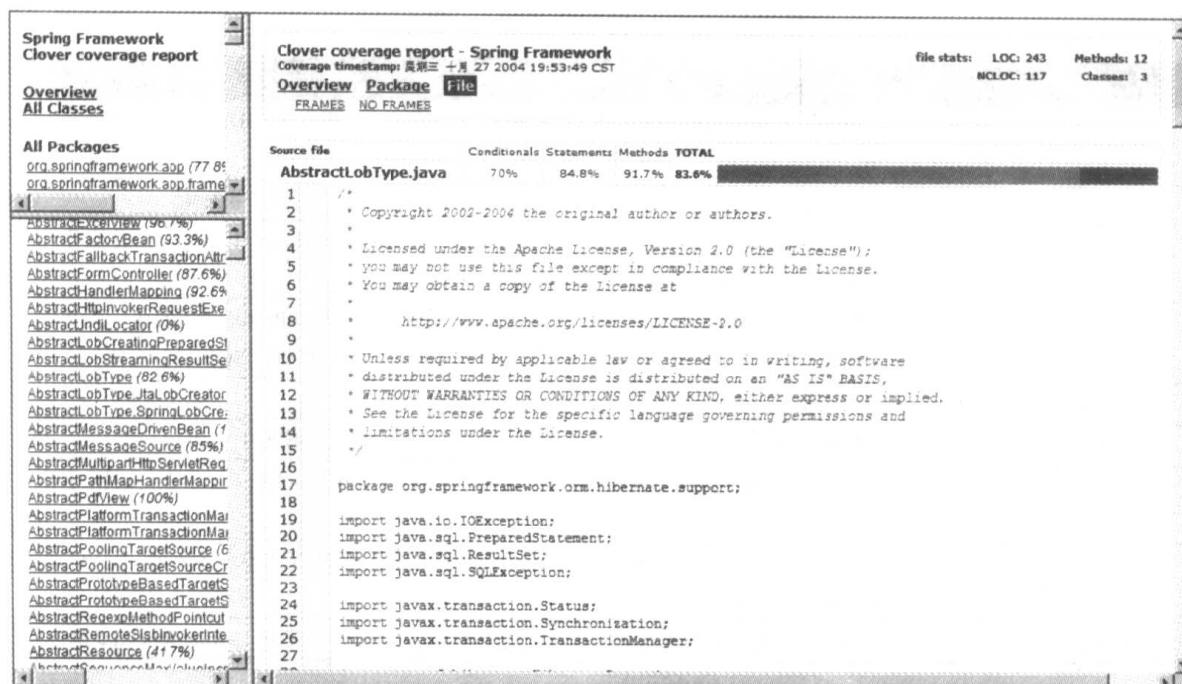


图 2-12 clover.html 执行结果

3. 文档生成

为生成 Spring 框架提供的文档，比如 Spring Reference 文档，开发者可以借助于 Ant build.xml 文件提供的如下 Ant 任务。

- docpdf
- dohtml
- dohtmlsingle

其中，docpdf 用于创建 PDF 文档；dohtml 和 dohtmlsingle 用于创建 HTML 文档，但 dohtmlsingle 将创建的 HTML 文档都保存在单一的 HTML 文件中。

具体内容，请开发者参考 build.xml 文件。

2.3 安装 Spring

开发者在下载 Spring 二进制发布版后（或者在构建 Spring 之后），能够在解压后的二进制发布版目录中看到 dist 目录（对于开发者构建的 Spring 而言，也存在对应的 dist 目录）。其中的内容就是 Spring 框架提供的，即开发应用需要使用 dist 目录包含的 jar 文件集合。对于不同的应用类型，比如 Web 应用和非 Web 应用，所需要的 jar 文件有所区别。主要的 Spring jar 文件解释如表 2-2 所示。

另外，由于 Spring 框架本身使用了第三方类库，因此开发者还需要将 lib 目录下的应用需要的 jar 文件拷贝到目的应用中，比如 Hibernate 库。具体详情，开发者可以参考 readme.txt 文件（位于二进制发布版或源代码发布版的根目录下）。

表 2-2 Spring 框架提供的 jar 文件

jar 文件	内 容
spring-mock.jar	Spring 框架提供的 Mock 类, 用于测试目的。其中, 主要包含 JNDI、Servlet API Mock 类
spring-core.jar	Spring 框架提供的核心类。其中, 主要包含 IoC 容器、核心实用工具
spring-context.jar	Spring 框架提供的上下文类。主要用于应用上下文、有效性、UI 支持、邮件服务、JNDI、JMS、EJB、远程服务、定时服务、缓存等
spring-orm.jar	Spring 框架提供的 O/R Mapping 类, 用于集成各种主流 O/R Mapping 工具。其中, 主要包括 Hibernate 支持、JDO 支持、Apache OJB 支持、iBATIS SQL 映射支持
spring-dao.jar	用于提供 DAO 支持、事务基础框架、JDBC 支持
spring-aop.jar	Spring AOP。其中, 主要用于 AOP 框架、代码级的元数据支持
spring-web.jar	用于提供 Web 应用上下文、多部分解析器 (Multipart Resolver)、Struts 支持、JSF 支持、Web 实用工具
spring-webmvc.jar	Spring 框架提供的 Web MVC 框架。其中, 主要包含 MVC 框架、Servlet、MVC 框架、Web 控制器、Web 视图等内容
spring.jar	Spring 框架提供的所有类, 包含上述所有 jar 文件内容。其中, spring-mock.jar 除外

对于不同的使用情形, 一般都可以根据如下给出的建议来安装 Spring。

- 对于简单的 Java 应用, 只需要将使用到的 jar 文件拷贝到 Java 应用的 classpath 路径中。一般情况下, 这类应用只需要 spring-core.jar 文件, 比如 example4。当然, 如果存在数据库访问逻辑, 则需要依据操作数据库方式的不同, 而拷贝不同的 jar 文件。
- 对于 J2EE 应用, 比如 B/S 应用, 则需要将使用到的 jar 文件拷贝到 Web 容器 (比如, Tomcat 和 Jetty) 或 J2EE 应用服务器 (比如, JBoss、WebSphere 以及 WebLogic) 中。如果需要将 Spring 框架提供的 jar 文件作为 Web 容器和 J2EE 应用服务器的全局库, 即供所有的 Web 应用使用, 则 jar 文件具体拷贝到的目标位置需要依据具体的服务器的情况而定; 如果仅仅需要将 Spring 框架提供的 jar 文件供单个 Web 应用使用, 则可以将 jar 文件拷贝到 Web 应用的 WEB-INF/lib 目录下。
- 由于 spring.jar 文件包含了其他的 jar 文件, 因此在不考虑目标应用打包大小的前提下, 简单地使用 spring.jar 文件最有效, 而且不易出错。

2.4 小 结

本章内容重点给出了 Spring 框架的安装和构建。其中, 还重点给出了若干个 Ant 任务。开发者可以使用它们, 从而为自己的项目服务, 尤其是 Clover 的使用。

第 3 章内容将开始研究 Spring 框架提供的 IoC 容器。

第 3 章 控制反转 (Spring IoC)

多年前, GoF 在《Design Patterns: Elements of Reusable Object-Oriented Software》一书中提出“Programming to an Interface, not an Implementation”, 即“针对接口编程, 而不是实现”。通常情况下, 开发者都会将业务对象抽象成 Java 接口, 然后将各个业务对象共性的内容实现为 Java 接口的抽象类, 并继承于 Java 接口。继而, 再依据具体的业务操作类型, 实现业务对象, 并继承于抽象类。其中的含义很明显: 子类只能够添加或重载操作, 而不能隐藏父类的操作。最终, 实现了抽象类的具体业务实现类便能够响应抽象类继承的接口发送的请求操作。

“针对接口编程, 而不是实现”, 这是开发者普遍达成的共识。GoF 基于“针对接口编程, 而不是实现”, 实现了许多设计模式。当然, 这些具体模式的阐述并不是本书的重点。

在实际应用系统或者产品开发中, 如何实施“针对接口编程, 而不是实现”呢? 比如, 在实际产品部署后, 如果需要更换使用的业务对象, 但又不能够修改源代码。现有的解决办法有很多, 比如 Service Locator 和 IoC 模式。其中, 借助于 IoC 模式实施“针对接口编程, 而不是实现”是本书推荐的办法, 而 Spring IoC 容器正是本章研究的重点。

3.1 IoC 背景知识

IoC 设计模式, 重点关注组件的依赖性、配置以及生命周期。当然, IoC 也适用于简单类, 而不只是组件。除了具有“Dependency Injection¹”(即依赖注入)的昵称外, IoC 还有另一个称呼, 即 Hollywood 原则(“Don’t call me, I’ll call you²”, 即请不要调用我, 我将调用你)。通常, 应用代码需要告知容器或框架, 让它们找到自身所需要的类, 然后再由应用代码创建待使用的对象实例。因此, 应用代码在使用实例之前, 需要创建对象实例。然而, IoC 将创建对象实例的任务交给 IoC 容器或者框架(注, 实现了 IoC 设计模式的框架, 有时候也称之为 IoC 容器), 使得应用代码只需要直接使用实例, 这就是 IoC。

至于 IoC 类型, 本书已经在第 1 章给出过。使用 IoC, 能够提供如下几方面的优势:

- 应用组件不需要在运行时寻找其协作者, 因此更易于开发和编写应用。在许多 IoC 容器中, 借助于设值方法能够在运行时将组件依赖的其他组件注入进来。比如, IoC 容器中组件对 JNDI 的查找工作。
- 由于借助了 IoC 容器管理组件的依赖关系, 使得应用的单元测试和集成测试更利于展开。

¹ 具体解释, 请参考 Robert C. Martin 的“The Dependency Inversion Principle”一文。通过如下网址能够下载到它, <http://www.objectmentor.com/resources/articles/dip.pdf>。

² 参考 Rod Johnson 发表在 TheServerSide.com 网站的“Introducing the Spring Framework”一文。其具体下载网址是 <http://www.theserverside.com/articles/article.tss?l=SpringFramework>。

- 通常，在借助于 IoC 容器管理业务对象的前提下，很少需要使用具体 IoC 容器提供的 API。这使得集成现有的遗留应用成为可能。

因此，通过使用 IoC 能够降低组件之间的耦合度。最终，能够提高类的重用性，更利于测试，而且整个产品或系统更利于集成和配置。

从目前来看，存在许多的 IoC 容器。主要如下：

- Spring
- PicoContainer
- Apache HiveMind

其中，Spring 除了提供 IoC 容器外，还为开发 Java/J2EE 应用提供了其他大量功能，比如 Spring AOP、Spring MVC Framework、集成各种 O/R Mapping 技术、集成各种视图技术、提供 DAO 支持、提供事务管理框架、基于 Spring 构建的 Acegi 安全框架等。这些功能使得在 Open Source 社区和开发者社区，Spring 比其他 IoC 容器更为流行和实用。

3.2 Spring IoC

Spring IoC 容器实现了 IoC 设计模式。本章并不关注 Spring IoC 容器的具体实现细节，而是重点关注如何访问和使用 Spring IoC 容器。

为实现对 Spring IoC 容器的访问，应用代码可以通过如下两个接口完成。

- **BeanFactory**：位于 `org.springframework.beans.factory` 包中。开发者借助于配置文件（比如，XML 或属性文件），能够实现对 JavaBean 的配置和管理。主要用于开发 Java 应用，尤其是在物理资源（内存有限）受限的场合，比如 Applet 应用。
- **ApplicationContext**：位于 `org.springframework.context` 包中。ApplicationContext 构建在 BeanFactory 基础之上，即继承于它。除了具有 BeanFactory 的功能之外，它还添加了其他大量功能，比如同 Spring IoC 集成、（为实现国际化而）处理消息资源、（为应用对象发布和注册通知事件而）添加了事件、声明（非）容器提供的服务等。主要用于开发 J2EE 应用，这也是 Spring 推荐使用的接口。

本章将分两部分深入研究 BeanFactory 和 ApplicationContext 相关内容。

3.2.1 BeanFactory

Spring BeanFactory 是非常轻量的，它处于 Spring 框架的核心。开发者可以将它用于 Applet 应用和单独的 Swing/SWT 应用中，因此开发者需要对 BeanFactory 有足够的了解。当然，对于 EJB 应用而言也同样合适。在整个 Spring 框架中都存在 BeanFactory 的概念，即统一使用 BeanFactory 访问 Spring IoC 容器。开发者通过使用 BeanFactory 能够定义和提供良好的业务对象层，从技术架构的角度考虑，这是很合理的。

当应用创建 BeanFactory 实例时，实际上是完成了 JavaBean 的实例化、配置以及管理，即 BeanFactory 在访问和操作 IoC 容器的初期充当了 IoC 容器的作用。其中，BeanFactory 具体使用的 Spring 配置文件定义了各个 JavaBean 之间的关系。

Spring 提供了 `org.springframework.beans.factory.BeanFactory` 接口的若干实现，比如 `XmlBeanFactory`。图 3-1 给出了 `BeanFactory` 及其实现。

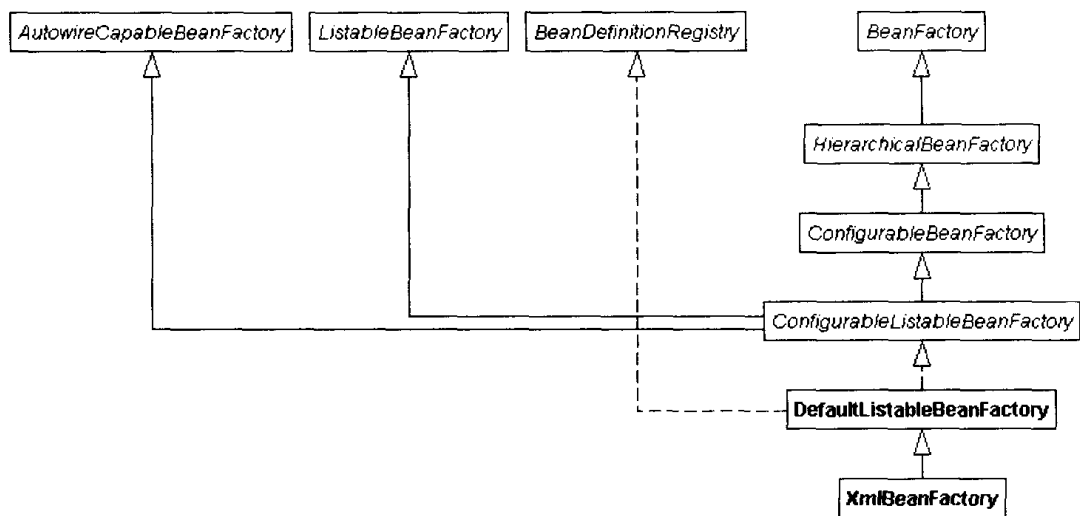


图 3-1 `BeanFactory` 及其实现

对于应用需要直接实例化 `BeanFactory` 实例的场合而言，`XmlBeanFactory` 使用尤为广泛。让我们再次回到 example4 吧。为便于阅读本书，在此再一次给出 `appcontext.xml` 和 `HelloWorldClient` 客户应用。

首先，看看 `appcontext.xml` 文件。

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean name="fileHelloWorld"
        class="com.openv.spring.HelloWorld">
        <constructor-arg>
            <ref bean="fileHello"/>
        </constructor-arg>
    </bean>
    <bean name="fileHello"
        class="com.openv.spring.FileHelloStr">
        <constructor-arg>
            <value>helloworld.properties</value>
        </constructor-arg>
    </bean>
</beans>
  
```

开发者是否注意到其中的粗斜体内容。Spring 配置文件 (`appcontext.xml`) 必须遵循 `spring-beans.dtd` 定义的内容模型。其中，`spring-beans.dtd` 为 JavaBean 命名空间提供了一种简单、一致的定义方式，供 `BeanFactory` 配置使用。大部分 Spring 框架提供的功能都需要使用到它，包括基于 `BeanFactory` 的 Web 应用上下文。本书附录 B 详细给出了 `spring-beans.dtd` 的内容模型。

其次，再次回顾一下 example4 中使用到的各个接口和类，图 3-2 所示给出了 UML 类图。

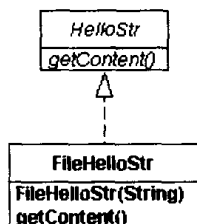


图 3-2 example4 UML 类图

第三，让我们仔细研究 HelloWorldClient.java 源文件（完整的内容请参考第 1 章）。

```
public class HelloWorldClient {
    protected static final Log log = LogFactory.getLog(HelloWorldClient.class);

    public HelloWorldClient(){
        Resource resource = new ClassPathResource("appcontext.xml");
        BeanFactory factory = new XmlBeanFactory(resource);
        HelloWorld hw = (HelloWorld)factory.getBean("fileHelloWorld");
        log.info(hw.getContent());
    }

    public static void main(String[] args) {
        new HelloWorldClient();
    }
}
```

请开发者注意粗体部分的内容。为配置 Spring appcontext.xml 文件，开发者需要告知 XmlBeanFactory 构建器，即 appcontext.xml 文件。当然，上述客户应用使用了 Spring 框架中 org.springframework.core.io 提供的实用类，另外还存在两种方式。

其一，基于文件找到 appcontext.xml。示例代码如下：

```
try {
    InputStream ins = new FileInputStream("src/appcontext.xml");
    BeanFactory factory = new XmlBeanFactory(ins);
    HelloWorld hw = (HelloWorld) factory.getBean("fileHelloWorld");
    log.info(hw.getContent());
} catch (FileNotFoundException ex) {
    log.error("文件 appcontext.xml 未找到", ex);
}
```

其二，基于 ApplicationContext 实现类找到 appcontext.xml 文件。示例代码如下：

```
ClassPathXmlApplicationContext appContext = new
ClassPathXmlApplicationContext(
    new String[] { "appcontext.xml" });
BeanFactory factory = (BeanFactory) appContext;
HelloWorld hw = (HelloWorld) factory.getBean("fileHelloWorld");
log.info(hw.getContent());
```

在多数场景中，应用并不需要显式地给出实例化 BeanFactory 实例的代码，因为 Spring

框架完成了这部分工作,这也体现了 Spring 框架实用性的一面。比如,对于 Web 应用而言,当 J2EE Web 应用部署并启动时, Spring ApplicationContext 将会自动被实例化,本书后续内容将会涉及到相关内容。

为获得 appcontext.xml 中定义的 JavaBean,应用代码只需要调用 BeanFactory 的 getBean 方法,并造型为 JavaBean 所属类型,进而对 JavaBean 对象进行操作。

至此, example4 的详细阐述结束。

接下来,本节将继续展开对 BeanFactory 相关内容的研究。

1. Bean 的生命周期

本节将重点研究一下 BeanFactory 中 Bean 的生命周期。借助于 IoC 容器,即通过 BeanFactory 能够实现对 JavaBean 的控制反转。IoC 容器定义了 Spring 配置文件中 JavaBean 应遵循的规则。Spring 将其称之为 BeanDefinition,即 Bean 定义。大体上看,任何处于 IoC 容器控制下的 JavaBean 的生命周期都存在 4 个阶段。

- 实例化 JavaBean。
- JavaBean 实例的初始化,即通过 IoC 注入其依赖性。这一阶段将完成 JavaBean 实例的初始化。
- 基于 Spring 应用对 JavaBean 实例的使用。
- IoC 容器销毁 JavaBean 实例。

图 3-3 示意了 JavaBean 的整个生命周期。

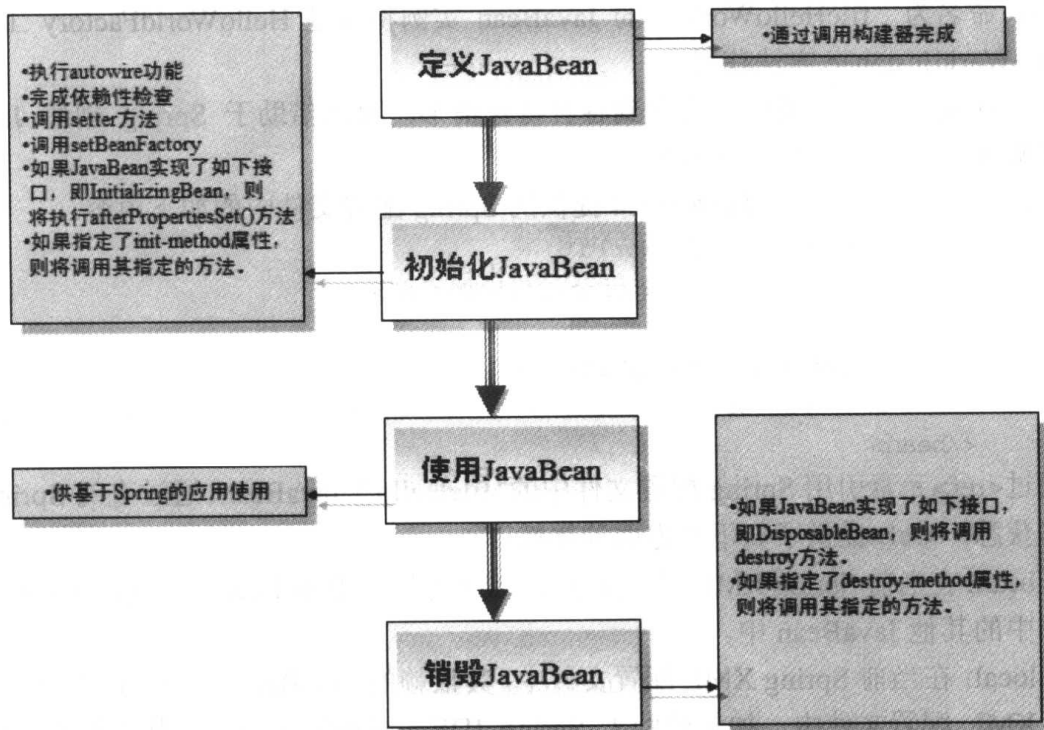


图 3-3 BeanFactory 中 Bean 的生命周期³

³ 参考《Spring Live》一书。

接下来, 本书将围绕上述内容进行讲解。至于 Spring DTD 的其他内容, 请开发者参考本书附录 B 给出的内容, 即 `spring-beans.dtd` 的内容模型。

2. Bean 创建

借助于构建器创建 **JavaBean** 实例 (对应于图 3-3 中的“定义 **JavaBean**”阶段) 最常见, 本书推荐尽量使用无参数构建器创建 **JavaBean** 实例。比如:

```
<bean name="fileHelloWorld"
      class="com.openv.spring.HelloWorld">
```

因此, **HelloWorld** 实例将使用无参数构建器创建出来。

当然, 很多时候, 由于遗留代码原因, 需要借助于工厂方法创建 **JavaBean** 实例。比如:

```
<bean name="fileHelloWorld"
      class="com.openv.spring.HelloWorld"
      factory-method="createHelloWorldInstance">
```

此时, **HelloWorld** 必须提供 `createHelloWorldInstance` 静态方法。

另外, 还有一种情形, 即类并没有提供静态方法的情况。比如:

```
<bean name="helloworldFactory"
      class="com.openv.spring.HelloWorldFactory"/>

<bean name="fileHelloWorld"
      factory-bean="helloworldFactory"
      factory-method="createHelloWorldInstance">
```

其中, 命名为“**fileHelloWorld**”的 **JavaBean** 实例将通过 **HelloWorldFactory** 工厂类的 `createHelloWorldInstance` 方法获得。

当然, 在实际应用环境中, 会遇到各种特殊情形, 灵活借助于 Spring 创建 **JavaBean** 的功能能够使得开发者提高开发效率。

另外, 开发者是否注意到 `example4` 提供的 Spring 配置文件中的如下内容。

```
<bean name="fileHelloWorld"
      class="com.openv.spring.HelloWorld">
  <constructor-arg>
    <ref bean="fileHello"/>
  </constructor-arg>
</bean>
```

即通过 `<ref>` 元素引用 Spring 配置文件中的“**fileHello**” **JavaBean**。通过查阅 Spring DTD 文件能够获悉, `<ref>` 提供了如下几方面的属性。

- **bean**: 在当前 Spring XML 配置文件中, 或者在同一 **BeanFactory** (**ApplicationContext**) 中的其他 **JavaBean** 中。
- **local**: 在当前 Spring XML 配置文件中。其依赖的 **JavaBean** 必须存在于当前 Spring XML 配置文件中。如果借助于 Spring IDE, 则在编译期可对其依赖的 **JavaBean** 进行验证。基于 **local** 方式, 开发者能够使用到 XML 本身提供的优势, 而进行验证。
- **parent**: 用于指定其依赖的父 **JavaBean** 定义。

3. 初始化 JavaBean

通过图 3-3, 开发者能够看出其提供了“初始化 JavaBean”阶段。本节正是阐述这一阶段的。

首先, 如果开发者使用了<bean>元素的 autowire 属性, 则借助于 Spring 提供的 autowire 功能, Spring 能够自动将目标 JavaBean 需要注入的 JavaBean 找到, 并注入进来, 详情请参考附录 B。

其次, 如果开发者指定了<bean>元素的 dependency-check 属性, 则能够保证各个 Spring 配置文件中各个 JavaBean 之间的相互关系, 详情请参考附录 B。

第三, 借助于 setter 方法, 能够将 JavaBean 的属性值注入进来。其中, 这些属性值可以是 Java 原型 (primitive)、对象类型、在 Spring 配置文件中定义的其他 JavaBean。甚至可以是 null。比如:

```
<bean id="example11Service"
```

```
    class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
```

```
    <property name="transactionManager">
```

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

```
    </property>
```

```
    <property name="target">
```

```
        <ref local="example11ServiceTarget"/>
```

```
    </property>
```

```
    <property name="transactionAttributes">
```

```
        <props>
```

```
            <prop key="get*">
```

```
                PROPAGATION_REQUIRED,readOnly
```

```
            </prop>
```

```
            <prop key="set*">
```

```
                PROPAGATION_REQUIRED
```

```
            </prop>
```

```
        </props>
```

```
    </property>
```

```
</bean>
```

其中, Spring 框架能够注入 TransactionProxyFactoryBean 中 transactionManager、target、transactionAttributes 所需的属性值。注意, 这些属性可以通过引用的方式注入进来, 比如 transactionManager 属性取值通过<ref>引用了“transactionManager” POJO 服务。如果 Spring 配置文件中不存在事务管理器, 则开发者有两种做法。其一, 可以单独定义新的 transactionManager。比如:

```
<bean id="transactionManager"
```

```
    class="org.springframework.transaction.jta.JtaTransactionManager">
```

```
    <property name="userTransactionName">
```

```
        <value>java:comp/UserTransaction</value>
```

```
    </property>
```

```
</bean>
```

供<ref>引用。这正是上述配置片断中的做法。另外，定义内部 **JavaBean**。比如，如下给出了示例配置。

```
<property name="transactionManager">
    <bean
        class="org.springframework.transaction.
            jta.JtaTransactionManager">
        <property name="userTransactionName">
            <value>java:comp/UserTransaction</value>
        </property>
    </bean>
</property>
```

通过定义内部 **JavaBean**，其他 **JavaBean** 便不能够引用到它。而且，开发者再也不能够重用 **JtaTransactionManager** 了。当然，现实场合还是存在这种需求的。存在即是合理，请记住这句富有哲理的话。

第四，如果 **JavaBean** 实现了如下接口，则还需要调用 **setBeanFactory** 方法（其含义本书将在后续内容重点阐述）。

```
org.springframework.beans.factory.BeanFactoryAware
```

其定义了如下方法：

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

第五，**Spring** 框架提供了若干接口，供开发者改变配置在 **BeanFactory** 中 **JavaBean** 的行为使用。其中，**InitializingBean** 接口介绍如下：

org.springframework.beans.factory.InitializingBean：在 **BeanFactory** 初始化 **JavaBean** 时，**BeanFactory** 会调用那些实现了 **InitializingBean** 接口的 **JavaBean** 中包含的如下方法：

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

其中，**afterPropertiesSet** 是 **InitializingBean** 定义的方法。

最后，通过在<bean>元素中包含 **init-method** 属性能够达到同 **InitializingBean** 一样的目的，即：

```
<bean name="fileHelloWorld"
    class="com.openv.spring.HelloWorld"
    init-method="init">
```

其中，**HelloWorld** 将实现 **init** 方法。

因此，通过上述 6 个步骤，完成了 **JavaBean** 实例的初始化工作。整个过程称之为“初始化 **JavaBean**”阶段。

4. 使用 **JavaBean**

一旦 **Spring** 创建，并初始化 **JavaBean** 实例后，应用就能够使用 **JavaBean** 实例了。因此，借助于 **getBean** 方法，开发者就能够在应用中使用它了。这样一个过程称之为“使用 **JavaBean**”阶段。

5. 销毁 **JavaBean**

一旦将基于 **Spring** 的（**Web**）应用停止，**Spring** 框架将调用那些 **JavaBean** 实例中存在

的生命周期方法，比如实现了 DisposableBean 接口的 JavaBean，或者那些在 Spring 配置文件中指定了 destroy-method 属性的 JavaBean。最终，Spring 将销毁 JavaBean 实例。请注意，这些内容只适合于那些通过“singleton”方式创建的 JavaBean 实例。对于那些以“prototype”方式创建的 JavaBean 实例，Spring 并不能够控制其生命周期，因为一旦这种 JavaBean 实例创建成功，整个 JavaBean 将交付给 Spring 应用去管理。上述整个过程构成了“销毁 JavaBean”阶段。

其中，DisposableBean 介绍如下。

org.springframework.beans.factory.DisposableBean: 在 BeanFactory 销毁 JavaBean 时，BeanFactory 会调用那些实现了 DisposableBean 接口的 JavaBean 中包含的如下方法。

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

其中，destroy 是 DisposableBean 定义的方法。

当然，通过在<bean>元素中包含 destroy-method 属性能够达到同 DisposableBean 一样的目的，即：

```
<bean name="fileHelloWorld"
      class="com.openv.spring.HelloWorld"
      destroy-method="destroy">
```

其中，HelloWorld 将实现 destroy 方法。

如果应用同时实现了上述两种方式，则 Spring 首先执行 DisposableBean 中的 destroy 方法，然后执行 destroy-method 属性指定的方法。

6. 抽象 Bean 和子 Bean 定义

对于包含大量配置信息的<bean>元素而言，其带来的管理复杂性给开发者也带来了许多不便。比如，有些<bean>元素包括初始方法、静态工厂方法、构建器参数、是否单实例以及属性值等内容。因此，如果能够将这些信息类似于 OO 一样，通过继承解决这种问题，那么将是多么好的事情啊。Spring 通过定义抽象 Bean 和子 Bean 能够简化对<bean>的管理，从而提高开发者的开发效率。

类似于使用 Tapestry 的页面模板一样，通过将公共的内容放置在抽象 Bean 中。注意，子 Bean 能够重用抽象 Bean 中定义的内容，而且还能够“重载”它们。这方面的实例，开发者可以参考 Spring 框架提供的 JPetStore。

7. PropertyPlaceholderConfigurer 和 PropertyOverrideConfigurer

对于 PropertyPlaceholderConfigurer 而言，它能够在 Spring 配置文件外部配置其他应用需要使用到的属性，比如通过 Java 属性文件配置数据库连接信息、LDAP、活动目录连接信息。请参考如下配置 Apache DBCP 数据源的示例：

```
<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>
```

其中, `${jdbc.driverClassName}`、`${jdbc.url}`、`${jdbc.username}`、`${jdbc.password}` 的实际值可以通过 Java 属性文件获得。比如:

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

而 `jdbc.properties` 属性文件的内容示例如下:

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost/example
jdbc.username=sa
jdbc.password=
```

对于 `PropertyOverrideConfigurer` 而言, 功能类似。它们的详细使用, 请开发者参考 Spring 框架的 JavaDoc 文档。

8. BeanFactoryAware 与 BeanNameAware

对于某些基于 Spring 的应用而言, 往往存在这样一种需求, 即将应用的 `BeanFactory` 实例注入到 `JavaBean` 实例中。比如, 为在某 `JavaBean` 实例中动态获得 `BeanFactory` 创建的某单例 `JavaBean`, 但是该单例 `JavaBean` 并没有显式地使用到它。此时, 借助于 `BeanFactoryAware` 能够满足开发者需求。

现有如下示例代码:

```
<bean id="dataSource"
      class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName">
        <value>java:/MySqlDS</value>
    </property>
</bean>

<bean id="example11ServiceTarget"
      class="com.openv.spring.service.impl.Example11ManagerImpl">
    <property name="userinfo">
        <ref local="userinfoDAO"/>
    </property>
</bean>
```

如果在 `example11ServiceTarget` 中需要使用到 `dataSource`, 但在 Spring 配置文件中

example11ServiceTarget 并没有显式地配置对 dataSource 的引用。此时, 需要借助于 BeanFactoryAware, 即 Example11ManagerImpl 需要实现 BeanFactoryAware 接口。对于很多企业级 Java 应用而言, 经常要使用到 BeanFactoryAware 接口。

如果需要在 JavaBean 实例中获悉其配置的名字, 则开发者可以考虑实现 BeanNameAware 接口。其介绍如下。

```
org.springframework.beans.factory.BeanNameAware
```

其定义了如下方法:

```
void setBeanName(String name);
```

注意

其执行时机在调用 InitializingBean afterPropertiesSet 或自定义 init-method 之前, 而在 JavaBean 属性的正常赋值之后。

3.2.2 ApplicationContext

开发者已经知道, BeanFactory 提供了管理和操作 JavaBean 的基本功能, 而且还是通过应用代码显式地实例化 BeanFactory 完成的。从实用性的角度出发, 为加强 BeanFactory 及其实现提供的功能, Spring 框架引入了 ApplicationContext 接口。开发者不需要手工创建 ApplicationContext 实例, 便可以以声明的方式使用它, 比如通过:

```
org.springframework.web.context.ContextLoaderServlet
```

或者

```
org.springframework.web.context.ContextLoaderListener
```

能够在 Web 应用启动的时候自动实例化 ApplicationContext 对象。注意, 对于 Spring BeanFactory 而言, 如果用户没有调用 getBean() 方法, 则使用到的 JavaBean 实例不会被创建。除非客户调用了 getBean() 方法, 则使用到的 JavaBean 实例将会被创建。因此, 在 BeanFactory 中使用了延迟装载的机制, 这主要是同 BeanFactory 的应用场合 (内存或其他资源受限的场合) 有关系。对于 Spring ApplicationContext 而言, 一旦 ContextLoaderServlet 或 ContextLoaderListener 初始化成功, 所有 JavaBean 实例将会被创建 (除非开发者改变了 ApplicationContext 的默认行为, 比如显式设置延迟装载行为)。因此, 希望开发者注意它们之间的这种细微区别。

其中, ContextLoaderServlet 和 ContextLoaderListener 的介绍分别见图 3-4、3-5 (后续内容将由其相关研究)。

其中, 开发者应该注意到 Log4jConfigServlet, 供配置 Spring 应用的日志使用, 其使用方法同 ContextLoaderServlet。比如:

```
<context-param>
  <param-name>log4jConfigLocation</param-name>
  <param-value>/WEB-INF/classes/log4j.properties</param-value>
</context-param>

<servlet>
  <servlet-name>log4jConofigServlet</servlet-name>
```

```
<servlet-class>
    org.springframework.web.util.Log4jConfigServlet
</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
```

另外，本书在研究 Spring Web MVC 时将重点阐述 DispatcherServlet。

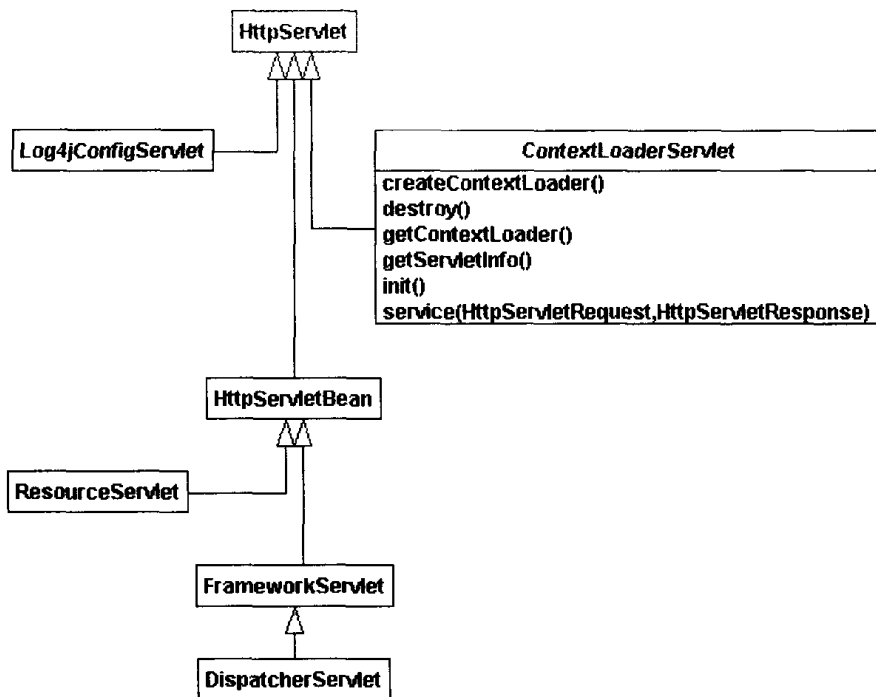


图 3-4 ContextLoaderServlet 类图

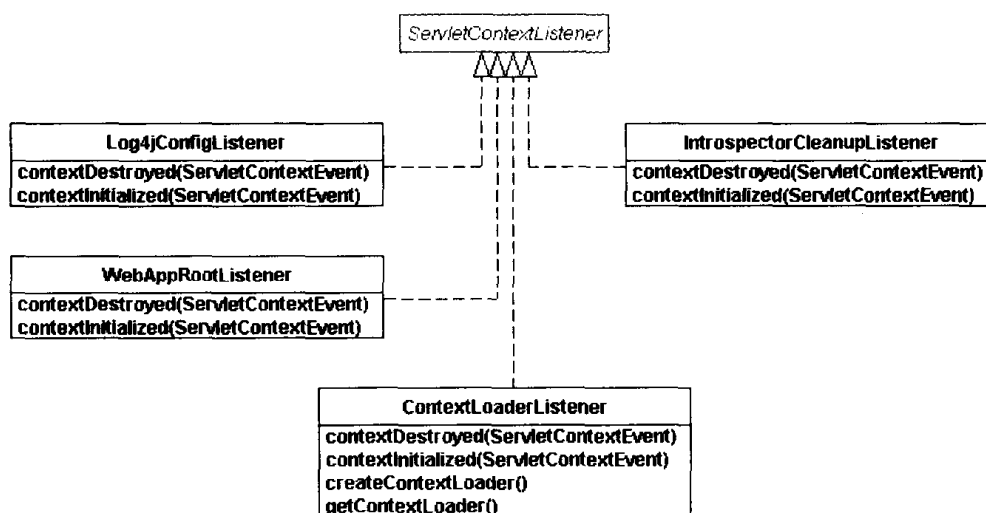


图 3-5 ContextLoaderListener 类图

其中，开发者应该注意到 Log4jConfigListener 类，供配置 Spring 应用的日志使用（配置在 web.xml 中）。比如：

```
<context-param>
    <param-name>log4jConfigLocation</param-name>
```

```

    <param-value>/WEB-INF/classes/log4j.properties</param-value>
</context-param>

<listener>
    <listener-class>
        org.springframework.web.util.Log4jConfigListener
    </listener-class>
</listener>

```

由于 `ApplicationContext` 含有 `BeanFactory` 的所有功能, 因此对于开发 J2EE 应用的场合, 本书推荐使用 `ApplicationContext`。同 `BeanFactory` 类似, 本节接下来给出 `ApplicationContext` 相关内容的研究。

1. Web 应用中创建 `ApplicationContext`

与 `BeanFactory` 相比, `ApplicationContext` 更具优势。因为通过 `ContextLoaderServlet` 或 `ContextLoaderListener` 能够自动创建 `ApplicationContext` 实例。当然, 开发者也可以手工创建 `ApplicationContext` 实例。

对于实现了 `Servlet 2.4` 规范的 Web 容器而言, 可以同时使用 `ContextLoaderServlet` 或 `ContextLoaderListener`。Web 应用启动过程中将自动初始化监听器, 而 `ContextLoaderListener` 就是监听器。示例如下 (摘自 `example11`)。

```

<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">

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

    .....
    <listener>

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

    .....

</web-app>

```

对于实现了 `Servlet 2.2` 规范的 Web 容器而言, 只能够使用 `ContextLoaderServlet`。对于一些实现了 `Servlet 2.3` 规范的 Web 容器而言, 也可以使用 `ContextLoaderListener`。示例如下。

```
.....
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext.xml</param-value>
</context-param>
.....
<servlet>
    <servlet-name>context</servlet-name>
    <servlet-class>
        org.springframework.web.context.ContextLoaderServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
.....
```

2. ApplicationContextAware

对于某些基于 Spring 的应用而言，往往存在这样一种需求，将 Spring 应用的 `ApplicationContext` 实例注入到 `JavaBean` 实例中。比如，为在某 `JavaBean` 实例中动态获得 `ApplicationContext` 创建的某单例 `JavaBean`，但是该单例 `JavaBean` 并没有显式地使用（即在 Spring 配置文件中没有显式地给出对单例 `JavaBean` 的引用，比如未使用 `<ref>` 元素。）到它。此时，借助于 `ApplicationContextAware` 能够满足开发者需求。其介绍如下。

`org.springframework.context. ApplicationContextAware`

其定义了如下方法：

```
void setApplicationContext(ApplicationContext applicationContext)
    throws BeansException;
```

现有如下示例代码：

```
<bean id="dataSource"
    class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName">
        <value>java:/MySqlDS</value>
    </property>
</bean>

<bean id="example11ServiceTarget"
    class="com.openv.spring.service.impl.Example11ManagerImpl">
    <property name="userinfo">
        <ref local="userinfoDAO"/>
    </property>
</bean>
```

如果在 `example11ServiceTarget` 中需要使用到 `dataSource`，但在 Spring XML 配置文件中 `example11ServiceTarget` 并没有显式地配置对 `dataSource` 的引用。此时，开发者需要借助于 `ApplicationContextAware`，即 `Example11ManagerImpl` 需要实现 `ApplicationContextAware` 接口。对于很多企业级 Java 应用而言，开发者经常要使用到 `ApplicationContextAware` 接口。示例代码如下。可以看到，`Example11ManagerImpl` 实现了 `ApplicationContextAware` 接口。

为实现该接口，定义了私有的 `applicationContext` 变量，而且提供了相应的 `setter` 方法。为了获得上述定义的“`dataSource`” `JavaBean` 实例（在 `Spring ApplicationContext` 中全局惟一），借助于 `getBean()` 方法能够获得它（需要提供 `JavaBean` 的 `id` 或者 `name`）。

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;

public class Example11ManagerImpl implements ApplicationContextAware {

    private ApplicationContext applicationContext;

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

    public DataSource getDataSource() {
        DataSource ds = null;
        ds = (DataSource) applicationContext.getBean("dataSource");
        return ds;
    }

}
```

因此，这同 `BeanFactoryAware` 类似。

3. `ApplicationContext` 接口实现

基于 `ApplicationContext` 接口，`Spring` 框架提供了若干实现。开发者也可以遵循 `Spring` 架构原则，而实现自己的 `ApplicationContext` 实现。

其中，开发者需要经常使用到如下 3 个实现。

- `ClassPathXmlApplicationContext`: 在 `Web` 应用中，开发者可以从其 `classpath` 中，即 `WEB-INF/classes` 或 `WEB-INF/lib` 的 `jar` 中装载 `Spring` 配置文件。在单元测试中，开发者经常使用到 `ClassPathXmlApplicationContext`。
- `FileSystemXmlApplicationContext`: 开发者可以从文件系统中装载 `Spring` 配置文件。在单元测试中，开发者经常使用到 `FileSystemXmlApplicationContext`。
- `XmlWebApplicationContext`: 供 `ContextLoaderListener` 或 `ContextLoaderServlet` 内部装载 `Spring` 配置文件使用。

3.3 IoC 其他内容

3.3.1 发布并监听事件

对于 `Spring ApplicationContext (BeanFactory)` 而言，在整个应用运行过程中（包括应用的启动、销毁），会发布各种应用事件。开发者也可以实现自己的事件，从而起到扩展

Spring 框架的作用。

Spring 借助于 `org.springframework.context.event.ApplicationEvent` 抽象类及其子类实现事件的发布；与此同时，借助于 `org.springframework.context.ApplicationListener` 接口及其实现者实现事件的监听。这两者构成了观察者（Observer）模式。

为实现 `ApplicationEvent` 事件的发布，开发者需要借助于 `ApplicationContext`，它提供了 `publishEvent` 方法。比如，Spring 提供了如下三种常见 `ApplicationEvent` 事件实现（见图 3-6）。它们的含义如下。可以看出，为发布 `ApplicationEvent`，需要获得 `ApplicationContext`。

- `org.springframework.web.context.support.RequestHandledEvent`: 开发者必须注意到，在 Spring 的 `WebApplicationContext` 中，一旦客户请求处理完成，将发布 `RequestHandledEvent` 事件。当然，如果需要，开发者也可以在企业应用代码中抛出这种事件。
- `org.springframework.context.event.ContextRefreshedEvent`: 开发者必须注意到，在 Spring `ApplicationContext` 容器初始化完成或刷新时，Spring 框架本身将发布 `ContextRefreshedEvent` 事件。
- `org.springframework.context.event.ContextClosedEvent`: 开发者必须注意到，在关闭 Spring `ApplicationContext` 容器时，Spring 框架本身将发布 `ContextRefreshedEvent` 事件。

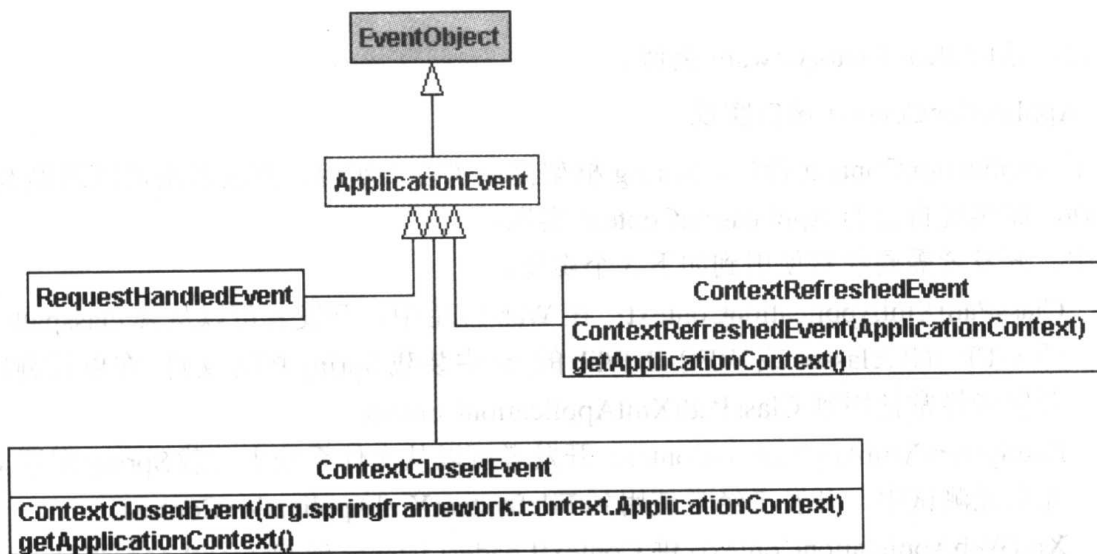


图 3-6 `ApplicationEvent` 及其子类

比如，在关闭 Spring `ApplicationContext` 容器时，Spring 将抛出 `ContextClosedEvent` 事件。示例代码如下（摘自 `org.springframework.context.support.AbstractApplicationContext` 源文件）。

```
public void close() {
    if (logger.isInfoEnabled()) {
        logger.info("Closing application context ["
            + getDisplayName() + "]");
    }
}
```

```

// publish corresponding event
publishEvent(new ContextClosedEvent(this));

// Destroy all cached singletons in this context,
// invoking DisposableBean.destroy and/or "destroy-method".
ConfigurableListableBeanFactory beanFactory = getBeanFactory();
if (beanFactory != null) {
    beanFactory.destroySingletons();
}
}

```

在处理完客户请求时，Spring 容器将抛出 `RequestHandledEvent` 事件。比如，示例代码如下（摘自 `org.springframework.web.servlet.FrameworkServlet` 源文件）。

```

if (isPublishEvents()) {
    // Whether or not we succeeded, publish an event.
    this.webApplicationContext
        .publishEvent(new RequestHandledEvent(this, request
            .getRequestURI(), processingTime, request
            .getRemoteAddr(), request.getMethod(),
            getServletConfig().getServletName(), WebUtils
                .getSessionId(request),
            getUsernameForRequest(request), failureCause));
}

```

为监听 `ApplicationEvent` 事件，开发者需要在目标 `JavaBean` 中实现 `ApplicationListener` 接口。其内容定义如下。

```

public interface ApplicationListener extends EventListener {

    /**
     * Handle an application event.
     * @param event the event to respond to
     */
    void onApplicationEvent(ApplicationEvent event);
}

```

因此，开发者只需要在目标类中实现 `onApplicationEvent` 方法即可。比如，Spring 框架提供的 `org.springframework.web.context.support.PerformanceMonitorListener` 类实现了应用监听器。示例代码如下（摘自 `PerformanceMonitorListener.java`）。

```

public class PerformanceMonitorListener implements ApplicationListener {

    protected final Log logger = LogFactory.getLog(getClass());

    protected final ResponseTimeMonitorImpl responseTimeMonitor =
        new ResponseTimeMonitorImpl();

    public void onApplicationEvent(ApplicationEvent event) {
        if (event instanceof RequestHandledEvent) {

```

```
RequestHandledEvent rhe = (RequestHandledEvent) event;
// could use one monitor per URL
this.responseTimeMonitor.recordResponseTime(rhe
    .getProcessingTimeMillis());
if (logger.isInfoEnabled()) {
    // Stringifying objects is expensive. Don't do it unless it will
    // show.
    logger.info("PerformanceMonitorListener: last=["
        + rhe.getProcessingTimeMillis() + "ms]; "
        + this.responseTimeMonitor + "; client=["
        + rhe.getClientAddress() + "]);
    }
}
}
```

由于 `ApplicationListener` 会监听所有的 Spring 使能应用的 `ApplicationEvent` 事件，因此需要借助于 `instanceof` 过滤不感兴趣的事件。

3.3.2 自定义 JavaBean 属性编辑器

开发者应该还记得，在前面出现过如下的 Spring 配置代码。

```
<property name="transactionAttributes">
    <props>
        <prop key="get *">
            PROPAGATION_REQUIRED,readOnly
        </prop>
        <prop key="set *">
            PROPAGATION_REQUIRED
        </prop>
    </props>
</property>
```

通过查看 `TransactionProxyFactoryBean.java` 代码，可以获悉 `transactionAttributes` 最终需要写入到 `TransactionAttributeSource` 类型的对象中。开发者可能会问，这是怎么回事？难道 Spring 会自动将上述内容转换为 `TransactionAttributeSource` 对象。

其实，这是 JDK 提供的功能。如果开发者对 JavaBean 规范本身很清楚，则对此应该不会有疑问。在 JDK 中，提供了 `java.beans.PropertyEditor` 接口。然后，JDK 又为该接口提供了一抽象类，即 `java.beans.PropertyEditorSupport`。注意，`PropertyEditor` 会自动将字符串转换为其他 Java 类型。比如，上述字符串将会被 `TransactionAttributeEditor` 属性编辑器转换为 `RuleBasedTransactionAttribute` 对象。

Spring 提供了大量的属性编辑器实现。图 3-7 完整地给出了 Spring 中提供的属性编辑器集合。

通过这些属性编辑器的名字便可以了解到它们的功能。比如，`FileEditor` 能够从字符串

取值获得 java.io.File 对象属性。

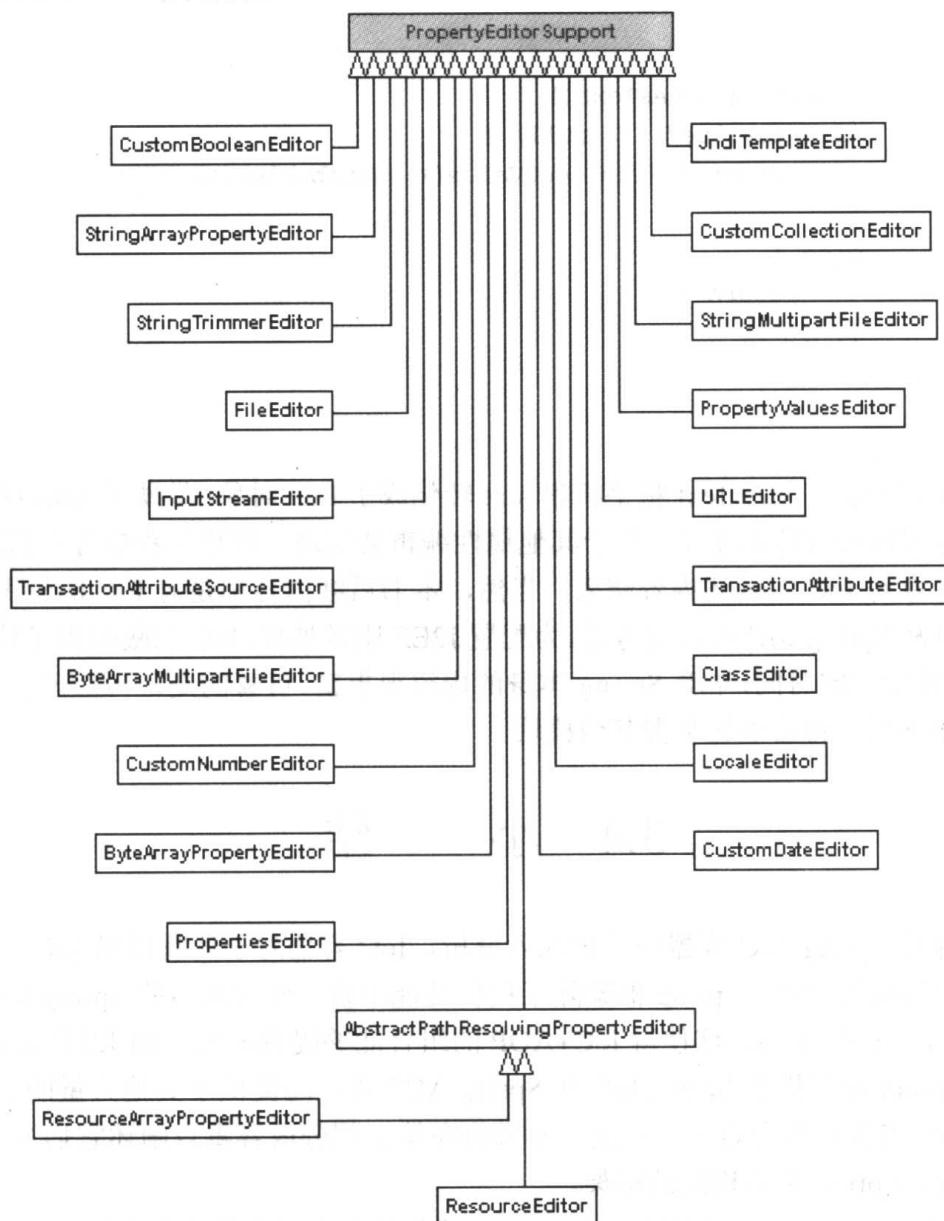


图 3-7 Spring 提供的属性编辑器集合

通常，为实现属性编辑器，开发者只需要重载 PropertyEditorSupport 中的 setAsText() 和 getAsText() 方法。为简单起见，来一起看看 FileEditor 的具体内容吧（下面的内容摘自 org.springframework.beans.propertyeditors.FileEditor.java）！

```

public class FileEditor extends PropertyEditorSupport {

    public void setAsText(String text) throws IllegalArgumentException {
        if (StringUtils.hasText(text)) {
            setValue(new File(text));
        }
        else {
            setValue(null);
        }
    }
}

```

```
    }  
}  
  
public String getAsText() {  
    if (getValue() != null) {  
        return ((File) getValue()).getAbsolutePath();  
    }  
    else {  
        return "";  
    }  
}  
}
```

可以看出,借助于 `setAsText` 将字符串 `text` 转化成了 `File` 对象;借助于 `getAsText` 将 `File` 对象所在的绝对路径打印出来了。至于其他属性编辑器实现,请开发者参考它们的源代码。

至此,本章对 `IoC` 的阐述内容结束。当然,本书后续内容都是对 `IoC` 的使用,因此开发者需要时刻从 `Spring IoC` 的角度考虑。为配置 `J2EE` 服务抽象,`IoC` 到底起到了什么作用?当然,这将会持续一定时间,毕竟 `Spring` 本身的内容很丰富。可喜的是,一旦开发者对 `Spring` 的某个模块熟悉后,将会迅速掌握其他模块。

3.4 小 结

本章内容对 `Spring IoC` 容器作了较深入分析。`IoC` 容器对于所有提供 `IoC` 容器的框架而言都是最为基础的内容。`Spring` 框架除了提供功能丰富、架构灵活的 `Spring IoC` 容器外,还提供了 `Spring AOP` 框架,这使得 `IoC+AOP` 的组合能够提供轻量级的 `J2EE` 架构。

当然, `Spring` 除了提供 `Spring IoC` 和 `Spring AOP` 外,还提供了其他大量的、利于开发者提高开发 `Java/J2EE` 应用效率的功能(本书将在第二部分内容重点阐述它们)。因此,从这个角度出发, `Spring` 是架构级的框架。

关于 `ApplicationContext` 的实例应用,请开发者参考本书其他精彩内容。

好了,让我们进入 `Spring AOP` 吧!

第 4 章 面向方面编程 (Spring AOP)

AOP, 即面向方面编程 (Aspect-Oriented Programming), 是最近在 J2EE 领域 (当然, 也包括 .NET 领域) 最热门的名词之一。面向对象编程 (OOP) 是当今软件开发的主要模式, 这使得架构并实现大型的企业级项目变得更简单。然而, OOP 更多地从系统的垂直切面关注问题, 对于系统的横切面关注甚少, 或者说很难关注。比如, 开发者可以看到, 在系统中到处都是日志、安全性、事务以及其他企业级服务方面的代码, 它们大量地存在于各个实现类中。

AOP 允许开发者动态修改 OOP 定义的静态模型, 即不用修改原始的 OO 模型, 甚至可以不用修改 OO 代码本身, 即可完成对横切面问题的解决。比如, 将系统中处理日志、安全性、事务及其他企业级服务的代码集中放置在一个地方。因此, AOP 使得 OOP 中的重复代码能够大范围减少。

Spring AOP 承诺: 尽管不提供最完整的 AOP 实现, 但是会将具体的 AOP 实现 (包括其他 AOP 实现) 与 Spring IoC 集成在一起, 以解决企业应用中常见的问题。

本章内容将阐述 AOP 基础知识, 并在其基础上深入研究 Spring AOP 提供的具体实现。其中, 将给出大量的实际使用实例。

4.1 AOP 及 Spring AOP 背景知识

通常, 为满足整个企业应用某方面的需求, 开发者 (架构师) 需要整理出系统的关注点。图 4-1 形象地描述了关注点, 它能够从 AOP Aspect 角度看待系统。比如, 持久化、日志、应用的业务逻辑通常被认为是应用需要重点解决的问题。因此, 它们通常作为关注点看待。从整个系统角度考虑, 它往往是由大量的关注点构成的。试想, 在收银系统 (POS) 中, 业务架构师往往需要考虑如下几方面的关注点: 收银管理、用户管理、销售管理、促销管理、外设管理、报表查询等。除了这些业务功能之外, 还需要涉及到系统的便携性、可维护性、对用户进行评审能力的强弱、健壮性。

因此, 大体而言, 所有的关注点可以分为两类: 核心关注点和横切关注点。其中, 核心关注点主要是关注系统的业务逻辑; 横切关注点主要关注系统级的服务, 供业务逻辑使用。在整个业务逻辑中, 到处都将涉及到横切关注点。因此, 对于各个已实现的模块 (业务逻辑) 而言, 都将有大量的横切关注点 (系统级服务, 比如日志、事务、安全性) 实现为它服务。

对于业务逻辑关注点, 使用 OO 技术能够很好地实现。对于不同企业应用而言, 在不同行业, 业务逻辑存在很大不同。因此, 借助于 OO 技术能够很好地实现它。但是, 由于业务逻辑的差异性, 使得开发者不能够提供统一的框架来满足各种各样的业务需求。

对于横切关注点, 可以使用 AOP 技术来实现。因为横切关注点关注的是系统级服务,

这类服务对于大部分应用而言都是最常见的，很容易将它们抽象出来，并加以实现。而本章内容正是关注于横切关注点的。

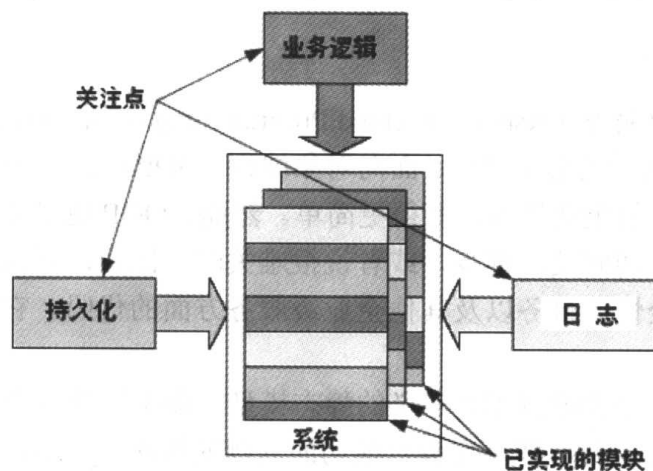


图 4-1 从 AOP Aspect 看待系统¹

面向对象技术将应用系统分解成由层次结构组成的对象，而 AOP 能够将它分解成方面（Aspect）看待。由于两者看待问题的角度不同，使得开发者必须区别对待系统。本书不对 AOP 的术语及基础知识进行过多的介绍。Spring 涉及到的内容很丰富，而且它集成了大量的 Open Source 项目和非 Open Source 项目，因此过多地关注各项目的背景及知识会使得本书负荷太大。而且，本书作者也不想为此而付出过多的篇幅。毕竟，本书是专注于 Spring 的。因此，在整本书的论述中，涉及到各种项目时，都没有细心地对它们进行论述。如果开发者不熟悉相应的内容，则可以查看相关资料。

基于 AOP，业界存在各种各样的 AOP 实现。比如，JBoss AOP、Spring AOP、AspectJ、AspectWerkz 等²。各自实现的功能也不一样。当然，连接点模型（joinpoint）的强弱在很大程度上决定了 AOP 实现功能的强弱。其中，joinpoint 是指 AOP Aspect 能够在应用系统中执行的地方，比如某个方法调用之前、修改某个域之前。本书专注于 Spring AOP。因此，开始进入 Spring AOP 视野吧！

Spring AOP 实现是 Spring 框架的重要组成部分，它实现了 AOP 联盟³约定的接口。当然，如果开发者不需要使用 AOP，而只需要 Spring IoC 容器（即 BeanFactory 和 ApplicationContext，前面章节有介绍），则可以不使用，Spring 框架能够提供这种灵活性。这类类似于 JBoss 中的 JMX MBean 服务实现，即如果某应用不需要 JBoss 提供的、基于 JMX MBean 实现的 MBean 服务，比如邮件服务，则开发者可以删除相应的服务配置文件。

Spring AOP 是由 100% Java 开发完成的，因此它能够秉承 Java 的一切优势，“一次编写，到处运行”。目前，Spring AOP 只实现了方法级的 joinpoint。有些 AOP 实现支持域级的 joinpoint，比如 JBoss AOP、AspectJ。当然，尽管 Spring 也能够用于 J2SE、J2ME 应用

¹ 基于《AspectJ in Action: Practical Aspect-Oriented Programming》一书 Page.8 制作而成。该书由 Manning 出版社于 2003 年出版发行。建议 Spring 开发者看看此书。通过此书，开发者将对 AOP 基础有更深入的认识。

² <http://aopalliance.sourceforge.net/motivations.html> 提供了很多 AOP 实现的链接。

³ <http://aopalliance.sourceforge.net/>

的开发,但开发者主要是在 J2EE 应用中使用它,更何况 Spring Team 也推荐在 J2EE 中使用。在 J2EE 应用中,AOP 能够拦截到方法级的操作已经足够了。因此,实用的 Spring 并没有遗憾。再者,OOP 倡导的是基于 setter/getter 方法访问域,而不是直接访问域,因此 Spring 有足够的理由仅提供方法级的 joinpoint。以后如何,谁也无法预料。为使得 Spring IoC 能够很方便地使用到非常健壮、灵活的企业级服务,则需要借助于 Spring AOP 实现。因为 Spring AOP 能够提供如下几方面的优势。

- 允许开发者使用声明式企业级服务,比如事务服务、安全性服务。EJB 开发者都知道,EJB 组件能够使用 J2EE 容器提供的声明式服务。但是,这些服务需要借助于 EJB 容器,否则 EJB 组件无法使用。而 Spring AOP 却不需要 EJB 容器,即借助于 Spring 的事务抽象框架便能够在 EJB 容器外部使用企业级、声明式服务。
- 开发者可以开发满足业务需求的自定义方面。类似于 JBoss 服务器中的拦截器开发一样,如果标准的 J2EE 安全性不能够满足业务需求,则必须开发拦截器。当然,对于 JBoss 4.0 而言,借助于 JBoss AOP 实现类似功能更简单。
- 开发 Spring AOP Advice 很方便。因为这些 AOP Advice 仅仅是 POJO 类。借助于 Spring 提供的 ProxyFactoryBean,能够快速搭建 Spring AOP Advice 使能应用。

为开发 AOP 使能应用,开发者需要开发 AOP Advice,这也是 AOP 开发中所需要开发的主要内容。因为,Advice 含有 AOP Aspect 的主要逻辑。其中,本书将 AOP 中的 Advice 一词约定为“装备”。在 AOP 中,通常存在如下 5 种装备 (Advice) 类型:

- Before 装备:在执行目标操作之前执行的装备。
 - Throws 装备:如果目标操作在执行过程中抛出了异常,则该装备会执行。由于 Spring AOP 提供了强类型化的 Throws 装备,因此开发者可以采用 Java 捕捉异常的机制来开发应用,而不用对异常信息或 Throwable 进行造型。
 - After 装备:在执行目标操作之后执行的装备。
 - Around 装备:在方法调用前后执行的装备。这种装备的功能最强大,因此它能够在目标操作执行前后实现特定的行为。而且,Around 装备的使用最为灵活。
 - Introduction 装备:由于 Introduction 装备能够为类新增方法,因此在所有 5 种装备中,它最复杂、也最难掌握。考虑到篇幅因素,本书不对这种装备进行阐述。
- 当然,借助于上述 5 种装备,基本上能够解决 J2EE 应用中的常见问题。

因此,本书将重点研究 Spring AOP 装备的内容。由于开发者在开发企业级应用过程中,经常会使用到上述几种装备类型,因此下节内容将重点阐述它们。

4.2 Spring AOP 装备

在开发实际应用过程中,由于各自的业务需求都不同,因此开发者需要灵活选用 Spring AOP 装备。对于 Spring AOP 装备而言,开发者通常会涉及到如下几个接口:

- org.springframework.aop.MethodBeforeAdvice: 用于实现 Before 装备。
- org.springframework.aop.AfterReturningAdvice: 用于实现 After 装备。
- org.springframework.aop.ThrowsAdvice: 用于实现 Throws 装备。

- `org.aopalliance.intercept.MethodInterceptor`: 可供实现 Around 装备使用。它们之间的关系如图 4-2 所示。

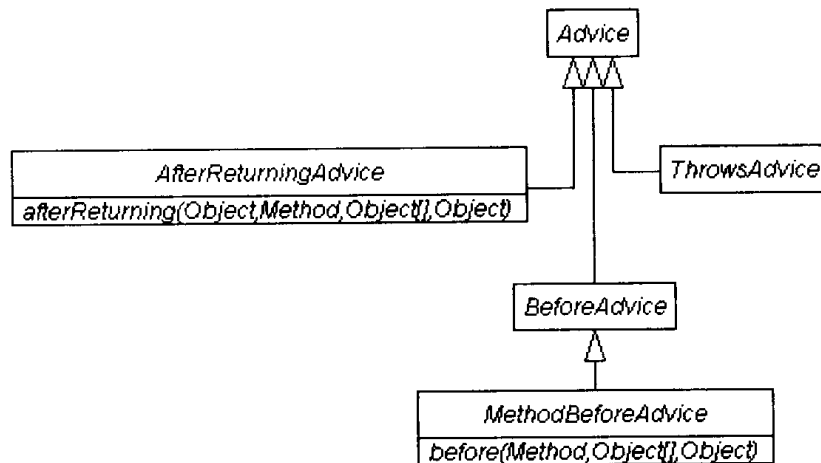


图 4-2 Advice 及相关接口的关系

4.2.1 Before 装备

本书将结合 example5 展开对 Before 装备的研究。为实现 Before 装备，开发者需要实现 `MethodBeforeAdvice` 接口。比如，example5 实现的 `LoggingBeforeAdvice` 装备如下。

```

package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.springframework.aop.MethodBeforeAdvice;

import java.lang.reflect.Method;

/**
 * Before 装备实现
 *
 * @author luoshifei
 */
public class LoggingBeforeAdvice implements MethodBeforeAdvice {
    protected static final Log log =
        LogFactory.getLog(LoggingBeforeAdvice.class);

    public void before(Method arg0, Object[] arg1, Object arg2)
        throws Throwable {
        log.info("before: The Invocation of getContent()");
    }
}
  
```

开发者是否注意到, Before 装备需要实现 before 方法。该方法将在调用目标操作前被调用。这很适用于那些有安全性要求的方法, 即在调用目标操作前检查客户的身份。为将上述 LoggingBeforeAdvice 装备配置在 Spring IoC 容器中, 开发者还需要提供 applicationContext.xml 文件。具体如下 (其中, 包含的 IHelloWorld 及其实现并没有给出, 开发者可以参考 Eclipse 中的 example5 项目)。

```
<?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="helloworldbean"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <value>com.openv.spring.IHelloWorld</value>
        </property>
        <property name="target">
            <ref local="helloworldbeanTarget"/>
        </property>
        <property name="interceptorNames">
            <list>
                <value>loggingBeforeAdvisor</value>
            </list>
        </property>
    </bean>

    <bean id="helloworldbeanTarget"
        class="com.openv.spring.HelloWorld"/>

    <bean id="loggingBeforeAdvisor"
        class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice">
            <ref local="loggingBeforeAdvice"/>
        </property>
        <property name="pattern">
            <value>.*</value>
        </property>
    </bean>

    <bean id="loggingBeforeAdvice"
        class="com.openv.spring.LoggingBeforeAdvice"/>

</beans>
```

其中, 借助于 RegexpMethodPointcutAdvisor 类实现了对 LoggingBeforeAdvice 装备的集成。通过运行 D:\workspace\example5 目录下 Ant build.xml 中的 run 任务, 开发者能够浏览到如下结果。

```
Buildfile: D:\workspace\example5\build.xml
compile:
run:
    [java] 2004-11-4 21:23:07 org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
    [java] 信息: Loading XML bean definitions from class path resource
[appcontext.xml]
    [java] 2004-11-4 21:23:07
org.springframework.beans.factory.support.AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'helloworldbean'
    [java] 2004-11-4 21:23:07 org.springframework.core.CollectionFactory
<clinit>
    [java] 信息: Using JDK 1.4 collections
    [java] 2004-11-4 21:23:07
org.springframework.beans.factory.support.AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean
'helloworldbeanTarget'
    [java] 2004-11-4 21:23:07
org.springframework.beans.factory.support.AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean
'loggingBeforeAdvisor'
    [java] 2004-11-4 21:23:08
org.springframework.beans.factory.support.AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean
'loggingBeforeAdvice'
    [java] 2004-11-4 21:23:08
org.springframework.aop.support.RegexpMethodPointcutAdvisor createPointcut
    [java] 信息: Using Perl5RegexpMethodPointcut for
RegexpMethodPointcutAdvisor
    [java] 2004-11-4 21:23:08 com.openv.spring.LoggingBeforeAdvice before
    [java] 信息: before: The Invocation of getContent()
    [java] 2004-11-4 21:23:08 com.openv.spring.HelloWorld getContent
    [java] 信息: luoshifei
    [java] 2004-11-4 21:23:08 com.openv.spring.HelloClient main
    [java] 信息: luoshifei
BUILD SUCCESSFUL
Total time: 1 second
```

开发者能够很容易看到，在执行目标方法 `getContent` 前，执行了 `LoggingBeforeAdvice` 中的 `before` 方法。

其中，使用了 `RegexpMethodPointcutAdvisor` 类，以完成 `pointcut` 和拦截器的定义。当然，开发者也可以使用 `org.springframework.aop.support.JdkRegexpMethodPointcut`，或者类 `Perl5RegexpMethodPointcut` 来单独定义 `pointcut`，而拦截器的定义另外给出，如图 4-3 所示。

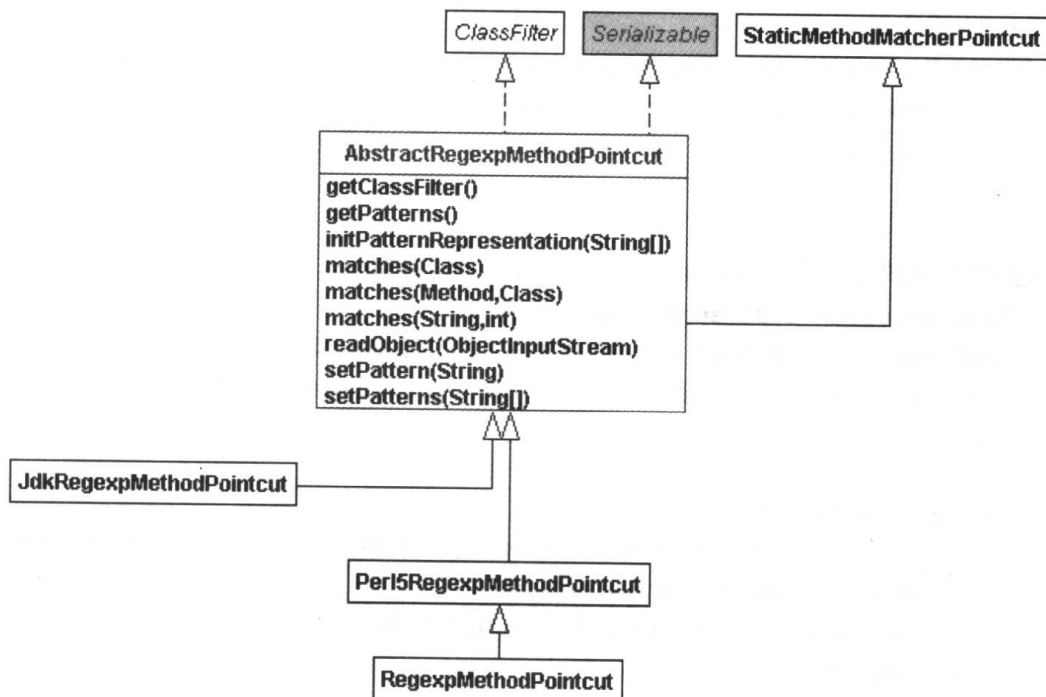


图 4-3 JdkRegexpMethodPointcut

更详细的内容，请开发者参考 `org.springframework.aop.support` 包。

4.2.2 After 装备

当然，After 装备同 Before 装备很类似。其中，After 装备在执行目标操作后执行装备中的 `afterReturning` 方法，而 Before 装备在执行目标操作前执行装备中的 `before` 方法。具体的 `LoggingAfterAdvice` 实现（见 example6）如下。

```

package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.springframework.aop.AfterReturningAdvice;

import java.lang.reflect.Method;

/**
 * After 装备实现
 *
 * @author luoshifei
 */
public class LoggingAfterAdvice implements AfterReturningAdvice {
    protected static final Log log =
        LogFactory.getLog(LoggingAfterAdvice.class);

```

```
        public void afterReturning(Object object, Method m, Object[] args,
            Object target) throws Throwable {
            log.info("after: The Invocation of getContent()");
        }
    }
}
```

Spring 配置文件同上述 (example5) 给出的, 具体如下。

```
<?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="helloworldbean"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <value>com.openv.spring.IHelloWorld</value>
        </property>
        <property name="target">
            <ref local="helloworldbeanTarget"/>
        </property>
        <property name="interceptorNames">
            <list>
                <value>loggingAfterAdvisor</value>
            </list>
        </property>
    </bean>

    <bean id="helloworldbeanTarget"
        class="com.openv.spring.HelloWorld"/>

    <bean id="loggingAfterAdvisor"
        class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice">
            <ref local="loggingAfterAdvice"/>
        </property>
        <property name="pattern">
            <value>.*</value>
        </property>
    </bean>

    <bean id="loggingAfterAdvice"
        class="com.openv.spring.LoggingAfterAdvice"/>

</beans>
```

通过执行 Ant build.xml run 任务, 能够浏览到具体执行结果。

Buildfile: D:\workspace\example6\build.xml

```
compile:
run:
    [java] 2004-12-12 20:07:50
org.springframework.beans.factory.xml.XmlBeanDefinitionReader
loadBeanDefinitions
    [java] 信息: Loading XML bean definitions from class path resource
[appcontext.xml]
    [java] 2004-12-12 20:07:50
org.springframework.beans.factory.support.AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'helloworldbean'
    [java] 2004-12-12 20:07:50 org.springframework.core.CollectionFactory
<clinit>
    [java] 信息: Using JDK 1.4 collections
    [java] 2004-12-12 20:07:50
org.springframework.beans.factory.support.AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean
'helloworldbeanTarget'
    [java] 2004-12-12 20:07:50
org.springframework.beans.factory.support.AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean
'loggingAfterAdvisor'
    [java] 2004-12-12 20:07:50
org.springframework.beans.factory.support.AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'loggingAfterAdvice'
    [java] 2004-12-12 20:07:50
org.springframework.aop.support.RegexpMethodPointcutAdvisor createPointcut
    [java] 信息: Using Perl5RegexpMethodPointcut for
RegexpMethodPointcutAdvisor
    [java] 2004-12-12 20:07:50 com.openv.spring.HelloWorld getContent
    [java] 信息: luoshifei
    [java] 2004-12-12 20:07:50 com.openv.spring.LoggingAfterAdvice
afterReturning
    [java] 信息: after: The Invocation of getContent()
    [java] 2004-12-12 20:07:50 com.openv.spring.HelloClient main
    [java] 信息: luoshifei
BUILD SUCCESSFUL
Total time: 2 seconds
```

有一点值得开发者注意, 即 **Spring AOP** 是以 100% **Java** 实现的, 而且开发者不需要对配置了装备的任何类进行编译工作。其中, 由于 **Spring** 并没有采用控制类装载器的方式来实现 **AOP**, 因此 **Spring AOP** 能够适用于任何 **Web** 容器和应用服务器中, 比如 **Tomcat** 和 **JBoss**。而且, 对于代理 **Java** 接口的场景, **Spring** 默认时是采用动态代理实现的。在很大程度上, **J2SE 1.4** 改进了动态代理的性能。对于代理 **Java** 类的场景, **Spring** 使用动态字节码 (byte-code) 生成技术, 比如使用了 **CGLIB⁴** 库 (Code Generation Library)。

⁴ <http://cglib.sourceforge.net>

其中，CGLIB 库的使用很广泛，比如 Hibernate、JBoss。

4.2.3 Throws 装备

Throws 装备对于处理事务，或者说特定的业务需求很有帮助。为实现 Throws 装备开发者需要实现 `org.springframework.aop.ThrowsAdvice` 接口。为模拟异常的抛出，开发者需要修改 `IHelloWorld` 接口和 `HelloWorld` 类（见 example7）。

`IHelloWorld` 接口如下。

```
package com.openv.spring;

/**
 * IHelloWorld 接口
 *
 * @author luoshifei
 */
public interface IHelloWorld {
    public String getContent(String helloworld) throws Exception;
}
```

`HelloWorld` 类如下。

```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * IHelloWorld 接口实现
 *
 * @author luoshifei
 */
public class HelloWorld implements IHelloWorld {
    protected static final Log log = LogFactory.getLog(HelloWorld.class);

    public String getContent(String helloworld) throws Exception {
        log.info(helloworld);
        throw new Exception();
    }
}
```

`LoggingThrowsAdvice` 装备代码如下。

```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.springframework.aop.ThrowsAdvice;
```



```
import java.lang.reflect.Method;

/**
 * Throws 装备实现
 *
 * @author luoshifei
 */
public class LoggingThrowsAdvice implements ThrowsAdvice {
    protected static final Log log =
        LoggerFactory.getLog(LoggingThrowsAdvice.class);

    public void afterThrowing(Method method, Object[] args, Object target,
        Throwable subclass) {
        log.info("应用抛出了异常");
    }
}
```

开发者应该注意到，其实现了 `afterThrowing` 方法。当异常抛出时，该装备即被激活。具体的 Spring 配置文件如下（即，`appcontext.xml`）。

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

    <bean id="helloworldbean"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <value>com.openv.spring.IHelloWorld</value>
        </property>
        <property name="target">
            <ref local="helloworldbeanTarget"/>
        </property>
        <property name="interceptorNames">
            <list>
                <value>loggingThrowsAdvisor</value>
            </list>
        </property>
    </bean>

    <bean id="helloworldbeanTarget"
        class="com.openv.spring.HelloWorld"/>

    <bean id="loggingThrowsAdvisor"
        class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice">
            <ref local="loggingThrowsAdvice"/>
        </property>
```

```
<property name="pattern">
    <value>.*</value>
</property>
</bean>

<bean id="loggingThrowsAdvice"
    class="com.openv.spring.LoggingThrowsAdvice"/>

</beans>
```

通过运行 Ant build.xml 中的 run 任务，开发者能够看到如下类似结果。

```
Buildfile: D:\workspace\example7\build.xml
compile:
run:
    [java] 2004-12-12 20:28:03 org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
    [java] 信息: Loading XML bean definitions from class path resource
[appcontext.xml]
    [java] 2004-12-12 20:28:03 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'helloworldbean'
    [java] 2004-12-12 20:28:03 org.springframework.core.CollectionFactory
<clinit>
    [java] 信息: Using JDK 1.4 collections
    [java] 2004-12-12 20:28:04 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean
'helloworldbeanTarget'
    [java] 2004-12-12 20:28:04 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean
'loggingThrowsAdvisor'
    [java] 2004-12-12 20:28:04
org.springframework.beans.factory.support.AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean
'loggingThrowsAdvice'
    [java] 2004-12-12 20:28:04 org.springframework.aop.support.
RegexMethodPointcutAdvisor createPointcut
    [java] 信息: Using Perl5RegexMethodPointcut for
RegexMethodPointcutAdvisor
    [java] 2004-12-12 20:28:04 org.springframework.aop.framework.adapter.
ThrowsAdviceInterceptor <init>
    [java] 信息: Found exception handler method [public void
com.openv.spring.LoggingThrowsAdvice.afterThrowing(java.lang.reflect.Meth
od,java.lang.Object[],java.lang.Object,java.lang.Throwable)]
    [java] 2004-12-12 20:28:04 com.openv.spring.HelloWorld getContent
    [java] 信息: luoshifei
```

```
[java] 2004-12-12 20:28:04 org.springframework.aop.framework.adapter.
ThrowsAdviceInterceptor getExceptionHandler
[java] 信息: Trying to find handler for exception of class java.lang.Exception
[java] 2004-12-12 20:28:04 org.springframework.aop.framework.adapter.
ThrowsAdviceInterceptor getExceptionHandler
[java] 信息: Looking at superclass class java.lang.Exception
[java] 2004-12-12 20:28:04 com.openv.spring.LoggingThrowsAdvice
afterThrowing
[java] 信息: 应用抛出了异常
BUILD SUCCESSFUL
Total time: 2 seconds
```

因此, 在客户应用抛出 `Exception` 异常后, `LoggingThrowsAdvice` 的 `afterThrowing` 即被激活。

4.2.4 Around 装备

Around 装备功能很强大, 灵活性也最好。它能够在执行目标操作前后执行, 因此这对于一些需要做资源初始化和释放操作的应用而言, 特别有用。对于 `example8` 而言, 它实现了 `LoggingAroundAdvice` 装备。具体代码如下。

```
package com.openv.spring;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * Around 装备实现
 *
 * @author luoshifei
 */
public class LoggingAroundAdvice implements MethodInterceptor {
    protected static final Log log =
        LogFactory.getLog(LoggingAroundAdvice.class);

    public Object invoke(MethodInvocation invocation) throws Throwable {
        log.info("before: The Invocation of getContent()");
        invocation.getArguments()[0] = "luoshifei";
        invocation.proceed();
        log.info("after: The Invocation of getContent()");

        return null;
    }
}
```

```
}
```

对应的 applicationContext.xml 如下。

```
<?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="helloworldbean"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <value>com.openv.spring.IHelloWorld</value>
        </property>
        <property name="target">
            <ref local="helloworldbeanTarget"/>
        </property>
        <property name="interceptorNames">
            <list>
                <value>loggingAroundAdvisor</value>
            </list>
        </property>
    </bean>

    <bean id="helloworldbeanTarget"
        class="com.openv.spring.HelloWorld"/>

    <bean id="loggingAroundAdvisor"
        class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
        <property name="advice">
            <ref local="loggingAroundAdvice"/>
        </property>
        <property name="pattern">
            <value>.*</value>
        </property>
    </bean>

    <bean id="loggingAroundAdvice"
        class="com.openv.spring.LoggingAroundAdvice"/>

</beans>
```

客户应用 (HelloClient.java) 很简单。

```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

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

```
import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

/**
 * HelloWorld 客户应用
 *
 * @author luoshifei
 */
public class HelloClient {
    protected static final Log log = LogFactory.getLog(HelloClient.class);

    public static void main(String[] args) {
        Resource resource = new ClassPathResource("appcontext.xml");
        BeanFactory factory = new XmlBeanFactory(resource);
        IHelloWorld hw = (IHelloWorld) factory.getBean("helloworldbean");
        log.info(hw.getContent("luoshifei"));
    }
}
```

最终的执行结果为 (Ant build.xml):

```
Buildfile: D:\workspace\example8\build.xml
compile:
run:
    [java] 2004-12-12 20:29:33
org.springframework.beans.factory.xml.XmlBeanDefinitionReader
loadBeanDefinitions
    [java] 信息: Loading XML bean definitions from class path resource
[appcontext.xml]
    [java] 2004-12-12 20:29:33
org.springframework.beans.factory.support.AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'helloworldbean'
    [java] 2004-12-12 20:29:33 org.springframework.core.CollectionFactory
<clinit>
    [java] 信息: Using JDK 1.4 collections
    [java] 2004-12-12 20:29:33
org.springframework.beans.factory.support.AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean
'helloworldbeanTarget'
    [java] 2004-12-12 20:29:33
org.springframework.beans.factory.support.AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean
'loggingAroundAdvisor'
    [java] 2004-12-12 20:29:33
org.springframework.beans.factory.support.AbstractBeanFactory getBean
```

```
[java] 信息: Creating shared instance of singleton bean
'loggingAroundAdvice'
[java] 2004-12-12 20:29:33
org.springframework.aop.support.RegexpMethodPointcutAdvisor createPointcut
[java] 信息: Using Perl5RegexpMethodPointcut for
RegexpMethodPointcutAdvisor
[java] 2004-12-12 20:29:33 com.openv.spring.LoggingAroundAdvice invoke
[java] 信息: before: The Invocation of getContent()
[java] 2004-12-12 20:29:33 com.openv.spring.HelloWorld getContent
[java] 信息: luoshifei
[java] 2004-12-12 20:29:33 com.openv.spring.LoggingAroundAdvice invoke
[java] 信息: after: The Invocation of getContent()
[java] 2004-12-12 20:29:33 com.openv.spring.HelloClient main
[java] 信息: null
BUILD SUCCESSFUL
Total time: 1 second
```

当然, Spring AOP 中的 `org.springframework.aop.interceptor` 包也提供了大量的 Around 装备实现, 见图 4-4 所示。

比如, `TraceInterceptor` 能够用于跟踪 Java 方法的执行情况等。它们的具体功能, 开发者可以参考 Spring API JavaDoc 文档。当然, 它们的使用同本书上述其他实例类似, 在此不再展开讨论。

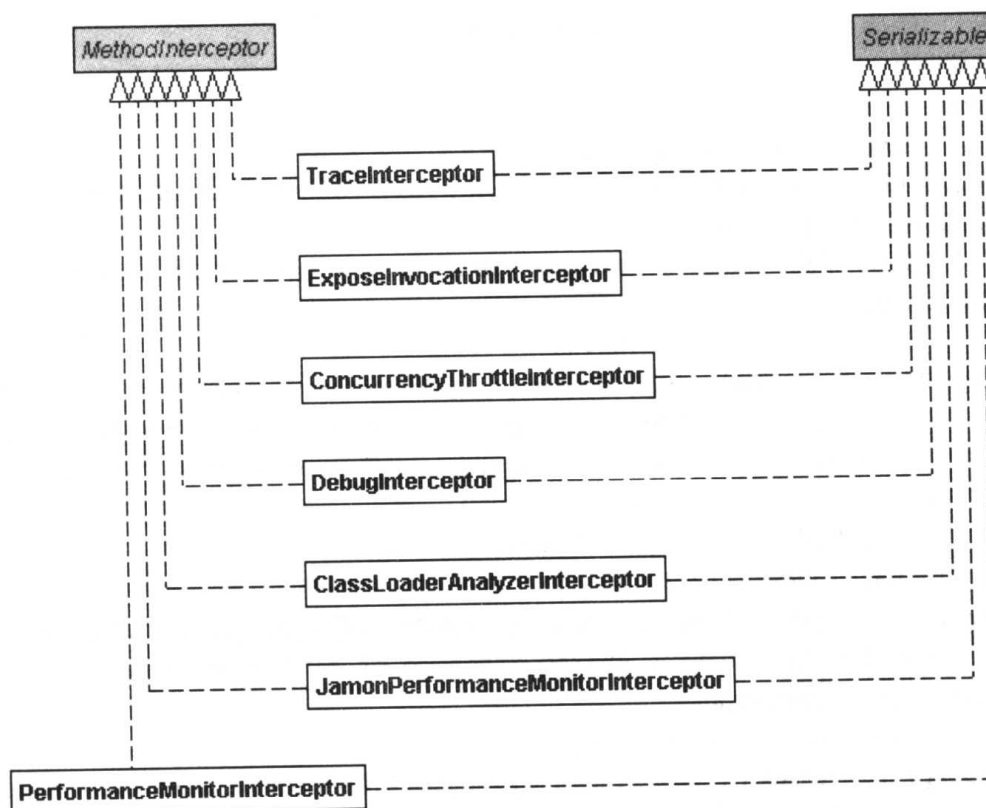


图 4-4 aop.interceptor 包提供的 Around 装备实现

4.3 ProxyFactoryBean

开发者通过研究上述4个实例后,对Spring AOP应该有了一些认识。其中,直接通过Spring AOP实现各种装备类型的灵活程度最高。在上述4个实例中,ProxyFactoryBean起了很重要的作用(位于appcontext.xml中)。

同其他Spring FactoryBean实现一样,ProxyFactoryBean引入了间接层。请注意,通过名字或者id(比如,“helloworldbean”)获得的引用对象并不是ProxyFactoryBean实例本身,而是ProxyFactoryBean中getObject()方法实现返回的对象。其中,getObject方法将创建AOP代理,并将目标对象包裹(wrapper)在其中。那么,开发者可能会问,ProxyFactoryBean到底是什么呢?

ProxyFactoryBean实现了org.springframework.beans.factory.FactoryBean接口(后续内容将具体研究FactoryBean),其本身也是JavaBean,图4-5给出了相应的类图。

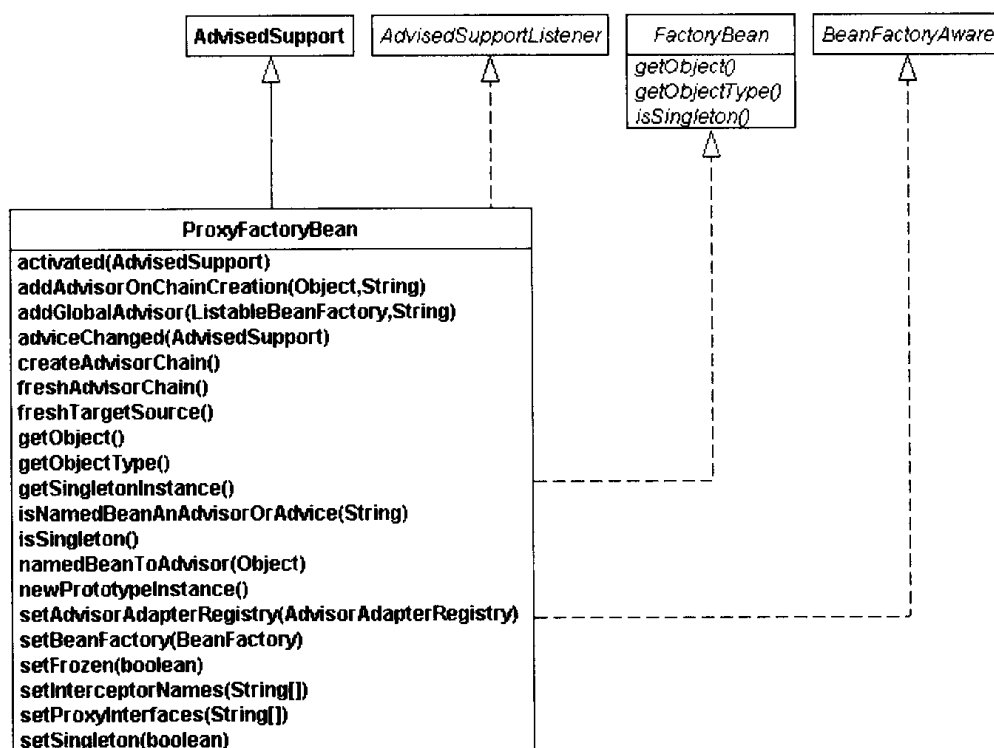


图 4-5 ProxyFactoryBean 类图

开发者可以看出,其具体包括如下几个主要属性。

- proxyInterfaces: 接口名构成的字符串列表,即接口集合。比如,example8 中 appcontext.xml 的取值, <value>com.openv.spring.IHelloWorld</value>。
- proxyTargetClass: 是否使用 CGLIB 代理目标类的标志位。开发者都知道,动态代理能够对接口指定代理,如果代理目标是类,则无能为力。因此,需要借助于 CGLIB 库,即实现类的子类,从而起到代理类的作用,这同动态代理在功能上异曲同工。

- **interceptorNames**: 拦截器名构成的字符串列表, 即拦截器集合。比如, example8 中 applicationContext.xml 的取值, `<value>loggingAroundAdvisor</value>`。
- **target**: 执行目标类, 即在 target 指定的 JavaBean 执行时, 将触发对上述拦截器的调用。比如, example8 中的取值, `<ref local="helloworldbeanTarget"/>`。
- **singleton**: 单实例的标志位, 即每次调用 ProxyFactoryBean 的 getObject() 方法 (BeanFactory 或 ApplicationContext 的 getBean() 方法) 时, 是返回同一对象, 还是返回不同的对象。

当然, 开发者也可以借助于 Spring AOP 手工创建 AOP 代理。因此, 这将会不依赖于 Spring IoC 容器, 而只是 Spring AOP 本身。接下来, 本书以 example9 为研究实例, 展开手工创建 AOP 代理的具体内容。

HelloClient.java 客户应用代码如下。

```
package com.openv.spring;

import org.aopalliance.aop.Advice;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.springframework.aop.framework.ProxyFactory;

/**
 * HelloWorld 客户应用
 *
 * @author luoshifei
 */
public class HelloClient {
    protected static final Log log = LogFactory.getLog(HelloClient.class);

    public static void main(String[] args) {
        //创建 LoggingAroundAdvice 装备
        Advice advice = new LoggingAroundAdvice();

        //创建 ProxyFactory, 从而不需要借助 Spring IoC 容器提供控制反转功能。
        ProxyFactory factory = new ProxyFactory(new HelloWorld());
        factory.addAdvice(advice);

        //调用业务操作
        IHelloWorld hw = (IHelloWorld) factory.getProxy();
        log.info(hw.getContent("luoshifei"));
    }
}
```

请开发者重点注意粗体部分。其具体执行结果如下 (Ant build.xml 中 run 任务)。


```
Buildfile: D:\workspace\example9\build.xml
```

```
compile:
```

```
run:
```

```
[java] 2004-12-12 20:47:23 org.springframework.core.CollectionFactory
<clinit>
[java] 信息: Using JDK 1.4 collections
[java] 2004-12-12 20:47:23 com.openv.spring.LoggingAroundAdvice invoke
[java] 信息: before: The Invocation of getContent()
[java] 2004-12-12 20:47:23 com.openv.spring.HelloWorld getContent
[java] 信息: luoshifei
[java] 2004-12-12 20:47:23 com.openv.spring.LoggingAroundAdvice invoke
[java] 信息: after: The Invocation of getContent()
[java] 2004-12-12 20:47:23 com.openv.spring.HelloClient main
[java] 信息: null
BUILD SUCCESSFUL
Total time: 2 seconds
```

好了，再也看不到 Spring IoC 容器初始化的内容了。因此，通过手工创建 AOP 代理，能够摆脱对 Spring IoC 容器的依赖。当然，从开发者实用性角度出发，这种方式不推荐使用。因为这不符合 Spring 框架的初衷。

Spring 框架开发 Team 推荐：借助于 Spring IoC 框架自动创建 AOP 代理，并将有关 AOP 代理的 Java 代码通过 Spring 配置文件配置。因此，它再次强调了 Spring 框架的实用性。

另外，FactoryBean 在 Spring 框架中起了很重要的作用。开发者已经看到，ProxyFactoryBean 实现了 FactoryBean 接口。同时，借助于 ProxyFactoryBean 的强大功能，开发者能够实现满足各种业务需求的实现，但是这要求开发者去额外开发很多辅助业务操作的功能，比如事务、数据库连接等操作。因此，Spring 框架针对具体方面而提供了大量的为开发者提供便利的类，比如 example11 实例中使用到的 TransactionProxyFactoryBean。通过研究 Spring 框架可以发现，它提供了如下一些实用类。至于它们的具体使用，本书将在有关章节分别给出方法。请注意，它们并不继承于 ProxyFactoryBean。

- TransactionProxyFactoryBean
- SimpleRemoteStatelessSessionProxyFactoryBean
- RmiProxyFactoryBean
- LocalStatelessSessionProxyFactoryBean
- JndiRmiProxyFactoryBean
- JmsProxyFactoryBean
- JaxRpcPortProxyFactoryBean
- HttpInvokerProxyFactoryBean
- HessianProxyFactoryBean
- BurlapProxyFactoryBean

Spring 框架在持续增加类似的实用类。

4.4 对 象 池

开发者在与数据库交互的过程中，通常都会使用到连接池。它能够在应用实际使用数据库连接前，同目标 RDBMS 建立物理连接。再比如，有过 EJB 组件开发经验的开发者都知道，EJB 容器维护了同一无状态会话 Bean（即 SLSB）的多个相同实例。一旦 EJB 客户调用 EJB 方法结束，将会把使用到的 EJB 实例返还到池中。Spring 框架也提供了类似的开发模型，即借助于 Jakarta Commons Pool 或者其他技术，从而提供有效的对象池实现。本书还是直接以实例展开具体研究吧！example10 就是为研究这样一个问题而开发的。当然，它是在前述实例的基础上演变而来的。

开发者可以研究下面给出的 appcontext.xml。其中，包括了详细的注释。请注意粗体部分。

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

    //helloworldbeanTarget 业务对象必须是原型（即 prototype）实现

    <bean id="helloworldbeanTarget"
        class="com.openv.spring.HelloWorld" singleton="false"/>
```

```

    </property>
    <property name="interceptorNames">
      <list>
        <value>loggingAroundAdvisor</value>
      </list>
    </property>
  </bean>

  <bean id="loggingAroundAdvisor"
    class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
      <ref local="loggingAroundAdvice"/>
    </property>
    <property name="pattern">
      <value>.*</value>
    </property>
  </bean>

  <bean id="loggingAroundAdvice"
    class="com.openv.spring.LoggingAroundAdvice"/>

</beans>

```

开发者可以运行 **Ant build.xml** 中的 **run** 任务。结果如下:

```

Buildfile: D:\workspace\example10\build.xml
compile:
run:
  [java] 2004-11-6 13:46:47 org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
  [java] 信息: Loading XML bean definitions from class path resource
[appcontext.xml]
  [java] 2004-11-6 13:46:48 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
  [java] 信息: Creating shared instance of singleton bean 'helloworldbean'
  [java] 2004-11-6 13:46:48 org.springframework.core.CollectionFactory
<clinit>
  [java] 信息: Using JDK 1.4 collections
  [java] 2004-11-6 13:46:48 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
  [java] 信息: Creating shared instance of singleton bean 'poolTargetSource'

//创建对象实例池

  [java] 2004-11-6 13:46:48 org.springframework.aop.target.
CommonsPoolTargetSource createPool
  [java] 信息: Creating Commons object pool
  [java] 2004-11-6 13:46:48

```

```
org.springframework.beans.factory.support.AbstractBeanFactory getBean
```

```
//创建 loggingAroundAdvisor 实例
```

```
[java] 信息: Creating shared instance of singleton bean  
'loggingAroundAdvisor'
```

```
[java] 2004-11-6 13:46:48 org.springframework.beans.factory.support.  
AbstractBeanFactory getBean
```

```
//创建 loggingAroundAdvice 实例
```

```
[java] 信息: Creating shared instance of singleton bean  
'loggingAroundAdvice'
```

```
[java] 2004-11-6 13:46:48 org.springframework.aop.support.  
RegexMethodPointcutAdvisor createPointcut
```

```
[java] 信息: Using Perl5RegexMethodPointcut for  
RegexMethodPointcutAdvisor
```

```
//日志信息略去了 Cglib2AopProxy 部分的输出内容
```

.....

```
//创建了 2 个 HelloWorld 实例
```

```
[java] 2004-11-6 13:46:48 com.openv.spring.HelloWorld <init>
```

```
[java] 信息: HelloWorld().....
```

```
[java] 2004-11-6 13:46:48 org.springframework.aop.target.  
AbstractPrototypeBasedTargetSource newPrototypeInstance
```

```
[java] 信息: Creating new target from bean 'helloworldbeanTarget'
```

```
[java] 2004-11-6 13:46:48 com.openv.spring.HelloWorld <init>
```

```
[java] 信息: HelloWorld().....
```

```
//调用 HelloWorld 的内容。在调用之前, LoggingAroundAdvice 拦截了客户请求。在
```

```
//调用之后, LoggingAroundAdvice 又作了其他一些处理操作
```

```
[java] 2004-11-6 13:46:48 com.openv.spring.LoggingAroundAdvice invoke
```

```
[java] 信息: before: The Invocation of getContent()
```

```
[java] 2004-11-6 13:46:48 com.openv.spring.HelloWorld getContent
```

```
[java] 信息: luoshifei
```

```
[java] 2004-11-6 13:46:48 com.openv.spring.LoggingAroundAdvice invoke
```

```
[java] 信息: after: The Invocation of getContent()
```

```
[java] 2004-11-6 13:46:48 com.openv.spring.HelloClient main
```

```
[java] 信息: null
```

```
BUILD SUCCESSFUL
```

```
Total time: 2 seconds
```

其中, 开发者能够注意到, Spring 框架初始化了 2 个 HelloWorld 实例, 并放置在池中, 供应用使用。详情请开发者参考 example10 提供的完整内容。

一般而言, 应用不需要使用这种池化无状态服务。Spring 框架开发 Team 推荐: 一般不需要使用, 因此默认时 Spring 框架开发的 Java/J2EE 应用都不会使用到这种对象池技术。

4.5 小 结

本章内容对 Spring AOP 作了较深入研究。其中, 主要从两方面进行。

其一, 从 Spring AOP 装备 (Advice) 角度, 给出了常见的 4 种装备的开发。当然, 这些开发实例只是演示了 Spring AOP 对 AOP 联盟定义的那些接口的实现。而且, 这种实现是在很低的层面, 即直接借助于 AOP 的基本接口实现 Advice。

其二, 给出了 ProxyFactoryBean 介绍。另外还对 FactoryBean 及其相关实现给出了大体介绍。借助于 ProxyFactoryBean, 开发者能够很抽象地在高层面使用 Spring AOP 技术, 比如 TransactionProxyFactoryBean。

另外, 还对对象池作了较深入介绍。本章对 Spring AOP 框架提供的其他一些重要功能并没有给出研究内容, 比如自动 AOP 代理 (它使得配置文件的可读性差)。不管如何, 从使用 Spring AOP 框架开发 Java/J2EE 的角度出发, 本章介绍的内容一般能够满足各种场合的需求。

无论如何, AOP 能够很好地补充 OOP 技术。通过 AOP Aspect 的形式, 将系统中的横切问题整理出来, 比如日志、事务、安全性 (借助于 Acegi 安全框架实现, 见第 17 章) 等, 进而减少重复代码。正因为有了 Spring AOP 的支持, 才使得声明式事务和安全性成为可能。因此, Spring AOP 使得 Spring 更具对抗性, 从而与 EJB 等标准技术进行对抗。如果 Spring 未提供 AOP 实现, 不知道在 J2EE 领域是否还会有 Spring 的存在。幸好, Spring 开发者能够享受到 Spring AOP。

让我们进一步进入 Spring 架构吧!

第5章 深入 Spring 架构

Spring 架构，本章将重点从 Spring 具体构件入手，即从 Spring 开发 Java/J2EE 应用的角度出发，分析 Spring 为开发者提供了哪些功能模块，这对于掌握 Spring 很重要。

最后，本章将详细给出 example11 实例的开发和实现，这对于掌握 Spring 架构起到了很重要的作用。“麻雀”虽小，但是 example11 功能很齐全，足够反映 Spring 架构的核心内容。

5.1 架构概述

Spring 框架的分层工作，即模块化，完成得非常好。大体上，存在如图 5-1 所示的几个模块。

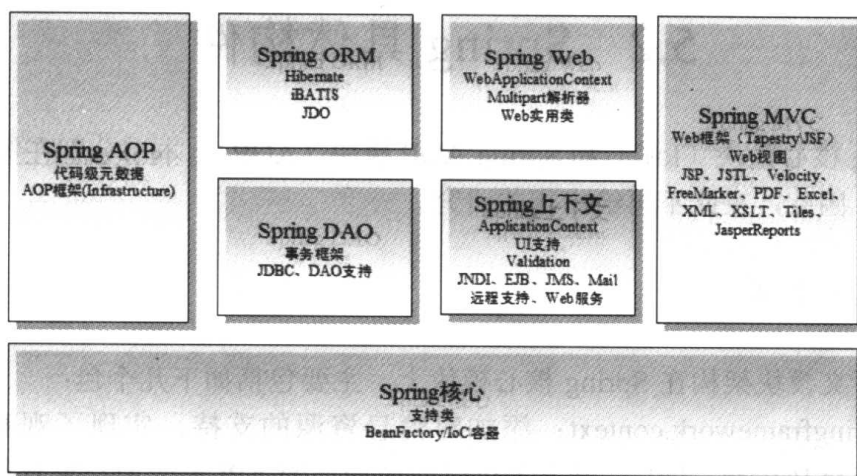


图 5-1 Spring 架构¹

其中：

- **Spring 核心模块**：Spring 框架中最为基础、重要的模块。它提供了 IoC 容器，即依赖注入。其中，BeanFactory 是最为重要的概念，这对于理解 IoC 起到关键的作用。另外，对理解 ApplicationContext 也起到了很重要的作用。
- **Spring AOP 模块**：实现了 AOP 联盟中定义的 AOP 编程实现。比如，提供拦截器实现事务管理。
- **Spring 上下文 (Context) 模块**：直接位于 Spring 核心模块之上。Spring 上下文模块除了继承 Spring 核心模块的功能外，还添加了用于资源绑定、事件移植、资源装载以及透明地装载上下文等功能。这对于 Web 应用和 J2EE 应用尤为有效，甚至可以认为 Spring 引入上下文模块更多地是为了简化开发 Web 应用和 J2EE 应用

¹ 基于 Spring Reference 官方文档制作而成。

目的的。这种对 J2EE 平台服务的抽象使得 Spring 在开发者中大受欢迎。

- Spring Web 模块：提供面向 Web 应用集成的功能。当然，这只是初步集成。其中，ContextLoaderServlet 和 ContextLoaderListener，正是 Web 模块提供的。当同 Tapestry、JSF 集成时，需要使用到 Spring Web 模块。
- Spring DAO 模块：提供了 JDBC 抽象层，使得开发者不用再去编写同 RDBMS 交互、非业务功能的 JDBC 代码。而且 DAO 模块还能够分析 RDBMS 厂商专有的 SQL 错误代码。最重要的一点，它同时能够提供编程方式和声明方式控制事务。
- Spring ORM 模块：为当前流行的 O/R Mapping 技术提供集成。比如，Hibernate、JDO、iBATIS。借助于 Spring 框架提供的简单事务声明，开发者能够很容易实现对 O/R Mapping 中操作的事务控制。
- Spring Web MVC 模块：提供的 MVC 实现。请注意，Spring Web MVC 实现遵循 MVC 架构，清晰地划分了 Web 应用中涉及到的各项内容，比如检验器、Web 表单生成等。

5.2 Spring 具体构件

对于 Spring 核心模块 (IoC) 和 Spring AOP 模块 (AOP)，本书分别在第 3、4 章给出了详细的阐述。因此，在此不再给出相关研究。

5.2.1 Spring 上下文

Spring 上下文模块架构在 Spring 核心模块上。主要包括如下几个包：

- org.springframework.context：添加对消息资源的支持，实现了观察者设计模式 (Design Pattern, DP)。开发者可以使用统一的 API 为应用对象获得应用资源。
- org.springframework.context.access：辅助 Spring 上下文模块，供定位和访问应用上下文使用。
- org.springframework.context.event：为应用事件 (比如标准上下文事件) 提供支持。
- org.springframework.context.support：供 context 包使用。

对于 Web 应用和 J2EE 应用而言，很少需要在应用代码中涉及到 Spring 框架提供的 API。这也是 Spring 框架开发 Team 的初衷。这使得移植基于 Spring 的 Java/J2EE 应用变得简单、可行。当然，Spring 上下文模块也不例外。

ApplicationContext 同 Spring 应用开发者的关系最为直接。图 5-2 给出了与它相关的接口和类的类图。

开发者都知道，ApplicationContext 在非 Web/J2EE 应用场合也可以使用。比如，在 example10 的 HelloClient.java 中存在如下代码：

```
Resource resource = new ClassPathResource("appcontext.xml");
BeanFactory factory = new XmlBeanFactory(resource);
IHelloWorld hw = (IHelloWorld) factory.getBean("helloworldbean");
log.info(hw.getContent("luoshifei"));
```

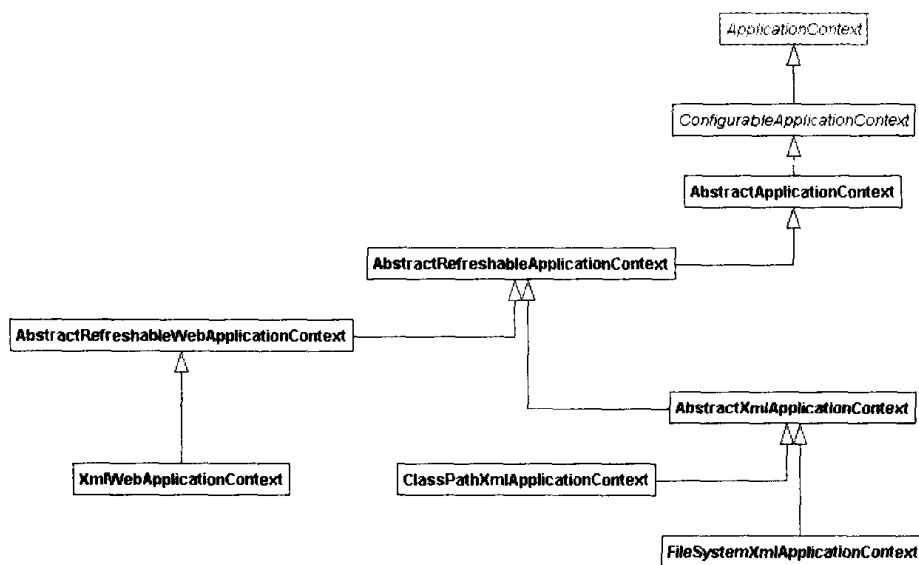



图 5-2 ApplicationContext 接口及其实现

如果使用 `ApplicationContext` 及其 `ClassPathXmlApplicationContext` 实现，则可以通过如下示例给出。

```

ApplicationContext ac = new
    ClassPathXmlApplicationContext("appcontext.xml");
IHelloWorld hw = (IHelloWorld)ac.getBean("helloworldbean");
log.info(hw.getContent("luoshifei"));

```

如果使用 `ApplicationContext` 及其 `FileSystemXmlApplicationContext` 实现，则可以通过如下示例给出。

```

ApplicationContext ac = new
    FileSystemXmlApplicationContext("src/appcontext.xml");
IHelloWorld hw = (IHelloWorld)ac.getBean("helloworldbean");
log.info(hw.getContent("luoshifei"));

```

请注意，上述的“`src/appcontext.xml`”粗体部分。

5.2.2 Spring Web

由于 Spring Web 和 Spring Web MVC 模块的关系很紧密，再加上 Spring 框架中用于这两部分的 Java 包（package）数量很多，因此本书在此需要同开发者做一个约定：`org.springframework.servlet` 及其子包在 Spring Web MVC 模块中阐述，其他包在 Spring Web 模块中阐述，即本节内容。否则，分析这两部分内容将变得很困难。

Spring Web 主要包括如下几个包：

- `org.springframework.web.bind`：为 Web 应用提供数据绑定功能。同时，还存在调用数据绑定和有效性校验的实用类。
- `org.springframework.web.context`：为 Web 应用提供 Web 上下文子接口和类。其中，包括了用于引导 Web 根上下文的监听器，即 `ContextLoaderListener` 和 `Servlet`（`ContextLoaderServlet`）。
- `org.springframework.web.context.support`：用于支持 `web.context` 包。比如其中包含

了 `WebApplicationContext` 实现、用于获取 Web 根上下文的实用类。

- `org.springframework.web.filter`: 供通用用途的过滤器基类，并可以通过类似于“<bean>”元素的方式进行配置。
 - `org.springframework.web.jsf`: 用于集成 JSF Web 层的支持类。
 - `org.springframework.web.multipart`: 供处理文件上传使用。其中，提供了策略接口，即 `MultipartResolver`。另外，还提供了 `HttpServletRequest` 接口的通用扩展，即 `MultipartHttpServletRequest` 接口，供访问 Web 应用代码中的多部分（multipart）文件使用。
 - `org.springframework.web.multipart.commons`: `MultipartResolver` 实现，即集成了 Jakarta Commons FileUpload。
 - `org.springframework.web.multipart.cos`: `MultipartResolver` 实现，即集成了 Jason Hunter 的 COS（`com.oreilly.servlet`²）。
 - `org.springframework.web.multipart.support`: 为 `web.multipart` 包提供支持。
- 本书后续内容（第二、三部分）会详细阐述 Spring Web 模块中的重要内容。

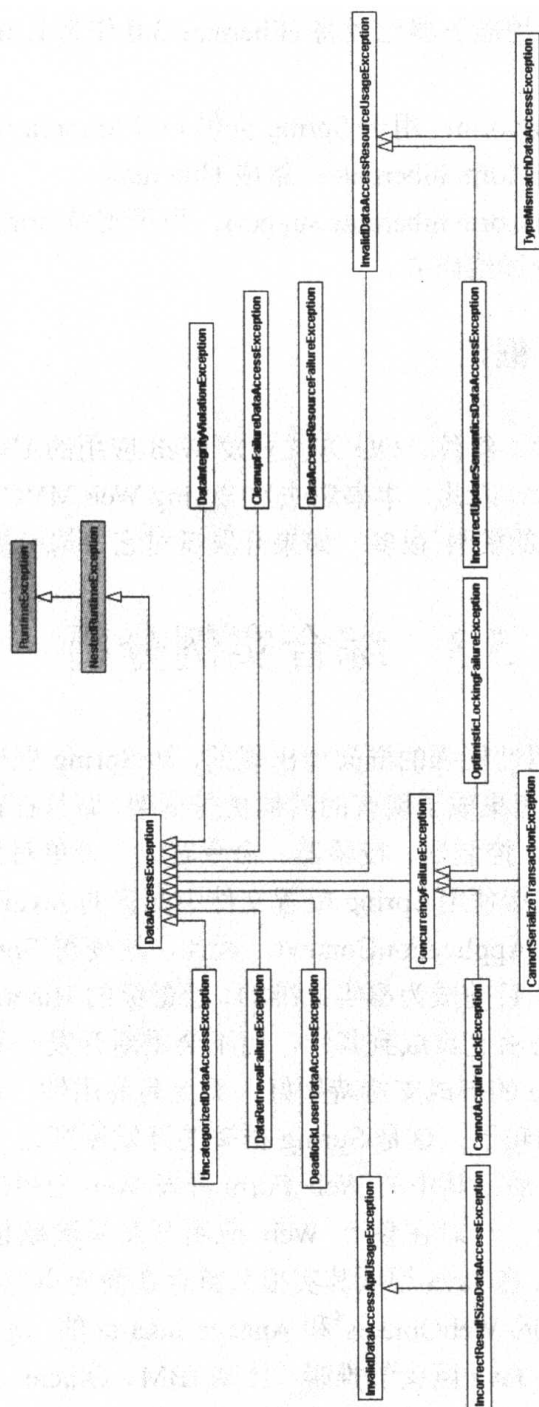
5.2.3 Spring 数据访问对象（DAO）

Spring DAO 模块包含了如下几个重要的包：

- `org.springframework.dao`: 提供异常处理类。所有的异常类都是未受查的，因此开发者可以选择不用在应用代码中处理它们。
- `org.springframework.dao.support`: 支持 dao 包的类。
- `org.springframework.jdbc`: 供简化对 RDBMS 的访问使用。其中，对异常处理的错误处理得很好，这在很大程度上要归功于 dao 包。
- `org.springframework.jdbc.core`: 提供基于 `JdbcTemplate` 的 JDBC 核心框架。另外，还提供相应的回调接口、辅助对象。
- `org.springframework.jdbc.core.support`: 用于支持 `jdbc.core` 包。
- `org.springframework.jdbc.datasource`: 提供访问数据源的实用类。其中，还提供了用于单一数据源的 `PlatformTransactionManager`、若干简单的数据源实现类。
- `org.springframework.jdbc.object`: 将 RDBMS 操作过程中的查询、更新以及存储过程表示成线程安全的可重用的对象。
- `org.springframework.jdbc.support`: 用于 JDBC 框架的支持类，供 `jdbc.core` 和 `jdbc.object` 包使用。其中，还提供了将 SQL 异常翻译成 Spring 的通用 `DataAccessException` 异常层次结构中的具体异常。
- `org.springframework.jdbc.support.incrementer`: 借助于序列号，为不同 RDBMS 提供自增字段值生成。
- `org.springframework.jdbc.support.lob`: 为处理大对象提供策略接口。其中，为若干数据库提供了具体实现。

² <http://www.servlets.com/cos/>

- 其中，在 `dao` 包中提供的异常处理类在整个 Spring 框架中占有很重要的位置。因此，这里重点给出该包的内容，其他包的内容将在第二部分涉及到。图 5-3 给出了 `dao` 包中所有类的类图。



一般情况下，开发者在应用代码中只会使用到 `DataAccessException` 异常类，这大大简化了异常处理的复杂度。

5.2.4 Spring ORM

Spring ORM 实现了对 O/R Mapping 技术的集成支持。目前,主要的 O/R Mapping 技术有, Hibernate、iBATIS、JDO 以及 OJB。本书将重点介绍对 Hibernate 的集成支持。其他的 O/R Mapping 技术集成,对于 Spring 的使用而言都是类似的,因此对于 iBATIS、JDO 以及 OJB 的集成支持不再给出介绍。再者,就 Hibernate 的发展现状而言,它已经成为了事实上的工业标准,甚至 JBoss 应用服务器已经将 Hibernate 3.0 作为 EJB 3.0 的重要实现技术。

主要的包如下:

- org.springframework.orm: 用于 Spring 提供 O/R Mapping 的集成类。
- org.springframework.orm.hibernate: 集成 Hibernate。
- org.springframework.orm.hibernate.support: 用于支持 orm.hibernate 包。
- 本书将在第二部分详细研究。

5.2.5 Spring Web MVC 框架

本书推荐使用基于事件、组件、OO 方式开发 Web 应用的 UI 层,比如用于 Web 应用的 Tapestry 和 JSF 组件技术。因此,本书略去对 Spring Web MVC 框架的深入分析。当然,现有介绍 Spring Web MVC 的资料³很多,如果开发者对它“情有独钟”的话。

5.3 综合实例分析

由于 Spring 是作为实用性极强的框架而出现的,即 Spring 框架本身在提供其他流行框架不具有的功能的同时,它还集成了现有的其他优秀框架,而且在持续改善中。比如, Spring MVC 框架,它清晰地分离了控制器、校验器、命令对象、表单对象、模型对象、视图解析器等模块。同时,它能够直接使用 Spring 配置文件中配置的 JavaBean,而不用类似于其他 MVC 框架需要集成 Spring ApplicationContext。因此,这使得 Spring MVC 框架开发 Web 应用更简单、高效。再比如,目前成为事实标准的、轻量级的 Hibernate 实现了 O/R Mapping 功能,因此 Spring 会尽量去将它集成到其中,而不会重新开发一套 O/R Mapping 框架。事实证明, Spring 对 Hibernate 的集成支持特别好,无论是易用性,还是性能上。

Spring 框架不重复发明轮子,这是 Spring 框架的开发原则之一。

在 .NET 推出 ASP.NET 后,其中的 Web Form 开发 Web 应用的模式,即面向事件、基于组件模式开发 Web 应用,一时在整个 Web 应用开发领域掀起了一阵狂潮。当然,在 ASP.NET Web Form 推出前,在 Java 领域其实很久就存在面向事件、基于组件模式开发 Web 应用的框架,即 Apple 公司的 WebObjects⁴和 Apache Jakarta 的 Tapestry⁵ Web 应用框架。其中,新近的 JSF 技术备受各 Java 巨头的推崇,比如 IBM、Oracle、Sun,从这可以看出面向事件、基于组件模式开发 Web 应用将是未来几年的热门。本书不讨论 WebObjects,而重点

³ <http://www.devx.com/Java/Article/22134/0/page/1>

⁴ 其详细介绍,可以通过如下网址获得: <http://developer.apple.com/webobjects/>。

⁵ <http://jakarta.apache.org/tapestry>

看一下 Tapestry 和 JSF。

开发者可能会问,上面给出的一段内容好像同 Spring 框架本身没有关系。是的,确实没有关系。但是,不要忘记, Spring 灵活的架构使得它集成各种现有的和未来产生的新兴框架很简单。甚至,开发者不用作任何集成工作,就能够直接混合使用 Spring 和其他优秀框架的各自功能,比如 Tapestry。

既然该小节内容需要给出综合实例分析。那么,什么样的实例具有代表意义呢?开发者可以给出一些意见。当然,本书认为能够符合如下几方面条件的一定具有说服力。

- 应用架构模式必须是 Web 应用。
- 需要体现 Spring IoC 容器和 Spring AOP 框架的核心功能。这体现了 Spring 框架本身的功能。
- 对于企业级应用而言,与关系数据库的交互必不可少,使用流行、实用以及高效的 O/R Mapping 框架搭建整个 Web 应用的后端功能很有必要。这体现了 Spring 框架在集成其他优秀框架所完成的杰出工作。
- 对于 Web 应用而言,开发 Web 界面的低效率一直使开发者很为难。那么选用高效的 Web 框架,并且配套 Web 框架的文档和工具应用很丰富,即社区很广泛、很成熟。这体现了 Spring 框架优秀的集成性、可扩展性以及柔韧性。

因此,在认真考虑和斟酌后,本书采用了 Spring/Tapestry/Hibernate 作为开发综合实例的框架。理由如下(选用 Spring 的理由就不用解释了)。

- Tapestry: 功能非常强大的、Open Source 开发、纯 Java 的框架,主要供开发 Web 应用的 Web 层使用。对于开发 Web 应用而言, Tapestry 不是采用传统的方式,即使用 URL 和查询参数,而是使用对象、方法以及属性来开发 Web 应用。这种方式同开发传统的桌面应用类似,比如基于 Dephi/VB/Swing/SWT 开发的桌面应用。其开发效率可想而知。采用面向对象技术开发 Web 应用,这就是 Tapestry。另外,在 Tapestry 中,全部都是组件,这在很大程度上提高了 Web 应用的复用性,不会在使用其他 Web 框架中将开发者带入“过程式”的面向对象中,比如 Struts (注意,本书并不是反对 Struts,而只是说使用 Struts 很容易将开发者带入其中。)。Tapestry 提供的文档很丰富,并且存在配套的、功能强大的开发工具⁶。Spring 不需要对 Tapestry 做任何框架专有的工作,就可在 Tapestry 应用中使用到 Spring 强大的功能。因此,有理由相信,选用 Tapestry 作为 Web 层是正确的。
- Hibernate: 对于开发者而言,将面向对象的软件和 RDBMS 一起使用显得有些麻烦、低效率。Hibernate 是一个面向 Java 开发环境的、O/R Mapping 工具,即对象/关系数据库映射工具。开发者都知道, Hibernate 不仅能够管理 Java 类(主要是 JavaBean,或者称之为 POJO)到 RDBMS 表的映射,它还能够供应用查询和获取数据库中的数据。因此,从开发效率上考虑, Hibernate 能够大大提高开发者使用 SQL 和处理 JDBC 与数据库交互的效率。目前, Hibernate 是事实上的 O/R Mapping

⁶ 目前存在两种辅助 Tapestry 开发 Web 应用的工具。其一,基于 JBuilder IDE 的插件,即 Weaver,具体网址位于 <http://cc.borland.com/codecentral/ccweb.exe/listing?id=21504>。其二,基于 Eclipse IDE 的插件,即 Spindle,具体网址位于 <http://spindle.sourceforge.net/>。本书推荐使用 Spindle。

功能,尤其是 EJB 3.0 中的实体 Bean 借鉴了 Hibernate 的大量经验。开发者都知道 Open Source 的框架,如果文档不够丰富,则学习曲线会很高。Spring/Hibernate/Tapestry 都做得不错,比如新近的 Hibernate 在每次发布时,都能够同时提供多种翻译版本。Hibernate 在 JBoss 公司的赞助下,全球的培训和支持服务已经广泛展开。Hibernate 的使用和执行效率很受开发者的青睐。不需要使用任何 Hibernate 的配套工具编写 Hibernate 相关配置文件和源代码,开发者就能够快速地完成它们的开发。因此,也没有理由不选用 Hibernate 作为开发综合实例的示范框架。

当然,作为示范实例,只要能够简洁、高效地说明问题,不追求业务的复杂度最好,这是开发者所乐意看到的。在此,我们需要对 JavaBean 和 POJO 之间的区别说明一下。JavaBean 类需满足 JavaBean 规范,比如采用标准的 setter/getter 方法设置成员变量。POJO,只是普通的 Java 类。可以认为,JavaBean 类都是 POJO,但 POJO 类未必是 JavaBean。

至此,让我们进入实例吧!

5.3.1 实例概述

本综合实例只有一个简单界面,简单得不够真实。注意,不要小看它,其中的内容还是很丰富的,这使得打破示范实例的底线再也不可能。

通常,在 Portal 网站注册个人信息时,都需要输入用户信息,其中包括用户名、密码(和密码的确认)、E-mail、个人爱好以及国别等信息。因此,这就是综合实例的应用场景,或者说业务流程所包括的内容。

首先,让我们先睹为快,看看实例的功能吧!(随后给出实例的安装和配置,请注意界面本身更多地体现了 Tapestry 的强大之处。而本实例的使用对于 Tapestry 而言,只是冰山一角。)

接下来,本书将给出综合实例的首页面。开发者看到的图 5-4 是综合实例的默认情形,通过类似于 <http://localhost:8080/example11> 的 URL 能够访问到它。其中,用户名、密码、确认密码以及 E-mail 输入域为必须输入表单域,图中用“*”图形表示。而兴趣和国别域为选择域,即兴趣域为多选域、国别域为单选域。

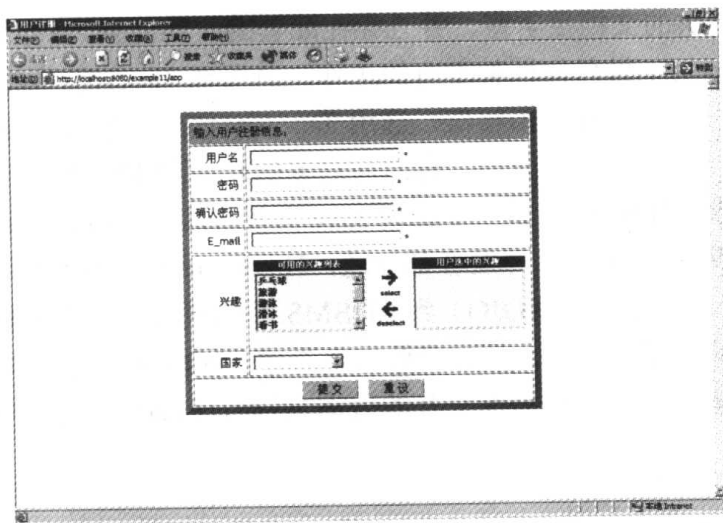


图 5-4 综合实例首界面

其次，图 5-5 给出了在开发者输入了无效的 E-mail 域时发生的场景。开发者应该注意到，E-mail 的格式是有一定要求的，即符合类似于“xxx@yyy.zzz”的模式。当然，对于其他必录项也存在相应的客户端校验逻辑。在 JBoss 服务器端（其实，与具体服务器无关），也存在相同的检验逻辑，只是实现的技术不同而已。在客户端，通常都是使用 JavaScript 脚本完成；在服务器端，使用 Java 代码。借助于 Tapestry，这些校验逻辑通常都不需要开发者编写，包括客户端 JavaScript 的自动生成。

图 5-5 自动在客户端完成输入数据的校验工作

第三，对于客户端输入了空格字符的情形，在浏览器中都不太好处理。没有关系，服务器端能够代劳，图 5-6 就是展示了这种情形。

图 5-6 服务器端对空格表单域的自动处理

第四，对于有些域存在字符长度要求，比如要求密码表单域必须是 3 位以上。Tapestry 提供的校验器能够很容易处理这些事情，图 5-7 展示了这种场景。当然，在服务器端同样

也存在功能相同的校验逻辑。对于 Web 应用的设计而言,服务器端的校验工作最为关键和可信。



图 5-7 客户端对字符串长度的自动校验

最后一步,在用户输入表单域无误的情况下,经过服务器端的校验,包括语法和内容校验,开发者能够看到如下场景,如图 5-8 所示。当然,这是服务器端 JBoss、Spring、Tapestry、Hibernate、MySQL 合作的结果,也不要忘记浏览器以及其他协作者的功劳。

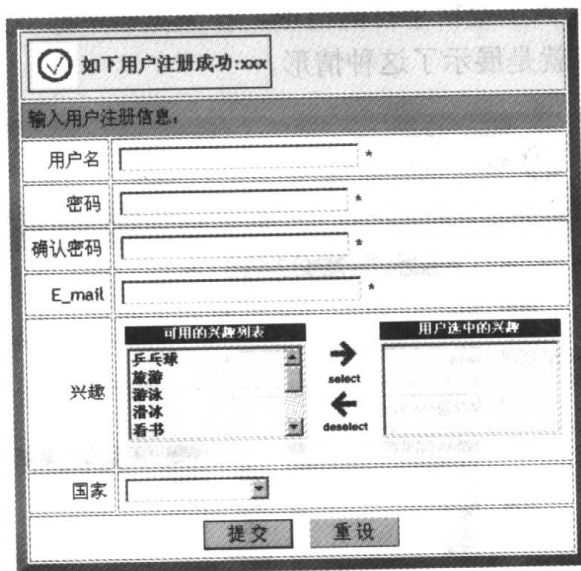


图 5-8 服务器端成功处理用户注册信息

不知道开发者是否注意到,多选项“兴趣”在一般情况下,开发交互性极强的多选功能,对于 Web 应用而言是一件痛苦的事情。但是,借助于 Tapestry 提供的 Palette 组件(用 Tapestry 创始人 Howard 在 2001 年所说⁷, Palette 组件是 Tapestry 较复杂的组件之一),这都

⁷ 位于如下网址, <http://www.onjava.com/pub/a/onjava/2001/11/21/tapestry.html>。

不是问题。

另外,本书还需要强调几点。其一,对于那些存在校验问题的表单域,实例会将相应内容用红色标识出来,使得用户更容易找到错误所在,这都是 Tapestry 提供的,开发者不需要编写任何代码。其二,在客户端, Tapestry 生成了很多 JavaScript 代码,本文并没有给出。其三,本书并不是阐述 Tapestry 的使用和架构的,因此都假定用户对 Tapestry 有所熟悉⁸,包括其他框架,比如 Hibernate。当然,本书在第三部分对 Tapestry 作了基础知识介绍。

至此,开发者是否已经等不及了,想跃跃欲试如何安装和配置综合实例,即 example11。

5.3.2 安装和配置 example11

全书实例的安装和配置,开发者可以参考附录 A,对于一些实例的安装细节,需要参考各个实例的相关部分。接下来,本书详细给出 example11 的安装和配置(本书假定读者都有一定 Java 基础,因此一些 Java 细节并不会给出。如果存在本书的相关问题,可以直接找本书的作者),本书实例安装在 D 盘。

步骤

(1) 开发者在安装和配置 example11 之前,需要将图 5-9 给出的所有 jar 文件拷贝到如下目录,即 D:\workspace\example11\context\WEB-INF\lib。当然,本书出版时,有些 jar 库可能已经作了更新,开发者可以根据实际情况使用。

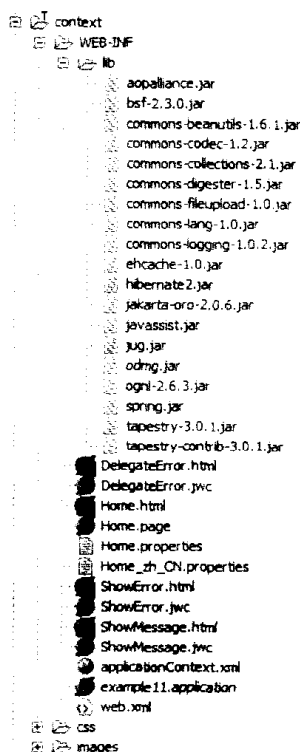


图 5-9 example11 使用的 jar 文件

⁸ 参考 Howard Ship 著的《Tapestry in Action》一书。

其中, jug.jar⁹是本书用来创建 RDBMS 表中主键 ID 的。其他的类库, 通过 Spring/Tapestry/Hibernate 发布版都能够找到, 具体内容可以参考各自给出的 readme.txt 文件。

(2) 开发者借助于 MySQL 客户端工具能够创建 example11 要求的数据库。SQL 脚本文件位于 D:\workspace\example11\other\example11.mysql.sql 文件中。

(3) 由于本实例使用了 JBoss 中的 MySQL 数据源, 因此开发者需要根据如下步骤完成。

- 获得 MySQL JDBC 驱动, 见附录 A。并拷贝到 D:\jboss-4.0.0\server\default\lib 目录。比如, mysql-connector-java-3.1.4-beta-bin.jar。
- 获得 mysql-ds.xml 文件。开发者可以从目录 (即, D:\jboss-4.0.0\docs\examples\jca) 获得 MySQL 数据源模板文件, 即 mysql-ds.xml。然后, 拷贝到 D:\jboss-4.0.0\server\default\deploy 目录。
- 修改 mysql-ds.xml 文件。示例配置见图 5-10。

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

<!-- $Id: mysql-ds.xml,v 1.3 2004/09/15 14:37:40 loubyansky Exp $ -->
<!-- Datasource config for MySQL using 3.0.9 available from:
http://www.mysql.com/downloads/api-jdbc-stable.html
-->

<datasources>
  <local-tx-datasource>
    <jndi-name>MySqlDS</jndi-name>
    <connection-url>jdbc:mysql://localhost:3306/example11</connection-url>
    <driver-class>com.mysql.jdbc.Driver</driver-class>
    <user-name>root</user-name>
    <password></password>

    <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml (optional) -->
    <metadata>
      <type-mapping>mySQL</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>
```

图 5-10 MySQL mysql-ds.xml 配置

- 保存 mysql-ds.xml 文件。

(4) 通过 JBoss IDE 提供的 Server Navigator 视图, 开发者能够启动 JBoss 应用服务器, 如图 5-11 所示。当然, 开发者也可以在 Eclipse IDE 外部启动 JBoss (直接在 D:\jboss-4.0.0\bin 目录运行 run.bat)。

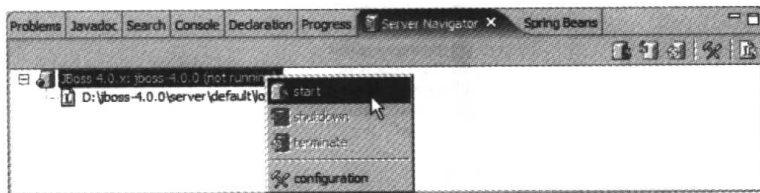


图 5-11 启动 JBoss 应用服务器

⁹ 位于如下网址, <http://www.doomdark.org/doomdark/proj/jug/>。

(5) 开发者在 Eclipse IDE 中编译 example11 后, 通过运行 D:\workspace\example11 下 Ant build.xml 中的 deploy 任务能够完成 example11.war 的部署。

其中, 具体内容见图 5-12。

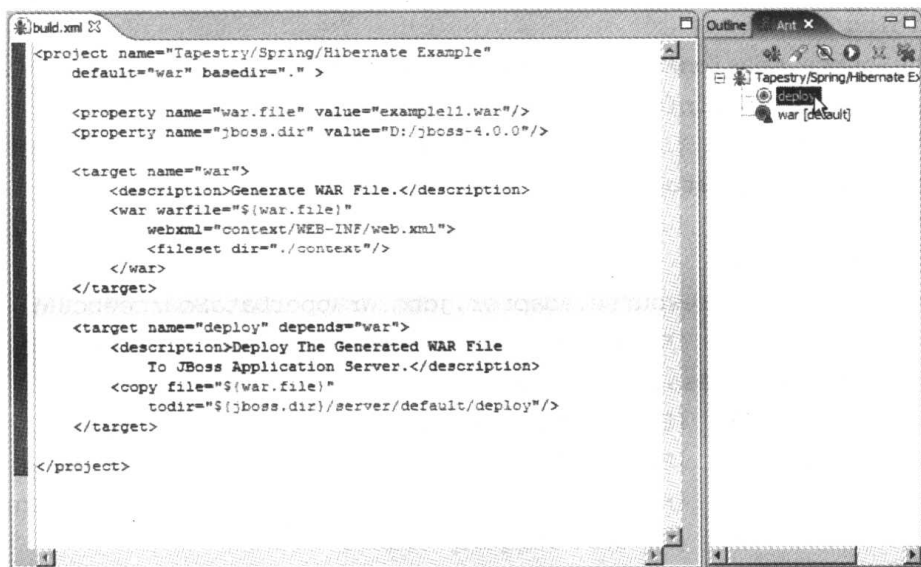


图 5-12 Ant build.xml 文件

通过上述过程, 本书完成了 example11 的部署工作。JBoss 部署 example11.war 的步骤见如下日志。

如果开发者看到类似日志, 则表明 example11.war 已成功部署。然后, 通过浏览器可以检验应用是否成功部署, 即 “http://localhost:8080/example11”。

```
22:09:53,896 INFO [TomcatDeployer] deploy, ctxPath=/example11,
warUrl=file:/D:/jboss-4.0.0/server/default/tmp/deploy/tmp32218example11-e
xp.war/
22:09:54,257 INFO [Engine] StandardContext[/example11]Loading root
WebApplicationContext
22:09:54,467 INFO [XmlBeanDefinitionReader] Loading XML bean definitions from
resource [/WEB-INF/applicationContext.xml] of ServletContext
22:09:54,787 INFO [XmlWebApplicationContext] Bean factory for application
context [Root XmlWebApplicationContext]:
org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans
[dataSource, sessionFactory, transactionManager, example11ServiceTarget, exam
ple11Service, userinfoDAO]; root of BeanFactory hierarchy
22:09:54,808 INFO [XmlWebApplicationContext] 6 beans defined in application
context [Root XmlWebApplicationContext]
22:09:54,828 INFO [XmlWebApplicationContext] No message source found for
context [Root XmlWebApplicationContext]: using empty default
22:09:54,848 INFO [XmlWebApplicationContext] No ApplicationEventMulticaster
found for context [Root XmlWebApplicationContext]: using default
22:09:54,868 INFO [UiApplicationContextUtils] No ThemeSource found for [Root
XmlWebApplicationContext]: using ResourceBundleThemeSource
```

```
22:09:54,888 INFO [XmlWebApplicationContext] Refreshing listeners
22:09:54,888 INFO [DefaultListableBeanFactory] Pre-instantiating singletons
in factory
[org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans
[dataSource, sessionFactory, transactionManager, example11ServiceTarget, exam
ple11Service, userinfoDAO]; root of BeanFactory hierarchy]
22:09:54,888 INFO [DefaultListableBeanFactory] Creating shared instance of
singleton bean 'dataSource'
22:09:55,118 INFO [JndiObjectFactoryBean] Located object with JNDI name
[java:/MySqlDS]:
value=[org.jboss.resource.adapter.jdbc.WrapperDataSource@bc8690]
22:09:55,128 INFO [DefaultListableBeanFactory] Creating shared instance of
singleton bean 'sessionFactory'
22:09:55,178 INFO [Environment] Hibernate 2.1.5
22:09:55,188 INFO [Environment] hibernate.properties not found
22:09:55,188 INFO [Environment] using CGLIB reflection optimizer
22:09:55,398 INFO [Binder] Mapping class:
com.openv.spring.service.hibernate.Interest -> interest
22:09:55,639 INFO [Binder] Mapping class:
com.openv.spring.service.hibernate.UserInterest -> userinterest
22:09:55,679 INFO [Binder] Mapping class:
com.openv.spring.service.hibernate.UserInfo -> userinfo
22:09:55,679 INFO [LocalSessionFactoryBean] Building new Hibernate
SessionFactory
22:09:55,679 INFO [Configuration] processing one-to-many association mappings
22:09:55,679 INFO [Configuration] processing one-to-one association property
references
22:09:55,679 INFO [Configuration] processing foreign key constraints
22:09:55,759 INFO [Dialect] Using dialect:
net.sf.hibernate.dialect.MySQLDialect
22:09:55,759 INFO [SettingsFactory] Use outer join fetching: false
22:09:55,769 INFO [ConnectionProviderFactory] Initializing connection
provider:
org.springframework.orm.hibernate.LocalDataSourceConnectionProvider
22:09:55,779 INFO [TransactionManagerLookupFactory] No
TransactionManagerLookup configured (in JTA environment, use of process level
read-write cache is not recommended)
22:09:56,159 INFO [SettingsFactory] Use scrollable result sets: true
22:09:56,159 INFO [SettingsFactory] Use JDBC3 getGeneratedKeys(): true
22:09:56,159 INFO [SettingsFactory] Optimize cache for minimal puts: false
22:09:56,159 INFO [SettingsFactory] echoing all SQL to stdout
22:09:56,159 INFO [SettingsFactory] Query language substitutions: {}
22:09:56,159 INFO [SettingsFactory] cache provider:
net.sf.ehcache.hibernate.Provider
22:09:56,179 INFO [Configuration] instantiating and configuring caches
```

```
22:09:56,450 INFO [SessionFactoryImpl] building session factory
22:09:56,971 INFO [SessionFactoryObjectFactory] no JNDI name configured
22:09:56,971 INFO [DefaultListableBeanFactory] Creating shared instance of
singleton bean 'transactionManager'
22:09:57,061 INFO [HibernateTransactionManager] Using DataSource
[org.jboss.resource.adapter.jdbc.WrapperDataSource@bc8690] of Hibernate
SessionFactory for HibernateTransactionManager
22:09:57,061 INFO [DefaultListableBeanFactory] Creating shared instance of
singleton bean 'example11ServiceTarget'
22:09:57,061 INFO [Example11ManagerImpl]
Example11ManagerImpl().....
22:09:57,081 INFO [DefaultListableBeanFactory] Creating shared instance of
singleton bean 'userinfoDAO'
22:09:57,181 INFO [XmlBeanDefinitionReader] Loading XML bean definitions from
class path resource [org/springframework/jdbc/support/sql-error-codes.xml]
22:09:57,221 INFO [XmlBeanFactory] Creating shared instance of singleton bean
'DB2'
22:09:57,231 INFO [XmlBeanFactory] Creating shared instance of singleton bean
'HSQL'
22:09:57,241 INFO [XmlBeanFactory] Creating shared instance of singleton bean
'MS-SQL'
22:09:57,261 INFO [XmlBeanFactory] Creating shared instance of singleton bean
'MySQL'
22:09:57,261 INFO [XmlBeanFactory] Creating shared instance of singleton bean
'Oracle'
22:09:57,261 INFO [XmlBeanFactory] Creating shared instance of singleton bean
'Informix'
22:09:57,261 INFO [XmlBeanFactory] Creating shared instance of singleton bean
'PostgreSQL'
22:09:57,261 INFO [XmlBeanFactory] Creating shared instance of singleton bean
'Sybase'
22:09:57,271 INFO [SQLExceptionCodesFactory] SQLExceptionCodes loaded: [HSQLDatabase
Engine, Oracle, Sybase SQL Server, Microsoft SQL Server, Informix Dynamic Server,
PostgreSQL, DB2*, MySQL]
22:09:57,271 INFO [SQLExceptionCodesFactory] Looking up default SQLExceptionCodes for
DataSource
22:09:57,351 INFO [DefaultListableBeanFactory] Creating shared instance of
singleton bean 'example11Service'
22:09:57,612 INFO [CollectionFactory] Using JDK 1.4 collections
22:09:58,613 INFO [ContextLoader] Using context class
[org.springframework.web.context.support.XmlWebApplicationContext] for root
WebApplicationContext
22:09:58,613 INFO [ContextLoader] Published root WebApplicationContext
[org.springframework.web.context.support.XmlWebApplicationContext:
displayName=[Root XmlWebApplicationContext]; startup date=[Mon Nov 01 22:09:54
CST 2004]; root of ApplicationContext hierarchy; config
```

```
locations=[/WEB-INF/applicationContext.xml]; ] as ServletContext attribute
with name [interface
org.springframework.web.context.WebApplicationContext.ROOT]
```

5.3.3 架构分析

架构分析，可以从很多角度审视。比如，可以从数据库架构角度、从 J2EE 架构分层角度，从 Spring 配置文件中的 JavaBean 命名空间架构角度等。

大体而言，其架构见图 5-13。

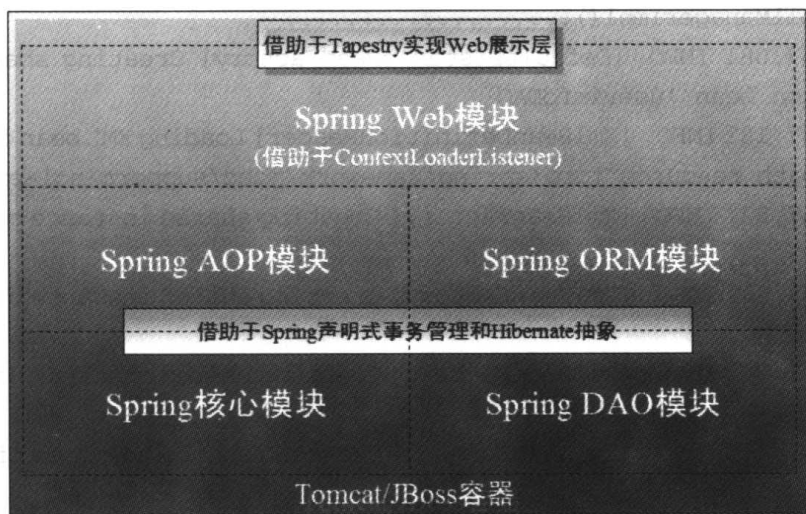


图 5-13 example11 架构

借助于 Tomcat/JBoss 容器，实现对 example11 的部署。借助于 Hibernate 实现数据的 CRUD 操作。借助于 Tapestry 实现 Web 层界面的开发。借助于 Spring 解决上述架构问题。

开发者都已经知道：Spring 配置文件，比如 example11 使用的 applicationContext.xml，是整个实例应用的总线。它将 example11 架构起来。考虑到本书许多地方都引用到本实例的相关代码和配置文件，因此本书将在本章给出其中一些重要的源代码和配置文件。

其中，完整的 applicationContext.xml 文件如下。

```
<?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="dataSource"
        class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName">
            <value>java:/MySqlDS</value>
        </property>
    </bean>

    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
```

```
<property name="dataSource">
    <ref local="dataSource"/>
</property>
<property name="mappingResources">
    <list>
        <value>
            com/opencv/spring/service/hibernate/Interest.hbm.xml
        </value>
        <value>
            com/opencv/spring/service/hibernate/UserInterest.hbm.xml
        </value>
        <value>
            com/opencv/spring/service/hibernate/UserInfo.hbm.xml
        </value>
    </list>
</property>
<property name="hibernateProperties">
    <props>
        <prop key="hibernate.dialect">
            net.sf.hibernate.dialect.MySQLDialect
        </prop>
        <prop key="hibernate.show_sql">
            true
        </prop>
    </props>
</property>
</bean>

<bean id="transactionManager"

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

<bean id="example11ServiceTarget"
    class="com.opencv.spring.service.impl.Example11ManagerImpl">
    <property name="userinfo">
        <ref local="userinfoDAO"/>
    </property>
</bean>

<bean id="example11Service"

class="org.springframework.transaction.interceptor.TransactionProxyFa
```

```
ctoryBean">
    <property name="transactionManager">
        <ref local="transactionManager"/>
    </property>
    <property name="target">
        <ref local="example11ServiceTarget"/>
    </property>
    <property name="transactionAttributes">
        <props>
            <prop key="get *">
                PROPAGATION_REQUIRED,readOnly
            </prop>
            <prop key="set *">
                PROPAGATION_REQUIRED
            </prop>
        </props>
    </property>
</bean>

<bean id="userinfoDAO"
    class="com.openv.spring.service.dao.impl.UserInfoDAO">
    <property name="sessionFactory">
        <ref local="sessionFactory"/>
    </property>
</bean>

</beans>
```

通过 Spring IDE 能够获得 applicationContext.xml 中定义的 JavaBean 命名空间的图形化表示, 如图 5-14 所示。

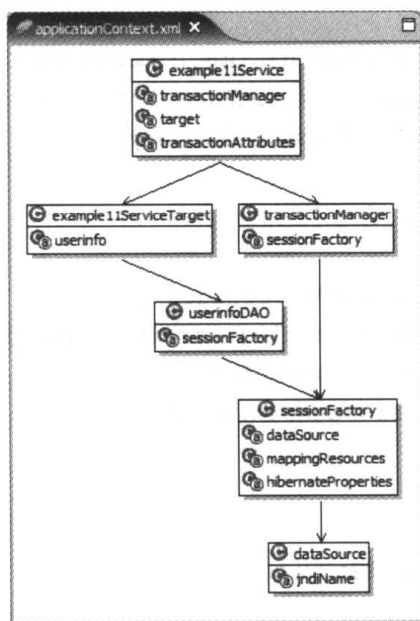


图 5-14 applicationContext.xml 文件的图形化表示

通过图 5-14, 开发者能够更清楚地看到整个 J2EE 应用中定义的 JavaBean 命名空间的架构。随后, 开发者将能够看到在 Web 层, 即 Tapestry 页面中只会涉及到最顶端的“example11Service”名字, 而其他的 JavaBean 类型实例都是通过 Spring IoC 容器和 Spring AOP 完成实例化和装载的。这些实用性极强的功能, 使得应用可以在运行时动态配置。当然, 各部分的具体机理, 本书将在后续章节依次展开。

通过上述内容, 开发者应该知道了 example11 综合实例包括的主要组件, 即它们都体现在 applicationContext.xml 配置文件中。但是, 这对于描述整个应用的业务架构还不是很清晰, 因此本节再次从业务架构的角度, 并给出 UML 类图来阐述实例的设计, 如图 5-15 所示。

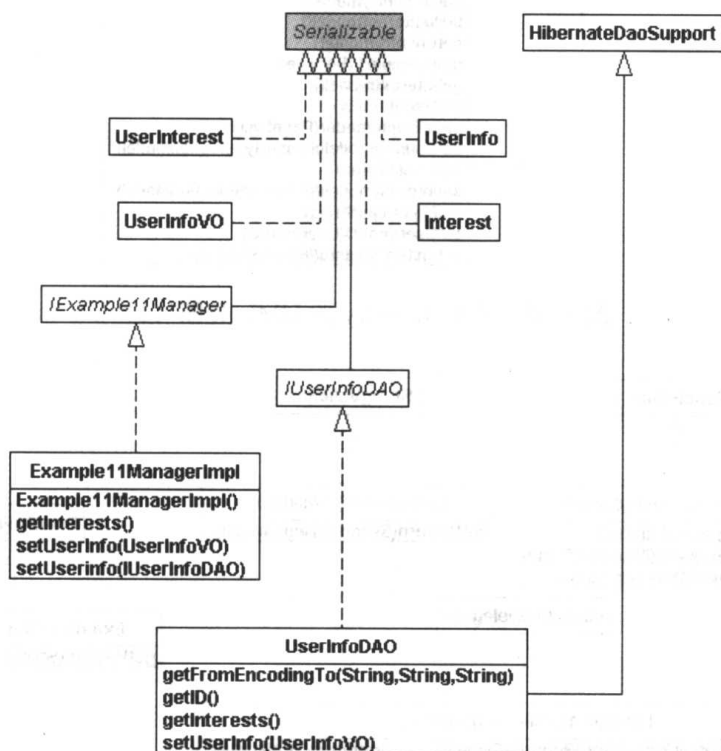


图 5-15 实例的 UML 类图

上述几个类体现了综合实例的业务架构设计与实现。其中, Example11ManagerImpl 实现了业务委派模式, 这使得应用的业务架构更加清晰。Example11ManagerImpl 遵循了针对接口编程的原则, 将业务操作暴露在 IExample11Manager 中。而且, 借助于 UserInfoDAO 和 IUserInfoDAO 实现了对数据访问的 DAO 模式, 从而可以动态修改具体的 DAO 实现。

Tapestry 本身的架构特别优异, 本实例使用到的国际化功能尤其突出。由于本实例使用到的 Web 页面只有一个, 即 Home 页面, 因此基于 Tapestry 框架的前端设计比较简单。图 5-16 和图 5-17 分别给出了相关类的 UML 类图和相关解释, 而具体源代码请参考下节内容。

其中, Home 类是 example11 应用的主界面对应的类。通过实现 PageRenderListener 接口, 开发者能够做一些初始化的工作, 比如装载 RDBMS 中的数据。本书在第三部分将详细阐述。

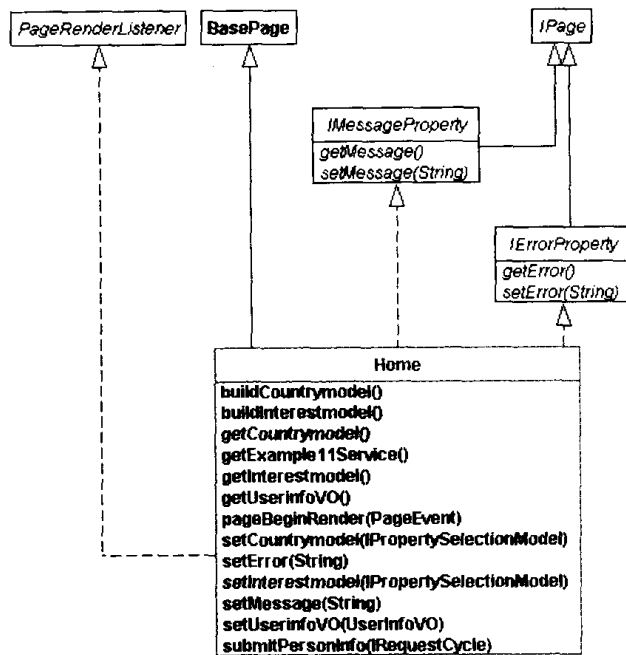


图 5-16 基于 Tapestry 的 UML 类图 (1)

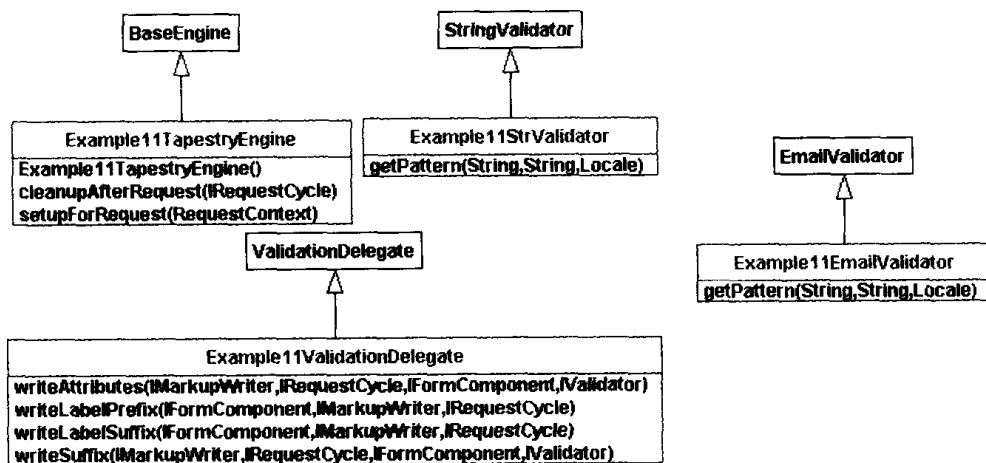


图 5-17 基于 Tapestry 的 UML 类图 (2)

其中，借助于 **Example11TapestryEngine** 能够实现 Spring 上下文的集成、使用。借助于 **Example11StrValidator** (**Example11EmailValidator**) 能够实现对字符串 (E-mail) 类型的数据在客户端和服务端同时对它们进行校验。借助于 **Example11ValidationDelegate** 能够实现错误信息、提示信息的展现修饰，即以更友好的方式实现这类信息的展示。

至于 **example11** 的具体实现，本书将在后续章节一一给出。

5.4 小 结

本章详细讲述了构成 Spring 框架的各个组成构件。这对于分析和掌握 Spring 框架的架构很有意义。当然，对于掌握 Spring 框架的使用也很有帮助。其次，本章还阐述了一综合

实例，即基于 Spring/Tapestry/Hibernate 构建的 Web 应用实例。这对于掌握 Spring 的使用特别有意义。

Spring 本身的分层架构很突出。无论开发者使用 Spring 开发何种应用，可以选择性地使用 Spring 不同模块的功能，比如 Spring IoC 和 AOP 模块。再比如，在选择不同 Web 视图技术构建应用时，无论开发者使用何种视图技术，都能够使用到 Spring IoC 容器和 AOP 实现。而且，Spring 本身的扩展性很好，基于 Spring 的 Acegi（具体内容见第 17 章）验证了这一点。

本书剩下的章节在具体研究基于 Spring 开发 J2EE 应用的过程中将围绕该综合实例展开。其中，它们将重点给出相关 J2EE/Spring 背景知识的介绍和实例的进一步展开。

开发专家之 Sun ONE

第二部分 Spring 应用开发

借助于 Spring 开发 J2EE 应用才是我们使用它的最根本驱动力。因此，这部分内容介绍如何借助于 Spring 开发 J2EE 应用。从简化 JNDI 的操作（第 6 章），到事务服务（第 7 章），到 JMS 服务（第 8 章），到 JavaMail 服务（第 9 章），到 EJB 组件技术（第 10 章），到持久化服务（第 11 章），到任务调度服务（第 12 章），到远程服务（第 13 章），这些都是这部分演绎的内容，它们一起构成了 Spring 提供的 J2EE 服务抽象。

无论何时，Spring 提供的 J2EE 服务抽象往往是开发者需要熟悉的、最为重要的知识点，否则很难体会到 Spring 的真正魅力。

记住，这些服务抽象是架构在 Spring IoC 和 Spring AOP 基础之上。如果您还未阅读第一部分内容，则赶紧去吧！当然，开发者可以依据自身的兴趣，选择性地阅读各章内容，因为它们之间是松“耦合”的。

注意，在您阅读完本部分内容后，请再次回到第一部分内容中，这将加深对 Spring 的掌握程度！

第 6 章 命名服务——JNDI

从现在开始，让我们一起步入 Spring 开发 Java/J2EE 应用的殿堂吧！来一起研究实用的 Spring 到底为开发者简化了哪些工作。Web 容器和 J2EE 应用服务器提供了大量的企业级服务，而开发 J2EE 应用在很多时候都需要在 J2EE API 级开始，这不利于提高开发效率。对于 J2EE 应用服务器而言，它有效地解决了平台对应用提供的服务。但是，对于开发者而言，如何在应用中有效地使用这些平台服务。就目前而言，这块市场还没有很好的解决方案。Spring 就是为了解决目前基于 J2EE 开发应用存在这样的问题而出现的。可以预测，在简化 J2EE 开发模型后，比如 J2EE 5.0 的推出，类似于轻量级 Spring 的架构级框架会越来越多，而且功能会越来越丰富。很多时候，如何简化 J2EE 开发模型是经常摆在 J2EE 相关角色（包括，J2EE 规范制定者、J2EE 开发工具提供商、J2EE 开发人员等）面前的问题。

通过研究 Spring 提供的具体实现，便能够隐约看到 J2EE 的未来发展趋势。这是令人兴奋的框架。

开发者通过使用 Spring 对 J2EE API 提供的服务抽象，再加上 Spring IoC 容器和 Spring AOP 框架的集成支持，能够快速、稳健地开发出优秀的企业级应用。最为重要的一点是 Spring 提供的各种 J2EE API 服务抽象都是如此的一致、合理，从而大大降低开发者的学习曲线。

6.1 背 景

JNDI，即 Java 命名目录服务。它类似于 DNS 一样，能够将对象赋予有意义的名字。在 DNS 中，通过逻辑名能够实现对 IP 地址的映射。在 JNDI 中，通过预先绑定的名字能够找到目标服务或者对象。比如，在 example11 中通过“java:/MySqlDS”名字能够获得 DataSource 对象。这些都是通过 Java 命名系统获得的。JNDI 在 Java/J2EE 中起到了很重要的作用，正如 DNS 在当今的互联网中的地位一样，而且会越来越重要。比如，Web 服务的兴起、SOA 架构的兴起，会使得 JNDI 在应用中扮演关键角色。

将应用对象存储在某注册中心，从而改善 Java 应用，这就是 JNDI。比如，开发者可以将数据源存储在提供商提供的命名与目录服务器中；JBoss 4.x 将 Hibernate 的 SessionFactory 存储在命名与目录服务器中；通过 JNDI 能够获得 JTA 事务管理器（存储在 J2EE 应用服务器提供的命名与目录服务器中）。尽管借助于 Spring 配置文件，也可以配置对象，但是相比应用服务器提供的对象而言，显得有些单薄，比如高端应用服务器提供的 JTA 事务支持（实现了事务容错、事务恢复等功能）。因此，引入 JNDI 很有必要。

那么，Spring 对 JNDI 的支持又如何呢？它通过提供 Spring JNDI 抽象，使得开发者能够将 JNDI 查找操作定义在配置文件中，再也不用在 Java 代码中硬编码了。在 Spring 配置

文件定义了 JNDI 后, 其他的 JavaBean 能够直接使用它, 从而提高开发效率。最为重要的一点是, 这些 JNDI 依赖都是通过 IoC 容器注入的, 因此开发者再也不用关注具体对象是从哪里来的。

6.2 Spring 对 JNDI 提供的支持

Spring 框架专门提供了 `org.springframework.jndi` 包, 以简化对 JNDI 的使用。目前, `jndi` 包一共存在 1 个接口和 8 个类。

其中, `JndiTemplateEditor` 类继承于 `PropertyEditorSupport` 类, 供实现 `JndiTemplate` 对象的属性编辑器使用, 即开发者能够通过字符串方式实现对 `JndiTemplate` 值的设置 (或者在 IDE 中编辑它)。开发者一般不需要涉及到它。

其次, 图 6-1 给出了 `org.springframework.jndi` 包中主要类的类图。

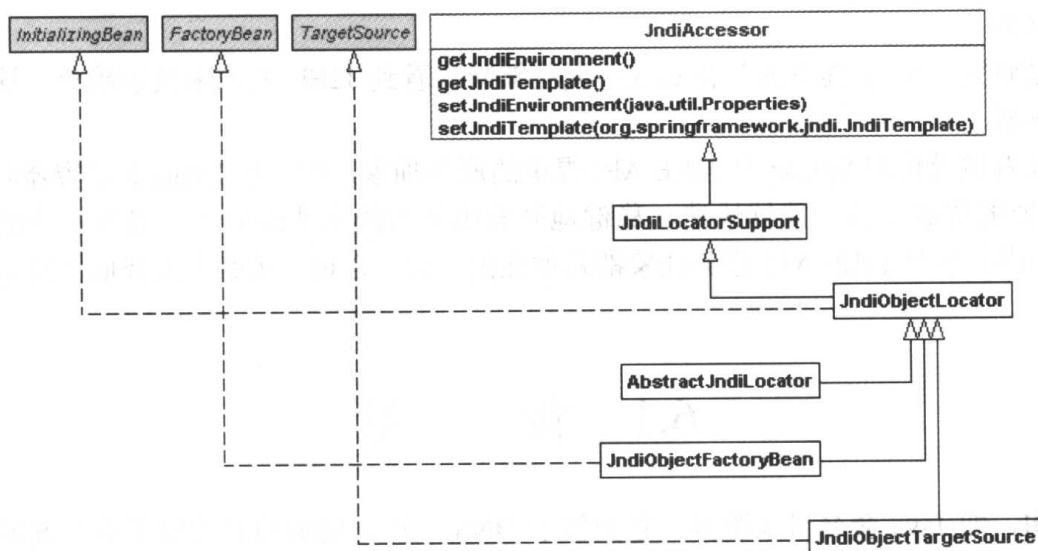


图 6-1 操作 JNDI 主要类的类图

其中, `JndiAccessor` 使用了 `JndiTemplate` 类, 见图 6-2 所示。

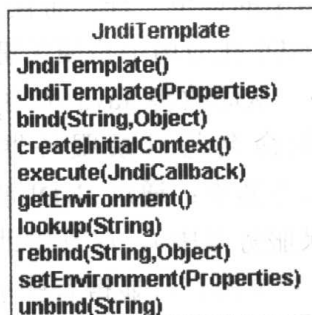


图 6-2 JndiTemplate 类图

借助于 Spring 框架, 为实现对 JNDI 的访问, 开发者可以通过如下 3 种方式。

- 开发者实现 `AbstractJndiLocator` 抽象类

- 使用 JndiObjectFactoryBean
- 使用 JndiObjectTargetSource

直接使用第 2、3 种方式最简单。如果开发者希望使用第 1 种方式，则也很简单，只需要实现如下方法。

```
protected abstract void located(Object jndiObject);
```

本书将重点阐述第 2、3 种方式。

6.2.1 JndiObjectFactoryBean

其中，example11 使用了 JndiObjectFactoryBean，相应的配置片断如下。

```
<bean id="dataSource"
      class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
    <value>java:/MySqlDS</value>
  </property>
</bean>
```

这里的 dataSource 主要是为了获得数据源而存在的。为了展示 JndiObjectFactoryBean 的“功力”，这里换一个 jndiName（数据源的 JNDI 名在 JBoss 中只是运行 JBoss 的 JVM 可见，然而这里的演示需要在 JBoss 外部进行。）。好了，开始 example12 吧！

通过 JBoss 的 JMX 控制台应用（如图 6-3），开发者能够获得全局 JNDI 名。

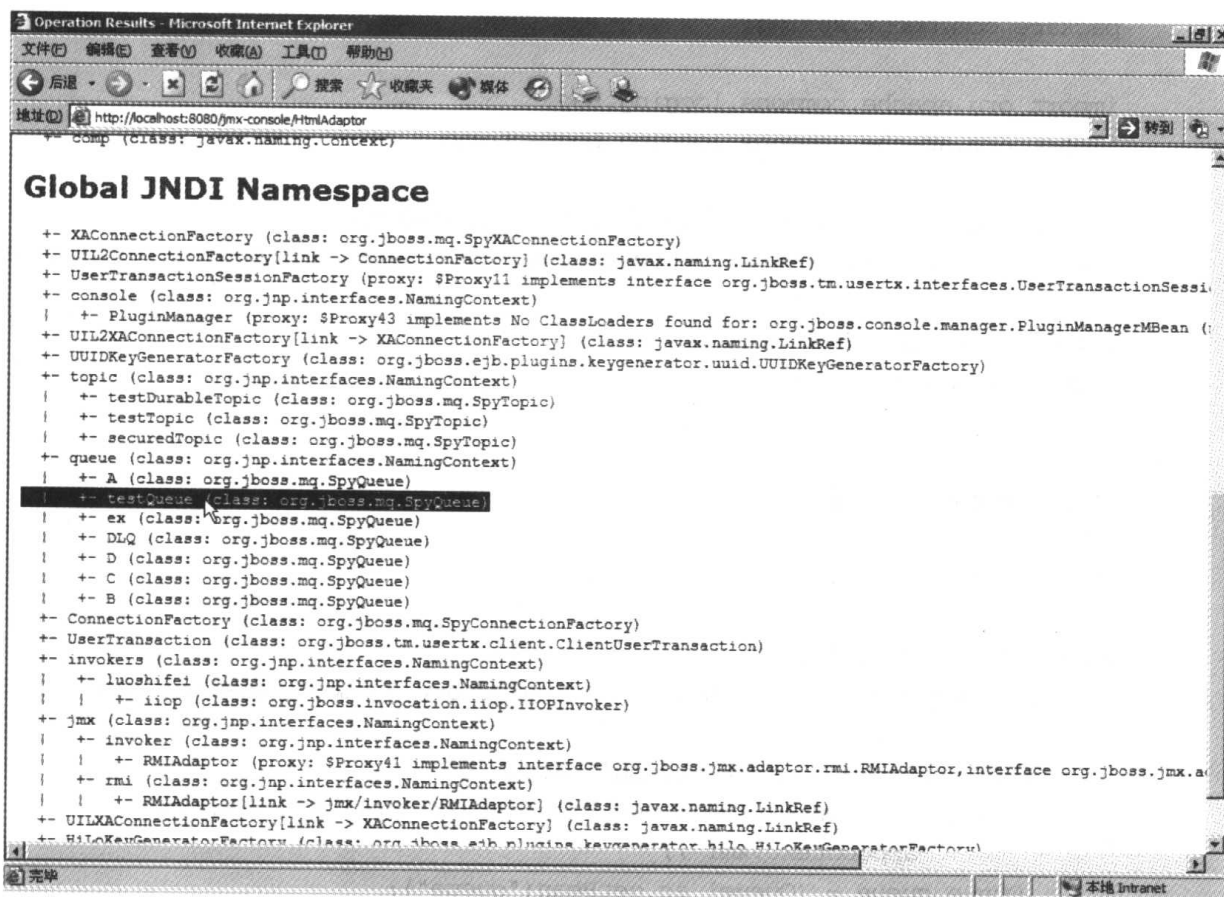


图 6-3 获得“queue/testQueue”JNDI 名

这里以消息队列“queue/testQueue”为例。首先，开发者需要给出 appcontext.xml 文件内容。其中，通过 JndiObjectFactoryBean 的 jndiName 属性能够设置目标 JNDI 名。在本实例中，“queue/testQueue”是 JNDI 名。JndiObjectFactoryBean 是工厂 JavaBean，它能够创建相应的对象类型（比如 Queue 对象）。

```
<?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="queue"
        class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName">
            <value>queue/testQueue</value>
        </property>
    </bean>

</beans>
```

然后，QueueTest.java 客户应用代码如下。在通过 ApplicationContext 获得 JavaBean 实例后，需要将它造型为 Queue 对象，从而可以根据返回的 queue 对象获得 JMS 目的地的名字。

```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

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

import javax.jms.JMSEException;
import javax.jms.Queue;

/**
 * QueueTest 客户应用
 *
 * @author luoshifei
 */
public class QueueTest {
    protected static final Log log = LogFactory.getLog(QueueTest.class);

    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "appcontext.xml");
        Queue queue = (Queue) ac.getBean("queue");

        try {
```

```

        log.info("消息队列名称: " + queue.getQueueName());
    } catch (JMSException ex) {
        log.error("消息异常", ex);
    }
}
}
}

```

另外, 开发者不要忘记将 `jndi.properties` 属性文件放置在 `src` 目录下 (如果借助于 `JndiTemplate` 实现 `InitialContext` 环境变量的配置, 则不需要 `jndi.properties`。详情见 6.2.3 节内容), 其具体内容如下。

```

java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost:1099
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces

```

最后, 运行 `QueueTest` (`Ant build.xml` 中的 `run` 任务)。

Buildfile: D:\workspace\example12\build.xml

compile:

run:

```

[java] 2004-11-6 21:34:08 org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
[java] 信息: Loading XML bean definitions from class path resource
[appcontext.xml]
[java] 2004-11-6 21:34:08 org.springframework.context.support.
AbstractXmlApplicationContext refreshBeanFactory
[java] 信息: Bean factory for application context [org.springframework.
context.support.ClassPathXmlApplicationContext;hashCode=8795033]:
org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [queue]; root of BeanFactory hierarchy
[java] 2004-11-6 21:34:08 org.springframework.context.support.
AbstractApplicationContext refresh
[java] 信息: 1 beans defined in application context [org.springframework.
context.support.ClassPathXmlApplicationContext;hashCode=8795033]
[java] 2004-11-6 21:34:08 org.springframework.context.support.
AbstractApplicationContext initMessageSource
[java] 信息: No message source found for context [org.springframework.
context.support.ClassPathXmlApplicationContext;hashCode=8795033]: using
empty default
[java] 2004-11-6 21:34:08 org.springframework.context.support.
AbstractApplicationContext initApplicationEventMulticaster
[java] 信息: No ApplicationEventMulticaster found for context
[org.springframework.context.support.ClassPathXmlApplicationContext;hashC
ode=8795033]: using default
[java] 2004-11-6 21:34:08 org.springframework.context.support.
AbstractApplicationContext refreshListeners
[java] 信息: Refreshing listeners
[java] 2004-11-6 21:34:08 org.springframework.beans.factory.support.
DefaultListableBeanFactory preInstantiateSingletons
[java] 信息: Pre-instantiating singletons in factory [org.springframework.

```

```
beans.factory.support.DefaultListableBeanFactory defining beans [queue];
root of BeanFactory hierarchy]
[java] 2004-11-6 21:34:08 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'queue'
[java] 2004-11-6 21:34:09 org.springframework.jndi.JndiLocatorSupport
lookup
[java] 信息: Located object with JNDI name [queue/testQueue]:
value=[QUEUE.testQueue]
[java] 2004-11-6 21:34:09 com.openv.spring.QueueTest main
[java] 信息: 消息队列名称: testQueue
BUILD SUCCESSFUL
Total time: 1 second
```

可以看出, testQueue 队列名成功找到。如果不借助于 Spring IoC 容器, 开发者应该也可以写出类似代码:

```
public static void main(String[] args) {
    try {
        InitialContext initialContext = new InitialContext();
        Queue queue = (Queue) initialContext.lookup("queue/testQueue");
        log.info("消息队列名称: " + queue.getQueueName());
    } catch (JMSEException ex) {
        log.error("消息异常", ex);
    } catch (NamingException ex) {
        log.error("命名异常", ex);
    }
}
```

至此, 我们可以一同来比较一下两种方法的优劣了。如果不借助于 Spring IoC, 则需要手工创建 InitialContext 对象, 而且还需要处理 NamingException 异常。更不好的地方在于, 目标 queue/testQueue 队列名硬编码 (hard-code) 在 Java 代码中。因此, 对于这种简单的 JNDI 查找, 都能够看出 Spring IoC 的优势了。

反之, 借助于 Spring 提供的 JNDI 服务抽象, 开发者不用手工创建和关闭 InitialContext、不用将查找的 JNDI 名在 Java 代码中硬编码、不用处理 NamingException 异常、不用配置 jndi.properties 属性文件。

JndiObjectFactoryBean 主要用于产品部署场合。依据作者的开发经验, 最好直接使用这种方式, 即借助于 JndiObjectFactoryBean 获得 JNDI 对象。

6.2.2 JndiObjectTargetSource

直接借助于 JndiObjectFactoryBean, 开发者能够通过 JNDI 名获得目标对象, 比如 “queue/testQueue”。Spring 框架的 jndi 包中还提供了 JndiObjectTargetSource 类。本节将结合 example13 展开对 JndiObjectTargetSource 的研究。

首先, 还是配置 appcontext.xml 文件。

```
<?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="queueTarget"
        class="org.springframework.jndi.JndiObjectTargetSource">
        <property name="jndiName">
            <value>queue/testQueue</value>
        </property>
    </bean>

    <bean id="queueFactory"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <value>javax.jms.Queue</value>
        </property>
        <property name="targetSource">
            <ref bean="queueTarget"/>
        </property>
    </bean>

</beans>
```

开发者是否注意到, appcontext.xml 中使用了 ProxyFactoryBean 类。这需要 Spring AOP 模块的支持, 即需要 spring-aop.jar 库文件。

然后, 需要开发 QueueTest 客户端应用。

```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

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

import javax.jms.JMSEException;
import javax.jms.Queue;

/**
 * QueueTest 客户应用
 *
 * @author luoshifei
 */
public class QueueTest {
    protected static final Log log = LogFactory.getLog(QueueTest.class);
```

```
public static void main(String[] args) {
    ApplicationContext ac = new ClassPathXmlApplicationContext(
        "appcontext.xml");

    Queue queue = (Queue) ac.getBean("queueFactory");

    try {
        log.info("消息队列名称: " + queue.getQueueName());
    } catch (JMSEException ex) {
        log.error("消息异常", ex);
    }
}
```

其实，主要在 JavaBean 的 id 上同 example12 有区别外，其他都一样。具体的执行结果如下（Ant build.xml 中的 run 任务）。

```
Buildfile: D:\workspace\example13\build.xml
compile:
run:
    [java] 2004-11-6 22:00:39 org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
    [java] 信息: Loading XML bean definitions from class path resource
[appcontext.xml]
    [java] 2004-11-6 22:00:40 org.springframework.context.support.
AbstractXmlApplicationContext refreshBeanFactory
    [java] 信息: Bean factory for application context [org.springframework.
context.support.ClassPathXmlApplicationContext;hashCode=8795033]:
org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [queueTarget,queueFactory]; root of BeanFactory hierarchy
    [java] 2004-11-6 22:00:40 org.springframework.context.support.
AbstractApplicationContext refresh
    [java] 信息: 2 beans defined in application context [org.springframework.
context.support.ClassPathXmlApplicationContext;hashCode=8795033]
    [java] 2004-11-6 22:00:40 org.springframework.context.support.
AbstractApplicationContext initMessageSource
    [java] 信息: No message source found for context [org.springframework.
context.support.ClassPathXmlApplicationContext;hashCode=8795033]: using
empty default
    [java] 2004-11-6 22:00:40 org.springframework.context.support.
AbstractApplicationContext initApplicationEventMulticaster
    [java] 信息: No ApplicationEventMulticaster found for context
[org.springframework.context.support.ClassPathXmlApplicationContext;hashC
ode=8795033]: using default
    [java] 2004-11-6 22:00:40 org.springframework.context.support.
AbstractApplicationContext refreshListeners
    [java] 信息: Refreshing listeners
```

```
[java] 2004-11-6 22:00:40 org.springframework.beans.factory.support.
DefaultListableBeanFactory preInstantiateSingletons
[java] 信息: Pre-instantiating singletons in factory [org.springframework.
beans.factory.support.DefaultListableBeanFactory defining beans [queueTarget,
queueFactory]; root of BeanFactory hierarchy]
[java] 2004-11-6 22:00:40 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'queueTarget'
[java] 2004-11-6 22:00:40 org.springframework.jndi.JndiLocatorSupport lookup
[java] 信息: Located object with JNDI name [queue/testQueue]:
value=[QUEUE.testQueue]
[java] 2004-11-6 22:00:40 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'queueFactory'
[java] 2004-11-6 22:00:40 org.springframework.core.CollectionFactory<clinit>
[java] 信息: Using JDK 1.4 collections
[java] 2004-11-6 22:00:40 com.openv.spring.QueueTest main
[java] 信息: 消息队列名称: testQueue
BUILD SUCCESSFUL
Total time: 1 second
```

JndiObjectTargetSource 主要用于产品开发场合。

6.2.3 JndiTemplate

在实际企业级 Java 应用场合,经常要使用到不同的 J2EE 中间件,比如 WebSphere MQ、JOTM、JORAM 等。访问它们的 InitialContext 环境属性都很不一样,因此如果仅仅将 jndi.properties 放置在应用执行的 classpath 中,则显然不能够满足实际应用需求。此时,借助于 JndiTemplate 能够解决这种应用问题。

比如,借助于 JndiTemplate,开发者可以改造 example12 中的 appcontext.xml。具体如下。

```
<?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="queue"
        class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName">
            <value>queue/testQueue</value>
        </property>
        <property name="jndiTemplate">
            <ref local="jndiTemplate"></ref>
        </property>
    </bean>
```

```
<bean id="jndiTemplate"
      class="org.springframework.jndi.JndiTemplate">
  <property name="environment">
    <props>
      <prop key="java.naming.factory.initial">
        org.jnp.interfaces.NamingContextFactory
      </prop>
      <prop key="java.naming.provider.url">
        jnp://localhost:1099
      </prop>
      <prop key="java.naming.factory.url.pkgs">
        org.jboss.naming:org.jnp.interfaces
      </prop>
    </props>
  </property>
</bean>
```

```
</beans>
```

因此，相应的运行结果如下。

Buildfile: D:\workspace\example12\build.xml

compile:

run:

```
[java] 2004-12-12 14:08:28 org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
```

```
[java] 信息: Loading XML bean definitions from class path resource
[appcontext.xml]
```

```
[java] 2004-12-12 14:08:28 org.springframework.context.support.
AbstractXmlApplicationContext refreshBeanFactory
```

```
[java] 信息: Bean factory for application context [org.springframework.
context.support.ClassPathXmlApplicationContext;hashCode=8795033]:
org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [queue,jndiTemplate]; root of BeanFactory hierarchy
```

```
[java] 2004-12-12 14:08:28 org.springframework.context.support.
AbstractApplicationContext refresh
```

```
[java] 信息: 2 beans defined in application context [org.springframework.
context.support.ClassPathXmlApplicationContext;hashCode=8795033]
```

```
[java] 2004-12-12 14:08:28 org.springframework.context.support.
AbstractApplicationContext initMessageSource
```

```
[java] 信息: No message source found for context [org.springframework.
context.support.ClassPathXmlApplicationContext;hashCode=8795033]: using
empty default
```

```
[java] 2004-12-12 14:08:28 org.springframework.context.support.
AbstractApplicationContext initApplicationEventMulticaster
```

```
[java] 信息: No ApplicationEventMulticaster found for context
[org.springframework.context.support.ClassPathXmlApplicationContext;hashC
ode=8795033]: using default
```



```

[java] 2004-12-12 14:08:28 org.springframework.context.support.
AbstractApplicationContext refreshListeners
[java] 信息: Refreshing listeners
[java] 2004-12-12 14:08:28 org.springframework.beans.factory.support.
DefaultListableBeanFactory preInstantiateSingletons
[java] 信息: Pre-instantiating singletons in factory [org.springframework.
beans.factory.support.DefaultListableBeanFactory defining beans [queue,
jndiTemplate]; root of BeanFactory hierarchy]
[java] 2004-12-12 14:08:28 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'queue'
[java] 2004-12-12 14:08:28 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'jndiTemplate'
[java] 2004-12-12 14:08:29 org.springframework.jndi.JndiLocatorSupport lookup
[java] 信息: Located object with JNDI name [queue/testQueue]:
value=[QUEUE.testQueue]
[java] 2004-12-12 14:08:29 com.openv.spring.QueueTest main
[java] 信息: 消息队列名称: testQueue
BUILD SUCCESSFUL
Total time: 2 seconds

```

开发者可以看出，Spring BeanFactory 创建了 jndiTemplate 单例，供整个 Spring 应用使用。因此，开发者可以针对不同的 JNDI 上下文，配置不同的 JndiTemplate。

再来看如何改造 example13。其中，appcontext.xml 如下。

```

<?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="queueTarget"
        class="org.springframework.jndi.JndiObjectTargetSource">
        <property name="jndiName">
            <value>queue/testQueue</value>
        </property>
        <property name="jndiTemplate">
            <ref local="jndiTemplate"></ref>
        </property>
    </bean>

    <bean id="queueFactory"
        class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <value>javax.jms.Queue</value>
        </property>
        <property name="targetSource">

```

```
<ref bean="queueTarget"/>
</property>
</bean>

<bean id="jndiTemplate"
class="org.springframework.jndi.JndiTemplate">
  <property name="environment">
    <props>
      <prop key="java.naming.factory.initial">
        org.jnp.interfaces.NamingContextFactory
      </prop>
      <prop key="java.naming.provider.url">
        jnp://localhost:1099
      </prop>
      <prop key="java.naming.factory.url.pkgs">
        org.jboss.naming:org.jnp.interfaces
      </prop>
    </props>
  </property>
</bean>

</beans>
```

相应的运行结果如下，这同期望的一致。

Buildfile: D:\workspace\example13\build.xml

compile:

run:

```
[java] 2004-12-12 14:06:27 org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
[java] 信息: Loading XML bean definitions from class path resource
[appcontext.xml]
[java] 2004-12-12 14:06:28 org.springframework.context.support.
AbstractXmlApplicationContext refreshBeanFactory
[java] 信息: Bean factory for application context [org.springframework.
context.support.ClassPathXmlApplicationContext;hashCode=8795033]: org.
springframework.beans.factory.support.DefaultListableBeanFactory defining
beans [queueTarget,queueFactory,jndiTemplate]; root of BeanFactory hierarchy
[java] 2004-12-12 14:06:28 org.springframework.context.support.
AbstractApplicationContext refresh
[java] 信息: 3 beans defined in application context
[org.springframework.context.support.ClassPathXmlApplicationContext;hashC
ode=8795033]
[java] 2004-12-12 14:06:28 org.springframework.context.support.
AbstractApplicationContext initMessageSource
[java] 信息: No message source found for context [org.springframework.
context.support.ClassPathXmlApplicationContext;hashCode=8795033]: using
empty default
```

```
[java] 2004-12-12 14:06:28 org.springframework.context.support.  
AbstractApplicationContext initApplicationEventMulticaster  
[java] 信息: No ApplicationEventMulticaster found for context  
[org.springframework.context.support.ClassPathXmlApplicationContext;hashC  
ode=8795033]: using default  
[java] 2004-12-12 14:06:28 org.springframework.context.support.  
AbstractApplicationContext refreshListeners  
[java] 信息: Refreshing listeners  
[java] 2004-12-12 14:06:28 org.springframework.beans.factory.support.  
DefaultListableBeanFactory preInstantiateSingletons  
[java] 信息: Pre-instantiating singletons in factory  
[org.springframework.beans.factory.support.DefaultListableBeanFactory  
defining beans [queueTarget,queueFactory,jndiTemplate]; root of BeanFactory  
hierarchy]  
[java] 2004-12-12 14:06:28 org.springframework.beans.factory.support.  
AbstractBeanFactory getBean  
[java] 信息: Creating shared instance of singleton bean 'queueTarget'  
[java] 2004-12-12 14:06:28 org.springframework.beans.factory.support.  
AbstractBeanFactory getBean  
[java] 信息: Creating shared instance of singleton bean 'jndiTemplate'  
[java] 2004-12-12 14:06:28 org.springframework.jndi.JndiLocatorSupport  
lookup  
[java] 信息: Located object with JNDI name [queue/testQueue]:  
value=[QUEUE.testQueue]  
[java] 2004-12-12 14:06:28 org.springframework.beans.factory.support.  
AbstractBeanFactory getBean  
[java] 信息: Creating shared instance of singleton bean 'queueFactory'  
[java] 2004-12-12 14:06:28 org.springframework.core.CollectionFactory  
<clinit>  
[java] 信息: Using JDK 1.4 collections  
[java] 2004-12-12 14:06:28 com.openv.spring.QueueTest main  
[java] 信息: 消息队列名称: testQueue  
BUILD SUCCESSFUL  
Total time: 3 seconds
```

6.2.4 JndiCallback

Spring 框架的 `jndi` 包中提供的惟一接口是 `JndiCallback`。开发者可以实现 `JndiCallback` 接口，即实现如下方法。

```
Object doInContext(Context ctx) throws NamingException
```

因此，借助于 `JndiCallback` 接口实现，开发者能够实现对 JNDI 上下文的操作，比如 lookup 某 JNDI 名、注册某 JNDI 名。

6.3 小 结

借助于 Spring IoC 和 Spring AOP，使得操作 JNDI 的过程很简单。与此同时，Spring 还专门提供了 jndi 包，以简化对 JNDI 的操作。因此，Spring 是高效的、实用的。

对于大部分 J2EE 应用而言，通常只需要完成 Java 命名的查找工作，因此本章介绍的实例是够用的，也是实用的。

其实，本书的第二部分都能够体现 Spring 的实用和高效。因此，开发者在阅读本书过程中，尤其是第二部分内容，可以采用“查字典”的方式进行。

第 7 章 事务服务——JTA

JTA 作为本章的标题有点“喧宾夺主”的味道，但是本书还是将 JTA 摆在标题最显眼的地方，因为 JTA 在 J2EE 中的地位太重要了。

Spring 为事务管理提供了一流的支持。它同时支持编程式和声明式事务。为实现健壮的企业级应用，事务能够担当重要的作用。无论开发者使用编程式事务，还是声明式事务，在开发 Spring 使能应用过程中，开发者都不需要同具体的事务管理实现进行交互。

事务管理抽象是 Spring 提供的最为重要的一种抽象。秉承 Spring 的设计原则，对于事务管理而言，Spring 的事务抽象具有如下几方面的优势：

- 对于采用手工控制事务，即程序控制事务的编程方式而言，Spring 提供的事务抽象易于使用。
- 无论底层的事务 API 是什么，Spring 都能够提供一致的编程模型。比如，对于提供分布式事务支持的 JTA、JDBC、Hibernate 等而言，基于 Spring 的应用代码都是一致的，而没有专属于某事务 API 的实现。其中，特定的事务 API 都可以通过 Spring 配置文件屏蔽掉。
- Spring 支持以声明方式管理事务。这其中主要依赖于 Spring AOP 模块提供的功能。因此，Spring AOP 在 Spring 事务抽象服务中起了重要的作用。
- 能够同 Spring 的 DAO 抽象进行集成。
- 在不同事务服务间切换，只会涉及到 Spring 配置文件的修改，而不会涉及到代码的修改。

因此，Spring 提供的事务服务很有价值，很值得开发者使用和研究。

7.1 背 景

通常，对于事务管理而言，存在两种事务类型：局部（本地，local）事务和全局（global）事务。对于本地事务而言，只涉及到单个事务性（可恢复）资源，通常都是 JDBC 级的事务。对于全局事务而言，通常都涉及到两个以上的事务性资源，比如两个不同的数据库、数据库和 JMS 服务器。如果应用只涉及到单个事务性资源，则无需借助于 J2EE 应用服务器提供的 JTA 能力，而只需要借助于本地事务即可解决事务管理服务方面的问题。如果应用涉及到两个以上的事务性资源，则需要 J2EE 应用服务器的 JTA 能力。

从性能上考虑，全局事务是“致命”的。能够不用全局事务，则尽量不用。但是对于企业级应用而言，经常不可避免地碰到这种场景。因此，对于这种问题，开发者也不可回避。JTA 在全局事务中起了事务管理器的作用。然而，为在应用代码中使用 JTA 功能，必须经过 JNDI 获得对 JTA 的引用，继而应用代码需要处理大量的受查异常。另外，对于 Web 容器而言，通常都不提供 JTA 的实现，即不支持全局事务，除非外挂 JTA 事务管理器实现，

比如 Open Source JOTM。对于 J2EE 应用服务器而言，比如 JBoss、WebLogic、WebSphere、Oracle Application Server、Sun Java System Application Server 等，都提供了对 JTA 的支持。当然，不同应用服务器对 JTA 的支持程度也不尽相同。

从易用性角度考虑，本地事务更容易使用。但是，它不能够保证同时处理多个事务性资源的正确性。通常，事务都具备 ACID 的特性。

- A，即原子性（Atomicity）。事务必须成功提交，或者回滚。在事务中，可能含有多项事务性资源，如果其中的一项资源操作失败，则事务必须回滚。如果其中的所有资源操作都成功完成，则事务将进入提交状态，继而保证事务的原子性。因此，对于事务而言，其中的各项资源的操作成功与否对事务的完成状态具有决定性的作用。
- C，即一致性（Consistency）。事务操作将导致一致性的结果，即使得其操作的资源处于一致性状态。无论是事务成功提交，还是回滚，都必须保证资源状态的一致性。当然，这种一致性在很大程度上需要由应用保证，因此 C 同 A、I、D 存在很大区别。
- I，即隔离性（Isolation）。在事务执行过程中，其操作的资源的状态信息不能够被其他事务操作到。因此，不同事务不能够并发对同一数据资源进行读取或写入操作，否则保证不了隔离性。在对 RDBMS 操作过程中，为保证隔离性，经常需要对表、列进行锁操作。
- D，即持久性（Durability）。一旦事务成功提交，则事务所导致的结果应该是持久的，即使系统瘫痪都需要保证持久性。当然，如果持久化存储器瘫痪（比如，硬盘），则持久性也没有办法保证。

ACID，是所有事务管理过程中必须保证的特性。

对于开发过 EJB 组件的开发者而言，应该都熟悉 EJB 提供的容器管理事务（Container Managed Transaction, CMT），即通过在 `ejb-jar.xml` 文件中进行事务声明，而不需要应用代码的介入，这种方式存在优势。其一，消除了对 JNDI 查找的要求（不考虑 EJB 组件本身）；其二，无需在应用代码中显式地控制事务，尽管也可以实现这种方式（针对 Bean 管理事务而言）。但是，CMT 要求有 EJB 容器支持，而且只是对 EJB 组件有效。如果实际应用中，需要对 POJO 生效类似 CMT 的功能，该怎么办？

Spring 能够解决上述问题。Spring 承诺：在任何场景中，使用一致的编程模型，而不会影响对事务的使用；或者说，在使用事务的地方，不会影响编程的模式。尽管 Spring 同时提供了对声明式和编程式的事务管理支持，本书还是推荐使用声明式的事务管理（本书将重点阐述 Spring 中对声明式事务的支持）。当然，在大部分场合，编程式的事务管理更具灵活性。不过，在引入 Spring AOP 后，情况未必是这样。

7.2 Spring 对事务管理提供的支持

类似于 EJB，Spring 提供的事务管理功能也很丰富，而且 Spring 提供的事务服务抽象功能更完善、强大。Spring 同时支持如下两种事务编程模型：声明式和编程式。

对于声明式事务而言, Spring 支持各种事务管理器, 而不像 EJB。EJB 仅仅支持 JTA (Java Transaction API)。注意, JTA 是 EJB (对于容器管理事务而言) 提供事务支持的惟一事务实现。借助于 Spring AOP 模块, 能够实现 Spring 提供的声明式事务支持。在 J2EE 平台中, 事务往往作为企业级服务看待, 因此它是系统级的; 企业应用在完成业务逻辑操作时, 需要借助于事务服务, 因此将 Spring 提供的事务抽象作为 AOP 中的 Aspect 看待是很合理的事情, 即在 Spring 中提供了事务方面 (Aspect), 这同 JBoss AOP 提供的事务方面类似。在 Spring 事务服务抽象中, 它提供了 TransactionProxyFactoryBean 类, 供实现声明式事务使用。

对于编程式事务而言, Spring 也支持各种事务管理器。而且, 它比声明式事务更灵活。当然, 对于 Bean 管理事务的 EJB 而言, 它也支持各种事务管理器实现。如果在 Spring 使用应用中, 仅仅使用到单个的事务性资源, 则开发者可以不使用 JTA 实现。比如, 可以直接使用 JDBC (DataSourceTransactionManager)、Hibernate、JDO、OJB、JMS、JTA。为支持全局事务, 即如果事务中存在 2 个以上的事务性资源, 则必须借助于第三方 JTA 实现, 比如 Open Source JOTM。

请开发者注意, Spring 提供的事务管理器仅仅是对现有的事务实现 API (比如, Hibernate、JDBC、JTA) 进行封装, 其本身并没有提供具体的事务服务实现 (比如, 它并没有实现 JTA 规范)。不重复发明轮子对于 Spring 事务服务抽象仍然适用。

在 Spring 提供的所有事务管理器中, PlatformTransactionManager 是最基本的接口。因此, 本章接下来将从介绍它开始。

7.2.1 PlatformTransactionManager

在 Spring 提供的事务抽象中, org.springframework.transaction.PlatformTransactionManager 接口是很重要的, 图 7-1 给出了其包含的方法。

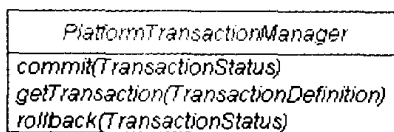


图 7-1 PlatformTransactionManager 接口

开发者应该很清楚, Spring 框架肯定会为它提供若干个实现, 供 Spring IoC 实现控制反转使用。比如, 在 exmple11 中就使用过它的实现, applicationContext.xml 的相应片断如下。对于 Hibernate 而言, 需要借助于 HibernateTransactionManager 事务管理器。开发者必须为它提供 sessionFactory 取值。实际上, HibernateTransactionManager 将会把事务处理的具体工作委派给 Hibernate 中的 net.sf.hibernate.Transaction 对象。其实, 在 Spring 提供的所有事务管理器中, 都是对底层事务对象的封装。它自身并没有实现底层事务的管理。这也符合 Spring 的设计初衷, “不重复发明轮子”。因此, 对 HibernateTransactionManager 的 commit 和 rollback 操作将委派给 Transaction 对象。

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

```
<property name="sessionFactory">
    <ref local="sessionFactory"/>
</property>
</bean>
```

其中, **HibernateTransactionManager** 就是实现了 **PlatformTransactionManager** 接口, 而且它还是通过 Spring IoC 容器加载的, 这也符合 Spring 风格。如下给出了 **Hibernate** 事务管理器处理事务提交的代码 (摘自 Spring 源码)。开发者是否注意到 **txObject** 对象。通过 **txObject** 对象能够获得 **sessionHolder**, 进而获得 **Hibernate Transaction** 对象, 最后再进行事务的 **commit** 操作。

```
protected void doCommit(DefaultTransactionStatus status) {
    HibernateTransactionObject txObject =
        (HibernateTransactionObject) status.getTransaction();
    if (status.isDebugEnabled()) {
        logger.debug("Committing Hibernate transaction on session [" +
            txObject.getSessionHolder().getSession() + "]");
    }
    try {
        txObject.getSessionHolder().getTransaction().commit();
    }
    catch (net.sf.hibernate.TransactionException ex) {
        // assumably from commit call to the underlying JDBC connection
        throw new TransactionSystemException("Could not commit Hibernate
transaction", ex);
    }
    catch (JDBCException ex) {
        // assumably failed to flush changes to database
        throw convertJdbcAccessException(ex.getSQLException());
    }
    catch (HibernateException ex) {
        // assumably failed to flush changes to database
        throw convertHibernateAccessException(ex);
    }
}
```

当然, 对于 **HibernateTransactionManager** 事务管理器的 **rollback** 操作, 也是通过 **Transaction** 对象进行的。比如, 相应的代码如下。

```
try {
    txObject.getSessionHolder().getTransaction().rollback();
}
```

通过整理 Spring 提供的 **PlatformTransactionManager** 接口实现, 可以获得图 7-2。

它给出了 Spring 实现的 **PlatformTransactionManager**。其中, 主要涉及到的事务管理器有:

- **JDBC 事务**: 借助于 **DataSourceTransactionManager** 实现。
- **JTA 事务**: 借助于 **JtaTransactionManager**、**WebLogicJtaTransactionManager** 实现。
- **Hibernate 事务**: 借助于 **HibernateTransactionManager** 实现。前面已经详细阐述过。
- **JDO 事务**: 借助于 **JdoTransactionManager** 实现。

- OJB 事务：借助于 PersistenceBrokerTransactionManager 实现。
- JMS 事务：借助于 JmsTransactionManager 实现。

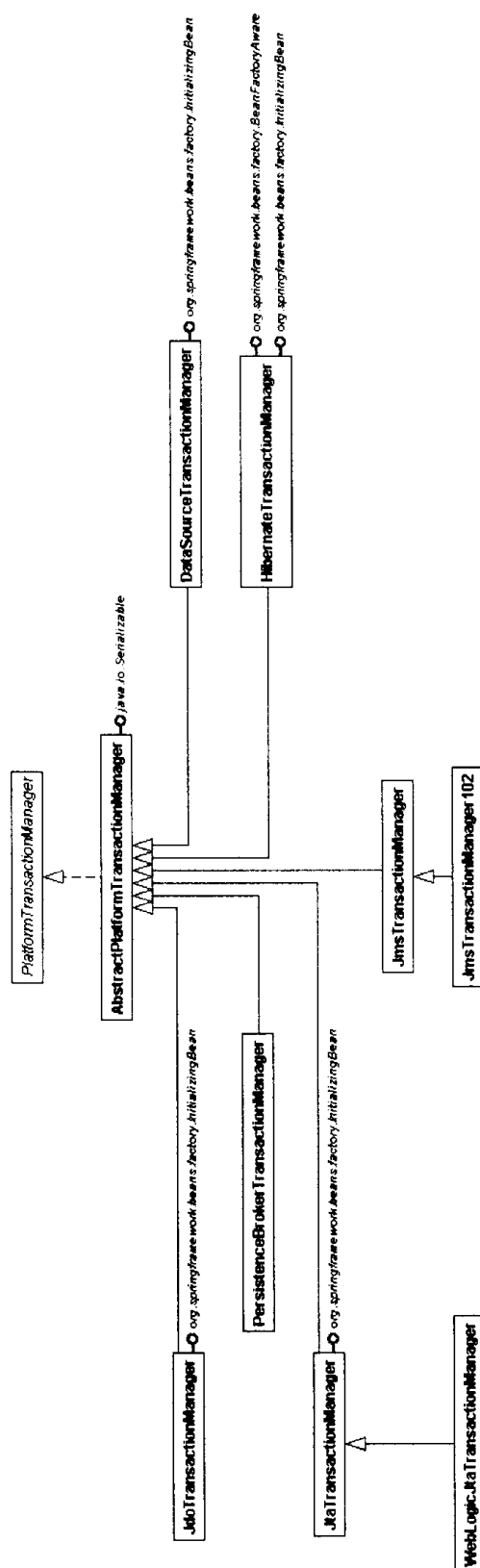


图 7-2 PlatformTransactionManager 接口实现

对于 JDBC 事务而言, 需要配置 `DataSourceTransactionManager`。配置示例如下。开发者需要为它准备 `dataSource` 取值。至于 `dataSource` 的具体来源 (是基于 JNDI, 还是其他机制), `DataSourceTransactionManager` 可以不用关注。这或许就是 IoC 的好处, 而且还可以在部署场合动态切换。`DataSourceTransactionManager` 会将事务 commit 和 rollback 操作委派给 `java.sql.Connection`。

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

<bean id="dataSource"
      class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
    <value>java:/MySqlDS</value>
  </property>
</bean>
```

对于 JDO 事务而言, 需要配置 `JdoTransactionManager`。配置示例如下。当然, 在配置 `persistenceManagerFactory` JavaBean 时, 开发者还需配置数据源。同其他事务管理器一样, `JdoTransactionManager` 将事务提交和回滚操作委派给了 `javax.jdo.Transaction API`。

```
<bean id="transactionManager"
      class="org.springframework.orm.jdo.JdoTransactionManager">
  <property name="persistenceManagerFactory">
    <ref local="persistenceManagerFactory"/>
  </property>
</bean>
```

对于 JTA 事务而言, 需要配置 `JtaTransactionManager` 或 `WebLogicJtaTransactionManager` 实现。配置示例如下。其中, 指定了 `userTransactionName` 属性。默认时, `JtaTransactionManager` 设定 `userTransactionName` 属性取值为 `java:comp/UserTransaction`, 因此如果目标 JTA 实现的 `UserTransaction` 名为 `java:comp/UserTransaction`, 则不用给出 `userTransactionName` 属性。当然, 它除了 `userTransactionName` 属性外, 还具有 `transactionManagerName`、`jndiTemplate`、`autodetectTransactionManager` 等属性。其中, `transactionManagerName` 指定事务管理器名字; `jndiTemplate` 指定 JNDI 模板, 供查找事务管理器和 `UserTransaction` 对象使用; `autodetectTransactionManager` 用于指定是否自动查找事务管理器 (默认时为 `true`)。

```
<bean id="transactionManager"
      class="org.springframework.transaction.
        jta.JtaTransactionManager">
  <property name="userTransactionName">
    <value>java:comp/UserTransaction</value>
  </property>
</bean>
```

因此, 开发者可以根据自身的业务需求、应用架构来选择相应的事务实现。借助于 Spring 提供的事务服务抽象, 开发者不仅可以使用 Spring 提供的各种事务管理器, 还可以自定义开发事务管理器。

7.2.2 声明式事务

在 J2EE 中, EJB 因为可以使用声明式事务而著称。现在, Spring 使能应用也可以享受到这一待遇了(不仅如此, 借助于 Acegi, Spring 应用还可以实现声明式安全性, 详情见第 17 章)。注意, 这里的声明式事务针对的是 POJO 对象。

借助于 Spring AOP, Spring 提供了 TransactionProxyFactoryBean。它是为简化声明式事务处理而引入的代理工厂 JavaBean。当然, 也可以使用 ProxyFactoryBean 和 TransactionInterceptor 的组合。在 TransactionProxyFactoryBean 内部使用了事务拦截器, 即 TransactionInterceptor。

在深入到声明式事务之前, 还是看看一个实例吧。好, 那就从事务的提交和回滚开始。

1. example11 中事务的提交和回滚

还是以 example11 展开论述吧! 如果开发者在图 7-3 中单击“提交”按钮, 则会触发事务操作。

图 7-3 example11 中的提交界面

开发者可以查看 D:\jboss-4.0.0\server\default\log 目录中的 server.log 文件。类似的日志如下。注意, 本书给出了详细的注释, 这些注释对于理解具体的事务操作很重要。

```
//Tapestry 准备进入 Web 页面的 rebind 阶段
```

```
2004-11-07 09:24:30,676 INFO [com.openv.spring.tapestry.Home]
pageBeginRender().....
2004-11-07 09:24:30,676 INFO [com.openv.spring.tapestry.Home]
buildCountrymodel().....
```

```
2004-11-07 09:24:30,706 INFO [com.openv.spring.tapestry.Home]
submitPersonInfo().....
```

```
//Spring TransactionInterceptor 拦截器“闻”到 setUserInfo
//其中, appcontext.xml 配置文件中的<prop key="set*">PROPAGATION_REQUIRED
//</prop>, 告知了 TransactionInterceptor 拦截器, 即碰到 set 开头的业务方法时, 需要
//保证这种业务方法处于事务中
//如果调用客户不处于事务中或者没有提供事务支持, 则会启动新的事务, 以完成对 setUserInfo
//的调用
```

```
2004-11-07 09:24:30,706 DEBUG
[org.springframework.transaction.interceptor.TransactionInterceptor]
Getting transaction for method 'setUserInfo' in class
[com.openv.spring.service.IExample11Manager]
```

```
//获得事务对象, 即 HibernateTransactionObject
//其中, 大部分企业 Bean 都是事务对象
```

```
2004-11-07 09:24:30,706 DEBUG
[org.springframework.orm.hibernate.HibernateTransactionManager] Using
transaction object
[org.springframework.orm.hibernate.HibernateTransactionObject@36fd8d]
```

```
//使用 HibernateTransactionManager 创建新事务
```

```
2004-11-07 09:24:30,706 DEBUG
[org.springframework.orm.hibernate.HibernateTransactionManager] Creating
new transaction
```

```
//打开 Hibernate 会话
```

```
2004-11-07 09:24:30,706 DEBUG
[org.springframework.orm.hibernate.SessionFactoryUtils] Opening Hibernate
session
```

```
2004-11-07 09:24:30,706 DEBUG [net.sf.hibernate.impl.SessionImpl] opened
session
```

```
2004-11-07 09:24:30,706 DEBUG
[org.springframework.orm.hibernate.HibernateTransactionManager] Opened new
session [net.sf.hibernate.impl.SessionImpl@404baf] for Hibernate transaction
```

```
//使用 JDBCTransaction 开始事务, 并判断当前 RDBMS 连接的自动提交状态
//然后, 将自动提交失效
```

```
2004-11-07 09:24:30,706 DEBUG [net.sf.hibernate.transaction.JDBCTransaction]
begin
```

```
2004-11-07 09:24:30,706 DEBUG [net.sf.hibernate.transaction.JDBCTransaction]
```

```
current autocommit status:true
2004-11-07 09:24:30,706 DEBUG [net.sf.hibernate.transaction.JDBCTransaction]
disabling autocommit

//注意, Hibernate 事务其实只是对 JDBC 事务的包裹 (wrap)

2004-11-07 09:24:30,706 DEBUG
[org.springframework.orm.hibernate.HibernateTransactionManager] Exposing
Hibernate transaction as JDBC transaction
[org.jboss.resource.adapter.jdbc.WrappedConnection@128dca6]

//触发 TransactionSynchronizationManager

2004-11-07 09:24:30,706 DEBUG
[org.springframework.transaction.support.TransactionSynchronizationManage
r] Bound value [org.springframework.jdbc.datasource.ConnectionHolder@1ae37b5]
for key [org.jboss.resource.adapter.jdbc.WrapperDataSource@1626c6d] to
thread [http-0.0.0.0-8080-Processor23]
2004-11-07 09:24:30,706 DEBUG
[org.springframework.transaction.support.TransactionSynchronizationManage
r] Bound value [org.springframework.orm.hibernate.SessionHolder@cb7e2c] for
key [net.sf.hibernate.impl.SessionFactoryImpl@13206fd] to thread
[http-0.0.0.0-8080-Processor23]
2004-11-07 09:24:30,706 DEBUG
[org.springframework.transaction.support.TransactionSynchronizationManage
r] Initializing transaction synchronization
2004-11-07 09:24:30,746 DEBUG
[org.springframework.transaction.support.TransactionSynchronizationManage
r] Retrieved value [org.springframework.orm.hibernate.SessionHolder@cb7e2c]
for key [net.sf.hibernate.impl.SessionFactoryImpl@13206fd] bound to thread
[http-0.0.0.0-8080-Processor23]

//利用 UUIDGenerator.getInstance().generateTimeBasedUUID() 获得主键

2004-11-07 09:24:30,746 DEBUG [net.sf.hibernate.impl.SessionImpl] generated
identifier: c61062a7-305b-11d9-9c77-27855cf279ba
2004-11-07 09:24:30,746 DEBUG [net.sf.hibernate.impl.SessionImpl] saving
[com.openv.spring.service.hibernate.UserInfo#c61062a7-305b-11d9-9c77-2785
5cf279ba]
2004-11-07 09:24:30,756 DEBUG
[org.springframework.transaction.support.TransactionSynchronizationManage
r] Retrieved value [org.springframework.orm.hibernate.SessionHolder@cb7e2c]
for key [net.sf.hibernate.impl.SessionFactoryImpl@13206fd] bound to thread
[http-0.0.0.0-8080-Processor23]
2004-11-07 09:24:30,756 DEBUG
[org.springframework.transaction.support.TransactionSynchronizationManage
```

```
r] Retrieved value [org.springframework.orm.hibernate.SessionHolder@cb7e2c]
for key [net.sf.hibernate.impl.SessionFactoryImpl@13206fd] bound to thread
[http-0.0.0.0-8080-Processor23]
2004-11-07 09:24:30,756 DEBUG [net.sf.hibernate.impl.SessionImpl] generated
identifier: c614f688-305b-11d9-9c77-27855cf279ba
2004-11-07 09:24:30,756 DEBUG [net.sf.hibernate.impl.SessionImpl] saving
[com.openv.spring.service.hibernate.UserInterest#c614f688-305b-11d9-9c77-
27855cf279ba]
2004-11-07 09:24:30,756 DEBUG
[org.springframework.transaction.support.TransactionSynchronizationManage
r] Retrieved value [org.springframework.orm.hibernate.SessionHolder@cb7e2c]
for key [net.sf.hibernate.impl.SessionFactoryImpl@13206fd] bound to thread
[http-0.0.0.0-8080-Processor23]
2004-11-07 09:24:30,756 DEBUG
[org.springframework.transaction.support.TransactionSynchronizationManage
r] Retrieved value [org.springframework.orm.hibernate.SessionHolder@cb7e2c]
for key [net.sf.hibernate.impl.SessionFactoryImpl@13206fd] bound to thread
[http-0.0.0.0-8080-Processor23]
2004-11-07 09:24:30,756 DEBUG [net.sf.hibernate.impl.SessionImpl] generated
identifier: c614f689-305b-11d9-9c77-27855cf279ba
2004-11-07 09:24:30,756 DEBUG [net.sf.hibernate.impl.SessionImpl] saving
[com.openv.spring.service.hibernate.UserInterest#c614f689-305b-11d9-9c77-
27855cf279ba]
2004-11-07 09:24:30,756 DEBUG
[org.springframework.transaction.support.TransactionSynchronizationManage
r] Retrieved value [org.springframework.orm.hibernate.SessionHolder@cb7e2c]
for key [net.sf.hibernate.impl.SessionFactoryImpl@13206fd] bound to thread
[http-0.0.0.0-8080-Processor23]
2004-11-07 09:24:30,756 DEBUG
[org.springframework.transaction.interceptor.TransactionInterceptor]
Invoking commit for transaction on method 'setUserInfo' in class
[com.openv.spring.service.IExample11Manager]
2004-11-07 09:24:30,756 DEBUG
[org.springframework.orm.hibernate.HibernateTransactionManager] Triggering
beforeCommit synchronization
2004-11-07 09:24:30,756 DEBUG
[org.springframework.orm.hibernate.HibernateTransactionManager] Triggering
beforeCompletion synchronization
2004-11-07 09:24:30,756 DEBUG
[org.springframework.orm.hibernate.HibernateTransactionManager] Initiating
transaction commit
2004-11-07 09:24:30,756 DEBUG
[org.springframework.orm.hibernate.HibernateTransactionManager] Committing
Hibernate transaction on session [net.sf.hibernate.impl.SessionImpl@404baf]
```

//触发 JDBC 事务提交

```
2004-11-07 09:24:30,756 DEBUG [net.sf.hibernate.transaction.JDBCTransaction]
commit
2004-11-07 09:24:30,756 DEBUG [net.sf.hibernate.impl.SessionImpl] flushing
session
2004-11-07 09:24:30,756 DEBUG [net.sf.hibernate.impl.SessionImpl] Flushing
entities and processing referenced collections
2004-11-07 09:24:30,756 DEBUG [net.sf.hibernate.impl.SessionImpl] Processing
unreferenced collections
2004-11-07 09:24:30,756 DEBUG [net.sf.hibernate.impl.SessionImpl] Scheduling
collection removes/(re)creates/updates
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.impl.SessionImpl] Flushed:
3 insertions, 0 updates, 0 deletions to 3 objects
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.impl.SessionImpl] Flushed:
0 (re)creations, 0 updates, 0 removals to 0 collections
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.impl.Printer] listing
entities:
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.impl.Printer]
com.openv.spring.service.hibernate.UserInterest{username=luoshifei,
name=????, id=c614f688-305b-11d9-9c77-27855cf279ba}
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.impl.Printer]
com.openv.spring.service.hibernate.UserInterest{username=luoshifei,
name=????é, id=c614f689-305b-11d9-9c77-27855cf279ba}
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.impl.Printer]
com.openv.spring.service.hibernate.UserInfo{password=luoshifei,
country=????ú, username=luoshifei, email=j2eebeans@yahoo.com.cn,
id=c61062a7-305b-11d9-9c77-27855cf279ba}
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.impl.SessionImpl] executing
flush
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.persister.EntityPersister]
Inserting entity:
[com.openv.spring.service.hibernate.UserInfo#c61062a7-305b-11d9-9c77-2785
5cf279ba]
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.impl.BatcherImpl] about to
open: 0 open PreparedStatements, 0 open ResultSets
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.SQL] insert into userinfo
(username, password, email, country, id) values (?, ?, ?, ?, ?)
2004-11-07 09:24:30,766 INFO [STDOUT] Hibernate: insert into userinfo
(username, password, email, country, id) values (?, ?, ?, ?, ?)
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.impl.BatcherImpl] preparing
statement
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.persister.EntityPersister]
Dehydrating entity:
[com.openv.spring.service.hibernate.UserInfo#c61062a7-305b-11d9-9c77-2785
5cf279ba]
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.type.StringType] binding
```

```
'luoshifei' to parameter: 1
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.type.StringType] binding
'luoshifei' to parameter: 2
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.type.StringType] binding
'j2eebeans@yahoo.com.cn' to parameter: 3
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.type.StringType] binding
'???ú' to parameter: 4
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.type.StringType] binding
'c61062a7-305b-11d9-9c77-27855cf279ba' to parameter: 5
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.impl.BatcherImpl] Adding to
batch
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.persister.EntityPersister]
Inserting entity:
[com.openv.spring.service.hibernate.UserInterest#c614f688-305b-11d9-9c77-
27855cf279ba]
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.impl.BatcherImpl] Executing
batch size: 1
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.impl.BatcherImpl] done
closing: 0 open PreparedStatements, 0 open ResultSets
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.impl.BatcherImpl] closing
statement
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.impl.BatcherImpl] about to
open: 0 open PreparedStatements, 0 open ResultSets
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.SQL] insert into userinterest
(name, username, id) values (?, ?, ?)
2004-11-07 09:24:30,766 INFO [STDOUT] Hibernate: insert into userinterest
(name, username, id) values (?, ?, ?)
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.impl.BatcherImpl] preparing
statement
2004-11-07 09:24:30,766 DEBUG [net.sf.hibernate.persister.EntityPersister]
Dehydrating entity:
[com.openv.spring.service.hibernate.UserInterest#c614f688-305b-11d9-9c77-
27855cf279ba]
2004-11-07 09:24:30,776 DEBUG [net.sf.hibernate.type.StringType] binding
'????' to parameter: 1
2004-11-07 09:24:30,776 DEBUG [net.sf.hibernate.type.StringType] binding
'luoshifei' to parameter: 2
2004-11-07 09:24:30,776 DEBUG [net.sf.hibernate.type.StringType] binding
'c614f688-305b-11d9-9c77-27855cf279ba' to parameter: 3
2004-11-07 09:24:30,776 DEBUG [net.sf.hibernate.impl.BatcherImpl] Adding to
batch
2004-11-07 09:24:30,776 DEBUG [net.sf.hibernate.persister.EntityPersister]
Inserting entity:
[com.openv.spring.service.hibernate.UserInterest#c614f689-305b-11d9-9c77-
27855cf279ba]
2004-11-07 09:24:30,776 DEBUG [net.sf.hibernate.persister.EntityPersister]
```


Dehydrating entity:

[com.openv.spring.service.hibernate.UserInterest#c614f689-305b-11d9-9c77-27855cf279ba]

2004-11-07 09:24:30,776 DEBUG [net.sf.hibernate.type.StringType] binding '???é' to parameter: 1

2004-11-07 09:24:30,776 DEBUG [net.sf.hibernate.type.StringType] binding 'luoshifei' to parameter: 2

2004-11-07 09:24:30,776 DEBUG [net.sf.hibernate.type.StringType] binding 'c614f689-305b-11d9-9c77-27855cf279ba' to parameter: 3

2004-11-07 09:24:30,776 DEBUG [net.sf.hibernate.impl.BatcherImpl] Adding to batch

2004-11-07 09:24:30,776 DEBUG [net.sf.hibernate.impl.BatcherImpl] Executing batch size: 2

2004-11-07 09:24:30,776 DEBUG [net.sf.hibernate.impl.BatcherImpl] done closing: 0 open PreparedStatements, 0 open ResultSets

2004-11-07 09:24:30,776 DEBUG [net.sf.hibernate.impl.BatcherImpl] closing statement

2004-11-07 09:24:30,776 DEBUG [net.sf.hibernate.impl.SessionImpl] post flush

//事务完成

2004-11-07 09:24:30,806 DEBUG [net.sf.hibernate.impl.SessionImpl] transaction completion

//将JDBC自动提交生效。准确地说，应该是返回事务前的状态

2004-11-07 09:24:30,806 DEBUG [net.sf.hibernate.transaction.JDBCTransaction] re-enabling autocommit

2004-11-07 09:24:30,806 DEBUG

[org.springframework.orm.hibernate.HibernateTransactionManager] Triggering afterCompletion synchronization

2004-11-07 09:24:30,806 DEBUG

[org.springframework.transaction.support.TransactionSynchronizationManager] Clearing transaction synchronization

2004-11-07 09:24:30,806 DEBUG

[org.springframework.transaction.support.TransactionSynchronizationManager] Removed value [org.springframework.orm.hibernate.SessionHolder@cb7e2c]

for key [net.sf.hibernate.impl.SessionFactoryImpl@13206fd] from thread [http-0.0.0.0-8080-Processor23]

2004-11-07 09:24:30,806 DEBUG

[org.springframework.transaction.support.TransactionSynchronizationManager] Removed value

[org.springframework.jdbc.datasource.ConnectionHolder@1ae37b5] for key [org.jboss.resource.adapter.jdbc.WrapperDataSource@1626c6d] from thread [http-0.0.0.0-8080-Processor23]

2004-11-07 09:24:30,806 DEBUG

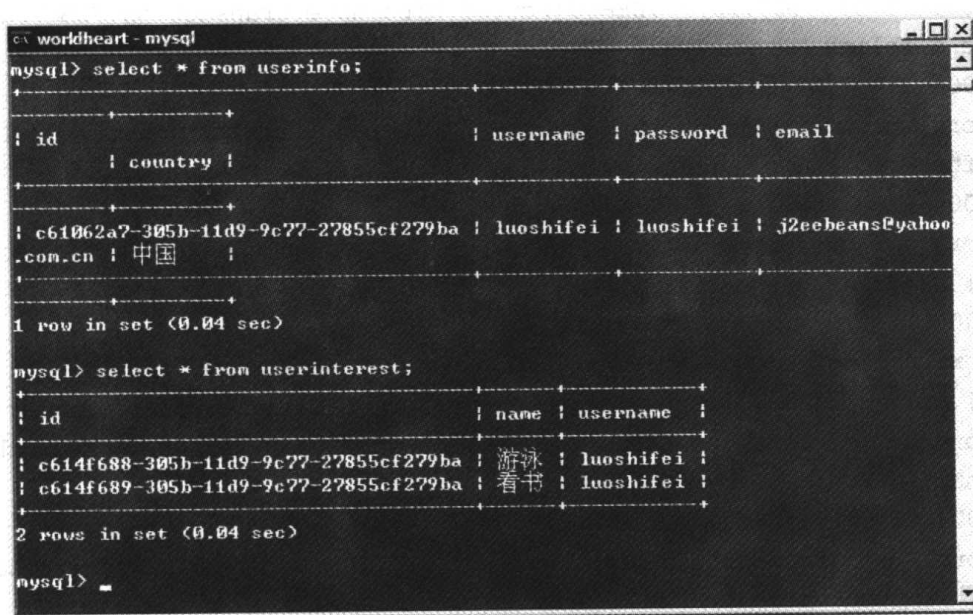
2004-11-07 09:24:30,806 DEBUG

```
[org.springframework.orm.hibernate.HibernateTransactionManager] Closing
Hibernate session [net.sf.hibernate.impl.SessionImpl@404baf] after
transaction
2004-11-07 09:24:30,806 DEBUG
[org.springframework.orm.hibernate.SessionFactoryUtils] Closing Hibernate
session
2004-11-07 09:24:30,806 DEBUG [net.sf.hibernate.impl.SessionImpl] closing
session
2004-11-07 09:24:30,806 DEBUG [net.sf.hibernate.impl.SessionImpl]
disconnecting session
2004-11-07 09:24:30,806 DEBUG [net.sf.hibernate.impl.SessionImpl]
transaction completion
```

//再次将 Tapestry Web 页面 render 出来给 IE

```
2004-11-07 09:24:30,806 INFO [com.openv.spring.tapestry.Home]
pageBeginRender().....
2004-11-07 09:24:30,846 DEBUG [net.sf.hibernate.impl.SessionImpl] running
Session.finalize()
2004-11-07 09:24:30,886 DEBUG
[org.springframework.beans.factory.support.DefaultListableBeanFactory]
Returning cached instance of singleton bean 'example11Service'
2004-11-07 09:24:30,886 DEBUG
[org.springframework.beans.factory.support.DefaultListableBeanFactory]
Bean with name 'example11Service' is a factory bean
```

开发者通过 MySQL 控制台可以获悉操作是否成功, 具体如图 7-4 所示。



```
mysql> select * from userinfo;
+-----+-----+-----+-----+
| id | country | username | password | email |
+-----+-----+-----+-----+
| c61062a7-305b-11d9-9c77-27855cf279ba | 中国 | luoshifei | luoshifei | j2eebeans@yahoo.com.cn |
+-----+-----+-----+-----+
1 row in set (0.04 sec)

mysql> select * from userinterest;
+-----+-----+-----+
| id | name | username |
+-----+-----+-----+
| c614f688-305b-11d9-9c77-27855cf279ba | 游泳 | luoshifei |
| c614f689-305b-11d9-9c77-27855cf279ba | 看书 | luoshifei |
+-----+-----+-----+
2 rows in set (0.04 sec)

mysql>
```

图 7-4 MySQL 中的数据

其中, 在上述过程中只是触发了 `UserInfoDAO.java` 中的 `setUserInfo(UserInfoVO userinfoVO)` 操作, 而该操作位于事务中。其内容如下:

```

/**
 * 存储用户注册信息
 *
 * @param userinfoVO
 *         用户注册信息
 *
 * @return boolean 存储用户注册信息是否成功
 */
public boolean setUserInfo(UserInfoVO userinfoVO)
    throws DataAccessException {
    if (userinfoVO == null)
        return false;
    UserInfo ui = new UserInfo();
    ui.setId(getID());
    ui.setUsername(getFromEncodingTo(userinfoVO.getUsername().trim(),
        "gbk", "iso-8859-1"));
    ui.setPassword(userinfoVO.getPassword().trim());
    ui.setEmail(userinfoVO.getEmail().trim());
    ui.setCountry(getFromEncodingTo(userinfoVO.getCountry(), "gbk",
        "iso-8859-1"));
    this.getHibernateTemplate().save(ui);
    List interestList = userinfoVO.getSelectedinterests();
    if (interestList == null)
        return true;
    UserInterest userinterest;
    for (int i = 0, k = interestList.size(); i < k; ++i) {
        userinterest = new UserInterest();
        userinterest.setId(getID());

        userinterest.setUsername(getFromEncodingTo(userinfoVO.getUsername()
            .trim(), "gbk", "iso-8859-1"));
        userinterest.setName(getFromEncodingTo(
            (String) interestList.get(i), "gbk", "iso-8859-1"));
        this.getHibernateTemplate().save(userinterest);
    }
    return true;
}

```

在上述方法¹中，借助于 `HibernateTemplate` 操作了 `userinfo` 和 `userinterest` 表。其中，`userinfo` 表除用户兴趣之外，用户的其他信息都存储到该表中。而 `userinterest` 表仅存储用户的兴趣列表。因此，这两张表构成了用户信息。

通过上述过程，开发者已经看到了事务的具体过程。本书再次强调一次，上面给出的

¹ 注意，如果开发者采用 MySQL 4.0.x 系列，则通过在 `connection-url` 后面附加如下的内容，将不需要手工处理编码问题，即 `useUnicode=true&characterEncoding=GBK`（如果是在 XML 文件中，比如 `mysql-ds.xml`，则需要为 `&` 提供转义，`&`）。当然，相应的 MySQL JDBC 驱动千万不要使用 3.0.11 版本的，因为它在处理中文时存在 bug。

输出日志，请开发者仔细多阅读几次。尤其是事务的启动、提交过程，值得开发者仔细研究。当然，本章后续内容将围绕整个事务过程进行讲述。

如果开发者希望采用 `JtaTransactionManager` 代替 `HibernateTransactionManager`，即使用 JTA 事务代替 Hibernate 事务，则 Spring 配置过程很简单，只需要将如下内容：

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

替换成：

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

开发者可以再次运行 JBoss 4.0 服务器，并重新部署 `example11.war` 应用。然后重复上述过程。太完美了，一切如你所愿。但是，请注意，本书使用了 MySQL 本地数据源，即 `local-tx-datasource`。而对于 JTA 而言，需要配置全局数据源。当然，使用 `local-tx-datasource` 演示 `example11.war` 还是没有问题的，因为 JBoss 4.0 允许开发者这样做，更何况 `example11.war` 仅仅涉及到一个事务性资源。

开发者现在应该可以举一反三了，如果同 RDBMS 交互的不是 Hibernate，那么该如何操作。其实是一样的，还记得前面的图 7-2 吗？好了，再次强调一下吧，因为正确使用 `PlatformTransactionManager` 接口及其实现对于正确使用事务管理而言太关键了！

- 如果直接采用 JDBC 同 RDBMS 交互，则需要使用 `DataSourceTransactionManager` 实现。
- 如果直接采用 Hibernate 同 RDBMS 交互，则需要使用 `HibernateTransactionManager` 实现。
- 如果直接采用 JDO 同 RDBMS 交互，则需要使用 `JdoTransactionManager` 实现。
- 如果采用 OJB 同 RDBMS 交互，则需要使用 `PersistenceBrokerTransactionManager` 实现。
- 如果直接采用 JTA 控制应用系统同 RDBMS 的交互，则开发者需要使用 `JtaTransactionManager` 实现。当然，对于不同的 J2EE 应用服务而言，具体使用的 `PlatformTransactionManager` 接口实现会有所区别。比如，Spring 框架还提供了针对 WebLogic 的 `WebLogicJtaTransactionManager` 实现。

当然，开发者可以修改 `Home.java` 源文件（或者直接修改 `UserInfoDAO.java` 源文件），以对事务回滚做些试验。本文修改了如下内容（位于 `UserInfoDAO.java` 文件中）。

```
for (int i = 0, k = interestList.size(); i < k; ++i) {
    userinterest = new UserInterest();
```

```
//验证事务回滚操作
```

```
userinterest.setId(null);
```

```
//      userinterest.setId(getID());
```

```
userinterest.setUsername(getFromEncodingTo(userinfoVO.getUsername()  
    .trim(), "gbk", "iso-8859-1"));  
userinterest.setName(getFromEncodingTo(  
    (String) interestList.get(i), "gbk", "iso-8859-1"));  
this.getHibernateTemplate().save(userinterest);  
}
```

然后重新部署 `example11.war` 应用。

对于使用了 `JtaTransactionManager` 事务管理器而言, JBoss 4.0 的 `server.log` 日志文件中会给出如下日志信息。开发者可以去 MySQL 控制台验证是否有新数据插入。

```
2004-11-07 10:44:36,637 INFO [com.openv.spring.tapestry.Home]  
pageBeginRender().....  
2004-11-07 10:44:36,637 INFO [com.openv.spring.tapestry.Home]  
buildCountrymodel().....  
2004-11-07 10:44:36,647 INFO [com.openv.spring.tapestry.Home]  
submitPersonInfo().....
```

```
//拦截到 setUserInfo 方法。依据 TransactionProxyFactoryBean 设定的事务规则,  
//需要使用事务对象
```

```
2004-11-07 10:44:36,657 DEBUG  
[org.springframework.transaction.interceptor.TransactionInterceptor]  
Getting transaction for method 'setUserInfo' in class  
[com.openv.spring.service.IExample11Manager]  
2004-11-07 10:44:36,657 DEBUG  
[org.springframework.transaction.jta.JtaTransactionManager] Using  
transaction object  
[org.springframework.transaction.jta.JtaTransactionObject@1354a1a]
```

```
//创建新的事务
```

```
2004-11-07 10:44:36,657 DEBUG  
[org.springframework.transaction.jta.JtaTransactionManager] Creating new  
transaction
```

```
//开始 JTA 事务
```

```
2004-11-07 10:44:36,657 DEBUG  
[org.springframework.transaction.jta.JtaTransactionManager] Beginning JTA  
transaction  
2004-11-07 10:44:36,657 DEBUG  
[org.springframework.transaction.support.TransactionSynchronizationManage  
r] Initializing transaction synchronization
```

```
//开始 Hibernate Session
```

```
2004-11-07 10:44:36,707 DEBUG
```

```
[org.springframework.orm.hibernate.SessionFactoryUtils] Opening Hibernate session
2004-11-07 10:44:36,707 DEBUG [net.sf.hibernate.impl.SessionImpl] opened session
2004-11-07 10:44:36,707 DEBUG
[org.springframework.orm.hibernate.SessionFactoryUtils] Registering Spring transaction synchronization for new Hibernate session
2004-11-07 10:44:36,707 DEBUG
[org.springframework.transaction.support.TransactionSynchronizationManager] Bound value [org.springframework.orm.hibernate.SessionHolder@12cfeb7] for key [net.sf.hibernate.impl.SessionFactoryImpl@17a0daa] to thread [http-0.0.0.0-8080-Processor23]
```

//利用 UUIDGenerator 创建主键

```
2004-11-07 10:44:36,707 DEBUG [net.sf.hibernate.impl.SessionImpl] generated identifier: f6a55d3c-3066-11d9-9627-13b813384cb8
2004-11-07 10:44:36,717 DEBUG [net.sf.hibernate.impl.SessionImpl] saving [com.openv.spring.service.hibernate.UserInfo#f6a55d3c-3066-11d9-9627-13b813384cb8]
2004-11-07 10:44:36,727 DEBUG
[org.springframework.transaction.support.TransactionSynchronizationManager] Retrievedvalue [org.springframework.orm.hibernate.SessionHolder@12cfeb7] for key [net.sf.hibernate.impl.SessionFactoryImpl@17a0daa] bound to thread [http-0.0.0.0-8080-Processor23]
2004-11-07 10:44:36,727 DEBUG
[org.springframework.transaction.support.TransactionSynchronizationManager] Retrievedvalue [org.springframework.orm.hibernate.SessionHolder@12cfeb7] for key [net.sf.hibernate.impl.SessionFactoryImpl@17a0daa] bound to thread [http-0.0.0.0-8080-Processor23]
2004-11-07 10:44:36,737 DEBUG
[org.springframework.transaction.support.TransactionSynchronizationManager] Retrievedvalue [org.springframework.orm.hibernate.SessionHolder@12cfeb7] for key [net.sf.hibernate.impl.SessionFactoryImpl@17a0daa] bound to thread [http-0.0.0.0-8080-Processor23]
```

//根据 Spring 事务服务抽象提供的拦截器, 即 RuleBasedTransactionAttribute 来判断
//事务是否应该回滚

```
2004-11-07 10:44:36,737 DEBUG
[org.springframework.transaction.interceptor.RuleBasedTransactionAttribute] Applying rules to determine whether transaction should rollback on org.springframework.orm.hibernate.HibernateSystemException: ids for this class must be manually assigned before calling save(): com.openv.spring.service.hibernate.UserInterest; nested exception is net.sf.hibernate.id.IdentifierGenerationException: ids for this class must
```

```
be manually assigned before calling save():
com.openv.spring.service.hibernate.UserInterest
2004-11-07 10:44:36,737 DEBUG
[org.springframework.transaction.interceptor.RuleBasedTransactionAttribut
e] Winning rollback rule is: null
2004-11-07 10:44:36,737 DEBUG
[org.springframework.transaction.interceptor.RuleBasedTransactionAttribut
e] No relevant rollback rule found: applying superclass default
2004-11-07 10:44:36,737 DEBUG
[org.springframework.transaction.interceptor.TransactionInterceptor]
Invoking rollback for transaction on method 'setUserInfo' in class
[com.openv.spring.service.IExample11Manager] due to throwable
[org.springframework.orm.hibernate.HibernateSystemException: ids for this
class must be manually assigned before calling save():
com.openv.spring.service.hibernate.UserInterest; nested exception is
net.sf.hibernate.id.IdentifierGenerationException: ids for this class must
be manually assigned before calling save():
com.openv.spring.service.hibernate.UserInterest]
2004-11-07 10:44:36,737 DEBUG
[org.springframework.transaction.jta.JtaTransactionManager] Triggering
beforeCompletion synchronization
2004-11-07 10:44:36,737 DEBUG
[org.springframework.transaction.support.TransactionSynchronizationManage
r] Removed value [org.springframework.orm.hibernate.SessionHolder@12cfeb7]
for key [net.sf.hibernate.impl.SessionFactoryImpl@17a0daa] from thread
[http-0.0.0.0-8080-Processor23]

//开始回滚

2004-11-07 10:44:36,737 DEBUG
[org.springframework.transaction.jta.JtaTransactionManager] Initiating
transaction rollback
2004-11-07 10:44:36,737 DEBUG
[org.springframework.transaction.jta.JtaTransactionManager] RollingbackJTA
transaction
2004-11-07 10:44:36,737 DEBUG
[org.springframework.transaction.jta.JtaTransactionManager] Triggering
afterCompletion synchronization
2004-11-07 10:44:36,737 DEBUG [net.sf.hibernate.impl.SessionImpl]
transaction completion

//关闭Hibernate Session

2004-11-07 10:44:36,737 DEBUG
[org.springframework.orm.hibernate.SessionFactoryUtils] Closing Hibernate
session
```

```
2004-11-07 10:44:36,737 DEBUG [net.sf.hibernate.impl.SessionImpl] closing
session
2004-11-07 10:44:36,737 DEBUG
[org.springframework.transaction.support.TransactionSynchronizationManage
r] Clearing transaction synchronization
2004-11-07 10:44:36,927 DEBUG [net.sf.hibernate.impl.SessionImpl] running
Session.finalize()
2004-11-07 10:44:37,448 DEBUG
[org.springframework.beans.factory.support.DefaultListableBeanFactory]
Returning cached instance of singleton bean 'example11Service'
2004-11-07 10:44:37,448 DEBUG
[org.springframework.beans.factory.support.DefaultListableBeanFactory]
Bean with name 'example11Service' is a factory bean
```

请开发者仔细阅读上述日志, 最好能够阅读多次。当然, 结合 `example11.war` 实例运行, 效果最佳。与此同时, 开发者可以去 RDBMS 中查看 `userinfo` 和 `userinterest` 表是否新加了记录。依据上述日志表明, 数据库应该没有新加记录。

另外, 在浏览器中, 将出现如图 7-5 所示页面。

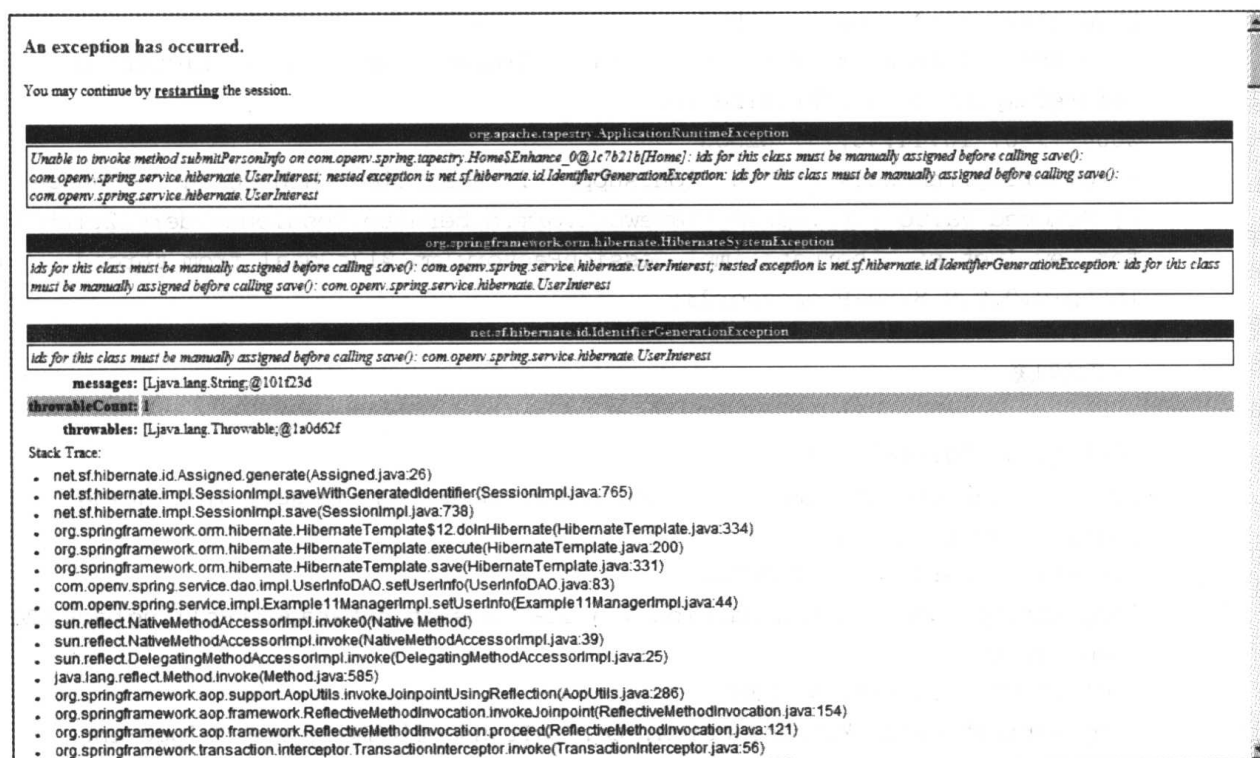


图 7-5 异常页面

对于使用 `HibernateTransactionManager` 事务管理器而言, JBoss 4.0 的 `server.log` 日志文件中会给出相关日志信息。开发者也可以去 MySQL 控制台验证, 是否有新数据插入。

好了, 通过上述过程, 开发者已经对 Spring 使能应用的事务过程有了一定的认识。而且, 这种认识来自于实践。那么, 实践背后的理论基础来自哪里呢? 继续往后阅读, 并查找答案吧!

2. TransactionDefinition

Java 对象本身由属性、方法构成。对于事务而言，它也存在若干事务属性。通常，事务会涉及到事务移植、隔离级别、是否只读、事务有效时限等。这些属性都是通过如下接口定义的：TransactionDefinition。它位于 org.springframework.transaction 包中。

在 CMT 中，开发者可以对事务移植性（propagation）进行配置。在 RDBMS 中，可以配置数据库的隔离度（isolation）。TransactionDefinition 的类图如图 7-6 所示。

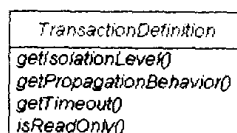


图 7-6 TransactionDefinition 接口定义

对于事务移植而言，TransactionDefinition 定义了 7 种移植策略。

- `int PROPAGATION_REQUIRED = 0`。它表明当前方法必须运行在事务中。这同 EJB 中的声明式事务语义相同。即，如果调用其客户处于事务中，则该方法直接使用该事务；否则，启动新的事务。
- `int PROPAGATION_SUPPORTS = 1`。它表明当前方法可以运行在事务中，也可以不运行在事务中。如果调用其客户处于事务中，则当前方法处于事务中。否则，当前方法不处于事务中。
- `int PROPAGATION_MANDATORY = 2`。它表明当前方法必须运行在事务中。如果调用其客户不处于事务中，则将抛出异常。
- `int PROPAGATION_REQUIRES_NEW = 3`。它表明当前方法必须运行在新事务中。无论客户如何，这都将新启事务。
- `int PROPAGATION_NOT_SUPPORTED = 4`。它表明当前方法不能够运行在事务中。如果调用其客户处于事务中，则在执行到它时，客户所处的事务必须挂起。如果调用其客户不处于事务中，则一切照旧。
- `int PROPAGATION_NEVER = 5`。它表明调用当前方法的客户不能够处于事务中。否则，将抛出异常。
- `int PROPAGATION_NESTED = 6`。它表明当前方法支持嵌入式事务，即允许事务嵌套。标准的 J2EE 平台不要求支持嵌套事务。这是对 EJB 声明式事务的扩展。通常，低端的 JTA 实现都不支持这种事务移植性。

对于隔离级别，Spring TransactionDefinition 定义了如下 5 种事务策略。

- `int ISOLATION_DEFAULT = -1`。它表明直接使用底层持久化源使用的隔离级别定义。
- `int ISOLATION_READ_UNCOMMITTED =`
`Connection.TRANSACTION_READ_UNCOMMITTED`。同 `Connection` 定义的 `TRANSACTION_READ_UNCOMMITTED` 定义。
- `int ISOLATION_READ_COMMITTED =`
`Connection.TRANSACTION_READ_COMMITTED`。含义同 `Connection` 定义的 `TRANSACTION_READ_COMMITTED`。

- `int ISOLATION_REPEATABLE_READ =`
`Connection.TRANSACTION_REPEATABLE_READ`。含义同 `Connection` 定义的 `TRANSACTION_REPEATABLE_READ`。
- `int ISOLATION_SERIALIZABLE = Connection.TRANSACTION_SERIALIZABLE`。
含义同 `Connection` 定义的 `TRANSACTION_SERIALIZABLE`。

对于只读属性而言, `Spring TransactionDefinition` 仅对当前方法新创建的事务有效。通过“`readonly`”能够标识它。在一些场合设置 `readonly` 很有效, 比如在使用 `Hibernate` 时, 如果只是从 `RDBMS` 读取数据, 则这将提高事务效率(避免不必要的 `flush` 操作)。

对于事务有效时限而言, `Spring TransactionDefinition` 仅对当前方法新创建的事务有效。通过设置以“`timeout_`”开头的字符串能够生效有效时限属性。比如, “`timeout_60`”表明事务的有效时限为 60 秒。默认时, 它会使用底层事务系统的超时设置, 即:

- `int TIMEOUT_DEFAULT = -1`。

上述就是 `Spring TransactionDefinition` 的定义, 这也是 `Spring` 所支持的事务属性设置。开发者应该根据实际产品开发环境和部署环境合理设置它们。

再次回到 `example11`, 结合实例阐述各个概念及其使用最直接, 也是最有效的方式。

首先, 本书将 `Spring` 配置文件, 即 `applicationContext.xml` 中相关的一段内容再次提取出来。

```
<bean id="example11Service" class="org.springframework.transaction.  
    interceptor.TransactionProxyFactoryBean">  
    <property name="transactionManager">  
        <ref local="transactionManager"/>  
    </property>  
    <property name="target">  
        <ref local="example11ServiceTarget"/>  
    </property>  
    <property name="transactionAttributes">  
        <props>  
            <prop key="get*">  
                PROPAGATION_REQUIRED,readOnly  
            </prop>  
            <prop key="set*">  
                PROPAGATION_REQUIRED  
            </prop>  
        </props>  
    </property>  
</bean>
```

请注意粗体部分, 其中对事务的移植性进行了约束。其中, 通过 `transactionManager` 属性指定事务管理器; 通过 `target` 属性指定 `TransactionProxyFactoryBean` 代理的目标业务对象; 通过 `transactionAttributes` 属性指定事务策略。在 `Spring` 框架中, 特意提供了 `org.springframework.transaction.interceptor.TransactionAttributeEditor`, 供实现事务策略的手工编辑使用。因此, 开发者借助于该属性编辑器实现事务策略的自定义, 它使得开发者自定义事务策略的过程更为友好。上述配置内容的具体解释如下:

- 对于“get*”：这表明以 get 开头的业务方法（前提是，位于 example11ServiceTarget 中定义的 JavaBean）必须处于事务中，这是通过 PROPAGATION_REQUIRED 限定的。如果调用客户不处在事务上下文中，或者没有提供事务支持，则会创建新的事务。其中，readOnly 表明这只是只读事务，这对于 Hibernate 而言特别有意义。
- 对于“set*”：这表明以 set 开头的业务方法（前提是位于 example11ServiceTarget 中定义的 JavaBean）必须处于事务中，这是通过 PROPAGATION_REQUIRED 限定的。如果调用客户不处在事务上下文中，或者没有提供事务支持，则会创建新的事务。
- 其中设置的事务属性顺序并不重要，因为 TransactionAttributeEditor 会自动处理它们。比如，“PROPAGATION_REQUIRED,readonly” get* 属性取值同 “readonly, PROPAGATION_REQUIRED” 含义相同。

因此，借助于 Spring 提供的声明式事务支持，开发者能够获得如下几方面的益处。

- 借助于 Spring AOP 提供了类似 EJB CMT 的功能。而且，能够对 POJO 进行事务性声明。
- Spring 框架提供的声明式事务能够适用于任何操作环境。比如，JDBC、Hibernate、JDO 等。如果有新的事务性资源的加入，对于 Spring 应用而言，只不过是配置文件需要修改而已。
- Spring 还提供了声明式的回滚规则。比如，可以声明：如果某异常一旦抛出，事务必须回滚。比如，在事务属性中设置“-UserException”，则表明在事务处理过程中一旦出现 UserException，则回滚当前事务。再比如，在事务属性中设置“+UserException”，则表明在事务处理过程中一旦出现 UserException，则提交当前事务。加号表示提交，减号表示回滚。因此，这是很诱人的技术。

当然，Spring 只是实现了对其他事务管理的抽象，自身并没有实现事务管理，开发者一定要明白这个道理。

另外，Spring 还提供了通过元数据（比如，借助于 Jakarta Commons Attributes）、指定方法名（org.springframework.transaction.interceptor.NameMatchTransactionAttributeSource）、自动代理等方式定义事务策略。无论如何，只要开发者掌握了 Spring 所支持的事务策略，即可选用自己喜欢或者擅长的配置方式。本书就不再讨论这些方法，具体内，开发者可以参考 Spring 源码。本书第2章详细介绍了获取、编译 Spring 项目的过程。

通常，在使用 Open Source 项目开发企业应用过程中，通过阅读源码理解 Open Source 项目的产生背景、内在机理很有帮助。尤其是在开发者对自身感兴趣的项目有所了解后，比如 Spring，再来阅读源码，则效果更佳。其实，阅读源码的过程，对于开发者而言，本身就是一个学习、提高的过程。

7.2.3 编程式事务

编程式事务比声明式事务更具灵活性。由于 Spring AOP 实现的拦截器只能拦截方法级的调用，因此如果需要事务支持，则整个方法处于事务中。如果存在遗留代码需要借助于 Spring 管理，则此时 Spring 可能不能够胜任。但无论如何，如果我们架构并设计新的企

业应用系统，可以考虑到 Spring 的这种特性，更何况这也是开发者大部分场合需要处理的情况。

Spring 为编程式事务提供了两种策略。

- 直接使用 `TransactionTemplate`：这同 `JndiTemplate`、`JdbcTemplate`、`HibernateTemplate` 类似。由于 `TransactionTemplate` 是线程安全的，因此开发者可以大胆地使用。
- 直接使用 `PlatformTransactionManager` 实现。开发者可以推导，`TransactionTemplate` 一定需要使用 `PlatformTransactionManager` 实现，否则它无法完成事务管理操作。因此，直接借助于 `PlatformTransactionManager` 实现一定也能够完成事务的管理。当然，直接借助于 `PlatformTransactionManager` 实现管理事务，会比使用事务模板完成更多的设置和准备工作。从这个角度考虑，Spring 开发者应该使用 `TransactionTemplate`。

如果开发者直接使用 `PlatformTransactionManager` 实现，则这同直接使用 JTA API 差不多。Spring 框架开发 Team 推荐：通常都最好使用 `TransactionTemplate`。

1. 使用 `TransactionTemplate`

Spring 框架提供的、类似 `TransactionTemplate` 的模板有很多。通常，它们是采用回调机制实现方法调用的。在方法调用期间，`TransactionTemplate` 会将要求的资源准备好（调用开始）和释放掉（调用结束），因此，这就是模板的威力。在 Spring 中，所有的模板都是线程安全的²。

在使用 `TransactionTemplate` 之前，开发者需要在 Spring 配置文件中配置好它（否则不能够启动 Spring 使能应用）。比如，下面给出了配置示例。请注意，`transactionManager` 是 `TransactionTemplate` 的属性，因此事务模板需要借助于 Spring 提供的事务管理器实现处理事务管理。可以看出，事务模板对事务管理器并没有过分的要求，无论是 `Hibernate`、`JTA`、`JDBC`、`JDO` 还是其他类型的事务管理器，它都能够接受。

```
<bean id="transactionTemplate"

class="org.springframework.transaction.support.TransactionTemplate">
    <property name="transactionManager">
        <ref local="transactionManager"></ref>
    </property>
</bean>

<bean id="transactionManager"

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

² 作者维护的 <http://www.open-v.com> 网站上提供了《开发线程安全的 Spring Web 应用》一篇优秀的译文，其中有对 `HibernateTemplate` 的介绍。这对于了解和掌握 Spring Web 应用的线程安全很有帮助。

当然，开发者也可以基于 `TransactionTemplate` 的构建器来注入对事务管理器的依赖。接下来，开发者便能够在 Java 代码中直接使用 `transactionTemplate` 了。依据 Spring 模板设计原则，Spring 为 `TransactionTemplate` 准备了两个回调接口：`TransactionCallback` 和 `TransactionCallbackWithoutResult`。

- `org.springframework.transaction.support.TransactionCallback`：用于事务的回调接口。通常，它都是伴随 `TransactionTemplate` 并以匿名方式使用的。它仅含有单个方法，即 `doInTransaction`。如果事务操作存在返回值，则使用 `TransactionCallback`。如果不存在返回值，则使用 `TransactionCallbackWithoutResult` 接口。
- `org.springframework.transaction.support.TransactionCallbackWithoutResult`：继承于 `TransactionCallback`。`TransactionCallbackWithoutResult` 供事务操作不存在返回值时使用。

对于存在返回值的业务方法而言，示例如下。

```
Object result = tt.execute(new TransactionCallback() {
    public Object doInTransaction(TransactionStatus status) {
        try {
            updateOperation1();
            updateOperation2();
            return updateOperation3();
        } catch (RuntimeException re) {
            status.setRollbackOnly();
        }
        return null;
    }
});
```

其中，开发者需要实现 `Object doInTransaction(TransactionStatus status)` 方法。现在，`updateOperation1()`、`updateOperation2()`、`updateOperation3()` 都处于同一事务中，因此它们构成了 ACID 了的特性。如果运行期间出现了 `Exception` 异常，则事务回滚（借助于 `status` 的 `setRollbackOnly` 实现）。

对于不存在返回值的业务方法而言，示例如下：

```
Object result = tt.execute(new TransactionCallbackWithoutResult() {
    public void doInTransactionWithoutResult(TransactionStatus status) {
        updateOperation1();
        updateOperation2();
        updateOperation3();
    }
});
```

如果应用需要回滚，则需要调用 `TransactionStatus` 对象的 `setRollbackOnly` 方法。对于有 EJB 组件开发经验的开发者而言，能够看出这同 CMT 极其相似。

2. 使用 PlatformTransactionManager 实现

借助于 Spring IoC 容器提供的控制反转，能够在应用代码中直接获得对 `<bean>` 元素中定义的 `PlatformTransactionManager` 实现。示例如下：

```
transactionManager = ....;
```

```
DefaultTransactionDefinition dtd = new DefaultTransactionDefinition()
dtd.setPropagationBehavior(TransactionDefinition.PROPGATION_REQUIRED);

TransactionStatus status = transactionManager.getTransaction(dtd);

try {
    updateOperation1();
    updateOperation2();
    updateOperation3();
}
catch (ApplicationException ex) {
    transactionManager.rollback(status);
    throw ex;
}

transactionManager.commit(status);
```

注意，这是在模拟 `TransactionTemplate` 的行为。因此，在程式事务中，最好直接使用 `TransactionTemplate`。

7.3 小 结

本章深入地探讨了 `Spring` 对事务管理的支持。对于保证业务操作的 `ACID`，事务管理起到了很关键的作用。对于企业级应用而言，没有事务管理的支持，结果不堪设想。当然，`Spring` 提供的事务抽象使得这些都不是问题，正如开发者看到的一样。

对于那些仅含有少量事务操作的企业级应用而言，本文推荐采用程式事务开发模型。因为对于声明式事务开发模型而言，开发者需要对 `Spring AOP` 模块较熟悉，而且这种应用确实也没有必要采用声明式事务开发模型。反之，对于那些包含大量事务操作的企业级应用而言，使用声明式事务开发模型是明智之举。因为这种应用本身的业务逻辑较复杂，应用代码本身需要更多地关注业务逻辑，而且在 `Spring` 中配置这种事务声明很简单。

开发者需要正确使用 `PlatformTransactionManager` 接口及其实现，正如本章给出的一样。至于 `JTA` 事务管理的更深入介绍，请参考《`Java Transaction Processsing Design and Implementation`》³一书，由 `Mark Little` 等人写作而成，`PTR` 出版社于 2004 年出版发行。这本书对于学习 `Java` 中的事务处理很适合。

³ <http://www.phptr.com/title/013035290X#>

第 8 章 消息服务——JMS

JMS，即 Java 消息服务。在 JMS 中 JMS 消息并不同应用直接交互，而是同 JMS 服务器的目的地（destination）进行交互，比如点对点（P2P）或者发布/订阅（Publish/Subscribe）。对于发送消息的应用程序而言，它不需要关注接收消息的应用程序是否在正常工作；对于接收消息的应用程序而言，它不需要关注发送消息的应用程序是否在正常工作。因此，它们仅仅同目的地进行交互。

Spring 提供了 JMS 服务抽象框架，以简化对 JMS API 的使用。同时，还能够将 JMS 1.0.2 和 JMS 1.1 API（在 J2EE 1.4 中引入）的差异性屏蔽掉。

8.1 背 景

对于 J2EE 环境而言，消息驱动 Bean 能够异步处理消息。当然，单独客户应用也能够处理消息，比如那些实现了 `MessageListener` 接口的应用或类。

另外，单独客户应用还能够生成消息，并发送到消息服务器的目的地。那么，何为 JMS 呢？

JMS，是能够异步处理客户请求的开发模型。为支持消息的异步处理，Java/J2EE 平台引入了 JMS API。JMS 支持两种消息模型：点对点和发布/订阅模式。点对点模式的含义是：消息生产者将消息发布到 Queue 中，在随后的操作中将会有消息消费者从该 Queue 中将消息消费掉。一旦消息消费掉后，Queue 中将不再存在它，因此其他消息消费者将不能够获得已消费的消息。注意，尽管 Queue 支持同时存在多个消息消费者，但对于单个的消息而言，仅仅会有一个消息消费者消费它。发布/订阅模式的含义是：消息生产者（发布者）将消息发布到 Topic 中，与此同时将会有多个消息消费者（订阅者）消费发布到该 Topic 的消息。它和点对点不同，即发布到 Topic 的消息将会被所有已订阅该 Topic 的订阅者消费。注意，如果消息发往 Topic 中时，当时不存在消息监听者（即，订阅者不处于监听状态），则该消息将丢弃掉。但有一点特殊，如果某订阅者是持久订阅者，则该消息将保留到该持久订阅者消费掉该消息为止。

有关 JMS 的深入介绍资料有很多，本书不再详细给出。但是有一点需要强调，即有关 JMS 1.0.2 和 JMS 1.1 的区别。对于 JMS 1.0.2 而言，定义了两种消息目的地类型，即点对点（Point-to-Point, Queue）和发布/订阅（Publish/Subscribe, Topic）。因此，借助于 JMS 1.0.2 提供的 API 就需要依据这两种目的地类型而使用不同的 API，比如 `TopicConnectionFactory` 和 `QueueConnectionFactory`。在 JMS 1.1 中，`TopicConnectionFactory` 和 `QueueConnectionFactory` 合并为 `ConnectionFactory`。从开发者使用 JMS API 开发消息应用的角度考虑，在 JMS 1.1 中再也不用考虑具体的目的地类型了。

同时，借助于 JMS 1.0.2 API 在同一 Session 中只能够同某类型的目的地进行交互。针

对这种情况，JMS 1.1 消除了这种开发限制，比如在同一 Session 中能够同不同类型的目的地进行交互，从而更好地实现事务控制。

8.2 Spring 对 JMS 提供的支持

JMS，能够保证客户/服务器间的可靠性，即保证消息消费者、生产者确实消费或者生产了某消息。这对于企业应用而言至关重要。可靠性不仅仅需要从应用层面保证，更多地是从系统级、中间层得以保证。在 Web 服务发展迅猛的今天，也存在类似的可靠性规范。如果 JMS 的服务质量（QoS）得不到保证，则这种松耦合的架构必然存在很多问题。无论如何，JMS 都是架构大型企业应用的主要利器之一。

Spring 对这种优秀的 J2EE API 肯定不能够置之不理。因此，它提供了 Spring JMS 抽象服务。

从 Spring 框架中 jms 相关包结构看，主要包括如下内容。

- `org.springframework.jms`: 定义 JMS 异常，以便于使用 JMS。这些 JMS 异常都是未受查的 `RuntimeException` 异常，因此开发者在应用中有更多选择来处理应用抛出的 JMS 异常。
- `org.springframework.jms.connection`: `connection` 包提供 `PlatformTransactionManager` 和 `SingleConnectionFactory`。比如，`JmsTransactionManager` 事务管理器用于 JMS 1.1；`JmsTransactionManager102` 事务管理器用于 JMS 1.0.2。
- `org.springframework.jms.core`: Spring 提供的 JMS 抽象框架的核心内容。其中，含有 `JmsTemplate` 和若干个回调接口，这同用于 JDBC 的 `JdbcTemplate`（当然也包括为其他 J2EE API 提供的模板方法实现）类似。Spring JMS 服务抽象分别为 JMS 1.1 和 1.0.2 提供了 `JmsTemplate` 和 `JmsTemplate102` 模板。借助于模板操作 JMS API，使得发送、消费消息变得简洁高效。其中，开发者再也不用手工处理受查 `javax.jms.JMSEException` 异常了。
- `org.springframework.jms.core.support`: 支持 `core` 包。
- `org.springframework.jms.support`: 提供辅助类，比如 `JmsUtils`。`JmsUtils` 负责将受查的 `JMSEException` 异常转换成 `org.springframework.jms` 包中定义的未受查运行时的异常。借助于 `JmsUtils` 中的 `convertJmsAccessException()` 静态方法能够完成异常的转换。比如，将 `javax.jms.JMSSecurityException` 转换为 `JmsSecurityException` 异常。
- `org.springframework.jms.support.converter`: 提供消息转换抽象，主要用于 Java 对象和 JMS 消息之间的转换。在实际应用中，开发者需要为 JMS 消息类型和业务对象提供转换。比如，开发者需要将 JMS `MapMessage` 消息类型转化为某业务实体，比如 `OrderEntry`。其中，`OrderEntry` 含有 `MapMessage` 消息中各个 key 对应的值。为此，Spring 为保证应用代码的简洁性，特提供了 `converter` 包，供完成消息转换使用。开发者能够实现其提供的 `MessageConverter` 接口，或者直接使用其提供的 `SimpleMessageConverter`（`SimpleMessageConverter102`）。

- `org.springframework.jms.support.destination`: 提供目的地管理辅助。

对于 `org.springframework.jms` 而言, 它同 Spring 框架提供的 DAO 抽象一样, 提供了类似的异常处理类结构, 如图 8-1 所示。

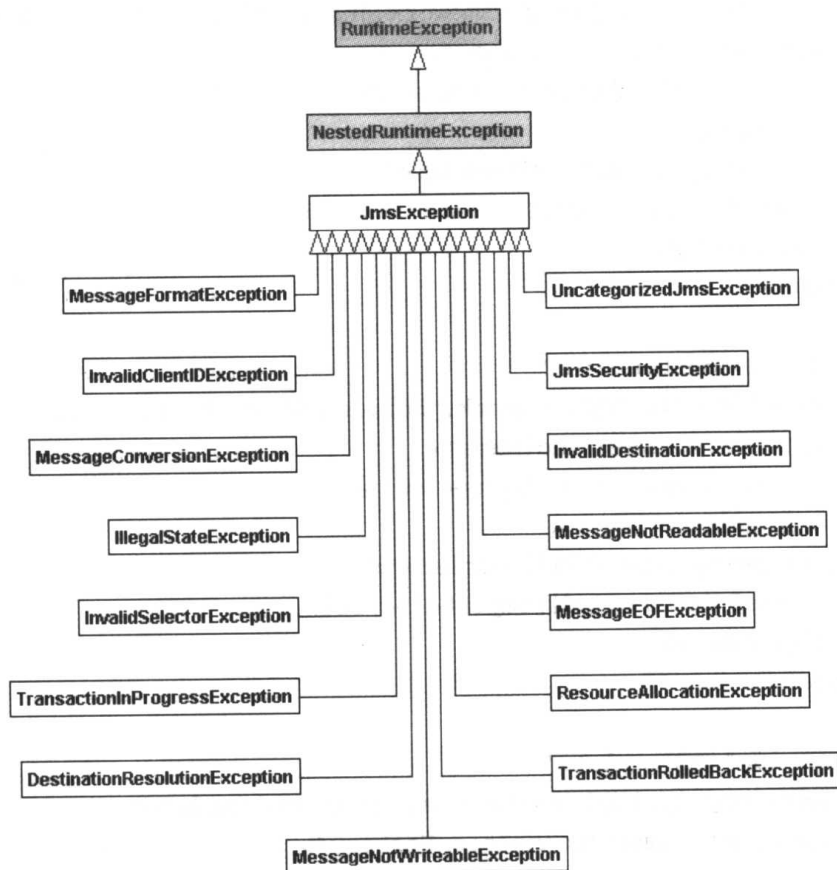


图 8-1 JmsException 异常及其子类

其中, `JmsException` 及其子类都为未受查运行时异常, 这同 Spring DAO 抽象类似。因此, Spring 为 JMS 提供的服务抽象使得开发者不用处理 JMS API 提供的受查 `javax.jms.JMSEException` 异常, 因为在消费或生产消息时产生的 `JMSEException` 异常将会被 Spring JMS 服务抽象转换为 `org.springframework.jms.JmsException`。此时, 开发者可以考虑处理 `JmsException`, 或者干脆不予理睬。

8.2.1 JmsTemplate

1. 发送消息

还是遵循本书的传统吧, 在实例中发现问题、解决问题。因此, 首先来看看 example14 能够给开发者带来什么内容。

由于上述谈到, JMS 1.0.2 与 JMS 1.1 在支持的 JMS API 方面有所不同。在此, 先看看 `JmsTemplate` 模板的使用, 即 JMS 1.1。了解一下 `appcontext11.xml` 的配置。

```

<?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="connectionFactory"
        class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName">
            <value>ConnectionFactory</value>
        </property>
        <property name="jndiTemplate">
            <ref local="jndiTemplate"></ref>
        </property>
    </bean>

    <bean id="destination"
        class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName">
            <value>queue/testQueue</value>
        </property>
        <property name="jndiTemplate">
            <ref local="jndiTemplate"></ref>
        </property>
    </bean>

    <bean id="jmsTemplate"
        class="org.springframework.jms.core.JmsTemplate">
        <property name="connectionFactory">
            <ref local="connectionFactory"/>
        </property>
        <property name="defaultDestination">
            <ref local="destination" />
        </property>
    </bean>

    <bean id="jndiTemplate"
        class="org.springframework.jndi.JndiTemplate">
        <property name="environment">
            <props>
                <prop key="java.naming.factory.initial">
                    org.jnp.interfaces.NamingContextFactory
                </prop>
                <prop key="java.naming.provider.url">
                    jnp://localhost:1099
                </prop>
                <prop key="java.naming.factory.url.pkgs">
                    org.jboss.naming:org.jnp.interfaces
                </prop>
            </props>
        </property>
    </bean>
</beans>
```

```

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

```

```

</beans>

```

其中，上述 Spring 配置文件定义了目的地（destination），即“queue/testQueue”；定义了连接工厂（connectionFactory）。请开发者注意，它们都是通过 Spring JNDI 抽象服务中的 org.springframework.jndi.JndiObjectFactoryBean 获得的。最后，定义了同客户应用交互的 jmsTemplate，即 org.springframework.jms.core.JmsTemplate，它引用了上述目的地和连接工厂 JavaBean。

客户应用代码如下：

```

package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

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

import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;

import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.Session;

/**
 * JmsTemplateTest11 客户应用
 *
 * @author luoshifei
 */
public class JmsTemplateTest11 {
    protected static final Log log =
LogFactory.getLog(JmsTemplateTest11.class);

    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "appcontext11.xml");

        JmsTemplate jt = (JmsTemplate) ac.getBean("jmsTemplate");
        jt.send(new MessageCreator() {
            public Message createMessage(Session session)
                throws JMSEException {
                return session.createTextMessage(
                    "Hello World(11),queue/testQueue!");
            }
        });
    }
}

```

```
    }  
    });  
    log.info("成功发送消息!");  
}  
}
```

开发者很容易看出, 在 Spring 框架提供的 JMS 抽象中, `JmsTemplate` 同 `MessageCreator` 结合能够发送消息。开发者需要实现 `MessageCreator` 回调接口定义的 `createMessage` 方法。通过执行 `Ant build.xml` 中的 `run11` 任务, 能够看到如下结果。

```
Buildfile: D:\workspace\example14\build.xml  
compile:  
run11:  
    [java] 2004-12-25 18:11:10 org.springframework.beans.factory.xml.  
XmlBeanDefinitionReader loadBeanDefinitions  
    [java] 信息: Loading XML bean definitions from class path resource  
[appcontext11.xml]  
    [java] 2004-12-25 18:11:11 org.springframework.context.support.  
AbstractXmlApplicationContext refreshBeanFactory  
    [java] 信息: Bean factory for application context [org.springframework.  
context.support.ClassPathXmlApplicationContext;hashCode=22540508]:  
org.springframework.beans.factory.support.DefaultListableBeanFactory  
defining beans [connectionFactory,destination,jmsTemplate,jndiTemplate];  
root of BeanFactory hierarchy  
    [java] 2004-12-25 18:11:11 org.springframework.context.support.  
AbstractApplicationContext refresh  
    [java] 信息: 4 beans defined in application context [org.springframework.  
context.support.ClassPathXmlApplicationContext;hashCode=22540508]  
    [java] 2004-12-25 18:11:11 org.springframework.context.support.  
AbstractApplicationContext initMessageSource  
    [java] 信息: No message source found for context [org.springframework.  
context.support.ClassPathXmlApplicationContext;hashCode=22540508]: using  
empty default  
    [java] 2004-12-25 18:11:11 org.springframework.context.support.  
AbstractApplicationContext initApplicationEventMulticaster  
    [java] 信息: No ApplicationEventMulticaster found for context  
[org.springframework.context.support.ClassPathXmlApplicationContext;hashC  
ode=22540508]: using default  
    [java] 2004-12-25 18:11:11 org.springframework.context.support.  
AbstractApplicationContext refreshListeners  
    [java] 信息: Refreshing listeners  
    [java] 2004-12-25 18:11:11 org.springframework.beans.factory.support.  
DefaultListableBeanFactory preInstantiateSingletons  
    [java] 信息: Pre-instantiating singletons in factory  
[org.springframework.beans.factory.support.DefaultListableBeanFactory  
defining beans [connectionFactory,destination,jmsTemplate,jndiTemplate];  
root of BeanFactory hierarchy]  
    [java] 2004-12-25 18:11:11 org.springframework.beans.factory.support.
```

```

AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'connectionFactory'
    [java] 2004-12-25 18:11:11 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'jndiTemplate'
    [java] 2004-12-25 18:11:13 org.springframework.jndi.JndiLocatorSupport lookup
    [java] 信息: Located object with JNDI name [ConnectionFactory]:
value=[org.jboss.mq.SpyConnectionFactory@17a29a1]
    [java] 2004-12-25 18:11:13 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'destination'
    [java] 2004-12-25 18:11:13 org.springframework.jndi.JndiLocatorSupport lookup
    [java] 信息: Located object with JNDI name [queue/testQueue]:
value=[QUEUE.testQueue]
    [java] 2004-12-25 18:11:13 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'jmsTemplate'
    [java] 2004-12-25 18:11:14 com.openv.spring.JmsTemplateTest11 main
    [java] 信息: 成功发送消息!
BUILD SUCCESSFUL
Total time: 8 seconds

```

通过查看 JBoss 提供的 JMX 控制台应用, 开发者能够浏览到发送到 JBossMQ 的消息内容。具体操作步骤如下。

步骤

(1) 打开 <http://localhost:8080/jmx-console/>。可以浏览到 jboss.mq.destination 部分, 如图 8-2 所示。

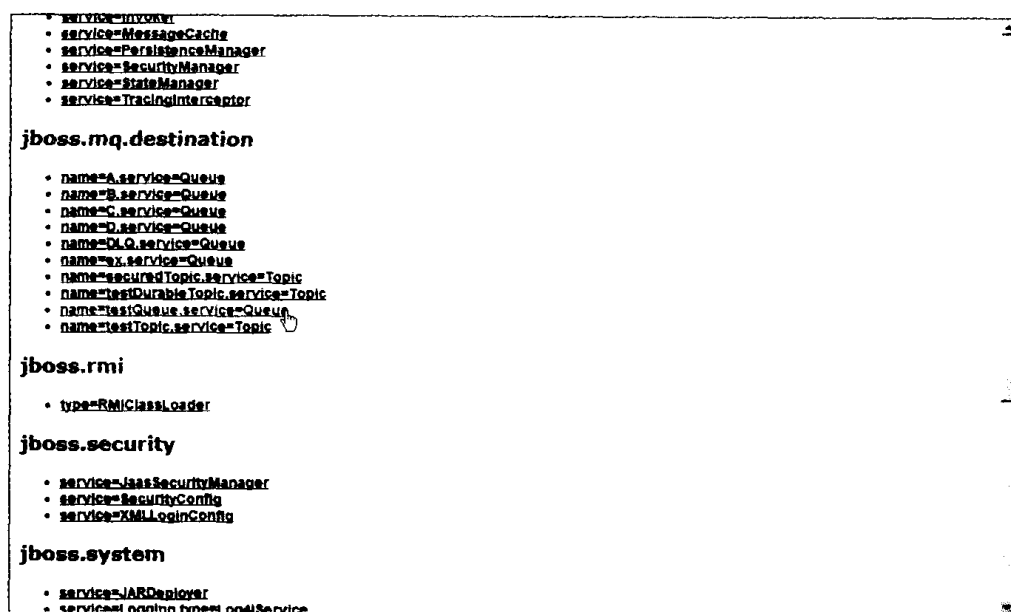


图 8-2 jmx-console 应用中的 jboss.mq.destination 部分

(2) 单击 “name=testQueue,service=Queue”，其界面如图 8-3 所示。

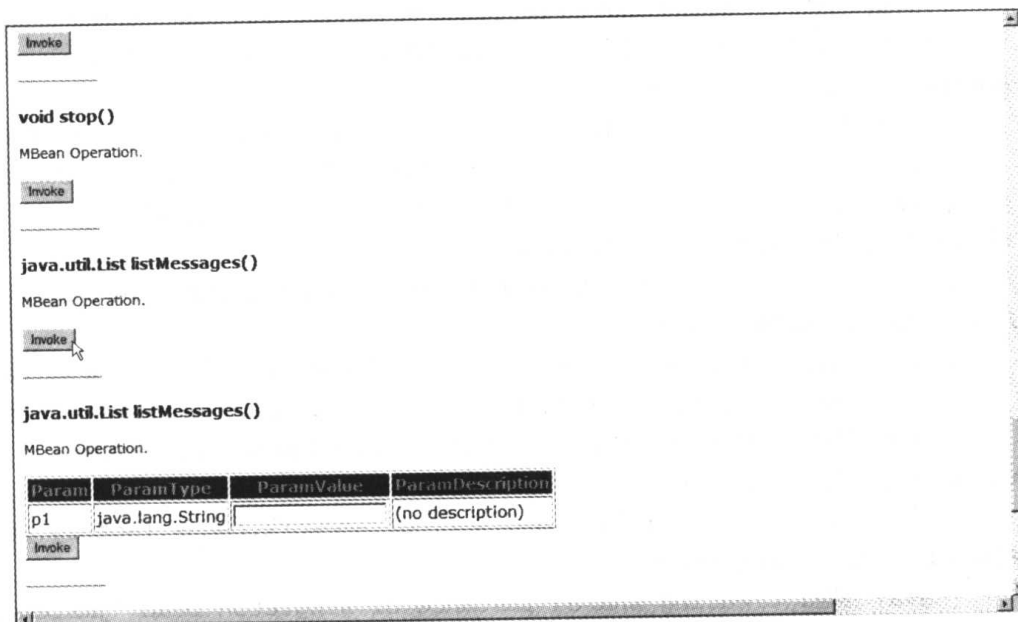


图 8-3 单击 “name=testQueue,service=Queue” 后打开的界面

(3) 调用 listMessages 方法，这时开发者能够浏览到发送到 JBossMQ 的消息内容，如图 8-4 所示。

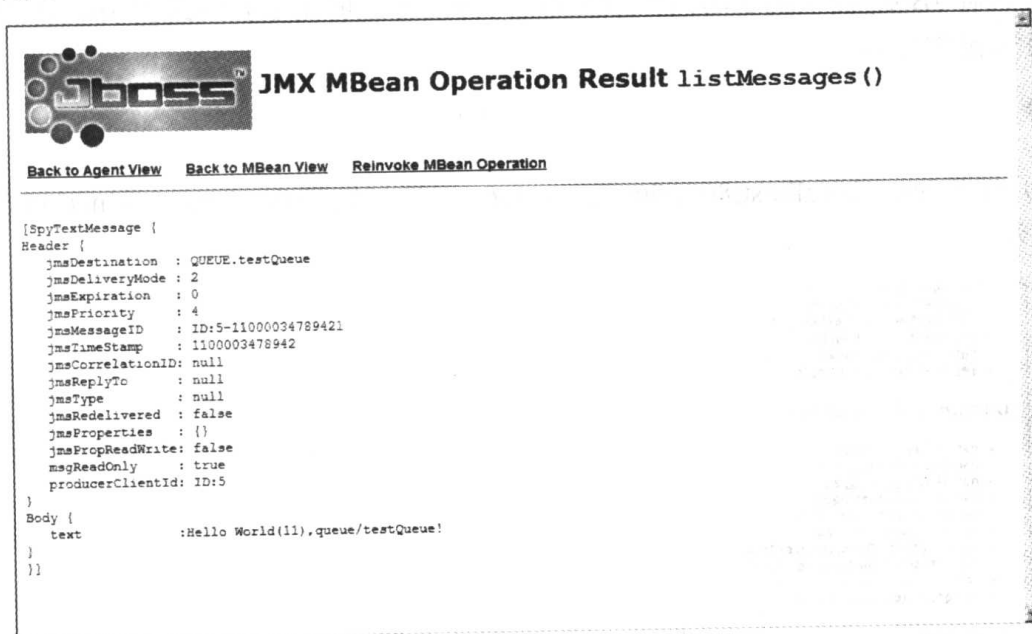


图 8-4 发送到 JBossMQ 的消息内容

因此，在 Spring 框架提供的 JMS 抽象的帮助下，发送 JMS 消息变得如此简单。对于 Web 应用而言，情况也是类似。其中，为设置 Queue/Topic 目的地，开发者除了可以通过设置 defaultDestination 属性外，还可以通过设置 destinationResolver 属性，即通过 JndiDestinationResolver 实例获得目的地，从而不需要使用 JndiObjectFactoryBean。这对于那些含有大量目的地的应用而言特别有效。

另外，有一点需要开发者关注。如果开发者未提供 defaultDestination 属性。则在使用

JmsTemplate 发送消息时，需要指定目的地（比如，“queue/testQueue”）。

至于 JmsTemplate102，它继承于 JmsTemplate（图 8-5 给出了相应的类图）。

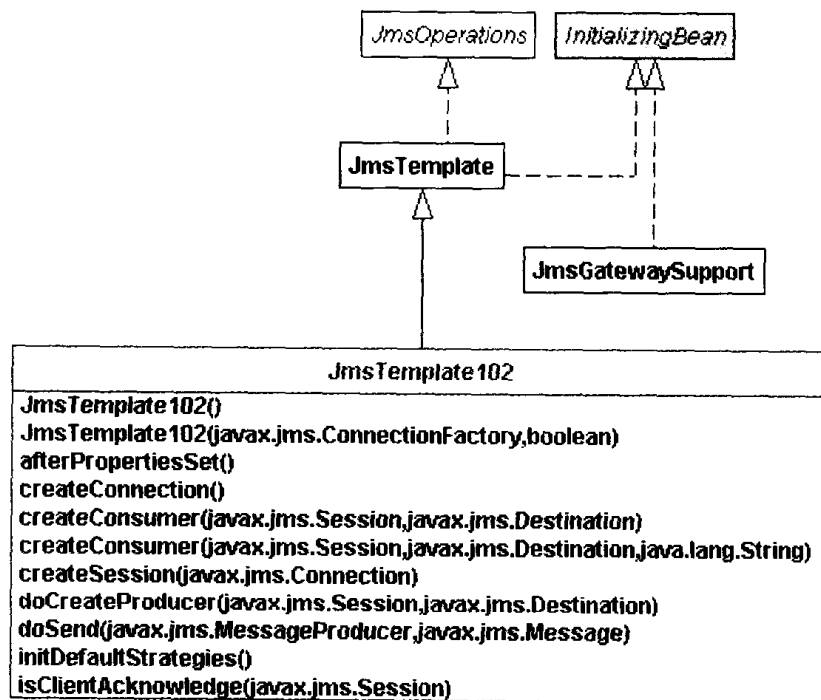


图 8-5 JmsTemplate 和 JmsTemplate102

为使用 JmsTemplate102，开发者需要将 appcontext11.xml 和 JmsTemplateTest11.java 相应的部分替换成 JmsTemplate102。其中，appcontext102.xml 如下。

```

<?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="connectionFactory"
        class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName">
            <value>ConnectionFactory</value>
        </property>
        <property name="jndiTemplate">
            <ref local="jndiTemplate"></ref>
        </property>
    </bean>

    <bean id="destination"
        class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName">
            <value>queue/testQueue</value>
        </property>
        <property name="jndiTemplate">

```

```
        <ref local="jndiTemplate"></ref>
    </property>
</bean>

<bean id="jmsTemplate"
    class="org.springframework.jms.core.JmsTemplate102">
    <property name="connectionFactory">
        <ref local="connectionFactory"/>
    </property>
    <property name="defaultDestination">
        <ref local="destination" />
    </property>
</bean>

<bean id="jndiTemplate"
    class="org.springframework.jndi.JndiTemplate">
    <property name="environment">
        <props>
            <prop key="java.naming.factory.initial">
                org.jnp.interfaces.NamingContextFactory
            </prop>
            <prop key="java.naming.provider.url">
                jnp://localhost:1099
            </prop>
            <prop key="java.naming.factory.url.pkgs">
                org.jboss.naming:org.jnp.interfaces
            </prop>
        </props>
    </property>
</bean>

</beans>
```

对应的 `JmsTemplateTest102.java` 源文件如下。

```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

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

import org.springframework.jms.core.JmsTemplate102;
import org.springframework.jms.core.MessageCreator;

import javax.jms.JMSEException;
import javax.jms.Message;
```



```
import javax.jms.Session;

/**
 * JmsTemplateTest102 客户应用
 *
 * @author luoshifei
 */
public class JmsTemplateTest102 {
    protected static final Log log =
        LogFactory.getLog(JmsTemplateTest102.class);

    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "appcontext102.xml");

        JmsTemplate102 jt = (JmsTemplate102) ac.getBean("jmsTemplate");
        jt.send(new MessageCreator() {
            public Message createMessage(Session session)
                throws JMSEException {
                return session.createTextMessage(
                    "Hello World(102),queue/testQueue!");
            }
        });
        log.info("成功发送消息!");
    }
}
```

执行结果如下。

```
Buildfile: D:\workspace\example14\build.xml
compile:
run102:
[java] 2004-12-25 18:16:18 org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
[java] 信息: Loading XML bean definitions from class path resource
[appcontext102.xml]
[java] 2004-12-25 18:16:19 org.springframework.context.support.
AbstractXmlApplicationContext refreshBeanFactory
[java] 信息: Bean factory for application context [org.springframework.
context.support.ClassPathXmlApplicationContext;hashCode=22540508]:
org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [connectionFactory,destination,jmsTemplate,jndiTemplate];
root of BeanFactory hierarchy
[java] 2004-12-25 18:16:19 org.springframework.context.support.
AbstractApplicationContext refresh
[java] 信息: 4 beans defined in application context [org.springframework.
context.support.ClassPathXmlApplicationContext;hashCode=22540508]
```

```
[java] 2004-12-25 18:16:19 org.springframework.context.support.  
AbstractApplicationContext initMessageSource  
[java] 信息: No message source found for context [org.springframework.  
context.support.ClassPathXmlApplicationContext;hashCode=22540508]: using  
empty default  
[java] 2004-12-25 18:16:19 org.springframework.context.support.  
AbstractApplicationContext initApplicationEventMulticaster  
[java] 信息: No ApplicationEventMulticaster found for context  
[org.springframework.context.support.ClassPathXmlApplicationContext;hashC  
ode=22540508]: using default  
[java] 2004-12-25 18:16:19 org.springframework.context.support.  
AbstractApplicationContext refreshListeners  
[java] 信息: Refreshing listeners  
[java] 2004-12-25 18:16:19 org.springframework.beans.factory.support.  
DefaultListableBeanFactory preInstantiateSingletons  
[java] 信息: Pre-instantiating singletons in factory  
[org.springframework.beans.factory.support.DefaultListableBeanFactory  
defining beans [connectionFactory,destination,jmsTemplate,jndiTemplate];  
root of BeanFactory hierarchy]  
[java] 2004-12-25 18:16:19 org.springframework.beans.factory.support.  
AbstractBeanFactory getBean  
[java] 信息: Creating shared instance of singleton bean 'connectionFactory'  
[java] 2004-12-25 18:16:19 org.springframework.beans.factory.support.  
AbstractBeanFactory getBean  
[java] 信息: Creating shared instance of singleton bean 'jndiTemplate'  
[java] 2004-12-25 18:16:21 org.springframework.jndi.JndiLocatorSupport lookup  
[java] 信息: Located object with JNDI name [ConnectionFactory]:  
value=[org.jboss.mq.SpyConnectionFactory@d80be3]  
[java] 2004-12-25 18:16:21 org.springframework.beans.factory.support.  
AbstractBeanFactory getBean  
[java] 信息: Creating shared instance of singleton bean 'destination'  
[java] 2004-12-25 18:16:21 org.springframework.jndi.JndiLocatorSupport lookup  
[java] 信息: Located object with JNDI name [queue/testQueue]:  
value=[QUEUE.testQueue]  
[java] 2004-12-25 18:16:21 org.springframework.beans.factory.support.  
AbstractBeanFactory getBean  
[java] 信息: Creating shared instance of singleton bean 'jmsTemplate'  
[java] 2004-12-25 18:16:21 com.openv.spring.JmsTemplateTest102 main  
[java] 信息: 成功发送消息!  
BUILD SUCCESSFUL  
Total time: 7 seconds
```

因此, Spring 框架提供的 JMS 抽象使得操作 JMS 1.0.2 和 JMS 1.1 API 的应用代码基本相同(至少发送消息的代码差不多)。

JmsTemplate 模板在默认时, 使用了 P2P 目的地类型, 即通过 pubSubDomain 属性能够改变目的地类型。

2. 接收消息

消息驱动 Bean 是接收消息的最好方式之一。当然, `JmsTemplate` 也支持消息的接收操作。为演示同步接收消息, 开发者可以使用 `JmsTemplate` 提供的 `receive` 方法。其中, 可以设定 `receiveTimeout` 属性, 以指定线程阻塞的最大时间。注意, 使用 `receive` 方法接收消息时, 请务必小心!

`JmsTemplateTest11Receive` 是能够同步接收消息的客户应用, 其源代码如下。

```
package com.openv.spring;

import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.Session;
import javax.jms.TextMessage;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;

/**
 * JmsTemplateTest11Receive 客户应用
 *
 * @author luoshifei
 */
public class JmsTemplateTest11Receive {
    protected static final Log log = LogFactory
        .getLog(JmsTemplateTest11Receive.class);

    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "appcontext11r.xml");

        JmsTemplate jt = (JmsTemplate) ac.getBean("jmsTemplate");
        jt.send(new MessageCreator() {
            public Message createMessage(Session session)
                throws JMSEException {
                return session
                    .createTextMessage("Hello World(11Receive),
                        queue/testQueue!");
            }
        });
        TextMessage message = (TextMessage) jt.receive();
        try {
            log.info(message.getText());
        }
    }
}
```

```
    } catch (JMSEException ex) {  
        log.error("JMSEException 异常", ex);  
    }  
  
    }  
  
}
```

借助于 `message.getText()` 获得消息内容时, 需要开发者捕获 `javax.jms.JMSEException` 异常, 因此 Apache Commons Logging 打印出错误信息。在此, 开发者也可以借助于 `JmsUtils` 提供的 `convertJmsAccessException` 静态方法将它转换为未受查运行期异常。代码片断如下。

```
try {  
    log.info(message.getText());  
} catch (JMSEException ex) {  
    throw JmsUtils.convertJmsAccessException(ex);  
//    log.error("JMSEException 异常", ex);  
}
```

相应的 Spring 配置文件 (`appcontext11r.xml`) 如下。

```
<?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="connectionFactory"  
        class="org.springframework.jndi.JndiObjectFactoryBean">  
        <property name="jndiName">  
            <value>ConnectionFactory</value>  
        </property>  
        <property name="jndiTemplate">  
            <ref local="jndiTemplate"></ref>  
        </property>  
    </bean>  
  
    <bean id="destination"  
        class="org.springframework.jndi.JndiObjectFactoryBean">  
        <property name="jndiName">  
            <value>queue/testQueue</value>  
        </property>  
        <property name="jndiTemplate">  
            <ref local="jndiTemplate"></ref>  
        </property>  
    </bean>  
  
    <bean id="jmsTemplate"  
        class="org.springframework.jms.core.JmsTemplate">  
        <property name="connectionFactory">  
            <ref local="connectionFactory"/>  
        </property>
```

```
<property name="defaultDestination">
    <ref local="destination" />
</property>
</bean>

<bean id="jndiTemplate"
class="org.springframework.jndi.JndiTemplate">
    <property name="environment">
        <props>
            <prop key="java.naming.factory.initial">
                org.jnp.interfaces.NamingContextFactory
            </prop>
            <prop key="java.naming.provider.url">
                jnp://localhost:1099
            </prop>
            <prop key="java.naming.factory.url.pkgs">
                org.jboss.naming:org.jnp.interfaces
            </prop>
        </props>
    </property>
</bean>

</beans>
```

具体执行结果如下。

Buildfile: D:\workspace\example14\build.xml

compile:

runllreceive:

[java] 2004-12-25 18:20:42 org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions

[java] 信息: Loading XML bean definitions from class path resource
[appcontext11r.xml]

[java] 2004-12-25 18:20:43 org.springframework.context.support.
AbstractXmlApplicationContext refreshBeanFactory

[java] 信息: Bean factory for application context
[org.springframework.context.support.ClassPathXmlApplicationContext;hashCode=10851992]: org.springframework.beans.factory.support.
DefaultListableBeanFactory defining beans [connectionFactory,destination,
jmsTemplate, jndiTemplate]; root of BeanFactory hierarchy

[java] 2004-12-25 18:20:43 org.springframework.context.support.
AbstractApplicationContext refresh

[java] 信息: 4 beans defined in application context [org.springframework.
context.support.ClassPathXmlApplicationContext;hashCode=10851992]

[java] 2004-12-25 18:20:43 org.springframework.context.support.
AbstractApplicationContext initMessageSource

[java] 信息: No message source found for context [org.springframework.
context.support.ClassPathXmlApplicationContext;hashCode=10851992]: using

```
empty default
[java] 2004-12-25 18:20:43 org.springframework.context.support.
AbstractApplicationContext initApplicationEventMulticaster
[java] 信息: No ApplicationEventMulticaster found for context
[org.springframework.context.support.ClassPathXmlApplicationContext;hashCode=10851992]: using default
[java] 2004-12-25 18:20:43 org.springframework.context.support.
AbstractApplicationContext refreshListeners
[java] 信息: Refreshing listeners
[java] 2004-12-25 18:20:43 org.springframework.beans.factory.support.
DefaultListableBeanFactory preInstantiateSingletons
[java] 信息: Pre-instantiating singletons in factory
[org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [connectionFactory,destination,jmsTemplate,jndiTemplate];
root of BeanFactory hierarchy]
[java] 2004-12-25 18:20:43 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'connectionFactory'
[java] 2004-12-25 18:20:43 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'jndiTemplate'
[java] 2004-12-25 18:20:43 org.springframework.jndi.JndiLocatorSupport
lookup
[java] 信息: Located object with JNDI name [ConnectionFactory]:
value=[org.jboss.mq.SpyConnectionFactory@13f3045]
[java] 2004-12-25 18:20:43 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'destination'
[java] 2004-12-25 18:20:43 org.springframework.jndi.JndiLocatorSupport
lookup
[java] 信息: Located object with JNDI name [queue/testQueue]:
value=[QUEUE.testQueue]
[java] 2004-12-25 18:20:43 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'jmsTemplate'
[java] 2004-12-25 18:20:44 com.openv.spring.JmsTemplateTest11Receivemain
[java] 信息: Hello World(11),queue/testQueue!
BUILD SUCCESSFUL
Total time: 2 seconds
```

为实现消息的异步接收，开发者需要在应用中实现 `MessageListener` 接口。为此，本书开发了 `JmsTemplateTest11MessageListener` 客户应用。其代码如下。

```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
```

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

import org.springframework.jms.core.JmsTemplate;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageListener;
import javax.jms.QueueSession;
import javax.jms.Session;
import javax.jms.TextMessage;

/**
 * JmsTemplateTest11MessageListener 客户应用
 *
 * @author luoshifei
 */
public class JmsTemplateTest11MessageListener implements MessageListener {
    protected static final Log log = LogFactory
        .getLog(JmsTemplateTest11MessageListener.class);

    private JmsTemplate jmsTemplate;

    private Connection conn;

    /**
     * 消息回调
     */
    public void onMessage(Message message) {
        if (message instanceof TextMessage) {
            TextMessage tm = (TextMessage) message;

            try {
                log.info(tm.getText());
            } catch (JMSException e) {
                log.error("JMSException", e);
            }
        }
    }

    public static void main(String[] args) throws JMSException {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
```

```
        "appcontext11ml.xml");
    JmsTemplateTest11MessageListener jtt11ml =
        new JmsTemplateTest11MessageListener();
    jtt11ml.setJmsTemplate((JmsTemplate) ac.getBean("jmsTemplate"));
    jtt11ml.init();

    try {
        Thread.sleep(5000);
    } catch (InterruptedException e) {
        log.error("InterruptedException", e);
    }

    jtt11ml.destroy();
}

/**
 * 启动 JMS 监听器
 *
 * @throws JMSEException
 */
public void init() throws JMSEException {
    ConnectionFactory qcf = this.jmsTemplate.getConnectionFactory();
    Destination destination = this.jmsTemplate.getDefaultDestination();
    conn = qcf.createConnection("guest", "guest");

    Session session = conn.createSession(false,
        QueueSession.AUTO_ACKNOWLEDGE);
    MessageConsumer mc = session.createConsumer(destination);
    mc.setMessageListener(this);
    conn.start();
}

public void destroy() throws JMSEException {
    conn.close();
}

/**
 * @param jmsTemplate
 *      The jmsTemplate to set.
 */
public void setJmsTemplate(JmsTemplate jmsTemplate) {
    this.jmsTemplate = jmsTemplate;
}

}
```

相应的 Spring 配置文件如下 (appcontext11ml.xml)。


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

    <bean id="connectionFactory"
        class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName">
            <value>ConnectionFactory</value>
        </property>
        <property name="jndiTemplate">
            <ref local="jndiTemplate"></ref>
        </property>
    </bean>

    <bean id="destination"
        class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName">
            <value>queue/testQueue</value>
        </property>
        <property name="jndiTemplate">
            <ref local="jndiTemplate"></ref>
        </property>
    </bean>

    <bean id="jmsTemplate"
        class="org.springframework.jms.core.JmsTemplate">
        <property name="connectionFactory">
            <ref local="connectionFactory"/>
        </property>
        <property name="defaultDestination">
            <ref local="destination" />
        </property>
    </bean>

    <bean id="jndiTemplate"
        class="org.springframework.jndi.JndiTemplate">
        <property name="environment">
            <props>
                <prop key="java.naming.factory.initial">
                    org.jnp.interfaces.NamingContextFactory
                </prop>
                <prop key="java.naming.provider.url">
                    jnp://localhost:1099
                </prop>
                <prop key="java.naming.factory.url.pkgs">
```

```
        org.jboss.naming:org.jnp.interfaces
      </prop>
    </props>
  </property>
</bean>
```

```
</beans>
```

运行结果如下（通过运行 Ant build.xml 中的 runI18nlistener 任务）。

```
Buildfile: D:\workspace\example14\build.xml
compile:
runI18nlistener:
  [java] 2004-12-25 19:16:34
org.springframework.beans.factory.xml.XmlBeanDefinitionReader
loadBeanDefinitions
  [java] 信息: Loading XML bean definitions from class path resource
[appcontext11ml.xml]
  [java] 2004-12-25 19:16:35 org.springframework.context.support.
AbstractXmlApplicationContext refreshBeanFactory
  [java] 信息: Bean factory for application context [org.springframework.
context.support.ClassPathXmlApplicationContext;hashCode=10851992]:
org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [connectionFactory,destination,jmsTemplate,jndiTemplate];
root of BeanFactory hierarchy
  [java] 2004-12-25 19:16:35 org.springframework.context.support.
AbstractApplicationContext refresh
  [java] 信息: 4 beans defined in application context [org.springframework.
context.support.ClassPathXmlApplicationContext;hashCode=10851992]
  [java] 2004-12-25 19:16:35 org.springframework.context.support.
AbstractApplicationContext initMessageSource
  [java] 信息: No message source found for context [org.springframework.
context.support.ClassPathXmlApplicationContext;hashCode=10851992]: using
empty default
  [java] 2004-12-25 19:16:35 org.springframework.context.support.
AbstractApplicationContext initApplicationEventMulticaster
  [java] 信息: No ApplicationEventMulticaster found for context
[org.springframework.context.support.ClassPathXmlApplicationContext;hashC
ode=10851992]: using default
  [java] 2004-12-25 19:16:35 org.springframework.context.support.
AbstractApplicationContext refreshListeners
  [java] 信息: Refreshing listeners
  [java] 2004-12-25 19:16:35 org.springframework.beans.factory.support.
DefaultListableBeanFactory preInstantiateSingletons
  [java] 信息: Pre-instantiating singletons in factory
[org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [connectionFactory,destination,jmsTemplate,jndiTemplate];
root of BeanFactory hierarchy]
```

```

[java] 2004-12-25 19:16:35 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'connectionFactory'
[java] 2004-12-25 19:16:35 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'jndiTemplate'
[java] 2004-12-25 19:16:35 org.springframework.jndi.JndiLocatorSupport
lookup
[java] 信息: Located object with JNDI name [ConnectionFactory]:
value=[org.jboss.mq.SpyConnectionFactory@1434234]
[java] 2004-12-25 19:16:35 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'destination'
[java] 2004-12-25 19:16:35 org.springframework.jndi.JndiLocatorSupport
lookup
[java] 信息: Located object with JNDI name [queue/testQueue]:
value=[QUEUE.testQueue]
[java] 2004-12-25 19:16:35 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'jmsTemplate'
[java] 2004-12-25 19:16:35
com.openv.spring.JmsTemplateTest11MessageListener onMessage
[java] 信息: Hello World(11),queue/testQueue!
[java] 2004-12-25 19:16:35
com.openv.spring.JmsTemplateTest11MessageListener onMessage
[java] 信息: Hello World(11),queue/testQueue!
BUILD SUCCESSFUL
Total time: 8 seconds

```

对于 JMS 1.02 接收消息的应用而言,过程差不多,本书不再给出介绍。至于通过消息驱动 Bean 异步接收消息的情形,本书将在第 10 章给出研究实例。

3. 使用消息转换器

JmsTemplate 提供了各种 send 方法,供发送消息使用。其中, JmsTemplate 中的 convertAndSend 和 receiveAndConvert 方法能够借助于 MessageConverter 接口实现将消息做相应的转换。MessageConverter 接口定义了 Java 对象同 JMS 消息之间的简单约定。Spring 框架的 JMS 抽象提供了 MessageConverter 的简单实现,即 SimpleMessageConverter (包括为 JMS 1.0.2 提供的 SimpleMessageConverter102)。借助于 SimpleMessageConverter,开发者能够实现 String 与 TextMessage、byte[] 与 BytesMessage、java.util.Map 与 MapMessage 之间的自动转化。当然,开发者也可以自定义其他的 MessageConverter 接口实现。通过使用消息转换器,使得开发者能够更专注于业务逻辑,而不用关注 JMS 消息的具体内容。

为此,借助于 org.springframework.jms.support.converter.MessageConverter 接口,开发者能够实现自定义 MessageConverter 消息转换器。比如,example14 中实现了 PersonConverter 转换器。其代码如下。

```
package com.openv.spring;
```

```
import javax.jms.JMSEException;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.Session;

import
org.springframework.jms.support.converter.MessageConversionException;
import org.springframework.jms.support.converter.MessageConverter;

/**
 * 自定义 MessageConverter 实现
 *
 * @author luoshifei
 */
public class PersonConverter implements MessageConverter {

    public Message toMessage(Object object, Session session)
        throws JMSEException, MessageConversionException {
        PersonVO personVO = (PersonVO) object;
        MapMessage mm = session.createMapMessage();
        mm.setString("firstname", personVO.getFirstname());
        mm.setString("lastname", personVO.getLastname());
        return mm;
    }

    public Object fromMessage(Message message) throws JMSEException,
        MessageConversionException {
        if (!(message instanceof MapMessage))
            return null;

        MapMessage mm = (MapMessage) message;
        PersonVO personVO = new PersonVO();
        try {
            personVO.setFirstname(mm.getString("firstname"));
            personVO.setLastname(mm.getString("lastname"));
        } catch (JMSEException jmse) {
            throw new MessageConversionException(jmse.getMessage());
        }
        return personVO;
    }
}
```

可以看出，它实现了 PersonVO 到 JMS MapMessage 的自动转换。其中，toMessage() 方法将 PersonVO 转换为 MapMessage；fromMessage() 方法，将 MapMessage 转换为 PersonVO 值对象。为将 PersonConverter 注册到 JmsTemplate 中，开发者首先需要在 Spring 配置文件

中定义 `personConverter` (详情见 `personconvertertest.xml`), 见如下配置代码。

随后, 需要在 `jmsTemplate` 中借助于 `messageConverter` 属性声明对 `personConverter` 的引用。期间, 还定义了 `receiveTimeout` 属性, 即 5 秒后将自动超时。

```
<bean id="personConverter" class="com.openv.spring.PersonConverter"/>

<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory">
    <ref local="connectionFactory"/>
  </property>
  <property name="defaultDestination">
    <ref local="destination" />
  </property>
  <property name="messageConverter">
    <ref local="personConverter"></ref>
  </property>
  <property name="receiveTimeout">
    <value>5000</value>
  </property>
</bean>
```

最后, 开发客户应用, 见 `PersonConverterTest.java`。它在发送消息后, 同时又接收了该消息。因此, 开发者能够对 `PersonConverter` 提供的 `toMessage` 和 `fromMessage` 有更全面的认识。

```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jms.core.JmsTemplate;

/**
 * PersonConverterTest 客户应用
 *
 * @author luoshifei
 */
public class PersonConverterTest {
  protected static final Log log = LogFactory
    .getLog(PersonConverterTest.class);

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

    JmsTemplate jt = (JmsTemplate) ac.getBean("jmsTemplate");
```

```
PersonVO personSend = new PersonVO();
personSend.setFirstname("Shifei");
personSend.setLastname("Luo");

//转换并发送消息
jt.convertAndSend(personSend);
log.info("成功发送消息!");

//接收并转换消息
PersonVO personReceive = (PersonVO) jt.receiveAndConvert();
log.info(personReceive.getFirstname() + ", " +
personReceive.getLastname());
log.info("成功接收消息!");
}
}
```

通过运行 Ant build.xml 中的 personconverter 任务，能够执行它。

```
Buildfile: D:\workspace\example14\build.xml
compile:
personconverter:
[java] 2005-2-11 13:20:01 org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
[java] 信息: Loading XML bean definitions from class path resource
[personconvertertest.xml]
[java] 2005-2-11 13:20:02 org.springframework.context.support.
AbstractXmlApplicationContext refreshBeanFactory
[java] 信息: Bean factory for application context [org.springframework.
context.support.ClassPathXmlApplicationContext;hashCode=3043939]:
org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [connectionFactory,destination,personConverter,jmsTemplate,
jndiTemplate]; root of BeanFactory hierarchy
[java] 2005-2-11 13:20:02 org.springframework.context.support.
AbstractApplicationContext refresh
[java] 信息: 5 beans defined in application context [org.springframework.
context.support.ClassPathXmlApplicationContext;hashCode=3043939]
[java] 2005-2-11 13:20:02 org.springframework.context.support.
AbstractApplicationContext initMessageSource
[java] 信息: No message source found for context [org.springframework.
context.support.ClassPathXmlApplicationContext;hashCode=3043939]: using
empty default
[java] 2005-2-11 13:20:02 org.springframework.context.support.
AbstractApplicationContext initApplicationEventMulticaster
[java] 信息: No ApplicationEventMulticaster found for context
[org.springframework.context.support.ClassPathXmlApplicationContext;hashC
ode=3043939]: using default
[java] 2005-2-11 13:20:02 org.springframework.context.support.
```

```

AbstractApplicationContext refreshListeners
    [java] 信息: Refreshing listeners
    [java] 2005-2-11 13:20:02 org.springframework.beans.factory.support.
DefaultListableBeanFactory preInstantiateSingletons
    [java] 信息: Pre-instantiating singletons in factory [org.springframework.
beans.factory.support.DefaultListableBeanFactory defining beans
[connectionFactory,destination,personConverter,jmsTemplate,jndiTemplate];
root of BeanFactory hierarchy]
    [java] 2005-2-11 13:20:02 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'connectionFactory'
    [java] 2005-2-11 13:20:02 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'jndiTemplate'
    [java] 2005-2-11 13:20:03 org.springframework.jndi.JndiLocatorSupport
lookup
    [java] 信息: Located object with JNDI name [ConnectionFactory]:
value=[org.jboss.mq.SpyConnectionFactory@af8358]
    [java] 2005-2-11 13:20:03 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'destination'
    [java] 2005-2-11 13:20:03 org.springframework.jndi.JndiLocatorSupport
lookup
    [java] 信息: Located object with JNDI name [queue/testQueue]:
value=[QUEUE.testQueue]
    [java] 2005-2-11 13:20:03 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'personConverter'
    [java] 2005-2-11 13:20:03 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'jmsTemplate'
    [java] 2005-2-11 13:20:03 com.openv.spring.PersonConverterTest main
    [java] 信息: 成功发送消息!
    [java] 2005-2-11 13:20:03 com.openv.spring.PersonConverterTest main
    [java] 信息: Shifei, Luo
    [java] 2005-2-11 13:20:03 com.openv.spring.PersonConverterTest main
    [java] 信息: 成功接收消息!
BUILD SUCCESSFUL
Total time: 3 seconds

```

默认时, 如果不设置 `JmsTemplate` 的 `messageConverter` 属性, Spring 会直接使用 JMS 抽象服务提供的 `org.springframework.jms.converter.SimpleMessageConverter`。然而, 为了修改消息本身的内容, 直接借助于 `MessageConverter` 肯定实现不了。此时, 需要借助于 Spring 提供的 `MessagePostProcessor` 接口。比如, `JmsTemplateTest11Converter.java` 使用了 `MessagePostProcessor` 接口, 其具体代码如下。

```
package com.openv.spring;
```

```
import javax.jms.JMSEException;
import javax.jms.Message;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessagePostProcessor;

/**
 * JmsTemplateTest11Converter 客户应用
 *
 * @author luoshifei
 */
public class JmsTemplateTest11Converter {
    protected static final Log log = LogFactory
        .getLog(JmsTemplateTest11Converter.class);

    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "appcontext11.xml");

        JmsTemplate jt = (JmsTemplate) ac.getBean("jmsTemplate");
        String str = "Hello World";
        jt.convertAndSend(jt.getDefaultDestination(), str,
            new MessagePostProcessor() {
                public Message postProcessMessage(Message message)
                    throws JMSEException {
                    message.setIntProperty("helloworldid", 12345678);
                    message.setJMSCorrelationID("1234-5678");
                    message.setJMSExpiration(40000000);
                    return message;
                }
            });
    }
}
```

期间，针对客户需求，开发者能够修改消息本身的属性，比如设置新的属性、消息有效期限等。其执行结果如下（Ant build.xml 中的 run11converter 任务）。

```
Buildfile: D:\workspace\example14\build.xml
compile:
run11converter:
    [java] 2004-12-25 18:27:18
org.springframework.beans.factory.xml.XmlBeanDefinitionReader
```



```
loadBeanDefinitions
    [java] 信息: Loading XML bean definitions from class path resource
    [appcontext11.xml]
    [java] 2004-12-25 18:27:18 org.springframework.context.support.
    AbstractXmlApplicationContext refreshBeanFactory
    [java] 信息: Bean factory for application context [org.springframework.
    context.support.ClassPathXmlApplicationContext;hashCode=22540508]:
    org.springframework.beans.factory.support.DefaultListableBeanFactory
    defining beans [connectionFactory,destination,jmsTemplate,jndiTemplate];
    root of BeanFactory hierarchy
    [java] 2004-12-25 18:27:18 org.springframework.context.support.
    AbstractApplicationContext refresh
    [java] 信息: 4 beans defined in application context [org.springframework.
    context.support.ClassPathXmlApplicationContext;hashCode=22540508]
    [java] 2004-12-25 18:27:18 org.springframework.context.support.
    AbstractApplicationContext initMessageSource
    [java] 信息: No message source found for context [org.springframework.
    context.support.ClassPathXmlApplicationContext;hashCode=22540508]: using
    empty default
    [java] 2004-12-25 18:27:18 org.springframework.context.support.
    AbstractApplicationContext initApplicationEventMulticaster
    [java] 信息: No ApplicationEventMulticaster found for context
    [org.springframework.context.support.ClassPathXmlApplicationContext;hashC
    ode=22540508]: using default
    [java] 2004-12-25 18:27:18 org.springframework.context.support.
    AbstractApplicationContext refreshListeners
    [java] 信息: Refreshing listeners
    [java] 2004-12-25 18:27:18 org.springframework.beans.factory.support.
    DefaultListableBeanFactory preInstantiateSingletons
    [java] 信息: Pre-instantiating singletons in factory [org.springframework.
    beans.factory.support.DefaultListableBeanFactory defining beans
    [connectionFactory,destination,jmsTemplate,jndiTemplate]; root of
    BeanFactory hierarchy]
    [java] 2004-12-25 18:27:18 org.springframework.beans.factory.support.
    AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'connectionFactory'
    [java] 2004-12-25 18:27:18 org.springframework.beans.factory.support.
    AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'jndiTemplate'
    [java] 2004-12-25 18:27:18 org.springframework.jndi.JndiLocatorSupport
    lookup
    [java] 信息: Located object with JNDI name [ConnectionFactory]:
    value=[org.jboss.mq.SpyConnectionFactory@17a29a1]
    [java] 2004-12-25 18:27:18 org.springframework.beans.factory.support.
    AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'destination'
```

```
[java] 2004-12-25 18:27:18 org.springframework.jndi.JndiLocatorSupport
lookup
[java] 信息: Located object with JNDI name [queue/testQueue]:
value=[QUEUE.testQueue]
[java] 2004-12-25 18:27:18 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'jmsTemplate'
[java] 2004-12-25 18:27:18 com.openv.spring.JmsTemplateTest11Converter
main
[java] 信息: 成功发送消息!
BUILD SUCCESSFUL
Total time: 2 seconds
```

通过 JBoss 提供的 jmx-console, 开发者能够获得其具体内容, 如图 8-6 所示。这对于一些应用场合很有帮助, 比如在一些工作流服务器 (Workflow) 中利用这种信息标识判断客户是否处理过工作列表 (消息)。

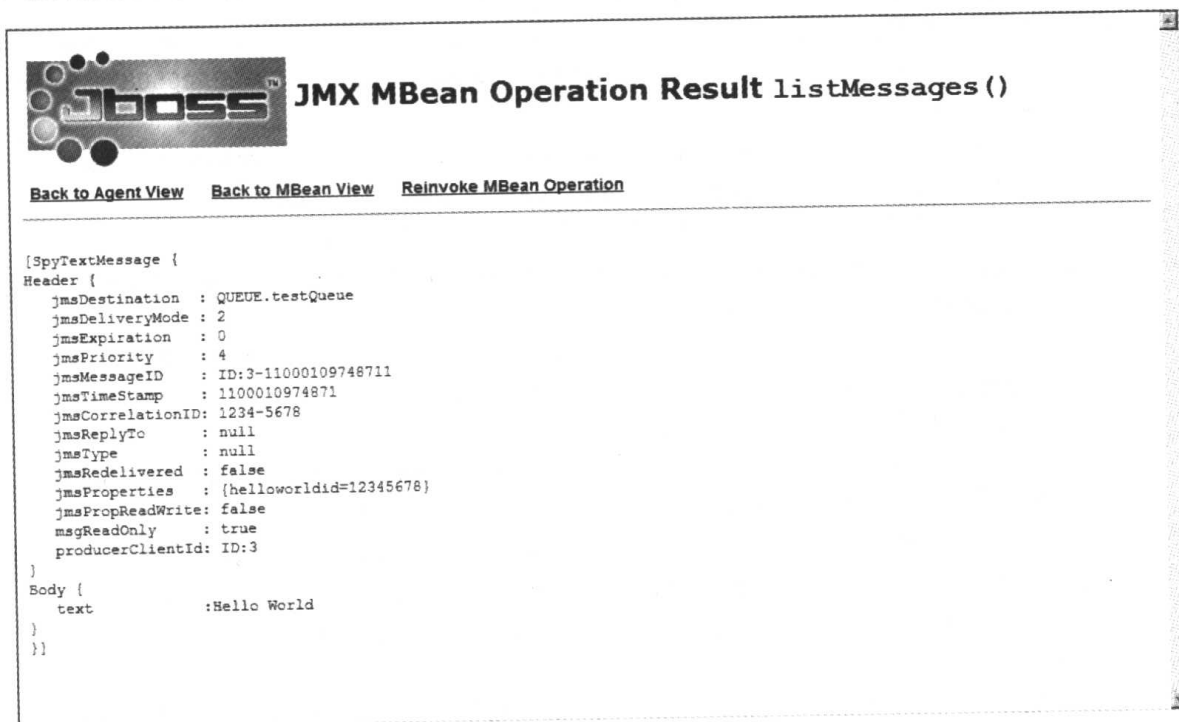


图 8-6 使用消息转换器后的消息内容

8.2.2 事务管理

Spring 提供了 JmsTransactionManager, 用于管理单个 JMS ConnectionFactory 的事务。借助于 JtaTransactionManager 和具有 XA 能力的 JMS ConnectionFactory 能够管理全局事务。其实, 无论是 JMS, 还是 JDBC, 它们的事务使用步骤都差不多。至于 JMS 事务管理的更深入介绍, 请参考《Java Transaction Processsing Design and Implementation》一书, 由 Mark Little 等人写作而成, PTR 出版社于 2004 年出版发行。这本书对于学习 Java 中的事务处理很适合。

8.3 小 结

本章内容深入地对 JMS、Spring 提供的 JMS 抽象作了介绍。随后，给出了具体研究实例，其中重点给出了 Queue 类型的目的地。由于 Topic 类型的目的地类似，因此开发者可以依次类推。其中，消息驱动 Bean 内容将在第 10 章研究。另外，Spring JMS 服务抽象还提供了消息转换器，使得发送和接收消息的应用代码更为简洁。

第 9 章 邮件服务——JavaMail

邮件服务，在企业应用中很常见。

借助于 Spring 提供的 JavaMail 抽象，开发者能够很容易同 JavaMail 进行交互，而不用理会太多的 JavaMail 细节。而且，这种抽象同 Spring 提供的其他 J2EE 组件类似，这对于降低 Spring 的学习曲线很有帮助。

9.1 背 景

在 Java/J2EE 平台中，JavaMail 提供了平台独立的、协议独立的框架，供构建邮件和消息应用使用。它作为 Java 2 标准版的可选包存在，是 J2EE 平台的重要组成部分。

为开发 Spring JavaMail 使能应用，借助于 `org.springframework.mail.MailSender` 接口及实现能够满足开发者要求。

9.2 Spring 对 JavaMail 提供的支持

目前，Spring 框架提供的 JavaMail 抽象主要由如下几个包构成。

- `org.springframework.mail`: Spring 框架提供的 JavaMail 抽象的主要内容都在这里。其中，包含了若干异常处理类，这同 Spring DAO 提供的异常处理类似，使得开发者有权选择是否需要捕捉未受查运行时异常。比如，`MailException`、`MailSendException`、`MailParseException` 等常见的未受查运行期异常。
- `org.springframework.mail.cos`: 基于 Jason Hunter 提供的 COS¹而开发的 `MailSender` 接口实现。这是一个基于 SMTP 协议的邮件发送实现。
- `org.springframework.mail.javamail`: 基于 JavaMail API 的 `MailSender` 实现。其中，还包括了 `MimeMessageHelper`、`MimeMessagePreparator` 等类。

其中，`org.springframework.mail` 包提供的异常处理类见图 9-1 所示。

因此，Spring 框架的 JavaMail 抽象使用上述异常类，使得开发者在使用 JavaMail 抽象时能够很灵活地处理。

发送邮件是邮件服务的主要功能，因此 Spring 提供的 JavaMail 抽象包括了 `MailSender` 接口。其中，Spring 框架包括了 `MailSender` 接口的两种实现。其一，基于 Jason Hunter 开发的 `com.oreilly.servlet.MimeMessage` 类，即 `org.springframework.mail.cos.CosMailSenderImpl` 类。其二，基于 JavaMail 类，即 `org.springframework.mail.javamail.JavaMailSenderImpl` 类。

¹ 由 O'Reilly 于 1998 年出版的《Java Servlet Programming》一书。开发者通过 <http://servlets.com/cos> 能够找到 COS。中国电力出版社已经引进并出版了该书的中文版。

尽管 `CosMailSenderImpl` 的功能没有 `JavaMailSenderImpl` 丰富，但是能够满足一般的应用需求。

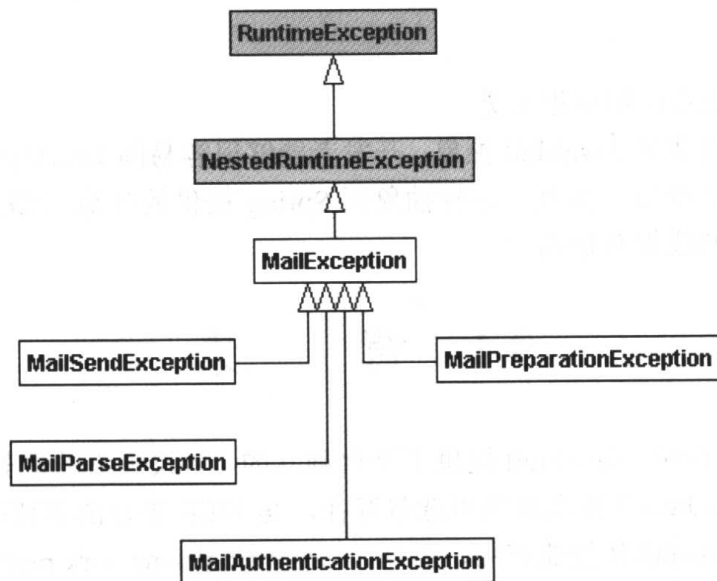


图 9-1 `org.springframework.mail` 包提供的异常类

9.2.1 使用 `CosMailSenderImpl`

针对 Jason Hunter 开发的 `MailMessage` 类（独立于 `JavaMail`），Spring 框架提供了相应的 `MailSender` 实现，即 `CosMailSenderImpl`。开发者能够借助于 `CosMailSenderImpl` 和 `SimpleMailMessage` 发送简单邮件。

比如，在 `example15` 中提供的示例代码如下（需要特定的邮件服务器支持）。其中，将构建的 `SimpleMailMessage` 消息作为 `MailSender` 实例的参数，从而完成邮件的发送。

```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

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

import org.springframework.mail.MailException;
import org.springframework.mail.MailSender;
import org.springframework.mail.SimpleMailMessage;

/**
 * CosMailSenderImplTest 客户应用
 *
 * @author luoshifei
 */
```

```

*/
public class CosMailSenderImplTest {
    protected static final Log log =
LogFactory.getLog(CosMailSenderImplTest.class);

    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "appcontextcos.xml");
        MailSender ms = (MailSender) ac.getBean("cosMailSender");

        SimpleMailMessage msg = (SimpleMailMessage) ac.getBean(
            "simpleMailMessage");

        try {
            ms.send(msg);
        } catch (MailException ex) {
            log.error("邮件发送异常", ex);
        }
    }
}

```

注意，开发者也可以不处理 `MailException` 异常。当然，如果需要更健壮的邮件服务支持，则需要使用 `JavaMail` 实现。相应的 `Spring` 配置文件，即 `appcontextcos.xml` 示例如下。

```

<?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="cosMailSender"
        class="org.springframework.mail.cos.CosMailSenderImpl">
        <property name="host">
            <value>smtp.openv.com</value>
        </property>
    </bean>

    <bean id="simpleMailMessage"
        class="org.springframework.mail.SimpleMailMessage">
        <property name="from">
            <value>masterspring@openv.com</value>
        </property>
        <property name="subject">
            <value>Hello World</value>
        </property>
        <property name="text">
            <value>HaHa</value>
        </property>
    </bean>

```

```
</beans>
```

9.2.2 使用 JavaMailSenderImpl

为使用内容更丰富的 MIME 消息，开发者可以使用 `JavaMailSenderImpl` 实现。类似于 JMS，发送 JavaMail MIME 消息可以借助于 `MimeMessagePreparator` 回调接口，即开发者只需要实现其给出的 `prepare` 回调方法。其中，example15 给出的示例代码如下（需要特定的邮件服务器支持）。

```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

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

import org.springframework.mail.MailException;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessagePreparator;

import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

/**
 * JavaMailSenderImplTest 客户应用
 *
 * @author luoshifei
 */
public class JavaMailSenderImplTest {
    protected static final Log log =
        LogFactory.getLog(JavaMailSenderImplTest.class);

    public static void main(String[] args) {
        ApplicationContext ac = new ClassPathXmlApplicationContext(
            "appcontextjavamail.xml");
        JavaMailSender ms = (JavaMailSender) ac.getBean("javaMailSender");
        MimeMessagePreparator mmp = new MimeMessagePreparator() {
            public void prepare(MimeMessage mimeMessage)
                throws MessagingException {
                mimeMessage.setRecipient(Message.RecipientType.TO,
```



```

        new InternetAddress("j2eebeans@yahoo.com.cn"));
        mimeMessage.setFrom(new InternetAddress(
            "masterspring@openv.com"));
        mimeMessage.setText("你好, 我的朋友!");
    }
};

try {
    ms.send(mmp);
} catch (MailException ex) {
    log.error("邮件发送异常", ex);
}
}
}

```

相应的 Spring 配置文件, 即 `appcontextjavamail.xml` 示例如下。

```

<?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="javaMailSender"
        class="org.springframework.mail.javamail.JavaMailSenderImpl">
        <property name="host">
            <value>smtp.openv.com</value>
        </property>
        <property name="username">
            <value>username</value>
        </property>
        <property name="password">
            <value>password</value>
        </property>
    </bean>

</beans>

```

开发者还可以使用用于处理 MIME 消息的辅助类, 即 `MimeMessageHelper`。利用辅助类, 开发者还可以发送附件和内嵌资源。对 JavaMail 本身的论述不是本书的重点内容, 因为 Spring 对 JavaMail 所作的抽象才是本章的重点。

当然, 开发者也可以通过 JBoss 提供的 JavaMail 服务² (通过 JNDI) 获得类似的功能。比如, 当应用运行在 JBoss 应用服务器中时, 开发者可以配置类似如下的 JavaMail 服务片段, 以获得 JavaMail Session。最终, 便能够实现对 JavaMail 的使用。

```

<bean id="mailSession"
    class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName">

```

² http://www.huihoo.com/jboss/online_manual/3.0/ch13s98.html

```
        <value> java:/Mail</value>
    </property>
</bean>
<bean id="mailSender"
    class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <property name="session">
        <ref bean="mailSession"/>
    </property>
</bean>
```

9.3 小 结

本章简要地对 Spring 框架提供的 JavaMail 抽象进行了介绍。

第 10 章 企业 Bean 服务——EJB

企业 Bean (Enterprise JavaBean, EJB), 在 J2EE 平台中占有重要的位置。EJB 是服务器端的组件架构, 它能够简化构建企业级、分布式组件应用, 并且使用 Java 开发。在 J2EE 1.4 中加强了 EJB 组件模型, 尤其是 Web 服务的支持 (在第 13 章有 Web 服务相关的研究) 和 JCA 连接器的支持能力。比如, 消息驱动 Bean 支持 JMS 以外的消息类型, 而这些消息类型可以通过 JCA 连接器进行支持 (比如, 处理邮件消息类型)。再比如, 通过将无状态会话 Bean 导出为 Web 服务的能力。

一般而言, EJB 也属于远程服务的范畴, 但本书将它作为单独一章进行研究。如果开发者对 Spring 提供的其他远程服务类型感兴趣, 可以直接阅读第 13 章内容。

Spring 框架为 EJB 提供了开发和访问支持。

10.1 背 景

借助于 Spring IoC 和 Spring AOP, 能够将企业服务, 比如事务管理、JNDI 注入功能, 应用于 POJO 对象。这同 EJB 组件使用 J2EE 平台服务类似。因此, Spring 作为轻量级容器, 在某种程度上能够替代 EJB 组件。当然, EJB 作为 J2EE 平台中的重要内容, 其使用面也是很广泛。考虑到现有的 EJB 规范并没有为开发者提供辅助开发 EJB 组件服务, 比如未提供抽象类, 以封装 EJB 中的回调接口。因此, 处于制定中的 EJB 3.0 却考虑到了这种需求, 从而降低开发者的工作强度。

因此, 为使得开发和实现 EJB 组件, Spring 提供了 EJB 抽象。这种抽象使得开发者能够灵活地、透明地在运行时替换服务实现, 比如可以在本地 EJB、远程 EJB、POJO 之间互换, 而且开发者不用修改客户代码。

在现有的 EJB 组件类型中, 无状态会话 Bean 和消息驱动 Bean 的使用尤为广泛。Spring 对无状态会话 Bean 的支持尤为优秀, 这将是本章的重点讨论对象。

10.2 Spring 对 EJB 提供的支持

目前, Spring 框架为 EJB 服务而提供的包主要有:

- `org.springframework.ejb.access`: 简化对 EJB 的访问。其中, 引入了许多拦截器, 以实现对 EJB 的访问。
- `org.springframework.ejb.support`: 简化对 EJB 的开发。其中, 实现了 EJB 组件对 Spring BeanFactory 的访问, 从而能够在 EJB 组件中访问到 POJO 服务。因此, 开发者能够很容易地借助于 EJB 实现事务管理、线程管理、远程服务; 借助于 POJO

实现业务逻辑。在某种程度上，这是借助于 EJB 声明式企业服务替换 Spring AOP。

为辅助开发者实现 EJB 组件的开发，Spring 框架提供了 `org.springframework.ejb.support` 包。可以看出，借助于 Spring EJB 抽象，能够将业务逻辑通过 POJO 实现、将声明式企业服务通过 EJB 实现。图 10-1 给出了 `support` 包中的抽象类。

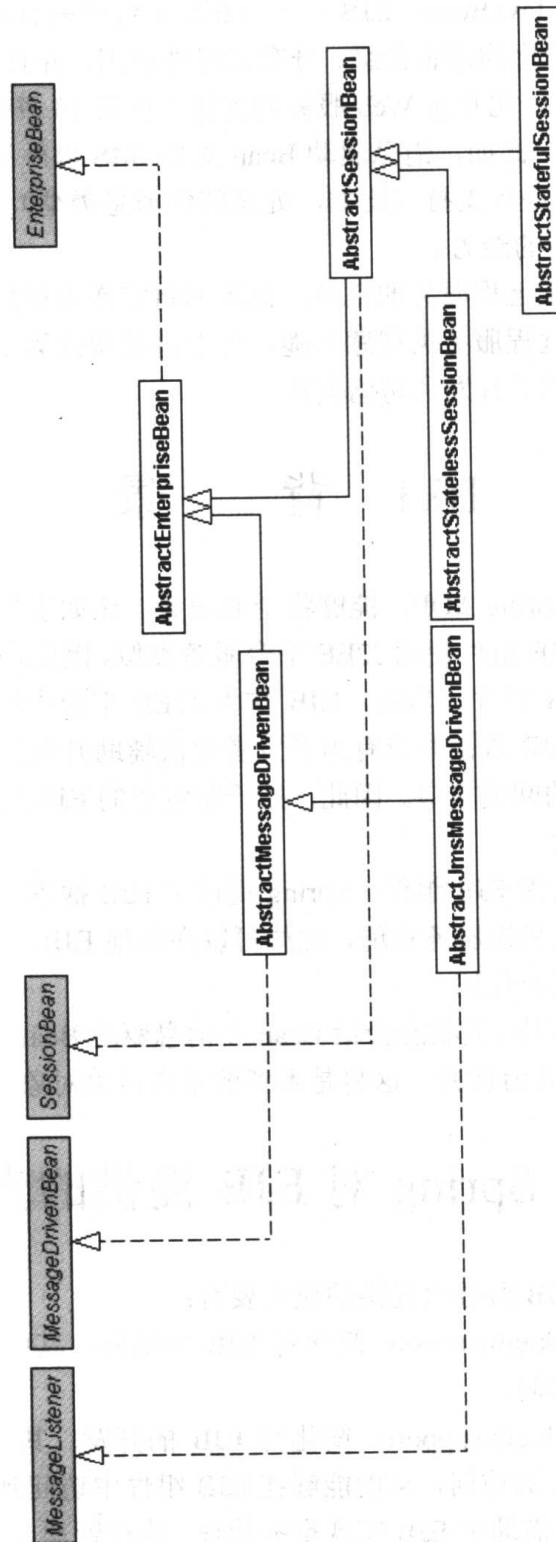


图 10-1 `support` 包中提供的抽象类

开发者只需要实现 `AbstractStatelessSessionBean`，或者 `AbstractStatefulSessionBean`，或者 `AbstractJmsMessageDrivenBean` (`AbstractMessageDrivenBean`)，便能够实现无状态会话 Bean、有状态会话 Bean 和消息驱动 Bean。这些抽象类的具体含义如下。

- `AbstractStatelessSessionBean`: 用于开发无状态会话 Bean。
- `AbstractStatefulSessionBean`: 用于开发有状态会话 Bean。
- `AbstractJmsMessageDrivenBean`: 用于开发消息驱动 Bean，仅供接收 JMS 类型的消息。
- `AbstractMessageDrivenBean`: 用于开发消息驱动 Bean，供接收其他类型（除 JMS 之外）的消息。这主要是考虑到 EJB 2.1 对 EJB 开发模型的加强而在 Spring 提供的 EJB 抽象服务中所引入的抽象类。

其中，这些抽象类主要在两方面对 EJB 组件开发过程进行了简化。其一，实现了 EJB 生命周期方法的默认空实现。EJB 规范要求各个 EJB 组件类型必须实现相应的生命周期方法，而这些方法在实际 EJB 应用中往往没有方法体，因此 Spring 提供的这些抽象类实现了空的生命周期方法。其二，EJB 组件能够访问到 Spring 提供的 `BeanFactory` IoC 容器。因此，业务逻辑可以通过 `JavaBean` 实现，而 EJB 容器提供的企业级服务由 EJB 组件使用。这使得 `JavaBean` 能够直接使用到企业级服务，应用更加健壮、稳定。

一般情况下，由于实体 Bean 运行的效率低，开发者很少使用它。

为实现对 EJB 的访问，Spring 框架提供的 `access` 包能够满足这种需求。图 10-2 给出了其中的 `FactoryBean` 实现。因此，开发者只需要在 Spring 配置文件中使用到如下两个 `FactoryBean` 即可。

- `LocalStatelessSessionProxyFactoryBean`
- `SimpleRemoteStatelessSessionProxyFactoryBean`

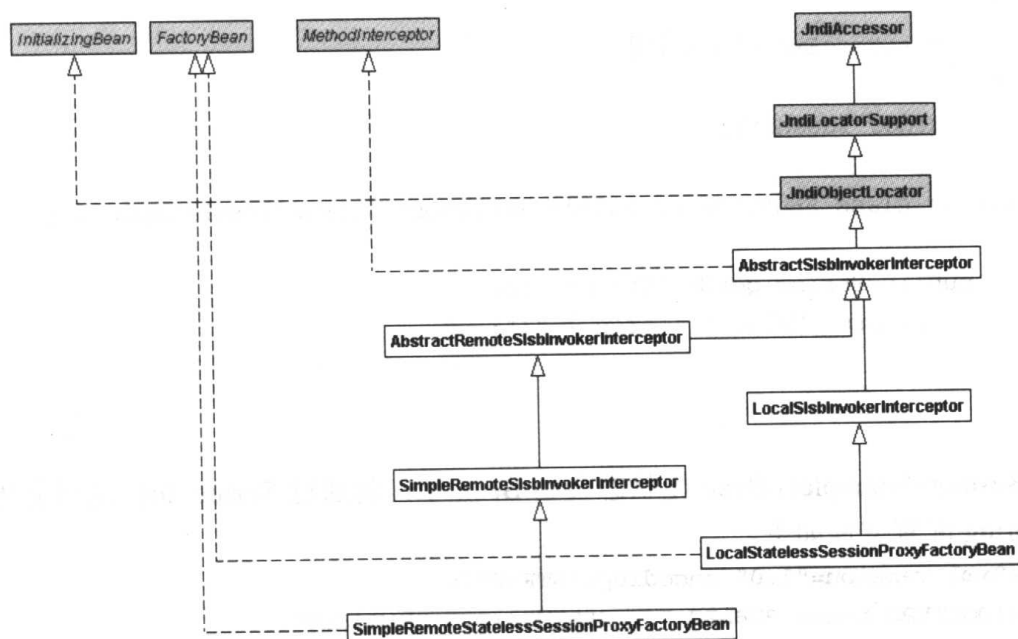


图 10-2 `access` 包中提供的 `FactoryBean` 实现

由于消息驱动 Bean 只是消息的消费者，因此它不存在客户。

10.2.1 开发 EJB

本节内容将重点研究基于 `AbstractStatelessSessionBean` 实现的会话 Bean 和基于 `AbstractJmsMessageDrivenBean` 实现的消息驱动 Bean。至于其他类型的 EJB 实现相类似，开发者可以依此类推。

1. AbstractStatelessSessionBean

其中，本章在 `example16` 实现了 `SBusinessExample16Bean` 无状态会话 Bean。接下来，看看如何实现它，并对它进行单元测试。

首先，实现 `ISBusinessExample16` 业务接口。注意，它并不需要在方法签名中抛出 `RemoteException`，其实现了 `getStr()` 方法，随后的业务接口实现类中将会实现 `getStr()` 方法。

```
package com.openv.spring;

/**
 * ISBusinessExample16 接口
 *
 * @author luoshifei
 */
public interface ISBusinessExample16 {
    public String getStr(String args);
}
```

其中，定义了 `getStr` 业务方法。相应的实现如下。

```
package com.openv.spring;

/**
 * ISBusinessExample16 实现
 *
 * @author luoshifei
 */
public class SBusinessExample16 implements ISBusinessExample16 {

    public String getStr(String args) {
        return "Hello," + args + "!";
    }

}
```

在 `SBusinessExample16Bean` 无状态会话 Bean 中，将通过 Spring IoC 调用上述实现。相应的 Spring 配置文件如下。

```
<?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="sbe16"
      class="com.openv.spring.SBusinessExample16"/>
```

```
</beans>
```

SBusinessExample16Bean 无状态会话 Bean 的 Home 接口如下。

```
package com.openv.spring;

import javax.ejb.EJBHome;

/**
 * SBusinessExample16Bean Home
 *
 * @author luoshifei
 */
public interface SBusinessExample16Home extends EJBHome {

    public SBusinessExample16Remote create() throws
        javax.ejb.CreateException,
        java.rmi.RemoteException;

}
```

Remote 接口如下。

```
package com.openv.spring;

import javax.ejb.EJBObject;

/**
 * SBusinessExample16Bean Remote
 *
 * @author luoshifei
 */
public interface SBusinessExample16Remote extends EJBObject {

    public String getStr(String args) throws java.rmi.RemoteException;

}
```

相应的 **Bean** 类如下。

```
package com.openv.spring;

import javax.ejb.CreateException;

import org.springframework.ejb.support.AbstractStatelessSessionBean;

/**
 * SBusinessExample16Bean 无状态会话 Bean 的 Bean 类
 *
```

```
* @author luoshifei
*/
public class SBusinessExample16Bean extends AbstractStatelessSessionBean
    implements ISBusinessExample16 {
    private ISBusinessExample16 sbe;

    protected void onEjbCreate() throws CreateException {
        sbe = (ISBusinessExample16) getBeanFactory().getBean("sbe16");
    }

    public String getStr(String args) {
        return sbe.getStr(args);
    }
}
```

开发者是否注意到上述粗体内容。在 EJB 中，并没有显式地初始化 BeanFactory 实例，因为 Spring 提供的 EJB 抽象帮开发者完成了上述工作。因此，开发者只需要在 Bean 类中调用 `getBeanFactory()`，从而加载 Spring IoC 容器。当然，开发者也可以修改 Spring EJB 抽象加载 BeanFactory 实例的默认行为¹。

然后，开发者需要提供 `ejb-jar.xml` 文件。

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/j2ee" version="2.1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd">
    <display-name>SBusinessExample16Bean</display-name>

    <enterprise-beans>
        <session>
            <ejb-name>SBusinessExample16Bean</ejb-name>
            <home>com.openv.spring.SBusinessExample16Home</home>
            <remote>com.openv.spring.SBusinessExample16Remote</remote>
            <ejb-class>com.openv.spring.SBusinessExample16Bean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
            <env-entry>
                <env-entry-name>ejb/BeanFactoryPath</env-entry-name>
                <env-entry-type>java.lang.String</env-entry-type>
                <env-entry-value>appcontext.xml</env-entry-value>
            </env-entry>
        </session>
    </enterprise-beans>
```

¹ <http://www.springframework.org/docs/reference/ejb.html>


```
</ejb-jar>
```

其中的粗体部分，请开发者注意。这是配置 Spring BeanFactory 的地方。默认时，Spring EJB 抽象是基于 JNDI 环境变量获得 Spring 配置文件的，比如 appcontext.xml。

另外，jboss.xml 文件如下。

```
<!DOCTYPE jboss PUBLIC
    "-//JBoss//DTD JBOSS 4.0//EN"
    "http://www.jboss.org/j2ee/dtd/jboss_4_0.dtd">
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>SBusinessExample16Bean</ejb-name>
      <jndi-name>SBusinessExample16Bean</jndi-name>
    </session>
  </enterprise-beans>
</jboss>
```

开发者需要编写 Ant build.xml，以打包 EJB jar 文件。开发者可以运行其中的 sbel6jar 任务，或者 deploy 任务。

```
<?xml version="1.0"?>
<project name="example16" default="sbel6jar" basedir=".">
  <property name="sbel6.file" value="sbel6.jar"></property>
  <property name="jboss.dir" value="D:/jboss-4.0.0"/>
  <property name="springjar.dir"
    value="D:/workspace/example16lib/spring-lib"/>

  <target name="sbel6jar">
    <description>Generate EJB jar file.</description>
    <jar jarfile="${sbel6.file}">
      <fileset dir="./classes">
        <include name="**"/>
      </fileset>
      <fileset dir="${springjar.dir}">
        <include name="spring.jar"/>
      </fileset>
    </jar>
  </target>

  <target name="deploy" depends="sbel6jar">
    <description>Deploy The Generated EJB JAR File
      To JBoss Application Server.</description>
    <copy file="${sbel6.file}"
      todir="${jboss.dir}/server/default/deploy"/>
  </target>

</project>
```

然后，请开发者将 EJB jar 文件，即 sbel6.jar 拷贝到 JBoss 应用服务器的 deploy 目录

中（或者直接运行 Ant deploy 任务），比如 D:\jboss-4.0.0\server\default\deploy。在 JBoss 控制台，将会出现如下类似信息。

```
11:04:08,790 INFO [EjbModule] Deploying SBusinessExample16Bean
11:04:09,321 INFO [EJBDeployer] Deployed: file:/D:/jboss-4.0.0/server/
default/deploy/sbe16.jar
```

这表明，sbe16.jar 部署成功。然后，基于 JBoss IDE 1.4 提供的 EJB Test Client 向导创建 JUnit 单元测试。具体代码如下。

```
package com.openv.spring;

import junit.framework.TestCase;

import java.util.Hashtable;
import javax.rmi.PortableRemoteObject;
import javax.naming.Context;
import javax.naming.InitialContext;

/**
 * EJB Test Client
 *
 * @author luoshifei
 */
public class SBusinessExample16BeanTest extends TestCase {

    /** Home interface */
    protected com.openv.spring.SBusinessExample16Home home;

    /**
     * Get the initial naming context
     */
    protected Context getInitialContext() throws Exception {
        Hashtable props = new Hashtable();
        props.put(Context.INITIAL_CONTEXT_FACTORY,
            "org.jnp.interfaces.NamingContextFactory");
        props.put(Context.URL_PKG_PREFIXES,
            "org.jboss.naming:org.jnp.interfaces");
        props.put(Context.PROVIDER_URL, "jnp://localhost:1099");
        Context ctx = new InitialContext(props);
        return ctx;
    }

    /**
     * Get the home interface
     */
    protected com.openv.spring.SBusinessExample16Home getHome()
        throws Exception {
```

```

Context ctx = this.getInitialContext();
Object o = ctx.lookup("SBusinessExample16Bean");
com.openv.spring.SBusinessExample16Home intf =
    (com.openv.spring.SBusinessExample16Home) PortableRemoteObject
        .narrow(o, com.openv.spring.SBusinessExample16Home.class);
return intf;
}

/**
 * Set up the test case
 */
protected void setUp() throws Exception {
    this.home = this.getHome();
}

/**
 * Test for com.openv.spring.SBusinessExample16Remote.getStr(String args)
 */
public void testGetStr() throws Exception {
    com.openv.spring.SBusinessExample16Remote instance;
    String result;

    // Parameters
    String param0 = "luoshifei";

    // Instance creation
    instance = this.home.create();

    // Method call
    result = instance.getStr(param0);

    System.out.println(result);

    // Various assertions
    // assertNotNull(result);
}
}

```

运行 `SBusinessExample16BeanTest` 后，在运行 JBoss 的控制台将会输出如下信息。

```

11:06:31,646 INFO [ContextJndiBeanFactoryLocator] BeanFactoryPath from JNDI
is [appcontext.xml]
11:06:31,676 INFO [XmlBeanDefinitionReader] Loading XML bean definitions from
class path resource [appcontext.xml]
11:06:31,706 INFO [ClassPathXmlApplicationContext] Bean factory for
application context
[org.springframework.context.support.ClassPathXmlApplicationContext;hashC
ode=10802604]:

```

```

org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [sbel6]; root of BeanFactory hierarchy
11:06:31,706 INFO [ClassPathXmlApplicationContext] 1 beans defined in
application context
[org.springframework.context.support.ClassPathXmlApplicationContext;hashC
ode=10802604]
11:06:31,706 INFO [ClassPathXmlApplicationContext] No message source found
for context
[org.springframework.context.support.ClassPathXmlApplicationContext;hashC
ode=10802604]: using empty default
11:06:31,706 INFO [ClassPathXmlApplicationContext] No
ApplicationEventMulticaster found for context
[org.springframework.context.support.ClassPathXmlApplicationContext;hashC
ode=10802604]: using default
11:06:31,706 INFO [ClassPathXmlApplicationContext] Refreshing listeners
11:06:31,706 INFO [DefaultListableBeanFactory] Pre-instantiating singletons
in factory
[org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [sbel6]; root of BeanFactory hierarchy]
11:06:31,706 INFO [DefaultListableBeanFactory] Creating shared instance of
singleton bean 'sbel6'

```

很容易看到, Spring EJB 抽象自动加载了 BeanFactory 实例。与此同时, 在 Eclipse 中能够有如下输出。

```
Hello, luoshifei!
```

当然, 对于一些消耗资源较大的 BeanFactory, 比如包含大量 Bean 定义、与 RDBMS 进行连接等, 开发者在处理这种场景需要谨慎。因此, 每次初始化 EJB 实例时, 会重新加载新的 BeanFactory 实例。显然, 这不利于运行效率。通过在 EJB Bean 类中重载 `setSessionConext` 方法能够实现这种在不同 EJB 实例间对同一 BeanFactory 实例的使用。具体细节, 请开发者参考 Spring Team 提供的指南文档。

2. AbstractJmsMessageDrivenBean

继续在 example16 基础上开发、部署以及测试消息驱动 Bean 吧! 如果开发者直接借助于 AbstractJmsMessageDrivenBean 开发 MDB, 则能够借助于 Spring EJB 抽象提供的服务。

首先, 开发 MDB Bean 类, 即 Example16MessageDrivenBean。可以看出, 这同会话 Bean 的 Bean 类类似。

```

package com.openv.spring;

import org.springframework.ejb.support.AbstractJmsMessageDrivenBean;

import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.TextMessage;

/**

```

```

* Example16MessageDrivenBean MDB
*
* @author luoshifei
*/
public class Example16MessageDrivenBean extends AbstractJmsMessageDrivenBean
{
    private ISBusinessExample16 sbe;

    protected void onEjbCreate() {

        //简化处理 Spring。即本实例参考本书 SBusinessExample16Bean 中 Spring 的用法。

        System.out.println("onEjbCreate().....");
        sbe = (ISBusinessExample16) getBeanFactory().getBean("sbe16");
    }

    public void onMessage(Message message) {
        System.out.println("onMessage().....");

        if (message instanceof TextMessage) {
            TextMessage tm = (TextMessage) message;

            try {
                System.out.println(sbe.getStr(tm.getText()));
            } catch (JMSEException e) {
                System.out.println(e.getMessage());
            }
        }
    }
}

```

然后，修改 ejb-jar.xml 部署描述符。

```

<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/j2ee" version="2.1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd">
    <display-name>SBusinessExample16Bean</display-name>

    <enterprise-beans>
        <session>
            <ejb-name>SBusinessExample16Bean</ejb-name>
            <home>com.openv.spring.SBusinessExample16Home</home>
            <remote>com.openv.spring.SBusinessExample16Remote</remote>
            <ejb-class>com.openv.spring.SBusinessExample16Bean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>

```

```

    <env-entry>
      <env-entry-name>ejb/BeanFactoryPath</env-entry-name>
      <env-entry-type>java.lang.String</env-entry-type>
      <env-entry-value>appcontext.xml</env-entry-value>
    </env-entry>
  </session>
  <message-driven>
    <display-name>Example16MessageDrivenBean</display-name>
    <ejb-name>Example16MessageDrivenBean</ejb-name>
    <ejb-class>
      com.openv.spring.Example16MessageDrivenBean
    </ejb-class>
    <transaction-type>Container</transaction-type>
    <message-destination-type>
      javax.jms.Queue
    </message-destination-type>
    <env-entry>
      <env-entry-name>ejb/BeanFactoryPath</env-entry-name>
      <env-entry-type>java.lang.String</env-entry-type>
      <env-entry-value>appcontext.xml</env-entry-value>
    </env-entry>
  </message-driven>
</enterprise-beans>

```

```
</ejb-jar>
```

请注意其中的粗体部分。

第三，修改 jboss.xml 部署描述符。

```

<!DOCTYPE jboss PUBLIC
  "-//JBoss//DTD JBOSS 4.0//EN"
  "http://www.jboss.org/j2ee/dtd/jboss_4_0.dtd">
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>SBusinessExample16Bean</ejb-name>
      <jndi-name>SBusinessExample16Bean</jndi-name>
    </session>
    <message-driven>
      <ejb-name>Example16MessageDrivenBean</ejb-name>
      <destination-jndi-name>queue/testQueue</destination-jndi-name>
    </message-driven>
  </enterprise-beans>
</jboss>

```

请注意其中的粗体部分。

第四，部署 EJB jar，即 sbel6.jar。最后，开发发送消息的客户端。

```
package com.openv.spring;
```

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

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

import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;

import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.Session;

/**
 * Example16MessageDrivenBeanClient, 用于发送消息。
 *
 * @author luoshifei
 */
public class Example16MessageDrivenBeanClient {
    protected static final Log log =
        LogFactory.getLog(Example16MessageDrivenBeanClient.class);

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

        JmsTemplate jt = (JmsTemplate) ac.getBean("jmsTemplate");

        for (int i = 0; i < 5; ++i) {
            jt.send(new MessageCreator() {
                public Message createMessage(Session session)
                    throws JMSEException {
                    return session.createTextMessage(
                        "Hello World(MDB),queue/testQueue!," +
                        System.currentTimeMillis());
                }
            });

            log.info("成功发送消息!");
        }
    }
}
```

在 JBoss 控制台将出现如下类似信息。

```
21:26:11,302 INFO [EjbModule] Deploying SBusinessExample16Bean
21:26:11,382 INFO [EjbModule] Deploying Example16MessageDrivenBean
```

```
21:26:11,552 WARN [JMSContainerInvoker] No message-driven destination given;
using; guessing type
21:26:11,642 INFO [EJBDeployer] Deployed: file:/D:/jboss-4.0.0/server/
default/deploy/sbel6.jar
21:26:23,970 INFO [ContextJndiBeanFactoryLocator] BeanFactoryPath from JNDI
is [appcontext.xml]
21:26:24,050 INFO [XmlBeanDefinitionReader] Loading XML bean definitions from
class path resource [appcontext.xml]
21:26:24,090 INFO [ClassPathXmlApplicationContext] Bean factory for
application context [org.springframework.context.support.
ClassPathXmlApplicationContext;hashCode=7761603]: org.springframework.bean
s.factory.support.DefaultListableBeanFactory defining beans [sbel6]; root of
BeanFactory hierarchy
21:26:24,090 INFO [ClassPathXmlApplicationContext] 1 beans defined in
application context [org.springframework.context.support.
ClassPathXmlApplicationContext;hashCode=7761603]
21:26:24,100 INFO [ClassPathXmlApplicationContext] No message source found
for context [org.springframework.context.support.
ClassPathXmlApplicationContext;hashCode=7761603]: using empty default
21:26:24,100 INFO [ClassPathXmlApplicationContext] No
ApplicationEventMulticaster found for context
[org.springframework.context.support.ClassPathXmlApplicationContext;hashC
ode=7761603]: using default
21:26:24,100 INFO [ClassPathXmlApplicationContext] Refreshing listeners
21:26:24,110 INFO [DefaultListableBeanFactory] Pre-instantiating singletons
in factory [org.springframework.beans.factory.support.
DefaultListableBeanFactory defining beans [sbel6]; root of BeanFactory
hierarchy]
21:26:24,110 INFO [DefaultListableBeanFactory] Creating shared instance of
singleton bean 'sbel6'
21:26:24,150 INFO [STDOUT] onEjbCreate().....
21:26:24,150 INFO [STDOUT] onMessage().....
21:26:24,150 INFO [STDOUT] Hello,Hello
World(MDB),queue/testQueue!,1103981183850!
21:26:24,170 INFO [STDOUT] onMessage().....
21:26:24,170 INFO [STDOUT] Hello,Hello
World(MDB),queue/testQueue!,1103981184160!
21:26:24,190 INFO [STDOUT] onMessage().....
21:26:24,190 INFO [STDOUT] Hello,Hello
World(MDB),queue/testQueue!,1103981184180!
21:26:24,210 INFO [STDOUT] onMessage().....
21:26:24,210 INFO [STDOUT] Hello,Hello
World(MDB),queue/testQueue!,1103981184200!
21:26:24,230 INFO [STDOUT] onMessage().....
21:26:24,230 INFO [STDOUT] Hello,Hello
World(MDB),queue/testQueue!,1103981184220!
```


因此，可以看出，Spring EJB 为开发 EJB 应用做了大量的工作。

10.2.2 访问 EJB

为调用 EJB 组件，开发者需要在客户代码中使用 JNDI 查找，以获得对 EJB Home 对象的引用。无论 EJB 组件是本地，还是远程类型，客户代码都需要完成上述工作。在获得 EJB Home 对象后，然后需要调用 create 方法，以获得对 EJB 对象的引用。最后，通过调用远程接口中的业务方法，以完成对 EJB Bean 类中业务方法的调用。

当然，为避免重复级的代码，EJB 应用需要使用服务定位器或者业务委派模式（比如，在 example11 就使用了业务委派模式）。这对于直接通过 JNDI 查找获得 EJB 而言，更具优势。但是，这使得对 EJB 的使用太依赖于这些模式实现。对于只使用了业务定位器的 EJB 客户应用而言，客户代码需要处理 EJB 编程模型中的 API，这不利于应用的封装性。对于使用了业务委派模式的 EJB 客户应用而言，在业务逻辑中将会出现大量的重复代码。因此，Spring 为解决上述问题，引入了访问 EJB 的 Spring 抽象。

基于 Spring IoC 和 Spring AOP 访问 EJB 组件，使得在 EJB 客户应用只能够看到对业务方法的调用，而后端 EJB 组件的业务逻辑全部配置在 Spring BeanFactory 中。

对于本地会话 Bean 而言，Spring 提供了：

```
org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean
```

对于远程会话 Bean 而言，Spring 提供了：

```
org.springframework.ejb.access.SimpleRemoteStatelessSessionProxyFactoryBean
```

比如，为实现对 SBusinessExample16Bean 无状态会话 Bean 的访问，开发者可以提供如下 Spring 配置文件，即 appcontextclient.xml。

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

    <bean id="sbe16client"
        class="org.springframework.ejb.access.
            SimpleRemoteStatelessSessionProxyFactoryBean">
        <property name="jndiName">
            <value>SBusinessExample16Bean</value>
        </property>
        <property name="businessInterface">
            <value>com.openv.spring.ISBusinessExample16</value>
        </property>
    </bean>

</beans>
```

由于 SBusinessExample16Bean 是远程会话 Bean，因此需要在 Spring 配置文件中使
用 org.springframework.ejb.access.SimpleRemoteStatelessSessionProxyFactoryBean。

SBusinessExample16Client.java 的代码很简单。

```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

/**
 * SBusinessExample16Bean EJB 客户应用
 *
 * @author luoshifei
 */
public class SBusinessExample16Client {
    protected static final Log log =
        LogFactory.getLog(SBusinessExample16Client.class);

    public static void main(String[] args) {
        Resource resource = new ClassPathResource("appcontextclient.xml");
        BeanFactory factory = new XmlBeanFactory(resource);
        ISBusinessExample16 se =
            (ISBusinessExample16) factory.getBean("sbel6client");
        log.info(se.getStr("luoshifei"));
    }
}
```

其运行结果见如下。

```
2004-11-13 11:53:46 org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
信息: Loading XML bean definitions from class path resource
[appcontextclient.xml]
2004-11-13 11:53:47
org.springframework.beans.factory.support.AbstractBeanFactory getBean
信息: Creating shared instance of singleton bean 'sbel6client'
2004-11-13 11:53:48 org.springframework.jndi.JndiLocatorSupport lookup
信息: Located object with JNDI name [SBusinessExample16Bean]:
value=[SBusinessExample16BeanHome]
2004-11-13 11:53:48 org.springframework.core.CollectionFactory <clinit>
信息: Using JDK 1.4 collections
2004-11-13 11:53:49 com.openv.spring.SBusinessExample16Client main
信息: Hello,luoshifei!
```

因此，借助于 Spring EJB 抽象，开发者在 EJB 客户代码中很难看到 EJB 组件的逻辑，这对于那些配置性极强的应用而言太有必要了。

10.3 小 结

本章详细研究了 Spring EJB 抽象提供的功能。其中,主要有对 EJB 开发的简化和对 EJB 访问的简化。结合 example16 给出的研究实例对于理解 Spring 框架提供的 EJB 抽象很有帮助。

开发者是否注意到,借助于 Spring IoC 容器能够实现 POJO 的自动管理、借助于 EJB 使用的 J2EE 平台服务能够实现比 Spring AOP 更强大的功能,而且能够使用到 J2EE 服务器的群集技术。可以看出,这同 EJB 3.0 类似。开发者可以去 JCP 查看 EJB 3.0 的进展情况,以及 EJB 3.0 的具体情况 (<http://www.jcp.org/en/jsr/detail?id=220>)。

第 11 章 持久化服务——DAO、JDBC、ORM

对于 RDBMS 相关的应用而言，应用提供的持久化服务很重要。如果持久化服务使用不当，则对于整个系统而言都会是瓶颈。在开发 Java/J2EE 应用过程中，这也是对系统架构师的挑战。如果基于 JDBC 直接同 RDBMS 交互，则应用代码需要实现很多同业务逻辑不相关的代码，同时会引入大量的 Bug。这对于开发应用而言，太不合理了。如果借助于 O/R Mapping 技术实现同 RDBMS 交互，则尽管应用代码不需要过多地处理 JDBC 级的数据，但是应用必须遵循具体的 O/R Mapping 设计要求而架构相应的业务系统。如果日后需要更换 O/R Mapping 工具或者技术，则直接使用 O/R Mapping 又显得不够灵活，因为大量的代码同这些 O/R Mapping 工具绑在了一起。

因此，如何解决这些持久化问题，就摆在开发者的“桌面”上。

Spring 框架提供的 DAO 抽象能够很好地解决这些问题。

11.1 背 景

为使开发者能够以统一的方式同数据访问技术（比如，JDBC、Hibernate、JDO、TopLink）进行交互，可以使用 Spring 框架提供的 DAO 抽象。Spring DAO 抽象允许开发者在不同数据访问技术间切换，而且在互换的同时不用考虑异常处理。因为，无论采用哪种 DAO 技术，Spring 都提供了统一的异常处理机制。

在 Java/J2EE 领域，JDBC 是访问 RDBMS 的标准。在 O/R Mapping 技术出来前，开发者需要直接借助于 JDBC 和 SQL 进行 RDBMS 操作。在 O/R Mapping 技术出来后，开发者能够有更多的选择来实现与 RDBMS 的交互。借助于 O/R Mapping 技术，开发者能够将对象属性映射到 RDBMS 表的列、将对象映射到 RDBMS 表、这些 Mapping 技术能够为应用自动创建高效的 SQL 语句等。除此之外，O/R Mapping 技术还提供延迟装载、（分布式）缓存等高级特性。因此，使用 O/R Mapping 技术，能够大量节省开发者的开发时间，无论是代码量还是开发的成本上，考虑使用 O/R Mapping 技术是值得的。

当然，在特定场合，还是需要直接借助于 JDBC 实现与 RDBMS 的交互。Spring 为 JDBC 提供了抽象层，这使得开发者使用 JDBC 更为高效。依据作者的经验，对于常见的单个 SQL 操作，使用 Spring 会比不使用 Spring 少 70%-80% 的代码量。在不使用 Spring 提供的 JDBC 抽象时，开发者需要创建数据库连接、创建 PreparedStatement 语句、关闭 ResultSet、关闭 PreparedStatement、关闭数据库连接、处理 SQLException 等。这些与 CRUD 无关的数据库操作都被 Spring 提供的 JDBC 抽象服务考虑到了，因此开发者没有理由不使用 Spring 提供

的 JDBC 抽象服务。另外，就目前的 O/R Mapping 技术而言，大部分局限在 RDBMS 表相关的操作，还不能够使用到存储过程等其他操作类型。这对于以数据为中心的应用而言，使用表之外的 RDBMS 功能尤为突出。因此，提供 Spring JDBC 抽象是实用的、必需的。

对于领域模型驱动的应用而言，OO 是最为基础的技术，然而在持久化技术广为使用的今天，OO 同 RDBMS 有点不相称的味道。O/R Mapping 能够完成对象到关系的映射；O/X (Object/XML Mapping) 能够完成对象到 XML 的映射。从如下角度出发，关系同 XML 的含义是一致的：关系更多的是从横向看待问题；而 XML 更多的是从纵向看待问题。无论是横向还是纵向，JavaBean 表达这种关系尤为适合，因为通过 JavaBean 的属性能够关联到其他的 JavaBean。因此，JavaBean (俗名 POJO) 在持久化服务中担任了重要的角色，比如 Hibernate (包括处于制定中的 EJB 3.0 持久化模型)。

借助于 O/R Mapping 技术，能够解决工业领域 80% 左右的持久化问题。常见的 O/R Mapping 技术有 JDO、Hibernate、Apache OJB、iBATIS SQL Maps、CMP、TopLink。Spring 为这些技术提供了一流的支持能力 (除了 CMP 之外，尽管通过调整 J2EE 应用服务器自身的参数能够改善实体 Bean 的性能，但是其使用面还是有限，尤其是轻量级持久化机制迅速发展的今天)。

通过使用 Spring 对它们的集成支持，开发者还能够享受到这种集成所带来的其他优势。比如，智能处理异常的能力、集成事务管理能力 (参见本书第 7 章内容)、通过 Spring 提供的线程安全模板类 (通过可重入实现)、资源管理能力 (比如连接池、数据库连接)、实用类等。

11.2 Spring 对 DAO 提供的支持

通过 Spring DAO 抽象，能够将具体 DAO 技术的异常，比如 `HibernateException`、`SQLException`，转换成以 `DataAccessException` 为根的异常处理体系中 (本书在第 5 章给出过阐述)。与此同时，由于以 `DataAccessException` 为根的异常处理体系中的异常类只是对具体 DAO 技术的异常进行包裹，即不会丢失任何异常信息，因此使用 `DataAccessException` 很安全。

对于使用 O/R Mapping 访问框架中的模板 (比如，`JdbcTemplate`、`HibernateTemplate`) 的情形而言，开发者不用考虑是否需要处理异常，因为 Spring DAO 代劳了这些工作。对于使用拦截器 (比如，`HibernateInterceptor`) 的情形而言，开发者必须手工处理具体 DAO 技术中的异常。当然，这对于那些需要处理业务相关异常的系统而言，更加灵活。借助于 Spring 提供的 `SessionFactoryUtils` 实用类能够将具体 DAO 技术中的异常转化为 Spring DAO 抽象中定义的异常。

Spring 提供了一套抽象 DAO 类，供开发者扩展，这有利于以统一的方式操作各种 DAO 技术，比如 JDO、JDBC。其中，这些抽象 DAO 类提供了设置数据源及相关辅助信息的方法，而其中的一些方法同具体 DAO 技术相关。目前，Spring DAO 抽象提供如下几种：

- `JdbcDaoSupport`: JDBC DAO 抽象类。开发者需要为它配置数据源 (`DataSource`)。通过其子类，开发者能够获得 `JdbcTemplate`。当然，也可以手工配置 `JdbcTemplate`。
- `HibernateDaoSupport`: Hibernate DAO 抽象类。开发者需要为它配置 `Hibernate`

SessionFactory。通过其子类，开发者能够获得 HibernateTemplate。当然，也可以手工配置 HibernateTemplate。

- JdoDaoSupport: Spring 为 JDO 提供的 DAO 抽象类。开发者需要为它配置 PersistenceManagerFactory。通过其子类，开发者能够获得 JdoTemplate。当然，也可以手工配置 JdoTemplate。

11.3 Spring 对 JDBC 提供的支持

JDBC 对于 Java 开发者而言，再熟悉不过了。无论何种 O/R Mapping 技术，其最终与 RDBMS 交互的技术一般都是 JDBC。因此，在 Java/J2EE 领域中，JDBC 是最为重要、基础的技术。

直接采用 JDBC 开发应用，开发者能够细粒度调整访问 RDBMS 的性能，而且能够使用到 RDBMS 专属的功能。对于主流 RDBMS 而言，一般都存在相应的 JDBC 规范实现，比如用于开发者的 JDBC Driver、JDBC 实用类等。

然而，JDBC 同 RDBMS 的关系是紧耦合的，即不便于在不同的 RDBMS 间移植。尽管 JDBC 规范的目标是实现 Java 数据库应用的便携，然而实际场合中，往往需要提供 RDBMS 专属的功能。比如，当今主流的 Hibernate 都需要为各种 RDBMS 提供不同的方言 (Dialect)，因此完全的 Java 数据库应用只是理想而言，离现实还是有些距离。同其他 J2EE 组件技术一样，Spring 也提供了相应的服务抽象。

借助于 Spring 提供的 JDBC 服务抽象，能够消除很多重复代码，比如异常处理、数据库连接、资源的打开和关闭等。

11.3.1 JdbcTemplate

JdbcTemplate 同其他 J2EE 组件技术类似，比如 JMS、JTA，是 Spring 为借助于 JDBC 操作 RDBMS 而提供的实用模板类。JdbcTemplate 自身能够自动管理数据库连接资源的打开和关闭操作，因此它能够简化 JDBC 的使用。另外，通过使用 JdbcTemplate，开发者能够避免很多 JDBC 相关的错误，因为 JdbcTemplate 代劳了很多基础工作。相信有过其他 Spring 模板使用经验的开发者都能够体会到 JdbcTemplate 的优势。因为，Spring 希望开发者快乐地开发 J2EE 应用，而这种快乐是建立在提供“模板”的基础上的。而这种模板是建立在模板方法 (Template Method) 模式的基础上的。图 11-1 给出了 JdbcTemplate 相关类的关系。

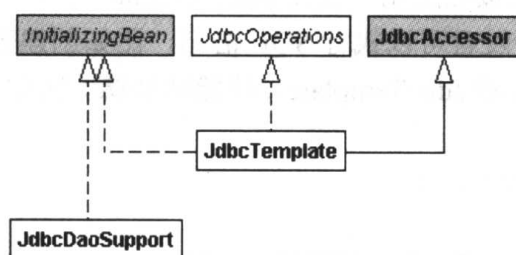


图 11-1 JdbcTemplate 相关类的 UML 图

本节将结合 example17 展开研究。开发者在使用 JdbcTemplate 之前，需要配置 JdbcTemplate 使用到的 DataSource。其中，appcontext.xml 内容如下。配置的数据源使用了 Apache DBCP，通过指定 driverClassName、url、username、password 等属性能够创建数据源，供 JdbcTemplate 引用。另外，由于 Spring 中所有的模板类都是线程安全的，因此对于各个 DataSource 而言，开发者只需要在 Spring 配置文件中配置单个的 JdbcTemplate，即可满足多线程使用。

```
<?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="jdbcTemplate"
        class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource">
            <ref local="dataSource"></ref>
        </property>
    </bean>

    <bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
        <property name="driverClassName">
            <value>org.gjt.mm.mysql.Driver</value>
        </property>
        <property name="url">
            <value>jdbc:mysql://localhost:3306/example11</value>
        </property>
        <property name="username">
            <value>root</value>
        </property>
        <property name="password">
            <value></value>
        </property>
    </bean>

</beans>
```

因此，客户代码可以直接在应用中使用 JdbcTemplate 了。

通过 JdbcTemplate，开发者可以执行 SQL 语句，比如创建表、插入数据、更新数据库等。只要提供了 DataSource 和 JdbcTemplate，就能够如愿。比如，创建 jdbc-template 表的客户代码如下。

```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
```



```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

import org.springframework.jdbc.core.JdbcTemplate;

/**
 * JdbcTemplateTest 客户应用
 *
 * @author luoshifei
 */
public class JdbcTemplateTest {
    protected static final Log log = LogFactory.getLog(JdbcTemplateTest.class);

    public static void main(String[] args) {
        Resource resource = new ClassPathResource("appcontext.xml");
        BeanFactory factory = new XmlBeanFactory(resource);
        JdbcTemplate jt = (JdbcTemplate) factory.getBean("jdbcTemplate");
        jt.execute(
            "create table jdbcTemplate(id int ,templatename varchar(100))");
    }
}
```

通过运行 Ant build.xml 中的 run 任务，能够输出如下结果。

```
Buildfile: D:\workspace\example17\build.xml
compile:
run:
[java] 2004-11-13 22:02:59 org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
[java] 信息: Loading XML bean definitions from class path resource
[appcontext.xml]
[java] 2004-11-13 22:02:59 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'jdbcTemplate'
[java] 2004-11-13 22:02:59 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'dataSource'
[java] 2004-11-13 22:02:59 org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
[java] 信息: Loading XML bean definitions from class path resource
[org/springframework/jdbc/support/sql-error-codes.xml]
[java] 2004-11-13 22:02:59 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
```

```
[java] 信息: Creating shared instance of singleton bean 'DB2'
[java] 2004-11-13 22:02:59 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'HSQL'
[java] 2004-11-13 22:02:59 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'MS-SQL'
[java] 2004-11-13 22:02:59 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'MySQL'
[java] 2004-11-13 22:02:59
org.springframework.beans.factory.support.AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'Oracle'
[java] 2004-11-13 22:02:59 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'Informix'
[java] 2004-11-13 22:02:59 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'PostgreSQL'
[java] 2004-11-13 22:02:59 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'Sybase'
[java] 2004-11-13 22:02:59 org.springframework.jdbc.support.
SQLExceptionCodesFactory <init>
[java] 信息: SQLExceptionCodes loaded: [HSQL Database Engine, Oracle, Sybase
SQL Server, Microsoft SQL Server, Informix Dynamic Server, PostgreSQL, DB2*,
MySQL]
[java] 2004-11-13 22:02:59 org.springframework.jdbc.support.
SQLExceptionCodesFactory getErrorCodes
[java] 信息: Looking up default SQLExceptionCodes for DataSource
BUILD SUCCESSFUL
Total time: 2 seconds
```

通过 MySQL 控制台，开发者能够获得刚创建的 jdbctemplate 表的信息，具体如下。

```
D:\jboss-4.0.0\bin>mysql
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 8 to server version: 4.1.5-gamma-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> use example11;
Database changed
mysql> show tables;
+-----+
| Tables_in_example11 |
+-----+
| interest             |
+-----+
```

```
| jdbctemplate      |
| userinfo          |
| userinterest      |
+-----+
4 rows in set (0.00 sec)
```

```
mysql> desc jdbctemplate;
```

```
+-----+-----+-----+-----+-----+-----+
| Field      | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id         | int(11)    | YES  |     | NULL    |       |
| templatenam | varchar(100) | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.01 sec)
```

```
mysql>
```

当然, `JdbcTemplate` 支持的 JDBC 操作很多。同 Spring 其他模板类似, Spring DAO JDBC 抽象还提供了同 `JdbcTemplate` 配套使用的若干回调接口, 比如 `PreparedStatementCreator`、`CallableStatementCreator`、`RowCallbackHandler` 接口。

example18 实例将给出 `RowCallbackHandler` 接口的使用。 `JdbcTemplateTest.java` 代码如下。

```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowCountCallbackHandler;

/**
 * JdbcTemplateTest 客户应用
 *
 * @author luoshifei
 */
public class JdbcTemplateTest {
    protected static final Log log = LogFactory.getLog(JdbcTemplateTest.class);

    public static void main(String[] args) {
```

```

Resource resource = new ClassPathResource("appcontext.xml");
BeanFactory factory = new XmlBeanFactory(resource);
JdbcTemplate jt = (JdbcTemplate) factory.getBean("jdbcTemplate");
RowCountCallbackHandler rcch = new RowCountCallbackHandler();
jt.query("select * from jdbcTemplate", rcch);
log.info("结果集中的列数量: " + rcch.getColumnCount());
log.info("结果集中的行数量: " + rcch.getRowCount());
log.info("结果集中的列名: ");

String[] str = rcch.getColumnNames();

for (int i = 0, k = str.length; i < k; ++i) {
    log.info(str[i]);
}
}
}

```

其中, **RowCountCallbackHandler** 实现了 **RowCallbackHandler** 接口。在执行客户应用之前, 请通过 **MySQL** 控制台输入一些示范数据。比如:

```

mysql> insert into jdbcTemplate values(3,'afdaf');
Query OK, 1 row affected (0.04 sec)

mysql> insert into jdbcTemplate values(4,'afafdafdaf');
Query OK, 1 row affected (0.00 sec)

mysql> insert into jdbcTemplate values(1,'afafdafadfafd');
Query OK, 1 row affected (0.00 sec)

mysql> insert into jdbcTemplate values(2,'afafd');
Query OK, 1 row affected (0.00 sec)

mysql> select * from jdbcTemplate;
+-----+-----+
| id  | templatename |
+-----+-----+
| 3  | afdaf       |
| 4  | afafdafdaf  |
| 1  | afafdafadfafd |
| 2  | afafd       |
+-----+-----+
4 rows in set (0.00 sec)

mysql>

```

然后, 执行 **Ant build.xml** 中的 **run** 任务, 其运行结果如下。

```

Buildfile: D:\workspace\example18\build.xml
compile:
run:

```

```
[java] 2004-11-13 22:35:40
org.springframework.beans.factory.xml.XmlBeanDefinitionReader
loadBeanDefinitions
[java] 信息: Loading XML bean definitions from class path resource
[appcontext.xml]
[java] 2004-11-13 22:35:40
org.springframework.beans.factory.support.AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'jdbcTemplate'
[java] 2004-11-13 22:35:40
org.springframework.beans.factory.support.AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'dataSource'
[java] 2004-11-13 22:35:41
org.springframework.beans.factory.xml.XmlBeanDefinitionReader
loadBeanDefinitions
[java] 信息: Loading XML bean definitions from class path resource
[org/springframework/jdbc/support/sql-error-codes.xml]
[java] 2004-11-13 22:35:41
org.springframework.beans.factory.support.AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'DB2'
[java] 2004-11-13 22:35:41
org.springframework.beans.factory.support.AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'HSQL'
[java] 2004-11-13 22:35:41
org.springframework.beans.factory.support.AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'MS-SQL'
[java] 2004-11-13 22:35:41
org.springframework.beans.factory.support.AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'MySQL'
[java] 2004-11-13 22:35:41
org.springframework.beans.factory.support.AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'Oracle'
[java] 2004-11-13 22:35:41
org.springframework.beans.factory.support.AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'Informix'
[java] 2004-11-13 22:35:41
org.springframework.beans.factory.support.AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'PostgreSQL'
[java] 2004-11-13 22:35:41
org.springframework.beans.factory.support.AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'Sybase'
[java] 2004-11-13 22:35:41
org.springframework.jdbc.support.SQLErrorCodesFactory <init>
[java] 信息: SQLErrorCodes loaded: [HSQL Database Engine, Oracle, Sybase
SQL Server, Microsoft SQL Server, Informix Dynamic Server, PostgreSQL, DB2*,
MySQL]
[java] 2004-11-13 22:35:41
```

```
org.springframework.jdbc.support.SQLExceptionCodesFactory getErrorCodes
[java] 信息: Looking up default SQLExceptionCodes for DataSource
[java] 2004-11-13 22:35:41 com.openv.spring.JdbcTemplateTest main
[java] 信息: 结果集中的列数量: 2
[java] 2004-11-13 22:35:41 com.openv.spring.JdbcTemplateTest main
[java] 信息: 结果集中的行数量: 4
[java] 2004-11-13 22:35:41 com.openv.spring.JdbcTemplateTest main
[java] 信息: 结果集中的列名:
[java] 2004-11-13 22:35:41 com.openv.spring.JdbcTemplateTest main
[java] 信息: id
[java] 2004-11-13 22:35:41 com.openv.spring.JdbcTemplateTest main
[java] 信息: templatename
BUILD SUCCESSFUL
Total time: 3 seconds
```

11.3.2 DataSourceTransactionManager

对于单一 JDBC 数据源而言, 开发者可以使用 PlatformTransactionManager 接口的如下具体实现, 即 DataSourceTransactionManager。为使用 DataSourceTransactionManager 实现事务管理, 开发者必须使用 DataSourceUtils.getConnection(DataSource)方法, 以获得 JDBC 连接, 而不能够使用标准的 J2EE DataSource.getConnection。当然, 这主要是考虑到如下两方面的原因。其一, DataSourceUtils.getConnection(DataSource)能够抛出 Spring DAO 抽象框架中的未受查异常, 因此便于 DAO 抽象框架进行异常信息的处理。其二, 界定事务边界, 即确保返回的数据库连接绑定到当前线程, 从而实现事务管理功能。

另外, DataSourceTransactionManager 支持设置事务超时时间, 即通过 DataSourceUtils的 applyTransactionTimeout 方法。对于不要求支持分布式事务的场合, 即使用 JTA, 开发者只需要使用 DataSourceTransactionManager 实现。

11.3.3 连接数据库的方式

JdbcTemplate 在操作数据库的过程中, 开发者需要为它提供数据源。一般情况下, 存在如下几种 RDBMS 数据源 (图 11-2 给出了它们之间的关系)。

- SingleConnectionDataSource: 实现了 SmartDataSource 接口。不具备多线程能力。主要供开发时使用。
- DriverManagerDataSource: 实现了 SmartDataSource 接口。每次客户请求, 都会返回新的 JDBC 连接。

另外, 基于目标环境的不同, 可以使用不同的数据源类型, 比如 DBCP BasicDataSource。当然, 在实际应用环境中, 基于 JNDI 获取数据源比较常见。

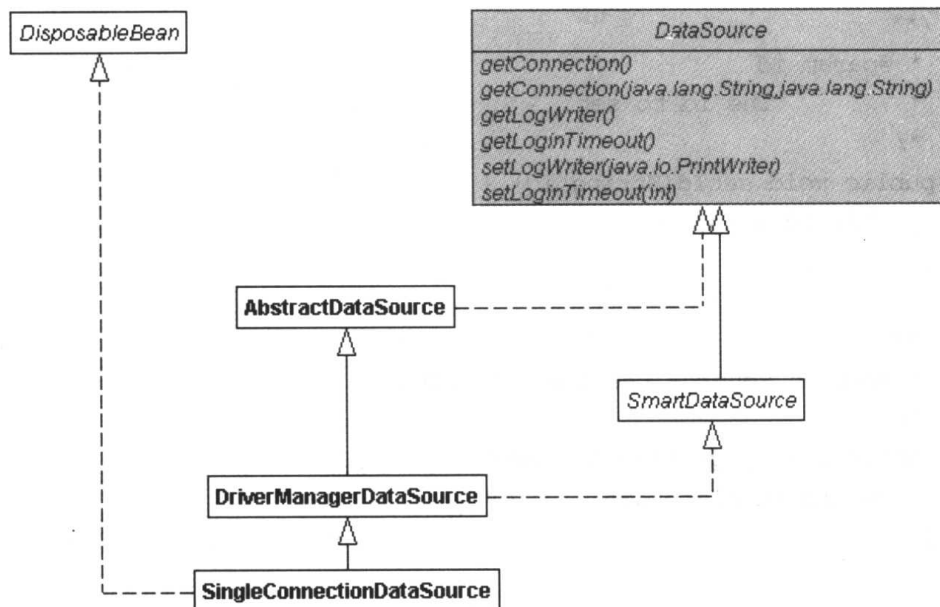


图 11-2 DataSource 相关类的 UML 图

11.3.4 将 JDBC 操作建模为 Java 对象

其中, `org.springframework.jdbc.object` 包中提供的类允许开发者以 OO 方式访问数据库。借助于 `object` 包, 开发者能够将查询结果中的数据库列映射到业务对象的属性上。与此同时, 还可以执行存储过程, 或者运行更新、删除、插入操作。

本书将结合 example19 给出一简单实例的研究。首先, 实现 `JdbcTemplateVO` 值对象。

```

package com.openv.spring;

import java.io.Serializable;

/**
 * 对应于 jdbcTemplate 表
 *
 * @author luoshifei
 */
public class JdbcTemplateVO implements Serializable {
    private String id;

    private String templatenamename;

    /**
     * @return Returns the id.
     */
    public String getId() {
        return id;
    }
}

```

```
/**
 * @param id
 *         The id to set.
 */
public void setId(String id) {
    this.id = id;
}

/**
 * @return Returns the templatenamename.
 */
public String getTemplatenamename() {
    return templatenamename;
}

/**
 * @param templatenamename
 *         The templatenamename to set.
 */
public void setTemplatenamename(String templatenamename) {
    this.templatenamename = templatenamename;
}
}
```

其次，开发者需要实现 `MappingSqlQuery` 接口类，即 `JdbcTemplateMappingQuery`。其中，`JdbcTemplateMappingQuery` 类实现了 `O/R Mapping`（类似 `Hibernate`、`TopLink`）功能。`RDBMS` 中的 `jdbcTemplate` 表现在将映射到 `JdbcTemplateVO` 上。因此，开发者直接使用 `JdbcTemplateMappingQuery` 便能够操作到 `RDBMS`。

```
package com.openv.spring;

import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.MappingSqlQuery;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Types;

import javax.sql.DataSource;

/**
 * MappingSqlQuery 实现
 *
 * @author luoshifei
 */
public class JdbcTemplateMappingQuery extends MappingSqlQuery {
    public JdbcTemplateMappingQuery(DataSource ds) {
```



```

        super(ds, "select id, templatename from jdbctemplate where id = ?");
        super.declareParameter(new SqlParameter("id", Types.INTEGER));
        compile();
    }

    public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
        JdbcTemplateVO jtVO = new JdbcTemplateVO();
        jtVO.setId(rs.getString("id"));
        jtVO.setTemplatename(rs.getString("templatename"));

        return jtVO;
    }
}

```

第三，定义 appcontext.xml 文件。

```

<?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="jtmq"
        class="com.openv.spring.JdbcTemplateMappingQuery">
        <constructor-arg><ref local="dataSource"></ref></constructor-arg>
    </bean>

    <bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
        <property name="driverClassName">
            <value>org.gjt.mm.mysql.Driver</value>
        </property>
        <property name="url">
            <value>jdbc:mysql://localhost:3306/example11</value>
        </property>
        <property name="username">
            <value>root</value>
        </property>
        <property name="password">
            <value></value>
        </property>
    </bean>

</beans>

```

第四，提供客户应用。

```

package com.openv.spring;

import org.apache.commons.logging.Log;

```

```
import org.apache.commons.logging.LogFactory;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

import java.util.List;

/**
 * Example19Test 客户应用
 *
 * @author luoshifei
 */
public class Example19Test {
    protected static final Log log = LogFactory.getLog(Example19Test.class);

    public static void main(String[] args) {
        Resource resource = new ClassPathResource("appcontext.xml");
        BeanFactory factory = new XmlBeanFactory(resource);
        JdbcTemplateMappingQuery jtmq =
            (JdbcTemplateMappingQuery) factory.getBean("jtmq");
        JdbcTemplateVO jtVO;
        jtVO = (new Example19Test()).getJdbcTemplateVO(jtmq, "2");
        log.info(jtVO.getId() + "," + jtVO.getTemplatename());
    }

    public JdbcTemplateVO getJdbcTemplateVO(JdbcTemplateMappingQuery custQry,
        String id) {
        Object[] parms = new Object[1];
        parms[0] = id;

        List customers = custQry.execute(parms);

        if (customers.size() > 0) {
            return (JdbcTemplateVO) customers.get(0);
        } else {
            return null;
        }
    }
}
```

最后，运行 Ant build.xml 中的 run 任务。

Buildfile: D:\workspace\example19\build.xml
compile:

```
run:
    [java] 2004-11-13 23:18:42 org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
    [java] 信息: Loading XML bean definitions from class path resource
[appcontext.xml]
    [java] 2004-11-13 23:18:42 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'jtmq'
    [java] 2004-11-13 23:18:42 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'dataSource'
    [java] 2004-11-13 23:18:42 org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
    [java] 信息: Loading XML bean definitions from class path resource
[org/springframework/jdbc/support/sql-error-codes.xml]
    [java] 2004-11-13 23:18:42 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'DB2'
    [java] 2004-11-13 23:18:42 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'HSQL'
    [java] 2004-11-13 23:18:42 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'MS-SQL'
    [java] 2004-11-13 23:18:42 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'MySQL'
    [java] 2004-11-13 23:18:42
org.springframework.beans.factory.support.AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'Oracle'
    [java] 2004-11-13 23:18:42 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'Informix'
    [java] 2004-11-13 23:18:42 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'PostgreSQL'
    [java] 2004-11-13 23:18:42 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'Sybase'
    [java] 2004-11-13 23:18:42 org.springframework.jdbc.support.
SQLExceptionCodesFactory <init>
    [java] 信息: SQLExceptionCodes loaded: [HSQL Database Engine, Oracle, Sybase
SQL Server, Microsoft SQL Server, Informix Dynamic Server, PostgreSQL, DB2*,
MySQL]
    [java] 2004-11-13 23:18:42 org.springframework.jdbc.support.
SQLExceptionCodesFactory getErrorCodes
```

```
[java] 信息: Looking up default SQLExceptionCodes for DataSource
[java] 2004-11-13 23:18:43 org.springframework.jdbc.object.
RdbmsOperation compile
[java] 信息: RdbmsOperation with SQL [select id, templatename from
jdbctemplate where id = ?] compiled
[java] 2004-11-13 23:18:43 org.springframework.beans.factory.support.
AbstractAutowireCapableBeanFactory autowireConstructor
[java] 信息: Bean 'jtmq' instantiated via constructor [public
com.openv.spring.JdbcTemplateMappingQuery(javax.sql.DataSource)]
[java] 2004-11-13 23:18:43 com.openv.spring.Example19Test main
[java] 信息: 2,afafd
BUILD SUCCESSFUL
Total time: 2 seconds
```

如果开发者仔细分析 `object` 包，则能够看到图 11-3 中包含的各个类及其相互关系。通过继承 `SqlUpdate`，开发者还能够插入记录到 RDBMS 中或更新 RDBMS。

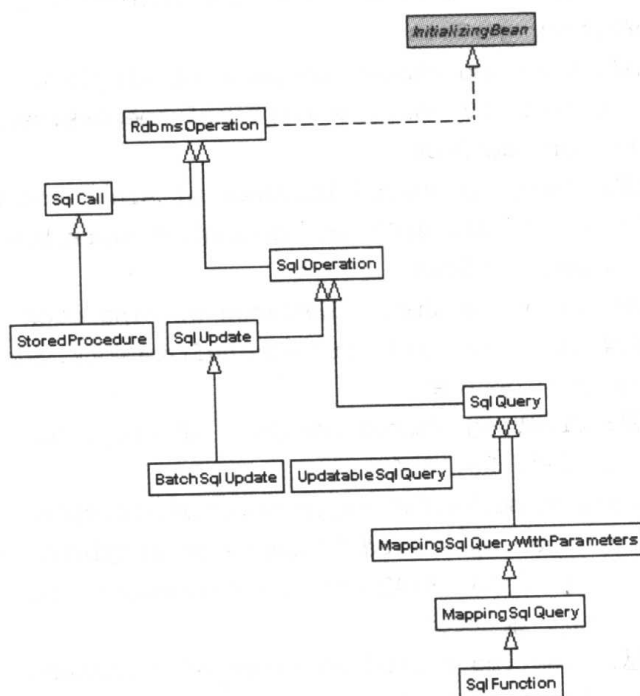


图 11-3 `object` 包中类的 UML 图

至于 Spring DAO JDBC 抽象的更详细内容，请开发者参考 Spring 源码，或者本书给出的其他参考资料。

11.4 Spring 对 ORM 提供的支持

Spring 框架依据资源管理、DAO 实现支持、事务策略，对 Hibernate、JDO、iBATIS SQL Maps 进行了集成。目前，Spring 框架对 Hibernate 进行了最深入的集成，即借助于 Spring IoC 和 Spring AOP 对 Hibernate 集成问题进行了最为有效的集成。从这里也能够看出，Spring

是非常实用的框架，因为 Hibernate 已经成为了事实上标准的 O/R Mapping 技术。

本书先对 Hibernate 进行介绍，然后再来研究 Spring 对 Hibernate 提供的支持。从这里也可以看出，Hibernate 确实是很重要的技术。

11.4.1 Hibernate 介绍

Hibernate 是高性能的 Open Source 框架。它不仅提供了 O/R Mapping 功能，还支持很多高级特性。尤其是 Hibernate 3.0 新增大量的特性，其中包括对存储过程和手工编写 SQL 的支持。通过 XML 配置文件能够将对象映射到 RDBMS，这就是 Hibernate。

还是秉承本书的传统，即结合实例研究具体技术，效果最佳。一般而言，开发 Hibernate 应用需要准备如下内容。

- Hibernate 配置文件。一般可以通过两种方式进行。其一，提供 hibernate.cfg.xml 文件，用于创建 Hibernate SessionFactory。其二，提供 hibernate.properties 文件。本书推荐使用 XML 文件。
- Hibernate 映射文件。比如 Interest.hbm.xml。
- POJO 类源文件。比如 com.openv.spring.Interest.java。

当然，现有的免费支持工具不少，比如 Hibernate Middlegen、Hibernate Synchronizer、Hibernate Tools¹（由 Hibernate Team 开发，本书写作之际还处于 3.0 alpha1 版）。本书推荐使用 Hibernate Synchronizer²和 Hibernate Tools（由于 Synchronizer 很成熟，Hibernate Tools 还处于开发初期，因此本书重点在 Hibernate Synchronizer 上）。

Hibernate Synchronizer 是免费的、能够同 Hibernate 持久框架配合生成代码的工具。它是以 Eclipse 插件方式运行的。当开发者修改了 Hibernate 映射文件后，Hibernate Synchronizer 会自动修改 POJO 代码。其中，生成的代码包括两部分。其一，以抽象基类存在的源代码。其二，开发者能够修改的扩展类。由于 Hibernate Synchronizer 并不会自动同步扩展类，因此不会因为 RDBMS 表结构的修改，而修改了扩展类，这对于在扩展类中开发业务相关逻辑，或者重载抽象类中的方法很有利。借助于 Hibernate Synchronizer，能够自动创建如下类型的对象。

- 值对象
- 代理接口
- 复合键对象
- 枚举对象
- 组件对象
- 子类
- DAO

在开发者下载并安装 Hibernate Synchronizer（通过 Eclipse 的软件更新功能，具体的下载网址位于 <http://www.binamics.com/hibernatesync>）后，就可以使用它了。本书将结合

¹ <http://tools.hibernate.org/>

² <http://www.binamics.com/hibernatesync/>

example11 中使用到的 Interest 表进行阐述。具体内容位于 example20 中。

1. 创建 Hibernate 配置文件

在开发者创建 example20 Java 项目后，将需要的类库包括到项目中，具体内容请参考本书提供的 example20 项目。然后，创建 Hibernate 配置文件，见图 11-4。

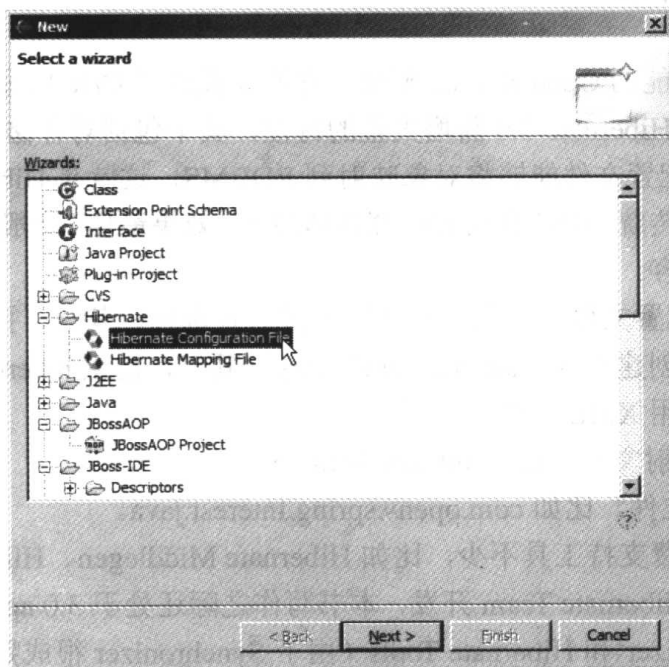


图 11-4 创建 Hibernate 配置文件向导 (1)

然后，在出现的对话框中输入类似于图 11-5 中的内容。

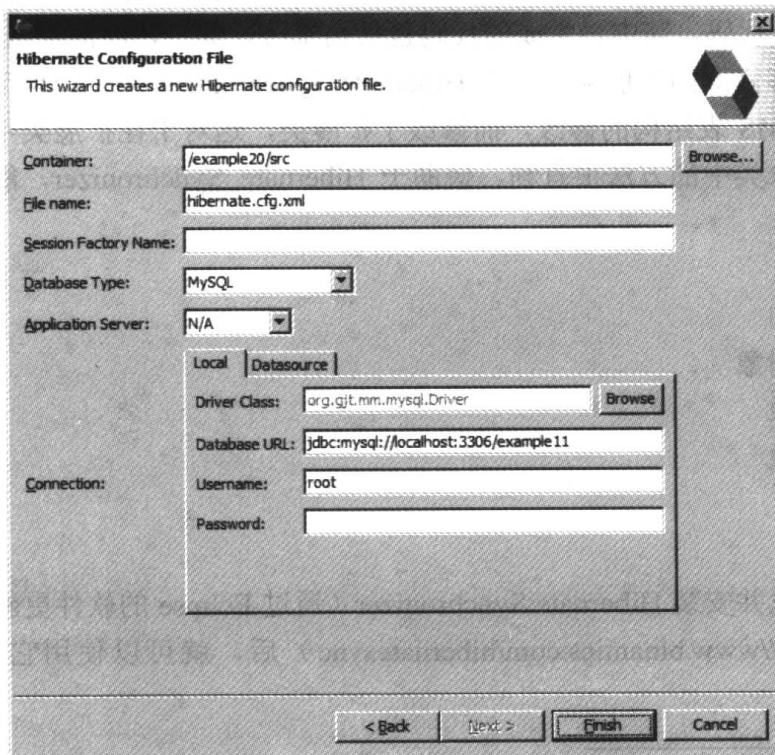


图 11-5 创建 Hibernate 配置文件向导 (2)

然后, 修改自动生成的 `hibernate.cfg.xml` 文件, 以适合自身的具体环境。

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration
PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">

<hibernate-configuration>
  <session-factory>

    <!-- local connection properties -->
    <property name="hibernate.connection.url">
      jdbc:mysql://localhost:3306/example11
    </property>
    <property name="hibernate.connection.driver_class">
      org.gjt.mm.mysql.Driver
    </property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password" />
    <!-- property name="hibernate.connection.pool_size"></property -->
    <!-- dialect for MySQL -->
    <property name="dialect">
      net.sf.hibernate.dialect.MySQLDialect
    </property>
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.use_outer_join">true</property>

  </session-factory>
</hibernate-configuration>
```

可以看出, 开发者很容易就创建了 **Hibernate** 配置文件。

2. 创建 Hibernate 映射文件

首先, 打开 Eclipse 向导, 见图 11-6。

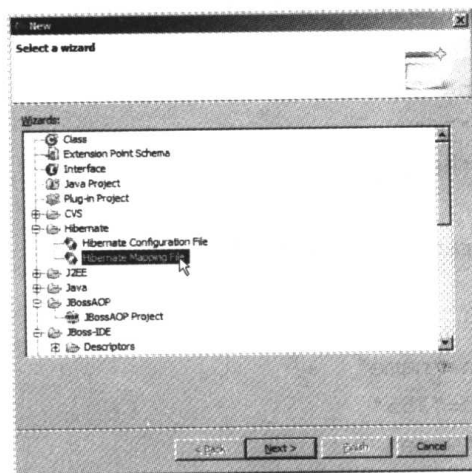


图 11-6 创建 Hibernate 映射文件向导 (1)

然后，在出现的向导中，输入 MySQL 相关 JDBC 信息。继而，单击“Refresh”按钮。最后，输入图 11-7 中的内容。

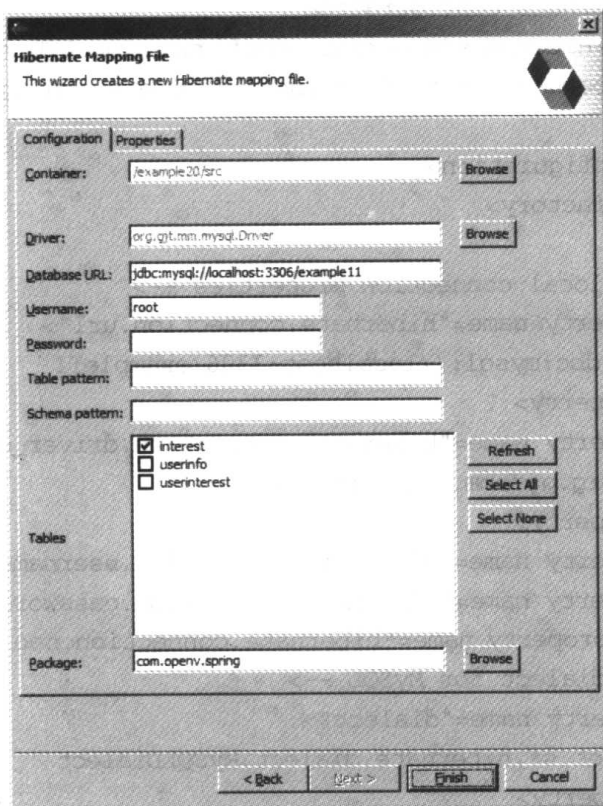


图 11-7 创建 Hibernate 映射文件向导 (2)

最后，开发者可以打开生成的 Interest.hbm 文件。

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd" >

<hibernate-mapping package="com.openv.spring">
    <class name="Interest" table="interest">
        <id
            column="id"
            name="Id"
            type="string"
        >
            <generator class="vm" />
        </id>
        <property
            column="name"
            length="255"
            name="Name"
            not-null="false"
            type="string"
        >
    </class>
</hibernate-mapping>
```



```
    />  
  </class>  
</hibernate-mapping>
```

3. 自动生成 Java 源文件

通过图 11-8 中的“Synchronize Files”菜单项能够自动生成所需的 Java 源文件。

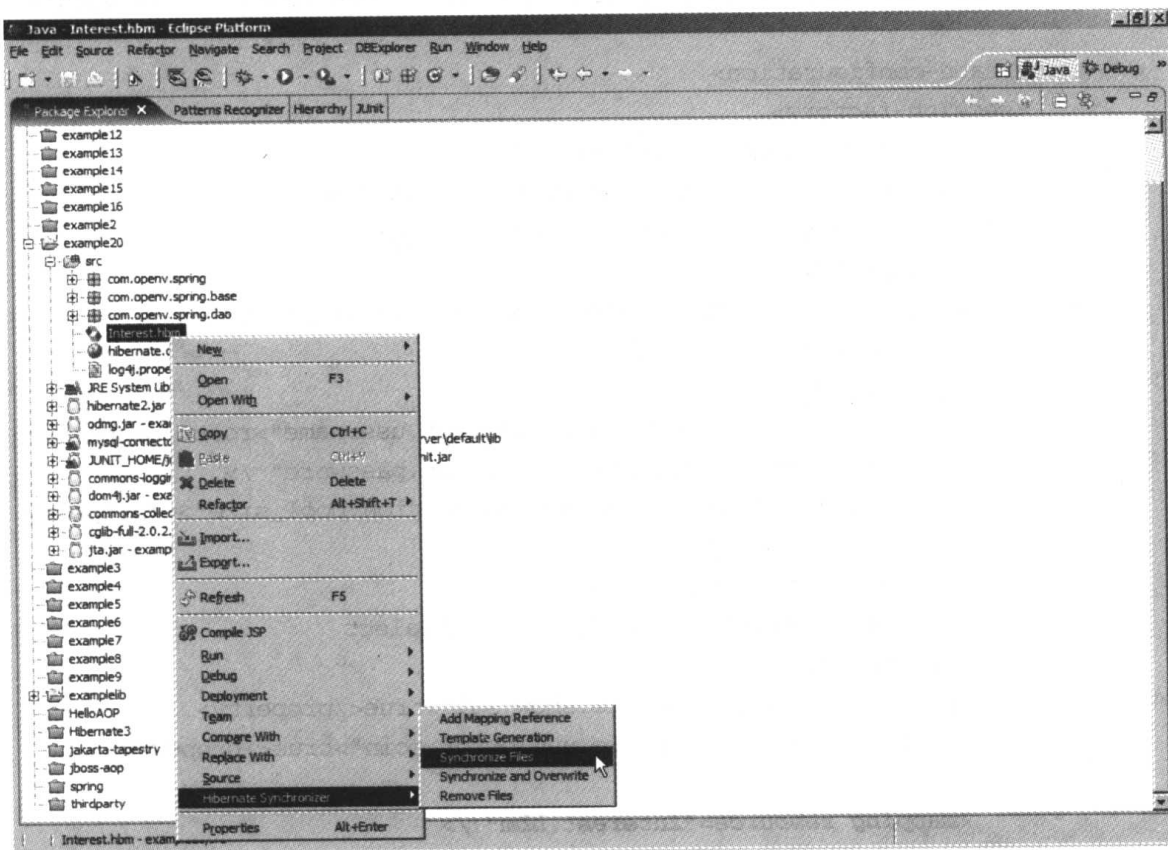


图 11-8 自动生成 Java 源文件的菜单项

同时，还可以通过“Add Mapping Reference”菜单项将 Interest.hbm 添加到 hibernate.cfg.xml 配置文件中，见图 11-9。

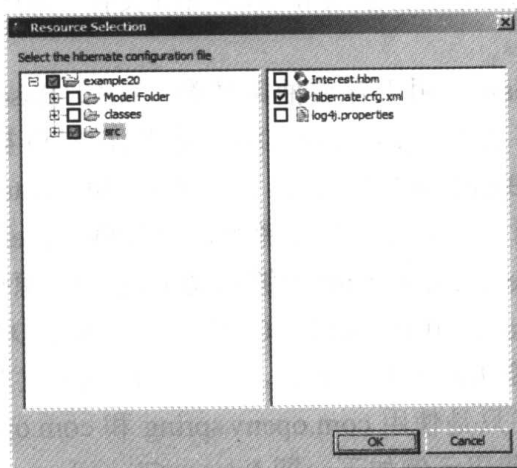


图 11-9 将 Interest.hbm 添加到 hibernate.cfg.xml 中

开发者如果打开 hibernate.cfg.xml 文件能够看到新增了一行内容, 见如下程序段的粗体部分。

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration
  PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">

<hibernate-configuration>
  <session-factory>
    <!-- local connection properties -->
    <property name="hibernate.connection.url">
      jdbc:mysql://localhost:3306/example11
    </property>
    <property name="hibernate.connection.driver_class">
      org.gjt.mm.mysql.Driver
    </property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password" />
    <!-- property name="hibernate.connection.pool_size"></property -->
    <!-- dialect for MySQL -->
    <property name="dialect">
      net.sf.hibernate.dialect.MySQLDialect
    </property>
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.use_outer_join">true</property>

    <mapping resource="Interest.hbm" />
  </session-factory>
</hibernate-configuration>
```

自动生成的 Java 源文件将分别放置在如下包中。

- com.openv.spring: 用于存放 POJO 类, 比如 Interest.java。当 Hibernate Synchronizer 自动同步源代码时, 并不会修改 Interest.java 文件。因此, 开发者可以定制 Interest 的行为。
- com.openv.spring.base: 用于存放抽象基类, 比如 _BaseRootDAO、BaseInterest、BaseInterestDAO。其中, _BaseRootDAO 是所有 DAO 类的基类。开发者应该对它仔细分析, 因为 _BaseRootDAO.java 包含了对 Hibernate API 的各种使用方法。对于开发者日常应用开发而言, 开发者可以时常将 _BaseRootDAO.java 作为参考文件。这是 Hibernate Synchronizer 能够自动生成 Java 源代码的主要卖点之一。
- com.openv.spring.dao: 用于存放 DAO 类, 比如 _RootDAO、InterestDAO。

在开发完成 Hibernate 要求的所有内容后, 开发者可以开发对 InterestDAO 的调用。很明显, Hibernate 客户代码中只是使用 com.openv.spring 和 com.openv.spring.dao 中的类。其中, example20 提供的客户代码示例如下, 即 InterestClient.java。

```
package com.openv.spring;
```

```
import com.openv.spring.dao.InterestDAO;
import com.openv.spring.dao._RootDAO;

import net.sf.hibernate.HibernateException;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import java.io.UnsupportedEncodingException;

import java.util.List;

/**
 * Hibernte Synchronizer 测试代码
 *
 * @author luoshifei
 */
public class InterestClient {
    protected static final Log log = LogFactory.getLog(InterestClient.class);

    public static void main(String[] args) {
        try {
            //初始化 SessionFactory
            _RootDAO.initialize();

            //实例化 InterestDAO
            InterestDAO intrDAO = new InterestDAO();

            //借助于Hibernate 获得所有的兴趣列表
            List list = intrDAO.findAll();

            Interest interest;

            if (null != list) {
                log.info(list.size() + "");
            }

            for (int i = 0, k = list.size(); i < k; ++i) {
                //将 list 项造型为实际 POJO 对象
                interest = (Interest) list.get(i);

                //输出 POJO 内容
                log.info(interest.getId() + "," +
                    getFromEncodingTo(interest.getName(), "ISO8859_1", "GBK"));
            }
        }
    }
}
```

```
    } catch (HibernateException he) {
        //捕捉HibernateException异常
        log.error("Hibernate 异常", he);
    }
}

/**
 * 字符串编码转换
 *
 * @param temp
 * @param fromEncoding
 * @param toEncoding
 * @return
 */
private static String getFromEncodingTo(String temp, String fromEncoding,
    String toEncoding) {
    String rString = "";

    try {
        byte[] rBytes = temp.getBytes(fromEncoding);
        rString = new String(rBytes, toEncoding);
    } catch (UnsupportedEncodingException ex) {
        log.error("编码转换过程出现过错误", ex);
    }

    return rString;
}
}
```

运行 InterestClient.java 后, 能够获得如下输出结果。

```
2004-11-13 16:56:24 net.sf.hibernate.cfg.Environment <clinit>
信息: Hibernate 2.1.6
2004-11-13 16:56:25 net.sf.hibernate.cfg.Environment <clinit>
信息: hibernate.properties not found
2004-11-13 16:56:25 net.sf.hibernate.cfg.Environment <clinit>
信息: using CGLIB reflection optimizer
2004-11-13 16:56:25 net.sf.hibernate.cfg.Configuration configure
信息: configuring from resource: /hibernate.cfg.xml
2004-11-13 16:56:25 net.sf.hibernate.cfg.Configuration
getConfigurationInputStream
信息: Configuration resource: /hibernate.cfg.xml
2004-11-13 16:56:25 net.sf.hibernate.cfg.Configuration addResource
信息: Mapping resource: Interest.hbm
2004-11-13 16:56:25 net.sf.hibernate.cfg.Binder bindRootClass
信息: Mapping class: com.openv.spring.Interest -> interest
2004-11-13 16:56:25 net.sf.hibernate.cfg.Configuration doConfigure
信息: Configured SessionFactory: null
```

```
2004-11-13 16:56:25 net.sf.hibernate.cfg.Configuration secondPassCompile
信息: processing one-to-many association mappings
2004-11-13 16:56:25 net.sf.hibernate.cfg.Configuration secondPassCompile
信息: processing one-to-one association property references
2004-11-13 16:56:25 net.sf.hibernate.cfg.Configuration secondPassCompile
信息: processing foreign key constraints
2004-11-13 16:56:25 net.sf.hibernate.dialect.Dialect <init>
信息: Using dialect: net.sf.hibernate.dialect.MySQLDialect
2004-11-13 16:56:25 net.sf.hibernate.cfg.SettingsFactory buildSettings
信息: Maximim outer join fetch depth: 2
2004-11-13 16:56:25 net.sf.hibernate.cfg.SettingsFactory buildSettings
信息: Use outer join fetching: true
2004-11-13 16:56:25
net.sf.hibernate.connection.DriverManagerConnectionProvider configure
信息: Using Hibernate built-in connection pool (not for production use!)
2004-11-13 16:56:25
net.sf.hibernate.connection.DriverManagerConnectionProvider configure
信息: Hibernate connection pool size: 20
2004-11-13 16:56:25
net.sf.hibernate.connection.DriverManagerConnectionProvider configure
信息: using driver: org.gjt.mm.mysql.Driver at URL:
jdbc:mysql://localhost:3306/example11
2004-11-13 16:56:25
net.sf.hibernate.connection.DriverManagerConnectionProvider configure
信息: connection properties: {user=root, password=}
2004-11-13 16:56:25
net.sf.hibernate.transaction.TransactionManagerLookupFactory
getTransactionManagerLookup
信息: No TransactionManagerLookup configured (in JTA environment, use of process
level read-write cache is not recommended)
2004-11-13 16:56:25 net.sf.hibernate.cfg.SettingsFactory buildSettings
信息: Use scrollable result sets: true
2004-11-13 16:56:25 net.sf.hibernate.cfg.SettingsFactory buildSettings
信息: Use JDBC3 getGeneratedKeys(): true
2004-11-13 16:56:25 net.sf.hibernate.cfg.SettingsFactory buildSettings
信息: Optimize cache for minimal puts: false
2004-11-13 16:56:25 net.sf.hibernate.cfg.SettingsFactory buildSettings
信息: echoing all SQL to stdout
2004-11-13 16:56:25 net.sf.hibernate.cfg.SettingsFactory buildSettings
信息: Query language substitutions: {}
2004-11-13 16:56:25 net.sf.hibernate.cfg.SettingsFactory buildSettings
信息: cache provider: net.sf.hibernate.cache.EhCacheProvider
2004-11-13 16:56:25 net.sf.hibernate.cfg.Configuration configureCaches
信息: instantiating and configuring caches
2004-11-13 16:56:26 net.sf.hibernate.impl.SessionFactoryImpl <init>
信息: building session factory
```

```
2004-11-13 16:56:26 net.sf.hibernate.impl.SessionFactoryObjectFactory
addInstance
信息: Not binding factory to JNDI, no JNDI name configured
Hibernate: select this.id as id0_, this.name as name0_ from interest this where
1=1 order by this.name asc
2004-11-13 16:56:26 com.openv.spring.InterestClient main
信息: 10
2004-11-13 16:56:26 com.openv.spring.InterestClient main
信息: faa75d3d-2b15-11d9-bf84-7b4f7823656e, 旅游
2004-11-13 16:56:26 com.openv.spring.InterestClient main
信息: faa7563d-2b15-11d9-bf84-7b4f7823656e, 篮球
2004-11-13 16:56:26 com.openv.spring.InterestClient main
信息: faa7573d-2b15-11d9-bf84-7b4f7823656e, 网球
2004-11-13 16:56:26 com.openv.spring.InterestClient main
信息: faa7543d-2b15-11d9-bf84-7b4f7823656e, 羽毛球
2004-11-13 16:56:26 com.openv.spring.InterestClient main
信息: faa7583d-2b15-11d9-bf84-7b4f7823656e, 游泳
2004-11-13 16:56:26 com.openv.spring.InterestClient main
信息: faa7523d-2b15-11d9-bf84-7b4f7823656e, 乒乓球
2004-11-13 16:56:26 com.openv.spring.InterestClient main
信息: faa7593d-2b15-11d9-bf84-7b4f7823656e, 高尔夫球
2004-11-13 16:56:26 com.openv.spring.InterestClient main
信息: faa7553d-2b15-11d9-bf84-7b4f7823656e, 滑冰
2004-11-13 16:56:26 com.openv.spring.InterestClient main
信息: faa7533d-2b15-11d9-bf84-7b4f7823656e, 看书
2004-11-13 16:56:26 com.openv.spring.InterestClient main
信息: faa7513d-2b15-11d9-bf84-7b4f7823656e, 足球
```

其中，输出了 Hibernate 对 RDBMS 调用的 SQL 语句，这对于调试 DB 的性能很有帮助。

11.4.2 Hibernate 集成支持

通过上述内容，开发者可以发现，基于 Hibernate 开发 Java 应用的过程很简单。开发者不用手工开发 Java 代码，即可完成同 RDBMS 的交互。但是，直接借助于 Hibernate API 开发应用也暴露了一些问题。

- 尽管借助于 Hibernate Synchronizer 工具能够生成 DAO 代码，但如果 RDBMS 表数量很多，维护这些 DAO 代码很费时间。
- 如果应用系统需要将 O/R Mapping 技术替换掉，则在开发应用系统时，开发者还需要引入另外一层，以处理这种“适配器”层。这不利于代码的维护，而且很难在不同项目中复用。
- 如果需要在 Web/J2EE 应用中使用 Hibernate，则还需要开发者开发集成层。

针对上述问题，Spring 引入了 Hibernate 集成。本章将结合 example11 深入分析 Spring 对 Hibernate 的集成支持。

1. 定义 Hibernate 资源

对于 Web/J2EE 应用而言,开发者可以通过在 Spring 配置文件中定义 sessionFactory 时,给出 Hibernate 映射文件的定义。比如, example11 中的 sessionFactory 定义如下。

//通过 JNDI 获得对 RDBMS 数据源的引用

```
<bean id="dataSource"
      class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
    <value>java:/MySqlDS</value>
  </property>
</bean>
```

//借助于 LocalSessionFactoryBean 定义 sessionFactory

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate.LocalSessionFactoryBean">

  //引用上述定义的 RDBMS 数据源, 即 dataSource

  <property name="dataSource">
    <ref local="dataSource"/>
  </property>

  //定义 Hibernate 映射文件 (资源)

  <property name="mappingResources">
    <list>
      <value>
        com/opensv/spring/service/hibernate/Interest.hbm.xml
      </value>
      <value>
        com/opensv/spring/service/hibernate/UserInterest.hbm.xml
      </value>
      <value>
        com/opensv/spring/service/hibernate/UserInfo.hbm.xml
      </value>
    </list>
  </property>

  //定义 Hibernate 配置属性

  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">
        net.sf.hibernate.dialect.MySQLDialect
      </prop>
```

```
<prop key="hibernate.show_sql">
    true
</prop>
</props>
</property>
</bean>
```

因此,使用 Spring DAO 抽象提供的 Hibernate 集成使得应用代码不用硬编码对资源进行查找,比如对 JDBC DataSource、Hibernate SessionFactory 的查找和获得。对于那些需要访问资源的应用对象而言,借助于 JavaBean 引用便能够实现对资源的使用,比如上述对 dataSource 资源的引用。

如果开发者需要动态更换 DataSource,则只需要修改 Spring 配置文件。比如,如果需要使用 Jakarta Commons 提供的 DBCP BasicDataSource,则可以使用如下类似的配置。

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <property name="driverClassName">
        <value>org.gjt.mm.mysql.Driver</value>
    </property>
    <property name="url">
        <value>jdbc:mysql://localhost:3306/example11</value>
    </property>
    <property name="username">
        <value>root</value>
    </property>
    <property name="password">
        <value></value>
    </property>
</bean>
```

另外,由于 JBoss 3.2.x 和 JBoss 4.0.x 都集成对 Hibernate 的支持,借助于 JNDI 能够实现 SessionFactory 的引用。

因此,但从 Spring 中对 Hibernate 资源的使用方式,就可以看出其实用性。基于 Spring 使用 Hibernate 是很灵活的。

2. HibernateTemplate、HibernateDaoSupport

同 JTA、JMS、JDBC 一样, Spring DAO Hibernate 抽象也提供了 HibernateTemplate。HibernateTemplate 充分利用了 Spring IoC 特性,从而实现对 Hibernate 资源的依赖注入。如果应用只是用 Spring IoC,则只需要在 Spring 配置文件中为它提供 sessionFactory。但是, Spring 框架还提供了另外一个实用类,即 HibernateDaoSupport。它们的关系如图 11-10 所示。

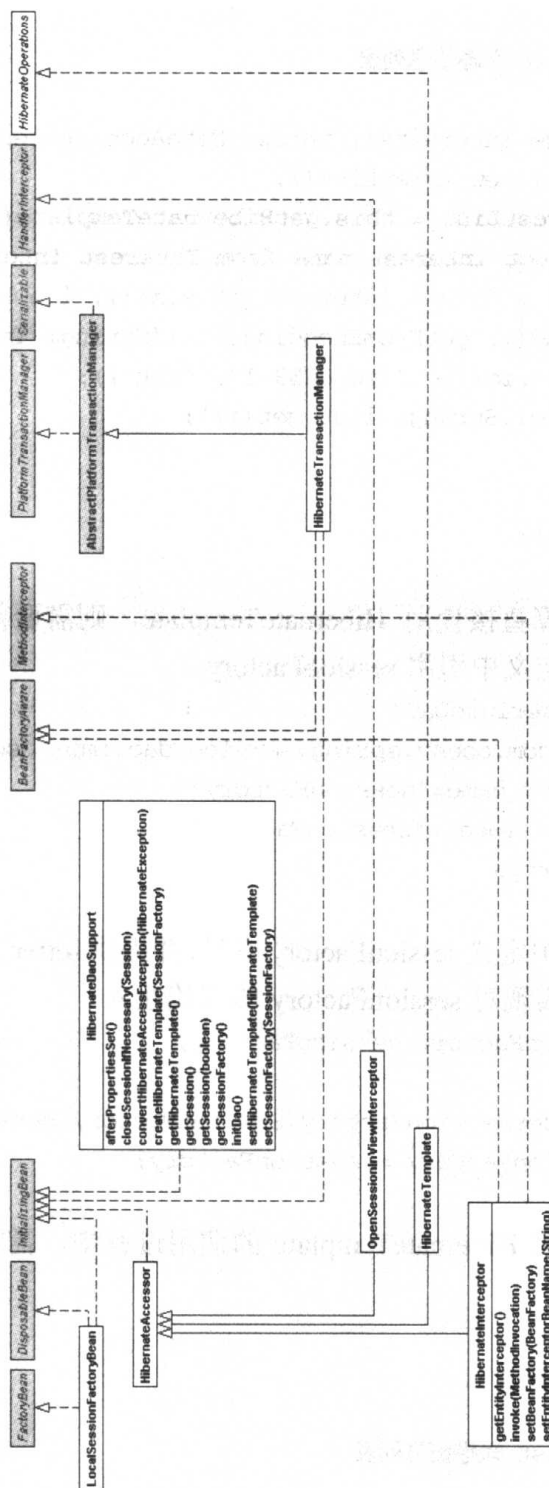


图 11-10 HibernateTemplate、HibernateDaoSupport 等类的 UML 图

比如，在 example11 中的 UserInfoDAO.java 类继承于 HibernateDaoSupport。开发者可以直接在 UserInfoDAO 中借助于 getHibernateTemplate() 以获得对 Hibernate 资源的操作。示例代码如下（摘自 UserInfoDAO.java）。

```
/**
```

```
* 获得兴趣信息列表
```

```

*
* @return List 兴趣信息列表
*/
public List getInterests() throws DataAccessException {
    List list = new ArrayList();
    List interestList = this.getHibernateTemplate().find(
        "select interest.name from Interest interest");
    for (int i = 0, k = interestList.size(); i < k; ++i) {
        list.add(i, getFromEncodingTo((((String) interestList.get(i))
            .trim()), "iso-8859-1", "gbk"));
        log.info((String) list.get(i));
    }

    return list;
}

```

当然，如果开发者打算直接使用 **HibernateTemplate**，则需要作如下几方面的工作。首先，需要在 **userInfoDAO** 定义中引用 **sessionFactory**。

```

<bean id="userInfoDAO"
    class="com.openv.spring.service.dao.impl.UserInfoDAO">
    <property name="sessionFactory">
        <ref local="sessionFactory"/>
    </property>
</bean>

```

需要在 **UserInfoDAO** 中定义 **sessionFactory** 变量，并提供 **setter** 方法，即 **setSessionFactory** 方法，供 **Spring IoC** 容器实现对 **sessionFactory** 的依赖注入。

```

private SessionFactory sessionFactory;

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

```

最后，开发者在实现对 **HibernateTemplate** 的调用过程中，需要实现 **HibernateCallback** 回调接口。示例代码如下。

```

/**
* 获得兴趣信息列表
*
* @return List 兴趣信息列表
*/
public List getInterests() throws DataAccessException {
    List list = new ArrayList();
    HibernateTemplate hibernateTemplate =
        new HibernateTemplate(this.sessionFactory);
    List interestList =
        (List) hibernateTemplate.execute(
            new HibernateCallback() {
                public Object doInHibernate(Session session) throws

```

```

        HibernateException {
            List result = session.find(
                "select interest.name from Interest interest");
            return result;
        }
    }
);

.....

return list;
}

```

开发者可看出，这种回调实现能够实现基于 Hibernate 的 DAO 访问。与此同时，HibernateTemplate 能够保证正确地打开和关闭 Hibernate Session，并自动参与到事务中。同其他模板一样，HibernateTemplate 是线程安全的（通过可重入实现），并且可以重用。对于简单的 Hibernate 操作，比如单个 find、saveOrUpdate 操作，直接使用 HibernateTemplate 比较有效。

3. HibernateInterceptor

另外，Spring 框架也为开发者提供了使用 Hibernate 的另一种方式，即不需要使用 HibernateTemplate，而使用 HibernateInterceptor 拦截器。具体操作方式是，直接将 Hibernate API 代码嵌入在 try/catch 块中，并且在 Spring 配置文件中配置好拦截器。示例 Spring 配置内容如下。

```

<beans>

...

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

<bean id="userinfoDAO"
    class="com.openv.spring.service.dao.impl.UserInfoDAO">
    <property name="sessionFactory">
        <ref local="sessionFactory"/>
    </property>
</bean>

<bean id="myProductDao" class="org.springframework.aop.
    framework.ProxyFactoryBean">
    <property name="proxyInterfaces">

```

```

        <value>com.openv.spring.service.dao.IUserInfoDAO</value>
    </property>
    <property name="interceptorNames">
        <list>
            <value>hibernateInterceptor</value>
            <value>userinfoDAO</value>
        </list>
    </property>
</bean>

```

...

</beans>

具体示例 Java 代码如下。

```

public List getInterests() throws DataAccessException {
    List list = new ArrayList();
    Session session =
        SessionFactoryUtils.getSession(getSessionFactory(), false);
    List interestList = null;
    try{
        interestList = session.find(
            "select interest.name from Interest interest");
        if (interestList == null) {
            throw new ExampleException("未找到兴趣列表");
        }
    } catch(HibernateException he) {
        throw SessionFactoryUtils.convertHibernateAccessException(he);
    }
    for (int i = 0, k = interestList.size(); i < k; ++i) {
        list.add(i, getFromEncodingTo((((String) interestList.get(i))
            .trim()), "iso-8859-1", "gbk"));
        log.info((String) list.get(i));
    }

    return list;
}

```

当然，HibernateInterceptor 会在每次调用 getInterests 之前，准备好线程安全的 session。在每次调用 getInterests 之后，将 session 关闭掉。请注意，务必使用 SessionFactoryUtils 获得 session，因为 HibernateInterceptor、HibernateTemplate 内部使用了 SessionFactoryUtils。同使用 HibernateTemplate 相比，使用 HibernateInterceptor 允许开发者捕捉任何受查异常，而基于 HibernateTemplate 的应用代码却不能够，因为它只能够捕捉回调接口中的未受查异常。但是，在使用便利方面，HibernateTemplate 更方便。

4. Hibernate 事务管理

事务管理的具体内容，本书在前面详细讨论过。这里只是针对 Hibernate 的声明式事务

再次做一下分析。

在实例 example11 中,借助于 TransactionProxyFactoryBean 能够实现声明式事务的使用,这对于 Hibernate 而言也适用。其 Spring 配置代码如下。

```
<bean id="transactionManager"

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

<bean id="example11ServiceTarget"
class="com.openv.spring.service.impl.Example11ManagerImpl">
    <property name="userinfo">
        <ref local="userinfoDAO"/>
    </property>
</bean>

<bean id="example11Service"
class="org.springframework.transaction.interceptor.
    TransactionProxyFactoryBean">
    <property name="transactionManager">
        <ref local="transactionManager"/>
    </property>
    <property name="target">
        <ref local="example11ServiceTarget"/>
    </property>
    <property name="transactionAttributes">
        <props>
            <prop key="get *">
                PROPAGATION_REQUIRED,readOnly
            </prop>
            <prop key="set *">
                PROPAGATION_REQUIRED
            </prop>
        </props>
    </property>
</bean>

<bean id="userinfoDAO"
class="com.openv.spring.service.dao.impl.UserInfoDAO">
    <property name="sessionFactory">
        <ref local="sessionFactory"/>
    </property>
</bean>
```

其中, TransactionProxyFactoryBean 是 Spring 框架提供的便利类。当然,使用它的前提

是在 Spring 配置中没有涉及到其他 Spring AOP 拦截器。它实现了 `TransactionInterceptor` 的功能。而且，在配置上没有 `TransactionInterceptor` 复杂。本书推荐，在一些场合（比如，业务较复杂时），直接使用 `TransactionProxyFactoryBean`，以实现声明式事务，这对于 Hibernate 和非 Hibernate 应用都适用。

11.5 小 结

本章详细地对 Spring 框架提供的 DAO 抽象进行了阐述。其中，结合大量的研究实例，对 Spring JDBC 和 Spring Hibernate 进行了重点研究。Spring DAO 抽象在整个 Spring 框架中的位置举足轻重。开发者可以试想，在解决企业级应用的数据存储中，如果不借助于 RDBMS 持久化业务数据，那该如何架构系统。开发者也已经看到，Spring 提供的 DAO 抽象对于解决持久化服务，确实做了大量的优秀工作。

Spring 会将大量的、其他优秀的 O/R Mapping 技术集成进来，从而为开发者提供很实用的功能。因此，这能够使开发者，尤其是对于开发企业级 Java/J2EE 应用的那些开发者，从繁重的开发任务中摆脱出来。

第 12 章 任务调度服务

——Quartz、Timer

在实际开发应用过程中，经常需要定时，或者重复执行一定的工作。比如，为执行每日构建；定期生成企业报表，包括周、月、季度等；零售领域中门店的日结、月结等。

自从 Java 2 SDK 1.3 以来，Java 2（包括 Swing 提供的定时支持）就内置了用于任务调度的定时器。在 JBoss 3.2.x 和 JBoss 4.0.x 中一直提供了任务调度的支持。Open Source 领域的 Quartz Scheduler¹就是能够提供任务调度服务的技术。JMX 规范中也提供了任务调度支持。在 EJB 2.1 中就加强了任务调度的支持。因此，任务调度服务在企业级应用中越来越重要了。

Spring 框架对 Java 2 提供的 Timer 和 Quartz Scheduler 提供了集成支持。

12.1 背 景

Java 2 借助于 java.util.Timer 和 java.util.TimerTask，这样两个类能够为应用提供简单定时器功能。当然，Swing 也提供了相应的定时器支持。对于那些熟悉 JMX 的开发者而言，也存在 javax.management.timer.TimerMBean 代理服务。通过 TimerMBean，开发者能够实现定时 JMX 通知。

Quartz 是 Open Source 社区努力的结果，它能够提供任务调度服务。任何 J2SE 和 J2EE 应用都能够使用它。借助于 Quartz，开发者能够完成各种复杂的任务调度。

Spring 集成了上述两种任务调度服务的支持。

12.2 Spring 对 Quartz 提供的支持

Quartz 为任务调度提供了大量的功能，比如在何时执行何种任务。Quartz 是 OpenSymphony 的 Open Source 项目，供开发任务调度应用使用。开发者能够在 J2EE 或单独的 J2SE 应用中使用它。无论是简单的任务调度，还是复杂的企业级应用，Quartz 都能够很好地胜任。其中，这些任务可以是标准的 JavaBean 组件，甚至还可以是 EJB 组件。

如果开发者需要开发如下方面的应用，则 Quartz 是您理想的选择。

- 驱动 workflow：比如，如果新创建的流程任务需要在 2 小时内处理完，则在 2 小时后 Quartz 会检查订单是否成功处理。如果没有处理，则 Quartz 会依据 workflow 定义

¹ <http://www.quartzscheduler.org>

的规则来对订单处理。销毁它，或者进行其他处理。

- 系统维护工作：比如，在每个工作日的固定时间将 RDBMS 中的内容导出为 XML 文件。

Spring 对 Quartz 提供了一流的集成支持。通常，开发者需要在应用中使用到 Spring 提供的如下若干类。

- **QuartzJobBean**: Quartz 中 Job 接口的简单实现（子类），Spring 为简化 Job 接口的实现而提供了 QuartzJobBean 类。QuartzJobBean 同 Java 2 SDK 中的 TimerTask 类似，用于定义任务本身。它实现了 org.quartz.Job 接口。其中，executeInternal() 方法定义待执行的任务，这同 TimerTask 中的 run() 类似。
- **JobDetailBean**: Quartz 中 JobDetail 的子类，Spring 为简化 JobDetail 子类的开发而提供了 JobDetailBean。JobDetailBean 是 org.quartz.JobDetail 的子类。借助于 JobDetailBean 中的 jobClass 属性能够设置 Job 对象类型。
- **SimpleTriggerBean**: Quartz 中 SimpleTrigger 类的子类，Spring 为简化 SimpleTrigger 子类的开发而提供了 SimpleTriggerBean。为安排任务，开发者需要设定运行任务的频率和时机。为此，Quartz 提供了 org.quartz.Trigger 类。其中，SimpleTriggerBean 同 Timer 中的 ScheduledTimerTask 类似。
- **CronTriggerBean**: Quartz 中 CronTrigger 类的子类，Spring 为简化 CronTrigger 子类的开发而提供了 CronTriggerBean。CronTriggerBean 比 SimpleTriggerBean 功能更强大。它能够控制任务执行的精确时间，比如早上 9 点 30 需要执行某 QuartzJobBean 中的给定的任务。借助于 CronTriggerBean 中的 cronExpression 属性能够设定任务的执行时机。
- **MethodInvokingJobDetailFactoryBean**: FactoryBean，将 JobDetail 对象暴露出来。开发者不用开发单独的 QuartzJobBean 类即可完成任务的调度，这就是该工厂 JavaBean 的神奇之处。Spring 为 Quartz 提供的 MethodInvokingJobDetailFactoryBean 同 Timer 中的 MethodInvokingTimerFactoryBean 类似。
- **SchedulerFactoryBean**: FactoryBean，用于设置 Quartz Scheduler，供暴露给应用使用。SchedulerFactoryBean 同 Timer 中的 TimerFactoryBean 等效。借助于它能够启动定时器。

它们（位于 org.springframework.scheduling.quartz 包中）之间的关系见图 12-1。

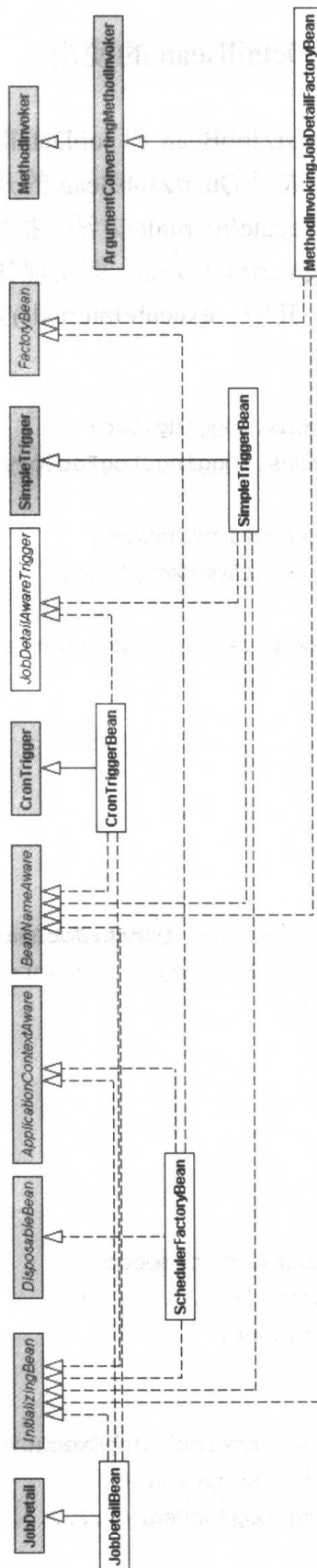


图 12-1 quartz 包中个各类之间的关系

12.2.1 QuartzJobBean 和 JobDetailBean 的使用

本章将结合 example21 对 QuartzJobBean 和 JobDetailBean 进行阐述。首先，开发者需要实现 LogJobBean.java 类，以完成对 QuartzJobBean 的开发。开发者是否注意到，其中定义了 timeout 属性，并且实现了 executeInternal() 方法。抽象类 QuartzJobBean 用于定义待完成的任务本身。开发者通过继承 QuartzJobBean，即实现其中的 executeInternal() 方法，即可完成 QuartzJobBean 实例的开发。其中，executeInternal() 方法用于定义待完成的任务。

```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;

import org.springframework.scheduling.quartz.QuartzJobBean;

/**
 * LogJobBean
 *
 * @author luoshifei
 */
public class LogJobBean extends QuartzJobBean {
    protected static final Log log = LogFactory.getLog(LogJobBean.class);
    private int timeout;

    /**
     * 设置超时时间
     *
     * @param timeout
     */
    public void setTimeout(int timeout) {
        log.info("setTimeout().....");
        this.timeout = timeout;
    }

    protected void executeInternal(JobExecutionContext arg0)
        throws JobExecutionException {
        log.info("Creating LogJobBean().....");
    }
}
```

其次，开发者需要提供 appcontext.xml Spring 配置文件。注意，Spring 框架提供了 SimpleTriggerBean 来定义 JobDetailBean 的运行频率和初始运行时机。通过 jobDetail 属性

指定待执行的任务（比如，logJob）；通过 repeatInterval 属性指定任务执行的频率（比如，2 秒表示每两秒执行一次 executeInternal() 方法）；通过 startDelay 属性指定在初次执行任务前必须等待的时间；通过 repeatCount 属性指定任务执行的次数。注意，startDelay 属性仅仅是给出了初次任务执行的相对时间，而不是绝对时间，比如上午 9 点 30。因此，如果客户需要精确定义到某时某刻，则需要使用 Quartz 中的 CronTriggerBean。

为启动 SimpleTriggerBean，开发者还需配置 SchedulerFactoryBean，即启动 Scheduler 器。SchedulerFactoryBean 能够担当此任。通过 triggers 属性能够指定所有的计划，即 SimpleTriggerBean。比如将 simpleTrigger 加入到其中。

另外，开发者是否注意到 logJob JavaBean 的配置。其中，借助于 jobClass 属性指定任务本身，比如 com.openv.spring.LogJobBean。借助于 jobDataAsMap 属性指定 LogJobBean 中定义的属性（这些属性必须提供 setter 方法）。

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

    <bean name="logJob"
        class="org.springframework.scheduling.quartz.JobDetailBean">
        <property name="jobClass">
            <value>com.openv.spring.LogJobBean</value>
        </property>
        <property name="jobDataAsMap">
            <map>
                <entry key="timeout">
                    <value>10</value>
                </entry>
            </map>
        </property>
    </bean>

    <bean id="simpleTrigger"
        class="org.springframework.scheduling.quartz.SimpleTriggerBean">
        <property name="jobDetail">

            <!-- 引用上述 LogJobBean -->

            <ref bean="logJob"/>
        </property>
        <property name="startDelay">

            <!-- 第一次执行任务前，需要等待 5 秒钟 -->

            <value>5000</value>
        </property>
    </bean>
</beans>
```

```
<property name="repeatInterval">

    <!-- 任务执行周期为 2 秒钟 -->

    <value>2000</value>
</property>
<property name="repeatCount">

    <!-- 执行次数 -->

    <value>3</value>
</property>
</bean>

<bean id="sfb"
      class="org.springframework.scheduling.quartz
            .SchedulerFactoryBean">
  <property name="triggers">
    <list>
      <ref local="simpleTrigger"/>
    </list>
  </property>
</bean>

</beans>
```

通过 Spring IDE 提供的图形工具，开发者能够获得如图 12-2 所示的内容。可以看出，开发者确实只要对 sfb 操作，而其他内容都是通过 Spring IoC 容器完成的。

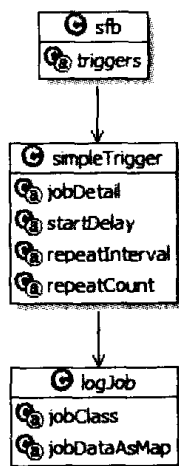


图 12-2 example21 中 applicationContext.xml 的 JavaBean 结构

最后，开发者需要提供客户应用。借助于 BeanFactory 获得 Scheduler 后，安排的任务即可根据设定的运行规则执行。

```
package com.openv.spring;
```

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.quartz.Scheduler;
import org.quartz.SchedulerException;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

/**
 * LogJobBean 客户应用
 *
 * @author luoshifei
 */
public class LogJobBeanTest {
    protected static final Log log = LogFactory.getLog(LogJobBeanTest.class);

    public static void main(String[] args) {
        Resource resource = new ClassPathResource("appcontext.xml");
        BeanFactory factory = new XmlBeanFactory(resource);
        Scheduler sfb = (Scheduler) factory.getBean("sfb");

        try {
            Thread.sleep(20000);
        } catch (InterruptedException e) {
            log.error("InterruptedException", e);
        }

        try {
            sfb.shutdown();
        } catch (SchedulerException se) {
            log.error("SchedulerException", se);
        }
    }
}
```

其中，在应用代码中，开发者只需要使用到 sfb，即 Scheduler。运行 Ant build.xml 中的 run 任务结果如下。

```
Buildfile: D:\workspace\example21\build.xml
compile:
run:
    [java] 2004-12-26 14:27:36 org.springframework.beans.factory.xml.
    XmlBeanDefinitionReader loadBeanDefinitions
```

```
[java] 信息: Loading XML bean definitions from class path resource
[appcontext.xml]
[java] 2004-12-26 14:27:36 org.springframework.core.CollectionFactory
<clinit>
[java] 信息: Using JDK 1.4 collections
[java] 2004-12-26 14:27:36 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'sfb'
[java] 2004-12-26 14:27:36 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'simpleTrigger'
[java] 2004-12-26 14:27:36 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'logJob'
[java] 2004-12-26 14:27:36 org.quartz.simpl.SimpleThreadPool initialize
[java] 信息: Job execution threads will use class loader of thread: main
[java] 2004-12-26 14:27:36 org.quartz.simpl.RAMJobStore initialize
[java] 信息: RAMJobStore initialized.
[java] 2004-12-26 14:27:36 org.quartz.impl.StdSchedulerFactory
instantiate
[java] 信息: Quartz scheduler 'DefaultQuartzScheduler' initialized from
default resource file in Quartz package: 'quartz.properties'
[java] 2004-12-26 14:27:36 org.quartz.impl.StdSchedulerFactory
instantiate
[java] 信息: Quartz scheduler version: 1.4.2
[java] 2004-12-26 14:27:36 org.springframework.scheduling.quartz.
SchedulerFactoryBean startScheduler
[java] 信息: Starting Quartz scheduler now
[java] 2004-12-26 14:27:36 org.quartz.core.QuartzScheduler start
[java] 信息: Scheduler DefaultQuartzScheduler_$_NON_CLUSTERED started.
[java] 2004-12-26 14:27:41 com.openv.spring.LogJobBean setTimeout
[java] 信息: setTimeout().....
[java] 2004-12-26 14:27:41 com.openv.spring.LogJobBean executeInternal
[java] 信息: Creating LogJobBean().....
[java] 2004-12-26 14:27:43 com.openv.spring.LogJobBean setTimeout
[java] 信息: setTimeout().....
[java] 2004-12-26 14:27:43 com.openv.spring.LogJobBean executeInternal
[java] 信息: Creating LogJobBean().....
[java] 2004-12-26 14:27:45 com.openv.spring.LogJobBean setTimeout
[java] 信息: setTimeout().....
[java] 2004-12-26 14:27:45 com.openv.spring.LogJobBean executeInternal
[java] 信息: Creating LogJobBean().....
[java] 2004-12-26 14:27:47 com.openv.spring.LogJobBean setTimeout
[java] 信息: setTimeout().....
[java] 2004-12-26 14:27:47 com.openv.spring.LogJobBean executeInternal
[java] 信息: Creating LogJobBean().....
```

```
[java] 2004-12-26 14:27:56 org.quartz.core.QuartzScheduler shutdown
[java] 信息: Scheduler DefaultQuartzScheduler_$_NON_CLUSTERED shutting
down.
[java] 2004-12-26 14:27:56 org.quartz.core.QuartzScheduler pause
[java] 信息: Scheduler DefaultQuartzScheduler_$_NON_CLUSTERED paused.
[java] 2004-12-26 14:27:56 org.quartz.core.QuartzScheduler shutdown
[java] 信息: Scheduler DefaultQuartzScheduler_$_NON_CLUSTERED shutdown
complete.
BUILD SUCCESSFUL
Total time: 24 seconds
```

可以看出, **LogJobBean** 的第一次执行等待了 5 秒钟, 然后每次执行周期为 2 秒钟。这同预期的一样。

12.2.2 MethodInvokingJobDetailFactoryBean 的使用

在前面的实例中, 开发者已经看出, 为实现任务调度, 开发者实现的 **Scheduler** 器任务类必须继承 **QuartzJobBean**, 而且必须实现其中的 **executeInternal()** 方法。对于一些遗留应用而言, 如果不存在源代码, 或者其他原因, 使得开发基于 **QuartzJobBean** 的 **Scheduler** 器任务不可能。该如何解决呢?

Spring 考虑到这种需求, 提供了 **MethodInvokingJobDetailFactoryBean**。其具体运行机理是 **MethodInvokingJobDetailFactoryBean** 会创建 **JobDetail** 对象, 并调用由 **targetObject** 属性指定的对象的方法, 而该执行方法由 **targetMethod** 属性指定。因此, 最终的执行效果将同直接实现 **QuartzJobBean** 一样, 只不过直接使用 **MethodInvokingJobDetailFactoryBean** 更简单罢了, 而且对于遗留应用而言特别有效。

本章将结合 example21 对 **MethodInvokingJobDetailFactoryBean** 进行阐述。首先, 开发者需要实现 **LogJobMethodBean.java** 类。注意, **LogJobMethodBean** 并没有继承于 **QuartzJobBean**, 它只是普通的 **JavaBean** 类。在随后的配置中, 开发者能够看到其定义的 **log()** 方法将会被周期地执行。

```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * LogJobMethodBean
 *
 * @author luoshifei
 */
public class LogJobMethodBean {
    protected static final Log log = LogFactory.getLog(LogJobMethodBean.class);

    public void log() {
```

```
log.info("Creating LogJobMethodBean().....");
```

```
}
```

```
}
```

其次，提供 appcontext.xml Spring 配置文件。其中，开发者能够清楚地看到，借助于 MethodInvokingJobDetailFactoryBean 将 LogJobMethodBean 及其中的 log() 方法分别指定为 targetObject 和 targetMethod 属性的取值。然后，SimpleTriggerBean 的 jobDetail 需要引用它，即 miJobDetail。

```
<?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="logJobMethodBean"
        class="com.openv.spring.LogJobMethodBean">
    </bean>

    <bean id="miJobDetail"
        class="org.springframework.scheduling.quartz
            .MethodInvokingJobDetailFactoryBean">
        <property name="targetObject">
            <ref bean="logJobMethodBean"/>
        </property>
        <property name="targetMethod">
            <value>log</value>
        </property>
    </bean>

    <bean id="simpleTrigger"
        class="org.springframework.scheduling.quartz
            .SimpleTriggerBean">
        <property name="jobDetail">

            <!-- 引用上述 LogJobBean -->

            <ref bean="miJobDetail"/>
        </property>
        <property name="startDelay">

            <!-- 第一次执行任务前，需要等待 5 秒钟 -->

            <value>5000</value>
        </property>
        <property name="repeatInterval">

            <!-- 任务执行周期为 2 秒钟 -->
```



```

        <value>2000</value>
    </property>
    <property name="repeatCount">

        <!-- 执行次数 -->

        <value>3</value>
    </property>
</bean>

<bean id="sfb"

class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <property name="triggers">
        <list>
            <ref local="simpleTrigger"/>
        </list>
    </property>
</bean>

</beans>

```

通过 Spring IDE 提供的图形工具，开发者能够获得如图 12-3 所示的内容。可以看出，开发者确实也只要对 sfb 操作，而其他内容都是通过 Spring IoC 容器完成的。但是，开发者应该注意到，这比图 12-2 多引入了一层，即 miJobDetail。

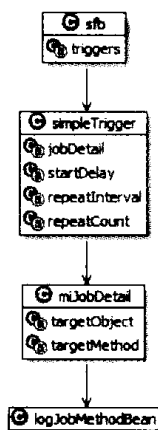


图 12-3 example21 中 appcontextmethod.xml 的 JavaBean 结构

最后，开发者需要提供客户应用。

```

package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.quartz.Scheduler;
import org.quartz.SchedulerException;

```

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

/**
 * LogJobMethodBean 客户应用
 *
 * @author luoshifei
 */
public class LogJobMethodBeanTest {
    protected static final Log log =
LogFactory.getLog(LogJobMethodBeanTest.class);

    public static void main(String[] args) {
        Resource resource = new ClassPathResource("appcontextmethod.xml");
        BeanFactory factory = new XmlBeanFactory(resource);
        Scheduler sfb = (Scheduler) factory.getBean("sfb");

        try {
            Thread.sleep(20000);
        } catch (InterruptedException e) {
            log.error("InterruptedException", e);
        }

        try {
            sfb.shutdown();
        } catch (SchedulerException se) {
            log.error("SchedulerException", se);
        }
    }
}
```

可以看出，这同上述实例没有区别。其中，在应用代码中，开发者只需要使用到 sfb，即 Scheduler。运行 Ant build.xml 中的 runmethod 任务结果如下。

```
Buildfile: D:\workspace\example21\build.xml
compile:
runmethod:
[java] 2004-12-26 14:33:15 org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
[java] 信息: Loading XML bean definitions from class path resource
[appcontextmethod.xml]
[java] 2004-12-26 14:33:16 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
```

```
[java] 信息: Creating shared instance of singleton bean 'sfb'
[java] 2004-12-26 14:33:16 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'simpleTrigger'
[java] 2004-12-26 14:33:16 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'miJobDetail'
[java] 2004-12-26 14:33:16 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'logJobMethodBean'
[java] 2004-12-26 14:33:16 org.quartz.simpl.SimpleThreadPool initialize
[java] 信息: Job execution threads will use class loader of thread: main
[java] 2004-12-26 14:33:16 org.quartz.simpl.RAMJobStore initialize
[java] 信息: RAMJobStore initialized.
[java] 2004-12-26 14:33:16 org.quartz.impl.StdSchedulerFactory
instantiate
[java] 信息: Quartz scheduler 'DefaultQuartzScheduler' initialized from
default resource file in Quartz package: 'quartz.properties'
[java] 2004-12-26 14:33:16 org.quartz.impl.StdSchedulerFactory
instantiate
[java] 信息: Quartz scheduler version: 1.4.2
[java] 2004-12-26 14:33:16 org.springframework.scheduling.quartz.
SchedulerFactoryBean startScheduler
[java] 信息: Starting Quartz scheduler now
[java] 2004-12-26 14:33:16 org.quartz.core.QuartzScheduler start
[java] 信息: Scheduler DefaultQuartzScheduler_$_NON_CLUSTERED started.
[java] 2004-12-26 14:33:21 com.openv.spring.LogJobMethodBean log
[java] 信息: Creating LogJobMethodBean().....
[java] 2004-12-26 14:33:23 com.openv.spring.LogJobMethodBean log
[java] 信息: Creating LogJobMethodBean().....
[java] 2004-12-26 14:33:25 com.openv.spring.LogJobMethodBean log
[java] 信息: Creating LogJobMethodBean().....
[java] 2004-12-26 14:33:27 com.openv.spring.LogJobMethodBean log
[java] 信息: Creating LogJobMethodBean().....
[java] 2004-12-26 14:33:36 org.quartz.core.QuartzScheduler shutdown
[java] 信息: Scheduler DefaultQuartzScheduler_$_NON_CLUSTERED shutting
down.
[java] 2004-12-26 14:33:36 org.quartz.core.QuartzScheduler pause
[java] 信息: Scheduler DefaultQuartzScheduler_$_NON_CLUSTERED paused.
[java] 2004-12-26 14:33:36 org.quartz.core.QuartzScheduler shutdown
[java] 信息: Scheduler DefaultQuartzScheduler_$_NON_CLUSTERED shutdown
complete.
BUILD SUCCESSFUL
Total time: 22 seconds
```

可以看出, **LogJobMethodBean** 的第一次执行等待了 5 秒钟, 然后每次执行周期为 2 秒钟, 同预期的一样。这同上述实例类似。

另外，如果开发者需要使用 `org.springframework.scheduling.quartz.CronTriggerBean` 类，则需要弄清楚 `cronExpression` 属性的含义。由于 `CronTriggerBean` 能够精确控制任务的具体执行时间，因此在实际应用中广为使用。如下给出了配置片断。

```
<bean id="cronTrigger"
      class="org.springframework.scheduling.quartz.CronTriggerBean">
  <property name="jobDetail">
    <ref bean="miJobDetail"/>
  </property>
  <property name="cronExpression">
    <value>0 23 8 * * ?</value>
  </property>
</bean>
```

通常，`cronExpression` 表达式为 6 或者 7 位。各个位之间通过空格隔开。对于每一位而言，可以通过某值直接给出，比如 8；也可以使用列表，比如“6,12,18”；也可以使用范围，比如 9-15；还可以使用通配符，比如*；对于存在互斥的情形，需要使用“?”表示。`cronExpression` 表达式含义的具体解释如表 12-1 所示。

表 12-1 `cronExpression` 的含义

第几位	从左到右	含 义	取值范围
1		秒	0-59
2		分钟	0-59
3		小时	0-23
4		日	1-31
5		月	1-12
6		星期几	1-7
7		年	1970-2099

至于 `cronExpression` 属性的更详细内容，请开发者参考 Quartz 网站。

12.3 Spring 对 Timer 提供的支持

自从 Java 2 SDK 1.3 开始，借助于 `java.util.Timer` 类能够为企业应用提供任务调度支持。尽管现有的 `Timer` 没有 Quartz 功能丰富，但是能够满足简单的企业应用需求。而且，开发者也不用另外学习 Quartz 这样一套全新的类库，因为 Java 2 本身就内置了 `Timer` 的支持。因此，Spring 对 `java.util.Timer` 还是提供了一流的支持。

Spring 对 `Timer` 提供了如下几个类的支持。

- **ScheduledTimerTask**: 对定时任务的描述，由 `TimerTask` 以及其他属性构成。
- **MethodInvokingTimerTaskFactoryBean**: `FactoryBean`，将某方法暴露为 `TimerTask` 对象。这是实现 `TimerTask` 的另一种方式。
- **TimerFactoryBean**: `FactoryBean`，设置 Java 2 定时器，并供应用使用。
- Spring 提供了 `org.springframework.scheduling.timer` 包，以支持 `Timer`，见图 12-4。

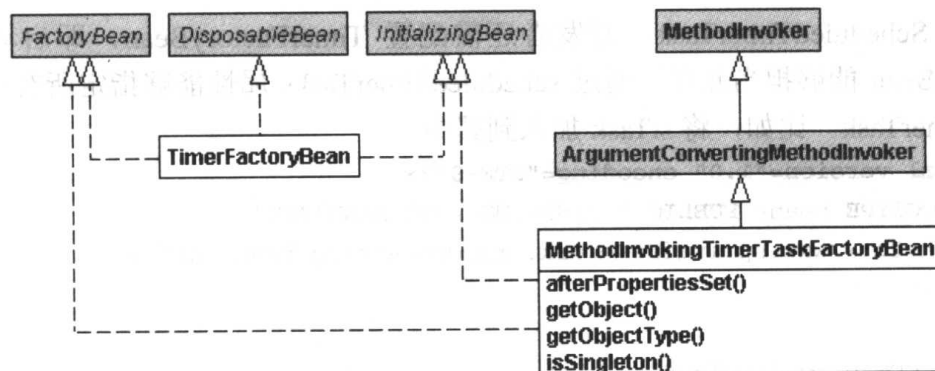


图 12-4 timer 包的类图

12.3.1 ScheduledTimerTask 的使用

本章将结合 example22 对 ScheduledTimerTask 进行阐述。首先，开发者需要实现 LogTask.java 类，以完成对 TimerTask 的开发。抽象类 TimerTask 用于定义待完成的任务本身。开发者通过继承 TimerTask，即实现其中的 run() 方法，即可完成 TimerTask 实例的开发。其中，run() 方法用于定义待完成的任务。

```

package com.openv.spring;
import java.util.TimerTask;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * 定时任务
 *
 * @author luoshifei
 */
public class LogTask extends TimerTask {
    protected static final Log log = LogFactory.getLog(LogTask.class);

    public void run() {
        log.info("Creating LogTask().....");
    }
}

```

其次，开发者需要提供 applicationContext.xml Spring 配置文件。注意，Spring 框架提供了 ScheduledTimerTask 来定义 TimerTask 的运行频率和初始运行时机。通过 timerTask 属性指定待执行的任务（比如 logTask）；通过 period 属性指定任务执行的频率（比如 2 秒表示每两秒执行一次 run() 方法）；通过 delay 属性指定在初次执行任务前必须等待的时间。注意，delay 属性仅仅是给出了初次任务执行的相对时间，而不是绝对时间，比如上午 9 点半。因此，如果客户需要精确定义到某时某刻，则需要使用 Quartz。

为启动 `ScheduledTimerTask`，开发者还需配置 `TimerFactoryBean`，即启动定时器。`TimerFactoryBean` 能够担当此任。通过 `scheduledTimerTasks` 属性能够指定所有的计划，即 `ScheduledTimerTask`。比如，将 `stTask` 加入到其中。

```
<?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="logTask"
        class="com.openv.spring.LogTask">
    </bean>

    <bean id="stTask"
        class="org.springframework.scheduling.timer.ScheduledTimerTask">

        <!-- 首次执行任务前，需要等待 5 秒 -->

        <property name="delay">
            <value>5000</value>
        </property>

        <!-- 任务执行的周期为 2 秒 -->

        <property name="period">
            <value>2000</value>
        </property>

        <!-- 具体执行的任务 -->

        <property name="timerTask">
            <ref local="logTask"/>
        </property>
    </bean>

    <bean id="timerFactory"
        class="org.springframework.scheduling.timer.TimerFactoryBean">
        <property name="scheduledTimerTasks">
            <list>
                <ref local="stTask"/>
            </list>
        </property>
    </bean>

</beans>
```

通过 Spring IDE 提供的图形工具，开发者能够获得如图 12-5 所示的内容。可以看出，

开发者只需要对 timerFactory 操作，而其他内容都是通过 Spring IoC 容器完成的。

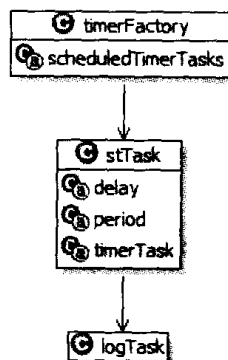


图 12-5 example22 中 appcontext.xml 的 JavaBean 结构

最后，开发者需要提供客户应用。借助于 BeanFactory 获得 Timer 后，安排的任务即可根据设定的运行规则执行。

```

package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

import java.util.Timer;

/**
 * LogTask 客户应用
 *
 * @author luoshifei
 */
public class LogTaskTest {
    protected static final Log log = LogFactory.getLog(LogTaskTest.class);

    public static void main(String[] args) {
        Resource resource = new ClassPathResource("appcontext.xml");
        BeanFactory factory = new XmlBeanFactory(resource);
        Timer timer = (Timer) factory.getBean("timerFactory");

        try {
            Thread.sleep(20000);
        } catch (InterruptedException e) {
            log.error("InterruptedException", e);
        }
    }
}

```

```
    }  
  
    timer.cancel();  
}  
}
```

其中, 在应用代码中, 开发者只需要使用到 `timeFactory`, 即 `Timer`。运行 Ant `build.xml` 中的 `run` 任务结果如下。

```
Buildfile: D:\workspace\example22\build.xml  
compile:  
run:  
    [java] 2004-12-26 14:44:31 org.springframework.beans.factory.xml.  
XmlBeanDefinitionReader loadBeanDefinitions  
    [java] 信息: Loading XML bean definitions from class path resource  
[appcontext.xml]  
    [java] 2004-12-26 14:44:32 org.springframework.beans.factory.support.  
AbstractBeanFactory getBean  
    [java] 信息: Creating shared instance of singleton bean 'timerFactory'  
    [java] 2004-12-26 14:44:32  
org.springframework.beans.factory.support.AbstractBeanFactory getBean  
    [java] 信息: Creating shared instance of singleton bean 'stTask'  
    [java] 2004-12-26 14:44:32 org.springframework.beans.factory.support.  
AbstractBeanFactory getBean  
    [java] 信息: Creating shared instance of singleton bean 'logTask'  
    [java] 2004-12-26 14:44:32 org.springframework.scheduling.timer.  
TimerFactoryBean afterPropertiesSet  
    [java] 信息: Initializing Timer  
    [java] 2004-12-26 14:44:37 com.openv.spring.LogTask run  
    [java] 信息: Creating LogTask().....  
    [java] 2004-12-26 14:44:39 com.openv.spring.LogTask run  
    [java] 信息: Creating LogTask().....  
    [java] 2004-12-26 14:44:41 com.openv.spring.LogTask run  
    [java] 信息: Creating LogTask().....  
    [java] 2004-12-26 14:44:43 com.openv.spring.LogTask run  
    [java] 信息: Creating LogTask().....  
    [java] 2004-12-26 14:44:45 com.openv.spring.LogTask run  
    [java] 信息: Creating LogTask().....  
    [java] 2004-12-26 14:44:47 com.openv.spring.LogTask run  
    [java] 信息: Creating LogTask().....  
    [java] 2004-12-26 14:44:49 com.openv.spring.LogTask run  
    [java] 信息: Creating LogTask().....  
    [java] 2004-12-26 14:44:51 com.openv.spring.LogTask run  
    [java] 信息: Creating LogTask().....  
BUILD SUCCESSFUL  
Total time: 22 seconds
```

可以看出, `LogTask` 的第一次执行等待了 5 秒钟, 然后每次执行周期为 2 秒钟。这同预期的一样。

12.3.2 MethodInvocationTimerTaskFactoryBean 的使用

在前面的实例中，开发者已经看出，为实现任务调度，开发者实现的定时器任务类必须继承 `TimerTask`，而且必须实现其中的 `run()` 方法。对于一些遗留应用而言，如果不存在源代码，或者其他原因，使得开发基于 `TimerTask` 的定时器任务不可能。该如何呢？

Spring 考虑到这种需求，提供了 `MethodInvokingTimerTaskFactoryBean`。其具体运行机理是，`MethodInvokingTimerTaskFactoryBean` 会创建 `TimerTask`，并调用由 `targetObject` 属性指定的对象的方法，而该执行方法由 `targetMethod` 属性指定。因此，最终的执行效果将同直接实现 `TimerTask` 一样，只不过直接使用 `MethodInvokingTimerTaskFactoryBean` 更简单罢了，而且对于遗留应用而言特别有效。

本章将结合 example23 对 `MethodInvokingTimerTaskFactoryBean` 进行阐述。首先，开发者需要实现 `LogMethodTask.java` 类。注意，`LogMethodTask` 并没有继承于 `TimerTask`，它只是普通的 `JavaBean` 类。在随后的配置中，开发者能够看到其定义的 `log()` 方法将会被周期地执行。

```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * LogMethodTask 类
 *
 * @author luoshifei
 */
public class LogMethodTask {
    protected static final Log log = LogFactory.getLog(LogMethodTask.class);

    public void log(){
        log.info("Creating LogMethodTask().....");
    }
}
```

其次，开发者需要提供 `appcontext.xml` Spring 配置文件。其中，开发者能够清楚地看到，借助于 `MethodInvokingTimerTaskFactoryBean` 将 `LogMethodTask` 及其中的 `log()` 方法分别指定为 `targetObject` 和 `targetMethod` 属性的取值。然后，`ScheduledTimerTask` 的 `timerTask` 需要引用它，即 `mtiTask`。

```
<?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="logMethodTask"
```

```
        class="com.openv.spring.LogMethodTask">
</bean>

<bean id="stTask"
    class="org.springframework.scheduling.timer.ScheduledTimerTask">

    <!-- 首次执行任务前, 需要等待 5 秒 -->

    <property name="delay">
        <value>5000</value>
    </property>

    <!-- 任务执行的周期为 2 秒 -->

    <property name="period">
        <value>2000</value>
    </property>

    <!-- 具体执行的任务 -->

    <property name="timerTask">
        <ref local="mtiTask"/>
    </property>
</bean>

<bean id="mtiTask"
    class="org.springframework.scheduling.timer
        .MethodInvokingTimerTaskFactoryBean">
    <property name="targetObject">
        <ref bean="logMethodTask"/>
    </property>
    <property name="targetMethod">
        <value>log</value>
    </property>
</bean>

<bean id="timerFactory"
    class="org.springframework.scheduling.timer.TimerFactoryBean">
    <property name="scheduledTimerTasks">
        <list>
            <ref local="stTask"/>
        </list>
    </property>
</bean>

</beans>
```

通过 Spring IDE 提供的图形工具，开发者能够获得如图 12-6 所示的内容。可以看出，开发者确实也只要对 timerFactory 操作，而其他内容都是通过 Spring IoC 容器完成的。但是，开发者应该注意到，这比 example22 多引入了一层，即 mtiTask。

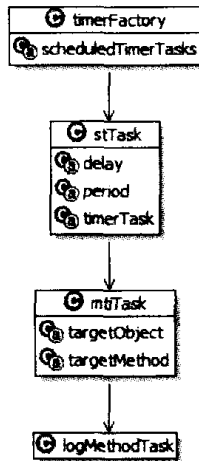


图 12-6 example23 中 applicationContext.xml 的 JavaBean 结构

最后，开发者需要提供客户应用。

```

package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

import java.util.Timer;

/**
 * LogMethodTask 客户应用
 *
 * @author luoshifei
 */
public class LogTaskTest {
    protected static final Log log = LogFactory.getLog(LogTaskTest.class);

    public static void main(String[] args) {
        Resource resource = new ClassPathResource("appcontext.xml");
        BeanFactory factory = new XmlBeanFactory(resource);
        Timer timer = (Timer) factory.getBean("timerFactory");
    }
}

```

```
try {
    Thread.sleep(20000);
} catch (InterruptedException e) {
    log.error("InterruptedException", e);
}

timer.cancel();
}
```

可以看出，这同 **example22** 没有区别。其中，在应用代码中，开发者只需要使用到 **timeFactory**，即 **Timer**。运行 **Ant build.xml** 中的 **run** 任务结果如下。

```
Buildfile: D:\workspace\example23\build.xml
compile:
run:
[java] 2004-12-26 14:49:55
org.springframework.beans.factory.xml.XmlBeanDefinitionReader
loadBeanDefinitions
[java] 信息: Loading XML bean definitions from class path resource
[appcontext.xml]
[java] 2004-12-26 14:49:55
org.springframework.beans.factory.support.AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'timerFactory'
[java] 2004-12-26 14:49:55
org.springframework.beans.factory.support.AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'stTask'
[java] 2004-12-26 14:49:55
org.springframework.beans.factory.support.AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'mtiTask'
[java] 2004-12-26 14:49:55
org.springframework.beans.factory.support.AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'logMethodTask'
[java] 2004-12-26 14:49:55
org.springframework.scheduling.timer.TimerFactoryBean afterPropertiesSet
[java] 信息: Initializing Timer
[java] 2004-12-26 14:50:00 com.openv.spring.LogMethodTask log
[java] 信息: Creating LogMethodTask().....
[java] 2004-12-26 14:50:02 com.openv.spring.LogMethodTask log
[java] 信息: Creating LogMethodTask().....
[java] 2004-12-26 14:50:04 com.openv.spring.LogMethodTask log
[java] 信息: Creating LogMethodTask().....
[java] 2004-12-26 14:50:06 com.openv.spring.LogMethodTask log
[java] 信息: Creating LogMethodTask().....
[java] 2004-12-26 14:50:08 com.openv.spring.LogMethodTask log
[java] 信息: Creating LogMethodTask().....
[java] 2004-12-26 14:50:10 com.openv.spring.LogMethodTask log
[java] 信息: Creating LogMethodTask().....
```

```
[java] 2004-12-26 14:50:12 com.openv.spring.LogMethodTask log
[java] 信息: Creating LogMethodTask().....
[java] 2004-12-26 14:50:14 com.openv.spring.LogMethodTask log
[java] 信息: Creating LogMethodTask().....
BUILD SUCCESSFUL
Total time: 21 seconds
```

可以看出, **LogMethodTask** 的第一次执行等待了 5 秒钟, 然后每次执行周期为 2 秒钟, 同预期的一样。这同 **example22** 类似。

12.4 小 结

本章对 **Spring** 支持的任务调度服务进行了深入研究。在很多企业级应用中, 经常需要使用到定时服务。注意, 借助于 **java.util.Timer**, 开发者不能够精确定义任务的执行时间, 比如上午 9 点半, 它只能约定到定时器启动的相对时间, 即通过 **delay** 属性给出。而 **Quartz** 却可以满足客户这方面的需求。尽管 **java.util.Timer** 提供的功能不如 **Quartz** 丰富, 但是 **Spring** 同样为两者提供了一流的集成支持, 因此 **Spring** 很实用。

第 13 章 远 程 服 务

将应用服务暴露给远程是很多应用开发过程中需要解决的问题。在 Java/J2EE 领域中，提供了 RMI、EJB、Web 服务等手段将服务暴露给远程。所谓远程，指客户同服务器进行的会话过程。比如在借助于 IE 浏览 Web 站点时，IE 是客户，而服务器是 J2EE 应用服务器或者是简单的 Jakarta Apache HTTP 服务器，甚至还有可能是 IIS。当然，客户与服务器通常都运行在不同进程、JVM、物理机器中，其间还可能存在防火墙。

在 Java/J2EE 中，无论是借助于 RMI，还是 EJB，还是 Web 服务，开发者都需要完成许多与业务逻辑无关的代码，比如借助于 RMI Naming 注册服务对象、借助于 JNDI 查找 EJB Home 对象、借助于标准 JAX-RPC 方式需要指定许多参数类型（WSDL 文档的 URL、QName、命名空间 URI、服务工厂）等。这些代码对于业务问题而言，毫无益处。

Spring 为解决远程支持，借助于各种技术提供了有效的解决方案，并且能够消除开发者编写与业务逻辑无关代码的必要性。最重要的一点是，开发者能够针对自身的业务需求，合理选用 Spring 提供的远程服务。因为使用 Spring 提供的远程服务，对于开发者而言，仅仅涉及到配置问题，而不用将具体的远程服务类型硬编码在代码中。毫无疑问，这些都是开发者所乐意看到的。

13.1 背 景

目前，Spring 提供的远程服务主要有：

- RMI：借助于 Spring 提供的 `RmiProxyFactoryBean` 和 `RmiServiceExporter`，开发者能够开发 RMI 应用。当然，Spring 同时支持两种方式开发 RMI 应用。其一，基于传统的方式，即同 `java.rmi.Remote` 和 `java.rmi.RemoteException` 配合使用。其二，基于 RMI Invoker，即可以使用任何 Java 接口（自从 JDK 5.0 开始，引入了新的 RMI 开发模型，即通过动态代理实现 RMI 应用开发，类似于 RMI Invoker。这使得开发者不用借助于 `rmic` 实用工具生成 RMI stub 存根类，从而能够简化 RMI 开发模型，但前提是 RMI 客户端和服务端同时必须是 JDK 5.0。）。这里的 RMI Invoker 同 JBoss 应用服务器 3.x/4.x 实现 Invoker 有些类似。这使得开发者再也不用手工编译 RMI 以获得 RMI 存根，从而大大提高应用的可部署性。
- Hessian：由 Caucho¹提供的、使用轻量级二进制的 HTTP 协议，而实现的远程服务机制。Spring 提供了配套的 `HessianProxyFactoryBean` 和 `HessianServiceExporter`。
- Burlap：基于 XML 形式，是对 Hessian 的替换。当然，如果需要开发基于 Burlap 的远程服务，则开发者需要借助于 Spring 提供的 `BurlapProxyFactoryBean` 和

¹ <http://www.caucho.com/>

BurlapServiceExporter。

- Spring 提供的 HTTP Invoker: 借助于 HTTP Invoker, 开发者能够实现服务的远程化。这同 RMI Invoker 类似 (仅局限于其使用和配置方面)。当然, 这需要借助于 Spring 框架提供的 HttpInvokerProxyFactoryBean 和 HttpInvokerServiceExporter 实现。
- Web 服务 (JAX-RPC): 借助于 Spring 提供的 JaxRpcPortProxyFactoryBean 和 JaxRpcPortClientInterceptor 实现服务的远程化。
- 企业 Bean: 借助于 Spring 提供的抽象类而实现 EJB 应用的访问和开发。本书已经在第 10 章深入研究过这方面的内容。

其中, 表 13-1 给出了 Spring 支持的远程服务类型及其适用场合。在本书前面的章节中, 开发者应该已经看到 Spring 提供的各种服务类型的实现都是十分相似的。因此, 在 Spring Team 实现 Spring 远程服务过程中也是依据类似原则而架构并实现的, 即通过 Spring 受管 JavaBean 实现。

表 13-1 Spring 支持的远程服务类型

远程服务类型	适用场合
RMI (远程方法调用)	如果客户和服务端不受防火墙, 或其他网络约束, 则可以使用 RMI 访问或暴露 POJO 服务
Hessian (Caucho)	借助于 HTTP 协议, 而实现对 POJO 的远程使能服务, 即通过 Hessian 访问或暴露 POJO 服务 (基于二进制消息)。其中, 它使用了专属的序列化机制
Burlap (Caucho)	借助于 HTTP 协议, 而实现对 POJO 的远程使能服务, 即通过 Hessian 访问或暴露 POJO 服务 (基于 XML 技术)。其中, 它使用了专属的序列化机制
HTTP Invoker	吸收 RMI、Hessian、Burlap 的优点, 借助于 Spring 原生的 HTTP Invoker 能够在 HTTP 协议基础上实现 POJO 远程服务的暴露或访问。其中, 能够秉承 Java 对象序列化的优点
Web 服务 (JAX-RPC)	借助于 JAX-RPC 方式访问 Web 服务
企业 Bean (Enterprise JavaBeans)	适合于访问 EJB 应用或开发 EJB 应用。本书已经在第 10 章深入研究过这方面的内容

在客户端, 借助于 Spring 提供的代理工厂 JavaBean, 开发者能够配置和使用上述各种类型的远程服务。开发者在熟悉某种远程服务类型后, 研究并使用其他类型的远程服务将变得很简单。代理工厂 JavaBean 将负责完成客户请求, 即它同远程服务进行交互。比如, 连接到远程服务、调用远程服务、对返回结果进行处理等都是代理工厂 JavaBean 需要完成的。最终, 客户操作远程服务就像操作本地对象一样。

在服务器端, 通过 Spring 受管 JavaBean 能够实现 POJO 远程服务的架构和暴露。

注意, 为实现 Spring 远程服务的访问和实现, 开发者几乎不用开发任何远程服务相关的代码。一般情况下, 在访问远程服务过程中, 开发者通常都需要在 Java/J2EE 代码中抛出 java.rmi.RemoteException 异常 (比如远程服务未启动、网络中断、客户信息有误等), 而借助于 Spring 提供的代理工厂 JavaBean 使用远程服务却不必要如此。原因是: 代理工厂 JavaBean 能够处理 java.rmi.RemoteException, 并且将其作为未受查 RemoteException

异常抛出。既然导致抛出 `RemoteException` 异常的原因往往是不可恢复的，比如网络中断、远程服务不可用，因此 `org.springframework.remoting.RemoteAccessException` 的抛出使得客户有选择性地对其进行处理。这就是未受查异常所带来的优势之一。

其中，Spring 为远程服务提供了如下若干异常类型。`RemoteLookupFailureException` 表明查找不到目标远程服务；`RemoteConnectFailureException` 表明连接不到目标远程服务。不知道开发者是否注意到，这些异常同具体的远程服务类型都没有直接关系，因此在开发 Spring 使能的远程服务过程中，客户代码并不需要处理同特定远程服务相关的异常。

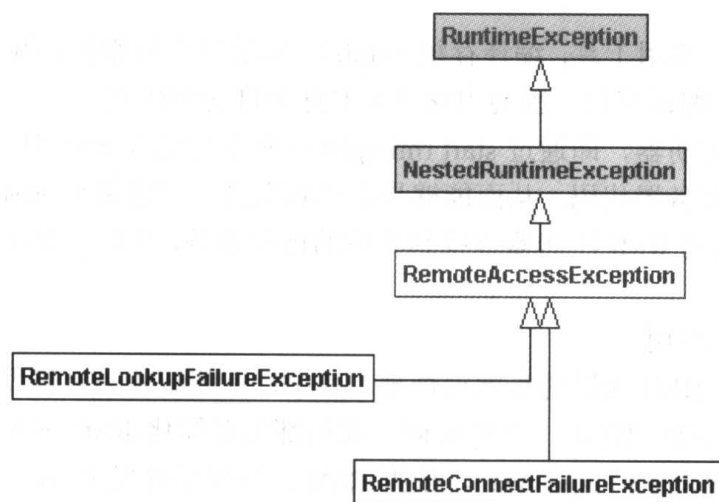


图 13-1 `RemoteAccessException` 及相关异常

接下来，本书将一一阐述各种远程服务的具体使用和机理。

13.2 Spring 对远程服务提供的支持

13.2.1 RMI 使能服务

RMI (Remote Method Invocation) 是 Java 2 标准版中的核心 API，它是自 JDK 1.1 开始就引入到 Java 平台的。同时，它还提供了相应的类库，供开发者在不同 JVM 间进行对象和方法访问操作，而这些 JVM 可以运行在不同的物理机器上。RMI 在 Java 分布式计算领域担当了重要的作用，而且整个 Java 社区在持续改进 RMI。在 RMI 出现之前，开发者只能够借助于 CORBA 同 Java 应用进行交互。注意，JDK 本身实现了 CORBA ORB，即对象请求 Broker。在 RMI 之后，J2EE 平台中引入了 RMI-IIOP，将 RMI 和 CORBA 的优点全部吸收过来，从而更好地实现 J2EE 应用间及 J2EE 应用与其他 CORBA 应用的交互²。

通常，开发 RMI 应用的步骤如下：

- 开发继承于 `java.rmi.Remote` 的业务接口类。开发者需要在每个业务方法的签名中抛出 `java.rmi.RemoteException`。

² 建议开发者参考《Java Programming with CORBA, 3RD Edition》一书，由 Wiley 于 2001 年出版。

- 实现业务接口类。在实现业务接口类过程中，除了需要实现业务接口类外，还需要继承于 `java.rmi.server.UnicastRemoteObject`。当然，如果开发者希望以静态方式，即 `UnicastRemoteObject.exportObject()` 来导出远程服务，则不用在实现业务接口类过程中继承 `UnicastRemoteObject`。
- 开发 RMI 服务器代码，将远程服务注册到 RMI 注册器（registry）中。开发者需要借助于 Naming 提供的绑定操作，从而供客户调用。
- 开发客户端代码。借助于 `Naming.lookup()` 方法能够查找到所需的远程服务，并且调用它。
- 借助于 `rmic` 实用工具，编译存根（stub）。如果客户与服务器端同时使用 JDK 5.0，则不用手工编译存根，因为 JDK 5.0 本身会自动创建它。
- 启动 RMI 注册器，即通过 `start rmiregistry` 命令行完成 RMI 注册器的启动。
- 启动 RMI 服务器代码，从而能够导出远程服务，并注册到 RMI 注册器中。进而，客户可以通过 RMI 注册器查找到具体的远程服务，然后能够调用到具体的远程服务。
- 运行客户端应用。

可以看出，开发 RMI 应用的步骤很烦琐。其中，有些步骤需要开发者手工完成，有些能够由 RMI 编程模型自动完成。无论如何，我们都需要简化 RMI 应用的开发和访问。借助于 Spring 提供的代理工厂 `JavaBean`，依据 RMI 开发模型开发 RMI 应用变得相当简单、简洁。这使得开发 RMI 应用就像开发本地应用或桌面应用一样。好了，开发者可能已经期盼很久了。下面，来看看具体开发过程。

本书将结合 example24 展开对 Spring 提供的 RMI 使能服务进行研究。首先，开发者需要开发一个值对象，它将作为 RMI 应用的参数进行传递。这个值对象由 `PersonVO.java` 类完成。它由姓（`lastname`）和名（`firstname`）这样两个 `Java` 域组成。借助于 Eclipse 提供的代码辅助功能，能够快速开发出这个 `PersonVO` 值对象类。

```
package com.openv.spring;

import java.io.Serializable;

/**
 * Person 值对象
 *
 * @author luoshifei
 */
public class PersonVO implements Serializable {
    private String firstname;

    private String lastname;

    /**
     * @return Returns the firstname.
     */
    public String getFirstname() {
```

```

        return firstname;
    }

    /**
     * @param firstname
     *      The firstname to set.
     */
    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    /**
     * @return Returns the lastname.
     */
    public String getLastname() {
        return lastname;
    }

    /**
     * @param lastname
     *      The lastname to set.
     */
    public void setLastname(String lastname) {
        this.lastname = lastname;
    }
}

```

其次，开发 `ILogPerson` 接口，供导出远程服务使用。开发者应该注意到，`ILogPerson` 业务接口中的方法签名中并没有抛出 `java.rmi.RemoteException` 异常。另外，`ILogPerson` 也没有继承于 `java.remote.Remote`。如果直接借助于 Java 本身提供的 RMI 编程模型（上述内容已经给出了具体步骤），则抛出 `RemoteException` 异常和继承 `Remote` 必不可少。Spring 借助于代理工厂 `JavaBean` 和 `RMIServiceExporter` 能够消除业务接口类中的这方面需求。因此，`ILogPerson` 是更加纯粹的 Java 接口，因为它没有感知到分布式计算模型的存在。最为重要的一点是，使用 `ILogPerson.java` 的客户代码再也不用处理 `RemoteException` 异常了。

```

package com.openv.spring;

/**
 * ILogPerson 接口
 *
 * @author luoshifei
 */
public interface ILogPerson {

    public String getPersion(PersonVO personVO);

}

```

第三, 实现 `ILogPerson` 接口。依据 `ILogPerson.java` 接口, 完成 `LogPerson` 类的开发。

```
package com.openv.spring;

/**
 * ILogPerson 实现
 *
 * @author luoshifei
 */
public class LogPerson implements ILogPerson {

    /**
     * 返回个人名字
     *
     * @param personVO 值对象
     *
     * @return 个人名字
     */
    public String getPersion(PersonVO personVO) {
        return personVO.getLastname() + ", "
            + personVO.getFirstname();
    }
}
```

通过上述过程, 整个 RMI 远程服务的 Java 代码已经开发完成。期间, 开发者并没有看到任何同 RMI 编程模型相关的内容。Spring 能够将任何 POJO 导出成 RMI 远程服务, 或者其他类型的远程服务(比如, HTTP Inoker)。同 RMI 编程模型相关的内容都是通过 Spring 配置文件完成的, 这也体现出 Spring 实用性的一面。

第四, 配置 RMI 服务端使用的 Spring 配置文件, 即 `appcontextrmiserver.xml`。

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

    <bean id="logPerson" class="com.openv.spring.LogPerson"/>

    <bean id="logPersonService"
        class="org.springframework.remoting.rmi.RmiServiceExporter">

        <!-- RmiServiceExporter 对服务名没有特殊要求 -->

        <property name="serviceName">
            <value>LogPerson</value>
        </property>

        <property name="service">
```

```

        <ref bean="logPerson"/>
    </property>

    <property name="serviceInterface">
        <value>com.openv.spring.ILogPerson</value>
    </property>

    <!-- 避免与默认 RMI 注册端口冲突, 因此修改为 1200 -->

    <property name="registryPort">
        <value>1200</value>
    </property>
</bean>

</beans>

```

其中, `RmiServiceExporter` 起到了很重要的作用。借助于它, 开发者不用直接使用 `rmic` 生成存根、不用通过程序手工将远程服务注册到 RMI 注册器中。`RmiServiceExporter` 为开发者完成了这些繁琐的工作。通过 `serviceName` 属性能够指定 RMI 服务名, 即对应于 Naming 绑定操作中的逻辑服务名; 通过 `service` 属性能够指定提供业务逻辑的实现类, 即 `LogPerson`; 通过 `serviceInterface` 属性能够指定 `LogPerson` 服务实现的接口, 即客户将通过该接口同 RMI 远程服务(`LogPerson`)进行交互; 通过 `registryPort` 属性能够修改默认 RMI 注册端口。

第五, 配置使用上述 RMI 远程服务, 即供客户使用的 Spring 配置文件 (`appcontextrmiclient.xml`)。

```

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

    <bean id="LogPerson"
        class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
        <property name="serviceUrl">
            <value>rmi://localhost:1200/LogPerson</value>
        </property>
        <property name="serviceInterface">
            <value>com.openv.spring.ILogPerson</value>
        </property>
    </bean>

</beans>

```

开发者是否注意到 `RmiProxyFactoryBean` 是工厂 `JavaBean`, 它是 RMI 远程服务的代理。而且, 它能够捕捉到 `RemoteException`, 并将 `RemoteException` 转化为未受查的异常。通过 `serviceUrl` 属性能够指定 RMI 的 URL; 通过 `serviceInterface` 属性能够指定业务接口。因此, 借助于 `RmiProxyFactoryBean` 简化了对 RMI 服务的访问。

最后, 开发者还需要提供 RMI 客户应用, 即 `LogPersonRmiClient.java` 源文件。

```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

/**
 * LogPerson RMI 客户端应用
 *
 * @author luoshifei
 */
public class LogPersonRmiClient {
    protected static final Log log =
        LogFactory.getLog(LogPersonRmiClient.class);

    public static void main(String[] args) {
        //初始化 appcontextrmiclient.xml
        Resource cresource = new ClassPathResource("appcontextrmiclient.xml");
        BeanFactory cfactory = new XmlBeanFactory(cresource);

        //实例化 Person 值对象
        PersonVO personVO = new PersonVO();
        personVO.setFirstname("Luo");
        personVO.setLastname("Shifei");

        //获得 RMI 服务
        ILogPerson clientLog = (ILogPerson) cfactory.getBean("LogPerson");

        //调用 RMI 服务
        log.info(clientLog.getPersion(personVO));
    }
}
```

通过“LogPerson”逻辑名能够获得 RMI 服务，最终调用到 RMI 服务提供的实例方法。通过上述 6 个步骤，开发者已经完成了 RMI 使能服务的开发。运行 Ant build.xml 中的 runserver 任务，即启动 RMI 使能服务器。开发者可以获得如下结果。

```
Buildfile: D:\workspace\example24\build.xml
compile:
runserver:
[java] 2004-12-26 17:05:36 org.springframework.beans.factory.
xml.XmlBeanDefinitionReader loadBeanDefinitions
```

```
[java] 信息: Loading XML bean definitions from class path resource
[appcontextrmiserver.xml]
[java] 2004-12-26 17:05:37 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'logPersonService'
[java] 2004-12-26 17:05:37
org.springframework.beans.factory.support.AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'logPerson'
[java] 2004-12-26 17:05:37 org.springframework.remoting.rmi.
RmiServiceExporter getRegistry
[java] 信息: Looking for RMI registry at port '1200'
[java] 2004-12-26 17:05:37 org.springframework.remoting.rmi.
RmiServiceExporter getObjectToExport
[java] 信息: RMI object 'LogPerson' is an RMI invoker
[java] 2004-12-26 17:05:37 org.springframework.core.CollectionFactory
<clinit>
[java] 信息: Using JDK 1.4 collections
[java] 2004-12-26 17:05:37
org.springframework.remoting.rmi.RmiServiceExporter afterPropertiesSet
[java] 信息: Binding RMI service 'LogPerson' to registry at port '1200'
```

然后, 运行 runclient 任务。

Buildfile: D:\workspace\example24\build.xml

compile:

runclient:

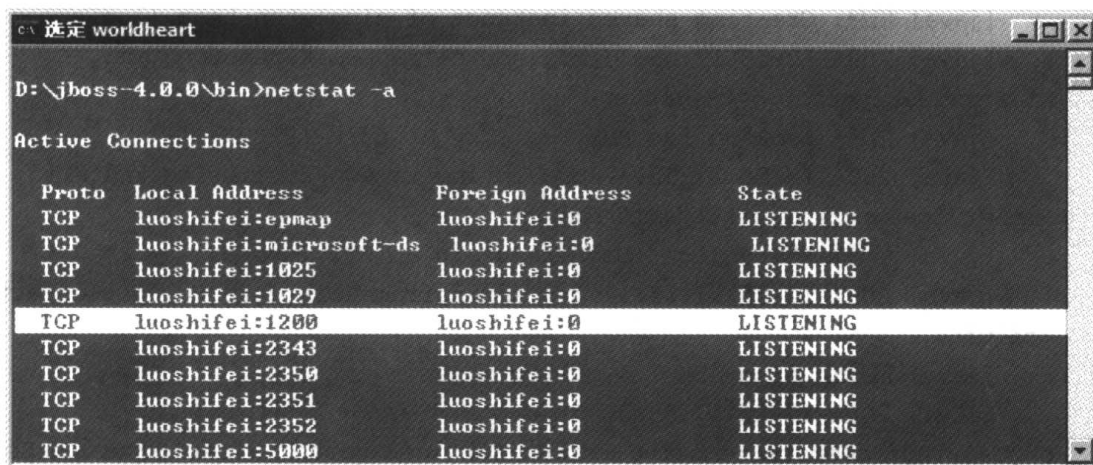
```
[java] 2004-12-26 17:06:17 org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
[java] 信息: Loading XML bean definitions from class path resource
[appcontextrmiclient.xml]
[java] 2004-12-26 17:06:18 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'LogPerson'
[java] 2004-12-26 17:06:18 org.springframework.remoting.rmi.
RmiClientInterceptor lookupStub
[java] 信息: Located object with RMI URL [rmi://localhost:1200/LogPerson]:
value=[RmiInvocationWrapper_Stub[UnicastRef [liveRef: [endpoint:
[10.33.5.46:1277] (remote), objID: [-1d4b83d6:1010e9f9fcb:-8000, 0]]]]]
[java] 2004-12-26 17:06:18
org.springframework.remoting.rmi.RmiClientInterceptor afterPropertiesSet
[java] 信息: RMI stub [rmi://localhost:1200/LogPerson] is an RMI invoker
[java] 2004-12-26 17:06:18 org.springframework.core.CollectionFactory
<clinit>
[java] 信息: Using JDK 1.4 collections
[java] 2004-12-26 17:06:18 com.openv.spring.LogPersonRmiClient main
[java] 信息: Shifei,Luo
BUILD SUCCESSFUL
Total time: 2 seconds
```

通过观察服务器端，开发者能够看到有如下日志输出。

```
[java] 2004-12-26 17:06:18 com.openv.spring.LogPerson getPersion
```

```
[java] 信息: Shifei,Luo
```

其中，example24 是使用 RMI Invoker 形式开发 RMI 使能服务的。如果开发者通过 Windows CMD 控制台，输入 netstat -a 命令，则能够看到 1200 端口被 RMI 远程服务占用了，如图 13-2 所示。



Proto	Local Address	Foreign Address	State
TCP	luoshifei:epmap	luoshifei:0	LISTENING
TCP	luoshifei:microsoft-ds	luoshifei:0	LISTENING
TCP	luoshifei:1025	luoshifei:0	LISTENING
TCP	luoshifei:1029	luoshifei:0	LISTENING
TCP	luoshifei:1200	luoshifei:0	LISTENING
TCP	luoshifei:2343	luoshifei:0	LISTENING
TCP	luoshifei:2350	luoshifei:0	LISTENING
TCP	luoshifei:2351	luoshifei:0	LISTENING
TCP	luoshifei:2352	luoshifei:0	LISTENING
TCP	luoshifei:5000	luoshifei:0	LISTENING

图 13-2 Windows CMD 控制台

开发者是否注意到，在 RMI 服务器端，需要借助于 RmiServiceExporter 将 POJO 导出为 RMI 服务；在 RMI 客户端，需要借助于 RmiProxyFactoryBean 访问并使用 RMI 使能服务。它们的关系见图 13-3。

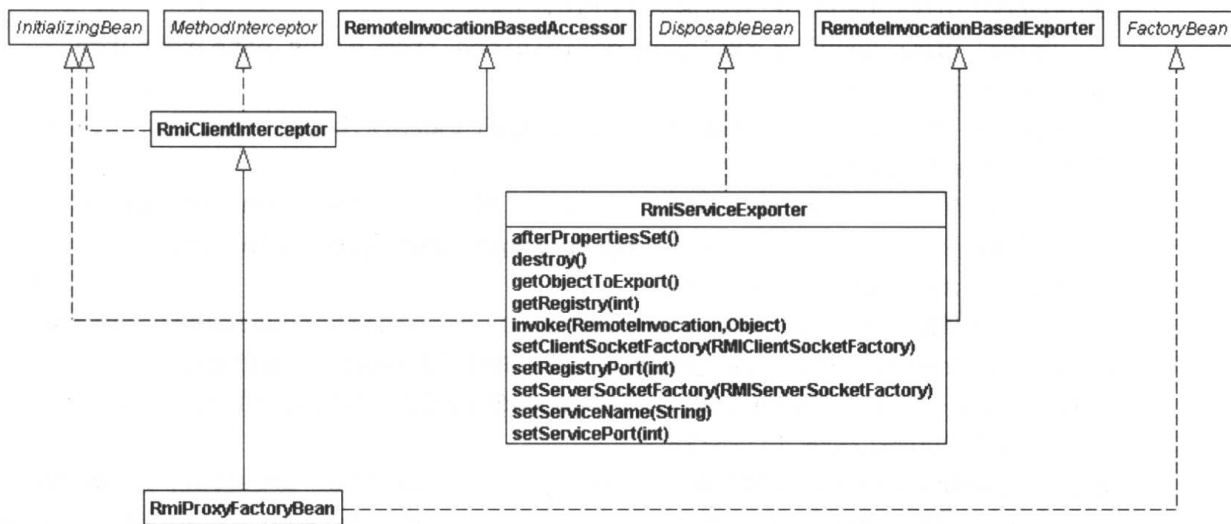


图 13-3 提供 RMI 使能服务的类图

当然，如果开发者需要借助于 JNDI 访问 RMI 使能服务，则需要借助于提供 RMI 使能服务的 JNDI 版本，见图 13-4 所示。由于机理类似，在此不给出详细的讨论，这可以留给开发者自行研究。

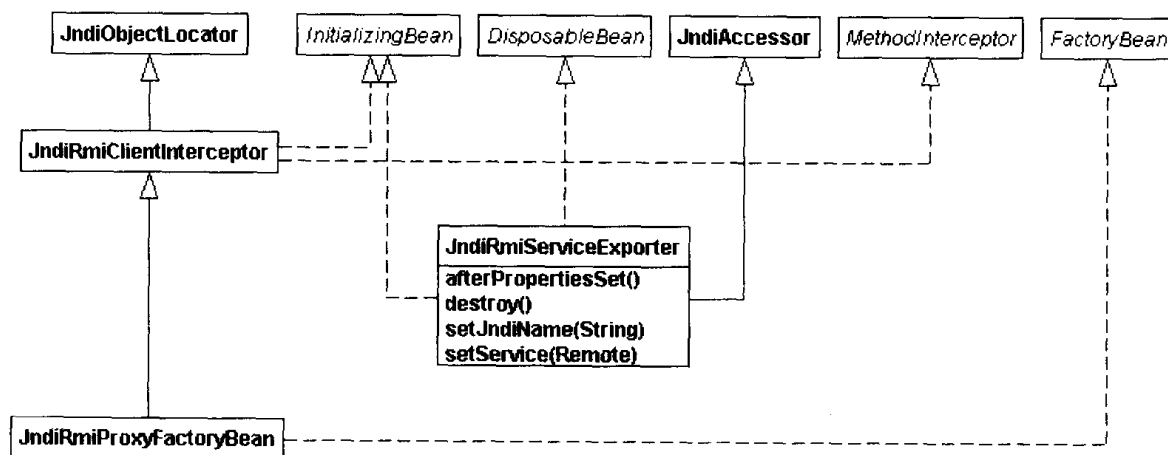


图 13-4 提供 JNDI 版本的 RMI 使能服务的类图

13.2.2 Hessian 使能服务

RMI 是 Java 平台中实施分布式计算的技术。毫无疑问，RMI 提供了一流的编程模型，但是技术本身都有特定的适用场合，正如表 13.1 给出的一样，RMI 也不例外。如果网络中存在防火墙约束，则 RMI 无能为力。在 RMI 开发模型中，客户和服务端必须同时采用 Java 语言，否则不能够使用 RMI 技术。

针对这种技术局限性，著名的 Resin 应用服务器提供商 Caucho Technology 公司在 2002 年初就开发实现了 Hessian 和 Burlap。它们能够克服 RMI 的局限性。发展至今，它们已经存在了 3 年时间，业界已经广泛部署了基于 Hessian 和 Burlap 的应用，因此它们是成熟的技术框架。

Hessian 和 Burlap 是基于 HTTP 的轻量级远程服务。一方面，Hessian 借助于二进制消息实现客户与服务端端的交互。注意，这种二进制消息能够在不同编程语言间移植，即它在语言间是便携的。另一方面，Burlap 借助于 XML 实现客户与服务端端的交互。只要目标语言支持 XML 的分析和处理，则它能够支持 Burlap。注意，Burlap 的 XML 消息结构很简单，这同 Web 服务的 SOAP 消息存在很大不同。

无论是 Hessian，还是 Burlap，因为其轻量性，使得在 Java Applet 或内存受限（比如一些手持设备）的场合广为实施。接下来，本书将分别阐述 Hessian 和 Burlap 提供的远程服务。

本节将结合 example25 研究 Hessian 使能服务。由于本实例是在 example24 基础之上开发的，因此对于与 example24 相同的部分本书不再给出。至于 example25 的详细情况，请开发者参考本书配套光盘。

由于 example25 是 Web 应用，因此开发者在部署 Web 应用过程中，所需要的类库可以根据部署过程中的错误或提示信息添加，图 13-5 给出了使用到的 jar 库。

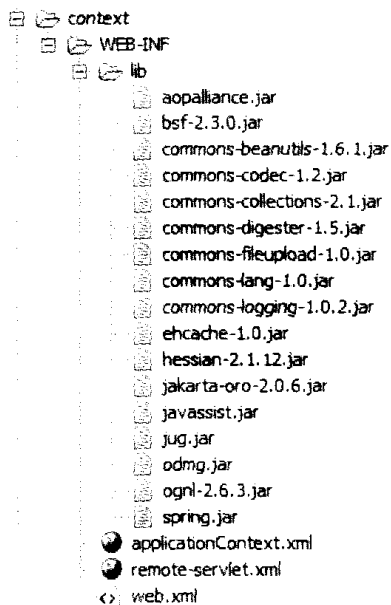


图 13-5 example25.war 使用到的 jar 库

首先，开发者需要提供 remote-servlet.xml Spring 配置文件，供导出 Hessian 使能服务使用。其中，Spring 提供的 DispatcherServlet 将需要定位到对应的 Spring 配置文件，即 remote-servlet.xml。开发者能够看出，HessianServiceExporter 同 RMIServiceExporter 类似，它们都用于实现远程服务。通过 service 属性指定业务实现实例；通过 serviceInterface 指定暴露给客户的调用接口。注意，这里并不存在 serviceName 属性，RMI 服务中存在 serviceName。由于 Hessian 不需要注册器完成服务的注册，因此不需要 serviceName 属性。

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

    <bean name="/logPersonHessian"

        class="org.springframework.remoting.caucho.HessianServiceExporter">
        <property name="service">
            <ref bean="logPerson"/>
        </property>
        <property name="serviceInterface">
            <value>com.openv.spring.ILogPerson</value>
        </property>
        </bean>

</beans>
```

注意，开发者还能够通过 Spring MVC 框架提供的 SimpleUrlHandlerMapping 负责 HTTP URL 的映射³。比如，开发者能够使用如下配置代码替换上述内容。这两种方式的不同点主

³ 第 14 章将作进一步阐述。

要有：其一，logPersonHessian 名不同，上述配置方案需要提供“/”，即不需要显示借助于 SimpleUrlHandlerMapping 实现 URL 的映射；其二，如果需要统一配置实现业务逻辑的 JavaBean，则显示借助于 SimpleUrlHandlerMapping 能够将所有的 URL 集中管理。比如在 mappings 属性中定义相应的值能够将逻辑 URL 映射到具体的业务逻辑 JavaBean。可以看出，它将/logPersonHessian 映射到 logPersonHessian。

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

    <bean name="logPersonHessian"

        class="org.springframework.remoting.caucho.HessianServiceExporter">
        <property name="service">
            <ref bean="logPerson"/>
        </property>
        <property name="serviceInterface">
            <value>com.openv.spring.ILogPerson</value>
        </property>
    </bean>

    <bean id="urlMapping"
        class="org.springframework.web.servlet.
            handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <props>
                <prop key="/logPersonHessian">
                    logPersonHessian
                </prop>
            </props>
        </property>
    </bean>

</beans>
```

然后，开发者还需提供 applicationContext.xml Spring 配置文件，供配置业务实现 JavaBean 使用。其中配置的 logPerson JavaBean 实例供 HessianServiceExporter 中的 service 属性使用。因此，在 HessianServiceExporter 处理客户请求时，将通过 service 属性引用的 JavaBean 实例完成具体的业务逻辑调用。

```
<?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="logPerson" class="com.openv.spring.LogPerson"/>
```

```
</beans>
```

第三，提供 web.xml 文件。由于 Hessian 是基于 HTTP 协议的使能服务，因此需要基于 HTTP 方式使用它。在 Spring MVC 框架中，提供了 DispatcherServlet，负责将客户请求转发给目标服务提供者，比如 “/logPersonHessian”。其中，HessianServiceExporter 充当了 Spring MVC 控制器的作用。

```
<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">

  <display-name>example25</display-name>

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

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

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

<servlet-mapping>
  <servlet-name>remote</servlet-name>
  <url-pattern>/remote/*</url-pattern>
</servlet-mapping>

</web-app>
```

开发者是否注意到，通过 contextConfigLocation 装载了 applicationContext.xml。其中，org.springframework.web.servlet.DispatcherServlet 对应的 remote 名构成了 “remote-servlet.xml” 中的 *-servlet.xml 前缀，即如果将 org.springframework.web.servlet.DispatcherServlet 的 Servlet 名设置为 remoting，则需要提供 remoting-servlet.xml Spring 配置文件。

第四，借助于 Ant build.xml 中的 deploy 任务，将 example25.war 部署到 JBoss 4.0.0 中。

在 JBoss 控制台中，将输出如下日志信息。

```
18:41:13,709 INFO [TomcatDeployer] deploy, ctxPath=/example25, warUrl=
file:/D:/jboss-4.0.0/server/default/tmp/deploy/tmp2123example25-exp.war/
18:41:14,059 INFO [Engine] StandardContext[/example25]Loading root
WebApplicationContext
18:41:14,319 INFO [XmlBeanDefinitionReader] Loading XML bean definitions from
resource [/WEB-INF/applicationContext.xml] of ServletContext
18:41:14,600 INFO [XmlWebApplicationContext] Bean factory for application
context [Root XmlWebApplicationContext]:
org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [logPerson]; root of BeanFactory hierarchy
18:41:14,630 INFO [XmlWebApplicationContext] 1 beans defined in application
context [Root XmlWebApplicationContext]
18:41:14,650 INFO [XmlWebApplicationContext] No message source found for
context [Root XmlWebApplicationContext]: using empty default
18:41:14,670 INFO [XmlWebApplicationContext] No ApplicationEventMulticaster
found for context [Root XmlWebApplicationContext]: using default
18:41:14,690 INFO [UiApplicationContextUtils] No ThemeSource found for [Root
XmlWebApplicationContext]: using ResourceBundleThemeSource
18:41:14,700 INFO [XmlWebApplicationContext] Refreshing listeners
18:41:14,720 INFO [DefaultListableBeanFactory] Pre-instantiating singletons
in factory
[org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [logPerson]; root of BeanFactory hierarchy]
18:41:14,720 INFO [DefaultListableBeanFactory] Creating shared instance of
singleton bean 'logPerson'
18:41:14,900 INFO [ContextLoader] Using context class
[org.springframework.web.context.support.XmlWebApplicationContext] for root
WebApplicationContext
18:41:14,900 INFO [ContextLoader] Published root WebApplicationContext
[org.springframework.web.context.support.XmlWebApplicationContext:
displayName=[Root XmlWebApplicationContext]; startupdate=[Sun Nov 14 18:41:14
CST 2004]; root of ApplicationContext hierarchy; config
locations=[/WEB-INF/applicationContext.xml]; ] as ServletContext attribute
with name [interface
org.springframework.web.context.WebApplicationContext.ROOT]
18:41:14,950 INFO [DispatcherServlet] Initializing servlet 'remote'
18:41:15,000 INFO [DispatcherServlet] Framework servlet 'remote' init
18:41:15,000 INFO [Engine] StandardContext[/example25]Loading
WebApplicationContext for servlet 'remote'
18:41:15,000 INFO [DispatcherServlet] Servlet with name 'remote' will try to
create custom WebApplicationContext context of class
'org.springframework.web.context.support.XmlWebApplicationContext' using
parent context
[org.springframework.web.context.support.XmlWebApplicationContext:
displayName=[Root XmlWebApplicationContext]; startupdate=[Sun Nov 14 18:41:14
```

```
CST 2004]; root of ApplicationContext hierarchy; config
locations=[/WEB-INF/applicationContext.xml]; ]
18:41:15,000 INFO [XmlBeanDefinitionReader] Loading XML bean definitions from
resource [/WEB-INF/remote-servlet.xml] of ServletContext
18:41:15,060 INFO [XmlWebApplicationContext] Bean factory for application
context [XmlWebApplicationContext for namespace 'remote-servlet']:
org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [/logPersonHessian]; parent:
org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [logPerson]; root of BeanFactory hierarchy
18:41:15,060 INFO [XmlWebApplicationContext] 1 beans defined in application
context [XmlWebApplicationContext for namespace 'remote-servlet']
18:41:15,060 INFO [XmlWebApplicationContext] No message source found for
context [XmlWebApplicationContext for namespace 'remote-servlet']: using empty
default
18:41:15,070 INFO [XmlWebApplicationContext] No ApplicationEventMulticaster
found for context [XmlWebApplicationContext for namespace 'remote-servlet']:
using default
18:41:15,070 INFO [UiApplicationContextUtils] No ThemeSource found for
[XmlWebApplicationContext for namespace 'remote-servlet']: using
ResourceBundleThemeSource
18:41:15,070 INFO [XmlWebApplicationContext] Refreshing listeners
18:41:15,070 INFO [DefaultListableBeanFactory] Pre-instantiating singletons
in factory
[org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [/logPersonHessian]; parent:
org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [logPerson]; root of BeanFactory hierarchy]
18:41:15,070 INFO [DefaultListableBeanFactory] Creating shared instance of
singleton bean '/logPersonHessian'
18:41:15,271 INFO [CollectionFactory] Using JDK 1.4 collections
18:41:15,341 INFO [DispatcherServlet] Using context class
[org.springframework.web.context.support.XmlWebApplicationContext] for
servlet 'remote'
18:41:15,341 INFO [DispatcherServlet] Published WebApplicationContext of
servlet 'remote' as ServletContext attribute with name
[org.springframework.web.servlet.FrameworkServlet.CONTEXT.remote]
18:41:15,341 INFO [DispatcherServlet] Unable to locate MultipartResolver with
name [multipartResolver]: no multipart handling provided
18:41:15,351 INFO [DispatcherServlet] Unable to locate LocaleResolver with
name 'localeResolver': using default
[org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver@1fae1cd]
18:41:15,381 INFO [DispatcherServlet] Unable to locate ThemeResolver with name
'themeResolver': using default
[org.springframework.web.servlet.theme.FixedThemeResolver@c4f9e6]
18:41:15,421 INFO [BeanNameUrlHandlerMapping] Mapped URL path
```

```
[/logPersonHessian] onto handler
[org.springframework.remoting.caucho.HessianServiceExporter@160a2f1]
18:41:15,421 INFO [DispatcherServlet] No HandlerMappings found in servlet
'remote': using default
18:41:15,431 INFO [DispatcherServlet] No HandlerAdapters found in servlet
'remote': using default
18:41:15,491 INFO [DispatcherServlet] No ViewResolvers found in servlet
'remote': using default
18:41:15,491 INFO [DispatcherServlet] Framework servlet 'remote' init
completed in 491 ms
18:41:15,491 INFO [DispatcherServlet] Servlet 'remote' configured
successfully
```

第五，提供客户应用使用的 Spring 配置文件，即 `appcontexthessianclient.xml`。其中，`HessianProxyFactoryBean` 充当了代理工厂的作用，即负责创建 Hessian 代理。这同用于 RMI 服务的 `RMIProxyFactoryBean` 很相似。通过 `serviceUrl` 属性指定目标服务的 URL；通过 `serviceInterface` 属性指定业务接口。在 `serviceUrl` 属性指定的值中，“`http://localhost: 8080`”是服务器名和端口、“`example25`”是 Web 应用上下文、“`remote`”是 `DispatcherServlet` 名、“`logPersonHessian`”是目标服务。因此，正确构建 `serviceUrl` 至关重要，否则客户查找不到相应的服务。

```
<?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="LogPerson"

        class="org.springframework.remoting.caucho.HessianProxyFactoryBean">
            <property name="serviceUrl">
                <value>http://localhost:8080/example25/remote/logPersonHessian
                </value>
            </property>
            <property name="serviceInterface">
                <value>com.openv.spring.ILogPerson</value>
            </property>
        </bean>

</beans>
```

第六，开发客户应用。这部分内容同 RMI 部分开发客户应用时几乎没有区别。

```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
```

```
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

/**
 * LogPerson Hessian 客户应用
 *
 * @author luoshifei
 */
public class LogPersonHessianClient {
    protected static final Log log =
        LoggerFactory.getLog(LogPersonHessianClient.class);

    public static void main(String[] args) {
        //初始化 appcontexthessianclient.xml
        Resource cresource = new ClassPathResource(
            "appcontexthessianclient.xml");
        BeanFactory cfactory = new XmlBeanFactory(cresource);

        //实例化 Person 值对象
        PersonVO personVO = new PersonVO();
        personVO.setFirstname("Luo");
        personVO.setLastname("Shifei");

        //获得 Hessian 服务
        ILogPerson clientLog = (ILogPerson) cfactory.getBean("LogPerson");

        //调用 Hessian 服务
        log.info(clientLog.getPersion(personVO));
    }
}
```

最后，调用客户应用（执行 Ant build.xml 中 runclient 任务），执行结果如下。另外，在 JBoss 控制台将会出现服务调用日志（见图 13-6）。

```
Buildfile: D:\workspace\example25\build.xml
runclient:
[java] 2004-12-26 17:44:28 org.springframework.
beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
[java] 信息: Loading XML bean definitions from class path resource
[appcontexthessianclient.xml]
[java] 2004-12-26 17:44:29 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'LogPerson'
[java] 2004-12-26 17:44:29 org.springframework.core.CollectionFactory
<clinit>
[java] 信息: Using JDK 1.4 collections
```



```
[java] 2004-12-26 17:44:30 com.openv.spring.LogPersonHessianClient main
[java] 信息: Shifei,Luo
BUILD SUCCESSFUL
Total time: 2 seconds
```

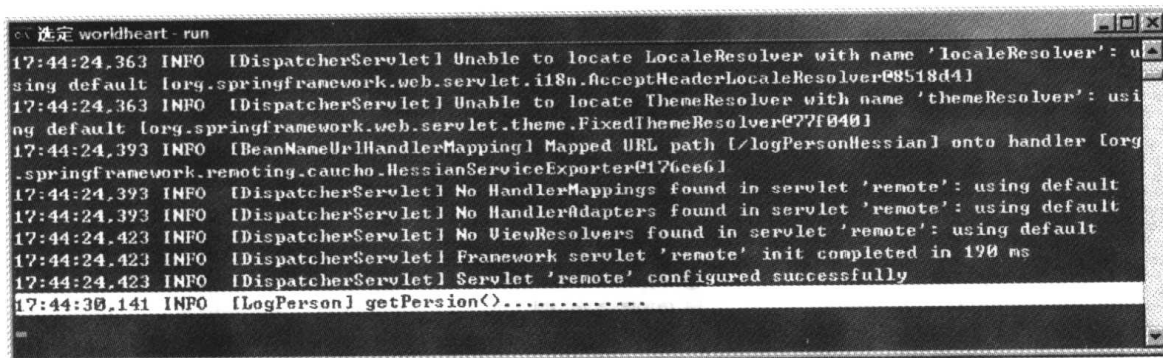


图 13-6 JBoss 控制台服务器日志

其中, Hessian 使能服务使用到的相关类见图 13-7。

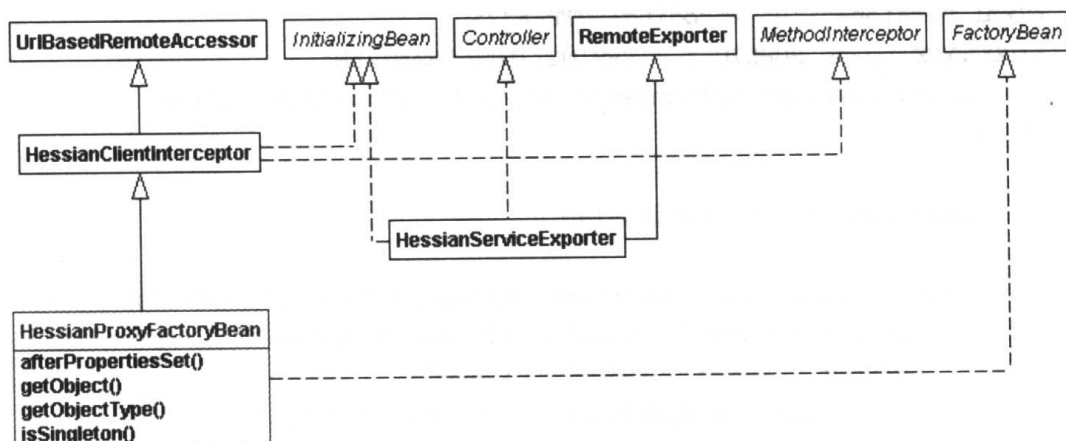


图 13-7 提供 Hessian 使能服务的类图

通过本节内容开发者能够看出, 实现 Hessian 使能服务同 RMI 使能服务的过程很类似, 这正是 Spring 架构远程服务 (包括其他企业级服务) 的优势之一。无论何种远程服务, 使用和开发过程都是类似的, 从而大大降低开发者学习曲线。借助于 Spring IoC 和 AOP, 开发者能够简化对 Hessian 的使用。至于 Hessian 更深入的研究, 请开发者参考 Hessian 网站。

13.2.3 Burlap 使能服务

本节内容将结合 example26, 来研究 Burlap 使能服务。当然, 由于 example26 是在 example25 基础上开发的, 因此本节内容只是给出两者的不同点。其中, example26.war 应使用到的 jar 库见图 13-8 所示。

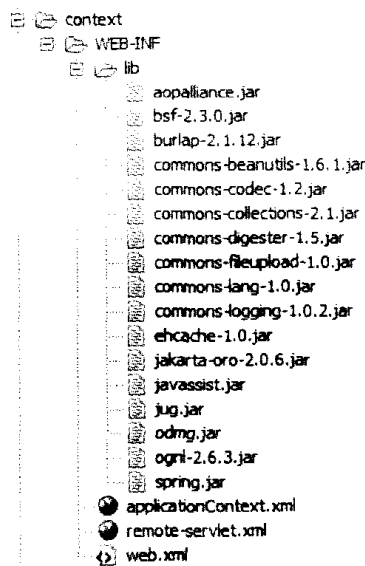


图 13-8 example26.war 使用到的 jar 库

首先，remote-servlet.xml 内容如下。开发者只需要将 HessianServiceExporter 替换成 Burlap 相应的 BurlapServiceExporter 即可完成 remote-servlet.xml 的配置。

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

    <bean name="/logPersonBurlap"

        class="org.springframework.remoting.caucho.BurlapServiceExporter">
            <property name="service"><ref bean="logPerson"/></property>
            <property name="serviceInterface">
                <value>com.openv.spring.ILogPerson</value>
            </property>
        </bean>

</beans>
```

其次，Spring 客户端使用的配置文件，即 appcontextburlapclient.xml。类似于前面阐述的 Hessian，开发者需要为 Burlap 提供 BurlapProxyFactoryBean。

```
<?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="LogPerson"

        class="org.springframework.remoting.caucho.BurlapProxyFactoryBean">
            <property name="serviceUrl">
                <value>http://localhost:8080/example26/remote/logPersonBurlap
```

```

        </value>
    </property>
    <property name="serviceInterface">
        <value>com.openv.spring.ILogPerson</value>
    </property>
</bean>

```

```
</beans>
```

第三，客户应用如下。可以看出，为 RMI、Hessian、Burlap 使能服务开发的客户应用几乎没有什么区别，Spring 为它们提供了合理的、一致的抽象层，这使得开发者在开发应用时可以灵活地更换所需的远程服务，而且不用修改源代码。

```

package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

/**
 * LogPerson Burlap 客户应用
 *
 * @author luoshifei
 */
public class LogPersonBurlapClient {
    protected static final Log log =
        LogFactory.getLog(LogPersonBurlapClient.class);

    public static void main(String[] args) {
        //初始化 appcontextburlapclient.xml
        Resource cresource = new
            ClassPathResource("appcontextburlapclient.xml");
        BeanFactory cfactory = new XmlBeanFactory(cresource);

        //实例化 Person 值对象
        PersonVO personVO = new PersonVO();
        personVO.setFirstname("Luo");
        personVO.setLastname("Shifei");

        //获得 Burlap 服务
        ILogPerson clientLog = (ILogPerson) cfactory.getBean("LogPerson");
    }
}

```

```
//调用 Burlap 服务
log.info(clientLog.getPersion(personVO));
}
}
```

最后，运行上述客户应用。可以看出，LogPersonBurlapClient 成功调用了服务器端的 Burlap 使能服务。

```
Buildfile: D:\workspace\example26\build.xml
runclient:
    [java] 2004-12-26 18:00:39 org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
    [java] 信息: Loading XML bean definitions from class path resource
[appcontextburlapclient.xml]
    [java] 2004-12-26 18:00:40 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
    [java] 信息: Creating shared instance of singleton bean 'LogPerson'
    [java] 2004-12-26 18:00:40 org.springframework.core.CollectionFactory
<clinit>
    [java] 信息: Using JDK 1.4 collections
    [java] 2004-12-26 18:00:41 com.openv.spring.LogPersonBurlapClient main
    [java] 信息: Shifei,Luo
BUILD SUCCESSFUL
Total time: 2 seconds
```

与此同时，在 JBoss 4.0.0 控制台中，开发者可以浏览到如下信息。

```
19:18:31,767 INFO [TomcatDeployer] undeploy, ctxPath=/example25, warUrl=
file:/D:/jboss-4.0.0/server/default/tmp/deploy/tmp65279example25-exp.war/
19:18:31,767 INFO [Engine] StandardContext[/example25]Closing
WebApplicationContext of servlet 'remote'
19:18:31,767 INFO [XmlWebApplicationContext] Closing application context
[XmlWebApplicationContext for namespace 'remote-servlet']
19:18:31,767 INFO [DefaultListableBeanFactory] Destroying singletons in
factory
{org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [/logPersonHessian]; parent:
org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [logPerson]; root of BeanFactory hierarchy}
19:18:31,767 INFO [DefaultListableBeanFactory] Destroying inner beans in
factory
{org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [/logPersonHessian]; parent:
org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [logPerson]; root of BeanFactory hierarchy}
19:18:31,767 INFO [Engine] StandardContext[/example25]Closing root
WebApplicationContext
19:18:31,767 INFO [XmlWebApplicationContext] Closing application context
[Root XmlWebApplicationContext]
19:18:31,767 INFO [DefaultListableBeanFactory] Destroying singletons in
```

```
factory
{org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [logPerson]; root of BeanFactory hierarchy}
19:18:31,767 INFO [DefaultListableBeanFactory] Destroying inner beans in
factory
{org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [logPerson]; root of BeanFactory hierarchy}
19:18:38,016 INFO [TomcatDeployer] deploy, ctxPath=/example26,
warUrl=file:/D:/jboss-4.0.0/server/default/tmp/deploy/tmp65280example26-e
xp.war/
19:18:40,159 INFO [Engine] StandardContext[/example26]Loading root
WebApplicationContext
19:18:40,399 INFO [XmlBeanDefinitionReader] Loading XML bean definitions from
resource [/WEB-INF/applicationContext.xml] of ServletContext
19:18:40,700 INFO [XmlWebApplicationContext] Bean factory for application
context [Root XmlWebApplicationContext]:
org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [logPerson]; root of BeanFactory hierarchy
19:18:40,740 INFO [XmlWebApplicationContext] 1 beans defined in application
context [Root XmlWebApplicationContext]
19:18:40,760 INFO [XmlWebApplicationContext] No message source found for
context [Root XmlWebApplicationContext]: using empty default
19:18:40,780 INFO [XmlWebApplicationContext] No ApplicationEventMulticaster
found for context [Root XmlWebApplicationContext]: using default
19:18:40,790 INFO [UiApplicationContextUtils] No ThemeSource found for [Root
XmlWebApplicationContext]: using ResourceBundleThemeSource
19:18:40,810 INFO [XmlWebApplicationContext] Refreshing listeners
19:18:40,810 INFO [DefaultListableBeanFactory] Pre-instantiating singletons
in factory
{org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [logPerson]; root of BeanFactory hierarchy}
19:18:40,810 INFO [DefaultListableBeanFactory] Creating shared instance of
singleton bean 'logPerson'
19:18:41,000 INFO [ContextLoader] Using context class
[org.springframework.web.context.support.XmlWebApplicationContext] for root
WebApplicationContext
19:18:41,000 INFO [ContextLoader] Published root WebApplicationContext
[org.springframework.web.context.support.XmlWebApplicationContext:
displayName=[Root XmlWebApplicationContext]; startupdate=[Sun Nov 14 19:18:40
CST 2004]; root of ApplicationContext hierarchy; config
locations=[/WEB-INF/applicationContext.xml]; ] as ServletContext attribute
with name [interface
org.springframework.web.context.WebApplicationContext.ROOT]
19:18:41,050 INFO [DispatcherServlet] Initializing servlet 'remote'
19:18:41,100 INFO [DispatcherServlet] Framework servlet 'remote' init
19:18:41,100 INFO [Engine] StandardContext[/example26]Loading
```

```
WebApplicationContext for servlet 'remote'
19:18:41,110 INFO [DispatcherServlet] Servlet with name 'remote' will try to
create custom WebApplicationContext context of class 'org.springframework.web.
context.support.XmlWebApplicationContext' using parent context
[org.springframework.web.context.support.XmlWebApplicationContext:
displayName=[Root XmlWebApplicationContext]; startup date=[Sun Nov 14 19:18:40
CST 2004]; root of ApplicationContext hierarchy; config
locations=[/WEB-INF/applicationContext.xml]; ]
19:18:41,110 INFO [XmlBeanDefinitionReader] Loading XML bean definitions from
resource [/WEB-INF/remote-servlet.xml] of ServletContext
19:18:41,170 INFO [XmlWebApplicationContext] Bean factory for application
context [XmlWebApplicationContext for namespace 'remote-servlet']:
org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [/logPersonBurlap]; parent:
org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [logPerson]; root of BeanFactory hierarchy
19:18:41,170 INFO [XmlWebApplicationContext] 1 beans defined in application
context [XmlWebApplicationContext for namespace 'remote-servlet']
19:18:41,170 INFO [XmlWebApplicationContext] No message source found for
context [XmlWebApplicationContext for namespace 'remote-servlet']: using empty
default
19:18:41,170 INFO [XmlWebApplicationContext] No ApplicationEventMulticaster
found for context [XmlWebApplicationContext for namespace 'remote-servlet']:
using default
19:18:41,170 INFO [UiApplicationContextUtils] No ThemeSource found for
[XmlWebApplicationContext for namespace 'remote-servlet']: using
ResourceBundleThemeSource
19:18:41,170 INFO [XmlWebApplicationContext] Refreshing listeners
19:18:41,170 INFO [DefaultListableBeanFactory] Pre-instantiating singletons
in factory
[org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [/logPersonBurlap]; parent:
org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [logPerson]; root of BeanFactory hierarchy]
19:18:41,170 INFO [DefaultListableBeanFactory] Creating shared instance of
singleton bean '/logPersonBurlap'
19:18:41,371 INFO [CollectionFactory] Using JDK 1.4 collections
19:18:41,441 INFO [DispatcherServlet] Using context class
[org.springframework.web.context.support.XmlWebApplicationContext] for
servlet 'remote'
19:18:41,441 INFO [DispatcherServlet] Published WebApplicationContext of
servlet 'remote' as ServletContext attribute with name
[org.springframework.web.servlet.FrameworkServlet.CONTEXT.remote]
19:18:41,441 INFO [DispatcherServlet] Unable to locate MultipartResolver with
name [multipartResolver]: no multipart handling provided
19:18:41,451 INFO [DispatcherServlet] Unable to locate LocaleResolver with
```

```

name 'localeResolver': using default
[org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver@14af469]
19:18:41,481 INFO [DispatcherServlet] Unable to locate ThemeResolver with name
'themeResolver': using default
[org.springframework.web.servlet.theme.FixedThemeResolver@7abaab]
19:18:41,521 INFO [BeanNameUrlHandlerMapping] Mapped URL path
[/logPersonBurlap] onto handler
[org.springframework.remoting.caucho.BurlapServiceExporter@6f3870]
19:18:41,521 INFO [DispatcherServlet] No HandlerMappings found in servlet
'remote': using default
19:18:41,531 INFO [DispatcherServlet] No HandlerAdapters found in servlet
'remote': using default
19:18:41,571 INFO [DispatcherServlet] No ViewResolvers found in servlet
'remote': using default
19:18:41,591 INFO [DispatcherServlet] Framework servlet 'remote' init
completed in 491 ms
19:18:41,591 INFO [DispatcherServlet] Servlet 'remote' configured
successfully
19:19:12,555 INFO [LogPerson] getPersion().....

```

其中, Burlap 使能服务使用到的相关类见图 13-9。

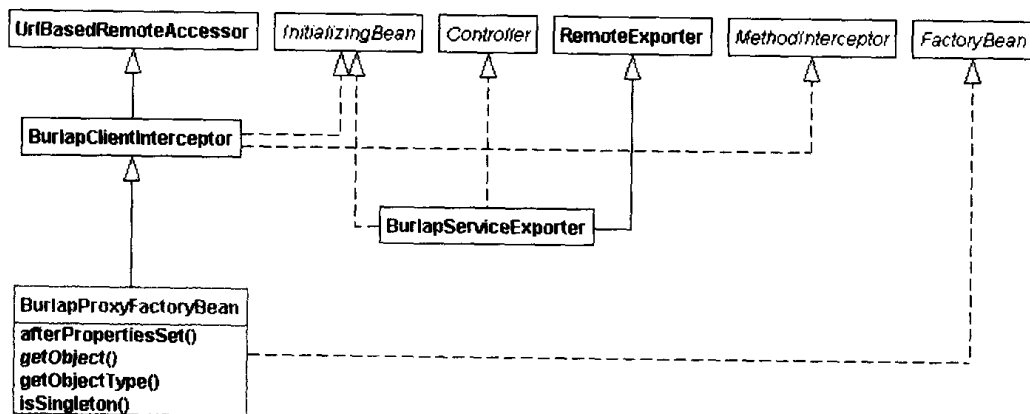


图 13-9 提供 Burlap 使能服务的类图

13.2.4 HTTP Invoker 使能服务

RMI 不适合在受网络约束条件（比如，不存在防火墙）下使用，而且客户和服务端必须同时使用 Java；Burlap 和 Hessian 由于其序列化机制是专属的，即不是 Java 本身提供的序列化机制，在一些使用场合受到限制。因此，它们都存在一定的局限性。是否还有更优秀的解决方案呢？

Spring 提供的原生 HTTP Invoker 就是这种使能服务。HTTP Invoker 不仅能够在防火墙约束下使用（借助于 HTTP 协议实现），还能够使用 Java 提供的序列化机制。它秉承了 RMI 和 Hessian/Burlap 的优点。由于 HTTP Invoker 是基于 HTTP 协议运行的，因此使用 HTTP Invoker 使能服务同 Hessian/Burlap 类似。

本节将结合 example27 对 Spring HTTP Invoker 使能服务进行研究。由于 example27 架构在 example26 基础之上，因此本节只是给出其差异性。其中，example27 使用到的 jar 库见图 13-10。

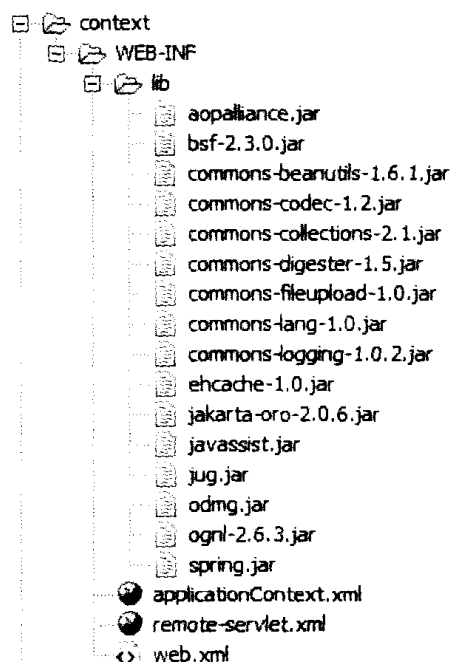


图 13-10 example27.war 使用到的 jar 库

首先，开发者需要提供 remote-servlet.xml。依据 Hessian 或 Burlap 使能服务的机理，开发者只需要将 Spring 提供的 HttpInvokerServiceExporter 替换掉。BurlapServiceExporter 或者 HessianServiceExporter，通过 HttpInvokerServiceExporter 能够导出 HTTP Invoker 使能服务。

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

    <bean name="/logPersonHttpInvoker"

        class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
        <property name="service">
            <ref bean="logPerson"/>
        </property>
        <property name="serviceInterface">
            <value>com.openv.spring.ILogPerson</value>
        </property>
    </bean>

</beans>
```

其次，开发者还需要提供 Spring 客户应用（LogPersonHttpInvokerClient.java）使用的 Spring 配置文件，即 appcontexthttpinvokerclient.xml。HttpInvokerProxyFactoryBean 能够代

理客户对 HTTP Invoker 使能服务的调用请求，这同 RMIProxyFactoryBean 类似。通过 serviceUrl 属性能够指定 HTTP Invoker 使能服务的具体位置信息；通过 serviceInterface 属性能够指定业务接口。

```
<?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="LogPerson"

        class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
        <property name="serviceUrl">
            <value>
                http://localhost:8080/example27/remote/logPersonHttpInvoker
            </value>
        </property>
        <property name="serviceInterface">
            <value>com.openv.spring.ILogPerson</value>
        </property>
    </bean>

</beans>
```

注意，如果客户需要传递用户 Principal 和凭证信息给服务器（比如用户需要访问一些受保护的资源），则使用默认的 HttpInvoker 代理工厂 Bean 是不能够满足客户要求的。默认时，Spring 框架提供了两种 HttpInvokerRequestExecutor 实现。一种只是简单地实现了客户请求的执行，即 SimpleHttpInvokerRequestExecutor。这就是 HttpInvokerProxyFactoryBean 默认使用的。另一种由 Apache Commons HttpClient 提供的，Spring 对其进行了封装，即 CommonsHttpInvokerRequestExcutor。开发者只需要为 HttpInvokerProxyFactoryBean 配置 httpInvokerRequestExecutor 属性值为 CommonsHttpInvokerRequestExcutor 的内容即可满足客户要求。

最后，还需要给出 Spring 客户应用。开发者可以看出，这同 RMI、Hessian、Burlap 类似。再次让开发者体会到 Spring 框架的优美性、极具“艺术性”。

```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;
```

```
/**
 * LogPerson Http Invoker 客户应用
 *
 * @author luoshifei
 */
public class LogPersonHttpInvokerClient {
    protected static final Log log =
        LogFactory.getLog(LogPersonHttpInvokerClient.class);

    public static void main(String[] args) {
        //初始化 appcontexthttpinvokerclient.xml
        Resource cresource = new ClassPathResource("appcontexthttpinvokerclient.xml");
        BeanFactory cfactory = new XmlBeanFactory(cresource);

        //实例化 Person 值对象
        PersonVO personVO = new PersonVO();
        personVO.setFirstname("Luo");
        personVO.setLastname("Shifei");

        //获得 HTTP Invoker 服务
        ILogPerson clientLog = (ILogPerson) cfactory.getBean("LogPerson");

        //调用 HTTP Invoker 服务
        log.info(clientLog.getPersion(personVO));
    }
}
```

在执行 **LogPersonHttpInvokerClient** 应用后, 开发者通过 **JBoss 4.0.0** 控制台可以获得如下输出信息。

```
19:45:00,835 INFO [TomcatDeployer] deploy, ctxPath=/example27, warUrl=file:/
D:/jboss-4.0.0/server/default/tmp/deploy/tmp7276example27-exp.war/
19:45:01,927 INFO [Engine] StandardContext[/example27]Loading root
WebApplicationContext
19:45:02,428 INFO [XmlBeanDefinitionReader] Loading XML bean definitions from
resource [/WEB-INF/applicationContext.xml] of ServletContext
19:45:02,908 INFO [XmlWebApplicationContext] Bean factory for application
context [Root XmlWebApplicationContext]: org.springframework.beans.factory.
support.DefaultListableBeanFactory defining beans [logPerson]; root of
BeanFactory hierarchy
19:45:02,998 INFO [XmlWebApplicationContext] 1 beans defined in application
context [Root XmlWebApplicationContext]
19:45:03,028 INFO [XmlWebApplicationContext] No message source found for
context [Root XmlWebApplicationContext]: using empty default
19:45:03,098 INFO [XmlWebApplicationContext] No ApplicationEventMulticaster
found for context [Root XmlWebApplicationContext]: using default
19:45:03,129 INFO [UiApplicationContextUtils] No ThemeSource found for [Root
XmlWebApplicationContext]: using ResourceBundleThemeSource
```

```
19:45:03,149 INFO [XmlWebApplicationContext] Refreshing listeners
19:45:03,159 INFO [DefaultListableBeanFactory] Pre-instantiating singletons
in factory [org.springframework.beans.factory.support.
DefaultListableBeanFactory defining beans [logPerson]; root of BeanFactory
hierarchy]
19:45:03,159 INFO [DefaultListableBeanFactory] Creating shared instance of
singleton bean 'logPerson'
19:45:03,479 INFO [ContextLoader] Using context class [org.springframework.
web.context.support.XmlWebApplicationContext] for root WebApplicationContext
19:45:03,479 INFO [ContextLoader] Published root WebApplicationContext
[org.springframework.web.context.support.XmlWebApplicationContext:
displayName=[Root XmlWebApplicationContext]; startupdate=[Sun Nov 14 19:45:02
CST 2004]; root of ApplicationContext hierarchy; config locations=[/WEB-INF/
applicationContext.xml]; ] as ServletContext attribute with name [interface
org.springframework.web.context.WebApplicationContext.ROOT]
19:45:03,579 INFO [DispatcherServlet] Initializing servlet 'remote'
19:45:03,709 INFO [DispatcherServlet] Framework servlet 'remote' init
19:45:03,709 INFO [Engine] StandardContext[/example27]Loading
WebApplicationContext for servlet 'remote'
19:45:03,719 INFO [DispatcherServlet] Servlet with name 'remote' will try to
create custom WebApplicationContext context of class 'org.springframework.web.
context.support.XmlWebApplicationContext' using parent context
[org.springframework.web.context.support.XmlWebApplicationContext:
displayName=[Root XmlWebApplicationContext]; startupdate=[Sun Nov 14 19:45:02
CST 2004]; root of ApplicationContext hierarchy; config
locations=[/WEB-INF/applicationContext.xml]; ]
19:45:03,719 INFO [XmlBeanDefinitionReader] Loading XML bean definitions from
resource [/WEB-INF/remote-servlet.xml] of ServletContext
19:45:03,870 INFO [XmlWebApplicationContext] Bean factory for application
context [XmlWebApplicationContext for namespace 'remote-servlet']:
org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [/logPersonHttpInvoker]; parent: org.springframework.beans.
factory.support.DefaultListableBeanFactory defining beans [logPerson]; root
of BeanFactory hierarchy
19:45:03,870 INFO [XmlWebApplicationContext] 1 beans defined in application
context [XmlWebApplicationContext for namespace 'remote-servlet']
19:45:03,870 INFO [XmlWebApplicationContext] No message source found for
context [XmlWebApplicationContext for namespace 'remote-servlet']: using empty
default
19:45:03,870 INFO [XmlWebApplicationContext] No ApplicationEventMulticaster
found for context [XmlWebApplicationContext for namespace 'remote-servlet']:
using default
19:45:03,870 INFO [UiApplicationContextUtils] No ThemeSource found for
[XmlWebApplicationContext for namespace 'remote-servlet']: using
ResourceBundleThemeSource
19:45:03,870 INFO [XmlWebApplicationContext] Refreshing listeners
```

```
19:45:03,870 INFO [DefaultListableBeanFactory] Pre-instantiating singletons
in factory [org.springframework.beans.factory.support.
DefaultListableBeanFactory defining beans [/logPersonHttpInvoker]; parent:
org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [logPerson]; root of BeanFactory hierarchy]
19:45:03,870 INFO [DefaultListableBeanFactory] Creating shared instance of
singleton bean '/logPersonHttpInvoker'
19:45:04,230 INFO [CollectionFactory] Using JDK 1.4 collections
19:45:04,390 INFO [DispatcherServlet] Using context class [org.springframework.
web.context.support.XmlWebApplicationContext] for servlet 'remote'
19:45:04,390 INFO [DispatcherServlet] Published WebApplicationContext of
servlet 'remote' as ServletContext attribute with name
[org.springframework.web.servlet.FrameworkServlet.CONTEXT.remote]
19:45:04,390 INFO [DispatcherServlet] Unable to locate MultipartResolver with
name [multipartResolver]: no multipart handling provided
19:45:04,430 INFO [DispatcherServlet] Unable to locate LocaleResolver with
name 'localeResolver': using default
[org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver@17dabf4]
19:45:04,450 INFO [DispatcherServlet] Unable to locate ThemeResolver with name
'themeResolver': using default
[org.springframework.web.servlet.theme.FixedThemeResolver@7f30f8]
19:45:04,571 INFO [BeanNameUrlHandlerMapping] Mapped URL path
[/logPersonHttpInvoker] onto handler [org.springframework.remoting.
httpinvoker.HttpInvokerServiceExporter@12768bb]
19:45:04,571 INFO [DispatcherServlet] No HandlerMappings found in servlet
'remote': using default
19:45:04,601 INFO [DispatcherServlet] No HandlerAdapters found in servlet
'remote': using default
19:45:04,711 INFO [DispatcherServlet] No ViewResolvers found in servlet
'remote': using default
19:45:04,711 INFO [DispatcherServlet] Framework servlet 'remote' init
completed in 1002 ms
19:45:04,711 INFO [DispatcherServlet] Servlet 'remote' configured successfully
19:45:48,444 INFO [LogPerson] getPerson().....
```

其中，客户端输出如下结果。通过日志信息，开发者能够确认客户应用确实访问到了Spring提供的HTTP Invoker使能服务，并且LogPerson输出了相应的日志。

```
Buildfile: D:\workspace\example27\build.xml
runclient:
```

```
[java] 2004-12-26 18:12:22 org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
[java] 信息: Loading XML bean definitions from class path resource
[appcontexthttpinvokerclient.xml]
[java] 2004-12-26 18:12:22 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'LogPerson'
[java] 2004-12-26 18:12:22 org.springframework.core.CollectionFactory <clinit>
```

[java] 信息: Using JDK 1.4 collections

[java] 2004-12-26 18:12:23 com.openv.spring.LogPersonHttpInvokerClient main

[java] 信息: Shifei,Luo

BUILD SUCCESSFUL

Total time: 3 seconds

其中, HTTP Invoker 使能服务使用到的相关类见图 13-11。

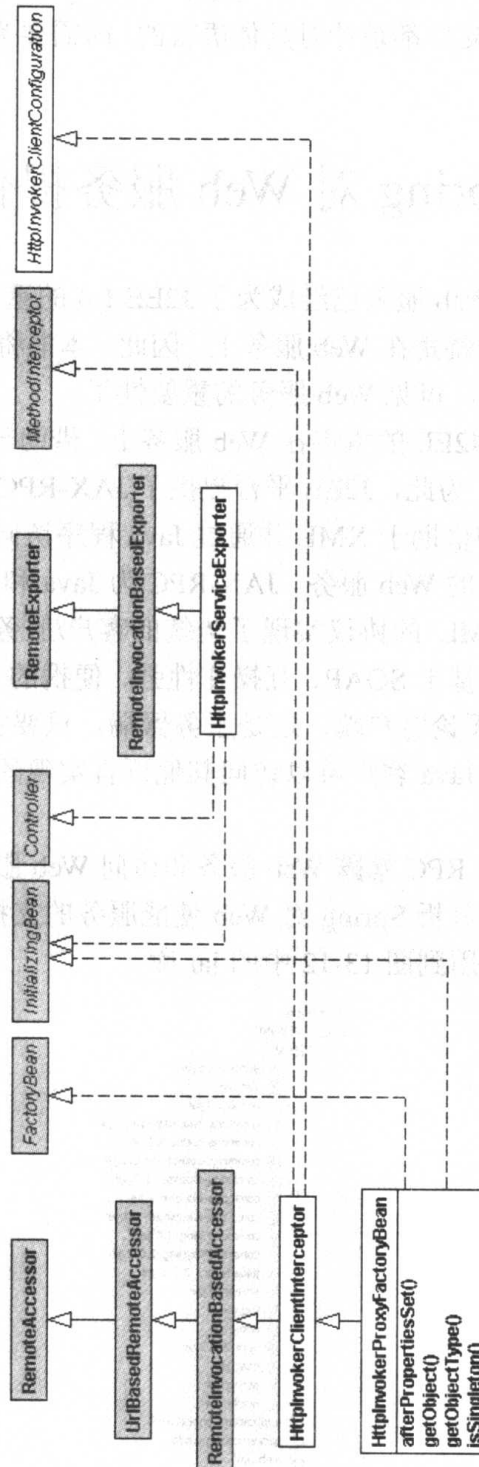


图 13-11 提供 HTTP Invoker 使能服务的类图

借助于 Spring HTTP Invoker 使能服务, 开发者确实体会到了 HTTP 和 Java 序列化机制所带来的优势。它克服了 RMI、Hessian、Burlap 的局限性。但是, 请记住哲学中的一句话, “存在即是合理”。技术本身都有其相应的使用场合, HTTP Invoker 也不例外。开发者在使用 Spring 提供的 HTTP Invoker 使能服务时, 客户和服务器端必须同时是基于 Spring 框架架构的, 而且它们都是基于 Java 编程语言的。尽管 HTTP Invoker 是基于 HTTP 协议的, 但是它还是脱离不了 Java 语言的限制。无论是否 Spring Team 以后提供何种编程语言的 HTTP Invoker 实现, 但这种支持都是针对具体语言的, 即需要为各种语言提供相应的 HTTP Invoker 实现支持。

13.3 Spring 对 Web 服务提供的支持

在 Java/J2EE 平台中, Web 服务已经成为了 J2EE 1.4 的重要组成部分。而且, J2EE 1.4 平台对 J2EE 1.3 的改进重点就是在 Web 服务上。因此, 本书将“Spring 对 Web 服务提供的支持”作为单独的一节研究, 可见 Web 服务的重要性了。

从 J2EE 1.4 平台开始, J2EE 的重点在 Web 服务上。借助于 SOAP/HTTP 开发者能够将应用程序暴露为 Web 服务。为此, J2EE 平台提供了 JAX-RPC (Java APIs for XML-based Remote Procedure Call), 即借助于 XML 并通过 Java 程序访问远程服务。这些远程服务通常都是使用 SOAP 协议暴露的 Web 服务。JAX-RPC 为 Java 和 J2EE 平台提供基于 XML 的 RPC 支持。它利用基于 XML 的协议实现了传统的客户/服务器端的 RPC 机制。借助于 JAX-RPC, 开发者能够开发基于 SOAP、互操作性强、便携的 Web 服务 (实现 Web 服务的具体编程语言不受限制)。无论客户端, 还是服务器端, 只要它们遵循 JAX-RPC 约定, 则它们就能够通畅地沟通, 即 Java 客户可以访问其他语言实现的 Web 服务; 其他客户能够访问 Java 实现的 Web 服务。

Spring 能够借助于 JAX-RPC 暴露 Web 服务和访问 Web 服务提供支持。

本节将结合 example28 分析 Spring 对 Web 使能服务的支持。其中, example28 也是基于 Web 的 J2EE 应用。它使用到图 13-12 中的 jar 库。

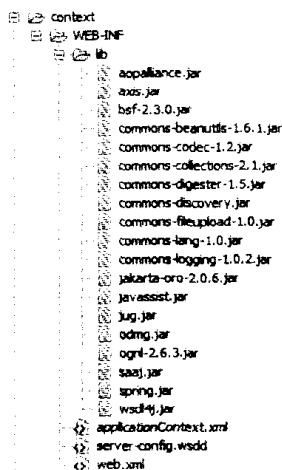


图 13-12 example28.war 使用到的 jar 库

首先, 开发者需要提供 `ILogPerson.java` 接口。`ILogPerson.java` 是业务接口类, 随后将实现它。其中, 定义了单个的业务方法, 即 `getPerson()`。

```
package com.openv.spring;

/**
 * ILogPerson 接口
 *
 * @author luoshifei
 */
public interface ILogPerson {

    public String getPerson(String username);

}
```

其次, 开发者需要提供 `LogPerson.java` 实现。注意, `ILogPerson` 和 `LogPerson` 并没有在方法签名中抛出 `RemoteException`, 这同 `RMI`、`Hessian`、`Burlap`、`HTTP Invoker` 类似。

```
package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * ILogPerson 实现
 *
 * @author luoshifei
 */
public class LogPerson implements ILogPerson {
    protected static final Log log = LogFactory.getLog(LogPerson.class);

    /**
     * 返回个人名字
     *
     * @param username
     *        用户名
     *
     * @return [用户名]
     */
    public String getPerson(String username) {
        log.info("getPersion().....");
        log.info(username);
        return "[" + username + "];"
    }

}
```

第三, 考虑到 JAX-RPC 本身的要求, 开发者需要提供 `ILogPerson` 的 RMI 版本⁴。这同 `ILogPerson` 存在细微的区别。在 `RemoteILogPerson` 中, 继承了 `java.rmi.Remote`, 它在 `getPerson()` 方法签名中抛出了 `RemoteException` 异常。在后续的 `JaxRpcPortProxyFactoryBean` 中, 将 `RemoteILogPerson` 作为 `portInterface` 属性值使用, 并将 `ILogPerson` 作为 `serviceInterface` 属性值使用。一旦有 `RemoteException` 异常抛出, `JaxRpcPortProxyFactoryBean` 创建的代理将捕获它, 这使得 JAX-RPC 客户再也不用处理 `RemoteException` 异常了, 因为在 `ILogPerson` 中并不存在 `RemoteException` 的踪影。

```
package com.openv.spring;

/**
 * RemoteILogPerson 接口
 *
 * @author luoshifei
 */

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteILogPerson extends Remote {

    public String getPerson(String username) throws RemoteException;

}
```

第四, 开发者需要开发 `JaxRpcLogPersonService.java` 类, 从而将 `LogPerson` 以 JAX-RPC 的方式暴露出来。通过继承 Spring 为 JAX-RPC 提供的实用类, 即 `ServletEndpointSupport`, 开发者能够快速实现 Web 服务。注意, `JaxRpcLogPersonService` 同时实现了 `ILogPerson` 和 `RemoteILogPerson` 接口。因此, 在 `JaxRpcLogPersonService` 中实现的 `getPerson()` 方法不用抛出 `RemoteException`。其中, `onInit()` 是回调方法, 在成功初始化上下文后, 会自动调用它。`JaxRpcLogPersonService` 使用 `onInit()` 方法初始化 `logPerson` `JavaBean`。

```
package com.openv.spring;

import org.springframework.remoting.jaxrpc.ServletEndpointSupport;

/**
 * JaxRpcLogPersonService
 *
 * @author luoshifei
 */

public class JaxRpcLogPersonService extends ServletEndpointSupport implements
    RemoteILogPerson, ILogPerson {
```

⁴ 如果使用 JAX-RPC 动态调用, 则可以不提供 RMI 版本。为演示 Web 服务操作细节, 在此还是给出了 RMI 版本。


```

private ILogPerson logPerson;

protected void onInit() {
    this.logPerson = (ILogPerson) getWebApplicationContext().getBean(
        "logPerson");
}

public String getPerson(String username) {
    return this.logPerson.getPerson(username);
}
}

```

第五，提供 web.xml 文件。针对 J2EE 1.4 Web 应用的约定，在此需要提供 web.xml 部署描述符。请注意，在 Burlap、Hessian、HTTP Invoker 等 Web 应用中使用了 DispatcherServlet 实现客户请求的转发，而在本实例中需要借助于 AxisServlet。它是由 Axis 框架提供的实用类。

```

<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">

    <display-name>example28</display-name>

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

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

    <servlet>
        <servlet-name>axis</servlet-name>
        <servlet-class>
            org.apache.axis.transport.http.AxisServlet
        </servlet-class>
        <load-on-startup>4</load-on-startup>
    </servlet>

    <servlet-mapping>

```

```

        <servlet-name>axis</servlet-name>
        <url-pattern>/axis/*</url-pattern>
    </servlet-mapping>

```

```

</web-app>

```

第六，提供 applicationContext.xml 文件，从而实现 LogPerson 的依赖注入。

```

<?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="logPerson" class="com.openv.spring.LogPerson"/>

</beans>

```

第七，提供 Axis 要求的 server-config.wsdd 文件，从而将 LogPerson Web 服务暴露给客户应用。

```

<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
    <globalConfiguration>
        <parameter name="adminPassword" value="admin"/>
        <parameter name="sendXsiTypes" value="true"/>
        <parameter name="sendMultiRefs" value="true"/>
        <parameter name="sendXMLDeclaration" value="true"/>
        <parameter name="axis.sendMinimizedElements" value="true"/>
        <requestFlow>
            <handler type="java:org.apache.axis.handlers.JWSHandler">
                <parameter name="scope" value="session"/>
            </handler>
            <handler type="java:org.apache.axis.handlers.JWSHandler">
                <parameter name="scope" value="request"/>
                <parameter name="extension" value=".jwr"/>
            </handler>
        </requestFlow>
    </globalConfiguration>
    <handler name="Authenticate"
        type="java:org.apache.axis.
            handlers.SimpleAuthenticationHandler"/>
    <handler name="LocalResponder"
        type="java:org.apache.axis.transport.local.LocalResponder"/>
    <handler name="URLMapper"
        type="java:org.apache.axis.handlers.http.URLMapper"/>
    <service name="LogPerson" provider="java:RPC">
        <parameter name="allowedMethods" value="*" />
        <parameter name="className"
            value="com.openv.spring.JaxRpcLogPersonService"/>
    </service>

```

```

</service>
<transport name="http">
    <requestFlow>
        <handler type="URLMapper"/>
        <handler type="java:org.apache.axis.handlers.
            http.HTTPAuthHandler"/>
    </requestFlow>
</transport>
<transport name="local">
    <responseFlow>
        <handler type="LocalResponder"/>
    </responseFlow>
</transport>
</deployment>

```

第八，开发客户应用，即 `LogPersonWebServiceClient`。开发者可能会再次感到惊讶，因为这同 `RMI`、`Hessian`、`Burlap`、`HTTP Invoker` 极其相似，几乎没有任何区别。

```

package com.openv.spring;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

/**
 * LogPersonWebServiceClient 客户应用
 *
 * @author luoshifei
 */
public class LogPersonWebServiceClient {
    protected static final Log log =
        LogFactory.getLog(LogPersonWebServiceClient.class);

    public static void main(String[] args) {
        //初始化 appcontextwsclient.xml
        Resource cresource = new ClassPathResource("appcontextwsclient.xml");
        BeanFactory cfactory = new XmlBeanFactory(cresource);

        //获得 Web 服务
        ILogPerson logPerson = (ILogPerson) cfactory.getBean("jaxRpcProxy");

        //调用 Web 服务
        log.info(logPerson.getPerson("Shifei,Luo"));
    }
}

```

}

}

第九，提供客户使用的 Spring 配置文件，即 `appcontextwsclient.xml`。默认时，`JaxRpcPortProxyFactoryBean` 的 `serviceFactoryClass` 属性值为 `javax.xml.rpc.ServiceFactory`。本实例使用了 Axis 框架实现，因此需要使用 `org.apache.axis.client.ServiceFactory`。另外，开发者还需要指定如下属性：通过 `wsdlDocumentUrl` 属性指定 WSDL 文档的 URL；通过 `namespaceUri`、`serviceName`、`portName` 属性构建 QName。

```
<?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="jaxRpcProxy"

class="org.springframework.remoting.jaxrpc.JaxRpcPortProxyFactoryBean">
        <property name="serviceFactoryClass">
            <value>org.apache.axis.client.ServiceFactory</value>
        </property>
        <property name="wsdlDocumentUrl">
            <value>
                http://localhost:8080/example28/axis/LogPerson?wsdl
            </value>
        </property>
        <property name="namespaceUri">
            <value>http://localhost:8080/example28/axis/LogPerson</value>
        </property>
        <property name="serviceName">
            <value>JaxRpcLogPersonServiceService</value>
        </property>
        <property name="portName">
            <value>LogPerson</value>
        </property>
        <property name="serviceInterface">
            <value>com.openv.spring.ILogPerson</value>
        </property>
        <property name="portInterface">
            <value>com.openv.spring.RemoteILogPerson</value>
        </property>
    </bean>

</beans>
```

第十, 运行 Ant build.xml 中的 deploy 任务, 从而将 example28.war 部署到 JBoss 4.0.0 上。然后, 运行 LogPersonWebServiceClient 客户应用⁵。开发者将浏览到如下输出结果。借助于 JAX-RPC, 实现了对 Web 服务的远程访问。

```
D:\workspace\example28>ant runclient
Buildfile: build.xml

runclient:
[java] 2004-12-26 18:40:59 org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
[java] 信息: Loading XML bean definitions from class path resource
[appcontextwsclient.xml]
[java] 2004-12-26 18:40:59 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
[java] 信息: Creating shared instance of singleton bean 'jaxRpcProxy'
[java] 2004-12-26 18:41:00 org.springframework.remoting.jaxrpc.
JaxRpcPortClientInterceptor afterPropertiesSet
[java] 信息: Using service interface [com.openv.spring.ILogPerson] for
JAX-RPC object [{http://localhost:8080/example28/axis/LogPerson}LogPerson]
- not directly implemented
[java] 2004-12-26 18:41:00 org.springframework.core.CollectionFactory <clinit>
[java] 信息: Using JDK 1.4 collections
[java] 2004-12-26 18:41:01 com.openv.spring.LogPersonWebServiceClient main
[java] 信息: [Shifei,Luo]

BUILD SUCCESSFUL
Total time: 4 seconds
D:\workspace\example28>
```

与此同时, 开发者还可以在 JBoss 控制台看到如下输出信息。通过日志信息, 开发者能够确认 LogPerson Web 服务确实工作了, 而且客户也确实成功调用了它。

```
23:21:08,836 INFO [TomcatDeployer] undeploy, ctxPath=/example28, warUrl=
file:/D:/jboss-4.0.0/server/default/tmp/deploy/tmp63251example28-exp.war/
23:21:08,876 INFO [Engine] StandardContext[/example28]Closing root
WebApplicationContext
23:21:08,876 INFO [XmlWebApplicationContext] Closing application context
[Root XmlWebApplicationContext]
23:21:08,876 INFO [DefaultListableBeanFactory] Destroying singletons in factory
{org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [logPerson]; root of BeanFactory hierarchy}
23:21:08,876 INFO [DefaultListableBeanFactory] Destroying inner beans in
factory
{org.springframework.beans.factory.support.DefaultListableBeanFactory
```

⁵ 请注意, 如果使用 JDK 5.0, 则调用将失败。建议开发者使用 JDK 1.4 运行客户应用。

```
defining beans [logPerson]; root of BeanFactory hierarchy
23:21:09,858 INFO [TomcatDeployer] deploy, ctxPath=/example28, warUrl=
file:/D:/jboss-4.0.0/server/default/tmp/deploy/tmp63253example28-exp.war/
23:21:10,198 INFO [Engine] StandardContext[/example28]Loading root
WebApplicationContext
23:21:10,459 INFO [XmlBeanDefinitionReader] Loading XML bean definitions from
resource [/WEB-INF/applicationContext.xml] of ServletContext
23:21:10,729 INFO [XmlWebApplicationContext] Bean factory for application
context [Root XmlWebApplicationContext]:
org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [logPerson]; root of BeanFactory hierarchy
23:21:10,759 INFO [XmlWebApplicationContext] 1 beans defined in application
context [Root XmlWebApplicationContext]
23:21:10,779 INFO [XmlWebApplicationContext] No message source found for
context [Root XmlWebApplicationContext]: using empty default
23:21:10,799 INFO [XmlWebApplicationContext] No ApplicationEventMulticaster
found for context [Root XmlWebApplicationContext]: using default
23:21:10,819 INFO [UiApplicationContextUtils] No ThemeSource found for [Root
XmlWebApplicationContext]: using ResourceBundleThemeSource
23:21:10,839 INFO [XmlWebApplicationContext] Refreshing listeners
23:21:10,859 INFO [DefaultListableBeanFactory] Pre-instantiating singletons
in factory
[org.springframework.beans.factory.support.DefaultListableBeanFactory
defining beans [logPerson]; root of BeanFactory hierarchy]
23:21:10,859 INFO [DefaultListableBeanFactory] Creating shared instance of
singleton bean 'logPerson'
23:21:11,019 INFO [ContextLoader] Using context class
[org.springframework.web.context.support.XmlWebApplicationContext] for root
WebApplicationContext
23:21:11,019 INFO [ContextLoader] Published root WebApplicationContext
[org.springframework.web.context.support.XmlWebApplicationContext:
displayName=[Root XmlWebApplicationContext]; startup date=[Sun Nov 14 23:21:10
CST 2004]; root of ApplicationContext hierarchy; config locations=
[/WEB-INF/applicationContext.xml]; ] as ServletContext attribute with name
[interface org.springframework.web.context.WebApplicationContext.ROOT]
23:21:21,565 INFO [LogPerson] getPersion().....
23:21:21,565 INFO [LogPerson] Shifei,Luo
```

另外，通过<http://localhost:8080/example28/axis/LogPerson?wsdl>，开发者还能够浏览到 LogPerson Web 服务的 WSDL，见图 13-13。

```

<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions targetNamespace="http://localhost:8080/example28/axis/LogPerson" xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:apacheSOAP="http://xml.apache.org/xml-soap" xmlns:impl="http://localhost:8080/example28/axis/LogPerson"
  xmlns:intf="http://localhost:8080/example28/axis/LogPerson" xmlns:SOAPENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <wsdl:message name="getPersonRequest">
    <wsdl:part name="username" type="xsd:string" />
  </wsdl:message>
  <wsdl:message name="getPersonResponse">
    <wsdl:part name="getPersonReturn" type="xsd:string" />
  </wsdl:message>
  <wsdl:portType name="JaxRpcLogPersonService">
    <wsdl:operation name="getPerson" parameterOrder="username">
      <wsdl:input message="impl:getPersonRequest" name="getPersonRequest" />
      <wsdl:output message="impl:getPersonResponse" name="getPersonResponse" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="LogPersonSoapBinding" type="impl:JaxRpcLogPersonService">
    <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="getPerson">
      <wsdlsoap:operation soapAction="" />
      <wsdl:input name="getPersonRequest">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="http://spring.openv.com"
          use="encoded" />
      </wsdl:input>
      <wsdl:output name="getPersonResponse">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="http://localhost:8080/example28/axis/LogPerson" use="encoded" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="JaxRpcLogPersonServiceService">
    <wsdl:port binding="impl:LogPersonSoapBinding" name="LogPerson">
      <wsdlsoap:address location="http://localhost:8080/example28/axis/LogPerson" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>

```

图 13-13 LogPerson Web 服务的 WSDL

其中，Web 服务使用到的相关类见图 13-14。

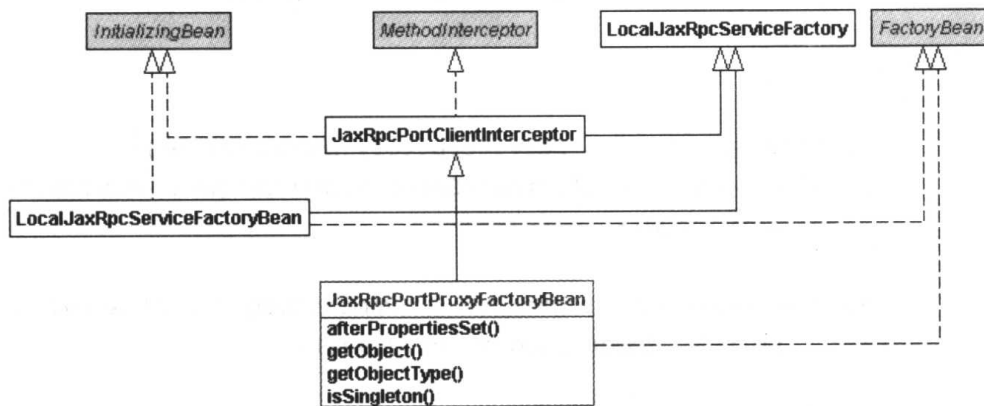


图 13-14 提供 Web 服务的类图

注意，如果在客户端不借助于 Spring 实现 JAX-RPC 调用，则实例代码如下。

```

package com.openv.spring;

import java.net.MalformedURLException;
import java.net.URL;
import java.rmi.RemoteException;

import javax.xml.namespace.QName;
import javax.xml.rpc.Service;
import javax.xml.rpc.ServiceException;
import javax.xml.rpc.ServiceFactory;

```

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * LogPersonWebServiceClientNoSpring 客户应用
 *
 * @author luoshifei
 */
public class LogPersonWebServiceClientNoSpring {
    private static final Log log = LogFactory
        .getLog(LogPersonWebServiceClientNoSpring.class);

    public static void main(String[] args) {
        String wsdlDocumentUrl =
            "http://localhost:8080/example28/axis/LogPerson?wsdl";
        String namespaceUri =
            "http://localhost:8080/example28/axis/LogPerson";
        String serviceName = "JaxRpcLogPersonServiceService";
        String portName = "LogPerson";

        QName serviceQN = new QName(namespaceUri, serviceName);
        QName portQN = new QName(namespaceUri, portName);

        try {

            ServiceFactory sf = ServiceFactory.newInstance();
            Service service = sf.createService(new URL(wsdlDocumentUrl),
                serviceQN);

            RemoteILogPerson iLogPerson = (RemoteILogPerson) service.getPort(
                portQN, RemoteILogPerson.class);

            log.info(iLogPerson.getPerson("Shifei,Luo"));

        } catch (ServiceException se) {
            log.error("ServiceException 异常", se);
        } catch (MalformedURLException mu) {
            log.error("MalformedURLException 异常", mu);
        } catch (RemoteException re) {
            log.error("RemoteException 异常", re);
        }
    }
}
```

可以看出，如果不借助于 Spring 提供的 JAX-RPC 支持，客户端代码量大大增加，而且开发者必须处理 `ServiceException`、`MalformedURLException`、`RemoteException` 等异常类型。

至此，开发者已经成功将 LogPerson 部署为 Web 服务，并且成功调用了它。结合前面阐述的 4 种方式，本章一共介绍了 5 种将一般性的 JavaBean 服务暴露为远程服务和 Web 服务的方式。

借助于 Spring，一切都是这么相似。

13.4 小 结

本章对 Spring 提供的远程服务做了较深入的介绍。在实际应用场合中，开发者需要结合具体的应用需求而选择不同的远程服务。

另外，处于开发中的 JBoss Remoting 子项目 (<http://www.jboss.org/products/remoting>) 将为 JBoss 5.x 应用服务器提供统一的 Invoker 架构，其实现的功能同 Spring 远程服务类似。在 JBoss 3.x/4.x 中，并不存在统一的 Invoker 架构，尽管 JMS、EJB 等模块都存在 Invoker 实现。通过 Spring 远程服务和 JBoss 远程服务，开发者能够看出，在分布式计算领域中，将传输调用协议独立出来，将使得应用的可部署性大大增强，这也是将来的趋势，因此开发者应该重视这方面的内容。尤其是开发者应该花些时间研究它们的具体使用和实现机理。注意，JBoss Remoting 更多地是为 JBoss 应用服务器服务，尽管它也可以单独使用；Spring 远程服务更多地是为开发 J2EE 应用服务。

至此，第二部分内容已经结束。通过这部分内容，开发者已经深深地领悟了 Spring 是如何使得开发者的开发效率得到提高的。Spring 提供的服务抽象使得使用 J2EE API 更为简单。Spring 是非常实用的框架，这对于开发者而言确实是非常实用的。在此，作者建议读者再次回到第一部分内容去，这样将大大加深对 Spring 的掌握程度。

开发专家之

Sun ONE

第三部分 Spring 高级主题

经过第一、二部分内容，开发者已经对 Spring IoC、Spring AOP、J2EE 服务抽象等内容很熟悉了。至此，我们需要对 Web 层进行关注了。无论是何种 Web 视图技术，还是 Web 框架，Spring 能够很好地支持、集成，从而利用到 Spring 提供的基础设施，比如 Spring IoC、Spring AOP、J2EE 服务抽象。因此，视图技术集成（第 14 章）、Tapestry 集成（第 15 章）、JSF 集成（第 16 章），这三章内容成了这部分的核心内容之一。如果 Web 应用中不能够采用好的 Web 视图或框架技术，则效率极低。

安全性往往是企业 Java 应用中最为重要的需求之一。为此，用于 Spring 的 Acegi 安全框架（第 17 章）进入了我们的视野中。Acegi 本身就是 Spring 使能应用。值得高兴的是，使用、研究 Acegi 本身就是一种学习 Spring 的过程。这或许就是双赢。

如果开发者对 Spring IoC、Spring AOP、J2EE 服务抽象还不熟悉，赶紧去研究前两部分内容吧！

第 14 章 视图技术集成

Spring MVC 框架将视图技术独立出来，即开发者可以不使用 JSP 技术作为 Web 应用的前端来展示页面，而换成其他的视图技术。比如，开发者可以使用 Struts、JSF、Tapestry、JSTL、Velocity 以及 FreeMarker 等开发 Web 应用的前端，而整个 Web 应用的架构仍然采用 Spring。这同 Spring 属于架构级的 Web 框架相吻合。

对于具体视图技术而言，通常都会存在相应的 Web 框架支持，比如 Tapestry 除了提供 Web 视图技术外，还提供了支持开发 Web 应用的其他功能。通常，比较这种 Web 框架¹会从如下几方面入手。

- 提供控制器和视图的能力：比如在 Struts 中，继承 `org.apache.struts.action.Action` 类能实现对 JSP 页面的控制；在 Spring Web MVC 中，继承 `SimpleFormController` 类能够实现对 Web 页面的控制；在 WebWork (2) 中，继承 `ActionSupport` 同样能够达到目的；在 Tapestry 中，继承 `BasePage` 能够实现对 Web 页面的控制。
- 支持分页的能力：这对于 Web 应用而言，很常见。其中，Struts、Spring Web MVC、WebWork 能够使用标签库实现分页支持；Tapestry 提供了 `contrib:Table` 组件，以支持分页；JSF 提供了 `dataTable` 组件，以提供分页支持。
- 友好的 URL 支持：这主要是考虑到能否直接使用容器提供的认证和授权。就目前而言，Tapestry 生成的 URL 不太利于 JAAS 的使用。当然，Tapestry 在努力修正这个问题，而且已经有了较好的办法解决（Tapestry 3.1 将解决这个问题）。
- 有效性校验：校验功能始终是应用很重要的内容。其中，有效性校验应该易于配置、能够提供健壮的客户支持。比如，Struts 和 Spring Web MVC 使用业界成熟的 Commons Validator；WebWork 使用 OGNL 库；Tapestry 存在非常完善的有效性校验功能，尤其适合本地化和国际化 Web 应用（在 Tapestry 3.0.2 中新加入了中文的本地化支持）；JSF 的有效性校验易于配置，但是在功能上有待改进。
- 页面的可测试能力：针对 TDD 的开发，尤其是针对现今流行的 XP 开发模式。在 Struts 中，可以使用 `StrutsTestCase`；在 Spring 和 WebWork 中，可以使用 `Mock` 类，比如 `EasyMock` 和 `jMock`。当然，Spring 自身还提供了 `Mock` 类。在 Tapestry 中，由于页面是抽象类，因此对于单元测试而言显得有些困难，但是集成测试不成问题。在 JSF 页面中，测试工作可以很顺利地进行。
- 与 Spring 框架的集成能力：Spring 框架出色的架构能力，使得开发者能够集成各种 Web 视图技术。甚至对于有些视图技术，比如 Tapestry，Spring 不需要为它进行任何专有开发即可在 Tapestry 中使用 Spring 的 IoC 和 AOP 框架功能。

本章将对上述提到的各种视图技术（Web 框架）给出大体的介绍。当然，Tapestry 和 JSF 将在本书的第 15、16 章详细阐述，其他视图技术的介绍只是停留在介绍性的层面上。

¹ <http://equinox.dev.java.net/framework-comparison/WebFrameworks.pdf>

14.1 Spring Web MVC

Spring Web MVC, 同 Struts 类似, 是基于 MVC 的 Web 框架。在 Spring Web MVC 使用应用中, 能够直接使用到 Spring IoC 和 AOP 的功能。借助于它提供的 `DispatcherServlet` 控制器 (见图 14-1, 它类似于 Struts 中的 `org.apache.struts.action.ActionServlet` 控制器), 能够统一分发 Web 请求, 比如来自 Web UI 的 HTTP 请求, 或者 HTTP Invoker 的请求。在整个 Spring Web MVC 中, `DispatcherServlet` 是最为重要的组件之一。直接在 `web.xml` 中配置使用它。

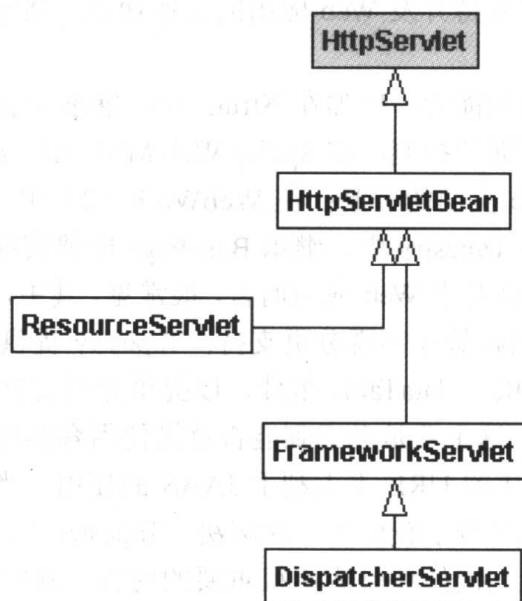


图 14-1 `org.springframework.web.servlet.DispatcherServlet` 前端控制器

通常, Spring Web MVC 处理 HTTP 请求的过程大致如下。

- 一旦客户 HTTP 请求到来, `DispatcherServlet` 将负责分发它。注意, `DispatcherServlet` 可以认为是 Spring 提供的前端控制器²。所有的客户请求都需要经过它的统一分发。
- 在 `DispatcherServlet` 将请求分发给 Spring Controller 之前, 需要借助于 Spring 提供的 `HandlerMapping` 定位到具体的 Controller。注意, `HandlerMapping` 是这样一种对象, 即它能够完成客户请求到 Controller 对象之间的映射。在 Struts 中, 开发者也需要在 `struts-config.xml` 配置文件中完成这种映射。其中, Spring 为 Controller (位于 `org.springframework.web.servlet` 包中) 接口提供了若干实现。比如, Spring 默认使用的 `BeanNameUrlHandlerMapping` (本书在第 13 章曾使用过)。当然, 还有 `SimpleUrlHanlderMapping`、`CommonsPathMapHandlerMapping` 等。
- Spring Controller 将处理来自 `DispatcherServlet` 的请求。注意, Controller 同 Struts

² <http://www.corej2eepatterns.com/index.htm>

中的 Action 类似, 即能够接受 `HttpServletRequest` 和 `HttpServletResponse`。Spring 提供了 Controller 接口 (位于 `org.springframework.web.servlet.mvc` 包中) 的若干实现 (抽象类)。由于 Controller 需要为并发用户处理上述请求, 因此开发者在实现 Controller 接口时, 必须保证它们是线程安全的而且可重用。Controller 将处理客户请求, 这同 Struts Action 扮演的角色是一致的。因此, Struts 同 Spring Web MVC 在很多方面很类似, 但是 Spring Web MVC 更灵活。如果开发者需要定制 Spring Web MVC, 则比定制 Struts 更容易、方便。从这里也可以看出, Spring 是为开发者提供的架构级框架。当然, 业务逻辑的处理不一定需要 Controller 来完成, 比如可以直接委派给 Spring IoC 容器中的受管 `JavaBean` 完成, 或者其他 EJB 分布式对象来完成。无论如何, Controller 可以满足各种应用架构需求。

- 一旦 Controller 处理完客户请求, 则返回 `ModelAndView` 对象给 `DispatcherServlet` 前端控制器。`ModelAndView` 中包含了模型 (Model) 和视图 (View)。从宏观角度考虑, `DispatcherServlet` 是整个 Web 应用的控制器; 从微观角度考虑, Controller 是单个 HTTP 请求处理过程中的控制器, 而 `ModelAndView` 是处理 HTTP 请求过程中返回的模型和视图。通常, 前端控制器 `DispatcherServlet` 返回的视图可以是视图的逻辑名, 或者实现了 View 接口 (位于 `org.springframework.web.servlet` 包中) 的对象。View 对象能够渲染客户响应结果。其中, `ModelAndView` 中的模型能够供渲染 View 时使用。借助于 Map 对象 (通过其中存储的值对中的名字标识具体的模型) 能够存储模型。
- 如果上述 `ModelAndView` 返回的视图只是逻辑名, 则需要借助于 Spring 提供的视图解析器 (`ViewResolver`) 在 Web 应用中查找 View 对象, 从而将响应结果渲染给客户。
- 最终, `DispatcherServlet` 将 View 对象渲染出的结果返回给客户。从而, 宣告整个 HTTP 请求-响应过程的终止。

为进一步理解 Spring Web MVC, 我们来结合 Spring 框架实现的 JPetStore 进行。注意, JPetStore 是 Spring 附带的经典实例。为学习、研究 Spring, 从 JPetStore 入手往往能够达到事半功倍的效果。本书也推荐开发者在研究 Spring 的过程中去认真地研究它。每次新版本的 Spring 发布, 对该实例都有不同程度的修改, 以同步 Spring 提供的最新功能。而且, 这种同步是不需要 Spring 开发者参与的, 何乐而不为呢?³

14.1.1 配置 DispatcherServlet

首先, 需要配置 `DispatcherServlet` 前端控制器。开发者在打开 `web.xml`⁴后, 能够浏览到 `DispatcherServlet` 的具体配置内容。因为它是 Servlet, 因此需要同时配置 `<servlet>` 和 `<servlet-mapping>`。由于 JPetStore 的 Web 展示层同时实现了基于 Struts 和 Spring Web MVC 的前台界面, 而且默认时是基于 Struts 配置的, 因此开发者需要注意该细节。注意, 有关

³ <http://www.open-v.com> 提供了《Spring—内容与形式并重》一文, 值得开发者阅读。

⁴ 位于 `spring-framework-1.1.3\samples\jpetstore\war\WEB-INF` 目录下。依据 Spring 发布版本的不同, `spring-framework-1.1.3` 路径名可能需要作相应的调整。

该应用实例中使用 Struts 的相关细节将在第 14.2 节研究，到时请注意。

```
<servlet>
    <servlet-name>petstore</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>2</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>petstore</servlet-name>
    <!--
    <servlet-name>action</servlet-name>
    -->
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

注意，在 web.xml 中同时还配置了另一个 DispatcherServlet，见如下配置片断。由于在该实例中使用到 Spring 远程服务（本书第 13 章详细研究过），因此还需配置这样的前端控制器。上述配置片断将通过 petstore-servlet.xml 文件定位 Spring Web MVC 使用的 DispatcherServlet 应用上下文；下面的配置片断将通过 remotingservlet.xml 文件定位 Spring 远程服务使用到的 DispatcherServlet 应用上下文（比如，Hessian、Burlap、HTTP Invoker）。开发者应该注意到，借助于<servlet-name>能够定位到具体的 XML 文件。这是 Spring 约定的配置规则。当然，本节的重点在于 Spring Web MVC，因此不考虑 remotingservlet.xml 文件的具体细节。如果开发者对 Spring 远程服务感兴趣，则可以参考第 13 章内容。

```
<servlet>
    <servlet-name>remoting</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>4</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>remoting</servlet-name>
    <url-pattern>/remoting/*</url-pattern>
</servlet-mapping>
```

14.1.2 开发及配置 Controller

其次，需要开发 Controller 接口的实现类（同 Struts Action 类似）。但是，尽管 Action 同 Controller 类似，但还是存在很大区别，尤其是对 Spring IoC 提供的受管 JavaBean（位于 ApplicationContext）的使用。在 Spring Web MVC 中，已经内置了对 ApplicationContext 的

引用支持（借助于 `ApplicationContextAware` 接口实现，本书在第 3 章详细研究过该接口及 Spring IoC 的具体细节），因此如果开发者使用 Web MVC 框架开发 Web 应用，本书推荐使用 Spring Web MVC（相比 Struts 而言）。为支持 Struts 中引用 Spring IoC 容器的功能，还需对它进行专有集成工作（第 14.2 节将阐述到这方面的详细内容）。在 Spring JPetStore 应用的 `petstore-servlet.xml` 中，存在如下配置片断。

```
<bean name="/shop/newAccount.do"
      class="org.springframework.samples.jpetsotre.
        web.spring.AccountFormController">
  <property name="petStore">
    <ref bean="petStore"/>
  </property>
  <property name="validator">
    <ref bean="accountValidator"/>
  </property>
  <property name="successView">
    <value>index</value>
  </property>
</bean>
```

其中，`AccountFormController` 引用了 Spring IoC 容器中的 `petStore` 和 `validator` 受管 JavaBean 服务。注意，`validator` 是校验器⁵。

`AccountFormController` 继承 `org.springframework.web.servlet.mvc.SimpleFormController`，Spring 为 Web 应用提供了若干 Controller 接口实现，见图 14-2 所示（各个 Controller 实现的具体含义及使用请参考 Spring API JavaDoc 文档）。

为实现对 `petStore` 受管 JavaBean 的引用，需要在 `AccountFormController.java` 中实现如下内容。

```
private PetStoreFacade petStore;

public void setPetStore(PetStoreFacade petStore) {
  this.petStore = petStore;
}
```

另外，开发者是否注意到，`successView` 属性（定义在 `SimpleFormController` 中）的取值为 `index`。其中，`successView` 属性的含义为，如果 `AccountFormController` 成功处理客户请求，则将返回相应的视图（View）给客户。很显然，`index` 是逻辑视图名。借助于 Spring `petstore-servlet.xml` 配置文件中的如下片断能够解析它。

```
<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">
```

⁵ 有关 Spring Web MVC 校验器的具体内容请开发者参考 Spring 官方指南和 Spring 框架源码。

```

        <value>/WEB-INF/jsp/spring/</value>
    </property>
    <property name="suffix">
        <value>.jsp</value>
    </property>
</bean>

```

很显然，逻辑视图名 index 将被解析成 “/WEB-INF/jsp/spring/index.jsp”。

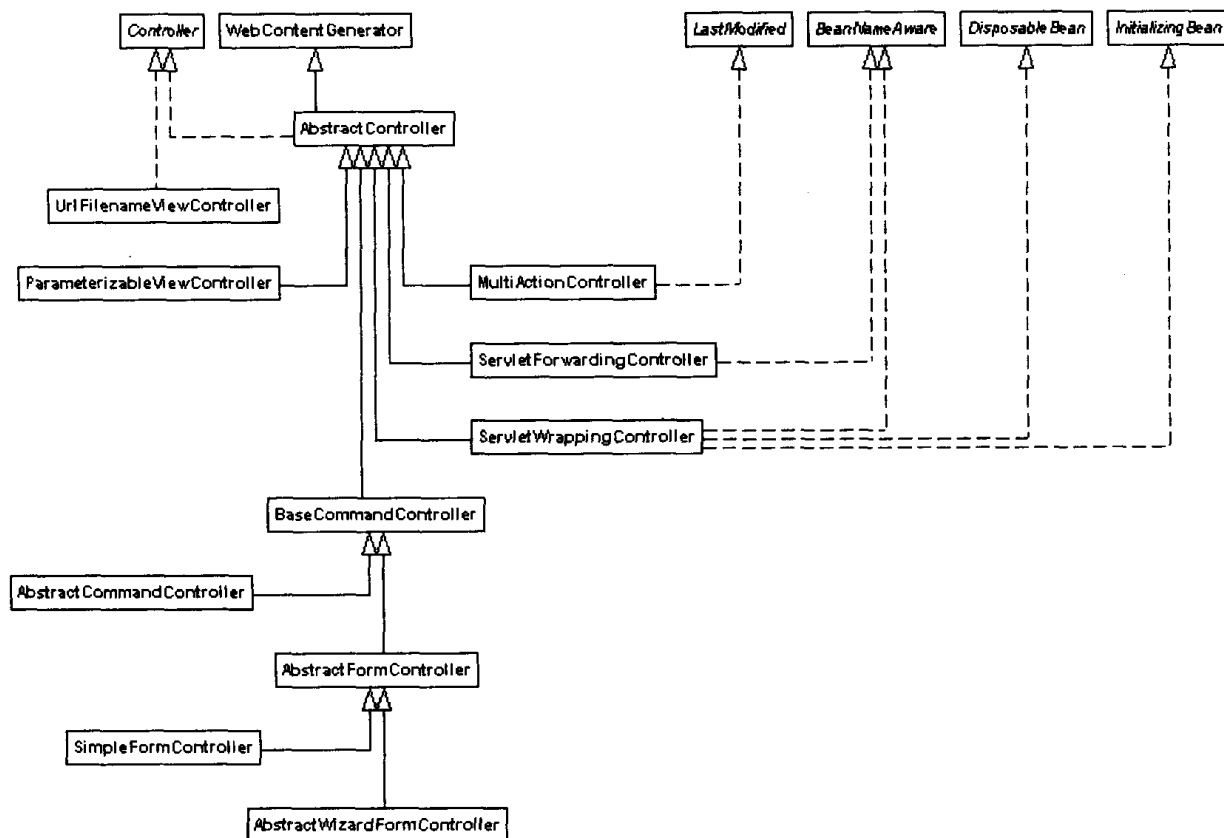


图 14-2 Controller 接口及其实现

14.1.3 配置 ViewResolver

为解析 index，上述配置使用了 InternalResourceViewResolver 视图解析器。通常，Spring Controller 将返回 ModelAndView 对象。比如，AccountFormController 的 onSubmit() 方法返回了 ModelAndView 对象，见如下方法签名。

```

protected ModelAndView onSubmit(HttpServletRequest request,
    HttpServletResponse response, Object command, BindException errors)
    throws Exception {

```

视图 (View) 是能够渲染服务器响应结果的对象。然而，在渲染 View 对象之前，需要借助于 ViewResolver 获得 Spring 配置文件中配置的 View 对象。注意，ViewResolver 接口位于 org.springframework.web.servlet 包中。该接口仅含有单个方法，见如下。

```

View resolveViewName(String viewName, Locale locale) throws Exception;

```

Spring 为 ViewResolver 提供了若干实现。图 14-3 给出了部分 ViewResolver 实现。在

Spring JPetStore 应用实例中, 配置使用了 InternalResourceViewResolver 视图解析器。

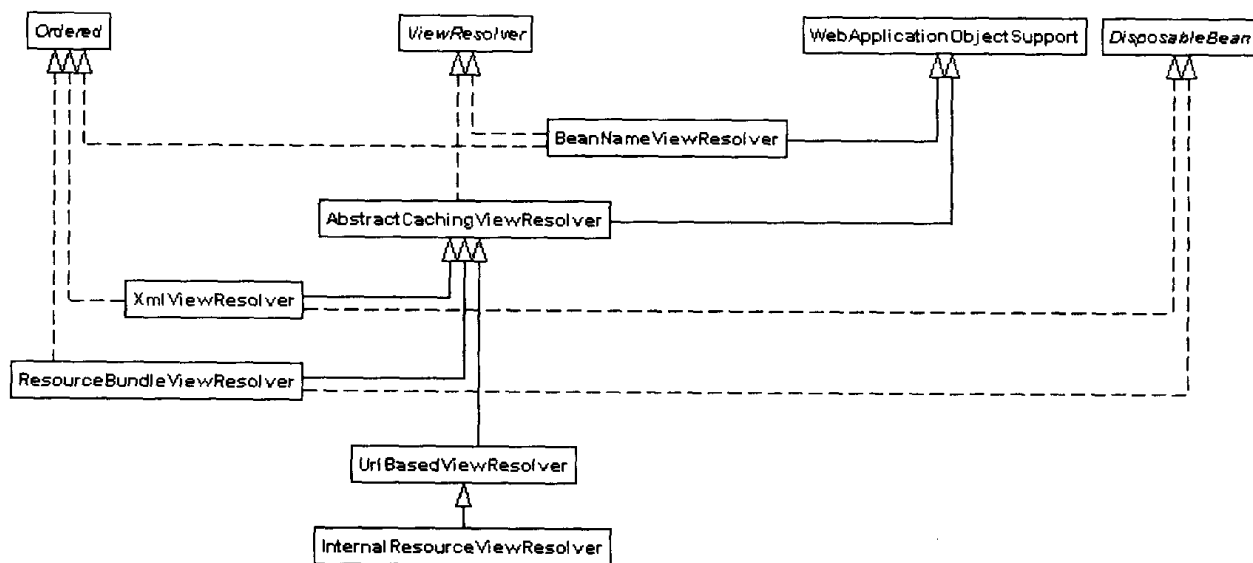


图 14-3 ViewResolver 接口及其实现

这些 ViewResolver 接口实现的具体含义如下。

- **BeanNameViewResolver**: 将逻辑视图名解析成 Spring ApplicationContext 中(注意, DispatcherServlet 前端控制器处于其中)。配置的 View 受管 JavaBean。
- **XmlViewResolver**: 含义同 BeanNameViewResolver。但是, XmlViewResolver 将会在单独的 XML 配置文件中查找 View 受管 JavaBean。如果 Web 应用存在大量的 View 受管 JavaBean, 则建议使用 XmlViewResolver。因此, 这将使得各个 Spring 配置文件定义的内容能够很好地分割, 并便于维护。默认时, 它会在 WEB-INF 目录中查找 views.xml 文件。当然, 开发者可以借助于 XmlViewResolver 的 location 属性修改其默认行为, 比如 “/WEB-INF/example11-views.xml”。
- **ResourceBundlerViewResolver**: 将逻辑视图名解析成属性文件 (.properties) 中定义的 View 对象。注意, 基于属性文件, 即通过 java.util.Locale 解析 View 对象能够较好地实现 Web 应用的国际化。Spring Web MVC 会基于客户的 Locale 而自动在相应的属性文件中查找 View 对象。如果借助于其他视图解析器, 则实现 View 对象的国际化相对困难些。默认时, 它使用 views.properties 文件。如果需要对简体中文提供 View 对象, 则还需提供相应的 views_zh_CN.properties 文件。最终, 通过在不同属性文件中定义各自所属 Locale 的 View 对象, 便能够实现国际化版本。
- **InternalResourceViewResolver**: 将逻辑视图名解析成 View 对象。这种 View 对象是能够借助于模板文件资源进行渲染的, 比如 JPetStore 使用的模板文件是 .jsp 文件。因此, 在该应用中使用了 InternalResourceViewResolver 视图解析器。当然, 开发者也可以采用其他模板技术, 比如 Velocity、FreeMarker (第 14.6 节阐述)。

14.1.4 配置 HandlerMapping

最后，开发者需要配置 HandlerMapping 接口实现。其中，HandlerMapping 接口位于 org.springframework.web.servlet 包中。注意，HandlerMapping 是这样一种对象，即它能够完成客户请求到 Controller 对象之间的映射。Spring Web MVC 提供了若干 HandlerMapping 接口实现，见图 14-4。

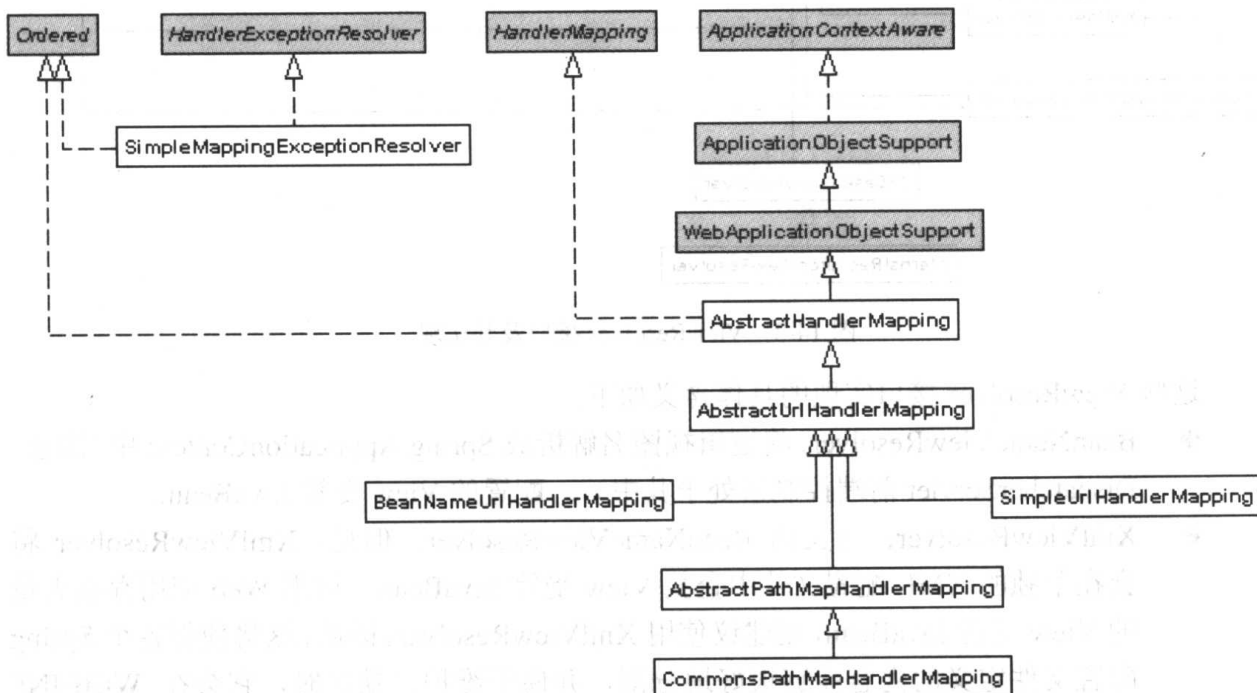


图 14-4 HandlerMapping 接口及其实现

上述 HandlerMapping 接口实现的含义如下。

- **BeanNameUrlHandlerMapping**: 直接借助于 Spring 配置文件中 Controller 的名字来实现 URL 映射。默认时，DispatcherServlet 将使用该 HandlerMapping 实现。比如在 Spring JPetStore 应用中，配置使用了它。当然，即使不显式指定它，Spring 默认也会使用它，只不过显式给出便于开发者阅读罢了。

```

<bean id="defaultHandlerMapping"
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

```

上述片断给出了 BeanNameUrlHandlerMapping 的定义。如果在应用中，出现了 “/shop/index.do” 请求，则将直接寻找名字为 “/shop/index.do” 的 Controller 实现。因此，最终定位到如下 Controller 实现。从而，完成了 URL 到 Controller 的映射。

```

<bean name="/shop/index.do" class="org.springframework.
web.servlet.mvc.ParameterizableViewController">
    <property name="viewName">
        <value>index</value>
    </property>
</bean>

```

```

    </property>
</bean>

```

- **SimpleUrlHandlerMapping**: 直接在 Controller 定义中指定带有 “/” 的名字总给人不协调的感觉。Controller 作为受管的 JavaBean 对象, 则在 Spring 中给它定义的名字最好是有意义的名字, 比如 “parameterizableViewController”。因此, Spring Web MVC 引入了 SimpleUrlHandlerMapping 实现。比如下面给出了 JPetStore 中配置的片断。

```

<bean id="secureHandlerMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="interceptors">
        <list>
            <ref bean="signonInterceptor"/>
        </list>
    </property>
    <property name="urlMap">
        <map>
            <entry key="/shop/editAccount.do">
                <ref local="secure_editAccount"/>
            </entry>
            <entry key="/shop/listOrders.do">
                <ref local="secure_listOrders"/>
            </entry>
            <entry key="/shop/newOrder.do">
                <ref local="secure_newOrder"/>
            </entry>
            <entry key="/shop/viewOrder.do">
                <ref local="secure_viewOrder"/>
            </entry>
        </map>
    </property>
</bean>

```

很显然, 通过<entry>中的 key 指定 URL 模式, 然后借助于 Spring <ref>引用相应的 Controller 定义。最终, 实现了同 BeanNameUrlHandlerMapping 类似的功能。

- **CommonsPathMapHandlerMapping**: 通过元数据实现 URL 到 Controller 的映射。本书并没有对元数据进行过多的分析和研究, 感兴趣的读者可以参考 Spring 官方指南。

14.2 Struts

Struts⁶是由 Open Source 社区开发的, 是基于 MVC 的, 它实现了 Model 2 模型的 Web 应用框架。Struts 框架的核心是其提供的灵活的控制层。其中, Struts 是基于 Java Servlet、

⁶ <http://struts.apache.org>

JavaBean、ResourceBundle、XML 以及 Jakarta Commons 等技术构建的。

在 Struts 提供控制层的前提下，开发者可以很容易集成其他技术。对于模型层而言，Struts 框架可以同 JDBC、EJB、Spring IoC、Hibernate、iBATIS、TopLink 等集成。对于视图层而言，Struts 框架可以同 JSP、JSTL、JSF、Velocity 模板、XSLT、Spring Web MVC 等视图技术集成。

因此，Struts 能够同现有的各种 Web 技术和非 Web 技术进行集成。

14.2.1 Spring JPetStore 的 ApplicationContext 集成方式

为了在 Struts 版本的 Spring JPetStore 中使用到 Spring IoC 和 AOP 功能，JPetStore 专门提供了抽象 `org.springframework.samples.jpetsyore.web.struts.BaseAction` 类，以集成 Spring ApplicationContext。具体代码如下。可以看到，借助于 Struts 中的 `ActionServlet` 前端控制器能够获得 `ServletContext`，进而借助于 `WebApplicationContextUtils` 获得 `ApplicationContext`。注意，`petStore` 为 `TransactionProxyFactoryBean`，其中包裹了 `PetStoreImpl`。最终，应用能够访问到 JPetStore 的业务逻辑，并使用到 Spring IoC 和 AOP 的功能。

```
public abstract class BaseAction extends Action {

    private PetStoreFacade petStore;

    public void setServlet(ActionServlet actionServlet) {
        super.setServlet(actionServlet);
        if (actionServlet != null) {
            ServletContext servletContext =
                actionServlet.getServletContext();
            WebApplicationContext wac = WebApplicationContextUtils
                .getRequiredWebApplicationContext(servletContext);
            this.petStore = (PetStoreFacade) wac.getBean("petStore");
        }
    }

    protected PetStoreFacade getPetStore() {
        return petStore;
    }
}
```

另外，`petStore` Spring 受管 JavaBean 服务的定义如下。借助于抽象受管 JavaBean，并在内部受管 JavaBean 的前提下，应用通过定义的 `petStore` 能够访问到整个 JPetStore 应用中的业务逻辑。

```
<bean id="baseTransactionProxy"
      class="org.springframework.transaction.
          interceptor.TransactionProxyFactoryBean"
      abstract="true">
    <property name="transactionManager">
```

```

        <ref bean="transactionManager"/>
    </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>

<bean id="petStore" parent="baseTransactionProxy">
    <property name="target">
        <bean class="org.springframework.samples.
            jpetstore.domain.logic.PetStoreImpl">
            <property name="accountDao">
                <ref bean="accountDao"/></property>
            <property name="categoryDao">
                <ref bean="categoryDao"/></property>
            <property name="productDao">
                <ref bean="productDao"/></property>
            <property name="itemDao">
                <ref bean="itemDao"/></property>
            <property name="orderDao">
                <ref bean="orderDao"/></property>
        </bean>
    </property>
</bean>

```

比如，在 Spring JPetStore 应用实例的 Struts ViewOrderAction.java 中（位于 org.springframework.samples.jpetsotre.web.struts 包），存在如下方法。显然，借助于 getPetStore() 方法能够访问到业务逻辑，而这里的 petStore 实例正是上述定义的受管 JavaBean。

```

protectedActionForwarddoExecute(ActionMappingmapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response)
    throws Exception {
    AccountActionForm acctForm = (AccountActionForm) form;
    int orderId = Integer.parseInt(request.getParameter("orderId"));
    Order order = getPetStore().getOrder(orderId);
    if (acctForm.getAccount().getUsername().
        equals(order.getUsername())) {
        request.setAttribute("order", order);
        return mapping.findForward("success");
    } else {
        request.setAttribute("message",
            "You may only view your own orders.");
        return mapping.findForward("failure");
    }
}

```

}

注意，上述集成方式并未使用到 Spring 提供的、任何对 Struts 的专属集成工作。Web 容器，比如 Tomcat，在初始化 Spring 使能 Web 应用期间，Spring ApplicationContext 将会注册到 ServletContext 中，因此借助于 ActionServlet 访问到的 ServletContext 能够获得 Spring 中的 ApplicationContext。

14.2.2 Spring 提供的集成方式

开发者需要注意的是，在 JPetStore 应用实例中，无论使用 Spring Web MVC，还是使用 Struts 技术作为前端，都是通过 org.springframework.web.context.ContextLoaderServlet 来初始化 Spring ApplicationContext 的（见 web.xml 部署描述符）。能够借助于其他方式初始化 Spring ApplicationContext 吗？答案是可以借助于 Spring 提供的 Struts 集成技术。具体细节请仔细阅读随后的内容。

Spring 为了集成 Struts，专门提供了 org.springframework.web.struts 包。比如，ActionSupport 在集成 Spring 上下文起到了很重要的作用，其提供的支持类见图 14-5。

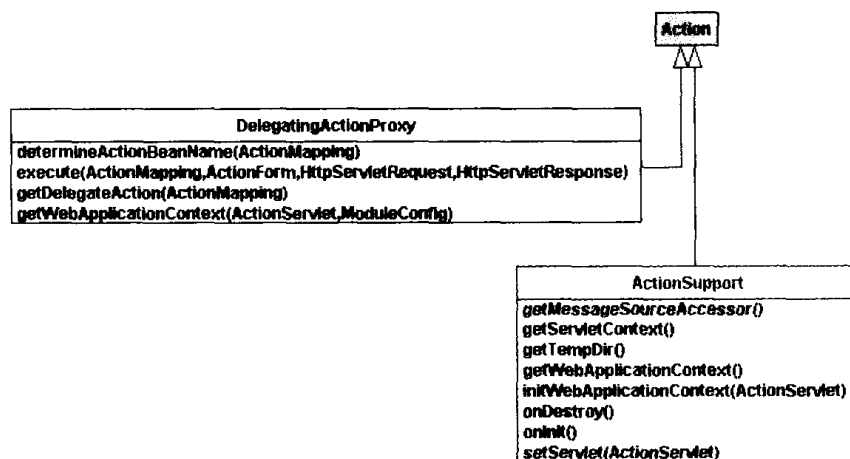


图 14-5 Spring 用于集成 Struts 的 ActionSupport 和 DelegatingActionProxy 类

为初始化 Spring ApplicationContext，开发者需要在 struts-config.xml 中配置如下类似内容。注意，contextConfigLocation 的具体取值同 Spring ContextLoaderServlet 中的对象属性。当然，这使用到 Struts 提供的扩展机制，并不是所有的 Web 框架都提供了类似的扩展机制。当然，初始化 Spring ApplicationContext 实例的对象（无论采用何种机制）并不重要，重要的是应用能够使用到 Spring IoC 和 AOP 所带来的优势，而且 Spring ApplicationContext 确实能够为应用所用。

```
<plug-in
    className="org.springframework.web.struts.ContextLoaderPlugIn">
    <set-property property="contextConfigLocation"
        value="/WEB-INF/action-servlet.xml /WEB-INF/myContext.xml"/>
</plug-in>
```

注意，如果开发者仔细浏览 ContextLoaderServlet.java 源代码，可以看到它确实是初始化了 ApplicationContext，并且往 ServletContext 上注册了。具体如下。


```

protected WebApplicationContext initWebApplicationContext()
    throws BeansException, IllegalStateException {
    getServletContext().log(
        "Initializing WebApplicationContext f
        or Struts ActionServlet '"
        + getServletName() + "', module '" + getModulePrefix()
        + "'");
    WebApplicationContext parent = WebApplicationContextUtils
        .getWebApplicationContext(getServletContext());

    WebApplicationContext wac = createWebApplicationContext(parent);
    if (logger.isInfoEnabled()) {
        logger.info("Using context class '" + wac.getClass().getName()
            + "' for servlet '" + getServletName() + "'");
    }

    // Publish the context as a servlet context attribute.
    String attrName = getServletContextAttributeName();
    getServletContext().setAttribute(attrName, wac);
    if (logger.isDebugEnabled()) {
        logger
            .debug("Published WebApplicationContext
                of Struts ActionServlet '"
                    + getServletName()
                    + "', module '"
                    + getModulePrefix()
                    + "' as ServletContext attribute with name ["
                    + attrName + "]");
    }

    return wac;
}

```

可以看到，`createWebApplicationContext(parent)` 创建了 Spring `ApplicationContext` 实例，并且通过 `ServetContext` 的 `setAttribute()` 方法注册了它。

其次，Spring 和 Struts 使能应用需要借助于 `ActionSupport`，才能够使 Struts Action 访问到 Spring `ApplicationContext`。其中，`initWebApplicationContext()` 方法如下（摘自 `org.springframework.web.struts.ActionSupport` 类）。通过该方法能够找到 Spring 的 `ApplicationContext`。

```

protected WebApplicationContext initWebApplicationContext(ActionServlet
    actionServlet)
    throws IllegalStateException {
    ServletContext sc = actionServlet.getServletContext();
    WebApplicationContext wac = (WebApplicationContext)
        sc.getAttribute(ContextLoaderPlugIn.SERVLET_
            CONTEXT_PREFIX);
}

```

```
if (wac == null) {  
    wac =  
        WebApplicationContextUtils.  
            getRequiredWebApplicationContext(sc);  
}  
return wac;  
}
```

然后, 开发者能够在继承于 `ActionSupport` 的 `Action` 中使用到 Spring 受管 `JavaBean` (借助于 Spring `ApplicationContext` 的 `getBean()` 方法获得它们)。进而, 享受到业务逻辑、安全性服务。具体的 `ActionSupport` 子类就不再给出了, 相信开发 `ActionSupport` 子类不是件困难的事情。而且, 如果您有 Struts 开发背景的话, 则可以看出, 它同开发 100% Struts `Action` 子类几乎不存在任何差别。

有一点值得开发者注意, 即无论何种 Web 框架 (视图技术), 对于 Spring 使能应用而言, 为使用到 Spring IoC 容器中的受管 `JavaBean` 服务, 必须在各种视图技术中拿到对 Spring `ApplicationContext` 的引用。如果能够拿到 Spring `ApplicationContext`, 则对于任何视图技术而言, 都不在话下。开发者应该明白 Spring `ApplicationContext` 的重要作用。另外, 为实现对 Spring `ApplicationContext` 的引用, Spring 提供了 `ApplicationContextAware` 接口 (本书在第 3 章讨论过它)。开发者只需要在自身的类中实现该接口, 即可使用到 `ApplicationContext`。因此, 在这种指导思想下, 理解各种视图技术的集成都是一脉相承的, 即在理解一种视图技术的集成后, 再理解其他的视图技术将没有什么技术障碍。这再次说明了 Spring 本身架构的极具“艺术性”。

好了, 再次回到 Spring 对 Struts 提供的集成支持吧!

请开发者仔细考虑如下问题: 如果直接借助于 `ActionSupport` 开发 Struts 应用, 则存在的优势和劣势在哪里呢? 很显然, 为了在 Struts `Action` 中使用到 Spring IoC 和 AOP, 开发者自身开发的 `Action` 必须继承于 `ActionSupport`。从开发过程考虑, 这同开发 100% Struts `Action` 并无多大区别。但是, 这将同 Spring 进行了紧耦合, 而且 `Action` 必须显式地使用 Spring 受管 `JavaBean` (借助于 `ApplicationContext` 的 `getBean()` 方法), 这不符合 IoC 的初衷。因此, 能有什么更好的方式使用 Spring IoC 和 AOP 呢?

`DelegatingRequestProcessor` (位于 `org.springframework.web.struts` 包中) 就是能够克服上述缺陷的“大救星”。其中心思想是: 将 Struts `Action` 配置在 Spring `ApplicationContext` 中, 而且这里的 `Action` 不必继承于 `ActionSupport`, 每次客户 (或者应用) 请求 Struts `Action` 时, `DelegatingRequestProcessor` 将充当代理的作用, 即通过它将 `Action` 请求转发给 Spring IoC 容器进行处理。可以看出, 此时 `Action` 将使用到 Spring IoC 容器的功能。具体使用步骤如下。

在 `struts-config.xml` 中配置 `DelegatingRequestProcessor`。

```
<action path="/userInfo.do"  
    type="org.springframework.web.struts.DelegatingActionProxy"/>
```

其次, 需要在 Spring 配置文件中定义名字为 `“/userInfo.do”` 的受管 `JavaBean`。

```
<bean name="/userInfo.do"  
    class="com.openv.spring.struts.UserInfoAction">  
    <property name="example11Service">
```

```
<ref bean="example11Service"/>
</property>
</bean>
```

可以看到, 每次对 `UserInfoAction` 请求时, `DelegatingRequestProcessor` 将充当代理的作用。与此同时, 开发者应该看到, `UserInfoAction` 使用到 `example11Service` 受管 `JavaBean`。因此, 在 `Action` 使用到 `Spring IoC` 容器功能的同时, 又不需要同 `Spring` 进行紧耦合, 何乐而不为呢? (注意, `DelegatingRequestProcessor` 的设计同第 17 章中 `Acegi FilterToBeanProxy` 的设计很相似。)

当然, 开发者需要为每个 `Action` 分别在 `struts-config.xml` 和 `Spring` 配置文件中配置相应的内容。如果能够不用在 `struts-config.xml` 中配置, 或者仅配置一次的话, 则最好不过了。现在, `Spring` 对 `Struts` 的集成包提供了 `DelegatingRequestProcessor` 实用类, 它就是克服上述缺陷的, 从而能够简化 `Spring` 和 `Struts` 使能应用的配置。具体配置过程如下。

在 `struts-config.xml` 配置文件中配置如下内容。

```
<controller processorClass=
    "org.springframework.web.struts.DelegatingRequestProcessor"/>
```

随后, 只需要在 `Spring` 配置文件中配置相应的 `Action` 即可。比如, 上述 `"/userInfo.do"`。

至此, `Spring` 对 `Struts` 提供的集成支持已经研究完毕。通过本节内容, 开发者对 `Struts` 集成应该认识很清楚了。好了, 继续整个行程吧!

14.3 Tapestry

`Tapestry`⁷, 本书将在第 15 章详细给出介绍。它以面向组件、OO, 基于事件模式, 类似于开发 `Swing`、`SWT` 一样开发 `Web` 应用, 是一种很成熟的 `Web` 框架。为在 `Tapestry` 中使用到 `Spring IoC` 和 `Spring AOP` 功能, 开发者几乎不用做任何集成工作。

14.4 JSF

`JSF`⁸, 本书在第 16 章详细给出介绍。它以面向组件、OO, 基于事件模式, 类似于开发 `Swing`、`SWT` 一样开发 `Web` 应用, 是一种正走向成熟的 `Web` 框架。`Spring` 对 `JSF` 的集成支持特别优秀。

14.5 JSP 和 JSTL

`JSP`⁹, 一种快速、简单创建动态 `Web` 内容的视图技术。从目前看, 其使用面最广泛。基于 `JSP` 开发的 `Web` 应用能够独立于具体的 `J2EE` 应用服务器和操作系统平台。

⁷ <http://jakarta.apache.org/tapestry/>

⁸ <http://java.sun.com/j2ee/javaserverfaces/>

⁹ <http://java.sun.com/products/jsp/index.jsp>

JSTL¹⁰，一种将 Web 应用中通用功能实现为简单标签的库。比如，JSTL 含有的数据迭代、条件判断、操作 XML 文档、国际化标签以及 SQL 标签等都是很常见的 Web 功能。

Spring 框架对 JSP 和 JSTL 提供了有效的集成支持。通过 Spring `WebApplicationContext` 中定义的视图解析器（`ViewResolver`）能够使用 JSP 和 JSTL。第 14.1 节给出了相关介绍。

14.6 Velocity 和 FreeMarker

Velocity¹¹和 FreeMarker¹²，都是模板语言，在 Spring Web MVC 应用中能够使用它们。它们实现的功能类似，而且易于使用。

其中，Spring 用于集成 Velocity 的支持类见图 14-6。借助于图中叶节点各个类，开发者能够快速构建 Velocity 应用。

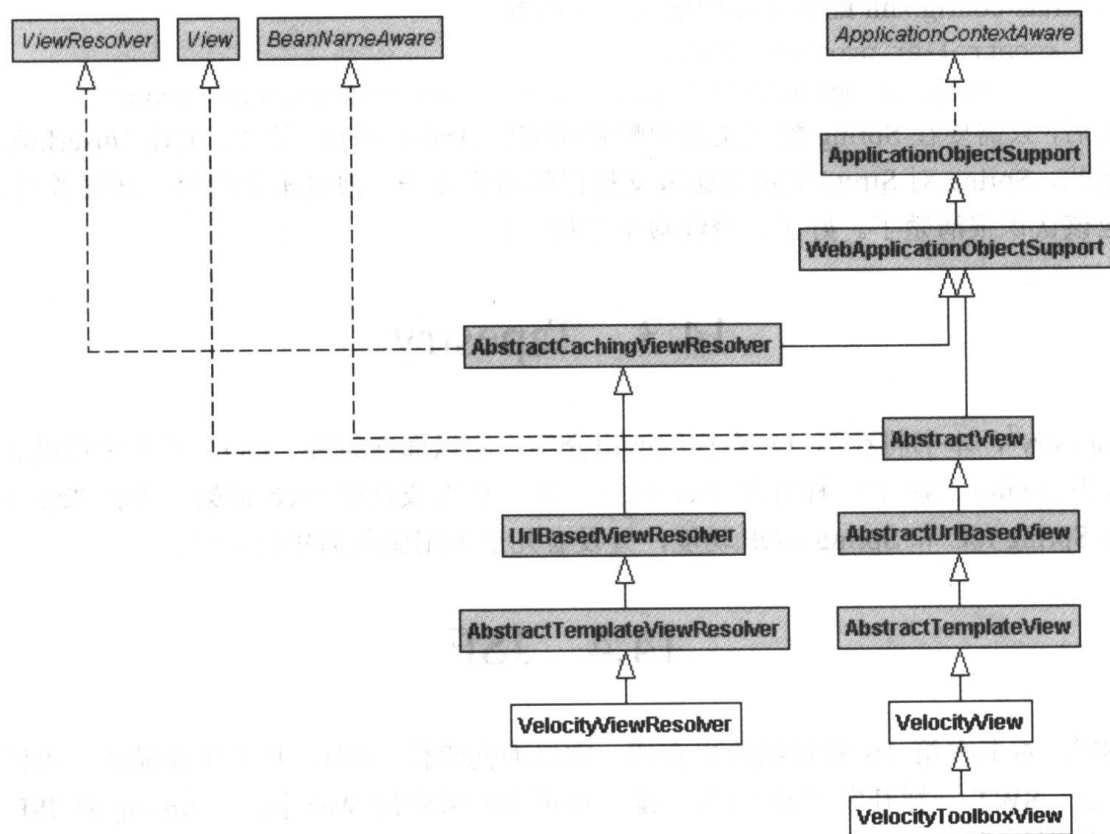


图 14-6 Spring 用于集成 Velocity 的类

Spring 用于集成 FreeMarker 的支持类见图 14-7。借助于图中叶节点各个类，开发者能够快速构建 FreeMarker 应用。

¹⁰ <http://java.sun.com/products/jsp/jstl/index.jsp>

¹¹ <http://jakarta.apache.org/velocity/>

¹² <http://freemarker.sourceforge.net/>

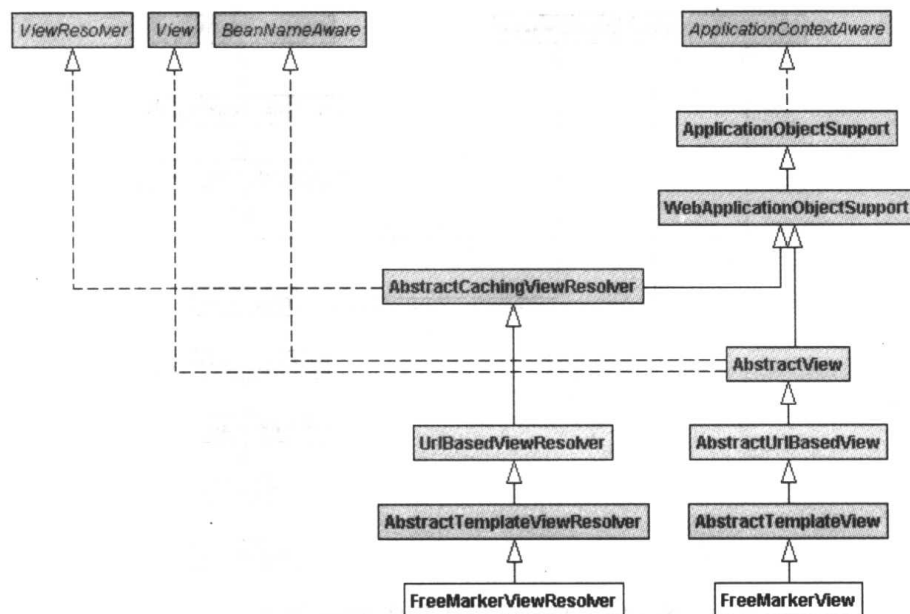


图 14-7 Spring 用于集成 FreeMarker 的类

14.7 XSLT

XSLT，用于转换 XML。通常，开发者都会以视图形式使用它。如果开发者的应用中存在大量的 XML 文档，或者基于 XML 的数据，则可以考虑使用 XSLT。

Spring 对 XSLT 提供了良好的开发支持，相应的支持类见图 14-8。借助于 AbstractXsltView，开发者能够快速构建基于 XSLT 的 Web 应用。

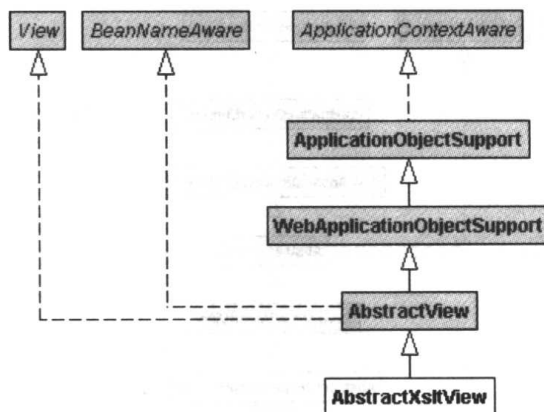


图 14-8 Spring 用于集成 XSLT 的类

14.8 Tiles

Tiles，同 Struts 绑定在一起的视图技术。Spring 对 Tiles 提供了一流的集成支持，相应的支持类见图 14-9。借助于图中叶节点各个类，开发者能够快速构建 Tiles 应用。

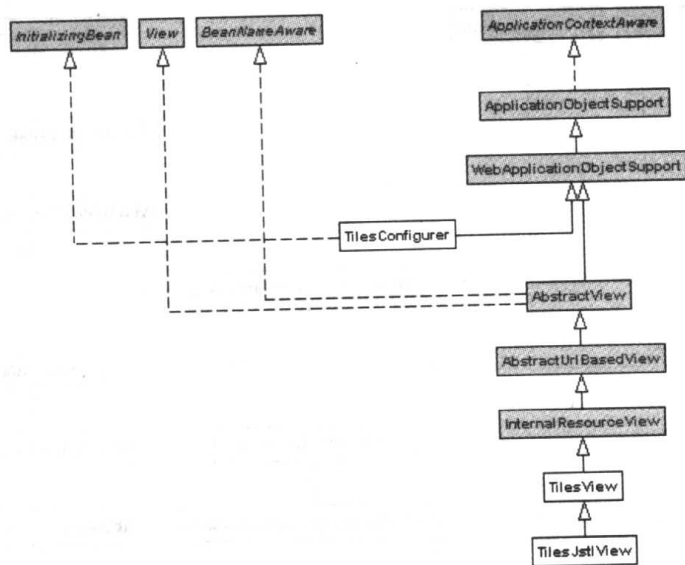


图 14-9 Spring 用于集成 Tiles 的类

14.9 JasperReports

JasperReports¹³, Open Source、Java 报表工具。开发者借助于 JasperReports 能够将内容 (Rich Content) 分发到 IE (HTML)、打印机、PDF、XLS、XML、CSV 等载体中。

其中, Spring 通过提供 `org.springframework.web.servlet.view.jasperreports` 包以支持基于 JasperReports 的 Java/J2EE 应用开发, 图 14-10 给出了 `jasperreports` 包的主要类。借助于图中叶节点的各个类, 开发者能够快速构建 JasperReports 应用。

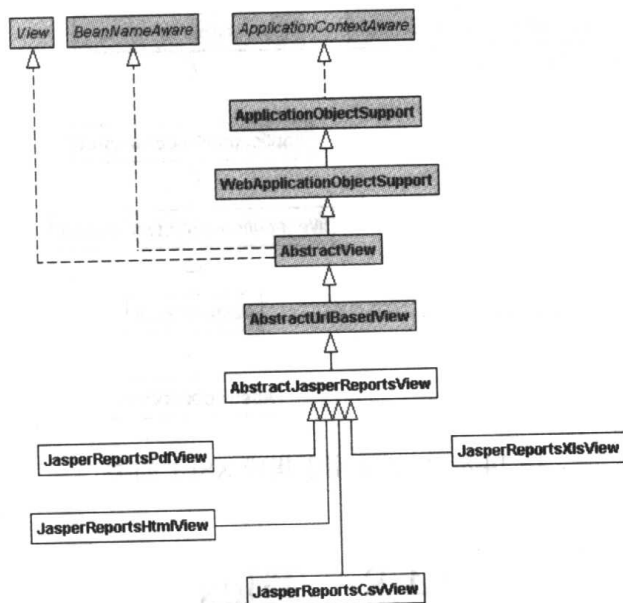


图 14-10 Spring 框架对 JasperReports 提供的支持

¹³ <http://jasperreports.sourceforge.net/>

14.10 文档视图

尽管 HTML 页面展示的数据内容很丰富,但是很多时候还需要实现其他文档视图,比如 PDF 和 Excel 视图。比如,不同浏览器对 HTML、HTTP 协议的支持程度不一样,使得相同的 HTML 页面在不同浏览器中渲染的效果不一样。借助于文档视图,比如在浏览器中嵌入相应的插件,便能够保证在不同浏览器中所浏览到的效果都是一样的。在现有的文档视图中,PDF 和 Excel 是事实上的标准。因此,本节来看看 Spring 对它们提供的具体支持。Spring 对 PDF 和 Excel 视图生成提供了支持。

其中, Spring 是借助于 iText¹⁴生成 PDF 的。Spring 借助于 Jakarta POI¹⁵创建 Excel 文档视图。当然,在使用 JasperReports 时,开发者也能够创建这些文档视图。Spring 为创建 PDF 和 Excel 文档视图提供了图 14-11 的支持类。借助于图中叶节点的各个类,开发者能够快速构建 Excel 和 PDF 文档。

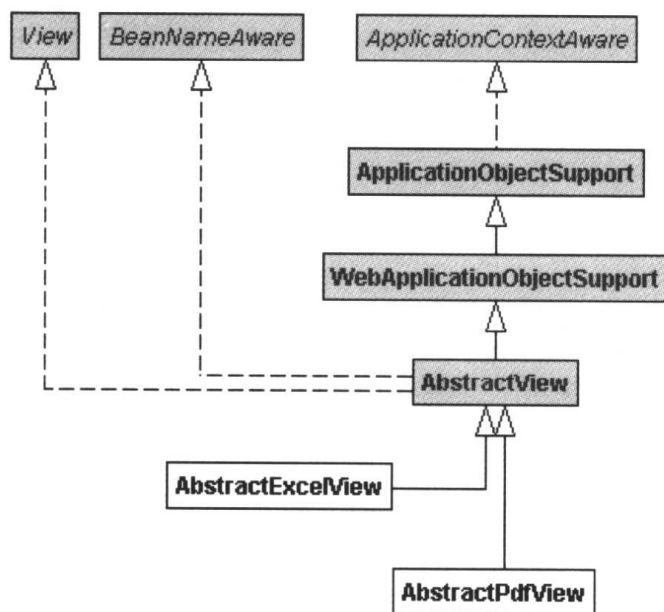


图 14-11 Spring 用于集成 iText 和 Jakarta POI 的类

14.11 小 结

Web 框架在如今的企业级应用开发中,往往扮演了重要的角色。从开发角度考虑,具体 Web 框架的开发模型决定了整个开发过程。比如借助于 Tapestry,使得 HTML 设计者能够同 Java 开发者进行合理的工作分工,这将导致很好的协同能力。借助于 Struts,使得 HTML 设计者和 Java 开发者之间的协作、分工不是很顺利,毕竟 JSP 页面很难达到所见即所得的

¹⁴ <http://www.lowagie.com/iText/>

¹⁵ <http://jakarta.apache.org/poi/>

效果，更何况 HTML 设计者对 Java 的掌握往往不是很熟练。因此，正确选用 Web 框架对于项目的成功往往起到了很关键的作用。

可喜的是，Spring 对现有的各种 Web 框架进行了集成。因此，Java 开发者在使用 Spring 架构应用过程（借助于 Spring IoC 和 AOP）中，在选用具体的 Web 框架时不会太被动。尤其是如果直接选用 Spring 原生的 Web MVC，则开发者做任何集成工作就能够在 Web 前端享受到 Spring IoC 和 AOP（但如果选用其他 Web 框架，比如 JSF，则还需提供它们的集成）。因为，Spring 本身在实现 Web MVC 时，已经考虑到这种需求。

本章对各种视图技术¹⁶作了较深入分析和比较。从目前看，Web 视图技术的发展趋势向着类似于 Swing 开发桌面一样的方式发展，即基于事件、组件、OO 方式开发 Web 应用（其中，Tapestry 和 JSF 就是这种视图技术）。

因此，本书接下来将在第 15、16 章花费大量篇幅研究 Spring 对 Tapestry 和 JSF 的集成和扩展支持。

¹⁶ 至于本章未作深入研究的视图技术，请开发者参考现有的其他 Spring 相关图书（本书附录 C 详细给出了资源介绍）。

第 15 章 Tapestry 集成

Tapestry, 基于组件的 Java Web 应用框架。它架构在 Java Servlet 基础之上, 用于创建动态、交互式网站。在 Tapestry 中, 一切都是组件。Tapestry 自身完成了 Web 应用开发中的底层开发工作, 比如分发新的 HTTP 请求、构建和解析 URL、处理本地化和国际化、表单数据的处理, 因此开发者能够以 OO、基于事件的方式开发 Web 应用。同时, 开发可重用 Tapestry 组件的过程很简单。随着 Tapestry 的深入使用, Web 应用的开发效率会越来越高。

Spring 对 Tapestry 的支持能力特别出色。为支持 Tapestry, Spring 几乎不用提供任何实用类或者集成模块。

15.1 Tapestry 介绍

在 Web 应用框架中, MVC 往往是实现 Web 应用前端分层的重要设计模式。由于 Struts 提供了优异的控制器, 这使得它成为了事实上的 Web 框架标准。在 Web MVC 中, 视图层 (V) 往往采用 JSP 或 HTML 技术, 而模型层 (M) 可用的技术较多 (比如 EJB、JDO、Spring、SDO¹等)。当然, 就技术本身而言, Struts 提供了优异的控制器实现, 才使得 Struts 在成熟的 MV 之间找到了很好的切入点, 而且 Struts 容易上手。开发者在基于 Struts 构建 Web 应用时, 自身能够发挥的余地较小。它提供的控制层对于整个 Web 应用而言很关键, 而尽管开发者能够提供自己的控制层, 但是开发者还是脱离不了使用 Struts Action。最为重要的一点是, 它在处理 Web 表单数据上并没有卓越的贡献, 尽管它提供了 Form 数据同 JavaBean 类的自动绑定, 但是开发者还是需要借助于 URL、参数这种传统的方式构建 Web 应用。Web 表单数据处理功能的强弱在很大程度上决定了 Web 框架的成熟程度。比如, 为实现常见的 CRUD 操作, 在 Web 应用中必须通过 Web 表单进行处理。因此, 我们需要新一代的 Web 框架。

开发者可以试想: 如果存在类似于 VB、Delphi、Swing、JBuilder (借助于它们开发桌面应用, 俗称“胖客户端”) 等提供的组件式、基于事件的编程方式来开发 Web 应用前端, 则开发者一定会感到兴奋。这种组件式、基于事件的编程框架往往能够实现 Web 表单自动处理, 而且开发者不需要提供单独的 JavaBean 处理表单数据的自动绑定 (注意, Struts 提供的表单数据自动绑定功能很有限, 相信有过 Struts 开发经验的开发者能够领悟到这一点)。因此, 我们期盼新一代 Web 框架的出现。当然, 面对多种选择是一种痛苦。无论如何, Struts 的使用还会持续很长时间, 但是 Web 技术的变革从来就没有停止过。Tapestry Web 框架就

¹ BEA 同 IBM 一起制定的服务数据对象 (Service Data Objects) 标准, 其中, 相应的技术规范交由 JCP 组织负责, 详情见 <http://www.jcp.org/en/jsr/detail?id=235>。

是开发者期盼的这种组件式、基于事件的编程框架。

目前, Tapestry Web 框架(图 15-1 给出了 Tapestry 项目主页)已经很成熟, 其最新版为 Tapestry 3.0.1。令人期待的 Tapestry 3.1 很令人心动, 因为它同 HiveMind² IoC 进行了紧密集成。当然, HiveMind 也考虑到与 Spring IoC 的集成³, 因此 Spring 开发者不用为此担心。

基于组件开发 Web 应用使得它(当然, 也包括 JSF)同其他 Web 应用框架(比如, JSP、Servlet)在很多方面都不一样。在一定程度上, 它们同使用 JFC Swing GUI 框架开发应用很类似。但由于 HTTP 是无状态协议, 因此在服务器端的状态管理上它们之间还是存在一些区别的。

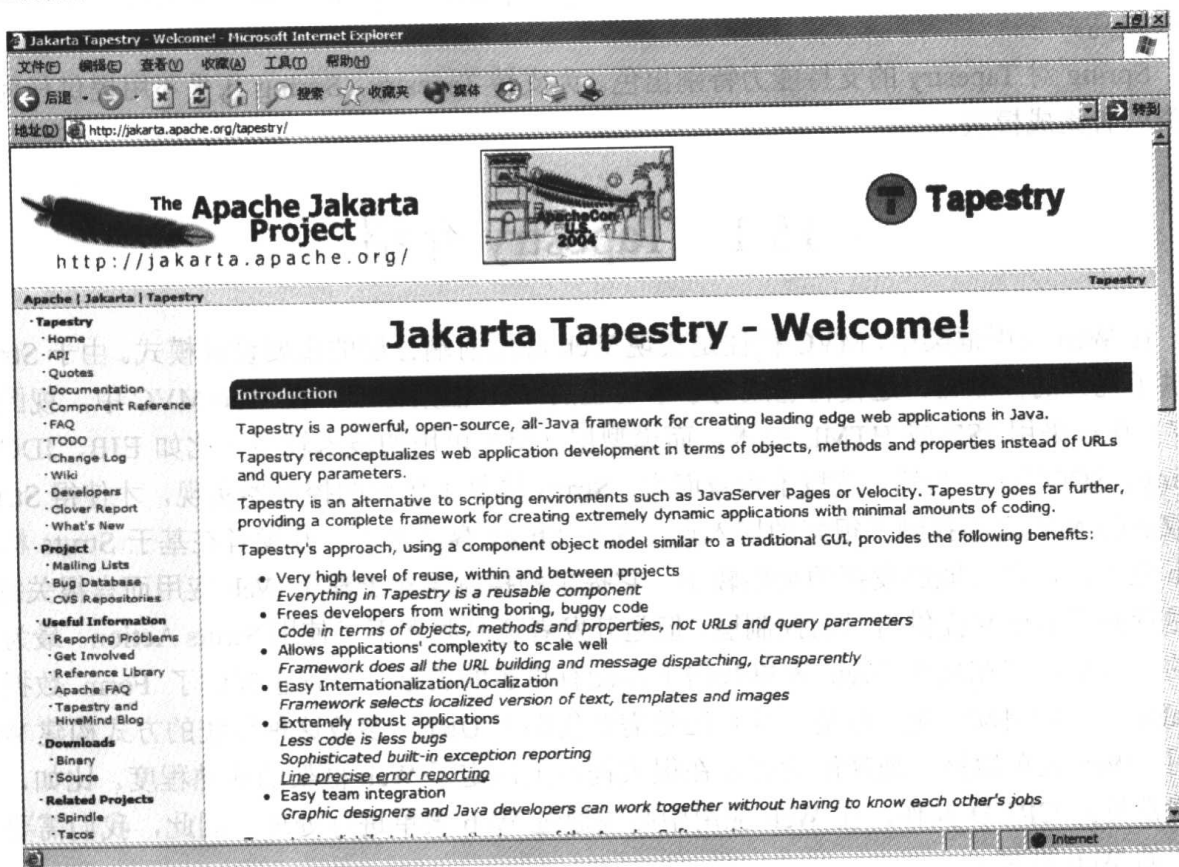


图 15-1 Tapestry 项目主页

Tapestry 的开发哲学是使得开发者能够以“对象、方法、属性”思考 Web 应用, 而不是传统的“HTTP 请求、HTTP 响应、会话(Session)、属性、参数、URL”。当然, 这对开发者进入 Tapestry 的研究和使用也给出了较高的门槛。在使用 Tapestry 开发 Web 应用的过程中, 开发者只要专注于对象、对象的方法以及属性。比如, 用户在使用 Tapestry 使能的 Web 应用中, 用户的行为(比如, 单击 URL 链接、提交表单)将触发对象属性的变化和事件(对对象中方法的调用)的发生。因此, 这些基础工作全部都是 Tapestry 关注的, 开发者只需要借助于 Tapestry 为开发者提供的 API 开发应用, 而不用理会底层的调用过程。这

² <http://jakarta.apache.org/hivemind/>

³ 通过 Spring 用户邮件列表可以获悉, HiveMind 在同 Spring 合作开发集成模块。这是很令人振奋的消息。

对于开发交互性极强的 Web 应用而言太适合了。

另外,开发者在使用 Tapestry 的过程中,一般不需要开发 Java Servlet,也不用构建 URL。一般情况下,开发者只需要使用 Tapestry 现有的组件,并配置相应的 listener 参数。通过 listener 参数指定的调用方法能够访问到后端系统,比如 EJB、Spring IoC 管理的 JavaBean、Web 服务(比如,基于 JAX-RPC)等内容。

有一点对于没有接触过 Tapestry (JSF) 的开发者而言值得注意,即面向组件的 Tapestry 同传统的面向过程操作的 Web 技术存在很大不同。比如,在 Struts 中,用户点击超链接(或者提交表单)能够触发 Action 的调用。开发者在开发 Struts 应用中,需要构建合适的 URL、查询参数的名字和类型等。然而, Tapestry 使能应用由 Page 组成,即 Web 页面。Page 由组件组成,而组件由可能是更小的组件组成。对于 Tapestry 而言, Page 是特殊组件,即仅含有一个 Tapestry 组件的页面。对于单个 Page 页面而言,各个组件的名字都不尽相同,通过组件的 ID 能够惟一标识组件。通过使用组件 ID 便形成了 Tapestry 的组件对象模型。

通常, Tapestry 将应用划分为若干个页面,其中每个页面都是使用 Tapestry 组件开发的。而且,组件一般都存在已定义好的参数类型,通过 Web 页面能够提供相应的参数值。请注意,这里的参数类型同 JavaBean 中方法的参数类型有所区别。对于 JavaBean 的方法参数而言,其中的参数只是单向的,即开发者只能向其中传入值,而不能够传出参数。对于 Tapestry 而言,方法参数是双向的,即可读出、也可写入。

在 Web 应用中,处理表单数据是最为基础、重要的内容。在 Tapestry 应用中,当用户提交表单时,处于表单中的组件将会从 HTTP 查询参数中获得值,并将这些值转换为合适的类型,然后更新对象的属性。这个过程在 Tapestry 中称之为 Rewind 过程。当 Tapestry 将页面传给浏览器时,称之为 Tapestry Render 过程。

Tapestry 应用由各个页面构成,而各个页面中的参数,比如 Form 待处理的参数,是由 OGNL 语言(Object Graph Navigation Language, OGNL)表达。OGNL⁴是 Java 表达语言,用于操作 Java 对象或属性。开发者都知道,在 JSP 2.0 中引入的表达语言(Expression Language, EL)就有类似的功能。但是,OGNL 比 EL 的功能更强大。

对于 Tapestry 应用而言,存在 Engine、Visit、Global 这样几个重要概念。其中,Global 是属于应用范围的,即存储供整个应用使用的信息,不包括具体用户或者 Session 信息。比如,在 EJB 应用中,开发者通过 Global 对象实现 JNDI 查找,即充当服务定位器的作用。对于 Visit 而言,它是用于管理服务器端状态的,即各个 Tapestry 使能应用都可以借助于 Visit 对象实现状态管理,比如管理各个用户的 Session 信息。

由于 Visit 是作为 Engine 的属性而存在的,而 Engine 又是存储在 HttpSession 中,因此 Visit 能够管理服务器端的状态。对于 Engine 而言,其实同 Visit 类似,即最终都存储在 HttpSession 中。对于 Tapestry 应用而言,Engine 和 Visit 类似,它们都是用户专有(Session)的,而不属于应用范围。

Visit 和 Engine 都配置在 Tapestry 应用的 application 文件中。比如,在 example11 中,相应的 example11.application 文件内容如下。

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

⁴ <http://www ognl.org>

```
<!DOCTYPE application PUBLIC
  "-//Apache Software Foundation//Tapestry Specification 3.0//EN"
  "http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd">
<!-- generated by Spindle, http://spindle.sourceforge.net -->
<application name="example11"
  engine-class="com.openv.spring.tapestry.Example11TapestryEngine">
  <description>
    Tapestry Application
  </description>
  <library id="contrib"
    specification-path="/org/apache/tapestry/contrib/Contrib.library"/>
</application>
```

开发者应用看到，其中定义了 Example11TapestryEngine Engine 对象。

15.2 Page 和组件模板

由于 Tapestry 使能应用由 Page 页面构成，而各个 Page 页面又由各个组件构成，因此有必要对 Page 和组件作较深入的阐述。请注意，Page 是特殊组件，即仅含有一个“组件”的页面。

在模板系统方面，Tapestry 与传统的 Web 框架（比如 Struts、WebWork）不同。因为 Tapestry 提供了自己的模板系统，而它们（使用 JSP、Velocity 等）都没有。当然，新近的 JSF 也是借助于其他模板系统的，比如 JSP。从这个角度出发，Tapestry 的架构比 JSF 先进。

大体上而言，Tapestry 模板使用了 HTML 文件，即对 HTML 文件进行了扩展。比如：

```
<tr>
  <td>
    <div align="right">国家</div>
  </td>
  <td><span jwcid="@PropertySelection"
    model="ognl:countrymodel"
    value="ognl:userinfoVO.country"/>
  </td>
</tr>
```

其中，jwcid、model、value 就是对 HTML 的扩展。这种扩展对于 HTML 工具（比如 Dreamweaver）而言是不可见的，即开发者在开发 Tapestry 的时候具有所见即所得的效果。这同其他模板系统相比，包括借助于 JSP 的那些框架，更具优势。

通常，Tapestry 由 3 部分构成，即 Tapestry HTML 模板、Tapestry 页面规范文件、Java 文件。开发者应该还记得本书在第 5 章给出过 Home.page、Home.java，这就是典型的 Tapestry 页面。Tapestry HTML 模板就是简单的 HTML 文件。其中，Tapestry 页面规范文件用于对 HTML 模板使用到的组件、对象、属性进行定义和初始化。Tapestry Java 文件用于处理页面事件。

Tapestry 会在如下几个地方寻找 HTML 模板文件。

- 与 Tapestry 页面规范文件同一目录中
- Web 应用的上下文根目录

其中, Tapestry HTML 模板可以包含如下内容:

- 静态 HTML 标签
- Tapestry 组件
- 本地化消息
- Tapestry 提供的特殊指令

一般情况下, Tapestry HTML 模板的大部分内容都是由 HTML 标签构成的。其中, 使用了 jwcid 属性 (即, Java Web Component 的缩写) 表示 Tapestry 组件, 以区分 HTML 标签。

对于模板中的组件而言, Tapestry 提供的 jwcid 属性能够表达它。比如, 在 example11 中 DelegateError 组件的 HTML 模板内容如下。

```
<span jwcid="$content$">
  <span jwcid="@Conditional" condition="ognl:delegate.hasErrors">
    <table class="error">
      <tr>
        <td>
          
        </td>
        <td>
          <span jwcid="@Delegator" delegate=
                                "ognl:delegate.firstError">错误信息</span>
        </td>
      </tr>
    </table>
  </span>
</span>
```

其中, 通过使用 @Delegator 能够表示对 Tapestry 组件的使用。至于本地化消息, Tapestry 提供了完善的支持。比如, Tapestry 允许各个 Page 页面存在各自使用的本地化消息。借助于 .properties 属性文件, 开发者能够存储用于页面和 Tapestry 组件的消息。比如, 通过如下文件给出中文消息, 即 Home_zh_CN.properties, 其内容如下。

```
success = \u5982\u4e0b\u7528\u6237\u6ce8\u518c\u6210\u529f
```

```
failure = \u521b\u5efa\u5982\u4e0b\u7528\u6237\u5931\u8d25
```

```
password = \u5bc6\u7801\u54c8\u786e\u8ba4\u5bc6\u7801\u57df\u5fc5\u987b\u4e00\u81f4
```

借助于 JDK 提供的 native2ascii 工具, 或者 Ant 对其的封装而提供的 native2ascii 任务, 开发者能够轻松构建本地化消息。然后, 通过如下示例代码, 能够访问到它们。

```
if (flag) {
    IMessageProperty page = (IMessageProperty) cycle.getPage();
    page.setMessage(getMessage("success") + ":"
        + getUserinfoVO().getUsername().trim());
}
```

```
setUserInfoVO(new UserInfoVO());  
return;  
} else {  
    IErrorProperty page = (IErrorProperty) cycle.getPage();  
    page.setError(getMessage("failure") + ":"  
        + getUserInfoVO().getUsername().trim());  
    return;  
}
```

15.3 创建 Tapestry 组件

基于组件对象模型的 Tapestry，是 Web 应用框架。Tapestry 组件实现了 IComponent 接口。对于简单组件，有可能只是由单个 HTML 文件构成的，或者没有 Tapestry Java 文件的组件，或者没有 Tapestry 页面规范的组件。比如，example11 中使用到的 ShowError 组件仅含有 ShowError.html 和 ShowError.jwc，而没有相应的 ShowError.java 支持。

当然，如果自定义组件很丰富，则开发者可以考虑将它们打包成组件库。比如，Tapestry 本身提供的 contrib 组件库。为引用组件库，开发者需要在 Tapestry 应用定义文件中给出组件库的 id 和路径，比如：

```
<library id="contrib"  
    specification-path="/org/apache/tapestry/contrib/Contrib.library"/>
```

其中，借助于<library>元素实现了上述需求。最后，开发者可以在 Tapestry HTML 模板文件中使用它，比如：

```
<tr>  
    <td>  
        <div align="right">兴趣</div>  
    </td>  
    <td>  
        <input jwcid="selectedinterests@contrib:Palette"  
            model="ognl:interestmodel"  
            availableTitleBlock="ognl:components.availableTitleBlock"  
            selectedTitleBlock="ognl:components.selectedTitleBlock"  
            sort="ognl:@org.apache.tapestry.contrib.palette.SortMode@LABEL"  
            selected="ognl:userInfoVO.selectedinterests" tableClass="interest"/>  
    </td>  
</tr>
```

15.4 Tapestry 校验子系统

基于作者的认识，Tapestry 校验子系统在现有的 Web 框架中应该是最完善的。开发者在 example11 中应该体会到了 Tapestry 校验子系统的威力了。为使用 Tapestry 提供的校验子系统，开发者需要完成这样几项工作。

其一，开发 Tapestry Delegate 器。其功能主要是格式化错误或提示信息的输出。示例代码如下。

```
package com.openv.spring.tapestry;

import org.apache.tapestry.IMarkupWriter;
import org.apache.tapestry.IRequestCycle;
import org.apache.tapestry.form.IFormComponent;
import org.apache.tapestry.valid.IValidator;
import org.apache.tapestry.valid.ValidationDelegate;

/**
 * Example11 Tapestry Delegate
 *
 * @author luoshifei
 */
public class Example11ValidationDelegate extends ValidationDelegate {

    public void writeLabelPrefix(IFormComponent component,
        IMarkupWriter writer, IRequestCycle cycle) {
        if (isInError(component)) {
            writer.begin("span");
            writer.attribute("class", "label-error");
        }
    }

    public void writeLabelSuffix(IFormComponent component,
        IMarkupWriter writer, IRequestCycle cycle) {
        if (isInError(component)) {
            writer.end();
        }
    }

    public void writeAttributes(IMarkupWriter writer, IRequestCycle cycle,
        IFormComponent component, IValidator validator) {
        if (isInError())
            writer.attribute("class", "field-error");
    }

    public void writeSuffix(IMarkupWriter writer, IRequestCycle cycle,
        IFormComponent component, IValidator validator) {
        if (validator != null && validator.isRequired()) {
            writer.printRaw("&nbsp;");
            writer.begin("span");
            writer.attribute("class", "required-marker");
            writer.print("*");
            writer.end();
        }
    }
}
```

```

    }

    if (isInError()) {
        writer.printRaw("&nbsp;");
        writer.beginEmpty("img");
        writer.attribute("src", "images/field-error.png");
        writer.attribute("width", 16);
        writer.attribute("height", 16);
    }
}

}

```

其二，在 **Tapestry Page** 页面规范中定义它。比如：

```

<description>Home Page</description>
<bean name="delegate"
class="com.openv.spring.tapestry.Example11ValidationDelegate"/>

```

其三，开发 **Tapestry** 校验器，或者开发者可以直接使用 **Tapestry** 提供的校验器。其中，**example11** 对 **Tapestry** 提供的校验器进行了扩充，以利于输出本地化消息，或者说统一定义消息的来源。比如，字符串校验器示例如下。

```

package com.openv.spring.tapestry;

import java.util.Locale;
import java.util.ResourceBundle;

import org.apache.tapestry.valid.StringValidator;

/**
 * 创建新的字符串校验器。主要用于对字符串的校验
 * 该实现用于处理中文的提示信息
 *
 * @author luoshifei
 */
public class Example11StrValidator extends StringValidator {

    protected String getPattern(String override, String key, Locale locale) {
        if (override != null)
            return override;

        ResourceBundle strings = ResourceBundle.getBundle(
            "com.openv.spring.tapestry.ValidationStrings", locale);

        return strings.getString(key);
    }
}

```


另外, E-mail 校验器示例如下。

```
package com.openv.spring.tapestry;

import java.util.Locale;
import java.util.ResourceBundle;

import org.apache.tapestry.valid.EmailValidator;

/**
 * 创建新的 E-mail 校验器。主要用于对 E-mail 格式的校验
 * 该实现使用了 example11 实例用于处理中文的提示信息
 *
 * @author luoshifei
 */
public class Example11EmailValidator extends EmailValidator {

    protected String getPattern(String override, String key, Locale locale) {
        if (override != null)
            return override;

        ResourceBundle strings = ResourceBundle.getBundle(
            "com.openv.spring.tapestry.ValidationStrings", locale);

        return strings.getString(key);
    }

}
```

开发者需要在 **ValidationStrings_zh_CN.properties** 文件中定义本地化消息, 即中文信息。

```
field-is-required={0} \u57df\u5fc5\u987b\u8933\u5165.
```

```
field-too-short={1} \u57df\u4e0d\u80fd\u591f\u5c11\u4e8e {0} \u4f4d.
```

```
invalid-email-format={0} \u57df\u683c\u5f0f\u4e0d\u5bf9\u3002\u6b63\u786e\u7684\u683c\u5f0f\u4e3ausername@xxx.yyy.
```

其四, 在 **Tapestry Page** 规范中声明待使用的校验器。声明校验器的示例代码如下:

```
<bean name="MyStrValidator"
    class="com.openv.spring.tapestry.Example11StrValidator"
    lifecycle="page">
    <set-property name="required" expression="true"/>
    <set-property name="minimumLength" expression="3"/>
    <set-property name="clientScriptingEnabled" expression="true"/>
</bean>

<bean name="MyEmailValidator"
    class="com.openv.spring.tapestry.Example11EmailValidator"
```

```

        lifecycle="page">
        <set-property name="required" expression="true"/>
        <set-property name="clientScriptingEnabled" expression="true"/>
    </bean>

```

其五，在 Tapestry HTML 模板文件中使用校验器。

//请注意，务必在 Form 组件中定义 `elegate` 属性，即告之 Tapestry 框架待校验的表单

```

<form jwcid="@Form" listener="ognl:listeners.submitPersonInfo"
    delegate="ognl:beans.delegate">
    <br>
    <table border="10" align="center"
        bordercolor="#0099CC" cellpadding="6" bordercolorlight="#999999">

```

//定义校验过程中错误或提示信息的输出

```

    <tr>
        <td colspan="2" bgcolor="#FFFF40">
            <span jwcid="@DelegateError"
                delegate="ognl:beans.delegate"/>
            <span jwcid="@ShowMessage"/>
            <span jwcid="@ShowError"/>
        </td>
    </tr>
    <tr>
        <td colspan="2" bgcolor="#66CCFF">输入用户注册信息: </td>
    </tr>
    <tr>
        <td>

```

//借助于 FieldLabel，使得 Tapestry Delegate 在处理错误或提示信息时能够给出更加明确的提示

```

        <div align="right"><span jwcid="@FieldLabel"
            field="ognl:components.username"/></div>
        </td>
        <td>
            <input type="text"
                jwcid="username@ValidField" displayName="用户名"
            //定义待使用的校验器
            validator="ognl:beans.MyStrValidator"
            value="ognl:userinfoVO.username" size="30"/>
        </td>
    </tr>
    <tr>
        <td>

```

//借助于 FieldLabel，使得 Tapestry Delegate 在处理错误或提示信息时能够给出更加明确的提示

```

        <div align="right">
            <span jwcid="@FieldLabel" field="ognl:components.password"/>
        </div>
    </td>
    <td>
        <input type="text" jwcid="password@ValidField"
            displayName="密码" hidden="ognl:true"

```

//定义待使用的校验器

```

        validator="ognl:beans.MyStrValidator"
        value="ognl:userinfoVO.password" size="30"/>
    </td>
</tr>
<tr>
    <td>

```

//借助于 FieldLabel, 使得 Tapestry Delegate 在处理错误或提示信息时能够给出更加明确的提示

```

        <div align="right">
            <span jwcid="@FieldLabel"
                field="ognl:components.repassword"/>
        </div>
    </td>
    <td>
        <input type="text" jwcid="repassword@ValidField"
            displayName="确认密码" hidden="ognl:true"

```

//定义待使用的校验器

```

        validator="ognl:beans.MyStrValidator"
        value="ognl:userinfoVO.repassword" size="30"/>
    </td>
</tr>
<tr>
    <td>

```

//借助于 FieldLabel, 使得 Tapestry Delegate 在处理错误或提示信息时能够给出更加明确的提示

```

        <div align="right">
            <span jwcid="@FieldLabel"
                field="ognl:components.email"/>
        </div>
    </td>
    <td>
        <input type="text" jwcid="email@ValidField"

```

```
//定义待使用的校验器
```

326

其六，开发者需要在处理事件时使用 `Delegate` 在后台对表单数据进行语法和语义上的校验。

```
/**
 * 处理表单的提交
 *
 * @param cycle
 */
public void submitPersonInfo(IRequestCycle cycle) {
    log.info("submitPersonInfo().....");

    //调用 Tapestry ValidationDelegate 给出的统一接口，对 Form 数据进行处理

    IValidationDelegate delegate = (IValidationDelegate) getBeans()
        .getBean("delegate");
    if (delegate.getHasErrors())
        return;
    String password = getUserinfoVO().getPassword().trim();
    String repassword = getUserinfoVO().getRepassword().trim();
    if (!password.equals(repassword)) {
        IErrorProperty page = (IErrorProperty) cycle.getPage();
        page.setError(getMessage("password"));
        return;
    }
    boolean flag = this.getExample11Service().setUserInfo(
        getUserinfoVO());
    if (flag) {
        IMessageProperty page = (IMessageProperty) cycle.getPage();
        page.setMessage(getMessage("success") + ":"
            + getUserinfoVO().getUsername().trim());
        setUserinfoVO(new UserInfoVO());
        return;
    } else {
        IErrorProperty page = (IErrorProperty) cycle.getPage();
        page.setError(getMessage("failure") + ":"
            + getUserinfoVO().getUsername().trim());
        return;
    }
}
```

15.5 管理服务器端状态

Web 应用的服务器端状态管理始终是最基础的、重要的内容之一。其中，服务器端状态可以是简单的标志位，还可以是 `JDBC ResultSet` 等内容。比如，将用户登录信息存储在服务器端。对于 Tapestry 而言，可以将用户登录信息存储到 `Visit` 中。

而对于其他应用框架,包括 Struts,很多信息需要开发者自身管理服务器端的状态。比如,开发者仅仅能够借助于 Java Servlet API 提供的 HttpSession 存储具体用户的登录信息。用户可以从 Session 中读取信息,也可以将信息写入到 Session 中。借助于 Tapestry,开发者可以直接操作 Visit、Engine 以及持久化属性管理服务器端的状态。

在 example11 中,存在如下示例代码。

```
<property-specification name="countrymodel"
    type="org.apache.tapestry.form.IPropertySelectionModel">
</property-specification>

<property-specification name="interestmodel"
    type="org.apache.tapestry.form.IPropertySelectionModel"
    persistent="yes">
</property-specification>
```

其中,interestmodel 借助于 persistent 属性将其自身通过 HttpSession 存储起来,这使得应用不用直接借助于底层 HttpSession API 实现服务器端状态的管理。

当然,对于 Web 应用而言,通过 Session 管理服务器状态的同时对目标环境(CPU 时间、内存空间、网络带宽等)也提出了更高的要求。因此,能够不通过 HttpSession 维护用户登录信息,而尽量避免使用。

在 Tapestry 应用中,HttpSession 的创建会通过如下两种方式进行。其一,创建了 Visit 对象;其二,初次存储持久化属性。一旦创建了 HttpSession, Tapestry 会将 Engine 存储到其中。因为,Visit 对象存储在 Engine 对象中。

至于 Tapestry Web 应用是否创建了以及何时创建了 HttpSession,开发者可以不用关注。

15.6 配置 Tapestry 应用

在 Tapestry 设计之初,考虑到对不同 JVM 的支持和 Java Servlet API 版本的支持。比如, Tapestry 支持 JDK 1.2.2、JDK 1.3.x、JDK 1.4.x、JDK 1.5.x 等。当然,Java Servlet 2.2、2.3、2.4 也不例外。比如,example11 就是基于 Java Servlet 2.4 的 Tapestry Web 应用。开发者可以通过在 Web 应用的 web.xml 部署描述符中配置 Tapestry 提供的 Servlet,即 org.apache.tapestry.ApplicationServlet,开发者即可开发 Tapestry 使能 Web 应用。比如,example11 提供的示例 web.xml 如下。借助于 contextConfigLocation 参数指定 Spring 配置文件的位置。

```
<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">

    <display-name>example11</display-name>
```

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

<filter>
  <filter-name>redirect</filter-name>
  <filter-class>org.apache.tapestry.RedirectFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>redirect</filter-name>
  <url-pattern>/</url-pattern>
</filter-mapping>

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

<servlet>
  <servlet-name>example11</servlet-name>
  <servlet-class>
    org.apache.tapestry.ApplicationServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>example11</servlet-name>
  <url-pattern>/app</url-pattern>
</servlet-mapping>

</web-app>
```

其中，开发者是否注意到，`org.apache.tapestry.RedirectFilter`。它负责将客户请求，比如 `HttpRequest`，统一由 Tapestry Web 框架处理。

15.7 与 Spring 集成

Spring 基本上无需对 Tapestry 做任何专有集成工作，开发者只需要在 Engine 中获取到 `Spring ApplicationContext` 即可。具体步骤如下。

首先，在 Engine 中借助于 Spring 提供的 `WebApplicationContextUtil`，开发者可以获得 `ApplicationContext`。然后存储到 Tapestry Global 中。其中，`example11` 提供的示例代码如下。

```
package com.openv.spring.tapestry;

import java.util.Map;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.tapestry.IRequestCycle;
import org.apache.tapestry.engine.BaseEngine;
import org.apache.tapestry.request.RequestContext;
import org.springframework.context.ApplicationContext;
import org.springframework.web.context.support.WebApplicationContextUtils;

/**
 * Example11 Tapestry 引擎
 *
 * @author luoshifei
 */
public class Example11TapestryEngine extends BaseEngine {
    private static final Log log = LogFactory
        .getLog(Example11TapestryEngine.class);

    public static final String APPLICATION_CONTEXT_KEY = "appContext";

    public Example11TapestryEngine() {
        log.info("Create a Example11TapestryEngine Engine Instance.");
    }

    /**
     * 通过 Spring 提供的 WebApplicationContextUtils,
     * 将 ApplicationContext 放置到 Tapestry 中。
     */
    protected void setupForRequest(RequestContext context) {
        super.setupForRequest(context);

        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);
        }
    }

    /**
     * 完成页面清理工作
     */
}
```



```

    */
    protected void cleanupAfterRequest(IRequestCycle cycle) {
        super.cleanupAfterRequest(cycle);
    }
}

```

其次, 需要在使用 Spring IoC 提供的 JavaBean 服务的 Tapestry Page 页面中引用 Spring 配置文件定义的服务名。比如:

```

<property-specification name="example11Service"
    type="com.openv.spring.service.IExample11Manager">
    global.appContext.getBean("example11Service")
</property-specification>

```

第三, 在 Page 页面对应的 Tapestry Java 中处理对它的使用。比如:

```

/**
 * 构建兴趣列表模型
 *
 * @return
 */
private IPropertySelectionModel buildInterestmodel() {
    log.info("buildInterestmodel().....");
    String[] interestmodelStr;
    List list = null;
    try {
        //开发者需要借助于 getExample11Service()方法获得对 Spring IoC 的调用
        list = getExample11Service().getInterests();
    } catch (DataAccessException ex) {
        log.error("访问数据库异常", ex);
        IErrorProperty page = (IErrorProperty) this.getRequestCycle()
            .getPage();
        page.setError("访问数据库期间出现异常, 请确保数据库成功创建和运行!");
        return new StringPropertySelectionModel(new String[] { "" });
    }
    interestmodelStr = new String[list.size()];
    for (int i = 0, k = list.size(); i < k; ++i) {
        if (Tapestry.isNotBlank((String) list.get(i))) {
            interestmodelStr[i] = (String) list.get(i);
        }
    }
    return new StringPropertySelectionModel(interestmodelStr);
}

```

可以看出, Spring 集成 Tapestry 的工作量很小。开发者只需要在 Tapestry Engine 中获得 Spring ApplicationContext 即可, 从而获得 Spring IoC 和 Spring AOP, 其中包括对事务抽象、基于 Acegi 安全框架的认证和授权服务的使用。在 Tapestry 3.1 发布后, Spring 同 Tapestry 的集成将更加通畅。

15.8 小 结

本章结合 Tapestry 实例，即 example11，详细对 Tapestry 进行讨论。其中，还讨论了 Tapestry 与 Spring 集成的内容。开发者应该知道，Tapestry 是事件驱动的 Web 框架。它使用事件驱动用户界面（表示层）和应用后端之间的交互。这同传统的 Swing、SWT 类似，因此采用 Tapestry 开发 Web 应用应该是时候了。目前，著名的 TheServerSide 社区就是基于 Tapestry 构建的⁵。当然，新近的 JSF 也是事件驱动的 Web 框架。同 Tapestry 一样，如果 Web 开发者（包括 Tapestry 开发者）对基于组件开发 Web 应用前端感兴趣，赶紧去领略 JSF 吧。第 16 章为您呈现 JSF。

⁵ <http://www.theserverside.com/articles/article.tss?l=TSSTapestry>

第 16 章 JSF 集成

JSF, 经过 JCP JSR-127 专家成员近 3 年的努力, 在 2004 年终于发布了 1.0 版。这是令开发者兴奋的 Web 框架技术。目前, 其最新版为 JSF 1.1, 而 JSF 1.2¹ 规范正处于制定中。从 JSF 的发展历程来看, 它其实存在了很长时间。但是, 还不能认为 JSF 是成熟的技术。比如, 1.0 与 1.1 版本之间的发布时间差很短, 而且 1.2 的变动比较大。

无论如何, 它将是 J2EE5.0 中的标准技术, 而且整个业界对 JSF 关注的程度相当高。其中, 包括 Sun (Java Creator Studio 支持 JSF 的开发)、IBM (WebSphere Studio 5.x/Rational Application Developer 6.x)、Oracle (JDeveloper 10g)、Borland (JBuilder 2005 Enterprise) 等。作为一项由 JCP 制定的标准, JSF 比 Tapestry 更具优势。比如, 文档更丰富、整个使用社区将更加广泛。但是, 事实将会如何呢? 正如 Hibernate 对实体 Bean 的颠覆一样, 在未来的几年中 Tapestry 是否也能够起同样的作用呢?

Spring 对 JSF 提供了一流的集成能力。另外, Sourceforge 项目 JSF-Spring² 也提供了不错的集成能力。本章将同时介绍它们。

16.1 Web 前端开发的趋势

就目前而言, Tapestry 成功实施的大型项目³ 还是很多的, 其中包括 TheServerSide。当然, 自从 Struts 的创始人 Craig McClanahan 加入到 Sun 参与 JSF 的开发后, JSF 慢慢得到升温。就目前而言, 在 Web 框架中, Struts 应该是事实上的标准。因此, Craig McClanahan 的工作应该得到肯定, 也就是说开发者应该对 JSF 持乐观态度。

其实, 细心的读者能够感觉到, 上述内容提出了一个问题, 即为什么 JSF 会慢慢升温? 难道只是 Craig McClanahan 个人的功劳吗? 其实, 不是的。单纯从技术本身而言, JSF 的架构并不是很前卫, Tapestry 在很多方面的架构还是略胜一筹。尤其是 Tapestry 3.1 在架构方面改进了很多。在架构企业应用时, 分层是常用的伎俩。因此, 现阶段, 前端表示层、中间业务逻辑层、后端持久化层等分层方法就出现了。哪怕是 Spring 的 XML 配置文件本身都追求分层的哲理, 即要求用于表示层的 Spring Web MVC 的内容单独配置在一配置文件中, 用于中间业务逻辑层也单独配置在一文件中, 至于后端持久化层也可以依此类推。

开发者可以看到, 后端持久化层已经发展得很完善。比如, JDO、Hibernate、实体 Bean、SDO、O/R Mapping、O/R XML⁴ 绑定等用于持久化的技术都很成熟。尽管实体 Bean

¹ <http://www.jcp.org/en/jsr/detail?id=252>

² <http://jsf-spring.sourceforge.net/>

³ <http://howardlewisship.com/>, 提供了 Tapestry 创始人 Howard M. Lewis Ship 开发过的 Tapestry 相关项目。

⁴ 比如 JAXB, 见 <http://java.sun.com/xml/jaxb/>。当然, Apache XMLBeans 也可以认为是这方面的成功案例, 见 <http://xmlbeans.apache.org>。

的操作效率差,但是在引入 Hibernate、TopLink、JDO 中的优点之后,相信 EJB 3.0 持久化模型能够做得很优秀。无论如何, O/R Mapping 是其中的主流,而且 O/R Mapping 技术已经相当成熟了。如果说不成熟的话,那只能说供开发者使用的 API 不够灵活、功能不够强大而已。因此,就目前而言,业界在后端持久化层已经没有花大力气进一步拓展它的必要了。可以认为,在持久化层 Open Source、非 Open Source 领域都取得了很大的成果。而且,可喜的是,它们在一起努力打造 EJB 3.0。

再来看中间业务逻辑层。现存的优秀解决方案很多,比如 Open Source 领域的 Spring(尽管它支持所有层的架构和开发,但不可否认将它作为中间业务逻辑层使用是最常见的做法)、J2EE 应用服务器(其中也包括 Open Source 和非 Open Source 领域,比如 JBoss)。上述两层都不缺乏标准的支持。

最后,再来看前端。在架构 Web 应用过程中,终端零部署模式被广大 ISV 和客户所接受。因此,开发基于 HTML 的 Web 企业应用成为了主流。不可否认,在 Java/J2EE 领域,开发 Web 前端往往生产率低下。借助于 JSP 开发 Web 前端,包括 Struts,对开发效率并不会产生质的飞跃。原因何在?因为它们基本上都是在对 HTML 简单抽象的基础上开发的。期间,没有使用到组件编程的概念、没有使用到 OO 所带来的优势。而且,在 Java/J2EE 领域,这一层缺乏标准的支持。尽管 Tapestry 存在多年,但是整个社区对它的接纳速度还是不够快的。当然,这同 Tapestry 的学习曲线⁵有一定关系。

在呼唤标准的同时,JSF 诞生了。相信仔细阅读过第 15 章的开发者,对 Tapestry 这种基于组件、事件驱动的 Web 前端开发模型应该很熟悉了。JSF 类似于 Tapestry。

基于组件、事件驱动的 Web 前端开发才是今后发展的趋势、主流。借助于 Delphi、PB、VB、Swing/SWT 开发桌面应用速度之快,效率之高是前所未有的。作者也相信,随着 Tapestry/JSF 的广泛、深入应用,Web 前端开发的效率将会有质的飞跃。您做好准备没有?

16.2 JSF 介绍

JSF (JavaServer Faces⁶) 是为 Java Web 应用提供用户界面的框架。它为简化 Web 页面的开发做了许多工作,尤其是引入了 POJO 的编程模型,这也是 Open Source 推动的结果。类似于 Tapestry,JSF 提供的功能⁷有:

- 借助于可重用的 Web UI 组件构建 Web 界面。这使其脱离底层 HTML 的开发模式。
- 能够简化 Web 页面和后端应用的数据传输。开发者通过 Tapestry 应用应该已经明白了这种优势所带来的后果。
- 使得开发者易于管理服务器端的 UI 状态。类似于 Tapestry,JSF 框架自身也能够管理服务器端的状态。
- 能够基于事件模式开发 Web 应用。这同传统的 Swing、SWT 开发桌面应用类似。

⁵ 作者使用 Tapestry 架构、开发 Web 应用已经有将近 1 年的开发经验。相比其他 Web 框架,Tapestry 较难掌握。但一旦开发者经历该阶段后,您会发现再来研究、学习其他 Web 框架是很简单的事情,包括 JSF。

⁶ <http://java.sun.com/j2ee/javaxserverfaces/index.jsp>

⁷ <http://www.jcp.org/en/jsr/detail?id=127>

- 开发者可以轻松构建、重用 UI 组件。

通过 David Geary⁸ 的 blog，可以获悉 JSF 至少存在这样 5 方面的优势。

- 基于组件模型。自从 OO 被开发者推崇以来，基于组件架构企业应用便成为主流。将组件模型引入到 Web 前端是开发者的一大福音。目前，尽管 JSF 本身内置的组件数量有限，但 JSF 是基于组件模型架构的，因此开发者能够灵活添加自身的 JSF 组件。当然，现存的第三方组件很丰富。比如，Oracle 的 ADF（架构在 JSF 基础之上）、JSFCentral⁹提供了 JSF 最新资讯。
- 支持简单的 IoC。Spring 实现了 IoC 容器。在 J2EE 应用服务器中，也内置了 IoC 的实现，这种实现供 J2EE 服务器使用。而 Spring 提供的 IoC 容器更多的是为企业应用服务。JSF 借用了 Spring、HiveMind、PicoContainer 等 IoC 容器的思想，也实现了简单的 IoC 容器。借助于 JSF-Spring 项目，开发者能够实现 JSF 受管 JavaBean 和 Spring 受管 JavaBean 之间的相互引用，即在 Spring ApplicationContext 中能够使用到 JSF 中配置的受管 JavaBean；反之亦然。
- 支持值引用。在 Tapestry/JSF 中，实现了 JavaBean 中参数传递所不具有的功能，即值的引用（和其中的参数传递）是双向的（在第 15 章阐述过这方面的内容）。开发者既可以读取在 Tapestry/JSF 页面中定义的属性域的取值，还可以给它赋值。比如在 `<h:inputText id="username" value="#{InfoBean.username}">` JSF 片断中的 `InfoBean.username`。在服务器渲染响应结果给客户时，能够读取 `username` 的取值；在客户分发请求到服务器端时，能够给 `username` 赋值。借助于 `#{InfoBean.username}` 能够实现值引用，而不是简单的参数传递。
- 支持方法引用。直接通过表达式语言（Expression Language, EL），就能够调用方法。比如，借助于 `<h:input validator="#{InfoBean.validatePassword}">`，JSF 能够调用 `InfoBean` 的 `validatePassword` 方法。
- 扩展性强。JSF 的扩展性相当好。借助于 Decorator 设计模式，使得开发者能够动态更换 JSF 中的状态管理器（State Manager）、导航处理器（Navigation Handler）、变量解析器（Variable Resolver）、属性解析器（Property Resolver）、视图处理器（View Handler）、默认 Action 监听器（Default Action Listener）。

JSF 的优势还有很多。当然，JSF 可能不适合于所有的 Web 应用。由于 JSF 是基于事件模式来开发 Web 应用的，这对于开发交互性极强的应用而言特别合适。因为传统的开发方式，比如 JSP 和 Servlet，在开发 Web 应用时，对于开发交互性极强的应用相对困难。

在 Web 应用中，引入 JSF、Tapestry 并不是为取代现有的技术而出现的，正如上述所提到的一样。

本书将结合 example29 实例对 JSF 的集成进行研究。其中，本书假设开发者对 JSF 有所了解，因此对 JSF 的具体细节¹⁰并不会在本书中研究。

⁸ 同 Cay Horstmann 一起写作《Core JavaServer Faces》一书。通过，<http://www.phptr.com/title/0131463055>，能够找到该书的介绍。David Geary 的 blog 位于 <http://jroller.com/page/dgeary/>。

⁹ <http://www.jsfcentral.com/>

¹⁰ 开发者可以参考另一本 JSF 方面的图书，比如《JSF in Action》一书，由 Manning 出版社于 2004 年出版发行。

16.3 Spring 和 JSF-Spring 提供的 JSF 集成

JSF 本身也提供了 IoC 容器的功能, 因此开发者可以不用使用 Spring 这样的 IoC 容器。但是, JSF 本身实现的 IoC 功能很有限。Spring 除了提供 IoC 容器外, 还提供 Spring AOP 实现、J2EE 服务抽象。比如, 开发者可以使用声明式事务、声明式安全性 (借助于 Acegi 实现, 在第 17 章研究它)。另外, 从企业应用分层架构的角度出发, 尽管 JSF 支持各层 (前端、中间层、持久化层等) 的架构和开发工作, 但是这使得应用的层次不清晰, 给应用的后维护工作带来不便。因此, 将 JSF 作为前端, Spring 作为中间层和持久化层使用使得应用的架构更加清晰、合理。

然而, 如何才能能够在 JSF 层使用到 Spring 中的受管 JavaBean 服务, 进而使用到 Spring 其他方面的强大功能? 默认时, JSF 使用了变量解析器来定位 JSF 应用中定义的受管 JavaBean。因此, 可以从此处入手分析 Spring 和 JSF-Spring 项目提供的 JSF 集成支持。

Spring 框架为集成 JSF, 而提供了 `org.springframework.web.jsf` 包。其具体内容如下。

- **DelegatingVariableResolver**: JSF `VariableResolver`。由于 JSF 允许开发者替换其自身的变量解析器, 因此借助于 Spring 提供的 `DelegatingVariableResolver`, 使得 JSF 使能应用能够享受到 Spring 受管 JavaBean 所带来的优势。
- **FacesContextUtils**: 为获得具体 `FacesContext` 的根 `WebApplicationContext`, Spring 框架提供了 `FacesContextUtils` 类。其中, 包括了若干个实用方法。类似于集成其他 Web 框架, 开发者可以不用替换 JSF 变量解析器, 就可以直接使用到 Spring 受管 JavaBean。在借助于 `FacesContextUtils` 获得 Spring `ApplicationContext` 后, 一切问题都能够得到妥善解决。

相应的类图见图 16-1。

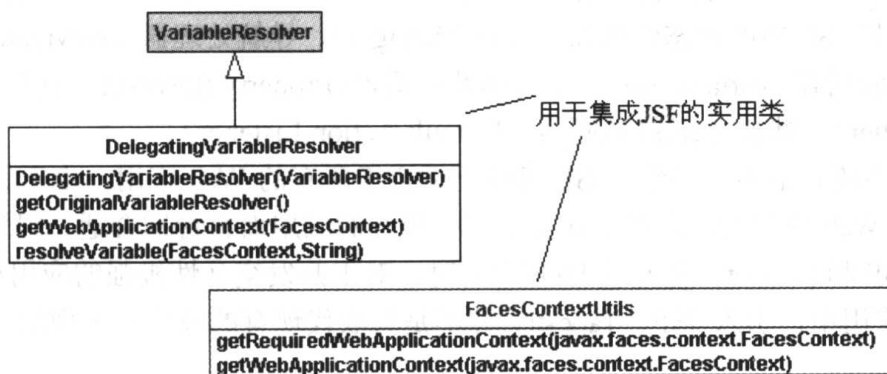


图 16-1 `org.springframework.web.jsf` 内容

其中, `DelegatingVariableResolver` 提供的 `resolveVariable()` 方法片断如下。可以看出, 它会首先通过 `javax.faces.el.VariableResolver` 提供的 `resolveVariable()` 方法查找受管 JavaBean, 即在 JSF 中查找。如果没有找到, 则将查找工作委派给 `WebApplicationContext`, 因此 Spring 开始起作用了。一旦找到, 则将返回 Spring 中的受管 JavaBean。

```
public Object resolveVariable(FacesContext facesContext, String name)
```

```
throws EvaluationException {
    // ask original resolver
    if (logger.isDebugEnabled()) {
        logger.debug("Attempting to resolve variable '" +
            name + "' in via original VariableResolver");
    }
    Object originalResult = this.originalVariableResolver.
        resolveVariable(facesContext, name);
    if (originalResult != null) {
        return originalResult;
    }

    // ask Spring root context
    if (logger.isDebugEnabled()) {
        logger.debug("Attempting to resolve variable '" + name +
            "' in root WebApplicationContext");
    }
    WebApplicationContext wac = getWebApplicationContext(facesContext);
    if (wac.containsBean(name)) {
        if (logger.isDebugEnabled()) {
            logger.debug("Successfully resolved variable '"
                + name + "' in root WebApplicationContext");
        }
        return wac.getBean(name);
    }

    if (logger.isDebugEnabled()) {
        logger.debug("Could not resolve variable '" + name + "'");
    }
    return null;
}
```

另外, JSF-Spring 项目也为集成 JSF, 而提供了相应的实现。借助于 JSF-Spring 中提供的 `FacesSpringVariableResolver` 变量解析器, 能够实现 Spring JSF 集成支持同样的功能。

- `de.mindmatters.faces.spring.FacesSpringVariableResolver`: JSF 变量解析器。它能够替换 JSF 默认使用的 `javax.faces.el.VariableResolver`。同 `DelegatingVariableResolver` 类似功能类似。

16.4 example29 实例研究

本书提供的 example29, 在功能上是对 example11 的简化、在 Web 视图层采用了 JSF (替换了 Tapestry Web 框架)。其中, Tapestry/JSF 都是基于事件模式开发 Web 应用的框架。本书接下来从 example29 的部署、使用、开发过程等方面进行阐述。

16.4.1 部署及使用

首先，开发者在部署 example29 之前需要准备好 example29 Web 应用使用到的 jar 库，其具体包含的 Java jar 库见图 16-2。

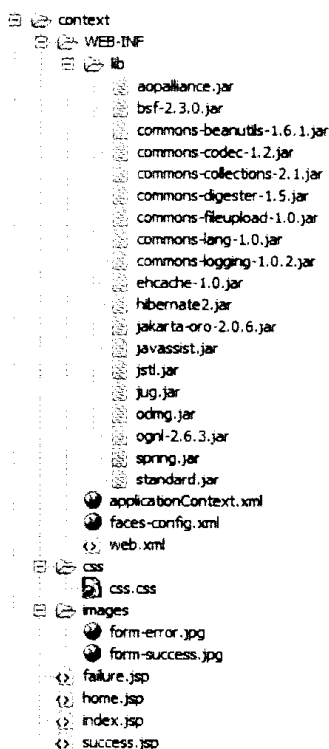


图 16-2 example29 JSF Web 应用结构

其次，开发者需要将 JBoss MySQL DataSource 配置文件配置好。比如，本书推荐的配置内容如下，即 D:\jboss-4.0.0\server\default\deploy\mysql-ds.xml。

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

<!-- $Id: mysql-ds.xml,v 1.3 2004/09/15 14:37:40 loubiansky Exp $ -->
<!-- Datasource config for MySQL using 3.0.9 available from:
http://www.mysql.com/downloads/api-jdbc-stable.html
-->

<datasources>
  <local-tx-datasource>
    <jndi-name>MySQLDS</jndi-name>
    <connection-url>jdbc:mysql://127.0.0.1:3306/example29</connection-url>
    <driver-class>com.mysql.jdbc.Driver</driver-class>
    <user-name>root</user-name>
    <password></password>
    <metadata>
      <type-mapping>mySQL</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>
```



```
</local-tx-datasource>
```

```
</datasources>
```

至于其中使用的 `connection-url`、`user-name` 及 `password`，开发者可以根据实际产品开发环境作相应的调整。然后，开发者还需要创建 MySQL `example29` 数据库，见 `other` 目录中的 `example29.mysql.sql`。

第三，提供 Ant `build.xml` 文件。如果开发者的 JBoss 4.0.0 服务器不是安装于 `D:\` 目录下，则还需调整 `jboss.dir` 的值。

```
<project name="JSF/Spring/Hibernate Example"
  default="war" basedir="." >

  <property name="war.file" value="example29.war"/>

  <property name="jboss.dir" value="D:/jboss-4.0.0"/>

  <target name="war">
    <description>Generate WAR File.</description>
    <war warfile="${war.file}"
      webxml="context/WEB-INF/web.xml">
      <fileset dir="context"/>
    </war>
  </target>

  <target name="deploy" depends="war">
    <description>Deploy The Generated WAR File
      To JBoss Application Server.</description>
    <copy file="${war.file}"
      todir="${jboss.dir}/server/default/deploy"/>
  </target>

</project>
```

第四，在编译整个 `example29` Eclipse 项目后，开发者可以运行 Ant `deploy` 任务，见图 16-3。

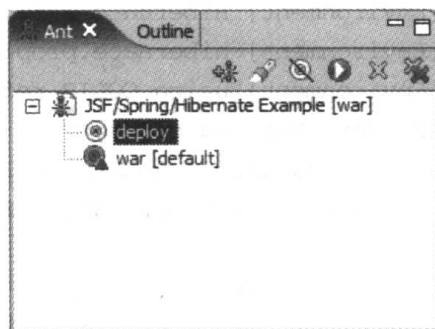


图 16-3 example29 Ant deploy 任务

在部署同时，JBoss 控制台将输出如下信息。通过分析日志，能够浏览到整个应用在启

动过程中所完成的部署内容。这对于学习 Spring, 或者说任何 Open Source 框架都有积极的作用。

```
21:42:38,317 INFO [TomcatDeployer] deploy, ctxPath=/example29, warUrl=
file:/D:/jboss-4.0.0/server/default/tmp/deploy/tmp43594example29-exp.war/
21:42:41,712 INFO [Engine] StandardContext[/example29]Loading root
WebApplicationContext
21:42:41,942 INFO [XmlBeanDefinitionReader] Loading XML bean definitions from
resource [/WEB-INF/applicationContext.xml] of ServletContext
21:42:42,423 INFO [XmlWebApplicationContext] Bean factory for application
context [Root XmlWebApplicationContext]: org.springframework.beans.factory.
support.DefaultListableBeanFactory defining beans [dataSource, sessionFactory,
transactionManager, example29ServiceTarget, example29Service, userinfoDAO];
root of BeanFactory hierarchy
21:42:42,453 INFO [XmlWebApplicationContext] 6 beans defined in application
context [Root XmlWebApplicationContext]
21:42:42,473 INFO [XmlWebApplicationContext] No message source found for
context [Root XmlWebApplicationContext]: using empty default
21:42:42,493 INFO [XmlWebApplicationContext] No ApplicationEventMulticaster
found for context [Root XmlWebApplicationContext]: using default
21:42:42,503 INFO [UiApplicationContextUtils] No ThemeSource found for [Root
XmlWebApplicationContext]: using ResourceBundleThemeSource
21:42:42,543 INFO [XmlWebApplicationContext] Refreshing listeners
21:42:42,553 INFO [DefaultListableBeanFactory] Pre-instantiating singletons
in factory [org.springframework.beans.factory.support.
DefaultListableBeanFactory defining beans [dataSource, sessionFactory,
transactionManager, example29ServiceTarget, example29Service, userinfoDAO];
root of BeanFactory hierarchy]
21:42:42,553 INFO [DefaultListableBeanFactory] Creating shared instance of
singleton bean 'dataSource'
21:42:42,774 INFO [JndiObjectFactoryBean] Located object with JNDI name
[java:/MySqlDS]: value=[org.jboss.resource.adapter.jdbc.
WrapperDataSource@14d0e66]
21:42:42,774 INFO [DefaultListableBeanFactory] Creating shared instance of
singleton bean 'sessionFactory'
21:42:42,884 INFO [Environment] Hibernate 2.1.5
21:42:42,894 INFO [Environment] hibernate.properties not found
21:42:42,904 INFO [Environment] using CGLIB reflection optimizer
21:42:43,114 INFO [Binder] Mapping class:
com.openv.spring.service.hibernate.UserInfo -> userinfo
21:42:43,324 INFO [LocalSessionFactoryBean] Building new Hibernate
SessionFactory
21:42:43,324 INFO [Configuration] processing one-to-many association mappings
21:42:43,324 INFO [Configuration] processing one-to-one association property
references
21:42:43,324 INFO [Configuration] processing foreign key constraints
21:42:43,425 INFO [Dialect] Using dialect: net.sf.hibernate.dialect.MySQLDialect
```

```
21:42:43,435 INFO [SettingsFactory] Use outer join fetching: false
21:42:43,435 INFO [ConnectionProviderFactory] Initializing connection provider:
org.springframework.orm.hibernate.LocalDataSourceConnectionProvider
21:42:43,455 INFO [TransactionManagerLookupFactory] No
TransactionManagerLookup configured (in JTA environment, use of process level
read-write cache is not recommended)
21:42:43,975 INFO [SettingsFactory] Use scrollable result sets: true
21:42:43,975 INFO [SettingsFactory] Use JDBC3 getGeneratedKeys(): true
21:42:43,975 INFO [SettingsFactory] Optimize cache for minimal puts: false
21:42:43,975 INFO [SettingsFactory] echoing all SQL to stdout
21:42:43,975 INFO [SettingsFactory] Query language substitutions: {}
21:42:43,975 INFO [SettingsFactory] cache provider:
net.sf.ehcache.hibernate.Provider
21:42:44,015 INFO [Configuration] instantiating and configuring caches
21:42:44,406 INFO [SessionFactoryImpl] building session factory
21:42:45,027 INFO [SessionFactoryObjectFactory] no JNDI name configured
21:42:45,027 INFO [DefaultListableBeanFactory] Creating shared instance of
singleton bean 'transactionManager'
21:42:45,137 INFO [HibernateTransactionManager] Using DataSource
[org.jboss.resource.adapter.jdbc.WrapperDataSource@14d0e66] of Hibernate
SessionFactory for HibernateTransactionManager
21:42:45,137 INFO [DefaultListableBeanFactory] Creating shared instance of
singleton bean 'example29ServiceTarget'
21:42:45,137 INFO [Example29ManagerImpl]
Example29ManagerImpl() .....
21:42:45,157 INFO [DefaultListableBeanFactory] Creating shared instance of
singleton bean 'userinfoDAO'
21:42:45,287 INFO [XmlBeanDefinitionReader] Loading XML bean definitions from
class path resource [org/springframework/jdbc/support/sql-error-codes.xml]
21:42:45,317 INFO [XmlBeanFactory] Creating shared instance of singleton bean 'DB2'
21:42:45,337 INFO [XmlBeanFactory] Creating shared instance of singleton bean 'HSQL'
21:42:45,337 INFO [XmlBeanFactory] Creating shared instance of singleton bean 'MS-SQL'
21:42:45,337 INFO [XmlBeanFactory] Creating shared instance of singleton bean 'MySQL'
21:42:45,337 INFO [XmlBeanFactory] Creating shared instance of singleton bean 'Oracle'
21:42:45,337 INFO [XmlBeanFactory] Creating shared instance of singleton bean
'Informix'
21:42:45,347 INFO [XmlBeanFactory] Creating shared instance of singleton bean
'PostgreSQL'
21:42:45,347 INFO [XmlBeanFactory] Creating shared instance of singleton bean 'Sybase'
21:42:45,347 INFO [SQLExceptionCodesFactory] SQLExceptionCodes loaded: [HSQLDatabase
Engine, Oracle, Sybase SQL Server, Microsoft SQL Server, Informix Dynamic Server,
PostgreSQL, DB2*, MySQL]
21:42:45,347 INFO [SQLExceptionCodesFactory] Looking up default SQLExceptionCodes for
DataSource
21:42:45,397 INFO [DefaultListableBeanFactory] Creating shared instance of
singleton bean 'example29Service'
```

```
21:42:46,749 INFO [CollectionFactory] Using JDK 1.4 collections
21:42:47,030 INFO [ContextLoader] Using context class
[org.springframework.web.context.support.XmlWebApplicationContext] for root
WebApplicationContext
21:42:47,030 INFO [ContextLoader] Published root WebApplicationContext
[org.springframework.web.context.support.XmlWebApplicationContext:
displayName=[Root XmlWebApplicationContext]; startup date=[Sun Nov 21 21:42:41
CST 2004]; root of ApplicationContext hierarchy; config
locations=[/WEB-INF/applicationContext.xml]; ] as ServletContext attribute
with name [interface
org.springframework.web.context.WebApplicationContext.ROOT]
```

第五,通过浏览器,开发者能够访问到 example29,比如 <http://localhost:8080/example29/>。其主界面如图 16-4 所示。



图 16-4 example29 主界面

用户在输入相关信息并提交表单后,如果用户名域不符合要求,即用户名的长度必须处于 6~18 个字符之间,则将出现图 16-5 的提示信息界面。

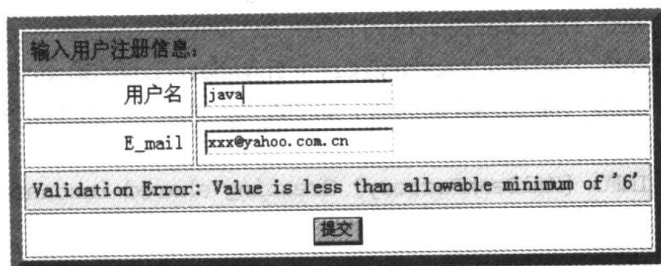


图 16-5 example29 页面错误信息提示

一旦输入的所有域能够满足应用的需求(比如,图 16-6),则在经过 Spring 处理后,将会把用户注册信息填充到 RDBMS 中。

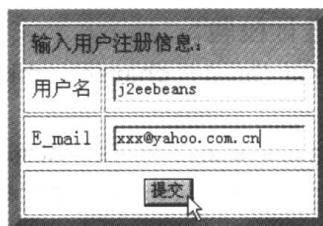


图 16-6 正确输入 example29 所要求的表单域

如果操作成功,则提示用户注册成功,见图 16-7。如果操作失败(比如,MySQL RDBMS 未启动),则提示用户注册失败,见图 16-8。

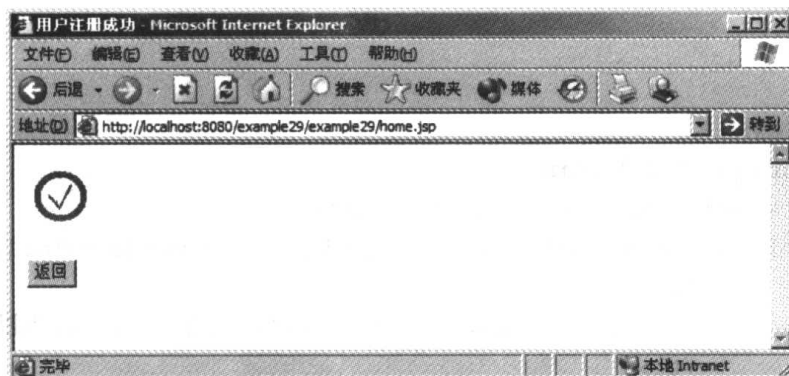


图 16-7 用户注册成功

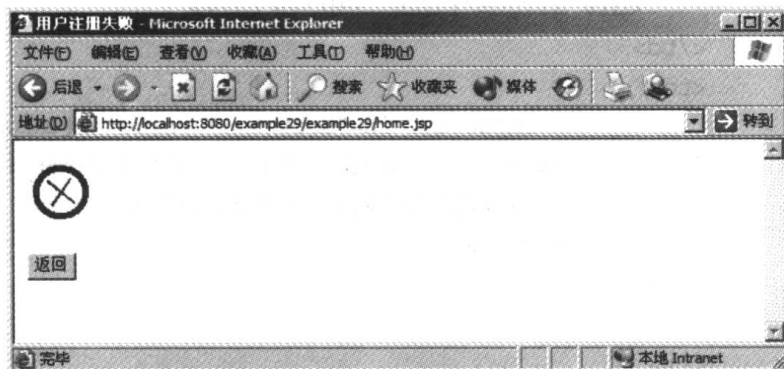


图 16-8 用户注册失败

通过 MySQL 控制台,开发者能够对用户注册信息进行确认。

16.4.2 开发过程

本实例同 example11 的架构相同,只是在 Web 层使用了 JSF,而不是 Tapestry。

对于 Web 层而言,具体 JSF 内容如下。

第一,开发者需要开发 example29 主页面,即 home.jsp。请开发者注意<f:view>标签库,它在 JSF 中占有很重要的位置。通过 taglib 能够引入 JSF 中的标签库。JSF 提供了 HTML 和 CORE 两套标签库。可以看到,username、email 输入域是分别通过 InfoBean.username 和 InfoBean.email 进行值引用的。另外,username 还存在校验器,即通过 minimum 和

maximum 属性域对 username 长度进行校验。

```
<%@ page contentType="text/html; charset=gbk" %>

<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<html>
<head>
<title>
用户注册
</title>
</head>
<br>
<f:view>
<h:form id="helloForm" >
    <table border="10" align="center"
        bordercolor="#0099CC" cellpadding="6" bordercolorlight="#999999">
        <tr>
            <td colspan="2" bgcolor="#66CCFF">输入用户注册信息: </td>
        </tr>
        <tr>
            <td>
                <div align="right">用户名</div>
            </td>
            <td>
                <h:inputText id="username" value="#{InfoBean.username}">
                <f:validateLength minimum="#{InfoBean.minimum}"
                    maximum="#{InfoBean.maximum}" />
                </h:inputText>
            </td>
        </tr>
        <tr>
            <td>
                <div align="right">E_mail</div>
            </td>
            <td>
                <h:inputText id="email" value="#{InfoBean.email}"/>
            </td>
        </tr>
        <tr>
            <td colspan="2" bgcolor="#FFFF40">
                <span>
                    <h:message id="message"
                        for="username" /></span>
                </td>
        </tr>
    </table>
</f:view>
```

```

<tr>
    <td align="center" colspan="2">
        <h:commandButton id="submit"
//action 是带触发的 Web 事件，这同传统的 JSP 存在很大区别
        action="#{InfoBean.submitPersonInfo}" value="提交" />
    </td>
</tr>
</table>
</h:form>
</f:view>
</html>

```

其中，“提交”按钮将触发后台的事件，即受管 InfoBean 中的 submitPersonInfo 方法。可以看出，submitPersonInfo 正是 JSF 中事件驱动的体现。用户每次对表单的操作都将涉及到不同（相同）的事件，而这些事件通常都可以通过 Java 方法体现。

第二，开发注册用户成功和失败的提示界面。用户成功注册提示页面的片断如下。借助于 graphicImage 和 outputText 组件（标签）能够显示图片和文字信息。借助于 commandButton 组件能够带领用户重回用户注册页面。

```

<f:view>
    <h:form id="responseForm">
        <h:graphicImage id="successImg"
            url="images/form-success.jpg" alt="注册成功! " />
        <h2>
            <h:outputText id="result"
                value="#{InfoBean.response}" /></h2>
        <h:commandButton id="back"
            value="返回" action="su" />
        <p>
    </h:form>
</f:view>

```

第三，开发 JSF 受管的后端 JavaBean，即 InfoBean.java。InfoBean 定义了 username、email 属性，用于存储用户名和 Email。而且，minimum 和 maximum 属性供校验 username 域使用。InfoBean 定义的 submitPersonInfo 方法能够完成表单数据的处理，期间使用到了 Spring 提供的受管 JavaBean。

```

package com.openv.spring.jsf;

import com.openv.spring.domainmodel.UserInfoVO;
import com.openv.spring.service.IExample29Manager;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.springframework.context.ApplicationContext;

import org.springframework.web.jsf.FacesContextUtils;

```

```
import javax.faces.context.FacesContext;

/**
 * InfoBean, 供处理 home.jsp 事件使用。
 *
 * @author luoshifei
 */
public class InfoBean {
    private static final Log log = LogFactory.getLog(InfoBean.class);

    private String username = null;

    private String email = null;

    private String response = null;

    private long maximum = 0;

    private boolean maximumSet = false;

    private long minimum = 0;

    private boolean minimumSet = false;

    public InfoBean() {
    }

    /**
     * @return Returns the email.
     */
    public String getEmail() {
        return email;
    }

    /**
     * @param email
     *         The email to set.
     */
    public void setEmail(String email) {
        this.email = email;
    }

    /**
     * @return Returns the username.
     */
}
```



```
public String getUsername() {
    return username;
}

/**
 * @param username
 *      The username to set.
 */
public void setUsername(String username) {
    this.username = username;
}

/**
 * 用于处理用户触发的事件
 *
 * @return <from-outcome>逻辑名
 */
public String submitPersonInfo() {
    log.info(username);
    log.info(email);

    ApplicationContext ac = FacesContextUtils
        .getWebApplicationContext(FacesContext.getCurrentInstance());
    IExample29Manager em = (IExample29Manager) ac
        .getBean("example29Service");
    UserInfoVO uiVO = new UserInfoVO();
    uiVO.setUsername(username);
    uiVO.setEmail(email);

    boolean flag = em.setUserInfo(uiVO);

    if (flag) {
        setResponse("注册成功");

        return "success";
    } else {
        setResponse("注册失败");

        return "failure";
    }
}

/**
 * @param response
 *      The response to set.
 */
```

```
public void setResponse(String response) {
    this.response = response;
}

public String getResponse() {
    return null;
}

public long getMaximum() {
    return (this.maximum);
}

public void setMaximum(long maximum) {
    this.maximum = maximum;
    this.maximumSet = true;
}

public long getMinimum() {
    return (this.minimum);
}

public void setMinimum(long minimum) {
    this.minimum = minimum;
    this.minimumSet = true;
}
}
```

第四, 编写 JSF faces-config.xml 配置文件, 从而将上述定义的所有内容集成起来。JSF 提供的导航处理器相当灵活。如果用户注册成功, 则将返回 success.jsp 页面; 如果失败, 将返回 failure.jsp 页面。可以看到, InfoBean 也定义在其中。

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE faces-config PUBLIC
    "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
    "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
<faces-config>

    <application>
        <locale-config>
            <default-locale>zh_CN</default-locale>
        </locale-config>
    </application>

    <navigation-rule>
        <description>
            JSF Home Page
        </description>
        <from-view-id>/home.jsp</from-view-id>
```

```
<navigation-case>
  <description>
    success
  </description>
  <from-outcome>success</from-outcome>
  <to-view-id>/success.jsp</to-view-id>
</navigation-case>
<navigation-case>
  <description>
    failure
  </description>
  <from-outcome>failure</from-outcome>
  <to-view-id>/failure.jsp</to-view-id>
</navigation-case>
</navigation-rule>

<navigation-rule>
  <description>
  </description>
  <from-view-id>/success.jsp</from-view-id>
  <navigation-case>
    <description>
    </description>
    <from-outcome>su</from-outcome>
    <to-view-id>/home.jsp</to-view-id>
  </navigation-case>
</navigation-rule>

<navigation-rule>
  <description>
  </description>
  <from-view-id>/failure.jsp</from-view-id>
  <navigation-case>
    <description>
    </description>
    <from-outcome>su</from-outcome>
    <to-view-id>/home.jsp</to-view-id>
  </navigation-case>
</navigation-rule>

<managed-bean>
  <description>
    InfoBean
  </description>
  <managed-bean-name>InfoBean</managed-bean-name>
  <managed-bean-class>
```

```
com.openv.spring.jsf.InfoBean
</managed-bean-class>

<managed-bean-scope>session</managed-bean-scope>
<managed-property>
  <property-name>minimum</property-name>
  <property-class>long</property-class>
  <value>6</value>
</managed-property>

<managed-property>
  <property-name>maximum</property-name>
  <property-class>long</property-class>
  <value>18</value>
</managed-property>
</managed-bean>
</faces-config>
```

第五，配置 web.xml 部署描述符。开发者需要配置 JSF 中的 FacesServlet 前端控制器。因此，这使得整个应用是 JSF 驱动的。

```
<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">

  <display-name>example29</display-name>

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

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

  <servlet>
    <display-name>FacesServlet</display-name>
    <servlet-name>FacesServlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
```

```

<servlet-name>FacesServlet</servlet-name>
<url-pattern>/example29/*</url-pattern>
</servlet-mapping>

```

```

</web-app>

```

通过上述过程，开发者已经完成了 JSF 相关内容的开发和配置。对于业务逻辑层而言，比如 DAO 相关内容，开发者需要实现如下内容。

开发者借助于基于 XSLT 的 SpringViz 工具，能够将 example29 中的 Spring 配置文件，即 applicationContext.xml 以图形化的方式（见图 16-9，这同 example11 中 Spring IDE 提供的图形化意义相同。）将其层次结构展现出来，这对于那些含有大量配置信息的 Spring 配置文件而言特别有意义。

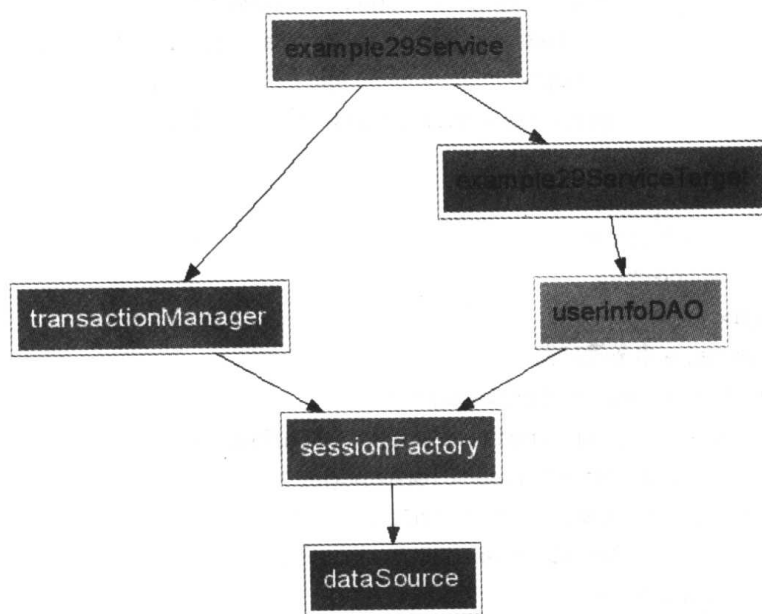


图 16-9 借助于 SpringViz 工具展示 applicationContext.xml

其中，applicationContext.xml 的配置内容如下。

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
//定义数据源
    <bean id="dataSource"
        class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName">
            <value>java:/MySqlDS</value>
        </property>
    </bean>
//定义Hibernate SessionFactory
    <bean id="sessionFactory"

        class="org.springframework.orm.hibernate.LocalSessionFactoryBean">

```

```
<property name="dataSource">
    <ref local="dataSource"/>
</property>
<property name="mappingResources">
    <list>
        <value>
            com/opencv/spring/service/hibernate/UserInfo.hbm.xml
        </value>
    </list>
</property>
<property name="hibernateProperties">
    <props>
        <prop key="hibernate.dialect">
            net.sf.hibernate.dialect.MySQLDialect
        </prop>
        <prop key="hibernate.show_sql">
            true
        </prop>
    </props>
</property>
</bean>
//定义Hibernate 事务管理器
<bean id="transactionManager"
    class="org.springframework.orm.hibernate.
        HibernateTransactionManager">
    <property name="sessionFactory">
        <ref local="sessionFactory"/>
    </property>
</bean>
//定义业务逻辑委派
<bean id="example29ServiceTarget"
    class="com.opencv.spring.service.impl.Example29ManagerImpl">
    <property name="userinfo">
        <ref local="userinfoDAO"/>
    </property>
</bean>
//定义事务代理工厂。这将使用到 Spring AOP 提供的功能
<bean id="example29Service"
    class="org.springframework.transaction.
        interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager">
        <ref local="transactionManager"/>
    </property>
    <property name="target">
        <ref local="example29ServiceTarget"/>
    </property>
```

```

        <property name="transactionAttributes">
            <props>
                <prop key="get *">
                    PROPAGATION_REQUIRED,readOnly
                </prop>
                <prop key="set *">
                    PROPAGATION_REQUIRED
                </prop>
            </props>
        </property>
    </bean>
//定义业务逻辑实现
    <bean id="userinfoDAO"
        class="com.openv.spring.service.dao.impl.UserInfoDAO">
        <property name="sessionFactory">
            <ref local="sessionFactory"/>
        </property>
    </bean>

</beans>

```

其中, userinfoDAO JavaBean 的内容如下。

```

package com.openv.spring.service.dao.impl;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.doomdark.uuid.UUIDGenerator;
import org.springframework.dao.DataAccessException;
import org.springframework.orm.hibernate.support.HibernateDaoSupport;

import com.openv.spring.domainmodel.UserInfoVO;
import com.openv.spring.service.dao.IUserInfoDAO;
import com.openv.spring.service.hibernate.UserInfo;

/**
 * IUserInfoDAO 实现。处理用户相关信息的 DAO 实现。比如, 存储用户注册信息。
 *
 * @author luoshifei
 */
public class UserInfoDAO extends HibernateDaoSupport implements IUserInfoDAO
{
    private static final Log log = LogFactory.getLog(UserInfoDAO.class);

    /**
     * 存储用户注册信息
     *
     * @param userinfoVO
     */
}

```

```
*          用户注册信息
*
* @return boolean 存储用户注册信息是否成功
*/
public boolean setUserInfo(UserInfoVO userinfoVO)
    throws DataAccessException {
    if (userinfoVO == null) {
        return false;
    }

    UserInfo ui = new UserInfo();
    ui.setId(getID());
    ui.setUsername(userinfoVO.getUsername().trim());
    ui.setEmail(userinfoVO.getEmail().trim());
    this.getHibernateTemplate().save(ui);

    return true;
}

/**
 * 模拟唯一 ID 的生成
 *
 * @return
 */
private String getID() {
    return
        UUIDGenerator.getInstance().generateTimeBasedUUID().toString();
}
}

example29ServiceTarget JavaBean 的内容如下。
package com.openv.spring.service.impl;

import com.openv.spring.domainmodel.UserInfoVO;
import com.openv.spring.service.IExample29Manager;
import com.openv.spring.service.dao.IUserInfoDAO;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.springframework.dao.DataAccessException;

/**
 * IExample29Manager 实现
 *
 * @author luoshifei
 */
```



```

public class Example29ManagerImpl implements IExample29Manager {
    private static final Log log = LogFactory
        .getLog(Example29ManagerImpl.class);

    private IUserInfoDAO userinfo;

    public Example29ManagerImpl() {
        log.info("Example29ManagerImpl().....");
    }

    public void setUserinfo(IUserInfoDAO userinfoDAO)
        throws DataAccessException {
        this.userinfo = userinfoDAO;
    }

    public boolean setUserinfo(UserInfoVO userinfoVO)
        throws DataAccessException {
        return userinfo.setUserinfo(userinfoVO);
    }
}

```

因此，可以看出在使用 Spring IoC 和 Spring AOP 的前提下，即使更换 Web 视图层技术实现，对于其他层（包括业务层和持久层）的影响不大，甚至可以说没有任何影响。

16.4.3 Spring 提供的 JSF 集成能力

开发者应该注意到 InfoBean.java 中的如下内容。

```

ApplicationContext ac = FacesContextUtils
    .getWebApplicationContext(FacesContext.getCurrentInstance());
IExample29Manager em = (IExample29Manager) ac
    .getBean("example29Service");

```

通过它们，开发者能够获得当前 FacesContext 实例中的 Spring WebApplicationContext。这同 Tapestry 很相似。当然，这种方式并没有使用到 Spring 提供的 VariableResolver。如果需要借助于 DelegatingVariableResolver，则开发者须遵循 16.4.4 节给出的步骤。当然，如果借助于 JSF-Spring 项目提供的 jar 库，则开发 example29 的过程也很类似。

16.4.4 JSF-Spring 项目提供的 JSF 集成能力

借助于 JSF-Spring 项目，开发者可以直接使用它为 JSF 提供的变量解析器，即 de.mindmatters.faces.spring.FacesSpringVariableResolver 类。具体步骤如下。

第一，需要修改 InfoBean.java 文件。添加 example29Manager 属性，其类型为 IExample29Manager。为了设置 example29Manager，还需提供 setter 和 getter 方法。

```

private IExample29Manager example29Manager;

/**

```

```

    * @return Returns the example29Manager.
    */
    public IExample29Manager getExample29Manager() {
        return example29Manager;
    }
    /**
    * @param example29Manager The example29Manager to set.
    */
    public void setExample29Manager(IExample29Manager example29Manager) {
        this.example29Manager = example29Manager;
    }

```

当然，开发和还需修改对 **example29Manager** 的引用。

```

//      ApplicationContext ac = FacesContextUtils
//      .getWebApplicationContext(FacesContext.getCurrentInstance())
//      );
//      IExample29Manager em = (IExample29Manager) ac
//      .getBean("example29Service");
//      UserInfoVO uiVO = new UserInfoVO();
//      uiVO.setUsername(username);
//      uiVO.setEmail(email);

//      boolean flag = em.setUserInfo(uiVO);
//      boolean flag = example29Manager.setUserInfo(uiVO);

```

第二，修改 **faces-config.xml** 配置文件。此时，开发者需要为 **InfoBean** 添加新的属性，即 **example29Manager**。

```

<managed-property>
    <property-name>example29Manager</property-name>
    <value>#{example29Service}</value>
</managed-property>

```

第三，新增 JSF 变量解析器的设置。

```

<application>
    <locale-config>
        <default-locale>zh_CN</default-locale>
    </locale-config>
    <variable-resolver>
de.mindmatters.faces.spring.FacesSpringVariableResolver
    </variable-resolver>
    <!--
        <variable-resolver>
org.springframework.web.jsf.DelegatingVariableResolver
    </variable-resolver>
    -->
</application>

```

如果需要使用 Spring 提供的 `DelegatingVariableResolver`¹¹, 则将注释部分替代成 JSF-Spring 提供的 `FacesSpringVariableResolver`。

至此, 整个 JSF 集成过程结束。

16.5 小 结

本章内容结合 example29 实例, 详细阐述了开发基于 JSF/Spring/Hibernate 应用的过程。开发者应该了解到, 开发基于 Tapestry/Spring/Hibernate 应用 (比如, example11) 的过程同 example29 类似。同时, 开发者也熟悉了 Spring 和 JSF-Spring 项目对 JSF 提供的集成支持。

这种基于事件、组件开发 Web 前端的框架技术将慢慢走向主流。

开发者已经深入了解到基于 Tapestry/JSF 框架技术开发 Web 应用所带来的冲击。本书也相信在未来的一段时间, Tapestry/JSF 将走向成熟、走向成功!

¹¹ 在本书写作之际, Spring 提供的 `DelegatingVariableResolver` 还不能够正常工作。日后, 希望 JSF 和 Spring 能够改进这方面的工作。另外, 如果开发者需要 JSF-Spring 提供的 Demo 配置, 则通过如下网址能够获得其提供的 Web 示例应用: http://sourceforge.net/project/showfiles.php?group_id=107519。

第 17 章 用于 Spring 的 Acegi

安全框架

安全性往往是企业应用开发中最为重要的问题之一。基于 JAAS 提供的用户认证和授权服务能够较好地解决安全性问题，但是这并不完整。很多应用系统要求用户登录，并且基于用户的角色而授予适当的权限，比如对一些敏感资源的受限访问。通过访问控制列表（Access Control List, ACL）能够有效地实现对资源的保护。因此，需要找到一种更好的替代方案。

Acegi¹安全框架，正是为 Spring 设计的。现有的 Spring 框架中并没有含有 J2EE 安全性相关的抽象，而且安全性往往认为是不与业务逻辑相关的，即不能够将应用的安全性需求同业务逻辑编写在一块。由于安全性很重要，Spring 专门通过单独的 Acegi 安全框架来实现应用的安全。可以看出，Spring 对安全的重视程度。

本章将结合 Acegi 安全框架附带的经典 contacts 实例展开论述。

17.1 Acegi 介绍

为 Spring 使能应用提供认证和授权功能，这正是 Acegi 安全框架的初衷。当前，Acegi 同主流的 Web 容器作了出色的集成工作，因此开发者再也不用担心的适用范围了。Spring 倡导的 Bean 上下文、拦截器及针对接口编程等使用模式都被 Acegi 采纳了。尽管开发者可以独立于 Spring 使用 Acegi，但是不可否认 Acegi 的使用者主要是存在 Spring 开发背景的用户。因此，这大大降低了 Spring 开发者的学习曲线，而且符合 Spring 社区的一贯做法。

通常，任何系统的安全都会涉及到两方面的内容，即认证和授权，这也是 JAAS 所支持的。认证，即判断用户是否是其宣称的；而授权，即决定已认证用户是否有权访问或操作目标资源（比如，某 HTML 资源或者某业务对象的特定方法）。这同 EJB 使用的企业级安全性服务类似，但 Acegi 针对的主要是 POJO 对象（对于方法调用而言，本书在后续内容讲述），这也体现出了 Acegi 的轻量特质。

最主要的一点是由 Open Source 社区开发的、并为 Spring 使能应用量身定做的 Acegi 易于使用、功能丰富。

开发者通过 <http://acegisecurity.sourceforge.net/>能够下载到 Acegi 的最新版，图 17-1 给出了其项目主页。

¹ <http://acegisecurity.sourceforge.net/>

SOURCEFORGE
netAcegi
**SECURITY
SYSTEM**
FOR SPRING

Last published: 04 January 2005 | Doc for 0.7.0-SNAPSHOT

Acegi Security on Sourceforge

Overview

- Home
- Building with Haven
- Downloads

Documentation

- Suggested Steps
- Reference Guide
- Sample SQL Schema
- FAC
- External Web Articles
- Upgrading to 0.7.0
- Upgrading to 0.5
- Upgrading to 0.4

Projects

- Core Framework
- CAS Adapter
- Catalina Adapter
- JBoss Adapter
- Jetty Adapter
- Resin Adapter

Samples

- Contacts
- Attributes

Search Acegi Security System for Spring

Google

Go

Project Documentation

- About Acegi Security System for Spring
- Uninstall
- Project Info
- Project Reports
- Development Process

Mission Statement

To provide comprehensive security services for The Spring Framework.

Key Features

- It is ready NOW.** As explained in the reference guide, the API is now quite stable. We also use the Apache APB Project Versioning Guidelines so you can identify backward compatibility.
- Fast results:** View our suggested steps for the fastest way to develop complex, security-compliant applications.
- Enterprise-wide single sign on:** Using Yale University's open source Central Authentication Service (CAS), the Acegi Security System for Spring can participate in an enterprise-wide single sign on environment. You no longer need every web application to have its own authentication database. Nor are you restricted to single sign on across a single web container. Advanced single sign on features like proxy support and forced refresh of logins are supported by both CAS and Acegi Security.

图 17-1 Acegi 项目主页

Acegi 的主要特征有:

- 易于使用: Acegi 本身的易用性, 加上现有的丰富文档使得开发者能够快速开发出业务复杂、安全性好的应用系统。
- 企业级的单点登录 (Single Sign On, SSO) 能力。Acegi 通过使用 Yale 大学的 CAS (Central Authentication Service) 使得 Spring 使能应用能够参与到企业级的 SSO 环境中。借助于 SSO, Web 应用再也不需要维护自身的认证信息资料了。因此, CAS 的强大功能都能够在 Acegi 中淋漓尽致地流露出来。
- 重用 Spring 开发经验。由于 Acegi 使用 Spring ApplicationContext (BeanFactory) 配置其本身, 因此开发者能够快速开发 Spring 使能应用的安全性功能。
- 对业务域对象实例的保护。很多的企业应用都要求为单个的域对象实例定义授权信息, 即访问控制列表 (Access Control List, ACL), 借助于 Acegi 能够满足这种需求。借助于 Spring AOP 的支持, Acegi 能够在对象实例的方法级对用户进行认证和授权处理, 而且整个过程是声明式的, 即开发者不用开发任何安全性代码。
- 非侵入式设置。借助于 Acegi 提供的过滤器, 开发者能够在 Web 应用中使用其提供的的安全性功能。其中, 不需要在目标 Servlet 或 EJB 容器中部署新的 jar 包或者修改特定的配置。这在很大程度上有利于 Acegi 使能应用的便携性。
- 对容器提供了一流的集成支持。为收集用户凭证和授权信息, Acegi 能够直接使用 Servlet 或 EJB 容器提供的这方面能力, 比如借助于 JBoss 提供的 LoginModule 能够获得用户的安全性信息。这在很大程度上简化了 Acegi 的使用。当前, Acegi 支持 Tomcat、Jetty、JBoss、Resin 等容器。

- 独立于业务对象。这是很重要的设计准则。应用的安全性需求应该同业务需求分开。
- 处理调用后的安全性。Acegi 不仅能够保护客户调用目标方法的入口，它还能够处理被调用方法的返回值的的安全性。这同 AOP 中的 Around 装备类似。
- 保护 HTTP 请求。Acegi 不仅能够保护 Spring 使能应用中配置的 POJO，它还能够保护 HTTP 请求。开发者都知道，通过在 Web 应用中配置安全性约束能够保护 HTTP 请求。单独借助于 Acegi 也能够达到同样的目的。
- 对具体传输 Channel 的支持。比如，开发者能够借助于 HTTPS 实现安全性信息的传输。Acegi 支持插入式传输 Channel。
- 支持 HTTP BASIC 认证方式。
- 提供了安全性标签库。开发者能够在 JSP 页面中使用 Acegi 标签库，从而保护受限访问的 HTTP 链接。
- 支持不同的后端认证。比如，开发者可以从 XML 文件、RDBMS、LDAP，或者活动目录中获得安全性信息。当然，还可以使用自身的认证方式。
- 兼容于各种 RDBMS。如果基于 RDBMS 对用户信息进行认证和授权，则 Acegi 使能应用能够兼容于各种 RDBMS，而且 Acegi 对目标 RDBMS 的模式 (Schema) 和数据内容不做限定。当然，开发者也可以提供自身的 DAO 对象，供自身的业务需求使用。
- 提供缓存能力。借助于 Spring 提供的 EHCACHE 工厂，Acegi 能够应用缓存认证信息，而不用基于每次认证请求都去 RDBMS 或 LDAP 等持久化源中获得相关信息。
- 插件式架构。由于 Acegi 的各个组成部分设计得相当灵活 (比如，针对接口编程、高度抽象、松耦合)，开发者能够动态更换其组成模块。
- 应用启动期间对所有的 Acegi 安全性配置进行校验。这主要依赖于 Spring，因为 Spring BeanFactory (ApplicationContext) 能够自动完成上述任务，无论是业务对象，还是安全性配置信息。因此，可以降低应用运行期间的出错几率。
- 远程支持。由于 Acegi 的配置使用架构在 Spring 基础之上，而且它同 Spring 提供的远程服务进行了一流的集成，因此通过配置 web.xml 和 Spring 配置文件 (比如，applicationContext.xml)，开发者就能够自动处理 HTTP BASIC 认证头信息。
- 高级密码加密功能。Acegi 面对的是企业级应用，在这种场合很少存储明文密码信息，因此对高级密码加密功能的支持很重要。Acegi 支持 SHA 和 MD5 加密方式。当然，也允许插入其他供应商的加密方式。这些都体现出 Acegi 强大的可扩展性。
- Run-As 替换。这同标准的 J2EE 的安全性约定一致，即动态修改已认证用户，从而满足复杂的业务需求。
- 透明地进行安全性移植。如果开发者需要将安全性信息在物理机器间进行移植，比如使用 RMI 和 Spring 提供的远程服务 (HTTP Invoker)，则 Acegi 能够传送主要的认证信息。
- 兼容于 HttpServletRequest.getRemoteUser()。尽管 Acegi 安全框架借助于各种插入式机制分发安全性，但是它还是能够兼容于标准 J2EE 的安全性约定，即借助于 getRemoteUser() 方法访问已认证对象。

- 单元测试覆盖率高。这在很大程度上能够保证 Acegi 的质量。
- 基于 Apache 授权。

17.2 Acegi 架构及使用

借助于 Spring AOP 模块, Spring 提供的事务服务抽象能够以声明式实现事务管理。事务服务在 J2EE 平台中属于系统级服务。因此, Spring 将其提供的事务管理服务以 AOP Aspect 形式展现给 Spring 使用。相比之下, 安全性服务也是 J2EE 平台中的系统级服务, 尽管 Spring 本身没有提供安全性服务抽象, 但是 Acegi 却依据类似的原则, 将安全性服务抽象以 AOP Aspect 形式发布。总之, 从 Spring AOP 角度出发, 事务同安全性是一致的, 它们都是 Aspect。既然 Acegi 能够担此重任, 其具体实现 Aspect 的架构和使用又是如何呢?

为使得开发者更好地理解 Acegi 的架构和使用, 本章将结合 Acegi 附带的 contacts.war 实例展开论述, 这也符合本书的一贯做法, 即“事实胜于雄辩”。

17.2.1 构建 contacts 应用

开发者可以参考本书对 Web 应用(比如 example11 实例)的 Eclipse 项目设置, 在 Eclipse 中配置好 contacts 项目内容(注意, 这只是可选步骤), 具体见图 17-2。

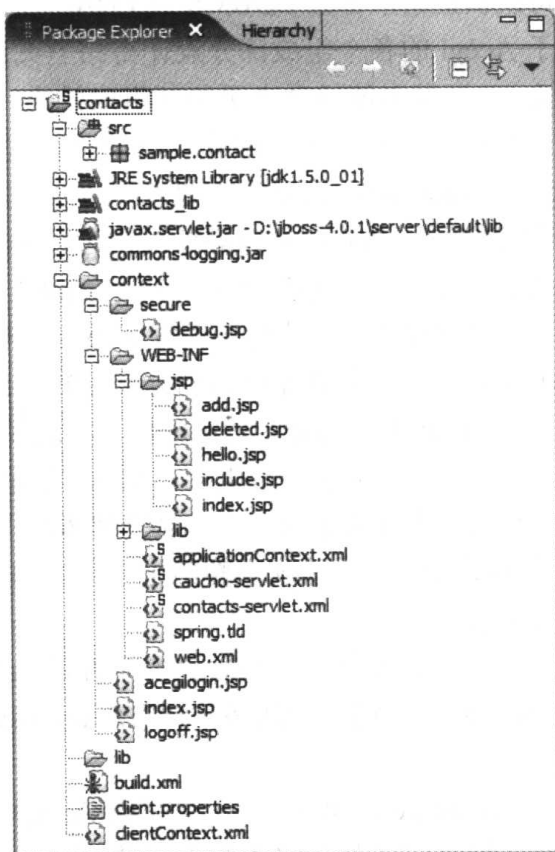


图 17-2 基于 Eclipse 的 contacts 项目结构

其中:

- src: 用于存放 contacts 项目中使用到的所有 Java 源文件。
- context: 用于存储 Web 应用的相关内容。其中, 这同 contacts.war 的结构安排一致。这使得整个 Web 应用的结构特别清晰。
- build.xml: 用于构建 contacts.war 的 Ant 文件, 其内容见图 17-3 (注意, 未包括 contacts 项目的编译任务)。
- client.properties 和 clientContext.xml: 供执行客户应用使用。

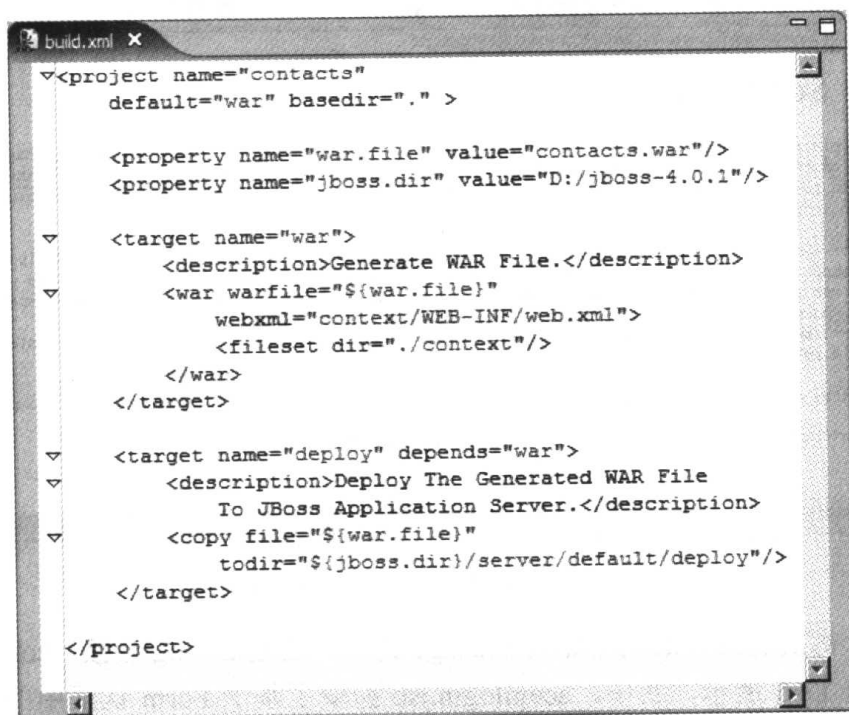


图 17-3 Ant build.xml 文件内容

当然, 如果开发者直接使用 contacts 附带的 build.xml 文件构建 contacts.war, 则可以不理睬本节介绍的方法(但通过手工配置 contacts 项目, 开发者能够理清楚 contacts 的各方面内容)。

在将 contacts.war 部署到 JBoss 应用服务器后, 开发者如果浏览到如下服务器日志(部分), 则表明部署成功。

```
.....
23:33:45,372 INFO [BeanNameUrlHandlerMapping] Mapped URL path
[/ContactManager-hessian] onto handler
[org.springframework.remoting.caucho.HessianServiceExporter@adf91]
23:33:45,372 INFO [BeanNameUrlHandlerMapping] Mapped URL path
[/ContactManager-burlap] onto handler
[org.springframework.remoting.caucho.BurlapServiceExporter@1dbe72f]
23:33:45,372 INFO [DispatcherServlet] No HandlerMappings found in servlet
'caucho': using default
23:33:45,372 INFO [DispatcherServlet] No HandlerAdapters found in servlet
'caucho': using default
23:33:45,372 INFO [DispatcherServlet] No ViewResolvers found in servlet
'caucho': using default
```

```
23:33:45,372 INFO [DispatcherServlet] Framework servlet 'caucho' init
completed in 60 ms
23:33:45,372 INFO [DispatcherServlet] Servlet 'caucho' configured
successfully
```

通过 IE 能够浏览到 contacts.war 的主页，见图 17-4（通过 index.jsp 重定向到 hello.jsp 而成，详情见 contacts 项目 JSP 代码）。

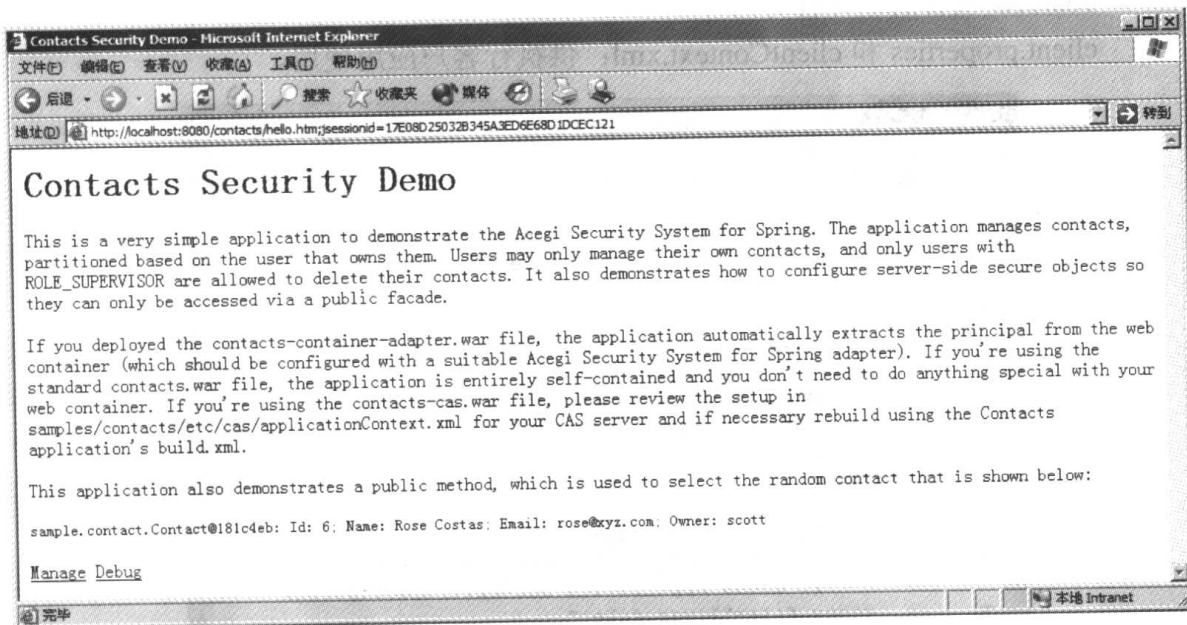


图 17-4 contacts Web 应用主页（未登录）

如果开发者初次登录，则无论点击 Manage 链接，还是 Debug 链接，contacts 都会给出登录页面，见图 17-5 所示。其中，acegilogin.jsp 扮演了基于 Form 认证中的登录和失败页面的角色。

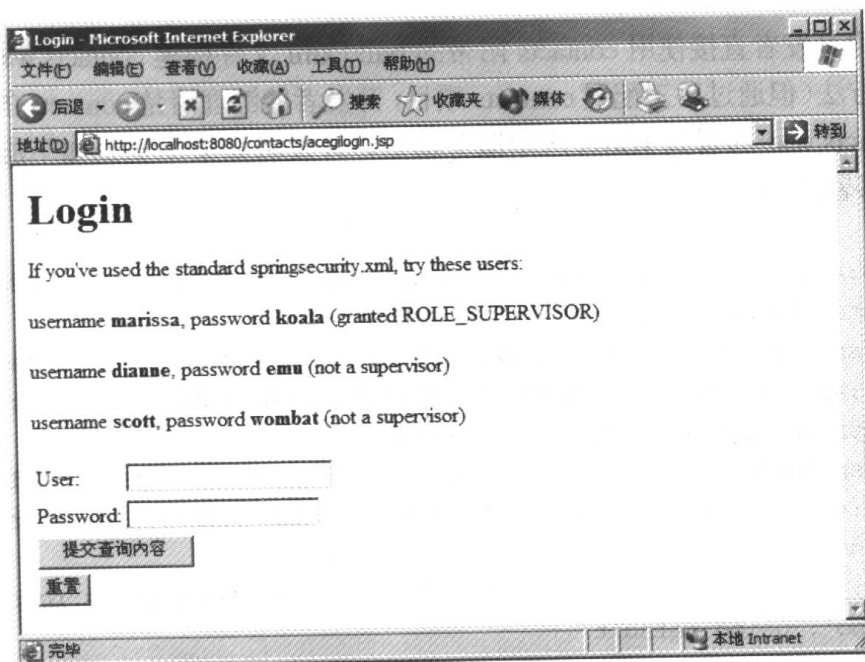


图 17-5 contacts Web 应用登录页面

如果用户不能够通过系统认证, 则将获得如图 17-6 所示的失败结果。

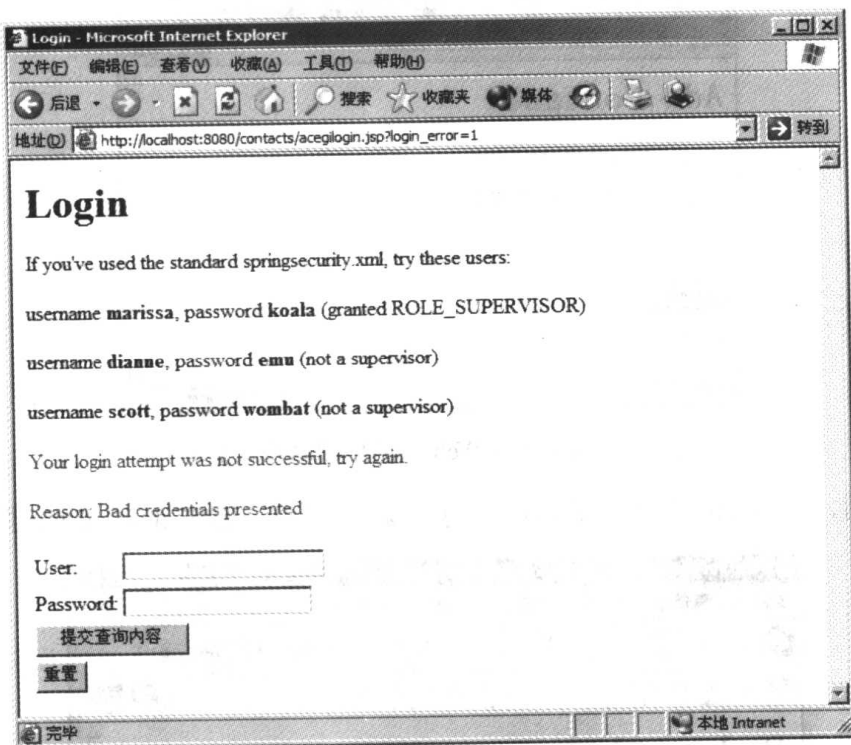


图 17-6 contacts Web 应用登录失败页面

一旦用户通过认证, 而且点击了 Manage 链接, 并且用户 (即 marissa) 是 ROLE_SUPERVISOR 角色, 则将出现如图 17-7 所示的界面 (通过 jsp 目录下的 index.jsp 页面展示)。

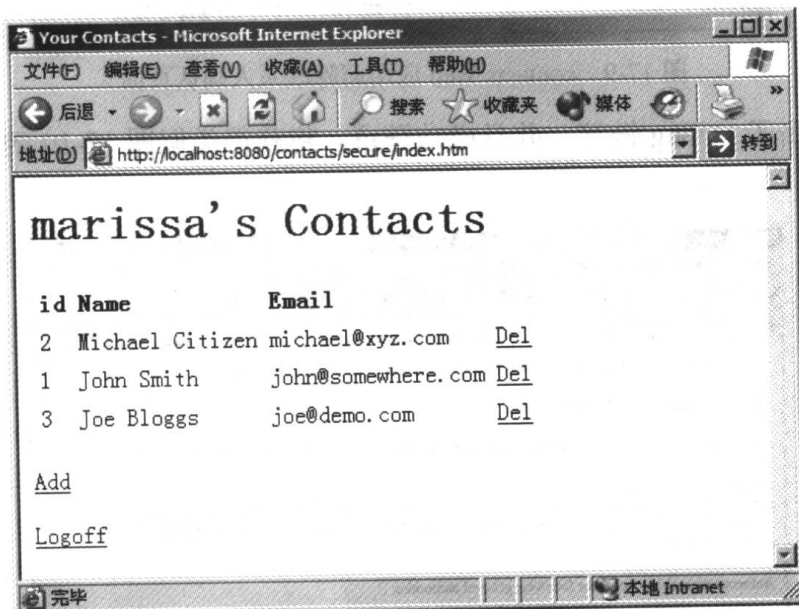


图 17-7 contacts Web 应用登录后的首页面

请注意, 超级用户 (比如, 具有 ROLE_SUPERVISOR 角色的用户) 才有新增联系人 (见图 17-8, 通过 add.jsp 页面展示) 的资格。

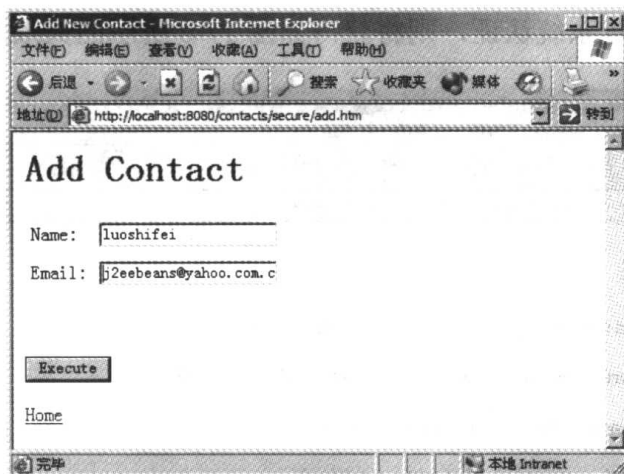


图 17-8 contacts Web 应用新增联系人页面

删除用户后, contacts 将会给出确认信息 (见图 17-9, 通过 deleted.jsp 页面展示)。

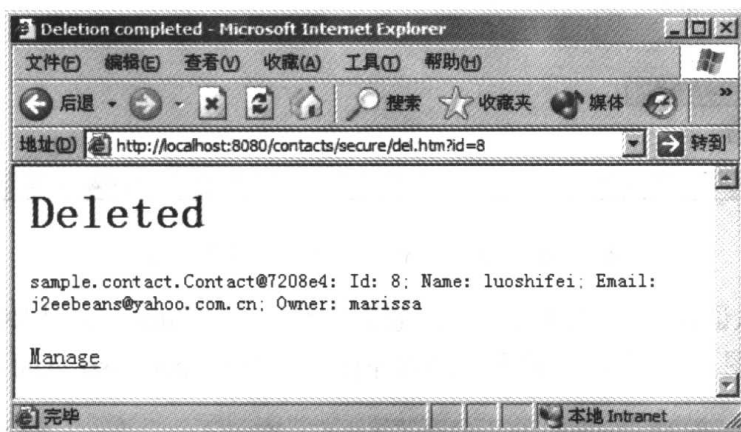


图 17-9 contacts Web 应用删除联系人确认页面

如果用户点击了 Debug 链接, 并且成功登录, 将获得类似图 17-10 的展示结果 (通过 debug.jsp 展示)。

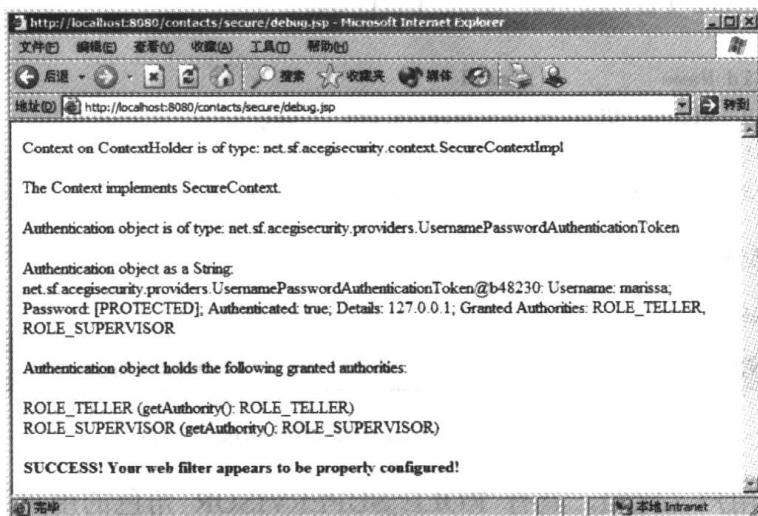


图 17-10 contacts Web 应用中 Debug 页面

其中, 位于 jsp 目录中的 include.jsp 页面用于导入标签库, 比如 Spring、JSTL、Acegi。相应的内容如下。

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<%@ taglib prefix="authz" uri="http://acegisecurity.sf.net/authz" %>
```

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

另外, 借助于 Spring 远程服务 (即 Hessian 和 Burlap, 具体远程服务的导出文件为 caucho-servlet.xml), 开发者还能够通过命令行的方式访问到 contacts 中的业务组件。注意, 这主要是演示如何将 Acegi 安全框架应用到 Spring 远程服务中。客户应用的具体执行方法有两种。其一, 如果直接使用 contacts 提供的 build.xml, 则可以去 client 目录执行 client.bat。比如:

```
D:\acegi-security-0.6.1\samples\contacts\client>clientmarissamarissakoala
```

```
D:\acegi-security-0.6.1\samples\contacts\client>D:\jdk1.5.0_01/bin/java
-cp ../../../../lib/aopalliance/aopalliance-1.0.jar;../../../../lib/caucho/hess
ian.jar;../../../../lib/caucho/burlap.jar;../../../../lib/jakarta-commons/commo
ns-collections.jar;../../../../lib/jakarta-commons/commons-logging.jar;../../
../../lib/spring/spring.jar;contacts.jar sample.contact.ClientApplication
marissa marissa koala
2005-1-9 0:27:43 org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
信息: Loading XML bean definitions from file
[D:\acegi-security-0.6.1\samples\contacts\client\clientContext.xml]
2005-1-9 0:27:44 org.springframework.context.support.
AbstractXmlApplicationContext refreshBeanFactory
信息: Bean factory for application context
[org.springframework.context.support.FileSystemXmlApplicationContext;hash
Code=1280484]: org.springframework.beans.factory.
support.DefaultListableBeanFactory defining beans [propertyConfigurer,hessi
anProxy,burlapProxy]; Root of BeanFactory hierarchy
2005-1-9 0:27:44
org.springframework.context.support.AbstractApplicationContextrefresh
信息: 3 beans defined in ApplicationContext [org.springframework.context.
support.FileSystemXmlApplicationContext;hashCode=1280484]
2005-1-9 0:27:44 org.springframework.beans.factory.
support.AbstractBeanFactory getBean
信息: Creating shared instance of singleton bean 'propertyConfigurer'
2005-1-9 0:27:44 org.springframework.beans.factory.config.
PropertyResourceConfigurer postProcessBeanFactory
信息: Loading properties from file
[D:\acegi-security-0.6.1\samples\contacts\client\client.properties]
2005-1-9 0:27:44 org.springframework.context.support.
AbstractApplicationContextinitMessageSource
信息: No MessageSource found for context
```

```
[org.springframework.context.support.FileSystemXmlApplicationContext;hash
Code=1280484]: using empty default
2005-1-9 0:27:44 org.springframework.context.support.
AbstractApplicationContextinitApplicationEventMulticaster
信息: No ApplicationEventMulticaster found for context [org.springframework.
context.support.FileSystemXmlApplicationContext;hashCode=1280484]: using
default
2005-1-9 0:27:44 org.springframework.context.support.
AbstractApplicationContextrefreshListeners
信息: Refreshing listeners
2005-1-9 0:27:44 org.springframework.beans.factory.
support.DefaultListableBeanFactory preInstantiateSingletons
信息: Pre-instantiating singletons in factory [org.springframework.
beans.factory.support.DefaultListableBeanFactory defining beans
[propertyConfigurer,hessianProxy,burlapProxy]; Root of BeanFactory
hierarchy]
2005-1-9 0:27:44 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
信息: Creating shared instance of singleton bean 'hessianProxy'
2005-1-9 0:27:44 org.springframework.beans.factory.support.
AbstractBeanFactory getBean
信息: Creating shared instance of singleton bean 'burlapProxy'Calling
ContactManager 'hessianProxy' for owner marissa
Trying to find setUsername(String) method
Found; Trying to setUsername(String) to marissa
Trying to find setPassword(String) method
Found; Trying to setPassword(String) to koala
Contact 0: sample.contact.Contact@1e9cb75: Id: 2; Name: Michael Citizen;
Email:michael@xyz.com; Owner: marissa
Contact 1: sample.contact.Contact@2c84d9: Id: 1; Name: John Smith; Email:
john@somewhere.com; Owner: marissa
Contact 2: sample.contact.Contact@c5c3ac: Id: 3; Name: Joe Bloggs; Email:
joe@demo.com; Owner: marissa

Calling ContactManager 'burlapProxy' for owner marissa
Trying to find setUsername(String) method
Found; Trying to setUsername(String) to marissa
Trying to find setPassword(String) method
Found; Trying to setPassword(String) to koala
Contact 0: sample.contact.Contact@b5f53a: Id: 2; Name: Michael Citizen; Email:
michael@xyz.com; Owner: marissa
Contact 1: sample.contact.Contact@1f6f0bf: Id: 1; Name: John Smith; Email:
john@somewhere.com; Owner: marissa
Contact 2: sample.contact.Contact@137c60d: Id: 3; Name: Joe Bloggs; Email:
joe@demo.com; Owner: marissa
```

```
StopWatch '1 ContactManager call(s)': running time (seconds) = 0.621
```

```
-----  
ms      %      Task name  
-----
```

```
00481 077% hessianProxy
```

```
00140 023% burlapProxy
```

```
D:\acegi-security-0.6.1\samples\contacts\client>
```

请注意, **contacts.jar** 的内容如下。

```
D:\acegi-security-0.6.1\samples\contacts\client>jar tf contacts.jar
```

```
META-INF/
```

```
META-INF/MANIFEST.MF
```

```
sample/
```

```
sample/contact/
```

```
sample/contact/ClientApplication.class
```

```
sample/contact/Contact.class
```

```
sample/contact/ContactManager.class
```

其二, 如果借助于 Eclipse, 即本节介绍的方法, 则需要配置应用参数, 见图 17-11。

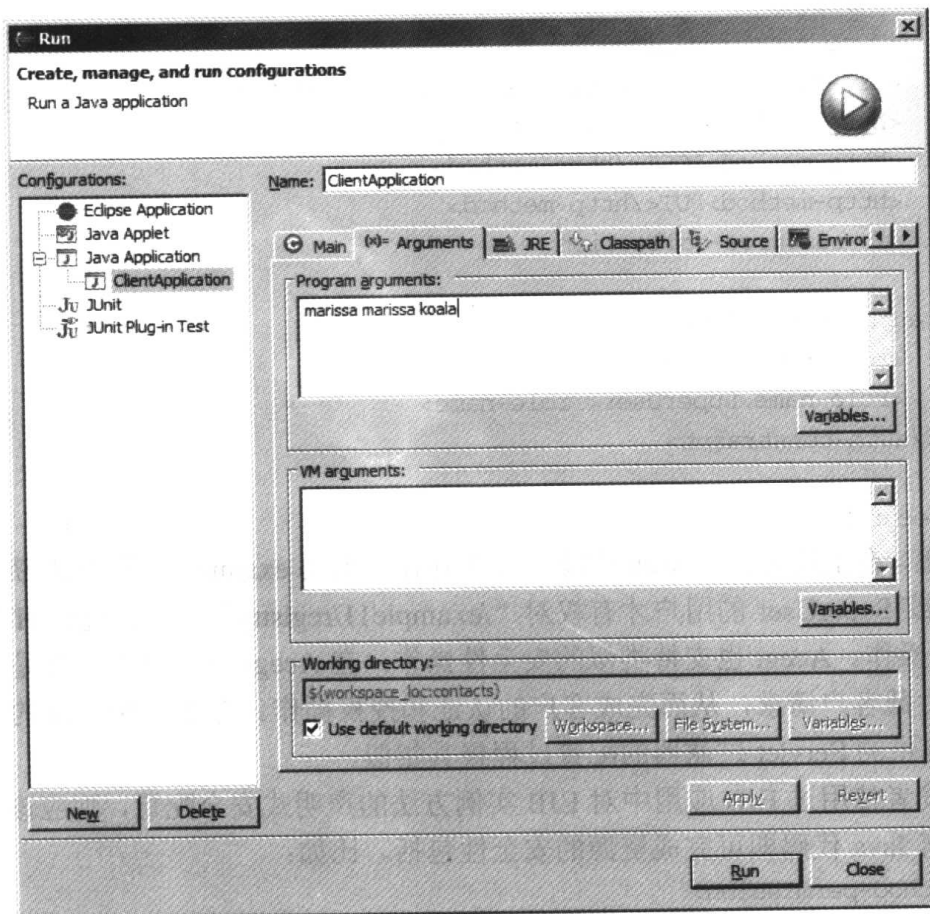


图 17-11 执行 contacts Web 应用中的 ClientApplication

整个 contacts Web 应用的功能就这些, 但其包含的内容基本上涉及到 Acegi 的各个基础方面, 因此结合 contacts Web 应用研究 Acegi 很有意义。至于上述过程中, 出现的日志、

演示的功能，本章随后的内容将一一讲述（如果开发者暂时不理解，则通过后续内容将能够认识清楚）。

17.2.2 Acegi 架构综述

本节内容将对 Acegi 的架构作概要性的介绍。

借助于 Acegi 安全框架，开发者能够在 Spring 使能应用中使用声明式安全，即应用的所有安全性需求都能够通过配置完成并实现。当然，Acegi 安全框架需要在 Spring BeanFactory (ApplicationContext) 中配置若干 JavaBean。其中，在实现声明式安全过程中，需要借助于 Spring AOP 模块，这同声明式事务类似。

如果开发者使用过 J2EE 中对 Web 资源的声明式安全配置，则应该很清楚：不需要开发任何 Java 代码即可完成资源的安全性保护。比如：

```
<security-constraint>

    <web-resource-collection>
        <web-resource-name>
            example11
        </web-resource-name>
        <url-pattern>/example11/registry</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
        <http-method>PUT</http-method>
    </web-resource-collection>

    <auth-constraint>
        <role-name>Admin</role-name>
        <role-name>SuperUser</role-name>
    </auth-constraint>

</security-constraint>
```

可以看出，在上述定义的 Web 资源集合中存在一名为 example11 的 Web 资源，只有角色为 Admin 或 SuperUser 的用户才有权对 “/example11/registry” URL 进行 HTTP GET、POST、PUT 操作。Acegi 也支持类似的安全性操作。在 Acegi 中，借助于其提供的 Servlet 过滤器能够拦截客户请求，从而完成客户的认证和授权操作。考虑到 Spring 的设计哲学，这些 Acegi 提供的 Servlet 过滤器的配置过程极其相似。

如果开发者使用过 EJB 应用中对 EJB 实例方法的声明式安全配置，则应该很清楚：不需要开发任何 Java 代码即可完成资源的安全性包括。比如：

```
<method-permission>

    <role-name>Admin</role-name>
    <method>
        <ejb-name>RegistrySessionBean</ejb-name>
        <method-name>registry</method-name>
    </method>
</method-permission>
```



```
</method>

</method-permission>

<method-permission>

    <role-name>SuperUser</role-name>
    <method>
        <ejb-name>LoginSessionBean</ejb-name>
        <method-name>*</method-name>
    </method>

</method-permission>
```

可以看出,上述定义了两套 EJB 方法允许。其一,如果用户需要访问 `RegistrySessionBean` 的 `registry` 方法,则其角色必须是 `Admin`。其二,如果用户需要访问 `LoginSessionBean` (的所有方法),则其角色必须是 `SuperUser`。这是一种方法级的声明式安全性。`Acegi` 也支持这种安全性配置。注意,在 J2EE 平台中,方法级的声明式安全只有 EJB 组件才能够享受到,而 `Acegi` 提供的声明式安全能够应用于任何 POJO。`Acegi` 将代理目标对象(类似于上述 EJB 中的受保护方法),并借助于 Spring AOP 来保证访问目标对象的用户具有合适的角色,否则用户无权访问目标对象及其方法。

为实现对 Web 资源和 POJO 对象中方法的保护, `Acegi` 提供了 4 项重要组件。它们分别是安全性拦截器 (`AbstractSecurityInterceptor`)、认证管理器 (`AuthenticationManager`)、访问决定管理器 (`AccessDecisionManager`)、Run-As 管理器 (`RunAsManager`)。它们的作用分别如下。

- 借助于安全性拦截器能够实现对客户请求的拦截,进而对用户进行认证 (`AuthenticationManager`)、授权 (`AccessDecisionManager`),或者给它赋予不同的角色 (`RunAsManager`)。
- 借助于认证管理器,能够实现对用户身份的识别。通常,它需要借助于用户名 (`Principal`) 和密码 (`Credential`, 凭证) 进行。
- 借助于访问决定管理器,能够实现对资源的授权。如果用户对应的角色不满足目标资源的安全性要求,则将不能够访问到它;否则,用户将能够享受到目标资源所带来的体验。
- 借助于 Run-As 管理器,能够更换用户对应的角色。在标准的 J2EE Web 应用和 EJB 应用中,也存在类似的行为。请记住,在客户访问 Servlet、EJB、POJO 资源 (服务提供者) 的同时,Servlet、EJB、POJO 资源 (此时,它们是客户) 也可能是其他资源的客户。因此,需要对用户对应的角色进行调整,这就是 Run-As 管理器所扮演的角色。

接下来,本章将分别论述 Web 资源的认证 (第 17.2.3 节)、Web 资源的授权 (第 17.2.4 节)、方法级的认证和授权 (第 17.2.6 节)。其中,第 17.2.5 节将研究 `Acegi` 中 Servlet 过滤器的配置。

17.2.3 Web 资源的认证

在访问受保护的 Web 应用中，需要对用户的身份进行认证。针对 J2EE 认证的基本原则，Acegi 为 J2EE 认证服务提供了如下两个接口：Authentication 和 AuthenticationManager。

其中，Authentication 接口持有 3 个重要对象，见图 17-12。其一，用于标识调用者（用户）的 Principal，即通过 `getPrincipal()` 方法获得它。在 Web 应用中，这一般都是用户的登录名。其二，用于证明调用者身份的凭证，通过 `getCredentials()` 方法获得它。在 Web 应用中，这一般都是用户的登录密码。其三，用户角色信息，即授权给 Principal 的权限列表，通过 `getAuthorities()` 方法获得它。在对用户进行认证的过程中，比如在 Web 应用中，通过用户名和密码构建 Authentication 对象。请注意，Acegi 内置了若干个 Authentication 接口实现，比如用于简单地表示用户名和密码的 `UsernamePasswordAuthenticationToken`、用于单元测试目的的 `TestingAuthenticationToken` 等。请注意，此时用户授权信息（即用户角色信息）还未获得。

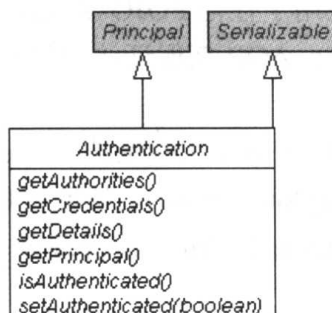


图 17-12 Authentication 接口

在整个用户认证过程中，AuthenticationManager 能够设置授权给 Principal 的权限列表信息。AuthenticationManager 接口仅含有单个的 `authenticate` 方法。

```
public Authentication authenticate(Authentication authentication)
    throws AuthenticationException;
```

从方法签名可以看出，如果成功通过 `authenticate` 方法认证，则传入的 `authentication` 对象会设置授权给 Principal 的权限列表，并返回处理后的 `authentication` 对象。如果认证失败，则抛出 `AuthenticationException` 异常。当然，Acegi 抛出的具体异常信息取决于具体的 `AuthenticationException` 子类。目前，Acegi 提供的 `AuthenticationException` 子类见图 17-13。比如，`UsernameNotFoundException` 表明 `authentication` 对象中持有的用户（Principal）不存在、`LockedException` 表明 Principal 所属账号被锁住、`DisabledException` 表明 Principal 所属账号失效、`BadCredentialsException` 表明提供了错误的 Principal 或凭证等。通常，如果 Principal 所属账号失效或者被锁住，Acegi 安全框架不会再去检查凭证，至于授权操作更不会去理会。

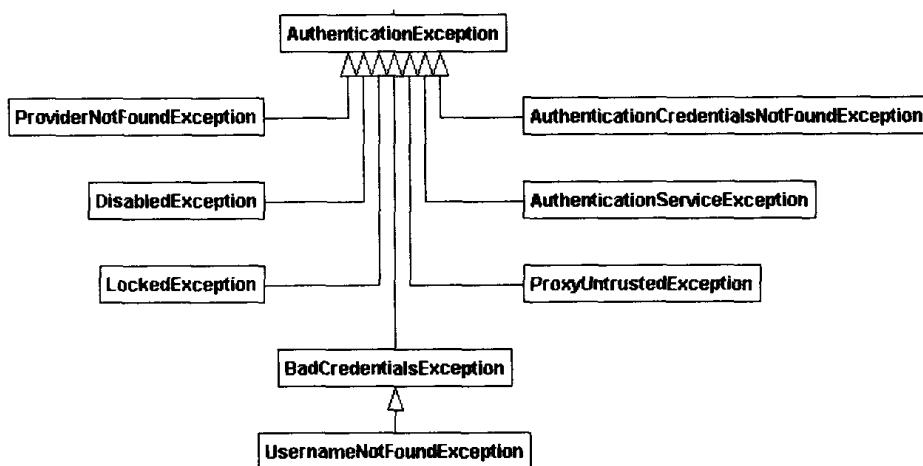


图 17-13 AuthenticationException 接口及其子类

Acegi 为 AuthenticationManager 接口提供了 ProviderManager 实现，见图 17-14。其中，ProviderManager 仅仅是对 AuthenticationProvider 列表的包裹，即通过 providers 属性给出。在认证过程中，将会遍历 providers 包含的所有 AuthenticationProvider，一旦找到合适的，将执行相应的 Authenticate 方法。注意，在使用 ProviderManager 实现过程中，开发者至少需要为它的 providers 属性提供一个 AuthenticationProvider。否则，Spring 使能应用将不能够成功启动。

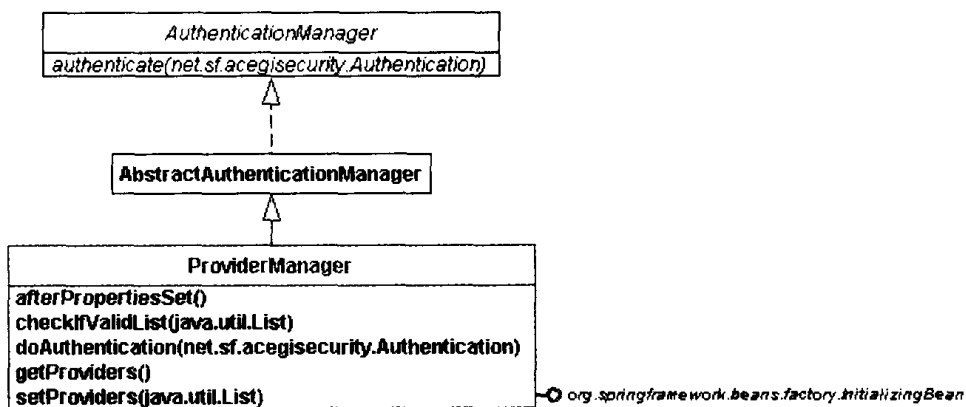


图 17-14 AuthenticationManager 接口及其子类

请注意 AuthenticationProvider 的存在，Acegi 为 AuthenticationProvider 提供了很多子类，见图 17-15 所示。各个 AuthenticationProvider 的具体含义见如下。

- net.sf.acegisecurity.providers.jaas.JaasAuthenticationProvider: 从 JAAS 登录配置中获得用户安全性信息，从而实现对用户的认证。
- net.sf.acegisecurity.providers.dao.PasswordDaoAuthenticationProvider: 借助于底层持久源（比如，AD、LDAP 服务器）完成用户认证。注意，它同 DaoAuthenticationProvider 的区别在于，PasswordDaoAuthenticationProvider 的底层持久源能够完成实际认证过程。在本书写作之际，最新的 Acegi 发布版中并未包括其实现。一般而言，Acegi 应该会推出基于 LDAP 服务器的 PasswordDaoAuthenticationProvider 实现。当然，由于微软活动目录（Active

Directory, AD) 也支持 LDAP V2/3 协议, 因此相应的 LDAP 实现也将支持活动目录。

- `net.sf.acegisecurity.providers.dao.DaoAuthenticationProvider`: 借助于 RDBMS、内存、XML 文件等存储源获得用户名和密码, 从而实现用户的认证。
- `net.sf.acegisecurity.providers.cas.CasAuthenticationProvider`: 借助于 Yale 提供的 CAS 进行认证。
- `net.sf.acegisecurity.adapters.AuthByAdapterProvider`: 使用容器适配器对用户进行认证。比如, 开发者可直接使用 JBoss 提供的 `LoginModule` 实现, 而不需借助于 Acegi 提供的其他 `AuthenticationProvider`。
- `net.sf.acegisecurity.runas.RunAsImplAuthenticationProvider`: 供认证已被 `RunAsManager` 替换用户身份的用户使用。
- `net.sf.acegisecurity.providers.rcp.RemoteAuthenticationProvider`: 用于对 Spring 远程服务的客户进行认证操作, 比如对 Hessian 和 Burlap 客户进行认证。
- `net.sf.acegisecurity.providers.TestingAuthenticationProvider`: 仅仅供单元测试使用。建议开发者不要用于实际产品部署场合。

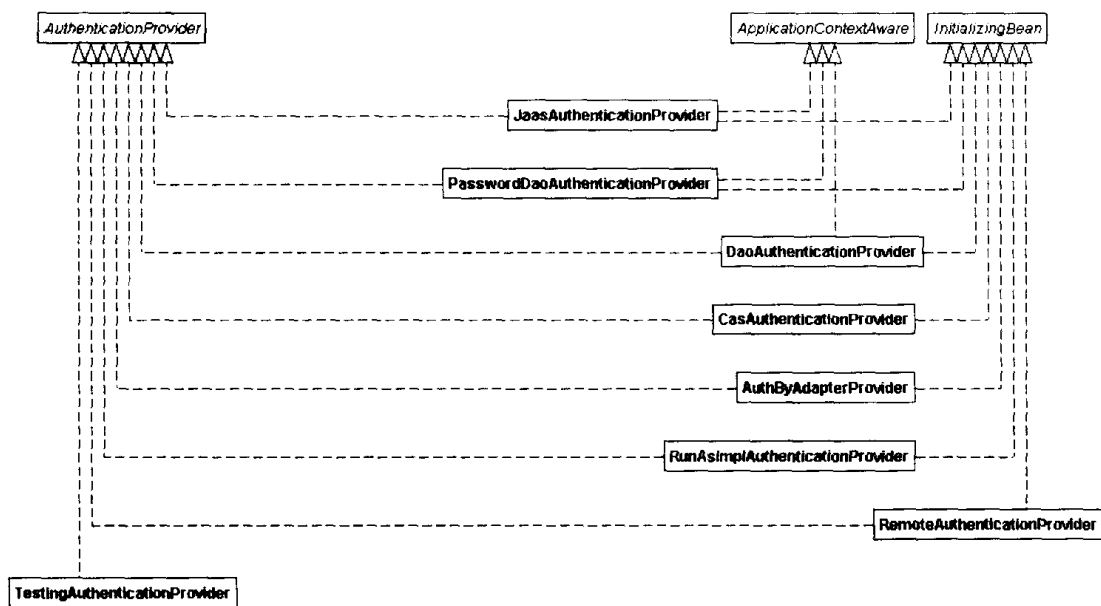


图 17-15 AuthenticationProvider 接口及其子类

开发者是否注意到, `contacts` Web 应用使用到 `AuthenticationProvider`: `RunAsImplAuthenticationProvider` 和 `DaoAuthenticationProvider`。

为实现认证过程中所要求的用户信息, 比如用户名、密码, 开发者可以借助于 `AuthenticationDAO` 接口及其实现。Acegi 为 `AuthenticationDAO` 接口提供了图 17-16 所示的实现。注意, 右下角的 `JdbcDaoImpl` 是用于用户授权的 `BasicAclDao`。在第 17.2.4 节内容将讨论到它。

其中, `InMemoryDaoImpl` 和 `JdbcDaoImpl` 的含义如下。

- `net.sf.acegisecurity.providers.dao.memory.InMemoryDaoImpl`: Acegi 借助于 Spring `ApplicationContext` 维护的内存列表来获得用户、密码、角色等信息。

- `net.sf.acegisecurity.providers.dao.jdbc.JdbcDaoImpl`: 借助于 RDBMS 获得用户认证信息。

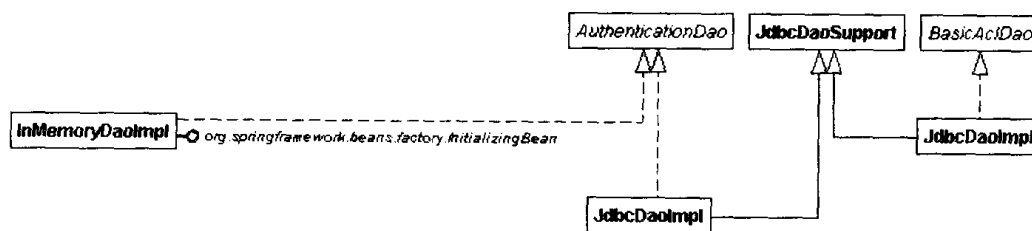


图 17-16 AuthenticationDao 接口及其子类

来看看 `contacts.war` 是如何实现 Web 认证的吧，以 `AuthenticationDao` 为切入点开始。开发者打开 `applicationContext.xml` 后，能够浏览到 `InMemoryDaoImpl` JavaBean 定义。依此指定用户名（Principal）、密码（凭证）及角色（权限列表）。可以看出，`userMap` 属性定义了用户安全性信息。其中，`userMap` 属性取值中各行内容定义了值对。等号左边是用户名；等号右边是使用逗号隔开的密码、用户账号使用状态、角色集合。比如，`scott` 是用户名、`2b58af6dddbd072ed27ffc86725d7d3a` 是加密后的密码、`ROLE_TELLER` 是 `scott` 对应的角色名。借助于 `disabled` 能够表明，当前账号未启用。

```

<bean id="inMemoryDaoImpl"
      class="net.sf.acegisecurity.providers.dao.memory.InMemoryDaoImpl">
  <property name="userMap">
    <value>
      marissa=a564de63c2d0da68cf47586ee05984d7,
      ROLE_TELLER,ROLE_SUPERVISOR
      dianne=65d15fe9156f9c4bbffd98085992a44e,ROLE_TELLER
      scott=2b58af6dddbd072ed27ffc86725d7d3a,ROLE_TELLER
      peter=22b5c9accc6e1ba628cedc63a72d57f8,disabled,ROLE_TELLER
    </value>
  </property>
</bean>

```

在实际产品部署场合，不应该使用 `InMemoryDaoImpl`。每次修改用户安全性信息后，必须重新部署应用，而且借助于 Spring 配置文件维护大量的用户信息太不方便了。因此，本书建议使用 `JdbcDaoImpl` 实现。

`JdbcDaoImpl`，继承于 Spring 提供的 `JdbcDaoSupport`。因此，如果需要使用它，则开发者必须提供数据源。通过该数据源，开发者能够获得用户安全性信息。进而，再也不用通过 Spring 配置文件获得用户安全性信息了。

其中，用户密码通过 MD5 加密算法加密（第 17.3.1 节内容将讨论它的具体使用）。

```

<bean id="passwordEncoder"
      class="net.sf.acegisecurity.providers.encoding.Md5PasswordEncoder"/>

```

然后需要定义出 `DaoAuthenticationProvider` 实例。请注意，其中使用了 `userCache` 实例。这主要是用于配置 User 对象实例的缓存（第 17.3.2 节内容将讨论它的具体使用）。

```

<bean id="daoAuthenticationProvider"
      class="net.sf.acegisecurity.providers.dao.DaoAuthenticationProvider">

```

```
<property name="authenticationDao">
    <ref local="inMemoryDaoImpl"/>
</property>
<property name="userCache">
    <ref local="userCache"/>
</property>
<property name="passwordEncoder">
    <ref local="passwordEncoder"/>
</property>
</bean>
```

userCache 定义如下（缓存有效时间为 5 分钟）。

```
<bean id="userCache"
    class="net.sf.acegisecurity.providers.dao.cache.EhCacheBasedUserCache">
    <property name="minutesToIdle">
        <value>5</value>
    </property>
</bean>
```

当然，开发者还需要定义 **ProviderManager** 实例，以配置使用 **daoAuthenticationProvider**（注意，**runAsAuthenticationProvider** 的具体使用同 **daoAuthenticationProvider** 类似，留给开发者自行研究）。

```
<bean id="authenticationManager"
    class="net.sf.acegisecurity.providers.ProviderManager">
    <property name="providers">
        <list>
            <ref local="runAsAuthenticationProvider"/>
            <ref local="daoAuthenticationProvider"/>
        </list>
    </property>
</bean>
```

至此，认证管理器（**authenticationManager**）已经完全配置成功，图 17-17 给出了 **authenticationManager** 的图形式表示。但是在使用它之前，还需要将它绑定到 Web 应用中。本章第 17.2.5 节内容将讨论这方面的内容。

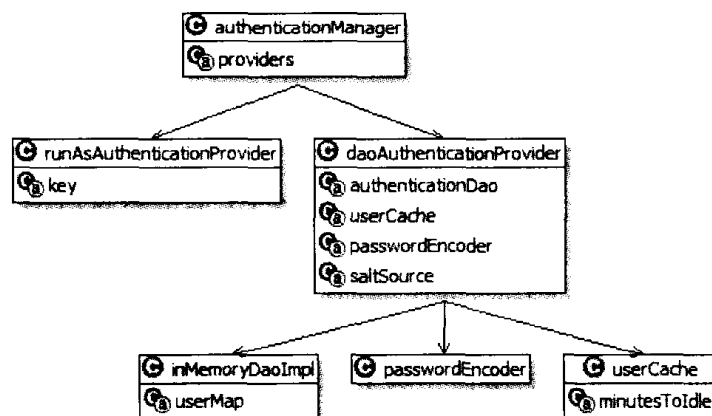


图 17-17 “authenticationManager”的图形化表示

17.2.4 Web 资源的授权

为保护 Web 应用资源，Acegi 需要对已认证用户进行授权。针对用户所属的角色，并根据待访问资源所要求的角色，Acegi 来决定用户是否有资格操作它。当然，Acegi 借助于安全性拦截器实现了对受保护资源的受限访问。

其中，`net.sf.acegisecurity.AccessDecisionManager` 在授权过程中起到了很关键的作用。Acegi 内置了它的若干实现，见图 17-18。尽管开发者能够开发自定义的 `AccessDecisionManager` 实现，但 Acegi 内置的实现基本能够满足各种业务需求。

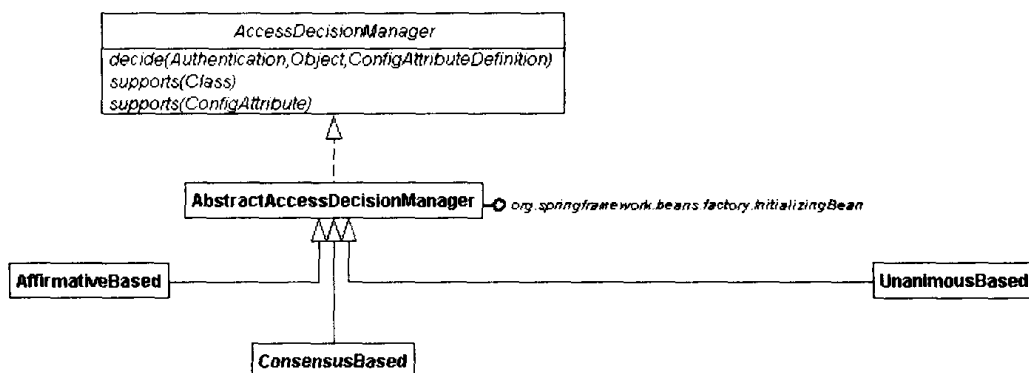


图 17-18 AccessDecisionManager 接口及其子类

具体解析如下：

- `net.sf.acegisecurity.vote.ConsensusBased`：在考虑同意票和反对票（不考虑弃权票）的前提下，对受保护资源的授权（要求同意票大于反对票）。
- `net.sf.acegisecurity.vote.UnanimousBased`：在忽略弃权票的前提下，要求其他票都是同意票，从而实现对受保护资源的授权。
- `net.sf.acegisecurity.vote.AffirmativeBased`：在忽略反对票的前提下，只要存在至少一个同意票，即可对受保护资源的访问。

具体的投票过程由 `net.sf.acegisecurity.vote.AccessDecisionVoter` 决定。基于角色授权是常见的 Web 策略，也是 J2EE 的安全性推荐策略，因此 Acegi 提供了 `RoleVoter` 投票实现，见图 17-19。注意，基于角色授权是大部分企业的安全策略。对于现有的企业组织划分，使用角色来表示授权信息尤为合适。在工作流、BPEL 服务器等场合，使用角色表示授权信息也是很常见的做法。

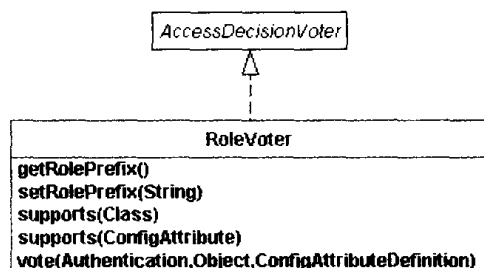


图 17-19 AccessDecisionVoter 接口及其子类

因此，基于 Web 的授权机理很简单，下面来看看 contacts.war 是如何实现授权配置的。首先，开发者需要在 applicationContext.xml 中定义出 roleVoter。Acegi 为 AccessDecisionVoter 提供了 net.sf.acegisecurity.vote.RoleVoter 实现。

```
<!-- An access decision voter that reads ROLE_* configuration settings -->
<bean id="roleVoter"
      class="net.sf.acegisecurity.vote.RoleVoter"/>
```

默认时，RoleVoter 仅仅对前缀为“ROLE_”的角色名进行投票，否则将投弃权票。如果需要改变角色名，则通过 RoleVoter 的 rolePrefix 属性能够修改 RoleVoter 的默认行为。其次，配置 httpRequestAccessDecisionManager 对象。其中，不允许所有的参与者都投弃权票。

```
<bean id="httpRequestAccessDecisionManager"
      class="net.sf.acegisecurity.vote.AffirmativeBased">
  <property name="allowIfAllAbstainDecisions">
    <value>>false</value>
  </property>
  <property name="decisionVoters">
    <list>
      <ref local="roleVoter"/>
    </list>
  </property>
</bean>
```

可以看出，decisionVoters 属性引用了 roleVoter。而且，借助于 allowIfAllAbstainDecisions 属性能够约定参与投票者的行为，即不允许所有的参与者都投弃权票。

至此，AccessDecisionManager 已经配置成功。图 17-20 给出了 contacts.war Web 应用中已配置的 AccessDecisionManager 集合。注意，businessAccessDecisionManager 和 contactSecurityVoter 是 contacts 应用对 Acegi 的扩展。

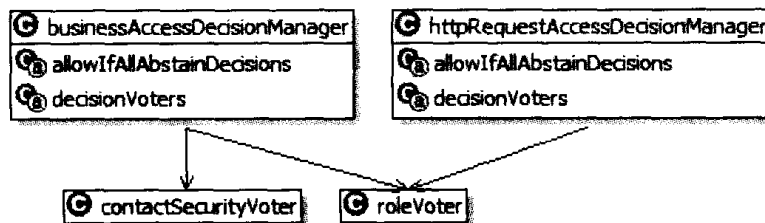


图 17-20 AccessDecisionManager 的图形化表示

17.2.5 配置 Acegi Servlet 过滤器

通过前两节内容，我们已经将 Web 资源认证和授权过程中的 AuthenticationManager 和 AccessDecisionManager 成功配置。接下来，开发者可能会问，如何将它们集成在一起，并供企业应用使用呢？本节内容将回答上述问题。其中，本节还将结合 contacts.war Web 应用中的相关内容阐述。

为实现 Acegi 使能 Web 应用，开发者需要配置其提供的 Servlet 过滤器。Servlet 过滤器，类似于 AOP Around 装备，在方法或者 Web 资源调用前后进行一些其他操作。对于 Acegi

而言, Servlet 过滤器能够在应用处理客户请求前能够完成客户的安全性处理操作。总的而言, Acegi 提供了如下 6 种 Servlet 过滤器。注意, 它们依次构成了 Servlet 过滤器链, 即这些过滤器会依次处理客户请求。

- `net.sf.acegisecurity.securechannel.ChannelProcessingFilter` 过滤器: 用于检查客户请求的分发 Channel (delivery channel) 是否满足受保护 Web 资源的安全性需求。通常, Web 资源可以通过 HTTP 或者 HTTPS 访问。如果目标资源需要 HTTPS 访问, 则客户是通过 HTTP 访问它, 则 `ChannelProcessingFilter` 会重定向 (redirect) 到 HTTPS。反之, 如果目标资源只是需要 HTTP 访问, 而客户是通过 HTTPS 访问它, 则它会重定向到 HTTP。它需要配置在其他 Servlet 过滤器前, 否则 Spring 使能应用不能够正常启动。
- `net.sf.acegisecurity.ui.webapp.AuthenticationProcessingFilter` 过滤器: 用于判断客户请求是否是认证请求。如果是则借助于认证管理器完成客户的实际认证过程。如果不是则后续的 Servlet 过滤器继续工作。`AuthenticationProcessingFilter` 用于处理基于表单的认证 (Form)。
- `net.sf.acegisecurity.ui.cas.CasProcessingFilter` 过滤器: 用于判断客户请求是否是认证请求。如果是则借助于认证管理器完成客户的实际认证过程。如果不是则后续的 Servlet 过滤器继续工作。`CasProcessingFilter` 用于处理基于 CAS 的认证。
- `net.sf.acegisecurity.ui.basicauth.BasicProcessingFilter` 过滤器: 用于判断客户请求是否是认证请求。如果是则借助于认证管理器完成客户的实际认证过程。如果不是则后续的 Servlet 过滤器继续工作。`BasicProcessingFilter` 用于处理基于 HTTP BASIC 的认证。以上 3 者 (包括 `BasicProcessingFilter`) 都是供认证客户使用的, 只不过客户使用的认证机制不同而已。
- `net.sf.acegisecurity.ui.AutoIntegrationFilter` 过滤器: 由于 HTTP 本身是无状态的, 因此 Acegi 需要提供某种机制, 实现客户请求间认证信息的存储。通常, 这种信息都是借助于 HTTP Session 完成的。`AutoIntegrationFilter` 能够获取用户认证信息。
- `net.sf.acegisecurity.intercept.web.SecurityEnforcementFilter` 过滤器: 对用户是否有权访问目标 Web 资源作出最后的决定。借助于 `SecurityEnforcementFilter`, 使得客户请求能够同 `FilterSecurityInterceptor` 实例进行交互。

接下来一一阐述上述过滤器及其相关内容。

1. Acegi Servlet 过滤器的设计

在 J2EE Web 应用中, Servlet 过滤器为开发者实现 Web 应用的国际化 (I18N)、安全性提供了很好的机制。同时, 在 Spring 使能应用中 (包括使用 Acegi 的场合), 开发者可以利用 IoC 注入各 POJO 服务。Servlet 过滤器本身并不支持 IoC。因此, Acegi 引入了一种机制, 使得它支持 IoC, 从而能够使用到 Spring IoC 容器。当然, 开发者可以直接在 Servlet 过滤器中使用 Spring 提供的 `ApplicationContext` 实用类, 但这使得 Servlet 过滤器的便携性差。

注意, `net.sf.acegisecurity.util.FilterToBeanProxy` 就是 Acegi Servlet 过滤器中的主要设计对象。借助于它, 使得 Servlet 过滤器代码不用同 Spring 耦合在一起。图 17-21 展示了 `FilterToBeanProxy` 过滤器。

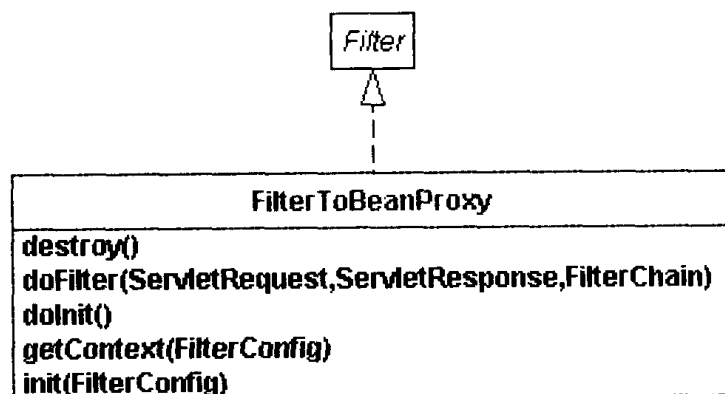


图 17-21 FilterToBeanProxy 过滤器

其中，`getContext()`方法的内容如下。

```
protected ApplicationContext getContext(FilterConfig filterConfig) {
    return WebApplicationContextUtils.
        getRequiredWebApplicationContext(filterConfig
            .getServletContext());
}
```

借助于 `Spring WebApplicationContextUtils` 类使得 `FilterToBeanProxy` 能够操作到 IoC 容器。为配置使用 `FilterToBeanProxy`，开发者需要遵循如下步骤进行。

其一，在 `web.xml` 中配置 `FilterToBeanProxy`。下面给出了 `contacts.war` 中的配置片断。可以看到，`targetClass` 指向了 `AuthnticationProcessingFilter`。随后，`FilterToBeanProxy` 将会在 `Spring` 配置文件（`applicationContext.xml`）中查找 `AuthnticationProcessingFilter` 类型的 POJO。注意，`FilterToBeanProxy` 还为 `targetClass` 属性提供了替代参数，即 `targetBean` 初始参数。`FilterToBeanProxy` 借助于 `targetBean`，将直接查找 `targetBean` 指定的 `Spring` POJO 实例，而不是 POJO 类型。

```
<filter>
    <filter-name>Acegi Authentication Processing Filter</filter-name>
    <filter-class>
        net.sf.acegisecurity.util.FilterToBeanProxy
    </filter-class>
    <init-param>
        <param-name>targetClass</param-name>
        <param-value>
            net.sf.acegisecurity.ui.webapp.AuthenticationProcessingFilter
        </param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>Acegi Authentication Processing Filter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

其二，在 `applicationContext.xml` 中配置 `AuthnticationProcessingFilter` 实例定义。如下给

出配置片断。

```
<bean id="authenticationProcessingFilter"
      class="net.sf.acegisecurity.ui.webapp.
          AuthenticationProcessingFilter">
  <property name="authenticationManager">
    <ref local="authenticationManager"/>
  </property>
  <property name="authenticationFailureUrl">
    <value>/acegilogin.jsp?login_error=1</value>
  </property>
  <property name="defaultTargetUrl">
    <value>/</value>
  </property>
  <property name="filterProcessesUrl">
    <value>/j_acegi_security_check</value>
  </property>
</bean>
```

在上述 FilterToBeanProxy 代理初始化过程中，将会查找 Spring applicationContext.xml 配置文件，并初始化相应的 Servlet 过滤器。如果未找到相应的 Spring POJO，则 Acegi 将抛出异常。如果找到多个 Spring POJO，则将使用第一个匹配的 POJO。

2. 配置分发 Channel Servlet 过滤器

为实现 HTTPS 传输，开发者需要配置 HTTPS。默认时，contacts.war 并没有启用 HTTPS。配置具体步骤如下。

首先，配置 web.xml 中的 ChannelProcessingFilter。

```
<filter>
  <filter-name>Acegi Channel Processing Filter</filter-name>
  <filter-class>
    net.sf.acegisecurity.util.FilterToBeanProxy
  </filter-class>
  <init-param>
    <param-name>targetClass</param-name>
    <param-value>
      net.sf.acegisecurity.securechannel.ChannelProcessingFilter
    </param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>Acegi Channel Processing Filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

其次，配置 Spring applicationContext.xml。在 Acegi 安全框架中，无论是认证管理器，还是访问决定管理器，整个架构的风格是一致的。因此，开发者理解下面的配置还是比较

轻松。其中，channelDecisionManager 属性指定 ChannelDecisionManager 实现；通过 filterInvocationDefinitionSource 属性指定 Web 资源是否受 HTTPS 保护。类似于 AccessDecisionManager，ChannelDecisionManager 借助于 ChannelProcessor 处理 channel 分发请求。对于 HTTP 和 HTTPS 而言，开发者需要分别使用 SecureChannelProcessor 和 InsecureChannelProcessor 处理器。

```
<bean id="channelProcessingFilter"
    class="net.sf.acegisecurity.securechannel.ChannelProcessingFilter">
    <property name="channelDecisionManager">
        <ref local="channelDecisionManager"/>
    </property>
    <property name="filterInvocationDefinitionSource">
        <value>
            CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
            \A/secure/.*\Z=REQUIRES_SECURE_CHANNEL
            \A/acegilogin.jsp.*\Z=REQUIRES_SECURE_CHANNEL
            \A/j_acegi_security_check.*\Z=REQUIRES_SECURE_CHANNEL
            \A.*\Z=REQUIRES_INSECURE_CHANNEL
        </value>
    </property>
</bean>

<bean id="channelDecisionManager"
    class="net.sf.acegisecurity.securechannel.ChannelDecisionManagerImpl">
    <property name="channelProcessors">
        <list>
            <ref local="secureChannelProcessor"/>
            <ref local="insecureChannelProcessor"/>
        </list>
    </property>
</bean>

<bean id="secureChannelProcessor"
    class="net.sf.acegisecurity.securechannel.SecureChannelProcessor"/>

<bean id="insecureChannelProcessor"
    class="net.sf.acegisecurity.securechannel.InsecureChannelProcessor"/>
```

另外，开发者需要了解 filterInvocationDefinitionSource 属性的内容。在前面讨论 InMemoryDaoImpl 时，开发者曾看过类似的值对。但这比 InMemoryDaoImpl 中定义的复杂些。其中，第一行表明，在使用 ChannelProcessor 处理 URL 之前，需要将所有的 URL 转换为小写格式。后续的各行分别给出了 HTTP 或 HTTPS 要求。这些 URL 是借助于正则表达式给出的。Acegi 还支持另一种 URL 模式，即 Ant 风格。比如，在使用 Ant 风格的 URL 转化上述 URL 后，结果如下。其中，最后一行，即“\A.*\Z”表示任何数量的任意字符(/*)。

```
CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
PATTERN_TYPE_APACHE_ANT
```

```

/secure/**=REQUIRES_SECURE_CHANNEL
/acegilogin.jsp=REQUIRES_SECURE_CHANNEL
/j_acegi_security_check.=REQUIRES_SECURE_CHANNEL
/*=REQUIRES_INSECURE_CHANNEL

```

通过图 17-22 能够浏览到 channelProcessingFilter 过滤器的图形表示。

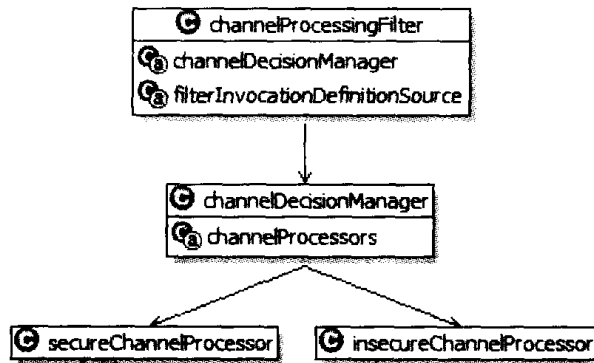


图 17-22 channelProcessingFilter 过滤器

3. 配置客户认证过滤器

目前, 为实现 Web 应用的认证, 存在很多办法。而基于 HTTP Session (表单) 和 HTTP BASIC 认证很常见。contacts 支持上述两种认证方式。当然, CAS 也是经常使用到的认证方式。本书不对 CAS 进行阐述。

Acegi 借助于 Servlet 过滤器实现认证, web.xml 具体配置如下 (摘自 contacts.war)。

```

<filter>
    <filter-name>Acegi Authentication Processing Filter</filter-name>
    <filter-class>
        net.sf.acegisecurity.util.FilterToBeanProxy
    </filter-class>
    <init-param>
        <param-name>targetClass</param-name>
        <param-value>
            net.sf.acegisecurity.ui.webapp.AuthenticationProcessingFilter
        </param-value>
    </init-param>
</filter>

<filter>
    <filter-name>Acegi HTTP BASIC Authorization Filter</filter-name>
    <filter-class>
        net.sf.acegisecurity.util.FilterToBeanProxy
    </filter-class>
    <init-param>
        <param-name>targetClass</param-name>
        <param-value>
            net.sf.acegisecurity.ui.basicauth.BasicProcessingFilter
        </param-value>
    </init-param>
</filter>

```

```

        </init-param>
    </filter>

    <filter-mapping>
        <filter-name>Acegi Authentication Processing Filter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <filter-mapping>
        <filter-name>Acegi HTTP BASIC Authorization Filter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

```

对于基于 HTTP Session 认证而言, `AuthenticationProcessingFilter` 需要从表单 (分别从 `j_username` 和 `j_password` 表单域) 中获得用户名和密码, 这同标准的 HTTP Form 认证相同。在 `contacts.war` 应用中 `acegilogin.jsp` (见如下 `acegilogin.jsp` 代码示例) 提供了该表单。

```

<form action="<c:url value='j_acegi_security_check' />" method="POST">
    <table>
        <tr>
            <td>
                User:</td><td><input type='text' name='j_username'>
            </td>
        </tr>
        <tr>
            <td>
                Password:</td><td><input type='password' name='j_password'>
            </td>
        </tr>
        <tr>
            <td colspan='2'><input name="submit" type="submit"></td>
        </tr>
        <tr>
            <td colspan='2'><input name="reset" type="reset"></td>
        </tr>
    </table>
</form>

```

Acegi 借助于 `FilterToBeanProxy` 将认证工作委派给 Spring `ApplicationContext` 中配置的具体过滤器。详情见 `applicationContext.xml` 中的相应部分。

```

//用于处理 HTTP BASIC
<bean id="basicProcessingFilter"
    class="net.sf.acegisecurity.ui.basicauth.BasicProcessingFilter">
    <property name="authenticationManager">
        <ref local="authenticationManager"/>
    </property>
    <property name="authenticationEntryPoint">
        <ref local="basicProcessingFilterEntryPoint"/>
    </property>

```

```

    </property>
</bean>

<bean id="basicProcessingFilterEntryPoint"
      class="net.sf.acegisecurity.ui.basicauth
        .BasicProcessingFilterEntryPoint">
  <property name="realmName">
    <value>Contacts Realm</value>
  </property>
</bean>

//用于处理 HTTP Session
<bean id="authenticationProcessingFilter"
      class="net.sf.acegisecurity.ui.webapp.AuthenticationProcessingFilter">
  <property name="authenticationManager">
    <ref local="authenticationManager"/>
  </property>
  <property name="authenticationFailureUrl">
    <value>/acegilogin.jsp? login_error=1</value>
  </property>
  <property name="defaultTargetUrl">
    <value>/</value>
  </property>
  <property name="filterProcessesUrl">
    <value>/j_acegi_security_check</value>
  </property>
</bean>

```

对于 HTTP BASIC 认证而言, 开发者需要配置 Realm 名, 即 realmName。对于 HTTP Session 认证而言, 开发者需要提供认证失败的页面。在 contacts.war 中认证失败页面也是 acegilogin.jsp, 相应的代码片断如下。

```

<!-- this form-login-page form is also used as the
      form-error-page to ask for a login again.
-->
<c:if test="${not empty param.login_error}">
  <font color="red">
    Your login attempt was not successful, try again.<BR><BR>
    Reason: <%= ((AuthenticationException) session.
      getAttribute(AbstractProcessingFilter.
        ACEGI_SECURITY_LAST_EXCEPTION_KEY)).getMessage() %>
  </font>
</c:if>

```

图 17-23 给出了 basicProcessingFilter 和 authenticationProcessingFilter 过滤器的图形化表示。其中, 对 authenticationManager 作了简化处理。

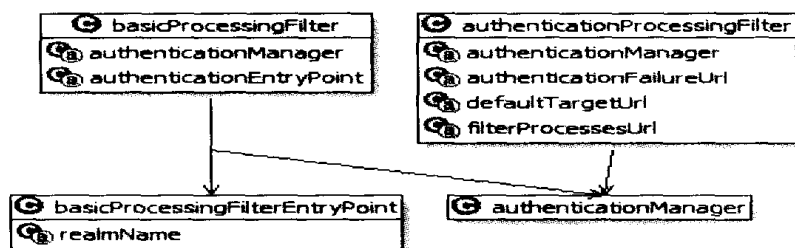


图 17-23 basicProcessingFilter 和 authenticationProcessingFilter 过滤器

4. 配置 AutoIntegrationFilter 过滤器

由于 HTTP 协议的无状态性，使得 Acegi 需要提供存储用户认证信息的机制。因此，这就是需要引入集成 Filter 的原因。通常，集成 Filter 会去 HTTP Session 中查找用户认证信息。由于服务器在处理客户请求过程（比如访问受保护的 Web 资源）中，需要用户认证信息，因此借助于集成 Filter 能够满足此要求。

同其他 Servlet 过滤器一样，首先需要在 web.xml 部署描述符中配置它。

```

<filter>
  <filter-name>
    Acegi Security System for Spring Auto Integration Filter
  </filter-name>
  <filter-class>
    net.sf.acegisecurity.util.FilterToBeanProxy
  </filter-class>
  <init-param>
    <param-name>targetClass</param-name>
    <param-value>
      net.sf.acegisecurity.ui.AutoIntegrationFilter
    </param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>
    Acegi Security System for Spring Auto Integration Filter
  </filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
  
```

其次，需要在 Spring applicationContext.xml 中配置 AutoIntegrationFilter。

```

<bean id="autoIntegrationFilter"
      class="net.sf.acegisecurity.ui.AutoIntegrationFilter"/>
  
```

5. 配置 SecurityEnforcementFilter 过滤器

在客户请求 Web 应用的资源时，需要 SecurityEnforcementFilter 过滤器判断目标资源是否受到保护。如果受到保护，则需要使用 Acegi 提供的认证和访问决定管理器来对用户进行审查。如果审查通过，则用户能够访问到目标资源。

类似于其他 Servlet 过滤器，开发者需要在 web.xml 中配置如下内容（摘自 contacts.war 中的 web.xml）。

```
<filter>
  <filter-name>Acegi HTTP Request Security Filter</filter-name>
  <filter-class>
    net.sf.acegisecurity.util.FilterToBeanProxy
  </filter-class>
  <init-param>
    <param-name>targetClass</param-name>
    <param-value>
      net.sf.acegisecurity.intercept.web.SecurityEnforcementFilter
    </param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>Acegi HTTP Request Security Filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

然后，需要配置 Spring applicationContext.xml 文件中的如下片断。

```
<bean id="securityEnforcementFilter"
  class="net.sf.acegisecurity.intercept.web.SecurityEnforcementFilter">
  <property name="filterSecurityInterceptor">
    <ref local="filterInvocationInterceptor"/>
  </property>
  <property name="authenticationEntryPoint">
    <ref local="authenticationProcessingFilterEntryPoint"/>
  </property>
</bean>
```

在上述片断中，authenticationEntryPoint 属性引用了 authenticationProcessingFilterEntryPoint POJO 服务；而 filterSecurityInterceptor 属性引用了 filterInvocationInterceptor POJO 服务。其中，authenticationProcessingFilterEntryPoint 的定义如下。

```
<bean id="authenticationProcessingFilterEntryPoint"
  class="net.sf.acegisecurity.ui.webapp.
    AuthenticationProcessingFilterEntryPoint">
  <property name="loginFormUrl">
    <value>/acegilogin.jsp</value>
  </property>
  <property name="forceHttps">
    <value>>false</value>
  </property>
</bean>
```

可以看出，authenticationProcessingFilterEntryPoint 中指定 acegilogin.jsp 为登录页面。并且，不使用 HTTPS。另外，filterInvocationInterceptor 的定义如下。

```
<bean id="filterInvocationInterceptor"
```

```

class="net.sf.acegisecurity.intercept.web.FilterSecurityInterceptor">
  <property name="authenticationManager">
    <ref local="authenticationManager"/>
  </property>
  <property name="accessDecisionManager">
    <ref local="httpRequestAccessDecisionManager"/>
  </property>
  <property name="runAsManager">
    <ref local="runAsManager"/>
  </property>
  <property name="objectDefinitionSource">
    <value>
      CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
      \A/secure/super.*\Z=ROLE_WE_DONT_HAVE
      \A/secure/.*\Z=ROLE_SUPERVISOR,ROLE_TELLER
    </value>
  </property>
</bean>

```

很明显, `FilterSecurityInterceptor` 引用了认证管理器(认证操作)和访问决定管理器(授权操作)。请开发者注意, `objectDefinitionSource` 属性定义了受保护的 Web 资源。其取值类似于 `ChannelProcessingFilter` 中的 `filterInvocationDefinitionSource` 属性。Acegi 安全框架为它提供了 `FilterInvocationDefinitionSourceEditor` 属性编辑器。它同时也是 `objectDefinitionSource` 属性的编辑器。

图 17-24 给出了 `securityEnforcementFilter` 过滤器的图形化表示。其中, 对不相关节点作了简化处理。

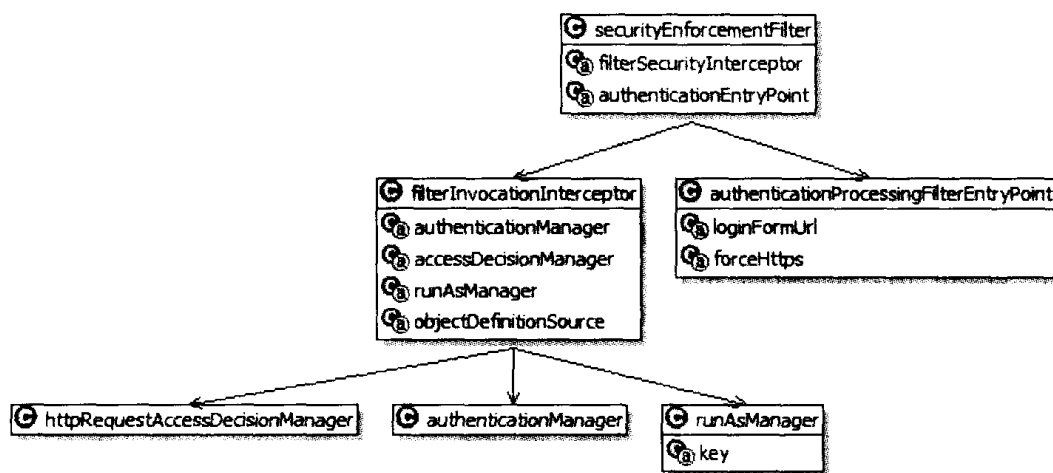


图 17-24 securityEnforcementFilter 过滤器

17.2.6 方法级的认证和授权

在开发 EJB 应用时, 开发者能够针对 EJB 的具体方法使用声明式安全性, 而在 Acegi 使能应用中开发者也可以存在类似的行为。

在借助于 Servlet 过滤器对 Web 资源进行受保护资源操作时，开发者最后需要借助于 SecurityEnforcementFilter 过滤器。其中，使用了 FilterSecurityInterceptor。此处，开发者需要为应用在方法级进行类似的安全性认证和授权操作，因此 Acegi 安全框架为开发者提供了 net.sf.acegisecurity.intercept.method.MethodSecurityInterceptor 拦截器。比如 contacts.war 中的 backendContactManagerSecurity 定义如下。

```
<bean id="backendContactManagerSecurity"

class="net.sf.acegisecurity.intercept.method.MethodSecurityInterceptor">
  <property name="authenticationManager">
    <ref local="authenticationManager"/>
  </property>
  <property name="accessDecisionManager">
    <ref local="businessAccessDecisionManager"/>
  </property>
  <property name="runAsManager">
    <ref local="runAsManager"/>
  </property>
  <property name="objectDefinitionSource">
    <value>
      sample.contact.ContactManager.delete=ROLE_RUN_AS_SERVER
      sample.contact.ContactManager.getAllByOwner=
        ROLE_RUN_AS_SERVER
      sample.contact.ContactManager.save=ROLE_RUN_AS_SERVER
      sample.contact.ContactManager.getById=ROLE_RUN_AS_SERVER
    </value>
  </property>
</bean>
```

其中使用的各个属性同 FilterSecurityInterceptor 中的没有任何区别。但是，这里的 MethodSecurityInterceptor 拦截器是借助于 Spring AOP 实现的，而 FilterSecurityInterceptor 完成用户认证和授权操作的过程是借助于 Servlet 过滤器实现的。开发者必须认清这一点。而且，MethodSecurityInterceptor 针对的是方法；FilterSecurityInterceptor 针对的是客户 Servlet 请求。但无论如何，它们共用了 Acegi 提供的安全性认证和授权操作。方法调用和 Servlet 请求只是在表现层不同而已，这在某种程度上符合 MVC 模式。形式与内容并重的 Acegi 使得开发者在日常企业应用开发中会慢慢熟悉，并喜欢上它。

至于 MethodSecurityInterceptor 的其他内容，本书不再论述。在开发者熟悉 Web 资源的认证和授权后，使用 MethodSecurityInterceptor 轻而易举。

17.3 其他内容

Acegi 包含的内容太丰富了（发展速度也很快，可能在本书出版时又新加了不少内容），而且 contacts.war 使用到 Acegi 的各个方面。借助于 SpringViz，开发者能够获得图 17-25 所示的 Spring JavaBean 层次结构图。通过该图能够对 Spring 配置文件有全面的认识，开发

者能够从宏观上认识清楚整个 contacts.war 应用涉及到的方方面面。通常，在开发 Spring 使用应用中，开发者都需要通过分析这种图形化表示，从而对整个应用系统有更全面、深入的认识。进而，在此基础上，能够对系统进行重构、模块化等工作。

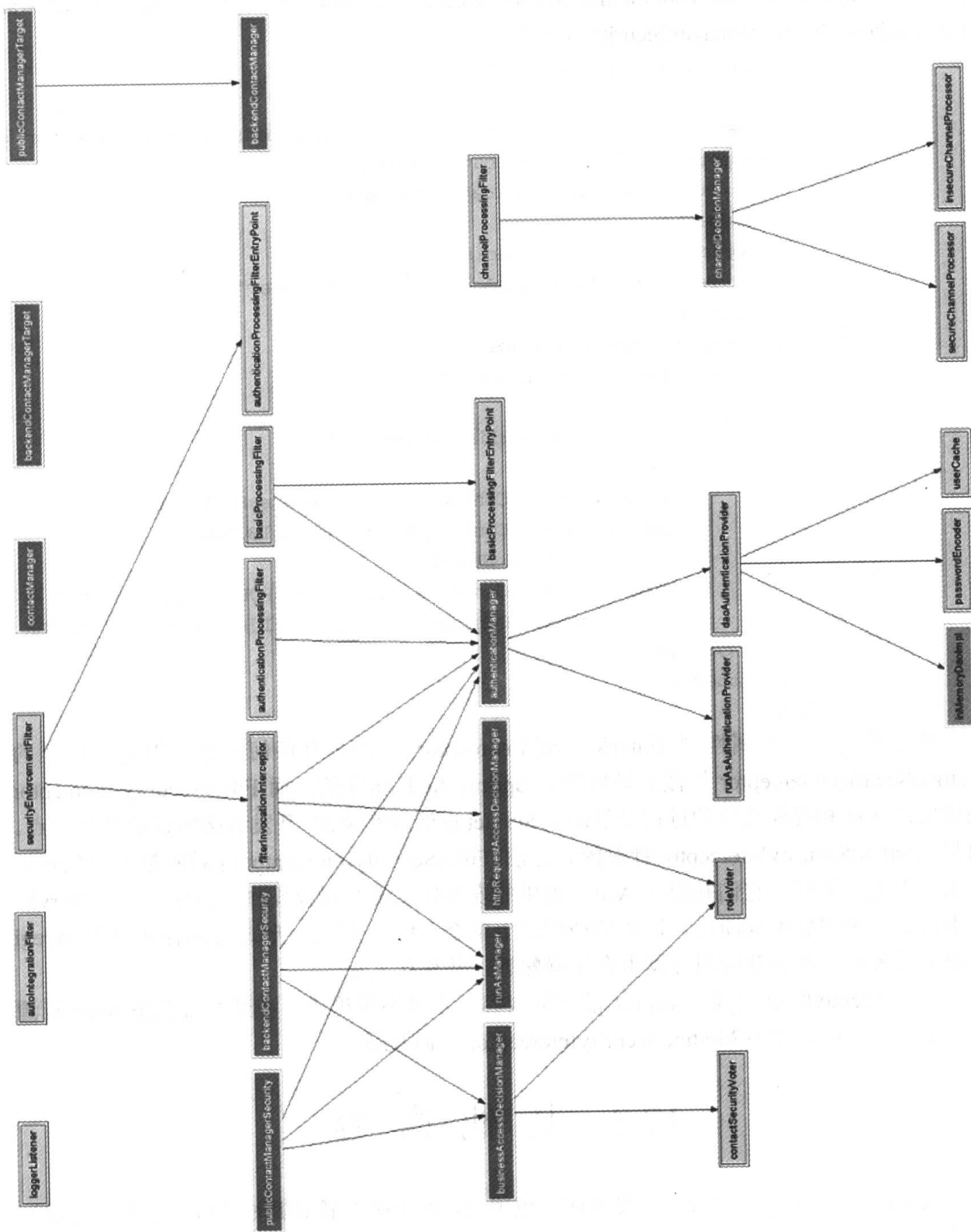


图 17-25 借助于 SpringViz 生成 applicationContext.xml 的结构图

17.3.1 实现密码的加密处理

默认时, DaoAuthenticationProvider 使用的密码是明文, 即未进行加密处理。如果内存或者 RDBMS 中存储的密码是进行过加密处理的, 则在对用户认证之前, 需要使用 PasswordEncoder 对明文 (即用户输入的密码) 进行转换, 从而可以保证认证过程的顺利进行。Acegi 为实现密码的加密处理内置了很多 PasswordEncoder 接口实现, 见图 17-26。

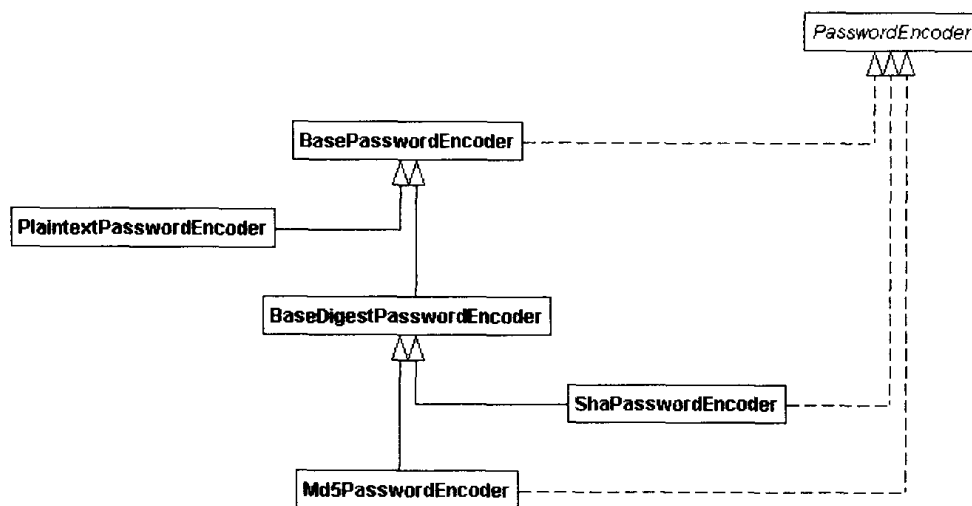


图 17-26 PasswordEncoder 接口及其子类

其中, 处于节点的 3 个 Encoder 的含义分别为:

- PlaintextPasswordEncoder: Acegi 默认使用的加密处理器。它并不对密码进行任何修改, 而是直接返回密码给其客户。
- Md5PasswordEncoder: 对密码进行 MD5 (Message Digest Algorithm) 加密处理。
- ShaPasswordEncoder: 对密码进行 SHA (Secure Hash Algorithm) 加密处理。

对于 contacts.war 中的 “daoAuthenticationProvider” 而言, 由于它使用了 MD5 算法对密码进行处理, 因此开发者还需为它指定密钥 (salt)。通过 DaoAuthenticationProvider 的 saltSource 能够指定密钥 (salt)。目前, Acegi 仅提供了如下两种 salt。

- net.sf.acegisecurity.providers.dao.salt.ReflectionSaltSource: 使用静态的字符串供单个用户使用。由于 ReflectionSaltSource 使用用户信息加密密码, 因此更安全 (相比 SystemWideSaltSource 而言)。其中, 开发者必须指定 userPropertyToUse 属性并给出非空值, 否则 Spring 的 InitializingBean (借助于 afterPropertiesSet 方法检查 userPropertyToUse 的取值) 将抛出 IllegalArgumentException 异常。比如 username。
- net.sf.acegisecurity.providers.dao.salt.SystemWideSaltSource: 使用静态的字符串供所有用户使用。在大部分场合, 本书都推荐直接使用 SystemWideSaltSource。其中, 开发者必须指定 systemWideSalt 属性并给出非空值, 否则 Spring 的 InitializingBean (借助于 afterPropertiesSet 方法检查 systemWideSalt 的取值) 将抛出 IllegalArgumentException 异常。比如 “luoshifei”。

如果使用 SystemWideSaltSource, 则修改后的 “daoAuthenticationProvider” 为:

```
<bean id="daoAuthenticationProvider"

class="net.sf.acegisecurity.providers.dao.DaoAuthenticationProvider">
  <property name="authenticationDao">
    <ref local="inMemoryDaoImpl"/>
  </property>
  <property name="userCache">
    <ref local="userCache"/>
  </property>
  <property name="passwordEncoder">
    <ref local="passwordEncoder"/>
  </property>
  <property name="saltSource">
    <bean class="net.sf.acegisecurity.providers.dao.
      salt.SystemWideSaltSource">
      <property name="systemWideSalt">
        <value>luoshifei</value>
      </property>
    </bean>
  </property>
</bean>
```

请注意粗体部分，借助于 Spring 内部 JavaBean 语法实现了 SystemWideSaltSource JavaBean 的定义。

如果使用 ReflectionSaltSource，则修改后的“daoAuthenticationProvider”为：

```
<bean id="daoAuthenticationProvider"
class="net.sf.acegisecurity.providers.
dao.DaoAuthenticationProvider">
  <property name="authenticationDao">
    <ref local="inMemoryDaoImpl"/>
  </property>
  <property name="userCache">
    <ref local="userCache"/>
  </property>
  <property name="passwordEncoder">
    <ref local="passwordEncoder"/>
  </property>
  <property name="saltSource">
    <bean class="net.sf.acegisecurity.providers.
      dao.salt.ReflectionSaltSource">
      <property name="userPropertyToUse">
        <value>username</value>
      </property>
    </bean>
  </property>
</bean>
```

注意，默认时“daoAuthenticationProvider”未指定 saltSource，即 saltSource 取值为 null。

17.3.2 缓存用户信息

如果不对用户信息进行缓存，则客户每次请求受保护资源时，认证管理器都需要从底层持久源（比如 LDAP、XML、RDBMS、AD）中获得用户信息。这显然不利于应用效率，尤其是应用系统的认证要求较多时（此时，底层持久源将成为整个企业应用的性能瓶颈）。通常，用户的认证信息不可能随时变更，因此如果将用户信息缓存起来，使得它在一定的时限内有效，则这将大大提高应用系统的性能。

为实现用户信息的缓存，Acegi 框架提供了 `net.sf.acegisecurity.providers.dao.UserCache` 接口，而且同时提供了相应的接口实现，见图 17-27。

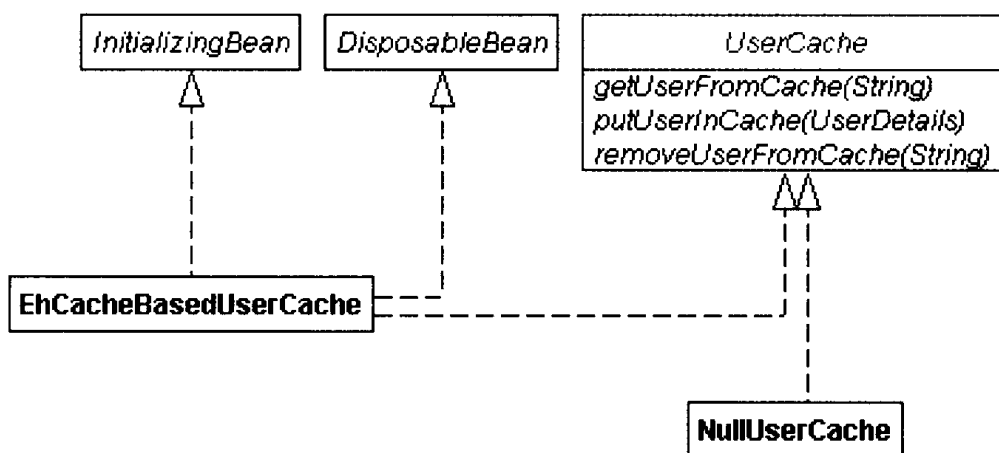


图 17-27 UserCache 接口及其子类

可以看出，UserCache 接口提供的方法使得删除缓存中的用户信息、往缓存中添加用户信息、从缓存中获得用户信息都可以实现。注意，NullUserCache 和 EhCacheBasedUserCache 的含义如下。

- `net.sf.acegisecurity.providers.dao.cache.NullUserCache`: 默认使用的 UserCache 实现。如果调用 `getUserFromCache()` 方法，则将返回 `null`。因此，这同未设置 UserCache 几乎没有区别。
- `net.sf.acegisecurity.providers.dao.cache.EhCacheBasedUserCache`: 通常，如果应用需使用用户缓存，则可以使用 `EhCacheBasedUserCache` 实现。`EhCache`² 是 100% Java 实现的、进程内缓存实现。因为速度快、使用简单、内存要求低等优点，使得许多产品部署场合都使用了它，其中包括 Hibernate。

当然，开发者也可以提供自身的 UserCache 接口实现。Acegi 的可扩展性使得开发者能够开发出满足自身业务需求的 Acegi 安全性框架。在 `contacts.war` Web 应用中，配置使用了 `EhCacheBasedUserCache`。其中，存在的 `minutesToIdle` 属性能够配置缓存信息的有效时限。比如，5 分钟后将失效缓存的用户信息。因此，在 5 分钟后，一旦有客户访问受保护的资源，则认证管理器需要再次去底层的持久化存储源中查找用户安全性信息。

² <http://ehcache.sourceforge.net/>

```
<bean id="userCache" class="net.sf.acegisecurity.providers.  
    dao.cache.EhCacheBasedUserCache">  
    <property name="minutesToIdle">  
        <value>5</value>  
    </property>  
</bean>
```

开发者应该注意到, `DaoAuthenticationProvider` 存在 `userCache` 属性。因此, 在配置它时, 只需要将 `userCache` POJO 挂到 `DaoAuthenticationProvider` POJO 配置上即可。

17.4 小 结

本章内容给出了 Acegi 安全框架基础知识的介绍。当前, 在现存的 Java 平台中, Acegi 是最好的安全框架之一。借助于 Spring 提供的 IoC 和 AOP, Acegi 实现了与业务逻辑的松耦合, 因此这使得 Acegi 的架构和设计初衷符合企业级应用的需求。本章不仅通过配置 Spring 应用的形式 (`contacts.war`) 而使用了 Acegi, 而且也展示了 Acegi 功能强大的一面 (比标准的 J2EE 安全性功能更为强大、更为灵活)。请注意, 开发者即使不打算使用 Spring, 本书也建议使用 Acegi, 或者研究 Acegi 框架的架构。借助于 Acegi 提供的安全性抽象, 开发者能够高效地使用它。在开发者几乎不用开发任何 (与安全性相关) Java 代码的前提下, 就可以架构并设计出一流的、受保护的企业级应用。

在此, 作者建议读者再次回到第一、二部分内容中, 这样将大大加深对 Spring 的掌握程度。

更加详细的内容, 请参考 Acegi 官方网站。另外, 在作者维护的 <http://www.open-v.com> 网站也将提供大量的 Acegi 及 Spring 资料介绍。

附录 A 实例代码安装

A.1 代码说明

全书的实例代码是基于如下环境开发完成的。

- 物理机器：IBM Thinkpad R50 笔记本（配置：CPU 1.4G、RAM 768M、硬盘 30GB）
- 操作系统：Windows XP Professional 简体中文版
- 开发工具：Eclipse 3.0.1、JBoss IDE 1.4、Spring IDE 1.1、Tapestry Spindle、Hibernate Synchronizer。它们都是功能强大的免费工具。
- J2EE 应用服务器：JBoss 4.0.0/4.0.1
- 数据库：MySQL 4.1.5-gamma-nt/4.0.23-debug
- JDK 1.5.x/1.4.x

A.2 钟情 JBoss

本书针对 Spring 的各个方面展开了论述和研究，尤其是从 Spring 简化开发者构建 J2EE 应用的角度详细给出了系统性的研究。可能很多开发者会问，为什么不使用 Tomcat（Jetty）作为开发服务器。当然，本书提供的大部分代码都能够在 Tomcat（Jetty）上运行，除了使用 JMS、EJB、基于 JNDI 获取 DataSource 等 Tomcat 未提供的功能外。

其实，本书认为，选用 JBoss 4.0.0 作为开发服务器比使用 Tomcat 更具优势。开发者可以试想，对于开发企业级应用系统而言，仅仅使用 Web 容器能够满足业务需求吗？如果因为 JBoss 4.0.0 提供了完整的 J2EE 平台技术而不去使用它，这部分开发者误解了 Spring 的初衷，误解了本书的初衷。具体解释如下。

- JBoss，本身将 Tomcat 容器作为其 Web 容器，而没有另外开发新的 Web 容器。因此，从开发者学习角度考虑，使用 JBoss 或者 Tomcat 是不存在冲突的。
- Tomcat，作为 JSP、Servlet 规范的参考实现，自然是遵循 J2EE 标准的，而 JBoss 已经通过了 J2EE 1.4 的认证。因此，使用 Tomcat 或者 JBoss 是不存在矛盾的。
- 从本书使用者的角度出发，如果选用 Tomcat，需要使用到它未提供的功能时（比如，JavaMail、JMS、JCA、EJB），额外地安装和学习其他的 J2EE 技术栈实现将会给读者带来不便。因此，从这一点可以看出，选用 JBoss 更具优势。
- 从产品功能角度考虑，JBoss 是 J2EE 应用服务器，它实现了整个 J2EE 平台技术栈；而 Tomcat 只是 JSP、Servlet 的参考实现。在工业强度上，Tomcat 不如 JBoss。Rod Johnson（也包括整个 Spring 开发 Team）对于 Spring 的贡献、对于 Java/J2EE 开

发者而言，确实是做了不错的工作。JBoss 同 Spring 的组合是否是开发 J2EE 应用中的杀手铜呢？本书可以给出肯定的答案。

A.3 工具下载与安装

Eclipse 3.0.1 的下载和安装，本书就不再介绍。

A.3.1 Spring IDE

Spring IDE 是 Spring 开发 Team 开发的，它是基于 Eclipse 插件方式运行的。本书写作之际，Spring IDE 1.1 还未正式发布。

通过 <http://springide-eclip.sourceforge.net/updatesite/> 能够下载到 Spring IDE 1.1。开发者在下载并安装 Spring IDE 后，需要通过如下方式来使用它。

首先，开发者需要打开 Spring Beans 视图。具体操作方式有两种，其一，通过“Navigate”菜单，如图 A-1 所示。

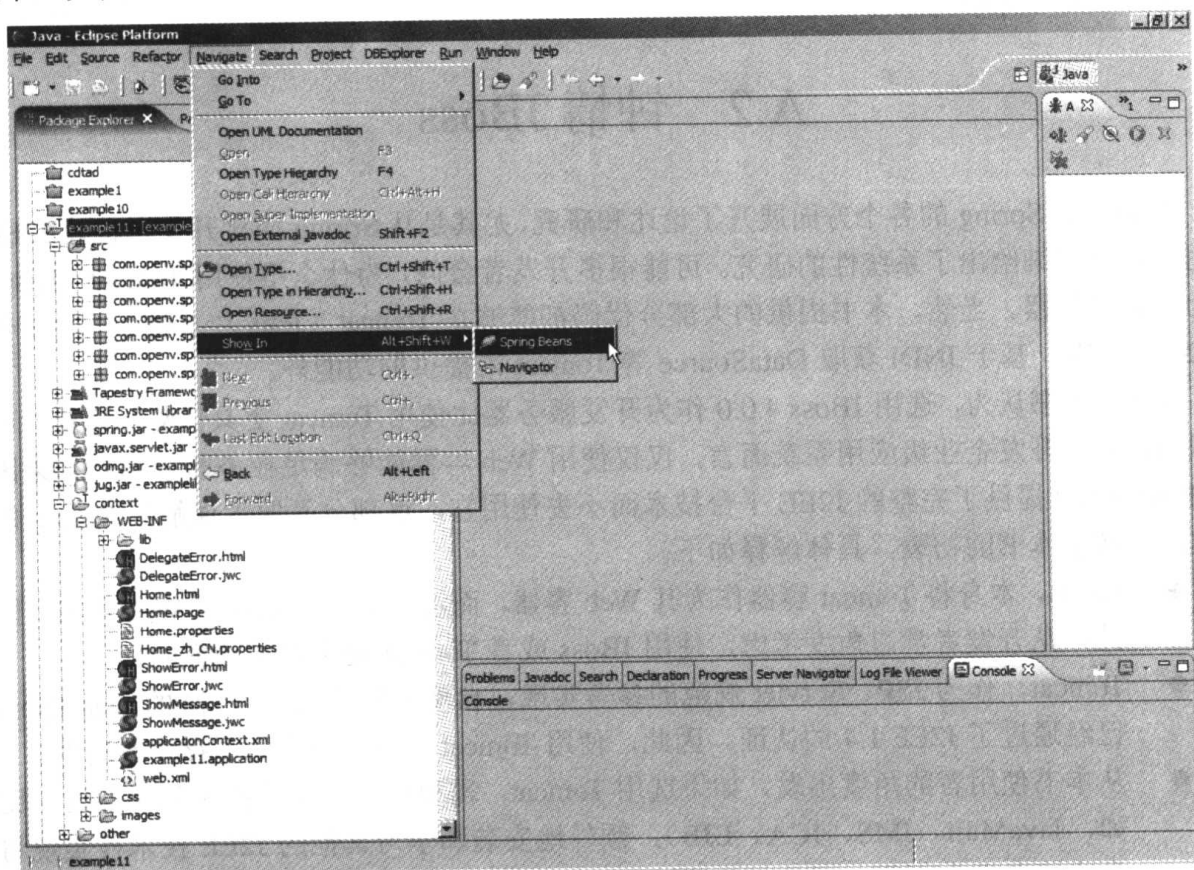


图 A-1 通过“Navigate”菜单打开 Spring Beans 视图

另外一种方式是，通过“Show View”对话框（位于“Window”→“Show View”→“Other”），如图 A-2 所示。

操作成功，开发者可以在右下角看到 Spring Beans 视图，见图 A-3。

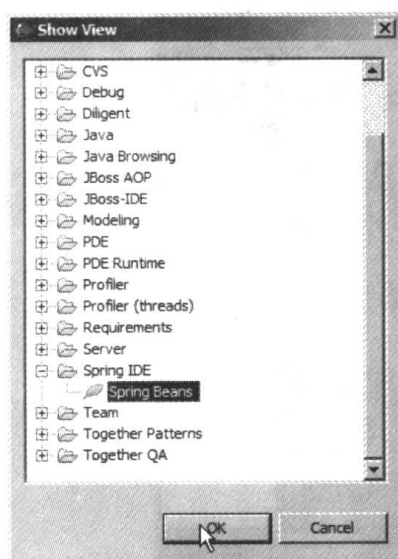


图 A-2 通过“Show View”对话框打开 Spring Beans 视图

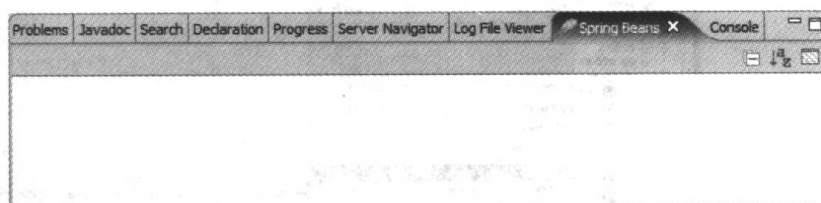


图 A-3 Spring Beans 视图

第二，将 Spring Beans 功能添加到项目中，见图 A-4。其中，添加 Spring Beans 功能主要是为了能够在编辑 Spring 配置文件时，Spring IDE 能够对配置文件进行校验和辅助开发。

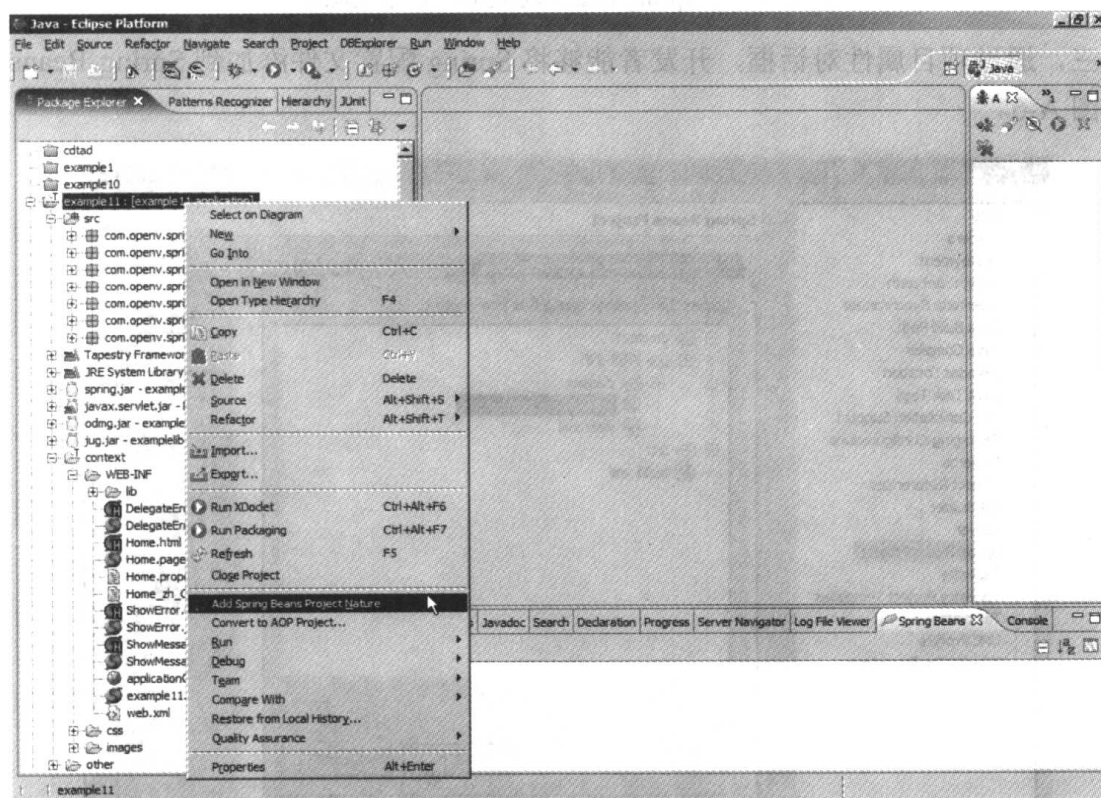


图 A-4 将 Spring Beans 功能添加到项目中

一旦操作成功，开发者也可以依据此菜单将 Spring Beans 功能从项目中删除掉，见图 A-5。

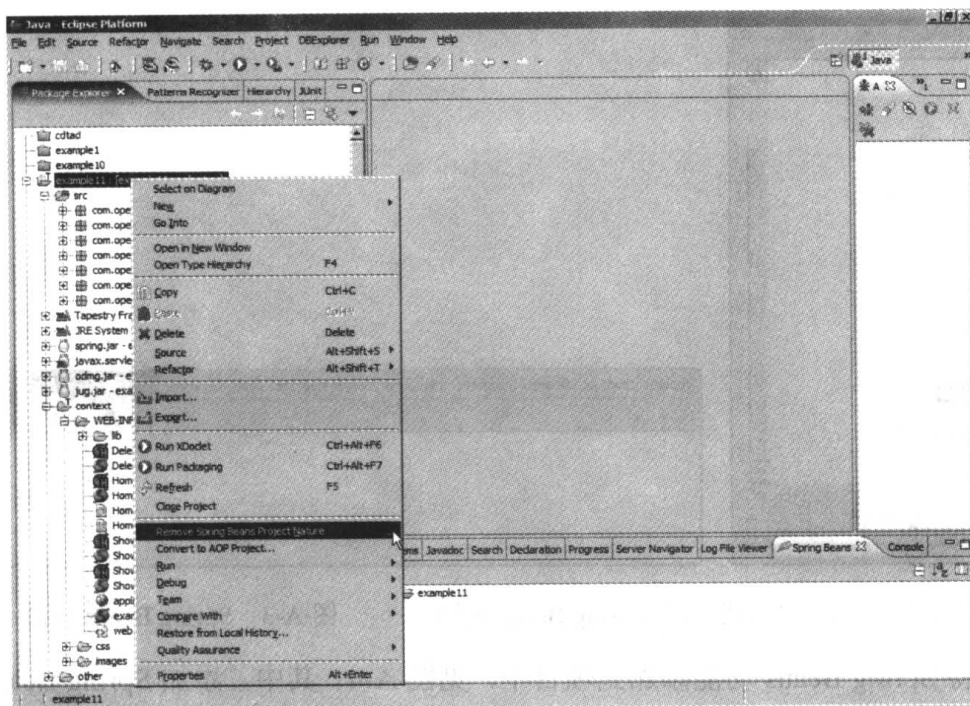


图 A-5 将 Spring Beans 功能从项目中删除掉

第三，通过项目属性对话框，开发者能够将 Spring 配置文件添加到 Spring Beans 视图中，见图 A-6。

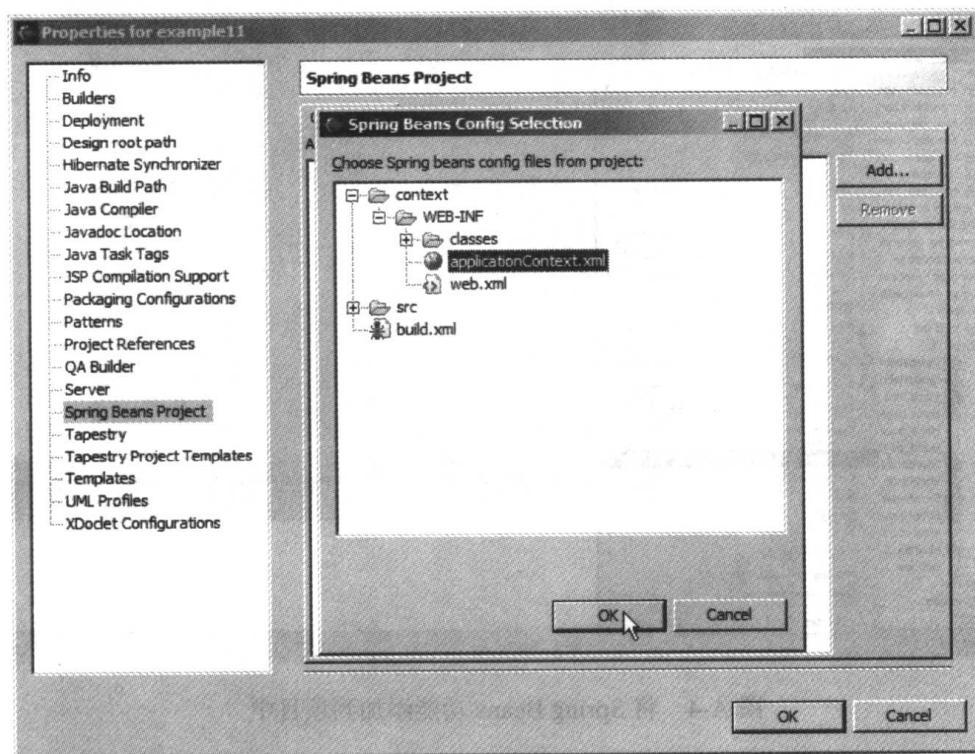


图 A-6 将 Spring 配置文件添加到 Spring Beans 视图

一旦操作成功，开发者可以在 Spring Beans 视图中看到已添加的 Spring 配置文件，见图 A-7。

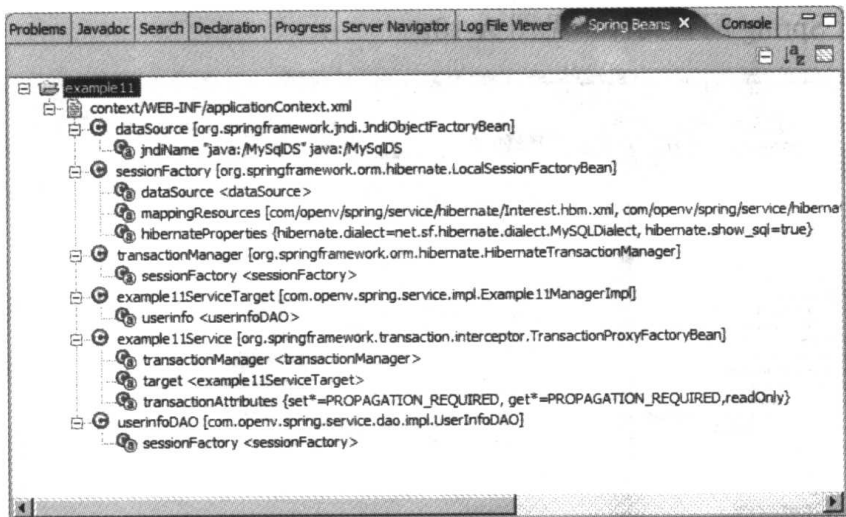


图 A-7 已添加 Spring 配置文件的 Spring Beans 视图

第四，以图形化方式¹查看 Spring 配置文件中定义的 JavaBean 命名空间，见图 A-8 和 A-9²。

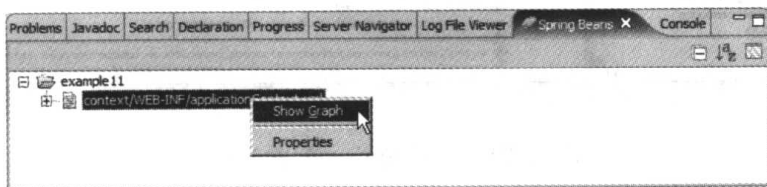


图 A-8 Show Graph 菜单项

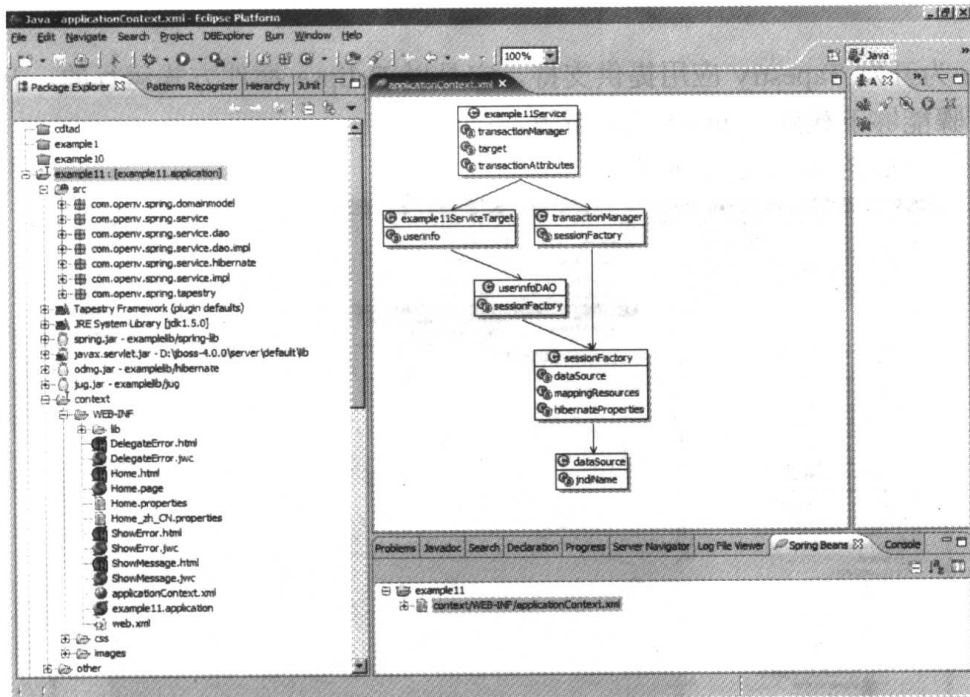


图 A-9 Spring 配置文件中定义的 JavaBean 命名空间

¹ 另一种以图形化方式查看 Spring 配置文件的工具，参见<http://www.samoht.com/wiki/wiki.pl?SpringViz>。

² 在这之前，开发者还需要安装 Eclipse GEF。Eclipse GEF 下载网址是<http://eclipse.org/gef/>。

第五, 编辑 Spring 配置文件, 见图 A-10。

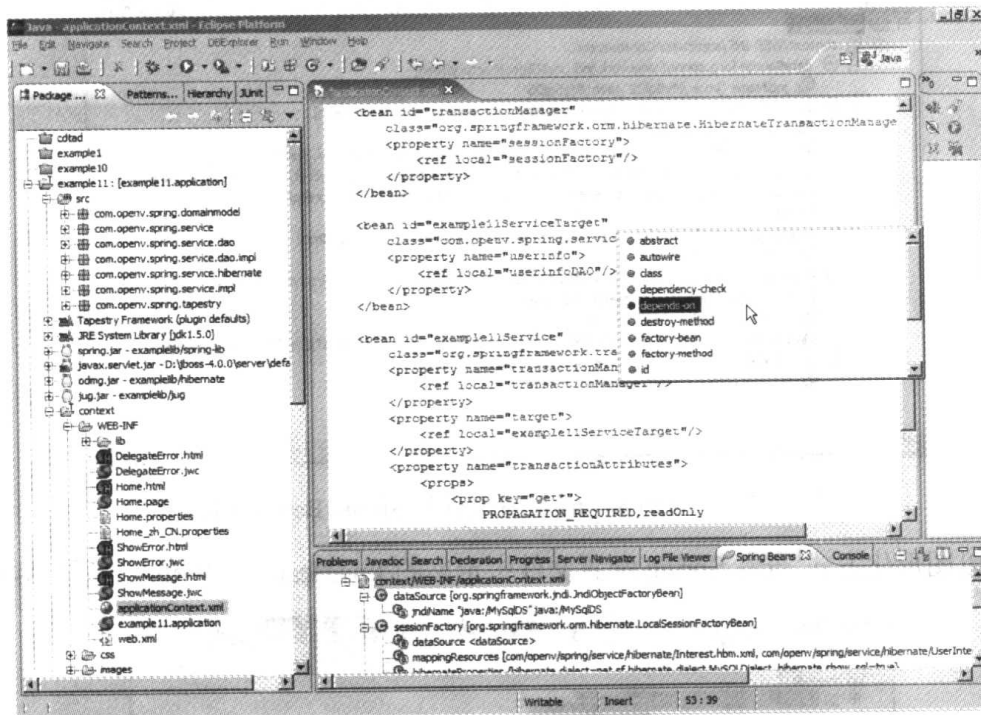


图 A-10 编辑 Spring 配置文件

A.3.2 Tapestry Spindle

Spindle 为开发 Tapestry 应用提供支持。它是以 Eclipse 插件方式运行的。开发者通过如下若干步骤能够下载到 Spindle 的最新版。

首先, 打开如图 A-11 所示的菜单。

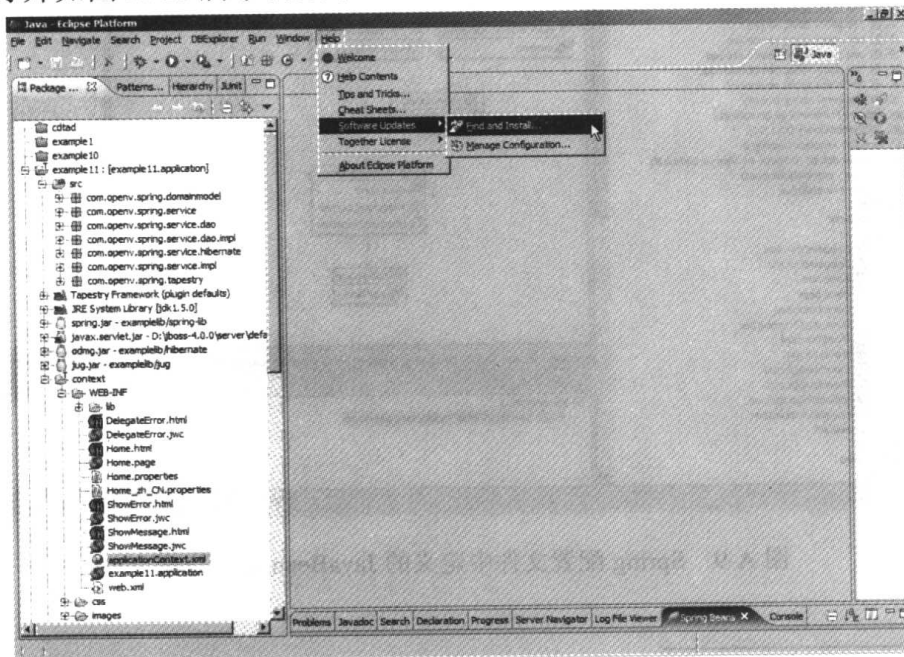


图 A-11 “Find and Install” 菜单项

然后，选择如图 A-12 所示的单选项。

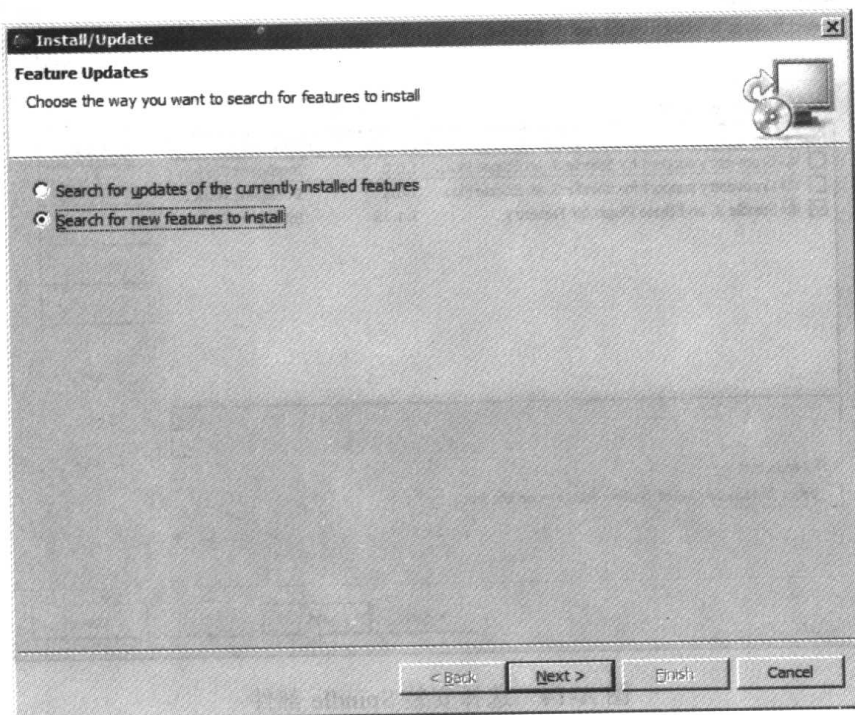


图 A-12 选择“Search for new features to install”单选项

第三，通过“New Remote Site”按钮，输入 <http://spindle.sourceforge.net/unstable/>，如图 A-13 所示。

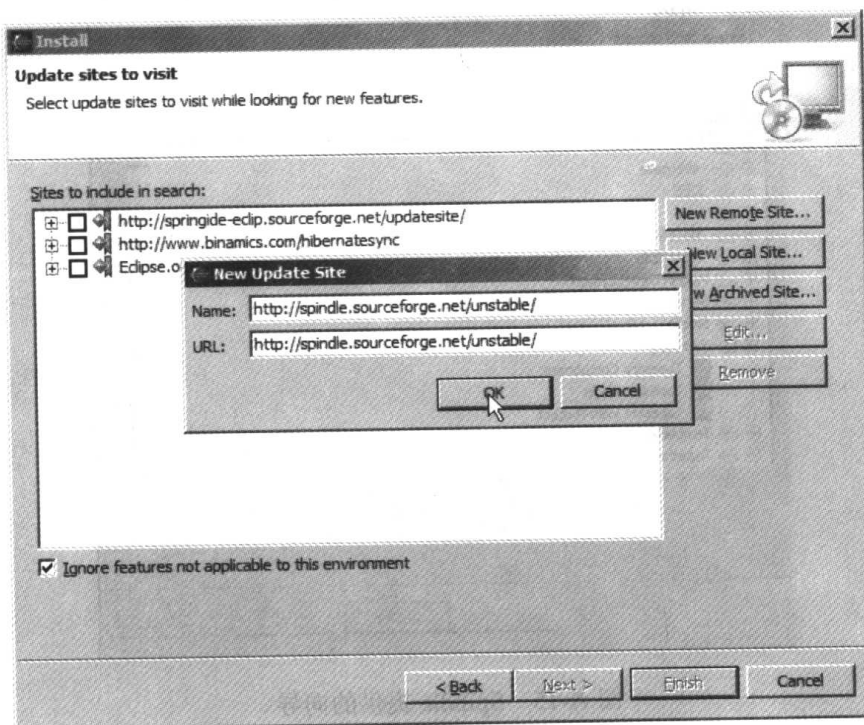


图 A-13 输入“<http://spindle.sourceforge.net/unstable/>”

第四，选中刚才输入的站点。然后，单击“Next”按钮。开发者可以安装最新的 Spindle 插件了，见图 A-14。

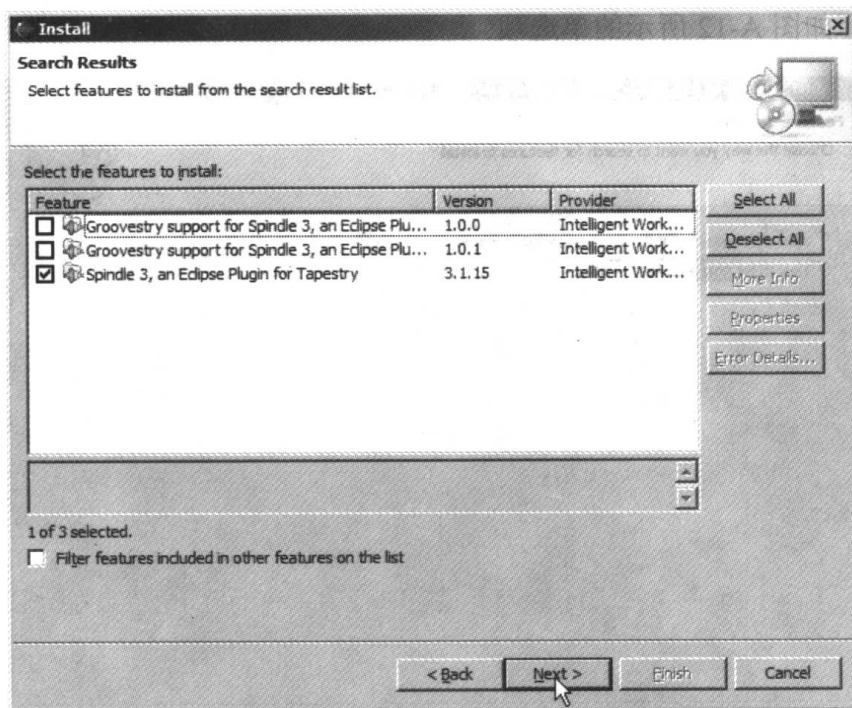


图 A-14 选择安装 Spindle 部件

第五, Tapestry Spindle 的使用。在成功安装 Spindle 后, 开发者可以用来开发 Tapestry 应用了, 图 A-15 给出了其提供的向导。

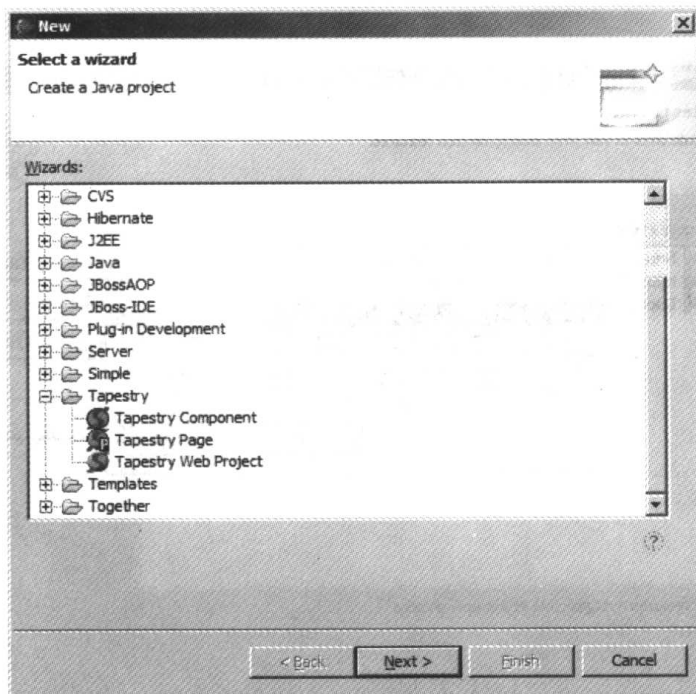


图 A-15 Spindle 提供的向导

一般而言, 开发者首先需要建立 Tapestry Web Project (见图 A-16), 然后开发 Tapstry Page 页面 (见图 A-17)。对于资深 Tapestry 开发者, 会涉及到 Tapestry Component 组件 (见图 A-18) 的开发。

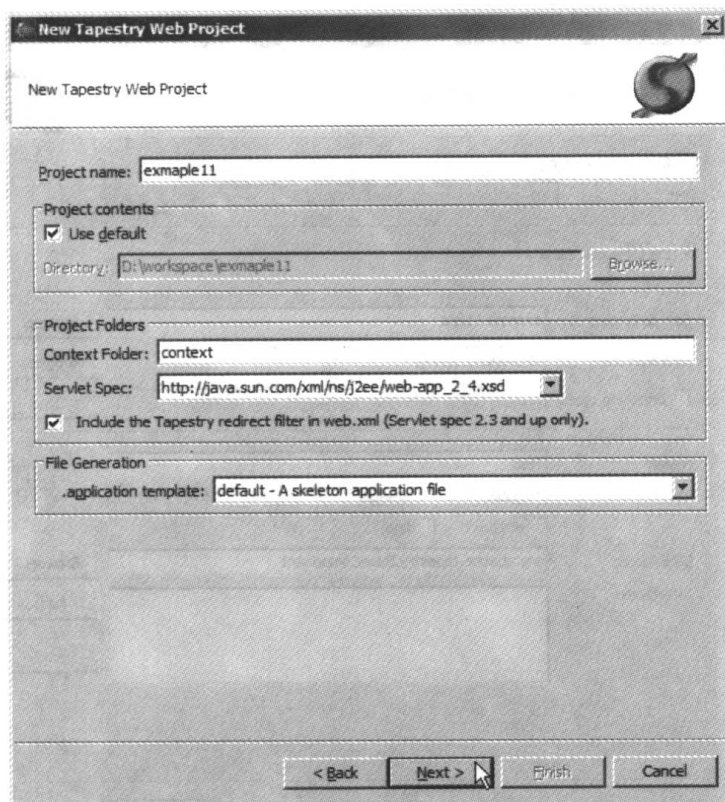


图 A-16 Spindle 提供的 Tapestry Web Project 向导

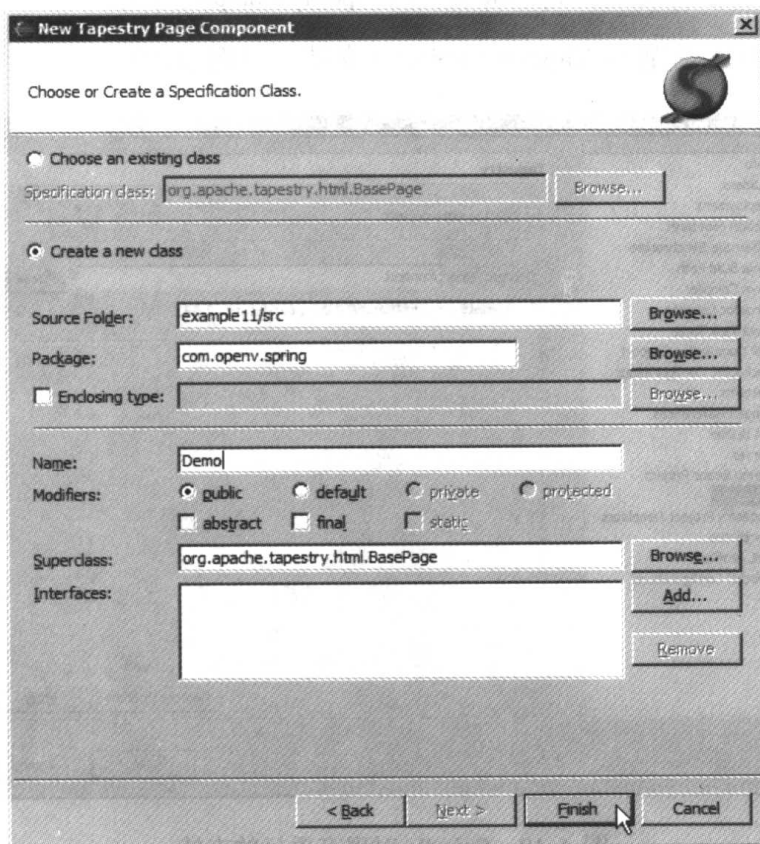


图 A-17 Spindle 提供的 Tapestry Page 向导

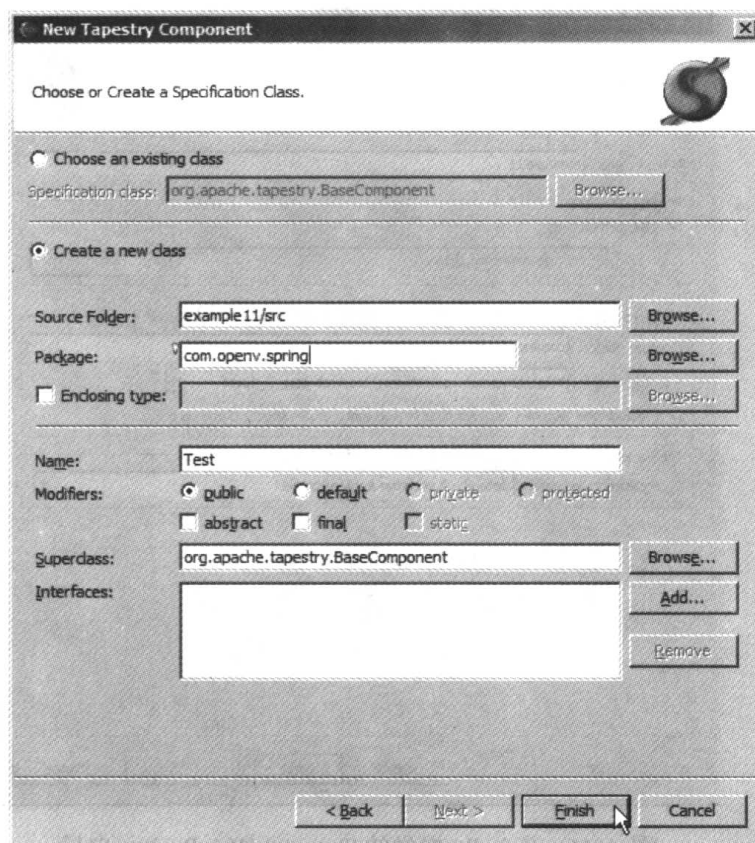


图 A-18 Spindle 提供的 Tapestry Component 向导

第六，将现有项目配置成 Tapestry Spindle 支持，见图 A-19。

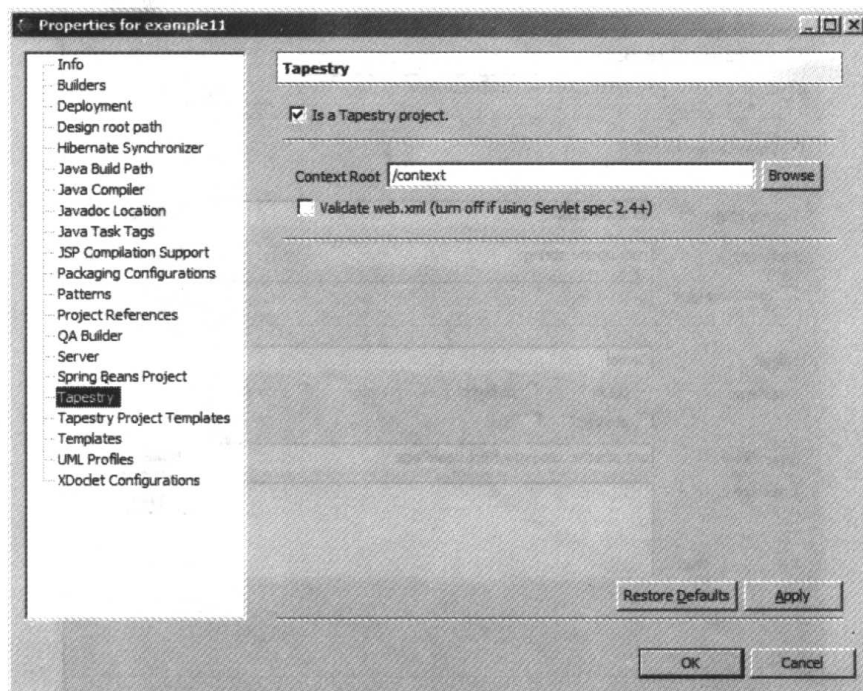


图 A-19 Spindle 对现有项目的支持

对于 Tapestry 项目，会在项目中加上“T”型标志，见图 A-20。

[illegible]

图 A-21 Tapestry Spindle 功能 1 (HTML 模板中组件属性智能提示)

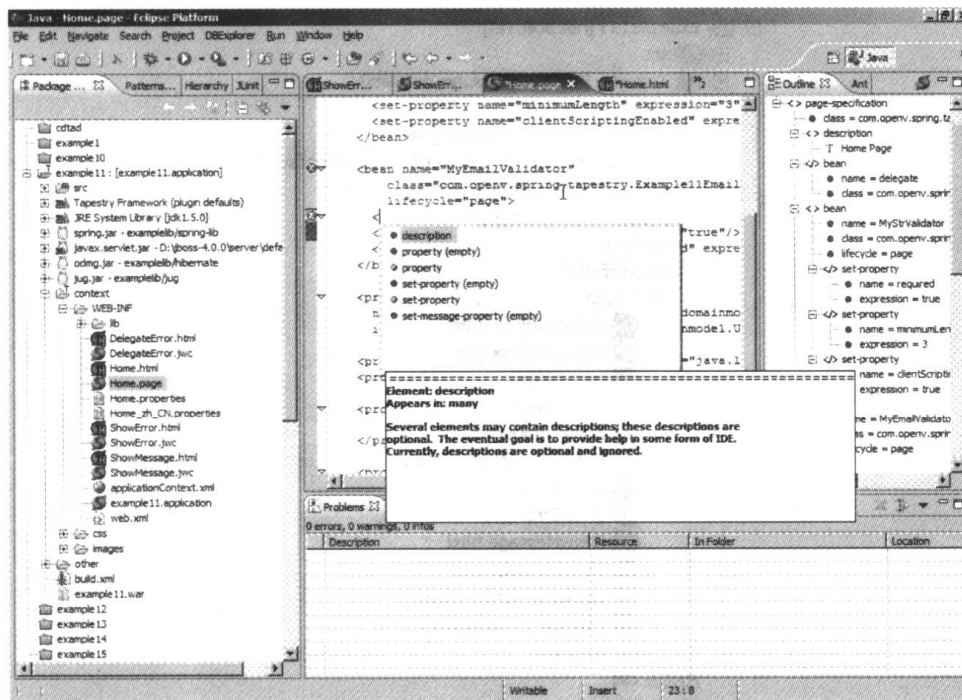


图 A-22 Tapestry Spindle 功能 2 (PAGE 中组件属性智能提示)

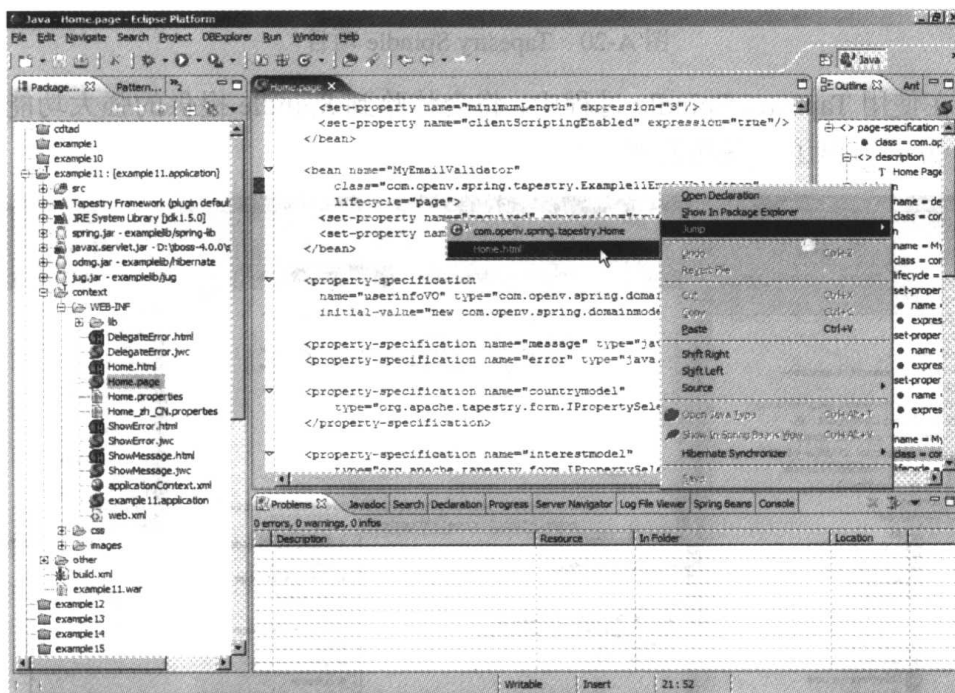


图 A-23 Tapestry Spindle 功能 3 (与代码、HTML 模板的智能关联)

A.3.3 JBoss IDE

JBoss IDE 是 JBoss Group 官方开发的、免费的开发工具。它是以 Eclipse 插件方式运行的，图 A-24 为其主页。

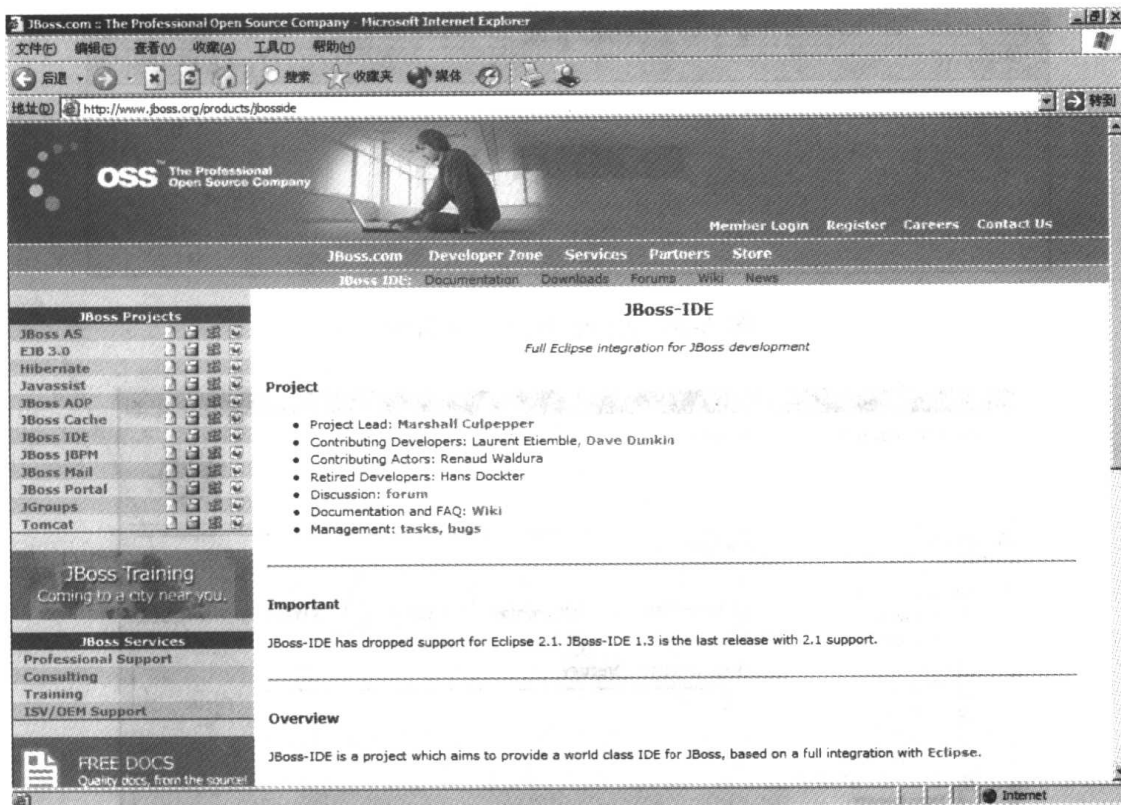


图 A-24 JBoss IDE 主页

为下载和安装 JBoss IDE，开发者需要根据如下若干步骤进行。

首先，去 http://prdownloads.sourceforge.net/jboss/?sort_by=date&sort=desc 下载 JBoss IDE，见图 A-25 所示。

jboss-4.0.1RC1.tar.bz2.MD5	0 kb	Nov 04, 2004 10:38
jboss-4.0.1RC1.tar.bz2	89103 kb	Nov 04, 2004 10:38
jboss-4.0.1RC1-src.tar.gz.MD5	0 kb	Nov 04, 2004 10:25
jboss-4.0.1RC1-src.tar.gz	54767 kb	Nov 04, 2004 10:25
jboss-4.0.1RC1-src.tar.bz2.MD5	0 kb	Nov 04, 2004 10:18
jboss-4.0.1RC1-src.tar.bz2	51969 kb	Nov 04, 2004 10:18
jboss-aop-1.0.0-FINAL.zip	12171 kb	Nov 01, 2004 06:22
org.jboss.ide.eclipse-1.4.0.bin.dist.zip	15784 kb	Oct 25, 2004 17:37
jboss-ide-aop-1.0.1.zip	2967 kb	Oct 25, 2004 17:34
jboss-3.2.6.zip.MD5	0 kb	Oct 13, 2004 22:34
jboss-3.2.6.zip	52864 kb	Oct 13, 2004 22:34
jboss-3.2.6.tar.gz.MD5	0 kb	Oct 13, 2004 22:31
jboss-3.2.6.tar.gz	52149 kb	Oct 13, 2004 22:31
jboss-3.2.6.tar.bz2	51850 kb	Oct 13, 2004 22:29
jboss-3.2.6.tar.bz2.MD5	0 kb	Oct 13, 2004 22:29
jboss-3.2.6-src.tar.gz.MD5	0 kb	Oct 13, 2004 22:26
jboss-3.2.6-src.tar.gz	38149 kb	Oct 13, 2004 22:26
jboss-3.2.6-src.tar.bz2	36576 kb	Oct 13, 2004 22:24

图 A-25 JBoss IDE 下载页面

其次，在安装 JBoss IDE 后，开发者首先需要在控制台打开 Server Navigator 视图，见图 A-26。然后，需要配置 JBoss 4.0.0 服务器，见图 A-27。

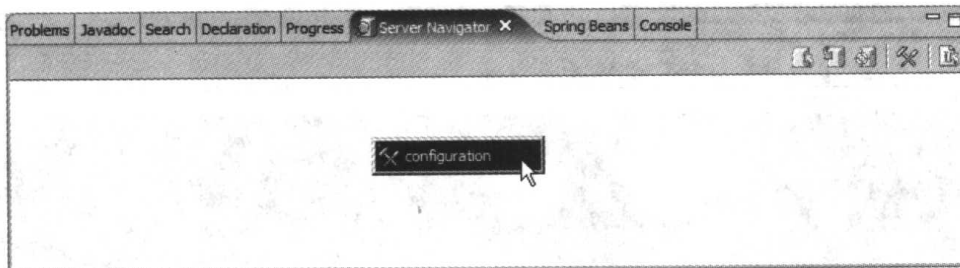


图 A-26 Server Navigator 视图

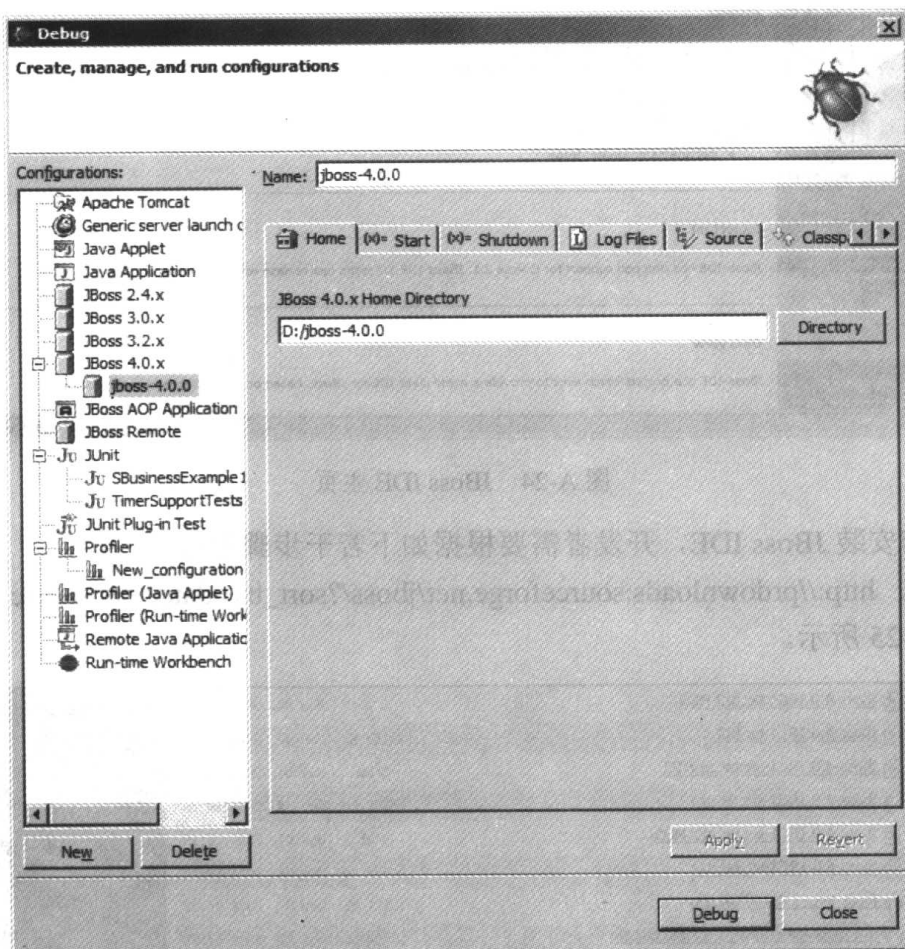


图 A-27 配置 JBoss 4.0.0 应用服务器

第三，开发者还可以通过 Log File Viewer 视图（图 A-28）查看服务器日志（图 A-29）。



图 A-28 Log File Viewer 视图

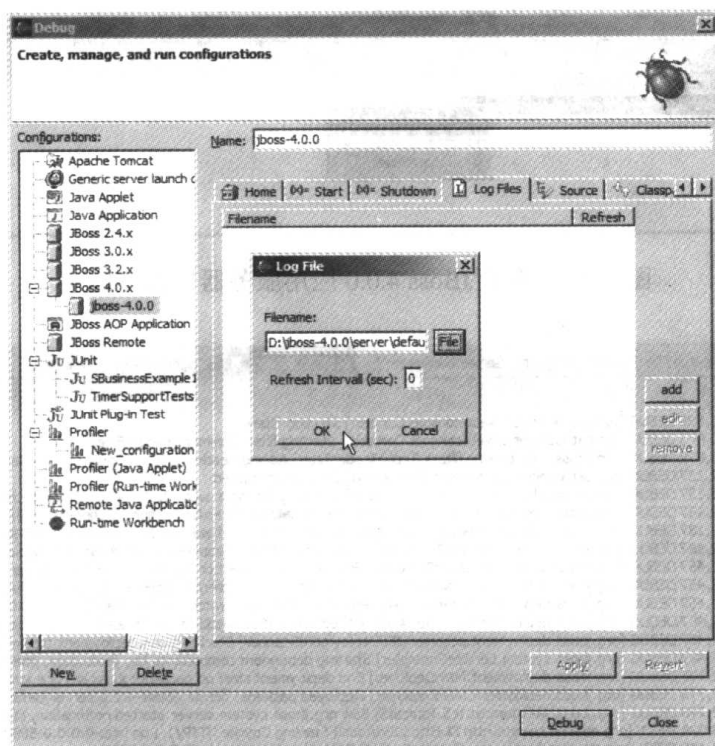


图 A-29 查看 JBoss 4.0.0 应用服务器日志

第四，启动 JBoss 4.0.0 应用服务器（图 A-30、A-31），并且查看服务器日志内容（图 A-32、A-33）。

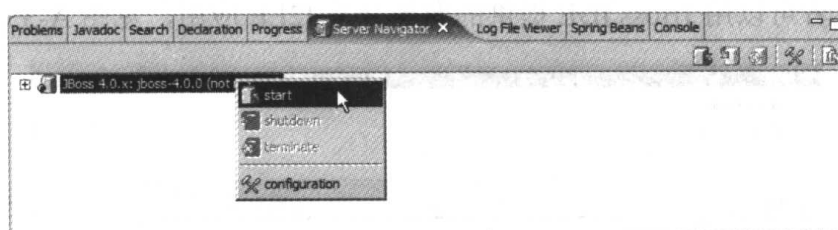


图 A-30 启动 JBoss 4.0.0 应用服务器

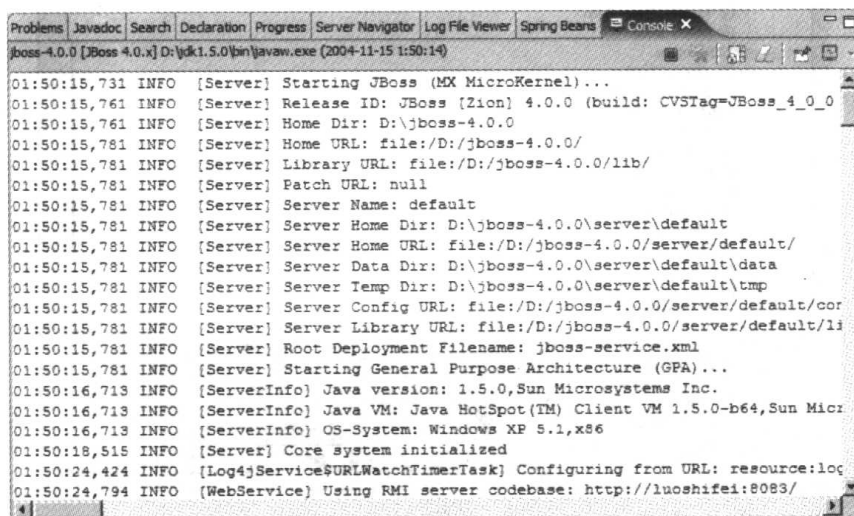


图 A-31 JBoss 4.0.0 应用服务器启动过程

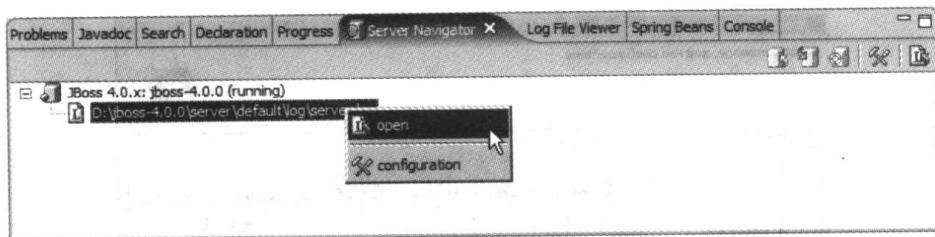


图 A-32 查看 JBoss 4.0.0 应用服务器日志 (1)

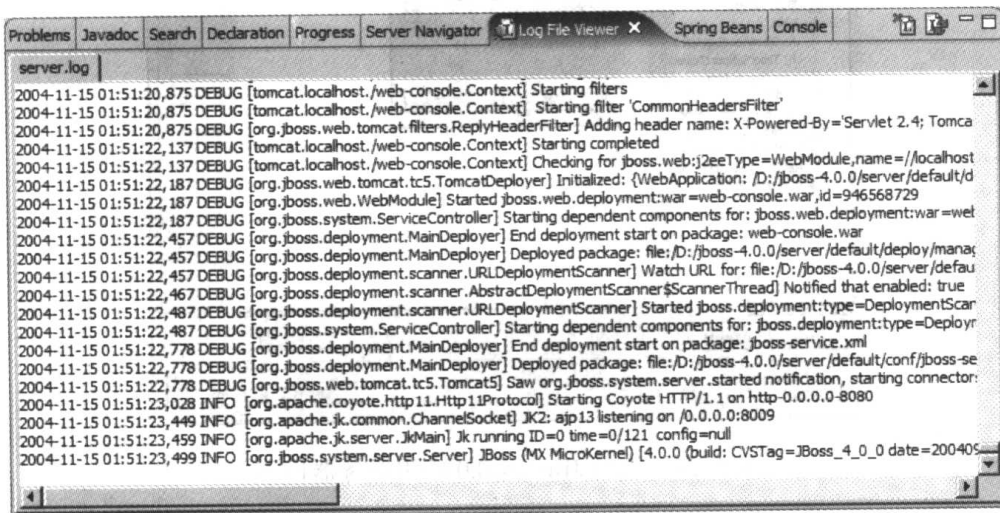


图 A-32 查看 JBoss 4.0.0 应用服务器日志 (2)

第五, 开发者可以用 JBoss IDE 提供的向导开发 J2EE 应用, 见图 A-33。

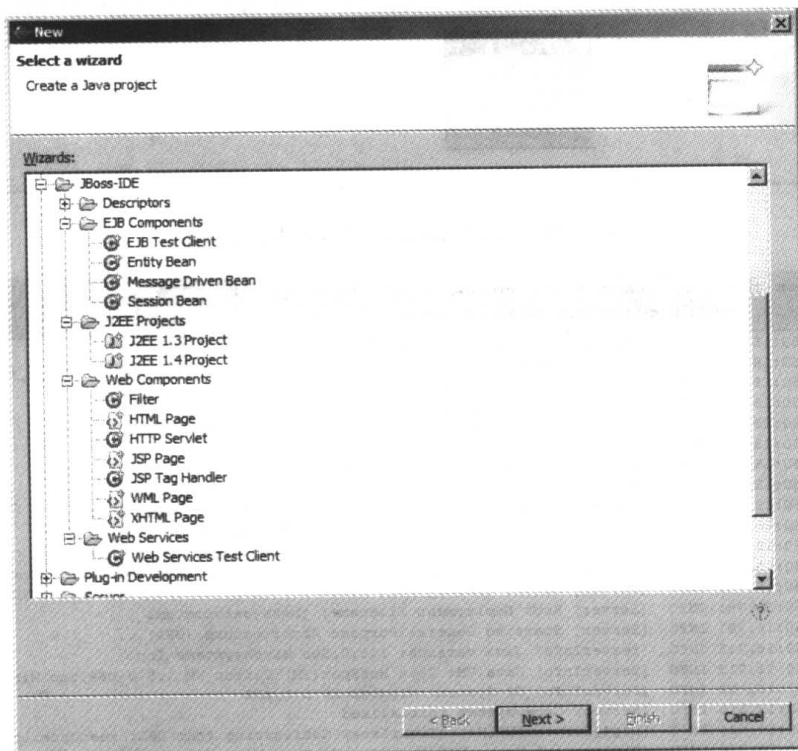


图 A-33 JBoss IDE 向导

A.3.4 Hibernate Synchronzier

Hibernate Synchronzier 是基于 Eclipse 插件方式运行的。本书在第 11 章给出了详细介绍。

A.4 代码使用

本书提供的开发实例有 29 个, 其中包括 Java/J2EE 应用。它们都是以 Eclipse 项目组织的, 因此只要开发者使用 Eclipse IDE, 便能够很容易使用它们。对于代码的具体使用, 开发者需要注意如下几个约定。

本书假定工具的安装路径如下。

- Eclipse: D:\eclipse。其中, Eclipse 使用的工作空间位于 D:\workspace。
- JBoss: D:\jboss-4.0.0
- JDK 5.0: D:\jdk1.5.0

开发者需要在使用代码前建立 examplelib 项目, 以存储 29 个实例所要求的 jar 库。当然, 开发者可以根据实际情况, 而做相应的调整。其中, examplelib 内容如图 A-34 所示。

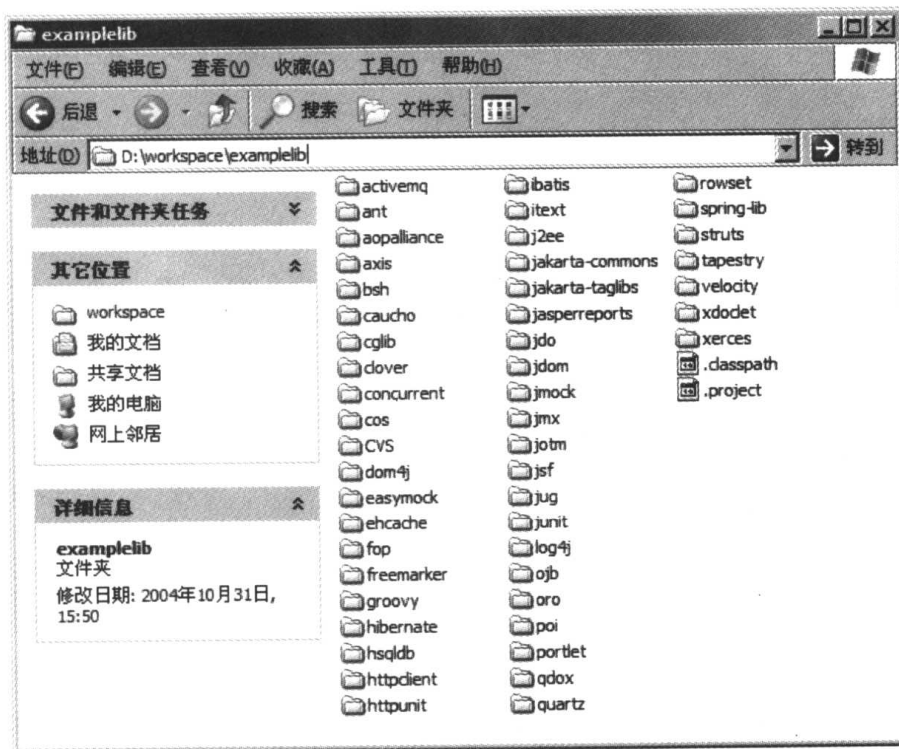


图 A-34 examplelib 项目内容

请注意, spring-lib 的内容包含了 Spring 框架发布包的所有 jar 文件。其他目录的具体内容都是从 Spring 框架根目录的 lib 中拷贝的。

附录 B spring-beans.dtd 的内容模型

本附录将研究 spring-beans.dtd 的内容模型，使得开发者能够对 spring-beans.dtd 有全局的认识。至于对 spring-beans.dtd 更具体的内容，则需要参考 spring-beans.dtd 文件本身。

如果开发者基于 spring-beans.dtd 配置 Spring 应用，则需要在相应的 XML 配置文件中加上如下 DOCTYPE，供对 Spring 配置文件验证使用。

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
```

B.1 beans 节点

图 B-1 给出了 spring-beans.dtd 中的根节点，即 beans 的内容模型，基于该 DTD 生成的 Spring XML 配置文件的内容结构。

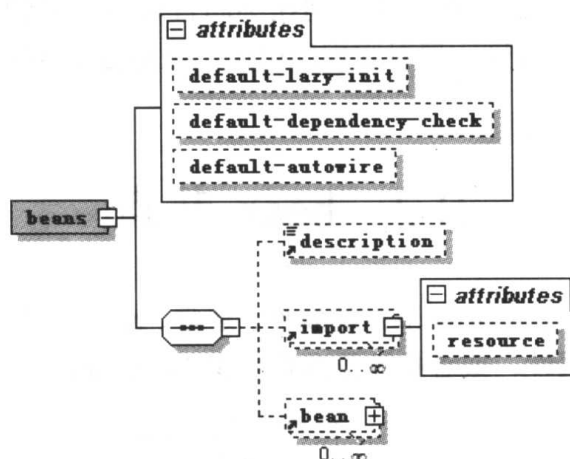


图 B-1 beans 节点的内容模型

其中，beans 的属性解释如表 B-1 所示。

表 B-1 beans 属性

属 性	定 义	描 述
default-lazy-init	<!ATTLIST beans default-lazy-init (true false) "false">	Spring 定义的 JavaBean 是否需要延迟加载，这是全局设定。开发者能够在 bean 定义级别通过 lazy-init 属性覆盖在 beans 中定义的 default-lazy-init 属性值。其默认值为 false
default-dependency-check	<!ATTLIST beans default-dependency-check (none objects simple all) "none">	Spring 是否需要为 JavaBean 及 JavaBean 之间的依赖关系进行判断，这是全局设定。开发者能够在 bean 定义级别通过 dependency-check 属性覆盖在 beans 中定义的 default-dependency-check 属性值。其默认值为 none。如

(续表)

属 性	定 义	描 述
		果设为 simple, 则表明需要对 JavaBean 中使用的 Java 原型、String 进行判断; 如果设为 objects, 则表明需要对 JavaBean 之间的依赖关系进行判断; 如果设为 all, 则表明上述两方面都需要判断。其默认值为 none, 即不进行依赖关系判断
default-autowire	<code><!ATTLIST beans default-autowire (no byName byType constructor autodetect) "no"></code>	是否借助于 Spring 提供的“autowire”功能, 以注入 JavaBean 之间的引用关系, 即开发者不用显式地在 Spring 配置文件中给出 JavaBean 对其他 JavaBean 的引用。目前, Spring 支持 5 种方式实现“autowire”, 即 no、byName、byType、constructor、autodetect。其默认值为 no。详情请参考 bean 节点一节内容

在 beans 节点中包含了 3 个子元素, 即 description、import、bean。可选 description 描述其所在 Spring 配置文件的内容描述。可选 import 用于导入其他 Spring 配置文件。可选 bean 用于定义 JavaBean。其中, 借助于 import 元素使得管理 Spring 配置文件更为简单, 尤其是对于那些含有很多配置文件的 Web 应用而言很有意义。比如, 如果某 Web 应用存在多层, 而且各层使用的 Spring 配置文件分别在不同目录下。尽管通过 web.xml 中定义的 contextConfigLocation 能够分别实现对它们的引用, 但是这给 Web 应用暴露了过多的外在内容, 不利于模块化。

B.2 bean 节点

图 B-2 展示了图 B-1 中的 bean 节点的属性。通常, bean 元素用于定义 JavaBean。开发者可以借助于 JavaBean 属性, 或者构建器参数定义 JavaBean。

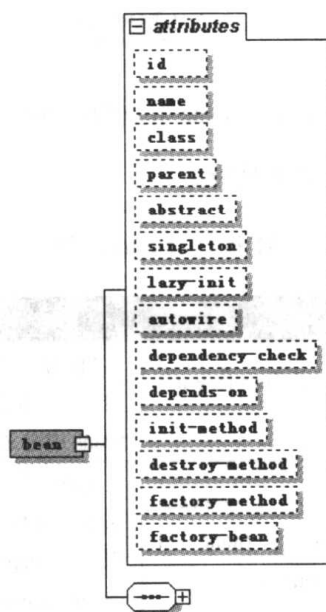


图 B-2 bean 节点的属性

其中, bean 的属性解释如 B-2 所示。

表 B-2 bean 属性

属 性	定 义	描 述
id	<!ATTLIST bean id ID #IMPLIED>	开发者可以通过 id 访问 Spring BeanFactory 中的 JavaBean
name	<!ATTLIST bean name CDATA #IMPLIED>	开发者可以通过 name 访问 Spring BeanFactory 中的 JavaBean。在某种意义上, name 是 id 的别称
class	<!ATTLIST bean class CDATA #IMPLIED>	Spring JavaBean 定义必须指定 class 属性, 即 JavaBean 类的全限定名(包名 + 类名)。如果在 JavaBean 定义中没有给出 id 或 name 属性, 则 id 的取值将会是 class 指定的类名
parent	<!ATTLIST bean parent CDATA #IMPLIED>	如果没有指定 class 属性, 则可以使用 parent。它们的含义是等效的。其中, 继承 parent 的 JavaBean 定义不仅能够继承 parent 中定义的所有内容, 还能够覆盖 parent 中定义的相关内容, 比如覆盖 JavaBean 方法、init、destroy 等内容。当然, 某些 parent 设置是不能够覆盖的, 比如依赖关系、autowire 模式、singleton 等。详情请参考 DTD 文件
abstract	<!ATTLIST bean abstract (true false) "false">	如果其值为 true, 则表明相应的 JavaBean 定义只是供 parent 使用, 其本身不能够实例化。其默认值为 false
singleton	<!ATTLIST bean singleton (true false) "true">	如果其值为 true, 则表明相应的 JavaBean 实例在全局(整个 Spring 上下文中)只有一个。如果为 false, 则表明针对不同的 getBean 调用都将返回不同的 JavaBean 实例, 即 prototype 模式的实现。一般而言, 使用单例能够满足大部分企业 Java 计算, 尤其是适合于那些多线程 JavaBean 对象。其默认值为 true
lazy-init	<!ATTLIST bean lazy-init (true false default) "default">	本书在 beans 节点一节已阐述过
autowire	<!ATTLIST bean autowire (no byName byType constructor autodetect default) "default">	<p>借助于 autowire 能够控制 JavaBean 属性的 autowire 能力, 即借助于 Spring 提供的 autowire 功能。如果开发者使用 autowire, 则其取值及相应的含义如下。</p> <ul style="list-style-type: none"> ● no: 默认值为 no, 即开发者必须使用<ref>元素显式地引用其他 JavaBean 定义。本书建议开发者不要使用 autowire 功能, 这将破坏 Spring 配置文件的可读性, 而且容易出错 ● byName: 通过 JavaBean 属性名进行 autowire 操作。比如, 如果在某 DAO 中暴露了 sessionFactory 属性, 则 Spring 将在当前 BeanFactory 中寻找到 sessionFactory JavaBean 定义 ● byType: 基于 JavaBean 的属性类型进行 autowire。但前提是, 在整个 Spring BeanFactory 中仅存在单个相应属性类型的 JavaBean 定义 ● constructor: 同 byType 类似 ● autodetect: 借助于 constructor 或 byType 进行 autowire 操作

属 性	定 义	描 述
dependency-check	<!ATTLIST bean dependency-check (none objects simple all default) "default">	本书在 beans 节点一节已阐述过
depends-on	<!ATTLIST bean depends-on CDATA #IMPLIED>	用于保证相应的 JavaBean 在其 depends-on 指定的 JavaBean 实例化之后才去实例化使用了 depends-on 的 JavaBean 本身
init-method	<!ATTLIST bean init-method CDATA #IMPLIED>	用于在实例化 JavaBean, 并设置 JavaBean 属性后, 待执行的初始化方法。比如, 如果某 JavaBean 是用于进行 JMS 目的地, 则可以借助于 init-method 指定的方法启动消息监听。请注意, 指定的方法应该是参数类型的
destroy-method	<!ATTLIST bean destroy-method CDATA #IMPLIED>	在 Spring BeanFactory 销毁时, 可以借助于 destroy-method 方法执行一些资源回收操作, 或者其他操作。比如, 关闭同 JMS 目的地的监听及连接。请注意, 指定的方法应该是参数类型的
factory-method	<!ATTLIST bean factory-method CDATA #IMPLIED>	指定用于创建 JavaBean 实例的工厂方法。这主要是考虑到一些遗留应用而提供的实用策略。在 Spring 框架中, 对象的创建最好直接通过 Spring BeanFactory 创建
factory-bean	<!ATTLIST bean factory-bean CDATA #IMPLIED>	指定用于创建 JavaBean 实例的工厂类。这主要是考虑到一些遗留应用而提供的实用策略。在 Spring 框架中, 对象的创建最好直接通过 Spring BeanFactory 创建

在 bean 元素中, 包含了其他很多子元素, 见图 B-3 所示。其中, 图 B-3 是针对图 B-1 中的 bean 节点展开的。

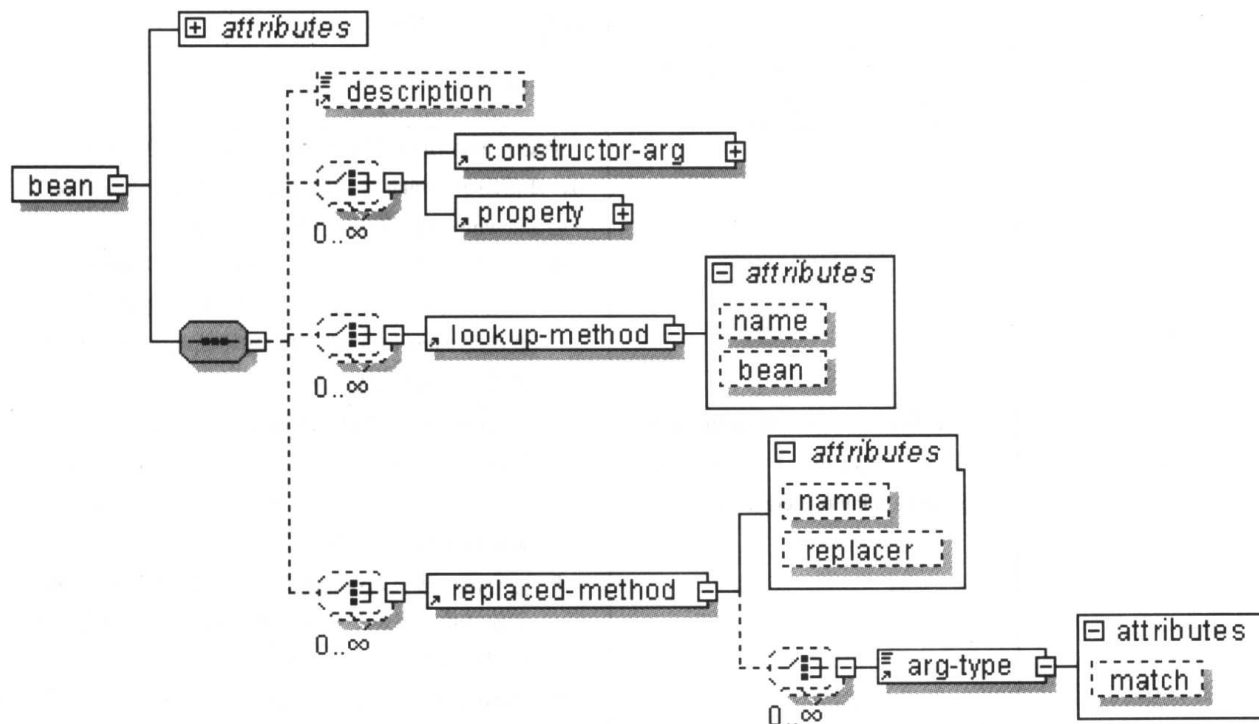


图 B-3 beans 节点的内容模型

B.3 constructor-arg 节点

其中, `constructor-arg` 节点的内容模型见图 B-4, 它是对图 B-3 中 `constructor-arg` 节点的展开。为构建 `JavaBean` 实例, 开发者可能需要为 `JavaBean` 指定构建器参数。当然, 这同 `autowire` 中的“`constructor`”取值意义等价。如果 `JavaBean` 仅存在单个构建器参数, 则直接使用即可。比如:

```
<bean name="fileHelloWorld"
      class="com.openv.spring.HelloWorld">
  <constructor-arg>
    <ref bean="fileHello"/>
  </constructor-arg>
</bean>
```

如果存在多个构建器参数, 则开发者能够通过 `index` 或者 `type` 指定构建器参数。其中, `index` 用于指定具体参数在构建器参数列表中的位置; `type` 用于指定具体参数的类型。

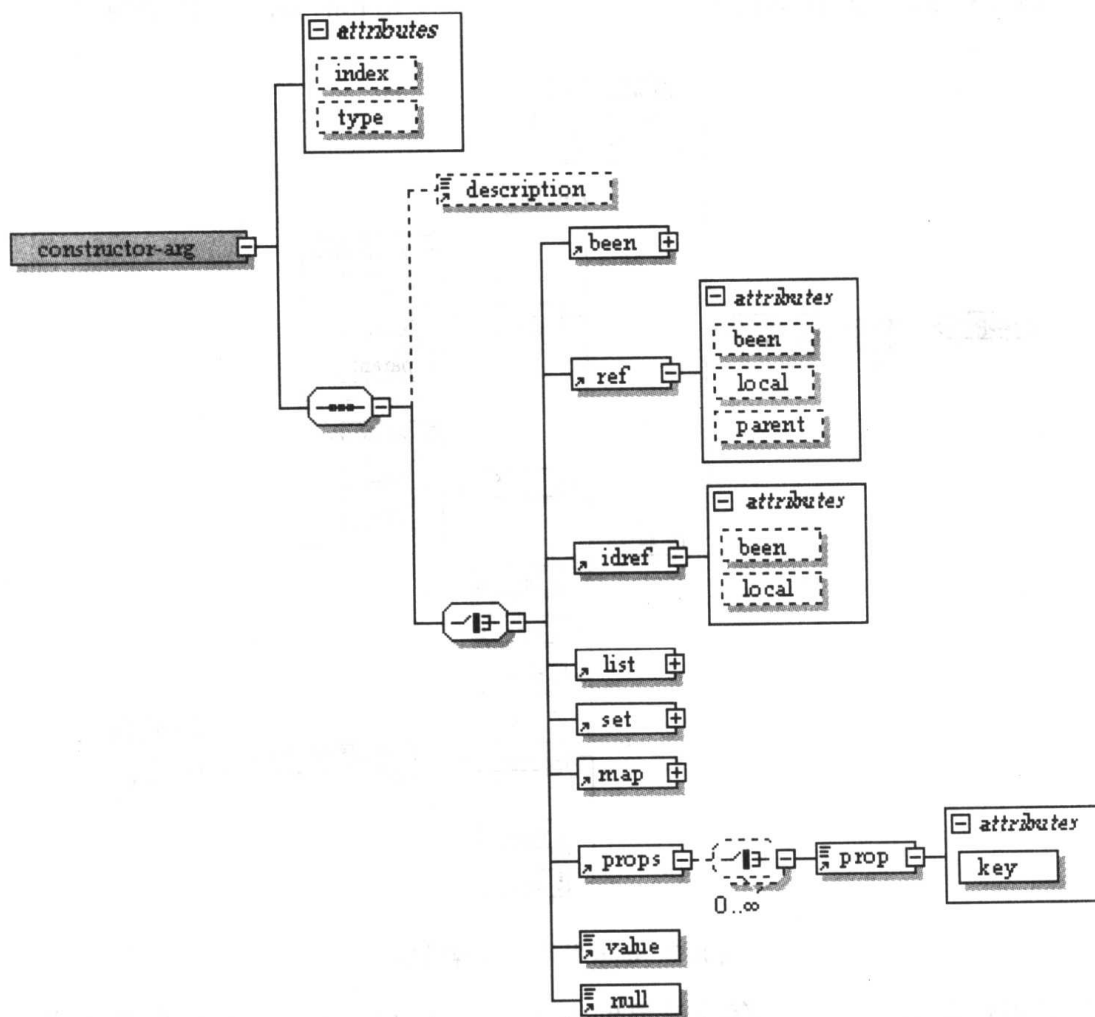


图 B-4 `constructor-arg` 节点的内容模型

其中，含有的 list、set、map 节点的内容模型分别见图 B-5、图 B-6 和图 B-7。

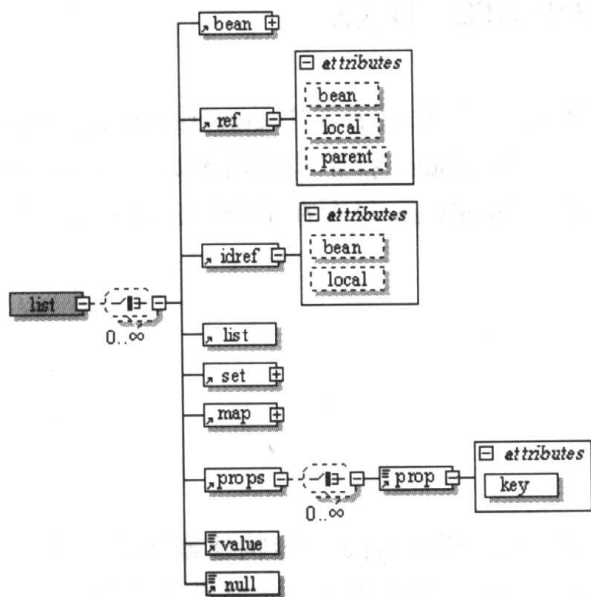


图 B-5 list 节点的内容模型

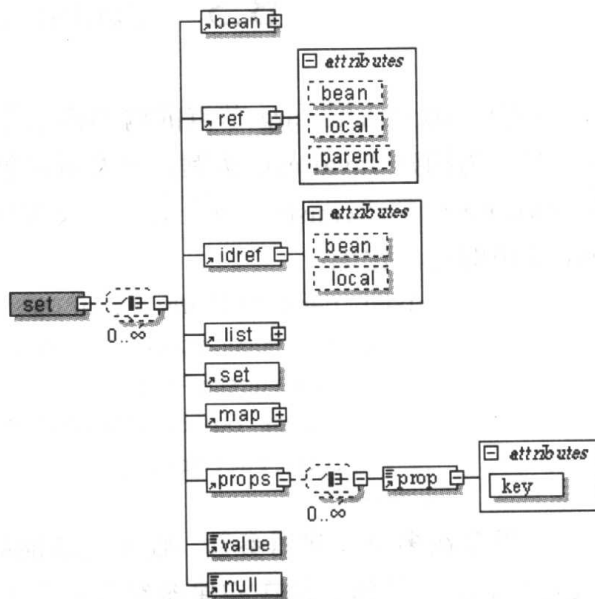


图 B-6 set 节点的内容模型

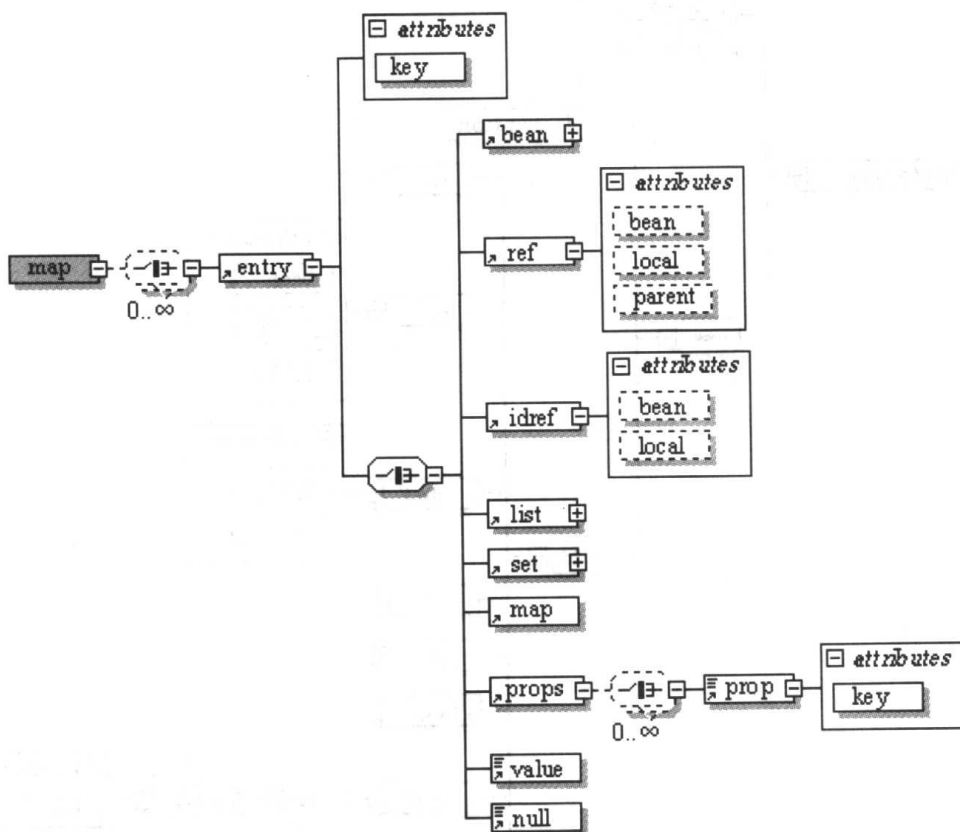


图 B-7 map 节点的内容模型

开发者可以认为，它们是存储对象的容器，这同 Java Collection 框架定义的那些 Collection 类似。更详细参数，请参考 DTD 文件。

B.4 property 节点

其中，property 节点的内容模型见图 B-8，它是对图 B-3 中 property 节点的展开。为构建 JavaBean 实例，开发者可能需要为 JavaBean 指定属性，即借助于其暴露的 setter 方法。

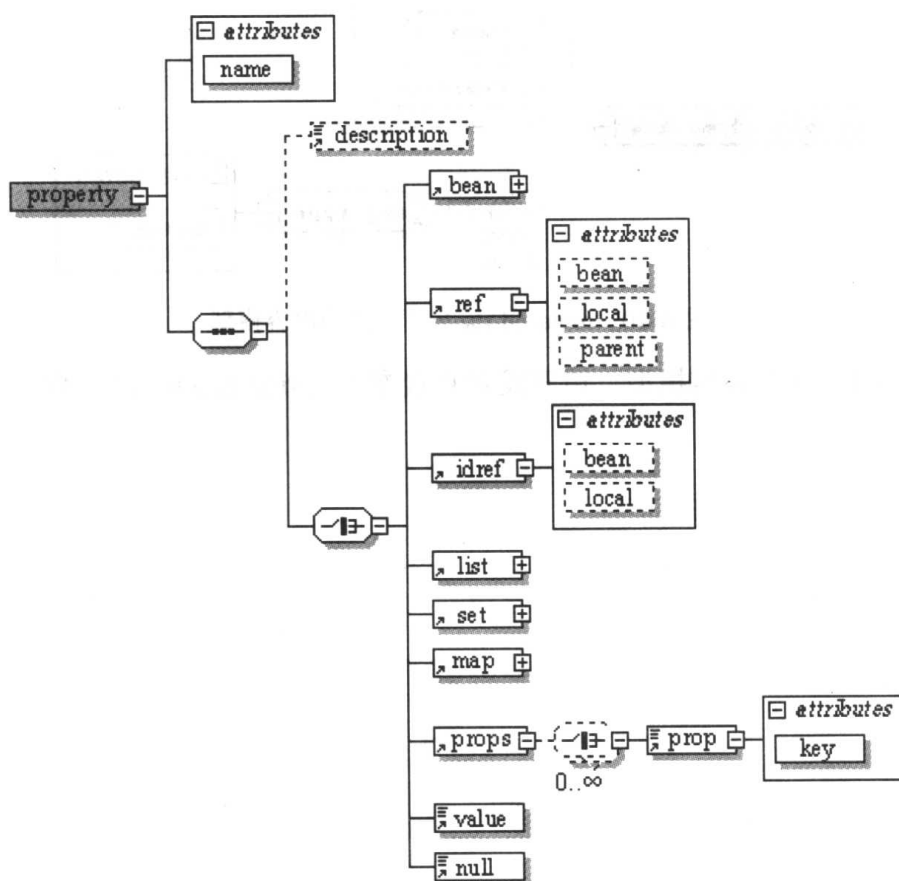


图 B-8 property 节点的内容模型

当然，其包括的 list、set、map 同 constructor-arg 节点一样。

B.5 lookup-method 节点

图 B-9 给出了 lookup-method 节点的内容模型。

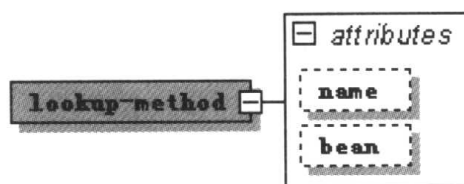


图 B-9 look-method 节点的内容模型

B.6 replaced-method 节点

图 B-10 给出了 replaced-method 节点的内容模型。

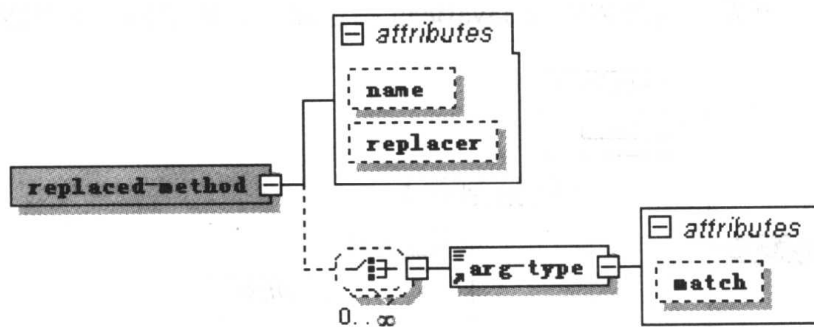


图 B-10 replaced-method 节点的内容模型

更多 Spring DTD 的内容模型，请开发者直接参考 `spring-beans.dtd` 文件。

附录 C 参 考 资 料

- <http://www.theserverside.com>

TheServerSide.com - News, Patterns, Reviews, Discussions, Articles, Books

几乎是第三方最权威的 Java 网站之一。其中汇集了 Java 领域的最新发展动态、新闻、论坛、图书样章、世界级 Java 专家访谈等精彩内容。

- <http://javaboutique.internet.com/>

Java(TM) Boutique - Free Java Applets, Games, Programming Tutorials, and Downloads - Applet Categories

定期有 Java 优秀文章，开发者可以订阅其邮件列表。

- <http://www.devx.com/Java>

DevX Java Zone

Java 优秀文章。

- <http://www-900.ibm.com/developerWorks/cn/java/index.shtml>

developerWorks Java 专区

IBM 相当精彩的网站。

- <http://www.sys-con.com/java/>

Java Developer's Journal

JDJ，很不错的杂志。开发者可以免费注册，以获得电子版的 JDJ。请注意，密码会定期更换。

- <http://www.javashelf.com/>

Java books - JavaShelf.com Your Java book store on the Web!

国外出版商最新出版和以往出版的 Java 图书介绍。基于 Web 服务实现，每天都会更新 Java 出版动态。

- <http://www.onjava.com/>

ONJava.com

著名 O'Reilly 出版商的 Java 网站。

- <http://java.sun.com/reference/blueprints/index.html>

Java BluePrints Guidelines, patterns, and code for end-to-end applications

Sun Java 权威业界实践。

- <http://www.springframework.org>

Spring - java-j2ee Application Framework

著名的 Spring 分层的、Java/J2EE 应用框架。

- <http://jakarta.apache.org/tapestry/>

Jakarta Tapestry - Welcome!

基于组件、OOP&D 的 Web 开发。

- <http://www.hibernate.org/>

Hibernate - Object-Relational Mapping and Transparent Object Persistence for Java and

SQL Databases

著名的 Hibernate 网站。

- <http://struts.apache.org/>

The Apache Struts Web Application Framework

关于 Struts 的网站。

- <http://www.waferproject.org/index.html>

Wafer - Web Application Framework Research project

分析各种主流的框架。

- <http://www.martinfowler.com/>

Martin Fowler

Martin Fowler 网站。

- <http://www.jboss.org/index.html>

JBoss Professional Open Source

著名的 JBoss 网站。

- <http://eclipse.org/aspectj/>

aspectj project

最为成熟的 AOP 实现，即 AspectJ。

- <http://aopalliance.sourceforge.net/>

AOP Alliance

AOP 联盟。

- <http://www.javaworld.com/>

JavaWorld.com

JavaWorld 网站。

- <http://www.eclipseplugincentral.com/>

Eclipse plugin resource center and marketplace for Eclipse and Plugin Ecosystem Eclipse

Plugin Central

Eclipse 插件中心。

- <http://java.sun.com/j2ee/javaxserverfaces/>

JavaServer Faces

官方网站。提供 JSF 最新资讯。

- <http://www.jsfcentral.com/>

JSF Central - Your JavaServer Faces Community - News

提供 JSF 最新资讯。

- <http://www.picocontainer.org/>

PicoContainer - Home

又一 IoC 容器实现。

- 《JBoss 管理与开发核心技术》(第三版)

由 JBoss 官方写作的这本 JBoss 图书, 绝对是对 JBoss 诠释最为优秀的一本书。

- 《The J2EE Tutorial》

建议 J2EE 初学者都能够看看这本书, 或者以这本书为切入点学习 J2EE, 它能够提高开发者的学习进度。绝对能够保证对 J2EE 理解和掌握的权威性、正确性。

<http://java.sun.com/j2ee/learning/tutorial/index.html>

- 《Spring Live》

著名的 AppFuse、Equinox 是 Matt Raible 努力的成果, 而这本书正是他写的。其中, 对于 TDD、XP 开发模式的实施, 作者给出了相当有效的办法。建议开发者能够看看这本书。

- 《Tapestry in Action》

由 Manning 于 2004 年度出版的重量级图书, Tapestry 创始人 Howard M. Lewis Ship 亲自执笔。

- 《Spring in Action》

由 Manning 于 2005 年度出版的重量级图书, 由 Craig Walls 和 Ryan Breidenbach 写作而成。

- 《Java Network Programming, 3rd Edition》

由 Oreilly 于 2004 年度出版的重量级图书, 由 Elliotte Rusty Harold 写作而成。

- 《Expert.One-on-one.J2EE.Development.Without.EJB》

由 Wiley 于 2004 年度出版的重量级图书, 由 Spring 创始人 Rod Johnson 和 Juergen Hoeller 写作而成。

- 《Hibernate in Action》

由 Manning 于 2004 年度出版的重量级图书, Hibernate 创始人 Gavin King 亲自执笔。

- <http://www.liferay.com/home/index.jsp>

提供了基于 Spring 开发的 Portal 实现。

Liferay - Home。

- <http://www.open-v.com>

本书作者维护的一个企业级 Java 网站, 通过它能够访问到本书的最新信息。

专注于企业级 Java 应用、培训以及咨询。

本
g
g
个
是
于
项
g
本
本

未
板

图

精

(

S

I

中

编

发

凡

耳

@

后 记

开发者已经同本书奋战了一段时间了。该是总结的时候了。正如开发者在序中期待的一样，全书都是围绕 Spring 展开的，对于不同内容的介绍深度有所区分。总体而言，本书应该同开发者达成了如下共识。

- 借助于 Spring IoC 和 Spring AOP 开发 Web 应用，确实是轻量级的架构。虽然这种组合没有 J2EE 容器的功能丰富（总体上而言），但是这种无缝集成确实能够解决实际应用中存在的很多问题。开发者在实际项目中，可以针对实际情况而使用 Spring 框架提供的部分内容，比如 Spring IoC。
- 借助于服务抽象，Spring 简化了 J2EE API 的使用。比如，借助于 Spring API 获得 JNDI 对象变得简单了、借助于 Spring API 获得 RDBMS 数据源变得简单了。所有的组件（包括 JavaBean），都是可动态配置的。
- Spring 简化了开发者对具体 Java/J2EE 平台技术的选型。比如，Spring 对 Hibernate 的、一流的集成，使得开发者确信：Hibernate 是不错的 O/R Mapping 技术，事实也是如此。
- 用于 Spring 的 Acegi 安全框架太优秀了！Acegi 本身就是 Spring 使能应用，因此这再次说明 Spring 本身的灵活性，即基于 Spring 构建 Acegi 或其他框架，并进一步完善 Spring 的功能。
- Spring 是非常实用的架构级框架。

当然，作为 Spring 框架的使用者，想仅仅从这本书获得所有答案太不现实了。如果这本书能够激发您学习 Spring 的兴趣、带领您进入新的视野，那么从这个角度出发，这本书就是成功的。在此，我将介绍其他几份重要的技术资料，以利于对 Spring 的使用。

- Rod Johnson 著（有些是合著的）的 3 本书。其中，包括《Expert One-on-One J2EE Design and Development》、《Expert One-on-one J2EE Development Without EJB》（它们分别已于 2002、2004 年出版发行）、《Professional Java Development with the Spring Framework》（本书写作之际，还未出版）。
- Craig Walls 和 Ryan Breidenbach 著的《Spring in Action》（已于 2005 年年初出版）。开发者通过 Craig Walls 的 Blog（<http://jroller.com/page/habuma/Weblog>）能够获得这本书的详情及 Spring 相关知识的深入介绍和研究。注意，Craig Walls 同时还是《XDoclet in Action》一书的合作者。
- Matt Raible 著的《Spring Live》。通过 <http://www.sourcebeat.com> 能够进一步了解它。如果您是 Spring 的初学者，则在阅读本书的同时，还请看看《Spring Live》。Matt Raible 将持续更新这本书，以反映最新的 Spring 技术。
- Spring 英文网站：<http://www.springframework.org>。

作者确信，Spring 会越来越实用、流行。当然，如果本书能够赢得读者的赞赏，则本书的作者也希望能够有再版，它将会更优秀。

另外，由于本人的技术和写作能力有限，而且 Spring 的发展也很快，这些使得作者对 Spring 的部分理解会造成误解、甚至错误，那么请开发者原谅，并请同作者联系。

与此同时，<http://www.open-v.com> 还提供了大量的 Spring 资料（当然，对 Acegi 的介绍也不能够缺少。）介绍。

前言
目录
目录

	第一部分	Spring 架构分析第 1
章	Spring 启程	
	1.1	背景知识
	1.2	运行 Spring 实例应用
1	1.2.1	实例 1: example
2	1.2.2	实例 2: example
	1.2.3	实例 3: example 3
4	1.2.4	实例 4: example
	1.3	Spring I/O 实用类
	1.4	小结
	第 2 章	安装和构建 Spring
	2.1	获得二进制文件
	2.2	基于源代码构建 Spring
码	2.2.1	基于 CVS 访问以获得源代
	2.2.2	构建 Spring 框架
	2.2.3	重要 Ant 任务
	2.3	安装 Spring
	2.4	小结
	第 3 章	控制反转 (Spring IoC)
	3.1	IoC 背景知识
	3.2	Spring IoC
	3.2.1	BeanFactory
o n t e x t	3.2.2	ApplicationContext
	3.3	IoC 其他内容
	3.3.1	发布并监听事件
性编辑器	3.3.2	自定义 JavaBean 属
	3.4	小结
	第 4 章	面向方面编程 (Spring AOP)

	4.1	AOP及Spring AOP
P 背景知识	4.2	Spring AOP 装备
	4.2.1	Before 装备
	4.2.2	After 装备
	4.2.3	Throws 装备
	4.2.4	Around 装备
	4.3	ProxyFactoryBean
	4.4	对象池
	4.5	小结
第5章	深入Spring 架构	
	5.1	架构概述
	5.2	Spring 具体构件
	5.2.1	Spring 上下文
	5.2.2	Spring Web
	5.2.3	Spring 数据访问对象
(DAO)	5.2.4	Spring ORM
	5.2.5	Spring Web M
VC 框架	5.3	综合实例分析
	5.3.1	实例概述
	5.3.2	安装和配置 example
II	5.3.3	架构分析
	5.4	小结
第二部分	Spring 应用开发	第6章
命名服务——JNDI	6.1	背景
支持	6.2	Spring 对JNDI 提供的
	6.2.1	JndiObjectFactoryBean
	6.2.2	JndiObjectTargetSource
	6.2.3	JndiTemplate
	6.2.4	JndiCallback

	6.3	小结
第7章	事务服务——JTA	
	7.1	背景
	7.2	Spring对事务管理提供的
支持		
	7.2.1	PlatformTransactionManager
	7.2.2	声明式事务
	7.2.3	编程式事务
	7.3	小结
第8章	消息服务——JMS	
	8.1	背景
	8.2	Spring对JMS提供的支
持		
	8.2.1	JmsTemplate
	8.2.2	事务管理
	8.3	小结
第9章	邮件服务——JavaMail	
	9.1	背景
	9.2	Spring对JavaMail
I提供的支持		
	9.2.1	使用CosMailSenderImpl
	9.2.2	使用JavaMailSenderImpl
	9.3	小结
第10章	企业Bean服务——EJB	
	10.1	背景
	10.2	Spring对EJB提供的
支持		
	10.2.1	开发EJB
	10.2.2	访问EJB
	10.3	小结
第11章	持久化服务——DAO、JDBC、O	
RM		
	11.1	背景
	11.2	Spring对DAO提供的

支持	1 1 . 3	S p r i n g对J D B C提供的支持
支持	1 1 . 3 . 1	J d b c T e m p l a t e
	1 1 . 3 . 2	D a t a S o u r c e T r a n s a c t i o n M a n a g e r
	1 1 . 3 . 3	连接数据库的方式
	1 1 . 3 . 4	将J D B C操作建模为J a v a对象
	1 1 . 4	S p r i n g对O R M提供的支持
	1 1 . 4 . 1	H i b e r n a t e介绍
	1 1 . 4 . 2	H i b e r n a t e集成
支持	1 1 . 5	小结
第 1 2 章		任务调度服务——Q u a r t z、T i m e r
	1 2 . 1	背景
	1 2 . 2	S p r i n g对Q u a r t z提供的支持
	1 2 . 2 . 1	Q u a r t z J o b B e a n和J o b D e t a i l B e a n的使用
	1 2 . 2 . 2	M e t h o d I n v o k i n g J o b D e t a i l F a c t o r y B e a n的使用
	1 2 . 3	S p r i n g对T i m e r提供的支持
	1 2 . 3 . 1	S c h e d u l e d T i m e r T a s k的使用
	1 2 . 3 . 2	M e t h o d I n o v k i n g T i m e r T a s k F a c t o r y B e a n的使用
	1 2 . 4	小结
第 1 3 章		远程服务
	1 3 . 1	背景
	1 3 . 2	S p r i n g对远程服务提供的支持

	1 3 . 2 . 1	R M I 使能服务
	1 3 . 2 . 2	H e s s i a n 使能服务
	1 3 . 2 . 3	B u r l a p 使能服务
	1 3 . 2 . 4	H T T P I n v o k e
r 使能服务		
	1 3 . 3	S p r i n g 对 W e b 服务提
供的支持		
	1 3 . 4	小结
第三部分		S p r i n g 高级主题第 1 4
章 视图技术集成		
	1 4 . 1	S p r i n g W e b M V
C		
	1 4 . 1 . 1	配置 D i s p a t c h e
r S e r v l e t		
	1 4 . 1 . 2	开发及配置 C o n t r o
l l e r		
	1 4 . 1 . 3	配置 V i e w R e s o
l v e r		
	1 4 . 1 . 4	配置 H a n d l e r M a
p p i n g		
	1 4 . 2	S t r u t s
	1 4 . 2 . 1	S p r i n g J P e
t S t o r e 的 A p p l i c a t i o n C o n t e x t 集		
成方式		
	1 4 . 2 . 2	S p r i n g 提供的集成
方式		
	1 4 . 3	T a p e s t r y
	1 4 . 4	J S F
	1 4 . 5	J S P 和 J S T L
	1 4 . 6	V e l o c i t y 和 F r e e
M a r k e r		
	1 4 . 7	X S L T
	1 4 . 8	T i l e s
	1 4 . 9	J a s p e r R e p o r t s
	1 4 . 1 0	文档视图
	1 4 . 1 1	小结
第 1 5 章		T a p e s t r y 集成

	1 5 . 1	T a p e s t r y 介绍
	1 5 . 2	P a g e 和组件模板
	1 5 . 3	创建 T a p e s t r y 组件
	1 5 . 4	T a p e s t r y 校验子系统
	1 5 . 5	管理服务器端状态
	1 5 . 6	配置 T a p e s t r y 应用
	1 5 . 7	与 S p r i n g 集成
	1 5 . 8	小结
第 1 6 章	J S F 集成	
	1 6 . 1	W e b 前端开发的趋势
	1 6 . 2	J S F 介绍
	1 6 . 3	S p r i n g 和 J S F - S p r i
n g 提供的 J S F 集成		
	1 6 . 4	e x a m p l e 2 9 实例研究
	1 6 . 4 . 1	部署及使用
	1 6 . 4 . 2	开发过程
	1 6 . 4 . 3	S p r i n g 提供的 J S
F 集成能力		
	1 6 . 4 . 4	J S F - S p r i n g 项
目提供的 J S F 集成能力		
	1 6 . 5	小结
第 1 7 章	用于 S p r i n g 的 A c e g i 安全框	
架		
	1 7 . 1	A c e g i 介绍
	1 7 . 2	A c e g i 架构及使用
	1 7 . 2 . 1	构建 c o n t a c t s 应
用		
	1 7 . 2 . 2	A c e g i 架构综述
	1 7 . 2 . 3	W e b 资源的认证
	1 7 . 2 . 4	W e b 资源的授权
	1 7 . 2 . 5	配置 A c e g i S e
r v l e t 过滤器		
	1 7 . 2 . 6	方法级的认证和授权
	1 7 . 3	其他内容
	1 7 . 3 . 1	实现密码的加密处理
	1 7 . 3 . 2	缓存用户信息
	1 7 . 4	小结

	附录A	实例代码安装
	A.1	代码说明
	A.2	钟情JBoss
	A.3	工具下载与安装
	A.3.1	Spring IDE
	A.3.2	Tapestry Spi
ndle		A.3.3 JBoss IDE
		A.3.4 Hibernate Sy
nchronzier		A.4 代码使用
	附录B	spring-beans.dtd的内
容模型		
	B.1	beans节点
	B.2	bean节点
	B.3	constructor-ar
g节点		
	B.4	property节点
	B.5	lookup-method节
点		
	B.6	replaced-metho
d节点		
	附录C	参考资料
	后记	