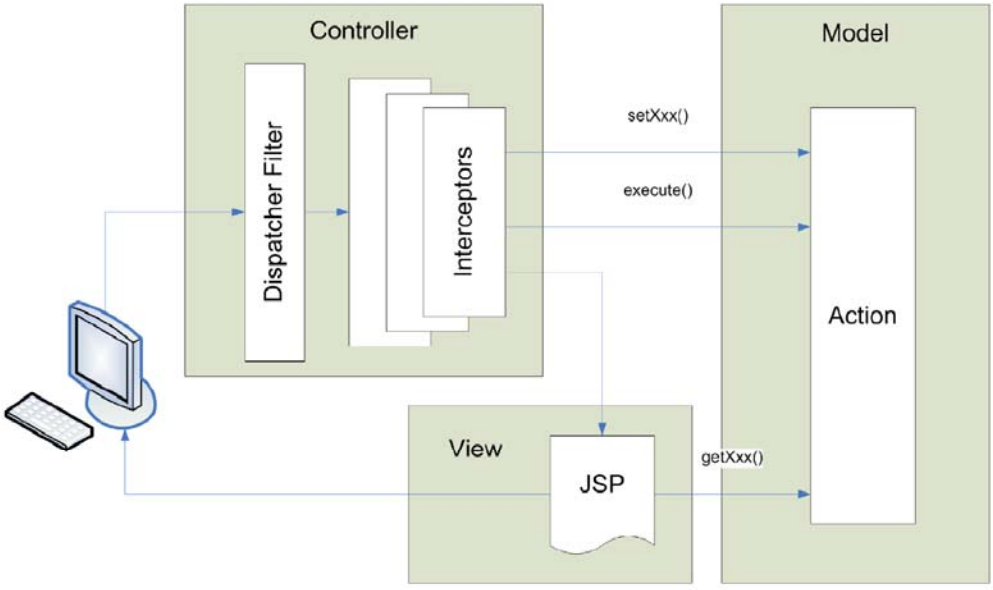
开发技巧和整合策略

深入浅出Struts 2

Starting Struts 2



Ian Roughley 著
李剑 译

InfoQ 企业软件开发丛书

免费在线版本

（非印刷免费在线版）

InfoQ 中文站出品

本书主页为

http://infoq.com/cn/minibooks/starting-struts2

深入浅出 Struts 2

Starting Struts 2

作者：**Ian Roughley**

翻译：李剑

C4Media 是 InfoQ.com 这一企业软件开发社区的出版商

本书属于 InfoQ 企业软件开发系列图书

如果您打算订购InfoQ的图书，请联系 books@c4media.com

英文版责任编辑：Diana Plesa
英文版封面设计：Dixie Press
英文版美术编辑：Dixie Press

中文版翻译：李剑
中文站审校：张凯峰 霍泰稳
中文版责任编辑：霍泰稳
中文版美术编辑：吴志民

欢迎共同参与InfoQ中文站的内容建设工作，包括原创投稿和翻译等，请联系 editors@cn.infoq.com。

10 9 8 7 6 5 3 2 1

作者致谢

如果没有 Webwork、XWork 和 Struts2 所有开发人员不懈的努力，这本书将永远无法面世。在我从一个开源项目的使用者变成一个开源项目的开发人员的过程中，Patrick Lightbody 和 Jason Carreira 对我的帮助将永远铭刻我心。

同时，我也应该感谢这几位技术评审人员——Don Brown，Philip Luppens 和 Rene Gielen，是他们为本书内容的不断扩充提出了最终的调整方案。还有 Jim Krygowski 和 James Walker，他们从繁忙的工作日程安排中特意抽出时间，站在 Struts2 以外的视角上，对本书的内容及连贯性提出了中肯的意见。如果没有他们的帮助，本书势必失色不少。我还要感谢 Floyd Marinescu，他对我给予了充分的信任，并为我提供了网络版和印刷版的写作机会。

我还要谢谢我才华横溢的妻子 LeAnn (也就是 STR Worldwide)，她一直支持着我的工作，并且长久以来一直对书稿进行评审和非技术层面的分析，这是我的无价之宝。

目 录

1 简介

自从 1997 年第一个 Servlet 规范发布以来，我们在用 Java 开发 Web 应用这条路上已经走了很远很远。在过去的时间里，我们学会了很多，也曾经不止一次地对开发 Web 应用的方式做出过改进。Apache Struts 的产生就是一个伟大的跨越，它的价值远远超过了我们目光所能及的极限。

Apache Struts 在 2000 年 5 月由 Craig McClanahan 发起，并于 2001 年 7 月发布了 1.0 版本。从技术的角度上来讲，它是在开发 Web 程序上的一次跨越性的进步，但更重要的是，它在最恰当的时候出现在了人们的眼前。

到那时为止，Web 开发已经度过了漫长的岁月，很多大型项目都已经完工进入了维护期，在这个过程中，代码的可重用性与可维护性也给开发人员好好的上了几课。.com 的兴起也引发了对 Web 应用开发更好的解决方案的诉求——当 Apache Struts 在 2000 年出现的时候，Web 项目的数量仍然显著地增长着，而且毫无终止之势。Struts 一出现便大受欢迎，更成为了以后几年内 web 开发的实际标准。

Struts2[1]是Struts的下一代产品。而最初提案Struts Ti所设想的发展方向，在Struts的现有代码的基础上是很难完成的。在发起提案的时候，Patrick Lightbody把多个不同的Web框架的领导者邀请到了一起，希望大家能够达成共识，协力完成一个通用框架。虽然最终由于各种原因，Patrick Lightbody 的愿望未能实现，但是 WebWork 和

[1] http://struts.apache.org/2.x

Struts Ti却发现了二者在技术与开发人员这两个层面上的共同之处，不久之后，两个项目就在WebWork的技术基础上进行了合并[2]。

> 当我们说起 WebWork 的时候，我们实际上说的是两个项目——XWork 和 WebWork。XWork 是一个通用的命令框架，它提供了很多核心的功能，例如 actions，验证和拦截器，它可以完全独立于执行上下文运行，并提供了一个内部的依赖注入机制，用来做配置和工厂实现的管理。
>
> 而 WebWork 则是一个完全独立的上下文。它用 Web 应用中运行所需的上下文把 XWork 包装起来，并提供了可以简化 Web 开发的特定实现。

Struts2 的目标很简单——使 Web 开发变得更加容易。为了达成这一目标，Struts2 中提供了很多新特性，比如智能的默认设置、annotation 的使用以及“惯例重于配置”原则的应用，而这一切都大大减少了 XML 配置。Struts2 中的 Action 都是 POJO，这一方面增强了 Action 本身的可测试性，另一方面也减小了框架内部的耦合度，而 HTML 表单中的输入项都被转换成了恰当的类型以供 action 使用。开发人员还可以通过拦截器（可以自定义拦截器或者使用 Struts2 提供的拦截器）来对请求进行预处理和后处理，这样一来，处理请求就变得更加模块化，从而进一步减小耦合度。模块化是一个通用的主题——可以通过插件机制来对框架进行扩展；开发人员可以使用自定义的实现来替换掉框架的关键类，从而获得框架本身所不具备的功能；可以用标签来渲染多种主题（包括自定义的主题）；Action 执行完毕以后，可以有多种结果类型——包括渲染 JSP 页面，Velocity 和 Freemarker 模板，但并不仅限于这些。最后，依赖注入也成了 Struts2 王国中的一等公民，这项功能是通过 Spring 框架的插件和 Plexus 共同提供的，与 PicoContainer 的结合工作还正在进行中。

本书的目的，是为了帮助读者掌握 Struts2 框架，并能够对组成框架的功能部件和可用的配置项有深刻的理解。我在书中还将介绍一些可以提高生产力的方法——包括默认配置项和应当注意的实现特性，可用的多种配置选项和一些开发技术。本书还会就与第三方软件进行集成的话题展开讨论。

[2] Don Brown, Struts Ti项目的领导，他在文章中详细介绍了Struts Ti的历史，详情请参见http://www.oreillynet.com/onjava/blog/2006/10/my_history_of_struts_2.html.

本书并不会对 Struts2 中所有的特性进行面面俱到的讲述。作为一个新项目，Struts2 仍然在不停地迎来变化、升级以及新的特性。我希望读者能够花一些时间访问一下项目的主页，从中你可以发现很多本书中未能囊括的选项和特性。

本书参考的是 Struts2 的 2.0.6 版本。

2

Web 世界中，Struts2 身处何方

今天，摆在开发人员面前的是众多的Web开发框架。它们有些来自于开源社区，有些来自于商业公司，还有些是由某些团队内部开发的，以满足当前Web开发所需。目前大约一共有 超过 40 个[3]开源框架，这个数目看上去挺大，但也许还有同样多（如果不是大大多于这个数量的话）的内部构建的框架部署在产品环境中。

既然我们有这么多选择，那为什么要选择 Struts2 呢？下面列出的这些特性，可能会促使你把 Struts2 作为你的选择：

- 基于 Action 的框架
- 拥有由积极活跃的开发人员与用户组成的成熟社区
- Annotation 和 XML 配置选项
- 基于 POJO 并易于测试的 Action
- 与 Spring，SiteMesh 和 Tiles 的集成
- 与 OGNL 表达式语言的集成
- 基于主题的标签库与 Ajax 标签
- 多种视图选项 (JSP，Freemarker，Velocity 和 XSLT)
- 使用插件来扩展或修改框架特性。

在选择一个框架的时候，框架风格的选择是最具有争议性的。让我们先来了解一下 Web 应用的发展过程，然后再来看看我们是怎样走到今天的，以及在现在的 Web 开发场景中，Struts2 到底占据了什么位置。

[3] 你可以访问 http://www.java-source.net/open-source/web-frameworks来获取一份各种web框架的列表。

Servlets

Servlets 是 Java 在 Web 应用中的开创性的尝试。在遵循 HTTP 协议的前提下，Servlets 可以将 URL 映射到一个特定的类上，而该类中的方法将会被调用。

人们马上就意识到，虽然这是一次大踏步式的前进，但是这种在 Java 代码中生成 HTML 代码的方式，对项目维护而言简直就是一场噩梦。每次当用户界面发生了变化，Java 开发人员就需要更改 Servlet 代码，重新编译，然后把应用重新部署到服务器上。

JSP和Scriptlet开发

这种“维护噩梦”的后果，又导致开发风格的正好颠倒。现在人们不是把 HTML 代码放在 Servlet 或者 Java 代码中，而是把 Java 代码（作为 script-lets）放在 HTML 代码中——这就是 Java Server Pages (JSP)。每一个 JSP 都同时负责处理请求和页面表现。

一个问题得到了解决，而另一个问题却又出现了。在 JSP 中，Java 代码的使用方式和在类中一样，但这里却没有方法和类的结构。在早期的每一个 JSP 文件中，都可以找到以下两种情况之一：

- **剪切和粘贴的代码** —— Java 代码从一个 JSP 中复制到第二个，第三个，等等。这种情况会导致在原始代码中存在的缺陷或者错误传播开来，并且大大增加了工作量，因此必须要对此做出改变。
- **调用通用的 Java 格式化对象**——通用的格式化代码或者是逻辑代码被组织到一个可重用的对象中，然后每一个 JSP 都会使用这个通用的对象。

基于这些情况，一种最佳实践，或者可以称之为一种模式，就应时而生了——在 JSP 中使用 Java 对象。

随着 JSP 规范的进一步完善，标签开始被引入进来对可重用的 Java 对象进行封装。标签提供了用以访问底层代码的表层代码，它与 HTML 很相像，设计人员（而不是开发人员）和 IDE 可以通过它与动态内容交互，组装出页面布局。像<jsp:useBean.. />和

<jsp:getProperty.. />就是 JSP 所提供的标签。JSP 在提供了一系列标签库的同时，还可以支持开发人员创建自定义的标签库。

基于Action的框架

基于 Action 的框架把 Servlet 和 JSP 的概念合并到了一起。它的想法是把对当前用户所见的页面请求的处理动作，分拆成处理逻辑和表现逻辑，让它们各司其职。这种实现方式使用了源自于 Smalltalk 的一个模式，名为模型-视图-控制器——最近的叫法是前端控制器，而 Sun 则给它起名为 Model 2。

在这个模式中，Servlet 是控制器，集中处理所有的客户端页面请求。它把所请求的 URL 与被称为 Action 的工作单元映射到一起。Action 的工作就是通过访问 HTTP 会话、HTTP 请求和表单参数等调用业务逻辑，最后把响应映射到以 POJO（plain old java object）形式存在的模型上，来完成特定的功能。最后，Action 返回的结果会通过配置文件映射到 JSP 页面上，JSP 会渲染视图并显示给用户。

Struts2 是一个基于 Action 的 MVC Web 框架。

基于组件的框架

当 Web 应用变得更加复杂的时候，人们便意识到一个页面已经不再是一个独立的逻辑了——一个页面上会存在多个表单，有内容更新的链接，还有其他很多自定义的 Widget——而这些都需要进行逻辑处理来完成各自的任务。

出于解决这种复杂度的需要，基于组件的框架开始流行起来。它们在用户界面组件和表示这些组件的类之间提供了一层紧密的连接，它们是事件驱动型的，并且比起基于 Action 的框架而言，更具有面向对象的特征。一个组件可以是一个 HTML 输入框，一个 HTML 表单，框架所提供的或是开发人员创建的 Widget。像提交表单或者是点击链接这样的事件，都与代表组件的类的方法或者是特定的监听类，有着一对一的映射关系。基于组件的框架还有一个好处，那就是你可以在多个 Web 应用之间重用组件。JSF，Wicket 和 Tapestry 等都是基于组件的框架。

伟大的均衡器——Ajax

在 2005 年初，Web 应用又增异彩。按照 Jesse James Garrett 的说法， Ajax 的全称是 “Asynchronous JavaScript and XML”。 相对说来，这些技术没有一样是新的。实际上，早在 6 年以前（从 Internet Explorer 5 开始），可进行异步调用的主要 Web 浏览器组件——XMLHttpRequest 对象就已经提供了支持。

真正推陈出新的是这些技术的应用。Google 地图第一个完全利用了这项技术所带来的好处，在 Google 地图中，网页是活动的，你可以通过各种窗体组件与它进行交互。你可以用鼠标来拖动屏幕上的地图；当你输入一个地址时，相关信息还会在地图的对应位置显示出来；最后，当它们都与 Web 程序完美结合以后，Web 应用终于攀上了新的巅峰。这些操作都不会带来页面的刷新！

当用户界面与 Ajax 结合以后，Web 浏览器就可以只在必需的时候，才会向服务器发起请求，获得少量的信息。服务器返回的结果都是被格式化或者处理过的，页面会直接把结果显示出来，然后用户就可以在浏览器中看到变化。因为只有发生变化的那一块区域会被重新渲染，而不是整个页面进行刷新，所以对于用户来说，响应速度就变得更快了。

从 UI 发出的请求和事件很相似——它们是不连续的，所传递的只是一个单独的组件或者功能的信息。现在的操作已经再也不需要获取整个页面的信息了，它们变得更加精细，跨应用的可重用性也变得更高。其结果就是，当一个 Ajax 用户界面调用基于 Action 的框架时，这个 Action 框架的反应机制就和基于组件的框架非常相似。实际上，这二者的结合为我们带来了耦合度更低、可重用性更高的系统。同样的操作，可以为 Ajax 组件提供 JSON、XML 或者 HTML 片段视图，又可以和其他操作组合，为非 Ajax 的用户界面提供 HTML 视图。

3 核心组件

从全局的角度来看，Struts2 是一个 pull（拉）类型的 MVC（或者 MVC2）框架，它与传统类型的 MVC 框架的不同之处就在于在 Struts2 中，Action 担任的是模型的角色，而非控制器的角色，虽然它的角色仍然有些重叠。“pull”的动作由视图发起，它直接从 Action 里拉取所需的数据，而不是另外还需要一个单独的模型对象存在。

我们刚刚从概念上讲述了一下 Struts2 里面的 MVC，那么从实现的层面上来看又是什么样子呢？在 Struts2 中，模型-视图-控制器模式通过五个核心组件来实现——Action、拦截器？值栈/OGNL？结果类型和结果/视图技术。

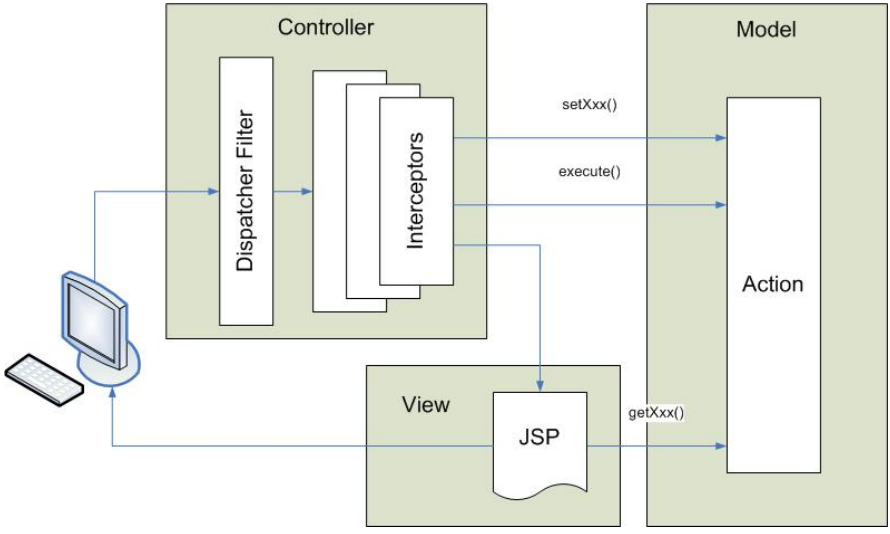


图 1: MVC / Struts2 架构

图 1 描述了 Struts2 架构中的模型、视图和控制器。控制器通过 Struts2 分发 Servlet 过滤器（也就是人们常说的拦截器）来实现，模型通过 Action 实现，视图则通过结果类型和结果组合实现。值栈和 OGNL 提供了公共的线程和链接，并使得不同组件可以相互集成。

当我们在本章讨论通用组件时，其中有大量的信息是与配置相关的，也就是对 Action、拦截器、结果等等的配置。 在阅读时要注意一点，这里的讲解只是帮助读者掌握一些背景知识，并没有采用最有效率的配置方式。在后续的章节中，我们会介绍一些更加简单有效的方式，诸如“惯例重于配置”、注解和零配置插件。

在我们对核心组件进行深入探讨之前，首先让我们来看看全局性的配置。

配置

在配置 Struts 2 之前，你首先要把发行版下载下来，或者在 Maven2 的“pom.xml”文件中声明如下的依赖：

```
<dependency>
<groupId>org.apache.struts</groupId>
<artifactId>struts2-core</artifactId>
<version>2.0.6</version>
</dependency>
```

Mave2 是一种管理项目整体构建过程的工具——包括编译，运行测试，生成报告以及管理构建的制品，其中对开发人员最有吸引力的一项就是对构建制品（artifact）进行管理。

应用程序的依赖库只需要在项目的“pom.xml”文件中通过 `groupId`、`artifactId` 和 version 进行定义即可。在使用一个制品之前，Maven 会从本地和远程的库中进行查询，查询的范围包括本地的缓存、网上其他组织提供的库以及 ibiblio.com 这个标准的库。如果在远程的库中找到了所需的制品，Maven 就会把它下载到本地的缓存中，以供项目使用。当请求所需要的制品时，这个制品相关联的依赖也会同样被下载到本地来（假设在该制品的“pom.xml”文件中对所有依赖都依次进行了声明）。

Struts2 是用 Maven2 来构建的，在 pom 文件中，它定义了所有依赖的配置项。您可以访问 http://maven.apache.org 来获得关于 Maven2 的更多信息。

Struts2 的配置可以分成三个单独的文件，如图 2 所示。

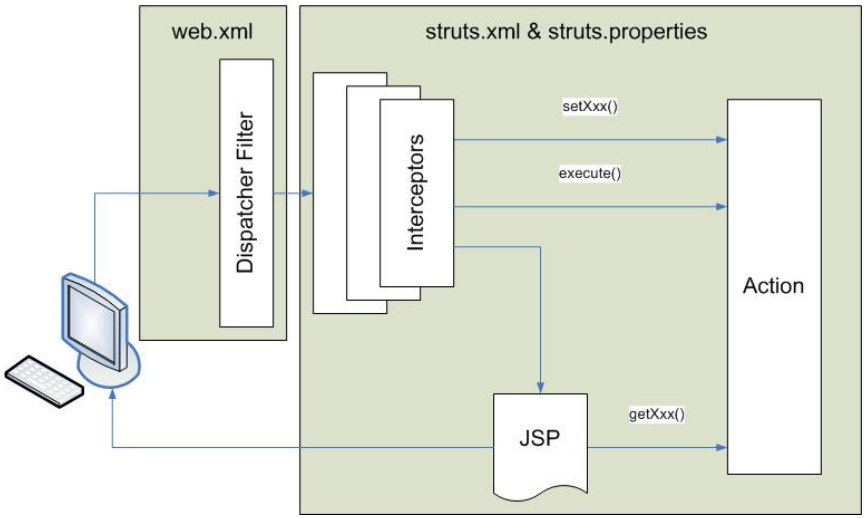


图 2： 框架元素的配置文件

FilterDispatcher 是一个 servlet 过滤器，它是整个 Web 应用的配置项，需要在“web.xml”中进行配置：

```
    <filter>
    <filter-name>action2</filter-name>
    <filter-
class>org.apache.struts2.dispatcher.FilterDispatcher</fil
ter-class>
    </filter>

    <filter-mapping>
    <filter-name>action2</filter-name>
    <url-pattern>/*</url-pattern>
    </filter-mapping>
```

如果是配置一个最基本的 Web 应用的话，这样就足够了。剩下的就是自定义一些 Web 应用的执行环境和配置应用中的组件。其中前者主要通过“struts.properties”来完成，而后者是在“struts.xml”中进行配置的。我们下面来看一下这两个配置文件的细节。

struts.properties 文件

这个文件提供了一种更改框架默认行为方式的机制。在一般情况下，如果不是打算让调试更加方便的话，你根本无须更改这个文件。在“struts.properties”文件中定义的属性都可以在“web.xml”文件的“init-param”标签中进行配置，或者通过“struts.xml”文件中的“constant”标签来修改（我们在下一章中会继续讨论这个标签）。

我们可以更改其中的一些属性来改变Freemarker的选项——改变Action映射类，判断是否要重新载入XML配置文件，以及默认的UI主题等等。在Struts2 的wiki上有这些属性的最新信息，地址为 http://struts.apache.org/2.x/docs/strutsproperties.html.

在 Struts2-Core 的 jar 发行版中，有一个默认的属性文件，名为“default.properties”。如果要对属性进行修改的话，只需要在项目的源代码目录下，创建一个叫做“struts.properties”的文件，然后把想要修改的属性添加到文件中，新的属性就会把默认的覆盖掉了。

在开发环境中，以下几个属性是可能会被修改的：

- **struts.i18n.reload = true**——激活重新载入国际化文件的功能
- **struts.devMode = true** ——激活开发模式，以提供更全面的调试功能。
- **struts.configuration.xml.reload = true**——激活重新载入 XML 配置文件的功能（这是为 Action 准备的），当文件被修改以后，就不需要重新载入 Servlet 容器中的整个 Web 应用了。
- **struts.url.http.port = 8080**——配置服务器运行的端口号（所有生成的 URL 都会被正确创建）
-

struts.xml 文件

“struts.xml”文件中包含的是开发 Action 时所需要修改的配置信息。在本章接下来的内容中，会针对特定的元素进行详细讲解，但现在让我们先来看一下文件的固定结构。

有的时候你甚至可以把整个“struts.xml”文件从应用中移走，这完全取决于应用程序的功能需求。我们在本章中所讨论的配置信息，

都可以被其他方式所代替，比如注解，“web.xml”启动参数和和可替换的 URL 映射方案。

必须要在“struts.xml”中进行配置的信息只有全局结果、异常处理和自定义的拦截器堆栈。

因为这是一个 XML 文件，所以最开始的元素就是 XML 版本和编码信息。接下来则是 XML 的文档类型定义（DTD）。DTD 提供了 XML 文件中各个元素所应使用结构信息，而这些最终会被 XML 解析器或者编辑器使用。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts
Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>

<package
       name="struts2"
       extends="struts-default"
       namespace="/struts2">

      …

</package>

</struts>
```

我们现在看到了<struts>标签，它位于 Struts2 配置的最外层，其他标签都是包含在它里面的。

Include 标签:

`<include … />` 是<struts>标签的一个子标签，它可以把其他配置文件导入进来，从而实现 Struts2 的模块化。它的“file”属性定义了要导入的文件的名称——该文件要和“struts.xml”一样有着相同的结构。比如说，如果要把一个记账系统的配置分解的话，你可能会把记账、管理和报表的功能各自组织到不同的文件中：

```
<struts>

<include file="billing-config.xml" />
```

```
<include file="admin-config.xml" />
<include file="reports-config.xml" />

 …

</struts>
```

当我们导入文件时，一定要注意导入的顺序。因为从文件被导入的那个点开始，该文件中的信息才能被访问到，也就是说，如果要使用另外一个文件中所定义的标签，那么该文件就必须要在被引用之前就配置好。

有些文件需要显式导入，有些则会被自动导入。“struts-default.xml”和“struts-plugin.xml”就属于后者。它们都包括有结果类型、拦截器、拦截器堆栈、包（package）以及 Web 应用执行环境（也可以在“struts.properties”中配置）的配置信息。二者的区别在于，“struts-default.xml”提供的是 Struts2 的核心配置信息，而“struts-plugin.xml”则描述了特定插件的配置信息。每个插件的 JAR 中都要有一个“struts-plugin.xml”文件，该文件会在系统启动时被装载。

The Package Tag:

`<package … />`标签是用来把那些需要共享的通用信息——例如拦截器堆栈或 URL 命名空间——的配置组织在一起的。它通常由 Action 的配置组成，但也可以包括任何类型的配置信息。它还可以用来组织管理各自独立的功能——它们也可以被进一步拆分到不同的配置文件中。

这个标签的属性包括有：

- ***name*** ——开发人员为这个 Package 指定的唯一的名字。
- ***extends*** —— 当前这个 Package 所继承的 Package 的名字，被继承的 Package 中所有的配置信息（包括 Action 的配置）都可以在新的命名空间下，新的 Package 里面被使用。
- ***namespace*** ——命名空间提供了从 URL 到 Package 的映射。也就是说，如果两个不同的 Package，其命名空间分别为“package1”和“package2”，那么 URL 差不多就是

“/myWebApp/package1/my.action” 和
“/myWebApp/package2/my.action”这样的形式。

- ***abstract*** ——如果这个属性的值为“true”，那么这个 Package 就只是一个配置信息的组合，也就无法通过 Package 的名字来访问其中配置的 Action。

只有继承了正确的父 Package，那么你才能用到所需的预先配置好的特性。在大多数情况下，我们都应该继承“struts-default.xml”配置文件中的“strust-default”Package，但是如果你想要使用插件的话，那就另当别论了。你必须要参考插件的说明文档来确认所应当继承的父 Package。

后面我们会继续讲解 Package 标签的详细配置信息。

还有两个标签也可以和`<struts>`标签一起使用，它们是`<bean … />`和`<constant … />`。这两个标签为重新配置框架提供了更高级的方式，我们在下一章讲述插件的时候会介绍它们的使用方法和配置细节。

Actions

在大多数的 Web 应用框架中，Action 都是一个最基本的概念，也是可以与用户发出的 HTTP 请求相关联的最小的工作单元。

在 Struts2 中，Action 可以以几种不同的方式来工作。

单个结果

Action 最常用也是最基本的用法就是执行操作后返回单个结果。这种 Action 看上去就是这样的：

```
class MyAction {

 public String execute() throws Exception {
     return "success";
 }

}
```

这样简单的几行代码当然说明不了什么。但首先，这个 Action 类不需要继承其它类，也不需要实现其他接口。这样的类就是一个简单的 POJO。

其次，在这个类中有一个名为“execute”的方法。这个方法名是依照惯例命名的，如果你想用其他名字的话，那么只需要在 Action 的配置文件中做出更改。无论方法名是什么，它们都被认为会返回一个 String 类型的值。Action 的配置文件会将该 Action 的返回代码与要呈现给用户的结果进行匹配。另外，该方法还可以在需要的时候抛出异常。

下面是最简单的配置信息：

```
<action name="my" class="com.fdar.infoq.MyAction" >
<result>view.jsp</result>
</action>
```

“name”属性提供了执行Action所对应的URL地址，在这里就是“my.action”。“.action”的扩展名是在“struts.properties”[4]文件中配置的。“class”属性指定了要执行的action所对应的类的全限定名。

多个结果

现在情况稍微复杂了一些，Action 需要根据逻辑运算的结果，来生成多个结果。下面的代码和刚才那个类看上去很像：

```
class MyAction {

 public String execute() throws Exception {
     if( myLogicWorked() ) {
             return "success";
     } else {
         return "error";
     }
 }

}
```

因为这里有两个结果，所以就为每一种不同的情况来配置要呈现给用户的结果。配置文件就变成了如下的样子：

[4] 只需要更改 “struts.action.extension”这个属性的值。

```
<action name="my" class="com.fdar.infoq.MyAction" >
<result>view.jsp</result>
<result name="error">error.jsp</result>
</action>
```

我们可以看到在 result 节点中多了“name”属性，实际上这个属性是一直都存在的，如果开发人员没有显式指定它的值，那么它的默认值就是“success”（第一个 result 的配置就是如此）。

前面我们已经看到了定义 Action 结果的最通用的方式。而实际上我们还有另外四种方式：

1. Action 方法返回一个字符串——这个返回的字符串与“struts.xml”的一个 action 配置相匹配。例子中已经演示这一种方式。
2. 使用Code behind插件 ——当使用这个插件的时候，它会将Action的名字和Action返回的结果字符串进行连接来得到视图模板。比如说，如果URL是“/adduser.action”，而Action返回了“success”，那么要渲染的页面就是“/adduser-success.jsp”。更多信息请参见http://struts.apache.org/2.x/docs/codebehind-plugin.html。
3. 使用 @Result 注解—— action 类可以用@Results 和@Result 注解来标注多个不同的结果。Action 所返回的字符串需要与所注解的结果之一相匹配。
4. 方法返回一个 Result 类的实例——Action 不必一定要返回一个字符串，它可以返回一个 Result 类的实例，该实例应当是已经配置好可使用的。

结果类型

Action 生成并返回给用户的结果可能会有多个值，而且也可能是不同的类型。“success”的结果可能会渲染一个 JSP 页面，而“error”的结果可能需要向浏览器发送一个 HTTP 头。

结果类型（本章中稍后会详细讨论）是通过 result 节点的“type”属性来定义的。和“name”属性一样，这个属性有一个默认值——“dispatcher”——用来渲染 JSP。大多数情况下，你只需要使用默认的结果类型就可以了，但是你可以提供自定义的实现。

请求和表单类型

Action 为了执行操作，并为数据库持久化对象提供数据，就必须要访问请求字符串和表单中的数据。

Struts2 采用了 JavaBean 的风格——要访问数据的话，就给字段提供一个 getter 和 setter，要访问请求字符串和表单也是一样的道理。每一个请求字符串和表单的值都是一个简单的名/值对，所以要设定一个特定名称的值的话，就要为它提供一个 setter。比如，如果一个 JSP 调用了“/home.action?framework=struts&version=2”这样一个请求，那么 action 就应该提供如下两个 setter：“setFramework(String frameworkName)”和“setVersion(int version)”。

我们可以看到，例子中的 setter 并不是只接受 String 类型的参数。在默认情况下，Struts2 可以把 String 类型的值转换成 action 所需要的类型，这条规则对于所有的 primitive 类型和基本对象类型的值都适用，当然你也可以对其进行配置，让它也适用于你所创建的类。Struts2 还可以在更加复杂的对象图中进行定位后赋值，比如说如果一个表单元素的名字是“person.address.home.postcode”，其值为“2”，那么 Struts2 就会调用“getPerson().getAddress().getHome().setPostcode(2)”这个方法。

访问业务服务

到现在为止，我们已经讨论了很多有关 Action 配置的问题，以及如何根据不同的结果代码来控制返回给用户的结果。这是 Action 的一个很重要的功能，但是，它必须要完成一些操作之后，才能返回结果。因此它们需要访问多种类型的对象——业务对象，数据访问对象或者其他资源。

Struts2 使用了名为依赖注入[5]——又名控制反转——的技术来降低系统的耦合性。依赖注入可以通过构造器注入，接口注入和setter注入来实现。Struts2 中用的是setter注入。这就是说，你只需要提供一个setter，对应的对象就可以被Action使用了。Struts2 推荐的依赖注入框架是Spring框架，并通过插件对它进行配置。你还可以使用Plexus，或者是提供自定义的实现。

[5] Martin Fowler写过一篇文章，对依赖注入进行了完整的描述：http://www.martinfowler.com/articles/injection.html

还有些对象是不受 Spring 框架管理的，例如 HttpServletRequest。这些对象是通过 setter 注入和接口注入混合处理的。对于每一个非业务的对象而言，都有一个对应的接口（也就是“aware”接口），需要 action 对其进行实现。

> 在最开始的时候，WebWork 有它自己独有的依赖注入框架。但自从 2.2 版本以后，该框架就被 Spring 取代了。原先的那种组件框架是基于接口的，所以每一个组件都需要有一个接口和一个对应的实现类。
>
> 另外，每一个对象都有一个“Aware”接口，为组件提供了 setter 方法。对于 UserDAO 接口而言，对应的 aware 接口按照惯例就会被命名为“UserDAOAware”，其中有一个 setter 方法——“void setUserDAO(UserDAO dao);”。

当必需的接口和 setter 齐备以后，拦截器就会对对象的注入进行管理了。

从 Action 中访问数据

在有些情况下我们需要查看被 Action 修改过的对象。有好几种技术可以帮助我们做到这一点。

对于开发人员而言，最常用的方法就是把所需要访问的对象放到 HttpServletRequest 或者 HttpSession 里面。这可以通过实现“aware”接口（让依赖注入来工作），并设置为可以通过要求的名称来访问的方式来达到。

如果你打算使用内建的标签库或者是 JSTL 支持的话，访问数据就会更简单了。这两种方法都可以通过值栈来访问 Action。开发人员唯一要另外做的就是在 Action 里面为所要访问的对象提供 getter 方法。在下个关于值栈的小节中我们会进行更深入的介绍。

Interceptors（拦截器）

Struts2 中提供的很多特性都是通过拦截器实现的，例如异常处理，文件上传，生命周期回调与验证。拦截器从概念上来讲和 Servlet 过滤器或者 JDK 的 Proxy 类是一样的。它提供了一种对

Action 进行预处理和事后处理的功能。和 Servlet 过滤器一样，拦截器可以被分层和排序。它还可以访问所执行的 Action 和所有的环境变量与执行属性。

让我们从依赖注入来开始对拦截器的介绍。正如我们已经看到的一样，依赖注入可以多种不同的实现方式。下面就是对应于不同实现方式的拦截器：

- Spring 框架——`ActionAutowiringInterceptor` 拦截器
- 请求字符串和表单值—— `ParametersInterceptor` 拦截器
- 基于 Servlet 的对象——`ServletConfigInterceptor` 拦截器

前两种拦截器可以独立工作，不需要 Action 的帮助，但是最后一种不同，它是在以下几种接口的辅助下工作的：

- `SessionAware` ——通过 Map 来提供对所有 session 属性的访问
- `ServletRequestAware`——提供对 `HttpServletRequest` 对象的访问
- `RequestAware` ——通过 Map 来提供对所有 request 属性的访问
- `ApplicationAware` ——通过 Map 来提供对所有 applicatin 属性的访问
- `ServletResponseAware` ——提供对 `HttpServletResponse` 对象的访问
- `ParameterAware` ——通过 Map 来提供对所有 request string 和表单数据的访问
- `PrincipalAware` ——提供对 `PrincipleProxy` 对象的访问；该对象实现了 `HttpServletRequest` 对象的有关 principle 和 role 的方法，但是它提供了一个 Proxy，因此所有的实现都是独立于 Action 的。
- `ServletContextAware` ——提供对 `ServletContext` 对象的访问

如果要把数据注入到 Action 中去，那么对应的数据就需要实现必需的接口。

配置

如果要在 Action 中激活依赖注入功能（或其他任何由拦截器提供的功能），就必须要对 Action 进行配置。和其他元素一样，许多

拦截器都已经提供了默认的配置项。你只需要确认一下 Action 所在的 Package 继承了“struts-default”package。

在配置一个新的拦截器之前，首先要对它进行定义。<interceptors … />和<interceptor … />标签都要直接放到<package>标签里面。像我们上面提到的那些拦截器，它们的配置项就是这样的：

```
<interceptors>
 …
<interceptor name="autowiring"
class="…xwork2.spring.interceptor.ActionAutowiringInte
rceptor"/>
</interceptors>
```

我们同时还要确保 Action 中应用了所需的拦截器。这可以通过两种方式来实现。第一种是把拦截器独立的分配给每一个 Action：

```
<action name="my" class="com.fdar.infoq.MyAction" >
<result>view.jsp</result>
<interceptor-ref name="autowiring"/>
</action>
```

在这种情况下，Action 所应用的拦截器是没有数量限制的。但是拦截器的配置顺序必须要和执行的顺序一样。

第二种方式是在当前的 Package 下面配置一个默认的拦截器：

```
<default-interceptor-ref name="autowiring"/>
```

这个声明也是直接放在<package … /> 标签里面，但是只能有一个拦截器被配置为默认值。

现在拦截器已经被配置好了，每一次 Action 所映射的 URL 接到请求时，这个拦截器就会被执行。但是这个功能还是很有限的，因为在大多数情况下，一个 Action 都要对应有多个拦截器。实际上，由于 Struts2 的很多功能都是基于拦截器完成的，所以一个 Action 对应有 7、8 个拦截器也并不稀奇。可以想象的到，如果要为每一个 Action 都逐一配置各个拦截器的话，那么我们很快就会变得焦头烂额。因此一般我们都用拦截器栈（interceptor stack）来管理拦截器。下面是 struts-default.xml 文件中的一个例子：

```
<interceptor-stack name="basicStack">
```

```
<interceptor-ref name="exception"/>
<interceptor-ref name="servlet-config"/>
<interceptor-ref name="prepare"/>
<interceptor-ref name="checkbox"/>
<interceptor-ref name="params"/>
<interceptor-ref name="conversionError"/>
</interceptor-stack>
```

这个配置节点是放在<package … />节点中的。每一个<interceptor-ref … />标签都引用了在此之前配置的拦截器或者是拦截器栈。

我们已经看到了如何在 Action 中应用拦截器，而拦截器栈的用法也是一模一样的，而且还是同一个标签：

```
<action name="my" class="com.fdar.infoq.MyAction" >
<result>view.jsp</result>
<interceptor-ref name="basicStack"/>
</action>
```

默认拦截器的情况也是一样的——只需要把单个拦截器的名字换成拦截器栈的名字就可以了。

```
<default-interceptor-ref name="basicStack"/>
```

由上面的种种情况可以得出，当配置初始的拦截器与拦截器栈时，必须要确保它们的名字是唯一的。

实现拦截器

在应用程序中使用自定义的拦截器是一种优雅的提供跨应用特性的方式。我们只需要实现 XWork 框架中一个简单的接口，它只有三个方法：

```
public interface Interceptor extends Serializable {

void destroy();

void init();

String intercept(ActionInvocation invocation) throws
Exception;
}
```

如果我们不需要执行其他初始化或清理动作的话，还可以直接继承 `AbstractInterceptor`。这个类对“destroy”和“init”方法进行了重写，但在方法中没有执行任何操作。

`ActionInvocation` 对象可以用来访问运行时环境，以及 Action 本身；上下文（包括了 Web 应用的请求参数，session 参数，用户 Local 等等）；Action 的执行结果；还有那些调用 Action 的方法并判断 Action 是否已被调用。

在上一小节中我们知道了配置拦截器和配置拦截器栈的方式是相同的，如果你需要创建自己的拦截器的话，那么也应当考虑一下创建自定义的拦截器栈。这样才能确保新的拦截器在应用的时候可以贯穿所有需要该拦截器的 Action。

值栈与 OGNL

本节所要介绍的两部分内容是密切相关的。值栈的含义正如它的名字所表示的那样——对象所组成的栈。OGNL 的全称是 Object Graph Navigational Language（对象图导航语言），提供了访问值栈中对象的统一方式。

值栈中的对象构成及其排列顺序如下所示：

1. **临时对象**——在执行过程中，临时对象被创建出来并放到了值栈中。举个例子来说，像 JSP 标签所遍历的对象容器中，当前访问到的值就是临时对象
2. **模型对象**——如果模型对象正在使用，那么会放在值栈中 action 的上面
3. **Action 对象**——*正在被执行的 action*
4. **固定名称的对象（Named Objects）**——这些对象包括有 `#application`，`#session`，`#request`，`#attr` 和 `#parameters`，以及相应的 servlet 作用域

访问值栈可以有很多方法，其中最常用的一种就是使用 JSP，Velocity 或者 Freemarker 提供的标签。还有就是使用 HTML 标签访问值栈中对象的属性；结合表达式使用控制标签（例如 if，elseif 和 iterator）；使用 data 标签（set 和 push）来控制值栈本身。

在使用值栈时，我们无须关心目标对象的作用域。如果要使用名为“name”的属性，直接从值栈中进行查询就可以了。值栈中的每一个元素，都会按照排列顺序依次检查是否拥有该属性。如果有的话，那么就返回对应的值，查询结束。如果没有的话，那么下一个元素就会被访问……直到到达值栈的末尾。这个功能非常强大，我们根本不需要知道所需要的值在什么地方——存在于 Action，模型或是 HTTP 请求中——只要这个值存在，它就会被返回。

但它也有个缺点。如果所请求的是很常见的属性（例如“id”），而你想要从某个特定的对象中（例如 action）获取该属性的值，这时候值栈中第一个满足条件的对象返回的属性值就可能不是所想要的结果了。返回结果的确是“id”属性的值，但它可能来自 JSP 标签，临时对象或者模型对象。这时候就需要用到 OGNL 来增强值栈的功能了。OSGL 并不仅限于访问对象属性，如果我们知道某个 action 在值栈中的深度，那么就可以用“[2].id”来替换掉“id”。

实际上OGNL是一套完整的表达式语言。在OGNL里面，可以用“.”来遍历对象图（比如说，使用“person.address”而不是“getPerson().getAddress()”），它还提供了类型转换，方法调用，集合的操作与生成，集合间的映射，表达式运算和lambda表达式。OGNL的网站上提供了一套完整的指南，地址为 http://www.ognl.org/2.6.9/Documentation/html/LanguageGuide/index.html。

结果类型

在前面我们已经演示过如何配置 Action 来向用户返回一个 JSP。但这只是 Action 的结果之一。Struts 2 里面提供了多种结果类型，有些是可见的，有些只是与运行环境之间的交互。

“type”属性被用来配置 Action 的结果类型，如果没有配置该属性的话，那么默认类型就是“dispatcher”，它将会渲染一个 JSP 结果并返回给用户。下面就是这样的 Action 配置：

```
<action name="my" class="com.fdar.infoq.MyAction" >
<result type="dispatcher">view.jsp</result>
</action>
```

配置

结果类型的配置项在<package … />标签内部，和拦截器的配置看上去很相似。“name”属性提供了结果类型的统一标识，“class”属性描述了实现类的全限定名。它多出了第三个属性：“default”——通过这个属性可以修改默认的结果类型。如果一个 Web 应用是基于 Velocity 而不是 JSP 的话，那么把默认结果类型修改一下，就可以节省很多配置的时间。

```
<result-types>
<result-type name="dispatcher" default="true"
       class="org.apache.struts2.dispatcher.ServletDis
   patcherResult"/>
<result-type name="redirect"
       class="org.apache.struts2.dispatcher.ServletRed
   irectResult"/>
…
</result-types>
```

实现结果类型

和拦截器一样，你也可以创建自己的结果类型并在 Web 应用中加以配置。Struts 2 中已经有了很多常用的结果类型，所以在创建自己的结果类型之前，你应该检查一下你想要的类型是否已经存在。

创建一个新的结果类型需要实现 Result 接口。

```
public interface Result extends Serializable {

public void execute(ActionInvocation invocation)
       throws Exception;
}
```

ActionInvocation 对象可以用来访问运行时环境，新的结果类型还可以通过它来访问刚刚执行的 Action 以及 A ction 执行的上下文。在上下文中包括有 HttpServletRequest 对象，可以用来访问当前请求的输入流。

结果和视图技术

在上面的所有示例中，我们假设了所使用的视图技术都是对 JSP 进行渲染，虽然这是最常用的方式，但并不是渲染结果的唯一方式。

结果类型是与所使用的视图技术联系在一起的。我们在前面的小节中看到过，如果没有“type”属性或者该属性的值是“dispatcher”的话，那么最后就会渲染 JSP 页面。在 Struts 2 应用中，有其他三种技术可以用来替代 JSP：

- Velocity Templates
- Freemarker Templates
- XSLT Transformations

另外，你还可以为任何一种已有的视图技术实现一种新的结果类型，这样就可以得到另外的结果了。

抛开各自的语法差异不谈，Freemarker 和 Velocity 还是和 JSP 很相似的。Action 的所有属性都可以被模板访问（通过 getter 方法），就像使用 JSP 标签库和在标签库中使用 OGNL 一样。在 action 的配置文件中，只需要把 JSP 模板的名字换成 Velocity 或者 Freemarker 模板的名字就可以了。下面的配置文件中，用 Freemarker 代替了 JSP 的返回结果：

```
<action name="my" class="com.fdar.infoq.MyAction" >
<result type="freemarker">view.ftl</result>
</action>
```

XSLT 结果和其他的有所不同。它不是用 stylesheet 名代替模板名，而是使用了另外的参数。“stylesheetLocation”参数提供了用于渲染 XML 文件的 stylesheet。如果没有为该参数赋值的话，那么返回给用户的就是未经转换过的 XML 文件。

“exposedValue”属性用来提供将要作为 XML 来展现的 Action 属性或是 OGNL 表达式。如果该参数没有赋值的话，那么 Action 本身就会被作为 XML 显示出来。

```
<result type="xslt">
    <param
name="stylesheetLocation">render.xslt</param>
    <param name="exposedValue">model.address</param>
</result>
```

在使用 XSLT 作为结果的时候，还可以使用“struts.properties”来进行属性配置。该属性为“struts.xslt.nocache”，它决定了 stylesheet 是否会被缓存。在开发的过程中，你可能需要移除所有的缓存，以提

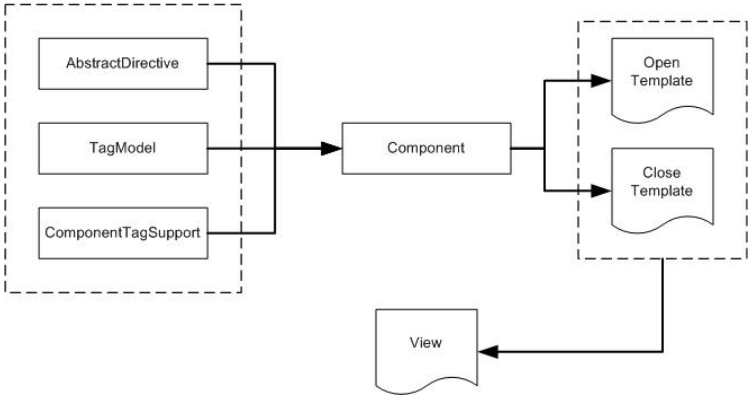
高开发速度。但是当应用被部署到产品环境以后，被缓存的 stylesheet 就会提高渲染性能。

标签库

标签库常用来定义一些提供了可重用性的 JSP 专属特性。在 Freemarker 和 Velocity 中虽然没有同样的概念，但却提供了渲染引擎的模型或者上下文，以及可以访问这些对象的被渲染的模板。当我们讨论 Struts 2 中的标签库时，我们实际上讨论的是与 JSP 标签库一样提供了相同功能的对象，它们可以被所有的视图技术——JSP、Velocity 和 Freemarker——访问。

在这里定义标签库时，步骤要更多一些，但是内在的功能是相同的——为对象内部的方法进行访问。通过这项功能，逻辑代码得到了更好的封装，同时也避免了到处的剪切和粘贴代码，从而提高了程序的可维护性。

JSP 的标签库中有另外一个今天看上去已经过时的特性——把要显示的文本放在标签库自己的 Java 代码中。Struts 2 中又重新拾起了这种思想，为标签专门创建了一种第二等的 MVC 模式。在该模式中，由 Java 类处理逻辑，但是页面渲染的工作放在了 Freemarker 模板中（默认情况下）。整体架构如下图所示：



这个架构的核心是一组组件对象。组件对象以最基本的形式来表示每一个标签，并且提供了所有必需的逻辑，例如管理和渲染模板。每一个不同的结果/视图技术都对组件进行了封装，在封装中把原始页面中的标签进行了转换，以满足特定的需要。

在结合 Freemarker 模板渲染使用标签的时候，还需要进行额外的配置。我们要在“web.xml”文件中配置一个 servlet，这样 Freemarker 才能得到渲染所需的信息：

```
<servlet>
      <servlet-name>jspSupportServlet</serlet-name>
      <servlet-class>
      org.apache.struts.action2.views.JspSupportServlet
.JspSupportServlet
      </servlet-class>
      <load-on-startup>10</load-on-startup>
</servlet>
```

每一个组件都有与其相关联的模板。如果原始的标签中还包含有其它标签（例如 form 标签），那么就会对应有一个开放的模板和封闭的模板。如果原始标签是自包含的（例如 checkbox 标签），那么就对应的就只有一个封闭的模板。就像把 UI 架构中的文本与逻辑分离一样，为标签提供模板也会带来额外的好处——开发人员可以为同样的标签来组合和匹配不同的模板，这种特性叫做“主题（themes）”

目前共有 3 个主题[6]：“simple”，“xhtml”和“css_xhtml”。“simple”主题只是简单输出标签，不做任何的格式调整。“xhtml”主题作出了一些格式处理；比如对于HTML的表单标签，这种主题使用HTML表格形式提供了两列的格式（即名称标签和输入域两列）。对CSS的信仰者来说，这里为他们提供了“css_xhtml”主题。与“xhtml”主题相似的是，这种主题也是提供了格式转换的功能，但是它用的不是HTML表格，而是CSS DIV。这种方式不会带来HTML的混乱。

“xhtml”和“css_xhtml”主题为开发者自己动手实践提供了很好的范例——通过实现一个主题来为 HTML 提供特定的格式。在同一个页面上，可以组合和匹配多种主题，当前标签所使用的主题是通过“theme”属性来定义的。如果在整个应用中，你只需要使用一个主题的话，那么就可以在“struts.properties”文件中，将“struts.ui.theme”属性的值配置为要使用的主题。

[6] 单纯从技术角度来讲，应该有 4 种主题——第 4 种是“ajax”主题。Struts2 已经决定在下个版本中把Ajax的功能从核心框架里移走，放到插件里面。所以在本节中没有对其进行介绍。

每一种标签（control 标签，data 标签，form 标签和 non-form 标签）都可以使用主题；但是如果要创建一个新主题的话，那么它只有对可视化的标签才能起到作用。

4

架构目标

对于一个特定的代码库而言，要谈它的架构目标是很困难的。通常在开发之前，其最终目标就写进了文档中；但这是理想化的状态，当开始开发之后，代码往往就会向着另外的方向发展。然后就有了代码库的典型特征；它们很难被发现，在不同的 package 或者特性之间有可能表现出不一致，并且更像是脱离计划之外的演变产物。

在本章中，我们将要讲述 Struts 2 代码库中五种这样的特征。从 2002 年的代码库演变起始——从最开始的 WebWork，到 WebWork 分离成为 WebWork2 和 XWork，再到最后的 Struts 2，这些架构思想一直延续到了今天。

概念分离

以下这些不同层次的功能都是 Web 开发人员应当了解的：

- 在请求/响应周期中，所完成的核心任务是执行特定于每个 Action 的逻辑功能。
- Action 要完成逻辑功能或者访问资源的话，必须要访问或者持有业务对象。
- 在把 HTML 中基于字符串的值转换成原始数据类型或其他对象类型，以及把视图对象转换成业务对象或者数据表表示的过程中，这期间需要完成多种转化，映射和变换。
- 有一些横切（cross-cutting）的目的是为成组的 action，或者应用中的所有 action 提供功能的。

在 Struts 2 的架构中，以上的每一种概念都是被单独分离的。功能和逻辑不再是 Action 独享的。让我们看一下上面提到的概念是如何被分离的：

- **每个 Action 的逻辑（Per-Action Logic）**——这是最简单的概念；每一个 Action 都要为它所提供的逻辑或功能负责。
- **访问/持有业务对象（Accessing/Obtaining Business Objects ）**—— Struts 2 使用了依赖注入机制来把 Action 所需的对象注入到 Action 中去。
- **转化/映射/变换（Translation/Mapping/Conversions ）**——这三个概念两两之间都有着细微的区别，但它们都为 Action 核心逻辑提供了辅助功能。类型的转化和变换是由框架本身完成的。在 Action 开始进行处理之前，HTML 中的字符串值就被转化成了基本类型，然后注入进 Action 里面——所需的东西就齐备了。映射是通过特定的拦截器完成的。我们需要通过某种方式对 Action 进行配置，让它拥有一个领域模型，并正确指定 HTML 中相应的字段，这样框架就会把 UI 与领域模型进行映射。它甚至还可以贯穿一个对象图。
- **横切的概念（Cross-cutting Concerns ）**——横切功能主要是由拦截器提供的。开发人员可以实现拦截器，然后让它们横切所有的 Action，横切特定 Package 中的 Action，或者还可以选择所要横切的 Action。 另外一个横切的概念是 UI 布局。在这里 Struts 2 提供了被称作“主题”的支持标签。不同的主题可以提供不同的布局选项，并可以应用到每一个独立的标签中，或是整个应用程序（把该主题设置为默认值）。

松耦合

WebWork 早期的目标之一就是提供一个松耦合的框架。WebWork2.0 版本更强化了这一点，把代码拆分成了两个项目：XWork——一个通用的命令框架；WebWork——XWork 的针对 Web 的接口。WebWork 在架构上做出的根本性变化创造了一种共生的关系。我们现在所知的“WebWork”实际上是 WebWork 和 XWork 的合并。

XWork作为一个独立的项目，可以作为其他项目的一部分加以利用——事实也是如此。Swingwork[7]就是这样一个项目。它是基于Swing的MVC框架，在底层使用了XWork。另外一个例子是一个JMS前端，它在Web UI中执行或共享XWork的action。这些都是高度松耦合的精彩示例。Struts 2 也是XWork的一个受益者。

松耦合的思想在 Struts 2 中得到了更好的传播，它贯穿了整个框架的始终——从最开始 Action 的处理过程到最后一步结束。实际上，在 Struts 2 中几乎没有什么是不可以被配置的——我相信这是 Struts 2 中最强有力的地方之一，但也是最虚弱的地方之一。

松耦合的配置有一些常见示例：

- 把 URL 映射到 Action
- 把不同的 Action 结果映射到被渲染的页面
- 把处理过程中发生的异常映射到被渲染的异常页面

稍微少见一些的 Struts 2 特有的示例有：

- 在不想用 Spring 的情况下配置业务对象工厂
- 改变 URL 与 Action 类的映射方式
- 添加新的 Action 结果类型
- 为新的框架功能添加插件
- 通过拦截器配置框架层的功能

松耦合系统的好处是广为人知的——增加易测试性，更容易扩展框架功能，等等。但是这里还有一点坏处。强大的配置管理功能，尤其是拦截器在其中至关重要的作用，使得开发人员已经很难理解某个 Action 的执行过程。这一点在调试时尤为突出。一个不了解内部流程的开发者很难提高调试的速度或效率，因为他根本就无法理解这中间发生了什么。这种问题可能简单如拦截器的配置错误，甚或是拦截器的配置顺序不对。如果我们能够充分理解处理过程中的每一个细节，那么很快就能找到解决方案了。

易测试性

[7] Swingwork 的地址为 https://swingwork.dev.java.net/。这个项目已经停止开发了。

在过去的几年里，单元测试已经成为了软件开发领域内的事实标准。测试不仅能够保证逻辑的正确性，而且在开发所要测试的类的时候（如果在此之前则更好），通过编写单元测试还可以得到更简洁强壮的设计。

Struts2 的前身——WebWork，就是在这种环境下开发的。在与框架元素保持松耦合的情况下，测试变得更加简单。Action、拦截器、结果、对象工厂以及在 Web 应用开发过程中开发出的其他组件，都可以不依赖于框架进行测试。

因为 Action 和拦截器是最常用的，所以下面我们来看一下怎么样对它们进行测试。

Actions

Actions 在框架中一般都是通过“execute()”方法被调用的，在更改配置以后，任何返回 String 类型值的方法都可以用来调用 Action。站在易测试性的角度来看，这已经无法更简单了。

我们先来看一个例子。下面这个 action 被用来增加一个数值：

```
public class MyAction {

 private int number;
 public int getnumber() { return number; }
 public void setNumber( int n ) { number = n; }

 public String execute() {
     number += 10;
     return “success”;
 }
}
```

因为 Action 都是 POJO，所以单元测试只需要实例化一个 Action，调用其中的方法，然后确保返回的结果与所期待的一致。Action 中的数据资源都是通过 setter 方法来提供的。所以 Action 中所需的数据都可以被直接赋值。

在我们的例子中需要两条断言——一个用来判断“execute”方法的输出，一个用来验证我们所期望的 Action 的状态。下面是单元测试代码：

```
public class myActionTest extends TestCase {
```

```
…

public void testExecute() {

     MyAction action = new MyAction();
     Action.setNumber(5);
          assertEquals("success", action.execute());
          assertEquals(15,action.getNumber());
}
}
```

对于资源来说情况就复杂一些了。我们可以用类似于JMock[8]的第三方库来提供资源的mock实现，然后测试Action与资源之间的交互是否正确。

虽然例子是用 JUnit 来写的，但是我们同样可以使用 TestNG 或者其他的测试框架。

拦截器

在构建拦截器时，测试会稍稍复杂一些。但是在代码框架中也为其提供了额外的帮助。下面是两个使用拦截器的场景。

第一个是拦截器在使用时与 ActionInvocation 对象交互。在执行完以后，你可以通过断言拦截器本身的状态来对逻辑处理进行验证。在这个场景中，拦截器的测试方式和 action 一模一样。实例化拦截器；创建一个 ActionInvocation 的 mock 实现，用它来测试拦截器；调用 intercept 方法；然后断言所期望发生的变化。所断言的可能是拦截器本身，也可能是方法调用的结果，抑或是可能会抛出的异常。

第二个场景是拦截器与运行环境或者是与拦截器栈中的其他拦截器交互。这种情况下，测试需要通过 ActionProxy 类与 Action 交互，同时断言也需要访问拦截器本身没有权限访问的其他环境对象。

XWork 库为 JUnit 测试提供了 XWorkTestCase 类，为 TestNG 测试提供了 TestNGStrutsTestCase 和 TestNGXWorkTestCase，用以帮助测试拦截器。它们都为 ConfigurationManager，Configuration，Container 和 ActionProxyFactory 类的实例提供了测试实现。XWork

[8] 请参见http://www.jmock.org以获得更多信息。

同时还提供了其他一些类，比如 XWorkTestCaseHelper 和 MockConfiguration。

现在我们已经有了建立测试环境的底层架构，测试本身就变得容易了——只需要按照第一个场景中的步骤执行就可以了。唯一的区别就是在调用拦截器的 intercept()方法时，同时还需要调用 ActionProxy 的 execute 方法，例如下面的代码：

```
        ActionProxy proxy =
        actionProxyFactory.createActionProxy(NAMESPACE,
    NAME,null);

        assertEquals("success", proxy.execute());
```

在这个场景中，测试代码可以断言的期望值包括：Action 的结果，Action 的值或是值栈中的值。下面的方法可以在 Action 的执行前或者执行后获取这个 Action 对象：

```
        MyAction action =
    (MyAction)proxy.getInvocation().getAction();
```

下面的方法可以用来得到值栈：

```
        proxy.getInvocation().getStack()
```

模块化

当应用的规模变大以后，把程序分拆成各个模块的重要性就不言而喻了。这样一来，我们就可以把一个项目中开发的功能或者新的框架特性打包，并且在其他项目中重用。Struts 2 已经把模块化作为了体系架构中的基本思想，开发人员可以独立工作，然后在其他人的工作基础上进行构建。

下面是将应用程序模块化的几种方法：

- **将配置信息拆分成多个文件**——它本身不会对应用程序的分包造成影响，但是配置信息按照功能的界限进行拆分以后，管理起来就容易了很多，也减小了开发的难度
- **把自包含的应用模块创建为插件**——所有用来提供某一种特性的功能组成——包括Action、拦截器、拦截器栈、视图模板（除了JSP以外）等等——都可以被打包在一起，并

作为独立的插件发布。例如配置浏览器插件[9]，该插件提供了一个完整的模块，该模块可以为所在的应用程序添加一个用来查看配置信息的Web接口。

- **创建新的框架特性**——与特定应用无关的新功能可以组织成插件，应用到多个应用中。

从技术角度来讲，这几种模块化的方式其实是一样的——它们都有同样的配置元素（除了名字可能不同以外，在使用插件时，“struts-plugin.xml”文件会作为配置文件被自动加载），同样的目录结构，并且可以包括同样的框架和应用元素。

上面的两种插件类型的唯一区别在于你从怎么概念上去理解它们，在发行包中放入的是哪些元素和配置文件。

额外的配置元素

因为插件可以为内部的框架功能提供替代的实现方式，所以需要做一些额外的配置。这些配置元素可以在“struts.xml”配置文件中使用，也用在了“struts-default.xml”文件中，但是它们更常用的情况是用来配置插件。

在插件中，替代的实现方式是分两步来配置的：

1. 使用 `<bean … />`标签来定义替换的接口实现，并用一个唯一的 key 来标识
2. 使用`<constant … />`标签来在多种可能的接口实现中进行选择

下面我们来看看每一步的具体细节。

`<bean … />`标签可以用来为扩展点提供实现信息。下面的例子是“struts-default.xml”配置文件中一个对象工厂的配置：

```
<bean name="struts"
type="com.opensymphony.xwork2.ObjectFactory"
class="org.apache.struts2.impl.StrutsObjectFactory" />
```

在配置项的属性中包含了用来在 Struts 2 里创建和使用替代对象实现（alternate object implementation）的所有信息。这些属性包括：

- class ——类的全名

[9] config browser插件的文档地址为http://struts.apache.org/2.x/docs/config-browser-plugin.html

- type ——类所实现的接口
- name ——每个 type 的唯一简称
- static ——是否把静态对象方法注入到类实例中
- scope ——实例的作用域，可以是“default”，“request”，“session”，“singleton”或“thread”
- optional—— 当值为“true”时，即使创建类实例的过程中发生错误，也会继续装载。

然后，开发人员可以用<constant … />标签来选择使用哪一个配置。这里只有两个属性——“name”属性提供了扩展点的名字，这个扩展点是在新的实现中会被更改的，“value”属性的值是在<bean … />标签中配置的唯一标识。

```
<constant name="struts.objectFactory" value="plexus"
/>
```

<constant … />标签是把一个新值赋给已知属性的一种方式，但并不是唯一的方式。新值还可以在“web.xml”文件的“init-param”中修改，或者作为一个名-值对写入“struts.properties”配置文件。

如果你不是在开发插件，而是在普通的“struts.xml”文件中使用这些技术，那么还有捷径可走。把一般来说会放在<bean … />标签中的类名放进在<constant … />标签中——这样就避免了使用<bean … />标签。

下表列出了可配置的扩展点对应的接口与属性名。

接口	属性名	Scope	Description
com.opensymphony.xwork2.ObjectFactory	struts.objectFactory	singleton	创建在框架中用到的对象——Action，结果，拦截器，业务对象等等。
com.opensymphony.xwork2.ActionProxyFactory	struts.actionProxyFactory	singleton	创建 ActionProxy
com.opensymphony.xwork2.util.	struts.objectTypeDeterminer	singleton	判断一个 map 或者 collection

ObjectTypeDeterminer			中的 key 和元素是什么
org.apache.struts2.dispatcher.mapper.ActionMapper	struts.mapper.class	singleton	从请求中得到 ActionMapping 并从 ActionMapping 中得到 URI
org.apache.struts2.dispatcher.multipart.MultiPartRequest	struts.multipart.parser	per request	解析 多部件 请求（文件上传）
org.apache.struts2.views.freemarker.FreemarkerManager	struts.freemarker.manager.classname	singleton	载入和处理 Freemarker 模板
org.apache.struts2.views.velocity.VelocityManager	struts.velocity.manager.classname	singleton	载入和处理 Velocity 模板

`<constant … />` 标签和 “web.xml”文件中的“init-param”不仅仅可以用来定义扩展点属性。“struts.properties”文件中的所有属性都可以通过二者进行修改。

惯例重于配置

惯例重于配置是 Rails 带入主流应用开发中的概念。它不是提供那些对于各个应用而言都很相似的配置文件，而是假定在*绝大多数*情况下，开发人员都会遵守特定的模式。这种模式具有足够的通用性，所以可以被认为是一种开发惯例，框架会默认使用这种模式，而不是为每一个新的应用都提供配置。在默认情况下，开发人员就不必再管理种种配置信息了。如果有的需求与惯例的配置信息不同，那么还可以根据需求进行修改，把默认模式覆盖掉。

Struts 2 采用了这个概念。松耦合在给 Struts 2 带来高度灵活性的同时，也带来了配置上的高度复杂性。惯例在这二者之间做出了平衡，为我们提供了简洁而高效的开发者体验。

Struts 2 中“惯例重于配置”的应用可以通过以下几个例子来说明：

- **隐式的配置文件加载**——不需要显式配置“struts-default.xml”和“struts-plugin.xml”（对每一个插件而言）文件，它们会被自动加载。
- **Code Behind 插件**——在使用 code behind 插件时，它会混合使用 action 名和结果字符串在结果模板中进行自动搜索，所以“/user/add.action” 这个 action 的“success”结果会返回“/user/add-success.jsp”结果模板，“error”结果会返回“/user/add-error.jsp”结果模板。
- **默认的结果和结果类型**—— 在配置 Action 的时候，如果使用默认的“success”结果和 JSP 结果类型的话，就不需要对它们进行配置
- **织入（Wiring）Spring 业务服务**——在安装了 Spring 框架插件以后，就不必为每个 Action 所需的 Spring 提供的业务服务进行配置，这些 业务服务会被自动织入到 Action 里。

在前面的章节里，我们看到了很多默认的设置，包括如何通过配置来重写属性值和提供新的默认设置。在下面一章中，我们将要介绍的是如何通过更多的配置项和惯例来提高生产力。

5 提高效率技巧

本章所要介绍的是一些在使用 Struts 2 开发 web 应用时，能够提高生产效率的技巧、技术与特性，这些都是开发人员应当牢记的。这些 tip 范围很广，包括从列出默认值，到说明应当实现的接口，以及如何创建自定义的声明式验证。

本章中描述的信息只是一个简单的介绍——如果你对哪一个tip感兴趣，就不妨深入研究一下。你可以参考Struts2 的文档 http://struts.apache.org/2.x/docs/guides.html，或者是通过搜索来看看其他开发人员的想法，以及他们是如何进行应用的。

最后，当你阅读每一个小节的时候，都应该想一下当前的 tip 应该如何与其他 tip 配合使用，它与其他 tip 的区别是什么，你又该怎样把它应用到 web 开发中去。这样你对它的理解才能更上一个台阶。

Struts2 的插件体系结构保证了它的可持续发展。你需要常常访问 Struts2 的插件注册页面以获得最新的开发进展。这个页面中包括了所有的插件相关的声明，地址为 http://cwiki.apache.org/S2PLUGINS/home.html。它上面已经有了很多第三方的插件，例如JSON，GWT和Spring WebFlow。

重用 Action 的配置

我们曾经讲过把 action 以 package 的组织形式来配置管理，并且描述了 package 如何继承其它的 package——但是这样做所带来的好处还没有说清楚。下面我们来看一个完整的例子。这个应用程序用来向游客提供动物园信息。每个洲都会对应有一个 portal 页面，提供动物，地图等等数据。

完成这项功能的一种方式是让用户可以调用类似“www.myzoo.com/home.action?continent=asia”这样的URL。这样强化了应用程序的逻辑性，并且可以很容易判断现在所请求的是哪个洲的信息。但是，当我们依赖于要渲染给用户的信息的定义方式时，灵活性就丧失了，因为硬编码的路径信息被添加到了action或者视图中。

灵活一点的方式是提供类似“www.myzoo.com/asia/home.action ”这样的URL。在使用这种方案时，我们可以提供一个action基类，并配置到默认的package下面。每一个继承的package都可以访问这个action基类。所以如果不进行额外配置的话，“www.myzoo.com/home.action”实际上调用的就是“www.myzoo.com/asia/home.action”。

更进一步来说，视图也可以不用任何额外的配置来进行自定义。如果默认的 package 下面的 action 是这样的：

```
   <action name="home" class="com.fdar.infoq.HomeAction"
>
   <result>portal.jsp</result>
   </action>
```

那么Struts2 所渲染的JSP就会依赖于URL所提供的，用户调用的命名空间。于是如果URL为“www.myzoo.com/home.action”，那么“/portal.jsp”就会被渲染，如果URL为“www.myzoo.com/asia/home.action”，那么被渲染的页面就是“/asia/portal.jsp”。强调一下，这个功能并没有进行多余的配置——因为配置信息提供的是一个JSP相对路径而不是特定的路径名。

在配置中使用模式匹配调配符

有的时候，action 的配置文件大小会以令人难以置信的速度增加。为了避免这种现象，我们需要使用模式匹配。它的工作方式是定义一个或多个模式，而 URL 会与这些模式保持一致。

举个例子会更容易说明这一点。假设你的应用中 URL 的模式是这样的：“/{module}/{entity}/{action}.action”。这是很常见的模

式，对应的 URL 可能是：“/admin/User/edit.action”，“/admin/User/list.action”和“/admin/User/add.action”。

在类的配置中，会有一个 Struts2 的 action 类，名字为“{entity}Action”，而每一个{Action}都是 action 类里面的一个方法。所有对 action 中方法的调用要么返回一个被显示的实体的更新页面，要么返回所有被显示的实体的一个列表。

在我们的例子中，“struts.xml”配置就是这样的：

```
<action name="*/*/*" method="{3}"
       class="com.infoq.actions.{1}.{2}Action">
 <result name="view">/{1}/update{2}.jsp</result>
 <result name="list">/{1}/list.jsp</result>
</action>
```

在 action 的名字中，每一个星号都是一个通配符。在这个例子中，我们全都用了星号——其实也不必如此。比如说，如果你想要把所有对实体的 view 动作映射到一起，那么类似于“name="/*/View*”这样的配置就能完成要求。形如{1}，{2}等等的标识符用来获取通配符所对应的值（数字表示了所要获取的值对应的通配符的位置，顺序是从左到右）。

在“struts.properties”文件中（或者使用“struts.xml”的 constant 标签），你需要确保正确配置了下面的属性：

```
struts.enable.SlashesInActionNames = true
```

这个属性设为 true 以后，action 的名字中就可以使用斜杠了。Struts2 的默认设置是不允许 action 的名字中出现斜杠的，需要用 package 来分割命名空间。

最后，如果你是提供验证和转换的属性文件而不是注解的话，那么就没有什么捷径了。我们会在下面的部分谈到这一点。每一项都要包括 action 的全名和所需的扩展：例如，前面的例子就要包括“edit-validation.xml”和“edit-conversion.xml”，以及“com.infoq.actions.admin”这个 package。

使用替代的URI映射方法

我们除了在配置中使用通配符以外，还可以使用自定义的映射方案，改变从 URI 到 action 和被调用方法的映射关系。用这种方法可以减少配置信息，并且可以在整个应用中保持一致。你可以把 URI 模式与 session 信息或其他任何你能想到的信息合并，来决定要调用哪一个 action。

这种方法需要实现 ActionMapper 接口，在该接口中有两个要实现的方法：一个是 getMapping()，它把 URI 转换成已知的配置；另一个是 getUriFromActionMapping()，它把一个 action 配置转换成 URI。如下所示：

```
public interface ActionMapper {

ActionMapping getMapping(
       HttpServletRequest request,
       ConfigurationManager configManager);

String getUriFromActionMapping(ActionMapping mapping);
}
```

`ActionMapping` 类中提供了要调用的 action 的命名空间，名称，方法，结果和参数。`ConfigurationManager` 可以用来访问配置信息的提供者（然后就可以进行进一步的自定义）。

Action mapper 需要在“struts.xml”中进行配置后才能被使用。Name 和 type 属性的值是不变的——只需要把 value 属性的值改成自定义的 ActionMapper 的实现类。

```
<constant name="struts.mapper.class"
        value="com.fdar.infoq.MyActionMapper" />
```

值得高兴的是我们不需要在所有用到的地方都去实现 ActionMapper 这个接口，Struts2 已经有了几种不同类型的实现。

`Restful2ActionMapper`类提供了一个ReST型接口的实现，它借鉴了Ruby on Rails里易于使用的URI的设计思路。要注意的是，在 Struts2 的文档中，这个实现还只是一个试验品。

`Restful2ActionMapper` 首先做的事情是判断要使用的命名空间和 action，与你所想的一样——URL 的最后一个元素是 action 名，之前的都是命名空间。但是也有例外，因为属性也会通过 URI

传入。把 action 名，属性名和属性值映射到 URI 中的模式是这样的：

```
http://HOST/PACKAGE/ACTION/PARAM_NAME1/PARAM_VALUE1/PARAM_NAME2/PARAM_VALUE2
```

“PARAM_NAME/PARAM_VALUE”这样的名/值对的数量是没有限制的，如果PARAM_NAME1 是“id”的话，那么URI可以简化为：

```
http://HOST/PACKAGE/ACTION/PARAM_VALUE1/PARAM_NAME2/PARAM_VALUE2
```

当 action 被确定下来以后，下一步就是要找到需要调用的方法。这可以使用 HTTP 方法来做出决定。因为 HTML 不支持 PUT 和 DELETE 方法，所以要有另外的请求属性“__http_method”来提供方法信息。

下面是 HTTP 方法和 URL 合并以后的结果：

- GET: “/user” ——当 action 被单独使用时，“index”方法会被调用
- GET: “/user/23” ——当 action 与参数名/值对一起使用时，“view”方法会被调用，在这里“id”属性的值被设为“23”
- POST: “/user/23” ——这里 HTTP 方法是 POST 而非 GET，于是“create”方法将会被调用，“id”或者其他用于标识的值会被包含在 URL 中，而包含更新信息的名-值对会放在 POST 数据中
- PUT: “/user” ——“update”方法会被调用，与 POST 场景类似，包含信息的名-值对会放在 POST 数据中而不是 URL 里面
- DELTE: “/user/23” ——“remove”方法会被调用，在 URL 里面提供了唯一的标识符（在这里是“id”，其值为“23”）
- GET: “/user/23!edit”——“!”被用来描述方法名，所以这里会被调用的是“edit”方法
- GET: “/user/new” ——“new”后缀表示“editNew”方法会被调用

这里还有一个 CompositeActionMapper 类。这个实现可以帮助你把多个独立的 ActionMapper 实现类串连起来。这个串连序列中的每一个类都会被检查，看它是否能够解析 URI。如果 URI 被解析成功，那么就返回结果，如果没有，序列中的下一个类就会被继续检查；如果最终也没有找到匹配的结果，就会返回一个 null。

和一般的 ActionMapper 配置一样，CompositeActionMapper 配置中包含有 constant 节点，它列出了要被串连在一起的 ActionMapper 的实现类的名字。

```
   <bean name="struts"
        type="org.apache.struts2.dispatcher.mapper.Action
Mapper"
        class="org.apache.struts2.dispatcher.mapper.Compo
siteActionMapper" />

   <constant name="struts.mapper.composite"
        value="org.apache.struts2.dispatcher.mapper.Defau
ltActionMapper,

        org.apache.struts2.dispatcher.mapper.RestfulActio
nMapper" />
```

了解拦截器的功能

拦截器在 Struts2 框架中起到了至关重要的作用。通过掌握已有的拦截器，你就可以理解在 action 处理过程中的每一个步骤了。

另外一个好处会在调试 action 的时候呈现出来。有时候 action 并没有那些应有的数据。这常常都是由于拦截器没有被正确应用，或者是拦截器应用的顺序有问题导致的。通过掌握每一个拦截器的作用，问题的定位和解决就容易了很多。

下面是一些可以立刻使用的拦截器，并提供了对应的功能描述。

拦截器名	描述
alias	将同一个参数在请求之间进行命名转换。
chain	使上一个 action 的属性对当前 action 可用。常与<result type="chain">结合使用（在上一个 action 中使用）。

conversionError	把 ActionContext 的转换错误添加到 Action 的字段错误中。
createSession	自动创建一个 `HttpSession`，当某一个拦截器需要 HttpSession 协同工作时（如 `TokenInterceptor`）是很有用的。
debugging	提供多个不同的调试场景，用来查看页面背后的数据结构信息。
execAndWait	在后台执行 action，并发送给用户一个等候页面。
exception	把异常映射到对应的结果。
fileUpload	为便捷的文件上传提供支持。
I18n	记忆用户 session 所使用的 locale。
logger	输出 action 名。
model-driven	如果 action 实现了 ModelDriven，该拦截器会将 getModel 方法的结果放到值栈上面。
scoped-model-driven	如果 action 实现了 `ScopedModelDriven`，该拦截器会从 scope 中获取并存储模型，然后通过 setModel 方法将其赋给 action。
params	把请求参数赋给 action。
static-params	把“struts.xml”定义的参数传给 action，它们是<action … />标签下的<param … />子标签。
scope	提供了将 action 状态存储在 session 或 application scope 中的简单机制。
servlet-config	提供了对表示 `HttpServletRequest` 和 `HttpServletResponse` 的 Map 的访问。
timer	输出 action 的执行时间（包括内嵌的拦截器和视图）。
token	验证 action 中的有效 token，避免提交重复的表单。
token-session	和 token 拦截器一行，但是在处理无效的 token 时，将提交的信息保存在 session 中。
validation	使用在 *action-validation.xml*中定义的验证器来进行验证。
workflow	调用 action 中的 validate 方法。如果有异常的话，就会返回 INPUT 视图。
store	从 session 中获取和保存实现了 ValidationAware 的 action 的消息/异常/字段异常。

checkbox	添加 checkbox 的自动处理代码，可以检测没有被选中的 checkbox，并给它一个默认值（通常是 false），然后作为参数添加。它使用了一个特定名称的隐藏字段来检查未提交的 checkbox。未选中的默认值可以被那些不是 boolean 值的 checkbox 重写。
profiling	激活对参数的性能监测
roles	只有当用户具有正确的 JAAS 角色时才能够执行 action。
prepare	如果 action 实现了 Preparable，那么就调用它的 prepare()方法。

使用提供的拦截器栈

拦截器栈可以将那些应用于各类 action 的拦截器进行功能分组。拦截器栈构建用于 CRUD 操作，验证 action 输入，或是其他任何你所需的功能。但是在创建自己的栈以前，最好先来看一下 Struts2 提供的拦截器栈。很多标准配置都已经配置完成并可以即刻使用了。另外，每一个插件都可以提供自己的拦截器栈，如果要使用插件所提供的功能的话就要使用插件的拦截器栈。

有两种方法可以使用框架提供的拦截器栈——或者把 action 放到提供了拦截器栈（通过零配置的注解或者“struts.properties”中的 constant）的 package 下面，或者让定义的新 package（其中包括了你创建的 action）继承提供了拦截器栈的 package：

```
<package name="mypackage"
extends="struts-default" namespace="/mypackage">

 …
</package>
```

我们曾经说过，在把应用程序部署为产品之前，你需要检查一下拦截器栈，看看你是否需要其中的某/每一个拦截器。“paramsPrepareParamsStack”和“defaultStack”里面包含有“chain”，“il8n”，“fileUpload”，“profiling”和“debugging”这几个拦截器。它们都是不怎么常用的，我们可以把它们移走，减少不必要的处理工作。

栈	Description
basicStack	Struts2 提供的最常用的栈。提供了异常处理，将 HTTP 对象和请求/表单参数注入 action，和处理转换错误的功能。
validationWorkflowStack	向 basic 栈中添加验证和工作流的功能。
fileUploadStack	向 basic 栈中添加对自动文件上传的支持。
modelDrivenStack	向 basic 栈中添加对模型驱动 action 的支持。
chainStack	向 basic 栈中添加对 action 串连的支持。
i18nStack	向 basic 栈中添加国际化的功能。
paramsPrepareParamsStack	这是 Struts2 提供的最复杂的一个栈。它的应用场合是当 action 的 prepare()方法被调用时，用传入的请求参数载入数据（或者执行其他任务），然后重新用请求参数来重写一些载入的数据。它的一个典型应用是更新对象。用 id 来从数据库中读取对象，然后用从请求中传入的参数重写一些读取的数据。
defaultStack	这是默认的栈。它为绝大多数的应用场景提供了所有的功能。实际上，它包括了核心发布版中几乎所有的可用拦截器。
completeStack	这个栈提供了“defaultStack”的别名，用来向后兼容 WebWork 的应用。
executeAndWaitStack	向 basic 栈中添加异步支持 action 的功能。

利用返回类型

返回类型使得开发者可以对要渲染给用户的元素进行组合和匹配。实际上，一个 action 可以有多个结果，而且每个结果都可以有不同的结果类型。

另外一点也需要注意：结果可以是可见的，也可以是不可见的。比如说，我们可以返回 HTTP 头。

下面是一些已经预配置好的结果类型和一些简要说明。如果你把 action 放到“struts-default”package 下面，或者是继承这个 package，这些结果类型就可以使用了：

名称	描述
chain	将一个 action 的执行与另外一个配置好的 action 串连起来。用第一个 action 的 getter 方法和第二个 action 的 setter 方法来完成 action 之间属性的复制。
dispatcher	渲染 JSP。这是默认的结果类型，如果在 action 配置中没有配置其他的结果类型，它就会被使用。
freemarker	渲染 Freemarker 模板。
httpheader	返回 HTTP 头和用户定义的值。
redirect	重定向到任意的 URL。
redirect-action	重定向到配置好的 action。可以用来提供 post 完成以后的重定向功能。
stream	将数据流返回给浏览器。可以用来把数据注入 PDF，Microsoft Work，图像或其他数据中。
velocity	渲染 Velocity 模板。
xslt	使用 XSLT 来转换已执行完毕的 action 的属性格式。

利用数据转换

Web 开发的一个常见任务就是把基于字符串的表单数据转换成适当的类型，交给模型或者业务服务方法使用。通常这是繁重的人工处理过程。Struts2 提供了数据转化功能，将这个过程做了简化。内置的转换机制可以将 String 转化成下面任意一种格式：

- `String`
- `Boolean` 或 `boolean`
- `Character` 或 `char`
- `Integer` 或 `int`
- `Float` 或 `float`
- `Long` 或 `long`
- `Double` 或 `double`
- `Date` ——使用与当前请求相关联的 locale

这样 action 中的 setter 方法就可以从“setId(String id)”改成“setId(int id)”了。我们再也无须为每一个值进行类型转换，因为它们会以正确的类型传给 action，我们只需简单使用即可。

要进行自定义的类型转换时，你需要实现 StrutsTypeConverter 这个类。里面有两个需要实现的方法；一个是把字符串转换成新类型的类，一个是把新的类型转换成字符串。

```
   public class MyTypeConverter extends
StrutsTypeConverter {

   public Object convertFromString(
                      Map context, String[] values,
                   Class toClass) {
        …
    }

   public String convertToString(Map context, Object o) {
    …
   }
   }
```

如果在类型转换时出现问题，那么应该抛出一个 TypeConversonException，告知框架转换无法完成。

在 action 中使用新的转换器之前，需要对它进行配置。这里可以使用注解（下一节会进行解释）或是使用一个单独的“*-conversion.properties”文件。如果正在使用转换功能的 action 叫做“MyAction”，那么你需要在同一个 package 下面创建一个名为“MyAction-conversion.properties”的文件。文件内容为：

```
   typeVal = MyTypeConverter
```

等式左边的“typeVal”是请求名或是需要转换的表单值，等式右边是转换类的全限定名。

如果你在多个 action 中都使用转换功能的话，那么就可以使用一个全局的配置文件“xwork-conversion.properties”。这个文件放在应用程序的 classpath 根目录下。其内容为：

```
MyType = MyTypeConverter
```

等式左右是要转换的类型和转换器所对应的类的全限定名。注意这里用的是要转换的类型的类名，而不是请求名或表单值。所以 action 的 setter 就是“setTypeValue(MyType type)”。

利用列表数据项支持

Struts2 可以通过 list 来把列表数据在 HTML 用户界面和 action 之间进行便捷的传递。让我们看一个例子。这是一个 Person 类，每一个属性都有对应的 getter 和 setter（这里省略掉了）：：

```
public class Person {
 int id;
String name;
 int age;
 float height;
}
```

我们的 action 需要有一个 person 列表：

```
public class MyAction {

 public List getPeopleList() { … }
 public void setPeopleList( List peopleList ) { … }

 …
}
```

在把 Person 类作为 MyAction 中的一个元素使用之前，我们需要添加一些配置信息。在 MyAction 的同一个 package 下面创建一个名为“MyAction-conversion.properties”的文件。该文件名和验证文件的命名方式一样，在 action 名字后面跟着“*-conversion.properties”后缀。文件内容为：

```
Element_peopleList=Person
```

“Element_”前缀是一个常量，它后面的“peopleList”是 action 类中的 list 属性名。等式右边是要放到 list 里面的类的全限定名。

最后，我们还需要把这个 list 渲染给用户：

```
   <ww:iterator value="peopleList" status="stat">
        <s:property value="peopleList[#stat.index].id" />
        <s:property value="peopleList[#stat.index].name"
/>
        <s:property value="peopleList[#stat.index].age"
/>
        <s:property
value="peopleList[#stat.index].height" />
    </ww:iterator>
```

因为 list 是有索引的，所以我们使用了迭代器状态对象的索引属性来引用被显示的对象。对当前的例子而言，这不是最有效率的办法；因为<s:property … />标签的 value 还可以简化成“id”，“name”，“age”和“height”。这里只是想让你明白，要得到可编辑的表单需要哪些东西。

对于一个使用了相同元素的列表型可编辑的表单，其 JSP 为：

```
   <s:form action="update" method="post" >

   <s:iterator value="peopleList" status="stat">

          <s:hidden
                name="peopleList[%{#stat.index}].id"
                value="%{peopleList[#stat.index].id}"/>
          <s:textfield label="Name"
                name="peopleList[%{#stat.index}].name"
                value="%{peopleList[#stat.index].name}"
             />
          <s:textfield label="Age"
                name="peopleList[%{#stat.index}].age"
                value="%{peopleList[#stat.index].age}" />
          <s:textfield label="Height"
                name="peopleList[%{#stat.index}].height"
                value="%{peopleList[#stat.index].height}"
             /> <br/>

   </s:iterator>
```

```
<s:submit value="Update"/>

</s:form>
```

注意这段代码中的“name”和“value”属性和上面的“value”属性很相似。区别在于，在“name”属性中，我们需要用正确的 token 把“#stat.index”包起来，从而得到实际的索引值，再用它来获取真正的值。而在“value”中，整个表达式都被包了起来。Struts2 用这段代码生成一个 ArrayList，里面都是计算后的 People 对象，然后用 setPeopleList()方法把这个 ArrayList 赋值给 action。

为了允许 Struts2 可以在这个 list 中创建新的对象（可能你是在用户界面中动态创建对象的），把下面这行代码加入“MyAction-conversion.properties”文件中：

```
CreateIfNull_peopleList = true
```

在 Action 中暴露领域模型

介绍过数据转换和容器后，有一扇门已经向你打开了。我们用不着非要在 HTML 表单字段中使用 String 值，使用正确的类型就行。我们还看到通过正确的 setter 方法，请求属性和表单字段值都可以直接赋给 action。

我们还可以更进一步。Struts2 中有一种名为“模型驱动 action”的概念，可以减少在调用业务服务时，用来把用户界面的字段映射为领域对象或者值对象的代码。

模型驱动的 action 需要实现 ModelDriven 接口。这个接口只有一个方法，用来把 action 被指派的对象作为模型返回。

```
public interface ModelDriven  {

    java.lang.Object getModel();
}
```

被用来处理 action 的拦截器栈也需要包含“模型驱动”的拦截器。包括这样一个拦截器的栈有以下几种：“modelDriven”，“defaultStack”，“paramsPrepareParamsStack”和“completeStack”。这个拦截器会从 action 中得到模型对象，把它放

到值栈中相应的 action 上面的位置，这样请求或者表单值就会直接赋值给模型而不是 action。同样，反过来讲，在 JSP 使用和显示的值也就会从模型中获取。

这不是那种“要么全是，要么全非”的场景，如果有的请求或者表单值在模型中找不到对应的 setter，那么它们就会被传递给 action。同时，如果 JSP 在模型中找不到所需的值，它也会在值栈中向下查找，然后从 action 得到想要的结果。

通过综合运用这几种技术，你可以在模型或者 action 中进行赋值，或者读取相应的值，而不必显式指定要操作的目标。

尽可能使用声明式验证

在 Struts2 应用中提供验证的方式有两种——编程式和声明式。

要提供编程式验证的话，action 就要实现 Validateable 接口。该接口只有一个方法，在方法中需要执行验证操作：

```
void validate();
```

为了将验证中出现的问题反馈给用户，action 还需要实现 `ValidationAware` 接口。这个接口更为复杂一些，它里面的方法可以用来添加验证错误，判断当前是否有验证错误，等等。

如果可能的话，你的 action 可以继承 ActionSupport 类，在该类中提供了以上这些接口的缺省实现。只有当验证操作极其复杂的时候，我们才该使用编程式验证。更好的解决方案是采用声明式验证。

每一个需要声明式验证的 action 都需要在类中进行注解（后面对此会进行讨论），或者要有一个相应的 XML 文件。对于 MyAction 这个类，其 XML 文件就是“MyAction-validation.xml”，并会和它放在同一个 package 下面。处理这个 action 的拦截器栈需要包含有“validation”（用来进行验证）和“workflow”（当验证失败时用来进行重定向，以返回到“input”结果）拦截器。这样的拦截器栈有以下几个：“validationWorkflowStack”，

“paramsPrepareParamsStack”，“defaultStack”和“completeStack”。

下面是验证文件的一个例子：

```
<!DOCTYPE validators
PUBLIC "-//OpenSymphony Group//XWork Validator
1.0.2//EN"
"http://www.opensymphony.com/xwork/xwork-validator-
1.0.2.dtd">
<validators>
<field name="count">
       <field-validator type="int" short-
   circuit="true">
              <param name="min">1</param>
              <param name="max">100</param>
              <message key="invalid.count">
                     Value must be between ${min} and
                 ${max}
              </message>
       </field-validator>
</field>
<field name="name">
       <field-validator type="requiredstring">
              <message>You must enter a name.</message>
       </field-validator>
</field>
<validator type="expression" short-circuit="true">
       <param
   name="expression">email.equals(email2)</param>
       <message>Email not the same as email2</message>
</validator>
</validators>
```

在这个例子中，有些东西是应该注意的：

- 每一个字段都可以含有一个或多个“field-validator”节点
- 每一个字段的 validator 是按照定义顺序执行的
- 每一个字段的 validator 都有一个“short-circuit”属性；如果其值为 true 而验证失败，那么剩下的所有验证都会被跳过，该字段就会直接返回一个失败的结果
- Message 节点可以包含一个“key”属性，用来从 message bundle 中查找需要显示给用户的消息；如果找不到对应的 message bundle key 的话，那么就返回节点的 value 属性的值

- Validator 的配置信息（例如 min 和 max）和值栈中的值一样，都可以通过把值放在“${”和“}”之间，来显示到验证消息中。
- Validators 的作用域可以是“field”或者“expression”；具有“expression”作用域的 validator 可以同时作用于多个字段

下面是关于validator类型和相应描述的一个完整列表。在 http://struts.apache.org/2.x/docs/validation.html可以得到包括配置相关的更多信息。

名称	描述
required	确保该属性不是 null
requiredstring	确保一个 String 类型的属性不是 null，并且非空
stringlength	检查 String 的长度范围是否与所期望的一致
int	检查 int 类型的数字是否超出所期望的大小范围
double	检查 double 类型的数字是否超出所期望的大小范围
date	检查 date 类型的属性是否超出所期望的范围
expression	使用值栈来估算一个 ONGL 表达式（必须要返回 boolean 值）
fieldexpression	使用 OGNL 表达式来验证字段
email	保证该属性是一个有效的 email 地址
url	保证该属性是一个有效的 URL
conversion	检查该属性是否有转换错误
regex	检查该属性的值是否与某个正则表达式相匹配。
visitor	Visitor validator 可以把对字段的验证动作推迟到这个字段所属的类的特有的另一个验证文件中执行。 比如说，你在使用模型驱动的 action，在模型中有一个对应于 Person 类的“person”属性。如果该模型正在被多个 Action 使用，那么你大概就会想要把验证信息抽取出来进行重用。Visitor 验证提供了这样的功能

另外，你还可以创建自己的 validator。你自定义的 validator 需要实现 Validator 接口（用于表达式验证）或 FieldValidator 接口（用于字段验证）.

新的 validator 需要在“validators.xml”中注册，该文件存在于 classpath 的根目录下。通常这个文件可以从发行的 JAR 包中访问到，但是如果人为提供了这个文件的话，那么发行包中的文件就会被忽略。所以在添加新的 validator 之前，你需要从 Struts2 JAR 包中把“validators.xml”这个文件拷贝出来，放到项目 classpath 的根目录下，然后再把新的 validator 添加进去。这样当前已有的 validator 才会被包含到应用中来。和其他配置一样，validator 的配置信息也很简单，只需要一个统一标识符和 validator 的类名。

```
<validators>
<validator name="postcode"
       class="com.validators.PostCodeValidator"/>
…
</validators>
```

把CRUD操作放到同一个Action中

把一个模型驱动的 Action 和“preparable”拦截器/接口，Action 配置中的通配符，还有验证与工作流结合以后，你就可以把 CRUD 操作放到一个 Action 里面去。这种方式与 `Restful2ActionMapper` 的功能很相似。

我们使用的 URL 模式为“/{model}/{method}.action”。例如，调用 UserAction 类中的“add”方法，那么 URL 就是“/User/add.action”。我们同样还需要确保有对应的结果映射——“success”（缺省情况），“input”（验证出现问题的情况）和“home”或是一个默认页面。每一个页面都是模型所特有的。“success”会按照 redirect-after-post 模式，重定向到 Action，

下面是相应的“struts.xml”文件：

```
<action name="*/*" method="{2}"
       class="com.infoq.actions.{1}Action">
 <result type="redirect">/{1}/view.action</result>
<result name="view">/{1}/view.jsp</result>
 <result name="input">/{1}/edit.jsp</result>
 <result name="home">/{1}/home.jsp</result>
</action>
```

这个 Action 需要继承 ActionSupport 类（提供了验证和错误消息处理的实现），并实现 ModelDriven 和 Preparable 接口。与这两个接

口协同工作的拦截器栈可以帮助我们简化对接口的实现，下面来看一下具体细节。

ModelDriven 接口中有一个方法，getModel()，它与“model-driven”拦截器相结合，把 Action 中的模型放到值栈中，位于 Action 之上。当请求参数被传入时，它们将会被赋给模型而不是 Action。这就是我们想要看到的——对模型赋值而非 Action——然后我们只需要更新 Action。但是如果模型里面已经有了数值，而我们并不想重写它们呢？

这时就需要“paramsPrepareParamsStack”拦截器栈发挥作用了。下面的步骤就是我们想要做，而栈中的拦截器帮我们做了的工作：

1. 设置 Action 中的 id—— **“params” 拦截器**
2. Action 执行某些逻辑，或是创建一个新模型，或是从 Service 上，从数据库中获取一个已有的模型—— **“prepare” 拦截器调用 Preparable 接口中的 `prepare()` 方法**
3. 当模型已经存在后，把请求参数赋值给模型——**先是“model-driven”拦截器，然后又是“params”拦截器。**
4. 查看模型是否存在验证错误，在必要的情况下重定向回 input 页面—— **“validation”和“workflow”拦截器**
5. 执行被调用的方法中的逻辑——**普通的 action 处理**

通过遵守这些约定，你的 Web 应用中所有的模型或者实体对象都可以通过上面的“struts.xml”文件来管理。只是 Action 的实现需要根据情况不同而发生变化。在我们的例子中，UserAction 的代码为：

```
public class UserAction
       extends ActionSupport
       implements ModelDriven, Preparable{

 private User user;
 private int id;
private UserService service;    // user 业务服务

 …

public void setId(int id) {
       this.id = id;
```

```
}

        /** 如果 user 不存在的话，就创建一个新的，否则就根据特定
    的 id 来载入 user */

public void prepare() throws Exception {
        if( id==0 ) {
                user = new User();
        } else {
                user = service.findUserById(id);
        }
}

public Object getModel() {
        return user;
}

        /** 创建或更新 user，然后查看该 user */
 public String update() {
        if( id==0 ) {
                service.create(user);
        } else {
                service.update(user);
        }
        return “redirect”;
 }

        /** 删除该 user，然后跳转到默认的 home 页面 */
 public String delete() {
      service.deleteById(id);
        return “home”;
 }

        /** 显示页面，让用户可以看到已存在的数据*/
 public String view() {
        return “view”;
 }

        /**显示页面，让用户可以看到已存在的数据并对其进行修改*/
 public String edit() {
        return “input”;
 }
}
```

当用户想创建一个新的 user 或是修改一个已存在的 user 时，edit()方法就会被调用。这个方法很简单，返回“input”结果，然后就会给用户一个 HTML 表单页面。该表单的 Action 指向一个映射到 update()方法上的 URL。update()方法实际上还可以被拆分成两个独立的方法，但是这会让 HTML 表单变得复杂，而且因为通过唯一的

key 字段可以很容易的判断一个对象是否存在，所以这样做的意义也不大。

最后，view()方法是一个简单的 pass-through 方法，它将页面跳转到一个显示 user 的页面上去。delete()方法会根据特定的 id 来删除 user，然后返回到用户默认的 home 页面。

所有这些方法基本上都没有什么逻辑实现，所以很容易就会被误解为什么什么都没做。实际上，这里还是有功能实现的，但是它是一个横切的关注点，并且被重构到了 prepare()方法中。对于 edit()，update()和 view()这几个方法而言，如果有模型存在的话，那么它就要获取到该模型；如果没有的话，那么就要创建一个模型。

这种 Action 实际上还是很简单的，而且很容易参数化，以用于不同的模型类和 service。有了这样的微架构，那么在开发 CRUD 的应用时，最复杂的事情就变成创建页面模板了。

在可能的地方使用注释

Struts2 是在JDK5 上面开发的，所以可以使用注解来代替配置。注解可以用多种方式来进行自我阐述。在Strut2 网站上有关于注解的更多信息，也包括代码示例。网址为 http://struts.apache.org/2.x/docs/annotations.html。

零配置

“零配置”通过使用注解来避免对“Action”进行 XML 配置，如果你一直都是继承既有的 package 的话，那么“struts.xml”文件也可以省略掉。它包括以下四种类级别的注解，分别为：

注解	描述
Namespace	所期望的命名空间（在“struts.xml”文件中也有定义）的字符串值
ParentPackage	所期望的父 package 的字符串值
Results	“Result”注解列表
Result	提供了 Action 结果的映射，它有四个属性： ▪ name ——action 方法的结果名 ▪ type—— 结果类型 ▪ value——任意的结果值。可以是 rediect

	结果类型对应的 action 名，也可以是 dispatcher 结果类型对应的 JSP ▪ parameters ——字符串参数组成的数组

在使用这些注解的时候，还需要进行额外的一些配置。在 web.xml 的 filter 配置中，需要指定哪些 package 是使用了注解的。配置如下所示，其中参数名必须为“actionPackages”，参数的值就是 package 的名称列表。

```
<filter>
<filter-name>struts</filter-name>
<filter-class>
       org.apache.struts2.dispatcher.FilterDispatcher
</filter-class>
<init-param>
       <param-name>actionPackages</param-name>
       <param-value>user.actions,other.actions</param-
   value>
</init-param>
</filter>
```

被配置过的每一个 package 和它的子 package 都会被扫描到，看其中哪些类实现了 Action 或者类名以“Action”结尾，然后注解就会被加入到运行时配置中去。如果没有使用 namespace 注解的话，那么命名空间就会由 package 名来生成。把“actionPackages”配置值中使用的 package 名称截掉，就得到了命名空间。也就是说，如果某个被配置好的 action 的名字是“actions.admin.user.AddAction”，而“actionPackages”的值为“actions”，那么这个 action 的命名空间就是“/admin/user”。

使用这些注解并不能完全避免 XML 的使用——但这是一个良好的开始。例如默认拦截器，package 层次结构之类的 package 信息仍然需要进行配置。

生命周期回调

关于方法级的生命周期回调的注解一共有三种，每一种都是在 Action 处理过程中的特定时刻被调用的。生命周期回调与拦截器及 Action 代理不同，它们特定于被调用的 Action 类存在的，并不是那种可以在多个 Action 中使用的单个的类。

注解	描述

Before	被注解的方法会在 Action 的逻辑执行前被调用。
BeforeResult	被注解的方法的调用时机是 Action 的逻辑执行之后，但执行结果还未被调用之前。
After	被注解的方法的调用时机是 Action 的结果被调用之后，但尚未被返回给用户之前

验证

任意一个 XML 配置的 validator 都有一个相应的注解。每一个注解的属性都和 XML 的配置很相似。还有些注解可以用来把一个类定义为使用基于注解的验证，或是配置自定义的 validator，还可以把一个属性或类的验证进行分组。

注解	相应的 XML	描述
RequiredFieldValidator	required	确保该属性不是 null
RequiredStringValidator	requiredstring	确保一个 String 类型的属性不是 null，并且非空
StringLengthFieldValidator	stringlength	检查 String 的长度范围是否与所期望的一致
IntRangeFieldValidator	int	检查 int 类型的数字是否超出所期望的大小范围
DoubleRangeFieldValidator	double	检查 double 类型的数字是否超出所期望的大小范围
DateRangeFieldValidator	date	检查 date 类型的属性是否超出所期望的范围
ExpressionValidator	expression	使用值栈来估算一个 ONGL 表达式（必须要返回 boolean 值）
FieldExpressionValidator	fieldexpression	使用 OGNL 表达式来验证字段

EmailValidator	email	保证该属性是一个有效的 email 地址
UrlValidator	url	保证该属性是一个有效的 URL
ConversionErrorFieldValidator	conversion	检查该属性是否有转换错误
RegexFieldValidator	regex	检查该属性的值是否与某个正则表达式相匹配。
VisitorFieldValidator	visitor	把对字段的验证动作推迟到这个字段所属的类的特有的另一个验证文件中执行。
StringRegexValidator	n/a	检查字符串是否与正则表达式匹配
CustomValidator	n/a	表示使用了一个自定义的 validator
ValidationParameter	n/a	作为 CustomValidator 注解的一个参数
Validation	n/a	表示该类使用了基于注解的验证——这个注解可以与接口或类一起使用
Validations	n/a	用来对一个属性或类组合使用多种验证

转换和容器

与“验证”类注解类似，“转换和容器”类注解为每一个使用了“*- conversion.properties”配置文件的对象都提供了相应的注解。

注解	描述
KeyProperty	用于指定作为 key 的属性
Key	用作 map 的 key 的类型
Element	用于容器 list 或 map 中的值或元素的类型
CreateIfNull	如果 list 或 map 中不存在元素的话，是否创建一个新的元素
Conversion	表示该类使用了基于注解的转换——这个注解可以与接口或类一起使用
TypeConversion	定义要使用的转换类。如果用在容器上，那么 PROPERTY， MAP， KEY， KEY_PROPERTY 或者 ELEMENT 组合的规则会被用来指定哪些元素会被进行转换

视图技术选项

我们在讨论框架的核心元素时提到过，标签库既可以在 JSP 中访问，又可以在 Velocity 和 Freemarker 模板中使用，还能被扩展用于其他视图技术中。由于 Velocity 和 Freemarker 是这个框架中的一等公民，框架对它们提供了完整的标签库支持，所以开发人员可以选择最适用于当前项目的视图技术来使用。

让我们通过文本框标签在每一种视图技术中的渲染方式，来看看这几种技术的相似点和区别。文本框标签是很常见的表单标签——用来生成 HTML 表单元素，让用户可以输入文本。这个元素有很多属性，不过这里我们只用其中两个——name 属性和 label 属性，前者表示元素名，用来在 HTML 和 action 属性中引用这个元素；后者用来在输入框之前显示一段文字标记。

在 JSP 中，标签库必须在使用之前被声明。声明只需要在每一个 JSP 页面的起始处出现一次即可，然后就可以用 prefix（在这里是“s”）来引用标签库了。

```
<%@taglib prefix="s" uri="/struts-tags" %>
```

```
<s:textfield label="Name" name="person.name" />
```

在 Freemarker 和 Velocity 里没有 JSP 那么复杂的结构，也没有标签库的概念。任何被添加到模板上下文中的对象都可以在页面模板中访问。JSP 标签库也是这样的对象。HTTP Servlet 请求与响应，值栈和刚刚执行完的 Action 也都被放在模板上下文中使用。

与 JSP 相似的是，Freemarker 模板中也用了“s”前缀来引用 JSP 标签库。模板中的其他属性和 HTML 很相似——用“<”开头，用“/>”结尾，属性名则是一样的。其中一个区别是，在第一个尖括号后面紧跟着一个“@”符号。

```
<@s.textfield label="First Name" name="person.name"/>
```

在http://freemarker.sourceforge.net/上面有关于Freemarker的更多信息。

Velocity 和 HTML 语法的相似性就比 Freemarker 少得多了。它在每一个 JSP 标签名前面都有一个“#s”前缀，名-值对都在圆括号内部被双引号引了起来。

```
#stextfield ("label=Name" "name=person.name")
```

在http://velocity.apache.org/上有关于Velocity的更多信息。

和 JSP，Velocity 和 Freemarker 一样，XML 也可以作为 Action 的结果直接呈现给用户，或者是使用 XSLT 结果类型，通过 XSL 转换后显示给用户。现在 Struts2 中正在进行一件很有意思的工作——接受 JSON 格式的表单或者请求，并且用 JSON 格式输出结果——这使得 Struts2 具有更高的灵活性，可以容纳所有的视图技术，甚至还可以在同一个项目同一个 Action 中混合使用多种视图技术。

了解框架提供的标签库及其特性

Struts2 中提供了各种可以在 JSP，Velocity 和 Freemarker 视图中使用的标签库，用来提供从 UI 到 Action 的集成和对这种信息的操作。

在默认情况下，表单标签接受 OGNL 表达式作为属性值，还有很多标签的属性也可以这样。如果你要使用的属性没有这个默认值——例如“label”属性——那么基本上你都可以用“%{”和“}”把属性值包起来，整体作为一个表达式来执行。

能够执行 OGNL 表达式是一个很强大的特性，尤其是标签库还可以访问值栈，这样一来你就可以做到以下几点：

- 访问 application、session、request 作用域的 HTTP 命名的对象，以及 request 的属性和参数
- 访问刚刚执行过的 Action
- 访问刚刚执行过的 Action 中的模型
- 访问临时对象——例如表示迭代中当前遍历的对象，或是被放到值栈中的对象
- 存储临时变量
- 获取最后一个执行的 Action 中的验证问题
- 获取提供的 key 所对应的国际化文本
- 调用静态对象中的方法，获取静态属性的值

在http://struts.apache.org/2.x/docs/tag-reference.html上有标签库的最新信息。为了帮助你有一些初步的认识，在下面列出了四类标签，并辅以说明。每一个标签的属性和更详尽的信息都可以从上面的链接中得到。

下面的标签中，带有星号（*）标记的标签要么是基于 Ajax 的标签，要么是在 Ajax 模式与非 Ajax 模式下都能工作的标签。

Control 标签

名称	描述
if, elseif & else	这三个标签提供了页面的流程控制逻辑。“if”和“elseif”都有一个“test”属性，它对应的值必须是一个 OGNL 表达式，执行后返回一个 boolean 值
append	用来将多个迭代器附加到一个迭代器上。每一个迭代器都通过内置的“param”标签来标识。每一个迭代器中的所有元素都会按照被指定的顺序添加到最终的迭代器中

generator	在给定的字符串列表的基础上生成一个迭代器。我们既可以指定要解析的列表，又可以指定自定义的转换器
iterator	对容器进行遍历。我们可以访问迭代状态对象（提供当前循环的位置信息），也可以指定当前循环的对象 id。
merge	用来将多个迭代器合并到一个迭代器上。每一个迭代器都通过内置的“param”标签来标识。最后的迭代器会依次包括所有迭代器的第一个元素，所有迭代器的第二个元素，等等。
sort	使用指定的 comparator 来对容器排序。排序后的容器可以在这个标签中遍历
subset	选择当前容器的子集。可以提供一个选择范围，也可以提供自定义的选择对象。选择结果可以在这个标签中遍历

Data 标签

名称	描述
a*	渲染一个 HTML 链接
action	在页面中调用 Action，经过配置后，还可以渲染结果。这个标签可以作为动态的 include 使用，在不渲染结果的情况下调用通用的 Action 来获取数据，或者是在页面模板之前调用 Action。
bean	根据特定的类来实例化一个 Bean，然后把它放到值栈中。使用它内置的“param”标签可以给新创建的对象设置属性值
date	格式化一个 Date 对象
debug	生成值栈信息以供调试使用。
i18n	把额外的 Resource Bundles 放到值栈中， 用来将文本国际化。
include	进行方法调用，并把结果动态的引入到当前页面中。用“param”标签来指定请求参数。
param	是一个用来指定名/值的的通用标签，其中的值可以是静态文本，或者是从值栈中获取的表达式。这个标签不能单独使用，只能作为其他标签的子标签。

push	把一个值放到值栈中。
set	把一个值栈中的值设置为特定 HTTP 作用域的属性 (application, session, request, page 或 action).
text	根据特定的 key，从 Resource Bundle 中取一个文本值。可以用“param”标签来指定结果文本中的变量值。
url	生成一个有效的 URL（包括 Servlet 上下文或 Portlet 信息），并把它赋值给值栈中的一个 id。
property	从值栈中获取一个属性的值。它（属性名）可以是一个复杂路径。如果没有对其赋值的话，那么就会被指定一个默认的值。

Form 标签

名称	描述
checkbox	生成一个简单的 HTML checkbox 元素
checkboxlist	把输入数据作为一个对象列表，由此生成一组 HTML checkbox 元素。
combobox	把 HTML input 和 HTML select 组合生成 combo-box 的功能。用户既可以从下拉列表中选择，又可以输入一个新值。
datetimepicker	生成 HTML 下拉框，让用户可以选择日期/时间。
doubleselect	生成两个关联的 HTML select 元素。第一个列表的选择可以影响第二个列表中显示的内容
head	生成 HTML header 信息，尤其是可以用来引入主题中使用的 CSS 和 JavaScript。
file	生成 HTML file input 元素。
form*	生成 HTML 表单元素。
hidden	生成 HTML hidden 元素。
label	生成 HTML label，在 UI 中可以保持一致的格式。
optiontransferselect	生成两个关联的 HTML select 元素，并在它们中间生成 HTML 按钮。这些按钮可以用来在列表中间转移数据。
optgroup	在一个 HTML select 元素中生成一个可选项组。
password	生成 HTML password 元素

reset	生成一个 reset 按钮，可以是 HTML button 类型，也可以是 HTML input 类型。
select	生成 HTML select 元素
submit*	生成 HTML submit 按钮或链接——可以是一个 HTML input，一个 image 或一个按钮。 在使用默认的ActionMapper的时候，它有四种特殊的“name”属性前缀，submit标签可以用它们来修改一般情况下表单所用的URL，action或者method。它们是：“method:”，“action:”，“redirect:”和“redirect-action:”。要调用的URL，方法或者action要放在冒号的后面。更多信息请参见 http://struts.apache.org/2.x/docs/actionmapper.html.
textarea	生成 HTML text area 元素
textfield	生成 HTML text field 元素
token	向用户提醒来防止用户进行二次提交。它是和“token” 或者“token-session”拦截器配合工作的。
updownselect	生成 HTML select 元素和用来在列表中上下移动数据的 HTML 按钮。当表单提交时，列表中的元素顺序会和排列顺序一样。

Non-Form UI 标签

名称	描述
checkbox	生成一个简单的 HTML checkbox 元素
checkboxlist	把输入数据作为一个对象列表，由此生成一组 HTML checkbox 元素。
combobox	把 HTML input 和 HTML select 组合生成 combo-box 的功能。用户既可以从下拉列表中选择，又可以输入一个新值。
datetimepicker	生成 HTML 下拉框，让用户可以选择日期/时间。
doubleselect	生成两个关联的 HTML select 元素。第一个列表的选择可以影响第二个列表中显示的内容
head	生成 HTML header 信息，尤其是可以用来引入

	主题中使用的 CSS 和 JavaScript。
file	生成 HTML file input 元素。
form*	生成 HTML 表单元素。
hidden	生成 HTML hidden 元素。
label	生成 HTML label，在 UI 中可以保持一致的格式。
optiontransferselect	生成两个关联的 HTML select 元素，并在它们中间生成 HTML 按钮。这些按钮可以用来在列表中间转移数据。
optgroup	在一个 HTML select 元素中生成一个可选项组。
password	生成 HTML password 元素
reset	生成一个 reset 按钮，可以是 HTML button 类型，也可以是 HTML input 类型。
select	生成 HTML select 元素
submit*	生成 HTML submit 按钮或链接——可以是一个 HTML input，一个 image 或一个按钮。 在使用默认的ActionMapper的时候，它有四种特殊的“name”属性前缀，submit标签可以用它们来修改一般情况下表单所用的URL，action或者method。它们是：“method:”，“action:”，“redirect:”和“redirect-action:”。要调用的URL，方法或者action要放在冒号的后面。更多信息请参见 http://struts.apache.org/2.x/docs/actionmapper.html.
textarea	生成 HTML text area 元素
textfield	生成 HTML text field 元素
token	向用户提醒来防止用户进行二次提交。它是和“token” 或者“token-session”拦截器配合工作的。
updownselect	生成 HTML select 元素和用来在列表中上下移动数据的 HTML 按钮。当表单提交时，列表中的元素顺序会和排列顺序一样。

自定义 UI 主题

作为标签库架构的一部分，每一个标签都有对应的类来管理模型和逻辑，以及可以控制显示在 HTML 中元素的一个或多个 Freemarker 模板。模板可以以实用为主，只提供必需的 HTML 元素；或者在表单和非表单的 UI 标签中，用它提供丰富的排版功能，包括将消息和错误信息提供给表单元素。

对于最基本的用户界面来说，默认的布局就足够了。但是一般情况下，用户需求都要复杂的多。在新的布局下你有两个选择——把要修改的 HTML 应用到程序的所有页面中，或者是修改默认的主题模板，来为要做出的变化提供新的主题。就像把视觉格式信息抽象到 CSS 文件中一样，创建新的主题可以给后续的维护带来极大的便利（尤其是当应用程序中的页面数量增加的时候）。

主题的创建和修改都很简单。在应用程序的根目录下创建一个名为“template”的文件夹，然后就可以进行操作了。

如果要创建一个新主题的话，那么就在“template”目录下创建一个与新主题同名的目录——我们在这里给它命名为“modified”。然后你可以一步步创建模板，或者从 Struts2 发行版中拷贝一个模板出来，在它的基础上进行修改。如果你想在标签中使用“modified”主题而不是默认的“xhtml”主题，那么就需要在每一个标签中修改 theme 属性，指向“modified”主题。

```
   <s:textfield label="Name" name="person.name"
theme=”modified” />
```

如果你只是想修改某一个模板，那么就只需要对它进行重写。在上面创建的“template”目录下，创建一个名为“xhtml”（这是 Struts2 的默认主题）的目录。然后创建新的模板或是修改 Struts2 中的已有模板，但要注意的是，这个模板需要和 Struts2 中的模板名保持一致。只有模板名一致的情况下，它们才是同一个主题。而 web 应用中的“template”目录会在 Struts2 的 JAR 包之前被检索——然后被修改的模板就会被直接应用，无需我们作做他配置。

如果你想要完全替换掉一个主题，就去修改“struts.properties”文件。在这个例子中，我们需要把“struts.ui.theme”属性指向新的

“modified”主题。在这个文件中，还可以修改主题模板所存放的目录。

```
struts.ui.theme=modified
struts.ui.templateDir=template
```

为通用的输出创建全局结果

结果的作用域可以是针对特定 Action 的，也可以是全局的。我们可以对通用的 Action 结果——例如“error”和“logon”——进行重构，这样每一个 Action 就可以只处理和自身特有逻辑相关的结果了。

在全局结果中使用的<result … />标签和普通的 Action 结果标签的形式基本相同——有一个唯一的“name”属性，还有一个“type”属性，描述了不同的渲染选项。它们的不同点在于，全局的结果标签是放在<package … />根标签下的<global-results … />标签里面的。

```
<package … >

 <global-results>
      <result name=”logon”>/logon.jsp</result>
      <result name=”error”>/error.jsp</result>
      …
 <global-results>

 …

</package>
```

全局结果在“struts.xml”定义了以后，它就可以被应用程序中的所有 Action 使用了。

声明式异常处理

在开发 web 应用时，需要处理不同种类的异常。有些异常是特定于服务或者正在调用的业务对象的——这些无法进行声明式处理，只能通过编程来处理它们。

但是还有另外一些异常：

- 无法处理，需要把用户重定向到一个错误页面，直到问题解决为止。这些常常是系统级别或者资源级别的问题，和 Web 应用的逻辑无关。因网络问题而导致的数据库连接失败就是这样一个例子。
- 与逻辑无关，但是需要对用户重定向到执行额外操作的页面。比如说，如果用户在未登录的情况下来访问一个 web 页面，就可能因为安全问题而抛出异常。当用户登录以后，他们就可以继续操作了。
- 与逻辑相关，可以通过修改用户的工作流程解决。这种问题常常是与资源相关的，包括唯一约束冲突的异常，对数据并发修改或是资源锁问题等等。

这些异常都可以进行声明式管理，无需修改 Action。

当一个异常可能会被应用中的所有 Action 抛出时，它应该被声明为全局异常。全局异常在“struts.xml”文件中进行声明，它位于<package … >标签下的<global-exception-mappings … />标签里面。

```
<global-exception-mappings>
<exception-mapping result="sqlException"
       exception="java.sql.JDBCConnectionException"/>
<exception-mapping result="unknownException"
       exception="java.lang.Exception"/>
</global-exception-mappings>
```

在<global-exception-mappings … />中，<exception-mapping … />标签的数量是没有限制的。标签中的每一个映射都包含两个属性——“exception”属性定义了异常类的全限定名，“result”标签定义了重定向的结果。

每一个异常映射都会按照被配置的顺序来进行检索。当检索到一个匹配的异常（或它的子类）时，处理过程就会终止，页面请求就会被转发给先前映射的结果。否则就会按照配置顺序向下继续检索能够匹配的异常。

如果一个异常的作用域只是单个的 action，那么就在<action … />标签内进行同样的<exception-mapping … />标签配置。

```
<action name="my" class="com.fdar.infoq.MyAction" >
<result>view.jsp</result>
```

```
    <interceptor-ref name="basiActioncStack"/>
    <exception-mapping result="exists"
                  exception="ConstraintViolationException"
/>
    </action>
```

这里的属性和全局异常的属性相同。如果在 action 级别上没有找到匹配的异常映射，那么就会从全局异常的定义中检索相应的异常。

同时，你还应该保证拦截器栈中要有“exception”拦截器，并对需要进行声明式异常处理的 Action 进行配置。在默认情况下，Struts2 提供的所有拦截器栈都包含有“exception”拦截器。

在对抛出异常时的结果进行修改的同时，“exception”拦截器也在值栈中添加了两个元素，用以提供异常信息。

名称	描述
exception	所抛出的异常对象
exceptionStack	stack trace 的字符串值

这些值可以用来向用户显示异常的堆栈信息，或是显示一些友好的用户提示信息，或者是重新组织页面布局，显示额外的数据项，并可以再次提交表单。

国际化

Struts2 通过 Resource Bundle，拦截器和标签库提供了可扩展的国际化支持。它的核心功能是通过 Resource Bundle 提供的，我们的讲述也将从这里开始。

Resource Bundles

Struts2 使用 Resource Bundle 来向用户提供多语言和多区域的选项。我们不需要在一个庞大的文件中提供整个应用中的所有文本（虽然这个选项是可以通过只使用一个属性文件来支持的）。相反，应用中的属性文件可以被分拆成易于管理的大小。

属性文件是根据 Action 类（包括基类和接口）和 package 作用域，或是任意的文件名来命名的。如果需要一个 key 所对应的值，

那么系统就会按照下面的顺序来检索属性文件，直到找到该 key 为止：

1. Action 类的属性文件——例如 MyAction.properties
2. Action 层次结构中的每一个基类的属性文件，直到 Object.properties 文件为止。
3. 每一个接口和子接口的属性文件。
4. 如果一个 Action 是模型驱动的，那么就有对应于模型对象（以及模型对象的每一个基类，接口和子接口）的属性文件
5. 从每一个 Action 的 package 到根 package，都有一个命名为“package.properties”的属性文件。
6. 在“struts.properties”文件中，“struts.custom.i18n.resources”配置项中定义的属性文件。

这样提供了高度灵活性。

拦截器与 Locale 选择

在默认情况下，Struts2 会在 HttpServletRequest 对象的 Session 中设置用户 Locale。它是直接从 Web 浏览器中得来的，基于 Accept-Language HTTP header。

当 Web 应用需要与 Web 浏览器 Locale 无关的多语言内容时，我们可以使用“i18n”拦截器。这个拦截器对名为“request_locale”的请求参数进行检查，并且把它保存到用户 Session 中。在它被请求参数再次修改之前，这个特定的 Locale 就会一直作为用户 Session 所其他部分的 Locale。

标签库

标签库是最后一处需要说明的地方。 所有的标签都是通过当前起作用的 Locale 来支持国际化的。比如说，“date”标签使用用户 Locale 来决定正确的日期格式；“actionerror”、“fielderror”和“actionmessage”标签都是通过声明式验证配置中所提供的 key 来获取要渲染的文本。还有一些其他的标签很值得关注。

有两种方式可以编程式获取一个页面所需的国际化文本，它们都需要 Action 继承 ActionSupport 这个类，这样才可以使用必需的国际化方法。第一种是使用“text”标签，这个标签通过“name”属性所提供的 key 来检索文本：

```
<s:text name="label.greeting"/>

<s:text name="label.greeting2">
   <s:param >Mr Smith</s:param>
</s:text>
```

额外的信息可以在 Resource Bundle 通过花括号来提供。上面的例子对应的属性文件内容的形式是这样的：

```
label.greeting=Hello there!
label.greeting2=Hello there {0}!
```

第二种方式是通过OGNL方法和“property”标签来获取文本值。这种技术和上面介绍的那中的不同在于开发者对风格的选择。[10]。在方法调用时，OGNL表达式可以被用在任何进行表达式运算的标签中。在第二种方式下，上面的页面就变成了：

```
<s:property value="getText('label.greeting')"/>

<s:property value="getText('label.greeting2')">
   <s:param >Mr Smith</s:param>
</s:text>
```

由于“label”属性在默认情况下不是 OGNL 表达式，所以我们需要使用“%{”和“}”符号来强制 Struts2 把它作为 OGNL 表达式解析。

```
<s:textfield label="%{getText('label.greeting')}"/>
```

在需要大段文字的时候，OGNL 表达式还可以和“include”标签一起使用，用来指定语言目录。在这种情况下，我们要确保每一个 Action 都继承了一个基类，同时在基类中向外暴露一个决定 Locale 目录的方法。在这里我们有一个 getLocaleDirectory()方法。对应的页面为：

```
   <s:include
value="/include/%{localeDirectory}/copyright.html" />
```

“i18n”标签提供了一种从被渲染页面的值栈中获取额外的 Resouce Bundle 的方式，它的“name”属性描述了 Resouce Bundle

[10] 这种说法没错，但是如果使用property标签，开发人员就可以访问值栈，并使用在开发时还不知道的key名来获取国际化文本。

的名称。“i18n”标签中的任何标签都可以访问新的 Resouce Bundle 文本。

```
<s:i18n name="myCustomBundle">
     The value for abc in myCustomBundle
     is <s:text name="abc"/>
</s:i18n>
```

6

其他技术集成

在先前的章节中，我们已经介绍了 Struts2 中用于与其他技术集成的一些技术。下面我们再来复习一下，它们包括：

- *拦截器*——可以更改用户的工作流程，修改结果，把对象注入 Action。
- *结果类型*——允许进行事后处理，额外的基于结果的处理，或是对 Action 返回的信息进行渲染.
- *插件包*——新的拦截器，结果类型，结果和 Action 可以被打包到插件里面，以便在多个项目中重用。
- *插件扩展点*——Struts2 中， 可以用新的核心框架类的实现来进行替换，从而改变框架的行为。

本章的目的不是对每一个集成点的细节进行面面俱到的描述，而是进行大致的概述，让读者了解到在 Struts2 中有哪些类型的集成，如何进行集成，同时还提供了一些基本的配置信息，并说明怎样获取更多的配置信息。本章也并不旨在说明如何使用那些用于集成的类库，但是假定读者已经有了相关的基础知识。

Struts2 中所有集成（Apache和第三方库）的最新信息都可以在 Struts2 wiki上找到，地址为 http://cwiki.apache.org/S2PLUGINS/home.html。新的项目正在不断的添加进来。如果你没有从中找到想要的信息，那么可以检查一下几个月前的更新，也许它已经被添加进来了。如果你想要在自己的 Web应用中添加新的集成，你可以考虑一下把它实现为一个插件，然后和其他人共享。

页面修饰和布局

开发Web应用通常都意味着要有一个标准的页面布局，并被应用到整个应用中，并且不同的模块，页面和向导可能还要使用额外的一些布局。根据个人偏爱不同——自己指定布局或是让URL指定布局——你很可能会选择Struts Tiles[11] 或是SiteMesh[12]。Struts2 中提供了对这两种布局技术的集成。

SiteMesh

SiteMesh可以通过两种方式来安装。一种是把插件[13]放到应用程序下的“/WEB-INF/lib”目录中，一种是在Maven2 “pom.xml”构建文件中添加依赖关系：

```
<dependency>
<groupId>org.apache.struts</groupId>
<artifactId>struts2-sitemesh-plugin</artifactId>
<version>2.0.6</version>
</dependency>
```

在安装完以后，需要配置 Servlet 过滤器。这个过滤器允许 SiteMesh 修饰器访问值栈，并且确保在修饰器完成工作之后（而不是完成之前），会清空 ActionContext。

```
<filter>
     <filter-name>struts-cleanup</filter-name>
     <filter-class>

     org.apache.struts2.dispatcher.ActionContextCleanU
p
     </filter-class>
</filter>
```

如果你使用了 Freemarker 或者 Velocity 来渲染页面，那么就需要添加下面两种过滤器之一：

```
<filter>
     <filter-name>sitemesh</filter-name>
     <filter-class>

     org.apache.struts2.sitemesh.FreeMarkerPageFilter
     </filter-class>
</filter>
<filter>
```

[11] http://tiles.apache.org
[12] http://www.opensymphony.com/sitemesh
[13] http://cwiki.apache.org/S2PLUGINS/sitemesh-plugin.html

```
        <filter-name>sitemesh</filter-name>
        <filter-class>

        org.apache.struts2.sitemesh.VelocityPageFilter
        </filter-class>
    </filter>
```

过滤器映射的顺序也非常重要。“struts-cleanup”和“sitemesh”（如果已经使用）过滤器都要在“struts”（FilterDispatcher）过滤器之前进行配置：

```
    <filter-mapping>
        <filter-name>struts-cleanup</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <filter-mapping>
        <filter-name>sitemesh</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    …
    <filter-mapping>
        <filter-name>struts</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```

完成这些以后，就可以对特定的修饰器文件进行开发了，然后对其进行配置，以匹配特定的 URL 模式，或者是“decorators.xml”中的元数据。

Tiles

Apache Tiles和SiteMesh一样，都可以把插件[14]放到Web应用的“/WEB-INF/lib”目录下，或是在Mave2 “pom.xml”文件中加入依赖：

```
    <dependency>
    <groupId>org.apache.struts</groupId>
    <artifactId>struts2-tiles-plugin</artifactId>
    <version>2.0.6</version>
    </dependency>
```

我们需要配置 Servlet Listener 才能载入 tiles 配置信息：

```
    <listener>
        <listener-class>
```

[14] http://cwiki.apache.org/S2PLUGINS/tiles-plugin.html

```
        org.apache.struts2.tiles.StrutsTilesListener
        </listener-class>
    </listener>
```

Listener 会从“WEB-INF”目录下载入“tiles.xml”配置文件，该文件中定义了应用中的每一个 tile。与 SiteMesh 不一样的是，Tiles 是作为一种新的结果类型实现的。要使用 Tiles 布局的 Action 结果都需要把“type”属性设置为“tiles”（或者是把 Tiles 结果设置为默认值），并指定要使用的 tile 名称。Tile 名称需要在“tiles.xml”文件中定义。

```
<action name="my" class="com.fdar.infoq.MyAction" >
<result type="tiles">myaction.layout</result>
</action>
```

Struts2 中使用了 Tiles 的第二版。这个版本还没有稳定的发布版，还会有更多的变化。所以 Struts2 中对 Tiles 的支持被标记为“试验性”。

业务服务/依赖注入

Spring Framework 插件是 Struts2 推荐使用的依赖注入（DI）或控制反转（IoC）容器，它可以为 Action 提供完全配置好的业务服务实例。

这里还有其他几种选择，每一种都具有不同层次上的稳定性：

- Plexus[15] 插件是刚刚加入代码库的，目前还是被标记为“试验性”。当我们在“struts.xml”配置文件的每一项中使用Plexus id而不是类名时，Plexus可以创建类的实例，并向其中注入所有它所知的依赖。关于该插件的详细信息可以在这里找到：http://cwiki.apache.org/S2PLUGINS/plexus-plugin.html
- PicoContainer[16] 是另外一个Ioc容器，不过虽然WebWork对它提供了支持，但现在还没有对应的Struts2 插件。

[15] http://plexus.codehaus.org
[16] http://picocontainer.codehaus.org

- EJB3，它虽然不是Ioc容器，但仍可以用来为Action提供业务服务。现在Strut2 中现在还没有提供插件来提供对EJB3 的支持，虽然实现起来并没有什么难度。在Struts2 中使用EJB3 有三种方式——实现一个自定义的ObjectFactory，为Action获取EJB引用，然后在“struts.properties”配置文件的“struts.properties”属性中加以定义，把这个新的工厂安装到应用程序中；创建一个新的拦截器，用来检查每一个Action，并把所需的EJB应用注入其中；或者还可以使用Spring插件来访问JPA或是EJB，http://cwiki.apache.org/S2WIKI/struts-2-spring-jpa-ajax.html这个链接提供了相关的指南。

因为 Spring 框架是推荐使用的库，所以下面我们将重点介绍它的使用。

Spring Framework

安装Spring支持的时候，需要下载Spring插件[17]，并把它拷贝到Web应用中的“/WEB-INF/lib”目录下，或者是在Maven2“pom.xml” 构建文件中添加对Spring插件的依赖：

```
<dependency>
<groupId>org.apache.struts</groupId>
<artifactId>struts2-spring-plugin</artifactId>
<version>2.0.6</version>
</dependency>
```

在“web.xml”配置文件中，你需要添加两部分代码。第一部分用于注册一个 listener，来激活应用程序对象与 Spring 的集成：：

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

然后是要指定 Spring 配置文件所存放的位置。在下面这种情况下，任何以“applicationContext”开头的 XML 文件都会被装载：

```
<context-param>
 <param-name>contextConfigLocation</param-name>
```

[17] http://cwiki.apache.org/S2PLUGINS/spring-plugin.html

```
    <param-
value>classpath*:applicationContext*.xml</param-value>
   </context-param>
```

现在你就可以在 Spring 的支持下进行开发了。所有需要创建的对象都会由 Spring 对象工厂代理创建。如果它知道如何创建对象实例的话，那它就会进行创建，如果它不知道的话，就会把创建工作交回给框架完成。

无论是由 Spring 对象工厂还是由 Struts2 创建的对象，框架都会判断是否有任何所依赖的对象被 Spring 管理着。作为默认的 DI 容器，Spring 会获取所有依赖对象的实例，并在所需的时候传给目标对象。这对 Action 而言尤为重要，因为虽然 Action 本身绝大多数情况下都由 Struts2 来创建，但是它需要由 Spring 来注入所需的业务服务。

使用 Spring 时还有一点要注意的是如何编织依赖关系。对下面这个类而言，Spring 是应该注入 id 值为“service”的 bean 呢，还是类型为“MyService”的 bean 呢？

```
public class MyAction {

 private MyService myService;

 public void setService( MyService service ) {
     myService = service;
}

 public String execute() {
     …
 }
}
```

答案是值为“service”的 id，但这也是可以配置的。在“struts.properties”修改“struts.objectFactory.spring.autoWire”属性就可以配置编织依赖关系的方式。这个属性的默认值是“name”，它可以是以下四个值之一：

值	描述
name	Spring 在 bean 的定义中，使用名/id 值来自动织入 bean。
type	Spring 在 bean 的定义中，使用类名来自动织入 bean。

auto	Spring 自己决定织入 bean 的最佳方法。
constructor	Spring 会通过 bean 的构造器自动织入 bean。

还有一种方法可以完全由 Spring 来管理 Action，不过配置起来就复杂多了。有兴趣的读者可以参考一下 Spring 插件的文档。

数据库

在 Struts2 中集成数据库并没有什么特别的，不过这里仍然有多种不同的方法来访问数据库：

- **通过标签库** ——既然你使用的是基于 Action 的框架，所以这并不是最佳的选择。不过它依然可行；数据可以直接从 JSP 中通过标签库来访问（JSTL 或者自定义的标签），然后将信息格式化。
- **通过依赖注入使用自定义的 DAO**——如果你使用了依赖注入，那么就可以将 Action 中所需的自定义的 DAO 注入到 Action 里面；只要 Action 中存在有对 DAO 的引用，那么它就可以直接调用 DAO 的方法，就如同它自己创建了 DAO 的实例一样。
- **通过依赖注入使用 DAO / ORM** ——如果你在使用高级的 DAO 或者 ORM 库（例如 iBatis 或 Hibernate），那么你就该考虑一下使用一个像 Spring 一样功能齐备的依赖注入框架；Spring 提供了配置和初始化大多数 DAO 和 ORM 库的所有功能，几乎不需要 Action 做什么事情；当 Action 需要执行业务逻辑时，所需的数据访问对象的实例就都已经就绪了。
- **通过业务服务来间接存取**——不是直接使用数据访问对象，而是通过业务服务来间接调用；和上面其他方式一样，业务服务也是通过依赖注入框架注入 Action 的。

附带说一下，如果你打算在项目中使用 Hibernate 作为 ORM 技术的话，那么需要研究一下“OpenSessionInView”过滤器或拦截器。它可以将 Hibernate Session 一直保持连接的状态，直到 JSP 渲染完毕以后才关闭。这样 Hibernate 就可以成功的完成延迟加载。否则的话，Action 或者业务服务或者 DAO 就必须在 JSP 显示数据之前，预先把所有需要的数据全都取出来。

安全

和数据库集成一样，在 Struts2 中提供安全集成也没有什么特别的地方。要在微架构或是应用程序的哪一层来实现授权和验证，是由系统架构师或者开发人员决定的。

进行验证的地方：

- **应用程序之外**——单点登录（SSO）服务器或者应用服务器提供的验证模块就是很好的例子
- **应用程序之内**——通过 HTML 表单或者另外一种 challenge-response 机制

进行授权的地方：

- **URI 级别**——每一个被请求的 URI 都需要和发起请求的用户帐户进行匹配来验证用户是否有权限访问
- **子页面级别** ——页面中的某些部分可能会需要具有特定的访问级别才能查看，修改或者执行操作
- **Action 级别** ——每一个 Action 在执行业务操作之前，都需要判断访问权限
- **业务服务级别**——业务服务中的每一个方法在执行逻辑之前都需要判断访问权限
- **领域对象或者数据库级别**——对一个用于获取数据或者领域对象的通用方法调用而言，调用者的权限不同，返回结果也会不同

在 Struts2 Web 应用的视图中，需要对用户是否有权访问 URL 进行验证。在用户有权访问应用程序的前提下，还可以分为如下的几种情况。

第一种是外部的解决方案，通过 HttpServletRequest 来提供用户帐户信息。这之后用户名和用户的角色信息就可以在 Action 中访问（通过实现 PrincipalAware 接口和在拦截器栈中配置“servlet-config”），并通过 Action 暴露给被渲染的页面。现有的 Struts2 标签都可以用来进行基于角色的访问限制。

下一种情况是用户访问 HttpServletRequest 的帐户信息尚未提供的时候。这时就需要写一个拦截器，用它来进行验证，获取所需要的角色信息，并组装信息。这个过程可以复杂如编写登录表单，登

录和注销代码，验证逻辑等等；也可以简单如从暴露的 API 中获取信息并传递给 Action。

最后一种情况是由第三方类库来做所有的事情，比如 Acegi。Acegi 提供了用于保护 Web 应用安全的所有组件——Servlet 过滤器，自定义的标签库，与 Spring 相集成来保护业务对象和领域对象——这些都是在 Struts2 Web 应用的外部来完成。

当然，只有当Action处理过程中需要授权信息的时候，才需要这种集成。在这种情况下，需要有一个拦截器和一个action接口相配合，向Action提供Authz接口的实例。更多细节请参见 http://struts.apache.org/2.x/docs/can-we-use-acegi-security-with-the-framework.html.

这里的话题与第三方安全类库的集成无关。在使用 Struts2 时的一个与安全相关的概念就是任何 Action 都可以访问值栈中的所有对象。但这种假设并不总是正确的。我们可以通过配置和利用 `com.opensymphony.xwork2.interceptor.ParameterFilterInterceptor` 拦截器（这个拦截器并没有像先前提到的那些拦截器一样有默认的配置）来使得特定的值栈对象对某个 Action 可用或是不可用。更多配置信息请参见拦截器的 JavaDoc。

Ajax

Struts2 中对 Ajax 的支持启动的有些晚。目前很多工作仍在进行中，待到尘埃落定还有很长的路要走。

从根本上说，任何一个 Action 都可以充当数据服务器。向 URI 发出的请求可以得到一个 HTML 片段（直接在 DIV 中显示），一个 XML 文档（通过 XSLT 结果类型）或者是一个 JSON 文档（将 JSP 转化为 JSON 而不是 HTML），之后就在 web 浏览器端由 JavaScript 进行处理。这是到目前为止，从实现的观点看最为稳妥的做法，而且也代表着高效和质量。

按照同样的理念，下面是三个需要一直关注的项目。它们都是很新的项目，需要时间才能发展成熟。

- 使用JSON插件项目来自动完成提供JSON结果的工作，这是个第三方的插件，可以在下面的网址上找到相关信息：

http://cwiki.apache.org/S2PLUGINS/2007/01/11/json-plugin.html

- 作为后单数据源来提供Google Web Toolkit (GWT) 和 Struts2 之间的互操作性，这是GWT 插件项目 – http://cwiki.apache.org/S2PLUGINS/2007/01/10/gwt-plugin.html
- DWR已经增加通过使用自己的框架对远程WebWork2 Action进行调用的支持；WebWork2 是Struts2 的先去，所以所有的技术都是直接可用的；更多的信息可以在 http://getahead.ltd.uk/dwr/server/webwork找到。

Struts2 也提供了实现 Ajax 功能的标签库，它们是：

- **a / link tag** ——使用 Ajax 向服务器发起远程调用
- **form** ——提供对表单域的基于 Ajax 的验证（使用 DWR），并可以通过 Ajax 远程提交表单
- **submit** ——结合基于 Ajax 的表单提交一起使用
- **div** ——通过 Ajax 远程调用来获取 DIV 的内容
- **tabbedpanel** ——通过 Ajax 远程调用来获取标签面板中每一个面板的内容

这些标签都是使用 dojo 库实现的，有些标签正在从 JavaScript 和 dojo 的结合向纯粹的 dojo 实现进行迁移。在不久的将来，Ajax 标签将会被移植到插件中。这样就可以基于不同的 Ajax 库来提供不同的实现，而现在已经有了这方面的需求了。

基于以上种种，Ajax 标签库现在还是被标记为试验阶段。

关于作者

Ian Roughley 是一位技术演讲人、作家及独立咨询顾问，住在马萨诸塞州的波士顿。他具有十多年提供架构设计、开发、过程改进以及指导等方面服务的经验，客户范围小至创业公司，大到财富 500 强前 10 名的公司。他曾经在金融、保险、制药、零售、e-learning、hospitality 和供应链等多个行业中工作过。

他专注于具有实效性且以结果为目标的方法，是开源及以敏捷开发为基础的过程和质量改进的支持者。Ian 参与了 WebWork 项目的开发，也是 Apache Struts PMC 的成员之一，同时还是 No Fluff Just Stuff 座谈会的演讲人。他同时还是 Sun 认证 Java 程序员和 J2EE 企业架构师，以及 IBM 认证解决方案架构师。

参考资料

i http://struts.apache.org/2.x
ii. Don Brown, Struts Ti项目的领导，他在文章中详细介绍了Struts Ti的历史，详情请参见http://www.oreillynet.com/onjava/blog/2006/10/my_history_of_struts_2.html
iii. 你可以访问 http://www.java-source.net/open-source/web-frameworks来获取一份各种web框架的列表。
iv. 只需要更改“struts.action.extension”这个属性的值。
v. Martin Fowler写过一篇文章，对依赖注入进行了完整的描述：http://www.martinfowler.com/articles/injection.html
vi. 单纯从技术角度来讲，应该有 4 种主题——第 4 种是“ajax”主题。Struts2 已经决定在下个版本中把 Ajax 的功能从核心框架里移走，放到插件里面。所以在本节中没有对其进行介绍。
vii. Swingwork 的地址为 https://swingwork.dev.java.net/。这个项目已经停止开发了。
viii. 请参见http://www.jmock.org以获得更多信息。
ix. config browser 插件的文档地址为http://struts.apache.org/2.x/docs/config-browser-plugin.html
x. 这种说法没错，但是如果使用 property 标签，开发人员就可以访问值栈，并使用在开发时还不知道的 key 名来获取国际化文本。
xi. http://tiles.apache.org
xii. http://www.opensymphony.com/sitemesh
xiii. http://cwiki.apache.org/S2PLUGINS/sitemesh-plugin.html
xiv. http://cwiki.apache.org/S2PLUGINS/tiles-plugin.html
xv. http://plexus.codehaus.org
xvi. http://picocontainer.codehaus.org
xvii. http://cwiki.apache.org/S2PLUGINS/spring-plugin.html