



Java EE框架

2. OGA

Copyr ight 2004-2006

目录

JBoss Seam简介	
1. Seam 入门	
1.1. 试试看	1
1.1.1. 在JBoss AS上运行示例	1
1.1.2. 在Tomcat服务器上运行示例	1
1.1.3. 运行测试	1
1.2. 第一个例子：注册示例	2
1.2.1. 了解代码	2
1.2.2. 工作原理	11
1.3. Seam中的可点击列表：消息示例	11
1.3.1. 理解代码	12
1.3.2. 工作原理	16
1.4. Seam和jBPM：待办事项列表（todo list）示例	16
1.4.1. 理解代码	17
1.4.2. 工作原理	22
1.5. Seam页面流：猜数字范例	22
1.5.1. 理解代码	23
1.5.2. 工作原理	27
1.6. 一个完整的Seam应用程序：宾馆预订范例	27
1.6.1. 介绍	27
1.6.2. 预订系统概况	29
1.6.3. 理解Seam业务对话(Conversation)	29
1.6.4. Seam的UI控制库	35
1.6.5. Seam调试页面	35
1.7. 一个使用Seam和jBPM的完整范例：DVD商店	36
1.8. 结合Seam和Hibernate的范例：Hibernate预订系统	38
1.9. 一个RESTful的Seam应用程序：Blog范例	38
1.9.1. 使用“拉”风格的MVC	39
1.9.2. 可收藏的搜索结果页面	40
1.9.3. 在RESTful应用程序中使用“推”风格（“push”-style）的MVC	43
2. 用Seam-gen起步	
2.1. 准备活动	46
2.2. 建立一个新的Eclipse项目	46
2.3. 创建新动作	49
2.4. 创建有动作的表单（form）	49
2.5. 从现有数据库生成应用程序	50
2.6. 将应用部署为EAR	51
2.7. Seam与增量热部署	51
2.8. 在Jboss 4.0下使用Seam	52
2.8.1. 安装JBoss 4.0	52
2.8.2. 安装JSF 1.2 RI	52
3. 上下文相关的组件模型	
3.1. Seam上下文	53
3.1.1. Stateless context（无状态上下文）	53

3.1.2. Event context (事件上下文)	53
3.1.3. Page context (页面上下文)	54
3.1.4. Conversation context (业务会话上下文)	54
3.1.5. Session context (Session上下文)	54
3.1.6. Business process context (业务流程上下文)	55
3.1.7. Application context (应用上下文)	55
3.1.8. Context variables (上下文变量)	55
3.1.9. Context搜索优先级	55
3.1.10. 并发模型	56
3.2. Seam 组件	56
3.2.1. 无状态Session Bean	57
3.2.2. 有状态Session Bean	57
3.2.3. 实体Bean	57
3.2.4. JavaBeans	58
3.2.5. 消息驱动Bean	58
3.2.6. 拦截	58
3.2.7. 组件名字	59
3.2.8. 定义组件范围 (Defining the Component Scope)	60
3.2.9. 具有多个角色的组件 (Components with multiple roles)	60
3.2.10. 内置组件	60
3.3. 双向注入	61
3.4. Lifecycle methods (生命周期方法)	63
3.5. 条件装载 (Conditional installation)	63
3.6. 日志	64
3.7. Mutable接口和@ReadOnly	65
3.8. Factory和Manager组件	66
4. 配置Seam组件	
4.1. 通过属性设置来配置组件	69
4.2. 通过 components.xml 来配置组件	69
4.3. 细粒度的配置文件	72
4.4. 可配置的属性类型	72
4.5. 使用XML命名空间	73
5. 事件、拦截器和异常处理	
5.1. Seam事件	77
5.1.1. 页面动作	77
5.1.2. 组件驱动的事件	83
5.1.3. 上下文事件	84
5.2. Seam 拦截器	85
5.3. 管理异常	87
5.3.1. 异常和事务	87
5.3.2. 激活Seam异常处理	87
5.3.3. 使用注解处理异常	88
5.3.4. 用XML处理异常	88
5.3.5. 一些常见的异常	89
6. 对话以及工作区管理	
6.1. Seam的对话模型	91
6.2. 嵌套对话	93
6.3. 使用GET请求来开始一个对话	93
6.4. 利用<s:link>以及<s:button>	94

6.5. 成功信息	96
6.6. 使用“显式”的对话id	96
6.7. 工作区管理	97
6.7.1. 工作区管理及JSF导航	97
6.7.2. 工作区管理和jPDL页面流	97
6.7.3. 对话转换器	98
6.7.4. 对话列表	98
6.7.5. 导航控件	99
6.8. 对话组件和JSF组件绑定	99
6.9. 对话组件的并发调用	100
6.9.1. RichFaces Ajax	101
7. 页面流和业务流程	
7.1. Seam中的页面流	103
7.1.1. 两种导航模型	103
7.1.2. Seam和后退按钮	106
7.2. 使用jPDL页面流	107
7.2.1. 安装页面流	107
7.2.2. 开始页面流	107
7.2.3. 页面节点和跳转	108
7.2.4. 流程控制	109
7.2.5. 流程的结束	109
7.2.6. 页面流组合	109
7.3. Seam中的业务流程管理	110
7.4. 使用jPDL业务流程定义	111
7.4.1. 安装流程定义	111
7.4.2. 初始化Actor id	111
7.4.3. 启动一个业务流程	111
7.4.4. 任务分配	111
7.4.5. 任务列表	112
7.4.6. 执行一个任务	112
8. Seam和对象/关系映射	
8.1. 简介	114
8.2. Seam管理的事务	114
8.2.1. 关闭Seam管理的事务	115
8.2.2. 配置Seam事务管理器	115
8.2.3. 事务同步	116
8.3. Seam管理的持久化上下文	116
8.3.1. 在Seam管理的持久化上下文中使用JPA	116
8.3.2. 使用Seam管理的Hibernate会话	117
8.3.3. Seam管理的持久化上下文和原子会话	117
8.4. 使用JPA “代理(delegate)”	118
8.5. 在EJB-QL/HQL中使用EL	119
8.6. 使用Hibernate过滤器	119
9. Seam中的JSF表单验证	
10. Groovy集成	
10.1. Groovy简介	126
10.2. 用Groovy编写Seam应用	126
10.2.1. 编写Groovy组件	126
10.2.2. seam-gen	128

10.3. 部署	128
10.3.1. 部署Groovy代码	128
10.3.2. 开发时部署本地.groovy文件	128
10.3.3. seam-gen	129
11. Seam应用程序框架	
11.1. 简介	130
11.2. Home对象	131
11.3. Query对象	134
11.4. Controller对象	136
12. Seam和JBoss规则	
12.1. 安装规则	138
12.2. 在Seam组件中使用规则	139
12.3. 在jBPM流程定义中使用规则	139
13. 安全	
13.1. 概述	141
13.1.1. 哪种模式更适合我的应用程序呢?	141
13.2. 需求	141
13.3. 取消安全	141
13.4. 验证	142
13.4.1. 配置	142
13.4.2. 编写验证方法	142
13.4.3. 编写登录表单	144
13.4.4. 简化配置 - 概述	144
13.4.5. 处理安全异常	145
13.4.6. 登录重定向	145
13.4.7. HTTP验证	146
13.4.8. 高级验证特性	147
13.5. 错误消息	147
13.6. 授权	147
13.6.1. 核心概念	147
13.6.2. 保护组件	148
13.6.3. 用户界面中的安全	149
13.6.4. 保护页面	150
13.6.5. 保护实体	150
13.7. 编写安全规则	152
13.7.1. 许可概述	152
13.7.2. 配置规则文件	152
13.7.3. 创建安全规则文件	153
13.8. SSL安全	154
13.9. 实现Captcha测试	155
13.9.1. 配置Captcha Servlet	155
13.9.2. 添加Captcha到页面	156
13.9.3. 定制Captcha图片	156
14. 国际化和主题	
14.1. 本地化	157
14.2. 标签	157
14.2.1. 定义标签	157
14.2.2. 标签显示	158
14.2.3. Faces Messages	159

14.3. 时区	159
14.4. 主题	159
14.5. 使用cookie保存locale和主题设置	160
15. Seam Text	
15.1. 基本格式化	161
15.2. 输入代码和有特殊字符的文本	162
15.3. 链接	163
15.4. 输入HTML	163
16. iText PDF生成	
16.1. 使用PDF支持	165
16.1.1. 创建一个文档	165
16.1.2. 基本的文本元素	166
16.1.3. 页眉和页脚	169
16.1.4. 章节	170
16.1.5. 列表	171
16.1.6. 表格	172
16.1.7. 文档常量	174
16.1.8. iText配置	175
16.2. 图表	175
16.3. 柱状图编码	182
16.4. 更详细的文档	182
17. 电子邮件	
17.1. 创建一条消息	183
17.1.1. 附件	184
17.1.2. HTML/Text 交替部分	185
17.1.3. 多个收件人	185
17.1.4. 多条信息	185
17.1.5. 模板	185
17.1.6. 国际化	186
17.1.7. 其它的标识头	186
17.2. 接收邮件	186
17.3. 配置	187
17.3.1. mailSession	187
17.4. Meldware	188
17.5. 标签	189
18. 异步和消息	
18.1. 异步	192
18.1.1. 异步方法	192
18.1.2. 包含Quartz Dispatcher的异步方法	194
18.1.3. 异步事件	196
18.2. Seam中的消息	196
18.2.1. 配置	197
18.2.2. 发送消息	197
18.2.3. 利用消息驱动Bean接收消息	198
18.2.4. 在客户端接收消息	198
19. 缓存	
19.1. 在Seam中使用JBossCache	199
19.2. 页片段缓存	200
20. Web Services	

20.1. 配置和打包	202
20.2. 对话的Web Services	202
20.2.1. 建议策略	203
20.3. Web Service范例	203
21. Remoting	
21.1. 配置	205
21.2. Seam对象	205
21.2.1. Hello World示例	206
21.2.2. Seam.Component	207
21.2.3. Seam.Remoting	208
21.3. EL表达式求值	209
21.4. 客户端接口	209
21.5. 上下文	210
21.5.1. 设置和读取对话ID	210
21.5.2. 当前对话范围内的远程调用	210
21.6. 批量请求	210
21.7. 使用数据类型	210
21.7.1. 原生 / 基本 类型	210
21.7.2. JavaBeans	211
21.7.3. Date和Time	211
21.7.4. Enums 枚举类型	211
21.7.5. Collections 集合	212
21.8. 调试	212
21.9. 加载消息	213
21.9.1. 修改信息	213
21.9.2. 隐藏加载信息	213
21.9.3. 自定义加载指示器	213
21.10. 控制返回数据	213
21.10.1. 一般字段的约束	214
21.10.2. 集合和映射的约束	214
21.10.3. 特定类型对象的约束	214
21.10.4. 组合约束	215
21.11. JMS消息	215
21.11.1. 配置	215
21.11.2. 订阅JMS主题	215
21.11.3. 退订主题	215
21.11.4. 调整轮询过程	216
22. Seam和Google的Web工具包 (GWT)	
22.1. 配置	217
22.2. 准备你的组件	217
22.3. 将GWT小组件接到Seam组件	218
22.4. GWT Ant Targets	219
23. Spring Framework集成	
23.1. 把Seam组件注入Spring Bean中	221
23.2. 将Spring Bean注入到Seam组件中	222
23.3. 将Spring Bean转换为Seam组件	223
23.4. Seam作用域的Spring Bean	223
23.5. 使用Spring PlatformTransactionManagement	224
23.6. 在Spring中使用Seam管理的持久化上下文	224

23.7.	在Spring中使用Seam管理的Hibernate会话	225
23.8.	作为Seam组件的Spring应用上下文	226
23.9.	使用Spring TaskExecutor的@Asynchronous	226
24.	Hibernate Search	
24.1.	简介	227
24.2.	配置	227
24.3.	用法	228
25.	Seam配置和Seam应用程序打包	
25.1.	Seam基本配置	230
25.1.1.	将Seam与JSF和servlet容器集成	230
25.1.2.	使用Facelets	230
25.1.3.	Seam Resource Servlet	231
25.1.4.	Seam Servlet过滤器	231
25.1.5.	将Seam与你的EJB容器集成	234
25.1.6.	切记!	235
25.2.	在Java EE 5中配置Seam	235
25.2.1.	打包	236
25.3.	在J2EE中配置Seam	237
25.3.1.	在Seam中引导Hibernate	237
25.3.2.	在Seam中引导JPA	238
25.3.3.	打包	238
25.4.	在Java SE中配置Seam, 没有内嵌JBoss	239
25.5.	用嵌入式的JBoss在Java SE中配置Seam	239
25.5.1.	安装嵌入式的JBoss	240
25.5.2.	打包	240
25.6.	在Seam中配置jBPM	241
25.6.1.	打包	242
25.7.	在Portal中配置Seam	242
25.8.	在JBoss AS中配置SFSB和会话超时	243
26.	Seam on OC4J	
26.1.	jee5/booking 实例	244
26.1.1.	预订酒店实例的依赖包	244
26.1.2.	OC4J需要的额外依赖包	244
26.1.3.	配置文件的改变	245
26.1.4.	构建 jee5/booking 实例	245
26.2.	部署Seam应用程序到OC4J中	246
26.3.	将一个使用 seam-gen 创建的应用程序部署到OC4J中。	247
26.3.1.	seam-gen之类的应用程序的OC4J部署描述符	251
27.	Seam注解	
27.1.	用于定义组件的注解	252
27.2.	用于双向注入的注解	254
27.3.	关于组件生命周期方法的注解	257
27.4.	用于声明上下文的注解	257
27.5.	用于在J2EE环境中使用Seam JavaBean组件的注解	260
27.6.	用于异常的注解	261
27.7.	用于Seam Remoting 的注解	261
27.8.	用于Seam拦截器(interceptor)的注解	262
27.9.	用于异步(asynchronicity)的注解	262
27.10.	用于JSF的注解	263

27.10.1. 和 dataTable 一起使用的注解	263
27.11. 用于数据绑定的元数据注解	264
27.12. 用于打包 (packing) 的注解	264
27.13. 用于和Servlet容器集成的注解	265
28. 内置Seam组件	
28.1. 上下文注入组件	266
28.2. 工具组件	266
28.3. 组件的国际化和主题	268
28.4. 控制对话组件	269
28.5. 与jBPM相关的组件	270
28.6. 与安全相关的组件	271
28.7. 与JMS相关的组件	272
28.8. 与邮件相关的组件	272
28.9. 基础组件	272
28.10. 杂项组件	275
28.11. 特殊组件	275
29. Seam的JSF控件	
29.1. 标签	277
29.2. 注解	290
30. 表达式语言增强	
30.1. 参数方法绑定	292
30.1.1. 用法	292
30.1.2. 限制	292
30.2. 参数值绑定	293
30.3. 映射	293
31. 测试Seam应用程序	
31.1. Seam组件的单元测试	295
31.2. Seam组件的集成测试	296
31.2.1. 在集成测试中使用Mock对象	297
31.3. 集成测试Seam应用程序中的用户交互	297
31.3.1. 利用Mock数据进行集成测试	300
31.3.2. Seam Mail集成测试	301
32. Seam工具	
32.1. jBPM设计器和查看器	303
32.1.1. 业务流程设计器	303
32.1.2. Pageflow查看器	303
33. 依赖包	
33.1. 项目依赖包	305
33.1.1. Core	305
33.1.2. Ajax4JSF / RichFaces	306
33.1.3. Seam Mail	306
33.1.4. Seam PDF	306
33.1.5. JBoss Rules	307
33.1.6. jBPM	307
33.1.7. GWT	308
33.1.8. Spring	308
33.1.9. Groovy	308
33.2. 使用Maven依赖管理	308
A. Seam 2.0 开发手册中文翻译项目	

A. 1. 声明	311
A. 2. 项目历程	311
A. 2. 1. Seam 1.2.1 开发手册翻译项目	311
A. 2. 2. Seam 2.0 Beta 1 开发手册翻译项目	311
A. 2. 3. Seam 2.0 正式版开发手册翻译项目	318

JBoss Seam简介

Seam是一种企业级Java的应用程序框架。它的灵感源自下列原则：

只有一种“工具”

Seam为你的应用程序中所有的业务逻辑定义了一种统一的组件模型。 Seam组件可能是有状态的，包含与几个定义良好的上下文中任何一个相关联的状态， 包括长时间运行上下文、持久化上下文、业务流程上下文， 以及用户交互中能够跨多个Web请求保存的对话上下文。

Seam中的表现层组件和业务逻辑组件之间并没有区别。 你可以根据你设计的任何架构给应用程序进行分层，而不是强制将你的应用程序逻辑硬塞进一个由你目前正在使用的任何框架组合所强加给你的不适当的分层配置中。

与简单的Java EE或者J2EE组件不同， Seam组件可以同时访问与Web请求相关的状态，以及保存在事务资源中的状态（而不必通过方法参数手工传播Web请求状态）。 你可能反对说由旧式的J2EE平台强加给你的应用程序分层是件好东西，没有什么可以阻止你利用Seam创建一个相当的分层架构——区别在于，你要自己架构应用程序，并决定有哪些层，以及它们是如何合作的。

将JSF与EJB 3.0整合

JSF和EJB 3.0是Java EE5的两个最好的新特性。EJB3是服务器端业务和持久逻辑的全新组件模型。同时，JSF也是表现层的一个优秀组件模型。不幸的是，这二者都无法独自解决所有的计算问题。实际上，JSF和EJB3结合使用后运作得最好。 但是Java EE5规范并没有提供如何整合这两个组件模型的标准方法。 所幸，这两种模型的创建者都前瞻到了这种状况，并且提供了标准的扩展点，允许对各自进行扩展，或者与其他解决方案集成。

Seam将JSF和EJB3的组件模型合二为一，消除了胶合代码，使得开发者专注于业务问题。

编写“一切”都是EJB的Seam应用程序是有可能的。如果你习惯把EJB当作是细粒度的所谓“重量级”的对象，这可能会令你很吃惊。 然而，从开发人员的角度来看，3.0版本已经完全改变了EJB的本质。 EJB是一个细粒度的对象——没有什么东西会比注解的JavaBean更复杂了。Seam甚至鼓励你使用会话Bean作为JSF动作监听者！

另一方面，如果你宁可不在这个时候采用EJB 3.0，不用勉强。 事实上，任何Java类都可以是一个Seam组件，并且Seam提供了你期待从“轻量化”的容器，甚至任何组件、EJB或者其他东西中获得的所有功能。

集成AJAX

Seam支持两个最好的、开源的基于JSF的AJAX解决方案：JBoss RichFaces和ICEfaces。 这两个解决方案让你无需编写任何JavaScript代码就可以为你的界面添加AJAX功能。

Seam也提供了内置的JavaScript远程访问层，它让你异步地从客户端JavaScript中调用组件，而不需要中间的action层。 你还可以订阅服务器端的JMS主题，并通过AJAX的push方法接收信息。

假若不是有Seam内置的并发和状态管理能力，以上这些方法将都无法很好地运作。 这两种方法确保服务器端能够安全而高效地处理多个并发细粒度的异步AJAX请求。

将业务流程作为首要的基础建筑

Seam可以选择通过jBPM提供透明的业务流程管理。使用jBPM和Seam共同实现复杂的工作流、合作和任务管理，简单到了让人难以置信的程度。

Seam甚至允许你利用与jBPM给业务流程定义所使用的相同语言（jPDL）来定义表现层页面流。

JSF为表示层提供了非常丰富的事件模型。通过以完全相同的事件处理机制暴露与jBPM业务流程相关的事件，Seam强化了这一模型，这就为Seam的统一组件模型提供了统一的事件模型。

声明式状态管理

从EJB早期开始，我们已经习惯于声明式事务管理和J2EE声明式安全的概念。EJB 3.0还引入了声明式持久上下文管理。一个更广泛的管理状态问题——管理与某个特殊context关联的状态，有三个特例，它确保这个上下文结束时进行所有必要的清理。Seam把声明式状态管理的概念推进的远得多，并把它应用于应用程序状态（application state）。J2EE应用程序一般通过手工实现状态管理，通过获取和设置Servlet Session和Request属性。假若程序没能清除Session属性，或者在多窗口的应用程序中，与不同的工作流关联的Session数据发生冲突，这种状态管理的方法就会成为很多Bug和内存泄漏的根源。Seam有可能几乎完全消除这类Bug。

声明式应用程序状态管理通过Seam定义的丰富的context model（上下文模型）而成为可能。Seam扩展了Servlet规范一定义的上下文模型——请求、会话、应用程序——增加了两个新的上下文——对话和业务流程，从业务逻辑的角度来看它们更具意义。

一旦你开始使用对话，将会惊讶于许多事情变得更加容易了。你曾经在像Hibernate或者JPA这样的ORM解决方案中痛苦地处理过延迟的关联抓取吗？Seam对话范围的持久化上下文意味着你将几乎看不到LazyInitializationException。你曾经遇到过刷新（Refresh）按钮或者后退（Back）按钮的问题吗？或者有过重复提交表单的问题吗？有通过post-then-redirect传播信息的问题吗？Seam的对话管理解决了这些问题，甚至无需你真正去关注它们。它们都是自Web最早期以来普遍的不良状态管理架构的征兆。

Bijection（双向注入）

Inversion of Control（控制反转）或者 dependency injection（依赖注入）的概念出现在JSF和EJB3以及很多所谓的“轻型容器”中。这类容器大多注重于实现 stateless services（无状态服务）的组件注射。即便在支持对有状态的组件进行注射的情况下（例如JSF），事实上也难以用于处理应用程序状态，因为有状态组件的范围难以有效而灵活地定义，并且属于更广范围的组件不能被注入到属于更窄范围的组件中。

Bijection（双向注入）和IoC的不同之处在于它是动态的、语境相关的以及双向的。你可以把这一机制理解成将语境相关的变量（与当前线程绑定的各种上下文中的名称）对应到组件的属性中。双向注入允许由容器对有状态的组件进行自动组装。它甚至允许组件可以安全而简单地处理上下文变量的值，只需要把它赋给组件的属性。

工作区管理（Workspace Management）和多窗口浏览

Seam应用程序让用户自由地在多个浏览器窗口中切换，每个窗口都与一个不同的、安全隔离的对话关联。应用程序甚至可以利用 workspace management，允许用户在一个浏览器窗口的多个对话（工作区）之间进行切换。Seam不仅提供正确的多窗口操作，还提供在一个窗口中模拟多个窗口的操作。

更喜欢XML注解

传统上，关于到底哪些元信息可以算作配置，Java社区一直处于一种极为混乱的状态。J2EE和流行的“轻型”容器为真正可以在不同的系统部署之间配置的东西，以及任何不容易用Java表达的其他声明都提供了基于XML的部署描述符。Java 5 注解改变了所有这一切。

EJB3.0 接受注解和“对例外配置”，这成了以声明的形式为容器提供信息的最简易方法。不幸的是，JSF仍然在十分依赖笨重的XML配置文件。Seam扩展了EJB 3.0 提供的注解，以用于声明式状态管理和声明式上下文划分。这让你摆脱了对繁琐的JSF managed bean(JSF受管bean)的配

置，减少了所需的XML，只剩下那些真正属于XML的信息（JSF导航规则）。

集成测试轻而易举

Seam组件作为POJO，天生就是可以进行单元测试的。但是对于复杂的应用程序，只有单元测试则还不够。对于Java Web应用程序来说，集成测试一般是一项笨拙且困难的任务。因此，Seam为Seam应用程序提供了可测试性作为该框架的一项核心功能。你可以轻易地编写重现与用户完整交互的JUnit或TestNG测试，来演习除了视图View（JSP或者Facelets页面）之外的所有系统组件。你可以直接从你的IDE中运行这些测试，Seam会在那里自动地利用JBoss Embeddable部署EJB组件。

规范也非尽善尽美

我们认为最新的Java EE规范很不错。但是我们知道它还远不够完美。在规范中有许多漏洞（例如，GET请求的JSF生命周期中的局限性），Seam修正了这些漏洞。Seam的创建者们正与JCP专家组一道，确保这些修正恢复到标准的下一次修订中。

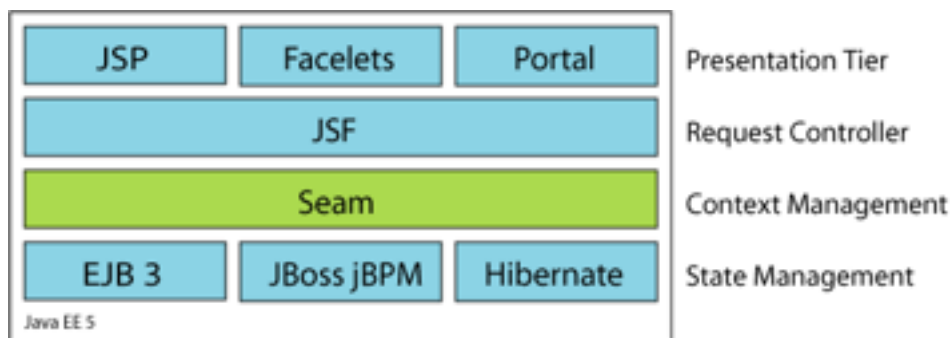
Web应用程序不只是服务HTML页面

当今的Web框架认为太小了。它们让你叫用户输入表单，并进入到你的Java对象。然后它们就让你悬着。真正完整的Web应用程序框架应该解决像持久化、并发、异步、状态管理、安全、电子邮件、信息、PDF和图表生成、工作流、wikitext渲染、Web Services、缓存等等更多的问题。一旦你尝到了Seam的甜头，就会惊讶地发现许多问题都变得更加简单了.....

Seam为持久化集成了JPA和Hibernate 3，为轻量化的异步性集成了EJB Timer Service和Quartz，为工作流集成了jBPM，为业务规则集成了JBoss规则，为电子邮件集成了Meldware Mail，为完整的文本搜索集成了Hibernate Search和Lucene，为消息集成了JMS，以及为页面片断捕捉集成了JBoss Cache。Seam在JAAS和JBoss规则之上，创建了一个新的基于规则的安全框架。甚至有用来渲染PDF、在线电子邮件和图表及wikitext的JSF标签库。Seam组件可以同时作为一个Web Service进行调用，异步地从客户端JavaScript或者Google Web Toolkit，或者当然也可以直接从JSF调用。

立刻开始吧！

Seam在任何Java EE应用程序服务器中都可以运行，甚至在Tomcat中也可以。如果你的环境支持EJB 3.0，好极了！如果不支持，也没关系，你可以使用Seam为持久化内置的包含JPA或者Hibernate3的事务管理。或者，你可以在Tomcat中部署JBoss Embedded，同时享有对EJB 3.0 的完整支持。



最终你会发现，Seam、JSF和EJB3的组合就是用Java编写复杂Web应用程序的最简单办法。你不会相信所需的代码是多么地少！

本文档翻译由俞黎敏作为Leader组织。翻译及一审、二审名单见下。王琳、马越对全书进行了三审。俞黎敏进行了全书统稿及发布的build工作。

表 1. 翻译及审核人员列表

章节	KB	页数	翻译	一审
master.xml	17K		CaoXiaogang	Echo
1. tutorial.xml (1.1-1.4)	130K	20P	seanchan	Jacky
(1.5-1.11)		24P	DigitalSonic	Jacky
2. gettingstarted.xml	21K	6P	seanchan	zaya
3. concepts.xml	56K	16P	CaoXiaogang	kuuyee
4. xml.xml	25K	7P	downpour	Echo
5. events.xml	39K	11P	mochow	xihuyu2000
6. conversations.xml	34K	10P	magice	Echo
7. jbpm.xml	32K	10P	差沙	ronghao
8. persistence.xml	23K	6P	pesome	caoer
9. validation.xml	9K	4P	pesome	DigitalSonic
10. groovy.xml	11K	4P	kuuyee	DigitalSonic
11. framework.xml	20K	7P	alexchang	CaoXiaogang
12. drools.xml	7K	3P	DigitalSonic	shaozhou
13. security.xml	51K	14P	YuLimin	xihuyu2000
14. i18n.xml	14K	4P	YY	DigitalSonic
15. text.xml	7K	3P	DigitalSonic	yeshucheng(万国辉)
16. itext.xml	51K	11P	lyfcdy	Echo
17. mail.xml	26K	7P	chentianyi	yeshucheng(万国辉)
18. jms.xml	11K	5P	YuLimin	caoer
19. cache.xml	11K	3P	crazycy	CaoXiaogang
20. webservices.xml	9K	3P	Echo	YuLimin
21. remoting.xml	37K	13P	crazycy	agile_boy
22. gwt.xml	10K	4P	yeshucheng(万国辉)	Echo
23. spring.xml	13K	4P	YY	caoer
24. hsearch.xml	7K	3P	yeshucheng(万国辉)	agile_boy
25. configuration.xml	48K	15P	yeby	kuuyee
26. annotations.xml	64K	14P	caoer	CaoXiaogang
27. components.xml	68K	11P	jiaochar	zaya
28. controls.xml	47K	13P	Echo	YuLimin

章节	KB	页数	翻译	一审
29. elenhancements.xml	5K	2P	CaoXiaogang	yeshucheng(万国辉)
30. testing.xml	10K	6P	agile_boy	CaoXiaogang
31. tools.xml	23K	9P	junjzheng	CaoXiaogang
32. oc4j.xml	31K	8P	yeshucheng(万国辉)	YuLimin
33. dependencies.xml	26K	5P	yeshucheng(万国辉)	DigitalSonic

第 1 章 Seam 入门

1.1. 试试看

本教程假定你已下载JBoss AS 4.0.5并安装了EJB 3.0 profile（请使用JBoss AS安装器）。你也得下载一份Seam并解压到工作目录上。

各示例的目录结构仿效以下形式：

- 网页、图片及样式表可在 `examples/registration/view` 目录中找到。
- 诸如部署描述文件及数据导入脚本之类的资源可在目录 `examples/registration/resources` 中找到。
- Java源代码保存在 `examples/registration/src` 中。
- Ant构建脚本放在 `examples/registration/build.xml` 文件中。

1.1.1. 在JBoss AS上运行示例

第一步，确保已安装Ant，并正确设定了 `$ANT_HOME` 及 `$JAVA_HOME` 的环境变量。接着在Seam的根目录下的 `build.properties` 文件中正确设定JBoss AS 4.0.5的安装路径。若一切就绪，就可在JBoss的安装根目录下敲入 `bin/run.sh` 或 `bin/run.bat` 命令来启动JBoss AS。（译注：此外，请安装JDK1.5以上以便能直接运行示例代码）

现在只要在Seam安装目录 `examples/registration` 下输入 `ant deploy` 就可构建和部署示例了。

试着在浏览器中访问此链接：<http://localhost:8080/seam-registration/>。

1.1.2. 在Tomcat服务器上运行示例

首先，确保已安装Ant，并正确设定了 `$ANT_HOME` 及 `$JAVA_HOME` 的环境变量。接着在Seam的根目录下的 `build.properties` 文件中正确设定Tomcat 6.0的安装路径。你需要按照25.5.1章节“安装嵌入式的Jboss”中的指导配置（当然，SEAM也可以脱离Jboss在TOMCAT上直接运行）。

至此，就可在Seam安装目录 `examples/registration` 中输入 `ant deploy.tomcat` 构建和部署示例了。

最后启动Tomcat。

试着在浏览器中访问此链接：<http://localhost:8080/jboss-seam-registration/>。

当你部署示例到Tomcat时，任何的EJB3组件将在JBoss的可嵌入式的容器，也就是完全独立的EJB3容器环境中运行。

1.1.3. 运行测试

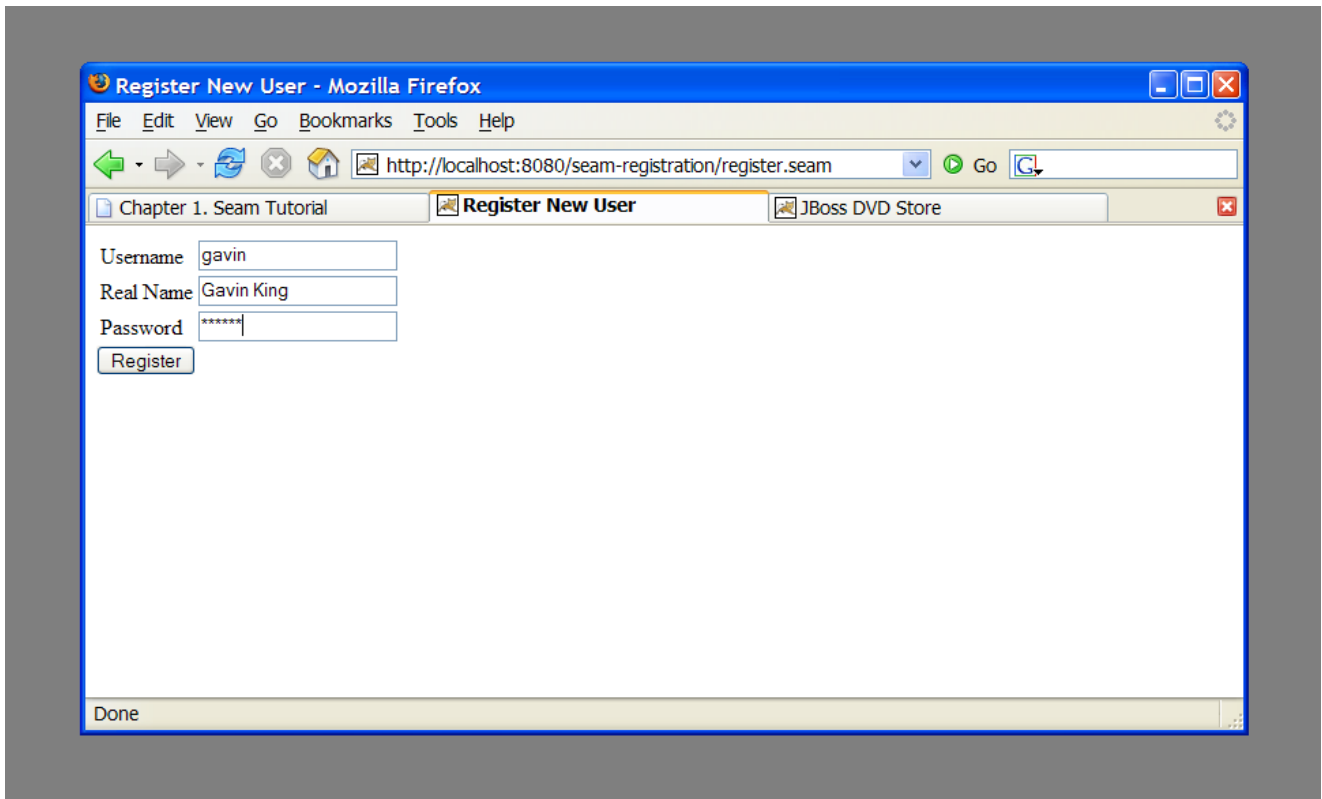
几乎所有的示例都有相应的TestNG的集成测试代码。最简便的运行测试代码的方法是在 `examples/registration` 目录中运行 `ant testexample`。当然也可在IDE开发工具中使用TestNG插件来运行测试。

1.2. 第一个例子：注册示例

注册示例是个极其普通的应用，它可让新用户在数据库中保存自己的用户名，真实的姓名及密码。此示例并不想一下子就把Seam的所有的酷功能全部秀出。然而，它演示了EJB3 会话Bean作为JSF动作监听器及Seam的基本配置的使用方法。

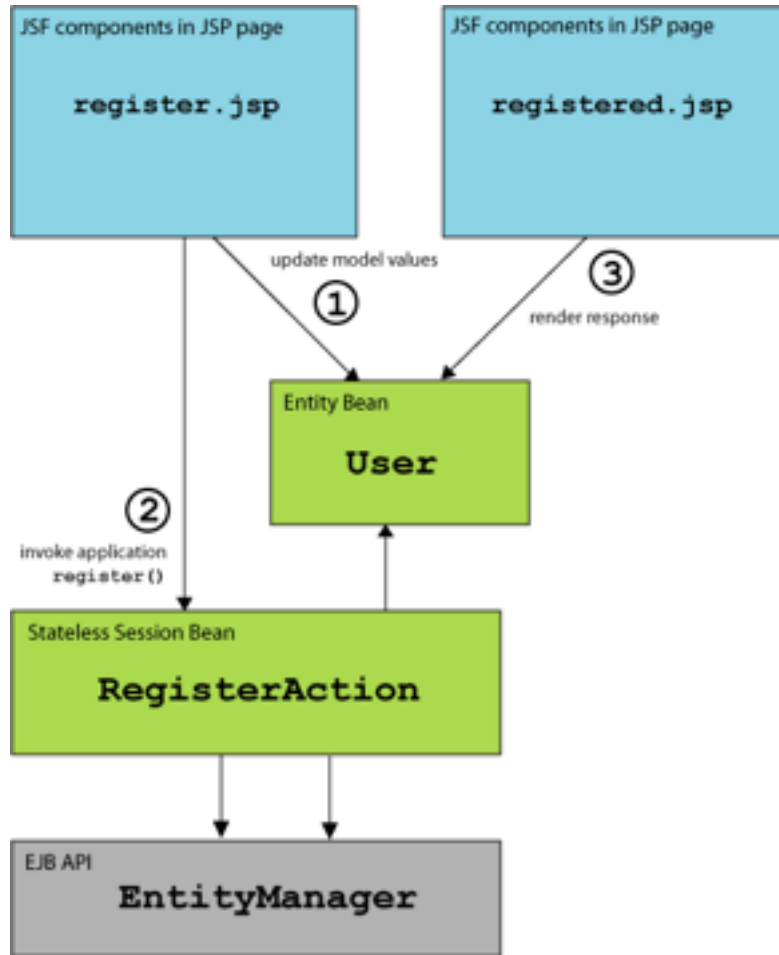
或许你对EJB 3.0还不太熟悉，因此我们会对示例的慢慢深入说明。

此示例的首页显示了一个非常简单的表单，它有三个输入字段。试着在表单上填写内容并提交，一旦输入数据被提交后就会在数据库中保存一个user对象。



1.2.1. 了解代码

本示例由两个JSP页面，一个实体Bean及无状态的会话Bean来实现。



让我们看一下代码，就从最“底层”的实体Bean开始吧。

1.2.1.1. 实体Bean: User.java

我们需要EJB 实体Bean来保存用户数据。这个类通过注解声明性地定义了 `persistence` 及 `validation` 属性。它也需要一些额外的注解来将这个类定义为Seam的组件。

例 1.1.

```

@Entity
@Name("user")
@Scope(SESSION)
@Table(name="users")
public class User implements Serializable
{
    private static final long serialVersionUID = 1881413500711441951L;

    private String username;
    private String password;
    private String name;

    public User(String name, String password, String username)
    {
        this.name = name;
        this.password = password;
        this.username = username;
    }

    public User() {}

```

Annotations 1-4 are on the first four lines. Annotation 5 is on the `private String username;` line. Annotation 6 is on the `public User() {}` line.

```

@NotNull @Length(min=5, max=15) ❶
public String getPassword()
{
    return password;
}

public void setPassword(String password)
{
    this.password = password;
}

@NotNull
public String getName()
{
    return name;
}

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

@Id @NotNull @Length(min=5, max=15) ❷
public String getUsername()
{
    return username;
}

public void setUsername(String username)
{
    this.username = username;
}
}

```

- ❶ EJB3标准注解 `@Entity` 表明了 `User` 类是个实体Bean。
- ❷ Seam组件需要一个 组件名称，此名称由注解 `@Name`来指定。此名称必须在Seam应用内唯一。当JSF用一个与组件同名的名称去请求Seam来解析上下文变量，且该上下文变量尚未定义（null）时，Seam就将实例化那个组件，并将新实例绑定给上下文变量。在此例中，Seam将在JSF第一次遇到名为 `user` 的变量时实例化 `User`。
- ❸ 每当Seam实例化一个组件时，它就将始化后的实例绑定给组件中 默认上下文 的上下文变量。默认的上下文由 `@Scope`注解指定。`User` Bean是个会话作用域的组件。
- ❹ EJB标准注解`@Table` 表明了将 `User` 类映射到 `users` 表上。
- ❺ `name`、`password` 及 `username` 都是实体Bean的持久化属性。所有的持久化属性都定义了访问方法。当JSF渲染输出及更新模型值阶段时需要调用该组件的这些方法。
- ❻ EJB和Seam都要求有空的构造器。
- ❼ `@NotNull` 和 `@Length` 注解是Hibernate Validator框架的组成部份，Seam集成了Hibernate Validator并让你用它来作为数据校验（尽管你可能并不使用Hibernate作为持久化层）。
- ❽ 标准EJB注解 `@Id` 表明了实体Bean的主键属性。

这个例子中最值得注意的是 `@Name` 和 `@Scope` 注解，它们确立了这个类是Seam的组件。

接下来我们将看到 `User` 类字段在更新模型值阶段时直接被绑定给JSF组件并由JSF操作，在此并不需要冗余的胶水代码来在JSP页面与实体Bean域模型间来回拷贝数据。

然而，实体Bean不应该进行事务管理或数据库访问。故此，我们无法将此组件作为JSF动作监听器，因而需要会话Bean。

1.2.1.2. 无状态会话Bean: RegisterAction.java

在Seam应用中大都采用会话Bean来作为JSF动作监听器（当然我们也可选择JavaBean）。

在我们的应用程序中确实存在一个JSF动作和一个会话Bean方法。在此示例中，只有一个JSF动作，并且我们使用会话Bean方法与之相关联并使用无状态Bean，这是由于所有与动作相关的状态都保存在 User Bean中。

这是示例中比较有趣的代码部份：

例 1.2.

```

@Stateless                                ❶
@Name("register")
public class RegisterAction implements Register
{

    @In                                    ❷
    private User user;

    @PersistenceContext                    ❸
    private EntityManager em;

    @Logger                                ❹
    private Log log;

    public String register()                ❺
    {
        List existing = em.createQuery(
            "select username from User where username=#{user.username}")
            .getResultList();                ❻

        if (existing.size()==0)
        {
            em.persist(user);
            log.info("Registered new user #{user.username}");
            return "/registered.jsp";        ❼
        }
        else
        {
            FacesMessages.instance().add("User #{user.username} already exists"); ❾
            return null;
        }
    }
}

```

- ❶ EJB标准注解 `@Stateless` 将这个类标记为无状态的会话Bean。
- ❷ 注解 `@In`将Bean的一个属性标记为由Seam来注入。在此例中，此属性由名为 `user` 的上下文变量注入（实例的变量名）。
- ❸ EJB标准注解 `@PersistenceContext` 用来注入EJB实体管理器。
- ❹ Seam的 `@Logger` 注解用来注入组件的 `Log` 实例。
- ❺ 动作监听器方法使用标准的EJB3 `EntityManager` API来与数据库交互，并返回JSF的输出结果。请注意，由于这是个会话Bean，因此当 `register()` 方法被调用时事务就会自动开始，并在结束时提交（commit）。
- ❻ 请注意Seam让你在EJB-QL中使用JSF EL表达式。因此可在标准JPA `Query` 对象上调用普通的JPA `setParameter()` 方法，这样岂不妙哉？

- ⑦ Log API为显示模板化的日志消息提供了便利。
- ⑧ 多个JSF动作监听器方法返回一个字符串值的输出，它决定了接下来应显示的页面内容。空输出（或返回值为空的动作监听器方法）重新显示上一页的内容。在普通的JSF中，用JSF的导航规则（navigation rule）来决定输出结果的JSF视图id是很常用的。这种间接性对于复杂的应用是非常有用的，值得去实践。但是，对于象示例这样简单的应用，Seam让你使用JSF视图id作为输出结果，以减少对导航规则的需求。请注意，当你用视图id作为输出结果时，Seam总会执行一次浏览器的重定向。
- ⑨ Seam提供了大量的 内置组件（built-in components）来协助解决那些经常遇到的问题。用 FacesMessages 组件就可很容易地来显示模板化的错误或成功的消息。内置的Seam组件还可由注入或通过调用 instance() 方法来获取。

这次我们并没有显式指定 @Scope，若没有显式指定时，每个Seam 组件类型就使用其默认的作用域。对于无状态的会话Bean，其默认的作用域就是无状态的上下文。实际上 所有的 无状态的会话Bean都属于无状态的上下文。

会话Bean的动作监听器在此小应用中履行了业务和持久化逻辑。在更复杂的应用中，我们可能要将代码分层并重构持久化逻辑层成 专用数据存取组件，这很容易做到。但请注意Seam并不强制你在应用分层时使用某种特定的分层策略。

此外，也请注意我们的SessionBean会同步访问与web请求相关联的上下文（比如在 User 对象中的表单的值），状态会被保持在事务型的资源里（EntityManager 对象）。这是对传统J2EE的体系结构的突破。再次说明，如果你习惯于传统J2EE的分层，也可以在你的Seam应用实行。但是对于许多的应用，这是明显的没有必要。

1.2.1.3. 会话Bean的本地接口：Register.java

很自然，我们的会话Bean需要一个本地接口。

例 1.3.

```
@Local
public interface Register
{
    public String register();
}
```

所有的Java代码就这些了，现在去看一下部署描述文件。

1.2.1.4. Seam组件部署描述文件：components.xml

如果你此前曾接触过许多的Java框架，你就会习惯于将所有的组件类放在某种XML文件中来声明，那些文件就会随着项目的不断成熟而不断加大到最终到不可收拾的地步。对于Seam应用，你尽可放心，因为它并不要求应用组件都要有相应的XML。大部份的Seam应用要求非常少量的XML即可，且XML文件大小不会随着项目的增大而快速增长。

无论如何，若能为 某些 组件（特别是Seam内置组件）提供某些 外部配置往往是有用的。这样一来，我们就有几个选择，但最灵活的选择还是使用位于 WEB-INF 目录下的 components.xml 配置文件。我们将用 components.xml 文件来演示Seam怎样在JNDI中找到EJB组件：

例 1.4.

```
<components xmlns="http://jboss.com/products/seam/components"
             xmlns:core="http://jboss.com/products/seam/core">
  <core:init jndi-pattern="@jndiPattern@"/>
</components>
```

此代码配置了Seam内置组件 `org.jboss.seam.core.init` 的 `jndiPattern` 属性。这里需要奇怪的@符号是因为ANT脚本会在部署应用时将正确的JNDI语法在标记处自动填补

1.2.1.5. Web部署描述文件: web.xml

我们将以WAR的形式来部署此小应用的表示层，因此需要web部署描述文件。

例 1.5.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

  <!-- Seam -->

  <listener>
    <listener-class>org.jboss.seam.servlet.SeamListener</listener-class>
  </listener>

  <!-- MyFaces -->

  <listener>
    <listener-class>
      org.apache.myfaces.webapp.StartupServletContextListener
    </listener-class>
  </listener>

  <context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>client</param-value>
  </context-param>

  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <!-- Faces Servlet Mapping -->
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.seam</url-pattern>
  </servlet-mapping>

</web-app>
```

此 web.xml 文件配置了Seam和JSF。所有Seam应用中的配置与此处的配置基本相同。

1.2.1.6. JSF配置: faces-config.xml

绝大多数的Seam应用将JSF来作为表示层。因而我们通常需要 faces-config.xml。SEAM将用Facelet定义视图表现层，所以我们需要告诉JSF用Facelet作为它的模板引擎。

例 1.6.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE faces-config
PUBLIC "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
"http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
<faces-config>

  <!-- A phase listener is needed by all Seam applications -->

  <lifecycle>
    <phase-listener>org.jboss.seam.jsf.SeamPhaseListener</phase-listener>
  </lifecycle>

</faces-config>
```

注意我们不需要申明任何JSF managed Bean! 因为我们所有的managed Bean都是通过经过注释的Seam组件。所以在Seam的应用中，faces-config.xml比原始的JSF更少用到。

实际上，一旦你把所有的基本描述文件配置完毕，你所需写的 唯一类型的 XML文件就是导航规则及可能的jBPM流程定义。对于Seam而言， 流程（process flow） 及 配置数据 是唯一真正属于需要XML定义的。

在此简单的示例中，因为我们将视图页面的ID嵌入到Action代码中，所以我们甚至都不需要定义导航规则。

1.2.1.7. EJB部署描述文件: ejb-jar.xml

ejb-jar.xml 文件将 SeamInterceptor 绑定到压缩包中所有的会话Bean上，以此实现了Seam与EJB3的整合。

```
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
  version="3.0">

  <interceptors>
    <interceptor>
      <interceptor-class>org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
    </interceptor>
  </interceptors>

  <assembly-descriptor>
    <interceptor-binding>
      <ejb-name>*</ejb-name>
      <interceptor-class>org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
    </interceptor-binding>
  </assembly-descriptor>
```



```
</ejb-jar>
```

1.2.1.8. EJB持久化部署描述文件：persistence.xml

persistence.xml 文件告诉EJB的持久化层在哪找到数据源，该文件也含有一些厂商特定的设定。此例在程序启动时自动创建数据库Schema。

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">
  <persistence-unit name="userDatabase">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
    </properties>
  </persistence-unit>
</persistence>
```

1.2.1.9. 视图：register.xhtml 和 registered.xhtml

对于Seam应用的视图可由任意支持JSF的技术来实现。在此例中，我们使用了JSP，因为大多数的开发人员都很熟悉，且这里并没有其它太多的要求。（我们建议你在实际开发中使用Facelets）。

例 1.7.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://jboss.com/products/seam/taglib" prefix="s" %>
<html>
<head>
  <title>Register New User</title>
</head>
<body>
  <f:view>
    <h:form>
      <table border="0">
        <s:validateAll>
          <tr>
            <td>Username</td>
            <td><h:inputText value="#{user.username}"/></td>
          </tr>
          <tr>
            <td>Real Name</td>
            <td><h:inputText value="#{user.name}"/></td>
          </tr>
          <tr>
            <td>Password</td>
            <td><h:inputSecret value="#{user.password}"/></td>
          </tr>
        </s:validateAll>
      </table>
      <h:messages/>
      <h:commandButton type="submit" value="Register" action="#{register.register}"/>
    </h:form>
  </f:view>
</body>
```

```
</html>
```

这里的 `<s:validateAll>` 标签是 Seam 特有的。该 JSF 组件告诉 JSF 让它用实体 Bean 中所指定的 Hibernate 验证器注解来验证所有包含输入的字段。

例 1.8.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<html>
  <head>
    <title>Successfully Registered New User</title>
  </head>
  <body>
    <f:view>
      Welcome, <h:outputText value="#{user.name}" />,
      you are successfully registered as <h:outputText value="#{user.username}" />.
    </f:view>
  </body>
</html>
```

这是个极其普通的使用 JSF 组件的 JSP 页面，与 Seam 毫无相干。

1.2.1.10. EAR 部署描述文件：application.xml

最后，因为我们的应用是要部署成 EAR 的，因此我们也需要部署描述文件。

例 1.9.

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/application_5.xsd"
  version="5">

  <display-name>Seam Registration</display-name>

  <module>
    <web>
      <web-uri>jboss-seam-registration.war</web-uri>
      <context-root>/seam-registration</context-root>
    </web>
  </module>
  <module>
    <ejb>jboss-seam-registration.jar</ejb>
  </module>
  <module>
    <java>jboss-seam.jar</java>
  </module>
  <module>
    <java>el-api.jar</java>
  </module>
  <module>
    <java>el-ri.jar</java>
  </module>
```

```
</application>
```

此部署描述文件联接了EAR中的所有模块，并把Web应用绑定到此应用的首页 `/seam-registration`。

至此，我们了解了整个应用中 所有的 部署描述文件！

1.2.2. 工作原理

当提交表单时，JSF请求Seam来解析名为 `user` 的变量。由于还没有值绑定到 `user` 上（在任意的Seam上下文中），Seam就会实例化 `user`组件，接着把它保存在Seam会话上下文后，然后将 `User` 实体Bean实例返回给JSF。

表单输入的值将由在 `User` 实体中所指定的Hibernate验证器来验证。若有非法输入，JSF就重新显示当前页面。否则，JSF就将输入值绑定到 `User` 实体Bean的字段上。

接着，JSF请求Seam来解析变量 `register`。Seam在无状态上下文中找到 `RegisterAction` 无状态的会话Bean并把它返回。JSF随之调用 `register()` 动作监听器方法。

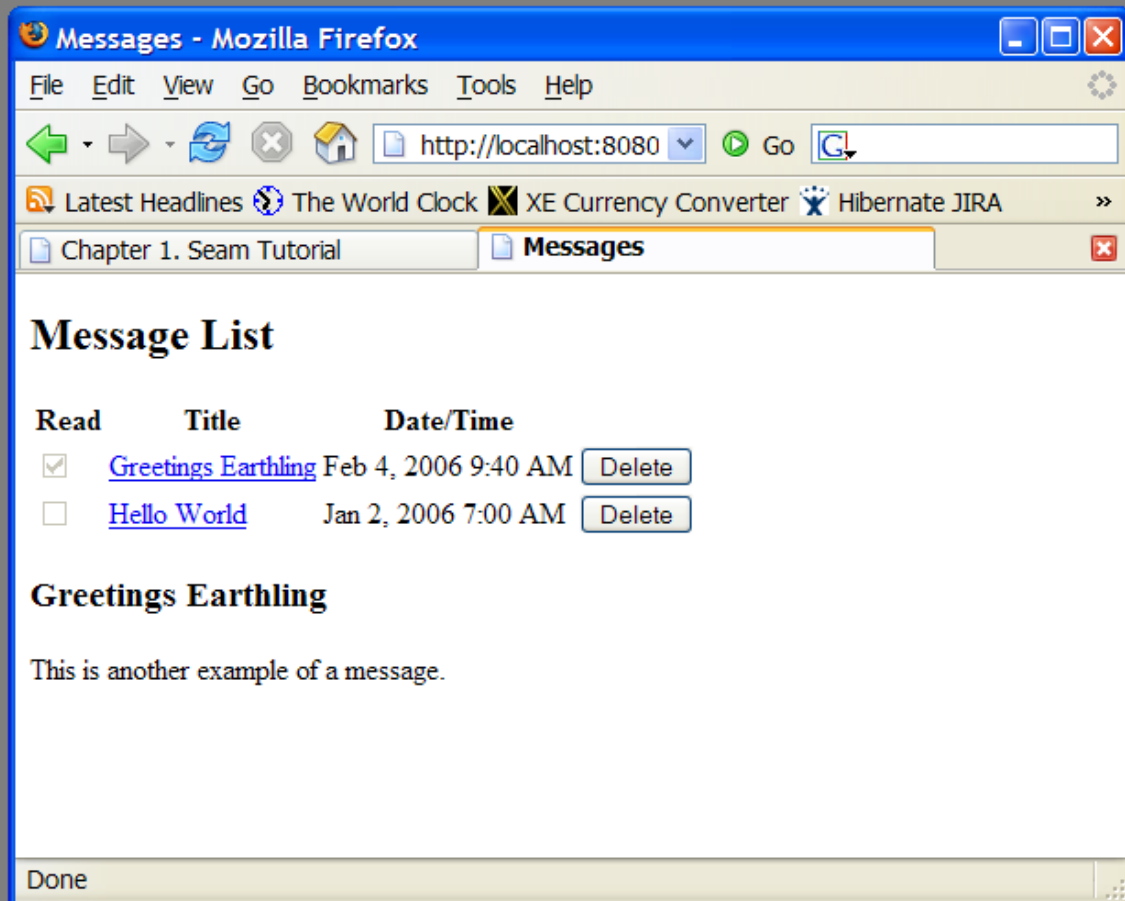
Seam拦截方法调用并在继续调用之前从Seam会话上下文注入 `User` 实体。

`register()` 方法检查所输入用户名的用户是否已存在。若存在该用户名，则错误消息进入 `facesmessages` 组件队列，返回无效结果并触发浏览器重显页面。`facesmessages` 组件嵌在消息字符串的JSF表达式，并将JSF `facesmessage` 添加到视图中。

若输入的用户不存在，`"/registered.jsp"` 输出就会将浏览器重定向到 `registered.jsp` 页。当JSF来渲染页面时，它请求Seam来解析名为 `user` 的变量，并使用从Seam会话作用域返回的 `User` 实体的属性值。

1.3. Seam中的可点击列表：消息示例

在几乎所有的在线应用中都免不了将搜索结果显示成可点击的列表。因此Seam在JSF层之上提供了特殊的功能，使得我们很容易用EJB-QL或HQL来查询数据并用JSF `<h:dataTable>` 将查询结果显示成可点击的列表。我们将在接下的例子中演示这一功能。



1.3.1. 理解代码

此消息示例中有一个实体Bean，Message，一个会话Bean MessageListBean 及一个JSP页面。

1.3.1.1. 实体Bean: Message.java

Message 实体定义了消息的title, text, date和time以及该消息是否已读的标志:

例 1.10.

```
@Entity
@Name("message")
@Scope(EVENT)
public class Message implements Serializable
{
    private Long id;
    private String title;
    private String text;
    private boolean read;
    private Date datetime;

    @Id @GeneratedValue
    public Long getId() {
        return id;
    }
}
```

```

public void setId(Long id) {
    this.id = id;
}

@NotNull @Length(max=100)
public String getTitle() {
    return title;
}
public void setTitle(String title) {
    this.title = title;
}

@NotNull @Lob
public String getText() {
    return text;
}
public void setText(String text) {
    this.text = text;
}

@NotNull
public boolean isRead() {
    return read;
}
public void setRead(boolean read) {
    this.read = read;
}

@NotNull
@Basic @Temporal(TemporalType.TIMESTAMP)
public Date getDatetime() {
    return datetime;
}
public void setDatetime(Date datetime) {
    this.datetime = datetime;
}
}

```

1.3.1.2. 有状态的会话Bean: MessageManagerBean.java

如此前的例子，会话Bean `MessageManagerBean` 用来给表单中的两个按钮定义个动作监听器方法，其中的一个按钮用来从列表中选择消息，并显示该消息。而另一个按钮则用来删除一条消息，除此之外，就没什么特别之处了。

在用户第一次浏览消息页面时，`MessageManagerBean` 会话Bean也负责抓取消息列表，考虑到用户可能以多种方式来浏览该页面，他们也有可能不是由JSF动作来完成，比如用户可能将该页加入收藏夹。因此抓取消息列表发生在Seam的工厂方法中，而不是在动作监听器方法中。

之所以将此会话Bean设为有状态的，是因为我们想在不同的服务器请求间缓存此消息列表。

例 1.11.

```

@Stateful
@Scope(SESSION)
@Name("messageManager")
public class MessageManagerBean implements Serializable, MessageManager
{

```

```

@DataModel
private List<Message> messageList;

@DataModelSelection
@Out(required=false)
private Message message;

@PersistenceContext(type=EXTENDED)
private EntityManager em;

@Factory("messageList")
public void findMessages()
{
    messageList = em.createQuery("from Message msg order by msg.datetime desc").getResultList();
}

public void select()
{
    message.setRead(true);
}

public void delete()
{
    messageList.remove(message);
    em.remove(message);
    message=null;
}

@Remove @Destroy
public void destroy() {}
}

```

- ❶ 注解 `@DataModel` 暴露了 `java.util.List` 类型的属性给JSF页面来作为 `javax.faces.model.DataModel` 的实例。这允许我们在JSF `<h:dataTable>` 的每一行中能使用可点击列表。在此例中，`DataModel` 可在变量名为 `messageList` 的会话上下文中被使用。
- ❷ `@DataModelSelection` 注解告诉了Seam来注入 `List` 元素到相应的被点击链接。
- ❸ 注解 `@Out` 直接暴露了被选中的值给页面。这样一来，每次可点击列表一旦被选中，`Message` 就会被注入给有状态Bean的属性，紧接着 向外注入（outjected）给变量名为 `message` 的事件上下文的属性。
- ❹ 此有状态Bean有个EJB3的 扩展持久化上下文（extended persistence context）。只要Bean存在，查询中获取的消息就会保留在受管理的状态中。这样一来，此后对有状态Bean的所有方法调用 勿需显式调用 `EntityManager` 就可更新这些消息了。
- ❺ 当我们第一次浏览JSP页面时，`messageList` 上下文变量尚未被初始化，`@Factory` 注解告诉Seam来创建 `MessageManagerBean` 的实例并调用 `findMessages()` 方法来初始化上下文变量。我们把 `findMessages()` 当作 `messages` 的 工厂方法。
- ❻ `select()` 将选中的 `Message` 标为已读，并同时更新数据库。
- ❼ `delete()` 动作监听器方法将选中的 `Message` 从数据库中删除。
- ❽ 对于每个有状态的会话Bean，Seam组件的所有方法中 必须 有一不带参数的方法被标为 `@Remove` `@Destroy` 以确保在Seam的上下文结束时删除有状态Bean，并同时清除所有服务器端的状态。

请注意，这是个会话作用域的Seam组件。它与用户登入会话相关联，并且登入会话的所有请求共享同一个组件的实例。（在Seam的应用中，我们通常使用会话作用域的组件。）

1.3.1.3. 会话Bean的本地接口：MessageManager.java

当然，每个会话Bean都有个业务接口。

```
@Local
public interface MessageManager
{
    public void findMessages();
    public void select();
    public void delete();
    public void destroy();
}
```

从现在起，我们在示例代码中将不再对本地接口作特别的说明。

由于XML文件与此前的示例几乎都一样，因此我们略过了 components.xml、persistence.xml、web.xml、ejb-jar.xml、faces-config.xml 及 application.xml 的细节，直接来看一下JSP。

1.3.1.4. 视图：messages.jsp

JSP页面就是直接使用JSF <h:dataTable> 的组件，并没有与Seam有什么关系。

例 1.12.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<html>
<head>
<title>Messages</title>
</head>
<body>
<f:view>
<h:form>
<h2>Message List</h2>
<h:outputText value="No messages to display" rendered="#{messageList.rowCount==0}" />
<h:dataTable var="msg" value="#{messageList}" rendered="#{messageList.rowCount>0}">
<h:column>
<f:facet name="header">
<h:outputText value="Read"/>
</f:facet>
<h:selectBooleanCheckbox value="#{msg.read}" disabled="true"/>
</h:column>
<h:column>
<f:facet name="header">
<h:outputText value="Title"/>
</f:facet>
<h:commandLink value="#{msg.title}" action="#{messageManager.select}" />
</h:column>
<h:column>
<f:facet name="header">
<h:outputText value="Date/Time"/>
</f:facet>
<h:outputText value="#{msg.datetime}">
<f:convertDateTime type="both" dateStyle="medium" timeStyle="short"/>
</h:outputText>
</h:column>
<h:column>
<h:commandButton value="Delete" action="#{messageManager.delete}" />
</h:column>
</h:dataTable>
<h3><h:outputText value="#{message.title}" /></h3>
<div><h:outputText value="#{message.text2}" /></div>
</h:form>
```

```
</f:view>
</body>
</html>
```

1.3.2. 工作原理

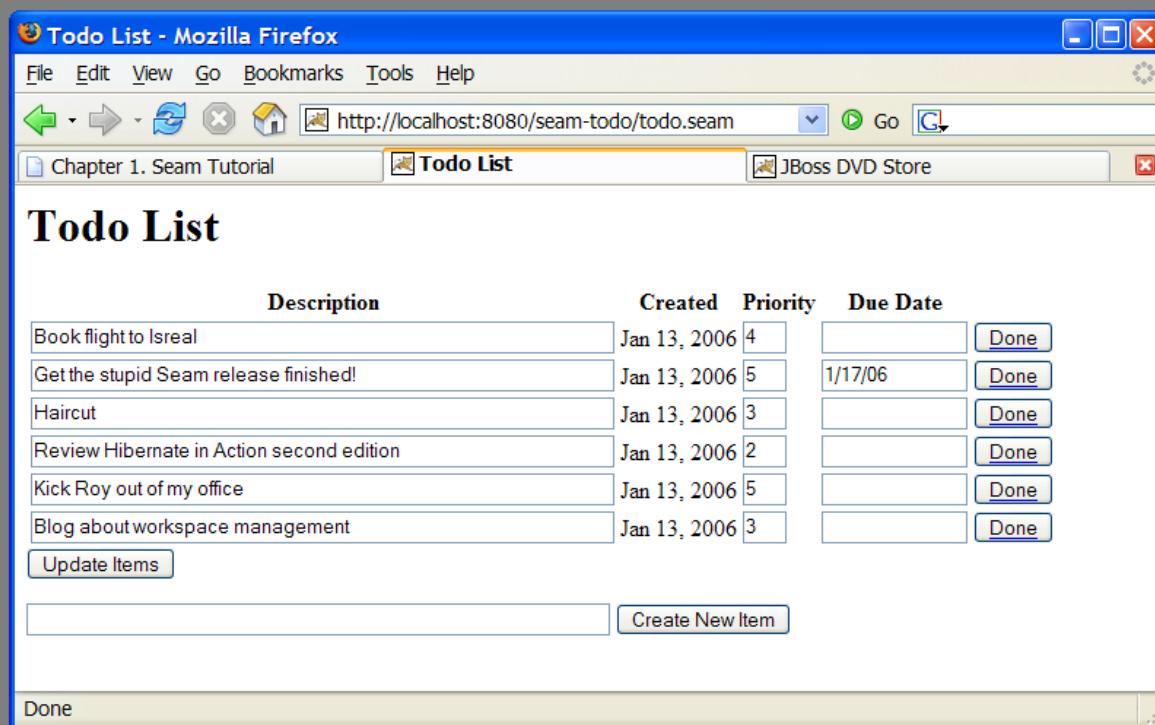
当我们首次浏览 `messages.jsp` 页面时，无论是否由回传（postback）的JSF（页面请求）或浏览器直接的GET请求（非页面请求），此JSP页面将设法解析 `messageList` 上下文变量。由于上下文变量尚未被初始化，因此Seam将调用工厂方法 `findmessages()`，该方法执行了一次数据库查询并导致 `DataModel` 被向外注入。`DataModel` 提供了渲染 `<h:dataTable>` 所需的行数据。

当用户点击 `<h:commandLink>` 时，JSF就调用 `Select()` 动作监听器。Seam拦截此调用并将所选行的数据注入给 `messageManager` 组件的 `message` 属性。而动作监听器将所选定的 `Message` 标为已读。在此调用结束时，Seam向外注入所选定的 `Message` 给名为 `message` 的变量。接着，EJB容器提交事务，将 `Message` 的已读标记写入数据库。最后，该网页重新渲染，再次显示消息列表，并在列表下方显示所选消息的内容。

如果用户点击了 `<h:commandButton>`，JSF就调用 `delete()` 动作监听器。Seam拦截此调用并将所选行的数据注入给 `messageManager` 组件的 `message` 属性。触发动作监听器，将选定的 `Message` 从列表中删除并同时在 `EntityManager` 中调用 `remove()` 方法。在此调用的最后，Seam刷新 `messageList` 上下文变量并清除名为 `message` 的上下文变量。接着，EJB容器提交事务，将 `Message` 从数据库中删除。最后，该网页重新渲染，再次显示消息列表。

1.4. Seam和jBPM：待办事项列表（todo list）示例

jBPM提供了先进的工作流程和任务管理的功能。为了体验一下jBPM是如何与Seam集成在一起工作的，在此将给你一个简单的管理“待办事项列表”的应用。由于管理任务列表等功能是jBPM的核心功能，所以在此例中只用了很少的Java代码。



1.4.1. 理解代码

这个例子的核心是jBPM的流程定义（process definition）。此外，还有两个JSP页面和两个简单的JavaBeans（由于他们不用访问数据库，或有其它事务相关的行为，因此并没有用会话Bean）。让我们先从流程定义开始：

例 1.13.

```
<process-definition name="todo">

  <start-state name="start">                                ❶
    <transition to="todo"/>
  </start-state>

  <task-node name="todo">                                    ❷
    <task name="todo" description="#{todoList.description}"> ❸
      <assignment actor-id="#{actor.id}"/>                    ❹
    </task>
    <transition to="done"/>
  </task-node>

  <end-state name="done"/>                                   ❺

</process-definition>
```

- ❶ 节点 `<start-state>` 代表流程的逻辑开始。一旦流程开始时，它就立即转入 `todo` 节点。
- ❷ `<task-node>` 节点代表 等待状态，就是在执行业务流程暂停时，等待一个或多个未完成的任务。
- ❸ `<task>` 元素定义了用户需要完成的任务。由于在这个节点只有定义了一个任务，当它完成，或恢复执行时我们就转入结束状态。此任务从Seam中名为 `todolist` 的组件（JavaBeans之一）获得任

务description。

- ④ 任务在创建时就会被分配给一个用户或一组用户时。在此示例中，任务是分配给当前用户，该用户从一个内置的名为 `actor` 的Seam组件中获得。任何Seam组件都可用来执行任务指派。
- ⑤ `<end-state>`节点定义业务流程的逻辑结束。当执行到达这个节点时，流程实例就要被销毁。

如果我们用jBossIDE所提供的流程定义编辑器来查看此流程定义，那它就会是这样：



这个文档将我们的 业务流程 定义成节点图。这可能是最常见的业务流程：只有一个 任务 被执行，当这项任务完成之后，业务流程就结束了。

第一个JavaBean处理登入界面 `login.jsp`。它的工作就是用 `actor` 组件初始化jBPM用户id（在实际的应用中，它也需要验证用户。）

例 1.14.

```

@Name("login")
public class Login {

    @In
    private Actor actor;

    private String user;

    public String getUser() {
        return user;
    }

    public void setUser(String user) {
        this.user = user;
    }

    public String login()
    {

```

```

    actor.setId(user);
    return "/todo.jsp";
}
}

```

在此我们使用了 `@In` 来将actor属性值注入到Seam内置的 `Actor` 组件。

JSP页面本身并没有什么特别之处：

例 1.15.

```

<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<html>
<head>
<title>Login</title>
</head>
<body>
<h1>Login</h1>
<f:view>
    <h:form>
        <div>
            <h:inputText value="#{login.user}"/>
            <h:commandButton value="Login" action="#{login.login}"/>
        </div>
    </h:form>
</f:view>
</body>
</html>

```

第二个JavaBean负责启动业务流程实例及结束任务。

例 1.16.

```

@Name("todoList")
public class TodoList {

    private String description;

    public String getDescription() ❶
    {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    @CreateProcess(definition="todo") ❷
    public void createTodo() {}

    @StartTask @EndTask ❸
    public void done() {}

}

```

- ❶ description属性从JSP页接受用户输入，并将它暴露给流程定义，这样就可让Seam来设定任务的description。
- ❷ Seam的 @CreateProcess 注解为指定名称的流程定义创建了一个新的jBPM流程实例。
- ❸ Seam的 @StartTask 注解用来启动任务，@EndTask 用来结束任务，并允许恢复执行业务流程。

在实际的应用中，@StartTask 及 @EndTask 不会出现在同一个方法中，因为为了完成任务，通常用应用中有许多工作要做。

最后，该应用的主要内容在 todo.jsp 中：

例 1.17.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://jboss.com/products/seam/taglib" prefix="s" %>
<html>
<head>
<title>Todo List</title>
</head>
<body>
<h1>Todo List</h1>
<f:view>
  <h:form id="list">
    <div>
      <h:outputText value="There are no todo items." rendered="#{empty taskInstanceList}"/>
      <h:dataTable value="#{taskInstanceList}" var="task" rendered="#{not empty taskInstanceList}">
        <h:column>
          <f:facet name="header">
            <h:outputText value="Description"/>
          </f:facet>
          <h:inputText value="#{task.description}"/>
        </h:column>
        <h:column>
          <f:facet name="header">
            <h:outputText value="Created"/>
          </f:facet>
          <h:outputText value="#{task.taskMgmtInstance.processInstance.start}"
            <f:convertDateTime type="date"/>
          </h:outputText>
        </h:column>
        <h:column>
          <f:facet name="header">
            <h:outputText value="Priority"/>
          </f:facet>
          <h:inputText value="#{task.priority}" style="width: 30"/>
        </h:column>
        <h:column>
          <f:facet name="header">
            <h:outputText value="Due Date"/>
          </f:facet>
          <h:inputText value="#{task.dueDate}" style="width: 100">
            <f:convertDateTime type="date" dateStyle="short"/>
          </h:inputText>
        </h:column>
        <h:column>
          <s:button value="Done" action="#{todoList.done}" taskInstance="#{task}"/>
        </h:column>
      </h:dataTable>
    </div>
    <div>
      <h:messages/>
    </div>
  </f:view>
</h:form>
</body>
</html>
```

```

<div>
  <h:commandButton value="Update Items" action="update"/>
</div>
</h:form>
<h:form id="new">
  <div>
    <h:inputText value="#{todoList.description}"/>
    <h:commandButton value="Create New Item" action="#{todoList.createTodo}"/>
  </div>
</h:form>
</f:view>
</body>
</html>

```

让我们对此逐一加以说明。

该JSP页面将从Seam内置组件 `taskInstanceList` 获得的任务渲染成任务列表，此列表在JSF表单内被定义。

```

<h:form id="list">
  <div>
    <h:outputText value="There are no todo items." rendered="#{empty taskInstanceList}"/>
    <h:dataTable value="#{taskInstanceList}" var="task" rendered="#{not empty taskInstanceList}">
      ...
    </h:dataTable>
  </div>
</h:form>

```

列表中的每个元素就是一个jBPM类 `taskinstance` 的实例。以下代码简单地展示了列表中每一任务的有趣特性。为了让用户能更改description、priority及due date的值，我们使用了输入控件。

```

<h:column>
  <f:facet name="header">
    <h:outputText value="Description"/>
  </f:facet>
  <h:inputText value="#{task.description}"/>
</h:column>
<h:column>
  <f:facet name="header">
    <h:outputText value="Created"/>
  </f:facet>
  <h:outputText value="#{task.taskMgmtInstance.processInstance.start}">
    <f:convertDateTime type="date"/>
  </h:outputText>
</h:column>
<h:column>
  <f:facet name="header">
    <h:outputText value="Priority"/>
  </f:facet>
  <h:inputText value="#{task.priority}" style="width: 30"/>
</h:column>
<h:column>
  <f:facet name="header">
    <h:outputText value="Due Date"/>
  </f:facet>
  <h:inputText value="#{task.dueDate}" style="width: 100">
    <f:convertDateTime type="date" dateStyle="short"/>
  </h:inputText>
</h:column>

```

该按钮通过调用被注解为 `@StartTask @EndTask` 的动作方法来结束任务。它把任务id作为请求参数传给Seam:

```
<h:column>
  <s:button value="Done" action="#{todoList.done}" taskInstance="#{task}" />
</h:column>
```

(请注意, 这是在使用Seam seam-ui.jar 包中的JSF `<s:button>` 控件。)

这个按钮是用来更新任务属性。当提交表单时, Seam和jBPM将直接更改任务的持久化, 不需要任何的动作监听器方法:

```
<h:commandButton value="Update Items" action="update" />
```

第二个表单通过调用注解为 `@CreateProcess`的动作方法来创建新的项目 (item)。

```
<h:form id="new">
  <div>
    <h:inputText value="#{todoList.description}" />
    <h:commandButton value="Create New Item" action="#{todoList.createTodo}" />
  </div>
</h:form>
```

这个例子还需要另外几个文件, 但它们只是标准的jBPM和Seam配置并不是很有趣。

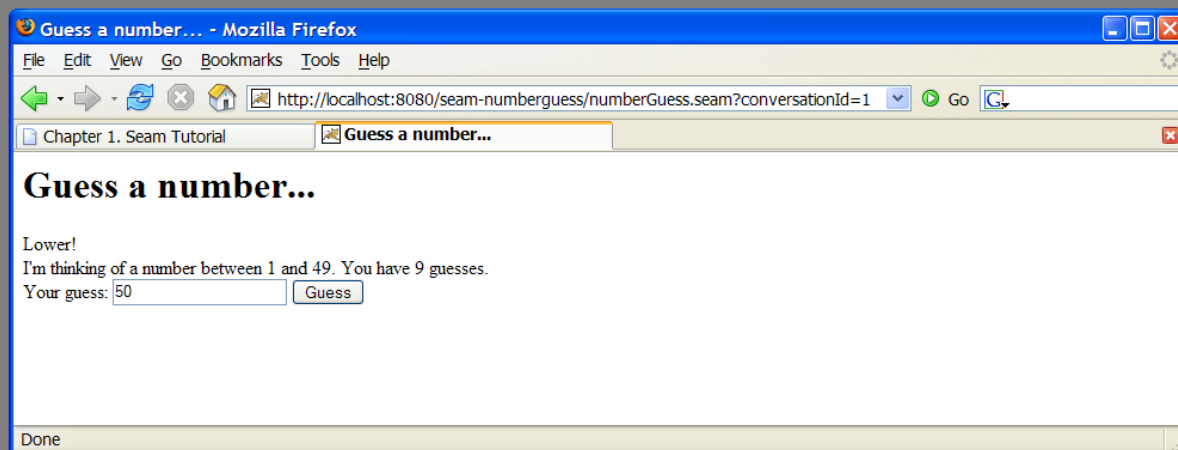
1.4.2. 工作原理

待完成

1.5. Seam页面流: 猜数字范例

对有相对自由 (特别) 导航的Seam应用程序而言, JSF/Seam导航规则是定义页面流的一个完美的方法。而对于那些带有更多约束的导航, 特别是带状态的用户界面而言, 导航规则反而使得系统流程变得难以理解。要理解整个流程, 你需要从视图页面、动作和导航规则里一点点把它拼出来。

Seam允许你使用一个jPDL流程定义来定义页面流。下面这个简单的猜数字范例将演示这一切是如何实现的。



1.5.1. 理解代码

这个例子由一个JavaBean、三个JSP页面和一个jPDL页面流定义组成。让我们从页面流开始：

例 1.18.

```
<pageflow-definition name="numberGuess">

  <start-page name="displayGuess" view-id="/numberGuess.jsp">
    <redirect/>
    <transition name="guess" to="evaluateGuess">
      <action expression="#{numberGuess.guess}" />
    </transition>
  </start-page>

  <decision name="evaluateGuess" expression="#{numberGuess.correctGuess}">
    <transition name="true" to="win"/>
    <transition name="false" to="evaluateRemainingGuesses"/>
  </decision>

  <decision name="evaluateRemainingGuesses" expression="#{numberGuess.lastGuess}">
    <transition name="true" to="lose"/>
    <transition name="false" to="displayGuess"/>
  </decision>

  <page name="win" view-id="/win.jsp">
    <redirect/>
    <end-conversation />
  </page>

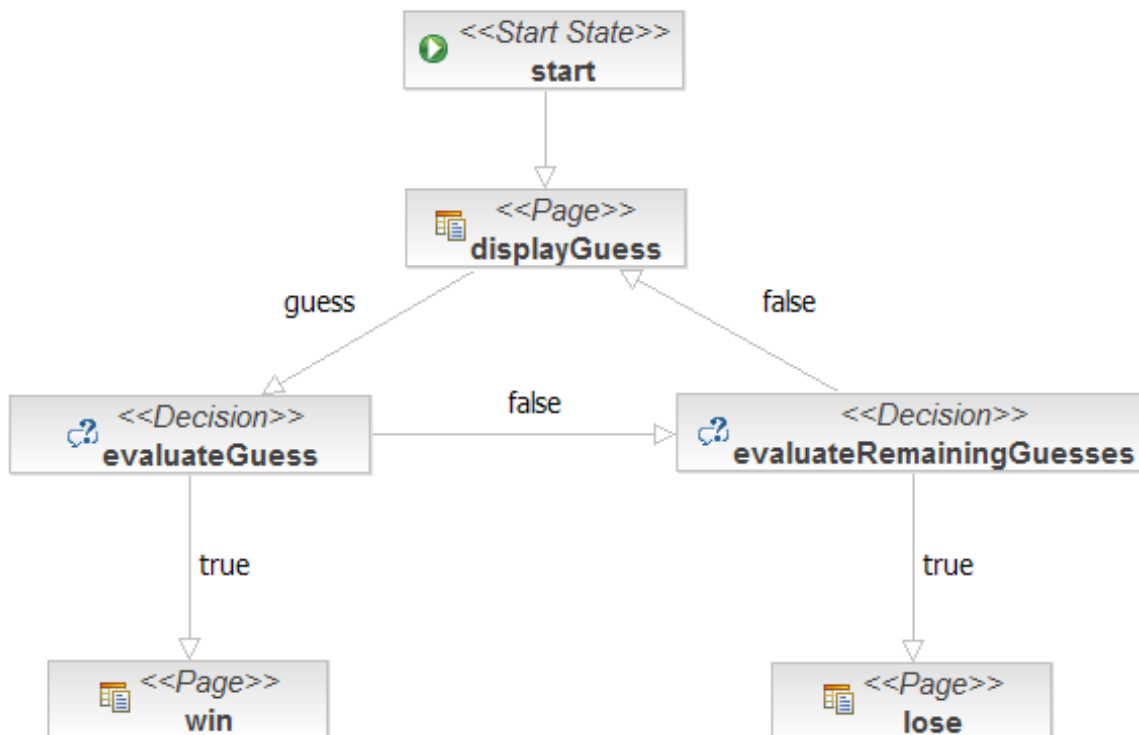
  <page name="lose" view-id="/lose.jsp">
    <redirect/>
    <end-conversation />
  </page>

</pageflow-definition>
```

- ❶ <page>元素定义了一个等待状态，在该状态中系统显示一个JSF视图等待用户输入。 view-id与简单JSF导航规则中的view id一样。 redirect属性告诉Seam在导航到页面时使用post-then-redirect。（这会带来友好的浏览器URL。）

- ❷ <transition> 元素命名了一个JSF输出。当一个JSF动作导致那个输出时会触发转换。在任何jBPM转换动作调用后，执行会进行到页面流程图的下一个节点。
- ❸ 一个转换动作 <action> 就像JSF动作，不同的就是它只发生在一个jBPM转换发生时。转换动作能调用任何Seam组件。
- ❹ <decision> 节点用来划分页面流，通过计算JSF EL表达式决定要执行的下一个节点。

这个页面流在JBossIDE页面流编辑器里看上去是这个样子的：



看过了页面流，现在再来理解剩下的程序就变得十分简单了！

这是应用程序的主页面numberGuess. jsp:

例 1. 19.

```

<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<html>
<head>
<title>Guess a number...</title>
</head>
<body>
<h1>Guess a number...</h1>
<f:view>
  <h:form>
    <h:outputText value="Higher!" rendered="#{numberGuess.randomNumber>numberGuess.currentGuess}" />
    <h:outputText value="Lower!" rendered="#{numberGuess.randomNumber<numberGuess.currentGuess}" />
    <br />
    I'm thinking of a number between <h:outputText value="#{numberGuess.smallest}" /> and
    <h:outputText value="#{numberGuess.biggest}" />. You have
    <h:outputText value="#{numberGuess.remainingGuesses}" /> guesses.
    <br />
    Your guess:
    <h:inputText value="#{numberGuess.currentGuess}" id="guess" required="true">
  
```



```

        <f:validateLongRange
            maximum="#{numberGuess.biggest}"
            minimum="#{numberGuess.smallest}" />
    </h:inputText>
    <h:commandButton type="submit" value="Guess" action="guess" />
    <br/>
    <h:message for="guess" style="color: red" />
</h:form>
</f:view>
</body>
</html>

```

请注意名为 `guess` 的命令按钮是如何进行转换而不是直接调用一个动作的。

`win.jspx` 页面的内容是可想而知的：

例 1.20.

```

<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<html>
<head>
<title>You won!</title>
</head>
<body>
<h1>You won!</h1>
<f:view>
    Yes, the answer was <h:outputText value="#{numberGuess.currentGuess}" />.
    It took you <h:outputText value="#{numberGuess.guessCount}" /> guesses.
    Would you like to <a href="numberGuess.seam">play again</a>?
</f:view>
</body>
</html>

```

`lose.jsp` 也差不多（我就不重复复制/粘贴了）。最后，JavaBean Seam组件是这样的：

例 1.21.

```

@Name("numberGuess")
@Scope(ScopeType.CONVERSATION)
public class NumberGuess {

    private int randomNumber;
    private Integer currentGuess;
    private int biggest;
    private int smallest;
    private int guessCount;
    private int maxGuesses;

    @Create
    @Begin(pageflow="numberGuess")
    public void begin()
    {
        randomNumber = new Random().nextInt(100);
        guessCount = 0;
        biggest = 100;
        smallest = 1;
    }
}

```

```
}

public void setCurrentGuess(Integer guess)
{
    this.currentGuess = guess;
}

public Integer getCurrentGuess()
{
    return currentGuess;
}

public void guess()
{
    if (currentGuess > randomNumber)
    {
        biggest = currentGuess - 1;
    }
    if (currentGuess < randomNumber)
    {
        smallest = currentGuess + 1;
    }
    guessCount++;
}

public boolean isCorrectGuess()
{
    return currentGuess == randomNumber;
}

public int getBiggest()
{
    return biggest;
}

public int getSmallest()
{
    return smallest;
}

public int getGuessCount()
{
    return guessCount;
}

public boolean isLastGuess()
{
    return guessCount == maxGuesses;
}

public int getRemainingGuesses() {
    return maxGuesses - guessCount;
}

public void setMaxGuesses(int maxGuesses) {
    this.maxGuesses = maxGuesses;
}

public int getMaxGuesses() {
    return maxGuesses;
}

public int getRandomNumber() {
    return randomNumber;
}
}
```

- ❶ 一开始，JSP页面请求一个 `numberGuess` 组件，Seam会为该组件创建一个新的实例，并调用 `@Create` 方法，允许组件初始化自己。
- ❷ `@Begin` 注解启动了一个Seam 业务会话(conversation)（稍后详细说明），并指定业务会话页面流所要使用的页面流定义。

如你所见，这个Seam组件是纯业务逻辑的！它不需要知道任何关于用户交互的东西。这点使得组件更易被复用。

1.5.2. 工作原理

TODO

1.6. 一个完整的Seam应用程序：宾馆预订范例

1.6.1. 介绍

该系统是一个完整的宾馆客房预订系统，它由下列功能组成：

- 用户注册
- 登录
- 注销
- 设置密码
- 搜索宾馆
- 选择宾馆
- 客房预订
- 预订确认
- 当前预订列表

jboss suites
seam framework demo
Welcome Gavin King | Search | Settings | Logout

State management in Seam

State in Seam is *contextual*. When you click "Find Hotels", the application retrieves a list of hotels from the database and caches it in the session context. When you navigate to one of the hotel records by clicking the "View Hotel" link, a *conversation* begins. The conversation is attached to a particular tab, in a particular browser window. You can navigate to multiple hotels using "open in new tab" or "open in new window" in your web browser. Each window will execute in the context of a different conversation. The application keeps state associated with your hotel booking in the conversation context, which ensures that the concurrent conversations do not interfere with each other.

[How does the search page work?](#)

Thank you, Gavin King, your confirmation number for Doubletree is 1

Search Hotels

Atlanta **Find Hotels**

Maximum results: 10

Name	Address	City, State	Zip	Action
Marriott Courtyard	Tower Place, Buckhead	Atlanta, GA, USA	30305	View Hotel
Doubletree	Tower Place, Buckhead	Atlanta, GA, USA	30305	View Hotel
Ritz Carlton	Peachtree Rd, Buckhead	Atlanta, GA, USA	30326	View Hotel

Current Hotel Bookings

Name	Address	City, State	Check in date	Check out date	Confirmation number	Action
Doubletree	Tower Place, Buckhead	Atlanta, GA	Apr 16, 2006	Apr 17, 2006	1	Cancel

Created with JBoss EJB 3.0, Seam, MyFaces, and Facelets

应用程序中使用了JSF、EJB 3.0和Seam，视图部分结合了Facelets。也可以选择使用JSF、Facelets、Seam、JavaBeans和Hibernate3。

在使用过一段时间后你会发现该应用程序非常健壮。你能使用回退按钮、刷新浏览器、打开多个窗口，或者键入各种无意义的数，会发现都很难让它崩溃。你也许会想我们花了几个星期测试修复该系统才达到了这个目标。事实却不是这样的，Seam的设计使你能够用它方便地构建健壮的web应用程序，而且Seam还提供了很多以前需要通过编码才能实现的健壮性。

在你浏览范例程序代码研究它是如何运行时，注意观察声明式的状态管理和集成的验证是如何被用来实现这种健壮性的。

1.6.2. 预订系统概况

这个项目的结构和上一个一样，要安装部署该应用程序请参考第 1.1 节 “试试看”。当应用程序启动后，可以通过 <http://localhost:8080/seam-booking/> 进行访问。

只需要用9个类（加上6个Session Bean的本地接口）就能实现这个应用程序。6个Session Bean动作监听器包括了以下功能的所有业务逻辑。

BookingListAction

ChangePasswordAction

MotelBookingAction 业务对话（conversation）

RegisterAction

应用程序的持久化模型由三个实体bean实现。

Motel

Booking

User

1.6.3. 理解Seam业务对话 (Conversation)

我们鼓励您随意浏览源代码。在这个教程里我们将关注功能中的某一特定部分：宾馆搜索、选择、预订和确认。从用户的角度来看，从选择宾馆到确认的每一步都是工作中的一个连续单元，属于一个业务对话。然而搜索却不是该对话的一部分。用户能在不同浏览器标签页中的相同搜索结果页面中选择多个宾馆。

大多数Web应用程序架构没有提供表示业务对话的一级构件(first class construct)。这在管理与对话相关的状态时带来了很多麻烦。通常情况下，Java的Web应用程序结合两种技术来应对这一情况：一是将某些状态丢入 HttpSession；二是将可持久化的状态在每个请求 (Request) 后写入数据库，并在每个新请求的开始将之重建。

由于数据库是最不可扩展的一层，因此这么做往往导致完全无法接受的扩展性低下。在每次请求时访问数据库所造成的额外流量和等待时间也是一个问题。要降低冗余流量，Java应用程序常引入一个(二级)数据缓存来保存被经常访问的数据。然而这个缓存是很低效的，因为它的失效算法是基于LRU（最近最少使用）策略，而不是基于用户何时结束与该数据相关的工作。此外，由于该缓存被许多并发事务共享，要保持缓存与数据库的状态一致，我们需要引入了一套完整的机制。

现在再让我们考虑将状态保存在 HttpSession 里。通过精心设计的编程，我们也许能控制session数据的大小。但这远比听起来要麻烦的多，因为Web浏览器允许特殊的非线性导航。但假设我们在系统开发到一半的时候突然发现一个需求，它要求用户可以拥有 多并发业务对话（我就碰到过）。要开发一些机制，以分离与不同并发业务会话相关的session状态，并引入故障保护，在用户关闭浏览器窗口或标签页时销毁业务会话状态。这对普通人来说可不是一件轻松的事情（我就实现过两次，一次是为一个客户应用程序，另一次是为Seam，幸好我是出了名的疯子）。

现在提供一个更好的方法。

Seam引入了 对话上下文 来作为一级构件。你能在其中安全地保存业务对话状态，它会保证状态有一个定义良好的生命周期。而且，你不用再不停地在应用服务器和数据库间传递数据，因为业务对话上下文就是一个天然的缓存，用来缓存用户的数据。

通常情况下，我们保存在业务对话上下文中的组件是有状态的Session Bean。（我们也在其中保存实体Bean和JavaBeans。）在Java社区中一直有一个谣传，认为有状态的Session Bean是扩展性的杀手。在1998年WebFoobar 1.0发布时的确如此。但今天的情况已经变了。像JBoss 4.0这样的应用服务器都有很成熟的机制处理有状态Session Bean的状态复制。（例如，JBoss EJB3容器可以执行很细致的复制，只复制那些属性值被改变过的bean。）请注意，所有那些传统技术中关于有状态Bean是低效的争论也同样发生在 HttpSession 上，所以说将状态从业务层的有状态Session Bean迁移到Web Session中以提高性能的做法毫无疑问是被误导的。不正确地使用有状态的Bean，或者是将它们用在错误的地方上都会使应用程序变得无法扩展。但这并不意味着你应该 永远不要 使用它们。总之，Seam会告诉你一个安全使用的模型。欢迎来到2005年。

OK，不再多说了，话题回到这个指南上吧。

宾馆预订范例演示了不同作用域的有状态组件是如何协同工作实现复杂的行为的。它的主页面允许用户搜索宾馆。搜索的结果被保存在Seam的session域中。当用户导航到其中一个宾馆时，一个业务会话便开始了，一个业务会话域组件回调session域组件以获得选中的宾馆。

宾馆预订范例还演示了如何使用Ajax4JSF在不用手工编写JavaScript的情况下实现富客户端（Rich Client）行为。

搜索功能用了一个Session域的有状态Session Bean来实现，有点类似于我们在上面的消息列表范例里看到的那个Session Bean。

例 1.22.

```

@Stateful                                ❶
@Name("hotelSearch")
@Scope(ScopeType.SESSION)
@Restrict("#{identity.loggedIn}")        ❷
public class HotelSearchingAction implements HotelSearching
{

    @PersistenceContext
    private EntityManager em;

    private String searchString;
    private int pageSize = 10;
    private int page;

    @DataModel
    private List<Hotel> hotels;            ❸

    public String find()
    {
        page = 0;
        queryHotels();
        return "main";
    }

    public String nextPage()
    {
        page++;
        queryHotels();
        return "main";
    }
}

```

```

}

private void queryHotels()
{
    String searchPattern = searchString==null ? "" : '%' + searchString.toLowerCase().replace('*', '%') + '%';
    hotels = em.createQuery("select h from Hotel h where lower(h.name) like :search or lower(h.city) like :search or lower(h.zip) li
        .setParameter("search", searchPattern)
        .setMaxResults(pageSize)
        .setFirstResult( page * pageSize )
        .getResultList();
}

public boolean isNextPageAvailable()
{
    return hotels!=null && hotels.size()==pageSize;
}

public int getPageSize() {
    return pageSize;
}

public void setPageSize(int pageSize) {
    this.pageSize = pageSize;
}

public String getSearchString()
{
    return searchString;
}

public void setSearchString(String searchString)
{
    this.searchString = searchString;
}

@Destroy @Remove
public void destroy() {}
}

```

- ❶ EJB标准中的 `@Stateful` 注解表明这个类是一个有状态的Session Bean。它们的默认作用域是业务对话上下文。
- ❷ `@Restrict`注解给组件加上了一个安全限制。只有登录过的用户才能访问该组件。安全章节中更详细地讨论了Seam的安全问题。
- ❸ `@DataModel` 注解将一个 `List` 作为JSF `ListDataModel` 暴露出去。这简化了搜索界面的可单击列表的实现。在这个例子中，宾馆的列表是以名为 `hotels` 的 `ListDataModel` 业务对话变量暴露给页面的。
- ❹ EJB标准中的 `@Remove` 注解指定了一个有状态的Session Bean应该在注解的方法被调用后被删除且其状态应该被销毁。在Seam里，所有有状态的Session Bean都应该定义一个标有 `@Destroy @Remove` 的方法。这是Seam在销毁Session上下文时要调用的EJB删除方法。实际上 `@Destroy` 注解更有用，因为它能在Seam上下文结束时被用来做各种各样的清理工作。如果没有一个 `@Destroy @Remove` 方法，那么状态会泄露，你就会碰到性能上的问题。

应用程序的主页面是一个Facelets页面。让我们来看下与宾馆搜索相关的部分：

例 1. 23.

```

<div class="section">
<h:form>

```

```

<span class="errors">
  <h:messages globalOnly="true"/>
</span>

<h1>Search Hotels</h1>
<fieldset>
  <h:inputText value="#{hotelSearch.searchString}" style="width: 165px;"
    <a:support event="onkeyup" actionListener="#{hotelSearch.find}"
      reRender="searchResults" />
  </h:inputText>
  &#160;
  <a:commandButton value="Find Hotels" action="#{hotelSearch.find}"
    styleClass="button" reRender="searchResults"/>
  &#160;
  <a:status>
    <f:facet name="start">
      <h:graphicImage value="/img/spinner.gif"/>
    </f:facet>
  </a:status>
  <br/>
  <h:outputLabel for="pageSize">Maximum results:</h:outputLabel>&#160;
  <h:selectOneMenu value="#{hotelSearch.pageSize}" id="pageSize">
    <f:selectItem itemLabel="5" itemValue="5"/>
    <f:selectItem itemLabel="10" itemValue="10"/>
    <f:selectItem itemLabel="20" itemValue="20"/>
  </h:selectOneMenu>
</fieldset>

</h:form>
</div>

<a:outputPanel id="searchResults">
  <div class="section">
    <h:outputText value="No Hotels Found"
      rendered="#{hotels != null and hotels.rowCount==0}"/>
    <h:dataTable value="#{hotels}" var="hot" rendered="#{hotels.rowCount>0}">
      <h:column>
        <f:facet name="header">Name</f:facet>
        #{hot.name}
      </h:column>
      <h:column>
        <f:facet name="header">Address</f:facet>
        #{hot.address}
      </h:column>
      <h:column>
        <f:facet name="header">City, State</f:facet>
        #{hot.city}, #{hot.state}, #{hot.country}
      </h:column>
      <h:column>
        <f:facet name="header">Zip</f:facet>
        #{hot.zip}
      </h:column>
      <h:column>
        <f:facet name="header">Action</f:facet>
        <s:link value="View Hotel" action="#{hotelBooking.selectHotel(hot)}"/>
      </h:column>
    </h:dataTable>
    <s:link value="More results" action="#{hotelSearch.nextPage}"
      rendered="#{hotelSearch.nextPageAvailable}"/>
  </div>
</a:outputPanel>

```

- ❶ Ajax4JSF的 `<a:support>` 标签允许一个JSF动作事件监听器在类似 `onkeyup` 这样的JavaScript事件发生时被异步的 `XMLHttpRequest` 调用。更棒的是，`reRender` 属性让我们可以在收到异步响应时渲染

染一个JSF页面的片段并执行一个页面的局部修改。

- ② Ajax4JSF的 <a:status> 标签使我们能在等待异步请求返回时显示一个简单的动画。
- ③ Ajax4JSF的 <a:outputPanel> 标签定义了一块能被异步请求修改的页面区域。
- ④ Seam的<s:link> 标签使我们能将一个JSF动作监听器附加在一个普通的（非JavaScript）HTML链接上。用它取代标准JSF的 <h:commandLink> 的好处就是它在“在新窗口中打开”和“在新标签页中打开”时仍然有效。值得注意的另一点就是我们用了个绑定了参数的方法：
#{hotelBooking.selectHotel(hot)}。在标准的统一EL中这是不允许的，但Seam对EL的扩展进行了扩展，使表达式能够支持带参数的方法。

这个页面根据我们的键入动态地显示搜索结果，让我们选择一家宾馆并将它传给 HotelBookingAction 的 selectHotel() 方法，这个对象才是真正有趣的地方。

现在让我们来看看宾馆预定范例程序是如何使用一个对话域的有状态的Session Bean的，这个Session Bean实现了业务会话相关持久化数据的天然缓存。下面的代码很长。但如果你把它理解为实现业务会话的多个步骤的一系列动作的话，它是不难理解的。我们把这个类当作故事一样从头开始阅读。

例 1.24.

```
@Stateful
@Name("hotelBooking")
@Restrict("#{identity.loggedIn}")
public class HotelBookingAction implements HotelBooking
{

    @PersistenceContext(type=EXTENDED)                                ❶
    private EntityManager em;

    @In                                                                ❷
    private User user;

    @In(required=false) @Out
    private Hotel hotel;

    @In(required=false)
    @Out(required=false)
    private Booking booking;

    @In
    private FacesMessages facesMessages;

    @In
    private Events events;

    @Logger
    private Log log;

    @Begin                                                            ❸
    public String selectHotel(Hotel selectedHotel)
    {
        hotel = em.merge(selectedHotel);
        return "hotel";
    }

    public String bookHotel()
    {
        booking = new Booking(hotel, user);
        Calendar calendar = Calendar.getInstance();
        booking.setCheckinDate( calendar.getTime() );
    }
}
```

```

        calendar.add(Calendar.DAY_OF_MONTH, 1);
        booking.setCheckoutDate( calendar.getTime() );

        return "book";
    }

    public String setBookingDetails()
    {
        if (booking==null || hotel==null) return "main";
        if ( !booking.getCheckinDate().before( booking.getCheckoutDate() ) )
        {
            facesMessages.add("Check out date must be later than check in date");
            return null;
        }
        else
        {
            return "confirm";
        }
    }

    @End
    public String confirm()
    {
        if (booking==null || hotel==null) return "main";
        em.persist(booking);
        facesMessages.add("Thank you, #{user.name}, your confirmation number for #{hotel.name} is #{booking.id}");
        log.info("New booking: #{booking.id} for #{user.username}");
        events.raiseEvent("bookingConfirmed");
        return "confirmed";
    }

    @End
    public String cancel()
    {
        return "main";
    }

    @Destroy @Remove
    public void destroy() {}
}

```

- ❶ 这个bean使用EJB3的 扩展持久化上下文，所以任意实体实例在整个有状态Session Bean的生命周期中一直受到管理。
- ❷ @Out 注解声明了一个属性值在方法调用后会 向外注入 到一个上下文变量中的。在这个例子中，名为 hotel 的上下文变量会在每个动作监听器调用完成后被设置为 hotel 实例变量的值。
- ❸ @Begin 注解表明被注解的方法开始一个 长期业务对话，因此当前业务对话上下文在请求结束后不会被销毁。相反，它会被关联给当前窗口的每次请求，在业务对话超时或者一个 @End 方法被调用后销毁。
- ❹ @End 注解表明被注解的方法被用来结束一个长期业务对话，所以当前业务对话上下文会在请求结束后被销毁。
- ❺ 这个EJB删除方法会在Seam销毁业务对话上下文时被调用。不要忘记定义该方法！

HotelBookingAction 包含了实现选择、预订和预订确认的所有动作监听器方法，并在它的实例变量中保存与之相关的状态。我们认为你一定会同意这个代码比起获取和设置 HttpSession 的属性来说要简洁的多。

而且，一个用户能在每个登录Session中拥有多个独立的业务对话。试试吧！登录系统，执行搜索，在多个浏览器标签页中导航到不同的宾馆页面。你能在同一时间建立两个不同的宾馆预约。如果某

个业务对话被闲置太长时间，Seam最终会判其超时并销毁它的状态。如果在结束业务对话后，你按了退回按钮回到那个会话的某一页，尝试执行一个动作，Seam会检测到那个业务对话已经被结束了，并将你重定向到搜索页面。

1.6.4. Seam的UI控制库

如果你查看下预订系统的WAR文件，你会在 WEB-INF/lib 目录中找到 seam-ui.jar。这个包里有许多Seam的JSF自定义控件。本应用程序在从搜索界面导航到宾馆页面时使用了 <s:link> 控件：

```
<s:link value="View Hotel" action="#{hotelBooking.selectHotel}"/>
```

这里的 <s:link> 允许我们在不中断浏览器的“在新窗口打开”功能的情况下给HTML链接附加上一个动作监听器。标准的JSF <h:commandLink> 无法在“在新窗口打开”的情况下正常工作。稍后我们会看到 <s:link> 还能提供很多其他有用的特性，包括业务会话传播规则。

宾馆预订系统里还用了些别的Seam和Ajax4JSF控件，特别是在 /book.xhtml 页面里。我们在这里不深入讨论这些控件，如果你想看懂这些代码，请参考介绍Seam的JSF表单验证功能的章节。

1.6.5. Seam调试页面

WAR文件还包括了 seam-debug.jar。如果把这个jar部属在 WEB-INF/lib 下，结合Facelets，你能在 web.xml 或者 seam.properties 里设置如下的Seam属性：

```
<context-param>
  <param-name>org.jboss.seam.core.init.debug</param-name>
  <param-value>true</param-value>
</context-param>
```

这样就能访问Seam调试页面了。这个页面可以让你浏览并检查任意与你当前登录Session相关的Seam上下文中的Seam组件。只需浏览 <http://localhost:8080/seam-booking/debug.seam> 即可。

JBoss Seam Debug Page

This page allows you to view and inspect any component in any Seam context associated with the current session.

Conversations

conversation id	activity	description	view id	
4	1:51:34 AM - 1:51:34 AM	Search hotels: M	/main.xhtml	Select conversation context
6	1:51:40 AM - 1:52:23 AM	Book hotel: Marriott Courtyard	/book.xhtml	Select conversation context

- Component (booking)

checkinDate	Fri Jan 20 20:52:20 EST 2006
checkoutDate	Sat Jan 21 20:52:20 EST 2006
class	class org.jboss.seam.example.booking.Booking
creditCard	
description	Marriott Courtyard, Jan 20, 2006 to Jan 21, 2006
hotel	Hotel(Tower Place, Buckhead,Atlanta,30305)
id	
user	User(gavin)

- Conversation Context (6)

booking
conversation
hotel
hotelBooking
hotels

- Business Process Context

Empty business process context

+ Session Context

+ Application Context

1.7. 一个使用Seam和jBPM的完整范例：DVD商店

DVD商店程序演示了如何在任务管理和页面流中使用jBPM。

用户界面应用jPDL页面流实现了搜索和购物车功能。

JBoss Seam DVD Store Demo

[Search for Movies](#)
[My Orders](#)

Search Results



Add to cart	Title	Actor	Price
<input type="checkbox"/>	Life is Beautiful	Roberto Benini	\$12.00
<input type="checkbox"/>	Finding Nemo	Albert Brooks	\$22.49
<input type="checkbox"/>	March of the Penguins	Morgan Freeman	\$16.98
<input type="checkbox"/>	Indiana Jones and the Temple of Doom	Harrison Ford	\$19.99
<input type="checkbox"/>	Clear and Present Danger	Harrison Ford	\$19.99
<input type="checkbox"/>	Roman Holiday	Audrey Hepburn	\$12.99
<input type="checkbox"/>	Breakfast at Tiffany's	Audrey Hepburn	\$12.99
<input type="checkbox"/>	Sabrina	Audrey Hepburn	\$12.99
<input type="checkbox"/>	Sabrina	Harrison Ford	\$19.99
<input type="checkbox"/>	Kill Bill Vol. 1	Uma Thurman	\$19.99
<input type="checkbox"/>	Kill Bill Vol. 2	Uma Thurman	\$19.99
<input type="checkbox"/>	Lost in Translation	Bill Murray	\$19.99
<input type="checkbox"/>	Broken Flowers	Bill Murray	\$19.99
<input type="checkbox"/>	Better Off Dead	John Cusak	\$8.99
<input type="checkbox"/>	Grosse Pointe Blank	John Cusak	\$11.99
<input type="checkbox"/>	High Fidelity	John Cusak	\$14.99
<input type="checkbox"/>	Somewhere in Time	Christopher Reeve	\$11.24
<input type="checkbox"/>	Superman - The Movie	Christopher Reeve	\$14.99
<input type="checkbox"/>	Superman II	Christopher Reeve	\$14.99
<input type="checkbox"/>	Superman III	Christopher Reeve	\$14.99

[Update Shopping Cart](#)

Welcome, Harry

Thank you for choosing the DVD Store

[Logout](#)

Search for DVDs:

Title:

Actor:

Category:

Any

Results Per Page:

20

[Search](#)

Shopping Cart

1 Napoleon Dynamite

Total:\$14.06

[Checkout](#)

Done

管理员界面使用jBPM来管理订单的审批和送货周期。业务流程可以通过选择不同的流程定义实现动态改变。

JBoss Seam DVD Store Demo

Manage Orders

Order Management

Pending orders are shown here on the order management screen for the store manager to process. Rather than being data-driven, order management is process-driven. A JBoss jBPM process assigns fulfillment tasks to the manager based on the version of the process loaded. The manager can change the version of the process at any time using the admin options box to the right.

- Order process 1 sends orders immediately to shipping, where the manager should ship the order and record the tracking number for the user to see.
- Order process 2 adds an approval step where the manager is first given the chance to approve the order before sending it to shipping. In each case, the status of the order is shown in the customer's order list.
- Order process 3 introduces a decision node. Only orders over \$100.00 need to be accepted. Smaller orders are automatically approved for shipping.

Task Assignment

Order Id	Order Amount	Customer	Task	
5	\$12.99	user1	ship	<input type="button" value="Assign"/>
7	\$77.70	user2	ship	<input type="button" value="Assign"/>

Order Acceptance

There are no orders to be accepted.

Shipping

Order Id	Order Amount	Customer	
6	\$94.95	user1	<input type="button" value="Ship"/>

Welcome, Albus

Thank you for choosing the DVD Store

Statistics

Inventory
 28 sold, 2473 in stock
Sales
 \$437.63 from 7 orders

Admin Options

Process Management

Done

TODO

见dvdstore目录。

1.8. 结合Seam和Hibernate的范例：Hibernate预订系统

Hibernate预订系统是之前客房预订系统的另一个版本，它使用Hibernate和JavaBeans代替了会话Bean实现持久化。

TODO

见hibernate目录。

1.9. 一个RESTful的Seam应用程序：Blog范例

Seam可以很方便地实现在服务器端保存状态的应用程序。然而，服务器端状态在有些情况下并不合适，特别是对那些用来提供内容的功能。针对这类问题，我们常需要让用户能够收藏页面，有一个

相对无状态的服务器，这样一来能够在任何时间通过书签来访问那些被收藏的页面。Blog范例演示了如何用Seam来实现一个RESTful的应用程序。应用程序中的每个页面都能被收藏，包括搜索结果页面。



Blog范例演示了“拉”风格（“pull”-style）的MVC，它不使用动作监听器方法来获取数据和为视图准备数据，而是视图在被显示时从组件中拉数据。

1.9.1. 使用“拉”风格的MVC

从 index.xhtml Facelets页面中取出的片断显示了blog的最近文章列表：

例 1.25.

```
<h:dataTable value="#{blog.recentBlogEntries}" var="blogEntry" rows="3">
  <h:column>
    <div class="blogEntry">
      <h3>#{blogEntry.title}</h3>
      <div>
        <h:outputText escape="false"
          value="#{blogEntry.excerpt==null ? blogEntry.body : blogEntry.excerpt}"/>
      </div>
      <p>
        <h:outputLink value="entry.seam" rendered="#{blogEntry.excerpt!=null}">
          <f:param name="blogEntryId" value="#{blogEntry.id}"/>
          Read more...
        </h:outputLink>
      </p>
    </div>
  </h:column>
</h:dataTable>
```

```

    <p>
      [Posted on
      <h:outputText value="#{blogEntry.date}">
        <f:convertDateTime timeZone="#{blog.timeZone}" locale="#{blog.locale}" type="both"/>
      </h:outputText>]
      &#160;
      <h:outputLink value="entry.seam">[Link]
        <f:param name="blogEntryId" value="#{blogEntry.id}" />
      </h:outputLink>
    </p>
  </div>
</h:column>
</h:dataTable>

```

如果我们通过收藏夹访问这个页面，那么 `<h:dataTable>` 的数据是怎么被初始化的呢？事实上，Blog 是延迟加载的，即在需要时才被名为 `blog` 的 Seam 组件“拉”出来。这与传统的基于动作的 web 框架（例如 Struts）的控制流程正好相反。

例 1.26.

```

@Name("blog")
@Scope(ScopeType.STATELESS)
public class BlogService
{
    @In
    private EntityManager entityManager;

    @Unwrap
    public Blog getBlog()
    {
        return (Blog) entityManager.createQuery("from Blog b left join fetch b.blogEntries")
            .setHint("org.hibernate.cacheable", true)
            .getSingleResult();
    }
}

```

- ❶ 这个组件使用了一个受 Seam 管理的持久化上下文（seam-managed persistence context）。与我们看过的其他例子不同，这个持久化上下文是由 Seam 管理的，而不是 EJB3 容器。持久化上下文贯穿于整个 Web 请求中，这使得在视图里访问未抓取的关联数据时可以避免发生任何异常。
- ❷ `@Unwrap` 注解告诉 Seam 将 `Blog` 而不是 `BlogService` 组件作为方法的返回值提供给客户端。这是 Seam 的管理员组件模式（manager component pattern）

这些看起来已经很不错了，那如何来收藏诸如搜索结果页这样的表单提交结果页面呢？

1.9.2. 可收藏的搜索结果页面

Blog 范例在每个页面的右上方都有一个很小的表单，这个表单允许用户搜索文章。这是定义在一个名为 `menu.xhtml` 的文件里的，它被 Facelets 模板 `template.xhtml` 所引用：

例 1.27.


```
<div id="search">
  <h:form>
    <h:inputText value="#{searchAction.searchPattern}" />
    <h:commandButton value="Search" action="/search.xhtml" />
  </h:form>
</div>
```

要实现一个可收藏的搜索结果页面，我们需要在处理搜索表单提交后执行一个浏览器重定向。因为我们用JSF视图id作为动作输出，所以Seam会在表单提交后自动重定向到该表单id。除此之外，我们也能像这样来定义一个导航规则：

例 1.28.

```
<navigation-rule>
  <navigation-case>
    <from-outcome>searchResults</from-outcome>
    <to-view-id>/search.xhtml</to-view-id>
    <redirect/>
  </navigation-case>
</navigation-rule>
```

然后表单看起来会是这个样子的：

例 1.29.

```
<div id="search">
  <h:form>
    <h:inputText value="#{searchAction.searchPattern}" />
    <h:commandButton value="Search" action="searchResults" />
  </h:form>
</div>
```

在重定向时，我们需要将表单的值作为请求参数包括进来，得到的书签URL会是这个样子：

<http://localhost:8080/seam-blog/search.seam?searchPattern=seam>。JSF没有为此提供一个简单的途径，但Seam却有。我们能在 `WEB-INF/pages.xml` 中定义一个 页面参数：

例 1.30.

```
<pages>
  <page view-id="/search.xhtml">
    <param name="searchPattern" value="#{searchService.searchPattern}" />
  </page>
  ...
</pages>
```

这告诉Seam在重定向时将 `#{searchService.searchPattern}` 的值作为名字是 `searchPattern` 的请求参数包括进去，并在显示页面前重新将这个值赋上。

重定向会把我们带到 `search.xhtml` 页面：

例 1.31.

```
<h:dataTable value="#{searchResults}" var="blogEntry">
  <h:column>
    <div>
      <h:outputLink value="entry.seam">
        <f:param name="blogEntryId" value="#{blogEntry.id}" />
        #{blogEntry.title}
      </h:outputLink>
      posted on
      <h:outputText value="#{blogEntry.date}">
        <f:convertDateTime timeZone="#{blog.timeZone}" locale="#{blog.locale}" type="both"/>
      </h:outputText>
    </div>
  </h:column>
</h:dataTable>
```

此处同样使用“拉”风格的MVC来获得实际搜索结果：

例 1.32.

```
@Name("searchService")
public class SearchService
{
    @In
    private EntityManager entityManager;

    private String searchPattern;

    @Factory("searchResults")
    public List<BlogEntry> getSearchResults()
    {
        if (searchPattern==null)
        {
            return null;
        }
        else
        {
            return entityManager.createQuery("select be from BlogEntry be where lower(be.title) like :searchPattern or lower(be.body) like :searchPattern")
                .setParameter("searchPattern", getSqlSearchPattern())
                .setMaxResults(100)
                .getResultList();
        }
    }

    private String getSqlSearchPattern()
    {
        return searchPattern==null ? "" : '%' + searchPattern.toLowerCase().replace('*', '%').replace('?', '_') + '%';
    }

    public String getSearchPattern()
    {
        return searchPattern;
    }

    public void setSearchPattern(String searchPattern)
    {
    }
```

```

        this.searchPattern = searchPattern;
    }
}

```

1.9.3. 在RESTful应用程序中使用“推”风格（“push”-style）的MVC

有些时候，用“推”风格的MVC来处理RESTful页面更有意义，为此Seam提供了 页面动作。 Blog 范例在文章页面 `entry.xhtml` 里使用了页面动作。请注意这里是故意这么做的，因为此处使用“拉”风格的MVC会更容易。

`entryAction` 组件工作起来非常像传统“推”风格MVC的面向动作框架例如Struts里的动作类（action class）：

例 1.33.

```

@Name("entryAction")
@Scope(STATELESS)
public class EntryAction
{
    @In(create=true)
    private Blog blog;

    @Out
    private BlogEntry blogEntry;

    public void loadBlogEntry(String id) throws EntryNotFoundException
    {
        blogEntry = blog.getBlogEntry(id);
        if (blogEntry==null) throw new EntryNotFoundException(id);
    }
}

```

在 `pages.xml` 里也定义了页面动作：

例 1.34.

```

<pages>
...

<page view-id="/entry.xhtml" action="#{entryAction.loadBlogEntry(blogEntry.id)}">
    <param name="blogEntryId" value="#{blogEntry.id}"/>
</page>

<page view-id="/post.xhtml" action="#{loginAction.challenge}"/>

<page view-id="*" action="#{blog.hitCount.hit}"/>
</pages>

```

范例中还将页面动作运用于一些其他的功能上 — 登录和页面访问计数器。另外一点值得注意的

是在页面动作绑定中使用了一个参数。这不是标准的JSF EL，是Seam为你提供的，你不仅能在页面动作中使用它，还可以将它使用在JSF方法绑定中。

当 entry.xhtml 页面被请求时，Seam先为模型绑定上页面参数 blogEntryId，然后运行页面动作，该动作获取所需的数据 — blogEntry — 并将它放在Seam事件上下文中。最后显示以下内容：

例 1.35.

```
<div class="blogEntry">
  <h3>#{blogEntry.title}</h3>
  <div>
    <h:outputText escape="false" value="#{blogEntry.body}" />
  </div>
  <p>
    [Posted on#{160;
    <h:outputText value="#{blogEntry.date}"
    <f:convertDateTime timeZone="#{blog.timeZone}" locale="#{blog.locale}" type="both"/>
    </h:outputText>]
  </p>
</div>
```

如果在数据库中没有找到blog entry，就会抛出 EntryNotFoundException 异常。我们想让该异常引起一个404错误，而非505，所以为这个异常类添加个注解：

例 1.36.

```
@ApplicationException(rollback=true)
@HttpError(errorCode=HttpServletResponse.SC_NOT_FOUND)
public class EntryNotFoundException extends Exception
{
    EntryNotFoundException(String id)
    {
        super("entry not found: " + id);
    }
}
```

该范例的另一个实现在方法绑定中没有使用参数：

例 1.37.

```
@Name("entryAction")
@Scope(STATELESS)
public class EntryAction
{
    @In(create=true)
    private Blog blog;

    @In @Out
    private BlogEntry blogEntry;

    public void loadBlogEntry() throws EntryNotFoundException
    {
        blogEntry = blog.getBlogEntry( blogEntry.getId() );
        if (blogEntry==null) throw new EntryNotFoundException(id);
    }
}
```

```
}  
  
}
```

```
<pages>  
  ...  
  
  <page view-id="/entry.xhtml" action="#{entryAction.loadBlogEntry}">  
    <param name="blogEntryId" value="#{blogEntry.id}" />  
  </page>  
  
  ...  
</pages>
```

你可以根据自己的喜好来选择实现。

第 2 章 用Seam-gen起步

Seam的发布包里已包含了命令行工具，用它可以很方便地搭建Eclipse项目，以及生成一些简单的Seam骨架代码，并能从已存在的数据库反向工程到应用程序。

它能让你感受到Seam给开发所带来的快捷，当你在电梯里看到那些令人厌烦的Ruby家伙在吹嘘他们的新玩艺儿是如何优美地在应用中 将繁琐的数据放进数据库时，你就可以取笑他们了。

在此版本中，seam-gen能很好地与JBoss AS一起工作。通过对项目配置进行些许的手工修改，seam-gen生成的项目就可与其它J2EE或Java EE 5应用服务器一起工作。

请注意，并不只限在Eclipse中使用seam-gen。但在本教程中，我们将为你演示如何在Eclipse中用它来完成调试与集成测试。若你不想安装Eclipse，你仍可跟随教程的步骤，因为所有的操作都是在命令行中完成的。

Seam-gen的Ant脚本与Hibernate工具包放一起，并同时提供了一些模板。这样我们就很容易地根据自己项目的需要来作些修改。

2.1. 准备活动

请确保已安装了JDK 5或者JDK6，JBoss AS 4.2和Ant 1.6，以及较新版的Eclipse、JBoss IDE和TestNG的Eclipse 插件。在Eclipse的JBoss Server View中将JBoss安装路径添加进去。然后以debug模式启动JBoss，并在弹出式命令窗口中进入Seam的目录。

JBoss很好地支持WAR和EAR的热重部署，但麻烦的是，由于在JVM中存在着多个Bug，在开发进程中多次的重部署EAR是常见的事，但这最终会耗尽PermGen 空间（Permanent Generation Space）。因此建议你在开发的过程中加大perm gen空间。若你是在JBoss IDE中运行JBoss，那你就可以在服务器运行配置中的VM arguments进行配置，建议依此修改：

```
-Xms512m -Xmx1024m -XX:PermSize=256m -XX:MaxPermSize=512
```

如果你没有那么多的可用内存，你只好用我们推荐的最小内存了：

```
-Xms256m -Xmx512m -XX:PermSize=128m -XX:MaxPermSize=256
```

若是在命令行模式中运行JBoss，那你就要在 bin/run.conf 文件中对JVM选项作修改了。

当然，我们可以先不理睬这些。当你在开发中第一次碰到 `OutOfMemoryException` 异常时再回过头来作此修改。

2.2. 建立一个新的Eclipse项目

首先，我们需要根据现有的开发环境对seam-gen进行配置：JBoss AS安装目录、Eclipse workspace及数据库连接。这些都很容易，只要敲入：

```
cd jboss-seam-2.0.x
seam setup
```

根据弹出的提示输入开发环境的相关信息：

```
C:\Projects\jboss-seam>seam setup
Buildfile: build.xml

setup:
[echo] Welcome to seam-gen :-)
[input] Enter your Java project workspace [C:/Projects]

[input] Enter your JBoss home directory [C:/Program Files/jboss-4.2.0.GA]

[input] Enter the project name [myproject]
helloworld
[input] Is this project deployed as an EAR (with EJB components) or a WAR (with no EJB support) [ear] (ear,war,)

[input] Enter the Java package name for your session beans [com.mydomain.helloworld]
org.jboss.helloworld
[input] Enter the Java package name for your entity beans [org.jboss.helloworld]

[input] Enter the Java package name for your test cases [org.jboss.helloworld.test]

[input] What kind of database are you using? [hsqldb] (hsqldb,mysql,oracle,postgres,mssql,db2,sybase,)
mysql
[input] Enter the Hibernate dialect for your database [org.hibernate.dialect.MySQLDialect]

[input] Enter the filesystem path to the JDBC driver jar [lib/hsqldb.jar]
../../mysql-connector.jar
[input] Enter JDBC driver class for your database [com.mysql.jdbc.Driver]

[input] Enter the JDBC URL for your database [jdbc:mysql:///test]

[input] Enter database username [sa]
gavin
[input] Enter database password []

[input] skipping input as property hibernate.default_schema.new has already been set.
[input] Enter the database catalog name (it is OK to leave this blank) []

[input] Are you working with tables that already exist in the database? [n] (y,n,)
y
[input] Do you want to drop and recreate the database tables and data in import.sql each time you deploy? [n] (y,n,)
n
[propertyfile] Creating new property file: C:\Projects\jboss-seam\seam-gen\build.properties
[echo] Installing JDBC driver jar to JBoss server
[echo] Type 'seam new-project' to create the new project

BUILD SUCCESSFUL
Total time: 1 minute 17 seconds
C:\Projects\jboss-seam>
```

该工具提供了相应的默认值，因此你可以直接按Enter键。

最重要的是你要对EAR部署还是WAR部署进行选择。EAR项目支持EJB 3.0 并需要Java EE 5。而WAR包不支持EJB 3.0，但可在J2EE环境中部署。另外WAR也更较简单，便于理解。假若你已安装了EJB3 profile，那你就用ear好了，否则，就只好用 war。在此假设我们选择了EAR部署，当然此教程也适用于WAR部署。

如果你手上有现成的数据模型，请确保你已输入现有数据库的表名。

这些设置保存在 seam-gen/build.properties 文件中，但你可通过运行 seam setup 来再次修改。

现在我们就可以在Eclipse workspace目录中创建一个新的项目，只需输入：

```
seam new-project
```

```
C:\Projects\jboss-seam>seam new-project
Buildfile: build.xml

validate-workspace:

validate-project:

copy-lib:
    [echo] Copying project jars ...
    [copy] Copying 58 files to C:\Projects\helloworld\lib
    [copy] Copying 9 files to C:\Projects\helloworld\embedded-ejb

file-copy-war:

file-copy-ear:
    [echo] Copying resources needed for EAR deployment to the C:\Projects\helloworld/resources directory...

new-project:
    [echo] A new Seam project named 'helloworld' was created in the C:\Projects directory
    [echo] Type 'seam explode' and go to http://localhost:8080/helloworld
    [echo] Eclipse Users: Add the project into Eclipse using File > New > Project and select General > Project (not Java Project)
    [echo] NetBeans Users: Open the project in NetBeans

BUILD SUCCESSFUL
Total time: 7 seconds
C:\Projects\jboss-seam>
```

这组操作复制了Seam jar文件及相应的jar文件与JDBC驱动jar到新建的Eclipse项目中，并生成了所需的源文件及其配置文件、一个模板文件和样式文件，及相应的Eclipse元数据及Ant构建脚本。只要你依此操作 New -> Project... -> General -> Project -> Next，输入Project name（此例为helloworld），并接着点击 Finish，就可将Eclipse项目自动部署到JBoss AS分解式的（exploded）目录结构中，请不要在新项目向导中选择 Java Project。

若Eclipse中的默认的JDK不是Java SE 5 或Java SE 6，你就得通过 Project -> Properties -> Java Compiler 来选择与Java SE 5 兼容的JDK。

另外，可在Eclipse之外输入 seam explode 来部署项目。

在 <http://localhost:8080/helloworld> 中查看此应用的首页。view/home.xhtml 是个使用view/layout/template.xhtml 模板生成的Facelets 页面，试着在Eclipse中编辑此页面或该模板，并在浏览器中刷新页面，立即看到结果。

别被在项目目录中的如此多的XML配置文件给吓晕了。那都是些标准的Java EE的东西，它们只需生成一次就不用再去理会了。在所有的Seam项目中，90%的配置内容都是一样的（这些可由seam-gen来帮我们完成）。

新生成的项目包含了三个数据库及持久化配置文件。jboss-beans.xml、persistence-test.xml 及 import-test.sql 文件是用在当TestNG对HSQLDB进行单元测试时。在 import-test.sql 中的数据库Schema及其测试数据总是在测试前就已转入数据库中。myproject-dev-ds.xml、persistence-dev.xml 及 import-dev.sql 文件是在部署应用到开发数据库时使用的。数据库schema是否可在部署时自动导出，取决于你是否在设置seam-gen环境时配置了已存在的数据库。myproject-prod-ds.xml、persistence-prod.xml 及 import-prod.sql 文件是在部署应用到生产数据库时使用的。在部署时数据库schema并不自动导出。

2.3. 创建新动作

若你熟知传统的action-style Web框架，你或许想知道在Java中如何来创建无状态action方法的简单Web页面。如果你输入：

```
seam new-action
```

则Seam将弹出一些信息并为你的项目生成新的Facelets页面及Seam组件。

```
C:\Projects\jboss-seam>seam new-action
Buildfile: build.xml

validate-workspace:

validate-project:

action-input:
    [input] Enter the Seam component name
ping
    [input] Enter the local interface name [Ping]

    [input] Enter the bean class name [PingBean]

    [input] Enter the action method name [ping]

    [input] Enter the page name [ping]

setup-filters:

new-action:
    [echo] Creating a new stateless session bean component with an action method
    [copy] Copying 1 file to C:\Projects\helloworld\src\action\org\jboss\helloworld
    [copy] Copying 1 file to C:\Projects\helloworld\src\action\org\jboss\helloworld
    [copy] Copying 1 file to C:\Projects\helloworld\src\action\org\jboss\helloworld\test
    [copy] Copying 1 file to C:\Projects\helloworld\src\action\org\jboss\helloworld\test
    [copy] Copying 1 file to C:\Projects\helloworld\view
    [echo] Type 'seam restart' and go to http://localhost:8080/helloworld/ping.seam

BUILD SUCCESSFUL
Total time: 13 seconds
C:\Projects\jboss-seam>
```

新增Seam组件后，我们需要重启分解式目录部署（exploded directory deployment）。输入seam restart，或在Eclipse中已生成项目的 build.xml 中运行 restart target就可完成。另一种方式是在Eclipse中通过编辑 resources/META-INF/application.xml 文件来强制重启。请注意，在每次修改应用程序时并不需要重启JBoss。

试着在浏览器中输入 <http://localhost:8080/helloworld/ping.seam> 地址并点击按钮，看看发生了什么。在项目的 src 目录中可看到完成此动作的源代码。试着在 ping() 方法中设置个断点，再次点击按钮，又发生了什么？

最后，在测试包中找到 PingTest.xml 文件，并用Eclipse的TestNG插件来运行测试。此外，还可用 seam test 或生成的build文件中的 test target来运行测试。

2.4. 创建有动作的表单（form）

下一步就是来创建表单了。请输入：

```
seam new-form
```

```
C:\Projects\jboss-seam>seam new-form
Buildfile: C:\Projects\jboss-seam\seam-gen\build.xml

validate-workspace:

validate-project:

action-input:
  [input] Enter the Seam component name
hello
  [input] Enter the local interface name [Hello]

  [input] Enter the bean class name [HelloBean]

  [input] Enter the action method name [hello]

  [input] Enter the page name [hello]

setup-filters:

new-form:
  [echo] Creating a new stateful session bean component with an action method
  [copy] Copying 1 file to C:\Projects\hello\src\com\hello
  [copy] Copying 1 file to C:\Projects\hello\src\com\hello
  [copy] Copying 1 file to C:\Projects\hello\src\com\hello\test
  [copy] Copying 1 file to C:\Projects\hello\view
  [copy] Copying 1 file to C:\Projects\hello\src\com\hello\test
  [echo] Type 'seam restart' and go to http://localhost:8080/hello/hello.seam

BUILD SUCCESSFUL
Total time: 5 seconds
C:\Projects\jboss-seam>
```

再次重启应用程序，并在浏览器中输入 `http://localhost:8080/helloworld/hello.seam` 就可看到结果了。接着看下所生成的代码，并运行测试。试着给表单加入一些字段及Seam组件（记着在每次更改Java代码时重新部署）。

2.5. 从现有数据库生成应用程序

在数据库中手工创建一些表。（如果你需要切换不同的数据库，只需再次运行 `seam setup` 即可。）现请输入：

```
seam generate-entities
```

接着重新部署，并在浏览器中输入 `http://localhost:8080/helloworld` 就可看到结果了。你可以试着浏览数据库，编辑现有的对象，并创建新的对象。如果你看下所生成的代码，你可能会对如此简单的代码感到惊讶。让开发人员，尤其是那些不甘于受Seam-gen摆布的开发人员，简单地手工编写数据访问代码，是Seam的设计目标之一。

2.6. 将应用部署为EAR

最后，我们想知道能否用标准的Java EE包来部署应用。首先，通过运行 `seam unexplode` 来移走分解式目录 (exploded directory)。在命令行中输入 `seam deploy` 或运行生成的Build脚本文件中的 `deploy target`就可完成EAR的部署，用 `seam undeploy` 命令或运行 `undeploy` 目标可卸下EAR。

默认情况下，应用程序会用 `dev profile` 来部署，EAR将包含 `persistence-dev.xml` 及 `import-dev.sql` 文件，`myproject-dev-ds.xml` 文件也会被部署。通过输入以下的命令你就可以更改profile，并可使用 `prod profile`：

```
seam -Dprofile=prod deploy
```

你甚至可以给你的应用程序定义新的部署profile，只需在项目中加入合适的文件，例如：`persistence-staging.xml`、`import-staging.sql` 及 `myproject-staging-ds.xml` — 并选择使用了 `-Dprofile=staging` 名字的profile。

2.7. Seam与增量热部署

将Seam应用部署成exploded目录的好处是，你能在开发时得到增量热部署的支持。你只需在 `components.xml` 中添加这一行来启用Seam和Facelet中的debug模式即可：

```
<core:init debug="true"/>
```

这样一来，重新部署以下文件时就不一定要完全地重启web应用了：

- 任意Facelet页面
- 任意 `pages.xml` 文件

若想对Java代码进行变更，就需要完全的应用重启。（在JBoss中，对于EAR部署，这需要用 `touch` 命令改变顶层的部署描述文件：对于EAR部署，则是 `application.xml`，而对于WAR部署，则是 `web.xml`。）

但你真正想加快编辑/编译/测试的流程，Seam支持对JavaBean组件进行增量式重部署。为了用上此功能，你必须把JavaBean组件部署到 `WEB-INF/dev` 目录中，以便它们能被特殊的Seam类加载器加载，而不是WAR或EAR类加载器。

请注意以下的限制：

- 必须是JavaBean组件，而不能是EJB3 Beans（此限制正在解决中）
- 实体Bean不可热部署
- 通过 `components.xml` 部署的组件可能无法热部署
- 在 `WEB-INF/dev` 之外部署的任何类都无法访问可热部署的组件
- 须启用Seam的debug模式

如果你用Seam-gen创建WAR项目，增量热部署对于src/action目录下的类是直接可用的，但是对于EAR项目不行。

2.8. 在Jboss 4.0下使用Seam

Seam 2.0是针对JavaServer Faces 1.2开发的，所以我们推荐在JBoss 4.2下使用Seam，因为它包含了JSF 1.2参考实现。然而仍然有办法在Jboss 4.0下使用Seam。需要两个步骤：安装启用了EJB3的Jboss 4.0版本并且把MyFaces替换为JSF1.2参考实现。你完成这两个步骤后，Seam2.0就可以在JBoss 4.0下部署了。

2.8.1. 安装JBoss 4.0

JBoss 4.0没有针对Seam的默认配置。想要运行Seam，你必须用JEMS 1.2安装器并且选择EJB3 profile。如果没有EJB3支持Seam是不能正常运行的。JEMS安装器可以在这里下载：
<http://labs.jboss.com/jemsinstaller/downloads>。

2.8.2. 安装JSF 1.2 RI

JBoss 4.0的配置可以在 server/default/deploy/jbossweb-tomcat55.sar 找到。你需要从 jsf-libs 目录删除 myfaces-api.jar 和 myfaces-impl.jar文件。你还需要把 jsf-api.jar、jsf-impl.jar、el-api.jar 和 el-ri.jar 复制到那个目录下。你可以在Seam的lib文件夹下找到这些JAR文件。EL JAR文件可以从Seam 1.2发行版中获取。

你还需要编辑 conf/web.xml 文件，把 myfaces-impl.jar 替换为 jsf-impl.jar。

第 3 章 上下文相关的组件模型

Seam中的两个核心概念是 context（上下文）思想和 component（组件）思想。组件是具有状态的对象，通常是EJB，组件的实例会和上下文绑定，在此上下文中具有一个名字。Bijection（双向注入）可以将内部的组件名（实例变量名）别名为上下文相关的名字，允许Seam动态组装组件树，还可以重新组装。

让我们从了解Seam内置的上下文开始。

3.1. Seam上下文

Seam上下文是由框架创建和销毁的。应用程序不能通过显式的Java API调用来控制上下文划分。上下文通常是隐含的。然而，在某些情况下，上下文可以通过annotation（注解）划分。

基本的Seam上下文有：

- Stateless context
- Event（or request） context
- Page context
- Conversation context
- Session context
- Business process context
- Application context

你可能在servlet及相关规范中已经见过其中一些上下文了，但其中有两个可能从未见过：conversation context（业务对话上下文），和 business process context（业务流程上下文）。在Web应用程序中，状态管理如此凌乱和容易出错的原因就是，内置的三个上下文（request, session和application）从业务逻辑的角度来看不是很有意义。例如，用户登录session的构建，对应用实际的工作流程来说就是相当随意的。因此，大部分的Seam组件被限定在业务会话或者业务流程上下文中，因为这些上下文从应用的角度来说最有意义。

让我们按顺序来考察每个context（上下文）。

3.1.1. Stateless context（无状态上下文）

那些确实没有状态的组件（主要是无状态Session Bean）总是运行在无状态上下文中（实际上就是上下文无关）。无状态组件没什么太大的意思，也有争议认为它们不十分面向对象。但不管怎么样，它们还是很重要，并且通常很有用。

3.1.2. Event context（事件上下文）

事件上下文是“最窄”的有状态上下文，是Web Request 上下文的泛化，用以包含其他种类的事

件。然而，与JSF请求的生命周期相关联的事件上下文是事件上下文最重要的实例，并且也是你最常打交道的。与事件上下文相关联的组件在请求结束时被销毁，但是它们的状态至少在请求的生命周期中是存在并且是定义良好的。

当你通过RMI或者Seam Remoting调用Seam组件的时候，一个事件上下文将为这个调用而被创建和销毁。

3.1.3. Page context（页面上下文）

页面上下文允许你将状态与一个渲染页面的实例相关联。你可以在Event Listener中初始化状态，或者在实际渲染页面的时候初始化状态，任何源于该页面的事件都可以访问到这些状态。这在支持像可点击列表这种的功能时特别有用，列表的内容通过服务器端的数据变化产生。实际上状态被序列化到了客户端，因此在多窗口操作或者回退按钮的时候，这种结构是非常健壮的。

3.1.4. Conversation context（业务会话上下文）

业务会话上下文是Seam中最核心的概念。conversation（业务会话）是从用户的视角看待的一个工作单元。它可能跨越与用户交互的多个Servlet、多个请求，和多个数据库事务。但是对用户来说，一个业务会话解决一个单一的问题。例如说：“预订酒店”，“批准合同”，“创建订单”都是业务会话。你可以将业务会话理解成对一个“use case（用例）”或“user story（用户故事）”的实现，当然特定的业务关联并非与此类例子完全一致。

业务会话保存关于“在此窗口中，用户正在干什么”的状态。在任何时间，一个用户可能同时位于多个业务会话活动中，一般是在几个不同窗口中。业务会话上下文让我们可以确保不同业务会话的状态不会互相干扰，不会导致Bug。

你可能要花上一点时间才能习惯以这一业务会话的观点来思考你的应用程序。但一旦你习惯于它，你会喜欢上这个术语，并且再不会不用业务会话来思考了！

一些业务会话仅存在在一次请求中。跨域多个请求的业务会话必须通过Seam提供的annotation注解来划分。

一些业务会话同时也是tasks（任务）。任务是一种业务会话，它特指一个长时间运行的业务流程，当正确完成后，可能会触发一个业务流程状态的转换。Seam为任务划分提供了专门的注解。

业务对话可以是nested（嵌套）的，一个业务对话嵌套“在”一个更大的业务对话中。这是一项高级特性。

通常，业务对话状态实际上由Seam保存在Servlet Session 中，跨越请求。Seam实现了可配置的conversation timeout，可以自动销毁不活动的业务会话，这就可以确保，如果用户取消对话，用户的登录Session中保存的状态不会无限增长。

对于在一个长时间运行的业务会话中所产生的并发请求，Seam按顺序执行。

除此之外，Seam也可以配置成把对话状态保存在客户端浏览器中。

3.1.5. Session context（Session上下文）

Session上下文保存与用户登录session相关联的状态。虽然当需要在多个业务会话中交换状态的时候这很有用，但我们一般不建议使用Session 上下文保存组件，除非是保存有关登录用户的全局信

息。

在JSR-168 Portal环境下，Session上下文代表Portlet上下文。

3.1.6. Business process context （业务流程上下文）

业务流程上下文保存了长时间运行的业务流程相关的状态。这种状态由BPM引擎（jBPM）管理和持久化。业务流程跨越多个用户的交互，因此状态在多个用户之间通过良好定义的方式共享。当前的任务决定当前的业务流程实例，业务流程的生命周期通过外置的 process definition language（流程定义语言）来定义，因此没有特别的annotation注解用于划分业务流程。

3.1.7. Application context（应用上下文）

Application上下文就是Servlet规范中的Servlet上下文。应用程序上下文在保存静态信息方面有用，例如配置数据，引用数据或者元模型。例如，Seam把自己的配置和元模型保存在应用程序上下文中。

3.1.8. Context variables（上下文变量）

上下文定义了命名空间，一组 context variables（上下文变量）。这些工作很类似Servlet规范中对Session或Request attributes的定义。你可以绑定任何你喜欢的值到Context Variable，但通常会绑定Seam组件实例到Context Variables。

因此，在上下文中，组件实例是通过上下文变量名字来辨别的（通常是这样，但并非绝对，就和组件名称一样）。你可以通过程序在特定范围内访问被命名的组件实例，这是通过 Contexts 类进行的，它提供了对 Context 接口的几个线程绑定的实例的访问：

```
User user = (User) Contexts.getSessionContext().get("user");
```

你也可以通过名字来设置或修改变量值：

```
Contexts.getSessionContext().set("user", user);
```

但通常，我们通过注射（injection）来从上下文中获得组件，并且通过反向注射（outjection）把组件实例返回上下文。

3.1.9. Context搜索优先级

有时候如上面的例子所示，组件实例是从某个特定的已知范围内获取的。其他的时候则是通过 priority order（优先级顺序）在所有有状态范围内搜寻。这个顺序是这样的：

- Event context
- Page context
- Conversation context
- Session context

- Business process context
- Application context

你可以通过调用 `Contexts.lookupInStatefulContexts()` 来执行带优先级的搜索。你在JSF页面中通过名字访问组件的时候，执行的就是这种带优先级的搜索。

3.1.10. 并发模型

Servlet和EJB规范都没有定义任何关于如何管理来自同一个客户端的并发请求的条款。Servlet容器简单地让所有的线程并发运行，把线程资源安全共享的任务交给应用程序代码。EJB容器允许无状态组件并发访问，但如果并发访问一个有状态Session Bean, 就会抛出一个异常。

旧式的Web应用程序是围绕细粒度的同步请求编写的，因此这种行为可能还OK。但对现代的程序而言，由于大量使用了很多细粒度的异步（AJAX）请求，并发是实际存在的，并且必须被程序模型支持。Seam在其上下文模型中加入了并发管理层。

Seam Session 和应用上下文是多线程的。Seam允许在一个上下文中并发请求，并发处理。事件（Event）和页面（Page）上下文自然是单线程的。业务流程（business process）上下文严格而言是多线程的，但实际情况中并发很少见，因此大多数情况不会出现并发。最后，Seam为Conversation Context提供了 每对话每进程单线程 模型，这是通过把同一个长时间运行的对话上下文中的并发请求序列化实现的。

因为Session上下文是多线程的，并且经常包含不稳定的状态，所以Session范围内的组件总是被Seam保护以防止并发操作。Seam默认把针对Session范围的Session Bean和JavaBean的请求序列化（并且检测、解决任何发生的死锁）。对Application Scoped的组件来说，这却不是默认行为，因为Application Scoped的组件通常不会包含的不稳定状态，并且在全局级别进行同步代价 极其 高昂。但是，你可以强制对任何Session Bean或JavaBean组件采用序列化的线程模型，要做的就是加上 `@Synchronized` 注解。

并发模型意味着AJAX客户端可以安全的使用不稳定的Session和会话状态，并且不需要开发者做任何特别的工作。

3.2. Seam 组件

Seam组件是POJO（Plain Old Java Objects）。特别地，他们是JavaBean或者EJB 3.0 enterprise bean。Seam并不强求组件是EJB，甚至可以不使用EJB 3.0兼容的容器，Seam在设计的时候处处考虑对EJB 3.0的支持，并且包含对EJB 3.0的深度整合。

- EJB 3.0 stateless Session Beans
- EJB 3.0 stateful Session Beans
- EJB 3.0 entity beans
- JavaBeans
- EJB 3.0 message-driven beans

3.2.1. 无状态Session Bean

无状态Session Bean组件无法在多次调用之间保持状态。因此，它们通常在不同的Seam上下文中，操作其他组件的状态。他们可以作为JSF的action listener，但是不能为JSF组件的显示提供属性。

因为每次请求都产生一个新的实例，无状态session bean可以并发访问。把其实例和请求相关联是EJB3容器的责任（通常这些实例会从一个可重用的池中分配，所以你可能会发现某些实例变量还保存着上次使用的痕迹。）

无状态Session Bean总是生活在无状态上下文中。

无状态Session Bean是Seam组件中最没趣的了。

Seam无状态Session Bean组件可以使用 `Component.getInstance()` 或者 `@In(create=true)` 实例化。它们不能直接使用JNDI或者 `new` 操作实例化。

3.2.2. 有状态Session Bean

有状态Session Bean不仅可以在bean的多次调用之间保持状态，而且在多次请求之间也可以保持状态。不由数据库保存的状态通常应该由有状态Session Bean保持。这是Seam和其他web框架之间的一个显著的不同点。其他框架把当前会话的信息直接保存在 `HttpSession` 中，而在Seam中你应该把它们保存在有状态Session Bean的实例中，该实例被绑定到会话上下文。这可以让Seam来替你管理状态的生命周期，并且保证在多个不同的并发会话中没有状态冲突。

有状态Session Bean经常被作为JSF action listener使用，也可以作为JSF显示或者form提交的backing bean(支持bean, 或称后台bean)，提供属性供组件访问。

默认情况下，有状态Session Bean会被绑定到Conversation Context。它们绝不会绑定到page或stateless context。

对Session范围的有状态Session Bean的并发请求，会被Seam按顺序串行处理。

Seam有状态Session Bean组件可以使用 `Component.getInstance()` 或者 `@In(create=true)` 实例化。它们不能直接使用JNDI或者 `new` 操作实例化。

3.2.3. 实体Bean

实体Bean可以被绑定到上下文变量，起到Seam组件的作用。因为Entity除了上下文标识之外，还有持久标识，Entity实体通常明确的由Java Code绑定，而非由Seam隐性初始化。

Entity Bean实体不支持双向注入或者上下文划分。对Entity Bean的调用也不会触发验证。

Entity Bean通常不作为JSF的action listener使用，但经常作为JSF组件用于显示或者form提交的后台bean, 提供属性功能。特别是，当Entity作为后台Bean的时候，它会和一个无状态Session Bean扮演的action listener联用，来实现CRUD之类的功能。

默认情况下，Entity Bean被绑定到Conversation Context。他们永远不能被绑定到无状态Context。

注意，在集群环境中，把Entity Bean直接绑定到Conversation或者Session范围的Seam上下文变量，与在有状态Session Bean中保持一个对Entity Bean的引用相比，性能比较差。因此，并非所有的

Seam应用程序都会把Entity Bean定义为Seam组件。

Seam实体Bean组件可以使用 `Component.getInstance()`、`@In(create=true)` 或者直接使用 `new` 操作来实例化。

3.2.4. JavaBeans

JavaBeans可以像无状态或者有状态Session Bean那样使用。但是，它们不能提供Session Bean那么多的功能（声明式事务划分、声明式安全性、高效的集群状态复制、EJB 3.0持久化、超时方法等等）。

在后面有一章，我们会展示如何在没有EJB容器的情况下使用Seam和Hibernate。此时，组件是JavaBeans，而非Session Beans。但是注意，在很多应用服务器中，对Conversation或Session 范围的Seam JavaBean组件集群操作，要比对有状态Session Bean组件集群慢。

默认，JavaBeans是绑定到Event Context的。

对Session范围的JavaBeans的并发请求总是会被Seam转化为串行执行。

Seam JavaBean组件可以使用 `Component.getInstance()`、`@In(create=true)` 或者直接使用 `new` 操作来实例化。

3.2.5. 消息驱动Bean

消息驱动Bean通常作为Seam组件。但是，消息驱动Bean与其他Seam组件的调用方式非常不同——它们并非通过Context变量调用，它们会监听发送到JMS Queue或者Topic的消息。

消息驱动Bean不能被绑定到Seam上下文。它们也不能访问它们的“调用者”的Session或者会话状态。但是，它们支持双向注入和一些其他的Seam功能。

消息驱动Bean不会被应用实例化，它是在接受到一条消息时由EJB容器来完成实例化的。

3.2.6. 拦截

为了表演Seam的魔术（双向注入，上下文划分，校验等），它必须对组件调用进行拦截。对JavaBean而言，Seam可以完全控制组件的初始化，不需要特别的配置。对于Entity Bean，也不需要拦截器，因为双向注入和上下文划分不起作用。对Session Bean，我们必须为它注册EJB拦截器。我们可以使用注解，比如：

```
@Stateless
@Interceptors(SeamInterceptor.class)
public class LoginAction implements Login {
    ...
}
```

但是更好的办法是在 `ejb-jar.xml` 中定义拦截器。

```
<interceptors>
  <interceptor>
    <interceptor-class>org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
  </interceptor>
</interceptors>
```

```
<assembly-descriptor>
  <interceptor-binding>
    <ejb-name>*/</ejb-name>
    <interceptor-class>org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
  </interceptor-binding>
</assembly-descriptor>
```

3.2.7. 组件名字

所有Seam组件都需要名字。我们可以通过 `@Name` 注解来命名组件：

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
    ...
}
```

这个名字是 seam component name，和EJB规范定义的任何其他名字都没有关系。但是，Seam组件名字就相当于JSF管理的Bean Name的名字，因此，可以理解为这两个概念是等同的。

`@Name` 不是定义组件名称的唯一方式，但是我们总得要在 某个地方 来指定名字。否则，Seam 所有的注解部分就无法工作。

就如同在JSF中，Seam组件实例绑定成上下文变量时，其名字通常和组件名相同。因此，例如我们可以通过 `Contexts.getStatelessContext().get("loginAction")` 来访问 `LoginAction`。特别是，不管Seam自己何时初始化一个组件，它将这个新实例以组件的名字绑定成一个变量。但是，又和JSF一样，应用程序也可以把组件绑定成其他的上下文变量，只需通过API编程调用。例如，当前登录的用户（User）可以被绑定成为Session上下文中的 `currentUser` 变量，而同时，另一个用作某种管理功能的用户则被绑定成对话上下文的 `user` 变量。

对非常大型的应用程序，经常使用全限定名；内置的Seam组件就是这样。

```
@Name("com.jboss.myapp.loginAction")
@Stateless
@Interceptors(SeamInterceptor.class)
public class LoginAction implements Login {
    ...
}
```

我们可以在Java代码和JSF表达式语言中使用全限定的组件名称。

```
<h:commandButton type="submit" value="Login"
  action="#{com.jboss.myapp.loginAction.login}"/>
```

这很啰嗦，Seam也提供了把全限定名简写的办法。在 `components.xml` 文件中加入类似这样的一行：

```
<factory name="loginAction" scope="STATELESS" value="#{com.jboss.myapp.loginAction}"/>
```

所有的Seam内置组件都有全限定名，但大多数都在Seam jar文件的 `components.xml` 中简写为简单的名字。

3.2.8. 定义组件范围 (Defining the Component Scope)

我们可以使用 `@Scope` 注解来覆盖默认的组件范围（上下文）。这可以让我们定义组件实例被Seam初始化后绑定到的具体上下文。

```
@Name("user")
@Entity
@Scope(SESSION)
public class User {
    ...
}
```

`org.jboss.seam.ScopeType` 定义了可能范围的枚举。

3.2.9. 具有多个角色的组件 (Components with multiple roles)

有些Seam组件类可以在系统中具有多个角色。例如，我们经常有一个 `User` 类用作Session-Scoped组件，代表当前用户，同时它又在用户管理界面中被用作Conversation-Scoped组件。`@Role` 注解让我们可以定义组件在另一个范围中的额外角色名 —— 这可以让我们把相同的组件类绑定成不同的上下文变量。（任何Seam组件实例都可以被绑定到多个上下文变量，但`@Role`使得我们也可以在类的级别做到这一点，从而享受自动实例化的优点。）

```
@Name("user")
@Entity
@Scope(CONVERSATION)
@Role(name="currentUser", scope=SESSION)
public class User {
    ...
}
```

`@Roles` 注解可以让我们为组件指定任意多的附加角色。

```
@Name("user")
@Entity
@Scope(CONVERSATION)
@Roles({@Role(name="currentUser", scope=SESSION),
        @Role(name="tempUser", scope=EVENT)})
public class User {
    ...
}
```

3.2.10. 内置组件

和很多优秀的框架一样，Seam自产自用，主要实现了一系列的内置Seam拦截器（后文详述）和Seam组件。这让应用程序在运行时和内置的组件交互变得很容易，甚至可以用自己编写的实现来替换掉内置组件，由此来定制Seam的基本功能。内置组件在 `org.jboss.seam.core` 这个Seam命名空间中定义，Java包名也是相同的。

像所有Seam组件一样，内置组件也可以被注射，但是它们也提供了便利的`instance()`静态方法：

```
FacesMessages.instance().add("Welcome back, #{user.name}!");
```

3.3. 双向注入

Dependency injection（依赖注入）和 inversion of control（控制反转）现在对大多数Java开发者来说都是熟悉的概念了。依赖注入允许一个组件通过容器“注入”另一个组件到一个setter方法或者实例变量的方式，来获得被“注入”组件的引用(reference)。我们之前看过的所有依赖注入的实现，注入发生在组件创建的时候，在此后实例的整个生命周期中不再改变。对无状态组件，这么做是有道理的。从客户端的角度来看，特定种类的无状态组件的所有实例都是可以替换的。另一方面，Seam着重处理有状态组件。此时传统的依赖注入不再是非常有效了。Seam引入了 bijection（双向注入）这个名词，用来作为注入的广义概括。和injection（单向注入）对比，bijection是：

- contextual（上下文相关的） - 双向注入用来针对不同的上下文来组装有状态组件（在较大范围的上下文中的组件，可以引用较小范围上下文中的组件）
- bidirectional（双向的） - 被触发后，值从上下文变量中注射到组件属性中，也可以从组件属性outjected（反向注入）回上下文, 这样被调用的组件可以只通过改写自己的实例变量就同时操作了上下文变量的值
- dynamic（动态的） - 因为上下文变量的值随着时间不断改变，而且因为Seam组件是有状态的，双向注入在每次组件被调用的时候都发生。

基本上，通过设置实例变量是需要注入、反向注入、还是二者皆是，双向注入让你将上下文变量映射到组件实例变量。当然，我们使用注解来设置双向注入。

@In 注解指明应该注入值，可能是注入实例变量：

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
    @In User user;
    ...
}
```

或者注入setter方法：

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
    User user;

    @In
    public void setUser(User user) {
        this.user=user;
    }

    ...
}
```

默认情况下，针对被注入的属性或者实例变量名，Seam会对所有的上下文进行优先级搜索。如果你希望明确指定上下文变量名，可以这样写：@In("currentUser")。

如果没有组件实例绑定到具名的上下文变量，你可能希望Seam创建一个，你可以指定 @In(create=true)。如果值是可选的（可以为null），请指定 @In(required=false)。

对于某些组件，到处指定 `@In(create=true)` 是很繁琐的。你可以注解整个组件为 `@AutoCreate`，它就会在任何需要的时候自动创建，不需要明确的指定 `create=true`。

你还可以注入表达式的值：

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
    @In("#{user.username}") String username;
    ...
}
```

（在下一章，有更多的关于组件生命周期和注射的内容。）

`@Out` 注解指定了某个属性需要对外注入，可能是从实例变量：

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
    @Out User user;
    ...
}
```

或者从某个getter方法：

```
@Name("loginAction")
@Stateless
@Interceptors(SeamInterceptor.class)
public class LoginAction implements Login {
    User user;

    @Out
    public User getUser() {
        return user;
    }

    ...
}
```

属性可以既是被注入的，也可以对外注入：

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
    @In @Out User user;
    ...
}
```

或者：

```
@Name("loginAction")
@Stateless
public class LoginAction implements Login {
    User user;

    @In
    public void setUser(User user) {
        this.user=user;
    }
}
```

```
@Out
public User getUser() {
    return user;
}

...
}
```

3.4. Lifecycle methods（生命周期方法）

Session Bean和实体Bean Seam组件支持所有通用的EJB3.0生命周期回调（@PostConstruct, @PreDestroy, 等等）。但是Seam也同样支持JavaBean组件使用任意的这些回调。然而，一旦这些注解在J2EE环境中失效，Seam定义了两个附加组件完成生命周期回调，这等同于 @PostConstruct 和 @PreDestroy。

@Create 方法在Seam实例化一个组件后被调用。组件只可以定义一个 @Create 方法。

@Destroy 方法在Seam组件被绑定的上下文结束时被调用。组件只可以定义一个 @Destroy 方法。

另外，有状态Session Bean组件 必须 定义一个无参并注解为 @Remove 的方法。这个方法在上下文结束时被Seam调用。

最后，相关的注解还有 @Startup，它可以用在任何Application或者Session范围的组件上。@Startup 注解告诉Seam在上下文开始的时候立刻初始化组件，而不是在被客户访问的时候才创建。控制startup组件的初始化顺序通过指定 @Startup(depends={...}) 进行。

3.5. 条件装载（Conditional installation）

@Install 注解让你控制组件的条件装载，允许随着不同的部署情形而改变。例如：

- 希望在测试中mock out一些基础组件。
- 希望在一些特殊的部署情形下改变组件的实现。
- 希望只有满足依赖条件的时候才安装某些组件（对框架作者很有用）。

@Install 通过让你指定 precedence（优先级）和 dependencies（依赖）来运作。

组件的优先级是一个数字，当在classpath中存在多个同组件名的类的时候，seam依靠它来决定安装哪个组件。Seam会选取优先级数字最大的。有一些预定义的优先级值：（按升序排列）：

1. BUILT_IN — 优先级最低的组件, 是内置在Seam中的组件。
2. FRAMEWORK — 第三方框架定义的组件可能覆盖内置组件，但被应用程序组件所重载。
3. APPLICATION — 默认优先级。大部分应用程序组件适合这一级别。
4. DEPLOYMENT — 和部署相关的应用程序组件

5. MOCK — 为在测试中使用的mock objects所准备。

假设我们有一个组件，名为 `messageSender`，和一个JMS队列交互。

```
@Name("messageSender")
public class MessageSender {
    public void sendMessage() {
        //do something with JMS
    }
}
```

在我们的单元测试中，我们并没有JMS队列可用，因此我们需要stub这个方法。我们会创建一个mock 组件，在单元测试运行时放在classpath中，但绝不会在部署应用程序时出现。

```
@Name("messageSender")
@Install(precedence=MOCK)
public class MockMessageSender extends MessageSender {
    public void sendMessage() {
        //do nothing!
    }
}
```

当seam在classpath中发现多个组件时，`precedence` 帮助Seam决定使用哪个版本。

如果我们能精确控制使用 classpath中存在的类，是很美妙的。但是如果我在编写一个可重用的框架，具有很多依赖条件，我不希望用那么多的jar来肢解框架。我希望通过已经安装了哪些了组件，以及classpath中存在哪些组件，来决定安装组件。`@Install` 注解也控制这一功能。Seam内部使用这一机制来控制很多内部组件的条件安装。虽然你可能不会在你的程序中使用它。

3.6. 日志

面对下面的代码，谁都会被搞得七窍生烟：

```
private static final Log log = LogFactory.getLog(CreateOrderAction.class);

public Order createOrder(User user, Product product, int quantity) {
    if ( log.isDebugEnabled() ) {
        log.debug("Creating new order for user: " + user.username() +
            " product: " + product.name()
            + " quantity: " + quantity);
    }
    return new Order(user, product, quantity);
}
```

难以想象为何简单的log信息会被搞得如此之复杂。用于log的代码函数比用于实际业务逻辑的还要多！很惊讶，Java社区10年内都没有对此加以改变。

Seam提供了可以显著简化上述代码的logging API：

```
@Logger private Log log;

public Order createOrder(User user, Product product, int quantity) {
    log.debug("Creating new order for user: #0 product: #1 quantity: #2", user.username(), product.name(), quantity);
    return new Order(user, product, quantity);
}
```


是否把 `log` 变量声明为 `static` 并不重要，它都可以工作，除非是 `Entity Bean` 组件，需要把 `log` 声明为静态的。

注意你并不需要繁杂的 `if (log.isDebugEnabled())` 保卫语句，因为字符串相加操作是在 `debug()` 方法 内部 发生的。也请注意通常不需要显式指定 `log` 类型，因为 `Seam` 知道哪个组件正在注入 `Log`。

假设 `User` 和 `Product` 是当前上下文中可用的 `Seam` 组件，写起来更加简便：

```
@Logger private Log log;

public Order createOrder(User user, Product product, int quantity) {
    log.debug("Creating new order for user: #{user.username} product: #{product.name} quantity: #0", quantity);
    return new Order(user, product, quantity);
}
```

`Seam logging` 自动选择发送结果到 `log4j` 或者 `JDK logging`。如果 `log4j` 在 `classpath` 中，`Seam` 会使用它，否则，`Seam` 会使用 `JDK logging`。

3.7. Mutable接口和@ReadOnly

很多应用服务器的 `HttpSession` 集群实现都有问题，对绑定到 `Session` 的可变对象状态的改变只有在明确调用 `setAttribute()` 的时候才会被复制。这是 `Bug` 的一个源头，这些 `Bug` 难以在开发阶段有效找出，因为它们只会在应用服务器失效切换的时候才会被发现。而且，实际的复制信息包含了绑定到 `Session` 的所有序列化对象图，这是低效的。

当然，`EJB` 有状态 `Session Bean` 必须进行自动 `dirty checking`，并进行可变状态的复制，并且 `EJB` 容器也应该引入优化，例如属性级别的复制。但不幸的是，并非所有的 `Seam` 用户都有这么好的运气，他们的环境可能并不支持 `EJB 3.0`。因此，对于 `Session` 和 `Conversation` 范围内的 `JavaBean` 和 `Entity Bean` 组件，在 `Web` 容器的 `Session` 集群之上，`Seam` 提供了额外的集群安全的状态管理层。

对于 `Session` 或 `Conversation` 范围的 `JavaBean` 组件，每次组件被引用程序调用的时候，`Seam` 自动通过调用一次 `setAttribute()` 来触发复制。当然，对大部分是读操作的组件来说，这效率不高。你可以通过实现 `org.jboss.seam.core.Mutable` 接口来控制这一行为。或者扩展 `org.jboss.seam.core.AbstractMutable`，在组件内实现自己的 `dirty-checking` 逻辑。例如，

```
@Name("account")
public class Account extends AbstractMutable
{
    private BigDecimal balance;

    public void setBalance(BigDecimal balance)
    {
        setDirty(this.balance, balance);
        this.balance = balance;
    }

    public BigDecimal getBalance()
    {
        return balance;
    }

    ...
}
```

或者，你可以使用 `@ReadOnly` 注解来达到类似的效果：

```
@Name("account")
public class Account
{
    private BigDecimal balance;

    public void setBalance(BigDecimal balance)
    {
        this.balance = balance;
    }

    @ReadOnly
    public BigDecimal getBalance()
    {
        return balance;
    }

    ...
}
```

对Session或Conversation范围的Entity Bean组件，在每次被请求的时候Seam自动通过一次 `setAttribute()` 调用来触发复制，除非（对话范围的）实体和一个Seam管理的持久化上下文相关联，此时无需复制。这一策略不是最高效的，因此Session或Conversation范围的Entity Bean应该小心使用。你总是可以编写有状态的Session Bean或者JavaBean组件来“管理”Entity Bean实例。例如，

```
@Stateful
@Name("account")
public class AccountManager extends AbstractMutable
{
    private Account account; // an entity bean

    @Unwrap
    public void getAccount()
    {
        return account;
    }

    ...
}
```

注意, 对于 `EntityHome` 类，Seam应用框架提供了一个非常好的例子来说明如何使用Seam组件来管理实体Bean的实例。

3.8. Factory和Manager组件

我们经常需要与非Seam组件的对象打交道。但是我们仍然希望把它们通过 `@In` 注入我们的组件，并在值和方法表达式中使用它们。有时候，我们甚至需要把它们绑定到Seam 上下文的生命周期里（例如`@Destroy`）。所以Seam上下文可以容纳非Seam组件的对象，并且Seam提供了一些很好的特性，这些特性使得我们与绑定到上下文里的非组件对象打交道更加容易。

factory component pattern（工厂组件模式）让Seam组件作为非组件对象的构造器。当上下文变量被引用，但是没有值被绑定到它时，会调用一个factory method（工厂方法）。我们通过`@Factory` 注解来定义工厂方法。工厂方法把一个值绑定到上述上下文变量，并且决定被绑定的值的范围。有两

种工厂方法。第一种返回一个值，Seam会把它绑定到上下文里：

```
@Factory(scope=CONVERSATION)
public List<Customer> getCustomerList() {
    return ... ;
}
```

第二种方法返回 void，它自己把值绑定到上下文变量：

```
@DataModel List<Customer> customerList;

@Factory("customerList")
public void initCustomerList() {
    customerList = ... ;
}
```

两种情况下，当我们引用 `customerList` 上下文变量，而其值为null时，工厂方法被调用，然后对这个值生命周期的其他部分就无法操纵了。更加强大的模式是 `manager component pattern`（管理者组件模式）。在这种情况下，有一个Seam组件绑定到上下文变量，它管理着上下文变量的值，对客户端不可见。

管理者组件可以是任何组件，它需要一个 `@Unwrap` 方法。该方法返回对客户端可见的值，每次 上下文变量被引用的时候都会被调用。

```
@Name("customerList")
@Scope(CONVERSATION)
public class CustomerListManager
{
    ...

    @Unwrap
    public List<Customer> getCustomerList() {
        return ... ;
    }
}
```

当你有一个对象并需要对其组件的生命周期更多的控制时，管理组件模式就显得尤其有用。例如，如果你有一个重量级的对象，当上下文结束时你想对其进行清除操作，你可以`@Unwrap`对象，并在管理组件的 `@Destroy` 方法中执行清除操作。

```
@Name("hens")
@Scope(APPLICATION)
public class HenHouse {

    Set<Hen> hens;

    @In(required=false) Hen hen;

    @Unwrap
    public List<Hen> getHens() {
        if (hens == null) {
            // Setup our hens
        }
        return hens;
    }

    @Observer({"chickBorn", "chickenBoughtAtMarket"})
    public addHen() {
        hens.add(hen);
    }
}
```

```
@Observer("chickenSoldAtMarket")
public removeHen() {
    hens.remove(hen);
}

@Observer("foxGetsIn")
public removeAllHens() {
    hens.clear();
}
...
}
```

这里，被管理的组件观察那些改变在下面的对象的事件。组件自己管理这些动作，并且由于对象在每一次访问中都被解开，所以这里提供了一个统一的视图。

第 4 章 配置Seam组件

Seam所崇尚的哲学是XML配置最小化。不过，基于不同的原因，我们有时候还是要利用XML来配置Seam组件。这些原因包括：将Java代码与特定于部署的信息分离；要建立可重用的框架；配置Seam的内置功能等等。Seam提供了两种基本的配置组件方法：通过在properties文件或者 `web.xml` 中设置属性来配置，或者通过 `components.xml` 进行配置。

4.1. 通过属性设置来配置组件

Seam组件的配置属性可以通过两种方式得到：通过servlet context参数，或者通过位于classpath下的 `seam.properties` 属性文件进行。

可配置的Seam组件必须为可配置的属性暴露JavaBean风格的属性setter方法。例如，一个名为 `com.jboss.myapp.settings` 的Seam组件拥有一个名为 `setLocale()` 的setter方法，我们就可以在 `seam.properties` 文件中提供一个名为 `com.jboss.myapp.settings.locale` 的属性，或者作为一个servlet context参数，这样，一旦该组件被实例化，Seam将自动为这个名为 `locale` 的属性注入相应的值。

Seam本身的配置也采用了相同的机制。例如，要设置对话超时，我们可以在 `web.xml` 或者 `seam.properties` 中为 `org.jboss.seam.core.manager.conversationTimeout` 提供一个值。（在Seam内置的组件 `org.jboss.seam.core.manager` 中，已经包含了一个名为 `setConversationTimeout()` 的setter方法。）

4.2. 通过 `components.xml` 来配置组件

`components.xml` 文件的功能要比属性设置的更强大一些。它让你：

- 配置那些已经被自动安装的组件—包括内置组件以及那些带有 `@Name` 注解，且被Seam的部署扫描器识别到的那些应用组件。
- 将那些没有 `@Name` 注解的类安装成为Seam组件—这一点对于那些需要以不同的名字进行多次安装的结构组件特别有用（例如，Seam管理的持久化上下文）。
- 安装那些仅具有 `@Name` 注解，但是默认情况下未被安装的Seam组件。因为 `@Install` 注解表明该组件不应当被安装。
- 覆盖组件的范围。

`components.xml` 文件可以出现在下面三个不同地方中的任何一处：

- `war` 包的 `WEB-INF` 目录下。
- `jar` 包的 `META-INF` 目录下。
- 包含带有 `@Name` 注解类的 `jar` 包下的任何目录。

通常情况下，当Seam部署扫描器在包含 `seam.properties` 文件或者 `META-INF/components.xml` 文件的文件夹中识别到一个包含 `@Name` 注解的类时，Seam将安装该组件。（除非这个组件具有一个 `@Install` 注解，表示它不应该被默认安装。）`components.xml` 文件让我们去处理那些需要覆盖注解的特殊情况。

例如，下面的 components.xml 文件安装了 jBPM：

```
<components xmlns="http://jboss.com/products/seam/components"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xmlns:bpm="http://jboss.com/products/seam/bpm">
  <bpm:jbpm/>
</components>
```

这个例子实现了相同的功能：

```
<components>
  <component class="org.jboss.seam.bpm.Jbpm"/>
</components>
```

这个例子安装并配置了Seam管理的两个不同的持久化上下文：

```
<components xmlns="http://jboss.com/products/seam/components"
             xmlns:persistence="http://jboss.com/products/seam/persistence">

  <persistence:managed-persistence-context name="customerDatabase"
                                           persistence-unit-jndi-name="java:/customerEntityManagerFactory"/>

  <persistence:managed-persistence-context name="accountingDatabase"
                                           persistence-unit-jndi-name="java:/accountingEntityManagerFactory"/>

</components>
```

这个例子也一样：

```
<components>
  <component name="customerDatabase"
             class="org.jboss.seam.persistence.ManagedPersistenceContext">
    <property name="persistenceUnitJndiName">java:/customerEntityManagerFactory</property>
  </component>

  <component name="accountingDatabase"
             class="org.jboss.seam.persistence.ManagedPersistenceContext">
    <property name="persistenceUnitJndiName">java:/accountingEntityManagerFactory</property>
  </component>
</components>
```

这个例子创建了一个Seam管理的session范围持久化上下文（这在实际项目中并不推荐使用）

```
<components xmlns="http://jboss.com/products/seam/components"
             xmlns:persistence="http://jboss.com/products/seam/persistence">

  <persistence:managed-persistence-context name="productDatabase"
                                           scope="session"
                                           persistence-unit-jndi-name="java:/productEntityManagerFactory"/>

</components>
```

```
<components>

  <component name="productDatabase"
             scope="session"
             class="org.jboss.seam.persistence.ManagedPersistenceContext">
    <property name="persistenceUnitJndiName">java:/productEntityManagerFactory</property>
  </component>
```

```
</components>
```

通常会给像持久化上下文这样的基础结构对象使用 `auto-create` 选项，它能在你使用 `@In` 注解时，不必显式地指定 `create=true`。

```
<components xmlns="http://jboss.com/products/seam/components"
  xmlns:persistence="http://jboss.com/products/seam/persistence"

  <persistence:managed-persistence-context name="productDatabase"
    auto-create="true"
    persistence-unit-jndi-name="java:/productEntityManagerFactory"/>

</components>
```

```
<components>

  <component name="productDatabase"
    auto-create="true"
    class="org.jboss.seam.persistence.ManagedPersistenceContext">
    <property name="persistenceUnitJndiName">java:/productEntityManagerFactory</property>
  </component>

</components>
```

`<factory>` 声明让你指定一个值或者方法来绑定一个表达式，当它第一次被引用时，将被执行用来初始化一个context变量的值。

```
<components>

  <factory name="contact" method="#{contactManager.loadContact}" scope="CONVERSATION"/>

</components>
```

你也可以为Seam组件创建一个别名（第二个名字），就像这样：

```
<components>

  <factory name="user" value="#{actor}" scope="STATELESS"/>

</components>
```

你甚至可以给常用的表达式定义别名：

```
<components>

  <factory name="contact" value="#{contactManager.contact}" scope="STATELESS"/>

</components>
```

`auto-create="true"` 用在 `<factory>` 声明中尤其常见。

```
<components>

  <factory name="session" value="#{entityManager.delegate}" scope="STATELESS" auto-create="true"/>

</components>
```

我们在部署或者测试期间，有时候想要通过略微的改动，来重用同一个 `components.xml` 文件。Seam 允许你在 `components.xml` 文件中使用 `@wildcard@` 形式的通配符，这些通配符可以在部署的时候被Ant构建脚本替换，也可以在开发时通过在classpath中提供一个名为 `components.properties` 的文件进行替换。你会在Seam的示例程序中找到这个用法。

4.3. 细粒度的配置文件

如果你有大量的组件需要在XML中进行配置，那么就很有必要将 `components.xml` 文件中的内容分散到多个文件中去。Seam允许你把类（例如名为 `com.helloworld.Hello`）的配置放到一个资源中（名为 `com/helloworld/Hello.component.xml`）。（你对这种模式可能很熟悉，因为它与我们在Hibernate中使用的相同）。文件的根元素应该是 `<components>` 或者 `<component>`。

第一个选项允许你在一个文件中定义多个组件：

```
<components>
  <component class="com.helloworld.Hello" name="hello">
    <property name="name">#{user.name}</property>
  </component>
  <factory name="message" value="#{hello.message}"/>
</components>
```

第二个选项只允许你定义或者配置单个组件，不过麻烦会少一点：

```
<component name="hello">
  <property name="name">#{user.name}</property>
</component>
```

在第二个选项中，类名与组件定义所在的文件是一致的。

你还可以选择将所有类的配置都放在 `com/helloworld/components.xml` 的 `com.helloworld` 包中。

4.4. 可配置的属性类型

String的属性、基本类型以及基本类型的包装类型可以像我们期望的那样进行配置：

```
org.jboss.seam.core.manager.conversationTimeout 60000
```

```
<core:manager conversation-timeout="60000"/>
```

```
<component name="org.jboss.seam.core.manager">
  <property name="conversationTimeout">60000</property>
</component>
```

也支持由String或者基本类型构成的数组、Set和List：

```
org.jboss.seam.bpm.jbpm.processDefinitions order.jpdl.xml, return.jpdl.xml, inventory.jpdl.xml
```

```
<bpm:jbpm>
  <bpm:process-definitions>
    <value>order.jpdl.xml</value>
```



```

    <value>return. jpd1. xml</value>
    <value>inventory. jpd1. xml</value>
  </bpm:process-definitions>
</bpm:jbpm>

```

```

<component name="org.jboss.seam.bpm.jbpm">
  <property name="processDefinitions">
    <value>order. jpd1. xml</value>
    <value>return. jpd1. xml</value>
    <value>inventory. jpd1. xml</value>
  </property>
</component>

```

甚至也支持那些包含String值为键、String或者基本类型值的Map:

```

<component name="issueEditor">
  <property name="issueStatuses">
    <key>open</key> <value>open issue</value>
    <key>resolved</key> <value>issue resolved by developer</value>
    <key>closed</key> <value>resolution accepted by user</value>
  </property>
</component>

```

最后，你可以利用值绑定表达式来将所有的组件装配起来。注意这与使用 `@In` 注解进行注入非常不同，因为它是在组件初始化而不是被调用时起作用的。因而它与传统的IoC容器例如JSF或者Spring所提供的依赖注入功能非常非常类似。

```

<drools:managed-working-memory name="policyPricingWorkingMemory" rule-base="#{policyPricingRules}"/>

```

```

<component name="policyPricingWorkingMemory"
  class="org.jboss.seam.drools.ManagedWorkingMemory">
  <property name="ruleBase">#{policyPricingRules}</property>
</component>

```

4.5. 使用XML命名空间

纵观整个示例，有两种完全不同的声明组件的方式：使用或者不使用XML命名空间。下面的示例展示了一个典型的 `components.xml` 文件，它没有使用命名空间，而是使用Seam Components DTD:

```

<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://jboss.com/products/seam/components"
  xsi:schemaLocation="http://jboss.com/products/seam/components http://jboss.com/products/seam/components-2.0.xsd">

  <component class="org.jboss.seam.core.init">
    <property name="debug">true</property>
    <property name="jndiPattern">@jndiPattern@</property>
  </component>

</components>

```

正如你所见，这样的配置有点繁琐。更糟的是，这些组件和属性的名称在开发时是无法被校验的

。

使用命名空间的配置看起来像这样：

```
<?xml version="1.0" encoding="UTF-8"?>
<components xmlns="http://jboss.com/products/seam/components"
             xmlns:core="http://jboss.com/products/seam/core"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation=
               "http://jboss.com/products/seam/core http://jboss.com/products/seam/core-2.0.xsd
               http://jboss.com/products/seam/components http://jboss.com/products/seam/components-2.0.xsd">

  <core:init debug="true" jndi-pattern="@jndiPattern@" />

</components>
```

虽然Schema的声明很繁琐，不过实际的XML内容是清晰而简单易懂的。Schema提供了关于每个可用组件和属性的详细信息，这使得XML编辑器可以发挥其自动完成的功效。所以，使用命名空间的元素使生成和维护正确的 `components.xml` 文件都变得更加简单。

现在，这种方式对于Seam内建的组件工作得很好，但是对于用户自定义的组件又如何呢？这里有两种选择：第一种，Seam支持两种模型的混合使用，允许使用普通的 `<component>` 声明来配置用户自定义的组件，同时也使用命名空间来配置内置组件。不过更好的方法是，Seam允许你快速地为你的组件声明命名空间。

任何Java包都可以通过用 `@Namespace` 注解该包，而与XML命名空间而关联起来。（包级别的注解是在一个名为 `package-info.java` 的文件中声明的，该文件处于包的同级目录下）。下面是一个来自 `seampay` 演示的例子：

```
@Namespace(value="http://jboss.com/products/seam/examples/seampay")
package org.jboss.seam.example.seampay;

import org.jboss.seam.annotations.Namespace;
```

这样，你就可以在 `components.xml` 中使用命名空间的方式了！现在，你可以这么写：

```
<components xmlns="http://jboss.com/products/seam/components"
             xmlns:pay="http://jboss.com/products/seam/examples/seampay"
             ... >

  <pay:payment-home new-instance="#{newPayment}"
                   created-message="Created a new payment to #{newPayment.payee}" />

  <pay:payment name="newPayment"
               payee="Somebody"
               account="#{selectedAccount}"
               payment-date="#{currentDatetime}"
               created-date="#{currentDatetime}" />

  ...
</components>
```

或者：

```
<components xmlns="http://jboss.com/products/seam/components"
             xmlns:pay="http://jboss.com/products/seam/examples/seampay"
             ... >

  <pay:payment-home>
    <pay:new-instance>#{newPayment}</pay:new-instance>
    <pay:created-message>Created a new payment to #{newPayment.payee}</pay:created-message>
```

```

</pay:payment-home>

<pay:payment name="newPayment">
  <pay:payee>Somebody</pay:payee>
  <pay:account>#{selectedAccount}</pay:account>
  <pay:payment-date>#{currentDatetime}</pay:payment-date>
  <pay:created-date>#{currentDatetime}</pay:created-date>
</pay:payment>
...
</components>

```

这些示例展示了命名空间元素的两种使用模式。在第一个声明中，`<pay:payment-home>` 指向 `paymentHome` 组件。

```

package org.jboss.seam.example.seampay;
...
@Name("paymentHome")
public class PaymentController
    extends EntityHome<Payment>
{
    ...
}

```

元素的名称是连字符（-）形式的组件名称。元素的属性是连字符（-）形式的属性名称。

在第二个声明中，`<pay:payment>` 元素指向 `org.jboss.seam.example.seampay` 包中的 `Payment` 类。在这个例子中，`Payment` 是一个被定义成Seam组件的实体。

```

package org.jboss.seam.example.seampay;
...
@Entity
public class Payment
    implements Serializable
{
    ...
}

```

如果我们需要用户自定义组件的验证和自动完成功能，我们就需要一个Schema。目前Seam还无法提供为一组组件自动生成Schema的机制，所以你必需手工生成。标准Seam包的Schema定义可以当作示范。

以下是Seam所使用的命名空间：

- `components` — <http://jboss.com/products/seam/components>
- `core` — <http://jboss.com/products/seam/core>
- `drools` — <http://jboss.com/products/seam/drools>
- `framework` — <http://jboss.com/products/seam/framework>
- `jms` — <http://jboss.com/products/seam/jms>
- `remoting` — <http://jboss.com/products/seam/remoting>
- `theme` — <http://jboss.com/products/seam/theme>

- security — <http://jboss.com/products/seam/security>
- mail — <http://jboss.com/products/seam/mail>
- web — <http://jboss.com/products/seam/web>
- pdf — <http://jboss.com/products/seam/pdf>
- spring — <http://jboss.com/products/seam/spring>

第 5 章 事件、拦截器和异常处理

两个更深入的概念补充了上下文组件模型，这两个概念推动了极端松耦合这一Seam应用程序的独特特性。第一个是强有力的事件模型，事件可以通过类似JSF绑定表达式的方法映射到事件监听器。第二个是普遍使用注解和拦截器，这使我们总能跨越式地切入到实现业务逻辑的组件。

5.1. Seam事件

Seam组件模型是为使用 事件驱动的应用程序 而开发的，特别是在一个细粒度的事件模型里进行细粒度的松耦合组件的开发。Seam的事件有几种类型，大部分是我们已经见过的：

- JSF事件
- jBPM的状态转移事件
- Seam页面动作
- Seam组件驱动事件
- Seam上下文事件

所有这些不同种类的事件都通过绑定了表达式的JSF EL方法映射到Seam组件去。JSF事件是在JSF模板中定义的：

```
<h:commandButton value="Click me!" action="#{helloWorld.sayHello}" />
```

对于jBPM的转换事件，是在jBPM过程定义或页面流定义中指定的：

```
<start-page name="hello" view-id="/hello.jsp">
  <transition to="hello">
    <action expression="#{helloWorld.sayHello}" />
  </transition>
</start-page>
```

你可以在其他地方找到更多关于JSF事件和jBPM事件的信息。我们现在主要关注由Seam定义的新增类型的事件上。

5.1.1. 页面动作

Seam的页面动作是指就在我们渲染页面之前发生的事件。我们在 WEB-INF/pages.xml 中声明页面动作。我们可以为任何一个特殊的JSF视图id定义一个页面动作：

```
<pages>
  <page view-id="/hello.jsp" action="#{helloWorld.sayHello}" />
</pages>
```

或者，我们可以使用一个通配符 * 作为 view-id 的后缀来指定一个动作，应用到所有符合该模式的视图ID中：

```
<pages>
```

```
<page view-id="/hello/*" action="#{helloWorld.sayHello}"/>
</pages>
```

如果多通配符的页面动作匹配当前的view-id，Seam将按照从最通用到最特殊的顺序来调用所有的动作。

页面动作方法可以返回一个JSF的结果。如果这个结果非空，Seam将用定义好的导航规则导航到一个视图中去。

此外，在元素 `<page>` 里提到的视图id不需要对应一个真实的JSP或Facelets页面！因此，我们可以再生传统的面向动作的框架的功能，就像Struts或WebWork使用页面动作那样。例如：

```
TODO: translate struts action into page action
```

如果你想要应non-faces的请求做点复杂的事情（例如HTTP GET请求），这就非常有用。

对于多页面或者条件页面的动作，可以使用 `<action>` 标签指定：

```
<pages>
  <page view-id="/hello.jsp">
    <action execute="#{helloWorld.sayHello}" if="#{not validation.failed}"/>
    <action execute="#{hitCount.increment}"/>
  </page>
</pages>
```

5.1.1.1. 页面参数

一个JSF faces请求（表单提交）同时封装了一个“动作action”（一个方法绑定）和“多个参数parameters”（输入值绑定）。一个页面动作也可能需要参数！

由于GET请求是可以做标记的，页面参数是作为人类易读的请求参数来传递的。（不像JSF form的输入，什么都有就是不具有可读性！）

你可以使用页面参数，带不带动作方法都可以。

5.1.1.1.1. 将请求参数映射到模型

Seam让我们提供一个值绑定，来将一个已命名的请求参数映射成一个模型对象的属性。

```
<pages>
  <page view-id="/hello.jsp" action="#{helloWorld.sayHello}">
    <param name="firstName" value="#{person.firstName}"/>
    <param name="lastName" value="#{person.lastName}"/>
  </page>
</pages>
```

`<param>` 的声明是双向的，就像一个JSF输入的值绑定：

- 当视图id的一个non-faces (GET) 请求发生时，Seam在执行了相应的类型转变之后，就在模型对象上设置已命名的请求参数的值。
- 任何 `<s:link>` 或 `<s:button>` 透明地或者说自动地包括request带有的参数。参数的值由渲染阶段（

当 `<s:link>` 被渲染）的绑定值来决定。

- 使用 `<redirect/>` 到视图id的任何导航规则很明显是含有请求参数。参数的值由调用应用程序阶段结束时的值绑定大小来决定。
- 这个值很明显是由带有视图id的被提交的任何JSF页面传播的。这意味着视图参数表现得就像faces请求的 `PAGE` 范围内上下文变量一样。

最理想的情形是 无论 我们从什么页面到 `/hello.jsp`（或者从`/hello.jsp`回到`/hello.jsp`），在值绑定中被引用的模型属性的值都应该被“记住”，而不需要对话来存储（或者其他的服务器端状态来存储）。

5.1.1.1.2. 传播请求参数

如果只是指定 `name` 属性，那么请求参数就会利用 `PAGE` 进行上下文传播（它没有被映射成模型属性）。

```
<pages>
  <page view-id="/hello.jsp" action="#{helloWorld.sayHello}">
    <param name="firstName" />
    <param name="lastName" />
  </page>
</pages>
```

如果你想要建立多层的复杂CRUD页面，页面参数的传递尤其有用。你可以用它“记住”你前面到过的页面（例如当按了保存按钮时）和正在编辑的实体。

- 很明显，如果参数是视图的页面参数的话，任何 `<s:link>` 或者 `<s:button>` 都会传播请求参数。
- 这个值很明显是由带有指定视图id的页面的任何jsf页面表单提交传播的。（这意味着视图参数表现得就像faces请求的`PAGE`范围内视图参数一样。）

所有这些听起来很复杂，你可能会想这么一个外来的构造是否真的值得去努力。实际上，一旦你“掌握了它”，有这种想法非常自然。理解这些资料显然需要花费时间的。页面参数是跨越 `non-faces` 请求来传播状态的最优雅方式。对于用可标记的结果页，搜索屏幕的问题尤其有效，在这种情况下，我们喜欢可以写应用程序代码、用同一段代码来处理POST和GET请求。页面参数消除了视图定义中请求参数的重复清单，并使得重定向更容易用代码实现。

5.1.1.1.3. 转换和验证

你可以为复杂的模型属性指定一个JSF转换器：

```
<pages>
  <page view-id="/calculator.jsp" action="#{calculator.calculate}">
    <param name="x" value="#{calculator.lhs}" />
    <param name="y" value="#{calculator.rhs}" />
    <param name="op" converterId="com.my.calculator.OperatorConverter" value="#{calculator.op}" />
  </page>
</pages>
```

或者：

```
<pages>
  <page view-id="/calculator.jsp" action="#{calculator.calculate}">
```

```

    <param name="x" value="#{calculator.lhs}"/>
    <param name="y" value="#{calculator.rhs}"/>
    <param name="op" converter="#{operatorConverter}" value="#{calculator.op}"/>
  </page>
</pages>

```

JSF验证器和 `required="true"` 也可以这样用：

```

<pages>
  <page view-id="/blog.xhtml">
    <param name="date"
      value="#{blog.date}"
      validatorId="com.my.blog.PastDate"
      required="true"/>
  </page>
</pages>

```

或者：

```

<pages>
  <page view-id="/blog.xhtml">
    <param name="date"
      value="#{blog.date}"
      validator="#{pastDateValidator}"
      required="true"/>
  </page>
</pages>

```

更好的方式，基于模型的Hibernate验证器注解会自动被识别和验证。

当类型转换或者验证失败后，一个全局的 `FacesMessage` 就会被添加到 `FacesContext` 中。

5.1.1.2. 导航

你可以使用在Seam应用程序的 `faces-config.xml` 中定义的标准JSF导航规则。然而，JSF导航规则也有许多烦人的限制：

- 在重定向时，不可能指定一个将要使用的请求参数。
- 不可能由一个规则来开始或者结束对话。
- 通过给动作方法求取返回值来运作规则；不可能去给一个任意的EL表达式取值。

更深层次的问题在于”管理“逻辑在 `pages.xml` 和 `faces-config.xml` 之间是分散的。最好是把这种逻辑统一进 `pages.xml` 中。

这个JSF导航规则：

```

<navigation-rule>
  <from-view-id>/editDocument.xhtml</from-view-id>

  <navigation-case>
    <from-action>#{documentEditor.update}</from-action>
    <from-outcome>success</from-outcome>
    <to-view-id>/viewDocument.xhtml</to-view-id>
    <redirect/>
  </navigation-case>

```



```
</navigation-rule>
```

可以重写如下：

```
<page view-id="/editDocument.xhtml">

  <navigation from-action="#{documentEditor.update}">
    <rule if-outcome="success">
      <redirect view-id="/viewDocument.xhtml"/>
    </rule>
  </navigation>

</page>
```

如果我们不必用字符类型的返回值（JSF的结果）来污染 `DocumentEditor` 组件的话会更好。因此 Seam 允许我们写成：

```
<page view-id="/editDocument.xhtml">

  <navigation from-action="#{documentEditor.update}"
              evaluate="#{documentEditor.errors.size}">
    <rule if-outcome="0">
      <redirect view-id="/viewDocument.xhtml"/>
    </rule>
  </navigation>

</page>
```

或者甚至可以写成：

```
<page view-id="/editDocument.xhtml">

  <navigation from-action="#{documentEditor.update}">
    <rule if="#{documentEditor.errors.empty}">
      <redirect view-id="/viewDocument.xhtml"/>
    </rule>
  </navigation>

</page>
```

第一种形式计算一个值绑定，来确定要被后面的一系列导航规则所使用的结果值。第二种方法忽略结果，并为每个可能的规则来计算值绑定。

当然，当一个更新成功，我们可能想要结束当前的对话。我们可以这样做：

```
<page view-id="/editDocument.xhtml">

  <navigation from-action="#{documentEditor.update}">
    <rule if="#{documentEditor.errors.empty}">
      <end-conversation/>
      <redirect view-id="/viewDocument.xhtml"/>
    </rule>
  </navigation>

</page>
```

由于我们终止了会话，后面的任何请求都无法知道我们对哪个文档感兴趣。我们可以将文档id作为一个请求参数传递，这样也使得视图变成是可标记的：

```
<page view-id="/editDocument.xhtml">

  <navigation from-action="#{documentEditor.update}">
    <rule if="#{documentEditor.errors.empty}">
      <end-conversation/>
      <redirect view-id="/viewDocument.xhtml">
        <param name="documentId" value="#{documentEditor.documentId}" />
      </redirect>
    </rule>
  </navigation>

</page>
```

在JSF中，null是一个特殊的结果。结果null被解释成“重新显示页面”。下面的导航规则符合任何非null的结果，而 不符合 null的结果：

```
<page view-id="/editDocument.xhtml">

  <navigation from-action="#{documentEditor.update}">
    <rule>
      <render view-id="/viewDocument.xhtml"/>
    </rule>
  </navigation>

</page>
```

如果结果出现null，你还想执行导航，就使用下面的形式：

```
<page view-id="/editDocument.xhtml">

  <navigation from-action="#{documentEditor.update}">
    <render view-id="/viewDocument.xhtml"/>
  </navigation>

</page>
```

view-id可以作为一个JSF EL表达式提供：

```
<page view-id="/editDocument.xhtml">

  <navigation if-outcome="success">
    <redirect view-id="/#{userAgent}/displayDocument.xhtml"/>
  </navigation>

</page>
```

5.1.1.3. 导航的定义、页面动作和参数的细粒度文件

如果你有很多不同的页面动作和页面参数，或者甚至是很多导航规则，你就会很想把这些声明开放到多个文件中去。你可以在一个名为 calc/calculator.page.xml 的资源中，为一个有着视图id /calc/calculator.jsp 的页面定义动作和参数。这个例子中的根元素是 <page> 元素，隐含着视图id：

```
<page action="#{calculator.calculate}">
  <param name="x" value="#{calculator.lhs}" />
  <param name="y" value="#{calculator.rhs}" />
```

```
<param name="op" converter="#{operatorConverter}" value="#{calculator.op}"/>
</page>
```

5.1.2. 组件驱动的事件

Seam组件可以通过方法间简单的调用相互影响。状态组件甚至实现 Observer/Observable 模式。但在组件直接调用彼此方法的时候，为了使组件在一个比可能存在的更加松耦合的方式下相互作用，Seam提供了 组件驱动事件。

我们在 components.xml 里指定了事件监听器（观察者）。

```
<components>
  <event type="hello">
    <action execute="#{helloListener.sayHelloBack}"/>
    <action execute="#{logger.logHello}"/>
  </event>
</components>
```

在这里，event type 是任意的字符串。

事件发生时，该事件已经注册过的动作将按照它们在 components.xml 中出现的顺序被依次调用。组件如何发起事件？Seam为此提供了一个内置的组件。

```
@Name("helloWorld")
public class HelloWorld {
  public void sayHello() {
    FacesMessages.instance().add("Hello World!");
    Events.instance().raiseEvent("hello");
  }
}
```

或者你可以使用注解。

```
@Name("helloWorld")
public class HelloWorld {
  @RaiseEvent("hello")
  public void sayHello() {
    FacesMessages.instance().add("Hello World!");
  }
}
```

注意这个事件产生器没有依赖任何事件消费者。事件监听器现在可以完全不依赖于产生器而实现：

```
@Name("helloListener")
public class HelloListener {
  public void sayHelloBack() {
    FacesMessages.instance().add("Hello to you too!");
  }
}
```

上述在 components.xml中定义的方法绑定关心把事件映射到消费者去。如果你不喜欢 components.xml 文件中的那一套，可以用注解来替代：

```
@Name("helloListener")
```

```
public class HelloListener {
    @Observer("hello")
    public void sayHelloBack() {
        FacesMessages.instance().add("Hello to you too!");
    }
}
```

你可能想知道为什么在这个讨论中没有提到关于任何事件对象的东西。在Seam中，对事件对象而言，不需要在事件生产者和监听器之间传播状态。状态保留在Seam上下文中，在组件之间共享。然而，如果你真想传递事件对象，你可以：

```
@Name("helloWorld")
public class HelloWorld {
    private String name;
    public void sayHello() {
        FacesMessages.instance().add("Hello World, my name is #0.", name);
        Events.instance().raiseEvent("hello", name);
    }
}
```

```
@Name("helloListener")
public class HelloListener {
    @Observer("hello")
    public void sayHelloBack(String name) {
        FacesMessages.instance().add("Hello #0!", name);
    }
}
```

5.1.3. 上下文事件

Seam定义了许多内置事件，应用程序可以用它们来进行特殊类型的框架集成。这些事件是：

- `org.jboss.seam.validationFailed` — JSF验证失败时被调用
- `org.jboss.seam.noConversation` — 没有长时间运行的对话在运行或者长时间运行的对话被请求时调用
- `org.jboss.seam.preSetVariable.<name>` — 设置上下文变量 `<name>` 时调用
- `org.jboss.seam.postSetVariable.<name>` — 设置上下文变量 `<name>` 时调用
- `org.jboss.seam.preRemoveVariable.<name>` — 未设置上下文变量 `<name>` 时调用
- `org.jboss.seam.postRemoveVariable.<name>` — 未设置上下文变量 `<name>` 时调用
- `org.jboss.seam.preDestroyContext.<SCOPE>` — 在 `<SCOPE>` 上下文被销毁之前调用
- `org.jboss.seam.postDestroyContext.<SCOPE>` — 在 `<SCOPE>` 上下文被销毁之后调用
- `org.jboss.seam.beginConversation` — 当一个长时间运行的对话开始的时候调用
- `org.jboss.seam.endConversation` — 当一个长时间运行的对话结束的时候调用
- `org.jboss.seam.beginPageflow.<name>` — 在页面流 `<name>` 开始时调用
- `org.jboss.seam.endPageflow.<name>` — 在页面流 `<name>` 结束时调用

- `org.jboss.seam.createProcess.<name>` — 在创建进程 `<name>` 时调用
- `org.jboss.seam.endProcess.<name>` — 在进程 `<name>` 结束时调用
- `org.jboss.seam.initProcess.<name>` — 在进程 `<name>` 与对话相关联时调用
- `org.jboss.seam.initTask.<name>` — 在任务 `<name>` 与对话相关联时调用
- `org.jboss.seam.startTask.<name>` — 在任务 `<name>` 开始时调用
- `org.jboss.seam.endTask.<name>` — 在结束任务 `<name>` 时调用
- `org.jboss.seam.postCreate.<name>` — 在创建组件 `<name>` 时调用
- `org.jboss.seam.preDestroy.<name>` — 在销毁组件 `<name>` 时调用
- `org.jboss.seam.beforePhase` — 在开始一个JSF阶段之前调用
- `org.jboss.seam.afterPhase` — 在一个JSF阶段结束之后调用
- `org.jboss.seam.postInitialization` — 当Seam被初始化并启动所有组件时被调用
- `org.jboss.seam.postAuthenticate.<name>` — 用户认证之后调用
- `org.jboss.seam.preAuthenticate.<name>` — 在尝试认证用户之前调用
- `org.jboss.seam.notLoggedIn` — 在不需要认证用户和需要认证的时候调用
- `org.jboss.seam.rememberMe` — 当Seam安全在cookie中发现用户名时发生
- `org.jboss.seam.exceptionHandled.<type>` — 在Seam处理未被捕捉的异常时被调用
- `org.jboss.seam.exceptionHandled` — 在Seam处理未被捕捉的异常时被调用
- `org.jboss.seam.exceptionNotHandled` — 在没有未被捕捉异常的处理器时被调用
- `org.jboss.seam.afterTransactionSuccess` — 当事务在Seam Application Framework中成功时调用
- `org.jboss.seam.afterTransactionSuccess.<name>` — 当管理具名 `<name>` 实体的事务在Seam Application Framework中成功时调用

Seam组件可以用它们观察任何其他组件驱动事件的同样方式来观察这些事件中的任何一种。

5.2. Seam 拦截器

EJB 3.0为会话Bean组件引入了一个标准的拦截器模型。要往Bean里添加拦截器，你需要写一个类，该类有一个被注解过的方法 `@AroundInvoke`，并用 `@Interceptors` 来注解这个Bean以指定拦截器类的名称。例如，下面的拦截器检查用户是否在允许调用动作监听器方法之前登录：

```
public class LoggedInInterceptor {

    @AroundInvoke
    public Object checkLoggedIn(InvocationContext invocation) throws Exception {

        boolean isLoggedIn = Contexts.getSessionContext().get("loggedIn")!=null;
    }
}
```

```

    if (isLoggedIn) {
        //the user is already logged in
        return invocation.proceed();
    }
    else {
        //the user is not logged in, fwd to login page
        return "login";
    }
}
}

```

要把这个拦截器应用到一个作为动作监听器的会话Bean上，我们必须注解这个会话Bean `@Interceptors(LoggedInInterceptor.class)`。这个注解有点丑陋。在EJB 3.0中，Seam通过允许将 `@Interceptors` 作为元注解使用，而依赖于拦截器框架。在我们的例子中，将创建一个 `@LoggedIn` 注解，如下所示：

```

@Target(TYPE)
@Retention(RUNTIME)
@Interceptors(LoggedInInterceptor.class)
public @interface LoggedIn {}

```

现在，我们可以简单地用 `@LoggedIn` 来注解我们的动作监听器Bean以应用拦截器。

```

@Stateless
@Name("changePasswordAction")
@LoggedIn
@Interceptors(SeamInterceptor.class)
public class ChangePasswordAction implements ChangePassword {

    ...

    public String changePassword() { ... }

}

```

如果拦截器的顺序很重要（通常是这样），你可以将 `@Interceptor` 注解添加到你的拦截器类，来指定拦截器的部分顺序

```

@Interceptor(around={BijectionInterceptor.class,
                    ValidationInterceptor.class,
                    ConversationInterceptor.class},
            within=RemoveInterceptor.class)
public class LoggedInInterceptor
{
    ...
}

```

你甚至可以有一个“客户端”的拦截器，运行关于任何EJB3的内置功能：

```

@Interceptor(type=CLIENT)
public class LoggedInInterceptor
{
    ...
}

```

EJB拦截器是有状态的，有着和它们所拦截组件相同的生命周期。对哪些不需要维护状态的拦截

器而言，Seam通过指定 `@Interceptor(stateless=true)` 让你获得性能优化。

Seam的很多功能是一套内置的Seam拦截器来实现的，包括前面例子里提到的拦截器。你没有必要通过注解你的组件来明确指定这些拦截器；它们为所有的可注解Seam组件而存在。

你甚至可以在JavaBean组件中使用Seam拦截器，不仅仅只有EJB3 Bean能用它们！

EJB定义拦截器，不仅为了业务方法（用`@AroundInvoke`），也为了生命周期方法 `@PostConstruct`，`@PreDestroy`，`@PrePassivate` 和 `@PostActive`。Seam支持组件和拦截器中所有这些生命周期方法，不仅仅支持EJB3 Bean，也支持JavaBean组件（除了`@PreDestroy` 对JavaBean组件而言没有意义之外）。

5.3. 管理异常

JSF在异常处理方面的能力有限得令人吃惊。作为解决这个问题的部分权宜之计，Seam让你定义如何通过注解这个异常类来处理异常的特殊类，或者在XML文件中声明这个异常类。这个工具是想要和EJB3.0标准的 `@ApplicationException` 的注解组合在一起，这个注解指定了这个异常是否应该导致一个事务回滚。

5.3.1. 异常和事务

EJB指定了定义良好的规则，用以控制异常是否立即标记当前的事务，以便在这个Bean的业务方法抛出一个异常时回滚：系统异常总是导致一个事务回滚，应用程序异常默认是不导致事务回滚的，但是如果指定了 `@ApplicationException(rollback=true)`，则会导致事物回滚。（应用程序异常是任何checked异常，或者任何用 `@ApplicationException` 注解过的unchecked的异常。系统异常是任何没有用 `@ApplicationException` 注解过的unchecked异常）。

注意：在标记事务回滚和实际的回滚两者之间有一点不同。异常规则说，只有被标记过的事务应该回滚，但是在异常抛出之后，事务仍然可以是有效的。

Seam对Seam JavaBean组件也应用EJB 3.0 异常回滚规则。

但是，这些规则仅仅应用于Seam组件层。没有捕捉到的异常传播到Seam组件层之外，或是传播到JSF层之外怎么办？恩，让一个悬空摇摆的事务处于打开状态是不对的，当异常发生，而你又没有在Seam组件层捕捉到它时，Seam会回滚任何活动的事务。

5.3.2. 激活Seam异常处理

要激活Seam的异常处理，需要确保已经在 `web.xml` 中声明了主要的Servlet过滤器：

```
<filter>
  <filter-name>Seam Filter</filter-name>
  <filter-class>org.jboss.seam.servlet.SeamFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>Seam Filter</filter-name>
  <url-pattern>*.seam</url-pattern>
</filter-mapping>
```

如果你想激活异常处理器，还需要禁用 `web.xml` 中Facelets的开发模式，和 `components.xml` 中的调

试模式。

5.3.3. 使用注解处理异常

每当异常传播到Seam组件层之外时，下列异常都会导致一个HTTP 404错误。抛出异常时，它并不立即回滚当前事务，但是如果这个异常没有被其他的Seam组件捕捉到，当前事务将被回滚。

```
@HttpError(errorCode=404)
public class ApplicationException extends Exception { ... }
```

每当异常传播到Seam组件层之外时，这个异常会导致浏览器的重定向。它也同时结束当前的对话，导致当前事务立即回滚。

```
@Redirect(viewId="/failure.xhtml", end=true)
@ApplicationException(rollback=true)
public class UnrecoverableApplicationException extends RuntimeException { ... }
```

注意：对于那些在JSF生命周期的渲染阶段发生的异常而言，`@Redirect` 无效。

你也可以用EL指定 `viewId` 来重定向。

当异常传播到Seam组件层之外时，这个异常导致一个重定向，并给用户一条消息。它也立即回滚当前事务。

```
@Redirect(viewId="/error.xhtml", message="Unexpected error")
public class SystemException extends RuntimeException { ... }
```

5.3.4. 用XML处理异常

考虑到不能对我们感兴趣的所有异常类添加注解，Seam也允许我们在 `pages.xml` 中指定这个功能。

```
<pages>

  <exception class="javax.persistence.EntityNotFoundException">
    <http-error error-code="404"/>
  </exception>

  <exception class="javax.persistence.PersistenceException">
    <end-conversation/>
    <redirect view-id="/error.xhtml">
      <message>数据库访问失败 Database access failed</message>
    </redirect>
  </exception>

  <exception>
    <end-conversation/>
    <redirect view-id="/error.xhtml">
      <message>意外的失败 Unexpected failure</message>
    </redirect>
  </exception>

</pages>
```

最后一个 `<exception>` 声明没有指定类，它捕捉所有的那些没有通过注解或在 `pages.xml` 中特别指定的任何异常。

你也可以通过EL指定 `view-id` 来重定向。

你也可以通过EL访问处理后的异常实例，Seam把它放在对话上下文中，比如访问异常的消息。

```
...
throw new AuthorizationException("You are not allowed to do this!");

<pages>

    <exception class="org.jboss.seam.security.AuthorizationException">
        <end-conversation/>
        <redirect view-id="/error.xhtml">
            <message severity="WARN">#{org.jboss.seam.handledException.message}</message>
        </redirect>
    </exception>

</pages>
```

`org.jboss.seam.handledException` 保存着实际上由异常处理器处理的嵌套异常。最外层的（包装器）异常也可以访问，如 `org.jboss.seam.exception`。

5.3.5. 一些常见的异常

如果你正在使用JPA：

```
<exception class="javax.persistence.EntityNotFoundException">
    <redirect view-id="/error.xhtml">
        <message>Not found</message>
    </redirect>
</exception>

<exception class="javax.persistence.OptimisticLockException">
    <end-conversation/>
    <redirect view-id="/error.xhtml">
        <message>另一个用户修改了相同的数据，请重试 Another user changed the same data, please try again</message>
    </redirect>
</exception>
```

如果你正在使用Seam应用框架：

```
<exception class="org.jboss.seam.framework.EntityNotFoundException">
    <redirect view-id="/error.xhtml">
        <message>Not found</message>
    </redirect>
</exception>
```

如果你正在使用Seam安全：

```
<exception class="org.jboss.seam.security.AuthorizationException">
    <redirect>
        <message>You don't have permission to do this</message>
    </redirect>
</exception>

<exception class="org.jboss.seam.security.NotLoggedInException">
    <redirect view-id="/login.xhtml">
        <message>Please log in first</message>
    </redirect>
</exception>
```

那么，对于JSF：

```
<exception class="javax.faces.application.ViewExpiredException">
  <redirect view-id="/error.xhtml">
    <message>您的会话已经超时，请重试 Your session has timed out, please try again</message>
  </redirect>
</exception>
```

如果用户会话过期并且返回到原来的页面，就会抛出 `ViewExpiredException` 异常。如果你在一个对话里面，`no-conversation-view-id` 和 `conversation-required` 可以让你更细粒度地控制会话超期。

第 6 章 对话以及工作区管理

现在该更详细地了解一下Seam的对话模型了。

从历史上看，Seam的“对话Conversation”概念是由三个不同的概念合并而成的。

- 工作区（workspace）的概念，是我2002年在给英国政府做项目中遇到的概念，当时我被迫在struts之上实现工作间，我祈求永远不要再重复这样的遭遇。
- 语义乐观的应用程序事务（application transaction with optimistic semantics）的概念，以及基于无状态构架的现有框架的实现，都无法提供对被扩展持久化上下文的有效管理。（Hibernate团队确实已经受够了由于LazyInitializationException异常的指责，但这实际上并不是Hibernate自身的错误，而是因为像Spring框架这样的无状态构架，或者J2EE中传统的无状态会话Facade（反）模式支持极端限制的持久化上下文模型所造成的。）
- 工作流 任务的概念。

通过统一以上这些概念并提供底层框架的支持，我们就有了一个强大的构造能力，它使我们能够用比以前更少的代码构建出功能更加丰富且更加高效的应用程序。

6.1. Seam的对话模型

我们目前为止所看到的例子仅仅使用非常简单的对话模型，它遵循以下这些规则：

- 在应用JSF请求值、处理验证、更新模型值、调用应用程序，以及渲染JSF请求生命周期的响应阶段期间，始终都有一个激活的会话上下文。
- 在JSF请求生命周期恢复视图阶段的最后，Seam将会试图恢复之前长时间运行的任何对话上下文。如果这种上下文不存在，Seam将会创建一个新的临时对话上下文。
- 当遇到 @Begin 方法时，临时对话上下文会被提升为“长时间运行”的对话。
- 当遇到 @End 方法时，任何“长时间运行”对话上下文都将会被降级为临时对话。
- 在JSF请求生命周期渲染阶段的最后，Seam会保存“长时间运行”对话的内容，或者销毁临时对话上下文的内容。
- 任何“faces request”（一种JSF postback）都会传播对话上下文。在默认情况下，非“faces request”（例如GET请求）都不会传播对话上下文，欲知详情，请看下面分解。
- 如果JSF请求生命周期被一个重定向redirect命令中止，Seam将会透明地保存并恢复当前的对话上下文——除非该对话已经通过 @End(beforeRedirect=true) 中止。

Seam透明地在JSF postback以及重定向redirect时传递对话上下文。如果你不需要做任何特殊的事情，使用“non-faces request”（例如GET请求）就不会传递对话上下文，并且它会在一个新的临时对话中被处理。这通常（但并非总是）是我们期望的一种行为。

如果你希望在“non-faces request”中传递Seam对话，就需要显式地将Seam的 conversation id 编写为一个request参数：

```
<a href="main.jsf?conversationId=#{conversation.id}">Continue</a>
```

或者更JSF的做法是：

```
<h:outputLink value="main.jsf">
  <f:param name="conversationId" value="#{conversation.id}" />
  <h:outputText value="Continue" />
</h:outputLink>
```

如果你使用Seam标签库，就等同于：

```
<h:outputLink value="main.jsf">
  <s:conversationId />
  <h:outputText value="Continue" />
</h:outputLink>
```

如果你不想给一个postback传播会话上下文，可以使用一个类似的小窍门：

```
<h:commandLink action="main" value="Exit">
  <f:param name="conversationPropagation" value="none" />
</h:commandLink>
```

如果你使用Seam标签库，则等同于：

```
<h:commandLink action="main" value="Exit">
  <s:conversationPropagation type="none" />
</h:commandLink>
```

注意不使用对话上下文传播与结束对话绝对不是同一回事。

请求参数 `conversationPropagation`，或者 `<s:conversationPropagation>` 标签甚至都可以用来开始和结束对话，或者开始一个嵌套对话。

```
<h:commandLink action="main" value="Exit">
  <s:conversationPropagation type="end" />
</h:commandLink>
```

```
<h:commandLink action="main" value="Select Child">
  <s:conversationPropagation type="nested" />
</h:commandLink>
```

```
<h:commandLink action="main" value="Select Hotel">
  <s:conversationPropagation type="begin" />
</h:commandLink>
```

```
<h:commandLink action="main" value="Select Hotel">
  <s:conversationPropagation type="join" />
</h:commandLink>
```

这种会话模型可以非常容易地创建基于多窗口操作的应用系统。对于许多应用程序来说，这已经足够了。但是另外一些复杂的应用程序还会需要以下额外需求中的一点或两点。

- 一个对话范围跨越多个更小的用户交互单元，这些小单元逐个或者同步地执行。 更小的 嵌套对话拥有它们自己的一套独立的对话状态，并且也可以访问外部对话的状态。
- 用户能够在同一个浏览窗口中的多个对话之间进行切换。这种功能称做 工作区管理 。

6.2. 嵌套对话

嵌套对话是通过在一个现有对话的范围内调用一个名为 `@Begin(nested=true)` 的方法进行创建的。嵌套对话有它自己的对话上下文，还可以只读地访问外部对话的上下文（它可以读取外部对话的上下文变量，但是不可以写）。随后当遇到 `@End` 时，嵌套对话会被销毁，并且外部对话会弹出会话堆栈继续运行。理论上，对话可以嵌套到任意层深。

某个用户活动（工作区管理，或返回按钮）可以在内部对话结束之前就恢复外部对话。在这种情况下，一个外部对话就有可能同时拥有多个嵌套对话。如果外部对话在嵌套对话之前就被结束，Seam将会把嵌套对话和外部对话一起销毁掉。

对话可以被认为是一个 连续的状态 。嵌套对话允许应用程序在不同的用户交互点捕捉一致连续的状态，因此必须确保在返回按钮以及工作区管理的面上有真正的正确行为。

TODO: 说明当你点击返回按钮时嵌套对话如何防止错误发生的一个例子。

通常，如果一个组件存在于当前嵌套对话的父对话中，嵌套对话会使用同一个实例。少数情况下，在每个嵌套对话中都使用不同的实例会很有用，以便存在于父对话中的组件实例对其子对话是不可见的。你可以通过给这个组件注解 `@PerNestedConversation` 来实现。

6.3. 使用GET请求来开始一个对话

JSF并没有定义任何类型的action监听器，这种监听器会在通过非JSF请求“non-faces request”访问页面的时候被触发（例如，一个HTTP GET请求）。这种触发会发生在当用户用书签保存了这个页面，或者在我们通过 `<h:outputLink>` 访问页面的时候。

有时候，我们希望在访问页面的时候立即开始一个对话。由于没有JSF action方法，我们不能以寻常的通过用 `@Begin` 标注action的方式来解决这个问题，

当页面需要把一些状态抓取到上下文变量中时，另一个问题也就随之产生了。我们已经看到有两种方法可以解决这个问题。如果这个状态是Seam组件所持有的，我们就可以通过 `@Create` 方法来抓取。如果不是，我们就可以为这个上下文变量定义一个 `@Factory` 方法。

如果以上两种办法都不适合你，Seam还允许你在 `pages.xml` 文件中定义一个 `page action` 。

```
<pages>
  <page view-id="/messageList.jsp" action="#{messageManager.list}"/>
  ...
</pages>
```

这个action方法在开始渲染响应阶段的时候被调用，即在页面就要被渲染的任何时候。如果页面action返回一个非空的值，Seam将会根据Seam导航规则处理任何合适的JSF，可能导致渲染另外一个完全不同的页面。

如果你在渲染页面之前想要做的 仅仅 是开始一个对话，那你可以使用一个内建的action方法，它正好具备这种功能：

```
<pages>
  <page view-id="/messageList.jsp" action="#{conversation.begin}"/>
  ...
</pages>
```

注意你也可以从JSF控制器中调用内建的action来开始一个对话，同样地，你可以使用 `#{conversation.end}` 来结束一个对话。

如果你想要更多的控制，以加入现有的对话或开始一个嵌套对话，开始一个页面流或者开始一个原子的对话，你应该使用 `<begin-conversation>` 元素。

```
<pages>
  <page view-id="/messageList.jsp">
    <begin-conversation nested="true" pageflow="AddItem"/>
  </page>
  ...
</pages>
```

`<end-conversation>`元素也可以结束一个对话。

```
<pages>
  <page view-id="/home.jsp">
    <end-conversation/>
  </page>
  ...
</pages>
```

为了解决第一个问题，我们现在有五种选择：

- 用 `@Begin` 注解 `@Create` 方法
- 用 `@Begin` 注解 `@Factory` 方法
- 用 `@Begin` 注解Seam页面action
- 在 `pages.xml` 中使用 `<begin-conversation>`
- 利用 `#{conversation.begin}` 作为Seam页面action方法

6.4. 利用<s:link>以及<s:button>

JSF命令链始终通过JavaScript来执行一个表单提交，它打破了浏览器的“在新窗口中打开”或者“在新标签中打开”这种特点。在普通的JSF中，如果你需要这项功能，就需要使用 `<h:outputLink>`。但是 `<h:outputLink>` 标签有两大限制。

- JSF没有提供将action监听器附加给 `<h:outputLink>` 的方法。
- 由于实际上没有提交表单，JSF并没有传播 `DataModel` 中的选中行。

Seam提供了一个 page action 的概念来帮助解决第一个问题，但是这对于第二个问题却无能为力。我们可以利用REST的方法传递请求参数以及重新查询服务端的选中对象来解决这个问题。在某些情况下——例如Seam博客上范例应用程序那样——这实际上最好的方法。REST风格支持书签，因为它不需要服务器端的状态。在其他那些我们不需要关心书签的情况下，使用 `@DataModel` 以及 `@DataModelSelection` 就很方便也很透明！

为了填补这项缺失的功能，也为了使对话传播的管理变得更加简单，Seam提供 `<s:link>` 这样一个JSF标签。

这个连接可以仅指定JSF视图的id：

```
<s:link view="/login.xhtml" value="Login"/>
```

或者，它可以指定一个action方法（在这种情况下action的输出将会决定结果页面）：

```
<s:link action="#{login.logout}" value="Logout"/>
```

如果你把JSF视图id和action方法这两者都指定的话，“视图”将会被使用，除非action方法返回一个非空的结果：

```
<s:link view="/loggedOut.xhtml" action="#{login.logout}" value="Logout"/>
```

这个连接自动地利用内部的 `<h:dataTable>` 传播 `DataModel` 的所选定。

```
<s:link view="/hotel.xhtml" action="#{hotelSearch.selectHotel}" value="#{hotel.name}"/>
```

你可以不指定现有对话的范围：

```
<s:link view="/main.xhtml" propagation="none"/>
```

你可以开始、结束或者嵌套对话：

```
<s:link action="#{issueEditor.viewComment}" propagation="nest"/>
```

如果一个连接开始了一个对话，你甚至可以指定一个要使用的页面流：

```
<s:link action="#{documentEditor.getDocument}" propagation="begin"
pageflow="EditDocument"/>
```

如果使用jBPM任务列表，那么你可以使用 `taskInstance` 属性：

```
<s:link action="#{documentApproval.approveOrReject}" taskInstance="#{task}"/>
```

（请见DVD Store演示的应用程序中针对以上用法的范例。）

最后，如果你希望“链接”被渲染成为一个按钮，就使用 `<s:button>`：

```
<s:button action="#{login.logout}" value="Logout"/>
```

6.5. 成功信息

给用户显示一条action执行成功或者失败的消息是相当常见的功能。为此使用JSF的 `FacesMessage` 是非常方便的。不幸的是，成功的action通常需要一个浏览器重定向。这使得在普通的JSF中显示成功信息变得相当困难。

内建的会话范围的Seam组件 `facesMessages` 解决了这个问题。（你必须安装Seam重定向过滤器。）

```
@Name("editDocumentAction")
@Stateless
public class EditDocumentBean implements EditDocument {
    @In EntityManager em;
    @In Document document;
    @In FacesMessages facesMessages;

    public String update() {
        em.merge(document);
        facesMessages.add("Document updated");
    }
}
```

对于当前的会话来说，任何加入到 `facesMessages` 的消息都正好用在下一个渲染阶段中。甚至当没有“长时间运行”对话的时候也会奏效，因为Seam甚至在重定向过程中保留了临时对话。

你甚至可以在faces message概述中包含JSF EL表达式：

```
facesMessages.add("Document #{document.title} was updated");
```

你可以按照通常的方式显示消息，例如：

```
<h:messages globalOnly="true"/>
```

6.6. 使用“显式”的对话id

通常情况下，Seam会给每个对话产生一个无意义且唯一的id。你可以在你开始一个对话的时候定制id的值。

这个特性可以用来定制会话id生成算法，像这样：

```
@Begin(id="#{myConversationIdGenerator.nextId}")
public void editHotel() { ... }
```

或者它可以用来分配一个有意义的对话id：

```
@Begin(id="hotel#{hotel.id}")
public String editHotel() { ... }
```

```
@Begin(id="hotel#{hotelsDataModel.rowData.id}")
public String selectHotel() { ... }
```

```
@Begin(id="entry#{params['blogId']}")
```



```
public String viewBlogEntry() { ... }
```

```
@BeginTask(id="task#{taskInstance.id}")
public String approveDocument() { ... }
```

毫无疑问，每当选一家特殊的酒店、一篇特殊的博客或者一项特殊的任务时，这些例子都会产生一个相同的对话id。那么，如果一相新对话开始时已经存在一个包含相同对话id的对话时，会发生什么情况呢？嗯，Seam竟然会发现现有的对话，并重定向到该对话，而不去再次运行 @Begin 方法。这个特性会帮助我们控制在在工作区管理时创建的多个工作区。

6.7. 工作区管理

工作区管理指的是可以在一个单独的窗口中“切换”多个对话的能力。Seam在Java代码级别完全透明地管理工作区。为了启用工作区管理，你所需要做的全部事情如下：

- 为每个视图id（在使用JSF或Seam导航规则时）或者页面节点（在使用JPD页面流时）提供一个描述文本。这个描述文本通过工作区切换器显示给用户。
- 在你的页面中包含一个或多个标准JSP或facelets片断的工作区转换器。标准片断支持通过下拉菜单、对话列表或者导航控件来管理工作区。

6.7.1. 工作区管理及JSF导航

当你使用JSF或者Seam导航规则的时候，Seam会通过恢复对话的当前 view-id 切换到该对话。工作区的描述文本在一个名为 pages.xml 的文件中定义，Seam希望在 WEB-INF 目录中找到它，这个文件就放在 faces-config.xml 旁边。

```
<pages>
  <page view-id="/main.xhtml">Search hotels: #{hotelBooking.searchString}</page>
  <page view-id="/hotel.xhtml">View hotel: #{hotel.name}</page>
  <page view-id="/book.xhtml">Book hotel: #{hotel.name}</page>
  <page view-id="/confirm.xhtml">Confirm: #{booking.description}</page>
</pages>
```

注意，如果找不到这个文件，Seam应用程序会继续正常地运行！只不过会失去切换工作区的功能。

6.7.2. 工作区管理和jPDL页面流

当你使用jPDL页面流程定义的时候，Seam通过恢复当前jBPM流程状态切换到一个对话。这是一个更加灵活的模型，因为它允许同一个 view-id 根据当前的 <页面> 节点而拥有不同的描述。这个描述文本通过 <page> 节点来定义。

```
<pageflow-definition name="shopping">

  <start-state name="start">
    <transition to="browse"/>
  </start-state>

  <page name="browse" view-id="/browse.xhtml">
```

```

    <description>DVD Search: #{search.searchPattern}</description>
    <transition to="browse"/>
    <transition name="checkout" to="checkout"/>
</page>

<page name="checkout" view-id="/checkout.xhtml">
    <description>Purchase: $#{cart.total}</description>
    <transition to="checkout"/>
    <transition name="complete" to="complete"/>
</page>

<page name="complete" view-id="/complete.xhtml">
    <end-conversation />
</page>

</pageflow-definition>

```

6.7.3. 对话转换器

在你的JSP或facelets页面中包含以下代码片断，以获得使你可以转换到任何当前对话或者应用程序的任何其他页面的一个下拉菜单：

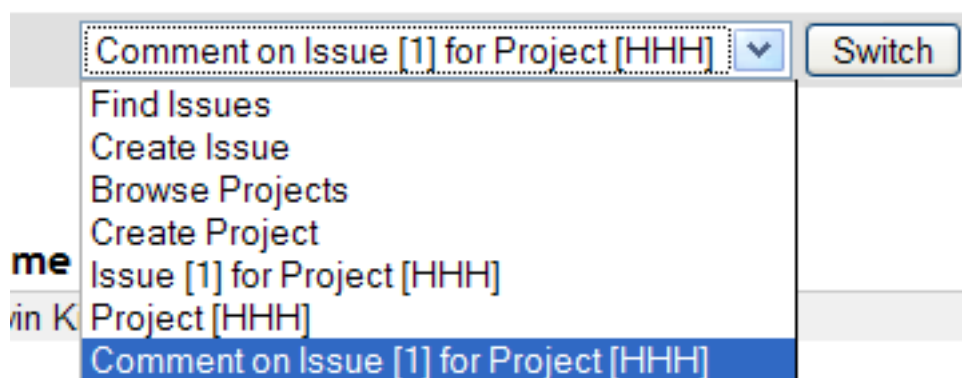
```

<h:selectOneMenu value="#{switcher.conversationIdOrOutcome}">
    <f:selectItem itemLabel="Find Issues" itemValue="findIssue"/>
    <f:selectItem itemLabel="Create Issue" itemValue="editIssue"/>
    <f:selectItems value="#{switcher.selectItems}"/>
</h:selectOneMenu>
<h:commandButton action="#{switcher.select}" value="Switch"/>

```

In this example, we have a menu that includes an item for each conversation, together with two additional items that let the user begin a new conversation. 在这个例子中，我们有一菜单，它为每一个对话都包含了一个条目，另外还有两个让用户开始新对话的条目。

Only conversations with a description will be included in the drop-down menu.



6.7.4. 对话列表

除了对话列表会显示成为一个表格以外，它与对话转换器非常相似：

```

<h:dataTable value="#{conversationList}" var="entry"
    rendered="#{not empty conversationList}">
    <h:column>

```

```

<f:facet name="header">Workspace</f:facet>
<h:commandLink action="#{entry.select}" value="#{entry.description}"/>
<h:outputText value="[current]" rendered="#{entry.current}"/>
</h:column>
<h:column>
<f:facet name="header">Activity</f:facet>
<h:outputText value="#{entry.startDatetime}">
<f:convertDateTime type="time" pattern="hh:mm a"/>
</h:outputText>
<h:outputText value=" - "/>
<h:outputText value="#{entry.lastDatetime}">
<f:convertDateTime type="time" pattern="hh:mm a"/>
</h:outputText>
</h:column>
<h:column>
<f:facet name="header">Action</f:facet>
<h:commandButton action="#{entry.select}" value="#{msg.Switch}"/>
<h:commandButton action="#{entry.destroy}" value="#{msg.Destroy}"/>
</h:column>
</h:dataTable>

```

我们设想你想要根据你自己的应用程序去定制这个。

Workspace	Workspace activity	Action
Comment on Issue [1] for Project [HHH]	01:18 PM - 01:18 PM	<input type="button" value="Switch"/> <input type="button" value="Destroy"/>
Issue [1] for Project [HHH]	01:18 PM - 01:18 PM	<input type="button" value="Switch"/> <input type="button" value="Destroy"/>
Project [HHH]	01:18 PM - 01:18 PM	<input type="button" value="Switch"/> <input type="button" value="Destroy"/>

只有带有描述的对话才会被包含在列表中。

注意对话列表允许用户销毁工作区。

6.7.5. 导航控件

导航控件在使用嵌套对话模式的应用程序中很有用。导航控件是当前对话堆栈中对话链接的一个列表。

```

<ui:repeat value="#{conversationStack}" var="entry">
<h:outputText value=" | "/>
<h:commandLink value="#{entry.description}" action="#{entry.select}"/>
</ui:repeat>

```

[Home](#) | [Find Issues](#) | [Create Issue](#) | [Project \[HHH\]](#) | [Issue \[1\] for Project \[HHH\]](#)

Issue Attributes

6.8. 对话组件和JSF组件绑定

对话组件有一个小小的限制：它们不能够被用来保存对JSF组件的绑定。（除非绝对必要，否则我们通常不喜欢使用JSF的这个特性，因为它创建了从应用程序逻辑到视图的强依赖关系。）在一个postback请求中，组件绑定会在视图恢复阶段中且在Seam对话上下文恢复之前被更新。

为了解决这个问题，使用一个事件范围的组件来保存组件绑定，并将它注入到需要它的对话范围

组件中。

```
@Name("grid")
@Scope(ScopeType.EVENT)
public class Grid
{
    private HtmlPanelGrid htmlPanelGrid;

    // getters and setters
    ...
}
```

```
@Name("gridEditor")
@Scope(ScopeType.CONVERSATION)
public class GridEditor
{
    @In(required=false)
    private Grid grid;

    ...
}
```

另外一种选择是，你可以通过隐式的 `uiComponent` 句柄来访问JSF组件树。下面这个例子访问在迭代中支持数据表的 `UIData` 组件的 `getRowIndex()`，它打印当前的行数：

```
<h:dataTable id="lineItemTable" var="lineItem" value="#{orderHome.lineItems}">
    <h:column>
        Row: #{uiComponent['lineItemTable'].rowIndex}
    </h:column>
    ...
</h:dataTable>
```

在这个map中，可以得到JSF UI组件和它们的客户标识符。

6.9. 对话组件的并发调用

在 ??? 中可以找到Seam组件并发调用的全部讨论。在这里，我们要讨论是常见的一种并发情形，在这种情形下，你会遇到从AJAX请求中访问对话组件的并发。我们就要讨论一个Ajax客户端库应该提供用来控制源自客户端的事件的选项，并看看RickFaces为你提供的选项。

对话组件实际上并不允许真正的并发访问，因此Seam会给每一个请求排一个队列并依次处理它们。这样允许每个请求都以一种特定的方式被执行。但是，简单的队列不是那么强大——首先，由于某些原因，如果一个方法需要非常长的时间来完成，并且每次生成一个错误的请求时就一次又一次的反复运行（潜在拒绝服务攻击的可能）。其次，AJAX经常被用于给用户快速的状态更新，因此持续地长时间运行该动作并没有什么用处。

因此Seam给action事件排队等待一段时间（并发的请求超时）；如果不能及时处理事件，它就会创建一个临时的对话并且给用户打印一条信息，让他们了解情况。所以不要给服务器发送泛滥的AJAX事件，这是非常重要的。

我们可以在 `components.xml` 文件中给并发的请求超时设置一个合理的超时默认值（ms）。

```
<core:manager concurrent-request-timeout="500" />
```

到目前为止，我们已经讨论了同步的AJAX请求——客户端告诉服务器发生了一个事件，然后根据返回值来重新渲染局部页面。当AJAX请求是轻量级时，采用这种方法就相当好（这些方法的调用也简单，如：计算一系列数字的总和）。但是当我们需要做复杂的计算时我们应该怎么做呢？

对于大量的计算我们应当使用真正的异步（基于轮询“Poll”的）方法——客户端发送一个AJAX请求到服务器，这使得一个action在服务器端被异步地执行（因此立即就会响应客户端）；然后客户端会在服务器中查询更新。当你运行一个长时间运行的操作时它很有用，因为每一个action都可以执行是非常重要的（你不会希望某些action由于重复或者超时而被丢弃）。

我们应如何来设计对话的AJAX应用程序呢？

首先，你需要决定是否想使用更简单的“同步”请方式，或者是否想要利用“Poll风格”的方法。

如果你要使用“同步”请求的方法，那么你需要评估一下你的AJAX请求需要多长时间才可以结束——它是不是比并发请求超时值要短？如果不是，你也许要修改这个方法的并发请求超时时间（如上所述）。接下来你或许需要在客户端给请求排一个队，防止请求全部涌入服务器。如果这是一个经常发生的事件（例如，按钮按下和输入域的onblur），并且立即更新客户端又不是优先需要考虑的情况，你就应该在客户端设置一个请求延时。当消耗完你的请求延时的时候，该事件中的操作也会在服务器段排成一个队列。

最后，客户端库可能会提供一个选项，它可以放弃最近未完成的重复请求。对这个选项你需要很谨慎，因为在服务端没能放弃未完成的请求时，这个选项可能导致请求全部涌入服务器端。

使用“Poll风格”的设计比较不需要细调。你只要给你的action方法 `@Asynchronous` 进行标注，并确定一个查询时间间隔就可以了：

```
int total;

// This method is called when an event occurs on the client
// It takes a really long time to execute
@Asynchronous
public void calculateTotal() {
    total = someReallyComplicatedCalculation();
}

// This method is called as the result of the poll
// It's very quick to execute
public int getTotal() {
    return total;
}
```

6.9.1. RichFaces Ajax

RichFaces AJAX是最常与Seam一起使用的Ajax库，它提供了上面讨论过的所有控制：

- `eventsQueue` — 提供一个放置事件的队列。所有的事件都排成队列，并且请求被依次发送给服务器端。当服务器没有被拒绝服务攻击时，若一个请求在服务器上需要花费一些时间来执行时，这个是很有用的（比如：大量的计算，从慢的数据源里获取信息）。
- 如果最近在队列中已经有‘相似的’请求，`ignoreDupResponses` — 就会忽略由该请求产生的响应。`ignoreDupResponses="true"` 不会取消 请求在服务端的处理 — 它只是在客户端防止不必要的更新。

这个选项与Seam对话一起使用时应该很小心，因为它允许创建多个并发请求。

- `requestDelay` — 定义请求存在于队列中的时间（ms）。如果这个请求在这个时间内没有被处理，它就会被发送（不管是否接收到response）或者丢弃（如果队列中有更近的相似事件）。

这个选项与Seam的对话一起使用应该很小心，因为它允许创建多个并发请求。你要确定你所设置的延时时间（结合并发请求超时的时间）要长于action的执行时间。

- `<a:poll reRender="total" interval="1000" />` — 请求服务器端，并重新渲染一个需要的区域。

第 7 章 页面流和业务流程

JBoss jBPM 是一个可以运行于任何Java SE或EE环境的业务流程管理引擎。jBPM使用一组节点组成的图形来描述一个业务流程或用户交互过程，这些节点分别表示等待状态、决策、任务、Web页面等等。这个图形使用一种简单的、非常易读的，称为jPDL的XML方言来定义，并且可以使用一个Eclipse插件图形化显示和编辑。jPDL是一种可扩展的语言，适合于解决许多问题，从定义Web应用程序页面流到传统的工作流管理，直到在SOA环境中组织服务。

Seam应用程序使用jBPM解决两个不同的问题：

- 定义包含复杂用户交互的页面流，jPDL流程定义可以用来为一次对话定义页面流。一次Seam对话可以理解为与一个单一用户进行的一次相关的短暂交互。
- 定义主控业务流程。业务流程可能是横跨多个用户的多次对话。流程的状态被保存在jBPM数据库中，所以流程被认为是长时间运行的。协调多个用户的行为远比处理一个单一用户的交互复杂得多，对此jBPM提供了成熟的工具来解决任务管理和多路并发执行的问题。

不要把这两个事情弄混了！它们运行在两个非常不同的层面或粒度中。页面流Pageflow、对话Conversation 和 任务Task 全部来自于一次与单一用户的单一交互。而一个业务流程则横跨许多任务。进一步说，这两种jBPM应用是完全正交的，可以在一起使用或是分开单独使用，或者都不用。

使用Seam不必了解jPDL。如果你完全乐于使用JSF或是Seam导航规则定义页面流，并且你的应用程序更多的是数据驱动而不是流程驱动，可能就不需要jBPM。但是我们发现，根据一个定义良好的图形考虑用户交互有助于我们创建更加有生命力的应用程序。

7.1. Seam中的页面流

在Seam中定义页面流有两种方法：

- 使用JSF或是Seam导航规则 - 无状态的导航模型
- 使用jPDL - 有状态的导航模型

非常简单的应用程序只需要无状态的导航模型。非常复杂的应用程序则应该在不同的场合结合使用这两种模型。每种模型各有优缺点。

7.1.1. 两种导航模型

无状态的模型定义一种映射，把事件的一组命名的逻辑结果直接映射到视图的结果页面。导航规则除了记录哪个页面是事件来源之外不会保留任何状态。这意味着Action监听方法有时必须进行一些页面流转的处理，因为只有它们能访问到应用程序的当前状态。

下面是一个使用JSF导航规则定义页面流的例子：

```
<navigation-rule>
  <from-view-id>/numberGuess.jsp</from-view-id>

  <navigation-case>
    <from-outcome>guess</from-outcome>
    <to-view-id>/numberGuess.jsp</to-view-id>
```

```

        <redirect/>
    </navigation-case>

    <navigation-case>
        <from-outcome>win</from-outcome>
        <to-view-id>/win.jsp</to-view-id>
        <redirect/>
    </navigation-case>

    <navigation-case>
        <from-outcome>lose</from-outcome>
        <to-view-id>/lose.jsp</to-view-id>
        <redirect/>
    </navigation-case>
</navigation-rule>

```

同一个例子，下面使用Seam导航规则定义：

```

<page view-id="/numberGuess.jsp">

    <navigation>
        <rule if-outcome="guess">
            <redirect view-id="/numberGuess.jsp"/>
        </rule>
        <rule if-outcome="win">
            <redirect view-id="/win.jsp"/>
        </rule>
        <rule if-outcome="lose">
            <redirect view-id="/lose.jsp"/>
        </rule>
    </navigation>

</page>

```

如果你觉得导航规则过于繁琐，可以在Action监听方法里直接返回View的id。

```

public String guess() {
    if (guess==randomNumber) return "/win.jsp";
    if (++guessCount==maxGuesses) return "/lose.jsp";
    return null;
}

```

注意这会导致一个重定向，甚至可以指定要在重定向中使用的一些参数。

```

public String search() {
    return "/searchResults.jsp?searchPattern=#{searchAction.searchPattern}";
}

```

有状态模型在一组具名的、合乎逻辑的应用状态之间定义一组重定向。在这种模型中，可以完全使用jPDL页面流定义来表示任意用户的交互流程，并在编写Action监听方法完全不必知道用户的交互流程。

下面是一个使用jPDL定义页面流的例子：

```

<pageflow-definition name="numberGuess">

    <start-page name="displayGuess" view-id="/numberGuess.jsp">
        <redirect/>
    </start-page>

```



```

    <transition name="guess" to="evaluateGuess">
      <action expression="#{numberGuess.guess}" />
    </transition>
  </start-page>

  <decision name="evaluateGuess" expression="#{numberGuess.correctGuess}">
    <transition name="true" to="win"/>
    <transition name="false" to="evaluateRemainingGuesses"/>
  </decision>

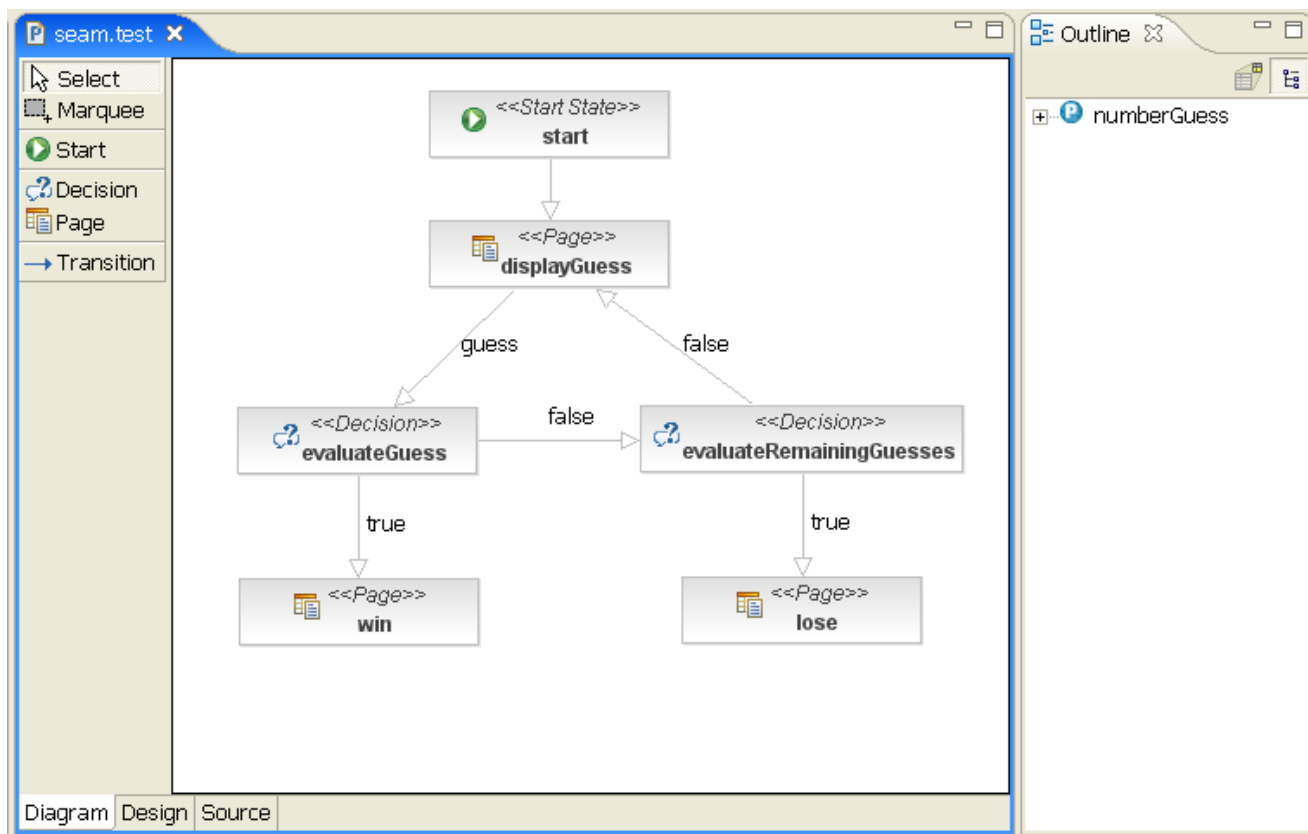
  <decision name="evaluateRemainingGuesses" expression="#{numberGuess.lastGuess}">
    <transition name="true" to="lose"/>
    <transition name="false" to="displayGuess"/>
  </decision>

  <page name="win" view-id="/win.jsp">
    <redirect/>
    <end-conversation />
  </page>

  <page name="lose" view-id="/lose.jsp">
    <redirect/>
    <end-conversation />
  </page>

</pageflow-definition>

```



这里我们马上注意到了两件事：

- JSF/Seam导航规则 更加 简单。（然而，隐藏在后面的Java代码会相当复杂。）
- jPDL让用户交互的过程立刻变得直接易懂，我们甚至不需要考虑JSP或Java代码。

另外，有状态模型更加 受约束 。对于每一种逻辑状态（页面流中的每一步）而言，都存在一组

受限的可能转到其他状态的重定向。而无状态模型则是一种实时的模型，它适用于相对不受约束的、形式自由的导航，在这种情况下，由用户决定他/她下一步想重定向到哪里，而不是应用程序。

有状态和无状态导航之间的差别与传统的模态和非模态交互视图之间的差别很类似。现在，Seam应用程序通常不是模态的，简单地说——的确，使用对话的一个主要原因就是避免应用程序中的模态行为！然而，在一个具体对话的级别里Seam应用程序可以是（甚至经常是）模态的。众所周知，应该尽可能地避免模态行为；预知用户想要做事的顺序是非常困难的！然而，毫无疑问，有状态模型也是需要的。

这两种模型之间的最大不同是后退按钮行为。

7.1.2. Seam和后退按钮

当使用JSF或Seam导航规则时，Seam允许用户通过后退、前进和刷新按钮自由地导航。当用户有以上操作时，应用程序的职责是确保发生对话状态的内部保持一致。许多开发者都将Web应用框架例如Struts或WebWork——它们不支持对话模型——和无状态组件模型例如EJB无状态Session Beans或Spring框架相结合过，这使得他们认为这几乎是不可能的一件事。然而，就我们的经验，在Seam的上下文中，存在一个定义良好的对话模型，基于有状态的Session Bean，实现会话状态的内部一致实际上非常容易。通常，在Action监听方法的开始把no-conversation-view-id的使用和null检查结合起来是很简单的。我们认为支持自由导航将几乎总是可行的。

既然如此，pages.xml中就包含一个no-conversation-view-id声明。它告诉Seam，如果在一个对话当中有一个页面渲染的请求，而这个对话不再存在，就重定向到一个不同的页面。

```
<page view-id="/checkout.xhtml"
      no-conversation-view-id="/main.xhtml"/>
```

另一方面，在有状态的模型中，后退按钮被认为是一个返回前一状态的未定义跳转。因为有状态模型强迫当前状态必须有一组明确定义的跳转，所以后退按钮在有状态模型中是被默认禁止的！Seam明确地监测到后退按钮的使用，并且阻止执行前一个Action和访问过期的页面，而后简单地将用户重定向回当前页面（并且显示一个消息）。这到底是有状态模型的一个特性还是一个限制，则取决于你在其中的角色：作为一个开发者，这是一个特性；作为用户，这可能是一个限制！可以为一些特殊的Page节点设置back="enabled"属性，从而允许后退按钮导航。

```
<page name="checkout"
      view-id="/checkout.xhtml"
      back="enabled">
  <redirect/>
  <transition to="checkout"/>
  <transition name="complete" to="complete"/>
</page>
```

这个配置允许使用后退按钮从checkout状态转到任意前一个状态！

当然，如果在一个页面流中有一个页面渲染的请求，并且这个页面流的对话不再存在，我们依然需要决定如何处理这种情况。既然这样，需要在页面流定义中包括一个no-conversation-view-id声明：

```
<page name="checkout"
      view-id="/checkout.xhtml"
      back="enabled"
      no-conversation-view-id="/main.xhtml">
  <redirect/>
  <transition to="checkout"/>
  <transition name="complete" to="complete"/>
</page>
```

```
</page>
```

在实践中，这两种导航模型都有它们自己的用处，你将很快学会如何在这两种模型之间做出选择。

7.2. 使用jPDL页面流

7.2.1. 安装页面流

我们需要安装Seam jBPM相关的组件，并且告诉它们在哪里找到页面流定义。我们可以在components.xml 配置文件中指定这个配置。

```
<bpm:jbpm>
  <bpm:pageflow-definitions>
    <value>pageflow.jpdl.xml</value>
  </bpm:pageflow-definitions>
</bpm:jbpm>
```

第一行安装jBPM，第二行指向一个基于jPDL的页面流定义。

7.2.2. 开始页面流

我们可以通过在 @Begin、@BeginTask 或 @StartTask 的注解中指定流程定义的名字来“启动”一个基于jPDL的页面流。

```
@Begin(pageflow="numberguess")
public void begin() { ... }
```

作为选择，我们也可以使用pages.xml来启动一个页面流：

```
<page>
  <begin-conversation pageflow="numberguess"/>
</page>
```

在上面这个例子中，如果我们在渲染阶段 RENDER_RESPONSE 阶段——例如，在一个 @Factory 或 @Create 方法中——启动一个页面流，那就意味着我们已经处在已产生的页面中，就要使用页面流中的 <start-page> 节点作为页面流的第一个节点，如上例所示。

但是如果这个页面流作为一个Action监听器的执行结果而启动的话，这个action监听器的结果则将决定哪个页面作为第一个被渲染的页面。既然如此，我们在页面流中使用一个 <start-state> 作为第一个节点，并且为每一个可能的结果声明一个跳转。

```
<pageflow-definition name="viewEditDocument">

  <start-state name="start">
    <transition name="documentFound" to="displayDocument"/>
    <transition name="documentNotFound" to="notFound"/>
  </start-state>

  <page name="displayDocument" view-id="/document.jsp">
    <transition name="edit" to="editDocument"/>
  </page>
</pageflow-definition>
```

```

    <transition name="done" to="main"/>
  </page>

  ...

  <page name="notFound" view-id="/404.jsp">
    <end-conversation/>
  </page>

</pageflow-definition>

```

7.2.3. 页面节点和跳转

每一个 `<page>` 节点描绘一种状态，在该状态下系统等待用户的输入：

```

<page name="displayGuess" view-id="/numberGuess.jsp">
  <redirect/>
  <transition name="guess" to="evaluateGuess">
    <action expression="#{numberGuess.guess}" />
  </transition>
</page>

```

其中的 `view-id` 是一个JSF 视图id。 `<redirect/>` 元素跟JSF导航规则中的 `<redirect/>` 有相同的作用：即，一个post-then-redirect（提交后跳转）行为，这样做是为了防止用户点击了浏览器的刷新按钮进行重复刷新。（需要注意的是，Seam会在这些浏览器重定向中保存对话的上下文，所以在Seam中不需要类似ROR（Ruby On Rails）风格的”flash”构造！）

跳转的名字是一个JSF输出的名字，在 `numberGuess.jsp` 中通过点击一个Command按钮或是Command链接触发这个跳转。

```

<h:commandButton type="submit" value="Guess" action="guess"/>

```

当点击这个按钮触发这个跳转之后，jBPM将会通过调用 `numberGuess` 组件的 `guess()` 方法激活跳转Action。注意到jPDL中用于指定Action的语法类似于JSF的EL表达式，并且这个跳转的Action Handler不过是Seam当前上下文中的一个组件里的一个方法。同JSF事件一样，我们可以在jBPM中拥有完全相同的事件模型！（同一类型准则）

在没有指定输出的情况下（例如，一个Command按钮没有定义 `action` 属性），如果存在没有指定name的跳转，Seam将激活该跳转，或者如果所有的跳转都有name的话，Seam简单地重新显示该页。所以我们可以稍微地简化示例页面流和这个按钮：

```

<h:commandButton type="submit" value="Guess"/>

```

将激活一个未命名的跳转：

```

<page name="displayGuess" view-id="/numberGuess.jsp">
  <redirect/>
  <transition to="evaluateGuess">
    <action expression="#{numberGuess.guess}" />
  </transition>
</page>

```

甚至可以通过点击一个按钮来执行一个Action方法，在这种情况下，Action执行的结果将决定采

用哪个跳转：

```
<h:commandButton type="submit" value="Guess" action="#{numberGuess.guess}"/>
```

```
<page name="displayGuess" view-id="/numberGuess.jsp">
  <transition name="correctGuess" to="win"/>
  <transition name="incorrectGuess" to="evaluateGuess"/>
</page>
```

然而，这是一种不可取的方式，因为它把流程控制的职责从页面流定义转到了其他组件。最好还是把这些相关的职责集中到页面流本身。

7.2.4. 流程控制

通常，我们定义页面流的时候不需要更多强大的jPDL特性，然而我们还是需要 `<decision>` 节点。

```
<decision name="evaluateGuess" expression="#{numberGuess.correctGuess}">
  <transition name="true" to="win"/>
  <transition name="false" to="evaluateRemainingGuesses"/>
</decision>
```

通过在Seam上下文中执行一个JSF EL表达式来决定如何跳转。

7.2.5. 流程的结束

使用 `<end-conversation>` 或是 `@End` 结束对话。（实际上，为了程序的易读性，鼓励两者共同使用。）

```
<page name="win" view-id="/win.jsp">
  <redirect/>
  <end-conversation/>
</page>
```

可选择的，可以指定一个jBPM的 状态转移（transition） 名字结束一个任务。在这种情况下，Seam将给主控的业务流程发送一个结束当前任务的信号。

```
<page name="win" view-id="/win.jsp">
  <redirect/>
  <end-task transition="success"/>
</page>
```

7.2.6. 页面流组合

多个页面流可以进行组合，并当另一个页面流执行时暂停一个页面流。这个 `<process-state>` 节点的作用就是暂停外部的页面流，同时开始执行一个命名的页面流。

```
<process-state name="cheat">
  <sub-process name="cheat"/>
  <transition to="displayGuess"/>
</process-state>
```

这个内部页面流从 `<start-state>` 节点开始执行。当执行到 `<end-state>` 节点时，内部页面流执行完毕，同时外部的页面流会以 `<process-state>` 定义的跳转恢复执行。

7.3. Seam中的业务流程管理

一个业务流程由一系列定义良好的任务组成，这些任务必须由用户或是软件系统遵照一系列定义良好的规则来完成，这些规则规定了 谁 可以执行任务，什么时候 应该执行。Seam对jBPM的整合使得给用户显示任务列表以及使用户管理他们的任务变得简单。Seam还使应用程序在 `BUSINESS_PROCESS` 的上下文中存储与业务流程相关的状态，通过jBPM变量来持久化这个状态。

一个简单的业务流程定义看起来跟页面流定义非常相似（是同一种类型的东西），不同的是用 `<task-node>` 节点替换了 `<page>` 节点。在一个长运行期的业务流程中，等待状态表示系统正在等待用户登录并执行任务。

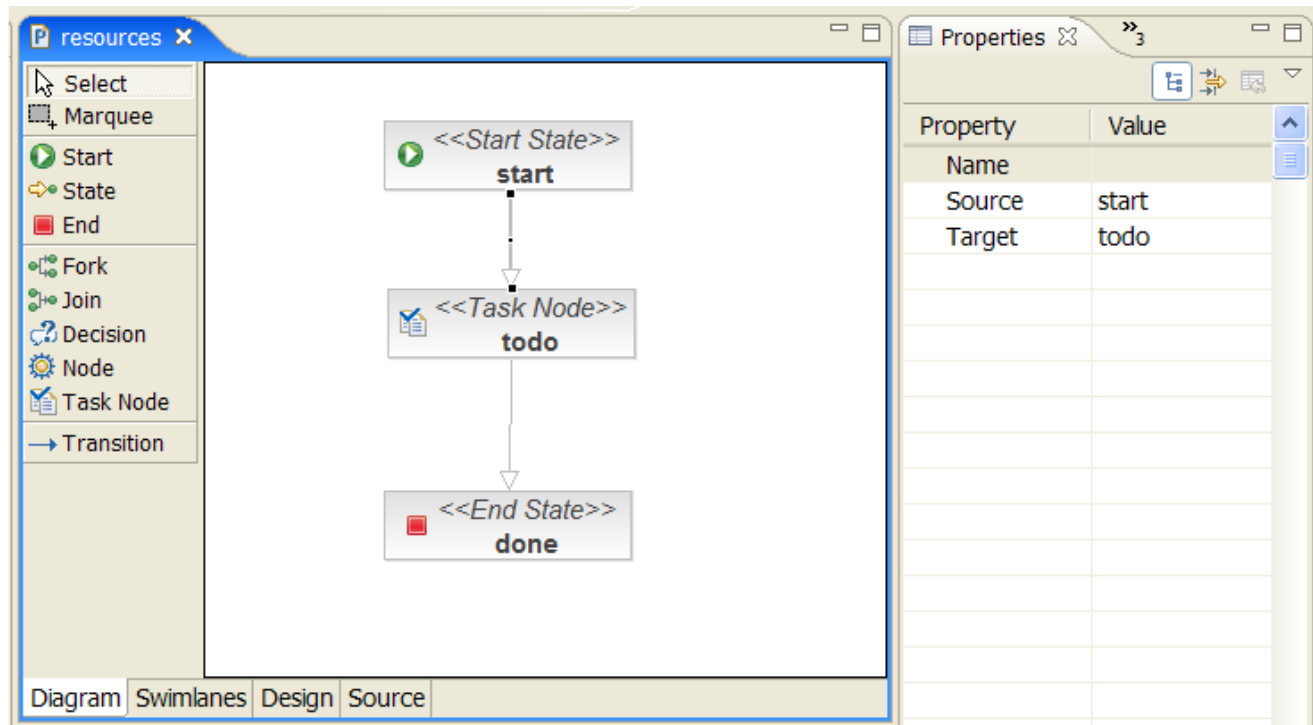
```
<process-definition name="todo">

  <start-state name="start">
    <transition to="todo"/>
  </start-state>

  <task-node name="todo">
    <task name="todo" description="#{todoList.description}">
      <assignment actor-id="#{actor.id}"/>
    </task>
    <transition to="done"/>
  </task-node>

  <end-state name="done"/>

</process-definition>
```



我们很有可能在项目中同时使用jPDL业务流程定义和jPDL页面流程定义。如果这样，他们二者的关系是：一个业务流程中的 `<task>` 对应一个完整的页面流 `<pageflow-definition>`。

7.4. 使用jPDL业务流程定义

7.4.1. 安装流程定义

我们需要安装jBPM，并且告诉它到哪里可以找到业务流程定义文件：

```
<bpm:jbpm>
  <bpm:process-definitions>
    <value>todo.jpdl.xml</value>
  </bpm:process-definitions>
</bpm:jbpm>
```

7.4.2. 初始化Actor id

我们总是需要知道当前的登录用户。jBPM使用 actor id 和 group actor id “识别”用户。我们使用Seam内置的 actor 组件指定当前用户的id。

```
@In Actor actor;

public String login() {
  ...
  actor.setId( user.getUserName() );
  actor.getGroupActorIds().addAll( user.getGroupNames() );
  ...
}
```

7.4.3. 启动一个业务流程

使用 @CreateProcess 注解来启动一个业务流程实例。

```
@CreateProcess(definition="todo")
public void createTodo() { ... }
```

也可用使用pages.xml来启动一个业务流程。

```
<page>
  <create-process definition="todo" />
</page>
```

7.4.4. 任务分配

当一个流程执行到一个任务节点时，会创建任务实例。这些任务实例必须分配给用户或是用户组。我们可以手动编码指定actor id或是委托给一个Seam组件。

```
<task name="todo" description="{todoList.description}">
  <assignment actor-id="{actor.id}" />
</task>
```

在这里例子中，我们简单的将任务分配给当前用户。也可以将任务分配给一批用户（pool actor

) :

```
<task name="todo" description="#{todoList.description}">
  <assignment pooled-actors="employees"/>
</task>
```

7.4.5. 任务列表

几个内置的Seam组件使得显示任务列表变得简单。pooledTaskInstanceList 是一个汇集任务集合，用户可以把这些任务分配给他们自己。

```
<h:dataTable value="#{pooledTaskInstanceList}" var="task">
  <h:column>
    <f:facet name="header">Description</f:facet>
    <h:outputText value="#{task.description}"/>
  </h:column>
  <h:column>
    <s:link action="#{pooledTask.assignToCurrentActor}" value="Assign" taskInstance="#{task}"/>
  </h:column>
</h:dataTable>
```

请注意，我们可以使用普通的JSF标签 <h:commandLink> 来替代 <s:link> 标签：

```
<h:commandLink action="#{pooledTask.assignToCurrentActor}">
  <f:param name="taskId" value="#{task.id}"/>
</h:commandLink>
```

pooledTask 组件是一个内置组件，它简单把任务分配给当前用户。

taskInstanceListForType组件包含一种特殊类型的任务，这些任务分配给当前用户：

```
<h:dataTable value="#{taskInstanceListForType['todo']}" var="task">
  <h:column>
    <f:facet name="header">Description</f:facet>
    <h:outputText value="#{task.description}"/>
  </h:column>
  <h:column>
    <s:link action="#{todoList.start}" value="Start Work" taskInstance="#{task}"/>
  </h:column>
</h:dataTable>
```

7.4.6. 执行一个任务

我们在监听方法上使用 @StartTask 或者 @BeginTask 注解来执行一个任务。

```
@StartTask
public String start() { ... }
```

我们也可以使用pages.xml来执行一个任务。

```
<page>
  <start-task />
</page>
```


这些注解启动一种特殊类型的对话，这个对话在整个业务流程中具有意义。通过该对话可以访问保存在业务流程上下文中的状态。

如果我们使用 `@EndTask` 注解来结束该对话，Seam会发出信号完成该任务。

```
@EndTask(transition="completed")
public String completed() { ... }
```

作为选择，我们还可以使用pages.xml

```
<page>
  <end-task transition="completed" />
</page>
```

也可以使用EL在pages.xml指定transition。

此时，jBPM接受指令并且继续执行业务流程定义。（在更加复杂的流程中，在流程向下执行之前，可能还需要完成其他一些任务。）

若想对jBPM处理复杂业务流程的高级特性有一个更加彻底的认识，请参阅jBPM的参考文档。

第 8 章 Seam和对象/关系映射

Seam给两个最流行的Java持久化架构：Hibernate3和由EJB 3.0引入的Java Persistence API提供了广泛支持。Seam独有的状态管理架构允许任意web应用框架与大多数成熟的ORM进行集成。

8.1. 简介

Seam是从Hibernate团队试图生成典型的无状态Java应用架构的挫折中成长起来的。上一代Java应用程序的无状态特性让Hibernate团队饱受挫折，Seam吸取了他们的经验。Seam的状态管理架构最早是用来解决持久化冲突相关问题的，特别是乐观事务处理相关的问题。可扩展的在线应用经常使用乐观事务。一个原子(database/JTA)级的事务不应该跨用户交互，除非系统设计时就是只支撑很少量的并发客户端。但几乎所有涉及到的工作都是先将数据展现给用户，没多久后更新这个数据。所以Hibernate是依据支持一种跨乐观事务的持久化上下文的思想设计的。

不幸的是这个先于Seam和EJB3.0出现的所谓“无状态”架构并不对乐观事务进行支持。而相反，这些架构提供对于原子事务级的持久化上下文的支持。这当然给用户带来了很大麻烦，这也是用户抱怨排名第一的Hibernate的 `LazyInitializationException` 问题的原因。我们需要的是在应用层构建对于乐观事务的支持。

EJB3.0认识到了此问题，并且也引入了有状态组件（有状态会话bean）的思想，它使用一个扩展持久化上下文来跟踪组件的生命周期。这是该问题的部分解决方案（对它自身而言也是一个有用的构想），然而还有两个问题：

- 有状态会话bean的生命周期必须在Web层通过代码手动管理（这是个麻烦的问题，而且实践起来比听上去更复杂）。
- 在同一个乐观事务的不同有状态组件间，传播持久化上下文是可行的，但很困难。

Seam通过提供对话(Conversation)和对话期间的有状态Session Bean组件来解决第一个问题（大多数会话实际上在数据层支持乐观事务）。这对于很多不需要传递持久化上下文的简单应用（比如Seam的订阅演示程序）已经足够了。对于更复杂的在每一个对话中的有很多松耦合组件的应用来说，组件间传播持久化上下文就成为一个重要的问题了。所以Seam扩展了EJB 3.0的持久化上下文管理模型，以此来提供对话作用域的扩展持久化上下文。

8.2. Seam管理的事务

EJB会话Bean有声明式事务管理功能。当Bean被调用时，EJB容器能够透明地开始一个事务，在调用结束时关闭此事务。如果我们写了一个作为JSF动作监听器的会话Bean方法，我们就可以在一个事务内处理所有与此action相关的工作，并且当我们完成此动作处理时事务必须被提交或回滚。这是一个很棒的功能，在很多Seam应用程序中这是必需的。

但是，此方法还是有问题。Seam应用可能无法在对会话Bean的一次方法调用请求中完成所有的数据访问。

- 此请求可能由几个松耦合组件处理，Web层独立地调用每一个组件。在Seam中，Web层的一个请求对EJB组件发起几次甚至多次调用的现象是很常见的。

- 视图渲染可能需要延迟关联获取（lazy fetching of associations）。

每个请求的事务量越多，当我们的应用处理大量并发请求时越可能碰到原子和隔离问题。当然，所有的写操作要在一个事务中执行。

Hibernate用户开发了“Open Session in View”模式来解决该问题。在Hibernate社区，“Open Session in View”曾经非常重要，这是因为像Spring这样的框架使用了事务作用域持久化上下文。所以当未获得的关联被访问时渲染视图将引起 `LazyInitializationException` 异常。

这个模式通常作为一个跨越整个请求的事务来实现。此实现方式会有几个问题，其中最严重的是只有我们提交了事务才能确认它成功完成——但在“Open Session in View”的事务提交时，视图已经完全渲染了，甚至渲染好的应答可能已经刷新到客户端。我们怎样才能通知用户他们的事务已失败呢？

Seam在解决“Open Session in View”问题时，也解决了事务隔离和关联获取问题。该方案有两个部分：

- 使用使用已扩展持久化上下文，可以覆盖一个会话作用域而不是单个事务作用域。
- 每次请求使用两个事务：第一个从更新模型值的起始阶段到应用程序调用结束；第二个跨越渲染响应阶段。

下一节，我们将会告诉你如何安装一个会话作用域的持久化上下文。但首先我们需要你知道如何启用Seam事务管理。注意你可以脱离Seam的事务管理来使用会话作用域持久化上下文。当你不使用Seam管理的持久化上下文时，你也有很多使用Seam事务管理的理由。然而这两种功能被设计为一起使用的，一起使用时效果最好。

即使你使用EJB 3.0容器管理事务上下文，Seam事务管理也是很有用的。如果你在Java EE 5环境外使用Seam，或者在任何你想使用Seam管理的持久化上下文时，它同样很有用的。

8.2.1. 关闭Seam管理的事务

所有的JSF请求默认开启Seam事务管理。如果你想 关闭 该功能，你能在 `components.xml` 文件中做如下设置：

```
<core:init transaction-management-enabled="false"/>

<transaction:no-transaction />
```

8.2.2. 配置Seam事务管理器

Seam为事务的开始，提交，回滚，同步提供了一个事务管理抽象。默认情况下，Seam使用一个JTA事务组件，它同容器管理的EJB和编程式EJB事务集成。

Seam还为以下事务API提供事务组件：

```
javax.persistence.EntityTransaction
```

```
org.hibernate.Transaction
```

`org.springframework.transaction.PlatformTransactionManager`

- 向`components.xml`文件增加以下项来配置JPA `RESOURCE_LOCAL`事务，在配置文件中，`{entityManager}` 是 `persistence:managed-persistence-context` 组件的名称。（参考 Seam管理的持久化上下文。）

```
<transaction:entity-transaction entity-manager="{entityManager}" />
```

向你的`components.xml`文件声明以下项来配置Hibernate管理的事务，在配置文件中，`{hibernateSession}` 是项目中 `persistence:managed-hibernate-session` 组件的名称。（参考Seam管理的持久化上下文）

```
<transaction:hibernate-transaction session="{hibernateSession}" />
```

在`components.xml`中声明以下内容来显式地关闭Seam管理的事务：

```
<transaction:no-transaction />
```

参考 使用Spring `PlatformTransactionManagement` 来配置Spring管理的事务。

8.2.3. 事务同步

事务同步为各种各样的事务相关事件提供了回调功能，例如 `beforeCompletion()` 和 `afterCompletion()` 事件。默认情况下，Seam使用它自己的事务同步组件，一个事务被提交时，需要显式地使用Seam事务组件以确保同步回调被正确执行。如果在Java EE 5环境里，应该在`components.xml`文件中声明 `<transaction:ejb-transaction/>`，从而保证容器在Seam的可预见范围外提交事务时，Seam同步回调被正确调用。

8.3. Seam管理的持久化上下文

如果你是在Java EE 5环境外使用Seam，你不能依靠容器来为你管理持久化上下文生命周期。即使在Java EE 5 环境中，你可能有一个很多松耦合组件在会话作用域内相互协作的复杂应用，这种情况下你可能发现在组件间传递持久化上下文既困难又容易出错。

在任何一种情况下，你都需要在你的组件中使用一个 受管持久化上下文（在JPA中）或者一个 受管会话（Hibernate中）。一个Seam管理的持久化上下文是在会话上下文中管理一个 `EntityManager` 实例或者 `Session` 实例的内置Seam组件。你可以使用 `@In` 注入它。

Seam管理的持久化上下文在集群环境中尤其有效。EJB 3.0规范中不允许容器使用容器管理的扩展持久化上下文，Seam能够对此进行优化。Seam支持扩展持久化上下文的透明故障恢复，而无需在节点间复制持久化上下文状态。（我们希望在EJB规范的下个版本中修复此漏洞。）

8.3.1. 在Seam管理的持久化上下文中使用JPA

配置一个Seam管理的持久化上下文很简单，在 `components.xml` 中加上：

```
<persistence:managed-persistence-context name="bookingDatabase"
    auto-create="true"
```

```
persistence-unit-jndi-name="java:/EntityManagerFactories/bookingData"/>
```

这个配置创建了一个名为 `bookingDatabase` 的对话作用范围Seam组件，它管理给持久化单元（`EntityManagerFactory` 实例）使用的 `EntityManager` 实例的生命周期。其JNDI名为 `java:/EntityManagerFactories/bookingData`。

当然，你需要确认已经在JNDI中绑定了 `EntityManagerFactory`。在JBoss中，你能在 `persistence.xml` 中通过增加以下属性设置来绑定它。

```
<property name="jboss.entity.manager.factory.jndi.name"
  value="java:/EntityManagerFactories/bookingData"/>
```

现在我们能以下方式来注入 `EntityManager` 了：

```
@In EntityManager bookingDatabase;
```

如果你正在使用EJB3，并且你的类或者方法上加了 `@TransactionAttribute(REQUIRES_NEW)`，那么事务和持久化上下文不应该被传播到这个对象的方法调用上。但是Seam管理的持久化上下文会被传播到会话内的所有组件上，它也会被传播到标有`REQUIRES_NEW`的方法上。因此，如果你标记了一个`REQUIRES_NEW`方法，那么就应该用`@PersistenceContext`来访问实体管理器。

8.3.2. 使用Seam管理的Hibernate会话

Seam管理的Hibernate sessions和前者很相似。在 `components.xml` 中加入如下内容：

```
<persistence:hibernate-session-factory name="hibernateSessionFactory"/>
<persistence:managed-hibernate-session name="bookingDatabase"
  auto-create="true"
  session-factory-jndi-name="java:/bookingSessionFactory"/>
```

`java:/bookingSessionFactory` 是在 `hibernate.cfg.xml` 中配置的会话工厂名。

```
<session-factory name="java:/bookingSessionFactory">
  <property name="transaction.flush_before_completion">true</property>
  <property name="connection.release_mode">after_statement</property>
  <property name="transaction.manager_lookup_class">org.hibernate.transaction.JBossTransactionManagerLookup</property>
  <property name="transaction.factory_class">org.hibernate.transaction.JTATransactionFactory</property>
  <property name="connection.datasource">java:/bookingDatasource</property>
  ...
</session-factory>
```

注意：Seam并没有清空此会话，所以你应启用 `hibernate.transaction.flush_before_completion`，确保在JTA事务提交前会话被自动地清空缓存。

现在我们可以用以下代码向JavaBean组件中注入一个受管Hibernate Session了：

```
@In Session bookingDatabase;
```

8.3.3. Seam管理的持久化上下文和原子会话

会话期间的持久化上下文让你能编写跨越多个服务器请求的乐观事务，而且无需使用`merge()`操作，也不需要每次请求开始时重载数据，也不需要处理 `LazyInitializationException` 异常或 `NonUniqueObjectException` 异常。

对任何乐观事务管理来说，事务隔离性和一致性可以使用乐观锁来获得。幸运的是，Hibernate 和EJB 3.0都通过使用 `@Version` 注解来使用乐观锁。

默认情况下，持久化上下文在每个事务结束时清空缓存（与数据库同步）。有时这是我们期望的方式。但经常我们希望所有的改变在内存中保存，且只有在会话成功结束时写回数据库。这对于真正的原子会话来说是允许的。作为非JBoss，非Sun，非Sybase的EJB 3.0专家组成员的愚蠢和短视决策的结果，使用EJB 3.0持久化还不能简单、有效和方便的实现原子会话。但是Hibernate通过扩展规范定义的 `FlushModeType` 提供了这个功能，我们也期望其它的厂商能尽快提供类似扩展。

Seam允许你在开始会话时指定`FlushModeType.MANUAL`参数。目前，只有Hibernate作为持久化底层提供者时它才能正常工作，但是我们计划支持其它同类计算机厂商扩展。

```
@In EntityManager em; //a Seam-managed persistence context

@Begin(flushMode=MANUAL)
public void beginClaimWizard() {
    claim = em.find(Claim.class, claimId);
}
```

现在 `claim` 对象仍然在其余会话中被持久化上下文管理。我们能对此`claim`进行一些修改：

```
public void addPartyToClaim() {
    Party party = ....;
    claim.addParty(party);
}
```

除非我们进行显式的强制提交，否则这些改变不会被写到数据库中。

```
@End
public void commitClaim() {
    em.flush();
}
```

当然你可以在`pages.xml`文件中将 `flushMode` 值设置为 `MANUAL`，例如在一个导航规则中写入：

```
<begin-conversation flush-mode="MANUAL" />
```

8.4. 使用JPA “代理（delegate）”

`EntityManager` 接口能让你通过使用 `getDelegate()` 方法访问某个厂商特定的API。很自然地，最让人感兴趣的厂商是Hibernate，其最强大的代理接口是 `org.hibernate.Session`。你不必再用别的东西，相信我，我没有任何偏见。

不管你使用Hibernate（明智！）还是其他的东西（受虐狂，或者说不太聪明），你当然想在你的Seam组件中经常使用代理。下面是一个方法：

```
@In EntityManager entityManager;
```

```
@Create
public void init() {
    ( (Session) entityManager.getDelegate() ).enableFilter("currentVersions");
}
```

但类型转化毫无疑问是Java语言中最难看的语法。因此大家都应尽可能避免使用它们。这有另一个获得代理的方法。首先将下列语句加到 `components.xml` 文件中：

```
<factory name="session"
    scope="STATELESS"
    auto-create="true"
    value="#{entityManager.delegate}"/>
```

现在我们可以直接注入此会话：

```
@In Session session;

@Create
public void init() {
    session.enableFilter("currentVersions");
}
```

8.5. 在EJB-QL/HQL中使用EL

无论你使用一个Seam管理的持久化上下文还是用 `@PersistenceContext` 注入一个容器管理的持久化上下文。Seam都代理 `EntityManager` 或者 `Session` 对象，这使你能在查询语句中安全有效地使用EL表达式。例如这个例子：

```
User user = em.createQuery("from User where username=#{user.username}")
    .getSingleResult();
```

和下面等价：

```
User user = em.createQuery("from User where username=:username")
    .setParameter("username", user.getUsername())
    .getSingleResult();
```

当然你不要写成下面这样：

```
User user = em.createQuery("from User where username=" + user.getUsername()) //BAD!
    .getSingleResult();
```

（这在遭受SQL注入攻击时会变得低效且不堪一击。）

8.6. 使用Hibernate过滤器

Hibernate最酷最独特的功能就是 过滤器（filter）。过滤器能让你提供一个数据库中数据的受限视图。但是应该有一个将过滤器合并到Seam应用中的简便方法，让它和Seam应用框架能工作得很好。

Seam管理的持久化上下文可以定义一系列的过滤器，这些过滤器在一个 `EntityManager` 或者 `Hibernate Session` 被创建时启用。（当然，它们只能在Hibernate做持久化底层时使用）

```
<persistence:filter name="regionFilter">
  <persistence:name>region</persistence:name>
  <persistence:parameters>
    <key>regionCode</key>
    <value>#{region.code}</value>
  </persistence:parameters>
</persistence:filter>

<persistence:filter name="currentFilter">
  <persistence:name>current</persistence:name>
  <persistence:parameters>
    <key>date</key>
    <value>#{currentDate}</value>
  </persistence:parameters>
</persistence:filter>

<persistence:managed-persistence-context name="personDatabase"
  persistence-unit-jndi-name="java:/EntityManagerFactories/personDatabase">
  <core:filters>
    <value>#{regionFilter}</value>
    <value>#{currentFilter}</value>
  </core:filters>
</persistence:managed-persistence-context>
```


第 9 章 Seam中的JSF表单验证

在普通JSF中，验证在视图中定义：

```
<h:form>
  <h:messages/>

  <div>
    Country:
    <h:inputText value="#{location.country}" required="true">
      <my:validateCountry/>
    </h:inputText>
  </div>

  <div>
    Zip code:
    <h:inputText value="#{location.zip}" required="true">
      <my:validateZip/>
    </h:inputText>
  </div>

  <h:commandButton/>
</h:form>
```

在实践中，这种方式常常违背了DRY原则，因为很多“validation”实际上依赖的约束是数据模型的一部分，而且也有很多方法引入数据库Schema定义。Seam使用Hibernate Validator来提供对基于model的约束支持。

让我们从定义 Location 类的约束开始：

```
public class Location {
    private String country;
    private String zip;

    @NotNull
    @Length(max=30)
    public String getCountry() { return country; }
    public void setCountry(String c) { country = c; }

    @NotNull
    @Length(max=6)
    @Pattern("^[0-9]*$")
    public String getZip() { return zip; }
    public void setZip(String z) { zip = z; }
}
```

好，这是一个不错的切入点，但在实践中使用自定义的约束可能比 Hibernate Validator 更优雅

:

```
public class Location {
    private String country;
    private String zip;

    @NotNull
    @Country
    public String getCountry() { return country; }
    public void setCountry(String c) { country = c; }

    @NotNull
    @ZipCode
    public String getZip() { return zip; }
    public void setZip(String z) { zip = z; }
}
```

```

public String getZip() { return zip; }
public void setZip(String z) { zip = z; }
}

```

无论我们使用哪种方式，都不需要在JSF页面中指定验证类型。我们可以使用 `<s:validate>` 来验证定义在model对象上的约束。

```

<h:form>
  <h:messages/>

  <div>
    Country:
    <h:inputText value="#{location.country}" required="true">
      <s:validate/>
    </h:inputText>
  </div>

  <div>
    Zip code:
    <h:inputText value="#{location.zip}" required="true">
      <s:validate/>
    </h:inputText>
  </div>

  <h:commandButton/>
</h:form>

```

注意：在model上指定 `@NotNull` 并不能在控制上省去 `required="true"`！这是因为JSF验证架构的限制。

这种方式在model中定义约束，然后在表现层中展示约束违例——这显然是一种更好的设计。

但是，这并不比我们之前的方法简便多少，所以让我们试试 `<s:validateAll>`：

```

<h:form>

  <h:messages/>

  <s:validateAll>

    <div>
      Country:
      <h:inputText value="#{location.country}" required="true"/>
    </div>

    <div>
      Zip code:
      <h:inputText value="#{location.zip}" required="true"/>
    </div>

    <h:commandButton/>

  </s:validateAll>

</h:form>

```

这个标签只是简单地给表单中的每个输入框增加 `<s:validate>` 标签。对于一个大的表单来说，这能够减少很多打字工作量！

现在我们需要做些事情来显示验证失败时的反馈消息。当前我们是在表单的上方显示所有消息。而我们真正想做的是在值域后面显示错误的提示信息（普通JSF也可以实现），高亮显示字段和标签（普通JSF无法实现），更好的是在字段后面显示一些图片（普通JSF同样无法实现）。我们还希望每个必须输入的字段所对应的标记前显示一个有颜色的*号。

这的确给表单中的每个字段增加了不少功能。我们不希望给表单中的每一个字段指定图片、消息和输入字段的高亮显示和布局。所以我们将通用布局定义在facelets模板中。

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:s="http://jboss.com/products/seam/taglib">

    <div>

        <s:label styleClass="{invalid?'error':''}">
            <ui:insert name="label"/>
            <s:span styleClass="required" rendered="{required}"/*</s:span>
        </s:label>

        <span class="{invalid?'error':''}">
            <h:graphicImage src="img/error.gif" rendered="{invalid}"/>
            <s:validateAll>
                <ui:insert/>
            </s:validateAll>
        </span>

        <s:message styleClass="error"/>

    </div>

</ui:composition>
```

我们可以通过 `<s:decorate>` 让每一个表单字段都使用这个模板。

```
<h:form>

    <h:messages globalOnly="true"/>

    <s:decorate template="edit.xhtml">
        <ui:define name="label">Country:</ui:define>
        <h:inputText value="{location.country}" required="true"/>
    </s:decorate>

    <s:decorate template="edit.xhtml">
        <ui:define name="label">Zip code:</ui:define>
        <h:inputText value="{location.zip}" required="true"/>
    </s:decorate>

    <h:commandButton/>

</h:form>
```

最后，我们可以使用 RichFaces Ajax 来在用户浏览表单时显示验证消息：

```
<h:form>

    <h:messages globalOnly="true"/>

    <s:decorate id="countryDecoration" template="edit.xhtml">
```

```

<ui:define name="label">Country:</ui:define>
<h:inputText value="#{location.country}" required="true">
    <a:support event="onblur" reRender="countryDecoration" bypassUpdates="true"/>
</h:inputText>
</s:decorate>

<s:decorate id="zipDecoration" template="edit.xhtml">
    <ui:define name="label">Zip code:</ui:define>
    <h:inputText value="#{location.zip}" required="true">
        <a:support event="onblur" reRender="zipDecoration" bypassUpdates="true"/>
    </h:inputText>
</s:decorate>

<h:commandButton/>
</h:form>

```

最好为页面上的重要控件定义显式的id，特别是当你希望用像 Selenium 这样的工具来进行UI自动化测试时。如果你没有提供显式的id，JSF会生成它们，但任何页面上的改动都会导致生成的值发生变化。

```

<h:form id="form">

    <h:messages globalOnly="true"/>

    <s:decorate id="countryDecoration" template="edit.xhtml">
        <ui:define name="label">Country:</ui:define>
        <h:inputText id="country" value="#{location.country}" required="true">
            <a:support event="onblur" reRender="countryDecoration" bypassUpdates="true"/>
        </h:inputText>
    </s:decorate>

    <s:decorate id="zipDecoration" template="edit.xhtml">
        <ui:define name="label">Zip code:</ui:define>
        <h:inputText id="zip" value="#{location.zip}" required="true">
            <a:support event="onblur" reRender="zipDecoration" bypassUpdates="true"/>
        </h:inputText>
    </s:decorate>

    <h:commandButton/>
</h:form>

```

但是，如果你想在验证失败时指定去显示不同的消息又该怎么做呢？可以使用Seam对Hibernate Validator的消息绑定：（同样也包括其中那些类似el表达式和独立视图消息绑定之类的各种好处）

```

public class Location {
    private String name;
    private String zip;

    // Getters and setters for name

    @NotNull
    @Length(max=6)
    @ZipCode(message="#{messages['location.zipCode.invalid']}")
    public String getZip() { return zip; }
    public void setZip(String z) { zip = z; }
}

```

```
location.zipCode.invalid = The zip code is not valid for #{location.name}
```



第 10 章 Groovy集成

JBoss Seam的一个特性就是具有RAD（快速应用开发）能力。虽然动态语言与RAD并非同一个意思，但它却是该领域内的一个十分有趣的工具。直到最近，选择一种动态语言就必须选择完全不同的开发平台（一个带有一系列API和运行环境的开发平台，如果你不想再使用旧的Java API，这可能是一种幸运，因为你可能不得不被迫使用平台提供的私有API）。[Groovy](#) 打破了这个约束，它是构建在Java虚拟机之上的动态语言。

现在，JBoss Seam通过静态语言和动态语言的无缝集成把动态语言世界和Java EE世界结合起来。JBoss Seam让开发人员在任务中使用最佳的工具，而不需要关心上下文切换。编写一个动态Seam组件和编写普通的Seam组件没什么两样，你使用相同的注释、相同的API，所有的一切都是相同的。

10.1. Groovy简介

Groovy是一个基于Java虚拟机的敏捷动态语言，它融合了从Python，Ruby和Smalltalk等语言中的诸多特性。Groovy的强大体现在两个方面：

- Groovy支持Java语法：Java代码就是Groovy代码，使学习曲线非常平滑，即学习的难度变得非常低。
- Groovy对象就是Java对象，Groovy类就是Java类：Groovy无缝集成所有已经存在的Java对象和类库。

TODO：再写一个Groovy语法的快速入门

10.2. 用Groovy编写Seam应用

这个没有什么可多说的，Groovy对象就是Java对象，你可以使用Groovy编写任何Seam组件或者Java类并部署它们。你也可以在一个应用中混合使用Groovy和Java类。

10.2.1. 编写Groovy组件

你可能已经注意到，Seam大量的使用注解（annotation）。要想Groovy支持注解必须确保其版本在1.1 Beta1以上。下面是在Seam应用中使用Groovy代码的例子。

10.2.1.1. 实体

```
@Entity
@Name("hotel")
class Hotel implements Serializable
{
    @Id @GeneratedValue
    Long id

    @Length(max=50) @NotNull
    String name

    @Length(max=100) @NotNull
    String address
}
```

```

    @Length(max=40) @NotNull
    String city

    @Length(min=2, max=10) @NotNull
    String state

    @Length(min=4, max=6) @NotNull
    String zip

    @Length(min=2, max=40) @NotNull
    String country

    @Column(precision=6, scale=2)
    BigDecimal price

    @Override
    String toString()
    {
        return "Hotel(${name},${address},${city},${zip})"
    }
}

```

Groovy本身就支持（getter/setter）方法特性，所以不用显示的编写冗长的getter和setter方法：在前面的例子中，Hotel类可以通过

```
<code>hotel.getCity()</code>
```

这样的语法被Java访问到，getter和setter方法是在Groovy编译时生成的。这样的语法让实体代码变得非常简洁。

Groovy1.1 Beta1暂时还不支持泛型（Generics）。一个负面影响是实体关系没有内置的类型信息。这就必须适当的使用 `@ToMany` 注解来代替简单的范型定义，就像

```
<code>Collection<Entity></code>
```

。出于同样的原因，你也不能从非常有用的 第 11 章 Seam应用程序框架 中受益。好消息是Groovy1.1正式版将支持范型（Groovy1.1 Beta2正在编写中）。

10.2.1.2. Seam组件

使用Groovy编写Seam组件与使用Java没有什么区别：使用注解将类标记为Seam组件。

```

@Scope(ScopeType.SESSION)
@Name("bookingList")
class BookingListAction implements Serializable
{
    @In EntityManager em
    @In User user
    @DataModel List<Booking> bookings
    @DataModelSelection Booking booking
    @Logger Log log

    @Factory public void getBookings()
    {
        bookings = em.createQuery('''
            select b from Booking b
            where b.user.username = :username
            order by b.checkinDate''')
            .setParameter("username", user.username)
            .getResultList()
    }

    public void cancel()
    {

```

```
log.info("Cancel booking: #{bookingList.booking.id} for #{user.username}")
Booking cancelled = em.find(Booking.class, booking.id)
if (cancelled != null) em.remove( cancelled )
getBookings()
FacesMessages.instance().add("Booking cancelled for confirmation number #{bookingList.booking.id}", new Object[0])
}
}
```

10.2.2. seam-gen

Seam gen透明地集成了Groovy。你可以在有seam-gen支持的项目中编写Groovy代码而不需要任何附加支持。当你编写一个Groovy实体，只需要把 .groovy 文件放在 src/model 目录中即可。同样的如果编写一个action，只要把 .groovy 文件放在 src/action 目录中就可以了。

10.3. 部署

部署Groovy类与部署Java类非常相像。（令人惊讶的是，不需要编写或者遵循某个3个字母的规范以支持多国语言组件框架）。

JBoss Seam拥有超越标准部署的能力，就是在开发时重新部署JavaBean Seam组件类而不必重启应用程序，这在开发 / 测试周期中节省了很多时间。在 .groovy 文件被部署时，Seam对GroovyBean Seam组件也提供了同样的热部署支持。

10.3.1. 部署Groovy代码

Groovy类就是Java类，和Java类有着同样的字节码。部署一个Groovy实体、Groovy Session Bean或者Groovy Seam组件，编译步骤是不可缺少的。一个通用的方法是使用 groovyc Ant任务。一旦编译，Groovy类和Java类不再有区别，应用服务器将它与Java类同样对待。这让Groovy与Java类无缝集成起来。

10.3.2. 开发时部署本地.groovy文件

JBoss Seam本身支持 .groovy 文件（不用编译）的增量热部署（必须是开发模式）。这让编辑/测试周期非常短。为了设置.groovy部署，请按照 第 2.7 节 “Seam与增量热部署” 进行配置，然后部署你的Groovy代码（.groovy文件）到 WEB-INF/dev 目录下。不需要重启应用程序，GroovyBean组件将被增量启用（当然运行应用程序的服务器也不必重启）。

本地的.groovy文件的部署和其他的Seam热部署有同样的局限：

- 组件必须是JavaBean或GroovyBean。不能是EJB3 Bean
- 实体不能热部署
- 被热部署的组件对部署在 WEB-INF/dev 以外的任何类都是不可见的
- 必须用Seam Debug模式

10.3.3. seam-gen

Seam-gen透明地支持Groovy文件的部署和编译。包括本地 `.groovy` 文件在开发模式（未编译的）下的部署。如果你构建了一个WAR类型的seam-gen项目，在 `src/action` 目录下的Java和Groovy类会自动参与增量热部署。如果你在生产模式下，Groovy文件会在部署之前被编译。

你将在 `examples/groovybooking` 目录下的Booking Demo中找到一个支持增量热部署的完全用Groovy写的例子。

第 11 章 Seam应用程序框架

Seam通过编写带有注解的简单Java类来让创建应用程序的工作变得非常简单，不需扩展任何特定接口和父类。但常见的编程任务还能进一步简化，这是通过一组预先创建的组件进行的，它们能够由 `component.xml` 文件配置（最简单的情况）或者类扩展而实现复用。

在一个Web应用程序中使用Hibernate或者JPA进行基本的数据库操作时，Seam Application Framework（Seam应用程序框架）能够减少你需要书写的代码量。

我们需要强调的是，这个框架非常的简单，只是少量的易于理解和扩展的简单类。“魔力”来自于Seam自身——即使没有用这个框架来创建任何Seam应用程序的时候，你也同样用到这一“魔力”。

11.1. 简介

有两种不同的方法使用Seam Application Framework所提供的组件。第一种方法是像处理其他种类的Seam内置组件一样，在 `components.xml` 中安装和配置组件的实例。举例来说，下列 `components.xml` 中的片段安装了一个能够为 `Person` 实体执行基本的CRUD（创建(Create)、读取(Retrieve)、更新(Update)和删除>Delete)）操作的组件：

```
<framework:entity-home name="personHome"
    entity-class="eg.Person"
    entity-manager="{personDatabase}">
    <framework:id>#{param.personId}</framework:id>
</framework:entity-home>
```

如果上面的代码按你的口味来说太像“用XML编程”，你可以改为使用扩展：

```
@Stateful
@Name("personHome")
public class PersonHome extends EntityHome<Person> implements LocalPersonHome {
    @RequestParameter String personId;
    @In EntityManager personDatabase;

    public Object getId() { return personId; }
    public EntityManager getEntityManager() { return personDatabase; }
}
```

第二种方法有一个很大的优点：你能够方便地添加额外的功能，覆盖内置的功能（框架的类都精心设计以便于扩展和定制）。

第二个优点是：如果你喜欢的话，你的类可以有状态会话Bean（这不是必须的，也可以是普通的JavaBean组件，如果你喜欢的话）。如果你正在使用JBoss AS，你需要使用4.2.2.GA或更高的版本。

目前，Seam应用框架提供了四个内置的组件：用于CRUD操作的 `EntityHome` 和 `HibernateEntityHome` 以及用于查询的 `EntityQuery` 和 `HibernateEntityQuery`。

你得编写Home和Query组件，它们能在session、event或conversation作用范围中运行，至于选择哪个scope取决于你所希望在你的应用程序中使用的状态模型。

Seam应用框架仅在Seam管理的持久化上下文中工作。默认情况下，这些组件会寻找一个叫做

entityManager 的持久化上下文。

11.2. Home对象

Home对象对特定的实体类提供持久化操作，假设我们有个可靠的 Person 类：

```
@Entity
public class Person {
    @Id private Long id;
    private String firstName;
    private String lastName;
    private Country nationality;

    //getters and setters...
}
```

我们可以通过配置定义一个 personHome 组件：

```
<framework:entity-home name="personHome" entity-class="eg.Person" />
```

也可以通过类的扩展

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {}
```

Home对象提供了如下的操作：persist()、remove()、update() 和 getInstance()。在你能够调用 remove() 或 update() 操作之前，你必须首先使用 setId() 方法定义你感兴趣的对象的标识符。

我们可以直接从一个JSF页面使用一个Home，如下例：

```
<h1>Create Person</h1>
<h:form>
    <div>First name: <h:inputText value="#{personHome.instance.firstName}" /></div>
    <div>Last name: <h:inputText value="#{personHome.instance.lastName}" /></div>
    <div>
        <h:commandButton value="Create Person" action="#{personHome.persist}" />
    </div>
</h:form>
```

通常，只用person 指明person漂亮得多，所以在 components.xml 中添加一行语句来实现。

```
<factory name="person"
    value="#{personHome.instance}" />

<framework:entity-home name="personHome"
    entity-class="eg.Person" />
```

（如果我们使用配置的方法。）或者，我们可以通过向 PersonHome 中添加一个 @Factory 方法来实现：

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {

    @Factory("person")
    public Person initPerson() { return getInstance(); }
```

```
}

```

（如果我们使用类扩展的方法） 这个修改使我们的JSF页面简化如下：

```
<h1>Create Person</h1>
<h:form>
  <div>First name: <h:inputText value="#{person.firstName}"/></div>
  <div>Last name: <h:inputText value="#{person.lastName}"/></div>
  <div>
    <h:commandButton value="Create Person" action="#{personHome.persist}"/>
  </div>
</h:form>

```

好，这就可以用来创建新的 Person 实体了。是的，这就是所需的全部代码！现在，如果我们想显示，更新，删除数据库中已经存在的 Person 实体，我们需要将实体标识符传递给 PersonHome。页面参数是一种非常好的实现方式：

```
<pages>
  <page view-id="/editPerson.jsp">
    <param name="personId" value="#{personHome.id}"/>
  </page>
</pages>

```

现在，我们可以向JSF页面中增加其他的操作：

```
<h1>
  <h:outputText rendered="#{!personHome.managed}" value="Create Person"/>
  <h:outputText rendered="#{personHome.managed}" value="Edit Person"/>
</h1>
<h:form>
  <div>First name: <h:inputText value="#{person.firstName}"/></div>
  <div>Last name: <h:inputText value="#{person.lastName}"/></div>
  <div>
    <h:commandButton value="Create Person" action="#{personHome.persist}" rendered="#{!personHome.managed}"/>
    <h:commandButton value="Update Person" action="#{personHome.update}" rendered="#{personHome.managed}"/>
    <h:commandButton value="Delete Person" action="#{personHome.remove}" rendered="#{personHome.managed}"/>
  </div>
</h:form>

```

当我们没有带任何请求参数链接到该页面时，会显示“Create Person”页面，当我们为 personId 这个请求参数设定一个值时，会显示“Edit Person”页面。

假设我们需要创建一些 Person 实体，并且初始化这些人的国籍。我们可以通过配置很轻松地完成：

```
<factory name="person"
  value="#{personHome.instance}"/>

<framework:entity-home name="personHome"
  entity-class="eg.Person"
  new-instance="#{newPerson}"/>

<component name="newPerson"
  class="eg.Person">
  <property name="nationality">#{country}</property>
</component>

```

也可以通过扩展类

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {

    @In Country country;

    @Factory("person")
    public Person initPerson() { return getInstance(); }

    protected Person createInstance() {
        return new Person(country);
    }
}
```

当然，Country 是一个被其它的Home对象管理的对象，比如说，CountryHome。

为了增加更多复杂的操作（联合管理等等），我们可以向 PersonHome 中添加方法。

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {

    @In Country country;

    @Factory("person")
    public Person initPerson() { return getInstance(); }

    protected Person createInstance() {
        return new Person(country);
    }

    public void migrate()
    {
        getInstance().setCountry(country);
        update();
    }
}
```

当事务成功之后（调用 persist()、update() 或 remove() 成功后），Home对象会发出一个 org.jboss.seam.afterTransactionSuccess 事件。通过监听这一事件，我们可以在底层实体改变后，刷新查询。如果我们只需要在特定的实体保存、修改或删除后刷新特定查询，我们可以监视 org.jboss.seam.afterTransactionSuccess.<name> 事件（<name> 是实体的名字）。

当一个操作成功时，Home对象可以自动地显示Faces信息，我们可以再一次通过配置来定制信息。

```
<factory name="person"
    value="#{personHome.instance}"/>

<framework:entity-home name="personHome"
    entity-class="eg.Person"
    new-instance="#{newPerson}">
    <framework:created-message>New person #{person.firstName} #{person.lastName} created</framework:created-message>
    <framework:deleted-message>Person #{person.firstName} #{person.lastName} deleted</framework:deleted-message>
    <framework:updated-message>Person #{person.firstName} #{person.lastName} updated</framework:updated-message>
</framework:entity-home>

<component name="newPerson"
    class="eg.Person">
    <property name="nationality">#{country}</property>
```

```
</component>
```

或者扩展：

```
@Name("personHome")
public class PersonHome extends EntityHome<Person> {

    @In Country country;

    @Factory("person")
    public Person initPerson() { return getInstance(); }

    protected Person createInstance() {
        return new Person(country);
    }

    protected String getCreatedMessage() { return "New person #{person.firstName} #{person.lastName} created"; }
    protected String getUpdatedMessage() { return "Person #{person.firstName} #{person.lastName} updated"; }
    protected String getDeletedMessage() { return "Person #{person.firstName} #{person.lastName} deleted"; }
}
```

但是指定信息最好的方法是把信息置于Seam所知的resource bundle中（在默认情况下，这个bundle叫做 messages ）。

```
Person_created=New person #{person.firstName} #{person.lastName} created
Person_deleted=Person #{person.firstName} #{person.lastName} deleted
Person_updated=Person #{person.firstName} #{person.lastName} updated
```

这样方便进行国际化，从表现层的角度考虑也保持了代码和配置的整洁。

最后一步是使用 <s:validateAll> 和 <s:decorate> 向页面中添加验证功能，我会把这个留给你们自己去实现。

11.3. Query对象

如果我们需要数据库中所有 Person 实例的列表，我们可以使用Query对象，例如：

```
<framework:entity-query name="people"
    ejbql="select p from Person p"/>
```

我们可以从一个JSF页面中使用它：

```
<h1>List of people</h1>
<h:dataTable value="#{people.resultList}" var="person">
    <h:column>
        <s:link view="/editPerson.jsp" value="#{person.firstName} #{person.lastName}">
            <f:param name="personId" value="#{person.id}" />
        </s:link>
    </h:column>
</h:dataTable>
```

我们可能需要支持分页：

```
<framework:entity-query name="people"
```

```
ejbql="select p from Person p"
order="lastName"
max-results="20"/>
```

我们可以使用page参数来决定被显示的页面

```
<pages>
  <page view-id="/searchPerson.jsp">
    <param name="firstResult" value="#{people.firstResult}"/>
  </page>
</pages>
```

用于分页的JSF代码可能有点冗长，但仍然是便于管理的：

```
<h1>Search for people</h1>
<h:dataTable value="#{people.resultList}" var="person">
  <h:column>
    <s:link view="/editPerson.jsp" value="#{person.firstName} #{person.lastName}">
      <f:param name="personId" value="#{person.id}"/>
    </s:link>
  </h:column>
</h:dataTable>

<s:link view="/search.xhtml" rendered="#{people.previousExists}" value="First Page">
  <f:param name="firstResult" value="0"/>
</s:link>

<s:link view="/search.xhtml" rendered="#{people.previousExists}" value="Previous Page">
  <f:param name="firstResult" value="#{people.previousFirstResult}"/>
</s:link>

<s:link view="/search.xhtml" rendered="#{people.nextExists}" value="Next Page">
  <f:param name="firstResult" value="#{people.nextFirstResult}"/>
</s:link>

<s:link view="/search.xhtml" rendered="#{people.nextExists}" value="Last Page">
  <f:param name="firstResult" value="#{people.lastFirstResult}"/>
</s:link>
```

真实的搜索界面能够通过让用户输入一系列的可选的搜索标准来缩小返回的结果列表。Query对象通过让你指定可选的“约束”来支持这个重要的用例。

```
<component name="examplePerson" class="Person"/>

<framework:entity-query name="people"
  ejbql="select p from Person p"
  order="lastName"
  max-results="20">
  <framework:restrictions>
    <value>lower(firstName) like lower( concat("#{examplePerson.firstName}','') )</value>
    <value>lower(lastName) like lower( concat("#{examplePerson.lastName}','') )</value>
  </framework:restrictions>
</framework:entity-query>
```

注意“example”对象的使用。

```
<h1>Search for people</h1>
<h:form>
  <div>First name: <h:inputText value="#{examplePerson.firstName}"/></div>
  <div>Last name: <h:inputText value="#{examplePerson.lastName}"/></div>
```

```

<div><h:commandButton value="Search" action="/search.jsp"/></div>
</h:form>

<h:dataTable value="#{people.resultList}" var="person">
  <h:column>
    <s:link view="/editPerson.jsp" value="#{person.firstName} #{person.lastName}">
      <f:param name="personId" value="#{person.id}" />
    </s:link>
  </h:column>
</h:dataTable>

```

在底层实体发生改变后，可以通过监听 `org.jboss.seam.afterTransactionSuccess` 事件来刷新查询：

```

<event type="org.jboss.seam.afterTransactionSuccess">
  <action execute="#{people.refresh}" />
</event>

```

或者，在发生持久化、更新或者删除时，通过 `PersonHome` 来刷新查询：

```

<event type="org.jboss.seam.afterTransactionSuccess.Person">
  <action execute="#{people.refresh}" />
</event>

```

这个部分所有的例子都是通过配置来体现重用的，但是，对Query对象通过扩展来进行重用也是可行的。

11.4. Controller对象

Controller 类以及它的子类 `EntityController`，`HibernateEntityController` 和 `BusinessProcessController` 是Seam Application Framework的可选部分。这些类只是提供了一些访问常用内置组件及这些组件方法的便利手段，它们能够减少一些键盘输入量，也为探索Seam内置丰富功能的初学者提供了一个非常好的跳板。

例如，这就是Seam注册实例中的 `RegisterAction`：

```

@Stateless
@Name("register")
public class RegisterAction extends EntityController implements Register
{
    @In private User user;

    public String register()
    {
        List existing = createQuery("select u.username from User u where u.username=:username")
            .setParameter("username", user.getUsername())
            .getResultList();

        if ( existing.size()==0 )
        {
            persist(user);
            info("Registered new user #{user.username}");
            return "/registered.jspx";
        }
        else
        {
            addFacesMessage("User #{user.username} already exists");
        }
    }
}

```



```
        return null;
    }
}
```

正如你所看到的一样，这不是什么惊世骇俗的提高...

第 12 章 Seam和JBoss规则

Seam简化了在Seam组件或jBPM过程定义中对JBoss Rules (Drools) 规则库的调用。

12.1. 安装规则

第一步是使一个 `org.drools.RuleBase` 的实例在Seam的上下文变量中可用。基于测试目的，Seam提供了一个内置组件用来编译来自class搜索路径的一组静态规则。你可以通过 `components.xml` 文件安装此组件：

```
<drools:rule-base name="policyPricingRules">
  <drools:rule-files>
    <value>policyPricingRules</value>
  </drools:rule-files>
</drools:rule-base>
```

这个组件编译来自一组 `.dr1` 文件中的规则并在Seam的 `APPLICATION` 上下文中缓存一个 `org.drools.RuleBase` 实例。需要注意的是这和在规则驱动的应用程序中需要安装多个规则库很相似。

如果你想要使用Drools DSL，你还需要指定DSL定义：

```
<drools:rule-base name="policyPricingRules" dsl-file="policyPricing.dsl">
  <drools:rule-files>
    <value>policyPricingRules</value>
  </drools:rule-files>
</drools:rule-base>
```

在大多数规则驱动的应用程序中，规则需要是可被动态部署的，所以一个生产环境应用程序会需要用Drools规则代理来管理规则库。规则代理可以连接一个Drools规则服务器 (BRMS) 或者热部署来自本地文件仓库的规则包。规则代理管理的规则库也在 `components.xml` 中配置：

```
<drools:rule-agent name="insuranceRules"
  configurationFile="/WEB-INF/deployedrules.properties" />
```

属性文件包含有规则代理RulesAgent所特有的属性。这里是一个来自Drools范例发型中的配置文件例子。

```
newInstance=true
url=http://localhost:8080/drools-jbrms/org.drools.brms.JBRMS/package/org.acme.insurance/fmeyer
localCacheDir=/Users/fernandomeyer/projects/jbossrules/drools-examples/drools-examples-brms/cache
poll=30
name=insuranceconfig
```

绕过配置文件，直接配置组件的选项也是可行的。

```
<drools:rule-agent name="insuranceRules"
  url="http://localhost:8080/drools-jbrms/org.drools.brms.JBRMS/package/org.acme.insurance/fmeyer"
  local-cache-dir="/Users/fernandomeyer/projects/jbossrules/drools-examples/drools-examples-brms/cache"
  poll="30"
  configuration-name="insuranceconfig" />
```

接下来，我们需要使 `org.drools.WorkingMemory` 实例对每个对话都可用。（每个 `WorkingMemory` 累积与当前对话相关的fact。）

```
<drools:managed-working-memory name="policyPricingWorkingMemory" auto-create="true" rule-base="{policyPricingRules}" />
```

请注意，我们通过 `ruleBase` 配置属性给了 `policyPricingWorkingMemory` 一个指回规则库的引用。

12.2. 在Seam组件中使用规则

现在可以将我们的 `WorkingMemory` 注入进任意的Seam组件中了，进行判断并执行规则：

```
@In WorkingMemory policyPricingWorkingMemory;

@In Policy policy;
@In Customer customer;

public void pricePolicy() throws FactException
{
    policyPricingWorkingMemory.assertObject(policy);
    policyPricingWorkingMemory.assertObject(customer);
    policyPricingWorkingMemory.fireAllRules();
}
```

12.3. 在jBPM流程定义中使用规则

你甚至可以用一个规则库来充当jBPM动作处理器、决定处理器或者分配器——无论是在页面流或者业务流程定义中。

```
<decision name="approval">

    <handler class="org.jboss.seam.drools.DroolsDecisionHandler">
        <workingMemoryName>orderApprovalRulesWorkingMemory</workingMemoryName>
        <assertObjects>
            <element>#{customer}</element>
            <element>#{order}</element>
            <element>#{order.lineItems}</element>
        </assertObjects>
    </handler>

    <transition name="approved" to="ship">
        <action class="org.jboss.seam.drools.DroolsActionHandler">
            <workingMemoryName>shippingRulesWorkingMemory</workingMemoryName>
            <assertObjects>
                <element>#{customer}</element>
                <element>#{order}</element>
                <element>#{order.lineItems}</element>
            </assertObjects>
        </action>
    </transition>

    <transition name="rejected" to="cancelled"/>

</decision>
```

`<assertObjects>` 元素指定了用来返回要被作为fact设给 `WorkingMemory` 的对象或者对象集合的EL表达

式。

除此之外，Seam还支持在jBPM任务分配中使用Drools：

```
<task-node name="review">
  <task name="review" description="Review Order">
    <assignment handler="org.jboss.seam.drools.DroolsAssignmentHandler">
      <workingMemoryName>orderApprovalRulesWorkingMemory</workingMemoryName>
      <assertObjects>
        <element>#{actor}</element>
        <element>#{customer}</element>
        <element>#{order}</element>
        <element>#{order.lineItems}</element>
      </assertObjects>
    </assignment>
  </task>
  <transition name="rejected" to="cancelled"/>
  <transition name="approved" to="approved"/>
</task-node>
```

这些对象作为Drools global提供给规则，即jBPM Assignable 对象（assignable）和Seam Decision对象（decision）。处理决定的规则应该调用 `decision.setOutcome("result")` 来决定结果。执行分配的规则要调用 `Assignable` 设置参与者id。

```
package org.jboss.seam.examples.shop

import org.jboss.seam.drools.Decision

global Decision decision

rule "Approve Order For Loyal Customer"
when
  Customer( loyaltyStatus == "GOLD" )
  Order( totalAmount <= 10000 )
then
  decision.setOutcome("approved");
end
```

```
package org.jboss.seam.examples.shop

import org.jbpm.taskmgmt.exe.Assignable

global Assignable assignable

rule "Assign Review For Small Order"
when
  Order( totalAmount <= 100 )
then
  assignable.setPooledActors( new String[] { "reviewers" } );
end
```

第 13 章 安全

Seam Security API是个可选的Seam特性，它为了保护您的Seam项目中的领域和页面资源提供验证和授权特性。

13.1. 概述

Seam Security提供两种不同的操作模式：

- 简化模式 - 这个模式支持验证服务和简单的基于角色的安全性检查。
- 高级模式 - 这个模式支持简化模式的所有特性，还利用JBoss Rules提供基于规则的安全性检查。

13.1.1. 哪种模式更适合我的应用程序呢？

这完全取决于应用程序的需求。如果你的安全性需求不高，例如，如果只希望对登录的用户或者属于某个特定角色的用户限制某些页面和动作，那么简化模式可能就足够了。这个模式的好处在于它是一种更简化的配置，要包括的库明显更少，占用的内存空间（memory footprint）也更小。

另一方面，如果应用程序需要根据上下文或者复杂的业务规则进行安全性检查，那就需要高级模式提供的特性了。

13.2. 需求

如果使用Seam Security的高级模式特性，下列jar文件就要配置成 `application.xml` 中的模块。如果你正在简化模式下使用Seam Security，则不需要这些了：

- `drools-compiler-4.0.0.MR2.jar`
- `drools-core-4.0.0.MR2.jar`
- `janino-2.5.7.jar`
- `antlr-runtime-3.0.jar`
- `mvel14-1.2beta16.jar`

对于基于Web的安全性来说，`jboss-seam-ui.jar` 也必须包括在应用程序的war文件中。

13.3. 取消安全

在某些场景下，可能需要取消Seam Security，例如在单元测试的过程中。可以通过调用静态方法 `Identity.setSecurityEnabled(false)` 来取消安全检查。这样一来，就可以阻止执行如下的任何安全检查：

- Entity Security
- Hibernate Security Interceptor
- Seam Security Interceptor
- Page restrictions

13.4. 验证

Seam Security提供的验证特性建立在JAAS（Java Authentication和Authorization Service）之上，给处理用户验证提供稳健的、非常容易配置的API。然而，对于并不复杂的验证需求，Seam提供了一种更加简化的验证方法，隐藏了JAAS的复杂性。

13.4.1. 配置

简化的验证方法使用一种内置的JAAS登录模块：SeamLoginModule，把验证委托给你项目中的一个Seam组件。这个登录模块已经在Seam内部配置为默认的应用策略的一部分，因此不需要任何额外的配置文件。它允许你利用你自己的应用程序提供的实体类编写验证方法。配置这个简化的验证形式需要在 components.xml 中配置 identity 组件。

```
<components xmlns="http://jboss.com/products/seam/components"
  xmlns:core="http://jboss.com/products/seam/core"
  xmlns:security="http://jboss.com/products/seam/security"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://jboss.com/products/seam/components http://jboss.com/products/seam/components-2.0.xsd
    http://jboss.com/products/seam/security http://jboss.com/products/seam/security-2.0.xsd">

  <security:identity authenticate-method="#{authenticator.authenticate}"/>

</components>
```

如果你希望使用高级的安全性特性，如基于规则的许可检查，你所要做的就是将Drools（JBoss Rules）jars包含在你的classpath中，并添加一些额外的配置，后面会讲到。

EL表达式 `#{authenticator.authenticate}` 是一种方法绑定，表示 `authenticator` 组件的 `authenticate` 方法将用来验证该用户。

13.4.2. 编写验证方法

在 components.xml 中给 identity 指定的 `authenticate-method` 属性，规定了哪种方法将被 SeamLoginModule用来验证用户。这个方法没有参数，并将要返回一个布尔值，表示验证是否成功。用户的用户名和密码可以分别地通过 `Identity.instance().getUsername()` 和 `Identity.instance().getPassword()` 获取，用户所属的任何角色都应该利用 `Identity.instance().addRole()` 分配。下面是JavaBean组件内部一个验证方法的完整实例：

```
@Name("authenticator")
public class Authenticator {
    @In EntityManager entityManager;

    public boolean authenticate() {
```

```
try
{
    User user = (User) entityManager.createQuery(
        "from User where username = :username and password = :password")
        .setParameter("username", Identity.instance().getUsername())
        .setParameter("password", Identity.instance().getPassword())
        .getSingleResult();

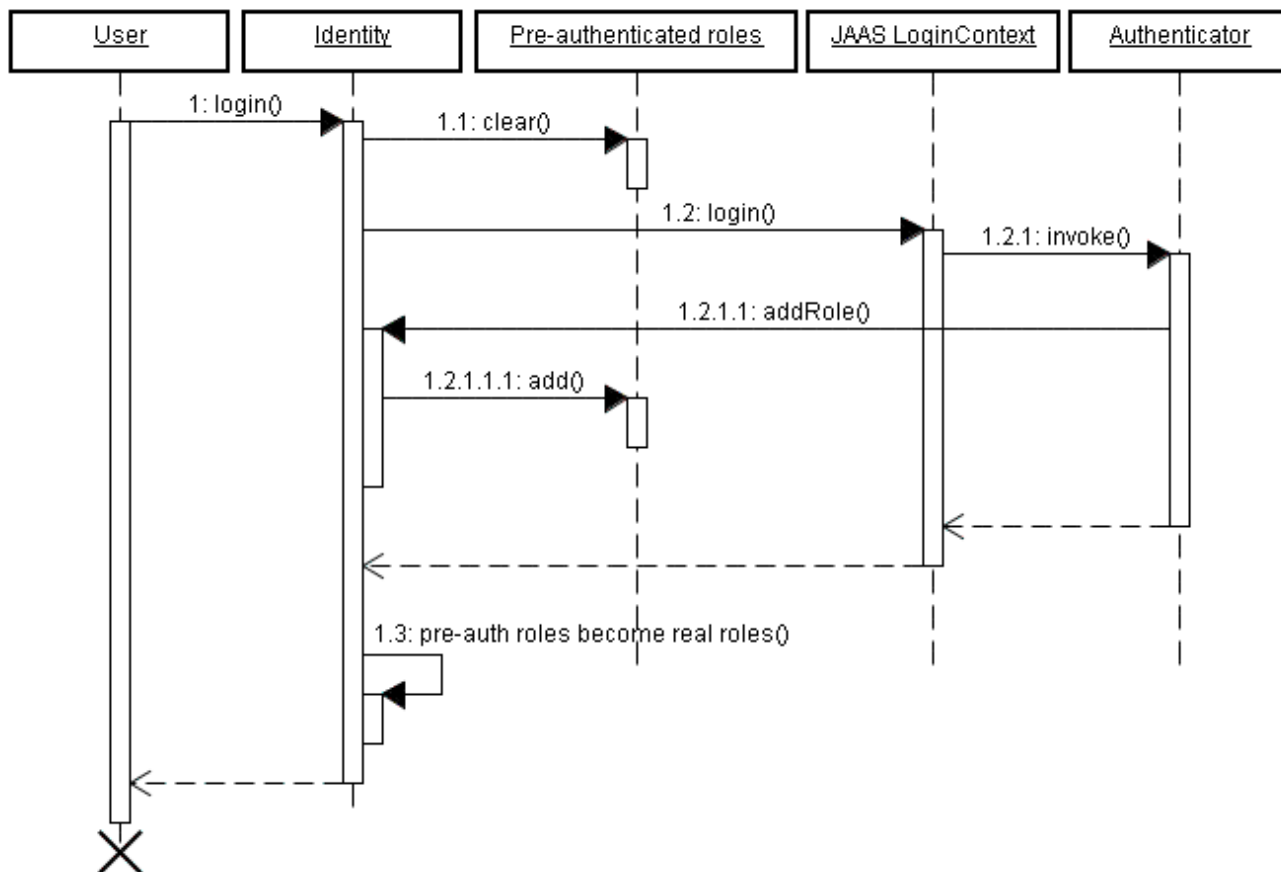
    if (user.getRoles() != null)
    {
        for (UserRole mr : user.getRoles())
            Identity.instance().addRole(mr.getName());
    }

    return true;
}
catch (NoResultException ex)
{
    return false;
}
}
```

在上面的例子中，User 和 UserRole 都是应用程序指定的实体Bean。roles 参数以用户所属的角色填充，它应该添加到 Set 作为文字型字符串值，如“admin”、“user”。在这个例子中，如果没有找到用户记录，并抛出一个 NoResultException，验证方法就返回false，表示验证失败。

13.4.2.1. Identity.addRole()

Identity.addRole() 方法表现的不同取决于当前会话是否已经验证过了。如果会话没有验证过，那么 addRole() 就只能在验证的过程中被调用。当被调用到这里时，角色名称被放进一个预验证角色的临时列表中。一旦验证成功，预验证角色就变成“真实的”角色了，并且调用 Identity.hasRole() 就可以返回true了。下面的序列图展示了预验证角色列表作为第一类对象，以更清楚地表明它如何适应在整个验证过程。



13.4.3. 编写登录表单

Identity 组件提供 `username` 和 `password` 属性，适合最常见的验证场景。这些属性可以直接绑定到登录表单上的用户名和密码字段。一旦设置了这些属性，调用 `identity.login()` 方法就可以验证使用所提供的证书的用户。下面是一个简单的登录表单的例子：

```

<div>
  <h:outputLabel for="name" value="Username"/>
  <h:inputText id="name" value="#{identity.username}"/>
</div>

<div>
  <h:outputLabel for="password" value="Password"/>
  <h:inputSecret id="password" value="#{identity.password}"/>
</div>

<div>
  <h:commandButton value="Login" action="#{identity.login}"/>
</div>

```

类似地，注销用户通过调用 `#{identity.logout}` 来完成。调用这个动作将清除当前被验证用户的安全性状态。

13.4.4. 简化配置 - 概述

因此归结起来，配置验证有三个简单的步骤：

- 在 `components.xml` 中配置一种验证方法。

- 编写一种验证方法。
- 编写一个登录表单，以便用户可以验证。

13.4.5. 处理安全异常

为了防止用户随安全错误而收到默认的错误页面，建议 `pages.xml` 配置为把安全错误重定向到一个更“漂亮”的页面。由Security API抛出的两种主要的异常类型是：

- `NotLoggedInException` - 如果用户试图在没有登录的情况下访问一个受限的动作或者页面，就会抛出这个异常。
- `AuthorizationException` - 只有当用户已经登录，并且试图访问他们没有必要权限的受限动作或者页面时，才会抛出这个异常。

在 `NotLoggedInException` 抛出的情况下，建议把用户限制到一个登录页面或者注册页面，以便登录。对于 `AuthorizationException`，把用户重定向到一个错误页面可能更合适。以下是 `pages.xml` 文件的一个例子，它把这两种安全异常都处理了：

```
<pages>

...

<exception class="org.jboss.seam.security.NotLoggedInException">
  <redirect view-id="/login.xhtml">
    <message>You must be logged in to perform this action 你必须登录执行这个动作</message>
  </redirect>
</exception>

<exception class="org.jboss.seam.security.AuthorizationException">
  <end-conversation/>
  <redirect view-id="/security_error.xhtml">
    <message>You do not have the necessary security privileges to perform this action. 你没有执行这个动作的必要权限。</message>
  </redirect>
</exception>

</pages>
```

大多数Web应用程序需要更复杂的登录重定向处理，因此Seam包含了处理这个问题的一些特殊功能。

13.4.6. 登录重定向

当未被验证的用户试图访问某个特定的视图（或者通配符id）时，可以让Seam把用户重定向到一个登录页面：

```
<pages login-view-id="/login.xhtml">

  <page view-id="/members/*" login-required="true"/>

  ...

</pages>
```

（这个不像上述的异常处理器那么简洁，但是可能要结合起来使用。）

用户登录以后，我们要自动返回到原来的地方（网址），以便可以重试要求登录之后才能进行的操作。如果把下列事件监听器添加到 `components.xml`，如果用户在没有登录的情况下去访问受限的页面，那么页面信息就会被记录下来。用户登录之后，就会被重定向到刚才的受限页面，并且把原来的请求参数也一起传过去。

```
<event type="org.jboss.seam.notLoggedIn">
  <action execute="#{redirect.captureCurrentView}" />
</event>

<event type="org.jboss.seam.postAuthenticate">
  <action execute="#{redirect.returnToCapturedView}" />
</event>
```

注意登录重定向被实现为一种对话范围的机制，因此不要在 `authenticate()` 方法中终止对话。

13.4.7. HTTP验证

虽然不建议使用，除非绝对必要，Seam提供HTTP Basic或HTTP Digest（RFC 2617）方法的验证。为使用表单的验证方式，`authentication-filter` 组件必须能够在`components.xml`中激活：

```
<web:authentication-filter url-pattern="*.seam" auth-type="basic"/>
```

为了激活Basic验证的过滤器，设置 `auth-type` 为 `basic` 或者对于Digest验证，设置 `auth-type` 为 `digest`。如果是使用Digest验证，`key` 和 `realm` 也必须进行设置：

```
<web:authentication-filter url-pattern="*.seam" auth-type="digest" key="AA3JK34aSDlkj" realm="My App"/>
```

`key` 可以是任何字符串值。`realm` 就是进行用户验证时所用的验证Realm的名称。

13.4.7.1. 编写Digest验证者

如果是使用Digest认证，你的鉴定者类需要继承抽象类 `org.jboss.seam.security.digest.digestauthenticator`，并使用 `validatePassword()` 方法，以根据Digest的要求来验证用户的纯文本密码。下面是一个例子：

```
public boolean authenticate()
{
    try
    {
        User user = (User) entityManager.createQuery(
            "from User where username = :username")
            .setParameter("username", identity.getUsername())
            .getSingleResult();

        return validatePassword(user.getPassword());
    }
    catch (NoResultException ex)
    {
    }
}
```

```

        return false;
    }
}

```

13.4.8. 高级验证特性

本节探讨Security API提供的部分高级特性，用来满足更复杂的安全需求。

13.4.8.1. 使用容器的JAAS配置

如果你宁可不用Seam Security API提供的简化的JAAS配置，可以委托给默认的系统JAAS配置，该配置是在 `components.xml` 中提供的一个 `jaasConfigName`。例如，如果你正在使用JBoss AS，并希望使用其它策略（使用JBoss AS提供的 `UsersRolesLoginModule` 登录模块），那么 `components.xml` 中的项看起来就像这样：

```

<security:identity authenticate-method="#{authenticator.authenticate}"
                    jaas-config-name="other"/>

```

13.5. 错误消息

Security API给各种安全相关的事件生成许多默认的展现消息。下表列出了可以用来覆盖这些消息的Key，它们在 `message.properties` 资源文件中指定。为了禁止消息，只要在资源文件里给相应的Key赋予空值就行了。

表 13.1. 安全消息Key

<code>org.jboss.seam.loginSuccessful</code>	这个消息在用户通过Security API成功登录时生成。
<code>org.jboss.seam.loginFailed</code>	这个消息在登录过程失败时生成。失败是由于用户提供了错误的用户名或者密码，或者由于其他问题造成的。
<code>org.jboss.seam.NotLoggedIn</code>	这个消息在用户试图执行一个动作，或者访问一个需要安全检查的页面，并且用户目前未被验证时生成。

13.6. 授权

Seam Security API提供了大量授权特性，用来保护对组件、组件方法和页面的访问。本节阐述这每一种授权特性。要注意的一件重要的事是，如果你希望使用任何高级特性（如基于规则的许可），那么你的 `components.xml` 文件就必须配置成支持该特性 - 请见前面的配置小节。

13.6.1. 核心概念

Seam Security API提供的每种授权机制，都构建在用户被授与了角色和 / 或许可的概念之上。

角色是用户的一个 群组，或者一种 类型，该用户可能已经被授与某种特权，用来在应用程序内部执行一个或者更多的特定操作。许可是执行单个指定动作的一种（有时是一次性的）特权。只用许可而不用其它任何东西来构建应用程序是完全可能的，然而角色在授与用户群组特权时更灵活方便。

角色很简单，只由一个名称组成，如 “admin”、“user”、“customer” 等等。许可由一个名称和一个动作组成，在这个文档中以 `name:action` 的形式表示，例如 `customer:delete`，或者 `customer:insert`。

13.6.2. 保护组件

我们从检验授权的最简单的形式开始：组件安全，从 `@Restrict` 注解开始。

13.6.2.1. @Restrict注解

Seam组件可以利用 `@Restrict` 注解在方法或者类级上得到保护。如果方法和声明它的类都通过 `@Restrict` 注解，方法限制将优先（类限制是不起作用）。如果方法调用在安全检查中失败，那么根据对 `Identity.checkRestriction()` 的约定就会抛出一个异常（请见行内限制）。组件类自身上的 `@Restrict` 相当于把 `@Restrict` 添加到了它的每一个方法上。

空的 `@Restrict` 意味着 `componentName:methodName` 的一个许可检查。以下面的组件方法为例：

```
@Name("account")
public class AccountAction {
    @Restrict public void delete() {
        ...
    }
}
```

在这个例子中，调用 `delete()` 方法需要的隐含许可是 `account:delete`。还有一种方式，就是写 `@Restrict("#{s:hasPermission('account','delete',null)})`。现在来看另一个例子：

```
@Restrict @Name("account")
public class AccountAction {
    public void insert() {
        ...
    }
    @Restrict("#{s:hasRole('admin')}")
    public void delete() {
        ...
    }
}
```

这一次，组件类本身通过 `@Restrict` 注解。这意味着任何没有覆盖 `@Restrict` 注解的方法都需要一个隐式的许可检查。在这个例子中，`insert()` 方法需要 `account:insert` 的一个许可，而 `delete()` 方法则要求用户必须是 `admin` 角色的一员。

在进一步探讨之前，先看看在前面的例子中见过的 `#{s:hasRole()}` 表达式。`s:hasRole` 和 `s:hasPermission` 都是EL方法，它们委托给 `Identity` 类相应的具名方法。这些函数可以在所有Security API的任何EL表达式中使用。

作为一个EL表达式，`@Restrict` 注解的值可以引用Seam上下文中存在的任何对象。这在给特定的对象实例执行许可检查时极为有用。看看这个例子：

```
@Name("account")
```

```
public class AccountAction {
    @In Account selectedAccount;
    @Restrict("#{s:hasPermission('account','modify',selectedAccount)}")
    public void modify() {
        selectedAccount.modify();
    }
}
```

这个例子中最值得关注的东西是，对在 `hasPermission()` 函数调用中见到的 `selectedAccount` 的引用。这个变量的值将从 Seam 上下文内部查找，并传递到 `Identity` 中的 `hasPermission()` 方法，在这个例子中，它随后可以确定用户是否具有修改指定 `Account` 对象所需的许可。

13.6.2.2. 行内限制

有时候，可能希望在代码中执行安全检查，而不用 `@Restrict` 注解。在这种情况下，只要用 `Identity.checkRestriction()` 来计算安全表达式，像这样：

```
public void deleteCustomer() {
    Identity.instance().checkRestriction("#{s:hasPermission('customer','delete',selectedCustomer)}");
}
```

如果指定的表达式没有取值为 `true`，或者

- 如果用户没有登录，就抛出 `NotLoggedInException` 异常，或者
- 如果用户登录了，就抛出 `AuthorizationException` 异常。

直接在 Java 代码中调用 `hasRole()` 和 `hasPermission()` 方法也是可能的：

```
if (!Identity.instance().hasRole("admin"))
    throw new AuthorizationException("Must be admin to perform this action");

if (!Identity.instance().hasPermission("customer", "create", null))
    throw new AuthorizationException("You may not create new customers");
```

13.6.3. 用户界面中的安全

设计优良的用户界面的一种表现是，不会向用户展现他们没有必要的权限进行使用的选项。Seam Security 允许条件性的渲染：1) 一个页面的几个部分，或者 2) 独立的控制，根据用户的权限，使用与给组件安全的完全相同的 EL 表达式。

我们来看看界面安全的一些例子。首先，我们假设有一个登录表单，它应该只在用户还没有登录的情况下才被渲染。我们可以利用 `identity.isLoggedIn()` 属性像这样编写：

```
<h:form class="loginForm" rendered="#{not identity.loggedIn}">
```

如果用户没有登录，登录表单就渲染 – 目前为止这还是非常直白易懂。现在假设页面上有一个菜单，包含一些应该只对 `manager` 角色中的用户可用的操作。下面是一种编写方式：

```
<h:outputLink action="#{reports.listManagerReports}" rendered="#{s:hasRole('manager')}">
    Manager Reports
</h:outputLink>
```

这也很容易理解。如果用户不是 `manager` 角色的一员，那么 `outputLink` 就不渲染。 `rendered` 属性本身一般来说可以用在控制，或者周围的 `<s:div>` 或者 `<s:span>` 控制上。

现在探讨一些更复杂的东西。我们假设你在页面上有一个 `h:dataTable` 控制，该页面罗列了可能希望或者不希望根据用户的权限，对其渲染动作链接的记录。 `s:hasPermission` EL函数允许我们传进一个对象参数，它可以用来确定用户是否具有对该对象的必要许可。下面展现了带有受保护的链接的 `dataTable` 可能是什么样子：

```
<h:dataTable value="#{clients}" var="cl">
  <h:column>
    <f:facet name="header">Name</f:facet>
    #{cl.name}
  </h:column>
  <h:column>
    <f:facet name="header">City</f:facet>
    #{cl.city}
  </h:column>
  <h:column>
    <f:facet name="header">Action</f:facet>
    <s:link value="Modify Client" action="#{clientAction.modify}"
      rendered="#{s:hasPermission('client','modify',cl)"/>
    <s:link value="Delete Client" action="#{clientAction.delete}"
      rendered="#{s:hasPermission('client','delete',cl)"/>
  </h:column>
</h:dataTable>
```

13.6.4. 保护页面

页面安全要求应用程序使用 `pages.xml` 文件，但是配置极为简单。只要在你希望保护的 `page` 元素内部包括一个 `<restrict/>` 元素。如果没有通过 `restrict` 元素指定明确的限制，当网页通过 `non-faces` (GET) 的请求被访问时， `/viewId.xhtml:render` 隐含的许可将被检查，并且当网页上任何的JSF `postback` (表单提交) 都需要 `/viewId.xhtml:restore` 许可。否则，特定的约束将作为一个标准的安全表达式进行取值。下面有几个例子：

```
<page view-id="/settings.xhtml">
  <restrict/>
</page>
```

本页面已经隐含了 `/settings.xhtml:render` 所需的 `non-faces` 请求的许可，并隐含了 `/settings.xhtml:restore` 所需的 `faces` 请求的许可。

```
<page view-id="/reports.xhtml">
  <restrict>#{s:hasRole('admin')}</restrict>
</page>
```

这个页面的 `faces` 和 `non-faces` 请求，都需要用户是 `admin` 角色的一个成员。

13.6.5. 保护实体

Seam Security也使得对实体的读取、插入、更新和删除动作应用安全限制成为可能。

为了保护一个实体类的所有动作，在类自身上添加一个 `@Restrict` 注解：

```
@Entity
```

```
@Name("customer")
@Restrict
public class Customer {
    ...
}
```

如果没有在 `@Restrict` 注解中指定任何表达式，执行的默认安全检查就是 `entityName:action` 的许可检查，在这里，`entityName` 是实体的Seam组件名（或者如果指定了`@Name`，则是完全匹配类名），并且 `action` 可以是 `read`、`insert`、`update` 或者 `delete`。

也可能通过把`@Restrict`注解放在相关的实体生命周期的方法上（被注解如下），而只限制某些动作：

- `@PostLoad` - 在实体实例从数据库中加载之后调用。用这个方法配置一个 `read` 许可。
- `@PrePersist` - 在插入实体的一个新实例之前调用。用这个方法配置一个 `insert` 许可。
- `@PreUpdate` - 在实体更新之前调用。用这个方法配置一个 `update` 许可。
- `@PreRemove` - 在实体删除之前调用。用这个方法配置一个`delete`许可。

这里有一个例子，说明实体方法如何配置成给任何 `insert` 操作执行安全检查。 请注意不需要方法去做任何事情，有关安全的唯一重要的东西是它如何被注解：

```
@PrePersist @Restrict
public void prePersist() {}
```

这里还有一个例子，说明实体许可规则检查被验证的用户是否被允许（从seamspac例子中）插入新的 `MemberBlog` 记录。 正在进行安全检查的实体，被自动断言到工作内存中（在这个例子中是 `MemberBlog`）：

```
rule InsertMemberBlog
    no-loop
    activation-group "permissions"
    when
        check: PermissionCheck(name == "memberBlog", action == "insert", granted == false)
        Principal(principalName : name)
        MemberBlog(member : member -> (member.getUsername().equals(principalName)))
    then
        check.grant();
    end;
```

这个规则将授予许可 `memberBlog:insert`，如果当前被验证的用户（由 `Principal` fact表明）有着与正在为其创建Blog项的一样的成员。 可以在 `Principal` fact（和其它地方）中见到的“`name : name`”结构是一个变量绑定 - 它把 `Principal`的 `name` 属性绑定到一个具名 `name` 的变量上。 变量绑定允许值在其它地方被引用，例如下面的行把成员的用户名与 `Principal` 名称进行比较。 想了解更多细节，请参考JBoss Rules文档。

最后，需要安装一个监听器类，把Seam Security与你的JPA提供者整合起来。

13.6.5.1. 使用JPA的实体安全

对EJB3实体Bean的安全检查通过一个 `EntityListener` 执行来完成的。 你可以利用下列 `META-INF/orm.xml` 文件安装这个监听器：

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
    version="1.0">

    <persistence-unit-metadata>
        <persistence-unit-defaults>
            <entity-listeners>
                <entity-listener class="org.jboss.seam.security.EntitySecurityListener"/>
            </entity-listeners>
        </persistence-unit-defaults>
    </persistence-unit-metadata>

</entity-mappings>
```

13.6.5.2. 使用Hibernate的实体安全

如果你正在使用一个经由Seam配置的Hibernate `SessionFactory`，使用实体安全就不需要做任何特别的事情。

13.7. 编写安全规则

迄今为止，我们已经提到了许多许可，但是没有提及许可事实上如何定义或者授与。 本节将对此进行阐述，解释许可检查如何进行，以及如何给一个Seam应用程序实现许可检查。

13.7.1. 许可概述

Security API如何知道用户是否具有对一个特定客户的 `customer:modify` 许可？ Seam Security提供一种新奇的方法，根据JBoss Rules确定用户许可。 使用规则引擎的两个好处在于：1）它是每个用户许可背后的业务逻辑的一个集中位置， 2）速度 - JBoss Rules使用非常有效的算法，用来给涉及多个条件的大量复杂规则取值。

13.7.2. 配置规则文件

Seam Security希望找到一个称作 `securityRules` 的 `RuleBase` 组件，Seam Security用它来给许可检查取值。 这在 `components.xml` 中配置如下：

```
<components xmlns="http://jboss.com/products/seam/components"
    xmlns:core="http://jboss.com/products/seam/core"
    xmlns:security="http://jboss.com/products/seam/security"
    xmlns:drools="http://jboss.com/products/seam/drools"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
        "http://jboss.com/products/seam/core http://jboss.com/products/seam/core-2.0.xsd
        http://jboss.com/products/seam/components http://jboss.com/products/seam/components-2.0.xsd
        http://jboss.com/products/seam/drools http://jboss.com/products/seam/drools-2.0.xsd"
        http://jboss.com/products/seam/security http://jboss.com/products/seam/security-2.0.xsd">

    <drools:rule-base name="securityRules">
        <drools:rule-files>
```



```

        <value>/META-INF/security.drl</value>
    </drools:rule-files>
</drools:rule-base>

</components>

```

一旦配置了 RuleBase 组件，就可以编写安全规则了。

13.7.3. 创建安全规则文件

对于这个步骤，要在应用程序的jar文件的 /META-INF 目录中创建一个名为 security.drl 的文件。事实上，这个文件可以命名为你喜欢的任何名字，并存在于任何位置，只要它在 components.xml 中进行了适当的配置。

那么安全规则文件应该包含什么呢？目前，至少浏览一下JBoss Rules文档可能是个好主意，但是这里介绍的是一个极为简单的例子：

```

package MyApplicationPermissions;

import org.jboss.seam.security.PermissionCheck;
import org.jboss.seam.security.Role;

rule CanUserDeleteCustomers
when
    c: PermissionCheck(name == "customer", action == "delete")
    Role(name == "admin")
then
    c.grant();
end;

```

我们来分解一下。我们见到的第一个东西是包声明。包在JBoss Rules中基本上是一个规则的集合。包名可以是喜欢的任何名称 — 它不影响规则基础范围之外的任何其它东西。

然后，我们可能注意到二个对 PermissionCheck 和 Role 类的导入语句。这些导入通知规则引擎我们将在规则中引用这些类。

最后是规则的代码。包中的每一个规则都应该有一个唯一的名称（一般描述规则的目的）。在这个例子中，我们的规则称作 CanUserDeleteCustomers，用来检查用户是否被允许删除一个客户记录。

看看规则定义的主体，可以发现两个独特的部分。规则有称为LHS(左手边)和RHS(右手边)的东西。LHS由规则的条件部分组成，如必须满足规则以便触发的一系列条件。LHS由 when 部分表示。RHS是结果，或者是规则的动作部分，该规则只在满足LHS的所有条件时才触发。RHS由 then 部分组成。规则的末端用 end; 线表示。

如果看看规则的LHS，就可见到那里列出了两个条件。我们先来检验第一个条件：

```

c: PermissionCheck(name == "customer", action == "delete")

```

说得通俗些，这个条件声明工作内存中必须存在一个 PermissionCheck 对象，它具有与“customer”相当的 name 属性，以及与“delete”相当的 action 属性。什么是工作内存？它是个会话范围的对象，包含规则引擎进行有关许可检查决策时所需的上下文信息。每次调用 hasPermission() 方法时，一个临时的 PermissionCheck 对象或者 Fact 就被断言到工作内存中。这个 PermissionCheck 正好对应于正被检查的许可，因此，例如如果调用 hasPermission("account", "create", null)，那么带有相当于“account”的

name 和相当于“create”的 action 的 PermissionCheck 对象，就将在许可检查持续期间被断言到工作内存中。

工作内存中还有什么其它的东西？除了在许可检查期间断言的短期临时fact被插入之外，工作内存中还有一些长期的对象，在用户验证的整个持续期间都保留在那。这些包括作为验证过程一部分而创建的任何 java.security.Principal 对象，还包括用户所属角色中的每一个角色的 org.jboss.seam.security.Role。通过调用 ((RuleBasedIdentity) RuleBasedIdentity.instance()).getSecurityContext().insert()，也可能断言额外的长期fact到工作内存中，把对象当作参数传递。

回到我们的简单例子上来，也会注意到我们LHS的第一行加上了 c: 的前缀。这是个变量绑定，用来指回到符合条件的对象。移到LHS的第二行，会看到：

```
Role(name == "admin")
```

这个条件只声明工作内存中必须有 Role 对象带有“admin”的 name。如前所述，用户角色被作为长期fact断言到工作内存中。因此，把两个条件放在一起，这个规则基本上等于在说：“如果你检查 customer:delete 许可，并且用户是 admin 角色的一员时，我将会触发。”

那么规则触发的结果会怎样？我们来看看规则的RHS：

```
c.grant()
```

RHS由Java代码组成，在这个例子中是调用 c 对象的 grant() 方法，如前面提到过的，它是个对 PermissionCheck 对象的变量绑定。除了 PermissionCheck 对象的 name 和 action 属性之外，也有一个 granted 属性，它的初始值设置为 false。在 PermissionCheck 上调用 grant() 方法，设置 granted 属性为 true，这意味着许可检查成功了，允许用户执行许可检查预定的任何动作。

13.7.3.1. 通配符许可检查

通过在规则中给 PermissionCheck 删除 action 约束，可能实现通配符许可检查（对一个指定的许可名称允许所有动作），像这样：

```
rule CanDoAnythingToCustomersIfYouAreAnAdmin
when
  c: PermissionCheck(name == "customer")
  Role(name == "admin")
then
  c.grant();
end;
```

这个规则允许带有 admin 角色的用户对任何 customer 许可检查执行任何操作。

13.8. SSL安全

Seam包括对通过HTTPS协议提供敏感的页面的基本支持。这很容易通过在 pages.xml 中给页面指定 scheme 而配置。下列例子说明视图 /login.xhtml 如何配置为使用HTTPS：

```
<page view-id="/login.xhtml" scheme="https">
```

这个配置自动扩展为 `s:link` 和 `s:button` JSF控制，它（在指定 `view` 时）还将渲染使用了正确协议的链接。在前一个例子的基础上，下列链接将使用HTTPS协议，因为 `/login.xhtml` 配置为使用HTTPS：

```
<s:link view="/login.xhtml" value="Login"/>
```

使用 错误 协议时，直接浏览视图将导致重定向到与使用 正确 协议一样的视图。例如，浏览一个让 `scheme="https"` 使用HTTP的页面时，将重定向到与使用HTTPS一样的页面。

也可能给所有页面配置一个 默认的scheme。这事实上相当重要，就像你可能只希望给一些页面使用HTTPS一样。如果没有指定默认的scheme，那么默认的行为就是继续使用当前的scheme。这意味着一旦你通过HTTPS进入了一个页面，HTTPS就将持续使用，即使你导航到了另一个非HTTPS的页面（糟糕的事！）。因此强烈建议包括一个默认的 `scheme`，通过在默认的 `"*"` 视图上配置它：

```
<page view-id="*" scheme="http" />
```

当然，如果你的应用程序中 没有 任何页面使用HTTPS，那就不需要指定默认的scheme。

你可以配置Seam来自动地在每次scheme改变时使当前的HTTP会话失效。只要在 `components.xml` 文件中加入这一行：

```
<core:servlet-session invalidate-on-scheme-change="true"/>
```

这个选项可以让你的系统减少被嗅探到Session ID或者因为页面从使用HTTPS转到HTTP时导致敏感数据的泄露。

13.9. 实现Captcha测试

虽然严格来说它不是Security API的一部分，但是它在某些环境下（例如新用户注册，发布到一个公共的blog或者论坛），可以用来实现Captcha（Completely Automated Public Turing test to tell Computers and Humans Apart），以防止自动的机器人与应用程序进行交互。Seam提供与JCaptcha的无缝整合，是产生Captcha challenge的一个极好的库。如果你希望在应用程序中使用Captcha特性，就要把来自Seam lib目录的jcaptcha-* jar文件包括在项目中，并在 `application.xml` 中把它注册为一个Java模块。

13.9.1. 配置Captcha Servlet

为了建立并运行起来，需要配置Seam Resource Servlet，这将给你的页面提供Captcha challenge映射。这在 `web.xml` 中需要下列项：

```
<servlet>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <servlet-class>org.jboss.seam.servlet.SeamResourceServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <url-pattern>/seam/resource/*</url-pattern>
</servlet-mapping>
```

13.9.2. 添加Captcha到页面

添加一个Captcha challenge到页面极为容易。 Seam提供一个page-scoped组件， captcha， 它提供它所需要的一切， 包括内置的captcha校验。 这里有个例子：

```
<div>
  <h:graphicImage value="/seam/resource/captcha?#{captcha.id}"/>
</div>

<div>
  <h:outputLabel for="verifyCaptcha">Enter the above letters</h:outputLabel>
  <h:inputText id="verifyCaptcha" value="#{captcha.response}" required="true">
    <s:validate />
  </h:inputText>
  <div class="validationError"><h:message for="verifyCaptcha"/></div>
</div>

<div>
  <h:commandButton action="#{register.next}" value="Register"/>
</div>
```

这就是关于它的所有信息。 graphicImage 控制显示Captcha challenge， inputText 接收用户的响应。 当表单被提交时这个响应会根据Captcha被自动地校验。

13.9.3. 定制Captcha图片

Captcha图片本身可以通过 ImageCaptchaService 和默认的 (DefaultManageableImageCaptchaService) 进行定制。 要配置不同的 ImageCaptchaService， 在 components.xml 文件中添加以下条目：

```
<component name="org.jboss.seam.captcha.captchaImage" service="#{customCaptcha.service}"/>
```

service 属性指明 ImageCaptchaService 实例用来生成Captcha图片。 更多关于配置一个 ImageCaptchaService 的信息， 请参见JCaptcha文档。 这里有一个定制图片生成器的例子（可以在 seamspace例子里找到）：

```
@Name("customCaptcha")
public class CustomCaptcha
{
  public ImageCaptchaService getService()
  {
    BasicGimpyEngine customCaptcha = new BasicGimpyEngine();
    GimpyFactory factory = new GimpyFactory(
      new RandomWordGenerator("ABCDEFGHIJKLMNOPQRSTUVWXYZ23456789"),
      new ComposedWordToImage(new RandomFontGenerator(new Integer(15),
        new Integer(15)), new UniColorBackgroundGenerator(new Integer(150),
          new Integer(30)), new RandomTextPaster(new Integer(4),
            new Integer(7), Color.BLACK)));
    GimpyFactory[] factories = {factory};
    customCaptcha.setFactories(factories);

    return new DefaultManageableImageCaptchaService(
      new FastHashMapCaptchaStore(),
      customCaptcha,
      180,
      120000,
      75000);
  }
}
```

第 14 章 国际化和主题

Seam通过提供几个内置部件来为UI提供多语言支持，从而使构建国际化的应用程序变得十分容易。

14.1. 本地化

每一个用户登录会话都有一个相关的 `java.util.Locale` 实例（以名为 `locale` 的组件形式提供给应用程序）。一般情况下，不需要做任何特别的配置设置locale，Seam 委托JSF来判断当前的活动locale：

faces-config.xml

faces-config.xml

通过Seam的以下几个配置属性来手工设置locale也是你可能的：

`org.jboss.seam.international.localeSelector.language`、`org.jboss.seam.international.localeSelector.country`和`org.jboss.seam.international.localeSelector.variant`，但是并不推荐这种做法。

然而，允许用户通过应用程序的用户界面来手工设置locale也是很有益处的。Seam提供了内置的功能来覆盖通过上述算法决定的locale。你所要做的只是在JSP或Facelet的Form中增加以下代码段：

```
<h:selectOneMenu value="#{localeSelector.language}">
  <f:selectItem itemLabel="English" itemValue="en"/>
  <f:selectItem itemLabel="Deutsch" itemValue="de"/>
  <f:selectItem itemLabel="Francais" itemValue="fr"/>
</h:selectOneMenu>
<h:commandButton action="#{localeSelector.select}" value="#{messages[' ChangeLanguage'] }"/>
```

或者，如果你想要一个 `faces-config.xml` 支持的所有locale的列表，就用：

```
<h:selectOneMenu value="#{localeSelector.localeString}">
  <f:selectItems value="#{localeSelector.supportedLocales}" />
</h:selectOneMenu>
<h:commandButton action="#{localeSelector.select}" value="#{messages[' ChangeLanguage'] }"/>
```

当在下拉列表中选择一项，并按下按钮后，随后会话中Seam和JSF的locale就被刷新了。

14.2. 标签

JSF 通过使用 `<f:loadBundle />` 来支持用户界面标签和描述文本的国际化。这个方法同样可以用在Seam应用程序中。或者，可以利用Seam的 `messages` 组件用内嵌的EL表达式来显示模板标签。

14.2.1. 定义标签

Seam提供了一个 `java.util.ResourceBundle` （以`org.jboss.seam.core.resourceBundle` 的名字提供给应用程序）。你需要通过这个指定的资源包来使你的国际化标签可用。默认情况下，Seam 使用名为`messages`

的资源包， 你需要在 `messages.properties`、`messages_en.properties`、`messages_en_AU.properties` 等文件中定义你的标签。这些文件通常在 `WEB-INF/classes` 目录下。

因此，在 `messages_en.properties`中：

```
Hello=Hello
```

和在 `messages_en_AU.properties`中：

```
Hello=G' day
```

你可以通过设置Seam的配置属性 `org.jboss.seam.core.resourceLoader.bundleNames` 为资源包选择一个不同的名字。 甚至可以指定一个资源包名称列表，以深度优先进行消息的搜索。

```
<core:resource-loader>
  <core:bundle-names>
    <value>mycompany_messages</value>
    <value>standard_messages</value>
  </core:bundle-names>
</core:resource-loader>
```

如果想为一个特殊页定义消息，可在以一个和JSF View id同名的资源包中指定，去掉前置 `/` 和文件扩展名。 这样，如果我们只想在 `/welcome/hello.jsp` 中显示消息，就把它置于 `welcome/hello_en.properties` 中。

你还可以在 `pages.xml` 中指定一个显式的绑定名称：

```
<page view-id="/welcome/hello.jsp" bundle="HelloMessages"/>
```

这样，我们就可以在 `/welcome/hello.jsp` 中使用定义在 `HelloMessages.properties` 中的消息了。

14.2.2. 标签显示

如果使用Seam的资源包来定义标签，就不用每页再写 `<f:loadBundle ... />` 了，可以使用这种简单的形式：

```
<h:outputText value="#{messages['Hello']}" />
```

或者：

```
<h:outputText value="#{messages.Hello}" />
```

更好的一点是，message自身可以包含EL表达式：

```
Hello=Hello, #{user.firstName} #{user.lastName}
```

```
Hello=G' day, #{user.firstName}
```

你也可以在代码中这样使用消息：

```
@In private Map<String, String> messages;
```

```
@In("#{messages['Hello']}") private String helloMessage;
```

14.2.3. Faces Messages

`facesMessages` 组件是一个向用户显示成功或者失败消息的非常方便的途径。 我们之前描述的功能对Faces Messages同样有效：

```
@Name("hello")
@Stateless
public class HelloBean implements Hello {
    @In FacesMessages facesMessages;

    public String sayIt() {
        facesMessages.addFromResourceBundle("Hello");
    }
}
```

这将根据用户的locale显示 `Hello, Gavin King` 或者 `G' day, Gavin`。

14.3. 时区

Seam中还有一个session范围的 `java.util.Timezone` 实例，叫做 `org.jboss.seam.international.timezone`， 和一个名为 `org.jboss.seam.international.timezoneSelector` 的用于设置时区的组件。默认情况下，时区取服务器的默认时区。不幸的是，JSF规范中讲所有的日期和时间都假设是UTC 的，并且显示为UTC，除非使用 `<f:convertDateTime>` 明确地为其指定时区。这是一个非常不方便的默认行为。

Seam覆写了这个行为，默认所有的日期和时间都是Seam的时区。另外，Seam提供了 `<s:convertDateTime>` 标签，用来处理Seam 时区的转化。

14.4. 主题

Seam应用程序可以很方便地改变皮肤。Theme API和本地化API非常相似，但是它们二者的关注点截然不同，一些应用同时支持本地化和主题。

首先，配置所支持的主题集合：

```
<theme:theme-selector cookie-enabled="true">
    <theme:available-themes>
        <value>default</value>
        <value>accessible</value>
        <value>printable</value>
    </theme:available-themes>
</theme:theme-selector>
```

注意，第一个是默认的主题。

主题定义在一个和该主题同名的属性文件中。例如，`default` 主题定义在`default.properties`中。`default.properties`可能是这样定义的：

```
css ../screen.css
template /template.xhtml
```

通常主题资源包的内容是CSS样式或图片的路径和facelet模板（不像本地化资源包那样通常是文本）。

现在我们可以使用JSP或者Facelet页面中使用这些内容了。例如，一个Facelet页的风格可以这样：

```
<link href="#{theme.css}" rel="stylesheet" type="text/css" />
```

或者，当页面定义在一个子目录中时可以这样：

```
<link href="#{facesContext.externalContext.requestContextPath}#{theme.css}"
      rel="stylesheet" type="text/css" />
```

最强大的是，Facelet让我们通过 `<ui:composition>` 把模板主题化：

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core"
  template="#{theme.template}">
```

正如locale选择器一样，有一个内置的主题选择器允许用户在各主题间自由地切换：

```
<h:selectOneMenu value="#{themeSelector.theme}">
  <f:selectItems value="#{themeSelector.themes}" />
</h:selectOneMenu>
<h:commandButton action="#{themeSelector.select}" value="Select Theme" />
```

14.5. 使用cookie保存locale和主题设置

locale选择器、主题选择器和时区选择器全都支持持久化，把参数保存到cookie中。仅需要在components.xml中设置 `cookie-enabled` 配置属性：

```
<theme:theme-selector cookie-enabled="true">
  <theme:available-themes>
    <value>default</value>
    <value>accessible</value>
    <value>printable</value>
  </theme:available-themes>
</theme:theme-selector>

<international:locale-selector cookie-enabled="true" />
```


第 15 章 Seam Text

面向协作的网站需要一个友好的标记语言来简化对论坛帖子、Wiki页面、博客（Blog）、评论等等格式化文本的处理。Seam提供了 `<s:formattedText/>` 控件来显示符合 Seam Text 语言的格式化文本。Seam Text是用一个基于ANTLR的语法分析器来实现的。但你不需要知道ANTLR就能方便地使用它了。

15.1. 基本格式化

下面是个简单的例子：

```
It's easy to make *bold text*, /italic text/, |monospace|,  
~deleted text~, superscripts or _underlines_.
```

如果我们用 `<s:formattedText/>` 来显示它，会产生下面的HTML：

```
<p>  
It's easy to make <b>bold text</b>, <i>italic text</i>, <tt>monospace</tt>  
<del>deleted text</del>, super<sup>scripts</sup> or <u>underlines</u>.  
</p>
```

可以用一个空行来表示新的段落，用 `+` 来表示一个标题：

```
+This is a big heading  
You /must/ have some text following a heading!  
  
++This is a smaller heading  
This is the first paragraph. We can split it across multiple  
lines, but we must end it with a blank line.  
  
This is the second paragraph.
```

（请注意简单的换行是被忽略的，你需要一个额外的空行把文本换行到一个新的段落中。）下面就是HTML的结果：

```
<h1>This is a big heading</h1>  
<p>  
You <i>must</i> have some text following a heading!  
</p>  
  
<h2>This is a smaller heading</h2>  
<p>  
This is the first paragraph. We can split it across multiple  
lines, but we must end it with a blank line.  
</p>  
  
<p>  
This is the second paragraph.  
</p>
```

用 `#` 字符来建立有序的列表。至于无序的列表就用 `=` 字符：

```
An ordered list:
```

```
#first item
#second item
#and even the /third/ item
```

An unordered list:

```
=an item
=another item
```

```
<p>
```

An ordered list:

```
</p>
```

```
<ol>
```

```
<li>first item</li>
```

```
<li>second item</li>
```

```
<li>and even the <i>third</i> item</li>
```

```
</ol>
```

```
<p>
```

An unordered list:

```
</p>
```

```
<ul>
```

```
<li>an item</li>
```

```
<li>another item</li>
```

```
</ul>
```

引用的部分应该用双引号括起来:

The other guy said:

```
"Nyeah nyeah-nee
/nyeah/ nyeah!"
```

But what do you think he means by "nyeah-nee"?

```
<p>
```

The other guy said:

```
</p>
```

```
<q>Nyeah nyeah-nee
```

```
<i>nyeah</i> nyeah!</q>
```

```
<p>
```

But what do you think he means by <q>nyeah-nee</q>?

```
</p>
```

15.2. 输入代码和有特殊字符的文本

像 *、| 和 # 这样的特殊字符，和诸如 <、> 和 & 之类的HTML字符可以用来 \ 来转义:

You can write down equations like 2*3=6 and HTML tags like \<body\> using the escape character: \.

```
<p>
```

You can write down equations like 2*3=6 and HTML tags

like <body> using the escape character: \.

```
</p>
```

代码段可以用倒单引号（```）括起来：

```
My code doesn't work:

`for (int i=0; i<100; i--)
{
    doSomething();
}`

Any ideas?
```

```
<p>
My code doesn't work:
</p>

<pre>for (int i=0; i<100; i--)
{
    doSomething();
}</pre>

<p>
Any ideas?
</p>
```

请注意行间的空格会转码（大部分的空格都会被格式化成文本形式，实际上有很多的特定字符都是代码或者标签）。因此你可以写成：

```
This is a |<tag attribute="value"/>| example.
```

在这里面的空格没有转换成任何的字符。下面你不能通过任何方式来格式化行间的空格文本（斜体字，下划线等等）。

15.3. 链接

可以用下面的语法来建立一个链接：

```
Go to the Seam website at [=>http://jboss.com/products/seam].
```

如果你想指定链接的文本，也可以这样：

```
Go to [the Seam website=>http://jboss.com/products/seam].
```

对于高级用户，甚至可以自定义Seam Text解析器，让它诠释这个语法书写的Wiki词汇。

15.4. 输入HTML

文本可能会包含一个HTML的有限子集（不用担心，这个子集可以抵御跨站脚本攻击）。这在建立链接时很有用：

```
You might want to link to <a href="http://jboss.com/products/seam">something
cool</a>, or even include an image: 
```

建立表格时也一样:

```
<table>
  <tr><td>First name:</td><td>Gavin</td></tr>
  <tr><td>Last name:</td><td>King</td></tr>
</table>
```

如果你想的话你还能做得更多!

第 16 章 iText PDF生成

Seam目前包括了一个利用iText生成文档的组件集。Seam的iText文档支持主要关注于PDF文档的生成，但它也对生成RTF文档提供基本的支持。

16.1. 使用PDF支持

iText支持由 `jboss-seam-pdf.jar` 提供。这个JAR包中含有用来构造可以渲染成PDF的视图的iText JSF控件和把渲染好的文档提供给用户的DocumentStore组件。为了在你的应用程序中包括PDF支持，要把 `jboss-seam-pdf.jar` 和iText JAR 文件一起放在你的 `WEB-INF/lib` 路径下。要使用Seam的iText支持就只是这样，无需更多的配置了。

Seam iText模块需要使用Facelets作为视图技术。这个库的未来版本也可能支持使用JSP。此外，这个模块还需要用到seam-ui包。

工程 `examples/itext` 包含了一个PDF支持实践的例子。这个例子示范了正确的部署包，它包含一些例子，用来示范目前支持PDF生成的关键特性。

16.1.1. 创建一个文档

<code><p:document></code>	<div>描述</div> <p>文档是由facelets文档利用命名空间 <code>http://jboss.com/products/seam/pdf</code> 中的标签生成的。文档应该总是以 <code>document</code> 标签作为文件的根结点。 <code>document</code> 标签为Seam产生文档到文档库并渲染一个HTML重定向到存储内容做准备。</p> <div>属性</div> <ul style="list-style-type: none"><code>type</code> — 要生成文档的类型。有效值为PDF、RTF 和 HTML 模式。Seam默认为PDF生成，并且很多特性只有在生成PDF文档时才能正确工作。最常用的值是 <code>LETTER</code> 和 <code>A4</code>。被支持页面大小的全部列表可以在类 <code>com.lowagie.text.PageSize</code> 中找到。也可以这样，<code>pageSize</code>可以直接指定宽度和高度。例如，值“612 792”与LETTER的页面大小是一样的。<code>orientation</code> — 页面的定位。有效值是 <code>portrait</code> 和 <code>landscape</code>，在前景模式下，页面大小的高度和宽度值是颠倒的。<code>margins</code> — 有左边距，右边距，上边距和下边距值。<code>marginMirroring</code> — 显示页边距设置应该反转一个交换页面。 <div>Metadata属性</div> <ul style="list-style-type: none">标题主题
---------------------------------	---

	<ul style="list-style-type: none">• 关键字• 作者• 创建者 <p>用法</p> <pre><p:document xmlns:p="http://jboss.com/products/seam/pdf"> The document goes here. 文档指向这里。 </p:document></pre>
--	---

16.1.2. 基本的文本元素

有用的文档需要包含的不仅仅是文本。但是标准的UI组件只适合生成HTML，而不适合生成PDF内容。反之，Seam则提供了一种特殊的UI组件，用来生成适当的PDF内容。像 `<p:image>` 和 `<p:paragraph>` 这样标签是简单文档的根本。像 `<p:font>` 这样的标签给它们周围的所有内容都提供了样式信息。

<code><p:paragraph></code>	<p>描述</p> <p>为了使文本片段能够按照逻辑分组布局、格式化、修饰，段落标签中包含了文本的大部分用法。</p> <p>属性</p> <ul style="list-style-type: none">• <code>firstLineIndent</code> 行首缩进• <code>extraParagraphSpace</code> 额外段落空间• <code>leading</code> 段首• <code>multipliedLeading</code> 复合段首• <code>spacingBefore</code> — 段落元素前要插入的空白• <code>spacingAfter</code> — 段落元素后要插入的空白• <code>indentationLeft</code> 左缩进• <code>indentationRight</code> 右缩进• <code>keepTogether</code> 保持对齐 <p>用法</p> <pre><p:paragraph alignment="justify"> This is a simple document. It isn't very fancy. 这是一个简单的文档，它不是很常用。 </p:paragraph></pre>
----------------------------------	--

<code><p:text></code>	<p>描述</p> <p><code>text</code> 标签能够使用一般的JSF转换机制从应用程序数据中生成成为文本片段。 在渲染HTML文档的时候，它的用法非常类似于 <code>outputText</code> 标签。</p> <p>属性</p> <ul style="list-style-type: none"><code>value</code> — 要显示的这个值，通常是一个值绑定表达式。 <p>用法</p> <pre><p:paragraph> The item costs 显示一件商品的价格<p:text value="#{product.price}"> <f:convertNumber type="currency" currencySymbol="\$"/> </p:text> </p:paragraph></pre>
-----------------------------	---

<code><p:font></code>	<p>描述</p> <p><code>font</code>标签为它内部的所有文本定义要使用的默认字体。</p> <p>属性</p> <ul style="list-style-type: none"><code>name</code> — 字体名称，例如： <code>COURIER</code>、<code>HELVETICA</code>、<code>TIMES-ROMAN</code>、<code>SYMBOL</code> 或者 <code>ZAPFDINGBATS</code>。<code>size</code> — 字体大小<code>style</code> — 字体样式，下面这些的任意组合： <code>NORMAL</code>、<code>BOLD</code>、<code>ITALIC</code>、<code>OBLIQUE</code>、<code>UNDERLINE</code>、<code>LINE-THROUGH</code><code>encoding</code> — 字符设置编码 <p>用法</p> <pre><p:font family="courier" style="bold" size="24"> <p:paragraph>My Title</p:paragraph> </p:font></pre>
-----------------------------	--

<code><p:newPage></code>	<p>描述</p> <p><code>p:newPage</code> 插入一个新页面。</p> <p>用法</p> <pre><p:newPage /></pre>
--------------------------------	---

<p:image>

描述

`p:image` 将一张图片插入到文档中。 利用 `value` 属性从 `classpath` 或者 Web 应用程序上下文加载图片。

资源也可以由应用程序代码动态地生成。 `imageData` 属性可以指定一个值为 `java.awt.Image` 对象的值绑定表达式。

属性

- `value` — 一个资源名称或者是一个应用程序生成的图片的方法表达式绑定。
- `rotation` — 图片旋转角度。
- `height` — 图片高度。
- `width` — 图片宽度。
- `alignment` — 图片对齐方式。（可能的值请见 第 16.1.7.2 节 “对齐方式值”）
- `alt` — 替换图片的文本。
- `indentationLeft` 左缩进
- `indentationRight` 右缩进
- `spacingBefore` — 元素前要插入的空白。
- `spacingAfter` — 元素后要插入的空白
- `widthPercentage` 宽度百分比
- `initialRotation` 初始旋转
- `dpi` 像素
- `scalePercent` (图片放缩比例) — 图片放缩比例因子（百分比）。 可以是一个百分比值，也可以两个分别代表X方向和Y方向的百分比值。
- `wrap`
- `underlying` 下划线

用法

```
<p:image value="/jboss.jpg" />
```

```
<p:image value="#{images.chart}" />
```


<p><p:anchor></p>	<p>描述</p> <p>p:anchor 定义文档中的活链接。它支持下面的属性：</p> <p>属性</p> <ul style="list-style-type: none"> • name — 文档中目标锚点的名称。 • reference — 链接指向的目标。文档中其他点的链接应该以一个“#”开头。例如，“#link1”用 link1 的一外名称指向 name 另一个锚点位置。链接也可以是指向文档之外的一个资源的完整URL路径。 <p>用法</p> <pre><p:listItem><p:anchor reference="#reason1">Reason 1</p:anchor></p:listItem> ... <p:paragraph> <p:anchor name="reason1">It's the quickest way to get "rich"</p:anchor> ... </p:paragraph></pre>
-------------------------	---

16.1.3. 页眉和页脚

<p><p:header></p> <p><p:footer></p>	<p>描述</p> <p>p:header 和 p:footer 组件提供了将页眉和页脚文本放在生成文档的每个页面上的能力，除了第一个页面之外。页眉和页脚声明应该出现在文档的顶部。</p> <p>属性</p> <ul style="list-style-type: none"> • alignment — 页眉 / 页脚框的对齐方式。（对齐方式的取值请见 第 16.1.7.2 节 “对齐方式值”） • backgroundColor — 页眉 / 页脚框的背景色。（颜色值请见 第 16.1.7.1 节 “颜色值”） • borderColor — 页眉 / 页脚框的边框颜色。单独设置各边框颜色用 borderColorLeft、borderColorRight、borderColorTop 和 borderColorBottom。（颜色值请见 第 16.1.7.1 节 “颜色值”） • borderWidth — 边框的宽度。每一条单独的边可以利用 borderWidthLeft、borderWidthRight、borderWidthTop 和 borderWidthBottom 来指定。 <p>用法</p> <pre><p:facet name="header"> <p:font size="12"> <p:footer borderWidthTop="1" borderColorTop="blue" borderWidthBottom="0" alignment="center"></pre>
---	---

	<pre> Why Seam? [<p:pageNumber />] </p:footer> </p:font> </f:facet> </pre>
<p:pageNumber>	<p>描述</p> <p>通过 <code>p:pageNumber</code> 标签能够将当前页码放到页眉或者页脚的位置。该标签只能用在页眉或者页脚的上下文中，并且只能使用一次。</p> <p>用法</p> <pre> <p:footer borderWidthTop="1" borderColorTop="blue" borderWidthBottom="0" alignment="center"> Why Seam? [<p:pageNumber />] </p:footer> </pre>

16.1.4. 章节

<p:chapter> <p:section>	<p>描述</p> <p>如果按照书籍 / 文章的结构生成文档，<code>p:chapter</code> 和 <code>p:section</code> 标签就可以用来提供必要的结构。小节标签只能在章的内部使用，但是它们之间却可以任意地嵌套。大多数的PDF浏览工具都提供可以在文档内部章节之间自由切换的功能。</p> <p>属性</p> <ul style="list-style-type: none"> <code>alignment</code> — 页眉 / 页脚框的对齐方式。（对齐方式取值请见 第 16.1.7.2 节 “对齐方式值”） <code>number</code> — 章序号。每章都应该有一个章序号。 <code>numberDepth</code> — 文档中章节的层次数。所有的小节都有一个相对于它们周围章 / 节的序号。如果文章的默认显示层次数是3，那么3.1.4就表示第三章第一节的第四小节。如果忽略章序号，那么层数就是2，上述小节的序号就应该显示为1.4。 <p>用法</p> <pre> <p:document xmlns:p="http://jboss.com/products/seam/pdf" title="Hello"> <p:chapter number="1"> <p:title><p:paragraph>Hello</p:paragraph></p:title> <p:paragraph>Hello #{user.name}!</p:paragraph> </p:chapter> <p:chapter number="2"> <p:title><p:paragraph>Goodbye</p:paragraph></p:title> </pre>
--------------------------------	--

	<pre><p:paragraph>Goodbye #{user.name}.</p:paragraph> </p:chapter> </p:document></pre>
--	---

<p:header>	<p>描述</p> <p>任何一个章或节都可以包含一个 <code>p:title</code> 标签，用于显示下一章节的序号。 标签的主体可以包含原始文本，或者是一个 <code>p:paragraph</code> 标签。</p>
------------	---

16.1.5. 列表

标签 `p:list` 和 `p:listItem` 可以显示列表结构。 列表里可以包含任意嵌套的子列表。列表中的项不能在列表之外使用。 在文档中通过以下文档使用 `ui:repeat` 标签显示从Seam组件获取到的值的列表。

```
<p:document xmlns:p="http://jboss.com/products/seam/pdf"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  title="Hello">
  <p:list style="numbered">
    <ui:repeat value="#{documents}" var="doc">
      <p:listItem>#{doc.name}</p:listItem>
    </ui:repeat>
  </p:list>
</p:document>
```

<p:list>	<p>属性</p> <ul style="list-style-type: none"><code>style</code> — 列表的有序/无序排列方式。 可选项：NUMBERED、LETTERED、GREEK、ROMAN、ZAPFDINGBATS、ZAPFDINGBATS_NUMBER。 如果没有指定样式，列表项就以无序排列。<code>listSymbol</code> — 针对无序列表，指定列表符号。<code>indent</code> — 列表的缩进程度。<code>lowerCase</code> — 针对以字母方式排序的列表，表明字母是否应该为小写。<code>charNumber</code> — 针对ZAPFDINGBATS排序，指明无序字符的字符编码。<code>numberType</code> — 针对ZAPFDINGBATS_NUMBER排序，指明编号方式。 <p>用法</p> <pre><p:list style="numbered"> <ui:repeat value="#{documents}" var="doc"> <p:listItem>#{doc.name}</p:listItem> </ui:repeat> </p:list></pre>
----------	---

<p><p:listItem></p>	<p>描述</p> <p>p:listItem 支持以下属性：</p> <p>属性</p> <ul style="list-style-type: none"> • alignment — 页眉 / 页脚框的对齐方式。（对齐方式取值请见 第 16.1.7.2 节 “对齐方式值”） • alignment — 列表项目的对齐方式。（可能取值请见 第 16.1.7.2 节 “对齐方式值”） • indentationLeft — 左缩进的量。 • indentationRight — 右缩进的量。 • listSymbol — 覆盖这个列表项目的默认列表符号。 <p>用法</p> <div>...</div>
---------------------------	---

16.1.6. 表格

可以使用标签 `p:table` 和 `p:cell` 创建表格结构。和许多表格结构不同，这里的表格结构没有明确的行声明。如果一个表格有3列，那么每3个单元格会自动组成一行。可以声明标题行和注脚行，并且当一个表格结构跨越多个页面的时候，标题行和注脚行就会重复地出现在每个页面上。

<p><p:table></p>	<p>描述</p> <p>p:table 支持以下属性。</p> <p>属性</p> <ul style="list-style-type: none"> • columns — 组成一个表行的列（单元格）的数量。 • widths — 每个列的相对宽度。每个列应该都要有一个值。例如 <code>widths="2 1 1"</code> 表示这个表格有3列，第一列的宽度是第二列和第三列的两倍。 • headerRows — 初始行数量，可以认为是标题或者注脚行的数量，在表格跨多个页面的时候应该重复的行数。 • footerRows — 被认为是注脚行的行数。这个值应该减去 <code>headerRows</code> 值。如果文档有2行构成标题行、1行构成注脚行，那么 <code>headerRows</code> 应该设置为3，<code>footerRows</code> 应该设置为1 • widthPercentage — 表格占页面宽度的百分比。
------------------------	--

- `horizontalAlignment` — 表格的水平对齐方式。（可能取值请见 第 16.1.7.2 节 “对齐方式值”）
- `skipFirstHeader`
- `runDirection`
- `lockedWidth`
- `splitRows`
- `spacingBefore` — 元素前要插入的空白。
- `spacingAfter` — 元素后要插入的空白。
- `extendLastRow`
- `headersInEvent`
- `splitLate`
- `keepTogether`

用法

```
<p:table columns="3" headerRows="1">
  <p:cell>name</p:cell>
  <p:cell>owner</p:cell>
  <p:cell>size</p:cell>
  <ui:repeat value="#{documents}" var="doc">
    <p:cell>#{doc.name}</p:cell>
    <p:cell>#{doc.user.name}</p:cell>
    <p:cell>#{doc.size}</p:cell>
  </ui:repeat>
</p:table>
```

<p:cell>

描述

`p:cell` 支持下面的属性。

属性

- `colspan` — 通过声明 `colspan` 值大于1，单元格可以跨多个列。表格不具备跨多行的能力。
- `horizontalAlignment` — 单元格的水平对齐方式。（可能取值请见 第 16.1.7.2 节 “对齐方式值”）
- `verticalAlignment` — 单元格的垂直对齐方式。（可能取值请见 第 16.1.7.2 节 “对齐方式值”）
- `padding` — 填充指定的边还可以通过以下属性来指定：`paddingLeft`、`paddingRight`、`paddingTop` 和 `paddingBottom`。

- `useBorderPadding`
- `leading`
- `multipliedLeading`
- `indent` 缩进
- `verticalAlignment` 垂直对齐
- `extraParagraphSpace` 额外段落空间
- `fixedHeight` 固定高度
- `noWrap`
- `minimumHeight` 最小高度
- `followingIndent` 底部缩进
- `rightIndent` 右缩进
- `spaceCharRatio` 字符间距比
- `runDirection` 排列方向
- `arabicOptions` 阿拉伯语选项
- `useAscender` 递增
- `grayFill` 灰色填充
- `rotation` 旋转度

用法

```
<p:cell>...</p:cell>
```

16.1.7. 文档常量

本节例举了一些被属性在多个页面共享的常量。

16.1.7.1. 颜色值

Seam 尚未支持全部颜色的定义。目前只支持下面的颜色：`white`、`gray`、`lightgray`、`darkgray`、`black`、`red`、`pink`、`yellow`、`green`、`magenta`、`cyan` 和 `blue`。

16.1.7.2. 对齐方式值

在用到对齐方式值的地方，Seam PDF支持下列水平对齐值：`left`、`right`、`center`、`justify` 和 `justifyall`。垂直对齐值为：`top`、`middle`、`bottom` 和 `baseline`。

16.1.8. iText配置

生成的文档无需其它的额外配置就可以使用。但是更加严格的应用程序则需要一些配置的要点。

默认的实现能够从一般的URL得到PDF文档，/seam-doc.seam。许多浏览器（和用户）更倾向于见到包含实际PDF文档名诸如 /myDocument.pdf的链接。这个功能需要一些配置。为了在浏览器中浏览PDF文件，所有的*.pdf资源都应该被映射到Seam Servlet过滤器以及DocumentStoreServlet：

```
<filter>
  <filter-name>Seam Servlet Filter</filter-name>
  <filter-class>org.jboss.seam.servlet.SeamServletFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>Seam Servlet Filter</filter-name>
  <url-pattern>*.pdf</url-pattern>
</filter-mapping>

<servlet>
  <servlet-name>Document Store Servlet</servlet-name>
  <servlet-class>org.jboss.seam.pdf.DocumentStoreServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>Document Store Servlet</servlet-name>
  <url-pattern>*.pdf</url-pattern>
</servlet-mapping>
```

文档存储组件中的 useExtensions 选项保证了文档存储时的扩展名和生成时的一致。

```
<components xmlns="http://jboss.com/products/seam/components"
  xmlns:pdf="http://jboss.com/products/seam/pdf">
  <pdf:documentStore useExtensions="true" />
</components>
```

生成的文档保存在当前对话范围中直到对话结束。此时，文档的引用就无效了。当对文档不存在时，你可以利用documentStore的 errorPage 属性指定要显示的默认视图。

```
<pdf:documentStore useExtensions="true" errorPage="/pdfMissing.seam" />
```

16.2. 图表

<p><p:barchart></p>	<p>描述</p> <p>显示柱状图。</p> <p>属性</p> <ul style="list-style-type: none"> • borderVisible — 控制图表边框是否要显示。 • borderPaint — 边框可见时的边框颜色。 • borderBackgroundPaint — 图表的默认背景色。
---------------------------	--

- `borderStroke` —
- `domainAxisLabel` — 域坐标轴的文本标签。
- `domainAxisPaint` — 域坐标轴标签的颜色。
- `domainGridlinesVisible` — 是否在图表上显示域坐标上的网格。
- `domainGridlinePaint` — 域坐标上网格可见时的颜色。
- `domainGridlineStroke` — 域坐标轴上网格线可见时的画笔风格。
- `height` — 图表的高度。
- `width` — 图表的宽度。
- `is3D` — 表示图表应该以3D而不是2D渲染的一个布尔值。
- `legend` — 表示图表中是否应该显示图例的一个布尔值。
- `legendItemPaint` — 图例中文本标签的默认颜色。
- `legendItemBackgroundPaint` — 图例的背景色与图表的背景色不一致时的颜色。
- `orientation` — 绘图方向，可以是
`<code>vertical</code>`
 （默认），也可以是
`<code>horizontal</code>`
 。
- `plotBackgroundPaint` — 绘图背景的颜色。
- `plotBackgroundAlpha` — 绘图区域背景的alpha（透明度）级别。 它应该是0（完全透明）到1（完全不透明）之间的一个数字。
- `plotForegroundAlpha` — 绘图区域的alpha（透明度）级别。 它应该是0（完全透明）到1（完全不透明）之间的一个数字。
- `plotOutlinePaint` — 绘图区域边线可见时的颜色。
- `plotOutlineStroke` — 绘图区域边线可见时的画笔风格。
- `rangeAxisLabel` — 值坐标轴上的文本标签。
- `rangeAxisPaint` — 值坐标轴上的标签颜色。
- `rangeGridlinesVisible` — 是否显示值坐标轴上的网格。
- `rangeGridlinePaint` — 值坐标轴上网格可见时的颜色。
- `rangeGridlineStroke` — 值坐标轴上网格线可见时的画笔风格。
- `title` — 图表标题文字。

	<ul style="list-style-type: none"> • titlePaint— 图表标题文字的颜色。 • titleBackgroundPaint— 图表标题文字的背景色。 • width — 图表的宽度。 <p>用法</p> <pre><p:barchart title="Bar Chart" legend="true" width="500" height="500"> <p:series key="Last Year"> <p:data columnKey="Joe" value="100" /> <p:data columnKey="Bob" value="120" /> </p:series> <p:series key="This Year"> <p:data columnKey="Joe" value="125" /> <p:data columnKey="Bob" value="115" /> </p:series> </p:barchart></pre>
--	--

<p:linechart>	<p>描述</p> <p>显示一个折线图。</p> <p>属性</p> <ul style="list-style-type: none"> • borderVisible — 控制是否显示整个图表的边框。 • borderPaint — 图表边框可见时的颜色。 • borderBackgroundPaint — 图表的默认背景色 • borderStroke — • domainAxisLabel — 域坐标轴的文本标签。 • domainAxisPaint — 域坐标轴标签的颜色。 • domainGridlinesVisible— 控制是否在图表上显示域坐标的网格。 • domainGridlinePaint— 域坐标上网格可见时的颜色。 • domainGridlineStroke — 域坐标轴上网格线可见时的画笔风格。 • height — 图表的高度。 • width — 图表的宽度。 • is3D — 表示应该以3D而不是2D图表渲染的一个布尔值。 • legend — 表示是否应该在图表中包括图例的一个布尔值。 • legendItemPaint— 图例中文本标签的默认颜色。
---------------	---

- `legendItemBackgroundPaint` — 图例的背景色与图表的背景色不一致时的颜色。
- `orientation` — 绘图区的方向，可以是
`<code>vertical</code>`
(默认)，也可以是
`<code>horizontal</code>`
- `plotBackgroundPaint` — 绘图区域的背景色。
- `plotBackgroundAlpha` — 绘图区域背景的alpha（透明度）级别。它应该是0（完全透明）到1（完全不透明）之间的一个数字。
- `plotForegroundAlpha` — 绘图区域的alpha（透明度）级别。它应该是0（完全透明）到1（完全不透明）之间的一个数字。
- `plotOutlinePaint` — 绘图区域边线可见时的颜色。
- `plotOutlineStroke` — 绘图区域边线可见时的画笔风格。
- `rangeAxisLabel` — 值坐标轴上的文本标签。
- `rangeAxisPaint` — 值坐标轴上的标签颜色。
- `rangeGridlinesVisible` — 控制是否在图表上显示值坐标轴的网格。
- `rangeGridlinePaint` — 值坐标轴上网格可见时的颜色。
- `rangeGridlineStroke` — 值坐标轴上网格线可见时的画笔风格。
- `title` — 图表的标题文字。
- `titlePaint` — 图表标题文字的颜色。
- `titleBackgroundPaint` — 图表标题的背景色。
- `width` — 图表的宽度。

用法

```
<p:linechart title="Line Chart"
  width="500" height="500">
  <p:series key="Prices">
    <p:data columnKey="2003" value="7.36" />
    <p:data columnKey="2004" value="11.50" />
    <p:data columnKey="2005" value="34.625" />
    <p:data columnKey="2006" value="76.30" />
    <p:data columnKey="2007" value="85.05" />
  </p:series>
</p:linechart>
```

<p:piechart>

描述

显示一个饼状图。

属性

- title 标题
- label 标签
- legend 图例
- is3D 是否3D
- labelLinkMargin 标签链接边距
- labelLinkPaint 标签链接颜色
- labelLinkStroke 标签链接画笔风格
- labelLinksVisible 标签链接是否可见
- labelOutlinePaint 标签外边线颜色
- labelOutlineStroke 标签外边线画笔风格
- labelShadowPaint 标签阴影颜色
- labelPaint 标签颜色
- labelGap 标签间隔
- labelBackgroundPaint 标签背景色
- startAngle 起始角度
- circular 圆形
- direction 方向
- sectionOutlinePaint 截面外边线颜色
- sectionOutlineStroke 截面外边线画笔风格
- sectionOutlinesVisible 截面外边线是否可见
- baseSectionOutlinePaint 基本截面外边线颜色
- baseSectionPaint 基本界面颜色
- baseSectionOutlineStroke 基本界面外边线画笔风格

用法



<code><p:series></code>	<p data-bbox="539 185 595 219">描述</p> <p data-bbox="480 257 1406 331">类数据可以分解成系列。series标签用于按照系列给一组数据分类，并且将样式应用到所有系列。</p> <p data-bbox="539 369 595 403">属性</p> <ul data-bbox="480 463 1374 846" style="list-style-type: none">• key — 系列名。• seriesPaint — 系列中每个项的颜色。• seriesOutlinePaint — 系列中每个项的外边线颜色。• seriesOutlineStroke — 系列中每个项所使用的画笔风格• seriesVisible — 表示系列是否显示的一个布尔值。• seriesVisibleInLegend — 表示是否在图例中列出系列的一个布尔值。 <p data-bbox="539 884 595 918">用法</p> <pre data-bbox="499 956 1198 1099"><p:series key="data1"> <ui:repeat value="{data.pieData1}" var="item"> <p:data columnKey="{item.name}" value="{item.value}" /> </ui:repeat> </p:series></pre>
-------------------------------	---

<code><p:data></code>	<p data-bbox="539 1247 595 1281">描述</p> <p data-bbox="539 1319 1139 1352">该数据标签描述要在图表中显示的每个数据点。</p> <p data-bbox="539 1391 595 1424">属性</p> <ul data-bbox="480 1485 1362 2018" style="list-style-type: none">• key — 数据项的名称。• series — 系列名，当该标签没有内嵌在<code><p:series></code>中的时候。• value — 数字化的数据值。• explodedPercent — 对于饼状图，表示分离块的百分比大小。• sectionOutlinePaint — 对于柱状图，表示截面外边线的颜色。• sectionOutlineStroke — 对于柱状图，表示截面外边线的画笔风格。• sectionPaint — 对于柱状图，表示截面的颜色。 <p data-bbox="539 2056 595 2089">用法</p>
-----------------------------	--

	<pre> <p:data key="foo" value="20" sectionPaint="#111111" explodedPercent=".2" /> <p:data key="bar" value="30" sectionPaint="#333333" /> <p:data key="baz" value="40" sectionPaint="#555555" sectionOutlineStroke="my-dot-style" /> </pre>
--	--

<p:color>	<p>描述</p> <p>颜色组件声明一种颜色或者一组渐变的颜色，可以在绘制填充图形时使用。</p> <p>属性</p> <ul style="list-style-type: none"> • color — 颜色值，对于渐变的颜色，这表示起始的颜色值。 第 16.1.7.1 节 “颜色值” • color2 — 对于渐变的颜色，这表示结束渐变的颜色值。 • point — 渐变颜色的起始坐标 • point2 — 渐变颜色的结束坐标 <p>用法</p> <pre> <p:color id="foo" color="#0ff00f"/> <p:color id="bar" color="#ff00ff" color2="#00ff00" point="50 50" point2="300 300"/> </pre>
-----------	--

<p:stroke>	<p>描述</p> <p>描述一种用来在图表中画线的画笔风格。</p> <p>属性</p> <ul style="list-style-type: none"> • width — 画笔的宽度。 • cap — 线端类型，有效值为 butt、round 和 square • join — 线交汇点的类型，有效值为 miter, round 和 bevel • miterLimit — 线边缘交汇，这个值限制交汇点的大小。 • dash — 设置要用来画线的虚线模式的虚线值。整数代表交替画实线和虚线的长度。 • dashPhase — 表示虚线模式中实线的偏移量。 <p>用法</p>
------------	---

	<div><p:stroke id="dot2" width="2" cap="round" join="bevel" dash="2 3" /></div>
--	---

16.3. 柱状图编码

...

<p:barcode>	描述
	属性
	<ul style="list-style-type: none">• type 类型• code 编码• xpos 横坐标• ypos 纵坐标• rotDegrees 旋转角度• barHeight 高度• textSize 文本大小• minBarWidth 最小柱状图宽度• barMultiplier
	用法 <div></div>

16.4. 更详细的文档

关于iText的更多信息，请见：

- [iText 主页](#)
- [iText in Action](#)

第 17 章 电子邮件

Seam现在包含了一个用于模板和发送邮件的可选组件。

邮件支持是由 `jboss-seam-mail.jar` 提供的。这个JAR包包括用于创建邮件的mail JSF控件，以及mailSession 管理组件。

examples/mail项目包括一份实用的email支持示例。该例子示范了恰当的打包方式，并且包含了一些当前支持的关键特性。

你也可以使用Seam的集成测试环境来测试你的mail程序，参见 第 31.3.2 节 “Seam Mail集成测试”。

17.1. 创建一条消息

为了使用Seam Mail，你并不需要完整的学习一门模板语言——一封邮件仅仅是一个facelet！【Facelet是用来建立JSF应用程序时的一个可供选择的表現层技术】

```
<m:message xmlns="http://www.w3.org/1999/xhtml"
  xmlns:m="http://jboss.com/products/seam/mail"
  xmlns:h="http://java.sun.com/jsf/html">

  <m:from name="Peter" address="peter@example.com" />
  <m:to name="#{person.firstname} #{person.lastname}"#{person.address}</m:to>
  <m:subject>Try out Seam!</m:subject>

  <m:body>
    <p><h:outputText value="Dear #{person.firstname}" /></p>
    <p>You can try out Seam by visiting
    <a href="http://labs.jboss.com/jbossseam">http://labs.jboss.com/jbossseam</a>.</p>
    <p>Regards,</p>
    <p>Pete</p>
  </m:body>

</m:message>
```

`<m:message>` 标签包装整个消息，并且通知Seam开始渲染一封邮件。在这个 `<m:message>` 标签内，我们使用标签 `<m:from>` 来设置这个消息是来自谁，使用标签 `<m:to>` 来标识发送者（注意我们就和在普通的facelet里一样使用EL），和 `<m:subject>` 标签。

`<m:body>` 标签包装邮件的主体。你可以像JSF组件那样将正规的HTML标签用在邮件主体内。

好，现在你已经有了email的模板，你将如何发送它呢？在 `m:message` 的结尾，`mailSession` 将被调用，用于发送邮件。所以，你所有要做的仅仅是请求Seam渲染这个视图：

```
@In(create=true)
private Renderer renderer;

public void send() {
    try {
        renderer.render("/simple.xhtml");
        facesMessages.add("Email sent successfully");
    }
    catch (Exception e) {
        facesMessages.add("Email sending failed: " + e.getMessage());
    }
}
```

```
}
```

假如：你输入了一个无效的email地址，将会抛出一个异常。该异常将被捕捉并显示给用户。

17.1.1. 附件

Seam中邮件添加附件的操作变得轻而易举。在处理文件时，它支持绝大多数的标准Java类型。

如果你想通过邮件发送 `jboss-seam-mail.jar`：

```
<m:attachment value="/WEB-INF/lib/jboss-seam-mail.jar"/>
```

Seam将通过classpath加载文件，并将其附件加入到到邮件中。默认情况下，它将被像 `jboss-seam-mail.jar` 一样加载；如果你想为它添加别名，只需要添加 `fileName` 属性即可。

```
<m:attachment value="/WEB-INF/lib/jboss-seam-mail.jar" fileName="this-is-so-cool.jar"/>
```

你同样可以附加 `java.io.File`、`java.net.URL`：

```
<m:attachment value="#{numbers}" />
```

也可以是 `byte[]` 或是 `java.io.InputStream`：

```
<m:attachment value="#{person.photo}" contentType="image/png"/>
```

你会注意到对于 `byte[]` 和 `java.io.InputStream`，你需要指定附件的MIME类型（因为这两种文件不帶此类信息。）

更好的是，你可以附Seam产生的PDF或任意标准的JSF视图，只需将你使用的普通标签用

`<m:attachment>` 封装起来即可：

```
<m:attachment fileName="tiny.pdf">
  <p:document>
    A very tiny PDF
  </p:document>
</m:attachment>
```

如果你想将多个文件添加到附件中（例如从数据库加载的一套照片），你只需要使用 `<ui:repeat>`

:

```
<ui:repeat value="#{people}" var="person">
  <m:attachment value="#{person.photo}" contentType="image/jpeg" fileName="#{person.firstname}_{person.lastname}.jpg"/>
</ui:repeat>
```

如果你想直接显示一个附上的图片：

```
<m:attachment
  value="#{person.photo}"
  contentType="image/jpeg"
  fileName="#{person.firstname}_{person.lastname}.jpg"
  status="personPhoto"
  disposition="inline" />
```



```

```

你可能会问 `cid:#{...}` 的作用是什么。是这样的，IETF明确规定将这个标签加入作为你图片的 `src`（源文件），当试着定位图片（Content-ID必须匹配）时，它就能够被查找到。— 多么神奇！

在访问状态对象之前你必须声明附件。

17.1.2. HTML/Text 交替部分

尽管现在绝大多数的邮件查看器都支持HTML格式的邮件，但还是有一些不支持，所以你可以在邮件体里添加一个无格式的文本作为替换。

```
<m:body>
  <f:facet name="alternative">Sorry, your email reader can't show our fancy email,
  please go to http://labs.jboss.com/jbossseam to explore Seam.</f:facet>
</m:body>
```

17.1.3. 多个收件人

很多时候你希望向一个收件组（比如你的用户们）发送邮件。所有的收件人标签可以被放在一个 `<ui:repeat>` 标签中：

```
<ui:repeat value="#{allUsers}" var="user">
  <m:to name="#{user.firstname} #{user.lastname}" address="#{user.emailAddress}" />
</ui:repeat>
```

17.1.4. 多条信息

有时候，你需要向每一个收件人发送一条稍微有差别的信（例如：重设密码）。最好的方法就是将整个信息放在 `<ui:repeat>` 标签中：

```
<ui:repeat value="#{people}" var="p">
  <m:message>
    <m:from name="#{person.firstname} #{person.lastname}">#{person.address}</m:from>
    <m:to name="#{p.firstname}">#{p.address}</m:to>
    ...
  </m:message>
</ui:repeat>
```

17.1.5. 模板

邮件模板示例显示（facelets模板）可以和Seam的mail标签很好的结合。

我们的 `template.xhtml` 包括：

```
<m:message>
  <m:from name="Seam" address="do-not-reply@jboss.com" />
  <m:to name="#{person.firstname} #{person.lastname}">#{person.address}</m:to>
  <m:subject>#{subject}</m:subject>
  <m:body>
    <html>
      <body>
```

```

        <ui:insert name="body">This is the default body, specified by the template.</ui:insert>
    </body>
</html>
</m:body>
</m:message>

```

我们的 `templating.xhtml` 包括：

```

<ui:param name="subject" value="Templating with Seam Mail"/>
<ui:define name="body">
    <p>This example demonstrates that you can easily use <i>facelets templating</i> in email!</p>
</ui:define>

```

你也可以在你的邮件中使用facelet的源标签，但你必须将它们置于一个jar包中并放在 `WEB-INF/lib` 目录下 — 当使用Seam Mail从 `web.xml` 引用 `.taglib.xml` 并不可靠。（因为如果你异步的发送你的邮件，Seam Mail无法访问到完整的JSF或Servlet上下文，所以并不知道 `web.xml` 的配置参数）

发送邮件时，如果你需要更多的配置Facelets或JSF，你需要重载Renderer组件，并且程式地做配置工作 — 仅限于高级用户。

17.1.6. 国际化

Seam支持发送国际化的信息。默认情况下，使用JSF提供的编码，但也可以由如下的模板重写：

```

<m:message charset="UTF-8">
    ...
</m:message>

```

邮件内容、主题和收件人（和发件人）的名称都会被编码。通过设置模板的编码，你需要确认facelets是否使用了正确的编码方式来解析你的页面。

```

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

```

17.1.7. 其它的标识头

有时候你会想在邮件上添加其他的头信息。Seam提供了一部分支持（请看 第 17.5 节 “标签”）。例如：我们可以设置邮件的重要程度，或着请求一个阅读回执。

```

<m:message xmlns:m="http://jboss.com/products/seam/mail"
    importance="low"
    requestReadReceipt="true"/>

```

另外你也可以通过使用 `<m:header>` 标签，为消息添加其它任意的头信息。

```

<m:header name="X-Sent-From" value="JBoss Seam"/>

```

17.2. 接收邮件

如果你正在使用EJB，你可以使用MDB（消息驱动Bean:Message Driven Bean）来接收消息。JBoss提供JCA适配器——mail-ra.rar 但是跟随JBoss发布的版本有一定的限制（某些版本没有做捆绑）。因此我们建议采用跟随推荐的Seam发布的 mail-ra.rar（不在Seam包的 mail 目录）。mail-ra.rar 应该被放置在 \$JBoss_HOME/server/default/deploy 目录下；如果你正在使用的JBoss版本已经有了这个文件，就替换了它。

您可以向这样配置：

```
@MessageDriven(activationConfig={
    @ActivationConfigProperty(propertyName="mailServer", propertyValue="localhost"),
    @ActivationConfigProperty(propertyName="mailFolder", propertyValue="INBOX"),
    @ActivationConfigProperty(propertyName="storeProtocol", propertyValue="pop3"),
    @ActivationConfigProperty(propertyName="userName", propertyValue="seam"),
    @ActivationConfigProperty(propertyName="password", propertyValue="seam")
})
@ResourceAdapter("mail-ra.rar")
@Name("mailListener")
public class MailListenerMDB implements MailListener {

    @In(create=true)
    private OrderProcessor orderProcessor;

    public void onMessage(Message message) {
        // Process the message
        orderProcessor.process(message.getSubject());
    }

}
```

每一个接收到的消息都将导致 onMessage(Message message) 被调用。大多数Seam的注释将会在MDB内部运行，但你不可以访问持久上下文。

在链接 <http://wiki.jboss.org/wiki/Wiki.jsp?page=InboundJavaMail> 的 mail-ra.rar 上你可以找到更多的信息。

如果你没有使用JBoss，你依然可以使用 mail-ra.rar，或许你可以在你的程序服务器上找到类似的适配器。

17.3. 配置

为了在你的应用程序中能够使用电子邮件，要确保 jboss-seam-mail.jar 包含在 WEB-INF/lib 目录中。如果你在使用JBoss AS，则使用Seam的邮件支持不需要做更多的配置工作了。否则你可能需要确认你是否有JavaMail的API，一个可供使用的JavaMail API的实现（JBoss AS中使用的API和实现正如作为 lib/mail.jar 跟随Seam发布的包），和一份Java Activation Framework的拷贝（作为 lib/activation.jar 跟随Seam发布）。

Seam的Email模块需要Facelets作为视图技术。将来库的版本可能会添加对JSP的支持。另外，它需要用到seam-ui包。

mailSession 组件使用JavaMail就像与‘真实的’SMTP服务器通讯。

17.3.1. mailSession

如果你在使用JEE环境工作，可以通过JNDI查找可用的JavaMail Session，你也可以使用Seam配置好的Session。

在 第 28.8 节 “与邮件相关的组件” 中有关于邮件会话组件属性的详细介绍。

17.3.1.1. 在JBoss AS中查找JNDI

JBossAS deploy/mail-service.xml 配置JavaMail会话捆绑到JNDI。 你需要修改默认的服务配置再应用到你的网络中。这里描述了更加详细的服务

<http://wiki.jboss.org/wiki/Wiki.jsp?page=JavaMail>

```
<components xmlns="http://jboss.com/products/seam/components"
  xmlns:core="http://jboss.com/products/seam/core"
  xmlns:mail="http://jboss.com/products/seam/mail">

  <mail:mail-session session-jndi-name="java:/Mail"/>

</components>
```

这里我们告诉Seam在JNDI中是通过 java:/Mail 来获得邮件Session的。

17.3.1.2. Seam配置会话

邮件会话可以通过 components.xml 配置来访问的。 这里我们告诉Seam使用 smtp.example.com 作为SMTP服务器。

```
<components xmlns="http://jboss.com/products/seam/components"
  xmlns:core="http://jboss.com/products/seam/core"
  xmlns:mail="http://jboss.com/products/seam/mail">

  <mail:mail-session host="smtp.example.com"/>

</components>
```

17.4. Meldware

Seam的邮件示例采用Meldware（来自 buni.org）作为邮件服务器。 Meldware是提供 SMTP、POP3、IMAP、WebMail、共享日历和图形化的管理工具的于一身的软件； 它是作为一个JEE应用程序编写的，因此可以和你的Seam程序一起部署到JBoss上。

和Seam一起分发的Meldware的版本（在文件夹mail/buni-meldware）为了开发都被特别修改过 — 邮箱、用户和别名（邮件地址）在每次程序部署的时候创建。 如果你希望在产品中不仅仅使用Meldware发送邮件，建议你使用vanilla拷贝。 你也可以使用 meldware 组件来创建邮箱，用户和别名等。

```
<components xmlns="http://jboss.com/products/seam/components"
  xmlns:core="http://jboss.com/products/seam/core"
  xmlns:mail="http://jboss.com/products/seam/mail">

  <mail:mail-session host="smtp.example.com"/>

  <mail:meldware>
    <mail:users>
      <value>#{duke}</value>
      <value>#{root}</value>
    </mail:users>

  </mail:meldware>

</components>
```

```

</mail:meldware>

    <mail:meldware-user name="duke" username="duke" password="duke">
        <mail:aliases>
            <value>duke@jboss.org</value>
            <value>duke@jboss.com</value>
        </mail:aliases>
    </mail:meldware-user name="root" username="root" password="root" administrator="true" />
</components>

```

这里我们创建了两个用户，拥有两个邮件地址的 duke 和名为 root 的管理员。

17.5. 标签

邮件通过使用命名空间 `http://jboss.com/products/seam/mail` 的标签生成。文档中在消息的根部通常应该有 `message` 标签，`message` 标签使 Seam 准备生成一封邮件。

标准的 facelets 的模板标签可以如同往常一样地使用。你可以在主体内部使用任何 JSF 标签；如果需要访问外部资源（stylesheets、javascript），那么就要确认是否设置了 `urlBase`。

<m:message>

邮件消息的根标签

- `importance` — 低、正常或是高。默认是正常，这是设置邮件消息重要程度的标签。
- `precedence` — 设置消息的优先级（例如：突出）
- `requestReadReceipt` — 默认是 `false`，如果设置，将会添加阅读回执，阅读回执将会被发给 `From:` 地址。
- `urlBase` — 如果设置，预设的 `requestContextPath` 将允许你在邮件中使用形如 `<h:graphicImage>` 的组件。

<m:from>

设置邮件的发件地址。每封邮件只允许有一个这样的值。

- `name` — 邮件应该来自的名称。
- `address` — 邮件应该来自的地址。

<m:replyTo>

设置回复地址给邮件。每封邮件同样只能有一个这样的值。

- `address` — 邮件来源的地址。

<m:to>

添加一个收件人到邮件。有多个收件人时使用复合的 `<m:to>` 标签。这个标签可以被安全的放置在重复标签 `<ui:repeat>` 之类。

- `name` — 收件人的名字。

- address — 收件人的地址。

<m:cc>

添加抄送地址到邮件。有多个抄送地址时使用复合的<m:cc>标签。这个标签可以被安全的放置在重复标签<ui:repeat>之类中。

- name — 收件人的名字。
- address — 收件人的邮件地址。

<m:bcc>

添加一个秘文抄送人到邮件。有多个秘密抄送地址时使用复合的<m:bcc>标签。 这个标签可以被安全的放置在重复标签<ui:repeat>之类中。

- name — 收件人的名字。
- address — 收件人的邮件地址。

<m:header>

向邮件添加一个头（例如：X-Sent-From: JBoss Seam）。

- name — 要添加的头的名字（例如：X-Sent-From）。
- value — 要添加的头的值（例如：JBoss Seam）。

<m:attachment>

添加一个附件到邮件。

- value — 要添加的附件：
 - String — 在classpath中一个 String 作为到文件的路径被解析。
 - java.io.File — 一个指向 File 对象的EL表达式。
 - java.net.URL — 一个指向URL对象的EL表达式。
 - java.io.InputStream — 一个指向 InputStream 类型的EL表达式。 这种情况下，fileName 和 contentType 都必须指定。
 - byte[] — 一个指向 byte[] 类型的EL表达式。 这种情况下，fileName 和 contentType 都必须指定。

如果值属性被省略：

- 如果这个标签包含一个 <p:document> 标签，这个被描述的文档将会被生成并且附加到邮件上。 fileName 应该被指定。
- 如果这个标签包含其它的JSF标签，将会通过它们生成HTML文档并附加到邮件。 fileName应该被指定。

- `fileName` — 指定可供使用的已经被附上的文件。
- `contentType` — 指定已附上的文件的MIME类型。

<m:subject>

设置邮件主题。

<m:body>

设置邮件主体。支持 `alternative facet`。 比如生成的一个HTML邮件可能包含针对不支持html的读者的备选的文本。

- `type` — 如果设为 `plain`，将会生成一份简单文本邮件，否则将会生成一份HTML邮件。

第 18 章 异步和消息

Seam使得异步执行一个来自Web请求的工作变得非常容易。当大多数人在Java EE里考虑异步时，他们想到用JMS。在Seam中，这确实是一种解决方案，当你有严格和明确定义的QoS服务需求时，这是正确的。Seam利用Seam组件让发送和接收JMS消息更容易进行。

但是对于多数用例来说，用JMS无异于杀鸡用牛刀。Seam将简单的异步方法和事件应用分层，置于你选择的 dispatchers 之上。

- `java.util.concurrent.ScheduledThreadPoolExecutor` （默认）
- EJB Timer Service （针对 EJB 3.0 环境）
- Quartz

18.1. 异步

异步的事件和方法调用与底层的分配机制有着相同的服务期待质量。基于 `ScheduledThreadPoolExecutor` 的默认dispatcher执行得很好，但不提供对持久化异步任务的支持，因此不保证一项任务真正会被执行。如果你在一个支持EJB 3.0的环境中工作，并将下面这一行添加到 `components.xml` 中：

```
<async:timer-service-dispatcher/>
```

那么，你的异步任务将由容器的EJB定时服务处理。如果你不熟悉Timer服务，也不必担心，如果你想要在Seam中使用异步方法，并不需要与它直接交互。要了解一件重要的事情：任何好的EJB 3.0实现都将有使用持久化定时器的选择，它为任务最终得到处理提供了一些保证。

另一种选择是使用开源的Quartz库来管理异步的方法。你要将Quartz库JAR（在 `lib` 路径中）绑定在你的EAR中，并在 `application.xml` 中将它声明成一个Java模块。另外，你还需要将下面的行添加到 `components.xml` 中来安装Quartz Dispatcher。

```
<async:quartz-dispatcher/>
```

Seam的API对于默认的 `ScheduledThreadPoolExecutor` 的Seam API，及EJB3 Timer与Quartz Scheduler 大体相同。它们可以只是通过在 `components.xml` 中添加一行来进行”即插即用（plug and play）“。

18.1.1. 异步方法

最简单的形式，一个异步的调用只是异步地处理来自访问者的方法调用（在不同的线程中）。当我们要返回一个即时响应给客户端时，通常使用一个异步调用，并让一些费时的工作在后台处理。此模式在使用AJAX的应用程序中运行良好，在AJAX应用中客户端能够自动地从服务器上获得工作结果。

对于EJB组件，我们在本地接口上进行注解，来指定某个方法要被异步地处理。

```
@Local
public interface PaymentHandler
{
    @Asynchronous
```



```

public void processPayment(Payment payment);
}

```

（对于JavaBean组件，如果喜欢的话，我们可以注解组件实现类。）

异步的使用对于Bean类来说是透明的：

```

@Stateless
@Name("paymentHandler")
public class PaymentHandlerBean implements PaymentHandler
{
    public void processPayment(Payment payment)
    {
        //do some work!
    }
}

```

并且对客户端也是透明的：

```

@Stateful
@Name("paymentAction")
public class CreatePaymentAction
{
    @In(create=true) PaymentHandler paymentHandler;
    @In Bill bill;

    public String pay()
    {
        paymentHandler.processPayment( new Payment(bill) );
        return "success";
    }
}

```

异步方法在一个全新的事件上下文中处理，而且无法访问调用者的会话或对话上下文状态。然而，业务流程上下文 得到了 传播。

异步方法调用可以利用 `@Duration`、`@Expiration` 和 `@IntervalDuration`注解为后续的执行定时。

```

@Local
public interface PaymentHandler
{
    @Asynchronous
    public void processScheduledPayment(Payment payment, @Expiration Date date);

    @Asynchronous
    public void processRecurringPayment(Payment payment, @Expiration Date date, @IntervalDuration Long interval)
}

```

```

@Stateful
@Name("paymentAction")
public class CreatePaymentAction
{
    @In(create=true) PaymentHandler paymentHandler;
    @In Bill bill;

    public String schedulePayment()
    {
        paymentHandler.processScheduledPayment( new Payment(bill), bill.getDueDate() );
        return "success";
    }
}

```

```

public String scheduleRecurringPayment()
{
    paymentHandler.processRecurringPayment( new Payment(bill), bill.getDueDate(), ONE_MONTH );
    return "success";
}
}

```

客户端和服务端两者都可以访问与调用相关联的 `Timer` 对象。当使用EJB3 Dispatcher时，The `Timer` 对象会显示在下面。对于默认的`ScheduledThreadPoolExecutor`，返回的是JDK的对象`Future`。对于`Quartz Dispatcher`，返回`QuartzTriggerHandle`，我们会在下部分对此进行讨论。

```

@Local
public interface PaymentHandler
{
    @Asynchronous
    public Timer processScheduledPayment(Payment payment, @Expiration Date date);
}

```

```

@Stateless
@Name("paymentHandler")
public class PaymentHandlerBean implements PaymentHandler
{
    @In Timer timer;

    public Timer processScheduledPayment(Payment payment, @Expiration Date date)
    {
        //do some work!

        return timer; // 注意返回值被完全忽略
    }
}

```

```

@Stateful
@Name("paymentAction")
public class CreatePaymentAction
{
    @In(create=true) PaymentHandler paymentHandler;
    @In Bill bill;

    public String schedulePayment()
    {
        Timer timer = paymentHandler.processScheduledPayment( new Payment(bill), bill.getDueDate() );
        return "success";
    }
}

```

异步方法不能返回任何其它值给调用者。

18.1.2. 包含Quartz Dispatcher的异步方法

`Quartz dispatcher`（它的安装方法请见前文）允许你使用 `@Asynchronous`、`@Duration`、`@Expiration` 和 `@IntervalDuration` 注解。但它还有一些其他的强大功能。`Quartz dispatcher`还支持三种新注解。

`@FinalExpiration` 注解指定一个重现任务的终止日期。

```
// Defines the method in the "processor" component
@Asynchronous
public QuartzTriggerHandle schedulePayment(@Expiration Date when,
                                           @IntervalDuration Long interval,
                                           @FinalExpiration Date endDate,
                                           Payment payment)
{
    // do the repeating or long running task until endDate
}

... ..

// Schedule the task in the business logic processing code
// Starts now, repeats every hour, and ends on May 10th, 2010
Calendar cal = Calendar.getInstance ();
cal.set (2010, Calendar.MAY, 10);
processor.schedulePayment(new Date(), 60*60*1000, cal.getTime(), payment);
```

注意该方法返回 QuartzTriggerHandle 对象，你以后可以用它来中止、暂停和恢复定时器。

QuartzTriggerHandle 对象是可序列化的，因此，如果你需要保留更久一点，可以把它存到数据库中。

```
QuartzTriggerHandle handle =
    processor.schedulePayment(payment.getPaymentDate(),
                             payment.getPaymentCron(),
                             payment);
payment.setQuartzTriggerHandle( handle );
// Save payment to DB

// later ...

// Retrieve payment from DB
// Cancel the remaining scheduled tasks
payment.getQuartzTriggerHandle().cancel();
```

@IntervalCron 注解支持Unix cron语法的任务调度。例如，下面的异步方法在三月份每周三的2:10pm和2:44pm运行。

```
// Define the method
@Asynchronous
public QuartzTriggerHandle schedulePayment(@Expiration Date when,
                                           @IntervalCron String cron,
                                           Payment payment)
{
    // do the repeating or long running task
}

... ..

// Schedule the task in the business logic processing code
QuartzTriggerHandle handle =
    processor.schedulePayment(new Date(), "0 10,44 14 ? 3 WED", payment);
```

@IntervalBusinessDay 注解支持在“第n个Business Day”调用。例如，下面的异步方法在每个月第2个business day的14:00运行。默认时，它从business day中排除了2010年之前的所有周末和米国联邦假期。

```
// Define the method
@Asynchronous
```

```

public QuartzTriggerHandle schedulePayment(@Expiration Date when,
                                           @IntervalBusinessDay NthBusinessDay nth,
                                           Payment payment)
{
    // do the repeating or long running task
}

... ..

// Schedule the task in the business logic processing code
QuartzTriggerHandle handle =
    processor.schedulePayment(new Date(),
                              new NthBusinessDay(2, "14:00", WEEKLY), payment);

```

NthBusinessDay 对象包含调用触发器的配置。你可以通过 additionalHolidays 属性指定更多的假期（例如，公司假期、非美国的假期等等。）

```

public class NthBusinessDay implements Serializable
{
    int n;
    String fireAtTime;
    List <Date> additionalHolidays;
    BusinessDayIntervalType interval;
    boolean excludeWeekends;
    boolean excludeUsFederalHolidays;

    public enum BusinessDayIntervalType { WEEKLY, MONTHLY, YEARLY }

    public NthBusinessDay ()
    {
        n = 1;
        fireAtTime = "12:00";
        additionalHolidays = new ArrayList <Date> ();
        interval = BusinessDayIntervalType.WEEKLY;
        excludeWeekends = true;
        excludeUsFederalHolidays = true;
    }
    ... ..
}

```

@IntervalDuration、@IntervalCron 和 @IntervalNthBusinessDay 注解相互排斥。如果把它们用在同一个方法中，就会抛出 RuntimeException。

18.1.3. 异步事件

组件驱动的事件也可以是异步的。为了给异步处理提出事件，只要调用 Events 类的 raiseAsynchronousEvent() 方法就可以了。要安排一个定时的事件，要调用 raiseTimedEvent() 的一个方法，并传递一个 schedule 对象（对于默认的dispatcher或者定时服务dispatcher，要使用 TimerSchedule）。组件可以用正常方式观察异步事件，但是要记住，只有业务处理上下文才被传播到异步线程上。

18.2. Seam中的消息

Seam让JMS消息发送到Seam组件和从Seam组件接收变得很容易。

18.2.1. 配置

为了给发送JMS消息配置Seam的基础结构，你需要告诉Seam关于任何你想发送消息到的主题（Topic）和队列（Queue），并且也要告诉Seam到哪里寻找 `QueueConnectionFactory` 和 / 或 `TopicConnectionFactory`。

Seam默认使用 `UIL2ConnectionFactory`，它是使用JBossMQ时常用的连接工厂。如果你正使用其他的JMS提供者，就需要在 `seam.properties`、`web.xml` 或 `components.xml` 文件中设置一个或两个 `queueConnection.queueConnectionFactoryJndiName` 和 `topicConnection.topicConnectionFactoryJndiName`。

你也需要在 `components.xml` 文件中列出主题（Topic）和队列（Queue），来安装Seam受控的 `TopicPublisher` 和 `QueueSender`：

```
<jms:managed-topic-publisher name="stockTickerPublisher" auto-create="true" topic-jndi-name="topic/stockTickerTopic"/>
<jms:managed-queue-sender name="paymentQueueSender" auto-create="true" queue-jndi-name="queue/paymentQueue"/>
```

18.2.2. 发送消息

现在，你可以注入一个JMS `TopicPublisher` 和 `TopicSession` 到任何组件里：

```
@In
private TopicPublisher stockTickerPublisher;
@In
private TopicSession topicSession;

public void publish(StockPrice price) {
    try
    {
        stockTickerPublisher.publish( topicSession.createObjectMessage(price) );
    }
    catch (Exception ex)
    {
        throw new RuntimeException(ex);
    }
}
```

或用来同Queue一起使用

```
@In
private QueueSender paymentQueueSender;
@In
private QueueSession queueSession;

public void publish(Payment payment) {
    try
    {
        paymentQueueSender.send( queueSession.createObjectMessage(payment) );
    }
    catch (Exception ex)
    {
        throw new RuntimeException(ex);
    }
}
```

18.2.3. 利用消息驱动Bean接收消息

你可以利用任何EJB3消息驱动Bean来处理消息。消息驱动Bean甚至可以是Seam组件，在这种情况下，它可能注入其他事件和应用程序作用域的Seam组件。

18.2.4. 在客户端接收消息

Seam Remoting允许你在客户端的JavaScript代码中订阅JMS主题（Topic）。这个在下一章里讲述。

第 19 章 缓存

数据库成为了大多数企业应用的主要瓶颈，也成为了运行环境中最不具伸缩性的层。PHP/Ruby的用户会说什么都不共享（share nothing）的架构照样具有很好的伸缩性。从表面上看也许是对的，可惜我不知道是否存在这样的多用户应用，其实现是能够在集群的不同结点间不共享资源。这些傻瓜真正想的是“除了数据库以外什么都不共享(Share nothing except for the database)”的架构。当然，共享数据库是多用户应用伸缩性的主要问题——因此声称这样的架构具有高伸缩性是荒谬的，你可要知道它们花费了这些人的大部分时间。

通常，几乎所有通过共享数据库做的事情并不值得这样做。

这就是缓存（Cache）产生的原因。嗯，当然并不只是一个缓存。一个设计良好的Seam应用将具有丰富的多层缓存策略，这也影响着应用的每一层：

- 当然，数据库有它自己的缓存，这是超级重要的，但是它不能像应用层的缓存一样具有伸缩性。
- 对从数据库提取出的数据，你的ORM解决方案（Hibernate，或者别的JPA实现）具有两级缓存。这是一种很强大的能力，但是经常被误用。在一个集群环境里，保持缓存中的数据在整个集群中具有事务一致性，并且和数据库一致，其代价是相当昂贵的。这对于共享在多个用户间，且很少被更新的数据最有意义。在传统的无状态架构里，人们经常使用二级缓存来保存会话状态。这种做法总是糟糕的，在Seam中更是大错特错的。
- Seam会话上下文是会话状态的缓存。存储于会话上下文中的组件可以保持并缓存与当前用户交互相关的状态。
- 特别的，Seam管理的持久化上下文（或者一个扩展受管EJB容器持久化上下文，它与会话范围的无状态会话Bean相关）成为了当前会话中数据的缓存。这种缓存趋向于拥有一个相当高的命中率！Seam优化了集群环境中受管Seam持久化上下文的复制，也不需要保证数据库事务的一致性（乐观锁已足够），因此你不必担心这种缓存的性能问题，除非你把成千上万个对象读取到一个单独的持久化上下文中。
- 应用可以在Seam应用上下文中缓存非事务性状态。相应的，保存在应用上下文中的状态不能被集群中其它结点访问。
- 应用通过Seam的 `pojoCache` 组件可以缓存事务性状态，这个组件把JBossCache集成到了Seam环境中。如果你在集群模式下运行了JBossCache，那么这个状态是可以被别的结点访问的。
- 最后，Seam让你能够缓存生成的JSF页面的部分内容（rendered fragments）。与ORM的二级缓存不一样的是，当数据发生变化时，这种缓存不能自动的失效，因此你需要写应用代码来使它显式的失效，或者设置适当的过期策略。

如要获得更多关于二级缓存的信息，你可以参考你的ORM解决方案的文档，因为这是个极为复杂的话题。在这节中我们会直接讨论通过 `pojoCache` 组件使用JBossCache，或者通过 `<s:cache>`控制充当页面片段（page fragment）缓存。

19.1. 在Seam中使用JBossCache

内建的 `pojoCache` 组件用来管理 `org.jboss.cache.aop.PojoCache` 实例。你可以安全的把任何不可变的Java对象保存在缓存里，然后它将通过集群被复制（假设打开了复制）。如果你想在缓存里保存可变

的Java对象，你需要运行JBossCache字节码预处理器来确保这些对象的变化能够自动的被检测到并被复制。

为了使用 `pojoCache`，你需要做的就是将JBossCache jar文件放在classpath中，并提供一个名为 `treecache.xml` 的文件，它应该有一个合适的cache配置。JBossCache有很多让人觉得恐怖和迷惑的配置，因此我们不在这里讨论它们。请参考JBossCache文档来获得更多信息。

在 `examples/blog/resources/treecache.xml` 里有一个 `treecache.xml` 的范例。

对Seam的EAR部署，我们推荐把JBossCache jar文件和配置文件直接放到EAR里。确保你在EAR的lib目录里放置了 `jboss-cache.jar` 和 `jgroups.jar`。

现在你可以把缓存注入到任何Seam组件：

```
@Name("chatroom")
public class Chatroom {
    @In PojoCache pojoCache;

    public void join(String username) {
        try
        {
            Set<String> userList = (Set<String>) pojoCache.get("chatroom", "userList");
            if (userList==null)
            {
                userList = new HashSet<String>();
                pojoCache.put("chatroom", "userList", userList);
            }
            userList.put(username);
        }
        catch (CacheException ce)
        {
            throw new RuntimeException(ce);
        }
    }
}
```

如果你想在应用中使用多个JBossCache配置，需要使用 `components.xml`：

```
<core:pojo-cache name="myCache" cfg-resource-name="myown/cache.xml"/>
```

19.2. 页片段缓存

使用JBossCache最有趣的是 `<s:cache>` 标签，Seam用它来解决JSF的页片段缓存问题。`<s:cache>` 内部使用了 `pojoCache`，因此你在使用它时需要遵循上面列出的步骤。（把jar文件放到EAR里，并进行令人恐怖的配置选项等。）

`<s:cache>`被用来缓存一些不太变化的内容。例如我们blog欢迎页面显示的近期blog：

```
<s:cache key="recentEntries-#{blog.id}" region="welcomePageFragments">
    <h:dataTable value="#{blog.recentEntries}" var="blogEntry">
        <h:column>
            <h3>#{blogEntry.title}</h3>
            <div>
                <s:formattedText value="#{blogEntry.body}" />
            </div>
        </h:column>
    </h:dataTable>
</s:cache>
```


key使得每个页片段拥有多个缓存版本。在这种情形下，每个blog拥有一个缓存版本。region 决定了所有版本都将保存在哪个JBossCache结点里。不同结点可能有不同的过期策略。（这是你设置上述令人恐怖的配置选项做的事情。）

当然，<s:cache>的一个重要问题是它无法知道数据的变化（例如当博客作者发布一个新实体）。因此你需要自己管理被缓存的片段。

```
public void post() {  
    ...  
    entityManager.persist(blogEntry);  
    pojoCache.remove("welcomePageFragments", "recentEntries-" + blog.getId() );  
}
```

另一种方法是，如果不必把数据的变化立刻反馈给用户，那么你可以在JbossCache结点上设置一个短的过期时间。

第 20 章 Web Services

Seam与JBossWS整合，有助于标准的JEE Web Services完全利用Seam上下文框架的优势，包括支持对话的Web Services。本章概述帮助Web Services在Sema环境内部运行所需要的步骤。

20.1. 配置和打包

为了允许Seam拦截Web Service请求，以便能够为该请求创建必要的Seam上下文，必须配置一种特殊的SOAP处理器；`org.jboss.seam.webservice.SOAPRequestHandler` 是一种 `SOAPHandler` 实现，它完成在一个Web Service请求范围期间管理Seam生命周期的工作。将一个项目配置为使用这个处理器的最容易方法是，将一个称作 `standard-jaxws-endpoint-config.xml` 的文件放到包含Web Service类的 `jar` 文件的 `META-INF` 路径下。这个文件包含下列SOAP处理器配置：

```
<jaxws-config xmlns="urn:jboss:jaxws-config:2.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:javaee="http://java.sun.com/xml/ns/javaee"
  xsi:schemaLocation="urn:jboss:jaxws-config:2.0 jaxws-config_2_0.xsd">
  <endpoint-config>
    <config-name>Seam WebService Endpoint</config-name>
    <pre-handler-chains>
      <javaee:handler-chain>
        <javaee:protocol-bindings>##SOAP11_HTTP</javaee:protocol-bindings>
        <javaee:handler>
          <javaee:handler-name>SOAP Request Handler</javaee:handler-name>
          <javaee:handler-class>org.jboss.seam.webservice.SOAPRequestHandler</javaee:handler-class>
        </javaee:handler>
      </javaee:handler-chain>
    </pre-handler-chains>
  </endpoint-config>
</jaxws-config>
```

20.2. 对话的Web Services

那么对话在Web Service请求之间如何传播呢？Seam使用一个在SOAP请求和响应消息中都有的SOAP标头元素，将对话ID从消费者传输到服务，并再传回来。下面是一个包含对话ID的Web Service请求的例子：

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:seam="http://seambay.example.seam.jboss.org/">
  <soapenv:Header>
    <seam:conversationId xmlns:seam='http://www.jboss.org/seam/webservice'>2</seam:conversationId>
  </soapenv:Header>
  <soapenv:Body>
    <seam:confirmAuction/>
  </soapenv:Body>
</soapenv:Envelope>
```

就如你在上面的SOAP消息中所见，在包含请求的对话ID（在这个例子中是2）的SOAP头内部，有一个 `conversationId` 元素，遗憾的是，由于Web Services可能被以不同语言编写的不同Web Service客户端消费，因此在准备用于单个对话范围内的单独Web Services之间实现对话ID传播，由开发人员决定。

要注意的一件重要的事情是，conversationId 头元素必须满足 `http://www.jboss.org/seam/webservice` 的命名空间条件，否则Seam将无法从请求中读取对话ID。以下是对上述请求消息响应的一个例子：

```
<env:Envelope xmlns:env='http://schemas.xmlsoap.org/soap/envelope/'>
  <env:Header>
    <seam:conversationId xmlns:seam='http://www.jboss.org/seam/webservice'>2</seam:conversationId>
  </env:Header>
  <env:Body>
    <confirmAuctionResponse xmlns='http://seambay.example.seam.jboss.org/'/>
  </env:Body>
</env:Envelope>
```

如你所见，响应消息包含与请求相同的 conversationId 元素。

20.2.1. 建议策略

因为Web Services必须被实现为一个无状态的会话Bean或者POJO，对于对话的Web Services，建议用Web Service充当一个对话Seam组件的外观（facade）。



如果Web Service是作为无状态会话Bean编写的，那它也可能通过提供一个 @Name，把它变成一个Seam组件。这么做允许Seam的双向注入（和其它）特性在Web Service类自身中使用。

20.3. Web Service范例

让我们浏览一个Web Service的例子。本节中的代码全部来自Seam的 `/examples` 目录下的seamBay范例应用程序，并遵循前一节中所述的建议策略。让我们先来看看Web Service类和它的其中一个Web Service方法：

```
@Stateless
@WebService(name = "AuctionService", serviceName = "AuctionService")
public class AuctionService implements AuctionServiceRemote
{
  @WebMethod
  public boolean login(String username, String password)
```

```

{
    Identity.instance().setUsername(username);
    Identity.instance().setPassword(password);
    Identity.instance().login();
    return Identity.instance().isLoggedIn();
}

// snip
}

```

如你所见，我们的Web Service是一个无状态的会话Bean，并如JSR-181所规定的，利用 `javax.jws` 包中的JWS注解进行注解。`@WebService` 注解告诉容器，这个类实现一个Web Service，且 `login()` 方法中的 `@WebMethod` 注解将该方法当成是一个Web Service方法。`@WebService` 注解中的 `name` 和 `serviceName` 属性是可选的。

根据规范要求，要暴露作为Web Service方法的每个方法也都必须在Web Service类的远程接口进行声明（当Web Service是一个无状态会话Bean的时候）。在上述例子中，`AuctionServiceRemote` 接口必须声明 `login()` 方法，实际上就是注解 `@WebMethod`。

就如你可以在上述代码中所见，Web Service实现一个委托给Seam内建的 `Identity` 组件的 `login()` 方法。为了与我们的建议策略一致，Web Service是作为一个简单的外观编写的，将实际的工作委托给Seam组件。这样有助于最好地在Web Services和其他客户端之间重用业务逻辑。

让我们看另一个例子。这个Web Service方法通过委托给 `AuctionAction.createAuction()` 方法，开始了一个新对话。

```

@WebMethod
public void createAuction(String title, String description, int categoryId)
{
    AuctionAction action = (AuctionAction) Component.getInstance(AuctionAction.class, true);
    action.createAuction();
    action.setDetails(title, description, categoryId);
}

```

下面是来自 `AuctionAction` 的代码：

```

@Begin
public void createAuction()
{
    auction = new Auction();
    auction.setAccount(authenticatedAccount);
    auction.setStatus(Auction.STATUS_UNLISTED);
    durationDays = DEFAULT_AUCTION_DURATION;
}

```

从中我们可以看到，Web Services是如何通过充当外观，并将实际的工作委托给一个对话的Seam组件，来参与长运行对话的。

第 21 章 Remoting

Seam使用AJAX来为Web页面远程访问组件提供便捷方法。使用该框架几乎不需要预先的开发准备——你只需要在组件中增加简单的注解，就可以通过AJAX来访问你的组件了。本章描述了建立一个支持AJAX的Web页面所必须的步骤，然后用更多细节继续解释Seam Remoting框架的特性。

21.1. 配置

要使用Remoting，必须先在 `web.xml` 文件中配置Seam Resource Servlet。

```
<servlet>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <servlet-class>org.jboss.seam.servlet.SeamResourceServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <url-pattern>/seam/resource/*</url-pattern>
</servlet-mapping>
```

接下来是在Web页面引入必要的JavaScript。至少有两段脚本必须被引入。第一段脚本包含了支持远程功能的所有客户端框架代码。

```
<script type="text/javascript" src="seam/resource/remoting/resource/remote.js"></script>
```

第二段脚本包含了你希望调用的组件的存根和类型定义。它是根据组件的本地接口动态生成的，同时也包含了调用远程接口的方法时所用到的全部类的类型定义。脚本中的名称反射到组件的名称。例如，如果有一个标有 `@Name("customerAction")` 的无状态会话Bean，那么脚本标签应该类似于此：

```
<script type="text/javascript"
  src="seam/resource/remoting/interface.js?customerAction"></script>
```

如果想在同一个页面上访问多个组件，则需要把它们全部作为脚本标签的参数：

```
<script type="text/javascript"
  src="seam/resource/remoting/interface.js?customerAction&accountAction"></script>
```

或者，你也可以使用 `s:remote` 标签来引入这些必要的JavaScript，注意要通过逗号来分隔每个组件或者类的名称。

```
<s:remote include="customerAction,accountAction"/>
```

21.2. Seam对象

客户端通过 Seam JavaScript对象与你的组件进行交互。这个JavaScript对象在 `remote.js` 中定义，你将一直使用它来异步调用你的组件。它被划分成两个功能域：`Seam.Component` 包含了与组件一起工作的方法，`Seam.Remoting` 包含了执行远程请求的方法。熟悉这个对象的最简单方法是从一个简单的例

子开始。

21.2.1. Hello World示例

让我们从这个简单的示例中逐步弄清楚 Seam 对象是怎样工作的。首先，我们创建一个名为 `helloAction` 的新的Seam组件。

```
@Stateless
@Name("helloAction")
public class HelloAction implements HelloLocal {
    public String sayHello(String name) {
        return "Hello, " + name;
    }
}
```

同时需要为这个新组件创建一个本地接口 —— 特别要注意 `@WebRemote` 注解，因为它是能远程访问我们方法所必须的。

```
@Local
public interface HelloLocal {
    @WebRemote
    public String sayHello(String name);
}
```

这是我们所编写的所有服务器端代码。接下来就是我们的Web页面 —— 创建一个新的页面然后引入以下脚本：

```
<s:remote include="helloAction"/>
```

为了完成一个完整的用户交互体验，我们增加一个按钮：

```
<button onclick="javascript:sayHello()">Say Hello</button>
```

同时，我们还需要增加更多脚本以使得按钮在被点击后真正能够做出相应的反应：

```
<script type="text/javascript">
    /*

    function sayHello() {
        var name = prompt("What is your name?");
        Seam.Component.getInstance("helloAction").sayHello(name, sayHelloCallback);
    }

    function sayHelloCallback(result) {
        alert(result);
    }

    // ]]&gt;
&lt;/script&gt;</pre>
</div>
<div data-bbox="82 854 907 908" data-label="Text">
<p>到此已经全部完成了！部署这个应用程序并且浏览页面，点击按钮，按照提示输入一个名字，会出现一个hello消息框，这个消息框的出现表明了这次调用是成功的。如果你想节省时间，你可以从 Seam的 <code>/examples/remoting/helloworld</code> 目录中找到这个Hello World示例的所有源代码。</p>
</div>
<div data-bbox="119 921 917 940" data-label="Text">
<p>那我们的脚本到底做了什么呢？我们把它分解成更小的部分。开始你可以从列出的JavaScript代</p>
</div>
<div data-bbox="393 964 600 981" data-label="Page-Footer">Seam Framework (2.0GA)</div>
<div data-bbox="883 964 923 981" data-label="Page-Footer">206</div>
```

码中看到我们已经实现了两个方法 —— 第一个方法负责提示用户输入姓名，然后产生一个远程请求。看一下下面这行：

```
Seam.Component.getInstance("helloAction").sayHello(name, sayHelloCallback);
```

这一行的第一部分 `Seam.Component.getInstance("helloAction")`，返回了一个代理，或者 `helloAction` 组件的存根。我们可以通过这个存根调用组件的方法，这也是剩余部分即 `sayHello(name, sayHelloCallback)`；所做的事情。

这整行的代码是在调用我们组件的 `sayHello` 方法，并且传进来 `name` 作为参数。第二个参数 `sayHelloCallback` 并不是我们组件的 `sayHello` 方法的参数，相反它告诉Seam Remoting框架一旦它收到与请求对应的响应，则要把这个响应传递到JavaScript脚本的 `sayHelloCallback` 方法。这个回调参数是完全可选的，因此当你调用一个返回值为 `void` 方法或者你不关心结果时，你可以不使用这个参数。

一旦 `sayHelloCallback` 方法收到了远程请求的响应，就会弹出一个警告消息，以显示方法调用的结果。

21.2.2. Seam.Component

`Seam.Component` JavaScript对象提供了许多客户端方法来与Seam组件一起工作，它的两个主要方法是 `newInstance()` 和 `getInstance()`，将在下面的小节中讲解，它们的主要区别是 `newInstance()` 总是创建一个组件类型的新实例，而 `getInstance()` 却返回一个单例的实例。

21.2.2.1. Seam.Component.newInstance()

使用`newInstance()`方法来创建一个实体或者JavaBean组件的一个新实例。这个方法返回的对象将含有相同的获取 / 设置方法作为服务器端组成部分，或者你可以直接访问它的字段。以下面的Seam实体组件为例：

```
@Name("customer")
@Entity
public class Customer implements Serializable
{
    private Integer customerId;
    private String firstName;
    private String lastName;

    @Column public Integer getCustomerId() {
        return customerId;
    }

    public void setCustomerId(Integer customerId) {
        this.customerId = customerId;
    }

    @Column public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    @Column public String getLastName() {
        return lastName;
    }
}
```

```
public void setLastName(String lastName) {
    this.lastName = lastName;
}
}
```

为了创建一个客户端Customer对象，你要编写如下代码：

```
var customer = Seam.Component.newInstance("customer");
```

然后从这里起，你可以设置customer对象的字段：

```
customer.setFirstName("John");
// Or you can set the fields directly
customer.lastName = "Smith";
```

21.2.2.2. Seam.Component.getInstance()

使用 `getInstance()` 方法来获得一个Seam会话Bean组件的存根的引用，这个存根可以用来远程执行组件的方法。这个方法返回特定组件的单例，因此使用同样的组件名调用它两次也将返回该组件的同一个实例。

接着看我们先前的例子，如果我们创建了一个新的 `customer`，同时我们想保存它，我们需要把它传递给 `customerAction` 组件的 `saveCustomer()` 方法：

```
Seam.Component.getInstance("customerAction").saveCustomer(customer);
```

21.2.2.3. Seam.Component.getComponentName()

把一个对象传递到该方法，如果对象是组件则将返回它的组件名，否则将返回 `null`。

```
if (Seam.Component.getComponentName(instance) == "customer")
    alert("Customer");
else if (Seam.Component.getComponentName(instance) == "staff")
    alert("Staff member");
```

21.2.3. Seam.Remoting

Seam Remoting相关的大多数客户端功能都包含在 `Seam.Remoting` 对象中。你不必直接调用它的大多数方法，但是非常有必要解释一下两个非常重要的方法。

21.2.3.1. Seam.Remoting.createType()

如果你的应用程序包含或者使用了不是Seam组件的JavaBean类，那么你就需要在客户端创建这些类型并把它们作为参数传递到组件的方法中。使用 `createType()` 方法来创建你所需类型的实例，并以完整限定的Java类名传进来作为参数：

```
var widget = Seam.Remoting.createType("com.acme.widgets.MyWidget");
```

21.2.3.2. Seam.Remoting.getTypeName()

`Seam.Remoting.getTypeName()` 与 `Seam.Component.getComponentName()` 等价，但它是针对非组件类型的。它将返回对象实例的类型名，如果不知道类型名，则返回 `null`。这个名称是完整限定的Java类名。

21.3. EL表达式求值

Seam Remoting也对EL表达式求值提供了支持，即提供了另外一种便利的方法来从服务器端获取数据。通过使用 `Seam.Remoting.eval()`，一个EL表达式可以在服务器端被远程求值，并且将其结果返回给客户端的回调方法。此方法接受两个参数，第一个是要被求值的EL表达式，第二个是调用表达式值的回调方法。示例如下：

```
function customersCallback(customers) {
  for (var i = 0; i < customers.length; i++) {
    alert("Got customer: " + customers[i].getName());
  }
}

Seam.Remoting.eval("#{customers}", customersCallback);
```

在这个例子中，表达式 `#{customers}` 将被Seam求值，并且表达式的值（此处是Customer对象列表）被返回给 `customersCallback()` 方法。一定要记住，以这种方式返回的对象必须导入他们的类型（通过 `s:remote`）这样才能与JavaScript一起工作。因此，为了与 `customer` 对象列表一起工作，它需要导入 `customer` 类型。

```
<s:remote include="customer"/>
```

21.4. 客户端接口

在上节的配置中，接口或组件的存根通过 `seam/resource/remoting/interface.js` 被导入到我们页面里：

或者使用`s:remote`标签

```
<script type="text/javascript"
  src="seam/resource/remoting/interface.js?customerAction"></script>
```

```
<s:remote include="customerAction"/>
```

通过在页面中包含这段脚本，组件的接口定义，及执行组件的方法所需要的任何别的组件或者类型都将被生成并且可被Seam Remoting框架所使用。

这会生成两种客户存根：可执行的存根和类型存根。可执行的存根具有一定的行为，能被用来执行会话bean组件的方法。相反，类型存根包含状态，也表示参数或者返回值的类型。

客户存根类型的生成依赖于Seam组件的类型。如果组件是会话Bean，那么将生成可执行的存根，否则如果组件是实体或者JavaBean，那么将生成类型存根。这里也有一个例外；如果你的组件是JavaBean（例如它既不是会话Bean也不是实体Bean）并且任何它的方法被注解为`@WebRemote`，那么将生成一个可执行的存根而不是一个类型存根。这允许你在非EJB环境中使用Remoting来调用JavaBean组件而不需要访问会话Bean。

21.5. 上下文

Seam Remoting上下文包含了附加的信息，并在发送和接收中作为远程请求 / 响应周期的一部分。目前它只包含了对话ID，但是将来它可能被扩展。

21.5.1. 设置和读取对话ID

如果你打算在对话范围内使用远程调用，那么你需要能从Seam Remoting上下文中读取或者设置对话ID。在发起远程请求调用 `Seam.Remoting.getContext().getConversationId()` 后读取对话ID。在发起请求前通过调用 `Seam.Remoting.getContext().setConversationId()` 来设置对话ID。

如果对话ID没有通过 `Seam.Remoting.getContext().setConversationId()` 显式地进行设置，那么它将自动地被赋值为任意远程调用返回的第一个有效的对话ID。如果你的页面有多个对话，那么你需要在每次调用之前显式地设置对话ID。如果你只是工作于单个对话中，那么你不需要额外做任何事情。

21.5.2. 当前对话范围内的远程调用

在某些情况下，可能会要求在当前视图的对话范围内发起一个远程调用，为此，你必须要在调用之前显式地设置对话ID。以下一小段JavaScript代码将在当前视图会话ID远程调用的时候，设置会话ID。

```
Seam.Remoting.getContext().setConversationId( '#{conversation.id}' );
```

21.6. 批量请求

Seam Remoting允许在单个请求中执行多个组件的调用。只要能减少网络流量，那么极力推荐使用这个特性。

`Seam.Remoting.startBatch()` 方法将启动一个新的批处理，启动批处理后任何组件的调用都将进入队列，而不是立刻的发送。当所有的组件调用都被加到批处理以后，`Seam.Remoting.executeBatch()` 方法将发送一个包含所有调用队列的请求到服务器，服务器将顺序地执行这些调用。当这些调用被执行之后，单个响应将返回客户端，它包含了所有的返回值，同时回调函数（如果提供的话）也将按与执行相同的顺序被触发。

如果你通过 `startBatch()` 方法启动了一个新的批处理方法，然后你决定不发送它，那么你需要调用 `Seam.Remoting.cancelBatch()` 方法，它将丢弃任何队队中的调用并退出批处理模式。

使用批处理的例子请见 </examples/remoting/chatroom>。

21.7. 使用数据类型

21.7.1. 原生 / 基本 类型

这部分描述了基本数据类型的支持。一般来说，在服务器端这些值是与它们的原生类型或者相应的包装类相兼容的。

21.7.1.1. String

当设置字符串参数值时可以简单地使用JavaScript字符串对象。

21.7.1.2. Number

支持Java语言支持的所有数字类型。在客户端数字值总是被序列化为字符串，在服务器端他们被转化成正确的目标类型。进行转化时 Byte、Double、Float、Integer、Long 和 Short 的原生类型或者包装类型都被支持。

21.7.1.3. Boolean

Boolean在客户端表示为JavaScript Boolean值，在服务器端表示为Java Boolean。

21.7.2. JavaBeans

一般来说，JavaBean一般是Seam实体或者JavaBean组件，或者其他别的非组件类。需要使用适当的方法（Seam组件使用 `Seam.Component.newInstance()`，其它使用 `Seam.Remoting.createType()`）来创建对象的新实例。

注意只有这两个方法创建的对象才能被用作参数值，这些参数值不是本节所提到的有效类型之一。在不能确定参数类型的情况下，你可以使用如下的组件方法：

```
@Name("myAction")
public class MyAction implements MyActionLocal {
    public void doSomethingWithObject(Object obj) {
        // code
    }
}
```

在这个示例中你可能要传进 `myWidget` 组件的一个实例，然而 `myAction` 接口并没有包含 `myWidget`，因为它并没有被它的任何方法直接引用。为了能够把参数传进来，需要显式地引入 `MyWidget`：

```
<s:remote include="myAction,myWidget"/>
```

这允许使用 `Seam.Component.newInstance("myWidget")` 创建 `myWidget` 对象，并允许创建后的对象传递到 `myAction.doSomethingWithObject()` 中。

21.7.3. Date和时间

日期值被序列化成字符串表示，并且精确到毫秒。在客户端，使用JavaScript日期对象来使用日期值。在服务器端，使用 `java.util.Date` 类（或者派生类，如 `java.sql.Date` 或 `java.sql.Timestamp`）。

21.7.4. Enums 枚举类型

在客户端，Enum也被作为String处理。当为Enum参数设置值时，简单地使用Enum的字符串表示就行了。以下面的组件为例：

```
@Name("paintAction")
public class paintAction implements paintLocal {
    public enum Color {red, green, blue, yellow, orange, purple};
}
```

```
public void paint(Color color) {
    // code
}
```

为了调用 `paint()` 方法，并且传递给参数`color`的值是 `red`，只要把`red`作为字符串传入就可以了：

```
Seam.Component.getInstance("paintAction").paint("red");
```

反过来也是成立的 —— 也就是说，如果一个组件方法返回Enum型参数（或者返回的对象图里包含一个Enum字段），那么在客户端仍将被表示为一个字符串。

21.7.5. Collections 集合

21.7.5.1. Bags

Bags囊括了所有的集合类型，包含arrays、collections、lists、sets，（但不包含Maps —— 见下一章），它在客户端的实现是JavaScript array。当调用一个接收上述类型为参数的组件方法时，你的参数应该是JavaScript array。如果一个组件方法返回上述类型之一，那么返回值将是JavaScript array。发生组件方法调用时，在服务器端Seam Remoting框架能够非常聪明地把Bag类型转化为适当的类型。

21.7.5.2. Maps

JavaScript并没有对Map的本地支持，Seam Remoting框架支持简单的Map实现。通过创建Seam.Remoting.Map对象以支持把Map用做远程调用的参数。

```
var map = new Seam.Remoting.Map();
```

这个Javascript实现提供了处理Map的基本方法：`size()`、`isEmpty()`、`keySet()`、`values()`、`get(key)`、`put(key, value)`、`remove(key)` 和 `contains(key)`。每一个方法等同于Java中对应的方法。在Java中一些方法将返回集合对象，例如 `keySet()` 和 `values()`，相应地，在JavaScript中包含key或者value对象的JavaScript Array对象也将被返回。

21.8. 调试

为了能够跟踪Bug，可以启用调试模式，在调试模式下，所有在客户端和服务端发出和返回的数据包的内容都被显示在一个弹出窗口中。为了启用调试模式，或者在JavaScript脚本中执行`setDebug()` 方法：

```
Seam.Remoting.setDebug(true);
```

或者通过`components.xml`配置它：

```
<remoting:remoting debug="true"/>
```

如果要关闭调试模式，则需要调用 `setDebug(false)`。如果你要在调试日志中记录一些自己定义的

信息，那需要调用 `Seam.Remoting.log(message)`。

21.9. 加载消息

默认加载消息显示在屏幕的右上角，并且是可以修改的，它的表现形式可以自定义甚至可以关掉。

21.9.1. 修改信息

如果要把默认的“Please Wait...”消息改成其它内容，则需要设置 `Seam.Remoting.loadingMessage` 的值：

```
Seam.Remoting.loadingMessage = "Loading...";
```

21.9.2. 隐藏加载信息

如果要尽可能少的显示加载消息，可以通过覆写 `displayLoadingMessage()` 和 `hideLoadingMessage()` 的实现为反之不显示任何消息的函数：

```
// don't display the loading indicator
Seam.Remoting.displayLoadingMessage = function() {};
Seam.Remoting.hideLoadingMessage = function() {};
```

21.9.3. 自定义加载指示器

如果你需要覆写加载指示器以显示一个动画图标或者其他东西，那么你需要覆写 `displayLoadingMessage()` 和 `hideLoadingMessage()`。

```
Seam.Remoting.displayLoadingMessage = function() {
    // Write code here to display the indicator
};

Seam.Remoting.hideLoadingMessage = function() {
    // Write code here to hide the indicator
};
```

21.10. 控制返回数据

当远程方法被执行后，执行结果被序列化成一个XML响应并返回客户端。这个响应被客户端反射成一个JavaScript对象。对于复杂的类型（例如Javabean），它们包含了其他对象的引用，所有这些被引用的对象也将被序列化为这个响应的一部分。这些被引用的对象可能又引用了其他对象，同时还可能存在更深层次的引用关系。如果不检查，这个对象图可能是巨大的，具体取决于对象间的关系。作为一个次要的问题（除了响应可能很冗长之外），你可能也不希望把敏感的信息暴露给客户端。

Seam Remoting提供了一个简单的方式来限制对象图，即指定远程方法的 `@WebRemote` 注解中的 `exclude` 字段。这个字段接受包含用.号指定的一个或多个路径的字符串数组。当调用一个远程方法时，执行结果的对象图中的对象如果与这些路径匹配，就将被从序列化结果包中去掉。

我们使用下面的 `Widget` 类来说明整个示例：

```
@Name("widget")
public class Widget
{
    private String value;
    private String secret;
    private Widget child;
    private Map<String,Widget> widgetMap;
    private List<Widget> widgetList;

    // getters and setters for all fields
}
```

21.10.1. 一般字段的约束

如果远程方法返回 `Widget` 实例，但你不希望暴露 `secret` 字段，因为它包含一些敏感信息，你可以用如下的方式限制它：

```
@WebRemote(exclude = {"secret"})
public Widget getWidget();
```

值“secret”指向了返回对象的 `secret` 字段。现在假定我们不关心这个特殊字段会暴露给客户端。相反，我们看到返回值 `Widget` 含有一个 `child` 字段，它也是 `Widget`。我们该如何来隐藏 `child` 的值呢？我们可以在结果对象图中使用 `.` 号指定的字段路径来达到这个目的：

```
@WebRemote(exclude = {"child.secret"})
public Widget getWidget();
```

21.10.2. 集合和映射的约束

对象存在于一个对象图中的另一个方式是 `Map` 或者一些集合类（`List`、`Set`、`Array` 等等）。集合类很简单，可以和其它别的字段一样来看待。例如如果 `Widget` 在它的 `widgetList` 字段里包含了一个 `Widget` 列表，为了限定这个列表中的 `Widget` 的 `secret` 字段的注解大概类似于这样：

```
@WebRemote(exclude = {"widgetList.secret"})
public Widget getWidget();
```

如果要限定 `Map` 的键或者值，那么注解会有点不同。在 `Map` 的字段名后面增加 `[key]` 将限定 `Map` 键对象，同时，`[value]` 将限定值对象。下面的例子描述了 `widgetMap` 字段怎么样限定了它们的 `secret` 字段：

```
@WebRemote(exclude = {"widgetMap[value].secret"})
public Widget getWidget();
```

21.10.3. 特定类型对象的约束

还有一个注解可用来限定对象的字段，不管字段出现在结果对象图的哪个位置它都起作用。这个注解既可以使用组件的名称（如果对象是一个 `Seam` 组件），也可以是一个完全限定的类名（仅当对象不是 `Seam` 组件时），并且是以 `[]` 的形式来限定的。

```
@WebRemote(exclude = {"[widget].secret"})
public Widget getWidget();
```

21.10.4. 组合约束

限定也可以合并，用多个路径对对象图中的对象进行过滤：

```
@WebRemote(exclude = {"widgetList.secret", "widgetMap[value].secret"})
public Widget getWidget();
```

21.11. JMS消息

Seam Remoting利用实践经验对JMS消息提供了支持。本节描述了当前对JMS已有的支持，但是请记住在将来这可能发生变化。因此现在并不推荐在产品环境中使用这个特性。

21.11.1. 配置

为能够订阅JMS主题，你必须先配置一个可以通过Seam Remoting订阅的主题列表。需要在 `seam.properties`、`web.xml` 或者 `components.xml` 的 `org.jboss.seam.remoting.messaging.subscriptionRegistry.allowedTopics` 中列出所有的主题。

```
<remoting:remoting poll-timeout="5" poll-interval="1"/>
```

21.11.2. 订阅JMS主题

下面的例子说明了如何来订阅一个JMS主题：

```
function subscriptionCallback(message)
{
    if (message instanceof Seam.Remoting.TextMessage)
        alert("Received message: " + message.getText());
}

Seam.Remoting.subscribe("topicName", subscriptionCallback);
```

`Seam.Remoting.subscribe()` 方法具有两个参数，第一个参数是可被订阅的JMS主题名，第二个参数是接收到一个消息时要调用的回调函数。

支持两种类型的消息：`Text`消息和`Object`消息。若要测试传给回调函数的消息类型，你可以调用 `instanceof` 操作符来测试这个消息是 `Seam.Remoting.TextMessage` 还是 `Seam.Remoting.ObjectMessage`。`TextMessage` 的 `text` 字段是文本值，`ObjectMessage` 的 `object` 字段（或者调用 `getObject()` 方法得到的）是对象值。

21.11.3. 退订主题

如果要取消一个主题的订阅，则需要调用 `Seam.Remoting.unsubscribe()`，并且传进这个主题的名称：

```
Seam.Remoting.unsubscribe("topicName");
```

21.11.4. 调整轮询过程

你可以通过修改两个参数来控制轮询的发生方式。第一个参数是`Seam.Remoting.pollInterval`，它控制新消息的并发轮询间隔。这个参数的单位是秒，默认值是10。

第二个参数是 `Seam.Remoting.pollTimeout`，它的单位也是秒。它控制在超时和发送一个空的响应之前，发送到服务器端的请求等待新消息要等多久。它的默认值是0秒，也就是说当服务器端被轮询到后，如果没有已经准备好的消息发送，则立即发送一个空的响应。

若要设置一个高的 `pollTimeout` 值，你应该非常谨慎；每一个请求都必须等待一个响应消息，也就是说服务器端线程在收到消息前，或者请求超时前是被阻塞的。由于这个原因，如果很多请求同时被处理，会导致大量的线程被阻塞。

建议通过`components.xml`来设置这些选项，但是如果需要，也可以通过JavaScript来覆盖它们。下面的例子说明了如何配置轮询以使得它更具侵略性。你应该在应用程序中为这些参数设置适当的值。

通过`components.xml`配置：

```
<remoting:remoting poll-timeout="5" poll-interval="1"/>
```

通过JavaScript配置：

```
// Only wait 1 second between receiving a poll response and sending the next poll request.  
Seam.Remoting.pollInterval = 1;  
  
// Wait up to 5 seconds on the server for new messages  
Seam.Remoting.pollTimeout = 5;
```


第 22 章 Seam和Google的Web工具包 (GWT)

对于那些宁愿使用Google Web Toolkit（以下简称：GWT）去开发动态的AJAX的应用程序的人们来说，Seam提供了一个允许GWT widget直接与Seam组件交互的整合层。

为了使用GWT，我们假设你已经很熟悉GWT工具 - 更多信息可以在<http://code.google.com/webtoolkit/> 中找到。 本章就不去解释GWT如何工作以及如何使用了。

22.1. 配置

在Seam应用程序中使用GWT并不需要特别的配置，但是Seam资源的servlet是必须安装的。更多详情请看 第 25 章 Seam配置和Seam应用程序打包 。

22.2. 准备你的组件

准备一个要通过GWT进行调用的Seam组件的第一步，是给你想要调用的方法创建同步和异步的服务接口。这两个接口都应该扩展GWT接口 `com.google.gwt.user.client.rpc.RemoteService`：

```
public interface MyService extends RemoteService
{
    public String askIt(String question);
}
```

异步接口应该完全相同，除此之外，它还要给它声明的每一个方法包含一个额外的 `AsyncCallback` 参数：

```
public interface MyServiceAsync extends RemoteService
{
    public void askIt(String question, AsyncCallback callback);
}
```

在这个范例 `MyServiceAsync` 中，异步接口将通过GWT实现，并且永远不应该直接实现。

下一步，是创建一个实现同步接口的Seam组件：

```
@Name("org.jboss.seam.example.remoting.gwt.client.MyService")
public class ServiceImpl implements MyService
{
    @WebRemote
    public String askIt(String question)
    {
        if (!validate(question))
        {
            throw new IllegalStateException("Hey, this shouldn't happen, I checked on the client, " +
                "but its always good to double check.");
        }
        return "42. Its the real question that you seek now.";
    }

    public boolean validate(String q)
    {
        ValidationUtility util = new ValidationUtility();
        return util.isValid(q);
    }
}
```

```
}
```

应该做成能通过GWT访问的那些方法，需要通过标识 `@WebRemote` 进行注解，这是所有能够进行Web远程访问的方法都要求的。

22.3. 将GWT小组件接到Seam组件

下一步，是编写一个将异步接口返回给组件的方法。这个方法可以放在小组件类里面，将被小组件用来获得一个对异步客户端存根（stub）的引用：

```
private MyServiceAsync getService()
{
    String endpointURL = GWT.getModuleBaseURL() + "seam/resource/gwt";

    MyServiceAsync svc = (MyServiceAsync) GWT.create(MyService.class);
    ((ServiceDefTarget) svc).setServiceEntryPoint(endpointURL);
    return svc;
}
```

最后一步就是编写小组件代码，它在客户端存根上调用方法。以下代码创建一个简单的用户接口，包含label（标签）、text input（文本输入框）和一个button（按钮）：

```
public class AskQuestionWidget extends Composite
{
    private AbsolutePanel panel = new AbsolutePanel();

    public AskQuestionWidget()
    {
        Label lbl = new Label("OK, what do you want to know?");
        panel.add(lbl);
        final TextBox box = new TextBox();
        box.setText("What is the meaning of life?");
        panel.add(box);
        Button ok = new Button("Ask");
        ok.addClickListener(new ClickListener()
        {
            public void onClick(Widget w)
            {
                ValidationUtility valid = new ValidationUtility();
                if (!valid.isValid(box.getText()))
                {
                    Window.alert("A question has to end with a '?'");
                }
                else
                {
                    askServer(box.getText());
                }
            }
        });
        panel.add(ok);

        initWidget(panel);
    }

    private void askServer(String text)
    {
        getService().askIt(text, new AsyncCallback()
        {
            public void onFailure(Throwable t)
            {
            }
        });
    }
}
```

```

    {
        Window.alert(t.getMessage());
    }

    public void onSuccess(Object data)
    {
        Window.alert((String) data);
    }
    });
}

...

```

当点击按钮时，它就调用 `askServer()` 方法来传递输入框的内容（在这个例子中，还执行了验证，以确保输入的是有效的问题）。这个 `askServer()` 方法获得一个对异步客户端存根的引用（由 `getService()` 方法返回），并调用 `askIt()` 方法。这个结果（或者调用失败时的错误信息）显示在以一个警告窗口中。

HelloWorld

This is an example of a host page for the HelloWorld application. You can attach a Web Toolkit module to any HTML page you like, making it easy to add bits of AJAX functionality to existing pages without starting from scratch.

OK, what do you want to know?

这个例子的完整代码可以在Seam发行包的 `examples/remoting/gwt` 中找到。

22.4. GWT Ant Targets

对于GWT应用程序的发布（部署）来说，有一个编译成JavaScript的步骤（它压缩和混淆了代码）。有一个Ant实用程序可以用来取代GWT提供的命令行或者GUI实用程序。为了使用这个功能，你不仅在Ant classpath中要有Ant的任务jar包，还要下载GWT（无论如何，你在本机模式下时也会需要它）。

接下来在你的Ant文件中（在你的Ant文件顶头附近）

```

<taskdef uri="antlib:de.samaflost.gwttasks"
    resource="de/samaflost/gwttasks/antlib.xml"
    classpath="./lib/gwttasks.jar"/>

<property file="build.properties"/>

```

创建一个 `build.properties` 文件，它包括以下内容：

```
gwt.home=/gwt_home_dir
```

这当然应该指向GWT的安装路径。然后用它创建一个Target：

```

<!-- the following are are handy utilities for doing GWT development.
    To use GWT, you will of course need to download GWT seperately -->
<target name="gwt-compile">
    <!-- in this case, we are "re homing" the gwt generated stuff, so in this case
        we can only have one GWT module - we are doing this deliberately to keep the URL short -->

```

```
<delete>
  <fileset dir="view"/>
</delete>
<gwt:compile outDir="build/gwt"
  gwtHome="${gwt.home}"
  classBase="${gwt.module.name}"
  sourceclasspath="src"/>
<copy todir="view">
  <fileset dir="build/gwt/${gwt.module.name}"/>
</copy>
</target>
```

This target when called will compile the GWT application, and copy it to the specified directory (which would be in the webapp part of your war - remember GWT generates HTML and Javascript artifacts). You never edit the resulting code that gwt-compile generates - you always edit in the GWT source directory. 当这个Target被调用时，将编译GWT应用程序，并将它复制到指定的目录中（它会在你war的 webapp 部分中 —— 记住GWT生成HTML和Javascript工件）。你永远不用编辑 gwt-compile 生成的结果代码 —— 你始终在GWT源路径中进行编辑。

记住GWT有一个本机模式浏览器 —— 你在用GWT开发时使用的应该就是它。如果你没有使用该浏览器，而只是每次对它进行编译，那你就没有充分利用这个工具包（事实上，如果你无法或者没有使用本机模式浏览器，我只能说你根本不应该使用GWT —— 它非常有用！）

第 23 章 Spring Framework集成

Spring集成模块可以把基于Spring的项目轻松移植到Seam，并且允许Spring应用Seam的一些关键特性，例如对话（Conversation）和Seam的高级持久化上下文管理。

请注意！Spring集成代码包含在jboss-seam-ioc库中。在本章涉及的所有seam-spring集成技术中都需要引用这个依赖。

Seam对Spring提供了如下一些支持：

- 把Seam的组件实例注入到Spring Bean中
- 把Spring Bean注入到Seam组件中
- 把Spring Bean转换成Seam组件
- 允许Spring Bean存在于任何Seam的上下文中
- 使用Seam组件来启动一个Spring WebApplicationContext
- 支持Spring PlatformTransactionManagement
- 为Spring的 `OpenEntityManagerInViewFilter` 和 `OpenSessionInViewFilter` 提供一个Seam管理的替代品
- 支持由 `@Asynchronous` 调用的Spring `TaskExecutors`

23.1. 把Seam组件注入Spring Bean中

要把Seam组件注入到Spring Bean中，需要使用到 `<seam:instance/>` 命名空间处理器。要启用该处理器，Spring Bean的定义文件中必须添加Seam命名空间：

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:seam="http://jboss.com/products/seam/spring-seam"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://jboss.com/products/seam/spring-seam
    http://jboss.com/products/seam/spring-seam-2.0.xsd">
```

现在，每一个Seam组件都可以被注入到任意Spring Bean中了：

```
<bean id="someSpringBean" class="SomeSpringBeanClass" scope="prototype">
  <property name="someProperty">
    <seam:instance name="someComponent"/>
  </property>
</bean>
```

可以用EL表达式来代替组件名：

```
<bean id="someSpringBean" class="SomeSpringBeanClass" scope="prototype">
  <property name="someProperty">
    <seam:instance name="#{someExpression}"/>
  </property>
</bean>
```

```
</bean>
```

Seam组件实例甚至还可以通过Spring Bean id来注入到Spring Bean中。

```
<seam:instance name="someComponent" id="someSeamComponentInstance"/>

<bean id="someSpringBean" class="SomeSpringBeanClass" scope="prototype">
  <property name="someProperty" ref="someSeamComponentInstance">
</bean>
```

警告！

Seam使用多个上下文来完全支持有状态组件。Spring不是这样。和Seam的双向注入（bijection）不同，Spring的注入并不是在方法调用时，而是发生在Spring Bean初始化时。因此，Bean初始化时被用到的那个实例，会在Bean的整个生命周期中一直被使用。例如，一个Seam的 CONVERSATION 域组件被直接注入到一个单例Spring Bean中，这个单例的Bean将长期持有这个实例的引用，直到对话结束！我们把这种问题称为 域阻抗（scope impedance）。Seam的双向注入可以很自然地管理域阻抗，就好象系统中的调用一样。在Spring中，我们需要注入一个Seam组件的代理，并且在代理被调用的时候解析该引用。

`<seam:instance/>` 标签可以自动代理Seam组件。

```
<seam:instance id="seamManagedEM" name="someManagedEMComponent" proxy="true"/>

<bean id="someSpringBean" class="SomeSpringBeanClass">
  <property name="entityManager" ref="seamManagedEM">
</bean>
```

这个例子演示了一种在Spring Bean中使用Seam管理的持久化上下文的方法。（想要了解如何更健壮地使用Seam管理的持久化上下文来替换Spring的 `OpenEntityManagerInView` 过滤器，请见 在Spring中使用Seam管理的持久化上下文）

23.2. 将Spring Bean注入到Seam组件中

将Spring Bean注入到Seam组件实例中更容易，有二种方法：

- 使用EL表达式注入Spring Bean
- 把Spring Bean转化为Seam组件

我们将在下一小节中讨论第二种方法。访问Spring Bean最简单的方法是通过EL表达式。

Spring的`DelegatingVariableResolver`是Spring用于整合JSF的一个集成点。`VariableResolver` 允许所有的Spring Bean通过Bean id在EL中被使用。你需要在 `faces-config.xml` 中添加 `DelegatingVariableResolver`：

```
<application>
  <variable-resolver>
    org.springframework.web.jsf.DelegatingVariableResolver
  </variable-resolver>
</application>
```

接下来你可以使用 `@In` 来注入 Spring Bean:

```
@In("#{bookingService}")
private BookingService bookingService;
```

Spring Bean在EL中的应用不单单只有注入。Seam的任何EL表达式中都可以使用Spring Bean: 过程和页面流的定义, 工作内存断言 (working memory assertions) 等等...

23.3. 将Spring Bean转换为Seam组件

`<seam:component/>` 命名空间处理器用于将Spring Bean转换成一个Seam组件。只需要在你希望转换为Seam组件的Bean的声明中加上 `<seam:component/>` 标签即可:

```
<bean id="someSpringBean" class="SomeSpringBeanClass" scope="prototype">
  <seam:component/>
</bean>
```

默认情况下, `<seam:component/>` 将使用Bean定义中提供的类和名称来创建一个 无状态 (STATELESS) 的Seam组件。有时候, 在使用 `FactoryBean` 时, Spring Bean的类可能不是Bean定义中的那个类。在这种情况下, `class` 应该是被明确指定的。在可能存在命名冲突时需要明确给出Seam的组件名。

如果你希望Spring Bean在一个特定的Seam域中受管理, 就使用 `<seam:component/>` 的 `scope` 属性。如果指定了任何非 无状态 的Seam域, Spring Bean就必须限定为 `prototype` 的。先前存在的Spring Bean通常都有基础的无状态的特征, 所以通常并不需要这个属性。

23.4. Seam作用域的Spring Bean

Seam集成包中同样允许你像Spring 2.0风格的自定义作用域那样来使用Seam的上下文。你可以在任意Seam上下文中定义Spring Bean。但是, 需要重申的是, Spring的组件模型并非设计为支持状态 (Statefulness) 的, 所以请小心使用这一特性。特别是Session和Conversation作用域的Spring Bean集群很有问题, 从大作用域注入到小作用域时也要格外小心。

一旦在Spring的Bean Factory配置中指定了 `<seam:configure-scopes/>`, 所有的Seam作用域都将以自定义作用域的形式暴露给Spring Bean。要将一个Spring Bean与某个特定的Seam作用域联系起来时, 请在bean定义的 `scope` 属性中指定Seam作用域。

```
<!-- Only needs to be specified once per bean factory-->
<seam:configure-scopes/>

...

<bean id="someSpringBean" class="SomeSpringBeanClass" scope="seam.CONVERSATION"/>
```

作用域名的前缀可以通过 `configure-scopes` 定义的 `prefix` 属性来修改。(默认的前缀是seam。)

以这种方式定义的Seam作用域Spring Bean可被注入到其它Spring Bean而无需使用 `<seam:instance/>`。但是, 仍要小心确认域阻抗是否得到维护。通常, 在Spring中的一般做法是在Bean定义中指定 `<aop:scoped-proxy/>`。但是, Seam作用域的Spring Bean并不兼容于 `<aop:scoped-proxy/>`。所以如果你需要向某个单例中注入Seam作用域的Spring Bean, 必须使用 `<seam:instance/>`:

```
<bean id="someSpringBean" class="SomeSpringBeanClass" scope="seam.CONVERSATION"/>

...

<bean id="someSingleton">
    <property name="someSeamScopedSpringBean">
        <seam:instance name="someSpringBean" proxy="true"/>
    </property>
</bean>
```

23.5. 使用Spring PlatformTransactionManagement

Spring提供了支持多种事务API（JPA、Hibernate、JDO和JTA）的可扩展事务管理抽象，还提供了与诸如WebSphere和WebLogic之类的应用服务器TransactionManagers的紧密集成。Spring事务管理支持很多高级特性，例如内嵌事务和完整的Java EE事务传播规则（REQUIRES_NEW、NOT_SUPPORTED等等）。想要获得更多信息，请见[Spring文档](#)。

如下配置Seam将启用SpringTransaction组件来使用Spring事务：

```
<spring:spring-transaction platform-transaction-manager="{transactionManager}"/>
```

spring:spring-transaction组件将利用Spring事务同步能力来同步回调。

23.6. 在Spring中使用Seam管理的持久化上下文

Seam的最强大的功能之一是它的对话作用域（conversation scope）和为对话周期提供一个EntityManager的能力。这消除了很多与实体的分离和重组相关的问题，减少了LazyInitializationException 的发生。Spring没有管理超出单个Web请求作用域的持久化上下文的方法（OpenEntityManagerInViewFilter）。所以，如果Spring开发者能够用与Spring集成JPA所用的相同工具来访问一个Seam管理的持久化上下文的话就再好不过了。（例如PersistenceAnnotationBeanPostProcessor、JpaTemplate等等。）

Seam可以让Spring利用它的JPA工具访问Seam管理的持久化上下文，这让Spring应用拥有了对话作用域的持久化上下文的能力。

该集成提供以下功能：

- 使用Spring提供的工具透明地访问一个Seam管理持久化上下文
- 在非Web请求中访问Seam会话作用域的持久化上下文（例如异步Quartz任务中）
- 考虑使用Seam管理的持久化上下文和Spring管理的事务（将需要手动清除缓冲的持久化上下文）

Spring的持久化上下文传播模型允许每个EntityManagerFactory仅有一个打开的EntityManager，所以Seam集成就封装一个EntityManagerFactory，其中放入Seam管理的持久化上下文。

```
<bean id="seamEntityManagerFactory" class="org.jboss.seam.ioc.spring.SeamManagedEntityManagerFactoryBean">
    <property name="persistenceContextName" value="entityManager"/>
</bean>
```


'persistenceContextName' 是Seam管理的持久化上下文组件的名字。默认情况下，该 EntityManagerFactory 有一个和Seam组件名一样的unitName，或者像例子中那样名为'entityManager'。如果你希望提供一个不同的unitName，你能够通过提供一个persistenceUnitName来实现，如下所示：

```
<bean id="seamEntityManagerFactory" class="org.jboss.seam.ioc.spring.SeamManagedEntityManagerFactoryBean">
    <property name="persistenceContextName" value="entityManager"/>
    <property name="persistenceUnitName" value="bookingDatabase:extended"/>
</bean>
```

这个EntityManagerFactory能在任何Spring提供的工具中被使用。例如，可以像以前那样使用Spring的 PersistenceAnnotationBeanPostProcessor。

```
<bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"/>
```

如果你在Spring中定义你真正的EntityManagerFactory但希望使用一个Seam管理的持久化上下文，你能够告诉 PersistenceAnnotationBeanPostProcessor 你默认希望使用哪个persistenceUnitName，可以通过指定 defaultPersistenceUnitName 来实现。

applicationContext.xml 文件可能像下面这样：

```
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
    <property name="persistenceUnitName" value="bookingDatabase"/>
</bean>
<bean id="seamEntityManagerFactory" class="org.jboss.seam.ioc.spring.SeamManagedEntityManagerFactoryBean">
    <property name="persistenceContextName" value="entityManager"/>
    <property name="persistenceUnitName" value="bookingDatabase:extended"/>
</bean>
<bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor">
    <property name="defaultPersistenceUnitName" value="bookingDatabase:extended"/>
</bean>
```

component.xml 文件可能像下面这样：

```
<persistence:managed-persistence-context name="entityManager"
    auto-create="true" entity-manager-factory="#{entityManagerFactory}"/>
```

JpaTemplate 和 JpaDaoSupport 的配置方法不变。

```
<bean id="bookingService" class="org.jboss.seam.example.spring.BookingService">
    <property name="entityManagerFactory" ref="seamEntityManagerFactory"/>
</bean>
```

23.7. 在Spring中使用Seam管理的Hibernate会话

Seam的Spring集成支持使用Spring的工具来完整访问Seam管理的Hibernate会话（Hibernate Session）。这和 JPA集成 很像。

与Spring的JPA集成一样，在Spring的工具中，Spring的传播模型只允许每个 EntityManagerFactory 在一个事务里拥有一个打开的EntityManager。所以Seam Session集成封装了一个代理SessionFactory，其中包含一个Seam管理的Hibernate会话上下文。

```
<bean id="seamSessionFactory" class="org.jboss.seam.ioc.spring.SeamManagedSessionFactoryBean">
    <property name="sessionName" value="hibernateSession"/>
</bean>
```

'sessionName' 是 persistence:managed-hibernate-session 组件的名字。该SessionFactory可被用于任意Spring提供的工具中。该集成支持对 SessionFactory.getCurrentInstance() 的调用，只要调用 SeamManagedSessionFactory 的 getCurrentInstance() 方法。

23.8. 作为Seam组件的Spring应用上下文

尽管你可以使用Spring的ContextLoaderListener 来启动应用程序的Spring ApplicationContext，但这种做法存在一些局限。

- Spring的ApplicationContext必须开始于 SeamListener 之后
- 要为Seam的单元和集成测试启动一个Spring ApplicationContext有些麻烦

为突破这二个局限，Spring集成包括一个启动Spring ApplicationContext的Seam组件。在 components.xml 中添加 <spring:context-loader/> 定义就能使用该组件。在 config-locations 属性中指定Spring上下文文件位置。如果需要配置多个配置文件，你可以按照标准 components.xml 的多值配置实践，把它们置于内嵌的 <spring:config-locations/> 元素中。

```
<components xmlns="http://jboss.com/products/seam/components"
    xmlns:spring="http://jboss.com/products/seam/spring"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://jboss.com/products/seam/components
        http://jboss.com/products/seam/components-2.0.xsd
        http://jboss.com/products/seam/spring
        http://jboss.com/products/seam/spring-2.0.xsd">

    <spring:context-loader context-locations="/WEB-INF/applicationContext.xml"/>

</components>
```

23.9. 使用Spring TaskExecutor的@Asynchronous

Spring提供了名为 TaskExecutor 的异步代码执行抽象。在调用有 @Asynchronous 的方法时，Spring Seam集成可以使用 TaskExecutor。要启用该功能，需配置 SpringTaskExecutorDispatcher 并提供一个定义了taskExecutor的Spring Bean:

```
<spring:task-executor-dispatcher task-executor="#{springThreadPoolTaskExecutor}"/>
```

因为Spring的 TaskExecutor 并不支持异步事件调度，所以可以提供一個回调的Seam Dispatcher 来处理异步事件调度:

```
<!-- Install a ThreadPoolDispatcher to handle scheduled asynchronous event -->
<core:thread-pool-dispatcher name="threadPoolDispatcher"/>

<!-- Install the SpringDispatcher as default -->
<spring:task-executor-dispatcher task-executor="#{springThreadPoolTaskExecutor}" schedule-dispatcher="#{threadPoolDispatcher}"/>
```

第 24 章 Hibernate Search

24.1. 简介

如Apache Lucene™ 之类的全文搜索引擎是一种非常强大的技术，给我们的应用程序带来了高效的文本查询。 在处理一个对象域模型的时候有可能会遇到多处不匹配（保持最新的索引，索引结构和域模型之间的不匹配，查询不匹配...） Hibernate Search根据一些注解来索引域模型、管理数据库与索引的同步，并从自由文本查询中带给你正常的受管对象。 Hiberante Search在目前的版本里使用了Apache Luncene。

Hibernate Search在设计之初就是要很好并且尽可能自然地集成JPA和Hibernate，因此自然而然的，JBoss Seam也就提供了Hibernate Search的集成。

请参考Hibernate Search项目文档信息 [Hibernate Search documentation](http://hibernate.org/search/documentation/)。

24.2. 配置

Hibernate Search既可以在 META-INF/persistence.xml 中也可以在 hibernate.cfg.xml 中进行配置。

对大多数配置参数来说，Hibernate Search配置为其提供合理的默认值，在这里有一个最小化配置的描述。

```
<persistence-unit name="sample">
  <jta-data-source>java:/DefaultDS</jta-data-source>
  <properties>
    [...]
    <!-- use a file system based index -->
    <property name="hibernate.search.default.directory_provider"
      value="org.hibernate.search.store.FSDirectoryProvider"/>
    <!-- directory where the indexes will be stored -->
    <property name="hibernate.search.default.indexBase"
      value="/Users/prod/apps/dvdstore/dvdindexes"/>
  </properties>
</persistence-unit>
```

如果计划使用Hibernate Annotations或者EntityManager 3.2.x（已经嵌入到JBoss AS 4.2.GA中），那也需要配置相应的事件监听器。

```
<persistence-unit name="sample">
  <jta-data-source>java:/DefaultDS</jta-data-source>
  <properties>
    [...]
    <!-- use a file system based index -->
    <property name="hibernate.search.default.directory_provider"
      value="org.hibernate.search.store.FSDirectoryProvider"/>
    <!-- directory where the indexes will be stored -->
    <property name="hibernate.search.default.indexBase"
      value="/Users/prod/apps/dvdstore/dvdindexes"/>

    <property name="hibernate.ejb.event.post-insert"
      value="org.hibernate.search.event.FullTextIndexEventListener"/>
    <property name="hibernate.ejb.event.post-update"
      value="org.hibernate.search.event.FullTextIndexEventListener"/>
    <property name="hibernate.ejb.event.post-delete"
```

```

        value="org.hibernate.search.event.FullTextIndexEventListener"/>

    </properties>
</persistence-unit>
    
```



注意

如果是使用了Hibernate Annotation或者EntityManager 3.3.x，这一步就不再需要了。

除了配置文件之外，把下面的jars包需要部署到服务器中或者打包进入你的工程中：

- hibernate-search.jar
- hibernate-commons-annotations.jar
- lucene-core.jar



注意

如果部署的是一个EAR包，别忘了要更新 application.xml 文件

24.3. 用法

Hibernate Search使用注解来映射实体类给Lucene索引，访问 [reference documentation](#) 来获取更多的说明。

Hibernate Search完全集成JPA/Hibernate的API和语法。只需要几行代码，就可以在基于HQL或者Criteria的查询间进行切换。主API和应用程序是通过 FullTextSession API （Hibernate Session 的子类）相互交互的。

当有Hibernate Search的时候，JBoss Seam注入 FullTextSession 。

```

@Stateful
@Name("search")
public class FullTextSearchAction implements FullTextSearch, Serializable
{
    @In
    FullTextSession session;

    public void search(String searchString) {
        org.apache.lucene.query.Query luceneQuery = getLuceneQuery();
        org.hibernate.Query query session.createFullTextQuery(luceneQuery, Product.class);
        searchResults = query
            .setMaxResults(pageSize + 1)
            .setFirstResult(pageSize * currentPage)
            .list();
    }
    [...]
}
    
```



注意

FullTextSession 继承自 org.hibernate.Session，因此它可以被当作正常的Hibernate Session来使用。

如果使用Java Persistence API，建议做一个平滑的集成。

```
@Name("search")
public class FullTextSearchAction implements FullTextSearch, Serializable
{
    @In
    FullTextEntityManager em;

    public void search(String searchString) {
        org.apache.lucene.query.Query luceneQuery = getLuceneQuery();
        javax.persistence.Query query = em.createFullTextQuery(luceneQuery, Product.class);
        searchResults = query
            .setMaxResults(pageSize + 1)
            .setFirstResult(pageSize * currentPage)
            .getResultList();
    }
    [...]
}
```

当有Hibernate Search的时候，FulltextEntityManager 被注入。FullTextEntityManager 继承了 EntityManager 并带有特定的搜索方法，同样 FullTextSession 继承了 Session。

当注入 EJB 3.0 Session 或者 Message Driven Bean 时（例如使用 @PersistenceContext 注解的Bean），就不可能在定义声明中使用 FullTextEntityManager 接口来替换 EntityManager 接口。然而，将注入一个 FullTextEntityManager 实现：进行类型转换后就可以了。

```
@Stateful
@Name("search")
public class FullTextSearchAction implements FullTextSearch, Serializable
{
    @PersistenceContext
    EntityManager em;

    public void search(String searchString) {
        org.apache.lucene.query.Query luceneQuery = getLuceneQuery();
        FullTextEntityManager ftEm = (FullTextEntityManager) em;
        javax.persistence.Query query = ftEm.createFullTextQuery(luceneQuery, Product.class);
        searchResults = query
            .setMaxResults(pageSize + 1)
            .setFirstResult(pageSize * currentPage)
            .getResultList();
    }
    [...]
}
```



小心

对于那些经习惯了在Seam之外使用Hibernate Search的人来说，要注意使用 Search.createFullTextSession 是不需要的了。

对于Hibernate Search更详细的示例用法，请查看JBoss Seam发行包中的DVDStore或者Blog示例。

第 25 章 Seam配置和Seam应用程序打包

配置是一个非常枯燥的话题，也是一种极其乏味的消遣。不幸的是，为了将Seam集成到你的JSF实现和Servlet容器中，有些XML语句是必须的。你不必再被以下部分拖延时间了；你将永远不必亲自键入以下任何内容，因为你可以从示例应用程序中拷贝和粘帖。

25.1. Seam基本配置

首先，让我们看看每当将Seam和JSF一起使用时所需要的基本配置。

25.1.1. 将Seam与JSF和servlet容器集成

当然，你需要一个Faces Servlet！

```
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>*.seam</url-pattern>
</servlet-mapping>
```

（你可以调整URL模式以适合你的需要。）

另外，Seam需要在你的 web.xml 文件中有以下项：

```
<listener>
  <listener-class>org.jboss.seam.servlet.SeamListener</listener-class>
</listener>
```

这个listener负责引导Seam，以及销毁会话和应用程序。

有些JSF实现在服务器端状态保持方面实现得很差，它们会与Seam的对话传播发生冲突。如果你在表单提交时遇到了对话传播的问题，就尝试着切换到客户端状态保持。在你的 web.xml 中需要有这些：

```
<context-param>
  <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
  <param-value>client</param-value>
</context-param>
```

25.1.2. 使用Facelets

如果你愿意听从我们的建议，使用Facelets代替JSP，就需要在 faces-config.xml 中增加以下几行：

```
<application>
  <view-handler>com.sun.facelets.FaceletViewHandler</view-handler>
```

```
</application>
```

并将以下几行加到 web.xml 中：

```
<context-param>
  <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
  <param-value>.xhtml</param-value>
</context-param>
```

25.1.3. Seam Resource Servlet

Seam Resource Servlet提供资源，这些资源被Seam Remoting、captchas（请参考“安全”章节）和一些JSF UI控件所使用。配置Seam Resource Servlet时，需要在 web.xml 中有以下项：

```
<servlet>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <servlet-class>org.jboss.seam.servlet.SeamResourceServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>Seam Resource Servlet</servlet-name>
  <url-pattern>/seam/resource/*</url-pattern>
</servlet-mapping>
```

25.1.4. Seam Servlet过滤器

Seam的基本操作不需要任何Servlet过滤器。可是，有些功能依赖于过滤器的使用。为了让一切对你来说更容易，Seam让你可以增加和配置Servlet过滤器，就像你配置其他内建Seam组件一样。为了利用这个特性，我们必须首先在 web.xml 中装载一个主过滤器：

```
<filter>
  <filter-name>Seam Filter</filter-name>
  <filter-class>org.jboss.seam.servlet.SeamFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>Seam Filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Seam主过滤器 必须 是在 web.xml 中指定的第一个过滤器。这样就能保证它首先运行。

为了取消内建的过滤器，你可以在一个特定的过滤器中设置 disabled="true"

增加主过滤器使得以下内建过滤器可用。

25.1.4.1. 异常处理

这个过滤器在 pages.xml 中提供异常映射功能（几乎所有的应用程序都需要这个）。当未捕获的异常发生时，它也负责回滚未提交的事务。（根据Java EE规范，Web容器应该能自动做到这一点，但我们发现在所有应用程序服务器中，都不支持这一行为。当然在普通Servlet引擎如Tomcat中，它并不是必需的。）

默认情况下，异常处理过滤器将处理所有请求，但是这种行为可以通过增加一个 `<web:exception-filter>` 项到 `components.xml` 文件来进行调整，如下面的例子所示：

```
<components xmlns="http://jboss.com/products/seam/components"
             xmlns:web="http://jboss.com/products/seam/web">

    <web:exception-filter url-pattern="*.seam"/>

</components>
```

- `url-pattern` — 被用来指定过滤哪些请求，默认是所有请求。

25.1.4.2. 通过重定向传播对话

这个过滤器允许Seam通过浏览器重定向来传播对话上下文。它拦截任何浏览器重定向，并增加一个指定Seam对话标识符的请求参数。

默认情况下，重定向过滤器将处理所有的请求，但这个行为也可以在 `components.xml` 文件中进行调整：

```
<web:redirect-filter url-pattern="*.seam"/>
```

- `url-pattern` — 被用来指定过滤哪些请求，默认是所有请求。

25.1.4.3. 多重表单提交

当使用Seam文件上传JSF控件时，这个特性很有必要。它依据多重表单/数据规范（RFC-2388）检测并处理多重表单请求。要覆盖默认设置，向 `components.xml` 中增加以下项：

```
<web:multipart-filter create-temp-files="true"
                     max-request-size="1000000"
                     url-pattern="*.seam"/>
```

- `create-temp-files` — 如果设置为 `true`，加载的文件被写到临时文件中（而不是保留在内存中）。如果期望加载大文件，这个设置可能是一个很重要的考虑因素。默认的设置是 `false`。
- `max-request-size` — 如果加载请求中的文件大小（由读取请求中 `Content-Length` 头的值决定）超过这个值，该请求将被中止。默认设置是0（大小不限）。
- `url-pattern` — 被用来指定过滤哪些请求，默认是所有请求。

25.1.4.4. 字符编码

设置被提交的表单数据的字符编码

默认没有装载这个过滤器，它需要由 `components.xml` 中的一个项来启用：

```
<web:character-encoding-filter encoding="UTF-16"
                              override-client="true"
                              url-pattern="*.seam"/>
```


- `encoding` — 要使用的编码
- `override-client` — 如果它被设置为 `true`，则请求中的编码将被设置为 `encoding` 指定的任何值，而不管请求中是否已经指定一种编码。如果设置为 `false`，当请求中没有指定任何一种编码时，才会设置请求编码。默认设置是 `false`
- `url-pattern` — 被用来指定过滤哪些请求，默认是所有请求。

25.1.4.5. RichFaces

如果在你的工程中使用了Ajax4jsf，Seam会确保在装载其他所有内建过滤器之前，为你装载Ajax4jsf过滤器。你不必亲自在 `web.xml` 中装载Ajax4jsf过滤器。

只有当你的工程中有RichFaces jar包时，才会安装RichFaces Ajax过滤器。

为了覆盖默认的设置，要在 `components.xml` 中增加以下项。这些选项与RichFaces开发者指南中指定的一样：

```
<web:ajax4jsf-filter force-parser="true"
    enable-cache="true"
    log4j-init-file="custom-log4j.xml"
    url-pattern="*.seam"/>
```

- `force-parser` — 强制Richfaces的XML语法检验器去验证所有的JSF页面。如果为 `false`，则只有AJAX响应才被验证并被转换成合适的XML。设置`force-parser` 为 `false` 可以提高性能，但会在AJAX更新时提供可视化的工件。
- `enable-cache` — 启用框架生成资源的缓存（如，JavaScript，CSS，Images等）。当开发定制的JavaScript或CSS时，设置为`true`以阻止浏览器缓存这些资源。
- `log4j-init-file` — 被用于安装 per-application 记录。应该提供一个相对于web程序上下文的 `log4j.xml` 配置文件路径。
- `url-pattern` — 用来指定过滤哪些请求，默认是所有请求。

25.1.4.6. Identity Logging

这个过滤器将被验证的用户名添加到log4j映射诊断上下文中，以便如果喜欢，可能通过在模式中添加`%X{username}`，使它能够被包含在格式化过的日志输出中。

默认情况下，记录过滤器会处理所有请求，但是这一行为可以通过在 `components.xml` 中添加 `<web:logging-filter>` 项来进行调整，如下面的例子所示：

```
<components xmlns="http://jboss.com/products/seam/components"
    xmlns:web="http://jboss.com/products/seam/web">
    <web:logging-filter url-pattern="*.seam"/>
</components>
```

- `url-pattern` — Used to specify which requests the filter is active for. The default is all requests. `url-pattern` 用来指定该过滤器是为哪些请求而激活的。默认是所有请求。

25.1.4.7. 定制Servlet的上下文管理

直接发送到某些Servlet而不是JSF Servlet的请求不在JSF生命周期中被处理，因此Seam提供了一个Servlet过滤器供其他的Servlet使用，这些Servlet需要访问Seam组件。

这个过滤器允许定制的Servlet与Seam上下文交互。它在每个请求的开始设立Seam上下文，并在请求结束时卸掉它们。你必须确保这个过滤器 永远不 被用于JSF FacesServlet。Seam在JSF请求中为上下文管理使用阶段监听器（phase listener）。

默认没有装载这个过滤器，它需要由 `components.xml` 中的一个项来启用：

```
<web:context-filter url-pattern="/media/*"/>
```

- `url-pattern` — 被用来指定过滤哪些请求，默认是所有请求。如果上下文过滤器的URL模式已指定，则将启用该过滤器（除非明确取消）。

上下文过滤器期望在名称为 `conversationId` 的请求参数中，找到所有对话上下文的对话id。你要负责确保对话ID在请求中被发送。

你还需要负责确保将所有新的对话id传送回客户端。Seam将对话id作为内建的 `conversation` 组件的一个属性暴露出来。

25.1.4.8. 增加定制的过滤器

Seam能为你装载过滤器，允许你指定过滤器要放在过滤器链中的 什么位置 （如果你在 `web.xml` 中指定自己的过滤器，servlet规范就没有提供一个定义良好的顺序）。只需要给你的Seam组件加上 `@Filter` 注解（你的Seam组件必须实现 `javax.servlet.Filter`）。

```
@Startup
@Scope(APPLICATION)
@Name("org.jboss.seam.web.multipartFilter")
@BypassInterceptors
@Filter(within="org.jboss.seam.web.ajax4jsfFilter")
public class MultipartFilter extends AbstractFilter {
```

增加 `@Startup` 注解意味着该组件在Seam启动时有效；双向注入（bijection）在这里无效（`@BypassInterceptors`）；并且该过滤器在链中应该比Ajax4jsf过滤器（`@Filter(within="org.jboss.seam.web.ajax4jsfFilter")`）更靠后一些。

25.1.5. 将Seam与你的EJB容器集成

我们需要将 `SeamInterceptor` 用于我们的Seam组件。在整个程序中完成这个的最简单方式是在 `ejb-jar.xml` 中增加以下拦截器配置：

```
<interceptors>
  <interceptor>
    <interceptor-class>org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
  </interceptor>
</interceptors>

<assembly-descriptor>
  <interceptor-binding>
    <ejb-name>*</ejb-name>
    <interceptor-class>org.jboss.seam.ejb.SeamInterceptor</interceptor-class>
```

```
</interceptor-binding>
</assembly-descriptor>
```

Seam需要知道在JNDI中到哪里去寻找会话Bean。其中一种方法是在每个会话Bean Seam组件中指定@JndiName注解。然而，这样相当乏味。更好的一种方式是指定一种模式，Seam可以根据这个模式从EJB名称里推算出JNDI的名称。不幸的是，在EJB3规范里面，没有定义到全局JNDI的标准映射，所以这个映射是特定于供应商的。我们通常在 components.xml 中指定这个选项。

对于JBoss AS，以下的模式是正确的：

```
<core:init jndi-name="myEarName/#{ejbName}/local" />
```

myEarName就是部署Bean所在EAR的名称。

在一个EAR的外面（当使用嵌入式的JBoss的EJB3容器时），以下模式就是要用到的那个：

```
<core:init jndi-name="#{ejbName}/local" />
```

你不得不尝试着去找到其他应用程序服务器的正确配置。注意有些服务器（比如GlassFish）要求你为所有EJB组件直接（而且很乏味地）指定JNDI名称。在这种情况下，你可以选择你自己的模式。:-)

在一个EJB3的环境中，我们建议给事务管理使用一个特殊的内建组件，它完全知道容器事务，并且可以正确地处理用 Events 组件注册的事务成功事件。如果你没有在你的 components.xml 文件中增加这几行，Seam就不会知道容器管理的事务什么时候结束：

```
<transaction:ejb-transaction/>
```

25.1.6. 切记！

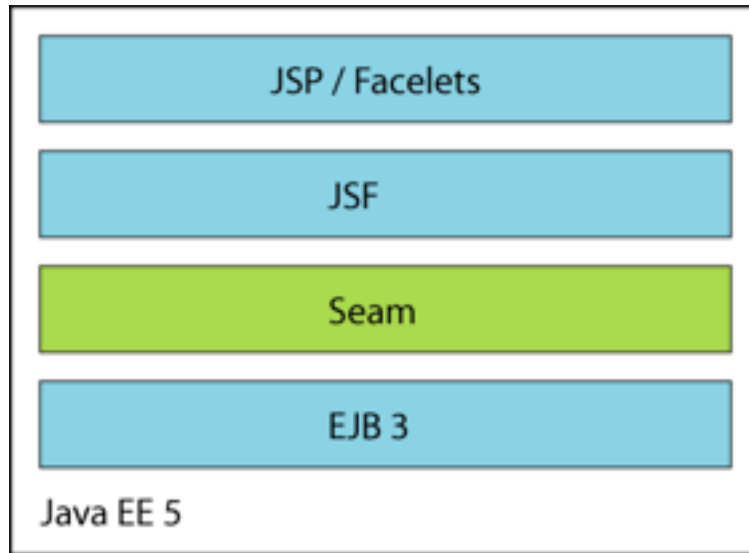
还有最后一点你要知道的。你必须在部署你的Seam组件的每个压缩文档中（即使空的属性文件也要这样），放置一个 seam.properties、META-INF/seam.properties 或META-INF/components.xml文件。在启动时，Seam将会根据 seam.properties 文件扫描所有压缩档案寻找Seam组件。

在一个Web压缩档案（WAR）文件中，如果在 WEB-INF/classes 目录下，包含有任何Seam组件，你必须在该目录中放置一个 seam.properties文件。

这就是为什么所有的Seam示例程序都有一个空的 seam.properties 文件的原因。你不能删掉这些文件，还期待一切仍然正常运行！

你可能会觉得这很愚蠢，并且框架的设计者做一个空文件来影响他们的软件该是多么的白痴啊？当然，这是针对Java虚拟机的限制所采取的一种权宜之计——如果我们不使用这个机制，那么我们的下一个最佳选项将逼迫你显式地在 components.xml 中列出所有的组件，就像其他某些竞争框架所做的那样！我想你会喜欢更喜欢我们这种方式。

25.2. 在Java EE 5中配置Seam



如果你是在Java EE 5的环境下运行，这就是开始使用Seam所需的全部配置！

25.2.1. 打包

一旦你将所有这些东西都打包到一个EAR文件中，该压缩文档看起来就会像这样：

```
my-application.ear/
  jboss-seam.jar
  lib/
    jboss-el.jar
  META-INF/
    MANIFEST.MF
    application.xml
  my-application.war/
    META-INF/
      MANIFEST.MF
    WEB-INF/
      web.xml
      components.xml
      faces-config.xml
    lib/
      jsf-facelets.jar
      jboss-seam-ui.jar
    login.jsp
    register.jsp
    ...
  my-application.jar/
    META-INF/
      MANIFEST.MF
      persistence.xml
    seam.properties
    org/
      jboss/
        myapplication/
          User.class
          Login.class
          LoginBean.class
          Register.class
          RegisterBean.class
          ...
```

你要在 META-INF/application.xml 中将 jboss-seam.jar 声明为一个EJB模块。 jboss-el.jar 应该被放置在EAR的lib目录下（将它放在EAR的classpath中）。

如果你想使用jBPM或Drools，你必须在EAR的lib目录下包含所需要的jar文件。

如果你想使用Facelets（我们建议使用），你必须在WAR的 WEB-INF/lib 目录下包含 jsf-facelets.jar

如果你想使用Seam标签库（大多数Seam应用程序都这么做），你就必须在WAR的 WEB-INF/lib 目录下包含 jboss-seam-ui.jar。如果你想用PDF或者Email标签库，则需要在 WEB-INF/lib 目录下放置 jboss-seam-pdf.jar 或 jboss-seam-mail.jar。

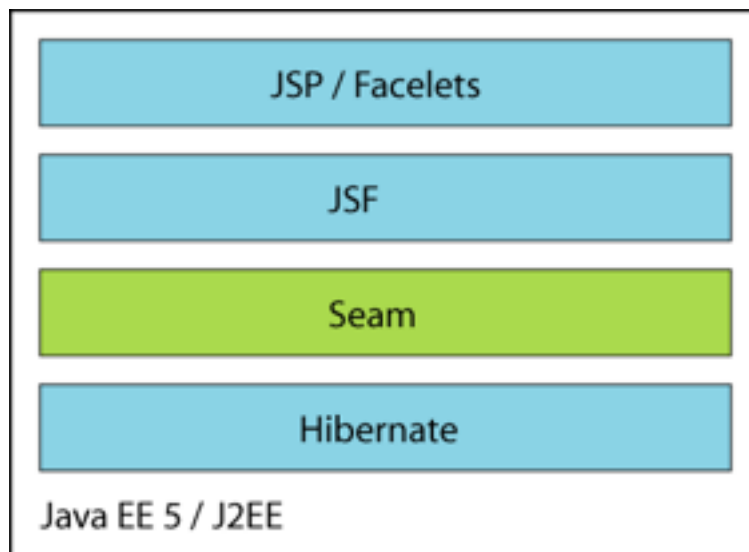
如果你想要使用Seam调试页面（只在使用Facelets的应用程序中有效），你必须在WAR的 WEB-INF/lib 目录下包含 jboss-seam-debug.jar

Seam发行时包括几个示例程序，他们能在任何支持EJB 3.0的Java EE容器中部署。

我真的希望这就是配置专题所要说的全部内容，可惜的是，我们只涉及到了它的三分之一。如果你实在无法忍受所有这些关于配置的废话，尽管放心地跳过本节剩下的部分，以后再回来这里。

25.3. 在J2EE中配置Seam

Seam很有用，即使你还没有冒险尝试EJB 3.0。在这种情况下，你可以使用Hibernate3或者JPA代替EJB 3.0持久化技术，并用简单的JavaBeans代替会话Bean。你可能会错过会话bean的某些很好的特性，但是当你准备好时，这将会很容易迁移到EJB 3.0，同时，你将能利用Seam独特的声明式状态管理架构。



Seam JavaBean组件不会像会话Bean那样提供声明式事务划分。你可以使用 JTA UserTransaction 手动管理你的事务，或者使用Seam的 @Transactional 标注声明。但是当与JavaBean一起使用Hibernate时，大部分应用程序只会用到由Seam管理的事务。

Seam发行包里包括预订示例程序的一种使用Hibernate3和JavaBeans而不是EJB3的版本，以及另一种使用JPA和JavaBeans的版本。这些示例程序可以随时部署到任何J2EE应用程序服务器中。

25.3.1. 在Seam中引导Hibernate

如果你装载了一个内建的组件，Seam将会从你的 hibernate.cfg.xml 文件中引导一个 Hibernate SessionFactory：

```
<persistence:hibernate-session-factory name="hibernateSessionFactory"/>
```

如果你想要通过注入得到一个由Seam管理的Hibernate Session，还需要配置一个 managed session

。

```
<persistence:managed-hibernate-session name="hibernateSessionFactory"
    session-factory="#{hibernateSessionFactory}"/>
```

25.3.2. 在Seam中引导JPA

如果你装载了一个内建的组件，Seam将会从你的 persistence.xml 文件中引导一个 JPA

EntityManagerFactory：

```
<persistence:entity-manager-factory name="entityManagerFactory"/>
```

如果你想要通过注入得到一个由Seam管理的JPA EntityManager，还需要配置一个 managed persistence context。

```
<persistence:managed-persistence-context name="entityManager"
    entity-manager-factory="#{entityManagerFactory}"/>
```

25.3.3. 打包

我们可以将我们的应用程序按照以下结构打包成一个WAR文件：

```
my-application.war/
  META-INF/
    MANIFEST.MF
  WEB-INF/
    web.xml
    components.xml
    faces-config.xml
    lib/
      jboss-seam.jar
      jboss-seam-ui.jar
      jboss-el.jar
      jsf-facelets.jar
      hibernate3.jar
      hibernate-annotations.jar
      hibernate-validator.jar
      ...
    my-application.jar/
      META-INF/
        MANIFEST.MF
      seam.properties
      hibernate.cfg.xml
      org/
        jboss/
          myapplication/
            User.class
            Login.class
            Register.class
            ...
  login.jsp
  register.jsp
  ...
```

如果我们想要将Hibernate部署在一个非EE的环境中（如Tomcat或TestNG），我们需要多做一些工作。

25.4. 在Java SE中配置Seam，没有内嵌JBoss

完全在一个EE环境之外使用Seam是有可能的。在这种情况下，你需要告诉Seam怎样去管理事务，因为没有JTA可用。如果你在使用JPA，就可以告诉Seam去使用JPA本地资源的事务，如EntityTransaction，像这样：

```
<transaction:entity-transaction entity-manager="#{entityManager}"/>
```

如果你在使用Hibernate，就可以告诉Seam像下面这样使用Hibernate事务API：

```
<transaction:hibernate-transaction session="#{session}"/>
```

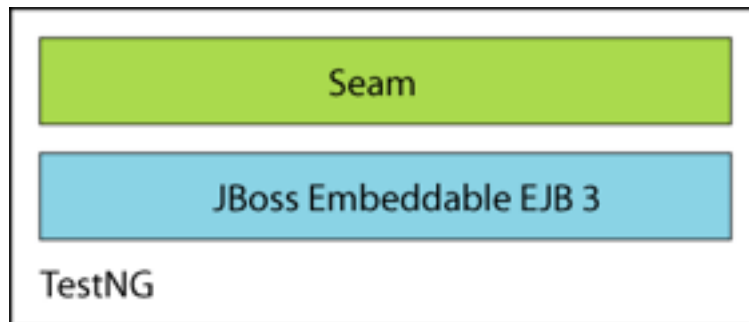
当然，你还需要去定义一个数据源。

一种更好的替代方案是使用嵌入式的JBoss去访问EE的API。

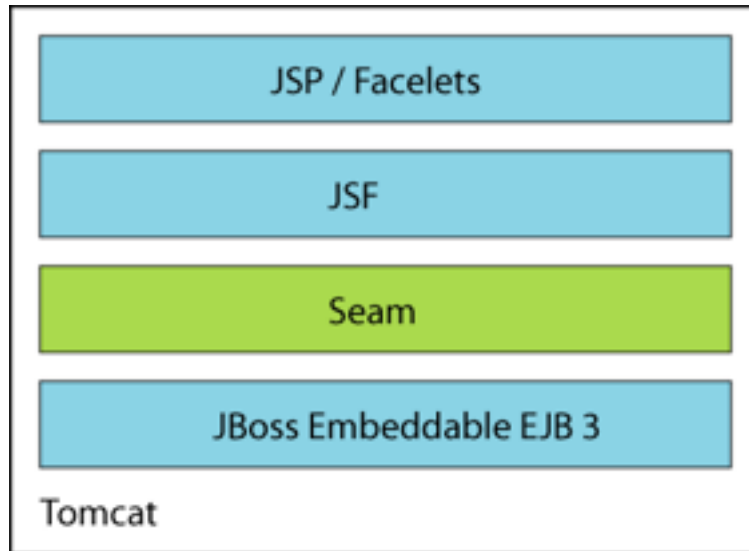
25.5. 用嵌入式的JBoss在Java SE中配置Seam

嵌入式的JBoss让你可以在Java EE 5应用程序服务器的上下文的外面运行EJB3组件。这个不但对测试有用，而且是特别有用。

Seam的booking示例程序包括一个通过 SeamTest 在嵌入式的JBoss上运行的TestNG集成测试套件。



booking示例程序甚至可以在Tomcat上部署。



25.5.1. 安装嵌入式的JBoss

为了让Seam应用程序在Tomcat上正确运行，必须将嵌入式JBoss装载到Tomcat中。嵌入式的JBoss可以在[这里](#)下载。将嵌入式的JBoss安装到Tomcat 6的过程非常简单。首先，你必须将嵌入式JBoss的JAR和配置文件都拷贝到Tomcat中。

- 将嵌入式JBoss的 `bootstrap` 和 `lib` 目录下的所有文件和目录，除了 `jndi.properties` 文件之外，都拷贝到Tomcat的 `lib` 目录下。
- 从Tomcat的 `lib` 目录中移除 `annotations-api.jar` 文件。

接下来，需要更新两个配置文件，用来增加嵌入式JBoss特有的功能。

- 将嵌入式JBoss listener增加到 `conf/server.xml` 中。它在文件中应该排列在所有其他listener的后面。

```
<Listener className="org.jboss.embedded.tomcat.EmbeddedJBossBootstrapListener" />
```

- 应该通过增加一个listener到 `conf/context.xml` 文件中来启用WAR文件扫描功能。

```
<Listener className="org.jboss.embedded.tomcat.WebinfScanner" />
```

关于更多的配置选项，请参考Tomcat集成嵌入式JBoss [Wiki条目](#)。

25.5.2. 打包

一个在Servlet引擎（如Tomcat）中基于WAR部署的压缩文档，它的结构有时看起来像这样：

```
my-application.war/
  META-INF/
    MANIFEST.MF
  WEB-INF/
    web.xml
    components.xml
    faces-config.xml
    lib/
```



```

jboss-seam.jar
jboss-seam-ui.jar
jboss-el.jar
jsf-facelets.jar
jsf-api.jar
jsf-impl.jar
...
my-application.jar/
    META-INF/
        MANIFEST.MF
        persistence.xml
    seam.properties
    org/
        jboss/
            myapplication/
                User.class
                Login.class
                LoginBean.class
                Register.class
                RegisterBean.class
                ...
login.jsp
register.jsp
...

```

绝大部分Seam示例程序可以通过运行 `ant deploy.tomcat` 部署到Tomcat中。

25.6. 在Seam中配置jBPM

Seam的jBPM集成没有被默认装载，因此你需要通过装载一个内建的组件来启用jBPM。你还需要显式地列出你的流程和页面流定义。在 `components.xml` 文件中：

```

<bpm:jbpm>
  <bpm:pageflow-definitions>
    <value>createDocument.jpdl.xml</value>
    <value>editDocument.jpdl.xml</value>
    <value>approveDocument.jpdl.xml</value>
  </bpm:pageflow-definitions>
  <bpm:process-definitions>
    <value>documentLifecycle.jpdl.xml</value>
  </bpm:process-definitions>
</bpm:jbpm>

```

如果你只有页面流，就不需要更多特殊的配置。如果你的确有业务流程定义，就需要为jBPM提供一个jBPM配置和一个Hibernate配置。Seam的DVD Store Demo包括了与Seam共同工作的 `jbpm.cfg.xml` 和 `hibernate.cfg.xml` 文件的例子：

```

<jbpm-configuration>

  <jbpm-context>
    <service name="persistence">
      <factory>
        <bean class="org.jbpm.persistence.db.DbPersistenceServiceFactory">
          <field name="isTransactionEnabled"><false/></field>
        </bean>
      </factory>
    </service>
    <service name="tx" factory="org.jbpm.tx.TxServiceFactory" />
    <service name="message" factory="org.jbpm.msg.db.DbMessageServiceFactory" />
    <service name="scheduler" factory="org.jbpm.scheduler.db.DbSchedulerServiceFactory" />
  </jbpm-context>
</jbpm-configuration>

```

```

<service name="logging" factory="org.jboss.logging.db.DbLoggingServiceFactory" />
<service name="authentication"
        factory="org.jboss.security.authentication.DefaultAuthenticationServiceFactory" />
</jboss-context>

</jboss-configuration>

```

在这儿需要注意的最重要事情是JBPM事务处理控制功能是禁用的。Seam或EJB3应该控制JTA事务。

25.6.1. 打包

对于JBPM配置和流程 / 页面流定义文件，还没有任何明确的打包格式。在Seam的示例程序中，我们决定简单地将所有这些文件打包到EAR的根目录下。在以后，我们可能设计其他一些标准的打包格式。因此EAR看起来有点像这样：

```

my-application.ear/
  jboss-seam.jar
  lib/
    jboss-el.jar
    jbp-3.1.jar
  META-INF/
    MANIFEST.MF
    application.xml
  my-application.war/
    META-INF/
      MANIFEST.MF
    WEB-INF/
      web.xml
      components.xml
      faces-config.xml
      lib/
        jsf-facelets.jar
        jboss-seam-ui.jar
      login.jsp
      register.jsp
      ...
  my-application.jar/
    META-INF/
      MANIFEST.MF
      persistence.xml
    seam.properties
    org/
      jboss/
        myapplication/
          User.class
          Login.class
          LoginBean.class
          Register.class
          RegisterBean.class
          ...
    jbp-3.1.jar
    hibernate.cfg.xml
    createDocument.jpdl.xml
    editDocument.jpdl.xml
    approveDocument.jpdl.xml
    documentLifecycle.jpdl.xml

```

25.7. 在Portal中配置Seam

要将Seam程序作为一个portlet运行，你需要提供某些portlet元数据（portlet.xml，等）作为对通常Java EE元数据的补充。参考 examples/portal 目录，作为booking demo预配置以便在JBoss Portal上运行的一个例子。

25.8. 在JBoss AS中配置SFSB和会话超时

将有状态会话Bean的超时值设置得比HTTP的超时值高一些是很重要的，否则SFSB可能在HTTP会话结束前已经超时。JBoss程序服务器有一个30分钟的默认会话Bean超时值，它在 server/default/conf/standardjboss.xml（用你自己的配置取代 default）中配置。

默认的SFSB超时值可以通过修改 LRUStatefulContextCachePolicy 缓存配置中 max-bean-life 的值得到调整。

```
<container-cache-conf>
  <cache-policy>org.jboss.ejb.plugins.LRUStatefulContextCachePolicy</cache-policy>
  <cache-policy-conf>
    <min-capacity>50</min-capacity>
    <max-capacity>1000000</max-capacity>
    <remover-period>1800</remover-period>

    <!-- SFSB timeout in seconds; 1800 seconds == 30 minutes -->
    <max-bean-life>1800</max-bean-life>

    <overager-period>300</overager-period>
    <max-bean-age>600</max-bean-age>
    <resizer-period>400</resizer-period>
    <max-cache-miss-period>60</max-cache-miss-period>
    <min-cache-miss-period>1</min-cache-miss-period>
    <cache-load-factor>0.75</cache-load-factor>
  </cache-policy-conf>
</container-cache-conf>
```

可以为JBoss 4.0.x在 server/default/deploy/jbossweb-tomcat55.sar/conf/web.xml 中修改，或者为JBoss 4.2.x在 server/default/deploy/jboss-web.deployer/conf/web.xml 中修改默认的HTTP会话超时值。这个文件中的以下项控制着所有Web应用程序的默认会话超时值：

```
<session-config>
  <!-- HTTP Session timeout, in minutes -->
  <session-timeout>30</session-timeout>
</session-config>
```

要为你自己的应用程序覆盖这个值，只需在你应用程序自己的 web.xml 文件中包含这个项。

第 26 章 Seam on OC4J

OC4J（用于Java的Oracle容器）11g（目前是一个“技术预览”版本）是Oracle的JEE 5应用服务器。我们将从Seam自带的酒店预订（Hotel Booking）的示例应用程序开始，来了解它的构建和部署，接着了解如何部署seam-gen生成的工程。这个工程将集成Seam、RichFaces Ajax和组件、Seam Security（包含Drools）、Facelets和Hibernate提供的JPA。

本节要求你使用OC4J 11g技术预览版（不是OC4J 10g）。你可以从<http://www.oracle.com/technology/tech/java/oc4j/11/> 下载OC4J 11g。

26.1. jee5/booking 实例

jee5/booking 实例基于一个（运行在JBoss AS中的）酒店预订的例子。它是被设计为Glassfish开箱即用，但是可以很容易地构建它，并运行在OC4J上。

26.1.1. 预订酒店实例的依赖包

首先，让我们来看一下预订实例的依赖包。有了这方面的知识，我们就可以了解由于OC4J的引入，带来了哪些额外的依赖包。

- jboss-seam.jar — 我们把这个声明为一个EJB3模块（为什么呢？Seam需要能够和容器管理事务CMT交互；它实现为一个EJB3的有状态会话Bean。）
- jboss-el.jar
- jboss-seam-ui.jar — Seam的JSF Control，依赖Apache的commons-beanutils
- jboss-seam-debug.jar
- jsf-facelets.jar
- richfaces-api.jar — 它需要Apache commons-digester和commons-beanutils
- richfaces-impl.jar 和 richfaces-ui.jar — 需要Apache commons-digester和commons-beanutils

26.1.2. OC4J需要的额外依赖包

- Hibernate — 当然，我们决定使用Hibernate作为JPA提供者（而不是OC4J中自带的TopLink Essentials）

为了使用Hibernate作为JPA提供者，你需要三个jar包（hibernate3.jar、hibernate-annotations.jar、hibernate-entitymanager.jar）和他们的依赖包（jboss-common.jar、jboss-archive-browsing.jar和ejb3-persistence.jar）。你可以在Seam发行包的hibernate/lib目录中找到这些jar包。

- thirdparty-all.jar — 是Seam所依赖的第三方库的集合（如javassist）

在大多数应用程序服务器中运行Seam（例如JBoss AS或是Glassfish），你只需要包含一些你真正需要的依赖包（例如：如果你使用Seam Text，你就需要包含ANTLR）；但是，在OC4J中，由于它“奇

异的” classloading，你必须始终包含它们：

- antlr-2.7.6.jar — Seam Text需要的。（在本例中没有用到）。
- jbpmm-jpdl.jar — Seam的JBPM集成所需要的（在本例中没有用到）。
- Drools — Seam Security所需要的。尽管我们没有通过Drools使用Seam Security，但还是必须包含它。Drools由5个jar包组成 - drools-core-4.0.0.jar、drools-compiler-4.0.0.jar、janino-2.5.7.jar、mvel14-1.2rc1.jar 和 antlr-runtime-3.0.jar。在本例中没有用到Drools集成。

26.1.3. 配置文件的改变

只有一些地方需要更改：

web.xml

你需要在 web.xml 中声明所有的EJB。这是许多JEE 5应用程序服务器的一个无聊要求 — 例如OC4J和Glassfish。

```
<ejb-local-ref>
  <ejb-ref-name>
    jboss-seam-jee5/AuthenticatorAction/local
  </ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home/>
  <local>
    org.jboss.seam.example.booking.Authenticator
  </local>
  <ejb-link>AuthenticatorAction</ejb-link>
</ejb-local-ref>
```

persistence.xml

你需要提供一个正确的配置给你的JPA实现。我们是使用Hibernate，由于OC4J还是绑定老的ANTLR，我们需要使用可选的查询工厂，我们也想使用OC4J的事务管理器：

```
<property
  name="hibernate.query.factory_class"
  value="org.hibernate.hql.classic.ClassicQueryTranslatorFactory" />
<property
  name="hibernate.transaction.manager_lookup_class"
  value="org.hibernate.transaction.OrionTransactionManagerLookup" />
```

26.1.4. 构建 jee5/booking 实例

1. 在工程中修改以下文件：

- build.xml — 反注释OC4J相关的依赖包
- resources/META-INF/persistence.xml — 注释掉Glassfish属性，反注释OC4J的属性

2. 通过运行 ant 来构建demo应用程序。最终构建成 dist/jboss-seam-jee5.ear

3. 将 `hsqldb.jar` 复制到OC4J中: `cp ../../seam-gen/lib/hsqldb.jar $ORACLE_HOME/j2ee/home/applib/` (由于OC4J并没有嵌入式数据库, 因此我们决定使用HSQLDB)。

26.2. 部署Seam应用程序到OC4J中

这个迷你型的教程(有点乏味)描述了将一个JEE 5应用程序部署到OC4J中所需的步骤。假设你正在部署使用了嵌入式的`hsqldb`数据库的 `jee5/booking` 实例。为了部署其他的应用程序, 你需要更改数据源名称和应用程序名称。

1. 下载和解压OC4J
2. 请确认你已经设置了 `$JAVA_HOME` 和 `$ORACLE_HOME` 作为环境变量 (`$ORACLE_HOME`这个是指解压后OC4J的目录名称)。想了解更多有关安装OC4J的信息, 请参考OC4J发布包中的 `Readme.txt`。
3. 编辑OC4J数据源 `$ORACLE_HOME/j2ee/home/config/data-sources.xml`, 并在 `<data-sources>` 中增加

```
<managed-data-source
  connection-pool-name="jee5-connection-pool"
  jndi-name="jdbc/__default"
  name="jee5-managed-data-source" />
<connection-pool name="jee5-connection-pool">
  <connection-factory
    factory-class="org.hsqldb.jdbcDriver"
    user="sa"
    password=""
    url="jdbc:hsqldb:." />
</connection-pool>
```

在 `persistence.xml` 里, `jndi-name` 被用作 `jta-data-source`。

4. 编辑 `$ORACLE_HOME/j2ee/home/config/server.xml`, 在 `<application-server>`中增加

```
<application name="jboss-seam-jee5"
  path="../../home/applications/jboss-seam-jee5.ear"
  parent="default"
  start="true" />
```

为了让事情简单些, 就采用你给项目所用的相同名字。

5. 编辑 `$ORACLE_HOME/j2ee/home/config/default-web-site.xml`, 在 `<web-site>` 中增加

```
<web-app application="jboss-seam-jee5"
  name="jboss-seam-jee5"
  load-on-startup="true"
  root="/seam-jee5" />
```

`root` 就是你将输入到Web浏览器用来访问应用程序的上下文路径。

6. 将应用程序复制到OC4J: `cp dist/jboss-seam-jee5.ear $ORACLE_HOME/j2ee/home/applications/`
7. 启动OC4J: `$ORACLE_HOME/bin/oc4j -start`

如果是第一次启动OC4J, 您将被要求设置管理员密码

8. 在 `http://localhost:8888/seam-jee5` 中检验应用程序
9. 你可以通过在服务器运行的控制台上按下 `CTRL-C` 来停止服务器。

26.3. 将一个使用 `seam-gen` 创建的应用程序部署到OC4J中。

接下来的说明假设你正在使用命令行的方式和一个简单的文本编辑器，当然你也可以使用自己熟悉的IDE — `seam-gen` 项目，同时支持Eclipse和Netbeans。

我们从使用 `seam-gen` 来创建一个非常简单的应用程序开始。 `seam-gen` 使用Hibernate Tools 来将数据库的Schema反向工程为JPA的实体Bean； 它也可以生成Seam应用程序的框架组件和JSF的CRUD（创建、读取、更新和删除）视图。 这个教程使用MySQL数据库（当然你也可以使用任何其他数据库，更改相应的SQL就可以了）； 安装、配置和运行MySQL，然后创建数据库，并带有一些范例数据。

接下来，在Seam的目录中运行 `./seam setup`。

```
> ./seam setup
Buildfile: build.xml

setup:
[echo] Welcome to seam-gen :-)
[input] Enter your Java project workspace (the directory that contains your Seam projects) [/home/pmuir/workspace] [/home/pmuir/workspac

[input] Enter your JBoss home directory [/home/pmuir/java/jboss-4.2.1.GA] [/home/pmuir/java/jboss-4.2.1.GA]

[input] Enter the project name [oc4j-example] [oc4j-example]

[input] Is this project deployed as an EAR (with EJB components) or a WAR (with no EJB support) [ear] ([ear], war, )

[input] Enter the Java package name for your session beans [org.jboss.seam.tutorial.oc4j.action] [org.jboss.seam.tutorial.oc4j.act

[input] Enter the Java package name for your entity beans [org.jboss.seam.tutorial.oc4j.model] [org.jboss.seam.tutorial.oc4j.model

[input] Enter the Java package name for your test cases [org.jboss.seam.tutorial.oc4j.test] [org.jboss.seam.tutorial.oc4j.test]

[input] What kind of database are you using? [mysql] (hsqldb, [mysql], oracle, postgres, mssql, db2, sybase, enterprisedb, )

[input] Enter the Hibernate dialect for your database [org.hibernate.dialect.MySQLDialect] [org.hibernate.dialect.MySQLDialect]

[input] Enter the filesystem path to the JDBC driver jar [lib/mysql.jar] [lib/mysql.jar]

[input] Enter JDBC driver class for your database [com.mysql.jdbc.Driver] [com.mysql.jdbc.Driver]

[input] Enter the JDBC URL for your database [jdbc:mysql:///oc4j] [jdbc:mysql:///oc4j]

[input] Enter database username [user] [user]

[input] Enter database password [password] [password]

[input] skipping input as property hibernate.default_schema.new has already been set.
[input] Enter the database catalog name (it is OK to leave this blank) [] []

[input] Are you working with tables that already exist in the database? [y] ([y], n, )

[input] Do you want to drop and recreate the database tables and data in import.sql each time you deploy? [n] (y, [n], )

[propertyfile] Updating property file: /home/pmuir/workspace/jboss-seam/seam-gen/build.properties
[echo] Installing JDBC driver jar to JBoss server
[echo] Type 'seam new-project' to create the new project
```

BUILD SUCCESSFUL

输入 `./seam new-project` 来创建你的工程，并 `cd` 进入到刚创建的工程里。

输入 `./seam generate-entities` 来运行创建的实体，Seam应用程序框架类和相关的视图。

我们现在需要对生成的工程进行一些修改。让我们从配置文件开始：

`resources/META-INF/persistence-dev.xml`

- 更改 `jta-data-source` 为 `jdbc/_oc4jExample` （当在 `data-sources.xml` 创建数据源时，用这个作为 `jndi-name` ）
- 添加上述的属性：

```
<property name="hibernate.query.factory_class"
  value="org.hibernate.hql.classic.ClassicQueryTranslatorFactory" />
<property name="hibernate.transaction.manager_lookup_class"
  value="org.hibernate.transaction.OrionTransactionManagerLookup" />
<property name="hibernate.transaction.flush_before_completion"
  value="true"/>
<property name="hibernate.cache.provider_class"
  value="org.hibernate.cache.HashtableCacheProvider"/>
```

- 删除JBoss AS中暴露EntityManagerFactory的方法：

```
<property
  name="jboss.entity.manager.factory.jndi.name"
  value="java:/oc4j-exampleEntityManagerFactory">
```

- 同样地，如果你想使用先前的文件部署到OC4J中，就需要更改 `persistence-prod.xml` 。

`resources/META-INF/jboss-app.xml`

你可以删除这个文件，由于我们没有部署到JBoss AS中（`jboss-app.xml` 用于在JBoss AS中激活 `classloading` 隔离）。

`resources/*-ds.xml`

你可以删除这些文件，由于我们也没有部署到JBoss AS中（在JBoss AS中这些文件定义了数据源，在OC4J中，你必须编辑主要的 `data-sources.xml` 文件。）

`resources/WEB-INF/components.xml`

- 激活CMT（container managed transaction：容器管理事务）集成 — 添加 `<transaction:ejb-transaction />` 组件， 和它的命名空间声明 `xmlns:transaction="http://jboss.com/products/seam/transaction"`
- 将 `jndi-pattern` 更改为 `java:comp/env/oc4j-example/#{ejbName}/local`
- 我们想在我们的应用程序当中使用Seam的MPC（Managed Persistence Context：持久化上下文管理）。不幸的是，OC4J没有以JNDI形式暴露EntityManagerFactory，但是Seam提供了一个内置的管理器组件：

```
<persistence:entity-manager-factory
  auto-create="true"
  name="oc4jEntityManagerFactory">
```



```
persistence-unit-name="oc4j-example" />
```

接下来我们需要告知Seam来使用它，因此我们更改注入实体管理器工厂的
managed-persistence-context:

```
<persistence:managed-persistence-context
  name="entityManager"
  auto-create="true"
  entity-manager-factory="#{oc4jEntityManagerFactory}" />
```

resources/WEB-INF/web.xml

在这里，你需要声明你所有的EJB。记住包括Seam的容器管理事务集成：

```
<ejb-local-ref>
  <ejb-ref-name>
    oc4j-example/EjbSynchronizations/local
  </ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local>
    org.jboss.seam.transaction.LocalEjbSynchronizations
  </local>
  <ejb-link>EjbSynchronizations</ejb-link>
</ejb-local-ref>
```

build.xml

改变默认的target来完成（我们不打算覆盖OC4J的自动部署）。

现在，让我们来添加一些额外的依赖包：

- Hibernate —

- 将Seam发行包中 hibernate/lib 目录下所有的jar文件拷贝到 oc4j-example/lib 目录下：
cp ../jboss-seam/hibernate/lib/*.jar lib/
- 更改 build.xml，将它们包含在ear包中 — 要添加的这些包括在其他备份库下方：

```
<include name="lib/hibernate-annotations.jar" />
<include name="lib/hibernate-entitymanager.jar" />
<include name="lib/hibernate3.jar" />
<include name="ejb3-persistence.jar" />
<include name="lib/jboss-archive-browsing.jar" />
<include name="lib/jboss-common.jar" />
```

- thirdparty-all.jar — 更改 build.xml 包含它 — 添加这个：

```
<include name="lib/thirdparty-all.jar" />
```

- antlr-2.7.6.jar — 更改 build.xml 包含它 — 添加这个：

```
<include name="lib/antlr-*.jar" />
```

- 由于我们使用Drools来提供Seam Security规则，我们需要添加在Eclipse JDT编译器中（在JBoss AS中你不需要这个；再说一次，这是由于OC4J的classloading造成的）：

```
cp ../jboss-seam/seam-gen/lib/org.eclipse.jdt.core*.jar lib/
```

- 更改 build.xml 将它们包含在ear包中：

```
<include name="lib/org.eclipse.jdt.core*.jar" />
```

你应该类似于下面这样结束：

```
<fileset dir="${basedir}">
  <!-- other libraries added by seam-gen -->
  <include name="lib/hibernate-annotations.jar" />
  <include name="lib/hibernate-entitymanager.jar" />
  <include name="lib/hibernate3.jar" />
  <include name="lib/jboss-archive-browsing.jar" />
  <include name="lib/jboss-common.jar" />
  <include name="lib/thirdparty-all.jar" />
  <include name="lib/antlr-*.jar" />
  <include name="lib/org.eclipse.jdt.core*.jar" />
</fileset>
```

最后，让我们将 User 实体连接到Seam Security中（我们有一个包含username 列和 password 列的 User 表）。我们将使验证器变成一个无状态会话Bean（毕竟OC4J也是EJB3的容器！）：

1.

- 添加 @Stateless 注解。
- 将类重命名为 AuthenticatorAction
- 创建一个具名为 Authenticator 的接口，并由 AuthenticatorAction 来实现这个接口（EJB3要求会话Bean要有一个本地接口）。使用 @Local 来注解这个接口，然后增加一个与 AuthenticatorAction 中的 authenticate 方法同样签名的方法。

```
@Name("authenticator") @Stateless public class
    AuthenticatorAction implements Authenticator {
```

```
@Local public interface Authenticator {
    public boolean authenticate();
}
```

2. 使用 @PersistenceContext 注解来注入EntityManager：

```
@PersistenceContext private EntityManager entityManager;
```

3. 实现authenticate：

```
public boolean authenticate() {
    List <User> users = entityManager.createQuery("select u from User u where
    u.username = #{identity.username} and
```

```

u.password = #{identity.password}) .getResultList();
if (users.size() == 1) {
    identity.addRole("admin");
    return true;
} else {
    return false;
}
}

```

4. 然后在 web.xml 中添加EJB3的引用:

```

<ejb-local-ref>
  <ejb-ref-name>
    oc4j-example/AuthenticatorAction/local
  </ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local>
    org.jboss.seam.tutorial.oc4j.action.Authenticator
  </local>
  <ejb-link>AuthenticatorAction</ejb-link>
</ejb-local-ref>

```

现在你可以继续并定制你自己的应用程序了。

26.3.1. seam-gen之类的应用程序的OC4J部署描述符

为了使用上述部署指令来部署你的应用程序，要结合使用这些部署描述符：

\$ORACLE_HOME/j2ee/home/config/data-sources.xml

```

<managed-data-source
  connection-pool-name="oc4j-example-connection-pool"
  jndi-name="jdbc/__oc4jExample"
  name="oc4j-example-managed-data-source" />
<connection-pool
  name="oc4j-example-connection-pool">
  <connection-factory
    factory-class="com.mysql.jdbc.Driver"
    user="username"
    password="password"
    url="jdbc:mysql:///oc4j" />
  </connection-pool>

```

\$ORACLE_HOME/j2ee/home/config/server.xml

```

<application name="oc4j-example"
  path="../../../home/applications/oc4j-example.ear"
  parent="default"
  start="true" />

```

\$ORACLE_HOME/j2ee/home/config/default-web-site.xml

```

<web-app application="oc4j-example"
  name="oc4j-example"
  load-on-startup="true"
  root="/oc4j-example" />

```

第 27 章 Seam注解

编写Seam应用程序时需要使用大量的注解。Seam让我们使用注解获得声明式编程风格。大部分注解由EJB3.0规范定义。数据验证通过Hibernate Validator包定义。最后，Seam定义了它自己的注解集合，这就是我们这一章将要描述的。

所有这些注解在 `org.jboss.seam.annotations` 包中定义。

27.1. 用于定义组件的注解

我们要看的第一组注解让我们定义一个Seam组件。这些注解在组件（component）类中出现。

@Name

```
@Name("componentName")
```

为一个类定义一个Seam组件。所有Seam组件都需要该注解。

@Scope

```
@Scope(ScopeType.CONVERSATION)
```

定义默认的组件上下文。可以定义的值由 `ScopeType` 枚举：EVENT, PAGE, CONVERSATION, SESSION, BUSINESS_PROCESS, APPLICATION, STATELESS。

当范围没有显式定义时，默认的范围取决于组件类型。对于无状态会话bean，默认是 STATELESS。对于Entity Bean和Stateful Session Bean，默认是 CONVERSATION。对于JavaBean，默认是 EVENT。

@Role

```
@Role(name="roleName", scope=ScopeType.SESSION)
```

允许一个Seam组件绑定多个上下文变量。@Name/@Scope 注解定义一个“默认角色”。每一个 @Role 注解定一个附加角色。

- name — 上下文变量的名字。
- scope — 上下文变量的作用域。当没有显式定义作用域时，和上面一样默认取决于组件类型。

@Roles

```
@Roles({  
    @Role(name="user", scope=ScopeType.CONVERSATION),  
    @Role(name="currentUser", scope=ScopeType.SESSION)  
})
```

允许指定多个额外角色。

@BypassInterceptors

```
@BypassInterceptors
```

取消在特定组件或者一个组件方法上的所有拦截器。

@JndiName

```
@JndiName("my/jndi/name")
```

Seam查找EJB组件的JNDI名。如果没有显式指定JNDI名，Seam将使用由 `org.jboss.seam.core.init.jndiPattern` 指定的JNDI模式。

@Conversational

```
@Conversational
```

声明一个对话作用域组件是对话式的，亦即只有长期运行的对话处于活动状态时，组件中的方法才可以被调用。

@Startup

```
@Scope(APPLICATION) @Startup(depends="org.jboss.seam.bpm.jbpm")
```

指定某个Application Scope的组件在初始化时立即启动。它主要用于特别的内置组件，用于引导象JNDI，数据源等等关键性的设施。

```
@Scope(SESSION) @Startup
```

指定某个Session Scope的组件在Session建立时立即启动。

- `depends` — 依赖于，指定必须在此之前启动的命名组件（如果已安装）。

@Install

```
@Install(false)
```

指定组件是否应该被默认安装。没有@Install注解则表明该组件应该被安装。

```
@Install(dependencies="org.jboss.seam.bpm.jbpm")
```

如果所指定的依赖组件被安装，那么该组件才安装。

```
@Install(genericDependencies=ManagedQueueSender.class)
```

如果所指定的类的某个实现组件被安装，那么该组件才安装。当无法确定依赖组件的唯一公开名字时，这就有用了。

```
@Install(classDependencies="org.hibernate.Session")
```

如果所指定的类在classpath中，那么该组件才安装。

```
@Install(precedence=BUILT_IN)
```

指定组件的优先级别。如果具有相同名字的多组件存在，具有高优先级的才被安装。定义的优先级是（递增排序）：

- BUILT_IN — 所有内置的Seam组件的优先级别
- FRAMEWORK — 用于扩展Seam的框架组件的优先级别
- APPLICATION — 应用程序的组件优先级别（默认优先级）
- DEPLOYMENT — 在特定部署中重载应用程序组件的组件优先级别
- MOCK — 在测试时mock对象使用的优先级别

@Synchronized

```
@Synchronized(timeout=1000)
```

如果组件被多个客户端并发访问，Seam应该串行化请求。如果一个请求在给定时间段内没有得到组件的锁，将抛出一个例外。

@ReadOnly

```
@ReadOnly
```

声明JavaBean组件或者组件方法在调用结束时不要求状态复制。

@AutoCreate

```
@AutoCreate
```

声明组件将被自动建立，即使客户端不定义 `create=true`。

27.2. 用于双向注入的注解

下面两个注解控制双向注入。这些属性用于组件实例变量或者属性访问方法中。

@In

```
@In
```

在每次组件调用开始时，从上下文变量注入此组件属性。如果上下文变量是null，那么一个异常将被抛出。

```
@In(required=false)
```

在每次组件调用开始时，从上下文变量注入此组件属性。上下文变量可为null。

```
@In(create=true)
```

在每次组件调用开始时，从上下文变量注入此组件属性。如果上下文变量为null，那么Seam实例化这个组件。

```
@In(value="contextVariableName")
```

显式指定上下文变量的名字，而不再使用注解定义的实例变量名。

```
@In(value="#{customer.addresses['shipping']}")
```

在每次组件调用开始时，用一个JSF EL表达式的计算结果来注入组件属性。

- **value** — 指定上下文变量名。默认是组件属性名。可选地，指定一个JSF EL表达式，放在#{...} 符号中。
- **create** — 指定若上下文变量名在所有上下文中均未定义，Seam应该创建一个组件作为上下文变量，名字即为所要求的名字。默认为false。
- **required** — 指定若上下文变量名在所有上下文中均未定义，Seam应抛出异常。

@Out

```
@Out
```

在调用结束后注射出Seam组件属性到上下文变量。若属性为null，则抛出一个异常。

```
@Out(required=false)
```

在调用结束后注射出Seam组件属性到上下文变量。属性可以为null。

```
@Out(scope=ScopeType.SESSION)
```

在调用结束后注射出非Seam组件属性到指定scope。

或者，若没有明确指定scope，则使用此 @Out 属性所属组件的scope。（如果此组件是无状态的，则使用 `EVENTSCOPE`。）

```
@Out(value="contextVariableName")
```

显式指定上下文变量名，而非使用注解中指定的实例变量名。

- **value** — 指定上下文变量名。默认为组件属性名。
- **required** — 指定若注射出时组件属性为null，Seam应抛出异常。

注意一起使用这些注解相当常见，例如：

```
@In(create=true) @Out private User currentUser;
```

下一个注解支持 管理器组件（manager component） 模式，在该模式中一个Seam组件管理一些其他将被注入的class实例的生命周期。它在组件的getter方法中出现。

@Unwrap

```
@Unwrap
```

指定注解的getter方法返回的对象是被注入的，而非组件实例本身。

下一个注解支持 工厂组件（factory component） 模式，在该模式中，一个Seam组件负责初始化上下文变量值。如果出现非faces的request，在渲染response的时候，它用于初始化所需要的状态特别有用。它出现在组件方法中。

@Factory

```
@Factory("processInstance") public void createProcessInstance() { ... }
```

说明当上下文变量没有值时，此组件的方法被用来初始化上下文变量值。它用于返回值是 void 的方法。

```
@Factory("processInstance", scope=CONVERSATION) public ProcessInstance createProcessInstance() { ... }
```

声明方法返回一个值，当上下文变量没有值时Seam应使用此值初始化命名上下文变量值。它用于返回一个值的方法。若没有指明scope，则使用 @Factory 方法所在组件的scope（除非组件是无状态的，则使用 EVENT 上下文）。

- value — 指定上下文变量值。若为getter方法，默认为JavaBean属性名。
- scope — 指定Seam应绑定返回值的作用域。仅针对于返回一个值的工厂方法有意义。
- autoCreate — 无论什么时候请求变量，此工厂方法将自动被调用，即使@In未指定create=true。

下面的注解让你注入一个 日志（Log）：

@Logger

```
@Logger("categoryName")
```

使用 org.jboss.seam.log.Log 的实例注入一个组件字段。对于Entity Bean，该字段必须声明为static。

- value — 指定日志category。默认是组件类名。

最后一个注解让你注入一个request参数值：

@RequestParam

```
@RequestParam("parameterName")
```

将request的参数值注入组件属性。基本类型的转化被自动地完成。

- value —指定request参数名。默认为组件属性名。

27.3. 关于组件生命周期方法的注解

这些注解允许组件响应它自己的生命周期事件。它们作用于组件方法。对每个组件class来说，每种注解只允许出现一次。

@Create

```
@Create
```

当组件实例被Seam初始化时，该方法应被调用。注意仅有JavaBean和Stateful Session Bean支持create方法。

@Destroy

```
@Destroy
```

当上下文结束和它的上下文变量销毁时，该方法应被调用。注意仅有JavaBean和Stateful Session Bean支持destroy方法。

Destroy方法应仅仅用于清理工作。Seam 会捕捉、记录，然后消灭destroy方法传播的任何异常。

@Observer

```
@Observer("somethingChanged")
```

指定当特定类型的component-driven（组件驱动）事件发生时，该方法应被调用。

```
@Observer(value="somethingChanged", create=false)
```

当指定类型的一个事件发生时，该方法应被调用，但若实例都不存在，则不创建实例。若实例不存在并且create是false，事件将不会觉察到。create的默认值是true。

27.4. 用于声明上下文的注解

这些注解提供声明式对话分界（declarative conversation demarcation）。它们在Seam组件方法中使用，通常是动作监听器方法（Action Listener Method）。

每个Web请求有一个对话上下文和它关联。这些对话的大多数在请求结束时结束。如果你想要一个对话跨越多个请求，你必须通过调用标志为 `@Begin` 的方法来“提升”当然的对话为一个长期运行的对话（long-running conversation）。

`@Begin`

```
@Begin
```

当此方法无异常的返回一个非空结果时，一个长期运行的对话开始。

```
@Begin(join=true)
```

若已经处于长期运行对话中，简单的延续此对话上下文。

```
@Begin(nested=true)
```

若已经处于长期运行对话中，一个新的被 嵌套（nested） 对话上下文开始。该被嵌套的对话在遇到下一个 `@End` 时结束，并且外部上下文将恢复。在同一个外部对话中同时嵌套多个对话是完全合法的。

```
@Begin(pageflow="process definition name")
```

指定该对话的页面流（pageflow）的jBPM进程定义名。

```
@Begin(flushMode=FlushModeType.MANUAL)
```

指定任何Seam管理的持久上下文的flush模式。`flushMode=FlushModeType.MANUAL` 支持 原子对话（atomic conversations），这里所有写操作在会话上下文进入队列，直到显式调用 `flush()`（调用通常发生在对话结束时）。

- `join` — 定义当长期对话已经存在时的行为。若是`true`，传播上下文。若为 `false`，抛出一个异常。默认为`false`。当指定 `nested=true` 时，将忽略该设置。
- `nested` — 当长期对话已经存在时，一个嵌套对话应该建立。
- `flushMode` — 设置任何在此会话期间创建的，被Seam管理的Hibernate Session或JPA持久上下文的flush模式。
- `pageflow` — 由 `org.jboss.seam.bpm.jbpm.pageflowDefinitions` 部署的一个jBPM处理的进程定义名。

`@End`

```
@End
```

当这个方法无异常的返回一个非空输出时，长期对话结束。

- `beforeRedirect` — 默认情况下，若有重定向，直到所有的重定向结束后，对话才会被真正destory。设置`beforeRedirect=true`指定该对话应在当前request结束时就结束，并且在一个新的

临时对话上下文中处理重定向。

@StartTask

```
@StartTask
```

“开始”一个 jBPM 任务。当此方法无异常的返回一个非空输出时，长期运行对话开始。此对话同在某 request 具名参数中被指定的 jBPM 任务相关联。在该会话上下文中，还定义了一个业务流程上下文（business process context），用作任务实例的业务流程实例。

jBPM 的 `TaskInstance` 在 request context 中以 `taskInstance` 的名字作为变量出现。jBPM 的 `ProcessInstance` 在 request context 中以 `processInstance` 的名字作为变量出现。（当然，这些对象也可用于通过 `@In` 注入。）

- `taskIdParameter` — 保存有 task id 的 request 参数的名字。默认为“taskId”，同时也是 Seam taskList JSF component 使用的默认值。
- `flushMode` — 设置任何在此对话期间创建的，被 Seam 管理的 Hibernate Session 或 JPA 持久上下文的 flush 模式。

@BeginTask

```
@BeginTask
```

恢复一个未完成的 jBPM 任务。当此方法无异常的返回一个非空值时，长时间运行的对话开始。此对话同在某 request 参数中指定的 jBPM 任务相关联。在该对话上下文中，还定义了一个业务流程上下文（business process context），用作任务实例的业务流程实例。

jBPM 的 `org.jbpm.taskmgmt.exe.TaskInstance` 在 request context 中以 `taskInstance` 的名字作为变量出现。jBPM 的 `org.jbpm.graph.exe.ProcessInstance` 在 request context 中以 `processInstance` 的名字作为变量出现。

- `taskIdParameter` — 保存有 task id 的 request 参数的名字。默认为“taskId”，同时也是 Seam taskList JSF component 使用的默认值。
- `flushMode` — 设置任何在此会话期间创建的，被 Seam 管理的 Hibernate Session 或 JPA 持久上下文的 flush 模式。

@EndTask

```
@EndTask
```

“结束”一个 jBPM 任务。当此方法无异常返回一个非空输出时，结束长时间运行的会话。触发一个 jBPM 流转（transition）。若没有调用 `transition` 内置组件的 `Transition.setName()` 方法，实际被触发的将是默认的 transition。

```
@EndTask(transition="transitionName")
```

触发给定 jBPM 流转。

- `transition` — 当任务结束时触发的jBPM流转名。默认为默认的流转`transition`。
- `beforeRedirect` — 默认情况下，若有重定向，直到所有的重定向结束后，会话才会被真正`destory`。设置 `beforeRedirect=true` 指定该会话应在当前`request`结束时就结束，并且在一个新的临时会话上下文中处理重定向。

@CreateProcess

```
@CreateProcess(definition="process definition name")
```

当方法无异常返回一个非空输出时，建立一个新的jBPM流程实例。 `ProcessInstance` 对象在上下文中以 `processInstance` 的名字作为一个变量出现。

- `definition` — 通过 `org.jboss.seam.bpm.jbpm.processDefinitions` 部署的jBPM 流程定义的名字。

@ResumeProcess

```
@ResumeProcess(processIdParameter="processId")
```

当方法无异常返回一个非空输出时，重新进入一个已存的jBPM 流程实例的`context`。
`ProcessInstance` 对象在上下文中以 `processInstance` 的名字作为一个变量出现。

- `processIdParameter` — 保存有该流程`id`的`request`参数名。默认是 `"processId"`。

@Transition

```
@Transition("cancel")
```

当此方法返回一个非空结果时，向在当前jBPM流程实例中发送一个流转信号。

27.5. 用于在J2EE环境中使用Seam JavaBean组件的注解

Seam提供一个注解，来让你根据某个动作监听器的输出强制回滚JTA事务。

@Transactional

```
@Transactional
```

声明JavaBean组件具有与Session Bean组件默认事务行为类似的行为。也就是说，方法调用应该发生在一个事务中，如果当调用方法时没有事务存在，一个事务将特地为该方法启动。此注解可以在类或者方法级应用。不要把此注解用于EJB3.0组件，那时候应该用 `@TransactionAttribute`！

@ApplicationException

```
@Transactional
```

TDB

@Interceptors

```
@Transactional
```

TDB

这些注解大多用在JavaBean Seam组件中。若你用EJB3.0组件，你应改采用标准的@TransactionAttribute 注解。

27.6. 用于异常的注解

这些注解让你指定Seam应如何处理一个从Seam组件中传播出来的异常。

@Redirect

```
@Redirect(viewId="error.jsp")
```

声明被注解的异常引起浏览器重定向到特定视图id。

- viewId — 指定重定向到的JSF视图id。你可以在这里使用EL。
- message — 指定显示的信息，默认是异常信息。
- end — 指明是否终止长时间运行的对话，默认为 false。

@HttpError

```
@HttpError(errorCode=404)
```

声明被注解的异常导致发送一个HTTP错误。

- errorCode — HTTP错误码，默认为500。
- message — 和HTTP错误一起被发送的信息，默认为异常消息。
- end — 指明是否终止长时间运行的对话，默认为 false。

27.7. 用于Seam Remoting 的注解

Seam Remoting要求会话Bean的本地接口要采用以下的注解：

@WebRemote

```
@WebRemote(exclude="path.to.exclude")
```

说明被注解的方法可以被客户端JavaScript脚本调用。 exclude 属性是可选项，用于从结果的

对象图中排除特定对象（Remoting一章有更详细信息）。

27.8. 用于Seam拦截器（interceptor）的注解

以下注解出现在Seam拦截器类中。

请查阅EJB3.0规范文档获得EJB拦截器定义所要求的注解的信息。

@Interceptor

```
@Interceptor(stateless=true)
```

指定这个拦截器是无状态的，Seam可以优化复制。

```
@Interceptor(type=CLIENT)
```

指定次拦截器是一个“客户端”拦截器，在EJB容器之前调用它。

```
@Interceptor(around={SomeInterceptor.class, OtherInterceptor.class})
```

指定此拦截器在栈中的位置比给定的拦截器更高。

```
@Interceptor(within={SomeInterceptor.class, OtherInterceptor.class})
```

指定此拦截器在栈中的位置比给定的拦截器更深。

27.9. 用于异步（asynchronicity）的注解

以下注解用于声明一个异步方法，例如：

```
@Asynchronous public void scheduleAlert(Alert alert, @Expiration Date date) { ... }
```

```
@Asynchronous public Timer scheduleAlerts(Alert alert,  
                                           @Expiration Date date,  
                                           @IntervalDuration long interval) { ... }
```

@Asynchronous

```
@Asynchronous
```

指定方法调用被异步处理。

@Duration

```
@Duration
```

此异步调用其中的一个参数，是在调用处理之前所进行的时间（若为嵌套调用，则是第一个被

处理的调用之前所进行的时间)。

@Expiration

```
@Expiration
```

此异步调用其中的一个参数，是调用所发生的时间（若为嵌套调用，则是第一个被处理的调用所发生的时间）。

@IntervalDuration

```
@IntervalDuration
```

指明此异步调用方法会循环调用，这个注解参数是循环调用之间的时间间隔。

27. 10. 用于JSF的注解

以下注解让使用JSF变得更容易。

@Converter

允许一个Seam组件作为JSF转换器（JSF converter）。被注解类必须是一个Seam组件，必须实现 `javax.faces.convert.Converter` 接口。

- `id` — JSF转换器id。默认为组件名。
- `forClass` — 若被指定，注册该组件为一个类型的默认转化器。

@Validator

允许一个Seam组件作为JSF验证器（JSF Validator）。被注解类必须是一个Seam组件，必须实现 `javax.faces.validator.Validator` 接口。

- `id` — JSF验证器id。默认为组件名。

27. 10. 1. 和 dataTable 一起使用的注解

以下注解让用Stateful Session Bean作为后台实现可点击列表更容易。它们在属性成员（`attributes`）上使用。

@DataModel

```
@DataModel("variableName")
```

将一个类型为 `List`、`Map`、`Set` 或 `Object[]` 的属性作为一个JSF `DataModel` 置入所属组件的scope（如果所属组件是 `STATELESS`则为EVENT scope）。当它是 `Map` 时，`DataModel` 的每一行是一个 `Map.Entry`。

- `value` — 转换器上下文参数名。默认是属性名。
- `scope` — 若`scope=ScopeType.PAGE` 显式指定，则 `DataModel` 将在 `PAGE` 上下文作用域内。

@DataModelSelection

```
@DataModelSelection
```

从JSF `DataModel`（这是基本集合或者映射值的一个元素）注入一个选定值。如果组件只定义一个 `@DataModel` 属性，那么 `DataModel` 所选中的值将被注入。否则，每个 `@DataModel` 组件名称必须被指定到每个 `@DataModelSelection` 的值属性中。

如果在所关联的 `@DataModel` 上声明了 `PAGE scope`，除了注入的 `DataModel Selection`，所关联的 `DataModel` 也会被注入。此时，如果标注有 `@DataModel` 注解的属性是getter方法，该Seam组件必须有同名的setter方法作为Business API的一部分。

- `value` — 对话上下文参数名。若在组件中有一个明确的 `@DataModel`，则不需要该属性。

@DataModelSelectionIndex

```
@DataModelSelectionIndex
```

暴露JSF `DataModel` 的选择索引作为一个组件属性（这是底层集合的行号，或者map key）。如果组件只定义一个 `@DataModel` 属性，那么 `DataModel` 所选中的值将被注入。否则，每个 `@DataModel` 组件名称必须被指定到每个 `@DataModelSelection` 的值属性中。

- `value` — 对话上下文参数名。若在组件中有一个明确的 `@DataModel` 则不需要该属性。

27.11. 用于数据绑定的元数据注解

这些元数据注解使得list以外的其它数据结构实现类似 `@DataModel` 和 `@DataModelSelection` 的功能成为可能。

@DataBinderClass

```
@DataBinderClass(DataModelBinder.class)
```

指定此注解是一个数据绑定注解。

@DataSelectorClass

```
@DataSelectorClass(DataModelSelector.class)
```

指定此注解是一个dataselection注解。

27.12. 用于打包（packing）的注解

这个注解提供一个为声明关于一系列要打包在一起的组件信息的机制。它能应用于任何Java包。

@Namespace

```
@Namespace(value="http://jboss.com/products/seam/example/seampay")
```

指定目前包中的组件关联到给定命名空间。为了简化应用配置，声明的命名空间可以在 `components.xml` 文件中作为XML命名空间来使用。

```
@Namespace(value="http://jboss.com/products/seam/core", prefix="org.jboss.seam.core")
```

指定一个关联到给定包的命名空间。另外，它指定一个组件名前缀，用于XML文件中出现的组件名。例如，一个和该命名空间关联的叫 `init` 的XML元素可被理解为实际引用一个叫做 `org.jboss.seam.core.init` 的组件。

27.13. 用于和Servlet容器集成的注解

这些注解允许你将你的Seam组件和Servlet容器集成。

@Filter

使一个用 `@Filter` 注解的Seam组件（它实现了 `javax.servlet.Filter` 接口）作为一个Servlet 过滤器（Filter）使用。它将会被Seam的主Filter执行。

```
@Filter(around={"seamComponent", "otherSeamComponent"})
```

指定此过滤器的在栈中的位置比指定过滤器更高。

```
@Filter(within={"seamComponent", "otherSeamComponent"})
```

指定此过滤器在栈中的位置比既定的过滤器更深。

第 28 章 内置Seam组件

这一章节描述了Seam的内置组件和配置属性。即使你的 `components.xml` 文件没有列出，内部组件仍然会被创立。但是如果你想覆盖掉默认的属性配置或者对某一类型的组件指定若干不同的组件，`components.xml` 就会派上用场了。

你可以简单地通过在独立的类里使用 `@Name` 注解指定内置组件的名字的方式来替代任何一个内置组件。

另外，即使所有的构建组件都使用全名，但他们大部分在缺省情况下都有简化了的别名。这些别名指定 `auto-create="true"`，所以，当你使用简化名来注入内置组件时，你不需要用 `create=true`。

28.1. 上下文注入组件

第一个内建的组件集存在完全支持不同的上下文对象的注入。例如，下列各项组件实例变量将会注入Seam会话上下文对象：

```
@In private Context sessionContext;
```

`org.jboss.seam.core.contexts`

用以获取各种Seam上下文对象，例如 `org.jboss.seam.core.contexts.sessionContext['user']`。

`org.jboss.seam.faces.facesContext`

`FacesContext` 的上下文管理组件（不是一个真的Seam上下文）

所有的这些组件通常都已被自动安装。

28.2. 工具组件

这些组件非常有用。

`org.jboss.seam.faces.facesMessages`

允许faces的成功消息跨越浏览器的重定向。

- `add(FacesMessage facesMessage)` — 增加一个faces消息，这将会在当前对话的下一个渲染响应阶段显示出来。
- `add(String messageTemplate)` — 增加一个faces消息，这个消息从那些可能包含EL表达式的消息模板来渲染。
- `add(Severity severity, String messageTemplate)` — 增加一个faces消息，这个消息从那些可能包含EL表达式的消息模板来渲染。
- `addFromResourceBundle(String key)` — 增加一个faces消息，这个消息从在Seam资源包中定义的消息模板来渲染。该消息模板可能包含EL表达式。
- `addFromResourceBundle(Severity severity, String key)` — 增加一个faces消息，这个消息从在Seam资

源包中定义的消息模板来渲染。该消息模板可能包含EL表达式。

- `clear()` — 清除所有的消息。

`org.jboss.seam.faces.redirect`

一个简便的可以实现带参数重定向的API。它对把搜索结果储存作书签特别有用。

- `redirect.viewId` — 用于重定向的JSF视图ID。
- `redirect.conversationPropagationEnabled` — 确定对话是否会跨越重定向。
- `redirect.parameters` — 一个含有请求参数键值的映射表，用来在重定向间传递。
- `execute()` — 立即执行重定向。
- `captureCurrentRequest()` — 存储视图ID和当前GET请求的请求参数（在对话上下文中），要想访问这些参数需要调用 `execute()`。

`org.jboss.seam.faces.httpError`

一个简便的发送HTTP错误的API。

`org.jboss.seam.core.events`

一个用于唤起事件API。这些事件可以通过被 `@Observer` 方法或者绑定 `components.xml` 的方法观察。

- `raiseEvent(String type)` — 唤起一个特殊类型的事件，并分发到所有的观察者。
- `raiseAsynchronousEvent(String type)` — 唤起一个事件，该事件可以被EJB3计时服务异步处理。
- `raiseTimedEvent(String type,)` — 计划一个事件，该事件可以被EJB3计时服务异步处理。
- `addListener(String type, String methodBinding)` — 为某一特定事件类型增加一个观察者。

`org.jboss.seam.core.interpolator`

一个用于替换字符串形式的JSF EL表达式的API。

- `interpolate(String template)` — 浏览JSF EL表达式形式 `#{...}` 的模板和取代他们的计算值。

`org.jboss.seam.core.expressions`

一个用于创造值和方法绑定的API。

- `createValueBinding(String expression)` — 创造值绑定对象
- `createMethodBinding(String expression)` — 创造方法绑定对象

`org.jboss.seam.core.pojoCache`

JBoss 缓存PojoCache 实例管理组件

- `pojoCache.cfgResourceName` — 配置文件的名字。默认到 `treecache.xml`。

所有这些组件通常都已被自动安装。

28.3. 组件的国际化 and 主题

这一组的组件，让通过使用Seam来创建国际化用户接口变得更容易。

`org.jboss.seam.core.locale`

Seam locale（本地化）。

`org.jboss.seam.international.timezone`

Seam时间区域。时间区域是会话范围。

`org.jboss.seam.core.resourceBundle`

Seam资源绑定。它在stateless范围。执行深度优先查询，查询键值在一系列的JAVA资源包中。

`org.jboss.seam.core.resourceLoader`

资源加载器提供对应用资源和资源包的访问。

- `resourceLoader.bundleNames` — 被搜索的Java资源包的包名，在Seam资源绑定被使用的情形下，默认到 `messages`。

`org.jboss.seam.international.localeSelector`

支持locale（本地化）选择，不仅在配置时，也可以在用户运行时。

- `select()` — 选择指定的locale。
- `localeSelector.locale` — 当前的 `java.util.Locale`。
- `localeSelector.localeString` — locale（本地化）字符串化的表达。
- `localeSelector.language` — 指定locale语言。
- `localeSelector.country` — 指定locale国家。
- `localeSelector.variant` — 指定locale变量。
- `localeSelector.supportedLocales` — 一个 `SelectItems` 的列表，表示被支持的locales。它们在 `jsf-config.xml` 中列出。
- `localeSelector.cookieEnabled` — 指定locale（本地）的选择结果必须通过浏览器来保留。

`org.jboss.seam.international.timezoneSelector`

支持时间区域选择，不仅在配置时，也可以在用户运行时。

- `select()` — 选择明确的locale（本地）。
- `timezoneSelector.timezone` — 当前的 `java.util.TimeZone`。
- `timezoneSelector.timeZoneId` — 时间区域的字符串化表达。
- `timezoneSelector.cookieEnabled` — 指明时间区域的选择结果必须通过浏览器来保留。

`org.jboss.seam.international.messages`

一个包含从消息模板中获取的国际化消息的映射表。消息模板定义在Seam资源包中。

`org.jboss.seam.theme.themeSelector`

支持主题选择，不仅在配置时，也可以在用户运行时。

- `select()` — 选择明确的主题。
- `theme.availableThemes` — 一系列的已定义的主题列表。
- `themeSelector.theme` — 已选择的主题。
- `themeSelector.themes` — 一个`SelectItems` 的列表，描述了已定义的主题。
- `themeSelector.cookieEnabled` — 指明主题的选择结果必须通过浏览器来保留。

`org.jboss.seam.theme.theme`

一个包含主题实体的映射表。

所有这些组件通常都已被自动安装。

28.4. 控制对话组件

这一组组件，允许通过应用程序或用户界面来控制对话。

`org.jboss.seam.core.conversation`

用于对当前Seam对话的属性进行应用控制的API。

- `getId()` — 返回当前对话的ID
- `isNested()` — 当前对话是否嵌套在其它对话中？
- `isLongRunning()` — 当前对话是否是长时间运行？
- `getId()` — 返回当前对话的ID
- `getParentId()` — 返回父对话的对话ID
- `getRootId()` — 返回根对话的对话ID
- `setTimeout(int timeout)` — 设置当前对话的失效时间
- `setViewId(String outcome)` — 在通过对话切换器、对话列表或breadcrumbs切换当前对话时，设置视图ID。
- `setDescription(String description)` — 设置当前对话的说明。该说明被用来显示在对话切换器、对话列表或breadcrumbs中。
- `redirect()` — 重定向到当前对话中最后一个明确定义的视图ID（登录后有用）。
- `leave()` — 退出对话的范围，但实际上没有结束这次对话。
- `begin()` — 开始长时间运行的对话（等同于`@Begin`）。

- `beginPageflow(String pageflowName)` — 协同页面流，开始一个长时间运行的对话（等同于 `@Begin(pageflow="...")`）。
- `end()` — 结束长时间运行的对话（等同于 `@End`）。
- `pop()` — 弹出对话堆栈，返回到父对话。
- `root()` — 返回到对话堆栈的根对话。
- `changeFlushMode(FlushModeType flushMode)` — 改变对话的刷新模式。

`org.jboss.seam.core.conversationList`
对话列表的管理组件。

`org.jboss.seam.core.conversationStack`
对话堆栈（breadcrumbs）的管理组件。

`org.jboss.seam.faces.switcher`
对话切换器。

所有这些组件通常都已被自动安装。

28.5. 与jBPM相关的组件

这些组件要和jBPM一起使用。

`org.jboss.seam.pageflow.pageflow`
用于控制Seam页面流的API。

- `IsInProcess()` 如果在进程中有一个页面流，则返回 `true`
- `getProcessInstance()` — 为当前页面流返回jBPM `ProcessInstance`（流程实例）
- `begin(String pageflowName)` — 在当前对话的上下文中开始一个页面流
- `reposition(String nodeName)` — 为当前页面流复位到一个指定的节点

`org.jboss.seam.bpm.actor`
用于与当前Session相关的jBPM的角色属性的应用控制的API

- `setId(String actorId)` — 设置当前用户的jBPM角色标识。
- `getGroupActorIds()` — 返回一个更多的当前用户群的jBPM角色标识部分Set。

`org.jboss.seam.bpm.transition`
用于当前任务的jBPM转换的应用控制的API。

- `setName(String transitionName)` — 当当前任务以 `@EndTask` 结束时，设置jBPM临时名字来使用。

`org.jboss.seam.bpm.businessProcess`

用于对话与业务处理之间联系的程序控制的API。

- `businessProcess.taskId` — 与当前对话相关的任务标识。
- `businessProcess.processId` — 与当前对话相关的过程标识。
- `businessProcess.hasCurrentTask()` — 是否是一个与当前对话相关的任务实例？
- `businessProcess.hasCurrentProcess()` — 是否是一个与当前对话相关的过程实例？
- `createProcess(String name)` — 创建一个命名过程定义的实例，并与当前对话相关。
- `startTask()` — 启动与当前对话相关的任务。
- `endTask(String transitionName)` — 结束与当前对话相关的任务。
- `resumeTask(Long id)` — 结合当前对话和指定标识的任务。
- `resumeProcess(Long id)` — 结合当前对话和指定标识的过程。
- `transition(String transitionName)` — 触发临时过程。

`org.jboss.seam.bpm.taskInstance`
jBPM `TaskInstance` 的管理组件。

`org.jboss.seam.bpm.processInstance`
jBPM `ProcessInstance` （流程实例）的管理组件。

`org.jboss.seam.bpm.jBPMContext`
事件范围 jBPMContext 的管理组件。

`org.jboss.seam.bpm.taskInstanceList`
jBPM任务列表的管理组件。

`org.jboss.seam.bpm.pooledTaskInstanceList`
jBPM池任务列表的管理组件。

`org.jboss.seam.bpm.taskInstanceListForType`
jBPM任务列表的管理组件。

`org.jboss.seam.bpm.pooledTask`
用于池任务安排的行为操作。

`org.jboss.seam.bpm.processInstanceFinder`
过程实例任务列表管理者。

`org.jboss.seam.bpm.processInstanceList`
过程实例任务列表。

无论 `org.jboss.seam.core.jBPM` 这个组件是否已安装，所有以上组件都会被自动安装。

28.6. 与安全相关的组件

这些组件都与Web层安全有关。

`org.jboss.seam.web.userPrincipal`

当前用户(本人)Principal的管理组件。

`org.jboss.seam.web.isUserInRole`

使JSF页面可以根据当前用户拥有的角色来选择渲染某一控制。 `<h:commandButton value="edit" rendered="#{isUserInRole['admin']}" />`。

28.7. 与JMS相关的组件

这些组件用于管理 `TopicPublishers` (主题发布者) 和 `QueueSenders` (队列发送者) (见下文)。

`org.jboss.seam.jms.queueSession`

JMS `QueueSession` 的管理组件。

`org.jboss.seam.jms.topicSession`

JMS `TopicSession` 的管理组件。

28.8. 与邮件相关的组件

这些组件与Seam的邮件支持一起被使用。

`org.jboss.seam.mail.mailSession`

JavaMail `Session` 的管理组件。 `Session` 或者可以从JNDI上下文中获取 (通过设定 `sessionJndiName` 属性)， 或者可以根据配置选项来创建。 如果是创建， `host` 必须被设定。

- `org.jboss.seam.mail.mailSession.host` — 使用SMTP服务的主机名
- `org.jboss.seam.mail.mailSession.port` — 使用SMTP服务的端口
- `org.jboss.seam.mail.mailSession.username` — 用于连接到SMTP服务的用户名。
- `org.jboss.seam.mail.mailSession.password` — 用于来连接到SMTP服务的密码。
- `org.jboss.seam.mail.mailSession.debug` — 开启JavaMail debug模式。(非常详细)
- `org.jboss.seam.mail.mailSession.ssl` — 开启到SMTP的SSL连接 (默认端口465)

`org.jboss.seam.mail.mailSession.tls` — 默认为true， 决定是否开启在邮件会话中的TLS支持

- `org.jboss.seam.mail.mailSession.sessionJndiName` — 在JNDI之内的 `javax.mail.Session` 的名字， 如果它被提供， 所有其它属性都将被忽略。

28.9. 基础组件

这些组件提供了关键的平台基础设施。你可以通过在 `components.xml` 文件中设定 `install="true"` 来安装一个默认情形下未被安装的组件。

`org.jboss.seam.core.init`

为Seam提供了初始化设置，总是自动被安装。

- `org.jboss.seam.core.init.jndiPattern` — JNDI模式用于寻找会话Bean。
- `org.jboss.seam.core.init.debug` — 打开 Seam 调试模式。
- `org.jboss.seam.core.init.clientSideConversations` — 如果设为 `true`，Seam将会在客户端而不是在 `HttpSession` 中保存对话上下文变量。
- `org.jboss.seam.core.init.userTransactionName` — 当寻找JTA `UserTransaction` 对象时，使用JNDI命名。

`org.jboss.seam.core.manager`

内在的组件用于Seam页和对话上下文管理，总是自动被安装。

- `org.jboss.seam.core.manager.conversationTimeout` — 对话上下文的超时时间，单位为微秒。
- `org.jboss.seam.core.manager.concurrentRequestTimeout` — 当一个线程试图在长时间的对话上下文中得到一个锁时的最大等待时间。
- `org.jboss.seam.core.manager.conversationIdParameter` — 用于传送对话ID的请求参数，默认为 `conversationId`。
- `org.jboss.seam.core.manager.conversationIsLongRunningParameter` — 用于传送是否是长时间对话的请求参数，默认是长时间对话：`conversationIsLongRunning`。

`org.jboss.seam.navigation.pages`

内部组件用于Seam工作平台管理，总是自动被安装。

- `org.jboss.seam.core.pages.noConversationViewId` — 当对话在服务端没有被找到时，将重定向到视图的ID（全局设定）。
- `org.jboss.seam.navigation.pages.loginViewId` — 当非授权用户试图访问一个受保护的页面时，将重定向到视图的ID（全局设定）。
- `org.jboss.seam.navigation.pages.httpPort` — http配置请求所使用的端口（全局设定）。
- `org.jboss.seam.navigation.pages.httpsPort` — https配置请求所使用的端口（全局设定）。
- `org.jboss.seam.navigation.pages.resources` — 一个用来搜索 `pages.xml` 样式资源的列表。默认为 `WEB-INF/pages.xml`。

`org.jboss.seam.bpm.jBPM`

引导 `jBPMConfiguration`（jBPM配置）。安装相应的类为 `org.jboss.seam.bpm.jBPM`。

- `org.jboss.seam.core.jBPM.processDefinitions` — 一系列的jPDL文件的资源列表，用于业务进程的协

调 (orchestration)。

- `org.jboss.seam.core.jBPM.pageflowDefinitions` — 一系列的jPDL 文件的资源列表，用于对话页面流的定义 (orchestration)。

`org.jboss.seam.core.conversationEntries`

内在的Session范围的组件，用于在各请求间记录活动的长时间对话。

`org.jboss.seam.faces.facesPage`

内在的页范围的组件，用于记录一页当中相关的对话上下文。

`org.jboss.seam.persistence.persistenceContexts`

内在组件，用以记录当前对话中的持久层上下文。

`org.jboss.seam.jms.queueConnection`

管理一个JMS `QueueConnection` (队列连接)。 当管理的 `QueueSender` (队列发送者) 被安装的时候，它也被安装。

- `org.jboss.seam.jms.queueConnection.queueConnectionFactoryJndiName` — JMS `QueueConnectionFactory` (队列连接工厂) 的JNDI名称，默认为 `UIL2ConnectionFactory`。

`org.jboss.seam.jms.topicConnection`

管理一个JMS `TopicConnection` (主题连接)。 当管理的 `TopicPublisher` 被安装的时候，它也被安装。

- `org.jboss.seam.jms.topicConnection.topicConnectionFactoryJndiName` — 一个JMS `TopicConnectionFactory` (主题连接工厂) 的JNDI名称，默认为 `UIL2ConnectionFactory`

`org.jboss.seam.persistence.persistenceProvider`

抽象层，对于JPA提供者的非标准化特性。

`org.jboss.seam.core.validators`

缓存Hibernate `Validator`实例 `ClassValidator`。

`org.jboss.seam.faces.validation`

被应用程序用来判断验证是否成功。

`org.jboss.seam.debug.introspector`

支持Seam调试页面。

`org.jboss.seam.debug.contexts`

支持Seam调试页面。

`org.jboss.seam.exception.exceptions`

用以处理异常的内部组件。

`org.jboss.seam.transaction.transaction`

通过一个与JTA兼容的接口，用以控制事务和对底层事务管理提供抽象的API。

`org.jboss.seam.faces.safeActions`

通过检查视图中的操作表达式 (action expression)，判断在进来的URL里的操作表达式是否安全

。

28.10. 杂项组件

这些组件无法归类。

`org.jboss.seam.async.dispatcher`

为异步方法调度无状态会话Bean。

`org.jboss.seam.core.image`

用以图像处理。

`org.jboss.seam.core.pojoCache`

PojoCache实例的管理组件。

`org.jboss.seam.core.uiComponent`

管理一个以Component ID为键值的UIComponents的映射表。

28.11. 特殊组件

当在Seam配置中指定了名字，某些特殊的Seam组件类可安装多次。例如，接下来配置在 `components.xml` 下的几行，安装和配置了两个Seam组件：

```
<component name="bookingDatabase"
  class="org.jboss.seam.persistence.ManagedPersistenceContext">
  <property name="persistenceUnitJndiName">java:/comp/emf/bookingPersistence</property>
</component>

<component name="userDatabase"
  class="org.jboss.seam.persistence.ManagedPersistenceContext">
  <property name="persistenceUnitJndiName">java:/comp/emf/userPersistence</property>
  </component>
```

这两个Seam组件是 `bookingDatabase` 和 `userDatabase`。

`<entityManager>`， `org.jboss.seam.persistence.ManagedPersistenceContext`

对话范围的受管 `EntityManager` 的管理组件。该 `EntityManager` 有一个扩展的持久层上下文。

- `<entityManager>.entityManagerFactory` — 一个值绑定表达式，该表达式的值为一个 `EntityManagerFactory` 的实例。

`<entityManager>.persistenceUnitJndiName` — 实体管理者工厂的JNDI名称，默认为 `java:/<managedPersistenceContext>`。

`<entityManagerFactory>`， `org.jboss.seam.persistence.EntityManagerFactory`

管理一个JPA `EntityManagerFactory`（实体管理者工厂）。当不在EJB3.0支持的环境下，特别有用。

- `entityManagerFactory.persistenceUnitName` — 持久单元的名称。

可参考API JavaDoc，得到更多的配置属性信息。

`<session>`， `org.jboss.seam.persistence.ManagedSession`

管理者组件，用于在对话范围内管理Hibernate Session。

- `<session>.sessionFactory` — 一个值绑定表达式，其值为一个 `SessionFactory` 的实例。

`<session>.sessionFactoryJndiName` — Session 工厂的JNDI名称，默认为 `java:/<managedSession>`。

`<sessionFactory>`， `org.jboss.seam.persistence.HibernateSessionFactory`

管理Hibernate SessionFactory（会话工厂）。

- `<sessionFactory>.cfgResourceName` — 配置文件的路径，默认为 `hibernate.cfg.xml`。

可参考API JavaDoc，得到更多的配置属性信息。

`<managedQueueSender>`， `org.jboss.seam.jms.ManagedQueueSender`

管理者组件，用于在事件范围内管理JMS QueueSender（队列发送者）。

- `<managedQueueSender>.queueJndiName` — JMS队列的JNDI名称。

`<managedTopicPublisher>`， `org.jboss.seam.jms.ManagedTopicPublisher`

管理者组件，用于在事件范围内管理JMS TopicPublisher（主题发布者）。

- `<managedTopicPublisher>.topicJndiName` — JMS主题的JNDI名称。

`<managedWorkingMemory>`， `org.jboss.seam.drools.ManagedWorkingMemory`

管理者组件，用于管理对话范围内的一个受管Drools WorkingMemory。

- `<managedWorkingMemory>.ruleBase` — 值表达式，其值为一个 `RuleBase` 的实例。

`<ruleBase>`， `org.jboss.seam.drools.RuleBase`

管理者组件，用于应用程序范围内的Drools RuleBase（基本规则）。需要注意的是，这不是真正用于生产使用，因为它不支持新规则的动态安装。

- `<ruleBase>.ruleFiles` — 一系列包含（Drools）规则的文件列表。

`<ruleBase>.dslFile` — (Drools)DSL定义。

`<entityHome>`， `org.jboss.seam.framework.EntityHome`

`<hibernateEntityHome>`， `org.jboss.seam.framework.HibernateEntityHome`

`<entityQuery>`， `org.jboss.seam.framework.EntityQuery`

`<hibernateEntityQuery>`， `org.jboss.seam.framework.HibernateEntityQuery`

第 29 章 Seam的JSF控件

Seam包括许多有利于使用Seam的JSF控件。它们用来补充内建的JSF控件，以及来自其他第三方库的控件。 我们推荐以JBoss Ajax4jsf、JBoss Richfaces和Apache MyFaces Trinidad标签库来使用Seam。我们不建议使用Tomahawk标签库来使用Seam。

29.1. 标签

为了使用这些tagsd，要在你的页面中定义“s”命名空间如下（只用于Facelets）：

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:s="http://jboss.com/products/seam/taglib">
```

ui的例子，示范了这其中许多标签的用法。

表 29.1. Seam的JSF控件参考

<div><s:button></div>	<div><div>描述</div><div>通过控制对话传播支持动作调用的按钮。不提交表单。</div><div>属性</div><div><ul style="list-style-type: none">value — 标签action — 指定动作监听者的一种方法绑定。view — 链接的JSF view id。fragment — 链接的fragment标识符。disabled — 该链接处于取消状态吗？propagation — 确定对话传播风格：begin、join、nest、none 或者 end。pageflow — 起始的页面流定义。（这只在 propagation="begin" 或者 propagation="join" 的时候才有用。）</div><div>用法</div><div><div><s:button id="cancel" value="Cancel" action="#{hotelBooking.cancel}"/></div></div></div>
<div><s:cache></div>	<div><div>描述</div><div>利用JBoss的Cache缓存渲染过的页面片断。 注意 <s:cache> 实际上使用由内建的 pojoCache 组件管理的JBoss Cache的实例。</div><div>属性</div></div>
<div>Seam Framework (2.0GA)277</div>	

	<ul style="list-style-type: none"> key — 是缓存渲染过的内容的键，经常是一个值表达式。例如，如果我们在缓存一个显示文档的页面片断，我们可以使用 <code>key="Document-#{document.id}"</code>。 enabled — 是一个值表达式，决定是否应该使用缓存。 region — 是一个要使用的JBoss Cache节点（不同的节点可以有不同的过期策略）。 <p>Usage 用法</p> <pre><s:cache key="entry-#{blogEntry.id}" region="pageFragments"> <div class="blogEntry"> <h3>#{blogEntry.title}</h3> <div> <s:formattedText value="#{blogEntry.body}" /> </div> <p> [Posted on&#160; <h:outputText value="#{blogEntry.date}"> <f:convertDateTime timezone="#{blog.timeZone}" locale="#{blog.locale}" type="both"/> </h:outputText>] </p> </div> </s:cache></pre>
<s:conversationId>	<p>描述</p> <p>将对话id添加到一个输出链接（或者类似的JSF控件）。只用于Facelets。</p> <p>属性</p> <p>无。</p>
<s:conversationPropagation>	<p>描述</p> <p>给一个命令链接或者按钮定制对话传播（或者类似的JSF控件）。只用于Facelets。</p> <p>属性</p> <ul style="list-style-type: none"> propagation — 确定对话传播风格；begin、join、nest、none 或者 end。 pageflow — 是一个起始页面流定义。（这只在 propagation="begin" 或者 propagation="join" 的时候才有用。） <p>用法</p> <pre><h:commandButton value="Apply" action="#{personHome.update}"> <s:conversationPropagation type="join" /> </h:commandButton></pre>

<p><s:convertDateTime></p>	<p>描述</p> <p>在Seam的timezone中执行日期或者时间对话。</p> <p>属性</p> <p>无。</p> <p>用法</p> <pre><h:outputText value="#{item.orderDate}"> <s:convertDateTime type="both" dateStyle="full"/> </h:outputText></pre>
<p><s:convertEntity></p>	<p>描述</p> <p>给当前的组件分配一个实体转换器。这主要对单选按钮和下拉控件有用。</p> <p>转换器使用任何具有 @Id 注解（简单的或者复合的）的受控实体。</p> <p>属性</p> <p>无。</p> <p>配置</p> <p>你必须通过 <s:convertEntity /> 使用 Seam管理的事务（请见第 8.2 节 “Seam管理的事务”）</p> <p>如果你的 受控持久化上下文 不是称作 entityManager，那你就需要在 components.xml 中设置它：</p> <pre><component name="org.jboss.seam.ui.EntityConverter"> <property name="entityManager">#{em}</property> </component></pre> <p>如果你想在实体转换器上使用不止一个实体管理器，你可以在 componets.xml 中为每一个实体管理器创建一份实体转换器。</p> <pre><component name="myEntityConverter" class="org.jboss.seam.ui.converter.EntityConverter"> <property name="entityManager">#{em}</property> </component></pre> <pre><h:selectOneMenu value="#{person.continent}"> <s:selectItems value="#{continents.resultList}" var="continent" label="#{continent.name}" /> <f:converter converterId="myEntityConverter" /> </h:selectOneMenu></pre> <p>用法</p> <pre><h:selectOneMenu value="#{person.continent}" required="true"> <s:selectItems value="#{continents.resultList}" var="continent"</pre>

	<pre> label="#{continent.name}" noSelectionLabel="Please Select..." /> <s:convertEntity /> </h:selectOneMenu> </pre>
<s:convertEnum>	<p>描述</p> <p>给当前的组件分配一个enum转换器。这主要对单选按钮和下拉控件有用。</p> <p>属性</p> <p>无。</p> <p>用法</p> <pre> <h:selectOneMenu value="#{person.honorific}"> <s:selectItems value="#{honorifics}" var="honorific" label="#{honorific.label}" noSelectionLabel="Please select" /> <s:convertEnum /> </h:selectOneMenu> </pre>
<s:decorate>	<p>描述</p> <p>在验证失败或者设置了 <code>required="true"</code> 时，“装饰”一个JSF输入域。</p> <p>属性</p> <ul style="list-style-type: none"> <code>template</code> — 用来装饰组件的Facelets模板。 <p><code>#{invalid}</code> 和 <code>#{required}</code> 可以在 <code>s:decorate</code> 内使用；如果你按要求设置了正被装饰的输入组件，<code>#{required}</code> 就取值为 <code>true</code>，并且如果发生校验错误，<code>#{invalid}</code> 取值为 <code>true</code>。</p> <p>用法</p> <pre> <s:decorate template="edit.xhtml"> <ui:define name="label">Country:</ui:define> <h:inputText value="#{location.country}" required="true"/> </s:decorate> </pre> <pre> <ui:composition xmlns="http://www.w3.org/1999/xhtml" xmlns:ui="http://java.sun.com/jsf/facelets" xmlns:h="http://java.sun.com/jsf/html" xmlns:f="http://java.sun.com/jsf/core" xmlns:s="http://jboss.com/products/seam/taglib"> <div> <s:label styleClass="#{invalid?'error':''}"> <ui:insert name="label"/> <s:span styleClass="required" rendered="#{required}">*</s:span> </s:label> </pre>

	<pre> <s:validateAll> <ui:insert/> </s:validateAll> <s:message styleClass="error"/> </div> </ui:composition> </pre>
<s:div>	<p>描述</p> <p>渲染一个HTML<div>。</p> <p>属性</p> <p>无。</p> <p>用法</p> <pre> <s:div rendered="#{selectedMember == null}"> Sorry, but this member does not exist. </s:div> </pre>
<s:enumItem>	<p>描述</p> <p>从一个enum值中创建一个 SelectItem。</p> <p>属性</p> <ul style="list-style-type: none"> enumValue — 是enum值的字符串表示法。 label — 在渲染 SelectItem 时要使用的标签。 <p>用法</p> <pre> <h:selectOneRadio id="radioList" layout="lineDirection" value="#{newPayment.paymentFrequency}"> <s:convertEnum /> <s:enumItem enumValue="ONCE" label="Only Once" /> <s:enumItem enumValue="EVERY_MINUTE" label="Every Minute" /> <s:enumItem enumValue="HOURLY" label="Every Hour" /> <s:enumItem enumValue="DAILY" label="Every Day" /> <s:enumItem enumValue="WEEKLY" label="Every Week" /> </h:selectOneRadio> </pre>
<s:fileUpload>	<p>描述</p> <p>渲染一个文件上传控件。这个控件必须通过在form中使用 multipart/form-data 的编码类型，例如：</p> <pre> <h:form enctype="multipart/form-data"> </pre>

对于多部分请求，也必须在 `web.xml` 中配置Seam Multipart Servlet过滤器：

```
<filter>
  <filter-name>Seam Filter</filter-name>
  <filter-class>org.jboss.seam.servlet.SeamFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>Seam Filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

配置

下列多部分请求的配置选项可以在 `components.xml` 中进行配置：

- `createTempFiles` — 如果这个选择设置为`true`，上载好的文件就流向一个临时文件，而不是流向内存。
- `maxRequestSize` — 允许上载文件的最大字节数。

下面是一个例子：

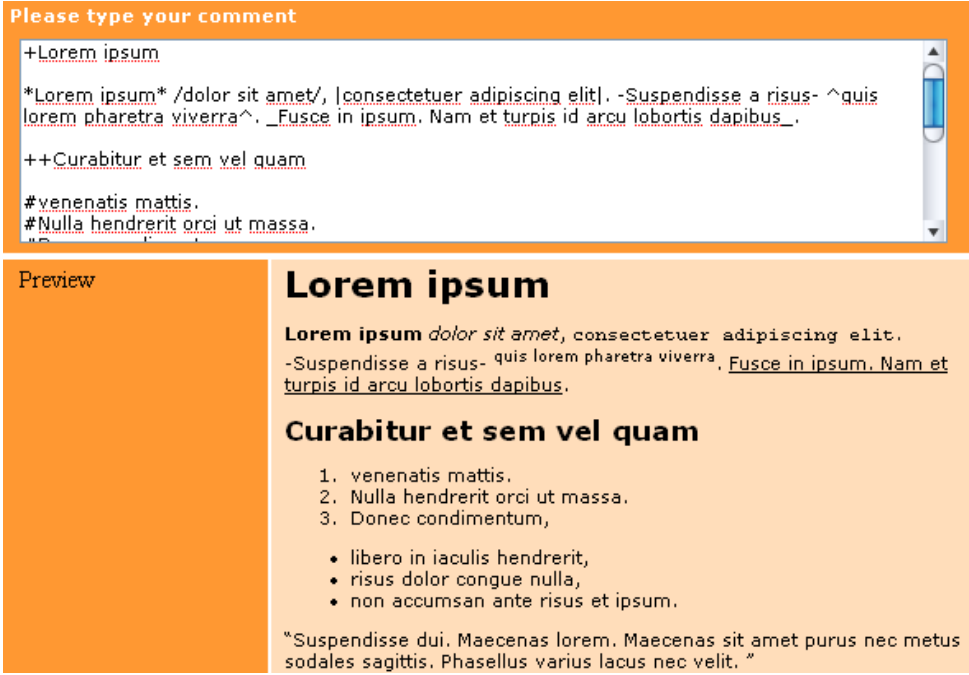
```
<component class="org.jboss.seam.web.MultipartFilter">
  <property name="createTempFiles">true</property>
  <property name="maxRequestSize">1000000</property>
</component>
```

属性

- `data` — 这个值绑定接收二进制文件数据。接收域应该声明为一个 `byte[]` 或者 `InputStream`（必要）。
- `contentType` — 这个值绑定接收文件的内容类型（可选）。
- `fileName` — 这个值绑定接收的文件名（可选）。
- `fileSize` — 这个值绑定接收的文件大小（可选）。
- `accept` — 可以接受的一个以逗号分隔的内容类型列表，可能浏览器不支持。例如 `"images/png, images/jpg"`、`"images/"`。
- `style` — 控件的样式，即CSS之类的
- `styleClass` — 控件的样式类

用法

```
<s:fileUpload id="picture" data="#{register.picture}"
  accept="image/png"
  contentType="#{register.pictureContentType}" />
```

<s:formattedText>	<p>描述</p> <p>输出 Seam Text，一种富文本标记，对于博客、Wiki和其他可能使用富文本的应用程序很有用。完整的用法请见Seam Text章节。</p> <p>属性</p> <ul style="list-style-type: none"> value — 一个指定要渲染的富文本标记的EL表达式。 <p>用法</p> <pre><s:formattedText value="#{blog.text}"/></pre> <p>实例</p> 
<s:validateFormattedText>	<p>描述</p> <p>检查提交的值是否合乎Seam Text</p> <p>属性</p> <p>无。</p>
<s:fragment>	<p>描述</p> <p>一个非渲染的组件，用于启用 / 取消它子组件的渲染。</p> <p>属性</p> <p>无。</p> <p>用法</p> <pre><s:fragment rendered="#{auction.highBidder ne null}"></pre>

	<div>Current bid: </s:fragment></div>
<s:graphicImage>	<p>描述</p> <p>一个允许在Seam Component中创建图片的扩展了的 <h:graphicImage>; 可以对图片进行进一步转换。</p> <p>支持 <h:graphicImage> 的所有属性, 以及:</p> <p>属性</p> <ul style="list-style-type: none"> • value — 要显示的图片。可以是一个路径 String (从classpath加载)、byte[]、java.io.File、java.io.InputStream 或者 java.net.URL。目前支持的图片格式有 image/png、image/jpeg 和 image/gif。 • fileName — 如果没有指定, 图片将有一个通用的文件名。如果你想要自己给文件命名, 就应该在这里指定。这个名称应该是唯一的。 <p>转换</p> <p>为了给图片应用一种转换, 你要嵌套一个指定要应用的转换的标签。Seam目前支持下面这些转换:</p> <p><s:transformImageSize> s:transformImageSize</p> <ul style="list-style-type: none"> • width — 图片的新宽度 • height — 图片的新高度 • maintainRatio — 如果为 true, 并且指定了其中 一个 width/height, 图片将利用不确定的、正被计算用来维持纵横比的尺寸调整大小。 • factor — 通过指定的比例缩放图片 <p><s:transformImageBlur></p> <ul style="list-style-type: none"> • radius — 利用指定的半径执行一个渐变模糊 <p><s:transformImageType></p> <ul style="list-style-type: none"> • contentType — 将图片的类型变成 image/jpeg 或者 image/png <p>创建你自己的转换很容易——创建一个 实现了 org.jboss.seam.ui.graphicImage.ImageTransform 的 UIComponent。在 applyTransform() 方法内部使用 image.getBufferedImage() 来获得原始图片, 用 image.setBufferedImage() 来设置你转换后的图片。转换以视图中指定的顺序进行。</p>

	<p>用法</p> <pre><s:graphicImage rendered="#{auction.image ne null}" value="#{auction.image.data}"> <s:transformImageSize width="200" maintainRatio="true"/> </s:graphicImage></pre>
<s:link>	<p>描述</p> <p>通过控制对话传播支持动作调用的链接。不提交表单。</p> <p>属性</p> <ul style="list-style-type: none">• value — 标签。• action — 指定动作监听者的一种方法绑定。• view — 链接的JSF view id。• fragment — 链接的fragment标识符。• disabled — 该链接处于取消状态吗？• propagation — 确定对话传播风格：begin、join、nest、none 或者 end。• pageflow — 起始的页面流定义。（这只在 propagation="begin" 或者 propagation="join"的时候才有用。） <p>用法</p> <pre><s:link id="register" view="/register.xhtml" value="Register New User"/></pre>
<s:message>	<p>描述</p> <p>”装饰“一个包含验证出错消息的JSF输入域。</p> <p>属性</p> <p>无。</p> <p>用法</p> <pre><f:facet name="afterInvalidField"> <s:span> &#160;Error:&#160; <s:message/> </s:span> </f:facet></pre>
<s:label>	<p>描述</p>

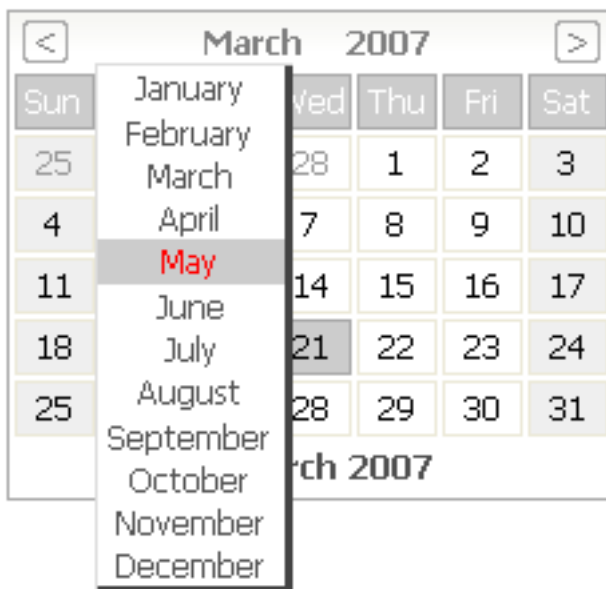
	<p>装饰一个包含标签的JSF输入域。这个标签放在HTML <code><label></code> 标签内部，且与最近的JSF输入组件相关联。它经常与 <code><s:decorate></code> 共用。</p> <p>Attributes 属性</p> <ul style="list-style-type: none"> • <code>style</code> — 控件的样式 • <code>styleClass</code> — 控件的样式类 <p>用法</p> <pre><s:label styleClass="label"> Country: </s:label> <h:inputText value="#{location.country}" required="true"/></pre>
<code><s:remote></code>	<p>描述</p> <p>用Seam Remoting生成所需要的JavaScript存根（stub）。</p> <p>属性</p> <ul style="list-style-type: none"> • <code>include</code> — 一个要为其生成Seam Remoting JavaScript 存根的以逗号分隔的组件名列表（或者合法的全类名）。更多详情请见 第 21 章 Remoting。 <p>用法</p> <pre><s:remote include="customerAction,accountAction,com.acme.MyBean"/></pre>
<code><s:selectDate></code>	<p>描述</p> <p>已被废弃。用 <code><rich:calendar /></code> 代替。</p> <p>显示一个动态的日期选择器组件，它给指定的输入域选择日期。<code>selectDate</code> 元素的主体应该包含HTML元素，例如文本或者图片，提示用户点击以显示日期选择器。日期选择器 必须 利用CSS定义样式。可以在Seam booking demo中找到CSS范例文件 <code>date.css</code>，或者可以利用seam-gen生成。用来控制日期选择器外观的CSS样式也在下面做了说明。</p> <p>属性</p> <ul style="list-style-type: none"> • <code>for</code> — 日期选择器要把选择的日期插入到其中的输入域的id。 • <code>dateFormat</code> — 日期格式的字符串。这应该与输入域的日期格式匹配。 • <code>startYear</code> — 弹出年选择器范围将从这一年开始。 • <code>endYear</code> — 弹出年选择器范围将从这一年终止。

- `firstDayOfWeek` — 控制哪一天是一周的第一天（0 = Sunday, 6 = Saturday）。如果没有设置这个属性，那么一周的第一天默认将基于用户所在的区域。

用法

```
<div class="row">
  <h:outputLabel for="dob">Date of birth<em>*</em></h:outputLabel>
  <h:inputText id="dob" value="#{user.dob}" required="true">
    <s:convertDateTime pattern="MM/dd/yyyy"/>
  </h:inputText>
  <s:selectDate for="dob" startYear="1910" endYear="2007">
    
  </s:selectDate>
  <div class="validationError"><h:message for="dob"/></div>
</div>
```

范例

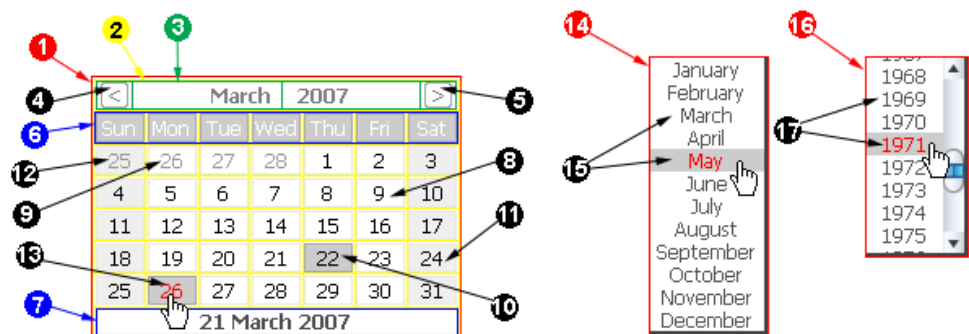


CSS样式

以下列表描述了用来控制selectDate控件样式的CSS类名。

- `seam-date` — 这个类用于包含弹出日历的外层 `div`。（1）它还用于控制日历内部布局的 `table`。（2）
- `seam-date-header` — 这个类用于日历头表行（`tr`）和头表单元（`td`）。（3）
- `seam-date-header-prevMonth` — 用于“前一个月”表单元（`td`），点击它时，导致日历显示当前显示的前一个月。（4）
- `seam-date-header-nextMonth` — 这个类用于“下一个月”表单元（`td`），点击它时，导致日历显示当前显示的下一个月。（5）
- `seam-date-headerDays` — 这个类用于日历header行（`tr`），它包含了周日期的名称。（6）

- seam-date-footer — 这个类用于日历的footer行（tr），它显示当前日期。（7）
- seam-date-inMonth — 这个类用于包含了当前显示月份中的一个日期的表单元(td)元素。（8）
- seam-date-outMonth — 这个类用于包含了当前显示月份之外的一个日期的表单元（td）元素。（9）
- seam-date-selected — 这个类用于包含当前选择日期的表单元td元素。（10）
- seam-date-dayOff-inMonth — 这个类用于包含当前选择月份之内的”休假“日（例如周末，周六和周日）的表单元元素。（11）
- seam-date-dayOff-outMonth — 这个类用于包含当前选择的月份之外的休假日（例如周末，周六和周日）的表单元元素。（12）
- seam-date-hover — 这个类用于鼠标经过的表单元（td）元素。（13）
- seam-date-monthNames — 这个类用于包含弹出月份选择器的 div 控件。（14）
- seam-date-monthNameLink — 这个类用于包含弹出月份名称的anchor（a）控件。（15）
- seam-date-years — 这个类用于包含弹出年选择器的div控件。（16）
- seam-date-yearLink — 这个类用于包含弹出年份的anchor（a）控件。（17）



<s:selectItems>

描述

从一个List、Set、DataModel或者Array中创建一个 List<SelectItem> 。

属性

- value — 一个EL表达式，指定支持 List<SelectItem> 的数据；
- var — 定义迭代期间保存当前对象的本地变量的名称。
- label — 渲染 SelectItem 时要使用的标签。可以参考 var 变量。

	<ul style="list-style-type: none"> disabled — 如果为true, SelectItem 将被取消渲染。可以参考 var 变量。 noSelectionLabel — 指定（可选）标签放在列表的顶部（如果也指定 required="true", 那么选择这个值将导致验证出错）。 hideNoSelectionLabel — 如果为true, 选择一个值时, noSelectionLabel 将被隐藏。 <p>用法</p> <pre><h:selectOneMenu value="#{person.age}" converter="#{converters.ageConverter}" <s:selectItems value="#{ages}" var="age" label="#{age}" /> </h:selectOneMenu></pre>
<s:span>	<p>描述</p> <p>渲染一个HTML的。</p> <p>属性</p> <p>无。</p> <p>用法</p> <pre><s:span styleClass="required" rendered="#{required}">*</s:span></pre>
<s:taskId>	<p>描述</p> <p>当任何可以通过 #{task} 使用的时候, 将任何id添加到一个输出链接（或者类似的JSF控件）。只用于Facelets。</p> <p>属性</p> <p>无。</p>
<s:validate>	<p>描述</p> <p>一个非可视化的控件, 利用 Hibernate Validator 对绑定属性验证一个JSF输入域。</p> <p>属性</p> <p>无。</p> <p>用法</p> <pre><h:inputText id="userName" required="true" value="#{customer.userName}" <s:validate /> </h:inputText> <h:message for="userName" styleClass="error" /></pre>

<pre> <s:validateAll> s:validateAll </pre>	<p>描述</p> <p>一个非可视化的控件，利用 Hibernate Validator 对它们绑定的属性验证所有的子JSF输入域。</p> <p>属性</p> <p>无。</p> <p>用法</p> <pre> <s:validateAll> <div class="entry"> <h:outputLabel for="username">Username:</h:outputLabel> <h:inputText id="username" value="#{user.username}" required="true"/> <h:message for="username" styleClass="error" /> </div> <div class="entry"> <h:outputLabel for="password">Password:</h:outputLabel> <h:inputSecret id="password" value="#{user.password}" required="true"/> <h:message for="password" styleClass="error" /> </div> <div class="entry"> <h:outputLabel for="verify">Verify Password:</h:outputLabel> <h:inputSecret id="verify" value="#{register.verify}" required="true"/> <h:message for="verify" styleClass="error" /> </div> </s:validateAll> </pre>
--	--

29.2. 注解

为了允许你用Seam组件作为JSF转换器和验证器，Seam也提供注解：

@Converter

```

@Name("fooConverter")
@BypassInterceptors
@Converter
public class FooConverter implements Converter {

    @In EntityManager entityManager;

    @Transactional
    public Object getAsObject(FacesContext context, UIComponent cmp, String value) {
        EntityManager entityManager = (EntityManager) Component.getInstance("entityManager");
        entityManager.joinTransaction();
        // Do the conversion
    }

    public String getAsString(FacesContext context, UIComponent cmp, Object value) {
        // Do the conversion
    }
}

```

```
}
```

将Seam组件注册为一个JSF转换器。这里展示的是，在将值转换回它的对象表示法的时候，能够访问JTA事务中的JPA EntityManager的转换器。

@Validator

```
@Name("barValidator")
@BypassInterceptors
@Validator
public class BarValidator implements Validator {

    @In FooController fooController;

    public void validate(FacesContext context, UIComponent cmp, Object value)
        throws ValidatorException {
        FooController fooController = (FooController) Component.getInstance("fooController");
        return fooController.validate(value);
    }
}
```

将Seam组件注册为一个JSF验证器。这里展示的是，一个注入另一个Seam组件的验证器；注入的组件用来验证值。

第 30 章 表达式语言增强

Seam提供了一个可扩展标准统一表达语言（EL）被称为JBoss EL。JBoss EL提供了许多增强表达和更强的EL语言方面的加强。

30.1. 参数方法绑定

标准EL默认任何参数方法都是Java代码方式。这就意味着参数方法不能被JSF方式来使用。Seam提供了一个加强式的EL允许参数包含在表达式中来表达自己。Seam提供的 `any` 表达方法，包括任何JSF的绑定。例如：

```
<h:commandButton action="#{hotelBooking.bookHotel(hotel)}" value="Book Hotel"/>
```

30.1.1. 用法

参数用括号表示，用逗号分隔：

```
<h:commandButton action="#{hotelBooking.bookHotel(hotel, user)}" value="Book Hotel"/>
```

参数 `hotel` 和 `user` 会被作为值表达式计算，并传递到组件的 `bookHotel()` 方法中。这让你有了 `@In` 的替代方法。

任何值表达式都可以用作参数：

```
<h:commandButton action="#{hotelBooking.bookHotel(hotel.id, user.username)}"
value="Book Hotel"/>
```

注意： 你不能通过对象的方式来表述！所有的都要是直接的名称，例如，`hotel.id` 和 `user.username`。如果你查看下先前的代码，你将可以看到按钮命令包括这些名称。当你触发按钮事件的时候，这些名称的表述将被提交到服务器端，Seam将在action调用前来查询这些提交过来的名称（在任何一个应用程序的上下文中）。如果这些名称表述无法在那时刻得到（因为 `hotel` 和 `user` 变量不能在任何应用程序的上下文中找到），action方法将返回一个 `null` 表述！

你甚至可以传递包含有单引号或双引号的字符串：

```
<h:commandLink action="#{printer.println('Hello world!')}" value="Hello"/>
```

```
<h:commandLink action="#{printer.println('Hello again')}" value="Hello"/>
```

你可能还想把这种标记用在你所有的action方法上，不管他们是不是需要传递参数。这也能提高程序的可读性，明确表明这是方法表达式而非值表达式：

```
<s:link value="Cancel" action="#{hotelBooking.cancel()}" />
```

30.1.2. 限制

请注意以下的限制：

30.1.2.1. 与JSP 2.1不兼容

这一扩展目前不兼容JSP 2.1。因此如果你希望将这一扩展用于JSF 1.2，你必须使用Facelets。目前这一扩展在JSP 2.0下运作正常。

30.1.2.2. 从Java代码中调用 `MethodExpression`

一般而言，`MethodExpression` 或 `MethodBinding` 被创建后，参数是通过JSF传递的。如果是方法绑定，JSF假设不会传递参数。使用这个扩展的话，我们直到表达式被计算前，都无法知道参数类型。这就带来两个小的副作用：

- 在Java代码中调用 `MethodExpression`，你传递的参数可能被忽略。定义在表达式中的参数被优先处理。
- 通常，在任何时间调用 `methodExpression.getMethodInfo().getParamTypes()` 都是安全的。但对于含有参数的表达式，你必须先调用 `MethodExpression`，才能调用 `getParamTypes()`。

以上两种情况都十分罕见，而且只发生在你在Java代码中手动调用 `MethodExpression` 的时候。

30.2. 参数值绑定

标准EL仅允许访问属性这个成了JavaBean命名的约定。例如，表达式 `#{person.name}` 要求 `getName()` 表示当前的值。然而许多的对象却没有恰当的命名属性或者参数。这些值能被使用的方法来调用，这个就和参数方法绑定很相似。例如，下面表达式就是使用 `length()` 方法返回一个字符串的大小。

```
#{person.name.length()}
```

你可以通过一个相类似的方法来访问一个集合的大小。

```
#{searchResults.size()}
```

在一般的 form `#{obj.property}` 表达式中可以等同于表达式 `#{obj.getProperty()}`。

当然参数的方式也是允许的，只要按照同样限制的方法来绑定是一样的。下面这个例子就是通过文字字符串调用 `productsByColorMethod` 的。

```
#{controller.productsByColor('blue')}
```

30.3. 映射

JBoss EL支持有限的映射语法。这里要强调的一点就是这个语法不能被Facelets或者JSP解析也不能在xhtml或者JSP中使用。我们期待在下一个JBoss EL版本中projection语法中将得到改变。

映射表达式是通过多值（list，set等等）表达式来体现一个子表达式的。例如，表达式：

```
#{company.departments}
```

可以返回一个department的list集合。如果你仅需要department name的集合，那么你的选项就要通过遍历迭代的方式来获取值。JBoss EL支持这样一个映射表达式。

```
# {company.departments. {d|d.name}}
```

子表达式被大括号 {} 包围起来。在这个例子中，表达式 `d.name` 是遍历得到的每个department值，使用 `d` 作为department对象的别名。这个表达式的结果是一个String值的集合。

任何一个有效表达式的都可以在表达式中使用。假如你想知道在公司所有department（部门）个数的时候，下面它就可以很好验证书写正确性。

```
# {company.departments. {d|d.size()}}
```

映射可以被嵌套。下面的表达式返回的就是在每个department每个雇员的名字。

```
# {company.departments. {d|d.employees. {emp|emp.lastName}}}
```

嵌套映射有时可能有点棘手，下面的表达式就是返回一个所有department所有employee的集合。

```
# {company.departments. {d|d.employees}}
```

但是，它实际上要返回单个department所有employee（雇员）的集合。为了同时具有这个值，它还需要一个稍微长点的表达式。

```
# {company.departments. {d|d.employees. {e|e}}}
```

第 31 章 测试Seam应用程序

大部分的Seam应用程序至少需要两种类型的自动测试：单元测试（unit test）是隔离测试特定的Seam组件，和脚本化的集成测试（integration test）是综合地测试应用中所有的Java层面（即除了表现层之外的所有内容）。

两种类型的测试都很容易编写。

31.1. Seam组件的单元测试

所有的Seam组件都是简单Java对象（POJO）。如果你想简化单元测试，那么这是个极好的开端，而且Seam将其重点放在组件间交互的双向注入和上下文对象的访问上，这使得Seam的组件在脱离其正常运行环境时，也可以很容易地被测试。

思考如下的Seam组件：

```
@Stateless
@Scope(EVENT)
@Name("register")
public class RegisterAction implements Register
{
    private User user;
    private EntityManager em;

    @In
    public void setUser(User user) {
        this.user = user;
    }

    @PersistenceContext
    public void setBookingDatabase(EntityManager em) {
        this.em = em;
    }

    public String register()
    {
        List existing = em.createQuery("select username from User where username=:username")
            .setParameter("username", user.getUsername())
            .getResultList();
        if (existing.size()==0)
        {
            em.persist(user);
            return "success";
        }
        else
        {
            return null;
        }
    }
}
```

测试上述组件的TestNG测试如下：

```
public class RegisterActionTest
{
    @Test
```

```

public testRegisterAction()
{
    EntityManager em = getEntityManagerFactory().createEntityManager();
    em.getTransaction().begin();

    User gavin = new User();
    gavin.setName("Gavin King");
    gavin.setUserName("lovthafew");
    gavin.setPassword("secret");

    RegisterAction action = new RegisterAction();
    action.setUser(gavin);
    action.setBookingDatabase(em);

    assert "success".equals( action.register() );

    em.getTransaction().commit();
    em.close();
}

private EntityManagerFactory emf;

public EntityManagerFactory getEntityManagerFactory()
{
    return emf;
}

@BeforeClass
public void init()
{
    emf = Persistence.createEntityManagerFactory("myResourceLocalEntityManager");
}

@AfterClass
public void destroy()
{
    emf.close();
}
}

```

Java Persistence API可以与Java SE和Java EE一起使用 — 当上述组件在应用服务器（Java EE）中使用时，由容器来负责事务管理；然而，在单元测试（Java SE）里，事务必须显式地使用本地资源实体管理器来进行管理。这要求在 `persistence.xml` 进行配置。

Seam的组件通常不直接依赖于容器的基础设施，因此大部分的单元测试跟上述一样容易。

31.2. Seam组件的集成测试

相比单元测试，集成测试有稍许的难度。在这里，我们不能再对容器的基础设施视而不见，相反这也正是需要测试的一部分！同时，我们也不想强制地将我们的应用程序部署到应用服务器上来运行这些自动化测试。那么为了能全面地测试我们的应用程序，且在不损失太多性能的条件下，我们需要在测试环境中再造必要的容器基础设施。

因此Seam采取的方法是在一个经修剪过的容器环境中（Seam，以及嵌入式的JBoss容器，需要JDK 1.5并且不支持JDK 1.6）编写测试用例来测试你的组件。

```
public class RegisterTest extends SeamTest
```



```

{

@Test
public void testRegisterComponent() throws Exception
{

    new ComponentTest() {

        protected void testComponents() throws Exception
        {
            setValue("#{user.username}", "lovthafew");
            setValue("#{user.name}", "Gavin King");
            setValue("#{user.password}", "secret");
            assert invokeMethod("#{register.register}").equals("success");
            assert getValue("#{user.username}").equals("lovthafew");
            assert getValue("#{user.name}").equals("Gavin King");
            assert getValue("#{user.password}").equals("secret");
        }

    }.run();

}

...

}

```

31.2.1. 在集成测试中使用Mock对象

有时候，Seam的一些组件所依赖的资源在集成测试环境中没有，那么我们需要替换这些组件。例如，假设现在有一些Seam组件，他们是对支付处理系统的facade，示例如下

```

@Name("paymentProcessor")
public class PaymentProcessor {
    public boolean processPayment(Payment payment) { .... }
}

```

为了能够集成测试，我们可以对此组件Mock如下：

```

@Name("paymentProcessor")
@Install(precedence=MOCK)
public class MockPaymentProcessor extends PaymentProcessor {
    public void processPayment(Payment payment) {
        return true;
    }
}

```

因为 MOCK 的优先级比应用组件的默认优先级要高，所以Seam将优先装配在classpath中的Mock对象。当部署到生产环境中的时候，那些Mock对象将不复存在，因此真正的组件将被装配进来。

31.3. 集成测试Seam应用程序中的用户交互

在测试中，一个更难的问题是模拟用户交互。因此第三个问题是：我们应该在那里放置断言（assertion）。一些测试框架通过在Web浏览器中重现用户交互来测试整个应用程序，这些测试有其适用之处，但他们并不适合在开发时使用。

在一个模拟的JSF环境中，SeamTest 可以让你编写 脚本化（scripted） 测试。这些脚本化测试的用处是为了重现视图和Seam组件之间的交互，换句话说，你要假装你是JSF的实现！

这种方法可以测试除了视图以外的所有事物。

让我们来看一个JSP视图，此视图对应的组件就是上述单元测试过那个组件：

```
<html>
<head>
  <title>Register New User</title>
</head>
<body>
  <f:view>
    <h:form>
      <table border="0">
        <tr>
          <td>Username</td>
          <td><h:inputText value="#{user.username}"/></td>
        </tr>
        <tr>
          <td>Real Name</td>
          <td><h:inputText value="#{user.name}"/></td>
        </tr>
        <tr>
          <td>Password</td>
          <td><h:inputSecret value="#{user.password}"/></td>
        </tr>
      </table>
      <h:messages/>
      <h:commandButton type="submit" value="Register" action="#{register.register}"/>
    </h:form>
  </f:view>
</body>
</html>
```

我们想测试一下应用程序的注册功能（即当用户点击注册按钮要发生的事情）。我们可以在TestNG的自动测试中重现JSF的请求生命周期：

```
public class RegisterTest extends SeamTest
{
    @Test
    public void testRegister() throws Exception
    {
        new FacesRequest() {

            @Override
            protected void processValidations() throws Exception
            {
                validateValue("#{user.username}", "lovthafew");
                validateValue("#{user.name}", "Gavin King");
                validateValue("#{user.password}", "secret");
                assert !isValidationFailure();
            }

            @Override
            protected void updateModelValues() throws Exception
            {
                setValue("#{user.username}", "lovthafew");
                setValue("#{user.name}", "Gavin King");
                setValue("#{user.password}", "secret");
            }
        }
    }
}
```

```
@Override
protected void invokeApplication()
{
    assert invokeMethod("#{register.register}").equals("success");
}

@Override
protected void renderResponse()
{
    assert getValue("#{user.username}").equals("lovthafew");
    assert getValue("#{user.name}").equals("Gavin King");
    assert getValue("#{user.password}").equals("secret");
}

}.run();

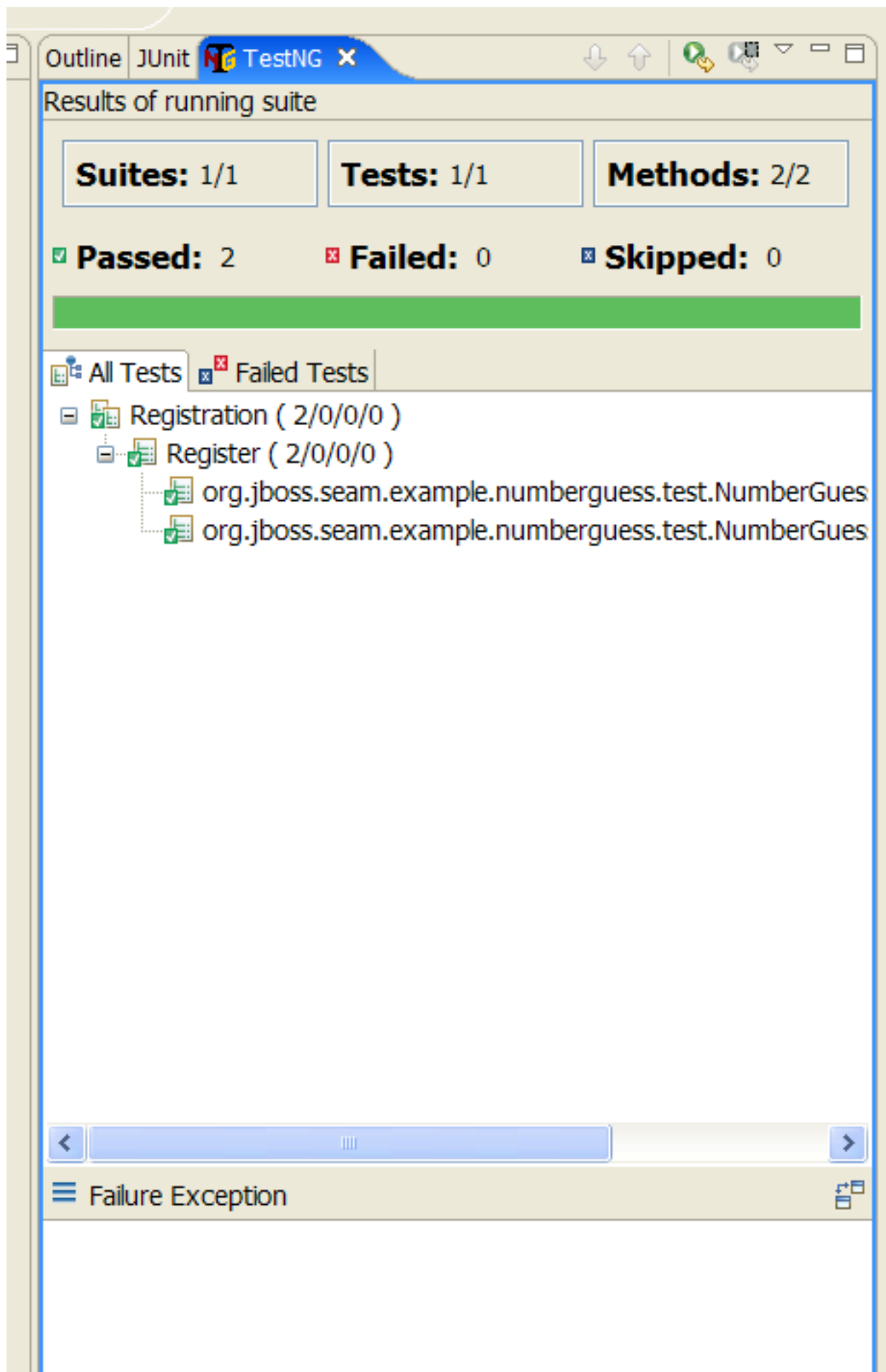
}

...

}
```

值得注意的是：我们继承了 `SeamTest`，其为我们的组件提供了一个Seam环境，并且我们还需要写一个继承了 `SeamTest.FacesRequest` 的匿名类，此匿名类模拟JSF的请求生命周期（还有一个 `SeamTest.NonFacesRequest` 是测试GET请求的）。为了模拟JSF对我们组件的调用，我们已经完成了JSF不同阶段的方法实现，接着我们还加入了各种断言。

你可以在Seam的更复杂的示例应用程序中找到大量关于集成测试的用法，还有在Ant或者Eclipse的TestNG插件下运行这些测试的使用说明。



31.3.1. 利用Mock数据进行集成测试

如果你需要在每个测试之前在数据库中插入或删除数据，你可以使用DBUnit进行Seam的集成测试。要做到这一点，要继承DBUnitSeamTest而不是SeamTest。

你需要提供数据集给DBUnit：

```
<dataset>

<ARTIST
  id="1"
  dtype="Band"
  name="Pink Floyd" />

<DISC
  id="1"
  name="Dark Side of the Moon"
  artist_id="1" />

</dataset>
```

并通过覆盖 `prepareDBUnitOperations()` 来告诉Seam：

```
protected void prepareDBUnitOperations() {
    beforeTestOperations.add(
        new DataSetOperation("my/datasets/BaseData.xml")
    );
}
```

如果没有指定其它的操作作为构造器参数 `DataSetOperation` 的操作默认是 `DatabaseOperation.CLEAN_INSERT`。在调用每个 `@Test` 方法前，上述的示例会先清除 `BaseData.xml` 中定义的所有表，然后插入 `BaseData.xml` 中定义的所有数据行。

如果你需要在一个测试方法执行后进行额外的清除工作，添加操作到 `afterTestOperations` 列表中。

你需要通过设置一个名为 `datasourceJndiName` 的TestNG测试参数来告诉DBUnit你正在使用的数据源：

```
<parameter name="datasourceJndiName" value="java:/seamdiscsDatasource"/>
```

31.3.2. Seam Mail集成测试

警告！这个功能仍在开发当中。

集成测试Seam Mail相当的简单：

```
public class MailTest extends SeamTest {

    @Test
    public void testSimpleMessage() throws Exception {

        new FacesRequest() {

            @Override
            protected void updateModelValues() throws Exception {
                setValue("#{person.firstname}", "Pete");
                setValue("#{person.lastname}", "Muir");
            }
        };
    }
}
```

```
        setValue("#{person.address}", "test@example.com");
    }

    @Override
    protected void invokeApplication() throws Exception {
        MimeMessage renderedMessage = getRenderedMailMessage("/simple.xhtml");
        assert renderedMessage.getAllRecipients().length == 1;
        InternetAddress to = (InternetAddress) renderedMessage.getAllRecipients()[0];
        assert to.getAddress().equals("test@example.com");
    }

    }.run();
}
}
```

我们与往常一样创建一个新的 `FacesRequest`。在 `invokeApplication` 里我们通过传递消息的 `viewId` 去渲染 `getRenderedMailMessage(viewId)` 的消息。这个方法返回已经渲染完成的消息，你可以继续进行你的测试。你当然可以同时使用任何一项标准JSF的生命周期的方法。

还有就是不支持渲染标准JSF组件，所以你不能方便地测试邮件消息的内容主体。

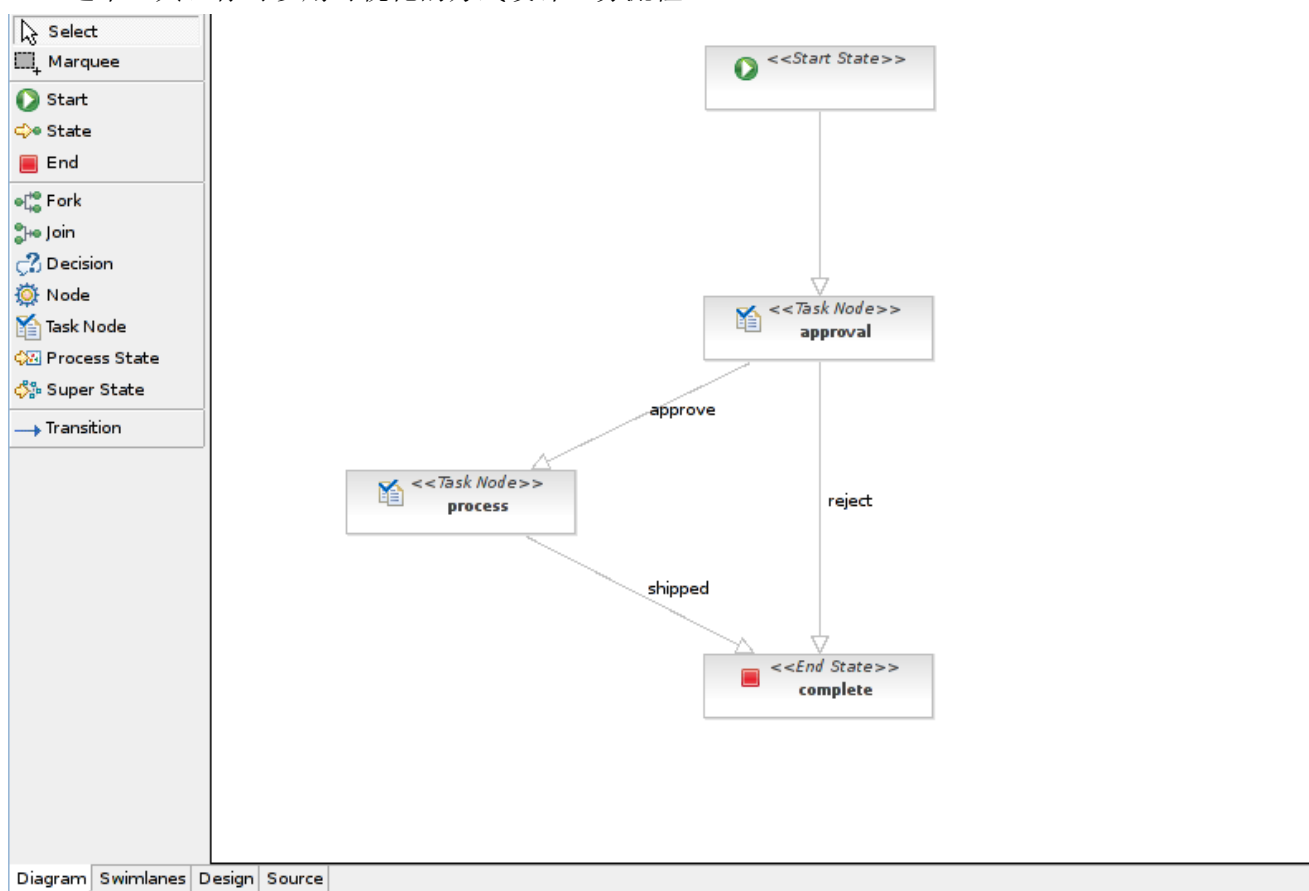
第 32 章 Seam工具

32.1. jBPM设计器和查看器

jBPM设计器和查看器会提供一种非常漂亮的方法让你设计并查看业务流程和页面流。这个便利的工具是JBoss Eclipse IDE集成开发环境的一部分，在jBPM的文档（<http://docs.jboss.com/jbpm/v3/gpd/>）中，你可以找到更详细的信息。

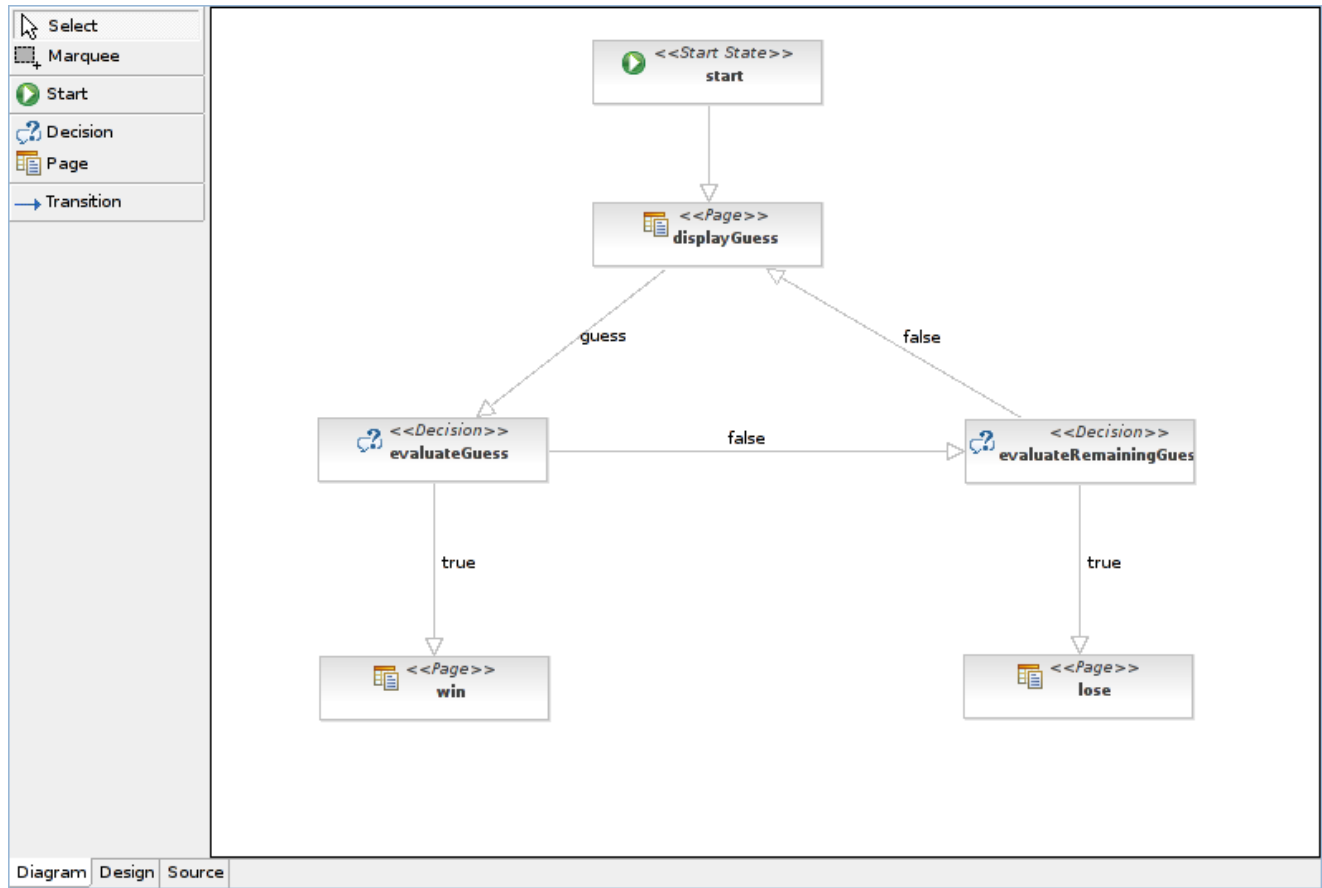
32.1.1. 业务流程设计器

这个工具让你可以用可视化的方式设计业务流程。



32.1.2. Pageflow查看器

这个工具可以帮助你设计出可以扩展的页面流（Pageflow），并且让你设计出页面流的图形表现形式，如此你便可以和其他人分享并比较如何设计页面流的思想了。



第 33 章 依赖包

33.1. 项目依赖包

在这一章节主要是列出了Seam在编译时和运行时需要用到的依赖包。列表中类型为 ear 的包应该放在应用程序ear文件的/lib目录中。 类型为 war 的应该放在应用程序war文件的 /WEB-INF/lib 目录里。 依赖包的Scope是all、runtime或provided（JBoss AS 4.2）。

最新的版本信息没有包含在这个文档中，可以在 /build/root.pom.xml Maven POM 文件找到。

33.1.1. Core

表 33.1.

名称	范围	类型	说明
commons-codec.jar	runtime	ear	Seam Security使用Digest认证时所必须的。
jboss-seam.jar	all	ear	Seam核心包，始终需要
jboss-seam-debug.jar	runtime	war	在项目开发阶段，开启Seam的调试功能时需要。
jboss-seam-ioc.jar	runtime	war	与Spring结合使用时需要。
jboss-seam-pdf.jar	runtime	war	当使用Seam的PDF功能时需要。
jboss-seam-remoting.jar	runtime	war	当使用Seam Remoting时需要。
jboss-seam-ui.jar	runtime	war	当使用Seam JSF控件时需要。
jsf-api.jar	provided		JSF API
jsf-impl.jar	provided		JSF参考实现Reference Implementation
jsf-facelets.jar	runtime	war	Facelets
urlrewrite.jar	war	no	URL Rewrite库

名称	范围	类型	说明
jcaptcha-all.jar	ear	no	用以支持Captcha。
quartz.jar	ear	yes	当你想使用Seam中Quartz的异步特性这个是必须的。

33.1.2. Ajax4JSF / RichFaces

表 33.2. RichFaces依赖包

名称	范围	类型	说明
richfaces-api.jar	all	ear	使用RichFaces时需要。为你的应用程序提供API类，比如：创建一棵树。
richfaces-impl.jar	runtime	war	使用RichFaces时需要。
richfaces-ui.jar	runtime	war	使用RichFaces时需要。提供所有的UI组件

33.1.3. Seam Mail

表 33.3. Seam Mail依赖包

名称	范围	类型	说明
activation.jar	runtime	ear	支持附件需要的包。
mail.jar	runtime	ear	支持发送邮件需要的包。
mail-ra.jar	compile only		支持接收邮件的包。 mail-ra.rar应当在运行时被部署到应用服务器中。
jboss-seam-mail.jar	runtime	war	Seam Mail

33.1.4. Seam PDF

表 33.4. Seam PDF依赖包

名称	范围	类型	说明
itext.jar	runtime	war	PDF库。
jfreechart.jar	runtime	war	图表库。
jcommon.jar	runtime	war	JFreeChart需要的库。
jboss-seam-pdf.jar	runtime	war	Seam PDF核心库。

33.1.5. JBoss Rules

The JBoss Rules的所有包都可以在Seam的drools/lib目录中找到。

表 33.5. JBoss Rules依赖包

名称	范围	类型	说明
antlr-runtime.jar	runtime	ear	ANTLR Runtime库。
core.jar	runtime	ear	Eclipse JDT
drools-compiler.jar	runtime	ear	
drools-core.jar	runtime	ear	
janino.jar	runtime	ear	
mvel.jar	runtime	ear	

33.1.6. JBPM

表 33.6. JBPM依赖包

名称	范围	类型	说明
jbpm-jpdl.jar	runtime	ear	

33.1.7. GWT

如果想在你的Seam应用程序中使用Google Web Toolkit (GWT)，这些包也是必须的。

表 33.7. GWT依赖包

名称	范围	类型	说明
gwt-servlet.jar	runtime	war	GWT Servlet库

33.1.8. Spring

如果你想在你的Seam应用程序中使用Spring框架，这些包也是必须的。

表 33.8. Spring Framework 依赖包

名称	范围	类型	说明
spring.jar	runtime	ear	Spring Framework库。

33.1.9. Groovy

如果你想在你的Seam应用程序中使用Groovy，这些包都是必须的。

表 33.9. Groovy依赖包

名称	范围	类型	说明
groovy-all.jar	runtime	ear	Groovy库

33.2. 使用Maven依赖管理

Maven提供依赖管理，可以用来管理Seam项目的依赖。可以使用Maven Ant Task来整合Maven到Ant构建中，或者使用Maven来构建和部署项目。

我们实际上不是在这里来讨论如何使用Maven，但只运行你能使用的一些基本POM。

Seam的发行版本在 <http://repository.jboss.org/maven2> 每夜快照在 <http://snapshots.jboss.org/maven2>。

所有Seam的工件都在Maven里可以得到。

```
<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>jboss-seam</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>jboss-seam-ui</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>jboss-seam-pdf</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>jboss-seam-remoting</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>jboss-seam-ioc</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>jboss-seam-ioc</artifactId>
</dependency>
```

下面示例的POM文件提供Seam、JPA（由Hibernate提供）和Hibernate Validator：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.jboss.seam.example</groupId>
  <artifactId>my-project</artifactId>
  <version>1.0</version>
  <name>My Seam Project</name>
  <packaging>jar</packaging>
  <repositories>
    <repository>
      <id>repository.jboss.org</id>
      <name>JBoss Repository</name>
      <url>http://repository.jboss.org/maven2</url>
    </repository>
  </repositories>

  <dependencies>

    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-validator</artifactId>
      <version>3.0.0.GA</version>
    </dependency>

    <dependency>
```

```
<groupId>org.hibernate</groupId>
<artifactId>hibernate-annotations</artifactId>
<version>3.3.0.ga</version>
</dependency>

<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>3.3.1.ga</version>
</dependency>

<dependency>
  <groupId>org.jboss.seam</groupId>
  <artifactId>jboss-seam</artifactId>
  <version>2.0.0.GA</version>
</dependency>

</dependencies>

</project>
```

附录 A. Seam 2.0 开发手册中文翻译项目

A.1. 声明

Seam中文参考手册得到Seam Framework开发团队及主要作者Gavin King的直接授权和支持，其目的是在中文世界推广优秀的开源技术。本次翻译活动由满江红开放技术研究组织（<http://www.redsaga.com>）进行翻译的组织、协调工作。项目负责人为俞黎敏。我们在此郑重宣布，本次翻译遵循原Seam Framework的授权协议，即LGPL协议。在完整保留全部文本包括本版权页，并不违反LGPL协议的前提下，允许和鼓励任何人进行全文转载及推广。我们在此宣布所有参与人员放弃除署名权外的一切权利。

A.2. 项目历程

A.2.1. Seam 1.2.1 开发手册翻译项目

我们2.0的翻译项目是基于 1.2.1 翻译项目的基础上进行的。

A.2.2. Seam 2.0 Beta 1 开发手册翻译项目

2007年09月05日

项目启动，启动一般都是在夜间啦。

并在Spring中文论坛：<http://spring.jactiongroup.net/viewtopic.php?t=3521>

（在Yanger的地盘上XXX一下，另据晓钢说Gavin King和Spring关系很好啊：））

JavaEye：<http://yulimin.javaeye.com/blog/120769>

Matrix：

<http://www.matrix.org.cn/thread.shtml?topicId=14ba3958-5bbc-11dc-b06d-09b637715141&forumId=17>

CSDN：<http://community.csdn.net/Expert/TopicView1.asp?id=5745111>

Dev2Dev：

<http://dev2dev.bea.com.cn/bbs/thread.jspa?forumID=64104&threadID=43900&tstart=0>

CJSDN：<http://www.cjsdn.net/post/view?bid=20&id=191168&sty=1&tpg=1&age=0>

等等论坛接受报名。

用于协调控制的wiki地址发布(<http://wiki.redsaga.com/confluence/display/SeamRef>)。

2007年09月07日

经过两天的大力宣传，终于有六位兄弟申请加入了，效果还不错啊，但是仍需要努力宣传，Robbin在JavaEye上帮忙置顶于论坛中的公告了，不仅仅是在海阔天空版，再次表示感谢，也由此希望

有更多的人来参加。

各位新来的兄弟有时间就把自己秀出来给大伙瞧瞧，让大家看一看不是比中国体操队的小伙子们还要帅呢：）至少我相信比司令帅是肯定的。。。：）

新来的兄弟先熟悉一下场地，然后热身热身，不着急，运动之前一定要热身，这样才能出好成绩，破纪录才有希望，热身动作有感觉不规范或不舒服的地方，就请GTalk或邮件与我，司令会手把手地帮你捏一捏：）BTW：司令百米11'6，差点达标二级，动作还是比较规范的，而且拿捏到位，重要的是免费；可惜现在仍处于三级的阵营里，就差点。。。。

基于原来1.2.1的任务有好几个领取了，但一直没有进度更新，各兄弟能否抽空瞅两眼呢？

周末，大家就好好休息了吧。。。休息好了，别忘记了还有人在挥鞭赶虎。。。。

2007年09月09日

有三个任务领取，并完成了两个，但是由于Wiki的问题导致新来的人员无法领取任务，这个问题需要晓钢尽快想办法解决一下。

其它的任务大家继续领取，Wiki上的任务领取表暂时由我来更新。

2007年09月11日

6. conversations.xml 34K 10P seanchan working

7. jbp.m.xml 32K 10P 差沙 翻译-85% 2007-04-15

17. mail.xml 26K 7P Xiaogang Cao Working

27. components.xml 68K 11P jiaochar Working

28. controls.xml 47K 13P blackwing Working

30. testing.xml 10K 6P geshe Working

31. tools.xml 23K 9P geshe Working

今天是个特别的日子，到目前为止，我们的工作进度也相当顺利，任务都有人进行领取并开始工作；

其中 yeshucheng 同学最为积极，领取了四个，完成了三个，效率之高令人佩服啊！（掌声响起来。。。），按我女儿的一贯说法是，你是第一名，奖一个星星给你：

另外，由于以上几个任务时间比较久了，能否报告一下现在的进度，如果时间上有困难，将由其它同学来领取，请在收到邮件通知后帮忙更新一下进度，若是时间上有困难就准备移交其它同学来接手了。谢谢合作。

根据进度，我们就即将进入比较艰苦的阶段了，一审阶段，一些需要注意的地方我发布在 翻译、校对注意事项（必读）特别是新加入者了，大家先去了解一下。

2007年09月13日

把原来已领取但是未完成而没有进度更新的几个任务收回了，并且分配给申请加入团队的人员进行试译，免得试译原来一大段的内容，浪费了时间，可惜了资源，现在讲的是资源节约型社会啊。我

们可不能浪费与铺张了啊。

yeshucheng 同学四个任务基本上都完成了，除了一个任务里有些地方他不是很确定之外，等一审时再进行了。不过听他说这两天严重感冒，司令也发表慰问，并严重关注事态的发展，祝早日康复，继续来进行一校的工作。

试译的进度中，mail.xml与testing.xml已经完成，并初步通过审核，基本上达到要求。将此任务直接分配给试译人员，希望能够再次进行一些Review工作，并提交CVS上面来。

2007年09月15日

越来越多的同学们申请加入了，目前的工作开始进入一审工作了，希望正在试译的同学们尽快完成试译，并提交回来，以便一审工作顺利地进行下去。

刚加入的小周同学zaya，在比利时取得计算机本科毕业，目前在国内，利用这个空余的时间加入了我们，我们热烈表示欢迎。并且在一审的工作中完成的相当快速。

已经完成了 gettingstarted.xml 的一审工作，关于他在一审工作中所做的审校记录，我在这里进行公布，并把他的一些所想与建议放上来，大家可以参考。

一审二审之最佳实践，审校者必看

2007年09月17日

为了方便大家集中精力进行一审工作，我索性把所有的文件进行了比对与合并，这样大家可以直接在cn-2.0.0B1下面的文件进行直接的一审了，希望可以加快大家的一审速度。

同时我也建立了相关的build及相关的文件，并生成了内部 CHM 版本了，等翻译阶段的任务全部完成后，我们就可以build出第一个版本来供内部阅读了。

一审的工作有四个开始领取并工作，加沿努力干啊，中秋就快要到了，我马上就要给大伙发月饼了啦：)

由于 richard 同学临时项目上马，暂且不能翻译 groovy.xml 章节，有兴趣的同学可以开始抢啦。。。不过我可就要近水楼台先得月了啊：)

2007年09月19日

又有两位新同学申请加入了，让他们工作在试审上面，待试审通过后正式加入。一审有任务完成情况，任务申请进度不错，继续加油啊， groovy.xml 也有人认领了，动作真快啊，现在就是 itext.xml 的问题了，不知道找得回来不？找不回来就得重新开始翻译了。

另外：试译与试审的同学速度要加快一些哦：) 这些天接连加班，累，眼睛都睁不开了。。。

2007年09月21日

groovy.xml 翻译完成，速度不错，晓钢的 concepts.xml 也翻译完成了啦，我合并了，kuuyee也开始一审了，抢得真快；) conversations.xml、configuration.xml 也翻译合并完成，可以开始一审工作，在正式加入的成员当中，翻译与一审的进度都不错。唯一感到有点儿遗憾的是刚申请加入的同学们在试译与试审的进度上面如果能够象已正式加入的同学们的速度一样快的话，那就很好了。

近日，秋风送爽，进入秋天了，中秋也就快到了。在这里提前给同学们发月饼了，祝大家中秋节快乐！月饼

小周同学 yaza 在一审的工作中相当的认真，实在是佩服！（掌声鼓励，响起来。。。），请看一审二审之最佳实践，审校者必看 他所做的工作记录。

2007年09月23日

到今天晚上为止，一审的工作有五个完成，一个完成80%，三个在一审中，接近一审工作量的20%，但是 webservices.xml controls.xml tools.xml 的试译工作还没有完成，itext.xml现在找不到原来的版本，而且Xu MingMing同学也一直没有提交上来，只能开始重新翻译了。目前也完成20%左右了。一审的工作还需要大家的挂号开工啊。。。

2007年09月26日

大家是否还沉醉于中秋佳节的欢乐当中呀，近来进度很一般呀，司令很生气呀。。。：（

有时间的兄弟看看进度表嘛，看看有没有适合自己的就领取回家过节呀。。。过完节就回归到CVS上嘛；）

另外要批评一下

20. webservices.xml 9K mazhao试译中 Working 2007-09-07

31. tools.xml 23K 9P junjzheng试译中 2007-09-12 28. controls.xml 47K 13P huwenjiemaster试译中 2007-09-13

三位试译的同学，时间也太久了吧，此邮件发出后，如果还没有收到你们的确切回复的话，就准备取消你们的试译请求了，多谢配合。

2007年09月28日

热烈欢迎 DigitalSonic 回归团队！

2007年09月29日

mazhao 的电话没有人接听，junjzheng 电话联系了，他在努力中，国庆节后提交，huwenjiemaster 电话联系过了，因近期出差暂时取消。

2007年09月30日

阶段性月总结：3 5 个小任务

一、翻译情况：完成 3 1 个，还有 4 个在试译中

16. itext.xml 51K 11P topquan试译中

20. webservices.xml 9K mazhao试译中

28. controls.xml 47K 13P xukai试译

31. tools.xml 23K 9P junjzheng试译中

二、一审情况：完成 8 个，正在试审 5 个，一审工作进行中 4 个，仍未申领 1 8 个（包含 4 个试译）

国庆节同乐！有时间来关注关注一下：）

2007年10月07日

国庆长假结束了，七天时间大家玩得开心不？别忘记把去玩的心得与相片 Share 出来呀：)

在这次国庆长假里，我们的 DigitalSonic 同学完成了 3 个一审，曹晓钢同学完成了 3 个一审，yeshucheng 同学完成了 1 个一审，1 个在进行中，大家鼓掌（掌声照样响起来。。。)

2007年10月09日

收到一个试译的反馈结果，晓钢进行了全文批注与一审，yeshucheng 同学也完成了 1 个一审。其它同学估计都还没有从长假综合症恢复过来：)

2007年10月11日

Hi，各位，醒醒啊，怎么没有动静呢？长假综合症 continuing ... ：（

2007年10月13日

欢迎一位新同学的申请加入，估计大家再休息完这一周后元气应当可以恢复过来了啊。。。长假综合症 continuing ... ：（

2007年10月15日

今天是一个特殊的日子！

由于两次无法联系上 webservises.xml 的试译者，现将之取消，另电话联系了 itext.xml 试译了20%的同学，他也因时间关系而无法继续参与进来，也将此任务暂时取消。其它同学可以继续领取了，请试审的同学与本周末之前提交试审的结果，过了本周末之后，将自动取消你的任务与加入请求，谢谢！

2007年10月19日

都在参加代表大会，忙碌中。。。)

2007年10月21日

这两天，我再次邀请到两位新同学加入我们的团队了。

翻译工作只剩下三个试译的工作，希望能够在本周之内完成全部的翻译工作。

一审工作还有 6 个没有人领取的，以目前的进度来看，11月5号我们应当可以预期完成翻译的全部工作。并且我们的一审工作也提前进行并完成了约 50%，可以乐观地估计我们应该能够按时完成此次的翻译工作，感谢大家这一段时间以来的辛苦工作与努力奉献！谢谢！

二审工作也可以同步开始进行了，希望有时间的同学积极地进行领取与审校！

努力！努力！再努力！我们一定可以完成这次光荣的任务：) 谢谢！

2007年10月24日

喜喜喜：

1、十七大会议胜利圆满地闭幕了！

2、嫦娥一号今天晚上也成功发射，并正常进入地球轨道了！

3、我们的翻译任务也只剩最后一个的20%的试译了！

4、一审工作也在紧张地进行中，希望大家继续领取任务！

5、开个小会，现在Seam 2.0.0出了RC2了，我们的二审工作是否考虑再次进行合并呢？还是等正式版出来后再进行合并？

05.10.2007: Seam 2.0.0.CR2 has been released.

18.09.2007: Seam 2.0.0.CR1 has been released.

27.06.2007: Seam 2.0.0.BETA1 has been released.

上面是Seam的发布时间，估计CR2后应当就是正式版了吧。

2007年10月25日

不喜不喜：

最后一位试译的经电话联系也因时间关系可能无法全部完成了。

试译的两位同学电话联系后也因各自的原因而退出了。

没有关系，本周末的目标是完成所有的翻译任务，并争取一审完成99%：)

2007年10月27日

看来最后一位试译的同学也无法及时完成任务了。。。

2007年10月29日

xukai的试译工作并没有完成，并且在电话通知三天后也没有提交任务进度情况与文档，特此取消其试译资格！

2007年10月29日

里程碑事件：

经过近两个月努力奋战，到今天为止，翻译工作全部完成，在此感谢所有参与的翻译人员！谢谢你们了，辛苦了！

现将翻译的工作任务进行统计如下：（按完成的任务个数的多少，并不准确，因为有页数的关系）

表 A.1. 参与人员列表

翻译人员	完成的翻译工作量	备注		
yeshucheng	4			
CaoXiaogang	3			
DigitalSonic	3			
crazycy	2			

翻译人员	完成的翻译工作量	备注		
Echo	2	女		
seanchan	2			
YuLimin	2			
YY	2	女		
agile_boy	1			
alexchang	1			
caoer	1			
chentianyi	1			
downpour	1			
jiaochar	1			
junjzheng	1			
kuuyee	1			
lyfcdy	1			
magice	1			
mochow	1			
pesome	1			
yeby	1			
差沙	1			

2007年10月31日

一审工作形势仍比较严峻啊：

由于一直联系不上无锡prajack许杰同学，故将其试审的权利取消。

目前一审未完成的情况如下：

一审中的有 8 个，仍有 3 个一审工作等待认领中。共计 11 个工作任务， 相对于 34 个任务而言，现在为止才完成 60 %， 4 周之内要完成的形势还是比较困难啊。要努力啊！！

2007年11月05日

一审继续进行，已经完成一个，新任务也被认领了一个。现在Seam 2.0.0 GA也出来了，我下载了xml文档，对比了一下，修改的地方不多，争取在二审的时候进行合并，同步发布中文版。

2007年11月07日

。。。。。。继续努力干活中。。。。。。

2007年11月09日

一审工作都有同学认领了，只剩下五个工作任务就完成一审工作了，提前完成一审的全部工作是胜利在望了啊。。。加油啊。。。

2007年11月11日

本来是想2007.11.11 11:11:11发布预览版，无奈节日到了，人都跑光了，任务也就。。。哈哈。。。。1111111111节快乐。。。

2007年11月13日

进展顺利，一审工作还有一个就全部完成了，现在开始着手准备预览版的发布与二审的工作了。

二审的工作将以正式版进行合并后开始二审。

2007年11月14日

里程碑事件：

一审工作结束，二审工作准备中，合并到G A版的工作我已经完成了，二审的工作采取Leader分配制，并由任务责任者完成任务。

A. 2. 3. Seam 2.0 正式版开发手册翻译项目

2007年11月15日

二审合并工作完成，二审的工作开始了。最后一关的审核工作，需要在审校的过程中把一些术语集中出来放到术语表中去。

术语表：<http://wiki.redsaga.com/confluence/pages/viewpage.action?pageId=1751>

注意：请更新CVS的内容，二审是在 cn2.0.0GA 目录下进行的！

2007年11月17日

二审的任务还是比较艰巨的，或许延期是不可避免的，但是质量把关是第一要素。加油。。。

2007年12月22日

太多事情太多话要说了。那就简明扼要吧：冬至节快乐！！！！

2007年12月24日

非常高兴我们又有两位漂亮的MM（王琳、胡燕）加入我们的翻译队伍中来了，现在我们的队伍到了“男女搭配，干活不累”的共产主义阶段了！：）

2007年12月30日

所有的二审工作完成，发布RC版，感谢所有参加翻译与审校的兄弟姐妹们，感谢一直关注我们翻译活动的朋友们，谢谢你们了！！！！

2008年03月27日

满江红开放技术研究组织发布Seam 2.0中文文档正式版
三个月前，我们发布了RC版，详见：

<http://yulimin.javaeye.com/blog/151917>

现在我们正式发布Seam 2.0中文文档正式版，希望对大家有所帮助，谢谢。

同时，再次感谢王琳、马越、晓钢在发布RC版后，对全部译文进行通读，发现并纠正其中的问题

。

虽然我们正式宣布正式版发布，但是也希望广大朋友能够在阅读的过程若发现有疑问的地方，及时提出来进行讨论并加以修正，共同提高文档的质量，共同为开源活动贡献自己的一份力量。

最后，感谢所有参与翻译与审校的兄弟姐妹们，所有的人员都将获得此次活动的纪念专题T-Shirt 一件！请在这里进行登记与领取。

<http://wiki.redsaga.com/confluence/pages/viewpage.action?pageId=2791>

错误难免，有则改之，无则加勉！

在线阅读与下载地址详见：

<http://wiki.redsaga.com/confluence/display/SeamRef/Home>