

Spring live中文版

Matt Raible

白汉奇[译]

Spring live中文版

Matt Raible

白汉奇[译]

版权 © 2004 SourceBeat, LLC

Many designations used by organizations to distinguish their products are claimed as trademarks. These trademarked names may appear in this book. We use the names in an editorial fashion only with no intention of infringing on the trademark; therefore you will not see the use of a trademark symbol with every occurrence of the trademarked name.

As every precaution has been taken in writing this book, the author and publisher will in no way be held liable for any loss or damages resulting from the use of information contained in this book.

致辞

谨以此书献给Abbie 和Jack---这个城镇上最酷的小精灵。

目录

关于作者	xii
致谢	xiii
简介	xiv
翻译札记	xvi
缘起	xvi
在NetBeans中配置Equinox	xvi
1. Spring 简介	26
Spring 诞生记	26
关于Spring	26
为什么每个人都钟情于它	26
对Spring的一些常见的批评	27
Spring 的工作原理	27
Spring如何简化J2EE开发	28
接口编程	29
简单的测试	29
降低耦合: Factory Pattern vs Spring	29
本章小结	31
2. Spring快速入门教程	33
概述	33
下载Struts和Spring	33
下载Struts和Spring	34
创建项目目录和Ant Build文件	35
Tomcat和Ant	36
为持久层编写单元测试	39
配置Hibernate和Spring	41
Equinox中Spring是如何配置的	42
用Hibernate实现UserDAO	44
进行单元测试, 用DAO验证CRUD操作	45
创建Manager, 声明事务处理	46
对Struts Action进行单元测试	50
为web层创建Action和Model(DynaActionForm)	51
运行单元测试, 验证Action的CRUD操作	56
填充JSP文件, 这样可以通过浏览器来进行CRUD操作	57
通过浏览器验证JSP的功能	58
用Commons Validator添加验证	60
在struts-config.xml中添加ValidatorPlugin	60
创建validation.xml, 指定lastName为必填字段	60
把 DynaActionForm 改为 DynaValidatorForm	61
为save()方法设置验证(validation)	61
本章小结	62
3. BeanFactory及其工作原理	63
关于BeanFactory	63
BeanFactory中一个bean的生命周期	63
反转控制	64
暴露bean定义	65
配置域属性和依赖关系	68
预先初始化bean	69
就绪状态	73
销毁bean	74
ApplicationContext: 与bean交互	74
抓取context	74

单元测试和导入context的一些技巧	75
国际化和MessageSource	76
事件的发布和订阅	77
深入探讨MyUsers的ApplicationContext.xml	77
本章小结	78
4. Spring MVC 框架	79
概述	79
Spring Controller单元测试	81
配置DispatcherServlet和ContextLoaderListener	81
修改web.xml, 使用Spring的DispatchServlet	82
为UserController编写单元测试	84
创建UserController, 配置action-servlet.xml	85
创建userList.jsp页面显示用户列表	87
为用户FormController创建单元测试	90
创建UserFormController, 并在action-servlet.xml中配置它	92
创建userForm.jsp页面, 以便编辑用户资料	97
为Spring配置Commons Validator	99
SimpleFormController: 方法生命周期回顾	102
Spring的JSP标签	104
本章小结	105
5. 高级的MVC框架——使用模板, 验证, 异常处理, 文件上传	106
SiteMesh模板技术	106
安装和配置	107
Tiles模板技术	111
安装和配置	111
Spring验证方法	117
使用Commons Validator	119
XDoclet	120
一系列的验证器	121
在业务委派中进行验证	121
Spring未来的声明式验证框架	122
控制器中的异常处理	123
上传文件	126
拦截请求	131
发送邮件	133
本章小结	135
6. View的多种选择	136
View(视图)和ViewResolver(视图解析器)	136
利用jWebUnit测试View	138
JSP	140
View Resolver配置	140
JSTL	142
Tiles	143
Velocity	144
在MyUsers中使用Velocity	144
SiteMesh和Velocity	146
创建Velocity模板	148
部署和测试	150
Velocity小结	151
FreeMarker	151
View Resolver配置	151
SiteMesh和FreeMarker	152
创建FreeMarker模板	155
部署和测试	158

XSLT	158
创建View类	159
部署和测试	161
Excel	162
创建View类	162
部署和测试	163
PDF	164
创建View类	164
部署和测试	165
本章小结	165
7. 持久性策略	166
概述	166
准备练习	167
Hibernate	168
依赖包	168
配置	169
缓存	174
延迟导入(lazy-load)依赖对象	176
社区和支持	177
iBATIS	177
依赖包	179
配置	179
缓存	183
获取生成的主键	184
社区与支持	186
Spring JDBC	186
将ResultSet映射到对象	186
执行更新查询	188
获取生成的主键	189
依赖包	189
配置	189
社区和支持	192
JDO	192
JDO和主键	193
依赖包	193
配置	194
缓存	201
社区和支持	201
OJB	202
依赖包	202
配置	202
缓存	206
社区和支持	206
本章小结	206
8. 测试Spring应用程序	207
概述	207
JUnit	208
模拟测试	209
集成测试	209
测试数据库层	210
DBUnit	213
数据库切换	215
覆写Bean便于测试	216
仅加载一次Context	220

测试Service层	221
模拟DAO对象	221
测试Web层	227
测试Controller	227
测试View	241
本章小结	251
9. AOP编程	252
概述	252
入门	252
日志示例	253
定义与概念	254
Pioncuts(切入点)	255
Weaving策略	256
常用的代理Bean	258
AOP实战	259
事务	259
中间层的缓存	260
事件通知	265
本章小结	270
10. 事务处理	271
概述	271
J2EE事务管理	274
用Spring管理事务	275
事务管理器概念	275
准备练习	276
修改UserManager以出现回滚	278
编程式事务	280
声明式事务	284
Spring事务管理器	293
DataSourceTransactionManager	293
HibernateTransactionManager	293
JdoTransactionManager	294
JtaTransactionManager	294
PersistenceBrokerTransactionManager	294
本章小结	294
11. Web框架集成	296
概述	296
章节练习	296
在web应用程序中集成Spring	296
JavaServer Faces	298
集成Spring与JSF	300
View的选择	302
JSF和Spring的CRUD样例	302
Struts	302
集成Struts与Spring	304
视图的选择	308
Struts与Spring的CRUD实例	308
Tapestry	308
Tapestry与Spring集成	310
View的选择	312
Tapestry与Spring的CRUD实例	312
WebWork	312
WebWork与Spring集成	314
View的选择	317

WebWork与Spring的CRUD实例 318

Web框架对比 318

 特性对比 320

锦囊妙计 323

本章小结 323

插图清单

1.	xvii
2.	xviii
3.	xix
4.	xx
5.	xxi
6.	xxii
7.	xxiii
8.	xxiv
9.	xxv
1.1. Spring的7个模块	28
1.2. 使用Spring的MyUsers应用程序框架	28
1.3. 一个典型的J2EE应用程序	30
2.1. MyUsers应用程序流程	33
2.2. MyUsers应用程序目录结构	35
2.3. 运行ant list命令的结果	38
2.4. Equinox欢迎页面	38
2.5. HSQL Database Manager	42
2.6. 运行ant test -Dtestcase=UserDAO命令的结果	46
2.7. 运行ant test -Dtestcase=UserManager命令的结果	50
2.8. 运行ant test -Dtestcase=UserAction命令的输出结果	56
2.9. 运行ant deploy reload命令的结果	58
2.10. 运行ant deploy命令的结果	62
3.1. BeanFactory中一个bean的生命周期	64
3.2. 运行ant test -Dtestcase=UserDAO的结果	72
4.1. 运行ant test -Dtestcase=UserController测试的结果	87
4.2. 运行ant populate测试的结果	90
4.3. 运行ant test -Dtestcase=UserForm测试的结果	97
4.4. 启用JavaScript运行ant deploy reload测试的结果	101
4.5. 禁用JavaScript运行ant deploy测试的结果	102
4.6. GET请求生命周期	103
4.7. POST请求生命周期	104
5.1. SiteMesh流程	106
5.2. 未经装饰的“Welcome to MyUsers”页面	107
5.3. 经过修饰的“Welcome to MyUsers”页面	111
5.4. 运行ant test -Dtestcase=UserManager命令的结果	122
5.5. 错误页面提示用户不存在	126
5.6. File Upload页面	130
5.7. 成功文件上传页面	130
5.8. 成功的邮件消息	135
6.1. 目录树样例	141
6.2. 使用此标签库的运行结果	143
6.3. Velocity模板目录结构	148
6.4. FreeMarker Templates目录结构	155
6.5. FreeMarker测试结果	158
6.6. View类测试结果	162
6.7. Excel View测试结果	163
6.8. Excel中显示的结果	164
6.9. PDF中显示的结果	165
7.1. 共用异常的继承关系	167
7.2. 运行ant test -Dtestcase=UserDAO的测试结果	173
7.3. 运行ant test -Dtestcase=UserDAO的测试结果	176

7.4. 运行ant clean enhance的结果	196
7.5. 运行ant test -Dtestcase=UserDAO的结果	199
7.6. 构建成功, 运行测试ant test -Dtestcase=UserDAO的结果	201
8.1. 测试先行开发模型	207
8.2. 运行OJB DAO时的警告信息	217
8.3. 运行ant test -Dtestcase=UserManagerJM测试的输出结果	226
8.4. 运行测试ant test -Dtestcase=UserControllerEM的输出结果	229
8.5. 非模拟测试的时间消耗	230
8.6. 运行测试ant test -Dtestcase=UserControllerJM的输出结果	231
8.7. 运行测试ant test -Dtestcase=FileUploadController的输出结果	241
8.8. ant test-canoos测试的结果	250
9.1. 运行ant test -Dtestcase=UserManagerTest测试的结果	254
9.2. 运行ant test -Dtestcase=UserManagerTest测试的结果	261
9.3. 运行ant test -Dtestcase=UserManagerTest测试的结果	263
9.4. 运行ant test -Dtestcase=UserManagerTest测试的结果	265
9.5. 运行ant test -Dtestcase=UserManagerTest测试的结果	269
10.1. 股票买卖事务样例	271
10.2. PlatFormTransactionManager接口	275
10.3. Spring框架事务基础结构	276
10.4. UserManagerTest测试结果	280
10.5. 失败事务的数据视图	280
10.6. UserManagerTest的日志	287
10.7. Hibernate不支持Repeatable Read	287
11.1. JSF与Spring集成	298
11.2. JSF生命周期	299
11.3. Struts与Spring集成	303
11.4. Struts生命周期	304
11.5. Tapestry与Spring集成	308
11.6. Tapestry生命周期	310
11.7. WebWork与Spring集成	313
11.8. WebWork的生命周期	314

表格清单

3.1. 由spring-beans.dtd文件定义的bean定义属性	67
3.2. Spring事件	77
4.1. 内置的Property Editor ³	95
6.1. View Resolvers	137
6.2. 额外的TemplateViewResolver属性	145
6.3. SiteMesh Velocity Context对象	146
6.4. SiteMesh FreeMarker Context对象	152
7.1. JDO Identity Types	193
9.1. AOP定义	255
10.1. 隔离层及读取错误	274
11.1. 常用Struts Action	304
11.2. 每种框架的正反两面	319
11.3. web框架特性对比	321

关于作者

Matt Raible是一位J2EE顾问和开发人员，他生活在科罗拉多州丹佛市。他在蒙大纳州偏远地区的一个小木屋中长大，他喜欢蒙大纳和那里可口的越桔，并且尽可能一年在那里逗留几次。

在Netcape 1.0问世之前，Matt就一直从事网站开发。90年代还在大学时，他就写过大量的HTML，JavaScript，CSS代码。当他毕业，获得俄语，国际贸易与金融学位时，他才发现他实际上喜欢计算机更多一些，后来他就成了一名顾问。

自1999年来，Matt一直从事java web应用开发，并且是Raible Designs, Inc.的总经理。他本人也是J2EE 5.0专家组的成员之一，希望协助他们来简化J2EE开发。Matt并不只是只是口头上说说简化的技术，而是把他们以行动付诸到他的开源项目AppFuse [<http://>](一个应用程序入门工具箱)中。Matt积极参与了多个开源项目，并且常常在他的Blog网站www.raibledesigns.com [<http://www.raibledesigns.com>]上写下自己的经验。

致谢

我要感谢我的妻子，Jule，她是一位贤妻良母。不得不佩服你能够同时照顾好Abbie和Jack。感谢你一直料理家务，并且在我写书期间生下了Jack。Abbie和Jack，谢谢你们提醒我，和你们一起玩乐相比，工作并不是那么重要。你们的笑容和格格的笑声会充满我生活的每一天。

我要感谢我的父母，Joe和Barb。谢谢你们给我的爱，我从我的父亲那里中学到了大量的计算机知识，爸爸，谢谢你，你是我的楷模和最伟大的朋友。我的妹妹，Kalin，我要感激你在从家到公交站的途中不厌其烦的听我讲故事，难以置信，一个孩童般讲故事能使我变成了一名作家。

感谢Rod Johnson为J2EE开发人员提供了如此优秀的框架，使得J2EE应用开发变得更加简单。Juergen，Colin，Keith，还有Spring框架团队的其他人员——一直在跟进编码。感谢你们所花的精力，积极的参与和用户支持。开源是软件开发的一条光明大道，你们是最优秀的一支团队。

谢谢SourceBeat的奠基者，Matt Filios和James Goodwill，为我提供了写作的机会，并且是一种前所未有的途径。SourceBeat是一种不错的模型，读者能真正从中受益。Dion，我要感谢你花大量的时间整理SpringLive的代码，并确保所有的代码能够运行。你已经是一名很棒的技术编辑，Amy，感谢你将我所有的"we's"修改成"you's"，使语句看起来更加通顺。你能制定一本技术书的流程，很了不起啊。

简介

本书写给那些熟悉web框架的Java开发人员，主要目的是让开发人员快速熟悉Spring。本书强调代码演示，对于接触一门新技术来说，代码示例比理论有用得多。

本书包括一个有用的样例程序，它使用Spring和Hibernate管理持久层和中间层。这个应用程序使用了一个我所有开发的简单的入门web程序----Equinox。Equinox实际上仅仅是一个Ant build文件，一个目录结构和开发基于Spring的J2EE应用程序的所需要的jar文件。

在第2章，Equinox用来开发一个简单的应用程序，叫MyUsers，执行数据表的CRUD操作。JUnit,Struts,Hibernate和Spring用来开发MyUsers，HSQL和Tomcat作为部署平台。到第4章，将采用Spring MVC重构web层。各章节中所有开发的代码，都是采用测试驱动的方法完成的。

第1章：Spring 简介覆盖了Spring的基础，来历，以及为何它能得到如此多的报道和热评。此章中与传统解决依赖性的方法作了比较(使用Factory模式创建接口来实现)，而Spring是如何在XML中解决这些问题。此章还简短的说明了Spring如何简化Hibernate API。

第2章：Spring快速入门教程介绍了如何写一个简单的Spring web应用程序，它使用了Struts MVC Framework作为前端，Spring作中间层，Hibernate作后端。在第4章，这个程序将用Spring MVC 框架进行重构。

第3章：BeanFactory及其工作原理。BeanFactory是Spring核心，所以必须弄清它是如何工作的。这一章解释了如何书写bean定义，它们的属性，依赖关系及自动绑定。也解释了隐藏在单态bean与原形相比的背后的逻辑关系。然后剖析控制反转(IOC)，它的工作原理，以及它能带来哪些便利。这一章也诠释了BeanFactory中一个bean的生命周期来演示它的工作原理。这一章还分析了在第2章MyUsers中创建的applicationContext.xml文件。

第4章：Spring MVC框架描述了Spring MVC框架的许多特性。这一章展示了如何用Spring MVC替换Strutsweb层。内容包括DispatcherServlet，不同的Controller，Handler Mappings，View Resolvers，Validation和Internationalization.还涉及了Spring JSP 标签库。

第5章：高级的MVC 框架----使用模板，验证，异常处理，文件上传。这一章涉及了web框架的一些高级主题，特别是验证和页面布局。演示如何用Tiles和SiteMesh来装饰一个页面，还解释Spring横加如何处理验证，并通过实例演示在web业务层中的运用。最后还解释了在控制器处理异常的一种策略，以及如何上传文件和发送邮件。

第6章：视图的多种选择涉及Spring MVC架构中的view的多种选择。截止撰稿时，可供选择的有JSP，Velocity，FreeMarker，XSLT，PDF和Excel。这一章为各种Spring所支持的视图提供了参考，还简短的说明各种视图如何运作，并对每种视图分别构建一个页面进行了对比。另外还集中介绍了每种视图的国际化。

第7章：持久性策略: Hibernate，iBATIS，JDBC，JDO和OJB。Hibernate很快成为java应用程序持久层的一种流行的选择，但有时它不适合.如果你有一个已经存在的数据库模型，或者预先写好的SQL的语句，有时使用JDBC或是iBATIS(支持XML文件形式的SQL语句)更适合一些。这一章对MyUsers进行重构，使它同时支持JDBC和iBATIS，作为持久层框架的可选方案。这一章也用JDO和OJB实现了UserDAO以展示Spring能很好的支持这些框架。

第8章：测试Spring应用程序说明了如何运用测试驱动的方法创建质量，经过完好测试，基于Spring的应用程序。你将到如何用诸如EasyMock，jMock和DBUnit，测试你的组件。对于控制器，你将学到如何用Cactus进行容器内测试，用Spring Mock进行容器外测试。最后，你将学习如何用jWebUnit和Canoo的WebTest 测试web 接口。

第9章：AOP编程。AOP编程吸收了大量近年来来自java社区的灵感。什么是AOP,如何在你的应用程序运用AOP? 这一章将讲述AOP的基础，并且给出几个有用的实例，演示AOP能给你带来点什么。

翻译札记

缘起

去年(现在应该叫2004年)末在www.theserverside.com闲逛时,发现了公开的本书第2章(SpringQuickStart),觉得不错,并决定把它翻译出来,供正在学习Spring的朋友参阅。我一直不太喜欢看译作,因为大多数译作,没有把握好原作者的意图,以至看译本比看原著还吃力。每每买翻译作品,均有上当受骗的感觉,以至发誓以后一定要买原著。直到翻译样章(第2章)时,才发现翻译的艰辛,很多念起来很通顺,却无法翻译通顺。还好,我没有拿翻译来赚钱,不然已经是一身的臭鸡蛋。

如何阅读这本书?如果你懂英文的话,建议看英文原著,你可以从<http://www.itpub.net>,<http://www.matrix.org.cn>的论坛中找到原版本的下载链接。如果你想为本书作点什么,你可以对照中英来看,把本书的翻译错误,遗漏,或者是更确切的翻译意见通过email寄给我,在下次发布新的章节,你看到的将会是一个经过修正的版本。

下一章什么时候发布?确切的说没有一个时间表,我也无法确定我能坚持翻译完全书。我翻译的这几个部分都是利用节假日休息时间或者换工作的间隙完成的。能不能完成完成翻译,要视以后的情况而定。

和原来的第2章不一样(采用OpenOffice.org编辑, <http://www.openoffice.org>),现在全书翻译采用了Docbook(<http://docbook.sf.net>)编写。使用原始Docbook XSL转换,样式没有经过任何调整,所以页面看起来比较"朴素",希望你能见谅。整理第2章时,还发现我所拿到的这个9章的版本和www.theserverside.com上公开的版本有细微的差别,阅读时请以后者为准。

另外,从第3章起,不再为每一章提供单独的pdf文件。每完成一部分新的内容,我会及时的追加到原来的文件,并把反馈意见及时的反映出来。这样你得到的是一个不断更新的"Live Book"。

祝阅读愉快!

Hantsy(白汉奇)

Email:<hantsy@tom.com>

Blog:<http://hantsy.cublog.cn>

在NetBeans中配置Equinox

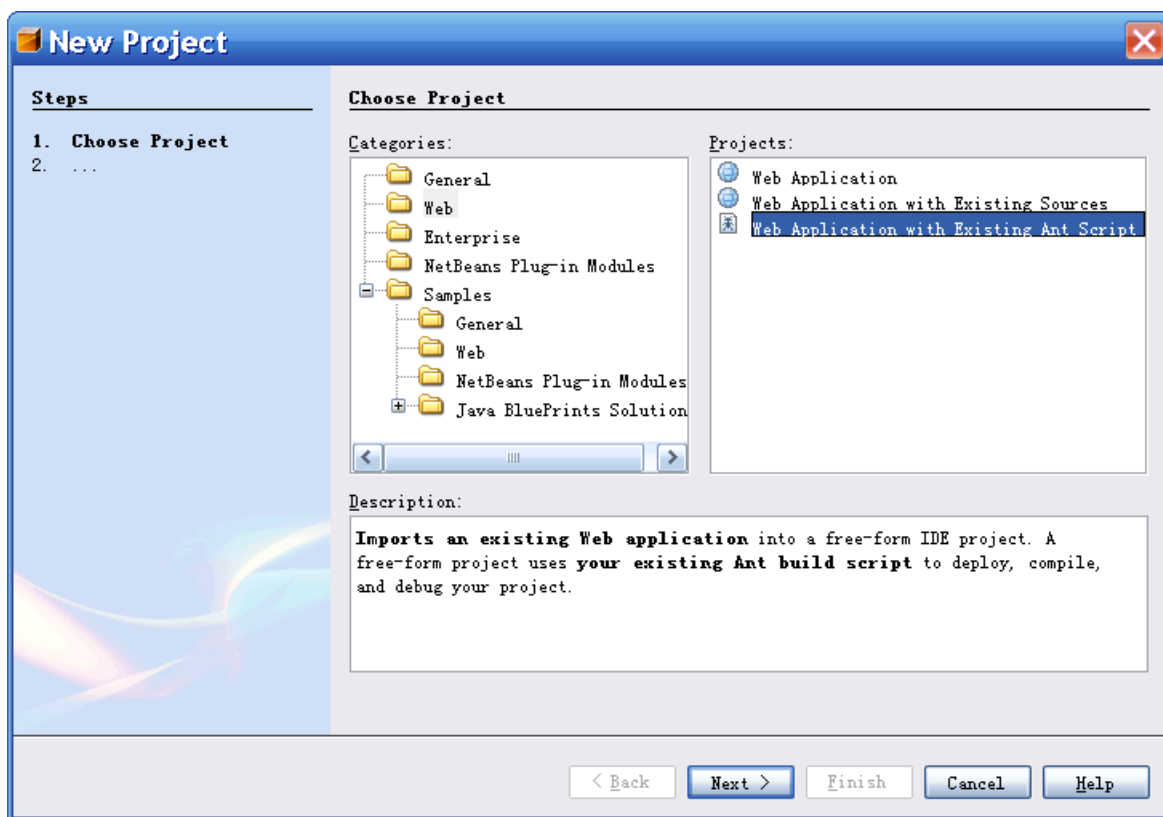
NetBeans从4.0发布引起了java社区的高度关注,这主要是因为4.0及以后的版本采用ant管理项目。5.0引入了Struts, EJB, Web Service, 并且有大量开源NB module可用, Spring, Hibernate, Docobook等也可以通过安装nbextra得到支持。5.5中会加入原来Sun Java Studio Enterprise中的特性,将引用UML(EJB1.0, EJB2.0, GoF), BPEL, SOA等。

下面以NetBeans5.0为例。

首先请确定下载了Equinox(<http://appfuse.dev.java.net>), 安装了JDK(<http://java.sun.com/j2se>), Tomcat(<http://tomcat.apache.org>), Ant(<http://ant.apache.org>)。

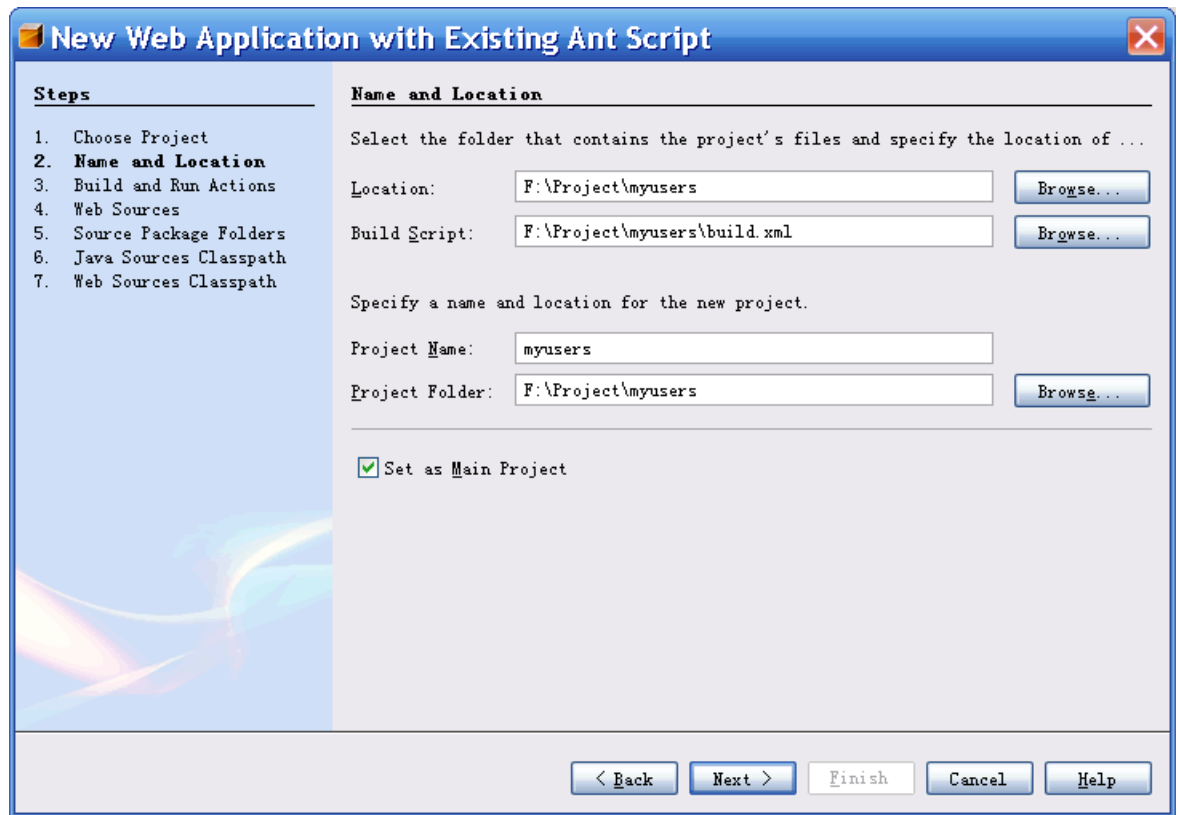
1. 启动Netbeans 5.0, 选择File->New Project。左边框中, 选择Web, 右边选择Web Application with Existing Ant Scripts。点击Next。

图 1.



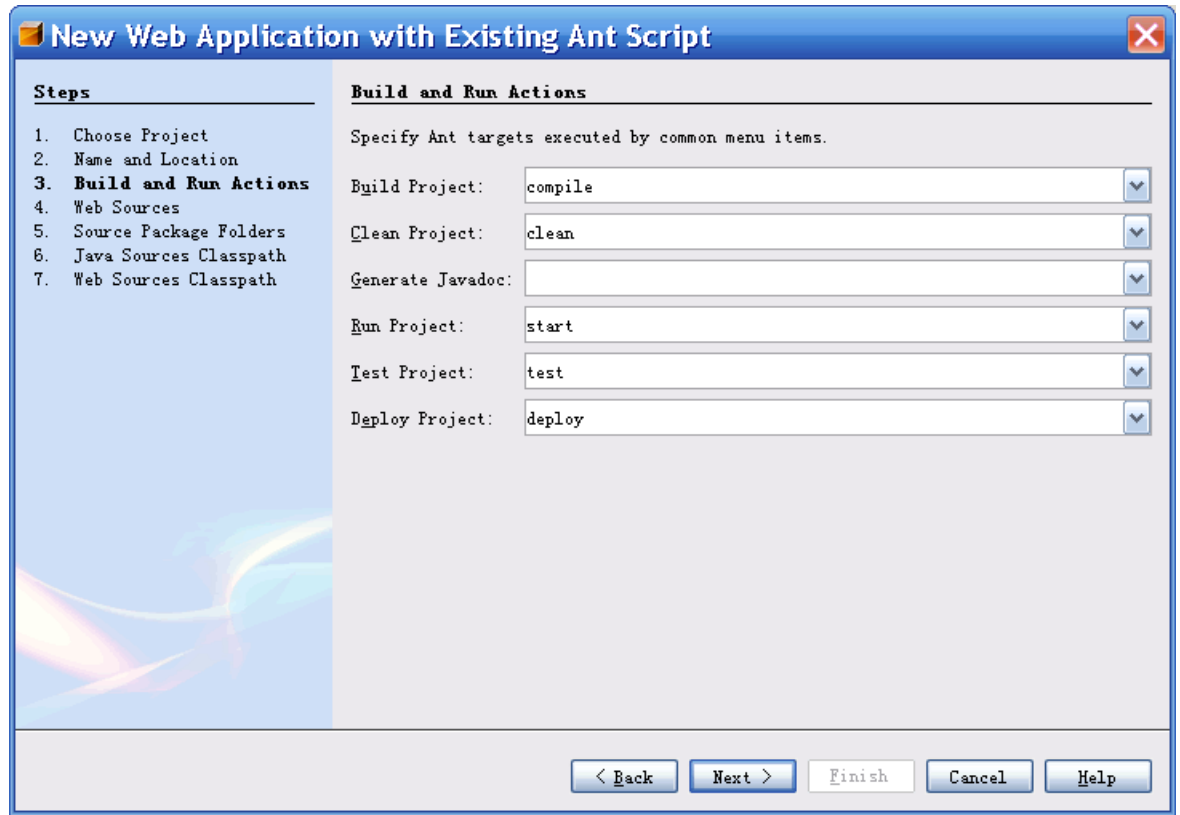
2. 选择现有项目的位置，输入新项目的名称和位置。点击Next。

图 2.



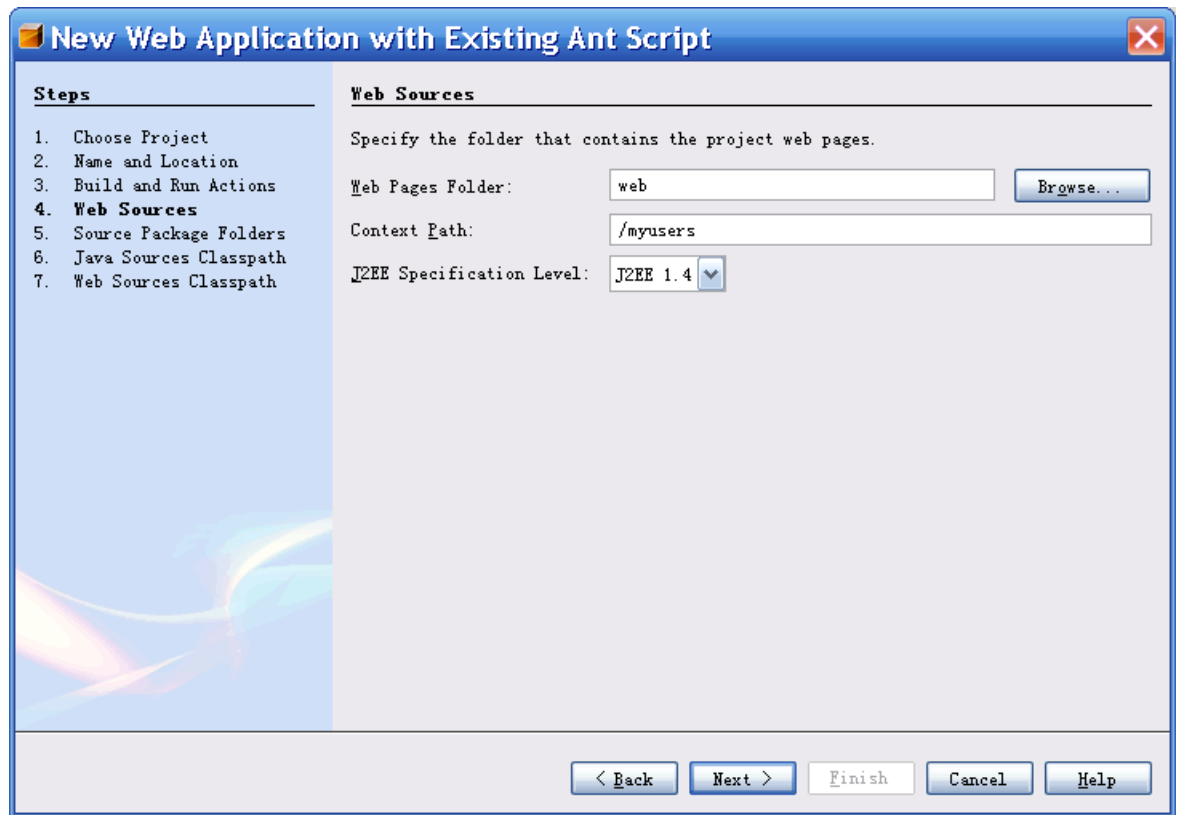
3. 指定与右键菜单相应的Ant target。点击Next。

图 3.



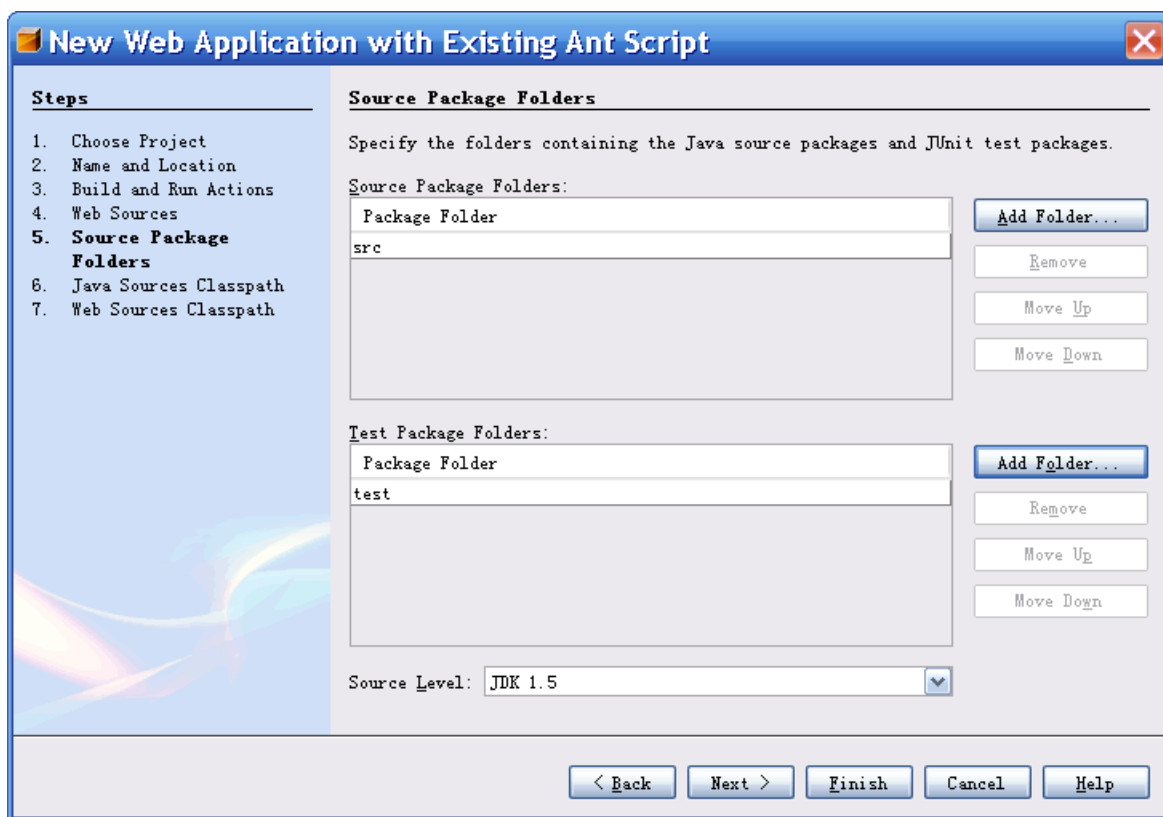
4. 输入web目录及context path。点击Next Step。

图 4.



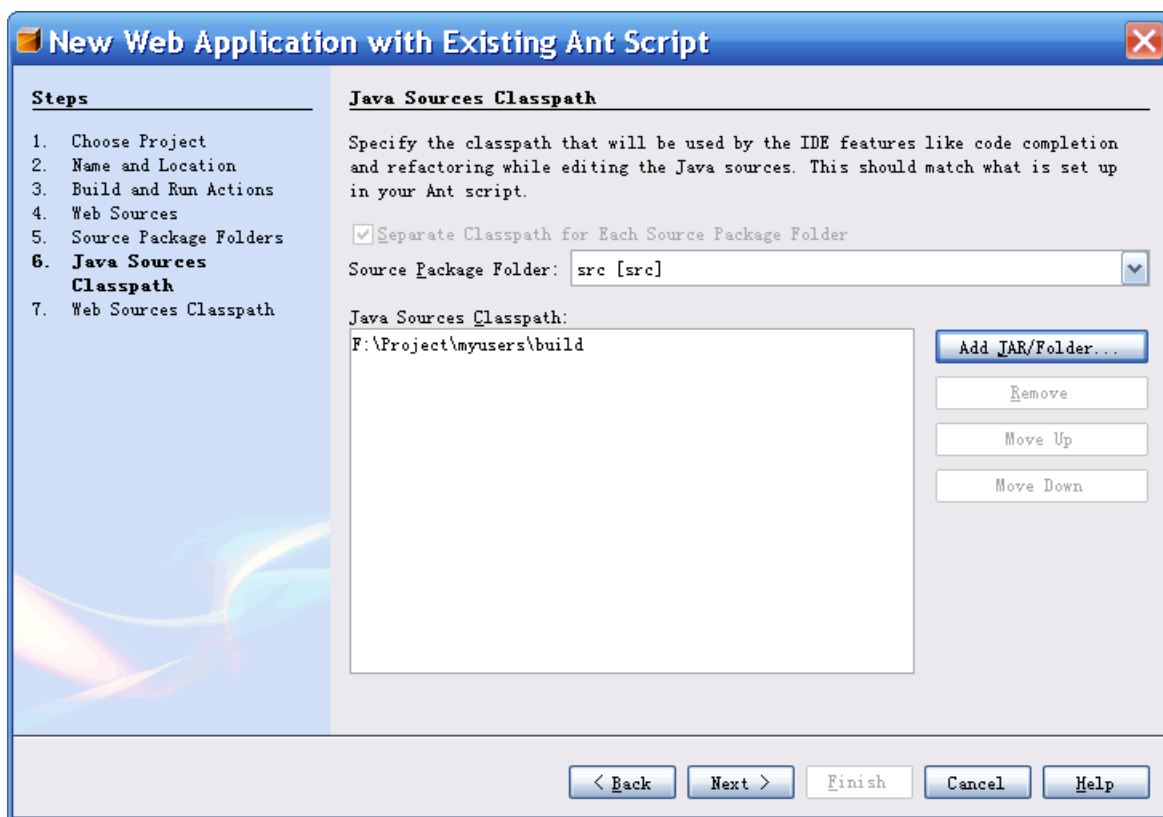
5. 指定源文件和测试包目录。点击Next Step。

图 5.



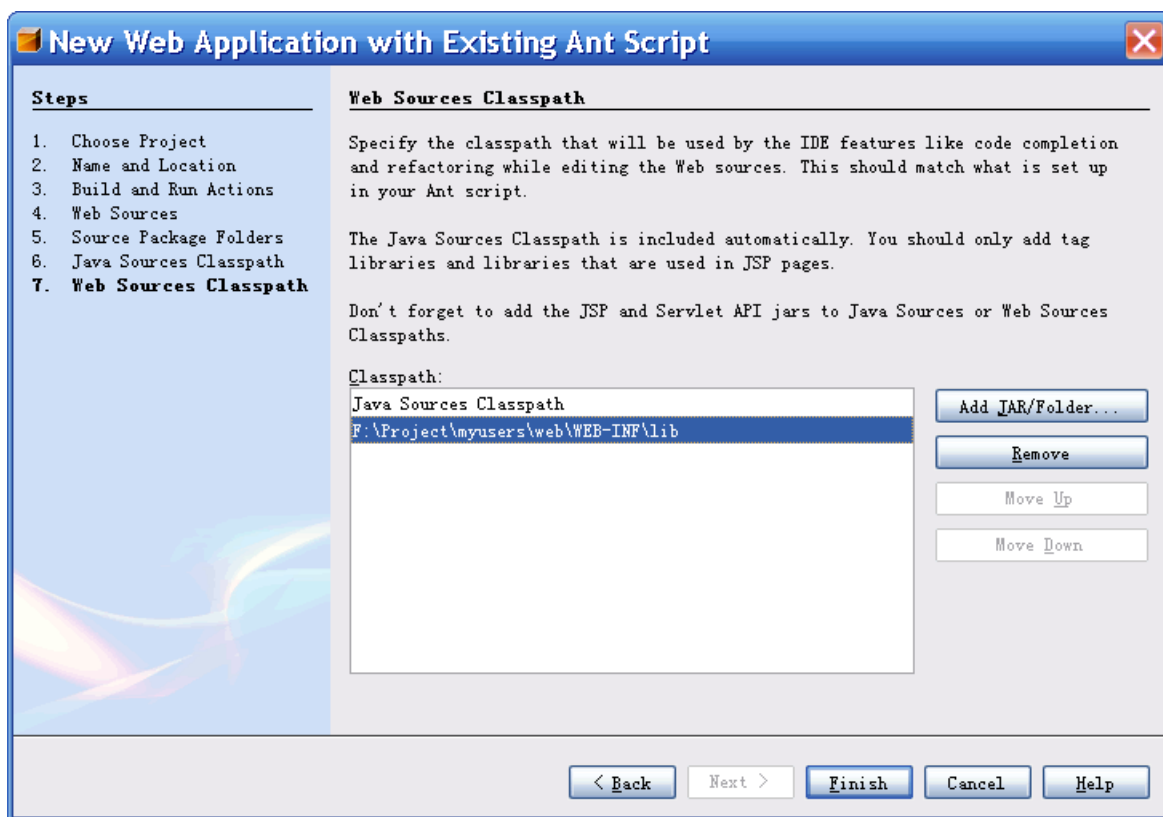
6. 指定java源文件的classpath。点击Next Step。

图 6.



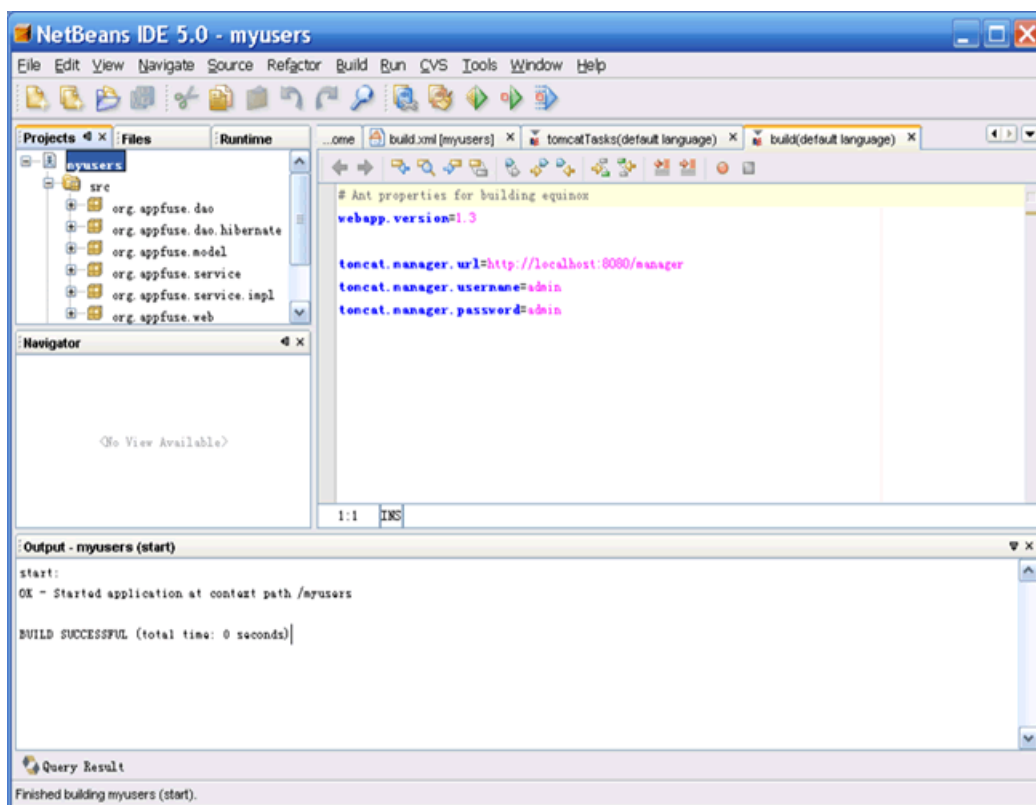
7. 指定web资源的classpath。点击完成。

图 7.



8. 如果事先设置了TOMCAT_HOME, ANT_HOME, JAVA_HOME, 并设置一个role为manager的管理用户(参见第2章), 在Project窗口右键点击myusers项目, 选择Build Project, 就会运行ant script中compile target, 编译整个项目。运行Redeploy Project, 就会运行ant script中的deploy target, 将项目部署到Tomcat运行环境中。运行Run Project, 就运行start target, 启动myusers应用程序。

图 8.



9. 此时打开浏览器，浏览 <http://localhost:8080/myusers> 就可以看到主页面。

图 9.



有关Eclipse, IntelliJ IDEA中配置Equinox的方法, 可以从Spring Live作者Blog上找到。

第 1 章 Spring 简介

Spring基础和它的来历

本章向你介绍Spring的基础，发展过程，以及为何它能得到如此多的报道和热评。还通过与传统解决依赖性的方法作了比较(使用Factory模式创建接口来实现)，说明Spring是如何在XML中解决这些问题。本章还简短的说明了Spring如何简化Hibernate API。

Spring 诞生记

Rod Johnson是Spring天才的缔造者。Spring起源他2002年末所著Expert One-on-One J2EE Design and Development一书的基础代码。在书中Rod介绍他自己丰富的J2EE经验，并且解释了EJB为何常常葬送了整个项目。他坚信一种轻量级的，基于javabean的框架完全可以满足大多数开发人员的要求。如果你是一名开发人员，还没有看过这本书的话，我推荐你读一下。

2003年2月，他把所描述的框架公开源代码，放在sourceforge.net上，最后，这个框架就是演变成了著名的Spring框架。那时候，Rod和Juergen Hoeller是Spring的首要开发人员和核心人物。在最近的几个月中，Rod和Juergen又邀请了大量开发人员的参与开发。在撰写本书时，Spring委员会名册中已经有16个成员。Rod和Juergen最近又合著了一本名为Expert One-on-One J2EE Development without EJB的书，讲解了如何运用Spring 来解决众多的J2EE问题。

早在2000年(在我所熟悉的Struts和其他框架之前)，Rod已经为Spring开发了架构基础。这些基础来源于Rod大量成功的商业项目经验,后来不断的得到数百(可能有上千)的开发人员的改进和加强。每个人都贡献自己的经验为其添砖加瓦，你可以目睹Spring变得日益强大。Spring的社区非常活跃，开发人员也非常热心和投入，我想这对J2EE来说是一件盛事。

关于Spring

根据Spring的官方网站描述，“Spring是一种多层的J2EE应用程序框架，它是以Rod Johnson编著的Expert One-on-One J2EE Design and Development一书的代码为基础发展而来的。”Spring的核心，提供一种新的机制来管理业务对象其依赖关系。例如，利用IoC(反转控制)，你可以指定一个DAO(数据访问对象)类依赖一个DataSource类。它也允许开发人员通过接口编程，使用xml文件来简单的定义其实现。Spring有许多类用来支持其它的框架(如Hibernate和Struts)，这使得集成变得易如反掌。

一味的遵循J2EE设计模式(Design Pattern)有时会感到寸步难移，并且无济于事(事实上常常演变成反模式(anti-pattern))。Spring似乎也遵循了设计模式，但一切都得到了简化。例如，你再也没有必要写一个ServiceLocator来查找Hibernate Sessions，你可以在Spring中配置一个SessionFactory。这样，让你领会J2EE领域专家的最佳实践，而不是创建出最新的模式。

为什么每个人都钟情于它

如果你常常游览诸如TheServerSide.com或是JavaLobby.org的在线论坛，就会发现经常提及Spring。它在java博客社区(如JavaBlogs.com和JRoller.com)更有吸引力。很多开发人员叙叙他们的开发经验，称赞其易用性。

Spring不仅解决了开发人员的实际问题，也促使他们养成良好的编程习惯，例如，接口编程，减少耦合及提供简易的测试。在当今的编程领域中，特别是在java中，一些优秀的开发人员正在实践TDD(Test Driven Development，测试驱动开发)。TDD让你用测试类或是client类驱动其它类的设计。而不是先建好一个类，再修正client，它要求事先写client。通过这种方法，你恰好可以知道你正在开发的类中需要哪些东西。Spring本身提供了丰富的测试套件，这使得你在测试类时变得轻松自如。

比较一下J2EE的“最佳实践”，其blueprints(蓝本)中建议你使用EJB来处理业务逻辑。EJB需要一个EJB容器来运行，所以你必须先启动它来进行测试。什么时候才是你最后一次启动服务器(如Weblogic, WebSphere和JBoss)? 如果你不得不反复的测试你的类，它实际上也在测试你的耐性。

注意

最近有一篇题为“Testing EJBs with OpenEJB”的文章，展示一种快速测试EJB的方法，但仅仅是围绕EJB内部的可测试性问题。

对Spring的一些常见的批评

取得成功的同时，也听到一些不和谐的音符。其中我所见到的反对Spring的最激烈的争论莫过于指责它不标准，意即它不是J2EE规范的一部分，不是由JCP完成开发的。那些反对Spring的人理所当然的拥护EJB，因为它才是标准。然而，树立标准的首要因素是保证在各个应用服务器之间的可移植性。你为某个服务器所写代码应该可以在另一个服务器上运行，然而在服务器之间移植EJB并非易事。不同的供应商要求使用不同的部署描述器(文件)，在配置datasource(数据源)和其它依赖容器的东西时也是大相径庭。然而用Spring编写业务逻辑可以在各容器间轻松移植，甚至不用修改你的代码或部署文件。

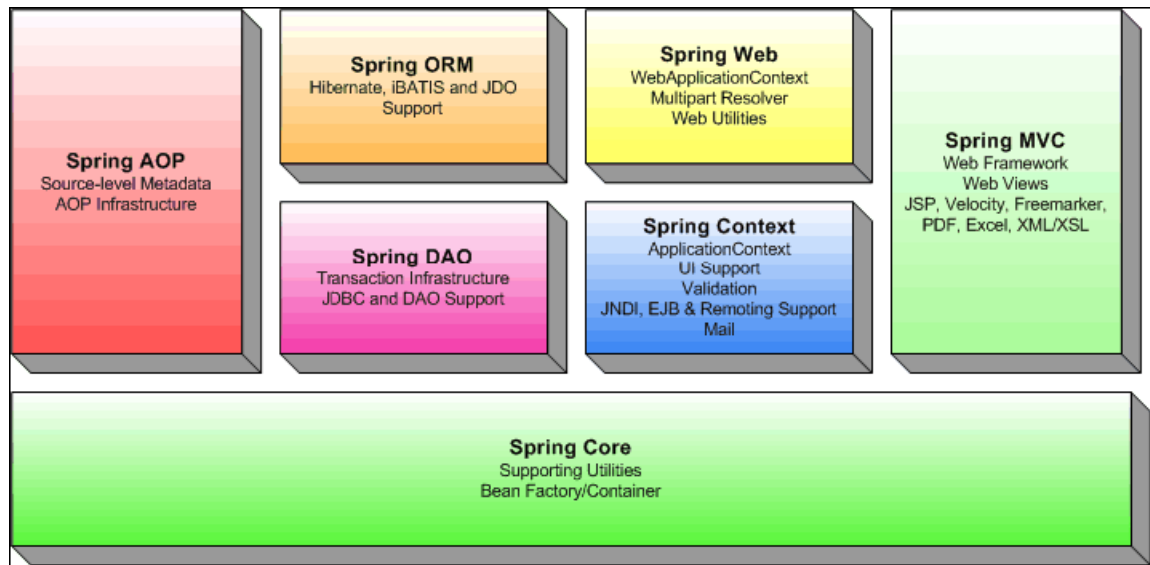
在Spring“把一切变得更加简单”的同时，一些开发人员开始抱怨它“太肥了”。然而，Spring实际上是一种菜单式的框架，你可以根据需要进行点选。(Spring)开发团队已经对发布产品进行了分割，你可以根据需要选择所要的jar文件。

Spring 的工作原理

J2EE Design and Development一书向我们阐述了Spring及其工作原理。Spring是一种用javaBean配置应用程序的方法。当我提及javaBean时，我指的那些带有getter和setter(所谓的accessors和mutators)的java类。特别是，如果一个类暴露了setter，Spring可以配置这个类。运用Spring，你可以通过一个setter来暴露一个类任何依赖关系(如，一个数据库连接，database connction)，并且在Spring中进行配置来解决依赖性。更妙的是，你根本不用写一个数据连接，这也可以在Spring进行配置。这种解决依赖性的方法称为：反转控制(Inversion of Control)或是依赖性注射(Dependency Injection)。从根本上说，这是一个技术术语，用来解释使用某种容器来贯穿相互依赖的对象。

Spring有7个相互独立的模块，每个模块都有一个对应的jar文件。下面7个模块的示意图。

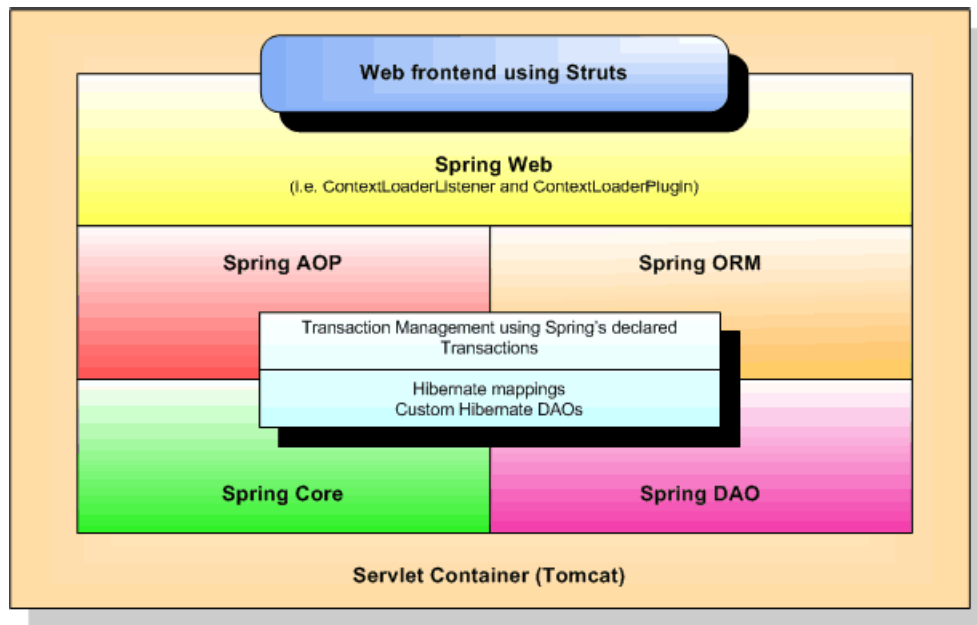
图 1.1. Spring的7个模块



在第2章将要开发的MyUsers程序中，会用到上面的一些模块，但不是全部。此外要说明的是，你用到的仅仅是每个模块的一些功能片断。

下图显示了MyUsers程序中将会用到的模块。

图 1.2. 使用Spring的MyUsers应用程序框架



Spring如何简化J2EE开发

在图1.2中，Spring提供了大量的你要创建的部分。初一看，这似乎也太霸道，你不得不去熟悉Spring。其实不然，在大多数情况下，你甚至看不到使用Spring API。例如，在中间层，你只要设置一个声明式的业务处理，并把DAO放在业务委派(business delegate)中。你的代码中，既看不到有

导入Spring的部分，也找不到任何可以判定出DAO使用了哪种实现的Factory模式。所有的一切都是在xml文件中搞定。如此清晰的设计非你莫属。

下面几个部分讨论Spring简化开发的一些过程。

接口编程

接口编程让开发人员知道他们要用到的类中将会使用哪些方法。使用接口对设计你的应用程序非常有用，因为你在具体实现时有很大的灵活性。此外，各层之间的通信使用接口有利于代码的解耦。

简单的测试

使用TDD(Test-Driven Development)是产生质量代码的最佳途径。它要求你在编写接口或实现之前，通过编写client(测试类)来推动你的设计。事实上，在你编写测试程序时，现在的大多数IDE如Eclipse, IntelliJ IDEA都支持拖放操作来创建方法和接口。启用Spring的项目测试方便是基于两个方面的原因：

- 你可以在JUnit测试类中轻松导入和使用所有Spring托管的bean。你可以从client和它们进行正常交互。
- 你不用绑定它们的依赖关系。这样你可以在测试程序忽略Spring的存在而使用mock object设置依赖关系。

降低耦合：Factory Pattern vs Spring

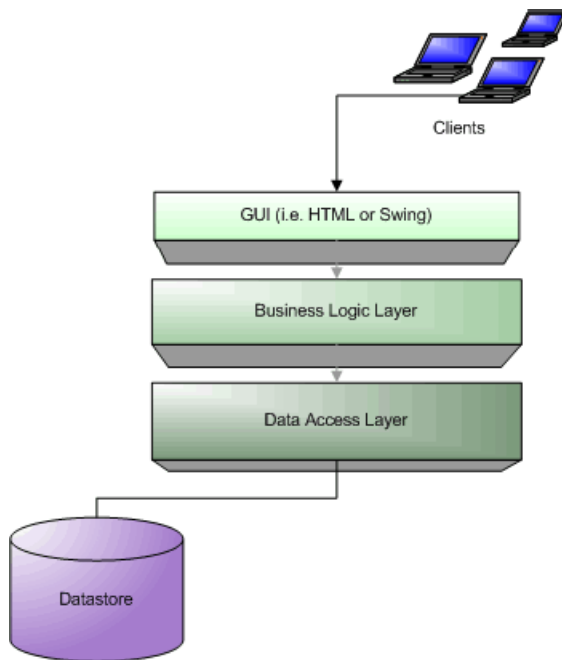
要想拥有一个方便管理并且有扩展性的程序，将你的代码锁定某一特定的资源(例如，你编写的SQL功能只针对某一数据库类型)上并不理想。当然，如果使用某种特性能帮你更快的完成工作，那么针对某一特定的数据库编程通常会容易些。当你在类不再使用这种特性时,J2EE Patterns建议你使用一个Factory Pattern从实现类对你的程序进行解耦。

一般来说，在各层之间创建接口是个不错主意。这样各个单独的层会被其实现类忽略。一个典型的J2EE应用程序有3层。

- 数据层(Data Layer):负责与数据或是其它存储系统通信的类。
- 业务逻辑(Business Layer):在GUI层和数据之间搭一座桥，控制业务逻辑的类。
- 用户界面(User Interface):为用户创建web页面和桌面程序接口的类和视图(view)文件。

图1.3是一个典型的J2EE应用程序的图形表示。

图 1.3. 一个典型的J2EE应用程序



Factory Pattern(GoF模式中也称为Abstract Factory和Factory Method)让你在只使用少数几个factory类情况下，可以轻松从一种实现切换到另一种实现。一般来说，你要编写一个DAOFactory类和某种特定实现的factory类(例如，DAOFactoryMySQL)。更多信息，参考J2EE Patterns Catalog中的Core J2EE Patterns – Data Access Object。

配置和绑定类的依赖关系

Factory Pattern是一种复杂的J2EE模式。你不但要写两个类来进行设置，还需要引入管理被工厂化(factoryed)的类的依赖关系的内容。例如，如果你要从factory中获得一个DAO，你如何在唯一的一个数据库连接(而不是为每个方法打开一个新的连接)中进行传递？你可以把它作为构造器的一部分进行传递，但是如果你使用DAO实现需要一个Hibernate Session呢？可以为构造器添加一个参数java.lang.Object，然后根据需要进行强制转换，只是看起来比较难看罢了。

一种更好的途径是使用Spring来绑定接口和其实现。所有的东西都是在xml文件中配置，你只要修改一直xml文件就可以方便切换到另一种实现。更好的是，你写了一个单元测试，而没有人知道你用的是哪种实现，你可以使用不同的实现来运行测试程序。测试类实际上是一个client，它能保证你的业务逻辑可以和另一种备用的实现很好的结合在一起，这是个不错的主意。这里有一个利用Spring来获得一个UserDAO具体实现的例子：

```

ApplicationContext ctx =
    new ClassPathXmlApplicationContext("/WEB-INF/applicationContext.xml");
UserDAO dao = (UserDAO) ctx.getBean("userDAO");
  
```

你必须在/WEB-INF/applicationContext.xml文件中配置一个UserDAO bean。下面是第2章中的代码片断：

```

<bean id="userDAO" class="org.appfuse.dao.hibernate.UserDAOHibernate">
  <property name="sessionFactory">
    <ref local="sessionFactory"/>
  </property>
</bean>
  
```



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

如果你想修改UserDAO的实现，你所要做的只是修改上面xml文件片断中的class属性。你可以在你的应用使用这种更加简捷的模式。你要做的是只是在你的bean定义文件中添加几行代码。此外，Spring通过一个sessionFactory属性来管理这个DAO的Hibernate Session。你不必再担心没有关闭它。

注意

Spring作为一种“轻量级的容器”常常被提及到，因为你可以通过它的“ApplicationContext”来获得一个实例化的对象。这些对象定义在一个context文件(也称为bean定义文件)中。这个文件只是一个简单的xml文件，由一系列的bean元素组成。从某种角度讲，它是一个“bean容器(bean container)”或是“对象库(object library)”，所有的东西都已经设置好了，随时可以调用。从传统的角度(如Tomcat，Weblogic)来讲，它并不是真正的容器，更象一个“配置好的bean供给装置(configured bean provider)”。

对象关系映射工具

Spring另一个可用性的例子是它为ORM工具(Object/Relational Mappings，对象关系映射)提供了最好的支持。使用ORM支持类的第一个好处是，你不需要在处理检测式异常(checked exception)使用try/catch。Spring将检测式异常包装成运行时异常(runtime exception)，让开发人员自己决定是否要捕捉异常。下面的例子是没有使用Spring时UserDAOHibernate类的getUsers()方法。

```
public List getUsers() throws DAOException {
    List list = null;
    try {
        list = ses.find("from User u order by upper(u.username)");
    } catch (HibernateException he) {
        he.printStackTrace();
        throw new DAOException(he);
    }
    return list;
}
```

下面的例子使用了Spring的Hibernate支持(通过继承HibernateDaoSupport),代码看起来更为简短。

```
public List getUsers() {
    return getHibernateTemplate()
        .find("from User u order by upper(u.username)");
}
```

从这些例子中，你可以看到Spring在应用程序各层之间，各依赖关系中解耦及处理配置和绑定一个类的依赖关系时变得更为方便。它也大大简化了使用ORM工具如Hibernate的API。

本章小结

本章讨论了Spring的历史，人们喜欢它的原因，及它是如何简化J2EE开发的。并举了实例对使用传统Factory(工厂)模式与Spring的ApplicationContext进行了比较，还提供了一个使用Hibernate开发前后的对照。Spring的ApplicationContext可以认为是一个bean provider，用来处理实例化的类，绑定它们的依赖关系，并对它们进行预配置。

第2章是一个web应用开发的教程，使用Spring,Struts及Hibernate来管理数据库里面的用户。这个应用程序演示了Spring如何使J2EE和TDD变得更加简单。

第3章探讨Spring的核心模块，以及它所管理的bean的生命周期。这是Spring的心脏和大脑--控制着各对象的协调运作，并为它们提供必要的支持。

第 2 章 Spring快速入门教程

开发第一个Spring程序

本章学习用Struts MVC框架作前端，Spring作中间层，Hibernate作后端来开发一个简单的Spring应用程序。在第4章将使用Spring MVC框架对它进行重构。

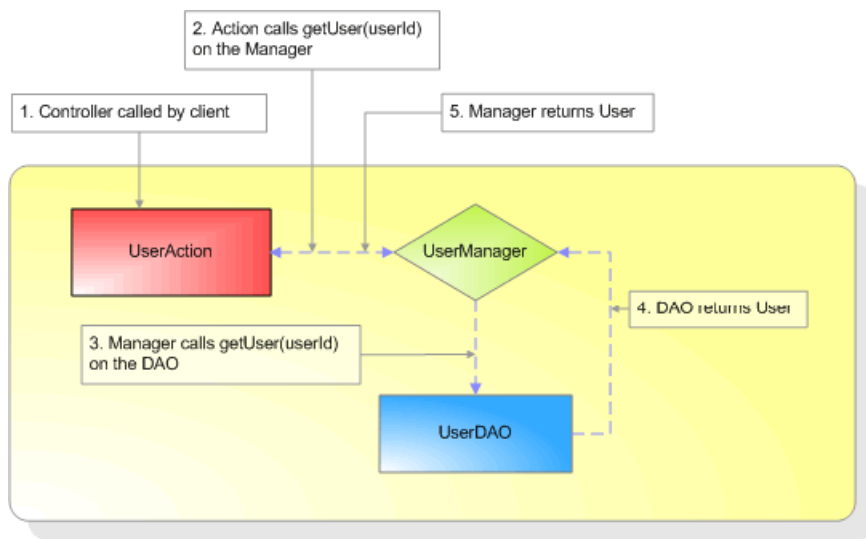
- 编写功能测试。
- 配置Hibernate和Transaction。
- 载入Spring的ApplicationContext.xml文件。
- 设置业务委派(business delegates)和DAO的依赖性。
- 集成Spring和Struts。

概述

下载Struts和Spring

你将会创建一个简单的程序完成最基本的CRUD(Create, Retrieve, Update和Delete)操作。这个程序叫MyUsers，作为本书的样例。这是一个三层架构的web程序，通过一个Action来调用业务委派，再通过它来回调DAO类。下面的流程图表示了MyUsers是如何工作的。数字表明了流程的先后顺序，从web层(UserAction)到中间层(UserManager)，再到数据层(UserDAO)，然后返回。

图 2.1. MyUsers应用程序流程



鉴于大多数读者都比较熟悉Struts，本程序采用它作为MVC框架。Spring的过人之处在于它声明式的事务处理，依赖性的绑定和持久性的支持(如Hibernate和iBATIS)。第4章中将用Spring框架对它进行重构。

接下来你会进行以下几个步骤：

1. 下载Struts和Spring。
2. 创建项目目录和Ant Build文件。
3. 为持久层创建一个单元测试(unit test)。
4. 配置Hibernate和Spring。
5. 编写Hibernate DAO的实现。
6. 进行单元测试，通过DAO验证CRUD。
7. 创建一个Manager来声明事务处理。
8. 为Struts Action 编写测试程序。
9. 为web层创建一个Action和model(DynaActionForm)。
10. 进行单元测试，通过Action验证CRUD。
11. 创建JSP页面，以通过浏览器来进行CRUD操作。
12. 通过浏览器来验证JSP页面的功能。
13. 用Velocity模板替换JSP页面。
14. 使用Commons Validator进行验证。

下载Struts和Spring

1. 下载安装以下组件：
 - JDK 1.4.2(或以上)
 - Tomcat 5.0+
 - Ant 1.6.1+
2. 设置以下环境变量：
 - JAVA_HOME
 - ANT_HOME
 - CATALINA_HOME
3. 把以下路径添加到PATH中：
 - JAVA_HOME/bin
 - ANT_HOME/bin
 - CATALINA_HOME/bin

为了开发基于Java的web项目，开发人员必须事先下载必需的jars，准备好开发目录结构和Ant build文件。对于单一的Struts项目，可以利用Struts中现成的struts-blank.war。对于基于Spring MVC框架的项目，可以用Spring中自带的webapp-minimal.war。这些都是不错的起点，但两者都没有进行Struts-Spring集成，也没有考虑单元测试。为此，我们为读者准备了Equinox。

Equinox为开发Struts-Spring的程序提供一个基本框架。它已经定义好了目录结构，和Ant build文件(针对compiling,deploying,testing)，并且提供了Struts，Spring，Hibernate开发要用到的jars文件。Equinox中大部分目录结构和Ant build文件来自我的开源项目——AppFuse。可以说，Equinox是一个简化的AppFuse，它在最小配置情况下，为快速web开发提供了便利。由于Equinox源于AppFuse，所以在包名，数据库名，及其它地方都找到它的影子。这样做是为了让你从基于Equinox的程序过渡到更为复杂的AppFuse。

从SourceBeat [<http://sourcebeat.com/downloads>]上下载Equinox，解压到一个合适的位置，开始准备MyUsers的开发。

创建项目目录和Ant Build文件

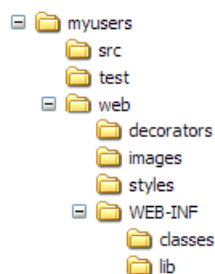
为了设置初始的目录结构，把下载的Equinox解压到硬盘上。建议Windows用户把项目放在C:\Source，UNIX/Linux用户放在~/dev(译注：在当前用户目录建一个dev目录)中。Windows用户可以设置一个HOME环境变量，值为C:\Source。最简单的方法是把Equinox解压到你的喜欢的地方，进入equinox目录，从命令行运行ant new -Dapp.name=myusers。

注意

在Windows上，我使用cygwin(www.cygwin.org)，这样就可以像UNIX/Linux系统一样使用正斜杠，本书所有路径均采用正斜杠。其它使用反斜杠系统(如Windows中命令行窗口)的用户请作相应的调整。

现在MyUsers程序已经有如下的目录结构：

图 2.2. MyUsers应用程序目录结构



Equinox包含一个简单而功能强大的build.xml，它可以用Ant来进行编译，布署，和测试。要查看所有可用的Ant target,在MyUsers目录下键入ant,回车后将看到如下内容：

```

[echo] Available targets are:
[echo] compile --> Compile all Java files
[echo] war --> Package as WAR file
[echo] deploy --> Deploy application as directory
[echo] deploywar --> Deploy application as a WAR file
[echo] install --> Install application in Tomcat
[echo] remove --> Remove application from Tomcat
[echo] reload --> Reload application in Tomcat
  
```

```
[echo] start --> Start Tomcat application
[echo] stop --> Stop Tomcat application
[echo] list --> List Tomcat applications
[echo] clean --> Deletes compiled classes and WAR
[echo] new --> Creates a new project
```

Equinox支持Tomcat的Ant task(任务)。这些task已经集成在Equinox中，解讲一下如何进行集成的有助于理解它们的工作原理。

Tomcat和Ant

Tomcat中定义了一组任务，可以通过Manager程序来安装(install)，删除(remove)，重载(reload)webapps。要使用这些任务，可以把所有的定义写在一个property文件中。在Equinox的根目录下，有一个名为tomcatTasks.properties的文件，其内容如下。

```
deploy=org.apache.catalina.ant.DeployTask
undeploy=org.apache.catalina.ant.UndeployTask
remove=org.apache.catalina.ant.RemoveTask
reload=org.apache.catalina.ant.ReloadTask
start=org.apache.catalina.ant.StartTask
stop=org.apache.catalina.ant.StopTask
list=org.apache.catalina.ant.ListTask
```

在build.xml定义一些task来安装，删除，重新加载应用程序。

```
<!-- Tomcat Ant Tasks -->
<taskdef file="tomcatTasks.properties">
  <classpath>
    <pathelement
      path="${tomcat.home}/server/lib/catalina-ant.jar"/>
    </classpath>
  </taskdef>
<target name="install" description="Install application in Tomcat">
  depends="war">
    <deploy url="${tomcat.manager.url}"
      username="${tomcat.manager.username}"
      password="${tomcat.manager.password}"
      path="/${webapp.name}"
      war="file:${dist.dir}/${webapp.name}.war"/>
  </target>
<target name="remove" description="Remove application from Tomcat">
  <undeploy url="${tomcat.manager.url}"
    username="${tomcat.manager.username}"
    password="${tomcat.manager.password}"
    path="/${webapp.name}"/>
  </target>
<target name="reload" description="Reload application in Tomcat">
  <reload url="${tomcat.manager.url}"
    username="${tomcat.manager.username}"
    password="${tomcat.manager.password}"/>
  </target>
```

```
    path="/${webapp.name}"/>
</target>
<target name="start" description="Start Tomcat application">
    <start url="${tomcat.manager.url}"
        username="${tomcat.manager.username}"
        password="${tomcat.manager.password}"
        path="/${webapp.name}"/>
</target>
<target name="stop" description="Stop Tomcat application">
    <stop url="${tomcat.manager.url}"
        username="${tomcat.manager.username}"
        password="${tomcat.manager.password}"
        path="/${webapp.name}"/>
</target>
<target name="list" description="List Tomcat applications">
    <list url="${tomcat.manager.url}"
        username="${tomcat.manager.username}"
        password="${tomcat.manager.password}"/>
</target>
```

在上面列出的target中，必须预先定义一些\${tomcat.*}变量。在根目录下有一个build.properties默认定义如下：

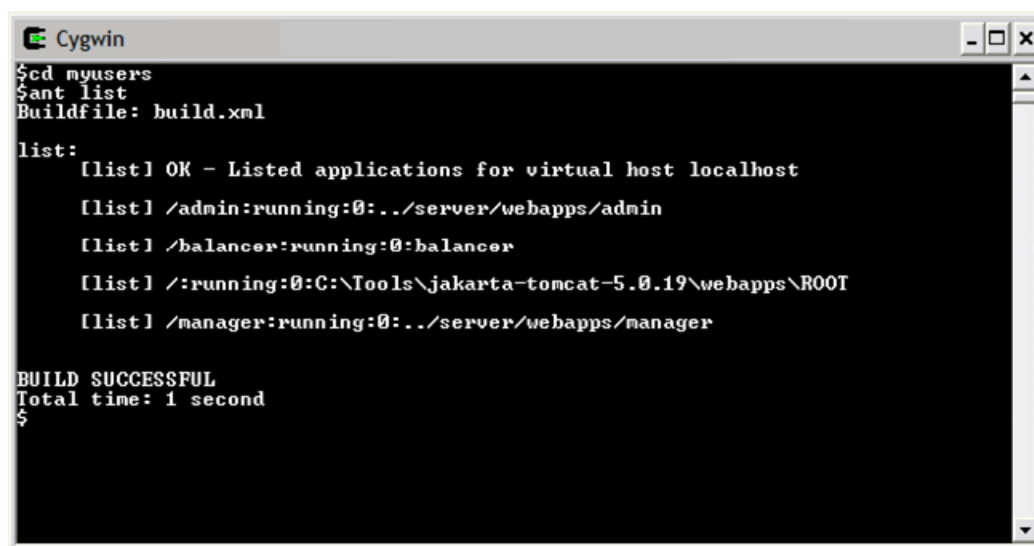
```
# Properties for Tomcat Server
tomcat.manager.url=http://localhost:8080/manager
tomcat.manager.username=admin
tomcat.manager.password=admin
```

确保admin用户可以访问Manager应用程序，打开\$CATALINA_HOME/conf/tomcat-users.xml中是否存在下面一行。如果不存在，请自己添加。注意，roles属性可能是一个以逗号(",")隔开的系列。

```
<user username="admin" password="admin" roles="manager"/>
```

为了测试所有修改，保存所有文件，启动Tomcat。从命令行中进行MyUsers目录，运行ant list，可以看到Tomcat服务器上运行的应用程序。

图 2.3. 运行ant list命令的结果



```
Cygwin
$cd myusers
$ant list
Buildfile: build.xml

list:
[list] OK - Listed applications for virtual host localhost
[list] /admin:running:0:../server/webapps/admin
[list] /balancer:running:0:balancer
[list] /:running:0:C:\Tools\jakarta-tomcat-5.0.19\webapps\ROOT
[list] /manager:running:0:../server/webapps/manager

BUILD SUCCESSFUL
Total time: 1 second
$
```

好了，现在运行ant deploy来安装MyUsers。打开浏览器，在地址栏中输入http://localhost:8080/myusers，出现如图2.4的“Equinox Welcome”画面。

图 2.4. Equinox欢迎页面



警告

为了使驻留在内存的HSQL能很好的结合MyUsers程序，从你运行Ant的同一目录中启动Tomcat。在UNIX/Linux下键入\$CATALINA_HOME/bin/startup.sh，Windows下运行%CATALINA_HOME%\bin\startup.bat。你也可以修改数据库设置使用绝对路径。

在接下来的几节中，你将会创建一个User对象和一个维护其持久性的Hibernate DAO对象。用Spring来管理DAO类及其依赖关系。最后，还会创建一个业务委派，来使用AOP和声明式事务处理。

为持久层编写单元测试

在MyUsers程序，使用Hibernate作为持久层。Hibernate是一个O/R映像框架，用来关联Java对象和数据库中的表(tables)。它使得对象的CRUD操作变得非常简单，Spring结合了Hibernate变得更加容易。从Hibernate转向Spring+Hibernate会减少75%的代码。这主要是因为，ServiceLocator和一些DAOFactory类的废弃，Spring的运行时异常代替了Hibernate的检测式异常(checkedException)。

写一个单元测试有助于规范UserDAO接口。为UserDAO写一个JUnit测试程序，需要完成以下几步：

1. 在test/org/appfuse/dao下新建一个UserDAOTest类。它继承了同一个包中的BaseDAOTestCase，其父类初始化了Spring的ApplicationContext(来自web/WEBINF/ApplicationContext.xml)，以下是Spring测试的代码。

```
package org.appfuse.dao;
// use your IDE to handle imports

public class UserDAOTest extends BaseDAOTestCase {
    private User user = null;
    private UserDAO dao = null;

    protected void setUp() throws Exception {
        super.setUp();
        dao = (UserDAO) ctx.getBean("userDAO");
    }

    protected void tearDown() throws Exception {
        super.tearDown();
        dao = null;
    }
}
```

这个类还无法通过编译，因为还没有创建UserDAO接口。在这之前，来写一些来验证User的CRUD操作。

2. 在UserDAOTest类中添加testSave和testAddAndRemove方法，如下所示：

```
public void testSaveUser() throws Exception {
    user = new User ();
    user.setFirstName("Rod");
    user.setLastName("Johnson");
    dao.saveUser(user);
    assertNotNull("primary key assigned", user.getId());
    log.info(user);
    assertNotNull(user.getFirstName());
}

public void testAddAndRemoveUser() throws Exception {
```

```
user = new User();
user.setFirstName("Bill");
user.setLastName("Joy");
dao.saveUser(user);
assertNotNull(user.getId());
assertEquals(user.getFirstName(), "Bill");
if (log.isDebugEnabled()) {
    log.debug("removing user...");
}
dao.removeUser(user.getId());
assertNull(dao.getUser(user.getId()));
}
```

从这些方法中可以看到，你需要在UserDAO创建以下方法：

- saveUser(User)
- removeUser(Long)
- getUser(Long)
- getUsers() (返回数据库的所有用户)

3. 在src/org/appfuse/dao目录下建一个名为UserDAO.java类的，输入以下代码：
注意

如果你使用Eclipse, IntelliJ IDEA之类的IDE，左边会出现在一个灯泡，提示类不存在，可以即时创建。

```
package org.appfuse.dao;
// use your IDE to handle imports

public interface UserDAO extends DAO {
    public List getUsers();
    public User getUser(Long userId);
    public void saveUser(User user);
    public void removeUser(Long userId);
}
```

为了让UserDAO.java, UserDAOTest.java编译通过，还要建一个User.java类。

4. 在src/org/appfuse/model下建一个User.java文件， 添加几个成员变量：id, firstName, lastName, 如下所示。

```
package org.appfuse.model;

public class User extends BaseObject {
    private Long id;
    private String firstName;
    private String lastName;
}
```



```
/*
Generate your getters and setters using your favorite IDE:
In Eclipse:
Right-click -> Source -> Generate Getters and Setters
*/
}
```

注意，你继承了BaseObject类，它包含几个有用的方法：toString()，equals()，hashCode()，后两个是Hibernate必需的。

建好User对象后，用IDE打开UserDAO和UserDAOTest两个类，优化导入。

配置Hibernate和Spring

现在已经有了POJO(Plain Old Java Object),写一个映射文件Hibernate就可以维护其持久性。

1. 在org/appfuse/model中新建一个名为User.hbm.xml文件，内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 2.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
<class name="org.appfuse.model.User" table="app_user">
  <id name="id" column="id" unsaved-value="0">
    <generator class="increment" />
  </id>
  <property name="firstName" column="first_name"
    not-null="true"/>
  <property name="lastName" column="last_name" not-null="true"/>
</class>
</hibernate-mapping>
```

2. 在web/WEB-INF/目录下的web/WEB-INF/ApplicationContext.xml文件中添加映射关系。打开文件，找到<property name="mappingResources">，修改成如下：

```
<property name="mappingResources">
  <list>
    <value>org/appfuse/model/User.hbm.xml</value>
  </list>
</property>
```

在ApplicationContext.xml文件中，你可以看到数据库是如何设置的，使用Spring时如何配置Hibernate的。Equinox预设置会使用一个名为db/appfuse的HSQL数据库。它将在你的Ant的db目录下创建，详细配置在“[How Spring Is Configured in Equinox](#)”一节中描述。

3. 运行ant deploy reload(Tomcat处于运行状态)，在Tomcat控制台的日志中可以看到数据表的创建过程。

```

INFO - SchemaExport.execute(98) | Running hbm2ddl schema export
INFO - SchemaExport.execute(117) | exporting generated schema to database
INFO - ConnectionProviderFactory.newConnectionProvider(53) | Initializing
connection provider:
org.springframework.orm.hibernate.LocalDataSourceConnectionProvider
INFO - DriverManagerDataSource.getConnectionFromDriverManager(140) | Creating
new JDBC connection to [jdbc:hsqldb:db/appfuse]
INFO - SchemaExport.execute(160) | schema export complete

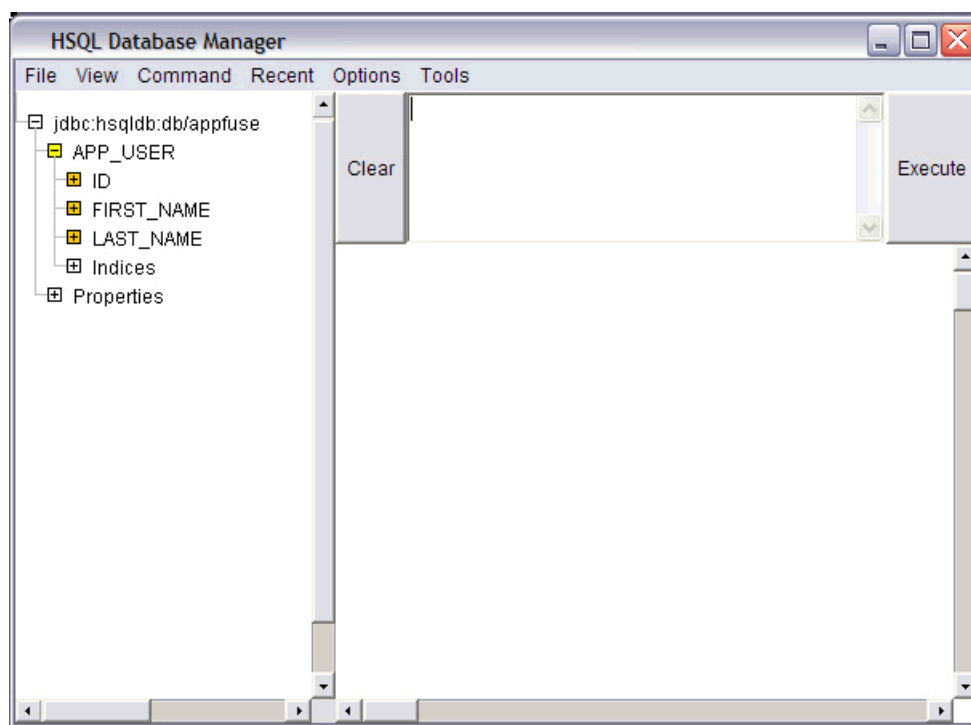
```

提示

如果你想看到更多或更少的日志，修改web/WEB-INF/classes/log4j.xml中log4j的设置。

4. 为了验证数据库是否已经建好，运行 `ant browser` 启动HSQL控制台。你会看到如下的HSQL Database Manager。

图 2.5. HSQL Database Manager



Equinox中Spring是如何配置的

任何基于J2EE的应用中使用Spring，配置都很简单。在最少的情况下，你只要简单的添加Spring的ContextLoaderListener到你的web.xml中。

```

<listener>
  <listener-class>

```

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

这是一个ServletContextListener，它会在启动web应用过程中进行初始化。默认情况下，它会查找web/WEB-INF/ApplicationContext.xml文件，你可以指定一个名为contextConfigLocation的<context-param>元素来进行更改，例如：

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

<param-value>元素可以是以空格或是逗号隔开的一系列路径。在Equinox中，Spring的配置使用了这个Listener和其默认的contextConfigLocation。

那么，Spring是如何知道Hibernate的存在？这就Spring的魅力所在，它让依赖性的绑定变得非常简单。请参阅ApplicationContext.xml的全部内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName">
      <value>org.hsqldb.jdbcDriver</value>
    </property>
    <property name="url">
      <value>jdbc:hsqldb:db/appfuse</value>
    </property>
    <property name="username"><value>sa</value></property>
    <!-- Make sure <value> tags are on same line - if they're not,
      authentication will fail -->
    <property name="password"><value></value></property>
  </bean>
  <!-- Hibernate SessionFactory -->
  <bean id="sessionFactory"
    class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
    <property name="dataSource">
      <ref local="dataSource"/>
    </property>
    <property name="mappingResources">
      <list>
        <value>org/appfuse/model/User.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.dialect">
```

```

        net.sf.hibernate.dialect.HSQLDialect
    </prop>
    <prop key="hibernate.hbm2ddl.auto">create</prop>
</props>
</property>
</bean>
<!-- Transaction manager for a single Hibernate SessionFactory (alternative
to JTA) -->
<bean id="transactionManager"
    class="org.springframework.orm.hibernate.HibernateTransactionManager">
    <property name="sessionFactory">
        <ref local="sessionFactory"/>
    </property>
</bean>
</beans>

```

第一个bean(dataSource)代表HSQL数据库，第2个bean(sessionFactory)依赖它。Spring仅仅是调用LocalSessionFactoryBean的setDataSource(DataSource)使之工作。如果你想用JNDI DataSource替换，可以bean的定义改成类似下面的几行：

```

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

```

同时请注意sessionFactory定义中的hibernate.hbm2ddl.auto属性。这个属性会在应用启动时自动创建数据表，可选的值还可以是update或create-drop。

最后一个配置的bean是transactionManager(你也可以使用JTA transaction)，它在处理跨越两个数据库的分布式的事务处理中必不可少。如果你想使用jta transaction manager，将此bean的class属性改成org.springframework.transaction.jta.JtaTransactionManager。

现在你可以用Hibernate实现UserDAO类。

用Hibernate实现UserDAO

要实现Hibernate UserDAO，需要完成以下几步：

1. 在目录src/org/appfuse/dao/hibernate下创建一个文件UserDAOHibernate.java，这个类继承了HibernateDaoSupport类，并实现了UserDAO接口。

```

package org.appfuse.dao.hibernate;
// use your IDE to handle imports

public class UserDAOHibernate extends HibernateDaoSupport implements UserDAO {
    private Log log = LogFactory.getLog(UserDAOHibernate.class);

```

```

public List getUsers() {
    return getHibernateTemplate().find("from User");
}

public User getUser(Long id) {
    return (User) getHibernateTemplate().get(User.class, id);
}

public void saveUser(User user) {
    getHibernateTemplate().saveOrUpdate(user);
    if (log.isDebugEnabled()) {
        log.debug("userId set to: " + user.getId());
    }
}

public void removeUser(Long id) {
    Object user = getHibernateTemplate().load(User.class, id);
    getHibernateTemplate().delete(user);
}
}

```

Spring的HibernateDaoSupport类是一个方便实现Hibernate DAO的超类，你可以利用其一些有用的方法，来获得Hibernate DAO或是SessionFactory。最方便的方法是getHibernateTemplate()，它返回一个HibernateTemplate对象。这个模板把检测式异常(checkedException)包装成运行时异常(runtime exception)，这使得你的DAO接口无需抛出Hibernate异常。

程序还没有把UserDAO绑定到UserDAOHibernate上，必须创建它们之间的关联。

2. 在Spring配置文件(web/WEB-INF/ApplicationContext.xml)中添加以下内容：

```

<bean id="userDAO" class="org.appfuse.dao.hibernate.UserDAOHibernate">
    <property name="sessionFactory">
        <ref local="sessionFactory"/>
    </property>
</bean>

```

这样就在你的UserDAOHibernate(从HibernateDaoSupport的setSessionFactory继承)中建了一个Hibernate Session Factory。Spring会检测一个Session(也就是，它在web层是开放的)是否已经存在，并且直接使用它，而不是新建一个。这样你可以使用Spring流行的“Open Session in View”模式来延迟载入collections。

进行单元测试，用DAO验证CRUD操作

在进行第一个测试之前，把你的日志级别从“INFO”调到“WARN”。

1. 把log4j.xml(在web/WEB-INF/classes目录下)中<level value="INFO"/>改为<level value="WARN"/>。

- 键入`ant test`来运行`UserDAOTest`。如果你有多个测试，你必须用`ant test -Dtestcase=UserDAOTest`来指定要运行的测试。运行之后，控制台中会出现一些测试的日志信息，如下所示。

图 2.6. 运行`ant test -Dtestcase=UserDAO`命令的结果

```

$ ant test -Dtestcase=UserDAO
Buildfile: build.xml

compile:

test:
[junit] DEBUG - UserDAOHibernate.saveUser(36) : userId set to: 1
[junit] INFO - UserDAOTest.testSaveUser(27) : org.appfuse.model.User@1a9d1b1f

[junit] id=1
[junit] firstName=Rod
[junit] lastName=Johnson
[junit] ]
[junit] DEBUG - UserDAOHibernate.saveUser(36) : userId set to: 2
[junit] DEBUG - UserDAOTest.testAddAndRemoveUser(42) : removing user...
[junit] TestSuite: org.appfuse.persistence.UserDAOTest
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0.469 sec

BUILD SUCCESSFUL
Total time: 4 seconds
$ _

```

创建Manager，声明事务处理

J2EE开发中强烈建议将各层进行分离。换言之，不要把数据层(DAO)和web层(servlets)混在一起。使用Spring很容易做到这一点，但使用“业务委派”(business delegate)模式，可以对这些层进一步分离。

使用业务委派模式的主要原因是：

- 大多数持久层组件执行一个业务逻辑单元，把逻辑放在一非web类中的最大好处是，web service或是胖客户端(rich platform client)可以像使用servlet一样来用同一API。
- 大多数业务逻辑都在同一方法中完成，当然可能多个DAO。使用业务委派，使得你可以在一个更高的业务委派层(level)使用Spring的声明式业务委派特性。

MyUsers应用中`UserManager`和`UserDAO`拥有相同的一个方法。主要不同的是Manager对于web更为友好(web-friendly)，它可以接受String，而UserDAO只能接受Long，并且它可以在`saveUser`方法中返回一个User对象。这在插入一个新用户比如，要获得主键，是非常方便的。Manager(或称为业务委派)中也可以添加一些应用中所需要的其它业务逻辑。

- 首先在`test/org/appfuse/service`(你必须先建好这个目录)中新建一个`UserManagerTest`类，开始创建“service”，这个类继承了JUnit的`TestCase`类，代码如下：

```

package org.appfuse.service;
// use your IDE to handle imports

public class UserManagerTest extends TestCase {
    private static Log log = LogFactory.getLog(UserManagerTest.class);
    private ApplicationContext ctx;

```

```

private User user;
private UserManager mgr;

protected void setUp() throws Exception {
    String[] paths = {"/WEB-INF/applicationContext.xml"};
    ctx = new ClassPathXmlApplicationContext(paths);
    mgr = (UserManager) ctx.getBean("userManager");
}

protected void tearDown() throws Exception {
    user = null;
    mgr = null;
}
// add testXXX methods here
}

```

在`setUp()`方法中，使用`ClassPathXmlApplicationContext`把`ApplicationContext.xml`载入变量`ApplicationContext`中。载入`ApplicationContext`有几种途径，从`classpath`中，文件系统，或`web`应用内。这些方法将在第三章(`BeanFactory`及其运行原理)中描述。

2. 编写第一个测试方法，来验证使用`UserManager`成功的完成添加和删除一个`User`对象操作。

```

public void testAddAndRemoveUser() throws Exception {
    user = new User();
    user.setFirstName("Easter");
    user.setLastName("Bunny");
    user = mgr.saveUser(user);
    assertNotNull(user.getId());
    if (log.isDebugEnabled()) {
        log.debug("removing user...");
    }
    String userId = user.getId().toString();
    mgr.removeUser(userId);
    user = mgr.getUser(userId);
    assertNull("User object found in database", user);
}

```

这个测试实际上是一个集成测试(integration test)，而不是单元测试(unit test)。为了更接近单元测试，可以使用`EasyMock`或是类似工具来“伪装”(fake) DAO。这样，就不必关心`ApplicationContext`和任何依赖Spring API的东西。建议创建这样的测试，因为它可以测试项目所有依赖(Hibernate, Spring自己的类)的内部构件，包括数据库。第9章，讨论重构`UserManagerTest`，使用mock处理DAO的依赖性。

3. 为了编译`UserManagerTest`，在`src/org/appfuse/service`中新建一个接口——`UserManager`。在`org.appfuse.service`包中创建这个类，代码如下：

```

package org.appfuse.service;
// use your IDE to handle imports

```

```
public interface UserManager {  
    public List getUsers();  
    public User getUser(String userId);  
    public User saveUser(User user);  
    public void removeUser(String userId);  
}
```

4. 建一个名为org.appfuse.service.impl的子包, 新建一个类,实现UserManager接口。

```
package org.appfuse.service.impl;  
// use your IDE to handle imports  
  
public class UserManagerImpl implements UserManager {  
    private static Log log = LogFactory.getLog(UserManagerImpl.class);  
    private UserDao dao;  
  
    public void setUserDAO(UserDao dao) {  
        this.dao = dao;  
    }  
  
    public List getUsers() {  
        return dao.getUsers();  
    }  
  
    public User getUser(String userId) {  
        User user = dao.getUser(Long.valueOf(userId));  
        if (user == null) {  
            log.warn("UserId '" + userId + "' not found in database.");  
        }  
        return user;  
    }  
  
    public User saveUser(User user) {  
        dao.saveUser(user);  
        return user;  
    }  
  
    public void removeUser(String userId) {  
        dao.removeUser(Long.valueOf(userId));  
    }  
}
```

这个类看不出你在使用Hibernate。当你打算把持久层转向一种不同的技术时,这样做很重要。

这个类提供一个私有dao成员变量,还有setUserDAO()方法。这样能够让Spring能够表演“依赖性绑定”魔术(perform “dependency binding” magic),把这些对象扎在一起。在使用mock重构这个类时,你必须在UserManager接口中添加setUserDAO()方法。

5. 在进行测试之前，配置Spring，以便使用`getBean("userManager")`返回一个`userManagerImpl`类。在`web/WEB-INF/ApplicationContext.xml`文件中，添加以下几行代码：

```
<bean id="userManager"
    class="org.appfuse.service.UserManagerImpl">
    <property name="userDAO"><ref local="userDAO"/></property>
</bean>
```

剩下的问题，你还没有使用Spring的AOP，特别是让声明式的事务处理发挥作用。

6. 为了实现这一点，使用`ProxyFactoryBean`代替`userManager`。`ProxyFactoryBean`可以创建一个类的不同的实现，这样AOP能够解释和覆盖方法调用。在事务处理中，使用`TransactionProxyFactoryBean`代替`UserManagerImpl`类。在`context`文件中添加下面`bean`的定义：

```
<bean id="userManager"
    class="org.springframework.transaction.interceptor.TransactionProxy
    FactoryBean">
    <property name="transactionManager">
        <ref local="transactionManager"/>
    </property>
    <property name="target">
        <ref local="userManagerTarget"/>
    </property>
    <property name="transactionAttributes">
        <props>
            <prop key="save*">PROPAGATION_REQUIRED</prop>
            <prop key="remove*">PROPAGATION_REQUIRED</prop>
            <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
        </props>
    </property>
</bean>
```

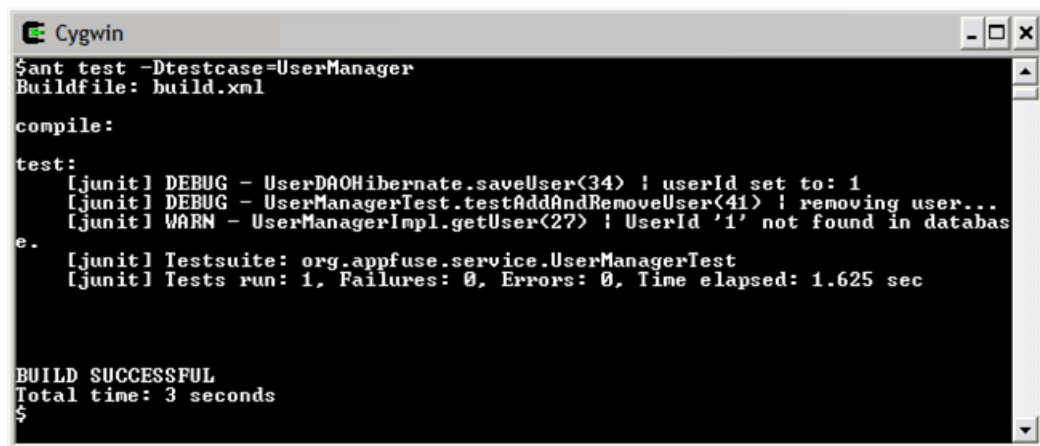
从这个xml代码片断中可以看出，`TransactionProxyFactoryBean`必须设置一个`transactionManager`属性，定义`transactionAttributes`。

7. 让事务处理代理(`Transaction Proxy`)知道你要模仿的对象：`userManagerTarget`。作为新`bean`的一部分，修改原来的`userManager` `bean`，使之拥有一个值为`userManagerTarget`的`id`属性。

```
<bean id="userManagerTarget"
    class="org.appfuse.service.UserManagerImpl">
    <property name="userDAO"><ref local="userDAO"/></property>
</bean>
```

编辑`applicationContext.xml`添加`userManager`和`userManagerTarget`的定义后，运行`ant test -Dtestcase=UserManager`，看看终端输出的结果：

图 2.7. 运行ant test -Dtestcase=UserManager命令的结果



```

Cygwin
$ant test -Dtestcase=UserManager
Buildfile: build.xml

compile:

test:
[junit] DEBUG - UserDaoHibernate.saveUser(34) : userId set to: 1
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(41) : removing user...
[junit] WARN - UserManagerImpl.getUser(27) : UserId '1' not found in database.
e.
[junit] Testsuite: org.appfuse.service.UserManagerTest
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 1.625 sec

BUILD SUCCESSFUL
Total time: 3 seconds
$

```

8. 如果你想看看事务处理的执行和提交情况，在log4j.xml中添加：

```

<logger name="org.springframework.transaction">
  <level value="DEBUG"/> <!-- INFO does nothing -->
</logger>

```

重新运行测试，将看到大量日志信息，如它相关的对象，事务的创建和提交等。测试完毕，最好删除上面的日志定义(logger)。

祝贺你！你已经实现了一个web应用的Spring/Hibernate后端解决方案。并且你已经用AOP和声明式业务处理配置好了业务委派。了不起，自我鼓励一下！（This is no small feat; give yourself a pat on the back!）

对Struts Action进行单元测试

现在业务委派和DAO都在起作用，我们看看MVC框架吸盘(sucker)的上部。哦！那里--不是吧！你应该进行C(Controller)，而不是V(View)。为了管理用户，创建一个Struts Action，继续进行测试驱动(Test-Driven)开发。

Equinox是为Struts配置的。配置Struts需要在web.xml中进行一些设置，并在web/WEB-INF下定义一个struts-config.xml文件。由于Struts开发人员比较多，这里先使用Struts。第4章用Spring进行处理。如果你想跳过这一节，直接学习Spring MVC方法，请参考第4章：Spring MVC框架。

在test/org/appfuse/web目录下新建一个文件UserActionTest.java，开发你的第一个Struts Action单元测试。这个类继承了MockStrutsTestCase，文件内容如下：

```

package org.appfuse.web;
// use your IDE to handle imports

public class UserActionTest extends MockStrutsTestCase {
  public UserActionTest(String testName) {
    super(testName);
  }
}

```

```
public void testExecute() {
    setRequestPathInfo("/user");
    addRequestParameter("id", "1");
    actionPerform();
    verifyForward("success");
    verifyNoActionErrors();
}
}
```

为web层创建Action和Model(DynaActionForm)

1. 在目录src/org/appfuse/web下新建一个类UserAction.java。这个类继承了DispatchAction，你可以花几分钟，在这个类中，创建CRUD方法。

```
package org.appfuse.web;
// use your IDE to handle imports

public class UserAction extends DispatchAction {
    private static Log log = LogFactory.getLog(UserAction.class);

    public ActionForward execute(ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
        request.getSession().setAttribute("test", "succeeded!");
        log.debug("looking up userId: " + request.getParameter("id"));
        return mapping.findForward("success");
    }
}
```

2. 配置Struts，使“/user”这个请求路径有意义。在web/WEB-INF/struts-config.xml中加入一个action-mapping。打开文件加入：

```
<action path="/user" type="org.appfuse.web.UserAction">
    <forward name="success" path="/index.jsp"/>
</action>
```

3. 执行命令ant test -Dtestcase=UserAction，你会看到友好的“BUILD SUCCESSFULLY”信息。
4. 在struts-config.xml中添加form-bean定义(在form-beans部分)。对于Struts ActionForm，使用DynaActionForm，这是一个javabean，可以从XML定义中动态的创建。

```
<form-bean name="userForm"
```

```
type="org.apache.struts.action.DynaActionForm">
  <form-property name="user" type="org.appfuse.model.User"/>
</form-bean>
```

这里使用这种表示方式，而没有使用了具体的ActionForm，因为你只需要一个User对象的瘦(thin)包装器。理想情况下，你可以User对象，但会失去Struts环境下的一些特性：验证属性(validate properties)，checkbox复位(reset checkboxes)。后面，将演示用Spring为何会更加简单，它可以让你在web层使用User对象。

5. 修改<action>定义，在request中使用这个form。

```
<action path="/user" type="org.appfuse.web.UserAction"
  name="userForm" scope="request">
  <forward name="success" path="/index.jsp"/>
</action>
```

6. 修改UserActionTest类，在Action中测试不同的CRUD方法。如下所示：

```
public class UserActionTest extends MockStrutsTestCase {

    public UserActionTest(String testName) {
        super(testName);
    }

    // Adding a new user is required between tests because HSQL creates
    // an in-memory database that goes away during tests.
    public void addUser() {
        setRequestPathInfo("/user");
        addRequestParameter("method", "save");
        addRequestParameter("user.firstName", "Juergen");
        addRequestParameter("user.lastName", "Hoeller");
        actionPerform();
        verifyForward("list");
        verifyNoActionErrors();
    }

    public void testAddAndEdit() {
        addUser();
        // edit newly added user
        addRequestParameter("method", "edit");
        addRequestParameter("id", "1");
        actionPerform();
        verifyForward("edit");
        verifyNoActionErrors();
    }

    public void testAddAndDelete() {
        addUser();
        // delete new user
```

```
    setRequestPathInfo("/user");
    addRequestParameter("method", "delete");
    addRequestParameter("user.id", "1");
    actionPerform();
    verifyForward("list");
    verifyNoActionErrors();
}

public void testList() {
    addUser();
    setRequestPathInfo("/user");
    addRequestParameter("method", "list");
    actionPerform();
    verifyForward("list");
    verifyNoActionErrors();
    List users = (List) getRequest().getAttribute("users");
    assertNotNull(users);
    assertTrue(users.size() == 1);
}
}
```

7. 修改UserAction, 这样测试程序才能通过, 并能处理(客户端)请求。最简单的方法是添加edit, save和delete方法, 请确保你事先已经删除了execute方法。下面是修改过的UserAction.java文件。

```
public class UserAction extends DispatchAction {
    private static Log log = LogFactory.getLog(UserAction.class);
    private UserManager mgr = null;

    public void setUserManager(UserManager userManager) {
        this.mgr = userManager;
    }

    public ActionForward delete(ActionMapping mapping, ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
        if (log.isDebugEnabled()) {
            log.debug("entering 'delete' method...");
        }
        mgr.removeUser(request.getParameter("user.id"));
        ActionMessages messages = new ActionMessages();
        messages.add(ActionMessages.GLOBAL_MESSAGE,
            new ActionMessage("user.deleted"));
        saveMessages(request, messages);
        return list(mapping, form, request, response);
    }

    public ActionForward edit(ActionMapping mapping, ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
```

```
throws Exception {
    if (log.isDebugEnabled()) {
        log.debug("entering 'edit' method...");
    }
    DynaActionForm userForm = (DynaActionForm) form;
    String userId = request.getParameter("id");
    // null userId indicates an add
    if (userId != null) {
        User user = mgr.getUser(userId);
        if (user == null) {
            ActionMessages errors = new ActionMessages();
            errors.add(ActionMessages.GLOBAL_MESSAGE,
                new ActionMessage("user.missing"));
            saveErrors(request, errors);
            return mapping.findForward("list");
        }
        userForm.set("user", user);
    }
    return mapping.findForward("edit");
}

public ActionForward list(ActionMapping mapping, ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception {
    if (log.isDebugEnabled()) {
        log.debug("entering 'list' method...");
    }
    request.setAttribute("users", mgr.getUsers());
    return mapping.findForward("list");
}

public ActionForward save(ActionMapping mapping, ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception {
    if (log.isDebugEnabled()) {
        log.debug("entering 'save' method...");
    }
    DynaActionForm userForm = (DynaActionForm) form;
    mgr.saveUser((User)userForm.get("user"));
    ActionMessages messages = new ActionMessages();
    messages.add(ActionMessages.GLOBAL_MESSAGE,
        new ActionMessage("user.saved"));
    saveMessages(request, messages);
    return list(mapping, form, request, response);
}
}
```

现在你已经修改了这个类的CRUD操作，继续以下步骤：

8. 修改struts-config.xml，使用ContextLoaderPlugin来配置Spring的UserManager设置。把下面内容添加到你的struts-config.xml中。

```
<plug-in
  className="org.springframework.web.struts.ContextLoaderPlugIn">
  <set-property property="contextConfigLocation"
    value="/WEB-INF/applicationContext.xml,
    /WEB-INF/action-servlet.xml"/>
</plug-in>
```

默认情况下这个插件会载入action-servlet.xml文件。要让Test Action能找到你的Manager，你还必须配置这个插件来载入ApplicationContext.xml文件。

注意

使用ContextLoaderPlugin是众多的Struts web层与Spring中间层集成的方法之一。其它的各种选择将在第11章：web框架集成中一一讲解。

9. 对每个使用Spring的Action，定义一个type="org.springframework.web.struts.DelegatingActionProxy"的action-mapping，为每个真实的Struts Action声明一个对应的Spring bean。这样修改一下你的action mapping 就能使用这个新类。
10. 修改Action mapping，使用DispatchAction。

为了让DispatchAction运行，在mapping中添加参数parameter="method"，它表示(在一个URL或是隐藏字段hidden field)要调用的方法，同时转向(forwards)edit和list forward(参考能进行CRUD操作的UserAction类)。

```
<action path="/user"
  type="org.springframework.web.struts.DelegatingActionProxy"
  name="userForm" scope="request" parameter="method">
  <forward name="list" path="/userList.jsp"/>
  <forward name="edit" path="/userForm.jsp"/>
</action>
```

确保web目录下已经建好userList.jsp和userForm.jsp两个文件。暂时不必在文件中写入内容。

11. 作为插件的一部分，配置Spring，以便能识别“ /user ” bean并把UserManager设置成它的属性。在web/WEB-INF/action-servlet.xml中添加以下定义。

```
<bean name="/user" class="org.appfuse.web.UserAction"
  singleton="false">
  <property name="userManager">
    <ref bean="userManager"/>
  </property>
</bean>
```

定义中，使用singleton="false"。这样就会为每个请求，新建一个Action，减少对线程安全Action的需求。既然你的Manager和DAO都没有成员变量，没有设置这一属性也不会出问题(默认singleton="true")。

12. 在message.properties资源绑定文件中配置信息(message)。

在UserAction类中，有一些对在完成操作时显示成功或错误信息引用。这些引用是指一个应用资源绑定文件(或messages.properties文件中)中各信息的键。这里是指：

- user.saved
- user.missing
- user.deleted

把这些键存入web/WEB-INF/classes下的messages.properties文件中。例如：

```
user.saved=User has been saved successfully.
user.missing=No user found with this id.
user.deleted=User successfully deleted.
```

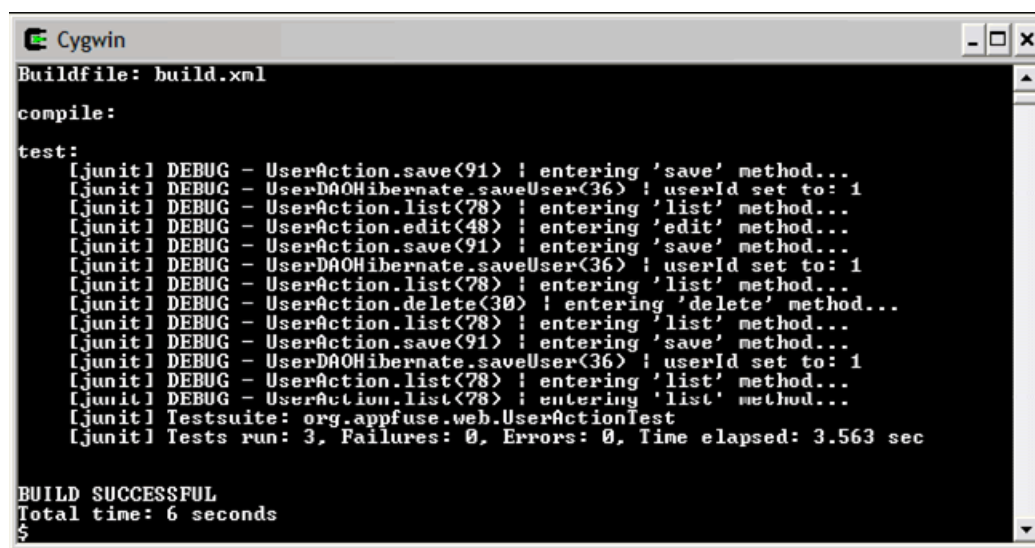
这个文件通过struts-config.xml中的<message-resources>元素进行加载。

```
<message-resources parameter="messages" />
```

运行单元测试，验证Action的CRUD操作

运行 `ant test -Dtestcase=UserAction`。输出结果如下：

图 2.8. 运行 `ant test -Dtestcase=UserAction` 命令的输出结果



```
Cygwin
Buildfile: build.xml

compile:

test:
[junit] DEBUG - UserAction.save(91) ! entering 'save' method...
[junit] DEBUG - UserDaoHibernate.saveUser(36) ! userId set to: 1
[junit] DEBUG - UserAction.list(78) ! entering 'list' method...
[junit] DEBUG - UserAction.edit(48) ! entering 'edit' method...
[junit] DEBUG - UserAction.save(91) ! entering 'save' method...
[junit] DEBUG - UserDaoHibernate.saveUser(36) ! userId set to: 1
[junit] DEBUG - UserAction.list(78) ! entering 'list' method...
[junit] DEBUG - UserAction.delete(30) ! entering 'delete' method...
[junit] DEBUG - UserAction.list(78) ! entering 'list' method...
[junit] DEBUG - UserAction.save(91) ! entering 'save' method...
[junit] DEBUG - UserDaoHibernate.saveUser(36) ! userId set to: 1
[junit] DEBUG - UserAction.list(78) ! entering 'list' method...
[junit] DEBUG - UserAction.list(78) ! entering 'list' method...
[junit] Testsuite: org.appfuse.web.UserActionTest
[junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 3.563 sec

BUILD SUCCESSFUL
Total time: 6 seconds
$
```


填充JSP文件，这样可以通过浏览器来进行CRUD操作

1. 在你的jsp文件(userFrom.jsp和userList.jsp)中添加以下代码，这样它们可以显示action处理的结果。如果还事先准备，在web目录下建一个文件userList.jsp。添加一些代码你就可以看到数据库中所有的用户资料。在下面代码中，第一行包含(include)了一个文件taglibs.jsp。这个文件包含了应用所有JSP Tag Library的声明。大部分是Struts Tag, JSTL和SiteMesh(用来美化JSP页面)。

```
<%@ include file="/taglibs.jsp"%>
<title>MyUsers ~ User List</title>
<button onclick="location.href='user.do?method=edit'">Add User</button>
<table class="list">
<thead>
<tr>
<th>User Id</th>
<th>First Name</th>
<th>Last Name</th>
</tr>
</thead>
<tbody>
<c:forEach var="user" items="${users}" varStatus="status">
<c:choose>
<c:when test="${status.count % 2 == 0}"><tr class="even"></c:when>
<c:otherwise><tr class="odd"></c:otherwise>
</c:choose>
<td><a href="user.do?method=edit&id=${user.id}">${user.id}</a></td>
<td>${user.firstName}</td>
<td>${user.lastName}</td>
</tr>
</c:forEach>
</tbody>
</table>
```

你可以看到有一行“标题头”(headings)(在<thead>中)。JSTL的<c:forEach>进行结果迭代，显示所有的用户。

2. 向数据库添加一些数据，你就会看到一些真实(actual)的用户(users)。你可以选择一种方法，手工添加，使用ant browse，或是在build.xml中添加如下的target:

```
<target name="populate">
<echo message="Loading sample data..." />
<sql driver="org.hsqldb.jdbcDriver"
url="jdbc:hsqldb:db/appfuse"
userid="sa" password="">
<classpath refid="classpath"/>
INSERT INTO app_user (id, first_name, last_name)
values (5, 'Julie', 'Raible');
INSERT INTO app_user (id, first_name, last_name)
```

```

        values (6, 'Abbie', 'Raible');
    </sql>
</target>

```

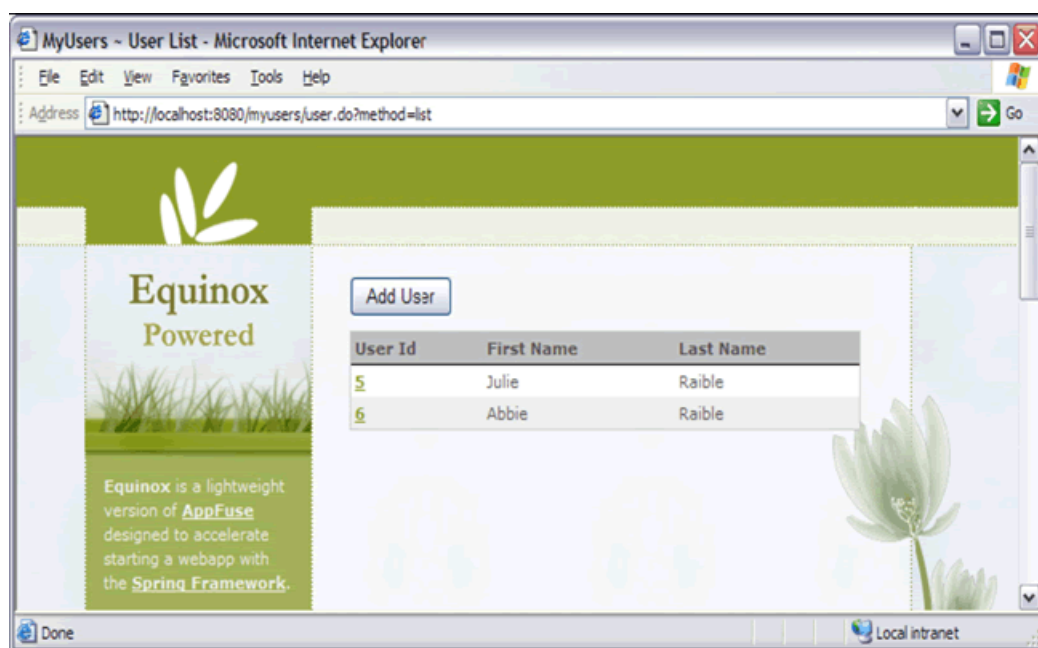
警告

为了使内置的HSQL正常工作，从能运行Ant的目录下启动Tomcat。在UNIX/Linux键入\$CATALINA_HOME/bin/startup.sh，在win上 %CATALINA_HOME%\bin\startup.bat。

通过浏览器验证JSP的功能

1. 有了这个JSP文件和里面的样例数据，就可以通过浏览器来查看这个页面。运行ant deploy reload，转到地址http://localhost:8080/myusers/user.do?method=list。出现以下画面：

图 2.9. 运行ant deploy reload命令的结果



2. 这个样例中，缺少国际化的页面标题头，和列标题头(column headings)。在 web/WEBINF/classes/messages.properties中加入一些键：

```

user.id=User Id
user.firstName=First Name
user.lastName=Last Name

```

修改过的国际化的标题头如下：

```
<thead>
```

```

<tr>
<th><bean:message key="user.id"/></th>
<th><bean:message key="user.firstName"/></th>
<th><bean:message key="user.lastName"/></th>
</tr>
</thead>

```

注意同样可以使用JSTL的<fmt:message key="...">标签。如果想为表添加排序和分布功能，可以使用Display Tag(<http://displaytag.sf.net>)。下面是使用这个标签的一个样例：

```

<display:table name="users" pagesize="10" styleClass="list"
requestURI="user.do?method=list">
<display:column property="id" paramId="id" paramProperty="id"
href="user.do?method=edit" sort="true"/>
<display:column property="firstName" sort="true"/>
<display:column property="lastName" sort="true"/>
</display:table>

```

请参考display tag文档中有关的列标题头国际化的部分。

3. 你已经建好了显示(list),创建form就可以添加/编辑(add/edit)数据。如果事先没有准备，可以在web目录下新建一个userForm.jsp文件。向文件中添加以下代码：

```

<%@ include file="/taglibs.jsp"%>
<title>MyUsers ~ User Details</title>
<p>Please fill in user's information below:</p>
<html:form action="/user" focus="user.firstName">
<input type="hidden" name="method" value="save"/>
<html:hidden property="user.id"/>
<table>
<tr>
<th><bean:message key="user.firstName"/>: </th>
<td><html:text property="user.firstName"/></td>
</tr>
<tr>
<th><bean:message key="user.lastName"/>: </th>
<td><html:text property="user.lastName"/></td>
</tr>
<tr>
<td></td>
<td>
<html:submit styleClass="button">Save</html:submit>
<c:if test="${not empty param.id}">
<html:submit styleClass="button"
onclick="this.form.method.value='delete'">
Delete</html:submit>
</c:if>
</td>
</tr>

```

```
</table>
</html:form>
```

注意

如果你正在开发一个国际化的应用，把上面的信息和按钮标签替换成`<bean:message>`或是`<fmt:message>`标签。这是一个很好的练习。对于信息message，建议把key名称写成`pageName.message`(例如：`userForm.message`)的形式，按钮名字写成`"button.name"`(例如`button.save`)。

4. 运行`ant deploy`，通过浏览器页面的user form来进行CRUD操作。

最后大部分web应用都需要验证。下一节中，配置Struts Validator，要求用户的last name是必填的。

用Commons Validator添加验证

为了在Struts中使用验证，执行以下几步：

1. 在`struts-config.xml`中添加`ValidatorPlugin`。
2. 创建`validation.xml`，指定`lastName`为必填字段。
3. 把`DynaActionForm`改用`DynaValidatorForm`。
4. 仅为`save()`方法设置验证(validation)。
5. 在`message.properties`中添加validation errors。

在struts-config.xml中添加ValidatorPlugin

配置Validator plugins,添加以下片断到`struts-config.xml`(紧接着Spring plugin):

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
  <set-property property="pathnames"
    value="/WEB-INF/validator-rules.xml,
    /WEB-INF/validation.xml"/>
</plug-in>
```

从这里你可以看出，Validator会查找WEB-INF目录下的两个文件`validator-rules.xml`和`validation.xml`。第一个文件，`validator-rules.xml`，是一个标准文件，作为Struts的一部分发布，它定义了所有可用的验证器(validators)，功能和客户端的JavaScript类似。第二个文件，包含针对每个form的验证规则。

创建validation.xml，指定lastName为必填字段

`validation.xml`文件中包含很多DTD定义的标准元素。但你只需要如下所示的`<form>`和`<field>`，更多信息请参阅Validator的文档。在`web/WEB-INF/validation.xml`中的`<form-validation>`标签之间添加`<formset>`元素。

```
<formset>
  <form name="userForm">
    <field property="user.lastName" depends="required">
      <arg0 key="user.lastName"/>
    </field>
  </form>
</formset>
```

把 DynaActionForm 改为 DynaValidatorForm

把struts-config.xml中的DynaActionForm改为DynaValidatorForm。

```
<form-bean name="userForm"
type="org.apache.struts.validator.DynaValidatorForm">
...

```

为save()方法设置验证(validation)

使用Struts的DispatchAction弊端是，验证会在映射层(mapping level)激活。为了在list和edit页面关闭验证。你必须单独建一个"validate=false"的映射。例如，AppFuse的UserAction有两个映射："/editUser"和"/listUser"。然而有一个更简单的方法，可以减少xml，只是多了一些java代码。

1. 在/user映射中，添加validate="false"。
2. 修改UserAction中的save()方法，调用form.validate()方法，如果发现错误，返回编辑页面。

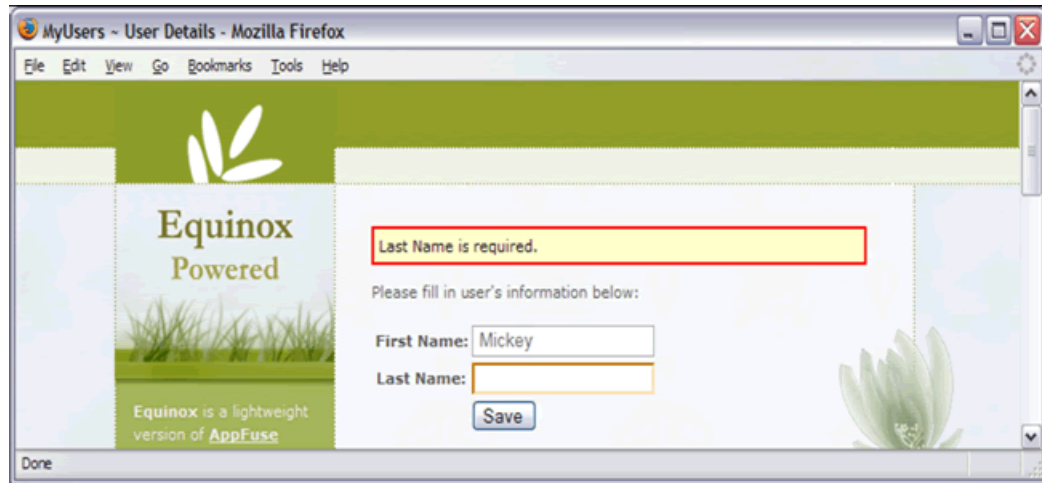
```
if (log.isDebugEnabled()) {
  log.debug("entering 'save' method...");
}
// run validation rules on this form
ActionMessages errors = form.validate(mapping, request);
if (!errors.isEmpty()) {
  saveErrors(request, errors);
  return mapping.findForward("edit");
}
DynaActionForm userForm = (DynaActionForm) form;
```

当DispatchAction运行时，与附带一个属性的两个映射相比，这样更加简洁。但用两个映射也有一些优点：

- 验证失败时，可以指定转向"input"属性。
- 在映射中可以添加"role"属性，可以指定谁有访问权限。例如，任何人都可以看到编辑(edit)页面，但只有管理员可以保存(save)。

3. 运行`ant deploy`重新载入(reload), 尝试添加一个新用户, 不要填写`lastName`。你会看到一个验证错误, 表明`lastName`是必填字段, 如下所示:

图 2.10. 运行`ant deploy`命令的结果



Struts Validator的另一种比较好的特性是客户端验证(client-side validation)。

4. 在form标签(`web/userForm.jsp`中)中添加`onsubmit` "属性, 在form末尾添加`<html:javascript>`标签。

```
<html:form action="/user" focus="user.firstName"
onsubmit="return validateUserForm(this)">
...
</html:form>
<html:javascript formName="userForm" />
```

现在如果运行`ant deploy`, 试图保存一个`lastName`为空的用户, 会弹出一个JavaScript提示: "Last Name is required"。这里有一个问题, 这个带JavaScript的form把validator的JavaScript功能都载入了页面。更好的方法是, 从外部文件导入JavaScript。参见第5章。

恭喜你! 你已经开发一个web应用程序, 它包含数据库交互, 验证实现, 成功信息和错误信息的显示。第4章, 将会把这个应用转向使用Spring框架。第5章中, 会添加异常处理, 文件上传, 邮件发送等特性。第6章会看一下JSP的替代技术, 在第7章, 会探讨DAO的不同实现, 包括iBATIS, JDO和Spring的JDBC。

本章小结

Spring是一个不错的框架, 它大大减少了编程的代码量。如果你看下本教程中的一些步骤, 大部分是进行设置和为Struts编写代码。Spring使DAO和Manager的实现变得非常容易。它也可以把很多Hibernate代码调用减少为一行代码, 并且允许删除那些有时感到乏味的异常处理。实际上, 我写本章(和MyUsers应用程序)的大部分时间花都在配置Struts上。

我基于两个原因使用Struts MVC框架来写本章。第一, 大部分人已经熟悉这种框架, 解释Struts-Spring的迁移比JSP/Servlet-Spring的迁移要容易得多。第二, 我想向你说明, 用Struts写MVC层有点麻烦。在第4章中, 你会用Spring MVC框架重构web层。我想到时你会发现它多么简单和直观。

第 3 章 BeanFactory及其工作原理

bean定义， BeanFactory及ApplicationContext简介

BeanFactory扮演着Spring的核心角色，所以弄清它怎么运行是很重要的。本章将阐述如何书写bean定义，它们的属性，依赖关系，及自动绑定。也解释了单态bean与其原形之间的逻辑关系。然后剖析了控制反转(IoC)，它的工作原理，以及它能带来哪些便利。这一章也诠释了BeanFactory中一个bean的生命周期来演示它的工作原理。这一章还分析了在第2章MyUsers中创建的applicationContext.xml文件。

Spring是一个出色的工具，让IoC与你的应用程序很好的集成。它使用一个BeanFactory来按管理和配置各种bean。然而，在大多数情况下，你根本不会与BeanFactory打交道，你将使用ApplicationContext，它拥有更多企业级的J2EE特性，如国际化，自定义转换器(用于把String转换成Object类型)，及事件的发布/通知机制。本章会解释BeanFactory如何运作，IoC与BeanFactory相关联，如何为BeanFactory配置bean，及如何使用ApplicationContext。

关于BeanFactory

BeanFactory实际是一个可以配置和管理任何Java类的内部接口。 XmlBeanFactory
[<http://www.springframework.org/docs/api/org/springframework/beans/factory/xml/XmlBeanFactory.html>]
从xml文件中读取bean定义，而 ListableBeansFactory
[<http://www.springframework.org/docs/api/org/springframework/beans/factory/ListableBeanFactory.html>]
是从属性文件中读取定义。当创建了一个BeanFactory时，Spring就会验证每个bean的配置。然而，每个bean的属性直到完成bean创建完成时才进行设置。单态bean是在BeanFactory加载时已经创建好了，其它bean则是根据需要进行创建。根据BeanFactory的Javadoc，"对采用哪种方式(LDAP，RDBMS，XML，属性文件等)存储定义，没有任何限制"，在写本书，可用的只有XML和properties文件两种方式。既然XmlBeanFactory是配置J2EE最常用的方法，本章所有的例子都使用XML。

注意

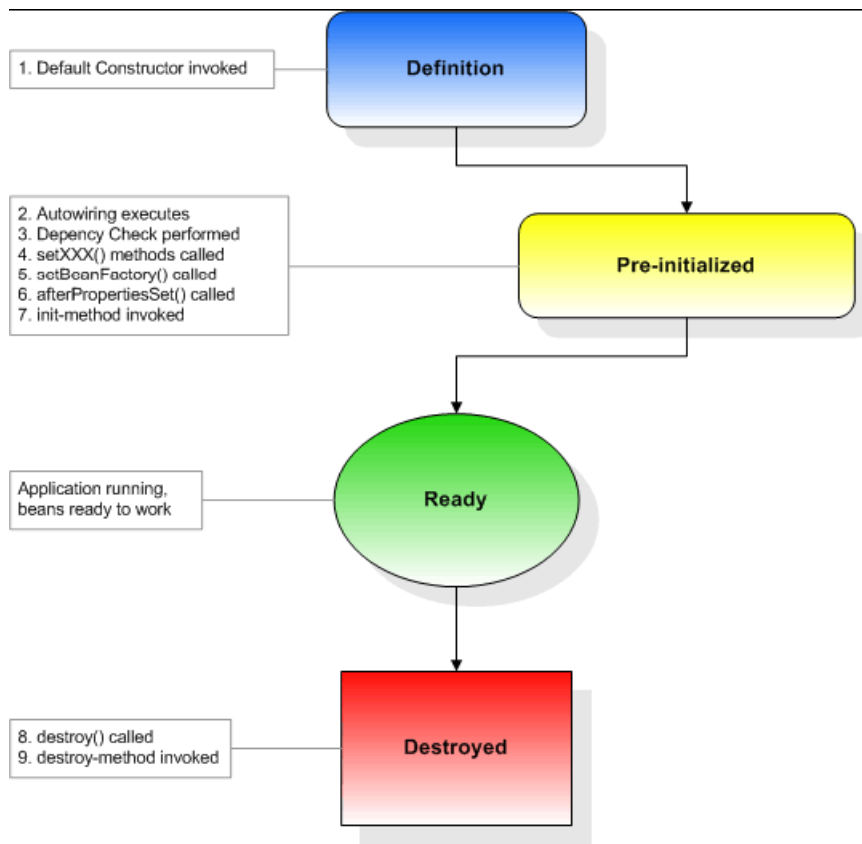
从属性文件中读取bean定义，你还可以使用 PropertiesBeanDefinitionReader
[<http://www.springframework.org/docs/api/org/springframework/beans/factory/support/PropertiesBeanDefinitionReader.html>]
有关使用这个类的更多信息，请参考R.J.Lorimer的文章Alternative Spring Bean Mappings
[<http://www.coffee-bytes.com/servlet/ShowEntryForId?id=56>]。

BeanFactory是一个超负荷的机器，负担着初始化各种bean和调用它们生命周期方法的重任。有一点要注意的是，大多数生命周期方法只应用于单态bean。Spring不能管理原型(非单态)bean的生命周期。正因为如此，当它们创建之后，无法传递给client，容器也不能进行跟踪。对于原型bean来说，Spring只是旧瓶装新酒罢了。

BeanFactory中一个bean的生命周期

图3.1图解一个bean的生命周期。某种外部力量控制着这个bean，那就是IoC容器。IoC容器定义了bean的操作规则。这些规则就是bean定义。bean通过它的依赖关系预先进行了初始化。当各个bean都准备就绪时，它进入准备状态。最后，IoC容器撤销这个bean。

图 3.1. BeanFactory中一个bean的生命周期



反转控制

IoC在应用开发中是一个非常有力的概念。如Martin Fowler所述 [http://www.martinfowler.com/articles/injection.html], IoC的一种表现形式就是依赖性注射。依赖性注射用的是好莱坞原则, "不要找我, 我会找你的。"。换句话说, 你的类不会去查找或是实例化它们所依赖的类。控制恰好是反过来的, 某种容器会设置这种依赖关系。使用IoC常常使代码更加简洁, 并且为相互依赖的类提供一种很好的方法。依赖性注射存在三种方式:

- 基于setter的(setter-based): 这些类是特指那些拥有一个无参数的构造器, 在绑定依赖关系时提供setter给IoC容器使用的javabean。这是Spring推荐使用的形式。当Spring支持基于构造器的注射时, 大量的构造器带的参数就很难管理。
- 基于构造器的(Constructor-based): 这种类拥有带有多个参数的构造器。IoC容器会查找和激活那些基于大量参数和对象类型的构造器。这种方法保证了一个bean不是在一个非法状态下创建的。
- 基于getter的(Getter-based)或称为方法注射: 与基于setter的方式类似, 不同的是为你的bean添加一个getter方法。当它在内部运行时, IoC容器会重载这个方法, 但在测试时你很容易指定你要用的getter方法。这种方法只是在最近被提及到。你可以从TheServerSide.com上找到更多的信息 [http://www.theserverside.com/news/thread.tss?thread_id=26205]。

下面是一个没有预设IoC一个类的例子。它是一个叫做listUsers的Struts Action, 它依赖一个UserDAO类, 而这个类需要一个Connection作为构造器的一部分。


```
public class ListUsers extends Action {

    public ActionForward execute(ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
        // get a connection from the database
        Connection conn = DatabaseUtils.getConnection();
        UserDAO dao = DAOFactory.createUserDAO("hibernate", conn);
        List users = dao getUsers();
        DatabaseUtils.closeConnection(conn);
        return mapping.findForward("success");
    }
}
```

上面的代码设计比较难看。通过实现一个预设IoC的类把代码清理一下。

```
public class ListUsers extends Action {
    private UserDAO dao;

    public void setUserDAO(UserDAO userDAO) {
        this.dao = userDAO;
    }

    public ActionForward execute(ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
        List users = dao getUsers();
        return mapping.findForward("success");
    }
}
```

上面的类没有包含任何查找(lookup)的代码。这样代码看起来更加清晰，并有利于测试。例如，在你的测试中，你可以使用针对DAO使用一个Mock Object来消除数据层的各依赖关系。一个好的IoC容器应该允许开发人员为UserDAO绑定一个连接(connection)(或是DataSource)。使用Spring你可以选择，把连接绑定到数据层，还是使用一个过滤器为每个请求打开一个新的连接。

暴露bean定义

一个bean定义或是<bean>实际上很简单，如下面的例子所示。

```
<bean id="example" class="org.appfuse.util.Converter"/>
```

一个bean至少需要一个id(或者是name)属性和一个class属性(attribute)。如果你不愿意,你根本不用为bean设置任何属性值(property),和"新建(new)"一个对象类似。大多数bean定义包含一些属性设置,除非你想简单的绑定接口和实现(模拟Factory Pattern)。

bean第一个必须的属性是id。各其它规范的XML文档一样,bean定义允许的属性和子元素是由DTD(文档定义类型)来规定的。Spring的DTD文件叫作spring-beans.dtd,并可以在其站点[<http://www.springframework.org/dtd/spring-beans.dtd>]上找到。id属性是一个真正的XML ID,这就意味在一个XML文件中要保证它是唯一的。要想给bean添加别名(alias)或是想在bean id中使用非法的XML字符,可以指定name属性。这个属性允许使用一个或多个id,用逗号或分号隔开。在配置bean时,使用一个id属性是推荐和首选的方法。

第2个必须的属性是class。Spring几乎可以管理任何Java类,最常用的方式是使用一个默认(空)的构造器,带有所依赖的属性(property)的setter方法。这个属性要指定的值必须是一个完整的类名(包名+类名)。另外你还可以使用parent属性,这个属性在你想复制一个类,重载它的属性(property)时可能有用。

下表列出了所有可以在bean元素中定义的属性。

表 3.1. 由spring-beans.dtd文件定义的bean定义属性

属性	描述	使用率
id	此XMLID元素能提供引用核对。要使用一个非法字符作为名id称的话，请使用可选的name属性。如果你没有指定id和name，Spring会把class值当id用。	高
name	使用这个属性创建一个或多个非法的id的别名。多个别名可以用name逗号或是空格隔开。利用这个属性，你可以通过ContextLoaderPlugin使用Struts和Spring来管理你的Action。	中
class	class和parent属性可以互换。一个bean定义必须指定此类的class parent全名(包名+类名)，或是父类的名称。注意：一个引用了父类的子bean可以重载它的singleton属性的属性值。它会继承父类的其它所有的特性，例如lazy-init和autowire。	高
parent		低
singleton	这个属性决定了这个bean是一个singleton(单态)bean(所有singleton的bean，都是通过调用getBean(id)返回一个共享的实例)还是一个prototype原型bean(每个bean通过调用getBean(id)返回一个独立的实例)。默认值为true。	低
abstract	如果为true，bean factory不会初始化这个bean。只在为某个具体的子类定义父类时使用。默认为false。	低
lazy-init	如果为true，则会延迟初始化。低若为false，就会在启动时，由lazy-init执行singleton初始化的bean factory进行实例化。	低
autowire	这个属性控制着bean属性的自动绑定。如果使用的話，Spring会自动使用以下的一种模式自动解决依赖关系：no:你必须在XML文件中使用<ref>定义bean的引用。这样使文档更清楚，推荐使用。byName:通过属性名称绑定。如果一个DAO暴露了一个dataSource属性，Spring会在当前的factory中把这个属性设置成此dataSource的值。byType:仅当一个属性类型的bean在此factory中时进行绑定。constructor:同byType，针对构造器参数。autodetect:通过对bean类的检测，选择是constructor还是byType。注意：此属性减少了XML文件的体积的同时，也降低了可读性和由声明属性提供的自身文件。对于大型应用，autowire不鼓励使用，因为它从协调类中移除了透明度和结构。	低
dependency-check	此属性检测所有bean的依赖关系(在属性中表示的)是否满足。None:不检测(默认)。没有指定值的属性不会设置。simple:检测类型依赖，包括原生类和String。object:检测factory中的其它对象。all:包括上述两种情况。	低
depends-on	此属性指定此bean初始化所依赖的各个bean。bean factory保证了这些bean优先初始化。	低
init-method	这是一个在设置了bean的属性后要调用的一个不带参数的方法。	低

destroy-method	当factory关闭时，调用这个不带参数的方法。	低
----------------	--------------------------	---

配置域属性和依赖关系

BeanFactory可以和EJB的生命周相比，只不过Spring要简单一些；你可以把任何类装配起来，不必从其它类继承或是实现接口。使用皮包骨头的EJB，你必须实现很多生命周期类，有的可能你根本就用不上。使用Spring管理的bean，你可以使用init-method和destory-method两个属性来管理生命周期。如果你想在初始化过程中与BeanFactory进行对话，你只需要实现接口。

一个bean的域属性是一个类的成员变量。例如你的bean可能有一个maxSize属性，和一个方法设置它。

```
private int maxSize;
public void setMaxSize(int maxSize) {
    this.maxSize = maxSize;
}
```

你可以在你的bean定义中使用以下XML片断来设置maxSize的属性值。

```
<property name="maxSize"><value>1000</value></property>
```

一个bean的依赖关系是指这个进行操作时所依赖的一种属性。依赖指向了其它的类，而不是简单的数值。例如，下面例子中的dataSource属性引用了另一个bean。这个dataSource是BeanFactory的一个依赖(属性)，而mappingResources仅仅是一个赋值的属性。

```
<bean id="sessionFactory" class="...">
  <property name="dataSource">
    <ref local="dataSource"/>
  </property>
  <property name="mappingResources">
    <list>
      <value>org/appfuse/model/User.hbm.xml</value>
    </list>
  </property>
  ...
</bean>
```

在这个例子中，指向dataSource的引用使用了一个<ref>标签。我们进一步看一下这个标记。

用<ref>指定依赖关系

指向dataSource的<ref>使用一个locle属性指向一个dataSource bean。除local外，还有其它几种指定所依赖的bean的途径。下面的清单展示了<ref>元素可用的属性。

- bean:从同一个XML中或另一个已经导入到ApplicationContext中的XML文件查找所依赖的bean。

- local:从另一个XML文件中查找依赖的bean。此属性是一个XML IDREF，所以它必须存在，否则验证失败。
- external:从当前XML文件中查找依赖的bean，而不当前XML文件中进行搜索。

从上面的列表可以看出，`<ref bean="..." />`和`<ref local="..." />`用法相似。bean是最灵活的选项，允许你在文件之间移动bean，而local有内置XML验证的方便性。

除了从String中指定值外，你还可以用 `PropertyPlaceholderConfigurer` [<http://www.springframework.org/docs/api/org.springframework.beans.factory.config.PropertyPlaceholderConfigurer.html>] 从一个.properties文件中读取值。下面是一个在classpath中使用database.properties的例子。

```
<bean id="propertyConfigurer"
      class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="location">
    <value>classpath:database.properties</value>
  </property>
</bean>
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName">
    <value>${db.driverName}</value>
  </property>
  <property name="url"><value>${db.url}</value></property>
  <property name="username"><value>${db.username}</value></property>
  <property name="password"><value>${db.password}</value></property>
</bean>
```

你可以用一个特殊的`<null/>`元素来指定一个属性为java null值。一个空的字符，`<value></value>`将会得到一个空的字符串("")。

预先初始化bean

要预初始化bean就是在你的应用程序中准备启用它们。你可以用通过配置属性和依赖关系达到目的。以下几节讨论这一过程的几个最关键的步骤。

Autowiring

如下面提示所示，虽然在大型的应用中不能推荐使用autowiring，但小一些的应用，你仍可以选用它。如果你选择自动绑定一个bean，我建议你使用`autowire="byName"`，因为它能保持让bean name和bean id同步。

例如，在第2章中你可以用autowiring定义UserDAO，那么你不必指定一个sessionFactory属性。

```
<bean id="userDAO" autowire="byName"
      class="org.appfuse.dao.hibernate.UserDAOHibernate"/>
```

如果你用`autowire="byType"`，Spring就会查找一个为Hibernate SessionFactory的bean。这种方法的问题是你可能成倍的增加了与两个数据库交互的session factory。使用`byName`的话，你可以给那些bean取不同的名称，并相应的修改你的setter。`constructor`和`autodetect`在底层使用了`byType`，所以会面临同样的问题。

注意

使用`autowiring`你的xml的XML失去了自我证明的特性。不使用`autowiring`的话，你可以知道一个bean的依赖关系，因为在XML中指明了。使用`autowiring`，可能不得不去看一个类的javadoc或源代码去查找。还有，虽然很少见，BeanFactory可以自动绑定一些你并不想设置的依赖关系。

依赖检测

当你想要保证一个bean的所有的属性都设置正确时，在你的bean的定义中定义一个`dependencycheck`非常有用。一个良好结构的bean都有默认值。在某些特定的场合，一些属性不是必需的。默认值为`none`，意味着依赖性检测没有激活，但你可以为每个bean启用它。`"simple"`值会验证原生类型，设置`collections`。`object`会检测一个bean的依赖关系(也叫`collaborator`)。`all`包括`simple`和`object`。

setXXX()

`setXXX()`方法是简单的setter，将依赖关系注射到一个类中。这些属性可以是配置在context文件中，可以是原生类型(也就是`int`，或是`boolean`)，对象类型(`Long,Integer`)，`null`值或是其它对象的引用。

为了演示每种类型是如何设置的，下面的Smorgasbord类包含所有前面的提到的属性类型。

```
package org.appfuse.model;
// organize imports with your IDE

public class Smorgasbord extends BaseObject {
    private Log log = LogFactory.getLog(Smorgasbord.class);
    private int daysToJavaOne;
    private boolean attendingJavaOne;
    private Integer streetsInDenver;
    private Long peopleInDenver;
    private DataSource dataSource;

    public void setDaysToJavaOne(int daysToJavaOne) {
        this.daysToJavaOne = daysToJavaOne;
    }

    public void setAttendingJavaOne(boolean attendingJavaOne) {
        this.attendingJavaOne = attendingJavaOne;
    }

    public void setStreetsInDenver(Integer streetsInDenver) {
        this.streetsInDenver = streetsInDenver;
    }

    public void setPeopleInDenver(Long peopleInDenver) {
        this.peopleInDenver = peopleInDenver;
    }
}
```

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

public String toString() {
    log.debug(super.toString());
    return super.toString();
}
}
```

不需要为这个类写单元测试，只要在bean的定义中设置dependency-check属性，也可以设置init-method来调用toString方法。

注意

对于专业的应用程序，我强烈建议总是写一个单元测试。

```
<bean id="smorgasbord" class="org.appfuse.model.Smorgasbord"
    dependency-check="all" init-method="toString">
    <property name="daysToJavaOne"><value>14</value></property>
    <property name="attendingJavaOne"><value>true</value></property>
    <property name="streetsInDenver"><value>334</value></property>
    <property name="peopleInDenver"><value>3000000</value></property>
    <property name="dataSource"><ref local="dataSource"/></property>
</bean>
```

将下面的部分添加到web/WEB-INF/classes/log4j.xml中，

你可以为包

org.springframework.beans启用信息日志。

```
<logger name="org.springframework.beans">
    <level value="INFO"/>
</logger>
```

现在如果你进行任何测试，或者是部署和运行MyUsers程序，你就可以在控制台看到以下输出。本例使用了ant test -Dtestcase=UserDAO。

图 3.2. 运行ant test -Dtestcase=UserDAO的结果

```

Cygwin
$ant test -Dtestcase=UserDAO
Buildfile: build.xml

compile:
[javac] Compiling 1 source file to C:\Source\myusers\build\classes

test:
[mkdir] Created dir: C:\Source\myusers\build\test\data
[junit] INFO - XmlBeanDefinitionReader.loadBeanDefinitions(118) : Loading XM
L bean definitions from class path resource [WEB-INF\applicationContext.xml]
[junit] INFO - DefaultListableBeanFactory.preInstantiateSingletons(168) : Pr
e-instantiating singletons in factory [org.springframework.beans.factory.support
.DefaultListableBeanFactory defining beans [smorgasbord,dataSource,sessionFacto
ry,transactionManager,userDAO,userManagerTarget,userManager]; Root of BeanFactory
hierarchy]
[junit] INFO - AbstractBeanFactory.getBean(158) : Creating shared instance o
f singleton bean 'smorgasbord'
[junit] INFO - AbstractBeanFactory.getBean(158) : Creating shared instance o
f singleton bean 'dataSource'
[junit] DEBUG - Smorgasbord.toString(38) : org.appfuse.model.Smorgasbord@89c
file[
[junit] log=org.apache.commons.logging.impl.Log4JLogger@62937c
[junit] daysToJavaOne=14
[junit] attendingJavaOne=true
[junit] streetsInDenver=334
[junit] peopleInDenver=3000000
[junit] dataSource=org.springframework.jdbc.datasource.DriverManagerDataSo
urce@a17083
[junit] ]

```

setBeanFactory()

在初始化方法调用后，BeanFactory检查那些实现

BeanFactoryAware

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

和

BeanNameAware

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

接口的类。这些接口为各bean找出其来源和自身的更多信息提供了一种途径。BeanFactoryAware定义一种方法。

```

public void setBeanFactory(BeansFactory beanFactory)
throws BeansException;

```

如果一个bean实现了这个接口，它就会引用BeanFactory，你可以通过它来查找其它bean。这基本上可以证明一个bean的来源。

为了让一个bean能找到它的id，可以使用BeanNameAware接口。此接口有一个唯一的方法。

```

public void setBeanName(java.lang.String name);

```

在大多数情况下，你不必访问BeanFactory，因为其它设置成依赖关系(用<ref bean="..." />)来访问它们。如果你想把同一个类配置成拥有不同依赖实现的两种不同的bean，访问这个bean的名称比较有效。用这种方法，你可以执行那些基于name配置的有条件的逻辑(bean)。

从BeanFactoryAware和BeanNameAware接口调用方法，bean进行一个就绪状态。这时，你的应用程序完全启动起来了(如，在Tomcat中)。

afterPropertiesSet()

为了进行初始化后的处理，你可以两种方法之一来配置你的bean：1)如上面例子中演示的一样，使用init-method属性。2)实现 InitializingBean

[<http://www.springframework.org/docs/api/org.springframework.beans.factory.InitializingBean.html>]及其`afterPropertiesSet()`方法。(图中显示了两方法，但只需要其中一种。)很明显，使用`init-method`是一种更为简洁的方法。然而，当你没有用Spring管理你的bean时，实现`InitializingBean`接口更有助于测试。例如，你可以在测试中调用这个方法，来验证你的mock object是否设置正确。这也有利于保证你的bean配置完全正确。你不必指望别人写的bean定义正确无误。

前面的例子把域属性值注射到`Smorgasbord`类中。它设置了原生类，对象，以及此factory中对另一bean的引用。Spring DTD不仅支持属性中使用简单的`<value>`元素，也支持设置`Properties`，`List`和`Map`。下面是一个使用复杂属性的例子(来自Spring文档 [<http://www.springframework.org/docs/reference/beans.html#beans-factory-properties-detailed>])。

```
<!-- results in a setPeople(java.util.Properties) call -->
<property name="people">
  <props>
    <prop key="HarryPotter">The magic property</prop>
    <prop key="JerrySeinfeld">The funny property</prop>
  </props>
</property>
<!-- results in a setSomeList(java.util.List) call -->
<property name="someList">
  <list>
    <value>a list element followed by a reference</value>
    <ref bean="dataSource"/>
  </list>
</property>
<!-- results in a setSomeMap(java.util.Map) call -->
<property name="someMap">
  <map>
    <entry key="yup an entry">
      <value>just some string</value>
    </entry>
    <entry key="yup a ref">
      <ref bean="dataSource"/>
    </entry>
  </map>
</property>
```

init-method

当一个bean中所有的属性设置好之后，bean定义中`init-method`方法会调用一个方法。这与实现`InitializingBean`接口等效，只不过后者不依附Spring。

就绪状态

当你的bean已经预初始化，所有设置的方法都已经被调用，这些bean进行一个就绪状态。就绪状态意味着，你的应用程序可以按需获取和使用这些bean。进行就绪状态的整个生命周期非常短。基于你的应用中bean的数目它会有所变化，然而，它仅仅是在启动时出现。

销毁bean

当你关闭(或是重启)你的应用时, singleton bean重新获得调用它们的生命周期方法。首先, 实现 DisposableBean的bean会调用它们的destroy()方法。其次, 在bean定义中指定destroy-method的bean会调用此方法。

ApplicationContext: 与bean交互

用Spring开发时理解BeanFactory很重要, 但是在你的应用程序中你几乎不必和它打交道。大多数情况下, 你会使用ApplicationContext, 它添加了一些企业级的J2EE特性, 如国际化(i18n), 自定义转换器(用于把转换成类型)及事件的发布/通知。一个ApplicationContext由bean定义文件实例, 可以简单的使用context.getBean("beanId")来获取bean。实例化ApplicationContext和导入bean定义文件是最困难的部分, 我们看一下几种不同的实现途径。

抓取context

Spring在导入bean定义方面为你提供了多种选择。在当前发行版本(1.0.2)中, bean定义必须由文件导入。你可能还要实现你自己的ApplicationContext, 添加从其它资源(比如说数据库)导入的支持。虽然很多context提供了导入bean的功能, 但你只需要少数几个, 如下列表所示。其它是内部类, 只有框架自身才会用到。

- `ClassPathXmlApplicationContext`
[<http://www.springframework.org/docs/api/org.springframework.context.support.ClassPathXmlApplicationContext.html>]: web应用程序中从classpath(也就是WEB-INF/classes 或是WEB-INF/lib下的jar文件)中入content文件。使用 `new ClassPathXmlApplicationContext(path)`来初始化, 其中path是文件的路径。path参数还可以用path的String数组, 并支持Ant形式的模式匹配规则来获得名称类似的多个文件(参见 `PathMatcher` [<http://www.springframework.org/docs/api/org.springframework.util.PathMatcher.html>]的javadoc)。这是一种在单元测试中好用的context。

```
String[] paths = {"/WEB-INF/applicationContext*.xml"};
ApplicationContext ctx = new
    ClassPathXmlApplicationContext(paths);
```

- `FileSystemXmlApplicationContext`
[<http://www.springframework.org/docs/api/org.springframework.context.support.FileSystemXmlApplicationContext.html>]: 从文件系统中导入context文件, 测试方便。使用`new FileSystemXmlApplicationContext(path)`初始化, path是文件的相对或是绝对路径。path参数也可以是一个String数组。

```
String[] paths = {"C:/source/myusers/web/WEB-INF/applicationContext*.xml"};
ApplicationContext ctx = new
    FileSystemXmlApplicationContext(paths);
```

- `XmlWebApplicationContext`
[<http://www.springframework.org/docs/api/org.springframework.web.context.support.XmlWebApplicationContext.html>]: 通过ContextLoaderListener从内部导入context文件, 但也可以在外部使用。例如, 如果

你运行的服务器没有按web.xml中指定的顺序加载Listener，你可以必须在另一个Listener中使用它。下面是使用这个Loader的代码。

```
XmlWebApplicationContext ctx =  
    new XmlWebApplicationContext();  
context.setServletContext(ctx);  
context.refresh();
```

提示

如果你的容器没有按指定的顺序来加载listener，你也可以通过继承ContextLoaderListener自己写一个listener。这种方法，你可以控制ApplicationContext的初始化过程，并在初始化后进行使用。

```
public class StartupListener extends ContextLoaderListener  
    implements ServletContextListener {  
  
    public void contextInitialized(ServletContextEvent event) {  
        // call Spring's context ContextLoaderListener to initialize  
        // all the context files specified in web.xml  
        super.contextInitialized(event);  
        // get beans from WebApplicationContext in servletContext  
    }  
}
```

大多数应用程序使用多个配置context文件进行配置。例如application-context-hibernate.xml， applicationContext-service.xml 和 applicationContext-security.xml。加载多个文件也不难，你只要一一指定每个文件的名称，或使用通配符。这是Spring开发人员中最常见的方法。另一种选择是在现有的context文件添加一个<import>元素，此元素必须位于<beans>和第1个bean之间。“resource”属性中的所有的“path”都是假定是相对当前文件。

```
<beans>  
    <import resource="applicationContext-hibernate.xml"/>  
    <import resource="conf/applicationContext-security.xml"/>  
    <bean id="firstBean" class="..." />
```

一旦你获得了一个context的引用，就可以通过ctx.getBean("beanId")来获得各个bean的引用。你需要将强制转换成某种类型，但这是很容易的部分。在上面的context中，ClassPathXmlApplicationContext最灵活。它不关心文件的位置，只要它们在classpath中。这样你移动了文件，只需要简单的改一下classpath。

单元测试和导入context的一些技巧

用Spring来写单元一般来说非常简单。为Spring预设的bean可以用mock放入setter中，在脱离容器的环境下进行实例化和测试。由于Spring基于接口的设计，测试也很简单，你可以选择如何测试实现类。测试最简单的步骤可以归结为，使用ClassPathXmlApplicationContext，获得一个

bean的引用，调用它的方法。这种要求你的测试是基于接口，而不实现类。如果你想换掉实现(通过修改的bean的class属性)，你不必修改你测试中任何代码。

然而，ApplicationContext要花几秒钟时间来初始化。随着你的应用程序不断的发展，初始化的时间也会随之增加。此外，如果你在是一个setUp()方法中导入context，那在调用一个testXXX()之前，context每次都会实例化。还好，有一些简单的方案让每个TestCase只导入一个context。

第一，可以在一个suite中使用JUnit的TestSetup类，或者把导入的代码放在测试的static块中。

```
protected static ApplicationContext ctx = null;
static {
    ctx = new ClassPathXmlApplicationContext("/appContext.xml");
}
```

第2种方案是直接测试实现类，不使用Spring的BeanFactory或ApplicationContext。这种方法一般包括通过"new"操作创建一个实例，手动设置依赖关系，调用方法进行测试。使用这种技术，你可以把所依赖的类替换成mock object，它能加速你的测试，并可以从环境的依赖性解脱出来。

单元测试的更多信息将在第8章(测试Spring应用程序)中讨论。

国际化和MessageSource

国际化或称为i18n，是应用开发特别是web应用中的一个重要的概念。你的用户可能来自不同的国家，讲不同语言。他们在显示网站时，可能把浏览器设置成优先使用母语。

ApplicationContext接口继承了MessageSource
[<http://www.springframework.org/docs/api/org/springframework/context/MessageSource.html>]接口，它提供了消息(i18n)机制。和HierarchicalMessageSource
[<http://www.springframework.org/docs/api/org/springframework/context/HierarchicalMessageSource.html>]一起，他们是Spring所提供的处理消息的基本接口。当导入一个ApplicationContext时，它根据context中定义的名称messageSource来查找一个bean。如果在当前context及上一级context中没有找到这样的bean，就会创建一个StaticMessageSource
[<http://www.springframework.org/docs/api/org/springframework/context/support/StaticMessageSource.html>]，这样getMessage()不致调用失败。

在目前的Spring版本中，有两种MessageSource实现。它们是ResourceBundleMessageSource
[<http://www.springframework.org/docs/api/org/springframework/context/support/ResourceBundleMessageSource.html>](从一个.properties文件中读取)和StaticMessageSource
[<http://www.springframework.org/docs/api/org/springframework/context/support/StaticMessageSource.html>](难用，但允许你添加编程性的消息)。下面是一个messageSource bean定义的例子，从classpath中导入messages.properties。

```
<bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename"><value>messages</value></property>
</bean>
```

如果你想指定多个ResourceBundle，你可以把单数的basename属性替换成basenames属性。

```
<property name="basenames">
  <list>
    <value>messages</value>
    <value>errors</value>
  </list>
</property>
```

下一章会定义一个messageSource bean，并利用它获得一个error消息和success消息

事件的发布和订阅

ApplicationContext 通过 `ApplicationEvent` [http://www.springframework.org/docs/api/org.springframework.context.ApplicationEvent.html]类和 `ApplicationListener` [http://www.springframework.org/docs/api/org.springframework.context.ApplicationListener.html]接口支持事件处理(Event Handling)。如果你愿意使用这个功能，你要bean实现一个 `ApplicationListener`，这样，当一个 `ApplicationEvent` 发布到context时，你的bean就会得到通知。参见Spring事件列表中的3种标准事件。

表 3.2. Spring事件

事件	描述
ContextRefreshedEvent	当初始化或是刷新ApplicationContext时，发布事件。这里初始化是指所有的bean已经导入了，单态bean已经预先实例化，ApplicationContext已经准备就绪可以使用。
ContextClosedEvent	ApplicationContext 关闭时 (在ApplicationContext上调用close()方法)事件发布。这里关闭意味销毁(destory)所有的singleton。
RequestHandledEvent	一种web特定事件用来通知一个HTTP请求服务完成(在请求完成时发布)。注意这里事件仅适用于web应用程序 (使用Spring的 DispatcherServlet)。

你也可以通过调用的ApplicationContext的publishEvent()方法来实现自定义的事件。请参考 `Spring 参考手册` [http://www.springframework.org/docs/reference/beans.html#context-functionality-events]中的例子。

深入探讨MyUsers的ApplicationContext.xml

在第2章的MyUsers应用中，你通过web/WEB-INF/ApplicationContext.xml导入bean定义的。在这个文件中，定义了7个bean。其中，3个(userDAO, userManagerTarget和/user)是指向你建的类。这表明了Spring的强大：你所用的类中4个(dataSource, sessionFactory, transactionManager和userManager)是内部Spring类(你用来设置属性的)。

现在你应该明白了一个bean定义XML文件如何组成的。我鼓励你进一步看一下MyUsers的ApplicationContext.xml，我想你会更容易理解。

本章小结

本章中，你学习了反转控制(Martin Fowler最近给它取了个别名叫"依赖性注射")。使用Spring这样的容器来注射依赖性是一种配置应用，减少耦合简洁而强大的方法。

BeanFactory和bean定义是Spring IoC容器幕后的驱动力量，允许你指定依赖关系，在XML文件控制你的类的生命周期。弄懂BeanFactory是如何工作的，bean定义是如何指定的，有利于使你成为一个高效的Spring开发人员。搞清属性如何设置——它们是否是String，object，其它类的引用，这是一笔巨大的财富。你也定义了更复杂的属性如Properties，List，Map。这是你贯穿整个应用的关键所在，而不是代码本身。现在，你的代码是松散耦合的，可以更多的关心其核心，你可以把ApplicationContext/BeanFactory想像成连接任何东西的容器。

下一章，你会把第2章的MyUsers应用程序转向使用Spring MVC框架。我想你在完成内部设置和配置后，你一定会惊奇，Spring的web开发多么的简单。

第 4 章 Spring MVC 框架

Spring MVC: 支持生命周期的框架

本章描述了Spring MVC框架的许多特性。还演示了如何用Spring MVC替换Struts web层。内容包括DispatcherServlet, 各种Controller, Handler Mappings, View Resolvers, Validation和Internationalization。还涉及了Spring JSP标签库。

概述

第3章探讨了Spring的BeanFactory及其生命周期, 利用它如何控制你的bean的调用和使用。Spring把这种理念引入到web层中, 并且在它的MVC框架中存在这样的概念。在流行的框架中, 如Struts, WebWork, Controller通常有包含一个唯一的方法: `execute()`。不管客户端发送的GET还是POST请求, 这些框架都会调用这个方法。对开发人员来说可以在这个方法编写任何的需要的逻辑。例如, 你可以填充下框的数据, 处理验证错误信息, 设置好视图页面以添加新记录。当然, 你可以在Struts或WebWork的Action中编写多个方法, 通过不同的请求参数或是按钮名称来进行转发。但有一点不可否认, 此方法调用根本不关心使用哪种请求方法(GET还是POST)。

Spring MVC可能更友好一些。它提供了两种控制器: 一个 `Controller` [<http://www.springframework.org/docs/api/org.springframework.web.servlet.mvc/Controller.html>] 接口 和一个 `SimpleFormController` [<http://www.springframework.org/docs/api/org.springframework.web.servlet.mvc/SimpleFormController.html>] 类。`Controller`最适合显示只读数据(如, 显示数据列表), `SimpleFormController`定位于处理表单(form)(如, 编辑, 保存, 删除)。`Controller`接口非常简单, 如下所示, 包含一个唯一的 `handleRequest(request, response)` 方法。

```
package org.springframework.web.servlet.mvc;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.ModelAndView;
public interface Controller {

    /**
     * Process the request and return a ModelAndView object which the
     * DispatcherServlet will render. A null return is not an error:
     * It indicates that this object completed request processing
     * itself, thus there is no ModelAndView to render.
     */
    ModelAndView handleRequest(HttpServletRequest request,
        HttpServletResponse response)
        throws Exception;
}
```

`handleRequest()` 方法返回一个 `ModelAndView` [<http://www.springframework.org/docs/api/org.springframework.web.servlet/ModelAndView.html>] 类, 这个类有两个截然不同的部分model和view。model是你想要显示的信息, view是你想要显示的位置的逻辑名称。model可以是一个对象的名称, 或是由多个对象组成的 `java.util.Map`。view可以是一个 `View` [<http://www.springframework.org/docs/api/org.springframework.web.servlet/View.html>] 类(多个不同 `View` 类型的接口) 或是一个由 `ViewResolver`

[<http://www.springframework.org/docs/api/org.springframework.web.servlet.ViewResolver.html>]能识别的字符串的name。Spring中提供丰富的View在第6章中详细的讨论。

另外，SimpleFormController一个具体的类，包括几个在处理输入数据的表单时会调用的方法。一个是接口，另一个为一个超类为开发提供了灵活性。Spring中所有的Controller都用到了Controller接口，而SimpleFormController只不过是它多个方法默认设置的一种实现。如果你不需要FormController提供的如此丰富的功能，你可以继承AbstractCommandController [<http://www.springframework.org/docs/api/org.springframework.web.servlet.mvc.AbstractCommandController.html>]类，由一个HttpServletRequest来组装你的command bean。Spring为FormController提供了良好的继承性，这已经超出了本章的范围。在大多数情况下，你用不到它们，SimpleFormController完全可以满足大多数的需求。

使用SimpleFormController时，一些方法在由GET请求调用，另一些则由POST请求调用。这符合大多数web应用的规则，一个"GET"意味着有一个"编辑(edit)"，"POST"则表示存在一个"保存"或"更新"。这样你可以把两种操作分开。在Struts中，你可以使用DispatchAction(或是它的一个子类)达到类似的效果。第2章中，我们使用了一个DispatchAction [<http://struts.apache.org/api/org/apache/struts/actions/DispatchAction.html>]把各种CRUD操作分开，放在各方法中。Spring的方法更为合理，你可以复用所有的CRUD方法。本章的"方法生命周期回顾"一节到最后会讨论不同的SimpleFormController方法(及它们何时被调用的)。

本章仅涉及开发一个使用验证的web程序你要了解哪些东西，包括以下主题：

- Spring Controller单元测试
- 配置 DispatcherServlet [<http://www.springframework.org/docs/api/org.springframework.web.servlet.DispatcherServlet.html>] 和 ContextLoaderListener [<http://www.springframework.org/docs/api/org.springframework.web.context.ContextLoaderListener.html>]
- 为UserController编写单元测试(用于显示用户列表)
- 编写UserController并在action-servlet.xml进行配置
- 创建userList.jsp文件显示用户列表
- 为用户FormController(编辑，保存，删除用户)创建单元测试
- 编写UserFormController并在action-servlet.xml进行配置
- 创建userForm.jsp以编辑用户资料
- 为Spring配置Commons Validator
- SimpleFormController – 方法生命周期回顾
- Spring JSP标签

正如前面提到的，Spring MVC框架与传统的框架，如Struts，WebWork等稍有不同。使用Spring，我往往是用两个控制器来处理主/细节(master/detail)页面。使用Struts时，我总是用一个Action来完成删除，编辑，保存，及列出数据表中的记录。这里"列出"是指从某个表中取出所有记录的过程。这能满足我大多数web开发的要求。使用SpringMVC，不要用一个控制器来揽掉所有的事，更简单的方法是使用一个控制器来进行显示(master)，另外一个来完成删除/编辑/保存(detail)。

注意

如果不想为每个显示页面创建一个新的Controller，你可以创建一个MultiActionController [<http://www.springframework.org/docs/api/org.springframework.web.servlet.mvc/multiaction/MultiActionController.html>]，为每个显示定义一个独立的方法。

第11章会把Spring MVC和更为流行的MVC框架(Struts, WebWork, Tapestry和JSF)作一个详细的分析。将会解释每个框架的强项和弱点，此外还会演示Spring中间层如何分别与它们进行集成。

Spring Controller单元测试

当我开始使用Spring MVC框架时，我发现它不大好测试。我非常好奇，因为Spring所鼓吹的一个好处便是“使用Spring构建的应用程序测试很简单¹”。测试Controller还容易，但是测试SimpleFormController就不那么简单。我所面临的问题是，找不到可行的方案(例如Mock Object [<http://www.mockobjects.com/>])有这样的API，来处理你平常在web应用中所做的事情:设置request参数/属性，从application作用域中抓取资料等。使用Struts的话，你可以使用StrutsTestCase [<http://strutstestcase.sf.net/>]，它在提供大多数Struts和Servlet API类模拟(mock)实现方面做得非常不错。

因为Spring是开源代码的，我可以探石问路，看看开发人员在对Controller进行内部测试时使用些什么东西。我发现他们使用了大量的自用(home-grown)的Mock，这些恰好涉及到了我所需要的大部分的Servlet API。在发现那些东西不久，Spring团队便对这些mock进行了整理，以便公开使用，并添加到Spring发布产品中。在你写单元测试时你会用到这些类。如果你打算在项目中使用类似的mock，请确保spring-mock.jar在你的classpath中。

和前几章类似，在本章中，你接下来会做一些例子。最简单的方法是从<http://sourcebeat.com/downloads>下载MyUsers Chapter 4捆绑包。这个包和你在第2章下载的Equinox包类似。但是，所有Spring相关的组件都删除了，它为纯Spring应用程序的设计的。web/WEB-INF/lib目录下已经包含了所有你将在本章要用到的jar文件。

你也可以使用第2章开发的应用程序。如果准备这么做，请从<http://sourcebeat.com/downloads>上下载Chapter 4 JAR。下面一节会讨论下载文件中Spring是如何配置的。还会向你演示，如果从上一章基于Struts的应用程序进行转换，需要作哪些修改。

配置DispatcherServlet和ContextLoaderListener

Spring MVC框架与Struts相似的地方在于，默认情况下它使用一个控制器的一个唯一的实例。通过在你的控制器的bean定义中添加singleton="false"，就可以把你的Controller改成为每个请求创建新的实例。另外，如果你倾向于WebWork的“为每个请求创建一个新的action”，你仍然使用那种功能。

和大多数Java web框架类似，Spring MVC用一个唯一的servlet来处理所有的请求²²。这个servlet就是DispatcherServlet。它负责把请求“转发”给处理器(handler)，处理器根据映射(mapping)确定下一步往哪里走。在第4章下载的包中，DispatcherServlet在web/WEB-INF/web.xml中已经配置好了。其mapping设置为 "*.html"，意思是任何以 ".html" 结尾的URL都由这个servlet进行处理。

¹来自TheServerSide.com上的Rod Johnson的文章Introducing the Spring Framework [<http://www.theserverside.com/articles/article.tss?l=SpringFramework>]。

²这是一种核心的J2EE模式，称为Front Controller [<http://java.sun.com/blueprints/corej2eepatterns/Patterns/FrontController.html>]。

如果你准备修改在第2章中创建的程序，你必须对MyUsers应用程序进行配置，使用DispatcherServlet作为前端控制器，来替换Struts的ActionServlet [http://struts.apache.org/api/org/apache/struts/action/ActionServlet.html]。下面一节会指导你完成这些操作。

修改web.xml，使用Spring的DispatchServlet

此时，你的硬盘上应该已经设置好了MyUsers项目。打开web/WEB-INF/web.xml，把“action”servlet的<servlet-class>由

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
    org.apache.struts.action.ActionServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

修改为：

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

另外，把action的<servlet-mapping>由*.do修改为*.html。你正是为了提供HTML，所以用它替换掉*.do看起来更有意义。再者，没有借所用框架来打广告的嫌疑。

```
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.html</url-pattern>
</servlet-mapping>
```

默认情况下，DispatcherServlet会在WEB-INF下查找名为servlet-name-servlet.xml的XML文件。在本例中，一旦你创建action-servlet.xml，它就查找和加载这个文件。这个文件包含了所有MyUsers应用中用到的web控制器和配置。

在第2章中，你使用了针对Spring的SpringPlugin(ContextLoaderPlugin [http://www.springframework.org/docs/api/org/springframework/web/struts/ContextLoaderPlugIn.html])来加载bean配置文件。但是同时，ContextLoaderListener也在你的web.xml中进行了配置。这导致重复加载了ApplicationContext.xml文件。这就是在对你的Action类进行单元测试，而不需要手动加载context文件的原因。

注意

这个Listener只能在Servlet2.3的容器中运作，如果你用的是较老的容器，请使用ContextLoaderServlet [<http://www.springframework.org/docs/api/org/springframework/web/context/ContextLoaderServlet.html>]。

既然ContextLoaderListener已经在web.xml中配置好了，在你这一方不需要再进行配置。如果你有多个bean定义的文件，可以添加一个contextConfigLocation参数来指明各个文件。例如，在MyUsers中你可以这么做，在web.xml中的“sitemesh” filter之后和它的<filter-mapping>之前直接加入下面所示的XML片断。

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    /WEB-INF/applicationContext1.xml
    /WEB-INF/applicationContext2.xml
  </param-value>
</context-param>
```

注意

注意这两个文件的路径是以空格隔开的，这些路径也可以以逗号隔开。

这些是使用Spring MVC框架配置Java web应用的一些基本步骤。这里回顾一下这些步骤。

1. 在web.xml文件中，添加一个DispatcherServlet的<servlet>定义，并且配置好它的<servlet-mapping>。
2. 如果你有一个以上的context文件，针对你的bean定义文件，定义一个带有多个路径的contextConfigLocation <context-param>。
3. 为ContextLoaderListener添加一个<listener>。

删除Struts，添加Spring文件

删除UserAction和UserActionTestStruts类，还有web/WEB-INF/lib下一些jar文件。下面是快速完成这一操作的几个命令。

```
rm src/org/appfuse/web/UserAction.java
rm test/org/appfuse/web/UserActionTest.java
rm web/WEB-INF/lib/struts*
rm web/WEB-INF/struts-config.xml
```

现在从action-servlet.xml中删除UserAction类的定义。删除文件中的以下几行：

```
<bean name="/user" class="org.appfuse.web.UserAction"
  singleton="false">
  <property name="userManager">
    <ref bean="userManager"/>
  </property>
```

```
</bean>
```

下载Chapter 4 JARs [<http://sourcebeat.com/downloads/splive/chapter4-jars.zip>]到你的硬盘。把jar文件放入web/WEB-INF/lib目录。spring.jar包含了Spring 1.1.1(Equinox 1.0附带的是Spring 1.0.2)，spring-mock.jar中包含了Servlet API的mock，spring-sandbox.jar用来支持Commons Validator。另一文件，validation-rule.xml也包含在下载文件中，把它放入web/WEB-INF中。本章后面会涉及到配置验证。

现在你可以准备开发你的Controller。

为UserController编写单元测试

为了实践测试驱动开发，从编写一个UserController单元测试开始。此类从业务逻辑类(UserManager)中返回所有用户的列表。如果你还不知道TDD，Dave Thomas的博客 [<http://www.pragprog.com/ragdave/Practices/TestDrivenOrTestFirst.rdoc>]中有一个不错的定义：

以我看，测试驱动开发是构思编码的一种重要的途径。它使用测试来描绘设计和实现的蓝图。你听从测试告诉你的一切，相应的变换编码。发现很难在隔离环境中进行测试？重构你的代码减少耦合。不大可能模拟出某个特定的子系统？试一下添加前端(facade，注：设计模式的一种)或是接口，让分层更清晰。测试推动了设计，测试验证了实现。

要为Controller编写一个单元测试，在test/org/appfuse/web(你可能要创建这个目录/包)下新建一个文件UserControllerTest.java。这个类应该继承junit.framework.TestCase，定义一个setUp()方法，使用XmlWebApplicationContext来加载context文件。在ClassPathXmlApplicationContext上使用这个ContextLoader的原因是，仅作用于web(webonly)的bean才能被实例化。这里仅作用于web(web-only)的bean，指的是需要有一个WebApplicationContext的bean。

```
public void setUp() {
    String[] paths = {"/WEB-INF/applicationContext.xml",
        "/WEB-INF/action-servlet.xml"};
    ctx = new XmlWebApplicationContext();
    ctx.setConfigLocations(paths);
    ctx.setServletContext(new MockServletContext(""));
    ctx.refresh();
}
```

上面的代码会加载它们各自的XML文件定义的任何bean。现在为了测试你的Controller写一个test方法。这里开始进入TDD，测试会推动你的设计。编写一个方法获取一个用户列表，验证操作成功，确认视图返回的是你想要的东西。下面是完整的UserControllerTest代码，所以你很容易集成到你的项目中去。testGetUsers()就是你最感兴趣的方法。

```
package org.appfuse.web;
// use your IDE (Eclipse and IDEA rock!) to add imports

public class UserControllerTest extends TestCase {
    private XmlWebApplicationContext ctx;

    public void setUp() {
        String[] paths = {"/WEB-INF/applicationContext.xml",
```

```

    "/WEB-INF/action-servlet.xml"};
    ctx = new XmlWebApplicationContext();
    ctx.setConfigLocations(paths);
    ctx.setServletContext(new MockServletContext(""));
    ctx.refresh();
}

public void testGetUsers() throws Exception {
    UserController c = (UserController)
        ctx.getBean("userController");
    ModelAndView mav =
        c.handleRequest((HttpServletRequest) null,
            (HttpServletResponse) null);
    Map m = mav.getModel();
    assertNotNull(m.get("users"));
    assertEquals(mav.getViewName(), "userList");
}
}

```

注意

当你为自己的项目编写单元测试时，我建议创建一个继承 `TestCase` 的 `BaseControllerTestCase`，所有的 `*ControllerTest` 继承它。在这个类的 `setUp()` 方法中，你可以为所有的子类加载 context。如果你需要这么做，确保在你的子类的 `setUp()` 方法中调用了 `super.setUp()`。

跟进 `testGetUsers()` 方法，你先是获得一个 `UserController`，并调用它的 `handleRequest()` 方法，它会返回一个 `ModelAndView` [http://www.springframework.org/docs/api/org.springframework.web.servlet.ModelAndView.html] 类。此方法对所有的 Spring Controller 公用的，所以你会用这个方法测试 `FormController`。`ModelAndView` 是 web 框架中一个独特的概念。它基本上包含了你下一页相关的资料 (view) 和要暴露给它的的数据 (model)。在 Struts，model 和 view 是分开的。model 通常放入 request (或是 session) 作用域内，Action 主要返回 `ActionForward`，它是 URL 一种特殊的包装器。

创建 UserController，配置 action-servlet.xml

现在你已经写好了单元测试，下面你该创建 `UserController` 类以便能够编译它。首先在 `src/org/appfuse/web` (你可能要创建这个目录/包) 下新建一个名为 `UserController.java` 文件。此类应该实现 `Controller` 接口，并且实现它的 `handleRequest(request, response)` 方法。你还需要通过 `userManager` 来获得数据列表，所以你要添加一个私有的成员变量 `userManager` 和 `setUserManager()` 方法供 Spring IoC 容器使用。当你在下一节中配置 Controller (也叫 bean) 时，你会添加一个 `userManager` 作为一个依赖关系。至此，你已经有如下的类结构。

```

package org.appfuse.web;
// Modern IDEs support easy importing

public class UserController implements Controller {
    private static Log log = LogFactory.getLog(UserController.class);
    private UserManager mgr = null;

    public void setUserManager(UserManager userManager) {

```

```

    this.mgr = userManager;
}
// put handleRequest() method here
}

```

现在实现handleRequest()来获得一个用户列表，并把用户传递到userList.jsp。

```

public ModelAndView handleRequest(HttpServletRequest request,
    HttpServletResponse response)
    throws Exception {
    if (log.isDebugEnabled()) {
        log.debug("entering 'handleRequest' method...");
    }
    return new ModelAndView("userList", "users", mgr.getUsers());
}

```

这个方法非常简单，实际上，如果没有开头的logging语句，它只有一行代码。

现在编译UserControllerTest类应该没问题，但是如果你试图运行测试(使用ant test -Dtestcase=UserController)，结果是失败。

```

[junit] Testcase: testGetUsers(org.appfuse.web.UserControllerTest): Caused an
ERROR
[junit] Line 7 in XML document from resource [/WEB-INF/action-servlet.xml] of
ServletContext is invalid; nested exception is org.xml.sax.SAXParseException:
Element "beans" requires additional elements.

```

错误表明，web/WEB-INF/action-servlet.xml文件不合法。这是因为其中没有bean定义。为了通过测试，编辑web/WEB-INF/action-servlet.xml文件。action-servlet.xml开始和其它的Spring配置文件类似，文件顶部是DTD，开头是<beans>元素。

添加一个userController的bean定义后，文件看起来如下：

```

<?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="userController" class="org.appfuse.web.UserController">
        <property name="userManager">
            <ref bean="userManager"/>
        </property>
    </bean>
</beans>

```

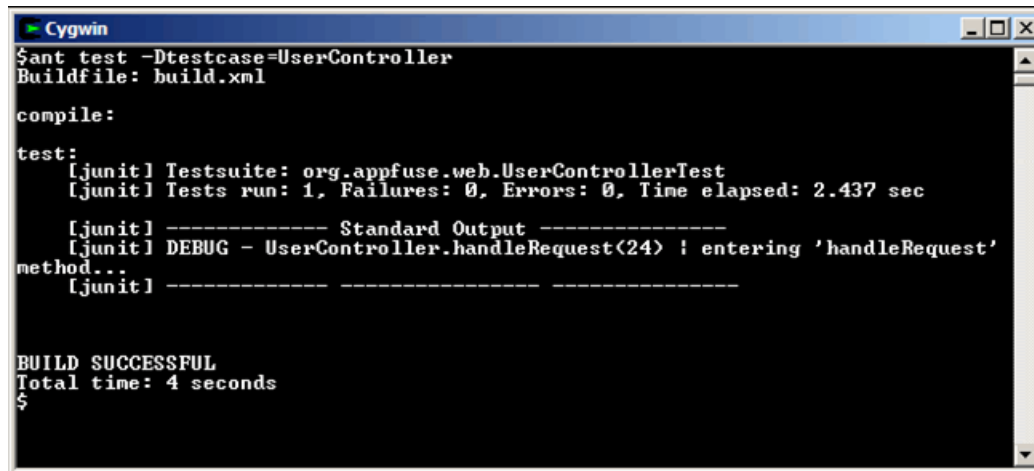
现在你的UserControllerTest应该可以正常的运行了，通过运行ant test -Dtestcase=UserController命令来执行它，或在Eclipse或是IntelliJ IDEA中把它作为一个JUnit Test来运行。

注意

在本书的FAQ [<http://confluence.sourcebeat.com/display/SPL/FAQ>]中，有在Eclipse和 IntelliJ IDEA中设置MyUsers的指导。

从命令行中，输出结果如下面的屏幕所示。

图 4.1. 运行ant test -Dtestcase=UserController测试的结果



```

Cygwin
$ant test -Dtestcase=UserController
Buildfile: build.xml

compile:
test:
[junit] Testsuite: org.appfuse.web.UserControllerTest
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 2.437 sec

[junit] ----- Standard Output -----
[junit] DEBUG - UserController.handleRequest(24) ! entering 'handleRequest'
method...
[junit] -----

BUILD SUCCESSFUL
Total time: 4 seconds
$

```

创建userList.jsp页面显示用户列表

现在UserController可以正常运行，你必须配置Spring让它知道，userList实际上是指向userList.jsp页面的。实现的最简方法是使用InternalResourceViewResolver，它将名称解析到对应的文件。它允许你添加一个prefix和一个suffix，你很容易掌握你的JSP文件位置。在actionServlet.xml中的UserController后面，添加如下的XML文件块。

```

<bean id="viewResolver" class="org.springframework.web.servlet.view.
    InternalResourceViewResolver">
    <property name="viewClass">
        <value>org.springframework.web.servlet.view.JstlView</value>
    </property>
    <property name="prefix"><value>/</value></property>
    <property name="suffix"><value>.jsp</value></property>
</bean>

```

上面的viewResolver定义中，注意你指定了一个 `JstlView` [<http://www.springframework.org/docs/api/org.springframework.web.servlet.view/JstlView.html>]类作为viewClass属性。这样你就可以使用<fmt:message>标签，在Spring MVC中使用它的i18n特性要求先作一些准备。前缀是"/"后缀是".jsp"。如果你要把JSP文件移到/WEB-INF/pages中，你只需要修改这个前缀。

注意

如果开发的是产品应用程序，我强烈建议你JSP文件放在WEB-INF下面。如果你使用的一个servlet.2.3以上的容器，WEB-INF下的任何文件都不能通过浏览器访问。这在开发过程中非常有用，因为你它加强了MVC，并且强迫你通过控制器来访问你的视图。

通过添加viewResolver视图，UserController中的视图名称userList被解析为/userList.jsp。根据你所选的视图技术，你也可以使用其它几种ViewResolver。这些将在"第6章：视图的多种选择"中讨论。

现在为你的应用配置URL，让/users.html这条URL能够调用UserController类。要做到这一点，Spring MVC要求你定义一个HandlerMapping [http://www.springframework.org/docs/api/org.springframework.web.servlet.HandlerMapping.html] bean，在其中定义URL指向的控制器。大多数情况下，你只需要 SimpleUrlHandlerMapping [http://www.springframework.org/docs/api/org.springframework.web.servlet.handler.SimpleUrlHandlerMapping.html]。它允许你指定URL匹配模式映射到bean名称。为了映射/users.html到UserController bean，在web/WEB-INF/action-servlet.xml中添加以下代码：

```
<bean id="urlMapping"
    class="org.springframework.web.servlet.handler.Simple
    UrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/users.html">UserController</prop>
        </props>
    </property>
</bean>
```

注意

你也可以用 BeanNameUrlHandlerMapping [http://www.springframework.org/docs/api/org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping.html] 代替SimpleUrlHandlerMapping。这个处理器只是查找匹配URL的bean的名称(不是id)。所以如果你想把UserController命名为"/users.html"，那么这个处理器就会正确的解析它。如果你没有定义处理器，默认情况下使用BeanNameUrlHandlerMapping。

如果你是修改第2章创建的应用程序，要修改几个JSP文件。web/taglibs.jsp文件应该包含下面的内容：

```
<%@ page language="java" errorPage="/error.jsp" %>
<%@ taglib uri="http://java.sun.com/jstl/core_rt" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
<%@ taglib uri="http://www.springframework.org/tags/commons-validator"
prefix="html" %>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<%@ taglib uri="http://www.opensymphony.com/sitemesh/decorator"
prefix="decorator"%>
```

并且web/messages.jsp文件中应该包含以下的代码：

```
<%-- Success Messages --%>
<c:if test="${not empty message}">
    <div class="message">${message}</div>
    <c:remove var="message"/>
```



```
</c:if>
```

1. 在web目录下创建userList.jsp文件。此文件可能已经存在。
2. 添加一些代码，以便看到数据库中所有的用户。下面的代码中第一行就include了taglibs.jsp文件。这个文件包含了此应用所用到的所有的标签库的声明，大部分是JSTL和SiteMesh(用来美化JSP页面)的。

```
<%@ include file="/taglibs.jsp"%>
<title>MyUsers ~ User List</title>
<button onclick="location.href='editUser.html'">Add User</button>
<table class="list">
<thead>
<tr>
<th><fmt:message key="user.id"/></th>
<th><fmt:message key="user.firstName"/></th>
<th><fmt:message key="user.lastName"/></th>
</tr>
</thead>
<tbody>
<c:forEach var="user" items="${users}" varStatus="status">
<c:choose>
<c:when test="${status.count % 2 == 0}"><tr class="even"></c:when>
<c:otherwise><tr class="odd"></c:otherwise>
</c:choose>
<td><a href="editUser.html?id=${user.id}">${user.id}</a></td>
<td>${user.firstName}</td>
<td>${user.lastName}</td>
</tr>
</c:forEach>
</tbody>
</table>
```

3. 启用i18n信息查找(<fmt:message>标签)，添加一个messageSource到action-servlet.xml文件中。

```
<bean id="messageSource" class="org.springframework.context.support.
ResourceBundleMessageSource">
<property name="basename"><value>messages</value></property>
</bean>
```

basename属性值为messages，意思是在classpath的根目录下查找messages.properties。如果你使用多个.properties文件，你可以使用basenames，值为多个<values>的<list>。

注意

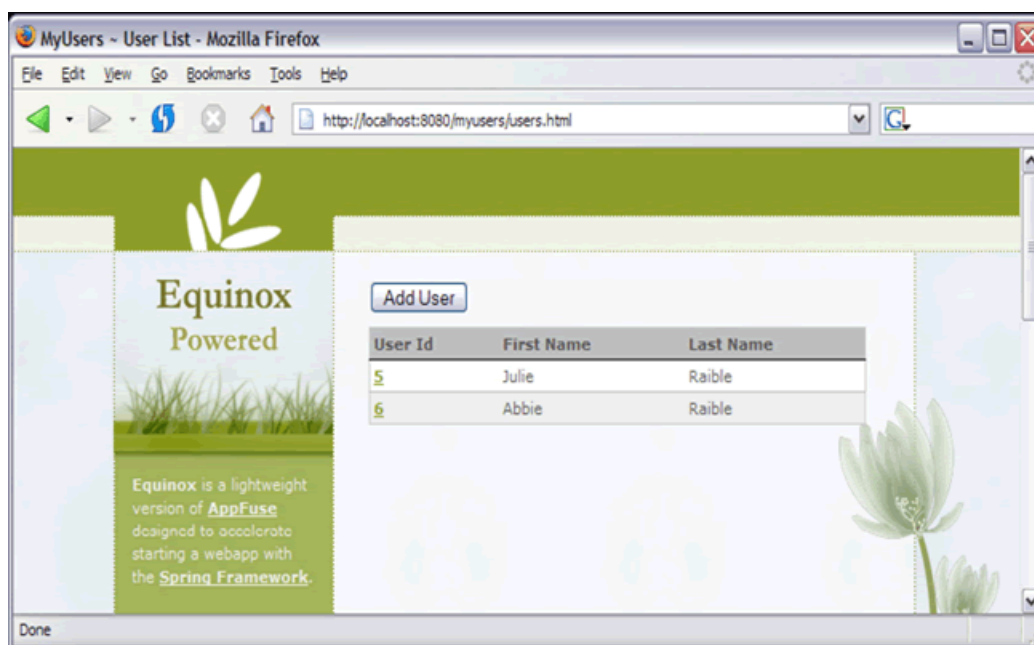
ResourceBundleMessageSource
[<http://www.springframework.org/docs/api/org.springframework.context.support.ResourceBundleMessageSource.html>]

依赖于Java的ResourceBundle [<http://java.sun.com/j2se/1.4.2/docs/api/java/util/ResourceBundle.html>], 它用来缓存动态加载的捆绑资源。使用这个类, 想在虚拟机运行期间重新加载资源是不可能的。如果要用这样的功能, 请参考 `ReloadableResourceBundleMessageSource` [<http://www.springframework.org/docs/api/org.springframework.context.support.ReloadableResourceBundleMessageSource.html>]。

为了测试你是正确的配置了Spring的处理器(handler)和解析器(resolver), 还有成功的修改 `userList.jsp` 文件, 启动Tomcat, 部署MyUsers(ant deploy), 然后通过浏览器浏览 `http://localhost:8080/myusers/users.html`。为了向数据库中添加一两个用户, 运行 `ant populate`。
注意

如果你运行了ant `populate`让用户显示出来, 参见这个的FAQ [<http://jroller.com/page/raible/20040809>]。

图 4.2. 运行ant populate测试的结果



如果你看到的和上面差不多, 那么恭喜你。如果不是, 发一封邮件Equinox用户列表 (users@equinox.dev.java.net)来获得帮助。

为UserFormController创建单元测试

显示屏幕只是抓取数据并显示出来, 是相对简单的部分。下面你要创建一个Controller和JSP页面来处理编辑数据库的记录。既然你已经在实践TDD, 从创建一个UserFormController(你还没有创建)的单元测试开始。在 `test/org/appfuse/web` 目录下创建文件 `UserFormControllerTest.java`。这个类继承了JUnit的 `TestCase`, 并且有如下的 `setUp()` 和 `tearDown()` 方法:

```
package org.appfuse.web;
// resolve imports using your IDE

public class UserFormControllerTest extends TestCase {
    private static Log log =
```

```

    LogFactory.getLog(UserFormControllerTest.class);
private XmlWebApplicationContext ctx;
private UserFormController c;
private MockHttpServletRequest request;
private ModelAndView mv;
private User user;

public void setUp() throws Exception {
    String[] paths = {"/WEB-INF/applicationContext.xml",
        "/WEB-INF/action-servlet.xml"};
    ctx = new XmlWebApplicationContext();
    ctx.setConfigLocations(paths);
    ctx.setServletContext(new MockServletContext(""));
    ctx.refresh();
    c = (UserFormController) ctx.getBean("userFormController");
    // add a test user to the database
    UserManager mgr = (UserManager) ctx.getBean("userManager");
    user = new User();
    user.setFirstName("Matt");
    user.setLastName("Raible");
    user = mgr.saveUser(user);
}

public void tearDown() {
    ctx = null;
    c = null;
    user = null;
}
// put testXXX methods here
}

```

这个类的`setUp()`方法创除了还要加载`action-servlet.xml`外，和第2章中`UserManagerTest`的`setUp()`方法类似。将业务组件和数据层的类用两个文件分开，有利于你在不改动`ApplicationContext.xml`的情况下轻松切换MVC框架。在需要对各层进行解耦时这种功能非常强大，还有利于重构。

下面添加几个方法，以测试你的CRUD操作(编辑，保存，删除)。下面的这些方法运用了Spring的Servlet API mock，以便测试Controller。

```

public void testEdit() throws Exception {
    log.debug("testing edit...");
    request = new MockHttpServletRequest("GET", "/editUser.html");
    request.addParameter("id", user.getId().toString());
    mv = c.handleRequest(request, new MockHttpServletResponse());
    assertEquals("userForm", mv.getViewName());
}

public void testSave() throws Exception {
    request = new MockHttpServletRequest("POST", "/editUser.html");
    request.addParameter("id", user.getId().toString());
    request.addParameter("firstName", user.getFirstName());
    request.addParameter("lastName", "Updated Last Name");
}

```

```

mv = c.handleRequest(request, new MockHttpServletRequest());
Errors errors =
    (Errors) mv.getModel()
        .get(BindException.ERROR_KEY_PREFIX + "user");
assertNull(errors);
assertNotNull(request.getSession().getAttribute("message"));
}

public void testRemove() throws Exception {
    request = new MockHttpServletRequest("POST", "/editUser.html");
    request.addParameter("delete", "");
    request.addParameter("id", user.getId().toString());
    mv = c.handleRequest(request, new MockHttpServletRequest());
    assertNotNull(request.getSession().getAttribute("message"));
}

```

在前面的几个方法中，你可能要好好考察几个类。首先是Spring的MockHttpServletRequest。给定了一个URI(统一资源定位器)，此类很容易调用GET和POST方法。它有多个不同的构造器，现列于下。

```

public MockHttpServletRequest(ServletContext servletContext)
public MockHttpServletRequest(ServletContext servletContext,
    String method, String URI)
public MockHttpServletRequest()
public MockHttpServletRequest(String method, String URI)

```

这是一个灵活的类，对测试Controller很有用。本来它仅用在Spring开发人员内部测试Controller。1.0.2发布后，它就包含在spring-mock.jar中。

第 2 个 类 是 E r r o r s
[\[http://www.springframework.org/docs/api/org.springframework.validation.Errors.html\]](http://www.springframework.org/docs/api/org.springframework.validation.Errors.html)接口，用来保存和暴露实现此接口对象的数据绑定错误信息。在你要创建的用户FormController中，必须注册一个数据绑定器为user.setId()处理java.lang.String 到 java.lang.Long的转换。

UserFormControllerTest类还不能通过编译，直到你创建了UserFormController类。如果你使用Eclipse或IntelliJ IDEA之类的IDE，在左侧有一图标提示你来自动创建新类。

创建UserFormController，并在action-servlet.xml中配置它

在src/org/appfuse/web下创建UserFormController.java文件。这个类应该继承SimpleFormController，它是FormController的一个具体的实现，提供了可配置的表单(form)和成功视图(view)。当验证出错时，它会自动重现表单视图，验证通过时，会显示成功视图。这个类提供了很多在显示表单和提交表单的生命周期中可以复写的方法。

与其它框架如Struts，WebWork相比，这是Spring MVC框架的一个独到之处。其它框架一般只提供一种方法进行复写，你无法更多的控制出现情况时到底发生了什么。当然，使用Spring MVC时，并没有强求实现它的生命周期方法。在你需要它时，它提供了一种选择。本章末，有一个关于各种生命周期方法及它们何时被调用的较详细的总览。

UserFormController设计得比较简单，以便你能轻松弄懂Spring MVC是如何工作的。实际上，你只需要复写两个方法：onSubmit()和formBackingObject()。onSubmit()处理表单post请

求, `formBackingObject()` 为请求提供一个对象, 它封装了HTML表单的字段值。对于获取现有的记录, 此方法是一个非常方便的位置, 它也是初始化空对象(例如, 显示空白表单)的好地方。此方法的默认实现只是创建了一个新的空对象。在Spring术语中, 此对象叫做"Command类"。下面熟悉一下SimpleFormController的内幕, 看一下代码。下面是UserFormController类的内容, 去掉了导入部分。

```
package org.appfuse.web;
// resolve imports using your IDE

public class UserFormController extends SimpleFormController {
    private static Log log =
        LoggerFactory.getLog(UserFormController.class);
    private UserManager mgr = null;

    public void setUserManager(UserManager userManager) {
        this.mgr = userManager;
    }

    public UserManager getUserManager() {
        return this.mgr;
    }

    /**
     * Set up a custom property editor for converting Longs
     */
    protected void initBinder(HttpServletRequest request,
        ServletRequestDataBinder binder) {
        NumberFormat nf = NumberFormat.getNumberInstance();
        binder.registerCustomEditor(Long.class, null,
            new CustomNumberEditor(Long.class, nf, true));
    }

    public ModelAndView onSubmit(HttpServletRequest request,
        HttpServletResponse response,
        Object command, BindException errors)
        throws Exception {
        if (log.isDebugEnabled()) {
            log.debug("entering 'onSubmit' method...");
        }
        User user = (User) command;
        if (request.getParameter("delete") != null) {
            mgr.removeUser(user.getId().toString());
            request.getSession().setAttribute("message",
                getMessageSourceAccessor().getMessage("user.deleted",
                    new Object[] {user.getFirstName() +
                        ' ' + user.getLastName()}));
        } else {
            mgr.saveUser(user);
            request.getSession().setAttribute("message",
                getMessageSourceAccessor().getMessage("user.saved",
                    new Object[] {user.getFirstName() +
                        ' ' + user.getLastName()}));
        }
    }
}
```

```
    }  
    return new ModelAndView(getSuccessView());  
  }  
  
  protected Object formBackingObject(HttpServletRequest request)  
  throws ServletException {  
    String userId = request.getParameter("id");  
    if ((userId != null) && !userId.equals("")) {  
      return mgr.getUser(request.getParameter("id"));  
    } else {  
      return new User();  
    }  
  }  
}
```

从这段代码中，你可以看到Spring是如何为"添加"创建一个空对象，为"编辑"创建一个填充了数据的对象。你也可以看到onSubmit()是多么的简单：它调用userManager来保存/删除User对象。onSubmit()返回一个ModelAndView。initBinder()主要负责command类的String到Property的转换。在本例中，

CustomNumberEditor

[\[http://www.springframework.org/docs/api/org/springframework/beans/propertyeditors/CustomNumberEditor.html\]](http://www.springframework.org/docs/api/org/springframework/beans/propertyeditors/CustomNumberEditor.html)
转换"user.id"属性，它是一个java.lang.Long型。这样做不是必须的，因为这个editor是默认注册的，但同时向你说明了你自定义行为。表4.1列出了内置property editor。

表 4.1. 内置的Property Editor³³

类	释义	默认是否注册
ByteArrayPropertyEditor	用于byte数组的Editor。String可以简单的转换成相应的byte表示形式。默认由BeanWrapperImpl注册。	是
ClassEditor	将Spring的表示类解析成实际类或其它相近的类。当一个类没有找到时，会抛出IllegalArgumentException异常。	
CustomBooleanEditor	能自定义的property editor，用于Boolean properties。默认由BeanWrapperImpl注册，但可以通过把它的自定义实例注册为自定义的editor方式进行覆写。	是，
CustomCollectionEditor	用于Collection的property editor，能将任何源Collection二项式转换成指定目标的Collection形式。	是，用于Set，SortedSet和List
CustomDateEditor	能自定义的property editor，用于java.util.Date，支持自定义的DateFormat。	否
CustomNumberEditor	能自定义的property editor，用于任何Number的子类，如Integer，Long，Float，Double。	是
FileEditor	能将String转化成File对象。	是
InputStreamEditor	单向的property editor。能通过一个文本字符串产生(通过一个中间的ResourceEditor或Resource)一个InputStream，所以InputStream可以直接设置。注意，默认使用时，没有为关闭InputStream。	是
LocaleEditor	能将字符串转换成Locale对象和通用形式(字符格式为[language]_[country]_[variant]，Locale提供的toString()方法能完成同样的操作)。	是
PropertiesEditor	用于java.util.Properties对象的Editor。	是
StringArrayPropertyEditor	用于String数组的Editor。	是
StringTrimmerEditor	Property editor，整理String(去掉两头的占位符)。	是
URLEditor	用于java.net.URL的Editor，直接使用URL property，代替String property。	是

你可能发现遗漏了异常处理，这主要是让一切看起来更简单。到第5章，会介绍到一种异常处理策略。

现在在action-servlet.xml中配置这个Controller。在这个bean的定义中，你将设置几个显式值：“successView”，“formView”，command类及它的名称。这里也会将它依关系注射到UserManager。下面是你要添加到web/WEB-INF/action-servlet.xml中的定义。

³³基于Spring参考文档，<http://www.springframework.org/docs/reference/validation.html#beans-beans-conversion>。

```

<bean id="userFormController"
  class="org.appfuse.web.UserFormController">
  <property name="commandName"><value>user</value></property>
  <property name="commandClass">
    <value>org.appfuse.model.User</value>
  </property>
  <property name="formView"><value>userForm</value></property>
  <property name="successView"><value>redirect:users.html</value></property>
  <property name="userManager"><ref bean="userManager"/></property>
</bean>

```

在上面的定义中，`redirect`这个前缀表明你会向下一个Controller发送一个`redirect`。另一种方法是用 `RedirectView` [http://www.springframework.org/docs/api/org.springframework.web.servlet.view.RedirectView.html] 类包装逻辑视图名称。

```
return new ModelAndView(new RedirectView(getSuccessView()));
```

另一种选择是使用一个`forward`前缀。下面是一个使用这个前缀的例子。使用`redirect`通常是解决web应用程序中常见的"重复提交"问题的一种最好的选择。

```

<property name="successView">
  <value>forward:users.html</value>
</property>

```

在此定义中，`commandName`属性是可选的，如果你选择不设置此属性，它默认是“`command`”。接下来的部分，当你为此控制器创建JSP页面时，你就会看到那里用到了`commandName`属性。这不需要在控制器和测试类中通过代码实现。

既然你正在编辑`action-servlet.xml`文件，添加一条URL映射，使用“`userFormController`”来解析`editUser.html`。在`urlMapping`bean的属性`mappings`添加额外的一行代码来完成此操作。

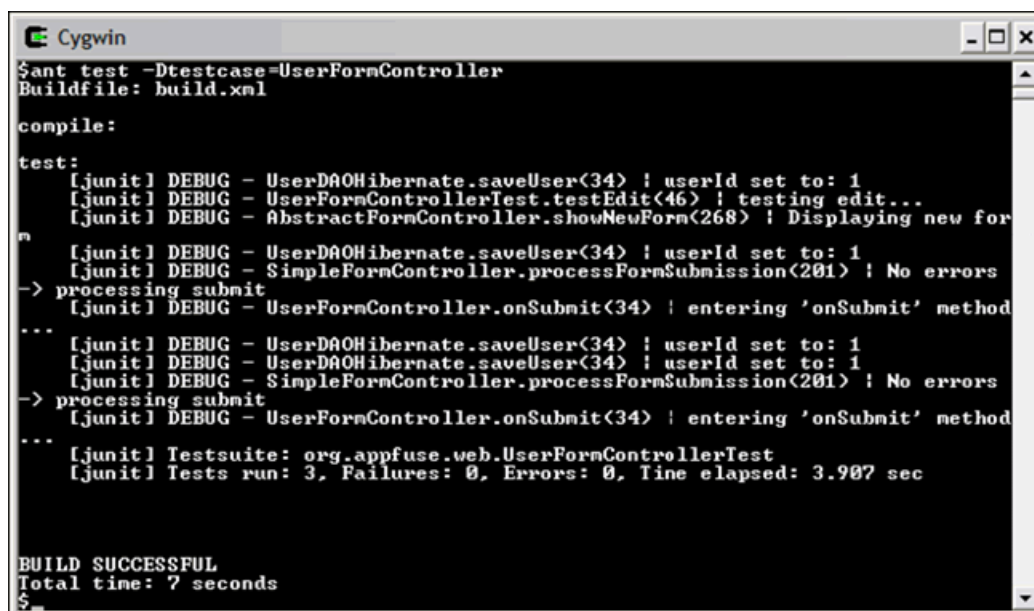
```

<property name="mappings">
  <props>
    <prop key="/users.html">userController</prop>
    <prop key="/editUser.html">userFormController</prop>
  </props>
</property>

```

现在你能通过IDE或是命令行来运行`UserFormControllerTest`。图4.3显示了从命令行运行 `ant test -Dtestcase=UserForm` 的输出结果。

图 4.3. 运行ant test -Dtestcase=UserForm测试的结果



```

Cygwin
$ant test -Dtestcase=UserFormController
Buildfile: build.xml

compile:
test:
[junit] DEBUG - UserDAOHibernate.saveUser(34) ! userId set to: 1
[junit] DEBUG - UserFormControllerTest.testEdit(46) ! testing edit...
[junit] DEBUG - AbstractFormController.showNewForm(268) ! Displaying new form
...
[junit] DEBUG - UserDAOHibernate.saveUser(34) ! userId set to: 1
[junit] DEBUG - SimpleFormController.processFormSubmission(201) ! No errors
-> processing submit
[junit] DEBUG - UserFormController.onSubmit(34) ! entering 'onSubmit' method
...
[junit] DEBUG - UserDAOHibernate.saveUser(34) ! userId set to: 1
[junit] DEBUG - UserDAOHibernate.saveUser(34) ! userId set to: 1
[junit] DEBUG - SimpleFormController.processFormSubmission(201) ! No errors
-> processing submit
[junit] DEBUG - UserFormController.onSubmit(34) ! entering 'onSubmit' method
...
[junit] Testsuite: org.appfuse.web.UserFormControllerTest
[junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 3.907 sec

BUILD SUCCESSFUL
Total time: 7 seconds
$

```

如果你看到类似上面的输出结果，表示一切正常。下面创建JSP页面与UserFormController进行交互。

创建userForm.jsp页面，以便编辑用户资料

在本应用程序的Struts版本中，userForm.jsp比较简单：主要是得益于Struts的<html:form>和<html:text> JSP标签。不巧的是，Spring没有提供类似的特定的form标签。然而，这有充分的理由。缺乏丰富的标签的主要前提是为是给了用户最大范围的控制HTML权利。由于Spring的表单处理标签不生成任何HTML，那么用户就可以完全控制它。很多Struts用户正在转向Spring，邮件列表中已经有一些相关的讨论。在撰写本书时，Spring的JIRA中有人提出一个需要加强的请求是添加JSP 2.0标签文件 [http://opensource.atlassian.com/projects/spring/browse/SPR-310]以简化HTML标记。

除了中规中矩的表单处理标签外，Spring还允许你把表单的action URL配置成你想要的形式。令人不快的是这需要更多的输入，因为你是用硬编码方式将URL映射到控制器。

Struts会预先使用contextPath，添加你在web.xml中的定义的后缀(如，*.do)。

```
<html:form action="/user">
```

使用Spring，有一种等同的声明方式看起来和下面的差不多。

```
<form action="<c:url value="/user.html"/>">
```

你也可以使用相对路径，这样Spring版本就Spring版本要少输入一些。

```
<form action="user.html">
```

`<c:url>`版本会预先考虑你的应用程序的`contextPath`(如, `/myusers`), 允许你轻而易举把JSP转移到一个子目录中。另外, 你可以完全控制你想用的扩展名。如果你想在应用中包含有安全和不安全部分, 可以分开处理。你可以在`web.xml`文件中定义`servlet-mappings`(如, `*.html`和`*.secure`), 保护`*.secure`; URL。这在Struts无法实现, 表单`action`的URL总是为你填充。

Struts JSP和Spring JSP另一个不同的地方是, Spring如何将对象值绑定到输入字段上的。Struts的输入标签会查找`<html:form>`的信息, 把属性匹配到`ActionForm`的`getter`。Spring的`<spring:bind>`允许你带有输入字段的`command`对象中绑定`getter`(属性)。我们比较一下`firstName`输入字段不同的语法。如果你在用Struts, 你应该使用`<html:text>`标签。

```
<html:text property="user.firstName"/>
```

使用Spring的话, 语句有点冗长。

```
<spring:bind path="user.firstName">
  <input type="text" name="${status.expression}"
    value="${status.value}"/>
  <span class="fieldError">${status.errorMessage}</span>
</spring:bind>
```

注意

你可以使用`${status.expression}`, 或是属性的名称(例如, `firstName`)。

Struts的`ActionForm` [<http://struts.apache.org/api/org/apache/struts/action/ActionForm.html>]和Spring的`command`对象在功能上很相似。但是Spring允许你绑定到域对象(domain object)上, 结束了开发一个form仅仅是为了处理页面输入的需求(在很多情况下)。只有在一种情况下你需要这种只用于web的form对象, 那就是你准备将两个域对象合并到一个form中时, 或是你想在一个页面放置两个form。

以下是`web/userForm.jsp`的全部代码。

```
<%@ include file="/taglibs.jsp"%>
<title>MyUsers ~ User Details</title>
<p>Please fill in user's information below:</p>
<form method="post" action="<c:url value="/editUser.html"/>"
  onsubmit="return validateUser(this)">
  <spring:bind path="user.id">
    <input type="hidden" name="id" value="${status.value}"/>
  </spring:bind>
  <table>
  <tr>
    <th><fmt:message key="user.firstName"/>:</th>
    <td>
      <spring:bind path="user.firstName">
        <input type="text" name="firstName" value="${status.value}"/>
      </spring:bind>
    </td>
  </tr>
  </table>
</form>
```

```

        <span class="fieldError">${status.errorMessage}</span>
    </spring:bind>
</td>
</tr>
<tr>
    <th><fmt:message key="user.lastName"/>:</th>
    <td>
        <spring:bind path="user.lastName">
            <input type="text" name="lastName" value="${status.value}"/>
            <span class="fieldError">${status.errorMessage}</span>
        </spring:bind>
    </td>
</tr>
<tr>
    <td></td>
    <td>
        <input type="submit" class="button" name="save" value="Save"/>
        <c:if test="${not empty param.id}">
            <input type="submit" class="button" name="delete" value="Delete"/>
        </c:if>
    </td>
</tr>
</table>
</form>
<html:javascript formName="user"/>

```

最后你要做的事添加验证。

为Spring配置Commons Validator

写到这里时，Spring 1.1.1发布了，它没有内置一个声明式的(declarative)验证框架。这个核心的验证框架要求你创建一些类来验证其它类。虽然这不难，我还是更情愿用我在使用Struts时学到的方法：使用Commons Validator，在XML文件中声明规则。Daniel Miller最近添加了在Spring使用Commons Validator的支持
[http://www.sourceforge.net/mailarchive/forum.php?thread_id=3931336&forum_id=28401]。

对于Struts用户，使用他们所熟悉的验证框架能降低Spring MVC的门槛。把validation.xml文件从Struts移植到Spring，你只需要修改几个属性：formName(userForm user)和字段名(删除user，你不再用DynaActionForm包裹它)。这个为Spring修改的版本如下：

```

<form-validation>
    <formset>
        <form name="user">
            <field property="lastName" depends="required">
                <arg0 key="user.lastName"/>
            </field>
        </form>
    </formset>
</form-validation>

```

上面的XML片断中，form的“name”与你在控制器的bean定义中的“commandName”属性值一致。字段的属性值应该与你的输入字段名称一致，键名指向web/WEB-INF/classes/messages.properties中定义的一个i18n键值。更多有关验证规则的信息，请参考Validator用户指南 [http://struts.apache.org/userGuide/dev_validator.html]。

本章的下载包中有一个validation-rules.xml(在web/WEB-INF中)，它与Struts中所带的不同。此文件定义了验证要用到的Spring特定的类和方法，另外还定义了客户端验证要用的JavaScript方法。

userForm.jsp文件底部是一个JSP标签，表示客户端验证要用到的JavaScript。

```
<html:javascript formName="user"/>
```

注意

上面的一行JavaScript标签在使用Commons Validator配置客户端验证时是不推荐使用的。这种方法导致在最终的HTML页面包含了所有的JavaScript函数。第5章讨论一种更为简洁的方法，所有的函数是从一外部文件中引用的。

为了让Spring知道你想使用Commons Validator作为你的验证引擎，在web/WEB-INF/action-servlet.xml添加几个<bean>定义：

```
<bean id="validatorFactory"
      class="org.springframework.validation.commons.DefaultValidatorFactory"
      init-method="init">
  <property name="resources">
    <list>
      <value>/WEB-INF/validator-rules.xml</value>
      <value>/WEB-INF/validation.xml</value>
    </list>
  </property>
</bean>
<bean id="beanValidator"
      class="org.springframework.validation.commons.BeanValidator">
  <property name="validatorFactory">
    <ref local="validatorFactory"/>
  </property>
</bean>
```

第一个bean(validatorFactory)加载了Validator的方法和类的映射(validation-rules.xml)，还有针对应用的验证规则(validation.xml)。第二个bean(beanValidator)应用验证到所有的POJO上。配置UserController使用beanValidator来验证，你只要简单的添加一个validator到UserController定义中。

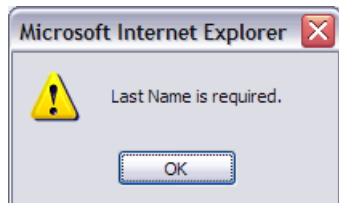
```
<property name="validator"><ref bean="beanValidator"/></property>
```

注意

Spring开发人员计划在未来的几个月中添加他们自己的声明式的验证框架。然而，Commons Validator是目前唯一与Spring兼容的声明式验证框架。目前在Spring的沙箱中有Commons Validator的支持，并讨论在以后把它集成到核心中去。

现在如果你运行`ant deploy reload`，打开浏览器，浏览 `http://localhost:8080/myusers/editUser.html`，尝试添加一个新用户，不要指定last name，你会看到下面的JavaScript警告。

图 4.4. 启用JavaScript运行ant deploy reload测试的结果



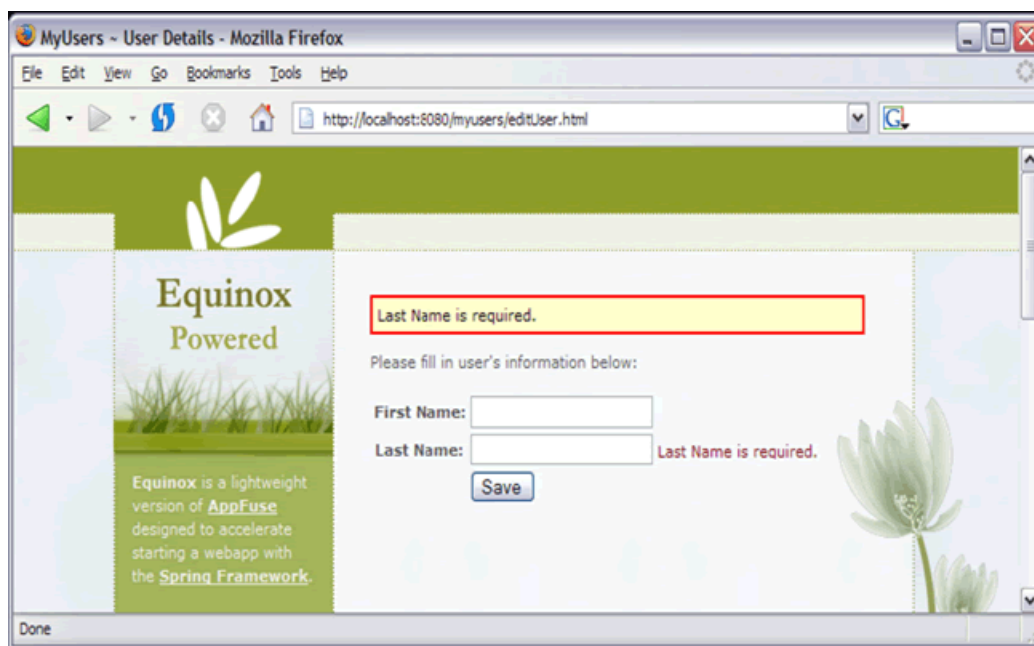
如果你确定一切如你所愿的话，关掉JavaScript，确保服务器端验证正常。在Mozilla Firefox中很简单，只要在Tools > Options > Web Features中，去掉选择“Enable JavaScript”。现在你如果清除lastName字段，保存form，你应该看如下所示。

把所捕捉的错误信息显示在JSP页面上部，在JSP页面底部添加以下的代码段，紧跟<title>。

```
<spring:bind path="user.*">
  <c:if test="${not empty status.errorMessages}">
    <div class="error">
      <c:forEach var="error" items="${status.errorMessages}">
        <c:out value="${error}" escapeXml="false"/><br/>
      </c:forEach>
    </div>
  </c:if>
</spring:bind>
```

如果添加了这段代码，运行`ant deploy`，在不填写lastName(关掉JavaScript)情况下，点击“save”按钮。你的屏幕与下面的截图相似。

图 4.5. 禁用JavaScript运行ant deploy测试的结果



这里回顾一下，集成Commons Validator和配置验证规则，可分为四步：

1. 把spring-sandbox.jar添加到你的classpath中，并定义“validatorFactory”和“beanValidator”。
下
<http://static.springframework.org/spring/docs/2.5.x/docs/spring-framework-libs/spring-validation-1.3.4-RELEASE/spring-validation-1.3.4-RELEASE-bin.zip>，把它放到WEB-INF。
2. 使用<ref bean=”beanValidator”/>作为控制器的“validator”属性。
3. 在WEB-INF/validation.xml定义form验证规则。
4. 在你的JSP文件添加一个<html:javascript>标签。添加一个提交处理方法(针对表单)来启用客户端验证。

你已经创建好了一个web应用程序，使用Spring来处MVC层。恭喜你!

SimpleFormController: 方法生命周期回顾

SimpleFormController是Spring MVC包中众多CommandController中的一种。这些controller设计用于与域对象(domain object)进行交互，动态的绑定请求参数到对象上。与Struts相比，Spring更为简洁，它不要求你的域对象实现某个接口或是继承一个超类。这里不一一介绍每个Command Controller及其功能。我简单的谈一下你可以用的两个：SimpleFormController和AbstractWizardFormController [<http://www.springframework.org/docs/api/org.springframework.web.servlet.mvc/AbstractWizardFormController.html>]。

- SimpleFormController是一个具体的FormController，提供可配置的表单和成功页面视图，和一个方便复写的onSubmit()链(chain)。如果出现验证错误，它会自动重复提交表单视图，如果的合法的提交，则生成成功视图。

- `AbstractWizardFormController` 是一种 `FormController`，用于典型向导式的工作流。与传统的 form 相比，向导拥有多个页面视图。基于这点原因，用户可以用不同的方法进行 `next`，`back`，`cancel` 和 `finish` 操作。

首先，`SimpleFormController` 有点技压群雄的味道。你推荐你阅读一下它的 javadoc [<http://www.springframework.org/docs/api/org.springframework.web.servlet.mvc.SimpleFormController.html>]，其中描述了它的方法生命周期(即工作流)。图4.7演示了一个 GET 请求的生命周期，图4.8演示了一个 POST 请求的生命周期。

图 4.6. GET 请求生命周期

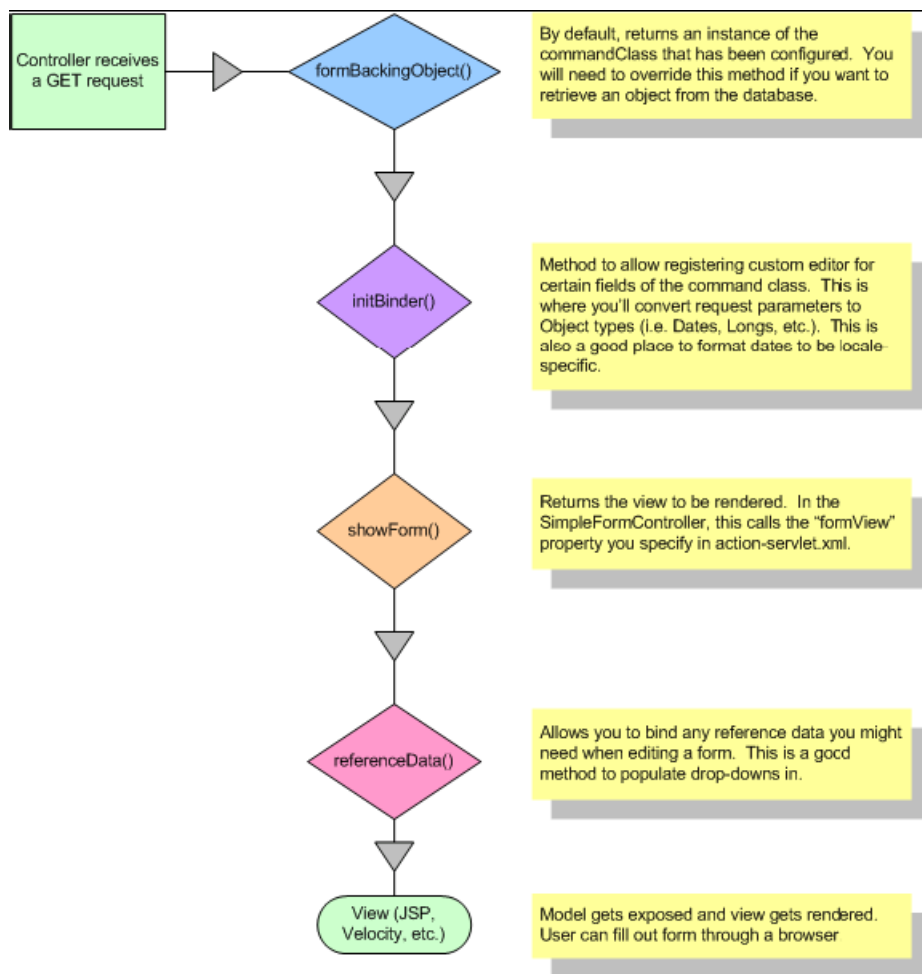
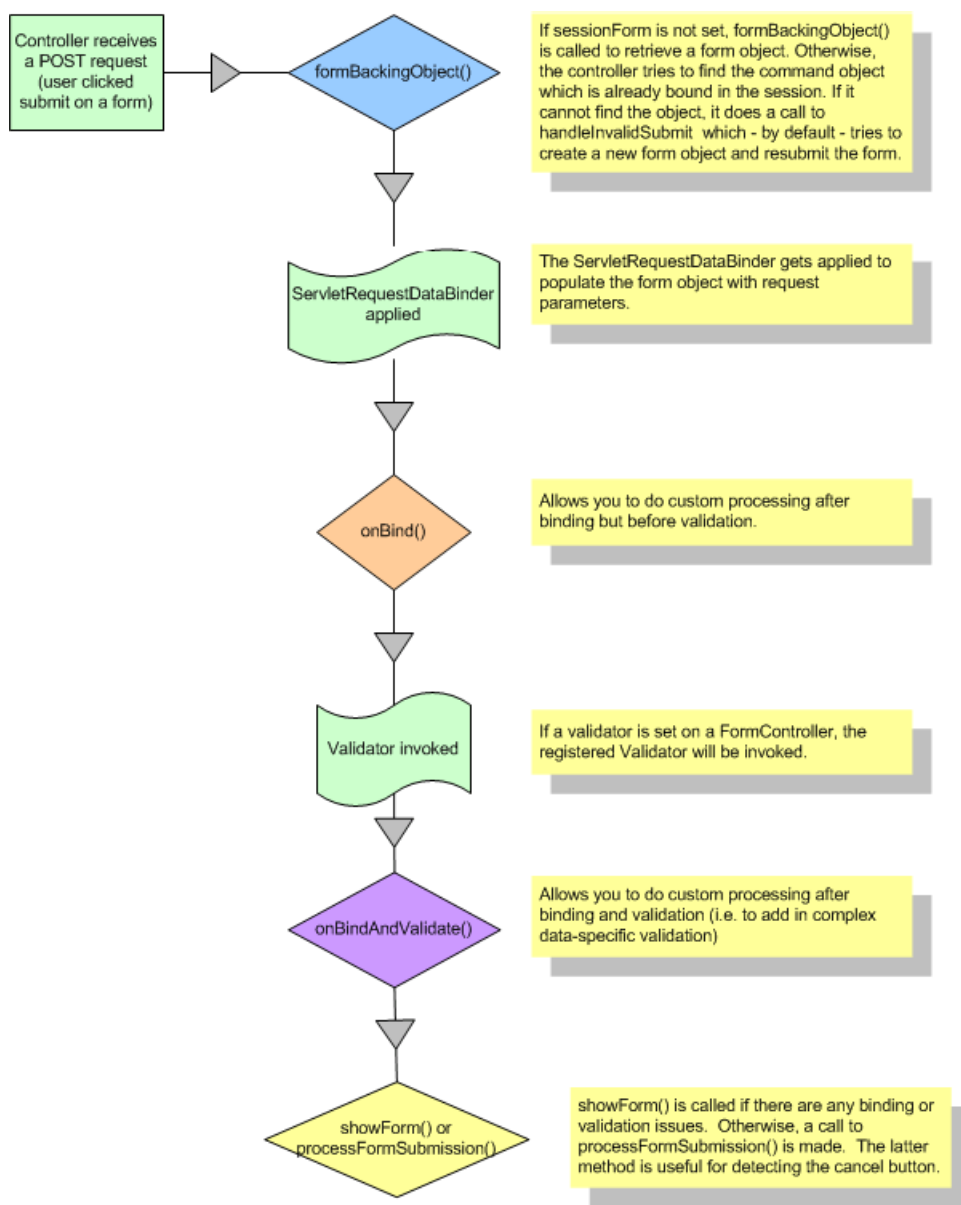


图 4.7. POST请求生命周期



Spring的JSP标签

我们在修改userForm.jsp文件已经接触过<spring:bind>标签；现在我们看一下其它可用的JSP标签。下面是默认情况下，Spring提供的标签列表。

- spring:hasBindErrors支持绑定错误信息到对象上。此标签在检测一个command对象是否有错误信息时经常使用。
- spring:bind把command对象的属性绑定到表单字段上。此标签暴露了一个“status”变量，放在pageContext中。此变量可以被JSP的EL抓到，`${status.value}`代表一个属性值，`${status.errorMessage}`的含义是一个属性相关的任何错误信息。这是在Spring标签库中的最常用的标签。

- `spring:transform`提供转换属性的支持，这些属性不是由`command`对象提供，使用Property Editor 关联到`command`对象。这在你需要从`referenceData()`方法(比如，下拉列表)转换数据时非常有用。此标签必须在`spring:bind`内使用。目前Spring的样例程序还没有用到它。
- `spring:message`与JSTL的`<fmt:message>`类似，但它支持Spring的MessageSource概念。它也可以支持Spring的本土化。JSTL的`<fmt:message>`标签可以胜任Spring MVC的国际化工作。目前Spring的样例程序还没有用到它。
- `spring:htmlEscape`用来设定当前页面的`escape`值。默认为`false`。你也可以在`web.xml`中设定一个`context-param: defaultHtmlEscape`。在JSP页面中使用这个标签会覆盖`web.xml`的设置。目前Spring的样例程序还没有用到它。
- `spring:theme`会查找当前页面范围内的主题(theme)信息。到第5章讨论模板时会更详细的探讨这个标签。

从上面列表中，你可以看出大部分核心标签你都没有用到。但你已经看到了最重要的一个：`spring:bind`。

本章小结

本章涉及内容比较多。讨论了单元测试，使用`spring-mock.jar`来测试Controller类。然后，演示了如何使用Spring MVC把MyUsers应用程序从Struts转向Spring。通过这一过程，你学到了如何进行Controller单元测试,如何在`action-servlet.xml`配置Controller, Handler和Resolver。你也看到了如何利用Spring标签来修改Struts JSP页面，如何使用Commons Validator来添加一个声明式的验证。最后，还讨论SimpleFormController的生命周期和Spring JSP标签。

本章的主要目的是向你演示使用Spring MVC开发web应用程序的一些基础。Spring MVC中我最中意的部分是它的方法生命周期。第5章会探讨更高级的验证和网站页面修饰技术(也叫模板)。你也会学习在Controller中如何处理异常，如何上传一个简单的文件。

第 5 章 高级的MVC框架----使用模板，验证，异常处理，文件上传

如何集成Tiles和SiteMesh进行页面布局，实现验证，处理异常，上传文件，发送邮件

本章涉及了web框架的一些高级主题，特别是验证和页面。演示如何用Tiles和SiteMesh来装饰一个页面，还解释Spring框架如何处理验证，并通过实例演示在web业务层中的运用。最后还解释了在控制器处理异常的一种策略，以及如何上传文件和发送邮件。

几年前，web开发人员还没有什么框架用来协助他们开发基于Java/JSP的应用程序。他们更关心的是完成任务，而不是把它做好。今天，无数的框架的涌现大大提高开发人员的效率。他们拥有内置的页面修饰，验证引擎，异常处理和文件上传技术。有的甚至支持拦截器(interceptor)，它能中断一个web请求，即时的执行逻辑处理。

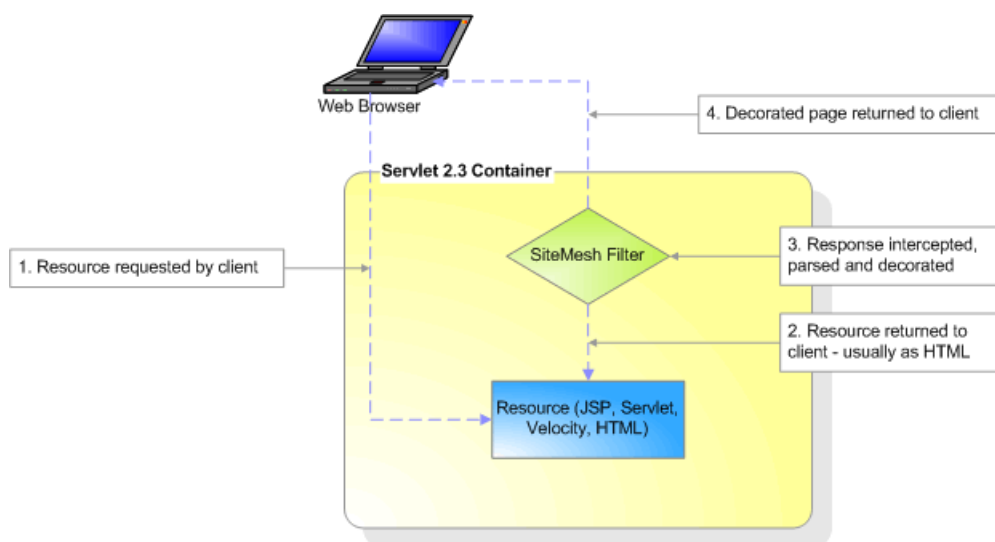
Spring就支持所有的这些特性。本章探讨在基于Spring的web应用程序中，如何配置页面修饰框架如SiteMesh和Tiles。然后演示如何以现有的Struts Validator知识为基础，在Spring中使用Commons Validator(支持客户端使用JavaScript验证)。介绍了验证之后，本章还涉及了异常处理，应用拦截器和上传文件。最后还简短的介绍了发送邮件。

听起来本章要完成的任务很多，但是都很简单，容易理解。另外，所有的这些技术和概念都会在MyUsers程序中表现出来。到本章结束时，你已经拥有一个使用了各种特性的参考程序。

SiteMesh模板技术

SiteMesh是来自OpenSymphony [<http://www.opensymphony.com/>]项目中的一个开源的页面布局和修饰技术框架。从它开始诞生到现在，5年时间过去了，当时Joe Walnes下载了sun的第1个servlet引擎，并用servlet chain开发了它。多年过去了，基于的设计还是一样的：内容是被截取和解析的，一个装饰映射器(mapper)查找一个装饰器(decorator)，将所有的东西合并到一起。下面的图表演示了其工作原理的最简单的例子。

图 5.1. SiteMesh流程



换肤(skinning)是每个web程序一个基本的要素。它能够通过修改一两个文件来改变整个web程序的外观，大大的提高了维护性。SiteMesh是一个简单的修饰框架，很容易安装和配置。

安装和配置

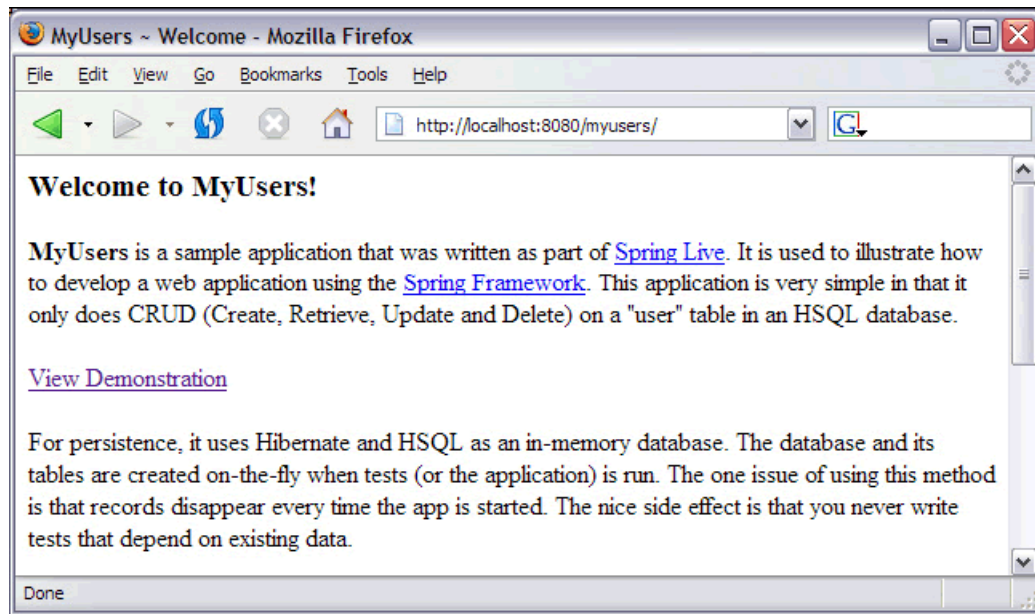
如果你正在使用第4章中开发的程序，SiteMesh已经配置好了。要想从头开始(不包括SiteMesh)，你必须从<http://sourcebeat.com/downloads>下载“MyUsers Chapter 5”捆绑包。在下载的程序中，本章所有的议题都没有进行配置。下载后，解压到“myusers-ch5”，复制一份myusersch5，命名为myusers-sitemesh。

注意

当你安装和配置Tiles时，你会复制一份myusers-ch5为myusers-tiles。

运行`ant remove clean install`(Tomcat处于运行状态)，当你打开<http://localhost:8080/myusers/>时，你的屏幕应该和下面类似。与前面你看到的屏幕相比，这个有很大不同，显得比较单调。安装和配置好SiteMesh能让它更好看一些。

图 5.2. 未经装饰的“Welcome to MyUsers”页面



第1步：在web.xml中配置SiteMesh

sitemesh-2.1.jar文件应该已经在myusers-sitemesh下web/WEB-INF/lib目录里，所以你没有必要安装它。但是，如果你是从头安装SiteMesh，你可能要从<http://www.opensymphony.com/sitemesh> [http://www.opensymphony.com/sitemesh] 上下载。

1. 打开web.xml(在web/WEB-INF)进行编辑。
2. 在文件的顶部，<display-name>之后，<context-param>之前，添加以下的<filter>定义。

```
<filter>
  <filter-name>sitemesh</filter-name>
```

```
<filter-class>
  com.opensymphony.module.sitemesh.filter.PageFilter
</filter-class>
</filter>
```

3. 在<context-param>之后，listener元素之前，添加以下<filtermapping>。

```
<filter-mapping>
  <filter-name>sitemesh</filter-name>
  <url-pattern>/*</url-pattern>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

<dispatcher>元素和<filter-mapping>是servlet2.4新增的，这些dispatcher(还有ERROR)允许过滤器映射到请求外的URL上。

第2步：创建配置文件

1. 在web/WEB-INF目录中新建sitemesh.xml文件，内容如下：

```
<sitemesh>
  <page-parsers>
    <parser default="true"
      class="com.opensymphony.module.sitemesh.parser.FastPageParser"/>
    <parser content-type="text/html"
      class="com.opensymphony.module.sitemesh.parser.FastPageParser"/>
    <parser content-type="text/html;charset=ISO-8859-1"
      class="com.opensymphony.module.sitemesh.parser.FastPageParser"/>
  </page-parsers>
  <decorator-mappers>
    <mapper
      class="com.opensymphony.module.sitemesh.mapper.
        ConfigDecoratorMapper">
      <param name="config" value="/WEB-INF/decorators.xml"/>
    </mapper>
  </decorator-mappers>
</sitemesh>
```

此文件配置了page-parser和decorator-mapper。page-parser用来解析特定的content-type，你可能不会再修改这些值。ConfigDecoratorMapper是几个decorator-mapper中的一种。它让SiteMesh从配置属性中取decorator和mapping，默认此文件是/WEB-INF/decorators.xml。因为你将使用默认方式，你可以从sitemesh.xml中删除<param>元素，一切都会照常工作。decorators.xml指定了URL模式和它使用的装饰器。

2. 在web/WEB-INF目录下新建decorators.xml，填充以下的XML代码。

```
<decorators defaultdir="/decorators">
```

```
<decorator name="default" page="default.jsp">
  <pattern>/*</pattern>
</decorator>
</decorators>
```

创建装饰器

最后，构造一个装饰器作为模板来包装应用中的页面。如果你熟悉Tiles，要做的事是一样的，就是创建一个基础页面布局(base layout)。

1. 创建web/decorators/default.jsp页面，已经在decorators.xml文件中配置好了。
2. 在default.jsp文件中，添加以下代码。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<%@ include file="/taglibs.jsp"%>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
    <title><decorator:title default="MyUsers"/></title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8"/>
    <c:set var="ctx" value="{pageContext.request.contextPath}"/>
    <link href="{ctx}/styles/global.css" type="text/css" rel="stylesheet"/>
    <link href="{ctx}/images/favicon.ico" rel="SHORTCUT ICON"/>
    <decorator:head/>
    <!-- HTML & Design contributed by Boer Attila (http://www.calcium.ro) -->
    <!-- Found at http://www.csszengarden.com/?cssfile=/083/083.css&page=2 -->
  </head>
  <body>
    <a name="top"></a>
    <div id="container">
      <div id="intro">
        <div id="pageHeader">
          <h1><span>Welcome to Equinox</span></h1>
          <div id="logo" onclick="location.href='<c:url value="/" />'"
            onkeypress="location.href='<c:url value="/" />'"></div>
          <h2><span>Spring Rocks!</span></h2>
        </div>
        <div id="quickSummary">
          <p>
            <strong>Equinox</strong> is a lightweight version of
            <a href="http://raibledesigns.com/appfuse">AppFuse</a> designed
            to accelerate starting a webapp with the
            <a href="http://www.springframework.org">Spring Framework</a>.
          </p>
          <p class="credit">
            <a href="http://www.csszengarden.com/?cssfile=/083/083.css">
              Design and CSS</a> donated by <a href="http://www.calcium.ro">
                Boér Attila</a>.
          </p>
        </div>
      </div>
    </div>
```

```
<div id="content">
  <%@ include file="/messages.jsp"%>
  <decorator:body/>
</div>
</div>
<div id="supportingText">
<div id="underground">
  <decorator:getProperty property="page.underground"/>
</div>
<div id="footer"></div>
</div>
<div id="linkList">
  <div id="linkList2">
    </div>
  </div>
</div>
</body>
</html>
```

这里考察的重要元素是<decorator:*>，用黄色高亮显示([译注]：原文没有黄色高亮显示)。在文档的头部，一个<decorator:title>从所包装的JSP页面中获得<title>，<decorator:head/>装入<head>中的定义的所有的内容。在中间，用一个<decorator:body/>来装载页面的body。最后，用一个<decorator:getProperty>来装入修饰器中定义的<content>标签。这里有一个index.jsp页面的样例。

```
<content tag="underground">
  <h3>Additional Information</h3>
  <!-- more content here -->
</content>
```

3. 在web/taglibs.jsp文件中添加<decorator>标签库。把下面一行添加到此文件中

```
<%@ taglib uri="http://www.opensymphony.com/sitemesh/decorator" prefix="decorator"%>
```

4. 运行ant deploy reload，打开浏览器浏览http://localhost:8080/myusers。你应该可以看到和下面相似的比较清爽的页面。

图 5.3. 经过修饰的“Welcome to MyUsers”页面



这是一个简洁但很有吸引力的设计方案。对大部分来说，你的修饰页面可以用标准的HTML元素。但不需要用到所有静态页面必须的元素(如，<head>，<body>)。更妙的是，SiteMesh不关心你在你的应用程序中采用了哪种服务器端技术，它能很好的结合CGI，PHP，Servlet，Velocity和FreeMarker。Spring支持多种视图技术，这样你能感到Spring非常友好。

Tiles模板技术

Tiles是一种和SiteMesh非常相似的模板和文档布局引擎。Tiles是Struts默认的布局引擎，大多数SiteMesh用户倾向于使用WebWork(主要是WebWork和SiteMesh同样来自OpenSymphony)，Struts最初是由Cedric Dumoulin编写的，并且在Struts 1.0之后很短的时间就发布了。开始，它是Struts一个独立的插件，和Validator非常相似。Struts 1.1的很多工作就是把一些有用的组件从Struts中剥离出来，放入Jakarta Commons项目中。但是没有人来分离Struts，以致它没有成为一个单独的项目，而成了Struts核心的一部分(在struts.jar中)。

接下来的几节演示结合Spring框架使用时，如何配置Tiles。

安装和配置

下面的介绍针对Struts 1.1，已经集成了Tiles，所以它没有自己的版本号。

复制myusers-ch5为myusers-tiles。如果你还没有下载myusers-ch5，请从<http://sourcebeat.com/downloads>上下载。此时如果你运行ant remove clean install，你的屏幕看起来和进行SiteMesh那一段操作之前的类似(图5.2)。

第1步：配置Spring让它能够识别Tiles

1. 为了让Spring能识别Tiles，并使用它来生成视图。在web/WEB-INF目录下的action-servlet.xml中新建一个tilesConfigurer bean定义，如下所示。

```
<bean id="tilesConfigurer"
      class="org.springframework.web.servlet.view.tiles.TilesConfigurer">
  <property name="factoryClass">
    <value>org.apache.struts.tiles.xmlDefinition.I18nFactorySet</value>
  </property>
  <property name="definitions">
    <list>
      <value>/WEB-INF/tiles-config.xml</value>
    </list>
  </property>
</bean>
```

2. 修改viewResolver bean的viewClass属性，把 JstlView [http://www.springframework.org/docs/api/org/springframework/web/servlet/view/JstlView.html] 修改为 TilesJstlView [http://www.springframework.org/docs/api/org/springframework/web/servlet/view/tiles/TilesJstlView.html]。你还可以删除这个bean定义中的prefix和suffix值。下面是经过替换的viewResolver bean定义。

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.
        InternalResourceViewResolver">
  <property name="requestContextAttribute">
    <value>rc</value>
  </property>
  <property name="viewClass">
    <value>org.springframework.web.servlet.view.tiles.
      TilesJstlView</ value>
  </property>
</bean>
```

现在TilesJstlView能根据定义解析视图名称。例如，在UserController中，从handleRequest()方法返回一个userList视图。它会把userList定义呈现出来。

```
return new ModelAndView("userList", "users", mgr.getUsers());
```

另一个例子是UserFormController中的formView。在action-servlet.xml把它设置为userForm，就会生成userForm定义内容。

第2步：创建基础布局

现在你必须创建一个含有基础布局的JSP文件。这基本上是一个模板(相当于SiteMesh的修饰器)，用来控制页面的布局，以及在哪个位置插入组件。

1. 新建web/layouts/baseLayout.jsp文件。在文件中填充以下代码。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<%@ include file="/taglibs.jsp"%>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
    <title><tiles:getAsString name="title"/></title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8"/>
    <c:set var="ctx" value="${pageContext.request.contextPath}"/>
    <link href="${ctx}/styles/global.css" type="text/css" rel="stylesheet"/>
    <link href="${ctx}/images/favicon.ico" rel="SHORTCUT ICON"/>
    <!-- HTML & Design contributed by Boer Attila (http://www.calcium.ro) -->
    <!-- Found at http://www.csszengarden.com/?cssfile=/083/083.css&page=2 -->
  </head>
  <body>
    <a name="top"></a>
    <div id="container">
      <div id="intro">
        <div id="pageHeader">
          <h1><span>Welcome to Equinox</span></h1>
          <div id="logo" onclick="location.href='<c:url value="/" />'"
            onkeypress="location.href='<c:url value="/" />'"></div>
          <h2><span>Spring Rocks!</span></h2>
        </div>
        <div id="quickSummary">
          <p>
            <strong>Equinox</strong> is a lightweight version of
            <a href="http://raibledesigns.com/appfuse">AppFuse</a> designed
            to accelerate starting a webapp with the
            <a href="http://www.springframework.org">Spring Framework</a>.
          </p>
          <p class="credit">
            <a href="http://www.csszengarden.com/?cssfile=/083/083.css">
              Design and CSS</a> donated by <a href="http://www.calcium.ro">
                Boér Attila</a>.
          </p>
        </div>
        <div id="content">
          <%@ include file="/messages.jsp"%>
          <tiles:insert attribute="content"/>
        </div>
      </div>
      <div id="supportingText">
        <div id="underground">
          <c:out value="${underground}" escapeXml="false"/>
        </div>
        <div id="footer"></div>
      </div>
    </div>
  </body>
</html>
```

```
</div>
<div id="linkList">
  <div id="linkList2">
  </div>
</div>
</div>
</body>
</html>
```

此文件和你为SiteMesh创建的文件web/decorators/default.jsp非常类似，除了这里使用tiles;JSP标签替换掉decorator标签。

2. 基于这个原因，你必须把Tiles标签添加到web/taglibs.jsp文件中。修改后，这个文件看起来应该与下面差不多。

```
<%@ page language="java" errorPage="/error.jsp" %>
<%@ taglib uri="http://java.sun.com/jstl/core_rt" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
<%@ taglib uri="http://www.springframework.org/tags/commons-validator"
prefix="html" %>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<%@ taglib uri="http://jakarta.apache.org/struts/tags-tiles"
prefix="tiles" %>
```

第3步：创建页面定义

Tiles支持两种方法配置页面定义：在某一JSP页面中配置一个页面的定义，或是用一个XML文件配置每个页面的定义。第2方法为所关心的内容提供了一种更为清晰的分离方式，让JSP页面在不知情的情况下使用Tiles。

1. 为MyUsers创建页面定义，在web/WEB-INF目录下新建了一个文件tiles-config.xml，填充以下XML内容。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE tiles-definitions PUBLIC
"-//Apache Software Foundation//DTD Tiles Configuration 1.1//EN"
"http://jakarta.apache.org/struts/dtds/tiles-config_1_1.dtd">
<tiles-definitions>
  <!-- base layout definition -->
  <definition name="baseLayout" path="/layouts/baseLayout.jsp">
    <put name="title" value="MyUsers"/>
  </definition>
  <!-- index definition -->
  <definition name="index" extends="baseLayout">
    <put name="title" value="MyUsers ~ Welcome"/>
    <put name="content" value="/index.jsp"/>
  </definition>
  <!-- user list definition -->
  <definition name="userList" extends="baseLayout">
```

```
<put name="title" value="MyUsers ~ User List"/>
<put name="content" value="/userList.jsp"/>
</definition>
<!-- user form definition -->
<definition name="userForm" extends="baseLayout">
  <put name="title" value="MyUsers ~ User Details"/>
  <put name="content" value="/userForm.jsp"/>
</definition>
</tiles-definitions>
```

上面的页面定义了页面标题，代替了在JSP中定义。

2. 为了在你的应用程序中产生洁净的HTML代码，删除web目录下userList.jsp和userForm.jsp文件中<title>标签。使用SiteMesh时找不到更好的方法来控制JSP页面的标题。
3. 配置Spring，添加一个bean定义，以便解析URL到对应的Tiles定义。在文件action-servlet.xml中声明 `UrlFilenameViewController` [[http://www.springframework.org/docs/api/org.springframework.web.servlet.mvc/UrlFilenameViewController.html](http://www.springframework.org/docs/api/org.springframework.web.servlet.mvc.UrlFilenameViewController.html)]，如下所示。

```
<bean id="filenameController"
      class="org.springframework.web.servlet.mvc.
      UrlFilenameViewController"/>
```

4. 为了生成MyUsers的主页面(/index.html)，配置urlMapping bean，添加一条额外的映射：

```
<prop key="/index.html">filenameController</prop>
```

5. 运行ant remove clean install，当你浏览<http://localhost:8080/myusers/users.html>时，会看到与SiteMesh相似的情景(图5.3)。

现在如果浏览<http://localhost:8080/myusers>时会碰到一个问题，它会显示没有经过修饰的index.jsp页面。

6. 重命名index.jsp为welcome.jsp来解决这个问题。新建一个index.jsp文件，填充以下内容。

```
<%@ include file="/taglibs.jsp"%>
<tiles:insert definition="index"/>
```

7. 修改tiles-config.xml使用welcome.jsp作为content页面。

```
<definition name="index" extends="baseLayout">
  <put name="title" value="MyUsers ~ Welcome"/>
```

```
<put name="content" value="/welcome.jsp"/>
</definition>
```

虽然上面的方案可以运行，但是像这样用两个JSP页面来解决问题实在很痛苦。所以我建议使用一个servlet过滤器来重定向某些特定的URL到其它URL上。使用这种方案，你必须配置你的应用程序，以便根URL能调用index.html URL。你在web.xml中的使用<welcome-file-list>无法达到目的，Paul Tuckey的URL Rewrite Filter [<https://urlrewrite.dev.java.net/>]使之成为可能。

注意

Paul Tuckey的Rewrite Filter是模仿Apache HTTP服务器的mod_rewrite设计的。为了生成干净的URL链接，检测浏览器，或是优雅地处理被移动的内容，它会重定向或转发请求的URL。

urlrewrite-1.2.jar已经存在于web/WEB-INF/lib目录下，你只需要在web.xml进行配置，然后添加它的配置文件。

8. 打开web.xml文件，添加以下<filter>定义。

```
<filter>
  <filter-name>UrlRewriteFilter</filter-name>
  <filter-class>
    org.tuckey.web.filters.urlrewrite.UrlRewriteFilter
  </filter-class>
</filter>
```

9. 添加它的映射。

```
<filter-mapping>
  <filter-name>UrlRewriteFilter</filter-name>
  <url-pattern>/*</url-pattern>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

10. 在web/WEB-INF目录中新建一个urlrewrite.xml来配置这个过滤器的规则。

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE urlrewrite PUBLIC "-//tuckey.org//DTD UrlRewrite 1.0//EN"
"http://tuckey.org/res/dtd/urlrewritel.dtd">
<urlrewrite>
  <rule>
    <from>/${</from>
    <to type="forward">index.html</to>
  </rule>
  <rule>
    <from>/index.jsp</from>
    <to type="forward">index.html</to>
```

```
</rule>
</urlrewrite>
```

此配置会把<http://localhost:8080/myusers>和<http://localhost:8080/myusers/index.jsp>发送到<http://localhost:8080/myusers/index.html>，从而调用Spring的 `DispatcherServlet` [<http://www.springframework.org/docs/api/org.springframework.web.servlet.DispatcherServlet.html>]。 `/index.html` 会调用 `filenameController` bean，它利用这条URL来判定它需要生成的index定义。

最后，你必须修正 `index.jsp`，将 `underground` 设置成一个请求变量，以便能获取定义内容。使用 `SiteMesh` 时，你可以用 `<content>` 的JSP文件和修饰器中的 `<decorator:getProperty>` 来设置内容。`Tiles` 没有类似的功能，你可以使用JSTL设置请求属性来摹拟这种方式。

11. 打开 `web/index.jsp` 文件，把 `<content tag="...">...</content>` 修改成：

```
<c:set var="underground" scope="request">
...
</c:set>
```

这段文字内容将由 `layouts/baseLayout.jsp` 文件中的一行代码来填充和显示。

```
<c:out value="${underground}" escapeXml="false"/>
```

注意

如果你不愿意用私有的 `<content>` 标签，这种方案同样适用于 `SiteMesh`。

上面两段内容向你介绍了配置两种最流行的布局和修饰引擎一些必要的知识。两种都能Spring MVC 框架很好的结合，并且容易配置(特别是你有了本指南)。我推荐使用 `SiteMesh`，它更容易协同工作，特别是针对新应用。当然这主要取决你喜欢哪一个。

Spring验证方法

目前，Spring没有提供 `Commons Validator` 来支持 `MyUsers` 中使用的设置。然而，如果你不打算使用 `Commons Validator`，它有一种非常简单的验证系统可供你使用。你要做的是写一个类实现 `org.springframework.validation.Validator` 接口，它包含以下方法。

```
/**
 * Return whether or not this object can validate objects
 * of the given class.
 */
boolean supports(Class clazz);
/**
 * Validate an object, which must be of a class for which
 * the supports() method returned true.
 * @param obj Populated object to validate
 * @param errors Errors object we're building. May contain
 * errors for this field relating to types.
 */
```

```
void validate(Object obj, Errors errors);
```

1. 在userForm.jsp文件中禁用Commons Validator。只需要简单的从<form>标签中删除onsubmit属性。

```
<form method="post" action="<c:url value="/editUser.html"/>">
```

2. 在src/org/appfuse/web目录中新建一个名为UserValidator的类。

```
package org.appfuse.web;
// use your IDE to organize imports

public class UserValidator implements Validator {
    private Log log = LogFactory.getLog(UserValidator.class);

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

    public void validate(Object obj, Errors errors) {
        if (log.isDebugEnabled()) {
            log.debug("entering 'validate' method...");
        }
        User user = (User) obj;
        ValidationUtils.rejectIfEmptyOrWhitespace(errors,
            "lastName", "errors.required", "Value required.");
    }
}
```

在上述的代码中, 这个Validator仅支持User类, 其validate()在lastName为空返回一个错误。

3. 在action-servlet.xml文件中, 添加以下bean定义来配置这个Validator。

```
<bean id="userValidator" class="org.appfuse.web.UserValidator"/>
```

4. 修改userFormControllerbean的validator属性, 用userValidator替换beanValidator。

```
<property name="validator"><ref bean="userValidator"/></property>
```

5. 运行`ant deploy reload`，添加一个新用户，不要填写`lastName`。为了证明`UserValidator`被调用了，检查进入方法时的调试信息日志。当你准备进行更为复杂的验证(例如比较属性值和数据库值)，这种验证机制功能非常强大。

下一节阐述在Spring应用程序中设置`Commons Validator`，并且使用它来验证`lastName`字段不能为空。然后演示如何用`Commons Validator`的声明式的验证方式和`Validator`接口来创建一个验证链(validation chain)。

使用Commons Validator

第4章中已经详细的解释了如何使用`Commons Validator`的声明式验证框架，所以我不打算重复这些细节。但这是一个有关验证的章节，这里提供一个step-by-step的总览，以明示你要做什么。

1. 在`web/WEB-INF`中新建一个`validation.xml`，在其中定义你的验证规则。
2. 下载
[https://appfuse.dev.java.net/nonav/source/browse/*checkout*/appfuse/extras/spring/lib/spring-1.0.2/validator-rules.xml]
针对Spring的`validation-rules.xml`文件，安装在`web/WEB-INF`中。这个文件已经包含在这一章下载的捆绑包中。
3. 添加`validatorFactory`和`beanValidator` bean定义到文件`web/WEB-INF/actionservlet.xml`。
4. 配置`FormController` bean，使用`beanValidator`作为它`validator`属性。

这些步骤可以启用服务器端验证，那客户端验证呢？第4章的方法是可行的，但是它在页面中包含了所有的JavaScript验证函数。更好的方法是引用一个独立的JavaScript文件，这样用户的浏览器可以进行缓存。以下步骤假定你没有为你的表单配置好客户端验证。

1. 在你想启用验证的`<form>`中添加`onsubmit`属性。

```
<form method="post" action="<c:url value="/editUser.html"/>"
onsubmit="return validateUser(this)">
```

2. 在表单的底部，添加几行代码，写入JavaScript函数调用表单的规则，包含单独的JavaScript文件。如果你在MyUsers进行尝试，请确保替换`<html:javascript>`。

```
<html:javascript formName="user"
staticJavascript="false" xhtml="true" cdata="false"/>
<script type="text/javascript"
src="<c:url value="/scripts/validator.jsp"/>"></script>
```

3. 在`web/scripts`(你必须创建`scripts`目录)创建`validator.jsp`，写入以下代码。

```
<%@ page language="java" contentType="javascript/x-javascript" %>
<%@ taglib uri="http://www.springframework.org/tags/commons-validator"
prefix="html" %>
```

```
<html:javascript dynamicJavascript="false" staticJavascript="true"/>
```

使用Commons Validator最麻烦的部分是设置和配置的过程。一旦你找到替代方法，创建规则非常简单。你甚至可以用XDoclet从你的POJO中生成这些规则。

XDoclet

XDoclet [<http://xdoclet.sourceforge.net/>]是一种开放源码的代码生成引擎。它为Java提供了面向属性的编程方法(Attribute-Oriented Programming)。这意味着通过在你的代码中添加元数据(metadata)(属性)来添加更多的意义。这在一些特殊的JavaDoc标签中完成。更多信息请参考XDoclet官方网站 [<http://xdoclet.sourceforge.net/xdoclet/resources.html>]。XDoclet使用的元数据属性也叫做注释(annotation)。这种理念得到了很多的褒扬和应用，以至注释作为一种新的特性添加到J2SE 5中。

写到这里时，XDoclet发行版本中还没有包含这样的功能，但是很已经添加到了XDoclet的CVS中。

1. 为了能从POJO中生成验证规则，定义一个<webdoclet>任务(task)，调用<springvalidationxml>任务。下面提供一个样例：

```
<target name="webdoclet"
description="Generate web deployment descriptors">
  <taskdef name="webdoclet"
    classname="xdoclet.modules.web.WebDocletTask">
    <classpath>
      <path refid="xdoclet.classpath"/>
    </classpath>
  </taskdef>
  <webdoclet destdir="${webapp.target}/WEB-INF"
    force="${xdoclet.force}"
    mergedir="metadata/web"
    excludedtags="@version,@author"
    verbose="true">
    <fileset dir="src"/>
    <springvalidationxml/>
  </webdoclet>
</target>
```

2. 在你的POJO的setter中添加@spring.validator标签，如下所示：

```
/**
 * @spring.validator type="required"
 */
public void setLastName(String lastName) {
    this.lastName = lastName;
}
```

上面的代码将会生成validation.xml文件，与你正在使用的MyUsers中的文件完全相同，它要求lastName为必填(required)字段。这个例子演示了XDoclet使声明式的验证变得多么简单。

如果你想查找更多有关XDoclet的信息，请参考AppFuse [<http://appfuse.dev.java.net>]或是从CVS构建XDoclet [http://xdoclet.sourceforge.net/install.html#Getting_XDoclet_from_CVS]。

一系列的验证器

前面两个例子是在`userFormController`中设置`validator`属性。这个属性所参考的bean引用自定义的`userValidator`或`Commons Validator`的`beanValidator`，后者是从XML文件中读取规则。使用Spring MVC，在设置`validators`属性时实际上可以添加多个验证器，如下所示：

```
<property name="validators">
  <list>
    <ref bean="beanValidator"/>
    <ref bean="userValidator"/>
  </list>
</property>
```

这样就创建好了一个验证链(validation chain)，可以使用`Commons Validator`实现简单的验证，或是利用一个自定义的`Validator`实现来完成更为复杂的验证。

在业务委派中进行验证

web层中的验证看起来是经常干的事，但同样需要对业务层进行验证。下面是一个在你的中间层如何使用Spring验证的一个简单的例子。

1. 在`UserManagerImpl.java`文件中，把`saveUser()`方法改成：

```
public User saveUser(User user) {
    BindException errors = new BindException(user, "user");
    new UserValidator().validate(user, errors);
    if (errors.hasErrors()) {
        throw new RuntimeException("validation failed!", errors);
    }
    dao.saveUser(user);
    return user;
}
```

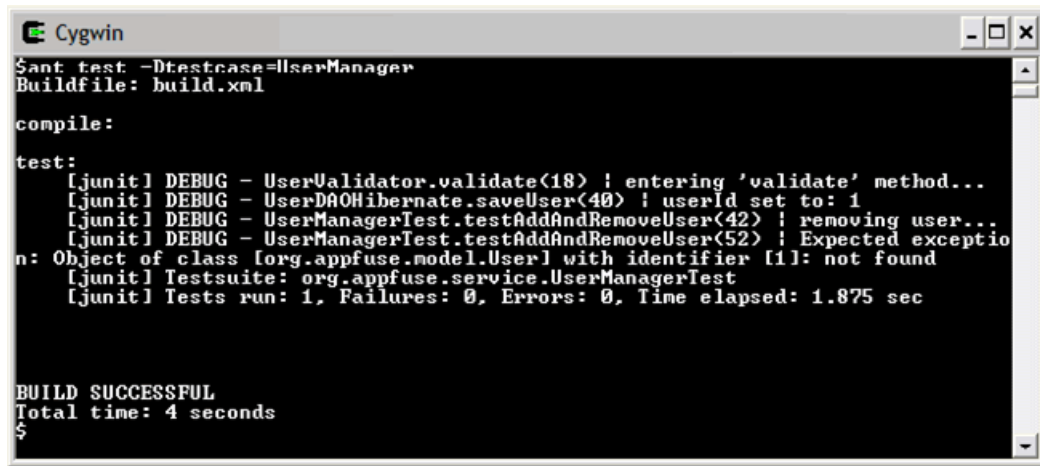
2. 在`UserManagerTest.java`中写一个单元测试来验证抛出的异常。

```
public void testWithValidationErrors() {
    user = new User();
    user.setFirstName("Bill");
    try {
        user = mgr.saveUser(user);
        fail("Validation exception not thrown!");
    } catch (Exception e) {
        log.debug(e.getCause().getMessage());
        assertNotNull(e.getCause());
    }
}
```

```
}
```

3. 运行`ant test -Dtestcase=UserManager`，你会看到图5.4类似的输出结果。

图 5.4. 运行`ant test -Dtestcase=UserManager`命令的结果



```
Cygwin
$ ant test -Dtestcase=UserManager
Buildfile: build.xml

compile:

test:
[junit] DEBUG - UserValidator.validate(18) ! entering 'validate' method...
[junit] DEBUG - UserDaoHibernate.saveUser(40) ! userId set to: 1
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(42) ! removing user...
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(52) ! Expected exception:
n: Object of class [org.appfuse.model.User] with identifier [1]: not found
[junit] Testsuite: org.appfuse.service.UserManagerTest
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 1.875 sec

BUILD SUCCESSFUL
Total time: 4 seconds
$
```

本例是一个纯编码的验证器。但是你完全没有必要用纯编码的方式处理你所用的验证器。你可以添加一个`setValidator()`方法到`UserManagerImpl`(还有它的接口)中，然后使用依赖注射，在`applicationContext.xml`文件中显式的设置`validator`属性。要完成这项工作，你可以在`applicationContext.xml`中声明你的`validator` bean，或者在测试中加载`action-servlet.xml`。总而言之，比在web层中配置和使用验证要简单。

Spring未来的声明式验证框架

Commons Validator是Spring MVC唯一支持的声明式验证框架。Keith Donald(一名Spring开发人员)正在努力开发一种更为内置和健壮的验证框架。我请求他能提供一些这方面的细节，这里是一些他发给我主要特性/不同之处。

- 用一个简单方便的接口定义新的验证的规则(规则提供方只需实现一个唯一的`boolean test(argument)`方法)。
- 支持bean属性表达式(例如，`minProperty`必须小于`maxProperty`)。属性访问策略将是可拔插的，并且不限于javabean(例如，允许支持映射的存储系统，或者在胖客户端(rich client side)缓存form objects)。
- 支持复杂的嵌套的表达式(and/or/not)及相关的操作符(>, >=, <, <=, !=, ==)。
- 支持应用各种基于上下文或是用例的规则设置。
- 有一个报告子系统能描述规则的结构，执行验证，并且获取/生成错误信息结果。这样你可以即时的组装规则，而不必在写死一些静态的信息。报告器(reporter)能从基本的结构中自动生成规则信息。
- 报告字段输入提示(一个字段相关的规则让用户知道他们期望输入什么)。
- 集成Spring Rich Client Platform(RCP)和Spring MVC;环境。

从上面的列表中，你可以看到Spring中的验证的前景是光明的。

控制器中的异常处理

异常处理是每个web应用所要面临的问题。Servlet API提供了一种简单的机制，把某种特定的异常和错误代码映射到特定的视图(view)。例如，在Equinox中，web.xml文件中下面的语句，意思是说“if a page is not found(如果一个页面没找到)”就转向404.jsp页面。

```
<error-page>
  <error-code>404</error-code>
  <location>/404.jsp</location>
</error-page>
```

下面的XML代码表示任何“500: Internal Server Errors(服务器内部错误)”都会转向error.jsp。如果你在开发应用的过程中遇到大量的500错误，应该考虑更好的异常处理方式。

```
<error-page>
  <error-code>500</error-code>
  <location>/error.jsp</location>
</error-page>
```

除了在web.xml中定义错误页面外，当JSP页面出现异常时，让它们转向一个错误页面，也是个不错的主意。MyUsers应用程序是这么做的，在web/taglibs.jsp文件中，定义了errorPage="/error.jsp"。在web.xml文件中，你还可以利用<exception-type>映射来声明某种异常转向某种特定的页面。

```
<error-page>
  <exception-type>
    org.appfuse.service.UserNotFound
  </exception-type>
  <location>/userNotFound.html</location>
</error-page>
```

警告

在Tomcat中使用SiteMesh有一个bug，它不会修饰你在web.xml文件中定义的<error-page>映射文件。如果你在web.xml文件中映射异常，要保证这些文件在不进行修饰的情况下，能独立运行。

虽然Servlet API提供了一种不错的异常处理途径，但是很难从异常中分离出一些信息，并针对这些信息进行逻辑处理。很难在Controller中放置try/catch语句，因为它们会占据几行的空间。更为简洁的是调用一个业务委派的方法。在你的Controller中避免try/catch的一种简单的途径是在一个FormController的onSubmit()方法后添加throws Exception。在Struts类似的框架中，你可以在Action完成这项操作，显示的将Exception映射到视图(view)。换句话说，在抛出异常时，你可以(在XML)进行指定，用户可以是转向到一个Action，一个Tiles定义，或是JSP页面。

使用Spring时也可以把一个指定的异常转向一个特定的视图。最简单的方法是在`action-servlet.xml`文件中定义一个`SimpleMappingExceptionResolver`，用来指定哪个异常转向哪个视图名称。

1. 在`web/WEB-INF/action-servlet.xml`中添加以下定义。

```
<bean id="exceptionResolver"
      class="org.springframework.web.servlet.handler.SimpleMappingException
      Resolver">
  <property name="exceptionMappings">
    <props>
      <prop>
        key="org.springframework.dao.DataAccessException">
          dataAccessFailure
        </prop>
      </props>
    </property>
  </bean>
```

在上面的代码中，`dataAccessFailure`指向一个`viewResolver`能识别的视图名称。对于Tiles用户，可能指向`tiles-config.xml`的定义。

2. 为了测试上面的映射可以运行，修改`UserDAOTest`(在`test/org/appfuse/dao`目录下)。在`testAddAndRemoveUser()`方法末尾，找到下面这行代码。

```
assertNull(dao.getUser(user.getId()));
```

3. 用以下的代码进行替换，当用户未找到时，确保异常能够抛出。

```
try {
  user = dao.getUser(user.getId());
  fail("User found in database");
} catch (DataAccessException dae) {
  log.debug("Expected exception: " + dae.getMessage());
  assertNotNull(dae);
}
```

4. 修改`UserDAOHibernate`(在目录`src/org/appfuse/dao`下)中`getUser()`方法，当用户找不到时，抛出一个异常。

```
public User getUser(Long id) {
  User user = (User) getHibernateTemplate().get(User.class, id);
  if (user == null) {
    throw new ObjectRetrievalFailureException(User.class, id);
  }
  return user;
}
```

```
}
```

5. 运行`ant test -Dtestcase=UserDAO`，验证一切是否按原计划那样的运行。
6. 在web目录下新建一个`dataAccessFailure.jsp`文件

```
<%@ include file="/taglibs.jsp" %>
<h3>Data Access Failure</h3>
<p>
<c:out value="${requestScope.exception.message}" />
</p>
<!--
<%
Exception ex = (Exception) request.getAttribute("exception");
ex.printStackTrace(new java.io.PrintWriter(out));
%>
-->
<a href="<c:url value='/' />">« Home</a>
```

如果你在MyUsers中使用的SiteMesh，所做的就这些。Tiles用户必须完成一些额外的步骤。

7. 在`tiles-config.xml`中添加`dataAccessFailure`定义。

```
<definition name="dataAccessFailure" extends="baseLayout">
  <put name="title" value="Data Access Failure"/>
  <put name="content" value="/dataAccessFailure.jsp"/>
</definition>
```

提示

你可以混合和匹配视图类，如某个视图中使用JSTL，下一个用Tiles。你可以使用 `ResourceBundleViewResolver` [<http://www.springframework.org/docs/api/org.springframework.web.servlet.view.ResourceBundleViewResolver.html>] 作为一个viewResolver，并且指定一个拥有视图名称和路径的属性文件来完成这项操作。你可以参考Spring自带的Petclinic样例程序中的这方面的例子。但是，这并不是鼓励你混合使用SiteMesh和Tiles。

8. 运行`ant deploy reload`，打开<http://localhost:8080/myusers/editUser.html?id=100>，当用户不存在时，你会看到如下的错误页面。

图 5.5. 错误页面提示用户不存在



如果你需要比SimpleMappingExceptionHandler提供的更为强大的功能，可以考虑实现HandlerExceptionHandler [http://www.springframework.org/docs/api/org.springframework.web.servlet.HandlerExceptionHandler.html] 来得到一个更为灵活的ExceptionHandler。

上传文件

你可能经常碰到要求在web应用程序中上传一个文件。令从欣慰的是Spring MVC支持文件上传。更好的是，你可以选择一种文件上传实现方式：Commons FileUpload [http://jakarta.apache.org/commons/fileupload]和COS FileUpload [http://www.servlets.com/cos/index.html]。

注意

如果你还在使用Servlet 2.2的容器，你无法使用Spring内置的上传文件支持方法。但Commons FileUpload [http://jakarta.apache.org/commons/fileupload/]对Servlet2.2容器进行文件上传操作提供了很好的支持。

1. 在web/WEB-INF/action-servlet.xml文件中定义一个id为multipartResolver的bean。

```
<bean id="multipartResolver"
      class="org.springframework.web.multipart.commons.
      CommonsMultipartResolver"/>
```

这个bean为DispatcherServlet提供了multipart支持。一个multipart请求是一个包含二进制和文本数据的HttpServletRequest [http://java.sun.com/j2ee/1.4/docs/api/javax/servlet/http/HttpServletRequest.html]。你要提交一个multipart请求的form的话，在HTML form>添加一个enctype="multipart/form-

data" 属性。当Spring检测到一个到一个multipart请求时，只是简单的用 `MultipartHttpServletRequest` [<http://www.springframework.org/docs/api/org.springframework.web.multipart/MultipartHttpServletRequest.html>] 来包装当前请求，这个类允许你访问普通的 `HttpServletRequest` 方法，和一些新增的功能(如 `getFile(String name)`)。

2. 为了在MyUsers中实现文件上传特性，新增一个预处理bean定义，和两个类：一个 `FileUploadCommand` 类 和一个 `SimpleFormController` [<http://www.springframework.org/docs/api/org.springframework.web.servlet.mvc/SimpleFormController.html>] 来处理上传过程。为此例在 `src/org/appfuse/web` 中创建这个command类，输入以下内容：

```
package org.appfuse.web;

public class FileUpload {
    private byte[] file;
    public void setFile(byte[] file) {
        this.file = file;
    }

    public byte[] getFile() {
        return file;
    }
}
```

3. 在同一目录中，新建一个 `FileUploadController` 类(从 `SimpleFormController` 继承)。在以下的代码中，最重要的部分是 `initBinder()` 方法，它注册了一个 `PropertyEditor` 来获取上传文件的字节流。没有这个方法，上传就会失败。

```
package org.appfuse.web;
// use your IDE to organize imports

public class FileUploadController extends SimpleFormController {
    private static Log log = LogFactory.getLog(FileUploadController.class);

    protected void initBinder(HttpServletRequest request,
        ServletRequestDataBinder binder)
        throws ServletException {
        binder.registerCustomEditor(byte[].class,
            new ByteArrayMultipartFileEditor());
    }

    protected ModelAndView onSubmit(HttpServletRequest request,
        HttpServletResponse response,
        Object command,
        BindException errors)
        throws ServletException, IOException {
        FileUpload bean = (FileUpload) command;
        byte[] bytes = bean.getFile();
        // cast to multipart file so we can get additional information
    }
}
```

```
MultipartHttpServletRequest multipartRequest =
    (MultipartHttpServletRequest) request;
CommonsMultipartFile file =
    (CommonsMultipartFile) multipartRequest.getFile("file");
String uploadDir = getServletContext().getRealPath("/upload/");
// Create the directory if it doesn't exist
File dirPath = new File(uploadDir);
if (!dirPath.exists()) {
    dirPath.mkdirs();
}
String sep = System.getProperty("file.separator");
if (log.isDebugEnabled()) {
    log.debug("uploading to: " + uploadDir + sep +
        file.getOriginalFilename());
}
File uploadedFile = new File(uploadDir + sep +
    file.getOriginalFilename());
FileCopyUtils.copy(bytes, uploadedFile);
// set success message
request.getSession().setAttribute("message", "Upload completed.");
String url = request.getContextPath() + "/upload/" +
    file.getOriginalFilename();
Map model = new HashMap();
model.put("filename", file.getOriginalFilename());
model.put("url", url);
return new ModelAndView(getSuccessView(), "model", model);
}
```

4. 在action-servlet.xml中写入Controller的bean定义。

```
<bean id="fileUploadController"
    class="org.appfuse.web.FileUploadController">
    <property name="commandClass">
        <value>org.appfuse.web.FileUpload</value>
    </property>
    <property name="formView"><value>fileUpload</value></property>
    <property name="successView">
        <value>fileUpload</value>
    </property>
</bean>
```

5. 为urlMapping bean添加一个<prop>，定义访问此Controller的URL。

```
<prop key="/fileUpload.html">fileUploadController</prop>
```


6. 新建文件web/fileUpload.jsp，其中包含一个upload form，和一个显示已经上传文件的链接。

```
<%@ include file="/taglibs.jsp"%>
<h3>File Upload</h3>
<c:if test="${not empty model.filename}">
  <p style="font-weight: bold">
    Uploaded file (click to view):
    <a href="${model.url}">${model.filename}</a>
  </p>
</c:if>
<p>Select a file to upload:</p>
<form method="post" action="fileUpload.html" enctype="multipart/form-data">
  <input type="file" name="file"/><br/>
  <input type="submit" value="Upload" class="button"
    style="margin-top: 5px"/>
</form>
```

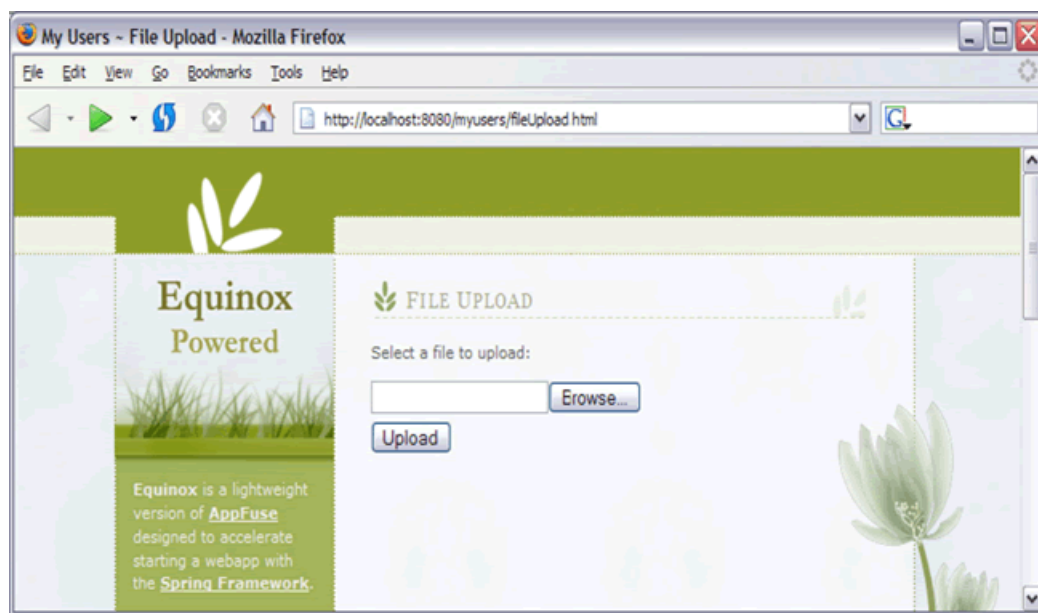
SiteMesh用户到这里就完成了。Tiles用户还要继续完成以下的步骤。

7. 添加fileUpload视图定义

```
<definition name="fileUpload" extends="baseLayout">
  <put name="title" value="My Users ~ File Upload"/>
  <put name="content" value="/fileUpload.jsp"/>
</definition>
```

8. 为了验证成功，打开浏览器浏览<http://localhost:8080/myusers/fileUpload.html>。你的浏览器窗口应该和图5.6类似。

图 5.6. File Upload页面

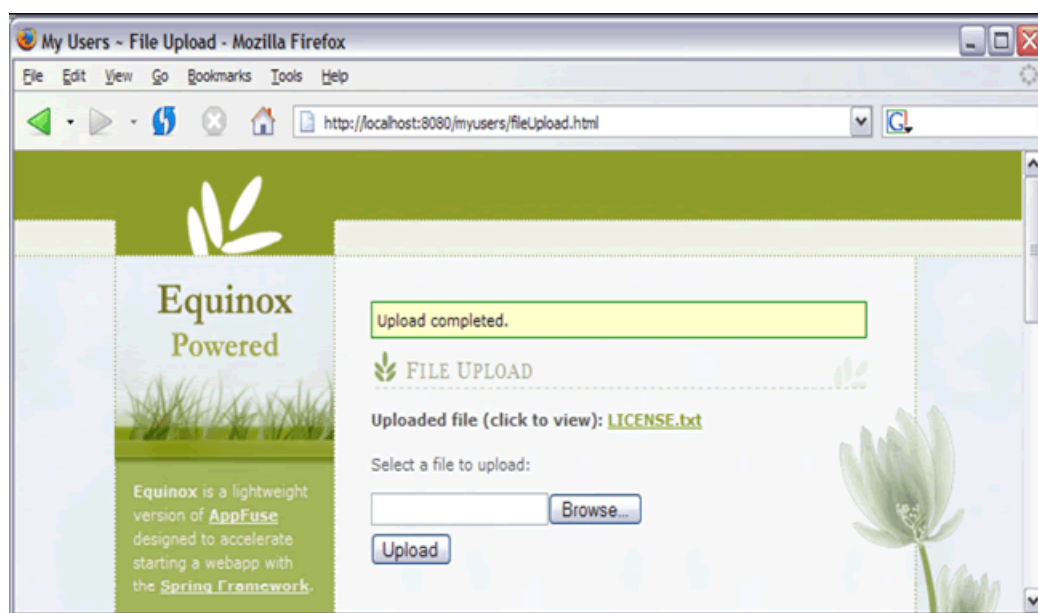


上传一个文件后你会看到一个如下类似的屏幕。

警告

一些文件(*.html)不允许通过点文件来浏览，所以我推荐选择MyUsers目录中的LICENSE.txt。

图 5.7. 成功文件上传页面



如果你重现了上面的屏幕截图，那么恭喜你!你已经弄清了如何使用Spring上传文件。第8章：Spring单元测试会涉及到有关FileUploadController的TDD开发。

拦截请求

很多MVC框架都能够为控制器(Controller)指定拦截器(Interceptor)。拦截器是一些能够拦截请求并执行一些逻辑(控制流，设置请求属性等)的类。它们与Servlet Filter相似，主要差别是Filter是在web.xml中进行配置，拦截器是在一个框架配置文件配置的。

`HandlerInterceptor` [<http://www.springframework.org/docs/api/org.springframework.web.servlet.HandlerInterceptor.html>] 是一个接口，包含一些方法用来拦截一个处理器执行前(preHandle)，执行后(postHandle)或是生成视图后(afterCompletion)的执行操作。一些有用的内置的Spring拦截器有 `OpenSessionInViewInterceptor` [<http://www.springframework.org/docs/api/org.springframework.orm.hibernate.support.OpenSessionInViewInterceptor.html>] 和 `UserRoleAuthorizationInterceptor` [<http://www.springframework.org/docs/api/org.springframework.web.servlet.handler/UserRoleAuthorizationInterceptor.html>]。前者是针对Hibernate的，后者用来阻止某一角色访问某一URL。

提示

`OpenSessionInViewInterceptor`有一个同样功能的姐妹过滤器(`OpenSessionInViewFilter` [<http://www.springframework.org/docs/api/org.springframework.orm.hibernate.support.OpenSessionInViewFilter.html>])。当生成视图时，两者都可以用作延迟载入由Hibernate管理的对象。这个Filter是MVC框架必备的。

下面是一个配置和使用`UserRoleAuthorizationInterceptor`的例子。配置MyUsers应用程序，将某一特定的url形式保护起来。监控整个应用程序，让具有“tomcat”身份的用户才可以访问。然后配置拦截器让仅具有“manager”角色的用户可以上传文件。

1. 在web/WEB-INF目录下的web.xml文件的末尾添加以下代码。

```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>tomcat</role-name>
    <role-name>manager</role-name>
  </auth-constraint>
</security-constraint>
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>My Users</realm-name>
</login-config>
<security-role>
  <role-name>tomcat</role-name>
</security-role>
<security-role>
  <role-name>manager</role-name>
</security-role>
```

2. 运行ant deploy reload，浏览http://localhost:8080/myusers。将会提示你要登录。“tomcat”角色使用用户名“tomcat”密码“tomcat”登入，“manger”用“admin/admin”来登录。这些用户和角

色是在Tomcat的conf/tomcat-users.xml(在\$CATALINA_HOME目录下)文件中进行配置的。

3. 添加一个拦截器，仅允许“manager”上传文件，在web/WEB-INF/action-servlet.xml中定义bean UserRoleAuthorizationInterceptor。

```
<bean id="managersOnly"
      class="org.springframework.web.servlet.handler.
        UserRoleAuthorizationInterceptor">
  <property name="authorizedRoles">
    <value>manager</value>
  </property>
</bean>
```

4. 添加一个新的SimpleUrlHandlerMapping，在/fileUpload.html这个映射路径上使用拦截器。

```
<bean id="managerMappings"
      class="org.springframework.web.servlet.handler.
        SimpleUrlHandlerMapping">
  <property name="interceptors">
    <list>
      <ref bean="managersOnly"/>
    </list>
  </property>
  <property name="mappings">
    <props>
      <prop key="/fileUpload.html">
        fileUploadController
      </prop>
    </props>
  </property>
</bean>
```

5. 关闭浏览器(登出)，重新打开http://localhost:8080/myusers/fileUpload.html。将会显示登录提示。如果你以“tomcat/tomcat”登录，会出现一个403错误页面，意思是“accessdenied(拒绝访问)”。

提示

你可以轻而易举的自定义这个页面，只要在web.xml文件指定一个403 <error-page>。

6. 重新以“admin/admin”登录。记住关闭浏览器来删除上次登录记录。

这只是一个使用拦截器控制URL访问的简单的例子。你可以使用同样的配置针对不同的类和URL映射来做其它事情(例如，确定某个属性一直在request作用域中)。

发送邮件

E-mail是一个优秀的通知系统，也充当一个基本的工作流系统的角色。Spring使发送邮件变得非常容易，它隐藏底层邮件系统的复杂性。Spring邮件支持的主要接口叫做MailSender。它还提供了一个SimpleMailMessage [http://www.springframework.org/docs/api/org.springframework.mail.SimpleMailMessage.html]类来封装一条消息的公共属性(from, to, subject, message)。如果你想发送带附件的邮件，你可以用MimeMessagePreparator [http://www.springframework.org/docs/api/org.springframework.mail.javamail.MimeMessagePreparator.html]来创建和发送消息。

在FileUploadController中创建一个简单的例子，当文件上传后发送一封邮件。在第9章：AOP，你会把这段逻辑抽取出来放入NotificationAdvice类中。

1. 在action-servlet.xml文件中添加一个mailSender bean。host属性应该对应一个不需要认证的SMTP服务器。如果服务器需要认证，你要添加“username”和“password”属性：

```
<bean id="mailSender"
      class="org.springframework.mail.javamail.JavaMailSenderImpl">
  <property name="host"><value>localhost</value></property>
</bean>
```

注意

使用JavaMailSenderImpl [http://www.springframework.org/docs/api/org.springframework.mail.javamail.JavaMailSenderImpl.html]类时，要求activation.jar和mail.jar在你的classpath中。这些文件已经包含在第5章的下载包中。

2. 在FileUploadController类中为MailSender添加一个属性和setter方法。同时，还要为SimpleMailMessage添加一个属性/setter方法的组合。设置SimpleMailMessage依赖注入，允许你在action-servlet.xml文件中配置一个默认的from和subject值。

```
private MailSender mailSender;
private SimpleMailMessage message;

public void setMailSender(MailSender mailSender) {
  this.mailSender = mailSender;
}

public void setMessage(SimpleMailMessage message) {
  this.message = message;
}
```

3. 指定一个邮件的默认值，添加以下的XML片断到action-servlet.xml文件中。

```
<bean id="mailMessage"
```

```
class="org.springframework.mail.SimpleMailMessage">
<property name="from">
  <!-- The <value> and CDATA below must be on the same line -->
  <value><![CDATA[Uploader <spring@sourcebeat.com>]]></value>
</property>
<property name="subject">
  <value>File finished uploading</value>
</property>
</bean>
```

4. 修改fileUploadController bean定义, 注入mailSender和message属性。

```
<bean id="fileUploadController"
class="org.appfuse.web.FileUploadController">
  <!-- other properties hidden for brevity -->
  <property name="mailSender"><ref bean="mailSender"/></property>
  <property name="message"><ref bean="mailMessage"/></property>
</bean>
```

5. 在FileUploadController类中的onSubmit()添加以下代码(在返回ModelAndView之前)。

```
// Notify user that file has finished uploading
SimpleMailMessage msg = new SimpleMailMessage(this.message);
msg.setTo("springlive@raibledesigns.com");
msg.setText("File \"" + file.getOriginalFilename() +
  "\" has finished uploading.");
try {
  mailSender.send(msg);
} catch (MailException ex) {
  log.error(ex.getMessage());
}
```

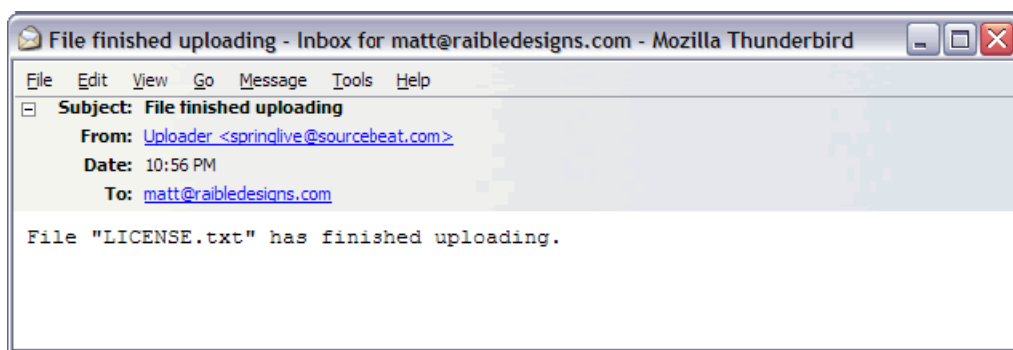
提示

确保修改msg.setTo()的邮件地址, 不然你就会这份邮件发给我。

这个例子只是简单的记下了异常日志, 因为这个消息发送不是很重要。如果你应用中通知邮件是绝密的, 请用一个SimpleMappingExceptionResolver类来处理异常。

6. 为了测试上面的配置(你必须有访问SMTP服务器权限), 运行ant deploy reload, 打开 <http://localhost:8080/myusers/fileUpload.html>, 以“admin/admin”登录。然后上传一个文件, 等候一封邮件。你会看到一个与下面类似的邮件结果。

图 5.8. 成功的邮件消息



提示

你也可以用JNDI中的一个MailSession对象来配置mailSessionbean。你可以从TheServerSide.com上学到更多这方面的知识，还有一篇题为Sending Velocity-based E-Mail with Spring [http://www.theserverside.com/blogs/showblog.tss?id=SpringVelocityEmail]讲述如何使用Velocity模板。

本章小结

本章涉及了几个高级的Spring MVC主题。你学习了如何配置和使用当前最为流行的页面修饰框架：SiteMesh和Tiles。验证是web程序中必不可少的部分，你了解了如何配置Commons Validator，还有如何实现一个内置的Spring Validator。异常处理同样很重要，现在你可以领会到了如何抛出和处理异常。文件上传也不难，如果你一个样例的话，这在本章提供了。拦截器与Servlet Filter类似，但是Spring有一些更为有用的内置特性，如UserRoleAuthorizationInterceptor。最后，你看到了使用Spring的JavaMail支持方式，如何让邮件这一通知机制变得更加简单。第6章会研究Spring支持的各种不同的视图，包括：Velocity，FreeMarker，XML/XSL，Excel和PDF。

第 6 章 View 的多种选择

Spring 中提供的各种 View 选择

本章介绍 Spring MVC 架构中 View 的各种选择。截止撰稿时，可供选择的有 JSP，Velocity，FreeMarker，XSLT，PDF 和 Excel。本章为各种 Spring 所支持的 View 提供了参考，还简短的说明各种 View 如何运作，并对每种 View 分别构建一个页面进行了对比。另外还集中介绍了每种 View 的国际化问题。

在 J2EE 应用程序中，由于在 J2EE 1.4 规范中 JSP 实际上是唯一指定的页面模板技术选择，JSP 已经成为 MVC 中 V(View) 事实上的标准。然而，其它的一些技术也引起了人们的关注。Velocity 和 FreeMarker 同样是模板技术，它们使用和 JSP 2.0 中类似的语法。当你抽取了 XML 数据并想对其进行转换时，或者你想即时的生成不同的 View(通过 XSL)，那么 XML/XSL 是一种不错的选择。XMLC 与模板技术类似，一个例外就是它使用普通的 HTML，利用其 id 属性来定位和替换动态文本。最后，输出 PDF 和 Excel 格式的数据对生成报表和产生可移植文档非常有用。在本章中，你将会学习如何结合 Spring 框架使用上述各种不同的 View。在学习如何配置各种不同的 View 之前，有必要弄清 Spring 是怎么识别 View 的。

View(视图)和 ViewResolver(视图解析器)

Spring 提供了好几个 ViewResolver，这样你可以从 View 中把 Controller 解脱出来(解耦)。在控制器中，你只要简单的指定一个 View 的逻辑名称，Spring 就能把这个名称解析为指定的 View 类型(view type)。View 接口会预处理请示，并把它传递给你事先配置好的 View 技术。在松散耦合的理念下，Spring 让你在一个 XML context 文件中配置你的 View 选择变得非常容易。

在第 4 章中，你已经看到控制器(Controller)如何返回一个 ModelAndView。此对象包含了一个 View 名称，指定的 View Resolver 利用它判定该向用户显示什么。通过本章，你将会学习各种 View 的工作原理。下面是目前 Spring 中提供的 View Resolver 的一个清单。“用途”列阐述了每种解析器的用途。

表 6.1. View Resolvers

View Resolver	描述	用途
AbstractCachingViewResolver	的ViewResolver,能缓存View。很多View在使用它们之前需要预处理。继承这个View Resolver可以启用View的缓存能力。	继承在一个自定义的解析器中使用缓存。
ResourceBundleViewResolver	ViewResolver的一种实现,用于从ResourceBundle(通过bundle的basename指定)中读取bean定义。此bundle通常是在一个properties文件中定义,位于classpath中。	使用各种的View,如JSP和Velocity
UrlBasedViewResolver	ViewResolver的一种简单实现,允许直接将View名称解析为url,而不需要额外的映射定义。如果你的象征名称能够直接匹配你的名称资源中的名称,而不需要专门的映射处理,用此解析器是适合的。	象征性的View名称解析为URL,而不需要额外的映射处理。
InternalResourceViewResolver	UrlBasedViewResolver一个方便的子类,支持InternalResourceView(如Servlet, jsp)及其子类,如JstlView和TilesView。通过设置setViewClass属性,你可以为这个解析器生成的所有的View指定这个View类。	JSP和Tiles, JstlView和TilesJstlView 是最方便的View类。
VelocityViewResolver	AbstractTemplateViewResolver一个方便的子类,支持VelocityView(也就是, Velocity模板)及其子类。	支持VelocityView。允许使用缓存和自定义属性。
FreeMarkerViewResolver	AbstractTemplateViewResolver一个方便的子类,支持FreeMarkerView(也就是, FreeMarker模板)及其子类。	支持FreeMarkerView。

在大多数情况下,你不必挑选该采用哪种ViewResolver,只是简单的使用你所选用的View技术所要求的那种。ResourceBundleViewResolver是一个例外,它允许你对每个View选择各种不同View类。在开始使用这种解析器之前,你将会学习如何配置和使用Spring支持的各种View技术。

每个View技术小节会向你演示如何配置View Resolver,如何针对此View技术实现相应的页面。学习的最佳途径是一起来做本章的练习。首先,从<http://sourcebeat.com/downloads>下载MyUsers Chapter 6压缩包。项目的树目录中已经包含了你在这章中将要用到的jars文件。你也可以从前一章开发的应用程序入手,如果你想这么做,请从<http://sourcebeat.com/downloads>下载Chapter 6 JARs。

提示

如果你选择从现有的应用开始,请务必删除在演示Interceptor时添加的安全验证。这里有一个小麻烦,每次测试你的View时,都会要求登录。

在正式探讨各种View选项之前,写一个单元测试,测试最基本的功能,显示,添加,保存和删除用户。

利用jWebUnit测试View

jWebUnit是SourceForge.net上的一个开源项目。它提供了一组简单的API(基于HttpUnit)来测试web应用程序。

1. 如果你已经下载好了Chapter 6压缩包(或是Chapter 6 jars文件), 你不需要为你的应用程序添加额外的jar文件。否则, 请下载jWebUnit(1.2版), 把下列jar文件放入web/WEB-INF/lib中。
 - jwebunit-1.2.jar
 - httpunit-1.5.4.jar
 - Tidy.jar
2. 如果你使用eclipse或IDEA作为开发工具, 修改一下classpath, 把这些jar文件加进去。在eclipse 3.0中, 位于Project → Properties → Project Build Path → Libraries选项卡。在idea中, 是File → Settings → Project Settings:Paths → Libraries选项卡。

提示

这里有关于在eclipse [\[http://jroller.com/page/raible?anchor=setup_myusers_in_eclipse\]](http://jroller.com/page/raible?anchor=setup_myusers_in_eclipse)的idea [\[http://jroller.com/page/raible?anchor=setup_myusers_in_idea\]](http://jroller.com/page/raible?anchor=setup_myusers_in_idea)构建和测试MyUsers应用的指导。

3. 新建一个单元测试, 在test/org/appfuse/web新建文件UserWebTest.java, 此类继承了WebTestCase。代码如下:

```
package org.appfuse.web;
// use your IDE to organize imports

public class UserWebTest extends WebTestCase {
    public UserWebTest(String name) {
        super(name);
        getTestContext().setBaseUrl("http://localhost:8080/myusers");
    }

    public void testWelcomePage() {
        beginAt("/");
        assertEquals("MyUsers ~ Welcome");
    }

    public void testAddUser() {
        beginAt("/editUser.html");
        assertEquals("MyUsers ~ User Details");
        setFormElement("id", "");
        setFormElement("firstName", "Spring");
        setFormElement("lastName", "User");
        submit("save");
        assertTextPresent("saved successfully");
    }

    public void testListUsers() {
        beginAt("/users.html");
    }
}
```

```
// check that table is present
assertTablePresent("userList");
//check that a set of strings are present somewhere in table
assertTextInTable("userList",
    new String[] {"Spring", "User"});
}

public void testEditUser() {
    beginAt("/editUser.html?id=" + getInsertedUserId());
    assertFormElementEquals("firstName", "Spring");
    submit("save");
    assertTitleEquals("MyUsers ~ User List");
}

public void testDeleteUser() {
    beginAt("/editUser.html?id=" + getInsertedUserId());
    assertTitleEquals("MyUsers ~ User Details");
    submit("delete");
    assertTitleEquals("MyUsers ~ User List");
}

/**
 * Convenience method to get the id of the inserted user
 * Assumes last inserted user is "Spring User"
 */
public String getInsertedUserId() {
    String[] paths = {"WEB-INF/applicationContext.xml"};
    ApplicationContext ctx =
        new ClassPathXmlApplicationContext(paths);
    List users = ((UserDAO) ctx.getBean("userDAO")).getUsers();
    // assumed that user inserted in testAddUser() is last user
    return ""+((User)users.get(users.size()-1)).getId();
}
}
```

4. 为了使用jWebUnit, 修改一下现有的JSP文件中的少许HTML代码。testListUsers()包含了下面一行:

```
assertTablePresent("userList");
```

这一行代码会检测生成的页面中是否存在<table>, 为了确保它能够运行, table必须添加一个值为userList的id属性。

5. 打开web/userList.jsp, 添加这个属性, 你的文件应该看起来和下面类似。

```
<table class="list" id="userList">
```

6. 修改一个Hibernate设置, 这样数据库就不会在JVM每次重启时重新创建。打开文件web/WEB-INF/applicationContext.xml, 修改“hibernate.hbm2ddl.auto”, 将其值由create改为update。

```
<prop key="hibernate.hbm2ddl.auto">update</prop>
```

7. 因为jWebUnit采用的是容器内的测试(它要求一个运行中的服务器)。运行ant clean deploy来启动Tomcat(如果Tomcat处于运行状态, 执行ant deploy reload)。
8. 从IDE中执行测试, 或者从命令行中执行ant test -Dtestcase=UserWeb。

警告

如果第一次尝试测试失败, 你可能要删除数据库(删除工作目录中“db”)。

9. 在build.xml把容器内和容器外测试分开。方法是, 在“test”目标(target)中名称中包含WebTest的类排除掉。在build.xml中找到这个目标, 添加下面一行:

```
<batchtest todir="${test.dir}/data" unless="testcase">
  <fileset dir="${test.dir}/classes">
    <include name="**/*Test.class"/>
    <exclude name="**/*WebTest.class"/>
  </fileset>
</batchtest>
```

10. 紧接着“test”目标, 新建一个名为“web-test”的目标。

```
<target name="test-web" depends="compile"
  description="Runs tests that required a running server">
  <property name="testcase" value="WebTest"/>
  <antcall target="test"/>
</target>
```

11. 执行所有容器外的测试时使用ant test, 所有的容器内测试则使用ant test-web。

JSP

你所开发的MyUsers样例程序已经用到了JSP 2.0。这个版本比先前的版本简化了许多, 你可以用它的表达式语言(EL)来提取数据。它不仅消除了脚本(scriptlet), 而且它的语法与Velocity和FreeMarker接近。JSF(JSF是J2EE标准的web应用程序框架)重新包装了JSP。很多人相信JSF会占领Java web开发的主导地位。下面的内容同样适用于JSF应用程序。

View Resolver配置

为了使用JSP, 特别是JSP标准标签库(JSTL), 你必须在web/WEB-INF/action-servlet.xml定义一个InternalResourceViewResolver。

```
<bean id="viewResolver"
  class="org.springframework.web.servlet.view.InternalResourceViewResolver"
```

```

>
<property name="requestContextAttribute"><value>rc</value></property>
<property name="viewClass">
  <value>org.springframework.web.servlet.view.JstlView</value>
</property>
<property name="prefix"><value>/</value></property>
<property name="suffix"><value>.jsp</value></property>
</bean>

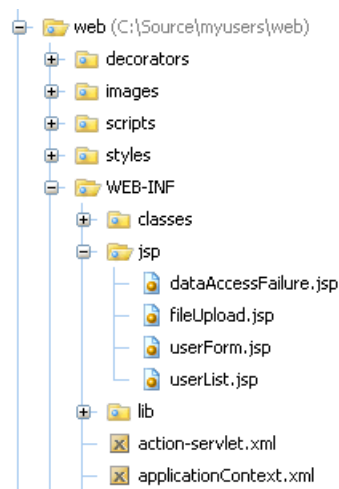
```

在上面的定义中，“requestContextAttribute”把Spring的RequestContext对象暴露为变量rc。如果你正在使用JSP，你可能用不到这个变量，但你利用\${rc.contextPath}来获取应用程序的contextPath(也就是/MyUsers)。此对象对于那些不访问servlet请求的View技术(也就是Velocity和FreeMarker模板)来说是必不可少的。prefix和postfix值提供方便的途径，使View名称变得更友好，而不是生硬的代码路径。

作为一个示例，把web目录中JSP文件转移到web/WEB-INF/jsp(你可以要创建这个目录)目录下。下面是此刻你的目录树应该和下面的这个局部快照看起来差不多。下列JSP保留在web目录，这些文件要么是错误页面，要么会在SiteMesh decorator用到。

- 404.jsp
- error.jsp
- index.jsp
- messages.jsp
- taglibs.jsp.

图 6.1. 目录树样例



配置Spring，让它能够找到JSP文件的新位置，把viewResolver的“prefix”属性修改为/WEB-INF/jsp/。

```

<property name="prefix"><value>/WEB-INF/jsp/</value></property>

```

为了测试以上配置，确保Tomcat牌运行状态，执行ant remove。执行“remove”会从Tomcat中删除应用程序，确保在应用的根目录没有遗留JSP文件。然后执行ant deploy，等等Tomcat重新安装应用程序。最后运行ant test-web。

JSTL

在前面的viewResolver，设置了“viewClass”属性。

```
<property name="viewClass">
  <value>org.springframework.web.servlet.view.JstlView</value>
</property>
```

此属性值引用了Spring用来识别View页面的View类(view class)。在本例中，它使用了JstlView。这个类非常有用，它把Spring的locale和message暴露给JSTL的格式化(format)和消息(message)标签。如果在用JSP，这个类正是你要用的。

Display标签

如果你正在使用JSP并且想输出一些数据列表，可以考虑一下Display标签库 [http://display.sf.net]。Display标签是一种JSP标签，提供了分页机制和对数据表进行排序的功能。为了演示它的特性，会在MyUsers应用程序使用它。本章下载的文件中已经包含其1.0版本的jar文件(1.0RC)，还有CSS文件和图片。你也可以从SourceForge.net上下载 [http://sourceforge.net/project/showfiles.php?group_id=73068]。

1. 修改文件userList.jsp(在目录web/WEB-INF/jsp)，其内容如下。

```
<%@ include file="/taglibs.jsp"%>
<%@ taglib uri="http://displaytag.sf.net" prefix="display" %>
<head>
  <title>MyUsers ~ User List</title>
  <link href="styles/displaytag.css" type="text/css" rel="stylesheet"/>
</head>
<button onclick="location.href='editUser.html'">Add User</button>
<c:set var="idKey"><fmt:message key="user.id"/></c:set>
<c:set var="firstNameKey"><fmt:message key="user.firstName"/></c:set>
<c:set var="lastNameKey"><fmt:message key="user.lastName"/></c:set>
<display:table name="${users}" class="list" requestURI="" id="userList"
export="true">
  <display:column property="id" sort="true" href="editUser.html"
paramId="id" paramProperty="id" title="${idKey}"/>
  <display:column property="firstName" sort="true"
title="${firstNameKey}"/>
  <display:column property="lastName" sort="true" title="${lastNameKey}"/>
  <display:setProperty name="basic.empty.showtable" value="true"/>
</display:table>
```

2. 运行ant deploy reload，用浏览器打开http://localhost:8080/myusers/users.html。你应该可以看到与下面类似的一张表，它是按First Name排序的。

图 6.2. 使用此标签库的运行结果



你可以用表格下的Excel，XML和CSV链接把表格的内容以标明的格式导出。

提示

在下一个演示中使用这个标签库，很多人就会对它能在web应用程序中进行排序的功能留下很深的印象。

Tiles

在第5章中，我们已经讨论了Tiles这种优雅的页面修饰框架。针对Tiles的viewResolver配置与JSTL非常类似，只要简单将“viewClass”属性值设置为TilesView或者是TilesJstlView。我建议使用JSTL的版本，因为它提供了所有的TilesView特性，还能在你的页面中使用JSTL的格式化标签。

```
<property name="viewClass">
  <value>org.springframework.web.servlet.view.tiles.TilesJstlView</value>
</property>
```

要使用Tiles，你必须在web/WEB-INF/action-servlet.xml设置一个bean，读取Tiles配置文件。最简单的方法是使用Spring的TilesConfigurer类，指定你的定义文件。

```
<bean id="tilesConfigurer"
  class="org.springframework.web.servlet.view.tiles.TilesConfigurer">
  <property name="factoryClass">
    <value>org.apache.struts.tiles.xmlDefinition.I18nFactorySet</value>
  </property>
```

```

<property name="definitions">
  <list>
    <value>/WEB-INF/tiles-config.xml</value>
  </list>
</property>
</bean>

```

使用JSP编写View页面是J2EE开发中一种常用的方法。但是，这不是唯一的途径。你可以用其它模板框架如Velocity和FreeMarker，两者都是轻量级的快捷的方案。更棒的是，Spring使它们变得更加易用，它去除了最棘手的环节：设置和配置。

Velocity

Velocity是一个开源项目，网址是<http://jakarta.apache.org/velocity>。它主要用于为浏览器生成动态网页。然而，既然它是一种模板技术，你可以把它用在任何需要模板的地方(例如，发送基于HTML的邮件)。Velocity同样是在J2EE容器内运行，相对JSP，它有一个优点，从一个容器跳转到下一个容器，它表现出更快的响应和更好的性能。Velocity自带一种模板语言，叫做Velocity Template Language(VTL)，语法和JSP 2.0的EL极其相似。例如，要打印一个user对象的“lastName”属性，在JSP中你应该使用`${user.lastName}`，在Velocity中，语法根本不用修改：`user.lastName`。这就是所谓的形式引用符号(formal reference notation)。你也可以使用速写语法：`$user.lastName`。由于它的语法和与JSP2.0和FreeMarker相似，本章中采用这种形式符号。

Velocity从context中抓取一些数据，并在页面中把它们表示出来。当然首先，你必须把数据放入context中，向context中填充数据本来就简单，在Spring更容易。

下一节把MyUsers从JSP转向Velocity。你将从配置Velocity入手。本章要求你的classpath(web/WEB-INF/lib)中包含以下jar文件。

- velocity-1.4.jar
- velocity-tools-view-1.1.jar

在MyUsers中使用Velocity

在MyUsers应用程序中，SiteMesh作为页面 decorator。SiteMesh可以让JSP页面修饰完美无缺，你可以使用基于JSP的 decorator来修饰Velocity驱动的面。然而你可以还是想使用Velocity作为 decorator模板。在这一节中，从在action-servlet.xml配置Velocity入手。然后，你会修改SiteMesh，使用一个Velocity decorator。

View Resolver配置

当你向一个web应用程序添加Velocity时，你通常要添加一个velocity.properties文件到classpath中。此文件中有一些设置，告诉Velocity该用哪种ResourceLoader导入模板。为了方便，Spring提供一个VelocityConfigurer类，允许你为模板(也就是页面)指定一个路径而不需要这样一个配置文件。

1. 向文件web/WEB-INF/action-servlet.xml中添加以下XML片断。它告诉Velocity从你的应用程序的WEB-INF/velocity中加载模板。

```
<bean id="velocityConfig"
```



```

class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
<property name="resourceLoaderPath">
  <value>/WEB-INF/velocity/</value>
</property>
</bean>

```

2. 如果对ResourceLoader获得更多的控制(例如, 从数据库加载模板), 请新建一个velocity.properties文件。要加载这个文件, 在velocityConfig bean中指定一个configLocation属性和属性值。你可以直接在bean定义中指定这些属性。关于使用备选的ResourceLoader更多信息, 请参考Spring Velocity属性文档
[<http://www.springframework.org/docs/reference/view.html>]。

3. 声明VelocityViewResolver作为应用程序的View Resolver。

提示

请确保注释掉了前面JSP中用到的InternalResourceViewResolver。

```

<bean id="viewResolver"
class="org.springframework.web.servlet.view.velocity.VelocityViewResolver">
  <property name="exposeSpringMacroHelpers"><value>true</value></property>
  <property name="requestContextAttribute"><value>rc</value></property>
  <property name="cache"><value>true</value></property>
  <property name="prefix"><value>/WEB-INF/velocity/</value></property>
  <property name="suffix"><value>.vm</value></property>
</bean>

```

定义中的前两个属性, 即“exposeSpringMacroHelpers”和“requestContextAttribute”, 非常重要。第一个暴露了处理表单和验证错误信息的宏操作。“requestContextAttribute”是RequestContext的一个别名。这个类用于从messages.properties文件打印本地化的信息。

VelocityViewResolver从它的父类即AbstractTemplateViewResolver中继承了好几个可选的属性。下表一一列出这些属性及其描述。这些属性适用于任何AbstractTemplateViewResolver(例如,FreeMarkerViewResolver)。

表 6.2. 额外的TemplateViewResolver属性

属性名	描述	默认值
exposeRequestAttributes	设置是否所有的request属性在与模板进行合并之前添加到model中	false
exposeSessionAttributes	设置是否所有的session属性在与模板进行合并之前添加到model中	false
exposeSpringMacroHelpers	设置是否通过Spring的宏库暴露一个RequestContext(名为springBindRequestContext)供外部使用。	false

为了和Spring一起使用Velocity，你还要在`*-servlet.xml`文件中指定两个bean定义(`VelocityConfigurer`和`VelocityViewResolver`)。当然，还有更多的东西要做。首先，你要准备Velocity模板文件，研究如何在VTL中输出错误信息。在MyUsers程序中，你还要学习如何使用SiteMesh的Velocity decorator进行页面修饰。

SiteMesh和Velocity

用Velocity替代JSP对MyUsers转换过程中最令人头痛的是为了使用Velocity decorator而配置SiteMesh。幸运的是，SiteMesh 2.1已经添加了这项功能，现在它已经非常简单。

1. 新建一个与`web/decorators/default.jsp`类似的 decorator，但这个文件中使用Velocity替换JSP标签来填充SiteMesh内容。SiteMesh内置了一个`VelocityDecoratorServlet`用来向context中预填充几个有用的变量。

表 6.3. SiteMesh Velocity Context对象

<code>\${request}</code>	HttpServletRequest对象
<code>\${response}</code>	HttpServletResponse对象
<code>\${base}</code>	<code>request.getContextPath()</code>
<code>\${title}</code>	解析过的页面title
<code>\${head}</code>	解析过的页面head
<code>\${body}</code>	解析过的页面body
<code>\${page}</code>	SiteMesh内部Page对象

2. 下面是用Velocity decorator重写的`default.jsp` decorator。下面几行代码作为创建`web/decorators/default.vm`模板的一个教程。删除线表示你要删除的文字。下划线表示你要修改的文字。为了节省页面空间，略去了那些不需要任何修改的部分，因而这里没有显示出来。

可以看到，VTL语法比JSP还要简洁一些(例如，`<decorator:title default="MyUsers"/>` 简化为`${title}`)。

提示

复制一份`web/decorators/default.jsp`保存为`web/decorators/default.vm`，为下面Velocity转换作准备。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<% include file="/taglibs.jsp"%>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>${title}</title>
<meta http-equiv="content-type" content="text/html; charset=utf-8"/>
<c:set var="ctx" value="${pageContext.request.contextPath}"
scope="request"/>
<link href="${base}/styles/global.css" type="text/css" rel="stylesheet"/>
<link href="${base}/images/favicon.ico" rel="SHORTCUT ICON"/>
${head}
```

```
...
<h1><span>Welcome to MyUsers</span></h1>
<div id="logo" onclick="location.href='${base}'"
onkeypress="location.href='${base}'"></div>
<h2><span>Spring Rocks!</span></h2>
...
<div id="content">
#parse("/messages.vm")
${body}
</div>
</div>
<div id="supportingText">
<div id="underground">${page.getProperty("page.underground")}</div>
<div id="footer">
...
<a href="http://bobby.watchfire.com/bobby/
bobbyServlet?URL=${request.requestURL}&output=Submit&gl=sec508&test="
title="Check the accessibility of this site according to U.S.
Section 508">508</a> .
<a href="http://bobby.watchfire.com/bobby/
bobbyServlet?URL=${request.requestURL}&output=Submit&gl=wcag1-
aaa&test="
title="Check the accessibility of this site according to
WAI Content Accessibility Guidelines 1">aaa</a>
</div>
```

3. 在文件的中间部分，它会解析和包含messages.vm文件。现在还没有这个文件，在web目录中新建一个。

```
## Success Messages
#if ($message)
<div class="message">${message}</div>
${request.session.removeAttribute("message")}
#end
```

4. 一旦你创建好了Velocity模板，配置SiteMesh来使用它。打开文件web/WEB-INF/decorators.xml，修改默认decorator的page，引用你刚刚才创建的default.vm文件。

```
<decorators defaultdir="/decorators">
  <decorator name="default" page="default.vm">
    <pattern>/*</pattern>
  </decorator>
</decorators>
```

5. 配置应用程序，使用SiteMesh的VelocityDecoratorServlet来解析Velocity模板。

编辑web.xml

为了让SiteMesh能够正确的解析Velocity decorator，在web/WEB-INF/web.xml添加一个servlet定义及相应的mapping。紧接着action声明添加以下XML代码。

```
<servlet>
  <servlet-name>sitemesh-velocity</servlet-name>
  <servlet-class>
    com.opensymphony.module.sitemesh.velocity.VelocityDecoratorServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>sitemesh-velocity</servlet-name>
  <url-pattern>*.vm</url-pattern>
</servlet-mapping>
```

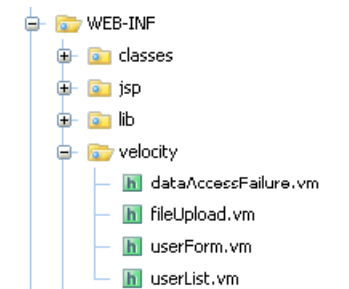
SiteMesh文档有相关操作说明 [<http://www.opensymphony.com/sitemesh/velocity-decorators.html>]。

创建Velocity模板

从JSP转向Velocity最后一步是修改所有的JSP文件，改用Velocity的VTL。

复制WEB-INF/jsp目录，另存为WEB-INF/velocity，将所有文件的扩展名改为.vm。你目录结构应该和图6.3所示的一样。

图 6.3. Velocity模板目录结构



下面是以上所示四种模板的一份清单，为了能和Velocity一起工作，所做的修改是必不可少的。每段代码后面还有对每个文件的特别说明。在模板中要用到的任何VTL代码都加上了下划线。

userList.vm

```
<title>MyUsers ~ User List</title>
<button onclick="location.href='editUser.html'">Add User</button>
<table class="list" id="userList">
  <thead>
    <tr>
      <th>${rc.getMessage("user.id")}</th>
      <th>${rc.getMessage("user.firstName")}</th>
      <th>${rc.getMessage("user.lastName")}</th>
```

```
</tr>
</thead>
<tbody>
#foreach ($user in $users)
#if ($velocityCount % 2 == 0) <tr class="even">
#else <tr class="odd">
#end
  <td><a href="editUser.html?id=${user.id}">${user.id}</a></td>
  <td>${user.firstName}</td>
  <td>${user.lastName}</td>
</tr>
#end
</tbody>
</table>
```

注意, `${rc.getMessage()}` 从文件 `web/WEB-INF/classes/messages.properties` 读取本地化信息。`$velocityCount` 变量是在迭代操作暴露出来的一个内部Velocity值。

userForm.vm

```
<title>MyUsers ~ User Details</title>
#springBind("user.*")
#if ($status.error)
<div class="error">
  #foreach ($error in $status.errorMessages)
    ${error}<br/>
  #end
</div>
#end
<p>Please fill in user's information below:</p>
<form method="post" action="editUser.html">
#springBind("user.id")
<input type="hidden" name="id" value="${status.value}"/>
<table>
<tr>
<th>${rc.getMessage("user.firstName")}:</th>
<td>
  #springBind("user.firstName")
  <input type="text" name="firstName" value="${status.value}"/>
  <span class="fieldError">${status.errorMessage}</span>
</td>
</tr>
<tr>
<th>${rc.getMessage("user.lastName")}:</th>
<td>
  #springBind("user.lastName")
  <input type="text" name="lastName" value="${status.value}"/>
  <span class="fieldError">${status.errorMessage}</span>
</td>
</tr>
<tr>
```

```
<td></td>
<td>
  <input type="submit" class="button" name="save" value="Save"/>
  #if ($user.id)
    <input type="submit" class="button" name="delete" value="Delete"/>
  #end
</td>
</table>
</form>
```

注意, #springBind宏调用每个字段暴露出来的变量, 美元符后面的叹号表示(`$!{...}`)表示在没有找到值的情况下, 什么也不打印。这些宏没有<spring:bind>JSP标签必须的关闭标签或是结束语句。当你把viewResolver bean的exposeSpringMacroHelpers属性值设置成true时, #springBind宏就是一个变量。

使用JavaScript和Commons Validator的客户端验证, 仅支持使用JSP。

fileUpload.vm

```
<h3>File Upload</h3>
#if ($model.filename)
  <p style="font-weight: bold">
    Uploaded file (click to view): <a
      href="{model.url}">${model.filename}</a>
  </p>
#end
<p>Select a file to upload:</p>
<form method="post" action="fileUpload.html" enctype="multipart/formdata">
  <input type="file" name="file"/><br/>
  <input type="submit" value="Upload" class="button"
    style="margin-top: 5px"/>
</form>
```

dataAccessFailure.vm

```
<h3>Data Access Failure</h3>
<p>
  ${exception}
</p>
<a href="{rc.contextPath}">« Home</a>
```

这个文件并没有像JSP在注释中打印出异常的Stack trace信息。我尝试在注释中使用`${exception.printStackTrace() }`, 但stack trace依然没有打印出来。

部署和测试

现在你已经让Spring配置和使用Velocity, 用一个Velocity Decorator配置好的SiteMesh, 以及所有由JSP转换过来的Velocity。现在该测试一切是否运行正常。启动Tomcat, 运行ant clean deploy reload。一旦新的context重启之后, 执行命令ant test-web来验证运行情况。

警告

如果数据库不存在某个用户时，导致测试失败，请删除数据库(`rm -r db`)，重启Tomcat重试一次。你也可以在“delete”目标中添加`<delete dir="db"/>`。

Velocity小结

在本节中，你学习了如何运用Velocity这种View技术来替代JSP。Velocity的语法往往比JSP简练(尽管JSP在2.0版本中得到了加强)。在你第一次Velocity支持的模板时，编译速度也会比JSP快一些。使用JSP时，初始加载时间可能需要好几秒，这在开发时令人有点不快，因为你不得不经常停下来等待。这些你可能并没有在意，直到你使用Velocity开发而转向JSP时才有感觉。Velocity的一个不足时，你无法使用现有的丰富的标签库(如`display`、`oscache`等等)。然而，已经有一种快速模板方案允许你使用JSP标签:FreeMarker。

FreeMarker

FreeMarker是一个开源项目，网址是<http://freemarker.sourceforge.net>。与Velocity类似，它同样是一种模板引擎，它能生成基于模板的文本输出。两种库的主要差别在于它的模板语言语法不同。FreeMarker更为健壮，使用起来更简单。它同样支持在你的模板中使用JSP标签(这意味你也可以使用`display`标签了!)。有关两者之间差别的更详细的信息，请参考FreeMarker vs. Velocity [<http://freemarker.sourceforge.net/fmVsVel.html>]。

下一节把MyUsers应用程序从Velocity转向FreeMarker。还是从在Spring中配置FreeMarker入手。本章要求你把你的`freemarker.jar`放在你的classpath中(`web/WEB-INF/lib`)。

注意

Spring要求FreeMarker2.3或更高的版本。

View Resolver配置

1. 在文件`web/WEB-INF/action-servlet.xml`中添加`configurer` bean定义。作为本练习的一部分，请注释掉`velocityConfig` bean。

```
<bean id="freemarkerConfig"
class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
  <property name="templateLoaderPath"><value>/</value></property>
</bean>
```

和`VelocityConfigurer`一样，你要用一个`properties`文件配置这个类(用一个`configLocation`属性指向这个文件)。你也可在这个bean本身上指定`freemarkerSettings`属性列表来设置属性(properties)。

2. 注释掉`velocityConfig` bean和`viewResolver` bean，为FreeMarker添加一个bean。这个resolver的属性和Velocity版本的非常相似，唯一不同的是“`prefix`”和“`suffix`”。

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.
freemarker.FreeMarkerViewResolver">
  <property name="exposeSpringMacroHelpers"><value>true</value></property>
```

```

<property name="requestContextAttribute"><value>rc</value></property>
<property name="prefix"><value>/WEB-INF/freemarker/</value></property>
<property name="suffix"><value>.ftl</value></property>
</bean>

```

请记住，你可以这个template resolver的`exposeRequestAttributes`属性和`exposeSessionAttributes`来暴露request属性和session属性。本教程中用不到它们，但它们很有价值。

SiteMesh和FreeMarker

和Velocity一样，SiteMesh有一个与FreeMarker相对应的decorator servlet，叫做FreeMarkerDecoratorServlet。它利用context属性来预填充数据模型，如下表所示。

表 6.4. SiteMesh FreeMarker Context对象

<code>\${base}</code>	<code>request.getContextPath()</code>
<code>\${title}</code>	解析过的页面title
<code>\${head}</code>	解析过的页面head
<code>\${body}</code>	解析过的页面body
<code>\${page}</code>	SiteMesh内部Page对象

FreeMarker支持读取request参数，request属性和session变量。这就是你为何看不到在Velocity用到的`$req`和`$res`的原因。这些值通过模板中的变量`Request`，`RequestParameters`，`Session`和`Application`进行访问(例如，`${Session["user"]}`)。

- 下面是上一节中Velocity decorator用FreeMarker重写的版本。清单仅仅显示了修改过部分，所以你不能复制整个文件的内容。用这几行代码作为创建`web/decorators/default.ftl`的一个指南。变量的引用和Velocity中一样，后面的逻辑语法有所不同。

提示

复制一份`web/decorators/default.vm`为`web/decorators/default.ftl`，为转换作准备。

```

<div id="content">
  <#include "/messages.ftl"/>
  ${body}
</div>
</div>
<div id="supportingText">
<div id="underground">
  <#if page.getProperty("page.underground")?exists>
    ${page.getProperty("page.underground")}
  </#if>
</div>
...
<a href="http://bobby.watchfire.com/bobby/
bobbyServlet?URL=${Request.requestURL}&output=Submit&gl=sec508&
test=" title="Check the accessibility of this site according to U.S.
Section 508">508</a> .

```



```
<a href="http://bobby.watchfire.com/bobby/
bobbyServlet?URL=${Request.requestURL}&output=Submit&gl=wcag1-
aaa&test=" title="Check the accessibility of this site according to
Content Accessibility Guidelines 1">aaa</a>
...
```

2. 创建文件web/messages.ftl，内容如下。

```
<!-- Success Messages -->
<#if message?exists>
<div class="message">${message}</div>
</#if>
```

3. 修改web/WEB-INF/decorators.xml，默认值设为FreeMarker。

```
<decorators defaultdir="/decorators">
  <decorator name="default" page="default.ftl">
    <pattern>*/</pattern>
  </decorator>
</decorators>
```

Velocity和FreeMarker之间值得注意的差别如下。

- 在Velocity中检测一个null值，你简单用`#if (object.property)`来检测这个值是否存在。使用FreeMarker时，你要追加`?exists`来测试null值。
- Velocity中暴露了个真实的`HttpServletRequest`对象，所以你可以调用`${request.contextPath}`和`${request.requestURL}`。FreeMarker仅会暴露作用域内的属性。因为如此，你无法简单的从文件messages.ftl的session中删除message。

FreeMarker有一些局限性，这也使得它成为一种洁净的MVC实现。解决上面问题最为简单有效的方法是创建一个ServletFilter来搜索session中的message，如果找到了，就把它们填充到request中。用这种方法，你完全可以忘记在view页面中删除它们，而让filter去完成这项工作。

下面是在src/org/appfuse/web中MessageFilter.java文件内容。在request中设置“requestURL”是为了在项目的footer页面中提供返回应用程序的链接。

```
package org.appfuse.web;
// use your IDE to organize imports

public class MessageFilter implements Filter {
    private static Log log = LogFactory.getLog(MessageFilter.class);

    public void doFilter(ServletRequest req, ServletResponse res,
        FilterChain chain)
        throws IOException, ServletException {
        HttpServletRequest request = (HttpServletRequest) req;
```

```
// grab messages from the session and put them into request
// this is so they're not lost in a redirect
Object message = request.getSession().getAttribute("message");
if (message != null) {
    request.setAttribute("message", message);
    request.getSession().removeAttribute("message");
}

// set the requestURL as a request attribute for templates
// particularly freemarker, which doesn't allow
// request.getRequestURL()
request.setAttribute("requestURL", request.getRequestURL());
chain.doFilter(req, res);
}

public void init(FilterConfig filterConfig) {}

public void destroy() {}
}
```

4. 使用这个filter, 你可以消除所有的message模板(web/messages.*)中任何session逻辑。要启用它, 在web/WEB-INF/web.xml中添加一个<filter>和一个<filter-mapping>。紧挨着sitemesh filter上面添加<filter>声明。

```
<filter>
  <filter-name>messageFilter</filter-name>
  <filter-class>org.appfuse.web.MessageFilter</filter-class>
</filter>
```

5. 在SiteMesh的filter-mapping上面, 添加<filter-mapping>。

```
<filter-mapping>
  <filter-name>messageFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

编辑web.xml文件

针对SiteMesh修改web.xml, 注释掉sitemesh-velocity servlet和它的mapping, 添加sitemesh-freemarker servlet和mapping。

```
<servlet>
  <servlet-name>sitemesh-freemarker</servlet-name>
  <servletclass>
    com.opensymphony.module.sitemesh.freemarker.FreemarkerDecoratorServlet
```

```

</servlet-class>
<init-param>
  <param-name>TemplatePath</param-name>
  <param-value>/</param-value>
</init-param>
<init-param>
  <param-name>default_encoding</param-name>
  <param-value>ISO-8859-1</param-value>
</init-param>
</servlet>
<servlet-mapping>
  <servlet-name>sitemesh-freemarker</servlet-name>
  <url-pattern>*.ftl</url-pattern>
</servlet-mapping>

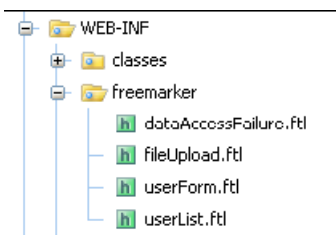
```

创建FreeMarker模板

实现FreeMarker的最后一步是用FreeMarker模板语言来创建页面模板。

1. 复制一份WEB-INF/velocity目录，另存为WEB-INF/freemarker，重新命名所有的文件，文件后缀改为ftl。你的目录结构看起来应该和图6.4的这个一样。

图 6.4. FreeMarker Templates 目录结构



2. 以下是上面显示的四份FreeMarker模板的一份清单，以及使用FreeMarker时进行的必要的修改。代码后面有每个文件的特别说明。与Velocity的VTL不同的FreeMarker代码都加上了下划线。

userList.ftl

```

<title>MyUsers ~ User List</title>
<button onclick="location.href='editUser.html'">Add User</button>
<table class="list" id="userList">
<thead>
<tr>
<th>${rc.getMessage("user.id")}</th>
<th>${rc.getMessage("user.firstName")}</th>
<th>${rc.getMessage("user.lastName")}</th>
</tr>
</thead>
<tbody>
<#list users as user>

```

```
<#if user_index % 2 == 0> <tr class="even">
<#else> <tr class="odd">
</#if>
<td><a href="editUser.html?id=${user.id}">${user.id}</a></td>
<td>${user.firstName}</td>
<td>${user.lastName}</td>
</tr>
</#list>
</tbody>
</table>
```

与 Velocity 类似，`${rc.getMessage() }` 调用会从文件 `web/WEB-INF/classes/messages.properties` 读取本地化 message。

userForm.ftl

```
<#import "/spring.ftl" as spring/>
<title>MyUsers ~ User Details</title>
<@spring.bind "user.*"/>
<#if spring.status.error>
<div class="error">
<#list spring.status.errorMessages as error>
${error}<br/>
</#list>
</div>
</#if>
<p>Please fill in user's information below:</p>
<form method="post" action="editUser.html">
<@spring.bind "user.id"/>
<input type="hidden" name="id"
value="${spring.status.value?default('')}" />
<table>
<tr>
<th>${rc.getMessage("user.firstName")}:</th>
<td>
<@spring.bind "user.firstName"/>
<input type="text" name="firstName"
value="${spring.status.value?default('')}" />
<span class="fieldError">${spring.status.errorMessage}</span>
</td>
</tr>
<tr>
<th>${rc.getMessage("user.lastName")}:</th>
<td>
<@spring.bind "user.lastName"/>
<input type="text" name="lastName"
value="${spring.status.value?default('')}" />
<span class="fieldError">${spring.status.errorMessage}</span>
</td>
</tr>
<tr>
```

```
<td></td>
<td>
<input type="submit" class="button" name="save" value="Save"/>
<#if user.id?exists>
<input type="submit" class="button" name="delete" value="Delete"/>
</#if>
</td>
</table>
</form>
```

这个文件中你要注意的是在第一行导入了spring.ftl。这个文件包含了spring.bind宏，用于暴露一个属性值。

你也可以注意到，每个字段通过在表达式的末尾添加?default('')来给定一个默认值。

这些宏并没有<spring:bind>JSP标签所必需一个关闭标签或是结束语句。当你把viewResolver bean的exposeSpringMacroHelpers属性值设置成true时，#springBind宏就是一个变量。

使用JavaScript和Commons Validator的客户端验证，仅支持使用JSP。

fileUpload.ftl

```
<h3>File Upload</h3>
<#if model?exists>
<p style="font-weight: bold">
Uploaded file (click to view): <a
href="{model.url}">${model.filename}</a>
</p>
</#if>
<p>Select a file to upload:</p>
<form method="post" action="fileUpload.html" enctype="multipart/formdata">
<input type="file" name="file"/><br/>
<input type="submit" value="Upload" class="button"
style="margin-top: 5px"/>
</form>
```

dataAccessFailure.ftl

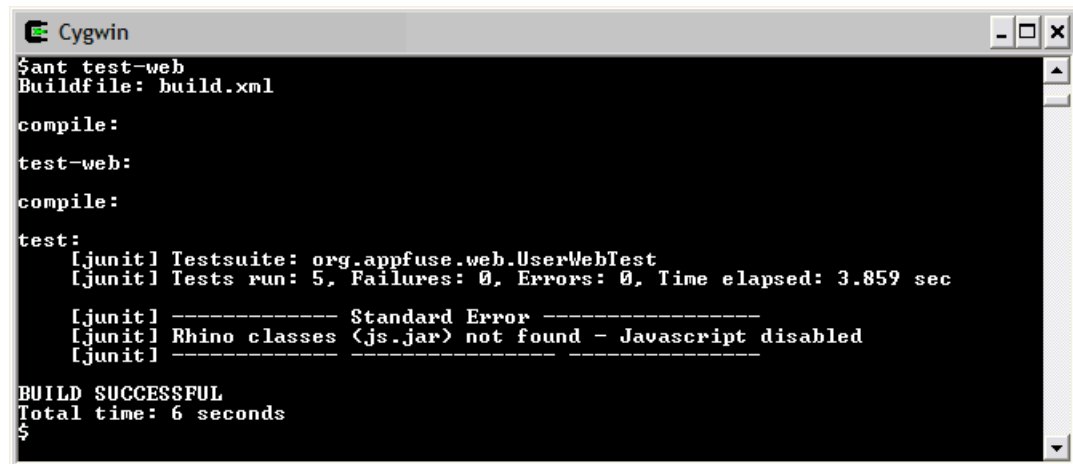
```
<h3>Data Access Failure</h3>
<p>
${exception}
</p>
<a href="{rc.contextPath}">« Home</a>
```

使用FreeMarker和Velocity时，此文件的代码是完全相同的。

部署和测试

现在你已经让Spring配置和使用FreeMarker，用一个Velocity Decorator配置好的SiteMesh，以及所有由JSP转换过来的FreeMarker。现在该测试一切是否运行正常。启动Tomcat，运行ant clean deploy reload。一旦新的context重启之后，执行命令ant test-web来验证运行情况。

图 6.5. FreeMarker测试结果



```

Cygwin
$ ant test-web
Buildfile: build.xml

compile:
test-web:
compile:
test:
[junit] Testsuite: org.appfuse.web.UserWebTest
[junit] Tests run: 5, Failures: 0, Errors: 0, Time elapsed: 3.859 sec
[junit] ----- Standard Error -----
[junit] Rhino classes (js.jar) not found - Javascript disabled
[junit] -----
BUILD SUCCESSFUL
Total time: 6 seconds
$

```

在使用Spring MVC时，JSP，Velocity和FreeMarker是占有统治地位的view选择。但其它的一些技术也可能很有用。它们是XSLT(用于显示和转换XML)，PDF和Excel。下面的几节中，你会用它们生成显示用户列表的报表。

XSLT

XSLT用于描述联合XML和XSLT，将XML转换成其它输出格式的过程。大多数情况下，这些输出是基于文件的，但是你也可以用XSL-FO来生成PDF。当你想加载和表示XML文档，或是你想简单的将你的模型转换成XML，Spring的XSLT view可能非常有用。

在创建一个XSLView之前，准备好UserController类来处理报表view的输出。

1. 打开UserControllerTest.java(在目录test/org/appfuse/web下)，添加以下测试。

```

public void testGetUsersAsXML() throws Exception {
    UserController c = (UserController) ctx.getBean("userController");
    MockHttpServletRequest request = new MockHttpServletRequest();
    request.addParameter("report", "XML");
    ModelAndView mav = c.handleRequest(request, (HttpServletResponse) null);
    assertEquals(mav.getViewName(), "userListXML");
}

```

2. 打开UserController.java(在目录src/org/appfuse/web中)，添加一些逻辑代码，如果传入一个report参数，就生成不同的view。在本章后面的部分，用此逻辑代码同样可以生成Excel和PDF View。

```
public ModelAndView handleRequest(HttpServletRequest request,
    HttpServletResponse response)
    throws Exception {
    if (log.isDebugEnabled()) {
        log.debug("entering 'handleRequest' method...");
    }
    String viewName = "userList";

    if (request != null && request.getParameter("report") != null) {
        viewName += request.getParameter("report");
    }

    return new ModelAndView(viewName, "users", mgr.getUsers());
}
```

3. 你想产生的报表仅包含以列表的方式显示用户的全名。为了偷懒，在User中添加一个getFullName()方法(在目录src/org/appfuse/model中)。

```
public String getFullName() {
    return firstName + ' ' + lastName;
}
```

创建View类

在创建一个类把用户列表转换成XML之前，新建一个类来进行测试。

1. 在目录test/org/appfuse/web，创建一个类UserXMLViewTest，它继承了JUnit的TestCase。

```
package org.appfuse.web;

// user your IDE to organize imports

public class UserXMLViewTest extends TestCase {

    private static Log log = LogFactory.getLog(UserXMLViewTest.class);

    public void testXMLCreation() throws Exception {
        // setup a user to print out as XML
        User user = new User();
        user.setFirstName("James");
        user.setLastName("Strachan");
        List users = new ArrayList();
        users.add(user);
        Map model = new HashMap();
        model.put("users", users);
    }
}
```

```
// invoke the XsltView and call its 'createDomNode' method
UserXMLView feed = new UserXMLView();
org.w3c.dom.Node node = feed.createDomNode(model, "users",
new MockHttpServletRequest(),
new MockHttpServletResponse());
assertEquals(node.getFirstChild().toString(),
"<users><user>James Strachan</user></users>");
}
}
```

2. 在目录src/org/appfuse/web中，创建一个UserXMLView，它继承了AbstractXsltView，文件内容如下。

注意

你所用到的XML类来自dom4j [http://www.dom4j.org]，它必须要在你的classpath中。还有Node返回类型(createDomNode())来自W3C的DOM(org.w3c.dom.Node)。

```
package org.appfuse.web;

// use your IDE to organize imports
public class UserXMLView extends AbstractXsltView {
    protected Node createDomNode(Map model, String rootName,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
        Document doc = DocumentHelper.createDocument();
        Element root = doc.addElement(rootName);
        doc.setRootElement(root);
        List users = (List) model.get("users");
        for (Iterator it = users.iterator(); it.hasNext();) {
            User user = (User) it.next();
            root.addElement("user").addText(user.getFullName());
        }
        response.setContentType("text/xml");
        return new DOMWriter().write(doc);
    }
}
```

3. 运行ant test -Dtestcase=UserXML看看是否得到你所期望的结果。

注意

在本例中，你会利用此XML文件来产生一个新XML文档(那就是为何把response设置成text/xml的原因)。如果你打算在XSL样式表中输出HTML，请删除这一行。

4. 在目录web/WEB-INF/xsl，创建文件users.xsl，向文件中添加以下XSL。

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
```



```
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" omit-xml-declaration="no"/>
<xsl:template match="/">
<users>
<xsl:for-each select="users/user">
<user><xsl:value-of select="."/></user>
</xsl:for-each>
</users>
</xsl:template>
</xsl:stylesheet>
```

5. 现在你已经创建所有的view文件，现在配置Spring让它能够识别userListXML名称。最简单的方法是再创建一个viewResolver，使用ResourceBundleViewResolver来解析这个View。既然你已经为FreeMarker创建了一个viewResolver view，你必须为这个bean指定一个不同的id。采用reportViewResolver是个不错的主意。

```
<bean id="reportViewResolver"
class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
<property name="order"><value>1</value></property>
</bean>
```

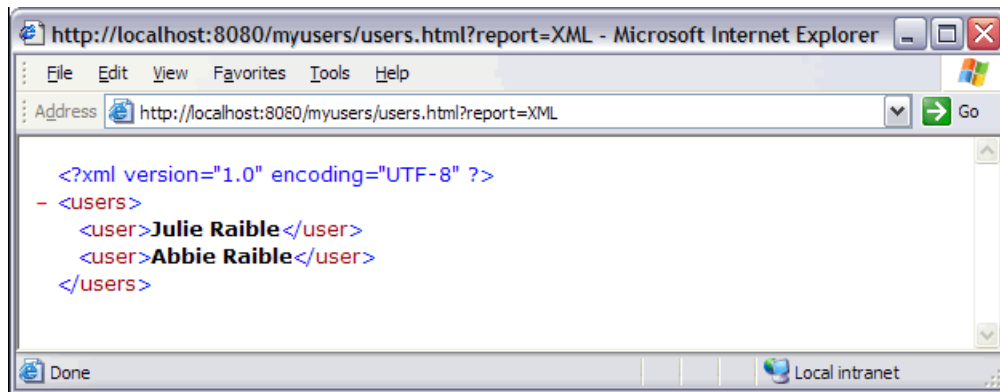
6. “order”属性指定了view resolver的优先级。为viewResolver添加一个同样的属性，值设为0。ResourceBundleViewResolver允许你在ResourceBundle(或是properties文件)中配置你的view名称，它们的类及其属性(property)。默认情况下，这个文件名为views.properties，而且应该在目录WEB-INF/classes中。如果你想重新命名，你必须为reportViewResolver指定一个“basename”属性。
7. 在目录web/WEB-INF/classes下创建一个view.properties文件，向文件中填充以下内容。

```
userListXML.class=org.appfuse.web.UserXMLView
userListXML.stylesheetLocation=/WEB-INF/xsl/users.xsl
userListXML.root=users
```

部署和测试

现在你可以准备部署和测试这个程序，运行ant deploy reload populate，在浏览器中打开http://localhost:8080/myusers/users.html?report=XML。你应该可以看到与下面屏幕类似的东西。

图 6.6. View类测试结果



一个报表已经产生了，添加Excel和PDF形式的报表。较简单的方法是，在当前的userlist中添加一系列的链接。如果还在用FreeMarker文件是userList.ftl，位于web/WEB-INF/freemarker中。打开文件，在文件的顶部的<button>和<table>之间添加以下HTML代码。

```
<p style="text-align: right; margin-bottom: -10px">
<strong>Export Options:</strong>
<a href="?report=XML">XML</a> .
<a href="?report=Excel">Excel</a> .
<a href="?report=PDF">PDF</a>
</p>
```

Excel

Excel文档对于用户要对输出的数据进行处理时是一种不错的方法。如果你只需要简单的在Excel中输出一个列表，我建议你使用Display Tag Library(前面介绍过)。如果你想寻找一种更为健壮的方法，这一节正是为你准备的。在下面的几个步骤中，将向你演示如何使用Jakarta的PIO库来构建和发送Excel表格。

注意

如果你下载并且安装了文件chapter6-jars.zip，你的classpath将会包含poi-2.5-final-20040302.jar，这个jar文件包含了要完成本练习所需要的类。

创建View类

1. 在目录src/org/appfuse/web创建一个UserExcelView类，这个类继承了AbstractExcelView，包含以下代码。

```
package org.appfuse.web;
//use your IDE to organize imports

public class UserExcelView extends AbstractExcelView {
    protected void buildExcelDocument(Map model,
        HSSFWorkbook wb,
        HttpServletRequest req,
```

```

        HttpServletResponse resp)
    throws Exception {
        HSSFWorkbook sheet = wb.createSheet("My Users");
        sheet.setDefaultColumnWidth((short) 12);
        List users = (List) model.get("users");
        for (int i = 0; i < users.size(); i++) {
            HSSFCell cell = getCell(sheet, i, 0);
            setText(cell, ((User) users.get(i)).getFullName());
        }
    }
}

```

Spring如何得知存在一个userListExcel，很简单，使用ResourceBundleViewResolver。

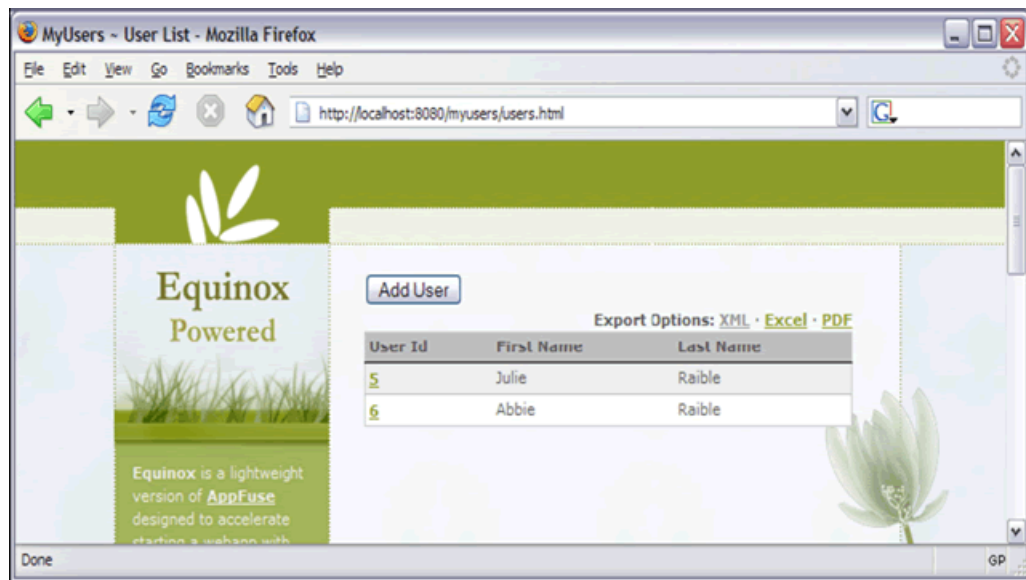
2. 打开目录web/WEB-INF/classes/views.properties文件，添加下面一行代码。

```
userListExcel.class=org.appfuse.web.UserExcelView
```

部署和测试

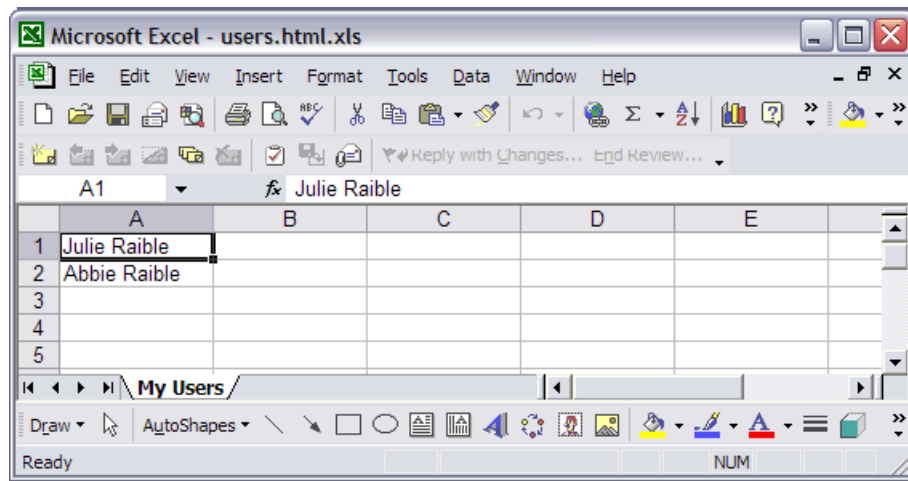
1. 运行ant deploy reload，浏览器指向http://localhost:8080/myusers/users.html。

图 6.7. Excel View测试结果



2. 点击表格右上角的Excel链接。如果你安装了Excel，它会启动Excel，显示当前用户的一个清单。

图 6.8. Excel中显示的结果



PDF

当你想生成可移植性文档时，PDF文档是一种非常好的选择。在大多数MVC框架中，你要用到一些诸如JasperReports之类的工具来输出PDF文件。使用Spring，你只要创建一个AbstractPdfView的子类，复写它的buildPdfDocument()方法。这个方法(method)会返回一个iText文件，这在浏览器中很容易查看。

注意

如果你下载并且安装了文件chapter6-jars.zip，你的classpath将会包含itext-1.02b.jar，这个jar文件包含了要完成本练习所需要的类。

创建View类

1. 在目录src/org/appfuse/web创建一个UserPDFView，在这个类中填充以下代码。

```
package org.appfuse.web;
// use your IDE to organize imports

public class UserPDFView extends AbstractPdfView {
    protected void buildPdfDocument(Map model, Document doc,
        PdfWriter writer,
        HttpServletRequest req,
        HttpServletResponse resp)
        throws Exception {
        List users = (List) model.get("users");
        for (int i = 0; i < users.size(); i++) {
            String fullName = ((User) users.get(i)).getFullName();
            doc.add(new Paragraph(fullName));
        }
    }
}
```

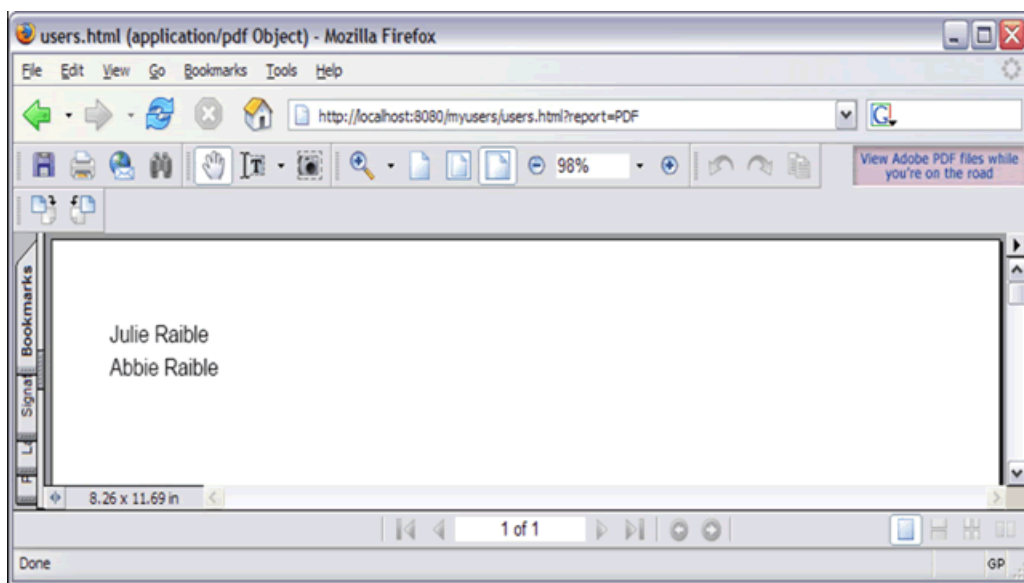
2. 为了让Spring知道userListPDF的位置，打开文件web/WEB-INF/classes/views.properties添加下面一行代码。

```
userListPDF.class=org.appfuse.web.UserPDFView
```

部署和测试

1. 运行ant deploy reload，浏览器指向http://localhost:8080/myusers/users.html。
2. 点击表格上面的PDF链接，将会打开一个包含用户列表的PDF。

图 6.9. PDF中显示的结果



本章小结

本章探讨了Spring MVC框架中提供的丰富的功能。向你演示了从JSP转向Velocity，而不需要修改一行Java代码。然后演示了通过更换一些XML文件，从而转向FreeMarker，只要对模板进行少量的修改工作。你已经学到了如何结合Velocity和FreeMarker使用SiteMesh。对于快速开发web应用程序来说，能够结合一种与SiteMesh一样的能高度配置的页面修饰引擎，来使用所有的J2EE模板，是一种非常有力的途径。使用Spring，从一种模板转向另一种模板技术变得非常容易，所以面临模板技术的选择不再生令人头痛的事。

能够产生Excel和PDF格式的报表是Spring MVC另一个功能强大而简单易用的特性。更好的是这些特性的背后是一些实力雄厚，有良好支持的开源项目，如iText, PIO。本章报表输出的例子很简单，主要目的是向你演示如何发球，而不是赢得比赛。

第 7 章 持久性策略

Hibernate, iBATIS, JDBC, JDO和OJB

Hibernate很快成为Java应用程序持久层的一种流行的选择,但有时它不适合.如果你有一个已经存在的数据库模型,或者预先写好的SQL的语句,有时使用JDBC或是iBATIS(支持XML文件形式的SQL语句)更适合一些。这一章对MyUsers进行重构,使它同时支持JDBC和iBATIS,作为持久层框架的可选方案。这一章也用JDO和OJB实现了UserDAO以展示Spring能很好的支持这些框架。

概述

当前大多数应用程序与数据库交互是为了存取资料。持久性是指从数据存储设施(通常是一个关系数据库)中读取,保存,或删除数据的过程。持久性是web应用程序中的一个重要的特性,只要用到读取和显示数据。多少年来,这一直是Java最令人厌恶的部分。诚然你可以使用JDBC,但依赖数据库提供商。毕竟,JDBC API为从Java访问数据库提供了一个统一的标准。但是,JDBC代码编写确不容易,特别是新手。它要求开发人员在final块中抓取异常和关闭数据库连接,但是很多Java新手常常忘记了这么做。另外,各个JDBC驱动程序提供商抛出的异常是没有标准的,所以在某个服务器上得到一个错误代码表明某种意义,而在另一个上表示的却完全不同。

随着Spring持久性支持的出现,所有JDBC的瑕疵都不复存在了。它的JDBC框架将JDBC的检测式的异常转化一个公共的RuntimeException结构。这些异常提供了精确的出错信息,这比SQLException报错要友好一些。它使用closure来处理数据连接关闭操作,它还包括了一组公共的SQL错误代码,针对各种不同数据库类型。

注意

一个closure是一个对象,可以描绘为一个代码块(在一个方法内)。你可以把个对象当任何Java一样用,如参数,变量等。更多信息,参考Java Glossary [<http://mindprod.com/jgloss/closure.html>]。Charles Miller还有一篇教程tutorial on Closures and Java [http://fishbowl.pastiche.org/2003/05/16/closures_and_java_a_tutorial]。

Spring提供了很多支持其它框架的类,如Hibernate, iBATIS, JDO, OJB。它使用了同一种机制,以进一步减少从一种框架转到另一框架的学习难度。

在本章中,你将进一步学习Spring的Hibernate支持技术,以及MyUsers是如何使用Hibernate。接下来,你将用iBATIS, Spring JDBC, JDO和OJB来实现DAO类。在本章的后面,你就会清楚如何运用这些技术,并且如何在Spring中进行配置。本章的目的是为了向你简短的介绍Spring中各种技术选择以及如何配置它们。但不会涉及到用每种框架来解决复杂的持久性问题。这方面的资料请参考框架的项目主页。

注意

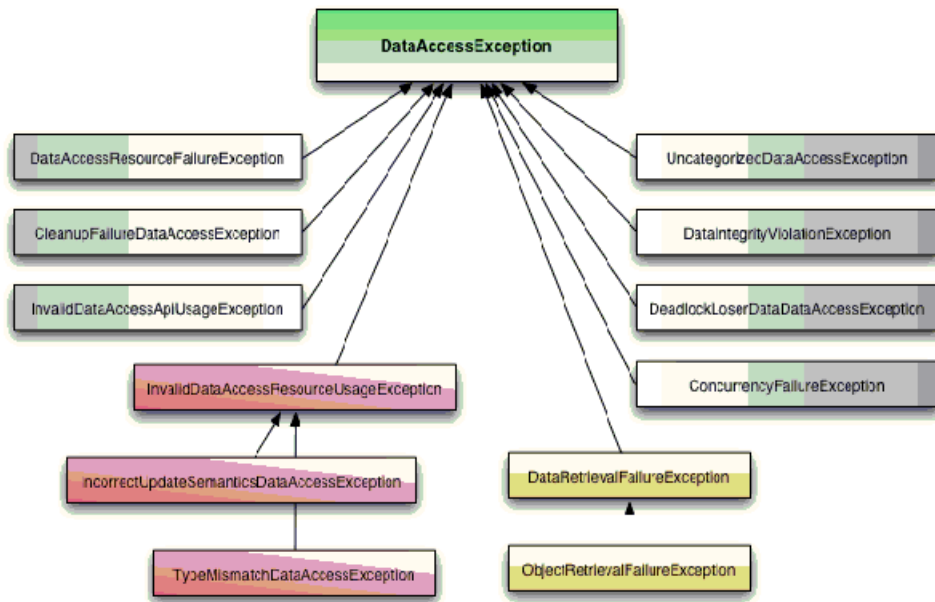
事务处理(Transaction)将在第十章介绍。

谈及持久性时, Spring有一些共同的话题。

- 利用依赖注射提供简易的配置。
- 为流行的持久性框架(如, Hibernate, JDBC, iBATIS, JDO和OJB)提供了DaoSupport基类。
- 将各种框架的检测式异常(checked exception)转换成公用的异常继承类,把你的高级类从框架的特殊性中解脱出来。图7.1演示这种继承关系。

- 提供简单易用的Template类，把很多DAO方法减少为一行代码。这些模板可以单独使用，或从DaoSupport的父类中获得。
- 为了打开和关闭任何必要的资源(例如数据库连接)。

图 7.1. 共用异常的继承关系



本章提供的代码很容易进行测试，因为你已经有一个UserDAOTest类，其它不含任何Hibernate特性的代码。这个测试会验证你的实现。每个持久性框架一节会向你演示如何在Spring配置每种框架，以及如何实现UserDAO类。

准备练习

学习最佳方法是跟着做本章的练习。最快捷的立法是从<http://sourcebeat.com/downloads>。[<http://sourcebeat.com/downloads>]下载MyUsers Chapter 7压缩包。项目的树型目录结构与第六章练习完成时一样，并且包含了本章中要用到的所有的jar文件。你也可以使用前一章开发的应用程序。如果准备这么做，请从<http://sourcebeat.com/downloads>。[<http://sourcebeat.com/downloads>]上下载Chapter 7 JARs。在每一节的配置小话题中会描述每个项目的依赖关系。你可以将本章作为一个集成各种原理到你的应用中的一个参考。

如果你要使用前一章中使用的应用程序，你必须变换方法来加载context文件。不仅仅要加载applicationContext.xml，修改一下test和web.xml来加载applicationContext*.xml。这是为方便。当我们为每种持久性技术创建context文件时，指定通配符来匹配一个文件要容易些。但是你还是修改tests和web.xml文件来切换持久层。

这些修改会影响以下文件：

- test/org/appfuse/dao/BaseDAOTestCase.java
- test/org/appfuse/service/UserManagerTest.java
- test/org/appfuse/web/UserControllerTest.java

- test/org/appfuse/web/UserFormControllerTest.java
- web/WEB-INF/web.xml

因为你要将所有的数据库配置文件统一到一个文件中，你必须把文件applicationContext.xml中的<ref local>的指令修改<ref bean>，这样其它文件才能包含bean定义。本章中的这个基线applicationContext.xml文件没有包含以下bean定义：dataSource，sessionFactory，transactionManager和UserDAO。如果你用的版本中还是存在，请删除它们。上面所做的种种修改是为了让你为不同的DAO实现创建不同的context文件。

Hibernate

Hibernate是一种开源的对象关系映射(ORM)解决方案。ORM是一门将对象模型映射到关系模型(通常是SQL数据库)的技术。Hibernate是由Gavin King和其它一些开发人员在2001年底创建的。之后，Hibernate成了Java社区中一种非常流行的持久性框架。正因为它变得如此的流行，迟到下一代的EJB和JDO借鉴Hibernate中好的思想。它流行的原因主要归结于它有很好的文档，易用，优秀的特性，智能项目管理(smart project management)。Hibernate使用的协议是LGPL，这就是意味你只要不修改源代码可以自由使用。有关协议的更多信息请参考Hibernate网站[<http://www.hibernate.org/196.html>]。

Hibernate使你从手工编写JDBC中解脱出来。无需再用SQL和JDBC，你可以使用域对象(通常是POJO)简单的创建基于XML的映射文件。这些文件指明了哪些字段(对象中)映射到哪些列(数据表中)。Hibernate拥有一个强大的查询语言叫做Hibernate查询语言(HQL)。这种语言允许你使用SQL，但使用的是面向对象的语义。它的查询语言最大好处之一，就是你可以从语法字面上进行猜测，正确的使用。

Hibernate的Session接口和数据库连接相似，那就是在适当的时候要进行打开和关闭操作，以免出现错误和内存溢出。以我的看法，结合Hibernate使用Spring的最大好处就是你不必管理打开和关闭Session操作，一切照常运行。

注意

Spring的Hibernate支持类位于包org.springframework.orm.hibernate和org.springframework.orm.hibernate.support中。虽然第2章已经提及了如何集成Hibernate到MyUsers程序中。本章同样涉及了Hibernate是为了提供一个单独的章节来描述Spring的Hibernate持久性的支持。如果你下载了myusers-ch7文件，Hibernate的配置已经被清除了，这样从一个干净的记录开始。

依赖包

Hibernate依赖好几个第三方包。所有的文件作为一部分提供在Hibernate下载文件[<http://hibernate.org/6.html>]中。下面的jar文件作为Hibernate2.1.6的一部分已经包含在MyUsers下载文件中。这些就是所有需要的包，除了一个标为可选的外。Spring要求它的支持类使用Hibernate2.1或以上版本。

- hibernate2.jar:Hibernate核心。
- c3p0-0.8.4.5.jar:基本的连接池，用于运行单元测试。
- cglib-full-2.0.2.jar:代码生成库文件，为持久类生成代理。
- dom4j-1.4.jar:XML库，用于解析配置文件和映射文件。
- ehcache-0.9.jar:一个纯Java，进程内的缓存。Hibernate的默认缓存。

- jta.jar:Java Transaction API。
- odmng-2.0.1.jar:对象关系映射产品标准，由JDO取代。
- (可选)oscache-2.0.1.jar和(cluster-aware)swarmcache-1.0rc2.jar:备选缓存实现方案。

注意

现在Spring还不支持Hibernate 3 (还处于beta版)，需要打一个补丁[<http://opensource.atlassian.com/projects/spring/browse/SPR-300>]。

配置

为了使用Hibernate来保持对象的持久性，首先创建一个映射文件。(本章假定你已经在src/org/appfuse/model中创建了User POJO，它有id，firstName和lastName等属性。)

1. 在目录src/org/appfuse/model创建一个文件User.hbm.xml，内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 2.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
  <class name="org.appfuse.model.User" table="app_user">
    <id name="id" column="id" unsaved-value="0">
      <generator class="increment"/>
    </id>
    <property name="firstName" column="first_name"
      not-null="true"/>
    <property name="lastName" column="last_name" not-null="true"/>
  </class>
</hibernate-mapping>
```

在上面的映射中，<id>元素用“increment”来指明用max value+1来表示生成的主键。“increment”这种生成器的方式对群集是不推荐使用的。幸运的是，Hibernate有多种选择。

2. 在目录web/WEB-INF创建一个文件applicationContext-hibernate.xml，添加一个DataSource bean定义。你可以用现有的applicationContext*.xml作为模板。在进行bean定义之前，此文件应该已经包含spring-beans.dtd和根beans元素定义。

本例中，dataSource bean使用了一个HSQL数据库，这是一个纯Java数据库，运行时通过目录web/WEB-INF/lib的hsqldb.jar文件来实现。然后，你会修改配置，转用MySQL，你可以看到转换数据库是多么容易。

```
<?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.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName">
```

```

    <value>org.hsqldb.jdbcDriver</value>
  </property>
  <property name="url">
    <value>jdbc:hsqldb:db/appfuse</value>
  </property>
  <property name="username"><value>sa</value></property>
  <property name="password"><value></value></property>
</bean>
<!-- Add additional bean definitions here -->
</beans>

```

DriverManagerDataSource是一个简单的DataSource，它通过bean属性来配置一个JDBC驱动程序。如果你想用容器预配置的DataSource，你也可以配置一个JNDIDataSource。例如，常用的策略是，使用DriverManagerDataSource进行测试，在成品时使用JNDIDataSource(如下)。

注意

第8章会向你演示模拟一个JNDI DataSource，并在测试和发布产品时使用以下配置。

```

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

```

3. 添加一个sessionFactory 定义，它依赖前面定义的dataSource bean定义和映射文件。dialect会根据数据库不同而不同，hibernate.hbm2ddl.auto属性会在应用程序启动时创建数据库。你可能注意到dataSource的引用(从前一章)发生了变化，使用<ref bean>代替<ref local>。这样你就可以在不同的文件中定义dataSource，可以从测试版本切换到容器内的版本。

```

<bean id="sessionFactory"
  class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
  <property name="dataSource"><ref bean="dataSource"/></property>
  <property name="mappingResources">
    <list>
      <value>org/appfuse/model/User.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">
        net.sf.hibernate.dialect.HSQLDialect
      </prop>
      <prop key="hibernate.hbm2ddl.auto">update</prop>
    </props>
  </property>

```

```
</bean>
```

作为映射到各个.hbm.xml的一种备选方案，使用mappingJarLocations或mappingDirectoryLocations来指向jar文件和目录位置。你也可以在文件hibernate.cfg.xml指定你的设置，使用configLocation属性指向它。

4. 添加一个transactionManager定义，它使用了Spring的HibernateTransactionManager类。这个类的javadoc对它作了最好的解释：“这个类由指定的factory绑定一个Hibernate Session到线程上，这意味你可以为个factory开一个线程。SessionFactoryUtils和HibernateTemplate会监控依附线程的Session，并且会自动地参与事务处理。对于用于Hibernate支持事务处理机制的访问代码，两个都是要用到的。”

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

UserHibernateDAO和UserDAOTest并没有明确的使用这个bean，但是userManager的bean定义中引用到它。在JDO一节中将向你演示如何在DAO测试中利用事务。

5. 在src/org/appfuse/dao/hibernate(你可能要创建这个目录/包)创建类UserDAOHibernate，这个类继承了HibernateDaoSupport并实现了UserDAO。

注意

如果你用前一章开发的MyUsers应用程序继续开发。修改UserDAO中的saveUser()方法，将User(而不是object)作为一个参数。

```
package org.appfuse.dao.hibernate;
// use your IDE to organize imports

public class UserDAOHibernate extends HibernateDaoSupport implements UserDAO {
  public List getUsers() {
    return getHibernateTemplate().find("from User");
  }

  public User getUser(Long id) {
    User user = (User) getHibernateTemplate().get(User.class, id);
    if (user == null) {
      throw new ObjectRetrievalFailureException(User.class, id);
    }
    return user;
  }

  public void saveUser(User user) {
    getHibernateTemplate().saveOrUpdate(user);
  }

  public void removeUser(Long id) {
```

```

        Object user = getHibernateTemplate().load(User.class, id);
        getHibernateTemplate().delete(user);
    }
}

```

6. 添加一个userDAO的bean定义到applicationContext-hibernate.xml文件中。

```

<bean id="userDAO" class="org.appfuse.dao.hibernate.UserDAOHibernate">
    <property name="sessionFactory">
        <ref local="sessionFactory"/>
    </property>
</bean>

```

在UserDAOHibernate类中，HibernateTemplate完成了大部分工作。使用模板来处理持久层调用，是Spring DAO支持类中一个共同的话题。还有要注意UserDAOHibernate类中的以下几点。

- getUser()使用了HibernateTemplate().get()，如果没有找到匹配的对象返回null。备选还有HibernateTemplate().load()，如果找不到对象会抛出异常。HibernateTemplate.get()在removeUser()中用到，你可以简单的用get()代替。
- getUser()在找不到user时抛出一个异常ObjectRetrievalFailureException。
- 这里没有用到测试式异常。使用Hibernate你或许要结束写大量的try/catch语句了。

HibernateDaoSupport中已经定义了一个logger变量，你可以在中子类中记录日志。例如，在saveUser()末尾部分添加以下代码。

```

if (logger.isDebugEnabled()) {
    logger.debug("User's id set to: " + user.getId());
}

```

测试

如果你用前一章的MyUsers应用程序进行开发，在能够顺利运行它之前，你要对UserDAOTest.testGetUsers()作一些修改。新方法如下所示。其中删除了在测试前检查空数据库表以及在测试后存在唯一记录的断言。其中的原因是将hibernate.hbm2ddl.auto(属性于sessionFactorybean)属性设置成了update。这会改变Hibernate的动作，在JVM启动时更新这些schema(而不是每次都要新建)。这样做的主要目的是其它框架要创建或是删除表并不容易，并且你想要是一种兼容的单元测试。

```

public void testGetUsers() {
    // add a record to the database so we have something to work with
    user = new User();
    user.setFirstName("Rod");
    user.setLastName("Johnson");
    dao.saveUser(user);
}

```

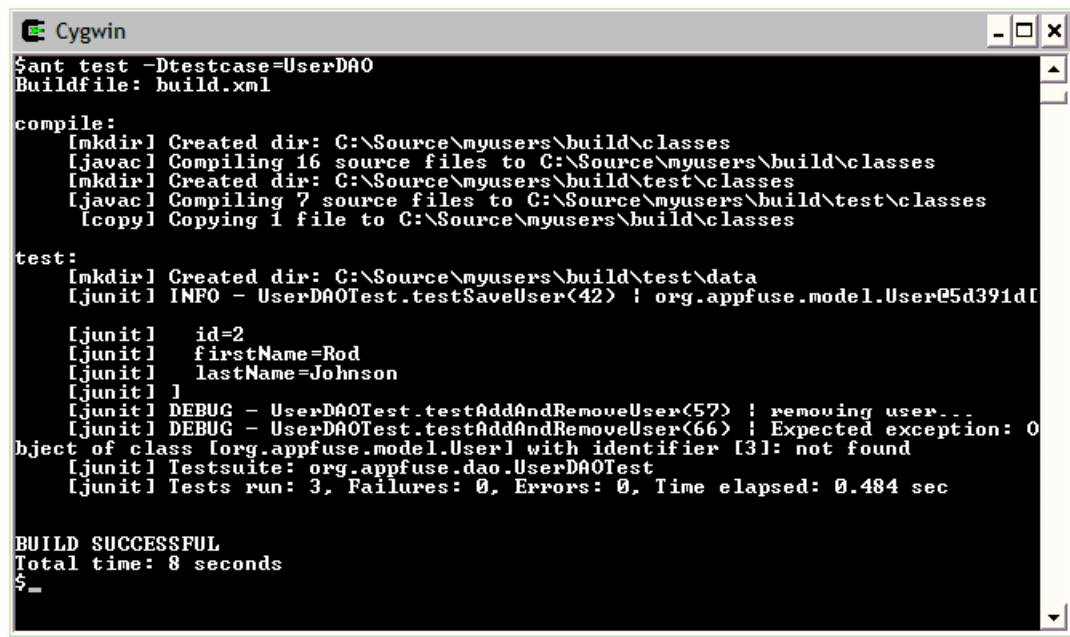
```

List users = dao getUsers();
assertTrue(users.size() >= 1);
assertTrue(users.contains(user));
}

```

运行 `ant test -Dtestcase=UserDAO`。得到的结果应该和图7.2相似。

图 7.2. 运行 `ant test -Dtestcase=UserDAO` 的测试结果



```

Cygwin
$ ant test -Dtestcase=UserDAO
Buildfile: build.xml

compile:
[mkdir] Created dir: C:\Source\myusers\build\classes
[javac] Compiling 16 source files to C:\Source\myusers\build\classes
[mkdir] Created dir: C:\Source\myusers\build\test\classes
[javac] Compiling 7 source files to C:\Source\myusers\build\test\classes
[copy] Copying 1 file to C:\Source\myusers\build\classes

test:
[mkdir] Created dir: C:\Source\myusers\build\test\data
[junit] INFO - UserDAOTest.testSaveUser(42) ! org.appfuse.model.User@5d391d1f
[junit] id=2
[junit] firstName=Rod
[junit] lastName=Johnson
[junit] ]
[junit] DEBUG - UserDAOTest.testAddAndRemoveUser(57) ! removing user...
[junit] DEBUG - UserDAOTest.testAddAndRemoveUser(66) ! Expected exception: 0
Object of class org.appfuse.model.User1 with identifier [3]: not found
[junit] Testsuite: org.appfuse.dao.UserDAOTest
[junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 0.484 sec

BUILD SUCCESSFUL
Total time: 8 seconds
$ _

```

MySQL配置

从HSQL转到MySQL非常容易。感谢Spring，这仅仅是配置设置的问题。

1. 确保安装了MySQL，并且正在运行。如果你使用前一个章开发的MyUsers程序，要为本章安装额外的 `jar` 文件或都下载 `MySQL JDBC Driver` [<http://dev.mysql.com/downloads/connector/j/3.0.html>]，复制到目录 `web/WEB-INF/lib` 下。
2. 在文件 `applicationContext-hibernate.xml` 中，把 `dataSource` 修改成下面的样子。

```

<property name="driverClassName">
  <value>com.mysql.jdbc.Driver</value>
</property>
<property name="url">
  <value>jdbc:mysql://localhost/myusers</value>
</property>
<property name="username"><value>root</value></property>
<property name="password"><value></value></property>

```

默认安装时，使用 `root` username，password 为空。你应该根据安装作相应的调整。

提示

你可以使用PropertyPlaceholderConfigurer从一个properties文件中设置以上的属性值。这会在第8章中介绍。

3. 将sessionFactory的hibernate.dialect属性修改为MySQL。

```
<prop key="hibernate.dialect">
  net.sf.hibernate.dialect.MySQLDialect
</prop>
```

4. 在运行单元测试或是启动Tomcat之前，创建MyUsers数据库。

```
mysqladmin -u root -p create myusers
```

如果你想在MySQL中使用browser target，你还可以修改build.xml文件中browser target的driver, url, userid属性。

运行ant test -Dtestcase=UserDAO，你应该可以得到图7.2同样的结果。

缓存

持久性框架一个强大的特性就是可以缓存(data)数据以避免频繁的访问数据库。Hibernate Session是持久性数据一个事务级的缓存，但是它在JVM或是群集层无法为每个类或是一个接一个collection提供缓存。但是，不管JVM还是群集级别，它都提供了拨插功能(这就是所谓的二级缓存)。关于一份所支持的缓存完整的清单，请参考Hibernate's Second Level Cache documentation [http://www.hibernate.org/hib_docs/reference/en/html/performance.html#performance-cache]。

下面的例子向你演示了如何为JVM级缓存配置EHCache。

注意

EHCache是默认的缓存，你没有必要在applicationContext-hibernate.xml配置一个hibernate.cache.provider_class设置。

1. 要对象启用缓存的最简单的方法，是添加一个<cache>到它的映射文件中。User对象要使用的话，向src/org/appfuse/model目录下的User.hbm.xml中添加一个<cache>元素。可供选择的值有read-write和read-only。如果你引用的对象或是表很少修改，你应该使用第2个选项。

```
<class name="org.appfuse.model.User" table="app_user">
  <cache usage="read-write"/>
  <id name="id" column="id" unsaved-value="0">
```

2. (可选)在EHCache配置文件中对这个类进行设置。在目录web/WEB-INF/classes中新建文件ehcache.xml，填充以下XML代码。

```
<ehcache>
  <!-- Only needed if overFlowToDisk="true" -->
  <diskStore path="java.io.tmpdir"/>
  <!-- Required element -->
  <defaultCache
    maxElementsInMemory="10000"
    eternal="false"
    timeToIdleSeconds="120"
    timeToLiveSeconds="120"
    overflowToDisk="true"/>
  <!-- Cache settings per class -->
  <cache name="org.appfuse.model.User"
    maxElementsInMemory="1000"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
    overflowToDisk="true"/>
</ehcache>
```

3. 为了证明User对象被缓存了，在文件web/WEB-INF/classes/log4j.xml中打开debug，记录EHCache。

```
<logger name="net.sf.ehcache">
  <level value="DEBUG"/>
</logger>
```

4. 运行ant test -Dtestcase=UserDAO，得到的结果应该和图7.3相似。

图 7.3. 运行ant test -Dtestcase=UserDAO的测试结果

```

Cygwin
$ant test -Dtestcase=UserDAO
Buildfile: build.xml

compile:

test:
[junit] DEBUG - CacheManager.create(178) ! Creating new CacheManager with default config
[junit] DEBUG - CacheManager.configure(152) ! Configuring ehcache from classpath
[junit] DEBUG - Configurator.configure(120) ! Configuring ehcache from ehcache.xml found in the classpath: file:/C:/Source/myusers/web/WEB-INF/classes/ehcache.xml
[junit] DEBUG - Configuration$DiskStore.setPath(182) ! Disk Store Path: c:\DOCUMENTS\mraible\LOCALS~1\Temp\...
[junit] DEBUG - MemoryStore.loadMapInstance(136) ! org.appfuse.model.UserCache: Using SpoolingLinkedHashMap implementation
[junit] DEBUG - MemoryStore.<init>(117) ! initialized MemoryStore for org.appfuse.model.User
[junit] DEBUG - CacheManager.create(183) ! Attempting to create an existing instance. Existing instance returned.
[junit] DEBUG - CacheManager.create(183) ! Attempting to create an existing instance. Existing instance returned.
[junit] DEBUG - UserDAOHibernate.saveUser(28) ! userId set to: 3
[junit] DEBUG - MemoryStore.get(193) ! org.appfuse.model.UserCache: MemoryStore miss for 1
[junit] DEBUG - Cache.get(286) ! org.appfuse.model.User cache - Miss
[junit] DEBUG - Cache.isExpired(741) ! 1 now: 1095454890092
[junit] DEBUG - Cache.isExpired(742) ! 1 Creation Time: 1095454890092 Next Time: Last Access Time: 0

```

Hibernate参考文档中有更多有关using and configuring Second Level Caches [http://www.hibernate.org/hib_docs/reference/en/html/performance.html#performance-cache]的资料。一般来说，不应该为应用程序配置缓存，而应该对数据进行调优(如，使用索引)。这实现缓存仅仅是为了演示而已。

延迟导入(lazy-load)依赖对象

Hibernate的众多特性之一就是能够延迟导入依赖的对象。一个由user对象组成的list引用了一个由role对象组成的collection。你可能要加载role来显示user清单。如果给role collection作上标记lazy-load="true"，在你对他们(通常在UI中)进行操作之前，它是不会被加载的。

为了在Spring使用这一点特性，在你的应用种配置好OpenSessionInViewFilter。当请求某个特定的URL时，会打开一个session，当页面完成加载后，就会关闭它。要启用这一特性，向web.xml添加下面的XML代码。

```

<filter>
  <filter-name>hibernateFilter</filter-name>
  <filter-class>
    org.springframework.orm.hibernate.support.OpenSessionInViewFilter
  </filter-class>
</filter>
<filter-mapping>
  <filter-name>hibernateFilter</filter-name>
  <url-pattern>*.html</url-pattern>
</filter-mapping>

```

当运行DAO测试时，使用这一特性会导致以下错误。


```
[junit] net.sf.hibernate.LazyInitializationException: Failed to lazily
initialize a collection - no session or session was closed
```

为了修正这一错误，在你的测试中添加以下代码到`setUp()`和`tearDown()`方法。

```
protected void setUp() throws Exception {
    // the following is necessary for lazy loading
    sf = (SessionFactory) ctx.getBean("sessionFactory");
    // open and bind the session for this test thread.
    Session s = sf.openSession();
    TransactionSynchronizationManager
        .bindResource(sf, new SessionHolder(s));
    // setup code here
}

protected void tearDown() throws Exception {
    // unbind and close the session.
    SessionHolder holder = (SessionHolder)
        TransactionSynchronizationManager.getResource(sf);
    Session s = holder.getSession();
    s.flush();
    TransactionSynchronizationManager.unbindResource(sf);
    SessionFactoryUtils.closeSessionIfNecessary(s, sf);
    // teardown code here
}
```

众所周知，使用`open-session-in-view`模式是Hibernate Session处理的惯用手法。一些Spring开发人员建议在`service`层初始化所有很必要的数据库。一次性的加载所有的数据会更加有效，并能保证提供同样的数据给所有的client。

Hibernate网站 [<http://hibernate.blumars.net/118.html>]上有很多Hibernate相关的小技巧。

社区和支持

Hibernate社区是非常活跃的。另外，它有很好的文档 [<http://www.hibernate.org/5.html>]和良好支持的用户论坛 [<http://www.hibernate.org/20.html>]，它在上万的开发人员中非常流行。它发布较早，并且常常更新，每次发布平均下载有30000。

Hibernate站点是有一份谁在用Hibernate [<http://www.hibernate.org/113.html>]的清单。还提供了商业支持和培训 [<http://www.hibernate.org/113.html>]。另外，Gavin King和Christian Bauer最近合著了Hibernate in Action [<http://www.hibernate.org/148.html>]。我推荐使用这本书来学习Hibernate。

iBATIS

iBATIS SQL Maps是一种开源的持久性框架，允许你对一种关系数据库使用模型对象进行操作。与Hibernate不同的是，你使用SQL，这样你可以更好利用JDBC。你可以在一个简单的XML文件中完成此操作，从Java类中制取SQL。iBATIS并不是一个O/R Mapper (ORM)，而是一种Data Mapper。在Martin's Fowler的Patterns of Enterprise Application Architecture [<http://www.martinfowler.com/books.html#eaa>]一书中，他描述了两种模式：Data Mapper

[<http://www.martinfowler.com/eaCatalog/dataMapper.html>]和Metadata Mapping [<http://www.martinfowler.com/eaCatalog/metadataMapping.html>]。区别在于，ORM(Metadata Mappers)将类映射到数据库的表，iBATIS (Data Mapper)将输入和输出映射到一个接口(例如，一种RDBMS的SQL接口)。当你拥有数据库的控制权时，ORM方案表现得不错。当数据库过于正统，你需要将多个表中的数据制取出来填充到一个对象时，像iBATIS这样的Data Mapper就有用武之地。

iBATIS是在2001年由Clinton Begin发起的一个开源项目的名称。Clinton有好几个产品，但几乎都没有引起人们的关注，直到.NET PetStore发布了，其声称.NET在开发效率和产品性能上都要优于Java。Microsoft发布了白皮书，声称Sun的PetStore的.NET版本要快10倍，开发效率也要高出4倍。得知这并非事实，2002年7月，Clinton以JPetStore 1.0 [<http://www.ibatis.com/jpetstore/jpetstore-1.html>]进行了回应。和.NET版相比，他完成这个程序花了更少代码，并拥有更好的设计，而且他是在几个星期的空余时间实现的。

Clinton写JPetStore的目的是为了从几点上进行驳斥：1)良好的设计。2)代码质量。3)开发效率。原来的.NET Petstore的设计糟透了，业务逻辑中包含大量的存储过程，相反JPetstore拥有一个清晰高效的框架。

这个框架很快吸引了开源社区的注意。今天，iBATIS，指的是“iBATIS Database Layer”，它由一个DAO框架和一个SQL Map框架组成。

Spring支持iBATIS SQL Map，提供了帮助类可以简单配置和使用它们。另外，Spring还包含了JPetstore作为它的一个样例程序，很多细节用Spring的特性重写了。

就我看来，iBATIS在开源社区并不出名。并没有多少人听说过它，但只要有所耳闻的人，都会爱上它。也许Spring的支持类能够提高它的知名度。

iBATIS使用的协议是Apache [<http://www.opensource.org/licenses/apache2.0.php>]，这就意味着，只要你在你的最终用户文档中使用指明你的产品包含由Apache软件基金会开发的软件，就可以自由的使用它。你可以修改代码，但是在没有授权的情况，你不能以Apache我名义重新对它进行发布。

注意

iBATIS最近在申请成为Apache的一个顶级项目。如果被接受的话，有可能被重新命名为iBATIS Data Mapper。

iBATIS是一种优秀的持久性框架，能够使用现在的或遗留的数据库系统。从一个基于JDBC的应用程序迁移到iBATIS非常容易(大部分工作是从Java类中提取SQL，保存在Java文件中)。iBATIS不但快速和高效，而且隐藏了SQL，这种迄今最为强大最古老的语言。利用iBATIS的SQL Map，开发人员可以将SQL写进XML文件，基于它们的查询结果来填充对象。和Spring/Hibernate组合一样，iBATIS DAO在每个方法中只需要为数不多的几行代码。

以我的经验，我发觉iBATIS真的有以下品质。

- 易学
- 查询效率高
- 由于有现成的SQL，容易变换。
- 和Hibernate相比一样的快速(如果不是更快的话)
- 写iBATIS DAO和Hibernate DAO差不多

注意

i B A T I S 的 S q l M a p C l i e n t
[\[http://www.ibatis.com/common/javadocs/com/ibatis/sqlmap/client/SqlMapClient.html\]](http://www.ibatis.com/common/javadocs/com/ibatis/sqlmap/client/SqlMapClient.html)与Hibernate的Session和JDBC的Connection。Spring的iBATIS支持类位于包org.springframework.orm.ibatis
[\[http://www.springframework.org/docs/api/org/springframework/orm/ibatis/package-summary.html\]](http://www.springframework.org/docs/api/org/springframework/orm/ibatis/package-summary.html)和org.springframework.orm.ibatis.support
[\[http://www.springframework.org/docs/api/org/springframework/orm/ibatis/support/package-summary.html\]](http://www.springframework.org/docs/api/org/springframework/orm/ibatis/support/package-summary.html)。

依赖包

对J2SE1.4, iBATIS仅需要一个第三方依赖包: Commons Logging。对于J2SE1.3, 请参阅下载文件[\[http://www.ibatis.com/common/download.html\]](http://www.ibatis.com/common/download.html)中的jar-dependencies.txt文件。下面的jar文件作为iBATIS2.0.5一部分包含在下载MyUsers程序。Spring支持iBATIS 1.x和2.x, 主要区别在2.x中类的“Client”这个词上。本节仅涉及2.x。

- ibatis-common-2.jar:公共类。
- ibatis-sqlmap-2.jar:SQL Maps必须。
- commons-logging.jar:日志框架, 在不同logging类库之间提供一个超薄的桥。
- (可选)cglib-full-2.0.2.jar:代码生成库, 能够在优化JavaBean属性访问和加强延迟导入方面提高运行时二进制代码性能。

配置

要集成iBATIS, 你必须为用户对象创建一个SQL Map。一个SQL Map是一个XML文件, 包含将查询的输入输出结果映射到对象的SQL语句。为了避免各bean的名称冲突, 将文件applicationContext-hibernate.xml(在web/WEB-INF)重命名为applicationContext-hibernate.xml.txt。这是为了防止Spring加载它。

1. 在目录src/org/appfuse/model中, 创建一个名为UserSQL.xml的文件, 内容如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap PUBLIC "-//ibatis.com//DTD SQL Map 2.0//EN"
"http://www.ibatis.com/dtd/sql-map-2.dtd">
<sqlMap namespace="UserSQL">
  <insert id="addUser" parameterClass="org.appfuse.model.User">
    insert into app_user (id, first_name, last_name)
    values (#id#, #firstName#, #lastName#);
    <selectKey resultClass="java.lang.Long" keyProperty="id" >
      select last_insert_id();
    </selectKey>
  </insert>
  <update id="updateUser" parameterClass="org.appfuse.model.User">
    update app_user set first_name = #firstName#,
    last_name = #lastName#
    where id = #id#;
  </update>
  <select id="getUser" parameterClass="java.lang.Long"
    resultClass="org.appfuse.model.User">
```

```

    select id, first_name as firstName, last_name as lastName
    from app_user where id=#id#;
</select>
<select id="getUsers" resultClass="org.appfuse.model.User">
    select id, first_name as firstName, last_name as lastName
    from app_user;
</select>
<delete id="deleteUser" parameterClass="java.lang.Long">
    delete from app_user where id = #id#;
</delete>
</sqlMap>

```

在这个文件中，你可以看到使用各种不同的元素(<insert>, <update>, <select>, <delete>)来标示数据库操作。注意，每个元素可以指定一个可选的parameterClass或returnClass属性。动态变量封装在#value#中，并在parameterClass中指明属性。

2. 在目录src/org/appfuse/dao/ibatis(你必须创建这个目录)创建一个文件sql-maps-config.xml，用来指明UserSQL.xmlSQL Map的位置。

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMapConfig PUBLIC
    "-//iBATIS.com//DTD SQL Map Config 2.0//EN"
    "http://www.ibatis.com/dtd/sql-map-config-2.dtd">
<sqlMapConfig>
    <settings enhancementEnabled="true" maxTransactions="5"
        maxRequests="32" maxSessions="10"/>
    <sqlMap resource="org/appfuse/dao/ibatis/UserSQL.xml" />
</sqlMapConfig>

```

更多有关<settings>的信息，请参阅iBATIS文档[<http://umn.dl.sourceforge.net/sourceforge/ibatisdb/iBATIS-SqlMaps-2.pdf>](PDF)。这里演示的属性值对大多数应用程序来说已经足够了。

3. 在目录web/WEB-INF创建一个文件applicationContext-ibatis.xml(将applicationContext-empty.txt作为一个模板使用)，添加一个javax.sql.DataSource的bean定义。本例中的dataSource bean使用你在上一节配置的MySQL数据库。

```

<?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.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName">
            <value>com.mysql.jdbc.Driver</value>
        </property>
        <property name="url">
            <value>jdbc:mysql://localhost/myusers</value>
        </property>
    </bean>

```

```

<property name="username"><value>root</value></property>
<property name="password"><value></value></property>
</bean>
<!-- Add bean definitions here -->
</beans>

```

4. 创建一个sqlMapClient bean定义用来集成iBATIS和Spring。

```

<bean id="sqlMapClient"
class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
<property name="configLocation">
<value>classpath:/org/appfuse/dao/ibatis/sql-map-config.xml
</value>
</property>
<property name="dataSource"><ref bean="dataSource"/></property>
</bean>

```

5. 添加一个transactionManager定义, 引用 `DataSourceTransactionManager` [<http://www.springframework.org/docs/api/org.springframework.jdbc.datasource.DataSourceTransactionManager.html>], `PlatformTransactionManager` [<http://www.springframework.org/docs/api/org.springframework.transaction.PlatformTransactionManager.html>] 实现类将指定一个JDBC 连接从指定数据库绑定到线程上, 这就意味着每个datasource可以使用一个独立的线程连接。

注意

你需要一个名为transactionManager的bean的主要原因是, userManager bean引用了一个同名的bean。

```

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

```

6. 在目录src/org/appfuse/dao/ibatis创建一个UserDAOiBatis.java文件, 这个类继承了 `SqlMapClientDaoSupport` [<http://www.springframework.org/docs/api/org.springframework.orm.ibatis.support.SqlMapClientDaoSupport.html>] 实现了UserDAO接口。

```

package org.appfuse.dao.ibatis;
// use your IDE to organize imports

public class UserDAOiBatis extends SqlMapClientDaoSupport
implements UserDAO {

public List getUsers() {
return getSqlMapClientTemplate().queryForList("getUsers", null);
}
}

```

```

    }

    public User getUser(Long id) {
        User user = (User) getSqlMapClientTemplate().queryForObject("getUser", id);
        if (user == null) {
            throw new ObjectRetrievalFailureException(User.class, id);
        }
        return user;
    }

    public void saveUser(User user) {
        if (user.getId() == null) {
            // Use iBatis's <selectKey> feature, which is db-specific
            Long id = (Long) getSqlMapClientTemplate()
                .insert("addUser", user);
            user.setId(id);
            logger.info("new User id set to: " + id);
        } else {
            getSqlMapClientTemplate().update("updateUser", user);
        }
    }

    public void removeUser(Long id) {
        getSqlMapClientTemplate().update("deleteUser", id);
    }
}

```

7. 在文件`applicationContext-ibatis.xml`中添加一个`userDAO` bean定义。下面的例子使用了`autowire`，而没有指定`sqlMapClient`属性。使用`autowire`时，在注释中指定注射的依赖是不错的主意。

```

<bean id="userDAO" autowire="byName"
    class="org.appfuse.dao.ibatis.UserDAOiBatis"/>
<!-- injected property: sqlMapClient -->

```

注意

在Spring 1.0.2以前版本，从 `SqlMapClientDaoSupport` [<http://www.springframework.org/docs/api/org.springframework.orm.ibatis.support.SqlMapClientDaoSupport.html>] 继承的bean需要在它们的bean定义中设置“`dataSource`”属性。在1.0.2版本中，“`dataSource`”被添加到 `SqlMapClientFactoryBean` [<http://www.springframework.org/docs/api/org.springframework.orm.ibatis.SqlMapClientFactoryBean.html>]，作为每个DAO `DataSource`引用的一种备选方案。

8. 由Hibernate创建的`app_user`表不允许`id`列为`null`。使用iBATIS，生成主键最简单的方法是插入`null`，然后从数据库中返回生成的`id`。为了达到上述目的，删除并重新创建`app_usr`表。
1. 从命令行键入`mysql -u root -p myusers`登录MySQL。
 2. 执行`drop table app_user`。

3. 执行以下SQL语句。

```
create table app_user (id bigint not null auto_increment,  
first_name varchar(50), last_name varchar(50),  
primary key (id));
```

测试

如果你是从前面的应用中开发MyUsers，在测试通过之前，你可以不得不修改build.xml。compile target包含以下XML代码，用于将Hibernate mapping文件复制到build目录中。

```
<!-- Copy hibernate mapping files to ${build.dir}/classes -->  
<copy todir="${build.dir}/classes">  
  <fileset dir="${src.dir}" includes="**/*.hbm.xml"/>  
</copy>
```

将上面的代码修改成：

```
<!-- Copy XML files to ${build.dir}/classes -->  
<copy todir="${build.dir}/classes">  
  <fileset dir="${src.dir}" includes="**/*.xml"/>  
</copy>
```

运行ant clean test -Dtestcase=UserDAO。这次测试输出结果应该和UserDAOHibernate类差不多。

缓存

iBATIS支持为SQLMap提供了多种缓存策略。要在UserSQL.xml添加一种缓存策略，添加下面的<cacheModel>元素。

```
<sqlMap namespace="UserSQL">  
  <cacheModel id="userCache" type="LRU">  
    <flushInterval hours="24"/>  
    <property name="size" value="1000"/>  
  </cacheModel>  
<insert id="addUser" parameterClass="org.appfuse.model.User">
```

添加一个cacheModel属性到任何<select>语句中。例如，添加到“getUser”语句中。

```
<select id="getUser" parameterClass="java.lang.Long"  
  resultClass="org.appfuse.model.User"  
  cacheModel="userCache">  
  select id, first_name as firstName, last_name as lastName
```

```
from app_user where id=#id#;
</select>
```

获取生成的主键

`addUser`语句使用数据库特性的途径来获取生成的主键。你可以用`<selectKey>`轻而易举的获取生成的主键。

```
<insert id="addUser" parameterClass="org.appfuse.model.User">
  insert into app_user (id, first_name, last_name)
  values (#id#, #firstName#, #lastName#);
  <selectKey resultClass="java.lang.Long" keyProperty="id" >
    select last_insert_id();
  </selectKey>
</insert>
```

这种方法的问题在于利用的是数据库的特性。`select last_insert_id()`局限于MySQL数据库。这也说明了SQL允许改动，而不是真正的标准。一种良好的ORM工具，如Hibernate和JDO实现方式对你来说应该是透明的。

注意

如果你想这个问题切换到回HSQL，只要将`<selectKey>`语句修改成`call identity()`。还要修改相应的`dataSource` bean。运行`ant browse`，删除/重新创建`app_user`表。下面的SQL脚本能帮你完成任务。

```
create table app_user (id integer identity, first_name varchar(50),
last_name varchar(50));
```

作为候选方案，你可以使用Spring的类生成主键。例如，使用 `MySQLMaxValueIncrementer` [<http://www.springframework.org/docs/api/org.springframework.jdbc.support.incrementer.MySQLMaxValueIncrementer.html>]，执行以下步骤。

1. 注释掉`UserSQL.xml`文件的`<selectKey>`。
2. 将`save()`修改成如下。

```
public void saveUser(User user) {
  if (user.getId() == null) {
    MySQLMaxValueIncrementer incrementer =
      new MySQLMaxValueIncrementer(
        getDataSource(), "user_sequence", "value");
    Long id = new Long(incrementer.nextLongValue());
    user.setId(id);
    getSqlMapClientTemplate().insert("addUser", user);
    logger.info("new User id set to: " + id);
  } else {
    getSqlMapClientTemplate().update("updateUser", user);
  }
}
```



```
}
```

创建user_sequence来获取主键。

```
create table user_sequence (value bigint auto_increment,
primary key (value));
insert into user_sequence values(0);
```

如果你清除和运行UserDAOTest，所有的测试都会通过。

现在的问题是，在你的类中放入MySQLMaxValueIncrementer，那么硬编码产生的DAO会依赖MySQL。使用select last_insert_id()语句看起来更具有可配置性，因为它是在XML文件中。重构这个类，使用依赖注射(dependency injection)。

1. 在UserDAOHibernate类加入incrementer和setter方法。

```
private DataFieldMaxValueIncrementer incrementer;

public void setIncrementer(DataFieldMaxValueIncrementer incrementer) {
    this.incrementer = incrementer;
}
```

2. 修改saveUser()方法，使用这个incrementer。只需要删除初始化incrementer的三行代码。
3. 在文件web/WEB-INF/applicationContext-ibatis.xml配置incrementer bean。因为autoDAO是通过名称绑定的，所以此属性由Spring注入。

```
<bean id="incrementer"
class="org.springframework.jdbc.support.incrementer.MySQLMaxValue
Incrementer">
<property name="dataSource"><ref local="dataSource"/></property>
<property name="incrementerName">
<value>user_sequence</value>
</property>
<property name="columnName"><value>value</value></property>
</bean>
```

4. 运行ant test -Dtestcase=UserDAO。

不幸的是，iBATIS并不支持JDBC3.0的getGeneratedKeys()方法，所以你必须用上述方法之一来生成主键。当然你也可以使用select max(id)语句，自己增加主键。

社区与支持

使用 i B A T I S 的开发人员都乐在其中。它带言简意赅的文档和教程 [<http://www.ibatis.com/common/download.html>]。这份文档只有53页，教程仅有9页。以我的经验，这是一种能在同一天学以致用的框架。i B A T I S 没有邮件列表，但是它有帮助论坛 [http://www.sourceforge.net/forum/?group_id=61326]。它的应用程序在Apache上成为一个顶级项目，也是另一个受欢迎的佐证。

Spring JDBC

如果你以前写过JDBC程序，你应该深知它是多么枯燥无味。你不仅要设置Connections, Statements 和 ResultSets，在你取得数据后你还要关闭它们。在JDBC代码中关闭资源往往是新手的盲区。他们常常忘记使用 `finally` 块，那怕它是最基本的JDBC编码模式 [<http://www.codenotes.com/articles/articleAction.aspx?articleID=84>]。

正如你前面看到的那样，Spring免去了开发人员打开/关闭资源的义务。它帮你管理这些操作，以便你能集中精力编写应用代码，而不是基础代码。Spring的JDBC核心是建立在J2SE的JDBC上的一层抽象，这样你可以写少量的代码对你的数据进行读取，保存和删除操作。

JDBC抽象层框架由4个包组成：core, datasource, object和support。每个包的功能如下：

- `org.springframework.jdbc.core`
[<http://www.springframework.org/docs/api/org/springframework/jdbc/core/package-summary.html>]: 核心类，JdbcTemplate和JdbcDaoSupport常用类。
- `org.springframework.jdbc.datasource`
[<http://www.springframework.org/docs/api/org/springframework/jdbc/datasource/package-summary.html>]: 提供简易的DataSource访问及基本的DataSource实现的类。
- `org.springframework.jdbc.object`
[<http://www.springframework.org/docs/api/org/springframework/jdbc/object/package-summary.html>]: 用于将数据库查询，更新，存储过程映射为线程安全可复用的对象的类。
- `org.springframework.jdbc.support`
[<http://www.springframework.org/docs/api/org/springframework/jdbc/support/package-summary.html>]: SQLException转换，DataFieldMaxValueIncrementer实现，jdbc.core和jdbc.object的支持类。

在本节中，你将用到 `JdbcTemplate` [<http://www.springframework.org/docs/api/org/springframework/jdbc/core/JdbcTemplate.html>] 和 `JdbcDaoSupport` [<http://www.springframework.org/docs/api/org/springframework/jdbc/core/support/JdbcDaoSupport.html>]。命名与你在Hibernate和iBATIS使用的Template和DaoSupport类类似。本节不会提及SQLException转换，因为你几乎用不到它。这些类是在内部使用的，很多默认的转换对以下的数据库是内建的：DB2, HSQL, SQL Server, Oracle, MySQL, Infomix, PostgreSQL和Sybase。你可以在文件 `jdbc/support/sql-error-codes.xml` [http://sourceforge.net/viewsvn/*checkout*/springframework/spring-3.0.0-RC1/springframework-jdbc/support/sql-error-codes.xml?view=svn&path=/springframework-jdbc/support/sql-error-codes.xml] 中找到异常对应的错误代码的一份完整的清单。

将ResultSet映射到对象

和Hibernate和iBATIS类似，Spring要求将查询结果映射到你的POJO。使用Hibernate和iBATIS时，你可以用XML来实现。使用Spring时，你不得不用程序实现。你没有必要将结果映射到对象，你

可以简单的以maps形式返回你的数据。例如，下面的代码会返回一个map，其中每个条目(entry)是一行记录。

```
public List getList() {
    return getJdbcTemplate().queryForList("select * from tablename");
}
```

然而，在大多数情况下，你想将结果映射到一个由对象(object)组成的List中。使用Spring JDBC实现时，你可以用 `MappingSqlQuery` [<http://www.springframework.org/docs/api/org.springframework.jdbc.object.MappingSqlQuery.html>]类将每一行JDBC ResultSet转换成一个对象。下面是这种类型的类的一个范例，返回 MyUsers应用程序中用户的清单。

```
public class UsersQuery extends MappingSqlQuery {
    public UsersQuery(DataSource ds) {
        super(ds, "SELECT * FROM app_user");
        compile();
    }

    protected Object mapRow(ResultSet rs, int rowNum)
        throws SQLException {
        User user = new User();
        user.setId(new Long(rs.getLong("id")));
        user.setFirstName(rs.getString("first_name"));
        user.setLastName(rs.getString("last_name"));
        return user;
    }
}
```

继承时MappingSqlQuery，你必须复写mapRow(ResultSet rs, int rowNum)方法。使用UsersQuery会得到一个由User对象组成的list。

```
public List getUsers() {
    return new UsersQuery(getDataSource()).execute();
}
```

使用MappingSqlQuery的declareParameter()方法，可以在你的查询上声明参数。

```
public UserQuery(DataSource ds) {
    super(ds, "SELECT * FROM app_user WHERE id = ?");
    super.declareParameter(new SqlParameter("id", Types.INTEGER));
    compile();
}
```

使用一个Object数据传入id参数。

```
public User getUser(Long id) {
    List users = new UserQuery(getDataSource())
        .execute(new Object[]{id});
    if (users.isEmpty()) {
        throw new ObjectRetrievalFailureException(User.class, id);
    }
    return (User) users.get(0);
}
```

执行更新查询

将ResultSet映射到对象能帮你获取数据，但不能帮你更新数据。对于这个，可以且使用SqlUpdate类针对更新记录创建一个可复用的对象。这里有一个例子。

```
public class UserUpdate extends SqlUpdate {
    public UserUpdate(DataSource ds) {
        super(ds, "INSERT INTO app_user (id, first_name, last_name) values
            (?, ?, ?)");
        declareParameter(new SqlParameter("id", Types.BIGINT));
        declareParameter(new SqlParameter("first_name", Types.VARCHAR));
        declareParameter(new SqlParameter("last_name", Types.VARCHAR));
        compile();
    }
}
```

调用这个类的代码样例。

```
Object[] params =new Object[] {user.getId(), user.getFirstName(),
    user.getLastName()};
new UserUpdate(getDataSource()).update(params);
```

另外，有一种更省事的方法是使用JdbcTemplate的update()方法。

```
getJdbcTemplate().update(
    "UPDATE app_user SET first_name = ?, last_name = ? WHERE id = ?",
    new Object[] {user.getFirstName(), user.getLastName(), user.getId()});
```

另外，你没有必须实现一个完整的SqlUpdate类。只需要在一个方法内声明一个内嵌的实例。

```
String sql = "INSERT INTO app_user (id, first_name, last_name) ";
sql += "values (?, ?, ?)";
SqlUpdate su = new SqlUpdate(getDataSource(), sql);
su.declareParameter(new SqlParameter("id", Types.BIGINT));
su.declareParameter(new SqlParameter("first_name", Types.VARCHAR));
su.declareParameter(new SqlParameter("last_name", Types.VARCHAR));
```

```
su.compile();
Object[] params = new Object[]
{user.getId(), user.getFirstName(), user.getLastName()};
su.update(params);
```

获取生成的主键

JDBC 3.0，作为J2SE 1.4的一部分，规定遵从JDBC 3.0的驱动必须实现 `java.sql.Statement.getGeneratedKeys()` [[http://java.sun.com/j2se/1.4.2/docs/api/java/sql/Statement.html#getGeneratedKeys\(\)](http://java.sun.com/j2se/1.4.2/docs/api/java/sql/Statement.html#getGeneratedKeys())]方法。这个方法会从数据库中获取生成主键(例如，在MySQL中调用 `select last_insert_id()`)。MyUsers中的MySQL驱动是遵从JDBC 3.0规范的。

为了利用Spring JDBC生成的主键，使用 `KeyHolder` [<http://www.springframework.org/docs/api/org.springframework.jdbc.support.KeyHolder.html>]作为 `SqlUpdate.execute()` [[http://www.springframework.org/docs/api/org.springframework.jdbc.support.SqlUpdate#execute\(java.lang.Object\[\]\)](http://www.springframework.org/docs/api/org.springframework.jdbc.support.SqlUpdate#execute(java.lang.Object[]))]的第2参数。

```
KeyHolder keys = new GeneratedKeyHolder();
su.update(params, keys);
user.setId(new Long(keys.getKey().longValue()));
```

对于不符规范的驱动可以使用前面介绍过的 `DataFieldMaxValueIncrementer` [<http://www.springframework.org/docs/api/org.springframework.jdbc.support.incrementer/DataFieldMaxValueIncrementer.html>]策略。

依赖包

因为Spring JDBC是Spring的一部分，所以不需要其它类库一样的第三方包。如果你仅想使用的Spring的JDBC功能，可以在你的应用程序中使用 `spring-dao.jar` (而不需要囊括一切的 `spring.jar`)。

配置

Spring JDBC无需映射文件或sql map，所以在MySQL应用程序中配置非常容易。

1. 在目录 `web/WEB-INF` 中创建一个文件 `applicationContext-jdbc.xml`，添加一个 `dataSource` bean 定义。请确保已经将前面的 `applicationContext-ibatis.xml` 重命名为 `applicationContext-ibatis.xml.txt`。

注意

你可以复制一份这个文件的iBATIS版本，删除其中的 `sqlMapClient` bean 定义。

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName">
    <value>com.mysql.jdbc.Driver</value>
  </property>
  <property name="url">
```

```

    <value>jdbc:mysql://localhost/myusers</value>
  </property>
  <property name="username"><value>root</value></property>
  <property name="password"><value></value></property>
</bean>

```

2. 添加transaction manager。

```

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

```

3. 在目录src/org/appfuse/dao/jdbc创建文件UserDAOJdbc.java。这个类继承了JdbcDaoSupport，并实现了UserDAO接口。
<http://www.springframework.org/docs/api/org.springframework.jdbc.core.support.JdbcDaoSupport.html>，

```

package org.appfuse.dao.jdbc;
// use your IDE to organize imports

public class UserDAOJdbc extends JdbcDaoSupport implements UserDAO {

    public List getUsers() {
        return new UsersQuery(getDataSource()).execute();
    }

    public User getUser(Long id) {
        List users = new UserQuery(getDataSource())
            .execute(new Object[]{id});
        if (users.isEmpty()) {
            throw new ObjectRetrievalFailureException(User.class, id);
        }
        return (User) users.get(0);
    }

    public void saveUser(User user) {
        if (user.getId() == null) {
            String sql = "INSERT INTO app_user (id, first_name, ";
            sql += " last_name) values (?, ?, ?)";
            SqlUpdate su = new SqlUpdate(getDataSource(), sql);
            su.declareParameter(new SqlParameter("id", Types.BIGINT));
            su.declareParameter(new SqlParameter("first_name", Types.VARCHAR));
            su.declareParameter(new SqlParameter("last_name", Types.VARCHAR));
            su.compile();
            Object[] params = new Object[] {
                user.getId(), user.getFirstName(),
                user.getLastName()};

```

```
KeyHolder keys = new GeneratedKeyHolder();
su.update(params, keys);
user.setId(new Long(keys.getKey().longValue()));
if (logger.isDebugEnabled()) {
    logger.info("user's id is: " + user.getId());
}
} else {
    getJdbcTemplate().update("UPDATE app_user SET first_name = ?,
    last_name = ? WHERE id = ?",
    new Object[] {user.getFirstName(), user.getLastName(),
    user.getId()});
}
}

public void removeUser(Long id) {
    getJdbcTemplate().update("DELETE FROM app_user WHERE id = ?",
    new Object[] {id});
}

// Query to get a single User
class UserQuery extends MappingSqlQuery {
    public UserQuery(DataSource ds) {
        super(ds, "SELECT * FROM app_user WHERE id = ?");
        super.declareParameter(new SqlParameter("id", Types.INTEGER));
        compile();
    }

    protected Object mapRow(ResultSet rs, int rowNum)
    throws SQLException {
        User user = new User();
        user.setId(new Long(rs.getLong("id")));
        user.setFirstName(rs.getString("first_name"));
        user.setLastName(rs.getString("last_name"));
        return user;
    }
}

// Query to get a list of User objects
class UsersQuery extends MappingSqlQuery {
    public UsersQuery(DataSource ds) {
        super(ds, "SELECT * FROM app_user");
        compile();
    }

    protected Object mapRow(ResultSet rs, int rowNum)
    throws SQLException {
        User user = new User();
        user.setId(new Long(rs.getLong("id")));
        user.setFirstName(rs.getString("first_name"));
        user.setLastName(rs.getString("last_name"));
        return user;
    }
}
```

```
}
```

这个类比前面的两种方案多出了很多代码。但它不需要任何配置文件。`UserQuery`和`UsersQuery`都是内部类，但很容易重构成独立的java文件。

4. 在目录web/WEB-INF下的`applicationContext-jdbc.xml`中添加一个`userDAO`bean。

测试

为了保证一个干净的数据库，删除并重新创建`app_user`表。

1. 从命令行键入`mysql -u root -p myusers`登录MySQL。
2. 执行`drop table app_user`。
3. 执行以下SQL语句。

```
create table app_user (id bigint not null auto_increment,
first_name varchar(50), last_name varchar(50),
primary key (id));
```

4. 运行`ant test -Dtestcase=UserDAO`。

社区和支持

SpringJDBC上建立在JDBC上一个抽象层。它让你从频繁的打开和关闭资源操作中解放出来，对于大量数据的批处理也表现不错。如果你想深入研究它，Spring的参考文档 [<http://www.springframework.org/docs/reference/jdbc.html>]是不可多得资料，如果你有疑问，Spring的Data Access Support Forum [<http://forum.springframework.org/viewforum.php?f=5>]也是个不错的去处。

Interface21 [<http://www.springframework.com/company>]为Spring和它的JDBC框架提供了商业支持。

JDO

JDO是种Java标准，这就是意味着它是作为JCP的一部分而开发的。作为一种标准，它并不是一种实际的产品，而是一种指导产品如何构建的规范。JDO的目标在JDO 2.0规范 [<http://www.jcp.org/aboutjava/communityprocess/final/jsr012/index2.html>]的1.1节中作了最好的描述。

JDO架构有两个主要目标：首先，为应用程序员提供一个透明的以Java为中心的持久数据的视图，包括企业数据和本地存储的数据。第二，能为数据存储进应用服务器提供一个可拔插的实现。

本节介绍如何用Spring来配置和使用你的JDO实现，它可能来自应用服务器提供商，或是由开源组织提供，如JPOX [<http://www.jpox.org/index.jsp>]和ObjectWeb的Speedo [<http://speedo.objectweb.org/>]。这些例子中采用了JPOX，因为它是JDO 2.0的开源参考实现。

注意

JDO的支持类位于包 `org.springframework.orm.jdo` [<http://www.springframework.org/docs/api/org/springframework/orm/jdo/package-summary.html>]和

`org.springframework.orm.jdo.support`
<http://www.springframework.org/docs/api/org.springframework.orm.jdo.support/package-summary.html>
 中。

JDO和主键

JDO处理主键(也叫object ids)的方式与其它持久性策略有很大的区别。这是因为在设计它时同时兼顾了对象数据库(object databases)和关系数据库(relational databases)。在大多数情况下, 你用到的还是关系数据库, 所以你应该用application identity。然而, 如果你在制作一个应用的原型, 使用datastore 可能会简单一些, 你不用关心主键。下面是规范中各种不同的identity类型的一份清单。

表 7.1. JDO Identity Types

Identity Type	JDO操纵的Schema	现有的Schema
datastore	#1生成表; 添加一个主键, 定义主键值	#2允许配置类映射到现有的表, 并定义主键列名
application	#3生成表, 允许你为主键定义类	#4映射到现有表, 允许你为主键定义类

datastore identity类型的目的是为了JDO完全处理主键。例如, 要使用类型#1的User对象, 删除id字段及其访问方法, 因为JDO用不到它。这样就在数据库中, 创建了特有的列(user_id)以及特有的表user。假定在你的操作过程中你并不知道主键。

对于类型#2, 你可以定义一个与User对象id属性相匹配的列名。但是在你对这个对象进行持久性操作时, 数据库无法对它进行填充。换句话说, 它没有告诉你生成的主键是什么。

如果你想知道主键, 你必须使用application identity类型。使用这种方案, 你仍然需要生成表(类型#3), 或者你可以用MetaData(实际上是一个与Hibernate中类似的映射文件)指定一个表名(类型#4)。使用application identity, 你必须指定一个主键类。在MyUsers应用中, UserDaoTest的testSaveUser()方法会验证分配的主键。

```
public void testSaveUser() throws Exception {
    user = new User();
    user.setFirstName("Rod");
    user.setLastName("Johnson");
    dao.saveUser(user);
    assertTrue("primary key assigned", user.getId() != null);
    log.info(user);
    assertTrue(user.getFirstName() != null);
}
```

基于这些原因, UserDao的JDO实现将采用application identity。

依赖包

本例中使用了JPOX的1.1.0-alpha-2版, 它是JDO2.0的一个实现。下面的jar文件作为JPOX一部分包含在MyUsers下载文件中。

- bcel-5.1.jar:Byte-Code Engineering Library增强二进制class文件的执行效率

- jpox-enhancer-1.1.0-alpha-2.jar:JPOX二进制提升器，在运行一个使用了JPOX的JDO的应用程序之前，提升类的执行效率
- jpox-1.1.0-alpha-2.jar:核心JPOX类

配置

和其它框架相比，JDO设置要繁琐一些。在构建过程中你必须添加一个额外的步骤，来提升你的可持续(persistable)的对象。为了避免和前面一节iBATIS产生冲突，将文件applicationContext-jdbc.xml(在目录web/WEB-INF)重命名为applicationContextjdbc.xml.txt。这样能够防止它被加载。

注意

本例中JDO实现只是JDO2.0的一个alpha版本。JDO规范已经提供了一个早期的草稿版，很快会得到一个社区复审结果。之后，很多厂商会提供JDO2.0实现方案。这里描述的提升的步骤很可能被cglib [<http://cglib.sourceforge.net/>]类似的二进制增强方案取代。

1. 打开MySQL中的build.xml文件，紧挨着test target之前添加以下enhance target。

```
<target name="enhance" description="JPOX enhancement"
  depends="compile">
  <taskdef name="jpoxenhancer"
    classname="org.jpox.enhancer.tools.EnhancerTask"
    classpathref="classpath"/>
  <jpoxenhancer dir="${build.dir}/classes" failonerror="true"
    fork="true" verbose="true" check="true">
    <classpath>
      <path refid="classpath"/>
      <path location="${build.dir}/classes"/>
      <path location="web/WEB-INF/classes"/>
    </classpath>
  </jpoxenhancer>
</target>
```

修改“test”和“deploy” target，添加依赖“enhance”。

```
<target name="test" depends="enhance"
  description="Runs JUnit tests">
  ...
<target name="war" depends="enhance"
  description="Packages app as WAR">
```

2. 创建一个MetaData(元数据)文件，将User对象映射到app_user表。在org/appfuse/model下，创建一个名为package.jdo文件，内容如下。

```
<?xml version="1.0"?>
<!DOCTYPE jdo PUBLIC
"-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
```

```

"http://java.sun.com/dtd/jdo_2_0.dtd">
<jdo>
  <package name="org.appfuse.model">
    <class name="User" identity-type="application"
      objectid-class="org.appfuse.model.PrimaryKey"
      table="app_user">
      <field name="id" primary-key="true"/>
      <field name="firstName" persistence-modifier="persistent">
        <column name="first_name" jdbc-type="VARCHAR" length="50"/>
      </field>
      <field name="lastName" persistence-modifier="persistent">
        <column name="last_name" jdbc-type="VARCHAR" length="50"/>
      </field>
      <extension vendor-name="jpox" key="use-poid-generator"
        value="true"/>
    </class>
  </package>
</jdo>

```

这个类中的大部分条目是一目了然的，你可以看到在何处设置表名，列名，大小，以及User对象中每个字段是如何定义成一系列的。注意，`package.jdo`可以存放包`org.appfuse.model`(由`<package>`定义)中所有类的元数据。你还要注意接近末尾如下所示的几行。

```
<extension vendor-name="jpox" key="use-poid-generator" value="true"/>
```

这是一个利用sequence表来生成主键的标志。有好几种方案可以控制如何生成主键。更多信息，请参考JPOX官方文档 [http://www.jpox.org/docs/1_1/identity_generation.html]。

注意

当JPOX完全实现JDO2.0时，此extension很有可能被替换掉。

3. 为了复制文件`package.jdo`并将它打进war包，修改`build.xml`文件中的`compile target`，将`*.jdo`文件包含进去。

```

<!-- Copy XML files to ${build.dir}/classes -->
<copy todir="${build.dir}/classes">
  <fileset dir="${src.dir}" includes="**/*.xml"/>
  <!-- Copy JDO mapping files -->
  <fileset dir="${src.dir}" includes="**/*.jdo"/>
</copy>

```

4. `package.jdo`文件中的`<class>`有一个`objectid-class`属性。这里指定了一个主键类，在使用`application` `identity`时必需的。在目录`src/org/appfuse/model`下创建`PrimaryKey.java`类，填充以下代码。

```
package org.appfuse.model;
```

```
// organize imports with your IDE

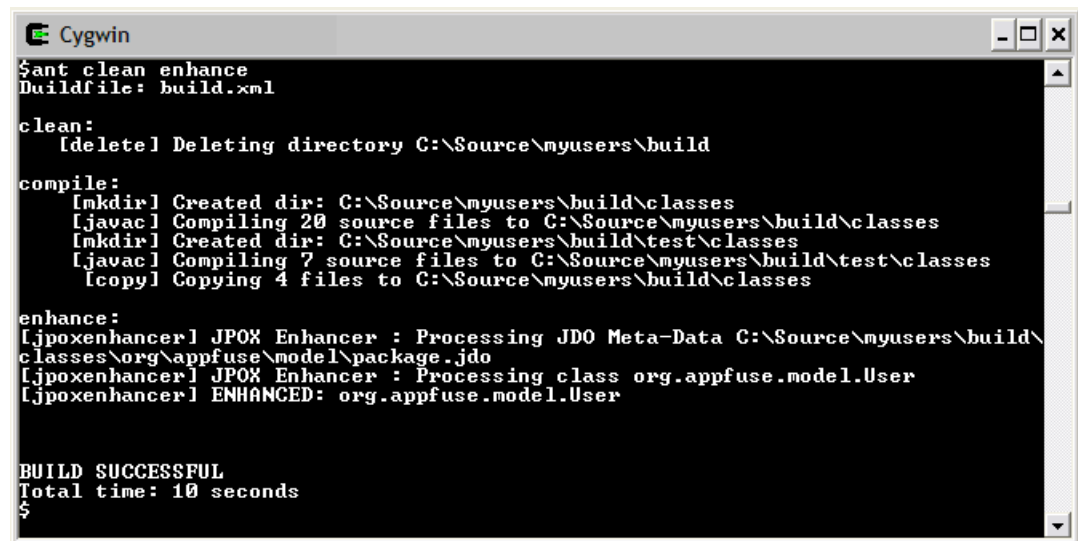
public class PrimaryKey extends BaseObject implements Serializable {
    public Long id;
    public PrimaryKey() {}
    public PrimaryKey(String value) {
        id = Long.valueOf(value);
    }
}
```

按照JDO规范定义，一个主键类应该有很多必要条件，如下所示。

- 此类必须一个public无参数的构造器。
- 此类必须是可串行化的。
- 此类必须提供的String构造器来返回一个实例，并且与toString()方法返回的实例相等。
- 此类必须实现equals(), hashCode() 和toString()方法。在上面的PrimaryKey类中，这些是在BaseObject类中通过反射机制处理的。

5. 运行ant clean enhance，可以提到元数据强化了org.appfuse.model.User类。

图 7.4. 运行ant clean enhance的结果



```
Cygwin
$ant clean enhance
Buildfile: build.xml

clean:
[delete] Deleting directory C:\Source\myusers\build

compile:
[mkdir] Created dir: C:\Source\myusers\build\classes
[javac] Compiling 20 source files to C:\Source\myusers\build\classes
[mkdir] Created dir: C:\Source\myusers\build\test\classes
[javac] Compiling 7 source files to C:\Source\myusers\build\test\classes
[copy] Copying 4 files to C:\Source\myusers\build\classes

enhance:
[jpoxenhancer] JPOX Enhancer : Processing JDO Meta-Data C:\Source\myusers\build\classes\org\appfuse\model\package.jdo
[jpoxenhancer] JPOX Enhancer : Processing class org.appfuse.model.User
[jpoxenhancer] ENHANCED: org.appfuse.model.User

BUILD SUCCESSFUL
Total time: 10 seconds
$
```

6. 在目录web/WEB-INF创建了文件applicationContext-jdo.xml(使用文件applicationContext-empty.txt作为模板)，添加下面的persistenceManagerFactory定义，引用了LocalPersistenceManagerFactoryBean[<http://www.springframework.org/docs/api/org.springframework.orm.jdo/LocalPersistenceManagerFactoryBean.html>]。

```
<bean id="persistenceManagerFactory"
    class="org.springframework.orm.jdo.LocalPersistenceManagerFactoryBean">
    <property name="jdoProperties">
```

```

<props>
  <prop key="javax.jdo.PersistenceManagerFactoryClass">
    org.jpox.PersistenceManagerFactoryImpl
  </prop>
  <prop key="javax.jdo.option.ConnectionDriverName">
    com.mysql.jdbc.Driver
  </prop>
  <prop key="javax.jdo.option.ConnectionUserName">root</prop>
  <prop key="javax.jdo.option.ConnectionPassword"></prop>
  <prop key="javax.jdo.option.ConnectionURL">
    jdbc:mysql://localhost/myusers
  </prop>
  <prop key="org.jpox.autoCreateSchema">true</prop>
  <prop key="org.jpox.validateTables">false</prop>
  <prop key="javax.jdo.option.NontransactionalRead">
    true
  </prop>
</props>
</property>
</bean>

```

7. 添加一个transactionManager定义, 引用 `JdoTransactionManager` [<http://www.springframework.org/docs/api/org.springframework.orm.jdo/LocalPersistenceManagerFactoryBean.html>]. 这个bean将一个JDO PersistenceManager从特定的factory绑定到线程上, 这就意味着可以为每个factory分配一个线程的PersistenceManager。

```

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

```

8. 在目录src/org/appfuse/dao/jdo(你必须创建这个目录)创建UserDAOJdo.java类, 这个类继承了 `JdoDaoSupport` [<http://www.springframework.org/docs/api/org.springframework.orm.jdo/support/JdoDaoSupport.html>], 实现了UserDAO接口。

```

package org.appfuse.dao.jdo;
// user your IDE to organize imports

public class UserDAOJdo extends JdoDaoSupport implements UserDAO {
  public List getUsers() {
    Collection users = getJdoTemplate().find(User.class);
    users = getPersistenceManager().detachCopyAll(users);
    return new ArrayList(users);
  }
}

```

```
public User getUser(Long id) {
    User user = (User) getJdoTemplate().getObjectById(User.class, id);
    if (user == null) {
        throw new ObjectRetrievalFailureException(User.class, id);
    }
    return (User) getPersistenceManager().detachCopy(user);
}

public void saveUser(User user) {
    if (user.getId() == null) {
        getJdoTemplate().makePersistent(user);
    } else {
        getPersistenceManager().attachCopy(user, true);
    }
}

public void removeUser(Long id) {
    getJdoTemplate().deletePersistent(
        getJdoTemplate().getObjectById(User.class, id));
}
}
```

9. 在文件`applicationContext-jdo.xml`添加`userDAO` bean定义。

```
<bean id="userDAO" class="org.appfuse.dao.jdo.UserDAOJdo">
    <property name="persistenceManagerFactory">
        <ref bean="persistenceManagerFactory"/>
    </property>
</bean>
```

测试

测试这个类却没有前面那个类那么容易。对于任何写操作，JDO需要一个事务处理，除非你特别的声明它不需要。如果你运行`ant test -Dtestcase=UserDAO`，你的控制台的堆栈中会“抱怨”事务没有激活。

图 7.5. 运行ant test -Dtestcase=UserDAO的结果

```

$ant test -Dtestcase=UserDAO
Buildfile: build.xml

compile:

enhance:
[jpoxenhancer] JPOX Enhancer : Processing JDO Meta-Data C:\Source\myusers\build\
classes\org\appfuse\model\package.jdo
[jpoxenhancer] JPOX Enhancer : Processing class org.appfuse.model.User
[jpoxenhancer] ERROR - GeneratorBase.enhance(1409) ! Class org.appfuse.model.Use
r is already enhanced.
[jpoxenhancer] WARN - GeneratorBase.update(1516) ! class not updated : org.appfu
se.model.User
[jpoxenhancer] ENHANCED: org.appfuse.model.User

test:
[junit] Testsuite: org.appfuse.dao.UserDAOTest
[junit] Tests run: 3, Failures: 0, Errors: 3, Time elapsed: 1.391 sec

[junit] Testcase: testGetUsers(org.appfuse.dao.UserDAOTest):          Caused a
n ERROR
[junit] Transaction is not active; nested exception is org.jpox.exceptions.T
ransactionNotActiveException: Transaction is not active
[junit] org.springframework.orm.jdo.JdoUsageException: Transaction is not ac
tive; nested exception is org.jpox.exceptions.TransactionNotActiveException: Tra
nsaction is not active
[junit] org.jpox.exceptions.TransactionNotActiveException: Transaction is no
t active

```

JDO规范已经向你说明了可以通过将`javax.jdo.option.NontransactionalWrite`的值设置为`true`的方法来关闭这项操作，但JPOX支持这么做。实际上，你只想在进行保存操作时参与一个事务。如果你不曾拥有一个活动的事务，你必须新建一个。这就是`userManager` bean所有的`save*`都有一个`PROPAGATION_REQUIRED`事务属性的原因所在。

在正常情况下，`UserManager`的保存方法会参与一个事务处理，而且只会在`UserDAOTest`发生。为了证明是在这个类中出现的，而不是在`UserDAOJdo`中，运行`ant test -Dtestcase=UserManager`。

修正这个问题的最简单的方法，就是覆盖`userDAO` bean定义，将它包装成声明式的事务处理。

1. 在目录`test/org/appfuse/dao`中，创建一个名为`applicationContext-test.xml`，填充以下内容：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <!-- userDAO for testing UserDAOJdo -->
  <bean id="userDAO"
    class="org.springframework.transaction.interceptor.TransactionProxy
    FactoryBean">
    <property name="transactionManager">
      <ref bean="transactionManager"/>
    </property>
    <property name="target">
      <bean class="org.appfuse.dao.jdo.UserDAOJdo"
        autowire="byName"/>
    </property>
    <property name="transactionAttributes">

```

```
<props>
  <prop key="*">PROPAGATION_REQUIRED</prop>
</props>
</property>
</bean>
</beans>
```

2. 修改UserDAOTest的中的setUp()方法，在获取userDAO之前加载这个文件。

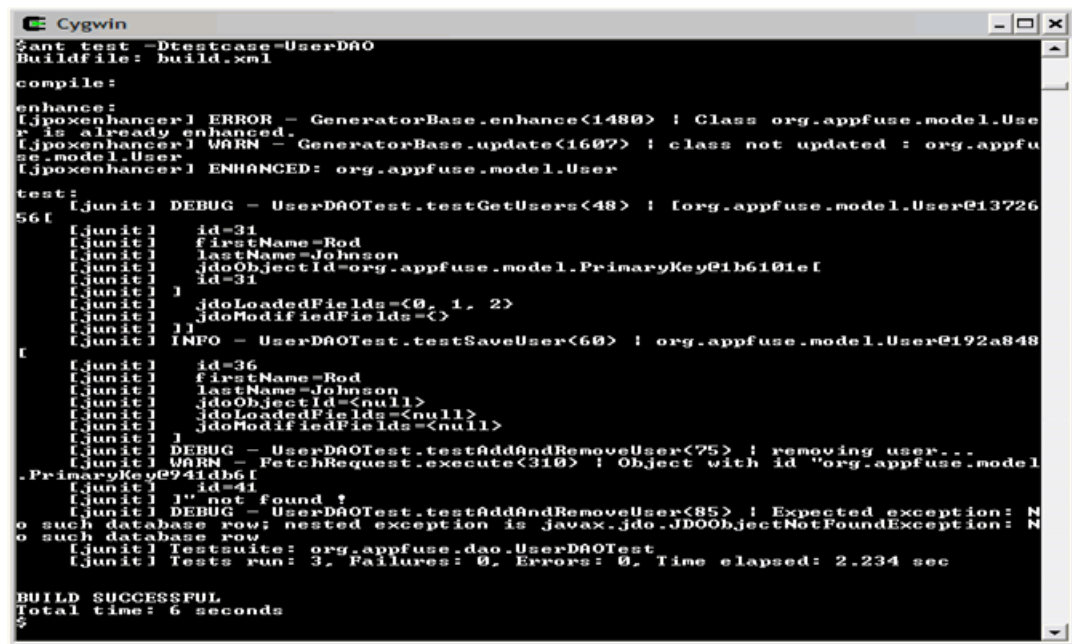
```
protected void setUp() throws Exception {
    String[] paths = {"/org/appfuse/dao/applicationContext-test.xml"};
    ctx = new ClassPathXmlApplicationContext(paths, ctx);
    dao = (UserDAO) ctx.getBean("userDAO");
}
```

3. 改进文件build.xml “compile” target，将这个文件复制到test的classpath中。

```
<!-- Copy JDO mapping files -->
<fileset dir="${src.dir}" includes="**/*.jdo"/>
</copy>
<!-- Copy overriding test files -->
<copy todir="${test.dir}/classes">
  <fileset dir="${test.src}" includes="**/*.xml"/>
</copy>
</target>
```

4. 运行ant test -Dtestcase=UserDAO。

图 7.6. 构建成功，运行测试ant test -Dtestcase=UserDAO的结果



```

Cygwin
$ ant test -Dtestcase=UserDAO
Buildfile: build.xml

compile:
[warn] [jpxenhancer] ERROR - GeneratorBase.enhance(1480) : Class org.appfuse.model.User
[warn] [jpxenhancer] WARN - GeneratorBase.update(1607) : class not updated : org.appfu
[warn] [jpxenhancer] ENHANCED: org.appfuse.model.User

test:
[debug] [junit] DEBUG - UserDAOTest.testGetUsers(48) : [org.appfuse.model.User@13726
56[
[debug] [junit] id=31
[debug] [junit] firstName=Rod
[debug] [junit] lastName=Johnson
[debug] [junit] jdoObjectId=org.appfuse.model.PrimaryKey@1b6101e1
[debug] [junit] id=31
[debug] [junit]
[debug] [junit] jdoLoadedFields=<0, 1, 2>
[debug] [junit] jdoModifiedFields=<>
[debug] [junit]
[debug] [junit] INFO - UserDAOTest.testSaveUser(60) : [org.appfuse.model.User@192a848
[
[debug] [junit] id=36
[debug] [junit] firstName=Rod
[debug] [junit] lastName=Johnson
[debug] [junit] jdoObjectId=<null>
[debug] [junit] jdoLoadedFields=<null>
[debug] [junit] jdoModifiedFields=<null>
[debug] [junit]
[debug] [junit] DEBUG - UserDAOTest.testAddAndRemoveUser(75) : removing user...
[debug] [junit] WARN - FetchRequest.execute(310) : Object with id "org.appfuse.model
.PrimaryKey@941db61
[debug] [junit] id=41
[debug] [junit] " not found ?
[debug] [junit] DEBUG - UserDAOTest.testAddAndRemoveUser(85) : Expected exception: N
o such database row; nested exception is javax.jdo.JDOObjectNotFoundException: N
[debug] [junit] Testsuite: org.appfuse.dao.UserDAOTest
[debug] [junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 2.234 sec

BUILD SUCCESSFUL
Total time: 6 seconds
$

```

另一种在单元测试中使用事务处理的方法，请参考Spring论坛 [http://forum.springframework.org/viewtopic.php?p=736#736]。

缓存

关缓存支持的更多信息，参考JPOX站点 [http://www.jpox.org/docs/1_1/cache.html]。

社区和支持

作为一种标准，JDO与其它持久性技术有很大的不同，各个厂商可以提供兼容的产品。现在已经有介绍JDO的书籍。从这个短小的教程中，你可能已经注意到，相对其它框架，它多出了一个需要改善的步骤。JDO项目提供了用户论坛 [http://www.jpox.org/servlet/forum/index]，优秀的文档 [http://www.jpox.org/docs/index.html/]，并且是提供JDO2.0支持的为数不多的项目之一。它还发布了一篇题为“JPOX with Spring [http://www.jroller.com/comments/raible/home/spring_jpox_tutorial]”的教程。

Kodo JDO是由SolarMetric提供的JDO的一种商业实现。和他们的产品一起，还提供了商业支持和培训。他们也提供了一个使用Spring和Kodo JDO的样例程序 [http://www.solarmetric.com/springjdo/springjdo-release-0.9.jar]。 [http://www.solarmetric.com/]

在过去的一年中，J2EE社区中的EJB3组和JDO组一直在较劲。两个组都在实现基于ORM的持久性框架，谁也不愿与另一方合作。EJB3组的大部分工作都放在基于Hibernate上，同时JDO组则试图把对关系数据库的更为友好些作为第一标准。

可喜的是，事情出现了转机，最近有报道 [http://www.javabloggy.org/thread.jspa?forumID=61&threadID=14630]称在J2EE5规划中，EJB3组和JDO组将展开合作为Java定义一种全新的，统一的POJO持久性模型。这种局面很可能催生一种新的ORM映射标准。两个组将使用这一标准定义他们的映射。

OJB

ObjectRelationalBride (OJB) [<http://db.apache.org/ojb>]是Apache上的一个项目，与Hibernate极其相似。实际上，它的介绍也差不多：一种透明的O/R映射工具，用来维护Java对象和关系数据库之间透明的持久性。基于这些相似性，本章不打算介绍它的工作原理。你可以在OJB的网站上找到一份完整的特性列表 [<http://db.apache.org/ojb/features.html>]。

本节介绍在Spring中如何介绍和使用OJB。

注意

Spring的OJB支持类位于包`org.springframework.orm.ojb` [<http://www.springframework.org/docs/api/org/springframework/orm/ojb/package-summary.html>]和`org.springframework.orm.ojb.support` [<http://www.springframework.org/docs/api/org/springframework/orm/ojb/support/package-summary.html>]中。

依赖包

本例中使用了OJB1.0.0。下面的jar文件作为OJB的一部分包含在MyUsers下载文件中。

- commons-dbcp.jar和commons-pool.jar:连接池的实现。
- db-ojb-1.0.0.jar:核心OJB类。

配置

集成OJB的第一步是创建一个repository文件 [<http://db.apache.org/ojb/docu/guides/repository.html>]，将对象映射到数据库表。

1. 新建一个名为`repository.xml`的文件，将放到`src/org/appfuse/model`目录中。

```
<descriptor-repository version="1.0">
  <jdbc-connection-descriptor jcd-alias="dataSource"
    default-connection="true" useAutoCommit="1" platform="MySQL">
    <sequence-manager
      className="org.apache.ojb.broker.util.sequence.
        SequenceManagerNativeImpl"/>
    </jdbc-connection-descriptor>
    <class-descriptor class="org.appfuse.model.User" table="app_user">
      <field-descriptor name="id" column="id" primaryKey="true"
        autoincrement="true" access="readonly"/>
      <field-descriptor name="firstName" column="first_name"/>
      <field-descriptor name="lastName" column="last_name"/>
    </class-descriptor>
  </descriptor-repository>
```

此文件的第一部分描述的是JDBC连接信息(<jdbc-connectiondescriptor>)，这有助于让OJB判定如何获取主键，所以对于匹配相应和数据库来说非常重要。jcd-alias属性指向了一个你要使用的dataSource bean。

2. 在web/WEB-INF目录下创建一个文件applicationContext-obj.xml。如果你在此目录，将文件applicationContext-jdo.xml重命名为applicationContext-jdo.xml.txt。同时还要依赖build.xml依赖关系，使test和war target不再依赖enhance，添加如下所示的dataSource bean。

```
<?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.jdbc.datasource.DriverManagerDataSource">
<property name="driverClassName">
<value>com.mysql.jdbc.Driver</value>
</property>
<property name="url">
<value>jdbc:mysql://localhost/myusers</value>
</property>
<property name="username"><value>root</value></property>
<property name="password"><value></value></property>
</bean>
</beans>
```

3. 添加一个transactionManager bean定义，引用PersistenceBrokerTransactionManager [http://www.springframework.org/docs/api/org.springframework.orm.obj/PersistenceBrokerTransactionManager.html]，此transaction manager与其它相似，它将一个PersistenceBroker [http://db.apache.org/obj/api/org/apache/obj/broker/PersistenceBroker.html]由指定的键名绑定到线程上。PersistenceBrokerTemplate [http://www.springframework.org/docs/api/org.springframework.orm.obj/PersistenceBrokerTemplate.html]会监控线程绑定的持久性代理，并自动参与事务处理。

```
<bean id="transactionManager"
class="org.springframework.orm.obj.PersistenceBrokerTransaction
Manager"/>
```

4. 添加一个objConfigurer bean。LocalObjConfigurer [http://www.springframework.org/docs/api/org.springframework.orm.obj/support/LocalObjConfigurer.html]将Spring的BeanFactory [http://www.springframework.org/docs/api/org.springframework.beans/factory/BeanFactory.html]给OBJ，这样它就可以使用Spring托管的DataSource。

```
<bean id="objConfigurer"
class="org.springframework.orm.obj.support.LocalObjConfigurer"/>
```

5. 配置Spring，让它能够识别Spring的dataSource bean。在默认的文件OBJ.properties中，覆盖一些设置来实现。

1. 下载默认的 OJB.properties [http://db.apache.org/ojb/OJB.properties.txt], 将它置于目录web/WEB-INF/classes中, 请确保其文件名为OJB.properties。
2. 将repositoryFile修改成如下:

```
repositoryFile=org/appfuse/model/repository.xml
```

3. 将ConnectionFactoryClass修改成如下, 使用Spring定义的DataSource。

```
ConnectionFactoryClass=org.springframework.orm.ojb.support.LocalDataSourceConnectionFactory
```

4. 修改ObjectCacheClass为每个代理提供第一级的缓存。

```
ObjectCacheClass=org.apache.ojb.broker.cache.ObjectCachePerBrokerImpl
```

6. 在目录src/org/appfuse/dao/ojb(你必须创建这个目录)创建类UserDAOObjb.java, 此类继承了 `PersistenceBrokerDaoSupport` [http://www.springframework.org/docs/api/org.springframework.orm.ojb.support/PersistenceBrokerDaoSupport.html], 实现了UserDAO接口。

```
package org.appfuse.dao.ojb;
// organize imports with your IDE

public class UserDAOObjb extends PersistenceBrokerDaoSupport implements
    UserDAO {
    public List getUsers() {
        return new ArrayList(getPersistenceBrokerTemplate().
            getCollectionByQuery(new QueryByCriteria(User.class)));
    }

    public User getUser(Long id) {
        Criteria criteria = new Criteria();
        criteria.addEqualTo("id", id);
        User user = (User) getPersistenceBrokerTemplate().
            getObjectByQuery(new QueryByCriteria(User.class,
                criteria));
        if (user == null) {
            throw new ObjectRetrievalFailureException(User.class, id);
        }
        return user;
    }

    public void saveUser(User user) {
```

```

    getPersistenceBrokerTemplate().store(user);
}

public void removeUser(Long id) {
    getPersistenceBrokerTemplate().delete(getUser(id));
}
}

```

7. 在文件web/WEB-INF/applicationContext-ojb.xml中加入这个类，作为userDAO。

```
<bean id="userDAO" class="org.appfuse.dao.ojb.UserDAOObjb"/>
```

8. 在JDO中创建的app_user表不允许id列为null。使用OJB时，最简单的方法就是为主键插入一个null值，并从数据库返回生成的主键。删除并重新创建app_user表。

1. 从命令行键入mysql -u root -p myusers登录MySQL。
2. 执行drop table app_user;。
3. 执行下面的SQL语句。

```

create table app_user (id bigint not null auto_increment,
first_name varchar(50), last_name varchar(50),
primary key (id));

```

测试

在运行UserDAOTest，修改setUp()方法，这样它不会覆盖你刚刚创建的用户DAO。下面是userDAO方法，除去了用于JDO时包装了事务的DAO。

```

protected void setUp() throws Exception {
    //String[] paths = {"/org/appfuse/dao/applicationContext-test.xml"};
    //ctx = new ClassPathXmlApplicationContext(paths, ctx);
    dao = (UserDAO) ctx.getBean("userDAO");
}

```

你也可以简单的修改文件applicationContextttest.xml中target bean的class属性来实现。

```

<property name="target">
    <bean class="org.appfuse.dao.ojb.UserDAOObjb"/>
</property>

```

运行 `ant test -Dtestcase=UserDAO`。如果你使用了第一个选项(注释 `setUp()` 方法的代码), 你会看到警告信息: “No running tx found”。第二个选项会消除这个警告。

缓存

访问OJB网站 [<http://db.apache.org/ojb/docu/guides/objectcache.html>]来获得有关缓存支持的信息。

社区和支持

OJB1.0于2004年6月30日发布。它是一个拥有大量实用实现的Apache项目。要获得支持, 你最好是使用它的邮件列表 [<http://db.apache.org/ojb/mail-lists.html>], 另外OJB的站点还有一些优秀的文档 [<http://db.apache.org/ojb/docu/tutorials/summary.html>]。

本章小结

本章探讨了Spring支持的几种持久性方案: Hibernate, iBATISSQLMap, 它内置的JDBC事务处理, JDO和OJB。你已经学习如何在MySQL应用程序中配置它们。个人认为, Hibernate和iBATIS是持久性方案的最佳工具。

Hibernate使用映射文件使得保持对象的持久性变得很容易, 你甚至可以用XDoclet来生成映射文件。iBATIS的理想情况是, 如果你有现有的复杂的数据结构或是遗留数据库, 或是你更愿意使用SQL。

本章向你演示了Spring作为一种配置工具和松散耦合辅助工具功能是多么强大。在5种持久性框架中, 你仅仅是要针对每种情况修改 `UserDAOTest`。Spring不仅促使了良好的设计, 而且让它变得更容易使用。

第 8 章 测试Spring应用程序

Spring如何简化应用程序的测试

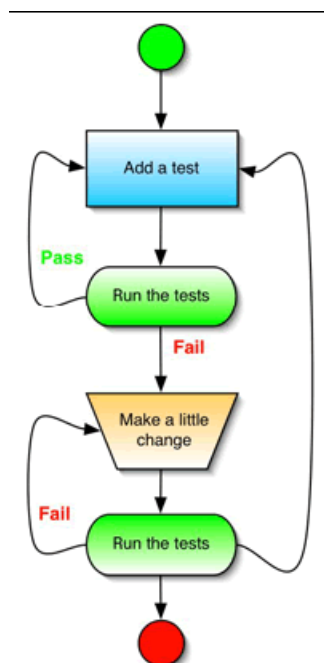
本章解释了如何运用测试驱动开发的方法创建高质量，经过完好测试，基于Spring的应用程序。你将到如何用诸如EasyMock，jMock和 DBUnit，测试你的组件。对于控制器，你将学到如何用Cactus进行容器内测试，用Spring Mock进行容器外测试。最后，你将学习如何用jWebUnit和Canoo的WebTest 测试web 接口。

概述

使用测试驱动开发是产生高质量代码的最佳途径。你不但可以在测试时思考你的应用应该是什么样子，而且你实际在设计你的应用程序应该履行的约定。如果你的测试没有按预期的通过，那么你的测试及其实现都会存在问题。如果测试和约定中期望的一样，你必须修改实现以产生预期结果。

测试先行(Test first)的开发是开发人员编写测试并且保证在继续之前它是失败的阶段。然后像婴儿学步一样在开发过程中不断的得到通过/失败的结果。左边的图¹演示了这种技术。

图 8.1. 测试先行开发模型



测试驱动和测试先行相似，最重要的一点是在编写测试的过程迫使你考虑你的类做些什么。测试先行鼓吹的是一种很多小步骤组成的更为严格的体系。

本章中，你将学习如何使用测试技术更加快速和有效的开发Spring应用程序。通过借鉴测试先行的开发，你完全可以消除基于浏览器应用程序所共有的屡试屡败(trial-and-error)过程。你还可以从编码中树立自信，并可以加入规则的自动化技术以保证没有人会打断这个构建程序。

¹此图基于Scott W. Ambler的一篇论文Test Driven Development [<http://www.agiledata.org/essays/tdd.html>]

测试简单的软件开发是一种非常轻松的体验，因为你很快可以问题所在。

Spring使得编写可测试的软件变得非常简单。它的IoC容器为你提供了自由空间按照你的意愿来设置类的依赖关系。你可以在你的测试中加载Spring的ApplicationContext，并且像你在应用程序中一样的使用它们，或者你可以利用mock object来设置bean的依赖关系。本章会探讨两种技术，并会向你演示使用诸如EasyMock和jMock之类的工具来为你类即时的创建不同的mock版本。你还可以看到在测试前如何用DBUnit从数据中加载数据。

注意

mock object是真实对象的赝品和抽象版。例如，你可以来初始化servlet API，这样你可以在一个servlet容器外进行测试。

为了测试应用程序中的Controller，你将会学习使用StrutsTestCase，Spring Mock和Cactus。为了测试view，你将学习如何使用jWebUnit和Canoo WebTest。

注意

今后会有一章涉及Anthill，CruiseControl和DamageControl。这些自动测试系统对于维护一个高质量的代码基础是必不可少的。

JUnit

JUnit很快就成了Java中TDD开发的事实上的标准。它是一个易用的框架，并且当今大多数IDE都支持它。如果你不使用IDE，你可能在使用Ant，通过<junit>任务同样很容易使用。在本章中，你将会使用JUnit来编写大部分测试。如果没有直接使用JUnit，你也在使用它的一种扩展技术。例如，DBUnit，Cactus和StrutsTestCase都是JUnit的扩展。

一个基本的JUnit测试包含一个testXXX方法，这里XXX是一个描述性的名称，它指明了这个测试要做什么。用一个名称来准确的描述测试很重要。例如，你有一个对象，要对结果进行过滤，testGetUsersWithNoFiltering()显然比一般性的testGetUsers()要好些。下面是一个演示JUnit测试的例子。

```
package org.appfuse.dao;
import junit.framework.TestCase;
import org.appfuse.model.User;

public class SimpleTest extends TestCase {
    public void testGetFullName() {
        User user = new User();
        user.setFirstName("Jack");
        user.setLastName("Raible");
        assertEquals(user.getLastName(), "Jack Raible");
    }
}
```

运行这个测试，得到失败的结果，因为你的取的是lastname，而不是fullname。首先运行一个失败的测试很重要，以确保它们和你的所期望的一致。将assertEquals()修改成下面的样子，将得到一个通过的测试。

```
assertEquals(user.getFullName(), "Jack Raible");
```


在一个JUnit测试中可以实现的可选的方法包括`setUp()`、`tearDown()`和`main(String[])`。`setUp()`和`tearDown()`在创建一个测试环境时非常有用，在进行一个测试前撤销环境设置。这些方法在每个`testXXX`执行前后调用。如果有一些跨测试的步骤，你应该在一个静态初始化块中实现这些方法，或是使用JUnit的`TestSetup`类。

对于从命令行使用`java`命令来运行测试，那么实现`main(String[])`方法是必不可少的。然而有了现在的IDE和Ant的支持，你没必要再实现这个方法了。为了证明这一点，删除`MyUsers`中`UserDAOTest`中`main(String[])`，测试照样可以运行。

在Java社区中，特别是在开源开发人员中，使用TDD开发渐渐成了标准。带有丰富测试套件的项目显然能从其它开发人员那里赢得更多的可信度。然而，对于如何实施测试驱动开发存在很大的分歧。很多人倡导使用Mock Object或是mocks，而另一些则认为集成测试已经足够了。

本章常常提到单元测试(unit test)。你可以认为这不是单元测试，而是集成测试(integrating test)。J. B. Rainsberger在JUnit Recipes [<http://www.manning.com/rainsberger>]作为很好的解释。

程序员所进行的测试通常叫做单元测试，但是我们不大愿意用这个术语。它被滥用了，与它本义相比，这产生了更多的混淆。作为一个社区，我们无法在什么是单元(unit)上达成一致意见，它是一个方法，一个类，还是代码路径？如果我们在单元意味什么上存在分歧，那么要想在什么是单元测试的问题上形成统一看法的机会是微乎其微的。

本章向你演示两种测试：模拟测试(mock testing)和集成测试(integration testing)。我们先看一下隐藏在它们背后的一些哲理。

模拟测试

模拟测试是指在隔离你的测试用例情况下来测试一个单独的工作单元(通常是一个类)的过程。使用mocks，你可以隔离环境中测试你的类，不需要依赖任何它所依赖的东西。在一个business façade(或Manager)实例中，你要模拟的是它所依赖的DAO。这样你可以测试business façade的某个特定的功能，而不需要放入DAO的功能。

一个mock通常是指一个stub(你自己写的一个仅用于测试的类)或是dynamic mock。Dynamic mocks是一种非常灵活的方法，可以在一个类或是接口上设置期望值(expectation)。使用dynamic mocks，你可以即你创建一个类的实例，写一些代码来设置期望的行为。使用mocks通用规则是，要么模拟你想修改的类型，要么模拟你的API。

集成测试

集成测试是在所有的依赖都很完整正常的环境中测试类的过程。集成测试最大的缺点是太快和非孤立的测试。例如，对于一个依赖实际DAO的business façade测试，你必须建立一个数据库连接来取得实际的数据。通过编写的运行非孤立的测试，你所写的测试依赖你的环境，实际上你的测试什么也没有做。

这里我们两种类型都要讨论。Dynamic mocks对一个团队开发来说太棒了，不同的开发人员负责不同的层。譬如，开发人员Bruce负责DAO层的开发，而开发人员负责业务逻辑层，它们根本没有理由依赖对方的实现。Bruce可以为James提供程序接口，James可以创建这些接口的mocks以便孤立的测试他的代码。那么Bruce是否提供他的DAO实现是无关紧要的。

如果你在一小团队或者一个人工作，那么集成测试已经足够了。当然，这意味你要按一定的顺序来开发程序，依赖的部分要优先开发，这样依赖它们的实现才能正常运行。

以我的观点，一个经过设计的应用程序，应该在数据库层进行集成测试，在业务层和web层进行模拟测试，在UI上进行集成测试。当然，所有这些依赖各层的复杂程度。对于数据层，重要的是测试与你的数据存储部件的交互情况。没有理由去测试一个数据库连接或是Spring所提供的Template。

这些是框架级的部件，不是应用程序要关心的内容。一旦你的DAO返回期望的结果，在业务层中验证这些是毫无意义的。所以你可以在业务(或服务)层模拟DAO，集中精力测试你的业务逻辑。

在web层，特别是使用MVC Controller(不管什么web框架)的情况下，模拟一个J2EE容器环境是非常有用的。Mocks可以模拟request, response, 或是其它与servlet相关的类，这样你就可以在容器外进行测试，这种方式常常会快些。在你的Controller的测试中使用真正的业务对象与mocks进行对比只是个大概的尝试。既然Controller中重要的功能是控制某种特定的输入页面返回的view，孤立地进行测试通常是个不错的主意。

UI层是测试全部集成功能的最佳位置。测试UI层包括隐藏使用程序时用户进行的操作：点击链接，输入数据，点击按钮等。这些测试不直接调用你的类，但扮演浏览器的动作。它们一般从一个URL中抽取HTML，解析它，然后提交基于你的测试代码的request参数。针对UI测试的框架让这一切变得简单，你可以仅写一行代码来提交表单，输入数据，验证页面标题或文本。

现在你理解了不同的测试策略，现在让我们看一下MyUsers应用程序中一些特例。本章中，和上一章一样，你可以跟着做练习。最简单的方法是从<http://sourcebeat.com/downloads>上下载MyUsers Chapter 8压缩包。项目树型目录结构即为第7章练习的结果。在目录web/WEB-INF/lib下，包含了本章要用到的所有jar文件。你可以继续用上一章开发应用程序。如果你这么做，请从<http://sourcebeat.com/downloads>上下载Chapter 8 JARs。每一节请注意你可能需要新jar文件，所以你可以用这一章作为集成这些原则到你的应用程序中的一份参考。

在第7章中，你已经集成了一些持久性策略。如果你完成了这一章的所有练习，你的MyUsers程序应该使用的是OJB，因为那是最后探讨的一个策略。如果你想在应用程序中改用其它框架，请参考第7章中针对那种框架的章节。如果你决定从OJB转用其它，有一些东西要记住。

- JDO需要对build.xml进行修改，这样test和war才能调用enhance。你可能还要修改UserDAOTest，这样你才能加载applicationContext-test.xml。
- 你可能要删除app_user表，重创它(可以参照不同的DAO类型)。

如果你对切换DAO有自己见解，可以发email给我<mattr@sourcebeat.com>或是到Atlassian [<http://www.atlassian.com/c/SB/10440>]提供的Spring Live Issue Tracker [<http://jira.sourcebeat.com/secure/BrowseProject.jspx?id=10000>]上发布自己的论点。

测试数据库层

正如前面所提到的，测试DAO的最好方法是直接在数据库上进行测试。这样能够保证你的持久层框架按照你的要求运作。测试数据层包括使用数据库相关的类进行查询，插入，更新和删除操作。回顾一下第2章中创建的用户DAOTest类。

```
package org.appfuse.dao;
// use your IDE to organize imports

public class UserDAOTest extends BaseDAOTestCase {
    private User user = null;
    private UserDAO dao = null;

    protected void setUp() throws Exception {
        dao = (UserDAO) ctx.getBean("userDAO");
    }

    protected void tearDown() throws Exception {
        dao = null;
    }
}
```

```
}

public void testGetUsers() {
    // add a record to the database so we have something to work with
    user = new User();
    user.setFirstName("Rod");
    user.setLastName("Johnson");
    dao.saveUser(user);
    List users = dao.getUsers();
    assertTrue(users.size() >= 1);
    assertTrue(users.contains(user));
}

public void testSaveUser() throws Exception {
    user = new User();
    user.setFirstName("Rod");
    user.setLastName("Johnson");
    dao.saveUser(user);
    assertTrue("primary key assigned", user.getId() != null);
    log.info(user);
    assertTrue(user.getFirstName() != null);
}

public void testAddAndRemoveUser() throws Exception {
    user = new User();
    user.setFirstName("Bill");
    user.setLastName("Joy");
    dao.saveUser(user);
    assertTrue(user.getId() != null);
    assertTrue(user.getFirstName().equals("Bill"));
    if (log.isDebugEnabled()) {
        log.debug("removing user...");
    }
    dao.removeUser(user.getId());
    try {
        user = dao.getUser(user.getId());
        fail("User found in database");
    } catch (DataAccessException dae) {
        log.debug("Expected exception: " + dae.getMessage());
        assertTrue(dae != null);
    }
}
}
```

这个类是一个集成测试，因为依赖Spring的ApplicationContext [http://www.springframework.org/docs/api/org.springframework.context/ApplicationContext.html]来获得UserDAO实现类。ApplicationContext使用BaseDAOTestCase(继承了这个类)中的ClassPathXmlApplicationContext [http://www.springframework.org/docs/api/org.springframework.context.support/ClassPathXmlApplicationContext.html]类进行初始化。

```
protected ApplicationContext ctx = null;
```

```
public BaseDAOTestCase() {
    String[] paths = { "/WEB-INF/applicationContext*.xml" };
    ctx = new ClassPathXmlApplicationContext(paths);
}
```

除了外 `ClassPathXmlApplicationContext`，还有一个 `FileSystemXmlApplicationContext` [<http://www.springframework.org/docs/api/org.springframework.context.support.FileSystemXmlApplicationContext.html>]。两种类都允许你使用Ant形式(*.xml)的语法来加载任何匹配指定模式的资源。如果你想用这个类加载context文件，你应该重写上面的代码，变成如下：

```
public BaseDAOTestCase() {
    String[] paths = { "web/WEB-INF/applicationContext*.xml" };
    ctx = new FileSystemXmlApplicationContext(paths);
}
```

提示

把你的bean写入一个.properties文件同样是可行的。参考 `PropertiesBeanDefinitionReader` [<http://www.springframework.org/docs/api/org.springframework.beans.factory.support.PropertiesBeanDefinitionReader.html>] 有关的更多细节 [<http://www.coffee-bytes.com/servlet/ShowEntryForId?id=56>]。

使用 `ClassPathXmlApplicationContext` 的优点在于你的文件不必在某个准确的位置，只要在你的classpath中即可。至于我使用 `/WEB-INF/applicationContext*.xml` 来代替 `/applicationContext*.xml` 的原因，就是因为第一个在我的classpath中，第二个却不在。classpath可以在Eclipse的.classpath文件和Ant的build文件中进行设置。

在Eclipse的.classpath中配置如下：

```
<classpathentry excluding="WEB-INF/classes/" kind="src" path="web"/>
```

Ant的test target设置如下：

```
<classpath>
  <path refid="classpath"/>
  <path location="${build.dir}/classes"/>
  <path location="${test.dir}/classes"/>
  <path location="web/WEB-INF/classes"/>
  <path location="web"/>
</classpath>
```

使用 `/WEB-INF/applicationContext*.xml` 是非常有用的，因为 `action-servlet.xml` 通过这个路径来引用 `validator` 资源文件，当使用那个类进行测试时classpath中需要包含 `web` 目录。当你把你的Controller测试类重构成一个独立的单元测试时，就不再需要加载这个context，你也没有必要修改这个路径。

DBUnit

DBUnit是一个测试框架，用于在运行测试之前把你的数据库置于一个已知状态。DBUnit包含一个你所需的(位于classpath中)唯一的jar文件，dbunit-2.1.jar。MyUsers下载文件和第8章的下载文件中已经包含了这个文件。你也可以从SourceForge.net上下载 [http://www.sourceforge.net/project/showfiles.php?group_id=47439&release_id=242511]。

在UserDAOTest中，在方法testGetUsers()添加一条新记录。重构这个类，使用DBUnit在运行测试前加载一条记录。你必须事先创建一些样例数据来填充数据库。使用DBUnit时，有好几个datasets [<http://www.dbunit.org/components.html#dataset>]可供使用，但最简单的是XmlDataSet [<http://www.dbunit.org/apidocs/org/dbunit/dataset/xml/XmlDataSet.html>]。

提示

要从已有的数据库中导出数据库，可以用DBUnit的Ant任务 [<http://www.dbunit.org/anttask.html>]。

1. 在test/data目录(你可能要创建这个目录)中创建一个sample-data.xml文件。填充如下的XML代码。

```
<?xml version="1.0" encoding="utf-8"?>
<dataset>
  <table name='app_user'>
    <column>id</column>
    <column>first_name</column>
    <column>last_name</column>
    <row>
      <value>1</value>
      <value>Rod</value>
      <value>Johnson</value>
    </row>
  </table>
</dataset>
```

2. 使用DBUnit，你可以继承它的DatabaseTestCase [<http://www.dbunit.org/howto.html#createTest>]或是你自己的TestCase。使用你自己的TestCase比修改父类会简单些。在BaseDAOTestCase添加以下两个变量作为成员变量。

```
private IDatabaseConnection conn = null;
private IDataset dataSet = null;
```

3. 创建setUp()方法来清理sampledata.xml中指定的所有的数据库表。

```
protected void setUp() throws Exception {
  DataSource ds = (DataSource) ctx.getBean("dataSource");
  conn = new DatabaseConnection(ds.getConnection());
  dataSet = new XmlDataSet(new FileInputStream(
    "test/data/sample-data.xml"));
  // clear table and insert only sample data
}
```

```
DatabaseOperation.CLEAN_INSERT.execute(conn, dataSet);
}
```

4. 在`tearDown()`中添加逻辑代码，关闭数据库，删除任何添加的数据。

```
protected void tearDown() throws Exception {
    // clear out database
    DatabaseOperation.DELETE.execute(conn, dataSet);
    conn.close();
    conn = null;
}
```

5. 更换`UserDAOTest`，这样`setUp()`和`tearDown()`就可以调用`super.setUp()`和`super.tearDown()`。

```
protected void setUp() throws Exception {
    super.setUp();
    dao = (UserDAO) ctx.getBean("userDAO");
}

protected void tearDown() throws Exception {
    super.tearDown();
    dao = null;
}
```

注意

如果你不愿显式的从context中抓取bean，可以在单元测试中使用Dependency Injection(依赖注射) [http://www.theserverside.com/news/thread.tss?thread_id=27445]。显式的获得bean允许你按自己的命名约定来使用这个变量。在Spring 1.1.1中，加入了`AbstractDependencyInjectionSpringContextTests` [http://www.springframework.org/docs/api/org.springframework.test.AbstractDependencyInjectionSpringContextTests.html]，以便你在测试中使用Dependency Injection。

6. 修改`getUsers()`验证数据库中，只有一条记录，就是DBUnit中添加那条。

```
public void testGetUsers() {
    List users = dao.getUsers();
    assertTrue(users.size() == 1);
    User user = (User) users.get(0);
    assertEquals(user.getFullName(), "Rod Johnson");
}
```

运行`ant -Dtestcase=UserDAO`你会得到“BUILD SUCCESSFUL”的结果，数据库`app_user`应该是空的。在性能测试对比中，本次测试比前面测试多用了0.21秒(0.056 vs.0.77)。

Ant是另一个可在你的应用程序使用DBUnit的途径。你可以配置一个target在运行测试前加载数据。AppFuse使用了这种策略。作为一个例子，你可以参考它的build.xml [<https://appfuse.dev.java.net/source/browse/appfuse/build.xml?rev=HEAD&content-type=text/plain>]及其dbload target。这样做很有用，因为你可以在一大批测试中使用预定义的数据，而不需要为每个测试加载数据。使用这种策略的好处是你的测试套件可以运行得更快，因为在测试运行之前，数据库不需要归位。在你的类中直接使用DBUnit的一个优点是，你可以在IDE中运行测试，而不用顾虑任何Ant依赖。

数据库切换

使用Spring配置数据库连接，让你在测试代码中变换数据库变得非常容易。例如，你可以在单元测试时使用HSQL数据库，而在出成品时切换到一个DB2数据库中。大多数情况下，针对你的成品数据库进行测试通常是个不错的主意，但如果你的开发人员使用的是PowerPack，DB2没有针对Mac OS X的安装包。

配置数据库切换的一种方法是使用两种不同的属性(properties)文件。大多数Spring样例程序(在CVS/samples)使用了这种策略。通过定义一个propertyConfigurer Bean(使用PropertyPlaceholderConfigurer [<http://www.springframework.org/docs/api/org.springframework.beans.factory.config.PropertyPlaceholderConfigurer.html>])，你可以使用Ant形式的点位符：\${...}来配置你的dataSource属性。这样，你可以引用测试设置和成品设置，还可以在你的Spring配置文件和build.xml之间共享数据库配置。要为MyUsers中的数据库配置一个属性文件，你要完成以下步骤：

1. 修改当前DAO策略的XML文件。这里是指applicationContext-obj.xml。将dataSource的属性替换成Ant形式的属性。

```
<bean id="dataSource"
  class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <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>
```

2. 在web/WEB-INF/classes目录中创建一个jdbc.properties文件。

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost/myusers
jdbc.username=root
jdbc.password=
```


- 向文件 `web/WEB-INF/applicationContext.xml` 中添加一个 `PropertyPlaceholderConfigurer` bean。这个定义的底部注释掉的部分向你演示了如何使用一个单一的 `location` 属性来替换 `locations` 属性。

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

要指定多个属性文件，使用 `locations` 属性代替 `location` 属性。

```
<property name="locations">
  <list>
    <value>classpath:jdbc.properties</value>
  </list>
</property>
```

- 暴露 `jdbc.properties` 给 `build.xml`，在 `build.xml` 的顶部把它作为一个属性文件添加进去。

```
<property file="build.properties"/>
<property file="web/WEB-INF/classes/jdbc.properties"/>
```

- 更改 `populate` target 中的 `<sql>` task，改成如下：

```
<sql driver="{jdbc.driverClassName}" url="{jdbc.url}"
      userid="{jdbc.username}" password="{jdbc.password}">
```

注意

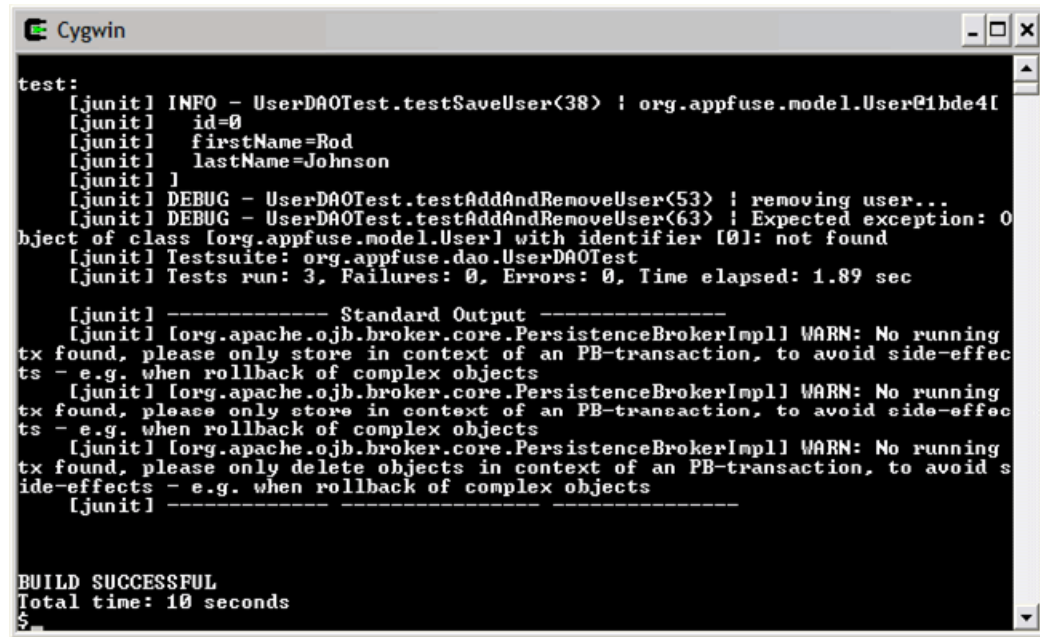
另一种策略是在测试时使用一个手动配置的 `Data Source` [https://appfusedevjava.net/source/browse/appfuse/test/applicationContext-database.xml?rev=HEAD&content-type=text/plain]，在成品时使用一个 `JNDI Data Source` [https://appfusedevjava.net/source/browse/appfuse/test/applicationContext-database.xml?rev=HEAD&content-type=text/plain]。

覆写Bean便于测试

如果你有一些DAO，其方法要求在其中包装事务处理，你可以针对单元测试定义一些bean。在第7章的JDO DAO部分已经这么做了，因为JDO要求任何 `makePersistent()` 都必须在一个事务内

部完成。在测试中使用Spring声明式的事务比使用Transaction要简单些。如果你仍在使用OJB DAO，你可能会得到一些警告信息提示没有运行事务。

图 8.2. 运行OJB DAO时的警告信息



```
test:
[junit] INFO - UserDaoTest.testSaveUser(38) : org.appfuse.model.User@1bde4f1
[junit] INFO - id=0
[junit] INFO - firstName=Rod
[junit] INFO - lastName=Johnson
[junit] INFO - ]
[junit] DEBUG - UserDaoTest.testAddAndRemoveUser(53) : removing user...
[junit] DEBUG - UserDaoTest.testAddAndRemoveUser(63) : Expected exception: 0
ject of class [org.appfuse.model.User] with identifier [0]: not found
[junit] Testsuite: org.appfuse.dao.UserDaoTest
[junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 1.89 sec

[junit] ----- Standard Output -----
[junit] [org.apache.ojb.broker.core.PersistenceBrokerImpl] WARN: No running
tx found, please only store in context of an PB-transaction, to avoid side-effec
ts - e.g. when rollback of complex objects
[junit] [org.apache.ojb.broker.core.PersistenceBrokerImpl] WARN: No running
tx found, please only store in context of an PB-transaction, to avoid side-effec
ts - e.g. when rollback of complex objects
[junit] [org.apache.ojb.broker.core.PersistenceBrokerImpl] WARN: No running
tx found, please only delete objects in context of an PB-transaction, to avoid s
ide-effects - e.g. when rollback of complex objects
[junit] -----

BUILD SUCCESSFUL
Total time: 10 seconds
$
```

要修正这个问题，创建一个bean定义，仅仅用于测试，在UserDaoTest中加载此定义，覆写正常的UserDAO。

1. 在目录test/org/appfuse/dao，新建一个文件applicationContext-test.xml，定义了有一个包装事务处理的UserDAO bean。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <!-- userDAO for testing UserDaoJdo -->
  <bean id="userDAO"
    class="org.springframework.transaction.interceptor.TransactionProxy
    FactoryBean">
    <property name="transactionManager">
      <ref bean="transactionManager"/>
    </property>
    <property name="target">
      <bean class="org.appfuse.dao.ojb.UserDAOObj" autowire="byName"/>
    </property>
    <property name="transactionAttributes">
      <props>
        <prop key="*">PROPAGATION_REQUIRED</prop>
      </props>
    </property>
  </bean>
```

```
</beans>
```

2. 重构UserDAOTest的setUp()方法，在每个测试之前加载这个文件。

```
protected void setUp() throws Exception {
    super.setUp();
    String[] paths = {"/org/appfuse/dao/applicationContext-test.xml"};
    ctx = new ClassPathXmlApplicationContext(paths, ctx);
    dao = (UserDAO) ctx.getBean("userDAO");
}
```

3. 现在UserDAOTest应该可以运行(ant test -Dtestcase=UserDAO)，而不会出现事务处理警告。

注意

你在UserManagerTest不会使用这个测试配置，因为事务中已经包装了userManager这个bean。

在测试中使用JNDI DataSource

正如前面提到的那样，你可以定义两种不同的JNDI DataSource，分别用于测试和成品阶段。成品bean很可能就是一个DataSource，可以通过JNDI来查找。下面是一种定义样例。

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

要在你的测试中使用这个bean，使用Spring的JNDI mock将它绑定到一个简单的JNDI实现上。

```
try {
    SimpleNamingContextBuilder builder =
        SimpleNamingContextBuilder.emptyActivatedContextBuilder();
    DriverManagerDataSource ds = new DriverManagerDataSource();
    ds.setDriverClassName("com.mysql.jdbc.Driver");
    ds.setUrl("jdbc:mysql://localhost/myusers");
    ds.setUsername("root");
    ds.setPassword("");
    builder.bind("java:comp/env/jdbc/myusers", ds);
} catch (NamingException ne) {
    // do nothing, test will fail on its own
}
```

为了进行尝试，将JNDI DataSource放在applicationContext-test.xml中，在文件BaseDAOTestCase.java开头的静态初始化块中添加context build代码(如上所示)。对于上面的

DriverManagerDataSource的属性，你可以使用一个ResourceBundle，从jdbc.properties提取数据。

要在Tomcat中使用JNDI DataSource，将你的DAO context文件中dataSource bean替换成容器中配置的相应的JNDI。下面的步骤是针对Tomcat5.x。

1. 在\$CATALINA_HOME/conf/Catalina/localhost中创建文件myusers.xml，填充以下XML代码。

```
<Context path="/myusers" docBase="myusers" debug="0" reloadable="true">
  <Resource name="jdbc/myusers" auth="Container"
    type="javax.sql.DataSource"/>
  <ResourceParams name="jdbc/myusers">
    <parameter>
      <name>factory</name>
      <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
    </parameter>
    <parameter><name>maxActive</name><value>30</value></parameter>
    <parameter><name>maxIdle</name><value>5</value></parameter>
    <parameter><name>maxWait</name><value>5000</value></parameter>
    <parameter><name>username</name><value>root</value></parameter>
    <parameter><name>password</name><value></value></parameter>
    <parameter>
      <name>driverClassName</name>
      <value>com.mysql.jdbc.Driver</value>
    </parameter>
    <parameter>
      <name>defaultAutoCommit</name>
      <value>true</value>
    </parameter>
    <parameter>
      <name>url</name>
      <value>jdbc:mysql://localhost/myusers</value>
    </parameter>
  </ResourceParams>
</Context>
```

2. 从web/WEB-INF/lib复制一份MySQL JDBC Driver(mysql-connector-java-3.0.14-production-bin.jar)到\$CATALINA_HOME/common/lib。
3. 在文件web/WEB-INF/web.xml的末尾添加以下XML片断。

```
<resource-ref>
  <description>Connection Pool</description>
  <res-ref-name>jdbc/myusers</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

现在如果运行ant deploy来重启Tomcat，结果和前面一样。

仅加载一次Context

为了提高你的单元测试的速度，特别是用于加载ApplicationContext的那个，修改你的测试，以便context在整个测试过程中仅加载一次。在前面例子中，context是在运行每个testXXX()方法之前，通过setUp()方法加载的。你可以通过两种方法来实现。

- 使用的JUnit的TestSetup来配置once-per-test方法。
- 在你的测试中使用静态初始化块(static initialization block)来一次性初始化资源。

不管你使用哪种方法，你需要将一些变量修改成静态的(static)。

1. 将BaseDAOTest中的ctx修改成静态的。

```
protected static ApplicationContext ctx = null;
```

2. 将UserDAOTest中的dao修改成静态的。

```
private static UserDAO dao = null;
```

3. 第2个静态初始化块比较容易实现，只要将下面程序块添加到BaseDAOTestCase的开头。

```
static {
    String[] paths = {"/WEB-INF/applicationContext*.xml",
        "/org/appfuse/dao/applicationContext-test.xml"};
    ctx = new ClassPathXmlApplicationContext(paths);
}
```

4. 删除UserDAOTest.setUp()中加载测试context的代码块。

```
public void setUp() throws Exception {
    super.setUp();
    String[] paths = {"/org/appfuse/dao/applicationContext-test.xml"};
    ctx = new ClassPathXmlApplicationContext(paths, ctx);
    dao = (UserDAO) ctx.getBean("userDAO");
}
```

现在运行ant test -Dtestcase=UserDAO应该会快一些。但差距可以忽略不计，但随着context文件的增长，差距就会拉大。

虽然使用TestSetup非常方便，但本章并不作介绍。使用一个静态初始化块已经可以很好的解决问题，并且速度比TestSetup快一些[http://www.jroller.org/page/raible?anchor=one_time_setup_with_testsetup]。使用TestSetup还是用

静态初始化块纯粹取决于个人偏好。还其它一些备选方案，如Cedric Beust的TestNG [<http://www.beust.com/testng/>]。更多有关TestSetup的信息，以及JUnit如何每个测试初始化对象，请参考Martin Fowler的网站 [<http://www.martinfowler.com/bliki/JunitNewInstance.html>]。

测试Service层

在上一节，你已经学习了如何测试数据库的DAO实现。这非常重要，数据库层可以用来验证存入数据库的数据和你从中取出的东西是否完全相同。但是，即使你的DAO通过了测试和验证，你没有理由在你的service层中使用它们。

Spring一个优秀的特性是，它不仅加强了基于接口的设计，而且在如何写单元测试上没有任何限制。正如最后一节演示的那样，你可以在测试中加载context文件，并可以像在产品环境中那样操作你的bean。这是一个容易被忽略的重要的概念。很多人都沉迷于模拟所有的东西，在孤立测试这当然不错，但是它遗漏了测试应用程序各层之间的交互问题。

你可以总是想优先尝试使用集成测试(在测试中加载context文件，并操作相应的bean)。因为它容易设置，并且测试的真实代码，而不是伪对象。但是在下面的两种情况下却行不通。

- 在团队开发环境下，每个开发人员负责不同的层。在写测试之前你没有必要等待上一层的实现。
- 应用程序的context文件日益增大，以致加载它要好几秒钟。

基于这些原因，使用mocks来虚拟类的依赖关系。

模拟DAO对象

本节重构UserManagerTest，使用Mock模拟userDAO。mock分为两类，stubs，需要自己动手创建，dynamic mocks，允许你即时的实现类。在创建dynamic mocks的过程中，开发人员应该设置好方法被调用时的期望结果。Dynamic mocks更容易使用，并且不会在你的源码树中产生垃圾类。

EasyMock

EasyMock [<http://www.easymock.org/>]是一个在JUnit测试中为接口提供了mocks的项目，它使用Java的Proxy机制 [<http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html>]即时生成。类的继承同样可用于模拟实现类。为了在测试使用EasyMock，执行以下操作：

1. 为你想模拟的接口创建一个MockControl。
2. 从MockControl中取得mock。
3. 指定mock的行为(录音状态)。
4. 利用此控制器激活mock(回放状态)。
5. 执行你想测试的方法。
6. 验证调用的方法是否和预期的一样。

下面的代码向你演示了这些步骤。

```
private UserManager mgr = new UserManagerImpl();
```

```
private MockControl control;
private UserDAO mockDAO;

protected void setUp() throws Exception {
    // Create a MockControl
    control = MockControl.createControl(UserDAO.class);
    // Get the mock
    mockDAO = (UserDAO) control.getMock();
    // Set required dependencies
    mgr.setValidator(new UserValidator());
    mgr.setUserDAO(mockDAO);
}

protected void testGetUser() {
    // Set expected behavior
    mockDAO.removeUser(new Long(1));
    control.setVoidCallable();
    // Active the mock
    control.replay();
    // Execute method to test
    mgr.removeUser("1");
    // Verify methods called
    control.verify();
}
```

这段代码并没有调用userDAO的实现类。而是使用EasyMock，从它的接口创建了一个mock和一个动态的实现。

在MyUsers应用程序中，有现成的UserManagerTest用于中间层的集成测试。这个类加载了applicationContext.xml文件，并且像你在应用程序中一样的使用这些bean。用EasyMock来重构这个类非常容易，并且在长跑中跑得更快。你还不不如保持原始的UserManagerTest不变，因为在以后还可以当一个不错的集成测试用。

在目录test/org/appfuse/service中创建一个名为UserManagerEMTest的新类。本例中，这个类不包含任何Spring的依赖关系，并且拥有和UserManagerTest类一样的基本功能。最大的区别在于这个类是孤立的(感谢mocks)，而且运行得非常快。其代码如下：

```
package org.appfuse.service;
// use your IDE to organize imports

public class UserManagerEMTest extends TestCase {
    private static Log log =
        LoggerFactory.getLog(UserManagerEMTest.class);
    private UserManager mgr = new UserManagerImpl();
    private MockControl control;
    private UserDAO mockDAO;

    protected void setUp() throws Exception {
        control = MockControl.createControl(UserDAO.class);
        mockDAO = (UserDAO) control.getMock();
        mgr.setValidator(new UserValidator());
        mgr.setUserDAO(mockDAO);
    }
}
```

```

    }

    public void testAddAndRemoveUser() throws Exception {
        User user = new User();
        user.setFirstName("Easter");
        user.setLastName("Bunny");
        // set expected behavior on dao
        mockDAO.saveUser(user);
        control.setVoidCallable();
        // switch from record to playback
        control.replay();
        user = mgr.saveUser(user);
        assertEquals(user.getFullName(), "Easter Bunny");
        control.verify();
        if (log.isDebugEnabled()) {
            log.debug("removing user...");
        }
        // set userId since Hibernate doesn't do it
        String userId = "1";
        user.setId(new Long(userId));
        // reset to record state
        control.reset();
        mockDAO.removeUser(new Long(1));
        control.setVoidCallable();
        control.replay();
        mgr.removeUser(userId);
        control.verify();
        try {
            // reset to record state
            control.reset();
            control.expectAndThrow(mockDAO.getUser(user.getId()),
                new ObjectRetrievalFailureException(
                    User.class, user.getId()));
            // switch to playback
            control.replay();
            user = mgr.getUser(userId);
            control.verify();
            fail("User '" + userId + "' found in database");
        } catch (DataAccessException dae) {
            log.debug("Expected exception: " + dae.getMessage());
            assertNotNull(dae);
        }
    }
}

```

通过`ant test -Dtestcase=UserManagerEM`来运行个测试。这个类与`UserManagerTest`之间的速度差别非常大。在我的测试中，`UserManagerEM`花了0.172秒，而`UserManagerTest`则花了1.59秒。

jMock

jMock [<http://www.jmock.org/>]是另一个开源项目，用于生成dynamic mocks。jMock的模拟方法和EasyMock有点不同。它要求写一个类继承MockObjectTestCase [<http://www.jmock.org/docs/javadoc/org/jmock/MockObjectTestCase.html>]。这个类执行的是方法调用验证，并且提供了一层糖衣，这使jMock测试更易于阅读。为了在测试中使用jMock，执行以下步骤：

- 为你要模拟的接口创建一个Mock [<http://www.jmock.org/docs/javadoc/org/jmock/Mock.html>]。
- 在此mock上设置预期的行为。
- 执行你要测试的方法。
- 验证预期结果。

下面的代码是以上几步的一个简单样例。

```
private UserManager mgr = new UserManagerImpl();
private Mock mockDAO;

protected void setUp() throws Exception {
    // Create a Mock
    mockDAO = new Mock(UserDAO.class);
    // Set dependencies
    mgr.setValidator(new UserValidator());
    mgr.setUserDAO((UserDAO) mockDAO.proxy());
}

protected void testGetUser() {
    // Set expected behavior
    mockDAO.expects(once()).method("getUser")
        .with( eq(new Long(1)));
    // Execute method to test
    mgr.removeUser("1");
    // Verify expectations
    mockDAO.verify();
}
```

jMock的语法更简洁，但它的预期设置语法却要复杂些。jMock另一个闪光点就它允许你写自定义的stub [<http://www.jmock.org/custom-stubs.html>]来模拟调用方法的副作用。例如，在UserDAOHibernate.saveUser()，User对象由Hibernate分配一个id(如果它不存在的话)。

为了在模拟UserDAO时模仿同样的功能，在目录test/org/appfuse/service中创建一个类AssignIdStub。此类的代码逻辑部分只是为用户对象的id设置一个随机的Long。

```
package org.appfuse.service;
// use your IDE to organize imports

public class AssignIdStub implements Stub {
    public StringBuffer describeTo(StringBuffer buffer) {
        return buffer.append("assigns random id to an object");
    }
}
```



```
}

public Object invoke(Invocation invocation) throws Throwable {
    Long id = new Long(new Random().nextInt(10));
    ((User) invocation.parameterValues.get(0)).setId(id);
    return null;
}
}
```

在同一目录中创建UserManagerJMTest类。

```
package org.appfuse.service;
// use your IDE to organize imports

public class UserManagerJMTest extends MockObjectTestCase {
    private static Log log =
        LogFactory.getLog(UserManagerJMTest.class);
    private UserManager mgr = new UserManagerImpl();
    private Mock mockDAO;

    protected void setUp() throws Exception {
        mockDAO = new Mock(UserDAO.class);
        mgr.setValidator(new UserValidator());
        mgr.setUserDAO((UserDAO) mockDAO.proxy());
    }

    public void testAddAndRemoveUser() throws Exception {
        User user = new User();
        user.setFirstName("Easter");
        user.setLastName("Bunny");
        // set expected behavior on dao
        mockDAO.expects(once()).method("saveUser")
            .with( same(user) ).will(assignId());
        user = mgr.saveUser(user);
        // verify expectations
        mockDAO.verify();
        assertEquals(user.getFullName(), "Easter Bunny");
        assertTrue(user.getId() != null);
        if (log.isDebugEnabled()) {
            log.debug("removing user...");
        }
        String userId = user.getId().toString();
        mockDAO.expects(once()).method("removeUser")
            .with( eq(new Long(userId)) );
        mgr.removeUser(userId);
        // verify expectations
        mockDAO.verify();
        try {
            // set expectations
            Throwable ex =
                new ObjectRetrievalFailureException(
                    User.class, user.getId());
```

```

mockDAO.expects(once()).method("getUser")
    .with( eq(new Long(userId)))
    .will(throwException(ex));
user = mgr.getUser(userId);
// verify expectations
mockDAO.verify();
fail("User '" + userId + "' found in database");
} catch (DataAccessException dae) {
    log.debug("Expected exception: " + dae.getMessage());
    assertNotNull(dae);
}
}

private Stub assignId() {
    return new AssignIdStub();
}
}

```

运行 `ant test -Dtestcase=UserManagerJM` 进行测试。这个测试应该和EasyMock的测试一样快，并且它的输出结果应该和下面的截图差不多。

图 8.3. 运行 `ant test -Dtestcase=UserManagerJM` 测试的输出结果

```

Cygwin
$ ant test -Dtestcase=UserManagerJM
Buildfile: build.xml

compile:

test:
[junit] DEBUG - UserValidator.validate(18) ! entering 'validate' method...
[junit] DEBUG - UserManagerJMTest.testAddAndRemoveUser(44) ! removing user..
-
[junit] DEBUG - UserManagerJMTest.testAddAndRemoveUser(67) ! Expected except
ion: Object of class org.appfuse.model.User1 with identifier [9]: not found
[junit] Testsuite: org.appfuse.service.UserManagerJMTest
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.157 sec

BUILD SUCCESSFUL
Total time: 2 seconds
$

```

本节向你演示了如何成功地对business façades进行单元测试，在你的测试策略中如何使用Mock来大大节省测试运行的时间。

在这些测试中，`UserManagerImpl`由默认的构造器进行初始化，并且用一个mock对象来模拟它与`UserDAO`的交互。Spring没有出现在任何模拟测试中，事实上它是不可缺少的。Spring的基于setter的依赖注射理论使得你按自己的想法设置任何依赖都变得非常容易。

如果你想进一步了解jMock和EasyMock之间的差别，jMock网站上提供了一份详细的比较[<http://www.jmock.org/easymock-comparison.html>]。以我的经验，jMock更加灵活，但它的API有点让人摸不着头脑。它强制要求你继承它的MockObjectTestCase，你却无法使用它。jMock拥有大量的活跃的邮件列表，但EasyMock得益于它发布较早，并且得到广泛应用。

测试Web层

在一个应用程序中测试web层是一个应用程序测试套件中最重要的部分。通过正确的测试你的Controller，你可以验证应用程序的流程控制，断定输入是否会返回预期的输出结果。在Spring MVC中，输入指的是request，输出是由Controller方法返回的ModelAndView [http://www.springframework.org/docs/api/org.springframework.web.servlet.ModelAndView.html]对象。Spring Mock允许你简单的模仿request，并验证结果。你可以用Cactus进行容器内测试。

测试Controller固然重要，但通过与用户界面(UI)进行交互的方式来测试View通常是确保你的应用程序能够正常运行的最佳方法。你可能需要一个质检部门(Quality Assurance, QA)来完成这项任务。但是，使用jWebUnit [http://jwebunit.sf.net/]和Canoo WebTest [http://webtest.canoo.com/]测试UI并不难，两者都是HttpUnit的简化版本。HttpUnit模拟了浏览相应的部分，包括表单提交，JavaScript，基本的http认证，cookies和页面自动跳转。它允许用Java测试代码来将返回的页面作为文本，XML DOM或是装载form，table，link的容器来分析。与UI交互的测试是大型的集成测试，它验证应用的程序的所有层。当然你不用验证颜色，字体或是位置是否正确；这需要用人肉眼去判定。

你可以自动完成本章提及的所有的测试。也就是说，你可以在没有人为干预的情况下完成测试。这种能力利用了一种健康的持续构建机制，你将在自动化测试一节中探讨。

测试Controller

在第4章中，你已经为MyUsers程序中的两个主要的Controller，UserController和UserFormController编写了JUnit [http://www.junit.org/]测试。本节中，你将重构这些测试，使其不再依赖Spring的ApplicationContext [http://www.springframework.org/docs/api/org.springframework.context.ApplicationContext.html]。你将使用EasyMock和jMock来模拟Controller的依赖关系。

你会创建Cactus测试来学习如何在容器内测试Controller。你将学习如何在这些测试中孤立你测试，使它们能够成为单元测试(仅测试单个类)。你还将学习如何为FileUploadController写一个测试，如何用Dumbster [http://www.quintanasoft.com/dumbster]来测试邮件发送。

注意

Dumbster是一个虚拟的SMTP服务器，用单元和系统测试程序发送邮件消息。它会响应回所有标准SMTP命令，而不会将消息递交给用户。这些消息储存在Dumbster里面，用于后面的提取和验证。

Spring Mock

Spring Mock是Spring用于测试JNDI [http://www.springframework.org/docs/api/org.springframework.mock.jndi/package-summary.html]和Context [http://www.springframework.org/docs/api/org.springframework.mock.web/package-summary.html]的mock类的一个能用的名称。这些类最初是Spring用来测试框架本身的。当开发人员用mock(这里特指Mock Object的Servlet API)来测试Spring的Controller时，很明显，Mock Object还无法提供丰富的够用的功能。到2002年6月，Spring开发人员意识到它们的mock功能已经非常丰富了，他们着手将它作为Spring的一部分发布出来(从1.0.2)。

Spring Mock的优秀特性之一就是这些类是用于测试Servlet API，并没有捆死在Spring上。这就是说，你可以用它来测试其它Controller，如WebWork。以我的经验看，Struts，Spring和WebWork在测试时都能支持得非常好。Struts通过StrutsTestCase实现，Spring有自己的mock，而WebWork可以利用常规的JUnit测试。以后会有一章在MyUsers中实现WebWork，Tapestry和JSF。和请求/响应的web框

架不一样，Tapestry和JSF是基于组件的，事件驱动的web框架。它们常常依赖JavaScript，目前针对它们Controller组件的单元测试并没有足够支持。

测试Controller

下面是在第4章创建的UserControllerTest的代码。

```
package org.appfuse.web;
// use your IDE to organize imports

public class UserControllerTest extends TestCase {
    private static Log log = LogFactory.getLog(UserControllerTest.class);
    private XmlWebApplicationContext ctx;

    public void setUp() {
        String[] paths = {"/WEB-INF/applicationContext*.xml",
            "/WEB-INF/action-servlet.xml"};
        ctx = new XmlWebApplicationContext();
        ctx.setConfigLocations(paths);
        ctx.setServletContext(new MockServletContext(""));
        ctx.refresh();
    }

    public void testGetUsers() throws Exception {
        UserController c =
            (UserController) ctx.getBean("userController");
        ModelAndView mav =
            c.handleRequest((HttpServletRequest) null,
                (HttpServletResponse) null);
        Map m = mav.getModel();
        assertNotNull(m.get("users"));
        assertEquals(mav.getViewName(), "userList");
    }
}
```

这个类用了两个Spring `mock: MockHttpServletRequest` [http://www.springframework.org/docs/api/org.springframework.mock.web.MockHttpServletRequest.html] 和 `MockServletContext` [http://www.springframework.org/docs/api/org.springframework.mock.web.MockServletContext.html]。它加载了context文件，并在从初始化的XmlWebApplicationContext [http://www.springframework.org/docs/api/org.springframework.web.context.support.XmlWebApplicationContext.html] 中获取bean。运行正常，但它超出了集成测试的范围，更不用说单元测试了。这是因为Spring绑定了UserController和它所有的依赖对象。为了创建一个单元测试，你必须删除所有ApplicationContext的跟踪信息，而在单元测试中手动的设置依赖关系。

要用EasyMock重构这个类，在目录中test/org/appfuse/web中创建一个类UserControllerEMTest.java。这个类继承了JUnit的TestCase类。

```
package org.appfuse.web;
// use your IDE to organize imports
```

```

public class UserControllerEMTest extends TestCase {
    private MockControl control = null;
    private UserManager mockManager = null;
    private UserController c = new UserController();

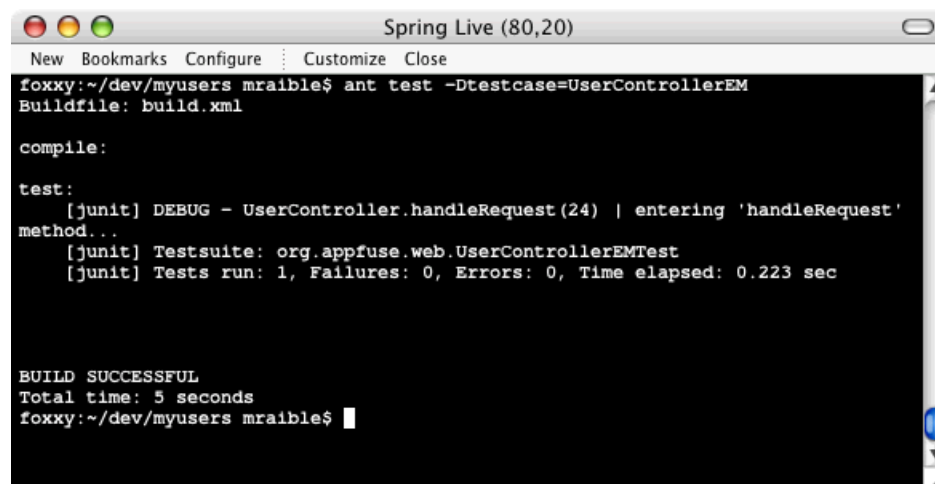
    protected void setUp() throws Exception {
        control = MockControl.createControl(UserManager.class);
        mockManager = (UserManager) control.getMock();
        c.setUserManager(mockManager);
    }

    public void testGetUsers() throws Exception {
        // set expected behavior on manager
        mockManager.getUsers();
        control.setReturnValue(new ArrayList());
        // switch from record to playback
        control.replay();
        ModelAndView mav =
            c.handleRequest((HttpServletRequest) null,
                (HttpServletResponse) null);
        Map m = mav.getModel();
        assertNotNull(m.get("users"));
        assertEquals(mav.getViewName(), "userList");
        // verify getUsers() was called on manager
        control.verify();
    }
}

```

要运行这个测试，执行`ant test -Dtestcase=UserControllerEM`。你控制台的输出内容应该下面的截图看起来差不多。

图 8.4. 运行测试`ant test -Dtestcase=UserControllerEM`的输出结果



```

Spring Live (80,20)
New Bookmarks Configure Customize Close
foxy:~/dev/myusers mraible$ ant test -Dtestcase=UserControllerEM
Buildfile: build.xml

compile:

test:
[junit] DEBUG - UserController.handleRequest(24) | entering 'handleRequest'
method...
[junit] Testsuite: org.appfuse.web.UserControllerEMTest
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.223 sec

BUILD SUCCESSFUL
Total time: 5 seconds
foxy:~/dev/myusers mraible$

```

这个测试比前面那一个快了很多。为了进行对比，运行`ant test -Dtestcase=UserControllerTest`，比较“Time elapsed”值。在我的1.33Mhz/512RAM PowerBook上，差别就很明显，模拟测试花了0.223秒，非模拟测试花了6.455秒。

图 8.5. 非模拟测试的时间消耗

```

Spring Live (80,20)
New Bookmarks Configure Customize Close
foxy:~/dev/myusers mraible$ ant test -Dtestcase=UserControllerEM
Buildfile: build.xml

compile:

test:
[junit] DEBUG - UserController.handleRequest(24) | entering 'handleRequest'
method...
[junit] Testsuite: org.appfuse.web.UserControllerEMTest
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.223 sec

BUILD SUCCESSFUL
Total time: 5 seconds
foxy:~/dev/myusers mraible$

```

你已经创建了UserControllerTest的EasyMock版本，现在创建jMock版本。在目录test/org/appfuse/web下创建一个新的UserControllerJMTTest.java类。这个类继承了jMock的MockObjectTestCase。

```

package org.appfuse.web;
// use your IDE to organize imports

public class UserControllerJMTTest extends MockObjectTestCase {
    private UserController c = new UserController();
    private Mock mockManager = null;

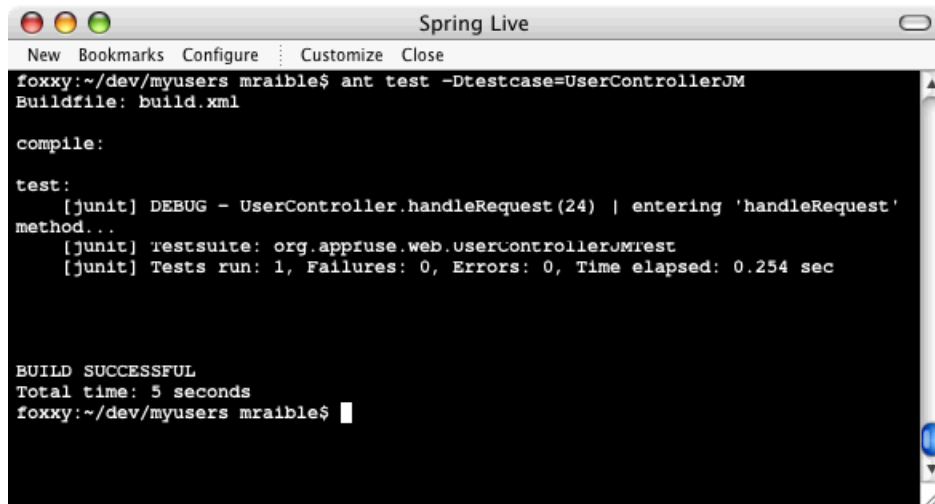
    protected void setUp() throws Exception {
        mockManager = new Mock(UserManager.class);
        c.setUserManager((UserManager) mockManager.proxy());
    }

    public void testGetUsers() throws Exception {
        // set expected behavior on manager
        mockManager.expects(once()).method("getUsers")
            .will(returnValue(new ArrayList()));
        ModelAndView mav =
            c.handleRequest((HttpServletRequest) null,
                (HttpServletResponse) null);
        Map m = mav.getModel();
        assertNotNull(m.get("users"));
        assertEquals(mav.getViewName(), "userList");
        // verify expectations
        mockManager.verify();
    }
}

```

执行ant test -Dtestcase=UserControllerJM运行测试。

图 8.6. 运行测试ant test -Dtestcase=UserControllerJM的输出结果



测试FormController

测试 form 类 (继承了 SimpleFormController [http://www.springframework.org/docs/api/org.springframework.web.servlet.mvc.SimpleFormController.html]) 与 测试 Controller [http://www.springframework.org/docs/api/org.springframework.web.servlet.mvc.Controller.html]) 的实现类非常相似。主要区别在于 FormController 要完成更多的操作。例如, 验证数据, 设置信息 (messages), 使用 request 参数。用于 UserFormController (来自第4章) 的测试代码如下。它验证 testSave() 方法中没有出现验证错误, 并设置了一个成功提示信息。

```

package org.appfuse.web;
// use your IDE to organize imports

public class UserFormControllerTest extends TestCase {
    private static Log log =
        LogFactory.getLog(UserFormControllerTest.class);
    private XmlWebApplicationContext ctx = null;
    private UserFormController c = null;
    private MockHttpServletRequest request = null;
    private ModelAndView mv = null;
    private User user = null;

    protected void setUp() throws Exception {
        super.setUp();
        String[] paths = {"/WEB-INF/applicationContext*.xml",
            "/WEB-INF/action-servlet.xml"};
        ctx = new XmlWebApplicationContext();
        ctx.setConfigLocations(paths);
        ctx.setServletContext(new MockServletContext(""));
        ctx.refresh();
        c = (UserFormController) ctx.getBean("userFormController");
        // add a test user to the database
        UserManager mgr = (UserManager) ctx.getBean("userManager");
        user = new User();
    }
}

```

```
        user.setFirstName("Matt");
        user.setLastName("Raible");
        user = mgr.saveUser(user);
    }

    public void testEdit() throws Exception {
        log.debug("testing edit...");
        request = new MockHttpServletRequest("GET", "/editUser.html");
        request.addParameter("id", user.getId().toString());
        mv = c.handleRequest(request, new MockHttpServletResponse());
        assertEquals("userForm", mv.getViewName());
    }

    public void testSave() throws Exception {
        request = new MockHttpServletRequest("POST", "/editUser.html");
        request.addParameter("id", user.getId().toString());
        request.addParameter("firstName", user.getFirstName());
        request.addParameter("lastName", "Updated Last Name");
        mv = c.handleRequest(request, new MockHttpServletResponse());
        Errors errors =
            (Errors) mv.getModel().get(BindException.ERROR_KEY_PREFIX + "user");
        assertNull(errors);
        assertNotNull(request.getSession().getAttribute("message"));
    }

    public void testRemove() throws Exception {
        request = new MockHttpServletRequest("POST", "/
        editUser.html");
        request.addParameter("delete", "");
        request.addParameter("id", user.getId().toString());
        mv = c.handleRequest(request, new MockHttpServletResponse());
        assertNotNull(request.getSession().getAttribute("message"));
    }
}
```

虽然可以运行，但在运行却花了好几秒时间(在我的Mac OS X10.2 PowerBook上)。现在进行重构，删除Spring依赖关系，使用jMock来迅速提高速度。

要重构这个类，在目录test/org/appfuse/web下创建一个新的类UserFormControllerJMTest.java。这个类继承了MockObjectTestCase，代码如下。

```
package org.appfuse.web;
// use your IDE to organize imports

public class UserFormControllerJMTest extends MockObjectTestCase {
    private static Log log =
        LogFactory.getLog(UserFormControllerJMTest.class);
    private UserFormController c = new UserFormController();
    private MockHttpServletRequest request = null;
    private ModelAndView mv = null;
    private User user = new User();
    private Mock mockManager = null;
```



```
protected void setUp() throws Exception {
    super.setUp();
    mockManager = new Mock(UserManager.class);
    // manually set properties (dependencies) on userFormController
    c.setUserManager((UserManager) mockManager.proxy());
    c.setFormView("userForm");
    // set context with messages avoid NPE when controller calls
    // getMessageSourceAccessor().getMessage()
    StaticApplicationContext ctx = new StaticApplicationContext();
    Map properties = new HashMap();
    properties.put("basename", "messages");
    ctx.registerSingleton("messageSource",
        ResourceBundleMessageSource.class,
        new MutablePropertyValues(properties));
    ctx.refresh();
    c.setApplicationContext(ctx);
    // setup user values
    user.setId(new Long(1));
    user.setFirstName("Matt");
    user.setLastName("Raible");
}

public void testEdit() throws Exception {
    log.debug("testing edit...");
    // set expected behavior on manager
    mockManager.expects(once()).method("getUser")
        .will(returnValue(new User()));
    request = new MockHttpServletRequest("GET", "/editUser.html");
    request.addParameter("id", user.getId().toString());
    mv = c.handleRequest(request, new MockHttpServletResponse());
    assertEquals("userForm", mv.getViewName());
    // The getCommandName() method is available in Spring 1.1.1+
    User editUser = (User) mv.getModel().get(c.getCommandName());
    assertEquals(editUser.getFullName(), "Matt Raible");
    // verify expectations
    mockManager.verify();
}

public void testSave() throws Exception {
    // set expected behavior on manager
    // called by formBackingObject()
    mockManager.expects(once()).method("getUser")
        .will(returnValue(user));
    User savedUser = user;
    savedUser.setLastName("Updated Last Name");
    // called by onSubmit()
    mockManager.expects(once()).method("saveUser")
        .with(eq(savedUser));
    request = new MockHttpServletRequest("POST", "/editUser.html");
    request.addParameter("id", user.getId().toString());
    request.addParameter("firstName", user.getFirstName());
    request.addParameter("lastName", "Updated Last Name");
    mv = c.handleRequest(request, new MockHttpServletResponse());
}
```

```

Errors errors =
    (Errors) mv.getModel().get(BindException.ERROR_KEY_PREFIX + "user");
assertNull(errors);
assertNotNull(request.getSession().getAttribute("message"));
// verify expectations
mockManager.verify();
}

public void testRemove() throws Exception {
    // set expected behavior on manager
    // called by formBackingObject()
    mockManager.expects(once()).method("getUser")
        .will(returnValue(user));
    // called by onSubmit()
    mockManager.expects(once()).method("removeUser").with(eq("1"));
    request = new MockHttpServletRequest("POST", "/editUser.html");
    request.addParameter("delete", "");
    request.addParameter("id", user.getId().toString());
    mv = c.handleRequest(request, new MockHttpServletResponse());
    assertNotNull(request.getSession().getAttribute("message"));
    // verify expectations
    mockManager.verify();
}
}

```

在上面类的 `setUp()` 方法中，Spring 的 `StaticApplicationContext` [<http://www.springframework.org/docs/api/org.springframework.context.support.StaticApplicationContext.html>] 提供一种方便的途径在单元测试中添加和使用bean。它的 `registerSingleton()` 可以注册 `messages.properties`，提供信息访问。如果不这样做，在 `FormController` 试图访问 `messageSource` bean 时会抛出一个异常 `NullPointerException`。你也可以使用 `StaticMessageSource` [<http://www.springframework.org/docs/api/org.springframework.context.support.StaticMessageSource.html>] 在测试中通过编程的方式添加消息。有关更多在测试如何使用这个类资料，可参考Spring的 `StaticApplicationContextTestSuite` [<http://monkeymachine.co.uk/spring/xref-test/org.springframework.context.support.StaticApplicationContextTestSuite.html>]。

注意

Spring的内部测试 [<http://monkeymachine.co.uk/spring/xref-test/index.html>] 是一个不可多得的有关如何编写自己的单元测试方面资料源。

Cactus

本节使用Cactus在部署环境中测试Controller。Cactus [<http://jakarta.apache.org/cactus/>] 是JUnit的一个扩展，用于测试服务器端的Java代码。

要用Cactus测试Controller，修改MyUsers的build.xml文件，添加一个test-cactus target。

```

<target name="test-cactus" depends="war"
    description="Runs Cactus tests in-container">
    <!-- Define Cactus Tasks -->
    <taskdef resource="cactus.tasks" classpathref="classpath"/>

```

```

<cactifywar srcfile="${dist.dir}/${webapp.name}.war"
  destfile="${dist.dir}/${webapp.name}-cactus.war">
  <classes dir="${test.dir}/classes"/>
  <classes dir="test" includes="cactus.properties"/>
  <!-- If you use EasyMock or Spring Mocks in a Cactus test
  it needs to be included in the WAR -->
  <lib dir="web/WEB-INF/lib" includes="*mock.jar"/>
  <servletredirector/>
</cactifywar>
<mkdir dir="${test.dir}/data/tomcat"/>
<cactus warfile="${dist.dir}/${webapp.name}-cactus.war"
  printsummary="yes" failureproperty="tests.failed">
<classpath>
  <path refid="classpath"/>
  <path location="${build.dir}/classes"/>
  <path location="${test.dir}/classes"/>
</classpath>
<containerset>
  <tomcat5x dir="${tomcat.home}" if="tomcat.home"
    port="8080" todir="${test.dir}/data/tomcat"/>
</containerset>
<formatter type="xml"/>
<formatter type="brief" usefile="false"/>
<batchtest todir="${test.dir}/data" if="testcase">
  <fileset dir="${test.dir}/classes">
    <include name="**/*${testcase}*" />
    <exclude name="**/*TestCase.class" />
  </fileset>
</batchtest>
<batchtest todir="${test.dir}/data" unless="testcase">
  <fileset dir="${test.dir}/classes">
    <include name="**/*Cactus*Test.class*" />
  </fileset>
</batchtest>
</cactus>
<fail if="tests.failed">Cactus test(s) failed.</fail>
</target>

```

上面清单中<cactifywar> task负责修改web.xml和WAR文件，这样Cactus才能对它进行测试。这个target中间有一个<tomcat5x> task。这个task指定了哪个容器用于测试。参见Cactus项目站点上Ant集成的更多信息 [http://jakarta.apache.org/cactus/integration/ant/index.html]，以及<cactus>所支持的容器清单。

现在你以配置好了Cactus，在目录test/org/appfuse/web中创建一个类UserCactusTest.java，这个类继承了ServletTestCase [http://jakarta.apache.org/cactus/api/framework-13/org/apache/cactus/ServletTestCase.html]。第一个版本手动的创建UserController和UserFormController，设置必要的依赖关系(UserManager和用于messageSource的context)。这个测试中的ApplicationContext是从ServletContext中得到的，而不是由ClassPathXmlApplicationContext初始化而来的。这是因为这个测试将会运行在应用程序的context中，Spring的application context是由web.xml的ContextLoaderListener

[<http://www.springframework.org/docs/api/org.springframework.web.context.ContextLoaderListener.html>] 进行初始化。

```
package org.appfuse.web;
// use your IDE to organize imports

public class UserCactusTest extends ServletTestCase {
    private static Log log = LogFactory.getLog(UserCactusTest.class);
    private UserController list = new UserController();
    private UserFormController form = new UserFormController();

    protected void setUp() throws Exception {
        super.setUp();
        ApplicationContext ctx =
            WebApplicationContextUtils
                .getRequiredWebApplicationContext(
                    session.getServletContext());
        UserManager userManager =
            (UserManager) ctx.getBean("userManager");
        list.setUserManager(userManager);
        form.setUserManager(userManager);
        // needed to prevent NPE with getMessageSourceAccessor()
        form.setApplicationContext(ctx);
    }

    public void beginAddUser(WebRequest wRequest) {
        wRequest.addParameter("firstName", "Dion", "post");
        wRequest.addParameter("lastName", "Almaer", "post");
    }

    public void testAddUser() throws Exception {
        form.handleRequest(request, response);
        assertTrue(request.getSession().getAttribute("message") != null);
    }

    public void testUserList() throws Exception {
        ModelAndView mav = list.handleRequest(request, response);
        Map m = mav.getModel();
        assertNotNull(m.get("users"));
        assertEquals(mav.getViewName(), "userList");
    }
}
```

在test目录下创建文件cactus.properties, Cactus需要下面的信息运行它的servlet测试。

```
# Web app Context under which our application to test runs
cactus.contextURL=http://localhost:8080/myusers-cactus
# Default Servlet Redirector Name. Used by ServletTestCase test cases.
cactus.servletRedirectorName=ServletRedirector
```

执行`ant test-cactus -Dtestcase=UserCactusTest`来运行这个测试。

编写这个测试的另一种途径是，直接从`ApplicationContext`中获取`Controllers`，它们的依赖关系已经在其中设置好了。

```
...
public class UserCactusTest extends ServletTestCase {
    private static Log log = LogFactory.getLog(UserCactusTest.class);
    private UserController list = null;
    private UserFormController form = null;

    protected void setUp() throws Exception {
        super.setUp();
        ApplicationContext ctx =
            WebApplicationContextUtils
                .getRequiredWebApplicationContext(
                    session.getServletContext());
        list = (UserController) ctx.getBean("userController");
        form = (UserFormController) ctx.getBean("userFormController");
    }

    public void beginAddUser(WebRequest wRequest) {
        ...
    }
}
```

最后，如果你愿意，你可以用`EasyMock`来模拟`UserManager`。下面的`UserCactusEMTest`类向你演示了这种技巧。你不能在`Cactus`中使用`jMock`，因为这两者都要求你必须继承它们的`TestCase`类。

```
package org.appfuse.web;
// user your IDE to organize imports

public class UserCactusEMTest extends ServletTestCase {
    private UserController list = new UserController();
    private UserFormController form = new UserFormController();
    private MockControl control = null;
    private UserManager mockManager = null;

    protected void setUp() throws Exception {
        control = MockControl.createControl(UserManager.class);
        mockManager = (UserManager) control.getMock();
        list.setUserManager(mockManager);
        form.setUserManager(mockManager);
        // needed to prevent NPE with getMessageSourceAccessor()
        ApplicationContext ctx =
            WebApplicationContextUtils
                .getRequiredWebApplicationContext(
                    session.getServletContext());
        form.setApplicationContext(ctx);
    }

    public void beginAddUser(WebRequest wRequest) {
    }
}
```

```

        wRequest.addParameter("firstName", "Dion", "post");
        wRequest.addParameter("lastName", "Almaer", "post");
    }

    public void testAddUser() throws Exception {
        // set expected behavior on manager
        User user = new User();
        user.setFirstName("Dion");
        user.setLastName("Almaer");
        mockManager.saveUser(user);
        control.setReturnValue(user);
        // switch from record to playback
        control.replay();
        form.handleRequest(request, response);
        // verify saveUser() was called
        control.verify();
        assertTrue(request.getSession()
            .getAttribute("message") != null);
    }

    public void testUserList() throws Exception {
        // set expected behavior on manager
        control.expectAndReturn(mockManager.getUsers(),
            new ArrayList());
        // switch from record to playback
        control.replay();
        ModelAndView mav = list.handleRequest(request, response);
        // verify getUsers() was called
        control.verify();
        Map m = mav.getModel();
        assertNotNull(m.get("users"));
        assertEquals(mav.getViewName(), "userList");
    }
}

```

执行 `ant test-cactus -Dtestcase=UserCactusEM`, 运行测试。

第二种方法(直接获取controller)通常是最好的方法。运行Cactus测试要费时一些, 因为它会分析/修改WAR文件, 启动/停止容器。所以, 使用mock未必能够提高你的测试效率。然而, 在你的容器中进行孤立的测试仍然非常有用的。

测试文件上传和Email

正如第五章中承诺的那样, 现在你将学习如何测试FileUploadController类。这个负责上传文件, 发送通知邮件。

1. 在目录test/org/appfuse/web下, 创建一个类FileUploadControllerTest, 继承了JUnit的TestCase。这个类的内容如下, 用一个setUp()方法来初始化context文件。

```

package org.appfuse.web;
// organize imports using your IDE

```

```

public class FileUploadControllerTest extends TestCase {
    private static Log log =
        LoggerFactory.getLog(FileUploadControllerTest.class);
    private XmlWebApplicationContext ctx = null;
    private FileUploadController fileUpload = null;

    public void setUp() {
        String[] paths = {"/WEB-INF/applicationContext*.xml",
            "/WEB-INF/action-servlet.xml"};
        ctx = new XmlWebApplicationContext();
        ctx.setConfigLocations(paths);
        ServletContext servletContext =
            new MockServletContext("file:web");
        ctx.setServletContext(servletContext);
        ctx.refresh();
        fileUpload = (FileUploadController)
            ctx.getBean("fileUploadController");
        // Create a mailSender that'll use Dumbster's ports
        JavaMailSenderImpl mailSender = new JavaMailSenderImpl();
        mailSender.setHost("localhost");
        mailSender.setPort(2525);
        fileUpload.setMailSender(mailSender);
    }
    // continued below

```

setUp() 包含唯一的配置。首先，MockServletContext 利用 file:web 进行初始化。
R e s o u r c e L o a d e r
<http://www.springframework.org/docs/api/org.springframework.core.io.ResourceLoader.html> 使用这个值来判定 web 应用程序的根目录。如果不这么做，FileUploadController 中下面的参数分配会导致失败。

```

String uploadDir =
    getServletContext().getRealPath("/upload/");

```

第二，setUp() 方法中包含一个自定义的 MailSender [\[http://www.springframework.org/docs/api/org.springframework.mail.MailSender.html\]](http://www.springframework.org/docs/api/org.springframework.mail.MailSender.html)，通过一个非标准的端口(2525，25是标准)发送邮件。这就是你要用 Dumbster [\[http://www.quintanasoft.com/dumbster/\]](http://www.quintanasoft.com/dumbster/) 原因所在。你可以在运行测试 FileUploadControllerTest 前后，使用 Dumbster 来启动/停止 2525 端口的 SMTP 服务器。
注意

本例中使用 2525 是为了避免与可能正在 25 端口运行的服务冲突。

2. 添加以下 testUpload() 方法完成测试。

```

public void testUpload() throws Exception {
    log.debug("testing upload...");
    MockHttpServletRequest request =
        new MockHttpServletRequest("POST", "/fileUpload.html");

```

```
MockCommonsMultipartResolver resolver =
    new MockCommonsMultipartResolver();
ctx.getServletContext();
request.setContentType("multipart/form-data");
request.addHeader("Content-type", "multipart/form-data");
assertTrue(resolver.isMultipart(request));
MultipartHttpServletRequest multipartRequest =
    resolver.resolveMultipart(request);

// setup a simple mail server using Dumbster
SimpleSmtpServer server = SimpleSmtpServer.start(2525);
ModelAndView mav =
    fileUpload.handleRequest(multipartRequest,
        new MockHttpServletRequest());
server.stop();

// the getReceieved() method is spelled wrong in the API. ;-)
assertEquals(1, server.getReceievedEmailSize());
log.debug("model: " + mav.getModel());
assertNotNull(request.getSession().getAttribute("message"));

// ensure the file got uploaded
Resource uploadedFile =
    ctx.getResource("file:web/upload/test.xml");
assertTrue(uploadedFile.exists());

// delete the upload directory
Resource uploadDir = ctx.getResource("file:web/upload");
uploadedFile.getFile().delete();
uploadDir.getFile().delete();
assertFalse(uploadDir.exists());
}

public static class MockCommonsMultipartResolver extends
    CommonsMultipartResolver {
    private boolean empty;

    protected void setEmpty(boolean empty) {
        this.empty = empty;
    }

    protected DiskFileUpload newFileUpload() {
        return new DiskFileUpload() {
            public List parseRequest(HttpServletRequest request) {
                if (request instanceof MultipartHttpServletRequest) {
                    throw new IllegalStateException(
                        "Already a multipart request");
                }
                List fileItems = new ArrayList();
                MockFileItem fileItem = new MockFileItem(
                    "file", "text/html", empty ? "" :
                    "test.xml", empty ? "" : "<root/>");
                fileItems.add(fileItem);
            }
        };
    }
}
```



```

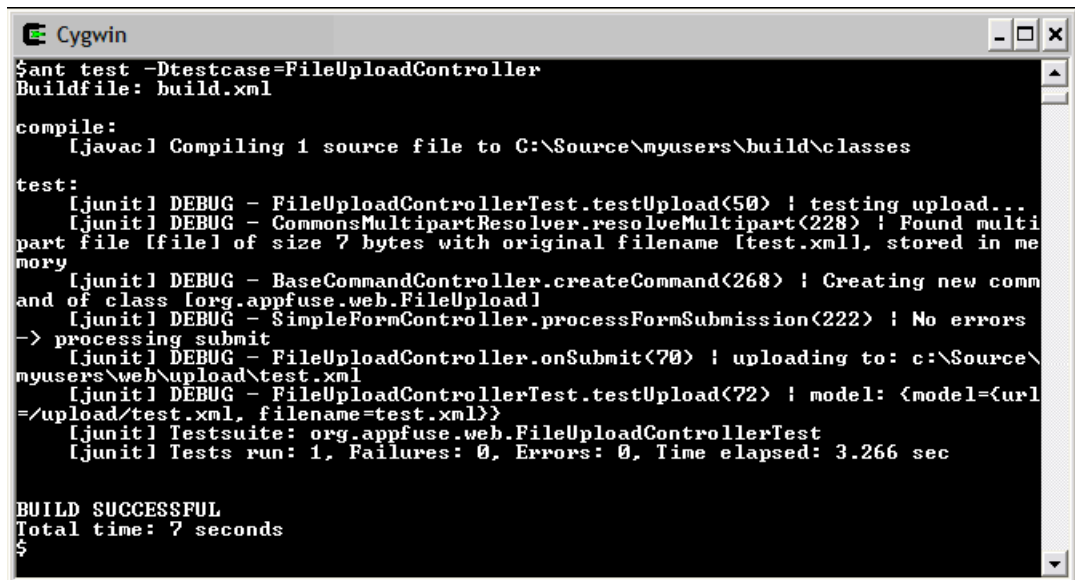
        return fileItems;
    }
    };
}
}
}

```

这个测试中MockCommonsMultipartResolver引用了MockFileItem。这个类模拟了一个待上传的文件。MockFileItem.java文件已经包含在本章的下载包中。Spring的CommonsMultipartResolverTests [http://monkeymachinecook/spring/servlet/org/springframework/web/multipart/commons/CommonsMultipartResolverTests.html#343]中也提供了这个类。你也可以像Spring中那样，写一个内部类来实现它。本例中将它提取出来，以节省空间。

3. 执行ant test -Dtestcase=FileUploadController来运行测试。测试的输出结果应该和下面的截图差不多。

图 8.7. 运行测试ant test -Dtestcase=FileUploadController的输出结果



```

Cygwin
$ant test -Dtestcase=FileUploadController
Buildfile: build.xml

compile:
[javac] Compiling 1 source file to C:\Source\myusers\build\classes

test:
[junit] DEBUG - FileUploadControllerTest.testUpload(50) : testing upload...
[junit] DEBUG - CommonsMultipartResolver.resolveMultipart(228) : Found multi
part file [file] of size 7 bytes with original filename [test.xml], stored in me
mory
[junit] DEBUG - BaseCommandController.createCommand(268) : Creating new comm
and of class [org.appfuse.web.FileUpload]
[junit] DEBUG - SimpleFormController.processFormSubmission(222) : No errors
-> processing submit
[junit] DEBUG - FileUploadController.onSubmit(70) : uploading to: c:\Source\
myusers\web\upload\test.xml
[junit] DEBUG - FileUploadControllerTest.testUpload(72) : model: {model={url
=/upload/test.xml, filename=test.xml}}
[junit] Test suite: org.appfuse.web.FileUploadControllerTest
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 3.266 sec

BUILD SUCCESSFUL
Total time: 7 seconds
$

```

从本节中提供的例子中可以看到，测试Controller有很多方法可供选择。最简单的方法是在你的测试类中初始化一个context，从中提取一个Controller(使用ctx.getBean("controllerName")), 在上面执行一些方法。这种技巧允许Spring来装配Controller的依赖关系。使用mock多出一些额外的任务，因为你必须弄清Controller的依赖关系。你还要对你所模拟的对象调用哪些方法了如指掌。优势在于，你确切的掌握了你的Controller依赖什么，它是如何与他们的依赖进行交互的。

测试View

在第六章，你创建了一个jWebUnit测试(UserWebTest)，通过链接和按钮操作来验证UI的功能。jWebUnit是一种优雅的测试view的框架，因为它允许你在Java代码中测试view。本节会修改build.xml文件中的test-web target，以便在任何jWebUnit测试运行之前，启动和停止Tomcat。你将更改当前测试来切换locale，验证messages.properties定义的文本信息，而不是直接在

类文件中用硬编码的方式。最后你会使用Canoo WebTest作为测试UI的一种替代方式。Canoo WebTest对于非程序员来说显得更为友好，因为测试是以XML的方式写在一个Ant build文件中。

本节中的测试通常是指集成测试。通过用户在浏览器中来操作UI方式来验证所有的层是否集成正确。jWebUnit和Canoo WebTest都支持JavaScript测试。这就意味着，如果你的某一个按钮上存在一个“onclick”操作，你点击了这个按钮，JavaScript就会执行。但是这种JavaScript支持有点古怪，而且在没有错误时JavaScript中会报错。基于这些原因，我建议1)UI开发尽量不要使用JavaScript2)针对任何采用了JavaScript的UI编写测试。这就是说，如果某个URI通过addPage()函数调用了<button onclick="addPage()"/>，它也可以在测试中被调用。

jWebUnit

在修改MyUsers中的UserWebTest之前，回顾一下它的源代码。构造器总会包含一个调用setBaseUrl()。因为jWebUnit通过名称来定位提交按钮，你必须在你的按钮上指定一个name属性。如果你想测试表格及其数据，你必须指定表格的id属性。MyUsers应该已经包含所有必要的name和id属性。

```
package org.appfuse.web;
// use your IDE to organize imports

public class UserWebTest extends WebTestCase {

    public UserWebTest(String name) {
        super(name);
        getTestContext().setBaseUrl("http://localhost:8080/myusers");
    }

    public void testWelcomePage() {
        beginAt("/");
        assertEquals("MyUsers ~ Welcome");
    }

    public void testAddUser() {
        beginAt("/editUser.html");
        assertEquals("MyUsers ~ User Details");
        setFormElement("firstName", "Spring");
        setFormElement("lastName", "User");
        submit("save");
        assertTextPresent("saved successfully");
    }

    public void testListUsers() {
        beginAt("/users.html");
        // check that table is present
        assertTablePresent("userList");
        //check that a set of strings are present somewhere in table
        assertTextInTable("userList",
            new String[] {"Spring", "User"});
    }

    public void testEditUser() {
        beginAt("/editUser.html?id=" + getInsertedUserId());
        assertFormElementEquals("firstName", "Spring");
    }
}
```

```
        submit("save");
        assertEquals("MyUsers ~ User List");
    }

    public void testDeleteUser() {
        beginAt("/editUser.html?id=" + getInsertedUserId());
        assertEquals("MyUsers ~ User Details");
        submit("delete");
        assertEquals("MyUsers ~ User List");
    }

    /**
     * Convenience method to get the id of the inserted user
     * Assumes last inserted user is "Spring User"
     */
    public String getInsertedUserId() {
        String[] paths = {"/WEB-INF/applicationContext*.xml"};
        ApplicationContext ctx =
            new ClassPathXmlApplicationContext(paths);
        List users = ((UserDAO) ctx.getBean("userDAO")).getUsers();
        // assumed that user inserted in testAddUser() is last user
        return "" + ((User) users.get(users.size() - 1)).getId();
    }
}
```

为了将这个测试国际化，首先你得把MyUsers的标题国际化。目前，它是以硬编码的方式放在相应的模板中。

1. 在文件messages.properties新建一些条目：

```
# Page titles
index.title=MyUsers ~ Welcome
userList.title=MyUsers ~ User List
userForm.title=MyUsers ~ User Details
```

2. 本章下载文件的view部分使用Velocity模板。所有要编辑Velocity模板。

- web/WEB-INF/velocity/userList.vm:

```
<title>${rc.getMessage("userList.title")}</title>
```

- web/WEB-INF/velocity/userForm.vm:

```
<title>${rc.getMessage("userForm.title")}</title>
```

- web/index.vm:

```
<title>${rc.getMessage("index.title")}</title>
```

3. 欢迎页面(index.vm)需要一些额外的工作来解析rc变量。在本章的下载文件中, index.vm位于web目录下, 通过web.xml的welcome-file来配置的。要将messages.properties文件暴露给SiteMesh Servlet [http://www.opensymphony.com/sitemesh/api/com/opensymphony/module/sitemesh/velocity/VelocityDecoratorServlet.html] 并不容易。但是, 如果结合剩下的模板, 使用Paul Tuckey的UrlRewriteFilter [http://www.tuckey.org/urlrewrite/]将它解析成欢迎页面, 你依然可以展开这项工作。下面是修改所必须的步骤。

1. 将index.vm移到web/WEB-INF/velocity目录中。
2. 在文件web/WEB-INF/actionservlet.xml中添加将/index.html作为一个URL指定给urlMapper。

```
<prop key="/index.html">filenameController</prop>
```

3. 将它作为欢迎页面, 使用UrlRewriteFilter。在web/WEB-INF/web.xml添加以下filter进行配置。

```
<filter>
  <filter-name>rewriteFilter</filter-name>
  <filter-class>
    org.tuckey.web.filters.urlrewrite.UrlRewriteFilter
  </filter-class>
</filter>
```

4. 紧跟sitemesh, 为这个filter添加一个mapping。

```
<filter-mapping>
  <filter-name>rewriteFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

5. 将web/WEB-INF目录下urlrewrite.xml文件内容修改成如下:

```
<urlrewrite>
  <rule>
    <from>/$</from>
    <to type="forward">index.html</to>
```

```
</rule>
<rule>
  <from>/index.vm</from>
  <to type="forward">index.html</to>
</rule>
</urlrewrite>
```

注意

对于FreeMarker模板可以如法炮制。对于JSP页面，<title>中间可以使用JSTL的<fmt:message>标签。

现在启动Tomcat，运行`ant deploy test-web`验证所有修改是否已经奏效。证明一切正常后，修改这个类利用jWebUnit的国际化功能。也就是，它能设置一个ResourceBundle，验证基于键名的标题和文本，而不是文本值。有关此特性及其相关的信息请参考jWebUnit's QuickStart Guide [<http://jwebunit.sourceforge.net/quickstart.html>]。

下面这个类与前面未进行国际化的类的主要差别是testAddUser()方法验证是结果页面的标题，而不是成功消息文本。这是因为成功消息使用新用户的名称作为最终输出，那么，键值与所显示的 值 完 全 不 同 。 这 里 有 一 个 j W e b U n i t 补 丁 [https://sourceforge.net/tracker/?func=detail&atid=497984&aid=993058&group_id=61302]允许使用动态值测试键。

修改过的代码以下划线标出，前面的断言被注释掉了。

```
package org.appfuse.web;
import java.util.List;
import java.util.Locale;
import net.sourceforge.jwebunit.WebTestCase;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.appfuse.dao.UserDAO;
import org.appfuse.model.User;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class UserWebTest extends WebTestCase {

    public UserWebTest(String name) {
        super(name);
        getTestContext().setBaseUrl("http://localhost:8080/myusers");
        getTestContext().setResourceBundleName("messages");
        getTestContext().setLocale(Locale.ENGLISH);
    }

    public void testWelcomePage() {
        beginAt("/");
        //assertTitleEquals("MyUsers ~ Welcome");
        assertTitleEqualsKey("index.title");
    }
}
```

```
public void testAddUser() {
    beginAt("/editUser.html");
    assertEquals("MyUsers ~ User Details");
    setFormElement("firstName", "Spring");
    setFormElement("lastName", "User");
    submit("save");
    //assertTextPresent("saved successfully");
    assertEqualsKey("userList.title");
}

public void testListUsers() {
    beginAt("/users.html");
    // check that table is present
    assertTablePresent("userList");
    //check that a set of strings are present somewhere in table
    assertTextInTable("userList",
        new String[] {"Spring", "User"});
}

public void testEditUser() {
    beginAt("/editUser.html?id=" + getInsertedUserId());
    assertFormElementEquals("firstName", "Spring");
    submit("save");
    //assertEquals("MyUsers ~ User List");
    assertEqualsKey("userList.title");
}

public void testDeleteUser() {
    beginAt("/editUser.html?id=" + getInsertedUserId());
    //assertEquals("MyUsers ~ User Details");
    assertEqualsKey("userForm.title");
    submit("delete");
    //assertEquals("MyUsers ~ User List");
    assertEqualsKey("userList.title");
}

/**
 * Convenience method to get the id of the inserted user
 * Assumes last inserted user is "Spring User"
 */
public String getInsertedUserId() {
    String[] paths = {"WEB-INF/applicationContext*.xml"};
    ApplicationContext ctx =
        new ClassPathXmlApplicationContext(paths);
    List users = ((UserDAO) ctx.getBean("userDAO")).getUsers();
    // assumed that user inserted in testAddUser() is last user
    return "" + ((User) users.get(users.size() - 1)).getId();
}
}
```

要以一个不同的Locale(和messages.properties文件)测试这个类, 按下面的步骤进行:

1. 复制一份messages.properties，命名为message_de.properties，修改一些标题值，这样你就能发现它使用了新的消息绑定。
2. 修改构造器中的setLocale()调用，将locale设置成German(德国)。

```
getTestContext().setLocale(Locale.GERMAN);
```

这样会加载德语消息文件，但它不会把request设置成此locale。这是一个bug，已经提交给jWebUnit [https://sourceforge.net/tracker/?func=detail&atid=497982&aid=693598&group_id=61302]。为了避开这个bug，设置Accept-Language文件头，作为HttpUnit WebClient的一部分。

```
getTestContext().getWebClient().  
    setHeaderField("Accept-Language", "de");
```

3. 使用ant deploy reload test-web，运行测试。

自动启动Tomcat

目前用于jWebUnit的测试的设置要求在测试运行之前启动Tomcat。为了减轻痛苦，在文件build.xml中添加一个新的target，在运行测试之前和之后自动启动和关闭Tomcat。Cargo [<http://cargo.codehaus.org/>]是由Vincent Massol(他还创建了Cactus)启动的一个开源项目。它提供发Java API和Ant task来启动/停止Tomcat和配置Java容器。本章下载文件中已经包含了Cargo jar文件。你只要将下面的target添加到build.xml。

```
<target name="test-tomcat" depends="war"  
    description="Runs jWebUnit tests in a running server">  
    <taskdef resource="cargo.tasks" classpathref="classpath"/>  
    <cargo-tomcat5x homeDir="${tomcat.home}"  
        output="${test.dir}/cargo.log"  
        workingDir="${test.dir}/tomcat5x" action="start">  
        <war warFile="${dist.dir}/${webapp.name}.war"/>  
    </cargo-tomcat5x>  
    <antcall target="test-web"/>  
</target>
```

现在你应该可以停止Tomcat，运行ant test-tomcat在Tomcat运行UserWebTest，而不需要手动启动Tomcat。

注意

Cargo是一个非常年轻的项目，MyUsers应用程序中使用的jar文件是2004年9月初发布的一个alpha版本。一些小功能需要增强，如允许启动/关闭日志消息打印到控制台。目前只能导出到由output属性指定的文件中。

Canoo WebTest

Canoo WebTest [<http://webtest.canoo.com/>]是一个自由开放的工具，用web应用程序测试自动化。它与jWebUnit相似，不同的是你用XML进行测试，利用Ant执行。它的一个亮点是它能够测试PDF及其内容。为了将Canoo WebTest测试添加到MyUsers程序中，执行以下步骤。

1. 在web/WEB-INF/velocity目录下的userList.vm页面添加链接，这样你可以通过点击PDF链接得到一个PDF。紧跟着Add按钮添加以下HTML代码。

```
<p style="text-align: right; margin-bottom: -10px">
  <strong>Export Options:</strong>
  <a href="?report=XML">XML</a> .
  <a href="?report=Excel">Excel</a> .
  <a href="?report=PDF">PDF</a>
</p>
```

2. 在build.xml中添加一个test-canoo target。

```
<target name="test-canoo" depends="deploy"
  description="Runs Canoo WebTests in Tomcat to test JSPs">
  <taskdef file="webtestTasks.properties">
    <classpath>
      <path refid="classpath"/>
      <!-- for log4j.xml -->
      <path location="web/WEB-INF/classes"/>
    </classpath>
  </taskdef>
  <mkdir dir="${test.dir}/data"/>
  <!-- Delete old results file if it exists -->
  <delete file="${test.dir}/data/web-tests-result.xml"/>
  <property name="testcase" value="run-all-tests"/>
  <ant antfile="test/web-tests.xml" target="${testcase}"/>
</target>
```

3. 在test目录下新建一个config.xml文件，添加以下代码。

```
<config host="localhost" port="8080"
  protocol="http" basepath="${webapp.name}"
  resultpath="${test.dir}/data"
  resultfile="web-tests-result.xml"
  summary="true" saveresponse="true"/>
```

4. 在test目录下新建一个web-tests.xml文件，内容如下。

```
<!DOCTYPE project [
```



```

<!ENTITY config SYSTEM "file:./config.xml">
]>
<project basedir="." default="run-all-tests">
  <!-- Include messages.properties so we can test against
  keys, rather than values -->
  <property file="web/WEB-INF/classes/messages.properties"/>
  <!-- runs all targets -->
  <target name="run-all-tests" depends="UserTests"
    description="Call and executes all test cases (targets)">
    <echo>All UserTests passed!</echo>
  </target>
  <!-- Verify adding a user, viewing and deleting a user works -->
  <target name="UserTests"
    description="Adds a new user profile">
    <canoo name="userTests">
      &config;
      <steps>
        <!-- View add screen -->
        <invoke url="/editUser.html"/>
        <verifytitle text="${userForm.title}"/>
        <!-- Enter data and save -->
        <setinputfield name="firstName" value="Test"/>
        <setinputfield name="lastName" value="Name"/>
        <clickbutton label="Save"/>
        <!-- View user list -->
        <verifytitle text="${userList.title}"/>
        <!-- Verify PDF contains new user -->
        <clicklink label="PDF"/>
        <verifyPdfText text="Test Name"/>
        <!-- Delete first user in table -->
        <invoke url="/users.html"/>
        <clicklink href="editUser.html"/>
        <verifytitle text="${userForm.title}"/>
        <clickbutton label="Delete"/>
        <verifytitle text="${userList.title}"/>
      </steps>
    </canoo>
  </target>
</project>

```

文件中的UserTests target定义了一些task，用于测试MyUsers UI中各种操作。这个target的中间是通过一个调用产生PDF验证新添加的用户。

警告

如果你的应用无法生成PDF，请确定action-servlet.xml文件中的viewResolver2 bean包含以下属性。

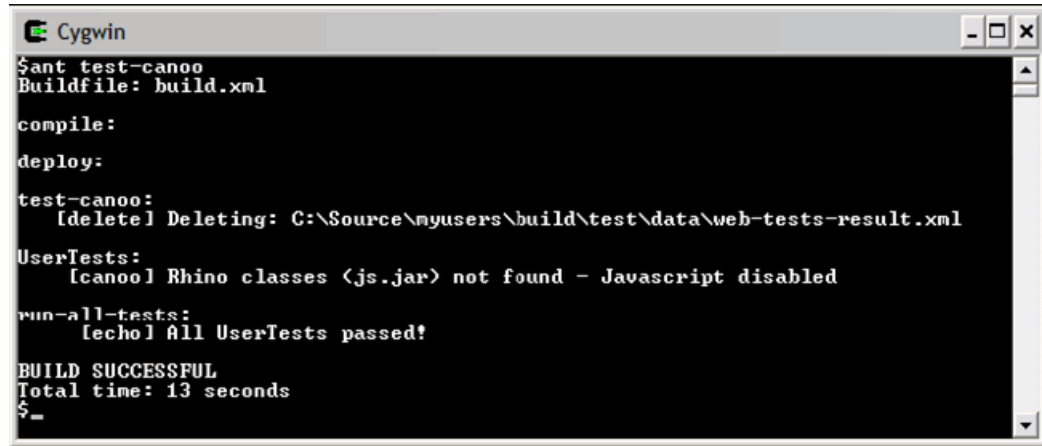
```
<property name="order"><value>1</value></property>
```

如果想查看log4j的错误信息，请确定你的log4j.xml文件中包含以下代码(在web/WEB-INF/classes目录下)。

```
<logger name="com.canoo">
  <level value="WARN" />
</logger>
```

5. 运行ant deploy, 启动Tomcat然后运行ant test-canoo。所有测试应该都可以通过, 你将看到一个与图8.8控制台输出画面。

图 8.8. ant test-canoo测试的结果



```
Cygwin
$ant test-canoo
Buildfile: build.xml

compile:
deploy:
test-canoo:
  [delete] Deleting: C:\Source\myusers\build\test\data\web-tests-result.xml
UserTests:
  [canoo] Rhino classes <js.jar> not found - Javascript disabled
run-all-tests:
  [echo] All UserTests passed!

BUILD SUCCESSFUL
Total time: 13 seconds
$_
```

自动启动Tomcat

和jWebUnit一样, 要在运行Canoo WebTest测试之前和之后自动启动和关闭Tomcat非常容易。还是使用Cargo API, 添加以下的test-view target到文件build.xml。

```
<target name="test-view" depends="war"
  description="Runs canoo tests in a running server">
  <taskdef resource="cargo.tasks" classpathref="classpath"/>
  <cargo-tomcat5x homeDir="${tomcat.home}"
    output="${test.dir}/cargo.log"
    workingDir="${test.dir}/tomcat5x" action="start">
    <war warFile="${dist.dir}/${webapp.name}.war"/>
  </cargo-tomcat5x>
  <antcall target="test-canoo"/>
</target>
```

即使Tomcat没有运行, 你也能通过ant test-view测试这个target。

提示

Canoo WebTest开发团队维护着一个项目相关的weblog tips
[<http://webtest-community.canoo.com/wiki/space/start>].

通过这个对jWebUnit和Canoo WebTest的简短考察，我希望你已经学到了如何在项目进行运用。很多开发人员还不知道自动测试UI，这个两种工具都能节省你大量的时间。当然，你还要在浏览器中经常查看你的UI来跟踪字体，颜色，位置。

本章小结

本章讨论了大量有关如何测试Spring的信息，还介绍了如何写单元测试。你学习了如何使用JUnit，DBUnit，EasyMock，jMock，Spring Mock，Cactus，jWebUnit和Canoo WebTest。对于数据库，你可以看到与一个数据库交互，验证DAO操作的最佳途径。还介绍了测试时覆盖bean的技巧(例如，在DAO实现中包装事务处理)。

在业务代码层，以EasyMock和jMock为例进行演示，两种都大大简化了模拟对象。决定模拟对象的最佳策略是，永远不模拟不属于你的API。既然UserDAO接口和它的实现是你创建的，那么在UserManagerTest测试中模拟它是有意义的。使用mock为孤立测试提供了一条有效的途径，并且在大多数情况下，它能使你的测试运行得更快。

在web层，你学习了如何使用jWebUnit来写一个基于Java的测试来验证UI特性。Canoo WebTest演示了同样的功能，但使用Ant/XML代替了Java。新生的Cargo项目向你演示了如何将启动/停止Tomcat作为运行测试的一部分。

本章没有介绍rich clients(例如，SWT和swing应用程序)。这主要是因为Spring Rich在写作时还没有发布。还有，似乎缺乏工具来测试这类程序。

测试是任何软件开发的一个重要的部分。本章向你提供了在项目中使用TDD开发的工具和技巧，以便产生高质量的软件。

第 9 章 AOP编程

什么是AOP，Spring如何简化AOP

AOP编程吸收了大量近年来来自java社区的灵感。什么是AOP,如何在你的应用程序中使用AOP?这一章将讲述AOP的基础，并且给出几个有用的实例，演示AOP能给你带来点什么。

概述

如果你一直活跃在Java社区，阅读weblog，杂志，浏览TheServerSide.com，你可能听说过AOP编程。AOP虽然出现了多年，但直到现在才引起人们的重视。这主要是因为很多基于Java的框架能够简单使用AOP。

AOP的最基本的定义是“它是一种剔除代码冗余的方法”。Java是一种面向对象的编程语言。它允许你创建应用程序，按照一定层次结构创建对象。然而，它没有提供一种简单的方法，来剔除处于不同层次中类的冗余代码。本章中，你将学习如何在项目中使用AOP。你会修改MyUsers程序，使用AOP控制日志，缓存，事务处理和e-mail。

一个应用程序一般有两个方面的因素：core concerns(核心因素)和crosscutting concerns(交叉因素)。core concerns与程序的功能相关，crosscutting concerns不仅影响特定的类，模块，还会影响整个系统。用于管理crosscutting concerns创建的模块就是截面(aspect)，它衍生了这个名称，面向截面的编程(AOP)。

AOP提供了一种优雅的方法，将你的类的crosscutting concerns的逻辑进行模块化，抽象成应用范围的因素，例如，日志，安全，事务处理等等。通过写一些通用的AOP类，你就可以其它项目中应用。使用Spring的IoC容器，你可以简单的配置这些类，将它们打包成一个jar文件，放在项目的classpath中。

在Spring术语中，截面(aspect)就是一个配置好了的监视方法调用的拦截器(interceptor)。在拦截过程中，拦截器会做很多事，记录正在调用的方法，修改返回的对象或是发送通知消息。Servlet Filter是AOP的一种表现形式，它能对servlet调用进行预处理和后续处理。

有关AOPJava社区一致认为，“对于AOP只有一种观点是你的系统并不依赖它。”。换句话说，如果有些东西对你的应用程序非常重要(比如业务逻辑)，那么它应该保留在你的应用程序代码中。这就是AOP代码不是高度透明的原因。使用Spring，你可以通过一个XML文件来判断哪个拦截器应用到哪个方法上。一些新手并不知道这些，而花费大量的时间试图找出为什么会执行特定的动作(由拦截器执行)。如果你想查看类的某种功能，那么它不应该在一个截面中。

AOP在你应该看到什么和不应该看到什么这方面有很大的帮助。例如，用Spring处理事务处理代码要优于自己手写。这种情况下在一个有初级和高级程序员的环境中表现得非常好，高级开发人员可以对初级程序员隐藏一些截面(aspect)。

本章面向AOP新用户。已经有讨论AOP的专著，这已超出了本书的范围。我推荐你阅读Ramnivas Laddad所著的AspectJ in Action [<http://www.manning.com/laddad/>]。书中第一章对AOP作了很好的介绍。这一章还向你演示了如何使用AOP，理论与实际相结合。通过向你演示Spring中如何使用AOP，你可以集中精力做自己的事，而不是关心AOP的方方面面。

入门

在本章中，和前面一章一样，你接下来会做一些练习。最简单的方法是从<http://sourcebeat.com/downloads>上下载MyUsers Chapter 9捆绑包。项目的树型目录结构即为第8章练习的结果。web/WEB-INF/lib目录中已经包含了所有你本章中要用到的jar文件。你也可以

使用前一章开发的应用程序。如何你打算这么做，从<http://sourcebeat.com/downloads>上下载Chapter 9 JARs。每一节会有一个针对所用新技术新添加的jar文件清单。你可以将本章作为一个参考，把这些理论应用到你自己的应用程序中去。

注意

如果你对下载和运行这些例子有任何异议，请发邮件给我<mattr@sourcebeat.com>，或在Spring Live Issue Tracker [<http://jira.sourcebeat.com/secure/BrowseProject.jspa?id=10000>]上新开一个话题。

日志示例

在大多数AOP书籍和论文中，作演示的第1个例子都是如何把AOP用于日志。出于调试目的，常常在方法的开头和结尾处加上日志消息。Spring中很容易实现，它不需要任何编码(仅需要XML)。

注意

为了在bean上应用拦截器，将它们配置为被代理的类。这就是JDK动态代码能在运行时创建接口实现的原因所在。在下面的日志示例后面，本节会讨论不同的AOP实现方法。

Spring 内置了一个 `TraceInterceptor` [<http://www.springframework.org/docs/api/org.springframework.aop.interceptor/TraceInterceptor.html>]，你可以将它作为你的被代理类的一个拦截器。在MyUsers中，`userManager`已经被代理了(使用 `TransactionProxyFactoryBean` [<http://www.springframework.org/docs/api/org.springframework.transaction.interceptor/TransactionProxyFactoryBean.html>]，所以添加一个拦截器很容易。

1. 打开目录 `web/WEB-INF` 下的 `applicationContext.xml`，添加一个 `loggingInterceptor` bean。

```
<bean id="loggingInterceptor"
      class="org.springframework.aop.interceptor.TraceInterceptor"/>
```

2. 在 `userManager` bean 中添加一个 `preInterceptors` 属性，使用这个拦截器。

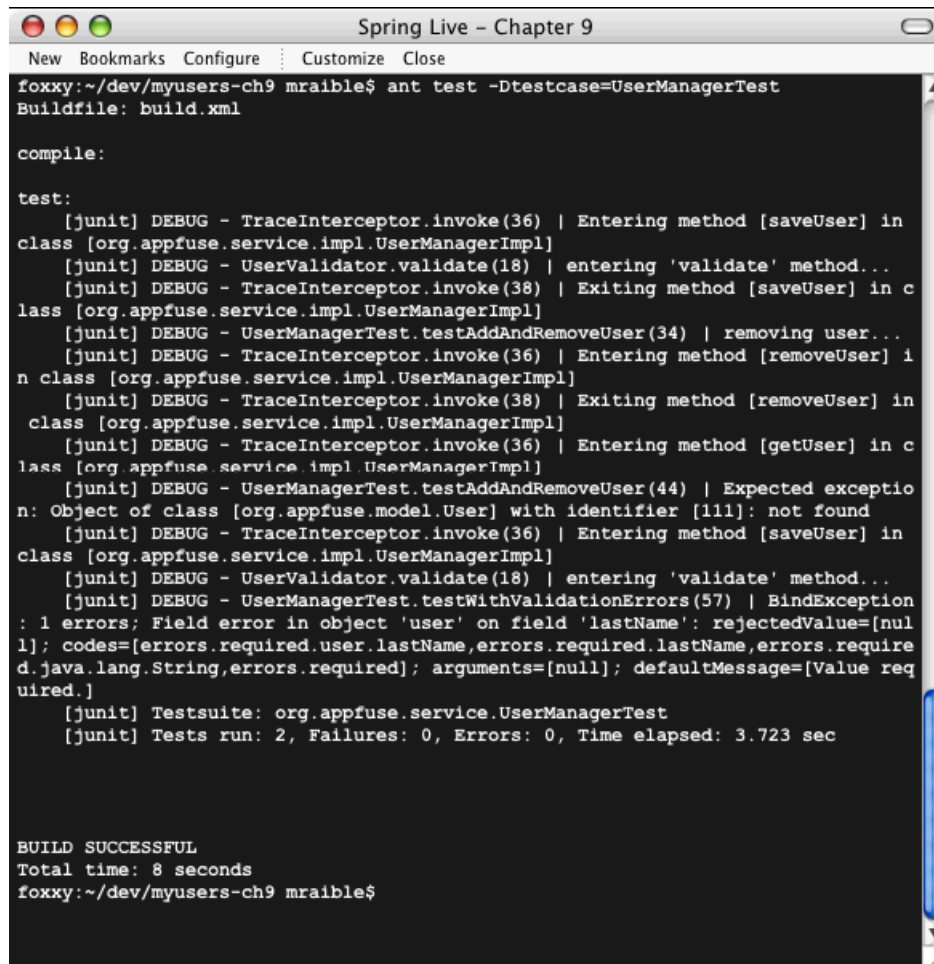
```
<property name="preInterceptors">
  <list>
    <ref bean="loggingInterceptor"/>
  </list>
</property>
```

3. 在 `web/WEB-INF/classes/log4j.xml` 添加一个 logger，显示 `TraceInterceptor` 的日志输出。

```
<logger name="org.springframework.aop.interceptor">
  <level value="DEBUG"/>
</logger>
```

4. 运行 `ant test -Dtestcase=UserManagerTest`。你得到的结果应该与下面的屏幕截图相似。

图 9.1. 运行 `ant test -Dtestcase=UserManagerTest` 测试的结果



```

Spring Live - Chapter 9
New Bookmarks Configure Customize Close
foxy:~/dev/myusers-ch9 mraible$ ant test -Dtestcase=UserManagerTest
Buildfile: build.xml

compile:

test:
[junit] DEBUG - TraceInterceptor.invoke(36) | Entering method [saveUser] in
class [org.appfuse.service.impl.UserManagerImpl]
[junit] DEBUG - UserValidator.validate(18) | entering 'validate' method...
[junit] DEBUG - TraceInterceptor.invoke(38) | Exiting method [saveUser] in c
lass [org.appfuse.service.impl.UserManagerImpl]
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(34) | removing user...
[junit] DEBUG - TraceInterceptor.invoke(36) | Entering method [removeUser] i
n class [org.appfuse.service.impl.UserManagerImpl]
[junit] DEBUG - TraceInterceptor.invoke(38) | Exiting method [removeUser] in
class [org.appfuse.service.impl.UserManagerImpl]
[junit] DEBUG - TraceInterceptor.invoke(36) | Entering method [getUser] in c
lass [org.appfuse.service.impl.UserManagerImpl]
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(44) | Expected exceptio
n: Object of class [org.appfuse.model.User] with identifier [111]: not found
[junit] DEBUG - TraceInterceptor.invoke(36) | Entering method [saveUser] in
class [org.appfuse.service.impl.UserManagerImpl]
[junit] DEBUG - UserValidator.validate(18) | entering 'validate' method...
[junit] DEBUG - UserManagerTest.testWithValidationErrors(57) | BindException
: 1 errors; Field error in object 'user' on field 'lastName': rejectedValue=[nul
l]; codes=[errors.required,user.lastName,errors.required.lastName,errors.require
d.java.lang.String,errors.required]; arguments=[null]; defaultMessage=[Value req
uired.]
[junit] Testsuite: org.appfuse.service.UserManagerTest
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 3.723 sec

BUILD SUCCESSFUL
Total time: 8 seconds
foxy:~/dev/myusers-ch9 mraible$

```

如果你看一下 `TraceInterceptor` 的源代码 [<http://monkeymachine.co.uk/spring/xref/org/springframework/aop/interceptor/TraceInterceptor.html>]，你会发现AOP是多么简单。`org.springframework.aop.interceptor` [<http://www.springframework.org/docs/api/org/springframework/aop/interceptor/package-summary.html>] 有好几个拦截器，在项目中非常有用。

这个简单的示例是对AOP的初步介绍。

定义与概念

AOP，和其它编程方法一样，拥有自己的一套词汇。下表定义一些词语或短语，你在学习或使用AOP时可能会碰到。这些定义并不是Spring特定的。

表 9.1. AOP定义

术语	定义
Concern(要素)	一个特别的观点, 概念, 或是一个程序的敏感区。例如, 事务管理, 持久层, 日志, 安全。
Crosscutting Concern(横向要素)	一种concern, 特定的实现会在其中横切各个类。这与OOP的实现和维护截然不同。
Aspect(截面, 剖面)	crosscutting concern的模块化, 通过集中和隔离代码实现。
Join Point(交叉点)	程序或类执行过程中的一个点。在Spring的AOP实现中, 一个join point通常是指方法调用。险些之外还包括访问字段(accessing fields, 读写访问出现在一个实例变量上)和异常处理(exception handling)。
Advice(切片, 楔子)	<p>在某一特定的join point执行的一个动作。Spring中的几种不同类型的advice, 包括around, before, throws和after returning。其中, around功能最为强大, 在一个方法调用前后, 你都可以执行一些操作。前面的TraceInterceptor通过实现AOP Alliance的 MethodInterceptor [http://aopalliance.sourceforge.net/doc/org/aopalliance/intercept/MethodInterceptor.html] 接口使用了around advice。要使用其它advice, 可以实现以下Spring接口。</p> <ul style="list-style-type: none"> • <code>MethodBeforeAdvice</code> [http://www.springframework.org/docs/api/org/springframework/aop/MethodBeforeAdvice.html] • <code>ThrowsAdvice</code> [http://www.springframework.org/docs/api/org/springframework/aop/ThrowsAdvice.html] • <code>AfterReturningAdvice</code> [http://www.springframework.org/docs/api/org/springframework/aop/AfterReturningAdvice.html]
Pointcut(切入点, 切口)	一系列的join point, 指明何时该触发某个advice。Pointcut一般使用正则表达式或是通配符。
Introduction	向一个advised类添加字段或方法。Spring允许你向你任何被切入的(advised)对象引入接口。例如, 你可以使用一个introduction, 让任何对象实现一个IsModified接口, 以简化缓存。
Weaving	装配aspect, 创建一个被切入的(advised)对象。这可以在编译时(AspectJ [http://www.eclipse.org/aspectj/])使用这种方式或运行时完成。本章后面Weaving策略一节会详细讨论各种weaving策略(也就是, 实现AOP)。
Interceptor(拦截器)	一种AOP实现策略。针对某个特定的join point可能会有一连串的Interceptor。
AOP Proxy(AOP代理)	由AOP框架创建的对象, 包含advice。在Spring中, 一个AOP Proxy要么是一个JDK dynamic proxy或是一个CGLIB proxy。
Target Object(目标对象)	包含join point的对象。在使用拦截框架中, 它是处于拦截器链末尾的一个对象实例。也叫advised或proxied对象。

下一节探讨Pointcuts, 即应用advice的规则。因为Spring的AOP是基于拦截器, 下面提到的advice即指拦截器(interceptor)。

Pointcuts(切入点)

Pointcuts是AOP的一个重要部分。有了它们你就可以决定何时何地调用拦截器。在某种意义上, 它们和声明式的验证相似, 但不是指定要验证的某个字段, 而是指定一个要拦截的方法。在表中, pointcuts定义为“一系列的join point, 指明何时该触发某个advice(拦截器)”。既然Spring仅支持方法调用的join point, pointcut就是拦截器所应用的方法的声明。

在Spring AOP中定义pointcuts最简单的方法是在一个context文件使用正则表达式。下面的是定义一个用于数据处理操作的pointcut实例。

```
<bean id="dataManipulationPointcut"
  class="org.springframework.aop.support.JdkRegexpMethodPointcut">
  <property name="patterns">
    <list>
      <value>.*save.*</value>
      <value>.*remove.*</value>
    </list>
  </property>
</bean>
```

这个pointcut喻示，“拦截以save或remove开头的方法”。

注意

上面的 `JdkRegexpMethodPointcut` [<http://www.springframework.org/docs/api/org/springframework/aop/support/JdkRegexpMethodPointcut.html>] 类需要J2SE1.4，它内置了正则表达式的支持。你也可以使用 `Perl5RegexpMethodPointcut` [<http://www.springframework.org/docs/api/org/springframework/aop/support/Perl5RegexpMethodPointcut.html>]，它需要Jakarta ORO [<http://jakarta.apache.org/oro/>] (包含在MyUsers中)。

在大多数情况下，你不必像上面列出的那个那样定义单个的pointcuts。Spring提供了advisor类，可以将拦截器(interceptors)和pointcuts封装在同一bean定义中。

对于正则表达式的pointcuts，你可以使用 `RegexpMethodPointcutAdvisor` [<http://www.springframework.org/docs/api/org/springframework/aop/support/RegexpMethodPointcutAdvisor.html>]。下面是使用`RegularExpressionPointcutAdvice`的一个例子，当保存用户资料时，触发`NotificationInterceptor`。

```
<bean id="notificationAdvisor"
  class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
  <property name="advice">
    <ref bean="notificationInterceptor"/>
  </property>
  <property name="pattern">
    <value>.*saveUser</value>
  </property>
</bean>
```

目前，`RegexpMethodPointcutAdvisor`仅支持Perl5 pointcuts，言下之意就是，如果你想使用它，就要把`jakarta-oro.jar`添加到你的classpath中。`org.springframework.aop.support` [<http://www.springframework.org/docs/api/org/springframework/aop/support/package-summary.html>]包中罗列了所有的pointcuts及其有用的advisors。

Weaving策略

Weaving是将aspect应用到目标(target object)的过程。下面清单列出了实现AOP的主要策略，由简单到复杂进行排序。

注意

本节中大量资料来自 J 2 E E w i t h o u t E J B
[http://www.theserverside.com/articles/article.tss?l=J2EEWithoutEJB_BookReview]。

- JDK Dynamic Proxies(JDK动态代理)
- Dynamic Byte-Code Generation(动态字节码生成)
- Custom Class Loading(自定义类加载)
- Language Extensions(语言扩展)

这些策略在不同的框架中得已实现，都是开源的。

注意

所有的这些框架中最令人欣慰的是他们愿意在各自的API中使用同一标准。为了支持这一作法，他们建立了AOP Alliance [<http://aopalliance.sourceforge.net/>]项目，规范要实现的接口。要查看有谁参与了这个项目，请查看成员列表 [<http://aopalliance.sourceforge.net/members.html>]。

下面的几节详细介绍各种weaving策略。

JDK Dynamic Proxies(JDK动态代理)

动态代理是J2SE1.3以上版本内建的一种特性。它允许你即时的创建一个或多个接口的实现。动态代理内建进了JDK，以消除不同环境中未知操作带来的风险。它们有一个限制，就是不能代理类，不能代理接口。如果你的应用程序是用接口设计的，这不成问题。

使用动态代理时，会过多的使用反射机制(reflection)，但在J2SE1.4以上的JVM中，性能影响是微乎其微的。Spring在代理接口时默认使用了动态代理。dynaop [v]在代理接口时也采用这种策略。

有关动态代理的更多信息，请参考 J a v a 2 S D K 文 档
[<http://java.sun.com/j2se/1.4.2/docs/guide/reflection/proxy.html>]。

Dynamic Byte-Code Generation(动态字节码生成)

Spring在代理类时使用了动态字节码生成。相关的一种流行工具叫CGLIB [<http://cglib.sourceforge.net/>](Code Generation Library)。它通过动态生成子类的方式拦截方法。这些子类覆写父类的方法，间接的调用拦截器实现。Hibernate中大量的使用CGLIB，这也证明在J2EE中它是一种非常稳妥的方案。

这里有一个限制就是动态子类不能覆写和代理final方法。

Custom Class Loading(自定义类加载)

使用一个自定义的类加载器允许你对所创建的类的所有的实例进行切入(advice)操作。这项功能非常强大，因为它为你提供了机会改变new操作的行为。JBoss AOP
[<http://www.jboss.org/developers/projects/jboss/aop>]和AspectWerkz [<http://aspectwerkz.codehaus.org/>]采用了这种方法，通过在XML文件中预先定义加载类，组织(weaving)它们的切片(advice)。

这种方法的主要问题是J2EE服务器必须控制类加载的层次结构。在一个服务器上运行得不错，不行保证能在另一服务器上运行。

Language Extensions(语言扩展)

AspectJ [<http://www.eclipse.org/aspectj/>]Java中一种领先的AOP框架。它不再是一种组装切面简单策略，它包含自己的语言扩展 和自己的一套编译器。

AspectJ是一种非常强大和成熟的AOP的实现，它的语法有点复杂，并且不大直观。AOP本身就不大直观，所以试图用一门新语法来实现不那么容易。这种方法的另一种限制就是一种新语法的学习曲线。但是，如果你想拥有AOP的强大功能，包括字段级的拦截，那么AspectJ可能是你的知音。本章后面将介绍AspectJ与Spring集成。

在AOP实现上，Spring采用了一个实用的“80/20规则”的折中方案。它并没有尝试满足每个人的要求，而是解决一些最常见的问题，把剩余的问题留给更为专业的AOP框架。

常用的代理Bean

正如前面的提到的那样，如果你想将切片(advice)应用到context文件中定义的bean上，它们必须是被代理的。Spring中有好几个支持类(代理类)，实现代理很简单。第一个是 ProxyFactoryBean [<http://www.springframework.org/docs/api/org/springframework/aop/framework/ProxyFactoryBean.html>]，它允许你指定要代理哪个bean及要应用的拦截器。下面是一个样例，使用这个bean创建一个业务对象代理。

```
<bean id="businessObject"
  class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target">
    <bean class="org.appfuse.service.BusinessObject"/>
  </property>
  <property name="interceptorNames">
    <list><ref bean="loggingInterceptor"/></list>
  </property>
</bean>
```

上面的例子中的target属性值引用一个内部bean。这是一个通过代理来隐藏业务对象的常用方法，所以它总是在获取ApplicationContext时被切入。

T r a n s a c t i o n P r o x y F a c t o r y B e a n
[<http://www.springframework.org/docs/api/org/springframework/transaction/interceptor/TransactionProxyFactoryBean.html>]是最有用也是最常用的Proxy Bean。它允许你使用AOP在目标对象上显式的定义事务。这个有用的特性以前只在EJB的容器托管事务中提供。这个bean的用法将AOP实战一节中描述。

AutoProxy Beans

前面提到的Proxy Bean提供了单个bean的简单操作，但是如果你对多个bean，或是同一个context中的所有进行代理呢？Spring提供两个类（在包org.springframework.aop.framework.autoproxy [<http://www.springframework.org/docs/api/org/springframework/aop/framework/autoproxy/package-summary.html>]中）来简化这个步骤。

第一个是 **B e a n N a m e A u t o P r o x y C r e a t o r**
[<http://www.springframework.org/docs/api/org/springframework/aop/framework/autoproxy/BeanNameAutoProxyCreator.html>]，它允许你指定一个bean名称组成的list作为一个属性。这个属性支持文字(完整的bean名称)或是通配符，如*Manager。你可以通过指定interceptorNames属性来设置拦截器。

```
<bean id="managerProxyCreator"
  class="org.springframework.aop.framework.autoproxy.
  BeanNameAutoProxyCreator">
  <property name="beanNames"><value>*Manager</value></property>
  <property name="interceptorNames">
    <list>
      <value>loggingInterceptor</value>
    </list>
  </property>
</bean>
```

第二个，更常用，多个bean的代理bean creator是 `DefaultAdvisorAutoProxyCreator` [<http://www.springframework.org/docs/api/org.springframework.aop.framework.autoproxy/DefaultAdvisorAutoProxyCreator.html>]。要用这个bean，在你的context文件简单的定义。

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

和`BeanNameAutoProxyCreator`不同的是，不必指定拦截器。它会拦截这个文件中所有的advisor，计算它们的pointcut是否能让它们适用于其它bean。更多有关advisors的信息，请参考Spring文档 []。

AOP实战

本切包含几个在应用中使用Spring管理crosscutting concerns例子。这里concerns包括事务，缓存，及事件通知。

事务

指定了应该在事务内出现的操作往往导致DAO产生大量的冗余代码。使用事务处理时，传统方法需要在数据处理方法中大量调用`tx.begin()`和`tx.commit()`。可喜的是Spring和AOP可以在同一位置和配置中处理事务。

你可能还没有觉察到，你在快速入门一章中的MyUsers程序中已经使用了AOP。可以在`userManager` bean声明一个`transaction`属性，这是由Spring的AOP和`TransactionProxyFactoryBean` [<http://www.springframework.org/docs/api/org.springframework.transaction.interceptor/TransactionProxyFactoryBean.html>]支持的。下面的代码清单显示了一个经过重构的`userManager` bean，这里使用了个`transaction` template bean和一个`inner-bean`。

```
<bean id="txProxyTemplate" abstract="true"
  class="org.springframework.transaction.interceptor.
  TransactionProxyFactoryBean">
  <property name="transactionManager">
    <ref bean="transactionManager"/>
  </property>
  <property name="transactionAttributes">
```

```

<props>
  <prop key="save*">PROPAGATION_REQUIRED</prop>
  <prop key="remove*">PROPAGATION_REQUIRED</prop>
  <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
</props>
</property>
<property name="preInterceptors">
  <list>
    <ref bean="cacheInterceptor"/>
  </list>
</property>
</bean>
<bean id="userManager" parent="txProxyTemplate">
  <property name="target">
    <bean class="org.appfuse.service.impl.UserManagerImpl">
      <property name="userDAO"><ref bean="userDAO"/></property>
      <property name="validator">
        <ref bean="userValidator"/>
      </property>
    </bean>
  </property>
</bean>
</property>
</bean>

```

警告

在1.1.2之前的版本，你可以使用`lazy-init="true"`代替`abstract="true"`。在1.1.2及以后的版本中，Spring会抛出一个异常：“`java.lang.IllegalArgumentException: 'target' is required`”。1.1.1及以前的版本允许使用`lazy-init="true"`代替`abstract`属性。`abstract`是在1.1中加进去的，用来标记那些不必进行初始化父类。

更多有关声明式事务处事和异常处理，及事务撤销将在第10章中介绍。

中间层的缓存

使用crosscutting concern的另一个很好的例子是缓存数据。增加缓存的主要原因是为了提高性能，特别是提取数据是代价很大的操作。最后一章讨论的大多数框架都有各自的一套缓存方案：缓存是一套实用的crosscutting concern。

既然引入缓存的动机是提高性能，在User-ManagerTest添加一个测试来测试UserMangerImpl。在`test/org/appfuse/service/UserManagerTest.java`中添加以下代码。

注意

本次测试中的 `StopWatch` [<http://www.springframework.org/docs/api/org/springframework/util/StopWatch.html>]是一个计时的工具类。

```

public void testGetUserPerformance() {
    user = new User();
    user.setFirstName("Easter");
    user.setLastName("Bunny");
}

```

```

user = mgr.saveUser(user);
String name = "getUser";

StopWatch sw = new StopWatch(name);
sw.start(name);
log.debug("Begin timing of method '" + name + "'");
for (int i=0; i < 200; i++) {
    mgr.getUser(user.getId().toString());
}
sw.stop();
log.info(sw.shortSummary());
log.debug("End timing of method '" + name + "'");
}

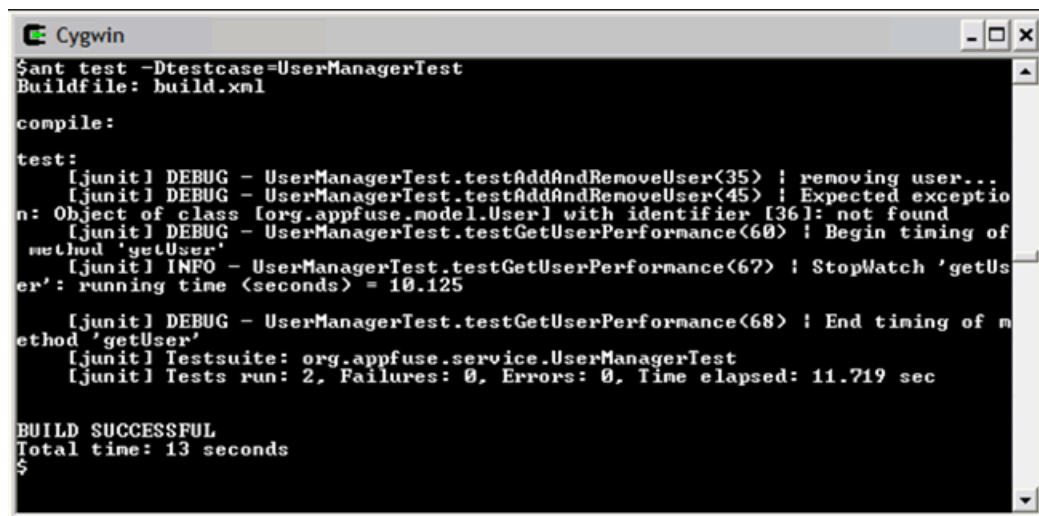
```

运行测试`ant test -Dtestcase=UserManagerTest`(在`applicationContext.xml`配置拦截器的情况下)得到的结果与图9.2类似。在这个最初的测试中，提取同一个用户资料大约要花10秒。

警告

如果你使用`-Dtestcase=UserManager`(没有带Test后缀)测试，它将运行我们在上一章创建的mock测试。这些测试是孤立的，因此它们无法演示应用`ApplicationContext.xml`中定义的拦截。

图 9.2. 运行`ant test -Dtestcase=UserManagerTest`测试的结果



```

$ ant test -Dtestcase=UserManagerTest
Buildfile: build.xml

compile:
test:
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(35) ! removing user...
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(45) ! Expected exception
n: Object of class [org.appfuse.model.User] with identifier [36]: not found
[junit] DEBUG - UserManagerTest.testGetUserPerformance(60) ! Begin timing of
method 'getUser'
[junit] INFO - UserManagerTest.testGetUserPerformance(67) ! Stopwatch 'getUs
er': running time <seconds> = 10.125
[junit] DEBUG - UserManagerTest.testGetUserPerformance(68) ! End timing of m
ethod 'getUser'
[junit] Testsuite: org.appfuse.service.UserManagerTest
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 11.719 sec

BUILD SUCCESSFUL
Total time: 13 seconds
$

```

为了提高性能，在`UserManagerImpl`添加一个简单的缓存。此缓存的基本要求如下：

1. 无论何时你从数据库中的提取数据，将它们放入缓存中(本例中使用一个简单的HashMap)。
2. 当调用`save()`或`remove`方法时，从缓存中删除对象。

在`MyUsers`程序中，你可以通过增加一个`BaseManager`实现一个简单的方案(非AOP)，其它所有的`ManagerImpls`可以从它继承。

```
package org.appfuse.service.impl;
```

```
// use your IDE to organize imports

public class BaseManager {
    // Adding this log variable will allow children to re-use it
    protected final Log log = LogFactory.getLog(getClass());
    protected Map cache;

    protected void putIntoCache(String key, Object value) {
        if (cache == null) {
            cache = new HashMap();
        }
        cache.put(key, value);
    }

    protected void removeFromCache(String key) {
        if (cache != null) {
            cache.remove(key);
        }
    }
}
```

修改这个UserManagerImpl，以使用这个缓存，确保它继承了BaseManager，然后在每个方法中调用缓存。下面是添加调用之前一个简单的UserManagerImpl版本。

```
public User getUser(String userId) {
    return dao.getUser(Long.valueOf(userId));
}

public User saveUser(User user) {
    dao.saveUser(user);
    return user;
}

public void removeUser(String userId) {
    dao.removeUser(Long.valueOf(userId));
}
```

添加这些方法后，它们变得有些冗长。

```
public User getUser(String userId) {
    // check cache for user
    User user = (User) cache.get(userId);
    if (user == null) {
        // user not in cache, fetch from database
        user = dao.getUser(Long.valueOf(userId));
        super.putIntoCache(userId, user);
    }
    return user;
}
```

```

public User saveUser(User user) {
    dao.saveUser(user);
    // update cache with saved user
    super.putIntoCache(String.valueOf(user.getId()), user);
    return user;
}

public void removeUser(String userId) {
    dao.removeUser(Long.valueOf(userId));
    // remove user from cache
    super.removeFromCache(userId);
}

```

运行 `ant test -Dtestcase=UserManagerTest` 提高性能，从图9.3中可以得到，运行时间为9秒(快了1秒)。

图 9.3. 运行 `ant test -Dtestcase=UserManagerTest` 测试的结果

```

Cygwin
$ ant test -Dtestcase=UserManagerTest
Buildfile: build.xml

compile:

test:
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(35) ! removing user...
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(45) ! Expected exception: Object of class [org.appfuse.model.User] with identifier [36]: not found
[junit] DEBUG - UserManagerTest.testGetUserPerformance(60) ! Begin timing of method 'getUser'
[junit] INFO - UserManagerTest.testGetUserPerformance(67) ! Stopwatch 'getUser': running time <seconds> = 10.125
[junit] DEBUG - UserManagerTest.testGetUserPerformance(68) ! End timing of method 'getUser'
[junit] Testsuite: org.appfuse.service.UserManagerTest
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 11.719 sec

BUILD SUCCESSFUL
Total time: 13 seconds
$

```

虽然添加从缓存中添加删除对象的调用对于MyUsers这样的简单程序只是小菜一碟，但是对于大型程序麻烦会日益增加。你不得不记住对每个UserManager添加这些方法。既然缓存不是一个应用程序的core concern，你不应该干预它。

通过使用AOP，你可以剔除这些方法调用，拦截相应的方法来使用缓存。另外，通过从代码把缓存抽取出来，你可以把精力集中在业务逻辑，项目开发上。

要添加一个缓存拦截器，执行下面的步骤。

1. 在包 `org.appfuse.aop` (你必须创建) 中创建一个类 `CacheInterceptor`，填充以下代码。

```

package org.appfuse.aop;
// use your IDE to organize imports

public class CacheInterceptor implements MethodInterceptor {
    private final Log log = LogFactory.getLog(
        (CacheInterceptor.class));

```

```

public Object invoke(MethodInvocation invocation) throws Throwable {
    String name = invocation.getMethod().getName();
    Object returnValue;

    // check cache before executing method
    if (name.indexOf("get") > -1 && !name.endsWith("s")) {
        String id = (String) invocation.getArguments()[0];
        returnValue = cache.get(id);
        if (returnValue == null) {
            // user not in cache, proceed
            returnValue = invocation.proceed();
            putIntoCache(id, returnValue);
            return returnValue;
        } else {
            //log.debug("retrieved object id '" + id + "' from cache");
        }
    } else {
        returnValue = invocation.proceed();
        // update cache after executing method

        if (name.indexOf("save") > -1) {
            Method getId = returnValue.getClass().getMethod("getId", new
                Class[]{});
            Long id = (Long) getId.invoke(returnValue, new Object[]{});
            putIntoCache(String.valueOf(id), returnValue);
        } else if (name.indexOf("remove") > -1) {
            String id = (String) invocation.getArguments()[0];
            removeFromCache(String.valueOf(id));
        }
    }
    return returnValue;
}

protected Map cache;
protected void putIntoCache(String key, Object value) {
    if (cache == null) {
        cache = new HashMap();
    }
    cache.put(key, value);
}

protected void removeFromCache(String key) {
    if (cache != null) {
        cache.remove(key);
    }
}
}

```

2. 在文件 `applicationContext.xml` (`web/WEB-INF` 目录下) 添加一个 `cacheInterceptor` bean。

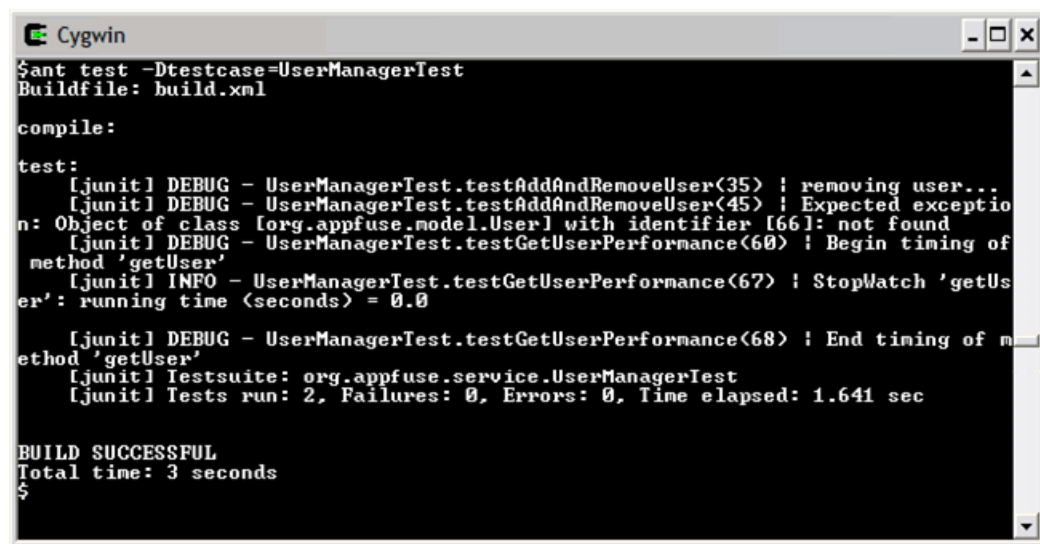

```
<bean id="cacheInterceptor" class="org.appfuse.aop.CacheInterceptor"/>
```

3. 将userManager bean的loggingInterceptor替换成cacheInterceptor。

```
<property name="preInterceptors">
  <list>
    <ref bean="cacheInterceptor"/>
  </list>
</property>
```

运行 `ant test -Dtestcase=UserManagerTest`，可以看到性能有戏剧性地提高。

图 9.4. 运行 `ant test -Dtestcase=UserManagerTest` 测试的结果



```
Cygwin
$ant test -Dtestcase=UserManagerTest
Buildfile: build.xml

compile:

test:
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(35) : removing user...
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(45) : Expected exception
n: Object of class [org.appfuse.model.User] with identifier [66]: not found
[junit] DEBUG - UserManagerTest.testGetUserPerformance(60) : Begin timing of
method 'getUser'
[junit] INFO - UserManagerTest.testGetUserPerformance(67) : Stopwatch 'getUs
er': running time <seconds> = 0.0
[junit] DEBUG - UserManagerTest.testGetUserPerformance(68) : End timing of m
ethod 'getUser'
[junit] Testsuite: org.appfuse.service.UserManagerTest
[junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 1.641 sec

BUILD SUCCESSFUL
Total time: 3 seconds
$
```

使用caching aspect的话，你可以时间消耗减少到0秒。在性能上出现这种戏剧性提升，是由于几乎在UserManagerImpl所有的方法调用之前，CacheInterceptor已经返回了数据。

上面提供的样例过于简单。要想了解一种更为健壮的缓存实现机制，请参考Pieter Coucke的Spring AOP Cache [http://www.onthoo.com/blog/programming/2004/09/spring-aop-cache.html]或是using E H C a c h e w i t h S p r i n g [http://opensourceatlassian.com/confluence/spring/display/DISC/Caching+the+result+of+methods+using+Spring+and+EHCACHE].

事件通知

如果你开发的web程序是对外公开的，那么当有新用户注册时，你可能想得到一个通知。如果你想帐户授权之前审查用户资料，这样做可以非常有用。在下面的例子中，你将创建一个NotificationInterceptor，配置好后，当新用户登记时发送一封邮件。

1. 修改UserManagerTest，使用Dumbster(???)保证在有用户注册时消息已经发送出去了。下面仅列出的相关的部分。这个类中新添加的代码以下划线标出。添加后，运行`ant test -Dtestcase=UserManagerTest`会得到失败的结果。

```
protected void setUp() throws Exception {
    String[] paths = {"/WEB-INF/applicationContext*.xml"};
    ctx = new ClassPathXmlApplicationContext(paths);
    mgr = (UserManager) ctx.getBean("userManager");

    // Modify the mailSender bean to use Dumbster's ports
    JavaMailSenderImpl mailSender =
        (JavaMailSenderImpl) ctx.getBean("mailSender");
    mailSender.setPort(2525);
}

public void testAddAndRemoveUser() throws Exception {
    user = new User();
    user.setFirstName("Easter");
    user.setLastName("Bunny");

    // setup a simple mail server using Dumbster
    SimpleSmtpServer server = SimpleSmtpServer.start(2525);
    user = mgr.saveUser(user);
    server.stop();
    assertEquals(1, server.getReceievedEmailSize()); // spelling is correct
    SmtppMessage sentMessage =
        (SmtppMessage) server.getReceivedEmail().next();
    assertTrue(sentMessage.getBody().indexOf("Easter Bunny") != -1);
    log.debug(sentMessage);
    assertTrue(user.getId() != null);
}
```

2. 在目录src/org/appfuse/aop中新建一个类NotificationInterceptor，填充以下代码。

```
package org.appfuse.aop;
// organize imports using your IDE

public class NotificationInterceptor implements MethodInterceptor {
    private final Log log =
        LoggerFactory.getLog(NotificationInterceptor.class);
    private MailSender mailSender;
    private SimpleMailMessage message;

    public void setMailSender(MailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void setMessage(SimpleMailMessage message) {
        this.message = message;
    }
}
```

```

    }

    public Object invoke(MethodInvocation invocation) throws Throwable {
        User user = (User) invocation.getArguments()[0];
        if (user.getId() == null) {
            if (log.isDebugEnabled()) {
                log.debug("detected new user...");
            }
            Object returnValue = invocation.proceed();
            StringBuffer sb = new StringBuffer(100);
            sb.append("A new account has been created for " +
                user.getFullName());
            sb.append(".\n\nView this users information at:\n\t ");
            sb.append("http://localhost:8080/myusers/editUser.html?
                id=" + user.getId());
            message.setText(sb.toString());
            mailSender.send(message);
        }
        return user;
    }
}

```

3. 在web/WEB-INF/applicationContext.xml配置好这个拦截器及其依赖的bean(MailSender和SimpleMailMessage)。

警告

请确定修改了下面的accountMessage bean中to属性中email地址。如果不这么做，会导致失败。

```

<bean id="notificationInterceptor"
    class="org.appfuse.aop.NotificationInterceptor">
    <property name="mailSender"><ref bean="mailSender"/></property>
    <property name="message"><ref bean="accountMessage"/></property>
</bean>
<bean id="mailSender"
    class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <property name="host"><value>localhost</value></property>
</bean>

<bean id="accountMessage" singleton="false"
    class="org.springframework.mail.SimpleMailMessage">
    <property name="to">
        <value><![CDATA[MyUser Admin <youremailhere>]]></value>
    </property>
    <property name="from">
        <value><![CDATA[MyUsers <mattr@sourcebeat.com>]]></value>
    </property>
    <property name="subject">
        <value>MyUsers Account Information</value>
    </property>

```

```
</bean>
```

4. 在一个advisor中配置这个拦截，使用RegexpMethodPointcutAdvisor激活。

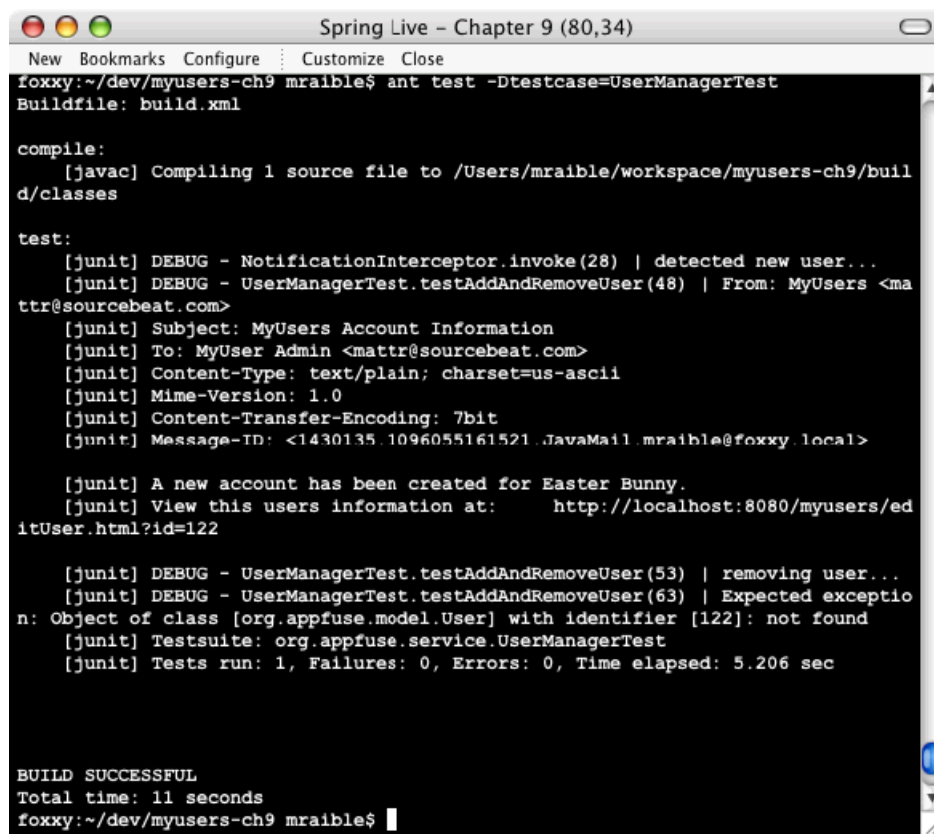
```
<bean id="notificationAdvisor"
      class="org.springframework.aop.support.
        RegexpMethodPointcutAdvisor">
  <property name="advice">
    <ref bean="notificationInterceptor"/>
  </property>
  <property name="pattern">
    <value>.*saveUser</value>
  </property>
</bean>
```

5. 在userManager bean的preInterceptors list中notificationAdvisor bean。我建议注释掉其它用例。

```
<property name="preInterceptors">
  <list>
    <!--ref bean="loggingInterceptor"/-->
    <!--ref bean="cacheInterceptor"/-->
    <ref bean="notificationAdvisor"/>
  </list>
</property>
```

6. 保存所有的文件，运行ant test -Dtestcase=UserManagerTest。你的控制台输出应该和图9.5类似。

图 9.5. 运行ant test -Dtestcase=UserManagerTest测试的结果



```
Spring Live - Chapter 9 (80,34)
New Bookmarks Configure Customize Close
foxy:~/dev/myusers-ch9 mraible$ ant test -Dtestcase=UserManagerTest
Buildfile: build.xml

compile:
[javac] Compiling 1 source file to /Users/mraible/workspace/myusers-ch9/build/classes

test:
[junit] DEBUG - NotificationInterceptor.invoke(28) | detected new user...
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(48) | From: MyUsers <mraible@sourcebeat.com>
[junit] Subject: MyUsers Account Information
[junit] To: MyUser Admin <mraible@sourcebeat.com>
[junit] Content-Type: text/plain; charset=us-ascii
[junit] Mime-Version: 1.0
[junit] Content-Transfer-Encoding: 7bit
[junit] Message-ID: <1430135.1096055161521.JavaMail.mraible@foxy.local>

[junit] A new account has been created for Easter Bunny.
[junit] View this users information at: http://localhost:8080/myusers/editUser.html?id=122

[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(53) | removing user...
[junit] DEBUG - UserManagerTest.testAddAndRemoveUser(63) | Expected exception: Object of class [org.appfuse.model.User] with identifier [122]: not found
[junit] Testsuite: org.appfuse.service.UserManagerTest
[junit] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 5.206 sec

BUILD SUCCESSFUL
Total time: 11 seconds
foxy:~/dev/myusers-ch9 mraible$
```

通过这些步骤设置，你很容易添加基于数值和条件的通知机制。

这些例子是一个很好的有关AOP及如何在应用中使用它的入门教程。其它你可以在应用中会用到AOP的地方包括：

- 资源池技术
- 线程池技术
- XSLT样式表缓存
- 认证和授权

AspectJ in Action一书中提供了相关实现的样例和代码。

AOP功能非常强大，新用户在它们的基础代码中植入时要特别小心。下面列出了学习AOP和深入的两种途径。

1. 通过现成处方：你可以从书籍和论文中拷贝和复制代码。
2. 在非产品中使用：AOP的一个优点，是没有必要一次使用所有的特性。这可以循序渐进的学习。你可以在开发阶段开始使用(例如，日志记录/跟踪的例子)，然后你可以转移到处理认证和测试辅助等方面。当你真正适应后，就可以在产品中使用它。

虽然它不是为每个准备的，它为创建模块化，可维护的Java程序占有一席之地。

本章小结

AOP虽然已经出现多年，但直到最后才在Java领域中流行起来。这主要归功于大量实现了AOP的开源框架的涌现。写本书时，已经有AspectJ [<http://www.eclipse.org/aspectj/>]，AspectWerkz [<http://aspectwerkz.codehaus.org/>]，dynaop [<https://dynaop.dev.java.net/>]，JCA [<http://jac.objectweb.org/>]，JBoss AOP [<http://www.jboss.org/products/aop>]和Spring AOP几种。

Spring提供了一种简单易用的AOP API，并能很好与其它AOP框架集成。在Spring1.1中，它提供了与AspectJ的强大集成功能 [<http://www.springframework.org/docs/reference/ch06.html>]。它也能好的与AspectWerkz集成 [http://blogs.codehaus.org/people/jboner/archives/000826_spring_and_aspectwerkz_a_happy_marriage.html]。所有的这些集成方案中，Spring的IoC容器能够配置和管理所建aspects的生命周期。

在本章中，你学习了AOP的各种概念，从aspects到pointcuts到weaving。在定义了AOP的各个术语后，我们探讨了各种实现策略。JDK动态代理和字节码处理是前面提及的AOP框架中最为流行的weaving策略。AspectJ是最为成熟和复杂的AOP实现，在集成aspects和类时提供专有的语言和编译器。最后，你学习了如何在MyUsers应用程序中创建aspects和advisors，来实现日志，缓存，事务处理和事件通知机制。

那么Spring AOP框架的前景如何？根据其发展路线 [<http://www.springframework.org/docs/reference/aop.html#d0e3961>]，很多目标在1.1版本中得以实现，还有提高性能和AspectJ集成。Spring AOP Cache [<http://www.onthoo.com/blog/programming/2004/09/spring-aop-cache.html>]和Acegi Security System for Spring [<http://acegisecurity.sourceforge.net/>]是在缓存和安全方面的典范。当更多的用户熟悉AOP并开始在项目中使用它时，这方面的资料就会大量出现。

第 10 章 事务处理

在Spring中使用声明式和编程式的事务

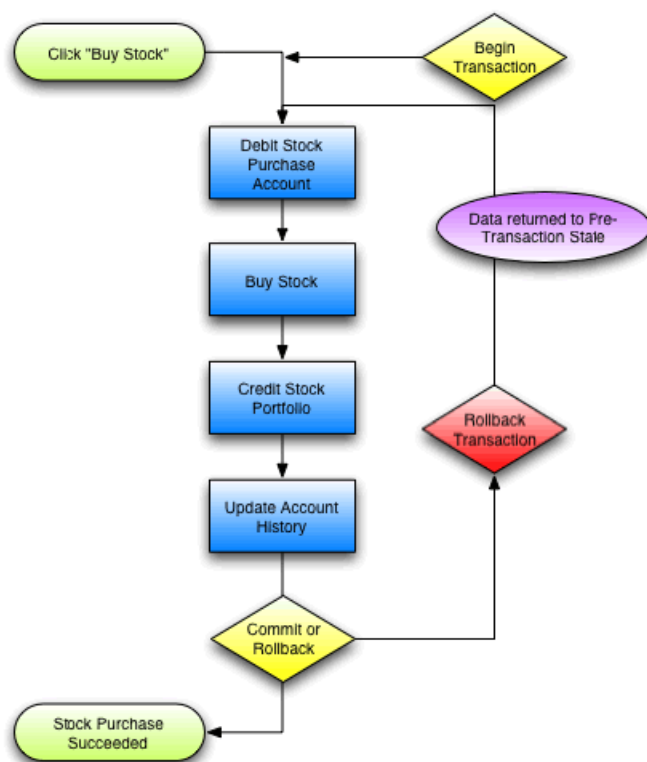
事务是J2EE中一个非常重要的部分，允许你将几个数据库调用当作一个来看待，并且在不是全部分成功时全部进行回滚操作。EJB的众多亮点之一就有它声明式的事务处理。本章演示Spring如何简化使用声明式和编程式的事务处理。

概述

Java应用程序往往以分布式的程序运行。分布式程序允许多个程序(或是客户端)同时出现，如同只有一个一样。对于用户，它们可以视为一体，即使有很多用户同时访问，更新，删除同一数据。事务保证了多个用户时数据的一致性。

事务常常出现在一个应用程序的事务层中。这是因为业务逻辑常常包括多个步骤(或数据操作)，每个步骤可以视为一个工作单元。图10.1演示了一个股票买卖系统中事务是什么样子的。

图 10.1. 股票买卖事务样例



蓝色方块表示组成一个独立单元的4个步骤。它们在同一事务中完成，或根本不执行。事务的4个属性即为ACID¹。¹

¹此定义出自Martin Fowler所著的Patterns of Enterprise Application Architecture [<http://www.martinfowler.com/books.html#eaa>]一书。

- 原子性：在这个序列中，一个事务边界内执行的行为的每个步骤必须成功的完成，否则所有操作必须回滚。
- 一致性：一个系统的资源必须处于一个一致的、清晰的状态，不管在事务开始还是结束阶段。
- 独立性：一个独立的事务的结果对于任何其它事务都是不可见的，直到事务成功的提交。
- 可靠性：任何已经提交的事务的结果是无法扭转的，换名话，“能够承受各种意外情况”。

在基于Spring的应用程序中使用事务，能够保证数据的可靠性，让你集中精力写业务逻辑。在传统的J2EE容器中，你有两种选择来管理事务边界：通过编码方式从JNDI中获取一个 `UserTransaction` [http://java.sun.com/products/jta/javadocs-1.0.1/javax/transaction/UserTransaction.html]，然后调用 `UserTransaction.begin()`，`UserTransaction.commit()` 或 `UserTransaction.rollback()`；或者，使用容器托管的事务(Container-Managed Transactions, CMT)。要使用CMT，你可能要在部署描述文件中为每个EJB方法定义事务属性(transaction attribute)，让容器决定何时开始和结束一个事务。

你可以通过定义传播行为(propagation behavior)的方式在CMT和Spring中设置事务的行为。传播行为会告诉宣传品如何处理新事务的创建，现有事务的延续。共有六种选项：

- Required：在一个当前事务中执行，如果没有现成的事务，新建一个。
- Supports：在一个当前事务中执行，如果没有现成的事务，在没有事务情况下执行。
- Mandatory：在一个当前事务中执行，如果没有现成的事务，抛出一个异常。
- Requires New：创建一个新事务，如果已经存在，挂起当前事务。
- Not Supported：在没有事务情况下执行，如果已经存在，挂起当前事务。
- Never：在没有事务情况下执行，如果已经存在一个事务，抛出异常。

默认行为是required，这常常是最合适的。除了设置传播行为外，你还可以设置一个read-only条件。很多资源可以用此方法来优化只读事务。

使用CMT时，你可以在部署描述文件中使用transaction attribute值或使用编码的方式来设置传播行为。Spring允许你通过编程的方式或是在XML中设置事务属性的方法来支持同一机制。另外，你还可以编写代码级的元数据来定义事务行为(使用Commons Attributes 和 JDK 5 Annotations)。以下列出所有可用的事务属性。

- TX_BEAN_MANAGED(表示编程式的事务划分)
- TX_NOT_SUPPORTED
- TX_REQUIRED
- TX_REQUIRES_NEW
- TX_SUPPORTS
- TX_MANDATORY
- TX_NEVER

Spring拥有一套类似的传播属性，它是 `TransactionDefinition` [http://www.springframework.org/docs/api/org.springframework.transaction.TransactionDefinition.html] 接口的一部分。

- `PROPAGATION_NOT_SUPPORTED`
[http://www.springframework.org/docs/api/org.springframework.transaction.TransactionDefinition.html#PROPAGATION_NOT_SUPPORTED]
- `PROPAGATION_REQUIRED`
[http://www.springframework.org/docs/api/org.springframework.transaction.TransactionDefinition.html#PROPAGATION_REQUIRED]
- `PROPAGATION_REQUIRES_NEW`
[http://www.springframework.org/docs/api/org.springframework.transaction.TransactionDefinition.html#PROPAGATION_REQUIRES_NEW]
- `PROPAGATION_SUPPORTS`
[http://www.springframework.org/docs/api/org.springframework.transaction.TransactionDefinition.html#PROPAGATION_SUPPORTS]
- `PROPAGATION_MANDATORY`
[http://www.springframework.org/docs/api/org.springframework.transaction.TransactionDefinition.html#PROPAGATION_MANDATORY]
- `PROPAGATION_NEVER`
[http://www.springframework.org/docs/api/org.springframework.transaction.TransactionDefinition.html#PROPAGATION_NEVER]

它还包含一个全新的选项允许你使用嵌套的事务，它有自己的独特的回滚规则。

- `PROPAGATION_NESTED`
[http://www.springframework.org/docs/api/org.springframework.transaction.TransactionDefinition.html#PROPAGATION_NESTED];
如果存在一个当前事务，在事务内执行，否则退回到 `PROPAGATION_REQUIRED` 状态。

事务有效的保证了流程的结束(或回滚)，这有利于保证被流程修改的数据。隔离层用来描述更新的数据对其它事务的可见形式。当一全事务的数据对其它事务来说是不可见的，它被称为完全事务隔离(full transaction isolation)或是可串行化事务(serializable transaction)。这只是一个概念，但却与性能关系密切。使用一个较少限制的隔离层有助于获得更好的性能。SQL标准定义了4种隔离。

- **Serializable:** 事务可以由同一结果并发执行，就像它们是分开执行的一样。
- **Repeatable Read:** 允许phantoms(错觉)。当向数据库中添加多行数据时，reader中只读取了其中的一部分。
- **Read Committed:** 允许unrepeatable reads(非重复读取)，当一个事务处于进行状态时，从同一查询语句会得到不同的结果，这时发生unrepeatable read。
- **Read Uncommitted:** 允许dirty reads(脏读取)。当另一个事务还未提交时，reader已经可以发现它的数据。

使用serializable隔离是保证数据精确性的最佳选择，但同时也是效率最差的。经过周密设计的应用程序会针对不同的事务使用不同隔离方式。当性能比数据完整性更重要时，建议使用一个更低级别的隔离。

注意

大多数性能瓶颈问题来源于没有对SQL查询进行调优，没有建立适当的索引。在为事务隔离层发愁之前，应该集中精力进行数据库调优。

下表列出了各隔离层及其矛盾的读取错误。

表 10.1. 隔离层及读取错误

Isolation Level	Phantom	Unrepeatable Read	Dirty Read
Serializable	否	否	否
Repeatable Read	是	否	否
Read Committed	是	是	否
Read Uncommitted	是	是	是

J2EE事务管理

在事务管理上，J2EE开发人员有两种选择：locally(局部)和globally(全局)。局部事务特指一种资源(如，一个数据库)，而全局事务主要依赖一个使用JTA的事务管理器。全局事务能够跨多个资源(通常是数据库)。虽然全局事务功能强大，但是局部事务在大多数情况下已经足够了。Spring让使用局部和全局事务都变得更加容易。你可以简单的利用IoC容器注入你想用的事务Transaction Manager。

在深入使用Spring进行事务管理之前，学习一下J2EE Blueprint中有关JTA和J2EE中的事务的解释。

JTA事务是由J2EE平台管理和协调的事务。一种J2EE产品要求支持JTA事务，这是在J2EE规范中定义的。JTA事务能够跨越多个组件和企业信息系统(EIS)。一个事务可以在组件间进行传递，并且在事务内通过组件访问的方式传递给企业信息系统(EIS)。例如，一个JTA事务可能包含一个访问多个enterprise bean的servlet和JSP页面，这些bean的一部分访问了一个或多个资源管理器²。

CMT是与容器全局事务管理服务交互的一种简单的途径。但你也可以通过从JNDI中获得事务认知(transaction-aware)的资源或JTA事务对象的方式，在应用中直接使用JTA。

一种与规范兼容的J2EE应用服务器必须具备一个能处理分布式事务的事务管理器。这样开发人员可以更多的关注代码编写，而不需要知道事务是如何应用到各种的资源的。事实上，虽然资源是为各种容器基准配置的，J2EE应用程序并不包含任何有关分布事务的信息。容器负责处理各种事务，并能优化单个资源的事务(可选)。

J2EE规范建议J2EE容器依照X/Open XA规范支持2 Phase Commit (2PC)协议。使用这一协议允许一个XA事务协调器操纵整修过程，而不是资源本身。

提示

Mike Spille有一篇深入分析的文章how 2 Phase Commit works in J2EE [http://www.jroller.com/page/pyrasun/20040105#xa_exposed]。

要配置资源参与一个2PC事务的过程，它们必须是XA认知(XA-ware)的。这就是说你要应该使用一个 `javax.sql.XADataSource` [http://java.sun.com/j2se/1.4.2/docs/api/javax/sql/XADataSource.html] 实现(通常由JDBC驱动程序提供)来设置一个JNDIDataSource。大多数容器允许非XA认知(non-XA-aware)数据资源参与全局事务，但是它们可能不具备2PC能力，这可能导致一些意想不到的问题。

其它用于配置XA认知(XA-ware)的资源的选择包括使用JCA [http://java.sun.com/j2ee/connector/index.jsp] 或是获得JTA内部的 `javax.transaction.TransactionManager` [http://java.sun.com/j2ee/1.4/docs/api/javax/transaction/TransactionManager.html]。你可以使用TransactionManager对象注册回应信息来获得有关全局JTA事务的信息。

²出自J2EE BluePrints [http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/transactions/transactions4.html]。

用Spring管理事务

Spring的事务API允许你更简单的使用事务。这与J2EE截然不同，在J2EE中对于编程式的事务，你必须从JNDI查找一个UserTransaction对象。使用Spring的API好处是你并没有失去J2EE给你带来的种种好处，Spring只是让它更简单。你仍然可以使用JTA(通过使用JtaTransactionManager [http://www.springframework.org/docs/api/org.springframework.transaction.jta/JtaTransactionManager.html])或是你的容器内置的事务管理器。Spring提供的各种事务管理将在Spring事务管理器一节中介绍。

事务管理器概念

Spring提供了一套丰富的基础结构来实现和控制J2EE(或是J2SE)环境中事务处理。它全部是从PlatformTransactionManager [http://www.springframework.org/docs/api/org.springframework.transaction.PlatformTransactionManager.html] 接口开始的，用图10.2表示。

图 10.2. PlatformTransactionManager接口

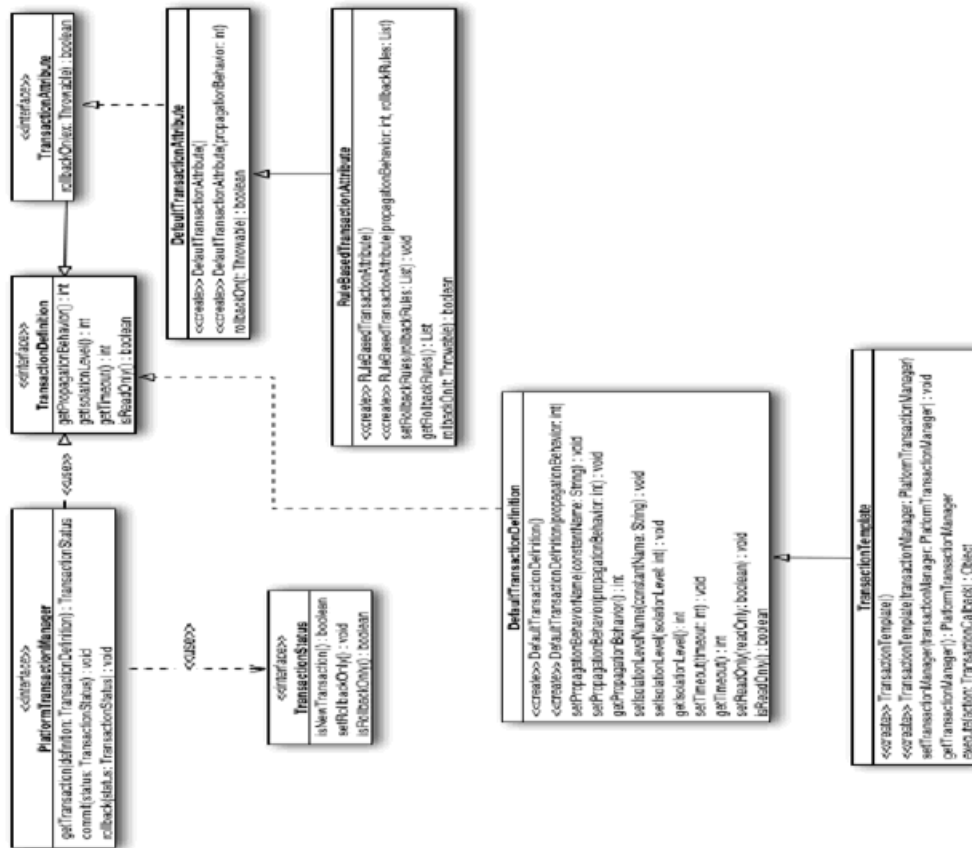


在大多数情况下，你不必直接使用这个接口。对于编程式事务，你会使用TransactionTemplate [http://www.springframework.org/docs/api/org.springframework.transaction.support/TransactionTemplate.html] 类，对于声明式事务，结合AOP使用TransactionInterceptor [http://www.springframework.org/docs/api/org.springframework.transaction.interceptor/TransactionInterceptor.html]。两种策略都会与一个PlatformTransactionManager交互，并从TransactionDefinition中获得。

TransactionDefinition接口声明了事务设置，如传播行为，隔离级别及超时设置。DefaultTransactionDefinition [http://www.springframework.org/docs/api/org.springframework.transaction.support/DefaultTransactionDefinition.html] 类是TransactionDefinition的一种带有预设值(PROPAGATION_REQUIRED, ISOLATION_DEFAULT, TIMEOUT_DEFAULT, readOnly=false)的实现。它是TransactionTemplate和DefaultTransactionAttribute [http://www.springframework.org/docs/api/org.springframework.transaction.interceptor/DefaultTransactionAttribute.html] (用于声明式定义)的基类。图10.3中的“图^{5 3}”演示了所有的接口和实现是如何与Spring的事务基础结构相吻合的。

³此“图”基于J2EE without EJB [http://www.wiley.com/WileyCDA/WileyTitle/productCd-0764558315.html] 书中的一幅。

图 10.3. Spring框架事务基础结构



编程式事务一节中会演示如何TransactionTemplate来界定Java代码中的事务。你也将学习如何在一个context文件中配置声明式事务，在源代码使用元数据。最后会为userManager bean添加一些事务属性，设置一些回滚规则，针对出现某种特定的异常。

准备练习

为了跟着做下面的练习，你必须从<http://www.sourceforge.com/downloads>下载>MyUsers Chapter 10。其源代码结构与第9章结束时一样，包含本章中你要用到的所有的jar文件。

本章中，你可以使用喜欢的数据库，大部分数据库都具备事务能力。接下来会介绍安装和配置MySQL和PostgreSQL数据库。本章下载文件是为HSQL配置的。

注意

如果你正在查找一种有嵌入功能数据库，请尝试Derby(以前的IBM的Cloudscape)，最近它成了Apache的一种开源项目。

本章中要用的三种JDBC驱动程序都放在web/WEB-INF/lib。本例中展示的截图和错误信息使用的是PostgreSQL 8.0 Beta 4。

MySQL

在第7章中，演示如何从HSQL转向MySQL。默认情况下，MySQL创建的是非事务表。要想使用事务和成功地进行回滚操作，你必须创建类型为InnoDB的表。下面的步骤演示如何为本章配置一个事务认知(transaction-aware)的MySQL数据库。

1. 要使用类型为InnoDB的表，安装或修改MySQL，如下。
 - 下载和安装MySQL的最新发布版本 [<http://www.sourcebeat.com/downloads>](写本书时，可用的4.1.7)。请确保在安装时选择了表类型InnoDB。
 - 修改现有的数据库安装设置，以便在创建数据表时默认使用InnoDB。修改文件C:\Windows\my.ini(Windows)或/usr/local/mysql/data/my.conf(UNIX/Linux)，确保包含以下内容，然后修改你的数据库服务。

```
[mysqld]
default-table-type=innodb
```

2. 如果你使用的是前一章的myusers数据库，删除并重新创建一个。
 - 命令行：使用mysql登录。执行mysql create myusers。
 - Windows GUI：使用MySQL Administrator工具(4.1中附带)。
 - OS X GUI：使用CocoaMySQL Tool。
3. 修改web/WEB-INF/classes/jdbc.properties配置，指向这一数据库，使用Hibernate的MySQLDialect。

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost/myusers?autoReconnect=true
jdbc.username=root
jdbc.password=hibernate.dialect=net.sf.hibernate.dialect.MySQLDialect
```

4. 既然你在使用Hibernate，请将hibernate.hbm2ddl.auto属性设置成create。虚拟机启动时数据库表会自动创建。要验证一切是否正常，运行ant test -Dtestcase=UserDAO。

PostgreSQL

PostgreSQL [<http://www.postgresql.org/>]是一种开源的数据库，支持事务处理(和大多数库一样)。装上它做下面的练习，执行下面的步骤。

1. 下载PostgreSQL 8.0的PGInstaller [<http://www.pgfoundry.org/projects/pginstaller>]安装程序(写本书时最发布版本是Beta 4)。对于，Marc Liyanage [<http://www.entropy.ch/software/macosx/postgresql/>]提供了一方便的安装程序，或者你可以使用Fink [<http://developer.apple.com/internet/opensource/postgres.html>]来安装。大部分Linux发行版都提供了PostgreSQL的预安装包。

2. 要创建myusers数据库，完成以下操作。

- 对于Windows用户，使用安装目录(在C:\Program Files\PostgreSQL\8.0.0-beta4\pgAdmin III)中的PG3Admin工具。添加一个server，右键点击数据库结点，创建一个名为myusers的数据库。指定用户postgres为数据库所有者，或创建一个拥有此数据库的新用户。
- 对于UNIX或Linux用户，请从命令行执行psql template1 postgres登录PostgreSQL数据库。执行create database myusers owner postgres。如果你使用Mac OS X, PostMan Query [http://www.xceltech.net/products/freeware_products.html]是个不错的工具。

3. 修改web/WEB-INF/classes/jdbc.properties，指向这个数据库。

```
jdbc.driverClassName=org.postgresql.Driver
jdbc.url=jdbc:postgres://localhost/myusers
jdbc.username=postgres
jdbc.password=postgres
hibernate.dialect=net.sf.hibernate.dialect.PostgreSQLDialect
```

注意

PostgreSQL没有设置默认密码，请确认这里与你安装时的设置一致。

4. 既然你在使用Hibernate，请将hibernate.hbm2ddl.auto属性设置成create。虚拟机启动时数据库表会自动创建。要验证一切是否正常，运行ant test -Dtestcase=UserDAO。

修改UserManager以出现回滚

为了演示事务，对本章下载文件进行了一些修改。

- hibernate.hbm2ddl.auto属性(sessionFactory bean)设置了create，这样数据库每次都会清理干净，遗留的数据不会影响测试。
- UserManagerTest仅有唯一的方法testAddUser()。

```
public void testAddUser() throws Exception {
    user = new User();
    user.setFirstName("Easter");
    user.setLastName("Bunny");
    mgr.saveUser(user);
    assertTrue(user.getId() != null);
}
```

- 任何与userManager bean相关的事务都被删除了。它是简单的POJO，无其它特殊的行为。

```
<bean id="userManager"
    class="org.appfuse.service.impl.UserManagerImpl">
```

```
<property name="userDAO"><ref bean="userDAO"/></property>
</bean>
```

上面的修改使得在手稿成功或失败更容易测试。为了证明事务已经启用，能按预期进行回滚。对于MyUsers程序进行一些修改。

1. 修改 `app_user` 表，要求 `last_name` 列是唯一的。在文件 `src/org/appfuse/model/User.hbm.xml` 中的 `lastName` 属性添加 `unique="true"`。

```
<property name="lastName" column="last_name" not-null="true" unique="true"/>
```

2. 修改 `UserManagerImpl` 类中的 `saveUser()` 方法，这样可以插入多条记录。在插入第三条记录时，应该会失败，因为它试图插入一条与第一条记录 `last name` 相同的记录。

```
public void saveUser(User user) {
    String lastName = user.getLastName();
    for (int i=0; i < 3; i++) {
        user.setId(null);
        user.setLastName(lastName + i);
        // make the last one a duplicate record
        if (i == 2) {
            user.setLastName(lastName + 0);
        }
        log.debug("entering record " + user);
        dao.saveUser(user);
    }
}
```

运行 `ant test -Dtestcase=UserManagerTest`；最后一条记录插入失败。

图 10.4. UserManagerTest测试结果

```

Cygwin
compile:
test:
[junit] DEBUG - UserManagerImpl.saveUser(34) ! entering record org.appfuse.m
odel.User@a918d5[
[junit] id=<null>
[junit] firstName=Easter
[junit] lastName=Bunny0
[junit] ]
[junit] DEBUG - UserDAOHibernate.saveUser(35) ! userId set to: 1
[junit] DEBUG - UserManagerImpl.saveUser(34) ! entering record org.appfuse.m
odel.User@a918d5[
[junit] id=<null>
[junit] firstName=Easter
[junit] lastName=Bunny1
[junit] ]
[junit] DEBUG - UserDAOHibernate.saveUser(35) ! userId set to: 2
[junit] DEBUG - UserManagerImpl.saveUser(34) ! entering record org.appfuse.m
odel.User@a918d5[
[junit] id=<null>
[junit] firstName=Easter
[junit] lastName=Bunny0
[junit] ]
[junit] WARN - JDBCExceptionReporter.logExceptions(38) ! SQL Error: 0, SQLSt
ate: null
[junit] ERROR - JDBCExceptionReporter.logExceptions(46) ! Batch entry 0 inse
rt into app_user (first_name, last_name, id) values ( was aborted. Call getNextE
xception() to see the cause.
[junit] WARN - JDBCExceptionReporter.logExceptions(38) ! SQL Error: 0, SQLSt
ate: 23505

```

现在你可以看到事务处理时出现的问题，在最后一记录上，saveUser()方法失败，但是头两记录没有受影响，还是插入了数据库中。

图 10.5. 失败事务的数据视图

	oid	id int8	first_name varchar	last_name varchar
1	24641	1	Easter	Bunny0
2	24642	2	Easter	Bunny1
.				

2 rows.

下面几节演示如何使用事务保证saveUser()中的过程是原子性的。要么全部成功，要么无一成功。

编程式事务

Spring的事务API可以让你在代码中使用事务更加简单。使用传统的J2EE，你除了使用UserTransaction对象，别无选择。从JNDI中查找这个对象不仅麻烦，而且你还要捕捉一大堆在查找和使用UserTransaction时抛出的异常。这里有一个在UserManagerImpl.saveUser()使用这种事务策略的例子。

```
public void saveUser(User user) {
```



```

UserTransaction tx = null;
try {
    InitialContext ctx = new InitialContext();
    tx = (javax.transaction.UserTransaction)
        ctx.lookup( "java:comp/UserTransaction" );
    // begin transaction
    tx.begin();
    dao.saveUser(user);
    // commit transactions
    tx.commit();
} catch (NamingException ne) {
    log.error("NamingException occurred: " + ne.getMessage());
} catch (SystemException se) {
    log.error("SystemException occurred: " + se.getMessage());
} catch (NotSupportedException nse) {
    log.error("NotSupported!");
} catch (SecurityException e) {
    log.error("How many of these are there?");
    e.printStackTrace();
} catch (IllegalStateException e) {
    log.error("Is this the longest catch block ever?");
    e.printStackTrace();
} catch (RollbackException e) {
    e.printStackTrace();
} catch (HeuristicMixedException e) {
    e.printStackTrace();
} catch (HeuristicRollbackException e) {
    e.printStackTrace();
}
}
}

```

你可以将所有的try/catch逻辑放入到一个父类或一个helper类中(或仅仅捕捉Exception), 你还是要写一大堆代码, 在一个方法中包装一个简单的事务。本例中没有包含提交失败时必须的回滚操作。最糟糕的是上述方法很难进行单元测试。你必须在容器中运行代码, 使用一些mock JNDI设置。

上面的代码还要求你为应用服务器安装和配置一个事务管理器, 否则JNDI会查找失败。大部分J2EE容器中通过EJB实现, servlet容器如Tomcat你还要额外安装一个事务管理器[<http://jotm.objectweb.org/current/jotm/doc/howto-tomcat-jotm.html>]。

TransactionTemplate

使用Spring, 不仅代码简洁, 并且不需要捕捉异常。Spring为你提供了两种不同的划分事务的方法: PlatformTransactionManager [PlatformTransactionManager]和 TransactionTemplate []。最常用的方法是使用TransactionTemplate类。这个类有一个核心的execute()方法, 使用一种回滚途径, 使你从不停的抓取和释放资源中解脱出来, 还任何try/catch/catch逻辑。

使用TransactionTemplate时, 你可以使用两类回滚: TransactionCallback [http://www.springframework.org/docs/api/org/springframework/transaction/support/TransactionCallback.html]类, 它允许你execute方法中返回一个值, 还有 TransactionCallbackWithoutResult [http://www.springframework.org/docs/api/org/springframework/transaction/support/TransactionCallbackWithoutResult.html], 它可以使用void saveUser()方法。

要使用TransactionTemplate，你必须提供一个事务管理器。事务器应该在一个context文件中配置好了，并且传递给业务对象使用。

下面步骤详细介绍了如何使用在userManagerImpl类中使用TransactionTemplate。

1. 编辑userManager bean映射，为transactionManager添加一个属性，引用transactionManager bean。

```
<bean id="userManager" class="org.appfuse.service.impl.UserManagerImpl">
  <property name="transactionManager">
    <ref bean="transactionManager"/>
  </property>
  <property name="userDAO"><ref bean="userDAO"/></property>
</bean>
```

2. web/WEB-INF/applicationContexthibernate.xml文件中已经定义了transactionManager，引用 `HibernateTransactionManager` [<http://www.springframework.org/docs/api/org.springframework.orm.hibernate.HibernateTransactionManager.html>]。
3. 在userManagerImpl中添加一个私有的transactionTemplate变量，为PlatformTransactionManager添加一个transactionManager setter。在此方法中，配置TransactionTemplate。

```
private TransactionTemplate txTemplate;

public void setTransactionManager(PlatformTransactionManager txManager) {
  this.txTemplate = new TransactionTemplate(txManager);
  // set propagation behavior, isolation level etc. template
  // we don't need to since REQUIRED is the default
}
```

4. 重构saveUser()方法，使用此事务模板。

```
public void saveUser(final User user) {
  txTemplate.execute(new TransactionCallbackWithoutResult() {
    public void doInTransactionWithoutResult(TransactionStatus status) {
      // do business logic
      String lastName = user.getLastName();
      for (int i = 0; i < 3; i++) {
        // can't just set user.id to null or Hibernate
        // complains that identifier was changed
        User u = new User();
        u.setFirstName(user.getFirstName());
        u.setLastName(user.getLastName() + i);
        // make the last one a duplicate record
        if (i == 2) {
          u.setLastName(lastName + 0);
        }
      }
    }
  });
}
```

```

        if (log.isDebugEnabled()) {
            log.debug("entering record " + u);
        }
        dao.saveUser(u);
    }
}
});
}

```

运行 `ant test -Dtestcase=UserManagerTest`，你会看到与图10.4类似的错误信息，但是一个事务会促使此方法执行 `execute()` 一个要么全有要么全无的过程。`app_user`表应该是空的。

提示

要打开更多有关事务处理的日志，在 `web/WEB-INF/classes/log4j.xml` 文件中添加下面的代码。

```

<logger name="org.springframework.transaction">
    <level value="DEBUG"/>
</logger>

```

PlatformTransactionManager

另一种办公室事务的方法是直接使用 `PlatformTransactionManager`。这种方法的问题是你必须管理逻辑，在抛出异常时回滚事务。要使用 `PlatformTransactionManager`，执行下面的步骤。

1. 和 `TransactionTemplate` 例子一样，在你的类中设置一个 `PlatformTransactionManager` 变量。

```

private PlatformTransactionManager transactionManager;

public void setTransactionManager(PlatformTransactionManager txManager) {
    this.transactionManager = txManager;
}

```

2. 重构 `saveUser()` 方法，这样会捕捉一个 `DataAccessException`，并作相应的回滚。

```

public void saveUser(final User user) {
    DefaultTransactionDefinition txDef =
        new DefaultTransactionDefinition();

    // set propagation behavior, isolation level etc. template
    // we don't need to since REQUIRED (the default) is good enough
    TransactionStatus status =
        transactionManager.getTransaction(txDef);
}

```

```

try {
    // do business logic
    String lastName = user.getLastName();
    for (int i = 0; i < 3; i++) {

        // can't just set user.id to null or Hibernate
        // complains that identifier was changed
        User u = new User();
        u.setFirstName(user.getFirstName());
        u.setLastName(user.getLastName() + i);

        // make the last one a duplicate record
        if (i == 2) {
            u.setLastName(lastName + 0);
        }
        if (log.isDebugEnabled()) {
            log.debug("entering record " + u);
        }
        dao.saveUser(u);
    }
} catch (DataAccessException ex) {
    transactionManager.rollback(status);
    throw ex;
}
transactionManager.commit(status);
}

```

使用TransactionTemplate代码较少，PlatformTransactionManager允许你使用任何现在异常处理逻辑。使用TransactionTemplate是推荐的方法。两种方法都允许你在一个J2EE容器中简易的测试你的代码。

编程式事务很好，却要写更多的代码。在上一个例子中，业务逻辑花了大约20行代码，整个方法共34行代码。使用声明式的或是源码级的元数据来指定事务属性，允许你将方法裁剪至仅包含业务逻辑代码，而且不需要具备事务基础。

声明式事务

声明式事务描述的是使用XML或是元数据来指定事务属性的能力。与编码的直接在代码中包装行为的方式不同，你声明的是行为。这种方法的优点是，在大程序上，代码并不知道它的事务行为。它还可以减少输入，提高跨越多个对象的事务的复用性。

AOP with TransactionProxyFactoryBean

在Spring中使用声明式事务最常用的方法，就是用一个 `TransactionProxyFactoryBean` [<http://www.springframework.org/docs/api/org.springframework.transaction.interceptor/TransactionProxyFactoryBean.html>] 来包装你的代码。它的javadoc中解释了它的功能。

这个类可能是声明式事务隔离一种典型的例子：也就是，用一个事务代理包装一个(singleton) 目标对象，代理所目标对象要实现的接口。

你使用的是上一章中的技巧。接下来完成以下步骤。

1. 修改`UserManagerImpl.saveUser()`方法，包含唯一的业务逻辑。你还可以删除任何`TransactionManager/Template`相关的变量和方法。

```
public void saveUser(final User user) {

    // do business logic
    String lastName = user.getLastName();
    for (int i = 0; i < 3; i++) {

        // can't just set user.id to null or Hibernate
        // complains that identifier was changed
        User u = new User();
        u.setFirstName(user.getFirstName());
        u.setLastName(user.getLastName() + i);
        // make the last one a duplicate record
        if (i == 2) {
            u.setLastName(lastName + 0);
        }
        if (log.isDebugEnabled()) {
            log.debug("entering record " + u);
        }
        dao.saveUser(u);
    }
}
```

2. 修改`userManager` bean定义，用`TransactionProxyFactoryBean`包装`UserManagerImpl`，`target`属性指向要应用事务行为的那个类。

```
<bean id="userManager"
      class="org.springframework.transaction.interceptor.
        TransactionProxyFactoryBean">
  <property name="transactionManager"><ref bean="transactionManager"/>
</property>
  <property name="target"><ref bean="userManagerTarget"/></property>
  <property name="transactionAttributes">
    <props>
      <prop key="save*">PROPAGATION_REQUIRED</prop>
      <prop key="remove*">PROPAGATION_REQUIRED</prop>
      <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
    </props>
  </property>
  <bean id="userManagerTarget" class="org.appfuse.service.impl.
    UserManagerImpl">
    <property name="userDAO"><ref bean="userDAO"/></property>
  </bean>
```

注意

如果你看到这样一个错误，“只读模式下不能进行写操作”，请确保你没有对方法设置只读属性。

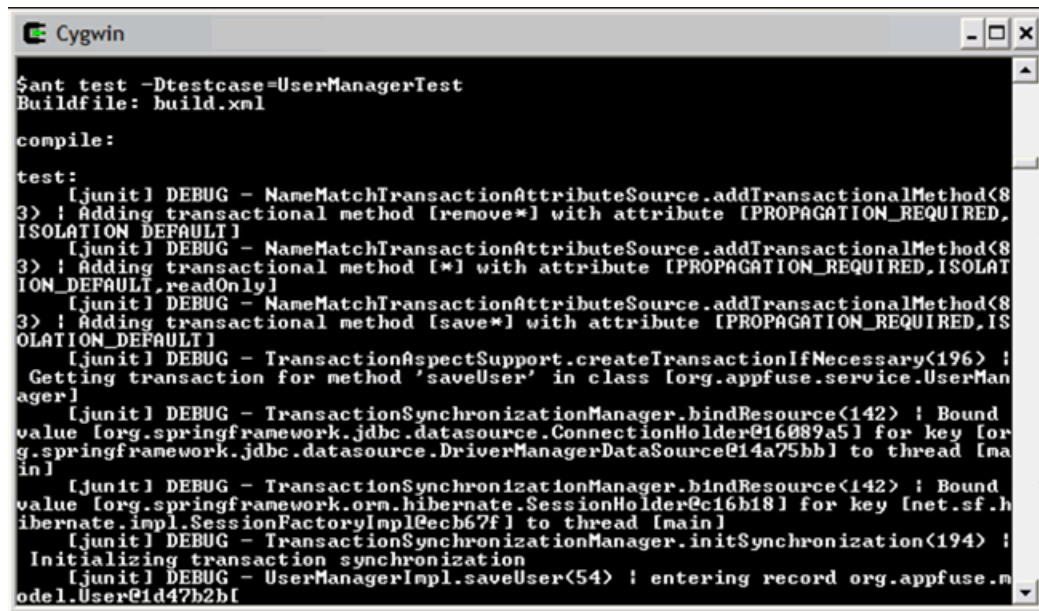
`transactionAttribute` 属性是通过在 `NameMatchTransactionAttributeSource` [<http://www.springframework.org/docs/api/org.springframework.transaction.interceptor.NameMatchTransactionAttributeSource.html>] 类中指定一个 Properties 格式进行设置的。这种格式非常灵活，可以看出传播行为，隔离级别和其它属性如何映射到一个 `TransactionDefinition` [<http://www.springframework.org/docs/api/org.springframework.transaction.TransactionDefinition.html>]。

本例的问题是，`userManagerTarget` 暴露给了开发人员，有可能某人使用 `userManagerImpl` 类，确没有事务行为。要避免这一问题，你可以将 `userManagerTarget` 放入一个匿名内部类中，这样任何人都无法从 `ApplicationContext` 中获取它。下面的例子演示这种做法。

```
<bean id="userManager"
  class="org.springframework.transaction.interceptor.
    TransactionProxyFactoryBean">
  <property name="transactionManager">
    <ref bean="transactionManager"/>
  </property>
  <property name="target">
    <bean class="org.appfuse.service.impl.UserManagerImpl">
      <property name="userDAO"><ref bean="userDAO"/></property>
    </bean>
  </property>
  <property name="transactionAttributes">
    <props>
      <prop key="save*">PROPAGATION_REQUIRED</prop>
      <prop key="remove*">PROPAGATION_REQUIRED</prop>
      <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
    </props>
  </property>
</bean>
```

`transactionAttributes` 属性可以定义传播行为，隔离级别，只读标志和回滚属性。使用上面的设置会导致使用 `ISOLATION_DEFAULT` 隔离级别，如图 10.6 所示。

图 10.6. UserManagerTest的日志



```

Cygwin
$ant test -Dtestcase=UserManagerTest
Buildfile: build.xml

compile:

test:
[junit] DEBUG - NameMatchTransactionAttributeSource.addTransactionalMethod(8
3) : Adding transactional method [remove*] with attribute [PROPAGATION_REQUIRED,
ISOLATION_DEFAULT]
[junit] DEBUG - NameMatchTransactionAttributeSource.addTransactionalMethod(8
3) : Adding transactional method [*] with attribute [PROPAGATION_REQUIRED,ISOLAT
ION_DEFAULT,readOnly]
[junit] DEBUG - NameMatchTransactionAttributeSource.addTransactionalMethod(8
3) : Adding transactional method [save*] with attribute [PROPAGATION_REQUIRED,IS
OLATION_DEFAULT]
[junit] DEBUG - TransactionAspectSupport.createTransactionIfNecessary(196) :
Getting transaction for method 'saveUser' in class [org.appfuse.service.UserMan
ager]
[junit] DEBUG - TransactionSynchronizationManager.bindResource(142) : Bound
value [org.springframework.jdbc.datasource.ConnectionHolder@16089a5] for key [or
g.springframework.jdbc.datasource.DriverManagerDataSource@14a75bb] to thread [ma
in]
[junit] DEBUG - TransactionSynchronizationManager.bindResource(142) : Bound
value [org.springframework.orm.hibernate.SessionHolder@ecb67f] to thread [main]
[junit] DEBUG - TransactionSynchronizationManager.initSynchronization(194) :
Initializing transaction synchronization
[junit] DEBUG - UserManagerImpl.saveUser(54) : entering record org.appfuse.m
odel.User@1d47b2b[

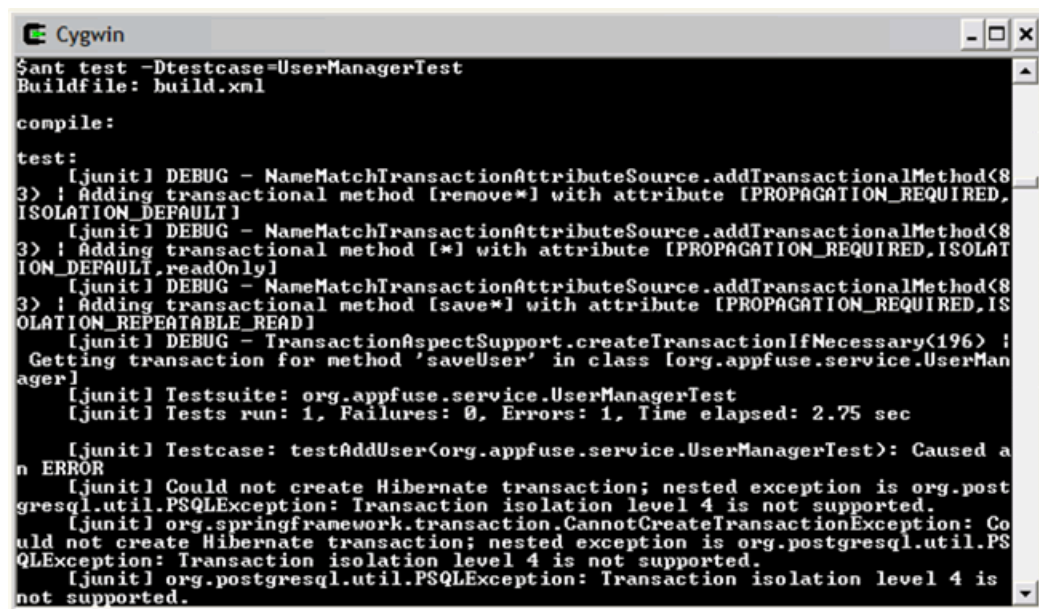
```

添加5种级别中一种到一个方法的事务定义中，你很容易修改隔离级别。

```
<prop key="save*">PROPAGATION_REQUIRED,ISOLATION_REPEATABLE_READ</prop>
```

如果你的Transaction Manager支持指定的隔离级别，运行不会有什么问题。HibernateTransactionManager不支持Repeatable Read(但经支持Serializable)。

图 10.7. Hibernate不支持Repeatable Read



```

Cygwin
$ant test -Dtestcase=UserManagerTest
Buildfile: build.xml

compile:

test:
[junit] DEBUG - NameMatchTransactionAttributeSource.addTransactionalMethod(8
3) : Adding transactional method [remove*] with attribute [PROPAGATION_REQUIRED,
ISOLATION_DEFAULT]
[junit] DEBUG - NameMatchTransactionAttributeSource.addTransactionalMethod(8
3) : Adding transactional method [*] with attribute [PROPAGATION_REQUIRED,ISOLAT
ION_DEFAULT,readOnly]
[junit] DEBUG - NameMatchTransactionAttributeSource.addTransactionalMethod(8
3) : Adding transactional method [save*] with attribute [PROPAGATION_REQUIRED,IS
OLATION_REPEATABLE_READ]
[junit] DEBUG - TransactionAspectSupport.createTransactionIfNecessary(196) :
Getting transaction for method 'saveUser' in class [org.appfuse.service.UserMan
ager]
[junit] Testsuite: org.appfuse.service.UserManagerTest
[junit] Tests run: 1, Failures: 0, Errors: 1, Time elapsed: 2.75 sec

[junit] Testcase: testAddUser(org.appfuse.service.UserManagerTest): Caused a
n ERROR
[junit] Could not create Hibernate transaction; nested exception is org.post
gresql.util.PSQLException: Transaction isolation level 4 is not supported.
[junit] org.springframework.transaction.CannotCreateTransactionException: Co
uld not create Hibernate transaction; nested exception is org.postgresql.util.PS
QLException: Transaction isolation level 4 is not supported.
[junit] org.postgresql.util.PSQLException: Transaction isolation level 4 is
not supported.

```

我推荐使用ISOLATION_DEFAULT(使用数据库默认特性)。

回滚规则和异常

除了设置传播行为和只读标示之外，你还可以要进行回滚的地方设置条件。这是使用CMT的一个优点，这也是你唯一可以设置`setRollbackOnly()`的地方。

要指定回滚规则，将异常名称添加到事务属性中。减号(-)前缀表示强加一个回滚，加号(+)前缀表示任何情况下都提交事务。默认情况下，任何运行时异常会导致事务回滚。为了表示在业务逻辑抛出`UserExistsException`时继续提交事务，将`save*`方法的行为修改成如下：

```
<prop key="save*">PROPAGATION_REQUIRED,+UserExistsException</prop>
```

Transaction Template Bean

如果要给多个对象添加类似的事务属性，那么目前针对`userManager` bean的配置会变得臃肿不堪。为了避免这个问题，你可以使用Transaction Template Bean来继承这个bean的所有bean指定属性。前一章使用了这种策略，这在Colin Sampaleanu's weblog [<http://blog.exis.com/colin/archives/2004/07/31/concise-transaction-definitions-spring-11/>]作了详细的解释。这也是应用程序中推荐的策略。

在事务中使用template bean包括两步：

1. 在context文件中创建一个bean定义，作为其它bean的模板。请确保这个bean的定义中包含一个`abstract="true"`属性。

```
<bean id="txProxyTemplate" abstract="true"
      class="org.springframework.transaction.interceptor.
      TransactionProxyFactoryBean">
  <property name="transactionManager">
    <ref bean="transactionManager"/>
  </property>
  <property name="transactionAttributes">
    <props>
      <prop key="save*">PROPAGATION_REQUIRED</prop>
      <prop key="remove*">PROPAGATION_REQUIRED</prop>
      <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
    </props>
  </property>
</bean>
```

2. 当你创建一个应用此模板的bean时，使用`parent`属性来引用此模板的id。然后在`target`属性中将这个类定义为一个内部bean。

```
<bean id="userManager" parent="txProxyTemplate">
  <property name="target">
    <bean class="org.appfuse.service.impl.UserManagerImpl">
      <property name="userDAO"><ref bean="userDAO"/></property>
    </bean>
  </property>
```



```
</bean>
```

TransactionAttributeSource

另一种配置声明式事务的是指定一个引用了NameMatchTransactionAttributeSource类并定义了方法和它们行为的bean。这种策略不及template bean简洁，因为它要求用TransactionProxyFactoryBean来包装所有的bean。

要使用这种bean，完成以下步骤：

1. 创建一个bean定义，描述事务属性。

```
<bean name="txAttributes"
class="org.springframework.transaction.interceptor.
NameMatchTransactionAttributeSource">
  <property name="properties">
    <props>
      <prop key="save*">PROPAGATION_REQUIRED</prop>
      <prop key="remove*">PROPAGATION_REQUIRED</prop>
      <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
    </props>
  </property>
</bean>
```

2. 创建一个包装事务的bean定义，在transactionAttributeSource属性中引用这个bean。

```
<bean id="userManager"
class="org.springframework.transaction.interceptor.
TransactionProxyFactoryBean">
  <property name="transactionManager">
    <ref bean="transactionManager"/>
  </property>
  <property name="target">
    <bean class="org.appfuse.service.impl.UserManagerImpl">
      <property name="userDAO">
        <ref bean="userDAO"/>
      </property>
    </bean>
  </property>
  <property name="transactionAttributeSource">
    <ref bean="txAttributes"/>
  </property>
</bean>
```

更多有关此策略的信息请参考Chris
[http://www.cwinters.com/news/display/?news_id=3149]。

Winter's weblog

BeanNameAutoProxyCreator

最后还有一种配置声明式事务的方法，使用 `BeanNameAutoProxyCreator` [<http://www.springframework.org/docs/api/org.springframework.aop.framework.autoproxy/BeanNameAutoProxyCreator.html>]。这种策略允许你指定一个bean名称列表，可以应用一系列事务属性。在Spring的参考文档 [<http://www.springframework.org/docs/reference/transaction.html#d0e4560>]中有一个样例配置。除了使用这种策略，你还可能使用代码级元数据来动态创建代理。

Source-Level Metadata(代码级元数据)

除了在XML中指定事务行为，你也可以直接在类中声明。在Spring中你可以通过两种途径实现：使用Commons Attributes [<http://jakarta.apache.org/commons/attributes/>]或使用JDK 5 Annotations [<http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html>]。两种有异曲同工之效，便Commons Attributes可以在JDK 1.2及以后的版本上运行。这种策略可以让你避免TransactionProxyFactoryBean。包含事务属性的bean会被自动拦截，用声明式事务管理包装。

使用Commons Attributes

Commons Attributes要求你将编译属性作为整个构建过程的一部分。要在你的项目中使用Commons Attributes，你应该添加几jar文件到你的classpath中。

- commons-attributes-api.jar
- commons-attributes-compiler.jar
- commons-collections.jar
- xjavadoc-1.1.jar

这些jar文件已经包含在本章的下载文件中。要在userManager bean上使用Commons Attributes，完成以下步骤。

1. 在文件build.xml添加一个compileAttributes target。

```
<target name="compileAttributes"
  description="compiles classes enhanced with commons attributes">
  <taskdef resource="org/apache/commons/attributes/anttasks.properties">
    <classpath refid="classpath"/>
  </taskdef>
  <!-- Compile to a temp directory -->
  <mkdir dir="${build.dir}/attributes"/>
  <attribute-compiler destdir="${build.dir}/attributes">
    <fileset dir="${src.dir}" includes="**/*ManagerImpl.java"/>
  </attribute-compiler>
</target>
```

2. 修改compile target，让它依赖compileAttributes，包含进目录\${build.dir}/attributes，将它作为一个源目录。

```
<target name="compile" depends="compileAttributes"
```

```

description="Compile main source tree java files">
<mkdir dir="${build.dir}/classes"/>
<javac destdir="${build.dir}/classes" debug="true"
deprecation="false" optimize="false" failonerror="true">
<src path="${src.dir}"/>
<src path="${build.dir}/attributes"/>
<classpath refid="classpath"/>
</javac>
...
</target>

```

3. 新建一个context文件，定义几个你打算启用元数据事务属性的bean。创建文件web/WEB-INF/applicationContext-metadata.xml，填充以下XML代码。

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
<!--
This bean is a post-processor that will automatically apply
relevant advisors any bean in child factories.
-->
<bean id="autoProxyCreator" class="org.springframework.aop.framework.
autoproxy.DefaultAdvisorAutoProxyCreator"/>
<!--
This bean specifies the implementation to use for reading attributes
-->
<bean id="transactionAttributeSource" class="org.springframework.
transaction.interceptor.AttributesTransactionAttributeSource"
autowire="constructor"/>
<bean id="transactionInterceptor" class="org.springframework.
transaction.interceptor.TransactionInterceptor"
autowire="byType"/>
<!--
AOP advisor that will provide declarative transaction
management on attributes. It's possible to add arbitrary
custom Advisor implementations as well, and they will also
be evaluated and applied automatically.
-->
<bean id="transactionAdvisor" class="org.springframework.transaction.
interceptor.TransactionAttributeSourceAdvisor"
autowire="constructor"/>
<!-- Commons Attributes Attributes implementation. -->
<bean id="attributes"
class="org.springframework.metadata.commons.CommonsAttributes"/>
</beans>

```

在saveUser()方法的javadoc中添加一个DefaultTransactionAttribute元素。

```

/**
 *
 * @org.springframework.transaction.interceptor.DefaultTransactionAttribute()
 */

```

你也可以在类这一级设置这一属性，这样可以在类中的所有方法上应用此默认属性(正如前面讨论的那样)。如果你不想指定完整有效的类名，你可以导入这个类，仅使用它的名称但是，当今大多数IDE可能会将它识别为“unused import”，如果你组织导入时，会将其删除掉。

4. 修改userManager bean，使其与本章开始的配置相同。

```

<bean id="userManager" class="org.appfuse.service.impl.UserManagerImpl">
  <property name="userDAO"><ref bean="userDAO"/></property>
</bean>

```

5. 运行 `ant test -Dtestcase=UserManagerTest`，验证事务在回滚提交后，表 `app_user` 是否为空。

有关传播行为和其它属性的样例，请参考 `TxClassImpl.java`
[\[http://monkeymachine.co.uk/spring/xref-test/org.springframework.aop.framework.autoproxy/metadata/TxClassImpl.html\]](http://monkeymachine.co.uk/spring/xref-test/org.springframework.aop.framework.autoproxy/metadata/TxClassImpl.html)
 和 `TxClsWithClassAttribute.java`
[\[http://monkeymachine.co.uk/spring/xref-test/org.springframework.aop.framework.autoproxy/metadata/TxClsWithClassAttribute.html\]](http://monkeymachine.co.uk/spring/xref-test/org.springframework.aop.framework.autoproxy/metadata/TxClsWithClassAttribute.html)
 的源代码。

使用JDK 5.0 Annotations

如果你在用JDK 5.0，你也可以使用JDK 5 Annotations。Spring的CVS中有一个事务管理的annotations的实现。你可以从sample模块，在tiger目录中找到它。为了方便使用，此目录的class已经打了包，并放在 `web/WEB-INF/lib` 目录中，文件名为 `spring-annotations.jar`。

1. 删除文件 `build.xml` 中 `compile target` 的依赖关系。
2. 修改文件 `applicationContext-metadata.xml` 中的 `transactionAttributeSource`，使用 `annotations` 实现。

```

<bean id="transactionAttributeSource"
  class="org.springframework.transaction.annotations.
    AnnotationsTransactionAttributeSource"
  autowire="constructor"/>

```

3. 修改 `attributes` bean，使用 `annotations` 实现。

```

<bean id="attributes"
  class="org.springframework.metadata.annotations.
    AnnotationsAttributes"/>

```

4. 在`saveUser()`方法中添加一个注释(annotation)。Annotations不在javadoc标签中。

```
@org.springframework.transaction.annotations.Transactional
public void saveUser(final User user) {
```

5. 运行`ant test -Dtestcase=UserManagerTest`, 验证表`app_user`是否为空。

和Commons Attributes一样, 如果你想在所有的方法上应用, 你也可以在类一级上使用前面的annotation。有关各种属性使用的源代码, 请参考Spring CVS库中的<http://svn.apache.org/viewsvn/view.cgi?view=rev&rev=104848>。

Spring事务管理器

本章探讨了在Spring中使用事务的多种方法。所有的例子中, 你要在context文件中配置一个PlatformTransactionManager的实现。本节探讨目前可用的Transaction Manager实现, 以及该何时该使用它。

DataSourceTransactionManager

当使用一个单独的JDBC DataSource时, DataSourceTransactionManager [http://www.springframework.org/docs/api/org/springframework/jdbc/datasource/DataSourceTransactionManager.html] 非常有用。它将DataSource中的一个JDBC Connection绑定到线程上。它支持自定义隔离级别和超时。如果你在使用JDBC 3.0, DataSourceTransactionManager通过JDBC 3.0 Savepoints [http://www-128.ibm.com/developerworks/java/library/j-jdbcnew/#h27445]支持嵌套事务。

当使用唯一的数据库时, 这种实现可以作为JtaTransactionManager的一种备选方案。在这种实现与JTA之间切换, 仅仅是配置的问题。如果你要支持多个数据库, 或者你想使用应用服务器的事务管理器, 你只需要JTA Manager。

我推荐你在iBATIS和Spring JDBC中使用这种事务管理器。

HibernateTransactionManager

HibernateTransactionManager [http://www.springframework.org/docs/api/org/springframework/orm/hibernate/HibernateTransactionManager.html] 是为使用唯一的Hibernate SessionFactory [http://www.hibernate.org/hib_docs/api/net/sf/hibernate/SessionFactory.html]而设计的。它factory中取得的一个Session [http://www.hibernate.org/hib_docs/api/net/sf/hibernate/Session.html]绑定到线程上。Hibernate辅助类(SessionFactoryUtils [http://www.springframework.org/docs/api/org/springframework/orm/hibernate/SessionFactoryUtils.html]和HibernateTemplate [http://www.springframework.org/docs/api/org/springframework/orm/hibernate/HibernateTemplate.html])能够找到线程绑定的session, 并自动参与事务。和DataSourceTransactionManager一样, 这种实现支持JDBC 3.0 Savepoints, 并允许你设置自定义隔离级别和超时。

如果你需要多个资源之间的跨事务的支持, 你可以修改配置, 使用JtaTransactionManager。你也可以使用Hibernate JCA Connector, 直接进行容器集成。不幸的是, 好象还没有关于使用和配置Hibernate JCA的文档。

JdoTransactionManager

`J d o T r a n s a c t i o n M a n a g e r`
[\[http://www.springframework.org/docs/api/org/springframework/orm/jdo/JdoTransactionManager.html\]](http://www.springframework.org/docs/api/org/springframework/orm/jdo/JdoTransactionManager.html)
 是为使用唯一的 `J D O` `P e r s i s t e n c e M a n a g e r F a c t o r y`
[\[http://java.sun.com/products/jdo/javadocs/javax/jdo/PersistenceManagerFactory.html\]](http://java.sun.com/products/jdo/javadocs/javax/jdo/PersistenceManagerFactory.html)而设计的。和前面两种策略类似，这种实现将factory中取得的一个 `P e r s i s t e n c e M a n a g e r`
[\[http://java.sun.com/products/jdo/javadocs/javax/jdo/PersistenceManager.html\]](http://java.sun.com/products/jdo/javadocs/javax/jdo/PersistenceManager.html)绑定到线程。
`P e r s i s t e n c e M a n a g e r F a c t o r y U t i l s`
[\[http://www.springframework.org/docs/api/org/springframework/orm/jdo/PersistenceManagerFactoryUtils.html\]](http://www.springframework.org/docs/api/org/springframework/orm/jdo/PersistenceManagerFactoryUtils.html)
 和 `J d o T e m p l a t e`
[\[http://www.springframework.org/docs/api/org/springframework/orm/jdo/JdoTemplate.html\]](http://www.springframework.org/docs/api/org/springframework/orm/jdo/JdoTemplate.html)能够识别线程绑定的事务，并参与事务。当JDO是主要的事务性数据访问方法时，这种实现非常有效。它支持JDBC 3.0 Savepoints，只要你的JDBC驱动程序支持它们。

如果你想在多个资源之间使用事务，你必须使用 `JtaTransactionManager`。但是，你必须使用配置你的JDO实现，参与JTA事务。

JtaTransactionManager

如果你要处理分布式事务或一个J2EE上(通过JCA注册)的事务，你应该使用 `J t a T r a n s a c t i o n M a n a g e r`
[\[http://www.springframework.org/docs/api/org/springframework/transaction/jta/JtaTransactionManager.html\]](http://www.springframework.org/docs/api/org/springframework/transaction/jta/JtaTransactionManager.html)。对于单个资源，前面提及的事务管理器应该可以满足你的要求。

使用JTA实现，事务同步默认是启用的。这样数据访问类会在事务提交时，将资源注册为关闭状态。当在一个事务中打开资源时，Spring的JDBC，Hibernate和JDO支持类会执行同样的注册。标准的JTA并不能保证打开的资源会被关闭，所以这是Spring JTA实现的一个亮点。有关高级使用的信息，包括在特定应用服务器中如何挂起事务，请参考这人类的 `javadoc`
[\[http://www.springframework.org/docs/api/org/springframework/transaction/jta/JtaTransactionManager.html\]](http://www.springframework.org/docs/api/org/springframework/transaction/jta/JtaTransactionManager.html)。

PersistenceBrokerTransactionManager

在数据访问层中使用OJB时， `P e r s i s t e n c e B r o k e r T r a n s a c t i o n M a n a g e r`
[\[http://www.springframework.org/docs/api/org/springframework/orm/ojb/PersistenceBrokerTransactionManager.html\]](http://www.springframework.org/docs/api/org/springframework/orm/ojb/PersistenceBrokerTransactionManager.html)
 就可以派上用场。它将一个OJB `P e r s i s t e n c e B r o k e r`由指定的键名绑定到线程上。`O j b F a c t o r y U t i l s`
[\[http://www.springframework.org/docs/api/org/springframework/orm/ojb/OjbFactoryUtils.html\]](http://www.springframework.org/docs/api/org/springframework/orm/ojb/OjbFactoryUtils.html)和 `P e r s i s t e n c e B r o k e r T e m p l a t e`
[\[http://www.springframework.org/docs/api/org/springframework/orm/ojb/PersistenceBrokerTemplate.html\]](http://www.springframework.org/docs/api/org/springframework/orm/ojb/PersistenceBrokerTemplate.html)
 可以找到线程绑定 `persistence brokers`，并自动参与事务。和其它实现一样，在一个事务中需要访问多个资源时，你应该使用 `JtaTransactionManager`。

本章小结

在Spring中能够简单有效的管理事务是其众多优点之一。它不仅简单，而且还提供了多种选择。你可以减轻从JNDI中查找 `UserTransaction`带来的痛苦，而直接使用 `PlatformTransactionManager`。为了减少和消除处理异常的必要操作，你可使用 `TransactionTemplate`及其回调操作，你可以在一个XML中使用声明式事务处理。另外，利用 `Commons Attributes`和 `JDK 5.0 Annotations`，你还可以使用源码级的元数据。

J2EE的最大优点之一，就是能够创建事务支持的企业系统。在Spring出现之前，你必须在一个成熟的应用服务器中运行你的应用来实现事务，如果想使用声明式事务，你不得不使用EJB。使用Spring，你可以为任何POJO定义事务，在进行扩展时，你也有选择。对于大多数应用程序，使用单个的DataSource及其相应的事务管理器完全可以满足你的需求。一个强大的数据库能够提高跨越多个数据库的性能。然而，你使用多个DataSource时，你仍然有选择的余地，只要简单的修改事务管理器实现，使用JtaTransactionManager。

第 11 章 Web 框架集成

Spring 与四种流行的框架集成

Spring 拥有自己的 web 框架，但它也能和其它框架很好的集成。这样你可以权衡现有的知识，使用 Spring 管理你的业务对象和数据层。本章探讨 Spring 与 4 种流行的框架集成：JSF，Struts，Tapestry 和 WebWork。

概述

选择一种 web 框架在 Java 社区是一种争论不休的话题。很多开发人员精通某种特定的框架，在哪个框架最好这一点存在的分歧也是与日俱增。但是，如何你是一名 Java web 开发人员，而你仅懂一种 Web 框架，你低估了自己。你限制自己的机遇，你应该进行多样化投资。通过学习多种框架，你的技能地位会显得更有价值，在如何用自己首选的框架把事情变得更加简单这一点，你会赢得更多的关注。

虽然 Spring 拥有自己的框架，但它也能很多其它框架。它内置支持 JSF 和 Struts，Spring 与 Tapestry 和 WebWork 集成也很简单。本章会简短的解释每种框架及如何与 Spring 集成。还向你演示了如何配置各种框架的验证，并涉及了程序员的测试策略。本章还介绍了视图的选择(例如，Velocity 和 JSP)，还演示了如何转换复杂类型，如日期。

章节练习

本章与前一章略有不同，代码是基于 Equinox 1.2，而不是在 MyUsers 程序上建构的。这样做的好处是 web/WEB-INF/lib 目录中不会包含大量用不到的 jar 文件，源码树中仅包含要用到的代码。

JSF，Tapestry 和 WebWork 小节都例举了如何构建一个简单的应用程序，其拥有与第 2 章和第 4 章同样的功能。基本上说，仅仅是对一个有主/细节页面的数据库进行 CRUD 操作。

样例向你演示了如何进行测试，实现验证，进行国际化设置和显示成功消息。但没有涉及到验证，授权和上传文件。如果你对某些主题感兴趣，请自选研究 AppFuse [<http://appfuse.dev.java.net/>]，它包含了所有特性的实现。

要完成本章中 JSF，Tapestry 和 WebWork 练习，从 <http://sourcebeat.com/downloads> 上下载各自的捆绑包。每个包含一个 Equinox 1.2 的完整版本。Struts 一节没有捆绑包或练习，因为在第 2 章已经介绍了。

下载文件向你演示了如何运用每种框架从头创建一个应用程序。本章并不是 JSF，Struts，Tapestry 和 WebWork 的一个完整参考。主要目的是向你介绍每种框架，以及在 Spring 中如何使用它们。各个小节还包含了额外的资源，以帮助你深入学习各个框架。本章还包含了使用它们进行开发时一些技巧。

在 web 应用程序中集成 Spring

在现有的 web 程序中集成 Spring 的最简方法是在你的 web.xml 中声明一个 ContextLoaderListener，使用一个 contextConfigLocation <context-param> 设置要加载哪些 context 文件。本章的下载文件中已经包含了这一配置，这里重温一下。

```
<context-param>
```

```
<context-param>
```



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

```
<listener>
```

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

注意

Listener是在2.3版中新加进Servlet API的。你如果使用的是一个Servlet 2.2的容器，你也可以用ContextLoaderServlet [http://www.springframework.org/docs/api/org/springframework/web/context/ContextLoaderServlet.html] 实现同样的功能。

如果你没有指定contextConfigLocation context 参数，ContextLoaderListener会查找/WEB-INF/applicationContext.xml进行加载。一旦加载完成，Spring会创建一个基于bean定义的WebApplicationContext [http://www.springframework.org/docs/api/org/springframework/web/context/WebApplicationContext.html] 对象，并将它放进ServletContext中。

所有的web框架都是基于Servlet API。所以可以用下面的代码来获取Spring创建的ApplicationContext。

```
WebApplicationContext ctx =
    WebApplicationContextUtils.getWebApplicationContext(servletContext);
```

从特定的web框架中获取ServletContext的一种通用方法是使用下面的代码。

```
ServletContext servletContext =
    request.getSession().getServletContext();
```

WebApplicationContextUtils [http://www.springframework.org/docs/api/org/springframework/web/context/support/WebApplicationContextUtils.html] 类是为了方便，所以你没有必要记住ServletContext属性的名称。如果键为WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE的对象不存在，getWebApplicationContext()会返回一个null。为了避免在应用程序中抛出NullPointerException，更好的方法是使用getRequiredWebApplicationContext()方法。当找不到ApplicationContext时，此方法会抛出一个异常。

注意

如果你已经正确的配置好了应用程序，ApplicationContext已经存在于ServletContext..

一旦你引用了 `WebApplicationContext`，你可以通过 `name` [[http://www.springframework.org/docs/api/org.springframework.beans.factory.BeanFactory.html#getBean\(java.lang.String\)](http://www.springframework.org/docs/api/org.springframework.beans.factory.BeanFactory.html#getBean(java.lang.String))] 和 `type` [[http://www.springframework.org/docs/api/org.springframework.beans.factory.ListableBeanFactory.html#getBeansOfType\(java.lang.Class\)](http://www.springframework.org/docs/api/org.springframework.beans.factory.ListableBeanFactory.html#getBeansOfType(java.lang.Class))] 来获取bean。很多开发人员都是通过name来获取bean，然后将它们对应到一个实现的接口。

所幸的是，本章中大多数框架都有更简单的查找bean的方法。它们不仅能简化从BeanFactory中查找bean，并且允许你在控制器上应用依赖注射。各框架一节有更多关于集成策略的细节。

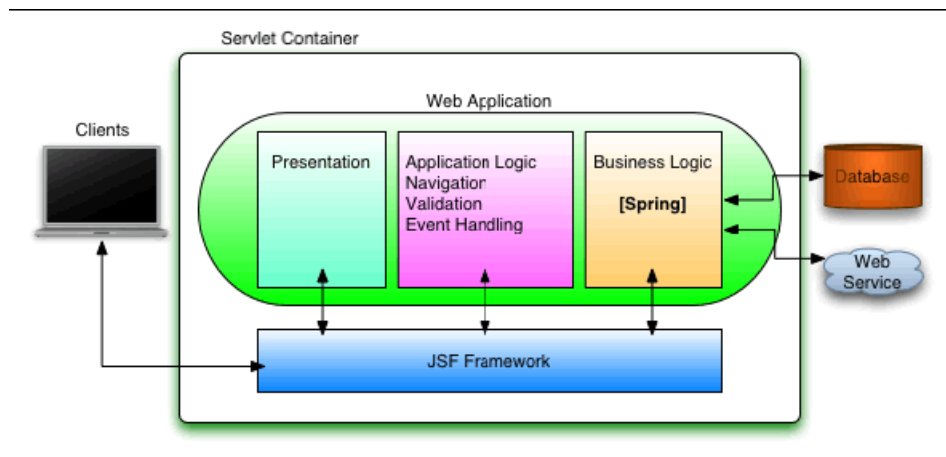
JavaServer Faces

JSF是一种基于组件，事件驱动的web框架。根据Sun Microsystems的JSF Overview [<http://java.sun.com/j2ee/javaxserverfaces/overview.html>]所述，JSF技术包括：

- 一套API，用于表示UI组件并管理它们的状态，处理事件和输入验证，定义页面导航，支持国际化和易用性。
- 一种自定义JSP taglib，用于在JSP页面内表达JSF接口。

图11.1¹演示了JSF如何适应一种web应用程序的框架。

图 11.1. JSF与Spring集成



JSF的突出特性之一就是能够将客户端生成的事件绑定到服务器端的事件处理器上。例如，当一个用户点击一个链接或是一个按钮，就可以调用一个类中方法。这些方法可以是listeners和actions。Listeners一般是改变页面返回Java类或托管bean(managed bean)的状态。它们也可以修改JSF的生命周期，但是它们通常不控制导航。Actions是无参数的方法，返回一个String，表示转向哪里。从一个Action中返回null，表示“保持在同一页面”。

JSF的生命周期由六阶段组成。

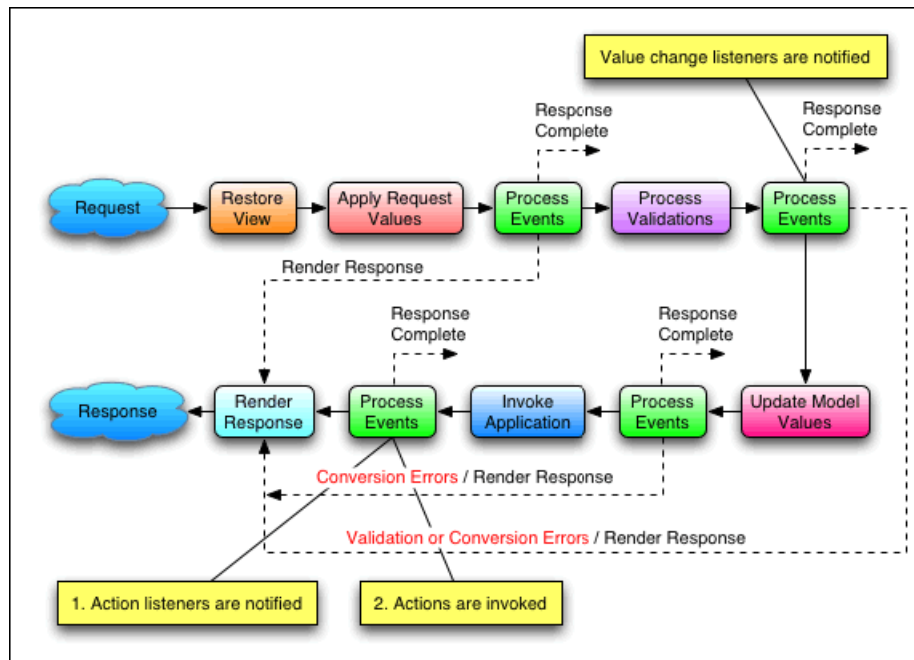
- **Restore View:** 当你重新返回一个JSF页面时，重新创建服务器端组件。
- **Apply Request Values:** 将request参数复制到submitted values组件中。
- **Process Validations:** 转换submitted values并进行验证。

¹此图来源Core JavaServer Faces [<http://www.corejsf.com/>](Geary, David, and Cay Horstmann. Core JavaServer Faces. Sun Microsystems Press, 2004.) 第23页。

- Update Model Values: 将转换过的已经验证过的数值复制到model对象中。
- Invoke Application: 激活listeners和actions, 适用于command组件(你可以使用actions来调用Spring bean来管理业务逻辑和持久层)。
- Render Response: 保存状态, 加载下一个view。

图11.2²演示了一个JSF应用程序中从请求到响应的各个阶段。

图 11.2. JSF生命周期



JSF应用程序中的导航由WEB-INF/faces-config.xml中定义一些导航规则控制的。这个文件还包含一些额外设置:

- ResourceBundle的名称和支持的locale
- 自定义VariableResolvers
- 托管bean(managed beans)及其属性

很多与JSF框架相关的配置也包含在这个文件中。下面是你本节中要完成的faces-config.xml文件的一个裁剪版本。

```

<?xml version="1.0"?>
<!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>
  <!-- Spring VariableResolver for JSF -->
  <application>

```

²此图来源Core JavaServer Faces [http://www.corejsf.com/](Geary, David, and Cay Horstmann. Core JavaServer Faces. Sun Microsystems Press, 2004.) 第274页。

```

<variableresolver>
  org.springframework.web.jsf.DelegatingVariableResolver
</variableresolver>
<locale-config>
  <default-locale>en</default-locale>
  <supported-locale>en</supported-locale>
  <supported-locale>es</supported-locale>
</locale-config>
<message-bundle>messages</message-bundle>
</application>
<navigation-rule>
  <from-view-id>/userList.jsp</from-view-id>
  <navigation-case>
    <from-outcome>add</from-outcome>
    <to-view-id>/userForm.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
<managed-bean>
  <managed-bean-name>userList</managed-bean-name>
  <managed-bean-class>org.appfuse.web.UserList</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>userManager</property-name>
    <value>#{userManager}</value>
  </managed-property>
</managed-bean>
</faces-config>

```

JSF属于一种web框架，没有提供管理业务逻辑和持久层逻辑的API。但是Spring有一个DelegatingVariableResolver类，为JSF应用程序提供了透明的依赖性注射。在文件faces-config.xml中，web/WEB-INF/applicationContext.xml中定义的userManager 设置在userList类上，同时，你通过添加一个setUserManager()方法将它设置在一个普通的Spring托管bean上。

```

<managed-property>
  <property-name>userManager</property-name>
  <value>#{userManager}</value>
</managed-property>

```

和其它框架不同的是，JSF是一种规范，它是JSF实现的一套规则和需求。JSF实现是指支持JSF规范的代码。到写本书时，有两自由的JSF实现可用。

1. Sun的参考实现，网址是<https://javaserverfaces.dev.java.net>。
2. Apache的MyFaces实现，网址是<http://incubator.apache.org/myfaces>。

集成Spring与JSF

在JSF应用程序中集成Spring bean最主要的方法是使用Spring的DelegatingVariableResolver [<http://www.springframework.org/docs/api/org.springframework.web.jsf/DelegatingVariableResolver.html>]。

要在Equinox-JSF程序中配置这一variable resolver，打开文件web/WEB-INF/faces-context.xml，在打开的<faces-config>元素后面，添加下面的<application>元素。

```
<faces-config>
  <application>
    <variable-resolver>
      org.springframework.web.jsf.DelegatingVariableResolver
    </variable-resolver>
    <locale-config>
      <default-locale>en</default-locale>
      <supported-locale>en</supported-locale>
      <supported-locale>es</supported-locale>
    </locale-config>
    <message-bundle>messages</message-bundle>
  </application>
```

通过指定Spring的variable resolver，你可以将Spring bean配置成为你的托管bean的一个托管属性。DelegatingVariableResolver首选会将值委派给所用JSF实现的默认resolver，然后再给Spring的根WebApplicationContext [http://www.springframework.org/docs/api/org/springframework/web/context/WebApplicationContext.html]。这样你很容易将依赖关系注射到你的JSF托管bean中。

托管bean是在文件web/WEB-INF/faces-config.xml中进行定义的。下面是一个样例，#{userManager}是从Spring BeanFactory中取得的一个bean。

```
<managed-bean>
  <managed-bean-name>userList</managed-bean-name>
  <managed-bean-class>org.appfuse.web.UserList</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>userManager</property-name>
    <value>#{userManager}</value>
  </managed-property>
</managed-bean>
```

注意

在DelegatingVariableResolver [http://www.springframework.org/docs/api/org/springframework/web/jsf/DelegatingVariableResolver.html]之前，JSF和Spring集成的最好方法是作用SourceForge.net上JSF-Spring [http://jsf-spring.sourceforge.net/]。

DelegatingVariableResolver是集成JSF与Spring的推荐策略。JSF-Spring项目提供了更多的特性，但是到了2004年12月，它无法支持Spring 1.1.1及以后的版本。

在faces-config.xml中，将属性映射到bean时，一个自定义的VariableResolver就可以胜任，但是有时你需要显式的获取一个bean。FacesContextUtils [http://www.springframework.org/docs/api/org/springframework/web/jsf/FacesContextUtils.html]可以简化这一操作。它与WebApplicationContextUtils，只不过它使用的是FacesContext参数，而不是ServletContext。

```

ApplicationContext ctx =
    FacesContextUtils.getWebApplicationContext(
        FacesContext.getCurrentInstance());

```

View的选择

JSP是JSF支持的唯一的视图技术。但是，它留有扩展空间，可以使用其它技术来配置UI。Hans Bergsten在他的论文Improving JSF by Dumping JSP [http://www.onjava.com/pub/a/onjava/2004/06/09/jsf.html]解释了如何实现。更多有关在JSF中JSP的角色问题，请参考Kito Mann的论文Getting around JSF: The Role of JSP [http://www.javaworld.com/javaworld/jw-12-2004/jw-1213-jsf.html]。要学习JSP的一般知识，请阅读Pro JSP, Third Edition [http://www.apress.com/book/bookDisplay.html?bID=256](Apress)，我也参与这本书的写作。

JSF的一个最大特色是它为扩展设计的API。因为它是基于组件的框架，它鼓励企业和开发人员创建和分享组件。

下面是一个可自由使用的JSF组件的列表。

- MyFaces Components [http://www.marinschek.com/myfaces/tiki/tiki-index.php?page=Features](开源-Apache协议)
- OurFaces Components [https://ourfaces.dev.java.net/](开源-SPL协议)
- Oracle ADF Faces Components [http://www.oracle.com/technology/products/jdev/htdocs/partners/addins/exchange/jsf/index.html](非开源)[译注：现已捐给Apache软件基金会]

JSF和Spring的CRUD样例

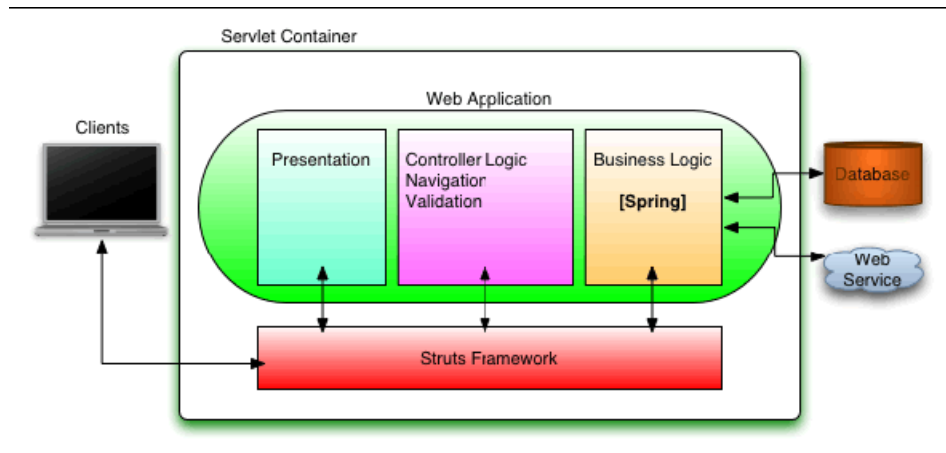
请参考附录A，学习如何用Spring作中间层，Hibernate作后端开发一个JSF程序。

Struts

Struts是Java应用程序中事实上的web框架，主要是因为它是最早发布的框架之一(2001年6月)。Struts由Craig McClanahan创建，是Apache软件基金会上一个开源项目。它大大简化了JSP/Servlet编程方式，并赢得了很多开发人员的偏爱。它简化了编程模式，它是开源的，并且它有庞大的社区，这个项目才能快速成长，在Java web开发人员中流行起来。

图11.3演示了Struts如何组成web程序框架。

图 11.3. Struts与Spring集成



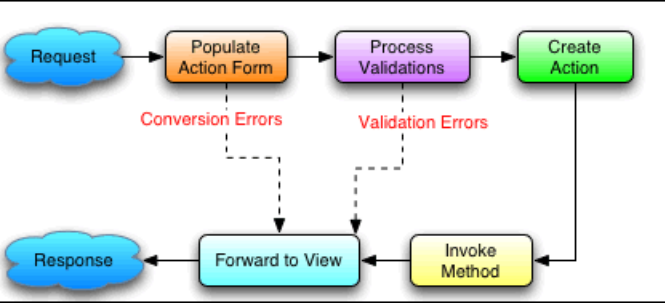
Struts在的Action(也就是控制器,Controller)中不在在生命周期。换句话说,它只有一个单一的入口,或称为一个方法,它是由框架调用的。在这一点上与Spring MVC形成了鲜明的对比,特别是在它的 `SimpleFormController` [http://www.springframework.org/docs/api/org.springframework.web.servlet.mvc.SimpleFormController.html]。Spring允许你复写一些方法,控制基于每个表单的值绑定,验证和表单处理。使用Struts,你必须创建一个 `ActionServlet` [http://struts.apache.org/api/org/apache/struts/action/ActionServlet.html] 子类,或者创建一个自定义的 `RequestProcessor` [http://struts.apache.org/api/org/apache/struts/action/RequestProcessor.html],并复写这一行为。

但是,在上面方法调用前后会出现5个步骤。

- 填充ActionForm: Struts调用ActionForm的 `reset()` 方法,并针对特定的Action填充request参数。从Action到ActionForm的映射(也叫action-mapping)是在一个应用程序的 `/WEB-INF/struts-config.xml` 中进行定义。
- 处理验证: 如果action-mapping设置了 `validate="true"`, Struts Validator首先会处理验证(如果配置的话),然后调用 `ActionForm` [http://struts.apache.org/api/org/apache/struts/action/ActionForm.html] 的 `validate()` 方法。这里的 `ActionForm` 必须继承 `ValidatorForm` [http://struts.apache.org/api/org/apache/struts/validator/ValidatorForm.html] 才能保证有效。
- 创建 Action: `ActionServlet` [http://struts.apache.org/api/org/apache/struts/action/ActionServlet.html] 会创建一个映射到特定的URL的Action。
- 调用方法: Struts调用Action的 `execute()` 方法。Struts有多个你可以继承的Action,所以这个方法名可以有所不同。
- 转发到View: Struts `Action` 必须返回一个 `ActionForward` [http://struts.apache.org/api/org/apache/struts/action/ActionForward.html],它是一个URL的薄包装器。ActionForward是在文件 `struts-config.xml` 中配置的。

图11.4向你演示了从请示到响应,一个Struts应用程序中生命周期过程。

图 11.4. Struts生命周期



Struts是多线程的。也就是说，多个客户端会共享类变量。基于这一原因，Struts建议将你所有的实例变量放在方法里面。使用Spring的Struts 插件允许改变这一行为，为每个客户端创建一个新的Action。

在一个Struts应用程序中创建的Action，都继承了Struts Action类。要继承Action，你必须实现execute()方法，它可以表示为如下。

```
public ActionForward execute(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
throws Exception;
```

除了从Action类继承外，你也可以从其它一些类继承。这些选择是基于某些预期行为的。表11.1所列的Action是最常用的。Spring内置了这些类的子类以简化Struts-Spring集成。

表 11.1. 常用Struts Action

Action名称	预期行为
Action	显示数据或处理某种特定的请求。 http://struts.apache.org/api/org/apache/struts/action/Action.html
DispatchAction	可以使用多个处理方法，方法名由请求参数名标明。一个form中有多个按钮时需要JavaScript支持。 http://struts.apache.org/api/org/apache/struts/actions/DispatchAction.html
LookupDispatchAction	与DispatchAction相似，但是把ResourceBundle的键值绑定到方法名。适用有多个按钮的form。 http://struts.apache.org/api/org/apache/struts/actions/LookupDispatchAction.html
MappingDispatchAction	可以在文件struts-config.xml的action-mapping中指定方法名。 http://struts.apache.org/api/org/apache/struts/actions/MappingDispatchAction.html

集成Struts与Spring

要在你的Struts应用程序中集成Spring，你有两种选择。

- 配置Spring，将你的Action作为bean管理，使用ContextLoaderPlugin
<http://www.springframework.org/docs/api/org/springframework/web/struts/ContextLoaderPlugin.html>，在一个Spring context文件中设置它们的依赖关系。

- 创建一个Spring的ActionSupport类的子类，使用getWebApplicationContext()方法显式的获取Spring托管bean。

ContextLoaderPlugin

ContextLoaderPlugin是一种Struts 1.1+的插件，它可以为Struts ActionServlet加载Spring context文件。这里context引用根WebApplicationContext(由ContextLoaderListener加载)作为它的父bean。context文件的默认名称是映射的servlet名加上-servlet.xml。如果ActionServlet在web.xml中定义为<servlet-name>action</servlet-name>，默认为/WEB-INF/action-servlet.xml。

要配置这一插件，在接近文件struts-config.xml的底部plug-in片断添加以下XML代码。

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
  <set-property property="contextConfigLocation"
    value="/WEB-INF/applicationContext-hibernate.xml,
      /WEB-INF/applicationContext.xml,
      /WEB-INF/action-servlet.xml"/>
</plug-in>
```

本例中不能使用ContextLoaderListener来加载applicationContext-hibernate.xml和applicationContext.xml等文件，因为StrutsTestCase无法初始化web.xml中<listener>部分。通过在插件中指定各个context文件，你的StrutsTestCase测试类就可以访问Spring托管bean。

警告

StrutsTestCase的ServletContextSimulator
[<http://strutstestcase.sourceforge.net/api/servletunit/ServletContextSimulator.html>] 不支持
getResourcesPaths()方法
[http://www.sourceforge.net/tracker/index.php?func=detail&aid=1088864&group_id=39190&atid=424562]，
而Spring用它来加载用通配符来表示多个文件(例如，/WEB-INF/applicationContext*.xml)。如果你打算用StrutsTestCase测试Action，你必须明确的设置每个文件。

在struts-config.xml配置好插件后，你可以配置Action，由Spring管理。Spring 1.1.3中提供两种实现方法。

- 使用Spring的DelegatingRequestProcessor
[<http://www.springframework.org/docs/api/org.springframework.web.struts/DelegatingRequestProcessor.html>]
复写Struts默认的RequestProcessor。
- 在<action-mapping>设置type属性，使用DelegatingActionProxy
[<http://www.springframework.org/docs/api/org.springframework.web.struts/DelegatingActionProxy.html>]。

不管哪种方法都允许文件action-context.xml中管理Action和它们的依赖关系。struts-config.xml中的Action和action-servlet.xml之间的桥梁是由action-mapping的“path”和相应bean的“name”搭建的。如果你的struts-config.xml包含下面的内容。

```
<action path="/users" .../>
```

你就必须在文件action-servlet.xml中定义一个Action的bean，名为“/users”。

```
<bean name="/users" .../>
```

DelegatingRequestProcessor

要在struts-config.xml中配置DelegatingRequestProcessor，复写<controller>元素的“processorClass”属性。紧跟着<action-mapping>添加以下几行。

```
<controller>
  <set-property property="processorClass"
    value="org.springframework.web.struts.
      DelegatingRequestProcessor"/>
</controller>
```

添加这一设置后，就可以在Spring的context文件中查找到它，不管它是什么类型。事实上，你根本不需要指定一个类型。以下两种代码片断同样奏效。

```
<action path="/user" type="org.appfuse.web.UserAction"/>
<action path="/user"/>
```

如果你使用Struts的“module”特性，那么你的bean名称必须包含module前缀。例如，有一个定义<action path="/user"/>的Action，其module前缀为“admin”，则bean名称必须为<bean name="/admin/user"/>。

警告

如果你在Struts应用程序中使用了Tiles，那么在配置<controller>要使用DelegatingTilesRequestProcessor [http://www.springframework.org/docs/api/org.springframework.web.struts/DelegatingTilesRequestProcessor.html]。

DelegatingActionProxy

如果你使用了一个自定义RequestProcessor，而不能使用DelegatingTilesRequestProcessor，你可以使用DelegatingActionProxy作为action-mapping的type属性。第2章是这样使用的。

```
<action path="/user"
  type="org.springframework.web.struts.DelegatingActionProxy"
  name="userForm" scope="request" parameter="method"
  validate="false">
  <forward name="list" path="/userList.jsp"/>
  <forward name="edit" path="/userForm.jsp"/>
</action>
```

action-servlet.xml中的bean定义保持不变，不管你使用了一个自定义的RequestProcessor和DelegatingActionProxy。

在一个bean定义了Action，你就使用Spring的IoC特性，你还可以为每个客户端初始化一个新的Action。要使用这一特性，在你的bean定义中添加`singleton="false"`。

```
<bean name="/user" singleton="false" autowire="byName"
      class="org.appfuse.web.UserAction"/>
```

这样你可以将成员变量放在类中定义，第每个用户也得到的是它的实例。创建一个实例的性能开销最小，很多web框架(如JSF和WebWork)在它们的控制器使用这种策略。

如果你不想为action而维护两个XML文件，你可以用XDoclet生成，或者使用Spring的ActionSupport类。

ActionSupport类

正如前面所提到的那样，你可以使用WebApplicationContextUtils类从ServletContext中获得一个WebApplicationContext。一个更简单的方法是继承Spring的Action类。例如，不要继承Struts的Action类，你可以创建一个Spring的ActionSupport类。[\[http://www.springframework.org/docs/api/org.springframework.web.struts.ActionSupport.html\]](http://www.springframework.org/docs/api/org.springframework.web.struts.ActionSupport.html)的子类。

Spring的ActionSupport提供了许多额外的方便的方法，如getWebApplicationContext()。下面的例子演示在一个Action中如何使用它。

```
public class UserAction extends DispatchActionSupport {
    public ActionForward execute(ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
        if (log.isDebugEnabled()) {
            log.debug("entering 'delete' method...");
        }
        WebApplicationContext ctx = getWebApplicationContext();
        UserManager mgr = (UserManager) ctx.getBean("userManager");
        // talk to manager for business logic
        return mapping.findForward("success");
    }
}
```

Spring中包含了所有标准Struts Action的子类。而Spring版本的仅仅是在名称后追加了Support:

ActionSupport [\[http://www.springframework.org/docs/api/org.springframework.web.struts.ActionSupport.html\]](http://www.springframework.org/docs/api/org.springframework.web.struts.ActionSupport.html),
 DispatchActionSupport [\[http://www.springframework.org/docs/api/org.springframework.web.struts/DispatchActionSupport.html\]](http://www.springframework.org/docs/api/org.springframework.web.struts/DispatchActionSupport.html),
 LookupDispatchActionSupport [\[http://www.springframework.org/docs/api/org.springframework.web.struts/LookupDispatchActionSupport.html\]](http://www.springframework.org/docs/api/org.springframework.web.struts/LookupDispatchActionSupport.html)
 和 MappingDispatchActionSupport [\[http://www.springframework.org/docs/api/org.springframework.web.struts/MappingActionSupport.html\]](http://www.springframework.org/docs/api/org.springframework.web.struts/MappingActionSupport.html)。

使用时选择最适合项目那种。创建子类使的你代码更具有可读性，你也可以弄懂依赖是如何解决的。但是，插件方式很容易在context文件中添加新的依赖关系。不管哪种方法，Spring都为集成两种框架提供了不错的选择。

视图的选择

JSP是Struts应用程序默认的view技术。但是，也有其它可用的方案。Velocity [<http://jakarta.apache.org/velocity>] 和它的子项目VelocityStruts [<http://jakarta.apache.org/velocity/tools/struts/>]允许你使用Velocity模板作为JSP的一种备选方案，或可在同一应用中结合使用JSP。你也可以XML和XSLT，现有两种开源方案提供了便利，StrutsCX [<http://it.cappuccinonet.com/strutsCX/index.php>]和Struts for Transforming XML with XSL (stxx) [<http://stxx.sourceforge.net/>]。

Struts与Spring的CRUD实例

请参考第2章，学习如何用Spring层，Hibernate作后端开发一个Struts应用程序。第2章中的教程为结合Spring开发一个Struts应用程序提供了一个很好的思路。Struts 2.0(代号为shale)将会合并Struts和JSF。更多有关Shale的信息请阅读http://www.theserverside.com/news/thread.tss?thread_id=29861。

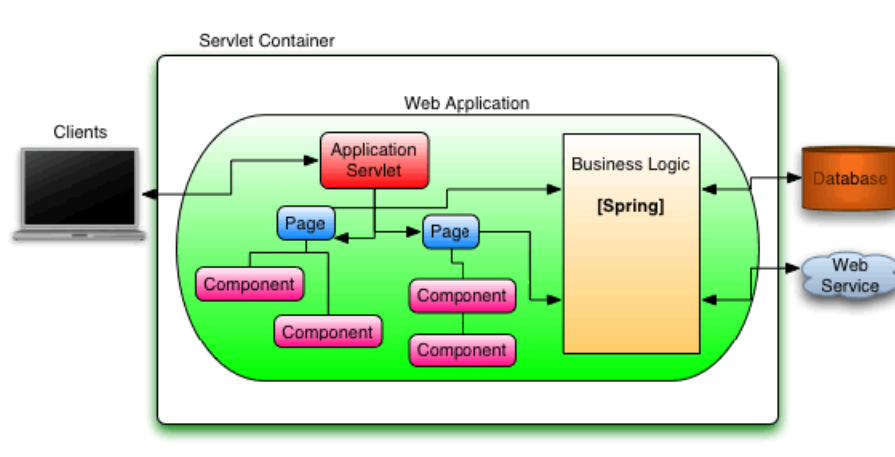
Tapestry

Tapestry是一个基于组件的框架，用于开发web程序。与其它框架有所不同，Tapestry使用一种与传统GUI框架类似组件对象模型。根据Tapestry的奠基人，Howard Lewis Ship所述。

Tapestry [<http://jakarta.apache.org/tapestry>]组件是一个整体框架中的一个对象。组件的责任由框架的设计和结构来定义。组件之所以是组件，当它遵循框架规则时，就不是简单的对象。这里规则是指要继承形式，应该遵循的命名规范(针对类和方法)，或是要实现的接口。组件可以用于框架context中。相对组件，框架扮演着一个容器的角色，控制着组件何时实例化和初始化，指示何时调用组件的方法。^{3 3}

图11.5演示了Tapestry的web应用程序框架

图 11.5. Tapestry与Spring集成



³Lewis Ship, Howard. Tapestry in Action [<http://www.manning.com/lewisship/index.html>]. Greenwich, CT: Manning Publications Co., 2004

Tapestry的组件模型允许你在项目中和项目间高度复用。你可以将它打包成jar文件，分发给其它团队和开发人员。

Tapestry试图向开发人员隐藏Servlet API。学习Tapestry可以描绘成一个“非学习”的过程。GUI程序员在调整到Tapestry环境时会较轻松。Tapestry根据对象，方法和属性进行操作，而不是URL和查询参数。所有的URL构建，页面转发和方法调用都是透明的。

使用Tapestry其它的一些好处包括，精确的行错误报告[<http://jakarta.apache.org/tapestry/images/LinePrecise.png>]和简单易用的HTML模板。其它框架使用外部模板系统，而Tapestry拥有自己的模板系统。Tapestry模板一般是HTML文件，但也可以是WML或XML。你可以通过在现有的HTML元素上使用Tapestry特定的属性来组装这些模板。

一个模板中大约90%的是普通HTML标记。HTML中包含一些Tapestry组件特定的标记。这些标通过一个jwcid属性组织起来。JWC是Java Web Component的缩写。下面是一个使用Insert[<http://jakarta.apache.org/tapestry/doc/ComponentReference/Insert.html>]组件的例子⁴

```
<span jwcid="@Insert" value="ognl:user.name">Joe User</span>
```

这种特殊的语言允许你使用WYSIWYG(所见即所得)HTML编辑器来编辑HTML模板，并可以使用浏览器进行查看。图像设计师和HTML开发人员也容易编辑web应用中的动态页面。

当你在Tapestry中提交一个form时，框架会执行6个基本步骤。

1. 初始化Page类：创建一个Page类，如果已经存在，就从缓存池取出它。Tapestry会设置所有持久性page属性。
2. 调用pageBeginRender()方法：在页面生成响应之前调用pageBeginRender()方法。这样允许设置或初始化某个页面属性。
3. 填充Page属性：填充HTML模板中基于表达式的Page类的属性。
4. 调用Listener方法：调用Page的listener方法。它首选调用Submit[<http://jakarta.apache.org/tapestry/doc/ComponentReference/Submit.html>]组件的listener方法，然后调用Form[<http://jakarta.apache.org/tapestry/doc/ComponentReference/Form.html>]组件的listener方法。Submit组件通常是按钮，但你也可以使用ImageSubmit[<http://jakarta.apache.org/tapestry/doc/ComponentReference/ImageSubmit.html>]和LinkSubmit[<http://jakarta.apache.org/tapestry/doc/ComponentReference/LinkSubmit.html>]获得同样的功能。
5. 激活下一个Page：开发人员在listener方法中激活下一个页面。
6. 调用pageBeginRender()方法：在生成一个成功页面后，调用pageBeginRender()方法。这样允许对象释放任何在生成页面期间所需的资源。

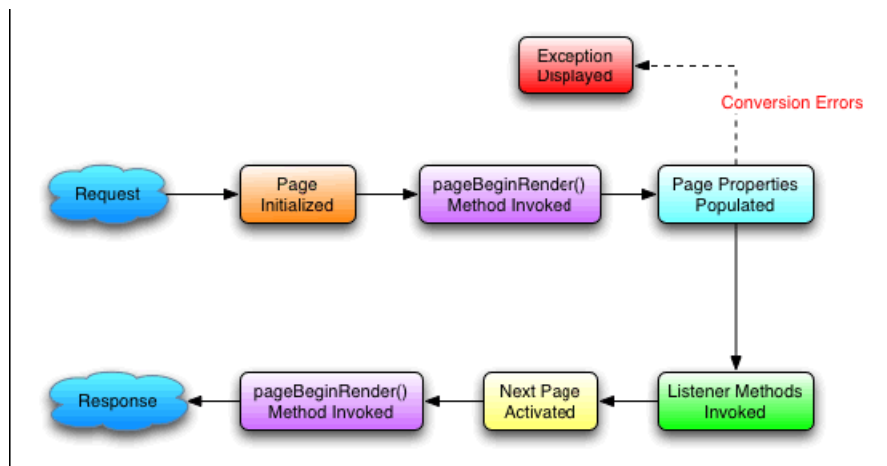
注意

还有一个pageEndRender()，但很少使用。这里为你提供空白的实现[[http://jakarta.apache.org/tapestry/doc/api/org/apache/tapestry/AbstractComponent.html#pageEndRender\(org.apache.tapestry.event.PageEvent\)](http://jakarta.apache.org/tapestry/doc/api/org/apache/tapestry/AbstractComponent.html#pageEndRender(org.apache.tapestry.event.PageEvent))]。

图11.6演示了Tapestry中提供的一个form的各个阶段。

⁴ <http://jakarta.apache.org/tapestry/doc/ComponentReference>上Tapestry组件的一份完整清单。

图 11.6. Tapestry生命周期



不像其它把无状态的servlet作为控制器的web框架，Tapestry的Page类是有状态的JavaBean。对于Tapestry应用中的每个页面视图，都有一个相应的页面规范的XML文件，一个HTML模板和一个继承BasePage [http://jakarta.apache.org/tapestry/doc/api/org/apache/tapestry/html/BasePage.html](可选项)。构建某个页面的一个新实例，需要少许工作。

1. 加载和分析page规范。
2. 动态加载和分析page的子类。
3. 实例化page。
4. 定位，分析，继承和实例化page的组件。
5. 读取和应用page模板到page。
6. 模板内的组件要能够找到它们的模板，并能分析应用它们。

以上过程需要大量的处理动力。所以，page实例是集中在中心页面池中，这与一个数据连接池非常类似。每个请求使用的各个页面的实例，它是为请求过程缓存的。这样在Tapestry应用程序中可以高效利用资源，快速的加载页面。

Tapestry与Spring集成

Spring并没有为Tapestry提供像JSF和Struts那样的支持。这主要是因为根本没必要。你花很小气力就可以将Spring集成到Tapestry应用程序中。

在一个Tapestry的page类中，可以使用WebApplicationContextUtils来获得WebApplicationContext。

```

ServletContext servletContext =
    getRequestCycle().getRequestContext().getServlet()
        .getServletContext();
WebApplicationContext appContext =
    WebApplicationContextUtils.getApplicationContext(servletContext);
userManager userManager =
  
```

```
(userManager) applicationContext.getBean("userManager");
```

一种更简洁的方法是利用Tapestry和其依赖注入特性。下面的步骤向你演示了如何创建和配置自己的 **B a s e E n g i n e** [http://jakarta.apache.org/tapestry/doc/api/org/apache/tapestry/engine/BaseEngine.html]。然后你就会看到如何在一个页面特定的文件中关联依赖关系。

1. 通过继承Tapestry的BaseEngine类，将ApplicationContext暴露给Tapestry。下面的代码将ApplicationContext放到Tapestry的Global变量中，这与HttpSession相似。事实上，它只是一个存放在session中的HashMap。

```
package org.appfuse.web;
// ...

public class BaseEngine extends org.apache.tapestry.engine.BaseEngine {
    public static final String APPLICATION_CONTEXT_KEY = "appContext";

    protected void setupForRequest(RequestContext context) {
        super.setupForRequest(context);
        Map global = (Map) getGlobal();
        ApplicationContext ac = (ApplicationContext) global
            .get(APPLICATION_CONTEXT_KEY);
        ac = WebApplicationContextUtils
            .getWebApplicationContext(context
                .getServlet().getServletContext());
        global.put(APPLICATION_CONTEXT_KEY, ac);
    }
}
```

2. 在application文件中，将“engine-class”属性修改为你的BaseEngine。

```
<application name="tapestry"
    engine-class="org.appfuse.web.BaseEngine">
```

3. 在你的page类中，为你想访问的Spring bean添加一个抽象的getter方法。

```
package org.appfuse.web;
import org.appfuse.service.UserManager;

public abstract class UserList extends BasePage {
    public abstract UserManager getUserManager();
}
```

4. 在page特定的文件中使用以下语法来装配你的page类中的依赖关系。


```
<property-specification name="userManager"
  type="org.appfuse.service.UserManager">
  global.appContext.getBean("userManager")
</property-specification>
```

Tapestry与Spring的CRUD实例一节会向你演示如何使用Tapestry和Spring将Spring托管bean注射到你的page类中。

注意

Hivemind [<http://jakarta.apache.org/hivemind>]是一个与Spring类似的IoC容器。Tapestry的下一版本(3.1)会使用Hivemind来关联你内部服务。有关Hivemind，Spring和PicoContainer容器一个对比，请参阅Mike Spille的论文 *Inversion of Control Containers* [<http://www.pyrasun.com/mike/mt/archives/2004/11/06/15.46.14/>]。

View的选择

Tapestry没有备选的模板引擎。然而，它在模板中支持XML和WML。

Tapestry与Spring的CRUD实例

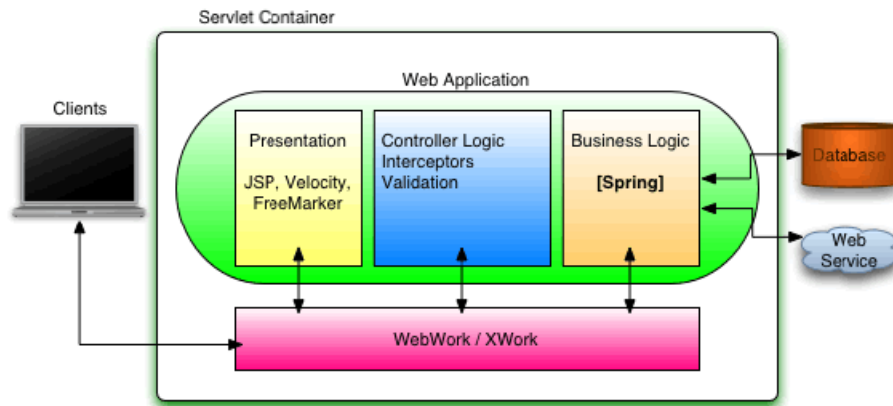
请参考附录A，学习如何用Spring作中间层，Hibernate作后端开发一个Tapestry程序。

WebWork

WebWork [<http://www.opensymphony.com/webwork>]是一种以简约主义为设计目标的web框架。它构建在XWork [<http://www.opensymphony.com/xwork>]这种通用command框架之上。XWork也有自己的IoC容器，但它没有提供Spring的所有特性，本节不作介绍。WebWork的控制器被称为Action，主要是因为它们必须实现 `Action` [<http://www.opensymphony.com/xwork/api/com/opensymphony/xwork/Action.html>]接口。`ActionSupport` [<http://www.opensymphony.com/xwork/api/com/opensymphony/xwork/ActionSupport.html>]类实现了这一接口，它是WebWork action最常用的父类。

图11.7演示了WebWork如何集成到web应用框架中。

图 11.7. WebWork与Spring集成



WebWork的action主要包含访问模型属性的方法和返回字符串的方法。这些字符串还必须与xwork.xml配置文件中“result”名称相匹配。

你在WebWork程序中开发的action与你在JSF应用程序中开发看起来类似。Action主要包含一个唯一的execute()方法。但是你很容易基于一个URL添加多个方法，实行控制。下面是一个简单的Action样例，以及xwork.xml中相应的配置。

```

public class UserAction extends ActionSupport {
    private UserManager mgr = null;
    private List users;

    public void setUserManager(UserManager userManager) {
        this.mgr = userManager;
    }

    public List getUsers() {
        return users;
    }

    public String execute() {
        users = mgr.getUsers();
        return SUCCESS;
    }
}

<action name="users" class="org.appfuse.web.UserAction">
    <result name="success" type="dispatcher">userList.jsp</result>
</action>

```

和Spring MVC非常相似，WebWork使用拦截器(interceptors)截取请求和响应过程。这一点与Servlet Filter类似，除你可以直接与action交互这一点外。WebWork在自己的框架中使用拦截器。其中一部分(拦截器)初始化Action，准备为它填充数据，设置参数并处理任何转换错误。下面针对每个请求的拦截器的默认堆栈。

```

<interceptor-stack name="defaultStack">
  <interceptor-ref name="servlet-config"/>
  <interceptor-ref name="prepare"/>
  <interceptor-ref name="static-params"/>
  <interceptor-ref name="params"/>
  <interceptor-ref name="conversionError"/>
</interceptor-stack>

```

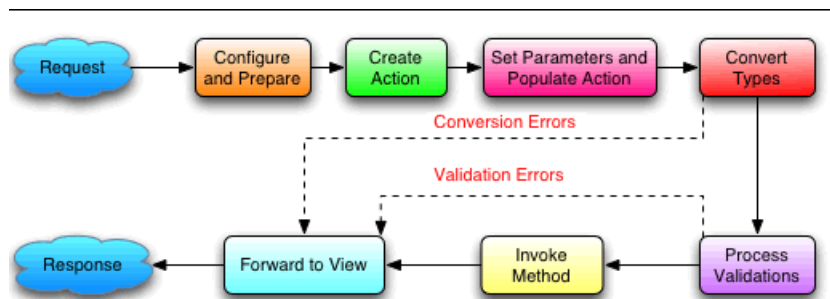
这些拦截器作为每个请求的一部分，并协助定义WebWork的生命周期。你也可以复写上面基于每个action的list。当你提交一个form时，它通常是先配置验证，发送用户并显示错误(workflow)。另外两种拦截器负责处理这些操作：

`ValidationInterceptor` [http://www.opensymphony.com/xwork/api/com/opensymphony/xwork/validator/ValidationInterceptor.html]
和
`DefaultWorkflowInterceptor` [http://www.opensymphony.com/xwork/api/com/opensymphony/xwork/interceptor/DefaultWorkflowInterceptor.html]。
WebWork的生命周期分为七步。

1. 配置和预备：为填充数据准备null值，为Action类读取配置。
2. 创建Action：创建Action类(每个请求创建一个)。
3. 设置参数并填充Action：将ValueStack中的请求参数转换成有意义的值，并设置好Action类属性。
4. 转换类型：将Strings转换成Action中指定的对象类型。如果出现错误，跳转到输入页面。
5. 处理验证：如果一个Action定义了验证规则。它会执行它们。如果出现错误，跳转到输入页面。
6. 调用方法：执行`execute()`或指定的方法。
7. 转发到View：转发到一个JSP，Velocity或FreeMarker View。

图11.8演示了一个WebWork应用程序中从请求到响应的各个阶段。

图 11.8. WebWork的生命周期



WebWork与Spring集成

WebWork维护着自己的Spring集成项目，在java.net上XWork-optional [https://xwork-optional.dev.java.net/] 项目中。本节中代码使用了1.1.1版本 [https://xwork-optional.dev.java.net/servlets/ProjectDocumentList?folderID=1425&expandFolder=1425&folderID=1425]。目前有三种选择可用于WebWork与Spring集成。

- 复写ObjectFactory: 复写XWork默认的ObjectFactory [http://www.opensymphony.com/xwork/api/com/opensymphony/xwork/ObjectFactory.html], 这样XWork可以在根WebApplicationContext中查找Spring bean。
- 自动装配拦截器: 使用一个拦截器自动装载Action的依赖关系。
- 根据<action>元素下的<external-ref>元素中定义的名称查找Spring bean。

SpringObjectFactory

复写XWork所提供的默认的ObjectFactory, 允许框架在对象的多个位置中查找。实现SpringObjectFactory的步骤归纳如下。

1. 在你的web.xml中添加SpringObjectFactoryListener。<listener>片断在ContextLoaderListener后面。

```
<listener>
<listener-class>
    com.opensymphony.xwork.spring.SpringObjectFactoryListener
</listener-class>
</listener>
```

你也可以配置好SpringObjectFactory在一个context文件中初始化为一个bean。这样免除了在web.xml中指定SpringObjectFactoryListener的必要性。添加的代码到一个由ContextLoaderListener加载的context文件中。

```
<bean id="springObjectFactory"
    class="com.opensymphony.xwork.spring.SpringObjectFactory"
    init-method="initObjectFactory" />
```

2. 在Spring的context文件(例如, WEB-INF/actionServlet.xml)中配置WebWork Action。文件的名称无关紧要, 必须由ContextLoaderListener加载。与常规情况下一样在Spring中绑定依赖关系。

```
<bean id="userAction" class="org.appfuse.web.UserAction"
    singleton="false">
    <property name="userManager">
        <ref bean="userManager" />
    </property>
</bean>
```

3. 修改文件xwork.xml中action的class属性值, 使其bean的id一致(使用的bean的“name”属性同样能够正常运行)。

```
<action name="users" class="userAction" method="list">
    <result name="success" type="dispatcher">userList.jsp</result>
```

```
</action>
```

这一技术也允许你在Spring中配置拦截器。

ActionAutowiringInterceptor

WebWork的拦截器是一个功能强大的概念。它们与Servlet Filter十分相似，但能够让你访问你要调用的Action。你可以使用ActionAutowiringInterceptor来设置Action的依赖关系。可按如下步骤进行配置。

1. 在文件xwork.xml中的拦截器列表中添加ActionAutowiringInterceptor。
2. 创建一个新的<interceptor-stack>，其包含了这一拦截器。
3. 复写默认的interceptor stack，引用你新创建的stack。

```
<package name="default" extends="webwork-default">
  <interceptors>
    <interceptor name="autowireDependencies"
      class="com.opensymphony.xwork.spring.interceptor.
        ActionAutowiringInterceptor"/>
    <interceptor-stack name="defaultActionStack">
      <interceptor-ref name="defaultStack"/>
      <interceptor-ref name="autowireDependencies"/>
    </interceptor-stack>
  </interceptors>
  <default-interceptor-ref name="defaultActionStack"/>
</package>
```

这样就可以自动设置与Spring bean对应(根据JavaBean命名规范)的action的setXXXX方法。

然而，这一方法还是需要少量配置，它也要求你复写任何你要用的默认的interceptor stack。这就意味着，你不能使用“validationWorkflowStack”来自动验证Action。你必须创建一个新的interceptor-stack，并在action的<interceptor-ref>中引用它。

SpringExternalReferenceResolver

XWork允许你在文件xwork.xml中使用<external-ref>来指定Action的依赖关系。在大多数情况下，引用是指向xwork.xml文件中的其它组件。要改变这一状况，以便能够在Spring的ApplicationContext中查找引用，请完成以下步骤。

1. 请确保spring-xwork-integration.jar放在目录WEB-INF/lib中。然后在web.xml文件中SpringContextLoaderListener的后面添加SpringExternalReferenceResolverSetupListener。

```
<listener>
  <listener-class>
    com.opensymphony.xwork.spring.SpringExternalReferenceResolverSetupListener
  </listener-class>
</listener>
```

```
</listener>
```

这个类将任何实现 `ApplicationContextAware` [http://www.springframework.org/docs/api/org/springframework/context/ApplicationContextAware.html] 接口的 `ExternalReferenceResolvers` [http://www.opensymphony.com/xwork/api/com/opensymphony/xwork/config/ExternalReferenceResolver.html] 上设置了 `ApplicationContext`。

2. 修改文件 `xwork.xml`，使用 `SpringExternalReferenceResolver`。

```
package name="default" extends="webwork-default"
externalReferenceResolver=
"com.opensymphony.xwork.spring.SpringExternalReferenceResolver">
```

3. 在你的拦截器列表中添加 `ExternalReferenceResolvers` [http://www.opensymphony.com/xwork/api/com/opensymphony/xwork/config/ExternalReferenceResolver.html]，复写默认的 `stack`，使用这一拦截器。

```
<interceptors>
  <interceptor name="referenceResolver"
    class="com.opensymphony.xwork.interceptor.ExternalReferencesInterceptor"/>
  <interceptor-stack name="defaultReferenceStack">
    <interceptor-ref name="defaultStack"/>
    <interceptor-ref name="referenceResolver"/>
  </interceptor-stack>
</interceptors>
<default-interceptor-ref name="defaultReferenceStack"/>
```

4. 使用一个 `<external-ref>` 元素来指定 Action 依赖关系。

```
<action name="users" class="org.appfuse.web.UserAction" method="list">
  <external-ref name="userManager">userManager</external-ref>
  <result name="success" type="dispatcher">userList.jsp</result>
</action>
```

在这个例子中，`UserAction` 应该有一个 `setUserManager()` 方法。

View的选择

WebWork 支持 JSP, Velocity 和 FreeMarker 等模板技术。更多有关配置和使用这些技术的信息，请参考 WebWork's Tutorials [http://www.opensymphony.com/webwork/wikidocs/TutorialLesson04.html]。

WebWork 的 JSP 标签的一个特性是，它们能够安全的使用 Velocity 模板来构建 HTML。这样就很容易修改由标签生成的 HTML。AppFuse 的 WebWork 版中有这样例子，位于 `web/template` [https://appfuse.dev.java.net/source/browse/appfuse/extras/webwork/web/template/] 目录中。

WebWork与Spring的CRUD实例

请参考附录A，学习如何用Spring作中间层，Hibernate作后端开发一个WebWork程序。

Web框架对比

现在你已经明白了如何集成Spring和JSP，Struts，Tapestry与WebWork，你必须选择其一。选择web框架常常是个人选择，基于你的产品需要或经验。下表指出每种框架的强项和弱点。这些内容主要基于我对每种框架使用经验。要注意的是我一名Struts忠实用户，所以我可以有点偏激。

表 11.2. 每种框架的正反两面

框架	优势	弱点
Struts	<ul style="list-style-type: none"> 历史悠久，是最早和最为广泛使用的框架之一。 良好的文档支持，有大量的书籍和实例。 全功能的JSP标签库用于处理表单。 使用StrutsTestCase很容易进行集成测试。 	<ul style="list-style-type: none"> ActionForm，所有其它框架都允许直接与模型对象交互。 很难孤立Action进行真正的单元测试。
JSF	<ul style="list-style-type: none"> J2EE的标准web框架，有大量的厂商支持。 一种基于view的框架，没必要捆在HTML。 如果你熟悉Struts和JSP，很容易上手。 容易集成和使用导航特性。 基于组件的框架，能够在对象间进行高度复用。 	<ul style="list-style-type: none"> 新生事物，还不够成熟。 “Tag soup”(标签煲的汤)，很多JSP文件不包含一行HTML代码。 在组件/Java中写HTML，看起来退了一步。 太注重工具提供商的利益，忽略了开发人员。
Spring MVC	<ul style="list-style-type: none"> 拥有生命周期，很容易复写控制器的行为。 JSP标签可以全面控制HTML。 内置了向导型的form支持。 支持Interceptors，可以使用IoC。 支持大多数的view技术。 很容易与Spring中间层集成。 	<ul style="list-style-type: none"> JSP标签要使用更多的代码用于输入控制。 不如其它框架流行。(正在壮大)
Tapestry	<ul style="list-style-type: none"> HTML模板支持所见即所得(WYSIWYG)，设计人员很容易与开发人员沟通。 API与GUI程序员更亲近。 基于组件的框架，能够在对象间进行高度复用。 	<ul style="list-style-type: none"> 对熟悉Servlet API的用户来说学习曲线大。 URL令人费解。(会在3.1中修正)

	<ul style="list-style-type: none">• 简单但功能强大的框架。• Interceptors功能强大，用于管理公共的aspect。• JSP标签库允许你自定义生成的HTML。• 对Velocity和JSP提供了良好的支持。	<ul style="list-style-type: none">• 缺少文档。^{a a}• 客户端的验证不成熟。
--	---	--

^a过去，WebWork的文档非常稀少。但是，两本WebWork专著计划在2005年中出版：WebWork in Action (Manning)和WebWork Live [<http://www.sourcebeat.com/TitleAction.do?id=6>](SourceBeat)。

特性对比

这里有几个在开发应用程序时在web框架中应该具备的小特性。现列于下表，并附出了每种框架是如何处理的。

表 11.3. web框架特性对比

特性	框架	支持程度
<p>可排序可分布的数据列表</p> <p>一种应该允许你集成组件对数据列表进行排序和分页</p>	Struts	基于JSP，你可以使用Display Tag [http://displaytag.sourceforge.net/]，Value List Tag [http://valuelist.sourceforge.net/]和Data Grid Tag [http://platachog/tags/sandbox/dc/datagriddc/index.html]。
	JSF	基于JSP，有很多标签库可用。内置的dataTable组件无法脱离环境排序。
	Spring	基于JSP，有很多标签库可用。
	Tapestry	contribution: Table [http://platachog/tapestry/dc/ComponentReference/enable.html]组件提供了这一功能。
	WebWork	基于JSP，有很多标签库可用。
<p>书签</p> <p>书签在应用程序中具有管理页面能力，并很容易返回。</p> <p>完全控制是指如下所示的URL可以简单的通过email发送给同事，并且他们能够点击它来编辑用户记录。</p> <p>http://company/401k.html?id=foo</p>	Struts	完全控制URL，你很容易创建URL来编辑和查看记录。
	JSF	一般情况下，全部是POST。两次发送同一post，每次都会得到不同的行为。
	Spring	完全控制URL，你很容易创建URL来编辑和查看记录。
	Tapestry	不能控制URL，它们看起来有点冗长和丑陋
	WebWork	完全控制URL，你很容易创建URL来编辑和查看记录。
<p>容易撤消和多按钮表单处理</p> <p>一个框架应该允许你轻易的取消操作并返回到前一屏幕。这应该是一种探测取消按钮是否被点击及取消验证或其它相关操作。</p>	Struts	内置<html:cancel>按钮，意即可以通过onclick handler取消客户端验证。
	JSF	简单易用的导航规则允许将cancel动作传递到其它页面。
	Spring	你可以复写processFormSubmission()方法来取消提交。
	Tapestry	你可以添加一个“cancel” listener，很容易将一个按钮映射到它。
	WebWork	你必须在那些提交表单所调用的方法中处理逻辑。

<p>容易测试</p> <p>一个框架应该为你提供在servlet容器外测试进行测试的能力，这样可以快速和有效的进行测试。</p>	Struts	StrutsTestCase 和 它的MockStrutsTestCase使这一操作变得很简单。
	JSF	最容易测试，因为托管bean是与Servlet API紧密联系在一起的。模拟所依赖的对象很容易。
	Spring	容易，因为spring-mock.jar针对Servlet API。这个库在测试其它框架时也很有用。
	Tapestry	只有少量有关测试controller(page)类的支持和例子。
	WebWork	最容易测试，因为action是与Servlet API紧密联系在一起的。模拟所依赖的对象很容易。
<p>成功信息</p> <p>显示成功信息是web应用程序中一个非常有用的特性。用户想知道到底发生了什么，一条成功信息是一种很好的确认操作成功的方法。它对于试图消除应用程序中重复提交的情况也很重要。当用户提交了一个表单然后在下一页面刷新浏览器时，就可以发生重复提交的情况。如果你是简单的转到下一页面，并且用户已经添加了一条记录，那么刷新操作会添加额外的一条记录。避免这种情况可以使用token和session变量，但是最简单的方法是在提交后进行重定向。</p>	Struts	拥有用于设置和读取成功信息的最佳API。你可以在任何Action父类中使用addMessage()方法，并可以用<HTML:messages>来读取。你可以将它们放进session中，JSP标签会自动删除它们。
	JSF	有设置成功信息的机制，但是不能穿透重定向(redirect)，在托管bean中取得一个应用的ResourceBundle。JSF中的国际化需要增强，这方面其它框架要做好些。
	Spring	可以将模型对象作为redirect一部分进行传递，所以你可以把你的messages放入其中。
	Tapestry	没有相关措施，但是它很容易在page父类中添加set/getMessage()来实现。有点别扭的是，Tapestry要求你用抛出异常方式来重定向。
	WebWork	拥有与Struts相似的一套工具，但是message不能穿透重定向，强制你实现自己的“从session中存取”逻辑。

验证 有一点很重要，当用户输入一个错误信息时，用户应该能及时得到反馈信息。一种健壮的客户验证实现方案应该允许窄带用户能更容易使用你的应用程序。	Struts	支持Commons Validator，它要求定义有的验证信息。为你提供了全面的错误显示控制。客户端验证是最好的方案之一(Commons Validator的一部分)。
	JSF	对于1.1版，默认的验证消息(validation messages)太“程序员主义”了。可以自定义，但不允许你在messages中使用label。(在1.2中会修正)
	Spring	支持Commons Validator，并可以创建自定义的validator。两种方法都要求定义所有的验证消息。为你提供了全面的错误显示控制。如果使用Commons Validator的话，客户端验证不错。
	Tapestry	由于内置的验证规则，默认情况下就很容易扩展的验证框架。客户端验证很好，但JavaScript函数放在页面中(相对于一个外部文件)。
	WebWork	验证规则中支持OGNL表达式，其功能非常强大。客户端验证刚刚起步。

^a你可以使用 URL Rewrite Filter [<http://www.tuckey.org/urlrewrite/>]清理难看的URL。

上面讨论的各种框架都是成熟的项目，能够更简单的开发web程序。很难下结论说某种就比其它的更好。你的选择可能是你热忠于用某一框架来开发。你可能想问自己，“我该学哪一种呢？”

锦囊妙计

这里有一些能简化web程序开发的建议。这里列出一二。

1. 使用一种类似SiteMesh和Tiles的修饰工具。这样你可以用单个文件控制你的应用的页面布局。结合CSS，会更加容易。当客户想通过布局来作调整(这是时有发生)，这可能只是修改一两个文件的事。
2. 使用基于扩展(extension-based)的映射。基于路径的映射如/faces/*和/do/*虽然不错，但是大量的状态跟踪引擎不会加快速度，所以用扩展会更好一些。
3. 用一个通用的扩展映射，如*.html。没有理由为你的套件提供动力的支撑支柱打广告。这样还方便你切换web框架，保持URL完整。URL Wirter Filter能帮你将以前的URL转译成新的。
4. 使用Servlet Filter。你放入filter中逻辑越多，那你对web框架的依赖就越少。
5. 在本年度学习一种新的框架。你会成功一位更为出色的开发人员，研究如何使用其它框架解决问题。当今web框架中的很多特性都是从其它框架基础上构想出来。学习如何利用你的竞争力保持你在框架上的优势。

本章小结

JSF允许你使用它的托管属性(managed-properties)，这和Spring差不多。这是一种值得期待的技术，特别是这么多大公司参与进来并提供了工具。Struts仍然是最流行的web框架之一。它拥有众多工具，大量的实例和丰富的文档。认为Struts很快会被取代不大可能的，特别是它的Shale子项目[<http://www.apachenews.org/archives/000552.html>]会将JSF作为它的主要view技术。Tapestry是一种优雅的基于组件的框架，包含很多JSF的预期目标。它的HTML模板使得设计人员使用起来很简单，

并且它有一个很活跃的社区来协助你解决问题。WebWork是一种简洁的框架，提供很多强大的功能。它的拦截器允许使用AOP的方式管理和控制action，保持你的action足够简单和轻巧。结合Spring并使用xwork-option项目提供的集成选项，你就很容易将Spring集成到你的WebWork项目中。

本章向你演示Spring如何与流行的web框架集成。还提供每种框架的优势和弱点，并比较它们对一些公共的web特性支持情况。Spring与每种框架集成的样例在附录A中或前面的章节中提供了。

要强调的是，本章中概述和样例主要是向你演示了在Spring中如何进行各种选择。它不要求使用指定的web框架。Spring通常保持中立，这是另一个留给开发人员更多选择空间的好例子。

推荐阅读：

- Core JavaServer Faces [<http://www.horstmann.com/corejsf/>]，David Geary 和 Cay Horstmann著。
- JSF in Action [<http://www.manning.com/mann>]，Kito Mann著。
- Struts Live [<http://www.sourcebeat.com/TitleAction.do?id=3>]，Jonathan Lehr 和 Rick Hightower著。
- Struts in Action [<http://www.manning.com/husted>]，Ted Husted等著。
- Tapestry in Action [<http://www.manning.com/lewisship/index.html>]，Howard Lewis Ship著。
- WebWork Live，Matthew Porter著。
- WebWork in Action，Patrick Lightbody等著。