OpenDoc Series’

Webwork2 开发指南

V1.0

作者：夏昕、唐勇

So many open source projects. Why not **Open** your **Doc**uments? **J**

文档说明

参与人员：

作者	联络
夏昕	xiaxin(at)gmail.com
唐勇	jlinux(at)gmail.com

(at) 为 email @ 符号

发布记录

版本	日期	作者	说明
0.9	2004.10.10	夏昕	创建
1.0	2004.10.15	唐勇	补充“Webwork 配置说明”部分

Open Doc Series 目前包括以下几份文档：

- **Spring 开发指南**
- **Hibernate 开发指南**
- **ibatis2 开发指南**
- **Webwork2 开发指南**

以上文档可从 http://blog.csdn.net/nuke 获取最新更新信息

目录

WebWork2 开发指南

很长一段时间内， OpenSymphony 作为一个开源组织，其光辉始终被 Apache 所掩盖。Java 程序员热衷于 Apache 组织 Struts 项目研讨之后，往往朦朦胧胧的感觉到，似乎还有另外一个框架正在默默的发展。

这种朦胧的感觉，则可能来自曾经在国内流行一时的论坛软件－Jive Forum。

很多软件技术人员不惜从各种渠道得到 Jive 的源代码，甚至是将其全部反编译以探其究竟。作为一个论坛软件能受到技术人员如此垂青，想必作者睡梦中也会乐醒。**J**

而 WebWork，就是 Jive 中，MVC 实现的核心[1]。

这里我们所谈及的 WebWork，实际上是 Webwork+XWork 的总集，Webwork1.x 版本中，整个框架采用了紧耦合的设计（类似 Struts），而 2.0 之后，Webwork 被拆分为两个部分，即 Webwork 2.x +XWork 1.x，设计上的改良带来了系统灵活性上的极大提升。这一点我们稍后讨论。

Webwork 发行包中的文档并不是很全面，如果开发中遇到什么问题，登录 Webwork Wiki 站点查看在线文档是个不错的选择：

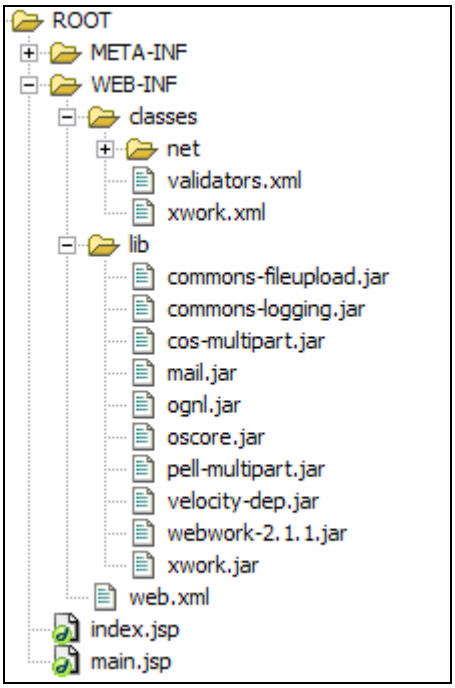
http://www.opensymphony.com/webwork/wikidocs

[1] Jive 对 WebWork 的源代码进行了重新封装，主要是包结构上的变化，如 com.opensymphony.webwork 在 Jive 中被修改为 com.jivesoftware.webwork，核心功能并没有太大改变

Quick Start

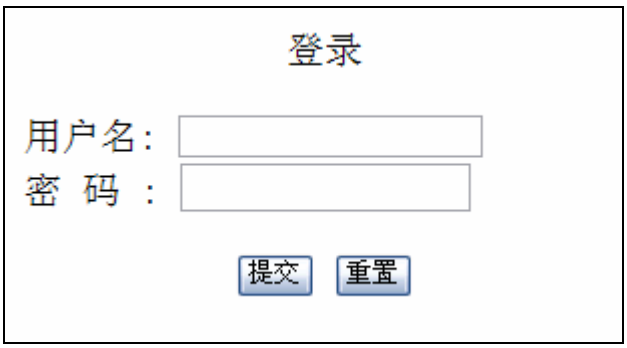
准备工作：首先下载 WebWork2 的最新版本(http://www.opensymphony.com/webwork/)。

WebWork2 发行包中的\lib\core 目录下包含了 WebWork2 用到的核心类库。将\webwork-2.1.1.jar 以及\lib\core*.jar 复制到 Web 应用的 WEB-INF\lib 目录。本例的部署结构如图所示：



这里我们选择了一个最常见的登录流程来演示 Webwork2 的工作流程。虽然简单，但足以明理。

页面流程如下：



登录成功页面，显示几条通知消息：

登录失败页面：

只是一个简单不过的登录流程，然而在这个简单的例子里，贯穿了 Webwork 框架的大部分应用技术。我们将 MVC 框架的运作流程拆分为以下几部分加以讨论：

1. 将 Web 页面中的输入元素封装为一个（请求）数据对象。
2. 根据请求的不同，调度相应的逻辑处理单元，并将（请求）数据对象作为参数传入。
3. 逻辑处理单元完成运算后，返回一个结果数据对象。
4. 将结果数据对象中的数据与预先设计的表现层相融合并展现给用户。

首先来看登录界面：

index.jsp

```
<html>
<body>
<form action="/login.action">

    <p align="center">
        登录<br>
    </p>

    用户名:
    <input type="text" name="model.username" />
    <br>
```

```
    密  码：
    <input type="password" name="model.password" />
    <br>

    <p align="center">
           <input type="submit" value="提交" name="B1"/>
           <input type="reset" value="重置" name="B2"/>
    </p>

</form>
</body>
</html>
```

这里的 index.jsp 实际上是由纯 html 组成，非常简单，其中包含一个表单：

```
    <form action="/login.action">
```

这表明其提交对象为/login.action

表单中同时包含两个文本输入框，

```
    <input type="text" name="model.username" />
    <input type="password" name="model.password" />
```

可以看到，两个输入框的名称均以“model”开头，这是因为在这里我们采用了 WebWork 中 Model-Driven 的 Action 驱动模式。这一点稍后再做介绍。

当表单被提交之时，浏览器会以两个文本框的值作为参数，向 Web 请求以/login.action 命名的服务。

标准 HTTP 协议中并没有.action 结尾的服务资源。我们需要在 web.xml 中加以设定：

```
.....
<servlet>
    <servlet-name>webwork</servlet-name>
       <servlet-class>
         com.opensymphony.webwork.dispatcher.ServletDispatcher
       </servlet-class>
    </servlet>

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

```
.....
```

此后，所有以.action 结尾的服务请求将由 ServletDispatcher 接管。

ServletDispatcher 接受到 Servlet Container 传递过来的请求，将进行一下几个动作：

1. 从请求的服务名（/login.action）中解析出对应的 Action 名称（login）
2. 遍历 HttpServletRequest、HttpSession、ServletContext 中的数据，并将其复制到 Webwork 的 Map 实现中，至此之后，所有数据操作均在此 Map 结构中进行，从而将内部结构与 Servlet API 相分离。

   至此，Webwork 的工作阶段结束，数据将传递给 XWork 进行下一步处理。从这里也可以看到 Webwork 和 xwork 之间的切分点，Webwork 为 xwork 提供了一个面向 Servlet 的协议转换器，将 Servlet 相关的数据转构转换成 xwork 所需要的通用数据格式，而 xwork 将完成实际的服务调度和功能实现。
   这样一来，以 xwork 为核心，只需替换外围的协议转换组件，即可实现不同技术平台之间的切换（如将面向 Servlet 的 Webwork 替换为面向 JMS 的协议转换器实现，即可在保留应用逻辑实现的情况下，实现不同外部技术平台之间的移植）。

3. 以上述信息作为参数，调用 ActionProxyFactory 创建对应的 ActionProxy 实例。ActionProxyFactory 将根据 Xwork 配置文件（xwork.xml）中的设定，创建 ActionProxy 实例，ActionProxy 中包含了 Action 的配置信息（包括 Action 名称，对应实现类等等）。

4. ActionProxy 创建对应的 Action 实例，并根据配置进行一系列的处理程序。包括执行相应的预处理程序（如通过 Interceptor 将 Map 中的请求数据转换为 Action 所需要的 Java 输入数据对象等），以及对 Action 运行结果进行后处理。ActionInvocation 是这一过程的调度者。而 com.opensymphony.xwork.DefaultActionInvocation 则是 XWork 中对 ActionInvocation 接口的标准实现，如果有精力可以对此类进行仔细研读，掌握了这里面的玄机，相信 XWork 的引擎就不再神秘。

下面我们来看配置文件：
xwork.xml：

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

<xwork>
    <include file="webwork-default.xml" />                    (1)
    <package name="default" extends="webwork-default">        (2)

        <action name="login"                                  (3)
            class="net.xiaxin.webwork.action.LoginAction">
            <result name="success" type="dispatcher">         (4)
                <param name="location">/main.jsp</param>
```

```
            </result>

            <result name="loginfail" type="dispatcher">
                <param name="location">/index.jsp</param>
            </result>

            <interceptor-ref name="params" />                    (5)
            <interceptor-ref name="model-driven"/>               (6)

        </action>

    </package>
</xwork>
```

(1) include

通过 include 节点，我们可以将其他配置文件导入到默认配置文件 xwork.xml 中。从而实现良好的配置划分。

这里我们导入了 Webwork 提供的默认配置 webwork-default.xml（位于 webwork.jar 的根路径）。

(2) package

XWork 中，可以通过 package 对 action 进行分组。类似 Java 中 package 和 class 的关系。为可能出现的同名 Action 提供了命名空间上的隔离。

同时，package 还支持继承关系。在这里的定义中，我们可以看到：

```
extends="webwork-default"
```

"webwork-default"是 webwork-default.xml 文件中定义的 package，这里通过继承，"default" package 自动拥有"webwork-default" package 中的所有定义关系。

这个特性为我们的配置带来了极大便利。在实际开发过程中，我们可以根据自身的应用特点，定义相应的 package 模板，并在各个项目中加以重用，无需再在重复繁琐的配置过程中消耗精力和时间。

此外，我们还可以在 Package 节点中指定 namespace，将我们的 action 分为若干个逻辑区间。如：

```
<package name="default" namespace="/user"
                    extends="webwork-default">
```

就将此 package 中的 action 定义划归为/user 区间，之后在页面调用 action 的时候，我们需要用/user/login.action 作为 form action 的属性值。其中的/user/就指定了此 action 的 namespace，通过这样的机制，我们可以将系统内的 action 进行逻辑分类，从而使得各模块之间的划分更加清晰。

(3) action

Action 配置节点，这里可以设定 Action 的名称和对应实现类。

⑷ result

通过 result 节点，可以定义 Action 返回语义，即根据返回值，决定处理模式以及响应界面。

这里，返回值"success"（Action 调用返回值为 String 类型）对应的处理模式为"dispatcher"。

可选的处理模式还有：

1. dispatcher
   本系统页面间转向。类似 forward。
2. redirect
   浏览器跳转。可转向其他系统页面。
3. chain
   将处理结果转交给另外一个 Action 处理，以实现 Action 的链式处理。
4. velocity
   将指定的 velocity 模板作为结果呈现界面。
5. xslt
   将指定的 XSLT 作为结果呈现界面。

随后的 param 节点则设定了相匹配的资源名称。

⑷ interceptor-ref

设定了施加于此 Action 的拦截器（interceptor）。关于拦截器，请参见稍后的“XWork 拦截器体系”部分。

interceptor-ref 定义的是一个拦截器的应用，具体的拦截器设定，实际上是继承于 webwork-default package，我们可以在 webwork-default.xml 中找到对应的"params"和"model-driven"拦截器设置：

```
<interceptors>
    ……
    <interceptor name="params"
        class="com.opensymphony.xwork.interceptor.ParametersInterceptor" />
    <interceptor name="model-driven"
        class="com.opensymphony.xwork.interceptor.ModelDrivenInterceptor" />
    ……
</interceptors>
```

"params"大概是 Webwork 中最重要、也最常用的一个 Interceptor。上面曾经将 MVC 工作流程划分为几个步骤，其中的第一步:

“将 Web 页面中的输入元素封装为一个（请求）数据对象”

就是通过"params"拦截器完成。Interceptor 将在 Action 之前被调用，因而，Interceptor 也成为将 Webwork 传来的 MAP 格式的数据转换为强类型 Java 对象的

最佳实现场所。

"model-driven"则是针对 Action 的 Model 驱动模式的 interceptor 实现。具体描述请参见稍后的“Action 驱动模式”部分

很可能我们的 Action 都需要对这两个 interceptor 进行引用。我们可以定义一个 interceptor-stack，将其作为一个 interceptor 组合在所有 Action 中引用。如，上面的配置文件可修改为：

```
<xwork>
  <include file="webwork-default.xml" />
  <package name="default" extends="webwork-default">
      <interceptors>
          <interceptor-stack name="modelParamsStack">
              <interceptor-ref name="params" />
              <interceptor-ref name="model-driven" />
          </interceptor-stack>
      </interceptors>

      <action name="login"
          class="net.xiaxin.webwork.action.LoginAction">
          <result name="success" type="dispatcher">
              <param name="location">/main.jsp</param>
          </result>

          <result name="loginfail" type="dispatcher">
              <param name="location">/index.jsp</param>
          </result>

          <interceptor-ref name="modelParamsStack" />

      </action>
  </package>
</xwork>
```

通过引入 interceptor-stack，我们可以减少 interceptor 的重复申明。

下面是我们的 Model 对象：

LoginInfo.java：

```
public class LoginInfo {
  private String password;
  private String username;
  private List messages = new ArrayList();
  private String errorMessage;
```

```
  public List getMessages() {
      return messages;
  }

  public String getErrorMessage() {
      return errorMessage;
  }

  public void setErrorMessage(String errorMessage) {
      this.errorMessage = errorMessage;
  }

  public String getPassword() {
      return password;
  }

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

  public String getUsername() {
      return username;
  }

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

}
```

很简单，这只是一个纯粹的值对象（Value-Object）。这里，它扮演着模型（Model）的角色，并与 Action 的输入输出密切相关。

与 Spring MVC 中的 Command 对象不同，Webwork 中的 Model 对象，扮演着承上启下的角色，它既是 Action 的输入参数，又包含了 Action 处理的结果数据。

换句话说，输入的 Http 请求参数，将被存储在 Model 对象传递给 Action 进行处理，Action 处理完毕之后，也将结果数据放置到 Model 对象中，之后，Model 对象与返回界面融合生成最后的反馈页面。

也正由于此，笔者建议在实际开发中采用 Model-Driven 模式，而非 Property-Driven 模式（见稍后“Action 驱动模式”部分），这将使得业务逻辑更加清晰可读。

对应的 Action 代码

```
public class LoginAction implements Action, ModelDriven {
```

```
    private final static String LOGIN_FAIL="loginfail";

    LoginInfo loginInfo = new LoginInfo();

    public String execute() throws Exception {

        if ("erica".equalsIgnoreCase(loginInfo.getUsername())
                && "mypass".equals(loginInfo.getPassword())) {

            //将当前登录的用户名保存到Session
            ActionContext ctx = ActionContext.getContext();
            Map session = ctx.getSession();
            session.put("username",loginInfo.getUsername());

            //出于演示目的，通过硬编码增加通知消息以供显示
            loginInfo.getMessages().add("message1");
            loginInfo.getMessages().add("message2");
            loginInfo.getMessages().add("message3");

            return SUCCESS;
        }else{
            loginInfo.setErrorMessage("Username/Password Error!");
            return LOGIN_FAIL;
        }
    }

    public Object getModel() {
        return loginInfo;
    }
}
```

可以看到，LoginAction 实现了两个接口：

1. Action

   Action 接口非常简单，它指定了 Action 的入口方法（execute），并定义了几个默认的返回值常量：

```
public interface Action extends Serializable {
    public static final String SUCCESS = "success";
    public static final String NONE = "none";
    public static final String ERROR = "error";
    public static final String INPUT = "input";
    public static final String LOGIN = "login";

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

SUCCESS、NONE、ERROR、INPUT、LOGIN 几个字符串常量定义了常用的几类返回值。我们可以在 Action 实现中定义自己的返回类型，如本例中的 LOGIN_FAIL 定义。
而 execute 方法，则是 Action 的入口方法，XWork 将调用每个 Action 的 execute 方法以完成业务逻辑处理。

2. ModelDriven
ModelDriven 接口更为简洁：

```
public interface ModelDriven {
    Object getModel();
}
```

ModelDriven 仅仅定义了一个 getModel 方法。XWork 在调度 Action 时，将通过此方法获取 Model 对象实例，并根据请求参数为其设定属性值。而此后的页面返回过程中，XWork 也将调用此方法获取 Model 对象实例并将其与设定的返回界面相融合。

注意这里与 Spring MVC 不同，Spring MVC 会自动为逻辑处理单元创建 Command Class 实例，但 Webwork 不会自动为 Action 创建 Model 对象实例，Model 对象实例的创建需要我们在 Action 代码中完成（如 LoginAction 中 LoginInfo 对象实例的创建）。

另外，如代码注释中所描述，登录成功之后，我们随即将 username 保存在 Session 之中，这也是大多数登录操作必不可少的一个操作过程。
这里面牵涉到了 Webwork 中的一个重要组成部分：ActionContext。

ActionContext 为 Action 提供了与容器交互的途径。对于 Web 应用而言，与容器的交互大多集中在 Session、Parameter，通过 ActionContext 我们在代码中实现与 Servlet API 无关的容器交互。

如上面代码中的：

```
ActionContext ctx = ActionContext.getContext();
Map session = ctx.getSession();
session.put("username",loginInfo.getUsername());
```

同样，我们也可以操作 Parameter:

```
ActionContext ctx = ActionContext.getContext();
Map params = ctx.getParameters();
String username = ctx.getParameters("username");
```

上述的操作，将由 XWork 根据当前环境，调用容器相关的访问组件（Web 应用对应的就是 Webwork）完成。上面的 ActionContext.getSession()，XWork 实际上将通过 Webwork 提供的容器访问代码“HttpServletRequest.getSession()”完成。

注意到，ActionContext.getSession 返回的是一个 Map 类型的数据对象，而非 HttpSession。这是由于 WebWork 对 HttpSession 进行了转换，使其转变为与 Servlet API 无关的 Map 对象。通过这样的方式，保证了 Xwork 所面向的是一个通用的开放结构。从而使得逻辑层与表现层无关。增加了代码重用的可能。

此外，为了提供与 Web 容器直接交互的可能。WebWork 还提供了一个 ServletActionContext 实现。它扩展了 ActionContext，提供了直接面向 Servlet API 的容器访问机制。

我们可以直接通过 ServletActionContext.getRequest 得到当前 HttpServletRequest 对象的引用，从而直接与 Web 容器交互。

获得如此灵活性的代价就是，我们的代码从此与 ServletAPI 紧密耦合，之后系统在不同平台之间移植就将面临更多的挑战（同时单元测试也难于进行）。

平台移植的需求并不是每个应用都具备。大部分系统在设计阶段就已经确定其运行平台，且无太多变更的可能。不过，如果条件允许，尽量通过 ActionContext 与容器交互，而不是平台相关的 ServletActionContext，这样在顺利实现功能的同时，也获得了平台迁移上的潜在优势，何乐而不为。

登录成功界面：

main.jsp:

```
<%@ taglib prefix="ww" uri="webwork"%>
<html>
<body>
    <p align="center">Login Success!</p>
    <p align="center">Welcome!
        <ww:property value="#session['username']"/>
    </p>
    <p align="center">
        <b>Messages:</b><br>
        <ww:iterator value="messages" status="index">
         <ww:if test="#index.odd == true">
           !<ww:property/><br>
         </ww:if>
         <ww:else>
           *<ww:property/><br>
         </ww:else>
        </ww:iterator>
    </p>
</body>
</html>
```

这里我们引入了 Webwork 的 taglib，如页面代码第一行的申明语句。

下面主要使用了三个 tag:

Ø `<ww:property value="#session['username']"/>`

读取 Model 对象的属性填充到当前位置。

`value` 指定了需要读取的 Model 对象的属性名。

这里我们引用了 LoginAction 在 session 中保存的'username'对象。

由于对应的 Model（LoginInfo）中也保存了 username 属性。下面的语句与之效果相同：

```
<ww:property value="username"/>
```

与 JSP2 中的 EL 类似，对于级联对象，这里我们也可以通过“.”操作符获得其属性值，如 `value="user.username"`将得到 Model 对象中所引用的 user 对象的 username 属性（假设 LoginInfo 中包含一个 User 对象，并拥有一个名为“username”的属性）。

关于 EL 的内容比较简单，本文就不再单独开辟章节进行探讨。

Webwork 中包括以下几种特殊的 EL 表达式：

- parameter['username'] 相当于 request.getParameter("username");
- request['username'] 相当于 request.getAttribute("username");
- session['username'] 从 session 中取出以“username”为 key 的值
- application['username'] 从 ServletContext 中取出以“username”为 key 的值

注意需要用“#”操作符引用这些特殊表达式。

另外对于常量，需要用单引号包围，如`#session['username']` 中的 `'username'`。

Ø `<ww:iterator value="messages" status="index">`

迭代器。用于对 java.util.Collection、java.util.Iterator、java.util.Enumeration,、java.util.Map、Array 类型的数据集进行循环处理。

其中，`value` 属性的语义与`<ww:property>`中一致。

而 `status` 属性则指定了循环中的索引变量，在循环中，它将自动递增。

而在下面的`<ww:if>`中，我们通过“`#`”操作符引用这个索引变量的值。

索引变量提供了以下几个常用判定方法：

- first 当前是否为首次迭代
- last 当前是否为最后一次迭代
- odd 当前迭代次数是否奇数
- even 当前迭代次数是否偶数

Ø `<ww:if test="#index.odd == true">`和`<ww:else>`

用于条件判定。

`test` 属性指定了判定表达式。表达式中可通过“`#`”操作符对变量进行引用。

表达式的编写语法与 java 表达式类似。

类似的，还有`<ww:elseif test=".....">`。

登录失败界面

实际上，这个界面即登录界面 index.jsp。只是由于之前出于避免干扰的考虑，隐藏了 index.jsp 中显示错误信息的部分。

完整的 index.jsp 如下：

```
<%@ page pageEncoding="gb2312"
contentType="text/html;charset=gb2312"%>
<%@ taglib prefix="ww" uri="webwork"%>
<html>
<body>
<form action="/login.action">

    <p align="center">
        登录<br>
        <ww:if test="errorMessage != null">
            <font color="red">
                <ww:property value="errorMessage"/>
            </font>
        </ww:if>

    </p>

    用户名:
    <input type="text" name="model.username" />
    <br>

    密　码：
    <input type="password" name="model.password" />
    <br>

    <p align="center">
           <input type="submit" value="提交" name="B1"/>
           <input type="reset" value="重置" name="B2"/>
    </p>

</form>
</body>
</html>
```

这里首先我们进行判断，如果 Model 中的 errorMessage 不为 null，则显示错误信息。这样，在用户第一次登录时，由于 Model 对象尚未创建，errorMessage 自然为 null，错误信息不会显示，即得到了与之前的 index.jsp 同样的效果。

WebWork 高级特性

Action 驱动模式

Webwork 中，提供了两种 Action 驱动模式：

1. Property-Driven
2. Model-Driven

上面的示例中，我们应用了 Model-Driven 的 Action 驱动模式。对于这种模式，相信大家已经比较熟悉。

下面介绍一下 Property-Driven 驱动模式。

Model-Driven 是通过 Model 对象（上例中的 LoginInfo）贯穿整个 MVC 流程，而 Property-Driven，顾名思义， 是由 Property（属性）作为贯穿 MVC 流程的信息携带者。

Property 固然无法独立存在，它必须依附于一个对象。在 Model-Driven 模式中，实际上这些属性被独立封装到了一个值对象，也就是所谓的 Model。而 Property-Driven 中，属性将依附在 Action 对象中。

对上例进行一些小改造：

LoginActoin.java

```
public class LoginAction implements Action{

    private final static String LOGIN_FAIL="loginfail";

    private String password;
    private String username;
    private List messages = new ArrayList();
    private String errorMessage;

    public List getMessages() {
        return messages;
    }

    public String getErrorMessage() {
        return errorMessage;
    }

    public void setErrorMessage(String errorMessage) {
        this.errorMessage = errorMessage;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
```

```
        this.password = password;
    }

    public String getUsername() {
        return username;
    }

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

    public String execute() throws Exception {
        if ("erica".equalsIgnoreCase(getUsername())
                && "mypass".equals(getPassword())) {
            getMessages().add("message1");
            getMessages().add("message2");
            getMessages().add("message3");
            return SUCCESS;
        }else{
            setErrorMessage("Username/Password Error!");
            return LOGIN_FAIL;
        }
    }
}
```

可以发现，LoginAction 无需再实现 ModelDriven 接口，而原本作为 Model 的 LoginInfo 类的属性，亦被转移到 LoginAction 之中。

在 Property-Driven 模式中，不在有单独的 Model 存在，取而代之的是一个 Model 和 Action 融合之后得到的整体类结构。

其他需要修改的还有：

index.jsp

```
    ……
    用户名:
    <input type="text" name="username" />
    <br>

    密  码 :
    <input type="password" name="password" />
    <br>
    ……
```

xwork.xml（删除对 model-driven Interceptor 的引用）：

```
    ……
    <action name="login"
        class="net.xiaxin.webwork.action.LoginAction">
```

```
        <result name="success" type="dispatcher">
            <param name="location">/main.jsp</param>
        </result>

        <result name="loginfail" type="dispatcher">
            <param name="location">/index.jsp</param>
        </result>
        <interceptor-ref name="params" />
    </action>
    ……
```

这样，我们的程序就可以以 Property-Driven 的模式运行。

Property-Driven 和 Model-Driven 模式的比较：

从上面改造后的例子可以感觉到，Property-Driven 驱动模式似乎更加简单，无需再实现 ModelDriven 接口。也减少了一个 Model 类。Jsp 文件和 xwork.xml 也有了些许简化。

出于对框架灵活性的考虑，Webwork 提供了如上两种驱动模式。但对于我们的应用系统而言，这样的灵活性有时反而使得我们有点无所适从－－使用 Property-Driven 模式似乎更加简单自由，而使用 Model-Driven 模式又似乎更加清晰……

Webwork 为我们提供了更多的选择和更大的自由度。面对这些选项我们该如何抉择？

就笔者的观点而言。Model-Driven 对于维持软件结构清晰性的贡献，超过了其所带来的微不足道的复杂性。

记得关于面向对象设计法则的著作《Object-Oriented Design Heuristics》[2]中有这样一个有趣的问题（可能与原文在文字细节上有所差异，一时找不到原书，只能凭回忆描述）：

> 当你面对一头奶牛的时候，你会对它说“请给我挤一杯牛奶”。
> 还是对身边的农场工人说“请给我挤一杯牛奶”？

大多程序员对于这样的提问都会不屑一顾，愚蠢的问题，不是么？

不过在软件开发过程中，程序员们却常常不停的对着奶牛大喊“挤牛奶！挤牛奶！”。

回头看看这里的 Property-Driven 模式，是不是也有点这样的味道……

作为贯穿 WebWork MVC 的信息载体，Model 扮演着奶牛的角色，它携带了我们所需要的数据资源（牛奶）。而如何操作这些数据，却不是奶牛的任务，而是农场工人（Action）

[2] 中文版已经由人民邮电出版社出版，名为《OOD 启思录》

的工作。

如此一来，Property-Driven 模式的身份似乎就有点混杂不清。

这也就是笔者所想要表达的意思，Webwork 出于框架灵活性的考虑，提供了 Property-Driven 模式供用户选择，但作为用户的我们，还是需要有着一定取舍原则，这里，笔者推荐将 Model-Driven 驱动模式作为 WebWork 开发的首选。

此外，对于 Property-Driven 模式还有一种应用方法值得一提：即将业务逻辑从 Action 中抽离，而在 Action 之外的逻辑单元中提供对应的实现。

这个改进方案，借用 LoginAction 表示大致可以描述如下：

```
public class LoginAction implements Action{
    …Property getter/setter略…
    public String execute() throws Exception {
        return LoginLogic.doLogin (getUsername(),getPassword());
    }
}
```

`LoginLogic` 是系统中的逻辑实现类，它提供了一个方法 `doLogin` 以完成实际的 `Login` 工作。

如果采用以上方案，那么其中的 Action 就成为实际意义上的 Model Object。这里的 Property-Driven 模式也就实际演变成了 Model-Driven 模式，只是将逻辑层更加向下推进了一层。原本的逻辑层在 Action 中实现，现在转到了框架之外的逻辑实现单元中。

通过这样的 Property-Driven 模式，我们的 LoginAction 无需再实现 ModelDriven 接口，xwork 配置中减少了一行配置代码。

而最为重要的是，应用系统的关键所在——业务逻辑，将与框架本身相分离，这也就意味着代码的可移植性将变得更强。

是不是我们的系统中就应该采用这样的模式？

还是老话，根据实际需求而定。如果产品之后可能需要提供 Web 之外的表现渠道，那么采用上述 Property-Driven 模式是个不错的选择。否则，对于一般的 Web 应用开发而言（如网上论坛），采用 Model-Driven 模式即可。

为什么这里并不建议在所有项目中都采用上述模式？原因有几个方面：首先，Web 系统开发之后，切换框架的可能性微乎其微，而同时，Xwork 的侵入性已经很小，移植的代价并不高昂（这也是 Xwork 相对 Struts 的一个突出优势）。

此外，还有一个最重要的问题。这样的模式（业务逻辑只允许在独立的逻辑单元中实现，而不能混杂在 Action 中）需要较强的开发规范的约束，而人本的约束往往最不可靠。

规则制定人固然能保证自己遵循这个开发模式，但在团队协同开发的过程中，是否真

的能保证每个人都按照这样的模式开发，这还是一个疑问。随之而来的代码复查工作的必然增加，将带来更多的生产率损耗。

当然，具体如何抉择，仁者见仁智者见智，根据客户的真实需求选择适用的开发模式，将为您的团队带来最佳的效益比。

XWork 拦截器体系

Interceptor（拦截器），顾名思义，就是在某个事件发生之前进行拦截，并插入某些处理过程。

Servlet 2.3 规范中引入的 Filter 算是拦截器的一个典型实现，它在 Servlet 执行之前被触发，对输入参数进行处理之后，再将工作流程传递给对应的 Servlet。

而今年来兴起的 AOP（Aspect Oriented Programming），更是将 Interceptor 的作用提升到前所未有的高度。

Xwork 的 Interceptor 概念与之类似。即通过拦截 Action 的调用过程，为其追加预处理和后处理过程。

回到之前讨论过的一个问题：

"将 web 页面中的输入元素封装为一个（请求）数据对象"

对于 Xwork 而言，前端的 Webwork 组件为其提供的是一个 Map 类型的数据结构。而 Action 面向的却是 Model 对象所提供的数据结构。

在何时、何处对这两种不同的数据结构进行转换？自然，我们可以编写一个辅助类完成这样的工作，并在每次 Action 调用之前由框架代码调用他完成转换工作，就像 Struts 做的那样。

这种模式自然非常简单，不过，如果我们还需要进行其他操作，比如验证数据合法性，那么，我们又需要增加一个辅助类，并修改我们的框架代码，加入这个类的调用代码。显然不是长久之计。

Xwork 通过 Interceptor 实现了这一步骤，从而我们可以根据需要，灵活的配置所需的 Interceptor。从而为 Action 提供可扩展的预处理、后处理过程。

前面曾经提及，ActionInvocation 是 Xworks 中 Action 调度的核心。而 Interceptor 的调度，也正是由 ActionInvocation 负责。

ActionInvocation 是一个接口，而 DefaultActionInvocation 则是 Webwork 对 ActionInvocation 的默认实现。

Interceptor 的调度流程大致如下：

1. ActionInvocation 初始化时，根据配置，加载 Action 相关的所有 Interceptor
   参见 ActionInvocation.init 方法中相关代码：

```
private void init() throws Exception {
    ……
    List interceptorList = new
            ArrayList(proxy.getConfig().getInterceptors());
    interceptors = interceptorList.iterator();
}
```

2. 通过 ActionInvocation.invoke 方法调用 Action 实现时，执行 Interceptor:
   下面是 DefaultActionInvocation 中 Action 调度代码：

```
public String invoke() throws Exception {
    //调用intercaptors
    if (interceptors.hasNext()) {
        Interceptor interceptor =
                    (Interceptor) interceptors.next();
        resultCode = interceptor.intercept(this);
    } else {
        if (proxy.getConfig().getMethodName() == null) {
            resultCode = getAction().execute();
        } else {
            resultCode = invokeAction(
                                getAction(),
                                proxy.getConfig()
                            );
        }
    }
    ……
}
```

可以看到，ActionInvocation 首先会判断当前是否还有未执行的 Interceptor，如果尚有未得到执行的 Interceptor，则执行之，如无，则执行对应的 Action 的 execute 方法（或者配置中指定的方法）。

这里可能有点疑问，按照我们的理解，这里首先应该对 interceptorList 循环遍历，依次执行各 interceptor 之后，再调用 Action.execute 方法。

不过需要注意的是，类似 Servlet Filter，interceptor 之间并非只是独立的顺序关系，而是层级嵌套关系。

也就是说，Interceptor 的调用过程，是首先由外层的（如按定义顺序上的第一个）Inerceptor 调用次级的（定义顺序上的第二个）Interceptor，之后如此类推，直到最终的 Action，再依次返回。

这是设计模式“**Intercepting Filter**[3]”的一个典型应用。

同样，Servlet Filter 也是“Intercepting Filter”模式的一个典型案例，可以参照 ServletFilter 代码，进行一些类比：

```
public void doFilter(ServletRequest srequest,
                     ServletResponse sresponse,
                     FilterChain chain)
  throws IOException, ServletException {

  HttpServletRequest request = (HttpServletRequest)srequest;
  request.setCharacterEncoding(targetEncoding);
  //nest call
  chain.doFilter(srequest,sresponse);
```

[3] 关于 Intercepting Filter Pattern，请参见 http://java.sun.com/blueprints/patterns/InterceptingFilter.html

```
}
```

为了有进一步的感性认识，我们从 Interceptor 的实现机制上入手。

所有的拦截器都必须实现 Interceptor 接口：

```
public interface Interceptor {
    void destroy();
    void init();
    String intercept(ActionInvocation invocation) throws Exception;
}
```

在 Interceptor 实现中，抽象实现 AroundInterceptor 得到了最广泛的应用（扩展），它增加了预处理（before）和后处理（after）方法的定义。

AroundInterceptor.java:

```
public abstract class AroundInterceptor implements Interceptor {
    protected Log log = LogFactory.getLog(this.getClass());
    public void destroy() {
    }
    public void init() {
    }

    public String intercept(ActionInvocation invocation) throws
Exception {
        String result = null;

        before(invocation);
        result = invocation.invoke();
        after(invocation, result);

        return result;
    }

    protected abstract void after(ActionInvocation dispatcher, String
result)
        throws Exception;
    /**
     * Called before the invocation has been executed.
     */
    protected abstract void before(ActionInvocation invocation)
        throws Exception;
}
```

AroundInterceptor.invoke 方法中，调用了参数 invocation 的 invoke 方法。结合

前面 ActionInvocation.invoke 方法的实现，我们即可看出 Interceptor 层级嵌套调用的机制。

这样的嵌套调用方式并非 AroundInterceptor 所独有，通过浏览 Xwork 源代码，我们可以发现，DefaultWorkflowInterceptor 等其他的 Interceptor 实现也同样遵循这一模式。

最后，我们结合最常用的 ParametersInterceptor，看看 Xwork 是如何通过 Interceptor，将 Webwork 传入的 Map 类型数据结构，转换为 Action 所需的 Java 模型对象。

ParametersInterceptor.java:

```
public class ParametersInterceptor extends AroundInterceptor {

    protected void after(ActionInvocation dispatcher, String result)
        throws Exception {
    }

    protected void before(ActionInvocation invocation) throws Exception
{
        if (!(invocation.getAction() instanceof NoParameters)) {
            final Map parameters =
                        ActionContext.getContext().getParameters(); (1)

            if (log.isDebugEnabled()) {
                log.debug("Setting params " + parameters);
            }

            ActionContext invocationContext =
                        invocation.getInvocationContext();

            try {
                invocationContext.put(
                    InstantiatingNullHandler.CREATE_NULL_OBJECTS,
                    Boolean.TRUE);
                invocationContext.put(
                    XWorkMethodAccessor.DENY_METHOD_EXECUTION,
                    Boolean.TRUE);
                invocationContext.put(
                    XWorkConverter.REPORT_CONVERSION_ERRORS,
                    Boolean.TRUE);

                if (parameters != null) {
                    final OgnlValueStack stack =
                        ActionContext.getContext().getValueStack();   (2)

                    for (Iterator iterator =                          (3)
```

```
                        parameters.entrySet().iterator();
                        iterator.hasNext();
                    ) {
                    Map.Entry entry = (Map.Entry) iterator.next();
                    stack.setValue(                                    (4)
                        entry.getKey().toString(),
                        entry.getValue());
                }
            }
        } finally {
            invocationContext.put(
                InstantiatingNullHandler.CREATE_NULL_OBJECTS,
                Boolean.FALSE);
            invocationContext.put(
                XWorkMethodAccessor.DENY_METHOD_EXECUTION,
                Boolean.FALSE);
            invocationContext.put(
                XWorkConverter.REPORT_CONVERSION_ERRORS,
                Boolean.FALSE);
        }
    }
}
```

ParametersInterceptor 扩展了抽象类 AroundInterceptor。并在其预处理方法（before）中实现了数据的转换。

数据转换的过程并不复杂:

⑴ 首先由 ActionContext 获得 Map 型的参数集 parameters。

⑵ 由 ActionContext 获得值栈（OgnlValueStack）。

⑶ 遍历 parameters 中的各项数据。

⑷ 通过 OgnlValueStack，根据数据的键值，为 Model 对象填充属性数据。

OgnlValueStack 是 http://www.ognl.org[4]提供的一套可读写对象属性的类库（功能上有点类似 Jakarta Commons BeanUtils，以及 Spring BeanWrapper）。

OgnlValueStack 使用非常简单，下面这段示例代码将 User 对象的 name 属性设为“erica”。

```
OgnlValueStack stack = new OgnlValueStack();
stack.push(new User());//首先将欲赋值对象压入栈中
stack.setValue("name","erica");//为栈顶对象的指定属性名赋值
```

上面的代码中并没有发现将Model对象入栈的部分，是由于ActionInvocation在初始化的时候已经预先完成了这个工作，如DefaultActionInvocation.init方法中代码所示：

[4] OGNL=Object-Graph Navigation Language。提供基于表达式的对象属性访问功能。在许多项目中得到使用，如 Webwork、Apache Tapestry 等。Spring Framework 1.2 版本中也将引入。

```
    private void init() throws Exception {
        Map contextMap = createContextMap();

        createAction();

        if (pushAction) {
           stack.push(action);
        }
        ……
    }
```

输入校验

Web 应用开发中，我们常常面临如何保证输入数据合法性的头痛问题。实现输入数据校验的方法无外乎两种：

1. 页面 Java Script 校验
2. 服务器端、执行逻辑代码之前进行数据校验

Struts、Spring MVC 均为服务器端的数据校验提供了良好支持。
而对于客户端的校验，大多 MVC 框架力所不逮。
WebWork 则在提供灵活的服务器端数据校验机制的基础上，进一步提供了对页面 Java Script 数据校验的支持。这也可算是 WebWork 中的一个亮点。

与 Spring MVC 类似，XWork 也提供了一个 Validator 接口，所有数据校验类都必须实现这个接口。
XWork 发行时已经内置了几个常用的数据校验类（位于包 com.opensymphony.xwork.validator.validators），同时也提供了 Validator 接口的几个抽象实现（抽象类），以便于用户在此基础上进行扩展（如 FieldValidatorSupport）。

服务器端数据合法性校验的动作，发生在 Action 被调用之前。回忆之前关于 Xwork 拦截器体系的讨论，我们自然想到，通过 Interceptor 在 Action 运作之前对其输入参数进行校验是一个不错的思路。事实的确如此。WebWork 中提供了一个 ValidationInterceptor，它将调用指定的 Validator 对输入的参数进行合法性校验。
对这一细节感兴趣的读者可参考 Xwork 中的 ActionValidatorManager 和 ValidationInterceptor 的实现代码。

下面我们将首先讨论 Validator 的使用，之后再完成一个针对特定需求的 Validator 实现。

为“用户登录”示例加入数据合法性校验：

1. 首先，为 Action 配置用于数据校验的 Interceptor：

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

      <action name="login"
          class="net.xiaxin.webwork.action.LoginAction">

          <result name="success" type="dispatcher">
              <param name="location">/main.jsp</param>
          </result>

          <result name="loginfail" type="dispatcher">
              <param name="location">/index.jsp</param>
```

```
        </result>

        <interceptor-ref name="params" />
        <interceptor-ref name="model-driven" />
        <interceptor-ref name="validationWorkflowStack" />

    </action>
  </package>
</xwork>
```

这里我们并没有直接引用 ValidationIncepteror。而是引用了 validationWorkflowStack，webwork-default.xml 中对其定义如下：

```
<interceptor-stack name="validationWorkflowStack">
    <interceptor-ref name="defaultStack"/>
    <interceptor-ref name="validation"/>
    <interceptor-ref name="workflow"/>
</interceptor-stack>
```

这表明 validationWorkflowStack 实际上是三个 Interceptor 的顺序组合。

为什么要引用这样一个 Interceptor 组合，而不是直接引用 validation Inceptor？
这与我们的实际需求相关，在 Web 应用中，当输入数据非法时，一般的处理方式是返回到输入界面，并提示数据非法。而这个 Interceptor 组合则通过三个 Interceptor 的协同实现了同样的逻辑。由此也可见 Webwork 中拦截器体系设计的精妙所在。

2. 在 Class Path 的根目录（如 WEB-INF/classes）创建一个 validators.xml 文件，此文件中包含当前应用中所有需要使用的 Validator。

   Webwork 发行包中提供了一个 validators.xml 示例（\bin\validators.xml）可供参考，一般而言，我们只需根据实际需要在此文件上进行修改即可。

   对于我们的登录示例而言，需要校验的有两个字段，用户名和密码，校验逻辑分别为：

   1. 用户名不可为空
   2. 密码不可为空，且长度必须为 4 到 6 位之间

   这里，我们通过 Xwork 提供的 RequiredStringValidator 和 StringLengthFieldValidator 来实现这一校验逻辑。

```
<validators>

    <validator name="required"
    class="com.opensymphony.xwork.validator.validators.Requ
iredStringValidator" />

    <validator name="length"
```

```
    class="com.opensymphony.xwork.validator.validators.Stri
ngLengthFieldValidator" />
</validators>
```

配置非常简单，只需指定此 Validator 的实现类和及其名称。下面的数据校验配置文件中，将通过此 Validator 名称，对实际的 Validator 实现类进行引用。

3. 针对页面表单字段配置对应的 Validator。
在我们的登录页面中，共有两个字段 model.username 和 model.password。为其建立一个配置文件：LoginAction-validation.xml：

```
<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork
Validator 1.0.2//EN"
"http://www.opensymphony.com/xwork/xwork-validator-1.0.2.d
td">

<validators>

    <field name="model.username">
        <field-validator type="required">
            <message>Please enter Username!</message>
        </field-validator>
    </field>

    <field name="model.password">
        <field-validator type="length">
            <param name="minLength">4</param>
            <param name="maxLength">6</param>
            <message>
              Password length must between ${minLength} and
${maxLength} chars!
             </message>
        </field-validator>
    </field>
</validators>
```

配置文件有两种命名约定方式：

1. Action 类名-validation.xml
   如上面的 LoginAction-validation.xml
2. Action 类名-Action 别名-validation.xml
   如 LoginAction-login-validation.xml
   其中 Action 别名就是 xwork.xml 中我们申明 Action 时为其设定的名称。

配置文件必须放置在与对应 Action 实现类相同的目录。

配置文件格式非常简单，结合 validators.xml，我们可以很直观的看出字段的校验关系。
值得注意的是，通过 param 节点，我们可以为对应的 Validator 设置属性值。这一点与 Spring IOC 非常类似，是的，实际上 XWork 也提供了内置的 IOC 实现，不过与 Spring 的 IOC 支持想比还有一些差距，这里就不再深入探讨。有兴趣的读者可参见一下文档：

另外 message 定义中，我们可以通过“${}”操作符对属性进行引用。

4. 修改 LoginAction，使继承 ActionSupport 类。

```
public class LoginAction extends  ActionSupport implements
Action, ModelDriven
{
    ……
}
```

ActionSupport 类实现了数据校验错误信息、Action 运行错误信息的保存传递功能。通过扩展 ActionSupport，LoginAction 即可携带执行过程中的状态信息，这为之后的错误处理，以及面向用户的信息反馈提供了基础数据。

5. 修改页面，增加数据合法性校验错误提示：

```
<%@ page pageEncoding="gb2312"
contentType="text/html;charset=gb2312"%>
<%@ taglib prefix="ww" uri="webwork"%>
<html>
    <style type="text/css">
    .errorMessage {
        color: red;
    }
    </style>
<body>
<form action="/login.action">
    <ww:if test="hasFieldErrors()">
        <span class="errorMessage">
            <b>Errors:</b><br>
            <ww:iterator value="fieldErrors">
              <li><ww:property value="value[0]"/></li>
            </ww:iterator>
        </span>
    </ww:if>
    …以下略…
</form>
```

```
</body>
</html>
```

首先，我们通过<ww:if test="hasFieldErrors()">判断是否存在字段验证错误，hasFieldErrors()是 ActionSupport 中提供的方法，用于判定当前是否存在字段验证错误。LoginAction 扩展了 ActionSupport 类，自然继承了这个方法，我们即可在页面中通过 EL 对其进行调用。

如果存在 FieldErrors，则通过一个迭代，在页面上显示错误信息：

```
<ww:iterator value="fieldErrors">
    <li><ww:property value="value[0]"/></li>
</ww:iterator>
```

fieldErrors 是 ActionSupport 中保存字段校验错误信息的 Map 结构。针对这个 Map 进行迭代，得到的每个迭代项都是一个 key-value Entry，key 中保存着字段名，value 则是一个 List 数据结构，里面包含了针对这个 Key 的错误信息列表，错误信息的数量取决于字段验证规则配置中的设定。这里 value="value[0]"，也就是取当前条目 value 中保存的第一条错误信息。

我们在表单中，输入一个符合校验条件的 Password 之后提交，显示结果如下：

为了显示一条错误信息，页面中需要增加这么多的代码，似乎繁琐了些。
为了提供简便、统一的页面构成组件。WebWork 提供了一套 UI Tag。UI Tag 对传统的 HTML TAG 进行了封装，并为其增加了更多辅助功能。如服务端校验错误信息的显示，甚至还可以自动为表单元素生成用于客户端校验的 JavaScript 脚本。

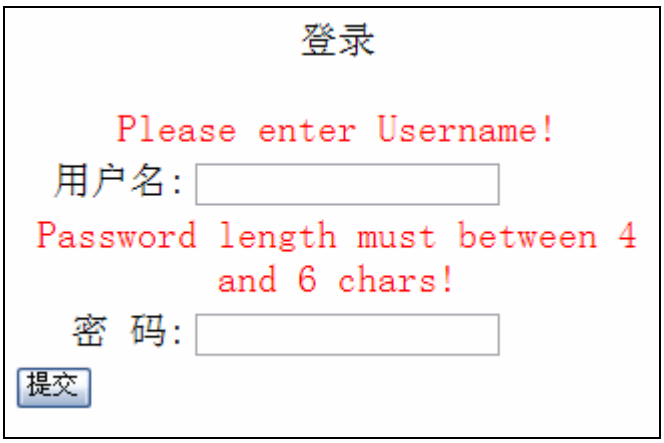
Webwork 提供的 UI_Tag 非常丰富，这里就不逐个介绍。Opensymphony 的 Wiki 站点中提供了丰富的信息，包括功能描述和实例，是开发实践过程中的必备参考资料：
http://wiki.opensymphony.com/display/WW/UI+Tags

下面来看上例的 UI Tag 实现版本：

```
<%@ page pageEncoding="gb2312"
contentType="text/html;charset=gb2312"%>
<%@ taglib prefix="ww" uri="webwork"%>

<html>
  <style type="text/css">
  .errorMessage {
      color: red;
  }
  </style>
<body>

<p align="center">登录</p>
<ww:form name="'login'" action="'login'" method="'post'">

  <ww:textfield label="'用户名'" name="'model.username'" />
  <ww:password label="'密  码'" name="'model.password'" />
  <ww:submit value="'提交'"/>

</ww:form>

</body>
</html>
```

（其中的表单属性值均为常量，因此均以单引号包围，以与 Model 中的变量相区分）

运行结果如下：



我们看到，使用 ui-tag 的页面代码简洁了许多，不过，上面的页面布局似乎有点凌乱。

原因一方面在于代码中并没有特意考虑页面布局，毕竟这只是一个用作演示的界面。

另一方面，也就是使用 ui-tag 所需付出的代价——界面灵活性上的牺牲。当然这并不意味着使用了 ui-tag 我们就无法对 tag 内部的 html 结构和风格做出修改，而是作出修改的难度有所增加。

使用 Webwork UI-Tag 构建的 jsp 页面，在编译的时候会从\template\xhtml 目录（默认）中读取 Tag 模板以生成最终的页面 html。

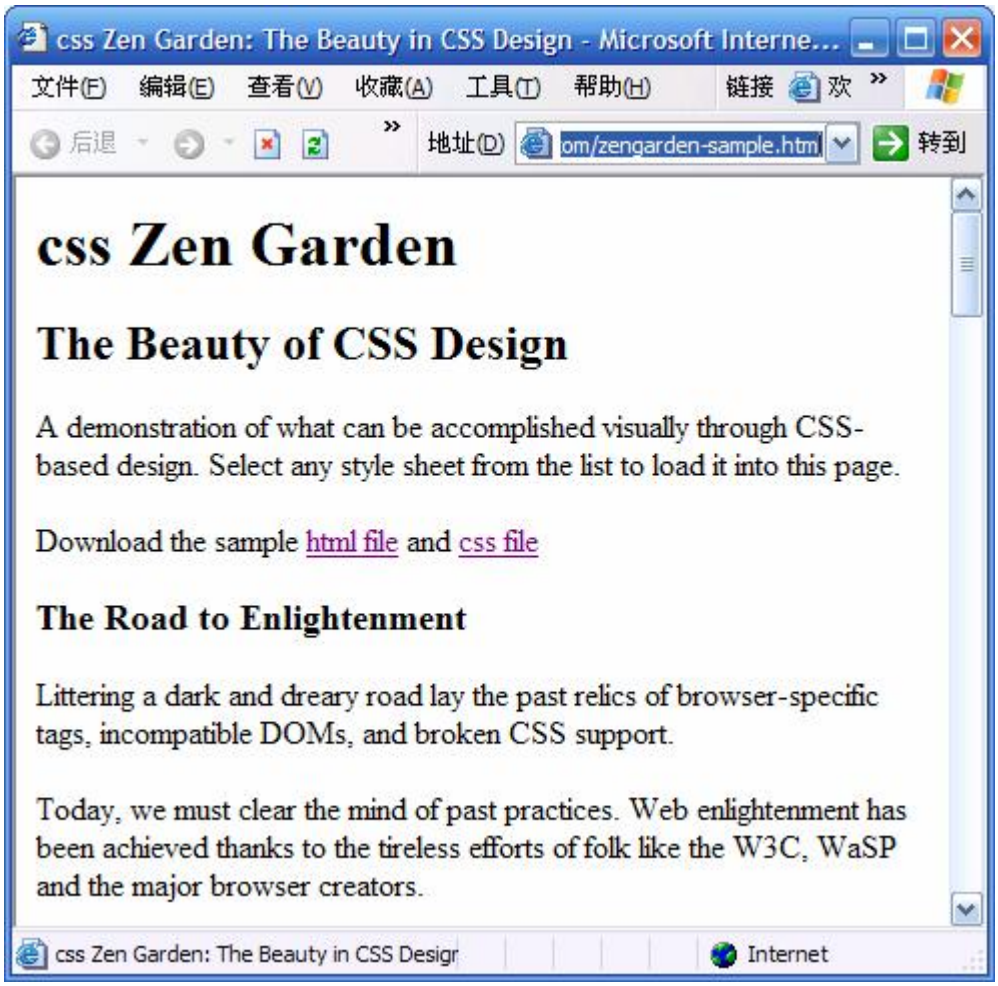
Webwork.jar 中包含了一些预制的模板（\template）。但这些预制模板面对复杂、个性化的页面设计需求可能有点力不从心。这就要求我们具备编写 Webwork 模板文件的能力。

前面曾经提及，这些模板是基于 Velocity 编写，velocity 语法虽然并不复杂，但如果界面设计多样、修改这些琐碎的模板也不是一件轻松的事情。

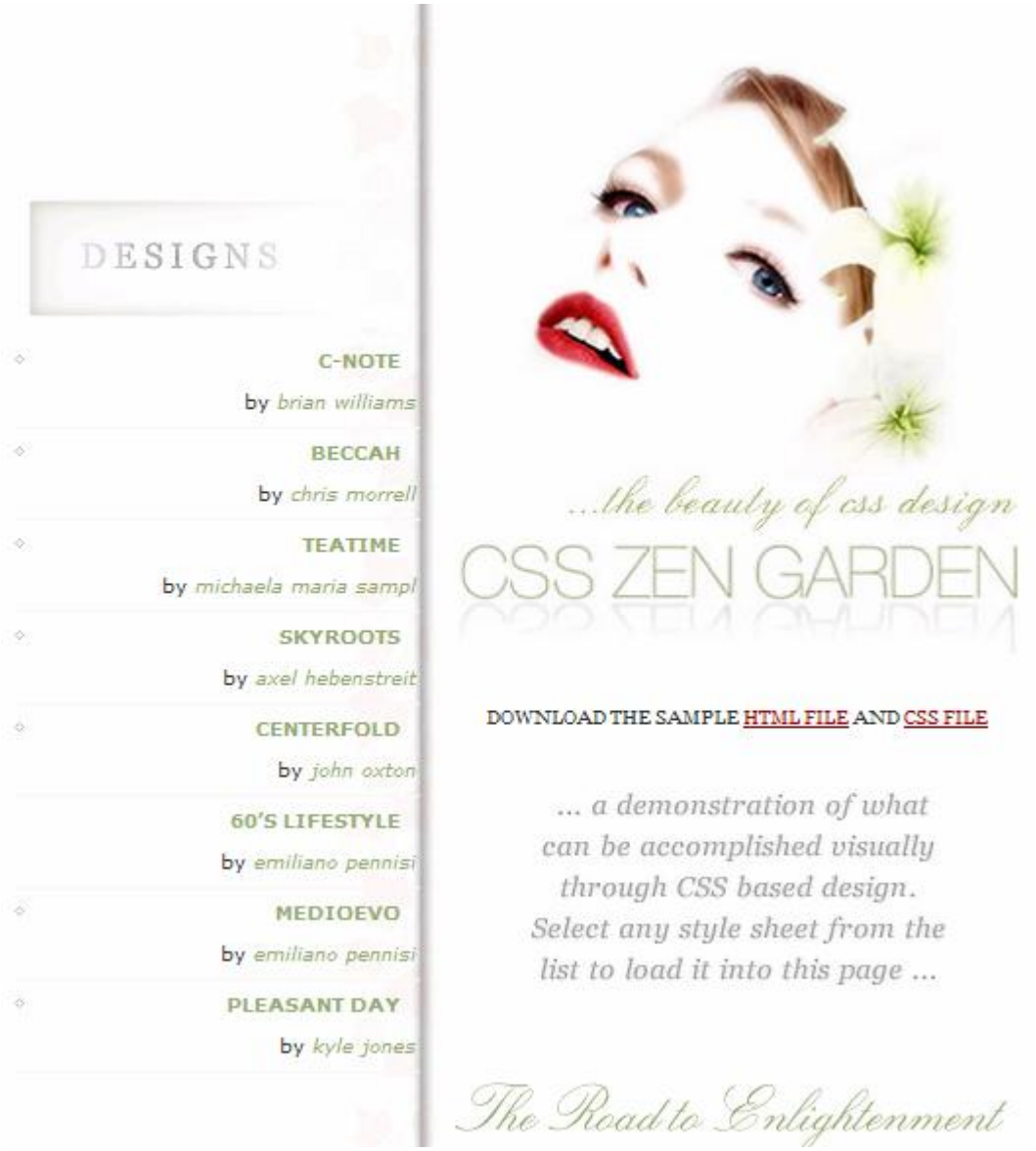
除了通过修改模板改变页面布局，我们还可以通过另外一种手段，实现对页面风格的改造，即 CSS。

通常情况下，CSS 对于大多数 Web 开发者而言，只是用作统一指定页面字体和表格边框属性的一个样式设定工具。不过，CSS 的功能远超于此。我们不但可以指定字体和表格边框这些页面元素属性，也可以通过 CSS 对页面的结构进行调整，同样的一个页面，应用不同的 CSS，可能是完全不同的页面表现效果。

CSS 禅意花园（http://www.csszengarden.com）展示了样式表的神奇之处，效果可能出乎大多数人意料之外：
下面是一个未应用 CSS 的纯 Html 页面样本：



下面是匹配不同 CSS 之后的效果：


DESIGNS
C-NOTE
by brian williams
BECCAH
by chris morrell
TEATIME
by michaela maria sampl
SKYROOTS
by axel hebenstreit
CENTERFOLD
by john oxton
60'S LIFESTYLE
by emiliano pennisi
MEDIOEVO
by emiliano pennisi
PLEASANT DAY
by kyle jones
...the beauty of css design
CSS ZEN GARDEN
DOWNLOAD THE SAMPLE HTML FILE AND CSS FILE
... a demonstration of what
can be accomplished visually
through CSS based design.
Select any style sheet from the
list to load it into this page ...
The Road to Enlightenment

想必大家已经感受到冲击性的效果，CSS 原来也可以被发挥得如此淋漓尽致。同样一个 JSP 页面，应用不同的 CSS 设计，完全两样。

不过，这里并没有暗示您应该尽快投入到 CSS 的怀抱，精通 CSS 并能将其发挥到淋漓尽致的人并不多。我们必须根据团队中成员的技术水平和专业分工，在各种选择之间的优劣与代价之前作出权衡。

对于持续的产品开发而言，UI-Tag 结合 CSS 无疑在页面代码简洁性和模板重用上达到了较高水平。一旦开发出一套符合公司产品风格的模板，随后的项目即可对这些模板进行重用。在长期来看，可以带来生产率的提高。另一方面， Web 系统可以轻易的实现换肤功能。

然而对于短期、界面设计多变而人力投入有限的小型项目开发而言。使用 UI-Tag 可能就有点让人疲惫。为了调整一个表单的结构往往要来回修改调试多次，远不如使用传统 jsp 简单高效。建议初次使用 Webwork 进行项目开发的用户不要轻易使用 UI-Tag，除非您已经积累了足够的经验。

关于模板和 Theme，请参见：
http://wiki.opensymphony.com/display/WW/Themes
看过这个主题之后，相信可以对 UI-Tag 的特点和使用成本有所把握。

使用 UI-Tag 的注意点：

请确定/WEB-INF/web.xml 文件中的 ServletDispatcher 设定为自动加载，如下：

```
<servlet>
  <servlet-name>webwork</servlet-name>
     <servlet-class>
        com.opensymphony.webwork.dispatcher.ServletDispatcher
     </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

UI tag内部是基于Velocity实现（无论我们选用的表现层技术是JSP还是Velocity，UI tag内部都是通过velocity生成最后的界面元素）。

ServletDispatcher在初始化过程中将完成包括创建Velocity管理器在内的一系列工作。

如果ServletDispatcher没有设置为自动加载模式，则只有当第一个Action 请求被提交时，ServletDispatcher才由Web容器加载。

这也就意味着，当你在应用启动完毕，首次访问含有UI tag的页面（*.jsp）时，由于ServletDispatcher的初始化工作尚未进行（尚未有.Action请求被调用），UI tag所依赖的Velocity管理器尚未创建，页面也就无法正常构建，我们将不得不面对充斥着异常信息的错误提示页面。

最后，来看看客户端数据校验。

Webwork的客户端数据校验功能是一个亮点。它会根据用户的配置，根据预定义的JavaScript模板，自动在页面生成验证脚本。不过使用这一功能的前提条件就是，必须使用UI-Tag构建表单。

下面我们尝试对上面示例进行修改，为其增加客户端校验功能。

非常简单，修改validators.xml和LoginAction-validation.xml，使其包含JavaScriptRequiredStringValidator的定义：

validators.xml

```
<validators>
  <validator name="required"
  class="com.opensymphony.webwork.validators.JavaScriptRequiredStri
ngValidator" />
</validators>
```

LoginAction-validation.xml:

```
<validators>

  <field name="model.username">
      <field-validator type="required">
          <message>Please enter Username!</message>
      </field-validator>
  </field>

</validators>
```

修改jsp页面，为其ww:form加上validate="true"选项：

```
<%@ page pageEncoding="gb2312"
contentType="text/html;charset=gb2312"%>
<%@ taglib prefix="ww" uri="webwork"%>

<html>
<body>

<p align="center">登录</p>
<ww:form name="'login'" action="'login'" method="'post'"
    validate="true">

  <ww:textfield label="'用户名'" name="'model.username'"/>
  <ww:password label="'密  码'" name="'model.password'" />
  <ww:submit value="'提交'" align="center"/>

</ww:form>
```

```
</body>
</html>
```

此时，在浏览器中访问此页面，不填用户名提交，则将得到一个JavaScript警告窗口：

查看页面源代码可以看到，Webwork在生成的页面内，自动加入了针对表单域的JavaScript校验代码。

这一切实现是如何实现的？看看Webwork客户端校验器代码也许有助于加深理解。

下面是Webwork内置的客户端/服务器数据校验类JavaScriptRequiredStringValidator:

```
public class JavaScriptRequiredStringValidator extends
RequiredStringValidator implements ScriptValidationAware {

    public String validationScript(Map parameters) {
        String field = (String) parameters.get("name");
        StringBuffer js = new StringBuffer();

        js.append("value = form.elements['" + field + "'].value;\n");
        js.append("if (value == \"\") {\n");
        js.append("\talert('" + getMessage(null) + "');\n");
        js.append("\treturn '" + field + "';\n");
        js.append("}\n");
        js.append("\n");

        return js.toString();
    }
}
```

JavaScriptRequiredStringValidator扩展了服务器端校验类RequiredStringValidator并实现了ScriptValidationAware接口，从而同时具备了客户端和服务器两端校验的功能，这样即使浏览器的JavaScript功能被禁用，也可以通过服务器端校验防止非法数据的提交。

从代码中我们可以看出，ScriptValidationAware接口定义的validationScript方法提供了JavaScript校验脚本，Webwork进行页面合成时，即可通过调用此方法获得校验脚本并将其插入到页面中。

使用客户端校验的注意点：

1. 必须为ww:form标签指定form name，Webwork生成JavaScript代码时，将通过此Form name作为表单元素的索引。

2. 对于namespace中的action而言，ww:form中必须指定namespace属性。
如对于以下namespace中的login action:

```
<package name="default" namespace="/user"
                extends="webwork-default">
     <action name="login" …>
     ……
</package>
```

对应的ww:form标签申明为：

```
<ww:form name="'login'" namespace="'/user'" action="'login'"
     method="'post'" validate="true">
     ……
</ww:form>
```

3. ww:form中的action无需追加.action后缀，webwork会自动在生成的页面中为其追加.action后缀。

国际化支持

Webwork 的国际化支持主要体现在两个部分：

1. UI-Tag
2. Validator

UI-Tag 中的<ww:i18n/>和<ww:text/>标记为页面显示提供了国际化支持，使用非常简单，下面的例子中，假定登录页面中，针对不同语种（中英文）需要显示欢迎信息“欢迎“或 Welcome”。

```
<ww:i18n name="'messages'">
  <ww:text name="'welcome'"/>
</ww:i18n>
```

上面的脚本，在运行期将调用 JDK ResourceBundle 读取名为 messages 的国际化资源，并从中取出 welcome 对应的键值。

ResourceBundle 会自动在 CLASSPATH 根路径中按照如下顺序搜寻配置文件并进行加载（以当前 Locale 为 zh_CN 为例）：

messages_zh_CN.properties
messages_zh.properties
messages.properties
messages_zh_CN.class
messages_zh.class
messages.class

本示例对应的两个配置文件如下：

messages_zh_CN.properties：

```
welcome=欢迎
```

注意中文资源文件必须通过JDK中内置的native2ascii工具进行转码（转为Unicode）方可正常显示。

转码后的messages_zh_CN.properties：

```
welcome=\u6B22\u8FCE
```

messages_en_US.properties：

```
welcome=Welcome
```

之后，Webwork 会自动根据当前的 Locale 设置，读取对应的资源文件填充页面。在中文系统上，这里即显示“欢迎”，如果是英文系统，则显示“Welcome”。

如果需要在文本中附带参数，则可通过 ww:param 指定参数值，如：

```
<ww:i18n name="'messages'">
  <ww:text name="'welcome'">
    <ww:param>192.168.0.2</ww:param>
```

```
    <ww:param>Erica</ww:param>
  </ww:text>
</ww:i18n>
```

未转码前的 messages_zh_CN.properties：

```
welcome=欢迎来自{0}的用户{1}
```

运行时，{0}和{1}将被 ww:param 指定的参数值替换，显示结果如下：

欢迎来自 192.168.0.2 的用户 Erica

此外，在 UI Tag 中，我们还可以通过 getText 方法读取国际化资源，如对于刚才的登录界面，如果要求对于输入框的标题栏也实现国际化支持，则可对表单中的内容进行如下改造：

```
<ww:i18n name="'messages'">
  <ww:text name="'welcome'">
    <ww:param>192.168.0.2</ww:param>
    <ww:param>Erica</ww:param>
  </ww:text>

  <ww:textfield label="getText('username')"
                name="'model.username'"/>
  <ww:password  label="getText('password')"
                name="'model.password'" />
  <ww:submit value="getText('submit')" align="center"/>

</ww:i18n>
```

同时在资源文件添加 username，password 和 submit 对应的键值配置即可。

除了页面上的国际化支持，我们注意到，另外一个可能影响到用户界面语种问题的内容就是 Validator，我们在 Validator 中定义了校验失败时的提示信息，如：

```
<field name="model.username">
    <field-validator type="required">
        <message>Please enter Username!</message>
    </field-validator>
</field>
```

这里的 message 节点，定义了校验失败时的提示信息，对于这样的提示信息我们必须也为其提供国际化支持。

Validator 中国际化支持通过 message 节点的 key 属性完成：

```
<field name="model.username">
    <field-validator type="required">
```

```
            <message key="need_username">
                    Please enter Username!
            </message>
        </field-validator>
    </field>
```

这里的 key 指定了资源文件中的键名，显示提示信息时，Webwork 将首先在资源文件中查找对应的键值，如果找不到对应的键值，才以上面配置的节点值作为提示信息。

Validator 对应的资源文件，与 Action 校验配置（这里就是 LoginAction_validation.xml）位于同一路径，命名方式为“<ActionName><_LOCALE>.properties”，这里对应的就是“LoginAction_zh_CN.properties”。

对应上例，只需在资源文件中配置 need_username 键值，Webwork 在运行期就会自动从此资源文件中读取数据，以之作为校验失败提示信息。

Webwork2 in Spring

Spring MVC 在设计时，针对 Webwork 的不足提出了自己的解决方案。不过，设计者一旦脱离实际开发，开始陷入框架本身设计的完美化时，往往容易陷入过度设计的陷阱。

请原谅这里笔者对 Rod Johnson 及其开发团队的一点善意批评。Rod Johnson 开始脱离实际开发，上升到框架设计时，如同大多数框架开发者一样，完美化的思想充斥了设计过程，就 Rod Johnson 针对 Struts 和 Webwork 的评论来看，其注意力过多的集中在设计上的完善，而忽略了实际开发中另外一个重要因素：易用性。

就 Spring MVC 本身而言，设计上的亮点固然无可非议，但在设计灵活性和易用性两者之间，Rod Johnson 的天平倒向了完美的设计。这导致 Spring MVC 使用起来感觉有点生涩，使用者赞叹其功能强大的同时，面对复杂的开发配置过程，难免也有点抓耳挠腮。

随着角色的转变（一个应用开发者到一个框架设计者），思考角度必然产生一些微妙的变化，希望 Rod Johnson 在 Spring MVC 后继版本的开发中，能更多站在使用者的角度出发，针对 Spring MVC 的易用性进行进一步改良。

就笔者在实际项目开发中的感觉来看，Webwork2 虽然也并不是及易上手，但在一定程度上，较好的兼顾了易用性。其代价是设计上不如 Spring MVC 完美，但我们开发中节省的脑细胞，应该可以弥补这一缺陷。

就 MVC 框架设计的角度来看，Webwork2 在目前的主流实现中（Struts、Webwork、Spring MVC）恰恰处于中间位置。其设计比 Struts 更加清晰，比 Spring 更加简练。

而从使用者角度而言，其接受难度也处于中等位置，比 Struts 稍高（Webwork 中也引入了 DI 等新的设计思想导致学习过程中需要更广泛的技术知识），比 Spring MVC 难度稍低。

而正是这一平衡点的把握，为 Webwork2 提供了与其他竞争对手一较高下的资本。

那么，Webwork 和 Spring Framework 的关系是怎样？

笔者曾经参与过一些讨论，很多开发者的观点是“如果选择了 Webwork，就不必再引入 Spring，两种框架同时使用必将导致认识上的冲突和技术感觉的混杂。”

然而，这里，正如之前所说，Spring 是一个高度组件化的框架。如果选择了 Webwork 作为开发基础框架，那么 Spring MVC 自然不必同时使用，但 Spring Framework 中的其他组件还是对项目开发颇有裨益。

站在 Web 应用层开发的角度而言，Spring 中最重要的组件，除了 MVC，还有另外一个令人欣赏的部分：持久层组件。持久层封装机制也许是 Spring 中应用级开发最有价值的部分，就笔者的视野来看，目前无论商业还是开源社区，尚无出其右者。

这里想要表达的意思就是：Webwork+Spring（Core+Persistence+Transaction Management）也许是目前最好的 Web 开发组合。对 Web 应用最重要的技术组成部分（MVC、持久层封装、事务管理）在这个组合中形成强大的合力，可以在很大程度上提高软件产品的质量和产出

效率。

当然，局限于笔者的知识范围，并不能保证这句话就一定正确，不同的出发点，固然有不同的看法。崇尚简单至上的方案（JSP+JavaBean+JDBC），以及皇家正统的企业级策略（JSP+SLSB+CMP），在不同的出发点上，也都是不错的选择。上面是笔者的看法，供大家参考。

下面我们就 Webwork+Spring（WS）组合的应用方式进行探讨。

首先来看，WS 组合中，Webwork 和 Spring 各司何职？

对一个典型的 Web 应用而言，MVC 贯穿了整个开发过程。选用 Webwork 的同时，也就确定了 MVC 框架的实现。

MVC 提供了纵向层面的操控。也就是说，Webwork 将纵向贯穿 Web 应用的设计，它担负了页面请求的接收、请求数据的规格统一、逻辑分发以及处理结果的返回这些纵向流程。

Spring 则在其中提供横向的支持。Model 层的情况比较典型，Webwork 将请求分发到 Action 之后，所能做的事情就是等待 Action 执行完毕然后返回最后的结果。至于 Action 内部如何处理，Webwork 并不关心。而这里，正是 Spring 大展身手的地方。

场景有点类似在 Pizzahut 用餐，伺服人员（Webwork）负责接受客户定餐(请求)，并将用户口头的定餐要求转化为统一的内部数据格式（Pizzahut 订单），然后将订单递交给厨师制作相应的餐点（执行 Action），之后再从厨房将餐点送到客户餐位。而厨师具体如何操作，伺服并不参与。

定单传递到厨师手上之后，厨师即按照烹饪流程（业务逻辑）开始制作餐点，烹饪的过程中，厨具必不可少，厨具可以选用乡间的柴灶、锅、碗、瓢、盆五件套，也可以选择自动化的配套厨具。

相信大家也已经明白我的意思，Spring 就是这里的自动化配套厨具，没有它固然也可以使用传统工具作出一份点心，但是借助这样的工具，你可以做的更好更快。

下面我们通过一个实例来演示 WS 组合的应用技术。

还是上面的用户登录示例。我们使用 Webwork 作为 MVC 框架，而在逻辑层（Action 层面），结合 Spring 提供的事务管理以及 Hibernate 封装，实现一个完整的登录逻辑。

界面部分并没有什么改变，主要的变化发生后台的执行过程。

第一个问题，Webwork 如何与 Spring 相融合？

假设 LoginAction 中调用 UserDAO.isValidUser 方法进行用户名、密码验证。最直接的想法，可能就是在 LoginAction 中通过代码获取 ApplicationContext 实例，然后通过 ApplicationContext 实例获取对应的 UserDAO Bean，再调用此实例的 isValidUser 方法。类似：

```
……
ApplicationContext ctx=new
      FileSystemXmlApplicationContext("bean.xml");
UserDAO userDAO = (UserDAO) ctx.getBean("userDAO");
if (userDAO.isValidUser(username,password)){
    ……
};
```

没问题，这样的确可以达到我们的目的。不过，这样的感觉似乎不是很好，太多的代码，更多的耦合（Action 与 Spring 相耦合）。

事实上，这样的实现方式并不能称之为“融合”，只是通过 API 调用的方式硬生生将两者捆绑。

我们期望能做到以下的效果：

```
……
if (userDAO.isValidUser(username,password)){
    ……
};
……
```

userDAO 自动由容器提供，而无需代码干涉。

这并不难实现，只是需要追加一个类库，以及一些配置上的修改。[5]

首先下载 http://www.ryandaigle.com/pebble/images/webwork2-spring.jar，并将其放入 WEB-INF\lib。

webwork2-spring.jar 中包含了 Spring 与 Webwork 融合所需的类库。

修改 web.xml，为 Web 应用增加相应的 Spring ContextLoaderListener 以及 webwork2-spring.jar 中的 ResolverSetupServletContextListener 配置。如下：

```
……
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
<listener>
    <listener-class>
com.atlassian.xwork.ext.ResolverSetupServletContextListener
    </listener-class>
```

[5] 参见 Webwork Wiki Document：WebWork 2 Spring Integration

```
    </listener>
    ……
```

修改 xwork.xml，为 LoginAction 配置外部引用关系：

```
<xwork>
    <include file="webwork-default.xml" />

    <package name="default" extends="webwork-default"
    externalReferenceResolver="com.atlassian.xwork.ext.SpringServl
etContextReferenceResolver">
        <interceptors>
            <interceptor name="reference-resolver"
    class="com.opensymphony.xwork.interceptor.ExternalReferencesIn
terceptor" />

            <interceptor-stack name="WSStack">
                <interceptor-ref name="params" />
                <interceptor-ref name="model-driven" />
                <interceptor-ref name="reference-resolver" />
            </interceptor-stack>
        </interceptors>

        <action name="login" class="net.xiaxin.action.LoginAction">
            <external-ref name="userDAO">
                    userDAOProxy
               </external-ref>

            <result name="success" type="dispatcher">
                <param name="location">/main.jsp</param>
            </result>

            <result name="loginfail" type="dispatcher">
                <param name="location">/index2.jsp</param>
            </result>

            <result name="input" type="dispatcher">
                <param name="location">/index2.jsp</param>
            </result>

            <interceptor-ref name="WSStack" />
        </action>
    </package>
</xwork>
```

可见，interceptors 中增加了一个用于获得外部引用的拦截器 ExternalReferencesInterceptor。我们将其与 params、model-driven 组合为一个拦截器序列用于简化之后 Action 中的拦截器配置。

“login”中增加了一个外部引用“userDAO”，其值为 userDAOProxy，这是 Spring 中 userDAO Transaction Proxy 实例的引用名。

通过以上配置，我们实现了 Webwork、Spring 之间的融合。Webwork 将在运行期从 Spring Context 中获取资源引用，而 Spring 则将对此资源进行管理，并提供相关的调度服务（事务管理等）。

下面，我们在 LoginAction.java 中加入对应的 userDAO 申明：

```
public class LoginAction implements Action, ModelDriven {

    LoginInfo loginInfo = new LoginInfo();

    private UserDAO userDAO;

    public String execute() throws Exception {

        if (userDAO
            .isValidUser(loginInfo.getUsername(),
                         loginInfo.getPassword())
             ) {
            return SUCCESS;
        } else {
            return ERROR;
        }
    }

    public UserDAO getUserDAO() {
        return userDAO;
    }

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

    public Object getModel() {
        return loginInfo;
    }
}
```

最后，在配置文件 applicationContext.xml 对 userDAO 进行配置，下面的操作与正常情

况下的 Spring 配置方法完全一样。

applicationContext.xml:

```
<beans>

    <bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">

        <property name="driverClassName">
            <value>org.gjt.mm.mysql.Driver</value>
        </property>

        <property name="url">
            <value>jdbc:mysql://localhost/sample</value>
        </property>

        <property name="username">
            <value>sysadmin</value>
        </property>

        <property name="password">
            <value>security</value>
        </property>

    </bean>

    <bean id="sessionFactory"
    class="org.springframework.orm.hibernate.LocalSessionFactoryBe
an">
        <property name="dataSource">
            <ref local="dataSource" />
        </property>
        <property name="mappingResources">
            <list>
                <value>net\xiaxin\db\entity\User.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">
                    net.sf.hibernate.dialect.MySQLDialect
                </prop>
                <prop key="hibernate.show_sql">
```

```
                    true
                </prop>
            </props>
        </property>
    </bean>

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

    <bean id="userDAO" class="net.xiaxin.db.dao.UserDAOImp">
        <property name="sessionFactory">
            <ref local="sessionFactory" />
        </property>
    </bean>

    <bean id="userDAOProxy"
    class="org.springframework.transaction.interceptor.Transaction
ProxyFactoryBean">

        <property name="transactionManager">
            <ref bean="transactionManager" />
        </property>

        <property name="target">
            <ref local="userDAO" />
        </property>

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

            </props>
        </property>
    </bean>

</beans>
```

UserDAOImp.java:

```
public class UserDAOImp extends HibernateDaoSupport implements
UserDAO {
    private SessionFactory sessionFactory;

    private static String hql = "from User u where u.username=? ";

    public boolean isValidUser(String username, String password) {

        List userList = this.getHibernateTemplate().find(hql,
username);//出于演示的简洁性考虑，这里并没有校验密码

        if (userList.size() > 0) {
            return true;
        }
        return false;
    }
}
```

UserDAO.java:

```
public interface UserDAO {
    public abstract boolean isValidUser(
                            String username,
                            String password
                            );
}
```

至于 Hibernate 的 OR 映射操作这里就不浪费篇幅进行描述。如果有兴趣，可以自行创建一个简单的 User 表（只需一个 username 字段即可）进行测试（可参见笔者的另一篇文档《Hibernate 开发指南》）。

可以看到，在 Webwork 中引入 Spring 组件，同时整合了两大框架的优势技术，这为我们的 Web 应用创造了极佳的框架基础。

WebWork 配置说明

和 WebWork2 应用相关的所有配置文件:

文件	可选	位置	目的
Web.xml	必须	/WEB-INF/	Web 应用的描述文件, 包含所有必须的 WebWork 组件.
xwork.xml	必须	/WEB-INF/classes/	WebWork 最主要的配置文件, 其中包含结果/视图类型, action 映射,拦截器等等.
webwork.properties	不是	/WEB-INF/classes/	Webwork 属性
webwork-default.xml	不是	/WEB-INF/lib/ webwork-x.x.jar	WebWork2 提供的默认配置文件, 可以包含到 xwork.xml 文件中.
velocity.properties	不是	/WEB-INF/classes/	可以用来覆盖默认的 velocity 配置.
validators.xml	不是	/WEB-INF/classes/	定义输入信息的验证器.
components.xml	不是	/WEB-INF/classes/	定义 IOC 组件
taglib.tld	必须	/WEB-INF/lib/ webwork-x.x.jar	WebWork 标记库的描述文件.

web.xml 中的配置信息

1. 配置 ServletDispatcher

ServletDispatcher 将处理 **Web** 应用中映射到 **Webwork2 Action** 的所有请求**.**

```
<!—This entry is required to have the framework process calls to WebWork actions
-->
<servlet>
  <servlet-name>webworkDispatcher</servlet-name>
<servlet-class>com.opensymphony.webwork.dispatcher.ServletDispatcher</servle
t-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>webworkDispatcher</servlet-name>
  <url-pattern>*.action</url-pattern>
</servlet-mapping>
```

注意**:** 以上的配置假定 **actions** 将使用**".action"**作为扩展名**.** 如果你想使用其他扩展名**,** 可以通过修改 **url-pattern** 元素来指定你自己的扩展名。

2. 配置 CoolUriServletDispatcher (可选)

这个是一个自定义 servlet 分配器, 它把 servlet 路径映射到 Actions。这个配置器可以用来替代上面的 ServletDispatcher。 它的格式如下:

```
http://HOST/ACTION_NAME/PARAM_NAME1/PARAM_VALUE1/PARAM_NAME2/PARAM_VALUE2
```

开发人员可以设置所有想要的参数. URL 也可以缩短成以下格式:

```
http://HOST/ACTION_NAME/PARAM_VALUE1/PARAM_NAME2/PARAM_VALUE2 或者
```

```
http://HOST/ACTION_NAME/ACTION_NAME/PARAM_VALUE1/PARAM_NAME2/PARAM_VALUE2
```

下面是根据 id 显示文章的 URL 的例子:

```
http://HOST/article/ID
```

我们需要做的就是把/article/*映射到这个 servlet 并且在 WebWork 中定义一个名叫 article 的 action.

```
<servlet>
  <servlet-name>coolDispatcher</servlet-name>
       <servlet-class>com.opensymphony.webwork.dispatcher.CoolUriServletDispa
       tcher</servlet-class>
  </servlet>
<servlet-mapping>
  <servlet-name>coolDispatcher</servlet-name>
  <url-pattern>/article/*</url-pattern>
</servlet-mapping>
```

3. 配置对 Velocity 的支持 (可选)

WebWork 可以使用 Velocity 作为基础模板系统来自定义 JSP 标记库。当然，这样也不需要配置 WebWorkVelocityServlet。只有当需要直接调用 velocity 模板或者作为一种处理结果的类型，才需要配置 WebWorkVelocityServlet.

```
<servlet>
  <servlet-name>velocity</servlet-name>
      <servlet-class>com.opensymphony.webwork.views.velocity.WebWorkVelocityS
      ervlet</servlet-class>
  <load-on-startup>10</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>velocity</servlet-name>
  <url-pattern>*.vm</url-pattern>
</servlet-mapping>
```

4. 配置标记库(可选)

标记库是 WebWork 框架中一个可选的部分，它提供了对 JSP 中 forms 和 actions 的支持.

```
<taglib>
  <taglib-uri>webwork</taglib-uri>
  <taglib-location>/WEB-INF/lib/webwork-2.1.jar</taglib-location>
</taglib>
```

如果应用服务器不支持包含在 jar 中的标记库配置，你可以按照一下方式来配置：

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

5. 配置对 Freemarker 支持(可选)

通过以下配置提供对 Freemarker 的支持:

```
<servlet>
```

```
  <servlet-name>freemarker</servlet-name>
      <servlet-class>com.opensymphony.webwork.views.freemarker.FreemarkerServ
      let</servlet-class>
  <load-on-startup>10</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>freemarker</servlet-name>
  <url-pattern>*.ftl</url-pattern>
</servlet-mapping>
```

6. 配置 IoC / LifeCycle (可选)

参见 IoC/LifeCycle 的配置

xwork.xml

xwork.xml 是 WebWork 最主要的配置文件, 其中包含结果/视图类型, action 映射,拦截器等元素。

```
<!DOCTYPE xwork PUBLIC "-//OpenSymphony Group//XWork 1.0//EN"
      "http://www.opensymphony.com/xwork/xwork-1.0.dtd">
<xwork>
      <include file="webwork-default.xml" />                          (1)
      <package name="default" extends="webwork-default">              (2)
       <action name="login"                                           (3)
           class="net.xiaxin.webwork.action.LoginAction">
           <result name="success" type="dispatcher">                  (4)
              <param name="location">/main.jsp</param>
           </result>
           <result name="loginfail" type="dispatcher">
              <param name="location">/index.jsp</param>
           </result>
           <interceptor-ref name="params" />                          (5)
           <interceptor-ref name="model-driven"/>                     (6)
       </action>
      </package>
</xwork>
```

1. include

通过 include 节点，我们可以将其他配置文件导入到默认配置文件 xwork.xml 中。从而实现良好的配置划分。在这里例子中我们导入了 Webwork 提供的默认配置 webwork-default.xml（位于 webwork.jar 的根路径）。

所有通过 include 引入的配置文件的格式和 xwork.xml 是一样的。

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

2. package

XWork 中，可以通过 package 对 action, results, result types, interceptors 和 interceptor stacks 进行分组。这类似 Java 中 package 和 class 的关系，为可能出现的同名 Action 提供了命名空间上的隔离。

以下是 package 元素的属性:

属性	是否必须	描述
name	是的	pachage 的唯一的标识,可以被其他 pachage
extends	不是	指明其继承至那个 package
namespace	不是	Namespace 属性允许开发人员把 action 分割到不同的命名空间中，这样就可以在不同的 package 下配置同样的 action 别名.
abstract	不是	指明这个 package 是抽象的，指定为抽象以后，就可以不用在 package 中定义 action

在这里可以看出， xwork 中的 package 有两个个高级特性:

a) 支持继承关系。

在上面的例子中，我们可以看到：`extends="webwork-default"` 。

"webwork-default"是 webwork-default.xml 文件中定义的 package，这里通过继承，"default" package 自动拥有"webwork-default" package 中的所有定义关系。

这个特性为我们的配置带来了极大便利。在实际开发过程中，我们可以根据自身的应用特点，定义相应的 package 模板，并在各个项目中加以重用，无需再在重复繁琐的配置过程中消耗精力和时间。

b) 在 Package 节点中指定 namespace

这样就将 action 分为若干个逻辑区间。如：

```
<package name="default" namespace="/user"
        extends="webwork-default">
```

就将此 package 中的 action 定义划归为/user 区间，之后在页面调用 action 的时候，我们需要用/user/login.action 作为 form action 的属性值。其中的/user/就指定了此 action 的 namespace，通过这样的机制，我们可以将系统内的 action 进行逻辑分类，从而使得各模块之间的划分更加清晰

3. action

```
<action
      name="formTest"
      class="com.opensymphony.webwork.example.FormAction"
      method="processForm"
>
```

Actions 是 WebWork 中定义的最小工作单元。在 WebWork 中一个 Action 通常是一个来至页面的请求（比如：点击一个按钮， 提交一个表单）。

以下是 action 元素的属性:

属性	是否必须	描述
Name	是的	action 的唯一的标识,在同一个命名空间下是唯一的

Class	是的	这个 Action 所对应的类
method	不是	"method"用于指定 Action 中的那一个被 WebWork 调用。 如果没有指定特定的值，WebWork 将调用默认的 execute()方法。

4. result

```
<result name="missing-data" type="dispatcher">
    <param name="location">/form.jsp</param>
    <param name="parameterA">A</param>
    <param name="parameterB">B</param>
</result>
```

通过 result 节点，可以定义 Action 返回语义，即根据返回值，决定处理模式以及响应界面。在 webwork-default.xml 中 WebWork 已经定义了一些常用的处理模式，随后的 param 节点则设定这些处理模式所需要的特定参数。

这些常用模式如下：

a) dispatcher

本系统页面间转向，类似 jsp 中的 forward。

param 参数	是否必须	描述
location	是的	请求转发到的地址
parse	不是	默认为 true. 如果设置为 false，location 参数将不会被解析为 Ognl 表达式

b) redirect

浏览器重定向，可转向其他系统页面。

param 参数	是否必须	描述
location	是的	重定向的地址
parse	不是	默认为 true. 如果设置为 false，location 参数将不会被解析为 Ognl 表达式

c) chain

将处理结果转交给另外一个 Action 处理，以实现 Action 的链式处理， 在这个过程中前一个 Action 的 **ActionContext** 将传递给下一个 **Action**。

param 参数	是否必须	描述
actionName	是的	请求转发到的 Action
namespace	不是	设置这个 Action 的所在的命名空间。

d) velocity

将指定的 velocity 模板作为结果呈现界面。

param 参数	是否必须	描述
location	是的	velocity 模板的地址
parse	不是	默认为 true. 如果设置为 false，location 参数将不会被解析为 Ognl 表达式

e) FreeMarker

将指定的 FreeMarker 模板作为结果呈现界面。

param 参数	是否必须	描述
location	是的	FreeMarker 模板的地址
parse	不是	默认为 true. 如果设置为 false，location 参数将不会被解析为 Ognl 表达式
contentType	不是	默认为 "text/html"

f) JasperReports

使用指定格式（默认为 PDF）生成一个 JasperReports 报告

param 参数	是否必须	描述
location	是的	报告模板的地址
parse	不是	默认为 true. 如果设置为 false，location 参数将不会被解析为 Ognl 表达式
dataSource	是的	通过使用 Ognl 表达式从值栈中提取报告所需要的数据
format	不是	指定报告格式（默认为 PDF）

g) xslt

将指定的 XSLT 作为结果呈现界面。

param 参数	是否必须	描述
location	是的	xslt 文件的地址
parse	不是	默认为 true. 如果设置为 false，location 参数将不会被解析为 Ognl 表达式

5. Interceptors

WebWork 的拦截器允许开发人员定义在 Action 调用前或者后执行的代码（具体使用方式请参见对 XWork 拦截器体系的说明）。

```
<interceptor name="timer"
      class="com.opensymphony.xwork.interceptor.TimerInterceptor"/>
```

WebWork 在 `webwork-default.xml` 提供了一些默认的拦截器程序：

名称	描述
timer	输出 Action 执行的时间(包括其后续包含拦截器和视图)
logger	输出这个 Action 的名字
chain	在 chain 结果处理模式下，把上一个 Action 的属性继承到当前 Action。
static-params	把 xwork.xml 的 Action 元素中的参数(< param >元素)设置到 Action 中。
params	把 Web 应用程序中，把 request 中的参数设置到 Action
model-driven	当 Action 继承 ModelDriven， 拦截器把 getModel()得到的结果放到值栈中。
component	Enables and makes the components available to the Actions. Refer to components.xml
token	检查 Action 的令牌环
token-session	作用和 token 一样，只是令牌环出错的把提交的数据放到 session 中。
validation	根据{Action}-validation.xml 中定义的字段验证配置来检查传入的参数。
workflow	调用 Action 中的验证方法。 如果 action 报错， 那么返回输入视图。
servlet-config	允许开发人员访问 HttpServletRequest 和 HttpServletResponse (在使用该拦截器前请再三考虑， 是否需要把你的程序和 Servlet 的 api 捆绑起来)
prepare	如果 Action 实现了 Preparable 接口，拦截器将调用这个 Action 的 prepare()方法。
conversionError	adds conversion errors from the ActionContext to the Action's field errors

fileUpload	添加文件上载支持。
execAndWait	该拦截器使 action 在后台执行而在客户端暂时显示一个等待的页面。