高效 Java Web 应用开发框架

—— JessMA v3.5.1-20150225

Bruce Liang

前 言

JessMA Java MVC & REST 应用开发框架（简称 JessMA）是一套功能完备的高性能 Full-Stack Web 应用开发框架，内置稳定高效的 MVC 基础架构和 DAO 框架（已内置 Hibernate、MyBatis 和 JDBC 支持），集成 Action 拦截、Form Bean / Dao Bean / Spring Bean 装配、国际化、文件上传下载和缓存等基础 Web 应用组件，提供高度灵活的纯 Jsp/Servlet API 编程模型，完美整合 Spring / Guice，支持 Action Convention “零配置”，能快速开发传统风格和 RESTful 风格的 Web 应用程序，文档和代码清晰完善，非常容易学习。

JessMA 在设计之初就充分注重功能、性能与使用体验。JessMA 主要特点：

1) **功能全面：**内置稳定高效的 MVC 基础架构和 DAO 框架，支持 Action 拦截、Form Bean / Dao Bean / Spring Bean 装配和声明式事务，提供国际化、文件上传下载、缓存和页面静态化等常用 Web 组件，能满足绝大部分 Web 应用的需要。

2) **高度扩展：**JessMA 通过的 plug-in 机制可以灵活扩展，JessMA 发布包中自带的 jessma-ext-rest 和 jessma-ext-spring 均以插件的形式提供，用户可根据需要加载或卸载这些插件。应用程序开发者也可以根据实际需要编写自定义插件来扩展 JessMA。

3) **强大的整合能力：**JessMA 是一个 Full-Stack 框架，同时也是一个开放式框架，可以以非常简单的方式整合第三方组件。本开发手册会详细阐述如何在 JessMA 中整合 FreeMarker、Velocity、UrlRewrite、EHCache-Web、Spring、Hibernate 和 MyBatis 等常用框架和组件。

4) **高性能：**性能要求是 JessMA 的硬性指标，从每个模块的设计到每行代码的实现都力求简洁高效。另外，JessMA 并没有对 JSP/Servet API 进行过多封装，开发者仍然使用 JSP/Servet API 开发应用程序，没有过多的迂回，性能得到保证。

5) **优秀的使用体验：**JessMA 的设计目标之一是提供良好的开发体验，尽量减少应用程序开发者的工作，API 的设计力求简单、完整、明确。同时，JessMA 为应用开发提供了大量 Util 工具，用来处理应用程序开发过程中通常会遇到的一般性问题，进一步减少应用程序开发者的工作负担。

6) **平缓的学习曲线：**学习使用 JessMA 只需掌握一定的 Core Java 与 JSP/Servlet 知识，本开发手册会循序渐进阐述每个知识点，每个知识点都会结合完整的示例进行讲述，知识点之间前后呼应，确保学习者在学习时温故知新，融会贯通。

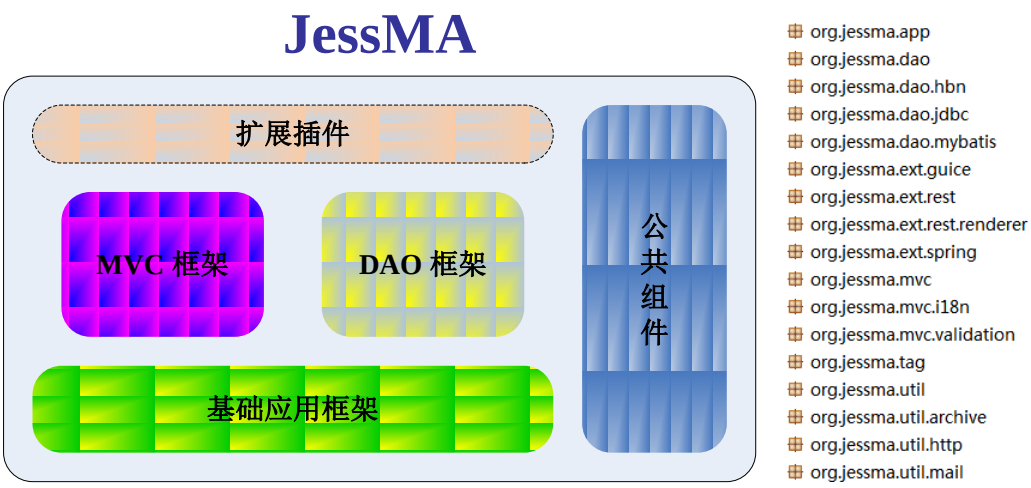
7) **完善的技术支持：**除了提供完善的开发手册和示例代码外，还提供博客、Q~Q~群和官方网站用于解答使用 JessMA 过程中碰到的所有问题。

目 录

1 概 述

1.1 总体架构

JessMA 主要包括以下 5 个部分：



- **基础应用框架**

基础应用框架加载应用程序配置文件（默认：app-config.xml），监听应用程序的生命周期事件，并向上层应用发送应用程序启动和关闭通知，应用程序可以处理这些通知进行额外的初始化或清理工作。基础应用框架在 org.jessma.app 包中是实现。

- **MVC 框架**

MVC 框架加载 MVC 配置文件（默认：mvc-config.xml），通过前端控制器 ActionDispatcher 接收和解析所有的客户 HTTP 请求，然后交由相应的 Action 进行处理，最后生成相应的视图返回给客户端。MVC 框架在 org.jessma.mvc 包中实现。

- **DAO 框架**

DAO 框架封装了所有的数据库访问操作，内置 JDBC、Hibernate 和 MyBatis 数据库访问组件以及 Druid、Proxool、JNDI 等连接池。DAO 框架是可扩展的，用户可以通过扩展 org.jessma.dao.AbstractFacade 和 org.jessma.dao.AbstractSessionMgr 实现自己的数据库访问组件。DAO 框架在以下包中实现：

  - ✓ org.jessma.dao
  - ✓ org.jessma.dao.hbn
  - ✓ org.jessma.dao.jdbc
  - ✓ org.jessma.dao.mybatis

- **公共组件**

公共组件提供多种通用功能帮助类（如：字符串处理、类型转换、分页算法、压缩/解压、加解密、邮件发送等），这些类与框架无关，可在任何应用程序中使用。公共组件在以下包中实现：

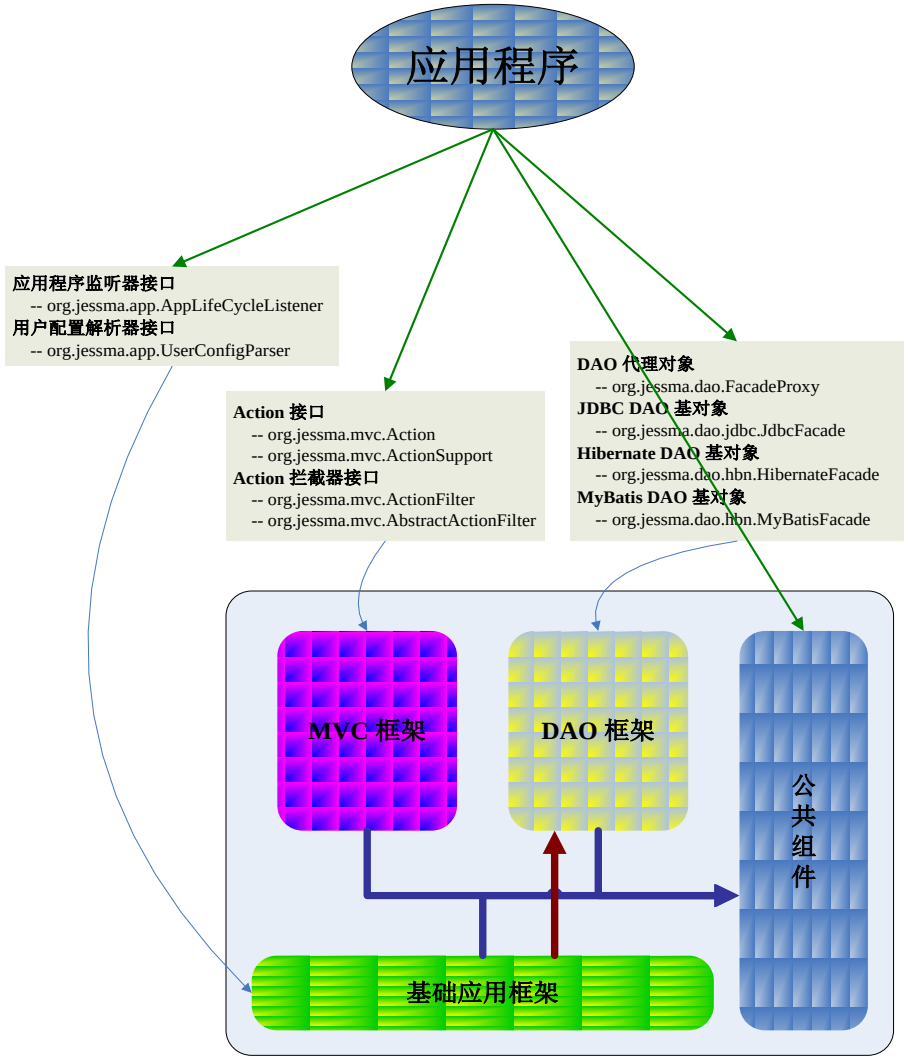
- ✓ org.jessma.util
- ✓ org.jessma.util.archive
- ✓ org.jessma.util.http
- ✓ org.jessma.util.mail

- **扩展插件**

扩展差插件是基于 JessMA 核心框架基础上的功能延伸，不是 JessMA 的必要组件。可由应用程序开发人员根据需要自行定制。JessMA 发行包中也自带了一些扩展插件（如：jessma-ext-spring、jessma-ext-guice 和 jessma-ext-rest）。

1.2 依赖关系

JessMA 各部分的依赖关系如下图所示：



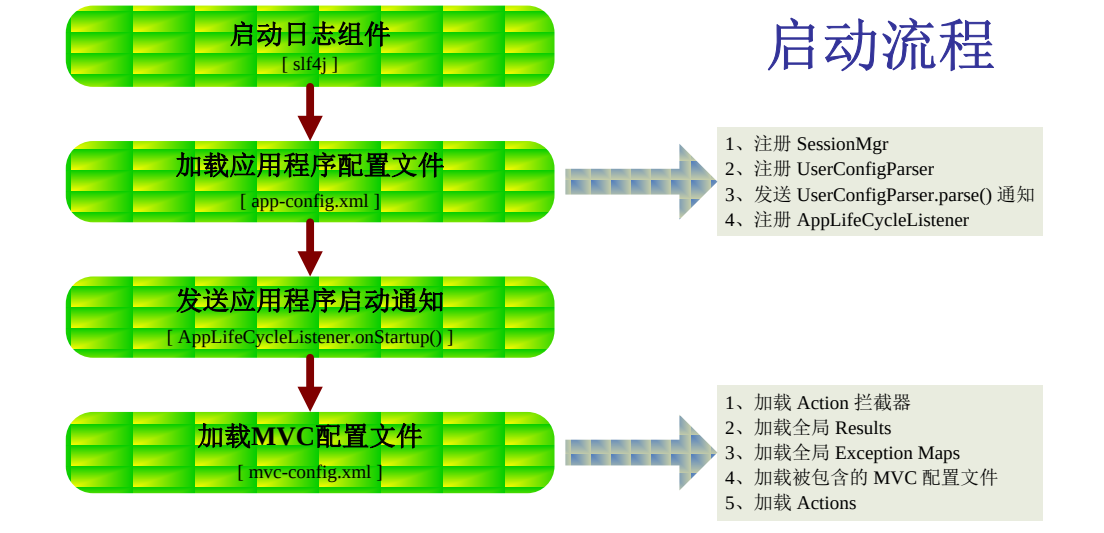
基础应用框架、MVC 框架和 DAO 框架都依赖于公共组件，其中基础应用框架同时依赖于 DAO 框架，因此，MVC 框架和 DAO 框架能脱离 JessMA 单独使用（当然，要附带上公共组件）。例如：可以把 MVC 框架和 DAO 框架用于 JessMA 之外的其它 Web 项目；也可以把 DAO 框架用于非 Web 项目（如：Swing / SWT 桌面应用）。

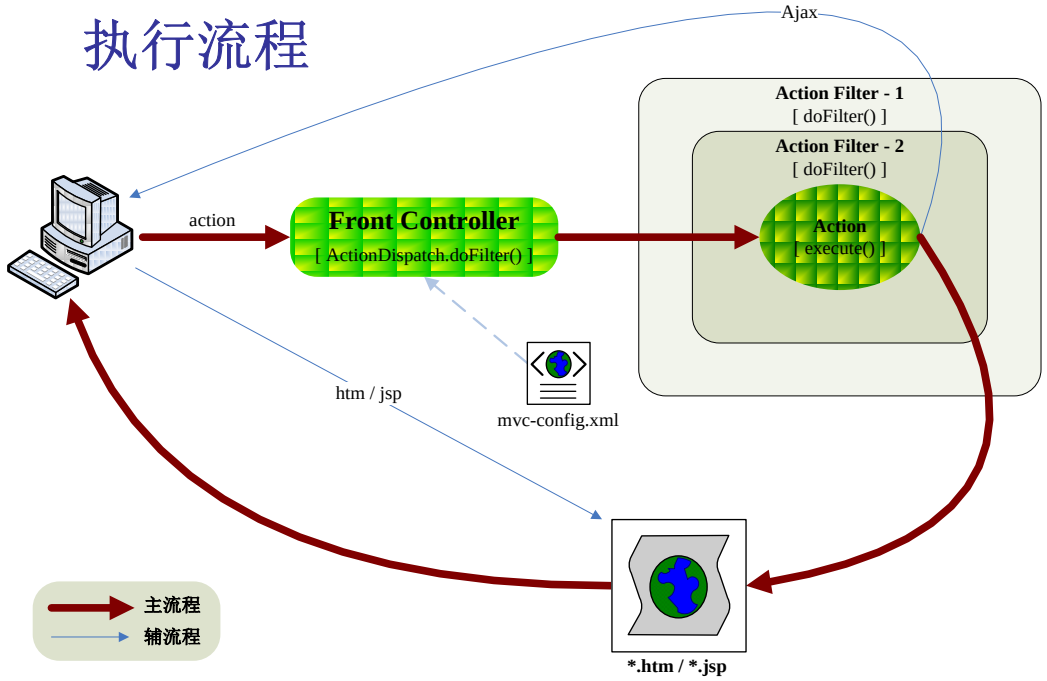
从应用程序的角度来看，应用程序通过以下几种方式与 JessMA 交互：

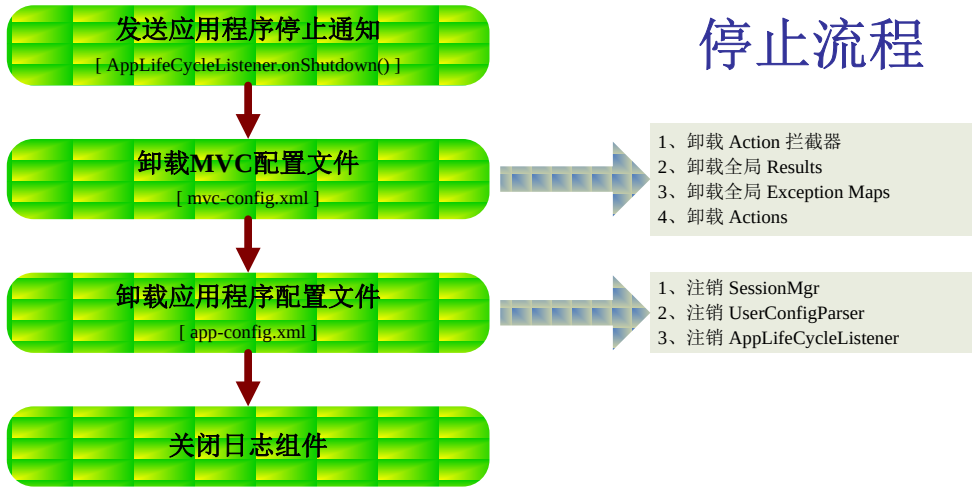
1) 实现 AppLifeCycleListener 接口，接收应用程序的启动和关闭通知（可选）
2) 实现 UserConfigParser 接口，读取应用程序的私有配置信息（可选）
3) 实现 Action 接口或继承 ActionSupport 类，建立 Action 对象
4) 实现 ActionFilter 接口或继承 AbstractActionFilter 类，建立 Action 拦截器对象
5) 继承 JdbcFacade 类，建立基于 JDBC 的 DAO 对象
6) 继承 HibernateFacade 类，建立基于 Hibernate 的 DAO 对象
7) 继承 MyBatisFacade 类，建立基于 MyBatis 的 DAO 对象
8) 通过 FacadeProxy 的 getManualCommitProxy() / getAutoCommitProxy() / create() 方法或 @DaoBean / @DaoBean 注解获取 DAO 的代理对象

1.3 工作流程

JessMA 的生命周期分为三个阶段：启动、执行和停止，下图列出各个阶段的主要工作流程：







- **启动阶段**

启动日志组件，加载应用程序配置文件和 MVC 配置文件，并向应用发送启动通知，让应用有机会执行额外的初始化工作。

- **执行阶段**
  - **主流程：**前端控制器接收所有 action 请求，根据 MVC 配置文件实例化相应的 Action 对象，并调用 Action 对象的 execute() 方法处理请求（在进入 execute() 方法前可能会先经过若干个 Action 拦截器的 doFilter() 方法），然后把处理结果传递到 JSP 生成视图，最后把视图返回给客户端。
  - **辅流程-1：**前端控制器接收所有 Ajax 方式的 action 请求，根据 MVC 配置文件实例化相应的 Action 对象，并调用 Action 对象的 execute() 方法处理请求（在进入 execute() 方法前可能会先经过若干个 Action 拦截器的 doFilter() 方法），

然后直接把处理结果返回给客户端。处理结果通常为格式化的文本，如：XML 或 JSON。

- **辅流程-2**：客户发出非 action 请求（如：htm 或 jsp）则直接转到相应页面。
- * 特殊流程（上图没有展示）：在上述“主流程”中，若 Action 处理完请求后不把处理结果传递到 JSP，也不把处理结果直接返回给客户端，而是通过 Action 链传递给另外一个 Action，对于后面的那个 Action 来说，会重新执行“主流程”或“辅流程-1”。

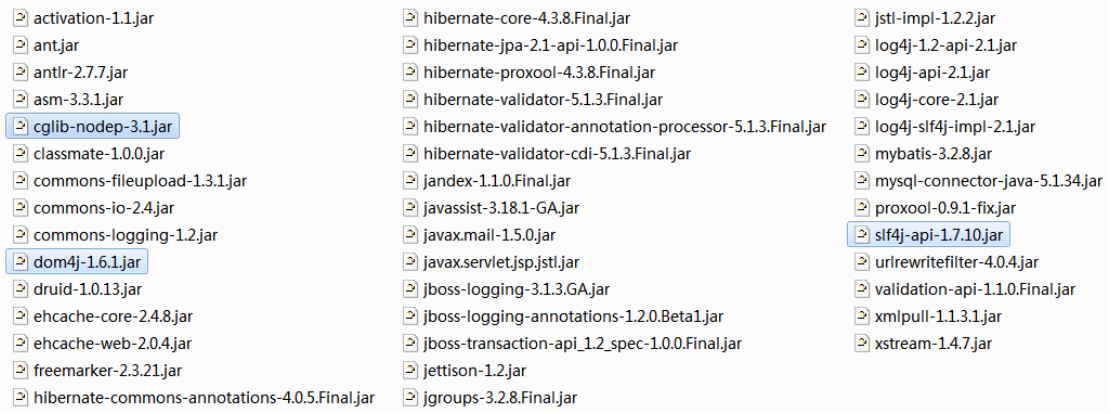
● 停止阶段

首先向应用程序发送关闭通知，让应用有机会执行额外的清理工作，然后卸载 MVC 配置文件和应用程序配置文件，最后关闭日志组件。

1.4 Web 项目搭建

● 依赖包

JessMA 对开发环境没有特殊要求，只需把它的 jar 包及其依赖包加入工程即可，对于一般项目，JessMA 依赖于以下包：



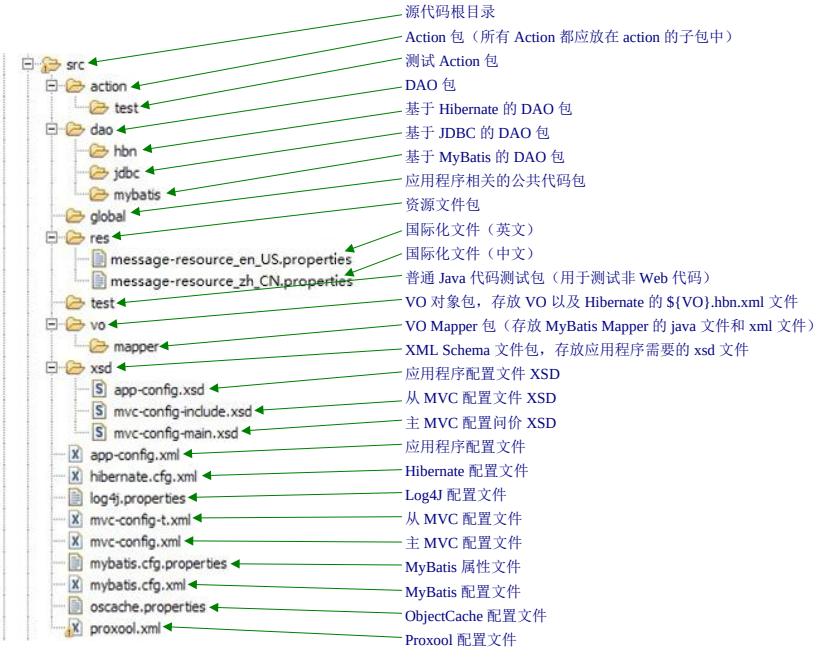
这里的绝大部分包都是可选的。如 Hibernate、MyBatis、MySQL、FreeMarker 的依赖包，如果用不到则可以把它们及其依赖包删除。JessMA 核心依赖的包只有以下几个：

- ✓ JessMA 依赖 3 个基础包：cglib-nodep-x.x.x.jar、dom4j-x.x.x.jar、slf4j-api-x.x.x.jar
- ✓ 文件上传功能则需要 2 个依赖包：commons-fileupload-x.x.x.jar、commons-io-x.x.x.jar
- ✓ 其他依赖包均为可选依赖包，可根据需要加入到项目中

● 工程目录结构

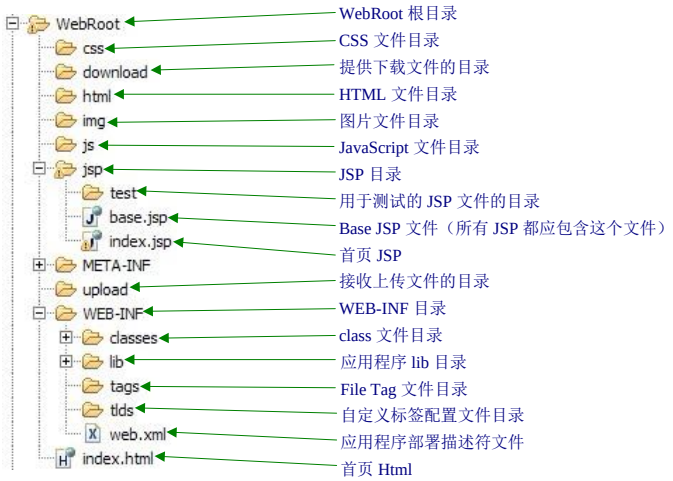
JessMA 对工程目录结构没有特殊要求，但为了更好地组织演示后续章节的所有示例，因此建立如下的 ${SRC} 和 ${WebRoot} 目录结构：

源代码目录结构



本文所有例子的 Action 实现类都放在 action.test 或其子包中。另外，上图中某些文件和目录不是必须的，如果不需要国际化可以删除 res 文件夹及其下面的文件；xsd 文件也不是必须的，但是这些 xsd 文件可以帮助我们在编辑应用程序配置文件和 MVC 配置文件时提供代码提示和错误检测，因此强烈建议保留；通常，一个 Web 程序只采用一种数据库访问方式，因此可根据需要删除无关的数据库配置文件。

WebRoot 目录结构



HTML 文件放在 WebRoot 根目录或 html 目录中，JSP 文件放在 jsp 目录及其子目录中，本文所有例子的 JSP 页面都放在 jsp/test 目录中。

- **日志配置文件**

JessMA 使用 slf4j 输出应用程序日志，下面是一个 slf4j + log4j 2.x 的日志配置文件示例，配置文件为 ${CLASS}/log4j2.xml。其中 JessMA 的默认 Logger 名称固定为“**JessMA**”，因此**所有应用程序都必须配置一个名称为“JessMA”的 Logger**：

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
	<Appenders>
		<Console name="STDOUT" target="SYSTEM_OUT">
			<PatternLayout pattern="%d{ABSOLUTE} %5p [%t]: %m%n"/>
		</Console>
		<Console name="STDERR" target="SYSTEM_ERR">
			<PatternLayout pattern="%d{ABSOLUTE} &lt;%c&gt; %5p [%t]: %m%n"/>
		</Console>
		<RollingRandomAccessFile
			name="ROLLFILE" fileName="${sys:catalina.home}/logs/jessma.log"
	 filePattern="$${sys:catalina.home}/logs/$${date:yyyy-MM}/jessma-%d{yyyy-MM-dd}-%i.log.gz">
			<ThresholdFilter level="INFO" onMatch="ACCEPT" onMismatch="DENY" />
			<PatternLayout pattern="%d{ABSOLUTE} &lt;%c&gt; %5p [%t]: %m%n"/>
			<Policies>
				<TimeBasedTriggeringPolicy />
			</Policies>
			<DefaultRolloverStrategy />
		</RollingRandomAccessFile>
	</Appenders>
	<Loggers>
		<!-- Root Logger -->
		<Root level="INFO">
			<AppenderRef ref="STDOUT"/>
		</Root>
		<!-- JessMA Logger -->
		<Logger name="JessMA" level="DEBUG" additivity="false">
			<AppenderRef ref="STDERR" />
			<AppenderRef ref="ROLLFILE" />
		</Logger>
	</Loggers>
</Configuration>
```

- **jessma-base.jsp**

jessma-base.jsp 主要工作是导入常用标签库和设置“${__base}”，代码如下：

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
<%@ page import="java.util.*, org.jessma.util.*, org.jessma.util.http.HttpHelper, org.jessma.mvc.Action" %>
<%-- 导入 JSTL 常用标签 --%>
```

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<%-- 导入 JessMA 标签 --%>
<%@ taglib prefix="p" uri="http://www.jessma.org/jsp/tags" %>

<%
/* '__base' 设置为 ${WebRoot} 根目录 */
Action.BaseType baseType = (Action.BaseType)application
                                .getAttribute(Action.Constant.APP_ATTR_BASE_TYPE);
if(baseType == Action.BaseType.AUTO)
{
	String __base = (String)request.getAttribute(Action.Constant.REQ_ATTR_BASE_PATH);
	if(__base == null)
	{
		__base = HttpHelper.getRequestBasePath(request);
		request.setAttribute(Action.Constant.REQ_ATTR_BASE_PATH, __base);
	}
}
%>
```

1.5 Maven 项目搭建

JessMA 通过 Maven 构建了 4 个 jar 包：jessma-core、jessma-ext-rest、jessma-ext-guice 和 jessma-ext-spring。Maven 项目可根据实际需要导入相应 jar 包：

（详情可参考发布包中自带的 ***jessma-sample-hello*** 和 ***jessma-sample-set*** 示例工程）

- **jessma-core**

JessMA 核心包，打包了 JessMA 应用程序基础组件、MVC 组件、DAO 组件和工具类：

```
<dependency>
	<groupId>org.jessma</groupId>
	<artifactId>jessma-core</artifactId>
	<version>3.5.1</version>
</dependency>
```

- **jessma-ext-rest**

JessMA RESTful 扩展包，用于开发 RESTful 应用程序：

```
<dependency>
	<groupId>org.jessma</groupId>
	<artifactId>jessma-ext-rest</artifactId>
	<version>3.5.1</version>
```

```
</dependency>
```

- **jessma-ext-guice**

Guice 扩展包，用于在 JessMA 应用程序中整合 Google Guice：

```
<dependency>
	<groupId>org.jessma</groupId>
	<artifactId>jessma-ext-guice</artifactId>
	<version>3.5.1</version>
</dependency>
```

- **jessma-ext-spring**

Spring 扩展包，用于在 JessMA 应用程序中整合 Spring：

```
<dependency>
	<groupId>org.jessma</groupId>
	<artifactId>jessma-ext-spring</artifactId>
	<version>3.5.1</version>
</dependency>
```

1.6 Hello World 示例

我们以 MyJessMA 示例工程的首页作为第一个例子，展示 JessMA 的使用方法，你将会发现 JessMA 的使用非常简单。在这里不深入解释每项配置和代码工作的具体含义，这些内容将在后面各章节陆续阐述。建立测试程序的步骤如下：

1、 用 Eclipse / MyEclipse 新建一个 Web Project，命名为：MyJessMA
2、 根据前面章节的描述导入相关 jar 包，并建立目录结构
3、 加入日志配置文件 log4j2.xml 和首页 HTML 文件 index.html
4、 修改 web.xml，加入 JessMA 的启动器和 MVC 前端过滤器
5、 建立应用程序配置文件 app-config.xml 和 MVC 配置文件 mvc-config.xml
6、 建立 action.test.IndexAction 类处理“index.action”请求
7、 编辑 mvc-config.xml 配置“index action”
8、 编辑 index.jsp 显示首页内容

其中 1-5 步只需在最初建立应用程序时执行一次，6-8 步则需对所有 Action 执行类似的工作。现在从步骤 4 开始介绍这个例子的创建过程。

- **修改 ${WebRoot}/WEB-INF/web.xml，加入 JessMA 启动器和前端过滤器：**

```
<!-- MVC 前端控制器 -->
<filter>
	<filter-name>ActionDispatcher</filter-name>
	<filter-class>org.jessma.mvc.ActionDispatcher</filter-class>
```

```
</filter>
<filter-mapping>
	<filter-name>ActionDispatcher</filter-name>
	<url-pattern>*.action</url-pattern>
	<dispatcher>REQUEST</dispatcher>
	<dispatcher>FORWARD</dispatcher>
	<dispatcher>INCLUDE</dispatcher>
	<dispatcher>ERROR</dispatcher>
</filter-mapping>
<!-- JessMA 启动器 -->
<listener>
	<listener-class>org.jessma.app.AppListener</listener-class>
</listener>
```

◆ **建立 ${SRC}/app-config.xml：**

```
<?xml version="1.0" encoding="UTF-8"?>
<CONFIG xmlns="http://www.jessma.org"
		xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
		xsi:schemaLocation="http://www.jessma.org
		http://www.jessma.org/schema/app-config-3.5.xsd">
	<!-- 系统配置（必填） -->
	<system></system>
	<!-- 用户配置（可选） -->
	<user></user>
</CONFIG>
```

◆ **建立 ${SRC}/mvc-config.xml：**

```
<?xml version="1.0" encoding="UTF-8"?>
<app xmlns="http://www.jessma.org"
		xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
		xsi:schemaLocation="http://www.jessma.org
		http://www.jessma.org/schema/mvc-config-main-3.5.xsd">
	<global>
		<!-- request 和 response 的默认编码（可选，默认：不设置） -->
		<encoding>UTF-8</encoding>
		<!-- Action 请求的后缀（可选，默认：.action） -->
		<action-suffix>.action</action-suffix>
		<!-- 指定 base path （可选，默认：type="auto", href=""） -->
		<base-path type="auto" />
	</global>

	<actions>
```

```
    </actions>
</app>
```

◆ **建立 ${SRC}/action.test.IndexAction.java：**

```
package action.test;

import org.jessma.mvc.ActionSupport;

public class IndexAction extends ActionSupport
{
    @Override
    public String execute() throws Exception
    {
        Integer times   = getSessionAttribute("times");
        int ts          = times == null ? 1 : times + 1;
        setSessionAttribute("times", ts);
        return SUCCESS;
    }
}
```

◆ **修改 ${SRC}/mvc-config.xml，加入 index action：**

```
<?xml version="1.0" encoding="UTF-8"?>
<app xmlns="http://www.jessma.org"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.jessma.org
        http://www.jessma.org/schema/mvc-config-main-3.5.xsd">
    <global>
        <!-- request 和 response 的默认编码（可选，默认：不设置） -->
        <encoding>UTF-8</encoding>
        <!-- Action 请求的后缀（可选，默认：.action） -->
        <action-suffix>.action</action-suffix>
    </global>

    <actions>
        <action name="index" class="action.test.IndexAction">
            <result>/jsp/index.jsp</result>
        </action>
    </actions>
</app>
```

◆ **编辑 ${WebRoot}/jsp/index.jsp**

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@include file="jessma-base.jsp" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <base href="${__base}">
    <title><p:msg key="jsp-index.header"/></title>
     <meta http-equiv="pragma" content="no-cache">
     <meta http-equiv="cache-control" content="no-cache">
     <meta http-equiv="expires" content="0">
  </head>
  <body>
     <br>
    <p:msg key="jsp-index.sayhello" p0="${times}"/>
     <br>
  </body>
</html>
```

index.jsp 首先包含了 jessma-base.jsp，并在 head 中把 <base> 定义为 ${__base}，页面标题和输出内容并未直接指定，而是通过 <p:msg> 国际化标签指定，如果此时访问程序首页 http://localhost:8080/jessma 会引发异常（java.util.MissingResourceException: Can't find resource for bundle java.util.PropertyResourceBundle），原因是没找到 <p:msg> 标签 key 对应的文本，因此，还需做最后一步工作：

◆ **建立 ${SRC}/res/application-message**

在 ${SRC}/res 目录下建立两个文件： application-message_en_US.properties 和 application-message_zh_CN.properties ，并在这两个文件中分别加入“jsp-index.header”和“jsp-index.sayhello”属性（推荐使用 MyEclipse 自带的 Properties Editor 可视化编辑器）。如下图所示：

name	value
jsp-index.header	Index
jsp-index.sayhello	Well Come, You come into index page {0} times !

name	value
jsp-index.header	首页
jsp-index.sayhello	欢迎，您是第 {0} 次进入首页！

现在重新发布应用程序，应该能看到正常的首页了：

欢迎，您是第 14 次进入首页！

如果需要感受下英文页面效果，可以小改一下 IndexAction 的代码：

```
package action.test;

import java.util.Locale;
import org.jessma.mvc.ActionSupport;

public class IndexAction extends ActionSupport
{
    @Override
    public String execute() throws Exception
    {
        // 注意：为避免影响以后的国际化测试示例，测试完成后请删除下面三行代码
        Locale locale = getLocale();
        if(locale != Locale.US)
            setLocale(Locale.US);

        Integer times      = getSessionAttribute("times");
        int ts             = times == null ? 1 : times + 1;
        setSessionAttribute("times", ts);
        return SUCCESS;
    }
}
```

重新运行程序后应该能看到英文页面：

2 应用篇（一）—— 配置文件

注：本章内容相对来说有点枯燥，部分内容可能不能立刻理解透切，但随着后面章节的深入讲解，所有知识点将会融会贯通 ^_

2.1 基础应用框架配置文件

基础应用框架相关的配置文件有两个：应用程序部署描述符文件（web.xml）和应用程序配置文件，其中，web.xml 配置 JessMA 启动器、MVC 前端控制器、应用程序配置文件位置和 MVC 主配置文件位置；应用程序配置文件配置应用程序监听器、用户自定义配置解析器、数据库连接管理器（DAO Session Manager）和用户自定义配置项。

2.1.1 应用程序部署描述符文件（web.xml）

web.xml 包括以下配置选项：

- **JessMA 启动器（必须）**：org.jessma.app.AppListener（实现为： Listener）
- **MVC 前端控制器（必须）**：org.jessma.mvc.ActionDispatcher（实现为：Filter）
- **MVC 主配置文件（可选）**：（默认为：${CLASS}/mvc-config.xml）
- **应用程序配置文件（可选）**：（默认为：${CLASS}/app-config.xml）

参考以下示例：

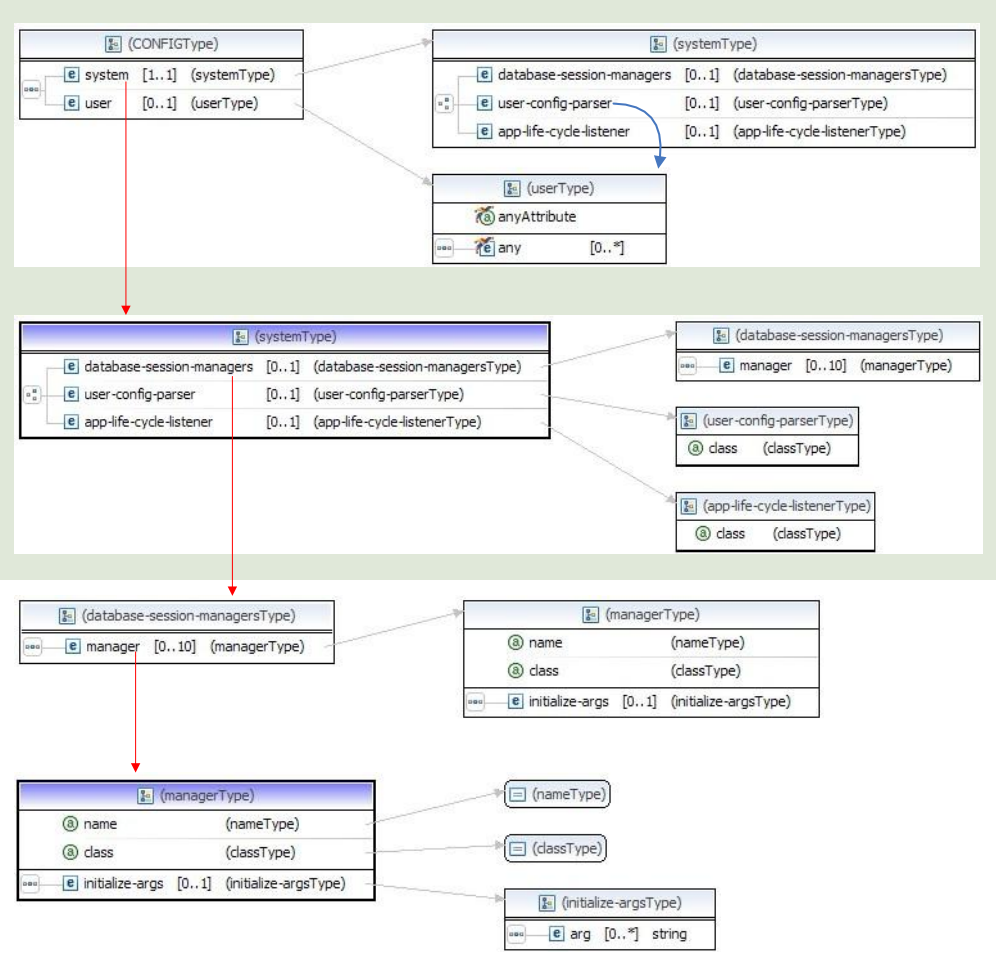
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
	http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

	<!-- 应用程序配置文件 -->
	<context-param>
		<param-name>app-config-file</param-name>
		<param-value>conf/app-config.xml</param-value>
	</context-param>

	<!-- MVC 前端控制器 -->
	<filter>
		<filter-name>ActionDispatcher</filter-name>
		<filter-class>org.jessma.mvc.ActionDispatcher</filter-class>
```

```
            <!-- MVC 主配置文件 -->
            <init-param>
                <param-name>mvc-config-file</param-name>
                <param-value>conf/mvc-config.xml</param-value>
            </init-param>
        </filter>
        <filter-mapping>
            <filter-name>ActionDispatcher</filter-name>
            <url-pattern>*.action</url-pattern>
            <dispatcher>REQUEST</dispatcher>
            <dispatcher>FORWARD</dispatcher>
            <dispatcher>INCLUDE</dispatcher>
            <dispatcher>ERROR</dispatcher>
        </filter-mapping>

        <!-- JessMA 启动器 -->
        <listener>
            <listener-class>org.jessma.app.AppListener</listener-class>
        </listener>
    </web-app>
```

上例中蓝色部分为可选配置项，上例中应用程序配置文件、MVC 配置文件和日志配置文件都放在 ${CLASS}/conf 目录，而不是他们的默认目录 ${CLASS}。

2.1.2 应用程序配置文件

应用程序配置文件包含两部分：系统配置项和用户自定义配置项。其中系统配置项是必须的，用户自定义配置项则为可选。配置结构定义如下图所示：



◆ **系统配置项（system）**

- **数据库连接管理器：database-session-managers（可选）**

配置数据库连接信息，留待以后讲述数据库访问的专题时再详细讲解。

- **用户自定义配置解释器：user-config-parser（可选）**

“用户自定义配置项”解析器实现 org.jessma.app.UserConfigParser 接口。如果使用了“用户自定义配置项”，则通过该类解析这些配置。

- **应用程序监听器：app-life-cycle-listener（可选）：**

应用程序生命周期事件监听器实现 org.jessma.ap.AppLifeCycleListener 接口。它在应用程序的启动和关闭时执行额外的初始化和清理工作。

1. 用户自定义配置项（user）

JessMA 考虑到很多实际的应用程序都需要一些额外的配置信息，因此，为了方便组织和处理这些配置信息，特意在应用程序配置文件中加入“用户自定义配置项”，程序可以在这里配置任何信息，并通过“用户自定义配置解析器”进行解析。

注意：动态更新用户自定义配置请参考：《更新用户自定义配置》章节

2.1.3 日志配置文件

日志配置文件是普通的 slf4j 日志配置文件,对于 log4j 2.x 和 logback 则分别为 ${CLASS}/log4j2.xml 和 ${CLASS}/logback.xml。其中 JessMA Logger 的名称固定为“JessMA”，配置示例参考：《日志配置文件》

注意：

- ***JessMA Logger 对象获取方式：org.jessma.util.LogUtil.getJessMALogger(…)***
- ***其它 Logger 对象获取方式：org.jessma.util.LogUtil.getLogger(…)***

2.1.4 应用示例

本例在 MyJessMA 示例工程中加入应用程序监听器、用户自定义配置解析器和一个用户自定义配置项，并在程序启动和关闭时把相关执行信息打印到日志中。

1、 编辑应用程序配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<CONFIG xmlns="http://www.jessma.org"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.jessma.org
        http://www.jessma.org/schema/app-config-3.5.xsd">

    <!-- 系统配置（必填） -->
    <system>
        <app-life-cycle-listener class="global.MyLifeCycleListener" />
        <user-config-parser class="global.MyConfigParser" />
    </system>

    <!-- 用户配置（可选） -->
    <user>
        <my-home>http://www.cnblogs.com/ldcsaa/</my-home>
    </user>
</CONFIG>
```

2、 编写用户自定义配置解析器

```
package global;

import org.dom4j.Element;
import org.jessma.app.UserConfigParser;
import org.jessma.util.Logger;
```

```
public class MyConfigParser implements UserConfigParser
{
	private static final String MY_HOME = "my-home";

	@Override
	public void parse(Element user)
	{
		Element mh = user.element(MY_HOME);
		if(mh != null)
		{
			String myHome = mh.getTextTrim();
			LogUtil.getJessMALogger().info("My Home is: " + myHome);
		}
	}
}
```

3、 编写应用程序监听器

```
package global;

import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import org.jessma.app.AppLifeCycleListener;
import org.jessma.util.Logger;

public class MyLifeCycleListener implements AppLifeCycleListener
{
	Logger logger = LogUtil.getJessMALogger();

	@Override
	public void onStartup(ServletContext context, ServletContextEvent sce)
	{
		logger.info(this.getClass().getName() + " -> onStartup()");
	}

	@Override
	public void onShutdown(ServletContext context, ServletContextEvent sce)
	{
		logger.info(this.getClass().getName() + " -> onShutdown()");
	}
}
```

启动应用程序时应该能看到以下信息：

```
16:52:52,484 <JessMA>    INFO [main]: ->->->->->->->->->->->->->->->->->->->->->->->->->->->->->->->->->->->->->->->
16:52:52,484 <JessMA>    INFO [main]: starting: org.jessma.app.AppListener@16b352c ...
16:52:52,484 <JessMA>    INFO [main]: load application configuration ...
16:52:52,703 <JessMA>    WARN [main]: 'database-session-managers' element not found
16:52:52,703 <JessMA>    INFO [main]: register USER CONFIG PARSER 'global.MyConfigParser' ...
16:52:52,875 <JessMA>    INFO [main]: parse user configuration ...
16:53:05,890 <JessMA>    INFO [main]: My Home is: http://www.cnblogs.com/ldcsaa/
16:53:08,546 <JessMA>    INFO [main]: register APP LIFE CYCLE LISTENER 'global.MyLifeCycleListener' ...
16:53:08,578 <JessMA>    INFO [main]: global.MyLifeCycleListener -> onStartup()
```

关闭应用程序时应该能看到以下信息：

```
17:00:48,312 <JessMA>    INFO [main]: global.MyLifeCycleListener -> onShutdown()
17:00:48,312 <JessMA>    INFO [main]: unregister JDBC DRIVER 'sun.jdbc.odbc.JdbcOdbcDriver@7b7b5d' ...
17:00:48,312 <JessMA>    INFO [main]: stoping: org.jessma.app.AppListener@16b352c ...
17:00:48,312 <JessMA>    INFO [main]: <-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-<-
```

2.2 MVC 配置文件

MVC 配置文件有两种：MVC 主配置文件和 MVC 从配置文件。其中，主配置文件默认为：${CLASS}/mvc-config.xml，在主配置文件中可配置全局选项和 Actions；从配置文件只用于配置 Actions。从配置文件不是必须的，但实际项目通常包含多个功能模块，并且多人联合开发，因此把 Actions 划分到不同配置文件中非常有好处。由于从配置文件的 Actions 配置与主配置文件的一致，因此不再单独阐述从配置文件的配置方法。

注意：动态更新 MVC 配置请参考：《更新 MVC 配置》章节

MVC 主配置文件包含 3 个配置项：

- **全局配置：global（可选）**

  配置 MVC 框架的全局选项，包括编码格式、Action 后缀名、Action 拦截器、全局 Results 和全局 Exception Mappings。

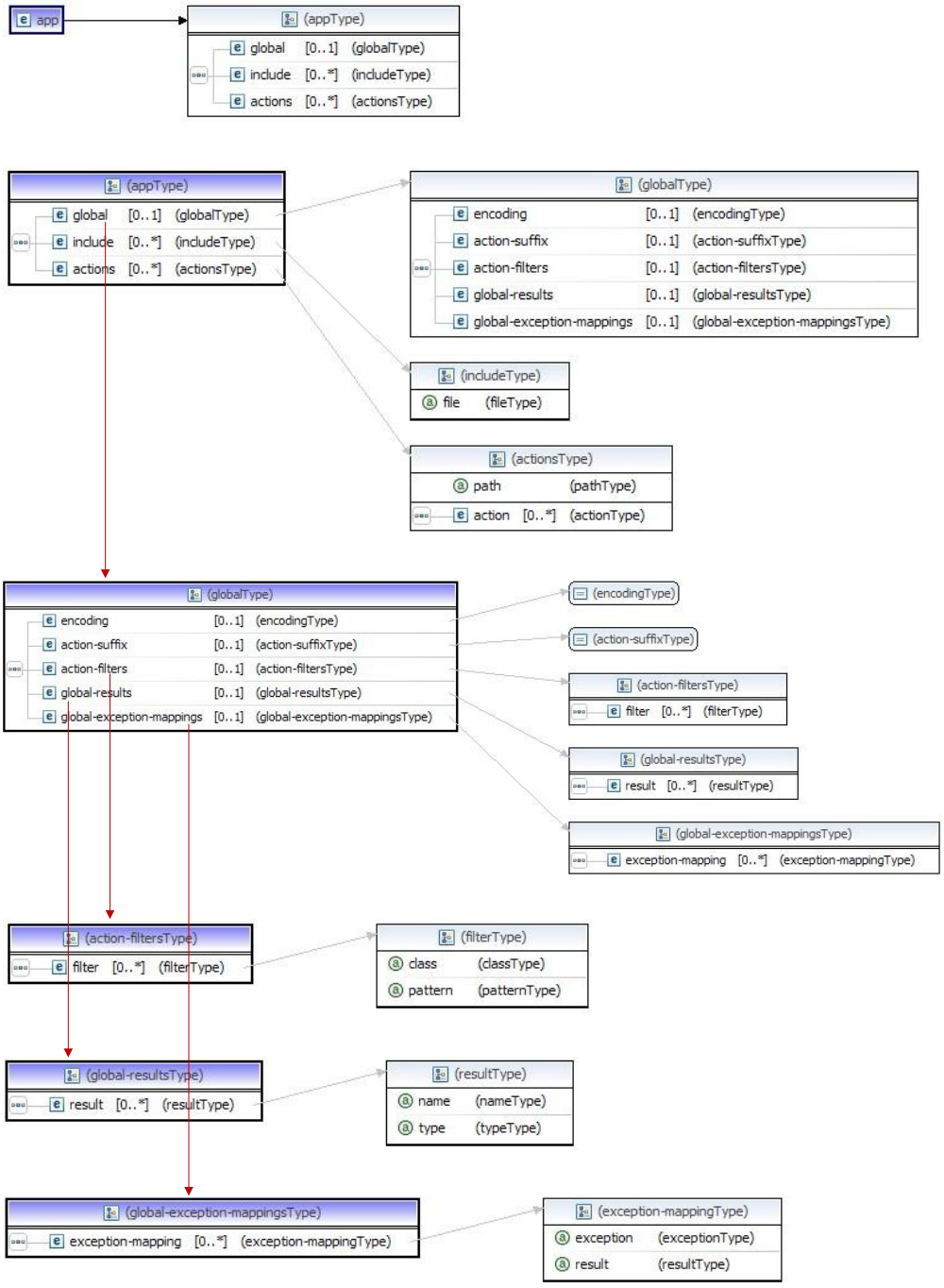
- **包含从配置文件：include（可选）**

  配置要包含的 MVC 从配置文件。

- **Actions 定义：actions（可选，数量：0 – N）**

  配置 Action 及其 Results 和 Exception Mappings。主配置文件和从配置文件的 Actions 配置方法是一样的。

主配置文件的配置结构定义如下图所示：



上面的结构定义图看起来有点复杂，但实际配置起来是非常简单的。先看一个典型的配置文件示例：

```
<?xml version="1.0" encoding="UTF-8"?>
<app xmlns="http://www.jessma.org"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.jessma.org
        http://www.jessma.org/schema/mvc-config-main-3.5.xsd">
    <global>
        <!-- request 和 response 的默认编码（可选，默认：不设置） -->
```

```
        <encoding>UTF-8</encoding>
        <!-- Action 请求的后缀（可选，默认：.action） -->
        <action-suffix>.action</action-suffix>
        <!-- 指定 base path （可选，默认：type="auto", href=""） -->
        <base-path type="auto" />
        <!--
            default-locale    ：应用程序默认语言选项（默认：与当前操作系统一致）
            default-bundle    ：应用程序默认 Bundle（默认：res.application-message）
        -->
        <i18n default-locale="en_US" default-bundle="com.bruce.res.app-msg" />
        <!--
            enable      ：是否开启 Bean Validation 机制（默认：开启）
            bundle      ：验证文本消息的 Bundle（默认：res.validation-message）
            validator   ：验证器类（默认：org.jessma.mvc.validation.HibernateBeanValidator）
        -->
        <bean-validation
            enable="true"
            bundle="com.bruce.res.vld-msg"
            validator=""
        />
        <!-- Action 拦截器 -->
        <action-filters>
            <filter pattern="action.*" class="global.ActionFilter1"/>
            <filter pattern="action.test.*" class="global.ActionFilter2"/>
        </action-filters>
        <!-- Result 路径别名 -->
        <result-path-aliases>
            <alias name="index" path="/jsp/index.jsp"/>
            <alias name="jsp_base" path="/jsp/test"/>
        </result-path-aliases>
        <!-- 全局 Results -->
        <global-results>
            <result name="none" type="finish"/>
            <result name="login">/jsp/login.jsp</result>
            <result name="$9">/jsp/error_1.jsp</result>
            <result name="exception">jsp/error_2.jsp</result>
        </global-results>
        <!-- 全局 exception mappings -->
        <global-exception-mappings>
            <exception-mapping exception="java.lang.RuntimeException" result="$9"/>
            <exception-mapping exception="java.lang.Exception" result="exception"/>
        </global-exception-mappings>
</global>
```

```
    <!-- 包含从配置文件 -->
    <include file="mvc-config-1.xml"/>
    <include file="mvc-config-2.xml"/>

    <!-- Actions 定义 - 1 -->
    <actions>
        <action name="index">
            <result>jsp/index.jsp</result>
        </action>
        <action name="testdownload" class="action.test.TestDownload" />
        <action name="checkupload" class="action.test.CheckUpload">
            <result name="success">/jsp/test/upload/success.jsp</result>
            <result name="error">/jsp/test/upload/error.jsp</result>
        </action>
    </actions>
    <!-- Actions 定义 - 2 -->
    <actions path="/main">
        <action name="testi18n" class="action.test.TestI18N">
            <result>/jsp/test/i18n/set_locale.jsp</result>
        </action>
    </actions>
</app>
```

2.2.1 全局配置（global）

2. HTTP 编码：encoding（可选，默认：不设置）

设置 request 和 response 的字符编码格式，字符编码格式应该和 JSP 页面的编码格式一致，因此通常设置为 UTF-8 或 GBK。

```
    示例：<encoding>UTF-8</encoding>
```

3. Action 后缀名：action-suffix（可选，默认："**.action**"）

设置 Action 的后缀名，前端控制器会把包含此后缀名的请求解析为 Action 请求，并把请求转发给 Action 处理。Action 后缀名可以设置为任意值（设置为“html”也可以）。

```
    示例：<action-suffix>.do</action-suffix>
```

4. Base 路径：base-path（可选，默认：type="auto", href=""）

JessMA 支持显式设置 **${__base}** 变量以满足不同应用的需要，**base-path.type** 有三种类型：auto、manual 和 none，其中 auto 为默认类型：

- **auto**：根据当前请求的路径信息自动设置 ${__base}
  （${__base} 保存在 Request Attribute 中）

  ${__base} = *{scheme}://{server_name}:{server_port}/{app_context}*

- **manual**：根据 **base-path** 的 **href** 属性手工设置 ${__base}
  （${__base} 保存在 Application Attribute 中）

- **none**：不设置 ${__base}

示例：<base-path type="manual" href="www.baidu.com" />

5. 国际化配置：i18n（可选）

设置应用程序的默认国际化选项：

- **默认语言：default-locale（可选，默认：与 *Locale.getDefault()* 一致）**

应用程序的默认语言，默认语言必须是当前操作系统支持的语言（存在于 *Locale.getAvailableLocales()* 的返回值数组中）。

- **默认 Bundle：default-bundle（可选，默认："res.application-message"）**

应用程序的默认国际化资源 Bundle，如：在不指定 ***res*** 属性的情况下，<***p:msg***/> 标签会从默认 Bundle 中查找资源字符串。

示例：<i18n default-locale="en_US" default-bundle="com.bruce.res.app-msg" />

6. Bean 验证配置：bean-validation（可选）

设置 JSR 303 Bean Validation 选项：

- **是否开启 Bean Validation 机制：enable（可选，默认："true"）**

如果开启了 Bean Validation 机制，应用程序可对由 @FormBean 注解的 Form Bean 执行自动验证；还可以调用 *ActionSupport* 的 *validateBean() / validateBeanAndAddErrors()* 方法验证其它任何 Bean。

- **验证消息 Bundle：bundle（可选，默认："res.validation-message"）**

应用程序的验证消息 Bundle，<***p:err***/> 标签会从该 Bundle 中查找资源字符串。

- **验证器：validator（可选，默认："org.jessma.mvc.validation.HibernateBeanValidator"）**

实现了 *org.jessma.mvc.**BeanValidator*** 接口的 Bean 验证器，JessMA 提供了基于 Hibernate Validator 的 *org.jessma.mvc.validation.**HibernateBeanValidator*** 作为默认 Bean 验证器。

```
示例：
<bean-validation
    enable="true"
    bundle="com.bruce.res.validation-message"
    validator="org.jessma.mvc.validation.HibernateBeanValidator"
/>
```

7. Action 拦截器：action-filters（可选）

设置 Action 拦截器，拦截器在调用 Action 的 execute() 方法前后执行额外的代码，具体用法将在后面章节详细说明。<action-filters/> 中可以配置多个拦截器（filter），每个拦截器均需提供以下两个配置项：

- **匹配模式：pattern（可选，默认：".*"）**

pattern 为正则表达式，它与 Action 类的全名称（**{包名}.{类名}**）进行匹配，如果匹配成功，则该拦截器会拦截这个 Action。拦截器与 Action 是多对多的关系，一个拦截器可能会拦截多个 Action，一个 Action 也可能有多个拦截器。如果一个 Action 有多个拦截器，这些拦截器的执行顺序与拦截器声明的顺序一致。

- **实现类：class（必须）**

拦截器实现类，该类实现 org.jessma.mvc.ActionFilter 接口或继承 org.jessma.mvc.AbstractActionFilter。

```
示例：
<action-filters>
    <!-- 拦截 action 包及其子包的所有 Action -->
    <filter pattern="action.*" class="global.ActionFilter1"/>
    <!-- 只拦截实现类为 action.test.TestI18N 的 Action -->
    <filter pattern="action.test.TestI18N" class="global.ActionFilter2"/>
</action-filters>
```

- **入口方法：methods（可选，默认：".*"）**

methods 属性用于设置与 Action Entry 的入口方法进行匹配的正则表达式，例如下面三个 Entry 配置：

```
示例：
    <action-filters>
```

```
        <filter pattern=".*" methods="(save|update|delete)\w*" class="global.MyFilter1"/>
        <filter pattern="action.test.CheckBean.+" class="global.MyFilter2"/>
        <filter pattern="action.test.UserAction" methods="(?!find)\w*" class=" global.MyFilter3"/>
    </action-filters>
```

- ✓ **第一个 Filter：**
  拦截任何 Action 的以“save / update / delete” 开头的入口方法。
- ✓ **第二个 Filter：**
  拦截实现类名以“action.test.CheckBean”开头的 Action 的任何入口方法。
- ✓ **第三个 Filter：**
  拦截实现类名为“action.test.UserAction”的 Action，且方法名称不以“find”开头的入口方法。

8. Result 路径别名：result-path-aliases（可选）

Result 路径别名的作用是使用别名占位符置换 Action Result 配置中或 @Result 注解中的 Result Path 配置。从而简化修改或批量移动 Result Path 时引发的配置文件修改工作，只需修改别名对应的路径即可自动更新所有包含该别名的 Result Path 的实际路径。Result 路径别名需提供以下两个配置项：

- **别名：name（必须，非空）**

  为 Result 路径指定一个有意义的别名。

- **路径：path（必须）**

  别名对应的路径，可以是绝对路径或相对路径，可以是完整路径或部分路径。

```
示例：
<result-path-aliases>
    <!-- ${index} 对应路径：/jsp/index.jsp -->
    <alias name="index" path="/jsp/index.jsp"/>
    <!-- ${jsp_base} 对应路径：/jsp/test -->
    <alias name="jsp_base" path="/jsp/test"/>
</result-path-aliases>
```

当配置了上述别名时，以下几个 Action Result 是等效的：

```
1) 不使用别名
    <result>/jsp/index.jsp</result>          或注解 @Result(path="/jsp/index.jsp")
    <result>/jsp/test/my.jsp</result>        或注解 @Result(path="/jsp/test/my.jsp")
2) 使用别名
    <result>${index}</result>                或注解 @Result(path="${index}")
    <result>${jsp_base}/my.jsp</result>      或注解 @Result(path="${jsp_base}/my.jsp")
```

9. 全局 Results：global-results（可选）

当 Action 的 execute() 方法执行完成后，会得到一个 String 类型的返回值（Result Name），程序根据 Result Name 来查找配置文件的 Result 配置条目并确定后续的处理行为（如：终止处理流程或跳转到某个 JSP 文件等）。

有时，多个 Action 的 Result 会共享相同的行为，例如：那些需要用户登录才能访问的 Action 在检测到用户还没登录时会返回 Result Name 为“input”的结果值并跳转到登录页面。如果为所有这类 Action 都自定义这个 Result 会增加很多重复工作，配置文件也不美观。因此，可把这类 Result 配置为全局 Result，给所有 Action 共享。

Result 的查找规则：首先在 Action 自身配置中查找 Result，如果找不到则到全局 Result 中查找。Result 的具体配置参考：《Action Result 定义》。

```
示例：
<global-results>
    <result name="none" type="finish"/>
    <result name="login">/jsp/login.jsp</result>
</global-results>
```

注意：如果全局 Results 中没有配置“none”Result，JessMA 会为全局 Results 加入一个默认“none”Result，该 Result 的 Type 为“finish”。

10. 全局 Exception Mappings：global-exception-mappings（可选）

Action 的 execute() 方法执行过程中，可能会引发各种异常（Exception）。对于那些可预期的异常可以在 execute() 方法内部加入 try/catch 语句块捕获并处理，但有些异常是非预期的，当这类异常抛出后会向用户返回页面错误（500），这样不太友好。因此可以配置 Action 的 Exception Mapings，把异常转换为 Result，当引发某种类型的异常时转到其对应的 Result 并执行后续处理。

有时，多个 Action 的在处理某种异常时会共享相同的行为，例如：当发生运行时错误（RuntimeException）时跳转到异常报告页面。如果为所有这类 Action 都自定义这个 Exception Mapping 会增加很多重复工作，配置文件也不美观。因此，可把这类 Exception Mapping 配置为全局 Exception Mapping，给所有 Action 共享。

Exception Mapping 的查找规则：首先在 Action 自身配置中查找 Exception Mapping，如果找不到则到全局 Exception Mapping 中查找。

Exception Mapping 的具体配置参考：《Action Exception Mappings 定义》。

```
示例：
<global-results>
    <result name="exception">/jsp/some_exception.jsp</result>
</global-results>
<global-exception-mappings>
    <exception-mapping exception="java.lang.RuntimeException" result="exception"/>
</global-exception-mappings>
```

2.2.2 包含配置文件（include）

MVC 配置文件中可以包含其它配置文件，并能自动过滤重复文件。配置格式为：

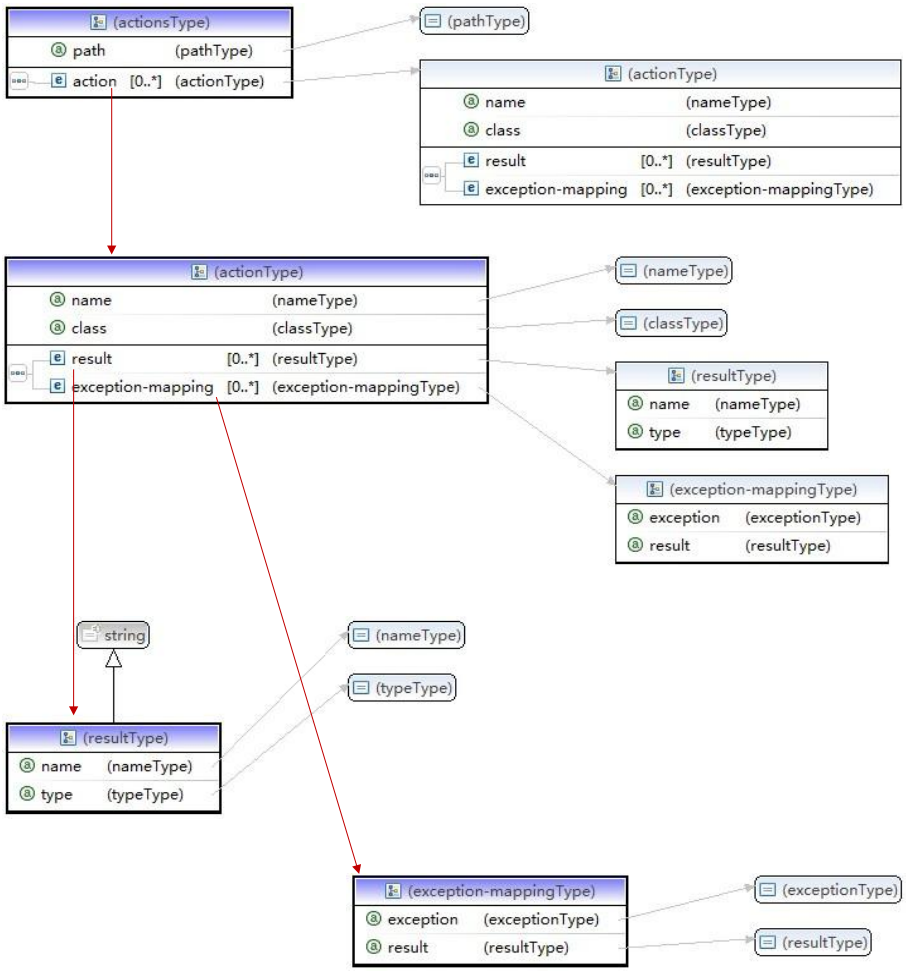
```
<include file="配置文件路径"/>
```

注意：*“配置文件路径”是相对于 ${CLASS} 目录的路径，而不是相对于主配置文件的路径。* 例如：主配置文件在 ${CLASS}/conf 目录下，要包含其相同目录下的从配置文件 mvc-config-1.xml 应该这样写：

```
<include file="conf/mvc-config-1.xml"/>
```

2.2.3 Action 定义（actions）

MVC 的主配置文件和从配置文件中都能定义多个 <actions/> 元素，在<actions/> 中可以定义多个 <action/> 元素，在 <action/> 中可以定义多个 <result/> 和 <exception-mapping/> 元素，其结构定义如下图所示：



11. Action 路径：actions.path（可选，默认："/"）

<actions/> 的 path 属性用于设置 actions 里面所有 Action 的访问路径，例如下面两个 Action 配置：

```
示例：
    <actions>
        <action name="myaction_1" />
    </actions>
    <actions path="/a/b/c">
        <action name="myaction_2" />
    </actions>
```

myaction_1 的访问路径：http://${My Host}/${My App}**/myaction_1.action**
myaction_2 的访问路径：http://${My Host}/${My App}**/a/b/c/myaction_2.action**

***** 关于 actions.path 属性的几点特别说明：**

1、 不同的 <actions/> 可以使用相同的 path 属性，只要他们所有的 <action/> 元素的名称（action.name）都不相同；不同的 <actions/> 也可以包含相同名称的 <action/>，只要他们的 path 属性不相同。
2、 对于 Result Name 为 “chain” 的 Action Result，如果没有指定目标 Action 的路径，则在默认路径中查找 Action。如：

```
示例：
    <actions path="/mypkg">
        <action name="..." class="...">
            <!-- 转向: /action1 -->
            <result name="..." type="chain">action1</result>
            <!-- 转向: /mypkg/action1 -->
            <result name="..." type="chain">./action1</result>
            <!-- 转向: /a/b/c/action1 -->
            <result name="..." type="chain">/a/b/c/action1</result>
        </action>
    </actions>
```

12. Action：action（可选）

<action/> 用于配置处理用户请求的 Action，有 name 和 class 两个属性：

- **名称：name（必须）**

Action 名称是相同路径下不同 Action 的唯一标识，前端控制器通过 **${actions.path}/${action.name}.${action-suffix}** 来确定 Action 的访问地址。

- **实现类：class（可选，默认：org.jessma.mvc.ActionSupport）**

实现业务逻辑的 Action 类，实现 org.jessma.mvc.Action 接口或继承 org.jessma.mvc.ActionSupport。如果不设置该属性则默认为 ActionSupport，ActionSupport 的 execute() 方法不做任何事情，直接返回 "success"。

13. Action Result：result（可选）

<result/> 用于配置 Action 的 execute() 方法可能的返回值及其后续处理方式，有 name 和 type 两个属性：

- **结果名称：name（可选，默认："success"）**

Action 的 execute() 方法的执行结果与 Result 的名称进行匹配，查找相应的 Result，再根据该 Result 的配置执行后续处理。理论上，Result 的名称可以随意配置，但为了更规范地使用 Result，org.jessma.mvc.**Action** 接口预定义了大量常用的 Result Name：

- ✓ Action.SUCCESS (**success**) ：操作成功
- ✓ Action.FAIL (**fail**) ：操作失败
- ✓ Action.EXCEPTION (**exception**) ：抛出异常
- ✓ Action.ERROR (**error**) ：执行错误
- ✓ Action.LOGIN (**login**) ：未登陆
- ✓ Action.INPUT (**input**) ：非法输入
- ✓ Action.ELSE (**else**) ：其他结果
- ✓ Action.NONE (**none**) ：不处理
- ✓ Action.$0 (**$0**) ：（保留）
- ✓ Action.$1 (**$1**) ：（保留）
- ✓ Action.$2 (**$2**) ：（保留）
- ✓ Action.$3 (**$3**) ：（保留）
- ✓ Action.$4 (**$4**) ：（保留）
- ✓ Action.$5 (**$5**) ：（保留）
- ✓ Action.$6 (**$6**) ：（保留）
- ✓ Action.$7 (**$7**) ：（保留）
- ✓ Action.$8 (**$8**) ：（保留）
- ✓ Action.$9 (**$9**) ：（保留）

- **结果类型：type（可选，默认：action.name == "none" ? "finish" : "dispatch"）**

Result Type 声明了执行完 Action 的 execute() 方法后的后续处理方式，有以下四个可选值：

- ✓ **dispatch** ：服务端重定向（需要配置目标 URL）
- ✓ **redirect** ：客户端重定向（需要配置目标 URL）

- ✓ **chain** ：转到下一个 Action（需要配置目标 Action Name）
- ✓ **finish** ：不转发（不需要配置目标）

finish 类型通常用于处理 Ajax 请求，Action 处理完 Ajax 请求后通常会把数据直接返回给客户端，因此不需要继续转发，当 Result Name 为 "none" 的时候，它为默认 Result Type；**chain** 类型处理多个 Action 协同工作的情形，例如一个负责注册的 Action 处理完毕后转给登录 Action 处理，让用户注册的同时完成登录，配置示例参考《关于 path 属性的特别说明》；**dispatch** 和 **redirect** 类型需要配置目标 URL（通常为服务器内部的 JSP），URL 有两种形式：相对地址或绝对地址。必须注意其差别，参考下面的例子：

```
示例：
    <actions path="/a/b/c">
        <action name="..." class="...">
            <!-- 转向: /a/b/c/xyz.jsp -->
            <result name="..." type="dispatch">xyz.jsp</result>
            <!-- 转向: /xyz.jsp -->
            <result name="..." type=" dispatch">./xyz.jsp</result>
            <!-- 转向: /a/b/c/mypkg/xyz.jsp -->
            <result name="..." type=" dispatch">mypkg/xyz.jsp</result>
            <!-- 转向: /mypkg/xyz.jsp -->
            <result name="..." type=" dispatch">/mypkg/xyz.jsp</result>
        </action>
    </actions>
```

注意：

- ✓ ***接收 dispatch 重定向的 JSP 页面可以使用 EL 表达式 ${__action} 获取 Action 对象***
- ✓ ***接收 chain 转发的 Action 可以使用 getRequestAttribute("__action") 获取前一个 Action 对象***

- **Result Path：（可选，Result 节点的值）**

Result Path 指定 Action 的跳转路径，可以是相对路径或绝对路径（见：上节示例）。

14. Action Exception Mapping：exception-mapping（可选）

<exception-mapping/> 用于配置 Action 的 execute() 方法抛出异常时的处理措施，异常抛出后会转换为相应的 Action Result。<exception-mapping/>有 exception 和 result 两个属性：

- **异常类型：exception（可选，默认：java.lang.Exception）**

异常类型指定要捕捉的任意异常，必须为 java.lang.Exception 或其子类。

- **结果类型：result（可选，默认：Action.EXCEPTION ("exception")）**

异常抛出时，被转换成某个 Action Result，该 Action Result 必须在 Action 的 Result

中配置或在全局 Result 中配置。

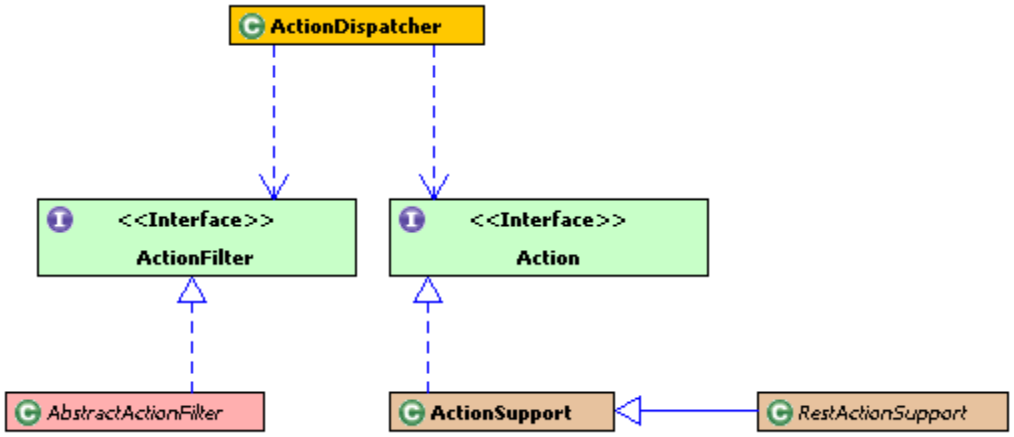
关于 exception-mapping 规则说明：

1、 先检查 Action 局部配置，如果找不到匹配项再检查全局配置
2、 按顺序续行查找匹配，如果找到匹配行则停止查找
3、 exception 匹配指定的类及其子类，因此应该把继承层次较高的异常类配置在后面
4、 即使是全局配置项中找到的 exception-mapping 配置条目，其 result 也会先从 Action 的局部 result 开始查找

注意：在处理异常的 JSP 页面中可以使用 EL 表达式 ${__exception} 获取异常对象。

3 应用篇（二）—— Action 使用

3.1 Action 接口和 ActionSupport



Action 接口（org.jessma.mvc.Action）定义了程序常用的几个 Action Result Name 常量（参考《Action Result》）以及以下几个方法：

- **Action 入口方法：String execute() throws Exception**

execute() 方法是 Action 处理请求的入口，因此，所有 Action 都应该实现这个方法处理各自的业务流程并生成数据模型，execute() 方法执行完毕后返回一个 Action Result Name，应用程序根据 Result 的配置转到相应页面展现数据或执行其他工作。

org.jessma.mvc.ActionSupport 实现了 ***Action*** 接口，并提供了 ***execute()*** 方法的默认实现：*不做如何事情，直接返回“success”*。

- **输入校验方法：boolean validate()**

Action 在执行 execute() 方法前会先调用 validate() 方法，可以在此方法中手工执行输入校验工作，如果校验不通过返回 false，此时会跳过 execute() 方法，并定向到 *Action.INPUT* 视图。因此，**实现了 validate() 方法的 Action 应该配置 *“input”* Result**。ActionSupport 的 validate() 方法默认 *不做如何事情，只是直接返回 true*。

- **上下文感知方法：**

```
void setRequest(HttpServletRequest request)
void setResponse(HttpServletResponse response)
void setServletContext(ServletContext servletContext)
```

在创建完 Action 对象后，会调用上述三个方法设置 Action 的上下文对象（ServletContext、Request、Response 和 Session）。

普通 Action 应该直接继承 ActionSupport 而不是从头实现 Action 接口，因为 ActionSupport 提供了大量用于处理通用功能的辅助方法。ActionSupport 的辅助方法有以下几类：

- **获取 HTTP 对象方法：**getRequest()、getResponse()、getSession()、……
- **获取 HTTP 对象属性方法：**getXxxAttribute()、……
- **获取请求参数方法：**getXxxParam()、……
- **Cookie 处理方法：**getCookie()、addCookie()、……
- **Locale 处理方法：**get/setLocale()、get/setLocaleByCookie()、……
- **Form Bean 装配方法：**createFormBean()、fillFormBeanProperties()、……
- **Bean 验证方法：**validateBean()、validateBeanAndAddErrors()、……
- **验证消息相关方法：**addError()、addErrorByResource()、getError()、……

◆ **Action 的内置对象**

Action 在处理请求时会在适当的时机创建一些内置对象供页面使用（对象名称在 **Action.Constant** 中声明），可以用 JSP 代码或 EL 表达式在页面中访问这些内置对象：

常 量	类 型	范 围	含 义
REQ_ATTR_ACTION = *"__action"*	**Action**	**Request**	当前的 Action 对象
REQ_ATTR_EXCEPTION = *"__exception"*	**Exception**	**Request**	Action 抛出的 Exception
REQ_ATTR_BASE_PATH = *"__base"*	**String**	**Request**	当前请求的 BASE URL
I18N_ATTR_LOCALE = *"__local"*	**Locale**	**Request** **Session**	当前请求或 Session 的 Locale
APP_ATTR_BASE_PATH = *"__base"*	**String**	**Application**	全局 BASE URL
APP_ATTR_BASE_TYPE = *"__base_type"*	**BaseType**	**Application**	全局 BASE TYPE
APP_ATTR_CONTEXT_PATH = *"__context"*	**String**	**Application**	应用程序的 Context Path
APP_ATTR_DEFAULT_APP_BUNDLE = *"__default_app_bundle"*	**String**	**Application**	应用程序默认 Bundle
APP_ATTR_DEFAULT_VLD_BUNDLE = *"__default_vld_bundle"*	**String**	**Application**	验证信息默认 Bundle

3.2 Form Bean 装配

普通 JSP 应用程序在处理客户端的请求参数时，通常利用 **request.getParameter()** 系列获取方法获取请求参数，然后执行类型转换把参数值转为目标类型。ActionSupport 为简化这类操作提供了 **getXxxParam()** 系列辅助方法。但有时候这些辅助方法还不够简便，如果

客户端提交的表单包含很多请求参数（如：用户注册表单），把这些参数一个个取出来后再转换还是需要不少的代码量，编写起来也很乏味。

请求表单通常会对应某个 VO 对象或对应 Action 的某些属性，JessMA 的 MVC 框架基于这一现实情形提供了三种 Form Bean 装配方式：

1、 **createFormBean() / fillFormBeanProperties() 半自动装配**
2、 **@FormBean(value="<属性名>") 全自动装配**（value 参数为需要装配的 VO 属性名）
3、 **@FormBean 全自动装配**（不带 value 注解参数，Action 对象本身作为被装配的 VO）

注意：《多入口 Action》章节将会提到，可以在 Action 的入口方法中声明 @FormBean，在执行自动装配时优先使用入口方法的 @FormBean。另外，@FormBean 还能注入私有成员变量（即：没有公共 setter 方法的成员变量）并支持联级注入。

方式 1 非常灵活，被装配的表单域的名称与 VO 的属性名可以不一致，可以自行决定装配时机。createFormBean() / fillFormBeanProperties() 内部调用 org.jessma.util.BeanHelper 的 createBean() / setProperties() 实现装配，这两个方法在装配文件上传表单中的非文件表单域时非常有用。

方式 2 使用简单，在执行 Action 的 execute() 方法前已经自动完成装配操作，无需人工干预，但要求被装配的表单域的名称与 VO 的属性名称一致，这种方式非常适合于装配表单域较多的表单。

方式 3 与方式 2 类似，他们之间的区别是在方式 3 中 Action 对象本身作为被装配的 VO。从程序设计的角度考虑，Action 通常不应包含太多属性，因此这种装配方式适合于装配表单域较少的表单（如：用户登录表单）。

MVC 框架支持对以下类型的 VO 属性执行自动装配：

- ✓ **8 种基础数据类型及其包装类型**
- ✓ **8 种基础数据类型及其包装类型的数组**
- ✓ **java.lang.String 和 java.util.Date 类型**
- ✓ **java.lang.Collection 及其子类型（如：List、Set）**
- ✓ **嵌套 Bean 中的上述类型的属性（如：aaa.bbb.xxx）**

注意：Collection 类型的属性必须声明泛型参数（如：List<Integer>、HashSet<Date>），否则不会执行装配。

下面通过三个例子展示这三种装配方式的使用方法，它们实现相同的效果：当用户按“确定”按钮提交表单后显示用户资料：

3.2.1 createFormBean() / fillFormBeanProperties()

1、 创建 VO 类：vo.Person

```
package vo;

import java.util.Date;
import java.util.List;

public class Person
{
	private String firstName;
	private String lastName;
	private Date birthday;
	private boolean gender;
	private int workingAge;
	private List<Integer> interest;
	private List<String> photos;

	// getters & setters
	// .....
}
```

上面的表单中没有“photos”表单域，因此 Person 的 photos 属性不会被装配。

2、 修改 /jsp/index.jsp：加入测试链接

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
```

```
<%@include file="jessma-base.jsp" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <base href="${__base}">
    <title><p:msg key="jsp-index.header"/></title>
     <meta http-equiv="pragma" content="no-cache">
     <meta http-equiv="cache-control" content="no-cache">
     <meta http-equiv="expires" content="0">
  </head>

  <body>
     <br>
    <p:msg key="jsp-index.sayhello" p0="${times}"/>
    <br>
    <br>
    <ol>
        <li><a href="test/testBean_1.action">测试 Bean 装配 - 1</a></li>
    </ol>
    <br>
  </body>
</html>
```

上面的链接的目标地址是 test/testBean_1.action，并没有直接指向/jsp/test/bean/test_bean_1.jsp，这是一个技巧，到第 6 步我们将会解开这个谜团。

3、创建表单页面：/jsp/test/bean/test_bean_1.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@include file="../../jessma-base.jsp" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <base href="${__base}">
    <title>测试 Bean 装配 - 1</title>
   <meta http-equiv="pragma" content="no-cache">
   <meta http-equiv="cache-control" content="no-cache">
   <meta http-equiv="expires" content="0">
  </head>

  <body>
  <br>
```

```
        <div align="right">
            <a href="index.action">首 页</a>
        </div>
          <br><br><br><br>
          <div align="center">
            <form action="test/checkBean_1.action" method="post">
                    First Name: <input type="text" name="firstName" value="丑"><br>
                    Last Name: <input type="text" name="lastName" value="怪兽"><br>
                    Birthday: <input type="text" name="birthday" value="1978-11-03"><br>
                    Gender:  男  <input type="radio" name="gender" value="false"> 
                            女  <input type="radio" name="gender" value="true" checked="checked"><br>
                    Working age: <select name="working-Age">
                            <option value="-1">-请选择-</option>
                            <option value="3">三年</option>
                            <option value="5" selected="selected">五年</option>
                            <option value="10">十年</option>
                            <option value="20">二十年</option>
                    </select><br>
                    Interest:  游泳  <input type="checkbox" name="its" value="1" checked="checked">
                     打球  <input type="checkbox" name="its" value="2" checked="checked">
                     下棋  <input type="checkbox" name="its" value="3">
                     打麻将  <input type="checkbox" name="its" value="4">
                     看书  <input type="checkbox" name="its" value="5" checked="checked">
                    <br><br>
                    <input type="submit" value="确  定">  <input type="reset" value="重  置">
            </form>
          </div>
      </body>
    </html>
```

上面的表单中，蓝色标识的几个表单域名称（firstName / lastName / birthday / gender）与 VO 类 Person 的相应属性名称一致，而红色标识的两个表单域名称（working-Age / its）则与 Person 相应的属性名称（workingAge / interest）不一致。

4、 创建 Action：action.test.CheckBean1

```
package action.test;

import java.util.HashMap;
import java.util.Map;
import vo.Person;
import org.jessma.mvc.ActionSupport;

public class CheckBean1 extends ActionSupport
```

```
{
	@Override
	public String execute()
	{
		// 如果表单元素的名称和 Form Bean 属性名不一致则使用 keyMap 进行映射
		// key: 表单元素名称, value: Form Bean 属性名
		Map<String, String> keyMap = new HashMap<String, String>();
		keyMap.put("working-Age", "workingAge");
		keyMap.put("its", "interest");

		// 使用表单元素创建 Form Bean
		// 如果表单元素的名称和 Form Bean 属性名完全一致则不需使用 keyMap 进行映射
		Person p = createFormBean(Person.class, keyMap);

		/* 或者这样使用: 先创建 Form Bean 对象, 然后再填充它的属性 */
		// Person p = new Person();
		// fillFormBeanProperties(p, keyMap);

		// 设置 Request Attr
		setRequestAttribute("person", p);

		return SUCCESS;
	}
}
```

上面代码中的注释已清楚地描述了 Form Bean 的装配方法，Form Bean 可以通过 ActionSupport 的 createFormBean() 或 fillFormBeanProperties() 方法执行装配，两者的区别是：前者*先创建 Bean 实例，然后执行装配，最后返回这个 Bean 实例*；而后者是*对已有的 Bean 对象执行装配，填充其属性*。

5、 创建结果页面：/jsp/test/bean/result_1.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@include file="../../jessma-base.jsp" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <base href="${__base}">
    <title>测试 Bean 装配 - 1</title>
     <meta http-equiv="pragma" content="no-cache">
     <meta http-equiv="cache-control" content="no-cache">
     <meta http-equiv="expires" content="0">
  </head>
```

```
<body>
<br>
<div align="right">
    <a href="index.action">首 页</a>
</div>
   <br><br><br><br>
   <div align="center">
   <table border="1">
   <caption>Person Attributs</caption>

<tr><td>Name</td><td><c:out value="${person.firstName} ${person.lastName}"/> </td></tr>
<tr><td>Brithday</td><td><c:out value="${person.birthday}"/> </td></tr>
<tr><td>Gender</td><td><c:out value="${person.gender}"/> </td></tr>
<tr><td>Working Age</td><td><c:out value="${person.workingAge}"/> </td></tr>
<tr><td>Interest</td><td><c:forEach var="its" items="${person.interest}">
                                        <c:out value="${its}" />  
                                    </c:forEach> </td></tr>
<tr><td>Photos</td><td><c:forEach var="p" varStatus="status" items="${person.photos}">
                                        <c:out value="${p}" />
                                        <c:if test="${not status.last}"><br></c:if>
                                    </c:forEach> </td></tr>
    </table>
  </div>
</body>
</html>
```

结果页面通过 JSTL 打印出 Person 的所有属性，由于 photos 属性为 null，因此它所在的单元格将看不到任何内容。

6、 修改 MVC 配置文件，加入 Action：testBean_1 和 checkBean_1

我们希望更好地组织 MVC 配置文件，把所有测试 Action 都放在独立的从配置文件 mvc-config-t.xml 中，因此先修改主配置文件 mvc-config.xml，让其包含从配置文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<app xmlns="http://www.jessma.org"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.jessma.org xsd/mvc-config-main.xsd
        http://www.jessma.org/schema/mvc-config-main-3.5.xsd">
    <global>
        <encoding>UTF-8</encoding>
        <action-suffix>.action</action-suffix>
    </global>
```

```
    <!—包含从配置文件 -->
    <include file="mvc-config-t.xml"/>

    <actions>
        <action name="index" class="action.test.IndexAction">
            <result>jsp/index.jsp</result>
        </action>
    </actions>
</app>
```

然后在从配置文件中配置 Action：

```
<?xml version="1.0" encoding="UTF-8"?>
<app xmlns="http://www.jessma.org"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.jessma.org
        http://www.jessma.org/schema/mvc-config-include-3.5.xsd">

    <!-- 设置 Action 路径为：/test -->
    <actions path="test">
        <!-- 没有配置 action.class 属性，则 class 默认为 org.jessma.mvc.ActionSupport -->
        <action name="testBean_1">
            <!-- 没有配置 result.name，则 name 默认为 success -->
            <!-- 没有配置 result.type，则 type 默认为 dispatch -->
            <result>/jsp/test/bean/test_bean_1.jsp</result>
        </action>
        <action name="checkBean_1" class="action.test.CheckBean1">
            <result>/jsp/test/bean/result_1.jsp</result>
        </action>
    </actions>
</app>
```

（后面两个例子将不再列出完整代码，只列出关键代码！）

3.2.2 @FormBean(value="<属性名>") 注解装配

1、 修改 /jsp/index.jsp：加入测试链接

```
    <li><a href="test/testBean_2.action">测试 Bean 装配 - 2</a></li>
```

2、 创建表单页面：/jsp/test/bean/test_bean_2.jsp

（参考 MyJessMA 工程源码：WebRoot/jsp/test/bean/test_bean_2.jsp）

上面的表单中，所有表单域的名称与 VO 类 Person 的相应属性名称一致。

3、 创建 Action：action.test.CheckBean2

（参考 MyJessMA 工程源码：src/action.test.CheckBean2）

装配不需人工干预，在执行 execute() 方法前已自动完成。

4、 创建结果页面：/jsp/test/bean/result_2.jsp

（参考 MyJessMA 工程源码：WebRoot/jsp/test/bean/result_2.jsp）

结果页面通过隐藏的 Request 属性“**__action**”获取 Action 对象，再通过 Action 对象间接获取 person 的所有属性。

5、 修改 MVC 配置文件，加入 Action：testBean_2 和 checkBean_2

```
<action name="testBean_2">
    <result>/jsp/test/bean/test_bean_2.jsp</result>
</action>
<action name="checkBean_2" class="action.test.CheckBean2">
    <result>/jsp/test/bean/result_2.jsp</result>
</action>
```

3.2.3 @FormBean 注解装配（无 value 注解参数）

1、 修改 /jsp/index.jsp：加入测试链接

```
<li><a href="test/testBean_3.action">测试 Bean 装配 - 3</a></li>
```

2、 创建表单页面：/jsp/test/bean/test_bean_3.jsp

（参考 MyJessMA 工程源码：WebRoot/jsp/test/bean/test_bean_3.jsp）

上面的表单中，所有的表单域的名称都与 Action 类 **CheckBean3** 的相应属性名称一致。

3、 创建 Action：action.test.CheckBean3

（参考 MyJessMA 工程源码：src/action.test.CheckBean3）

装配不需人工干预，在执行 execute() 方法前已自动完成。这种装配方式可以理解为 Action 自身也充当了 VO 的角色。

4、 创建结果页面：/jsp/test/bean/result_3.jsp

（参考 MyJessMA 工程源码：WebRoot/jsp/test/bean/result_3.jsp）

结果页面通过隐藏的 Request 属性“**__action**”获取 Action 对象，然后再获取其属性值。

5、 修改 MVC 配置文件，加入 Action：testBean_3 和 checkBean_3

```
<action name="testBean_3">
    <result>/jsp/test/bean/test_bean_3.jsp</result>
</action>
<action name="checkBean_3" class="action.test.CheckBean3">
    <result>/jsp/test/bean/result_3.jsp</result>
</action>
```

3.3 手工输入校验

org.jessma.mvc.Action 的 validate() 方法声明了手工输入校验功能，只需在 Action 的 validate() 方法中实现校验逻辑即可。下面使用一个非常简单的例子展示输入校验的基本用法：用户在页面中输入一个昵称，如果昵称不存则转到结果页面，否则要求用户重新输入。

1、 修改 /jsp/index.jsp：加入测试链接

```
<li><a href="test/testValidate.action">测试输入校验</a></li>
```

2、 创建表单页面：/jsp/test/validate/test_validate.jsp

（参考 MyJessMA 工程源码：WebRoot/jsp/test/validate/test_validate.jsp）

3、 创建 Action：action.test.CheckValidate

（参考 MyJessMA 工程源码：src/action.test.CheckValidate）

4、 创建结果页面：/jsp/test/validate/result.jsp

（参考 MyJessMA 工程源码：WebRoot/jsp/test/validate/result.jsp）

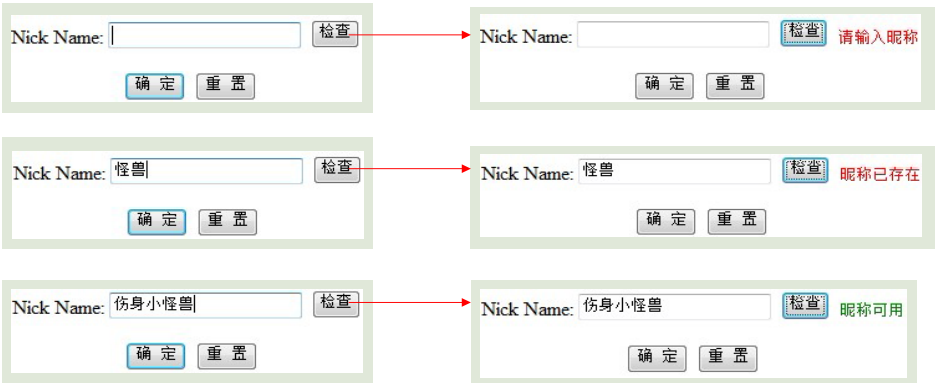
5、 修改 MVC 配置文件，加入 Action：testValidate 和 checkValidate

```
<action name="testValidate">
    <result>/jsp/test/validate/test_validate.jsp</result>
</action>
<action name="checkValidate" class="action.test.CheckValidate">
    <result>/jsp/test/validate/result.jsp</result>
</action>
```

3.4 处理 Ajax 请求

服务器在处理 Ajax 请求时，通常直接返回 XML 或 JSON 等格式化模型数据，不会跳转到任何视图。JessMA 特别定义了“**none**”Result Name 和“**finish**”ResultType 来支持这种情形。“none”Result Name 的默认 Result Type 为“finish”，“finish”Result Type 的行为是当 Action 的 execute() 方法执行完成后不跳转到任何视图。一般情况下，“none”和“finish”是成对出现的，因此可以把“none”Result 定义为全局 Result，从而避免在每个处理 Ajax 的 Action 中重复定义。

现在我们接着上面的例子展示一下 Action 处理 Ajax 请求的情形：在上例的基础上为用户输入加上 Ajax 验证，当用户点击“检查”按钮时立即检查输入的昵称是否可用。另外，为了减少例子的代码量，我们用到 jQuery，把 jQuery 的 js 文件放到${WebRoot}/js 目录中。



1、 修改表单页面：/jsp/test/validate/test_validate.jsp

（参考 MyJessMA 工程源码：WebRoot/jsp/test/validate/test_validate.jsp）

2、 创建 Action：action.test.CheckNickName

（参考 MyJessMA 工程源码：src/action.test.CheckNickName）

3、 修改 MVC 配置文件，加入 Action： checkNickName

我们不想为每个处理 Ajax 请求的 Action 都重复配置“none”Result，因此，在主配置文件 mvc-config.xml 中加入一个“none”全局 Result：

```
<global-results>
    < !-- none 的默认 Result Type 为 finish，所以不用显式指定 Result Type -->
    <result name="none" />
</global-results>
```

注意：上述“none”Result 配置并不是必须的。因为当 JessMA 检测到应用程序没有配置全局“none”Result 时，会自动创建一个 Result Type 为“finish”的全局“none”Result。

然后在从配置文件中配置 checkNickName Action：

```
< !-- 不需要为这个 Action 指定 Result -->
<action name="checkNickName" class="action.test.CheckNickName" />
```

3.5 Action 拦截器

Action 拦截器的配置方法请参考：《Action 拦截器》，它有两个主要功能：

- 在执行 Action 的 execute() 方法前后添加额外的处理逻辑
- 根据实际情况需要短路 Action 的 execute() 方法，不让其执行

Action 拦截器的工作方式与 JSP/Servlet API 中的过滤器（javax.servlet.Filter）非常相似，但它们之间有三个显著的不同点：

1) Action 拦截器只拦截 Action 的 execute() 方法，而过滤器拦截整个请求处理过程。
2) Action 拦截器的拦截依据是 Action 实现类的全名称（包名+类名），而过滤器的拦截依据是请求的 URL 路径名称。
3) Action 拦截器的 pattern 采用正则表达式，而过滤器有自己的 pattern 规则。

3.5.1 ActionFilter 接口和 AbstractActionFilter

所有拦截器都必须实现 **org.jessma.mvc.ActionFilter** 接口，该接口提供了三个方法：

- **void init()**：初始化方法，应用程序启动时调用
- **void destroy()**：销毁方法，应用程序关闭时调用
- **String doFilter(ActionExecutor executor)**：拦截方法，每当有满足拦截条件的 Action 请求到达时调用

每个拦截器的 init() 和 destroy() 方法在应用程序整个生命周期中只会调用一次，init() 方法调用的顺序与拦截器声明的顺序一致，destroy() 方法调用的顺序与 init() 方法调用的顺序相反。下面几种情况需特别说明：

- 每个拦截器实现类只有一个实例，如果多个拦截器配置为相同的实现类，则他们共享相同的拦截器实例，并且 init() 和 destroy() 方法也只调用一次。
- 如果多个拦截器共享相同拦截器实例，并且他们都满足 Action 的拦截条件，则这个实例的 doFilter() 方法在同一次请求中只会被调用一次。
- 应用程序启动时就会实例化所有拦截器对象，并根据声明顺序依次调用它们的 init() 方法，应用程序结束时则反序调用它们的 destroy() 方法。

org.jessma.mvc.AbstractActionFilter 继承了 **ActionFilter** 接口，并实现了默认的 init() 和 destroy() 方法，这两个方法是空方法，不做任何事情。因此，如果用户的拦截器不需要执行额外的初始化和销毁工作则可以直接继承 **AbstractActionFilter**，从而少写一些代码。

3.5.2 ActionExecutor 接口

ActionFilter 的 doFilter() 方法接受一个 **org.jessma.mvc.ActionExecutor** 接口类型参数，该接口提供以下几个方法：

- **Action getAction()**：获取当前被拦截的 Action 对象
- **Method getEntryMethod()**：获取当前被拦截的 Action 入口方法
- **ServletContext getServletContext()**：获取应用程序的 ServletContext 对象
- **HttpServletRequest getRequest()**：获取当前请求的 HttpServletRequest 对象
- **HttpServletResponse getResponse()**：获取当前请求的 HttpServletResponse 对象
- **String invoke() throws Exception**：调用拦截器堆栈中的下一个拦截器的 doFilter() 方法或调用 Action 的 execute() 方法

一般情况下，当前拦截器通过调用 **ActionExecutor** 的 invoke() 方法进入下一个拦截器或进入 Action，如果想终止后续处理则不调用这个方法，这种情况下，后续拦截器接收不到拦截请求，Action 的 execute() 方法也不会执行。

3.5.3 应用示例

我们现在通过一个简单的例子展示拦截器的执行规则。我们定义三个拦截器：

- 第一个拦截器拦截 action 包及其子包下所有的 Action
- 第二个拦截器拦截名称以 action.test.CheckBean 开头的所有的 Action
- 第三个拦截器拦截 action.fake 包及其子包下所有的 Action

由于 action.fake 包根本不存在，所以可以观测到第三个拦截器被加载了但不起作用。

1、 创建拦截器：global.MyActionFilter1 / 2 / 3

（参考 MyJessMA 工程源码：src/global.ActionFilter1）
（参考 MyJessMA 工程源码：src/global.ActionFilter2）

（参考 MyJessMA 工程源码：src/global.ActionFilter3）

2、 修改 MVC 主配置文件，加入拦截器

```
<action-filters>
    <filter pattern="action.*" class="global.MyActionFilter1"/>
    <filter pattern="action.test.CheckBean.+" class="global.MyActionFilter2"/>
    <filter pattern="action.fake.*" class="global.MyActionFilter3"/>
</action-filters>
```

做完上述工作后可以启动程序，观测控制台的输出结果。

注：在制作本示例的过程中，发生了一点小“意外”：在测试“输入校验”的页面中点击“检查”按钮时，发现拦截器有时“不工作”，在 CheckNickName 这个 Ajax Action 中设置的断点也没有反应，但是页面显示的结果却是正常的，真奇怪！花了好一会功夫才解决，原来是浏览器缓存所致，请求根本没有到达服务器，因此把浏览器选项设置为“每次访问网页时检查所存网页的较新版本”就 OK 了 ^_

4 应用篇（三）—— Bean Validation

JessMA 支持 JSR 303 的 Bean Validation 机制（参考：《Bean Validation 中文参考手册》）。通过该机制，应用程序可以对任何 Bean（主要是 Form Bean）执行自动验证。关于如何开启 Bean Validation 机制请参考《Bean 验证配置》。开启了 Bean Validation 机制后应用程序能提供以下功能：

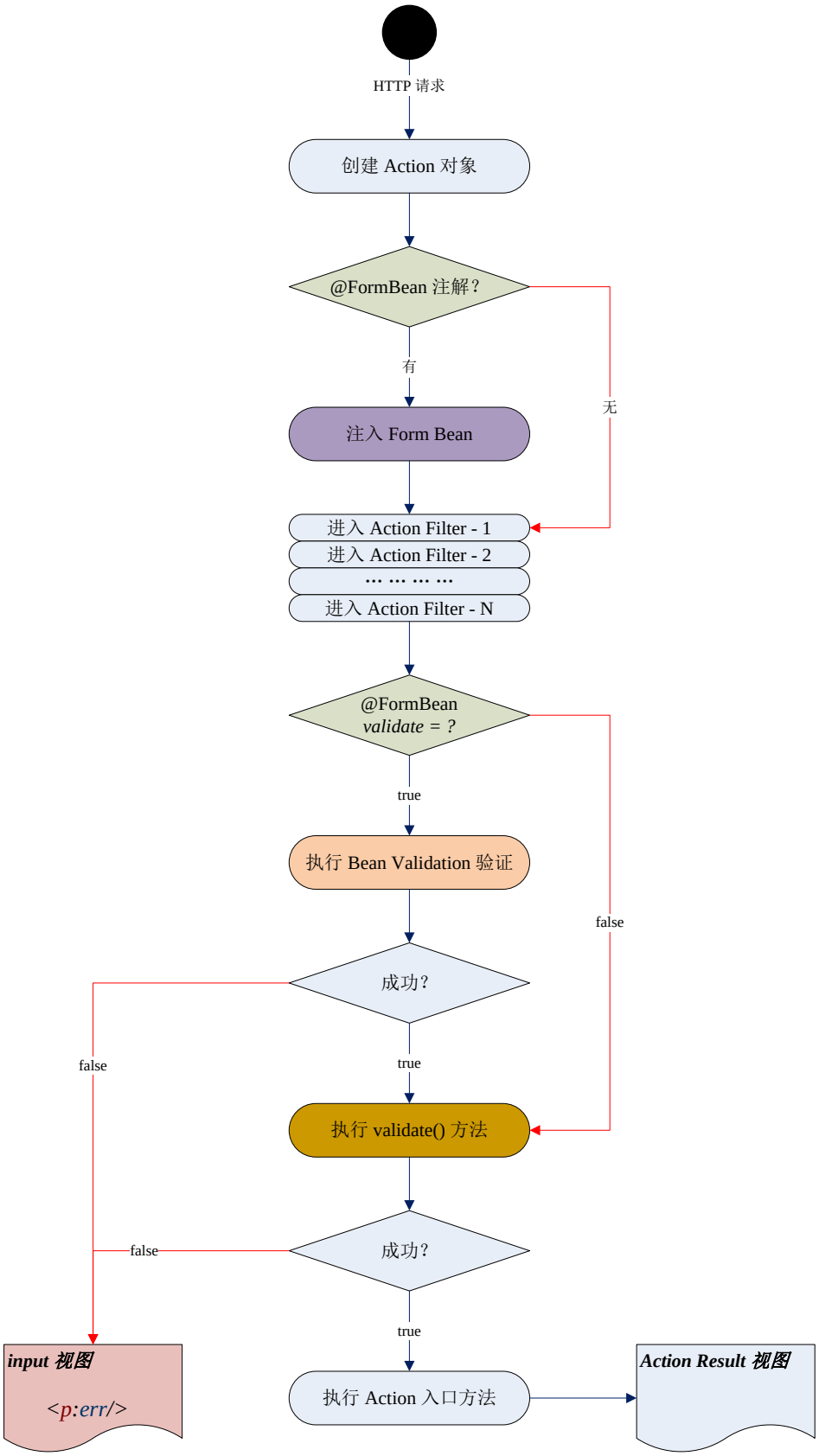
1. 应用程序可对由 @FormBean 注解的 Form Bean 执行自动验证。
2. @FormBean 注解的 ***validate()*** 和 ***groups()*** 两个属性分别指示是否对被它注解的 Form Bean 执行自动验证，以及自动验证的验证组。*validate()* 默认为：“**true**”，*groups()* 默认为：“**{}**”（默认组）。
3. 如果自动验证成功则转入 Action 的 *validate()* 方法继续执行，否则会把错误信息写入 Action 的 ***errors*** 集合并定向到 *Action.INPUT* 视图。因此，**使用了自动 Bean 验证的 Action 应该配置 *“input”* Result**。，
4. 应用程序可以调用 ActionSupport 的 ***validateBean() / validateBeanAndAddErrors()*** 方法验证其它任何 Bean。

4.1 ActionSupport 的 Bean Validation 支持

ActionSupport 提供以下方法支持 Form Bean 及其它 Bean 的自动验证：

方 法	描 述
validateBean(Object bean, Class<?> ... groups)	验证 Bean 对象，并返回验证结果集
validateBeanAndAddErrors(Object bean, Class<?> ... groups)	验证 Bean 对象，并把错误信息记录到 Action 的 ***errors*** 集合
getBeanValidator()	获取 Bean 验证器对象
isBeanValidationEnable()	检查是否开启 Bean Validation 机制
getFormBean()	获取当前 Action 入口方法的 Form Bean 对象
isAutoValidation()	检查当前 Action 入口方法是否执行自动验证
getValidationGroups()	获取当前 Action 入口方法的自动验证组
addError(String key, String message)	添加错误信息到 Action 的 ***errors*** 集合
addErrorByResource(String key, String resKey, Object ... params)	添加错误信息到 Action 的 ***errors*** 集合（根据默认 Validation Bundle 和默认 Locale 设置信息内容）
addErrorByResource(String key, String resKey, String bundle, Locale locale, Object ... params)	添加错误信息到 Action 的 ***errors*** 集合（根据指定的 Validation Bundle 和指定的 Locale 设置信息内容）
getAllErrors()	获取所有验证错误信息
getErrors(String key)	获取指定键的验证错误信息列表
getError(String key, int index)	获取指定键的第 index 条验证错误信息
getfirstError(String key)	获取指定键的第一条验证错误信息

Action 的 Form Bean 验证和整体执行如下图所示：


HTTP 请求
创建 Action 对象
@FormBean 注解？
有
无
注入 Form Bean
进入 Action Filter - 1
进入 Action Filter - 2
… … … …
进入 Action Filter - N
@FormBean
validate = ?
true
false
执行 Bean Validation 验证
成功？
false
true
执行 validate() 方法
成功？
false
true
input 视图
<p:err/>
执行 Action 入口方法
Action Result 视图

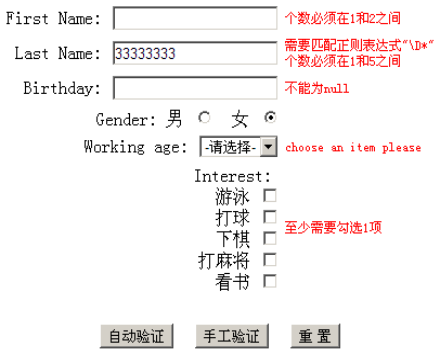
4.2 <p:err/> 标签

<p:err/> 通过指定方式输出当前 Action 的 ***errors*** 集合中的错误消息，由于验证错误信息被写入 Action 的 *errors* 集合，因此可以使用 **<p:err/>** 标签在页面中输出验证错误信息。**<p:err/>** 它拥有以下属性：

属 性	描 述
key	消息键，如果不设置则为所有消息建
index	某个键下的第 index 个消息（从 **0** 开始，默认：**-1**） *（如果设置为 **-1** 则获取该键下的所有消息）*
element	使用 DIV, SPAN 或 UL/OL 元素显示消息（默认：SPAN）
escape	是否过滤 XML 特殊字符（默认：**true**）
cssClass	DIV, SPAN 或 UL/OL 的 class 属性
cssStyle	DIV, SPAN 或 UL/OL 的 style 属性
cssAlign	DIV 的 align 属性（只对 DIV 有效）
cssId	DIV, SPAN 或 UL/OL 的 id 属性
cssDir	DIV, SPAN 或 UL/OL 的 dir 属性
cssLang	DIV, SPAN 或 UL/OL 的 lang 属性
cssTitle	DIV, SPAN 或 UL/OL 的 title 属性
cssType	UL/OL 的 type 属性（只对 UL/OL 有效）
cssCompact	UL/OL 的 compact 属性（只对 UL/OL 有效）
cssStart	OL 的 start 属性（只对 OL 有效）
cssAttrs	DIV, SPAN 或 UL/OL 的自由定义属性 *（如：cssAttrs="class='myclass' onclick='myonclick()'"）*

4.3 应用示例

本例展示如何通过 Bean Validation 机制实现 Form Bean 自动验证和手工验证。执行效果如下图所示：



1、 创建 VO 类：vo.Person2

（参考 MyJessMA 工程源码：src/vo.Person2）

2、 编辑 MVC 主配置文件 mvc-config.xml：加入 Bean Validation 支持

```
<!-- Bean Validation 配置 -->
<bean-validation
	enable="true"
	bundle="res.validation-message"
	validator="org.jessma.mvc.validation.HibernateBeanValidator"
/>
```

3、 编辑 Bean Validation 资源文件 /res/validation-message：加入验证消息文本

（参考 MyJessMA 工程源码：src/res/validation-message_zh_CN.properties）
（参考 MyJessMA 工程源码：src/res/validation-message_en_US.properties）

4、 修改 /jsp/index.jsp：加入测试链接

```
<li><a href="test/testBeanValidation.action"><p:msg key="jsp-index.TestBeanValidation"/></a></li>
```

6、 创建表单页面：/jsp/test/bean/test_bean_vld.jsp

（参考 MyJessMA 工程源码：WebRoot/jsp/test/bean/test_bean_vld.jsp）

注：上面的表单页面同时也是 Action 的 INPUT 视图页面。

7、 创建 Action：action.test.CheckBeanValidation

（参考 MyJessMA 工程源码：src/action.test.CheckBeanValidation）

注：ChckBeanValidation 利用 Action Convention 自动配置（参考：《Action Convention》章节），因此不必在 MVC 配置文件中配置该 Action。

8、 创建结果页面：/jsp/test/bean/result_vld.jsp

（参考 MyJessMA 工程源码：WebRoot/jsp/test/bean/result_vld.jsp）

9、 修改 MVC 配置文件，加入 Action：testBeanValidation

```
<action name="testBeanValidation">
	<result>/jsp/test/bean/test_bean_vld.jsp</result>
</action>
```

5 应用篇（四）—— 国际化

国际化有两方面含义：页面静态文本的多语言支持与应用程序数据的多语言支持，而后者需由上层应用实现，与框架无关。因此，JessMA 的国际化功能提供了对页面静态文本的多语言支持。从本文第一个《HelloWord 示例》就可以看出，在 JessMA 中实现国际化是非常简单的事情，只需执行三步工作：

1. 用 <p:msg/> 标签替代页面文本（其中动态数据作为标签属性传入）
2. 编辑国际化资源文件，加入相应文本条目
3. 使用 ActionSupport 的 ***setLocale()*** 方法设定当前用户的语言环境

其中，国际化资源文件是普通的 Java Properties 文件，这里不再详述。需要说明的是，资源文件中可包含{0}、{1}……{N}等占位符（占位符的顺序可以随意），这些占位符将被 <p:msg/> 标签的 p0、p1、pN 或 params 属性替代。

5.1 <p:msg/> 标签

<p:msg/> 标签用于输出国际化消息，它拥有 ***key***、***res***、***locale***、***pN*** 和 ***params*** 几个属性：

1. 文本 Key：key（必须）

文本 Key 对应资源文件中文本字符串的 key，<p:msg/> 通过文本 Key 获取实际的文本内容。

2. 资源 Bundle：res（可选，默认：由 MVC 配置项 "<i18n default-bundle='xxx'/>" 指定）

在不指定资源 Bundle 的情况下，标签会在默认资源文件中搜索文本 Key，资源文件名称格式为：

${CLASS}/${BUNDLE_PATH}/${BUNDLE_NAME}_${语言代号}_${区域代号}.properties

如果文本 Key 放在非默认的资源文件中，则需要指定 res 属性，例如：*res="mydir.my-msg-resource"* 的含义是，资源文件放在 *${CLASS}/mydir* 目录，Bundle 名称为 *my-msg-resource*，如果支持美国英语、简体中文和繁体中文，则在该目录下必须有以下三个文件：

- ✓ my-msg-resource_en_US.properties
- ✓ my-msg-resource_zh_CN.properties
- ✓ my-msg-resource_zh_TW.properties

3. 区域语言：locale（可选，默认：null，使用当前请求的区域语言）

区域语言格式为“***${语言代号}[_${区域代号}]***”（如：en_US），区域语言确定具体要读

取哪个资源文件，因为 <p:msg/> 能自动使用当前请求的区域语言，所以通常不必指定这个参数，除非一些非常特殊的情形，例如：当前的语言环境为中文，但要从英文资源文件中读取某些文本。

4. 文本参数：pN ，[*N = 0 ~ 9*]（可选，默认：null）

资源文件的文本中可能会有动态内容，例如：“*我的名字叫{0},今年{1}岁*”（“*my name is {0}, {1} years old*”）。 <p:msg/> 预定于了 p0 ~ p9 十个可选属性对应资源文件的 {0} ~ {9} 占位符。

5. 文本参数：params（可选，默认：null）

params 是指定文本参数的另一种方式，如果 <p:msg/> 同时指定了 params 和 pN，则优先使用 params 而忽略 pN，params 可以为任意类型：

- ✓ **params 为普通对象**：对应资源文本的 {0}
- ✓ **params 为数组**：数组的第 N 个元素对应资源文本的 {N}
- ✓ **params 为集合（List、Set）**：集合的第 N 个元素对应资源文本的 {N}

params 在以下两种情形下非常有用

- 当资源文本的参数超过 10 个时，只能用 params 指定参数值
- 当参数值本身就已经存放在数组或 Collection 中时，用 params 则更方便

注意：动态更新国际化资源文件请参考：《更新国际化资源文件》章节

5.2 应用示例

在本例中，我们完成两项工作：

1、 建立一个页面用于设置用户的语言环境
2、 把首页（index.jsp）显示的文本内容国际化

执行效果如下图所示：

1、 编辑资源文件 /res/application-message：加入国际化文本

message-resource_zh_CN.properties | message-resource_en_US.properties

message-resource_zh_CN.properties

name	value
jsp-index.header	首页
jsp-index.sayhello	欢迎，您是第 {0} 次进入首页！
jsp-index.TestBean	测试 Bean 装配 - {0}
jsp-index.TestValidate	测试输入校验
jsp-index.TestI18N	测试国际化
jsp-set_locale.back-to-index	返回首页
jsp-set_locale.header	国际化测试
jsp-set_locale.english	英 文
jsp-set_locale.chinese	中 文
jsp-set_locale.test	多参数测试: {0}, {1}, {2}, {3}, {4}, {9}, {8}, {7}, {6}, {5}

message-resource_zh_CN.properties | message-resource_en_US.properties

message-resource_en_US.properties

name	value
jsp-index.header	Index
jsp-index.sayhello	Well Come, You come into index page {0} times !
jsp-index.TestBean	Test Bean Assembly - {0}
jsp-index.TestValidate	Test Input Validate
jsp-index.TestI18N	Test i18n
jsp-set_locale.back-to-index	back to index
jsp-set_locale.header	i18n Test
jsp-set_locale.english	English
jsp-set_locale.chinese	Chinese
jsp-set_locale.test	test multi-params: {0}, {1}, {2}, {3}, {4}, {9}, {8}, {7}, {6}, {5}

2、 修改 /jsp/index.jsp：加入测试链接并用 <p:msg/> 替换静态文本

```
<ol>
    <li><a href="test/testBean_1.action"><p:msg key="jsp-index.TestBean" p0="1"/></a></li>
    <li><a href="test/testBean_2.action"><p:msg key="jsp-index.TestBean" p0="2"/></a></li>
    <li><a href="test/testBean_3.action"><p:msg key="jsp-index.TestBean" p0="3"/></a></li>
    <li><a href="test/testValidate.action"><p:msg key="jsp-index.TestValidate"/></a></li>
    <li><a href="test/testI18N.action"><p:msg key="jsp-index.TestI18N"/></a></li>
</ol>
```

3、 创建语言设置页面：/jsp/test/i18n/change_locale.jsp

（参考 MyJessMA 工程源码：WebRoot/jsp/test/i18n/change_locale.jsp）

4、 创建 Action：action.test.ChangeLocale

（参考 MyJessMA 工程源码：src/action.test.ChangeLocale）

5、 修改 MVC 配置文件，加入 Action：testI18N

```
<action name="testI18N" class="action.test.ChangeLocale">
    <result>/jsp/test/i18n/change_locale.jsp</result>
</action>
```

5.3 其它国际化方式

上例展示的是 JessMA 的默认国际化方式，该方式是基于 Session 的国际化（通过 ***setLocale()*** 方法设置当前的 Session 的区域语言属性），但在某些情形下（如：集群环境），应用程序可能不支持 Session。因此，JessMA 还提供了以下几种国际化方式：

1. 基于 URL 参数的国际化

在应用程序的 MVC 主配置文件中加入一个基于 URL 请求参数的国际化拦截器（***org.jessma.mvc.i18n.URLI18nFilter***），该拦截器会检查请求 URL 中是否包含名称为“*__locale*”的请求参数，如果有则把当前请求的区域语言属性设置为其参数值。

2. 基于 Cookie 的国际化

在应用程序的 MVC 主配置文件中加入一个基于 Cookie 的国际化拦截器（***org.jessma.mvc.i18n.CookieI18nFilter***），该拦截器会检查请是否包含名称为“*__locale*”的 Cookie，如果有则把当前请求的区域语言属性设置为其 Cookie 值。应用程序通过调用 ActionSupport 的 ***setLocaleByCookie()*** 方法设置“*__locale*”Cookie。

3. 基于浏览器语言首选项的国际化

在应用程序的 MVC 主配置文件中加入一个基于浏览器语言首选项的国际化拦截器（***org.jessma.mvc.i18n.BrowserI18nFilter***），该拦截器会检查请求是否包含“*accept-language*”头，如果有则把当前请求的区域语言属性设置为其指定值。

```
<!-- 例如：下面同时配置了 URL、Cookie 和 Browser 三个国际化拦截器 -->
<action-filters>
    <!-- 优先检查客户请求中是否带有名称为 '__locale' 的请求参数 -->
    <filter class="org.jessma.mvc.i18n.URLI18nFilter"/>
    <!-- 然后检查请求是否带有名称为 '__locale' 的 Cookie -->
    <filter class="org.jessma.mvc.i18n.CookieI18nFilter"/>
    <!-- 最后根据客户端浏览器的语言首选项设置当前请求的语言属性 -->
    <filter class="org.jessma.mvc.i18n.BrowserI18nFilter"/>
</action-filters>
```

注意：上述三种国际化方式都是基于请求属性（Request Attribute）的，它们的优先级高于基于 Session 属性（Session Attribute）的默认国际化方式。

5.4 应用示例

在本例中，利用 Cookie 方式实现上例的国际化功能，并展示一下基于 URL 请求参数的国际化示例。执行效果如下图所示：

1、 编辑资源文件 /res/application-message：加入相应国际化文本

（参考 MyJessMA 工程源码：src/res/application-message_zh_CN.properties）
（参考 MyJessMA 工程源码：src/res/application-message_en_US.properties）

2、 修改 /jsp/index.jsp：加入测试链接

```
<li><a href="test/testI18N_Cookie.action"><p:msg key="jsp-index.TestI18N"  p0="Cookie"/></a></li>
```

3、 创建语言设置页面：/jsp/test/i18n/change_locale_cookie.jsp

（参考 MyJessMA 工程源码：WebRoot/jsp/test/i18n/change_locale_cookie.jsp）

4、 创建 Action：action.test.ChangeLocaleByCookie

（参考 MyJessMA 工程源码：src/action.test.ChangeLocaleByCookie）

5、 修改 MVC 配置文件，加入 Action：testI18N_Cookie

```
<action name="testI18N_Cookie" class="action.test.ChangeLocaleByCookie">
    <result>/jsp/test/i18n/change_locale_cookie.jsp</result>
</action>
```

6、 修改 MVC 主配置文件，加入 URL 和 Cookie 国际化拦截器

```
<filter class="org.jessma.mvc.i18n.URLI18nFilter"/>
<filter class="org.jessma.mvc.i18n.CookieI18nFilter"/>
```

6 应用篇（五）—— 文件上传和下载

文件上传和下载是 Web 应用程序的常用功能，JessMA 为了简化文件上传和下载操作分别提供了两个帮助类：**org.jessma.mvc.FileUploader** 和 **org.jessma.mvc.FileDownloader**。这两个类的使用方法相似，主要包括以下四个使用步骤：

1. **创建实例**：创建 FileUploader 或 FileDownloader 实例
2. **设置属性**：通过 ***setter*** 系列方法设置特定属性选项
3. **执行**：调用 FileUploader.upload() 或 FileDownloader.downlaod() 执行上传或下载
4. **处理结果**：根据执行结果进行后续处理

从上面的操作步骤可以看出，在应用程序中使用文件上传和下载功能非常简单，所有操作都在 FileUploader 或 FileDownloader 内部实现，应用程序无需过多参与。

6.1 文件上传

文件上传功能通过 **org.jessma.mvc.FileUploader** 实现，FileUploader 内部使用 ***apache commons fileupload*** 实现组件上传，因此需要在项目中加入相关的 jar 包。

6.1.1 创建实例

FileUploader 提供五个构造函数，通过这些构造函数能预设一些常用属性，可以避免再次调用 ***setter*** 方法来设置这些属性。

6.1.2 设置属性

4. 文件保存路径：savePath（必须，*不包含文件名*）

上传文件时会检查该路径是否已存在，如果不存在则会尝试创建该路径。文件保存路径可能是相对路径或绝对路径：

- ✓ **绝对路径**：以根目录符开始（如：'/'、'D:\'），是服务器文件系统的路径
- ✓ **相对路径**：不以根目录符开始，是相对于 ${WebRoot} 的路径
  （如：'mydir/myfiles' 是指'${WebToot}/mydir/myfiles'目录）

5. 总文件大小限制：sizeMax（可选，默认：*NO_LIMIT_SIZE_MAX*）

总文件大小是指表单中所有要上传的文件的字节数总和。

6. 单个文件大小限制：fileSizeMax（可选，默认：*NO_LIMIT_FILE_SIZE_MAX*）

单文件大小是指表单中每个要上传的文件的字节数。

7. 可接受的文件类型集合：acceptTypes（可选，默认：不限制）

文件扩展名集合，不区分大小写，可加或不加 '*'和'.'（如：RAR / .jpg / *.pdf）。

8. 文件名生成器：fileNameGenerator（可选，默认：*CommonFileNameGenerator*）

文件名生成器是一个实现了 **FileUploader.FileNameGenerator** 接口的对象，在保存每个上传文件前都会询问该对象的 **String generate(FileItem item, String suffix)** 方法获取文件名。generate() 方法生成的文件名可带相对目录前缀，例如：生成的文件名为 "subdir/abc.txt"，则实际保存的文件为 ${savePath}/subdir.abc.txt。

默认文件名生成器 ***CommonFileNameGenerator*** 根据序号和系统时间信息生成唯一的文件名。如果要自定义文件名生成规则，应用程序必须提供自己的 **FileNameGenerator** 实现类，并在 generate() 方法中设置文件名生成规则。

9. commons-fileupload 上传组件相关属性（可选，默认：预设值）

通过设置 commons-fileupload 上传组件相关属性可以控制文件上传参数或获取文件上传过程通知，具体可参考 commons-fileupload 的文档。

- ✓ **factorySizeThreshold**：The default threshold above which uploads will be stored on disk.
- ✓ **factoryRepository**：Sets the directory used to temporarily store files that are larger than the configured size threshold.
- ✓ **factoryCleaningTracker**：Sets the tracker, which is responsible for deleting temporary files.
- ✓ **servletHeaderencoding**：Specifies the character encoding to be used when reading the headers of individual part. When not specified, or null, the request encoding is used. If that is also not specified, or <code>null</code>, the platform default encoding is used.
- ✓ **servletProgressListener**：Sets the progress listener.

6.1.3 执行

Action 的 execute() 方法中调用 FileUploader 的 **Result upload(HttpServletRequest request, HttpServletResponse response)** 方法执行文件上传，执行结果以枚举类型 **FileUploader.Result** 的形式返回，Result 定义了以下枚举值：

- ✓ ***SUCCESS***：成功
- ✓ ***SIZE_EXCEEDED***：失败：文件总大小超过限制
- ✓ ***FILE_SIZE_EXCEEDED***：失败：单个文件大小超过限制
- ✓ ***INVALID_CONTENT_TYPE***：失败：请求表单类型不正确
- ✓ ***FILE_UPLOAD_IO_EXCEPTION***：失败：文件上传 IO 错误
- ✓ ***OTHER_PARSE_REQUEST_EXCEPTION***：失败：解析上传请求其他异常
- ✓ ***INVALID_FILE_TYPE***：失败：文件类型不正确
- ✓ ***WRITE_FILE_FAIL***：失败：文件写入失败
- ✓ ***INVALID_SAVE_PATH***：失败：文件保存路径不正确

执行成功返回 Result.SUCCESS，返回其它值表示执行失败。执行失败时会自动删除已经保存的文件。

6.1.4 处理结果

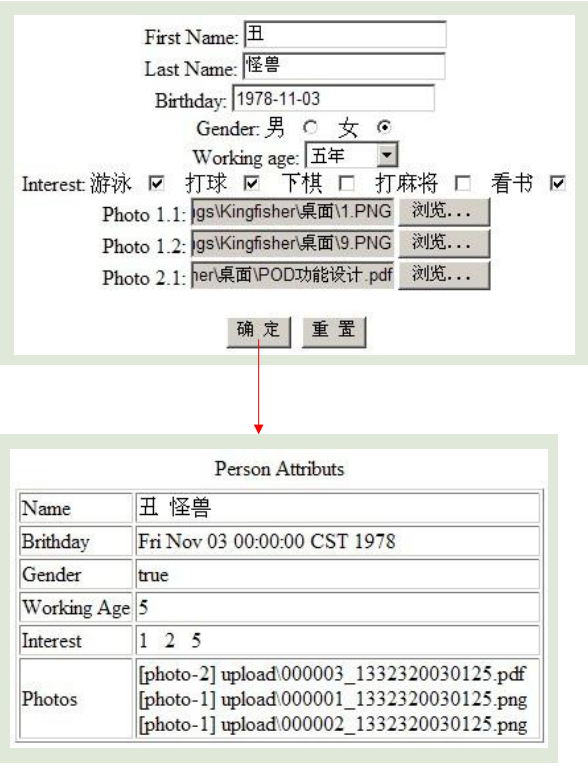
如果 FileUploader 的 upload() 方法执行成功（*Result.SUCCESS*），可以调用 **Map<String, String[]> getParamFields()** 和 **Map<String, FileInfo[]> getFileFields()** 方法分别获取**非文件表单域**和**文件表单域**的信息，返回值为 Map 类型，其中 Key 为表单域的 Name；如果 upload() 方法执行失败，可以调用 **getCause()** 方法获取包含具体错误信息的异常对象。

处理非文件表单域时，可以使用 org.jessma.util.BeanHelper 的 createBean() / setProperties() 把 getParamFields() 的返回值装配到 VO 对象中。

处理文件表单域时，通过 **FileUploader.FileInfo** 可以获取上传的源文件名称和已保存的文件的 **File** 对象。

6.1.5 应用示例

本例中，用户在表单上传页面提交若干个表单域，并在结果页面中显示已提交的信息。



1、 修改 /jsp/index.jsp：加入测试链接

```
<li><a href="test/testUpload.action">测试文件上传</a></li>
```

2、 创建表单页面：/jsp/test/upload/test_upload.jsp

（参考 MyJessMA 工程源码：WebRoot/jsp/test/upload/test_upload.jsp）

上面的表单有三个文件表单域，前两个的 Name 为 "**photo-1**" 两另外一个的 Name 为 "**photo-2**" 。另外，所有非文件表单域的名称都与 VO 类 **Person** 的相应属性名称一致。

3、 创建 Action：action.test.CheckUpload

（参考 MyJessMA 工程源码：src/action.test.CheckUpload）

Action 有两个属性 person 和 cause 以及它们的 ***getter*** 方法，用于获取表单信息和失败描述信息。person 使用 BeanHelper.createBean() 创建并填充它的非文件域属性，文件域属性则通过轮询 FileInfo[] 取得。

4、 创建结果页面：/jsp/test/upload/success.jsp 和 fail.jsp

a) success.jsp

（参考 MyJessMA 工程源码：WebRoot/jsp/test/upload/success.jsp）

b) fail.jsp

（参考 MyJessMA 工程源码：WebRoot/jsp/test/upload/fail.jsp）

5、 修改 MVC 配置文件，加入 Action：testUpload 和 checkUpload

```
<action name="testUpload">
    <result>/jsp/test/upload/test_upload.jsp</result>
</action>
<action name="checkUpload" class="action.test.CheckUpload">
    <result>/jsp/test/upload/success.jsp</result>
    <result name="fail">/jsp/test/upload/fail.jsp</result>
</action>
```

6.2 文件下载

文件下载功能通过 **org.jessma.mvc.FileDownloader** 实现，FileDownloader 支持**断点续传**与**多线程下载**，支持以**物理文件**（***File***）、**字节数组**（***byte[]***）或**字节流**（***InputStream***）作为下载源。

6.2.1 创建实例

FileDownloader 提供多个构造函数，通过这些构造函数能预设一些常用属性，可以避免再次调用 ***setter*** 方法来设置这些属性。

6.2.2 设置属性

10. 目标文件路径：filePath（*下载源为物理文件时必须*）

要下载的目标文件的路径（**包含文件名**）。文件路径可能是相对路径或绝对路径：

- ✓ **绝对路径**：以根目录符开始（如：'/'、'D:\'），是服务器文件系统的路径
- ✓ **相对路径**：不以根目录符开始，是相对于 ${WebRoot} 的路径
  （如：'mydir/myfiles/xy.jpg' 是指'${WebToot}/mydir/myfiles/xy.jpg'）

11. 目标字节数组：bytes（*下载源为字节数组时必须*）

要下载的字节数组内容。

12. 目标字节流：stream（*下载源为字节流时必须*）

要下载的字节流内容。

13. 下载模式：mode（只读）

FileDownloader 对象当前的下载模式，由 FileDownloader 的***构造函数***或 ***setFilePath()***、***setBytes()*** 与 ***setStream()***方法决定。**Mode** 定义了以下枚举值：

- ✓ ***FILE***： 物理文件
- ✓ ***BYTES***： 字节数组
- ✓ ***STREAM***： 字节流

14. 目标文件 Mime 类型：contentType（可选，默认：*"application/force-download"*）

用于设置 Response 的 content-type，如果设置为其它值可能会在浏览器中直接打开文件。

15. 默认保存文件名：saveFileName（必须）

显示在浏览器的下载对话框中的保存文件名。当下载模式为 **Mode.FILE** 时，默认与目标文件名一致；当下载模式为 **Mode.BYTES** 或 **Mode.STREAM** 时则必须显式指定。

16. 文件读写缓冲区大小：bufferSize（可选，默认：4096 字）

每次读取目标文件的字节数，如果目标文件很大，为提高下载速度可以设置为较大的值。

6.2.3 执行

Action 的 execute() 方法中调用 FileDownloader 的 **Result download(HttpServletRequest request, HttpServletResponse response)** 方法执行文件下载，执行结果以枚举类型 **FileDownloader.Result** 的形式返回，Result 定义了以下枚举值：

- ✓ ***SUCCESS***：成功
- ✓ ***ILLEGAL_STATE***：失败：非法状态
- ✓ ***ILLEGAL_ARG***：失败：非法参数
- ✓ ***FILE_NOT_FOUND***：失败：文件不存在
- ✓ ***READ_WRITE_FAIL***：失败：读写操作失败
- ✓ ***UNKNOWN_EXCEPTION***：失败：未知异常

执行成功返回 Result.SUCCESS，返回其它值表示执行失败。

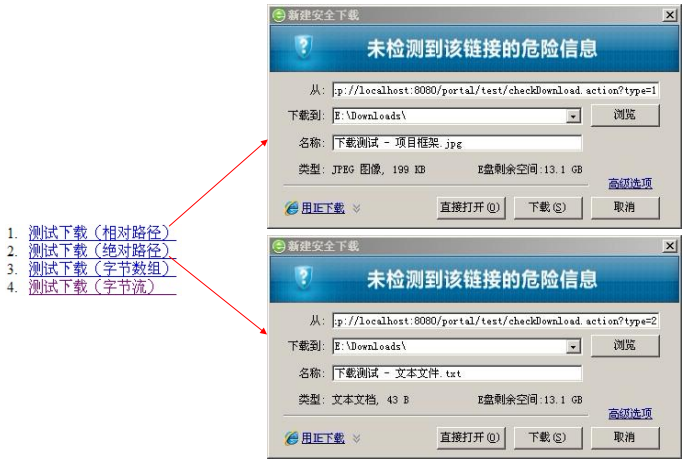
6.2.4 处理结果

调用了 Filedownloader 的 download() 方法后，**Response** 的 **OutputStream** 很可能已经被关闭，因此，**无论 download() 方法是否执行成功（*Result.SUCCESS*），Action 的 execute() 方法都应该返回"finish"类型的 Result**（结束处理，不再跳转到其他任何页面）；如果 download() 方法执行失败，可以调用 **getCause()** 方法获取包含具体错误信息的异常对象。

注意：如果客户端采用多线程方式执行下载，在下载过程中某些下载请求线程可能会接收到 Result.READ_WRITE_FAIL 类型的 Result（由于抛出了 ClientAbortException 异常导致），这是正常现象。

6.2.5 应用示例

本例提供 4 个下载测试链接，它们分别以文件相对路径、文件绝对路径、字节数组和字节流方式指定下载源。



1、 在 ${WebRoot}/download 目录中加入两个待下载的目标文件

2、 修改 /jsp/index.jsp：加入测试链接

```
<li><a href="test/testDownload.action">测试文件下载</a></li>
```

3、 创建下载页面：/jsp/test/download/test_download.jsp

（参考 MyJessMA 工程源码：WebRoot/jsp/test/download/test_download.jsp）

4、 创建 Action：action.test.CheckDownload

（参考 MyJessMA 工程源码：src/action.test.CheckDownload）

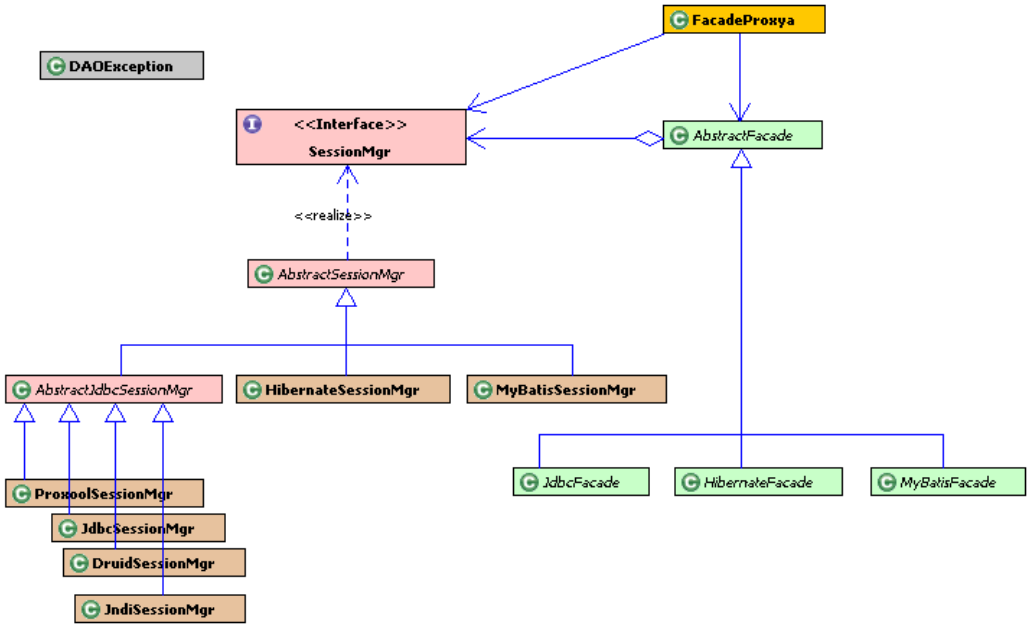
5、 修改 MVC 配置文件，加入 Action：testDownload 和 checkDownload

```
<action name="testDownload">
    <result>/jsp/test/download/test_download.jsp</result>
</action>
<action name="checkDownload" class="action.test.CheckDownload">
</action>
```

7 应用篇（六）—— DAO 框架

7.1 框架概述

DAO 框架内置支持 JDBC、MyBatis 和 Hibernate 数据库访问方法，并对数据库访问机制进行了抽象，为不同的数据库访问方法提供一致的使用接口。主要的类关系结构如下图所示：



DAO 框架主要由以下三种组件构成：

- **Session Manager：org.jessma.dao.SessionMgr**

Session Manager 实现 SessionMgr 接口，负责创建和维护数据库连接对象（Session）。JessMA 通过配置的方式在应用程序启动过程中创建 Session Manager。DAO 框架为 JDBC、MyBatis 和 Hibernate 实现了相应的 Session Manager：

- ✓ **JDBC** ：org.jessma.dao.jdbc.**JdbcSessionMgr**
  - ：org.jessma.dao.jdbc.**JndiSessionMgr**
  - ：org.jessma.dao.jdbc.**DruidSessionMgr**
  - ：org.jessma.dao.jdbc.**ProxoolSessionMgr**
- ✓ **MyBatis** ：org.jessma.dao.mybatis.**MybatisSessionMgr**
- ✓ **Hibernate** ：org.jessma.dao.hbn.**HibernateSessionMgr**

- **DAO Facade：org.jessma.dao.AbstractFacade**

DAO Facade 派生于抽象类 AbstractFacade，是所有用户 DAO 的基类，用户 DAO 继承

DAO Facade 并实现应用程序的数据库访问方法。应用程序不会直接创建 DAO Facade 或用户 DAO 对象实例，而是通过 FacadeProxy 创建其代理对象。DAO 框架为 JDBC、MyBatis 和 Hibernate 实现了各自的 DAO Facade：

- ✓ **JDBC** ：org.jessma.dao.jdbc.**JdbcFacade**
- ✓ **MyBatis** ：org.jessma.dao.mybatis.**MybatisFacade**
- ✓ **Hibernate** ：org.jessma.dao.hbn.**HibernateFacade**

- **DAO Facade Proxy：org.jessma.dao.FacadeProxy**

DAO Facade Proxy 负责创建 DAO 代理对象，该代理对象隐含事务处理和连接管理等操作。因此，**应用程序应该使用由 DAO Facade Proxy 创建的代理对象访问数据库，而不该直接创建 DAO 对象**。

当执行 DAO 代理对象的数据库访问方法时，如果发生错误会抛出 **DAOException**（org.jessma.dao.**DAOException**），它是 Runtime Exception，如果有需要，程序可以在代码中捕捉这个异常。

在应用程序中使用 DAO 框架一般需要执行以下四个步骤：

1. 在应用程序配置文件（默认：app-config.xml）中配置 Session Manager
2. 根据应用程序的需要创建 DAO 类，并实现数据库访问方法
3. 通过 DAO Facade Proxy 获取 DAO 代理对象
4. 通过 DAO 代理对象执行数据库访问方法

注意：DAO 框架是可扩展的，如果有必要，用户可以通过实现 SessionMgr 接口或继承 AbstractSessionMgr 定义新的 Session Manager；同理，也可以继承 AbstractFacade 实现新的 DAO Facade。

7.2 Session Manager

7.2.1 初始化

Session Manager 实现了 SessionMgr 接口，SessionMgr 接口中的两个方法：

- ✓ **void initialize(String ... args)**
- ✓ **void unInitialize()**

分别用于初始化和销毁 Session Manager。如果在应用程序配置文件中配置了 Session Manager，则会在应用程序启动时创建该 Session Manager 的实例并调用它的 initialize(String ... args) 方法；在应用程序关闭前调用它的 unInitialize() 方法。对于不同类型的 Session Manager，initialize(String ... args) 方法的参数个数和意义不确定，由 Session Manager 自行解析，下面分别阐述 JessMA 内置的 Session Manager 的初始化参数含义：

- **JDBC Session Manager： ProxoolSessionMgr 、 JdbcSessionMgr 、 DruidSessionMgr、JndiSessionMgr**

ProxolSessionMgr 内部使用了 Proxool 连接池，它的 initialize(...) 方法可接收 **0 ~ 2** 个参数，含义如下：

- ✓ **args[0] ：** Proxool 连接池的 connection id
  （默认："**proxool.jessma**"）
- ✓ **args[1] ：** 相对于 **${CLASS}** 目录的 Proxool 配置文件路径
  （默认："**proxool.xml**"）

（其他 JDBC Session Manager ： **JdbcSessionMgr**、**DruidSessionMgr** 和 **JndiSessionMgr** 的参数说明请参考 API 文档）

- **MyBatis Session Manager：org.jessma.dao.mybatis.MyBatisSessionMgr**

MyBatis Session Manager 的 initialize(...) 方法可接收 **0 ~ 3** 个参数，含义如下：

- ✓ **args[0] ：** 相对于 **${CLASS}** 目录的 MyBatis 配置文件路径
  （默认："**MyBatis.cfg.xml**"）
- ✓ **args[1] ：** Mybatis 配置文件中的环境（***environment***）名称
  （默认：**null**，使用 Mybatis 配置文件的默认环境）
- ✓ **args[2] ：** SQL Mapper 接口所在包名正则表达式
  （用于自动扫描 SQL Mapper 接口，默认：**null**，不自动扫描 SQL Mapper）

- **Hibernate Session Manager：org.jessma.dao.hbn.HibernateSessionMgr**

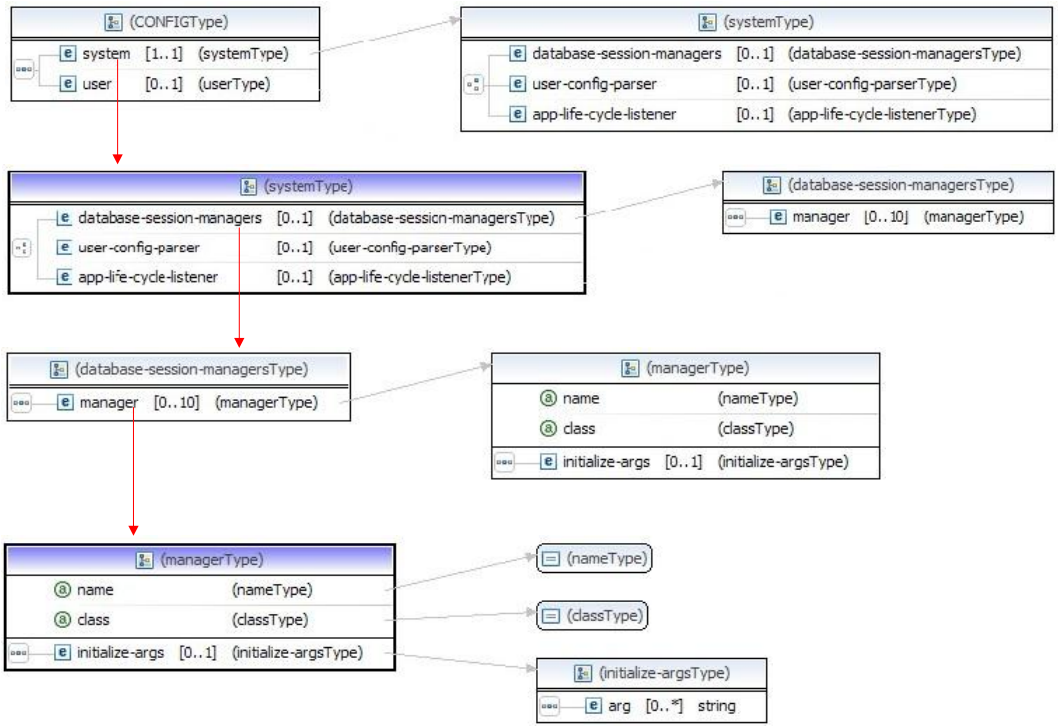
Hibernate Session Manager 的 initialize(...) 方法可接收 0 ~ 1 个参数，含义如下：

- ✓ **args[0] ：** 相对于 **${CLASS}** 目录的 Hibernate 配置文件路径
  （默认："**hibernate.cfg.xml**"）
- ✓ **args[1] ：** 实体类所在包名正则表达式
  （用于自动扫描声明了@ Entity 注解的实体类，默认：**null**，不自动扫描实体类）

注意：JessMA 内置的 Session Manager 会在初始化时检查数据库连接是否正常，如果连接不正常会抛出异常并立刻卸载应用程序。因此，在启动应用程序前应确保数据库服务器已运行。

7.2.2 配置文件

Session Manager 在应用程序配置文件的 <**system**/> 节点下的 <**database-session-managers**> 节点中配置，XSD 定义如下图所示：



每一个 <**manager**/> 节点定义一个 Session Manager，最多可定义 10 个。<manager/> 有以下两个属性：

- **名称：name（必须）**

用于标识 Session Manager，可以随意命名，但不能重复。应用程序可以使用 org.jessma.app.**AppConfig** 的 **getSessionManager**(String) 或 **getSessionManagers**() 方法获取 Session Manager。

- **实现类：class（必须）**

实现了 SessionMgr 接口的类，根据实际需要设置为 JdbcSessionMg、DruidSessionMg、MybatiSessionMgr、HibernateSessionMgr 等，甚至可以使用用户自定义的任何 SessionMgr。

如果要指定 Session Manager 的初始化参数，可以在 <manager/> 的 <**initialize-args**> 节点下设置，例如以下三个 Session Manager：

```
<system>
    <database-session-managers>
        <!-- JDBC Proxool Session Manager -->
        <manager name="ssmgr-1" class="org.jessma.dao.jdbc.ProxoolSessionMgr">
            <initialize-args>
                <!-- Connection ID：proxool.myid -->
                <arg>myid</arg>
                <!-- 配置文件：${CLASS}/myproxool.properties -->
                <arg>myproxool.properties</arg>
            </initialize-args>
```

```
        </manager>
        <!-- MyBatis Session Manager -->
        <manager name="ssmgr-2" class="org.jessma.dao.mybatis.MyBatisSessionMgr">
            <initialize-args>
                <!--配置文件：默认配置文件（${CLASS}/mybatis.cfg.xml） -->
                <arg></arg>
                <!-- 环境名称：product -->
                <arg>product</arg>
                <!-- 自动扫描位于 com.bruce.<任意子包>.mapper 中的 SQL Mapper 接口-->
                <arg> com\.bruce\..+\.mapper </arg>
            </initialize-args>
        </manager>
        <!-- Hibernate Session Manager -->
        <!-- 配置文件：默认配置文件（${CLASS}/hibernate.cfg.xml） -->
        <!-- 不自动扫描实体类 -->
        <manager name="ssmgr-3" class="org.jessma.dao.hbn.HibernateSessionMgr" />
    </database-session-managers>

    <!-- 其他配置 -->
    <!-- 。。。。。。 -->
</system>
```

7.3 DAO Facade

DAO Facade 派生于抽象类 AbstractFacade。作为用户 DAO 的基类，JessMA 内置的 DAO Facade 都提供了很多非常有用的底层数据库访问方法，用户 DAO 使用这些方法执行数据库访问能大大减少编程的代码量，这些方法的使用说明请参考 API 文档。

- **JDBC DAO Facade：org.jessma.dao.jdbc.JdbcFacade**

  - ✓ ***call*(…)** ：执行存储过程
  - ✓ ***query*(…)** ：执行查询操作
  - ✓ ***update*(…)** ：执行更新操作
  - ✓ ***updateAndGenerateKeys*(…)** ：执行更新操作，并返回被影响行的主键
  - ✓ ***updateBatch*(…)** ：执行批量更新操作

- **MyBatis DAO Facade：org.jessma.dao.jdbc.MybatisFacade**

  - ✓ ***selectXxx*(…)** ：执行查询操作
  - ✓ ***insert*(…)** ：执行插入操作
  - ✓ ***update*(…)** ：执行更新操作
  - ✓ ***delete*(…)** ：执行删除操作
  - ✓ ***clearCache*(…)** ：清空 Session 缓存
  - ✓ ***getMapper*(…)** ：获取 VO Mapper

- ✓ ***changeSessionExecutorTypeToXxx*(…)** ：更改 Session 的 Executor Type

- **Hibernate DAO Facade：org.jessma.dao.jdbc.HibernateFacade**

  - ✓ ***hqlQueryX*(…)** ：执行 HQL 查询操作
  - ✓ ***hqlUpdateX*(…)** ：执行 HQL 更新操作
  - ✓ ***namedQueryX* (…)** ：执行命名查询操作
  - ✓ ***namedUpdateX* (…)** ：执行命名更新操作
  - ✓ ***qbcQuery*(…)** ：执行 QBC 查询操作
  - ✓ ***sqlQueryX*(…)** ：执行 SQL 查询操作
  - ✓ ***sqlUpdate*(…)** ：执行 SQL 更新操作
  - ✓ ***save*(…)** ：保存 VO
  - ✓ ***update*(…)** ：更新 VO
  - ✓ ***saveOrUpdate* (…)** ：保存或更新 VO
  - ✓ ***delete* (…)** ：删除 VO
  - ✓ ***get*(…)** ：获取 VO
  - ✓ ***load*(…)** ：加载 VO
  - ✓ ***clear*(…)** ：清空 Session 缓存
  - ✓ ***flush*(…)** ：刷新 Session 缓存

JessMA 内置的 DAO Facade 有以下特点：

- ✓ DAO Facade 定义为抽象类，因此不能直接创建 DAO Facade 的实例。
- ✓ DAO Facade 有一个 ***protected*** 访问级别构造函数，该构造函数的唯一参数的类型为其对应的 Session Manager 类型，DAO Facade 会把这个参数会保存为它的 manager 属性。DAO Facade 就是通过这个 Session Manager 来确定具体要访问的数据库，由此也可以看出 DAO Facade 或 DAO 对象与 Session Manager 之间的耦合是非常松散的，通过传入不同的 Session Manager，在同一个程序中完全可以让同一个 DAO 类同时访问多个数据库（虽然这种情况并不多见）。例如：一个程序要访问两个 mysql 数据库和一个 Oracle 数据库，访问方式均为 Hibernate，因此在应用程序配置文件中配置三个 HibernateSessionMgr 类型的 Session Manager，在创建 DAO 实例（准确来说应该是 DAO 代理的实例）时为其传入不同的 Session Manager 它就会操作不同的数据库。

7.4 DAO Facade Proxy

FacadeProxy 是一个工具类，它的作用是为 DAO 类创建代理对象，该代理对象与这个 DAO 类完全兼容，能执行这个 DAO 类的所有方法，代理对象在执行 DAO 方法的过程中会在适当的时机自动执行事务代码和释放数据库连接对象。因此，DAO 的数据库访问方法只需关注数据处理逻辑，不用担心事务和连接释放等问题。因此，应用程序不应直接创建 DAO 对象，应该通过 FacadeProxy 创建它的代理对象，用这个代理对象访问数据库。

FacadeProxy 通过下面几个静态方法创建 DAO 的代理对象：

- ✧ **F getAutoCommitProxy(Class<F> *daoClass*)**
- ✧ **F getAutoCommitProxy(Class<F> *daoClass*, M *mgr*)**
- ✧ **F getManualCommitProxy(Class<F> *daoClass*)**
- ✧ **F getManualCommitProxy(Class<F> *daoClass*, M *mgr*)**
- ✧ **F getManualCommitProxy(Class<F> *daoClass*, TransIosLevel *level*)**
- ✧ **F getManualCommitProxy(Class<F> *daoClass*, M *mgr*, TransIosLevel *level*)**

通过 getAutoCommitProxy() 方法获取的代理对象不会在执行 DAO 方法的过程中插入事务管理代码，这类代理对象用于执行**纯数据库查询**（***SELECT***）操作或**手工管理事务**的操作；通过 getManualCommitProxy() 方法获取的代理对象会在执行 DAO 方法的过程中插入事务管理代码（***beging trans***、***commit***、***rollback***），这类代理对象用于执行**包含多步骤数据库更改**（***INSERT***、***UPDATE***、***DELETE***）并**框架自动管理事务**的操作。

getAutoCommitProxy() 和 getManualCommitProxy() 的重载方法说明：

- **重载方法一：接收一个 "*Class*<*F*> *daoClass*" 参数**

参数 *daoClass* 为 DAO 类的 **class** 对象，这种重载方法要求 DAO 类必须实现一个**访问级别为 protected 以上的无参数构造函数**。但是，前面已经说过在创建 DAO Facade 或 DAO 对象时，需要在其构造函数中传入一个 Session Manager 类型的参数，那么，DAO 类如何实现这个无参数的构造函数呢？参考下面的示例代码：

```
public abstract class MyBatisDao extends MyBatisFacade
{
	// 无参数构造函数
	protected MyBatisDao()
	{
		// 从 DAO 外部获取一个 Session Manager
		this(AppConfig.getSessionManager("mymgr"));
	}

	// 接收一个 Session Manager 类型参数的构造函数
	protected MyBatisDao(MyBatisSessionMgr mgr)
	{
		super(mgr);
	}

	// 其它方法 。。。
}
```

一般应用程序通常只访问一个数据库，因此，使用这种重载方法获取 DAO 代理对象的情形是最普遍的。

- **重载方法二：接收 "*Class*<*F*> *daoClass*" 和 "*M mgr*" 参数**

参数 *daoClass* 为 DAO 类的 **class** 对象，参数 *mgr* 为 Session Manager 对象，这种重载方

法要求 DAO 类必须实现一个**访问级别为 protected 以上的接收一个 *SessionMgr* 类型参数的构造函数**（如上例第二个构造函数）。这种重载方法通常只用于一个 DAO 类同时访问多个数据库的场合。

- **getManualCommitProxy() 的重载方法三：接收 "*TransIsoLevel level*" 参数**

默认情况下 getManualCommitProxy() 使用系统底层的默认事务隔离级别启动事务，如果有特殊需要可以为 getManualCommitProxy() 指定 ***TransIsoLevel*** 参数，显式设置它所创建的 DAO 代理对象的事务隔离级别。

注意：对创建用户 DAO 类的几点建议：

1、 ***把 DAO 类声明为抽象类（abstract）：****避免用户直接创建 DAO 对象*
2、 ***把 DAO 构造函数的访问级别设置为 protected：****避免用户直接创建 DAO 对象*
3、 ***为 DAO 类定义一个无参数的构造函数：****以便利用简易版 getXxxCommitProxy() 方法获取其代理对象*
4、 ***定义一个公共 DAO 基类，在这个基类中定义一个无参数的构造函数：****所有继承于该类的 DAO 就不用定义任何构造函数*

参考以下示例：

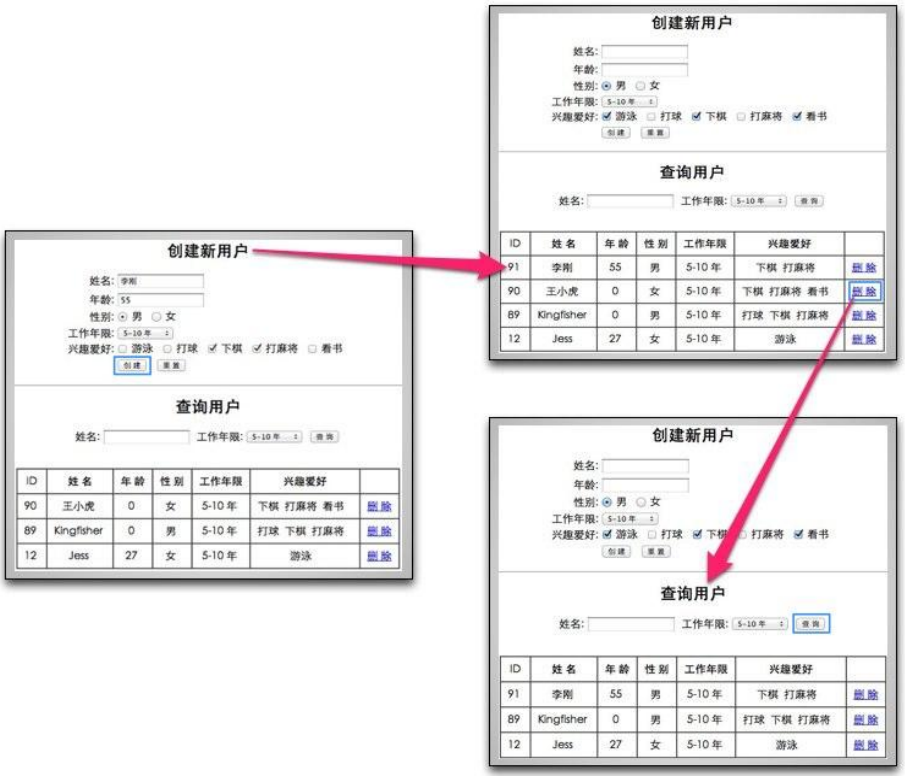
```
public abstract class MyBaseDao extends MyBatisFacade
{
	// 无参数构造函数
	protected MyBatisDao()
	{
		// 从 DAO 外部获取一个 Session Manager
		this(AppConfig.getSessionManager("mymgr"));
	}

	// 接收一个 Session Manager 类型参数的构造函数
	protected MyBatisDao(MyBatisSessionMgr mgr)
	{
		super(mgr);
	}

	// 其它方法 ...
}
```

```
public abstract class MyDaoX extends MyBaseDao
{
	// 无需定义构造函数
	// 其它方法 ...
}
```

```
public abstract class MyDaoY extends MyBaseDao
{
	// 无需定义构造函数
	// 其它方法 ...
}
```

7.5 应用示例

现在通过三个简单示例分别展示应用程序通过 DAO 框架使用 JDBC、MyBatis 和 Hibernate 访问数据库的方式和步骤，这三个示例实现相同的效果：对数据库中的 user 表执行增、删、查操作。为了尽量简单并突出重点，本示例把所有操作放都在同一个页面中实现，没有输入验证，当然也没有加入 JavaScript 控制和 Ajax 效果。示例界面如下图所示：



7.5.1 准备工作

在制作示例之前需先完成以下准备工作：

1、 创建数据库表

本例创建一个名为“**myjessma**”的 MySQL 数据库作为访问目标，并创建两张表：**user** 和 **user_interest**，分别存储用户信息和用户的兴趣爱好信息。表结构定义如下：

```
-- ----------------------------
--  Table structure for `user`
-- ----------------------------
DROP TABLE IF EXISTS `user`;
CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(50) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  `gender` int(11) DEFAULT NULL,
  `experience` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=82 DEFAULT CHARSET=utf8;

-- ----------------------------
--  Table structure for `user_interest`
-- ----------------------------
DROP TABLE IF EXISTS `user_interest`;
CREATE TABLE `user_interest` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` int(11) NOT NULL,
  `interest_id` int(11) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=187 DEFAULT CHARSET=utf8;
```

user_interest 表通过 user_id 字段与 user 表关联，当创建或删除用户时也会在 user_interest 表联级插入或删除用户相关的兴趣爱好信息。

2、 创建 VO 类

本例中，用户（User）是核心的操作对象，因此需为其创建 VO 类；另外，性别（Gender）、工作年限（Experience）和兴趣爱好（Interest）在数据库中表现为固定整数值（例如：1 - 男、2 - 女），但从另外一个角度看，它们完全可以被看作为拥有“id”和“name”两个属性的基础数据 VO（例如：id = 1 的 Gender VO 它的 name 属性值为："男"），通过为这些基础数据类型创建 VO 类能大大简化程序处理工作以及提高程序的表现力。因此，也需为这三种数据类型创建 VO 类，VO 类的定义代码：

```
package vo;

// 基础数据类型 VO 基类
public class SimpleVO
{
	private int id;
	private String name;
```

```
	public SimpleVO()
	{
	}

	public SimpleVO(int id, String name)
	{
		this.id		= id;
		this.name	= name;
	}

	// getters & setters ...
}
```

```
package vo;

public class Gender extends SimpleVO
{
	public Gender()
	{
		super();
	}

	public Gender(int id, String name)
	{
		super(id, name);
	}
}
```

```
package vo;

public class Experience extends SimpleVO
{
	public Experience()
	{
		super();
	}

	public Experience(int id, String name)
	{
		super(id, name);
	}
}
```

```
package vo;

public class Interest extends SimpleVO
{
	public Interest()
	{
		super();
	}

	public Interest(int id, String name)
	{
		super(id, name);
	}
}
```

```
package vo;

import java.util.List;

public class User
{
	private int id;
	private String name;
	private int age;
	private int gender;
	private Gender genderObj;
	private int experience;
	private Experence experienceObj;
	private List<Integer> interests;
	private List<Interest> interestObjs;

	// getters & setters
}
```

上面代码中，User VO 的定义值得注意，它除了定义数据库表对应的属性外还定义了几个额外的属性，很快我们将会看到，添加这些属性能大大简化程序处理（VO 在你手，你想怎么定义就怎么定义，只要你觉得方便 ^_*）。

3、 创建公共基础数据缓存类

上面已经提到，性别（Gender）、工作年限（Experience）和兴趣爱好（Interest）是应用程序的基础数据。把基础数据缓存起来供应用程序随时检索是非常好的实践，能大大减少数据库访问操作与降低硬编码依赖。

数据库中通常有很多基础数据库表（本例的 Gender、Experience 与 Interest 在实际项目

中通常保存在数据库表中），很多程序经常通过传入基础数据的 ID 查询数据库获取基础数据对象，这样的操作是毫无必要的，如果在程序启动时就把基础数据加载并缓存起来效果会更好。

另一种情况是硬编码，典型例子是 JSP 页面文字的硬编码。如本例中可以把性别单选按钮（radio）、工作年限列表（select）以及兴趣爱好多选按钮（checkbox）的每个选项的文字描述硬编码在页面中，但如果这样做页面将很难维护，页面代码量也会大幅增加，因此，本例采用了更聪明的方式（您很快将会看到 ^_*）。

基础数据缓存类代码：

```
package global;

import java.lang.reflect.Field;
import java.util.*;
import org.jessma.util.Logger;
import vo.Experence;
import vo.Gender;
import vo.Interest;

/** 基础数据缓存类 */
public class Cache
{
	/** 保存在 Servlet Context 中的缓存对象的 Key */
	public static final String CACHE_KEY = "__cache";
	/** 全局唯一缓存对象 */
	private static final Cache instance = new Cache();

	/** 性别列表 */
	private List<Gender> genders = new ArrayList<Gender>();
	/** 性别 Map */
	private Map<Integer, Gender> gendersMap = new HashMap<Integer, Gender>();
	/** 兴趣列表 */
	private List<Interest> interests = new ArrayList<Interest>();
	/** 兴趣 Map */
	private Map<Integer, Interest> interestsMap = new HashMap<Integer, Interest>();
	/** 工作年限列表 */
	private List<Experence> experences = new ArrayList<Experence>();
	/** 工作年限 Map */
	private Map<Integer, Experence> experencesMap = new HashMap<Integer, Experence>();

	/** 私有构造函数 */
	private Cache()
	{
	}
```

```
/** 缓存对象获取方法 */
public static final Cache getInstance()
{
	return instance;
}

/** 加载基础数据缓存 */
synchronized void loadBasicData()
{
	genders.add(new Gender(1, "男"));
	genders.add(new Gender(2, "女"));

	for(Gender o : genders)
		gendersMap.put(o.getId(), o);

	interests.add(new Interest(1, "游泳"));
	interests.add(new Interest(2, "打球"));
	interests.add(new Interest(3, "下棋"));
	interests.add(new Interest(4, "打麻将"));
	interests.add(new Interest(5, "看书"));

	for(Interest o : interests)
		interestsMap.put(o.getId(), o);

	experences.add(new Experence(1, "3 年以下"));
	experences.add(new Experence(2, "3-5 年"));
	experences.add(new Experence(3, "5-10 年"));
	experences.add(new Experence(4, "10 年以上"));

	for(Experence o : experences)
		experencesMap.put(o.getId(), o);
}

/** 卸载基础数据缓存 */
synchronized void unloadBasicData()
{
	Field[] fields = this.getClass().getDeclaredFields();

	for(Field f : fields)
	{
		Class<?> type = f.getType();

		if(type.isAssignableFrom(List.class) || type.isAssignableFrom(Map.class))
```

```
                {
                    try
                    {
                        f.set(this, null);
                    }
                    catch(Exception e)
                    {
                        LogUtil.getJessMALogger().exception(e,
                            String.format("unload basic data '%s'", f), Logger.Level.ERROR, true);
                    }
                }
            }
        }

        /** 通过 ID 查找 Gender 对象 */
        public Gender getGenderById(Integer id)
        {
            return gendersMap.get(id);
        }

        /** 通过 ID 查找 Interest 对象 */
        public Interest getInterestById(Integer id)
        {
            return interestsMap.get(id);
        }

        /** 通过 ID 查找 Experence 对象 */
        public Experence getExperenceById(Integer id)
        {
            return experencesMap.get(id);
        }

        // getters ...
}
```

从上面的代码可以看出，Cache 保存了基础数据的列表（List）和映射（Map），应用程序可以通过 Cache 提供的 ***getter*** 方法或 *getXxxById()* 方法进行检索，Cache 分别使用 **loadBasicData**() 和 **unloadBasicData**() 加载和卸载缓存数据。

4、 加载公共基础数据缓存

有了 Cache 缓存类剩下的问题就是何时加载与卸载缓存数据了，最佳的加载时机是应用程序启动时，最佳的卸载时机是应用程序关闭时。你是否立刻想到——应用程序监听器？没错，就是它！如果你忘记了也不用慌张，请回顾《应用程序配置文件》章节。

修改应用程序监听器，加入缓存处理代码：

```
package global;

import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import org.jessma.app.AppLifeCycleListener;
import org.jessma.util.Logger;

public class MyLifeCycleListener implements AppLifeCycleListener
{
	Logger logger = LogUtil.getJessMALogger();

	@Override
	public void onStartup(ServletContext context, ServletContextEvent sce)
	{
		logger.info(this.getClass().getName() + " -> onStartup()");

		// 加载基础数据缓存
		Cache cache = Cache.getInstance();
		cache.loadBasicData();
		// 把缓存对象设置为全局 Attribute
		context.setAttribute(Cache.CACHE_KEY, cache);
	}

	@Override
	public void onShutdown(ServletContext context, ServletContextEvent sce)
	{
		// 删除缓存对象全局 Attribute
		context.removeAttribute(Cache.CACHE_KEY);
		// 卸载基础数据缓存
		Cache cache = Cache.getInstance();
		cache.unloadBasicData();

		logger.info(this.getClass().getName() + " -> onShutdown()");
	}
}
```

7.5.2 JDBC

1、 创建数据库连接配置文件：${CLASS}/proxool.xml

（参考 MyJessMA 工程源码：src/proxool.xml）

2、 修改应用程序配置文件，加入 JDBC Session Manager：ProxoolSessionMgr

```
<!-- 系统配置（必填） -->
<database-session-managers>
    <manager name="session-mgr-1" class="org.jessma.dao.jdbc.ProxoolSessionMgr" />
</database-session-managers>
```

由于使用了默认的配置文件和配置选项，因此不必指定 ProxoolSessionMgr 的启动参数。

3、 （可选）修改用户自定义配置解析器，保存 JDBC Session Manager 实例

本步骤不是必须的，应用程序在任何时候都可以调用 **AppConfig** 的 **getSessionManager("session-mgr-1")** 方法获取 JDBC Session Manager 的实例，但出于美观和便捷考虑，最好还是在应用程序中保存该实例，存放该实例最理想的地方是用户自定义配置解析器（关于用户配置解析器的内容请参考：《应用程序配置文件》）。

```
public class MyConfigParser implements UserConfigParser
{
    private static final String MY_HOME = "my-home";
    // JDBC Session Manager 静态属性
    private static ProxoolSessionMgr jdbcSessionMgr;

    Logger logger = LogUtil.getJessMALogger();

    @Override
    public void parse(Element user)
    {
        // 保存 JDBC Session Manager
        jdbcSessionMgr  = (ProxoolSessionMgr)AppConfig.getSessionManager("session-mgr-1");

        Element mh = user.element(MY_HOME);
        if(mh != null)
        {
            String myHome = mh.getTextTrim();
            logger.info("My Home is: " + myHome);
        }
    }

    // JDBC Session Manager 的 getter 方法
    public static final ProxoolSessionMgr getJdbcSessionMgr()
    {
        return jdbcSessionMgr;
    }
}
```

4、 创建 DAO 类：dao.jdbc.UserDAO

本例采用《对创建用户 DAO 类的几点建议》中推荐的方式创建 DAO 类，先创建一个公共 DAO 基类，然后再从该类派生出业务 DAO 类：

a) JDBC DAO 公共基类

```
package dao.jdbc;

import global.MyConfigParser;
import org.jessma.dao.jdbc.AbstractJdbcSessionMgr;
import org.jessma.dao.jdbc.JdbcFacade;

public class JdbcBaseDao extends JdbcFacade
{
	protected JdbcBaseDao()
	{
		this(MyConfigParser.getJdbcSessionMgr());
	}

	protected JdbcBaseDao(AbstractJdbcSessionMgr mgr)
	{
		super(mgr);
	}
}
```

b) UserDAO

（参考 MyJessMA 工程源码：src/dao.jdbc.UserDAO）

直接使用 JDBC API 访问数据库时，由于要手工执行 ORM 转换，因此查询操作的代码比较长。无论如何，上述代码的逻辑其实相当简单，也是比较容易理解的。ADO 定义了三个操作方法：

- ✓ **createUser**(User user) ： 创建用户
- ✓ **deleteUser**(int id) ： 删除指定 ID 的用户
- ✓ **findUsers**(String name, int experience) ： 根据姓名和工作年限查询用户

5、 修改 /jsp/index.jsp：加入测试链接

```
<li> <a href="test/dao/testJdbc.action">测试 DAO (JDBC)</a></li>
```

6、 创建 Action：JdbcCreateUser、JdbcDeleteUser、JdbcQueryUser

（参考 MyJessMA 工程源码：src/action.test.JdbcCreateUser）
（参考 MyJessMA 工程源码：src/action.test.JdbcDeleteUser）
（参考 MyJessMA 工程源码：src/action.test.JdbcQueryUser）

7、 创建测试页面：/jsp/test/dao/test_user_1.jsp

（参考 MyJessMA 工程源码：WebRoot/jsp/test/dao/test_user_1.jsp）

8、 修改 MVC 配置文件，加入 Action：createUser1、deleteUser1、queryUser1

```
<actions path="test/dao">
	<action name="testJdbc">
		<result>/jsp/test/dao/test_user_1.jsp</result>
	</action>
	<action name="createUser1" class="action.test.JdbcCreateUser">
		<result type="chain">./queryUser1</result>
	</action>
	<action name="queryUser1" class="action.test.JdbcQueryUser">
		<result>/jsp/test/dao/test_user_1.jsp?fromQueryAction=true</result>
	</action>
	<action name="deleteUser1" class="action.test.JdbcDeleteUser">
		<result type="chain">./queryUser1</result>
	</action>
</actions>
```

在 MVC 配置文件中创建了一个 path 为 "test/dao" 的 <actions/>（本例所有测试 Action 均在这里配置）。其中 createUser1 和 deleteUser1 的 Result Type 为 "chain"，当这两个 Action 处理完本身的事务后会跳转到 queryUser1，从而使页面的内容得到及时刷新。另外，在 queryUser1 的 Result 中，跳转的目标 URL 带一个参数 "fromQueryAction" 用来标识来源，目标 JSP 通过这个参数确定某些部分的显示逻辑（参考上面的 JSP 页面代码）。

7.5.3 MyBatis

1、 创建数据库连接配置文件：${CLASS}/mybatis.cfg.xml（.properties）

（参考 MyJessMA 工程源码：src/mybatis.cfg.xml）
（参考 MyJessMA 工程源码：src/mybatis.cfg.properties）

${CLASS}/mybatis.cfg.xml 使用 ${CLASS}/mybatis.cfg.properties 属性文件存储它的数据库连接 URL、用户名和密码，因此，需要创建这个属性文件，内容如下：

```
url=jdbc\:mysql\://localhost\:3306/myjessma?autoReconnect\=true&failOverReadOnly\=false&useUnicode\=true&characterEncoding\=UTF-8
```

```
username=bruce
password=ppmm
```

2、 修改应用程序配置文件，加入 MyBatis Session Manager：MyBatisSessionMgr

```
<manager name="session-mgr-2" class="org.jessma.dao.mybatis.MyBatisSessionMgr" />
```

3、 （可选）修改用户自定义配置解析器，保存 MyBatis Session Manager 实例

在 MyConfigParser 中加入 *org.jessma.dao.mybatis.MyBatisSessionMgr* 类型的静态属性 **myBatisSessionMgr**，具体操作方法和上一节类似（参考：《保存 JDBC Session Manager 实例》）。

4、 创建 VO Mapper：vo/mapper/UserMapper.java（.xml）

a) 定义 User Mapper 接口：

```
（参考 MyJessMA 工程源码：src/vo.mapper.UserMapper）
```

b) 创建 vo/mapper/UserMapp.xml

```
（参考 MyJessMA 工程源码：src/vo/mapper/UserMapper.xml）
```

5、 创建 DAO 类：dao.mybatis.UserDAO

a) MyBatis DAO 公共基类

```
package dao.mybatis;

import org.jessma.dao.mybatis.MyBatisFacade;
import org.jessma.dao.mybatis.MyBatisSessionMgr;
import global.MyConfigParser;

public class MyBatisBaseDao extends MyBatisFacade
{
	protected MyBatisBaseDao()
	{
		this(MyConfigParser.getMyBatisSessionMgr());
	}

	protected MyBatisBaseDao(MyBatisSessionMgr mgr)
	{
		super(mgr);
	}
```

```
}
```

b) UserDAO

```
（参考 MyJessMA 工程源码：src/dao.mybatis.UserDAO）
```

6、 修改 /jsp/index.jsp：加入测试链接

```
<li> <a href="test/dao/testMyBatis.action">测试 DAO (MyBatis)</a></li>
```

7、 创建 Action：MyBatisCreateUser、MyBatisDeleteUser、MyBatisQueryUser

```
（参考 MyJessMA 工程源码：src/action.test.MyBatisCreateUser）
（参考 MyJessMA 工程源码：src/action.test.MyBatisDeleteUser）
（参考 MyJessMA 工程源码：src/action.test.MyBatisQueryUser）
```

（这三个 Action 的代码与上一节 JDBC 的三个 Action 代码完全一样，***只需把 UserDao 改为 dao.mybatis.UserDao***。代码请参考：JdbcCreateUser、JdbcDeleteUser、JdbcQueryUser）

8、 创建测试页面：/jsp/test/dao/test_user_2.jsp

测试页面的代码与上一节 JDBC 的测试页面代码完全一样，只需把几个 Action 链接分别改为 ***test/dao/createUser2.action（queryUser2.action、deleteUser2.action）***。（代码请参考：《/jsp/test/dao/test_user_1.jsp》）

9、 修改 MVC 配置文件，加入 Action：createUser2、deleteUser2、queryUser2

```
<action name="testMyBatis">
    <result>/jsp/test/dao/test_user_2.jsp</result>
</action>
<action name="createUser2" class="action.test.MyBatisCreateUser">
    <result type="chain">./queryUser2</result>
</action>
<action name="queryUser2" class="action.test.MyBatisQueryUser">
    <result>/jsp/test/dao/test_user_2.jsp?fromQueryAction=true</result>
</action>
<action name="deleteUser2" class="action.test.MyBatisDeleteUser">
    <result type="chain">./queryUser2</result>
</action>
```

10、 （可选）修改日志配置文件：${CLASS}/log4j2.xml

如果希望在控制台中察看 MyBatis 的数据库访问日志和 SQL 执行记录，可以在日志配置文件中加入以下配置：

```
    <!-- MyBatis Logger -->
    <Logger name="org.apache.ibatis" level="DEBUG" additivity="false">
        <AppenderRef ref="STDOUT" />
    </Logger>
    <!-- JDBC Logger -->
    <Logger name="java.sql" level="DEBUG" additivity="false">
        <AppenderRef ref="STDOUT" />
    </Logger>
```

7.5.4 Hibernate

1、 创建 Hibernate 专用数据表：user_hbn、user_interest_hbn

vo.User 中有一个 List<Integer> 类型属性 ***interests*** 用于保存用户的兴趣爱好 ID，Hibernate 在映射 List 类型属性时需要数据表提供额外的排序字段。为了避免与其他测试示例冲突，因此为 Hibernate 示例创建专用的数据表 **user_hbn** 和 **user_interest_hbn**，并在 **user_interest_hbn** 中加入额外的排序字段 "*list_order*"。表结构定义如下：

```
-- ----------------------------
--  Table structure for `user_hbn`
-- ----------------------------
DROP TABLE IF EXISTS `user_hbn`;
CREATE TABLE `user_hbn` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(50) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  `gender` int(11) DEFAULT NULL,
  `experience` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=82 DEFAULT CHARSET=utf8;

-- ----------------------------
--  Table structure for `user_interest_hbn`
-- ----------------------------
DROP TABLE IF EXISTS `user_interest_hbn`;
CREATE TABLE `user_interest_hbn` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` int(11) NOT NULL,
  `interest_id` int(11) NOT NULL,
  `list_order` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=187 DEFAULT CHARSET=utf8;
```

2、 创建数据库连接配置文件：${CLASS}/hibernate.cfg.xml

（参考 MyJessMA 工程源码：src/hibernate.cfg.xml）

上面的 Hibernate 配置文件使用了 Proxool 数据库连接池作为数据源，数据源配置文件为 ${CLASS}/proxool-2.xml，连接池 ID 为 "***proxool-2***" （注：应该使用与上例 JDBC 数据库连接配置不同的配置文件和连接池 ID，避免重复加载等冲突问题），proxool-2.xml 的内容与上例 JDBC 连接配置文件类似，只是连接池 ID 不同：

（参考 MyJessMA 工程源码：src/proxool-2.xml）

3、 创建 VO 映射文件：${CLASS}/vo/User.hbm.xml

（参考 MyJessMA 工程源码：src/vo/User.hbm.xml）

4、 修改应用程序配置文件，加入 Hibernate Session Manager：HibernateSessionMgr

```
<manager name="session-mgr-3" class="org.jessma.dao.hbn.HibernateSessionMgr" />
```

5、 （可选）修改用户自定义配置解析器，保存 Hibernate Session Manager 实例

在 MyConfigParser 中加入 *org.jessma.dao.hbn.HibernateSessionMgr* 类型的静态属性 **hibernateSessionMgr**，具体操作方法和上一节类似（参考：《保存 JDBC Session Manager 实例》）。

6、 创建 DAO 类：dao.hbn.UserDAO

a) Hibernate DAO 公共基类

```
package dao.hbn;

import org.jessma.dao.hbn.HibernateFacade;
import org.jessma.dao.hbn.HibernateSessionMgr;
import global.MyConfigParser;

public class HibernateBaseDao extends HibernateFacade
{
	protected HibernateBaseDao()
	{
		this(MyConfigParser.getHibernateSessionMgr());
	}

	protected HibernateBaseDao(HibernateSessionMgr mgr)
	{
```

```
            super(mgr);
        }
}
```

b) UserDAO

```
（参考 MyJessMA 工程源码：src/dao.hbn.UserDAO）
```

7、 修改 /jsp/index.jsp：加入测试链接

```
<li> <a href="test/dao/testHibernate.action">测试 DAO (Hibernate)</a></li>
```

8、 创建 Action：HibernateCreateUser、HibernateDeleteUser、HibernateQueryUser

```
（参考 MyJessMA 工程源码：src/action.test.HibernateCreateUser）
（参考 MyJessMA 工程源码：src/action.test.HibernateDeleteUser）
（参考 MyJessMA 工程源码：src/action.test.HibernateQueryUser）
```

（这三个 Action 的代码与上一节 JDBC 的三个 Action 代码完全一样，***只需把 UserDao 改为 dao.hbn.UserDao***。代码请参考：JdbcCreateUser、JdbcDeleteUser、JdbcQueryUser）

9、 创建测试页面：/jsp/test/dao/test_user_3.jsp

测试页面的代码与上一节 JDBC 的测试页面代码完全一样，只需把几个 Action 链接分别改为 ***test/dao/createUser3.action（queryUser3.action、deleteUser3.action）***。（代码请参考：《/jsp/test/dao/test_user_1.jsp》）

10、 修改 MVC 配置文件，加入 Action：createUser3、deleteUser3、queryUser3

```
<action name="testHibernate">
        <result>/jsp/test/dao/test_user_3.jsp</result>
</action>
<action name="createUser3" class="action.test.HibernateCreateUser">
        <result type="chain">./queryUser3</result>
</action>
<action name="queryUser3" class="action.test.HibernateQueryUser">
        <result>/jsp/test/dao/test_user_3.jsp?fromQueryAction=true</result>
</action>
<action name="deleteUser3" class="action.test.HibernateDeleteUser">
        <result type="chain">./queryUser3</result>
</action>
```

8 应用篇（七）—— 页面静态化

页面静态化是一种非常有用的 Web 技术，常用于从页面获取数据的请求（Get Request），如：网站首页以及一些查询页面；而向页面发送数据的请求（Post/Put Request）则不必也不适合静态化。页面静态化有两方面的含义，它们的关注点不同：

1. **URL 重写：**

利用 URL 转换技术把动态页面的 URL 转换成“看似”静态页面的 URL 展现在浏览器的地址栏中。这能起到一定的信息隐藏效果，同时也有利于搜索引擎收录。由于只是简单的地址转换，并没有真正把动态页面变为静态页面，因此不会带来任何性能提升，所以这种静态化方式也称为“伪静态”。

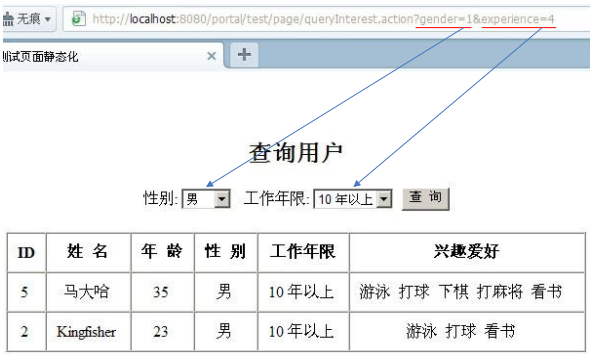
2. **页面缓存：**

利用缓存技术，把动态页面的内容缓存到内存或磁盘中，在指定的时间段内再次访问该 URL 时，应用程序会直接从缓存中获取页面内容，从而避免频繁的访问数据库和生成页面等开销，能大大提高系统的性能。

URL 重写关注页面的 URL 表现形式，而页面缓存则关注性能提升，这两种技术可以单独使用。但如果把它们有机整合起来则能得到非常好的效果。无论是 URL 重写还是页面缓存，它们的逻辑原理并不深奥，实现起来也不困难。但当前已有很多非常好用并且成熟的开源组件能实现了 URL 重写或页面缓存，并能方便灵活地集成到现有的应用程序中，因此没有必要在 JessMA 中重新实现这些功能，只需把这些开源组件集成到程序即可。下面以 **UrlRewrite** 和 **EHCache-Web** 为例展示如何在应用程序中集成 URL 重写和页面缓存功能。

8.1 查询示例

首先，我们用常规方法创建一个简单的查询示例，该示例以“性别”和“工作年限”查询用户的“兴趣爱好”，执行效果如下图所示：



每次查询时都会向“test/page/queryInterest.action”发送 Get 请求，并传递“gender”和“experience”参数。当 queryInterest Action 接收到请求后，根据请求参数设定的查询条件查

询数据库，然后把查询结果展现在 JSP 页面中。

1、 修改 VO Mapper：vo/mapper/UserMapper.java（.xml）

a) 修改 User Mapper 接口，增加查询方法 queryInterest()

```
// queryInterest(int, int) 的执行策略定义在 UserMapper.xml 中
List<User> queryInterest(@Param("gender") int gender, @Param("experience") int experience);
```

b) 修改 vo/mapper/UserMapp.xml，增加 queryInterest 查询定义

```
<select id="queryInterest" resultMap="usersResult">
    SELECT * FROM user
    <where>
      <if test="gender != 0"> gender = #{gender} </if>
      <if test="experience != 0"> AND experience = #{experience} </if>
    </where>
    ORDER BY id DESC
</select>
```

2、 修改 DAO 类：dao.mybatis.UserDAO，增加查询方法 queryInterest()

```
public List<User> queryInterest(int gender, int experience)
{
    Cache cache          = Cache.getInstance();
    UserMapper mapper    = getMapper(UserMapper.class);
    List<User> users     = mapper.queryInterest(gender, experience);

    for(User user : users)
    {
        user.setGenderObj(cache.getGenderById(user.getGender()));
        user.setExperienceObj(cache.getExperenceById(user.getExperience()));

        List<Integer> ints = user.getInterests();

        if(ints != null)
        {
            List<Interest> intsObjs = new ArrayList<Interest>();

            for(Integer i : ints)
                intsObjs.add(cache.getInterestById(i));

            user.setInterestObjs(intsObjs);
        }
```

```
            }

            return users;
        }
}
```

3、 修改 /jsp/index.jsp：加入测试链接

```
    <li><a href="test/page/queryInterest.action">测试页面静态化</a></li>
```

4、 创建 Action：action.test.QueryInterest

```
    （参考 MyJessMA 工程源码：src/action.test.QueryInterest）
```

5、 创建测试页面：/jsp/test/static/test_query.jsp

```
    （参考 MyJessMA 工程源码：WebRoot/jsp/test/static/test_query.jsp）
```

6、 修改 MVC 配置文件，加入 Action：test/page/queryInterest

```
    <actions path="test/page">
        <action name="queryInterest" class="action.test.QueryInterest">
            <result>/jsp/test/static/test_query.jsp</result>
        </action>
    </actions>
```

8.2 UrlRewrite 实现 URL 重写

UrlRewrite 为应用程序提供一个 Filter 用于实现 URL 重写功能，这个 Filter 拦截那些符合其 url-pattern 的 URL 请求，然后把该 URL 与其配置文件（urlrewrite.xml）中定义的所有 URL 映射条目进行对比匹配，如果找到匹配条目，则根据该条目的映射规则把该 URL 重写为目标 URL 后再传给后续处理模块。执行效果如下图所示：

从上图的地址栏中的 URL 来看，这个页面仿佛是一个静态的 HTML 页面，而实际上它

却是由/jsp/test/static/test_query.jsp 生成的，毫无疑问是动态页面，应用程序中也根本不存在 interest-1-4.html 这个 HTML 文件。实现这个魔法变换一点也不困难：

1、 加入 UrlRewrite 程序包

把 urlrewrite-4.0.4.jar 放入应用程序的 /WEB-INF/lib 目录。

2、 修改 /WEB-INF/web.xml：加入 UrlRewriteFilter

```
<!-- URL Rewrite 过滤器 -->
<filter>
    <filter-name>UrlRewriteFilter</filter-name>
    <filter-class>org.tuckey.web.filters.urlrewrite.UrlRewriteFilter</filter-class>
    <init-param>
        <!-- UrlRewireFilter 的配置文件（可选，默认：/WEB-INF/urlrewrite.xml） -->
        <param-name>confPath</param-name>
        <param-value>/WEB-INF/classes/urlrewrite.xml</param-value>
    </init-param>
    <init-param>
        <!-- UrlRewireFilter 的状态监控页面地址（可选，默认：/rewrite-status） -->
        <param-name>statusPath</param-name>
        <param-value>/page/rewrite-status</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>UrlRewriteFilter</filter-name>
    <!-- 拦截 URL 的 Pattern -->
    <url-pattern>/page/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
    <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

上面的配置中，UrlRewriteFilter 拦截所有以“**/page/**”开头的 URL。注意：**UrlRewriteFilter 的定义必须放在 MVC 前端控制器 ActionDispatcher 的前面**（为什么呢？）。UrlRewriteFilter 的其他初始化选项请参考：《Url Rewrite Filter 3.2.0 配置文档》。

3、 创建 UrlRewrite 配置文件：${CLASS}/urlrewrite.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE urlrewrite PUBLIC "-//tuckey.org//DTD UrlRewrite 3.2//EN"
        "http://tuckey.org/res/dtds/urlrewrite3.2.dtd">
<urlrewrite>
```

```
        <rule>
          <!--
                把 URL 格式为 /page/interest-xxx-yyy.html 的请求
                重写为：/test/page/queryInterest.action?gender=xxx&experience=yyy
          -->
            <from>/page/interest-([0-9]+)-([0-9]+).html</from>
            <to>/test/page/queryInterest.action?gender=$1&amp;experience=$2</to>
        </rule>
    </urlrewrite>
```

上述 UrlRewriteFilter 配置文件把所有符合“/page/interest-([0-9]+)-([0-9]+).html”正则表达式的 URL 请求重写为“/test/page/queryInterest.action?gender=$1&experience=$2”。可以在配置文件中定义多个映射规则条目（<**rule**>），配置文件的详细配置选项请参考：《Url Rewrite Filter 3.2.0 配置文档》。

4、 修改 /jsp/index.jsp：更换测试链接地址

```
< !-- <li><a href="test/page/queryInterest.action">测试页面静态化</a></li> -->
<li><a href="page/interest-0-0.html">测试页面静态化</a></li>
```

5、 修改测试页面：/jsp/test/static/test_query.jsp，更换查询提交链接地址

```
// var href      = $("base:first").attr("href") +
//                  "test/page/queryInterest.action?gender=$1&experience=$2";
var href         = $("base:first").attr("href") + "page/interest-$1-$2.html";
```

至此，所有配置工作已经完成。除了对页面中一些 URL 链接地址进行替换外，程序基本上不用进行修改。现在，这个查询页面神奇地变成了“静态页面”。

UrlRewriteFilter 通过映射规则（<**rule**>）把伪地址重写为实际地址。此外，它还有反向重写功能 —— 把实际地址地址重写为伪地址。这个功能非常有用，JSP 页面通常会有一些超链接用于页面跳转，UrlRewriteFilter 通过反向映射规则（<**outbound-rule**>）能把 JSP 页面中由“<**c:url**>”标签或“<%= **response**.encodeURL(*String*) %>”语句生成的 URL 进行反向重写为伪地址。下面以修改首页中本示例的测试链接为例展示反向重写功能：

1、 修改 /WEB-INF/web.xml：为 UrlRewriteFilter 加入一个 url-pattern

```
<filter-mapping>
    <filter-name>UrlRewriteFilter</filter-name>
    <!-- 拦截 URL 的 Pattern -->
    <url-pattern>/page/*</url-pattern>
    <!—拦截所有 JSP 请求 -->
    <url-pattern>*.jsp</url-pattern>
    <dispatcher>REQUEST</dispatcher>
```

```
        <dispatcher>FORWARD</dispatcher>
        <dispatcher>INCLUDE</dispatcher>
        <dispatcher>ERROR</dispatcher>
    </filter-mapping>
```

2、 修改 UrlRewrite 配置文件 urlrewrite.xml：加入 <outbound-rule> 条目

```
    <outbound-rule>
     <!--
          把 JSP 页面生成的格式为：
          /test/page/queryInterest.action?gender=xxx&experience=yyy
          的 URL 地址改写为：/page/interest-xxx-yyy.html
     -->
        <from>
            /test/page/queryInterest.action\?gender=([0-9]+)&amp;experience=([0-9]+)
        </from>
        <to>/page/interest-$1-$2.html</to>
    </outbound-rule>
```

3、 修改 /jsp/index.jsp：更换测试链接地址

```
    < !-- <li><a href="test/page/queryInterest.action">测试页面静态化</a></li> -->
    < !-- <li><a href="page/interest-0-0.html">测试页面静态化</a></li> -->
    <li><a href="
     <c:url value="/test/page/queryInterest.action?gender=0&experience=0" />
     ">测试页面静态化</a></li>
     <!—也可以写成以下形式 -->
     <!—
          <li><a href="
               <c:url value="/test/page/queryInterest.action">
                    <c:param name="gender" value="0" />
                    <c:param name="experience" value="0" />
               </c:url>
          ">测试页面静态化</a></li>
    -->
```

index.jsp 中填入的是真实地址，但从浏览器中看到的却仍然是伪地址。

8.3 EHCache-Web 实现页面缓存

EHCache-Web 在 EHCache-Core 的基础上实现，因此它的配置方法与普通的 EHCache 配置完全一致。EHCache-Web 以 Filter 的形式实现页面缓存功能，Filter 拦截那些符合其 url-pattern 的 URL 请求，并以 URL 的请求路径（包含请求参数，但不包含主机和端口信息）作为键（Key）缓存页面内容。EHCache-Web 内置多种 Filter 供应用程序使用，应用程序可

以根据需要使用其中一种或多种 Filter，常用的 Filter 有以下三种：

- **SimplePageCachingFilter**

用于缓存完整页面内容，并支持对页面内容进行 gzipping 压缩。

- **SimpleCachingHeadersPageCachingFilter**

SimpleCachingHeadersPageCachingFilter 继承于 **SimplePageCachingFilter**，除了缓存页面内容之外，它还会缓存 Etag、Last-Modified 和 Expires 等 HTTP 头信息。因此它能利用客户端浏览器的缓存机制减少对服务端的访问，但可能会发生“客户端缓存页面与服务端页面不一致”的风险。

- **SimplePageFragmentCachingFilter**

用于缓存页面片段内容（如：Header、Footer），由于页面片段内容需要被包含到外部页面中，因此 **SimplePageFragmentCachingFilter** 不会对页面内容进行 gzipping 压缩。

- **GzipFilter**

仅对页面的返回内容进行 gzipping 压缩，并不缓存页面，因此 **GzipFilter** 并不需要 EHCache 配置文件支持。

关于 EHCache-Web 的详细说明请参靠软件包自带的 API 帮助文档和《EHCache Web User Guide》。

现在我们着手对上例中的查询页面进行缓存处理。由于该页面已经实现了 URL 重写，因此我们有两个选择：

1、 把伪 URL 地址用作页面缓存 Key
2、 把真实 URL 地址用作页面缓存 Key

这两种方式都可行，但从性能方面考虑第一种方式更优，因为采用第一种方式时需要把 EHCache-Web Filter 放置到 UrlRewrite Filter 之前，当页面已被缓存且未失效的情形下就不必进入 UrlRewrite Filter；而如果采用第二种方式，则需要把 EHCache-Web Filter 放置到 UrlRewrite Filter 之后，因此每次处理请求都会先进入 UrlRewrite Filter。现以第一种方式为例阐述具体的操作过程：

1、 加入 EHCache-Web 程序包

把 ehcache-web-2.0.4.jar 和 ehcache-core-2.4.8.jar 放入应用程序的 /WEB-INF/lib 目录。

注意：ehcache-web-2.0.4 依赖 ehcache-core-2.4.x，如果 ehcache-core 的版本太高（如：ehcache-core-2.6.x）可能会引发一些异常。

2、 修改 /WEB-INF/web.xml：加入 SimplePageCachingFilter 和 ShutdownListener

```
<!-- EHCache-Web 页面缓存过滤器 -->
<filter>
    <filter-name>PageCacheFilter</filter-name>
        <filter-class>net.sf.ehcache.constructs.web.filter.SimplePageCachingFilter</filter-class>
    <init-param>
        <!--
          EHCache 的 Cache Name，需要在 ehcache.xml 中配置
          （可选，默认：SimplePageCachingFilter）
        -->
        <param-name>cacheName</param-name>
        <param-value>MyWebCache</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>PageCacheFilter</filter-name>
    <!-- 拦截 URL 的 Pattern -->
    <url-pattern>/page/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
    <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

```
<!-- EHCache 清理器 -->
<listener>
    <listener-class>net.sf.ehcache.constructs.web.ShutdownListener</listener-class>
</listener>
```

上面的配置中，SimplePageCachingFilter 拦截所有以“**/page/**”开头的 URL。注意：**SimplePageCachingFilter 的定义必须放在 UrlRewriteFilter 的前面**（为什么呢？）。SimplePageCachingFilter 的其他初始化选项请参考相关文档。ShutdownListener 监听应用程序生命周期事件，用于在应用程序结束时关闭所有 EHCache Manager。ShutdownListener 应该放置于 JessMA 的启动监听器 AppListener 的前面。

3、 创建 EHCache 配置文件：${CLASS}/ehcache.xml

（参考 MyJessMA 工程源码：src/ehcache.xml）

配置文件的详细配置选项请参考：《EHCache 配置文档》。

通过上述三步操作，我们已经为查询页面加入了缓存功能。重启服务器后可以验证页面

缓存效果：在 300 秒内，如果每次查询的间隔不超过 60 秒，则相同查询条件下的查询结果不会变化，即使期间数据库中的实际数据已被更改。

■ *后续思考：*

1、 本例中 EHCache-Web 的 SimplePageCachingFilter 为何要配置在 UrlRewriteFilter 前面？如果把真实 URL 地址用作页面缓存 Key 又该如何配置？

答：这个问题留给大家思考 ^_^

2、 如果页面查询结果要根据用户标识（如：Session）为不同用户返回不同的查询结果，如何实现？

答：这个问题分三步来回答：

- 这种页面是否有缓存的必要？通常情况下，应用程序应该缓存的是公共页面，这样当访问的用户与访问次数越多就越能体现缓存的作用与价值。如果为不同用户缓存各自的页面，首先，会导致缓存页面的数量急剧增大；更重要的是，缓存的作用却急剧降低，因为一个用户在短时间内多次作同一个查询的几率是非常低的（不可能在一分钟内作一百次同样的查询吧）。因此这样做非但不能提升系统性能反而更可能使得系统缓存了太多无用数据而降低性能。
- 还有一种情况：页面查询结果为不同的用户组（而不是单个用户）返回不同的页面。在这种情况下，如果用户组数目少，而组内用户数目多并且访问频繁，那么真的有需要对每组用户的查询结果进行缓存了，如何实现呢？
- 前面说过，EHCache-Web 以 URL 的请求路径作为 Key 来缓存页面，其中 Key 通过 Filter 的 **String calculateKey(HttpServletRequest)** 方法生成。因此，只需改写这个方法，为 Key 附加上用户组信息，不同的用户组生成不同的 Key 即可实现。

3、 应用程序还可以通过哪些简单机制提供系统性能？

答：通常情况下，可以从以下几个方面着手：

- **客户端：**尽量利用浏览器的缓存功能，在 Response 的 HTTP 头以及页面的 Head 中指示出缓存相关的信息。
- **Web Server：**把 HTML、JS 以及图片等静态内容交由 Apache / Nginx 等 Web Server 处理，App Server 只处理动态内容。
- **App Server：**综合利用页面缓存、对象缓存和连接池等多种缓存技术，尽量减少耗时操作。
- **程序设计：**把性能目标作为重要的质量指标贯彻到系统规划、架构设计、模块设计以及代码编写等所有开发环节。
- **运营部署：**利用负载均衡、集群、服务器调优和网络调优等方法提高生成系统的综合服务能力。

9 应用篇（八）—— 整合模板引擎

FreeMarker 和 Velocity 是常用的 JavaEE 模板引擎，它们都提供一个 Servlet（freemarker.ext.servlet.**FreemarkerServlet** 和 org.apache.velocity.servlet.**VelocityServlet**）用于与 MVC 框架整合，整合的方式基本一致。因此，本章只以 FreeMarker 为例阐述如何在 JessMA 中整合模板引擎。

FreeMarker 是一款模板引擎：即一种基于模板、用来生成输出文本（任何来自于 HTML 格式的文本用来自动生成源代码）的通用工具。它是为 Java 程序员提供的一个开发包，或者说是一个类库。它不是面向最终用户的，而是为程序员提供的一款可以嵌入他们所开发产品的应用程序。

FreeMarker 实际上是被设计用来生成 HTML 页面，尤其是通过实现了基于 MVC（Model View Controller，模型-视图-控制器）模式的 Java Servlet 应用程序。使用 MVC 模式的动态页面的设计构思使得你可以将前端设计师（编写 HTML 页面的人员）从程序员中分离出来。那么，所有人各司其职，发挥其最擅长的一面。网页设计师可以改写页面的显示效果而不受程序员编译代码的影响，因为应用程序的逻辑（这里是 Java 程序）和页面设计（这里是 FreeMarker 模板）已经被分开了。页面模板代码不会受到复杂程序代码的影响。这种分离的思想即便对一个程序员和页面设计师是同一个人的项目来说也都是非常有用的，因为分离使得代码保持简洁而且易于维护。

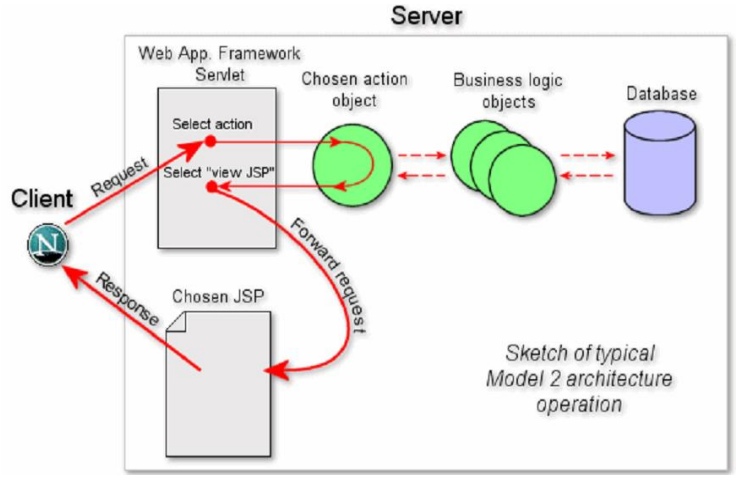
尽管 FreeMarker 也拥有一些编程能力，但是它却不像 PHP 那样是的一种全面的编程语言。反而，Java 程序准备的数据来进行显示（比如 SQL 数据库查询），FreeMarker 仅仅是使用模板生成文本页面来呈现已经准备好的数据而已。

FreeMarker 不是 Web 开发的应用程序框架。它是一个适用于 Web 应用程序框架中的组件，但是 FreeMarker 引擎本身并不知道 HTTP 协议或 Java Servlet 的存在。它仅仅来生成文本内容。既然是这样，它也非常适用于非 Web 应用程序的开发环境。只是要注意的是，我们使用 FreeMarker 作为视图层的组件，是为了给诸如 Struts 这样的 Model 2 应用框架提供现成的解决方案。

9.1 FreemarkerServlet 简介

Freemarker 为了能方便地与各种 Model 2 应用框架整合，它提供了一个 Servlet（freemarker.ext.servlet.**FreemarkerServlet**）来完成这项工作。这个 Servlet 处理所有的 FTL

请求（URL 通常是以 ***.ftl*** 结尾），它负责加载 FTL 模板文件，并把 FTL 模板与模型数据合并，生成最终的输出视图。Model 2 应用框架模型如下图所示：



整合 Freemarker 时，可用 FreemarkerServlet 取代上图中的 JSP 页面，把视图请求定向到 FreemarkerServlet，再由 FreemarkerServlet 生成输出视图。

9.2 整合示例

本例用 Freemarker 创建一个“用户列表”页面，执行效果如下图所示：

查询用户

ID	姓 名	年 龄	性 别	工作年限	兴趣爱好
94	小甜甜	33	女	5-10 年	游泳 下棋 看书
93	QiQi	0	男	3 年以下	游泳 下棋 看书
91	李刚	55	男	5-10 年	下棋 打麻将
89	Kingfisher	0	男	5-10 年	打球 下棋 打麻将
48	马大哈	36	女	3-5 年	游泳 打球 打麻将
13	伤神小怪兽	33	男	10 年以上	游泳 打球 看书
12	Jess	27	女	5-10 年	游泳

为简单起见，该页面只列出所有用户的列表，并不提供任何过滤功能。

1、 加入 Freemarker 程序包

把 freemarker-2.3.21.jar 放入应用程序的 /WEB-INF/lib 目录。

2、 修改 /WEB-INF/web.xml：加入 FreemarkerServlet

```
<!-- Freemarker 模板处理器 -->
<servlet>
    <!-- 模板处理器 Servlet -->
    <servlet-name>freemarker</servlet-name>
```

```
        <servlet-class>freemarker.ext.servlet.FreemarkerServlet</servlet-class>
        <!-- <servlet-class>global.MyFreemarkerServlet</servlet-class> -->

        <!-- ########## FreemarkerServlet 配置参数 ########## -->
        <init-param>
            <!-- 模板文件路径 -->
            <param-name>TemplatePath</param-name>
            <param-value>/</param-value>
        </init-param>
        <init-param>
            <!-- 是否禁止浏览器缓存 -->
            <param-name>NoCache</param-name>
            <param-value>true</param-value>
        </init-param>
        <init-param>
            <!--  页面编码格式 -->
            <param-name>ContentType</param-name>
            <param-value>text/html; charset=UTF-8</param-value>
        </init-param>
        <!-- ################################################ -->

        <!-- ####### Freemarker Configuration 配置参数 ###### -->
        <init-param>
            <!-- 模板更新延时（ 0 只对开发使用! 否则使用大一点的值） -->
            <param-name>template_update_delay</param-name>
            <param-value>0</param-value>
        </init-param>
        <init-param>
            <!-- 模板文件的编码方式. -->
            <param-name>default_encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
        <init-param>
            <!-- 是否去除页面中多余的空格符 -->
            <param-name>whitespace_stripping</param-name>
            <param-value>true</param-value>
        </init-param>
        <!-- ################################################ -->

        <!-- 应用程序启动时加载 FreemarkerServlet -->
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
      <!-- 处理所有以 ".ftl" 结尾的 URL 请求 -->
```

```
        <servlet-name>freemarker</servlet-name>
        <url-pattern>*.ftl</url-pattern>
    </servlet-mapping>
```

上面的配置中，FreemarkerServlet 处理所有以“**.ftl**”结尾的 URL 请求。它自身有三个配置参数：

- **TemplatePath**：FTL 模板文件的根目录
- **NoCache**：是否禁止客户端缓存
- **ContentType**：默认页面编码格式

除此以外，还可以配置 freemarker.template.**Configuration** 的相关参数：

ANGLE_BRACKET_TAG_SYNTAX	1
AUTO_DETECT_TAG_SYNTAX	0
AUTO_IMPORT_KEY	"auto_import"
AUTO_INCLUDE_KEY	"auto_include"
CACHE_STORAGE_KEY	"cache_storage"
DEFAULT_ENCODING_KEY	"default_encoding"
DEFAULT_INCOMPATIBLE_ENHANCEMENTS	"2.3.0"
INCOMPATIBLE_ENHANCEMENTS	"incompatible_enhancements"
LOCALIZED_LOOKUP_KEY	"localized_lookup"
SQUARE_BRACKET_TAG_SYNTAX	2
STRICT_SYNTAX_KEY	"strict_syntax"
TAG_SYNTAX_KEY	"tag_syntax"
TEMPLATE_UPDATE_DELAY_KEY	"template_update_delay"
WHITESPACE_STRIPPING_KEY	"whitespace_stripping"

ARITHMETIC_ENGINE_KEY	"arithmetic_engine"
AUTO_FLUSH_KEY	"auto_flush"
BOOLEAN_FORMAT_KEY	"boolean_format"
CLASSIC_COMPATIBLE_KEY	"classic_compatible"
DATE_FORMAT_KEY	"date_format"
DATETIME_FORMAT_KEY	"datetime_format"
LOCALE_KEY	"locale"
NEW_BUILTIN_CLASS_RESOLVER_KEY	"new_builtin_class_resolver"
NUMBER_FORMAT_KEY	"number_format"
OBJECT_WRAPPER_KEY	"object_wrapper"
OUTPUT_ENCODING_KEY	"output_encoding"
STRICT_BEAN_MODELS	"strict_bean_models"
TEMPLATE_EXCEPTION_HANDLER_KEY	"template_exception_handler"
TIME_FORMAT_KEY	"time_format"
TIME_ZONE_KEY	"time_zone"
URL_ESCAPING_CHARSET_KEY	"url_escaping_charset"

FreemarkerServlet 还提供了很多钩子方法，可根据自己的需要创建 FreemarkerServlet 子类，修改 FreemarkerServlet 的行为或定制一些功能，具体使用方法说明请参考 API 文档。例如，如果需要确保在直接访问“**.ftl**”地址（不经过 Action 处理）的情形下也能正确设置“**__base**”，那么，可以实现自己的 FreemarkerServlet，在处理请求时检测“**__base**”设置：

```
package global;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.jessma.mvc.Action.BaseType;
import org.jessma.mvc.Action.Constant;
import org.jessma.util.http.HttpHelper;
import freemarker.ext.servlet.FreemarkerServlet;

@SuppressWarnings("serial")
public class MyFreemarkerServlet extends FreemarkerServlet
{
    @Override
```

```
    protected boolean preprocessRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException
    {
            setBasePath(request);
            return false;
    }

    private void setBasePath(HttpServletRequest request)
    {
            BaseType baseType = (BaseType)HttpHelper
                                    .getApplicationAttribute(Constant.APP_ATTR_BASE_TYPE);
            if(baseType == BaseType.AUTO)
            {
                    String __base = (String)request.getAttribute(Constant.REQ_ATTR_BASE_PATH);
                    if(__base == null)
                    {
                            __base = HttpHelper.getRequestBasePath(request);
                            request.setAttribute(Constant.REQ_ATTR_BASE_PATH, __base);
                    }
            }
    }
}
```

3、 修改 /jsp/index.jsp：加入测试链接

```
    <li><a href="test/ftl/queryInterest.action">测试页面静态化</a></li>
```

4、 创建 Action：QueryInterest

（直接使用前例的 **action.test.QueryInterest**，不用再额外创建）

5、 创建测试 FTL 模板：/ftl/test/test_query.ftl

（参考 MyJessMA 工程源码：WebRoot/ftl/test/test_query.ftl）

6、 修改 MVC 配置文件，加入 Action：test/ftl/queryInterest

```
    <actions path="test/ftl">
            <action name="queryInterest" class="action.test.QueryInterest">
                    <result>/ftl/test/static/test_query.ftl</result>
            </action>
    </actions>
```

10 应用篇（九）—— 多入口 Action

前文所用到的 Action 均只有一个入口方法“***String execute()***”。Action 通过 ***execute()*** 方法处理请求。因此 Action 的名称通常是“动名词”（*如：CreateUser*、*DeleteUser*、*QueryUser* …）。

所谓多入口 Action 是指，Action 的入口方法有多个，不一定是“***execute()***”。多入口 Action 有以下好处：

- ✓ **用“名词”来命名 Action：**例如一个名为“*UserAction*”的 Action，它内部可提供 ***create()***、***delete()*** 和 ***query()*** 等入口方法，处理 User 相关的所有请求。
- ✓ **更好的代码组织结构：**相关的操作都集中到同一个 Action 中有利于数据、方法重用和代码维护，也有利于配置文件的组织与管理。

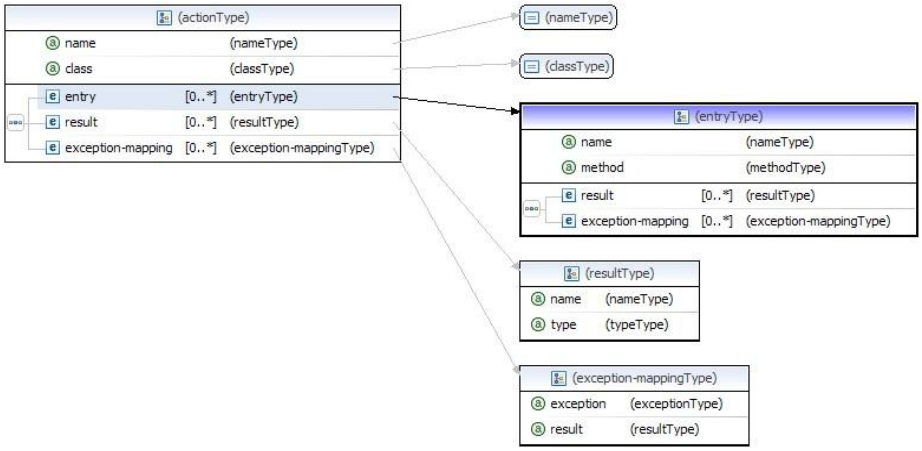
10.1 Action 配置

JessMA 利用以下两个配置项来支持多入口 Action：

- **Action（<actions>/<action>）：**增加 Action Entry 配置项
- **Action Filter（<global>/<action-filters>/<filter>）：**拦截条件细化到 Action Entry

10.1.1 Action Entry

<action/> 中可以定义多个 <entry/> 元素，在< entry/> 中可以定义多个 <result/> 和 <exception-mapping/> 元素，其结构定义如下图所示：



17. Entry 名称：entry.name（可选）

<entry/> 的 name 属性用于设置 Action Entry 的名称，例如下面两个 Entry 配置：

```
示例：
    <actions>
        <action name="myaction">
            < !--默认入口：myaction.action -->
            <entry />
            < !--入口：myaction!myentry.action -->
            <entry name="myentry" />
        </action>
    </actions>
```

✓ **第一个 Action Entry：**
访问路径 --> http://${My Host}/${My App}/**myaction.action**

✓ **第二个 Action Entry：**
访问路径 --> http://${My Host}/${My App}/**myaction!myentry.action**

*** **关于 entry.name 属性的几点特别说明：**

1、 普通 Action Entry 的访问地址：

{action.name}!{entry.name}.{action_suffix}

2、 **entry.name** 为空的 Action Entry 称为“默认 Action Entry”，它的访问地址：

{action.name}.{action_suffix}

3、 如果 Action 没有定义默认 Action Entry，JessMA 会为 Action 自动创建默认一个，该默认 Action Entry 的 **entry.method** 属性值为“***execute***”，并且没有 <result/> 和 <exception-mapping> 子元素，等同于：

<entry method=”execute” />

18. Entry 入口方法：entry.method（可选）

<entry/> 的 method 属性用于声明 Action Entry 对应的 Action 实现类的入口方法，例如下面几个 Entry 配置：

```
示例：
    <actions>
        <action name="myaction" class="com.mypkg.SomeAction">
            < !--默认入口：myaction.action -->
            <entry method= "method_1" />
```

```
            < !--入口：myaction!myentry_2.action -->
            <entry name="myentry_2" />
            < !--入口：myaction!myentry_3.action -->
            <entry name="myentry_3" method= "method_x" />
        </action>
    </actions>
```

- ✓ **第一个 Action Entry：**
  访问路径 --> http://${My Host}/${My App}/**myaction.action**
  入口方法 --> ***com.mykpg.SomeAction#method_1()***
- ✓ **第二个 Action Entry：**
  访问路径 --> http://${My Host}/${My App}/**myaction!myentry_2.action**
  入口方法 --> ***com.mykpg.SomeAction#myentry_2()***
- ✓ **第三个 Action Entry：**
  访问路径 --> http://${My Host}/${My App}/**myaction!myentry_3.action**
  入口方法 --> ***com.mykpg.SomeAction#method_x()***

*** **关于 entry.method 属性的几点特别说明：**

1、 如果 **entry.method** 不为空，**entry.name** 为空则，则该 Action Entry 是默认 Action Entry，它的入口方法名是 **entry.method**（*上例的第一个 Action Entry*）。
2、 如果 **entry.method** 为空，**entry.name** 不为空则，则 Action Entry 的入口方法名与 **entry.name** 一致（*上例的第二个 Action Entry*）。
3、 如果 **entry.method** 为空，**entry.name** 也为空则，则该 Action Entry 是默认 Action Entry，它的入口方法名是“***execute***”。

19. Entry Result：result（可选，参考：《Action Result》）

Action Entry 查找 Result 的顺序：

Action Entry Results
→ ***Action Results***
→ ***Global Results***

20. Entry Exception：exception-mapping（可选，参考：《Action Exception》）

Action Entry 查找 Exception Mapping 的顺序：

Action Entry Exception Mappings
→ ***Action Exception Mappings***
→ ***Global Exception Mappings***

10.1.2 Action Filter

为了支持多入口 Action，需要把 Action Filter 的拦截规则细化到入口方法级别，因此在 <filter/> 中加入了“methods”属性用来匹配 Action 的入口方法，其结构定义如下图所示：

（action filter 的配置请参考：《Action 拦截器》）

10.2 Action 使用

10.2.1 入口方法

多入口 Action 拥有多个入口方法，能处理多种请求。入口方法的签名格式：

public String entry_method() *throws Exception*

和“***execute()***”方法一样，**entry_method** 必须是返回值类型为 **String** 的无参数公共方法，可选择声明抛出或不抛出任何异常。它的返回值作为 Action Result。

10.2.2 @FormBean

由于不同入口方法需要装配的 Form Bean 不一样，或者有些需要装配，有些不需要装配，因此，@FormBean 可以在 Action 类定义和 Action 入口方法中声明，@FormBean 声明的规则如下：

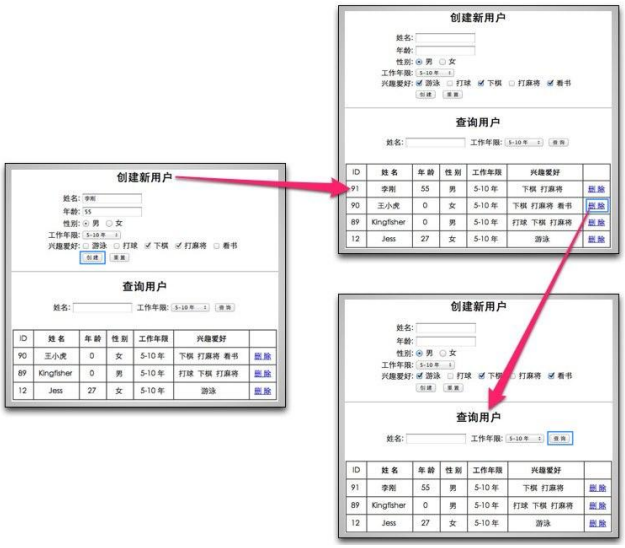
- **Action 类中声明的 @FormBean：**
  Action 所有入口方法的默认 Form Bean 注入规则，如果入口方法没有声明自己的 @FormBean 则使用这个使用该注入规则。

- **Action 入口方法中声明的 @FormBean：**
  入口方法特定的 Form Bean 注入规则，会覆盖默认 Form Bean 注入规则。

注意： @FormBean 能注入私有成员变量（即：没有公共 setter 方法的成员变量）。@FormBean 首先检查待注入的成员是否有公共 setter 方法，如果有则调用它的 setter 方法执行注入；否则，直接设置成员变量的值。

10.3 应用示例

本例修改 7.5.3 小节中的示例，把 **createUser2、deleteUser2、queryUser2** 这三个 Action 修改为一个多入口 Action。示例界面与以前一样：



1、 修改 /jsp/index.jsp：加入测试链接

```
<li> <a href="test/entry/user.action">测试多入口 Action</a></li>
```

2、 修改 MVC 配置文件，加入 Action：user

```
<actions path="/test/entry">
    <action name="user" class="action.test.UserAction">
        <entry name="create">
            <result type="chain">./user!query</result>
        </entry>
        <entry name="delete">
            <result type="chain">./user!query</result>
        </entry>
        <entry name="query" method="findUsers">
            <result>/jsp/test/entry/test_user.jsp?fromQueryAction=true</result>
        </entry>
        <result>/jsp/test/entry/test_user.jsp</result>
    </action>
</actions>
```

上述配置定义了一个名称为“user”的 Action，其实现类为 **action.test.UserAction**。并定义了 3 个 Action Entry（实际上有 4 个），这些 Action Entry 的属性如下：

名 称	访问地址	入口方法	Action Result
create	user!create.action	**create**	**[success|chain]：**./user!query
delete	user!delete.action	**delete**	**[success|chain]：**./user!query
query	user!query.action	**findUsers**	**[success]：**/jsp/test/entry/test_user.jsp?from...=true
	user.action	***execute***	***[success]：****/jsp/test/entry/test_user.jsp*

3、 创建 Action：action.test.UserAction

（参考 MyJessMA 工程源码：src/action.test.UserAction）

UserAction 定义了 3 个入口方法，这些入口方法的代码与 7.5.3 小节中定义的三个 Action（MybatisCreateUser、MybatisCreateUser、MybatisCreateUser）的“*execute()*”方法完全相同。

“*create()*”方法声明了 @FormBean，它使用自己的 Form Bean 注入规则，而其它方法没有声明 @FormBean，因此它们使用 UserAction 的 @FormBean 定义的默认注入规则。

注意：虽然 UserAction 没有改写 execute() 方法，execute() 方法也没有任何参数需要被注入，但程序还是会为 execute() 方法检查注入内容。

4、 创建测试页面：/jsp/test/entry/test_user.jsp

测试页面的代码与 7.5.3 小节中的测试页面代码完全一样，只需把几个 Action 链接分别改为 ***test/entry/user!create.action（user!query.action 、user!delete.action）***。（代码请参考：《/jsp/test/dao/test_user_1.jsp》）

5、 创建 Action 拦截器：UserActionFilter

到上述第 4 步为止，示例的制作已经完成。但为了展示 Action 拦截器的拦截规则能细化到入口方法级别，特此增加了本步骤。本例中假设我们希望拦截 UserAction 的所有“非查”询入口方法（不以“*find*”或“*execute*”开头的方法），只需执行下面两个步骤：

a) 创建 Action 拦截器类：global.UserActionFilter

（参考 MyJessMA 工程源码：src/global.UserActionFilter）

b) 配置 Action 拦截器

在 MVC 主配置文件中加入拦截器定义：

```
<filter pattern="action.test.UserAction" methods="(?!find|execute)\w*" class="global.UserActionFilter"/>
```

当 UserAction 的入口方法不以“*find*”或“*execute*”开头，则会打印出被拦截的方法名。

11 应用篇（十）—— 新 DAO 访问接口

JessMA 的 DAO 框架以前一直使用 **FacadeProxy** 的 **getAutoCommitProxy(...)** 或 **getManualCommitProxy(...)** 获取 DAO 对象（参考：《DAO Facade Proxy》）。通过这两个方法获得的 DAO 对象绑定了下列 4 个属性：

- ***DAO Class*** ：**被代理的 DAO 类**
- ***Session Manager*** ：**连接管理器**
- ***Auto Commit*** ：**是否强制启动事务**
- ***Transaction Level*** ：**事务隔离级别**

当 DAO 对象要执行多个数据库访问方法时，使用起来可能不够灵活。例如：当 DAO 对象执行数据库更新方法时希望强制启动事务，但当它执行数据库查询方法时，出于性能等方面的原因考虑不希望强制启动事务。这时只有两个选择：

1、创建一个强制启动事务的 DAO 对象
（能保证安全性，但会影响查询性能）
2、创建两个 DAO 对象，一个用于执行更新操作，一个用于执行查询操作
（使用起来不方便，并且必须知道哪些方法会更新数据库，哪些方法只是单纯查询）

其实，是**“否强制使用事务”**和**“事务隔离级别”**这两个属性是和具体的数据库访问方法相关的，数据库访问方法知道自身是否需要事务支持以及是否要修改事务隔离级别。因此， JessMA 提供了一组新的 DAO 访问接口：

- **FacadeProxy.create(...)：创建 DAO 对象（绑定 *DAO Class* 和 *Session Manager*）**
- **@Transaction：DAO 事务属性注解（绑定 *Auto Commit* 和 *Transaction Level*）**

11.1 FacadeProxy.create(...) 和 @Transaction

FacadeProxy 提供以下两个方法创建只绑定 *DAO Class* 和 *Session Manager* 的 DAO 对象。

- **F create(Class<F> *daoClass*)**
- **F create(Class<F> *daoClass*, M *mgr*)**

create() 方法中参数 *daoClass* 和 *mgr* 的含义与 *getXxxCommitProxy()* 方法相应参数一样，但是，用 *create()* 方法创建的 DAO 对象不绑定 *Auto Commit* 和 *Transaction Level* 这两个事务属性，它会在执行具体数据库访问方法时查找方法的 @Transaction 注解来获得方法的事务属性。

注意：create(...) 的第一个重载方法要求： daoClass 必须实现一个访问级别为 protected 以上的无参数构造函数；第二个重载方法要求：daoClass 必须实现一个访问级别为 protected

以上的接收一个 SessionMgr 类型参数的构造函数。

@Transaction 有以下两个注解参数：

- **boolean value()** ：是否强制启动事务（默认：**true**）
- **TransIsoLevel level()** ：事务隔离级别（默认：**TransIsoLevel.DEFAULT**）

*** 关于 **@Transaction** 的几点特别说明：

1、声明在 DAO 类中的 @Transaction 定义该 DAO 类的默认事务属性，任何没有声明 @Transaction 的 DAO 方法都使用类中声明的默认事务属性。
2、声明在 DAO 方法中的 @Transaction 定义该 DAO 方法的事务属性。
3、如果 DAO 类及其方法都没有声明 @Transaction，则调用该方法时使用 @Transaction 的默认属性（**value = true, level = TransIsoLevel.DEFAULT**）。
4、对于嵌套调用 DAO 方法的情形（DAO 方法内部调用该 DAO 或其他 DAO 对象的其它方法），整个方法调用期间，事务属性由最外层的 DAO 方法指定。因此，外层方法的事务属性安全级别应该设置为不低于它调用的任何一个内层方法。（关于 DAO 方法嵌套调用的具体阐述请参考 10.6 节）
5、由上述第 4 点得出，@Transaction 其实只对 **public** 方法有意义，其他非 **public** 方法可以不声明 @Transaction。
6、***注意：***@Transaction 的 **value** 参数的含义与“*Auto Commit*”的含义是相反的，如果要强制启动事务@Transaction 的 **value** 参数要设置为 **true**，因此也就不是“*Auto Commit*”，而是“*Manual Commit*”。

11.2 应用示例

本例把 10.3 小节示例的 DAO 对象获取方式改为 FacadeProxy.create(...) + @Transaction(...) 的形式，需要做以下工作：

1、修改 dao.mybatis.UserDao：加入 @Transaction 注解

```
public class UserDao extends MyBatisBaseDao
{
	@Transaction(level=TransIsoLevel.REPEATABLE_READ)
	public boolean createUser(User user)
	{
		// 方法内容
		// （略）.....
	}

	public boolean deleteUser(int id)
	{
		// 方法内容
		// （略）.....
```

```
    }

    @Transaction(false)
    public List<User> findUsers(String name, int experience)
    {
        // 方法内容
        // （略）.....
    }

    @Transaction(false)
    public List<User> queryInterest(int gender, int experience)
    {
        // 方法内容
        // （略）.....
    }
}
```

上述代码中，每个方法的事务属性如下：

名 称	强制启动事务	Auto Commit	事务隔离级别
（默认）	*true*	*false*	*TransIsoLevel.DEFAULT*
createUser	true	false	TransIsoLevel.REPEATABLE_READ
deleteUser	true	false	TransIsoLevel.DEFAULT
findUsers	false	true	（忽略）
queryInterest	false	true	（忽略）

2、 修改 Action：UserAction

把 UserAction 中所有获取 DAO 对象的方法（*FacadeProxy.getAutoCommitProxy(...)* 和 *FacadeProxy.getManualCommitProxy(...)*）改为 ***FacadeProxy.create(...)***。

（参考 MyJessMA 工程源码：src/action.test.UserAction2）

11.3 DaoInjectFilter 和 @DaoBean / @DaoBeans

为了进一步简化 DAO 对象的获取操作，JessMA 为 Action 提供了一种基于 org.jessma.app.**DaoInjectFilter** Action 拦截器和 **@DaoBean / @DaoBeans** 注解的 DAO 注入方法。在执行 Action 入口方法之前，创建 DAO 对象并注入到 Action 的成员属性中。从而避免显式调用 FacadeProxy.create(...) 方法。从执行效果上看，有点类似于 **@FormBean**，只是他们注入的内容不同，**@DaoBean / @DaoBeans** 注入 DAO 对象，而**@FormBean** 注入请求参数。

应用程序如果要开启 DAO 注入功能需要做以下工作：

1） 在 MVC 主配置文件中配置 DaoInjectFilter 拦截器
2） 把需要注入的 DAO 变量声明为 Action 的实例属性
3） 在 Action 入口方法或类中声明 @DaoBean 或 @DaoBeans

DaoInjectFilter 拦截 Action 的入口方法，它会检查与入口方法相关的 @DaoBean / @DaoBeans 声明，并根据注解参数信息把 Dao 对象注入到相应的 Action 属性成员中。

@DaoBean 有以下三个注解参数：

- **String value()** ：**待注入的成员属性名**
  （默认：""，Action 中第一个 Dao 类型的成员属性）
- **Class daoClass()** ：**注入的 DAO 对象类型**
  （默认：AbstractFacade.class，根据成员属性声明的类型注入）
- **String mgrName()**：**app-config.xml 中配置的 Session Manager 名称**
  （默认：""，使用 daoClass 的默认构造函数创建 DAO 对象）

@DaoBeans 只有一个注解参数：

- **DaoBean[] value()**：**@DaoBean 数组，用于注入多个 DAO 成员属性**
  （默认：{@DaoBean}，Action 中所有 Dao 类型的成员属性）

***** 关于 @DaoBean / @DaoBeans 的几点特别说明：**

1、 声明在 Action 类中的 @DaoBean / @DaoBeans 定义该 Action 类的默认注入规则，任何没有声明 @DaoBean / @DaoBeans 的 Action 入口方法都使用类中声明的默认注入规则。
2、 声明在 Action 入口方法中的 @DaoBean / @DaoBeans 定义该入口方法的注入规则。只要入口方法声明了 @DaoBean / @DaoBeans 的其中之一，就会忽略 Action 类中的 @DaoBean / @DaoBeans。
3、 如果 Action 类中或入口方法中同时声明了 @DaoBean 和 @DaoBeans，应用程序会同时处理它们的并集。
4、 如果 @DaoBean 的 value() 参数为空字符串（默认值），应用程序会注入 Action 的第一个 DAO 类型的成员属性，如果找不到则递归搜索所有父类。
5、 如果 @DaoBeans 中存在一个 value() 参数为空字符串的 @DaoBean 元素，应用程序会注入 Action 及其所有父类的所有 DAO 类型的成员属性。由于默认的 @DaoBeans 包含了这样一个 @DaoBean 元素，因此默认的 @DaoBeans 会注入 Action 及其所有父类的所有 DAO 类型的成员属性。
6、 应用程序会自动过滤 value() 参数重复的 @DaoBean 声明，采用它所找到的第一个 @DaoBean。
7、 如果 @DaoBean 的 daoClass() 参数的值为 **AbstractFacade.class** 或非 **public** 类，应用程序会尝试使用成员属性的声明类型来创建 DAO 对象。由于 @DaoBean 的 daoClass() 参数的默认值为 **AbstractFacade.class**，因此默认的 @DaoBean 会根据成员属性的声明类型创建要注入的 DAO 对象。

8、 ***注意：*** 如果 @DaoBean 的 mgrName() 参数为空字符串，效果等同于调用 ***FacadeProxy.create(Class daoClass)*** 来创建 DAO 对象。这种情况下，daoClass() 必须提供一个没有任何参数的构造函数；如果 @DaoBean 的 mgrName() 参数不为空字符串，效果等同于调用 ***FacadeProxy.create(Class daoClass, M mg)*** 来创建 DAO 对象。这种情况下，daoClass() 必须提供一个接收 ***SessionMgr*** 类型参数的构造函数。

9、 如果 @DaoBean 注入的属性有 ***public setter*** 方法。则会调用其 ***setter*** 方法执行注入，否则直接设置成员变量的值。

注意：不要混淆 @Transaction 和 @DaoBean / @DaoBeans。@Transaction 声明在 DAO 类及其数据库访问方法中，定义其事务属性；@DaoBean / @DaoBeans 声明在 Actiong 及其入口方法中，用来注入 DAO 对象，目的是消除对 FacadeProxy.create(...) 方法的显式调用。

11.4 应用示例

本例把 11.2 小节示例的 DAO 对象获取方式改为 @DaoBean / @DaoBeans 注入，取代对 *FacadeProxy.create(...)* 方法的显式调用：

1、 在 MVC 主配置文件中加入拦截器：DaoInjectFilter

```
<filter pattern=".*" class="org.jessma.app.DaoInjectFilter"/>
```

2、 修改 Action：UserAction

删除 UserAction 中所有 *FacadeProxy.create(...)*方法，把入口方法中的局部变量“*UserDao dao*”改为的成员属性，然后在类定义或入口方法中声明 @DaoBean / @DaoBeans：

```
（参考 MyJessMA 工程源码：src/action.test.UserAction3）
```

上述代码中，每个方法的注入内容如下：

名 称	注解参数	注解值	解析值
（默认）	***value()***	*""*	*"dao"*
	daoClass()	*AbstractFacade.class*	*UserDao.class*
	mgrName()	*""*	*null*
create	**value()**	""	"dao"
	daoClass()	AbstractFacade.class	UserDao.class
	mgrName()	""	null
delete	**value()**	"dao"	"dao"
	daoClass()	UserDao.class	UserDao.class
	mgrName()	"session-mgr-2"	"session-mgr-2"
findUsers	**value()**	""	"dao"
	daoClass()	AbstractFacade.class	UserDao.class
	mgrName()	""	null

execute	**value()**	""	"dao"
	daoClass()	AbstractFacade.class	UserDao.class
	mgrName()	""	null

注意：虽然 UserAction 没有改写 execute() 方法，execute() 方法也不需要访问数据库，但由于类中声明了默认 @DaoBeans，所以程序还是会为 execute() 方法检查注入内容。

3、 修改 DAO 类：UserDao

由于在声明 @DaoBean / @DaoBeans 时，一些地方使用了默认的 mgrName() 参数，另一些则指定了 mgrName() 参数，因此 UserDao 必须同时提供下面两个构造函数。

```
protected UserDao()
{
	super();
}

protected UserDao(MyBatisSessionMgr mgr)
{
	super(mgr);
}
```

11.5 自定义事务

JessMA 的事务是 DAO 层事务，也就是说当外部调用某个 DAO 方法时，该方法作为一个事务单元执行。这种事务的优点是避免了由于过多的非数据库操作而导致的事务延长，并适用于绝大多数使用场合。但在一些特殊情形下可能需要在 DAO 外部执行 Service 层事务（例如：事务需要调用多个 DAO 对象的多个方法），此时需要创建一个自定义事务（CustomTransaction），并调用 FacadeProxy 的 *executeCustomTransaction(...)* 方法来执行该自定义事务。

FacadeProxy 提供以下两个方法执行自定义事务。

- ✧ **void executeCustomTransaction(M *mgr*, CustomTransaction<M, S> *trans*)**
- ✧ **void executeCustomTransaction(M *mgr*, TransIsoLevel *level*, CustomTransaction<M, S> *trans*)**

executeCustomTransaction() 方法中参数 *mgr* 为 Session 管理器，指定事务的数据连接信息；参数 *level* 为事务隔离级别，默认使用应用程序的默认事务隔离级别执行事务；参数 *trans* 为自定义事务对象，它提供事务入口方法 *execute(SessionMgr)* 来指定事务逻辑。为了简化自定义事务的创建工作，应用程序可以基于以下几个 CustomTransaction 子接口来创建自定义事务：

- **JdbcTransaction** : JDBC 自定义事务接口
- **MyBatisTransaction** : MyBatis 自定义事务接口

- **HibernateTransaction** ：Hibernate 自定义事务接口

执行自定义事务时有以下注意事项：

1） ***FacadeProxy.executeCustomTransaction(...)*** 一般在 DAO 方法外部调用
2） ***FacadeProxy.executeCustomTransaction(...)*** 也可以在 DAO 方法内部调用，但如果调用它的 DAO 方法与它拥有共同的 **SessionMgr**，则事务属性由该 DAO 方法控制
3） ***CustomTransaction.execute(mgr)*** 事务入口方法作为一个事务单元
4） ***CustomTransaction.execute(mgr)*** 如果抛出异常则回滚事务
5） ***CustomTransaction.execute(mgr)*** 中创建的 DAO 对象必须使用同一个 **SessionMgr**
6） ***CustomTransaction.execute(mgr)*** 中可以用 **new** 方式直接创建的 DAO 对象，也可以用 **FacadeProxy** 的代理方法（**create() / getXxxCommitProxy()**）创建 DAO 对象，DAO 对象的事务属性均由 **FacadeProxy.executeCustomTransaction(...)** 指定

```
/* *** 执行自定义事务示例 ***** */

// Service 方法
public static void serviceMethod()
{
	// 获取 SessionMgr
	HibernateSessionMgr mgr = (HibernateSessionMgr)AppConfig.getSessionManager("mgr-1");
	// 执行自定义事务
	FacadeProxy.executeCustomTransaction(mgr, new HibernateTransaction(){
		// 实现自定义事务方法
		@Override
		public void execute(HibernateSessionMgr mgr) throws DAOException
		{
			// 创建 dao1 （也可以使用 FacadeProxy 创建 DAO 对象）
			MyDaoA dao1 = new MyDaoA(mgr);
			// 创建 dao2 （也可以使用 'new MyDaoB(mgr)' 创建 DAO 对象）
			MyDaoB dao2 = FacadeProxy.create(MyDaoB.class, mgr);

			// 执行 DAO 方法
			dao1.methodXxx();
			dao2.methodYyy();
			dao1.methodZzz();
			dao2.methodNnn();
		}});
}
```

11.6 嵌套调用 DAO 方法

在实际的应用环境下，往往会遇到 DAO 方法嵌套调用的情形，如：

1） DAO 方法内部调用本 DAO 的另一个方法
2） DAO 方法内部创建其它 DAO 对象并调用它的方法
3） DAO 方法内部用 FacadeProxy.executeCustomTransaction(…)
4） FacadeProxy.executeCustomTransaction(…) 创建 DAO 对象并调用它的方法

JessMA DAO 框架的事务是**与 SessionMgr 关联的平面型事务**（不存在嵌套事务）。也就是说，无论 DAO 方法有多少层嵌套调用，只要它们使用的是同一个 SessionMgr，则他们都在同一个事务单元中，并且事务属性由最外层的 DAO 方法指定，忽略所有内层方法的事务属性。参考下面的示例，TestNestedDao 的 test1()方法展示了前 3 种嵌套情形，test2()方法展示了最后 1 种嵌套情形：

```
class MyBatisDao extends MyBatisFacade
{
	protected MyBatisDao()
	{
		this(AppConfig.getSessionManager("mgr-1"));
	}

	protected MyBatisDao(MyBatisSessionMgr mgr)
	{
		super(mgr);
	}
}

class DaoOuter extends MyBatisDao
{
	@Transaction(value=true, level=TransIsoLevel.READ_COMMITTED)
	public void funcA()
	{
		// (1)
		funcB();
	}

	@Transaction(value=false)
	public void funcB()
	{
		// (2)
		DaoInner innerDao1 = new DaoInner();
		innerDao1.funcX();

		// (2)
		DaoInner innerDao2 = FacadeProxy.create(DaoInner.class);
		innerDao2.funcY();
	}
```

```
	@Transaction(value=true, level=TransIsoLevel.REPEATABLE_READ)
	public void funcC()
	{
		// (3)
		FacadeProxy.executeCustomTransaction(
			getManager(), TransIsoLevel.READ_COMMITTED, new MyBatisTransaction()
		{
			@Override
			public void execute(MyBatisSessionMgr mgr) throws DAOException
			{
				// ……
			}
		});
	}
}

class DaoInner extends MyBatisDao
{
	@Transaction(value=false)
	public void funcX()
	{
		// ……
	}

	@Transaction(value=true, level=TransIsoLevel.SERIALIZABLE)
	public void funcY()
	{
		// ……
	}
}

class TestNestedDao
{
	static void test1()
	{
		DaoOuter dao = FacadeProxy.create(DaoOuter.class);

		// funcA() 作为一个事务单元，事务属性由 funcA() 指定
		// {TRANS=ON, LEVEL=READ_COMMITTED}
		dao.funcA();

		// funcB() 作为一个事务单元，事务属性由 funcB() 指定
		// {TRANS=OFF}
```

```
            dao.funcB();

            // funcC() 作为一个事务单元，事务属性由 funcC() 指定
            // {TRANS=ON, LEVEL=REPEATABLE_READ}
            dao.funcC();
        }

        static void test2()
        {
            MyBatisSessionMgr mgr = FacadeProxy.create(DaoOuter.class).getManager();

            FacadeProxy.executeCustomTransaction(
                mgr, TransIsoLevel.READ_COMMITTED, new MyBatisTransaction()
            {
                // execute() 作为一个事务单元，事务属性由 execute() 指定
                // {TRANS=ON, LEVEL=READ_COMMITTED}
                @Override
                public void execute(MyBatisSessionMgr mgr) throws DAOException
                {
                    // (4)
                    DaoInner innerDao1 = new DaoInner();
                    innerDao1.funcX();

                    // (4)
                    DaoInner innerDao2 = FacadeProxy.create(DaoInner.class);
                    innerDao2.funcY();
                }
            });
        }
}
```

嵌套调用 DAO 方法的规则总结：

- 最外层的 DAO 对象必须由 FacadeProxy 创建
- 内层的 DAO 对象可以用 **new** 或 FacadeProxy 等任意方式创建
- 最外层的 DAO 方法作为一个事务单元，并且不会有嵌套事务
- 事务属性由最外层的 DAO 方法指定，忽略所有内层 DAO 方法的事务属性

12 应用篇（十一）—— 整合 Spring / Guice

JessMA 和 Spring / Guice 都是开放式框架，它们之间的整合非常简单，所有整合工作都在 JessMA 的 Spring / Guice 扩展插件（***jessma-ext-spring-x.x.x.jar / jessma-ext-guice-x.x.x.jar***）中实现。JessMA 整合 Guice 相对简单，因此，本节主要讨论 JessMA 和 Spring 的整合细节。

整合内容包括以下 3 部分：

- **Spring ApplicationContext 生命周期管理**
- **JessMA MVC 子框架与 Spring 整合**
- **JessMA DAO 子框架与 Spring 整合**

12.1 Spring ApplicationContext 生命周期管理

Spring ApplicationContext 需要在应用程序启动时加载，并在结束时关闭。通常 Web 应用程序会配置一个监听器（*org.springframework.web.context.**ContextLoaderListener***）来完成这个工作，在 ***jessma-ext-spring*** 中，这个配置不是必须的，当它检测到应用程序当前没有加载 Spring ApplicationContext 时会启动一个私有的 ***ContextLoaderListener***，并由插件自身管理 ***ContextLoaderListener*** 的生命周期。

ContextLoaderListener 默认的 Spring 配置文件为：***/WEB-INF/applicationContext.xml***，如果要指定为其他配置文件则需在 ***web.xml*** 中指定。如：通常情况下，大家更喜欢把 Spring 配置文件放在 ${CLASS_PATH}（***/WEB-INF/classes***）中，可在 ***web.xml*** 中加入下面配置：

```
<!-- Spring 配置文件：/WEB-INF/classes/applicationContext.xml -->
<context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>classpath:applicationContext.xml</param-value>
</context-param>
```

应用程序可以也在 ***web.xml*** 中显式配置一个 ***ContextLoaderListener***，如果显式配置了 ***ContextLoaderListener***，则由应用服务器管理这个 ***ContextLoaderListener*** 的生命周期。

```
<!-- Spring 监听器（可选，如果 web.xml 中没用配置 ContextLoaderListener，
则 SpringInjectFilter 会启动一个私有 ContextLoaderListener） -->
<listener>
   <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

12.2 JessMA MVC 子框架与 Spring 整合

整合目标：在 JessMA 的 Action 中能自动注入 Spring Bean。

jessma-ext-spring 为 Action 提供了一种基于 org.jessma.ext.spring.**SpringInjectFilter** Action 拦截器和 **@SpringBean / @SpringBeans** 注解的 Spring Bean 注入方法。在执行 Action 入口方法之前，把 Spring Bean 注入到 Action 的成员属性中。从执行效果上看，有点类似于 **@FormBean**，只是他们注入的内容不同，**@SpringBean / @SpringBeans** 注入 Spring Bean，而**@FormBean** 注入请求参数。

开启 Spring Bean 注入功能需要做以下工作：

1） 在 MVC 主配置文件中配置 SpringInjectFilter 拦截器
2） 把需要注入的 Spring Bean 声明为 Action 的实例属性
3） 在 Action 入口方法或类中声明 @SpringBean 或 @SpringBeans

SpringInjectFilter 拦截 Action 的入口方法，它会检查与入口方法相关的 @SpringBean / @SpringBeans 声明，并根据注解参数信息把 Spring Bean 注入到相应的 Action 属性成员中。

@SpringBean 有以下三个注解参数：

- **String value()** ：**待注入的成员属性名**
  （必须指定）
- **Class type()** ：**Bean 类型**
  （默认：Object.class，根据 Bean 名称注入）
- **String name()** ：**Bean 名称**
  （默认：""，与成员属性名一致）

@SpringBean 的注入规则：

1） **同时指定 name() 和 type()** ：按 Bean 名称和类型注入
2） **只指定 name()** ：按 Bean 名称注入
3） **只指定 type()** ：按 Bean 类型注入
4） **name() 和 type() 都不指定** ：按 Bean 名称注入，其中 Bean 名称与 value() 一致

@SpringBeans 只有一个注解参数：

- **SpringBean[] value()** ：**@SpringBean 数组，用于注入多个 Spring Bean**
  （默认：{}，不注入任何 Spring Bean）

注意：有时默认 @SpringBeans 非常有用，例如，在 Action 类中声明了一些默认的注入规则，但某个方法不想注入任何 Spring Bean，则可在方法中声明一个默认 @SpringBeans，覆盖类中的声明；同理，子类也可以声明一个默认 @SpringBeans 覆盖的声明来取消注入任何 Spring Bean。

*** 关于 **@SpringBean / @SpringBeans** 的几点特别说明：

1、 声明在 Action 类中的 @SpringBean / @SpringBeans 定义该 Action 类的默认注入规则，任何没有声明 @SpringBean / @SpringBeans 的 Action 入口方法都使用类中声明的默认注入规则。
2、 声明在 Action 入口方法中的 @SpringBean / @SpringBeans 定义该入口方法的注入规则。只要入口方法声明了 @SpringBean / @SpringBeans 的其中之一，就会忽略 Action 类中的 @SpringBean / @SpringBeans。
3、 如果 Action 类中或入口方法中同时声明了 @SpringBean 和 @SpringBeans，应用程序会同时处理它们的并集。
4、 应用程序会自动过滤 value() 参数重复的 @SpringBean 声明，采用它所找到的第一个 @SpringBean。
5、 如果 @SpringBean 注入的属性有 ***public*** ***setter*** 方法。则会调用其 ***setter*** 方法执行注入，否则直接设置成员变量的值。

*** 现在总结一下到目前为止已经接触到的 JessMA 的 4 种注解：

名 称	可继承	声明位置	作 用
@FormBean	是	Action 类和入口方法	注入请求参数
@DaoBean/@DaoBeans	是	Action 类和入口方法	注入 DAO 对象
@SpringBean/@SpringBeans	是	Action 类和入口方法	注入 Spring Bean
@Transaction	是	DAO 类和方法	声明事务属性

12.3 JessMA DAO 子框架与 Spring 整合

JessMA 和 Spring 都提供了自己的 DAO 子框架，因此整合后可以根据实际需要，选用 JessMA 或 Spring 的 DAO 子框架访问数据库：

1、Spring DAO 子框架
（功能齐全，能支持 JTA 分布式事务）
2、JessMA DAO 子框架
（简单灵活，适用于只需要 JDBC 事务的场合）

关于 Spring DAO 子框架，网上有非常丰富的资料，因此本节不作讨论。本节主要讲述 Spring 与 JessMA DAO 子框架整合。在 Spring 环境下使用 JessMA DAO 子框架其实只需解决一个问题：***如何在 Spring 中创建和获取 DAO 对象？***

解决方法非常简单，只需把 DAO 对象定义为 Spring Bean，由 Spring 负责创建 DAO 对象以及把 DAO 对象注入到其他对象中。例如：

```
<!-- DAO Bean -->
<bean id="myDao" class="org.jessma.dao.FacadeProxy" factory-method="create">
    <constructor-arg value="dao.MyDao"/>
</bean>
```

```
<!-- Service Bean -->
<bean id="myService" class="service.MyService">
    <property name="dao" ref="myDao" />
</bean>
```

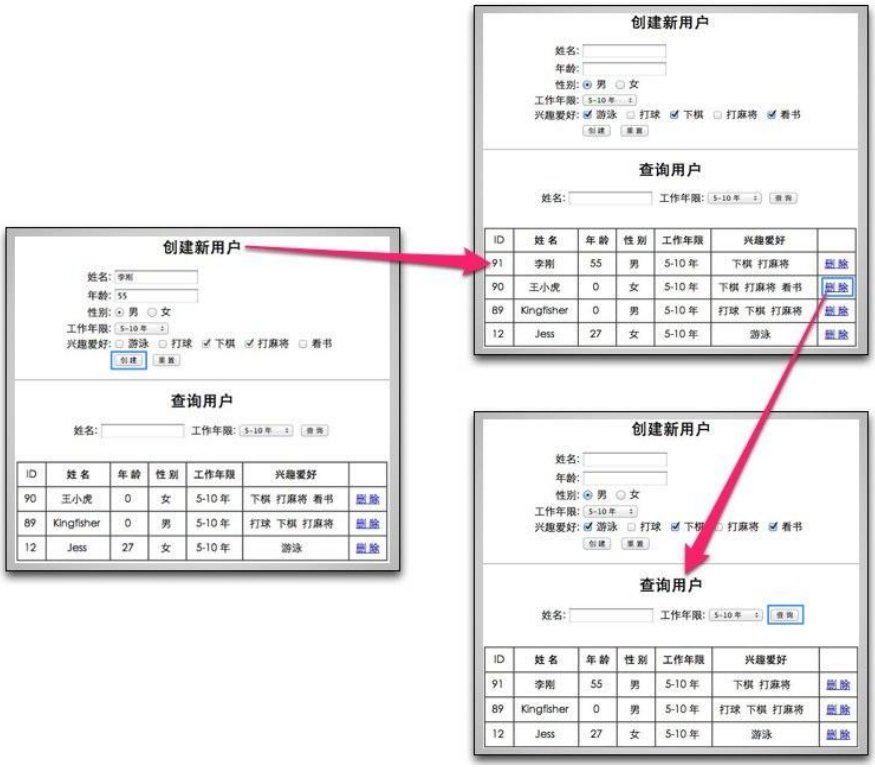
上述配置中，定义了一个 ID 为“myDao”的 DAO Bean，然后在“myService”Bean 中注入这个 DAO Bean。注意“myDao”的定义方式，它**必须使用 *FacadeProxy.create()* 静态工厂方法创建**。千万不要在 Spring 中使用下面的方式定义 JessMA DAO Bean：

```
<!-- 错误的 JessMA DAO Bean 定义方式 -->
<bean id="myDao" class="dao.MyDao" />
```

注意：JessMA 事务是 DAO 层事务，也就是说当外部调用某个 DAO 方法时，该方法作为一个事务单元；Spring 事务则是 Service 层事务，整个 Service 方法作为一个事务单元。

12.4 应用示例

本例修改 12.2 小节中的示例，为应用程序增加 Spring 支持，并使用典型的 Spring 程序结构（Action-Service-DAO）实现示例功能：把 Dao Bean 注入到 Service Bean 中，然后再把 Service Bean 注入到 Action 中。示例界面与以前一样：



1、 加入 Spring 相关程序包

把 Spring 的 jar 包和 jessma-ext-spring-x.x.x.jar 放入应用程序的 /WEB-INF/lib 目录。

2、 修改 /WEB-INF/web.xml（可选）

（1）设置 Spring 配置文件位置

```
<!-- Spring 配置文件：/WEB-INF/classes/applicationContext.xml -->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value>
</context-param>
```

（2）定义 Spring 监听器

```
<!-- Spring 监听器（可选，如果 web.xml 中没用配置 ContextLoaderListener，
则 SpringInjectFilter 会启动一个私有 ContextLoaderListener） -->
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

3、 在 MVC 主配置文件中加入拦截器：SpringInjectFilter

```
<filter pattern=".*" class="org.jessma.ext.spring.SpringInjectFilter"/>
```

4、 创建 Service：UserService

```
（参考 MyJessMA 工程源码：src/service.UserService）
```

5、 创建 Spring 配置文件 applicationContext.xml，定义 DAO Bean 和 Service Bean

```
<bean id="userDao" class="org.jessma.dao.FacadeProxy" factory-method="create">
    <constructor-arg value="dao.mybatis.UserDao"/>
</bean>
<bean id="userService" class="service.UserService">
    <property name="userDao" ref="userDao" />
</bean>
```

6、 修改 MVC 配置文件，加入 Action：user

```
<actions path="/test/spring">
    <action name="user" class="action.test.SpringUserAction">
        <entry name="create">
            <result type="chain">./user!query</result>
        </entry>
        <entry name="delete">
```

```
                <result type="chain">./user!query</result>
            </entry>
            <entry name="query" method="findUsers">
                <result>/jsp/test/spring/test_user.jsp?fromQueryAction=true</result>
            </entry>
            <result>/jsp/test/spring/test_user.jsp</result>
        </action>
    </actions>
```

7、 创建 Action：action.test.SpringUserAction

（参考 MyJessMA 工程源码：src/action.test.SpringUserAction）

上述代码中，每个方法的注入内容如下：

名 称	启用注入	成员属性	Bean 名称	Bean 类型
（默认）	*否*	-	-	-
create	是	userService	userService	service.UserService
delete	是	userService	userService	-
findUsers	是	userService	-	service.UserService
execute	否	-	-	-

8、 创建测试页面：/jsp/test/spring/test_user.jsp

测试页面的代码与 12.2 小节中的测试页面代码完全一样，只需把几个 Action 链接分别改为 ***test/spring/user!create.action***（***user!query.action***、***user!delete.action***）。（代码请参考：《/jsp/test/dao/test_user_1.jsp》）

9、 修改 /jsp/index.jsp：加入测试链接

```
<li> <a href="test/spring/user.action">测试 Spring</a></li>
```

12.5 JessMA 与 Guice 整合

Guice 是一个轻量级的 IoC 容器，没有完整的生命周期管理机制。因此 JessMA 与整合时 Guice 只需关注 JessMA 的 MVC 和 DAO 子框架与 Guice 整合。

上述配置中，定义了一个 ID 为“myDao”的 DAO Bean，然后在“myService”Bean 中注入这个 DAO Bean。注意“myDao”的定义方式，它**必须使用 *FacadeProxy.create()* 静态工厂方法创建**。千万不要在 Spring 中使用下面的方式定义 JessMA DAO Bean：

◆ **MVC 子框架与 Guice 整合**

与整合 Spring 类似，***jessma-ext-guice*** 提供了 org.jessma.ext.guice.**GuiceInjectFilter** 拦截器和 **@GuiceBean** / **@GuiceBeans** 注解用于在 Action 中注入 Guice Bean。详细使用方法可参考 12.2 小节。

◆ **DAO 子框架与 Guice 整合**

由于 JessMA 不能在 DAO 方法外部直接创建 DAO 对象并使用（自定义事务的情形除外），只能使用它们的代理对象。因此应用程序需要显式或隐式地创建它们的代理对象（通过 FacadeProxy 的 *create(...)* / *getXxxCommitProxy(...)* 方法或 *@DaoBean* / *@DaoBeans* 注解），而 Guice 并不能像 Spring 那样可以在配置文件中通过工厂方法创建 DAO 代理对象，所以 JessMA DAO 子框架与 Guice 整合时显得有点“别扭”，具体的整合方法是：**创建 GuiceInjectFilter 的子类，并在该子类的 *configModules()* 方法中逐个绑定 DAO 对象，然后把该子类配置为实际的 Guice 拦截器。**参考以下示例代码：

1、 创建自定义 Guice 拦截器：global.filter.MyGuiceInjectFilter

```
package global.filter;

public class MyGuiceInjectFilter extends GuiceInjectFilter
{
	@Override
	protected Collection<Module> configModules()
	{
		Set<Module> modules = new HashSet<Module>();

		// 绑定 DAO 对象
		modules.add(new Module()
		{
			@Override
			public void configure(Binder binder)
			{
				// 配置绑定规则
				binder.bind(a.b.c.Dao1.class).toInstance(FacadeProxy.create(a.b.c.Dao1.class));
				binder.bind(a.b.c.Dao2.class).toInstance(FacadeProxy.create(a.b.c.Dao2.class));
				binder.bind(a.b.c.Dao3.class).toInstance(FacadeProxy.create(a.b.c.Dao3.class));
				// .....
			}
		});

		// 绑定其它对象
		// .....

		return modules;
	}
```

```
}
```

2、 在 MVC 主配置文件中加入拦截器：MyGuiceInjectFilter

```
<filter pattern=".*" class="global.filter.MyGuiceInjectFilter"/>
```

具体的整合细节请参考 MyJessMA 示例工程中的相关整合例子。

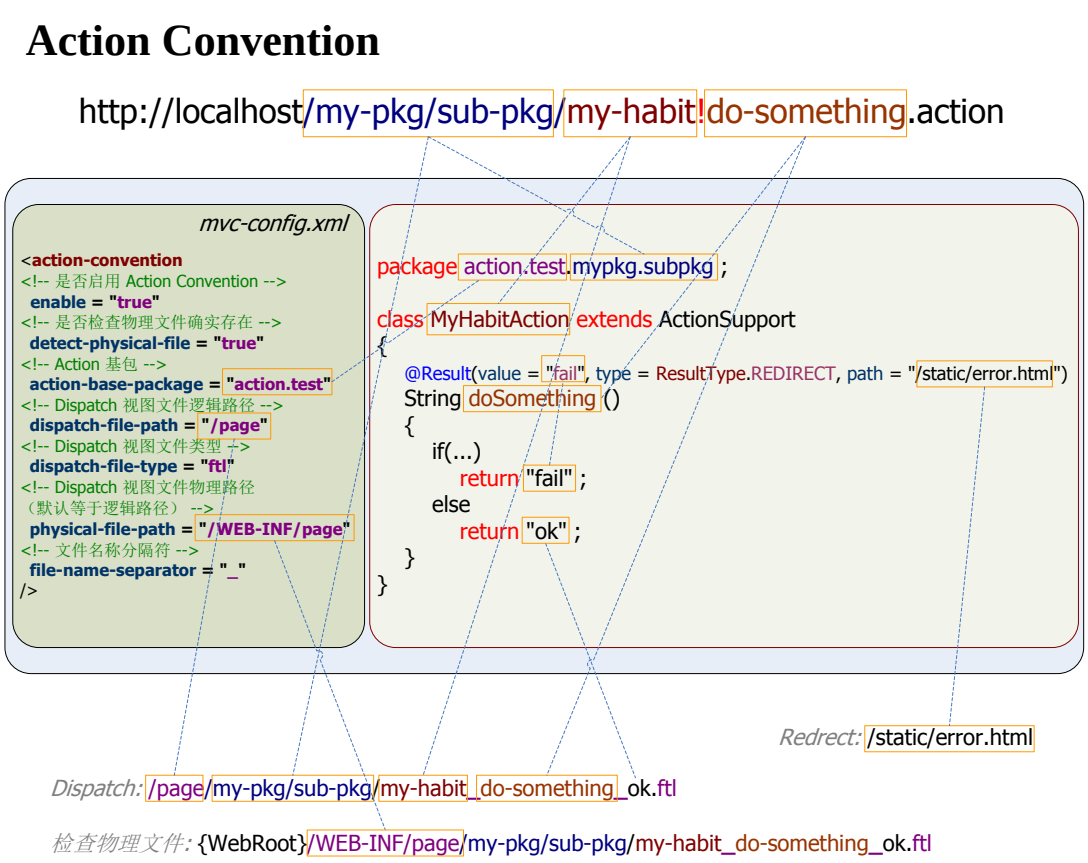
13 应用篇（十二）—— Action Convention

前面所有章节中 Action 都是通过 MVC 配置文件来表述的，包括 Action Path、Action Entry、Action Result 和 Exception Mapping。应用程序启动时会读取 MVC 配置文件并加载所有 Action。通过配置文件定义 Action 的优点是非常灵活，可以随时调整 Action 配置。但另一方面，随着 Action 数量的增加，配置文件也会变得越来越庞大，通过分割配置文件虽然可以降低配置文件的复杂度，但配置 Action 的工作量还是会随着 Action 的数目呈比例增加。另一方面，应用程序在使用的过程中，Action 配置是通常是固定的的，极少会发生需要修改 Action 配置的情形。

为了尽量简化 MVC 配置文件，并减少配置 Action 的工作量，JessMA 加入了 Action Convention 机制，Action Convention 基于一些编程约定和注解自动推导出 Action 配置并动态加载 Action，从而避免大量纷繁枯燥的 Action 配置工作。

注意：Action Convention 与 MVC 配置文件之间是互补关系而不是替代关系，Action Convention 简化了绝大部分 Action 的配置工作，但 MVC 的全局配置项以及一些个性化 Action 的配置还是需要 MVC 配置文件来定义。

下图的示例展示了 Action Convention 的基本原理：



✓ **Action 实现类所在包 ：**

由 <action-base-package> 配置和 Action 路径确定。

- ✓ **Action 实现类 ：**

  由 Action 名称确定。

- ✓ **Action 入口方法 ：**

  由 Action 名称的附加部分确定（“!”后面的部分）。

- ✓ **输出视图 URL ：**

  a) 默认由 <dispatch-file-path>、<dispatch-file-type>、<file-name-separator> 配置和 Action 路径、Action 名称、Action Result 确定。

  b) 通过 Action 入口方法或 Action 类中声明的 @Result/@Results 注解确定。

通过上述示例可以看出，Action Convention 主要通过以下几方面配合来实现：

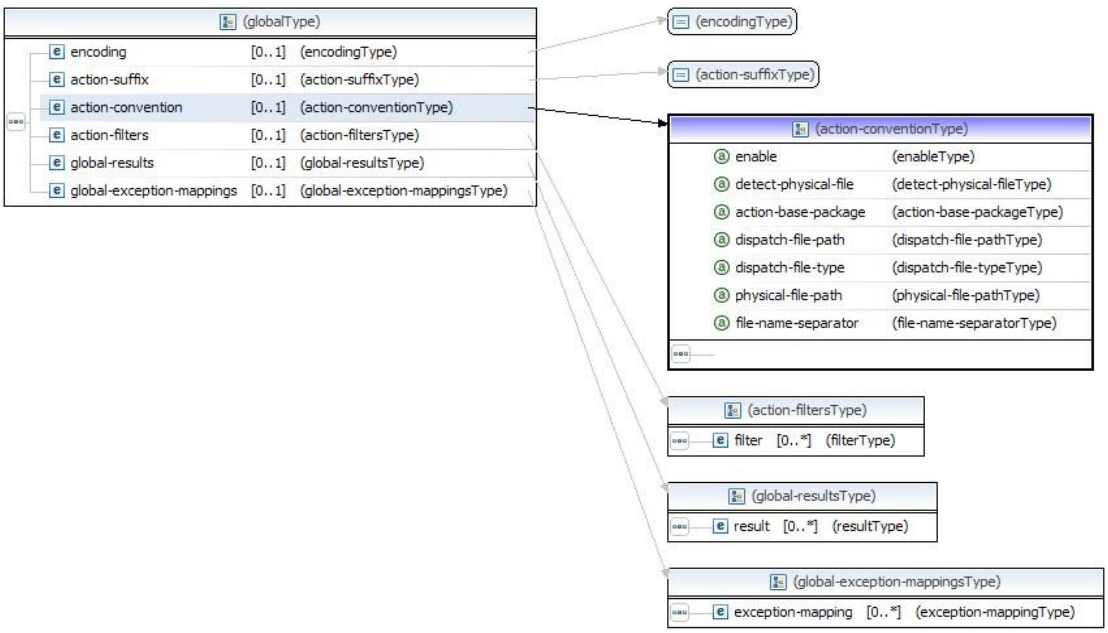
- **MVC 配置文件的 <action-convention> 配置项**
- **Action 的 URL 请求地址信息**
- **@Result / @Results 和 @ExceptionMapping / @ExceptionMappings 注解**

13.1 Convention 配置

JessMA 利用以下配置项来支持 Action Convention：

- **<global>/<action-convention>**

<action-convention/> 用来配置 Action Convention 的全局属性，其结构定义如下图所示：



21. 是否启用 Action Convention 机制：enable（可选，默认：true）

如果该属性为 false，则应用程序不会启用 Action Convention 机制，所有 Action 必须通

过 MVC 配置文件配置。

22. Action 实现类基包：action-base-package（必须）

Action 实现类的完整类名由基包和 Action 请求的地址信息确定，如：

```
示例：
/* ------------------------------------- 配置 ------------------------------------- */
// action-base-package ： aaa.bbb
/* ------------------------------------ 请求地址 ------------------------------------ */
// Action 请求地址     ： http://${host}/${context}/ccc/ddd-eee/xyz!mmm.action
/* ------------------------------------ 推导结果 ------------------------------------ */
// Action 实现类       ： aaa.bbb.ccc.dddeee.Xyz
// Action 入口方法     ： String mmm()
```

23. 输出视图根路径：dispatch-file-path（可选，默认 :" /WEB-INF/page"）

输出视图的完整路径由其根路径、Action 请求的地址信息和 Action Result 等确定，如：

```
示例：
/* ------------------------------------- 配置 ------------------------------------- */
// dispatch-file-path     ： /aaa/bbb
// dispatch-file-type     ： ftl
// file-name-separator    ： _
/* ------------------------- 请求地址和 Action Result ------------------------- */
// Action 请求地址     ： http://${host}/${context}/ccc/ddd-eee/xyz!mmm.action
// Action Result       ： rrr
/* ------------------------------------ 推导结果 ------------------------------------ */
// 输出视图完整路径： /aaa/bbb/ccc/ddd-eee/xyz_mmm_rrr.ftl
```

24. 输出视图扩展名：dispatch-file-type（可选，默认 :" jsp"）

用于设置输出视图的文件扩展名（见上例）。

25. 输出视图名称分隔符：file-name-separator（可选，默认 :" _"）

用于组合视图文件名。视图文件名由 Action URL 的 Action Name 和 Action Entry Name 部分与 Action Result 组成，它们之间以文件名分隔符连接（见上例的 Result Path）。

26. 是否检测视图物理文件：detect-physical-file（可选，默认：true）

如果该属性设置为 true，则应用程序会对输出视图进行检测，检查其对应的物理文件是否存在，不存在则会抛出异常；否则，如果该属性设置为 false，应用程序不会检测物理视图文件。

27. 视图物理文件根路径：physical-file-path（可选，默认 :""）

视图物理文件的完整路径由物理文件根路径、Action 请求的地址信息和 Action Result 等确定，如：

```
示例：
/* ---------------------------------------- 配置 ---------------------------------------- */
// dispatch-file-path          : /aaa/bbb
// physical-file-path          : /fff/ggg
// dispatch-file-type          : ftl
// file-name-separator         : _
/* ---------------------------- 请求地址和 Action Result ---------------------------- */
// Action 请求地址             : http://${host}/${context}/ccc/ddd-eee/xyz!mmm.action
// Action Result               : rrr
/* ---------------------------------- 推导结果 ---------------------------------- */
// 输出视图完整路径            : /aaa/bbb/ccc/ddd-eee/xyz_mmm_ rrr.ftl
// 视图物理文件完整路径        : /fff/ggg/ccc/ddd-eee/xyz_mmm_ rrr.ftl
```

- ✓ 当 detect-physical-file 为 true 时，会检查输出视图对应的物理文件是否存在。
- ✓ 当 physical-file-path 为空（默认值）时，表示输出视图路径与其物理文件路径一致。如果使用 JSP 作为输出视图，由于输出视图路径与其物理文件路径肯定是一致的，所以不必配置 physical-file-path；但如果使用 FreeMarker 或 Velocity 等模板作为输出视图，结合模板引擎的配置，输出视图路径与其物理文件路径可能会不同，这种情况下则需要配置 physical-file-path。

13.2 Convention 推导规则

通过 Action Convention 机制处理 Action 请求时，需要推导出以下 5 个部分（下面定义一些元素代号，然后结合代号来详细阐述每个部分的推导规则）：

◆ **Action Convention 配置**

- **Action 实现类基包 ：{base-package}**
- **输出视图根路径 ：{dispatch-path}**
- **输出视图扩展名 ：{ext}**
- **输出视图名称分隔符 ：{_}**
- **视图物理文件根路径 ：{physical-path}**

◆ **Action 请求 URL**

- **Action 请求路径 ：{action-path}**
- **Action 名称 ：{action-name}**
- **Action 入口名称 ：{entry-name}**

- **Action 请求 URL ：**
  http://{host}/{context}/**{action-path}**/**{action-name}**[**!{entry-name}**].action

◆ **其他**

- **Action Result ：{result}**
- **应用程序根目录 ：{WebRoot}**

*** **推导内容：**

1、 Action 实现类所在包

{base-package}.{action-path}

{action-path} 的路径分隔符（“/”）替换成包分隔符（“.”），去除包路径中所有横线间隔符（“-”），包路径中所有字母均为小写。

如：action.base + my-pkg/sub-pkg –> action.base.mypkg.subpkg

2、 Action 实现类

{action-name}[Action]

去除 {action-name} 中所有横线间隔符（“-”），原来由横线分隔符隔开的各个部分的首字母转为大写。如果找不到这个类则会尝试在类名后面加上“Action”字样再次查找。

如：my-user –> MyUser 或 MyUserAction

3、 Action 入口方法：

{entry-name}

去除 {entry-name} 径中所有横线间隔符（“-”），{entry-name} 的第一个字母转为小写，其余的原来由横线分隔符隔开的各个部分的首字母转为大写。如果请求路径中没有“!{entry-name}”部分，则入口方法为 ***execute()***。

如：my-user –> MyUser#execute() 或 MyUserAction#execute()
my-user!my-entry –> MyUser#myEntry() 或 MyUserAction# myEntry()

4、 输出视图：

{WebRoot}/{dispatch-path}/{action-path}/{action-name}[{_}{entry-name}][{_}{result}.{ext}

1) 如果请求路径中没有“!{entry-name}”部分，则输出视图的名称将没有

“{_}{entry-name}”部分。

2) 如果推导出的视图名（“{action-name}[{_}{entry-name}]{_}{result}”）以“{_}success”结尾，则应用程序首先会检查视图对应的物理文件是否存在，如果不存在则尝试去除名称尾部的“{_}success”再次检查。

如：/my-pkg/my-user (success) –> /my-pkg/my-user_success.jsp 或 /my-pkg/my-user.jsp
/my-pkg/my-user!my-entry (other-result) –> /my-pkg/my-user_my-entry_ other-result.jsp

5、 视图物理文件：

{WebRoot}/{physical-path}/{action-path}/{action-name}[{_}{entry-name}]{_}{result}.{ext}

除了用 {physical-path} 取代 {dispatch-path} 之外，视图物理文件路径的推导规则与输出视图的推导规则一致。

13.3 Convention 注解

除了上节所阐述的推导规则以外，Action Convention 还提供了 @Result / @Results 与 @ExceptionMapping / @ExceptionMappings 两类注解用于覆盖默认的 Action Result 和 Exception Mapping 处理行为。

13.3.1 @Result / @Results

@ Result 有以下三个注解参数：

- **String value()** ：**Result Name**
  （默认："*success*"）
- **Action.ResultType type()** ：**Result Type**
  （默认：value() == *"none" ? FINISH : DISPATCH*）
- **String path()** ：**Result Path**
  （默认：""）

@Results 只有一个注解参数：

- **Result[] value()** ：**@Result 数组，用于定义多个 @Result**
  （默认：{}，空数组）

*** **关于 @Result / @Results 的几点特别说明：**

1、 声明在 Action 类中的 @Result / @Results 定义该 Action 类的公共 Result，所有入口方法都共享这些 Result 定义。公共 Result 的效果相当于 MVC 配置文件中的“<actions>/<action>/<result>”。

2、 声明在 Action 入口方法中的 @Result / @Results 定义该入口方法的私有 Result，只

对该入口方法有效，私有 Result 的优先级高于公共 Result。私有 Result 的效果相当于 MVC 配置文件中的“<actions>/<action>/<entry>/<result>”。

3、 如果 Action 类中或入口方法中同时声明了 @Resut 和 @Results，应用程序会同时处理它们的并集，自动过滤 value() 参数重复的 @Resut 声明，采用它所找到的第一个 @Resut。

13.3.2 @ExceptionMapping / @ExceptionMappings

@ExceptionMapping 有以下两个注解参数：

- **Class<? extneds Exception> value()** ：**Exception 类型**（默认：java.lang.Exception）
- **String result()** ：**Result Name**（默认：*"exception"*）

@ExceptionMappings 只有一个注解参数：

- **ExceptionMapping[] value()** ：**@ExceptionMapping 数组**（默认：{}，空数组）

*** **关于 @ExceptionMapping / @ExceptionMappings 的几点特别说明：**

1、 声明在 Action 类中的 @ExceptionMapping / @ExceptionMappings 定义该 Action 类的公共 Exception Mapping，所有入口方法都共享这些 Exception Mapping 定义。公共 Exception Mapping 的效果相当于 MVC 配置文件中的“<actions>/<action>/<exception-mapping>”。

2、 声明在 Action 入口方法中的 @ExceptionMapping / @ExceptionMappings 定义该入口方法的私有 Exception Mapping，只对该入口方法有效，私有 Exception Mapping 的优先级高于公共 Exception Mapping。私有 Exception Mapping 相当于 MVC 配置文件中的“<actions>/<action>/<entry>/< exception-mapping>”。

3、 如果 Action 类中或入口方法中同时声明了 @ExceptionMapping 和 @ExceptionMappings，应用程序会同时处理它们的并集，自动过滤 value() 参数重复的 @ExceptionMapping 声明，采用它所找到的第一个 @ExceptionMapping。

注意：JessMA 的 Action Convention 机制只提供了 @Result / @Results 和 @ExceptionMapping / @ExceptionMappings 两类注解，并没有像其他 MVC 框架那样提供更多的注解用于定义 Actin（如：@Action）和 Action Path（如：@Namespace）等。主要出于以下两点考虑：

1、 这类注解实用性不高，用 MVC 配置文件取代它们更直观。

2、 最重要的是，这类注解非常容易造成程序混乱，使用也麻烦。无法通过在 MVC 配置文件或 URL 特征信息迅速确定 Action 实现类及其入口方法。

13.4 查找顺序

加入 Action Convention 机制后，Action Entry、Action Result 和 Exception Mapping 可能会在多个地方定义，应用程序会按照规定的顺序查找它们，顺序如下：

28. 【Entry Entry】

→ *MVC 配置文件的 Action Entry*
→ *Action Convention 推导的 Action Entry*

29. 【Action Result】

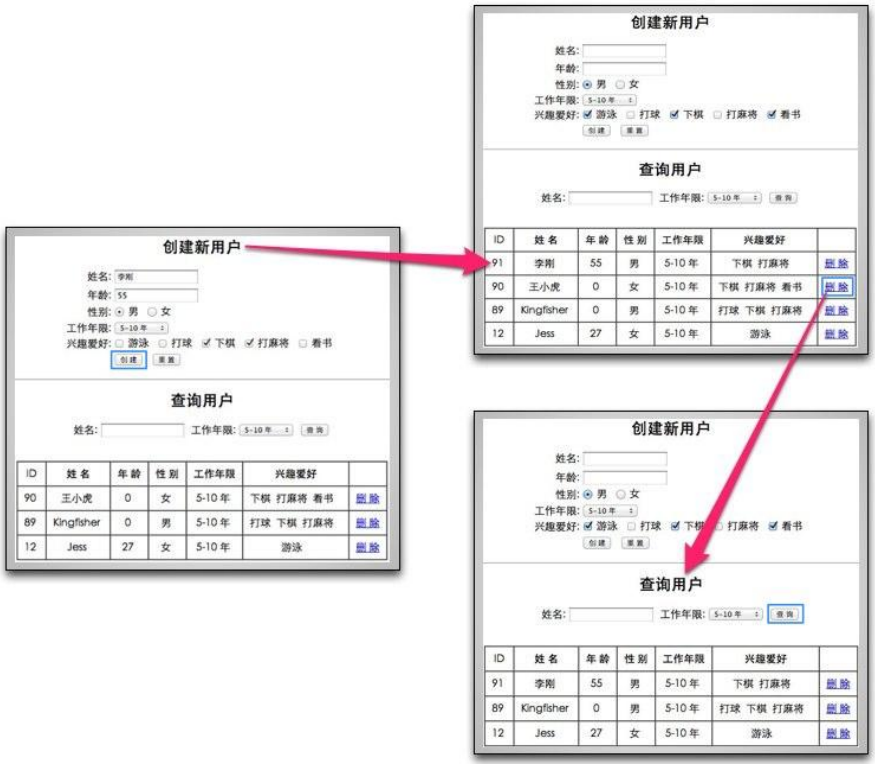
→ *MVC 配置文件的 Action Entry Result*
→ *MVC 配置文件的 Action Result*
→ *Action 入口方法的 @Result 注解*
→ *Action 类的 @Result 注解*
→ *MVC 配置文件的 Golbal Result*
→ *Action Convention 推导的 Result*

30. 【Action Exception-Mapping】

→ *MVC 配置文件的 Action Entry Exception-Mapping*
→ *MVC 配置文件的 Action Exception-Mapping*
→ *Action 入口方法的 @ExceptionMapping 注解*
→ *Action 类的 @ExceptionMapping 注解*
→ *MVC 配置文件的 Golbal Exception-Mapping*

13.5 应用示例

本例修改 11.2 小节中的示例，利用 Action Convention 机制实现示例的功能，不必在 MVC 配置文件中配置任何 Action。示例界面与以前一样：



1、 在 MVC 主配置文件中加入 Action Convention 支持

```
<!-- Action Convention 配置
        enable                  ：是否启用 Action Convention 功能（默认：true）
        detect-physical-file    ：是否检查 Action Result 物理视图文件的确存在（默认：true）
        action-base-package     ：Action 类的基包（默认：""）
        dispatch-file-path      ：转发视图的基路径（默认："/WEB-INF/page"）
        dispatch-file-type      ：转发视图的类型（默认："jsp"）
        physical-file-path      ：物理视图文件的基路径（默认：""，与 'dispatch-file-path' 一致）
        file-name-separator     ：视图名称分隔符（默认："_"）
-->
<action-convention
        enable="true"
        detect-physical-file="true"
        action-base-package="action"
        dispatch-file-path="/jsp"
        dispatch-file-type="jsp"
        physical-file-path=""
        file-name-separator="_"
/>
```

2、 创建 Action：action.test.conv.UserAction

（参考 MyJessMA 工程源码：src/action.test.conv.UserAction）

应用程序运行期间 Action Convention 机制会为 UserAction 动态创建以下 Action Entry：

Entry Name	访问地址	入口方法	Action Entry Result
create	user!create.action	**create**	
delete	user!delete.action	**delete**	
query	user!query.action	**query**	**[list|dispatch]**：/jsp/test/conv/user.jsp
	user.action	***execute***	***[success|dispatch]***：*/jsp/test/conv/user.jsp*

另外，还会为 UserAction 的所有 Entry 动态创建以下公共 Action Result：

Action	Action Result
action.test.conv.UserAction	***[query|chain]***：*./user!query*

3、 创建测试页面：/jsp/test/conv/user.jsp

测试页面的代码与 11.2 小节中的测试页面代码完全一样，只需把几个 Action 链接分别改为 ***test/conv/user!create.action*** （***user!query.action*** 、 ***user!delete.action***），由于本例中“**fromQueryAction**”以 Action Attribute 方式提供，而不是以请求参数方式提供，因此还需把“**param.fromQueryAction**”替换为“**__action.fromQueryAction**”。（代码请参考：《/jsp/test/dao/test_user_1.jsp》）

4、 修改 /jsp/index.jsp：加入测试链接

```
<li> <a href="test/conv/user.action">测试 Action Convention</a></li>
```

14 应用篇（十三）—— REST Convention

REST 是英文 Representational State Transfer 的缩写，这个术语由 Roy Thomas Fielding 博士在他的论文《Architectural Styles and the Design of Network-based Software Architectures》中提出。从这篇论文的标题可以看出：REST 是一种基于网络的软件架构风格。

REST 架构是针对传统 Web 应用提出的一种改进，是一种新型的分布式软件设计架构。对于异构系统如何进行整合的问题，目前主流做法都集中在使用 SOAP、WSDL 和 WS-* 规范的 Web Services。而 REST 架构实际上也是解决异构系统整合问题的一种新思路。如果开发者在开发过程中能坚持 REST 原则，将可以得到一个使用了优质 Web 架构的系统，从而为系统提供更好的可伸缩性，并降低开发难度。关于 REST 架构的主要原则如下：

- ✓ 网络上的所有事物都可被抽象为资源（Resource）。
- ✓ 每个资源都有一个唯一的资源标识符（Resource Identifier）。
- ✓ 同一资源具有多种表现形式。
- ✓ 使用标准方法操作资源。
- ✓ 通过缓存来提高性能。
- ✓ 对资源的各种操作不会改变资源标识符。
- ✓ 所有的操作都是无状态的（Stateless）。

JessMA 通过 REST 扩展插件（***jessma-ext-rest-x.x.x.jar***）全面支持 REST 风格应用程序。***jessma-ext-rest*** 利用两个组件配合处理 REST 请求：

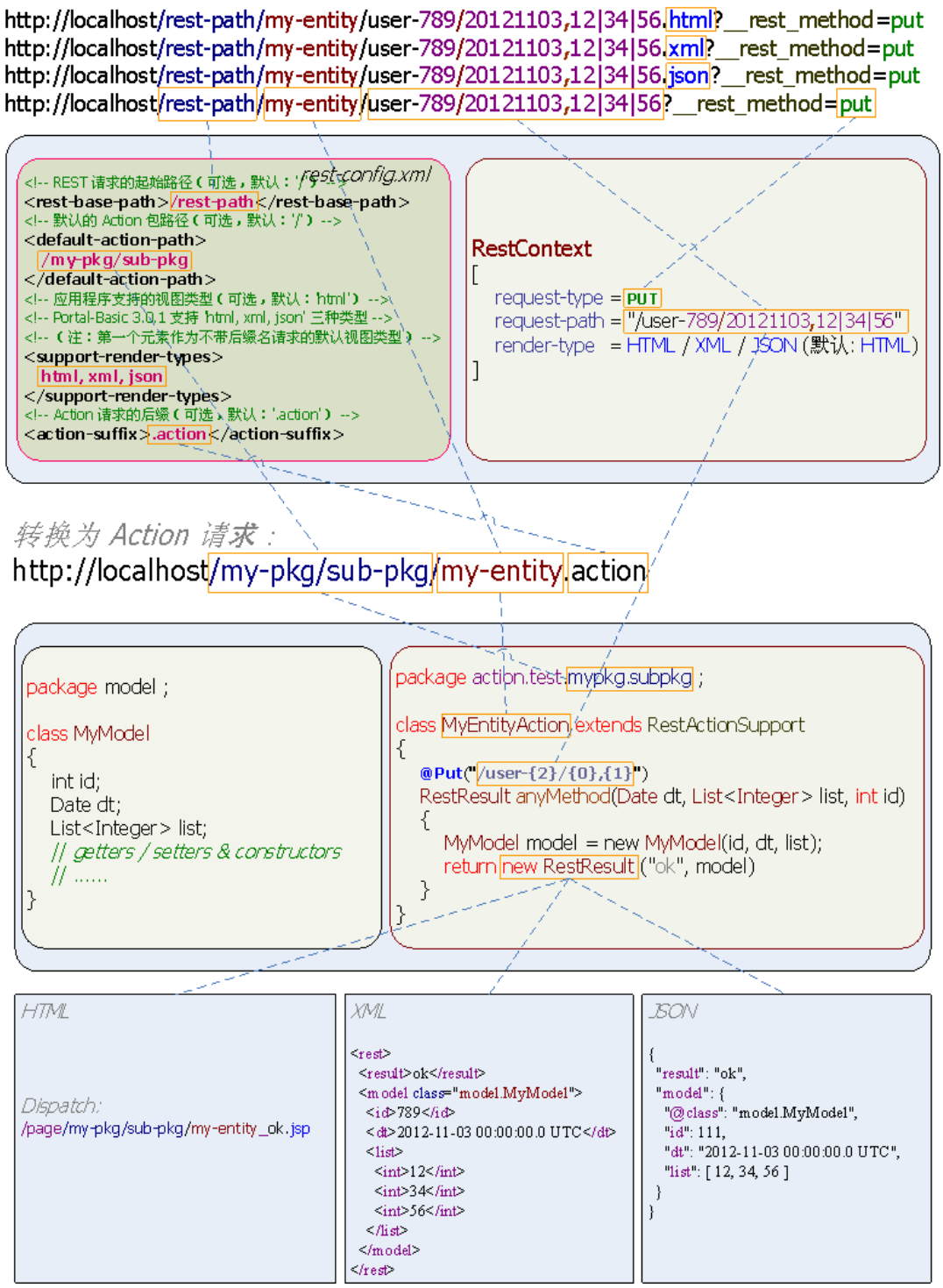
- **REST 过滤器**（org.jessma.ext.rest.**RestDispatcher**）
  - ✧ 根据请求信息创建 RestContext 对象
  - ✧ 把 REST 请求转换为 Action 请求并转发到 JessMA MVC
- **REST 控制器**（org.jessma.ext.rest.**RestActionSupport**）
  - ✧ 实现 REST 请求的整体处理逻辑
  - ✧ 通过子类处理具体的 REST 请求
  - ✧ 根据 RestContext 的 *render-type* 输出相应的结果视图

REST 风格的 JessMA 应用程序通常需做以下几项工作：

- ✓ 在 REST 配置文件（rest-config.xml）中配置 RestDispatcher 的转发规则
- ✓ 为每个 REST 实体创建 RestActionSupport 的子类，来处理实体相关的所有请求
- ✓ 如果输出视图类型为 HTML，还需要创建视图页面

下图的示例展示了 REST Convention 的基本原理：

REST Convention



1、 RestDispatcher 根据转发规则和 REST 请求地址信息创建一个 RestContext 对象，并把 REST 请求转换为 Action 请求：*http://localhost/my-pkg/sub-pkg/my-entity.action*。

2、 JessMA MVC 找到 *action.test.mypkg.subpkg.MyEntityAction* 来处理这个 Action 请求（通过 Action Convention 机制），MyEntityAction 继承于 RestActionSupport。Action 的入口方法为 RestActionSupport 提供的 *execute()*。

3、 *execute()* 扫描到“*ActionResult anyMethod(Date, List<Integer>, int)*”能匹配这个 RestContext 的请求模式（“*PUT：/user-{X}/{X},{X}*”），因此请求最终会交由 *anyMethod()* 处理。*anyMethod()* 处理完这个请求后返回一个 RestResult 对象。

4、 *execute()* 根据 RestContext 的 *render-type* 作不同的处理：

1） 如果 *render-type* 为 XML 或 JSON，则把 RestResult 对象转换为 XML 或 JSON 格式输出到客户端，并向 MVC 框架返回 Action Result：***none***。

2） 如果 *render-type* 为 HTML 则，直接向 MVC 框架返回 Action Result：***ok***。

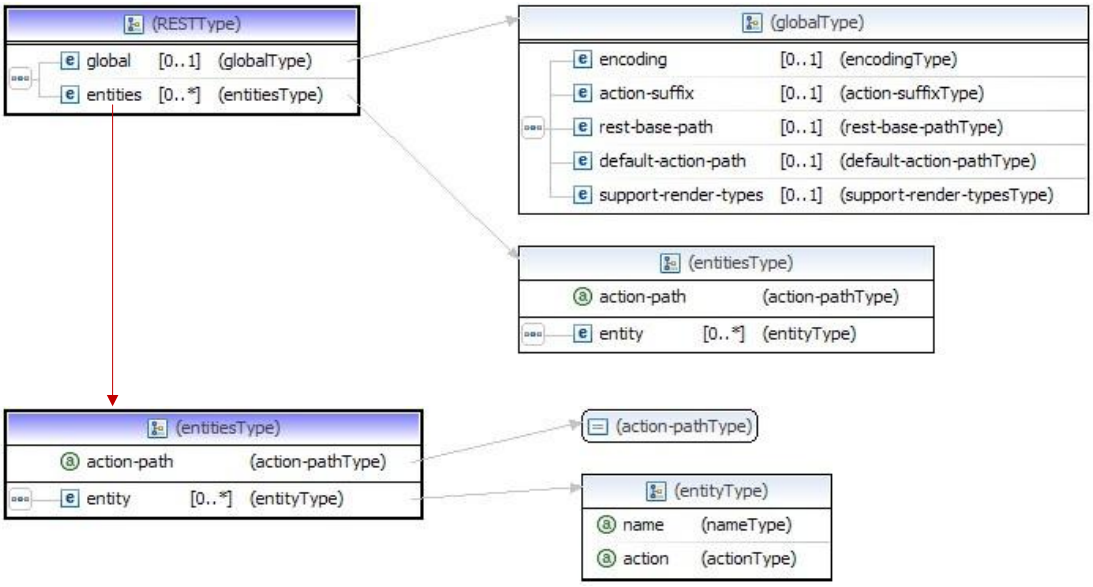
5、 MVC 框架接收到 Action Result 后，如果为 ***none*** 则通常不作任何处理；如果为 ***ok*** 则转发到“/page/my-pkg/sub-pkg/my-entity_ok.jsp”（通过 Action Convention 机制）。

14.1 REST 过滤器（RestDispatcher）

RestDispatcher 的职责是根据 REST 配置选项和 URL 特征信息创建 RestContext 对象，并把 REST 请求转换为 Action 请求，然后转发给 RestActionSupport 的子类处理。

14.1.1 RestDispatcher 配置

RestDispatcher 的配置文件默认为：***${CLASS}/rest-confing.xml***。它的 XSD 定义如下：



注意：动态更新 REST 配置请参考：《更新 REST 配置》章节

REST 配置文件包含 2 个配置项：

✓ **全局配置：global**

配置 REST 全局选项，包括编码格式、Action 后缀名和 REST 默认转发规则。

✓ **REST 实体配置：entities**

配置用户自定义的“REST 实体 —Action”映射规则。

31. 全局配置：global（可选）

- **HTTP 编码：encoding（可选，默认：不设置）**

设置 request 和 response 的字符编码格式，因此通常设置为 UTF-8 或 GBK。
（字符编码格式应该和 MVC 框架的编码格式一致）

```
示例：<encoding>UTF-8</encoding>
```

- **Action 后缀名：action-suffix（可选，默认："".action""）**

设置 Action 的后缀名，RestDispatcher 把 REST 请求转换为 Action 请求时，该值作为 Action URL 的后缀。
（Action 后缀名应该和 MVC 框架的 Action 后缀名一致）

```
示例：<action-suffix>.do</action-suffix>
```

- **REST 请求起始路径：rest-base-path（可选，默认："/"）**

REST 请求的 URL 格式为：

http://{host}/{context}/{rest-base-path}/{entity}[/{request-path}[.{render-type}]]

RestDispatcher 从 URL 的 rest-base-path 开始解释请求各个部分：

```
示例 1：
/* ------------------------------------ 配置 ------------------------------------ */
// rest-base-path        ：/aaa/bbb
// support-render-types：html, json
/* ---------------------------------- 请求地址 ---------------------------------- */
// REST 请求地址          ：http://${host}/${context}/aaa/bbb/ccc/ddd/eee
/* ---------------------------------- 推导结果 ---------------------------------- */
// entity                ：ccc
// request-path          ：/ddd/eee
// render-type           ：html
```

```
示例 2：
/* ------------------------------------ 配置 ------------------------------------ */
// rest-base-path        ：/
// support-render-types：html, json
/* ---------------------------------- 请求地址 ---------------------------------- */
// REST 请求地址          ：http://${host}/${context}/aaa/bbb/ccc/ddd/eee.json
/* ---------------------------------- 推导结果 ---------------------------------- */
```

```
// entity          : aaa
// request-path    : /bbb/ccc/ddd/eee
// render-type     : json
```

```
示例3：
/* ---------------------------------------- 配置 ---------------------------------------- */
// rest-base-path      : /aaa/bbb
// support-render-types : html, json
/* ---------------------------------------- 请求地址 ---------------------------------------- */
// REST 请求地址       : http://${host}/${context}/xxx/aaa/bbb/ccc/ddd/eee.xml
/* ---------------------------------------- 推导结果 ---------------------------------------- */
//（不是 REST 请求）不满足以下 2 个要求：
//    1) REST 请求不以 rest-base-path 开始
//    2) URL 后缀（xml）不是支持的视图类型
```

- **默认 Action 路径：default-action-path（可选，默认："/"）**

默认情况下，转发目标 Action 的完整路径由默认 Action 路径和 REST 实体组成：

```
示例：
/* ---------------------------------------- 配置 ---------------------------------------- */
// rest-base-path      : /aaa/bbb
// default-action-path : /my-path/sub-path
// support-render-types : html, json
// action-suffix       : .action
/* ---------------------------------------- 请求地址 ---------------------------------------- */
// REST 请求地址       : http://${host}/${context}/aaa/bbb/ccc/ddd/eee
/* ---------------------------------------- 推导结果 ---------------------------------------- */
// entity          : ccc
// request-path    : /ddd/eee
// render-type     : html
// Action URL      : http://${host}/${context}/my-path/sub-path/ccc.action
```

- **支持的视图类型集：support-render-types（可选，默认："html"）**

JessMA 目前支持 3 种视图类型：***html***，***xml*** 和 ***json***。support-render-types 用来设置应用程序支持的视图类型，每种类型之间用“,”隔开（如：“xml, json”），RestDispatcher 根据 REST 请求的后缀确定本次请求需要输出的视图类型。

1） 如果 REST 请求没有后缀则以 support-render-types 配置的第一个类型作为输出的视图类型（见上例）。
2） 如果 REST 请求的后缀不是 support-render-types 中支持的类型，则认为该请求不是 REST 请求（见上例）。

32. 自定义 REST 实体配置：entities（可选，*{0...n}*）

前面介绍 default-action-path 配置时说到：默认情况下，转发目标 Action 的完整路径由 default-action-path（默认 Action 路径）和 REST 实体组成（见上例）。但有时候应用程序基于一些特殊的原因，某些 REST 请求可能不希望直接把 default-action-path 和 REST 实体名称映射为 Action 完整路径。这时，需要用 entities 配置来自定义映射规则。

- **自定义 Action 路径：entities.action-path（可选，默认："/"）**

REST 请求转换为 Action 请求后的 Action 路径。

- **REST 实体名：entity.name（必须）**

待映射的 REST 实体。

- **自定义 Action 名：entity.action（可选，默认：""）**

REST 请求转换为 Action 请求后的 Action 名。如果 entity.action 为空（默认值）则 Action 名与实体名一致。

```
示例 1：
/* ------------------------------------ 配置 ------------------------------------ */
// rest-base-path        : /aaa/bbb
// default-action-path   : /my-path/sub-path
// action-suffix         : .action
// entities.action-path  : /xxx/yyy
// entity.name           : ccc
// entity.action         : my-act
/* ---------------------------------- 请求地址 ---------------------------------- */
// REST 请求地址          : http://${host}/${context}/aaa/bbb/ccc/ddd/eee
/* ---------------------------------- 推导结果 ---------------------------------- */
// entity                : ccc
// action-path           : /xxx/yyy
// action-name           : my-act
// Action URL            : http://${host}/${context}/xxx/yyy/my-act.action
```

```
示例 2：
/* ------------------------------------ 配置 ------------------------------------ */
// rest-base-path        : /
// default-action-path   : /my-path/sub-path
// action-suffix         : .action
// entities.action-path  : /xxx/yyy
// entity.name           : ccc
// entity.action         :
```

```
/* ---------------------------------- 请求地址 ---------------------------------- */
// REST 请求地址     : http://${host}/${context}/ccc
/* ---------------------------------- 推导结果 ---------------------------------- */
// entity            : ccc
// action-path       : /xxx/yyy
// action-name       : ccc
// Action URL        : http://${host}/${context}/xxx/yyy/ccc.action
```

14.1.2 Convention 推导规则

通过 REST Convention 机制处理 REST 请求时，需要推导出 RestContext 对象和 Action URL（下面定义一些元素代号，然后结合代号来详细阐述两者的推导规则）：

◆ **REST Convention 配置**

- **REST 请求起始路径 ：{rest-base-path}**
- **支持的视图类型 ：{render-types}**
- **默认 Action 路径 ：{def-action-path}**
- **Action 后缀 ：{suffix}**
- **自定义 Action 路径 ：{custom-action-path}**
- **自定义 Action 名 ：{custom-action-name}**

◆ **REST 请求 URL**

- **REST 请求路径 ：{rest-path}**
- **REST 实体名称 ：{entity}**
- **REST 请求附加信息 ：{request-path}**
- **REST 视图类型 ：{ext}**
- **REST 请求 URL ：**

http://{host}/{context}/**{rest-path}**/**{entity}**[/**{request-path}**][.**{ext}**]

*** **推导内容：**

1、 RestContext 对象

RestDispatcher 会为每个 REST 请求创建一个 RestContext 对象，RestContext 对象有以下 3 个属性：

- **RequestType requestType** ：请求类型（GET/POST/PUT/DELETE）
- **String requestPath** ：请求附加信息（**{request-path}**）
- **RenderType renderType** ：视图类型（HTML/XML/JSON）

✓ **requestType：**由于标准浏览器只支持 GET 和 POST 请求，为了模拟 PUT 和 DELETE 请求，RestDispatcher 接收到 REST 请求后会检查该请求是否带有“***__rest_method***”

参数，如果该参数存在则以该参数的值作为 RestContext 的 requestType，否则以当前请求类型作为 requestType。应用程序可以在 URL 地址或 Form 表单隐藏域中加入“***__rest_method***”参数。

- ✓ **requestPath：**对应于 REST 请求的 **{request-path}**，注意：不包含 **{entity}** 和 **{ext}** 部分，有可能为空。

- ✓ **renderType：**如果请求带有“***__rest_render***”参数，则以该参数的值作为 RestContext 的 renderType；否则，如果请求 URL 中的 **{ext}** 不为空，则以 **{ext}** 作为 RestContext 的 renderType；否则，以 **{render-types}** 的第一个元素作为 renderType。

2、 Action URL

- ➢ 默认转换规则：

  http://{host}/{context}/**{def-action-path}**/**{entity}**.**{suffix}**

- ➢ 自定义转换规则：

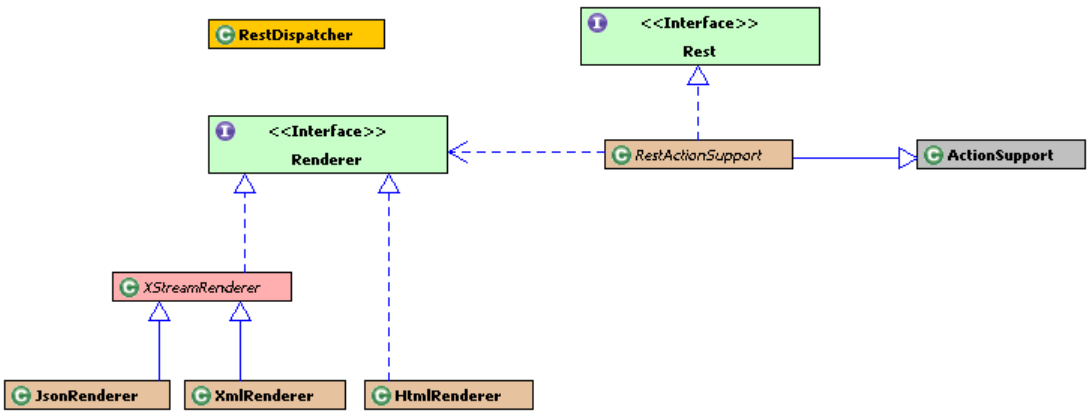
  http://{host}/{context}/**{custom-action-path}**/**{custom-action-name}**.**{suffix}**

RestDispatcher 执行 URL 转换时，在 REST 配置文件的 ***<entities>/<entity>*** 配置项中定义的 REST 实体遵循“自定义转换规则”，其他所有 REST 实体则遵循“默认转换规则”。

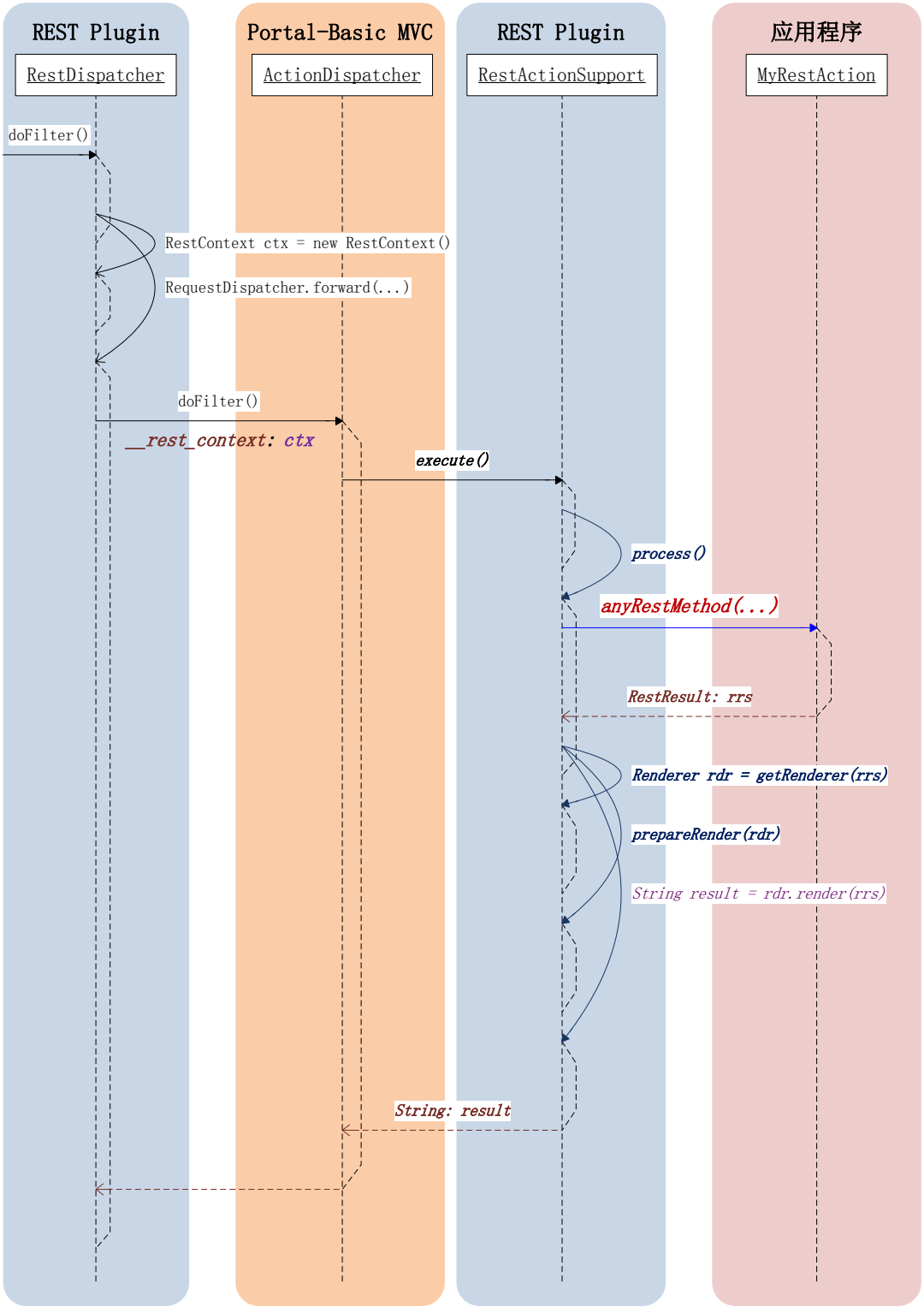
注意：RestDispatcher 会认为以下几种请求为非 REST 请求，不作任何处理：

- **请求路径不以 {rest-base-path} 开始**
- **{ext} 不为空且不包含在 {render-types} 中**
- **{entity} 为空**

14.2 REST 控制器（RestActionSupport 及其子类）



14.2.1 REST 请求处理流程



上图展示了应用程序处理 REST 请求的整个流程：

1、 RestDispatcher 过滤器的 *doFilter()* 接收到 REST 请求后，首先根据请求信息创建一

个 RestContext 对象，并把这个对象保存到名为“***__rest_context***”的 Request Attribute 中；然后把这个 REST 请求转换为 Action 请求并转发到 JessMA MVC。

2、 MVC 框架接收到 Action 请求后根据配置文件或 Action Convention 找到相应的 Action（图中的 *MyRestAction*）并调用它的入口方法。对于由 REST 请求转换得到的 Action 请求，其入口方法一般为 RestActionSupport 提供的 *execute()*。

3、 *execute()* 调用 *process()* 实现 Action 处理逻辑。*process()* 会找出并调用与 RestContext 匹配的 REST 入口方法（图中的 *anyRestMethod()*），REST 入口方法会返回一个 RestResult 对象（图中的 *RestResult : rrs*）。

4、 *process()* 接着调用 *getRenderer()* 获取视图对象，然后调用 *prepareRender()* 作输出前准备，最后调用视图对象（*Renender: rdr*）的 *render()* 输出视图。*render()* 会返回一个 ***String***，这个 ***String*** 最终作为 *execute()* 的 Action Result 返回到 JessMA MVC。

*** **注意：**

1） 默认 XML 和 JSON 视图（*XmlRenderer* / *JsonRenderer*）的 *render()* 通过 **XStream** 把 RestResult 对象转换为 XML 或 JSON 格式输出到客户端，并向 JessMA MVC 返回 Action Result：***none***。

2） 默认 HTML 视图（*HtmlRenderer*）的 *render()* 只做两件事情：

- ✓ 如果 RestResult 的 *model* 属性不为 ***null***，则把 *model* 设置为 Request Attribute：“***__rest_model***”供后续视图页面访问。
- ✓ 向 MVC 框架返回 RestResult 的 *result* 属性作为 Action Result。

3） RestActionSupport 预定义了一个特殊的 RestResult：***REST_NONE***，当 REST 入口方法返回 ***REST_NONE*** 时，则不向客户端输出任何内容。

因此，如果输出视图是 XML 或 JSON，或者 REST 入口方法返回的 RestResult 为 ***REST_NONE*** 时，应用程序不需要再做任何工作；如果输出视图是 HTML，则应用程序需要提供 Action Result 相应的视图页面。

RestActionSupport 在处理 REST 请求时，用到几个声明为 ***protected*** 的方法，子类可以改写这些方法从而修改 RestActionSupport 的默认行为：

- ✓ ***process()***：实现 REST 请求整体处理逻辑，一般情况下不建议子类改写本方法。
- ✓ ***processNotMatch()***：当 RestContext 不能与任何 REST 入口方法匹配时调用这个方法，默认：向客户端报告 404 错误，并向 JessMA MVC 返回 Action Result：***none***。
- ✓ ***getRenderer(RestResult)***：获取视图对象，默认：返回当前 RestContext 的 *renderType* 属性对应的默认视图。子类可以改写这个方法返回自定义视图对象。
- ✓ ***prepareRender(Renderer, RestResult)***：在执行 Renderer 的 *render()* 前调用，用于对视图进行额外处理，默认：根据 Renderer 类型设置 Content-Type HTTP 响应头。

14.2.2 REST 入口方法

REST 入口方法由 Action 入口方法 *execute()* 调用，它负责处理 REST 请求的业务逻辑，并把执行结果封装为 RestResult 对象传回 *execute()* 。REST 入口方法定义规则：

- **声明为 *public* 的实例方法**
- **返回值为 *RestResult* 类型或其子类型**
- **方法名称不限制**
- **参数个数不限制，但参数类型必须为：8 种基础类型或其包装类型、String、Date 及其数组或集合（List、Set）。其中集合必须指定类型参数。**

```
示例 1：
public RestResult myMethod()                          // OK
protected RestResult myMethod()                       // ERROR
public Sting myMethod()                               // ERROR
public RestResult myMethod(int, String, List<Double>) // OK
public RestResult myMethod(int, String, List)         // ERROR
public RestResult myMethod(int, MyClass)              // ERROR
```

RestActionSupport 内置了 8 个 Rails-style REST 标准入口方法，子类可以实现这些方法处理 Rails-style REST 标准请求：

- **RestResult index() throws Exception** // ***GET:*** */{entity}*
- **RestResult create() throws Exception** // ***POST:*** */{entity}*
- **RestResult deleteAll() throws Exception** // ***DELETE:*** */{entity}*
- **RestResult update(int id) throws Exception** // ***PUT:*** */{entity}/{id}*
- **RestResult delete(int id) throws Exception** // ***DELETE:*** */{entity}/{id}*
- **RestResult show(int id) throws Exception** // ***GET:*** */{entity}/{id}*
- **RestResult edit(int id) throws Exception** // ***GET:*** */{entity}/{id}/edit*
- **RestResult editNew() throws Exception** // ***GET:*** */{entity}/new*

注意：*不需为这 8 个内置 Rails-style REST 标准入口方法声明 @Get / @Post / @Put / @Delete REST 注解。*

很多情况下，REST 请求的 URL 格式并不一定遵循 Rails-style REST 标准，另外，有时也可能使用自定义的 REST 入口方法处理 Rails-style REST 标准请求。这种情况下，就需要在 REST 入口方法中声明 @Get / @Post / @Put / @Delete REST 注解来指定入口方法处理何种请求类型及匹配模式。

@Get / @Post / @Put / @Delete 只有一个注解参数：

- **String[] value()** ：匹配模式字符串数组（可以指定多个匹配模式）
  （默认：{""}，只有一个空字符串元素的数组）

匹配模式字符串用来匹配 REST 请求附加信息“**{request-path}**”（参考：《Convention 推导规则》），由固定字符、分隔符（“**/**”、“**,**”、“**;**”）和参数占位符（“**{int}**”）三部分组成。其中，参数占位符用于指定 REST 入口方法的参数注入顺序，编号从 0 开始，能匹配除 3 个分隔符以外的所有字符。

```
示例 1：
/* ------------------------------ REST 入口方法 ------------------------------ */
// REST 注解          ：@Post("/{2}/{0},{1}")
// REST 入口方法      ：RestResult anyRestMethod(Date date, String name, int id)
/* ---------------------------- REST 请求附加信息 ---------------------------- */
// POST {request-path}：  /123/20121103,bruce
/* ---------------------------------- 注入结果 ---------------------------------- */
// REST 入口方法      ：RestResult anyRestMethod(Date("20121103"), "bruce", 123)
```

```
示例 2：
/* ------------------------------ REST 入口方法 ------------------------------ */
// REST 注解          ：@Post("/user-{3}/{1},{2}")
// REST 入口方法      ：RestResult anyRestMethod(String name, int[] vals, int id)
/* ---------------------------- REST 请求附加信息 ---------------------------- */
// POST {request-path}：  /user-123/11|22|33,bruce
/* ---------------------------------- 注入结果 ---------------------------------- */
// REST 入口方法      ：RestResult anyRestMethod(null, int[11, 22, 33] , 0)
```

1） REST 入口方法的参数类型为数组或集合时，请求附加信息的相应字段用“|”分割各个元素（示例 2：*int[] vals*）。
2） REST 入口方法的参数没有对应的参数占位符时，注入空值：对象类型注入 null，基础类型注入 0（示例 2：*String name*）。
3） REST 入口方法的参数类型转换失败时，注入空值：对象类型注入 null，基础类型注入 0（示例 2：*int id*）。

为了避免避免产生歧义匹配，在设计 REST 请求附加信息时要注意尽量用分隔符（“/”、“,”、“;”）来分隔每个参数占位符。例如：

```
示例：
/* ------------------------------ REST 入口方法 ------------------------------ */
// REST 注解          ：@Post("/{0}")
// REST 入口方法      ：RestResult method1(String name)

// REST 注解          ：@Post("/{0}-{1}")
// REST 入口方法      ：RestResult method2(String name, int id)

/* ---------------------------- REST 请求附加信息 ---------------------------- */
// POST {request-path}：  /123-456
/* ---------------------------------- 注入结果 ---------------------------------- */
// REST 入口方法      ：（匹配 method1 还是 method2）？
```

JessMA 会先匹配参数占位符较少的方法，因此上例会匹配 *method1*，如果参数占位符之间用分隔符而不是“-”分隔则不会存在歧义。

*** 现在总结一下到目前为止已经接触到的 JessMA 的 7 种注解：

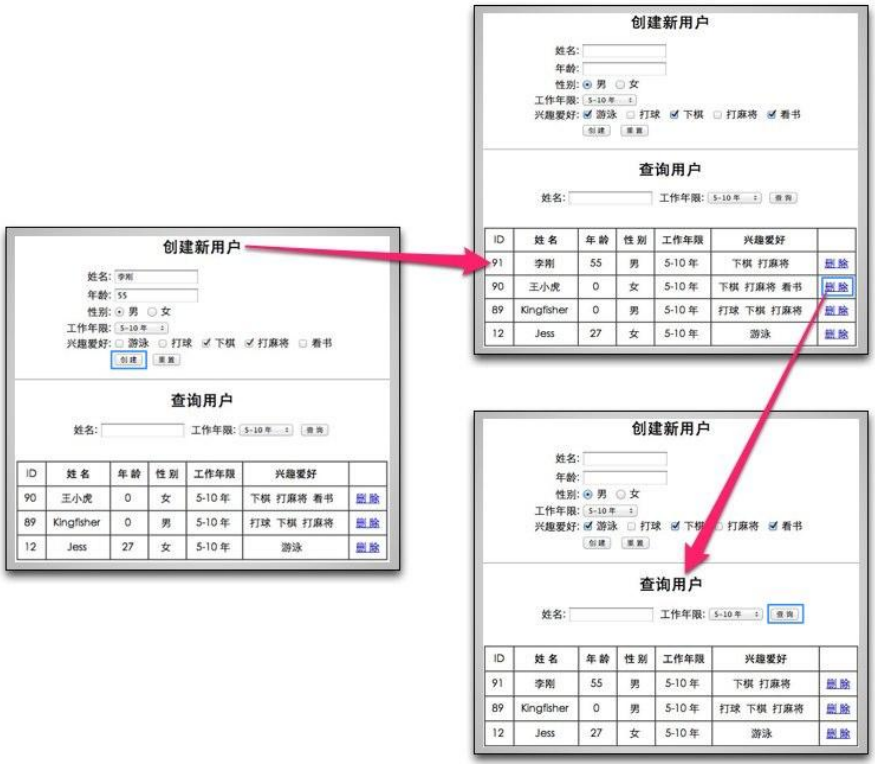
名 称	可继承	声明位置	作 用
@FormBean	是	Action 类和入口方法	注入请求参数
@DaoBean **@DaoBeans**	是	Action 类和入口方法	注入 DAO 对象
@SpringBean **@SpringBeans**	是	Action 类和入口方法	注入 Spring Bean
@Transaction	是	DAO 类和方法	声明事务属性
@Result **@Results**	是	Action 类和入口方法	定义 Action Result
@ExceptionMapping **@ExceptionMappings**	是	Action 类和入口方法	定义 Exception Mapping
@Get/@Post/@Put/@Delete	否	REST 入口方法	定义 REST 匹配模式

注意：REST 入口方法并非 Action 入口方法，REST Action 的 Action 入口方法通常为 RestActionSupport 提供的 *execute()*。因此，那些用于 Action 入口方法的注解（@FormBean / @DaoBean / @SpringBean / @Result / @ExceptionMapping）对 REST 入口方法不起作用。如果应用程序需要注入相关功能时有 3 种选择：

1、 在 Action 类中声明这些注解
2、 改写 RestActionSupport 的 execute()，在自己的 execute() 方法中声明这些注解
3、 用其它途径取代注解，如：
1） 用 createFormBean() 方法取代 @FormBean 注解
2） 用 FacadeProxy.create() 方法取代 @DaoBean
3） 用 WebApplicationContextUtils.getWebApplicationContext() 获取 Spring Context
4） 用 MVC 配置文件取代 @Result 和 @ExceptionMapping 注解

14.3 应用示例（一）

本例修改 13.5 小节中的示例，把请求模式改为 REST 风格，利用 REST Convention 和 Action Convention 机制实现“零配置”处理 REST 请求。示例界面与以前一样：



1、 加入 REST 插件和 Xstream 相关程序包

（1） 把 jessma-ext-rest-x.x.x.jar 放入应用程序的 /WEB-INF/lib 目录。

（2） JessMA 使用 XStream 输出 XML 和 JSON 视图，因此需把 Xstream 相关的 3 个 jar 包（xmlpull-1.1.3.1.jar、xstream-1.4.7.jar）放入应用程序的 /WEB-INF/lib 目录。

2、 修改 /WEB-INF/web.xml：加入 RestDispatcher

```
<!-- REST 请求过滤器 -->
<filter>
    <filter-name>RestDispatcher</filter-name>
    <filter-class>org.jessma.ext.rest.RestDispatcher</filter-class>
    <!-- REST 配置文件（可选，默认：rest-config.xml） -->
    <!--
    <init-param>
        <param-name>rest-config-file</param-name>
        <param-value>rest-config.xml</param-value>
    </init-param>
    -->
</filter>
<filter-mapping>
    <filter-name>RestDispatcher</filter-name>
    <!-- 一般设置为：REST 配置文件的 ${rest-base-path}/* -->
```

```
    <url-pattern>/*</url-pattern>
    <!-- 通常只需处理 REQUEST 请求 -->
    <dispatcher>REQUEST</dispatcher>
    <!--
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
    <dispatcher>ERROR</dispatcher>
    -->
</filter-mapping>
```

3、 在 REST 配置文件中加入 REST Convention 支持

```
<global>
    <!-- request 和 response 的默认编码（可选，默认：不设置） -->
    <encoding>UTF-8</encoding>
    <!-- Action 请求的后缀（可选，默认：'.action'） -->
    <action-suffix>.action</action-suffix>
    <!-- REST 请求的起始路径（可选，默认：'/'） -->
    <rest-base-path>/rest</rest-base-path>
    <!-- 默认的 Action 包路径（可选，默认：'/'） -->
    <default-action-path>/test/rest</default-action-path>
    <!-- 应用程序支持的视图类型（可选，默认：'html'） -->
    <!-- JessMA 支持 'html, xml, json' 三种类型 -->
    <!-- （注：第一个元素作为不带后缀名请求的默认视图类型） -->
    <support-render-types>html, xml, json</support-render-types>
</global>
```

4、 创建 Action：action.test.rest.UserAction

（参考 MyJessMA 工程源码：src/action.test.rest.UserAction）

UserAction 的 REST 入口方法匹配模式如下：

Entry Method	Rails 标准方法	匹配请求信息
index()	是	GET : ""
create()	是	POST : ""
delete(int)	是	DELETE : "/{0}"
query(String, int)	否	GET : "/{0},{1}" GET : "/{1}"

5、 创建测试页面：/jsp/test/rest/user.jsp

（参考 MyJessMA 工程源码：WebRoot/jsp/test/rest/user.jsp）

6、 修改 /jsp/index.jsp：加入测试链接

```
<li> <a href="user ">测试 REST Convention - 1</a></li>
```

14.4 应用示例（二）

本例修改上例作小小修改，在 REST 请求转换为 Action 请求时使用自定义转换规则替代上例的默认转换规则：

1、 在 REST 配置文件中加入自定义转换规则

```
<!-- REST 实体 user-2 映射到 Action：/test/my-rest-2/my-user-2.action -->
<entities action-path="/test/my-rest-2">
    <entity name="user-2" action="my-user-2"/>
</entities>
```

2、 创建 Action：action.test.myrest2.MyUser2

（参考 MyJessMA 工程源码：src/action.test.myrest2.MyUser2）

（MyUser2 的代码与上例 UserAction 完全一样）

3、 创建测试页面：/jsp/test/my-rest-2/my-user-2.jsp

（参考 MyJessMA 工程源码：WebRoot/jsp/test/my-rest-2/my-user-2.jsp）

（略：与上例 user.jsp 的代码基本一样，只需修改几个页面链接）

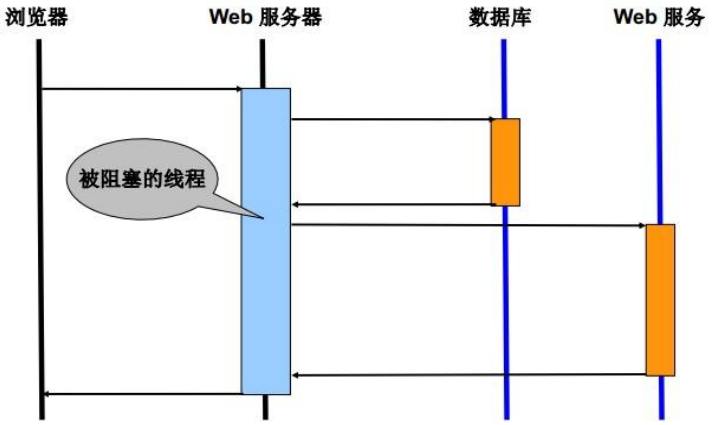
4、 修改 /jsp/index.jsp：加入测试链接

```
<li> <a href="user-2 ">测试 REST Convention - 2</a></li>
```

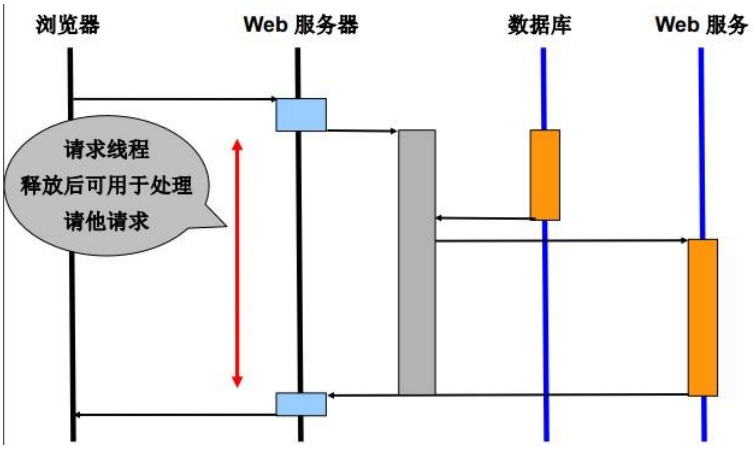
PS：可以尝试在请求地址中加上 .html、.xml 或 .json 后缀观察输出结果。

15 应用篇（十四）—— 异步 Action

异步处理是 Servlet 3.0 规范中的一个重要新特性，Servlet 3.0 之前，一个普通 Servlet 的主要工作流程大致如下：首先，Servlet 接收到请求之后，可能需要对请求携带的数据进行一些预处理；接着，调用业务接口的某些方法，以完成业务处理；最后，根据处理的结果提交响应，Servlet 线程结束。其中第二步的业务处理通常是最耗时的，这主要体现在数据库操作，以及其它的跨网络调用等，在此过程中，Servlet 线程一直处于阻塞状态，直到业务方法执行完毕。在处理业务的过程中，Servlet 资源一直被占用而得不到释放，对于并发较大的应用，这有可能造成性能的瓶颈。



现在通过使用 Servlet 3.0 的异步处理支持，之前的 Servlet 处理流程可以调整为如下的过程：首先，Servlet 接收到请求之后，可能首先需要对请求携带的数据进行一些预处理；接着，Servlet 线程将请求转交给一个异步线程来执行业务处理，Servlet 线程本身返回至容器，此时 Servlet 还没有生成响应数据，异步线程处理完业务以后，可以直接生成响应数据（异步线程拥有 ServletRequest 和 ServletResponse 对象的引用），或者将请求继续转发给其它 Servlet。如此一来，Servlet 线程不再是一直处于阻塞状态以等待业务逻辑的处理，而是启动异步线程之后可以立即返回。



由此可知，Servlet 3.0 异步处理的主要优点是提高了服务器接收请求的能力。请求处理线程把请求提交到后台线程后立刻返回并继续接收其它请求（服务器第一次响应）；当后台线程处理完请求后把最终结果返回到客户端（服务器第二次响应）。通过这种方式，请求处理线程在单位时间内能接收更多的请求，但单个请求的处理时间没有变少，客户端也要等到服务器第二次响应时才能完成请求，客户端的等待时间也没有变少。

默认情况下 Servlet 和 Filter 都不开启异步处理功能。开启异步处理功能的方法请参考相关资料（如：《Servlet 3.0 新特性详解》）。另外要注意：**要使某个请求支持异步处理，该请求所进入的所有 Filter 和 Servlet 都必须开启异步处理功能**。如：一个请求 R 发给 Servlet S 处理，而 R 在进入 S 前被两个 Filter F1 和 F2 率先拦截过滤。如果想在 S 中异步处理 R，那么 S、F1 和 F2 都必须开启异步处理功能。

另外，要使用异步处理功能需得到执行环境的支持（如：JavaEE 6 / Tomcat 7.0），否则当调用异步处理相关方法时会收到“ClassNotFound”或“not support”等异常。

15.1 常规 Action 的异步处理

JessMA 的 MVC 子框架提供了异步处理支持，从而让 JessMA 应用程序可以方便地使用异步处理功能。在 Action 处理方法中，通过调用 ActionSupport 提供的 ***startAsync(...)*** 方法即可启动异步处理。***startAsync(...)*** 有多个重载方法，这些重载方法拥有下列一个或多个参数：

```
void startAsync( AsyncTask task, long timeout,
                 AsyncTaskLauncher launcher, AsyncListener ... listeners    )
```

- ***task***：实现了 ***AsyncTask*** 接口的异步任务对象。**对于常规 Action，*task* 参数必须是 *ActionTask* 抽象类的子类**，*ActionTask* 提供了异步任务入口抽象方法 ***String run()***，*task* 对象通过实现该方法来执行异步任务，如果一切正常（没有发生超时或其它错误事件），则以该方法的返回值作为请求的最终处理结果。

- ***timeout***：异步任务超时值（毫秒），如果超过了指定时间任务还没执行完毕，则以 **Action.TIMEOUT**（"***timeout***"）作为请求的最终处理结果，并立刻返回到客户端。注意：此时 *task* 的 ***run()***方法仍然在继续执行，但会忽略它的返回值。因此，在实际应用环境中当发生超时的情形下，应用程序应当采取措施终止 *task* 的 ***run()***方法的执行。*timeout* 的取值意义如下：

  - **小于 0**：使用 Web 容器的默认超时值
  - **等于 0**：永不超时
  - **大于 0**：使用设定的超时值

- ***launcher***：异步任务启动器，如果为 ***null*** 则使用容器内置的线程池启动异步任务。通常情况下，应用程序不必设定自己的异步任务启动器。但在某些特定场合，应用程序可能希望用自定义线程或线程池来启动异步任务以获得更多的控制权，此时，需要创建一个实现了 ***AsyncTaskLauncher*** 接口的对象，并在该对象所实现的接口方

法 ***void start(Runnable taskRunner)*** 中用自定义线程或线程池来启动异步任务。

- ***listeners***：异步任务监听器（可选，可设置多个），用于监听异步任务的执行状态通知事件。***AsyncListener*** 提供了以下事件通知：

  - ***void onStartAsync(event)*** ：启动异步任务（容器不一定提供该事件通知）
  - ***void onComplete(event)*** ：异步任务执行完毕
  - ***void onTimeout(event)*** ：异步任务执行超时
  - ***void onError(event)*** ：异步任务执行出错

根据异步处理的特性可知，Action 在处理异步请求时会发生两次响应：

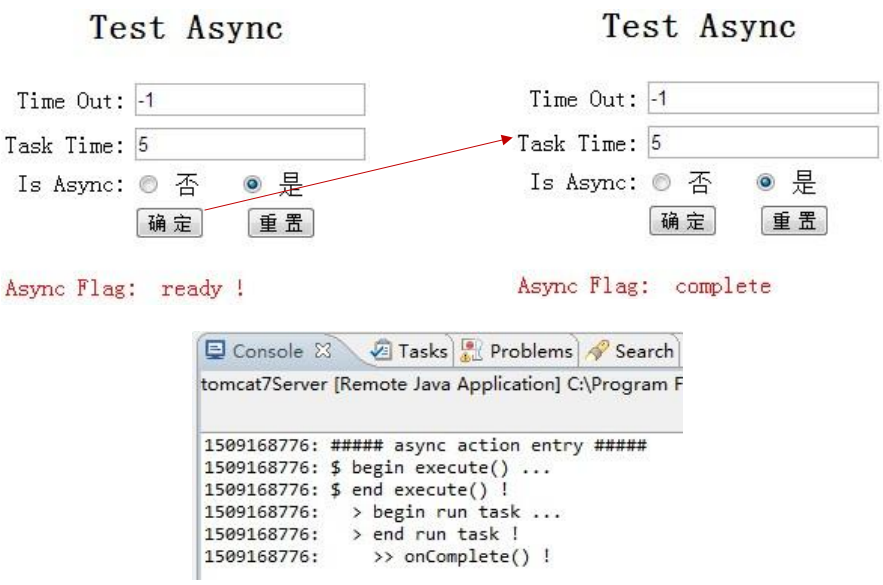
- **第一次响应**：Actin 处理方法在调用了 ***startAsync(...)*** 后，Actin 处理方法的工作已完成并退出。此时，Actin 处理方法应该返回 **Action.NONE**（"***none***"），指示 MVC 框架不要执行任何处理工作。
- **第二次响应**：异步任务执行完毕、超时或出错时，MVC 框架将会接收到最终的执行结果，应用程序应该根据实际情况处理其中一种或多种结果：

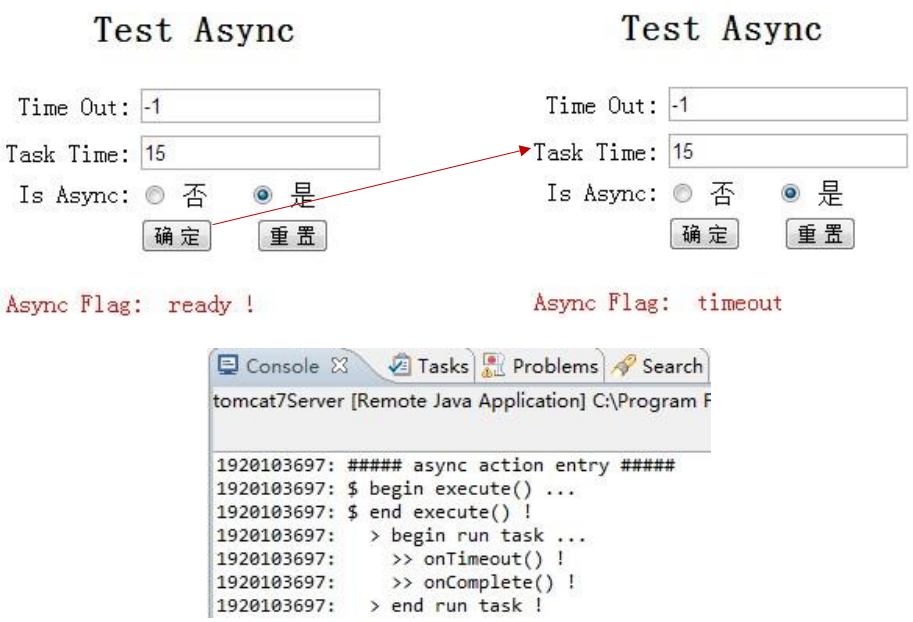
  - **任务执行完毕**：*task* 的 ***run()***方法的返回值
  - **任务执行超时**：**ActionSupport. ASYNC_TIMEOUT** （实际值："***timeout***"）
  - **任务执行出错**：**ActionSupport. ASYNC_ERROR** （实际值："***error***"）

注：应用程序可以改写 *getAsyncTimeoutResult()* / *getAsyncErrorResult()* 来更改任务执行超时或出错时的结果值。

15.2 应用示例（一）

下面通过一个简单的示例来演示一下常规 Action 的异步处理方式，示例界面如下：





1、 修改 /WEB-INF/web.xml：所有 Filter 都开启异步处理支持

1）修改 web.xml 的 Schema，设置为 Servlet 3.0 版本

```
<web-app version="3.0"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
```

2）对所有 Filter 都增加以下配置项

```
<async-supported>true</async-supported>
```

2、 创建 Action：action.test.async.TestAsync

（参考 MyJessMA 工程源码：src/action.test.async.TestAsync）

3、 创建测试页面：/jsp/test/async/test-async.jsp

（参考 MyJessMA 工程源码：WebRoot/jsp/test/async/test-async.jsp）

4、 修改 /jsp/index.jsp：加入测试链接

```
<li> <a href="test/async/test-async.action">测试异步 Action（Servlet 3.0）</a></li>
```

至此，测试示例已创建完毕，由于本例采用了 Action Convention 机制自动识别 Action，

因此不用在 MVC 配置文件中定义 Action，现在请尝试用不同的参数运行示例，观测页面结果和控制台的输出信息。

15.3 REST Action 的异步处理

REST Action 的异步处理方式与常规 Action 类似，都是通过调用 ActionSupport 提供的 ***startAsync(...)*** 方法启动异步处理。主要区别是 REST Action 用 REST 入口方法而非 Action 入口方法处理请求，REST 入口方法的返回值类型是 RestResult 而非 String。因此，在 REST Action 中处理异步请求时需要作相应调整。

- **在 REST Action 中，*startAsync(...)* 的 *AsyncTask task* 参数必须是 *RestActionTask* 抽象类的子类，*RestActionTask* 提供了异步任务入口抽象方法 *RestReslt run()*，*task* 对象通过实现该方法来执行异步任务，如果一切正常（没有发生超时或其它错误事件），则以该方法的返回值作为请求的最终处理结果。**

- **REST Action 在处理异步请求时同样会发生两次响应：**

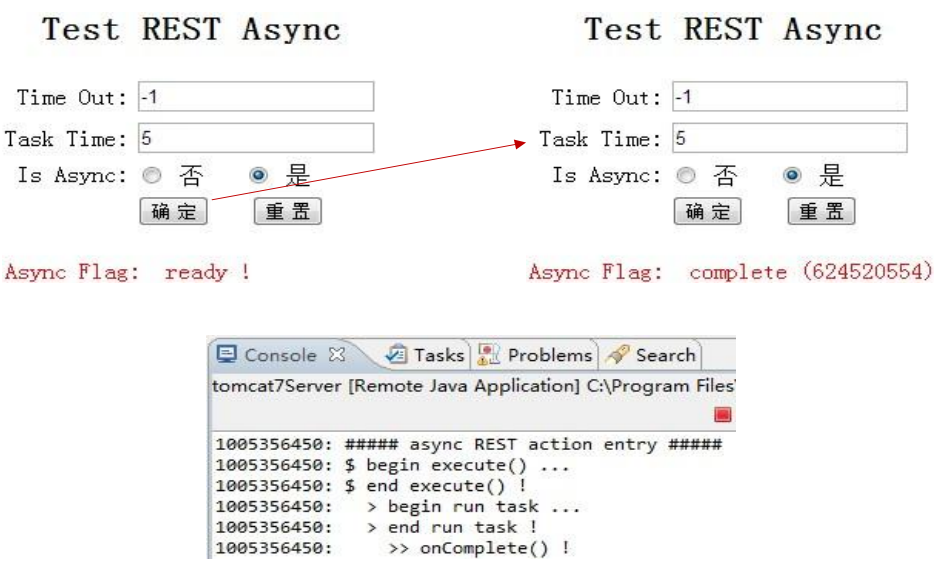
  - **第一次响应：**REST Actin 处理方法在调用了 ***startAsync(...)*** 后，REST Actin 处理方法的工作已完成并退出。此时，REST Actin 处理方法应该返回 **RestActionSupport.REST_NONE**（RestActionSupport 中定义的一个特殊 REST Result），指示 MVC 框架不要执行任何处理工作。
  - **第二次响应：**异步任务执行完毕、超时或出错时，MVC 框架将会接收到最终的执行结果，应用程序应该根据实际情况处理其中一种或多种结果：

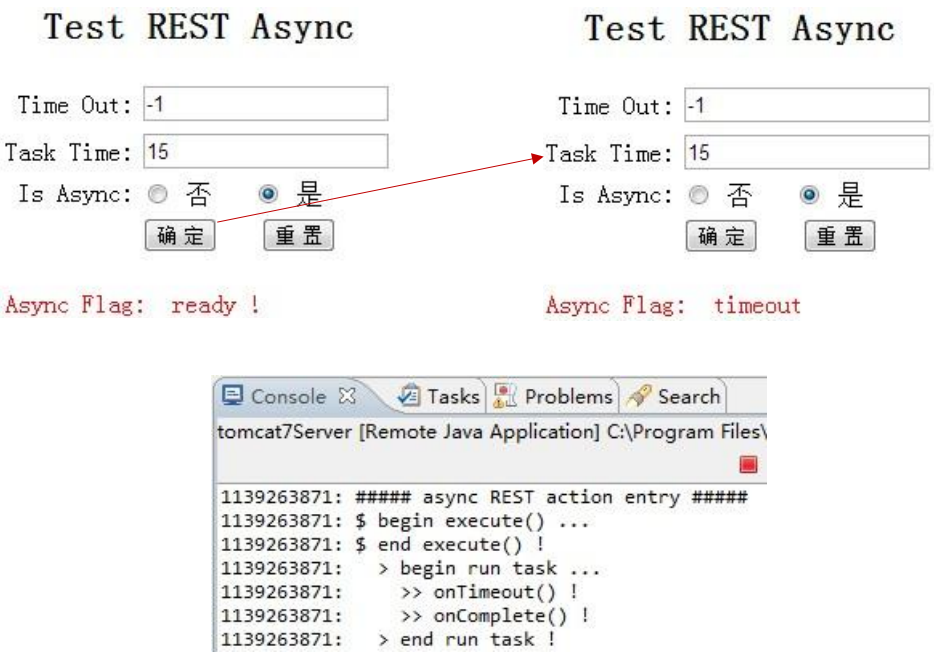
    - **任务执行完毕：***task* 的 ***run()***方法的返回值
    - **任务执行超时：RestActionSupport.REST_ASYNC_TIMEOUT**
      （实际值：***RestResult("timeout", null)***）
    - **任务执行出错：RestActionSupport.REST_ASYNC_ERROR**
      （实际值：***RestResult("error", null)***）

注：应用程序可以改写 *getAsyncTimeoutResult()* / *getAsyncErrorResult()* 来更改任务执行超时或出错时的结果值。

15.4 应用示例（二）

本例修改 15.2 小节中的示例，把请求模式改为 REST 风格，用 REST Action 取代常规 Action，示例界面如下：





1、 创建 REST Action：action.test.async.TestRest

（参考 MyJessMA 工程源码：src/action.test.async.TestRest）

2、 创建测试页面：/jsp/test/async/test-rest.jsp

（参考 MyJessMA 工程源码：WebRoot/jsp/test/async/test-rest.jsp）

3、 修改 rest-config.xml：加入 REST 映射

由于 Action“/test/async/test-rest”与当前的默认 REST Convention 映射配置不一致，因此需要在 REST 配置文件中手工指定它的 REST 映射：

```
<entities action-path="/test/async/">
    <entity name="test-rest"/>
</entities>
```

4、 修改 /jsp/index.jsp：加入测试链接

```
<li> <a href="test-rest/-1/0/true">测试异步 REST Action（Servlet 3.0）</a></li>
```

至此，测试示例已创建完毕，现在请尝试用不同的参数运行示例，观测页面结果和控制台的输出信息。

16 应用篇（十五）—— 动态更新配置

无论在开发调试环境下还是在生产环境下，支持动态更新应用程序配置都是一个非常实用的功能，避免了由于过多的重启应用服务器操作所导致的一系列副作用，如：需要登录到应用服务器的主机执行重启操作；重启应用服务器需要较大的时间成本；还会中断同一应用服务器下的其他应用程序等。JessMA 支持动态更新下列应用程序配置：

- **MVC 配置**（MVC 主配置文件 *mvc-config.xml* 及其从配置文件）
- **REST 配置**（REST 配置文件 *rest-config.xml*）
- **用户自定义配置**（应用程序配置文件 *app-config.xml* 的 <user> 节点）
- **国际化资源文件**（默认：*res.application-message_xx_YY.properties*）

在更新配置时，为避免出现数据不一致的情形，应根据实际情况处理下列问题：

✧ 确保在执行更新前，当前所有正在处理的请求的都已执行完毕
✧ 确保在执行更新时，暂停接收所有新请求，更新完毕后恢复接收

16.1 更新 MVC 配置

MVC 前端控制器 org.jessma.mvc.**ActionDispatcher** 提供以下方法动态更新 MVC 配置：

void reload(long delay) throws Exception

- **描述**：重新加载 MVC 配置
- **参数** ***delay***：执行更新操作的延时时间（毫秒），指定一个延时时间是为了确保所有当前正在处理的请求都执行完毕后才执行重新加载操作。
- **异常** ***Exception***：加载失败抛出异常，并恢复原来的配置

void pause()

- **描述**：暂停接收 HTTP 请求。在暂停状态下，对所有请求直接返回 HTTP 503 错误（通常在执行 *reload(long)* 前调用）。

void resume()

- **描述**：恢复接收 HTTP 请求（通常在执行 *reload(long)* 后调用）。

以上三个方法配合使用：***pause()*** -> ***reload(long delay)*** --> ***resume()***

16.2 更新 REST 配置

REST 过滤器 org.jessma.ext.rest.**RestDispatcher** 提供以下方法动态更新 REST 配置：

void reload(long delay) throws Exception

- **描述**：重新加载 REST 配置
- **参数** ***delay***：执行更新操作的延时时间（毫秒），指定一个延时时间是为了确保所有当前正在处理的请求都执行完毕后才执行重新加载操作。
- **异常** ***Exception***：加载失败抛出异常，并恢复原来的配置

void pause()

- **描述**：暂停接收 HTTP 请求。在暂停状态下，对所有请求直接返回 HTTP 503 错误（通常在执行 *reload(long)* 前调用）。

void resume()

- **描述**：恢复接收 HTTP 请求（通常在执行 *reload(long)* 后调用）。

以上三个方法配合使用：***pause()*** -> ***reload(long delay)*** --> ***resume()***

16.3 更新用户自定义配置

应用程序配置类 org.jessma.app.**AppConfig** 提供以下方法动态更新用户自定义配置：

void reloadUserConfig(long delay) throws Exception

- **描述**：重新加载用户自定义配置
- **参数** ***task***：执行更新操作的延时时间（毫秒），指定一个延时时间是为了确保所有当前正在处理的请求都执行完毕后才执行重新加载操作。
- **异常** ***Exception***：加载失败抛出异常

注意：

- 本方法会再次调用由 app-config.xml 的 <system>/<user-config-parser> 节点定义的 UserConfigParser 的 *parse(Element)* 方法重新加载 <user> 节点
- 本方法只重新加载 <user> 节点，不重新加载 <system> 节点， 因此如果更改了 <system> 节点的配置信息，必须重启应用程序才能使更改生效

16.4 更新国际化资源文件

国际化资源文件通过 java.util.**ResouceBundle** 加载并缓存，因此只需调用 ResourceBuldle

的 ***clearCache()*** 方法清除缓存，即可更新国际化资源文件。另外，更新资源文件不需要暂停应用程序服务。

16.5 应用示例

本示例在首页中加入若干个链接展示动态更新操作的效果，示例界面如下：

1、 修改 /jsp/index.jsp：加入测试链接

```
<div align="right" style="font-size: xx-small;">
        <a href="test/reload-cfg.action?type=mvc">更新 MVC 配置</a> 
        <a href="test/reload-cfg.action?type=rest">更新 REST 配置</a> 
        <a href="test/reload-cfg.action?type=user">更新用户配置</a> 
        <a href="test/reload-cfg.action?type=res">更新资源文件</a> 
        <a href="test/reload-cfg.action?type=all">更新所有配置</a> 
</div>
```

2、 创建 Action：action.test.ReloadCfg

（参考 MyJessMA 工程源码：src/action.test.ReloadCfg）

17 应用篇（十六）—— 公共组件

公共组件位于 **org.jessma.util** 包及其子包中，主要提供与应用无关的基础帮助类，各组件的具体使用方法说明请参考 API 文档，这里只对主要组件作简单介绍。

17.1 org.jessma.util

类 名	功 能	主要方法
BeanHelper	动态创建和装配 Java Bean	✓ createBean：动态创建 Bean ✓ setProperties：装配 Bean 属性 ✓ setFieldValues：装配 Bean 成员变量
CoupleKey	两个元素组成的 Key	✓ get/setKey1：获取和设置第一个 Key 元素 ✓ get/setKey2：获取和设置第二个 Key 元素
CryptHelper	提供各种加解密方法	✓ urlEncode：URL 编码 ✓ urlDecode：URL 解码 ✓ aesEncrypt：AES 加密 ✓ aesDecrypt：AES 解密 ✓ desEncrypt：DES 加密 ✓ desDecrypt：DES 解密 ✓ md5：MD5 加密 ✓ sha：SHA 加密
GeneralHelper	提供常用帮助方法	✓ str2Xxx：字符串转换为其它类型 ✓ isStrXxx：字符串检查 ✓ getClassResourceXxx：获取应用资源
KV	封装键-值对	✓ getter & setter：获取和设置键/值
LogUtil	slf4j 工具类	✓ getJessMALogger(...)：获取 JessMA Logger ✓ getLogger(...)：获取其它 Logger ✓ exception(...)：输出异常日志 ✓ fail(...)：输出操作失败日志
LStrMap	键为小写字母的 Map	与 Map 接口一致
LStrSet	元素为小写字母的 Set	与 Set 接口一致
PackageHelper	提供包资源访问方法	✓ getClasses：获取包中的类 ✓ getResourceNames：获取包中的资源名称 ✓ getPackages：获取包的子包
PageSplitter	封装分页算法	✓ getCurrentPage：获取当前页 ✓ setCurrentPage：设置当前页 ✓ getCurrentRange：获取当前页的索引范围 ✓ getPageCount：获取页数 ✓ getPageSize：获取分页大小 ✓ setPageSize：设置分页大小 ✓ nextPage：转到下一页

		✓ prePage：转到上一页
Pagination	对 List 对象分页	✓ getCurrentPage：获取当前页 ✓ setCurrentPage：设置当前页 ✓ getCurrentRange：获取当前页的索引范围 ✓ getPageCount：获取页数 ✓ getPageSize：获取分页大小 ✓ setPageSize：设置分页大小 ✓ nextPage：转到下一页 ✓ prePage：转到上一页 ✓ getCurrentList：获取当前页的元素 ✓ getList：获取这个 List
Range	封装数值范围	✓ get/setBegin：获取和设置起始索引 ✓ get/setEnd：获取和设置结束索引
Result	封装操作结果返回值	✓ get/setFlag：获取和设置返回标识 ✓ get/setValue：获取和设置返回值
Pair	封装操值对	✓ get/setFirst：获取和设置第一个值 ✓ get/setSecond：获取和设置第二个值
UStrMap	键为大写字母的 Map	与 Map 接口一致
UStrSet	元素为大写字母的 Set	与 Set 接口一致

17.2 org.jessma.util.archive

类 名	功 能	主要方法
Zipper	Zip 压缩	✓ execute：执行压缩
UnZipper	Zip 解压	✓ execute：执行解压
Tarrer	Tar 归档	✓ execute：执行归档
UnTarrer	Tar 解档	✓ execute：执行解档
BZipper	BZip 压缩	✓ execute：执行压缩
UnBZipper	BZip 解压	✓ execute：执行解压
GZipper	GZip 压缩	✓ execute：执行压缩
UnGZipper	GZip 解压	✓ execute：执行解压

上述几种压缩（归档）、解压（解档）组件的使用方法类似，它们内部使用 ***org.apache.tools.ant.Task*** 执行压缩和解压任务。因此，使用这些组件时应用程序需加入 **ant.jar**。下面以 Zipper / UnZipper 为例展示组件的使用方法：

（参考 MyJessMA 工程源码：src/test.TestZip）

17.3 org.jessma.util.http

类 名	功 能	主要方法
HttpHelper	提供 HTTP 交互操作方法	✓ getHttpConnection：创建 HTTP 请求

		✓ writeBytes/String：发送 HTTP 数据 ✓ readBytes/String：读取 HTTP 数据 ✓ getXxxParam：获取请求参数
FileUploader	文件上传	参考：《文件上传》
FileDownloader	文件下载	参考：《文件下载》

33. HttpHelper

HttpHelper 提供了很多操作 Request、Response、Session 和 ServletContext 等服务端对象的帮助方法，也提供一些 HTTP 客户端方法。参考以下示例：

（参考 MyJessMA 工程源码：src/test.TestHttpHelper）

17.4 org.jessma.util.mail

类 名	功 能	主要方法
MailSender	邮件发送客户端	✓ send：发送邮件

34. MailSender

MailSender 的使用方法非常简单，首先创建 MailSender 实例，然后设置实例属性，最后调用 ***send()*** 方法发送邮件。参考以下示例：

（参考 MyJessMA 工程源码：src/test.TestMail）