

## 第十二章 多模块开发与Struts扩展

多模块开发和软件扩展是现代软件开发过程中最重要的理念。对于一个框架型软件来说，能否进行多模块开发、能否根据需要进行扩展、能否与其他组件无缝合作是衡量一个优秀框架的重要因素。优秀的框架应具有良好的扩展性和协作性，Struts 框架也不例外。Struts 框架为开发人员提供了多模块开发的方法以及多个扩展点，本章将对这些内容进行介绍。

### 12.1 多模块开发

对于一些大型的 Web 应用，通常会分为几个模块，如用户管理模块，商品管理模块。如果设计得当，这些模块间就可以同时并行开发，大幅提高开发进度。模块化开发是现在大中型应用程序开发的流行选择。

并行开发的一个最大的问题就是资源访问冲突，如果处置不当，反而会影响开发效率。Struts 的配置文件 `struts-config.xml` 是 Struts 框架最重要的资源之一，并且是需要频繁改动的。如果并行开发的各个团队都是用这一个配置文件，势必造成访问冲突。Struts 框架的模块化机制就是专门应对这种情况的。

Struts 从 1.1 版本开始增加了模块化支持，并且一直在强化对模块化的支持。不同的应用模块可以拥有各自的 `struts-config` 配置文件、消息资源、Validator 框架配置文件。不同的模块可以协同开发，互不影响。

模块在 Struts 中的英文术语是 `module`，一些与模块化相关的组件的名称中都包含有 `module` 这样的字眼，如模块配置类为 `ModuleConfig`，模块实用工具类 `ModuleUtils` 等。

如果要将 Struts 应用配置为多模块应用，需要如下三个步骤：

- 为每个模块分别建立一个 `struts` 配置文件；
- 通知模块控制器；
- 使用特定的 Action 在模块间跳转。

下面分别对这个三步骤进行详细介绍，以掌握如何实现多模块的 Struts 应用程序。

#### 12.1.1 多模块Struts应用配置

Struts 应用程序是通过配置文件组织资源的，对于多模块应用，每个模块也是通过各自的配置文件组织各自的资源。当只有一个模块时，即只有默认模块，通常默认模块的配置文件名为 `struts-config.xml`。其他模块的命名方式一般为 `struts-config-模块名.xml`。如用户管理模块的配置文件可命名为 `struts-config-usermanage.xml`，商品管理模块的配置文件名为 `struts-config-productmanage.xml`。当然，这样的文件名并不是必须的，但按照这种方式给文件命名从配置文件名本身就可以看出对应的模块和意义。

在多模块环境中，每个模块都有各自独立的配置文件，十分有利于多个小组的协同开发。在小组协同开发的环境中，通常是一个小组负责一个开发模块。Struts 的多模块机制可以有效地避免多个小组协同开发中的资源访问冲突，因为每个模块的资源都由各自的配置文件组织，绝大部分的修改都限于本模块内。

### 12.1.2 通知控制器

通知控制器即是將多模块的配置信息注册到控制器中去。在单模块 Struts 应用中，只需要把默认的 Struts 配置文件注册到控制器，这是通过将默认配置文件作为 ActionServlet 类的一个初始化参数实现的。对于多模块的情况，配置的示例代码片段如下（该代码片段来自 web.xml）。

#### 代码 12.1

```
<init-param>
  <param-name>config</param-name>
  <param-value>/WEB-INF/struts-config.xml</param-value>
</init-param>
<init-param>
  <param-name>config/module1</param-name>
  <param-value>/WEB-INF/struts-config-module1.xml</param-value>
</init-param>
```

在这段配置中，配置了两个应用模块：默认模块和名为 module1 的模块。对于默认模块，配置文件对应的 ActionServlet 初始化参数名为 config。对于其他模块，ActionServlet 初始化参数的命名原则是“config/模块名”。如上面的代码示例中，module1 模块的配置文件对应的初始化参数为 config/module1。其中前缀“config/”是不能缺少的，后面跟模块名。在 Struts 控制器中，是通过模块名来区分不同模块的。在资源访问中，也是一模块名作为前缀来区分对不同模块的访问。如以“/module1”开头的路径会告诉控制器所要访问的将是 module1 模块的资源。

### 12.1.3 在模块间转换

在模块间跳转不同于模块内部的跳转，如果需要跳转到其他模块，在连接中必须明确指出。进行模块跳转的 URL 必须给定两个参数：prefix 和 page。其中参数 prefix 指明要跳转到的模块前缀，其值为以“/”开头的模块名。如前面配置片段中的 module1，其前缀 prefix 的值就是/module1。page 指明要跳转的页面或其他资源。

模块间的跳转之所以不同于一般的页面调转，是因为模块间的跳转涉及到资源的转换，包括消息资源和其他可配置资源。有 3 种方法可以实现模块间跳转。

#### 1. 使用 Struts 内建的 SwitchAction 类

SwitchAction 类是 Struts 内建的最有用的 Action 类之一，是专门为实现页面调转而设计的。在 SwitchAction 类内部，自动实现了消息资源和模块前缀的转换等操作。直接使用 SwitchAction 类只需要在 Struts 配置文件中声明即可，声明使用 SwitchAction 类的配置片段如下。

#### 代码 12.2

```
.....
<action-mappings>
  <action
```

```

        path="/toModule"
        type="org.apache.struts.actions.SwitchAction"/>
        .....
    </action-mappings>

```

其中 path="/toModule"指明了该 Action 类的访问路径。如果要从当前模块跳转到另一模块 moduleB，则该链接的形式为：

```
http://localhost:8080/xxx/toModule.do?prefix=/moduleB&page=/index.do
```

如果要调转到的模块是默认模块，默认模块的模块前缀是空串，链接的形式为：

```
http://localhost:8080/xxx/toModule.do?prefix=&page=/index.do
```

## 2. 使用转发

可以在全局或局部转发中显式地将转发配置为跨模块的转发。配置全局转发为跨模块转发的示例代码如下：

### 代码 12.3

```

<global-forwards>
  <forward
    name="toModuleB"
    contextRelative="true"
    path="/moduleB/index.do"
    redirect="true"/>
    .....
</global-forwards>

```

其中 contextRelative 属性设为 true 时表示当前 path 属性以/开头时，给出的是相对于当前上下文的 URL。

也可以把局部转发配置为跨模块转发，如代码 12.4 所示。

### 代码 12.4

```

<action-mappings>
  <action ... >
    <forward name="success" contextRelative="true"
path="/moduleB/index.do" redirect="true"/>
  </action>
  .....
</action-mappings>

```

## 3. 使用<html:link>标记

<html:link>是 Struts 自定义标记，对超链接的行为进行了定制和封装。利用<html:link>标记可以直接将超链接声明为跨模块的跳转，使用方法为：

```
<html:link module="/moduleB" path="/index.do"/>
```

## 12.2 使用定制的控制器

Struts 在控制器层提供了多种扩展点，如扩展 ActionServlet、扩展 RequestProcessor 类、扩展 Action 类和 ActionForm 类等。下面将详细介绍如何在这些扩展点上实现扩展。

### 12.2.1 使用自定义的ActionServlet

在 Struts 的早期版本中，ActionServlet 类承担了除 Action 类以外大部分的控制器能，Struts 一般都需要扩展 ActionServlet 类，来实现各种自定义的控制功能。在 Struts 1.1 之后，大多数执行实际功能调用的操作都被迁移到 RequestProcessor 类中执行，ActionServlet 类只是调用 RequestProcessor 对象的 process()方法处理用户请求，如代码 12.5 所示。

代码 12.5

```
public class ActionServlet extends HttpServlet {
    .....
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws IOException, ServletException {
        //调用process()方法处理Get请求
        process(request, response);
    }
    public void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws IOException, ServletException {
        //调用process()方法处理Post请求
        process(request, response);
    }
    .....
    //在process()方法中，调用RequestProcessor对象的process()方法处理请求
    protected void process(HttpServletRequest request,
        HttpServletResponse response)
        throws IOException, ServletException {

        //配置请求的模块信息
        ModuleUtils.getInstance().selectModule(request,
        getServletContext());
        ModuleConfig config = getModuleConfig(request);
        //获取请求对应的RequestProcessor实例
        RequestProcessor processor = getProcessorForModule(config);
        if (processor == null) {
            processor = getRequestProcessor(config);
        }
        //调用RequestProcessor对象的process()方法处理请求
        processor.process(request, response);
    }
    .....
}
```

因此，如果需要在处理机制和功能上扩展 Struts 框架，扩展 ActionServlet 类已经没有必要。但是，处理一些特殊的需求时，ActionServlet 类仍然具有扩展的意义。

在 Struts 控制器组件一章已经介绍过，Struts 应用程序启动时在 ActionServlet 类的 init()方法中将初始化 Struts 框架。如果想改变框架包的初始化行为，就可以通过扩展 ActionServlet 类实现。扩展 ActionServlet 类，可以创建一个 org.apache.struts.action.ActionServlet 类的子类，

然后复写 `init()` 方法，实现自定义的 Struts 框架初始化过程。

一旦扩展了 `ActionServlet` 类，就需要在部署描述文件 `web.xml` 文件配置中心 `Servlet` 为扩展后的 `Servlet`。

从 Struts 1.1 开始，绝大部分的请求处理行为已经交给 `RequestProcessor` 类，如要扩展 Struts 框架的请求处理方式，可以通过扩展 `RequestProcessor` 类完成，这将在下一小节中介绍。

### 12.2.2 使用自定义的 `RequestProcessor`

如果要改变 Struts 框架处理请求的过程，扩展 `RequestProcessor` 类是一个合适的选择。如果扩展了 `RequestProcessor` 类，需要通知 Struts 框架使用自定义的请求处理器。在 Struts 配置文件中 `controller` 元素用于配置请求处理器信息。从配置文件一章中我们知道，`controller` 元素位于 `action-mappings` 元素和 `message-resources` 元素之间。如果配置文件中还没有任何 `controller` 元素的配置信息，则意味着当前的 Struts 应用正在使用默认的 `RequestProcessor` 类及默认的 `controller` 属性。`controller` 元素的配置如下。

#### 代码 12.6

```
<controller nocache="true"
    inputForward="true"
    maxFileSize="2M"
    contentType="text/html; charset=UTF-8"
    processorClass=
        "org.digitstore.web.struts.CustomRequestProcessor" />
```

`controller` 元素的 `processorClass` 属性指定了请求处理类，默认时即为 `org.apache.struts.action.RequestProcessor` 类。Struts 框架在启动时将创建一个该类的实例，并用这个实例处理应用程序的所有请求。由于每个子应用模块都可以有独立的 Struts 配置文件，因此可以为每个子应用模块配置不同的 `RequestProcessor` 类。

`RequestProcessor` 类的一个典型的扩展点就是 `processPreprocess()` 方法。顾名思义，`processPreprocess()` 的意思就是在处理请求前的一些预处理操作。在默认的 `RequestProcessor` 类中，该方法不执行任何操作，直接返回 `true`。`processPreprocess()` 方法在默认 `RequestProcessor` 类中的实现如下。

```
protected boolean processPreprocess(HttpServletRequest request,
    HttpServletResponse response) {
    return (true); // 在默认的实现中什么操作也不做，直接返回 true
}
```

`RequestProcessor` 类在 `process()` 方法中处理请求，对 `processPreprocess()` 方法也存在该方法的过程中。代码 12.7 为 `processPreprocess()` 方法在 `process()` 方法中的调用情况。

#### 代码 12.7

```
public void process(HttpServletRequest request,
    HttpServletResponse response)
    throws IOException, ServletException {
    .....
}
```

```

// 调用processPreprocess方法,如果返回false,则终止请求处理
if (!processPreprocess(request, response)) {
    return;
}
.....
// 构建ActionForm
ActionForm form = processActionForm(request, response, mapping);
processPopulate(request, response, form, mapping);
if (!processValidate(request, response, form, mapping)) {
    return;
}
.....
// 构建请求对应的Action
Action action = processActionCreate(request, response, mapping);
if (action == null) {
    return;
}
// 调用Action执行请求逻辑
ActionForward forward =
processActionPerform(request, response, action, form, mapping);
.....
}

```

从代码 12.7 可以看到, process()方法在调用 processActionForm()方法构建 ActionForm 之前(当然也在调用 Action 类的 execute()方法之前)就调用 processPreprocess()方法。默认时该方法返回 true,使得处理流程继续前进。如果该方法返回 false, process()方法就将停止请求处理的执行,并从 doGet()或 doPost()方法中返回。

在实现自定义的 RequestProcessor 类时,可以覆盖 processPreprocess()方法来实现一些特定的逻辑。例如,可以在 processPreprocess()方法中实现用户验证:如果用户已经登录,则按照正常的流程处理请求;如果用户没有登录或 session 已过期,则把请求重定向到登录页面。实现这段逻辑的 processPreprocess()方法的代码如下。

### 代码 12.8

```

public class CustomRequestProcessor
    extends RequestProcessor {
    //自定义的processPreprocess()方法,检查用户是否已经登录
    protected boolean processPreprocess (
        HttpServletRequest request,
        HttpServletResponse response) {
        HttpSession session = request.getSession(false);

//如果用户请求的是登录页面则不需要检查
        if( request.getServletPath().equals("/loginInput.do")
            || request.getServletPath().equals("/login.do") )
            return true;
        //检查session中是否存在userName属性,如果存在则表示拥护已经登录
        if( session != null &&
            session.getAttribute("userName") != null)

```

```

        return true;
    else{
        try{
            //用户未登录则重定向到登录页面
            request.getRequestDispatcher
                ("/Login.jsp").forward(request,response);
        }catch(Exception ex){
        }
    }
    return false;
}
//自定义的processContent ()方法,用于设置响应类型
protected void processContent(HttpServletRequest request,
    HttpServletResponse response) {
    //检查用户是否请求ContactImageAction,如果是则设置
    // contentType为image/gif
    if( request.getServletPath().equals("/contactimage.do")){
        response.setContentType(" image/gif");
        return;
    }
    super.processContent(request, response);
}
}
}

```

processPreprocess()方法能够截获所有 Struts 请求,达到对用户进行验证的效果。当然 RequestProcessor 类中并非只有这一个扩展点。如果想按照自己的方式处理请求,可以根据需要扩展其他方法。

### 12.2.3 使用自定义的Action类

Action 类是 Struts 框架中应用最广泛的扩展点。一般情况下,其他的 Action 类都是直接扩展 org.apache.struts.action.Action 类来实现控制逻辑。在实现具体的 Struts 应用时,许多 Action 类都要执行一些公共的逻辑,如 session 验证、初始化 session 变量、错误处理等以及一些通用的方法等。这时就可以创建一个 Action 基类,在 Action 基类中定义应用中所有 Action 的一些公共逻辑,其他具体的 Action 都扩展这个 Action 基类,而不是直接扩展 org.apache.struts.action.Action 类。当然这个 Action 基类扩展 Struts 的 Action 基类或其他扩展 Struts 的 Action 基类的 Action。这种处理方式可以提高代码的可重用性,减少代码冗余。代码 12.9 为一个 Action 基类的代码示例,在该 Action 基类中实现了一些 Action 的公共逻辑。该 Action 基类扩展了 Struts 框架内建的 LookupDispatchAction 类。

#### 代码 12.9

```

import java.util.Date;
import java.util.Enumeration;
.....
// BaseAction扩展Struts内建的LookupDispatchAction
public class BaseAction extends LookupDispatchAction {
    //日志工具

```

```

protected transient final Log log = LoggerFactory.getLog(getClass());
private static final String SECURE = "secure";
protected Map defaultKeyNameKeyMap = null;
public Map getKeyMethodMap() {
    Map map = new HashMap();

    String pkg = this.getClass().getPackage().getName();
    ResourceBundle methods =
        ResourceBundle.getBundle(pkg + ".LookupMethods");
    Enumeration keys = methods.getKeys();
    while (keys.hasMoreElements()) {
        String key = (String) keys.nextElement();
        map.put(key, methods.getString(key));
    }
    return map;
}

public ActionForward execute(ActionMapping mapping, ActionForm form,
                            HttpServletRequest request,
                            HttpServletResponse response)
    throws Exception {
    //如果是“取消”操作直接返回
    if (isCancelled(request)) {
        ActionForward af = cancelled(mapping, form, request,
response);
        if (af != null) {
            return af;
        }
    }
    MessageResources resources = getResources(request);
    // 获取本地化的取消按钮的标识
    String edit = resources.getMessage(Locale.ENGLISH,
"button.edit").toLowerCase();
    String save = resources.getMessage(Locale.ENGLISH,
"button.save").toLowerCase();
    String search = resources.getMessage(Locale.ENGLISH,
"button.search").toLowerCase();
    String view = resources.getMessage(Locale.ENGLISH,
"button.view").toLowerCase();
    String[] rules = {edit, save, search, view};
    // 从配置文件中取得对应方法名称的参数名
    String parameter = mapping.getParameter();
    // 被调用的方法名称
    String keyName = null;

    //根据parameter从获取请求的方法名称
    if (parameter != null) {
        keyName = request.getParameter(parameter);
    }
    //如果调用的方法名为空，则选择一个最接近的方法

```

```

        if ((keyName == null) || (keyName.length() == 0)) {
            for (int i = 0; i < rules.length; i++) {
                // 如果Servlet路径中含有该请求规则,则调用相应方法
                if (request.getServletPath().indexOf(rules[i]) > -1) {
                    return dispatchMethod(mapping, form, request, response,
rules[i]);
                }
            }
            //无法将请求匹配到任何方法,调用unspecified()方法
            return this.unspecified(mapping, form, request, response);
        }
        // parameter存在,获取方法名
        String methodName =
            getMethodName(mapping, form, request, response,
parameter);
        //调用指定方法处理请求
        return dispatchMethod(mapping, form, request, response,
methodName);
    }

    //获取请求对应的ActionForm
    protected ActionForm getActionForm(ActionMapping mapping,
        HttpServletRequest request) {
        ActionForm actionForm = null;
        // 删除过时的form bean
        if (mapping.getAttribute() != null) {
            //如果ActionForm的作用域为request
            if ("request".equals(mapping.getScope())) {
                actionForm =
                    (ActionForm)
request.getAttribute(mapping.getAttribute());
            } else { //如果ActionForm的作用域为session
                HttpSession session = request.getSession();
                actionForm =
                    (ActionForm)
session.getAttribute(mapping.getAttribute());
            }
        }
        return actionForm;
    }

    //删除请求对应的ActionForm
    protected void removeFormBean(ActionMapping mapping,
        HttpServletRequest request) {
        // Remove the obsolete form bean
        if (mapping.getAttribute() != null) {
            if ("request".equals(mapping.getScope())) {
                request.removeAttribute(mapping.getAttribute());
            } else {
                HttpSession session = request.getSession();

```

```

        session.removeAttribute(mapping.getAttribute());
    }
}

//更新请求对应的ActionForm
protected void updateFormBean(ActionMapping mapping,
                               HttpServletRequest request, ActionForm
form) {
    // Remove the obsolete form bean
    if (mapping.getAttribute() != null) {
        if ("request".equals(mapping.getScope())) {
            request.setAttribute(mapping.getAttribute(), form);
        } else {
            HttpSession session = request.getSession();
            session.setAttribute(mapping.getAttribute(), form);
        }
    }
}

//从方法-名称映射表中获取方法名称
protected String getLookupMapName(HttpServletRequest request,
                                   String keyName,
                                   ActionMapping mapping)
    throws ServletException {
    String methodName = null;
    try {
        this.setLocale(request, request.getLocale());
        methodName = super.getLookupMapName(request, keyName,
mapping); //获取方法名
    } catch (ServletException ex) {
        System.out.println("BaseAction: keyName not found in
resource bundle with locale ");
    }
    // 无法在资源文件中找到对应的本地化消息文本, 就使用默认地区的消息文本
    if (defaultKeyNameKeyMap == null) {
        defaultKeyNameKeyMap =
this.initDefaultLookupMap(request);
    }
    // 获取消息文本
    String key = (String) defaultKeyNameKeyMap.get(keyName);
    if (key == null) {
        System.out.println("keyName '" + keyName + "' not found in
resource bundle with locale ");
    }
    return keyName;
}

//获取方法名称
methodName = (String) keyMethodMap.get(key);
if (methodName == null) {

```

```

        String message = messages.getMessage("dispatch.lookup",
mapping.getPath(), key);
        throw new ServletException(message);
    }
}
return methodName;
}
}

```

代码 12.9 是一段自定义的 Action 代码片断，定义了 BaseAction 类，该类扩展了 LookupDispatchAction 类，覆盖了 LookupDispatchAction 类的 execute() 方法，重新实现了方法匹配的过程，并定义了一些 Action 中常用的工具方法。

#### 12.2.4 使用自定义的 ActionForm 基类

在这里，笔者把对 ActionForm 的扩展放在控制器组件扩展部分，而不是放在视图组件扩展部分。这是因为 ActionForm 是介于视图层和控制器层之间的 JavaBean 组件，其扩展方式与扩展 Action 基类十分类似，放在这里是为了叙述上的统一。

与扩展 Action 类似，也可以自定义一个 ActionForm，作为所有 ActionForm 的基类。在自定义的 ActionForm 基类中，可以定义一些所有 ActionForm 都将用到的公共逻辑，如验证字段非空、保存错误信息等。代码 12.10 是一个自定义的 ActionForm 基类的示例。

#### 代码 12.10

```

package org.digitstore.web.struts.form;
import java.util.ArrayList;
import java.util.List;
import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;

public class BaseActionForm extends ActionForm {
    /* 公共方法 */
    public ActionErrors validate(ActionMapping mapping,
HttpServletRequest request) {
        ActionErrors actionErrors = null;
        ArrayList errorList = new ArrayList();
        doValidate(mapping, request, errorList);
        request.setAttribute("errors", errorList);
        if (!errorList.isEmpty()) {
            actionErrors = new ActionErrors();
            actionErrors.add(ActionErrors.GLOBAL_ERROR, new
ActionError("global.error"));
        }
        return actionErrors;
    }
}

```

```
public void doValidate(ActionMapping mapping, HttpServletRequest
request, List errors) {
}
/* 保护方法 */
protected void addErrorIfStringEmpty(List errors, String message,
String value) {
    if (value == null || value.trim().length() < 1) {
        errors.add(message);
    }
}
}
```

在 `BaseActionForm` 中，定义了一个 `addErrorIfStringEmpty()` 方法。该方法判断 `value` 是否为空，如果为空则添加到错误列表 `errors` 中。这种逻辑是许多 `ActionForm` 都需要的逻辑，其他 `ActionForm` 就可以直接调用该方法。

## 12.3 扩展视图组件

和控制器层中的组件相比，试图层中的组件在开发过程中通常较少扩展。一般来说，试图特定于应用程序，不同的应用会有不同的界面和外观，为一个应用程序设计的界面不大可能适用于另一个应用程序。但是，`Struts` 定制标记却是视图层中一个很好的扩展点。

### 12.3.1 扩展 `Struts` 自定义标记

`Struts` 框架提供的自定义标记库包括 `Html`、`Bean`、`Logic` 和 `Nested`。这些标记库本身就具有很高的重用性，视图中有意义的扩展也就在于对这些自定义标记库的扩展。标记符处理器都是常规的 `Java` 类，因此可以通过创建子类实现特定的功能。在 `Struts` 的标记库中对视图影响最大的是 `HTML` 标记库，`HTML` 标记库也是最有可能被扩展的地方。当扩展了标记后，必须定义存放这些标记的标记库。尽管可以包自定义的标记加入到标准的 `Struts` 标记库中，但是这样会使应用程序升级到新的 `Struts` 版本变得非常麻烦。因此应该定义单独的标记库，来存放和具体应用相关的自定义标记。

一旦创建了一个用于扩展的 `.tld` 文件并用 `Web` 应用程序部署描述符 (`web.xml`) 进行注册，用户就可以在 `JSP` 页面中方便地使用这些标记。

### 12.3.2 引入 `JSTL` 标记库

另一种对视图层扩展的方式是引入 `JSP` 标准标记库 `JSTL`。`JSTL` 是 `JSP 1.2` 定制标记库集，它们为典型的表示层任务（例如数据格式化、迭代或条件内容显示）提供标准实现。表达式语言 (`EL`) 是 `JSTL` 定制标记支持另一种用于指定动态属性值的机制。`EL` 提供了一些标识符、存取器和运算符，用来检索和操作驻留在 `JSP` 容器中的数据。

`Struts` 框架已经考虑到与 `JSTL` 的整合问题，`struts-el` 标记库便是 `Struts` 标记库的 `JSTL` 实现版本。`struts-el` 标记库中的所有标记均扩展自 `Struts` 标准标记库，不同的是，`struts-el` 采用了 `JSTL` 中的“表达式运算引擎”，而不是“运行时表达式 (`rtexprvalues`)”。举例来说，使用 `bean:write` 标记输出一段消息文本时可能会采用如下的形式：

```
<bean:message key="<%= stringvar %>" />
```

其中 stringvar 为 JSP 页面中的一个变量。如果采用 strut-el 标记库，将是下面的形式：

```
<bean-el:message key="${stringvar}" />
```

struts-el 标记库实现了绝大部分 Struts 标准标记库的标记，但是也有一些标记例外。其中 Html 标记全部实现，Bean 标记和 Logic 标记未实现的部分及其应该对应 JSTL 标记见表 12-1 和表 12-2。

表12-1 Bean标记库中未在struts-el中实现的标记

Struts标记	JSTL 标记
cookie	c:set, EL
define	c:set, EL
header	c:set, EL
include	c:import
parameter	c:set, EL
write	c:out

表12-2 Logic标记库中未在struts-el中实现的标记

Struts标记	JSTL 标记
empty	c:when, EL
equal	c:when, EL
greaterEqual	c:if, c:when, EL
greaterThan	c:if, c:when, EL
lessEqual	c:if, c:when, EL
lessThan	c:if, c:when, EL
notEmpty	c:if, c:when, EL
notEqual	c:if, c:when, EL

注意：使用 struts-el 标记的 Web 容器必须支持 Servlet 2.3 和 JSP 1.2 以上，并将 jstl.jar 置于应用程序的 WEB-INF\lib 目录下。另外，struts-el.jar 包存在于 Struts 下载包中的 contrib\lib 目录中。

## 12.4 Struts插件

相比于前面介绍的控制层和视图层组件的扩展，Struts 框架所提供的 Struts 插件机制，更像是 Struts 真正意义上的扩展。

Struts 框架从 1.1 开始提供了一种动态插入和加载组件的机制，这种机制被称为 Plugin 机制。Struts 插件实际上是一个 Java 类，它在 Struts 应用启动时被初始化，在应用关闭时被销毁。创建一个 Struts 插件，只需要实现 org.apache.struts.action.Plugin 接口即可。Plugin 接口包括两个方法，如代码 12.11 所示。

### 代码 12.11

```
public interface PlugIn {
    //应用程序关闭时将会调用Plugin的destroy()方法
    void destroy();

    /**
     * 在应用启动时将会调用Plugin的init()方法
     *
     * @param servlet ActionServlet : Struts的ActionServlet
```

```

    * @param config ModuleConfig : 与该模块相关联的配置类
    * @exception ServletException : 可能抛出的异常
    */
    void init(ActionServlet servlet, ModuleConfig config)
        throws ServletException;
}

```

一个 Struts 应用可以包含多个插件。在 Struts 应用程序启动时，Struts 框架会调用每个配置的 Plugin 的 `init()` 方法进行插件的初始化。在 `init()` 方法中一般会执行如创建数据库连接或创建与远程系统的连接等初始化操作。

当应用程序关闭时，Struts 框架会调用每个 Plugin 的 `destory()` 方法，来执行一些必要的销毁操作释放资源。例如，关闭数据库连接、远程服务界面或者释放任何插件正使用的其他资源。

Tiles 和 Validator 框架是最常用的两个 Struts 插件，并且已经成为 Struts 框架标准的一部分，关于 Tiles 和 Validator 插件的使用分别在各自独立的章节中介绍。像 Spring、Hibernate 等中间件也可以通过插件的形式与 Struts 集成。

在本书的代码示例 DigitStore 中，将使用 Hibernate 作为持久化机制。我们希望在应用程序启动时就初始化 Hibernate，以便当第一个用户请求到来时，Hibernate 就已经初始化完成并进入工作状态。相应的，在应用程序关闭时同时关闭 Hibernate。我们将创建一个 `HibernatePlugin` 类来完成这些工作。

首先，创建实现 Plugin 接口的 `HibernatePlugin` 类，`HibernatePlugin` 类的代码如下。

### 代码 12.12

```

package org.digitstore.util;
import javax.servlet.ServletContext;
...../其他需要导入的包
public class HibernatePlugIn implements PlugIn {

    //该属性作为保存在ServletContext中的SessionFactory实例的key
    public static final String SESSION_FACTORY_KEY
        = SessionFactory.class.getName();

    //指定是否在ServletContext中保存SessionFactory实例
    private boolean storedInServletContext = true;

    //Hibernate配置文件的路径
    private String configFilePaht = "/hibernate.cfg.xml";
    private ActionServlet _servlet = null;
    private ModuleConfig _config = null;
    private SessionFactory _factory = null;

    //应用程序关闭时将会调用destroy()方法
    public void destroy() {
        _servlet = null;
        _config = null;
    }
}

```

```
        try {
            _factory.close();//关闭SessionFactory实例
        } catch (Exception e) {
            System.out.println("Unable to destroy SessionFactory.. ");
            System.out.println("The Exception is " + e);
        }
    }
}

//在应用启动时，将会调用init()方法.
public void init(ActionServlet servlet, ModuleConfig config)
throws ServletException {
    _servlet = servlet;
    _config = config;

    Configuration configuration = null;
    URL configFileURL = null;
    ServletContext context = null;

    try {//获取配置文件的URL
        configFileURL =
HibernatePlugIn.class.getResource(this.configFilePath);
        context = _servlet.getServletContext();

        //初始化Hibernate配置信息
        configuration = (new
Configuration()).configure(configFileURL);
        _factory = configuration.buildSessionFactory();

        //保存SessionFactory实例到ServletContext中
        if (this.storedInServletContext) {
            context.setAttribute(SESSION_FACTORY_KEY, _factory);
        }
    } catch (Throwable t) {
        System.out.println("Exception while initializing
Hibernate...");
        throw (new ServletException(t));
    }
}

//可以在配置文件中设置的属性的setter方法
public void setConfigFilePath(String configFilePath) {
    if ((configFilePath == null) || (configFilePath.trim().length()
== 0)) {
        throw new IllegalArgumentException(
            "configFilePath cannot be blank or null.");
    }
    this.configFilePath = configFilePath;
}
}
```

```

        public void setStoredInServletContext(String storedInServletContext)
    {
        if ((storedInServletContext == null)
            || (storedInServletContext.trim().length() == 0)) {
            storedInServletContext = "false";
        }

        this.storedInServletContext
            = new Boolean(storedInServletContext).booleanValue();
    }
}

```

当 Struts 框架调用 HibernatePlugIn 类的 init()方法时，会把 ActionServlet 作为参数传给 init()方法。因此在 init()方法中可以通过 ActionServlet 的 getServletContext()方法来获得 ServletContext 对象的引用。

HibernatePlugIn 类的 init()方法根据配置文件的路径加载 Hibernate 配置文件，并立即初始化 Hibernate 配置信息。然后生成一个 SessionFactory 实例。如果将 SessionFactory 实例保存到 ServletContext 对象中，SessionFactory 实例就可以被整个应用程序共享。

插件需要由 Struts 框架来加载。这通过在 Struts 配置文件中定义新的 plug-in 元素来实现。Struts 框架会在启动时根据配置的 plug-in 元素初始化插件。代码 12.13 为配置 HibernatePlugIn 类的代码示例。

### 代码 12.13

```

<plug-in className=" org.digitstore.util.HibernatePlugIn">
  <set-property property="configFilePath"
    value="/hibernate.cfg.xml" />
  <set-property property="storeInServletContext" value="true" />
</plug-in>

```

Struts 框架允许在配置文件中设置插件的属性值，plug-in 元素中的 set-property 即是用来设置插件属性的值。同时，在自定义的插件类中，必须为该属性定义 JavaBean 风格的 setter 方法，getter 方法是可选的。以上 plug-in 元素的配置包含 2 个 set-property 子元素，它们定义了插件类中相应属性的值。在 HibernatePlugIn 类中，\_configFilePath 属性和 \_storedInServletContext 属性定义和 setter 方法如代码 12.14 所示。

### 代码 12.14

```

private boolean storedInServletContext = true;
private String configFilePath = "/hibernate.cfg.xml";
public void setConfigFilePath(String configFilePath) {
    if ((configFilePath == null) || (configFilePath.trim().length()
== 0)) {
        throw new IllegalArgumentException(
            "configFilePath cannot be blank or null.");
    }
    this.configFilePath = configFilePath;
}
}

```

```
public void setStoredInServletContext(String storedInServletContext)
{
    if ((storedInServletContext == null)
        || (storedInServletContext.trim().length() == 0)) {
        storedInServletContext = "false";
    }

    this.storedInServletContext
        = new Boolean(storedInServletContext).booleanValue();
}
```

Struts 框架在加载插件时，会调用插件类的 setName()方法，把 set-property 元素的属性值传给 HibernatePlugIn 实例的 setName()方法。

提示：如果在 Struts 配置文件中定义了多个插件，Struts 框架会按照这些插件在配置文件中的先后顺序依次初始化。

## 12.5 本章小结

本章首先介绍了 Struts 多模块开发有关的问题，阐述了在 Struts 框架中进行多模块开发的方式以及一些主要的问题。然后我们分别从控制器和视图的角度介绍了对 Struts 框架的扩展。在控制器扩展中主要是应用自定义的各种控制器组件代替默认的控制组件，包括自定义的 ActionServlet、自定义的 RequestProcessor、自定义的 Action 以及自定义的 ActionForm。在视图扩展中主要提到了对自定义标记和引入 JSTL 标记库。最后对 Struts 插件进行了详细介绍，并给出了应用实例。