J2EE 核心开发模式(V0.1)

Core J2EE Patterns

WebWork+Spring+iBATIS

版权申明
福建新东网科技有限公司版权所有，保留一切权利。
未经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档的部分或全部，并以任何形式传播。

文档控制页

修订历史			
版本号	日期	作者	修改说明
V0.1	2006-03-21	李壮相	初稿

目 录

1 导论......1
1.1 什么是 J2EE......1
1.2 什么是模式......2
1.2.1 模式的定义......2
1.2.2 模式的分类......3
1.3 模式目录......4
1.3.1 演化过程......4
1.3.2 使用模式的益处......6
1.4 模式、架构和重用......7
2 表现层设计考虑和不佳实践......9
2.1 表现层设计考虑......9
2.1.1 会话管理......9
2.1.2 控制客户端访问......10
2.1.2.1 保护视图......11
2.1.2.2 过滤配置实现保护......12
2.1.3 重复的表单提交......13
2.1.4 验证......15
2.1.4.1 在客户端验证......15
2.1.4.2 在服务器端验证......16
2.2 表现层不佳实践......21
2.2.1 视图中包含业务控制代码......21
2.2.2 表现层数据结构暴露在业务层......22
2.2.3 允许重复提交表单......22
2.2.4 敏感资源暴露在客户端......23
3 业务层设计考虑和不佳实践......24
3.1 业务层设计......24
3.1.1 使用 session bean......24
3.1.2 使用数据传输对象(DTO)......26
3.2 业务层不佳实践......27
3.2.1 把 HttpServletRequest 直接当做 DTO......27
3.2.2 把 ResultSet 直接当做 DTO......27
4 J2EE 重构......28
4.1 表现层的重构......28
4.1.1 引入 WebWork 标签......28
4.1.2 引入 WebWork Action......31
4.1.3 引入数据访问对象(DTO)......31
4.2 业务层的重构......32
4.2.1 引入数据访问对象(DTO)......32
4.2.2 引入数据访问对象(DAO)......33
4.2.3 引入业务门面(FACADE)......35
5 J2EE 模式概览......39
5.1 什么是模式......39
5.2 分层思路......39
5.3 J2EE 模式......41
5.3.1 表现层模式......41
5.3.2 业务层模式......41
5.3.3 持久层模式......41
5.4 J2EE 模式关系......42
6 表现层模式......43

6.1 使用 Actions....45
6.1.1 问题....45
6.1.2 解决方案....45
6.1.2.1 Action 接口....45
6.1.2.2 ActionSupport 基本类....46
6.1.2.3 理解基本校验....46
6.1.2.4 使用 ModelDriven Action....47
6.2 拦截器....49
6.2.1 问题....49
6.2.2 解决方案....49
6.2.2.1 拦截器调用....49
6.2.2.2 调用源码剖析....49
6.2.2.3 已封装的拦截器堆栈....50
6.3 使用 Results....52
6.3.1 问题....52
6.3.2 解决方案....52
6.3.3 Action 生命周期之后....52
6.3.4 常规 Results....52
6.4 使用标签库....56
6.4.1 问题....56
6.4.2 解决方案....56
6.4.2.1 数据标签....56
6.4.2.1.1 Property 标签....56
6.4.2.1.2 Set 标签....57
6.4.2.1.3 Push 标签....57
6.4.2.1.4 Bean 标签....58
6.4.2.1.5 Action 标签....59
6.4.2.2 控制标签....60
6.4.2.2.1 Iterator 标签....60
6.4.2.2.2 If/Else 标签....60
6.4.2.3 杂项....61
6.4.2.3.1 Include 标签....61
6.4.2.3.2 URL 标签....61
6.4.2.3.3 i18n 和 Text 标签....62
6.4.2.3.4 param 标签....63
6.5 Context 对象....63
6.5.1 问题....63
6.5.2 解决方案....63
6.5.2.1 web.xml....64
6.5.2.2 XWORK DTD....66
6.5.2.3 XWORK.XML....66
6.5.2.4 validators.xml....67
6.5.2.5 *Action-validation.xml....68
7 业务层模式....69
7.1 数据传输对象(DTO)....69
7.2 业务门面(Facade)....70
7.2.1 问题....70
7.2.2 解决方案....70
7.2.2.1 业务封装....70
7.2.2.2 事务控制....72
7.3 Context 对象....76

7.3.1 问题......76
7.3.2 解决方案......76
7.3.2.1 web.xml......76
7.3.2.2 Ibatis-Context.xml......77
8 持久层模式......80
8.1 数据传输对象(DTO)......80
8.2 数据访问对象(DAO)......80
8.2.1 问题......80
8.2.2 解决方案......81
8.2.2.1 经典分页......82
8.2.2.2 自定义分页......82
8.3 Context 对象......84
8.3.1 问题......84
8.3.2 解决方案......84
8.3.2.1 sqlMap-config.xml......84
8.3.2.1.1 settings 节点......86
8.3.2.1.2 transactionManager......86
8.3.2.1.3 dataSource 节点......87
8.3.2.1.4 sqlMap 节点......87
8.3.2.2 jdbc.properties......87
8.3.2.3 映射文件......87
8.3.2.3.1 tf_user.xml......87
8.3.2.3.2 数据关联......90
8.3.2.3.2.1 一对多......90
8.3.2.3.2.2 一对一......91
8.3.2.3.3 存储过程调用......91
9 尾声......94

1 导论

最近 5 年来，企业软件开发领域发生了极大的变化。处于变化中心的正是 Java2 企业版平台(J2EE)；它为开发分布式的、针对服务器的应用系统提供了一种统一的技术平台。J2EE 技术具有高度的战略意义和强大的功能支撑，因此，整个软件开发社区收益于这种开放的标准——我们可以依据此标准来为企业开发基于服务的软件架构。

新东网公司多年来一直采用 J2EE 进行软件开发，同时也积累了很多成功的经验，在开发过程也积累了很多固化的模式，但是随着技术的进步，我们有必要对原有的模式进行重新抽象并归纳，并且通过一种标准的模式模板记录、交流模式，从而构成了一种非常强大的沟通、重用机制，由此我们设计、构建软件方法也得到了极大的改进。

1.1 什么是 J2EE

J2EE 是一种用来开发分布式企业软件应用系统的平台。Java 语言从创生之日起，就获得了广泛接纳，经历了巨大的发展。越来越多的技术都成了 Java 平台的一部分，为了适应不同的需要也开发出了很多全新的 API 和标准。最终，SUN 公司联合了多家业界巨头，在开放的 Java 社区组织名义下，把所有与企业开发相关的标准，API 整合起来，构成了 J2EE 平台。

对于企业，J2EE 平台有很多的优势：

- J2EE 为企业级运算的许多领域(比如数据库连接、企业业务组件、面向消息的中间件(MOM)、Web 相关组件、通信协议以及互操作性)设立了标准。
- J2EE 促进人们基于开放的标准开发软件；如此构建的系统实现，出自名门、安全稳固，因此 J2EE 就构成了一种可靠的技术投资。

- J2EE 是一种标准的开发平台，基于此开发的软件组件能够在不同厂商的产品中相互移植，从而避免了被一个厂家锁定。
- 在软件开发过程中采用 J2EE 能够缩短开发周期，使产品尽快投放市场——这是以为，系统的很多底层架构和基础部分都已经有产品厂商按照 J2EE 规范标准实现出来了。因此大多数 IT 企业可以不再开发中间件，集中精力构建符合自己商业需要的应用。
- J2EE 提高了程序员的生成力，因为对于 Java 程序员们，相对来说很容易就能够学会基于 Java 语言的 J2EE 技术。所有企业软件开发都能够在 J2EE 平台上、利用 Java 语言完成。
- J2EE 增进了现存各种异构系统之间的互操作性。

1.2 什么是模式

1.2.1 模式的定义

模式是人们用来讨论问题和解决方案的。简言之，有了模式，我们就能够记录那种已知重复出现的问题，以及在特定上下文中对它的解决方案，并且，借助模式我们还能和他人讨论这些内容。“重复出现”这个词是前文的一个关键要素，因为模式的目的，就是要鼓励重复进行概念重用。

Christopher Alexander 在《模式语言》[Alex2]中给模式下了一个非常有名的定义：

“每个模式都是一个法则，有三部分组成。它表现的是一种特定的上下文，一个问题和一个解决方案之间的关系。”

——Christopher Alexander

其后 Alexander 进一步扩展了他的定义，另外模式领域的著名专家 Richard Gabriel[Gabriel]也更详尽地讨论过这个定义[Hillside]。Gabriel 把 Alexander 的定义改写成了一个适用于软件业的版本：

“每个模式都是一个法则，有三部分组成。它表现的是一种特定的上下文，一个特定的、在该上下文中重复出现的约束系统，以及一种能够引导这些约束自身解决的软件构造这三者之间的关系。”(见《Hacking 的永恒之道》)

——Richard Gabriel

这个定义看起来相当严格，不过也还另有一些比较宽松的定义。比如说，Martin Fowler 在《分析模式》[Flower2]中就是这么定义模式的：

一个模式，就是在某个实际上下文中有用，并且或许在其它上下文中也有用的想法。

——Martin Flower

如你所见，模式有很多种定义，但是所有这些定义似乎都有一个共同的主题，那就是在一种特定的上下文中反复出现的一个“问题/解决方案”对子。

- 模式有以下共通的特性：
- 模式是通过经验观察得出的。
- 一般来说模式都要按一种专门的结构格式来记录。
- 采用模式能够避免重新发明轮子。
- 不同的模式出于不同的抽象层次上。
- 模式要经历不断的改进、完善。
- 模式是可以重用的工作。
- 模式可以用来让大家交流系统设计和最佳实践。
- 多个模式可以拼合起来，从而解决一个大型问题。

1.2.2 模式的分类

既然模式表现的是在一种特定上下文中专家对反复出现的问题的解决方案，那么在很多不同的抽象层次、各种各样的业务领域中就会发现模式。对于软件模式，人们想出过很多种分类，最常见的如下：

- 设计模式
- 架构模式
- 分析模式
- 创建模式
- 结构型模式
- 行为型模式

我们仅仅简要地列出这几种类别，但其中已经包含了多种不同的抽象层次，也有很多种正交的分类模型了。事实上，人们已经提出过很多种不同的分类方法，但是为了利用模式记录你的思考，并没有哪种分类方法是唯一正确的。

对于我们提供的那个目录里的模式，我们就简单地称作 J2EE 模式。其中每个模式都既是设计模式也可以算是架构模式，而每个模式的"策略"部分，则会考察一些抽象层次比较低的技术细节。所以我们采用的分类方法，仅仅是把模式按照逻辑架构层次分为三类：

- 表现层
- 业务层
- 持久层

也许，当这个模式目录演化到一定地步的时候，模式的数量会比现在多很多，以至于必须放弃这么简单的三分法，而采用某种更复杂的分类方式。不过在目前我们还是保持简明，没有必要就不引入新的术语概念。

1.3 模式目录

1.3.1 演化过程

在早先的日子里，当我们基于 J2EE 平台设计、开发、实现各种各样不同的系统时，我们就已经开始记录自己的经验了，但是当时记录的方式还不是很正规，

多数都是以设计考虑、设计想法以及札记的形式完成的。随着这个知识库的增长，我们也认识到，需要以稍微正规一些的文档格式来固化、交流这一类知识。因此，我们改用模式的格式来记录这些想法，因为要想把那些关于重复出现的问题和解决方案的知识记录下来并用于交流，模式是最理想、最合适的一种形式了。

这项任务的第一步，就是要选定合适的抽象层次，用来划分这些模式。有一些问题的解决方案之间会发生重叠，因为有一些问题的核心是相同的，但解决方案的实现方式则有所差异。为了体现这种重叠，我们必须考虑抽象层次问题以及在定义每个模式采用何种粒度的问题。正如你现在所看到的这个 J2EE 模式目录这样，我们最终选取的抽象层次介乎设计模式和架构模式之间。与具体解决方案相关，有很多抽象层次较低的实现细节，在我们的模式模板中，我们把这些细节放到“策略”部分来讨论。这样一来，我们可以首先从一个比较高的抽象层次讨论模式，同时也没有牺牲对具体实现细节的考察。

每个模式都要经过很多次的命名和重命名。而且实际情况中，每个模式也都经过了很多次的重写。不用说，本文收录的这些模式，都处于持续的改进过程中，随着技术和规范的变化也一定会继续演进。

表 1-1 列出了目录中收入的模式。

表 1-1　J2EE 模式目录中的模式

<table>
<tr><th>层</th><th>模式名称</th></tr>
<tr><td rowspan="5">表现层
Webwork</td><td>Action 驱动模式</td></tr>
<tr><td>XWork 拦截器体系</td></tr>
<tr><td>Results 对象</td></tr>
<tr><td>标签(Tag)</td></tr>
<tr><td>Context 对象</td></tr>
<tr><td rowspan="3">业务层
Spring</td><td>业务门面(Facade)</td></tr>
<tr><td>数据传输对象(DTO)</td></tr>
<tr><td>Context 对象</td></tr>
<tr><td rowspan="3">持久层
iBATIS</td><td>数据访问对象(DAO)</td></tr>
<tr><td>数据传输对象(DTO)</td></tr>
<tr><td>Context 对象</td></tr>
</table>

1.3.2 使用模式的益处

你可以使用本文中的 J2EE 设计模式来改善你的系统设计，并且可以在项目生命周期的任何一点上应用这些模式。目录中记录的模式处于一个比较高的抽象层次上，所以在项目前期应用模式就会大有裨益。但也可以采用另一种方式：如果你在具体实现阶段应用一个模式，也许你就需要改写现有的代码。在这种情况下，第 4 章的“J2EE 重构”介绍的种种重构可能会对你有所帮助。

使用模式有什么益处？下面的段落中，我们介绍在项目中使用、采用模式的一些益处。简单地说，模式能够：

- 让你利用一个经过验证可行的解决方案。
- 为你提供一套共通语汇。
- 约束解决方案的空间。

让你利用一个经过验证可行的解决方案

模式提供的解决方案已经在不同的时间、不同的项目中被反反复复地用于解决类似的问题。所以，模式构成了一种强大的重用机制，能够让开发者、架构师避免重新发明轮子。

为你提供一套共通语汇

模式为软件设计者提供了一套共通的语汇。做为设计者，我们使用模式不仅有助利用、复制成功的设计，而且还有助于在开发者之间通过共通的语汇和格式进行交流想法。

一个设计者如果不依靠模式，那就需要花上更多的力气才能把自己的设计介绍给其它设计师和开发者。利用模式的语汇，软件设计师能够高效地进行交流。这和真实世界中的情形相似：日常生活中，我们相互沟通、交换意见也要用上一套共通的语汇。就像真实世界中一样，开发者通过学习、理解模式而积累起自己的语汇，而记录下来新的模式之后，大家的设计语汇也就相应增长了。

一旦开始应用这些模式，你就会注意到自己很快就能把模式名称融汇到自己的语汇中——而且你也就不再使用那些罗嗦冗长的说法，只简单地提及模式名称了。比如说，假设需要解决问题的方案必须要使用数据传输对象模式。一开始，你可能会直接描述这个问题，没有给它加上模式标签。你会说，你的应用需要在业务之间交换数据，需要在调用过程中网络负载以下尽可能提升系统性能，等等。可是，一旦你学会了对这个问题采用数据传输对象模式、再遇到类似情况，你就会使用这个简单的术语描述了，而且直接能够从这个模式入手进行开发。

约束解决方案的空间

模式的应用引入了一种重要的设计元素——约束。应用了模式，也就给最终的解决方案的空间带来了约束，或者说创造出了一种边界，设计和实现都必须在这个边界内完成。因此，模式强烈要求开发者要让系统实现遵从边界。如果实现越出了边界，就会破坏对模式和设计的遵从，这就可能导致出现未经意料的“反模式”。

但模式并不会扼杀创造性。恰恰相反，模式描述了某种抽象层次上的结构或构造。为了在这个边界之内实现模式，设计师和开发者还可以做出很多具体的选择。

1.4 模式、架构和重用

所谓“软件重用”，真是一个了不起的目标，我们多年以来一直追求实现重用，但至今只获得了相对的、不太显著的成功。事实上，多数商业软件的成功重用都出现在用户界面领域，而不是在业务组件领域——但后者恰恰是我们关注的焦点。作为业务系统架构师，我们力求推进重用，而我们关心的重用都处于设计和架构的层次上。对于推进这个层次上的重用，J2EE 模式目录是一种非常有效的途径。我们举一个简单的例子，两个同样采用对用户管理的模块，一个采用原生的 JDBC 进行关系型的模式，而一个则采用持久层进行对象型的模式，这显然功能如何相似，那怕是一模一样也很难进行重用，只有统一的模式，重用才成为可能。

目录中的各种模式之间存在多种关系，这些关系常常被视为是**模式语言**的一部分。另外，还有一种描述关系的方法，那就是利用“**模式框架**”的概念——所谓模式框架，也就是在一个整合的应用场景之下的一组模式组合。在很多时候，我们都要在模式的层面上把解决方案组合起来，或者把组件装配到一起，此时“模式框架”的概念就非常重要。“微架构”利用了这一概念，并为组合应用模式形成提供了一个更广阔的基础。

所以，开发者们不仅要孤立地理解单个模式，还必须注重它们的关系和组合；事实上开发者们也一直在询问怎样才能最好地把多个模式连接在一起，形成大型的解决方案。我们所说的“**利用 J2EE 模式框架**”，恰恰指的就是按这种方式把目录中的模式组合起来。在这个语境中，所谓“框架”也就是将模式连接起来，形成一个解决方案以实现一组需求。我们认为，这样的应用方式将推动下一代 J2EE 开发的发展，而对“模式驱动的开发过程”实现自动化需要：

确定应用场景，为系统的每个层次提出合适的模式。

确定模式组合(或者说解决方案的主旨)，从而给出模式框架。

为系统方案中的每个角色选定具体实现策略。

到此为止，你应该对模式的构成、有了比较充分的理解。下一章将讨论表现层的设计考虑和不佳实践。

2 表现层设计考虑和不佳实践

2.1 表现层设计考虑

当开发者应用本文 J2EE 目录中列出的表现层模式时，也应该考虑一些相关的设计问题。这些问题在不同的层面上和应用模式有关，而且，它们也会影响系统的很多方面，比如安全、数据完成性、可维护性与扩展性。我们将在本章讨论这些问题。

虽然很多类似的问题都可以用模式的形式记录，我们却决定不这样做，因为，与模式目录中的表现层相比，它们处理的是抽象层次更低的内容。所以，我们不按照模式的形式记录它们，而是采用一种更随意的方法：在基于我们模式目录实现系统的时候，你肯定会考虑到这些问题，我们正是从这种考虑的角度描述每个问题的。

2.1.1 会话管理

“用户状态”这个概念，描述的是在客户端和服务器之间的多次请求构成的一种“对话”，以下段落的讨论都是基于这个“用户会话”概念的。

在客户端保存会话状态

把会话状态保存在客户端，就需要把会话序列化，并且把它放进 HTML 视图页面中，传回客户端。相对来说容易实现，当要保存的状态比较少时，效果非常好。而且在多台物理服务器上实现负载均衡时，使用客户端保存会话状态策略，也不需要复制会话状态。可以采用两种方式——HTML 隐藏字段，和 HTTP

cookie。

在客户端保存会话状态时的安全问题

当你把会话状态保存在客户端时，也就引进必须考虑的安全问题。如果你不想把数据暴露给客户端，那么就需要采用某种加密手段来保护数据。

虽然一开始，在客户端保存会话状态相对容易实现，但是这种方法有很多缺点，就是要克服它们可能需要不少时间和脑力。所以，对于处理大量数据的项目来说——企业系统大多如此——这些缺点远远抵消了它的优点。

在表现层保存会话状态

当会话状态由服务器管理时，就通过了一个会话 ID(session ID)来获取，状态通常都在服务器端持久保存，除非发生以下情况中的一种：

- 超过了预先定义的会话超时期(timeout)。
- 人工指定会话无效。
- 一个状态从会话中被删除了。

注意：如果服务器关机(shutdown)了，一些在内存中进行会话管理的机制可能就没有方法恢复原有的会话数据。

显然，当会话状态量很大时，就应该在服务器端保存会话状态。状态保存在服务器上，你就不会收到数据量大小或是数据类型方面的限制。另外，既然会话状态不会在每个请求中都通过网络传输一次，系统性能也就不会受到影响。

2.1.2 控制客户端访问

出于很多原因，我们需要控制客户端对特定应用资源的访问。在本节中我们就考察两种场景。

限制或控制客户端访问的原因之一，是保护视图或视图的一部分，不令其被客户端直接访问到。比如只有注册/登录后的用户才能访问某个特定视图时，或者视图的某个特定部分只能被某种角色的用户访问时，就会出现这样的需求。

2.1.2.1 保护视图

在视图的处理逻辑中实现保护，有两种常见的变体。一种是阻塞对整个资源的访问，另一种只阻塞对局部资源的访问。

要么全部保护一要么没有**(all-or-nothing)**

```
<%@ taglib prefix="ww" uri="webwork" %>
<ww:i18n name="'messages'">
<ww:if test="#session['loginUser'].role =='manager'">
...
<HTML>
…
…
…
</HTML>
</ww:if>
</ww:i18n>
```

对页面局部的保护

```
<%@ taglib prefix="ww" uri="webwork" %>
<ww:i18n name="'messages'">
<HTML>
…
…
<ww:if test="#session['loginUser'].role =='manager'">
<b>This should be seen only by managers!</b>
</ww:if>
…
</HTML>
</ww:i18n>
```

2.1.2.2 过滤配置实现保护

采用过滤器方式，编写一个简单的基于角色的过滤器，在过滤器中对用户的角色进行判断，如果角色不符将不能判定非法。这种基于目录级的权限过滤，安全系数非常高。以下是一个过滤器的样例。

```
public class managerfilter extends HttpServlet implements Filter {
    private FilterConfig filterConfig;
    public void init(FilterConfig filterConfig) throws ServletException {
        this.filterConfig = filterConfig;
    }
    public void doFilter(ServletRequest request,
                         ServletResponse response,
                         FilterChain filterChain)
                         throws IOException, ServletException {
        try {
            if(filterConfig == null) {
                return;
            }
            UserInfo userinfo=
            (UserInfo)((HttpServletRequest)request)
                    .getSession()
                    .getAttribute("loginUser");
            if(userinfo!=null
             && userinfo.getRole().equalsIgnoreCase(“manager”))
            {
                filterChain.doFilter(request, response);
            }
            else
            {
                response.setContentType("text/html;charset=GBK");
                PrintWriter out = response.getWriter( );
                out.println("<script language=javascript>alert('权限不足！ ') ");
                out.println("window.location.href=\"/index.jsp\";");
                out.println("</script>");
            }
        }
```

```
            catch(ServletException sx) {
                  filterConfig.getServletContext().log(sx.getMessage());
            }
            catch(IOException iox) {
                  filterConfig.getServletContext().log(iox.getMessage());
            }
      }
          public void destroy() {
            this.filterConfig = null;
      }
}
```

并在 Web.xml 配置，配置到想设定的需要过滤的目录以及目录下的某些资源。

```
<filter>
            <filter-name>managerfilter</filter-name>
            <filter-class>com.doone.login.managerfilter</filter-class>
</filter>
<filter-mapping>
            <filter-name>managerfilter</filter-name>
            <url-pattern>/manager/*</url-pattern>
</filter-mapping>
```

2.1.3 重复的表单提交

使用浏览器客户端的用户一个不留意，就可能按了“后退”按钮，把已经提交过的表单重新提交一次，这就可以引起一个重复的事务操作。与此类似，用户还可能在接收确认页面之前就点击了“停止”按钮后又重新提交相同的表单。对于大多数这类情况，我们都希望能够捕捉、禁止这些反复提交，应用 servlet 控制器就能够提供一个处理这中问题的控制点。

同步器令牌

这种策略专门处理重复表单提交的问题。在用户会话中设置一个“同步器令牌”，每一个返回给客户端的表单中都包含者这个令牌。当表单提交时，就对表单中的同步器令牌和会话中的同步器令牌做出比较。当同一个表单第一次提交

时，两个令牌的值是匹配的。如果令牌不匹配，那么提交的表单就被禁用了，并会返回一个错误信息给用户。发生令牌不匹配的情况可能是：用户提交了表单之后，又点击了浏览器的“后退”按钮，然后重新提交同一表单。

另一方面，如果两个令牌的值匹配，那么我们就能够确认控制流程是运行无误的。这时，会话中的令牌值更改为一个新数值，同时表单提交也获得了接受。

在 Webwork 已经考虑到了这个重复提交的问题，采用拦截器解决这个问题。在 webwork-default.xml 的配置中，我们可以看到以下配置：

```
<interceptor name="token"
    class="com.opensymphony.webwork.interceptor.TokenInterceptor"/>
<interceptor name="token-session"
    class="com.opensymphony.webwork.interceptor.TokenSessionStoreInterceptor"/>
```

token：核对当前 Action 请求（request）的有效标识，防止重复提交 Action 请求(request)。

token-session：功能同上，但是当提交无效的 Action 请求标识时，它会将请求数据保存到 session 中。

我们在页面中提交使用 token 标签，

```
<%@ taglib prefix="ww" uri="webwork" %>
<ww:i18n name="'messages'">
<HTML>
…
…
<ww:token name="simpleInputToken"/>
…
</HTML>
</ww:i18n>
```

并在 xwork.xml 中添加黑体部分内容。

```
<!DOCTYPE xwork
    PUBLIC "-//OpenSymphony Group//XWork 1.0//EN"
    "http://www.opensymphony.com/xwork/xwork-1.0.dtd">
<xwork>
```

```
<include file="webwork-default.xml"/>
<package name="default" extends="webwork-default">

<default-interceptor-ref name="defaultStack"/>
        <action name="userreg" class="com.doone.module.action.UserregAction">
            <result name="success" type="redirect">
                <param name="location">/moduler/result.jsp</param>
            </result>
            ……
            <result name="invalid.token"   type="dispatcher">
                 <param name="location">/error/dupsubmit.jsp</param>
            </result>
            <interceptor-ref name="defaultStack"/>
            <interceptor-ref name="validationWorkflowStack" />
            <interceptor-ref name="token"/>
     </action>
</package>
</xwork>
```

2.1.4 验证

经常需要在客户端和服务器两边都进行验证。虽然客户端的验证处理一般比服务器端验证简单，但是它提供的是层次较高的检查，比如一个表单字段是否为空等等。服务器端验证经常更为复杂。这两种验证都适用于应用系统，但是我们不鼓励系统中只实现客户端验证，不能仅仅依靠客户端验证一个重要的原因是：客户端运行的脚本语言是可以在浏览器中配置的，所以用户可以在任何时候禁用脚本语言。

对验证策略的细节进行考察超出了本文范围。但是我们仍然想提及这些问题，因为在设计系统时不可避免地要考虑它们；我们也同时希望你参阅其它文献，进一步深研验证问题。

2.1.4.1 在客户端验证

输入验证在客户端进行。通常，这需要在客户端视图中加入脚本代码，比如

JavaScript，正如上文所描述的，客户端验证是对于服务器端验证的一个有益补充，但不能仅仅依靠客户端验证。

2.1.4.2 在服务器端验证

输入验证在服务器端进行。服务器端验证有几种常见的策略。其中包含基于表单的验证和基于抽象类型的验证。

基于表单的验证

基于表单的验证策略，会强制应用系统加入大量的方法，用以验证每个表单提交的各种状态，通常，这些方法在处理逻辑部分有很多重合之处，所以降低了系统重用度和模块化程度。对于提交的每个 WEB 表单，都要一个专门的验证方法，所以并没有代码对必填字段或数值字段之类的共通内容进行集中处理。这种情况下，虽然很多不同的表单都会有一个必填字段，但是每一次这种字段都要单独处理，因此也就在应用程序的很多地方造成了冗余。这种策略相对来说容易实现，也比较高效，但应用系统越大，它造成的重复代码就越多。

要想做出一个更为灵活、更可重用、更可维护的解决方案，就应该在另一个抽象层次上考虑模型数据，这也就是下面“基于抽象模型的验证”的处理思路。

基于表单验证的例子

```
public Vector validate()
{
   Vector errorCollection=new Vector();
   if((firstname==null)||(firstname.trim().length()<1))
   errorCollection.addElement(“firstname required”);
   if((lastname==null)||( lastname.trim().length()<1))
   errorCollection.addElement(“lastname required”);
   return errorCollection;
}
```

基于抽象类型的验证

无论是客户端或是服务器端都可以使用这种策略，不过最好还是利用基于浏览器或“瘦客户端”的环境，在服务器端实现这个策略。

从状态中抽象出类型和限制信息，放入到一个通用的框架中。这也就区分了模型的验证与应用这些模型的业务逻辑，从而降低了二者的偶合。

在验证模型时，进行的工作就是在模型状态与元数据、限制信息之间做比较。关于特定模型的元素和限制信息通常保存在某个简单的存储介质(比如一个属性文件中或 XML 文件)中。这种做法的一个优势就是系统能够更为通用，因为数据类型和限制信息都被从应用逻辑中提取出来了。

在 WebWork 提供了这种优质的策略。上文验证可以如下文中表单出来。

在 validators.xml 中包含以下信息：

```
<validators>
……
<validator name="required"
 class="com.opensymphony.xwork.validator.validators.RequiredFieldValidator"/>
……
</validators>
```

在 Action 提交时，例如 **UserregAction**.java 的同级目录下创建文件一个名字为 **UserregAction**-validation.xml，这里需要注意 Action 的名称相同的。

```
<?xml version="1.0" encoding="GB2312"?>
<!DOCTYPE validators
    PUBLIC "-//OpenSymphony Group//XWork Validator 1.0//EN"
    "http://www.opensymphony.com/xwork/xwork-validator-1.0.dtd">
<validators>
    <field name=" user.firstname">
        <field-validator type="required">
            <message> firstname required</message>
        </field-validator>
    </field>
    <field name=" user.lastname">
        <field-validator type="required">
            <message> lastname required</message>
        </field-validator>
    </field>
</validators>
```

在前文中的 xwork.xml 配置中，我们填加一个 input 的 result 就可以进行验证了。

```
<!DOCTYPE xwork
      PUBLIC "-//OpenSymphony Group//XWork 1.0//EN"
      "http://www.opensymphony.com/xwork/xwork-1.0.dtd">
<xwork>
<include file="webwork-default.xml"/>
<package name="default" extends="webwork-default">

<default-interceptor-ref name="defaultStack"/>
            <action name="counter" class="com.doone.module.action.UserregAction">
                  <result name="success" type="redirect">
                        <param name="location">/moduler/result.jsp</param>
                  </result>
                  ……
                  <result name="input" type=" dispatcher ">
                        <param name="location">/useradd.jsp</param>
                  </result>
                  ……
                  <interceptor-ref name="defaultStack"/>
                  <interceptor-ref name="validationWorkflowStack" />
       </action>
</package>
</xwork>
```

HTML 中包含输出错误信息的内容和提交的表单内容即可。

```
<%@ taglib prefix="ww" uri="webwork" %>
<ww:i18n name="'messages'">
<HTML>
<ww:if test="hasErrors()">
      <span class="errorMessage">
      <b>Error:</b><br>
            <ww:iterator value="fieldErrors">
              <ww:iterator value="value">
                  <img src="/images/iconWarning.gif"   alt="warning" class="icon" />
                    <ww:property/><br />
              </ww:iterator>
        </ww:iterator>
```

```
        </span>
  </ww:if>
…
…
<ww:form action="userreg">
<input type="text" name="user.firstname ">
<input type="text" name="user.lastname ">
…
</ww:form>
</HTML>
</ww:i18n>
```

同样，你可以对验证类进行扩展，例如我们编写一个能够对正则表达式进行验证的通用验证器(StringRegexValidator.java)：

```
public class StringRegexValidator extends FieldValidatorSupport   {
        private String regex="";
        private boolean doTrim = true;
        public void validate( Object object )throws ValidationException{

          String fieldName = getFieldName();
          String value = (String)this.getFieldValue( fieldName, object );
          if (doTrim) {
                  value = value.trim();
                }
          Pattern pattern = Pattern.compile( regex );
          Matcher matcher = pattern.matcher( value );
          if ( !matcher.find() ){
            addFieldError( fieldName, object );
          }
          }

        public String getRegex() {
          return regex;
        }

        public void setRegex(String regex) {
          this.regex = regex;
```

```
        }

        public boolean isTrim() {
          return doTrim;
        }

        public void setTrim(boolean trim) {
          doTrim = trim;
        }
}
```

那么我们就可以配置 validators.xml 如下：

```
<validators>
……
<validator name="stringregex"
    class="com.doone.module.validator.StringRegexValidator"/>
……
</validators>
```

修改 UserregAction-validation.xml 文件内容

```
<?xml version="1.0" encoding="GB2312"?>
<!DOCTYPE validators
      PUBLIC "-//OpenSymphony Group//XWork Validator 1.0//EN"
      "http://www.opensymphony.com/xwork/xwork-validator-1.0.dtd">

<validators>
……
       <field name="model.username">
           <field-validator type="stringregex">
                <param name="regex">^([a-zA-Z]{5,13})$</param>
                <message>你必须输入一个用户名,用户名只能由字母、数字及下划线组成且
长度必须在 5-13 之间。
           </message>
           </field-validator>
         </field>
……
</validators>
```

显然这种 HTML 表单的值和具体的验证类型之间建立映射更加层次分明，效率更高一些。

2.2 表现层不佳实践

所谓的不佳实践，也就是与模式推荐的思路冲突一些不太理想的解决方案。当我们记录下来模式和最佳实践的时候，自然而然地也就放弃了这些不太理想的方法。

在本文的这一部分，我们着重讨论了(我们所认为的)表现层不佳实践。

在以下每节中都简单地描述了一种不佳实践，然后提供了大量的参照内容，包含设计问题、重构、模式等等，这样就为解决特定的问题提供了进一步的信息和更好的方案。但是，对于每个不佳实践我们并不做进一步的深入讨论，而是给出了提纲挈领的描述，作为此后深入考察的出发点。

“问题概述”部分提供了对一个不太理想的解决方案的简述，参考解决方案部分则包括：

模式提供了这类应用场景的模式信息以及实现代价；

设计考虑提供了相关的设计细节；

重构描述了从一个不太理想的解决方案(所谓不佳实践)到一个理想的解决方案、一种最佳实践或一个模式的过程。

2.2.1 视图中包含业务控制代码

问题概述

每个 JSP 视图的中都含有大量的业务代码，这些代码导致系统难以维护，因为一点点的修改就可能引起多处的修改，甚至会延伸出更多的条件判断，最后控制的代码本身可能都超越了业务本身，业务条理变得难以理解。

还有把数据传输对象(DTO)当 Java Scriptlet 使用，首先是把 HTML 标记和 scriptlet 代码混用，直接迭代 DTO 输出数据，而事实上，这个工作应该交给

Webwork 标签去完成，DTO 应该体现更高层次的抽象。

参照解决方案

合并控制代码，引入 Webwork 标签库，视图彻底和业务层分离开来，视图中仅仅控制视图显示本身等问题而不在包含业务控制。

2.2.2 表现层数据结构暴露在业务层

问题概述

表现层的数据结构，比如说 HttpServletRequest，应该只限于表现层。把这一类细节暴露给业务层，都会增加这些层次之间的偶合，从而急剧降低服务的可重用度。如果业务层方法的参数表中有一个 HttpServletRequest 类型的输入参数，那么这个服务的任何客户端(甚至不是在 Web 环境中的客户端)也都要把它们的请求状态包装成 HttpServletRequest 对象。而且，在这种情况下，业务层的服务需要懂得如何跟表现层专用的数据结构通信，也就添加了业务层代码的复杂度，增加了各个层次之间的耦合。

参照解决方案

不要让业务层使用哪些原本专门属于表现层的数据结构，而应该把相关的状态复制到通用的数据结构中去，让两个层次共享这些通用的数据结构。或者，还可以把相关状态从表现层专用的数据结构中抽取出来，作为独立的参数在各层次之间传递。通常会采用数据传输对象(DTO)进行传递，DTO 事实上是一个典型的 JavaBean，它和数据库中的关系一一对应，也就是说表现层应该将自己的数据结构的数据复制到 DTO 中，从而降低耦合。

2.2.3 允许重复提交表单

问题概述

一个桌面应用程序可以控制客户端的用户导航，而基于浏览器的客户端环境缺乏这种控制，这是这种环境的一个局限。比如，用户可能提交了一个订单表单，这就产生了一个事务操作，但是，在收取确认页面之后，用户点击“后退”按钮，

同一份表单还会被重新提交。

参照解决方案

重构：引入同步器令牌。见前面 2.1.3 节部分。

2.2.4 敏感资源暴露在客户端

问题概述

安全性是企业环境开发的最重要的问题之一。如果并不需要让客户端直接访问某些信息，则这种信息必须要受到保护。如果某些特定的配置文件、属性文件、JSP 页面文件和类文件没有得到妥贴的保护，那么客户端就可能不经意地或恶意地获取敏感信息。

参考解决方案

重构：见前面节 2.1.2“控制客户端访问”部分

3 业务层设计考虑和不佳实践

3.1 业务层设计

在使用本文介绍的业务层和集成层模式时也需要了解一些相关的设计问题——本章节就讨论这些问题。这里包含的很多主题，也会影响到系统的很多问题。

当基于 J2EE 模式目录实现应用系统时，肯定要考虑这些问题，本章的讨论就是从这种考虑的角度，简要描述每个问题的。

3.1.1 使用 session bean

按照技术规范，session bean 是一种具备以下特征的分布式业务组件。

- 每个 session bean 专门服务于一个客户端或用户。
- 每个 session bean 的生命周期等于客户端的会话时间。
- session bean 在服务器崩溃后不能存活
- session bean 不是一个持久化对象。
- session bean 会超时(time out)
- session bean 可以涉及业务
- session bean 既可以用来构造客户端和业务层组件之间的有状态对话模型，也可以用来构造两者间的无状态对话模型。

所谓有状态的 session bean 保存了特定客户端的状态，它就不可能同时被多个客户端共享，而无状态的 session bean 可以在不同的客户端中传递。理论上有状态的 session bean 比无状态的 session bean 需要更多的资源负载。

Webwork 提供了 session bean 的容器，在使用应用中会大量的使用 session bean，在例如一些系统参数类的，基本上已经固化的值，可以采用无状态的

session bean 来保存，而对用户个人信息部分可能会更多的采用有状态的 session bean。但无论是有状态的还是无状态的 session bean，我们的模式采用了相同的方法来保存。由 EJB 规范的说法我们仍然可以保留。

下面是一个 Webwork 中继承过来通用的类，能够方便我们使用 session：

```
import com.opensymphony.xwork.ActionContext;
import com.opensymphony.xwork.ActionSupport;

public class AbstractAction extends ActionSupport{
        protected Object get(String name) {
            return ActionContext.getContext().getSession().get(name);
        }

        @SuppressWarnings("unchecked")
        protected void set(String name, Object value) {
            ActionContext.getContext().getSession().put(name, value);
        }
}
```

在调用时直接从 AbstractAction 继承过来后进行 set 方法就可以了。

```
import com.doone.transsms.util.TransEnum;
import com.opensymphony.xwork.ModelDriven;

public class UserregAction extends AbstractAction implements ModelDriven {
    private final static String REG_FAIL="regfail";
    TfUser loginInfo = new TfUser();
    public String execute() throws Exception {
    set("loginUser", loginInfo);
    }
    public Object getModel() {
        return loginInfo;
    }
}
```

3.1.2 使用数据传输对象(DTO)

在前面的模式目录中多次提及 DTO，并在 Webwork、Spring 和 iBATIS 中数据传递。如果你不能够理解 DTO 概念，那么你可以简单的理解，它是一个 JavaBean，但是事实上我们一直提倡层次分明的模式，那么 DTO 就起了一个非常重要的作用，它充当各个层次中的耦合的角色，将不同层次的模式贯穿起来，形成为一个非常稳固的应用系统。

上文 UserInfo.java 如例子中

```
public class UserInfo{
    //fields
    private java.lang.String firstname;
    private java.lang.String lastname;
    private java.lang.String role;
    public java.lang.String getFirstname(){
        return firstname;
    }
    public void setFirstname(java.lang.String firstname){
        this. firstname = firstname;
    }
    public java.lang.String getLastname(){
        return lastname;
    }
    public void setLastname(java.lang.String lastname){
        this. lastname = lastname;
    }
    public java.lang.String getRole(){
        return role;
    }
    public void setRole(java.lang.String role){
        this. role = role;
    }
}
```

层次分明的模式中，我们对数据的访问，就应该基于 DTO 进行，表现层将 HTML 特性的数据结构转换为 DTO，业务层则直接面向这个 DTO 对象进行业务

逻辑上的控制，而持久层负责将DTO和数据库进行映射关联。对于我们Webwork负责的表现层能够自动完成这种转换，而持久层 iBATIS 能够完成 O/R 映射将 DTO 存入数据库。

3.2 业务层不佳实践

3.2.1 把 HttpServletRequest 直接当做 DTO

问题概述

常见的一种做法就是直接把 HttpServletRequest 当做 DTO 来使用，直接去获取页面传输过来的数据进行简单的拼写 SQL 直接调用 JDBC 或类似数据库封装的方法入库。而如果在业务中传递时，则把该数据的关键值存入 session，然后在新的业务动作中重新获取对象的状态。这显然会使数据库负担加重，同时，也很难体现出面向对象设计的优势来。

参照解决方案

见 3.1.1“使用 session bean”中的 getModel 方法。

3.2.2 把 ResultSet 直接当做 DTO

问题概述

还有一种做法，就是通过 JDBC 或类似数据库封装的方法，将返回的 ResultSet 通过 rs.getString(1)等方式获取数据，直接当做业务参数进行业务逻辑运算。这样 SQL“硬编码”似乎难以避免，直接影响了系统的可扩展性。这种细微粒度的业务功能几乎谈不上重用，维护量却异常高，一个业务功能的调整都可能导致重新编码，这对一个稍微庞大的应用系统来说，可能是致命的。

参考解决方案

参考 iBATIS 的 DAO

4 J2EE 重构

4.1 表现层的重构

本章节介绍用于表现层的重构。

4.1.1 引入 WebWork 标签

WebWork JAR 文件(webwork.jar)包含了一个标签库定义(TLD)文件，分发了 WebWork 提供的所有的标签。我们截取中间的一个小片断，我们推荐你有时间可以阅读完整的 TLD，了解每个具体的标签的用处。

```
<tag>
        <name>bean</name>
        <tagclass>com.opensymphony.webwork.views.jsp.BeanTag</tagclass>
        <bodycontent>JSP</bodycontent>
        <info>
            Create a JavaBean and instantiate its properties. It
            is then placed in the ActionContext for later use.
        </info>
        <attribute>
            <name>name</name>
            <required>true</required>
            <rtexprvalue>true</rtexprvalue>
        </attribute>
        <attribute>
            <name>id</name>
            <required>false</required>
            <rtexprvalue>true</rtexprvalue>
        </attribute>
</tag>
```

这部分标签包含了两个属性 name 和 id，name 属性是必须的，为了能够使用这个标签，你要在 web 应用软件中通过编辑 web.xml 中注册它，以下就是一个添加新元素的 web.xml：

```
<taglib>
        <taglib-uri>webwork</taglib-uri>
        <taglib-location>/WEB-INF/webwork.tld</taglib-location>
</taglib>
```

然后你要使用所有的 WebWork 标签，则在你的 JSP 页面中，添加一个标准标签库的开头<%@taglib prefix=”ww” uri=”webwork”%>。例如：

```
<%@taglib prefix=”ww” uri=”webwork”%>
<html>
  <head>
      <title>A simple page</title>
  </head>
  <body>
      Hello,<ww:property value=”name”/>!
  </body>
</html>
```

在这个页面中我们可以观察到，事实上<ww:property value=”name”/>能够将 bean 中的 name 字段显示出来，类似<%=bean.getName()%>。而在页面中却找不到 bean 这个声明，Webwork 的 bean 标签中包含了对这个 bean 的识别，并认为 bean 是这个页面的组成部分。这样给我们带来的便利就是页面已经脱离了业务本身而仅仅成为展现的内容，这就回到了 HTML 真正应该做的事情。

那么对提交页面的重构也相当方便：

```
<%@ taglib prefix="ww" uri="webwork" %>
<%@ page contentType="text/html; charset=GBK" language="java"
 errorPage="/all_error.jsp" %>
<%request.setCharacterEncoding("GBK");%>
<ww:i18n name="'messages'">
<html>
<head>
    <link rel="stylesheet" href="/styles/style.css" type="text/css">
    </head>
```

```
<body>
 …..
<ww:form   action="/adduser.action">
<table cellpadding="1" cellspacing="0" align="center" border="1" >
<tr>
<td nowrap>username</td>
<td nowrap><ww:textfield name="user.username" size="20"/></td>
</tr>
<tr>
<td nowrap>password</td>
<td nowrap><ww:password name="user.password" size="20"/></td>
</tr>
<tr>
<td nowrap>AutoLogin</td>
<td nowrap><ww:checkbox name="user.autologin" fieldValue=”true”/></td>
</tr>
<tr>
<td nowrap>Role</td>
<td nowrap>
<ww:select label=”Role” name="user.role"
                list=”#{1:‘manager’,2:’normal’}”
                headerKey=”-1” headerValue=”Select a role”
                emptyOption=”true”/>
</td>
</tr>
<tr>
<td rowspan=2>
<input type="hidden" name="actionEvent" value="multiSubmit">
<input type="submit" name="submit" value="Submit"/></td>
</tr>
</table>
</ww:form>
</body>
</html>
</ww:i18n>
```

4.1.2 引入 WebWork Action

相对于 servlet，我们采用 WebWork Action 来重构提交处理的方法，采用 WebWork Action 非常明显的优势就是直接面向对象，省去了 servlet 中大量获取参数的值方法，例如 request.getParameter("username")等，这种层次上的分离给我们的带来的好处毋庸置疑。

```
public class AdduserAction implements Action {
    private final static String ADDUSER_FAIL="adduserfail";
    TfUser user = new TfUser();
    public String execute() throws Exception {
    ……
    }
    public void setUser(TfUser user){
            this.user = user;
    }
    public String getUser (){
            return this.user;
    }
}
```

4.1.3 引入数据访问对象(DTO)

上文中这里的 user 事实上就是一个 DTO，它的定义如下，可以看出它就是数据库中的字段的 Java 类的映射，getters/setters 的方法是标准的 JavaBean 写法。正是因为这些规范的方法，使得 DTO 能够在整个应用系统中自由穿梭，甚至在持久层的缓存中也表现的非常的优异。

```
public class TfUser{
   //fields
   private java.lang.String username;
   private java.lang.String password;
   private java.lang.Integer autologin;
   private java.lang.Integer role;
   public java.lang.String getUsername (){
       return username;
```

```
    }
    public void setUsername (java.lang.String username){
        this. firstname = firstname;
    }
    public java.lang.String getPassword (){
        return password;
    }
    public void setPassword(java.lang.String password){
        this. password = password;
    }
    public java.lang.Integer getAutologin (){
        return autologin;
    }
    public void setAutologin (java.lang.Integer autologin){
        this. autologin = autologin;
    }
    public java.lang.Integer getRole(){
        return role;
    }
    public void setRole(java.lang.Integer role){
        this.role = role;
    }
}
```

4.2 业务层的重构

4.2.1 引入数据访问对象(DTO)

正如上文所描述的，DTO 是一个信息的标准载体，那么在业务层就应该能够对这个载体加以重用，可以看出 DTO 就是一个重用的基本粒度，它可以组成各种各样的复合元素。在业务层中，我们正是基于这个粒度去扩展更多的业务逻辑中的数据传输。

4.2.2 引入数据访问对象(DAO)

DAO 是一系列的数据访问对象，这个对象能够对数据库进行访问，完成 DTO 和数据库之间的映射。我们抽象地认为 DAO 必须包含四个基本的访问元素，即添加、删除、修改和查询，除此之外的访问都是由这四种基本元素复合，因此在模式中，我们都会去编写这基于数据库主键的这四个方法。

```
import org.springframework.dao.DataAccessException;
public interface TfUserDAO{
    public java.util.List findTfCount(com.doone.module.dto. TfUser tfuser)
                        throws DataAccessException;
    public void insertTfCount(com.doone. module.dto. TfUser tfuser)
                        throws DataAccessException;
    public void updateTfCount(com.doone. module.dto. TfUser tfuser)
                        throws DataAccessException;
    public void deleteTfCount(com.doone. module.dto. TfUser tfuser)
                        throws DataAccessException;
}
```

在例子中我们可以看到 DAO 事实上就是将一个 DTO 作为操作对象的对数据访问的方案的集合。这么在下面的例子中我们可以去发现具体的实现方法，它采用了 iBATIS 的访问方法：

```
import org.springframework.dao.DataAccessException;
import org.springframework.orm.ibatis.support.SqlMapClientDaoSupport;

public class TfUserDAOImpl extends SqlMapClientDaoSupport
  implements com.doone.counter.dao.TfUserDAO
  {
    public java.util.List findTfCount(com.doone.counter.dto.TfUser tfuser)
        throws DataAccessException {
        return (java.util.List)getSqlMapClientTemplate()
            .queryForList("findTfUserDao", tfuser);
    }
    public void insertTfCount(com.doone.counter.dto.TfUser tfuser)
        throws DataAccessException {
        getSqlMapClientTemplate().update("insertTfUserDao", tfuser);
```

```
    }
    public void updateTfUser(com.doone.counter.dto.TfUser tfuser)
        throws DataAccessException {
        getSqlMapClientTemplate().update("updateTfUserDao", tfuser);
    }
    public void deleteTfUsercom.doone.counter.dto.TfUser tfuser)
        throws DataAccessException {
        getSqlMapClientTemplate().update("deleteTfUserDao", tfuser);
    }
}
```

在这方法中我们还是没有看见具体的 SQL 语句，模式中，我们要求 DAO 的粒度的重用，因此 DAO 提供了一系列的方法应该属于可以配置的，因此 DAO 中的 SQL 被剥离出来，放在具体的配置文件中，这对将来的业务调整和维护提供便利。这是一个远大的目标，能够应付日益增长的业务需求，同时 DAO 的逻辑清晰，代码“似曾相识”，就可以使我们的开发者采用机械化编程，因此新东网就能够提供类似代码的生成工具，从而提高 30%左右的生成效率。

下面的例子中，我们可以看见关于 iBATIS 上下文的定义：

```
<?xml version="1.0" encoding="gb2312" ?>
<sqlMap namespace="TfUser">
    <select id="findTfUserDao"
        resultClass="com.doone.module.dto.TfUser">
        <![CDATA[
        SELECT USERNAME, PASSWORD,AUTOLOGIN,ROLE from TF_USER
        WHERE USERNAME=#username#
        ]]>
    </select>
    <insert id="insertTfUserDao"
        parameterClass=" com.doone.module.dto.TfUser ">
        <![CDATA[
        INSERT INTO TF_USER( USERNAME, PASSWORD, AUTOLOGIN, ROLE)
        VALUES(#username#, #password#, #autologin#, #role#)
        ]]>
    </insert>
    <update id="updateTfUserDao"
        parameterClass=" com.doone.module.dto.TfUser ">
```

```
        <![CDATA[
        UPDATE TF_USER
        SET USERNAME=#username#, PASSWORD=#password#, ROLE=#role#
        WHERE USERNAME=#username#
        ]]>
    </update>
    <delete id="deleteTfUserDao"
        parameterClass=" com.doone.module.dto.TfUser ">
        <![CDATA[
        DELETE FROM TF_USER WHERE USERNAME=#username#
        ]]>
    </delete>
</sqlMap>
```

DAO 就是通过文件中的这个 ID 和数据库形成映射的。通过修改这个 XML 配置文件就能够影响业务的逻辑，而 DAO 本身仅仅是方法调用。当然，这个粒度只是细微的调整，真正的业务逻辑的组织应该在业务门面(FACADE)中完成，在下节我们描述门面的重构。

4.2.3 引入业务门面(FACADE)

业务门面是 DAO 的基本业务元素的根据业务逻辑的需要进行的组合，它可能包含一个或一个以上的 DAO，共同完成一个业务事务的要求。通过调用门面中的方法，就能够完成一个业务事务，这非常类似于业务工厂，业务工厂通常被实例化成一个单一的实例，并重复调用。门面就是基于这样一个动机，但思路扩展的更远一些，这里的业务门面涉及到事务的问题。业务工厂通常也包含事务的概念，但是一旦确定需要事务流程，就是在相应的方法中声明事务的开始和结束，并固化到程序中去，而且涉及到使用 DAO 时，代码可能就不够很优雅，开发者需要关心业务的事务在什么时候发生，并加以控制，而这部分的工作本来应该是设计师需要考虑的问题。我们组织成业务门面时，业务门面的事务也应该通过配置完成。尽管公司目前开发者和设计师往往是同一个人，但从长远角度来讲应该把他们的角色分离开来，即便师同一个人，也要意思到自己是两个不同的角色，而不应该将两个角色等同于一个人。

业务门面的组织采用 Spring 的这个微架构进行的，它的声明方式类似 DTO，所不同是 DTO 中的属性是一个数据库相应的字段，而 Facade 的属性是一个或多个 DAO，所以 Façade 是 DAO 更高一个层次的封装，封装的目的很明确，就是展现 Spring 最为闪亮的事务特性。在发布 Façade 的时候，我们可以通过配置是指定 Façade 的某个方法就是一个事务，产生的数据操作，要么全部提交，要么全部回滚。

在接口类中我们声明一个业务门面的基本结构：

```
public interface UserFacade{
    public com.doone.module.dao.TfUserDAO    getTfUserDAO();
    public void setTfUserDAO(com.doone.counter.dao.TfUserDAO tfuserdao);
    public com.doone.module.dao.TfSubjectDAO    getTfSubjectDAO();
    public void setTfSubjectDAO(com.doone.counter.dao.TfSubjectDAO tfsubjectdao);
    public java.util.List findTfClient(com.doone.module.dto.TfUser tfuser);
    public void addReguser(com.doone.module.dto.TfUser tfuser,
                          com.doone.module.dto.TfSubject tfsubject);
}
```

在实现类中声明两个类型为 DAO 的私有变量，作为 Facade 类的属性，并在 addReguesr()方法使用这个两个 DAO，做两个数据插入的操作。

```
public class UserFacadeImpl implements com.doone.module.domain.UserFacade{
     private com.doone.module.dao.TfUserDAO    tfuserdao;
     private com.doone.module.dao.TfSubjectDAO    tfsubjectdao;
     public TfUserDAO getTfUserDAO () {
          return tfuserdao;
     }
     public void setTfUserDAO (TfUserDAO tfuserdao) {
          this.tfuserdao = tfuserdao;
     }
     public TfSubjectDAO getTfSubjectDAO () {
          return tfsubjectdao;
     }
     public void setTfSubjectDAO (TfSubjectDAO tfuserdao) {
          this.tfsubjectdao = tfsubjectdao;
     }
     public void addReguser (TfUser tfuser,TfSubject tfsubject)
```

```
            throws DataAccessException {
            this. tfuserdao. insertTfUser(tfuser);
            this. tfsubjectdao. insertTfSubject(tfsubject);
        }
}
```

以下的上下文中，为这个门面配置一个事务，既凡是门面中 add 开头的方法均被 Spring 认为一个事务，这样 insertTfUser()和 insertTfSubject()有任何一个数据操作错误均会被回滚，这就给应用系统带来的非常好的灵活性。这种基于配置的事务控制，显然把事务从代码层脱离出来，维护性得到很大的提高。

```
……
<bean id="baseTransactionProxy"
   class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"
   abstract="true">
   <property name="transactionManager" ref="transactionManager" />
   <property name="transactionAttributes">
   <props>
      <prop key="add*">PROPAGATION_REQUIRED</prop>
      <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
   </props>
   </property>
</bean>
<bean id="tfuserDAO" class="com.doone.module.dao.ibatis.TfUserDAOImpl">
   <property name="sqlMapClient">
      <ref local="sqlMapClient" />
   </property>
</bean>
<bean id="tfsubjectDAO" class="com.doone.counter.dao.ibatis.TfSubjectDAOImpl">
   <property name="sqlMapClient">
      <ref local="sqlMapClient" />
   </property>
</bean>
<bean id="tfuserDAOProxy" parent="baseTransactionProxy">
   <property name="target">
      <bean class="com.doone.counter.domain.logic.TfUserFacadeImpl">
         <property name="tfuserdao" ref=" tfuserDAO "></property>
         <property name="tfsubjectdao" ref=" tfsubjectDAO "></property>
```

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

5 J2EE 模式概览

5.1 什么是模式

在第一章中，我们讨论了专家们对模式各种各样的定义。我们也讨论了与模式相关的一些问题，比如使用模式的益处。这里，我们要在 J2EE 模式目录语境中重新考察这些讨论。

比如第一章所述的，一些专家把模式定义为：在一种上下文中，一类问题的一种可重复使用的解决方案。

这里的术语－－上下文、问题、解决方案－－需要进行一些解释。首先，什么叫"上下文"？上下文就是一种环境、一些周边事务、一种情况，或者说是某物出于其中的一些相关条件。第二，什么叫问题？问题就是一种未经解决的疑问，也就是某些需要研究、解决的东西。通常，问题受到它所出现的上下文的约束。最后，所谓"解决方案"也就是，在特定上下文中，有助于解决问题的那个答案。

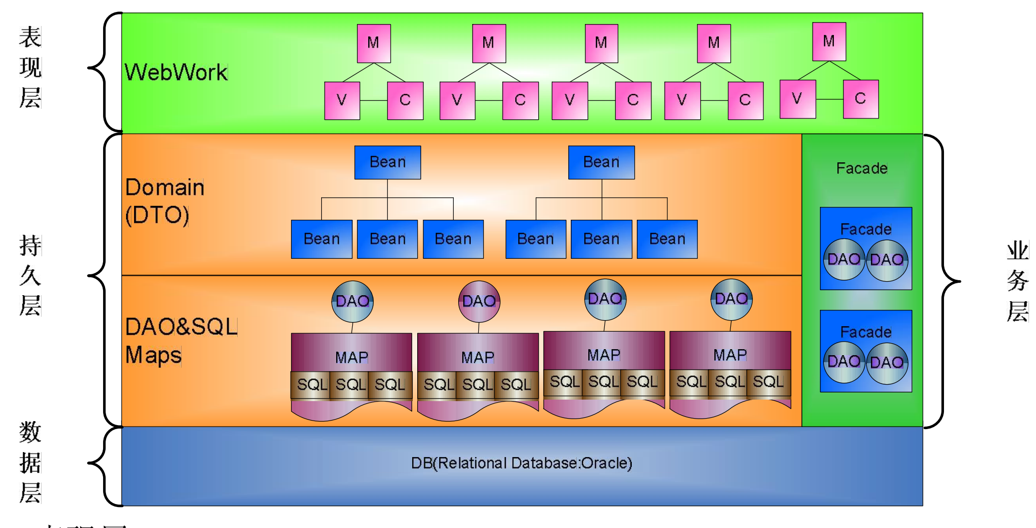
那么，如果我们有了在某种上下文对某个问题的解决方案，这就是一个模式了吗？不一定。因为模式的定义里还有一个特征，那就是"重复使用"。也就是说，只有能够反复应用，这个模式才能称得上有用。这就齐了吗？也许还不是。正如你所见，虽然模式的概念相当简单，但是实际上给这个术语下一个定义还是很复杂的事情。

我们给出的一些参考文献，你可以自己深研模式的历史，了解其他领域的模式。但是在我们的模式目录中，模式是按照以下主要特征描述的：问题，解决方案，以及其他重要方面的内容，如约束和效果。

5.2 分层思路

既然本目录描述了一些有助于构建 J2EE 应用程序的模式，既然 J2EE 平台

是多层系统，所有我们也就要从多个层面考虑整个系统。所谓“层”，也就是系统中按照不同考虑方案划分的逻辑区间。系统中每个层都负有独立的职责。我们把层之间的区别看成是一种逻辑区分。每个层与相邻的层之间都存在松耦合。整个系统就可以表现为多个层堆叠而成。



表现层

表现层封装了服务于访问系统的客户端的所有的表现逻辑。表现层拦截了客户端的请求，提供了单一的登录入口，构造了会话的管理，控制了对业务服务的访问，构造了响应，并把这些响应传递给客户端。我们的模式约束，表现层不得参与业务服务构造。

业务层

该层提供了应用客户端需要得各种业务服务。该层中包括业务数据和业务逻辑。通常，应用系统中得大多数业务处理都集中在该层。但是，也可能出现这样的情况：由于原本遗留的系统，有一些业务处理出现在资源层。对于实现业务层的业务对象，模式约束，业务层不得参与表现层的控制，也不要包含对数据库的访问。

持久层

这一层负责与外部资源、外部系统通信。只要业务对象需要处于资源层的数据和服务，那么业务层和集成层之间就存在耦合。这一层中的组件还可能包含其他厂商专用有的中间件。

5.3 J2EE 模式

5.3.1 表现层模式

模式名称	简要说明
Action 对象(Action Object)	实现了业务操作
拦截器(Intercepter)	用于请求的预处理和后处理
Results 对象(Result Object)	业务操作之后的连续动作
标签(Tag)	构造视图的助手
Context 对象(Context Object)	表现层上下文的功能组合、配置和索引功能

5.3.2 业务层模式

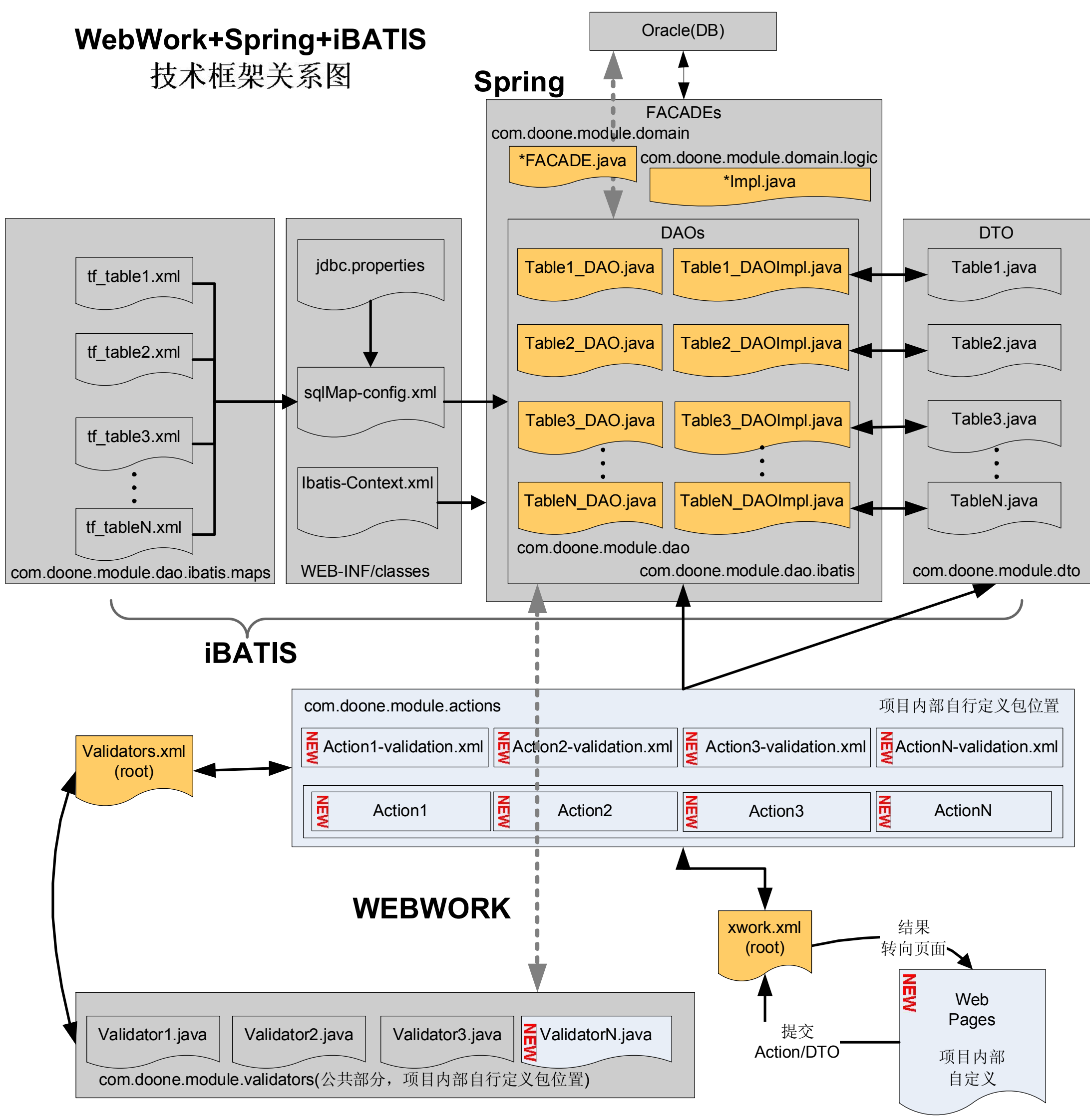
模式名称	简要说明
数据传输对象(Data Transfer Object)	在各个层之间传输数据
业务门面(Business Facade)	封装了业务层的组件，把粗粒度的服务暴露给客户端
Context 对象(Context Object)	业务层上下文的功能组合、配置和索引功能

5.3.3 持久层模式

模式名称	简要说明
数据传输对象(Data Transfer Object)	在各个层之间传输数据
数据访问对象(Data Access Object)	抽象并封装了对持久化存储的访问
Context 对象(Context Object)	上下文的功能组合、配置和索引功能

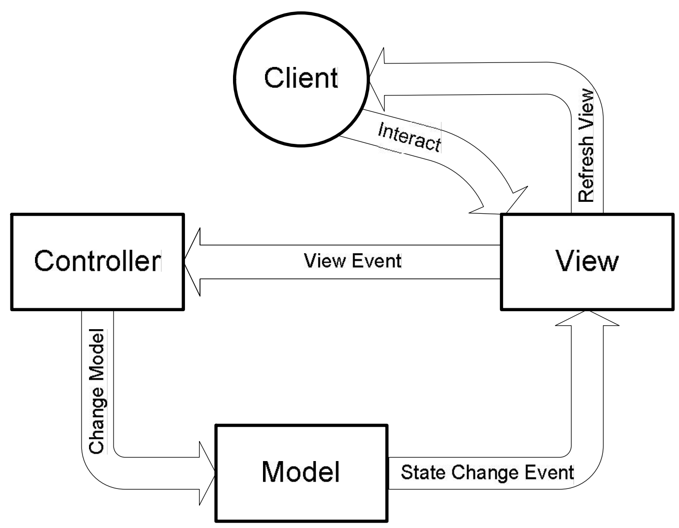
5.4 J2EE 模式关系

最近，一些架构师和设计师表达了这样的一种关注：对于怎样组合使用模式、构成一种大型的解决方案。开发人员们似乎缺乏理解。在这里我们采用一种层次的图示来表达这种模式关系。关系图中包含了具体的包的路径和所处的层次微框架技术的耦合关系。

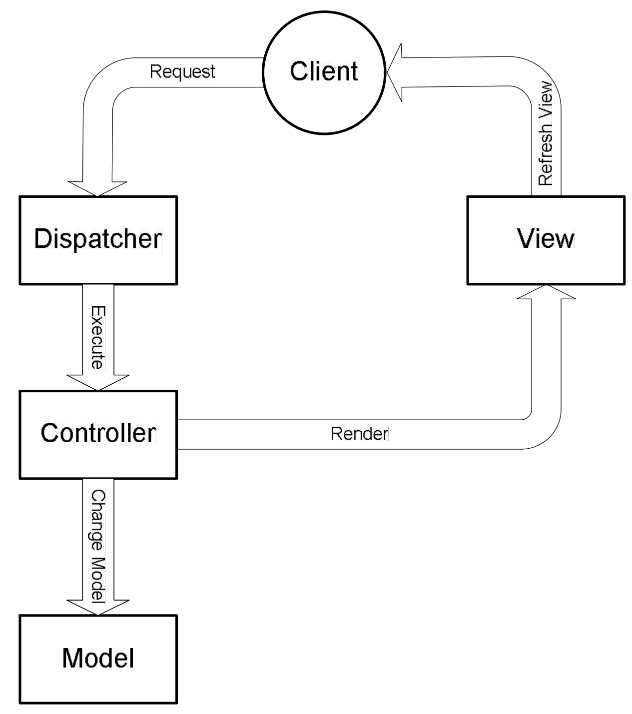


6 表现层模式

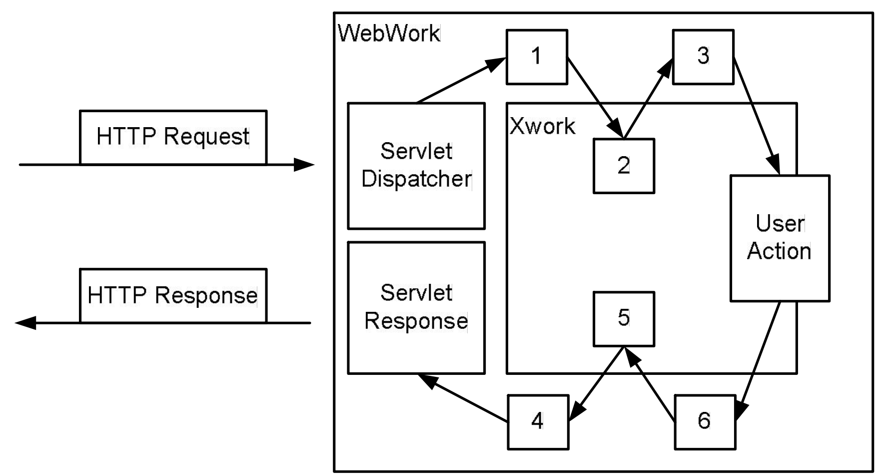
典型的 MVC 即将过时。如下图所示，客户端和视图进行交互，控制收到视图的事件去更新 Action 中的用户提供的数据，然后视图提供新的视图给客户端给用户。这种模式导致事件的控制保留在了视图中，也就是说，事实上视图控制了事件的流向，这就导致了视图功能强大，视图页面数量繁多，不能进行有效的配置，这在分层思路下显得非常不合时宜，因此我们将引进前端控制器的理论。



在新的模式中，我们应该将客户的请求拦截到前端控制器中，由前端控制器去判定执行相应的控制器中，并传递提交的数据，再控制器提供新的视图给客户。这种细微的模式调整，给模式带来了非常优越的优势。



下面的视图中可以看见，一个简单的提交，WebWork 对这个请求进行中断并进行了大量的逻辑上的流向判断，甚至进入了 Xwork 的层次的逻辑判断，层层分离，层层组合。本章正是基于这种新的模式进行讨论和实现的。



表现层的 WebWork 的书籍相对比较少，因此我们会对某些定义加以详细描述，篇幅相对会比较多一些。

6.1 使用 Actions

6.1.1 问题

传统的直接从 HttpServletRequest 开始继承，我们将采用大量的 request.getParameter()获取，即没有把提交过来的数据当做对象看待，这种基于字段集的方案给我们的编程带来了很大的麻烦和业务不清晰。下面就提供解决这类文档的方案。

6.1.2 解决方案

6.1.2.1 Action 接口

Action 仅仅需要实现 WebWork 的 com.opensymphony.xwork.Action 接口即可，而 Action 定义了一个方法：

```
public String execute() throws Exception
```

在此方法中，Action 必需返回执行结果，返回结果可以是任何一种字符串值，只要这个直接在 xwork.xml 中定义的值相同就可以了。当然你也可以指定另外一个方法做为 execute()的替换方法，从而提高灵活性。

```
public String saveCategory(){
    if(category==null){
      return INPUT;
    }
    categoryDAO.makePersistent(category);
    return SUCCESS;
}
```

而在 xwork.xml 中定义如下：

```
<action name=”saveCategory”
        class=”com.doone.module.actions.EditCategory”
        method=”saveCategory”>
        <interceptor-ref name=”crudStack”/>
```

```
        <result name="input">createCategory.jsp</result>
        <result name="success" type="redirect">dashboard.action</result>
</action>
```

6.1.2.2 ActionSupport 基本类

ActionSupport 提供了默认的很多可选的服务易于开发，如果你的 Action 从 ActionSupport 进行扩展，你可以实现以下功能，而不需要实现代码编写和实现。

- com.opensymphony.xwork.Validateable-提供 avlidate()方法，允许你的 Action 调用校验。
- com.opensymphony.xwork.ValidationAware-提供保存、检索 Action 和字段级的错误消息的方法。
- com.opensymphony.xwork.TextProvider-提供国际化的文本的方法。
- com.opensymphony.xwork.LocalProvider-提供 getLocale()方法，能够使用本地化的文本的方法。

6.1.2.3 理解基本校验

校验接口能够让 Action 自动校验，让我们看以下 CreateUser 的 Action 如何使用 validate()方法进行校验。如下显示原始的 execute()方法：

```
public String execute() throws Exception{
    //see if the name already exists
    TfUser existing=userDAO.findByUsername(this.user.getUsername());
    If(existing!=null){
      addFieldError("user.username","The user already exists");
      return INPUT;
    }
    userDAO.makePersistent(user);
    return SUCCESS;
}
```

这个方法事实上可以转化为两个方法：

```
public void validate(){
```

```
    TfUser existing=userDAO.findByUsername(this.user.getUsername());
    If(existing!=null){
      addFieldError("user.username","The user already exists");
     }
}
public String execute() throws Exception{
    userDAO.makePersistent(user);
    return SUCCESS;
}
```

ActionSupport 自动调用 validate()，并不需要返回 INPUT。ActionSupport 还做更多的工作，以至于能够采用配置的方式进行校验，请阅读 2.1.4 “验证”。

6.1.2.4 使用 ModelDriven Action

ModelDriven 接口函数提供了 getModel()方法：

```
public Object getModel()
```

你能够使用这个方法将页面的模型返回给 Action。模型的属性根据模型对象通过 getters/setters 来获取，你不能在 Action 去创建这个模型，这个模型是自动获取的，系统通过模型拦截器完成模型的创建。

```
protected void before(ActionInvocation invocation)
     thowns Exception{
     Action action =invocation.getProxy().getAction();
     if(action instanceof ModelDriven){
          ModelDriven modelDriven=(ModelDriven)action;
          OgnlValueStack stack=invocation.getStack();
          stack.push(modelDriven.getModel());
     }
}
```

那么我们就可以从数据堆栈中去获取数据。

一般的 Action 可以如下文描述的那样，直接从 ActionSupport 一样可以获取数据，而不需要 getModel()的这个方法：

```
public class AdduserAction extends ActionSupport
       implements ServletRequestAware,UserDAOAware{
     private final static String ADDUSER_FAIL="adduserfail";
     TfUser user = new TfUser();
```

```
    public String execute() throws Exception {
    ......
    }
    public void setUser(TfUser user){
            this.user = user;
    }
    public String getUser (){
            return this.user;
    }
}
```

但是如果涉及文件上传时，我们的 Action 就要应该如下表述：

```
public class DocUpload extends ActionSupport {
    File doc;
    String docContentType;
    String docFileName;
    public String execute() throws Exception {
      return SUCCESS;
    }
    public void setDoc(File doc){
            this.doc = doc;
    }
    public void setDocContentType(String docContentType){
            this.docContentType= docContentType;
    }
    public void setDocFileName (String docFileName){
            this.docFileName = docFileName;
    }
}
```

而页面中应该表述如下即可。

```
<form action=”upload.action”
      enctype=”multipart/form-data”
      method=”post”>
     <input type=”file” name=”doc”/>
</form>
```

6.2 拦截器

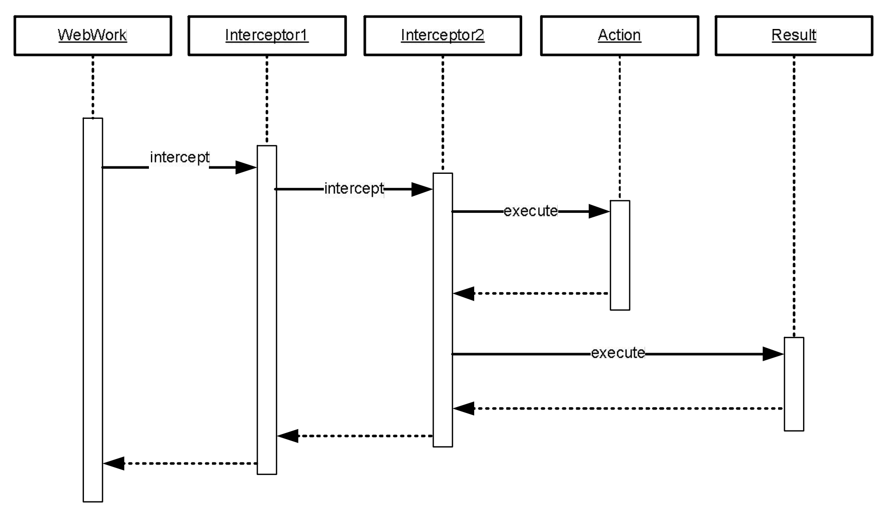
6.2.1 问题

传统的我们可能需要在调用一个 Action 之前做大量的工作，比如校验、权限控制等，那么就会为每个页面写上相应的验证器，或者为每个页面做权限控制。下面我们就提供一些能够配置的重复使用的解决方案。

6.2.2 解决方案

6.2.2.1 拦截器调用

WEBWORK 的拦截器非常优秀，它能够通过配置进行拦截 Action 的每个动作。我们重构前面的章节 2.1.2 “控制客户端访问”，原来基于页面的部分保护，和基于目录的安全保护，可以通过一个拦截器来达到我们的目的，而且可以灵活配置。



6.2.2.2 调用源码剖析

下面的这个样例就是能够把每个提供的 Action 调用之前进行了中断，并检查是否已经登录，如果没有登录就会拒绝并被返回到登录页。

样例

```
import com.doone.mudule.dto.TfUser;
import com.opensymphony.xwork.Action;
import com.opensymphony.xwork.ActionInvocation;
import com.opensymphony.xwork.interceptor.Interceptor;

public class AuthenticationInterceptor implements Interceptor{
	public void destroy() {
    }
	public void init() {
	}
	public String intercept(ActionInvocation actionInvocation) throws Exception {
		Map session = actionInvocation.getInvocationContext().getSession();
		TfUser user = (TfUser) session.get("logininfo");
		if (user == null) {
			return Action.LOGIN;
		} else {
			Object action = actionInvocation.getAction();
			if (action instanceof UserAware) {
				((UserAware)action).setUser(user);
			}
			return actionInvocation.invoke();
		}
	}
}
```

6.2.2.3 已封装的拦截器堆栈

拦截器	描述
defaultStack <interceptor-stack name=”defaultStack”> <interceptor-ref name=”servlet-config”/> <interceptor-ref name=”prepare”/> <interceptor-ref name=”static-params”/> <interceptor-ref name=”params”/> <interceptor-ref name=”conversionError”/>	把输入参数设置到 Action 的对应属性当中，这是最为基本的中断。

</interceptor-stack>	
validationWorkflowStack <interceptor-stack name=”validationWorkflowStack”> <interceptor-ref name=”defaultStack”/> <interceptor-ref name=”validation”/> <interceptor-ref name=”workflow”/> </interceptor-stack>	校验流程的中断拦截器
fileUploadStack <interceptor-stack name=”fileUploadStack”> <interceptor-ref name=”fileUpload”/> <interceptor-ref name=”defaultStack”/> </interceptor-stack>	文件上传的中断拦截器
componentStack <interceptor-stack name=”componentStack”> <interceptor-ref name=” component”/> <interceptor-ref name=”defaultStack”/> </interceptor-stack>	部件中断拦截器
modelDrivenStack <interceptor-stack name=”modelDrivenStack”> <interceptor-ref name=” model-driven”/> <interceptor-ref name=”defaultStack”/> </interceptor-stack>	模型驱动的中断拦截器
chainStack <interceptor-stack name=”chainStack”> <interceptor-ref name=”chain”/> <interceptor-ref name=”defaultStack”/> </interceptor-stack>	Action 链中断拦截器
executeAndWaitStack <interceptor-stack name=”executeAndWaitStack”> <interceptor-ref name=” execAndWait”/> <interceptor-ref name=”defaultStack”/> </interceptor-stack>	执行等待中断拦截器
completeStack <interceptor-stack name=”executeAndWaitStack”> <interceptor-ref name=”prepare”/> <interceptor-ref name=”servlet-config”/> <interceptor-ref name=” chain”/> <interceptor-ref name=”model-driven”/> <interceptor-ref name=” component”/>	完整版中断拦截器

```
   <interceptor-ref name="fileUpload"/>
   <interceptor-ref name=" static-params"/>
   <interceptor-ref name="params"/>
   <interceptor-ref name="conversionError"/>
   <interceptor-ref name="validation"/>
   <interceptor-ref name="workflow"/>
</interceptor-stack>
```

6.3 使用 Results

6.3.1 问题

当我们对一个动作完成之后可能还需要做进一步动作，那么充分利用结果集，那么传统的方案就需要包含大量的代码控制，甚至有时候还做不到。那么下面的这种解决方案就能够帮助你解决这个问题。

6.3.2 解决方案

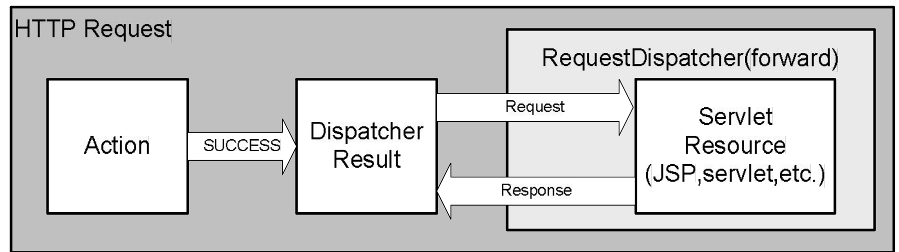
6.3.3 Action 生命周期之后

Action 生命周期之后，WebWork 会根据返回的动作，根据指定的配置做下一步的动作。

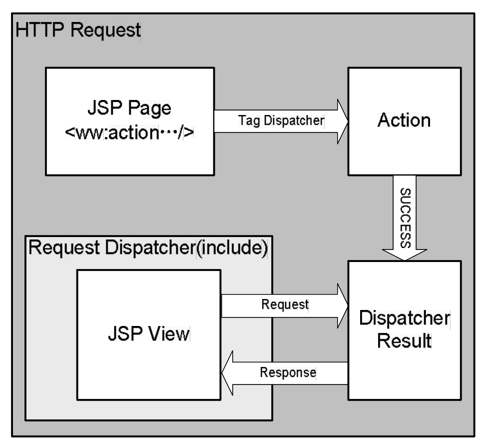
```
package com.opensymphony.xwork;
public interface Result{
    public void execute(ActionInvoction invocation) throws Exception;
}
```

6.3.4 常规 Results

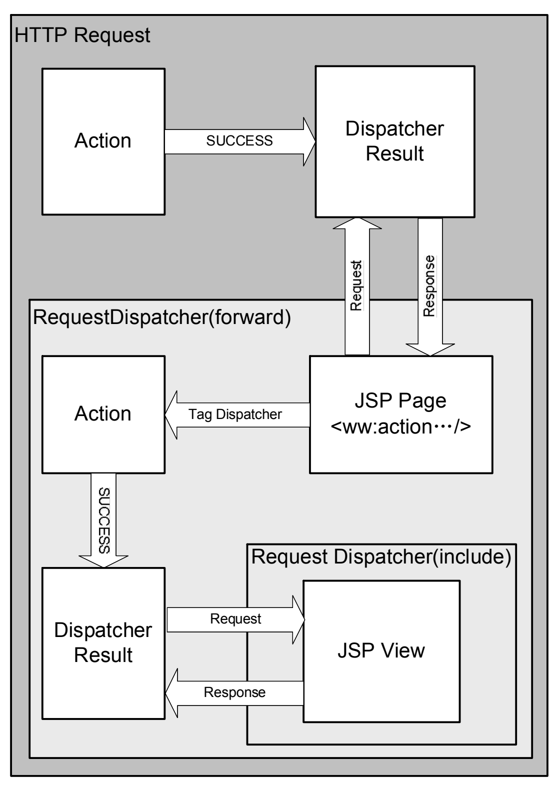
我们提供一个简单的 Action 样例，这是一个简单的阻断，前台请求一个 Action，返回一个成功的结果给前台。



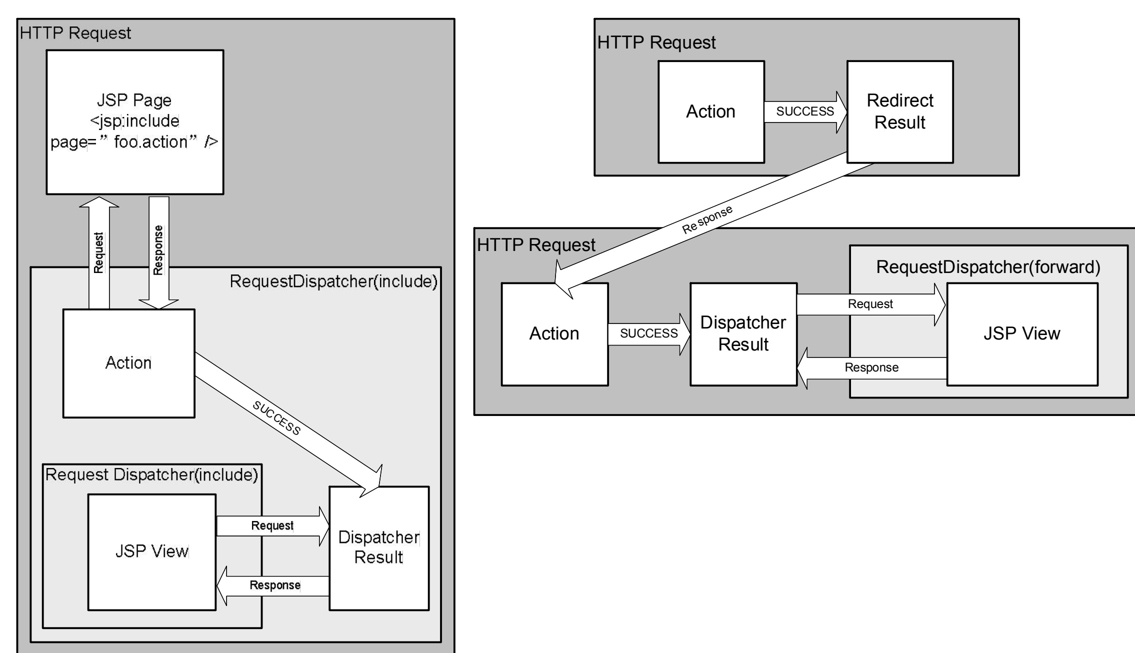
但是系统经常会出现下面这种情况，就是这个 Action 可能还需要调用另外一个 Action 的信息，这个时候返回的结果中就可能包含另外一个视图，也就是文章中出现的 include 的页面。这么结果就可能是一个结果集合了。



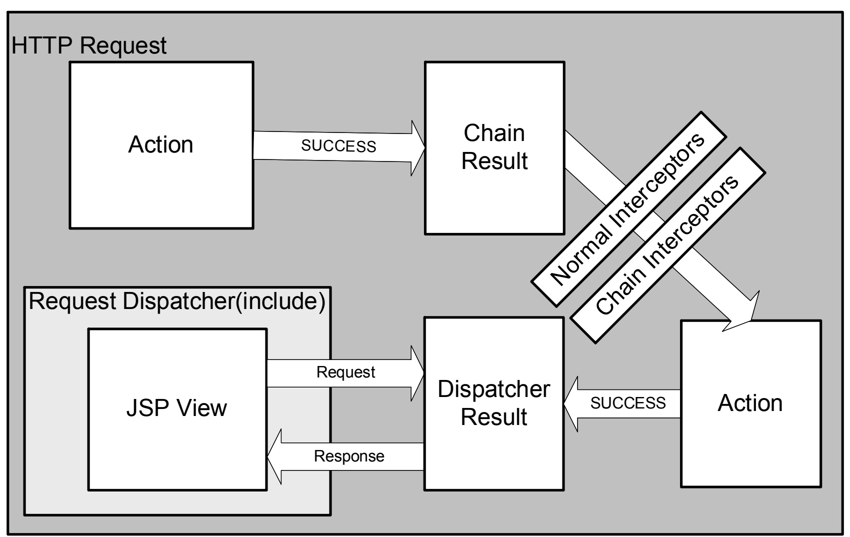
再进一步，第二个 Action，还不能满足实际的需求，还有再调用第三个 Action，这样我们可以清晰的看出，这是一个链状的 Action 表现形式，这个过程显然不是一个视图能够解决的，那么就充分利用返回结果的，通过配合的方式能够适应多种的变化形式。



例如以下两种形式：层层 Include 的形式，和层层跳转的形式：



综合起来，归纳以上形式，我们就引进以下的 ActionChain 的配置方式。



首先修改 web-default.xml：

```
<result-type name=”chain” class=”com.opensymphony.ActionChainResult”/>
```

然后配置一个 Action 的样例：

```
<package name=”example” extends=”webwork-default”>
   <action name=”authenticate”
   class=”com.doone.module.actions.Athenticate”>
      <result name=”success” type=”chain”>login</result>
      <result name=”input”>login.jsp</result>
   </action>
   <action name=”login” class=”com.doone.module.actions.Login”>
      <result name=”success” type=”chain”>home.jsp</result>
   </action>
</package>
```

6.4 使用标签库

6.4.1 问题

HTML 视图实现得非常早，因此没有包含足够的表达我们需要的很多关于交互式网页的要求，这就决定了需要制定一套能够符合我们要求的标识语言。以下就是我们的解决方案。

6.4.2 解决方案

WebWork 定义了很多不同类型的标签，基本上可以归类为数据标签、控制标签、UI 标签和杂项标签。数据标签关注从堆栈值中提取数据或将值设置到堆栈中去；控制标签提供根据“与/或”状态的条件判定流程的工具；杂项标签包含 include、URL 或国际化语言集，以及参数等标签。UI 标签能够影响网页的布局和风格等标签库。

6.4.2.1 数据标签

数据标签能够从数据堆栈中获取数据、变量或对象。在本节我们讨论 property 标签、set 标签、push 标签、bean 标签和 action 标签。property 标签用于提取数据堆栈中的值并显示给用户。set 标签用户定义你的页面中有用的临时变量。push 标签最大可能避免重复。bean 和 action 能够很好的构建重用的显示层的部件。

6.4.2.1.1Property 标签

Property 标签在 WebWork 标签中最通用的，表达式基于 OGNL。

属性	数据类型	必须项	描述

value	Object	No	获取值的表达式
default	String	No	数据没有获取值时的默认值
escape	Boolean	No	HTML 中可能包含的”&”是否转换为”&”?

样例：

```
<ww:property value=”name”/>
```

6.4.2.1.2Set 标签

将数据堆栈范围的某些数据指定为一个临时变量。

属性	数据类型	必须项	描述
name	String	Yes	指定引用范围的变量名称
value	Object	Yes	设置的值的表达式
scope	String	No	application,session,request,page,default

样例：

指定前：

```
<ww:property value=”#session[‘user’].username”/>
<ww:property value=”#session[‘user’].age”/>
<ww:property value=”#session[‘user’].address”/>
```

指定后：

```
<ww:set name=”user” value=” #session[‘user’]”/>
<ww:property value=”username”/>
<ww:property value=”age”/>
<ww:property value=”address”/>
```

6.4.2.1.3Push 标签

允许你往 action 的内容中赋值，将各种引用的值推到数据堆栈中去。

属性	数据类型	必须项	描述
value	Object	Yes	Push 到数据堆栈中的值的表达式

样例：

```
<ww:set name="user" value=" #session['user']"/>
<ww:push value="#user">
    <ww:property value="username"/>
    <ww:property value="age"/>
    <ww:property value="address"/>
</ww:push>
```

可以变化如下：

```
<%@taglib prefix="ww" uri="webwork"%>
<html>
   <head>
       <title></title>
   </head>
   <body>
        <ww:push value="cart.user">
        <%@include file="/share/jsp/user-details.jspf"%>
        </ww:push>
   </body>
</html>
```

而在另一个页面 user-details.jspf 中就可以直接显示内容了。

```
<%@taglib prefix="ww" uri="webwork"%>
<ul>
    <li>Username:<ww:property value="username"/></li>
    <li>Age:<ww:property value="age"/></li>
    <li>Address:<ww:property value="address"/></li>
</ul>
```

6.4.2.1.4Bean 标签

属性	数据类型	必须项	描述
name	String	Yes	bean 类的全路径名称
id	String	No	被引用 bean 的名称

样例：

该例子中 Counter Bean 被创建，并设置 last 为 100，那么从 1 到 100 循环打印出来。

```
<%@taglib prefix=”ww” uri=”webwork”%>
<ww:bean name=”com.opensymphony.webwork.util.Counter” id=”counter”>
    <ww:property name=”last” value=”100”/>
</ww:bean>
<ww:iterator value=”#counter”>
    <li><ww:property/></li>
</ww:iterator>
```

该例子格式化时间格式：

```
<%@taglib prefix=”ww” uri=”webwork”%>
<ww:set name=”user” value=”#session[‘user’]”/>
<ww:bean name=”com.opensymphony.webwork.util.DateFormatter” >
   <ww:property name=”format” value=”yyyy-MM-dd”/>
   <ww:property name=”date” value=”#user.birthdate”/>
   The user’s birthdate is:<ww:property value=”formattedDate”/>
</ww:bean>
```

6.4.2.1.5Action 标签

属性	数据类型	必须项	描述
name	String	Yes	Action 名称
namespace	String	No	Action 的命名空间，默认当前页的命名空间
id	String	No	被引用 Action 的名称
executeResult	Boolean	No	当设置为 true 时，执行 Action 后继续执行该结果，默认 false。

样例：

这个<ww:action/>动作会去提交一个 userinfo.action 的动作，并把值取出来放到名称为 uinfo 的 bean 中，并进行显示。

```
<ww:action name="userinfo" id="uinfo"/>
Browser:<ww:property value="#uinfo.browser"/><br>
Version:<ww:property value="#uinfo.version"/><br>
Supports GIF:<ww:if test="#uinfo.supportsType('image/gif') == true">Yes</ww:if>
<ww:else>No</ww:else><br>
```

6.4.2.2 控制标签

6.4.2.2.1Iterator 标签

属性	数据类型	必须项	描述
value	Collection,Map,Iterator Enumeration,Array	Yes	一个循环的对象
status	String	No	如果提供，它是一个迭代的状态

样例：

```
<ww:iterator value="items">
    <li>
    <ww:property value="name"/>,<ww:property value="description"/>
    </li>
</ ww:iterator>
```

6.4.2.2.2If/Else 标签

属性	数据类型	必须项	描述
test	Boolean	Yes	布尔表达式

样例：

下面一个表格对行有三种控制样式，奇数、偶数和选择样式：

```
<table>
<thead>
    <tr>
        <th>Name</th>
        <th>Description</th>
    </tr>
</thead>
<tbody>
   <ww:iterator value="items" status="status">
   <tr class="
```

```
        <ww:if test="id==itemId">row-selected</ww:if>
        <ww:elseif test="#status.even">row-even<ww:elseif>
        <ww:else>row-odd<ww:else>
    ">
         <td><ww:property value="name"/></td>
         <td><ww:property value="description"/></td>
    </tr>
    </ww:iterator>
</tbody>
</table>
```

6.4.2.3 杂项

6.4.2.3.1Include 标签

属性	数据类型	必须项	描述
value	String	Yes	page 的名称,action,servlet 或引用的 URL

样例：

```
<%@taglib prefix="ww" uri="webwork"%>
……
<ww:include page="helloWord.action"/>
……
```

6.4.2.3.2URL 标签

属性	数据类型	必须项	描述
value	String	No	URL；默认当前页的引用页
includeParams	String	No	选择 all,get,none 中选择参数，默认是 get
id	String	No	如果指定，URL 没有写出，保存在 Action 的 Context 中备将来使用。
includeContext	Boolean	No	如果 true,URL 生成时必需和应用软件的 Context 捆绑。

encode	Boolean	No	附加 sessionid 到 URL，如果 cookie 被禁用时
schema	String	No	允许指定通信协议如 http 或 https

6.4.2.3.3i18n 和 Text 标签

Text 标签

属性	数据类型	必须项	描述
name	String	Yes	查找资源(ResouceBundle)的主键
id	String	No	如果指定，保存在 action 的 context 中备用
value0	Object	No	参数 1
value1	Object	No	参数 2
value2	Object	No	参数 3
value3	Object	No	参数 4

样例：

```
<ww.text name=”searchResults”>
    <ww:param value=”totalItems”/>
    <ww:param value=”searchCount”/>
</ww:text>
```

i18n 标签

属性	数据类型	必须项	描述
name	String	Yes	资源(ResouceBundle)的名称

样例：

```
<ww:i18n name="'messages'">
……
      <ww:text name=”someKey”/>
……
</ww:i18n>
```

6.4.2.3.4param 标签

为标签附加一些参数扩展标签。

属性	数据类型	必须项	描述
name	String	No	all,get 或 none
value	Object	No	为请求页获取 URL

样例：

<ww:param/>没有作为提交参数而仅仅提交给<ww:text/>，那么 name 将给予忽略。样例给出了一个对非常的数，比如 500 万，这个数值在页面中可能很难显示，那么采用“Over 5 million”加以替换，可以表述如下：

```
<ww.text name=”searchResults”>
    <ww:param>Over 5 million</ww:param>
    <ww:param value=”searchCount”/>
</ww:text>
```

可以变体如下：

```
<ww.text name=”searchResults”>
    <ww:param><ww.text name=”fiveMillion”></ww:param>
    <ww:param value=”searchCount”/>
</ww:text>
```

6.5 Context 对象

6.5.1 问题

以上解决方案都一直要求可以配置，那么就需要一个能够将的这些解决方案进行耦合的强有力的工具。

6.5.2 解决方案

WebWork 的 Context 对象贯穿了整个表现层，它能够非常灵活地将表现层

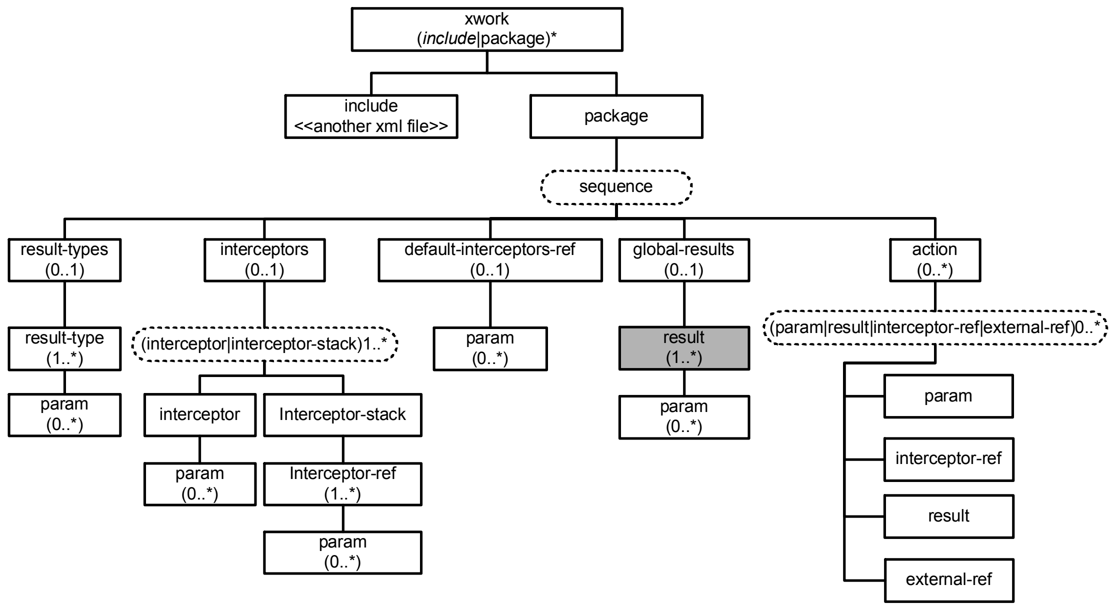
进行上下文的关联。其中包含如下几个重要文件：web.xml、xwork.xml、validators.xml、webwork.tld、xxxxAction-validation.xml。

6.5.2.1 web.xml

```
<web-app>
    <filter>
        <filter-name>container</filter-name>
        <filter-class>
            com.opensymphony.webwork.lifecycle.RequestLifecycleFilter
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>container</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <listener>
        <listener-class>
            com.opensymphony.webwork.lifecycle.ApplicationLifecycleListener
        </listener-class>
    </listener>
    <listener>
        <listener-class>
            com.opensymphony.webwork.lifecycle.SessionLifecycleListener
        </listener-class>
    </listener>
    <servlet>
        <servlet-name>velocity</servlet-name>
        <servlet-class>
            com.opensymphony.webwork.views.velocity.WebWorkVelocityServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet>
        <servlet-name>webwork</servlet-name>
        <servlet-class>
            com.opensymphony.webwork.dispatcher.ServletDispatcher
```

```
        </servlet-class>
        <init-param>
            <param-name>debug</param-name>
            <param-value>2</param-value>
        </init-param>
        <init-param>
            <param-name>detail</param-name>
            <param-value>2</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <!-- mappings for *.action and *.vm (needed for ww) -->
    <servlet-mapping>
        <servlet-name>webwork</servlet-name>
        <url-pattern>*.action</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>velocity</servlet-name>
        <url-pattern>*.vm</url-pattern>
    </servlet-mapping>
    <!--ww tag library-->
    <taglib>
        <taglib-uri>webwork</taglib-uri>
        <taglib-location>/WEB-INF/webwork.tld</taglib-location>
    </taglib>
</web-app>
```

6.5.2.2 XWORK DTD



WebWork 的 DTD 配置视图

6.5.2.3 XWORK.XML

下文包含了主要的包：default、public 和 secure：

```
<xwork>
<include file="webwork-default.xml"/>
<package name="default" extends="webwork-default">
   <interceptors>
   <interceptors name="auth"
      class="com.doone.module.web.interceptors.AuthenticationInterceptor"/>
   </interceptors>
   <global-results>
        <result name="login" type="redirect">/login.jsp</result>
   </global-results>
</package>
<package name="public" extends="default">
   <default-interceptor-ref name="completeStack"/>
   <!-- public facing actions-->
</package>
<package name="secure" extends="default" namespace="/secure">
   <interceptor-stack name="default">
```

```
      <interceptor-ref name="auth"/>
      <interceptor-ref name="completeStack"/>
   </interceptor-stack>
   <default-interceptor-ref name="default"/>
   <!-- protected actions -->
</package>
</xwork>
```

注意包的命名是唯一的，你不能在同一个 xwork.xml 中包含两个相同的名称。

```
<xwork>
<include file="webwork-default.xml"/>
<package name="default" extends="webwork-default">
<default-interceptor-ref name="defaultStack"/>
        <action name="reguser" class="com.doone.module.action.ReguserAction">
              <result name="success" type="dispatcher">
                    <param name="location">/main/index.htm</param>
              </result>
              <result name="regfail" type="dispatcher">
                    <param name="location">/index.jsp</param>
              </result>
              <result name="input" type="dispatcher">
                    <param name="location">/index.jsp</param>
              </result>
              <interceptor-ref name="defaultStack"/>
              <interceptor-ref name="validationWorkflowStack" />
      </action>
</package>
</xwork>
```

6.5.2.4 validators.xml

```
<validators>
     <validator name="required"
         class="com.opensymphony.xwork.validator.validators.RequiredFieldValidator"/>
     <validator name="requiredstring"
```

```
            class="com.opensymphony.xwork.validator.validators.RequiredStringValidator"/>
    ……
    <validator name="stringregex"
        class=" com.doone.module.web.validators.StringRegexValidator"/>
    <validator name="usernamecheck"
        class=" com.doone.module.web.validators.UsernameCheckValidator"/>
    <validator name="passwdcheck"
        class="com.doone.validator.PasswordCheckValidator"/>
    <validator name="fieldeq"
        class=" com.doone.module.web.validators.FieldEqualsValidator"/>
</validators>
```

6.5.2.5 *Action-validation.xml

```
<validators>
     <field name="model.username">
         <field-validator type="stringregex">
             <param name="regex">^([a-zA-Z0-9_]{5,13})$</param>
             <message>
             你必须输入由字母、数字及下划线组成且长度(5-13)的用户名。
             </message>
         </field-validator>
    </field>
    <field name="model.pwd">
         <field-validator type="requiredstring">
         <message>请输入密码</message>
         </field-validator>
        <field-validator type="stringlength">
        <param name="maxLength">13</param>
             <message>密码长度不能超过 13 个字节</message>
         </field-validator>
    </field>
</validators>
```

7 业务层模式

在业务层我们配置对包的路径做出，严格的命名要求。请各个项目组务必遵守。我们自动化生成的插件也是基于这么包的路径命名进行的。这种固化的命名将给项目组带来好处，就是开发者能够基于统一的命名模式去理解业务代码，开发者可能在项目组中流动，那么人员复用会变轻松一些。同时有个更深层次的原因就是代码共享，基于这种命名模式的代码更加易于集成，能够实现模块化的粒度的限定，我们实际项目中往往有很多的功能相似性，经过抽象后就是一个相同的功能模块，那么就无须重新开发。

包定义	描述
com.doone.[模块名].dao	DAO 的接口类，例如 TfUserDAO.java
com.doone.[模块名].dao.ibatis	DAO 的实现类，例如 TfUserDAOImpl.java
com.doone.[模块名].dao.ibatis.maps	IBATIS 的 XML 映射文件，例如 tf_user.xml
com.doone.[模块名].domain	业务门面接口类，例如 TfUserFacade.java
com.doone.[模块名].domain.logic	业务门面实现类，例如 TfUserImpl.java
com.doone.[模块名].dto	DTO 的映射类，例如 TfUser.java
WEB-INF/classes/xwork.xml	Action 配置文件
WEB-INF/classes/Ibatis-Context.xml	Spring 业务封装类配置
WEB-INF/classes/sqlMap-config.xml	IBATIS 映射类配置
WEB-INF/classes/validators.xml	WebWork 默认校验器配置
WEB-INF/classes/webwork-default.xml	WebWork 默认拦截器配置

7.1 数据传输对象(DTO)

数据传输对象在业务层和持久层的表现形式如出一辙，可以参考 3.1.2 章节。

7.2 业务门面(Facade)

7.2.1 问题

众多的业务需要一定的组织才能够达到模式要求的复用的目的，那么我们就一定需要对业务进行封装，封装后，事务又如何控制呢。

7.2.2 解决方案

所谓业务门面(Facade)，是业务层对业务逻辑上的封装，它包含业务逻辑的控制和操作。例如我们需要添加一个用户时，需要做很多个动作，首先我们判定用户名是否已经存在，如果不存在才添加用户，在添加用户时，需要把用户的基本信息建立完整，比如用户还包含哪些角色，这么就要涉及到角色类的数据操作，这些操作事实上是一个整体，那么我们就可以把注册用户的整个动作封装起来，形成门面的一个方法。

7.2.2.1 业务封装

业务封装就是把基本粒度的业务进行封装，并封装到一个业务门面的方法中。下面的例子就是基于最基本粒度封装的业务门面。

```
public class TfUserImpl implements com.doone.module.domain.TfUserFacade{
    private com.doone.module.dao.TfUserDAO tfuserDAO;
     public com.doone.module.dao.TfUserDAO    getTfuserDAO(){
        return this.tfuserDAO;
        }
    public void setTfuserDAO(com.doone.module.dao.TfUserDAO tfuserDAO){
        this.tfuserDAO=tfuserDAO;
        }
    public java.util.List findTfUser(com.doone.module.dto.TfUser tfuser){
        return this.tfuserDAO.findTfUser(tfuser);
    }
    public void insertTfUser(com.doone.module.dto.TfUser tfuser){
```

```
        this.tfuserDAO.insertTfUser(tfuser);
    }
    public void updateTfUser(com.doone.module.dto.TfUser tfuser){
        this.tfuserDAO.updateTfUser(tfuser);
    }
    public void deleteTfUser(com.doone.module.dto.TfUser tfuser){
        this.tfuserDAO.deleteTfUser(tfuser);
    }
}
```

在例子中我们可以看见，业务基于数据层面的 4 个基本动作被封装了起来。那么我们就对这个基础类封装一个注册用户的方法。

```
public class TfUserImpl implements com.doone.module.domain.TfUserFacade{

    private com.doone.module.dao.TfUserDAO tfuserDAO;
    private com.doone.module.dao.TfRoleDAO tfroleDAO;

    public com.doone.module.dao.TfUserDAO   getTfuserDAO(){
        return this.tfuserDAO;
        }
    public void setTfuserDAO(com.doone.module.dao.TfUserDAO tfuserDAO){
        this.tfuserDAO=tfuserDAO;
        }
    public com.doone.module.dao.TfRoleDAO   getTfroleDAO(){
        return this.tfroleDAO;
        }
    public void setTfroleDAO(com.doone.module.dao.TfRoleDAO tfroleDAO){
        this.tfroleDAO=tfroleDAO;
        }
    public java.util.List findTfUser(com.doone.module.dto.TfUser tfuser){
        return this.tfuserDAO.findTfUser(tfuser);
    }
    public void insertTfUser(com.doone.module.dto.TfUser tfuser){
        this.tfuserDAO.insertTfUser(tfuser);
    }
    public void updateTfUser(com.doone.module.dto.TfUser tfuser){
        this.tfuserDAO.updateTfUser(tfuser);
    }
```

```
    public void deleteTfUser(com.doone.module.dto.TfUser tfuser){
        this.tfuserDAO.deleteTfUser(tfuser);
    }
    public void addUser(com.doone.module.dto.TfUser tfuser,
                        com.doone.module.dto.TfRole tfrole)
    {
        if(findTfUser(tfuser).size()==0)
        {
            tfuserDAO.insertTfUser(tfuser);
            tfroleDAO.insertTfRole(tfrole);
        }
    }
}
```

这里我们用了两个 DAO 操作了一个注册用户的类。这个 AddUser()就是一个业务的封装，它将在注册用户时被反复调用。

7.2.2.2 事务控制

上面的例子，会出现这么一个问题。也就是当 insertTfUser()调用成功了，但是 insertTfRole()调用失败，典型的原因就是主键冲突，完整性得不到保障。这对我们的业务可能是致命的，会出现用户无法拥有一个正确的角色。如果是银行系统这种更为严格的数据，那么问题就可能导致整个系统被客户否认。这个使用我们就应该给它引进一个事务的概念。

Spring 在事务控制方面的功能是整个 Spring 微框架中最出色的功能，我们可以通过配置的方式告诉 Spring 某个方法，比如 addUser()方法就是一个事务。

Spring 使用 aop 机制管理 jdbc 的连接和事务。它使用 TransactionInterceptor 类，Spring 事务支持中的核心接口是:

org.springframework.transaction.PlatformTransactionManager

为了实际执行事务，Spring 所有的事务划分功能都通过传递适当的 TransactionDefinition 实例，委托给 PlatformTransactionManager。尽管 PlatformTransactionManager 接口可以直接使用，应用程序通常配置具体的事务管理器并使用声明性事务来划分事务。

Spring 具有多种 PlatformTransactionManager 实现，它们分为两类：局部

事务策略和全局事务策略。

局部事务策略即针对单个资源执行事务（主要是针对单个的数据库）。实现有 org.springframework.jdbc.datasource.DataSourceTransactionManager。它用于 jdbc 数据源的配置，调用 TransactionInterceptor 开是一个事务,从 DataSource 得到一个 connection 并确保 auto-commit 设为 disabled。他用 JdbcTemplate 在一个线程内绑定一个 JDBC connection，TransactionInterceptor 负责提交事务，DataSourceTransactionManager 调用 Connection.commit()关闭 connection,并解除绑定（potentially allowing for one thread connection per data source）。

我们在 Ibatis-Context.xml 中配置如下：

```
<beans>
<bean id="baseTransactionProxy"
   class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"
   abstract="true">
   <property name="transactionManager" ref="transactionManager" />
   <property name="transactionAttributes">
   <props>
      <prop key="add*">PROPAGATION_REQUIRED</prop>
      <prop key="find*">PROPAGATION_REQUIRED,readOnly</prop>
   </props>
   </property>
</bean>



<bean id="tfroleDAO" class="com.doone.module.dao.ibatis.TfRoleDAOImpl">
   <property name="sqlMapClient">
      <ref local="sqlMapClient" />
   </property>
</bean>

<bean id="tfuserDAO" class="com.doone.module.dao.ibatis.TfUserDAOImpl">
   <property name="sqlMapClient">
      <ref local="sqlMapClient" />
   </property>
</bean>
```

```
<bean id="tfuserDAOProxy" parent="baseTransactionProxy">
   <property name="target">
      <bean class="com.doone.module.domain.logic.TfUserImpl">
         <property name="tfuserDAO" ref="tfuserDAO"></property>
         <property name="tfroleDAO" ref=" tfroleDAO"></property>
     </bean>
  </property>
</bean>
</beans>
```

经过声明后，Spring 在调用这个 add 开头的方法时，就会被认为是一个事务，如果出现异常，事务就会被自动回滚。其中 transactionAttributesSource 属性指定每个方法的 transaction attribute，PROPAGATION_REQUIRED 说明在一个事务内这个方法被执行。默认的情况下，spring 只有当 unchecked exception 被抛出时，才 rollback 事务，也可以自己加入 checked exception。tatanTransactionScripts 被 TransactionInterceptor 封装，在一个事物内执行类的每一个方法。

对于特定的方法或方法命名模式，代理的具体事务行为由事务属性驱动，如下面的例子所示：

```
<prop key="insert*">
PROPAGATION_REQUIRED, ISOLATION_READ_COMMITTED
</prop>
```

下表对事务的定义关键字注释：

关键字	描述
PROPAGATION_REQUIRED	支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。
PROPAGATION_SUPPORTS	支持当前事务，如果当前没有事务，就以非事务方式执行。
PROPAGATION_MANDATORY	支持当前事务，如果当前没有事务，就抛出异常。
PROPAGATION_REQUIRES_NEW	新建事务，如果当前存在事务，把当前事务挂起。
PROPAGATION_NOT_SUPPORTED	以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

PROPAGATION_NEVER	以非事务方式执行，如果当前存在事务，则抛出异常。
PROPAGATION_NESTED	如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则进行与 PROPAGATION_REQUIRED 类似的操作。

全局事务管理还不是我们推荐的模式，因此简单的加以描述。

全局事务管理即执行有可能跨越多个资源的全局事务。主要对应的 Spring 类是 org.springframework.transaction.jta.JtaTransactionManager，它委托给遵循 JTA 规范的 J2EE 服务器，也有例外。

spring 支持 JTA，只需要一个标准的 JtaTransactionManager 定义，数据库必须支持 XA protocol，或者 J2EE 服务器提供支持 XA 规范的 DataSource。

默认的 Spring JtaTransactionManager 设置将从标准的 JNDI 位置获取 JTA 的 javax.transaction.UserTransaction 对象，该 JNDI 位置由 J2EE 指定：java: comp/UserTransaction。对于大多数标准 J2EE 环境下的用例来说，它工作良好。

但是，默认的 JtaTransactionManager 不能执行事务挂起操作

(即不支持 PROPAGATION_REQUIRES_NEW

和 PROPAGATION_NOT_SUPPORTED)。

原因是标准的 JTA UserTransaction 接口不支持挂起或恢复事务的操作；它只支持开始和完成新事务的操作。

为执行事务挂起操作，还需要提供 javax.transaction.TransactionManager 实例，按照 JTA 的规定，它提供标准的挂起和恢复方法。遗憾的是，J2EE 没有为 JTA TransactionManager 定义标准的 JNDI 位置！

因此，必须使用特定于供应商的(vendor-specific)查寻机制。J2EE 没有考虑把 JTA TransactionManager 接口作为它的公开 API 的一部分。JTA 规范规定的 TransactionManager 接口原本是打算用于容器集成的。

但是为 JTA TransactionManager 定义标准的 JNDI 位置还是有重大意义的，尤其是对于轻量级容器（如 Spring）；然后，便可以以同样的方式来定位任意的 J2EE 服务器的 JTA TransactionManager。

7.3 Context 对象

7.3.1 问题

和表现层的解决方案一样，我们同样就需要一个能够将的这些解决方案进行耦合的强有力的工具。

7.3.2 解决方案

加载 Spring 的上下文事实上非常容易，你可以在 web.xml 对 Spting 进行加载。加载时，Web 服务器会对 Spring 文件定义的业务门面和数据访问对象进行实例化，能够加速 Web 服务的启动。

7.3.2.1 web.xml

```
……
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/Ibatis-Context.xml /WEB-INF/Ibatis-Context-Ex.xml
        </param-value>
</context-param>
<context-param>
        <param-name>log4jConfigLocation</param-name>
        <param-value>/WEB-INF/classes/log4j.properties</param-value>
</context-param>

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

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

那么在 Aciton 中就可以直接查找这个实例：

```
WebApplicationContext wac;
wac = (WebApplicationContext)ActionContext.getContext()
   .getApplication()
   .get(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE);
TfUserFacade userFacade = (TfUserFacade) wac.getBean("tfuserDAOProxy");
```

但是如果还没有采用 WebWork 的用户应该采用以下方式获取实例：

```
WebApplicationContext wac = WebApplicationContextUtils
    .getRequiredWebApplicationContext(pageContext
                                    .getServletContext());
     TfUserFacade userFacade = (TfUserFacade) wac.getBean("tfuserDAOProxy");
```

而如果不在 WebServer 系统中使用 Spring，可以采用以下方式加载并获取实例：

```
ClassPathXmlApplicationContext appContext
    = new ClassPathXmlApplicationContext("Ibatis-Context.xml");
     BeanFactory factory = (BeanFactory) appContext;
     TfUserFacade userFacade = (TfUserFacade) factory.getBean("tfuserDAOProxy");
```

7.3.2.2 Ibatis-Context.xml

```
<?xml version="1.0" encoding="gb2312" ?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
<bean id="dataSource"
  class="org.apache.commons.dbcp.BasicDataSource"
  destroy-method="close">
```

```
  <property name="driverClassName">
    <value>oracle.jdbc.driver.OracleDriver</value>
  </property>
  <property name="url">
    <value>jdbc:oracle:thin:@localhost:1521:XE</value>
  </property>
  <property name="username">
    <value>webwork</value>
  </property>
  <property name="password">
    <value>webwork</value>
  </property>
</bean>

<bean id="sqlMapClient"
  class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
  <property name="configLocation">
    <value>/sqlMap-config.xml</value>
  </property>
  <property name="dataSource">
    <ref local="dataSource"/>
  </property>
</bean>
<bean id="transactionManager"
 class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
 <property name="dataSource">
    <ref local="dataSource"/>
 </property>
</bean>

<bean id="baseTransactionProxy"
  class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"
  abstract="true">
  <property name="transactionManager" ref="transactionManager" />
  <property name="transactionAttributes">
  <props>
    <prop key="insert*">PROPAGATION_REQUIRED</prop>
    <prop key="update*">PROPAGATION_REQUIRED</prop>
```

```
        <prop key="delete*">PROPAGATION_REQUIRED</prop>
        <prop key="find*">PROPAGATION_REQUIRED,readOnly</prop>
    </props>
    </property>
</bean>


<bean id="tfroleDAO" class="com.doone.module.dao.ibatis.TfRoleDAOImpl">
    <property name="sqlMapClient">
        <ref local="sqlMapClient" />
    </property>
</bean>

<bean id="tfroleDAOProxy" parent="baseTransactionProxy">
    <property name="target">
        <bean class="com.doone.module.domain.logic.TfRoleImpl">
            <property name="tfroleDAO" ref="tfroleDAO"></property>
      </bean>
  </property>
</bean>

<bean id="tfuserDAO" class="com.doone.module.dao.ibatis.TfUserDAOImpl">
    <property name="sqlMapClient">
        <ref local="sqlMapClient" />
    </property>
</bean>

<bean id="tfuserDAOProxy" parent="baseTransactionProxy">
    <property name="target">
        <bean class="com.doone.module.domain.logic.TfUserImpl">
            <property name="tfuserDAO" ref="tfuserDAO"></property>
      </bean>
  </property>
</bean>

</beans>
```

8 持久层模式

持久层是 ORM 概念的体现，能对数据苦结构提供了较为完整的封装，提供了从 POJO(Plain Ordinary Java Object，我们在本文中统一命名为 DTO：数据传输对象)到数据库表的全套映射机制。开发者往往只需要对定义好的 POJO 到数据库表之间的映射关系，就可以通过 iBATIS 提供的方法完成持久层的操作，设计师负责对业务逻辑进行抽象后，指定映射关系。这些方法我们本文统一归类名命名为 DAO 即数据访问对象。

8.1 数据传输对象(DTO)

数据传输对象在业务层和持久层的表现形式如出一辙，可以参考 3.1.2 章节。

8.2 数据访问对象(DAO)

8.2.1 问题

原先我们的程序是通过获取页面的请求之后，将参数进行传递给一个 SQL 语句，然后直接通过一个 JDBC 的链接将数据进行持久化的，这是极其高效但无法复用的办法，因此不能将其称为模式。但系统的业务不断积累到一定程度之后，对开发者的技术要求也就越来越高起来，还可能就会导致无法继续开发下去。如何解决这个问题，下面的章节给出相应的模式解决这个问题。

8.2.2 解决方案

我们抽象并封装了对持久化存储的访问，对于任何一种数据，在数据库里的操作就只有四种：添加、删除、修改和查询。我们完全可以抽象成四个方案。抽象后，我们就只要根据对象的方法进行访问就可以了。

同时我们约束我们的模式，就是不能将持久化逻辑和业务应用逻辑混淆起来，这样就会导致应用程序直接依赖于持久化存储机制的实现。一旦在组件中出现这样的依赖，再想把应用程序从一种数据源移植到另一种数据源就会出现困难重重。这就是我们的工具为什么仅仅生成这四个方法的原因。

```
package com.doone.module.dao.ibatis;

/*
 * This source file was generated by iBATIS Synchronizer v1.0.6 (build Eclipse 3.1.1)
 * on 2006-03-27 13:22:13
 *
 * (c) Copyright FUJIAN NEW DOONE SCIENCE&TECHNOLOGY CO.,LTD. 2005, 2006.
 * Contributor:lizx@doone.com.cn
 * All Rights Reserved.
 */

import org.springframework.dao.DataAccessException;
import org.springframework.orm.ibatis.support.SqlMapClientDaoSupport;

public class TfUserDAOImpl extends SqlMapClientDaoSupport
        implements com.doone.module.dao.TfUserDAO{
    public java.util.List findTfUser(com.doone.module.dto.TfUser tfuser)
            throws DataAccessException {
            return (java.util.List)getSqlMapClientTemplate()
                                .queryForList("findTfUserDao",tfuser);
    }
    public void insertTfUser(com.doone.module.dto.TfUser tfuser)
            throws DataAccessException {
            getSqlMapClientTemplate().update("insertTfUserDao",tfuser);
    }
```

```
    public void updateTfUser(com.doone.module.dto.TfUser tfuser)
            throws DataAccessException {
            getSqlMapClientTemplate().update("updateTfUserDao",tfuser);
    }
    public void deleteTfUser(com.doone.module.dto.TfUser tfuser)
            throws DataAccessException {
            getSqlMapClientTemplate().update("deleteTfUserDao",tfuser);
    }
}
```

8.2.2.1经典分页

ibatis 提供了 PaginatedList 的类，你能够对代码做如下稍微的改造即可。

```
public com.ibatis.common.util.PaginatedList
   findTfUser(com.doone.module.dto.TfUser tfuser,int pageSize)
   throws DataAccessException {
                    return (com.ibatis.common.util.PaginatedList)
                    getSqlMapClientTemplate()
                    .queryForPaginatedList("findTfUserDao",tfuser,pageSize);
    }
```

这个时候返回的对象是 PaginatedList，它提供了 listIterator()获取当前分页的 list 对象，你还可以通过 nextPage()和 previousPage()在不同的分页中进行滚动，但是这个方法有个很致命的弱点，就是所有的数据都被放到缓存中去，如果对于数据量非常庞大的系统来说，根本就不能接受。因此我们引起自定义一个分页的方法。

8.2.2.2 自定义分页

传统的分页我们会使用 SQL 来完成这个功能，那么我们经常会以下这个相对优化的 SQL 来分页。

```
select *
from (
select row_.*, rownum rownum_ from
```

```
(select * from tf_user) row_ where rownum <= 999)
where rownum_ > 0
```

那么我们改造映射文件，映射文件具体含义请阅读 8.3.2.3“映射文件”。

```
<sqlMap namespace="TfUser">
    <select id="findTfUserPageDao"
    resultClass="com.doone.module.dto.TfUser">
    <![CDATA[
    from (
    select row_.*, rownum rownum_ from
    (
    select ID AS ID,
         USERNAME AS USERNAME,
         PASSWORD AS PASSWORD,
         AUTOLOGIN AS AUTOLOGIN,
         ROLEID AS ROLEID
         from TF_USER
         ]]>
       <dynamic prepend="where">
           <isPropertyAvailable prepend="" property="id" >
               <isNotNull prepend="and" property="id">
               <![CDATA[ID=#id#]]>
               </isNotNull>
           </isPropertyAvailable>
           ……
       </dynamic>
       <![CDATA[
        ) row_ where rownum <= #startnum#)
        where rownum_ > #endnum#
         ]]>
  </select>
</sqlMap>
```

同时扩展 DAO 方法如下：

```
public java.util.List findTfUser(java.util.Map tfuser)
      throws DataAccessException {
      return (java.util.List)getSqlMapClientTemplate()
       .queryForList("findTfUserPageDao",tfuser);
```

```
    }
```

调用时初始化对象：

```
Map tfuser = new HashMap(2);
        tfuser.put("startrum",new Integer(0));
        tfuser.put("endrum",new Integer(19));
```

8.3 Context 对象

8.3.1 问题

持久层中我们看见了有很多千丝万缕的关系，这些个关系如何进行关系，又能够很清晰的进行组件，很大程度上需要 XML 的灵活性加以支持，解决这个问题，持久层一般都提供了相应的工具。

8.3.2 解决方案

持久层的 Context 对象复杂，我们可以通过阅读以下的内容加以理解。

8.3.2.1 sqlMap-config.xml

```
<?xml version="1.0" encoding="gb2312" ?>
<!DOCTYPE sqlMapConfig
PUBLIC "-//iBATIS.com//DTD SQL Map Config 2.0//EN"
"http://www.ibatis.com/dtd/sql-map-config-2.dtd">
     <sqlMapConfig>
     <properties resource="jdbc.properties"/>
     <!-- debug 环境下，将其设为 false. 正式运行时应设为 true,启用缓存  -->
     <settings cacheModelsEnabled="false"/>
     <!-- LRU 型 cache 样例  -->
     <!--当 Cache 达到预定设定的最大容量时，
         按照最少使用的原则将使用频率最少的对象从缓存中清除
     <cacheModel id="userCache" type="LRU">
```

```
    <flushInterval hours="24"/>
    <flushOnExecute statement="updateUser"/>
    <property name="size" value="1000"/>
</cacheModel>
-->
<!-- FIFO 型 cache 样例  -->
<!--先进先出型缓存，最先放入 Cache 中的数据将最先废除
<cacheModel id="userCache" type="FIFO">
    <flushInterval hours="24"/>
    <flushOnExecute statement="updateUser"/>
    <property name="size" value="1000"/>
</cacheModel>
-->
<!-- OSCache 样例  -->
<!--OSCache 来自第三方组织 Opensymphony,
    拥有自己的配置文件 oscache.properites,支持多机负载均衡
<cacheModel id="userCache" type="OSCACHE">
    <flushInterval hours="24"/>
    <flushOnExecute statement="updateUser"/>
    <property name="size" value="1000"/>
</cacheModel>
-->
<transactionManager type="JDBC">
     <dataSource type="SIMPLE">
          <property name="JDBC.Driver" value="${jdbc.driverClassName}"/>
          <property name="JDBC.ConnectionURL" value="${jdbc.url}"/>
          <property name="JDBC.Username" value="${jdbc.username}"/>
          <property name="JDBC.Password" value="${jdbc.password}"/>
          <property name="Pool.MaximumActiveConnections" value="10"/>
          <property name="Pool.MaximumIdleConnections" value="5"/>
          <property name="Pool.MaximumCheckoutTime" value="120000"/>
          <property name="Pool.TimeToWait" value="500"/>
          <property name="Pool.PingQuery" value="select 1 from dual"/>
          <property name="Pool.PingEnabled" value="false"/>
          <property name="Pool.PingConnectionsOlderThan" value="1"/>
          <property name="Pool.PingConnectionsNotUsedFor" value="1"/>
     </dataSource>
```

```
    </transactionManager>
    <!-- 非常简洁，将用到的 sqlMap 文件列到这儿就行了 -->
    <sqlMap resource="com/doone/module/dao/ibatis/maps/tf_role.xml" />
    <sqlMap resource="com/doone/module/dao/ibatis/maps/tf_user.xml" />
</sqlMapConfig>
```

8.3.2.1.1settings 节点

参数	描述
cacheModelsEnabled	是否启用 SqlMapclient 上的缓存机制。 建议设为”true”
enhancementEnabled	是否针对 POJO 启用字节码增强机制以及提升 getter/setter 的调整效能，避免使用 Java Reflect 所带来的性能开销。同时也为 Lazy Loading 带来性能上的提升。 建议设为”true”
errorTracingEnabled	是否启用延迟加载机制，建议设为”true”
lazyLoadingEnabled	是否启用延迟加载机制，建议设为”true”
maxRequests	最大并发请求数(Statement 并发数)
maxTransactions	最大并发事务数
maxSessions	最大 Session 数。即当前允许的并发 SqlMapClient 数。maxSessions 必须基于 maxTransactions 和 maxRequests 之间。maxTransactions<maxSessions<maxRequests
useStatementNamespaces	是否使用 Statement 命名空间。这里的命名空间指的是映射文件中,sqlMap 节点的命名空间 namespace 属性， 如果设为”true”，且<sqlMap namespace=”User”>， 那么调用是 sqlMap.update(“User.insertuserDAO”,user)

8.3.2.1.2transactionManager

transactionManager 节点定义了 ibatis 的事务管理器，这里我们采用 Spring 加以控制给以忽略。

8.3.2.1.3dataSource 节点

dataSource 从属于 transactionManager，用于设定 ibatis 运行期的 DataSource，这里我们采用 Spring 加以控制给以忽略。

8.3.2.1.4sqlMap 节点

sqlMap 节点指定了映射文件的位置，配置中可以出现多个 sqlMap 节点，以指定项目内所包含的所有映射文件。公司规定映射文件必须在 com/doone/[模块名]/dao/ibatis/maps/路径下。

8.3.2.2 jdbc.properties

这个文件指定了 DataSource 所需要的参数。Spring 下忽略。

```
jdbc.driverClassName=oracle.jdbc.driver.OracleDriver
jdbc.url=jdbc:oracle:thin:@localhost:1521:XE
jdbc.username=webwork
jdbc.password=webwork
jdbc.maxActive=3
jdbc.maxIdle=1
jdbc.maxWait=5000
```

8.3.2.3 映射文件

8.3.2.3.1tf_user.xml

```
<?xml version="1.0" encoding="gb2312" ?>
<!DOCTYPE sqlMap
   PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
   "http://www.ibatis.com/dtd/sql-map-2.dtd">
<sqlMap namespace="TfUser">
      <select id="findTfUserDao"
     resultClass="com.doone.module.dto.TfUser">
```

```
(3)        <![CDATA[
(4)         select ID AS ID,
             USERNAME AS USERNAME,
             PASSWORD AS PASSWORD,
             AUTOLOGIN AS AUTOLOGIN,
            ROLEID AS ROLEID
             from TF_USER
             ]]>
(5)        <dynamic prepend="where">
               <isPropertyAvailable prepend="" property="id" >
                   <isNotNull prepend="and" property="id">
(6)                    <![CDATA[ID=#id#]]>
                   </isNotNull>
               </isPropertyAvailable>
               <isPropertyAvailable prepend="" property="username" >
                   <isNotNull prepend="and" property="username">
(7)                    <![CDATA[USERNAME=#username#]]>
                   </isNotNull>
               </isPropertyAvailable>
               <isPropertyAvailable prepend="" property="password" >
                   <isNotNull prepend="and" property="password">
                   <![CDATA[PASSWORD=#password#]]>
                   </isNotNull>
               </isPropertyAvailable>
               <isPropertyAvailable prepend="" property="autologin" >
                   <isNotNull prepend="and" property="autologin">
                   <![CDATA[AUTOLOGIN=#autologin#]]>
                   </isNotNull>
               </isPropertyAvailable>
               <isPropertyAvailable prepend="" property="roleid" >
                   <isNotNull prepend="and" property="roleid">
                   <![CDATA[ROLEID=#roleid#]]>
                   </isNotNull>
               </isPropertyAvailable>
           </dynamic>
       </select>

(1)    <insert id="insertTfUserDao"
```

```
(2)      parameterClass="com.doone.module.dto.TfUser">
           <![CDATA[
           insert into TF_USER( ID, USERNAME, PASSWORD, AUTOLOGIN, ROLEID)
           values(#id#, #username#, #password#, #autologin#, #roleid#)
           ]]>
    </insert>

    <update id="updateTfUserDao"
           parameterClass="com.doone.module.dto.TfUser">
           <![CDATA[
           update TF_USER set ID=#id#, USERNAME=#username#,
           PASSWORD=#password#, AUTOLOGIN=#autologin#, ROLEID=#roleid#
           where ID=#id#
           ]]>
    </update>
    <delete id="deleteTfUserDao" parameterClass="com.doone.module.dto.TfUser">
           <![CDATA[
           delete from TF_USER where ID=#id#
           ]]>
    </delete>
</sqlMap>
```

(1)ID

指定了操作 ID，之后我们可以在代码中通过指定操作 id 来执行此节点所定义的操作，如：

```
getSqlMapClientTemplate().update("insertTfUserDao",tfuser);
```

ID 设定使得在一个配置文件中定义两个不同名节点成为可能，(两个 select 节点，以不同 ID 区分)。

(2)paramterClass

指定了操作所需的参数类型，此例中 insert 操作使用了 DTO 定义的以 com.doone.module.dto.TfUser 类型对象作为参数，目标是将提供 TfUser 实例更新到数据库中。当然我们还可以取一个别名 user，即 paramterClass=”user”，如示例配置文件中添加：

```
<typeAlias alias=”user” type=” com.doone.module.dto.TfUser”/>
```

(3)<![CDATA[……]]>

通过<![CDATA[……]]>节点，可以避免 SQL 中和 XML 规范相冲突的字符

对 XML 映射文件的合法性造成的影响。

(4)执行插入操作的 SQL，这里的 SQL 即实际数据库支持的 SQL 语句，有 ibatis 填入参数后交给数据库执行。

(5)dynamic 是告诉 ibatis，以下是 SQL 是动态部分，如果非空时会自动拼写，一般用于查询的动作。

(6)SQL 中所需的 ID 参数，”#id”在运行期有传入的 user 对象的 id 属性填充。

(7)SQL 中所需的用户名参数，”#username”在运行期有传入的 user 对象的 username 属性填充。其它类似。

8.3.2.3.2数据关联

在实际开发中，我们常常遇到关联数据的情况，如 TfUser 对象拥有若干 TfAddress 对象，每个对象描述了对应 TfUser 的一个联系地址，这种情况下，我们应该如何处理？

ibatis 中，提供了 Statement 嵌套支持，通过 Statement 嵌套，我们即可实现关联数据的操作：

8.3.2.3.2.1 一对多

```
<sqlMap namespace="TfUser">
<typeAlias alias=”user” type=”com.doone.module.dto.TfUser”/>
<typeAlias alias=” address” type=”com.doone.module.dto.TfAddress”/>
<resultMap id=”get_user_result” class=”user”>
   <result property=”id” column=”id”/>
   <result property=”name” column=”name”/>
   <result property=”sex” column=”sex”/>
   <result property=”addresses” column=”id” select=”User.getAddressByUserId”/>
   </resultMap>
<select id=”getUsers” parameterClass=”java.lang.String”
   resultMap=”get_user_result”>
   <![CDATE[
   select id,name,sex from tf_user where id=#id#
```

```
  ]]>
</select>
<select id="getAddressByUserId" parameterClass="int" resultClass="address">
  <![CDATA[
  select address,zipcode from tf_address where user_id=#userid#
  ]]>
</select>
</sqlMap>
```

8.3.2.3.2.2 一对一

```
<sqlMap namespace="TfUser">
<typeAlias alias="user" type="com.doone.module.dto.TfUser"/>
<typeAlias alias=" address" type="com.doone.module.dto.TfAddress"/>
<resultMap id="get_user_result" class="user">
  <result property="id" column="id"/>
  <result property="name" column="name"/>
  <result property="sex" column="sex"/>
  <result property="addresses" column="id" select="tf_address.address"/>
  <result property="zipcode" column="id" select="tf_address.zipcode"/>
  </resultMap>
<select id="getUsers" parameterClass="java.lang.String"
  resultMap="get_user_result">
  <![CDATE[
  select * from tf_user,tf_address where tf_user.id=tf_address.user_id
  ]]>
</select>
</sqlMap>
```

8.3.2.3.3存储过程调用

在模式要求下，业务层应该完成所有的业务操作，但是有很多情况下可能要去调用一个存储过程，原因是我们很多的系统和其他系统有大量的接口，比如计费系统，基本上就是开放一些存储过程给我们。我们不可能要求改造计费系统以适应我们的要求，那么我们就应该适应这种接口。

我们将存储过程归入持久层，那么我们在业务层调用时，同样需要进行抽象方法封装。例如：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap
    PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
    "http://www.ibatis.com/dtd/sql-map-2.dtd";>
<sqlMap namespace="tfUser">
<typeAlias alias="user" type="com.doone.module.dto.TfUser"/>
<resultMap id="userResult" class="user">
        <result property="id" column="ID" />
        <result property="username" column="USERNAME" />
        <result property="password" column="PASSWORD" />
        <result property="autologin" column="AUTOLOGIN" />

</resultMap>

<parameterMap id="userInput" class="map" >
        <parameter property="PASSWORD" jdbcType="long"
         javaType="java.lang.Long" mode="IN"/>
        <parameter property="USERNAME" jdbcType="long"
         javaType="java.lang.Long" mode="IN"/>
        <parameter property="PA_OUT_RETCURSOR"
         jdbcType="ORACLECURSOR"
         javaType="java.sql.ResultSet" mode="OUT"/>
</parameterMap>

<procedure id="getUserDetail" parameterMap="userInput"
resultMap="userResult">
{ call PR_RS_GET_USER_DETAILS (?, ?) }
</procedure>
</sqlMap>
```

参考的存储过程如下：

```
CREATE OR REPLACE PROCEDURE PR_RS_GET_USER_DETAILS (
    PA_IN_USERNAME      IN      VARCHAR2,
    PA_IN_PASSWORD      IN      VARCHAR2,
```

```
    PA_OUT_RETCURSOR    OUT          RS_GENERAL.REFCURSOR
)
IS

/BEGIN
    OPEN PA_OUT_RETCURSOR FOR
        SELECT *
        FROM    TF_USER
        WHERE    USERNAME = PA_IN_USERNAME
        AND       PASSWORD = PA_IN_PASSWORD;
END PR_RS_GET_USER_DETAILS;
/
```

9 尾声

全文我们一直表述微框架理论，即把 WebWork、Spring 和 iBATIS 当做一个微框架，是基于一个分层的理论，WebWork 在表现层上发挥作用可以由 Struts 替代，iBATIS 在持久层上发挥的作用可以有 Hibernate 替代，这种理论能够将一个很复杂的技术框架变成很简单的组合，那么我们可以在每一层次跟上技术最先进的技术发展，灵活的进行拆分和组合，而和我们的业务没有任何关系。使我们的开发者更多的去充当设计师的角色攻克业务上的问题，而不仅仅受困于技术的实践。如果本文能够给你带来帮助，那就是一件很愉快的事情。