简　介

你还在学习基础吗？

你还在为各种问题而烦恼吗？

Java 学习群：188620288

本群（15:00-17:00，20:30-23:00）每天这个时间段都有大神（有腾讯、阿里等内部工程师）免费讲课，分享学习资料！

免费讲课自由学习，只要你是抱着一颗学习的心态来的我们都欢迎！

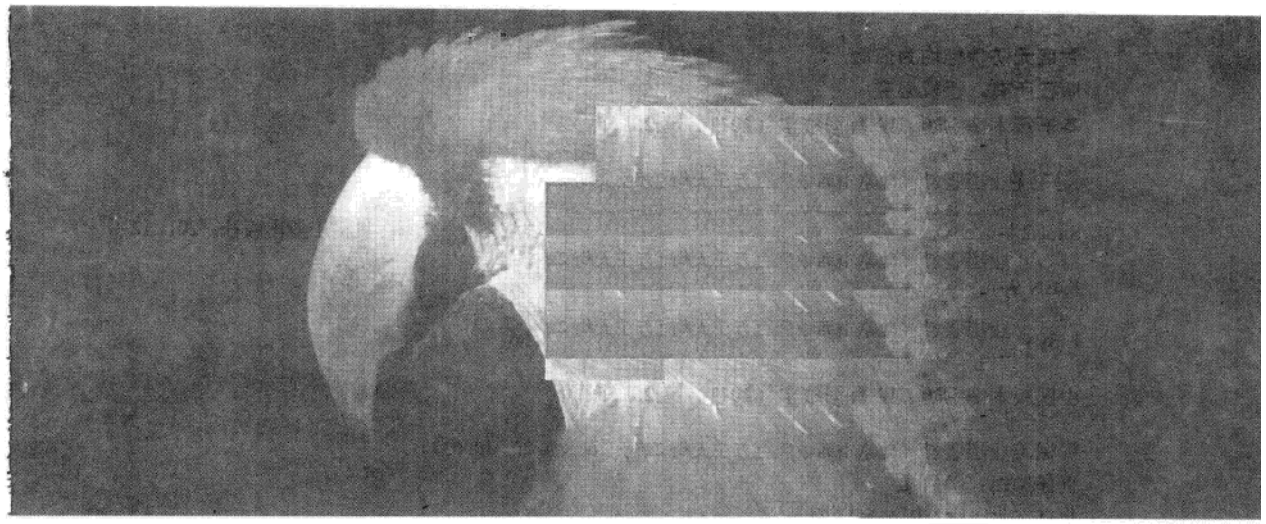
由于本人工作繁忙，所以每天前 20 名可以来免费听课！（java 群主：荡秋千）

揭秘 系列丛书 UNLEASH

Struts2 Internals

Struts2 技术内幕

深入解析Struts2架构设计与实现原理



陆舟 著

机械工业出版社
China Machine Press

本书由国内极为资深的Struts2技术专家（网名：downpour）亲自执笔，iteye兼CSDN产品总监范凯（网名：robbin）以及51CTO等技术社区鼎力推荐。

本书以Struts2的源代码为依托，通过对Struts2的源代码的全面剖析深入探讨了Struts2的架构设计、实现原理、设计理念与设计哲学，对从宏观上和微观上去了解Struts2的技术内幕提供了大量真知灼见。同样重要的是，本书还深入挖掘并分析了Struts2源代码实现中蕴含的大量值得称道的编程技巧和设计模式，这对开发者从Struts2的设计原理上去掌握和悟透Web层开发的要点和本质提供了绝佳的指导。

本书主要分为3大部分，内容安排具有极强的逻辑推理性，章和章之间互相呼应且互为印证。知识准备篇首先介绍了获取、阅读和调试Struts2源代码的方法，以及Struts2源代码的组织形式；然后厘清了Web开发中极易混淆的一些重要概念，以及Struts2的核心技术、宏观视图、微观元素、配置元素等，提纲挈领地对Struts2进行了多角度的讲解。核心技术篇首先分析了Struts2中多种具有代表性的设计模式，然后对Struts2中的精华——OGNL表达式引擎和XWork框架的原理及机制进行了全面深入的分析和讲解。运行主线篇首先对Struts2的两大运行主线——初始化主线和HTTP请求处理主线进行了深入的剖析，然后对Struts2的扩展机制进行了解读和抽象。

图书在版编目（CIP）数据

Struts2技术内幕：深入解析Struts架构设计与实现原理 / 陆舟著. —北京：机械工业出版社，2011.12

ISBN 978-7-111-36696-6

I. S…　II. 陆…　III. 软件工具－程序设计　IV. TP311.56

中国版本图书馆CIP数据核字（2011）第251652号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）
责任编辑：关　敏
北京京北印刷有限公司印刷
2012年1月第1版第1次印刷
186mm×240mm • 24.75印张
标准书号：ISBN 978-7-111-36696-6
定价：69.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991；88361066
购书热线：（010）68326294；88379649；68995259
投稿热线：（010）88379604
读者信箱：hzjsj@hzbook.com

前　言

为什么写本书

在基于 Java 的 Web 开发领域，Apache 旗下的 Struts 无疑具有非常重要的地位。从历史上看，Struts 是出现较早的 Web 层解决方案，它借助 Apache 的影响力积累了大量的客户群体。在之后的岁月中，Struts 吸收合并了另外一个开源社区的精品 Webwork2 成为 Struts2，借助 Webwork2 先进的设计理念和优雅的实现及原先 Struts 社区积累的人气，打造成新一代的 Web 开发解决方案。

无疑，Struts2 赢得了众多开发者的认同，也赢得了市场。目前，Struts2 已经成为 Web 开发解决方案的一股重要力量，并拥有庞大的开发者社群。

对开发者来说，随着 Web 开发技术的不断革新，往往都需要一个优秀的框架作为程序开发的骨架，并在这个基础上完成 Web 层所赋予的任务。而 Struts2，向我们提供了一个完整的 Web 层设计和开发的思路，为我们展示了许多 Web 层设计和开发的最佳实践。可以说，使用 Struts2 作为解决方案，已经成为绝大多数 Web 开发者的首选。

Struts2 的源码中，不仅包含了优秀的 Web 层设计理念，而且蕴含了许多编程技巧和设计模式。通过本书，读者可以加深对 Web 开发职责的理解，从而提高自己的开发水平，拓展自己的技术视野。除此之外，本书所介绍的一些哲学观点，相信也能成为读者重新思考 Web 开发的重要借鉴。

本书面向的读者

1. 学习 Java 语言和 Java EE 技术的中高级读者

对这部分读者来说，Struts2 和 XWork 的核心设计思想以及建立在此基础之上的源代码，是极佳的学习 Java 和 Java EE 技术的参考资料。

2.Struts2 的研究和开发人员

对这部分读者来说，本书的内容能够帮助他们加深对 Struts2 和 XWork 设计原理的哲学理解，并成为他们定制和扩展 Struts2 框架的宝贵参考资料。

3. 开源软件爱好者

在本书中，我们不仅提供了 Struts2 的学习方法，还向大家介绍了一整套完整的开源

软件的学习方法，可以帮助这部分读者提高使用开源软件的效率和质量。

4. 平台开发人员和架构师

Struts2 蕴含的深刻的软件设计理念，可以提高这部分读者对软件架构的认识和设计能力。

本书的主要内容

本书主要分为三个部分：知识准备篇、核心技术篇和运行主线篇。

知识准备篇（第 1 章～第 3 章）。除了介绍和分析解读 Struts2 的基本环境之外，这一篇的重要任务是帮助读者梳理 Web 开发中的主要概念和知识体系。

核心技术篇（第 4 章～第 8 章）。将对 Struts2 所依赖的一些核心技术一一做出详细解读，包括 Struts2 中所用到的设计模式、XWork 的容器实现、OGNL 表达式引擎、XWork 框架的控制流和数据流体系等等。

运行主线篇（第 9 章～第 12 章）。其中主要涉及对 Struts2 两大核心运行主线的研究以及对 Struts2 的扩展机制的分析。

本书的篇章安排有很强的逻辑性，章和章之间互相呼应、互相论证。读者在阅读时可以带着问题到后续章节中去寻找答案，而在每章的小结中，我们会为读者安排每章的概要性问题，大家可以在此做一个回顾并思考问题的答案，从而起到温故而知新的效果。

致谢

首先要感谢 iteye，感谢 iteye 的站长 robbin，是 iteye 给了我 Web 开发知识的启蒙教育。也是在 iteye 上，我第一次接触到了 Struts2 的前身 Webwork 2。而 iteye 多年来在 Web 开发领域所掀起的各种讨论，也成为本书许多重要观点的产生源泉。

感谢 robbin、Readonly、moxie，还有许多曾经活跃在 iteye 上的朋友，你们都是曾经为 Struts2 在国内的推广做出过杰出贡献的人。本书的所有成果，都只是“站在了巨人的肩膀之上”，集合了众家之言而形成的 Web 开发之道。

特别感谢 ahuaxuan 在本书创作过程中给予我的帮助。与你在许多编程哲学上的探讨，每次都能让我受益匪浅。在本书的众多观点中，有许多出自你的连珠妙语。

最后感谢本书的策划编辑杨福川和关敏，你们是我见过的脾气最好、业务能力最强的出版人。我从你们的身上看到了一种坚韧不拔的精神和精益求精的态度。这对我的一生都有帮助。

目　录

第一部分

知识准备篇

对任何事物的研究，总是需要由表及里、由浅入深地进行。作为本书研究对象的Struts2，是一个Web开发框架。因而在本书的开篇，我们试图向读者阐述一些Web开发的基本概念以及开发框架的学习方法。这些内容不仅是全书的精神纲领，也是我们进行后续讨论的基础核心。从这一部分的篇名“知识准备篇”可以看出，本篇的主要目的是为之后我们深入研究框架的实现机理打好坚实的理论基础。

“工欲善其事，必先利其器”。在本书的第1章，我们将首先为读者介绍搭建Struts2开发环境的步骤。一个好的开发环境，是进行框架研究的必要条件。尤其是当我们需要对一个开源框架进行源代码级别的分析时，好的开发环境就犹如一把利剑，能够为我们扫清所有的障碍。除了介绍环境搭建和源码调试的具体方法之外，我们在第1章中还将给出获取Struts2相关资料的途径和方法，并对Struts2项目的结构做初步的介绍。

面对纷繁复杂的框架，有的读者或许已经迷失其中，有的读者或许已经精通使用这些框架的方法并总结出一些行之有效的最佳实践。本书的第2章，我们试图带领读者去探寻日常开发中最为本质的问题：什么是面向对象的概念？什么是框架？我们为什么要引入框架进行编程？在解决这些根本问题的过程中，我们也会同时给出一系列Web开发模式和Web开发过程中的最佳实践，并以此为基础总结出Web开发中可能遇到的主要问题。在第2章的最后，我们还试图总结出正确学习开源框架的方法，这些方法不仅能够帮助那些迷茫于框架之中的程序员找到正确的方向，同时对那些有一定经验的程序员也有相当的借鉴作用。

在了解了框架存在的意义之后，本书的第3章将正式揭开Struts2的神秘面纱，并从宏观上讲述Struts2作为一个表示层框架是如何解决Web开发中的种种问题的。第3章的

内容将涵盖 Struts2 的方方面面，从 Struts2 的历史发展、Struts2 的外部环境、Struts2 的宏观运行主线、Struts2 的微观构成等不同的角度，为读者介绍 Struts2 的概况。

本篇中所有内容的侧重点，都将围绕着“方法论”这三个字展开。因为本书的主旨不仅仅是研究 Struts2 这个框架本身，更是想通过对 Struts2 的研究，总结出一套完整的并且行之有效的 Web 开发的经验和方法。所谓“授人以鱼不如授人以渔”，希望读者在阅读本书的过程中，始终能够站在一个比 Struts2 本身更加高的高度来看待整个 Web 开发，这样才能在不断发展的历史洪流中立于不败之地。

第 1 章　厉兵秣马——开发环境准备

1.1　准备源代码阅读环境

在开发与分析任何一个开源框架之前，都需要安装与配置基本的开发环境和源代码的阅读环境。这一系列内容将包括：安装与配置 JDK、安装开发调试 IDE、安装与配置 Web 服务器、搭建开发调试环境等。

1.1.1　安装与配置 JDK

Struts2 的运行环境要求 Java 5 以上的版本，所以 JDK 的版本必须为 1.5 或者 1.5 以上。打开 http://www.oracle.com/technetwork/java/javase/downloads/index.html 页面，我们可以下载到最新的 JDK 安装程序，下载页面如图 1-1 所示。本书所有的示例代码都运行在 JDK 1.6 之上，大家可以下载这个版本的 JDK 运行本书中的所有代码示例。

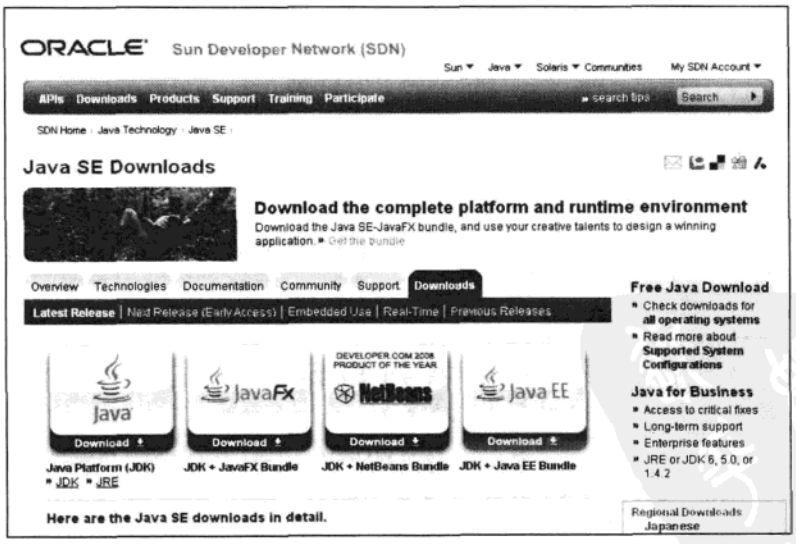


图 1-1　JDK 下载首页

安装完 JDK 后，需要正确配置 Java 运行时必需的环境变量值，它们是：JAVA_

HOME、CLASSPATH 和 PATH。无论是什么操作系统，都能够支持多个 JDK 版本的共存，读者可以根据应用程序的实际需求对不同的 JDK 版本进行管理。

正确管理 JDK 版本的方式，是在 JVM 运行时指定 JDK 版本对应的环境变量。为了方便起见，我们往往为操作系统本身指定一个系统级别的环境变量。例如，Windows 平台上的系统环境变量可以在“系统属性”的“高级”选项卡中找到，并在其中配置 JAVA_HOME、PATH 和 CLASSPATH 值。图 1-2 是 Windows XP 操作系统中为系统添加 CLASSPATH 环境变量的例子。

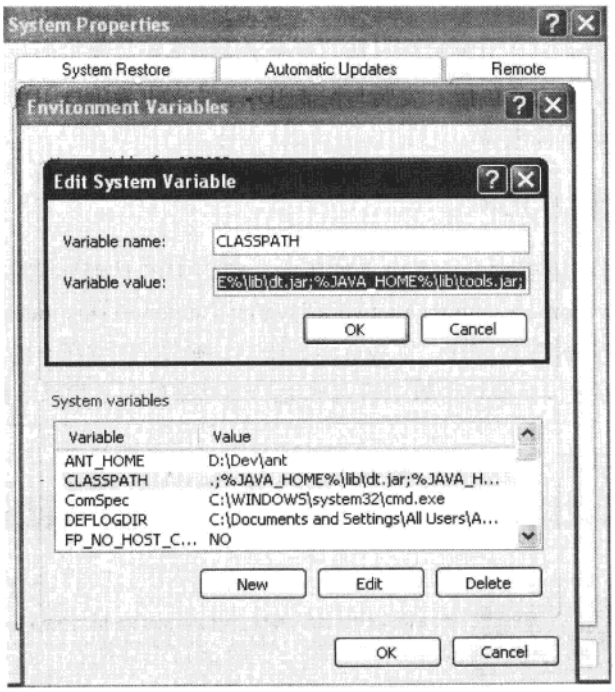


图 1-2 Windows XP 上配置 CLASSPATH

多 JRE 共存时的版本管理

Java 语言对于运行环境的管理比较宽松，在一个操作系统中可以同时运行多个 Java 程序的进程，每个 Java 进程所依赖的 JRE 版本也可以各不相同。当某一个 Java 进程启动时，操作系统会依次按照 Java 启动进程的当前目录、当前目录的父目录、PATH 值中所指定的目录进行 JRE 寻址，找到第一个返回的 JRE 版本并运行。因此，一个简单而有效的指定 JRE 寻址的方式是在启动 Java 进程的脚本中通过指定当前运行程序的 PATH 值来定制特定版本的 JRE 的执行环境，从而达到对不同版本的 JRE 进行

管理的目的。

JRE 管理和 CLASSPATH 的加载顺序问题是 Java 开发中最为基本的问题，它牵涉的是 Java 程序所依赖的最为基本的底层环境的配置。尤其是当一个应用程序运行在一个高级的商业应用服务器如 Websphere 或 Weblogic 之上时，我们应该密切关注程序的 JRE 运行参数和版本以及 CLASSPATH 的加载顺序（先加载优先原则），因为这些商业应用服务器往往有自定义的 JRE 管理机制和 CLASSPATH 的加载方式，而这两大内容，也将直接决定 Web 应用的运行特征。因此，在商用服务器中，我们往往通过自行修改服务器的启动脚本来设定服务器运行所依赖的 JAR 版本和 Library 加载方式（在某些服务器中，可以通过控制台进行配置）。

安装并配置完成后，我们可以在命令行窗口中输入 java -version 命令来检测一下当前的 JDK 运行版本。如果配置完全正确，当前客户端的 JRE 运行版本会显示出来，如图 1-3 所示。

图 1-3　JDK 安装成功

1.1.2　安装 Eclipse 与源码调试

在成功安装和配置 JDK 后，我们还需要安装进行 Java 开发调试的 IDE。目前比较常用的 Java 开发 IDE 主要有 Eclipse 和 NetBeans 等，读者可以任意选择自己习惯的 IDE 作为开发工具。在本书中，我们以 Eclipse 为例，着重介绍在 Eclipse 中开发与调试源码的方法。读者也可以举一反三，在其他 IDE 中做相应的尝试。

Eclipse 是一个界面友好的开源 IDE，并支持成千上万种不同的插件，为我们进行代码分析和源码调试提供了极大的便利。我们可以在 Eclipse 的官方网站（http://www.eclipse.org/）找到 Eclipse 的各个版本并下载安装。

本书不会对 Eclipse 的使用细节进行详细介绍，不过为了帮助读者更好地进行开源框架的研究，尤其是对开源框架源码的解读，我们在本章中将陆续给出一些结合 Eclipse 进行源码阅读的方法和最佳实践。相信这些方法对各个层次的读者都会有一定的启示。

在本节中，我们将首先列出一些 Eclipse IDE 中常用的快捷键（参见图 1-4、图 1-5 和图 1-6），这些快捷键在进行源码分析时非常有用。

- Ctrl+Shift+T——根据类名查找相应的类
- Ctrl+T——选中某个类或方法，使用此快捷键能够查看类或方法的组织结构
- Ctrl+Shift+G——选中某个类或方法，使用此快捷键能够查找所有调用类或者方法

的出处

快捷键是日常开发调试中最为仰仗的一个技巧。Eclipse 中的快捷键可谓博大精深，我们在这里不再一一列举。读者可以在实际开发中不断摸索并牢记这些快捷键，因为它们也是日常开发中必不可少的一部分。读者也可参照 Eclipse 中的这些快捷键，在其他的 IDE 中找到相应的快捷键设置。

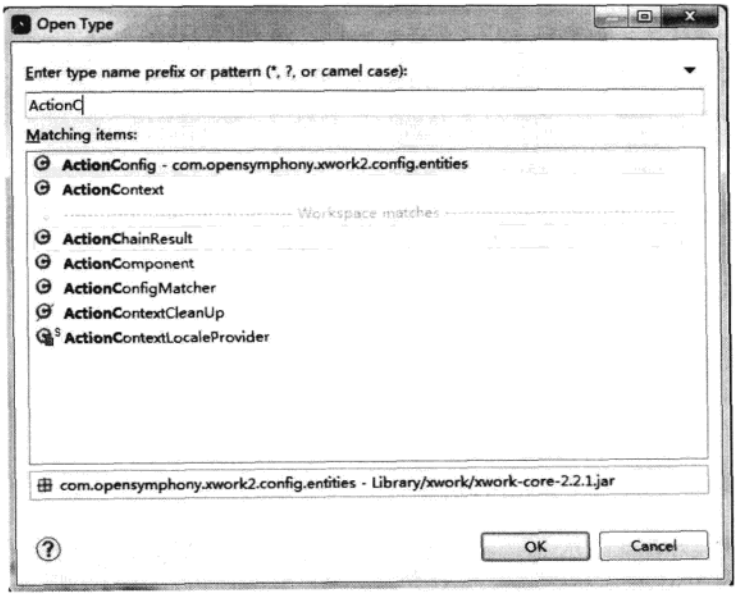


图 1-4 在 Eclipse 中使用 Ctrl+Shift+T

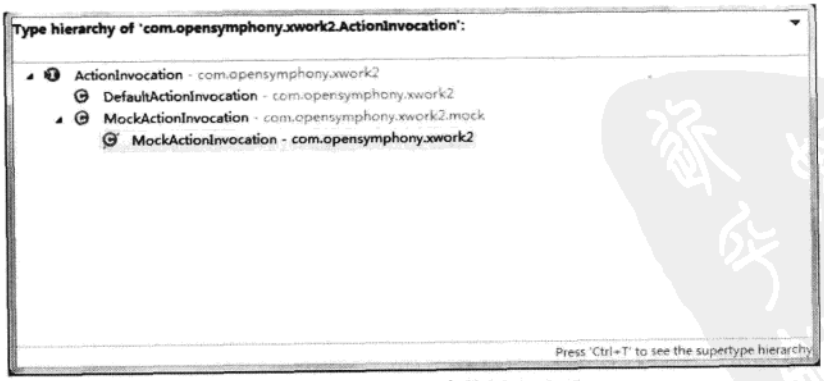


图 1-5 在 Eclipse 中使用 Ctrl+T

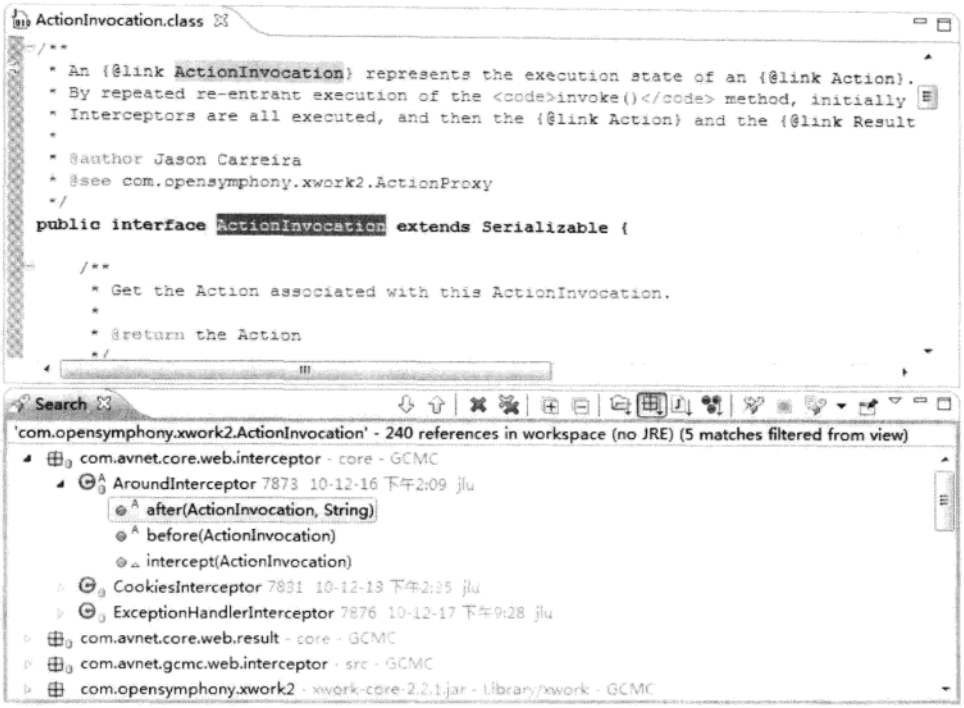


图 1-6　在 Eclipse 中使用 Ctrl+ Shift + G

1.1.3　安装与配置 Web 服务器

本书所涉及的绝大多数内容都属于 Web 开发的相关范畴，因而在 JDK 和 Eclipse 安装完成之后，我们需要紧接着安装 Web 服务器用以提供 Web 程序的运行环境。

在众多的 Web 应用服务器中，Apache 开源社区的 Tomcat 是最常用的 Web 服务器之一，读者可以在 http://tomcat.apache.org 找到各个版本的 Tomcat 服务器，本书以 Tomcat 6.028 为例，请大家下载适合自己的版本并安装。

安装完成后，可以在 Tomcat 安装目录下的 bin 目录中找到运行 Tomcat 的命令，运行成功的示意图如图 1-7 所示。

另外一款非常流行的轻量级的 Web 应用服务器是 Jetty。相对于 Tomcat 来说，Jetty 具有无须安装（其 API 以一组 JAR 包的形式发布）、速度更快等特点，成为当前众多程序员进行 Web 开发调试的首选。在 1.1.4 节中，我们将结合 Eclipse IDE 为大家讲解使用 Jetty 进行 Web 开发和调试的过程。

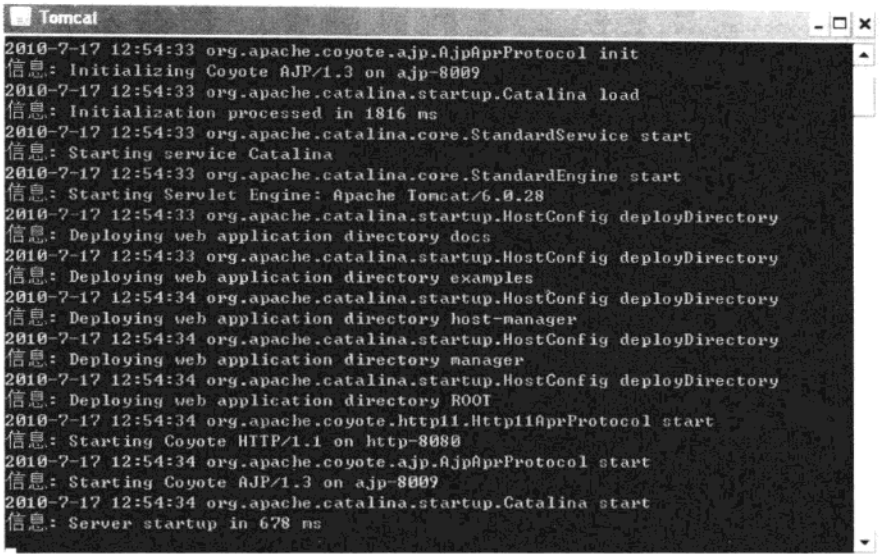


图 1-7 Tomcat 安装成功

1.1.4 在 Eclipse 中使用 Jetty 搭建 Web 开发环境

在 Eclipse IDE 中建立了一个 Web 项目后，我们可以通过引入 Jetty 搭建起一个不依赖任何插件的 Web 开发调试环境。使用 Jetty 搭建 Web 开发环境的基本步骤如下。

1.1.4.1 构建项目的基本目录结构

搭建 Web 开发调试环境的第一个步骤是建立一个 Web 项目的基本目录结构，如图 1-8 所示。

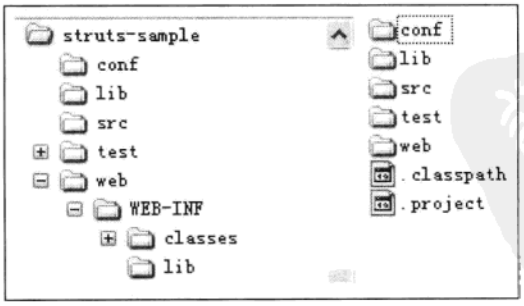


图 1-8 Web 项目的基本目录结构

这里简单解释一下这些基本目录结构的作用：

- src(source folder)——存放所有的 Java 源代码
- conf(source folder)——存放所有的配置文件
- test(source folder)——存放所有的 Java 测试代码和调试代码
- web——Web 项目的根目录，其子目录 WEB-INF 以及 classes 和 lib 是构成 Web 项目所必需的基本结构
- lib——存放 JAR 包，但是这里存放的 JAR 文件仅在开发调试时使用

读者可以根据项目的实际情况，创建适合于项目自身需求的目录结构，并导入到 Eclipse 中使之成为 Eclipse 项目，之后我们的开发调试都将围绕着 Web 项目的目录结构展开。

目前，使用 Maven 作为开发环境进行构建的开发人员越来越多。Maven 是一个非常优秀的项目构建和项目管理的工具。我们在这里并未介绍 Maven 的主要原因在于 Maven 的使用需要额外的开发环境搭建，甚至可能需要编写符合项目自身条件的 POM 文件，而这些内容并非本书介绍的重点。因而，对其有兴趣的读者可以自行使用 Maven 进行项目构建，**而使用 Maven 进行项目构建和 JAR 文件依赖关系的管理，也是我们向读者推荐的做法。**

1.1.4.2　引入 Jetty 服务器

搭建 Web 开发环境的第二步，是引入 Jetty 服务器作为我们的 Web 容器。引入 Jetty 的方法非常简单，只需要将 Jetty 服务器相关的 JAR 文件加入到项目的 CLASSPATH 中即可。整个过程如图 1-9 所示。

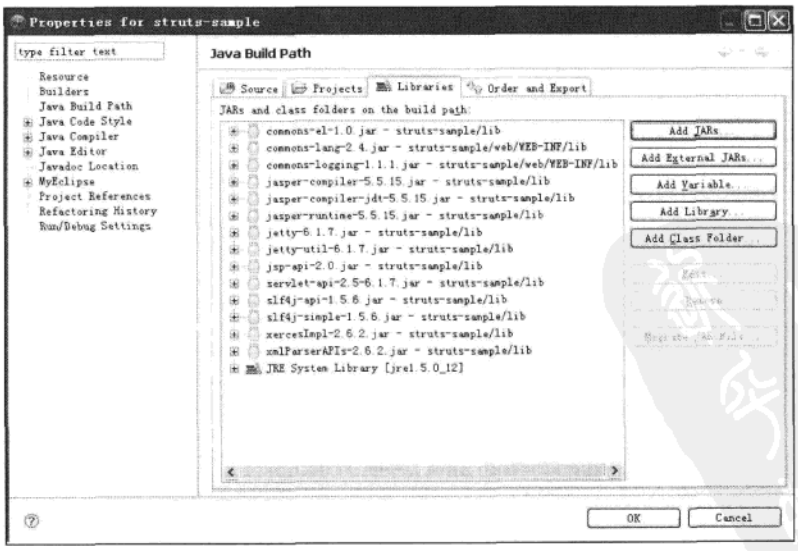


图 1-9　将构成 Jetty 服务器必要的 JAR 包加到 CLASSPATH 中

这个步骤完成之后，我们就可以将 Jetty 作为我们的 Web 容器提供者。在这里，我们可以看到 Jetty 服务器相对于其他 Web 服务器的一个重要区别在于它**不需要额外安装**，引入 Jetty 的过程只不过是将几个 JAR 文件加入到 CLASSPATH 中而已。这样一来，也就不存在平台依赖性和兼容性问题了。

1.1.4.3 启动 Jetty 进行开发调试

启动 Jetty 服务器进行基本的 Web 开发调试的步骤非常简单，只需要在项目中编写一个 JettyStarter 的 Java 类，并将这个类作为一个基本的 Application 执行即可。JettyStarter 的源码如代码清单 1-1 所示。

代码清单 1-1 JettyStarter.java

```
package runtime;

import org.mortbay.jetty.Connector;
import org.mortbay.jetty.Server;
import org.mortbay.jetty.nio.SelectChannelConnector;
import org.mortbay.jetty.webapp.WebAppContext;

/**
 * Jetty Server 启动类
 *
 * @author
 */
public class JettyStarter {

    /**
     * @param args
     * @throws Exception
     */
    public static void main(String[] args) throws Exception {
        long begin = System.currentTimeMillis();
        Connector connector = new SelectChannelConnector();
        connector.setPort(Integer.getInteger("jetty.port", 80).intValue());

        WebAppContext webapp = new WebAppContext("web", "/struts_example");

        Server server = new Server();
        server.setConnectors(new Connector[] { connector });
        server.setHandler(webapp);
        server.start();

    System.out.println("Jetty Server started, use " + (System.currentTimeMillis()
        - begin) + " ms");

    }
}
```

有了 JettyStarter 这个类之后，我们可以直接将这个类以 Application 方式来运行。此时，Jetty 服务器就会启动（如图 1-10 所示），从而成为一个 Web 容器的提供者。当然，我们也可以使用调试模式（Debug）执行这个类，从而使得整个 Jetty 服务器处于调试模式。

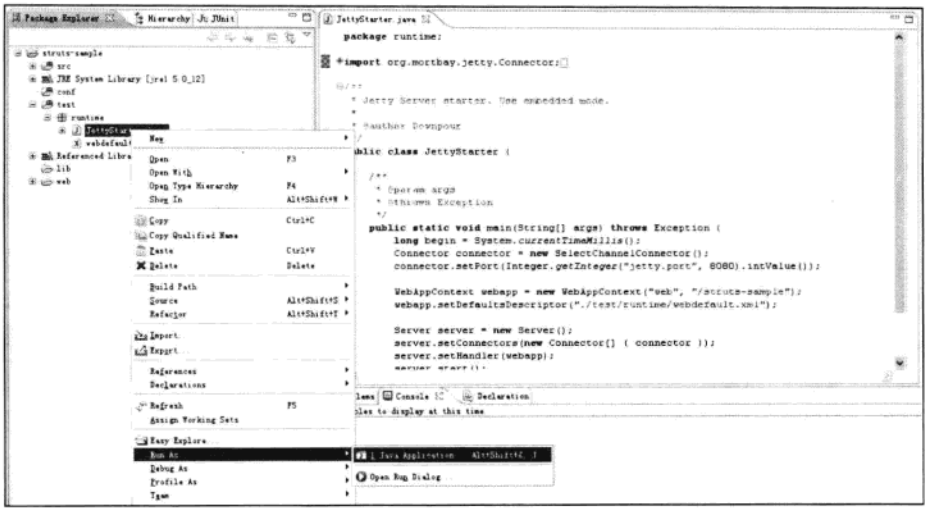


图 1-10　JettyStarter 启动界面

启动成功后，我们会在控制台看到成功启动的日志：

```
 [main] INFO org.mortbay.log - Logging to
org.slf4j.impl.SimpleLogger via
org.mortbay.log.Slf4jLog
 [main] INFO org.mortbay.log - jetty-6.1.7
 [main] INFO org.mortbay.log - Started
SelectChannelConnector@127.0.0.1:8080
Jetty Server started, use 1547 ms
```

此时，Jetty 服务器被引入作为 Web 应用的开发调试服务器，Web 开发调试环境也就成功搭建起来。我们可以通过浏览器对 Web 应用进行访问，而其中的 IP 地址、端口、Web 应用的入口名称，都可以在 JettyStarter 这个 Java 类中进行 API 级别的相应设置。

如果 JettyStarter 运行在调试模式下，我们就可以在 Web 项目中加入熟悉的调试元素，例如加入断点（Breakpoint）、加入监视（Watch）、单步运行（Step by），从而对整个项目的运行状态进行调试和监控。

相比许多其他的开发环境搭建方式，本书所介绍的 Web 开发环境至少拥有以下的好处：

- ❑ 开发环境的建立并不依赖于任何 IDE 或者相关的插件，只需要运行某个 Java 类就可以进行调试
- ❑ 开发环境的建立不依赖于任何外部 Web 服务器的安装，无论将项目迁移到什么样的 IDE 或者操作系统，只要是有 Java 运行环境的地方，都可以直接运行
- ❑ 在项目中内置开发环境的做法，降低了程序员的学习成本

1.2 获取 Struts2

1.2.1 Struts2 的相关资源下载

在 Struts 的官方网站（http://struts.apache.org）可以找到所有 Struts 项目相关的信息。由于历史原因，Struts 项目分为两个截然不同并互不兼容的版本 Struts1.X 和 Struts2.X（我们通常把 Struts1.X 的版本称为 Struts，而 Struts2.X 的版本称为 Struts2）。读者在获取相关信息时，需要正确区分不同版本的 Struts 项目，本书讲述的所有内容都是围绕着 Struts2.X 进行的。有关 Struts 项目的发展历史和它们之间的不同，在其他章节中将会做出更加详细的介绍。Struts 的官方网站如图 1-11 所示。

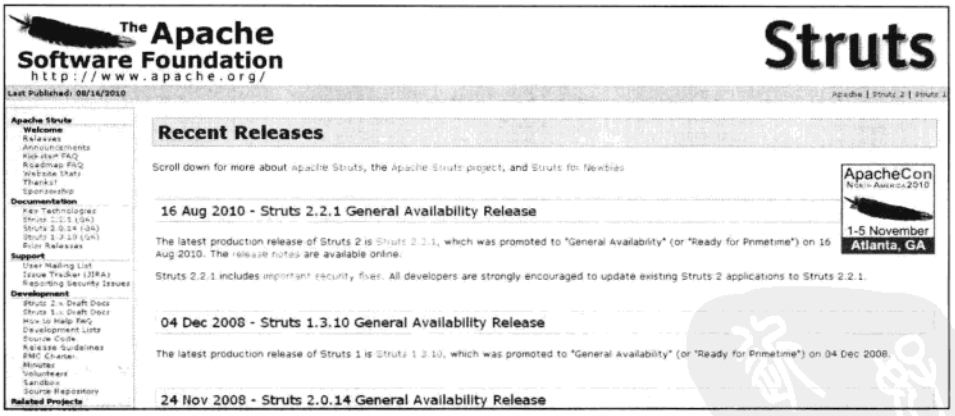


图 1-11 Sturts2 的主页

在左侧的“Documentation”分类中，我们能够找到 Struts2 的最新版本 Struts 2.2.1（GA）的链接，点击该链接后就进入到了 Struts2 相应版本的主页，其中可以找到 Struts2 的下载链接（一个很大的蓝色 Download Now 按钮），如图 1-12 所示。

点击“Download Now”按钮，将会看到 Struts2 的下载页面，其中包含各种不同类型

的 Struts2 的下载包，如图 1-13 所示。

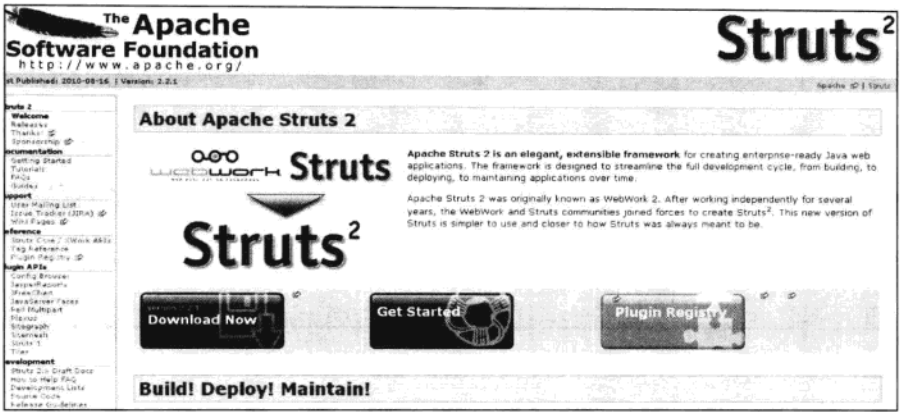


图 1-12　Struts 2.2.1 主页

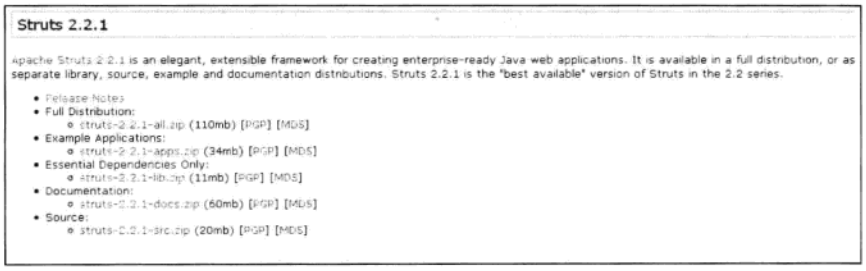


图 1-13　Struts2 的下载页面

在这里点击下载 struts-2.2.1-all.zip，就能获得 Struts2 项目所有的相关资源。在其中，我们可以看到 Struts2 的分发包、Struts2 的文档（Reference）、Struts2 的源码、Struts2 的示例项目等。接下来，我们就来看看 Struts2 项目基本的目录组织结构。

1.2.2　Struts2 项目的目录组织结构

打开 struts-2.2.1-all.zip 并将其解压到任意目录后，我们可以在其中找到 Struts2 的所有相关资源，如图 1-14 所示。

Struts2 项目的目录结构中主要包含了 4 个目录：apps、docs、lib 和 src，这些目录中所存放的基本内容分别介绍如下。

❑ **apps——存放了所有 Struts2 的示例项目**

位于 apps 目录下的所有 war 都是可以部署到 Web 服务器中直接运行的 Web 应用。这

些 Struts2 的示例项目对学习 Struts2 有相当大的指导作用。当我们对 Struts2 的特性使用有疑问时，可以通过直接研究这些项目的源码获得足够的支持。

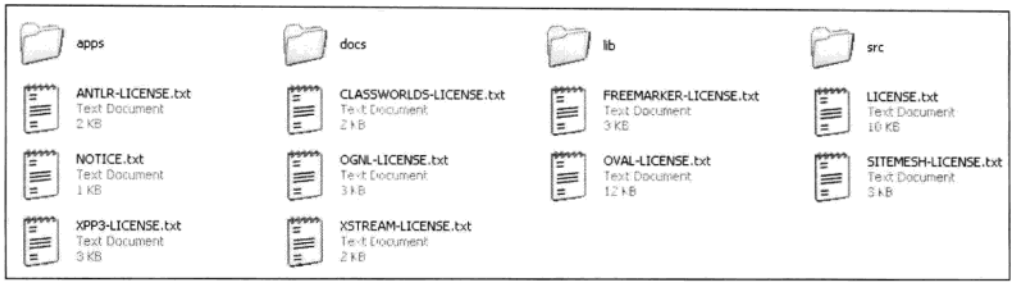


图 1-14 Struts2 分发包的目录结构

❑ **docs——存放了所有 Struts2 与 XWork 的文档**

docs 目录中存放的 Struts2 与 XWork 的相关文档基本上是以 Wiki Page 的形式出现的。这主要是由于 Struts2 来源于 Webwork2，因而其文档的表现形式也是一脉相承的。Wiki Page 形式的文档的好处在于可以将需要表述的框架特性划分为若干个专题，针对每个专题，有相应的理论知识讲解和示例代码的印证。

❑ **lib——存放了所有 Struts2 相关的 JAR 文件以及 Struts2 运行时所依赖的 JAR 文件**

lib 目录是一个完整的依赖资源集合。Struts2 项目运行所需要的 JAR 文件都位于这个目录下。我们可以看到，Struts2 的许多特性都是以插件的形式提供的，因而在 lib 目录下除了 Struts2 项目的基本依赖之外，绝大多数都是插件资源。

❑ **src——存放了所有 Struts2 的源码，以 Maven 所指定的项目结构目录存放**

Struts2 本身是根据 Maven 所指定的项目目录结构进行编写的，所以 src 目录的组织结构也与 Maven 所规定的目录结构相同。这种组织结构的好处在于我们可以在 src 目录中找到所有 Struts2 相关资源的单个源码文件。源码的另外一种组织形式我们将在下一节中着重介绍，读者可以就两种不同的形式进行比较。

1.3 Struts2 源码的初步研究

1.3.1 源码的组织形式

如果仔细浏览 Struts2 分发包中的 src 目录，我们会发现在 Struts2 的分发包中提供的源码是以目录形式存放的。这种源码组织形式允许读者访问单个源码文件，并可以在资源管理器中清晰地看到源码的目录组织结构。除了这种基于目录形式的源码组织方式以外，

还有另外一种组织源码的方式，就是将所有的源码打包成一个 JAR 文件或者 ZIP 文件。这种单个文件形式的源码在结合 IDE 进行管理时更加简单，故而受到广大程序员的欢迎。

虽然 Struts2 的分发包中没有提供这种形式的源码，我们依然可以在网络上查找得到，例如在 http://search.maven.org/ 中，我们可以查询到 Struts2 各个版本的分发包和对应的源码文件。打开上述链接后，我们可以根据实际要求进行源码查询，如图 1-15 所示。

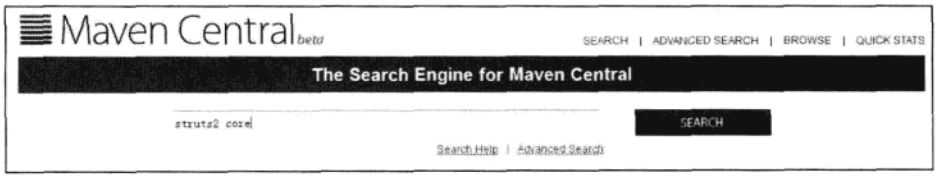


图 1-15　Struts2 源代码查询

点击查询后，我们就能在查询结果中看到以 JAR 文件形式进行管理的资源文件，如图 1-16 所示。

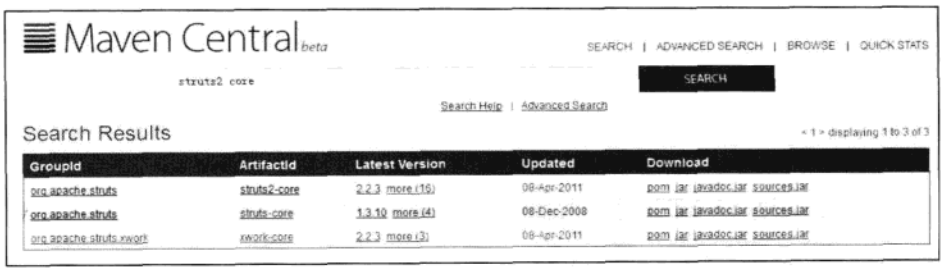


图 1-16　以 JAR 文件形式管理的 Struts Core 的各项资源列表

其中不仅包含 Struts2 的 JAR 文件（struts2-core-2.2.3.jar），还包含 对应的源码文件（struts2-core-2.2.3-sources.jar），大家可以直接点击链接进行下载。

提示　细心的读者或许可以发现，上面提供的含有 Struts2 源码的网站，实际上是一个代码资源库的一部分（Maven2 提供的一个网络代码库，用于在 Maven2 的项目建立时，作为一个基本依赖）。实际上在这个站点中，我们还可以找到许多其他开源框架的各种形式的源代码，本书中所依赖的绝大多数的源代码都是从此处获取的。

1.3.2　调试 Struts2 源码

在 Struts2 的分发包中，我们可以获取 Struts2 的绝大多数信息，包括 Struts2 的 Library 文件、Struts2 运行时依赖的 Library 文件和 Struts2 的源码等。接下来，我们将重

点介绍一下如何在 Eclipse IDE 中调试 Struts2 的源码。读者也可以顺着思路，在其他的 IDE 中获得相应的支持。

什么是调试 Struts2 源码呢？我们在日常开发中，经常听到的是调试某一段应用程序。将应用程序运行于调试（Debug）模式下，我们就可以采用 IDE 提供的一些特性进行程序调试：设置断点（Breakpoint）、加入监视（Watch）、单步运行（Step by）等等。通过对程序的调试，能够迅速了解程序运行的状态，从而帮助我们快速定位问题并解决问题。我们在这里所说的调试 Struts2 的源码，**实际上是在向读者推荐一种阅读源码的方式**。

传统的阅读开源框架源码的方式，是以逐个 package 或者逐个 module 的方式对框架的源码进行阅读。**不过我们在这里明确表示反对这种阅读源码的方式**。因为任何一个程序只有处于运行状态时，才能突显出其内在的价值。我们鼓励读者从程序运行角度来看问题，而不是死板地对着源代码本身。对于一个程序员来说，**只有抓住一个程序或者框架的运行时状态，才能够对它们有更加深刻的理解**。一个静态的程序是死的，只有处于运行状态的程序，我们才能看到其真正的生命力。接下来，我们就来看看在 Eclipse 中对一个开源框架进行源码调试的步骤和方法。

首先在 Eclipse 中创建一个新的项目，加入运行 Struts2 所需要的 JAR 文件，并将它们加到项目的 CLASSPATH 中，成功后的界面如图 1-17 所示。

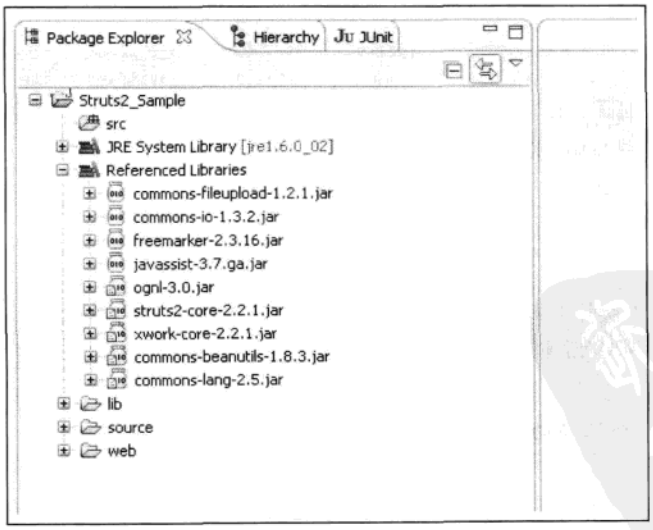


图 1-17 Eclipse 中新建项目截图

接下来，我们来为加入到 CLASSPATH 中的这些 JAR 文件附上其所对应的源代码。

以“struts2-core-2.2.1.jar”为例，右键单击“struts2-core-2.2.1.jar”，选择 Properties（Alt + Enter）选项卡，弹出 JAR 文件的属性选项框，如图 1-18 所示。

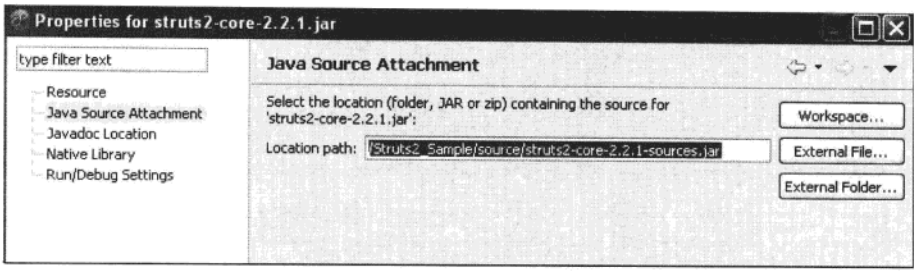


图 1-18　为 JAR 文件附上源码包

选择选项卡左侧的“Java Source Attachment”选项，点击输入框右侧的按钮，指定相应的源代码所在的目录或者源代码所在的 JAR 文件（ZIP 文件），点击 OK 按钮，Library 文件就被附上了相应的源代码（此时，我们会发现 JAR 文件形式的源码组织形式在管理上的方便性，我们甚至可以把源码文件存放到 Workspace 的某个 Repository 目录后被所有的 Library 文件引用）。

在 Library 文件被附上源代码之后，展开 Library 文件，并双击其中的 class 文件，就能看到 class 文件所对应的源码，如图 1-19 所示。

图 1-19　在 Eclipse 中查看源码

Library 文件被附上其对应的源码后，我们就可以使用 Eclipse IDE 中的功能和快捷键对源码进行查看和分析了。

除了基本的源码查看功能，Eclipse IDE 还能够对正在运行的程序进行源码级别的调试。例如，当一个程序运行在调试（Debug）模式时，我们可以直接打开 Struts2 的源码，使用断点功能，进入单步调试模式，并查看当前 Struts2 中类的运行状态，如

图 1-20 所示。

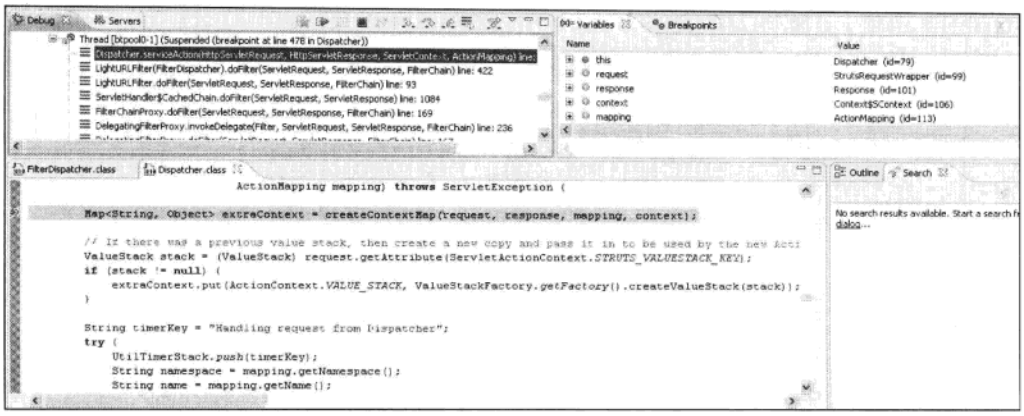


图 1-20 在 Eclipse 中调试源码

为 Library 附上源码，是 Eclipse IDE 在源码级别调试上的一个亮点。这为广大程序员进行框架级别的源码调试带来了极大的便利。在进行框架研究时，我们实际上根本无须将庞大的框架源程序作为一个项目下载到本地并花大力气把项目搭建起来。通过为 Library 附上源码的方式，我们可以轻易地查看框架的源码、获得框架中的实现类之间的逻辑结构和层次关系。

调试源码是除了日志输出以外，最有效甚至是唯一的了解程序内部运行时状态的方式。因而，**调试源码也是本书最为推荐的一种源码级别学习方法**，读者不仅可以将这种学习方法运用在 Struts2 上面，也可将其用于其他的开源框架上。

1.4 小结

本章中，我们首先向读者介绍了开发 Struts2 应用程序所必需的开发环境的搭建过程。这一过程包含：安装与配置 JDK、安装与配置 Eclipse、安装与配置 Web 服务器。除此之外，着重介绍了在 Eclipse 中搭建 Web 开发环境并进行源码级别调试的详细过程。

读者应该明确的是：学习一个框架，尤其是分析框架的内部实现机理，最为有效的途径就是对框架进行源码级别的调试并在调试的过程中深入探究框架内部元素在程序运行过程中的执行顺序和执行状态。这也是本书花许多笔墨向读者介绍开发环境搭建过程的初衷。希望广大读者能够融会贯通并举一反三，根据实际情况搭建起自己顺手的开发环境。

阅读完本章后，大家是否已经能够熟练掌握开发环境的搭建了呢？

- 如何在 Eclipse 中使用 Jetty 搭建 Web 开发环境？
- Struts2 的分发包中包含哪些主要内容？
- 开源项目的源代码有哪些组织形式？
- 如何获取一个开源框架的源码？
- 如何在项目中进行框架的源码调试？

第 2 章　固本清源——Web 开发浅谈

现今，在谈到 Web 开发有关的话题时，程序员们总是热衷于讨论一些我们耳熟能详的 Web 开发框架，如 Struts2、Spring、Hibernate 等。有些程序员将这些框架奉为宝典，并且趋之若鹜地挖掘框架的方方面面、比较各种开发框架的优劣。对这些框架的熟悉与否，似乎已成为衡量一个程序员是否精通 Java、精通 J2EE 开发的事实标准。甚至在广大程序员求职的过程中，这些主流的开发框架的知识细节也常常成为面试中必考的元素，答不上这些问题，无疑会为求职蒙上一层阴影。

面对这些框架，大家是否真的思考过，**我们为什么要学习这些框架？这些框架到底从何而来？框架的本质到底是什么？使用框架，又能够为我们的开发带来什么样的好处呢？**在深入分析 Struts2 及其源码之前，我们首先必须弄清楚这些比框架更为核心的问题。因为只有了解了**为什么**，我们才能知道**怎么做**，知道如何才能**做得更好**。

2.1　面向对象浅谈

在谈框架之前，我们不得不首先面对一个比框架更为重要的概念，那就是**面向对象**的概念。面向对象的概念是一个看起来、听起来简单，实际却蕴含着丰富内容的概念。众多的国内外学者为了讲清楚这个概念，采用了各种的不同的比喻、也给出了多种多样的代码示例，还为面向对象的概念建立起一套完整的理论体系。

不过迄今为止，能够完整地将面向对象的来龙去脉讲清楚、讲透彻的毕竟还是少数。随着编程语言从早期的“面向过程式”的 C 语言发展到后来的 C++、Java，直至近几年来非常热门的 Ruby，面向对象的基本概念已经逐渐成为编程语言的核心设计法则。因而“面向对象”的概念也逐渐成为每个程序员都认同，并且在日常编程过程中遵循的最高纲领。

限于篇幅，我们实在无法涉及面向对象概念的方方面面。不过我们可以将话题聚焦在构成面向对象概念最基本的元素之上，这个基本元素就是：**对象**。在接下来的章节中，我们将分析**对象的构成模型以及对象的关系模型**，并以此为基础阐述在面向对象编程过程中的一些基本观点。

2.1.1　对象构成模型

2.1.1.1　对象的构成分析

作为面向对象编程最基本的构成元素，对象是由一个叫做类（Class）的概念来描述的。因此，针对对象构成分析的研究，也就转化为针对编程语言中类的构成分析。以 Java 语言为例，我们可以对 Java 语言中类的定义进行一些构成上的分析，如图 2-1 所示。

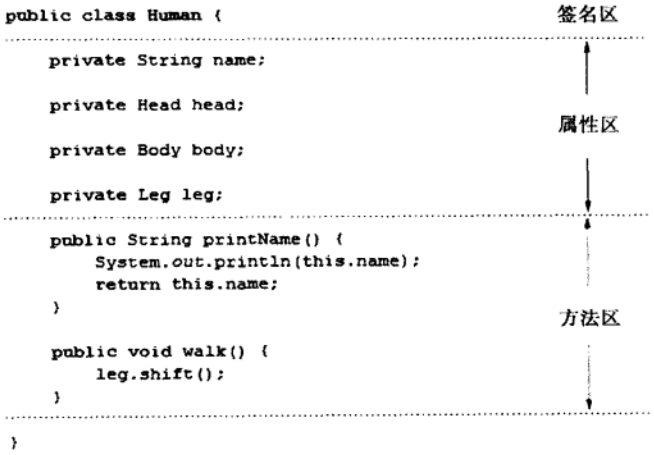


图 2-1　对象的构成分析

在图中，我们可以看到构成一个对象的基本要素主要有：

- **签名**（Signature）——对象的核心语义概括
- **属性**（Property）——对象的内部特征和状态的描述
- **方法**（Method）——对象的行为特征的描述

在进行面向对象的编程时，首先要做的就是**对世间万物进行编程元素的抽象**。这个过程说白了，就是通过使用编程语言所规定的语法，例如类（Class）或者接口（Interface）来表达事物的逻辑语义。在图 2-1 中我们所谈到的构成一个对象定义的基本要素，实际上不仅反映出我们对世间万物的抽象过程，也是人类使用高级编程语言来实现外部世界表述的基本方式。

从图 2-1 中我们可以看到，**签名**用以描述事物的核心语义，它的作用实际上是界定我们所描述的事物的范畴。而在对象的内部，作为对象内部构成的重要元素，**属性**和**方法**刚好从两个不同的角度对事物的内在特性给予了诠释。其中，**属性**所勾勒的是一个对象的**构成特性和内部状态的特性**；而**方法**则表达了一个对象的**动态行为特性**。这就像我们人一样，人由头、躯干、四肢构成，它们可以看作是人这个对象的“属性”。与此同时，人具

有“直立行走”的行为特性，我们可以定义一个“方法”来模拟这一行为。

以上这些分析，我们还停留在语法这个层面，因为无论是属性还是方法，它们都是Java语言的原生语法支持。将事物抽象成为对象，并且赋予这个对象属性和方法，是一个很自然的编程逻辑，这也符合面向对象编程语言的基本思路。不过我们也同时发现在实际编程过程中，对象将表现为三种不同的形态和运作模式。

❑ 属性－行为模式

这种模式是指一个对象同时拥有属性定义和方法定义。这是对象最为普遍的一种运行模式，绝大多数对象都运作在这种模式之上。

❑ 属性模式

这种模式是指一个对象只拥有属性定义，辅之以相应的setter和getter方法。Java规范为运行在这种模式下的对象取了一个统一的名称：JavaBean。JavaBean从其表现出来的特性看，可以作为数据的存储模式和数据的传输载体。

❑ 行为模式

这种模式是指构成一个对象的主体是一系列方法的定义，而并不含有具体的属性定义，或者说即使含有一些属性定义，也是一些无状态的协作对象。运行在这种模式之下的对象，我们往往称之为“无状态对象”，其中最为常见的例子是我们熟悉的Servlet对象。

我们发现，对象的运行模式的划分是根据对象的构成特点进行的。这三种对象的运行模式在我们日常编程中都已经见过并且亲自实践过。接下来的章节，我们将针对后两种构成模式做进一步的分析。

2.1.1.2 属性对象模式

属性对象模式又称为JavaBean模式。这种对象的运行模式我们在日常编程中见得非常多。作为数据存储和数据传输的载体，运行在JavaBean模式下的对象，在众多的编程层次都会被用到，并且根据作用不同被冠以各种不同的名称，如

- ❑ PO（Persistent Object）——持久化对象
- ❑ BO（Business Object）——业务对象
- ❑ VO（Value Object）——值对象
- ❑ DTO（Data Transfer Object）——数据传输对象
- ❑ FormBean——页面对象

对于这些纷繁复杂的缩写和对象类别，许多初学者会感到非常头疼。它们从形式上看是一系列难记的缩写，不过真正让程序员头疼的，不仅在于它们被用于不同的业务场景和编程层次，还在于它们在某些时候甚至只是同一个对象在不同层次上的不同名称。

不过我们大可不必在对象的名称和叫法上过分纠结。因为对于程序员来说，无论这些对象被冠以什么花里胡哨的名称，**它们只不过是对基本的、运行在JavaBean模式下对象**

的有效扩展或增强。

以 PO（Persistent Object）为例，当我们使用 Hibernate 作为 O/R Mapping 的工具时，一个典型的 PO 会被定义成如代码清单 2-1 所示的样子。

代码清单 2-1　User.java

```
@Entity
@Proxy(lazy = true)
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class User {

   @Id
   @GeneratedValue
   private Integer id;

   @Column
   private String name;

   @Column
   private String password;

   @Column
   private String email;

   // 这里省略了所有的 setter 和 getter 方法
}
```

假设去除那些 Annotation，我们会发现这个 PO 和一个普通的 JavaBean 并无不同，至少我们无法在形式上将它们区分开。因此，我们说 Annotation 在这里的所用是丰富了一个普通 JavaBean 的语义，从而使之成为一个持久化对象。而当我们使用 O/R Mapping 的工具 Hibernate 进行处理时，也是根据这些 Annotation 才能够对这些 PO 进行识别并赋予其相应功能的。也就是说，**JavaBean 自身的特性并没有发生改变，只是引入了一些额外的编程元素从而对 JavaBean 进行了增强。**

当一个对象运作在属性对象模式时，其本质是对象的 JavaBean 特性。我们可以从其表现形式和运行特征中得出下面这样的一个结论。

结论　JavaBean 对象的产生主要是为了强调对象的内在特性和状态，同时构造一个数据存储和数据传输的载体。

我们在本节开篇所提到的各种不同的对象名称的定义，它们之间的界定实际上是非常模糊的。同样一个对象，往往可以兼任多种不同的角色，在不同的编程层次表现为不同的对象实体，其最终目的是在特定的场合表现出其作用。

在上面的结论中，我们还读到另外一层意思，那就是 JavaBean 对象的一个重要的性质在于它是一个数据存储和数据传输的载体。有关这一点，我们将在之后的章节进行分析。

2.1.1.3 行为对象模式

根据之前有关对象构成的分析，运行在**行为对象模式**之下的对象，我们往往称之为**无状态对象**。在上一节中，我们提到对象的内在特性和状态是由构成对象的属性来表达的。所以，所谓的无状态对象实际上就是**指对象的方法所表达的行为特性并不依赖于对象的内部属性的状态**。而这种无状态的特性，非常适合进行一个请求的响应，并在方法体的内部实现中进行复杂的业务逻辑处理。

我们在这里可以针对对象的行为方法的语法做进一步的分析，如图 2-2 所示。

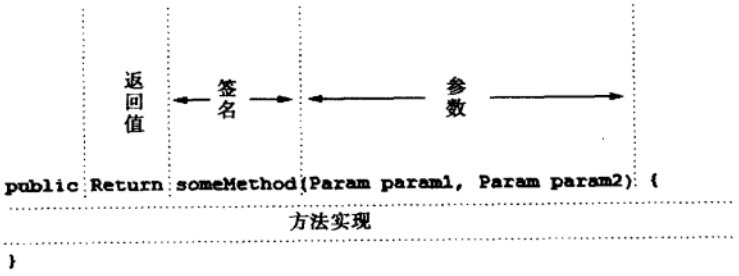


图 2-2 对象的行为方法的语法分析

从图 2-2 中，我们看到对象方法的语法定义同样体现出一定的规律性：

- **方法签名**（Signature）——行为动作的逻辑语义概括
- **参数**（Parameter）——行为动作的逻辑请求输入
- **返回值**（Return）——行为动作的处理响应结果输出

我们可以看到，方法的定义实际上是一种**触发式**的逻辑定义。当需要完成某种行为动作（业务逻辑）时，我们会将请求作为参数输入到方法的参数列表中，而返回值则自然而然地成为业务逻辑的处理结果。由此，我们可以得出以下结论：

结论 *对象中的方法定义是进行请求响应的天然载体。*

这个结论我们将在之后对各种 Web 框架对 Http 请求的响应设计中再次提及。因为在 Java 标准中，对 Http 请求的响应是通过 Servlet 标准来实现的。而我们知道，Servlet 对象就是一个非常典型的运行在行为对象模式之上的无状态对象。

上述结论对读者理解“请求 - 响应”的过程在编程语言中的逻辑表达有很大的帮助。因而读者应仔细体会这些构成要素在“请求 - 响应”过程中所起的作用，从而对 Web 框

架的设计有更深的感悟。

2.1.2　对象关系模型

对象的构成模型是从对象内部结构的角度对面向对象编程中的基本元素进行的分析。在本节中，我们分析的角度将由“内”转向“外”，考虑对象与对象之间的关系。

谈到对象之间的关系，我们很容易想到两个不同的层次：

- **从属关系**——一个对象在逻辑语义上隶属于另外一个对象
- **协作关系**——对象之间通过协作来共同表达一个逻辑语义

这两种关系在面向对象编程语言中分别拥有不同的表现形式和逻辑意义，这二者构成了绝大多数的对象关系模型。接下来，我们就来分别分析这两种对象关系模型。

2.1.2.1　对象的从属关系

对象的从属关系，主要指**一个对象在逻辑语义上隶属于另外一个对象**。这个定义实际上依然非常抽象。要理解这一定义，我们就必须从“隶属”这个核心词汇入手。逻辑语义上的“隶属”，主要有两种不同的含义。

- **归属**

归属的逻辑含义很直观。比如说，一个人总是归属于一个国家；一本书总是有作者。因而，当我们把人和国家、书和作者都映射成面向对象编程语言中所定义的一个个对象时，它们之间自然而然就形成了归属关系。**这种归属关系是由外部世界的逻辑关系映射到编程元素之上而带来的。**

- **继承**

继承的逻辑含义就有点晦涩。比如说，马、白马和千里马之间的关系。首先，白马和千里马都是马的一种，然而白马和千里马却各自拥有自己独有的特性：白马是白色的、千里马一日可行千里。此时，我们可以说白马和千里马都属于马，它们都继承了马的基本特征，却又各自扩展了自身独有的特质。

明确了“隶属”的两层含义，我们就需要把它们和面向对象编程语言联系在一起。归属和继承，它们在面向对象的编程语言中又以怎样的形式表现出来呢？

我们先来看看“归属”这层含义。对于“归属”关系的编程语言表现形式，我们可以得出下面的结论：

结论　*“归属”关系在面向对象编程语言中，主要以对象之间互相引用的形式存在。*

我们以书和作者之间的对象定义作为例子来说明，其相关源码如代码清单 2-2 所示。

对象的协作关系在对象运行在行为对象模式时显得尤为突出。因为当使用一个具体的方法来进行动作响应时，我们总是会借助一些辅助对象的操作来帮助我们共同完成动作的具体逻辑。也就是说，我们会将一个动作从业务上进行逻辑划分，将不同的业务分派到不同的对象之上去执行。这就成为我们所熟知的分层开发模式的理论依据。

2.1.3 面向对象编程的基本观点

在了解了对象的构成模型和对象的关系模型之后，读者不免要问，这些内容和我们的日常编程有关系吗？答案是有关系！而且不仅是有关系，还是有相当大的关系！在本节中，我们就以结论辅之以分析的方法，为读者展示面向对象编程中的一些基本观点。

结论 *每一种对象的构成模型，都有其特定的应用范围。*

根据之前我们有关对象的构成模型的分析，可以发现三种对象的构成模型在日常的编程过程中都曾经碰到过。因此，我们应该首先明确的观点是**每一种对象的构成模型都有其存在的合理性**，并没有任何一种模型是错误的模型这一说。

既然如此，我们所要做的就是认清这些对象构成模式的特性，并且能够在最恰当的业务场景中选择最合适的模型加以应用。那么，从面向对象思想的角度，如果我们将这些对象运作模式做一个纵向的比较，它们有没有优劣之分呢？

结论 *将对象运作在“属性－行为”模式上，最符合面向对象编程思想的本意。*

这一结论承接了上一个结论，可以说是对象建模方式的一种合理的理解和扩展，也回答了我们刚才的问题。当我们进行对象建模的时候，总是首先需要根据业务情况选择一个对象建模设计的角度，而这个角度往往取决于对象在整个程序中所起的作用。例如，当我们需要进行数据传输或者数据映射时，我们应该基于对象的“属性模式”来进行对象建模；当我们需要进行动作响应时，我们应该基于对象的“行为模式”来进行对象建模。

然而，运行在“属性模式”中的对象并不是说完全就不能具备行为动作。基于某一种模式进行建模，只是我们考虑对象设计的角度不同。如果我们站在一个“对象构成的完整性”这样一个高度来看待每一个对象，它们总是由属性和方法共同构成。因此，在任何一种对象的构成模式上走极端都是不符合面向对象编程思想本意的。

软件大师 Martin Fowler 就曾经撰文指出，在对象建模时不应极端地将对象设计成单一的“属性模式”。读者可以参考：http://www.martinfowler.com/bliki/AnemicDomainModel.html 获得全文描述。

The fundamental horror of this anti-pattern is that it's so contrary to the basic idea

```
}

public class ThousandMileHouse extends Horse {

}
```

白马和马之间，我们使用了 Java 中的关键字 extends 来表达前者继承自后者。这种方式与我们之前所看到的对象之间的引用模式完全不同，**它使用了编程语言中的原生语法支持**。

这种对象关系的表达方式非常简单而有效，不过当我们引入一个语法，就不得不遵循这个语法所规定的编程规范所带来的编程限制。例如在 Java 中，我们就必须遵循单根继承的规范。这一点也是“继承”这种方式经常被诟病的原因之一。

在 Java 中，除了 extends 关键字以外，还有 implements 关键字来表达一个实现类与接口类之间的关系。实际上这是一种特殊的“继承”关系，它无论从语义上还是表现形式上，与 extends 都基本相同。

2.1.2.2　对象的协作关系

对象的从属关系从现实世界逻辑语义的角度描述了对象与对象之间的关系。从之前的分析中，我们可以发现无论是“归属”关系还是“继承”关系，它们都在围绕着对象构成要素中的属性做文章。那么读者不禁要问，围绕着对象的行为动作特征，对象之间是否能够建立起关系模型呢？

从哲学的观点来看，万事万物都存在着普遍而必然的联系。从对象的行为特性上分析，一个对象的行为特征总是能够与另外一个对象的行为特征形成依赖关系。而这种依赖关系，在极大程度上影响着对象的逻辑行为模式。例如，一个人“行走”这样一个动作，需要手脚的共同配合才能完成，具体来说就是“摆手”和“抬脚”。而当我们把手和脚分别看作一个对象时，“摆”和“抬”就成为手和脚的行为动作了。

这样一说，似乎对象之间的协作关系就非常容易理解了，请看以下结论：

结论　*当对象的行为动作需要其他对象的行为动作进行配合时，对象之间就形成了协作关系。*

可以想象，**一个对象在绝大多数情况下都不是孤立存在的，它总是需要通过与其他对象的协作来完成其自身的业务逻辑**。这是软件大师 Martin Fowler 曾经提过的一个重要观点。然而这却为我们的编程带来了一些潜在的问题：如何来管理对象和协作对象之间的关系呢？有关这一问题，我们将在第 5 章详细进行讲解。

代码清单 2-2 Book.java

```
public class Book {

   private String name;

   private List<Author> authors;

}
```

我们在这里所表达的是书和作者之间的“归属”关系。从代码中，可以很明显看到，一本书可能有多个作者，所以我们在书（Book）的属性定义中加入了一个 List 的容器结构，List 中的对象类型是作者（Author）。这样一来，书和作者之间就形成了一个**引用关系**。

使用对象之间的引用来表达“归属”关系，是一种非常廉价的做法。因为这种关系的表达来源于对象定义的基本模式，不会对对象自身产生破坏性影响。

细心的读者还会发现，我们这里所说的“归属”关系，实际上还蕴含了一层“数量”的对应关系。在上面的例子中，我们发现书和作者的数量关系是“一对多”。除了“一对多”以外，对象之间的归属关系在数量这个维度上还可以是“一对一”和“多对多”。有趣的是，这三种归属关系正好也和我们关系型数据库模型中所定义的基本关系一一对应。这种关系模型也就为我们在 Java 世界和数据库世界之间进行 O/R Mapping 打下了理论基础。

看完了“归属”关系，我们再来看看“继承”关系。有关“继承”关系的编程形式表述，我们可以用下述结论来进行说明：

结论 *“继承”关系在面向对象编程语言中，主要以原生语法的形式获得支持。*

什么是“以原生语法的形式获得支持”呢？我们来看看之前说的那个白马的例子，其相关源码如代码清单 2-3 所示。

代码清单 2-3 Horse.java

```
public class Horse {

   public void run() {

   }

}

public class WhiteHorse extends Horse {
```

of object-oriented design; which is to combine data and process together.

有关这一点，也引起了许多国内软件开发人员的深入讨论，并且引申出许多极具特色的名词，诸如："贫血模型"、"失血模型"、"充血模型"、"胀血模型"等等。这些讨论非常有价值，对于对象建模有兴趣的读者可以使用搜索引擎就相关的讨论进行搜索。

既然存在着那么多有关"领域模型"的理解方式，为什么 Martin Fowler 这样的软件大师还是推荐尽可能使对象运行在"属性－行为"模式之上呢？除了它自身在构成形式上比较完整，能够比其他两种运行方式更容易表达对象的逻辑语义之外，还有什么别的特殊考虑吗？笔者通过思考和分析，给出可能的两个理由：

❑ **当对象运作在"属性－行为"模式上时，我们能够最大程度地应用各种设计模式**

对于设计模式有深入研究的读者，应该会同意这个观点。设计模式的存在基础是对象，因而设计模式自身的分类也围绕着对象展开。我们可以发现，绝大多数的设计模式需要通过类、接口、属性、方法这些语法元素的共同配合才能完成。因而，单一的属性模式和行为模式的对象，在设计模式的应用上很难施展拳脚。

❑ **当对象运作在"属性－行为"模式上时，我们能够最大程度地发挥对象之间的协作能力**

仔细分析对象的关系模型，我们会发现无论是对象的从属关系还是对象的协作关系，它们在绝大多数情况下是通过对象之间的属性引用来完成的。这种属性引用的方式，只是在形式上解决了对象和对象之间进行关联的问题。而真正谈到对象之间的配合，则不可避免地需要通过行为动作的逻辑调用来完成，这也是对象协作的本质内容。

对象建模是一个很深刻的哲学问题，它将直接影响我们的编程模式。所以对于建模这个问题，大家应该综合各家之言，并形成自己的观点。笔者在这里的观点是：**对象建模方式首先是一个哲学问题，取决于设计者本身对于业务和技术的综合考虑。任何一种建模方式都不是绝对正确或者绝对错误的方式。我们所期待看到的对象建模的结果是提高程序的"可读性"、"可维护性"和"可扩展性"。一切建模方式都应该首先服务于这一基本的程序开发的最佳实践。**

结论　建立对象之间的关系模型是面向对象编程的核心内容。

对象建模是一个很复杂的逻辑抽象过程。事实上，对象建模最难的地方并不在于设计某一个单体对象的属性构成或者方法构成。因为之前我们也提到，对象总不能以单体的形式孤立存在。对象与对象之间总是以某种方式相互关联、相互配合。这种关联要么形成对象之间的从属关系，要么通过对象的行为方法进行互相协作。

由此可见，我们在进行对象建模的时候，必须优先考虑的就是对象与对象之间的关系

模型，关系模型决定我们进行对象关联的具体形式，选择合适的编程语言语法进行关联关系的表达。

将对象之间的协作和关联关系作为设计对象的最重要的考虑因素，可以时刻提醒我们不要将过多的逻辑放在一个对象之中。因为当考虑到对象之间的协作和关联关系，我们就可以充分挖掘每一个对象的职责和语义，从而避免一个对象过于复杂而变得不可维护。

2.2 框架的本质

什么是框架？框架从何而来？为什么要使用框架？这是一系列简单而又复杂的问题。简单，是因为它们本身似乎不应该成为问题。框架实实在在存在，并且在开发中发挥着重要的作用，我们的日常工作，遵循着框架所规定的编程模式，在其指导之下，我们能够编写更为强大的程序。说其复杂，是因为框架本身又是如此纷繁复杂，我们在使用框架的同时，往往会迷失其中。

任何事物都有蕴含在其内部的本质。无论框架本身有多复杂，我们所需要探寻的，都是其最为内在的东西。框架为什么会产生？我们来看一个最最简单的例子。

在 Java 中，如果要判定一个输入是否为 null 或空字符串，我们会使用下面的代码：

```
if(str == null || str.length() == 0) {
    // 在这里添加你的逻辑
}
```

这段代码非常普通，简单学习过 Java 语法的程序员都能够读懂并编写。那么这段代码是如何运作的呢？我们所编写的 Java 程序，首先获得的是来自于 Java 的基本支持：语法支持与基本功能的 API 级别的支持（str.length() 方法实际上就是 JDK 所提供的字符串的基本 API）。换句话说，我们编写的所有程序，都依赖于一个最最基本的前提条件：**JDK 所提供的 API 支持**。

当一个需求被重复 1000 次，那么我们就需要重复 1000 次针对需求的解决办法，这是一个显而易见的道理。然而当上面的代码片段散落在我们的程序中 1000 次，我们不免会思考，是不是有什么简单有效的途径可以把事情做得更加漂亮一些呢？我们可以针对代码片段做一次简单的逻辑抽取重构，如代码清单 2-4 所示。

代码清单 2-4 StringUtils.java

```
// 定义一个类和一个静态工具方法来抽象出将被重复调用的逻辑
public abstract class StringUtils {
    // 封装了一个静态方法
    public static boolean isEmpty(String str) {
        return str == null || str.length() == 0;
```

```
    }
}

// 引用静态方法取代之前的代码片段
if(StringUtils.isEmpty(string)) {
    // 在这里添加你的逻辑
}
```

在上面的代码段中，我们定义了一个静态方法，将之前写的那段逻辑封装起来。这一层小小的封装虽然看上去是一个“换汤不换药”的做法，但是从深远意义上来说，我们至少可以从以下两个方面获得好处：

❑ 可读性

静态方法的签名从一个角度向我们揭示了一段逻辑的实际意义。比如在这个例子中，isEmpty 表示“判定某个输入是否为空”。与之前的代码片段相比，如果我们在一个 1000 行的程序代码片段中观察这 2 种不同的代码形式，那么前者往往会被你无视，它完全无法引起你的思维停顿，而后者却能够显而易见地在逻辑上给你足够且直观的提示。

❑ 可扩展性

如果我们对上述需求稍作改动，程序同时需要对输入为空格的字符串做出同样的判定。我们同样将上述的需求应用 1000 次，那么前者将导致我们在整个应用中进行搜索并替换修改 1000 次，而后者只需要针对我们封装的逻辑修改 1 次即可。

从上面的例子我们可以看出，虽然仅仅对代码做了一次简单的重构，却在上述的两个方面为我们解决了潜在的问题。这一现象或许直到现在你才意识到，但很多程序员前辈在很早以前就意识到了。因而，早就有人为此编写了类似的代码。比如说，类似的方法就存在于 Apache 的 commons-lang 的 JAR 包中，如代码清单 2-5 所示。

代码清单 2-5　StringUtils.java

```
package org.apache.commons.lang;

public class StringUtils {

// 这里省略了许多其他的代码

public static boolean isEmpty(String str) {
    return str == null || str.length() == 0;
}

}
```

当我们将 Apache 的 commons-lang 的 JAR 包加到 CLASSPATH 中时，就能在程序的

任何地方“免费地”使用上述方法。也就是说，我们自己无须自行编写代码对 JDK 进行扩展，因为 Apache 的 commons-lang 已经为我们做了。既然如此，我们唯一所需要做的，只是把别人做的东西加到 CLASSPATH 中并且使用它而已。

这是一个很熟悉的过程，不是吗？我们在搭建程序运行的基本环境时，指定程序所依赖的 JAR 文件是其中的一个重要步骤。而这一步骤，实际上包含了 Java 开发中最最基本而浅显的道理：

结论 *当我们加载一个 JAR 包到 CLASSPATH 时，实际上是获得了 JAR 中所有对 JDK 的额外支持。*

我们的程序就像一个金字塔形状。位于最底部的当然是 JVM，提供运行 Java 程序的基础环境，包括对整个 Java 程序的编译运行。在这个之上的是 JDK，JDK 是构建在 JVM 之上的基本的对象行为的定义（我们在搭建开发环境时所安装的 JDK 就是这个）。而再往上，是一个具备层次结构的 JAR 层，所有被加载到 CLASSPATH 中的 JAR 文件都搭建在 JDK 层次之上，它们之间可能形成互相依赖，但不管怎么说，**它们的作用都是提供 JDK 以外的功能支持**。最后，在金字塔尖的，才是我们日常编写的应用程序，它将依赖于金字塔低端的所有程序。这样一个结构如图 2-3 所示。

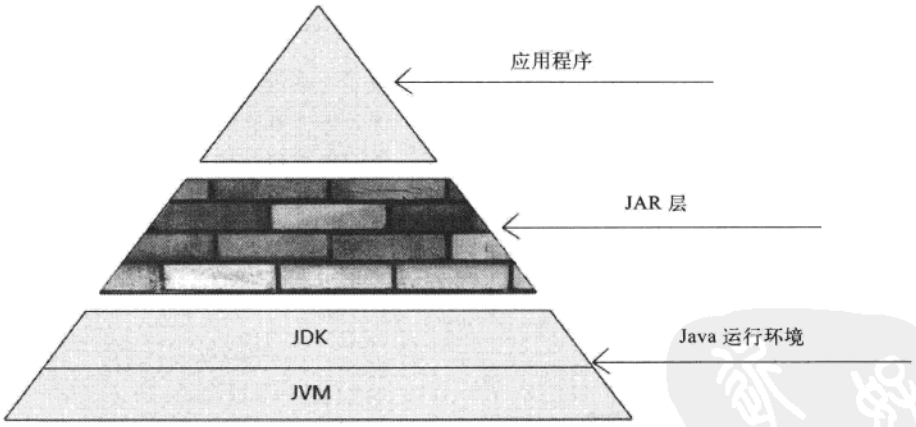


图 2-3 Java 应用的金字塔结构

仔细观察一下处于中间的 JAR 层，这个层次的组成结构与其他的层次不同。它是由一块块砖头堆砌而成，上层的砖块搭建在下层的砖块之上。如果我们把其中的每一块砖都比作一个 JAR 文件，它们之间也就形成了明显的具备层次的依赖关系。

这个层次中的任何 JAR 文件本身可能并不为最终的程序提供具体的功能实现，

但它却为我们编写程序提供了必要的支持。如果查看一个标准的 J2EE 程序运行时所依赖的 CLASSPATH 中的 JAR 包，会发现我们所熟悉的那些“框架”，实际上都蕴涵其中。我们在这里给出一个最简单的示例程序在 Eclipse 中的 CLASSPATH 截图，如图 2-4 所示。

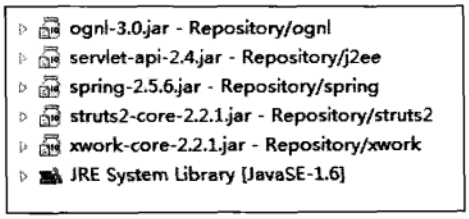


图 2-4　Eclipse 中的 CLASSPATH 示例

从图中我们看到，JRE System Library 是整个应用程序最基本的运行环境。而无论是 Struts2 还是 Spring，它们都以 JAR 文件的形式被加载到程序运行所依赖的 CLASSPATH 中，并为我们的应用程序使用。如果我们用更加通俗的话来表述这一现象，则是：

结论　*框架只是一个 JAR 包而已，其本质是对 JDK 的功能扩展。*

当我们说一个程序使用了 Spring 框架，隐藏在背后的潜台词实际上是说，我们把 Spring 的分发包加入到 CLASSPATH，并且在程序中使用了其功能。框架，其实就是这么回事！就是如此简单！

到现在为止，框架似乎还没有任何在我们的知识范畴以外的东西，它们的本质是如此一致，以至于我们很容易遗忘把一个 JAR 文件加入到 CLASSPATH 中的初衷：**解决在某个领域的开发中所碰到的困境**。正如我们在一开始使用的那个例子一样，框架作为一个 JAR 包，实际上是许许多多解决各种问题的类和方法的集合。当然，更多时候，它们包含了编写这些 JAR 包的作者所创造的许多最佳实践。

结论　*框架是一组程序的集合，包含了一系列的最佳实践，作用是解决某个领域的问题。*

只有解决问题才是所有框架的共同目标。框架的产生就是为了解决一个又一个在开发中所遇到的困境。不同的框架，只是为了解决不同领域的问题。所以，对于广大程序员来说，**千万不要为了学习框架而学习框架，而是要为了解决问题而学习框架，这才是一个程序员的正确学习之道**。

2.3 最佳实践

一切程序的编写，都需要遵循特定的**规范**。这里所说的规范，往往是建立在运行环境之上的一系列概念和实现方法的基本定义，并被归纳为一个完整的体系。例如，我们使用Java来进行Web开发，所需要遵循的最基本的规范就是我们所熟悉的Servlet标准、JSP标准，等等。

建立在标准和规范之上的，是各种针对这些标准和规范的实现。这些实现构成了**程序运行的基本环境**。例如，Tomcat有对Servlet标准的实现方式，而Websphere则有不同的实现方式。然而它们在本质上都实现了Servlet标准所规定的接口，从而让我们的应用程序可以透明地使用这些API，而无须关心真正的Web容器内部的实现机理。

我们所编写的程序，总是建立在一系列的**规范和基本运行环境之上**。面对纷繁复杂的业务需求，不同的程序员可以按照自己的意愿来编写程序，因此，即使为了表达相同的业务功能，不同的程序代码之间的差异性也是很大的。程序的差异性有时候会给人以创新的灵感，但是更多的时候会造成麻烦。因为差异性越大，维护起来就越麻烦。出于对可维护性和可读性的要求，我们所希望的程序最好能从宏观层面上看上去是一致的，使得每一个程序员都能够读懂并合理运用，这才是我们的目标。这一目标，我们习惯上称之为**最佳实践**。

结论 *最佳实践（Best Practice），实际上是无数程序员在经过了无数次的尝试后，总结出来的处理特定问题的特定方法。如果我们把每个程序员的自由发挥看作是一条通往成功的路径，最佳实践就是其中的最短路径，它能够极大地解放生产力。*

所有这些最佳实践，最终又以一个个JAR包的形式蕴含在框架之中，对我们的应用程序提供必要的支持，因此我们有必要在这里探寻一些最最基本的最佳实践，从更深的层次了解框架存在的意义和框架的设计初衷。在之后的章节中，我们会反复提及这些最佳实践，因为它们不仅能够指导我们进行程序开发，它们本身也蕴含在Struts2的内部。

最佳实践 *永远不要生搬硬套任何最佳实践，真理之锁永远只为最合适的那把钥匙开启。*

这是一条凌驾于任何最佳实践之上的最佳实践。在使用框架编写程序时，程序员最容易犯的毛病就是对某项技术或者某个框架绝对迷信，并将它生搬硬套于任何程序开发之中。应用程序永远服务于具体的业务场景，对于不同的业务需求，我们的解决方案也会有所区别，自然也会涉及不同的框架选择。

在实际开发中，我们遇到的许多编程开发的问题都是没有固定答案的哲学取向问题。所以，往往没有“最好”的答案，只有“最合适”的答案。这是在面对多种解决方案进行

取舍时的一个基本准绳。

最佳实践　始终保证程序的可读性、可维护性和可扩展性。

可读性、可维护性和可扩展性，就像三脚架的三个支撑脚，缺一不可。任何对程序的重构，实际上都围绕着这三个基本原则进行，而它们也同时成为衡量程序写得好坏的最基本标准。代码的不断重构、框架的产生实际上都来自于这三个程序内在属性的驱动。

我们之前已经反复提到了程序的可维护性和可扩展性。事实上，程序的可读性也是程序所应具备的必不可少的基本属性，失去了可读性的程序，其可维护性和可扩展性也就无从谈起了。这三大原则从方法论的角度规定了一切最佳实践都不能违背这三大程序的基本属性。否则，我们迟早会为一些蝇头小利而舍弃程序开发的源头根本。当一个程序失去了可读性、可维护性和可扩展性，它也就失去了生命力。

最佳实践　简单是美（Simple is Beauty）。

简单是美是一种指导思想，它其实包含两个层次的意思。第一层意思是**消除重复**（Don't repeat yourself），这是一个显而易见的代码重构标准。第二层意思则是要求我们**化繁入简**（Heavy to Light），用尽量少的代码语言来表达尽量多的逻辑意义。

简单是美，将最佳实践的要求细化到了方法论的层面。然而，无论我们的程序如何简单，都应该始终记得，**简单但必须可读，简单但必须可扩展**。切忌为了一些细节，而忘记更大的原则。

最佳实践　尽可能使用面向对象的观点进行编程。

我们可以看到，这个层面的最佳实践，已经从基本准则和指导思想转向了具体的编程层面。虽然面向对象自身也只是一种编程的指导思想，然而它却是和程序设计与实现息息相关并且对程序编写影响最大的一条编程准则。

面向对象这个概念本身就是一个非常耐人寻味的问题。要讨论面向对象的概念、设计和方法论，恐怕一天一夜都讲不完。在本章之初，我们从“对象”这个概念入手，通过对“对象”内部结构的分析，试图向读者展示面向对象编程中的一些重要理论。读者对这些理论不应停留在死记硬背的层面，而是要将它们融入到框架的设计理念中去理解。同时，这些理论也将成为我们判别框架和设计方案优劣的重要标准。

最佳实践　减少依赖（消除耦合）。

之前在分析框架的本质时已经提到，任何 Java 程序总是依赖于其运行环境（JVM 层）

和支持应用程序的 JAR 层。**加入到 CLASSPATH 中 JAR 越多，就意味着程序对外部环境的依赖度越高，对外部环境的依赖度越高，就意味着程序本身越难以脱离特定的外部环境进行单元测试**。因此，减少甚至消除依赖，也成为许多框架所追求的目标。

Struts2 在这一点上做得尤为成功。Struts2 不但实现了 Web 框架与 Web 容器之间的解耦合，还在此基础之上实现了各个编程元素之间的有效沟通。在之后的章节中，我们会深入探究 Struts2 在这条最佳实践上所做出的努力。

2.4 Web 开发的基本模式

到此为止，我们花了大量的篇幅介绍了许许多多与 Web 开发完全无关的东西。无论是面向对象的概念、框架的本质内容还是我们开发中应当遵循的最佳实践，它们都是程序员需要培养的内在修养。接下来，我们将话题真正转入 Web 开发，来看看在 Web 开发中应该遵循什么样的最佳实践。

2.4.1 分层开发模式

之前我们讨论了 Web 开发中几条基本的最佳实践，它们会成为贯穿本书始终的指导思想。明确了指导思想，我们有必要从方法论的角度来探讨一下 Web 开发的一些基本模式。

从宏观上来说，Web 开发模式中最最重要的一条是**分层开发模式**。分层开发模式是指，在开发 J2EE 程序时，将整个程序根据功能职责进行纵向划分。一个比较典型并为大家所熟知的划分方法是将整个程序分为：表示层、业务层和持久层，如图 2-5 所示。

图 2-5 分层开发模式示意图

不同的层次，实际上承担了不同的功能职责：

- **表示层**（Presentation Layer）——负责处理与界面交互相关的功能
- **业务层**（Business Layer）——负责复杂的业务逻辑计算和判断
- **持久层**（Persistent Layer）——负责将业务逻辑数据进行持久化存储

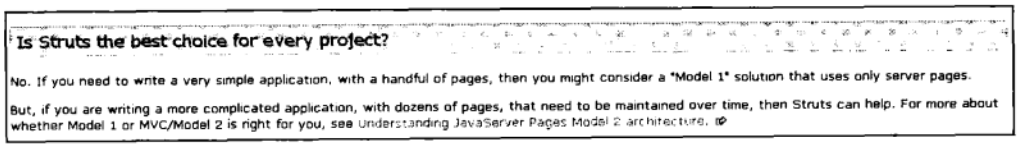
分层开发模式是技术层面的“分而治之”设计思想的一种体现。而蕴含在其内部的驱动力还是我们反复强调的：程序的可读性和可扩展性。出于可读性考虑，把不同功能职责

的代码分开，能够使程序流程更加清晰明了；出于可扩展性考虑，也只有把相类似的功能归结为一个纵向层次，才使得我们在这个层次之上研究通用的解决方案成为可能。

分层开发模式，从逻辑上是将开发职责分派到不同的对象之上去执行的一种设计思想。回顾我们在面向对象浅谈的章节中所提到的对象协作关系，也正是分层开发模式的理论依据。

既然是职责分派，我们就不得不分清什么是职责、什么样的对象适合什么样的职责。如此一来，有关分层开发模式的讨论就变成了一个典型的哲学问题。凡是哲学问题，都会出现正反两派。分层开发模式所涉及的争论主题主要包括两个方面：第一，分层开发到底有无必要？第二，对于一个 J2EE 程序到底分为多少层进行开发比较合适？

我们先来探讨第一个问题：对一个程序实施分层开发到底有无必要？分层开发模式，为程序的可扩展性提供了可能性，但是当问题的作用域变小时，分层开发模式反而成为一种累赘。有许多程序员会抱怨，一个简单的逻辑功能，动辄就要十几个文件配合上百行代码来完成。此时，我们不禁要问：我们的开发真的需要分层吗？分层开发到底为我们带来了多少好处呢？针对这一问题，我们不妨来看看 Struts2 的一个官方的 FAQ，如图 2-6 所示。


Is Struts the best choice for every project?

No. If you need to write a very simple application, with a handful of pages, then you might consider a "Model 1" solution that uses only server pages.

But, if you are writing a more complicated application, with dozens of pages, that need to be maintained over time, then Struts can help. For more about whether Model 1 or MVC/Model 2 is right for you, see Understanding JavaServer Pages Model 2 architecture.

图 2-6　Struts 的 FAQ

非常明显，当问题的作用域发生变化时，解决问题的方法也要相应做出改变。所以，**分层开发模式，对于大型企业应用或者产品级的应用程序开发是有着重要意义的；然而当一个应用程序足够小，并且需求的变更处于可控的范围之内时，我们对于分层开发模式的选择应该谨慎**。这就是所谓的“杀鸡焉用牛刀”。

我们再来看看第二个问题：对于一个 J2EE 程序，到底要分多少层进行开发比较合适？这是一个与整个应用程序构架相关的话题。有许多程序员赞同分层开发模式，不过他们都希望将层次分得尽量简单，崇尚“简单是美”的原则。对于这一问题，实际上也没有绝对正确的答案。因为**一切脱离了业务实际的架构设计都是虚幻的**。我们只能在实践中不断总结，并将前人的许多经验作为我们进行开发层次划分的重要依据，选择适合于实际业务需求的开发层次，才是程序开发的最佳实践。

这些有关分层开发的哲学问题的讨论，每个程序员都有自己的见解。然而从框架的角度，我们也能看出一些框架的设计者对于某个开发层次的理解，因为**我们最最熟悉的这些著名的框架，实际上就是为了应对各个开发层次的编程问题而设计的解决方案**。比如说：

Struts2 是表示层的框架；Spring 是业务层的框架；Hibernate 是持久层的框架。

在本书中，我们所有讨论的重点实际上是围绕着表示层的解决方案 —— Struts2 进行的。笔者花了那么多笔墨，才把 Struts2 这位主人公引出来的目的，是希望读者能够站在全局的高度来审视 Struts2，也只有这样，才能够真正学好每一个开源框架。

2.4.2 MVC 模式

在分层开发模式的前提下，每一个层次都可以单独研究，并寻找合适的解决方案和最佳实践。对于表示层，有一种称之为 MVC **模式**的最佳实践被广泛使用，并在此基础上创建了许多基于这种模式的开发框架。

MVC 模式实际上是众多经典的 Java 开发模式中的一种。它的基本原理是通过元素分解，来处理基于“请求 - 响应”模式的程序中的各种问题。

- M（Model）——数据模型
- V（View）——视图展现
- C（Control）——控制器

任何一个 B/S 应用，其本质实际上是一个“请求 - 响应”的处理过程的集合体。那么 MVC 模式是如何被提炼出来并成为一个模式的呢？ 我们来模拟一个“请求 - 响应”的过程，如图 2-7 所示。

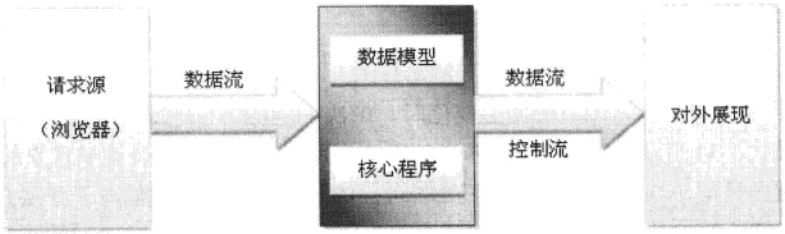


图 2-7 请求 - 响应模式

在整个请求 - 响应过程中，有哪些元素是必不可少的呢？

❑ 数据模型

在图中，就是顺着箭头方向进行传输的数据，它们是程序的核心载体，也是贯穿程序运行的核心内容。

❑ 对外交互

在图中，对外交互表现为一个“头”和一个“尾”。“头”指的是请求发起的地方，没有请求，一切之后的所有内容都无从谈起。“尾”指的是逻辑执行完成后，对外展现出来

的执行结果。在传统意义上，我们利用 HTML 扩展的技术（如 JSP 等）来实现对外交互，在展现结果时，我们还需要完成一定的展现逻辑，比如错误展示、分支判断，等等。

❑ **程序的执行和控制**

实际上它不仅是接受请求数据的场所，也是处理请求的场所。在请求处理完毕之后，它还要负责响应跳转。这个部分可能会存在着不同的表现形式。以前，我们用 JSP 和 Servlet，后来用 Struts1 或者 Struts2 的 Action。而这一变化，实际上包含了我们不断对程序进行重构的过程。

上面这 3 大元素，在不同的年代被赋予了不同的表现形式。例如，在很久以前，我们使用 Servlet 或者 JSP 来编写程序跳转的控制过程，有了 Struts1.X 后，我们使用框架所定义的 Action 类来处理。这些不同的表现形式有的受到时代的束缚，表现形式非常落后，有的甚至已经不再使用。但是我们忽略这些外在的表现形式就可以发现，这不就是我们已经熟悉的 MVC 吗？

❑ **数据模型**——Model

❑ **对外交互**——View

❑ **程序的执行和控制**——Control

MVC 的概念就这么简单，这些概念其实早已深入我们的内心，而我们所缺乏的是将其本质挖掘出来的能力。我们来看看如图 2-8 所示的这幅流行了很多年的讲述 MVC 模型的图。

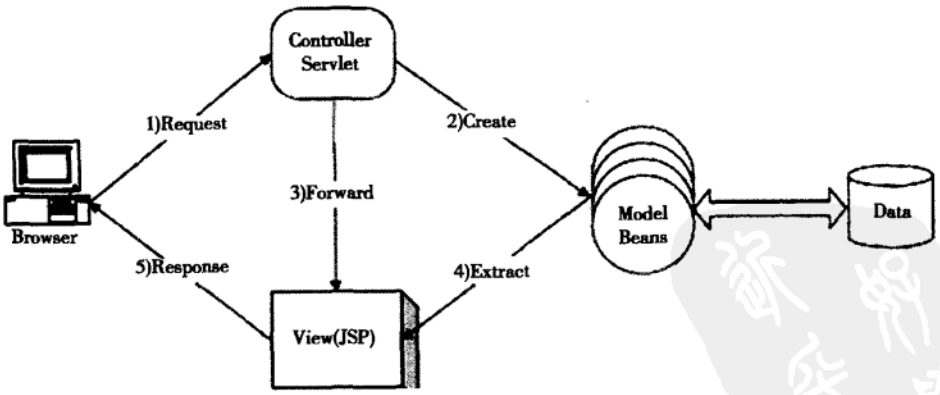


图 2-8　MVC 模型图

在这幅图中，MVC 三个框框各司其职，结构清晰明朗。这也成为我们进行编程开发的最强有力的理论武器，我们需要做的，只是为这些框框赋予不同的表现形式。实际上，框架就是这么干的！而框架的高明之处，仅仅在于它不仅赋予这些元素正确而

恰当的表现形式，同时解决了当元素运行起来时所碰到的各种问题。因此，我们始终应该做到：**程序时时有，概念心中留**。只要MVC的理念在你心中，无论程序怎么变，都能做到万变不离其宗。

2.5 表示层的困惑

当表示层有了MVC模式，程序开发就会变得有章可循。至少，我们不会像无头苍蝇一样无从入手。MVC模式很直观地规定了表示层的各种元素，只要能够通过恰当的程序表现形式来实现这些元素，我们实际上已经在履行最佳实践了。

至此，我们不妨返璞归真，忘记所谓的框架，用最简单的方式来实现一个简单的MVC雏形。在这个过程中，我们不妨回到框架的本质问题上，思考一下究竟一个框架为表示层解决了什么样的编程难题，难道框架只是实现MVC这三大元素那么简单而已？

我们选择Registration（注册）作为业务场景。首先，我们需要一个JSP页面来呈现用户注册的各个字段、一个User类来表示用户实体以及一个RegistrationServlet类来处理注册请求。相关实现源码如代码清单2-6、代码清单2-7和代码清单2-8所示。

代码清单2-6 registration.jsp

```
<form method="post" action="/struts2_example/registration">
   user name: <input type="text" name="user.name" value="downpour" />
   birthday: <input type="text" name="user.birthday" value="1982-04-15" />
   <input type="submit" value="submit" />
</form>
```

代码清单2-7 User.java

```
public class User {

    private String name;

    private Date birthday;

    public User() {

    }

    // 此处省略setter与getter方法
}
```

代码清单 2-8　RegistrationServlet.java

```
public class RegistrationServlet extends HttpServlet {

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse
resp) throws ServletException, IOException {

        // 从 request 获取参数

        String name = req.getParameter("name");
        String birthdayString = req.getParameter("birthday");

        // 做必要的类型转化

        Date birthday = null;
        try {
            birthday = new SmpleDateFormat("yyyy-MM-dd").
parse(birthdayString);
        } catch (ParseException e) {
        e.printStackTrace();
        }

        // 初始化 User 类，并设置字段到 user 对象中去

        User user = new User();
        user.setName(name);
        user.setBirthday(birthday);

        // 调用业务逻辑代码完成注册

        UserService userService = new UserService();
        userService.register(user);

        req.getRequestDispatcher("/success.jsp").forward(req, resp);
    }
}
```

除了上述这 3 段源代码外，我们还需要建立起 JSP 页面中的 form 请求与 Servlet 类的响应之间的关系。这一关系，是在 web.xml 中维护的，如代码清单 2-9 所示。

代码清单 2-9　web.xml

```
<servlet>
  <servlet-name>Register</servlet-name>
  <servlet-class>example.RegistrationServlet</servlet-class>
</servlet>
<servlet-mapping>
```

```
    <servlet-name>Register</servlet-name>
    <url-pattern>/struts2_example/registration</url-pattern>
</servlet-mapping>
```

我们来看看上面的这 4 段代码是如何构成 MVC 的雏形的。

- Model（数据模型）——User.java
- View（对外交互）——registration.jsp
- Control（程序执行和控制）——RegistrationServlet.java
- URL Mapping（请求转化）——web.xml

我们可以看到 MVC 的实现似乎并不复杂。在不借助额外的框架帮助的前提下，只要基本知晓 JSP 和 Servlet 标准（它们是使用 Java 进行 Web 开发的规范和标准），任何程序员都可以像模像样地实现 MVC 模式，因为从原理上讲，MVC 只是一个概念，我们只需要把这个概念中的各个元素赋予相应的程序实现即可。

不过程序终究是一个动态的执行过程。一旦程序开始运行，上面的这些程序实现就会开始遭遇种种困境。这些困境主要来源于两个方面：其一，出于程序自身的可读性和可维护性考虑，需要通过重构来解决程序的复杂性困境。其二，出于业务扩展的需求，需要通过框架级别的功能增强来解决可扩展性困境。

问题 1 *当浏览器发送一个 Http 请求，Web 容器是如何接收这个请求并指定相应的 Java 类来执行业务逻辑并返回处理结果的？*

这个问题是使用 Java 进行 Web 开发的核心问题之一，我们将这个问题简称为 URL Mapping 问题。这个问题的本质实际上来源于 Http 协议与 Java 程序之间的匹配和交互。Web 开发经过了多年的发展，这一核心的哲学问题也经历了多次重大变革，有的崇尚由繁至简，有的则从形式多样化入手。

在上面的例子中，我们可以看到使用 web.xml 来表达 URL Mapping 关系遇到的困境：当系统变大，这种配置上的重复操作会让 web.xml 变得越来越大而难以维护。不仅如此，web.xml 的配置也无法为 URL Mapping 建立起合适的规则引擎。

由此，解决 URL Mapping 问题的核心在于建立一套由 Http 协议中的 URL 表达式到 Java 世界中类对象的**规则匹配引擎**。额外的，这种规则匹配最好比较灵活而简单又不失必要的可维护性。

问题 2 *Web 应用是典型的“请求－响应”模式的应用，数据是如何顺利流转于浏览器和 Java 世界之间的？面对 Http 协议与 Java 世界数据形式的不匹配性，我们如何能够在流转时做到数据类型的自动转化？*

这个问题伴随着问题 1 而来，数据请求与数据返回相当于是基于“请求－响应”模式的 Web 程序的输入和输出。数据的本质是存储于其中的信息，只不过数据在不同的地方有不同的表现形式。例如，在浏览器中，数据总是以字符串形式展现出来，表现出“弱类型”的特征；在 Java 世界，数据则体现为一个个结构化的 Java 对象，表现出“强类型”的特征。于是，就需要有一个工具能够帮助我们解决在数据流转时的数据形式的相互转化。

在上面的例子中，我们可以看到 RegistrationServlet 中，我们编写了额外的代码，把页面上传递过来的日期值转化为 Java 中的 Date 对象。在参数的数量和 Java 对象越来越复杂的情况下，这种额外的代码就会变成一种灾难，甚至成为我们开发的主要瓶颈之一。

解决数据流转问题的方案是使用**表达式引擎**。将表达式引擎插入到程序逻辑执行之前，我们就能从复杂的对象转化中解放出来，从而进一步简化开发流程。

问题 3　*Web 容器是一个典型的多线程环境，针对每个 Http 请求，Web 容器的线程池会分配一个特定的线程进行处理。那么如何保证在多线程环境下，处理请求的 Java 类是线程安全的对象？如何保证数据的流转和访问都是线程安全的？*

这个问题与问题 1 一样，也是 Web 开发中的核心问题之一，因为它涉及 Web 开发中最为底层的处理机制问题。在上面的例子中，我们使用的是基于 Servlet 标准的方式进行编程，扩展 Servlet 用于处理 Http 请求。然而恰恰就是这种编程模型，是一种非线程安全的编程模型，因为 Servlet 对象是一个非线程安全的对象。也就是说，如果我们在 doPost 方法中访问 RegistrationServlet 中所定义的局部变量，就会产生线程安全问题（第 4 章会重点介绍线程安全问题产生的来龙去脉）。

传统的表示层框架对于这个问题的处理方式是采用**规避问题**的方式。既然 Servlet 对象不是一个线程安全的对象，那么我们就干脆禁止在 Servlet 对象的方法中访问 Servlet 对象的内部变量。这种鸵鸟算法固然是一种有效的方案，但它却不是一种合理的方案。最致命的一点是，**它是一种非语法检查级别的禁止，因此也就无法从根本上杜绝程序员犯这样的错误。**

另外一种解决方案就是在整个请求周期中引入 ThreadLocal 模式，通过 ThreadLocal 模式的使用，将整个过程的对象访问都线程安全化，彻底解决多线程环境下的数据访问问题（有关 ThreadLocal 模式的方方面面，我们在后续章节中会详细介绍）。ThreadLocal 模式的引入对于 Web 层框架的影响是深远并且颠覆性的，因为它为框架摆脱 Web 容器的依赖铺平了道路，意味着我们可以通过合理的设计，在脱离 Servlet 等 Web 容器元素的环境中进行编程。

问题 4　*Controller 层作为 MVC 的核心控制器，如何能够在最大程度上支持功能点上的扩展？*

问题 4 来源于我们对程序本身的自然属性（可读性和可扩展性）的需求。这一内在需求实际上也驱动着我们着手在整个 MVC 的构架级别设计更为成熟有效的自扩展方案。

从一个更加宏观的角度来帮助我们理解这个问题，我们来举一个制药工厂生产药品的例子。一个工厂在进行批量生产时，总是会引入“生产线”的概念。生产线能够把整个制药过程划分成若干道工序，当原材料经过每一道工序，最终就会成为一个可出厂销售的药品。某一天，由于市场推广的原因，需要改变药品的包装，那么我们对这条生产线的要求就是它能够改变“包装”这道工序的流程，更改成新的包装。

在上面的例子中，我们可以看到并没有一个“生产线”的概念。这种情况下，我们日后对于逻辑功能的扩展就变得困难重重。虽然我们发现，RegistrationServlet 或许和其他所有的 Servlet 有着非常类似的执行步骤：接收参数、进行类型转换、调用业务逻辑接口执行逻辑、返回处理结果。然而我们却缺乏一条可以任意配置调度的生产线将这个过程规范起来。

解决这个问题从直观上来讲似乎很容易：没有生产线，我们建一条生产线就行了。而事实上，“造轮子”实在是一件费时费力的事情，因为我们要考虑的方面实在太多。这时我们就不得不借鉴许多前辈的经验了，寻找某些事件定义的框架，遵循框架的定义规范来进行编程将是我们解决这个问题的主要途径。

问题 5 *View 层的表现形式总是多种多样的，随着 Web 开发技术的不断发展，MVC 如何在框架级别提供一种完全透明的方式来应对不同的视图表现形式？*

这一问题是基于 View（视图）技术的不断发展，造成传统的基于 HTML 的视图已经不能满足所有的需求而提出的。当今，越来越多新的视图技术被用于 Web 开发中，例如，模板技术、JSON 数据流、Stream 数据流、Flash 展现等等。

在上面的例子中，我们可以看到负责视图层跳转的 RegistrationServlet 是通过硬编码方式完成程序执行跳转的。这种方式不但无法支持多种新的视图技术，同时也无法使我们从复杂的视图跳转的硬编码中释放出来。

解决这个问题的最有效途径是把不同的视图技术进行分类，针对不同的分类封装不同的视图跳转逻辑，而最重要的一步是将这两者与之前我们所提到的生产线有机结合起来。

问题 6 *MVC 模式虽然很直观地为我们规定了表示层的各种元素，但是如何通过某种机制把这些元素有机整合在一起，从而成为一个整体呢？*

这个问题非常宏观，却是我们不得不去面对的一个问题。MVC 虽然在概念上被规定下来，在实现上却需要一个完整的机制来把这些元素都容纳在一起。通常情况下，我们往往把这种机制称之为配置元素。配置元素是构成程序的重要组成部分，它把各种形式的程

序通过某种配置规则联系在一起。之前我们提到的 URL Mapping 实际上也属于配置规则的一种，视图的跳转也是配置规则的一种。只有当这种配置规则被建立起来，MVC 模式才能真正运作起来。

这一系列配置元素在框架内部往往被定义成统一的可以被框架识别的数据结构并在系统初始化的时候进行缓存。而这些被缓存了的对象，也成为主程序的控制流在 MVC 框架中各个元素之间进行流转的依据。

如果从元素的表现形式上来看配置元素和控制流的关系，我们实际上可以看到整合过程的两个层面：数据结构和流程控制。所谓的框架，我们也只是在这两个层面上做文章，一方面规定好这些配置元素的定义，另一方面指定程序运转的流程，从而控制和整合散落在各处的表示层元素。

2.6　如何学习开源框架

正确的学习方法不仅能够事半功倍，也能够使我们更加接近真理。在了解了框架的本质和 Web 开发模式之后，我们来讨论一下学习开源框架的基本方法。

在这里为大家总结了一些正确的学习方法和最佳实践，这些不仅是笔者多年开发中的心得体会，也汲取了网络上的大家之言，希望对初学者或者正在为学习开源框架犯愁的朋友带来一些启示。这些学习方法，不仅适用于 Struts2，同样适用于许多其他的开源框架。

最佳实践　*阅读、仔细阅读、反复阅读每个开源框架自带的 Reference。*

这是学习框架最为重要，也是最开始最需要做的事情。不幸的是，事实上，绝大多数程序员对此并不在意，并且总是以种种理由作为借口不仔细阅读 Reference。

程序员的常见借口之一：**英语水平跟不上，英文文档阅读起来太吃力**。针对这样的借口，我们需要指出，阅读英文文档是每个程序员必须具备的基本素质之一，这就和调试程序需要耐心一样，对一个程序员来说非常重要。当然，阅读英文文档这一基本素质是一点一滴积累培养起来的，对于那些阅读起来实在觉得吃力的朋友，笔者的建议是结合中文的翻译版本一起看。国内有许多开源组织，例如满江红的开源支持者已经为大家精心做了许多很有价值的翻译，例如 Spring、Hibernate 等都有对应的中文翻译文档。但是大家必须注意，**看中文文档，必须和英文文档对照**，因为没有人可以确保翻译能够百分之百正确，语义的不匹配会给你带来极大的误导，通过对照，才能够将误解降到最低。

程序员的常见借口之二：**Reference 太长，抓不住重点**。在这里，笔者给出的建议是：**耐心，耐心，还是耐心**！从 Reference 的质量而言，其实大多数开源框架的 Reference

都是非常优秀的，基本包含了框架的方方面面。尤其是 Struts2，由于历史原因，Struts2 的 Reference 基本上都是一个一个的专题 Wiki 文章拼起来的文档，每篇文章都有一个固定的主题，不仅包含原理解析、注意事项，有的还包含源码解析和示例讲解。阅读 Reference 可能会非常枯燥，但是从价值的角度看，对 Reference 的阅读往往是对大家帮助最大的。因此，笔者对阅读 Reference 的建议是，**多看几遍**。**第一遍**，你可以采取浏览（scan）的方式，目的是了解框架的整体架构的大致功能。**第二遍**，挑重点章节仔细阅读，并且辅以一定的代码实践，目的是彻底掌握某个分支领域的知识。**第三遍**，带着问题阅读，在文档中寻找答案。

笔者之所以强烈推荐大家仔细阅读开源框架自带的 Reference，主要基于以下的两个原因：

❑ 权威性

这些自带的 Reference 多数出自这些开源框架的作者或者开发人员之手。还有谁能够比他们自己更了解他们自己编写的产品呢？自己写的程序，到底有哪些优点，如何使用，自己肯定是最最清楚的，所以要说到权威性，不可能有任何文档比自带的 Reference 更加权威。

❑ 正确性

自带的 Reference 几乎很少犯错，所以不会给你带来什么误导信息。不仅如此，许多 Reference 已经为你总结了框架使用过程中的许多最佳实践。所以我们没有理由不直接通过这些 Reference 来获得第一手的资料。

最佳实践 *精读网络教程。*

对于很多初学者来说，他们对看 Reference 这种学习方式的接受程度很低。相反，他们会去转而学习一些网络教程。一般而言，这些学习材料的实际价值要比 Reference 低很多。主要原因在于，作者在编写这些教程时，多数都会加入他们自己的学习思路，而忽略了框架本身所期望达到的程序开发最佳实践，甚至会给很多读者以："程序就是这么写的"的误导。 所以，对于网络上的绝大多数网络教程，**需要读者有足够的甄别能力，否则很容易被带入歧途**。

网络上还有很多原版教程，例如《XXX in Action》系列。《XXX in Action》系列的书籍在市场上深受好评。然而，这些系列的书籍有些内容也带有作者个人的感情色彩。当然，每个作者在编写书籍或撰写教程的过程中都会夹带自己的感情色彩，这本不是什么坏事，不过既然我们已经有了 Reference 作为阅读的主体了，对这类书籍，我们需要采取的态度是"**精读**"。

很多网络教程，尤其是中文的网络教程，基本上都是网友的经验之谈，也有写成系列

文章的。对于网络教程，笔者的建议是：**带着问题去读，去搜索你的答案，而不要当作核心文档来阅读**。在找到答案之后，也需要通过实践来反复验证，因为许多解决方案可能只是临时的，并不是程序开发中的最佳实践。

最佳实践　*搭建环境运行每个开源框架自带的 sample 项目。*

每个开源框架基本上都会自带有 sample 项目。以 Struts2 为例，在 Struts2 的分发包的 apps 目录下就有多个 sample 项目，如图 2-9 所示。

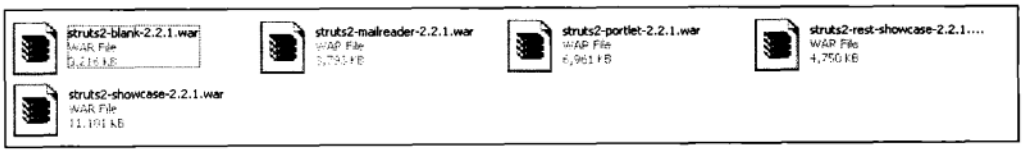


图 2-9　Struts2 自带的 sample 项目

Struts2 是一个典型的 Web 层框架，所以所有 Struts2 的 sample 项目都以 war 包的形式给出，大家可以将这些 war 包的任何一个复制到你的 Web 容器的运行目录下，启动 Web 容器就可以访问这些 sample 项目。

千万不要小看这些 sample 项目，我们可以从这些项目中获取许多重要的知识和信息。有些知识恐怕连 Reference 都不曾提及。这些原生态的东西，使得我们完全无须舍近求远地到网络上去到处寻找例子，只要学习这些例子，就足以掌握开源框架的种种特性了。

我们可以就其中的三个 sample 项目进行举例分析。

❑ struts2-blank-2.2.1.war

一般而言，名为 xx-blank-xxx.war 的 sample 项目是一个开源框架的一个最小可运行范例。所以，如果大家仔细学习这个 war 包中的内容，至少可以发现组成一个 Struts2 程序的最小元素到底有哪些。在其中的 WEB-INF/lib 目录下，我们能够找到 Struts2 程序运行所需要依赖的 JAR 包的最小集合（如图 2-10 所示），我们还能从中学习 Struts2 的各种基础配置的编写等。

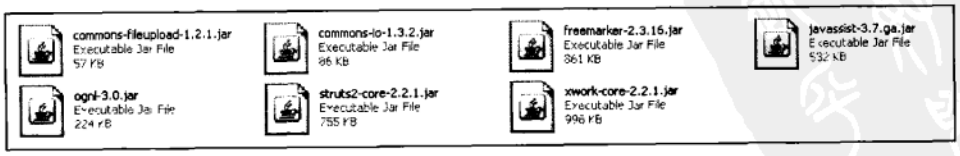


图 2-10　Struts2 所依赖的基本 JAR 包的最小集合

❑ struts2-portlet-2.2.1.war

这个 sample 项目告诉我们在 Portal 环境下的 Struts2 的应用应该如何编写。通过与

struts2-blank-2.2.1.war 这个项目的比较，大家可以发现，Struts2 在应对不同的应用服务器环境方面的不同。

❑ struts2-showcase-2.2.1.war

这个 sample 项目是 Struts2 特性的一个大杂烩，包含了绝大多数的 Struts2 的特性示例。这个 sample 项目对于大家阅读 Reference 是非常有帮助的。比如说，大家在阅读文档时看到了“文件上传”的章节，那么大家就可以参考这个项目中的 upload 子目录中的相关的类和配置。这相当于一边看文档，一边已经有一个现成的可以运行的例子辅助你进行学习。所以，这个项目与 Reference 的搭配是相得益彰、互为补充的，可以作为大家学习 Struts 的最佳资源。

最佳实践 *自己写一个 sample 项目亲身体验。*

这一点其实不用多说，大家也应该明白。不过笔者还是见过不少程序员，眼高手低，整天吹嘘这个框架的优点，那个框架的优势，但如果让他自己动手用这些框架写一段程序，又变得手足无措。

实践是检验真理的唯一标准。只有自己亲自动手去实践，才能说明你真正掌握了某种技术，理解了某个框架的特性。在编写自己的 sample 项目时，大家可以针对不同的特性，人为设置好业务场景（例如，使用“登录”作为一个基本的业务场景），在实践中不断地重构你的代码，从而领悟框架开发中的最佳实践，提升自己的开发水平。

最佳实践 *带着问题调试（Debug）开源框架的源码。*

如果大家对某个开源框架的使用已经比较熟练，对其内部的原理也基本掌握，或许你就会对其中的某些设计原理和设计思想产生兴趣。这个时候，通过开源框架的源码来寻找问题的答案不失为一个很好的进一步学习的途径。

在学习开源框架的源码时，笔者的建议是当程序运行在 Debug 模式的状态下，对源码进行调试，在 Debug 的过程中，查看在开源框架的内部到底运行了哪些类，它们的执行顺序是怎样的以及这些类中临时变量的执行状态。**笔者坚决反对逐个 package 地去阅读源码，这毫无意义**。因为程序本身是一个整体，程序之所以成为程序，其本质在于它是动态的、运行的。如果我们逐一去阅读源码，就相当于把一个完整的整体进行肢体分解，那么我们将永远无法看到一个完整的动态执行过程。学习源码，最重要的一点在于抓住一个程序在运行过程中某一时刻某个关键类的运行状态和最终状态，而这些都能通过调试源码来实现，这才是阅读源码的最佳实践。

2.7　小结

本章讨论的话题是非常重要的，因为任何细节都无法脱离基本概念而存在。如果我们要探寻 Struts2 的细节，就必须了解 Struts2 作为一个框架存在的基本意义。本章从面向对象的基本概念谈起，探讨了框架的本质，揭示了 Web 开发过程与框架之间的依存关系、Web 开发中的一些最佳实践，并由此提出 Web 开发中的一些核心问题。最后，笔者还给出了正确学习 Struts2 的方法供读者参考。这些学习方法是非常有价值的，建议有经验的程序员也看一看。

读完本章，大家不妨带着本章提出的这些核心问题，到本书其他的章节去寻找答案。等到所有的问题都迎刃而解之时，或许大家对于框架和 Web 开发的理解也将更上一层楼。

回顾本章的所有内容，大家对下面这些问题是否有了一个大致的答案呢？

- 对象有哪三种构成模式？
- 对象有哪些关系模型？
- 什么是框架？框架存在的根本目的是什么？
- 在整个 Web 开发的过程中，我们应该牢记哪些最佳实践？
- 什么是 MVC 模式？ MVC 模式对于 Web 开发的主要作用是什么？
- 在 Web 开发中，我们将遇到哪些主要的困境？
- Struts2 运行所依赖的最少的 JAR 文件资源的组合有哪些构成？
- 如何正确学习一个开源框架？

第 3 章　提纲挈领——Struts2 概览

3.1　Struts2 的来世今生

作为 Apache 旗下非常重要的开源项目，Struts 这个项目的历史发展进程可以说比较特殊。Struts2 的来世今生，从一个侧面反映出了整个 Web 开发的发展历程。在这里，我们简单对 Struts 项目的发展历程做一个简单的介绍，从而帮助大家对 Web 层框架的发展有更加深刻的了解。

查阅一下 Struts 的历史就可以发现，最早的 Struts 可以追溯到 2000 年 3 月 31 日，当时 Apache 有一则新闻，宣布开始编写一个新的 Web 开发框架——Struts。

> *Craig McClanahan has contributed a small framework for building web applications using the Model-View-Controller (MVC) design pattern commonly known as "Model 2" to the Jakarta project. The "Struts" project will serve as a platform for exploring the optimum approaches to implementing this design pattern, in a manner that can facilitate tool-based generation of application components.*

项目取名为“Struts”，有“基础构建”的含义，在那个开发框架尚处于朦胧阶段的年代，“基础构建”无疑是每个程序员梦寐以求的东西。在新闻发布后不久，Apache 便发布了 Struts1.0.X 的若干个版本和针对 Struts1.0.X 的升级版本 Struts1.1.X。自此以后，Struts1.X 系列开发框架就开始在世界范围内流行起来。在很长一段时间里，其流行程度几乎可以说垄断了整个 Web 开发领域，成为 Web 开发领域的实际开发标准。

Struts1.X 系列开发框架能够如此流行的主要原因在于：首先，它是 Apache 出品，所谓树大好乘凉，Apache 巨大的社区开发优势对它在全世界的流行起到了不可估量的作用；其次，Struts1.X 也是较早实现 MVC 模式中“Model 2”概念的 Web 开发框架，在一定程度上它占有“天时”方面的优势；最后，各大中间件厂商尤其是 IBM 等公司对 Struts1.X 的巨大支持，为 Struts1.X 的推广起到了推波助澜的作用。

随着时代的进步，越来越多的程序员在使用 Struts1.X 进行开发的过程中发现 Struts1.X 在设计上存在着严重不足。与此同时，各种各样的 Web 层开发框架也如雨后春笋般涌现出来。在这些框架中，有一个来自于 Opensymphony 开源社区的优秀框架 Webwork2 逐渐

被大家熟识和理解，并不断发展壮大。2004 年 12 月 14 日，Webwork2.1.7 版本发布，成为 Webwork2 的一个重要里程碑，它以优秀的设计思想和灵活的实现，吸引了大批 Web 层开发人员投入它的怀抱。在这之后，Webwork2.2.X 的若干个版本依次发布，并开始支持 JDK1.5 的相关特性，极大地解放了 Web 层开发的生产力。

或许是看到了 Struts1.X 发展上的局限性（主要是 Struts1.X 在设计思想和技术实现上的种种局限性），Apache 社区与 Opensymphony 开源组织在 2005 年 12 月 14 日宣布未来的 Struts 项目将与 Webwork2 项目合并，并联合推出 Struts2，通过 Apache 社区的人气优势与 OpenSymphony 的技术优势，共同打造下一代的 Web 层开发框架。当时的新闻写到：

> *Apache Struts, the leading web application framework for Java, and Open Symphony WebWork, a leader in technical innovation, are working to merge their communities and codebases.*

从这次合并中，我们可以非常明显地看到 Apache 社区在技术层面上的一个重大妥协。虽然 Struts1.X 借助 Apache 的社区力量占领了市场和人气，然而在技术上，Webwork2 却完胜 Struts1.X，合并之后的 Struts2 将完全采用 Webwork2 作为其代码的基础，摒弃 Struts1.X 的所有设计和代码。

所以，研究 Struts2 时必须明确一个非常重要的结论：

结论　Struts2 来自于 Webwork2，并且与 Struts1.X 完全不兼容。

认识这一点对所有 Struts 开发人员来说非常重要。而本书所讲解的内容，也全部都是针对 Struts2 而不是 Struts1.X，这将成为我们之后所有讨论的基础。

经过几年的发展，Struts2 又陆续发布了 Struts2.0.X、Struts2.1.X 和 Struts2.2.X 的若干个版本，伴随着版本的不断升级，我们不仅从中获得了越来越便利的开发模式，也可以体会到整个 Web 开发的发展历程，从一个侧面印证了技术的发展与设计思想的不断进步。

3.2　Struts2 面面观

在深入研究 Struts2 内部的实现机理之前，我们将从 Struts2 的运行环境、应用场景以及核心技术这三个方面对 Struts2 进行简单的介绍。这三个不同的方面是描述 Struts2 外部环境的三个不同角度，能够帮助我们站在一个更高的高度对 Struts2 作为表示层框架在 Web 开发中所起的作用和 Struts2 与其他核心技术之间的关系有一个更深刻的理解。

3.2.1 Struts2 的运行环境

在上一章中，我们就谈到本书的核心内容是通过对 Struts2 的分析，帮助读者领略 Web 开发的方方面面。从这一点上讲，我们的话题也就始终无法脱离 Web 开发以及 Web 应用程序的运行环境。Struts2 在典型的三层结构的开发模式中被视作“表示层”的解决方案。因而，其最为核心的内容就是**和 Web 容器打交道，帮助我们处理 Http 请求。**

结论 *Struts2 是一个运行于 Web 容器的表示层框架，其核心作用是帮助我们处理 Http 请求。*

请读者牢记这一条结论，因为这条结论不仅是整个 Struts2 框架的指导思想，也规定了 Struts2 作为一个框架所涉及的问题作用域。从这个结论中，我们很容易得到与 Struts2 的运行环境相关的两条推论：

推论 *Struts2 的运行环境是 Web 容器。*

这条推论显而易见，其中却也隐含着许多重要内容。而最为重要的就是 Struts2 对其运行环境的 Web 容器有版本要求。我们知道，运行于 Web 容器中的程序，首先必须遵循基本的开发标准和规范：Servlet 标准和 JSP 标准等等。而 Java 中的 Web 开发标准有着不同的版本，不同的 Web 服务器对于 Servlet 标准和 JSP 标准的支持程度也是不同的。**对于 Struts2 而言，它所支持的 Servlet 标准的最低版本要求是 2.4，相应的 JSP 标准的最低版本要求是 2.0。**这就对使用 Struts2 作为开发框架的应用程序的运行环境提出了要求，这一要求就像 Struts2 要求必须运行在 JDK1.5 版本之上一样。

结论 *Struts2 通过扩展实现 Servlet 标准来处理 Http 请求。*

如果把整个 Web 容器看作是一个黑盒，Struts2 无非也只是黑盒中所运行的一段程序代码段，如图 3-1 所示。

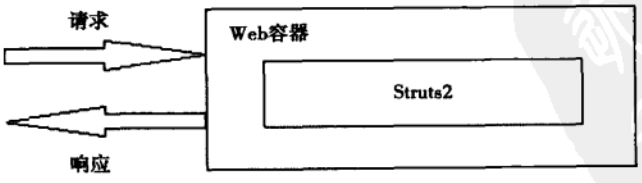


图 3-1 Web 容器的黑盒模型

从图中我们可以看到，Struts2 这个框架在其中的作用实际上只是处理 Http 请求

（Request），并进行内部处理（处理的过程被黑盒屏蔽了，我们暂时也无须关心），再进行 Http 返回（Response）。而**整个这个过程的代码级别的实现，无论如何进行封装，都离不开对 Servlet 标准或者 JSP 标准所指定的底层 API 的调用**。从这个角度来说，Struts2 只是实现了一个**具备通用性的 Http 请求处理机制**，并且能够被我们的应用程序使用和扩展而已。

我们在这里花费一些笔墨向读者强调 Struts2 运行环境的目的在于提醒读者不应该忘记程序运行的基础条件。不要把任何一个框架看得很神秘，它们的本质也只不过是实现了基本的开发协议或者开发规范的程序集合而已。

3.2.2　Struts2 的应用场景

在了解了 Struts2 的运行环境之后，Struts2 的应用场景的问题也就能够迎刃而解了。当我们使用 Java 语言编写一个 Web 应用程序时，Struts2 或许就能够帮得上忙。这就是 Struts2 的应用场景：**帮助我们编写 Web 应用程序**。

细心的读者会发现我们在这里使用了“或许”两个字。为什么要使用“或许”呢？Struts2 是否能够成为所有 Web 应用的最佳选择呢？这个问题，我们还得回到上一章中曾经提到过的一个 Struts2 的官方表述，如图 3-2 所示。

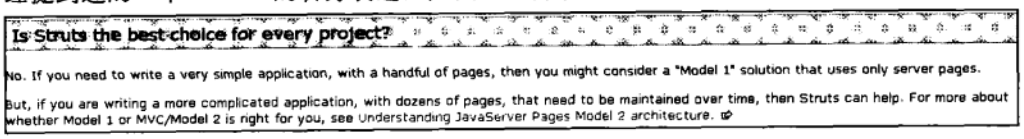


图 3-2　Struts2 官方对 Struts2 应用场景的解释

非常明显，Struts2 官方为我们描述的 Struts2 的应用场景是：

结论　*Struts2 并不适合所有的 Web 应用，但它却是复杂的、可扩展的 Web 应用的一剂良药。*

因为当需要建立一个由复杂的业务逻辑和众多页面构成的 Web 应用时，我们不得不采用分层开发模式。例如，我们可以采用典型的三层开发模式，将表示层、业务逻辑层和持久层的逻辑完全分开，从而获得更好的程序可扩展性和可维护性。此时，Struts2 就可以作为一个表示层的解决方案，帮助我们进行与表示层相关的开发工作，并提供在表示层范围之内高度的可扩展性和可维护性。

由此可见，在讨论 Struts2 的应用场景时，必须首先明确我们要开发的 Web 应用的规模和实际情况，并且理解 Struts2 自身的作用范围，从而能够合理地选择最适合的开发框架。

3.2.3 Struts2 的核心技术

作为一个 Web 开发的解决方案，Struts2 并不是一个可以独立运行的开源框架。Struts2 的实现，首先将基于最为基本的 Web 开发标准。除此之外，Struts2 自身还依赖于一些其他的开源框架和解决方案。我们在这里简单归纳一下这些 Struts2 的实现所涉及的核心技术，在本书的核心技术篇中，我们将对其中的核心技术进行详细分析。

3.2.3.1 Struts2 与表示层技术

Struts2 首先运行于 Web 容器之中。因而，它的核心依赖就是 Web 容器对于 **Servlet 标准和 JSP 标准**的实现。我们在之后的章节中对 Struts2 的主要分析，也将围绕着 Struts2 如何扩展实现这些基本的标准实现来展开。

作为一个服务于表示层的解决方案，Struts2 有时候不得不与许多其他的表示层技术进行整合。例如，以 Freemarker 或者 Velocity 为核心的**模板技术**、构建 Flash 应用的 **Flex 技术**、**Ajax 技术**等等。这些技术往往本身自成体系，而 Struts2 需要做的只是通过扩展实现一些 Servlet 标准与这些技术进行底层沟通从而完成与这些技术的整合。

Struts2 与这些表示层技术的整合，往往通过“插件”的方式进行。有关插件（Plugin）的相关知识，我们将在本书的第 12 章中重点展开。在这里，我们所需要了解的是：开发人员可以根据项目实际情况，选择合适的插件并将其引入到 Struts2 的运行环境中。Struts2 在运行时能够自动识别这些插件，从而完成与相关技术的自动整合。

3.2.3.2 Struts2 与设计模式

设计模式（Design Pattern）是我们在日常编程中经常听到的一个名词。实际上所有的设计模式只不过是代码级别最佳实践的具体表现。

因此，任何设计模式都不是什么核心技术，也不是 Struts2 的实现严格依赖的内容主体。然而，Struts2 的内部实现却离不开这些设计模式。有一些核心的设计模式，甚至贯穿了整个 Struts2 的逻辑主线，成为 Struts2 内部实现中不可或缺的重要组成部分。

在 Struts2 中，我们将先后接触到命令（Command）模式、ThreadLocal 模式、装饰（Decorator）模式、策略（Strategy）模式、构造（Builder）模式、责任链（Chain Of Responsibility）模式、代理（Proxy）模式等等。这些设计模式的反复使用，使得 Struts2 的实现本身就充满了最佳实践。我们将在本书的第 4 章中重点就 Struts2 所用到的一些核心设计模式进行深入剖析，帮助读者更加深刻地理解这些设计模式在框架中的运用场景。

3.2.3.3 Struts2 与 OGNL

有过 Web 程序开发经验的读者会发现，表达式引擎是 Web 编程必不可少的重要元素。所谓**表达式引擎**，指的是通过程序建立起某个实体对象与某种公式表达之间的联系。在

Java 世界，这种联系具体表现为：**使用某些符合特定规则的字符串表达式来对 Java 中的对象进行读和写的操作。**

表达式引擎对 Web 开发的重要之处在于它在一定程度上解决了 Web 应用与 Java 世界之间的沟通问题。既然要使用 Java 来开发 Web 应用，就必须使 Java 的编程要素能够与 Web 浏览器之间在数据层面保持良好的沟通，而这种沟通就是通过表达式引擎来完成的。

Struts2 作为 Web 层的开发框架，也势必要借助一个强大的表达式引擎来实现上述功能。因而，Struts2 选择了在表达式引擎这一领域表现极为非常出色的 OGNL 作为其所依赖的表达式引擎。换言之，**OGNL 是 Struts2 运行所依赖的基本核心技术之一**。OGNL 的意义不仅在于完成不同形式数据之间的转化和通信，它也是 Struts2 实现视图层的基本依据。在之后的章节中，我们会发现这将成为 Struts2 区别于其他 Web 层框架的实现视图层与 Web 服务器之间数据交互的最为显著的特点。

OGNL 表达式引擎也有较长的历史，它的早期版本的性能曾经遭到质疑。不过，它的性能问题已经在最新的 3.0 版本中得到了极大的改善。Struts2 的最新版本 Struts2.2.1 所依赖的 OGNL 为 3.0。有关 OGNL 的相关知识，我们将在第 5 章详细介绍。

3.2.3.4　Struts2 与 XWork

了解了 Struts2 的历史之后，我们知道 Struts2 来源于 Webwork2。Webwork2 之所以能够在技术上击败 Struts1.X 而成为新的框架 Struts2 的构建基础，主要是源于其优秀的设计思想。或许我们很难用一句话来准确描述这一设计思想的优越性体现在何处，但是当我们试图总结这种优秀思想中所蕴含的核心基石时，XWork 将当仁不让地成为其中关键字的第一位。我们可以在 XWork 的官方网站上找到对 XWork 的一个大概描述：

> *XWork is a command-pattern framework that is used to power Struts 2 as well as other applications. XWork provides an Inversion of Control container, a powerful expression language, data type conversion, validation, and pluggable configuration.*

XWork 是 Opensymphony 开源组织贡献的另外一个开源项目，从其官方网站的介绍来看，XWork 不仅提供了一系列基础构件，其中包括：一个 IoC 的容器、强大的表达式语言（OGNL）支持、数据类型转化、数据校验框架、可插拔的功能模块（插件模式）及其配置，并且在这一系列的基础构件之上，实现了一套基于 Command 设计模式的“事件请求执行框架”。

什么是“事件请求执行框架”呢？首先，这是一个基于“请求－响应”的处理器，能够对某一类“请求”做出相应的逻辑处理，并返回“响应”结果。其次，它定义了一套完整的事件逻辑处理的步骤，并且为每个步骤都提供了足够的扩展接口，使得整个事件的执行体系更为丰富、更加具备层次感和可扩展性。因而，“事件请求执行框架”就**如同一条**

定义好的生产流水线，能够为我们提供完整的事件处理模型。回顾一下我们在上一章中提到的一个事件处理流水线的问题，XWork 正是这样一个解决方案。

我们知道，所有 B/S 程序都是典型的基于“请求－响应”模式的 Web 应用。因而 XWork 天然地成了处理这种应用最合适的方案。有了 XWork 作为 Struts2 所依赖的底层核心，使得 Struts2 只需要关注与 Web 容器打交道的部分，而把其余的工作交给 XWork 即可。当 Struts2 收到一个 Http 请求时，Struts2 只需要接收请求参数，交给 XWork 完成执行序列，当 XWork 执行完毕后，将结果交还 Struts2 返回相应的视图。可以看到，在整个过程中，XWork 是这个“请求－响应”模式的执行核心。

XWork 对于 Struts2 的地位如此重要，以至于 Struts2 的重大版本总是跟随着 XWork 的升级而变更。需要注意的是，Struts2 的不同版本对 XWork 的版本依赖也是不同的，它们甚至是不兼容的。下面列出了 Struts2 发布以来它与 XWork 版本的兼容性情况：

- struts2-core-2.0.x——xwork2.0.x
- struts2-core-2.1.x——xwork2.1.x
- struts2-core-2.2.x——xwork2.2.x

XWork 不仅是 Struts2 的核心实现，也可以用于一切基于 Command 模式的 Java 程序。在实现 Command 模式时，XWork 在其周围定义了丰富的执行层次，在每个执行层次中，都有足够的扩展点，使得我们可以将 XWork 视作一个工具包，简化我们的开发。

3.3 多视角透析 Struts2

Struts2 的外部环境并不复杂，因为其核心内容非常明确：探究 Struts2 运行时所必需的基本要素。我们对 Struts2 的运行环境和 Struts2 所依赖的核心技术的讲解，主要是为了让读者了解 Struts2 能够顺利运行的条件。

在明确 Struts2 的外部环境之后，我们讨论的话题就将转向 Struts2 本身。在本节中，我们将从宏观和微观这两个不同的视角，阐述 Struts2 的总体架构和内部元素构成，以此揭开 Struts2 的神秘面纱。

3.3.1 透视镜——Struts2 的宏观视图

Struts2 的宏观视图是指站在整个框架的角度，了解程序的运行可以划分为哪些逻辑运行主线。对于一个框架的逻辑运行主线的研究，也是我们分析一个框架最为重要的切入点。而这一切入点，就位于 Struts2 的核心入口程序之中。

3.3.1.1 Struts2 的核心入口程序

Struts2 的核心入口程序，从功能上讲必须能够处理 Http 请求，这是表示层框架的基

本要求。为了达到这一要求，Struts2 毫无例外地遵循了 Servlet 标准，通过实现标准的 Filter 接口来进行 Http 请求的处理。我们通过在 web.xml 中指定这个实现类，就可以将 Struts2 框架引入到应用中来，如代码清单 3-1 所示。

代码清单 3-1　web.xml

```
<filter>
    <filter-name>struts</filter-name>
   <filter-class>org.apache.struts2.dispatcher.ng.filter.
StrutsPrepareAndExecuteFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>struts</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

打开 StrutsPrepareAndExecuteFilter 的源码，可以发现它只是一个实现了 Filter 接口的实现类。其方法列表如图 3-3 所示。

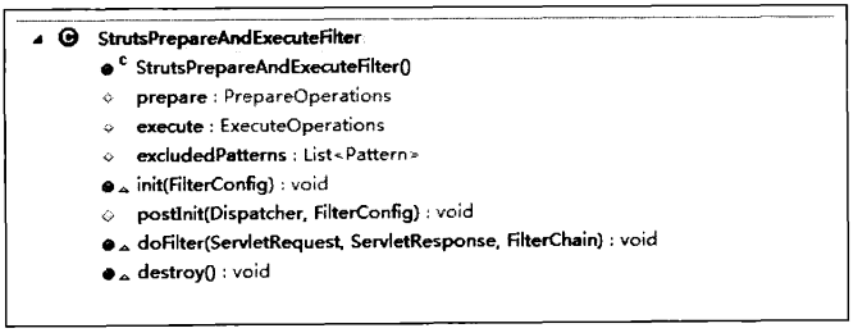


图 3-3　StrutsPrepareAndExecuteFilter 方法列表

根据 Servlet 标准中 Filter 的生命周期的相关知识，我们知道 Filter 中所定义的方法具有完全不同的执行时间段和生命周期，它们的执行互不影响，没有交叉。而 Filter 的生命周期也成为我们对整个 Struts2 进行运行逻辑主线划分的主要依据。

- **第一条主线 —— Struts2 的初始化：init 方法驱动执行**
- **第二条主线 —— Struts2 处理 Http 请求：doFilter 方法驱动执行**

在这里，我们首先不对 StrutsPrepareAndExecuteFilter 的源码做深入的探讨，不过可以先把这个入口程序的基本结构和功能结合我们对 Struts2 划分的两条逻辑主线，用示意图的形式表达出来，从而帮助读者对入口程序的运行逻辑主线有个初步的认识，如图 3-4 所示。

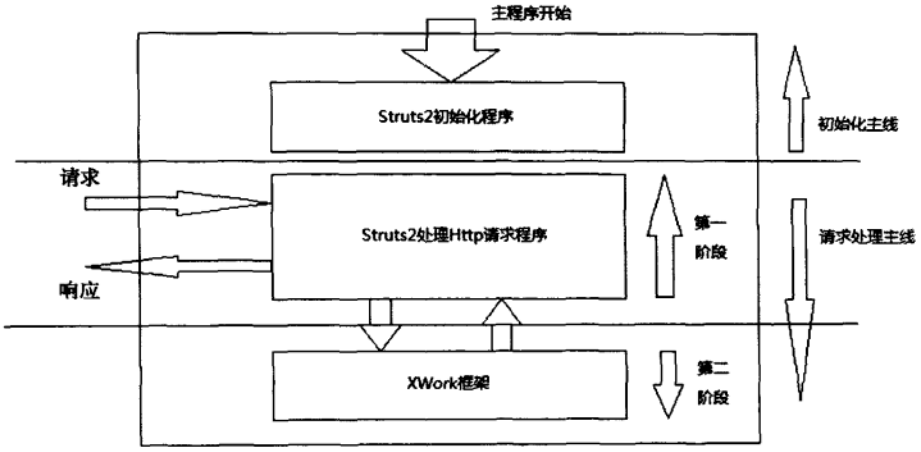


图 3-4 Struts2 入口程序执行示意图

从图中，我们可以清晰地看到 Struts2 的两条逻辑主线之间由一条分隔符分开。不同的逻辑主线之间完全没有交叉，驱动它们执行的**时间节点和触发条件**都不同。因而，我们日后对 Struts2 运行逻辑的分析，也将围绕着这两条主线分别展开。

Struts2 入口程序的示意图，可以看作是对 Web 容器黑盒模型的第一层细化，读者在这里应该更多关心程序运行的基本方向，其内部细节有待我们在之后的章节为大家一一解开。

3.3.1.2 Struts2 的初始化主线

Struts2 的初始化主线发生在 Web 应用程序启动之初，由入口程序的 init 方法驱动执行完成。这条运行主线的主要特点有：

❑ 仅在 Web 应用启动时执行一次

由于这条主线由 Filter 中的 init 方法驱动执行，执行完毕后，该主线结束。也就是说，这条主线本身不参与后面任何 Http 请求的处理过程，无论 Struts2 之后面再收到多少 Http 请求，这条主线都不会重复执行。

❑ init 方法的执行失败将导致整个 Web 应用启动失败

如果在 init 方法执行的过程中发生异常，整个 Web 应用将无法启动。这个特点从框架规范的角度规定了我们必须确保初始化过程的顺利执行。因为在这个过程中，所有框架内部定义的元素将被初始化，并支撑起整个 Struts2 进行 Http 处理的过程。

这两大特点本身其实来源于 Filter 这个 Servlet 规范的基本运行特性。然而，这两大特点却也为应用程序在框架的基础之上进行逻辑扩展提供了理论上的指导。在之后有关如何扩展框架的话题讨论中，我们可以看到所有的扩展方案都将基于这两大特点进行设计。

那么，Struts2 的初始化主线到底做了什么呢？对应于 Struts2 初始化的运行特点，Struts2 的初始化主线也有两大主要内容：

❑ 框架元素的初始化工作

这一初始化工作包含了对框架内部的许多内置对象的创建和缓存。我们发现，对于框架初始化工作的基本要求，就是在整个框架的运行过程中仅执行一次，这正好符合这条主线的基本特点。

❑ 控制框架运行的必要条件

框架的可扩展特性保证了我们可以在应用层面对框架的运行参数和执行模式进行定制化，而框架则有必要对这种定制化进行正确性校验。当这种校验失败时，Web 应用的启动会失败。这也就是 Struts2 在框架级别所能够提供的运行期的检查。

初始化主线贯穿了 Struts2 对其内置对象的创建和缓存的过程，这一过程相当于把整个 Struts2 作为一个框架的运行环境完整地创建出来。这条主线的顺利运行，为之后的 Http 请求处理主线提供了必要的框架运行环境。

我们在这里所说的运行环境和 Struts2 自身的运行环境不同，它是指建立在 Web 服务器之上，框架自身运行所必需的内置对象的集合。为了更好地对这些内置对象进行管理，Struts2 引入了框架级别“容器”的概念。因而 Struts2 的初始化主线，实际上最终转化为对这个“容器”的初始化过程。有关这个“容器”的定义和初始化过程的细节，我们将在第 5 章为读者解开谜团。

3.3.1.3　Struts2 的 Http 请求处理主线

Struts2 的 Http 请求处理主线是 Struts2 的核心主线，包含了 Struts2 处理 Http 请求、进行必要的数据处理和处理数据返回的全部过程。这条主线将在任何满足 web.xml 中所指定的 URL Pattern 的 Http 请求发生时进行响应，由 doFilter 方法负责驱动执行。

回顾一下 Struts2 核心入口程序的流程图（图 3-4），我们可以看到 Struts2 的 Http 请求处理主线又被一条分割线划分成了两个不同的执行阶段：

❑ 第一阶段——Http 请求预处理

在这个阶段中，**程序执行的控制权在 Struts2 手上**。这个阶段的主要工作是针对每个 Http 请求进行预处理，为真正的业务逻辑执行做必要的数据环境和运行环境的准备。

程序代码在这个阶段有一个非常显著的特点：**依赖于 Web 容器，并时时刻刻将与 Web 容器打交道作为主要工作**。

❑ 第二阶段——XWork 执行业务逻辑

在这个阶段，**程序执行的控制权被移交给了 XWork**。Struts2 在完成 Http 请求的预处理之后，将 Http 请求中的数据封装成普通的 Java 对象，并由 XWork 负责执行具体的业务逻辑。

程序代码在这个阶段的特点和第一阶段完全相反：**不依赖于 Web 容器，完全由 XWork 框架驱动整个执行的过程。**

从 Struts2 对于 Http 请求的处理过程中，我们可以看出 Struts2 的核心设计理念在于**解耦**。所谓解耦，实际上是尽可能地消除核心程序对外部运行环境的依赖，从而保证核心程序能够更加专注于应用程序的逻辑本身。在 Struts2 中，我们所说的外部运行环境就是 Web 容器。我们在这里可以看到，Struts2 的核心设计理念与 Struts2 的运行环境居然是一个矛盾体！

结论 *Struts2 的核心设计理念在于消除核心程序对运行环境（Web 容器）的依赖，而这一过程也是 Struts2 的解耦过程。*

这种设计实现与目的之间的矛盾，却是 Struts2 的设计始终围绕着 Web 开发中的最佳实践的最有力证明，也是 Struts2 从设计理念开始，就优于其他表示层框架的精要所在。在解耦方面，Struts2 也确实做到了 2 个不同的层面，从而使整个设计更加突显出其优秀之处：

❑ 从代码上进行物理解耦

Struts2 将第一阶段中的代码整合到 struts2-core-2.2.1.jar，而将第二阶段中的代码整合到 xwork-core-2.2.1.jar。

❑ 将逻辑分配到不同的执行阶段

Struts2 将处理数据的逻辑和处理业务的逻辑分配到 2 个不同的执行阶段，使得我们对于代码逻辑的关注点更为清晰。

因此，正如我们在之前的章节所谈到的，**严格意义上的 Struts2，实际上由 2 个不同的框架所组成。**一个是真正意义上的 Struts2，另外一个是 XWork。从职责上来说，**XWork 才是真正实现 MVC 的框架，**Struts2 的工作是在对 Http 请求进行一定处理后，委托 XWork 完成真正的逻辑处理。**将 Web 容器与 MVC 实现分离，是 Struts2 区别于其他 Web 框架的最重要的特性，也是最值得我们品味的一个宏观设计思路。**当读者真正理解了其中的奥秘，相信也就真正掌握 Web 开发之道了。

3.3.2 显微镜 —— Struts2 的微观元素

在了解了 Struts2 的宏观面之后，我们再来一起探究一下构成这些宏观面的微观元素。同样，不同的主线和不同的阶段所承担的职责不同，构成它们的微观元素也不尽相同。在本节中，我们将列出每条主线和每个执行阶段的主要组成元素。或许在这其中大家会接触到许多新名词，读者不妨首先感性地认识一下这些概念性的名词，可以不求甚解，因为我

们会在后续的章节中对 Struts2 的这些元素一一展开分析和讲解。

3.3.2.1　第一条主线 —— Struts2 的初始化

在对 Struts2 初始化主线的宏观分析中，我们曾经谈到为了帮助更好地管理 Struts2 中的内置对象，Struts2 引入了一个“容器”的概念，将所有需要被管理的对象全部置于容器之中。因而，整个 Struts2 初始化过程也始终围绕着这个“容器”展开。除了“容器”，Struts2 中的另一类配置元素 PackageConfig，也是 Struts2 初始化的主要内容之一。如果我们从“数据 + 行为”的角度来分析，那么构成 Struts2 整个初始化过程的主要元素，就可以分为数据结构的定义和初始化行为的操作接口两个部分。

从数据结构定义的角度，“容器”顺理成章地成为 Struts2 初始化主线中的核心构成元素，而 PackageConfig 作为事件请求映射的配置元素也成为我们所需要重点关注的构成元素。它们的接口定义和实现类如表 3-1 所示。

表 3-1　Struts2 中的容器及其实现类

元素名称	Java Package	描述
Container	com.opensymphony.xwork2.inject	容器定义接口，是 Struts2 内部进行对象管理的基础构建
ContainerImpl	com.opensymphony.xwork2.inject	容器的实现类，内部实现了 Struts2 进行对象生命周期管理和依赖注入的基本功能
PackageConfig	com.opensymphony.xwork2.config.entities	PackageConfig 实体类，其中定义了事件响应模型的完整数据结构

表 3-1 的定义从数据结构的角度指出了 Struts2 初始化流程的主要对象是哪些。而整个初始化的操作过程，则由另外两个相辅相成的元素配合共同完成，它们分别是**加载接口**（Provider）和**构造器**（Builder），其相关元素如表 3-2 所示。

表 3-2　Struts2 中容器的加载接口（Provider）和容器的构造器（Builder）

元素名称	Java Package	描述
ConfigurationProvider	com.opensymphony.xwork2.config	配置加载接口的统一接口。Struts2 将初始化元素分为 Container 和 PackageConfig 两类，这里使用了多重继承将两类配置加载接口进行统一
ContainerProvider	com.opensymphony.xwork2.config	Container 的配置加载接口，其实现类需要负责初始化容器中的所有对象
PackageProvider	com.opensymphony.xwork2.config	PackageConfig 的配置加载接口，其实现类需要负责初始化用于处理事件请求的配置对象

（续）

元素名称	Java Package	描述
ContainerBuilder	com.opensymphony.xwork2.inject	Container 的构造器，用于在初始化时构造容器
PackageConfigBuilder	PackageConfig 的内部类	PackageConfig 的构造器，用于在初始化时构造 PackageConfig

Struts2 初始化主线中还有一些辅助元素，它们主要用于承载这些配置加载接口并在初始化时驱动整个初始化流程的顺利执行。相关元素如表 3-3 所示。

表 3-3 Struts2 初始化主线中的辅助元素

元素名称	Java Package	描述
ConfigurationManager	com.opensymphony.xwork2.config	配置行为操作代理类，包含了所有 ContainerProvider 和 PackageProvider 的实现以及所有配置的结构化数据（Configuration）
Configuration	com.opensymphony.xwork2.config	Struts2 配置数据的管理类，作为运行时获取配置的基本接口。承载所有配置的结构化数据和操作方法

Struts2 的初始化是一个非常复杂的流程。在这里我们仅仅给出了部分主要的数据结构和基础操作接口。读者可以使用 IDE 的源码查看功能找到这些接口的众多实现类。从这些实现类中，大家会发现这些实现类实际上是对整个 Struts2 配置元素的一个总串联，之后的章节我们会通过源码分析它们的联系和运行机理。

3.3.2.2 第二条主线——第一阶段：Http 请求预处理

在这个阶段，我们知道程序的执行控制权还在 Struts2 手中。所以在这个阶段中所涉及的主要微观元素，都是 Struts2 的类。这些类的主要职责是与 Web 容器打交道。因而，从设计原则上，为了保持解耦，这个阶段做了大量的对象创建和对象转化的工作。当这些工作完成之后，就能交付第二个阶段继续执行。这个阶段的主要微观元素如表 3-4 所示。

表 3-4 Struts2 进行 Http 请求预处理阶段的主要微观元素

元素名称	Java Package	描述
Dispatcher	org.apache.struts2.dispatcher	Struts2 的核心分发器，是 Struts2 进行 Http 请求处理的实际场所
PrepareOperations	org.apache.struts2.dispatcher.ng	Struts2 进行 Http 请求预处理的操作集合
ExecuteOperations	org.apache.struts2.dispatcher.ng	Struts2 进行 Http 请求逻辑处理的操作集合

虽然这个阶段所涉及的元素很少，但是其中的 Dispatcher 却是整个 Struts2 框架的核心。Dispatcher 被称为核心分发器，是 Struts2 进行 Http 请求处理的实际场所。在整个处理流程中，Dispatcher 不仅是 Http 请求预处理的实际执行者，更是将 Http 请求与 Web 容器进行解耦并进行逻辑处理转发的执行驱动核心。事实上，我们对 Struts2 框架内部机理的研究，都将以 Dispatcher 作为重要的切入点而展开。

3.3.2.3　第二条主线 —— 第二阶段：XWork 执行业务逻辑

在这个阶段，程序的执行控制权被移交到了 XWork 框架中。我们可以想象位于这个执行阶段的微观元素区别于第一个阶段的显著特点：它们都是 XWork 框架所定义的类。其相关微观元素的定义如表 3-5 所示。

表 3-5　XWork 执行业务逻辑阶段的主要微观元素

元素名称	Java Package	描述
ActionProxy	com.opensymphony.xwork2	XWork 生产线中的执行环境，整个生产线的入口，如一个口袋一样封装了所有执行细节
ActionInvocation	com.opensymphony.xwork2	XWork 生产线中的调度者，负责调度整个生产线中各个元素的执行次序
Interceptor	com.opensymphony.xwork2.interceptor	XWork 生产线中的工序序列，可以丰富整个生产线的功能
Action	com.opensymphony.xwork2	XWork 生产线中的核心工序，负责核心业务逻辑调用和实现
ActionContext	com.opensymphony.xwork2	XWork 生产线的辅助设备，提供整个生产线工作运行所必需的数据环境
ValueStack	com.opensymphony.xwork2.util	XWork 数据环境中提供表达式运算的工具类，也是 XWork 中进行数据访问的基础
Result	com.opensymphony.xwork2	XWork 生产线中的末端设备，负责输出生产线的生产结果

在这里，我们形象地把 XWork 比喻成一条生产流水线。在这其中的每个元素，就像是生产线中的重要组成部分。XWork 框架就像一个完整的事件执行器，进入框架中的事件就如同进入生产线中的原材料，会按照生产线中的定义依次执行并产生结果。

表 3-5 中的七个元素，被称为 XWork 的七大元素，贯穿了 XWork 事件执行器的整个生命周期。它们各司其职、精心配合，提供了事件执行框架足够的扩展接口。不仅如此，它们之间的调用关系也成为 XWork 框架设计中的经典。这些元素的调用关系，如图 3-5 所示。

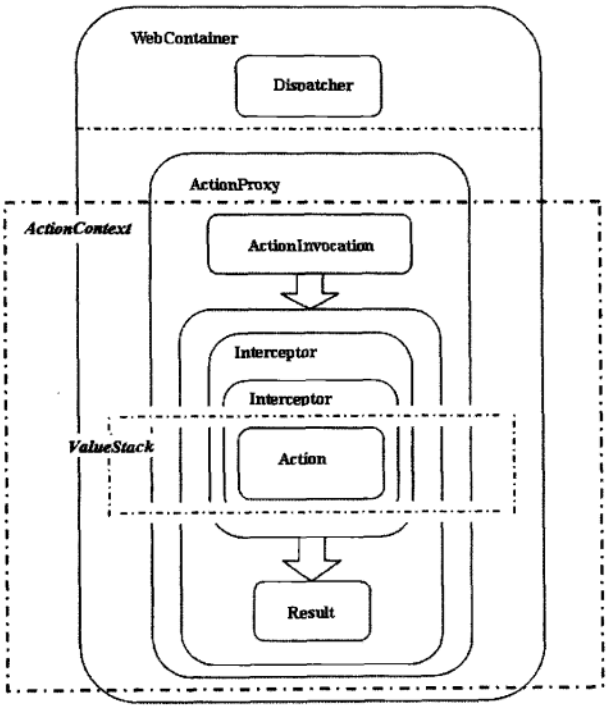


图 3-5 XWork 元素的调用关系图

这幅图不仅包含了 XWork 框架中各个元素的调用关系，还把整个 XWork 框架置于 Web 容器这样一个大的环境之中，同时给出了 XWork 框架的调用核心 Dispatcher。我们在这里不再对这幅图做深入的剖析，读者可以大致了解 XWork 中这些元素的结构和层次关系。在第 8 章中，我们将从数据流和控制流这两个不同的角度，对这张图的程序流转方向进行详细的分析。

3.4 Struts2 的配置元素

上一节中，我们分别从宏观层面和微观层面对 Struts2 进行了初步的透视分析。从框架中提炼出了 Struts2 运行的 2 条主线和 2 个执行阶段，并以此为基础列出了每条主线和每个执行阶段的核心元素。2 条主线和 2 个执行阶段的概念之所以被反复提及，是因为 Struts2 的所有内容全都无法脱离这些概念而独立存在，这些概念不仅支撑起了 Struts2 的核心构架，同时也是我们研究 Struts2 中构成元素和运行机理的主要依据。

我们在这里所说的“**构成元素**”和“**运行机理**”，实际上从另外一个角度表述了程序

的构成方式。任何程序，如果我们从组织结构上进行分析，总是由两大类元素组成：一类用于描述问题，这类元素我们通常称之为**数据结构（构成元素）**；另一类元素则是在数据结构的基础之上执行的逻辑代码，这类元素我们通常称之为**算法（运行机理）**。数据结构和算法的有机结合，构成了可运行的程序主体。这其实也是我们经常听到的一条结论：

结论　*程序 = 数据结构 + 算法（构成元素 + 运行机理）*

对“构成元素”的研究，有助于我们站在全局的观点来审视支撑整个 Struts2 运行的底层核心。而**这些构成元素，需要一个贯穿始终的粘合剂，不仅能够以一定的形式表现出这些构成元素互相之间的逻辑关系，同时能够将它们的执行逻辑串联起来。**这种粘合剂，实际上就是框架的核心配置。在本节中，我们就来重点讨论一下 Struts2 的核心配置的表现形式、元素构成和运行机理。

3.4.1　Struts2 配置详解

配置就像是程序的影子，与程序总是如影随形。无论是什么框架，配置总是作为一个重要的组成部分，在框架的运行过程中发挥作用。同时，配置所起的不同作用在一定程度上也决定了配置的存在形式。

3.4.1.1　配置概览

Struts2 提供了多种可选的配置文件形式。根据这些配置文件的名称、所在位置、作用范围和用途我们制作了一张图表，如表 3-6 所示。

表 3-6　Struts2 配置文件的表现形式

配置文件	所在位置	作用范围	用途
web.xml	/WEB-INF/	应用级	Struts2 的入口程序定义、运行参数定义
struts-default.xml	/WEB-INF/lib/ struts2-core.jar !struts-default.xml	框架级	Struts2 默认的框架级的配置定义 包含所有 Struts2 的基本构成元素定义
struts.xml	/WEB-INF/classes/	应用级	Struts2 默认的应用级的主配置文件 包含所有应用级别对框架级别默认行为的覆盖定义
default.properties	/WEB-INF/lib/ struts2-core.jar !org.apache.struts2. default.properties	框架级	Struts2 默认的框架级的运行参数配置
struts.properties	/WEB-INF/classes/	应用级	Struts2 默认的应用级运行参数配置 包含所有应用级别对框架级别的运行参数的覆盖定义

（续）

配置文件	所在位置	作用范围	用途
struts-plugin.xml	插件所在 JAR 文件的根目录	应用级	Struts2 所支持的插件形式的配置文件，文件结构与 Struts.xml 一致。其定义作为 struts.xml 的扩展，也可以覆盖框架级别的行为定义

表 3-6 就是 Struts2 中配置文件的一个概览。可以看到在表头部分，我们分别使用了配置文件、所在位置、作用范围、用途来对配置文件进行大致的描述。接下来，我们分别从不同的角度来深入挖掘一下隐藏在配置文件背后的秘密。

3.4.1.2 配置的表现形式

从表 3-6 的第一列中，我们可以看到 Struts2 所支持的所有配置形式：其中既有 XML 文件形式，也有 Properties 文件形式。这些不同形式的配置文件，它们格式不同、所在位置不同、作用范围也不同。那么这些配置文件有没有一个主心骨呢？答案是肯定的。

结论 *从形式上讲，Struts2 的配置元素的表现形式以 XML 为核心，而 Properties 文件则作为另外一种配置形式起到辅助作用。*

事实上，除了 XML 和 Properties 文件的形式，Struts2 对于配置的构成形式并没有一个明确而死板的规定，因而我们可以在很多 Struts2 的扩展中见到类似 Annotation 或者“约定大于配置”的配置形式。有关这一点，我们在后续的章节会详细展开。

3.4.1.3 配置的作用范围

在表 3-6 中，我们可以看到不同的配置文件有着不同的作用范围。

在这其中，struts-default.xml 和 default.properties 是框架级别的配置文件。这两个文件蕴含在 Struts2 的核心 JAR 包之中，它们将在应用程序启动时被 Struts2 的初始化程序读取并加载。

在应用级别，Struts2 提供了 2 个与框架级别的配置文件相对应的配置文件：struts.xml 和 struts.properties。它们的结构与框架级别的配置文件完全相同，但是其中定义的所有内容将覆盖框架级别的配置定义，从而为程序员提供进行应用级别配置扩展的基本方法。

除此之外，我们还可以通过 Struts2 的插件来进行应用级别的配置定义，这一配置定义在插件所在 JAR 包的根目录，并以 struts-plugin.xml 的文件名形式出现。这种插件形式的配置定义则从另外一个角度为程序员提供了足够多的配置扩展层次。

配置的作用范围从一定程度上应该与配置文件的加载顺序结合起来看。有兴趣的读者可以在本书的第 12 章找到相关内容。

3.4.1.4　配置文件的必要性

在配置文件的这些特性中，有一点值得我们注意：所有我们提到的应用级别的配置文件，**都不是必须存在的。**

因为在默认情况下，Struts2 框架级别的配置文件已经足以支撑起一个 Struts2 的应用了。因而，在表 3-6 所列出的应用级别的配置文件中，**只有 web.xml 中的配置是必需的。**正如前面介绍的，因为我们需要在 web.xml 中定义 Struts2 的入口程序。这也是驱动整个 Struts2 的两条主线和两个阶段运行的核心所在。缺了 web.xml 的配置，Struts2 本身也就无从谈起了。

3.4.1.5　配置元素的逻辑意义

虽然 XML 配置文件和 Properties 配置文件都是 Struts2 所支持的配置文件形式，但是从内容上说，XML 包含了所有 Struts2 内置对象的定义、运行参数的定义、结构化配置定义、事件响应映射关系定义等所有 Struts2 运行必不可少的运行元素；而 Properties 文件呢，主要用于指定 Struts2 的运行参数。因此，我们可以得出一个结论：

结论　Struts2 框架中的 XML 文件的配置元素定义是 Properties 文件的配置元素定义的超集。

也就是说，凡是能够在 Properties 文件中定义的配置元素，我们都可以在 XML 中找到相应的配置方式代替，反之则不成立。因而，**要研究 Struts2 的微观构成元素，我们可以从分析 Struts2 的 XML 配置文件的元素入手。**

3.4.2　Struts2 配置元素定义

根据 3.4.1 节中我们对 Struts2 配置元素的分析所得到的结论，对 Struts2 所有配置元素的研究，可以从位于 Struts2 核心 JAR 包中的 struts-default.xml 文件入手，struts-default.xml 部分源码如代码清单 3-2 所示。

代码清单 3-2　struts-default.xml

```
    <!DOCTYPE struts PUBLIC
        "-//Apache Software Foundation//DTD Struts Configuration 2.1.7//EN"
        "http://struts.apache.org/dtds/struts-2.1.7.dtd">

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

    // 这里省略了许多 bean 定义
```

```
<constant name="struts.multipart.handler" value="jakarta" />

    <package name="struts-default" abstract="true">
        <result-types>
          <result-type name="chain"
class="com.opensymphony.xwork2.ActionChainResult"/>
            <result-type name="dispatcher"
class="org.apache.struts2.dispatcher.ServletDispatcherResult"
default="true"/>

        // 这里省略了许多 result-type 定义

        </result-types>

        <interceptors>
            <interceptor name="alias"
class="com.opensymphony.xwork2.interceptor.AliasInterceptor"/>
            <interceptor name="autowiring"
class="com.opensymphony.xwork2.spring.interceptor.
ActionAutowiringInterceptor"/>
            <interceptor name="chain"
class="com.opensymphony.xwork2.interceptor.ChainingInterceptor"/>

            // 这里省略了许多 interceptor 定义

            <!-- Basic stack -->
            <interceptor-stack name="basicStack">
                <interceptor-ref name="exception"/>
                <interceptor-ref name="servletConfig"/>
                <interceptor-ref name="prepare"/>
                <interceptor-ref name="checkbox"/>
                <interceptor-ref name="multiselect"/>
                <interceptor-ref name="actionMappingParams"/>
                <interceptor-ref name="params">
                    <param
name="excludeParams">dojo\..*,^struts\..*</param>
                </interceptor-ref>
                <interceptor-ref name="conversionError"/>
            </interceptor-stack>

            // 这里省略了许多 interceptor-stack 定义

       </interceptors>

        <default-interceptor-ref name="defaultStack"/>

        <default-class-ref class="com.opensymphony.xwork2.ActionSupport" />
    </package>
```

```
</struts>
```

我们在这里为了显示 XML 配置文件的结构，省略了其中某些具体节点的定义。不过这丝毫不影响我们对 XML 配置中的一些主要节点进行研究。

3.4.2.1　include 节点

include 节点本身并没有在 struts-default.xml 中出现，但它却是 Struts2 配置文件所支持的第一层根节点之一。include 节点的主要作用是帮助我们管理 Struts2 的配置文件，实现配置文件的模块化。

include 节点最为常见的使用方法，是我们可以在应用级别的配置文件（struts.xml）里，将整个应用的配置根据一定的逻辑划分成若干个独立的配置文件，以方便进行模块化管理和团队开发。如代码清单 3-3 所示是一个典型的例子。

代码清单 3-3　struts.xml

```
<?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>

   <include file="web/struts-config.xml" />
   <include file="web/struts-system.xml" />
   <include file="web/struts-user.xml" />

</struts>
```

从上面的例子中，我们可以看到 Struts2 进行配置模块化的管理方式：利用结构完全相同的文件，通过 include 节点把这些文件串联起来。这是一个非常直观有效的管理方式，这种模式有点类似于 Java 语法中的“对象引用嵌套”。其最大的好处就在于方便理解，并且能够支持团队化开发，团队成员只需要关心他自身所工作的模块，从而避免了配置层面的冲突发生。

另外一种进行配置模块化管理的方式是使用“继承”机制。这种机制我们会在之后对 package 节点的分析中有所涉及，读者可以细细体会它们两者的不同之处。

3.4.2.2　bean 节点

bean 节点在 struts-default.xml 中广泛用于定义 Struts2 框架级别的**内置对象**。我们可以从 bean 节点的属性定义中发现这个节点实际上是**一个用于描述接口及其实现类映射关系的节点**。

在逻辑关系上，**bean 节点的寻址是通过 name 属性和 type 属性共同构成一个逻辑主键来共同决定一个 class 属性**。也就是说，我们可以通过 name 属性和 type 属性的值来控制一个接口的不同实现方式，这也是 Struts2 用来切换程序运行机制的最基本方式。

这些 bean 节点在 Struts2 内部是以何种形式存在的呢？ Struts2 在框架级别实现了一个对象容器，并将配置文件中所有 bean 节点所定义的对象纳入容器之中进行管理。Struts2 通过这个容器在框架级别负责这些对象的创建、销毁以及依赖关系的处理。熟悉 Spring 框架的读者会发现，在这里，这个容器的概念与 Spring 中容器的概念非常类似，都是用来管理程序运行过程中对象的生命周期的。Struts2 不仅实现了一个容器，并且还在容器的基础之上实现了依赖注入（IoC），从而使得所有 Struts2 中所定义的对象都可以被有条不紊地创建、执行和销毁。

在应用级别的配置文件中，通过增加 bean 节点，就可以把一个对象纳入 Struts2 的容器中进行管理。此时，我们可以通过 Struts2 提供的容器访问接口（原生 API 方式）或者依赖注入的方式，获取我们自定义的这个 bean 对象并使用。

3.4.2.3 constant 节点

constant 节点从结构上看是一个非常典型的键值对类型的配置，主要用于定义 Struts2 运行时的参数。在 struts-default.xml 中，我们很少看到 constant 节点的定义，因为 Struts2 主要使用了 Properties 文件来定义运行时的参数而并非将它们放在 XML 中。这实际上符合了**“让最合适的表现形式来表达最合适的配置语义”**的设计原则。

constant 节点的作用与 Struts2 的 Properties 文件的配置形式是完全重合的。这一点有助于我们理解 Struts2 所支持的 XML 配置形式实际上是 Properties 文件配置形式的超集这一结论。因而，constant 节点也成为两种配置形式在逻辑上的交集。

constant 节点中所有运行时的参数定义与 bean 节点一样，也会在系统初始化时被加载到 Struts2 的容器中进行统一管理。也就是说，Struts2 的容器不仅仅负责对 Struts2 所有内置对象的管理，还要负责对系统的运行参数进行管理。Struts2 将两者进行统一的主要好处在于 Struts2 在内部对与这些框架相关的运行对象或者运行参数可以一视同仁地进行处理，通过统一的容器访问接口（或者依赖注入的方式），Struts2 可以从容器中方便地组织起整个程序的架构。这种统一性可大大降低编程的复杂度，使 Struts2 的运行机制更为顺畅。

3.4.2.4 package 节点

package 节点是一个复杂的复合节点，在其中包含了众多子节点：ResultType、Interceptor、InterceptorStack、Action，等等。实际上，这些子节点全部继承自 XWork 框架。还记得第 2 章中我们对 XWork 框架的比喻吗？ XWork 就像一条生产流水线，定义了一系列事件执行的步骤和次序。一个 package 节点实际上可以被看作是一条简单的 XWork

生产流水线，其中包含 XWork 框架如何应对某些请求并选择相应的执行序列进行处理的具体方式。其中的 ResultType、Interceptor 子节点和 Action 子节点也正是我们之前提到过的 XWork 中最为重要的微观元素，也是构成事件执行序列的主要元素。

从功能上分析，package 节点与之前的 bean 节点和 constant 节点有着本质的区别。无论是 bean 节点还是 constant 节点，它们都**与框架自身的运行状态有关，并且被 Struts2 内部的容器所管理**。而 package 节点的作用却是定义一种映射关系，更多反映了框架如何与外部程序进行交互的过程。我们将在后续章节展开分析这些节点的实际作用，读者在这里只需要大致了解 package 节点的构成要素即可。

在 package 节点的属性之中，name 是一个唯一的标识符，namespace 则从命名空间的角度为整个事件请求机制划分不同的种类。在运行期，我们可以认为 name 属性和 namespace 属性都用于对请求进行逻辑划分。

package 节点中的 extends 属性允许 package 和 package 之间形成相应的继承关系。通过继承，子 package 自动获得父 package 的所有配置定义。extends 属性则从另外一个角度，允许开发人员进行配置的模块化管理，抽取公共的配置定义成为一个公共的父 package，子 package 可以任意引用父 package 中的任何配置定义，这极大地简化了配置的工作量。读者在这里可以体会“继承”和“引用”这两种不同的扩展模式在 Struts2 配置文件中的应用。

3.4.3　Struts2 配置元素的分类

在对 struts-default.xml 的分析中，我们实际上已经提供了在逻辑上对 Struts2 中的配置元素进行分类的思路：**从节点所表达的逻辑含义和节点在程序中所起的作用对配置元素进行分类**。

XML 配置文件中的 bean 节点和 constant 节点，它们其中一个是构成程序运行的对象，而另一个用于指定程序运行的执行参数。它们都与 Struts2 自身的运行机制有关，并且有一个统一的管理机制，将它们归为同一类的配置元素应该毫无异议。习惯上，我们把这类配置元素称之为“**容器配置元素**”。有关这个命名的具体由来，我们将在之后的章节详细解释。

XML 配置文件中的 package 节点定义了一种事件请求响应的映射关系，反映的是 Struts2 对于外部事件请求时如何进行响应的处理序列，它与 bean 节点和 constant 节点在整个框架中起的作用完全不同，是自成体系的另一类配置元素。对于这一类配置元素，我们通常称之为“**事件映射关系**”。

在这里我们需要强调的是，Struts2 的配置元素的定义体系与 XWork 框架是一脉相承的。无论是节点定义还是节点的逻辑关系，Struts2 都直接从 XWork 框架继承而来。

明确 Struts2 对于配置元素的分类，可为整个框架进行配置元素的对象化打下坚实的基础。在之后的章节中，我们可以看到 Struts2 针对不同的配置元素，定义了与之对应的数据结构和处理方法。

3.5 小结

在本章中，我们从 Struts2 的历史谈起，对 Struts2 的外部环境和内部构成机理都做了简要的分析，目的是使读者对 Struts2 框架有个初步的认识。其中，对 Struts2 的外部环境的介绍有助于使读者从技术的角度了解 Struts2 的适用范围及与其依赖的项目之间的联系。而对 Struts2 内部运行主线和构成元素的剖析则可帮助读者对 Struts2 自身的架构有所掌握。

本章的内容对于全书有着提纲挈领的作用，尤其是对 Struts2 运行的逻辑主线的分析，将成为我们研究 Struts2 内在机理的切入点。因而，本章中的许多概念和结论会在之后的分析讲解中被反复提及。读者应谨记这些结论，并与之后的章节对应起来，体会 Struts2 的总体架构。

回顾本章，大家可以重新思考下面的这些问题，是否在本章中得到了答案呢？

- ❑ Struts2 和 Struts1.X 有什么区别？它们各自的发展轨迹又如何？
- ❑ Struts2 依赖于哪些核心技术？
- ❑ Struts2 可以应用在什么样的项目中？
- ❑ Struts2 可以分成哪两条逻辑运行主线？
- ❑ Struts2 在处理 Http 请求时，可以分成哪两个主要阶段？
- ❑ Struts2 通过哪些元素的相互配合来完成初始化运行主线？
- ❑ XWork 框架主要由哪些元素构成？它们之间有什么关系？
- ❑ Struts2 有哪些配置表现形式？
- ❑ Struts2 中的配置元素可以分为哪两个大类？
- ❑ 什么是配置元素的对象化过程？
- ❑ Struts2 的配置元素的对象化过程由哪两大元素配合完成？

第二部分

核心技术篇

Struts2 作为一个优秀的 Web 框架，它自身充满了最佳实践。这也是本书将它作为研究课题的一个重要原因：希望通过对这些框架中最佳实践的挖掘，找到进行 Web 开发的最佳途径。在之前的分析中我们已经知道，Struts2 并不是孤立存在的。Struts2 的运行，依赖于众多其他的核心技术。如果说 Struts2 是一个有血有肉的人，那么这些核心技术实际上构成了人的骨架。

我们曾经在本书的第 2 章中介绍过 Web 开发中应该遵循的一些最佳实践。然而这些内容不应该刻板地表现为一条一条的规则，而应该成为源代码中的一个个闪光点。因此，在本书的第 4 章，我们就来重点总结这些闪光点：Struts2 所使用的一些设计模式。设计模式本身是一门学问，自成体系，因而它并不是 Struts2 运行时所必须依赖的核心技术。而我们在这里将设计模式单独罗列出来，并作为介绍 Struts2 的一个重要方面，是希望通过对这些设计模式的介绍，让读者能够更加了解 Struts2 在设计理念上的一些独特之处。

如果说对于设计模式的使用，是 Struts2 自身充满最佳实践的佐证。那么从第 5 章起，我们就开始真正对 Struts2 的运行所依赖的核心技术中的 XWork 框架进行讲解。我们首先要解决的是对象生命周期的管理问题。这是面向对象的概念本身所带来的一个核心问题，也是每个框架不得不去面对和解决的问题。在第 5 章中，我们将重点介绍 Struts2 / XWork 中的对象管理容器及其实现细节。容器，不仅支撑起一个框架中的所有对象，同时也成为框架运行过程中最为重要的一个辅助元素，也是整个框架得以运行的核心基础。

本书的第 6 章，我们所涉及的话题是 Struts2 运行所依赖的另外一个重要的核心技术：OGNL 表达式引擎。表达式引擎是不同编程层次之间进行数据沟通的重要桥梁，因而也是一个优秀的 Web 框架必不可少的组成要素。在第 6 章中，我们将以 Struts2 所依赖的

OGNL 作为研究表达式引擎的重要窗口。通过对表达式引擎内部结构和运作机制的研究，帮助读者体会它在 Web 开发中的重要作用。

XWork 作为 Struts2 的核心依赖，其重要性不言而喻。然而，为了讲清楚 XWork 的设计理念和初衷，我们还是不得不以请求－响应模式的实现机理为切入口。在第 7 章中，我们将有大量的篇幅涉及请求－响应这一基本的人机交互模式的讨论，并在此基础上提出控制流和数据流这两股隐藏在程序运行过程之中的重要力量。事实上，XWork 框架正是基于控制流和数据流来进行体系结构设计的。对此，我们将从宏观视图和微观构成这两个不同的角度进行详细阐述。读者会发现，我们在整个章节中，始终围绕着“请求”和“响应”的基本概念而非具体的编程实现。因为理清概念，逐步推理始终是本书的一个重要宗旨。

第 8 章的内容则承接第 7 章对 XWork 设计理念的阐述，继续针对 XWork 各个构成要素进行逐一分析。通过第 8 章的阅读，读者会发现，对于这些构成元素的详细解读，无论是控制流元素还是数据流元素，都不是孤立的。我们的逻辑主线总是从 XWork 的宏观视图着手，努力探寻蕴含在元素和元素之间的联系。而了解 XWork 的运行机制以及构成 XWork 的各个元素的细节，也是本书所要解决的核心问题之一。

本篇的内容繁多，然而所有的内容都没有涉及 Struts2 自身的运行机理。这一奇怪的现象实际上和 Struts2 本身的核心设计理念有极大的关系。我们可以发现，无论是对设计模式的研究，还是对 XWork 框架或者 OGNL 的研究，实际上都只是对 Struts2“围而不打”。因为我们一直坚信，概念永远大于方法、核心驱动力永远大于外在表现。因此，在阅读本篇的过程中，读者应该始终去把握的是书中所倡导的思考方法、设计理念。因为只有这些，才是程序的根本。

第 4 章　源头活水——Struts2 中的设计模式

设计模式（Design pattern）是经过程序员反复实践后形成的一套代码设计经验的总结。随着编程语言的发展，设计模式由最初的“编程惯例”逐步发展成为被反复使用并为绝大多数程序员所知晓的、完善的理论体系。我们使用设计模式的初衷，是使代码的重用度提高、让代码能够更容易被别人理解以及保证代码的可靠性。毫无疑问，在程序中使用设计模式无论是对于程序员自身还是对于应用程序都是双赢的结果。正确地使用设计模式，能够使编程真正实现工程化和规范化，并在一定程度上指导框架的设计和实现。

在深入探讨 Struts2 所依赖的核心技术之前，我们将首先带领读者领略一下在整个 Struts2 框架之中所使用到的一些最常用的设计模式。理解这些设计模式的运用场景和内部机理，将为日后我们对这些核心技术的分析打下坚实的基础。

4.1　ThreadLocal 模式

ThreadLocal 模式，严格意义上来说并不能称之为一种设计模式，因为它只是用来解决多线程程序中数据共享问题的一个方案。尽管如此，ThreadLocal 模式却贯穿了整个 Struts2 和 XWork 框架，成为 Struts2 框架进行“解耦”设计的核心依赖技术。那么，为什么要在 Struts2 中引入 ThreadLocal 模式呢？这不得不从 Web 开发中的线程安全问题谈起。

4.1.1　线程安全问题的由来

在传统的 Web 开发中，我们处理 Http 请求最常用的方式是通过实现 Servlet 对象来进行 Http 请求的响应。Servlet 是 J2EE 的重要标准之一，规定了 Java 如何响应 Http 请求的规范。通过 HttpServletRequest 和 HttpServletResponse 对象，我们能够轻松地与 Web 容器交互。

当 Web 容器收到一个 Http 请求时，Web 容器中的一个主调度线程会从事先定义好的线程池中分配一个当前工作线程，将请求分配给当前的工作线程，由该线程来执行对应的 Servlet 对象中的 service 方法。当这个工作线程正在执行的时候，Web 容器收到另外一个请求，主调度线程会同样从线程池中选择另一个工作线程来服务新的请求。Web 容器本身并不关心这个新的请求是否访问的是同一个 Servlet 实例。因此，我们可以得出一个结论：**对于同一个 Servlet 对象的多个请求，Servlet 的 service 方法将在一个多线程的环境中并**

发执行。所以，Web 容器默认采用**单实例（单 Servlet 实例）多线程**的方式来处理 Http 请求。这种处理方式能够减少新建 Servlet 实例的开销，从而缩短了对 Http 请求的响应时间。但是，这样的处理方式会导致变量访问的线程安全问题。也就是说，Servlet 对象并不是一个线程安全的对象。下面的测试代码将证实这一点，如代码清单 4-1 所示。

代码清单 4-1 ThreadSafeTestServlet.java

```
public class ThreadSafeTestServlet extends HttpServlet {
  // 定义一个实例变量，并非一个线程安全的变量
  private int counter = 0;

   public void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        doPost(req, resp);
  }

        public void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
     // 输出当前 Servlet 的信息以及当前线程的信息
       System.out.println(this + ":" + Thread.currentThread());
     // 循环，并增加实例变量 counter 的值
     for (int i = 0; i < 5; i++) {
          System.out.println("Counter = " + counter);
          try {
              Thread.sleep((long) Math.random() * 1000);
              counter++;
          } catch (InterruptedException exc) {
          }
     }
   }
}
```

这里参考了网络上一段著名的对 Servlet 线程安全性进行测试的代码[⊖]。运行之后，我们可以看一下这个例子的输出：

```
sample.SimpleServlet@11e1bbf:Thread[http-8081-Processor23,5,main]
Counter = 60
Counter = 61
Counter = 62
Counter = 65
Counter = 68
Counter = 71
Counter = 74
Counter = 77
Counter = 80
Counter = 83
```

⊖ 原文见 http://zwchen.iteye.com/blog/91088。

```
sample.SimpleServlet@11e1bbf:Thread[http-8081-Processor22,5,main]
Counter = 61
Counter = 63
Counter = 66
Counter = 69
Counter = 72
Counter = 75
Counter = 78
Counter = 81
Counter = 84
Counter = 87

sample.SimpleServlet@11e1bbf:Thread[http-8081-Processor24,5,main]
Counter = 61
Counter = 64
Counter = 67
Counter = 70
Counter = 73
Counter = 76
Counter = 79
Counter = 82
Counter = 85
Counter = 88
```

通过上面的输出，我们可以得出以下三个 Servlet 对象的运行特性：

- Servlet 对象是一个无状态的单例对象（Singleton），因为我们看到多次请求的 this 指针所打印出来的 hashcode 值都相同。
- Servlet 在不同的线程（线程池）中运行，如 http-8081-Processor22 和 http-8081-Processor23 等输出值可以明显区分出不同的线程执行了同一段 Servlet 逻辑代码。
- Counter 变量在不同的线程中共享，而且它的值被不同的线程修改，输出时已经不是顺序输出。也就是说，其他的线程会篡改当前线程中实例变量的值，针对这些对象的访问不是线程安全的。

有关线程安全的概念范畴

谈到线程安全，许多初学者很容易在概念上混淆。线程安全，指的是在多线程环境下，一个类在执行某个方法时，对类的内部实例变量的访问是安全的。因此，对于下面列出来的 2 类变量，不存在任何线程安全的说法：

1）方法签名中的任何参数变量。

2）处于方法内部的局部变量。

任何针对上述形式的变量的访问都是线程安全的，因为它们都处于方法体的内部，由当前的执行线程独自管理。

这就是线程安全问题的由来：**在传统的基于 Servlet 的开发模式中，Servlet 对象内部的实例变量不是线程安全的。**在多线程环境中，这些变量的访问需要通过特殊的手段进行访问控制。

解决线程安全访问的方法很多，比较容易想到的一种方案是使用同步机制，但是出于对 Web 应用效率的考虑，这种机制在 Web 开发中的可行性很低，也违背了 Servlet 的设计初衷。因此，我们需要另辟蹊径来解决这一困扰我们的问题。

4.1.2 ThreadLocal 模式的实现机理

在 JDK 的早期版本中，提供了一种解决多线程并发问题的方案：java.lang.ThreadLocal 类。ThreadLocal 类在维护变量时，实际使用了当前线程（Thread）中的一个叫做 ThreadLocalMap 的独立副本，每个线程可以独立修改属于自己的副本而不会互相影响，从而隔离了线程和线程，避免了线程访问实例变量发生冲突的问题。

ThreadLocal 本身并不是一个线程，而是通过操作当前线程中的一个内部变量来达到与其他线程隔离的目的。之所以取名为 ThreadLocal，所期望表达的含义是其操作的对象是线程的一个本地变量。如果我们看一下 Thread 的源码实现，就会发现这一变量，如代码清单 4-2 所示。

代码清单 4-2 Thread.java

```
public class Thread implements Runnable {
    // 这里省略了许多其他的代码
    ThreadLocal.ThreadLocalMap threadLocals = null;
}
```

这是 JDK 中 Thread 源码的一部分，从中我们可以看出 ThreadLocalMap 跟随着当前的线程而存在。不同的线程 Thread，拥有不同的 ThreadLocalMap 的本地实例变量，这也就是“副本”的含义。接下来我们再来看看 ThreadLocal.ThreadLocalMap 是如何定义的，以及 ThreadLocal 如何来操作它，如代码清单 4-3 所示。

代码清单 4-3 ThreadLocal.java

```
public class ThreadLocal<T> {

  // 这里省略了许多其他代码

  // 将 value 的值保存于当前线程的本地变量中
  public void set(T value) {
    // 获取当前线程
    Thread t = Thread.currentThread();
```

```
    // 调用 getMap 方法获得当前线程中的本地变量 ThreadLocalMap
    ThreadLocalMap map = getMap(t);
    // 如果 ThreadLocalMap 已存在，直接使用
    if (map != null)
         // 以当前的 ThreadLocal 的实例作为 key，存储于当前线程的
         // ThreadLocalMap 中，如果当前线程中定义了多个不同的 ThreadLocal
         // 的实例，则它们会作为不同 key 进行存储而不会互相干扰
         map.set(this, value);
    else
         // 如果 ThreadLocalMap 不存在，则为当前线程创建一个新的
         createMap(t, value);
}

 // 获取当前线程中以当前 ThreadLocal 实例为 key 的变量值
public T get() {
    // 获取当前线程
    Thread t = Thread.currentThread();
    // 获取当前线程中的 ThreadLocalMap
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        // 获取当前线程中以当前 ThreadLocal 实例为 key 的变量值
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null)
            return (T)e.value;
    }
    // 当 map 不存在时，设置初始值
    return setInitialValue();
    }

    // 从当前线程中获取与之对应的 ThreadLocalMap
    ThreadLocalMap getMap(Thread t) {
        return t.threadLocals;
    }

    // 创建当前线程中的 ThreadLocalMap
    void createMap(Thread t, T firstValue) {
        // 调用构造函数生成当前线程中的 ThreadLocalMap
        t.threadLocals = new ThreadLocalMap(this, firstValue);
    }

    // ThreadLoaclMap 的定义
    static class ThreadLocalMap {
        // 这里省略了许多代码
    }
}
```

从上述代码中，我们看到了 ThreadLocal 类的大致结构和进行 ThreadLocalMap 的操作。我们可以从中得出以下的结论：

- ThreadLocalMap 变量属于线程的内部属性，不同的线程拥有完全不同的 ThreadLocalMap 变量。
- 线程中的 ThreadLocalMap 变量的值是在 ThreadLocal 对象进行 set 或者 get 操作时创建的。
- 在创建 ThreadLocalMap 之前，会首先检查当前线程中的 ThreadLocalMap 变量是否已经存在，如果不存在则创建一个；如果已经存在，则使用当前线程已创建的 ThreadLocalMap。
- 使用当前线程的 ThreadLocalMap 的关键在于使用当前的 ThreadLocal 的实例作为 key 进行存储。

ThreadLocal 模式至少从两个方面完成了数据访问隔离，即横向隔离和纵向隔离。有了横向和纵向两种不同的隔离方式，ThreadLocal 模式就能真正地做到线程安全：

- **纵向隔离**——线程与线程之间的数据访问隔离。这一点由线程的数据结构保证。因为每个线程在进行对象访问时，访问的都是各个线程自己的 ThreadLocalMap。
- **横向隔离**——同一个线程中，不同的 ThreadLocal 实例操作的对象之间相互隔离。这一点由 ThreadLocalMap 在存储时采用当前 ThreadLocal 的实例作为 key 来保证。

ThreadLocal 模式并不是什么高深的学问，它甚至从 JDK 1.2 开始就存在于 Java 世界中。由此可见，我们掌握一种知识的最终目的是熟练而合理地运用它。

深入比较 ThreadLocal 模式与 synchronized 关键字

ThreadLocal 模式与 synchronized 关键字都用于处理多线程并发访问变量的问题，只是二者处理问题的角度和思路不同。

1）ThreadLocal 是一个 Java 类，通过对当前线程中的局部变量的操作来解决不同线程的变量访问的冲突问题。所以，ThreadLocal 提供了线程安全的共享对象机制，每个线程都拥有其副本。

2）Java 中的 synchronized 是一个保留字，它依靠 JVM 的锁机制来实现临界区的函数或者变量在访问中的原子性。在同步机制中，通过对象的锁机制保证同一时间只有一个线程访问变量。此时，被用作“锁机制”的变量是多个线程共享的。

同步机制（synchronized 关键字）采用了“以时间换空间”的方式，提供一份变量，让不同的线程排队访问。而 ThreadLocal 采用了“以空间换时间”的方式，为每一个线程都提供一份变量的副本，从而实现同时访问而互不影响。

4.1.3　ThreadLocal 模式的应用场景

在分析了 ThreadLocal 的源码之后，我们来看看 ThreadLocal 模式最合适的业务场景。在一个完整的“请求 – 响应”过程中，主线程的执行过程总是贯穿始终。当这个主线程的执行过程中加入 ThreadLocal 的读写时，会对整个过程产生怎样的影响呢？根据之前源码分析的结果，并结合分层开发模式，把整个流程画下来，如图 4-1 所示。

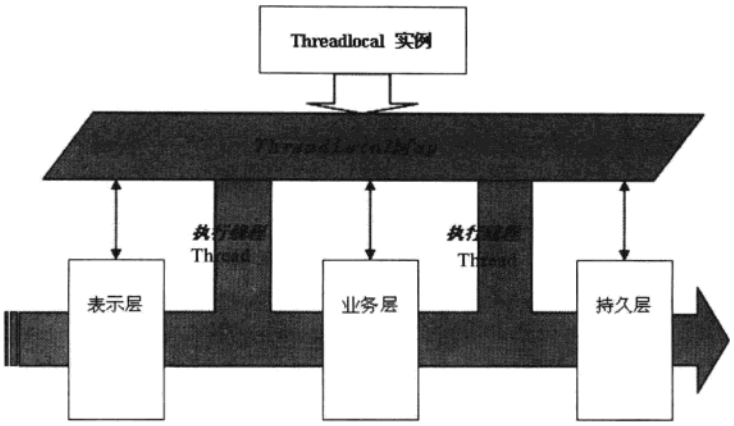


图 4-1　Thread 执行示意图

从图中我们可以看到，由于 ThreadLocal 所操作的是维持于整个 Thread 生命周期的副本（ThreadLocalMap），所以无论在 J2EE 程序的哪个层次（表示层、业务层或者持久层），只要在一个 Thread 的生命周期之内，存储于 ThreadLocalMap 中的对象都是线程安全的（因为 ThreadLocalMap 本身仅仅隶属于当前的执行线程，是执行线程内部的一个属性变量。我们用图中的阴影部分来表示这个变量的存储空间）。而这一点，正是我们用来解决多线程环境中的变量共享问题的核心技术。ThreadLocal 的这一特性也使其能够广泛应用于 J2EE 开发中的许多业务场景。

数据共享，还是数据传递

ThreadLocal 模式由于利用了 Java 自身的语法特性而显得异常简单和方便，因而被广泛应用于 J2EE 开发，尤其是应对跨层次的资源共享，例如在 Spring 中，就有使用 ThreadLocal 模式来管理数据库连接或者 Hibernate 的 Session 的范例。

在一些比较著名的论坛中，有很多关于使用 ThreadLocal 模式来做数据传递的讨论。事实上，这是对 ThreadLocal 模式的一个极大的误解。需要注意的是，ThreadLocal 模式解决的是同一线程中隶属于不同开发层次的数据共享问题，而不是在

不同的开发层次中进行数据传递。

1）ThreadLocal 模式的核心在于实现一个共享环境（类的内部封装了 ThreadLocal 的静态实例）。所以，在操作 ThreadLocal 时，这一共享环境会跨越多个开发层次而随处存在。

2）随处存在的共享环境造成了所有开发层次的共同依赖，从而使得所有的开发层次都耦合在了一起，从而变得无法独立测试。

3）数据传递应该通过接口函数的签名显式声明，这样才能够从接口声明中表达接口所表达的真正含义。ThreadLocal 模式位于实现的内部，从而使得接口与接口之间无法达成一致的声明契约。

Struts2 的解耦合设计理念使得 Struts2 的 MVC 实现成为使用 ThreadLocal 模式的天然场所。在第 3 章中，我们已经介绍了一些基本概念，Struts2 通过引入 XWork 框架，将整个 Http 请求的过程拆分成与 Web 容器有关和与 Web 容器无关的两个执行阶段。而这两个阶段的数据交互就是通过 ThreadLocal 模式中的线程共享副本安全地进行。在其中，我们没有看到数据传递，存在的只是整个执行线程的数据共享。

4.1.4 ThreadLocal 模式的核心元素

仔细分析图 4-1，我们可以发现，要完成 ThreadLocal 模式，其中最关键的地方就是**创建一个任何地方都可以访问到的 ThreadLocal 实例（也就是执行示意图中的菱形部分）**。而这一点，我们可以通过类的静态实例变量来实现，这个用于承载静态实例变量的类就被视作是一个共享环境。我们来看一个例子，如代码清单 4-4 所示。

代码清单 4-4 Counter.java

```
public class Counter {

  // 新建一个静态的 ThreadLocal 变量，并通过 get 方法将其变为一个可访问的对象
  private static ThreadLocal<Integer> counterContext = new
ThreadLocal<Integer>() {
    protected synchronized Integer initialValue() {
            return 10;
    }
  };

  // 通过静态的 get 方法访问 ThreadLocal 中存储的值
  public static Integer get() {
   return counterContext.get();
  }
```

```
    // 通过静态的 set 方法将变量值设置到 ThreadLocal 中
    public static void set(Integer value) {
     counterContext.set(value);
    }

    // 封装业务逻辑，操作存储于 ThreadLocal 中的变量
    public static Integer getNextCounter() {
      counterContext.set(counterContext.get() + 1);
      return counterContext.get();
    }

}
```

在这个 Counter 类中，我们实现了一个静态的 ThreadLocal 变量，并通过 get 方法将 ThreadLocal 中存储的值暴露出来。我们还封装了一个带有业务逻辑的方法 getNextCounter，操作 ThreadLocal 中的值，将其加 1，并返回计算后的值。

此时，Counter 类就变成了一个数据共享环境，我们也拥有了实现 ThreadLocal 模式的关键要素。有了它，我们来编写一个简单的测试，如代码清单 4-5 所示。

代码清单 4-5　ThreadLocalTest.java

```
public class ThreadLocalTest extends Thread {

  public void run() {
     for(int i = 0; i < 3; i++){
    System.out.println("Thread[" + Thread.currentThread().getName() +
"],counter=" + Counter.getNextCounter());
    }
  }
}
```

这是一个简单的线程类，循环输出当前线程的名称和 getNextCounter 的结果，由于 getNextCounter 中的逻辑所操作的是 ThreadLocal 中的变量，所以无论同时有多少个线程在运行，返回的值将仅与当前线程的变量值有关，也就是说，在同一个线程中，变量值会被连续累加。这一点可以通过如代码清单 4-6 所示的测试代码证实。

代码清单 4-6　Test.java

```
public class Test {

  public static void main(String[] args) throws Exception {

    ThreadLocalTest testThread1 = new ThreadLocalTest();
    ThreadLocalTest testThread2 = new ThreadLocalTest();
    ThreadLocalTest testThread3 = new ThreadLocalTest();
```

```
        testThread1.start();
        testThread2.start();
        testThread3.start();

    }
}
```

我们来运行一下上面的代码，并看看输出结果：

```
Thread[Thread-2],counter=11
Thread[Thread-2],counter=12
Thread[Thread-2],counter=13
Thread[Thread-0],counter=11
Thread[Thread-0],counter=12
Thread[Thread-0],counter=13
Thread[Thread-1],counter=11
Thread[Thread-1],counter=12
Thread[Thread-1],counter=13
```

上面的输出结果也证实了，counter 的值在多线程环境中的访问是线程安全的。从对例子的分析我们可以再次体会到，ThreadLocal 模式最合适的使用场景：**在同一个线程的不同开发层次中共享数据。**

从上面的例子中，我们可以简单总结出实现 ThreadLocal 模式的两个主要步骤：

- **建立一个类，并在其中封装一个静态的 ThreadLocal 变量，使其成为一个共享数据环境。**
- **在类中实现访问静态 ThreadLocal 变量的静态方法（设值和取值）。**

建立在 ThreadLocal 模式的实现步骤之上，ThreadLocal 的使用则更加简单。在线程执行的任何地方，我们都可以通过访问共享数据类中所提供的 ThreadLocal 变量的设值和取值方法安全地获得当前线程中安全的变量值。

这两个步骤我们之后会在 Struts2 的实现中多次提及，读者只要能充分理解 ThreadLocal 处理多线程访问的基本原理，就能对 Struts2 的数据访问和数据共享的设计有一个整体的认识。

讲到这里，我们回过头来看看 ThreadLocal 模式的引入，到底对我们的编程模型有什么重要的意义呢？

结论 *使用 ThreadLocal 模式，可以使数据在不同的编程层次得到有效共享。*

这一点是由 ThreadLocal 模式的实现机理决定的。因为实现 ThreadLocal 模式的一个重要步骤，就是构建一个静态的共享存储空间，从而使得任何对象在任何时刻都可以安全

地对数据进行访问。

结论　使用 ThreadLocal 模式，可以对执行逻辑与执行数据进行有效解耦。

这一点是 ThreadLocal 模式给我们带来的最为核心的一个影响。因为在一般情况下，Java 对象之间的协作关系，主要通过参数和返回值来进行消息传递，这也是对象协作之间的一个重要依赖。而 ThreadLocal 模式彻底打破了这种依赖关系，通过线程安全的共享对象来进行数据共享，可以有效避免在编程层次之间形成数据依赖。这也成为 XWork 事件处理体系设计的核心。

4.2　装饰（Decorator）模式

4.2.1　装饰模式的定义

装饰模式是对象的一种结构模式，其基本定义如下：

定义　装饰模式的基本含义是能够动态地为一个对象添加一些额外的行为职责。

谈到对象行为职责的扩展，我们很容易就能够想到面向对象编程语言的一个重要特征：继承。继承是绝大多数面向对象的编程语言在语法上的天然支持。通过使用继承，我们可以获得以下两种扩展特性：

- **现有对象行为的覆盖**——通过覆写（Override）父类中的已有方法完成
- **添加新的行为职责**——通过在子类中添加新的方法完成

既然我们已经有了天然的语法支持，那么为什么还要花力气构建一个设计模式来进行对象行为职责的扩展呢？因为“继承”这个语法，为对象类型所引入的是一种“静态”特性扩展。这种“静态”特性扩展的意思是说：我们必须编写一个子类，并在其中通过语法所支持的“函数覆盖”或者“函数添加”的方式扩展其行为特性。这一扩展后的行为特性的获取在“编译期”就被决定，而并非是一个“运行期”的扩展模式。随着子类的增多，虽然我们获得了更多的功能扩展，然而各种子类的组合（扩展功能的组合）将导致子类的极度膨胀。在 Java 语言中，一个类只能进行“单根继承”而无法支持“多重继承”，因而通过“继承”这种方式进行功能行为特性的扩展缺乏足够的灵活性。

装饰模式则正是为了解决“**过度依赖使用继承来进行对象的功能扩展**”这一问题而设计的。我们可以从装饰模式的文字定义中看到它作为一个设计模式的目的和特性：

- **目的**——进行对象行为职责扩展

❑ **特性**——动态（扩展特性在运行期自动获得）

在图 4-2 是装饰模式的一种基本实现示意图。

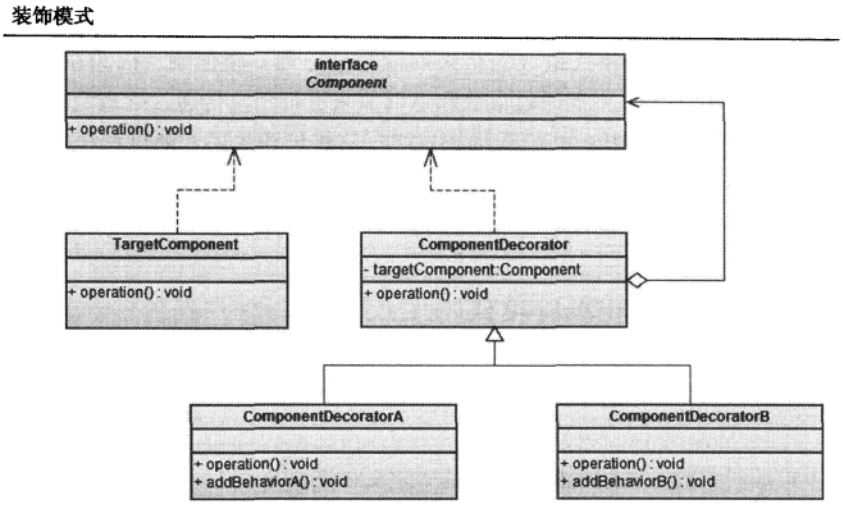


图 4-2 装饰模式的一种基本实现示意图

从图 4-2 中我们可以归纳出装饰模式的基本实现中所涉及的角色：

❑ **原始接口**（Component）——定义了一个接口方法。

❑ **默认目标实现类**（TargetComponent）——对于原始接口的默认实现方式。在装饰模式中，默认目标实现类被认为是有待扩展的类，其方法 operation 被认为是有待扩展的行为方法。

❑ **装饰实现类**（ComponentDecorator）——同样实现了原始接口，既可以是一个抽象类，也可以是一个具体实现类。其内部封装了一个原始接口的对象实例：targetComponent，这个实例的实现往往被初始化成默认目标实现类（TargetComponent）。

❑ **具体装饰实现类**（ComponentDecoratorA、ComponentDecoratorB）——继承自装饰类（ComponentDecorator），我们可以在 operation 方法中调用原始接口的对象实例 targetComponent 获得默认目标实现类的行为方式并在其中加入行为扩展实现，也可以自由添加新的行为职责 addBehaviorA 等。

4.2.2 装饰模式的构成要素

在图 4-2 中，我们给出了装饰模式的一种基本实现方式的类图表示，并以此为基础总

结了装饰模式的构成角色。从这些角色的调用关系来看，装饰模式之所以被命名为“装饰”，其要义主要有两点：

- **默认目标实现（TargetComponent）类封装于具体的装饰实现类（ComponentDecorator）或者其子类的内部，从而形成对象之间的引用关系**
- **具体装饰实现类（ComponentDecorator）同样实现了原始接口（Component）**

这样一来，当我们从外部调用者的角度来看待原始接口（Component）的默认目标实现（TargetComponent），就好像它被装饰了一番，如同穿了一件外衣，从而以装饰实现类（ComponentDecorator）的形式呈现出来。

这两点不仅是装饰模式的核心构成要素，也是装饰模式的精要所在。由于默认目标实现（TargetComponent）被原封不动地封装在装饰实现类（ComponentDecorator）的内部，我们就可以灵活地在装饰实现类（ComponentDecorator）中进行非常自由的扩展方式：一方面，我们可以引用到默认目标实现（TargetComponent）的行为方式并加以扩展，看上去就像我们在继承中使用覆写（Override）特性那样；另一方面，这样的设计完全不影响我们在具体装饰实现类（ComponentDecoratorA）中添加新的行为方法职责。

如果从实际效果的角度来比较一下装饰模式和继承这两种对象行为的扩展方式，我们会发现这两者之间几乎完全看不出差异。只不过，客户端对于原始接口（Component）的调用被委派到装饰类，我们就可以在接口完全不变的情况下，完成对默认目标实现（TargetComponent）的行为扩展。因而，装饰模式在这其中所体现出来的灵活之处在于：**这样的行为职责扩展方式对于客户端的调用而言是完全透明的。**

装饰模式不仅仅在形式上有别于“继承”，从之前的分析可以看出，装饰模式具备了一些比“继承”更加灵活的应用场景：

- 适合对默认目标实现（TargetComponent）中的多个接口进行排列组合调度
- 适合对默认目标实现（TargetComponent）进行选择性扩展
- 适合默认目标实现（TargetComponent）未知或者不易扩展的情况

装饰模式在实际运用的过程中也有一些简化的模型。例如，我们可以将图 4-2 的角色中的装饰实现类（ComponentDecorator）与具体装饰实现类（ComponentDecoratorA、ComponentDecoratorB）合并起来，从而简化装饰实现类的表现形式。其实现示意图如图 4-3 所示。

简化模型在实际运用中更为广泛，因为它并不死板地要求一个抽象装饰类的存在。在实际运用中，我们还是要根据实际情况在不同的模型中做出适当的取舍。

4.2.3　装饰模式的应用案例

无论是使用最基本的装饰模式模型还是简化了的模型，装饰模式的实现机理和适用场

景都是一样的。我们必须首先明确装饰模式的定义中所包含的模式的基本构成要素，才能以此为基础总结出装饰模式的应用场景。

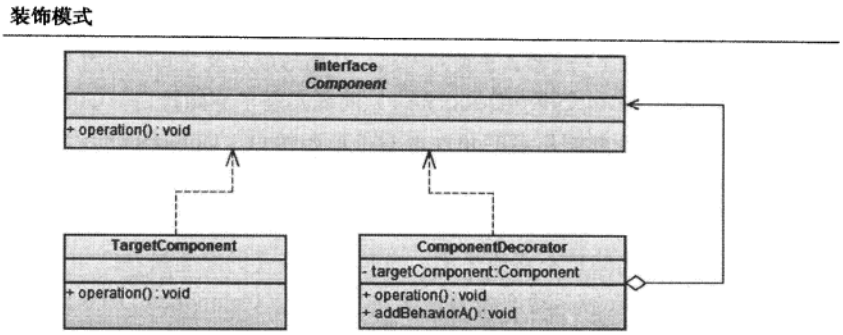


图 4-3 装饰模式的简化模型示意图

装饰模式如此简单易用，使得它被广泛运用在 JDK 以及 J2EE 规范的设计之中。例如，在我们熟知的 Servlet 标准接口设计中，天然包含了对 HttpServletRequest 和 HttpServletResponse 这两大接口的装饰实现类，如图 4-4 所示。

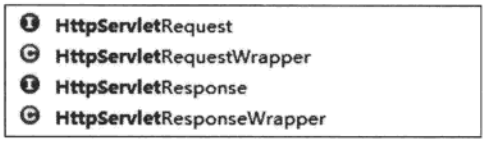


图 4-4 Servlet 标准接口中装饰模式的使用

我们知道，HttpServletRequest 和 HttpServletResponse 是 Servlet 标准所指定的 Java 语言与 Web 容器进行交互的接口。接口本身只规定 Java 语言对 Web 容器进行访问的行为方式，而具体的实现是由不同的 Web 容器在其内部实现的。例如，Tomcat、Weblogic 或者 Websphere 等不同的 Web 容器对 HttpServletRequest 和 HttpServletResponse 的实现解释就不同。

那么在运行期，当我们需要对 HttpServletRequest 或者 HttpServletResponse 的默认容器实现行为进行扩展时，我们就可以继承 HttpServletRequestWrapper 或者 HttpServletResponseWrapper 来实现。事实上，无论是 HttpServletRequestWrapper 还是 HttpServletResponseWrapper，它们都提供了一个可传入对应 HttpServletRequest 和 HttpServletResponse 接口的构造函数，并在构造函数中实现了将原始 HttpServletRequest 和 HttpServletResponse 接口的实现封装于内部的基本逻辑。这样一来，构成装饰模式的两大基本要素也就齐全了。

在下面的例子中，我们定义了一个 HttpServletRequestWrapper 的子类，并且在 Filter 中使用这个类来完成对原始接口 HttpServletRequest 中 getAttribute 方法的扩展。相关代码

如代码清单 4-7 所示。

代码清单 4-7　SampleRequestWrapper.java

```
public class SampleRequestWrapper extends HttpServletRequestWrapper {

    private CacheManager cacheManager;

    /**
     * @param req The request
     */
    public SampleRequestWrapper (HttpServletRequest req) {
        super(req);
    }

    /**
     * 根据 key 值首先在 Cache 中查找，再调用基类方法进行查找
     *
     * @param s The attribute key
     */
    public Object getAttribute(String s) {
        if (s != null && s.startsWith("cachable")) {
            String cacheKey = s.substring("cacheable.".length())
            return cacheManager.getValue(cacheKey);
        } else {
        return super.getAttribute(s);
        }
    }
}
```

有了这个 SampleRequestWrapper，我们就可以在处理 URL 请求的 Filter 中将原始的 HttpServletRequest 替换成这个装饰实现类，相关代码如代码清单 4-8 所示。

代码清单 4-8　SampleFilter.java

```
public void doFilter(ServletRequest req, ServletResponse res, FilterChain
chain) throws IOException, ServletException {

        HttpServletRequest request = (HttpServletRequest) req;
        request = new SampleRequestWrapper(request);
          chain.doFilter(request, response);
}
```

这样一来，我们就在整个 Request 生命周期获得了 HttpServletRequest 的扩展：因为对于 HttpServletRequest 来说，其原生的 getAttribute 方法的行为被加入了“从缓存中读取数据”的新特性。从扩展的灵活度来看，这种实现方法也做到了之前我们所提到的“对

默认目标实现（TargetComponent）进行选择性扩展”：我们可以在这个类的实现中发现，只有满足以 cacheable 起始的那些 Attribute 表达式才会被扩展，否则还将调用原始的 HttpServletRequest 完成逻辑。

值得注意的是，HttpServletRequestWrapper 和 HttpServletResponseWrapper 虽然是一个最为行之有效的对于 Web 容器接口进行扩展的方式，它却也是位于最最底层的一种扩展方式。因为在这个层面上的扩展，将直接改变整个 HttpServletRequest 或 HttpServletResponse 的行为方式，读者在使用时应该慎重。

如果对上面的这个应用案例做进一步的思考，我们还能挖掘出装饰模式作为对象行为的扩展方式比继承更为合理的地方：**虽然装饰模式产生的初衷是装饰类（ComponentDecorator）对默认目标实现类（TargetComponent）的行为扩展，然而装饰类（ComponentDecorator）却并不对默认目标实现类（TargetComponent）形成依赖**。

由于在装饰类（ComponentDecorator）内部封装的是目标接口（Component）而并不是默认目标实现类（TargetComponent），这样一来，我们在实现目标时，甚至无须知道具体的实现类是谁。在上面的 HttpServletRequest 和 HttpServletResponse 的扩展案例中，如果使用继承来进行行为扩展，我们不得不明确到底当前使用的是哪种 Web 容器，并且知晓当前 Web 容器对 HttpServletRequest 或 HttpServletResponse 的实现类到底是哪个类，才能达成目的。这在一个可扩展的应用程序中显然是无法接受的。

在 Struts2 中，我们可以随处看见装饰模式的应用场景。在之后的章节中，我们将结合源码做具体的讲解。

4.3 策略（Strategy）模式

4.3.1 策略模式的定义

策略模式是一种对象的行为模式，其基本定义如下：

定义 策略模式的基本含义是针对一组算法或行为特性，将它们抽象到具有共同接口函数的独立抽象类或接口中，从而使得它们可以相互替换。这样就使得某一个特定的接口行为可以在不影响客户端的情况下发生变化。

在面向对象的编程语言中，定义接口的目的在于规定一种特定的行为特征，而一个接口的不同实现类则蕴含着这种特定行为特征的不同实现机理。这种“接口－实现”的对应关系，是策略模式产生的语法基础。

策略模式的基本实现示意图如图 4-5 所示。

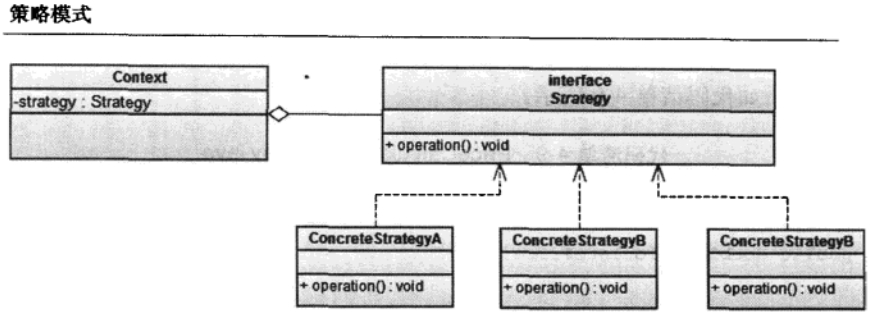


图 4-5　策略模式的基本实现模型示意图

表面看来，策略模式不能严格称之为一种设计模式，因为**它仅仅是一个接口的多个实现在运行期的选择性替换而已**。因而，我们关注的策略模式的重点并不在于这些不同实现类的具体算法的实现机制，而在于当我们面对一个业务场景可能具备有多种行为，且这多种行为相互之间可进行替换时，我们如何将它们行为特性的公共部分进行抽象，并最终形成一个统一的接口的过程。

与之前的装饰模式一样，我们同样可以从图中归纳出策略模式所涉及的主要角色：

- **环境（Context）角色**——持有一个 Strategy 类的引用，将决定调用哪种 Strategy 角色完成业务逻辑。
- **抽象策略（Strategy）角色**——这是一个抽象的策略角色，由一个接口或抽象类来扮演这个角色。这个角色是策略模式的核心，它是所有策略算法的核心归纳。因而对外表现为一个一致的接口函数。
- **具体策略（ConcreteStrategy）角色**——封装了具体的策略算法或行为。

策略模式的核心是对算法的包装，其最终目的是**把使用算法的责任（环境）和算法的实现进行解耦**。由于环境和算法的独立，算法的增加、减少或者修改都不会影响到环境和客户端的调用。当出现新的算法或者现有算法的实现发生变化时，我们只需要改变具体的策略类，并在客户端调用的地方进行注册即可。在这种情况下，策略模式中的算法实现都是“可插拔（Pluggable）”的。因此，策略模式也是插件模式的原始雏形之一。

4.3.2　策略模式的应用场景

策略模式最为基本的应用场景就是对于某种业务，可能存在多种不同的算法实现。例如，电子商务网站中的价格计算是策略模式比较典型的应用场景。用户所需支付的商品的价格，在最简单的情况下，使用货品的单价乘上数量获得。但是考虑到折扣计算时，实际

情况就复杂得多。比如，当用户购买满 50 元时，可享受一定折扣的优惠；当用户购买满 300 元，可享受折扣较大的优惠等等。

对于上述的业务场景，我们可以通过创建一个抽象策略（Strategy）角色对“价格计算”进行抽象，如代码清单 4-9 所示。

代码清单 4-9 PriceCalculationStrategy.java

```
public interface PriceCalculationStrategy {

   public double calculatePrice();

}
```

而对于不同的“价格计算”的算法，则被定义成不同的策略角色实现类，如代码清单 4-10 和代码清单 4-11 所示。

代码清单 4-10 NoDiscountPriceCalculationStrategy.java

```
public class NoDiscountPriceCalculationStrategy implements
PriceCalculationStategy {

  private double price;

  private double copies;

  public NoDiscountPriceCalculationStrategy(double price,double copies)
{
  this.price = price;
  this.copies = copies;
  }

  public BigDecimal calculatePrice() {
    return price * copies;
  }

}
```

代码清单 4-11 PercentageDiscountPriceCalculationStrategy.java

```
public class PercentageDiscountPriceCalculationStrategy implements
PriceCalculationStrategy {

  private double price;

  private double copies;
```

```
    private double rate;

    public PercentageDiscountPriceCalculationStrategy(double price, double
  copies,double rate) {
      this.price = price;
      this.copies = copies;
      this.rate = rate;
    }

    public BigDecimal calculatePrice() {
      return price * copies * rate;
    }

  }
```

从上述的例子中，我们可以看到策略模式的核心要义在于：**抽象**。所谓抽象，指的是对所有核心算法的行为接口的抽象统一。在上面的例子中，calculatePrice 就是对“价格计算”这一行为的统一化抽象。

在完成了抽象之后，策略模式在调用时需要完成其另外一个核心要义：**选择**。所谓选择，指的是在运行期不同的算法实现之间进行选择，根据我们在上一节中针对策略模式的类图分析（图 4-5）可知：这个选择的决定权掌握在客户端调用手中，具体的策略类是无从知晓当前这个策略类到底适用于哪一种实际业务场景的。

掌握了策略模式的核心要义，我们可以站在编程的角度来观察其为我们带来的好处：

- **策略模式提供了管理一组算法族的方法**——通过接口和多个算法实现之间的契约接口来完成业务场景。
- **策略模式提供了可以替换通过继承进行对象行为扩展的方法**——使用公共抽象接口的不同实现类而并非一个抽象类的继承链完成行为的扩展。
- **策略模式提供了将算法的调用责任与算法逻辑进行解耦的方法**——通过角色转移，将调用环境角色（Context）与算法抽象（Strategy）之间分开，形成引用关系。

在 Struts2 / XWork 中，策略模式的应用也非常广泛。XWork 核心组件之一的容器（Container）中所管理的对象就是建立在“接口 - 实现”模式之下，而这也同时成为 Struts2 的插件（Plugin）实现的理论基础。在之后的章节中，我们将在 Struts2 初始化和插件模式的实现机理中看到大量的策略模式的使用范例。

4.3.3　策略模式的深入思考

在策略模式的定义中，我们可以总结出一个策略模式使用过程中的重要的前提条件：**客户端环境（Context）必须知道所有的策略类、理解这些不同策略算法之间的区别，并**

自行决定使用哪一个策略类来完成业务逻辑。

在这种情况下，“做决定”将会成为客户端环境颇为头疼的一个问题。因为在“做决定”这件事情上，客户端将不得不借助某些复杂的多重条件判断来完成。例如，在上一节中的价格折扣计算的例子中，客户端的调用过程可能如代码清单 4-12 所示。

代码清单 4-12 Test.java

```
        PriceCalculationStrategy strategy = null;

        if( price <= 300) {
            strategy = new NoDiscountPriceCalculationStrategy(price, copies);
        } else if( price > 300 && price <= 1000 ) {
            strategy = new PercentageDiscountPriceCalculationStrategy(price, copies,
0.9);
        } else if( price > 1000 ) {
            strategy = new PercentageDiscountPriceCalculationStrategy(price, copies,
0.8);
        }

        strategy.calculatePrice();
```

这种 if/else 的多重条件判断，显然不是一种最佳实践的体现。那么，我们如何对这段代码进行优化呢？在实际情况中，我们会利用策略模式中“客户端必须知晓所有的策略类”这一特性，将多重条件判断进行一次再抽象，从而完成**策略决定权的下放**。整个重构过程，如代码清单 4-13 和代码清单 4-14 所示。

代码清单 4-13 PriceCalculationStrategy.java

```
    public interface PriceCalculationStategy {

        public boolean match(double price);

        public double calculatePrice();

    }
```

代码清单 4-14 Test.java

```
    List<PriceCalculationStrategy> strategies =
    getPriceCalculationStrategies();
    for(PriceCalculationStrategy strategy : strategies) {
       if(strategy.match(price)) {
              strategy.calculatePrice();
       }
    }
```

这样一来，客户端在策略的选择上，就不会陷入困境。因为策略类作为一个算法类，它自身的调用特性（满足什么条件下，策略类的使用条件成立并可以被调用）也被封装在策略类的内部。从代码的特性看，这一重构的过程成功地把多重条件判断转化为一个循环 + 判断的结构，从而极大地提高了代码的可读性和可维护性。

这就是我们对策略模式中有关“**决策权**”的深入思考。它在一定程度上颠覆了策略模式的设计结构，因为客户端在决定调用策略类的时候，并没有自己来决定，而是去询问具体的策略类。然而，这种决策权的转移，却也能够在某些特定的业务场景中显示出其强大的威力。

我们对策略模式的另外一层思考，是有关策略模式的核心要义之一：**选择**。在上一节中，我们谈到策略模式的核心要义在于：抽象和选择。在这其中，“抽象”的过程成为了“选择”的基础，因而“选择”是我们的目的，“抽象”则是手段。将“选择”作为使用策略模式的基本目的时，我们所需要解决的实际问题是：在多种可能出现的算法集合中选择一个应用的具体的业务场景。

事实情况是，“选择”是策略模式的唯一运用模式吗？有没有一种业务场景，我们需要在多个策略算法中，选择其中的多个进行执行并取得执行结果的合集呢？实际情况是肯定的，在 Struts2 中，我们就存在着这样的场景：Struts2 在初始化时需要将纷繁复杂的配置形式（有 XML 文件的配置形式、Properties 文件的配置形式、Annotation 的配置形式等）转化成 Struts2 内置配置对象。

Struts2 应对这种业务场景，设计一个配置加载的策略接口，并且应对不同的配置形式设计了不同的配置策略实现类。在初始化定义的过程中，我们将不再看到策略类的**选择**过程，而是对不同策略实现类的依次调度并将执行结果**搜集**起来的过程。在本书的第三部分讲解 Struts2 的初始化过程时，我们还将看到这种“**搜集**”模式在策略模式中的使用范例。

通过本节中对策略模式使用的两个角度的深入思考，我们更加应该体会到任何一个设计模式的使用都不是死板的，而是需要根据实际情况进行重构和调整。只有满足具体业务场景的编程模式，才是我们所追求的编程模式。

4.4　构造（Builder）模式

4.4.1　构造模式的核心要素

构造模式是一种对象的创建模式。它可以将一个复杂对象的内部构成特征与对象的构建过程完全分开。构造模式的种类很多，图 4-6 列出了构造模式的其中一种实现示意图。

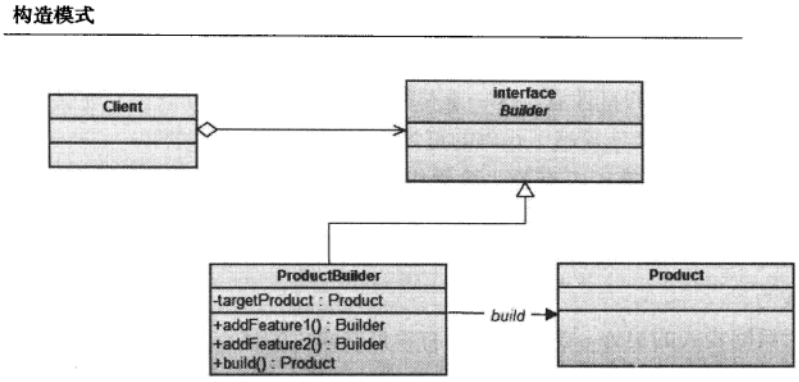


图 4-6 构造模式实现示意图

从图中，我们可以总结出构造模式所涉及的 4 个角色：

- **客户端（Client）角色**——调用具体的构造器完成对象构建。注意这个角色只负责构造器的创建和选择，对于产品的具体信息并不知晓。
- **抽象构造器（Builder）角色**——这是一个抽象的构造器角色，由一组接口方法来扮演。在实际情况中，这一组接口会由一个核心的 build 方法（负责最终对象的创建）以及若干辅助方法（帮助构建对象的内部属性特征）共同构成。
- **具体构造器（ProductBuilder）角色**——具体的构造器实现。具体构造器是整个构造模式的核心，其中封装了一个目标的对象实例。当其中的核心 build 方法被客户端调用时，在其内部缓存的目标对象实例会真正返回。而在这之前，所有其他辅助方法的调用都仅仅是将内部缓存的目标对象进行内容上的填充。
- **产品（Product）角色**——整个构造模式的产物。这个角色是构造模式执行的结果。

在实际应用中，构造模式有一种变种：当对于构造对象非常明确，而对于构造过程化的定义并不严格时，我们可以省略“抽象构造器”这一接口定义的角色而直接使用一个“具体构造器”，客户端通过直接调用“构造器”内部所定义的方法来完成对象构建。其示意图如图 4-7 所示。

构造模式之所以能够产生这样的变种，与所构造对象的特性和整个构造过程有直接的联系。我们在下一节中还将具体阐述。但无论是哪种形式的构造模式实现，客户端（Client）角色、构造器（Builder）和构造对象（Target）都是构造模式必不可少的核心要素。它们之间的调用关系，则构成了构造模式运行的基础。

构造模式

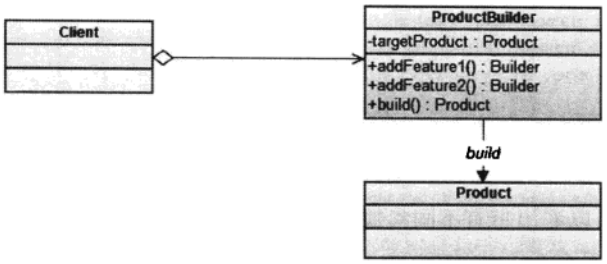


图 4-7　省略了“抽象构造器”的构造模式

4.4.2　构造模式的应用场景

我们知道，对象的构建过程本身并不复杂。许多面向对象的编程语言都提供了对象构建的基本语法支持：构造函数。通过定义和设计不同的构造函数，我们可以赋予对象一些特殊的特性。下面的代码列出了一个 User 类的设计，其中包含两个不同的构造函数，不同的构造函数构造出来的 User 对象表达出不同的“用户角色”定义，如代码清单 4-15 所示。

代码清单 4-15　User.java

```
public class User {

    private Integer id;

    private String name;

    private int age;

    /**
     * 构建一个仅仅包含标识符的 User 实例
     *
     * @param id
     */
    public User(Integer id) {
            this.id = id;
    }

    /**
     * 构建一个 User 实例，其中包含了用户名和年龄的属性
     *
     * @param name
```

```
     * @param age
     */
    public User(String name, int age) {
            this.name = name;
            this.age = age;
    }

}
```

从上述代码中，我们可以看出带有不同参数的构造函数在对象构造上的区别：目标对象的性质是完全不同的。

构造函数似乎已经可以解决绝大多数的对象构造问题。然而，当一个对象的内部结构比较复杂时，构造函数有时候就会显得力不从心。尤其是当对象的内部结构中如果嵌套了另外一个对象从而形成对象之间的依赖关系时，对象的构建过程就会变得更加复杂。例如，在某些情况下，对象内部的某些字段所表现出来的特性非常重要，当这些字段还没有被赋予恰当的值之前，这个对象不能作为一个“合格”的对象使用。在这种情况下，构造函数不得不处理非常复杂的逻辑，从而使整个对象树的构建成为一场噩梦。

我们还是以 User 对象为例，如果在 User 对象中增加一个 Department 嵌套对象，用于表示 User 对象所属的部门，再增加一个 basicSalary 字段表示 User 对象的基础工资。然而，basicSalary 是根据 Department 中的某个系数 Rate 计算得到。此时，整个 User 对象的设计就变成了如代码清单 4-16 所示的样子。

代码清单 4-16 User.java

```
public class User {

    private Integer id;

    private String name;

    private int age;

    private Department department;

    private double basicSalary;

    /**
     * 构建一个仅仅包含标识符的 User 实例
     *
     * @param id
     */
    public User(Integer id) {
            this.id = id;
```

```
    }

    /**
     * 构建一个 User 实例，其中包含了用户名和年龄的属性
     *
     * @param name
     * @param age
     */
    public User(String name, int age) {
            this.name = name;
            this.age = age;
    }

    public User(String name, int age, double salary, Department department) {
            this.name = name;
            this.age = age;
            this.department = department;
            this.basicSalary = department.getSalaryRate() * salary;
    }
}
```

从代码中，我们可以看到 User 对象的构建过程将依赖于 Department 对象的构建过程，在这其中还掺杂了计算逻辑（计算 basicSalary）。此时，User 对象的构建开始变得复杂，而当设计进一步扩展时，这种构建的过程就变得极为不可控。

由此，我们可以得出构造模式的一个常见的应用场景：

结论　构造模式适用于构建对象的构造过程十分复杂、构建对象的初始化对于其内部的对象有着强烈逻辑依赖的业务场景。

除了构造的过程可能出现复杂而难以控制的情况之外，还有一种情况是构造模式的天然应用场景。我们在这里以咖啡师的日常工作为例。咖啡的种类很多，有“摩卡”、“拿铁”、“美式咖啡”等等；消费者在购买咖啡时，还会提出不同的对咖啡制作的要求，例如“加糖”、“加牛奶”；咖啡本身从温度上还将分成“热饮”和“加冰”这两种情况。单单从咖啡的角度来看待问题，咖啡是咖啡师最终的制作产品，而我们所说的咖啡的不同种类、消费者对咖啡制作的要求、咖啡的温度等都可看作是咖啡内部的一些特性。咖啡师要制作一杯咖啡，必须通过多次操作咖啡机来为咖啡添加种种特性。

在这种情况下，构造模式就可以派上用场。在上述场景中，咖啡师就是构造模式中的客户端角色，负责操作的是咖啡机。而咖啡机就是构造模式中的构造器角色，相应的咖啡就是构造模式中的最终成品了。

结论　构造模式适用于构建对象拥有不确定的内部特性，客户端可以根据其自身需要选择

对象构建方式和对象内部特性的业务场景。

从上面的例子中，可以看到我们需要构建的对象虽然都是“咖啡”，但到底是“拿铁”还是“摩卡”、到底是“热饮”还是“加冰”都可以由咖啡师（客户端）来决定。咖啡师（客户端）只需要根据实际需求，进行咖啡机（构造器）的操作即可。

在 Struts2 中，有许多内置对象的数据结构是非常复杂的。尤其是 Struts2 在初始化时，将纷繁复杂的配置元素转化成为内置对象的过程，是一个由众多依赖和众多步骤构成的过程。我们在之后的章节中，将看到构造模式在这个过程中起着决定性的作用。

4.4.3 对象构造步骤

既然对象的构造有赖于构造模式的帮助，那么对象的构造过程一定是一个复杂的过程。针对这一复杂的过程，我们有必要从对象的构造模式的调用过程中提炼出一些共性的地方来，从而帮助我们更好地对构造模式的核心元素（构造器）进行设计。

在 4.4.2 节的“制造咖啡”的例子中，我们可以看到咖啡师（客户端）对咖啡机（构造器）的操作是有一定步骤的。可以把这个步骤用我们熟悉的时序图来表示，如图 4-8 所示。

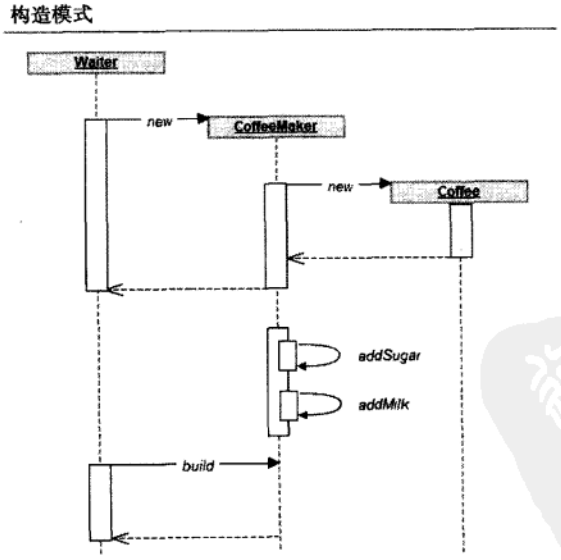


图 4-8 咖啡制作的时序图

在时序图中，我们可以看到整个构造模式中的核心元素“构造器”的调用过程，可以

分为三个主要的步骤：

- 构造器（Builder）首先会创建一个目标对象的实例加以缓存。
- 构造器依次调用构建目标对象特性的行为方法（在此例中就是 addSugar、addMilk 等），其目的在于为目标对象搜集并设置其内在特性。
- 构造器最终调用其核心方法 build 完成目标对象的创建并返回给客户端。

在整个过程中，我们可以看到后面两个步骤是构造模式真正涉及的“共性”部分。我们几乎可以在所有构造模式的使用过程中看到这两个步骤的存在。因为这两个步骤是构造模式能够将对象构造模式化的依据：前一个步骤能够确保将复杂的对象构造过程进行合理的逻辑拆分，从而使得每一个子步骤都能够独立完成逻辑而不造成对其他外部环境的依赖；后一个步骤则在前一个步骤的基础之上，将真正的对象创建封装起来。

在 Struts2 中，我们将经常看到构造模式中这两个步骤的接口函数定义，读者只要牢记不同的接口函数在设计上的目标不同即可。

4.5　责任链（Chain Of Responsibility）模式

4.5.1　责任链模式的定义

责任链模式是一种对象的行为模式。其基本定义如下：

定义　责任链模式的基本含义是将一个事件处理流程分派到一组执行对象上去，这一组执行对象形成一个链式结构，事件处理请求在这一组对象上进行传递，每一个执行对象要么进行逻辑处理，要么将请求沿着“链”继续传递给下一个执行对象，直到有一个执行对象完成所有的逻辑处理并返回结果。

责任链模式的定义很长而且不容易理解。我们在这里可以引入一个所有读者都知晓的游戏：击鼓传花。击鼓传花是一个典型的责任链模式的应用实例。当鼓声开始之后，花束开始沿着参与游戏的人进行传递，当鼓声停止时，依然拿着花束的人就要表演节目。

我们可以从击鼓传花的游戏过程中总结出责任链模式的主要参与角色：

- **请求（Request）角色**——请求角色指的是在整个游戏过程中的“花束”。在游戏进行的过程中，“花束”始终沿着一定的轨迹在所有参与游戏的人之间进行传递。
- **执行对象（ConcreteHandler）角色**——这里的执行对象角色就是指所有参与游戏的人。这个角色在整个过程中有两种截然不同的行为模式：将花束传递到下一个游戏者和游戏中止时表演节目。

如果用编程语言来表述整个过程，我们会把具体的执行对象角色进行行为抽象，得到一个抽象执行对象（Handler）角色。其实现示意图如图 4-9 所示。

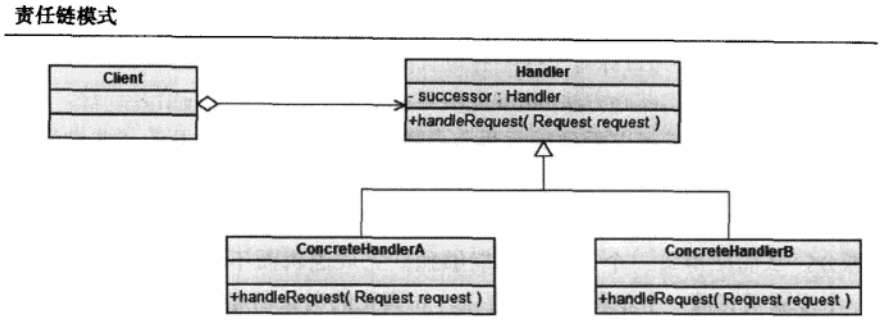


图 4-9 责任链模式实现示意图

根据之前对击鼓传花游戏的理解，我们可以总结责任链模式的一些重要特性：

- 请求（花束）在这个链上传递，直到链上的某一个执行对象决定处理此请求。在实际应用中，每个执行对象还可以进行请求的“部分处理”，并且将处理结果与请求角色一起传递到下一个执行对象进行继续处理。
- “链”的存在状态由对象的依赖协作关系决定。因此“链”既可以是一条顺序的直线，也可以是一个环，或者是一个树状结构。
- 负责调用责任链的客户端对于到底是哪个执行对象来进行请求的处理并不知晓，处理责任在这里完全转嫁到每一个具体的执行对象。

责任链模式降低了请求的发送端和接收端之间的耦合。因而，多个执行对象都可以有机会来处理这个请求。这就为我们在事件处理流程中划分合理的执行层次打下了坚实的基础。同时，这种解耦合的过程也给我们提供了在划分好的执行层次中进行逻辑扩展的空间。

责任链模式是站在对象行为的角度对对象行为进行扩展的一种方法。读者可以与其他的对象行为模式进行比较，体会责任链模式在处理问题时的特殊性。

4.5.2 责任链模式的逻辑意义

在了解了责任链模式的基本定义之后，我们来看看责任链模式的引入对整个事件处理机制在逻辑上有什么影响。

我们知道，任何编程语言都是由三种不同的逻辑共同构成：

- 顺序执行逻辑
- 循环执行逻辑

❑ 分支判断逻辑

当我们执行一段程序时，三种逻辑的穿插将共同决定程序的执行次序。在绝大多数情况下，一个事件处理的主流程是由一段顺序执行逻辑为主体进行驱动的。因为站在事件流程的角度，我们总是把处理事件的流程划分为若干个步骤，并进一步指定我们先做什么、再做什么、最后做什么。顺序执行逻辑是人类处理问题的基本思维方式，也是我们经常所说的“条理”。

然而当一个事件的处理流程非常复杂而导致整个顺序执行的流程变得很长时，程序代码也会相应地变得难以维护。在这种情况下，我们非常有必要将一个顺序执行的流程步骤分派到不同的执行对象上去。这就是**形成责任划分的最原始的初衷和想法**。由此可见，形成“责任”角色的抽象，是为了将每个执行流程规范化。

当我们把处理流程合理地划分到不同的执行对象上去之后，另外一个问题又显现出来：这些不同的执行对象在总的逻辑主线上依然顺序执行，我们如何能够将不同对象之间的执行逻辑以一定的方式串联起来，从而使得整个流程的执行更加自动化呢？如果仅仅划分“责任”，而并不改变程序的执行逻辑，那么事件处理的复杂度从本质上讲并没有降低。**在这种情况下，“链”的概念形成了**。我们可以利用面向对象的基本特性中的对象依赖协作将所有的执行对象串联起来形成一条“链”。根据上一节我们对“链”的概念的解析，“链”的状态不仅可以是直线，也可以是环状或者是树状。很显然，当“链”的状态是一个环状时，程序的执行逻辑由顺序变为循环；而当“链”的状态是一个树状时，程序的执行逻辑由顺序变为分支判断。

综上所述，责任链模式在程序的执行逻辑上能够达到两个主要的效果：

❑ 成为事件处理流程进行逻辑步骤划分的基础

❑ 改变事件处理机制中单一的顺序处理的执行逻辑

责任链模式是 XWork 框架中的核心设计模式。XWork 框架中的拦截器栈就是基于责任链模式实现的。从责任链模式所能够提供的两大功效来看，XWork 框架顺利完成了事件执行层次的划分以及执行逻辑的规范化。我们在第 8 章中将结合源码做进一步的分析。

4.6　小结

“问渠哪得清如许，为有源头活水来”。无论是什么框架，都离不开一系列最佳实践的积累。而设计模式，则是最佳实践的集中营。在本章中，我们带领读者领略了四种设计模式和一种编程模式的种种细节。设计模式本身是博大精深的，本书的内容也只涉及设计模式中的冰山一角。而本书之所以挑选这些设计模式进行详细剖析，主要原因有两点：第一，它们是 Struts2 / XWork 框架中较为核心的一些设计模式，有些甚至是 Struts2 / XWork

框架某些功能的实现基础；第二，这些设计模式所分属的类别截然不同，因而它们所处理问题的角度和适用范围也完全不同，我们的目的是能够覆盖到对象结构扩展、对象行为扩展、对象构造扩展的方方面面。因而，读者在这里应该将重点放在理解这些设计模式，并且能够指出这些设计模式在框架中的应用场景，从而能够在自己的应用编写中用到这些设计模式。

读完本章，大家不妨再来看看以下的这些问题是否迎刃而解了。

- 什么是线程安全？多线程环境中的线程安全问题，到底是指对哪类变量的访问发生的问题？
- 解决多线程并发访问有哪些基本方法？
- Servlet 对象是线程安全的吗？ Web 容器处理 Http 请求的基本流程是什么？
- 实现 ThreadLocal 模式的基本步骤有哪些？
- ThreadLocal 模式是被用作数据共享还是数据传递？引入 ThreadLocal 模式，对整个 Web 开发有什么优点和缺点？
- 什么是装饰模式？装饰模式与“继承”在对象行为扩展上有什么区别？
- 什么是策略模式？策略模式有哪些应用场景？
- 什么是构造模式？构造模式有哪些应用场景？
- 构造模式在进行对象构建时，可以分为哪些步骤？
- 什么是责任链模式？责任链模式有哪些基本要素？
- 责任链模式的引入，对事件处理机制的逻辑执行有什么意义？

第 5 章　生命之源——XWork 中的容器

对象的生命周期管理在基于面向对象的编程语言中是一个永恒的话题。从语法上讲，面向对象的高级编程语言都是以“对象”为中心的。而对象之间的继承关系、嵌套引用关系所形成的对象树结构为我们进行对象级别的逻辑操作提供了足够的语法支持。但这样一来，对象之间所形成的复杂关系也为对象生命周期的管理带来了问题：

- **在程序的运行期，应如何创建我们所需要的对象？**
- **当创建一个新的对象时，如何保证与这个对象所关联的依赖关系（其关联对象）也能够被正确地创建出来？**

这两大问题不仅是面向对象的编程语言中的核心问题，也是每个框架在进行设计时必须跨越的坎。业界对于这样的问题也早有公论：

结论　*为了更好地管理好对象的生命周期，我们有必要在程序逻辑中引入一个额外的编程元素，这个元素就是容器（Container）。*

在本章中，我们就来探讨这一额外的编程元素——容器的方方面面，并深入分析 XWork 框架的容器实现机制。

5.1　容器，对象生命周期管理的基石

5.1.1　对象的生命周期管理

在本章一开始，我们就提出了面向对象编程语言中的两个核心问题。这两大问题，实际上涵盖了对象生命周期管理的两个不同方面：

- **在程序的运行期，对象实例的创建和引用机制**
- **对象与其关联对象的依赖关系的处理机制**

这里所谈到的对象生命周期管理的两个方面与之前谈到的两个核心问题恰好一一对应。为了帮助读者更好地对这两大方面进行解读，我们在这里引入一个很重要的概念：**控制反转**（Inverse of Control）。

控制反转是对象生命周期管理中的一个核心概念，由软件大师 Martin Fowler 于 2004

年提出，并以此为基础创造了一个大家更为熟悉和理解的概念：**依赖注入**（Dependency Injection）。

那么，什么是控制反转呢？在使用基于面向对象的编程语言进行开发时，对象的概念是实现业务逻辑的基础核心。而要真正实现一个复杂的业务需求，则离不开多个对象彼此之间的通力协作来共同完成。在这里，我们似乎可以理解为什么要在第 2 章中强调对象的关系模型中的协作关系模型了。实际上，对象关系模型中协作的真正含义，**正是对象通过其依赖对象的帮助，完成其业务逻辑的过程**。而动作特性的对象的嵌套引用，是对象之间精诚合作、协作完成业务逻辑的基本途径，也是对象之间形成依赖关系的根本原因。在这种情况下，**每个对象不得不依赖于与其协作的对象（也就是它所依赖的对象）的引用和构建**。更加通俗地来说：

结论 *每个对象自身对于逻辑的执行能力，被其所依赖的对象反向控制了，这也就是控制反转的本质含义。*

当对象之间的这种依赖关系与运行期（Runtime）对象的构建机制相结合时，情况就会变得更为复杂。因为在对象创建的时候，除了需要考虑这个对象本身作为一个实例的创建过程，还需要考虑与这个对象形成依赖关系的关联对象的实例化过程。在这里，前一个过程能够使对象自身的生命周期得到控制和管理；而后一个过程，则保证了程序在运行期的使用不会受到其所依赖的对象的生命周期的限制。

因此，我们可以看出对象的生命周期管理的两个不同方面的内容是密不可分的。对象的创建是对象依赖关系管理的基础，没有对象的创建过程，其所关联的依赖对象的生命周期也就无从谈起；另外一个方面，如果对象的依赖关系无法被正确处理，那么我们创建出来的对象也就失去了基本的活力，因为它失去了一个对象最为基本的与其他对象协同合作的能力。

5.1.2 容器（Container）的引入

控制反转概念的提出对我们编写程序提出了额外的要求，因为我们不得不去实现“获取依赖对象”这一基本的逻辑功能，从而使得对象与对象之间的协作和沟通变得更为畅通。既然如此，那么如何实现这个获取依赖对象的过程就值得我们仔细掂量。如果这个过程靠程序逻辑自身来实现（也就是说在程序的运行过程中自行管理），那么我们至少可以预见到这种编程模式存在着三大弊端：

- **对象将频繁创建，效率大大降低（尽管在大多数情况下，这些对象都是无状态的单例对象）**

- **对象的创建逻辑与业务逻辑代码高度耦合，使得一个对象的逻辑关注度大大分散**
- **程序的执行效率大大降低，由于无法区分明确的职责，很难针对对象实现的业务逻辑进行单元测试**

正是看到了这些弊端，业界的软件大师就提出一条“获取依赖对象的过程”的最佳实践：**应该引入一个与具体的业务逻辑完全无关的额外的编程元素容器来帮助进行对象的生命周期管理**。这也就是我们在本章的一开始所提到的那个“业界公论”。

引入容器是以 Java 为代表的面向对象的编程语言发展过程中的一个重要里程碑。因而，几乎所有的开源框架都有自己的容器实现。甚至有些框架（诸如 Spring），更是以容器作为其最核心的基础构建。

既然引入容器的目的是为了解决对象生命周期管理中所遇到的问题，那么这一额外的编程元素自身的设计也必须遵循一定的原则：

- **容器应该被设计成一个全局的、统一的编程元素**

这一点几乎无须多做解释。在运行期获取对象实例可能会发生在程序的任何一个角落，因此容器作为一个辅助元素也可能随时在任何地方被调用。这就不得不要求容器对象是一个全局对象。

- **在最大程度上降低容器对业务逻辑的入侵**

这是进行容器设计时需要考虑的一个最重要的问题。因为我们引入容器这个额外元素来帮助管理对象的生命周期的初衷之一，就是由于我们希望能够把对象生命周期管理的部分从具体的业务逻辑中提取出来，使得业务逻辑本身能够方便地进行单元测试。

- **容器应该提供简单而全面的对象操作接口**

引入容器的目的就是为了进行对象操作。因而，简单而全面在这里所表达的意思，一方面是说针对对象的操作接口应该一目了然；另外一个方面，针对对象的操作接口应该涵盖对象生命周期管理的所有内容。

作为一个优秀的事件处理框架，XWork 也有框架级别的容器实现，并且 XWork 的容器在实现机理上有小巧、实用的特点。这一点，我们将会在之后的章节中通过源码体会出来。

5.1.3　容器（Container），不是容器（Collection）

在 Java 世界中有一个被称之为 Collection 的接口，用于表述一组对象的集合。由 Collection 引申开来的数据结构很多，比如 List、Set 等。这些接口都继承自 Collection 接口，各自表示不同特性的数据结构。从中文翻译的角度，我们习惯把 Collection 翻译为“容器”，而所有相关的接口及其实现类都被称为“容器结构”。

我们在这里所引入的容器（Container）的概念与 Collection 接口所表述的容器概念

完全不同。虽然它们在中文名称上是一致的，然而我们却可以从对应的英文表述中看出它们的不同：

结论 容器（Collection）≠ 容器（Container）

容器（Collection），是一个具体的数据结构类。在容器（Collection）之中存储的内容是**对象**。而容器（Container），却不是一个具体的数据结构类，它用于管理对象的生命周期，我们可以把它看作是一个全局的编程元素。因此，在容器（Container）之中，我们将看不到具体的对象存储。整个容器（Container）从外部看来，就如同是**一个可以进行对象操作的工具类**。

由此，我们就可以在容器（Container）的基本设计原则的基础之上，对容器（Container）的设计提出具体的要求。

结论 *容器（Container）由一系列对象的操作接口构成，其中应该至少包含获取对象实例以及管理对象之间的依赖关系这两类操作方法。*

明确了这一条结论，能够帮助我们对容器（Container）的表现形式有更加深刻的认识，并且从中体会到：容器的设计原则与引入容器的初衷也是一致的。

我们在本章中谈论的所有话题，显然是围绕着容器（Container）展开的。既然此容器（Container）不同于彼容器（Collection），那么其具体实现内部所存储的内容也当然完全不一样。有关这一点，我们将在下一节中结合 XWork 容器的实现进行具体分析。

5.2 XWork 容器概览

在上一节中，我们已经探讨了引入容器的重要意义以及容器在对象生命周期管理中的作用。XWork 作为一个优秀的开发框架，在其内部也实现了一个小型的容器。接下来，我们将对 XWork 中实现的容器做一个简单的介绍，其中包括容器的定义、容器的管辖范围和容器的基本操作。

5.2.1 XWork 容器的定义

XWork 框架中的容器被定义成为一个 Java 接口，其相关源码如代码清单 5-1 所示。

代码清单 5-1 Container.java

```
public interface Container extends Serializable {
```

```
    /**
     * 定义默认的对象获取标识
     */
    String DEFAULT_NAME = "default";

    /**
     * 进行对象依赖注入的基本操作接口，作为参数的 object 将被 XWork 容器进行处理。
     * object 内部声明有 @Inject 的字段和方法，都将被注入受到容器托管的对象，
     * 从而建立起依赖关系
     */
    void inject(Object object);

    /**
     * 创建一个类的实例并进行对象依赖注入
     */
    <T> T inject(Class<T> implementation);

    /**
     * 根据 type 和 name 作为唯一标识，获取容器中的 Java 类的实例
     */
    <T> T getInstance(Class<T> type, String name);

    /**
     * 根据 type 和默认的 name(default) 作为唯一标识，获取容器中的 Java 类的实例
     */
    <T> T getInstance(Class<T> type);

    /**
     * 根据 type 获取与这个 type 所对应的容器中所有注册过的 name

     * @param type
     * @return
     */
    Set<String> getInstanceNames(Class<?> type);

    /**
     * 设置当前线程的作用范围的策略
     */
    void setScopeStrategy(Scope.Strategy scopeStrategy);

    /**
     * 删除当前线程的作用范围的策略
     */
    void removeScopeStrategy();
}
```

从容器的接口定义方法来看，它完全能够符合我们之前所讨论的容器设计的基本原则之一：**简单而全面**。从接口的内容和表现形式来看，它也能符合我们对容器的基本要求：

容器首先被设计成一个接口而不是具体的实现类，而整个接口定义中既包含获取对象实例的方法，也包含管理对象依赖关系的方法。

在这里，我们可以看到容器设计的基本原则在一定程度上指导着容器的接口设计，因为我们**更加关心容器能够对外提供什么样的服务，而并不是容器自身的数据结构。**

从源码中，我们可以依照方法的不同作用对这些操作接口进行分类：

- **获取对象实例**——getInstance、getInstanceName
- **处理对象依赖关系**——inject
- **处理对象的作用范围策略**——setScopeStrategy、removeScopeStrategy

既然容器被定义为一个 Java 接口，那么我们同时也来关注一下容器的实现类的一些基本特性。

结论 *容器是一个辅助的编程元素，它在整个系统中应该被实例化为一个全局的、单例的对象。*

这是容器实现中最为基本的一个特性。这也是由容器自身的设计初衷所决定的。如果我们在整个系统中能够获取多个不同的容器的对象实例，或者容器的对象实例在整个系统中的作用域又存在局域性，那么我们依托容器进行对象生命周期管理就会变得混乱不堪。

结论 *容器在系统初始化时进行自身的初始化。系统应该提供一个可靠的、在任何编程层次都能够对这个全局的容器或者容器中管理的对象进行访问的机制。*

这一条结论是我们对容器实现的基本要求。从这条结论中，我们可以看到两个不同的方面：

- **容器的初始化需求**——应该掌握好容器初始化的时机，并考虑如何对容器实例进行系统级别的缓存
- **系统与容器的通信机制**——应该提供一种有效的机制与这个全局的容器实例进行沟通

有关这两个不同方面的实现机理，我们将在接下来的章节中陆续给出源码级别的解析。其中有关容器的初始化过程，蕴含在整个框架的初始化主线中。我们将在第 9 章详细解读。而系统与容器的通信机制，则涉及 XWork 容器自身的数据结构和实现机理，因而也成为本章的重点之一。读者在这里应体会 XWork 框架在容器的设计上与之前我们所提到的容器设计的基本原则之间的吻合度，这对我们整个面向对象的设计理念将有极大的提升。

5.2.2　XWork 容器的管辖范围

既然引入容器的主要目的在于管理对象的生命周期，那么在明确了 XWork 的容器定义之后，就非常有必要去了解一下 XWork 容器的管辖范围。换句话说，如果拥有了这个全局的容器实例，当我们调用容器的操作接口时，到底操作的是哪些对象呢？

从容器操作接口的角度来看，容器的两类操作接口：获取对象实例（getInstance）和实施依赖注入（inject），它们所操作的对象有所不同。接下来我们就对这两类不同的操作接口分别进行分析。

5.2.2.1　获取对象实例

当调用容器的 getInstance 方法来获取对象实例时，**我们只能够获取那些“被容器接管”的对象的实例**。那么，哪些对象属于“被容器接管”的对象呢？

在第 3 章中，我们已经介绍过 Struts2 / XWork 的配置元素以及这些配置元素的分类。当时，我们把 XML 配置文件中的基本节点分为两类：其中一类是 bean 节点和 constant 节点，这两个节点统称为“**容器配置元素**”；另外一类则是 package 节点，这个节点下的所有配置定义都被称之为“**事件映射关系**”。而我们进行配置元素分类的基本思路是按照**XML 节点所表达的逻辑含义和该节点在程序中所起的作用来进行**。

现在，当回过头再来看配置元素的分类时，我们就能理解“容器配置元素”的真正含义了。在 XML 配置元素中，bean 节点广泛用于定义框架级别的内置对象和自定义对象；而 constant 节点和 Properties 文件中的配置选项则被用于定义系统级别的运行参数。我们之所以把这两类节点统称为“容器配置元素”，就是因为它们所定义的对象的生命周期，都是由容器管理的，这些对象也就是所谓的“被容器接管”的对象。

结论　XWork 容器所管理的对象包括所有框架配置定义中的“容器配置元素”。

根据之前的分析，这些对象主要可以分为三类：

- **在 bean 节点中声明的框架内部对象**
- **在 bean 节点中声明的自定义对象**
- **在 constant 节点和 Properties 文件中声明的系统运行参数**

在这里需要注意的是，我们通过容器获取的这些对象的实例，不仅自身被初始化，对象内部的所有依赖对象也已经被正确地实施依赖注入。很显然，这就是我们使用容器进行对象生命周期管理的好处。

我们对这三类容器托管对象的归纳，实际上蕴含了我们对自定义对象纳入 XWork 容器管理的过程：**只要在 Struts2 / XWork 的配置文件中进行声明即可**。

5.2.2.2 对象的依赖注入

当调用容器的 inject 方法来实施依赖注入操作时，**所操作的对象却不仅仅限于“容器配置元素”中所定义的对象**。因为我们对 inject 方法的定义是，**只要传入一个对象的实例，容器将负责建立起传入对象实例与容器托管对象之间的依赖关系**。

由此可见，虽然传入 inject 的操作对象是任意的，实施依赖注入操作时的那些依赖对象却是被容器接管的对象。这就为任意对象与 XWork 容器中所管理的对象之间建立起一条通道提供了有效的途径。

结论 *调用 XWork 容器的 inject 方法，能够帮助我们将容器所管理的对象（包括框架的内置对象以及系统的运行参数）注入到任意的对象实例中去，从而建立起任意对象与框架元素沟通的桥梁。*

这一条结论非常关键，因为它不仅反映了容器的基本职责，也是我们日后进行应用级别对象操作的理论基础。有关容器的两大类操作的具体实现机制，我们将在之后的章节中陆续给出分析。

从方法的命名上，inject 非常直观，表达了“注入”的含义，与之前提到的“依赖注入”的概念是吻合的。深入思考一下 inject 方法的逻辑，我们就会发现这个方法的内部实现实际上蕴含了系统与容器对象之间的通信机制。根据之前对 XWork 容器对象的定义，我们可以看到 inject 方法的调用流程：当某个对象实例作为参数传入方法之后，该方法会扫描传入对象内部声明有 @Inject 这个 Annotation 的字段、方法、构造函数、方法参数并将它们注入容器托管对象，从而建立起传入对象与容器托管对象之间的依赖关系。

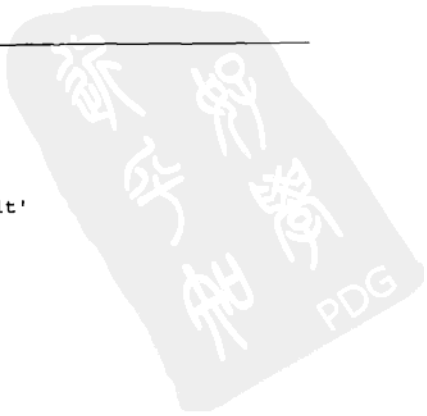
由此可见，整个流程的调用过程被一个神秘的 Annotation 有效地驱动。我们接下来就来看看 @Inject 这个 Annotation 的定义，如代码清单 5-2 所示。

代码清单 5-2 Inject.java

```
@Target({METHOD, CONSTRUCTOR, FIELD, PARAMETER})
@Retention(RUNTIME)
public @interface Inject {

  /**
   * 进行依赖注入的名称。如果不声明，这个名称会被设置为 'default'
   */
  String value() default DEFAULT_NAME;

  /**
   * 是否必须进行依赖注入，仅仅对于方法和参数有效
```



```
    */
    boolean required() default true;
}
```

从 @Inject 的定义中，我们看到这个 Annotation 可以被设置在**任何对象**的方法、构造函数、内部实例变量或者参数变量之中。在这里，我们可以看到对 @Inject 的使用并不受限于对象本身。它既可以加入到 Struts2 / XWork 的内置对象之上，也可以加到任意我们自行编写的对象之上。一旦它加入到我们自定义的对象之中，那么就建立起了自定义对象与容器托管对象之间的联系。因为被加入了 @Inject 这个 Annotation 的方法、构造函数、内部实例变量或者方法参数变量，实际上是在告诉容器：请为我注入由容器托管的对象实例。

细细考虑这个过程，这不正是我们引入容器来解决对象生命周期管理的目标吗？当我们需要寻求容器帮忙时，只要在恰当的地方加入一个标识符 Annotation，容器在进行依赖注入操作时，就能够知晓并接管整个过程了。在这里，我们看到两个过程共同构成了 XWork 容器进行对象依赖注入操作的步骤：

- **为某个对象的方法、构造函数、内部实例变量、方法参数变量加入 @Inject 的 Annotation**
- **调用容器的 inject 方法，完成被加入 Annotation 的那些对象的依赖注入**

因此，我们在这里顺利解决了在容器定义中所提到的一个核心问题：如何建立起系统到容器或者容器托管对象的沟通桥梁——通过 @Inject 声明来完成。

5.2.3　XWork 容器操作详解

5.2.3.1　通过容器接口进行对象操作

在了解 XWork 中容器的操作定义以及 XWork 容器的管辖范围之后，我们可以看看如何通过直接操作容器的实例来进行对象操作。

首先来看如何通过容器对象来获取对象实例。我们在这里摘取了 XWork 框架中的一个处理类 DefaultUnknownHandlerManager 进行说明，其相关源码如代码清单 5-3 所示。

代码清单 5-3　DefaultUnknownHandlerManager.java

```
public class DefaultUnknownHandlerManager implements
UnknownHandlerManager {

    protected ArrayList<UnknownHandler> unknownHandlers;
    private Configuration configuration;
```

```
    private Container container;

    @Inject
    public void setConfiguration(Configuration configuration) {
        this.configuration = configuration;
        build();
    }

    @Inject
    public void setContainer(Container container) {
        this.container = container;
        build();
    }

    protected void build() {
    // 如果 configuration 对象不为空，则依次从 configuration 对象
    // 以及 Container 中读取 UnknownHandler 的实例
    if (configuration != null && container != null) {
            List<UnknownHandlerConfig> unkownHandlerStack =
configuration.getUnknownHandlerStack();
            unknownHandlers = new ArrayList<UnknownHandler>();

            if (unkownHandlerStack != null
&& !unkownHandlerStack.isEmpty()) {
                // 根据一定顺序获取 UnknownHandlers 实例
                for (UnknownHandlerConfig unknownHandlerConfig :
unkownHandlerStack) {
                    // 调用 container 对象的 getInstance 方法获取 UnknownHandler
                    UnknownHandler uh =
container.getInstance(UnknownHandler.class,
unknownHandlerConfig.getName());
                    unknownHandlers.add(uh);
                }
            } else {
                // 调用 container 对象的 getInstanceNames 方法获取
                // 所有受到容器管理的 UnknownHanlder 实例名称
                Set<String> unknowHandlerNames =
container.getInstanceNames(UnknownHandler.class);
                if (unknowHandlerNames != null) {
                    // 根据名称调用 container 对象的 getInstance 方法获取实例
                    for (String unknowHandlerName : unknowHandlerNames) {
                        UnknownHandler uh =
container.getInstance(UnknownHandler.class, unknowHandlerName);
                        unknownHandlers.add(uh);
                    }
                }
            }
        }
    }
```

```
        // 这里省略了许多其他的代码

    }
```

在这里，我们看到了通过容器对象获取对象实例的两种方法：getInstance 和 getInstanceNames。其中，前者用于获取接受容器托管的具体对象实例，后者则用于对一个接口的多个不同实现类之间的实例获取的管理。在这里需要注意的是，在代码示例中的 build 方法调用的前提是 setContainer 方法对容器对象的正确初始化。

有关容器的另外一种操作：依赖注入，我们通过 XWork 框架中的核心类 ActionSupport 的源代码来进行解释说明，如代码清单 5-4 所示。

代码清单 5-4　ActionSupport.java

```
public class ActionSupport implements Action, Validateable, ValidationAware,
TextProvider, LocaleProvider, Serializable {

    // 这里省略了许多其他的代码

    private TextProvider getTextProvider() {
        if (textProvider == null) {
            TextProviderFactory tpf = new TextProviderFactory();
            if (container != null) {
                container.inject(tpf);
            }
            textProvider = tpf.createInstance(getClass(), this);
        }
        return textProvider;
    }

    @Inject
    public void setContainer(Container container) {
        this.container = container;
    }

    // 这里省略了许多其他的代码

}
```

在上面的代码中，我们看到两个主要的方法：getTextProvider 和 setContainer。从逻辑上讲，很明显 getTextProvider 以 setContainer 的存在为基础。setContainer 实际上就是框架帮助我们获取全局的容器实例的具体方法。值得我们注意的是 @Inject 这个 Annotation 的使用，使得 setContainer 方法将在 ActionSupport 初始化时被注入全局的 Container 对象。而 getTextProvider 则在运行期被调用，此时全局的容器对象中的

接口函数就可以被随意调用，并完成依赖注入操作。具体来说，就是代码中的 container.inject(tpf) 操作。

综合上述的操作容器进行的两类对象操作：获取受到容器托管的对象和对象的依赖注入操作，我们可以从中得出使用容器进行对象操作的几个要点：

- **通过操作容器进行对象操作的基本前提是当前的操作主体能够获得全局的容器实例。因而，全局的容器实例的获取在操作主体的初始化过程中完成**
- **通过操作容器进行的对象操作都是运行期（Runtime）操作**
- **通过操作容器所获取的对象实例，都是受到容器托管的对象实例**
- **通过操作容器进行的依赖注入操作，可以针对任意对象进行，该操作可以建立起任意对象和容器托管对象之间的联系**

读者在这里或许对这些结论还一知半解，在之后的章节中，我们将为读者一一解开这些容器操作中的疑惑。在这里，读者应首先谨记这四个要点，理解它们的基本要义，并且将它们作为 XWork 容器操作的基本结论。因为在之后的源码分析中经常会遇到需要直接操作全局容器实例的范例，牢记这些基本结论之后读者对 Struts2 / XWork 中内置对象的操作就不会产生障碍。

5.2.3.2 通过 Annotation 获取容器对象实例

在展开本节的话题之前，我们首先来回顾一下上一节中得出的一个重要结论：

结论 通过操作容器进行对象操作的基本前提是，当前的操作主体能够获得全局的容器实例。因而，全局的容器实例的获取在操作主体的初始化过程中完成。

这个重要结论在之前对容器进行操作的示例代码中也能够得到证实，那就是以下这样一段公共代码，如代码清单 5-5 所示。

代码清单 5-5 setContainer 方法

```
    @Inject
    public void setContainer(Container container) {
        this.container = container;
}
```

我们对这段公共代码实际含义的解读是：**在当前的对象操作主体进行初始化时，这个方法会被调用，而全局的容器对象则会被初始化到当前的对象操作主体之中。**然而，这个方法并不是对象构造函数的一部分，那么这个方法又是如何被包含到对象的初始化过程中去的呢？在这里，引发这一系列神秘操作的，就是加在方法之上的这个 Annotation：@Inject。

在上一节的分析中，我们得知 @Inject 是建立起任意对象实例与容器托管对象之间桥梁的唯一途径。因此，当我们需要在一个自定义对象（非容器托管）中获得容器托管对象的实例时，就可以借助 @Inject 这个 Annotation 来实现。下面的例子就展示了这样一个过程，如代码清单 5-6 所示。

代码清单 5-6　ObjectProviderTest.java

```
public class ObjectProviderTest {

    private ObjectFactory objectFactory;

    @Inject
    public void setObjectFactory(ObjectFactory objectFactory) {
        this.objectFactory = objectFactory;
    }
}
```

在这个例子中，我们使用的对象是一个自定义的对象 ObjectProviderTest，然而我们却需要在这个对象中获得容器托管的对象（在这里，ObjectFactory 是受到 XWork 容器托管的框架内置对象）的实例。整个过程通过 @Inject 的注入来完成。

在本节中，读者应始终沿着 XWork 容器进行对象依赖注入的操作步骤进行过程的解读。读到这里，或许读者已经迫不及待地想要弄清楚容器内部操作的实现细节了。在接下来的章节中，我们就将揭开这个神秘过程的种种细节。

5.3　深入浅出 XWork 容器

5.3.1　XWork 容器的存储结构

XWork 容器内部是什么样的呢？这里首先介绍 XWork 容器的存储结构，然后介绍 XWork 容器的实现机理。接下来我们从对象制造工厂和注入器两个方面来进行介绍。

5.3.1.1　对象制造工厂（factory）

我们知道，XWork 的容器被定义成一个接口，其内部封装了一组操作方法。既然它并不是一个具体的数据结构类，那么站在其实现类内部的数据存储的角度，容器内部到底是什么样子呢？我们先来看看 XWork 容器的实现类 ContainerImpl 的源码。其部分源码如代码清单 5-7 所示。

代码清单 5-7 ContainerImpl.java

```
class ContainerImpl implements Container {

  final Map<Key<?>, InternalFactory<?>> factories;
  final Map<Class<?>,Set<String>> factoryNamesByType;

  ContainerImpl(Map<Key<?>, InternalFactory<?>> factories) {
      this.factories = factories;
      Map<Class<?>,Set<String>> map = new HashMap<Class<?>,Set<String>>();
      for (Key<?> key : factories.keySet()) {
        Set<String> names = map.get(key.getType());
        if (names == null) {
          names = new HashSet<String>();
          map.put(key.getType(), names);
        }
        names.add(key.getName());
      }

      for (Entry<Class<?>,Set<String>> entry : map.entrySet()) {
        entry.setValue(Collections.unmodifiableSet(entry.getValue()));
      }

      this.factoryNamesByType = Collections.unmodifiableMap(map);
  }
  // 这里省略了许多其他的代码

}
```

从源码中，我们可以看到 ContainerImpl 内部所封装的两个内部实例变量：factories 和 factoryNamesByType。它们都是 Map 结构，其中 factories 是由构造函数传递进入并缓存于内部，而 factoryNamesByType 则在 factories 的基础之上做了一个根据名称进行寻址的缓存映射关系。

我们来看看 factories 这个存储结构中的 Key，其定义如代码清单 5-8 所示。

代码清单 5-8 Key.java

```
class Key<T> {

  final Class<T> type;
  final String name;
  final int hashCode;

  private Key(Class<T> type, String name) {
    if (type == null) {
```

```
      throw new NullPointerException("Type is null.");
    }
    if (name == null) {
      throw new NullPointerException("Name is null.");
    }

    this.type = type;
    this.name = name;

    hashCode = type.hashCode() * 31 + name.hashCode();
  }

  Class<T> getType() {
    return type;
  }

  String getName() {
    return name;
  }

  @Override
  public int hashCode() {
    return hashCode;
  }

  @Override
  public boolean equals(Object o) {
    if (!(o instanceof Key)) {
      return false;
    }
    if (o == this) {
      return true;
    }
    Key other = (Key) o;
    return name.equals(other.name) && type.equals(other.type);
  }

  @Override
  public String toString() {
    return "[type=" + type.getName() + ", name='" + name + "']";
  }

  static <T> Key<T> newInstance(Class<T> type, String name) {
    return new Key<T>(type, name);
  }
}
```

在这其中，我们可以看到 Key 实际上是由一个二元组构成的对象集合。而这个二元

组的构成对象 type 和 name 实际上正好可以和我们之前所提到的“容器配置元素”中 bean 节点的定义对应起来。也就是说，在我们之前所提到的 XML 定义中的 bean 节点，通过 type 和 name 的组合进行唯一性寻址。例如，下面的代码摘自 struts-default.xml，我们可以看到不同的 type 和 name 的组合对应了不同的实现类关系，如代码清单 5-9 所示。

代码清单 5-9 struts-default.xml

```
<bean type="com.opensymphony.xwork2.ActionProxyFactory" name="xwork"
class="com.opensymphony.xwork2.DefaultActionProxyFactory"/>
<bean type="com.opensymphony.xwork2.ActionProxyFactory" name="struts"
class="org.apache.struts2.impl.StrutsActionProxyFactory"/>
```

我们再来看看 factories 结构中的 Value，它的具体类型是 InternalFactory。InternalFactory 的定义的相关源码如代码清单 5-10 所示。

代码清单 5-10 InternalFactory.java

```
interface InternalFactory<T> extends Serializable {

  /**
   * 指定如何创建一个对象
   *
   * @param context of this injection
   * @return instance to be injected
   */
  T create(InternalContext context);
}
```

这个泛型的接口只有一个 create 方法。其基本含义是说：**一旦实现这个接口，我们就需要指定对象的创建机制。**

由此可见，factories 中存储的内容，是 Java 对象（type 所指向的 Java 类）的实例构建方法。正所谓“授人以鱼不如授人以渔”，有了对象实例的构建方法，我们就可以随时在运行期获得对象的实例。

结论 *在容器内部进行缓存的是对象实例的构建方法，而不是对象实例本身。这就让容器看起来像一个工厂的集合，能够根据不同的要求，制造出不同种类的对象实例。*

这个结论就是容器操作中获取对象实例部分的核心结论。从这个结论中我们也可以看出，容器之所以不被定义成一个具体的数据结构类而被定义成一系列操作接口的真正原因是：**它的内部的确就是一个工厂！**

5.3.1.2　注入器

除了“获取对象实例”外，XWork 容器的另外一个重要的操作接口是“实施对象的依赖注入”操作。因而，从数据结构的角度，XWork 容器的内部除了缓存一个对象制造工厂 factories 用以在运行期能够创建对象实例并返回之外，还需要另一类缓存的帮助，这类缓存用于记录对象与对象之间的依赖关系。这一类缓存数据在 XWork 容器的内部被称之为**注入器**（Injector），其相关源码如代码清单 5-11 所示。

代码清单 5-11　ContainerImpl.java

```
class ContainerImpl implements Container {

// 这里省略了许多其他的代码

/**
 * 字段和方法的注入器的初始化
 */
final Map<Class<?>, List<Injector>> injectors =
    new ReferenceCache<Class<?>, List<Injector>>() { // 这里采用了一个缓存
      @Override
      protected List<Injector> create(Class<?> key) {
        List<Injector> injectors = new ArrayList<Injector>();
        // 将当前的 class 类作为 key，查找所有满足条件的 Injector
        addInjectors(key, injectors);
        return injectors;
      }
};

// 这里省略了许多其他的代码

}
```

在这里，ReferenceCache 是 XWork 框架中对于缓存的一种简单实现。它提供了一种在运行期构建 Map 内容的机制。翻开其源码，我们会发现 ReferenceCache 中维护着一个 ConcurrentMap，并且在内部应用 ThreadLocal 模式很好地规避了对象操作的多线程问题，其相关源码如代码清单 5-12 所示。

代码清单 5-12　ReferenceCache.java

```
public abstract class ReferenceCache<K, V> extends ReferenceMap<K, V> {

  transient ConcurrentMap<Object, Future<V>> futures =
      new ConcurrentHashMap<Object, Future<V>>();

  transient ThreadLocal<Future<V>> localFuture = new
```

```
ThreadLocal<Future<V>>();

  /**
   *
   */
  protected abstract V create(K key);

  // 这里省略了许多其他的代码
}
```

有了 ReferenceCache，我们操作 Map 的方式会有所改变。当调用 Map 中的接口 get 时，ReferenceCache 会首先查找其内部是否已存在相应的缓存对象。如果存在，则直接返回；如果不存在，则会调用其抽象方法 create 根据 key 的内容产生对象并缓存起来。

在之前的源码中，我们可以看到注入器被设计成一个 ReferenceCache 的缓存，其中缓存的 Key 是每一个 Class 对象，而缓存的 Value 则是根据 Class 对象查找到的所有隶属于 Class 中的注入器。而 Key 和 Value 之间的建立过程，则通过 ReferenceCache 中的 create 方法来完成。

在这里，**注入器就是 XWork 容器中实施依赖注入的核心**。注入器本身是一个接口，规定了对象进行依赖注入的行为，其相关源码如代码清单 5-13 所示。

代码清单 5-13 ContainerImpl.java

```
interface Injector extends Serializable {
    void inject(InternalContext context, Object o);
}
```

在注入器的 ReferenceCache 定义中，我们看到所有注入器都是在运行期动态添加的，其核心方法 create 会调用 addInjectors 扫描所有满足条件的注入器，并加入到 ReferenceCache 之中，其相关源码如代码清单 5-14 所示。

代码清单 5-14 ContainerImpl.java

```
/**
 * 根据某个 class，查找所有满足条件的注入器
 */
void addInjectors(Class clazz, List<Injector> injectors) {
    if (clazz == Object.class) {
      return;
    }

    // 首先递归调用自身，以完成对父类的注入器查找
    addInjectors(clazz.getSuperclass(), injectors);
```

```
    // 针对所有属性查找满足条件的注入器，并加入到 injectors 中进行缓存
    addInjectorsForFields(clazz.getDeclaredFields(), false, injectors);
    // 针对所有方法查找满足条件的注入器，并加入到 injectors 中进行缓存
    addInjectorsForMethods(clazz.getDeclaredMethods(), false,
injectors);
}

// 针对所有方法查找满足条件的注入器，并加入到 injectors 中进行缓存
void addInjectorsForMethods(Method[] methods, boolean statics,
List<Injector> injectors) {
// 使用统一的接口进行查找，并在内部实现 InjectorFactory，指定相应的 Injector
// 的实际实现类
    addInjectorsForMembers(Arrays.asList(methods), statics, injectors,
        new InjectorFactory<Method>() {
          public Injector create(ContainerImpl container, Method method,
              String name) throws MissingDependencyException {
            // 这里指定 methodInjector 作为 Injector 实现
            return new MethodInjector(container, method, name);
          }
        });
}

// 针对所有属性查找满足条件的注入器，并加入到 injectors 中进行缓存
void addInjectorsForFields(Field[] fields, boolean statics,
List<Injector> injectors) {
    // 使用统一的接口进行查找，并在内部实现 InjectorFactory，指定相应的 Injector
    // 的实际实现类
    addInjectorsForMembers(Arrays.asList(fields), statics, injectors,
      new InjectorFactory<Field>() {
        public Injector create(ContainerImpl container, Field field,
            String name) throws MissingDependencyException {
          // 这里指定 fieldInjector 作为 Injector 实现
          return new FieldInjector(container, field, name);
        }
      });
}

// 统一的 Injector 查找方式
<M extends Member & AnnotatedElement> void addInjectorsForMembers(
      List<M> members, boolean statics, List<Injector> injectors,
      InjectorFactory<M> injectorFactory) {
    for (M member : members) {
      if (isStatic(member) == statics) {
        // 查找当前传入的 member 是否具备 @inject 的 Annotation
    Inject inject = member.getAnnotation(Inject.class);
        if (inject != null) {
          try {
            // 调用传入的 injectorFactory 中的 create 方法创建真正的 Injector 实例
```

```
            injectors.add(injectorFactory.create(this, member,
inject.value()));
          } catch (MissingDependencyException e) {
            if (inject.required()) {
              throw new DependencyException(e);
            }
          }
        }
      }
    }
}
```

其中包括一连串函数调用。我们最为关心的是最后一个 addInjectorsForMembers 方法。从源码中我们可以看到，无论是属性还是方法，Struts2 的依赖注入都通过 @Inject 这个 Annotation 作为标识来进行区分。针对任何一个对象，凡是加上 @Inject 这个 Annotation 的属性或者方法都会被初始化相应的注入器，并在调用 Container 的 inject 方法时实施依赖注入。

由此，我们对于 @Inject 这个 Annotation 的谜团也可以正式解开。读者可以发现，@Inject 虽然只是一个标识符，然而在容器内部它却被转化成为不同的 Injector 对象从而实施依赖注入。读者几乎感觉不到对象的依赖注入在字段、构造函数、方法、方法参数之间的区别。

5.3.2 XWork 容器的实现机理

5.3.2.1 统一操作——模板方法的运用

翻开 XWork 容器中 ContainerImpl 的源码，我们会发现有一个模板方法是所有容器操作的调用基础，这个方法就是 callInContext 方法，其相关源码如代码清单 5-15 所示。

代码清单 5-15 ContainerImpl.java

```
/**
 * 查找线程安全的执行上下文环境，根据要求创建或者销毁执行环境
 */
<T> T callInContext(ContextualCallable<T> callable) {
    // 从 ThreadLocal 变量中获取一个执行上下文环境
    Object[] reference = localContext.get();
    if (reference[0] == null) {
  // 如果执行上下文环境不存在，创建一个新的环境
      reference[0] = new InternalContext(this);
      try {
        // 直接调用回调接口完成逻辑
        return callable.call((InternalContext)reference[0]);
```

```
        } finally {
          reference[0] = null;
        }
      } else {
        // 如果执行上下文环境已经存在，直接调用回调接口完成逻辑
        return callable.call((InternalContext)reference[0]);
      }
    }
```

这个模板方法将被 ContainerImpl 中所有的接口方法调用。这里使用一个模板方法作为统一的操作入口的主要原因在于将所有的接口操作进行规范化定义，并将它们纳入一个线程安全的上下文执行环境中。

而具体的执行逻辑，则被封装于 ContextualCallable 接口的回调实现之内。对于不同的接口实现，它们会拥有不同的逻辑，而这些逻辑由 ContextualCallable 的实现类自行处理。

5.3.2.2　获取对象的实现

获取对象的接口是 getInstance 方法，其相关源码如代码清单 5-16 所示。

代码清单 5-16　ContainerImpl.java

```
public <T> T getInstance(final Class<T> type, final String name) {
    return callInContext(new ContextualCallable<T>() {
      public T call(InternalContext context) {
        return getInstance(type, name, context);
      }
    });
}

@SuppressWarnings("unchecked")
<T> T getInstance(Class<T> type, String name, InternalContext context) {
    ExternalContext<?> previous = context.getExternalContext();
    Key<T> key = Key.newInstance(type, name);
    context.setExternalContext(ExternalContext.newInstance(null, key, this));
    try {
      InternalFactory o = getFactory(key);
      if (o != null) {
          return getFactory(key).create(context);
      } else {
          return null;
      }
    } finally {
      context.setExternalContext(previous);
    }
}
```

从源码上看，getInstance 方法是一个模板方法的调用。在其内部的回调接口中，直接调用了 ContainerImpl 中的内部方法 getInstance。我们可以在这里看到 getInstance 方法的核心步骤在于根据 type 和 name 构成的 Key 去获取其所对应的 InternalFactory 实现。根据之前对 ContainerImpl 内部数据存储结构的分析，我们知道这个核心步骤的实现只需要直接通过其内部缓存的 factories 对象的寻址即可完成。因而，剩下的过程仅仅在于使用 InternalFactory 所规定的对象构建方法返回对象的实例即可。

5.3.2.3 依赖注入的实现

在之前的分析中，我们知道对象的依赖注入的实现的核心在于 Injector 这个接口。因而我们就来首先研究一下 Injector 的实现结构，如图 5-1 所示。

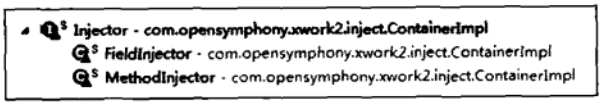


图 5-1 Injector 的实现结构

从图中，我们可以看到 Injector 的基本实现。它是基于 Field 和 Method 这两个最最基本的对象元素实现的。其相关源码如代码清单 5-17 所示。

代码清单 5-17 ContainerImpl.java

```
static class FieldInjector implements Injector {

    final Field field;
    final InternalFactory<?> factory;
    final ExternalContext<?> externalContext;

    // 构造函数，为实施依赖注入做数据准备
    public FieldInjector(ContainerImpl container, Field field, String name)
        throws MissingDependencyException {
        // 缓存 field
        this.field = field;
        // 检查 field 是否可写，并设置 field 的可写属性
        if (!field.isAccessible()) {
            SecurityManager sm = System.getSecurityManager();
            try {
                if (sm != null) sm.checkPermission(new
ReflectPermission("suppressAccessChecks"));
                field.setAccessible(true);
            } catch(AccessControlException e) {
                throw new DependencyException("Security manager in use, could
not access field: " + field.getDeclaringClass().getName() + "(" +
field.getName() + ")", e);
            }
        }
```

```
            // 根据 type 和 name 到 container 内部工厂中查找相应的对象构造工厂
            Key<?> key = Key.newInstance(field.getType(), name);
            factory = container.getFactory(key);
            // 如果没有找到相应的对象构造工厂，则注入失败
            if (factory == null) {
              throw new MissingDependencyException(
                  "No mapping found for dependency " + key + " in " + field + ".");
            }

            // 为对象构建设置 externalContext
            this.externalContext = ExternalContext.newInstance(field, key, container);
          }

          // 实际实施依赖注入的方法
          public void inject(InternalContext context, Object o) {
            ExternalContext<?> previous = context.getExternalContext();
            context.setExternalContext(externalContext);
            try {
              // 使用初始化时找到的对象构造工厂创建对象，并使用反射进行注入
              field.set(o, factory.create(context));
            } catch (IllegalAccessException e) {
              throw new AssertionError(e);
            } finally {
              context.setExternalContext(previous);
            }
          }
        }

        static class MethodInjector implements Injector {

          final Method method;
          final ParameterInjector<?>[] parameterInjectors;

          // 构造函数，为实施依赖注入做数据准备
          public MethodInjector(ContainerImpl container, Method method, String name)
              throws MissingDependencyException {
              // 缓存 method
              this.method = method;
             // 检查 method 是否可写，并设置 method 的可写属性
             if (!method.isAccessible()) {
                  SecurityManager sm = System.getSecurityManager();
                  try {
                      if (sm != null) sm.checkPermission(new
    ReflectPermission("suppressAccessChecks"));
                      method.setAccessible(true);
                  } catch(AccessControlException e) {
                      throw new DependencyException("Security manager in use, could
not access method: " + name + "(" + method.getName() + ")", e);
```

```
            }
          }

          // 利用反射查找方法的每一个参数
          Class<?>[] parameterTypes = method.getParameterTypes();
          if (parameterTypes.length == 0) {
            throw new DependencyException(
                method + " has no parameters to inject.");
          }
          // 针对每个参数查找对应的 Injector
          parameterInjectors = container.getParametersInjectors(
              method, method.getParameterAnnotations(), parameterTypes, name);
      }

        // 实际实施依赖注入的方法
        public void inject(InternalContext context, Object o) {
          try {
            // 调用反射完成方法的调用，实施依赖注入
            method.invoke(o, getParameters(method, context, parameterInjectors));
          } catch (Exception e) {
            throw new RuntimeException(e);
        }
      }

      }
```

有了之前对容器内部结构的分析，MethodInjector 和 FieldInjector 的初始化过程就显得非常简单了。在 MethodInjector 和 FieldInjector 的构造函数中，我们可以看到一个重要的步骤：**在容器内部根据 type 和 name 进行对象构造工厂 factory 的寻址**。之后的 inject 调用过程，只不过是调用 factory 构建对象，并使用 Java 中最为普通的反射机制来完成对象的依赖注入。

容器自身的 inject 方法的实现，如代码清单 5-18 所示。

代码清单 5-18 ContainerImpl.java

```
public void inject(final Object o) {
    // 使用回调模式，将不同类型的注入通过回调方法进行有机统一
    callInContext(new ContextualCallable<Void>() {
      public Void call(InternalContext context) {
        // 调用实际的注入方法
        inject(o, context);
        return null;
      }
    });
}
```

```
// 实际实施依赖注入的方法
void inject(Object o, InternalContext context) {
    // 在传入的对象中查找需要被实施依赖注入的字段或者方法
    List<Injector> injectors = this.injectors.get(o.getClass());
    for (Injector injector : injectors) {
  // 调用不同的 Injector 的实现类实施依赖注入
      injector.inject(context, o);
    }
  }
```

从源码上看，inject 依然还是一个模板方法的调用。在其内部的回调接口实现中，包含了 2 个非常简单的步骤：**查找当前对象所需要被注入的字段或者方法以及调用相应的 injector 实现类进行依赖注入**。有了之前对 injector 的分析，对象的依赖注入操作也就一目了然了。

5.4　统一的容器操作接口——ObjectFactory

从之前对 XWork 容器的操作范例可以看出，对象的创建与对象的依赖注入是对象生命周期管理的两个不同方面，因而它们在 Container 接口中所表现出来的具体接口方法也不同。我们同时也注意到，虽然这两者分属对象生命周期管理的两个不同的方面，但在实际应用中往往却像一对双胞胎兄弟那样形影不离：在我们创建一个新的对象之后，往往会调用 Container 中的 inject 方法为这个对象进行依赖注入的操作。这一点从逻辑上讲也能够说得通：当构建一个新的对象时，我们总应该负责任地把这个对象的依赖关系设置完整。这样，当我们需要再次从容器中获取这个对象时，就显得游刃有余。

既然对象管理的这两个方面之间的联系如此紧密，那么在 XWork 中有没有一个统一的操作接口将这两种对象操作的逻辑封装在一起呢？答案是肯定的。XWork 中提供了这样一个工具类 ObjectFactory，允许程序员在程序的运行期动态地构建一个新的对象，并且为这个新构建的对象实施依赖注入操作。ObjectFactory 的相关源码如代码清单 5-19 所示。

代码清单 5-19　ObjectFactory.java

```
public class ObjectFactory implements Serializable {

    // 内部缓存 ClassLoader
    private transient ClassLoader ccl;
    // 内部缓存 Container，用于实施依赖注入
    private Container container;

    protected ReflectionProvider reflectionProvider;
```

```
    // 对 ClassLoader 实施依赖注入
    @Inject(value="objectFactory.classloader", required=false)
    public void setClassLoader(ClassLoader cl) {
        this.ccl = cl;
    }

    @Inject
    public void setReflectionProvider(ReflectionProvider prov) {
        this.reflectionProvider = prov;
    }

    public ObjectFactory() {
    }

    public ObjectFactory(ReflectionProvider prov) {
        this.reflectionProvider = prov;
    }

    // 对 Container 实施依赖注入
    @Inject
    public void setContainer(Container container) {
        this.container = container;
    }

    /**
     * 使用 XWork 的 ClassLoaderUtil 根据 className 获取 Class 对象，
     * ClassLoaderUtil 内部采用了缓存机制来提高效率
     *
     * @param className
     * @return
     * @throws ClassNotFoundException
     */
    public Class getClassInstance(String className) throws
ClassNotFoundException {
        if (ccl != null) {
            return ccl.loadClass(className);
        }

        return ClassLoaderUtil.loadClass(className, this.getClass());
    }

    /**
     * 构建 XWork 中 Action 实例的快捷方法
     */
    public Object buildAction(String actionName, String namespace,
ActionConfig config, Map<String, Object> extraContext) throws Exception {
        return buildBean(config.getClassName(), extraContext);
    }
```

```
    /**
     * 根据给定的 clazz 和外部的上下文环境构建一个对象
     */
    public Object buildBean(Class clazz, Map<String, Object> extraContext)
throws Exception {
        return clazz.newInstance();
    }

/**
 * 针对给定的 object 实施依赖注入
 *
     * @param obj
     */
    protected Object injectInternalBeans(Object obj) {
        if (obj != null && container != null) {
            container.inject(obj);
        }
        return obj;
    }

    /**
     * 根据给定的 className 和外部上下文环境构建对象并实施依赖注入
     */
    public Object buildBean(String className, Map<String, Object>
extraContext)throws Exception {
        return buildBean(className, extraContext, true);
    }

    /**
     * 构建对象的实际操作方法
     *
     */
    public Object buildBean(String className, Map<String, Object>
extraContext, boolean injectInternal) throws Exception {
        // 首先创建对象
        Class clazz = getClassInstance(className);
        Object obj = buildBean(clazz, extraContext);

        // 针对对象实施依赖注入
        if (injectInternal) {
            injectInternalBeans(obj);
        }
        return obj;
    }

    /**
     * 构建 XWork 框架中 Interceptor 对象的快捷方法
     */
    public Interceptor buildInterceptor(InterceptorConfig
```

```
interceptorConfig, Map<String, String> interceptorRefParams) throws
ConfigurationException {
        String interceptorClassName = interceptorConfig.getClassName();
        Map<String, String> thisInterceptorClassParams =
interceptorConfig.getParams();
        Map<String, String> params = (thisInterceptorClassParams == null) ?
new HashMap<String, String>() : new HashMap<String,
String>(thisInterceptorClassParams);
        params.putAll(interceptorRefParams);

        String message;
        Throwable cause;

        try {
            // Interceptor 实例构成的栈结构生命周期较长，并且跨越 Session 执行，在这里不
               应传入额外的 Context 信息
            Interceptor interceptor = (Interceptor)
buildBean(interceptorClassName, null);
            reflectionProvider.setProperties(params, interceptor);
            interceptor.init();

            return interceptor;
        } catch (InstantiationException e) {
            cause = e;
            message = "Unable to instantiate an instance of Interceptor class
[" + interceptorClassName + "].";
        } catch (IllegalAccessException e) {
            cause = e;
            message = "IllegalAccessException while attempting to
instantiate an instance of Interceptor class [" + interceptorClassName + "].";
        } catch (ClassCastException e) {
            cause = e;
            message = "Class [" + interceptorClassName + "] does not implement
com.opensymphony.xwork2.interceptor.Interceptor";
        } catch (Exception e) {
            cause = e;
            message = "Caught Exception while registering Interceptor class "
 + interceptorClassName;
        } catch (NoClassDefFoundError e) {
            cause = e;
            message = "Could not load class " + interceptorClassName +
". Perhaps it exists but certain dependencies are not available?";
        }

        throw new ConfigurationException(message, cause, interceptorConfig);
    }

    /**
     * 构建 XWork 框架中 Result 对象的快捷方法
```

```
     */
    public Result buildResult(ResultConfig resultConfig, Map<String,
Object> extraContext) throws Exception {
        String resultClassName = resultConfig.getClassName();
        Result result = null;

        if (resultClassName != null) {
            result = (Result) buildBean(resultClassName, extraContext);
            Map<String, String> params = resultConfig.getParams();
            if (params != null) {
                for (Map.Entry<String, String> paramEntry :
params.entrySet()) {
                    try {
reflectionProvider.setProperty(paramEntry.getKey(),
paramEntry.getValue(), result, extraContext, true);
                    } catch (ReflectionException ex) {
                        if (result instanceof ReflectionExceptionHandler) {
                            ((ReflectionExceptionHandler) result).handle(ex);
                        }
                    }
                }
            }
        }

        return result;
    }

/**
 * 构建 XWork 框架中 Validator 对象的快捷方法
*/
public Validator buildValidator(String className, Map<String, String>
params, Map<String, Object> extraContext) throws Exception {
        Validator validator = (Validator) buildBean(className, null);
        reflectionProvider.setProperties(params, validator);

        return validator;
    }

    static class ContinuationsClassLoader extends ClassLoader {

    }
}
```

既然是一个工具类，从 ObjectFactory 的源码中处处可以透出方法的“工具”特征。我们可以将这些纷繁复杂的工具方法进行归纳，总结出其中具有代表性的两类工具方法：一类是 ObjectFactory 用于构建 XWork 框架内部对象 Action、Interceptor、Result 和 Validator 的快捷方法；另外一类则是 ObjectFactory 用于构建一个普通 bean 的核心方法 buildBean。

核心方法 buildBean 中，包含了对象创建和依赖注入这两个核心过程，因而也成为一个统一的对象初始化操作接口。我们可以在源码中发现，无论是哪类工具方法，最终实施对象构建的都是核心的 buildBean 方法。这一点，我们可以对 buildBean 这个方法在 Eclipse 使用 Ctrl+Shift+G 功能，查看其最终被哪些方法所调用来证明。

除了 buildBean 这个基本核心方法之外，ObjectFactory 中 buildAction、buildResult 等功能都在 Struts2 内部用于构建相应的对象。由此，我们还能得到一个非常重要的推论：

结论 全部使用 ObjectFactory 进行框架内置对象的构建保证了所有 XWork 框架中的执行对象都受到 XWork 容器的管理。

这一结论的明确，使我们能够自由地对 XWork 框架各个执行层次上的元素进行扩展而无须担心与容器的通信机制。我们可以编写任何 Interceptor、Result、Action 实例，并且在这些实例中使用 @Inject 引入框架内的一切内置对象供之后使用。

有了 ObjectFactory，我们就可以在实际应用中编写我们自己的 Java 类，并使用 ObjectFactory 来构建这个类的实例。由于 ObjectFactory 在构建对象实例时使用了 XWork 的容器进行依赖注入操作，我们就能将自己的 Java 类与 Struts2 / XWork 的内置对象通过 ObjectFactory 进行关联。因此，**ObjectFactory 成为了自定义 Bean 与 Struts2 的固有组件或者内置对象进行对话的窗口，也是对 Struts2 现有功能进行有机扩展的必要元素。**

ObjectFactory 与 Container 的联系和区别

我们已经接触了 ObjectFactory 和 Container，它们之间既有联系也有区别。在实际应用中，我们应该合理区分它们不同的使用场景。

Container 对象在 Struts2 进行初始化的时候被创建出来，其中包含了所有在 Struts2 应用中定义的对象的构造工厂。从而保证了在运行期，当我们需要获得一个已定义好的内置对象时，可以通过操作 Container 的接口完成。所以，针对 Container 的操作，以“取”为主。

ObjectFactory 对象则偏重于在程序运行过程中的对象构建，并且提供了一个与 Struts2 容器进行交互的窗口。因此，ObjectFactory 创建出来的对象往往是一个运行期的新对象而并非一个“单例”的工具对象，这与我们在容器中定义的许多对象都是不同的。所以，针对 ObjectFactory 的操作，以“建”为主。

5.5 小结

面向对象语言的基本构成元素是对象，对象的生命周期管理是所有面向对象语言的重

要课题。本章从概念入手，通过引入一个额外的编程元素“容器”来帮助我们进行对象生命周期的管理，并且在此基础上分析了 XWork 的容器实现。

容器，不仅支撑起一个框架中所有对象的生命周期，同时也成为框架运行过程中最为重要的一个辅助元素。读完本章，希望读者能够站在编程的角度，对对象生命周期的相关知识有一个更加深刻的理解。

以下的这些问题概括了本章的重点，你是否已经掌握了呢？

- 对象的生命周期管理包含哪两个方面的内容？
- 我们为什么要在面向对象语言中引入容器？
- 什么是依赖注入？
- 一个全局的容器设计，应遵循怎样一些原则？
- XWork 中的容器所管理的是哪些对象？
- 获取 XWork 容器所管理的对象实例，有哪两种方法？
- XWork 容器的内部到底存储了哪些内容？
- @Inject 这个 Annotation 在 XWork 框架中起着什么作用？
- ObjectFactory 有哪两大重要的意义？

第 6 章　灵丹妙药——OGNL，数据流转的催化剂

6.1　架起数据沟通的桥梁——表达式引擎

表达式引擎到底是如何与 Web 开发中的 MVC 模式关联起来的呢？要解开这一疑团，我们还是得回到 MVC 模式自身，重新审视 MVC 模式在运转过程中的困境，再看看引入表达式引擎之后是否能够真正帮助我们解决这些开发中的困境。

6.1.1　数据流转的困境

什么是数据流转的困境？数据为什么会流转？数据的流转又会遇到什么困境？在回答这些问题之前，我们必须首先了解一个事实：

结论　*有一股神秘的力量在 MVC 的各个模块中进行流转，并且它在不同的 MVC 层次中表现出不同的形式和状态。*

我们不妨用一段简单的 Struts2 程序来证明这一点。这里依然选择 Registration 作为业务场景，其源代码如代码清单 6-1、代码清单 6-2 和代码清单 6-3 所示。

代码清单 6-1　User.java

```
public class User {

    private String name;

    private String password;
    // 省略了 setter 与 getter 方法
}
```

代码清单 6-2　user.jsp

```
<form method="post" action="/registration.action">
     user name: <input type="text" name="user.name" value="downpour" />
     password: <input type="password" name="user.password" value="pass" />
     <input type="submit" value="submit" />
</form>
```

代码清单 6-3　UserAction.java

```
public class UserAction implements Action {

    private User user;

    public String execute() {
        // 可以直接在这里使用user对象，因为它已经被作为传入的参数了

        return "success";
    }

    // 省略了setter方法与getter方法
}
```

这是一个简单的 Struts2 入门程序。仔细观察其中的运转细节就会发现，User 是贯穿整个流转过程的数据载体，鉴于它的特殊作用，我们通常将 User 称之为数据模型（Model）。我们发现，User 这个数据模型在不同的 MVC 模块中表现出不同的形式：

❑ **View 层——表现为字符串展现**

在这里，View 层的数据模型将遵循 Http 协议，因而它没有数据类型的概念。所有数据在页面上的表现都是一个个扁平的、不带数据类型的字符串，无论数据结构有多复杂，数据类型有多丰富，到了展示的时候，全都一视同仁地当作字符串在页面上展现出来。数据在传递时，任何数据都被当作字符串或者字符串的数组来进行。

❑ **Controller 层——表现为 Java 对象**

在 Controller 层，数据模型遵循 Java 的语法和数据结构。所有数据载体在 Java 世界中可以表现为丰富的数据结构和数据类型，你可以自行定义喜欢的类，在类与类之间进行继承、嵌套。我们通常会把这种模型称之为对象树。数据在传递时，将以对象的形式进行。

结论　数据在不同的 MVC 层次上，扮演的角色和表现形式不同。这是由于 Http 协议与 Java 的面向对象性之间的不匹配造成的。

如果我们要数据在 View 层（页面）和 Java 世界中互相流转传递，就会在“字符串”与“对象树”之间存在不匹配性。这个不匹配性源于 Web 是一个“弱类型”的平台，Web 的目标是展示内容，而 Java 却是一个具有丰富数据类型的“强类型”的平台，目标是处理复杂的逻辑。同一个对象，在“弱类型”的平台和一个“强类型”的平台之间交互，就必须有一个非常重要的“翻译”角色解决这种“不匹配”。这个角色，就是我们所说的**表达式引擎**，它充当着桥梁作用，保证数据能够顺利地在 MVC 的各个层次进行流转。

Java 对象与其他数据形式的匹配

在我们的日常开发中，不仅要面对 Java 世界的对象，同时还要面对各种其他的数据元素表现形式（XML、HTML、关系型数据库、JSON 等）。这种情况下，我们总是采用一些工具来帮助解决不匹配的问题。例如，我们使用 Hibernate 或者 myBatis 这样的持久层框架来处理 Java 对象与关系型数据库的匹配。

6.1.2 数据访问的困境

数据模型（Model）除了表现出流动性的特征以外，我们同时更加关心数据的内容。在 Web 环境中，我们时不时会需要在 MVC 的任何执行层次对数据的内容进行访问。在上一节中，我们已经讲到，数据模型在 MVC 的不同层次的表现形式以及所遵循的协议都完全不同，因而在 MVC 的不同层次，我们进行数据访问的具体方式也不同。

数据访问的困境，主要还是来源于数据模型在某些层次的展现缺乏足够的表现力。例如，在 View 层，既没有数据类型的概念，也没有数据结构的概念。无论数据自身的结构有多复杂，一旦对外展现，都最终转化为统一的字符串形式。在这种情况下，我们就需要建立一种符号化的规则，使之与复杂而丰富的 Java 类型对应起来，这种规则就是表达式引擎的雏形。

在数据流转的过程中，我们可以看到两个截然不同的方向：

❑ 从 View 层到 Controller 层

这个方向的数据流转所强调的，是能够保证一个扁平而分散的数据能以一定的规则设置到 Java 世界的对象树中去。同时，能够聪明地进行由字符串类型到 Java 中各个类型的转化。

❑ 从 Controller 到 View 层

这个方向的数据流转所强调的，是保证在 View 层能够以某些简易的规则对 Java 的对象树进行访问。同时，在一定程度上控制对象树中的数据的显示格式。

我们这里所说的数据访问，其实指的就是数据从 Controller 层流转到 View 层的过程。在这个过程中，表达式引擎所起的作用，就是建立起访问规则与 Java 对象树之间的联系。在这个方面有不少成熟的方案，如 JSTL、模板语言等等，但无论是什么样的方案，其实现的基础都是一致的，那就是表达式引擎。

6.1.3 表达式引擎

从我们对所遇到的 Web 开发中的困境的讨论中，我们已经看到了在 Web 开发中引入表达式引擎的重要性。那么，表达式引擎又是如何工作的呢？页面上的参数到底是如何与

我们定义的 Java 类关联起来并实现交互的呢？我们以上一节的例子为基础，分析一下它们的存在形式，如表 6-1 所示。

表 6-1　页面参数与 Action 中 User 类的映射关系

页面中的参数名	页面中的参数值	User 类的字段（类型）	User 类中的字段值
user.name	Downpour	name (String)	Downpour
user.password	Pass	password (String)	Pass
user.birthday	1982-03-23	birthday (java.util.Date)	1982-03-23

实际上，第 2 列和第 4 列的内容是相同的，它们在本质上都是“值”。唯一的区别在于，第 2 列的值的存在空间是页面，因而无法为其定义所谓的数据类型，它们看上去都表现为字符串形式，而第 4 列的值的存在空间是 Java 类，此时，可以为之定义我们所需要的任何数据类型。

我们重点来关注一下第 1 列和第 3 列。如果把它们单独列出来，实际上我们所实现的是一种转化，这种转化存在于一个**规则化的表达式和 Java 对象之间**：

- user.name——user **实例的** name **字段**
- user.password——user **实例的** password **字段**

此时，我们就需要引入一个概念 —— **表达式引擎**。它的作用就是**帮助我们完成这种规则化的表达式与 Java 对象的互相转化**。从上面的例子也可以看出，表达式引擎在其中起到了非常关键的穿针引线的作用。

结论　*表达式引擎在 Web 开发中能够完成规则化字符串表达式与 Java 对象之间的互相转化，因而它成为架起 MVC 各个模块之间数据沟通的桥梁。*

发挥我们的主观能动性来思考一下，数据在交互的过程中到底会遇到什么样的困境，这些困境将成为我们选择和使用表达式引擎的依据：

- **表达式引擎应该能处理表达式与对象之间的映射关系，这种映射关系应是双向的。**
- **表达式引擎应该能支持丰富多样的表达式语法计算。**
- **表达式引擎应该能支持必要的数据类型转换。**

在 Java 世界中，有许多优秀的表达式引擎。OGNL 在内部实现机制和设计原理上有许多亮点，因而 Struts2 选择了 OGNL 作为其依赖的表达式引擎，并在 OGNL 的基础上做了一定的扩展，使 OGNL 的应用更加丰富。接下来我们将首先分析一下 OGNL 的实现细节，并在之后的章节中向读者展示 Struts2 是如何对 OGNL 进行进一步扩展的。

6.2 强大的 OGNL

OGNL (Object Graph Navigation Language) 是一个开源的表达式引擎。通过使用 OGNL，我们能够通过表达式存取 Java 对象树中的任意属性和调用 Java 对象树的方法等。也就是说，如果我们把表达式看成是一个带有语义的字符串，那么 OGNL 就是这个语义字符串与 Java 对象之间沟通的催化剂。通过 OGNL，我们可以轻松解决在数据流转过程中所遇到的各种问题。

6.2.1 深入 OGNL 的 API

我们采用最直观的方式：通过研究 OGNL 的原生 API 来看看如何使用 OGNL 进行对象的存取操作。首先看一下来自 OGNL 的静态方法，如代码清单 6-4 所示。

代码清单 6-4 Ognl.java

```
    /**
     * 通过传入的 OGNL 表达式，在给定的上下文环境中，从 root 对象里取值
     */
    public static Object getValue(String expression, Map context, Object root)
    throws OgnlException {
        return getValue(expression, context, root, null);
    }

    /**
     * 通过传入的 OGNL 表达式，在给定的上下文环境中，往 root 对象里写值
     */
    public static void setValue(String expression, Map context, Object root, Object
value) throws OgnlException {
        setValue(parseExpression(expression), context, root, value);
    }
```

OGNL 的 API 其实相当简单。上面的 2 个方法分别针对对象的“取值”和“写值”操作。因而，OGNL 的基本操作实际上是通过传入上述这 2 个方法的 3 个参数来实现的。OGNL 同时编写了许多其他的方法来实现相同的功能，上述的 2 个接口只是其中最简单并最具代表性的 2 个方法。读者可以通过阅读 Ognl.java 来获取更多的信息。

在初步浏览了 OGNL 的 API 后，我们可以编写一个单元测试来测试一下上面列出的 OGNL 静态方法接口，实现如代码清单 6-5 所示。

代码清单 6-5 BasicOgnlTest.java

```
    public class BasicOgnlTest extends TestCase {
```

```
    @SuppressWarnings("unchecked")
    @Test
    public void testGetValue() throws Exception {
        // 创建 Root 对象
        User user = new User();
        user.setId(1);
        user.setName("downpour");

        // 创建上下文环境
        Map context = new HashMap();
        context.put("introduction","My name is ");

        // 测试从 Root 对象中进行表达式计算并获取结果
        Object name =
    Ognl.getValue(Ognl.parseExpression("name"), user);
        assertEquals("downpour",name);

        // 测试从上下文环境中进行表达式计算并获取结果
        Object contextValue = Ognl.getValue(Ognl.parseExpression("#introduction"),
context, user);
        assertEquals("My name is ", contextValue);

        // 测试同时从将 Root 对象和上下文环境作为表达式的一部分进行计算
        Object hello = Ognl.getValue(Ognl.parseExpression("#introduction +
    name"), context, user);
        assertEquals("My name is downpour", hello);
    }

    @Test
    public void testSetValue() throws Exception {
        // 创建 Root 对象
        User user = new User();
        user.setId(1);
        user.setName("downpour");

        // 对 Root 对象进行写值操作
        Ognl.setValue("group.name", user, "dev");
        Ognl.setValue("age", user, "18");

        assertEquals("dev", user.getGroup().getName());
    }
}
```

我们可以看到，通过简单的 API 就能够完成对各种对象树的“取值”和“写值”操作。而“取值”和“写值”是我们日后所有工作的基础，如果要深入了解 OGNL 的细节，就需要对传入 OGNL 的这 3 个参数进行研究。这 3 个参数，我们称之为 **OGNL 的三要素**。在下一节中，我们会对 OGNL 的三要素做具体的解释。

OGNL 的 API 极其简单，无论何种复杂的功能，OGNL 会将其最终映射到 OGNL 的三要素中，通过调用底层引擎完成计算。OGNL 对于其构成要素的设计思路，完全契合了我们对表达式引擎的要求，因而也成为众多表达式引擎设计的一种标准。如果我们翻开其他的一些著名的表达式引擎，同样可以看到这些构成要素的身影。

以 Spring 框架所发布的内置表达式引擎 SpringEL 为例，我们可以在其核心的 Expression 操作接口中看到完全相同的构成要素定义。如图 6-1 所示。

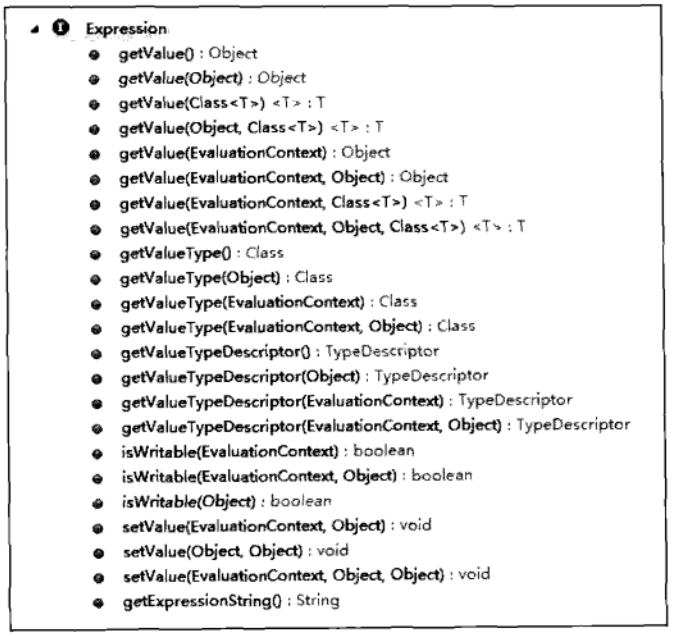


图 6-1 SpringEL 的 Expression 接口

读者在这里应该仔细品味表达式引擎自身的特性和构成要素之间的联系和共同点，领略其中的设计精髓并熟练运用到实际开发中去。

6.2.2 OGNL 三要素

从上一节的例子中我们可以看到，每进行一次 OGNL 操作都需要 3 个参数。OGNL 的所有操作实际上都是围绕着这 3 个参数而进行的。这 3 个参数被称为 OGNL 的三要素。

6.2.2.1　表达式

表达式（Expression）是整个 OGNL 的核心，所有 OGNL 操作都是针对表达式解析后进行的。表达式会规定此次 OGNL 操作到底要“**干什么**”。因此，表达式其实是一个带有语法含义的字符串，这个字符串将规定操作的类型和操作的内容。

OGNL 支持大量的表达式语法，不仅支持“链式”描述对象访问路径，还支持在表达式中进行简单的计算，甚至还能够支持复杂的 Lambda 表达式等。我们可以在接下来的章节中看到各种各样不同的 OGNL 表达式。

6.2.2.2　Root 对象

OGNL 的 Root 对象（Root Object）可以理解为 OGNL 的操作对象。当 OGNL 表达式规定了“干什么”以后，我们还需要指定“**对谁干**”。OGNL 的 Root 对象实际上是一个 Java 对象，是所有 OGNL 操作的实际载体。这就意味着，如果有一个 OGNL 的表达式，那么我们实际上需要针对 Root 对象来进行 OGNL 表达式的计算并返回结果。

6.2.2.3　上下文环境

有了表达式和 Root 对象，我们已经可以使用 OGNL 的基本功能了。例如，根据表达式针对 OGNL 中的 Root 对象进行“取值”或者“写值”操作。不过，事实上，在 OGNL 的内部，所有的操作都会在一个特定的数据环境中运行，这个数据环境就是 OGNL 的上下文环境（Context）。说得再明白一些，就是这个上下文环境将规定 OGNL 的操作“**在哪里干**”。

OGNL 的上下文环境是一个 Map 结构，称之为 OgnlContext。之前我们所提到的 Root 对象，事实上也会被添加到上下文环境中，并且被作为一个特殊的变量进行处理。

6.2.3　OGNL 的基本操作

6.2.3.1　对 Root 对象的访问

针对 OGNL 的 Root 对象的对象树的访问是通过使用“点号”将对象的引用串联起来实现的。通过这种方式，OGNL 实际上将一个树形的对象结构转化成一个链式结构的字符串来表达语义。如下所示：

```
// 获取 Root 对象中的 name 属性的值
name
// 获取 Root 对象 department 属性中的 name 属性的实际值
department.name
// 获取 Root 对象 department 属性的 manager 属性中 name 属性的实际值
department.manager.name
```

6.2.3.2 对上下文环境的访问

由于 OGNL 的上下文是一个 Map 结构，在 OGNL 进行计算时可以事先在上下文环境中设置一些参数，并让 OGNL 将这些参数带入进行计算。有时候也需要对这些上下文环境中的参数进行访问，访问这些参数时，需要通过 # 符号加上链式表达式来进行，从而表示与访问 Root 对象的区别。如下所示：

```
// 获取 OGNL 上下文环境中名为 introduction 的对象的值
#introduction
// 获取 OGNL 上下文环境中名为 parameters 的对象中 user 对象中名为 name 的属性的值
#parameters.user.name
```

6.2.3.3 对静态变量的访问

在 OGNL 中，对于静态变量或者静态方法的访问，需要通过 @[class]@[field / method] 的表达式语法来进行。如下所示：

```
// 访问 com.example.core.Resource 类中名为 ENABLE 的属性值
@com.example.core.Resource@ENABLE
// 调用 com.example.core.Resource 类中名为 get 的方法
@com.example.core.Resource@get()
```

6.2.3.4 方法调用

在 OGNL 中调用方法，可以直接通过类似 Java 的方法调用方式进行，也就是通过点号加方法名称完成方法调用，甚至可以传递参数。如下所示：

```
// 调用 Root 对象中的 group 属性中 users 的 size() 方法
group.users.size()
// 调用 Root 对象中 group 中的 containsUser 的方法，并将上下文环境中名为
requestUser 的值作为参数传入
group.containsUser(#requestUser)
```

6.2.3.5 使用操作符进行简单计算

OGNL 表达式中能使用的操作符基本与 Java 里的操作符一样，除了能使用 +、-、*、/、++、--、== 等操作符之外，还能使用 mod、in、not in 等。如下所示：

```
2+4 // 加

'hello' + 'world'' // 字符串叠加

5-3 // 减

9/2 // 除

9 mod 2 // 取模
```

```
foo++ // 递增

foo == bar // 等于判断

foo in list // 是否在容器中
```

6.2.3.6　对数组和容器的访问

OGNL 表达式可以支持对数组按照数组下标的顺序进行访问。同样的方法可以用于有序的容器，如 ArrayList、LinkedHashSet 等。对于 Map 结构，OGNL 支持根据键值进行访问。如下所示：

```
// 访问 Root 对象的 group 属性中 users 的第一个对象的 name 属性值
group.users[0].name
// 访问 OGNL 上下文中名为 sessionMap 的 Map 对象中 key 为 currentLogonUser 的值
#sessionMap['currentLogonUser']
```

6.2.3.7　投影与选择

OGNL 支持类似于数据库中的投影（projection）和选择（selection）功能。

投影是指选出集合中每个元素的相同属性组成新的集合，类似于关系数据库的字段操作。投影操作语法为 collection.{XXX}，其中 XXX 是这个集合中每个元素的公共属性。

选择就是过滤满足 selection 条件的集合元素，类似于关系数据库的结果集操作。选择操作的语法为：collection.{X YYY}，其中 X 是一个选择操作符，后面则是选择用的逻辑表达式。选择操作符有三种：

- ? 选择满足条件的所有元素
- ^ 选择满足条件的第一个元素
- $ 选择满足条件的最后一个元素

如下所示：

```
// 返回 Root 对象的 group 属性中 users 这个集合中所有元素的 name 构成的集合

group.users.{name}    // 新的以 name 为元素的集合

// 将 group 中 users 这个集合中的元素的 code 和 name 用 - 连接符拼起来构成的字符串集合

group.users.{code + '-' + name}    // 新的以 'code - name' 为元素的集合

// 返回 Root 对象的 group 中 users 这个集合所有元素中 name 不为 null 的元素构成的集合

group.users.{? #this.name != null}  // 过滤后的 users 集合
```

6.2.3.8　构造对象

OGNL 支持直接通过表达式来构造对象。构造的方式主要包括 3 种：

- **构造 List：使用 {}，中间使用逗号隔开元素的方式表达列表**
- **构造 Map ：使用 #{}，中间使用逗号隔开键值对，并使用冒号隔开 key 和 value 来构造 Map**
- **构造对象：直接使用已知对象的构造函数来构造对象**

构造对象的方法如下所示：

```
// 构造一个 List
{"green", "red", "blue"}
// 构造一个 Map
#{"key1" : "value1", "key2" : "value2", "key3" : "value3"}
// 构造一个 java.net.URL 对象
new java.net.URL("http://localhost/")
```

构造对象对于表达式语言来说是一个非常强大的功能，OGNL 不仅能够直接对容器对象构造提供语法层面的支持，还能够对任意的 Java 对象提供支持。这样一来就使得 OGNL 不仅仅具备了数据运算这一简单的功能，同时还被赋予了潜在的逻辑计算功能。

OGNL 带来的潜在问题

我们已经能够看到 OGNL 在语法层面所表现出来的强大之处。然而，越强大的东西，其自身也一定存在着致命的弱点，这也就是所谓的“物极必反”。正是由于 OGNL 能够支持完整的 Java 对象创建、读写过程，它就能被作为一个潜在的切入点，成为黑客的攻击目标。exploit-db 网站在 2010 年 7 月 14 日就爆出了一个 Struts2 的远程执行任意代码的漏洞。具体的声明链接为：http://www.exploit-db.com/exploits/14360/。

细心的读者会发现，这个漏洞的基本原理实际上就是利用了 OGNL 可以任意构造对象，并执行对象中方法的特性，构造了一个底层命令调用的 Java 类，并执行操作系统命令进行系统攻击。

在 Struts2.2.X 之后的版本中，这个漏洞被修复，修复方法主要是通过限制参数名称的方式，拒绝类似的代码执行。

6.2.4 深入 this 指针

我们知道，OGNL 表达式是以点进行串联的一个链式字符串表达式。而这个表达式在进行计算的时候，从左到右，表达式每一次计算返回的结果成为一个临时的“**当前对象**”，并在此临时对象之上继续进行计算，直到得到计算结果。而这个临时的“当前对象”会被存储在一个叫做 this 的变量中，这个 this 变量就称为 **this 指针**。

各种编程语言的 this 指针

this 指针是许多编程语言都具备的特殊关键字。绝大多数语言中的 this 指针的含义都类似，表示“当前所在函数的调用者”。无论一个表达式有多么复杂，只要读者能够仔细分析 this 指针所在的函数，并找到这个函数的调用者，就能很容易地找到 this 指针所指向的内容了。

在 OGNL 表达式中的 this 指针，**无疑指向了当前计算的“调用者”对应的实例**。如果读者从“调用者”这个角度来理解 this 指针，那么这个概念就能够被消化和理解了。需要注意的是，如果试图在表达式中使用 this 指针，需要在 this 之前加上 #，我们来看下面的例子。

```
// 返回 group 中 users 这个集合中所有 age 比 3 大的元素构成的集合
users.{? #this.age > 3}
// 返回 group 中 users 这个集合里的大小 +1 的值
group.users.size().(#this+1)
// 返回 Root 对象的 group 中 users 这个集合所有元素中 name 不为 null 的元素构成的集合
group.users.{? #this.name != null}
```

this 指针在 lambda 表达式中运用极为广泛，通过 this 指针，我们能够写出许多简单而又蕴含着复杂逻辑的 OGNL 表达式，大家可以在实践中慢慢领悟其中的奥妙。

6.2.5　有关 # 符号的三种用途

在之前的表达式范例中，我们已经了解了“#”操作符的几种不同用途。这是一个非常容易混淆的知识点，所以有必要在这里详细解释一下。

- **加在普通 OGNL 表达式前面，用于访问 OGNL 上下文中的变量**
- **使用 #{} 语法动态构建 Map**
- **加在 this 指针之前表示对 this 指针的引用**

这 3 种不同的用途在不同的地方有着不同的妙用，尤其是对 OGNL 上下文中的变量的访问，将成为 Struts2 在页面级别进行容器变量访问的重要理论基础。

6.3　深入 OGNL 内部

在上一节中，我们介绍了 OGNL 作为一个出色的表达式引擎在语法层面所表现出来的强大之处。正如一个强大的编程语言需要一个全面而精湛的底层实现机制一样，OGNL 在实现机理上也同样非常优秀。在本节中，我们就来试图揭秘 OGNL 的内部实现。

6.3.1　深入 OgnlContext

在 OGNL 的三要素中，OGNL 的上下文环境为 OGNL 提供了计算的场所，是

OGNL 计算的运行环境。这个运行环境在 OGNL 内部有一个实际的对象与之对应，即 OgnlContext，它是一个 Map 结构。

OGNL 的上下文环境实际上也参与到了 OGNL 表达式的计算中，这一点在上一节 OGNL 操作的讲解中已经给出了范例。OGNL 在针对其上下文环境的表达式计算上与针对 Root 对象的表达式计算之间存在着 OGNL 表达式写法上的差异，而从计算规则的角度上来说并没有任何区别。

在上一节中，我们还提到 OGNL 的上下文环境中实际上包含了许多参数设置，这些参数指定了 OGNL 在进行计算时使用的一些默认行为和默认值，同时这些参数也会参与到 OGNL 的计算中。

接下来，我们对 OgnlContext 的研究就从它的数据结构开始。从源码中，我们可以看到 OgnlContext 自身实现了 java.util.Map 接口。它与普通的 Map 实现不同的地方在于，OgnlContext 在内部维护的许多 Map 是 OGNL 在进行表达式计算时的参数，它们将作为 OGNL 的计算基础。这些参数在 OgnlContext 初始化的时候就会被指定。有关 OgnlContext 数据结构的部分源码，如代码清单 6-6 所示。

代码清单 6-6 OgnlContext.java

```
/**
 * 根据传入参数创建 OgnlContext，如果传入 Map 作为上下文，传入的 Map 会被封装于内部

 * @param root
 *          Root 对象
 * @param classResolver
 *          用于处理 class loading 的处理类
 * @param converter
 *          用于处理类型转化的处理类
 * @param memberAccess
 *          用于处理属性访问策略的处理类
 * @param context
 *          默认的上下文，如果不是 OgnlContext 类型，会被包装成 OgnlContext
 * @return
 *          返回 OGNL 上下文，并正确设置好 Root 对象
 */
public static Map addDefaultContext(Object root, ClassResolver
classResolver, TypeConverter converter, MemberAccess memberAccess,
Map context){
    // 定义 OgnlContext
    OgnlContext result;

    // 如果传入的 context 不是 OgnlContext 类型，则将传入的 Map 封装成
    // OgnlContext，并且将传入的 Map 中的所有键值对维护在 OgnlContext 内部
      if (!(context instanceof OgnlContext)) {
              result = new OgnlContext();
          result.setValues(context);
```

```
        } else {
            result = (OgnlContext) context;
        }

        // 设置 OGNL 计算时的默认行为
        if (classResolver != null) {
            result.setClassResolver(classResolver);
        }
        if (converter != null) {
            result.setTypeConverter(converter);
        }
        if (memberAccess != null) {
            result.setMemberAccess(memberAccess);
        }

        // 设置 Root 对象
        result.setRoot(root);
        return result;
    }
```

从上面的代码中我们可以看到，OgnlContext 会在 OGNL 进行计算时才动态生成。无论我们是否需要使用自己的 Map 来覆盖 OGNL 的上下文，OGNL 都会创建一个 OgnlContext 加以包装，这其实是一个典型的装饰（Wrapper）模式的应用。在创建 OgnlContext 时，Map 将作为参数传入，并维护在 OgnlContext 内部。同时，OgnlContext 在创建时还需要根据传入的参数设定 OGNL 计算的一些默认行为，并设置好 Root 对象。

继续深入探究 OgnlContext 的内部结构，我们会发现在 OgnlContext 的内部同时维护着许多内部变量。这些变量在 OGNL 计算时起到了很大的作用。在这里列举其中几个比较重要的内部变量并进行一定的分析，如代码清单 6-7 所示。

代码清单 6-7　OgnlContext.java

```
public class OgnlContext extends Object implements Map {

// Root 对象的引用
private Object _root;

// 通过一个 Map 来维护 Ognl 的上下文环境
private Map _values = new HashMap(23);

// 定义一些默认的 OGNL 行为
public static final ClassResolver DEFAULT_CLASS_RESOLVER = new
DefaultClassResolver();
public static final TypeConverter DEFAULT_TYPE_CONVERTER = new
DefaultTypeConverter();
public static final MemberAccess DEFAULT_MEMBER_ACCESS = new
```

```
    DefaultMemberAccess(false);

    // 设置 OGNL 计算时所使用的行为处理类
    private ClassResolver _classResolver = DEFAULT_CLASS_RESOLVER;
    private TypeConverter _typeConverter = DEFAULT_TYPE_CONVERTER;
    private MemberAccess _memberAccess = DEFAULT_MEMBER_ACCESS;

    // 此处省略了许多其他代码

    }
```

从源码中，我们可以看到封装于 OgnlContext 内部的一些对象，有的成为 OGNL 的操作对象、有的成为 OGNL 计算中的计算规则处理类：

❑ _root

在 OgnlContext 内部维护着的 Root 对象，它是 OGNL 主要的操作对象。

❑ _values

如果希望在 OGNL 计算时使用传入的 Map 作为上下文环境，OGNL 依旧会创建一个 OgnlContext，并将所传入的 Map 中所有的键值对维护在 _values 变量中。这个变量就被看作真正的容器，并在 OGNL 的计算中发挥作用。

❑ ClassResolver

指定处理 class loading 的处理类。实际上这个处理类是用于指定 OGNL 在根据 Class 名称来构建对象时，寻找 Class 名称与对应的 Class 类之间对应关系的处理方式。在默认情况下，会使用 JVM 的 class.forName 机制来实现。

❑ TypeConverter

指定处理类型转化的处理类。这个处理类非常关键，它会指定当一个对象属性转化成字符串以及字符串转化成 Java 对象时的处理方式。

❑ MemberAccess

指定处理属性访问策略的处理方式。

6.3.2 深入 OGNL 的计算规则

正所谓“没有规矩，不成方圆”，OGNL 在计算过程中需要遵循许多行为规则。这些行为规则主要被定义在两个地方：OgnlContext 和 OgnlRuntime。所有这些行为规则的集合就构成了 OGNL 的计算规则，也就是我们所说的“规矩”。

在上一节对 OgnlContext 的源码进行分析的过程中已经看到，OGNL 规定了计算时的几个默认行为，例如 ClassResolver、TypeConverter、MemberAccess 等。除此之外，我们还需要针对 OGNL 的运行状态建立一些行为准则，而这些内容设置在 OgnlRuntime 中。

表 6-2 列举了 OGNL 计算时所需要遵循的一些重要的计算规则和默认实现类。

表 6-2　OGNL 计算时需要遵循的重要计算规则及其对应的实现类

关键字	默认实现
ClassResolver (OgnlContext)	OGNL：ognl.DefaultClassResolver Struts2：com.opensymphony.xwork2.ognl.accessor.CompoundRootAccessor
TypeConverter (OgnlContext)	OGNL：ognl.DefaultTypeConverter Struts2：com.opensymphony.xwork2.conversion.OgnlTypeConverterWrapper
MemberAccess (OgnlContext)	OGNL：ognl.DefaultMemberAccess Struts2：com.opensymphony.xwork2.ognl.SecurityMemberAccess
MethodAccessor (OgnlRuntime)	OGNL：ognl.ObjectMethodAccess Struts2：com.opensymphony.xwork2.ognl.accessor.CompoundRootAccessor com.opensymphony.xwork2.ognl.accessor.XWorkMethodAccessor
PropertyAccessor (OgnlRuntime)	OGNL：ognl.ObjectPropertyAccess Struts2：com.opensymphony.xwork2.ognl.accessor.CompoundRootAccessor com.opensymphony.xwork2.ognl.accessor.ObjectAccessor
NullHandler (OgnlRuntime)	OGNL：ognl.ObjectNullHandler Struts2：com.opensymphony.xwork2.ognl.OgnlNullHandlerWrapper

从表 6-2 中可以发现，OGNL 在设计的时候已经充分考虑了扩展性，所有的计算规则被定义成接口，并且允许在运行期替换它的实现。

6.3.2.1　类的寻址方式——ClassResolver

ClassResolver 指定 OGNL 在进行计算时，根据字符串进行 Java 类寻址的处理方式。在默认情况下，输入的字符串是 Class 名称，而内置的实现方式则是通过 Java 中基本的反射原理根据 Class 名称返回与之对应的 Java 类。ClassResolver 的接口定义如代码清单 6-8 所示。

代码清单 6-8　ClassResolver.java

```
public interface ClassResolver {

    public Class classForName(String className, Map context) throws
ClassNotFoundException;

}
```

从其接口定义中我们就可以看到，ClassResovler 通过接收一个字符串作为 Java 类寻址的依据，而返回值是经过寻址后得到的与之对应的 Java 类。

OGNL 中的默认实现是 ognl.DefaultClassResolver，使用了 JVM 的 class.forName 机制来实现。而在 Struts2 中，为了处理一些特殊支持的 class 名称，例如 vs、vs2、vs3 等，采用了 com.opensymphony.xwork2.ognl.accessor.CompoundRootAccessor 作为实现方式，

如代码清单 6-9 所示。

代码清单 6-9 CompoundRootAccessor.java

```
public Class classForName(String className, Map context) throws
ClassNotFoundException {
        // 获取 OGNL 的 Root 对象
   Object root = Ognl.getRoot(context);

     try {
            // 如果 Root 对象是 CompoundRoot 类型，则需要处理特殊名称的 className
         if (root instanceof CompoundRoot) {
             if (className.startsWith("vs")) {
                 CompoundRoot compoundRoot = (CompoundRoot) root;
                // 支持直接使用 vs 作为 className
                 if ("vs".equals(className)) {
                     return compoundRoot.peek().getClass();
                 }
                // 支持使用 vs2、vs3 等形式作为 className
                 int index = Integer.parseInt(className.substring(2));

                 return compoundRoot.get(index - 1).getClass();
             }
         }
     } catch (Exception e) {
         // 仅仅是尝试一下旧式方法
     }

     return
Thread.currentThread().getContextClassLoader().loadClass(className);
}
```

CompoundRootAccessor 这个类是多重接口的实现类，是 Struts2 对 OGNL 的主要扩展。之后我们将反复看到这个类在 OGNL 计算中扮演着重要角色，有关这个类的详细分析，我们将在下一章具体展开。

6.3.2.2 类型转化方式——TypeConverter

TypeConverter 的作用是指定在 OGNL 计算的过程中如果需要进行类型转化应该遵循的实现方式。在 OGNL 的默认实现中，采用了 ognl.DefaultTypeConverter，它能够支持基本类型的转化，有兴趣的读者可以参照 ognl.OgnlOps 类的静态方法 convertValue，我们在这里不详细展开这个类源码实现，因为在 XWork 的实际使用中这个默认实现被替代了。不过我们在这里还是可以参阅一下 TypeConverter 的接口定义，如代码清单 6-10 所示。

代码清单 6-10 TypeConverter.java

```
public interface TypeConverter {
```

```
    public Object convertValue(Map context, Object target, Member member,
String propertyName, Object value, Class toType);

}
```

鉴于 TypeConverter 的特殊性，它几乎是最需要进行用户扩展的一个 OGNL 运行参数，因而 XWork 在使用它时建立了自己的 TypeConverter 接口，其名称和接口函数定义完全相同。XWork 自行定义 TypeConverter 接口的目的在于对外屏蔽类型转化的实现细节，从而能够将 XWork 对 TypeConverter 的扩展实现纳入到 XWork 的容器中管理，从而方便对 OGNL 原始的 TypeConverter 接口进行扩展并支持更加广泛的类型转化逻辑。

XWork 对类型转化方式 TypeConverter 的默认实现类是 XWorkConverter，并使用了一个装饰模式，将其接口的实现封装在 OgnlTypeConverterWrapper 的内部。这样一来，我们就可以完全利用装饰模式为我们带来的好处：不仅屏蔽原始的 TypeConverter 默认实现，并且有利于 XWork 自身在装饰类中插入框架所提供的用户扩展功能。OgnlTypeConverterWrapper 的源码如代码清单 6-11 所示。

代码清单 6-11　OgnlTypeConverterWrapper.java

```
public class OgnlTypeConverterWrapper implements ognl.TypeConverter {

    private TypeConverter typeConverter;

    public OgnlTypeConverterWrapper(TypeConverter conv) {
        if (conv == null) {
        throw new IllegalArgumentException("Wrapped type converter
cannot be null");
    }
    this.typeConverter = conv;
  }

  public Object convertValue(Map context, Object target, Member member,
        String propertyName, Object value, Class toType) {
    // 调用 XWork 内部实现的 TypeConverter，支持自定义的类型转化方式
      return typeConverter.convertValue(context, target, member,
propertyName, value, toType);
  }

  public TypeConverter getTarget() {
      return typeConverter;
  }
}
```

从源码中我们可以看到，被 OgnlTypeConverterWrapper 所装饰的实际实现类已经是一个 XWork 框架中的 TypeConverter。而这个 TypeConverter 接口就成为我们自由扩展自

定义类型转化方式的窗口。

在 XWork 框架中，对 TypeConverter 的默认实现是 DefaultTypeConverter，它实现了最为基本的类型转化方式，其相关源码如代码清单 6-12 所示。

代码清单 6-12 DefaultTypeConverter.java

```
public class DefaultTypeConverter implements TypeConverter {

    private static final String NULL_STRING = "null";

    private final Map<Class, Object> primitiveDefaults;

    public DefaultTypeConverter() {
        // 注册默认支持的基本类型
        Map<Class, Object> map = new HashMap<Class, Object>();
        map.put(Boolean.TYPE, Boolean.FALSE);
        map.put(Byte.TYPE, Byte.valueOf((byte) 0));
        map.put(Short.TYPE, Short.valueOf((short) 0));
        map.put(Character.TYPE, new Character((char) 0));
        map.put(Integer.TYPE, Integer.valueOf(0));
        map.put(Long.TYPE, Long.valueOf(0L));
        map.put(Float.TYPE, new Float(0.0f));
        map.put(Double.TYPE, new Double(0.0));
        map.put(BigInteger.class, new BigInteger("0"));
        map.put(BigDecimal.class, new BigDecimal(0.0));
        primitiveDefaults = Collections.unmodifiableMap(map);
}

/**
     * 根据 context 获取对应的 TypeConverter
 */
public TypeConverter getTypeConverter( Map<String, Object> context )
    {
        Object obj =
context.get(TypeConverter.TYPE_CONVERTER_CONTEXT_KEY);
        if (obj instanceof TypeConverter) {
            return (TypeConverter) obj;

        // 用于向后兼容
        } else if (obj instanceof ognl.TypeConverter) {
            return new XWorkTypeConverterWrapper((ognl.TypeConverter) obj);
        }
        return null;
    }

    /**
     * 根据 Class 类型，返回类型转换后的值
     *
```

```
     *
     * @param value
     * @param toType
     *  @return
     */
    public Object convertValue(Object value, Class toType) {
        Object result = null;

        if (value != null) {
            // 如果是数组，依次对数组中的每个元素进行类型转化
            if (value.getClass().isArray() && toType.isArray()) {
                Class componentType = toType.getComponentType();

                result = Array.newInstance(componentType, Array
                        .getLength(value));
                for (int i = 0, icount = Array.getLength(value); i < icount; i++) {
                    Array.set(result, i, convertValue(Array.get(value, i),
                            componentType));
                }
            } else {
                // 依次判断当前传入的 toType 类型是否为基本类型，并调用相应的转化函数
                if ((toType == Integer.class) || (toType == Integer.TYPE))
                    result = Integer.valueOf((int) longValue(value));
                if ((toType == Double.class) || (toType == Double.TYPE))
                    result = new Double(doubleValue(value));
                if ((toType == Boolean.class) || (toType == Boolean.TYPE))
                    result = booleanValue(value) ? Boolean.TRUE:
Boolean.FALSE;
                if ((toType == Byte.class) || (toType == Byte.TYPE))
                    result = Byte.valueOf((byte) longValue(value));
                if ((toType == Character.class) || (toType ==
Character.TYPE))
                    result = new Character((char) longValue(value));
                if ((toType == Short.class) || (toType == Short.TYPE))
                    result = Short.valueOf((short) longValue(value));
                if ((toType == Long.class) || (toType == Long.TYPE))
                    result = Long.valueOf(longValue(value));
                if ((toType == Float.class) || (toType == Float.TYPE))
                    result = new Float(doubleValue(value));
                if (toType == BigInteger.class)
                    result = bigIntValue(value);
                if (toType == BigDecimal.class)
                    result = bigDecValue(value);
                if (toType == String.class)
                    result = stringValue(value);
                if (Enum.class.isAssignableFrom(toType))
                    result = enumValue((Class<Enum>)toType, value);
            }
        } else {
```

```
            //  如果是基本类型，直接从定义好的map中获取结果
            if (toType.isPrimitive()) {
                result = primitiveDefaults.get(toType);
            }
        }
        return result;
    }

    // 注意这里对enum类型的转化，这是Struts2.1版本之后支持的功能
    public Enum<?> enumValue(Class toClass, Object o) {
        Enum<?> result = null;
        if (o == null) {
            result = null;
        } else if (o instanceof String[]) {
            result = Enum.valueOf(toClass, ((String[]) o)[0]);
        } else if (o instanceof String) {
            result = Enum.valueOf(toClass, (String) o);
        }
        return result;
    }

    // 这里省略了许多其他的基本类型转化函数

}
```

从源码中我们很容易发现，XWork已经默认处理了绝大多数我们日常编程中最常用的Java类型。虽然从代码上看这样的处理方式不够优雅，因为它使用了一堆if/else if，而不是将不同的类型转化职责分派到不同的执行实现类上（虽然在这里可以采用策略模式进行代码实现上的改进，但是考虑到绝大多数类型转化实现非常简单，我们也就没有必要单独创建一个策略类来实现了），然而它却是一个“大而全”的完整解决方案，能够帮助我们解决开发中所遇到的大部分问题。

细心的读者或许会问，在默认实现方式中，我们并没有看到XWork对用户自定义的类型转化方式进行处理的细节，那么我们又如何在XWork中注册扩展的TypeConverter呢？这个问题，我们将留在后续章节展开。

有关类型转化在各种表达式引擎中的实现比较

从对XWork的DefaultTypeConverter的源码分析中，我们可以看到其实现并没有采取“策略模式”来实现，而是使用了多重条件判断。我们在这里强烈推荐读者自行比较Spring3.0引入的SpringEL在类型转化上的实现方式。SpringEL在实现机理上使用了不同的实现类来对应不同方向的类型转化，在一定程度上其扩展性远远好于XWork的默认实现。

6.3.2.3　方法 / 属性访问策略——MemberAccess

MemberAccess 规定了 OGNL 的对象方法 / 属性访问策略，规定了 OGNL 在进行计算时，是否可以访问某个对象的方法或属性。MemberAccess 的源码如代码清单 6-13 所示。

代码清单 6-13　MemberAccess.java

```
public interface MemberAccess
{
    /**
     *  为 Member 设置访问策略
     */
    public Object setup(Map context, Object target, Member member, String
propertyName);

    /**
     * 恢复原始的 Member 的访问策略
     */
    public void restore(Map context, Object target, Member member, String
propertyName, Object state);

/**
 * 判断当前 Member 是否具备访问权限
 */
   public boolean isAccessible(Map context, Object target, Member
member, String propertyName);
}
```

在 OGNL 中，默认实现使用了 ognl.DefaultMemberAccess，其中禁止访问 private / protected / package protected 的属性。而在 XWork / Struts2 中，对这一对象属性访问策略做了一定的扩展，可以根据指定属性的名称的正则表达式来规定某些属性可以被访问，某些属性不能被访问。SecurityMemberAccess 的源码如代码清单 6-14 所示。

代码清单 6-14　SecurityMemberAccess.java

```
public class SecurityMemberAccess extends DefaultMemberAccess {

   private boolean allowStaticMethodAccess;
   Set<Pattern> excludeProperties = Collections.emptySet();
   Set<Pattern> acceptProperties = Collections.emptySet();

   public SecurityMemberAccess(boolean method) {
       super(false);
       allowStaticMethodAccess = method;
   }
   // 此处省略其余的代码
}
```

从上面的代码片段中我们可以看到，Struts2 在对象属性访问策略上进行了扩展：指定是否支持访问静态方法以及通过指定正则表达式来规定某些属性是否能够被访问。因而，在对象访问策略上，Struts2 的支持是非常全面的。

6.3.2.4 方法 / 属性访问机制——MethodAccessor 与 PropertyAccessor

MethodAccessor 和 PropertyAccessor 规定了 OGNL 在访问方法和属性时的实现方式。这两个接口的实现会真正影响到 OGNL 在根据表达式进行计算时的最终行为。其接口定义如代码清单 6-15 所示。

代码清单 6-15 MethodAccessor 和 PropertyAccessor

```
public interface MethodAccessor
{
    /**
     *  静态方法调用器
     *
     * @param context
     * @param targetClass
     * @param methodName
     * @param args
     *
     * @return
     * @exception MethodFailedException
     */
    Object callStaticMethod( Map context, Class targetClass, String
methodName, Object[] args )
            throws MethodFailedException;

    /**
     * 普通方法调用器
     *
     * @param context
     * @param target
     * @param methodName
     * @param args
     * @return
     * @exception MethodFailedException \
     */
    Object callMethod( Map context, Object target, String methodName,
Object[] args )
            throws MethodFailedException;
}

public interface PropertyAccessor
{

    /**
```

```
     * 获取属性值
     * @param context
     * @param target
     * @param name
     *
     * @return
     * @exception OgnlException
     *
     */
    Object getProperty(Map context, Object target, Object name)
        throws OgnlException;

    /**
     * 设置属性值
     * @param context
     * @param target
     * @param name
     * @param value
     *
     * @exception OgnlException
     */
    void setProperty(Map context, Object target, Object name, Object value)
        throws OgnlException;

    /**
     * 得到获取属性的方法签名，一般情况下，返回 'get'
     * @param context
     * @param target
     * @param index
     * @return
     */
    String getSourceAccessor(OgnlContext context, Object target, Object index);

    /**
     * 得到设置属性的方法签名，一般情况下，返回 'set'
     * @param context
     * @param target
     * @param index
     * @return
     */
    String getSourceSetter(OgnlContext context, Object target, Object index);
}
```

在 OGNL 的默认实现中，将 PropertyAccessor 的实现根据不同的 Class 类型分配到不同的实现类中，表明不同类型的取值方式是不同的。在 Struts2 中，同样会根据不同的 Class 类型来分配不同的 PropertyAccessor 实现去执行，如代码清单 6-16 所示。

代码清单 6-16 各种 PropertyAccessor 和 MethodAccessor 的实现定义

```
<!-- 下面是 PropertyAccessor 针对不同 class 的不同实现方式 -->
<bean type="ognl.PropertyAccessor"
name="com.opensymphony.xwork2.util.CompoundRoot"
class="com.opensymphony.xwork2.ognl.accessor.CompoundRootAccessor" />
<bean type="ognl.PropertyAccessor" name="java.lang.Object"
class="com.opensymphony.xwork2.ognl.accessor.ObjectAccessor" />
<bean type="ognl.PropertyAccessor" name="java.util.Iterator"
class="com.opensymphony.xwork2.ognl.accessor.XWorkIteratorPropertyAccessor" />
<bean type="ognl.PropertyAccessor" name="java.util.Enumeration"
class="com.opensymphony.xwork2.ognl.accessor.XWorkEnumerationAccessor" />
<bean type="ognl.PropertyAccessor" name="java.util.List"
class="com.opensymphony.xwork2.ognl.accessor.XWorkListPropertyAccessor" />
<bean type="ognl.PropertyAccessor" name="java.util.Set"
class="com.opensymphony.xwork2.ognl.accessor.XWorkCollectionPropertyAccessor" />
<bean type="ognl.PropertyAccessor" name="java.util.Map"
class="com.opensymphony.xwork2.ognl.accessor.XWorkMapPropertyAccessor" />
<bean type="ognl.PropertyAccessor" name="java.util.Collection"
class="com.opensymphony.xwork2.ognl.accessor.XWorkCollectionPropertyAccessor" />
<bean type="ognl.PropertyAccessor"
name="com.opensymphony.xwork2.ognl.ObjectProxy"
class="com.opensymphony.xwork2.ognl.accessor.ObjectProxyPropertyAccessor" />
<!-- 下面是 MethodAccessor 的两种不同实现方式 -->
<bean type="ognl.MethodAccessor" name="java.lang.Object"
class="com.opensymphony.xwork2.ognl.accessor.XWorkMethodAccessor" />
<bean type="ognl.MethodAccessor"
name="com.opensymphony.xwork2.util.CompoundRoot"
class="com.opensymphony.xwork2.ognl.accessor.CompoundRootAccessor" />
```

正如在介绍 ClassResolver 时谈到的，我们在这里看到 CompoundRootAccessor 类是 Struts2 中指定 OGNL 计算规则最主要的实现类，同时也成为我们对 OGNL 计算进行扩展的主要窗口。

方法和属性的访问机制，是进行对象读写访问的基础。从上面的配置中我们就可以看到，XWork/Struts2 目前所采取的策略主要是将访问机制的实现根据不同的 Java 类型派发到相应的实现类中去，并提供默认的实现方式。这样一来，不仅保证了在默认情况下我们能够获得框架自身所带来的便捷，也提供了相应的扩展接口。

6.3.2.5 空值处理机制——NullHandler

NullHandler 的作用是指定 OGNL 在计算时，如果遭遇到级联对象（Nested Object）时的处理方式。例如，一个 User 对象中包含着一个 Department 对象，如果你期望通过表达式 user.department.name 为 user 对象中的 department 对象设置 name 属性的值，那么前提条件是 OGNL 在处理时能够在遇到 department 对象时自动创建该对象，而不是抛出 NullPointException。NullHandler 的接口定义，如代码清单 6-17 所示。

代码清单 6-17　NullHandler.java

```
public interface NullHandler
{
    /**
     *  当所调用的方法不存在时的处理策略
     */
     public Object nullMethodResult(Map context, Object target, String
methodName, Object[] args);

    /**
     *  当所调用属性不存在时的处理策略
     */
    public Object nullPropertyValue(Map context, Object target, Object
property);
}
```

在 OGNL 的默认实现中采用了 ognl.DefaultNullHandler，也就是不做任何处理，直接返回一个 null。在这种情况下，我们上述所说的例子中的需求是无法实现的。而在 XWork/Struts2 中，为了支持扩展，实现了一个自己的 NullHandler 接口，与 TypeConverter 类似，也使用了装饰模式，将其接口的实现封装在 com.opensymphony.xwork2.ongl.OgnlNullHandlerWrapper 内部。其相关源码如代码清单 6-18 所示。

代码清单 6-18　OgnlNullHandlerWrapper.java

```
public class OgnlNullHandlerWrapper implements ognl.NullHandler {

   private NullHandler wrapped;

   public OgnlNullHandlerWrapper(NullHandler target) {
       this.wrapped = target;
   }

   public Object nullMethodResult(Map context, Object target,
           String methodName, Object[] args) {
           // 调用 XWork 的 NullHandler 实现，支持自定义的实现替换
       return wrapped.nullMethodResult(context, target, methodName, args);
   }

   public Object nullPropertyValue(Map context, Object target, Object property)
{
    // 调用 XWork 的 NullHandler 实现，支持自定义的实现替换
    return wrapped.nullPropertyValue(context, target, property);
   }
}
```

那么，在 XWork 内部到底使用了什么样的 NullHandler 处理方法呢？我们需要看看被

装饰模式隐藏在内部的真正实现，如 InstantiatingNullHandler 所示（见代码清单 6-19）。

代码清单 6-19 InstantiatingNullHandler.java

```
public class InstantiatingNullHandler implements NullHandler {

    private static final Logger LOG =
LoggerFactory.getLogger(InstantiatingNullHandler.class);
    private ReflectionProvider reflectionProvider;
    private ObjectFactory objectFactory;
    private ObjectTypeDeterminer objectTypeDeterminer;

    @Inject
    public void setObjectTypeDeterminer(ObjectTypeDeterminer det) {
        this.objectTypeDeterminer = det;
    }

    @Inject
    public void setReflectionProvider(ReflectionProvider prov) {
        this.reflectionProvider = prov;
    }

    @Inject
    public void setObjectFactory(ObjectFactory fac) {
        this.objectFactory = fac;
    }

    public Object nullMethodResult(Map<String, Object> context, Object
target, String methodName, Object[] args) {
        if (LOG.isDebugEnabled()) {
            LOG.debug("Entering nullMethodResult ");
        }
        // 不做任何处理
        return null;
    }

    public Object nullPropertyValue(Map<String, Object> context, Object
target, Object property) {
        if (LOG.isDebugEnabled()) {
            LOG.debug("Entering nullPropertyValue [target="+target+",
 property="+property+"]");
        }

        // 可以在运行期指定此次计算的 Null 值处理策略，在 context 中设置
        boolean c =
ReflectionContextState.isCreatingNullObjects(context);

        if (!c) {
            return null;
```

```
        }

        if ((target == null) || (property == null)) {
            return null;
        }

        try {
            String propName = property.toString();
            // 获取实际的对象
            Object realTarget = reflectionProvider.getRealTarget(propName,
context, target);
            Class clazz = null;

            if (realTarget != null) {
                PropertyDescriptor pd =
reflectionProvider.getPropertyDescriptor(realTarget.getClass(),
propName);
                if (pd == null) {
                    return null;
                }
                // 根据实际获取的对象，获取对应的 class 类型
                clazz = pd.getPropertyType();
            }

            if (clazz == null) {
                // can't do much here!
                return null;
            }
            // 调用相应的方法构造 null 值对象
            Object param = createObject(clazz, realTarget, propName,
context);

            reflectionProvider.setValue(propName, context, realTarget,
param);

            return param;
        } catch (Exception e) {
            LOG.error("Could not create and/or set value back on to object", e);
        }
        return null;
    }

private Object createObject(Class clazz, Object target, String
property, Map<String, Object> context) throws Exception {
        // 根据特殊的 clazz 类型，处理返回的结果对象
        if (Collection.class.isAssignableFrom(clazz)) {
            return new ArrayList();
        } else if (clazz == Map.class) {
            return new HashMap();
```

```
        } else if (clazz == EnumMap.class) {
            Class keyClass =
objectTypeDeterminer.getKeyClass(target.getClass(), property);
            return new EnumMap(keyClass);
        }
        // 普通的对象使用 ObjectFactory 来构造，默认使用 JVM 的反射方式
        return objectFactory.buildBean(clazz, context);
    }
}
```

由此可见，在 XWork 内部，会根据对象实际对应的类型，使用 ObjectFactory 来构造对象。而 ObjectFactory 正是一个“万能对象构造器”（我们将在之后的章节揭开它的神秘面纱）。在默认情况下，ObjectFactory 会直接使用 JDK 中的基本反射方法 clazz.newInstance() 来创建对象，但是如果指定了特殊的 ObjectFactory 的实现，则能够享受到一些容器的特殊服务，比如我们可以使用 SpringObjectFactory 作为 ObjectFactory 的实现，从而把对象的构建工作交给 Spring 容器来做。

有了 NullHandler，我们就无须再担心任何由于对象的级联关系而造成的 null 值的情况，因为 Struts2 在默认实现中已经帮我们处理并屏蔽了其中的实现细节。

6.3.3 深入 OGNL 的扩展方式

OGNL 的计算规则实际上指定了 OGNL 在进行表达式计算时的各种行为处理方式。这些行为处理方式同时也是 OGNL 计算的扩展点。当我们需要扩展 OGNL 的计算规则时，就需要继承或覆盖这些 OGNL 的默认处理方式。接下来，我们就来看看 XWork / Struts2 是如何扩展这些默认处理方式的。

6.3.3.1 扩展 ClassResolver

我们从 ClassResolver 的接口定义中就可以考虑到一种很直观的扩展方式，那就是在 ClassResolver 的实现类中指定与某些特定的字符串输入匹配的 Java 类。以 XWork 的默认实现 CompoundRootAccessor 为例，它就实现了一个特殊字符串 vs 或者 vs 数组作为特殊的 Java 类 ValueStack 的寻址方式，其源码如代码清单 6-20 所示。

代码清单 6-20 CompoundRootAccessor.java

```
public Class classForName(String className, Map context) throws
ClassNotFoundException {
        // 获取 OGNL 的 Root 对象
        Object root = Ognl.getRoot(context);

        try {
            // 如果 Root 对象是 CompoundRoot 类型，则需要处理特殊名称的 className
```

```
        if (root instanceof CompoundRoot) {
            if (className.startsWith("vs")) {
                CompoundRoot compoundRoot = (CompoundRoot) root;
            // 支持直接使用 vs 作为 className
                if ("vs".equals(className)) {
                    return compoundRoot.peek().getClass();
                }
            // 支持使用 vs2、vs3 等形式作为 className
                int index = Integer.parseInt(className.substring(2));

                return compoundRoot.get(index - 1).getClass();
            }
        }
    } catch (Exception e) {
        // 仅仅是尝试一下旧式方法
    }

    return
Thread.currentThread().getContextClassLoader().loadClass(className);
}
```

从源码中我们可以看到，当传入的 className 是 vs 或者类似 vs[0]、vs[1] 这样的字符串表达式时，OGNL 将在其计算的 Root 对象 CompoundRoot 中根据其序列号返回相应的对象。在之后的章节中，我们会着重分析 CompoundRoot 的结构，帮助读者理解 vs、vs[0]、vs[1] 这样的字符串表达式的真正用途。在这里，读者只需要理解 CompoundRootAccessor 所实现的 ClassResolver 对于特殊的 className 有着截然不同的 Java 类寻址方式。它实际上为我们提供了一种代码范例，读者可以仔细体会代码逻辑对 OGNL 计算规则的扩展所表现的层次实际上停留在 Java 类型级别。在之后的分析中，我们还可以看到针对属性、方法级别的扩展方式。

6.3.3.2　扩展 TypeConverter

我们在之前对 TypeConverter 的默认实现中已经看到 Struts2 在使用 TypeConverter 时采取的装饰模式的实现方式。当时我们给出了 TypeConverter 在 XWork 框架中的默认实现类 DefaultTypeConverter 的源码，并留下了一个问题，如何自定义 TypeConverter 从而能够将其插入到 XWork 框架之中，并被 XWork 所识别呢？

要回答这个问题，我们首先需要了解 XWork 框架中 TypeConverter 的实现结构，其类图结构如图 6-2 所示。

从这个结构图中，我们可以看到 XWork 框架在类型转化这个问题上所下的苦功夫：对于默认实现 DefaultTypeConverter，使用了两个完全不同的扩展方式。首先，使用“继承”的方式对 DefaultTypeConverter 进行基本的扩展（XWorkBasicConverter），覆盖其默

认的行为方式。然后，进一步使用装饰模式将这个被扩展的 XWorkBasicConverter 封装在内部，对外呈现出 XWorkConverter 这个装饰类！

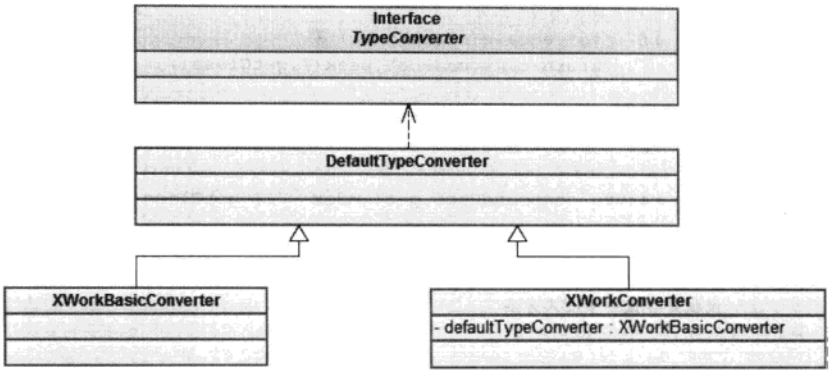


图 6-2 XWork 中 TypeConverter 的实现结构

这种具备了多层次扩展的实现机制，体现了对象行为扩展的不同方面：一方面，当需要针对对象的某些行为进行整体覆盖时，我们可以使用“继承”机制，覆写（Override）其父类的接口函数来完成；另一方面，当需要针对对象的某些行为进行选择性扩展时，我们可以使用装饰模式来完成。读者可以结合本书第 4 章中有关装饰模式的讲解，更加深刻地认识 XWork 的设计。

接下来我们研究一下 XWorkBasicConverter 的源码，看看它对 DefaultTypeConverter 到底做了哪些方面的扩展。其核心源码如代码清单 6-21 所示。

代码清单 6-21 XWorkBasicConverter.java

```
@Override
public Object convertValue(Map<String, Object> context, Object o, Member
member, String s, Object value, Class toType) {
    Object result = null;

    if (value == null || toType.isAssignableFrom(value.getClass())) {
        // 无须进行类型转化
        return value;
    }

    // 根据不同的类型调用不同的类型转化函数
    if (toType == String.class) {
        result = doConvertToString(context, value);
    } else if (toType == boolean.class) {
        result = doConvertToBoolean(value);
    } else if (toType == Boolean.class) {
        result = doConvertToBoolean(value);
```

```
    } else if (toType.isArray()) {
        result = doConvertToArray(context, o, member, s, value, toType);
    } else if (Date.class.isAssignableFrom(toType)) {
        result = doConvertToDate(context, value, toType);
    } else if (Calendar.class.isAssignableFrom(toType)) {
        Date dateResult = (Date) doConvertToDate(context, value,
Date.class);
        if (dateResult != null) {
            Calendar calendar = Calendar.getInstance();
            calendar.setTime(dateResult);
            result = calendar;
        }
    } else if (Collection.class.isAssignableFrom(toType)) {
        result = doConvertToCollection(context, o, member, s, value, toType);
   } else if (toType == Character.class) {
        result = doConvertToCharacter(value);
    } else if (toType == char.class) {
        result = doConvertToCharacter(value);
    } else if (Number.class.isAssignableFrom(toType) ||
toType.isPrimitive()) {
        result = doConvertToNumber(context, value, toType);
    } else if (toType == Class.class) {
        result = doConvertToClass(value);
    }

    if (result == null) {
        if (value instanceof Object[]) {
            Object[] array = (Object[]) value;
            if (array.length >= 1) {
                value = array[0];
            } else {
                value = null;
            }

             // 只转化第一个数组元素
            result = convertValue(context, o, member, s, value, toType);
        } else if (!"".equals(value)) {
            result = super.convertValue(context, value, toType);
        }

        if (result == null && value != null && !"".equals(value)) {
            throw new XWorkException("Cannot create type " + toType + " from
value " + value);
        }
    }

    return result;
}
```

从源码中，我们可以看到比 DefaultTypeConverter 更为广泛的类型转化支持。除了我们想象得到的基本类型之外，还包含了 Date 类型、Collection 类型等等。可见，**这个层次的对象行为扩展，还停留在行为运行机制本身，扩展的主要目的是增加更多的行为方式支持。**

而在 XWorkConverter 中，则使用了装饰模式，实现了我们万众期待的“自定义类型转化”的注册和使用。我们首先关注的是 XWorkConverter 的初始化部分，其相关源码如代码清单 6-22 所示。

代码清单 6-22 XWorkConverter.java

```
public class XWorkConverter extends DefaultTypeConverter {


    /**
     * Java 类内部的属性映射缓存表
     */
    protected HashMap<Class, Map<String, Object>> mappings = new
HashMap<Class, Map<String, Object>>(); // action

    /**
     * 无法找到类型转化处理类的那些 Java 类的缓存集合
     */
    protected HashSet<Class> noMapping = new HashSet<Class>(); // action

    /**
     * 系统级别的映射缓存表
     */
    protected HashMap<String, TypeConverter> defaultMappings = new
HashMap<String, TypeConverter>();

    /**
     * 未知类型转化处理类的 Java 类的缓存集合
     */
protected HashSet<String> unknownMappings = new HashSet<String>();

    private TypeConverter defaultTypeConverter;
    private ObjectFactory objectFactory;

    protected XWorkConverter() {
    }

    @Inject
    public void setObjectFactory(ObjectFactory factory) {
        this.objectFactory = factory;
        // 这个文件已经不再使用
        loadConversionProperties("xwork-default-conversion.properties");
```

```
        // 读取 classpath 中的 xwork-conversion.properties 文件中的映射关系
        loadConversionProperties("xwork-conversion.properties");
    }

    @Inject
    public void setDefaultTypeConverter(XWorkBasicConverter conv) {
        // 注意这里被注入的是 XWorkBasicConverter 类
        // 这样一来整个 XWorkConverter 看起来就像是一个装饰类
      this.defaultTypeConverter = conv;
    }
}
```

从源码中，我们可以看到 XWorkConverter 在其内部定义了 4 个重要的映射缓存表：defaultMappings、mappings、noMapping 和 unknownMappings。这 4 个映射关系分别表达了 4 种截然不同的逻辑意义：系统级别的映射缓存表、Java 类内部的属性映射缓存表、无法找到类型转化处理类的那些 Java 类的缓存集合和那些未知类型转化处理类的 Java 类的缓存集合。

在 setObjectFactory 方法中，XWorkConverter 进行系统级别的类型转化处理声明的加载。根据容器的相关知识，setObjectFactory 是一个被加入了 @Inject 注解的方法，因而它会在 XWorkConverter 进行初始化时被调用。因而系统级别的类型转化处理声明中的映射关系会首先缓存在 defaultMappings 中。

从初始化源码中，我们可以发现在 XWork / Struts2 中，可以支持三种不同的类型转化处理的声明。

❑ 在 xwork-conversion.properties 文件中声明

这是一个系统级别的类型转化处理的声明方式。其中，xwork-conversion.properties 是位于 CLASSPATH 下的文件。它将在 XWorkConverter 进行初始化时被加载并缓存其中类和对应类型转化处理类的对应关系。

❑ 在 Java 类所处的 package 目录的 ClassName-conversion.peroperties 文件中声明

这是一个类级别的类型转化处理的声明方式，它将在运行期被加载。一旦其中的映射关系被加载，它们就会被 XWorkConverter 缓存起来。

❑ 为 Java 类指定特殊的 Annotation

这同样是一个类级别的类型转化处理的声明方式，在运行期被加载和缓存。在这里，XWork 使用了 JDK1.5 的 Annotation 特性。有了它，我们无须再为每个需要进行特殊类型转化的 Java 类匹配一个额外的配置文件。

接下来，我们就来看看 XWorkConverter 真正进行类型转化相关的源码，揭秘上面这三种不同的类型转化处理的声明在 XWorkConverter 中的初始化和运用过程。其相关源码如代码清单 6-23 所示。

代码清单 6-23 XWorkConverter.java

```
@Override
public Object convertValue(Map<String, Object> context, Object target,
Member member, String property, Object value, Class toClass) {

        TypeConverter tc = null;

        if ((value != null) && (toClass == value.getClass())) {
            return value;
        }


        if (target != null) {
            // 获取目标 clazz
        Class clazz = target.getClass();

            Object[] classProp = null;

            // 如果对象是 CompoundRoot，需要做一些特殊处理
            if ((target instanceof CompoundRoot) && (context != null)) {
                classProp = getClassProperty(context);
            }

            if (classProp != null) {
                clazz = (Class) classProp[0];
                property = (String) classProp[1];
            }

            // 根据目标 clazz 和 property，查找对应的类型转化方式
            tc = (TypeConverter) getConverter(clazz, property);

            if (LOG.isDebugEnabled())
                LOG.debug("field-level type converter for property [" +
property + "] = " + (tc == null ? "none found" : tc));
        }

        if (tc == null && context != null) {
            // 如果 context 存在，可以根据 path 进行目标 clazz 和 property 的查找
            Object lastPropertyPath =
context.get(ReflectionContextState.CURRENT_PROPERTY_PATH);
            Class clazz = (Class)
context.get(XWorkConverter.LAST_BEAN_CLASS_ACCESSED);
            if (lastPropertyPath != null && clazz != null) {
                String path = lastPropertyPath + "." + property;
                tc = (TypeConverter) getConverter(clazz, path);
            }
        }
```

```
        if (tc == null) {
            if (toClass.equals(String.class) && (value != null)
&& !(value.getClass().equals(String.class) ||
value.getClass().equals(String[].class))) {
                // 如果待转化的类型是 String，直接使用 value 本身的 toString 方法
                tc = lookup(value.getClass());
            } else {
                tc = lookup(toClass);
            }

            if (LOG.isDebugEnabled())
                LOG.debug("global-level type converter for property [" +
 property + "] = " + (tc == null ? "none found" : tc));
        }


        if (tc != null) {
            try {
                // 如果此时找到了 typeConverter，则使用该 typeConverter 完成逻辑
                return tc.convertValue(context, target, member, property,
 value, toClass);
            } catch (Exception e) {
                if (LOG.isDebugEnabled())
                    LOG.debug("unable to convert value using type converter
[#0]", e, tc.getClass().getName());
                handleConversionException(context, property, value,
target);

                return TypeConverter.NO_CONVERSION_POSSIBLE;
            }
        }

        if (defaultTypeConverter != null) {
            try {
                if (LOG.isDebugEnabled())
                    LOG.debug("falling back to default type converter [" +
defaultTypeConverter + "]");
                // 如果没有找到对应的 typeConverter
                // 使用 XWorkBasicConverter 完成逻辑
                return defaultTypeConverter.convertValue(context, target,
 member, property, value, toClass);
            } catch (Exception e) {
                if (LOG.isDebugEnabled())
                    LOG.debug("unable to convert value using type converter
[#0]", e, defaultTypeConverter.getClass().getName());
                handleConversionException(context, property, value,
 target);

                return TypeConverter.NO_CONVERSION_POSSIBLE;
```

```
            }
        } else {
            try {
                if (LOG.isDebugEnabled())
                    LOG.debug("falling back to Ognl's default type
conversion");
                // 如果 XWorkBasicConverter 没有被正确注入
                // 使用 DefaultTypeConverter 完成逻辑
                return super.convertValue(value, toClass);
            } catch (Exception e) {
                if (LOG.isDebugEnabled())
                    LOG.debug("unable to convert value using type converter
[#0]", e, super.getClass().getName());
                handleConversionException(context, property, value,
target);

                return TypeConverter.NO_CONVERSION_POSSIBLE;
            }
        }
    }
```

在 XWorkConverter 真正进行类型转化时，会首先通过 getConverter 方法的调用来获取当前 Java 类所对应的类型转化处理类。如果此时在 defaultMappings 中无法找到这种对应关系，那么 XWorkConverter 将会尝试动态地从这个 Java 类所处的 package 目录中相应的配置文件或者这个 Java 类的 Annotation 中去加载。在这种情况下，我们可以看到 buildConverterMapping 和 addConverterMapping 方法会根据不同的情况，将 Java 类和其对应的类型转化处理类的映射关系缓存到不同的映射缓存表中去。由此可见，只要使用之前所介绍的 3 种自定义类型转化方式的声明，在 XWorkConverter 中，就能够正确识别并将其加入到处理流程中。在进行类型转化时，XWorkConverter 首先根据我们刚才所介绍的三个层次依次查找当前这个 Java 类所对应的类型转化处理类。如果没有找到相应的用户自定义类型转化处理器，再调用默认的常规类型转化处理器（XWorkBasicConverter）进行处理。这就是我们可以进行各种不同级别指定特殊的类型转化处理类的窗口。在这里，XWorkConverter 的多层次扩展方式为我们提供了完整的扩展途径。

6.3.3.3 扩展 MethodAccessor 和 PropertyAccessor

当一个对象的结构较为复杂时，MethodAccessor 和 PropertyAccessor 的扩展实现就能够派上用场，用于指定某些自定义的字段或者方法的特殊访问方式。

我们在这里摘取 XWork 中对 Iterator 这个类型的 PropertyAccessor 扩展进行说明，其相关源码如代码清单 6-24 所示。

代码清单 6-24　IteratorPropertyAccessor.java

```
public class IteratorPropertyAccessor extends ObjectPropertyAccessor
    implements PropertyAccessor {

    public Object getProperty( Map context, Object target, Object name )
throws OgnlException {
        Object      result;
        Iterator    iterator = (Iterator)target;

        if ( name instanceof String ) {
            // 当 property 的名称为 next 或者 hasNext 时，将返回特殊的对象
            if (name.equals("next")) {
                result = iterator.next();
            } else {
                if (name.equals("hasNext")) {
                    result = iterator.hasNext() ? Boolean.TRUE :
 Boolean.FALSE;
                } else {
                    result = super.getProperty( context, target, name );
                }
            }
        } else {
            result = super.getProperty(context, target, name);
        }
        return result;
    }

    public void setProperty( Map context, Object target, Object name, Object
value ) throws OgnlException
    {
        throw new IllegalArgumentException( "can't set property " + name +
" on Iterator" );
    }
}
```

从源码中，我们可以看到当 OGNL 表达式在处理 Iterator 类型的 next 字段时，会进行特殊的处理，而并非采用默认的使用对象的 getter 方法访问的方式。在下一章中，我们将看到这种扩展方式在 XWork 中的频繁使用。

6.4　小结

一旦程序处于运行状态，数据载体在 MVC 的不同层次之间的表现形式的差异就成为数据流转和数据访问的绊脚石。于是，以 OGNL 为代表的表达式引擎就如同一剂灵丹妙药，架起了数据沟通的桥梁。本章重点介绍了 OGNL 的基本操作语法和 OGNL 的基本要

素，并深入到 OGNL 的内部，分析了 OGNL 在实现机理和扩展机制上的独到之处。

本章的内容虽然偏于理论，却值得读者仔细品味。本章的内容可以脱离 Struts2 而独立，并为需要了解表达式引擎的读者提供一个很好的学习思路。读完本章之后，请读者自行就如下问题温故而知新。

- ❑ 为什么要引入表达式引擎？表达式引擎在 Web 开发中有什么重要作用？
- ❑ OGNL 的三要素是什么？
- ❑ OGNL 有哪些基本操作？
- ❑ 在 OGNL 中，针对 Root 对象的计算与针对上下文环境的计算有什么不同？
- ❑ OGNL 的计算行为是在哪些 OGNL 内部的变量中指定的？
- ❑ OGNL 在计算时是如何处理类型转化的？又是如何处理级联对象的初始化行为的？
- ❑ OGNL 表达式中的 this 指针是指什么？
- ❑ OGNL 表达式中的 # 符号有什么用途？
- ❑ OGNL 有哪些扩展点？
- ❑ 如何在 XWork 中扩展 OGNL 计算中的类型转化？

第 7 章　别具匠心——XWork 设计原理

众所周知，现代电子计算机由 5 大部件组成：运算器、控制器、存储器、输入设备和输出设备。其中，运算器和控制器合称 CPU，是计算机中最为核心的部分。**如果我们把整个 Struts2 框架比作一台计算机，那么 XWork 则是 Struts2 框架中的 CPU，是 Struts2 运行机制的核心。**

那么，XWork 到底是一个什么样的开发框架呢？我们曾经在本书的第 3 章中对 XWork 框架有一个初步的介绍：

> *XWork is a command-pattern framework that is used to power Struts 2 as well as other applications. XWork provides an Inversion of Control container, a powerful expression language, data type conversion, validation, and pluggable configuration.*

按照 XWork 框架官方的定义，**XWork 是一个灵活而可靠的基于命令模式的开发框架。**所谓“命令模式”，其本质上是请求－响应模式。因而，基于命令模式的开发框架，也就是指一个开发框架能够实现人机交互、实现请求－响应模式。

在本章中，我们就从人机交互的基本模式（请求－响应模式）开始讲起，通过对请求－响应模式实现机理的分析，引出 XWork 框架在实现请求－响应模式过程中独特的设计理念，并对 XWork 框架的体系结构进行系统的阐述。

7.1　请求－响应的哲学

7.1.1　请求－响应的基本概念

人和人之间的沟通与交流，在绝大多数情况下是通过语言来完成的。语言，就好比一个双方都能够认同的协议，只有交流的双方都能够听懂某一种语言，双方之间的沟通才有了起码的基础，交流才能有效地进行。你一言我一语的交流可以看作是一个回合制的请求－响应过程，因为每一个沟通的回合实际上可看作是一个发起方进行请求、响应方进行响应的交互过程。我们可以用一个示意图来表达整个交互的过程，这个过程如图 7-1 所示。

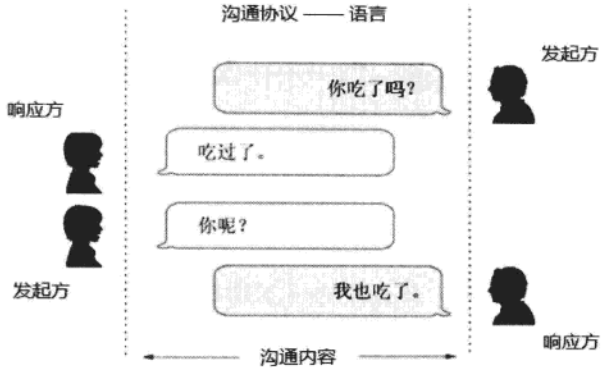


图 7-1 人与人的沟通示意图

在图中，我们可以看到一个完整的沟通流程，主要有四个构成要素：

❑ **沟通协议**——某种双方都能明白的沟通机制，例如语言、手势等。

❑ **发起方**——沟通的发起者。

❑ **沟通内容**—— 交流的具体内容，例如，“你吃了吗？”

❑ **响应方**——沟通的接收方和响应者。

这种沟通方式在人类日常生活中是最为常见的一种模式。在上述这四个构成要素中，我们又可以总结出两个主要特点：

❑ **沟通协议是沟通内容的基础，沟通内容是沟通协议的具体表现形式。**

❑ **发起方和响应方的角色并不固定，只有在一个交互回合中才能确定角色。**

如果我们把沟通延伸到人与机器之间，上述的模式就会发生一些变化。因为至少有一点值得我们注意，从人机交互的特点上讲，机器是不会主动发起沟通请求的。这样一来，人机交互的整个过程就如图 7-2 所示。

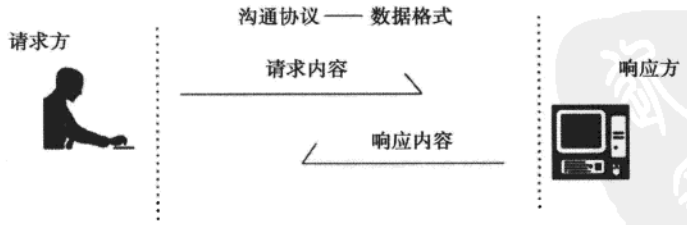


图 7-2 人机交互的示意图

从图中，我们可以看到人机交互的构成要素就变成了以下三个元素：

❑ **沟通协议**——人和机器都能够明白的数据通信格式。

❑ **请求内容**——人通过某种机制向机器发起的数据请求。

❑ **响应内容**——机器接收到数据请求并做逻辑处理之后，进行响应的数据内容。

因为请求方和响应方是固定的，因而在人机交互的过程中，我们关注的重点内容就转变为：**人作为沟通的发起方，其请求的内容是什么？机器作为命令的接收方和响应方，在处理完毕之后，返回的内容是什么？**在这种情况下，基于人机交互的请求 - 响应模式的概念也就应运而生了。

结论　*请求 - 响应模式是一种概念非常宽泛的人机交互模式，是人与计算机进行沟通的一种最基本的行为方式。*

把视角拉回到 Web 开发，我们会发现基于 B/S 体系的 Web 应用是一个典型的基于请求 - 响应模式的体系架构。对于之前所谈到的请求 - 响应模式的三要素，我们也可以在其中找到对应关系：

❑ **沟通协议**——Http 协议

❑ **请求内容**——Http 请求

❑ **响应内容**——Http 响应

浏览器发送 **Http 请求**到服务器端，服务器端的程序获得 Http 请求后进行逻辑处理，并将逻辑处理的结果返回，这个返回的过程就称为 **Http 响应**。请求和响应不断的交互过程，构成了所有 B/S 体系结构的应用构架的基础。如果把整个过程的通信基础冠以一个官方的名称，它就是 **Http 协议**。

根据请求 - 响应模式总结出来的这三大要素自然而然成为我们在进行 Web 开发编程中的设计依据。在 Servlet 标准中，我们所熟悉的 HttpServletRequest 对象，就对应于这里的 Http 请求；而 HttpServletResponse 对象，则对应于 Http 响应。

由此可见，我们日常所说的进行 Web 开发的核心内容实际上就是指如何编写可运行于 Web 容器的服务器端程序用于进行 Http 请求的响应、进行逻辑处理并返回处理结果这样一个完整的过程。

7.1.2　请求 - 响应的实现模式

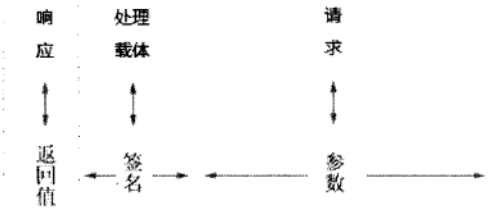
我们来回顾一下基于 Http 协议的人机交互过程：由客户端浏览器发起一个 Http 请求，服务器端接收到请求之后进行逻辑处理，并将处理结果返回给客户端进行 Http 响应。我们所提出的问题是：这样一个过程，我们如何使用 Java 语言来实现呢？

如果从 Java 的语法入手，通过分析对象的内部构成机理，我们可以得到三种风格迥异的请求 - 响应实现模式。这三种模式，在实际编程中也是不同的 Web 框架实现请求 - 响应模式的理论基础，接下来我们就来分别分析这三种不同的实现模式。

7.1.2.1 参数－返回值（Param-Return）模式

参数－返回值模式是一种最为直观的请求－响应的实现模式。我们曾经在第 2 章中详细分析过对象的一种运作模式：**行为对象模式**。这种行为对象模式，实际上就是我们这里所说的参数－返回值模式的产生原型。

在第 2 章中讲到有关行为对象模式的概念时，我们当时给出了一个针对方法的构成分析。我们在这里继续引用当时的分析，并把它和请求－响应模式对应起来，整个过程如图 7-3 所示。




```
public Return someMethod(Param param1, Param param2) {
                    方法实现
}
```

图 7-3 方法的构成分析

在这里，我们发现方法在语法上天然的能够与请求－响应模式相对应：

- **方法签名**——请求－响应模式的处理载体
- **方法参数**——请求内容映射
- **方法返回值**——处理结果响应

因此，对象的方法定义与我们之前的请求－响应模式的流程是相通的，从而使得**对象的方法成为请求－响应模式在 Java 世界中的一种直观抽象**。

在我们熟知的 Web 框架中，SpringMVC 就是基于这种模式来定义和实现 Http 请求的处理与响应的。我们不妨来看一个典型的 SpringMVC 的 Controller 的示例，如代码清单 7-1 所示。

代码清单 7-1 UserController.java

```
@Controller
public class UserController {

   @RequestMapping("/login")
   public ModelAndView login(String userName, String password) {

         // 这里省略了 login 的业务逻辑处理
```

```
        // 构建视图返回
        ModelAndView modelAndView = new ModelAndView("index");
        return modelAndView;
    }

}
```

在上述代码中，方法的参数（userName 和 password）被视作是 Http 请求的概括，它们已经被 SpringMVC 的框架有效处理并屏蔽了内在的处理细节，呈现出来的是与请求参数名称一一对应的参数列表。而返回值 modelAndView 则表示 Http 的响应是一个数据与视图的结合体，表示 Http 的处理结果。而函数体（login 方法）本身，在其内部包含了进行逻辑处理的整个过程。

深入研究 SpringMVC 框架在 Controller 的定义中所支持的参数列表和返回值列表，我们可以发现 SpringMVC 所支持的内容非常宽泛，不仅在参数中支持与参数请求名称一一对应的参数列表，也支持 HttpServletRequest 等 Web 容器原生对象。事实上，万变不离其宗，无论参数列表如何变化，无论返回值的形式如何变化，**参数－返回值这样一种响应模式都是其设计的核心内容**。这种非常直观并符合简单逻辑思维的抽象方式，应该说也完全符合“简单是美（Simple is Beauty）”的最佳实践。

7.1.2.2　参数－参数（Param-Param）模式

参数－参数模式实际上比参数－返回值模式更早出现。因为我们所熟知的 Servlet 标准就是基于参数－参数模式进行设计的，因此这种参数－参数模式也称之为 Servlet 模式。

Servlet 标准是 J2EE 的基本标准之一，我们在这里不妨来看看，在 Servlet 标准中定义了什么样的 Http 请求的处理接口。其相关接口定义如图 7-4 所示。

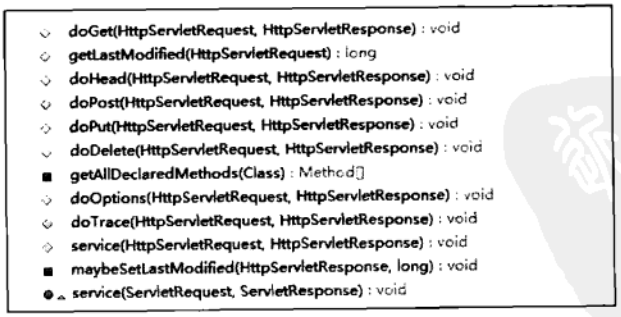


图 7-4　HttpServlet 接口定义

在 Servlet 标准的接口定义中，我们可以看到 service 方法是进行 Http 请求处理的实际

场所。因而就 Http 请求的响应本身而言，与之前所提到的参数－返回值模式并没有什么本质区别。然而，在 service 方法及其相关的 doGet 和 doPost 方法中，我们可以看到与之前参数－返回值模式的不同之处：

- **参数列表**——Http 请求被封装为一个 HttpServletRequest 对象（或者 ServletRequest 对象），而 Http 响应封装为一个 HttpServletResponse 对象（或者 ServletResponse 对象）
- **返回值**——方法不存在返回值（返回值为 void）

因此，这里最大的不同就在于将返回值的位置转移到了参数列表之中。**这种将请求和响应同时置于参数位置的模式，就是我们所说的参数－参数（Param-Param）模式。**

从上一节的分析中，我们可以看到参数－返回值模式是对 Java 语言中方法这种编程元素的解读。那么为什么 Servlet 作为 Java 规范中进行 Http 响应的标准，却要采用参数－参数模式来实现呢？

Servlet 作为一个开发标准，它所设计的接口已经无法再将任何处理职责继续往上层推送了，因为它是我们进行 Web 开发的底层标准。因而对于 Http 请求的处理，在 Servlet 中不仅需要知道返回给请求的发起者（浏览器）一个什么样的处理结果，还需要真实地将处理结果呈现在浏览器中。而这一点，我们不得不借助 HttpServletResponse 对象的操作在 doGet 和 doPost 方法体的内部调用相应的接口函数来完成。因而此时，方法的“返回值”对于 Servlet 对象来说没有任何操作上的意义，Servlet 必须通过 HttpServletResponse 的调用操作来完成浏览器的响应工作，这也就是我们不得不把 HttpServletResponse 置于参数位置的原因。

由此可见，**参数－参数模式是一种最为基础的请求－响应实现机制，也是底层规范不得不采用的一种实现机制。**这种实现机制虽然原始（因为它需要我们自行处理 Http 的实现细节），然而它却是实现完整请求－响应机制的唯一途径。

在绝大多数的 Web 开发框架中，我们并不会看到 Servlet 模式这种原始的实现模式，因为几乎所有的 Web 框架都会以这种模式为基础，将具体实现转化成另外两种实现模式。不过这种参数－参数模式在形式上带来的逻辑意义，依旧是值得我们深入思考的。

7.1.2.3 POJO 模式

如果说参数－返回值模式是请求－响应模式在 Java 世界中的一种直观抽象，那么 POJO 模式就是一种比较晦涩的请求－响应模式实现方式了。接下来，我们来看看 POJO 模式的代码构成要素，如图 7-5 所示。

从图中我们可以看出，POJO 模式与之前所介绍的两种实现模式之间的一个巨大不同就在于 **POJO 模式完全颠覆了前两种实现模式中以 Java 类中的方法的语法特性作为原型基础进行请求响应的传统模式。**在 POJO 模式中，我们看到虽然实际进行请求的处理和响

应的载体依然还是 POJO 中的一个具体的方法，但是我们却并没有在这个方法中看到任何参数。所有的请求参数，将以 POJO 内部属性变量的形式存在并被调用。不仅如此，所有的执行结果对象，也同样以 POJO 内部属性变量的形式存在。这样一来，POJO 相对于某一次的响应是一个**有状态响应**。因为响应的处理流程、处理机制和处理结果，与当前 POJO 实例的内部属性的状态有关。

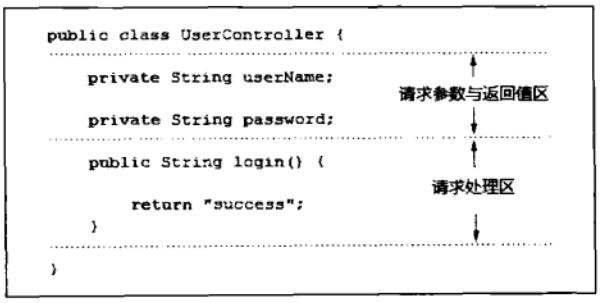


图 7-5　POJO 模式实现请求 − 响应模式的代码构成示意图

这就是 POJO 模式与前两种模式的最大区别：**进行请求响应的处理类自身是否是一个有状态的对象**。我们知道，传统的 Servlet 对象是一个无状态对象，也就是说它并不是一个线程安全的对象，我们不能在 Servlet 对象处理 Http 请求的过程中对 Servlet 对象内部的属性变量进行“安全”访问。这也成为 Servlet 对象处理 Http 请求无法逾越的一个障碍。从机制的角度讲，**无论是参数 − 返回值模式还是参数 − 参数模式，它们都只是 Servlet 模式的有效变种**。

而 POJO 模式则直接从概念上突破了 Servlet 对象的限制，将每一个请求的处理映射到一个线程安全的响应对象中去执行。因而从模式上讲，POJO 模式是对传统的 Servlet 模式的一个重大改进，是一种崭新的请求 − 响应模式的实现。

7.1.3　分歧和职责

7.1.3.1　分歧何来

无论哪一种对请求 − 响应的实现模式，实际上都是站在 Java 语言的语法角度，将请求 − 响应的过程进行编程元素的抽象化。很显然，三种不同的实现模式在实现上存在着一些分歧。它们之间的主要分歧在于：**不同的实现模式使用了不同的编程元素（方法参数、方法返回值、类的属性）来表达请求 − 响应模式中不同的逻辑语义**。而产生这些分歧的本质原因，我们可以从主观和客观两个不同的角度来分析：

❑ **主观上**——不同的实现模式，它们考虑问题的出发点不同

这一点我们曾经在比较参数－参数模式和参数－返回值模式之间的区别时提到过。前者作为一个底层的标准，不仅需要考虑处理结果的返回（这可能是一个偏向于数据层面的响应结果），还需要考虑后续的程序跳转（这就是偏向于控制层面的响应结果）。而后者，作为一个非底层的实现模式，则不需要考虑那么多的细节问题。

❑ **客观上**——不同的编程元素（方法参数、方法返回值、类的属性）所能够表达的逻辑功能存在着天然的差异

这一点在面向对象语言的语法层面表现得比较突出。很显然，既然一个编程语言设计了多种多样的编程元素，那么它本来的意思一定是期望不同的编程元素能够表达出不同的逻辑含义和功能。

以请求内容为例，我们可以看到请求内容主要有两种编程表现形式：方法参数和类的属性局部变量。很显然，作为方法参数更加符合“请求”这一动作的本意，也完全符合编程语言的天然设计原理。而作为类的属性，却站在另外一个角度考虑问题。类的属性作为一个类的局部变量，从外部访问的角度可以为它添加 setter 和 getter 方法，从而为这个类添加 JavaBean 的特性。在这种情况下，我们可以很轻松地对类的属性变量进行访问。相反，我们要对位于方法中的参数列表进行访问，却是一件难上加难的事情，因为这些参数列表只有在运行期才能够确定。

因此，我们不难得出以下结论：

结论 *对于编程元素作用的理解不同，直接导致了不同请求－响应模式之间的差异。*

我们可以发现，无论分歧发生在哪个具体的编程元素上，对于响应这个过程本身而言都只是一个“表象”而非“本质”。真正的本质，在于编程元素对于请求－响应过程不同元素职责的理解。

7.1.3.2 职责何在

虽然我们在这些实现模式上看到了分歧，不过既然它们都是对请求－响应模式的解读，它们之间一定有着共同的存在基础。我们可以发现，无论是哪种实现模式，**都是在使用 Java 语言的语法元素来和整个请求－响应模式的过程中的元素进行配对**。说得更加彻底一点，我们是在使用编程元素来对请求－响应的过程进行职责的划分。

那么，请求－响应模式的过程到底有哪些不可或缺的构成要素呢？我们还是使用一个示意图来进行描述，如图 7-6 所示。

从图中可以看到，我们把整个请求－响应的过程划分为“请求”和“响应”这两个相反的过程，并分别根据其特点继续将整个过程划分为四个部分：

- **请求内容**——请求的数据
- **请求处理载体**——进行逻辑处理的场所
- **响应内容**——逻辑处理的结果数据
- **响应跳转处理**——逻辑处理完成后的程序流转方式

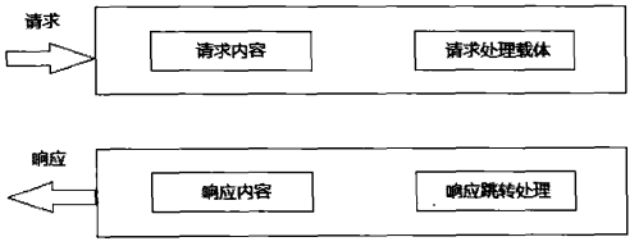


图 7-6　请求 – 响应模式的过程分析

如果把这上述四个部分的内容和请求 – 响应的实现模式结合起来看，我们就能看到它们与 Java 语言中的编程元素的对应关系，如表 7-1 所示。

表 7-1　请求 – 响应模式与编程元素的对应关系

请求 - 响应模式 / 编程元素	参数 – 返回值模式	参数 – 参数模式	POJO 模式
请求内容	方法参数	方法参数	类的属性变量
请求处理载体	响应方法	响应方法	响应方法
响应内容	返回值	方法参数	类的属性变量
响应跳转处理	返回值或框架组件	方法参数	返回值或框架组件

上述分析我们更多地站在请求 – 响应模式本身的角度进行。因而在图 7-6 中，我们发现请求和响应这两个过程似乎被人为阻断了。如果站在程序开发的角度来看待问题，我们就可以把图 7-6 进一步进行扩展，如图 7-7 所示。

在图 7-7 中，我们发现其中使用了类似数学中的“合并同类项”的方式，把站在程序开发的角度功能相类似的模块用虚线框在了一起，从而形成了两个重要的程序开发概念：

- **数据流**——描述程序运行过程中数据的流转方式及其行为状态
- **控制流**——控制程序逻辑执行的先后顺序

在整个请求 – 响应的过程中，**数据流实际上表现为数据内容，其核心包括数据请求和数据响应；而控制流实际上表现为方法（Method）进行逻辑处理的过程，包含程序的执行方向**。我们发现，不仅是请求 – 响应模式，任何处于运行状态的程序，在程序内部总是

有两股隐藏于内部的力量驱动着整个运行的过程：

结论 *数据流和控制流是两股隐藏于程序内部的神秘力量，是程序运行的核心驱动力。*

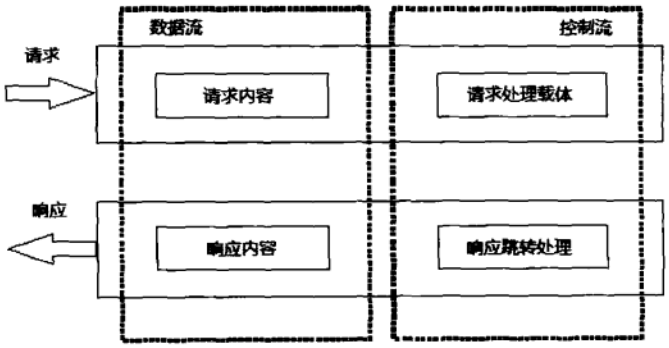


图 7-7 请求－响应模式的职责示意图

当我们在讨论请求－响应过程的职责划分时，不正是按照数据流和控制流的不同特性才对编程语言中的编程元素进行选择的吗？数据流和控制流，不也正是我们在程序开发的过程中所关注的两大职责吗？

基于以上结论，我们对 XWork 的研究，也将围绕着驱动程序运行的这两股核心力量。因而在下面的章节中，我们将首先讨论数据流和控制流的方方面面，再结合 XWork 的体系结构阐述 XWork 对于数据流和控制流独具匠心的诠释。

7.2 数据流和控制流

7.2.1 再谈 MVC

有了数据流和控制流的概念，我们不妨回过头来看看我们熟悉的 MVC 模式。我们还是通过引用第 2 章中曾经引用过的一张 MVC 模式的示意图来进行回顾，如图 7-8 所示。

在这张图中，我们又如何理解 MVC 模式与数据流和控制流之间的关系呢？

7.2.1.1 数据与模型

我们首先把关注的重点放到图 7-8 中靠近右侧的 Model 部分。从图中可以看到一个比较特殊的箭头，与 Data 部分形成了互相指向。这在逻辑意义上表明：**MVC 模型中的 Model 层，其本质就是“数据”。**

从动态的观点来看，数据在 MVC 的各个构成要素中流转（顺着箭头的方向）并在不

同的层次扮演着不同的角色。当程序运行起来之后，我们会发现正是由于数据的流转，才使得原本孤立和静态的元素形成了互动。因此，我们可以得出结论：

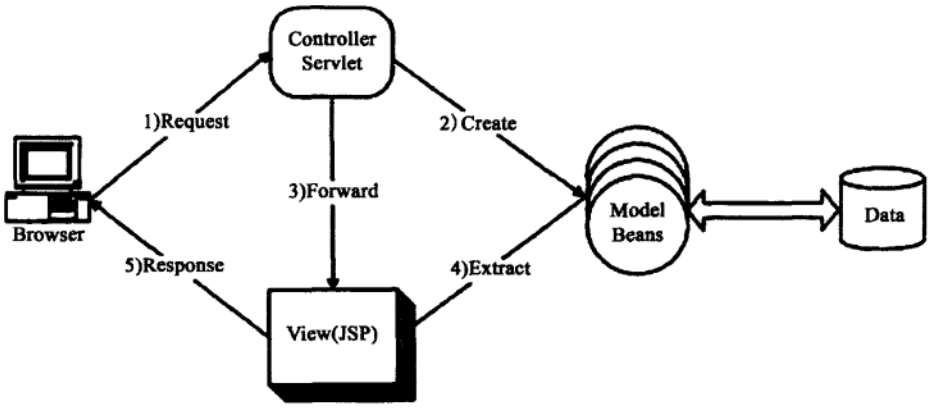


图 7-8　MVC 模型图

结论　*真正贯穿 MVC 框架并且将 MVC 的各个模块黏合在一起的是数据。数据作为黏合剂，构成了模块与模块间的互动载体，把 MVC 真正融合在了一起。*

从表面上看，MVC 三者是三个相互独立的职责模块，它们各司其职并有效地对 Web 层开发的各种元素解耦。此时，它们是三个完全“静态”的元素，是“分而治之”思想的体现。但是，一旦程序运转起来之后，模块与模块之间的交互关系实际上却由数据来承担。与那些“静态”的元素相比较，这些“动态”的流转关系似乎更加需要我们去关心。因为程序只有处于运行状态，我们才能从中获得所需要的功能特性。

Model 层的地位如此特殊，我们甚至无法为它找到一顶合适的“帽子”。因为我们可以看到在 MVC 模型中，Model 层实际上是一个动态元素，**它作为数据载体流转于程序之间，并在不同的程序模块中表现出不同的行为状态，这就是形成数据流的本质！**

7.2.1.2　控制层和流

让我们再来看图 7-8 的左半部分。我们发现，浏览器（Browser）、控制层（Controller）和视图层（View）三者之间形成了一个三角形，而这个三角形的两条三角边刚好反映出一个请求－响应的过程。其中，浏览器指向控制层的箭头，表示一个请求的过程；而视图层指向浏览器的箭头，则表示响应的过程。

在这里，我们可以很明显看到图的这半个部分所强调的是一个**方向性**。如果暂时忽略请求和响应的实际内容，我们会发现这整个流程之所以能够运转良好，得益于一个核心元素的掌控，这个元素就是位于图 7-8 顶端的控制层，所以控制层又被称为 MVC 的**核心控制器**。

从控制层所处的位置来看，它处于整个 MVC 示意图的顶端，这也显示出其独特之处。当我们从职责上去分析控制层，我们就不得不将连接控制层与其他 MVC 各个层次之间的箭头放在一起来看，从而得到控制层的四大职责：

- **控制层负责请求数据的接收**
- **控制层负责业务逻辑的处理**
- **控制层负责响应数据的收集**
- **控制层负责响应流程的控制**

对于一个 Web 开发人员而言，这四大职责实际上是非常重要的。不过到此为止，我们所讨论的还只是把控制层作为一个静态的元素来看待。那么我们为什么要提出**控制流**的概念呢？这里的“流”又指的是什么呢？

控制层到控制流的一字之差，实际上蕴含了一层关系模型：

结论 *控制流实际上是数据流融入控制层之后形成的逻辑处理和程序跳转的结果。*

也就是说，控制流之所以能够称为控制流，完全是因为它所控制的对象是数据，数据在逻辑处理过程中的形式和状态的变化，一定程度上促成了控制层的逻辑处理和程序跳转的结果。

这个结论对我们非常重要，因为它不仅将我们上一节中提到的程序运行的两股核心驱动力（控制流和数据流）黏合在了一起，它也同时成为众多 Web 框架在不同的请求 - 响应模式中进行选型的重要理论依据。在之后对 XWork 源码的分析中，我们还可以看到这一点。

7.2.2 数据载体的战争

我们在这里所说的 Model 层，实际上与第 2 章中所谈到的对象的运行模式可以结合起来看。在 Model 层，我们对于数据的关注点主要有两个：**数据存储和数据传输**。因而 Model 层总是以“**属性对象模式**”作为观察角度进行建模的。在这种情况下，数据（Model）所扮演的是一个**载体**的角色。

所谓载体，它首先必须具备一定的数据结构，那么到底什么样的数据结构和数据形式最适合用作 Web 层的传输介质呢？不同的 Web 层框架基于它们自己的理解，给出了众多不同的答案。在这里，我们选择其中比较典型的三种进行分析。

7.2.2.1 使用 Map 作为数据载体

Map 是一个基于键值对（Key-Value）的数据结构。使用 Map 作为数据载体，是一种非常直观的思维结果。因而，Servlet 标准在设计 HttpServletRequest 规范时就使用了类似 Map 的结构来进行数据交互。这样的设计被沿用至今，我们可以在其源码中看到接口的定

义，如代码清单 7-2 所示。

代码清单 7-2　ServletRequest.java

```
/**
 *
 * @return  返回一个不变的 java.util.Map 实例，包含所有以参数的名称作为 Key 值、参
 *  数实际值作为 Value 值的键值对。 Key 为 String 类型，Value 为 String[] 类型
 *
 */
public Map getParameterMap();

/**
 *
 * @return  返回根据参数名称返回参数的实际值，该名称的参数值可能有多个，以字符串数组
 *  的形式返回
 *
 */
public String[] getParameterValues(String name);

/**
 *
 * @return  返回根据参数名称返回参数的实际值，该名称的参数值可能有多个，以逗号隔开的
 *  形式返回一个字符串；如果该名称的参数值只有一个，则直接返回一个字符串
 *
 */
public String getParameter(String name);
```

如果在应用程序中使用 Map 作为数据交互的载体，实际上是沿袭了 Servlet 标准中规定的设计方式。这种方式虽然简单而有效，却存在着一些致命的问题：

❑ Map 作为一个原始的容器数据结构，弱化了 Java 作为一个强类型语言的功能

Java 作为一个强类型语言，丰富的数据类型和强类型语法检查成为其重要的特性。Map 的本质是一个容器结构，我们在操作容器结构时，更加关注其“容器”的性质而并非存储于容器内的元素。如果不使用 JDK 的泛型语法规则来规定容器内元素的数据类型，我们甚至可以毫无限制地将任何类型的数据置入容器之中。当然，前提条件是只要我们在取出元素时，能够自行分辨对应的数据类型。从我们的需求来看，我们更加关心的是隐藏在容器中的元素。所以，当这些很重要的数据元素失去了基本强类型语言的功能支持时，程序的可读性和可维护性显然都要大打折扣。

❑ 使用 Map 中的键值（Key）作为数据存取的依据，使得程序的可读性大大降低

当我们需要对 Map 结构进行存取时，我们的唯一依据就是键值（Key）。从 Map 本身的数据结构特性来看，它是一个不稳定的数据结构。Map 中的数据可以在任何情况下添加、修改和删除。当这些添加、修改和删除的逻辑散落在程序的各个角落时，我们将很难在某

一特定时刻确定当前 Map 中到底有哪些 Key 值可以进行存取。试想，如果我们在程序中使用一个 Map 作为参数进行传递，那么 Map 将对外屏蔽一切参数的细节。在运行期，我们甚至连 Map 中到底有哪些键值可用到都不清楚。从逻辑的角度，Map 的极其不稳定性也将造成程序逻辑的极度混乱。对于后期维护而言，这样的程序几乎是灾难性的，也违反了编程中最为基础的一条最佳实践：保证程序的可读性。

❑ **使用 Map 结构进行数据交互，无法实现必要的类型转化**

这一点从 ServletRequest 的接口设计中我们就可以发现。虽然可以根据 name 取到页面上提交上来的值，但那些值却全部都是 String 或者 String[] 类型。这与上面提到的第一点是相通的，Map 无法提供 Java 所具备的强类型语言的基本功能。这些类型转化方面的功能，我们需要在应用程序中自行处理。

值得一提的是，虽然 Map 结构有着这样或那样的缺陷，它却是最具备灵活度的一种数据结构，因为 Map 结构对于数据的格式、大小、数量都没有特殊的规定，从而成为最具扩展性的数据载体。在对数据格式要求不高、数据经常发生变化的情况下，还是有许多程序员乐意使用它作为数据交互的载体。

7.2.2.2 使用 FormBean 作为数据载体

针对 Map 结构的种种问题，以 Struts1.X 为代表的许多 Web 框架提出了使用 FormBean 作为数据交互载体的方案。FormBean 是对 Map 的一个改进，它的出现在一定程度上解决了 Map 这个容器结构带来的种种问题，因为 FormBean 在表现形式上是一个强类型的数据结构。就这一点而言，FormBean 是一种进步，至少在语法层面，我们获得了强类型语法检查的好处。

以 Struts1.X 为例，一个典型的 FormBean 看上去就像下面这样，如代码清单 7-3 所示。

代码清单 7-3 UserForm.java

```
public class UserForm extends ActionForm {

    private String userName;

    private String password;

    // 进行数据校验的方法，主要用于数据格式层面的校验，例如非空、类型判断等
    // 如果要进行业务验证，则应该在 Action 中进行
    public ActionErrors validate(ActionMapping mapping,
            HttpServletRequest request) {

        return null;
    }
```

```
    // 省略了 setter 方法与 getter 方法
}
```

除此之外，我们还需要在 struts-config.xml 中声明一下这个 FormBean，如代码清单 7-4 所示。

代码清单 7-4　struts-config.xml

```
<form-beans>
  <form-bean name="UserForm" type="example.UserForm" />
<form-beans>
```

看上去是不是很繁琐？站在 Struts1.X 的角度，FormBean 被设计成 MVC 模式的 M 部分，它承担着数据载体的重任。Struts1.X 在它的控制层中，把 FormBean 作为它的一个参数传入到执行逻辑的方法中去作为控制层获取 Model 层内容的交互途径，如代码清单 7-5 所示。

代码清单 7-5　UserAction.java

```
public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response) {
        // 从参数中获得 ActionForm，并强制转化成实际使用的 UserForm
        UserForm userForm = (UserForm) form;
        //从 Form 中取得数据
        String userName = userForm.getUserName();
        String userPwd = userForm.getUserPwd();

        return mapping.findForward(forward);
}
```

作为 MVC 设计的重要部分，FormBean 被框架赋予了过多的功能，而这些功能又需要被框架本身使用到。这就使得 FormBean 与整个框架形成了一个**强耦合**。除此之外，越来越多的针对 FormBean 的质疑被提出：

❑ **FormBean 被强制继承 ActionForm**

从 FormBean 的存在形式上看，我们可以看到它并不是一个纯正的 POJO，而是被强制依赖于 Struts1.X 的框架。这意味着我们几乎无法脱离框架和 Web 容器对 FormBean 中的逻辑进行测试。同时，FormBean 与框架的强耦合使得 FormBean 难以被其他诸如业务逻辑层或者持久层所使用。为此，曾经一度非常流行一个叫做 DTO（Data Transfer Object）的设计模式在中间做协调，这无疑从一个侧面显示了 FormBean 的尴尬位置。因为它只能在一个逻辑层次被某一类特定的框架相关的代码所使用。

❑ FormBean 被强制与框架的功能耦合在一起。

这是许多使用 Struts1.X 的程序员最无法接受的一点。例如，FormBean 从 ActionForm 中继承了数据校验机制，甚至 FormBean 与 Struts1.X 的标签也耦合在一起。这就使得程序失去了很多灵活性和扩展性。

❑ FormBean 在参数传递非常复杂的情况下，几乎无法工作

FormBean 看上去像一个普通的 POJO，却不具备 POJO 的基本功效。当页面所表达的逻辑非常复杂时，一个页面上的所有元素实际上需要被归并到多个 Java 实体中才足以满足数据传递的基本要求。然而，FormBean 却严格地对应一个页面中所有的元素映射，导致我们不得不在 FormBean 中编写大量互相之间毫无逻辑关联性的字段，这对于许多程序员而言几乎是一场噩梦。

面对种种质疑，Struts1.X 在后续的版本中做了种种改进。例如，FormBean 不被要求强制继承 ActionForm、提出 DynamicBean 的概念，等等。但是，FormBean 作为一个整体成为 MVC 架构中的一部分这样一个概念却从来没有被放弃，因此它也成为众多程序员对 Struts1.X 诟病的原因之一。

除了 Struts1.X 以外，SpringMVC 也是以 FormBean 作为基础概念进行数据载体的定义的。这一点，与 Struts1.X 和 SpringMVC 在请求－响应的实现模式有着深刻的关系。作为一个无状态的响应类，保持响应方法、响应数据和请求页面之间的一一对应关系是一种最简单的处理方法。然而，我们可以发现 FormBean 的种种弊端也直接宣布了 SpringMVC 在数据载体的流转中会遭遇到一些不雅观的实现。

应该说，FormBean 是一个时代的重要尝试。它的存在，为日后发展纯 POJO 的数据载体奠定了基础。同时，它在特定的历史时期也发挥了一定的作用。因此，针对 FormBean，我们需要辩证地来看待，不能一概而论，至少从数据形式上讲，它相对于 Map 是一个重大的进步。

7.2.2.3 使用 POJO 作为数据载体

FormBean 已经在数据载体的形式上做出了榜样。从 Java 语言自身的角度来讲，使用 POJO 进行数据交互也是我们的最终目标，因为它至少可以为我们带来一些显而易见的好处：

❑ 作为 JavaBean，POJO 是一个具备语义的强类型，不仅能够享受编译器的类型检查，还能够自由定义我们所需要表达的语义。

❑ POJO 不依赖于任何框架，可以在程序的任何一个层次（如业务逻辑层，甚至是持久层）被复用。

❑ POJO 突破了 FormBean 对于页面元素唯一对应的限制，我们可以将一个页面中的元素自由映射到多个 POJO 中去。

使用 POJO 作为数据交互载体，我们的程序会变得更加简单。我们不妨对上面使用

FormBean 的例子进行重构，看看使用 POJO 时，程序看起来是什么样子，其源代码如代码清单 7-6、代码清单 7-7 和代码清单 7-8 所示。

代码清单 7-6　User.java

```
public class User {

    private String name;

    private String password;
    // 省略了 setter 与 getter 方法
}
```

代码清单 7-7　user.jsp

```
<form method="post" action="/registration.action">
     user name: <input type="text" name="user.name" value="downpour" />
     password: <input type="password" name="user.password" value="pass" />
     <input type="submit" value="submit" />
</form>
```

代码清单 7-8　UserAction.java

```
public class UserAction implements Action {

     private User user;

     public String execute() {
          // 可以直接在这里使用 user 对象，因为它已经被作为传入的参数了

          return "success";
     }
     // 省略了 setter 方法与 getter 方法
}
```

整个过程显得更加清晰明了。作为一个 JavaBean，User 无须再继承任何框架相关的类，而是一个干净的 POJO。当 Action 引用 User 作为参数时，直接将其作为 Action 的局部变量使用即可。我们甚至可以增加多个局部变量来映射不同的页面元素，这也为程序设计带来了极大的便利。

7.2.3　控制流的细节

谈完了数据载体，我们再回到控制流的话题。在之前的讨论中，我们曾经列举了控制

层的四大职责：

- ❑ 控制层负责请求数据的接收
- ❑ 控制层负责业务逻辑的处理
- ❑ 控制层负责响应数据的收集
- ❑ 控制层负责响应流程的控制

如果我们不使用任何框架，下面这个使用 Servlet 的例子，可以从代码的角度大致勾勒出控制层的四大职责，如图 7-9 所示。

```
请求参数接收 ── String name = request.getParameter("name");
               Date age = new SimpleDateFormat("yyyy-MM-dd").parse(request.getParameter("age"));
               User user = new User(name, age);

业务逻辑处理 ── userService.register(user);

响应数据收集 ── request.setAttribute("user", user);

响应流程控制 ── RequestDispatcher dispatcher = request.getRequestDispatcher("/index");
               dispatcher.forward(request, response);
```

图 7-9 控制层的四大职责

这是我们在不使用任何框架的情况下编写的代码片段。或许有的读者会说，这里有刻意渲染控制层的四大职责之嫌。不过当我们静下心来仔细思考整个过程，会发现在这其中还真没有什么可以直接省略的代码段。因为这四段代码恰巧反映出一个完整的事件处理流程。

当一个请求－响应的过程重复 1000 遍，这样的四段式也就需要重复 1000 遍。我们不禁要问，在控制层中，我们真正关心的重中之重到底是什么？

结论 *控制层的核心职责是处理业务逻辑。*

明确这一结论，对于一个开发框架而言非常重要。因为它直接为开发框架指明了目标：**控制层应该更加关注其核心的职责，而其他的辅助逻辑则由框架帮忙来完成。**

为了完成这一目标，以 XWork 为首的开发框架就开始对位于控制层的这四段代码进行规划。因为我们发现，在上述这四段式的代码中，只有对 userService 的逻辑调用是我们在控制层所关心的核心内容。而除此之外的代码，则应该通过合理的设计，转化为一个标准而规范的事件处理流程。

要完成这种**对事件处理流程的规范化**，需要若干个循序渐进的步骤来完成：

- ❑ **划分事件处理流程步骤**

这个步骤偏向于理论的范畴，不同的框架对于事件处理流程的理解也完全不同。总体来说，可以对事件处理流程中的不同处理职责进行语义抽象，从而将类似的流程步骤归为一类。

❑ **定义事件处理节点对象**

这个步骤就偏向于实际编程的范畴了。根据上一个步骤中对于事件处理流程的划分，在这一步中我们所需要做的就是定义不同的 Java 对象来和每一个事件处理流程步骤相对应。每一个 Java 对象在这里就被称为**事件处理节点对象**。例如，XWork 框架中所定义的 Interceptor、Action、Result 等都是事件处理节点对象。

❑ **组织事件处理节点对象的执行次序**

这个步骤是整个框架对事件处理流程进行规范化的核心。它规定了在上一个步骤中所有的这些事件处理节点对象以何种方式相互调用并配合完成整个事件处理流程。这个步骤我们往往需要通过定义一些辅助对象以及设计模式的共同配合来完成，例如我们在第 4 章中所提到的责任链模式就可以用在这里。

在这里我们从另外一个侧面看到了框架的设计过程。控制流的实现细节，透过框架也能够更好地进行组织。在之后的章节中，我们不仅将看到 XWork 框架对于事件处理节点对象的定义，还能看到 XWork 如何通过有效的手段，将这些节点串联起来使之具备强大的事件处理能力。

7.3　XWork 概览

在了解了数据流和控制流的来龙去脉之后，我们再来看看 XWork 中实现这两大核心驱动力的编程元素以及它们之间的调用关系。相信有了之前的所有概念做铺垫，无论是 XWork 的宏观视图还是微观视图，读者理解起来应该都可以驾轻就熟。

7.3.1　XWork 的宏观视图

XWork 的宏观构成示意图是 XWork 体系结构的核心，这个示意图我们曾经在第 3 章中展示过，不过当时我们引入此图的主要目的是说明 XWork 框架是 Struts2 的一个重要组成部分，具有非常丰富的内容。因而当时我们并没有对其中的构成要素进行深入的分析。经过了之后各个章节的讲述，或许读者在这里会对这个示意图有一番更深刻的见解。整个示意图如图 7-10 所示。

事实上，这张 XWork 的宏观示意图是整个 XWork 乃至整个 Struts2 的核心。此图内涵丰富，几乎涵盖了 XWork 的元素构成、XWork 中元素的调用关系、XWork 的执行层次以及 XWork 与外部调用接口之间的关系等所有 XWork 框架的核心内容。

在这里，我们将首先帮助读者理解这张示意图中的一些基本概念和基本的逻辑层次，在下一章中，我们将对示意图中的每一个元素进行详细的分析。

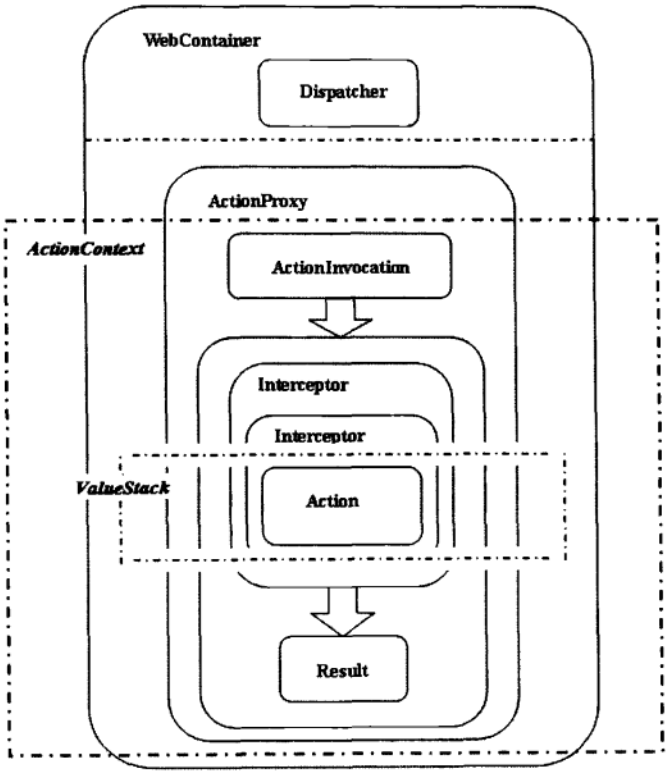


图 7-10 XWork 的宏观示意图

大家对于这张示意图的第一印象一定是这张图中不同类型的框框（有虚线的，也有实线的）。而这些不同的框框所围起来的构成要素，实际上代表着 XWork 框架中三个不同的体系结构：

❑ **核心分发器**

核心分发器 Dispatcher 位于整个示意图的最外层的 Web 容器中，它本身不属于 XWork 框架的组成部分。我们在这里把它纳入 XWork 宏观示意图中，并作为一个重要的组成体系的主要原因在于它在 Struts2 中有着非常重要的地位。被称为核心分发器的 Dispatcher 运行于 Web 容器中，却是 XWork 框架的调用者和驱动执行者。在图中，我们可以看到在核心分发器 Dispatcher 和下面的 XWork 元素之间有一条明显的虚线分割线作为它们之间的区分标志。

❑ **XWork 的控制流体系**

XWork 的控制流体系是指 XWork 进行请求响应的一系列执行元素，这一系列执行元素包括：ActionProxy、ActionInvocation、Interceptor、Action 和 Result。**这些元素位于图中的下半部分，并被一个实线框封装于内部**，由 ActionProxy 驱动执行。它们之所以被称为 XWork 的控制流体系，是因为这些元素是真正的请求响应的逻辑处理载体，控制着整个请求响应的执行过程。

❑ **XWork 的数据流体系**

XWork 的数据流体系是指 XWork 在进行请求响应时所依赖的一个数据环境，而这个数据环境中包含了两大元素：ActionContext 和 ValueStack。**这两大元素在图中所在的位置也比较特殊，它们都被虚线框包含其中**。ActionContext 所在的虚线框同时**囊括**了 XWork 控制流体系中的所有元素；而 ValueStack 所在的虚线框则**囊括**了核心控制元素的 Action。这表明 XWork 的数据流元素在不同的控制流执行元素之间形成了**有层次的共享**，它虽然不直接参与控制流的执行体系，却是控制流执行过程中必不可少的核心依赖。

如果单单把重心放在 XWork 框架本身，我们可以发现 XWork 的体系结构正是按照控制流和数据流这两大核心驱动力进行划分的。结合核心分发器 Dispatcher，我们可以归纳出这三大体系结构之间的层次关系：

❑ **调用关系**

“调用关系”，表明了 XWork 框架的空间范围，因为只有框架外的元素才会对框架本身产生“调用关系”。因而我们可以发现核心分发器 Dispatcher 并不属于 XWork 框架的范畴，但是它却驱动着 XWork 框架的调用执行。

❑ **映衬关系**

“映衬关系”，则是一种非常微妙的关系。所谓“映衬”，实际上体现了一种“你中有我，我中有你”的水乳交融的联系。XWork 框架的控制流体系的执行基础是其数据流中的元素；而另外一个方面，失去了控制流的执行流程，数据流的所有元素也就没有存在的意义了。

我们可以看到，“解耦合”这样一个开发中的最佳实践被 XWork 充分挖掘。XWork 将数据流和控制流这两大驱动程序运行的基本力量进行物理隔离，将它们分派到不同的执行元素之上。但在运行期，两者又通过某种编程手段有机联系在了一起。这种看似很松实际很紧的联系正是贯穿 XWork 框架总体设计的一个核心思想。

7.3.2　XWork 的微观视图

正如图 7-10 所示，实际上所有的 XWork 微观构成元素也通过示意图完全呈现在读者的面前。对于所有这些微观元素的解读，也自然离不开对图 7-10 中元素和元素之间关系的解读。因而在这里，我们还是从 XWork 的数据流和控制流这两个截然不同的体系上，

对 XWork 的微观构成给出一个大致的介绍。在下一章中，我们不仅会细化这些微观构成元素，还将具体展开这些微观元素之间的关系。

7.3.2.1 数据流元素

XWork 的数据流体系，在图 7-10 中以虚线框的形式存在。我们可以在虚线框中看到构成数据流的两大元素：ActionContext 和 ValueStack：

❑ **ActionContext——数据环境**

ActionContext 是一个独立的数据结构，其主要作用是为 XWork 的执行提供数据环境。无论是请求的参数，还是处理的返回值，甚至一些原生的 Web 容器对象，都封装于 ActionContext 的内部，成为 Struts2/XWork 执行时所依赖的数据基础。

❑ **ValueStack——数据访问环境**

ValueStack 本身是一个数据结构，其主要作用是对 OGNL 计算进行扩展。因而，位于 ActionContext 之中的 ValueStack 则赋予了 ActionContext 进行数据计算的功能，从而使得 ValueStack 自身成为一个可以进行数据访问的环境。

在 XWork 对数据流的设计中，首要的考虑因素是功能性。根据之前我们对数据流的分析，构成数据流的元素有两大基础的功能性要求：**数据存储**和**数据传输**。仔细分析 ActionContext 和 ValueStack 这两大元素，我们会发现 ActionContext 刚好是一个数据存储的容器，而 ValueStack 则接过了数据传输的责任大旗。这两大元素的相互配合，很好地诠释了数据流的完整过程。

那么，XWork 对于数据流的设计有什么独到之处呢？

结论 *XWork 对于数据流设计的亮点，在于数据流元素被设计成独立的数据结构。*

结合图 7-6，我们可以看到数据流的主要构成：请求内容和响应内容。回顾一下传统的参数 - 返回值模式和参数 - 参数模式，我们会发现无论是请求内容还是响应内容，它们在表现形式上都作为方法的一个重要组成部分（要么作为方法的参数，要么作为方法的返回值）。而**作为控制流主要载体的方法本身，对数据流元素形成了语法依赖**。也就是说，在这种情况下，数据流和控制流之间是天然耦合的。因为作为控制流实现的主体方法，它与参数和返回值在语法层面被紧密联系在一起。

而 XWork 采用了与控制流完全独立的对象实体来封装所有的数据流元素，并将控制流中的核心处理要素 Action 置于数据流之中，使两者形成水乳交融的关系。这种设计理念基于“解耦合”思想，使得原本无法分离的编程元素成为了形式上独立、运行上又紧密联系在一起的组件。这一点，正是 XWork 在数据流设计上的精华之处，也是读者需要细细品味的地方。

7.3.2.2　控制流元素

XWork 的控制流体系，在图 7-10 中以实线框的形式存在。我们在实线框中可以看到构成控制流的元素主要有五个：ActionProxy、ActionInvocation、Interceptor、Action 和 Result。

这五大元素从功能逻辑上进行划分，还可以分成两类：其中的 Interceptor、Action 和 Result 被用于定义事件处理的基本流程，我们称之为事件处理节点；ActionProxy 和 ActionInvocation 在事件处理的过程中起到的作用主要是对事件处理节点进行调度执行，我们将其称之为**事件处理驱动元素**。

我们在 7.2.3 节曾经深入分析过控制流的内部实现细节，并归纳了控制流的四大职责。不过当时我们并没有点出其中蕴含的一个 XWork 设计中的潜台词：

结论　XWork 认为，一个事件处理的流程是有规律并可以被继续细化的。

正是基于这样一个基本观点，XWork 建立起了一套定义事件处理流程的方法，并将它们映射到了具体的 Java 对象中。

❑ **Action——核心处理类**

Action 是 XWork 所定义的一个核心的事件处理接口。这个接口定义仅仅定义了一个没有参数的响应方法，从而使得所有实现 Action 接口的事件处理类都自然而然地工作在属性 - 行为模式之上。其中，响应方法的内部完成对核心业务的处理，而事件处理类的内部属性则成为响应方法进行业务处理的数据来源和响应结果。

❑ **Interceptor——拦截器**

Interceptor 本身是 AOP 的概念，表示对程序的某个逻辑执行点进行拦截，从而能够在这个逻辑执行点之前、之后或者环绕着这个逻辑执行点进行逻辑扩展。在 XWork 中，Interceptor 的拦截对象是核心处理类 Action，在 Action 的周围定义了一个环绕的逻辑扩展层次，其主要作用就在于能够在核心处理类 Action 的执行之前、之后进行自定义的逻辑行为扩展。

❑ **Result——执行结果**

Result 是 XWork 定义用以对核心处理类 Action 执行完毕之后的响应处理动作。Result 被定义为一个独立的逻辑执行层次，其主要作用是使核心处理类 Action 能够更加关注核心业务流程的处理，而将程序的跳转控制逻辑交给 Result 来完成。

通过 Action、Interceptor 和 Result 这三大元素的相互配合，一个完整的事件处理流程可以定义为：**以 Action 为业务处理核心，Interceptor 进行逻辑辅助，Result 进行响应逻辑跳转的具有丰富的执行层次的事件处理体系**。

回顾一下在 7.2.3 节有关控制流细节的描述，我们会发现三大元素的完整定义实际上完成了我们**对事件处理流程进行规范化**中的前两个步骤：**划分事件处理流程步骤**和**定义事件处理节点对象**。而整个规范化流程中最为关键的步骤，也就是**组织事件处理节点对象的执行次序**，这一点 XWork 通过另外两个辅助对象来完成：

❑ **ActionProxy——执行环境**

ActionProxy 是整个 XWork 框架的执行入口。ActionProxy 的存在，相当于定义了一个事件处理流程的执行范围，规定在 ActionProxy 内部的一切都属于 XWork 框架的管辖范围，而在 ActionProxy 之外，只能以调用者的身份，与整个 XWork 的事件执行体系进行通信。因此，ActionProxy 的主要作用就在于对外屏蔽整个控制流核心元素的执行过程，对内则为 Action、Interceptor、Result 等事件处理节点对象提供一个无干扰的执行环境。

❑ **ActionInvocation——核心调度器**

ActionInvocation 是组织起 Action、Interceptor、Result 等事件处理节点对象执行次序的核心调度器。ActionInvocation 被封装于 ActionProxy 的内部，是 XWork 内部真正事件处理流程的总司令，它的执行调度过程，实际上控制着整个 XWork 事件处理体系的执行命脉。

在 XWork 的微观构成中，我们可以发现 XWork 的设计理念始终是**将逻辑职责分派到最合适的编程元素之上**。或许在这里还无法体会出 XWork 对这些具体元素的划分依据，不过我们已经可以从这些元素的名称中看出它们在框架中所处的不同地位。在下一章中，我们将对这些元素进行详细解读。

XWork 控制流元素的一个形象比喻

XWork 控制流被划分为五大元素：Action、Interceptor、Result、ActionProxy、ActionInvocation。我们可以使用一个战斗序列来对这五大元素之间的关系进行诠释。

每当一个战役打响的时候，总指挥部总是需要分派一个具体番号的部队（ActionProxy）来执行战斗。任何一支部队，都有主力军（Action）和策应部队（Interceptor）。主力军（Action）负责核心战斗，而策应部队（Interceptor）则负责对主力部队进行策应和援助。然而，所有的战斗指令都是由部队的指挥官（ActionInvocation）决定的。指挥官（ActionInvocation）是一个部队（ActionProxy）的核心，他负责主力部队（Action）和策应部队（Interceptor）的调度。当一场战斗结束以后，指挥官（ActionInvocation）还要负责指挥部队下一步的动向（Result），是乘胜追击敌人，还是继续待命。

7.4　小结

别具匠心是我们对 XWork 的基本设计原理所做的评价。这里所谈到的别具匠心，我们主要在本章中看到了两个方面：其一，XWork 对于请求 - 响应模式采取的是颠覆传统 Servlet 模式的实现方式；其二，XWork 对于事件处理流程的定义，突破了“单一元素”和“顺序执行”这两大壁垒。

我们在本章中所谈到的请求 - 响应模式的实现哲学，是当下所有基于 MVC 模式的开发框架的一个基本总结。事实上，当读者理解了这些不同的实现模式所蕴含的内在含义就会发现，所谓的框架都是浮云。我们关注的核心，应该是 Web 开发中所涉及的职责和处理流程的合理定义。因此，学习框架只是帮助我们更好地理解这些流程的内在含义罢了。

当我们对 XWork 的宏观视图和微观视图有了大致的了解以后，会发现 XWork 对于职责的诠释正是我们梦寐以求的。虽然还未对 XWork 的实现细节进行展开，然而从框架的宏观视图和微观构成中，我们已经可以领略到其独到之处。

以下问题是本章的重点，读者可以回过头来再仔细品味一下这些概念性的问题：

- 基于人机交互的请求 - 响应模式主要由哪三大要素构成？
- 在 Java 语言中，有哪三种主要的请求 - 响应的实现模式？它们的主要分歧在哪里？
- 什么是数据流？什么是控制流？
- 数据在进行存储和传输时，有哪三种具体的表现形式？
- 如何才能对一个事件处理流程进行规范化？有哪些步骤？
- XWork 的宏观视图中蕴含着哪三大体系？它们互相之间形成了什么样的关系？
- XWork 的数据流体系由哪两大元素构成？它们之间有什么关系？
- XWork 的控制流体系由哪五大元素构成？它们在逻辑上又如何进行划分？

第 8 章　庖丁解牛——XWork 元素详解

在上一章中，我们讨论了 XWork 框架的设计原理，围绕着请求－响应这一基本的人机交互模式，XWork 采取了控制流与数据流相互交织、相互映衬的方式来实现其基本的体系结构。如果我们把一个框架比作一个人，那么框架的设计原理就是这个人的魂之所在；框架的宏观视图就好比架起了整个人的骨架；而框架的微观构成则是一个人的血和肉。

在本章中，我们就来重点关注构成 XWork 框架的这些有血有肉的微观元素。不过本书一贯坚持一个观点：**孤立地看待任何一个框架的构成元素势必造成“盲人摸象”的结果。**因为程序总是处于运行状态，构成框架的各个元素之间也总是有着千丝万缕的联系。因此，我们在讲解 XWork 构成元素的过程中，也将以这些元素之间的关系作为逻辑主线。

8.1　深入 XWork 宏观视图

在第 7 章中，我们给出了 XWork 的宏观视图，并且将 XWork 的宏观视图划分为三个不同的体系以及两大关联关系。事实上，XWork 的宏观视图中所蕴含的远不止这些。在本章中，我们还是从对 XWork 的宏观视图的解析入手，只不过这里的关注重点将不再停留于 XWork 的体系结构，而是努力探寻 XWork 框架的各个元素之间的关系。

8.1.1　数据流体系

在对 XWork 宏观视图的解读中，我们很容易形成元素和元素之间的类别划分。因为在图中（参见图 7-10），我们已经为读者明确标明了元素和元素之间的区分标识：**实线框**和**虚线框**。

首先来看看虚线框。我们在这里把虚线框的部分单独抽取了出来，如图 8-1 所示。

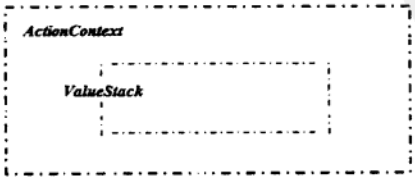


图 8-1　XWork 数据流体系简图

我们在这里特地刨去了所有的控制流元素，仅仅剩下 ActionContext 和 ValueStack 这两大数据流元素，目的在于希望读者能够更加清楚地看到数据流体系中这两大元素之间的关系。从图中我们可以很明显地得出结论：

结论　ActionContext 与 ValueStack 之间的关系是从属关系。

还记得我们在第 2 章讲面向对象概念的时候所谈到的“从属关系”吗？我们在这里所说的从属关系，实际上是说：**ValueStack 是 ActionContext 的一个组成部分**。有关这一点，我们可以用源码来证实。ActionContext 相关的源码如代码清单 8-1 所示。

代码清单 8-1　ActionContext.java

```
public class ActionContext implements Serializable {

    public static final String VALUE_STACK = ValueStack.VALUE_STACK;

// ActionContext 中存储数据的上下文环境，ValueStack 就存储在其中
Map<String, Object> context;

    /**
     *
     * @param key
     * @param value
     */
    public void put(String key, Object value) {
        context.put(key, value);
    }

    /**
     *
     * @param key
     * @return value
     */
    public Object get(String key) {
        return context.get(key);
    }

    /**
     * 设置 ValueStack
     *
     * @param stack the OGNL value stack
     */
    public void setValueStack(ValueStack stack) {
        put(VALUE_STACK, stack);
    }
```

```
    /**
     * 获取 ValueStack
     *
     * @return the OGNL value stack
     */
    public ValueStack getValueStack() {
        return (ValueStack) get(VALUE_STACK);
}

// 这里省略了许多其他代码
}
```

这种从属关系，无论是在源代码级别还是在宏观视图上都表达得非常清晰。那么XWork在设计时，基于什么样的考虑而将ValueStack置于ActionContext之中呢？这种从属关系到底为数据流体系带来了什么样的好处呢？为了讲清楚这个问题，我们还是要从ActionContext和ValueStack各自的特性和职责谈起。

我们在上一章曾经谈到，**数据流有两大特性：数据和流**。所谓“数据”，强调的是数据作为一个载体，其蕴含的信息和内容。所谓“流”，强调的是一个动态的过程，也就是数据访问和数据传输。

在XWork所定义的这两大元素中，ActionContext刚好满足了“数据”的特性。这一点我们从它的名称上就可以看出来：Context的含义是“环境”，是一个具有空间概念的元素。因而，**ActionContext所表现出来的空间的概念，恰好成为数据载体进行存储的天然基石**。

结论 ActionContext是XWork的数据流实现元素。作为一个数据载体，它既负责数据存储，又负责数据共享。

而XWork数据流中的另外一个元素ValueStack，我们似乎无法从其名称上获得什么有价值的信息。不过我们在第7章中已经对这个元素做过一个简单的介绍：

结论 ValueStack是一个具备表达式引擎计算能力的数据结构。

有了这个结论，大家是否豁然开朗了呢？我们曾经反复提到，表达式引擎的引入，就是为了解决数据访问和数据传输中的困境。ValueStack正是为了解决数据流体系中的数据访问和数据传输的特性而定义的特殊对象。因此，ValueStack提供了一个可以进行表达式引擎计算的场所。这样一来，它也成为能够与ActionContext配合得相得益彰的重要元素，其数据计算功能恰好能够使得ActionContext如虎添翼。

结论　XWork 将 ValueStack 置于 ActionContext 中的目的在于为静态的数据添加动态计算的功能。

这个结论我们需要从正反两个方面来理解：

- ActionContext 无法脱离 ValueStack 而单独存在，否则所有的数据载体就如一潭死水。失去了流动性的数据流，只能称之为数据载体而非数据流
- ValueStack 无法脱离 ActionContext 而单独存在，否则 ValueStack 就没有了数据计算的基础。失去了数据的数据流，只能称之为一个表达式计算工具而非数据流

由此可见，ActionContext 和 ValueStack 这两大元素的不同特性，互为对方的有力补充，为两者的精诚合作打下了坚实的基础。

8.1.2　控制流体系

在了解虚线框所代表的数据流体系后，我们再来看看实线框所代表的控制流体系。与之前讨论数据流体系时一样，我们刨去了所有的数据流元素，整个控制流元素就如图 8-2 所示。

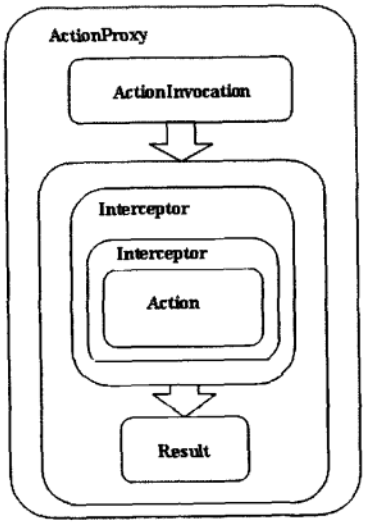


图 8-2　XWork 的控制流体系

控制流体系从元素构成上要比数据流复杂得多。至少从数量上，控制流体系的构成元素要比数据流体系多。在其中，有两层关系值得我们深入探讨。这两层关系分别是：事件处理节点与事件驱动元素之间的关系以及核心事件处理节点与辅助事件处理节点之间的关系。

8.1.2.1 事件处理节点与驱动元素

在第 7 章中，我们曾经对控制流元素按照其职责的不同进行划分，主要将控制流元素分成两类：**事件处理节点元素**（由 Action、Interceptor 和 Result 构成）和**事件处理驱动元素**（由 ActionProxy 和 ActionInvocation 构成）。

这两大分类，实际上我们可以通过图 8-2 得到充分的验证。作为事件处理驱动元素核心的 ActionInvocation 通过箭头对事件处理节点进行逻辑调用。这也引出了 XWork 控制流体系中的一个重要结论：

结论 *在 XWork 的控制流中，事件处理驱动元素对事件处理节点元素形成调用关系。*

这一层 XWork 控制流体系元素之间的关系，实际上是通过“看图说话”的方式得到的，这并不难理解。事实上，这一层调用关系也完全符合我们对事件处理流程的规范化要求。这一点我们已经在第 7 章讨论过，这里就不再详述了。

值得重视的是，ActionInvocation 对事件处理节点的执行调度，是整个 XWork 控制流体系的运行核心。因为 XWork 中的每一个事件处理节点，都被定义为接口。使用 XWork 来进行事件处理的过程，实际上是对每一个事件处理节点实例化的过程。但是无论如何定义和扩展这些节点，都离不开 ActionInvocation 的统一调度执行。正如我们在上一章中提到过的一个比喻：事件处理节点，无论是 Action、Interceptor 还是 Result，它们都只是战斗序列中的某一支部队。到底谁先打响战斗、谁主攻、谁策应，都离不开 ActionInvocation 这位总司令的指挥和调度。

8.1.2.2 主力部队与策应部队

事件处理节点和事件处理驱动元素之间的关系，是控制流体系中的核心逻辑。我们在这里还需要关注的，是图 8-2 下半部分中不同的事件处理节点之间的关系。具体来说，就是 Action 和 Interceptor 之间的关系。如果用上一章中我们所提到的那个战斗序列的比喻，那就是主力部队与策应部队之间的关系。

在图 8-2 中，我们可以看到 Interceptor 和 Action 之间形成了一个包裹结构。一个 Interceptor 套着另外一个 Interceptor，一层接着一层，并最终把 Action 包裹在最里面。众所周知，包裹结构在计算机语言中是一个典型的先进后出的“栈”结构。这也正好印证了我们曾经在介绍配置元素时提到过的 interceptor-stack 这个节点。而这样的数据结构，有着以下的特点：

- **在整个栈结构中，除了位于栈底的 Action 以外，栈中的其他元素都是 Interceptor 对象**
- **由于 Action 位于栈的底部，根据栈结构“先进后出”的特性，当试图把 Action 对**

象拿出来执行时，我们必须首先把位于 Action 之上的所有 Interceptor 依次拿出来执行

- **每个位于栈中的 Interceptor，除了需要完成它自身的逻辑外，还需要指定下一步的执行对象**

因此，Interceptor 与 Action 在数据结构层面上形成的“栈结构”在一定程度上直接影响了它们互相之间的执行次序。

ActionInvocation 在对事件处理节点进行逻辑调用时，是将 Action 和 Interceptor 这两大元素捆绑在一起进行调度的。结合这两大元素在控制流示意图中所表现出来的关系，我们往往把 Action 和 Interceptor 共同构成的结构称为 **XWork 的执行栈**。而整个调度的过程，也充分体现出了 Action 和 Interceptor 之间所表现出来的栈特性：逐次调用 Interceptor 元素并执行，如果所有的 Interceptor 元素都执行完毕了，那么调用 Action 执行。

XWork 对于 Action 和 Interceptor 的设计，充分体现了三个核心的设计理念：

- **对于事件处理，能够进行进一步的规范化的流程细化**
- **对于细化的流程，能够具备高度的可扩展性**
- **流程细化后的事件处理节点之间表现出明显的执行层次**

在之后的章节中，我们还将看到这三个核心理念在源码中如何得以体现。不过在这里，我们还是应该强调一下 Action 和 Interceptor 在职责上的不同。正如本节标题所说的那样，主力和策应之分，是两者最重要的区别。因而，大家在具体的实践过程中，应时刻牢记这一原则，并对每一个请求的处理进行合理的职责抽象，最后将它们分派到不同的对象上去执行。

8.2　数据流体系——相互依存

8.2.1　ActionContext——一个平行世界

ActionContext 是 XWork 中最重要的概念之一，它是整个 XWork 事件处理体系的数据环境。在这个数据环境中包含了所有事件处理过程中所需要的数据对象。因而，每一次的事件响应，都有一个 ActionContext 与之对应。

在之前的分析中，我们知道 ActionContext 作为一个数据环境，其主要职责有两个：数据存储和数据共享。接下来，我们就从源码角度对这两大职责进行分析。

8.2.1.1　数据存储空间

从 ActionContext 这个类的名称中，我们可以读到一丝数据存储的信息。因为 Context

这个词，在日常编程中往往被翻译成“上下文环境”，而通常意义上我们所说的上下文环境，就是一个巨大的数据存储空间。这一点，我们可以在 ActionContext 的源码中找到一些端倪。

ActionContext 的部分源码，如代码清单 8-2 所示。

代码清单 8-2 ActionContext.java

```
public class ActionContext implements Serializable {

    // 此处省略了许多常量定义

    // ActionContext 内部真正进行数据存储的场所
    Map<String, Object> context;

    /**
     * 创建一个新的 ActionContext，将传入的 Map 作为 context
     *
     * @param context
     */
    public ActionContext(Map<String, Object> context) {
     this.context = context;
    }

    /**
     * 设置 ActionInvocation
     *
     * @param actionInvocation
     */
    public void setActionInvocation(ActionInvocation actionInvocation) {
     put(ACTION_INVOCATION, actionInvocation);
    }

    /**
     * 返回 ActionInvocation
     *
     * @return the action invocation (the execution state)
     */
    public ActionInvocation getActionInvocation() {
        return (ActionInvocation) get(ACTION_INVOCATION);
    }

    /**
     * 设置一个被 Map 封装后的 ServletContext
     *
     * @param application
     */
    public void setApplication(Map<String, Object> application) {
     put(APPLICATION, application);
```

```
    }

    /**
     * 返回被 Map 封装后的 ServletContext
     *
     * @return a Map of ServletContext or generic application level Map
     */
    public Map<String, Object> getApplication() {
        return (Map<String, Object>) get(APPLICATION);
    }

    /**
     * 设置当前 ActionContext 中的 contextMap
     *
     * @param contextMap
     */
    public void setContextMap(Map<String, Object> contextMap) {
        getContext().context = contextMap;
    }

    /**
     * 返回当前 ActionContext 中的 contextMap
     *
     * @return the context map
     */
    public Map<String, Object> getContextMap() {
        return context;
    }

    /**
     * 设置错误处理转化类，这些类可能会在 action 执行时被触发
     *
     * @param conversionErrors
     */
    public void setConversionErrors(Map<String, Object> conversionErrors) {
        put(CONVERSION_ERRORS, conversionErrors);
    }

    /**
     * 设置错误处理转化类
     *
     * @return
     */
    public Map<String, Object> getConversionErrors() {
        Map<String, Object> errors = (Map) get(CONVERSION_ERRORS);
        // 如果不存在，则返回一个空的 HashMap
        if (errors == null) {
            errors = new HashMap<String, Object>();
            setConversionErrors(errors);
```

```
        }

        return errors;
    }

    /**
     * 设置当前action的Locale
     *
     * @param locale the Locale for the current action
     */
    public void setLocale(Locale locale) {
        put(LOCALE, locale);
    }

    /**
     * 返回当前action的Locale
     *
     * @return the Locale of the current action
     */
    public Locale getLocale() {
        Locale locale = (Locale) get(LOCALE);
        // 如果当前action的Locale不存在，调用Locale.getDefault()返回
        if (locale == null) {
            locale = Locale.getDefault();
            setLocale(locale);
        }

        return locale;
    }

    /**
     * 设置当前Action的名称
     *
     * @param name
     */
    public void setName(String name) {
        put(ACTION_NAME, name);
    }

    /**
     * 获取当前Action的名称
     *
     * @return name
     */
    public String getName() {
        return (String) get(ACTION_NAME);
    }

    /**
```

```
 * 设置传入当前 Action 的参数，所有的参数被封装在 Map 里
 *
 * @param parameters
 */
public void setParameters(Map<String, Object> parameters) {
    put(PARAMETERS, parameters);
}

/**
 * 返回所有传入当前 Action 的位于 HttpServletRequest 中的参数
 *
 * @return
 */
public Map<String, Object> getParameters() {
    return (Map<String, Object>) get(PARAMETERS);
}

/**
 * 设置以 Map 封装的 HttpSession
 *
 * @param session
 */
public void setSession(Map<String, Object> session) {
    put(SESSION, session);
}

/**
 * 返回以 Map 封装的 HttpSession 的值
 *
 * @return
 */
public Map<String, Object> getSession() {
    return (Map<String, Object>) get(SESSION);
}

/**
 * 设置 OgnlValueStack
 *
 * @param stack the OGNL value stack
 */
public void setValueStack(ValueStack stack) {
    put(VALUE_STACK, stack);
}

/**
 * 返回 OgnlValueStack
 *
 * @return the OGNL value stack
 */
```

```
    public ValueStack getValueStack() {
        return (ValueStack) get(VALUE_STACK);
    }

    /**
     * 设置 Struts2 的配置容器
     *
     * @param cont
     */
    public void setContainer(Container cont) {
        put(CONTAINER, cont);
    }

    /**
     * 返回 Struts2 的配置容器
     *
     * @return The container
     */
    public Container getContainer() {
        return (Container) get(CONTAINER);
    }

    /**
     * 返回位于内部 context 的值，按照 key 值查找
     *
     * @param key
     * @return the value
     */
    public Object get(String key) {
        return context.get(key);
    }

    /**
     * 将键值对存入内部的 context 中
     *
     * @param key
     * @param value
     */
    public void put(String key, Object value) {
        context.put(key, value);
    }
}
```

从源码中，我们可以看到 ActionContext 真正的数据存储空间，是位于其内部的一个 Map 类型的变量 context。ActionContext 将所有的数据对象以特定的键值（Key）存储于 context 之中。而与此同时，ActionContext 也提供了存取这些对象的方式。当然，我们在这里还看到了一些读取框架中比较重要的元素的快捷方法：如 getValueStack，getSession 等等。

8.2.1.2　数据共享空间

ActionContext 的另一大职责是数据共享。一个数据结构要能够做到其内部存储的信息载体的共享，其实并不是一件很容易的事情。因为所谓的数据共享，其本质是数据结构开放，可供外界访问。而一旦外界获得了一个相同存储空间的访问权（这里包含了存储和读取两个相反的操作），在一个多线程的环境中势必造成资源访问的“线程安全”问题。

那么，所谓的“线程安全”问题在 ActionContext 这个数据结构中是否存在呢？这一点，可以运用我们曾经在第 4 章提到过的两个线程安全的判别标准来进行论证：

- **进行数据共享的数据信息是否是类的内部实例变量**
- **外部对共享数据的访问是否存在多线程环境**

这两点，对于 ActionContext 来说全都满足。其一，我们已经在上一节的讲解中通过源码得到证实：ActionContext 中实际的数据存储空间是位于其内部的 context 实例变量；其二，Web 环境本来就是一个多线程环境，进行 Http 请求响应的 Filter 是一种特殊的 Servlet，它自身并不是线程安全的对象。

既然如此，ActionContext 又是如何完成数据共享重任的呢？我们还是试图从源码中来探寻答案，其相关源码如代码清单 8-3 所示。

代码清单 8-3　ActionContext.java

```
public class ActionContext implements Serializable {

    // 此处省略了许多代码

    // 封装了一个 ThreadLocal 变量，存储的内容是 ActionContext 本身
    static ThreadLocal actionContext = new ThreadLocal();

    /**
     * 在 ThreadLocal 中设置 ActionContext，绑定到当前线程
     *
     * @param context
     */
    public static void setContext(ActionContext context) {
        actionContext.set(context);
    }

    /**
     * 返回当前线程中存储的 ActionContext
     *
     * @return
     */
    public static ActionContext getContext() {
        // 返回当前线程中存储的 ActionContext
        return (ActionContext) actionContext.get();
```

```
    }
    // 此处省略了许多代码
    }
```

从上面的源码中，我们可以看到 ActionContext 在内部封装了一个静态 ThreadLocal 的实例，而这一静态的 ThreadLocal 实例所操作和存储的对象，又是 ActionContext 本身。所以 ActionContext 完全满足了 ThreadLocal 模式运行所必需的两大条件，从而保证每一个 ActionContext 的实例都是线程安全的。

使用 ThreadLocal 模式来处理多线程访问问题，是 Struts2 / XWork 框架与其他 Web 框架的一个重大哲学分歧。这当然首先与 Struts2 / XWork 所选择的请求 - 响应模式与传统 Web 框架不同有着必然的联系。一个基于传统 Servlet 模式来实现请求 - 响应的框架，对于多线程访问问题，绝大多数是采用了“绕过去”的方案，也就是避免对核心请求处理元素内部实例变量的访问，尽可能采用参数、返回值等原生的 Java 语法来解决问题。然而，从不同的 Web 框架处理问题的方式来看，“绕过去”还是“勇敢地与之战斗”不仅反映出一个框架处理问题的能力，也从另一个方面反映出框架对待问题的态度。这一点，是非常值得读者进一步思考的。

8.2.1.3 数据存储内容

ActionContext 的两大职责，通过 ActionContext 内部的数据结构体现得淋漓尽致。不过我们在分析中却忽略了 ActionContext 中所存储的内容。数据环境的存储内容，决定于其初始化的时候到底放了什么样的对象进去，不过我们在这里并不打算对 ActionContext 的初始化过程做过多的研究。我们可以换一个研究的角度，从 ActionContext 对外提供的访问接口这一个侧面来说明问题。ActionContext 对外提供的访问接口，如图 8-3 所示。

从图中，我们可以看到两种不同的访问接口类型：

- **对 XWork 框架对象的访问**——getContainer、getValueStack、getActionInvocation
- **对数据对象的访问**——getSession、getApplication、getParameters 等

从这两种不同的访问接口类型可以看出，ActionContext 中所包含的访问接口是包罗万象的。无论是框架对象还是数据对象，只要是整个事件请求处理过程中用得上的对象，我们几乎都可以在 ActionContext 中找到。

在这里，细心一些的读者会发现，ActionContext 所提供的针对数据对象的访问接口，返回的都是 Map 类型的数据对象而并不是真正的 HttpSession 或者 ServletContext 这样的纯正的 Web 容器对象。也就是说，ActionContext 的存储内容虽然包罗万象，但是它对于存放于其中的对象也有着基本的要求：**与 Web 容器无关**。

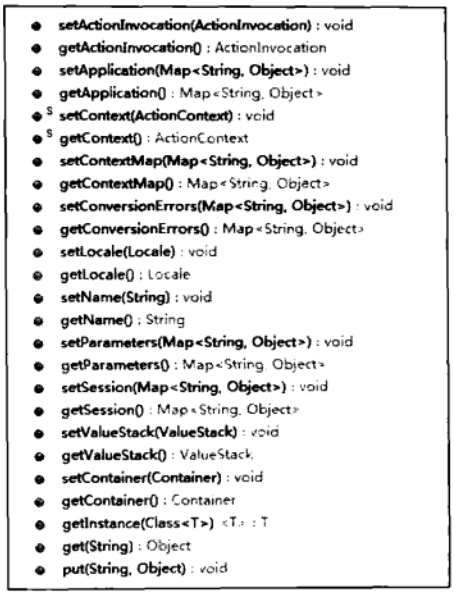


图 8-3　ActionContext 对外提供的访问接口

ActionContext 的这一看似不合理的要求，恰恰反映出 ActionContext 在设计上的严谨之处。我们知道，ActionContext 是 XWork 框架中所定义的元素，而 XWork 框架自身又是一个脱离 Web 容器而单独存在的事件处理框架。因此，一旦 ActionContext 的访问接口中引入了任何 Web 对象，这就势必与 XWork 解耦合的基本设计思想相背离。

在这种情况下，XWork 完成了将原生的 Web 容器对象封装为普通 Java 对象的过程。在表 8-1 中，我们给出了原生 Web 对象与 XWork 封装后对象的对应关系。

表 8-1　Web 原生对象与 XWork 封装后的对象的对应关系

Web 原生对象	XWork 封装对象
HttpServletRequest	RequestMap
HttpSession	SessionMap
ServletContext	ApplicationMap

XWork 的这一封装过程在本质上考虑到了两个重要方面：

❑ **被封装后的 SessionMap 等对象，能够进一步保证数据访问的线程安全性**

我们知道，对 HttpSession、ServletContext 等原生的 Web 容器对象的本身不是线程安全的。而通过 XWork 的封装，我们可以彻底消除这一隐患。

❑ **保持所有存储对象的 Map 结构，可以统一数据访问方式**

我们知道，ActionContext 中的数据信息是要被外部访问的。保持统一的访问方式，可以使各种各样的数据访问工具在访问时都显得游刃有余。

不过，作为一个数据环境，ActionContext 在设计上也考虑到了与 Web 容器对象进行交互的可能性。因而，为了提供一个完整的与 Web 容器打交道的方案，XWork 在 ActionContext 的基础上扩展了一个 ServletActionContext 的子类，并在其中封装了与原生的 Web 容器对象打交道的接口方法，其相关源码如代码清单 8-4 所示。

代码清单 8-4 ServletActionContext.java

```
public class ServletActionContext extends ActionContext implements
StrutsStatics {

    // 此处省略了许多定义

    @SuppressWarnings("unused")
    private ServletActionContext(Map context) {
        super(context);
    }

    /**
     * 根据当前的 HttpServletRequest 获取对应的 ActionContext
     *
     * @param req The request
     * @return The current action context
     */
    public static ActionContext getActionContext(HttpServletRequest req) {
        ValueStack vs = getValueStack(req);
        if (vs != null) {
            return new ActionContext(vs.getContext());
        } else {
            return null;
        }
    }

    /**
     * 根据当前的 HttpServletRequest 获取对应的 ValueStack
     *
     * @param req The request
     * @return The value stack
     */
    public static ValueStack getValueStack(HttpServletRequest req) {
        return (ValueStack) req.getAttribute(STRUTS_VALUESTACK_KEY);
    }

    /**
     * 获取当前的 ActionMapping 对象
```

```
     *
     * @return The action mapping
     */
    public static ActionMapping getActionMapping() {
        return (ActionMapping)
ActionContext.getContext().get(ACTION_MAPPING);
    }

    /**
     * 获取当前请求的 PageContext 对象
     *
     * @return the HTTP page context
     */
    public static PageContext getPageContext() {
        return (PageContext)
ActionContext.getContext().get(PAGE_CONTEXT);
    }

    /**
     * 将当前的 HttpServletRequest 对象存储于 ActionContext 中
     *
     * @param request the HTTP servlet request object
     */
    public static void setRequest(HttpServletRequest request) {
        ActionContext.getContext().put(HTTP_REQUEST, request);
    }

    /**
     * 获取存储于 ActionContext 中线程安全的 HttpServletRequest 对象
     *
     * @return the HTTP servlet request object
     */
    public static HttpServletRequest getRequest() {
        return (HttpServletRequest)
ActionContext.getContext().get(HTTP_REQUEST);
    }

    /**
     * 将当前的 HttpServletResponse 对象存储于 ActionContext 中
     *
     * @param response the HTTP servlet response object
     */
    public static void setResponse(HttpServletResponse response) {
        ActionContext.getContext().put(HTTP_RESPONSE, response);
    }

    /**
     * 获取存储于 ActionContext 中线程安全的 HttpServletRequest 对象
     *
```

```
     * @return the HTTP servlet response object
     */
    public static HttpServletResponse getResponse() {
        return (HttpServletResponse)
ActionContext.getContext().get(HTTP_RESPONSE);
    }

    /**
     * 获取存储于 ActionContext 中的 ServletContext 对象
     *
     * @return the servlet context
     */
    public static ServletContext getServletContext() {
        return (ServletContext)
ActionContext.getContext().get(SERVLET_CONTEXT);
    }

    /**
     * 将 ServletContext 对象存储到 ActionContext 中
     *
     * @param servletContext The servlet context to use
     */
    public static void setServletContext(ServletContext servletContext) {
        ActionContext.getContext().put(SERVLET_CONTEXT, servletContext);
    }
}
```

从上面的源代码中我们可以看到，这些静态方法可以在任何编程层次被随处调用。由于其内部实现是通过操作线程安全的 ActionContext 来完成的，所以我们可以通过调用 ServletActionContext 所暴露的接口来完成对原生 Web 容器 HttpServletRequest、HttpServletResponse 的访问。

8.2.2 ValueStack —— 对 OGNL 的扩展

8.2.2.1 ValueStack 的基本概念

之前已经提到，ValueStack 是 XWork 用以对 OGNL 的计算进行扩展的一个特殊的数据结构。有关 ValueStack 的概念，我们来看看 XWork 的 Reference 是如何解释的：

> *The biggest addition that XWork provides on top of OGNL is the support for the ValueStack. While OGNL operates under the assumption there is only one "root", XWork's ValueStack concept requires there be many "roots".*

从上述解释中，我们可以看到 ValueStack 针对 OGNL 计算的扩展，实际上是**针对 OGNL 三要素中的 Root 对象所进行的扩展**。简单来说，ValueStack 的扩展方式可以分

为两个主要步骤：其一，ValueStack 从数据结构的角度被定义为一组对象的集合；其二，ValueStack 规定在自身这个集合中的所有对象，在进行 OGNL 计算时都被视作是 Root 对象。

根据之前的 OGNL 相关知识，我们知道在默认情况下，OGNL 的计算都是直接针对 Root 对象进行的，所以 ValueStack 这一扩展使得我们在使用 OGNL 表达式进行计算时显得更加便利：

结论　ValueStack 是 XWork 对 OGNL 所做的重要扩展，使得在 Struts2 在使用 ValueStack 进行 OGNL 计算时，可以将一组对象都视作 Root 对象。

从 ValueStack 的名称上，我们就能发现其数据结构的特征：**栈（Stack）结构**。栈（Stack）结构是一个容器结构，之前我们所提到的“一组 Root 对象”，正是存储于 ValueStack 的内部。而同时，ValueStack 自身又具备了对 OGNL 计算进行扩展的能力。因此，ValueStack 从数据结构的角度来说就具备了两面性。接下来，我们将结合源码对 ValueStack 的两面性进行具体的讲解。

8.2.2.2　ValueStack 的数据结构

如果从数据结构的角度来对 ValueStack 进行分析，我们可以非常清晰地看到 ValueStack 的一些基本特点。ValueStack 的部分源码，如代码清单 8-5 所示。

代码清单 8-5　ValueStack.java

```
public interface ValueStack {

    // 这里省略了许多其他代码

    /**
     * 获取 CompoundRoot，这是一个包含了所有栈内元素的容器结构
     *
     * @return the root
     */
    public abstract CompoundRoot getRoot();

    /**
     * 获取栈顶元素，并不改变栈本身
     *
     * @return 栈顶元素
     * @see CompoundRoot#peek()
     */
    public abstract Object peek();

    /**
```

```
     * 获取栈顶元素，并将其从栈中删除
     *
     * @return 栈顶元素
     * @see CompoundRoot#pop()
     */
    public abstract Object pop();

    /**
     * 将对象压入栈，置于栈顶
     *
     * @param o
     * @see CompoundRoot#push(Object)
     */
    public abstract void push(Object o);

    /**
     * 获取栈的大小
     *
     * @return 返回栈的大小
     */
    public abstract int size();
}
```

从源码上来看，ValueStack 的数据结构本身并不复杂，我们可以在任何一本讲述数据结构的教材中找到这种被称为“栈”的数据结构的介绍。而“栈”这种数据结构，至少具备以下两个显著的特点：

- **栈是一个容器，可以存放多个对象**
- **栈是“先进后出”的链表结构**

在这里，我们发现 ValueStack 被定义成了一个接口而非一个实现类，不过我们还是可以从 ValueStack 的接口方法定义中，看到许多“栈”的影子。因为无论是 peek、pop 还是 push，它们都是非常普遍的“栈”的基本操作。那么，ValueStack 在 XWork 中的基本实现，是否能够体现出“栈”这种数据结构的两大特点呢？让我们来看看 XWork 中 ValueStack 的实现类：OgnlValueStack，其部分源码如代码清单 8-6 所示。

代码清单 8-6 OgnlValueStack.java

```
public class OgnlValueStack implements Serializable, ValueStack,
ClearableValueStack, MemberAccessValueStack {

// 使用装饰模式，将根对象（栈结构）封装在 ValueStack 内部
// 从外界来看，所有的操作就像针对单一对象的操作，实际上在内部，实现了对栈的遍历
CompoundRoot root;

// 构造函数，这里将制定 OGNL 计算时所需的参数
```

```
protected OgnlValueStack(ValueStack vs, XWorkConverter
xworkConverter, CompoundRootAccessor accessor, boolean allowStaticAccess) {
        // 调用 setRoot 方法完成初始化
        setRoot(xworkConverter, accessor, new CompoundRoot(vs.getRoot()),
allowStaticAccess);
    }

    // 真正的 OgnlValueStack 的初始化过程
protected void setRoot(XWorkConverter xworkConverter,
CompoundRootAccessor accessor, CompoundRoot compoundRoot, boolean
allowStaticMethodAccess) {
    // 根对象是一个 CompoundRoot 类型的栈结构
        this.root = compoundRoot;
    // 设定 OGNL 所需的 MemeberAccess 实现类
        this.securityMemberAccess =  new
SecurityMemberAccess(allowStaticMethodAccess);
        // 创建 OGNL 的上下文
        this.context = Ognl.createDefaultContext(this.root, accessor, new
OgnlTypeConverterWrapper(xworkConverter), securityMemberAccess);
        context.put(VALUE_STACK, this);
        // 设置 OGNL 上下文的其他相关参数
        Ognl.setClassResolver(context, accessor);
        ((OgnlContext) context).setTraceEvaluations(false);
        ((OgnlContext) context).setKeepLastEvaluation(false);
}


// 此处省略了部分代码
   }
```

在这里，我们可以看到 OgnlValueStack 使用了一个典型的装饰模式，真正在其内部起核心作用的是一个叫做 CompoundRoot 的数据结构。它的数据结构定义如代码清单 8-7 所示。

代码清单 8-7　CompoundRoot.java

```
public class CompoundRoot extends ArrayList {
// 此处省略了部分代码
}
```

原来，CompoundRoot 是一个继承了 ArrayList 的数据结构。而我们知道，ArrayList 刚刚好可以通过稍加改造来实现“栈”结构的两大特性：首先，ArrayList 实现了 List 接口，其天生就是一个容器结构；其次，ArrayList 作为 List 的一个基本实现，天生就是一个链表结构，于是我们只要通过对 ArrayList 中元素的相关操作，就可以实现“先进后出”的效果了。我们在这里先不涉及 CompoundRoot 的相关源码分析，因为这个数据结构在整个

ValueStack 中太重要了，我们不得不独立开辟一节来加以分析。在这里，读者可以首先从数据结构的角度来理解 CompoundRoot 在整个 ValueStack 中所起的作用。

单单通过 ValueStack 的基本数据结构，似乎还不足以说明 ValueStack 对 OGNL 计算的扩展方式。因为至今为止，我们还没有看到任何 ValueStack 的计算功能。所有围绕着 ValueStack 内部构成结构的研究，仅仅体现了 ValueStack 作为 Stack 的职责。那么，有关它 Value 的那一面，又如何体现呢？我们还是需要通过 ValueStack 的接口定义来寻找它的特性。其相关源码如代码清单 8-8 所示。

代码清单 8-8 ValueStack.java

```
public interface ValueStack {

    // 这里省略了许多其他代码

    /**
     * 根据传入的字符串表达式进行求值计算
     * @param expr    传入进行计算的字符串表达式
     * @return
     */
    public abstract Object findValue(String expr);

    /**
     * 根据传入的字符串表达式和对象进行写值计算
     *
     * @param expr  字符串表达式
     * @param value 需要被设置的值
     */
    public abstract void setValue(String expr, Object value);
}
```

从上面的代码中我们可以发现，ValueStack 的定义充分兼顾了 Value 和 Stack 这两个方面。其中，Stack 部分强调了它数据结构的特性，而 Value 部分则体现了其逻辑操作特性。由此，我们可以得到一个针对 ValueStack 的综合结论：

结论 ValueStack 是一个被设计成“栈”的数据结构，并且还是一个具备表达式引擎计算能力的数据结构。

在 XWork 中 ValueStack 的默认实现类是 OgnlValueStack。从字面上我们就能够看出 OgnlValueStack 所表达的含义，其实就是将 OGNL 作为表达式引擎的计算基础引入到“栈”结构中来，并将其打造成为一种综合了两者优点的数据结构。

OGNL 和 Stack 的有机结合，共同构成了 Struts2 / XWork 中表达式引擎的使用基础。

两者结合的重要意义在于，**ValueStack 成为了 Struts2 / XWork 框架进行 OGNL 操作的一个窗口**。换句话说，扮演着数据流转催化剂角色的表达式引擎有了一个代理操作的接口，这个操作接口可以向外界有效屏蔽所有底层实现细节，帮助我们将对 OGNL 原生 API 的操作，转化为对一个特定数据结构（ValueStack）自身的操作。

结论　*ValueStack 是 XWork 框架中的核心元素之一，是 XWork 框架进行 OGNL 计算的实际场所，也是整个框架进行数据访问的基础。*

上述这个结论，是我们对 ValueStack 的一个总评定。如果对这个结论进行断句，其实可以从中读到两层意思：

- **ValueStack 是 XWork 框架进行 OGNL 计算的场所**
- **ValueStack 是 XWork 框架进行数据访问的基础**

这其中的第一层意思表明在 XWork 框架的范围内，如果要针对某一对象进行表达式引擎的计算，我们只要将这个对象置于 ValueStack 之中，通过 ValueStack 的计算接口进行操作即可。这一个相对直观的结论是我们站在 ValueStack 的角度所得出的。在之后章节中，我们还会看到这个结论在整个 XWork 中所起的重要作用，因为它实际上也是数据流和控制流进行交汇的理论基础。

而第二层意思，则是站在第一层意思的基础之上，表明在 XWork 中，如果要通过表达式引擎对数据进行访问，ValueStack 将是我们的一个重要操作接口。这一层意思在目前阶段理解起来还需要花费一番工夫。我们会在数据流和控制流的交互体系中对这一结论进行详细的阐述。

8.2.2.3　ValueStack 对 OGNL 计算规则的影响

在对 ValueStack 基本概念的描述中，我们知道 ValueStack 对 OGNL 计算的扩展主要是针对 OGNL 三要素中的 Root 对象所进行的扩展。如果对这一扩展进行逻辑归纳，可以得到如下结论：

结论　*ValueStack 对 OGNL 的计算对象进行了“无差别化”处理。*

所谓的“无差别化”处理，其实就是指在 ValueStack 中所有的对象都是 OGNL 计算的 Root 对象，它们之间是一个等同的关系。

然而，在研究了 ValueStack 的数据结构后，我们会发现这种“无差别化”处理实际上并不是一个绝对的无任何差别的对待。因为从本质上讲，ValueStack 实现“无差别化”处理的数据结构基础在于其内部实现的一个链表结构。有了这个链表结构，整个 OGNL 的计算规则就发生了改变。我们可以通过一个具体的实例进行说明，如图 8-4 所示。

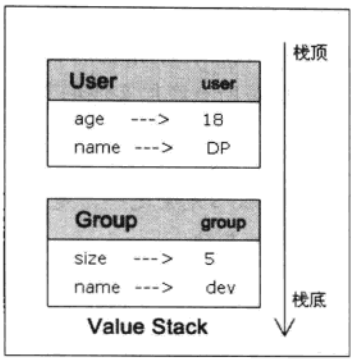


图 8-4 ValueStack 示意图

在上述示意图中，如果针对 ValueStack 进行 OGNL 表达式的计算，我们会得到如下结果：

```
// 调用 user.getAge() 方法
age      //-> 18
// 调用 group.getSize() 方法
size     //-> 5
// 从栈顶开始查找，user 位于栈顶部，所以 user.getName() 方法被调用
name     //-> DP
```

从上述计算规则和计算结果中，我们可以看到 ValueStack 对 OGNL 计算的影响：任何位于 ValueStack 中的对象在计算时一概被视作 OGNL 的 Root 对象（无论是位于 ValueStack 上层的 user 对象，还是下层的 group 对象，它们都被视作是 Root 对象）。OGNL 在进行表达式计算时的基本逻辑是：**从栈的顶端开始对每个栈内元素进行表达式匹配计算，并返回第一个成功匹配的表达式计算结果**。

以 age 为例，OGNL 在对 age 进行表达式计算时，首先找到栈的顶端元素 user，将 user 作为 Root 对象进行表达式求值，user.getAge() 方法会被成功调用。针对栈的顶端元素的表达式求值成功，于是 OGNL 计算过程中止，将 user.getAge() 方法的调用结果返回作为 age 这个表达式的计算结果。

再以 size 为例，OGNL 在对 size 进行表达式计算时，还是首先找到栈的顶端元素 user，将 user 作为 Root 对象进行表达式求值。然而此时，user 对象中并不存在 user.getSize() 方法，因而针对栈的顶端元素的表达式求值失败。此时，OGNL 计算过程并不会中止，而是继续寻找 ValueStack 中的下一个元素 group，重复表达式求值的计算过程，直到返回第一个成功匹配的表达式计算结果。

结论 ValueStack 支持多个“根对象”操作的基本流程是自上而下遍历其中的元素逐一进行表达式求值计算，返回第一个成功匹配表达式的计算结果。

因此，ValueStack 支持多个 Root 对象的 OGNL 操作的本质是**栈内元素遍历**。而整个元素遍历的顺序则反映出 ValueStack 在进行 OGNL 计算时的层次感，这与我们之前所强调的“无差别化”似乎有所出入。不过在绝大多数情况下，我们其实并不关心 ValueStack 中元素的顺序，因为 OGNL 表达式在 ValueStack 中能够返回多个值的可能性并不大，我们却能从这一扩展中获得极大的操作便利性。

8.2.2.4　栈顶元素和子栈

在上一节中，我们强调了 ValueStack 对 OGNL 的计算规则：**栈内元素遍历**。栈内元素遍历的结果，是返回第一个满足 OGNL 表达式运算结果的值。事实上，栈内元素遍历是 ValueStack 在设计时，为了应对 OGNL 计算时 Root 对象的“无差别化”而采取的一种**计算策略**。

然而 ValueStack 在进行计算时，**除了“无差别化”以外，还提供了一些“有差别化”的 OGNL 计算方式**。这种“有差别化”的计算方式，实际上就是针对特定栈内元素的 OGNL 计算。

“有差别化”的 OGNL 计算，从逻辑上讲很容易实现。因为我们知道在 ValueStack 的内部所维护的栈结构实际上只是一个 ArrayList。既然是 ArrayList，**我们就可以通过类似于数组下标的形式对栈结构中的每一个元素进行访问**。我们不妨举个 ValueStack 的例子来进行说明，其相关示意图如图 8-5 所示。

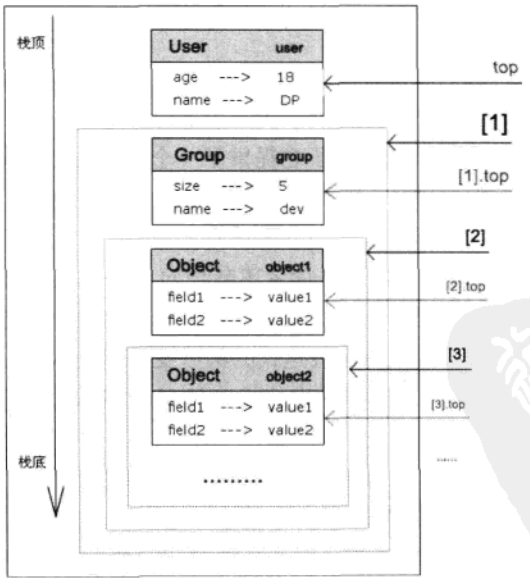


图 8-5　ValueStack 的示意图

如果把 ValueStack 看作一个自上而下的链表（ArrayList）结构，那么我们很容易就能够得出对每一个元素的访问方式：通过数组下标进行访问。

```
// 栈中的第一个元素，以数组下标 0 进行访问，其对应的是 user 对象
[0]     //-> user 对象
// 栈中的第二个元素，以数组下标 1 进行访问，并可继续套用 OGNL 表达式进行计算
[1].name     //-> Dev
```

这就是 ValueStack 中对于“有差别化”OGNL 计算的访问方式，简单而有效。读者可以发现，一旦理解了 ValueStack 内部的数据结构，这种表达式访问的逻辑是自然而然形成的。

ValueStack 的这种“有差别化”OGNL 计算的访问方式，进一步引申出了 ValueStack 中的两个非常重要的概念：栈顶元素（top）和子栈（Substack）。

所谓栈顶元素，就是**指位于 ValueStack 顶端的那个元素**。换句话说，也就是那个可以通过 [0] 进行访问的元素，只不过在这里，ValueStack 为 [0] 这个元素定义了一个特殊的关键字访问方式：top。由于 ValueStack 是一个栈结构，因而从栈结构的操作特性上讲，栈顶元素也就是最后一个被压入 ValueStack 中的元素。

还是来看上面的例子，我们可以以 top 关键字来代替一切使用 [0] 进行对象访问的地方，因而表达式计算看起来可以这样：

```
// 返回栈顶元素
top      //-> user 对象
// top 是 user 对象，所以 top.name 等同于 [0].name，也等同于 user.name
top.name     //-> DP
```

引入 top 关键字的一个重要意义在于它提供了一个具有特定含义的名称作为访问表达式。相对于数组下标的访问方式，top 关键字提供了良好的可读性，可以帮助广大程序员组合出各种形式丰富而可读性强的 OGNL 表达式。

既然 [0] 被 top 关键字取代了，那么读者不禁要问，类似 [1] 或者 [2] 这样的表达式是否有类似的替代方案呢？答案是否定的。因为 ValueStack 无意引入过多的关键字，否则不仅会带来昂贵的学习成本，也会使整个表达式的构造变得复杂不堪。

虽然如此，类似 [1] 或者 [2] 这样的表达式却被 ValueStack 赋予了另外一个重要的扩展概念：子栈。

在 ValueStack 中，使用 [1] ～ [n] 的数组下标序列来定义蕴含在 ValueStack 中的所有子栈。例如，**[n] 表示的 ValueStack 是指去除栈结构中前 N 个元素之后所构成的元素集合**。

子栈的官方定义看上去非常复杂。读者在理解的时候完全可以不必理会这拗口的定义。因为子栈本质也不过就是通过数组下标来访问栈内的特定元素而已。不过既然被称之为子栈，我们还是为其总结一些基本特性：

- 一个大小为 N 的 ValueStack，除了自身外，有 N – 1 个子栈。
- 每一个子栈自身也是一个 ValueStack，相对于 ValueStack 形成了一个递归的数据结构。

如果 ValueStack 的目的是将 OGNL 的计算趋同化，我们可以看到，子栈存在的目的则是明确区分 ValueStack 中的所有元素，从而使得 OGNL 的计算趋于异化。子栈对 ValueStack 中元素的区分，比栈顶元素更为具体。从子栈的定义中，我们也可以充分感受到“栈”的数据结构特性，因为无论从定义还是访问方式来看，子栈都沿用了“栈”的可“序列”特性。

如果将子栈和栈顶元素的概念整合起来，我们可以构造出更加灵活多样的 OGNL 表达式：

```
// 返回位于栈顶的那个数据元素
top     //-> user 对象
// 调用 user.getName() 方法
top.name     //-> downpour
// 调用 user.getAge() 方法
top.age     //-> 18
// 抛去 user 对象之后的子栈，group 对象成为子栈的栈顶元素
[1].name     //-> Dev
// 抛去 user 对象之后的子栈，group 对象成为子栈的栈顶元素
[1].top       //-> group 对象
```

8.2.3　深入 ValueStack 的实现

8.2.3.1　CompoundRoot 详解

之前我们提到，CompoundRoot 对象被维护在 OgnlValueStack 的内部，是 ValueStack 对 OGNL 计算进行扩展的核心数据结构。

我们知道，OGNL 有三要素：OGNL 表达式、Root 对象和上下文环境。而 CompoundRoot，实际上就是 ValueStack 中进行 OGNL 计算的“Root 对象”，是真正承担着实际意义上“栈”的职责的数据结构。从之前对 OgnlValueStack 源码的分析中，我们可以发现 CompoundRoot 被封装在 OgnlValueStack 的内部，成为其最重要的一个成员变量。而这正是一个典型的装饰模式的运用，只不过在这里运用得非常含蓄。若顺着思路去查看 CompoundRoot 的具体实现，会发现它居然只是一个 ArrayList！其源码如代码清单 8-9 所示。

代码清单 8-9　CompoundRoot.java

```
public class CompoundRoot extends ArrayList {
```

```
    public CompoundRoot() {
    }

    public CompoundRoot(List list) {
        super(list);
    }

    // 分割 Substack 的操作源头
    public CompoundRoot cutStack(int index) {
        return new CompoundRoot(subList(index, size()));
    }

    // 返回栈顶元素
    public Object peek() {
        return get(0);
    }

    public Object pop() {
        return remove(0);
    }

    public void push(Object o) {
        add(0, o);
    }
}
```

从源码中我们发现，CompoundRoot 只是一个非常简单的 ArrayList 的扩展而已，其增加的只是栈的基本操作：peek、pop、push 等。而这些栈的基本操作，恰恰是运用了 ArrayList 这个数据结构的自然属性完成的。

在这里，我们要特别注意在 CompoundRoot 中的 cutStack 方法，这个方法会返回一个新的 CompoundRoot 对象，实际上它对应的就是我们在上一节中所提到过的一个重要概念：子栈。也就是说，当我们需要通过 OGNL 表达式对子栈进行数据访问时，CompoundRoot 的操作是首先调用 cutStack 方法返回一个子栈的结构，再进行链式递归调用。这一递归调用进一步反映出 ValueStack 内部所蕴含的精妙之处。

8.2.3.2 CompoundRootAccessor 详解

OgnlValueStack 是通过 OgnlValueStackFactory 创建出来的，在 OgnlValueStackFactory 对 ValueStack 进行初始化的过程中，会为 OGNL 的计算指定许多默认的实现方式。其相关源码如代码清单 8-10 所示。

代码清单 8-10 OgnlValueStackFactory.java

```
public class OgnlValueStackFactory implements ValueStackFactory {
```

```
    private XWorkConverter xworkConverter;
    private CompoundRootAccessor compoundRootAccessor;
    private TextProvider textProvider;
    private Container container;
    private boolean allowStaticMethodAccess;

    @Inject(value="allowStaticMethodAccess", required=false)
    public void setAllowStaticMethodAccess(String
allowStaticMethodAccess) {
        this.allowStaticMethodAccess =
"true".equalsIgnoreCase(allowStaticMethodAccess);
    }

public ValueStack createValueStack(ValueStack stack) {
    // 初始化 OgnlValueStack，设定 OGNL 计算时需要的参数等
        ValueStack result = new OgnlValueStack(stack, xworkConverter,
compoundRootAccessor, allowStaticMethodAccess);
    // 对 OgnlValueStack 进行参数注入
        container.inject(result);
        stack.getContext().put(ActionContext.CONTAINER, container);
        return result;
    }

// 初始化 Container，同时初始化 OGNL 的相关设置
// 这里主要是初始化 OgnlRuntime 中的三大规则
// PropertyAcessor、MethodAccessor、NullHandler
    @Inject
public void setContainer(Container container) throws
ClassNotFoundException {
    // 从 XWork 的 container 中获取所有 ognl.PropertyAccessor 的实现类
        Set<String> names = container.getInstanceNames(PropertyAccessor.class);
        if (names != null) {
            for (String name : names) {
                Class cls = Class.forName(name);
                if (cls != null) {
                    if (Map.class.isAssignableFrom(cls)) {
                        PropertyAccessor acc =
container.getInstance(PropertyAccessor.class, name);
                    }
                // 根据不同的类名，为 OGNL 分配对应的 PropertyAccessor 实现
                    OgnlRuntime.setPropertyAccessor(cls,
container.getInstance(PropertyAccessor.class, name));
                    // 找到 CompoundRootAccessor 的实现，并初始化
                    if (compoundRootAccessor == null &&
CompoundRoot.class.isAssignableFrom(cls)) {
                        compoundRootAccessor = (CompoundRootAccessor)
container.getInstance(PropertyAccessor.class, name);
                    }
                }
```

```
            }
        }
        // 从 XWork 的 container 中获取所有 ognl.MethodAccessor 的实现类
        names = container.getInstanceNames(MethodAccessor.class);
        if (names != null) {
            for (String name : names) {
                Class cls = Class.forName(name);
                if (cls != null) {
                    // 根据不同的类名，为 OGNL 分配对应的 MethodAccessor 实现
                    OgnlRuntime.setMethodAccessor(cls,
container.getInstance(MethodAccessor.class, name));
                }
            }
        }
        // 从 XWork 的 container 中获取所有 ognl.NullHandler 的实现类
        names = container.getInstanceNames(NullHandler.class);
        if (names != null) {
            for (String name : names) {
                Class cls = Class.forName(name);
                if (cls != null) {
                    // 根据不同的类名，为 OGNL 分配对应的 NullHandler 实现
                    OgnlRuntime.setNullHandler(cls, new
OgnlNullHandlerWrapper(container.getInstance(NullHandler.class,
name)));
                }
            }
        }
        if (compoundRootAccessor == null) {
            throw new IllegalStateException("Couldn't find the compound root
 accessor");
        }
        // 初始化 container
        this.container = container;
    }

    // 此处省略部分代码

}
```

看到这里，对 ValueStack 内部到底是如何扩展 OGNL 计算的问题就迎刃而解了。原来，所有的初始化工作都在 OgnlValueStackFactory 中有条不紊地完成。由于所有 OGNL 的运行参数都属于 Struts2 / XWork 需要定义的内置变量，因而 Struts2 / XWork 选择了在 OgnlValueStackFactory 实施依赖注入时（也就是在 setContainer 方法中）完成这些参数的实现方式的选择和设置。

有了 OGNL 运行所需要的所有参数，我们就可以在 Struts2 中以 ValueStack 作为一个重要的中转站进行 OGNL 操作。而 OGNL 操作本身，不过是调用 OgnlUtil 完成对 OGNL

的原始 API 的调用而已。由于 OGNL 运算需要的所有参数都已经准备好，所以这些计算反而变得非常简单了。

在这其中，有一个叫做 CompoundRootAccessor 的类被反复使用了多次。它不仅用来指定为 ClassResolver 的默认实现，同时还被指定为 PropertyAccessor 和 MethodAccessor 的默认实现。

我们在讲解 OGNL 时曾经讲过，ClassResolver、PropertyAccessor 和 MethodAccessor 都属于 OGNL 计算时所需要指定的默认行为规则，因而 Struts2 / XWork 在进行 OGNL 计算时，分别为它们指定了默认的实现方式，它们都通过 CompoundRootAccessor 来实现，源码如代码清单 8-11 所示。

代码清单 8-11　CompoundRootAccessor.java

```
// 分别实现了 ClassResolver 接口、PropertyAccessor 接口和 MethodAccessor 接口
public class CompoundRootAccessor implements PropertyAccessor,
MethodAccessor, ClassResolver {

    // 实现 OGNL 的 PropertyAccessor 中的 setProperty 方法
    public void setProperty(Map context, Object target, Object name, Object
value) throws OgnlException {
        // 类型转化，得到 OGNL 的根对象和上下文
        CompoundRoot root = (CompoundRoot) target;
        OgnlContext ognlContext = (OgnlContext) context;

        // 循环查找 CompoundRoot 堆栈中的所有元素
        for (Object o : root) {
            if (o == null) {
                continue;
            }

            try {
                // 调用 OgnlRuntime 的方法判定当前元素是否能够被写值
                if (OgnlRuntime.hasSetProperty(ognlContext, o, name)) {
                    OgnlRuntime.setProperty(ognlContext, o, name, value);
              // 设值完成，停止查找下一个元素，直接返回
              return;
                } else if (o instanceof Map) {
                    // 如果是一个 Map 对象，直接往 Map 对象中写值即可
                    Map<Object, Object> map = (Map) o;
                    try {
                        map.put(name, value);
                        return;
                    } catch (UnsupportedOperationException e) {
                    }
                }
            } catch (IntrospectionException e) {
```

```
            }
        }

        Boolean reportError = (Boolean)
context.get(ValueStack.REPORT_ERRORS_ON_NO_PROP);

        final String msg = "No object in the CompoundRoot has a publicly
accessible property named '" + name + "' (no setter could be found).";

        // 如果在OGNL计算中发生异常，OGNL上下文中会决定是否抛出这个异常
        if ((reportError != null) && (reportError.booleanValue())) {
            throw new XWorkException(msg);
        } else {
            if (devMode) {
                LOG.warn(msg);
            }
        }
    }

    // 实现OGNL的PropertyAccessor中的getProperty方法
public Object getProperty(Map context, Object target, Object name)
throws OgnlException {
        // 类型转化，得到OGNL的根对象和上下文
        CompoundRoot root = (CompoundRoot) target;
        OgnlContext ognlContext = (OgnlContext) context;

        // 如果访问表达式是数字，表示获取ValueStack的一个子栈
        if (name instanceof Integer) {
            Integer index = (Integer) name;

            return root.cutStack(index.intValue());
        } else if (name instanceof String) {
            // 处理关键字top，返回的是ValueStack的栈顶元素
            if ("top".equals(name)) {
                if (root.size() > 0) {
                    return root.get(0);
                } else {
                    return null;
                }
            }

            // 对于普通的属性名称，需要循环整个栈，找到第一个匹配的元素并返回
            for (Object o : root) {
                if (o == null) {
                    continue;
                }

                try {
```

```
                    // 调用 OgnlRuntime 的方法判定是否包含该属性
                    if ((OgnlRuntime.hasGetProperty(ognlContext, o, name))
|| ((o instanceof Map) && ((Map) o).containsKey(name))) {
                        // 找到属性后，停止查找，并直接返回属性值作为结果
                        return OgnlRuntime.getProperty(ognlContext, o, name);
                    }
                } catch (OgnlException e) {
                    if (e.getReason() != null) {
                        final String msg = "Caught an Ognl exception while
getting property " + name;
                        throw new XWorkException(msg, e);
                    }
                } catch (IntrospectionException e) {
                }
            }

            return null;
        } else {
            return null;
        }
    }

    // 此处省略了许多其他的代码
}
```

在上面的代码中，我们看到 ValueStack 支持多个 OGNL 操作“Root 对象”的入口方法的逻辑：XWork 无论对于属性访问，还是方法访问都做了重新的过程定义。也就是说，在 ValueStack 进行 OGNL 计算时，都会循环扫描 CompoundRoot 中的所有元素，并找到第一个符合表达式定义的元素进行计算，然后返回结果。这也是我们从源码级别对“栈内元素遍历”的结论进行的论证。

不仅如此，我们还可以在 CompoundRootAccessor 的源码中看到 ValueStack 对于特殊的 OGNL 表达式的处理。例如，对于 top 关键字或者访问表达式是一个数字的情况（访问子栈）。所有 OGNL 的计算规则都覆盖在这个类中，从而形成了 ValueStack 特有的访问逻辑。

8.2.4　形影不离、相互依存的 Actioncontext 与 ValueStack

ActionContext 和 ValueStack 都是数据流体系中的重要元素。我们可以用一个四字成语来描述这两大元素之间的关系：**相互依存**。这也就是我们在本章一开始所谈到的那一层职责上的依存关系。

单从职责上分析，或许还不能足以使人信服。接下来，我们就使用源码来对它们之间相互依存的关系进行说明，并为它们的关系打上一个更加密切的标签，那就是：**形影不离**。

为什么要使用“形影不离”来形容 ActionContext 和 ValueStack 之间的关系呢？这就需要从 ActionContext 的创建过程谈起。ActionContext 的创建过程如代码清单 8-12 所示。

代码清单 8-12 PrepareOperations.java

```
    /**
     * 创建 ActionContext，并初始化到当前线程
     */
    public ActionContext createActionContext(HttpServletRequest request,
    HttpServletResponse response) {
    ActionContext ctx;
    // 为当前线程的 ActionContext 进行计数，为之后 cleanup 做准备
        Integer counter = 1;
        Integer oldCounter = (Integer) request.getAttribute(CLEANUP_RECURSION_
COUNTER);
        if (oldCounter != null) {
            counter = oldCounter + 1;
        }
        // 获取当前线程中的 ActionContext
        ActionContext oldContext = ActionContext.getContext();
        if (oldContext != null) {
            // 当前线程中存在 ActionContext，复制为新的 ActionContext
            ctx = new ActionContext(new HashMap<String,
    Object>(oldContext.getContextMap()));
    } else {
        // 创建 ValueStack
            ValueStack stack =
    dispatcher.getContainer().getInstance(ValueStackFactory.class).
    createValueStack();
            // 设置 ValueStack 的数据环境，注意这里的数据环境通过 dispatcher 的转换，
            // 实际上已经成为普通的 Java 对象，这就消除了 ValueStack 对容器的依赖
            stack.getContext().putAll(dispatcher.createContextMap(request,
    response, null, servletContext));
            // 将 ValueStack 的数据环境与 ActionContext 等同起来
            ctx = new ActionContext(stack.getContext());
        }
    request.setAttribute(CLEANUP_RECURSION_COUNTER, counter);
    // 设置 ActionContext 到当前线程
        ActionContext.setContext(ctx);
        return ctx;
    }
```

这是一段位于 Struts2 第二条运行主线中 Http 请求预处理阶段的代码片段，其核心逻辑是根据 Web 容器对象构建出 ActionContext。我们可以看到使用了 HttpServletRequest 和 HttpServletResponse 这样的 Web 对象作为参数。

不过在这里，我们想要强调在源码中蕴含的两个重要结论：

结论　ActionContext 的创建，总是伴随着 ValueStack 的创建。

这一点在源码中我们可以看得非常清楚。ValueStackFactory 负责创建 ValueStack，并为 ValueStack 设置上下文环境；紧接着 ValueStack 创建的就是 ActionContext，并且 ActionContext 的创建以 ValueStack 的上下文环境作为参数。可见，ValueStack 的构建是 ActionContext 构建的基础，两者总是在几乎相同的时刻被同时创建出来。

结论　ValueStack 的上下文环境与 ActionContext 的数据存储空间是等同的。

这一结论蕴含在 new ActionContext(stack.getContext()) 这样一句极为简单的 ActionContext 构造函数的调用之中，然而这一结论对我们却拥有举足轻重的意义。

我们知道，ValueStack 是对 OGNL 表达式引擎的扩展，并提供了整个 XWork 进行 OGNL 计算的空间。而对于 OGNL 来说，最重要的无疑是 OGNL 三要素：表达式、Root 对象和上下文环境。一旦 ValueStack 的上下文环境与 ActionContext 的数据存储空间等同起来，就等于为 ValueStack 进行数据计算打通了最重要的一个环节：数据基础。

当然，除了上下文环境之外，ValueStack 进行 OGNL 计算的另外一个环节——Root 对象也非常重要。只不过 Root 对象的设置和引用，已经涉及 ValueStack 进行表达式引擎计算的具体过程。有关这一点，我们将在控制流和数据流的交互体系中为读者解开其中的秘密。

在这里，我们不仅进一步证明了 ActionContext 与 ValueStack 之间相互依存的关系，还揭示了 ValueStack 进行数据计算的基础之一的上下文环境的来龙去脉。读者在这里可以稍作停顿，重新思考一下数据流中每个元素的职责和它们具体实现职责的方式，从而为我们继续控制流体系的介绍打下基础。

8.3　控制流体系 —— 有条不紊

8.3.1　Action —— 革命性突破

作为控制流元素中的核心，Action 在整个控制流体系中拥有至高无上的地位。我们将其列为控制流体系的第一个元素来讲解。

8.3.1.1　Action 的定义

我们首先来看看作为 XWork 控制流体系核心元素的 Action 的定义，其相关代码如代码清单 8-13 所示。

代码清单 8-13 Action.java

```
/**
 * 所有进行请求处理的核心处理类都将实现此接口，暴露 execute() 方法作为默认处理方法
 */
public interface Action {
/**
 * 业务逻辑执行的场所
 *
 * @return
 *             一个表示 Action 执行完毕之后逻辑后续走向的标志字符串
 * @throws Exception
 *              系统级别的异常将被抛出，应用程序级别的异常应由程序自行处理并
 *              返回类似 Action.ERROR 进行系统转向
 */
 public String execute() throws Exception;
}
```

XWork 将进行事件处理的核心类设计为一个接口。具体针对每一个请求的处理，则通过该接口的类来完成。Action 接口看上去非常简单，它仅仅定义了一个没有参数的方法。不过当我们从语法的角度来对这个接口进行分析，就会发现这个 Action 接口的定义提供了一些额外的信息：

❑ 方法主体

Action 接口的具体实现类是 XWork 进行核心业务逻辑处理的场所

❑ 运行参数

接口方法中并不包含参数，表明 Action 执行时所需要的数据，无论是请求参数还是响应返回数据，**都以 Action 中属性变量的形式出现**。这就使得 Action 作为一个对象，看上去更像是一个 POJO，这个 POJO 不仅包含了属性（封装了请求和返回的数据），同时也具备了方法（进行逻辑处理的行为动作）。

❑ 返回值

接口方法中的返回值是一个字符串形式，很显然它只是一个表示业务逻辑执行是否成功的标志，在 Action 中起到流程控制的标志作用。

从上述三个语法特征的分析中，我们很容易看出 XWork 对于请求 - 响应模式的实现采用了上一章中曾经提到过的 **POJO 模式**。而 POJO 模式相对于 Servlet 模式（其他两种实现模式其实都是 Servlet 模式的有效变种）在实现上有着重大的分歧。那么，这些分歧究竟是不是 Web 开发中的核心分歧？不同的实现模式，究竟孰优孰劣呢？我们又如何来理解这些分歧？接下来，我们将对 Action 定义中的各个组成要件进行构成分析。

8.3.1.2　孰优孰劣

要回答孰优孰劣这个问题，首先要讲清楚 POJO 模式和 Servlet 模式分歧的核心究竟在哪里。这个问题的答案在上一章中曾经提到过，只不过当时并没有把它作为一个核心结论提炼出来。我们在这里可以回顾一下：

结论　*在针对请求－响应的实现中，Servlet 模式和 POJO 模式分歧的核心就在于具体负责请求处理的核心响应类是否是一个有状态的对象。*

Servlet 对象是一个无状态对象，而 POJO 却以描述对象状态为目的。那么，对于 Http 请求的处理而言，究竟是无状态对象好，还是有状态对象好呢？这又是一个耐人寻味的哲学选择。我们可以就当下最为流行的两大 Web 框架 SpringMVC 和 Struts2 进行一番比较。因为它们两者在实现模式上恰恰做出了两种截然不同的选择：SpringMVC 坚守了 Servlet 模式（实际上 SpringMVC 所采用的是一个游离于参数 - 参数模式和参数 - 返回值模式之间的，整合了两者各自优势的实现模式），而 Struts2 则秉承着 Webwork 以来一直遵循的 POJO 模式。

以下，我们分别站在请求数据、响应数据以及响应逻辑跳转这三个不同的角度来看看 SpringMVC 和 Struts2 分别采取了什么样的编程元素表现形式：

❑ 请求数据——参数，还是属性变量？

我们曾经在第 2 章中谈到过方法的元素构成。当时我们对方法中参数的逻辑定义就是“请求参数”，因为这是一种对方法构成元素的**最为自然而有效的逻辑定义**。

SpringMVC 对于请求数据的处理，就没有破坏这种天然的语法自身所带来的逻辑语义划分；而 Struts2 则从另外一个角度来对响应类进行定义：无论是请求数据还是响应数据，都应该成为构成响应类自身状态的构成元素。因此，从描述类自身状态来看，属性变量似乎是一个不错的选择。

❑ 响应数据——参数、返回值，还是属性变量？

在请求处理完毕之后，将请求的处理结果返回给请求的发起者是整个请求－响应过程必不可少的环节之一。然而这却是不同实现模式分歧最大的一项。正如我们在上一章中曾经谈到的，不同的请求－响应实现模式在这个问题上的思考角度是不同的。

参数－返回值模式使用返回值来表达响应数据，因为它可以将响应数据发送给请求端的工作继续交给其上层的处理对象来完成。而参数－参数模式作为一个基础的 Servlet 模式，不得不通过一个参数获得对响应数据的处理句柄，从而在响应方法的内部自行完成将响应数据发送给请求端的工作。

❑ 响应逻辑跳转——返回值还是参数？

在请求处理完毕之后，我们不得不为程序的继续运行指明方向，这就是所谓的“响应

逻辑跳转”工作。事实上，响应逻辑跳转和响应数据的处理，分别是从控制流和数据流这两个不同的方面对请求－响应模式中的响应过程进行的诠释。不过我们发现，正是数据和控制这两种截然不同的逻辑分工，使得响应方法在表达形式上显得捉襟见肘。我们知道，作为最合适的表达元素：返回值，它很难同时表达两个不同的逻辑过程，如果“合二为一”，势必造成两种不同的逻辑处理的耦合。如果使用参数，并在响应方法内部自行完成逻辑跳转的功能，则直接造成核心逻辑处理和非核心逻辑处理的耦合。

综合上述这三个角度的分析，这里可以简单和大家探讨一下笔者的个人观点。在这里，我们并不想引起框架之争，仅仅是就事论事地对于不同的请求－响应实现模式进行讨论。读者对于这类哲学问题，应该博采众长，细细品味各种框架解决方案的核心思想，并形成自己的观点。这才是正确的学习之道。

观点 *对于请求数据，参数比属性变量更为直观，也更符合 Java 原生的语法含义。*

这一点上，SpringMVC 无疑略胜 Struts2 一筹。回顾一下我们曾经在第 2 章讲面向对象的概念时所提到的方法的构成体系，会发现从方法自身所表达的逻辑语义来看，参数恰好是一个逻辑输入的表达载体，也完全符合 Java 原生语法对于方法的逻辑定义。另外，从程序开发的角度，带有参数的响应方法可以从语义上表达更为确切的操作含义，从而使代码具备更好的可读性。

观点 *对于响应数据，最适合的表达方式是方法的返回值。*

在这一点上，两个框架都没有完全进行这样的选择。不过正如我们之前的分析，Web 框架在响应的处理上，**必须包含响应数据和响应流程控制这两个不同的方面**。而框架的困境就在于**单一的元素（或者是返回值，或者是参数），很难同时表达数据和控制这两个不同的层面**。

在这种情况下，SpringMVC 采取了两种不同的应对策略。其一，将响应数据和响应流程控制这两个方面合成一个对象：ModelAndView。从这个对象的命名上，我们就可以看出它不仅在内部存储了响应的数据，同时具备了处理响应流程控制（视图的跳转）的功能。其二，将这两个不同的逻辑处理方面区分开，使用返回值来表达响应流程控制，同时为响应方法增加一个参数：Model 对象，用以存储响应数据。这种做法相当于结合了参数-返回值和参数-参数这两种实现模式各自的特性，将原本就归属于两个不同职责的逻辑片段分派到两个不同的编程元素上去。当然，无论是哪一种应对策略，都是 SpringMVC 退而求其次的做法，属于不得以而为之。

Struts2 / XWork 的应对策略则比较简单。因为 Struts2 使用 POJO 模式来实现请求－

响应的过程，因而无论是请求数据还是响应数据，都可以作为响应对象的内部状态的一部分存在于对象的内部属性之中。在这种情况下，响应方法的返回值就可以用于响应流程的控制。因而，Struts2/XWork 的简单之处就是它在请求－响应实现模式上的天然优势。不过，如果站在 Java 原生语法的角度，抛开请求－响应模式而单单看方法，我们就会发现在这个时候方法体中缺失了“处理结果”这一重要的构成要素。

观点　使用有状态的 POJO 对象来进行请求响应，在数据访问上具有天然的优势。

SpringMVC 并非不知道这一点，但是限于设计理念和实现模式选择而无法支持。相反，这对于 Struts2 / XWork 而言却是一个天然的支持的特性。我们知道，任何编程元素都有其作用域，无论是参数还是返回值，它们的作用域限于方法定义本身。因此，对于参数和返回值的引用，在方法的内部是没有任何问题的，然而要在方法之外来访问这些元素就比较困难了。在现代编程中，我们可以使用 AOP（面向方面的编程）的方式对方法进行拦截，从而获得方法体定义中的所有元素。不过这与 JavaBean 这样一个耳熟能详的数据访问标准相比，毕竟要复杂得多。

所以，翻开 SpringMVC 中的源码就会发现，其中定义的 Interceptor 接口虽然提供了对控制层的拦截，但是其应用并不广泛。而 Struts2 中的 Interceptor 却是整个 Struts2 进行请求处理的核心。这与不同框架所选择的实现模式有着必然的联系。

8.3.1.3　状态与动作的合体

把目光回到 Struts2 / XWork 的 Action 本身，本节的标题是“革命性突破”，这是对 Action 的一个综合评价。通过之前的分析，我们知道在这其中突破点主要有两个：

- **突破了传统 Servlet 模式中相应对象对 Web 容器的依赖**
- **突破了传统 Servlet 模式中响应对象无状态的限制**

第一个突破点，我们从 Action 的定义本身就可以看得出来。任何响应对象，只要实现了简单的 Action 接口，它就能成为一个 Http 请求的响应类，我们在响应类中将看不到任何对 Web 容器对象的引用。

而这其中的第二个突破点，就值得我们细细品味一番。在 XWork 框架中的 Action 是一个**有状态的**响应处理类。我们在第 2 章描述面向对象的基本概念时曾经探讨过，**一个类的状态和内在特性主要是通过类的属性变量加以描述的**。在这种情况下的响应动作，则是以对象的内部状态作为一种数据依赖所做出的一个逻辑处理过程。因此，我们综合这两方面因素，得出如下结论：

结论　对于有状态的响应类，响应类的属性变量是描述自身状态的核心元素，也是响应方法进行逻辑处理的核心数据依赖。

从对象的运作模式的角度来看，XWork 中负责对请求进行响应的 Action 类是一个运作在**属性－行为模式**下的对象。因而，“属性特征”和“行为特征”作为面向对象概念中的两个不同方面，也使得 Action 对象表现出两种截然不同的特性：

❑ 属性特征

对象的属性特征主要表述对象的三个不同特性：对象的自身状态、对象与其他对象之间的从属关系和对象与其他对象之间的协作关系。在 XWork 的 Action 中，主要体现了其中的第一点和第三点。其中，对于对象自身状态的描述，**成为 XWork 进行数据访问的基础**；而对于对象之间协作关系的描述，**成为 Action 与业务逻辑操作接口进行整合的基础**。

❑ 行为特征

对象的行为特征主要用于表述一个对象的动作特性，因为其主要表现元素的方法从语法的角度来讲，就是一个行为动作的概括。在 XWork 的 Action 中，对象的行为特征就是指对特定请求进行响应的过程。

因此，**XWork 的 Action 是一个状态与动作的合体**。从 Action 自身的接口定义中，我们可以看到其动作特性，这也正好迎合了一个响应类的基本要素。从 Action 实现类的构成体系这一角度来看，这个响应类还是一个可以进行自描述的有状态对象。作为一个请求响应类，Action 的状态就自然而然地被解读为请求数据和响应数据，它们共同构成了响应动作的数据基础。

8.3.2 Interceptor——腾飞的翅膀

8.3.2.1 Interceptor 的基本概念

Interceptor（拦截器）是原本属于 AOP（面向切面编程）中的概念，**其本质是一个代码段**，可以通过定义织入点（一个特定的编程元素，既可以是对象，也可以是对象中的方法），来指定 Interceptor 的代码逻辑在织入点元素的之前或者之后执行，从而起到“拦截”的作用。在这里，我们可以首先了解一下 AOP 中的相关概念。因为这些概念对于我们深入了解 Interceptor 的设计原理起着举足轻重的作用。

AOP 的相关概念

切面（Aspect）——一个关注点的模块化，这一关注点的实现可能横切多个对象，而这个模块化过程，由 Interceptor 来实现。例如，数据库的事务管理就是一个典型的切面。

通知（Advice）——在特定的连接点，AOP 框架执行的动作。各种通知类型包括：Before 通知、After 通知、Around 通知和 Throw 通知等。

切入点（Pointcut）——指定一个通知将被引发的一系列连接点的集合。AOP 框架必须允许开发者指定切入点，例如，使用正则表达式来指定触发通知的集合特征。

连接点（Joinpoint）——程序执行过程中明确的点，如方法的调用或特定的异常被抛出。

为了了解 XWork 中的 Interceptor，我们不妨首先打开 XWork 自带的 Reference，看看其中对 Interceptor（拦截器）是如何进行描述的：

Interceptors can execute code before and after an Action is invoked.

事实上在这段话中，我们已经可以看到 XWork 中 Interceptor 的概念与基本的 AOP 概念之间的对应关系：

- **切面**（Aspect）——Interceptor 实现
- **通知类型**（Advice）——环绕通知（Around 通知）
- **切入点**（Pointcut）——Action 对象
- **连接点**（Joinpoint）——Action 的执行

通过把这些概念综合起来，我们大致能够得出这样的结论：

观点　在 XWork 中所定义的 Interceptor，是一组环绕在切入点（Action 对象）的执行切面，可以在 Action 调用之前或者 Action 调用之后执行，从而对 Action 对象起到拦截作用。

了解了 Interceptor 的基本概念，我们再来通过 XWork 的宏观视图观察一下 Interceptor 对象在整个 XWork 框架中所处的位置。此时，我们需要把宏观视图再度缩小，并抽取其中与 Interceptor 对象连接的部分，构成如图 8-6 的示意图。

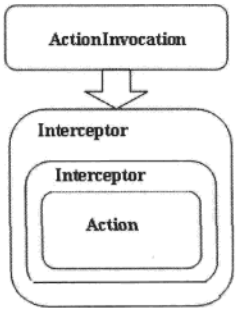


图 8-6　Interceptor 在 XWork 框架中的位置示意图

从这个示意图中，我们至少可以看到 Interceptor 在框架中所表现出的特点：

❑ **群居**

Interceptor 对象是一个群居对象，在同一时刻总有多个 Interceptor 对象同时存在并协同工作。

❑ **栈结构**

Interceptor 对象与 Interceptor 对象之间互相“包裹”，从而形成了数据结构上的“栈”结构。同时，Interceptor 对象所构成的“栈”结构还将 Action 对象包裹在最里端，这样一来 Action 对象就处于“栈”的底部，是一个栈底元素。

❑ **栈内元素**

整个“栈”结构形成了一个密不可分的整体，除了位于栈底的 Action 对象，其余的栈内元素都是 Interceptor 对象。

❑ **执行栈**

ActionInvocation 所进行的逻辑调度，针对的是整个栈结构，我们通常把这个栈结构称之为**执行栈**。也就是说，Interceptor 对象和 Action 对象对于 ActionInvocation 来说是一个整体，在调度执行时按照栈操作中的“先进后出”的原则，Interceptor 按照顺序依次执行，最后才轮到 Action 的执行。

这些 Interceptor 对象的特点，基本上都是通过“看图说话”的方式获得的，因而我们有必要通过源码进行验证。接下来，我们就来看看 Interceptor 的源码定义。

8.3.2.2 Interceptor 的定义

在 XWork 中，Interceptor 对象被定义成一个接口，其相关源码如代码清单 8-14 所示。

代码清单 8-14 Interceptor.java

```
public interface Interceptor extends Serializable {

    /**
     * 在 Interceptor 销毁时进行调用
     */
    void destroy();

    /**
     * 在 Interceptor 初始化时调用，主要用于初始化 Interceptor 需要使用的资源，
     * 在整个 Interceptor 的生命周期中只执行一次
     */
    void init();

    /**
     * 针对每一个请求，在 Interceptor 元素执行的过程中被调用
     *
     * @param invocation
```

```
     * @return the return code
     * @throws Exception
     */
    String intercept(ActionInvocation invocation) throws Exception;

}
```

这个接口定义非常普通，其中给出了 Interceptor 对象在生命周期的不同阶段可以进行逻辑扩展操作的方法。在这其中，init 和 destroy 方法仅仅会在 Interceptor 对象初始化和销毁的时候被调用一次，主要用于对 Interceptor 内部所依赖的外部资源进行初始化和销毁的管理。而 intercept 方法才是在整个接口定义中真正起到对 Action 对象进行拦截作用的核心方法。我们在这里结合一个具体的例子，来对接口中的这一核心方法进行大致的分析。相关的示例代码如代码清单 8-15 所示。

代码清单 8-15　EmptyInterceptor.java

```
public class EmptyInterceptor extends AbstractInterceptor {

    public String intercept(ActionInvocation invocation) throws Exception {
        return invocation.invoke();
}

}
```

在这个例子中，我们并没有加入任何复杂的逻辑。在方法体中，仅仅调用了 ActionInvocation 中的核心调度方法。我们的目的在于帮助读者对 Interceptor 的组成结构有一个初步的认识。

我们首先来看看 intercept 方法的参数：ActionInvocation。通过之前的分析，我们知道 ActionInvocation 是 XWork 控制流元素中的核心调度元素。在 intercept 方法中使用这样一个元素作为参数，主要基于两点考虑：

❑ 便于 Interceptor 随时与控制流和数据流的其他元素沟通

ActionInvocation 的操作接口中不仅包含对控制流元素 Action 和 ActionProxy 的访问接口，同时也包含对数据流元素 ActionContext 和 ValueStack 的访问接口。这样一来，在 Interceptor 内部，我们可以与 XWork 的所有重要元素进行沟通。

❑ 便于 ActionInvocation 在 Interceptor 的内部进行执行调度

ActionInvocation 的作用就是对 Interceptor 和 Action 进行执行调度。在 Interceptor 的内部，我们可以通过操作 ActionInvocation 的调度接口，对整个控制流元素进行执行调度。

我们再来看看 intercept 方法的内部实现。在这里只有一句简单的对 ActionInvocation 的操作接口调用：

```
invocation.invoke();
```

翻开 ActionInvocation 的所有操作接口就可以发现，invoke 方法是整个 ActionInvocation 的核心。其所表达的意思是说：**对 Interceptor 对象和 Action 对象共同构成的执行栈进行逻辑执行调度**。这个对于执行栈的调度，是 XWork 事件处理器的核心。我们在之后的章节会进行源码分析。不过这个 invoke 方法的调用，却代表了执行栈的一种逻辑执行方向。

在 Interceptor 中，存在着两种不同的执行逻辑：

❑ **调用 ActionInvocation 的 invoke 方法来指定对执行栈的进一步调度执行**

这是绝大多数 Interceptor 对象所选择的方式。因为一个执行栈代表了一个请求的执行总过程，而 Interceptor 只是执行栈中的一段逻辑。Interceptor 在完成了自身的逻辑之后，**有责任把执行的控制权转移到执行栈中的下一个元素继续执行**，而执行栈中的下一个元素不是 Interceptor 对象就是 Action 对象，因此它们又一次形成了一个自动的**递归调用链**。而 invoke 方法既是触发整个递归调用链的入口，其调用的结果也是整个执行栈的执行结果。也就是说，**invoke 方法的调用触发了执行栈中剩余 Interceptor 对象和 Action 对象的执行完成并返回结果**。

❑ **直接返回一个 String 类型的 ResultCode 来中止执行栈的调度执行**

这是 Interceptor 对象逻辑处理过程中的一种特殊情况。相当于在一个正常事件流执行过程中所发生的异常事件分支，其主要目的在于提供 Interceptor 中止整个执行栈继续执行的功能。

综合 Interceptor 的参数和核心逻辑实现，我们可以发现 Interceptor 对象是一个异常灵活的控制流元素。它丰富了整个控制流的执行层次，成为对一个请求的处理过程进行划分的基础元素。围绕在核心处理类 Action 周围的 Interceptor，为 Action 插上了腾飞的翅膀，从而彻底解放了整个控制流的生产力。

8.3.2.3 是 AOP 还是 IoC

众所周知，AOP 的实现有多种方式。Spring Framework 在 AOP 的实现和推广上为整个开源社区做了一个典范。因为 Spring 将 AOP 的实现与它另外一个核心构建：容器有机结合在一起。在这里并不对 Spring 的 AOP 实现方式展开分析，不过我们可以引用一个非常经典的对 Spring Framework 的 AOP 实现的总结作为一个核心结论：

结论 Spring 是用 IoC 来实现 AOP。

这个结论的意思是说，Spring 通过实现容器完成对象生命周期和关联关系的管理。而我们在使用 Spring 进行面向切面的编程时，只需要对 AOP 的众多要素进行 bean 的定义，

同时指明它们之间的调用关系，并将它们纳入到 Spring 的容器中进行管理，就可以利用 Spring 的组件来实现 AOP 编程。

不过在本书中，我们讨论的核心是 XWork 的 Interceptor。同样作为 AOP 编程的实现，XWork 的实现机制与 Spring 的实现机制之间又有什么区别呢？我们采用一个对照表的形式，给出两者在 AOP 构成要素中的不同来进行说明，如表 8-2 所示。

表 8-2　XWork 的 AOP 实现与 Spring 的 AOP 实现之间的对照

AOP 元素	Spring 实现机制	XWork 实现机制
切面（Aspect）	拦截器	拦截器
通知（Advice）	Around、Before、After	Around、Before、After
切入点（Pointcut）	满足特定条件的方法	Action 对象
连接点（Joinpoint）	方法的执行	Action 的执行

从这个表中，我们可以看到 Spring 和 XWork 对于 AOP 实现之间的本质区别就在于**两者对于切入点的定义完全不同**。或者说，**拦截器所拦截的对象不同**。Spring 主要偏重的是方法级别的 AOP 拦截，而 XWork 则针对 Action 对象进行拦截。

那么，XWork 对 Action 对象的拦截，对 Action 的执行有什么样的影响呢？我们可以将上一节中曾经使用过的那个 EmptyInterceptor 的例子做一个扩展，成为一个真正意义上的 AroundInterceptor，并以它作为一个例子进行说明。相关代码如代码清单 8-16 所示。

代码清单 8-16　AroundInterceptor.java

```
public abstract class AroundInterceptor extends AbstractInterceptor {

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

        // 在整个执行栈执行完毕之前的逻辑扩展
        before(invocation);

        // 驱动执行栈的进一步执行
        result = invocation.invoke();

        // 在整个执行栈执行完毕之后的逻辑扩展
        after(invocation, result);

        return result;
    }

    public abstract void before(ActionInvocation invocation) throws Exception;
```

```
    public abstract void after(ActionInvocation invocation, String
resultCode) throws Exception;

}
```

在上面的例子中，我们使用了 before、after 这两个抽象方法来对 Interceptor 的执行过程进行说明。整个代码的逻辑过程十分简单，before 和 after 这两个方法，分别在核心调度方法 invocation.invoke 调用之前和调用之后执行。这在效果上看，实现了以 ActionInvocation 的 invoke 方法调用为中心的拦截扩展。在上一节中，我们谈到 invoke 方法的逻辑意义在于，它触发并完成了对执行栈中剩余执行元素（包括其余的 Interceptor 对象，尤其是 Action 对象）的执行调度。那么，我们就可以将整个 AroundInterceptor 的逻辑理解为：

- **位于 invocation.invoke 方法调用之前的代码，会在 Action 对象执行之前执行**
- **位于 invocation.invoke 方法调用之后的代码，会在 Action 对象执行之后执行**

这不正好实现了 Interceptor 对象对 Action 对象的拦截吗？因此，我们可以看到 XWork 中的 AOP 实现，完全是通过 Interceptor 和 Action 共同构成的执行栈与 ActionInvocation 的配合调度共同完成的。在这期间并没有借助容器的帮助。

除了没有借助容器帮助以外，XWork 的 AOP 实现区分于 Spring 的 AOP 实现的唯一核心要点就在于切入点。在上面的代码中，我们已经看到了 Interceptor 对于 Action 执行顺序的影响。稍有一些面向切面编程经验的程序员应该会敏感地意识到，执行顺序的改变只是 AOP 编程的基本要求，而 **AOP 编程的本质，是对切入点运行状态的改变**。

在 Spring 的 AOP 实现中，切入点是方法。因此在拦截器内部，我们所围绕的是针对构成方法的一些基本要素的操作。这可以从拦截器的接口定义中看出来，其相关源码如代码清单 8-17 所示。

代码清单 8-17 InvocationHandler.java

```
public interface InvocationHandler {

   public Object invoke(Object proxy, Method method, Object[] args)
          throws Throwable;

}
```

实现 InvocationHandler 接口的拦截器，在其内部的实现逻辑中，总是围绕代表着方法定义的 Method 对象或者代表着参数内容的 args 数组来做一些文章：或者改变 Method 对象的执行逻辑，或者根据某些条件改变 args 的内容等等。无论这个文章怎么做，**拦截器自身的逻辑都离不开通过对拦截对象状态的改变而进行的行为扩展**。

在 XWork 的 AOP 实现中，切入点变成了 Action 对象，也就是 Interceptor 的拦截对象。这样一来，我们就可以简单地从上面的结论中得到一个推论：**XWork 中 Interceptor 的主要职责，就是通过对 Action 状态的改变而对 Action 的行为进行扩展。**而我们知道，一个对象状态主要是由身处这个对象之中的属性来表达的，那么结合这个推论，我们就可以得出在 XWork 中 Interceptor 对象与 Action 对象之间的操作关系：

结论　XWork 中的 Interceptor 是对 Action 对象中的属性进行管理的天然场所。

这个结论是通过不断推导而得出的。接下来，我们就有必要对 Action 对象的属性做一番深刻的研究。Action 本身作为一个运行在属性 - 行为模式上的对象，其内部属性主要表现出以下三个方面的特质：

- **构成 Action 这个响应类与外部环境的交互接口**
- **构成 Action 这个响应类的请求数据和响应数据**
- **构成 Action 的行为动作与业务逻辑处理对象之间的协作关系**

此时，我们可以换一个角度再来观察这个 Action 对象。如果把 Action 对象看作是需要进行生命周期管理的对象，那么针对 Action 内部属性变量的管理，不正是对 Action 运行过程中最重要的三大关系（环境依赖关系、数据依赖关系和协作依赖关系）的管理吗？这一管理的过程，与容器进行对象依赖关系的 IoC 过程有着惊人的相似。因此，对于 XWork，我们依然可以回到在本节开头曾经提到过的一个结论，只不过在这里，这个结论的逻辑发生了逆转性的改变：

结论　XWork 是用 AOP 来实现 IoC。

读者在这里可以把本节开头的一个结论和这个结论结合起来看，从而更加深刻地理解 XWork 的设计原理。可以说，使用 AOP 来实现 IoC，是整个 XWork 框架的核心结论，也是本书最为重要的几大结论之一。读者还可以结合我们在接下来的 ActionInvocation 章节中对控制流元素的调度的分析，加深对这个结论的理解。

8.3.2.4　Interceptor 的逻辑意义

回顾一下到目前为止对整个 XWork 框架的解读，我们发现 XWork 的逻辑性和条理性特别强。无论是 XWork 对控制流和数据流的划分，还是对每个构成体系中元素的定义，都充分体现出 XWork 对请求 – 响应模式的逻辑梳理。

在这里，我们把视线范围缩小到控制流元素，并聚焦在 Interceptor 上，Interceptor 对象的定义对于整个 XWork 的控制流又有什么逻辑意义呢？我们可以从几个不同的角度来进行分析。

第一个观察角度，我们站在事件处理流程规范化的角度来看待 Interceptor 对象。Interceptor 连同 Action 和 Result 被称为事件处理节点，它们都是事件处理流程规范化过程中对事件的流程进行进一步细化的结果。我们曾经在之前的分析中这样描述 Interceptor 和 Action 之间的关系：这是一个策应部队和主力部队之间的关系。这就如同做任何事情都要分清主次，当我们将事件处理的主要职责（进行核心业务逻辑处理）交付给 Action 对象，而将一些次要的职责（设置 Action 的运行参数等）交付给 Interceptor 对象，这就意味着我们在逻辑上对整个事件的处理流程有着深刻的认识，并能将不同的职责分而治之，并分派到不同的对象上去执行。分而治之、各司其职的思想可以说在这里充分得以体现。因此，有关 Interceptor 对象的第一个逻辑意义也就归纳出来了：

结论 Interceptor 对象的引入实际上是对事件处理流程中主要职责和次要职责的有效划分，并让每一个执行层次都能够完成其必要的职责归属。

第二个观察角度，我们站在 Interceptor 对象与 Action 对象共同构成的执行栈的角度来看 Interceptor 对象。站在这个观察角度，我们可以看到两个重要的特性：其一，多个 Interceptor 对象在核心响应处理类 Action 的周围形成了众多的执行层次；其二，整个执行栈的结构成为 XWork 控制流实现 AOP 的数据结构的基础。在传统的 Servlet 实现模式中，由于响应类是一个无状态类，因而在响应方法的内部实现对编程层次的划分。而在 XWork 的实现模式下，这完全不成为问题。Interceptor 对象反而成为 Action 类进行数据和状态维护的最佳场所。因而，我们在这里可以得出有关 Interceptor 对象的第二个逻辑意义：

结论 Interceptor 对象的引入能够极大地丰富整个事件处理流程的执行层次，从而为实现 AOP 编程打下坚实的数据结构基础。

最后一个观察角度，我们站在程序执行流程的角度来看 Interceptor 对象所起的作用。我们发现，Interceptor 对象与 Action 对象共同构成的执行栈在被 ActionInvocation 调度执行的过程中，应用了责任链（Chain Of Responsibility）模式。在本书第 4 章对责任链模式的讲解中，我们曾经强调过责任链模式的一个重大逻辑意义在于改变事件处理流程中顺序处理的执行逻辑。整个执行栈，在执行时顺序调度的过程被转化为对一个链表元素进行遍历的循环过程，当这个过程与 ActionInvocation 形成递归调用时，就自然而然形成了一个 AOP 模型的雏形。因而，我们得到有关 Interceptor 对象的第三个逻辑意义：

结论 Interceptor 对象的引入能够使得责任链模式在整个执行栈的调度过程中顺利实施，从而改变事件处理流程中的顺序处理逻辑，自然形成 AOP 模型的雏形。

8.3.3　ActionInvocation——核心调度

在之前对 Interceptor 的讲解中，我们已经多次提到 ActionInvocation。我们也已经看到 ActionInvocation 在 XWork 控制流中的重要作用：**核心调度器**。

8.3.3.1　ActionInvocation 全景图

在深入分析 ActionInvocation 的调度功能之前，我们将首先为读者带来 ActionInvocation 的全景图：ActionInvocation 的接口定义。其相关源码如代码清单 8-18 所示。

代码清单 8-18　ActionInvocation.java

```
public interface ActionInvocation extends Serializable {

    /**
     * 获取与当前 ActionInvocation 绑定的 Action 对象
     *
     * @return the Action
     */
    Object getAction();

    /**
     * 获取位于 ActionInvocation 的一个标识，该标识表明 ActionInvocation 是否已经
     * 成功完成了对 Action 和 Result 对象的调度执行
     *
     * @return
     */
    boolean isExecuted();

    /**
     * 获取与当前 ActionInvocation 绑定的 ActionContext
     *
     * @return ActionContext
     */
    ActionContext getInvocationContext();

    /**
     * 获取与当前 ActionInvocation 绑定的 ActionProxy
     *
     * @return ActionProxy
     */
    ActionProxy getProxy();

    /**
     * 获取 Result 对象
     *
     * @return
     * @throws Exception
```

```
     */
    Result getResult() throws Exception;

    /**
     * 获取 ActionInvocation 调度执行的结果代码，以一个字符串形式表达
     *
     * @return
     */
    String getResultCode();

    /**
     * 设置 ActionInvocation 的调度执行的结果代码，往往用于重置 Action 对象的
     * 执行结果
     *
     * @param resultCode
     * @throws IllegalStateException
     * @see #isExecuted()
     */
    void setResultCode(String resultCode);

    /**
     * 获取与当前 ActionInvocation 绑定的 ValueStack
     *
     * @return the ValueStack
     */
    ValueStack getStack();

    /**
     * 注册一个 PreResultListener 的实现类，这个实现类中的扩展逻辑将于 Action 对
     * 象执行完毕、Result 对象执行之前执行
     *
     * @param listener
     */
    void addPreResultListener(PreResultListener listener);

    /**
     * ActionInvocation 的核心调度方法
     *
     * @throws Exception
     * @return
     */
    String invoke() throws Exception;

    /**
     * 单单执行 Action 的操作接口，这个接口在自行控制 ActionInvocation 的调度逻辑时
     * 非常有用
     *
     *
     * @return the return code
```

```
     * @throws Exception can be thrown
     */
    String invokeActionOnly() throws Exception;

    /**
     * 设置 ActionEventListener
     *
     * @param listener the listener
     */
    void setActionEventListener(ActionEventListener listener);

    void init(ActionProxy proxy) ;

}
```

从 ActionInvocation 的接口定义全景图中，我们可以大致根据接口方法的不同作用对接口进行逻辑分类：

- **对控制流元素和数据流元素的访问接口**——getAction、getActionProxy、getStack 等等
- **对执行调度流程的扩展接口**——addPreListener、setActionEventListener
- **对执行栈进行调度执行的接口**——invoke、invokeActionOnly

而这三类不同的接口，恰巧反映出 ActionInvocation 的设计逻辑。其中，对执行栈进行调度执行是 ActionInvocation 的核心功能，也是这个类存在的主要意义。可以说，其他两类接口都是起辅助作用并为了这一接口功能服务的。在这些辅助功能中，对控制流元素和数据流元素的访问接口可以保证 ActionInvocation 在调度的过程中能够畅通无阻地与任何 XWork 框架内的元素进行沟通。作为 XWork 的核心调度，它就如同一个指挥官，必须能够实时知晓每一个组成部分的运行状态，从而能够根据情况对这些元素的执行进行调度。

8.3.3.2　ActionInvocation 调度分析

ActionInvocation 的核心调度功能是通过 invoke 方法完成的。我们在之前有关 Interceptor 的分析中，已经见过这个方法。不过站在 Interceptor 的角度，我们期望展示的是 invoke 方法的执行结果。因此，当时对于 invoke 这个方法的解读是：**对 Interceptor 对象和 Action 对象共同构成的执行栈进行逻辑执行调度。**

这样的初步解读显然无法令我们彻底理解 ActionInvocation 调度的实现机理。因而，我们有必要深入探究一下 invoke 这个方法的实现要点。在进行具体分析之前，我们可以先来看看这个类的作者对 invoke 方法所做的注释：

If there are more Interceptors, this will call the next one. If Interceptors choose not

to short-circuit ActionInvocation processing and return their own return code, they will call invoke() to allow the next Interceptor to execute. If there are no more Interceptors to be applied, the Action is executed.

从这段注释中，我们可以解读出三层意思：

- ❑ 如果执行栈中的下一个元素是 Interceptor 对象，那么执行该 Interceptor
- ❑ 如果执行栈中的下一个元素是 Action 对象，那么执行该 Action 对象
- ❑ 如果执行栈中找不到下一个执行元素，那么执行中止，返回执行结果 ResultCode

这三层意思合起来，就构成了 invoke 方法的大致雏形。invoke 方法的相关源码如代码清单 8-19 所示。

代码清单 8-19 DefaultActionInvocation.java

```
    public String invoke() throws Exception {
        String profileKey = "invoke: ";
        try {
            UtilTimerStack.push(profileKey);

            // 首先判断 ActionInvocation 的执行状态，如果已经执行过，则抛出异常
            if (executed) {
                throw new IllegalStateException("Action has already executed");
            }

            // 首先对所有的 Interceptor 对象进行调度
            // 所有的 Interceptor 在 ActionInvocation 中被有序地置于一个迭代器中
            // 对 Interceptor 对象的调度，实际上是对迭代器的遍历过程
            if (interceptors.hasNext()) {
                final InterceptorMapping interceptor = (InterceptorMapping)
interceptors.next();
                String interceptorMsg = "interceptor: " +
interceptor.getName();
                UtilTimerStack.push(interceptorMsg);
                try {
                        // 这里是整个 ActionInvocation 调度的核心
                    // 将 ActionInvocation 的实现类作为参数传入 Interceptor 执行
                    // 结合拦截器的实现代码就会发现，这里蕴含了一个递归调用
                        resultCode =
interceptor.getInterceptor().intercept(DefaultActionInvocation.this);
                }
                finally {
                    UtilTimerStack.pop(interceptorMsg);
                }
            } else {
                // 如果执行栈中没有 Interceptor 对象了，直接执行 Action 对象
                resultCode = invokeActionOnly();
```

```
            }

            // Interceptor 和 Action 调度完毕，执行 PreResultListener 逻辑
            if (!executed) {
                if (preResultListeners != null) {
                    for (Object preResultListener : preResultListeners) {
                        PreResultListener listener = (PreResultListener)
preResultListener;

                        String _profileKey = "preResultListener: ";
                        try {
                            UtilTimerStack.push(_profileKey);
                            listener.beforeResult(this, resultCode);
                        }
                        finally {
                            UtilTimerStack.pop(_profileKey);
                        }
                    }
                }

                // 最后执行 Result 对象的逻辑
                if (proxy.getExecuteResult()) {
                    executeResult();
                }

                executed = true;
            }

            return resultCode;
        }
        finally {
            UtilTimerStack.pop(profileKey);
        }
    }
```

从整个调度的源码中，我们可以看到 Interceptor、Action、PreResultListener 和 Result 这四大元素构成了 ActionInvocation 进行执行调度的主体对象。这四大主体对象实际上与我们之前所强调的事件处理节点是一一对应的。每一个事件处理节点代表一个响应处理的执行层次，而每一个执行层次也恰巧成为我们可以进行逻辑扩展的扩展点。

结合在上一节中分析的 Interceptor 所支持的通知（Advice）类型，我们可以总结出 XWork 事件处理流程所支持的三种不同的拦截方式：

❑ before 拦截

before 拦截，是指在拦截器中定义的代码，它们存在于 invocation.invoke() 代码执行之前。这些代码将依照拦截器栈所定义的顺序顺序执行。

❑ after 拦截

after 拦截，是指在拦截器中定义的代码，它们存在于 invocation.invoke() 代码执行之后。这些代码将依照拦截器栈所定义的顺序逆序执行。

❑ PreResultListener 拦截

PreResultListener 拦截，是指实现了 PreResultListener 接口的实现逻辑，在 Interceptor 和 Action 执行完毕之后、Result 执行之前执行。

我们看到 XWork 能够支持如此多的拦截类型，与 XWork 对于请求－响应的实现模式有着很大的关系。正因为 XWork 采用了 POJO 实现模式，才使得设计模式可以在这里有充分的施展空间。XWork 对于整个事件处理节点的定义，无论是 Interceptor、Action 还是 Result，每一层都有着明确的职责，并在每个节点之间都设立了恰如其分的逻辑扩展空间，使得整个 Controller 层的扩展性得到了史无前例的提升。

在这里，我们最为关心的莫过于 Interceptor 和 Action 共同构成的执行栈的执行顺序。从上面的分析中，我们可以看到 before 拦截和 after 拦截同时在 Interceptor 对象中完成，而它们却拥有完全不同的执行次序。这又是为什么呢？我们可以把关注的焦点放在源码中最核心的一个调用上：

```
interceptor.getInterceptor().intercept(DefaultActionInvocation.this);
```

表面上，它只是执行了拦截器中的 intercept 方法。但是如果结合拦截器的实现来看，就能看出其中的奥秘了。我们可以引用之前曾经引用过的一个 AroundInterceptor 的代码示例来进行说明：

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

        // 在整个执行栈执行完毕之前的逻辑扩展
        before(invocation);

        // 驱动执行栈的进一步执行
        result = invocation.invoke();

        // 在整个执行栈执行完毕之后的逻辑扩展
        after(invocation, result);

        return result;
}
```

原来，在拦截器的逻辑实现中，又对 ActionInvocation 的 invoke 方法实施了**递归调用。递归调用的嵌套执行使得以递归调用逻辑为中心的代码段（invocation.invoke 的调用）成为逻辑上的分水岭**，从而彻底改变了原本正常的执行逻辑：

- 递归调用使得位于分水岭之后的代码逻辑被暂时封存，等待完整的递归调用结束之后才能继续执行
- 递归调用使得在执行栈中的下一个元素将被拿出来执行。从表面上看形成了栈结构的顺序遍历，实际上却形成了递归调用的嵌套，这就意味着在分水岭之前的逻辑代码将随着栈结构的遍历顺序执行
- 当整个递归调用完成之后，情况刚刚好相反。原本最后一个执行的元素对象将首先执行在分水岭之后的代码逻辑，从而触发整个逆序的执行过程

一个有序链表通过递归调用变成了一个堆栈执行过程，将一段有序执行的代码变成了两段执行顺序完全相反的代码过程，从而巧妙地实现了 AOP。我们在这里可以用一个例子验证我们的分析。

如果有一个 interceptor-stack 的定义如下：

```
<interceptor-stack name="xaStack">
    <interceptor-ref name="thisWillRunFirstInterceptor"/>
    <interceptor-ref name="thisWillRunNextInterceptor"/>
    <interceptor-ref name="followedByThisInterceptor"/>
  <interceptor-ref name="thisWillRunLastInterceptor"/>
</interceptor-stack>?
```

那么，经过 ActionInvocation 的调度，整个执行的序列如图 8-7 所示。

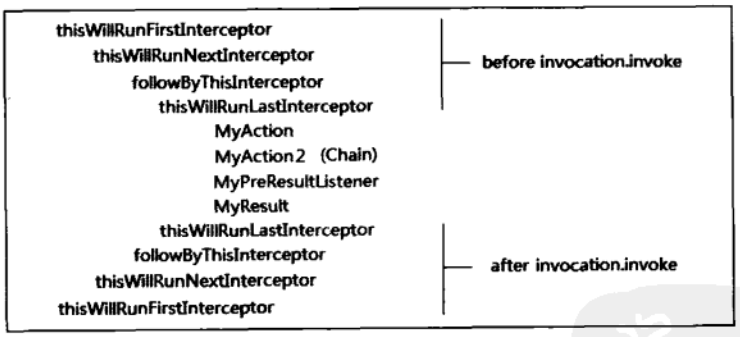


图 8-7　ActionInvocation 的调度示意图

在这里，我们不仅为读者展示了 Interceptor 的执行顺序，同时也给出了 Action 和 PreResultListener 结合在一起的执行顺序。

可以说，XWork 对于 AOP 实现的设计，实在是一个天才之作。在这里，框架的作者充分利用了普通数据结构的特性，并结合一个递归调用，寥寥数行代码就实现了 AOP。我们在这里应该学习这种将最为普通的知识转化为最佳实践的思想境界，并将其作为我们在实际开发中的目标。

8.3.4 ActionProxy——执行窗口

ActionProxy 是位于整个 XWork 控制流体系最外层的元素，也是整个 XWork 框架的门户。我们从 ActionProxy 的命名中就可以体会到它在整个 XWork 框架中所起的作用：**XWork 事件处理框架的总代理**。

8.3.4.1 ActionProxy 的定义

为了说明 ActionProxy 的代理作用，我们首先来看看 ActionProxy 的定义，其源码如代码清单 8-20 所示。

代码清单 8-20 ActionProxy.java

```
public interface ActionProxy {

    /**
     * 获取当前 ActionProxy 所代理的 Action 对象
     *
     * @return the Action instance
     */
    Object getAction();

    /**
     * 获取映射到当前 ActionProxy 对象的别名
     *
     * @return the alias name
     */
    String getActionName();

    /**
     * 获取当前 ActionProxy 所对应的配置对象
     *
     * @return the ActionConfig
     */
    ActionConfig getConfig();

    /**
     * 设置当前 ActionProxy 是否在 Action 执行完毕后执行指定 Result 的标志
     *
     * @param executeResult <tt>true</tt> to also execute the Result
     */
    void setExecuteResult(boolean executeResult);

    /**
     * 获取当前 ActionProxy 是否直接在 Action 执行完毕后执行指定 Result 的标志
     *
     * @return the status
```

```
     */
    boolean getExecuteResult();

    /**
     * 获取与当前 ActionProxy 绑定的 ActionInvocation
     *
     * @return the ActionInvocation
     */
    ActionInvocation getInvocation();

    /**
     * 获取当前 ActionProxy 所对应的配置元素的 namespace 值
     *
     * @return the namespace
     */
    String getNamespace();

    /**
     * ActionProxy 的执行接口

     *
     * @return
     * @throws Exception
     */
    String execute() throws Exception;

    /**
     * 获取 Action 对象中进行请求响应的方法名称。如果为空，则使用默认的 execute 方法
     *
     * @return the method to execute
     */
    String getMethod();

}
```

从 ActionProxy 的接口定义中，我们可以发现 ActionProxy 与 ActionInvocation 一样，也是一个包罗万象的接口。只不过身处比 ActionInvocation 更高的层次，它与 ActionInvocation 的操作接口的侧重点完全不同。我们可以从 ActionProxy 的定义中看到 ActionProxy 与众多配置元素相关的操作接口。这就表明 ActionProxy 作为一个代理类，它有一个重大的职责：

结论　ActionProxy 的首要职责是维护 XWork 的执行元素与请求对象之间的配置映射关系。

一方面，ActionProxy 对于调用者而言屏蔽了 XWork 的调用细节。对于调用者，可

以并不知晓 Interceptor 对象或者 Action 对象的存在，只要将请求的内容通过代理类 ActionProxy 进行中转即可。另外一个方面，ActionProxy 对于 XWork 内部的元素而言，则提供了 ActionProxy 调用者的相关信息。当然，这些信息经过了一定的处理，已经被归纳为一些通用而可识别的对象。例如 namespace、method 等。

8.3.4.2 ActionProxy 的初始化

了解了 ActionProxy 的定义，我们再来看看 ActionProxy 的初始化过程。与众多 XWork 的内置元素一样，ActionProxy 的构建过程也是通过工厂方法完成的，其相关源码如代码清单 8-21 所示。

代码清单 8-21 DefaultActionProxyFactory.java

```
public class DefaultActionProxyFactory implements ActionProxyFactory {

    protected Container container;

// 通过依赖注入操作获得容器接口
    @Inject
    public void setContainer(Container container) {
        this.container = container;
}

    public ActionProxy createActionProxy(String namespace, String
actionName, String methodName, Map<String, Object> extraContext, boolean
executeResult, boolean cleanupContext) {

        // 创建与 ActionProxy 所关联的 ActionInvocation 对象
        ActionInvocation inv = new DefaultActionInvocation(extraContext,
true);
        // 对 ActionInvocation 实施依赖注入
        container.inject(inv);
        return createActionProxy(inv, namespace, actionName, methodName,
executeResult, cleanupContext);
}

    public ActionProxy createActionProxy(ActionInvocation inv, String
namespace, String actionName, String methodName, boolean executeResult,
boolean cleanupContext) {
        // 创建 DefaultActionProxy 的实现类
        DefaultActionProxy proxy = new DefaultActionProxy(inv, namespace,
actionName, methodName, executeResult, cleanupContext);
        // 对 ActionProxy 对象实施依赖注入
        container.inject(proxy);
        // 完成 ActionProxy 自身的初始化工作
        proxy.prepare();
        return proxy;
```

```
    }
    // 这里省略了许多其他的代码
}
```

首先观察构成 createActionProxy 方法的参数，这些参数主要分为两类：

- **配置关系映射**——namespace、actionName、methodName 等
- **运行上下文环境**——extraContext

我们可以发现，ActionProxy 的调用者首先必须明确当前 ActionProxy 与具体的 XWork 执行元素之间的配置映射关系。这一点我们在上一小节已经得出结论，而这一映射关系通过第一类参数传入 ActionProxy 的初始化方法。

而第二类参数，我们把它称为运行上下文环境。从参数的表现形式来看，它是一个 Map 结构，其中封装了所有在 XWork 执行过程中所需要的数据对象。因此，这个 extraContext 变量也是 XWork 构建其数据环境的基础。我们将在后面的章节中详细剖析 extrContext 的构成要素。

8.3.4.3　XWork 的执行门户

ActionProxy 的代理作用我们已经阐述得非常清楚。作为 XWork 的执行门户，ActionProxy 还有一个重要的作用，那就是在需要对 XWork 的事件处理流程进行单元测试的时候，我们必须借助 ActionProxy 来完成。因为真正的 XWork 控制流元素都被 ActionProxy 有效地隐藏在了背后，我们唯一能够打交道的就是 ActionProxy。

在 XWork 中有一个单元的范例类 XWork，它向我们展示如何进行 XWork 事件处理流程的测试，我们在这里摘取其中的一些核心代码来进行说明，如代码清单 8-22 所示。

代码清单 8-22　XWork.java

```
public void executeAction(String namespace, String name, String method,
Map<String, Object> extraContext) throws XWorkException {
        Configuration config = configurationManager.getConfiguration();
        try {
            ActionProxy proxy =
config.getContainer().getInstance(ActionProxyFactory.class).createActi
onProxy(
                    namespace, name, method, extraContext, true, false);

            proxy.execute();
        } catch (Exception e) {
            throw new XWorkException(e);
        } finally {
            ActionContext.setContext(null);
```

```
        }
    }
```

在这里，我们看到为了进行 XWork 事件处理流程的单元测试，需要将它和 XWork 框架的配置元素联系在一起。有关这一点，我们在后续章节中还会详细说明。

8.4 交互体系——水乳交融

之前我们分别从数据流体系和控制流体系内部挖掘了 XWork 构成元素之间的关系。然而，整个 XWork 视图的核心，实际上在于**数据流和控制流之间的交互体系**。为了讲清楚数据流和控制流之间的交互方式，我们重新把整个宏观视图呈现出来。如图 8-8 所示。

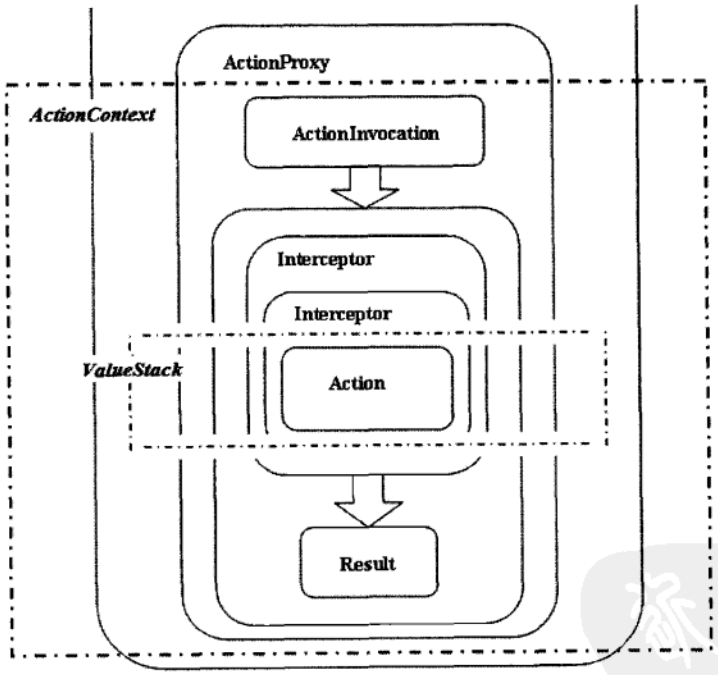


图 8-8 数据流和控制流的交互示意图

当我们讨论的话题转移到两个体系的交互关系时，“看图说话”的方法就不太够用了。因为图在这个时候并不像之前那么直观。而数据流和控制流之间的沟通方式，也比数据流内部元素之间简单的“从属关系”或者控制流内部元素之间的“调度关系”复杂得多。

不过通过对宏观视图的“看图说话”，我们也能够获取一些有用的信息。比如说，我

们可以发现 ActionContext 所在的虚线框，把整个控制流的元素都围了起来；而 ValueStack 所在的虚线框，把控制流的核心元素 Action 围在其中。这些很直观的信息如果加以分析，同样可以成为重要的结论。因而，我们在这里采取的将是先“看图说话”，再结合“理论分析”的方法对控制流和数据流的交互体系进行详细阐述。

8.4.1　数据环境的生命周期

首先，我们来谈谈作为数据环境最主要实现元素的 ActionContext 的生命周期问题。所谓生命周期，实际上牵涉 ActionContext 的创建一直到 ActionContext 销毁这样一个完整的过程。如果我们还是先进行“看图说话”，我们能够从图 8-8 中看到：ActionContext 所在的虚线框几乎包含了 XWork 执行的全过程。这一现象告诉了我们在这其中就蕴含的一个核心结论：

结论　*作为 XWork 的数据环境，ActionContext 的数据内容在整个 XWork 控制流的生命周期中共享。*

从图中虚线框和实线框所围起来的面积来看，似乎**代表着虚线框的数据流的生命周期比代表着实线框的控制流的生命周期更长**。事实上，这是一个正确的结论，图中虚线框和实线框的大小关系实际上也是笔者刻意为之。当然，这一点我们也可以通过源码来证明，相关源码如代码清单 8-23 所示。

代码清单 8-23　StrutsPrepareAndExecuteFilter.java

```
public void doFilter(ServletRequest req, ServletResponse res, FilterChain
chain) throws IOException, ServletException {

        try {

        // 这里省略了许多其他的代码
            prepare.createActionContext(request, response);

        // 这里省略了许多其他的代码
  if ( excludedPatterns != null && prepare.isUrlExcluded(request,
excludedPatterns)) {
            chain.doFilter(request, response);
  } else {

            // 这里省略了许多其他的代码

        execute.executeAction(request, response, mapping);
  }
        } finally {
```

```
                // 这里包含了 ActionContext 的销毁过程
                prepare.cleanupRequest(request);
            }
    }
```

从源码中，我们可以看到，ActionContext 是在控制流元素执行之前被构建，并在整个控制流元素执行完毕之后销毁，这表明 **ActionContext 是一个横跨了整个 XWork 控制流执行周期的元素**。

在介绍数据流体系时，我们曾经对 ActionContext 有一个职责上的认定：ActionContext 既负责数据存储，又负责数据共享。而 ActionContext 之所以能够横跨整个 XWork 的控制流执行周期而实施数据共享，主要是源于 ActionContext 采用了 ThreadLocal 模式进行数据结构的设计。ThreadLocal 模式保证了它可以将逻辑执行和运行参数有效解耦，从而达到**在整个当前执行线程共享数据**的目的。因此，**ActionContext 所进行的数据共享，不仅对整个 XWork 的控制流元素有效，它甚至对整个当前执行线程都有效**。

这个时候可以回过头来看看我们为什么使用“虚线”而非“实线”来定义 ActionContext 这一数据流元素，因为它看上去就像一个来自于平行世界的元素，它并不在任何控制流元素的定义中出现，可它却是隐藏在控制流背后的重要元素，支持着控制流元素的顺利执行。

结论 *作为数据环境，ActionContext 是隐藏于 XWork 控制流执行背后的重要元素，成为控制流元素与数据流元素的沟通桥梁。*

8.4.2 三军会师之地

在 XWork 中，如果站在控制流的角度来看待数据流，它就好像是另外一个平行世界中的元素。因为它的存在与控制流似乎毫无关系。我们根本无法在任何控制流的执行元素中看到它的身影。然而，平行世界与现实世界总是有一个交叉点，这个交叉点又在哪里呢？这就需要我们引入另外一个重要结论：

结论 *控制流中的核心处理元素 Action，被置于数据流元素 ValueStack 中。*

这个结论，恰好是我们把视线范围锁定在 ValueStack 所在的虚线框中得到的。我们发现，在 XWork 的宏观视图中，身为控制流元素的 Action，正好被身为数据流元素的 ValueStack 包围着。笔者这么画是否正确呢？我们还是从源代码中寻找答案吧。相关源码如代码清单 8-24 所示。

代码清单 8-24　DefaultActionInvocation.java

```
public void init(ActionProxy proxy) {

        // 这里省略了其他代码

        if (pushAction) {
            stack.push(action);
            contextMap.put("action", action);
        }

        // 这里省略了其他代码
}
```

在传统的请求－响应的实现模式中，**无论是参数－返回值模式还是参数－参数模式，控制流元素对于数据流元素都是有绝对的掌控权的。**因为作为数据流主要实现元素的参数和返回值，都被置于控制流元素的执行方法之中，并成为方法必不可少的构成要素。这主要是由于传统的请求－响应模式的实现是建立在 Java 编程语言的语法规则基础之上的。

而在 XWork 中，这种情况却被另外一种更加复杂的“映衬关系”代替。首先，控制流的核心元素被反过来置于数据流元素之中，使得这一核心元素可以完全摆脱对 Web 容器的依赖，由一个无状态的响应对象变为一个有状态的 POJO；其次，在控制流的核心元素执行时，由于它身处数据流元素的包围之中，因而它又能够轻而易举地对数据流的元素随时取用。

我们发现，在控制流的核心元素 Action 执行的过程中，它所依赖的请求参数实际上来源于我们所说的“平行世界”。这种与平行世界的交互，不仅是数据流和控制流解耦思想的体现，实际上还成为 XWork 框架中的数据访问基础。我们将在后续章节为大家详细揭开 XWork 框架中数据访问的种种谜团。

8.4.3　Action 交互体系

Action 作为 XWork 的核心处理元素，其对外交互体系主要是由构成 Action 的属性作为其主要的表现形式。在有关 Interceptor 的分析章节中，我们曾经就 Action 对象中属性的功能进行过分类，而这一分类体现了 Action 交互体系的三个不同的方面：

- **外部环境交互体系——构成 Action 与外部环境的交互接口**
- **数据交互体系——构成 Action 的请求数据和响应数据**
- **协作交互体系——构成 Action 与业务逻辑处理对象之间的协作关系**

同时，我们还指出，Action 的属性作为核心处理类交互体系的主要表现形式，而 Interceptor 则成为交互体系中的主要操作窗口。在接下来的分析中，我们就分别站在这三

个不同的角度，通过对相应的 Interceptor 源码的分析来说明 Action 对象的交互体系。

8.4.3.1 外部环境交互体系

Action 交互体系中的外部环境交互体系，主要是指在 XWork 的 Action 中如何与 XWork 执行环境之外的调用环境进行交互。具体来说，当 XWork 与 Struts2 配合运行在 Web 容器中，Web 容器就成为了 Action 对象最重要的外部环境，因而这里所说的交互体系实际上就是指 **Action 与 Web 容器中 HttpServletRequest、HttpServletResponse、HttpSession 对象的交互过程**。

读者或许要说，Action 与 Web 容器对象的交互在讲 ActionContext 的时候就介绍过了：通过 ServletActionContext 对象就可以实现。作为 ActionContext 的子类，ServletActionContext 中存储了所有线程安全的 Servlet 对象。也就是说，在一个请求－响应过程中，我们只要调用 ServletActionContext 中的静态方法，就可以随时获得 Web 容器对象了。

```
HttpServletRequest request = ServletActionContext.getRequest();
HttpServletResponse response = ServletActionContext.getResponse();
ServletContext servletContext = ServletActionContext.getServletContext();
```

使用静态方法调用来获取 Web 容器对象，虽然是一个安全而廉价的方法，但是看上去总不是那么面向对象。于是乎，Struts2 构建了另外一套 Web 容器对象的访问机制。这一套对象的访问机制首先建立在一系列的 Aware 接口之上。这些接口如图 8-9 所示。

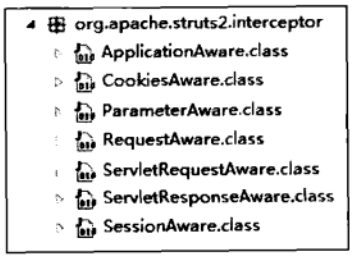


图 8-9 Struts2 的 Aware 接口

这些接口中仅仅包含了“设置相关对象操作接口”的方法。我们可以以 ServletRequestAware 接口的源码为例进行说明。相关源码如代码清单 8-25 所示。

代码清单 8-25 ServletRequestAware.java

```
public interface ServletRequestAware {

    /**
     *
     * @param request
```

```
     */
    public void setServletRequest(HttpServletRequest request);
}
```

若 Action 实现了如图 8-9 所示的接口，我们就可以在这些由 Aware 接口所带来的设置外部环境操作接口的方法中实现这些操作接口的本地存储。当然，绝大多数的情况下，我们会将它们存储在 Action 内部的属性变量中。

此时，ServletConfigInterceptor、CookiesInterceptor 等拦截器的作用就是在 Action 执行之前对 Action 进行接口匹配，一旦 Action 实现了某一个 Aware 接口，就调用该接口的方法完成逻辑。这样一来，在 Action 执行之时，就能够通过 Aware 接口所设置的这些操作对象，完成对 Web 容器对象的操作。

下面以 ServletConfigInterceptor 为例来说明，其相关的源码如代码清单 8-26 所示。

代码清单 8-26　ServletConfigInterceptor.java

```
public class ServletConfigInterceptor extends AbstractInterceptor
implements StrutsStatics {

    /**
     *
     * @param invocation
     * @throws Exception
     */
    public String intercept(ActionInvocation invocation) throws Exception {
        final Object action = invocation.getAction();
        final ActionContext context = invocation.getInvocationContext();

        if (action instanceof ServletRequestAware) {
            HttpServletRequest request = (HttpServletRequest)
context.get(HTTP_REQUEST);
            ((ServletRequestAware) action).setServletRequest(request);
        }

        if (action instanceof ServletResponseAware) {
            HttpServletResponse response = (HttpServletResponse)
context.get(HTTP_RESPONSE);
            ((ServletResponseAware) action).setServletResponse(response);
        }

        if (action instanceof ParameterAware) {
            ((ParameterAware)
action).setParameters((Map)context.getParameters());
        }

        if (action instanceof ApplicationAware) {
```

```
            ((ApplicationAware)
action).setApplication(context.getApplication());
        }

        if (action instanceof SessionAware) {
            ((SessionAware) action).setSession(context.getSession());
        }

        if (action instanceof RequestAware) {
            ((RequestAware) action).setRequest((Map)
context.get("request"));
        }

        if (action instanceof PrincipalAware) {
            HttpServletRequest request = (HttpServletRequest)
context.get(HTTP_REQUEST);
            if(request != null) {
                ((PrincipalAware) action).setPrincipalProxy(new
ServletPrincipalProxy(request));
            }
        }
        if (action instanceof ServletContextAware) {
            ServletContext servletContext = (ServletContext)
context.get(SERVLET_CONTEXT);
            ((ServletContextAware)
action).setServletContext(servletContext);
        }
        return invocation.invoke();
    }
}
```

这里的代码十分简单，不做过多的解释：逐次扫描 Action 是否实现了某个 Aware 接口，然后将 Action 强制转化成这个 Aware 接口并调用其方法完成交互对象的设置。

那么，我们从这些简单的代码中可以读到一些什么信息呢？实际上，ServletConfigInterceptor 这个拦截器为我们打造了一个对 Action 进行扩展的范例。这个范例的打造分为三个过程：

- **定义一个接口**
- **让 Action 实现该接口**
- **在拦截器中将 Action 强制转化为接口，完成接口方法的逻辑调用**

我们可以看到，这三个过程的代码逻辑调用蕴含了两种不同的 Action 扩展结果：其一，通过 Action 实现的接口，向 Action 传递外部信息；其二，通过 Action 实现的接口，完成功能扩展。

这里所采用的交互方式，其主要依据实际上是**面向对象概念中实现类与接口之间的关系**。读者可以细细品味实现这种交互方式的三个过程，因为它们实际上也是我们在进行 Struts2 扩展时使用最多的扩展方式之一。

8.4.3.2　数据交互体系

Action 的数据交互体系则反映了 Action 与数据流元素进行交互的过程。只是在这个过程中，我们看不到数据流元素的身影。因为数据流元素来自一个平行世界，其内容载体则以另外一种方式映射到 Action 的内部属性变量之中。我们要探究的就是这个“映射”的过程。

从之前的分析中，我们知道 Action 对于请求－响应的实现采用的是 POJO 模式。也就是说所有的请求数据和响应数据在 Action 中都以属性变量的形式出现，那么将请求数据从数据环境中读取，并设置到 Action 的属性变量的过程就是我们所说的“映射”的过程，而这个过程是由 ParametersInterceptor 这个拦截器来完成的。其核心源码如代码清单 8-27 所示。

代码清单 8-27　ParametersInterceptor.java

```
public String doIntercept(ActionInvocation invocation) throws Exception {
        Object action = invocation.getAction();
        if (!(action instanceof NoParameters)) {
            ActionContext ac = invocation.getInvocationContext();
            final Map<String, Object> parameters = retrieveParameters(ac);

            if (parameters != null) {
                Map<String, Object> contextMap = ac.getContextMap();
                try {
                 ReflectionContextState.setCreatingNullObjects(contextMap,
                 true);
                 ReflectionContextState.setDenyMethodExecution(contextMap,
                 true);
                 ReflectionContextState.setReportingConversionErrors(contextMap,
                 true);

                    ValueStack stack = ac.getValueStack();
                    setParameters(action, stack, parameters);

                } finally {
                 ReflectionContextState.setCreatingNullObjects(contextMap,
                 true);
                 ReflectionContextState.setDenyMethodExecution(contextMap,
                 true);
                 ReflectionContextState.setReportingConversionErrors(contextMap,
```

```
                    true);

                }
            }
        }
        return invocation.invoke();
    }
```

ParametersInterceptor的逻辑很复杂，因为它提供了对Action数据的全方位支持，包括Action对象是否支持接收请求参数、请求参数的排序、请求参数的过滤等等。这些功能支持都被封装在setParameters方法中。我们在这里截取的代码片段实际上是ParametersInterceptor拦截器的骨架部分。

从代码中，我们可以看到其中的核心方法其实是一个ValueStack的写值操作。为什么从ActionContext中获取的参数最终映射到Action的属性变量中，反而变成了一个ValueStack操作呢？要回答这个问题，我们不得不使用之前的分析中所得出的一系列结论将这个过程重新推导一遍，整个推导过程如图8-10所示。

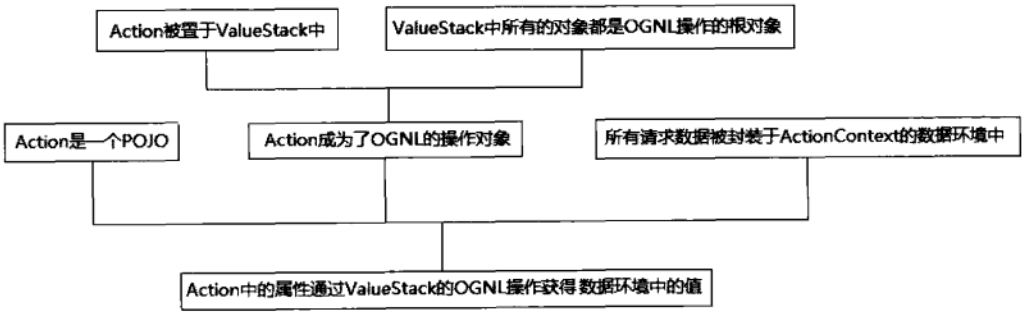


图8-10 Action中属性值来源的推导过程

由此可见，XWork在数据流和控制流的交互体系的设计上可谓是“步步精心”。因为我们从图8-10中可以看出，每一个推导步骤几乎都是我们之前分析中得出的重要结论。而这些结论在ParametersInterceptor中一经融合，Action的数据交互体系就自然而然形成了。这种水到渠成的过程，恰巧反映出XWork设计的精妙之处！

在ParametersInterceptor中，有一个方法非常值得我们注意，那就是retrieveParameters方法。在ParametersInterceptor的实现中返回的是来自ActionContext中的ParameterMap，实际上就其源头，也就是来自HttpServletRequest中的ParameterMap。如果站在整个ParametersInterceptor的角度来看待这个方法，我们可以得出一个很有用的结论：

结论　ParametersInterceptor 是一个将外部环境数据与 Action 的属性变量进行融合的骨架，而其中的 retrieveParameters 方法，就是外部环境数据的提供者。

这个结论对我们意义非凡。因为这个结论实际上**提供了一个数据环境映射到 Action 对象的途径和方法**。具体来说，我们只要继承 ParametersInterceptor，然后覆写其中的 retrieveParameters 方法，将其替换为自己的扩展逻辑，就可以轻松实现 Action 对象的数据交互。有关这里的扩展示例，我们不再具体展开，读者可以参考 Struts2 中的一个拦截器 ActionMappingInterceptor。

8.4.3.3　协作交互体系

Action 的协作交互体系说起来就比较拗口。因为从 Action 的运作模式来看，Action 对象与数据环境的交互体系是对于 Action 状态的一种管理模式。而 Action 的协作交互体系指的是将 Action 对象看作一个门户，如何获得业务逻辑操作对象的过程。

在这种情况下，业务逻辑操作对象往往是一个单例对象，甚至这个单例对象并不受 XWork 容器的管理。此时，Action 就表现为一个普通的请求响应类，这个类与普通的 Servlet 响应类的性质并无二异。在拦截器中要做的，只不过是将业务逻辑操作对象注入到 Action 类中。这个操作实际上也是一个 IoC 的过程。

在 Struts2 中，我们可以找到 ActionAutowiringInterceptor 作为一个例子来描述 Action 与 Spring Framework 管理的业务逻辑对象之间的整合关系。其相关源码如代码清单 8-28 所示。

代码清单 8-28　ActionAutowiringInterceptor.java

```
public String intercept(ActionInvocation invocation) throws Exception {
        if (!initialized) {
            ApplicationContext applicationContext = (ApplicationContext)
ActionContext.getContext().getApplication().get(

WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE);

            if (applicationContext == null) {
                LOG.warn("ApplicationContext could not be found.  Action
classes will not be autowired.");
            } else {
                setApplicationContext(applicationContext);
                factory = new SpringObjectFactory();
                factory.setApplicationContext(getApplicationContext());
                if (autowireStrategy != null) {

factory.setAutowireStrategy(autowireStrategy.intValue());
```

```
                }
            }
            initialized = true;
        }

        if (factory != null) {
            Object bean = invocation.getAction();
            factory.autoWireBean(bean);

            ActionContext.getContext().put(APPLICATION_CONTEXT, context);
        }
        return invocation.invoke();
    }
```

从源码中，我们可以看到 ActionAutowiringInterceptor 首先会初始化 Spring 容器的操作对象 ApplicationContext。当然，这个过程由 initialized 变量控制，保证只会执行一次。在这之后，为 Action 注入 Spring 容器中 bean 的过程就可以完全交给 ApplicationContext 来做，我们只需要将 Action 对象作为参数传入即可。

Action 的协作交互体系所反映出来的是 Action 作为请求响应门面（Facade）类的特性。而我们从 ActionAutowiringInterceptor 中可以读到的，实际上是 Struts2 作为一个表示层框架，与业务逻辑层框架进行整合的范例。

8.5 小结

在本章中，我们从 XWork 的宏观视图分析开始，试图为读者揭示构成 XWork 框架的每一个元素的实现细节以及这些元素之间的内在关系。

数据流和控制流是构成 XWork 的两大体系结构，构成这两大体系结构的元素本身就存在着千丝万缕的联系。在本章的一开始，我们采取“看图说话”的方式对这些联系进行了研究。紧接着，对于体系中的每一个元素，我们都从数据结构定义、行为方式以及设计要点等多方面进行了深入的剖析。最后，我们将两大体系结构的交互过程也呈现在读者的面前。通过本章的阅读，相信读者已经能够对 XWork 框架的细节了如指掌。

回顾本章，实在有太多的细节需要读者去慢慢品味。我们在这里列出一些问题，供读者思考：

- 构成 XWork 数据流体系的两大元素之间有什么样的关系？
- 什么是事件处理节点？什么是事件处理驱动元素？
- 如何理解 Action 对象与 Interceptor 对象之间的关系？

- ActionContext 有哪两大职责？
- 什么是 ValueStack？ ValueStack 对 OGNL 表达式的计算有什么样的影响？
- 作为核心响应类的 Action 与传统 Servlet 实现模式下的响应类有什么区别？
- 什么是执行栈？ ActionInvocation 对于执行栈是如何进行调度的？
- ActionProxy 在整个 XWork 框架中起什么作用？
- 控制流体系与数据流体系是如何进行交互的？
- Action 的交互体系主要分为哪三个方面？

第三部分

运行主线篇

Struts2 建立在一系列核心技术之上，这些技术共同构成了 Struts2 的骨架。程序骨架的作用是支撑起程序运行的基本依赖条件，因此也是整个程序的灵魂所在。有了程序骨架，我们再对整个程序运行机制的进行研究就可以畅通无阻。因而，本篇的主要内容就是以这些核心技术为基础，对 Struts2 的运行机制进行系统的分析。

我们知道，Struts2 有一个非常核心的设计理念，就是将程序的初始化过程与运行期逻辑进行解耦，成为两个完全独立的运作体系。这一点首先就在整个 Struts2 的运行主线上体现出来。本书的第 9 章和第 10 章，我们就试图带领读者详细解读 Struts2 的两条运行主线，其中的内容不仅涉及 Struts2 的核心运行机制，还将以源码作为我们最有力的武器，验证 Struts2 为解决 Web 开发中的种种问题所做的努力。围绕着 Struts2 的运行主线，我们还将在本书的最后两章介绍服务于 Struts2 运行主线的 Web 交互方式以及 Struts2 的扩展机制，从而帮助读者完善对整个 Struts2 体系结构的认识。

正如之前所说的，Struts2 任何一条运行主线都离不开核心技术的支撑。在 Struts2 的初始化主线中，有关 Struts2 / XWork 的配置元素和容器的相关知识就成为了整个初始化主线运作的核心依赖。在第 9 章中，我们正是站在核心技术的基础之上，为读者总结出了 Struts2 初始化主线的核心驱动力以及构成 Struts2 初始化主线的核心元素和辅助元素。在了解所有这些元素的基本原理和互相之间的关系之后，我们将通过对核心分发器（Dispatcher）和容器（Container）这两大元素的初始化过程的分析，深刻揭示出初始化主线的全貌。

了解了 Struts2 的初始化主线，相当于我们做好了完备的战斗准备。因而在第 10 章中，我们正式打响与 Http 请求的战斗。在之前的分析中，我们已经把 Struts2 的 Http 请求处理主线划分为两个不同执行阶段，我们在第 10 章的讲解中，也将与 Http 请求的战斗的过程划分为两个不同的战场。不同战场的战斗，其所需要解决的问题各不相同，因而对于每一个执行阶段源码的分析，将引领我们解决在 Web 开发中所遇到的一个个核心问题，这也是本书的主要内容之一。

在完成对 Struts2 中两条运行主线的分析之后，我们把目光转向 Web 开发中与 Web 交互关系最为密切的视图部分，也就是 MVC 模式中的视图（View）层。视图层作为整个框架的“输入输出设备”，其重要性不言而喻。然而视图层却是整个 Web 框架中最富有变化的一个抽象层次，不仅因为视图技术的种类繁多，还因为视图层与 MVC 框架中其他层次的交互是每个表示层框架设计的要点。在第 11 章中，我们会针对视图层中涉及的多个哲学问题进行探讨。

在第 12 章中，我们将为读者介绍 Struts2 的扩展机制。事实上，在对 Struts2 运行主线的研究中，我们已经接触到了 Struts2 支持的所有扩展点。在这里，我们之所以单独用一章的篇幅进行讲解，主要是由于 Struts2 对程序的扩展机制进行了框架级别的抽象。我们所研究的运行主线，实际上只是一个很大的运行平台，Struts2 通过插件的形式可以对框架中的众多逻辑扩展点进行自由扩展。

本篇从内容上看有一些枯燥。因为在对 Struts2 整个过程的讲解中，我们将看到大量的源码解析。在这些对源代码的解析中，我们总是试图为大家进行流程的总结并配以示意图讲解。读者在阅读本篇时，可以以所有的示意图为蓝本，带着问题来分析源码。相信读完本篇，大家对 Web 框架运行机理的认识能够上一个新的台阶。

第 9 章　包罗万象——Struts2 初始化主线

Struts2 有两条运行主线，这两大主线无论从运行空间、运行特点以及所承担的职责上来看都完全不同。从运行空间上讲，两者没有任何交集，运行的触发条件也完全不同；从运行特点上讲，初始化主线在系统启动时运行一次，Http 请求处理主线则在系统启动完毕后以侦听请求的方式运行；从所承担的职责上讲，初始化主线是为整个 Struts2 建立起运行所必备的运行环境，而 Http 请求处理主线则是 Struts2 的核心功能。

本章的主要内容涉及初始化主线的来龙去脉。具体来说就是 Struts2 / XWork 如何对支撑起框架运行的基本元素进行的规划和管理。所谓包罗万象，我们首先应该弄清楚纷繁复杂的框架元素、初始化编程元素以及它们之间的关系；然后再考虑将这些元素融合在一起的过程。接下来，我们就来带领读者一步一步揭开其中的细节。

9.1　配置元素与初始化主线

9.1.1　从入口程序开始

在第 3 章中，我们就曾经提到过 Struts2 的两大主线的入口程序是相同的：StrutsPrepareAndExecuteFilter。只不过入口程序中的不同方法分别驱动了两条完全不同的运行主线。我们也正是基于 Filter 所实现的 Servlet 规范中不同方法的生命周期的不同，规划了 Struts2 不同的运行主线。其中，init 方法就是 Struts2 初始化主线的入口方法，其相关源码如代码清单 9-1 所示。

代码清单 9-1　StrutsPrepareAndExecuteFilter.java

```
/**
 * Http 请求的处理类，实现了 Servlet 标准中的 Filter
 */
public class StrutsPrepareAndExecuteFilter implements StrutsStatics,
Filter {

// 进行 Http 请求预处理的操作类
protected PrepareOperations prepare;

// 进行 Http 请求的逻辑执行处理类
```

```
protected ExecuteOperations execute;

// 配置哪些形式的URL模式被排除在Struts2的处理之外
protected List<Pattern> excludedPatterns = null;

    // Filter的初始化过程
    public void init(FilterConfig filterConfig) throws ServletException {
        // 初始化操作类
        InitOperations init = new InitOperations();
        try {
          、 FilterHostConfig config = new FilterHostConfig(filterConfig);
            init.initLogging(config);

            // 初始化核心分发器Dispatcher
            Dispatcher dispatcher = init.initDispatcher(config);

            // 初始化静态资源加载器
            init.initStaticContentLoader(config, dispatcher);

            // 初始化进行Http预处理的操作类
         prepare = new
PrepareOperations(filterConfig.getServletContext(), dispatcher);
            // 初始化进行Http请求处理的逻辑执行的操作类
         execute = new
ExecuteOperations(filterConfig.getServletContext(), dispatcher);
                  this.excludedPatterns =
init.buildExcludedPatternsList(dispatcher);
            // 自定义的初始化过程，留作用户扩展
            postInit(dispatcher, filterConfig);
        } finally {
            // 初始化中的清理工作
            init.cleanup();
        }

    }

    /**
     * 初始化中的扩展工作
     */
    protected void postInit(Dispatcher dispatcher, FilterConfig filterConfig) {
    }

    public void destroy() {
    // 清理核心分发器Dispatcher
        prepare.cleanupDispatcher();
    }

// 这里省略了许多其他代码
}
```

从源码上来看，init 方法的主要内容是针对三个元素展开的，而这三个元素也成为之后 Http 请求处理这一条逻辑主线的执行依据。这三个元素分别是：

- Dispatcher——核心分发器
- PrepareOperations——Http 预处理类
- ExecuteOperations——Http 处理执行类

其实，PrepareOperations 和 ExecuteOperations 只是 Struts2 在第二条运行主线中的执行句柄，其真正的意义仅仅在于两者对 Http 请求的处理职责不同。当然，职责的不同也直接成为 Struts2 对 Http 请求的处理流程进行逻辑划分的依据。有关这一点，我们在后续章节还会讲到。

而入口程序 StrutsPrepareAndExecuteFilter 的 init 方法的核心，则是通过 init.initDispatcher 方法的调用返回核心分发器 Dispatcher 的实例。该方法的相关源码如代码清单 9-2 所示。

代码清单 9-2　InitOperations.java

```
/**
 * 创建并初始化 dispatcher 对象
 */
public Dispatcher initDispatcher( HostConfig filterConfig ) {
    // 创建核心分发器 Dispatcher 的实例
Dispatcher dispatcher = createDispatcher(filterConfig);
// 调用 Dispatcher 实例的 init 方法完成核心分发器的初始化
    dispatcher.init();
    return dispatcher;
}

/**
 * 创建 Dispatcher 实例
 */
private Dispatcher createDispatcher( HostConfig filterConfig ) {
    Map<String, String> params = new HashMap<String, String>();
    for ( Iterator e = filterConfig.getInitParameterNames();
e.hasNext(); ){
        String name = (String) e.next();
        String value = filterConfig.getInitParameter(name);
        params.put(name, value);
    }
    return new Dispatcher(filterConfig.getServletContext(), params);
}
```

在这里，我们还是没有从代码中看出什么实质的内容。createDispatcher 方法将 filterConfig 中的初始化参数通过构造函数传入 Dispatcher 来创建一个新的

Dispatcher 实例并返回。其中，filterConfig 中所包含的参数其实来自于 web.xml 中的初始化参数配置。

要是读者以为这个过程只是像看上去那么简单，就大错特错了。因为在这段代码中，虽然 Dispatcher 的实例创建过程并不复杂，但其中 init 方法的内涵却是极其丰富的。这个 init 方法是核心分发器 Dispatcher 进行自身初始化的入口方法，同时也是整个 Struts2 初始化主线的入口方法，其内容将包含整个 Struts2 的初始化主线的全部过程，我们将在之后单独进行源码解析。

9.1.2 初始化主线的核心驱动力

任何程序的运行都有其内在的核心驱动力。所谓核心驱动力，其实是指某一段程序运行的最终目的是什么。因而，要弄清楚一段程序或者一条程序运行的主线，我们必须从逻辑和形式这两个不同的方面对核心驱动力进行解读：

- **核心驱动力的逻辑**——对于程序运行目的的描述
- **核心驱动力的形式**——推动程序运行的编程元素

对于 Struts2 的初始化主线而言，我们已经反复强调过核心驱动力的逻辑：对 Struts2 / XWork 元素的规划和管理。那么，作为核心驱动力的具体表现形式，到底是什么样的编程元素推动着初始化主线的运作呢？我们依然采用算法和数据结构这两个构成程序的抽象元素来进行描述：

- **数据结构**——框架的核心配置元素
- **算法**——围绕着核心配置元素的初始化过程

在第 3 章中，我们曾经提到过与框架的核心配置元素相关的一个重要观点，我们在这里不妨重新来审视一下：

结论 *框架的核心配置是一种贯穿始终的核心驱动力，它不仅能够以一定的形式表现出框架的构成元素互相之间的逻辑关系，同时能够将它们的执行逻辑串联起来。*

之前对于这个结论的理解，停留在了对框架元素的构成层面。结合初始化主线的算法逻辑，我们就会发现整个初始化主线所围绕的核心，不正是对框架的核心配置进行管理吗？事实上，在初步领略了 Struts2 所支持的所有配置表现形式（包括 XML 文件的形式，也有 Properties 文件的形式；既有系统级别的配置，也有应用级别的配置）之后，我们就不难从这些纷繁复杂的配置形式中总结出一条逻辑主线：

结论 *Struts2 初始化主线的核心驱动力，正是对各种配置形式所进行的一次统一的对象化处理。*

这个结论不仅是 Struts2 初始化过程的核心结论，也是贯穿所有 Struts2 配置元素运作方式的一条逻辑主线。有了这一结论，我们才可以把 Struts2 的配置元素的运行特性真正讲清楚。那么，什么是“对象化处理”呢，对配置元素进行“对象化处理”的最终目的又是什么呢？

因为配置元素本身无论从形式上还是内容的定义上，都有太多的不确定性。这些不确定性直接导致了框架在处理这些配置元素时，缺乏一个统一的处理模式。但是，如果换一个角度来思考这个问题，我们会发现无论哪种形式的配置文件，它们最终反映到框架运行之中，都表现为一个个 Java 对象或者运行参数。因此，“对象化处理”实际上说的是 **Struts2 在初始化的时候，将各种各样的配置元素，无论是 XML 形式还是 Properties 文件形式（甚至可能是其他自定义的配置形式）转化为 Struts2 所定义的 Java 对象或者 Struts2 运行时的参数的处理过程。**

由此，我们从逻辑上总结了 Struts2 初始化主线的核心驱动力。站在数据结构的角度，Struts2 初始化主线的操作对象正是这些配置元素；而站在算法的角度，我们还需要探究这些操作对象的逻辑调度元素。

在代码清单 9-1 中，我们已经看到了这一逻辑调度元素：**核心分发器**（Dispatcher）。对于核心分发器我们并不陌生，我们在本书的第二部分讲 XWork 的时候就曾经介绍过它。当时，我们还为核心分发器与 XWork 框架之间的关系单独做了一番描述：它们之间形成一个明显的调用关系。对于初始化主线而言，我们可以非常清楚地看到整个初始化的最终过程，被分派到了 Dispatcher 的 init 方法中去执行。所以 Dispatcher 对象的来龙去脉，也就成为了整个初始化主线的重中之重。

就初始化主线本身而言，它是一个非常复杂的过程。因而它离不开许多辅助对象的帮助。Dispatcher 对象作为一个核心元素，也需要这些辅助对象的帮助才能够顺利承担起核心分发器的重要职责。接下来，我们就来看看构成初始化主线的主要元素。

9.1.3　初始化主线的构成元素

本章的标题是“包罗万象”，这一标题实际上包含了两层意思：其一，在 Struts2 的初始化主线中需要处理的内置元素非常多，谓之“万象”；其二，Struts2 对于所有这些元素对象进行的管理，谓之“包罗”。

有关“万象”的话题，其实早在第 3 章中就已经给读者进行了介绍，并为这些对象找到了一个合适的切入点：**配置元素**。配置元素不仅表现形式纷繁复杂，其内在含义也各有不同。然而，有关“包罗”的话题，我们却未曾提及。这也可以理解，因为对于配置元素的处理过程正是 Struts2 初始化主线所要做的，也是本章的重点内容。

Struts2 在初始化主线中，为了对形式上纷繁复杂、内容含义又各异的配置元素进行处

理，不得不创建一些额外的辅助对象，这些对象实际上是对上述配置元素进行操作的流程元素。构成这些流程元素的对象，主要有三个不同的类别：

❑ 配置元素的加载器

配置元素的加载器主要用于将纷繁复杂的配置表现形式转化为框架元素，相当于在不同的配置形式和框架之间建立起沟通的桥梁。

❑ 配置元素的构造器

配置元素的构造器主要用于对框架元素进行初始化操作。在这里，Struts2 采用的是设计模式中的构造模式。

❑ 配置元素的管理类

配置元素的管理类主要是指在初始化主线运行的过程中，对配置元素的数据的存储和配置元素的初始化行为进行控制的配置管理元素。

这三大不同类别的元素都是构成 Struts2 初始化主线中的主要流程元素，结合上一节中提到的核心驱动力元素：核心分发器，我们可以发现整个 Struts2 在初始化主线的设计上考虑得也十分周到。在接下来的章节中，我们将分别对这些元素进行分析和讲解，最后将它们综合在一起，探寻它们的执行轨迹。

9.2 核心分发器 —— Dispatcher

9.2.1 核心分发器的核心驱动作用

Dispatcher 之所以被称为 Struts2 的核心分发器，主要是基于它在整个 Struts2 框架中的特殊地位。我们经常会使用“起—承—转—合”这 4 个不同的阶段来描述一个事件的整个过程，对于 Struts2 而言，Dispatcher 实际上就是囊括这 4 个阶段的核心分发器。

9.2.1.1 起 —— 负责系统初始化

Dispatcher 在初始化时负责整个 Struts2 的初始化工作。在 Dispatcher 中，init 方法会在 Dispatcher 创建之初被调用，从而触发整个 Struts2 的初始化过程。所有的初始化方法在 Dispatcher 中都以 init 加上下划线作为起始进行命名，如图 9-1 所示。

从图中，我们可以看到 init 方法是一个 public 方法，是整个初始化过程中的操作窗口。而其余的所有以 init 加上下划线命名的方法，都将在 init 方法中以一定的顺序被依次调用。这些方法涵盖了在 Dispatcher 进行初始化过程中逻辑的方方面面。我们将在之后的源码分析中为读者详细解读。

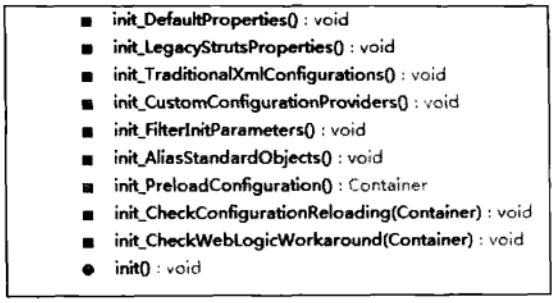


图 9-1　Dispatcher 的初始化方法列表

9.2.1.2　承——接收并预处理 Http 请求

Dispatcher 需要负责对 Http 请求进行预处理。这些预处理的过程主要包括：设置 Encoding 和 Locale、对 HttpServletRequest 进行封装以及准备 MVC 运行的数据环境等。

这些预处理过程，由 PrepareOperations 在进行 Http 请求预处理的这个阶段调用执行。而我们知道 PrepareOperations 对象本身只是一个操作代理接口，真正完成逻辑功能的是 Dispatcher 本身。PrepareOperations 在内部实现中所做的，只是对每个操作做了一个简单的转发处理。这些方法如图 9-2 所示。

图 9-2　Dispatcher 的 Http 请求预处理方法

注意在这其中的两个 createContextMap 方法。在这两个方法中，Dispatcher 把 Web 容器相关的数据对象封装成了普通的 Java 对象（实际上被封装成了 Map 对象）。这对于整个框架的意义不亚于任何其他的精妙设计。因为这两个方法的核心要义在于将 Web 请求数据进行“去容器化”处理，使得后续依赖于 Web 请求的任何操作不再受限于任何的 Web 容器对象，从而真正做到了解耦合。读者可以结合在之前 XWork 章节中，我们对 ActionContext 初始化的相关描述，createContextMap 方法实际上成为了 ActionContext 这个数据环境内部构造上下文环境的数据基础。

9.2.1.3　转——将 Struts2 转入 XWork

在第 3 章中，我们提到 Struts2 在处理 Http 请求时还可以分为 2 个阶段，而划分这 2 个阶段的临界点实际上就在于 Dispatcher 对象。

在第一个阶段中，所有的 Http 请求通过 Dispatcher 对象的 Http 请求的预处理，请求中与 Web 容器相关的对象全部被封装成与 Web 容器无关的对象，并构造出一个数据环

境。于是，在第二个阶段中 Dispatcher 就能够将这一数据环境转发到 XWork 框架中执行，从而保证解耦合在这一刻彻底完成。Dispatcher 中进行请求转发的方法是 serviceAction，如图 9-3 所示。

图 9-3 Dispatcher 的请求转发方法

我们可以看到，serviceAction 方法与 Servlet 规范中的 service 方法有着异曲同工之妙。只是在这里，serviceAction 方法并不是 Http 请求处理的“终点”，反而是 Struts2 / XWork 中进行 Http 请求处理的“起点”和“中转站”。

serviceAction 方法是整个 Dispatcher 对象的逻辑调度核心方法，也是整个 Struts2 在 Http 请求处理阶段的核心逻辑。我们将在下一章中对这个方法进行源码解读。

9.2.1.4 合——垃圾清理

Dispatcher 不仅负责逻辑执行，还要负责在执行完毕之后进行对象清理。这一工作是由 cleanup 方法完成的，如图 9-4 所示。

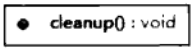


图 9-4 Dispatcher 的垃圾清理方法

在 Dispatcher 的 cleanup 方法中，将完成对 Http 请求的处理过程中产生的请求周期的对象的清理工作。在这里不再展开对其源码的分析，不过我们可以将其源码中所涉及的清理对象进行逻辑分类：

❑ 对于在整个请求周期中定义了完整的生命周期的框架元素的清理

例如，我们之前讲过的 XWork 中 Interceptor 对象，其接口定义中就蕴含了 destroy 方法。这个方法就会在 cleanup 的流程中被调用执行。

❑ 对于线程安全的 ThreadLocal 对象的清理

我们知道，XWork 中的 ActionContext 作为一个数据环境使用了 ThreadLocal 模式在整个当前线程共享数据，那么在整个请求结束时，与当前线程绑定的 ActionContext 的副本也在 cleanup 方法中被清理。

9.2.2 核心分发器的数据结构

Dispatcher 中的内容如此丰富，以至于它成为我们进行 Struts2 研究的最重要线索。之前我们已经看到了 Dispatcher 在不同阶段所承担的不同职责。接下来，我们从数据结构的角度来看看 Dispatcher 有什么独特之处。翻开 Dispatcher 的源码，我们可以看到

Dispatcher 内部的属性构成，源代码如代码清单 9-3 所示。

代码清单 9-3　Dispatcher.java

```
public class Dispatcher {

    // 这里省略了许多其他代码

    /**
     * 提供了一个静态的 ThreadLocal 变量
     */
    private static ThreadLocal<Dispatcher> instance = new
ThreadLocal<Dispatcher>();

    /**
     * 提供一个接口方法，用于获取当前线程安全的 Dispatcher 实例
     *
     * @return
     */
    public static Dispatcher getInstance() {
        return instance.get();
    }

    /**
     * 将 Dispatcher 实例绑定到当前线程
     *
     * @param instance
     */
    public static void setInstance(Dispatcher instance) {
        Dispatcher.instance.set(instance);
}
}
```

这个非常惹眼的内部变量 instance，它不仅是一个静态的实例变量，同时它也被定义成为 ThreadLocal 类型，而 ThreadLocal 所承载的实际类型是 Dispatcher 本身！这正是一个典型的 ThreadLocal 模式的应用范例！

从这个数据结构的定义来看，我们可以确认，Struts2 对 Dispatcher 应用了 ThreadLocal 模式。其中，Dispatcher 本身就是实现 ThreadLocal 模式的核心要素，因为我们实际上在 Dispatcher 中看到了实现 ThreadLocal 模式的两大步骤。在之后的源码分析中，我们还将看到 Dispatcher 在处理 Http 请求时与 ThreadLocal 变量的交互过程。

使用 ThreadLocal 模式的初衷是为了解决多线程环境下对象访问的线程安全问题。Dispatcher 作为核心分发器，成为入口程序 StrutsPrepareAndExecuteFilter 中的实例变量 PrepareOperations 和 ExecuteOperations 的实际操作句柄（Dispatcher 是它们的重要实例变量），从而被所有的线程共享访问。因此，Dispatcher 被打造成 ThreadLocal 模式是一个理

所当然的选择。

然而，这一理所当然的选择，却使得 Struts2 能够摆脱传统的表示层框架对 Web 容器的高度依赖。由此可见，ThreadLocal 模式是贯穿整个 Struts2 的核心技术之一，是 Struts2 将 Web 容器与 Java 开发解耦合的内在基础。不仅如此，我们还将在之后的章节中看到 ThreadLocal 模式在 Struts2 数据流转的过程中所起到的决定性作用。

最后，有两个针对 Dispatcher 的重要结论需要强调一下：

结论 作为一个线程安全的对象，Dispatcher 涵盖了 Struts2 的整个生命周期。无论是 Struts2 的初始化，还是处理 Http 请求，实际都在 Dispatcher 中完成。

这个结论所针对的角度是 Dispatcher 的内容和职责。将它牢记于心有助于日后我们对 Struts2 相关问题的调试和分析。实际上，对于 Struts2 的运行机理的一切研究，都离不开 Dispatcher 这个核心分发器。Dispatcher 中所包含的所有方法，从一个侧面反映了 Struts2 框架的几个主要功能，而 Dispatcher 则处于一个“居中调度”的重要位置。

结论 Dispatcher 是 Struts2 与 XWork 的分界点，也是将 MVC 实现与 Web 容器隔离的分界点。

这个结论从另外一个角度描述了 Dispatcher 在整个 Struts2 框架中的重要地位。它也成为 Struts2 实现“解耦合”的核心所在。我们将在对 Struts2 的第二条主线的分析中看到这个分界点以及它在框架中的逻辑意义。

9.3 配置元素的加载器（Provider）

9.3.1 配置元素加载器的作用

配置元素的加载器是 Struts2 初始化主线的重要组成元素。我们知道，Struts2 的配置元素的表现形式是纷繁复杂的。这些配置元素不仅表现形式完全不同，它们所表达的逻辑含义也不尽相同。为了能够兼容各种不同的配置形式，Struts2 定义了一个配置加载接口，来对不同形式的配置元素进行处理。

结论 配置加载接口的各种实现类（Provider）架起了各种配置表现形式到 Java 世界的桥梁。

这个结论非常容易理解，因为配置的加载方式本来就是起“翻译”作用的。通过“翻译”，框架可以透明地看待所有的配置形式。这就为框架根据配置元素的不同特点，进行

配置表现形式的设计提供了理论基础。

对于配置加载器，Struts2 定义了一个统一的操作接口 ConfigurationProvider，其源码如代码清单 9-4 所示。

代码清单 9-4　ConfigurationProvider.java

```
/**
 * 定义所有加载配置元素的接口
 */
public interface ConfigurationProvider extends ContainerProvider,
PackageProvider {
}
```

从源码上看，这里采用了 Java 中并不常使用的接口多重继承机制。这样做的好处在于，**它能够使 Struts2 在进行配置加载时拥有比较统一的操作路径和操作方法**。在这种情况下，我们大可不必关心具体的 Provider 实现到底针对的是哪种类型的配置元素，而将关注重点放在这些 Provider 所指定的接口方法的逻辑处理顺序上。

从 ConfigurationProvider 进行多重继承的两个接口 ContainerProvider 和 PackageProvider 来看，它们正好与我们在第 3 章中曾经提到过的配置元素的分类相对应。接下来，我们分别就这两个不同的配置加载接口进行分析。

9.3.2　容器加载器——ContainerProvider

我们首先来看看容器加载接口 ContainerProvider，其源码如代码清单 9-5 所示。

代码清单 9-5　ContainerProvider.java

```
public interface ContainerProvider {

    /**
     * 当 Container 从 configurationManager 中销毁时调用
     */
    public void destroy();

    /**
     * 在系统初始化时调用
     *
     * @param configuration
     * @throws ConfigurationException
     */
    public void init(Configuration configuration) throws
ConfigurationException;

    /**
```

```
     * 设置配置是否支持 reload 特性
     *
     */
    public boolean needsReload();

    /**
     * 在 Container 中注册 bean 和 properties
     *
     * @param builder
     * @param props
     * @throws ConfigurationException
     */
    public void register(ContainerBuilder builder, LocatableProperties
props) throws ConfigurationException;

}
```

这个接口从行为上定义了 Struts2 / XWork 是如何处理各种各样不同的配置形式并将它们转化为 Container 对象的。我们在这里不妨来看看 ContainerProvider 的实现类操作接口到底支持哪些配置加载方式，如图 9-5 所示。

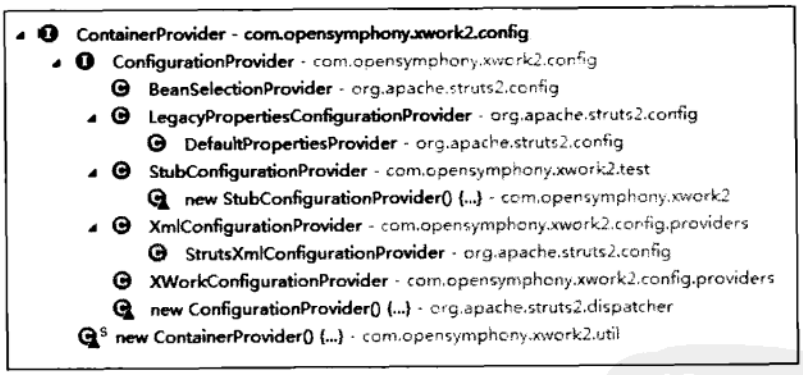


图 9-5 ContainerProvider 的实现类

从名称上我们就可以大致猜出这些实现类的用途，LegacyPropertiesConfigurationProvider 应该是用于加载 Properties 文件形式的配置，StrutsXmlConfigurationProvider 应该是用于加载 XML 文件形式的配置。

从 ContainerProvider 接口定义的方法签名所表现出来的含义来看，这些方法在 Struts2 / XWork 初始化过程中的调用时机和调用顺序也不尽相同。我们在这里有必要重点来看看和 Struts2 初始化相关的两个方法 init 和 register 的调用顺序，其相关源码如代码清单 9-6 所示。

代码清单 9-6　DefaultConfiguration.java

```
ContainerBuilder builder = new ContainerBuilder();
for (final ContainerProvider containerProvider : providers) {
    containerProvider.init(this);
    containerProvider.register(builder, props);
}
```

从这一调用顺序我们可以发现：**不同的 ContainerProvider 之间并没有依赖关系**。这一点告诉我们，所有的配置元素的加载器对于框架而言是一视同仁的。这个结论非常关键，因为这个结论实际上是我们对配置元素的形式进行扩展的理论依据。例如，如果要使用一种新的配置形式，我们只需要编写针对这种配置形式的 ContainerProvider 的实现类，并将它配置到 Struts2 / XWork 的容器中，就可以实现框架的自动识别。因为从代码上看，所有的 Container Provider 都会在这个过程中遍历一次，这也就是框架提供的隐含扩展点。

在整个初始化的过程中，我们只是看到了 ContainerProvider 的进行配置加载的生命周期：先进行初始化（调用 init 方法），再进行元素注册（调用 register 方法）。那么，这里的 register 到底做了些什么事呢？作为配置元素的加载器，其核心当然是围绕着当前加载器所操作的配置元素类型，而 ContainerProvider 所针对的配置元素类型，不正是容器（Container）吗？因此，register 在这里的语义，正如其注释上所言：**在容器中注册所有的 Bean 和 Properties**。有关容器的初始化过程，我们在后续章节中还有相关的源码解读。

9.3.3　事件映射加载器——PackageProvider

事件映射关系，是我们对 Struts2 / XWork 框架中的配置元素所划分的另外一个类别。针对事件映射关系，也有相应的配置加载接口的定义：PackageProvider，其源码如代码清单 9-7 所示。

代码清单 9-7　PackageProvider.java

```
public interface PackageProvider {

    /**
     * 初始化时调用
     *
     * @param configuration
     * @throws ConfigurationException
     */
    public void init(Configuration configuration) throws
```

```
ConfigurationException;

    /**
     * 设置是否支持 reload 特性
     *
     */
    public boolean needsReload();

    /**
     * 加载所有的 package 定义并创建 PackageConfig 对象
     *
     * @throws ConfigurationException
     */
    public void loadPackages() throws ConfigurationException;

}
```

与之前的 ContainerProvider 一样，作为一个配置元素加载器，PackageProvider 同样承担了将各种各样不同的配置形式转化为框架所定义的对象的职责。只不过在这里，PackageProvider 的操作对象在之前的章节中并没有详细介绍过，这个对象就是 PackageConfig。其源码如代码清单 9-8 所示。

代码清单 9-8 PackageConfig.java

```
public class PackageConfig extends Located implements Comparable,
Serializable, InterceptorLocator {

    private Map<String, ActionConfig> actionConfigs;
    private Map<String, ResultConfig> globalResultConfigs;
    private Map<String, Object> interceptorConfigs;
    private Map<String, ResultTypeConfig> resultTypeConfigs;
    private List<ExceptionMappingConfig> globalExceptionMappingConfigs;
    private List<PackageConfig> parents;
    private String defaultInterceptorRef;
    private String defaultActionRef;
    private String defaultResultType;
    private String defaultClassRef;
    private String name;
    private String namespace = "";
    private boolean isAbstract = false;
    private boolean needsRefresh;

    protected PackageConfig(String name) {
        this.name = name;
        actionConfigs = new LinkedHashMap<String, ActionConfig>();
        globalResultConfigs = new LinkedHashMap<String, ResultConfig>();
```

```
        interceptorConfigs = new LinkedHashMap<String, Object>();
        resultTypeConfigs = new LinkedHashMap<String, ResultTypeConfig>();
        globalExceptionMappingConfigs = new
ArrayList<ExceptionMappingConfig>();
        parents = new ArrayList<PackageConfig>();
    }

    protected PackageConfig(PackageConfig orig) {
        this.defaultInterceptorRef = orig.defaultInterceptorRef;
        this.defaultActionRef = orig.defaultActionRef;
        this.defaultResultType = orig.defaultResultType;
        this.defaultClassRef = orig.defaultClassRef;
        this.name = orig.name;
        this.namespace = orig.namespace;
        this.isAbstract = orig.isAbstract;
        this.needsRefresh = orig.needsRefresh;
        this.actionConfigs = new LinkedHashMap<String,
ActionConfig>(orig.actionConfigs);
        this.globalResultConfigs = new LinkedHashMap<String,
ResultConfig>(orig.globalResultConfigs);
        this.interceptorConfigs = new LinkedHashMap<String,
Object>(orig.interceptorConfigs);
        this.resultTypeConfigs = new LinkedHashMap<String,
ResultTypeConfig>(orig.resultTypeConfigs);
        this.globalExceptionMappingConfigs = new
ArrayList<ExceptionMappingConfig>(orig.globalExceptionMappingConfigs);
        this.parents = new ArrayList<PackageConfig>(orig.parents);
    }
    // 这里省略了所有的操作接口和 getter 方法
}
```

这里的 PackageConfig，正好与我们在第 3 章中所介绍的“事件映射关系”这种类型的配置元素相对应，也是 Struts2 初始化主线中对“事件映射关系”进行对象化处理的结果。也就是说，我们在 XML 配置文件中的 package 节点内部的所有内容都会在这里被“翻译”成一个个 PackageConfig 对象供 Struts2 运行时使用。

在这里，值得我们注意的是 PackageConfig 的构造函数。**PackageConfig 所提供的两个构造函数都被定义成了 protected**。也就是说，我们无法通过正常的构造函数的途径直接构造一个 PackageConfig 对象，那么 PackageConfig 在 Struts2 内部是如何被初始化的呢？Struts2 提供了与 PackageConfig 所对应的特殊的构造器，专门用于构造 PackageConfig 对象。在下一节中，我们还会接触到这个构造器，并给出更加详细的分析。

如果从操作的角度来看 PackageConfig 这个类，其中也直接暴露了针对所有节点的操作接口。这些操作接口的定义如图 9-6 所示。

图 9-6 PackageConfig 中的操作接口

其中，绝大多数的操作接口是获取 PackageConfig 内部的属性值。因而 PackageConfig 作为与配置元素对应的实际载体，从数据结构的角度为 Struts2 的初始化工作打下了理论基础。

9.4 配置元素的构造器（Builder）

在第 3 章中我们曾经提到，Struts2 总是将初始化划分为一条重要的逻辑主线，而运行期的程序控制作为另外一条逻辑主线。这是 Struts2 的核心设计理念之一，这种设计理念的存在基础在于 Struts2 总是试图做到元素之间的解耦合。

基于这种设计理念，对于“配置元素的对象化”这样一个小模块，Struts2 同样使用了初始化和运行期控制严格分开的设计思想。具体来说，Struts2 / XWork 针对不同的配置元素定义了专门与之对应的构造器，用于构造 Container 对象。也就是说，Struts2 / XWork 在所有的配置元素的构建上，使用了构造（Builder）模式。

Struts2 / XWork 所引入的“构造器”的概念，与传统的使用构造函数的方式来创建对象的方式可谓是大相径庭。我们在本书的第 4 章有关设计模式的讲解中曾经提到过，构造器自身是一系列操作方法的集合类。这些操作方法主要可以分成两种不同的类型，分别对应于构造器进行对象构造的两个步骤：**参数搜集操作**和**创建对象操作**。当我们需要构造一个对象时，总是首先调用那些参数搜集的操作方法，每次调用，构造器会在其内部缓存这些参数，并在最后调用创建对象的操作一次性把对象创建出来。

Struts2 为什么要引入“构造器”而并非使用传统的构造函数方式来构建对象呢？在我们仔细分析 PackageConfig 和容器的构造器的内部原理之后，相信这一问题就能迎刃而解。

9.4.1　容器构造器——ContainerBuilder

在分析容器的加载方式时，其实我们已经与这个构造器有过一面之缘。在 ContainerProvider 的 register 方法中，我们就已经见到过针对容器的构造器 ContainerBuilder。它作为参数被传入 register 方法中，负责初始化过程中容器的初始化工作。

我们首先来看看 ContainerBuilder 的定义。在 Struts2 中，ContainerBuilder 被定义为一个接口，我们首先不探究其源码，来看看 ContainerBuilder 中到底定义了哪些操作接口，如图 9-7 所示。

由于容器是一个接口，所以 ContainerBuilder 所构造的实际对象是 Container 这个接口所对应的实现类 ContainerImpl。正如我们在第 5 章中谈到的，我们其实并不真正关心容器的内部数据结构，因为从容器自身的特点来说，容器的重点在于对其所管理的对象进行生命周期的管理和依赖关系的管理。从这个角度来讲，ContainerBuilder 在进行容器的初始化时，其核心语义在于通过一定的操作，首先将容器所需要管理的对象搜集起来，然后再通过一个创建方法把容器确实有效地创建出来。这也就是 ContainerBuilder 被设计成为一个接口的原因，因为 ContainerBuilder 同样不关心容器的数据结构。很显然，这也符合构造模式的基本过程。

从 ContainerBuilder 所表达的逻辑语义来看，它的设计恰恰符合构造模式中对于构造器的两大要求：

- **搜集参数**——factory 方法、alias 方法和 constant 方法
- **构造对象**——create 方法

到此为止，即使撇开 ContainerBuilder 的源码，我们也已经可以想象在 Struts2 内部的调用过程了：首先创建一个 ContainerBuilder 的实例，接着调用 factory 方法和 constant 方法等搜集各种配置元素，最后调用 create 方法把容器创建出来。这整个过程，**与构造模式所定义的执行过程是完全一致的**。

图 9-7 ContainerBuilder 的操作接口

9.4.2 事件映射构造器 —— PackageConfig.Builder

事件映射构造器则呈现出与容器构造器完全不同的景象。身为事件映射构造器的 PackageConfig.Builder，它不仅是一个内部类，还是一个具体的实现操作类而非一个接口。这就与我们之前谈到的 ContainerBuilder 大相径庭。

让我们首先来对这两个构造器之间的不同加以分析：由于容器是一个接口，这直接导致了 ContainerBuilder 在进行容器初始化时，并不关心容器自身的数据结构，而

是强调容器中元素的搜集过程，这也就是 ContainerBuilder 被定义成接口的原因。而 PackageConfig.Builder 的操作对象则完全不同，PackageConfig 自身是一个与 XML 配置元素节点一一对应的具体的数据结构，因而 PackageConfig.Builder 不得不去关心 PackageConfig 的数据结构。根据我们之前对 PackageConfig 的了解，PackageConfig 的所有构造函数都被设置成 protected，从而使得针对 PackageConfig 的构造器也不得不定义成一个内部类。因为在 package 级别之外，我们甚至无法构建 PackageConfig 的实例。

PackageConfig.Builder 的源码，如代码清单 9-9 所示。

代码清单 9-9　PackageConfig.Builder.java

```
public static class Builder implements InterceptorLocator {

        private PackageConfig target;

        public Builder(String name) {
            target = new PackageConfig(name);
        }

        public Builder(PackageConfig config) {
            target = new PackageConfig(config);
        }

        public Builder name(String name) {
            target.name = name;
            return this;
        }

        public Builder isAbstract(boolean isAbstract) {
            target.isAbstract = isAbstract;
            return this;
        }

        public Builder defaultInterceptorRef(String name) {
            target.defaultInterceptorRef = name;
            return this;
        }

        public Builder defaultActionRef(String name) {
            target.defaultActionRef = name;
            return this;
        }

        public Builder defaultClassRef( String defaultClassRef ) {
            target.defaultClassRef = defaultClassRef;
            return this;
        }
```

```
        /**
         * 设置默认的Result类型
         *
         * @param defaultResultType
         */
        public Builder defaultResultType(String defaultResultType) {
            target.defaultResultType = defaultResultType;
            return this;
        }

        public Builder namespace(String namespace) {
            if (namespace == null) {
                target.namespace = "";
            } else {
                target.namespace = namespace;
            }
            return this;
        }

        public Builder needsRefresh(boolean needsRefresh) {
            target.needsRefresh = needsRefresh;
            return this;
        }

        public Builder addActionConfig(String name, ActionConfig action) {
            target.actionConfigs.put(name, action);
            return this;
        }

        public Builder addParents(List<PackageConfig> parents) {
            for (PackageConfig config : parents) {
                addParent(config);
            }
            return this;
        }

        public Builder addGlobalResultConfig(ResultConfig resultConfig) {
            target.globalResultConfigs.put(resultConfig.getName(), resultConfig);
            return this;
        }

        public Builder addGlobalResultConfigs(Map<String, ResultConfig>
resultConfigs) {
            target.globalResultConfigs.putAll(resultConfigs);
            return this;
        }

        public Builder addExceptionMappingConfig(ExceptionMappingConfig
```

```
exceptionMappingConfig) {
                target.globalExceptionMappingConfigs.add(exceptionMappingConfig);
                return this;
            }

            public Builder
    addGlobalExceptionMappingConfigs(List<ExceptionMappingConfig>
    exceptionMappingConfigs) {
    target.globalExceptionMappingConfigs.addAll(exceptionMappingConfigs);
                return this;
            }

            public Builder addInterceptorConfig(InterceptorConfig config) {
                target.interceptorConfigs.put(config.getName(), config);
                return this;
            }

            public Builder addInterceptorStackConfig(InterceptorStackConfig config) {
                target.interceptorConfigs.put(config.getName(), config);
                return this;
            }

            public Builder addParent(PackageConfig parent) {
                if (this.equals(parent)) {
                    LOG.error("A package cannot extend itself: " + target.name);
                }

                target.parents.add(0, parent);
                return this;
            }

            public Builder addResultTypeConfig(ResultTypeConfig config) {
                target.resultTypeConfigs.put(config.getName(), config);
                return this;
            }

            public Builder location(Location loc) {
                target.location = loc;
                return this;
            }

            public boolean isNeedsRefresh() {
                return target.needsRefresh;
            }

            public String getDefaultClassRef() {
                return target.defaultClassRef;
            }
```

```
        public String getName() {
            return target.name;
        }

        public String getNamespace() {
            return target.namespace;
        }

        public String getFullDefaultResultType() {
            return target.getFullDefaultResultType();
        }

        public ResultTypeConfig getResultType(String type) {
            return target.getAllResultTypeConfigs().get(type);
        }

        public Object getInterceptorConfig(String name) {
            return target.getAllInterceptorConfigs().get(name);
        }

        // 对于每个 PackageConfig 中的属性，在构造时使用了 Collections 的不可变处理
        public PackageConfig build() {
            target.actionConfigs =
Collections.unmodifiableMap(target.actionConfigs);
            target.globalResultConfigs =
Collections.unmodifiableMap(target.globalResultConfigs);
            target.interceptorConfigs =
Collections.unmodifiableMap(target.interceptorConfigs);
            target.resultTypeConfigs =
Collections.unmodifiableMap(target.resultTypeConfigs);
            target.globalExceptionMappingConfigs =
Collections.unmodifiableList(target.globalExceptionMappingConfigs);
            target.parents = Collections.unmodifiableList(target.parents);

            PackageConfig result = target;
            target = new PackageConfig(result);
            return result;
        }

        @Override
        public String toString() {
            return "[BUILDER] "+target.toString();
        }
    }
```

在上面的代码中，我们可以清楚地看到 PackageConfig 的构造步骤：首先构造一个新的“构造器”的实例，接着往构造器中设置参数，最后调用 build 方法完成对象创建。也就是说，虽然 PackageConfig.Builder 被设计成一个具体的实现类，但是其构造步骤还是符

合构造模式的基本调用步骤。

如此一来，我们就可以从上面的源码中，总结出一些 PackageConfig.Builder 作为“构造器”的特点：

- Builder 构造器中绝大多数操作的返回值都是这个构造器本身，这有助于程序员在构造 PackageConfig 时使用链式操作的方式进行连续调用。
- 外部程序构造 PackageConfig 的唯一途径是使用这个 PackageConfig.Builder 构造器，因为 PackageConfig 自身的构造函数被定义成了 protected。
- Builder 构造器的核心方法 build 在搜集属性值并创建真正的 PackageConfig 对象时，调用 Collections 中的 unmodifiableList 方法进行**不可变处理**。

上述后两个特点，一个体现了 PackageConfig 的初始化工作具有非常严格的**隔离性**，另外一个体现了被创建出来的 PackageConfig 的**稳定性**。这两点对于一个框架级别的配置元素而言，都是极为重要的。

9.5　配置元素的管理类

之前谈到的初始化主线的三大元素：核心分发器（Dispatcher）、配置元素的加载器（Provider）以及配置元素的构造器（Builder），是初始化主线的三大核心要素。这三大元素实际上从三个不同的角度对初始化元素进行了诠释：

- **初始化主线的核心驱动力**。
- **初始化主线的操作载体**。
- **初始化主线中元素的构建方式**。

这三个不同的角度都是围绕着初始化主线的操作对象（配置元素）在做文章。在整个初始化主线中还有一些辅助元素，这些元素的作用是将上述这三大元素串联起来，控制它们的执行顺序。这些辅助元素与核心要素之间的关系，有点像 XWork 中的 Action、Interceptor、Result 这些事件处理节点和 ActionProxy、ActionInvocation 这样的事件处理驱动元素之间的关系。

在这些辅助元素中，ConfigurationManager 和 Configuration 无疑是其中最重要的两个元素。我们在这里同样为 ConfigurationManager 和 Configuration 定义了一个名称：**初始化驱动元素**。

ConfigurationManager 和 Configuration 这两个元素之间的关系，又好比是 XWork 中 ActionProxy 和 ActionInvocation 之间的关系。其中，**ConfigurationManager 是整个配置元素进行操作的代理接口类，而真正对所有配置元素的初始化进行调度的是 Configuration 对象**。在接下来的章节中，我们就来重点讨论一下这两个辅助元素。

9.5.1 配置管理元素 —— Configuration

9.5.1.1 Configuration 的职责概述

作为初始化驱动元素，其主要的目的就是为了对初始化主线中的配置元素进行管理。这一管理工作，Struts2 / XWork 通过 Configuration 来完成。因此，Configuration 对象也被我们称为**配置管理元素**。

如果从框架的职责角度来进行分析，对配置元素进行管理实际上蕴含了两个不同的方面：

- **提供框架级别对所有配置元素的访问操作接口。**
- **对所有的配置元素进行初始化调度。**

从这两个方面来讲，我们对 Configuration 的期望实际上转化为一系列的操作接口。因而，Configuration 在 Struts2 中也被定义成一个接口。而在这个接口中，也同时包含了两类不同的操作方法，分别对应于上述对配置元素管理的两个不同方面。

我们首先来看看 Configuration 中所提供的对所有配置元素的访问接口，其相关源码如代码清单 9-10 所示。

代码清单 9-10 Configuration.java

```
public interface Configuration extends Serializable {

    PackageConfig getPackageConfig(String name);

    Set<String> getPackageConfigNames();

    Map<String, PackageConfig> getPackageConfigs();

    RuntimeConfiguration getRuntimeConfiguration();

    void addPackageConfig(String name, PackageConfig packageConfig);

    PackageConfig removePackageConfig(String packageName);

    Container getContainer();

    Set<String> getLoadedFileNames();

    List<UnknownHandlerConfig> getUnknownHandlerStack();

    void setUnknownHandlerStack(List<UnknownHandlerConfig> unknownHandlerStack);
}
```

从源码中我们可以看到，Configuration 对配置元素的访问操作接口恰巧反映出 Struts2

配置元素的两种不同的分类：容器（Container）和事件映射对象（PackageConfig）。换句话说，在 Configuration 实现类的内部，应该持有对这些配置元素的缓存实例。否则，那些对配置元素的操作接口在这里也就失去了意义。不过这一点目前来说只是我们对 Configuration 实现的猜测，之后在对 Configuration 实现类的源码进行解析时，我们会来证实这一点。

不过在这些访问操作接口中，我们并没有看到配置元素加载器和配置元素构造器的访问接口；相反的，其中倒是暴露了一些直接对配置元素的操作接口。这在一定程度上给予我们对 Configuration 对象的一个定位上的提示：**Configuration 的操作对象是配置元素**。

Configuration 对象对配置元素管理的另一个方面是对所有的配置元素进行初始化调度，其相关源码如代码清单 9-11 所示。

代码清单 9-11　Configuration.java

```
public interface Configuration extends Serializable {

    void rebuildRuntimeConfiguration();

    void destroy();

    /**
     * @deprecated Since 2.1
     * @param providers
     * @throws ConfigurationException
     */
    @Deprecated void reload(List<ConfigurationProvider> providers) throws
ConfigurationException;

    /**
     * @since 2.1
     * @param containerProviders
     * @throws ConfigurationException
     */
    List<PackageProvider> reloadContainer(List<ContainerProvider>
containerProviders) throws ConfigurationException;

}
```

这些操作接口在命名上也非常直观。不过在这里有一点非常值得我们注意，那就是 reloadContainer 方法的参数是一组配置元素的加载器（ConfigurationProvider）。这就从一个侧面反映出之前我们的一个重要观点：**Configuration 对象对于配置元素的调度本质，是对框架配置元素的操作**。这实际上也是 Configuration 对象将内容和操作进行分离的一种设计思路。

9.5.1.2 Configuration 的实现

我们在上一节中留下了一个问题：Configuration 实现类的内部，应该持有对这些配置元素的缓存实例。这个问题实际上是整个 Configuration 对象的实现核心的一个重要结论，我们可以用源码来进行证明。

Configuration 的默认实现类是 DefaultConfiguration，其相关源码如代码清单 9-12 所示。

代码清单 9-12 DefaultConfiguration.java

```
public class DefaultConfiguration implements Configuration {


    // 配置
    protected Map<String, PackageConfig> packageContexts = new
LinkedHashMap<String, PackageConfig>();

    protected RuntimeConfiguration runtimeConfiguration;

    protected Container container;

    protected String defaultFrameworkBeanName;

    protected Set<String> loadedFileNames = new TreeSet<String>();

    protected List<UnknownHandlerConfig> unknownHandlerStack;

    ObjectFactory objectFactory;

    public DefaultConfiguration() {
        this("xwork");
    }

    public DefaultConfiguration(String defaultBeanName) {
        this.defaultFrameworkBeanName = defaultBeanName;
    }
}
```

原来，**所有的配置元素都被完整地封装在 DefaultConfiguration 之中，成为 Configuration 进行配置元素管理的数据基础**。我们可以看到，所有 Configuration 的操作接口中对配置元素的访问操作接口，都是通过操作其内部的缓存对象来完成的。这一点，也是 Configuration 成为配置管理元素的佐证。

而整个 DefaultConfiguration 中对所有配置元素的核心调度接口，是 reloadContainer 方法。这个方法将在整个初始化主线的运行过程中被调用，并成为 Struts2 / XWork 对容器进行初始化的逻辑核心。鉴于这个方法是整个初始化主线的核心内容，我们将在之后的章

节中单独开辟一节对其进行分析。

9.5.2　配置操作接口 —— ConfigurationManager

在了解了 Configuration 的来龙去脉后，ConfigurationManager 就比较简单了。因为 ConfigurationManager 在整个初始化主线中所起到的作用，就如同 ActionProxy 在整个 XWork 框架中所起到的作用：**配置元素的操作接口**。

为了说明 ConfigurationManager 的配置操作接口作用，我们来看看 ConfigurationManager 的数据结构，如代码清单 9-13 所示。

代码清单 9-13　ConfigurationManager.java

```
public class ConfigurationManager {

// 持有对 Configuration 对象的操作句柄
    protected Configuration configuration;

// 内部封装了所有的容器加载器 ContainerProvider
    private List<ContainerProvider> containerProviders = new
CopyOnWriteArrayList<ContainerProvider>();

// 内部封装了所有的事件映射加载器 PackageProvider
    private List<PackageProvider> packageProviders = new
CopyOnWriteArrayList<PackageProvider>();

// 默认的配置读取策略名称
    protected String defaultFrameworkBeanName;

    public ConfigurationManager() {
        this("xwork");
    }

    public ConfigurationManager(String name) {
        this.defaultFrameworkBeanName = name;
    }
}
```

从源码中，我们可以看到之前所介绍的配置元素的两类加载器：容器加载器（ContainerProvider）和事件映射加载器（PackageProvider）都封装于 ConfigurationManager 的内部，成为其进行配置初始化的操作句柄。

如果之前我们对 Configuration 的调度接口 reloadContainer 还有所疑问，那么看到这里疑问就应该可以消除了。原来，Configuration 实例与配置元素加载器（ConfigurationProvider）被作为平行的元素封装于 ConfigurationManager 对象的内部。这

样一来，**我们就可以在数据结构上，对真正的配置元素与配置元素加载器进行严格的分离，使得 Configuration 对象和 ConfigurationManager 对象的职责划分更为清晰。**

作为一个配置操作接口，ConfigurationManager 将元素的初始化和元素的逻辑调度整合在了一起。这一点，我们可以从其核心的 getConfiguration 方法中得到印证，其相关源码如代码清单 9-14 所示。

代码清单 9-14 ConfigurationManager.java

```
public synchronized Configuration getConfiguration() {
        if (configuration == null) {
            // 如果 configuration 对象不存在，创建一个新的
            setConfiguration(new
DefaultConfiguration(defaultFrameworkBeanName));
            try {
                // 调用 configuration 中的 reloadContainer 对象初始化 Container
                configuration.reloadContainer(getContainerProviders());
            } catch (ConfigurationException e) {
                setConfiguration(null);
                throw new ConfigurationException("Unable to load
configuration.", e);
            }
        } else {
            conditionalReload();
        }

        return configuration;
}
```

从 getConfiguration 的源码实现中我们看到，当 ConfigurationManager 内部的 Configuration 对象不存在时，getConfiguration 会首先创建一个 DefaultConfiguration 的实例，并且调用其核心方法 reloadContainer 完成真正的 Configuration 的初始化工作。

因而我们可以看到，Configuration 的初始化和 Configuration 的核心调度被合而为一。这样做的目的在于屏蔽上层对 ConfigurationManager 对象内部实现的种种细节，从而简化 ConfigurationManager 操作接口的语义。

值得我们注意的是，正是这种合而为一的设计使得 getConfiguration 方法反而成为 Struts2 / XWork 进行初始化主线的一个核心调用。因为在其内部，虽然它的核心目的是完成初始化，但它的核心调度功能却真正完成了对容器的初始化。

9.6 Struts2 初始化主线详解

之前我们介绍了所有在 Struts2 初始化主线中所涉及的元素。接下来，我们将这些元

素串联起来，详细解读这些元素在初始化主线中的执行轨迹。

在整个执行轨迹中，有两个元素的初始化过程是我们重点关注的对象。这两个元素分别是：核心分发器以及容器（Container）。

9.6.1　核心分发器的初始化

在本章的一开始，我们就给出了 Struts2 初始化主线的入口程序。在对入口程序的分析中，我们已经指出整个初始化主线唯一的核心驱动力是对核心分发器的初始化过程。因而，我们在入口程序的 init 方法中看到的 Dispatcher 本身作为一个对象实例的创建过程并不复杂，但是 Dispatcher 对象的初始化过程，也就是 init 方法的逻辑处理才是整个初始化主线的逻辑调用核心。其源码如代码清单 9-15 所示。

代码清单 9-15　Dispatcher.java

```
/**
 * Dispatcher 的初始化方法
 */
public void init() {

    // ==== 步骤 1 开始 ====
    // 初始化 configurationManager
    if (configurationManager == null) {
                configurationManager = new
ConfigurationManager(BeanSelectionProvider.DEFAULT_BEAN_NAME);
        }
        // ==== 步骤 1 结束 ====

    try {
        // ==== 步骤 2 开始 ====
        // 初始化各种形式的配置加载方式
        // 这里只是定义了配置加载的方式，并没有真正执行加载配置的逻辑
        init_DefaultProperties(); // [1]
        init_TraditionalXmlConfigurations(); // [2]
        init_LegacyStrutsProperties(); // [3]
        init_CustomConfigurationProviders(); // [4]
        init_FilterInitParameters() ; // [5]
        init_AliasStandardObjects() ; // [6]
        // ==== 步骤 2 结束 ====

        // ==== 步骤 3 开始 ====
        // 初始化容器，这里包含两个不同的过程
        // 1. 对所有定义的配置加载方式的逻辑执行，将它们初始化到 Configuration 中
        // 2. 初始化 Struts2 中的对象管理容器
        Container container = init_PreloadConfiguration();
        // 对 Container 进行依赖注入
```

```
            // 所有在 Struts2 中定义的对象的生命周期在此操作后都被纳入容器管理
            container.inject(this);
            // ==== 步骤 3 结束 ====

            // ==== 步骤 4 开始 ====
            // 额外初始化，检查是否支持 reloading 加载和对 weblogic 服务器的特殊设置
            init_CheckConfigurationReloading(container);
            init_CheckWebLogicWorkaround(container);

            if (!dispatcherListeners.isEmpty()) {
                // 对所有定义的 dispatcherListener 执行自定义的初始化逻辑
                for (DispatcherListener l : dispatcherListeners) {
                    l.dispatcherInitialized(this);
                }
            }
            // ==== 步骤 4 结束 ====

        } catch (Exception ex) {
            if (LOG.isErrorEnabled())
                LOG.error("Dispatcher initialization failed", ex);
                throw new StrutsException(ex);
            }
        }
    }
```

洋洋洒洒的初始化代码，看起来似乎很难。在此，我们特地将整个过程用流程示意图的方式画出来供读者进行参考，如图 9-8 所示。

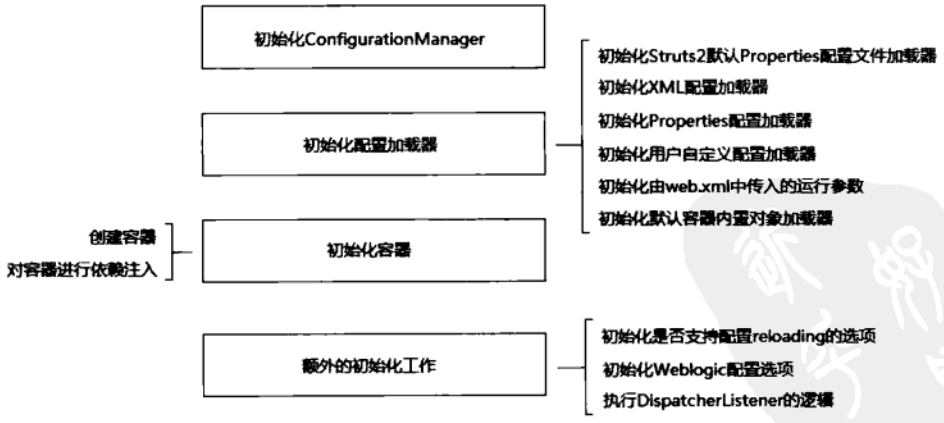


图 9-8 核心分发器 Dispatcher 的初始化过程

画出这张流程示意图的目的在于帮助读者理解 init 方法的调用步骤。而整个过程，我

们也已经在源代码级别加以注释。我们可以将流程示意图和源代码的注释结合起来，从而了解整个 init 方法的逻辑步骤及其意义。

整个方法自上而下可以分成 4 个步骤，这 4 个步骤一环紧扣着一环，后者以前者为基础，每个步骤都有其特定的逻辑含义和作用。

9.6.1.1　步骤 1——创建 ConfigurationManager

初始化的第一步是创建 ConfigurationManager。我们曾经在之前的章节中提到过 ConfigurationManager 这个类并且深入分析过其内部的数据结构。我们对 ConfigurationManager 的定义，**是一个配置操作接口，也是之后对所有配置元素进行的操作调度句柄。**

由此，ConfigurationManager 在 Dispatcher 进行初始化时，作为第一个步骤首先被创建出来。我们可以在其中看到 ConfigurationManager 在初始化时依赖一个特定的名称，而这个名称也是日后 Configuration 对象在整个框架的配置结构中进行对象寻址的关键要素。如果结合之前在第 3 章中所提到的配置元素来考虑，这个名称实际上就是配置元素中 <bean> 节点的 name 属性。XWork 的配置通过 name 属性来支持一个接口的多种实现方式共存。对于框架而言，根据 name 进行寻址就变成了一个重要的过程。

9.6.1.2　步骤 2 —— 初始化各种配置加载器

Dispatcher 进行初始化的第二步，主要是完成对配置元素的加载器的初始化工作。在源码中，我们分别使用了 [1] ～ [6] 的数字对不同配置元素加载器进行的初始化过程加以标注（请参考代码清单 9-15 的注释部分）。当结合 ConfigurationManager 的数据结构一起来看 Dispatcher 初始化的这一步骤时，我们就可以发现，这些步骤不过是在初始化 ConfigurationManager 这个配置行为管理类的内部，对不同配置元素加载接口：ContainerProvider 以及 PackageProvider 的实现类进行初始化和缓存。因而，ConfigurationManager 准确地说是将不同类型的配置元素的加载接口进行区分的场所。

通过之前的分析我们知道，配置元素加载器（ContainerProvider 或者 PackageProvider）的作用是将纷繁复杂的配置表现形式转化成框架可识别的配置数据。因而在这些初始化序列中，我们可以结合 Struts2 所支持的多种配置形式来理解这几个方法的实际作用，如表 9-1 所示。

表 9-1　Struts2 所支持的配置形式和 ConfigurationProvider 的对应关系

初始化方法调用	对应的配置加载实现类	作用描述
init_DefaultProperties	DefaultPropertiesProvider	初始化 Struts2 默认的 Properties 文件，该文件是 struts 分发 org/apache/struts2/default 目录下的 default.properties 文件

（续）

初始化方法调用	对应的配置加载实现类	作用描述
init_TraditionalXmlConfigurations	StrutsXmlConfigurationProvider	初始化Struts2的XML文件配置加载器
init_LegacyStrutsProperties	LegacyPropertiesConfigurationProvider	初始化Struts2的Properties文件配置加载器
init_CustomConfigurationProviders	用户自定义配置加载实现类	初始化用户自定义的配置加载器
init_FilterInitParameters	直接实现配置加载器接口	初始化由web.xml传入的运行参数
init_AliasStandardObjects	BeanSelectionProvider	初始化默认容器内置对象

这些配置加载器的具体实现核心，是一个针对不同数据格式的配置元素的读取过程。我们可以看到，既有XML形式也有Properties文件形式，甚至我们还能在其中看到硬编码形式的配置加载过程。在后续的章节中，我们可能会涉及其中几个配置元素加载器的内部实现机理的研究。不过，读者在这里应该明确的是配置元素加载器这一概念在整个初始化过程中的作用，而不是去深究读取XML文件或者Properties文件的细节。

从源码中，我们还可以看到Struts2在这里使用了策略（Strategy）模式。由于配置的形式是多种多样的，这就导致了读取配置的“算法”也各不相同。尽管如此，我们的目的在于拿到这些不同“算法”读取配置的结果，并将它们进行统一化处理，这就自然而然成为策略模式的一个典型的扩展应用。在这里，无论有多少种策略实现，只要它们的行为特征是一样的，我们就可以从中获取策略实现的结果并加以处理。这种应用方式的最大特点就在于：**策略实现可以任意进行灵活的扩展，而对客户端调用者而言则是透明的。**

不仅如此，我们在之前的分析中还讲到，配置元素的加载器有一个统一的接口ConfigurationProvider，这个接口采取的是接口的多重继承机制。这样对两种不同配置元素的加载方式做了进一步的抽象和统一。这可以说又是对策略模式的一个加强。

值得注意的是，这些配置加载器的初始化工作只是首先在ConfigurationManager的内部“占了个位子”，并没有进行这些配置加载器的真正逻辑调用。这些配置加载器的逻辑调用过程，将在步骤3的“初始化容器”中完成。这也从一个侧面体现出Struts2在设计时始终遵循着“初始化”和“行为逻辑”分离的最佳实践。

9.6.1.3 步骤3——初始化容器

在完成了各种配置加载器的初始化工作之后，Struts2用2行代码完成了核心容器的初始化。这一核心容器正是我们在第5章中详细介绍过的XWork中对对象的生命周期进行管理的核心组件。

这 2 行代码看上去简单，实际上却包含了 Struts2 中最重要的基础构建的初始化过程。虽然还没有揭晓其中的实现细节，但是我们却可以从代码本身看出一些容器操作的影子来：第一行代码中的 init_PreloadConfiguration 方法必然是负责创建容器，其中包含了对容器的“构造器”的逻辑调用（参数搜集和对象构造）；而第二行代码，则是对刚刚创建出来的容器进行依赖注入，从而完成对整个容器的生命周期和依赖关系的管理。

这 2 行代码是 Struts2 初始化主线的重中之重，我们将独立开辟一节剖析其调用过程中的种种细节。

9.6.1.4　步骤 4——执行额外的初始化工作

在完成主要工作之后，Struts2 还做了一些额外的初始化工作，比如配置是否支持 reloading 的特性，为 Weblogic 服务器单独设置某些系统级别的运行参数。

除此之外，Struts2 允许对 Dispatcher 的行为进行自定义扩展。而这一扩展则可以通过实现 DispatcherListener 接口来完成。其源码如代码清单 9-16 所示。

代码清单 9-16　DispatcherListener.java

```
public interface DispatcherListener {

    /**
     * 在 Dispatcher 初始化时调用
     *
     * @param du dispatcher 实例
     */
    public void dispatcherInitialized(Dispatcher du);

    /**
     * 在 Dispatcher 销毁时调用
     *
     * @param du dispatcher 实例
     */
    public void dispatcherDestroyed(Dispatcher du);
}
```

通过源码我们可以看到 Struts2 允许在初始化和销毁这两个事件的执行节点上执行自定义的逻辑。我们可以通过 Dispatcher 中的静态方法将自定义逻辑插入到 Struts2 的初始化或者销毁的逻辑执行中，Dispatcher 中的静态操作方法的源码如代码清单 9-17 所示。

代码清单 9-17　Dispatcher.java

```
public static void addDispatcherListener(DispatcherListener listener) {
    dispatcherListeners.add(listener);
}
```

```
public static void removeDispatcherListener(DispatcherListener listener) {
    dispatcherListeners.remove(listener);
}
```

有了 DispatcherListener 接口，相当于我们拥有了在 Dispatcher 初始化过程中进行自由扩展的一个接口。这也是 Struts2 在初始化主线中为我们提供的一个重要扩展点。

9.6.2 容器的初始化

容器的初始化过程是 Struts2 初始化的核心。这就是我们将上一节 Dispatcher 初始化中这一步骤单独抽取出来进行讲解的原因。在上一节中，我们知道容器的初始化工作是由 2 行代码完成的，这正好印证了我们在第 5 章中曾经提到过的容器定义的两个方面：

- **容器的创建过程**——init_PreloadConfiguration()
- **容器的依赖注入**——container.inject(this)

有关其中的第二步，我们在第 5 章中已经为读者详细剖析了其中的过程。而对于这第一步，我们有必要在这里深入挖掘一下其中的实现细节，其相关源代码如代码清单 9-18 所示。

代码清单 9-18 Dispatcher.java

```
private Container init_PreloadConfiguration() {
        // 获取 Configuration 对象
        Configuration config = configurationManager.getConfiguration();
        // 从 Configuration 中获取 Container 实例
        Container container = config.getContainer();
        // 设置是否进行 i18n 的重加载
        boolean reloadi18n =
Boolean.valueOf(container.getInstance(String.class,
StrutsConstants.STRUTS_I18N_RELOAD));
        LocalizedTextUtil.setReloadBundles(reloadi18n);

        return container;
}
```

这个方法并没有什么神秘之处，我们看到熟悉的 Configuration 和 ConfigurationManager 对象成为了构建容器的操作句柄。还记得我们之前对 ConfigurationManager 对象的一个评价吗？这是一个将初始化过程与元素调度过程合而为一的对象，主要目的是为了简化上层对象对配置调度的过程。因而，在 ConfigurationManager 的 getConfiguration 方法的调用中，我们实际上不仅获得了 Configuration 对象的操作接口，同时也在这个方法的调用中完成了对所有配置元素的调度。其相关源码如代码清单 9-19 所示。

代码清单 9-19　ConfigurationManager.java

```
public synchronized Configuration getConfiguration() {
        if (configuration == null) {
            // 如果 configuration 对象不存在，创建一个新的
            setConfiguration(new DefaultConfiguration(defaultFrameworkBeanName));
            try {
                // 调用 configuration 中的 reloadContainer 对象初始化 Container
                configuration.reloadContainer(getContainerProviders());
            } catch (ConfigurationException e) {
                setConfiguration(null);
                throw new ConfigurationException("Unable to load
configuration.", e);
            }
        } else {
            conditionalReload();
        }

        return configuration;
}
```

重新回到这段源码时，我们理解的角度发生了变化。因为我们在这里看到了 getConfiguration 的调用环境。于是，对 reloadContainer 这个核心方法的研究，就成为我们对容器初始化过程研究的重点。其源码如代码清单 9-20 所示。

代码清单 9-20　DefaultConfiguration.java

```
public synchronized List<PackageProvider>
reloadContainer(List<ContainerProvider> providers) throws
ConfigurationException {
      // 内部缓存对象的清理工作
      packageContexts.clear();
      loadedFileNames.clear();
      List<PackageProvider> packageProviders = new
ArrayList<PackageProvider>();

      // ==== 步骤 1 开始 ====
      // 创建 ContainerProperties 对象，这将是初始化所有 Properties 文件中
      // 相关运行参数的主要操作对象
      ContainerProperties props = new ContainerProperties();
      // 创建 ContainerBuilder，Container 的构造器
      ContainerBuilder builder = new ContainerBuilder();
      // ==== 步骤 1 结束 ====

      // ==== 步骤 2 开始 ====
      // 完成 ContainerProvider 配置加载方式的所有逻辑
      for (final ContainerProvider containerProvider : providers){
          // 首先初始化 ContainerProvider
```

```
            containerProvider.init(this);
            // 调用 register 方法，完成所有 Container 中的对象注册
            // Container 中的对象注册实际上全部会转化为调用 ContainerBuilder 中的
            // factory 方法的过程，这一过程也是构造器“搜集参数”的过程
            containerProvider.register(builder, props);
        }
        // 为 ContainerProperties 设置 ContainerBuilder，从而帮助其完成
        // 在 ContainerBuilder 中的 register 过程
        props.setConstants(builder);

        // 直接调用 factory 方法对当前的 Configuration 对象进行 Container 级别缓存
        builder.factory(Configuration.class, new Factory<Configuration>() {
            public Configuration create(Context context) throws Exception {
                  return DefaultConfiguration.this;
            }
        });
        // ==== 步骤 2 结束 ====

        ActionContext oldContext = ActionContext.getContext();
        try {

            // ==== 步骤 3 开始 ====
            // 为 ActionContext 构建基本的对象
            Container bootstrap = createBootstrapContainer();
            setContext(bootstrap);
            // 调用 ContainerBuilder 完成真正的容器初始化
            container = builder.create(false);
            setContext(container);
            // ==== 步骤 3 结束 ====

            // ==== 步骤 4 开始 ====
            objectFactory = container.getInstance(ObjectFactory.class);

            // 依次调用 PackageProvider 完成对 PackageConfig 对象的初始化
            for (final ContainerProvider containerProvider:providers){
                if (containerProvider instanceof PackageProvider) {
                    // 容器已经被创建出来，可以实施依赖注入了
                    container.inject(containerProvider);
                    ((PackageProvider)containerProvider).loadPackages();
packageProviders.add((PackageProvider)containerProvider);
                }
            }

            // 依次调用容器中出现的其他 PackageProvider 的加载逻辑
            // 这些 PackageProvider 可能来自于用户自扩展的 PackageProvider
            Set<String> packageProviderNames =
container.getInstanceNames(PackageProvider.class);
            if (packageProviderNames != null) {
                for (String name : packageProviderNames) {
```

```
                    PackageProvider provider =
container.getInstance(PackageProvider.class, name);
                    provider.init(this);
                    provider.loadPackages();
                    packageProviders.add(provider);
                }
            }
            // ==== 步骤 4 结束 ====

            // ==== 步骤 5 开始 ====
            // 对所有的 PackageConfig 进行 namespace 级别的重新分类
            // 这样做的目的在于当 XWork 框架响应 Http 请求时，
            // 可以根据 namespace 进行 URL 匹配
            rebuildRuntimeConfiguration();
            // ==== 步骤 5 结束 ====
        } finally {
            if (oldContext == null) {
                ActionContext.setContext(null);
            }
        }
        return packageProviders;
    }
```

容器的初始化与 Dispatcher 初始化几乎一样复杂，不过整个过程的逻辑思路却尽在我们的掌握之中。与 Dispatcher 的分析思路相同，我们也把容器初始化的过程自上而下分成五个步骤分别进行讲解。

在这五个步骤中，我们首先可以从配置元素的角度进行逻辑区分：前三个步骤是针对容器类的配置元素的初始化过程；而后两个步骤，则是对另一类配置元素 PackageConfig 进行初始化。因而，容器的初始化过程实际上不仅仅包含容器本身，其实是一个泛指的配置元素的初始化过程。

接下来，我们就对这五个步骤进行一一的解读。

9.6.2.1　步骤 1——定义容器构造器

这个步骤相当简单，由两行初始化代码构成。其中的 ContainerBuilder 已经是我们的老朋友了，它是构建 Container 必不可少的构造器，而 ContainerProperties 我们却没有接触过，我们不妨来看看它的源码，它是一个位于 DefaultConfiguration 内部的内部类，其源码如代码清单 9-21 所示。

代码清单 9-21　DefaultConfiguration.java

```
// 一个继承自 LocatableProperties 类的内部类，对应于 Properties 文件的配置形式
class ContainerProperties extends LocatableProperties {
```

```
        @Override
        public Object setProperty(String key, String value) {
            String oldValue = getProperty(key);
            if (oldValue != null && !oldValue.equals(value)
&& !defaultFrameworkBeanName.equals(oldValue)) {
                LOG.info("Overriding property "+key+" - old value:
"+oldValue+" new value: "+value);
            }
            return super.setProperty(key, value);
        }

        // 调用此方法后，所有在 Properties 文件中的运行参数都会被处理
        // ContainerBuilder 将对每个运行参数键值对调用 factory 方法进行参数搜集
        public void setConstants(ContainerBuilder builder) {
            for (Object keyobj : keySet()) {
                String key = (String)keyobj;
                builder.factory(String.class, key,
                        new
LocatableConstantFactory<String>(getProperty(key),
getPropertyLocation(key)));
            }
        }
}
```

从源码中我们可以看出，这只不过是一个用于将 Properties 文件中的键值对与 ContainerBuilder 关联起来的一个工具类。因而，它的逻辑调用实际上只不过是 ContainerBuilder 进行参数搜集的过程之一。

9.6.2.2 步骤 2——使用 ContainerBuilder 进行参数搜集

步骤 2 是 ContainerBuilder 进行参数搜集的过程。那么，刚刚创建出来的 ContainerBuilder 到哪里去搜集参数呢？这个时候，在 Dispatcher 初始化的第二个步骤中所初始化的那些配置加载器 ConfigurationProvider 就派上了用场。我们发现，在 ContainerProvider 的 register 方法中，本身就定义了 ContainerBuilder 作为其参数。因此，在不同的配置加载方式的实现类中，ContainerBuilder 将会被调用，用于对不同形式的配置元素进行参数搜集的操作。

为了验证这一推论，我们不妨打开 ContainerBuilder 的其中一个实现类 BeanSelectionProvider 来观察一下其 register 方法到底做了些什么事情，其源码如代码清单 9-22 所示。

代码清单 9-22 BeanSelectionProvider.java

```
public void register(ContainerBuilder builder, LocatableProperties props) {
        alias(ObjectFactory.class, StrutsConstants.STRUTS_OBJECTFACTORY,
 builder, props);
        alias(XWorkConverter.class,
```

```
StrutsConstants.STRUTS_XWORKCONVERTER, builder, props);
        alias(TextProvider.class,
StrutsConstants.STRUTS_XWORKTEXTPROVIDER, builder, props, Scope.DEFAULT);
        alias(ActionProxyFactory.class,
StrutsConstants.STRUTS_ACTIONPROXYFACTORY, builder, props);
        alias(ObjectTypeDeterminer.class,
StrutsConstants.STRUTS_OBJECTTYPEDETERMINER, builder, props);
        alias(ActionMapper.class,
StrutsConstants.STRUTS_MAPPER_CLASS, builder, props);
        alias(MultiPartRequest.class,
StrutsConstants.STRUTS_MULTIPART_PARSER, builder, props, Scope.DEFAULT);
        alias(FreemarkerManager.class,
StrutsConstants.STRUTS_FREEMARKER_MANAGER_CLASSNAME, builder, props);
        alias(VelocityManager.class,
StrutsConstants.STRUTS_VELOCITY_MANAGER_CLASSNAME, builder, props);
        alias(UrlRenderer.class,
StrutsConstants.STRUTS_URL_RENDERER, builder, props);
        alias(ActionValidatorManager.class,
StrutsConstants.STRUTS_ACTIONVALIDATORMANAGER, builder, props);
        alias(ValueStackFactory.class,
StrutsConstants.STRUTS_VALUESTACKFACTORY, builder, props);
        alias(ReflectionProvider.class,
StrutsConstants.STRUTS_REFLECTIONPROVIDER, builder, props);
        alias(ReflectionContextFactory.class,
StrutsConstants.STRUTS_REFLECTIONCONTEXTFACTORY, builder, props);
        alias(PatternMatcher.class,
StrutsConstants.STRUTS_PATTERNMATCHER, builder, props);
        alias(StaticContentLoader.class,
StrutsConstants.STRUTS_STATIC_CONTENT_LOADER, builder, props);
        alias(UnknownHandlerManager.class,
StrutsConstants.STRUTS_UNKNOWN_HANDLER_MANAGER, builder, props);

        // 这里省略了许多其他的代码
}

void alias(Class type, String key, ContainerBuilder builder, Properties
props, Scope scope) {
        if (!builder.contains(type)) {
            String foundName = props.getProperty(key, DEFAULT_BEAN_NAME);
            if (builder.contains(type, foundName)) {
                if (LOG.isDebugEnabled()) {
                    LOG.info("Choosing bean ("+foundName+") for "+type);
                }
                // 调用 ContainerBuilder 的 alias 方法完成参数搜集
                builder.alias(type, foundName, Container.DEFAULT_NAME);
            } else {
                try {
                    Class cls = ClassLoaderUtil.loadClass(foundName,
this.getClass());
```

```
                    // 调用 ContainerBuilder 的 factory 方法完成参数搜集
                    builder.factory(type, cls, scope);
                } catch (ClassNotFoundException ex) {
                    // 这里省略了许多其他代码
                }
            }
        } else {
            LOG.warn("Unable to alias bean type "+type+", default mapping
already assigned.");
        }
}
```

这里除了调用 ContainerBuilder 的 alias 方法和 factory 方法外，竟然已经没有其他的逻辑了！由此可见，对我们来说一直很神秘的 **ContainerBuilder 的各种实现的核心逻辑不过是 ContainerBuilder 搜集参数的过程！**

在整个步骤 2 中，最值得我们学习的其实是配置加载方式 ContainerProvider 的执行次序。对于 ContainerProvider 而言，其 init 方法和 register 方法会在整个过程中被依次调用。我们可以想象，init 方法一定是用于解析各种不同的配置形式，并将其缓存于 ContainerProvider 内部。有兴趣的读者可以顺着思路查看 ContainerProvider 其他实现类的 init 方法。我们在这里就不再展开进行分析了。

9.6.2.3 步骤 3——使用 ContainerBuilder 进行容器创建

这个步骤涉及的代码量也不大。因为最为复杂的 ContainerBuilder 搜集参数的过程已经完成，剩下的过程也不过是调用一下 create 方法把容器创建出来而已。

在整个代码段中 setContext 方法有两次调用，这似乎有一点值得商榷。这里的调用，实际上是容器和 ActionContext 的一个互动。不过这里的互动只是强迫 ActionContext 使用在 Struts2 / XWork 容器中所指定的 ValueStack 实现类，从而完成 ActionContext 与 ValueStack 关系的设定。按照笔者的理解，这里所期望表达的逻辑似乎并不是特别清晰。

9.6.2.4 步骤 4——初始化事件映射元素 PackageConfig

在容器构建完成之后，Struts2 配置元素中最重要的配置元素已经被成功创建出来。因而，接下来的工作就是初始化另外一类配置元素：事件映射关系。

对事件映射关系的初始化过程，实际上也不过是对事件映射加载器 PackageProvider 的生命周期的调用过程。当 reloadContainer 接过了初始化逻辑的执行权，对 Configuration 对象内部所缓存的所有 PackageConfig 进行初始化就显得非常简单了。与之前我们谈到的 ContainerProvider 中接口的调用类似，所有的 PackageProvider 之间并没有区别，它们通过一个循环，依次调用所有实现类中的 init 方法和 loadPackages 方法。其最终的结果，实际上是填充 Configuration 对象中的事件映射配置对象。

9.6.2.5　步骤 5——对 PackageConfig 进行运行期改造

所有的 PakcageConfig 已经被初始化完成，Struts2 最后对 PackageConfig 的结构进行运行期改造，使之支持 namespace 的寻址方式。这种寻址方式有助于框架在响应事件时便捷地获取当前事件相应序列的完整配置元素。

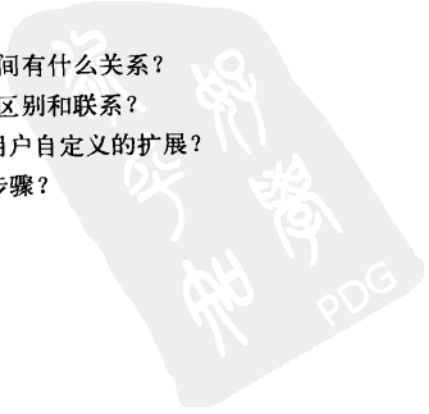
9.7　小结

在本章中，我们阐述了 Struts2 的初始化主线的运行过程，并在整个过程中认识了核心分发器（Dispatcher）、配置元素的加载器（ConfigurationProvider）、配置元素的构造器（Builder）、配置管理元素（Configuration）以及配置操作接口（ConfirugationManager）这样一些站在不同角度对初始化主线做出贡献的不同元素。

本章的内容是非常枯燥的，因为我们所看到的是一个又一个概念的嵌套，一个又一个对象的定义。然而，当我们将这些概念和对象联系起来，站在宏观的高度去看待整个初始化过程中各个不同元素的作用时，我们就会发现 Struts2 对于初始化主线的设计着实下了一番苦工夫。事实上，无论是配置元素加载器的定义还是配置元素构造器的使用，都可以成为我们在日常编程过程中对资源初始化设计的一个模板。如果读者能够将这些设计思想和理念真正运用到日常的编程中，那么本书的目的也就达到了。

以下是 Struts2 初始化主线中的一些核心问题，读者可以试着在本章中寻找答案：

- Struts2 初始化主线的核心驱动力是什么？
- 什么是“配置元素的对象化过程”？这个过程对于框架的意义是什么？
- Struts2 在配置元素的初始化过程中做了哪些准备工作？
- 什么是配置元素的加载方式？它的主要作用是什么？它在运行期的执行次序是怎么样的？
- 什么是配置元素的构造器？ Struts2 的配置元素的构造器在运行期可以分成哪两个主要步骤？
- 什么是配置元素的管理类？它与配置元素的核心元素之间有什么关系？
- Configuration 对象和 ConfigurationManager 之间有什么区别和联系？
- 在 Struts2 初始化的过程中，我们可以在哪些地方提供用户自定义的扩展？
- 核心分发器（Dispatcher）的初始化过程可以分为哪些步骤？
- 容器（Container）的初始化过程可以分为哪些步骤？



第 10 章　井然有序——与 Http 请求的战斗

早在第 3 章中，我们就曾经对 Struts2 作为一个 Web 层开发框架的核心内容做过一番评述。我们在这里首先回顾一下当时总结出来的一个重要结论：

结论　Struts2 是一个运行于 Web 容器的表示层框架，其核心作用是帮助我们处理 Http 请求。

因此，**处理 Http 请求是 Struts2 的首要任务**。相比 Struts2 的初始化主线，对 Http 请求进行处理这样一条运行主线，其核心地位也就不言而喻了。

在本章的标题中，我们把 Struts2 对 Http 请求进行的处理比作是一场战斗，并且冠以“井然有序”的描述。如果大家对于在第 7 章中我们有关 XWork 的控制流体系的讲解还有印象，可知此时对 Http 请求的战斗比喻正是我们当时所引入的一个战斗序列的比喻的延伸。而“井然有序”这四个字，则进一步反映出 Struts2 对于整个 Http 请求的处理过程有两大特点：

- **Struts2 对于 Http 请求的处理流程的设计非常完备**
- **Struts2 对于构成 Http 请求处理的各个元素之间调用关系的处理非常妥当**

在接下来的章节中，我们将首先站在全局的角度对整个 Http 请求的处理过程进行宏观分析。紧接着，对于 Http 请求处理的不同阶段，我们分别进行详细的流程分析。希望通过这一章的讲解，能够帮助读者了解 Struts2 进行 Http 请求处理的完整过程。

10.1　制定作战计划

要打好一场战役，就必须做好详细而完备的作战计划。俗话说，知己知彼方能百战百胜。所以我们在这里首先分析一下敌我的形势，看看我们手里到底有哪些战斗资源。在了解了这些基础资源之后，我们再来看看如何进行战斗资源的合理分配、如何开辟战场、如何组织起一个完整的战斗流程。

10.1.1　战斗资源

制定作战计划的一个重要步骤，就是分析我们拥有哪些可用的**战斗资源**。对于 Http

请求处理来说，所谓的战斗资源可以理解为：在 Http 请求处理的过程中可以使用的编程元素和编程技术。此时，我们在本书的核心技术篇中讲到的那些技术就可以派上用场了。

事实上，在整个核心技术篇中，我们的话题始终没有离开两大利器：OGNL 和 XWork。我们可以回顾一下这两大利器的主要作用：

- OGNL——表达式引擎，架起外部世界与 Java 世界沟通的桥梁
- XWork——请求处理器，将请求划分为若干处理步骤并分派到不同处理元素调度执行

对这两大利器都无须做过多的解释，因为我们可以把它们看作是通用的组件，在任何满足业务场景的 Java 程序中使用。用它们来处理 Http 请求则更加方便。OGNL 在这里承担着 Web 与 Java 进行沟通的职责。而 Http 请求作为请求的一种，由 XWork 进行处理也是一个自然而然的选择。

然而，单有这些通用组件还不够。因为 Http 请求作为一个特殊的请求，其运行环境是在 Web 容器中，因而当我们需要在 Web 环境中使用这些组件时，就不得不构建一些与 Web 容器打交道的对象组件，并将这些对象组件与通用组件的使用紧密融合在一起。这些与 Web 容器打交道的对象组件，实际上也是我们可依赖的重要战斗资源。

有了这些战斗资源，我们又如何将它们拧成一股绳，打赢整个与 Http 请求的战斗呢？我们需要将这些战斗资源进行合理分配，从而勾勒出一个完整的战斗进程。

10.1.2　战斗进程

事实上，我们在第 3 章中已经就 Struts2 的 Http 请求处理主线做了一个运行阶段的划分。所谓运行阶段，对应于整个战斗序列来说，其实就是勾勒了一个战斗的进程。

当时，我们对 Http 请求处理主线进行阶段划分的依据，主要可以归纳为两个方面：

- **代码上的物理解耦——两个阶段分属于不同的框架主导执行**
- **逻辑职责上的解耦——两个阶段的主要职责完全不同**

在这种大背景之下，再结合两个不同的执行阶段涉及的所有微观元素，我们把整个战斗的进程采用如图 10-1 所示的时序图画出来。这幅时序图，包含了整个 Struts2 与 Http 请求进行战斗的全部过程。我们在这里试着为大家进行初步解读。在之后的章节中，我们将对其中的每一个调用流程用源码加以分析。

在这幅时序图中首先看到的，是我们对**战场的划分**。在时序图中，我们为不同的元素指出了不同的逻辑执行归属，从而形成了两个主战场。**而这两个主战场，与我们所说的两个执行阶段（Http 请求的预处理阶段和 XWork 事件处理阶段）是完全一致的。**

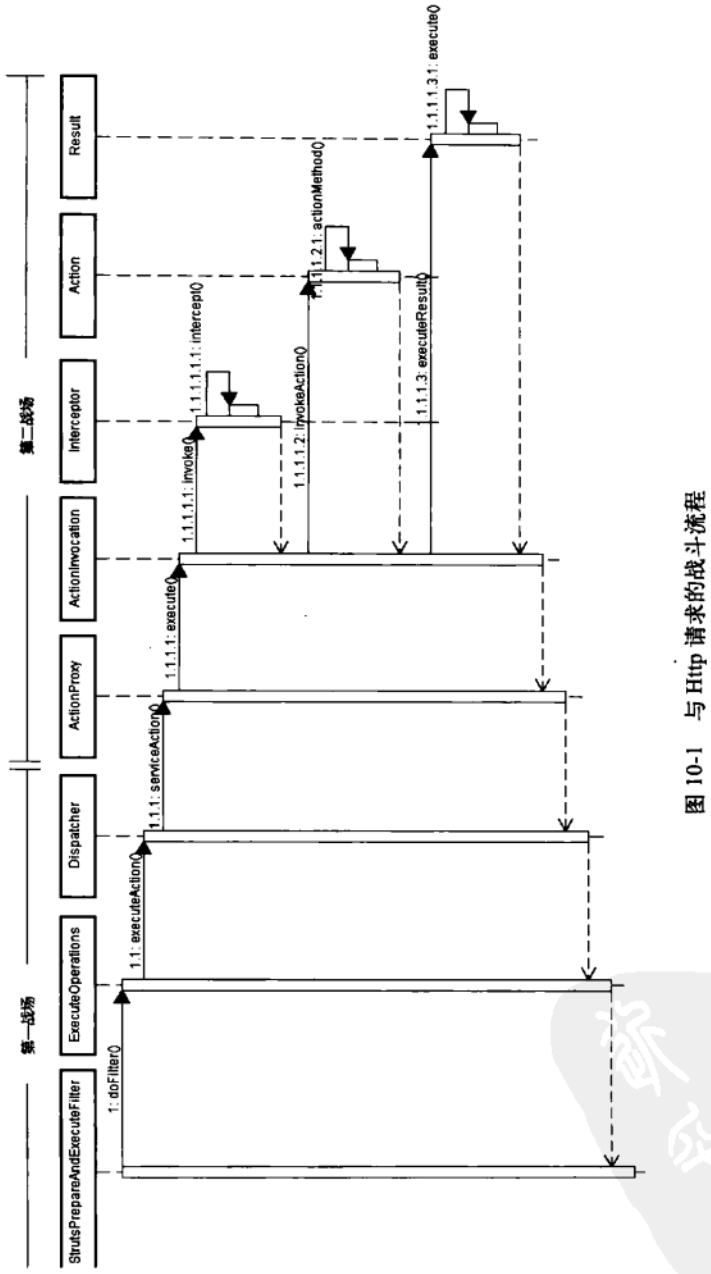


图 10-1 与 Http 请求的战斗流程

时序图中表示第一战场的部分，是对 Http 请求的预处理阶段。这个阶段主要由 Struts2 完成，其主要职责是**与 Web 容器打交道，将 Http 请求处理成为普通的 Java 对象**。我们可以看到在这个阶段的时序图中显现出来的调用逻辑比较简单，从入口程序开始依次调用相关的执行对象并最终交给核心分发器 Dispatcher 进行执行调度。

时序图中表示第二战场的部分，则是 XWork 事件处理阶段。程序的执行控制权在此时交给了 XWork 框架，其主要职责是**对请求进行核心逻辑处理**。在这个阶段的时序图中，所有的元素都是我们所熟悉的 XWork 事件处理框架中的元素，而它们之间所表现出来的执行顺序与我们在第 8 章中对 XWork 的执行分析所得出的结论是一致的。

接下来，我们就根据不同的执行阶段，对每个阶段的执行逻辑进行详细的解读。

10.2　第一战场——Http 请求的预处理阶段

10.2.1　三探入口程序

回到 Struts2 处理 Http 请求的问题，我们再次试图探究入口程序 StrutsPrepareAndExecuteFilter。但是，这一次我们所关注的部分是 Filter 中真正处理 Http 请求的 doFilter 方法，其源码如代码清单 10-1 所示。

代码清单 10-1　StrutsPrepareAndExecuteFilter.java

```
/**
 * Http 请求的处理类，实现了 Servlet 标准中的 Filter
 */
public class StrutsPrepareAndExecuteFilter implements StrutsStatics,
Filter {
    // 执行 Http 请求预处理的操作类
    protected PrepareOperations prepare;
    // 执行 Http 请求处理的逻辑执行的操作类
    protected ExecuteOperations execute;

    // 这里省略了许多其他的代码

    // 处理 Http 请求的入口方法
    public void doFilter(ServletRequest req, ServletResponse res,
FilterChain chain) throws IOException, ServletException {

        HttpServletRequest request = (HttpServletRequest) req;
        HttpServletResponse response = (HttpServletResponse) res;

        try {
```

```
            // ==== 步骤 1 开始 ====
            // 设置 Encoding 和 Locale
            prepare.setEncodingAndLocale(request, response);
            // ==== 步骤 1 结束 ====

            // ==== 步骤 2 开始 ====
            // 创建 Action 执行所对应的 ActionContext
            prepare.createActionContext(request, response);
            // ==== 步骤 2 结束 ====

            // ==== 步骤 3 开始 ====
            // 把核心分发器 Dispatcher 分配至当前线程
            prepare.assignDispatcherToThread();
            // ==== 步骤 3 结束 ====

            // 处理被过滤的 URL
   if ( excludedPatterns != null && prepare.isUrlExcluded(request,
excludedPatterns)) {
            chain.doFilter(request, response);
   } else { // 真正需要进行 URL 处理的逻辑起始

            // ==== 步骤 4 开始 ====
            // 首先对 request 进行一定的封装
            request = prepare.wrapRequest(request);
            // ==== 步骤 4 结束 ====

            // ==== 步骤 5 开始 ====
            // 根据 request 请求查找 ActionMapping
            ActionMapping mapping = prepare.findActionMapping(request,
response, true);
            // ==== 步骤 5 结束 ====

            // 没有找到对应的 ActionMapping
            if (mapping == null) {
                  // 判断是否将 URL 处理为静态资源
                  boolean handled =
execute.executeStaticResourceRequest(request, response);
                  if (!handled) {
                        chain.doFilter(request, response);
                  }
           } else {
                        // 执行 URL 请求的处理
                  execute.executeAction(request, response, mapping);
           }
        }
        } finally {
            // 清理 Request
            prepare.cleanupRequest(request);
```

```
            }
        }
    }
```

doFilter 方法是整个 Struts2 的第二条主线的入口方法。在这个方法中，我们可以看到许多在第一条主线中被初始化的对象开始发挥作用，我们在上一章中已经见过它们，它们主要包括：

- Dispatcher——核心分发器，执行 Struts2 处理逻辑的实际场所。
- PrepareOperations——Http 预处理类，对所有的 Http 请求进行预处理。
- ExecuteOperations——Http 处理执行类，执行 Http 请求的场所。

在进行 URL 处理时，doFilter 方法会根据职责的不同，有条不紊地将 URL 处理过程分配给 PrepareOperations 和 ExecuteOperations 来分别完成。不过在这里，PrepareOperations 和 ExecuteOperations 实际上也只是一层薄薄的代理层，实际在其中起决定性作用的还是核心分发器 Dispatcher。这一点将在我们后面对源码的分析中看到。

那么 PrepareOperations 和 ExecuteOperations 存在的实际意义又是什么呢？这里我们不得不讲 StrutsPrepareAndExecuteFilter 的两个小兄弟：StrutsPrepareFilter 和 StrutsExecuteFilter。它们位于与 StrutsPrepareAndExecuteFilter 相同的 package 下，如图 10-2 所示。

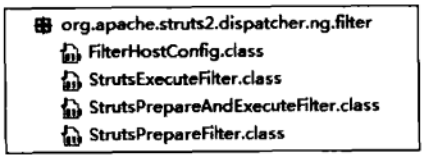


图 10-2　StrutsPrepareFilter 和 StrutsExecuteFilter 所在位置

如果把这两个类与 Http 请求处理的两个执行阶段联系在一起，我们就会发现 StrutsPrepareFilter 和 StrutsExecuteFilter 正是这两个执行阶段的物理划分的具体表现。其中，StrutsPrepareFilter 负责 Struts2 的初始化工作和每个 Http 请求的预处理工作，而 StrutsExecuteFilter 则负责 Http 请求的实际逻辑处理。

Struts2 在这里将两个执行阶段进行明确的物理划分有着更加深远的意义。因为我们的应用总是存在一种需求，需要在 Struts2 的两个执行阶段之间插入特定的逻辑片段（比如我们有时需要对 HttpServletRequest 或者 HttpServletResponse 对象进行额外的处理等）。这个时候，Struts2 额外定义的这两个小兄弟就可以派上用场了。因为只需要在 web.xml 中依次定义 Filter 链，将我们自定义的 Filter 插入 StrutsPrepareFilter 和 StrutsExecuteFilter 的定义之间即可。

由此可见，Struts2 在 Http 请求的处理上下了一番苦工夫，不仅将 Http 请求过程中的每个步骤进行了必要的逻辑划分，还在步骤和步骤之间提供了足够的空间以供扩展使用。这也使得我们可以非常自由地使用 Struts2 的所有特性，而不必担心与其他框架整合所产生的兼容性问题。

10.2.2 Http 请求预处理类——PrepareOperations

PrepareOperations 的作用在于对 Http 请求进行预处理。PrepareOperations 的存在可以有效地对外屏蔽 Dispatcher，从而起到为 Dispatcher 保驾护航的作用。这样一来，核心分发器 Dispatcher 就可以顺利地将 Web 容器与 MVC 实现进行有效隔离。

PrepareOperations 的几大作用，我们已经可以在之前的 StrutsPrepareAndExecuteFilter 处理 Http 请求的源码中一一解读出来。在代码清单 10-1 中，我们也为整个源码的执行过程做了不同的逻辑步骤注释。接下来，我们就来对其中的实现细节进行源码级别的分析。

10.2.2.1 步骤 1——设置 Encoding 和 Locale

Struts2 对每个请求都设置 Encoding 和 Locale 的原因是 Struts2 本身支持在系统级别进行默认的 Encoding 和 Locale 设置。其源代码如代码清单 10-2 所示。

代码清单 10-2 PrepareOperations.java

```
/**
 * 准备 Request，设置 Encoding 和 Locale
 */
public void setEncodingAndLocale(HttpServletRequest request,
HttpServletResponse response)
    dispatcher.prepare(request, response);
}
```

这里只是做了一个简单的转发，实际由 Dispatcher 来完成对 HttpServletRequest 的设置，源代码如代码清单 10-3 所示。

代码清单 10-3 Dispatcher.java

```
/**
 * 准备 Request，设置 Encoding 和 Locale
 *
 * @param request The request
 * @param response The response
 */
public void prepare(HttpServletRequest request, HttpServletResponse
response) {
    String encoding = null;
```

```
    // 找到系统级别默认的 encoding 值
    if (defaultEncoding != null) {
        encoding = defaultEncoding;
    }

    Locale locale = null;
    // 找到系统级别默认的 locale 值
    if (defaultLocale != null) {
        locale = LocalizedTextUtil.localeFromString(defaultLocale,
request.getLocale());
    }

    if (encoding != null) {
        try {
        // 设置 encoding 值
            request.setCharacterEncoding(encoding);
        } catch (Exception e) {
            LOG.error("Error setting character encoding to '" + encoding +
"' - ignoring.", e);
        }
    }

if (locale != null) {
    // 设置 locale 值
        response.setLocale(locale);
    }
}
```

Encoding 和 Locale 是 Web 页面上的重要属性，也是 Web 程序进行国际化（i18n）处理的核心参数。因而在 Http 请求预处理的第一步就做这两个参数的设置也相当合理。

10.2.2.2　步骤 2——创建 ActionContext

ActionContext 是 XWork 中的概念，回顾一下我们在 XWork 章节中有关 ActionContext 生命周期的讲解，就会发现 ActionContext 拥有比 XWork 控制流体系更长的生命周期。因此，在 PrepareOperations 中创建 ActionContext 实际上是为了**之后 XWork 能够在非 Web 容器环境下接手 Http 请求的处理工作打下坚实的基础**。这也是 Struts2 另辟蹊径，将 Http 请求的实际处理与 Web 容器进行解耦合的核心步骤之一。

与 Dispatcher 对象一样，ActionContext 同样使用了 ThreadLocal 模式，从而保证在整个线程执行过程中的数据共享以及数据访问的线程安全性。这一点我们在之前的分析中已经详细阐述，这里就不再展开 ActionContext 数据结构的分析。在这里需要展开的是与 ActionContext 创建相关的源代码，如代码清单 10-4 所示。

代码清单 10-4 PrepareOperations.java

```
/**
 * 创建 ActionContext，并初始化到当前线程
 */
public ActionContext createActionContext(HttpServletRequest request,
HttpServletResponse response) {
ActionContext ctx;
// 为当前线程的 ActionContext 进行计数，为日后 cleanup 做准备
    Integer counter = 1;
    Integer oldCounter = (Integer)
request.getAttribute(CLEANUP_RECURSION_COUNTER);
    if (oldCounter != null) {
        counter = oldCounter + 1;
    }
    // 获取当前线程中的 ActionContext
    ActionContext oldContext = ActionContext.getContext();
    if (oldContext != null) {
        // 当前线程中存在 ActionContext，复制为新的 ActionContext
        ctx = new ActionContext(new HashMap<String,
Object>(oldContext.getContextMap()));
} else {
    // 创建 ValueStack
        ValueStack stack =
dispatcher.getContainer().getInstance(ValueStackFactory.class).
createValueStack();
        // 设置 ValueStack 的数据环境，注意这里的数据环境通过 Dispatcher 的转换，
        // 实际上已经成为普通的 Java 对象，这就消除了 ValueStack 对容器的依赖
        stack.getContext().putAll(dispatcher.createContextMap(request,
response, null, servletContext));
        // 将 ValueStack 的数据环境与 ActionContext 等同起来
        ctx = new ActionContext(stack.getContext());
    }
request.setAttribute(CLEANUP_RECURSION_COUNTER, counter);
// 设置 ActionContext 到当前线程
    ActionContext.setContext(ctx);
    return ctx;
}
```

ActionContext 与 ValueStack 形影不离的特性充分体现在这段源码中。ActionContext 的创建总是伴随着 ValueStack 的共同创建，更为值得注意的是，两者的上下文环境也是等同的。

通过 PrepareOperations 来创建 ActionContext，我们还可以发现，原本在 HttpServletRequest、HttpServletResponse 中的数据被封装成普通的 Java 对象，这是 Struts2 将 MVC 实现与 Web 容器进行解耦合的第一步，也是最重要的一步。如果从更深层次的角度去考虑，解耦合的步骤实际上是把 Http 请求与 Web 容器依赖关系转嫁到最为基

础的 Java 对象映射中去。如此一来，所有之后的逻辑处理过程就具备了相同的数据基础并且完全消除了对 Web 容器实现标准的依赖。

10.2.2.3　步骤 3——将核心分发器 Dispatcher 绑定至当前线程

之前我们已经分析了 ThreadLocal 模式的运行机理，由于 Dispatcher 本身采用了 ThreadLocal 模式，因而，在 PrepareOperations 中，我们需要为 Dispatcher 绑定当前线程的执行副本。其源代码如代码清单 10-5 所示。

代码清单 10-5　PrepareOperations.java

```
/**
 * 将 dispatcher 对象绑定到当前线程
 */
public void assignDispatcherToThread() {
Dispatcher.setInstance(dispatcher);
}
```

这里的代码已经没有任何神秘之处了，它只不过是 ThreadLocal 模式中另外一个重要组成要件的调用封装。通过 Dispatcher 中静态方法的调用，保证了当前的 Dispatcher 实例就是一个线程安全的实例，可以在多线程环境中放心使用。

从代码中可以看出，只要 ThreadLocal 模式的两个步骤被正确定义，将实例变量绑定到当前线程只是一个简单的访问方法的调用而已。

10.2.2.4　步骤 4——对 HttpServletRequest 进行一定的封装

Struts2 在处理 Http 请求时，并不是直接使用容器的 HttpServletRequest 的实现，而是使用了装饰模式，使得在 Struts2 框架范围内的 Request 请求会首先经过 Struts2 的扩展处理。Struts2 对 HttpServletRequest 实现进行装饰的类是：StrutsRequestWrapper 和 MultiPartRequestWrapper。

Struts2 为什么要对 HttpServletRequest 进行装饰呢？主要原因在于，传统的 Web 容器元素的数据是通过 HttpServletRequest 的接口实现访问的；而在 Struts2 中，数据源的存储位置发生了变化，它们变得不再与 Web 容器对象绑定在一起，而是以 ActionContext 的形式存在于当前线程中。因此，传统的访问方式将无法顺利访问到 Struts2 中的数据。为了解决这一问题，Struts2 做出了扩展。这一扩展是 Struts2 非常经典的一个手笔，它直接填补了框架与 Web 容器之间由于设计理念之间的不同而造成的鸿沟。

Struts2 对 HttpServletRequest 的装饰，在 PrepareOperations 的 wrapRequest 方法中完成，其源码如代码清单 10-6 所示。

代码清单 10-6 PrepareOperations.java

```
public HttpServletRequest wrapRequest(HttpServletRequest oldRequest)
throws ServletException {
    HttpServletRequest request = oldRequest;
    try {
        request = dispatcher.wrapRequest(request, servletContext);
    } catch (IOException e) {
        String message = "Could not wrap servlet request with
MultipartRequestWrapper!";
        throw new ServletException(message, e);
    }
    return request;
}
```

代码本身很简单，依然是转发到 Dispatcher 中去完成。这里需要注意的是，对于含有文件上传的 Request 处理与传统的 Request 处理是不同的。而 Dispatcher 中的 wrapRequest 方法，如代码清单 10-7 所示。

代码清单 10-7 Dispatcher.java

```
public HttpServletRequest wrapRequest(HttpServletRequest request,
ServletContext servletContext) throws IOException {
        // 判断 request 是否已经被装饰过
        if (request instanceof StrutsRequestWrapper) {
            return request;
        }
        // 根据 request 的 content type 的不同，区分不同的装饰类型
        String content_type = request.getContentType();
        if (content_type != null &&
content_type.indexOf("multipart/form-data") != -1) {
            MultiPartRequest mpr = null;
            // 从容器中寻找 MultiPartRequest 的实现类
            Set<String> multiNames =
getContainer().getInstanceNames(MultiPartRequest.class);
            if (multiNames != null) {
                for (String multiName : multiNames) {
                    if (multiName.equals(multipartHandlerName)) {
                        // 根据名称获取容器中设置的 MultiPartRequest 的实现类
                        mpr =
getContainer().getInstance(MultiPartRequest.class, multiName);
                    }
                }
            }
            if (mpr == null ) {
                mpr = getContainer().getInstance(MultiPartRequest.class);
            }
            request = new MultiPartRequestWrapper(mpr, request,
```

```
getSaveDir(servletContext));
        } else {
            // 将 request 装饰为新的 StrutsRequestWrapper
            request = new StrutsRequestWrapper(request);
        }

        return request;
    }
```

有关 StrutsRequestWrapper 的实现细节，我们会在后续章节中详细分析。在这里读者需要明确的是，HttpServletRequest 经过 Struts2 的处理，已经不是 Web 容器默认的实现了，而是被装饰成一个包装类，其中蕴含了 Struts2 自身的数据访问的逻辑。

10.2.2.5　步骤 5——根据 Http 请求查找 ActionMapping

从数据结构上看，ActionMapping 只是一个很普通的 Java 对象，然而它却意义非凡，因为它所表达的意思是将一个 URL 形式的 Http 请求与 Struts2 中的 Action 建立起关系来。如果读者还记得我们曾经在第 2 章中谈到过的 Web 开发中的一些主要困境，可知 URL Mapping 问题是我们当时提到的一个问题之一。而 ActionMapping 正是 Struts2 中表示 URL 与 Java 对象对应关系的一个组件类。

ActionMapping 的内部本身不包含逻辑。翻开 ActionMapping 这个类的源码，我们可以发现它只是一个简单的 JavaBean 而已。在 Struts2 进行 Http 请求处理时，由 ActionMapper 的实现类在运行期查找相应的事件映射关系并生成 ActionMapping 对象。因此，查找 ActionMapping 也就成为 Http 请求预处理的步骤之一。其源码如代码清单 10-8 所示。

代码清单 10-8　PrepareOperations.java

```
public ActionMapping findActionMapping(HttpServletRequest request,
HttpServletResponse response, boolean forceLookup) {
    ActionMapping mapping = (ActionMapping)
request.getAttribute(STRUTS_ACTION_MAPPING_KEY);
    if (mapping == null || forceLookup) {
        try {
            // 使用 ActionMapper 的实现类来查找 ActionMapping
            mapping = dispatcher.getContainer().getInstance(ActionMapper
.class).getMapping(request, dispatcher.getConfigurationManager());
            if (mapping != null) {
                request.setAttribute(STRUTS_ACTION_MAPPING_KEY, mapping);
            }
        } catch (Exception ex) {
            dispatcher.sendError(request, response, servletContext,
HttpServletResponse.SC_INTERNAL_SERVER_ERROR, ex);
        }
```

```
        }

        return mapping;
    }
```

从源码中我们可以发现，**ActionMapper 才是 Struts2 进行 URL Mapping 关系查找的核心类**。因此 ActionMapper 的实现机制决定了 Struts2 进行 URL Mapping 关系的处理方法。在 Struts2 中，存在着多个 ActionMapper 的实现类，它们分别代表不同的 URL Mapping 关系的查找策略，如图 10-3 所示。

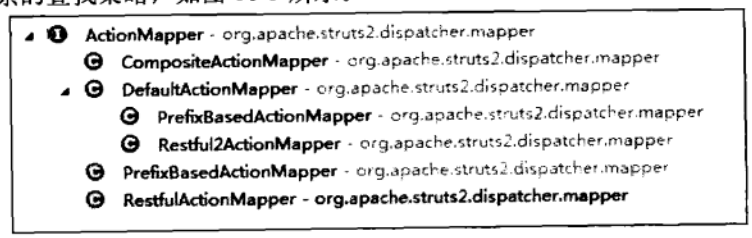


图 10-3 ActionMapper 的实现类

从图中，我们除了看到 DefaultActionMapper 作为 Struts2 的默认 URL Mapping 关系的查找策略实现类以外，Struts2 还提供了许多其他的实现机制，包括 Restful 风格的 URL 的映射查找策略和基于前缀的 URL 映射查找策略的实现。因而，ActionMapper 也成为 Struts2 的一个重要的扩展点，大家可以自己实现 ActionMapper 从而指定各种不同的 URL Mapping 的查找方式。

10.2.3 Http 请求的执行类——ExecuteOperations

ExecuteOperations 是 Http 请求的逻辑的执行场所。在 ExecuteOperation 中仅仅包含 2 个方法，分别用于处理静态资源和实际的 Http 请求。其源代码如代码清单 10-9 所示。

代码清单 10-9 ExecuteOperations.java

```
public class ExecuteOperations {
    // 执行操作时所需要的 servletContext 引用和绑定到当前线程的 dispatcher
    private ServletContext servletContext;
    private Dispatcher dispatcher;

    // 初始化构造函数
    public ExecuteOperations(ServletContext servletContext, Dispatcher
dispatcher) {
        this.dispatcher = dispatcher;
        this.servletContext = servletContext;
    }
```

```
    /**
     * 处理静态资源
     *
     * @return 返回值表示静态资源是否被处理
     * @throws IOException
     * @throws ServletException
     */
    public boolean executeStaticResourceRequest(HttpServletRequest
request, HttpServletResponse response) throws IOException,
ServletException {
        // 查找静态资源的根路径
        String resourcePath = RequestUtils.getServletPath(request);

        if ("".equals(resourcePath) && null != request.getPathInfo()) {
            resourcePath = request.getPathInfo();
        }

        // 获取静态资源的加载处理类
        StaticContentLoader staticResourceLoader =
dispatcher.getContainer().getInstance(StaticContentLoader.class);
        // 判断静态资源加载处理类是否能处理 URL 所对应的静态资源
        if (staticResourceLoader.canHandle(resourcePath)) {
            staticResourceLoader.findStaticResource(resourcePath,
request, response);
            return true;
        } else {
            // 这只是一个普通的 URL 请求，直接进入下一个处理流程
            return false;
        }
    }

    /**
     * 执行 Action
     *
     * @throws ServletException
     */
    public void executeAction(HttpServletRequest request,
HttpServletResponse response, ActionMapping mapping) throws
ServletException {
    // 将所有的参数传入 dispatcher 中执行逻辑
    dispatcher.serviceAction(request, response, servletContext, mapping);
    }
}
```

可见，在这里 ExecuteOperation 只是一个执行句柄。所有的逻辑执行实际上还是围绕着 Dispatcher 这个核心分发器展开。我们所需要重点关注的是 Dispatcher 中完成逻辑的 serviceAction 方法，其源码如代码清单 10-10 所示。

代码清单 10-10 Dispatcher.java

```
public void serviceAction(HttpServletRequest request, HttpServletResponse
response, ServletContext context, ActionMapping mapping) throws
ServletException {

    // 创建 MVC 运行的数据环境
    Map<String, Object> extraContext = createContextMap(request, response,
mapping, context);

    // 如果之前存在创建过的 ValueStack，则直接做一个 ValueStack 的复制
    ValueStack stack = (ValueStack)
request.getAttribute(ServletActionContext.STRUTS_VALUESTACK_KEY);
    boolean nullStack = stack == null;
    // 没有找到已存在的 ValueStack，则从 ActionContext 中获取当前线程的 ValueStack
    if (nullStack) {
        ActionContext ctx = ActionContext.getContext();
        if (ctx != null) {
            // ActionContext 是线程安全的，所以这里获取的是当前线程的 ValueStack
            stack = ctx.getValueStack();
        }
}
// 将 ValueStack 也设置到数据环境中
    if (stack != null) {
        extraContext.put(ActionContext.VALUE_STACK,
valueStackFactory.createValueStack(stack));
    }
    String timerKey = "Handling request from Dispatcher";
    try {
        UtilTimerStack.push(timerKey);
        // 从 ActionMapping 中分别获取 Action 的 3 大要素
        String namespace = mapping.getNamespace();
        String name = mapping.getName();
        String method = mapping.getMethod();

        Configuration config = configurationManager.getConfiguration();
        // 创建一个 ActionProxy，这里已经完全进入 XWork 的世界了
        ActionProxy proxy = config.getContainer().getInstance(ActionProxyFactory
.class).createActionProxy(namespace, name, method, extraContext,true,false);

        request.setAttribute(ServletActionContext.STRUTS_VALUESTACK_KEY,
proxy.getInvocation().getStack());

        // 如果 ActionMapping 中包含 Result 对象，表明直接跳过 Action 而执行 Result
        if (mapping.getResult() != null) {
            Result result = mapping.getResult();
            result.execute(proxy.getInvocation());
       } else {
            // 执行 ActionProxy，真正运行 XWork 中的 MVC 实现
```

```
            proxy.execute();
        }

        // 如果之前存在 ValueStack，重新设回 Request 对象中
        if (!nullStack) {
        request.setAttribute(ServletActionContext.STRUTS_VALUESTACK_KEY,
stack);
        }
    } catch (ConfigurationException e) {
    if(devMode) {
            String reqStr = request.getRequestURI();
            if (request.getQueryString() != null) {
                reqStr = reqStr + "?" + request.getQueryString();
            }
            LOG.error("Could not find action or result\n" + reqStr, e);
        }
        else {
               LOG.warn("Could not find action or result", e);
        }
        // 向 Web 容器发送错误返回信息
        sendError(request, response, context,
HttpServletResponse.SC_NOT_FOUND, e);
    } catch (Exception e) {
        sendError(request, response, context,
HttpServletResponse.SC_INTERNAL_SERVER_ERROR, e);
    } finally {
        UtilTimerStack.pop(timerKey);
    }
}
```

这里是 Struts2 处理 Http 请求的真正位置。我们终于在代码中看到了 XWork 框架的入口类：ActionProxy。而这段代码也向我们揭示了之前所提到的关于核心分发器 Dispatcher 的三个重要结论：

- XWork 框架的相关逻辑实际上由 Dispatcher 创建并负责驱动执行
- Dispatcher 负责 Http 请求不同处理阶段的数据转化工作，从而保证代码在不同执行阶段的无缝切换
- 在 XWork 框架的调用接口 ActionProxy 中，我们将不再看到任何与 Web 容器相关的对象

因此，Dispatcher 是真正将 Http 请求与 MVC 实现（XWork 框架）分离的核心分发器。而 Dispatcher 的逻辑，又被分散到 PrepareOperations 和 ExecuteOperations 这两个代理类中去执行调度，从而保证了程序逻辑的扩展性。到此为止，Struts2 的工作已经完成，所有剩下的 Http 请求的处理工作将移交给 XWork。

10.3 第二战场 —— XWork 处理阶段

进入第二战场，似乎所有的一切都已经在我们的掌握之中。因为有关 XWork 对事件处理流程，我们在讲解 XWork 时已经为大家做过深入的分析。在这里，我们再次探寻整个过程，把在之前的分析讲解中没有强调的部分再度挖掘一下。

10.3.1 执行控制权的移交

核心分发器 Dispatcher 中的 serviceAction 方法（也就是我们在上一节中分析过的源码，见代码清单 10-10）是核心分发器 Dispatcher 进行 Http 处理的场所。也正是在这个方法中，Struts2 将处理执行权转交给 XWork。

由于 ActionProxy 是 XWork 框架的入口，对整个 XWork 的运行细节起到了一个代理作用，因而我们也可以从源码中发现，Struts2 的执行权移交过程实际上围绕着 ActionProxy 来进行，并且采用了 Struts2 一贯的伎俩：先初始化，再执行逻辑。

ActionProxy 的执行逻辑并不复杂，因为我们知道 ActionProxy 只是 XWork 的一个门户，真正在其中负责复杂的逻辑调度的是 ActionInvocation。因而我们在这里将关注的重点放在 ActionProxy 的初始化过程。这一个过程，我们也称为 **XWork 执行环境的初始化**。

一旦使用了 XWork 执行环境这个称谓，整个研究的范围似乎一下子就变大了。虽然 ActionProxy 的初始化过程在代码清单 10-10 所示的源码中只有一行代码就完成了：

```
ActionProxy proxy =
config.getContainer().getInstance(ActionProxyFactory.class).createActi
onProxy(namespace, name, method, extraContext, true, false);
```

事实上，整个初始化过程却是非常复杂的。因为仅仅从外在的表现形式来看，这里主要针对的是 ActionProxy 和 ActionInvocation 的初始化工作，而实际在其内部的实现中却蕴含了整个 XWork 控制流元素和数据流元素的初始化过程和关系设定。

接下来，我们就分别从 ActionProxy 的初始化数据源以及 ActionProxy 中核心元素 ActionInvocation 的初始化过程这两个角度，对 XWork 执行环境的初始化过程进行源码解析。

10.3.1.1 ActionProxy 的数据源

我们曾经在第 8 章中对 ActionProxy 的初始化有过一个粗略的讲解。当时的讲解主要围绕着 createActionProxy 方法的参数展开，我们在这里可以回过头来再看一下这个参数列表：

❑ **配置关系映射**——namespace、actionName、methodName 等

❑ **运行上下文环境**——extraContext

这两类参数，构成了 XWork 执行环境的数据基础。因而我们非常有必要研究一下这两类参数的来源。

第一类参数，我们称之为配置关系映射。其中主要包含 namespace、actionName、methodName 这三个参数。从代码清单 10-10 所示的源码中，我们可以看到这三个参数都是来源于 ActionMapping 对象。如果继续追溯其源头，我们就会发现这类表示 ActionProxy 与具体 XWork 事件执行框架之间映射关系的参数来源于 Http 请求处理的第一阶段的最后一个步骤：**使用 ActionMapper 根据 Http 请求查找 ActionMapping 对象。**

第二类参数，我们称之为运行上下文环境。我们同样从代码清单 10-10 所示的源码中追溯其源头，发现它实际上来源于 Dispatcher 中的 createContextMap 方法。而这个方法，我们曾经在 ActionContext 的创建过程中已经分析过。由此，我们可以得出结论：

结论　ActionContext *在创建时所依赖的上下文环境与* ActionProxy *初始化时所依赖的上下文环境是等同的。*

得出这个结论，我们并不意外。因为 ActionProxy 的初始化任务之一就是构建起 XWork 的执行环境，而 ActionContext 作为 XWork 的数据流元素当然也是 XWork 的大执行环境的一部分。这一点，我们可以从 extraContext 变量在后续的初始化过程中的调用看得出来。相关源码位于 ActionInvocation 的 createContextMap 中，如代码清单 10-11 所示。

代码清单 10-11　DefaultActionInvocation.java

```
protected Map<String, Object> createContextMap() {
        Map<String, Object> contextMap;

        if ((extraContext != null) &&
(extraContext.containsKey(ActionContext.VALUE_STACK))) {
            // 当 ValueStack 已经通过 extraContext 传入时
            // 直接使用 ValueStack 的上下文环境
            stack = (ValueStack)
extraContext.get(ActionContext.VALUE_STACK);

            if (stack == null) {
                throw new IllegalStateException("There was a null Stack set
 into the extra params.");
            }

            contextMap = stack.getContext();
        } else {
            // 当 ValueStack 没有通过 extraContext 传入，重新创建一个 ValueStack
            stack = valueStackFactory.createValueStack();
```

```
            // create the action context
            contextMap = stack.getContext();
        }

        // 将 extraContext 中所有的内容置入上下文环境中
        if (extraContext != null) {
            contextMap.putAll(extraContext);
        }

        // 将 ActionInvocation 和 Container 也置入上下文环境中
        contextMap.put(ActionContext.ACTION_INVOCATION, this);
        contextMap.put(ActionContext.CONTAINER, container);

        return contextMap;
    }
```

从上述代码中我们可以看到，ActionInvocation 在初始化时，对 ActionContext 的上下文环境有一个再次检查并重置的过程（这里主要考虑到的是非 Struts2 作为其外部调用，XWork 独立被使用成为一个请求处理框架的情况）。而在整个过程中，extraContext 成为其最为核心的数据基础，因为 extraContext 中所有的内容都被设置到了上下文环境中。除此之外，ActionInvocation 仅仅额外地将容器对象和 ActionInvocation 对象本身设置到上下文环境中去而已。

10.3.1.2 ActionInvocation 的初始化

ActionInvocation 的初始化过程由 ActionProxy 驱动完成。事实上，由于 ActionProxy 只是一个对外门户，ActionInvocation 作为 XWork 内部的核心调度元素，它必须持有当前请求过程中所有需要被调度的执行元素的控制权。而对于控制权的把控过程，从 ActionInvocation 的初始化就开始了。

ActionInvocation 的核心初始化方法，如代码清单 10-12 所示。

代码清单 10-12 DefaultActionInvocation.java

```
public void init(ActionProxy proxy) {
        this.proxy = proxy;

        // 创建上下文环境，这里的 contextMap 与 ActionContext 的上下文环境一致
        Map<String, Object> contextMap = createContextMap();

        // 将 ActionInvocation 对象设置到 ActionContext 中
        // 这样做的好处在于可以利用 ActionContext 的数据共享特性，
        // 将 ActionInvocation 在整个执行周期共享
        ActionContext actionContext = ActionContext.getContext();
```

```
        if (actionContext != null) {
            actionContext.setActionInvocation(this);
        }

        // 创建 Action 对象
        createAction(contextMap);

        // 将 Action 对象置于 ValueStack 中
        // 这就是将 XWork 的数据流元素与控制流元素进行融合的关键步骤
        if (pushAction) {
            stack.push(action);
            contextMap.put("action", action);
        }

        // 构建 ActionInvocation 的上下文环境
        // 将 ActionInvocation 的上下文环境与 ActionContext 等同起来
        invocationContext = new ActionContext(contextMap);
        invocationContext.setName(proxy.getActionName());

        // 获取拦截器堆栈，并将拦截器堆栈的迭代器指针处于栈顶位置
        List<InterceptorMapping> interceptorList = new
ArrayList<InterceptorMapping>(proxy.getConfig().getInterceptors());
        interceptors = interceptorList.iterator();
    }
```

这段源码的内容就相当丰富了。我们将其自上而下地划分为若干个步骤：

- 步骤 1——创建上下文环境
- 步骤 2——ActionInvocation 对象的共享
- 步骤 3——创建 Action 对象
- 步骤 4——将 Action 对象置入 ValueStack 中
- 步骤 5——创建 ActionInvocation 的上下文环境
- 步骤 6——将拦截器堆栈置于初始调度状态

在这其中，绝大多数的步骤都围绕着 XWork 的控制流元素的初始化展开。比如，步骤 1 的创建上下文环境，我们已经在之前的分析中提到过，它与 ActionContext 的上下文环境是一致的，并成为步骤 3 的创建 Action 对象和步骤 5 的创建 ActionInvocation 上下文环境的数据基础。再比如步骤 3 和步骤 6，直接完成对 Action 对象和 Interceptor 状态的初始化设置。

我们在这里需要强调的是步骤 4：**将 Action 对象置入 ValueStack 中**。这个步骤的重大意义，我们曾经在第 8 章讲到 XWork 中数据流和控制流这两股驱动力的融合关系时作为一个核心结论反复强调过。在这里，我们终于通过源码证明了这个结论的正确性。这个步骤不仅将两个毫无关系的程序驱动力捏合在一起，它也同时成为 Struts2 中数据访问的

基础。有关这一点，我们将在下一章加以说明。

10.3.2 ActionInvocation 调度的再分析

在第 8 章中，我们对 ActionInvocation 中的核心调度方法 invoke 的执行流程进行了深入分析。invoke 方法中 ActionInvocation 与 Interceptor 对象之间的递归嵌套调用给我们留下了很深的印象，而这一递归调用的过程也成为 XWork 实现 AOP 编程的基础。

在这里，我们再次谈及 invoke 方法时，观察角度将发生一点改变。在第 8 章中，我们的研究重点在于 Action、Interceptor 之间的调度关系和执行顺序，因为需要强调的是“调度”这两个字。在这里，我们将具体来看看每个控制流元素在各自执行的过程中有什么值得挖掘的地方。invoke 方法的源码，如代码清单 10-13 所示。

代码清单 10-13 DefaultActionInvocation.java

```
        public String invoke() throws Exception {
            String profileKey = "invoke: ";
            try {
                UtilTimerStack.push(profileKey);

                // 首先判断 ActionInvocation 的执行状态，如果已经执行过，则抛出异常
                if (executed) {
                    throw new IllegalStateException("Action has already executed");
                }

                // 对所有的 Interceptor 对象进行调度
                // 所有的 Interceptor 在 ActionInvocation 中被有序地置于一个迭代器中
                // 对 Interceptor 对象的调度，实际上是对迭代器的遍历过程
                if (interceptors.hasNext()) {
                    final InterceptorMapping interceptor = (InterceptorMapping)
     interceptors.next();
                            String interceptorMsg = "interceptor: " + interceptor.
getName();

                    UtilTimerStack.push(interceptorMsg);
                    try {
                            // 这里是整个 ActionInvocation 调度的核心
                            // 将 ActionInvocation 的实现类作为参数传入 Interceptor 执行
                            // 结合拦截器的实现代码就会发现，这里蕴含了一个递归调用
                            resultCode =
    interceptor.getInterceptor().intercept(DefaultActionInvocation.this);
                    }
                    finally {
                        UtilTimerStack.pop(interceptorMsg);
                    }
                } else {
```

```
                // 如果执行栈中没有 Interceptor 对象，直接执行 Action 对象
                resultCode = invokeActionOnly();
            }

            // Interceptor 和 Action 的调度完毕，执行 PreResultListener 逻辑
            if (!executed) {
                if (preResultListeners != null) {
                    for (Object preResultListener : preResultListeners) {
                        PreResultListener listener = (PreResultListener)
preResultListener;

                        String _profileKey = "preResultListener: ";
                        try {
                            UtilTimerStack.push(_profileKey);
                            listener.beforeResult(this, resultCode);
                        }
                        finally {
                            UtilTimerStack.pop(_profileKey);
                        }
                    }
                }

                // 最后执行 Result 对象的逻辑
                if (proxy.getExecuteResult()) {
                    executeResult();
                }

                executed = true;
            }

            return resultCode;
        }
        finally {
            UtilTimerStack.pop(profileKey);
        }
    }
```

再次回到这段源码，我们将不再详细展开 Interceptor 的调度过程，而是重点来看看 Action 对象的调度。根据之前对 Interceptor 对象调度的分析，我们就很容易理解为什么在代码中，Action 的调度与 Interceptor 的调度分别位于一个互斥的 if / else 逻辑块里。因为 Action 位于执行栈的底部，只有当位于 Action 之上的 Interceptor 对象全部执行完毕之后，Action 对象才会被调度执行。这也就是 interceptors.hasNext() 这个逻辑判断语句在这里的含义。

让我们来看看 invokeActionOnly 方法具体做了什么，其源码如代码清单 10-14 所示。

代码清单 10-14 DefaultActionInvocation.java

```
    public String invokeActionOnly() throws Exception {
        return invokeAction(getAction(), proxy.getConfig());
    }

    protected String invokeAction(Object action, ActionConfig actionConfig)
    throws Exception {
            String methodName = proxy.getMethod();

            if (LOG.isDebugEnabled()) {
                LOG.debug("Executing action method = " +
    actionConfig.getMethodName());
            }

            String timerKey = "invokeAction: " + proxy.getActionName();
            try {
                UtilTimerStack.push(timerKey);

                boolean methodCalled = false;
                Object methodResult = null;
                Method method = null;
                try {
                    method = getAction().getClass().getMethod(methodName,EMPTY_
    CLASS_ARRAY);
                } catch (NoSuchMethodException e) {
                    try {
                            String altMethodName = "do" + methodName.substring(0,
1).toUpperCase() + methodName.substring(1);
                        method = getAction().getClass().getMethod(altMethodName,
EMPTY_CLASS_ARRAY);
                    } catch (NoSuchMethodException e1) {
                        //
                        if (unknownHandlerManager.hasUnknownHandlers()) {
                            try {
                                methodResult =
    unknownHandlerManager.handleUnknownMethod(action, methodName);
                                methodCalled = true;
                            } catch (NoSuchMethodException e2) {
                                throw e;
                            }
                        } else {
                            throw e;
                        }
                    }
                }

                if (!methodCalled) {
                    methodResult = method.invoke(action, new Object[0]);
```

```
            }

            if (methodResult instanceof Result) {
                this.explicitResult = (Result) methodResult;

                // 直接构造 Result 对象
                container.inject(explicitResult);
                return null;
            } else {
                return (String) methodResult;
            }
        } catch (NoSuchMethodException e) {
            throw new IllegalArgumentException("The " + methodName + "() is
not defined in action " + getAction().getClass() + "");
        } catch (InvocationTargetException e) {
            Throwable t = e.getTargetException();

            if (actionEventListener != null) {
                String result = actionEventListener.handleException(t,
getStack());
                if (result != null) {
                    return result;
                }
            }
            if (t instanceof Exception) {
                throw (Exception) t;
            } else {
                throw e;
            }
        } finally {
            UtilTimerStack.pop(timerKey);
        }
    }
```

洋洋洒洒的代码，除了对 Action 中具体执行方法的一些逻辑判断之外，竟然没有什么更加复杂的逻辑了！ Action 执行的本质居然如此简单，仅仅是采用 Java 中最普通的反射技术而已！

如此简单的调用，反而给了我们重新考察 Action 执行方法的空间。在此，我们有两个不同的观察角度：

- **执行方法的参数**——没有参数
- **执行方法的返回值**——String 或者 Result 对象

这两个观察角度，实际上也是许多程序员对 Struts2 产生诟病的两个重要的方面。

第一，我们都知道响应方法中的参数是表现请求数据的最佳编程元素。然而在这里，Struts2 却“死板地”使用无参数的响应方法。这当然与 Struts2 本身对于请求－响应的实

现模式有关（因为 Struts2 所采用的是 POJO 模式，请求参数被封装在 Action 的属性成员变量中），不过这毕竟不够直观、不够 OO。因为从接口的逻辑语义角度，一个没有参数的接口并不能反映出一个请求 – 响应过程的全部。

第二，传统 Action 接口中的返回值是 String，表示 Action 的执行结果状态码。这个状态码，同时也成了 Result 对象的寻址依据。这里的问题在于我们是否一定需要通过一个复杂的由状态码到 Result 对象的中转过程呢？因而，Struts2 也同时支持显式的 Result 对象被加入到所支持的执行方法返回值中。不过这也带来了另外一个问题：我们不得不在 Action 的业务逻辑处理过程中手工处理 Result 对象的创建工作。

正所谓世界上本就没有十全十美的东西。我们对于 Struts2 的 Action 对象执行方法的思考和质疑，实际上提供给读者一个重新认识框架设计的思维方式。希望大家能够仔细地思考这些问题，并形成自己的观点，从而加强自身对各种程序的设计能力。

10.4 小结

本章的内容有炒冷饭之嫌。不过应该说，这个冷饭炒得很值！虽然在本章之前，我们已经对 Struts2 的设计理念、XWork 的实现机理了如指掌，但是作为一个完整的运行体系，我们直到本章才真正将它们串联起来。

从本章的结构来看，我们还是采取了先总后分的方式，首先对运行主线的宏观体系加以分析，再针对不同的执行阶段，分别对每个阶段的完整执行过程进行源码分析。不过对于读者来说，要掌握整个过程的重点，还是在于领悟本章一开始所给出的那张调用时序图，这是整个 Struts2 的核心。

通过本章的阅读，大家对以下问题是否有了更加深刻的认识呢？

- ❑ Struts2 的 Http 请求处理主线主要分为哪两个阶段？
- ❑ PrepareOperations 和 ExecuteOperations 有什么作用？
- ❑ Struts2 是如何进行 URL Mapping 查找的？
- ❑ 为什么说核心分发器 Dispatcher 是整个 Struts2 的中转站？
- ❑ ActionProxy 在初始化的时候有哪两大数据来源？
- ❑ ActionInvocation 的初始化有哪些步骤？
- ❑ Action 的执行方法从语法构成上看有什么特点？

第 11 章　展翅高飞 —— 让视图放开手脚

MVC 框架中的视图（View）层，是框架与外界进行交互的接口。如果把一个 MVC 框架比作是一份礼物，视图就是我们所能直接接触到的外包装。外包装的精致和优美程度将直接从感官上影响我们对礼物好坏的评判，所以视图的重要性不言而喻。

之前，我们还有一个关于计算机组成结构的比喻，就是把 MVC 框架中的 Controller 层比作是计算机五大部件中的 CPU（控制器和运算器的合体）。从这个角度来观察视图，它就自然而然地成为计算机部件中的输入设备和输出设备。这一比喻也体现出视图层最重要的职责特性：**对外交互**。

在本章中，我们将首先对视图的本质进行思考和分析，并以此为基础向读者介绍在整个视图层所涉及的一些技术以及这些技术如何诠释视图层的交互特性，从而揭开这层华丽的“外包装”的真正面纱。

11.1　视图（View）概述

为了探寻视图层蕴含的秘密，我们首先从技术的角度来分析一下在 Java 世界中实现视图的主要手段：JSP 技术和 Servlet 技术，并试着从中找到视图的一些共性特征和本质内容。

11.1.1　视图表现技术

常用的视图表现技术有 JSP 技术和 Servlet 技术。

11.1.1.1　JSP 技术

我们知道，浏览器上的主要编程语言是 HTML。HTML 的主要功能，是构建浏览器可识别的页面元素，从而将这些元素以完美的“视图”形式展现给用户，这或许也是视图一词的本意。不过我们知道，HTML 语言是一种静态语言，它自身**缺乏数据沟通的能力**。也就是说，HTML 语言需要另外一种机制的帮助才能完成与服务器端程序的沟通和逻辑控制。

在这种情况下，不同的编程语言建立在 HTML 语言之上，对其进行了增强。例如，微软创立了 ASP 技术、受到脚本语言的影响而产生了 PHP 技术。而在 Java 世界，则对应产生了 JSP 技术。从 JSP 的命名来看，Java Server Page 的定义非常直观：**允许在构成 Page 的**

HTML 语言之上嵌入 Java 的语法片段，从而加强其与 Server 的交互能力。在这里，我们对 JSP 的解读并非是官方定义，而是为了帮助读者理解而将 JSP 进行了构词拆解。

但是无论如何，JSP 的引入从两个方面解决了构建视图的主要问题：

- **解决了页面视图的构建问题**——以 HTML 代码作为构建基础
- **解决了与服务器端进行数据沟通的问题**——以 Java 语法作为沟通基础

这两个方面，不仅是视图层构建的主要问题，也是视图作为一个独立的 MVC 层次的最重要的职责。有关这一点，我们在之后的分析中还将详细展开。

为了帮助广大程序员更好地解决上述两大问题，在 JSP 标准的实现中，还陆续引入了一些辅助的语法手段。例如，标签库（Taglib）技术、EL（表达式语言）技术等等。当然，这些技术往往是 JSP 在形式上所做出的重大改进。从本质上讲，JSP 的两大核心语义：构建页面展现和构建数据逻辑通信始终没有改变。

JSP 技术是很多程序员最为熟悉的 J2EE 标准之一。许多程序员学习 Web 开发，都是将 JSP 作为最基础的课件来学习的。所以，有关 JSP 的实现细节，并不在我们这里的讨论之列。读者需要明确的是 JSP 的构成要件以及它们分别承担的职责。

11.1.1.2 Servlet 技术

Servlet 技术是一个比 JSP 技术出现得更早的视图表现技术。在这里不对这些规范产生的历史做过多的纠缠，不过我们还是可以从运行的角度对 JSP 和 Servlet 之间的关系进行一番阐述，从而帮助读者领略 Servlet 技术的本质：

结论 *所有的 JSP 在运行期都被编译成 Servlet 在 Web 容器中运行。*

原来，JSP 的本质是 Servlet！换句话说，Servlet 规范才是我们所要探寻的实现视图层的底层规范。在之前的章节与 Servlet 规范已经多次打过交道，在这里我们不妨再来看看 Servlet 规范中所指定的一些接口方法，如图 11-1 所示。

图 11-1 HttpServlet 的接口方法

从图中，我们可以非常清楚地看到在 Servlet 规范中定义的所有接口，几乎都以 HttpServletRequest 和 HttpServletResponse 这两个 Web 容器对象作为接口参数。很显然，Servlet 的接口实现模式，正是我们在之前的章节中所提过的参数 - 参数模式。

鉴于此，Servlet 接口实现中最重要的就是构成其接口的主要参数：HttpServletRequest 和 HttpServletResponse 对象。因而我们也有必要对这两个对象在 Servlet 规范中所起的重要作用做一些更加深入的分析。

结论　HttpServletRequest 对象主要用于处理整个 Http 生命周期中的数据。

这个结论，似乎与 HttpServletRequest 这个对象的名称不太符合。因为所谓 Request，肯定是更加偏向“请求”这个动作。而在上述的结论中，我们不仅把 HttpServletRequest 的作用范围扩大到整个 Http 的生命周期，还为它做了一个功能上的定义：**处理数据**。这一点，我们可以从 HttpServletRequest 的主要接口定义中看出来，其相关源码如代码清单 11-1 所示。

代码清单 11-1　HttpServletRequest

```
/**
 *
 * @return  返回一个不变的 java.util.Map 实例，包含所有以参数的名称作为 Key 值、参
 *  数实际值作为 Value 值的键值对。 Key 为 String 类型，Value 为 String[] 类型
 *
 */
public Map getParameterMap();

/**
 *
 * @return  返回根据参数名称返回参数的实际值，该名称的参数值可能有多个，以字符串数组
 * 的形式返回
 *
 */
public String[] getParameterValues(String name);

/**
 *
 * @return  返回根据参数名称返回参数的实际值，该名称的参数值可能有多个，以逗号隔开的
 * 形式返回一个字符串；如果该名称的参数值只有一个，则直接返回一个字符串
 *
 */
public String getParameter(String name);

/**
 *
 * @return  返回根据 attr 名称返回 Http 生命周期中 attr 所对应的对象。这个方法是
```

```
     *  HttpServletRequest 对象在一次请求生命周期中进行对象访问的主要方法
     *
     */
    public Object getAttribute(String name);

    /**
     *
     * @return  根据 attr 名称和 value 参数，在一次请求的生命周期内进行对象存储。这个
     * 方法是 HttpServletRequest 对象在一次请求生命周期中进行对象存储的主要方法
     *
     */
    public void setAttribute(String name, Object value);
```

在上述接口中，getParameter 系列的方法主要用于处理“请求数据”，**是服务器端程序获取浏览器所传递参数的主要接口**。而 getAttribute 和 setAttribute 方法则向我们展示了 HttpServletRequest 对象如何在一次请求生命周期中完成对数据值的存储和访问的管理过程。

结论 HttpServletResponse 对象主要用于处理 Http 的响应结果。

什么是 Http 的响应结果呢？我们还是用代码加以说明。例如，我们可以指定某个 Http 的响应结果为转向到一个 JSP 页面，其相关源码如代码清单 11-2 所示。

代码清单 11-2 dispatcher 转向示例代码

```
RequestDispatcher dispatcher = request.getRequestDispatcher("/idx.jsp");
dispatcher.forward(request, response);
```

我们知道，Http 响应的种类很多，上述代码所展示的是被称为 Forward 的跳转，其主要作用是将浏览器的输出转向为一个 JSP 页面。当然，还有重定向 Redirect 的方式，甚至还有直接通过“流”（Stream）的形式输出到浏览器的方式。其相关源码如代码清单 11-3 所示。

代码清单 11-3 redirect 转向和 stream 输出示例代码

```
response.sendRedirect("www.google.com");


OutputStream oOutput = response.getOutputStream();
// 将 inputStream 复制到 outputStream
byte[] oBuff = new byte[bufferSize];
int iSize;
while (-1 != (iSize = inputStream.read(oBuff))) {
```

```
        oOutput.write(oBuff, 0, iSize);
    }
```

我们可以看到，身为底层的技术规范，HttpServletRequest 和 HttpServletResponse 这两个对象相互配合共同完成了视图的核心语义。有了这些知识作为基础，我们就可以总结出视图的本质了。

11.1.2　视图的本质

什么是视图的本质？在了解了主要的视图表现技术之后，我们为什么还要探寻视图的本质呢？这里面最主要的原因在于我们需要借助对视图本质的研究，找到在 Web 框架中实现视图层的途径。

回顾一下之前对请求 - 响应模式实现机理的讨论，我们就会发现之前所有讨论的侧重点都是围绕着**服务器端**进行的，无论是编程元素的抽象还是请求 - 响应的实现模式都直接指向服务器端的编程规范。而对于客户端的请求发起方（Request）以及同为客户端的结果响应方（Response），我们却少有涉及。这主要是由于在 B/S 模式下，浏览器的行为似乎并不在整个 Web 框架的研究范畴内。

如果把整个请求 - 响应过程再仔细进行一番梳理，我们就会发现浏览器在请求 - 响应模式中所起的作用却是举足轻重的。因为在 B/S 模式下，浏览器既是 Http 请求发起方，同时也是服务器端逻辑处理结果的呈现方。此时，如果把 MVC 框架中视图层的交互职责与浏览器联系起来，我们可以发现：

结论　*浏览器是视图层的表现载体，所有视图层的交互职责都是通过浏览器进行的操作。*

之前已经分析过典型的人机交互过程的原理。结合上面的结论，我们再来回顾一下浏览器与服务器端程序交互的三大要素：

- **沟通协议**——Http 协议
- **请求内容**——Http 请求：HttpServletRequest
- **响应内容**——Http 响应：HttpServletResponse

在这里，我们将第 7 章中提到的人机交互模式的三大要素做了进一步的升华：将请求内容和响应内容与之前所介绍的 Servlet 技术中的两大主要对象结合在了一起。不过这一升华，反而为我们指出了视图层的交互本质：

结论　*视图的本质是 Web 容器对象 HttpServletRequest 和 HttpServletResponse 对浏览器行为的控制。*

读者对于这个结论必须非常明确。因为我们知道，视图是整个 Web 开发中变化最大的一个层次，而视图的本质却是纷繁复杂的视图表现中唯一“万变不离其宗”的东西。而这个结论也成为我们推导视图层主要职责的重要依据。

11.1.3 视图的职责

在讨论了视图的本质之后，我们再来看看视图的职责。事实上，从讲 JSP 开始，视图的职责就已经以特殊的形式展现在我们的面前：

❑ **视图内容呈现**——以 HTML 为基础进行的构建

❑ **数据和逻辑呈现**——以 Java 语法为基础进行的构建

这两大职责，是我们站在 JSP 语法构成的角度所得出的结论。以 HTML 为基础所进行的构建之所以被称为**内容呈现**，主要原因在于 HTML 语言本身就是一个浏览器上的编程语言，内容呈现是 HTML 语言所带来的职责语义。而以 Java 语法为基础进行的构建被称为数据和逻辑呈现，主要原因在于 HTML 语言所缺乏的与服务器端进行数据沟通和逻辑控制的功能恰巧能由 Java 语言的处理能力来弥补。

如果站在 Servlet 规范的角度，我们同样能够得出上述结论。因为从 HttpServletRequest 对象和 HttpServletResponse 对象的作用来看（前者负责数据处理，后者负责内容呈现），两个对象只有相互配合，才能共同完成完整的视图表现。

接下来，问题就来了：既然视图的职责如此清晰明了，视图层没有理由仅仅将 JSP 或者 Servlet 作为其唯一的表现形式。因为就浏览器而言，它并不关心究竟以什么样的方式输出内容。因此，我们应该站在一个更高的高度来审视整个视图层：**只要实现了视图的两大职责，它就能称为一个“合格”的视图表现技术**。如此一来，各种各样的视图表现技术就如雨后春笋般涌现出来。

在这其中，有一些重要的视图表现技术非常值得我们注意。

❑ **模板技术**

模板技术是一个独立的内容展现技术。通过事先定义好的模板和数据模型的整合，模板技术能够非常自由地将内容呈现发挥到极致。由于模板技术可以在模板中定义属于模板自身的语法，因而从各个方面讲都具备了极大的可扩展性和可操作性。因而，模板技术相比 JSP 而言，越来越成为广大程序员所青睐的一种视图表现技术。

❑ **AJAX 技术**

AJAX 技术严格来说并不能称之为一个视图表现技术，然而它却是近些年来发展最为蓬勃的表示层技术之一。AJAX 技术所带来的对视图表示的最重要的影响就是通过 JavaScript 操作 HTML 的 DOM 节点来进行视图输出的控制。它也逐渐成为视图表现的一个重要选择方向。不过我们可以发现，使用 JavaScript 来进行视图输出的控制实际上是把

视图呈现的职责转移到了客户端编程的范围，这是一种“职责转移”。

❑ Flex 技术

Flex 技术突破了以传统的 HTML 来构建页面内容的原则，使用基于 Flash 的技术来进行内容呈现。这一革命无疑比使用 JavaScript 来进行视图输出的控制更为彻底。不过虽然页面内容的呈现基础不再是 HTML，但是 Flex 技术中的数据交互以及内容展现的本质却没有任何改变。

因此，**视图的表现形式是多样化的**。不过无论是什么样的视图表现技术，无论它们的外在表现形式如何，从本质上来说它们都是对浏览器行为的不同控制方式，它们拥有相同的视图职责：**内容呈现和数据呈现**。

11.2　深入 Result 机制

Result 是 XWork 框架中的元素，在第 7 章和第 8 章讲 XWork 框架时曾经多次提到过这个对象，不过我们始终没有将 Result 作为一个独立的元素进行讲解。这与 Result 对象的特殊性有着很大的关系，那么 Result 对象与 XWork 框架中的其他对象有什么不同呢？对于 Result，我们需要从不同的角度来理解。

11.2.1　Result 的不同视角

11.2.1.1　XWork 视角——事件处理节点

Result 对象既然是由 XWork 提出的，那么我们就首先来看看站在 XWork 框架的视角，Result 对象是一个什么样的元素呢？

在之前对 XWork 控制流元素的分析中，我们将整个 XWork 的控制流元素划分为事件处理节点和事件处理驱动元素这两大类。其中，Result 对象与 Action 和 Interceptor 一起，被划到事件处理节点这一大类中。

因此，**站在 XWork 的视角来看 Result 对象，它只是我们对事件处理流程步骤的划分结果而已**。Result 处于整个事件处理流程的末端，是 Action 和 Interceptor 对象在经过了复杂的调度执行完毕之后所调用的一个为整个事件进行“收尾”的逻辑处理元素。

从职责上来看，我们之前有一个有关战斗序列的比喻：将 Action 比喻为主力部队，而将 Interceptor 比喻为策应部队。也就是说，在整个控制流元素的执行过程中，Action 负责核心的逻辑处理；而 Interceptor 则为 Action 保驾护航，成为 Action 的好帮手。那么 Result 对象在这里又可以比喻为什么呢？实际上，作为整个事件处理流水线的最后一步，**Result 对象为战斗部队的下一步行动指明了方向**。从这个比喻中，我们看到 Result 对于程

序控制权的把握。正是这样的把握，使得 Result 在整个 XWork 控制流中的重要性体现出来了。因为**它将告诉程序该何去何从，成为在整个 XWork 控制流元素中实施控制职责的重要体系元素**。

同样作为事件处理节点，Result 对象也是 ActionInvocation 进行调度的控制流元素之一。始终站在事件处理节点的观点来解释 Result，我们会发现它只是 XWork 对事件处理流程的合理抽象。这一抽象所带来的结果，不仅为事件处理流程画上了一个完美的句号，同时也完善了整个事件处理的体系。

11.2.1.2 Struts2 视角——视图的操作窗口

如果从 Struts2 的角度来看待 Result 对象，我们会产生一些截然不同的印象。这一点，我们首先从 Result 对象的其中一个实现类 ServletDispatcherResult 的源码谈起，其相关源码如代码清单 11-4 所示。

代码清单 11-4 ServletDispatcherResult.java

```
HttpServletRequest request = ServletActionContext.getRequest();
HttpServletResponse response = ServletActionContext.getResponse();
RequestDispatcher dispatcher =
request.getRequestDispatcher(finalLocation);

  // 这里省略了许多其他的代码

  // 如果视图不存在，直接转到 404 状态
  if (dispatcher == null) {
      response.sendError(404, "result '" + finalLocation + "' not found");
      return;
  }

  // 判断是否存在 action 的 tag 标签，成为 dispatcher 选择 include 或 forward 的标志
  Boolean insideActionTag = (Boolean)
ObjectUtils.defaultIfNull(request.getAttribute(StrutsStatics.STRUTS_
ACTION_TAG_INVOCATION), Boolean.FALSE);


   // 根据 include 或 forward 的选择，进行 dispatcher 方法调用
  if (!insideActionTag && !response.isCommitted() &&
(request.getAttribute("javax.servlet.include.servlet_path") == null)) {
      request.setAttribute("struts.view_uri", finalLocation);
      request.setAttribute("struts.request_uri",
request.getRequestURI());

      dispatcher.forward(request, response);
  } else {
      dispatcher.include(request, response);
  }
```

上述代码我们非常熟悉。在本章一开始我们讲 Servlet 技术时就看到过其中的核心代码实现。这一视图跳转 Forward 的逻辑片段，在 Result 的实现类中出现，是否是一种巧合呢？我们不妨再来看看另外一个 Result 的实现类：StreamResult，其相关源码如代码清单 11-5 所示。

代码清单 11-5　StreamResult.java

```
    OutputStream oOutput = null;

    try {
        if (inputStream == null) {
           // 从 ValueStack 中查找需要输出的 inputstream
           inputStream = (InputStream)
    invocation.getStack().findValue(conditionalParse(inputName,
    invocation));
        }

        if (inputStream == null) {
            throw new IllegalArgumentException(msg);
        }

        // 获取 HttpServletResponse 对象
          HttpServletResponse oResponse = (HttpServletResponse) invocation.
getInvocationContext().get(HTTP_RESPONSE);

        // 这里省略了许多其他的代码

        // 获取 outputstream
        oOutput = oResponse.getOutputStream();


        // 从 inputStream 复制到 outputStream
        byte[] oBuff = new byte[bufferSize];
         int iSize;
         while (-1 != (iSize = inputStream.read(oBuff))) {
              oOutput.write(oBuff, 0, iSize);
         }

          // Flush 到浏览器
          oOutput.flush();
      }
      finally {
          if (inputStream != null) inputStream.close();
          if (oOutput != null) oOutput.close();
      }
```

又是我们熟悉的代码！看上去 Result 把我们在 Servlet 中 HttpServletResponse 对象的

职责全都揽过来了。只是根据不同的视图类型，Result 采用了不同的实现类进行区分。这不正是一个典型的策略模式的应用吗？

我们可以回过头来把整个 Result 接口的实现结构展示出来，如图 11-2 所示。

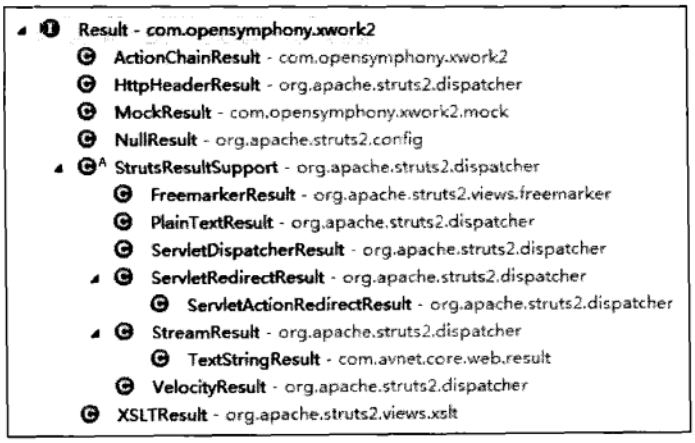


图 11-2 Result 接口的实现结构

大家是否能从这幅 Result 接口的实现结构示意图中发现什么重要的特点呢？原来，Result 接口虽然是一个 XWork 的定义接口，但是其绝大多数实现类都是 Struts2 的实现。**接口与实现在这里真正做到了在框架级别的解耦合！**

从上述分析中，我们可以得出一个重要结论：

结论 Result 对象在 Struts2 中被看作是视图的操作窗口，相当于完成了 Servlet 对象中 HttpServletResponse 的职责。

这个结论，就是我们站在 Struts2 的视角对 Result 对象的理解。由于使用了策略模式，Result 对象通过接口和实现类的分离，得以将其接口的逻辑意义保留在 XWork 中，而将其实现类作用于 Struts2 的执行体系。这可谓是整个 Struts2 设计中又一经典的手笔。

11.2.2 Result 职责分析

回到 Result 对象职责的话题，当我们站在不同的视角对 Result 对象进行了一番分析之后，我们不难从整个分析的过程中得到有关 Result 职责的结论：

结论 Result 最大的职责，就是架起 Controller 层到 View 层的桥梁。

那么，我们为什么要设立这样一个对象来架起从 Controller 层到 View 层的桥梁呢？

主要原因不外乎三个：

❑ **需要一个起到中转作用的对象，完成程序执行权的转移**

当我们站在 XWork 的视角看待 Result 对象时，就曾经指出 Result 对象具有程序逻辑执行权的控制力。这种控制力，正是 XWork 将其纳入事件处理节点的重要原因。从效果上看，Result 作为一个中转对象，也完成了程序逻辑由 XWork 向 Struts2 的移交。

❑ **需要一个接口与实现分离的机制，将 XWork 与 Struts2 进行解耦合**

当我们站在 Struts2 的视角看待 Result 对象时，我们发现 Result 机制是一个典型的策略模式的应用。而接口与实现分离的结果，是两大框架的彻底解耦合。这也是使用设计模式来实现 Struts2 核心理念的一种方式。

❑ **需要一个 Web 容器的代理对象，屏蔽底层对象的操作细节**

从之前展示的两个 Result 对象的实现类中，我们可以看到 Result 实现类的主要逻辑，就是通过操作 HttpServletResponse 对象完成对 Http 的响应。有了 Result 机制，它就成为了一个极佳的代理类，我们就大可不必操心任何与 Web 容器打交道的过程细节了。

在分析完“为什么”后，我们再来看看“怎么样”。具体来说，就是 Result 对象在实现上到底有哪些主要的内容。这些内容也恰巧体现在 3 个不同的方面：

❑ **封装跳转逻辑**

❑ **准备显示数据**

❑ **控制输出行为**

有关这三个方面的内容，我们在这里不再详细展开。读者可以顺着这三个方面的内容在图 11-2 中所指出的所有 Result 实现类中找到答案。

11.3　标签库，永恒的争论话题

标签库（Taglib），几乎是每个 Web 框架的重要组成部分。翻看一些著名的 Web 框架诸如 Struts1.X、Webwork2、Spring MVC 以及 JSF 的组件构成，会发现一套自定义的标签库是每个框架都致力于去实现的重要内容。所以，针对标签库使用的熟练程度，一度被认为是判定一个程序员对 Web 框架了解程度的重要标准之一。有许多 Web 开发人员，也长期存在着一定的误解：只要学好标签库，能够将标签库熟记于心，那么这个框架就算掌握了七八成。

实际上，标签库只是 MVC 框架中视图层的一个小小的组成部分，远远无法代替整个框架所引入的开发模式上的革命。作为一个程序员，千万不能把标签的作用神化，也不要认为学习标签库是学习 Web 框架的重中之重，因为它们对整个 Web 框架而言也无足轻重。所以在这里，我们也有必要对标签库产生的初衷、标签库设计的原理和标签库的发展趋势做一个详尽的分析。

11.3.1 标签库产生的初衷

在对视图本质的分析中，我们谈到了多种多样的视图表现形式。在这些多种多样的视图表现形式中，JSP 是使用最多也最为流行的一种视图表现元素。

在 JSP 诞生之初，最为吸引广大程序员的一个编程特性就是：允许在 HTML 代码中嵌入 Java 代码。这种特性在一个缺乏 Web 开发解决方案的年代是革命性的，因为它使我们可以比较容易地利用 Java 强类型语言的优势，直接在页面上完成许多复杂的业务逻辑和数据访问逻辑。更重要的是，**这种编程方式直接把 Java 开发与 Web 开发融为了一体**。

不过随着时间的推移，我们发现在 HTML 代码中嵌入过多的 Java 代码，非常不利于 JSP 的维护和扩展，这直接违反了我们所崇尚的 Web 开发的最佳实践。这种不可维护性主要体现在：一个 JSP 文件很容易在 HTML 代码、JavaScript 代码和 Java 代码的三重作用下迅速膨胀；同时，不同的语法结构使得不同的编程语言之间的融合变得极其困难，整个代码的可读性也非常差。

在这种情况下，有一种叫做标签库的新技术就被引入来解决这一问题。标签库解决代码可读性的基本思路在于：**消除在 HTML 代码中的 Java 代码，以一种自定义的扩展标签来代替原本需要 Java 代码表达的业务逻辑**。

使用标签库来提高 JSP 文件的可读性和可维护性，主要表现为三个方面：

- 避免在 JSP 页面中使用 Java 代码，而改用类似 HTML 的标签的形式来表达页面逻辑，能够达到让业务逻辑与显示分离的目的，提高 JSP 自身的关注度。
- 将数据展示彻底抽象成为一个独立的功能模块，能够规范编程习惯。
- 由于 HTML 自身的标签表达能力不足，通过使用 JSP 标签，可以对 HTML 语义进行扩展，从而完成许多 HTML 自身标签无法完成的工作。

这三个方面正好体现了 Web 编程过程中的不同角度，因而也成为我们对标签库进行分类的标准。结合具体的框架，我们也可以看到不同的框架在标签库的实现上，也无不遵循上述三个角度来进行设计。

那么，有了标签库以后，整个 Web 开发到底是变得简单了还是复杂了？标签到底是毒药还是解药呢？

11.3.2 标签库，毒药还是解药

从表面上看，标签库的引入为我们勾勒了一幅美妙的 Web 编程画卷：当我们使用上了标签库，似乎页面就不会变得那么凌乱不堪了；在页面上看不到 Java 代码，我们的编程思路也就不用再来回切换于 Java 语言和 HTML 语言之间。一时间，似乎每个 Web 开发框架都看到了标签库内在所蕴含的“巨大价值”，从而将标签库作为框架的一个重要组成部

分加以实现。然而，事情的真相远没有想象中那么简单。有关标签库究竟是毒药还是解药的争论，也始终存在着。

早在 2004 年的时候，iteye 的站长 robbin 就以《炮打 Taglib，我的一张大字报！》一文引发了众人对标签库的讨论。当时，robbin 的观点大概是这样的：

> 我认为 JSP 里面使用 Tag，就是一个错误！我反对在 JSP 里面使用 Tag，我推荐大家在 JSP 里面写 Java 代码，没错，就是在 JSP 里面写 Java 代码，我就是一直这么干！从 Sun 在 JSP 里面引入 Taglib，我就认为这是一个谎言！我认为大家都被 Sun 欺骗了，我做 JSP 编程，但凡我写过的 JSP，我从来不用 Tag，我觉得写 Java 代码让我很舒服，我不需要再去学习那别扭而无意义的 Tag 语法来增加我的工作量，来增加我的 JSP 页面调试难度。

不知道时隔多年，robbin 的观点是否有所变化。然而，在这些年中，JSP 技术本身并没有发生很大的变化，标签库也并没有消亡。不知道这是否能够成为标签库顽强生命力的一个佐证。那么，对于同样一个编程元素，不同程序员之间的分歧为什么如此之大呢？

对于拥护标签的程序员来说，标签库无疑是 Web 开发的一大利器。我们可以把这些人当作辩论的正方，看看他们的一些观点：

- 标签库产生的初衷没有错，它的存在能够简化 JSP 开发的难度，并对 HTML 的许多标签进行功能扩展
- 标签库从效果上的确在一定程度上解决了在 JSP 页面中避免频繁使用 Java 代码的情况
- 由于 Java 的语法和表现能力上的优势，使用标签库能够极大程度地封装成块的 HTML 代码，进而形成一套完整的页面组件

看到最后一条，大家是不是看出点儿端倪来了？原来，标签库最吸引程序员的一点：**在于组件化开发**。

结论　标签库技术能够经久不衰，成为一个主流技术，最重要的原因在于广大程序员对组件化开发的憧憬。

反对标签库的一方，则针锋相对地提出了众多的反驳意见。我们再来看看一些比较尖锐的反方观点：

- 标签库只是为了尝试避免在 JSP 页面中使用 Java 代码，实际上，这种情况很难避免。有时候，为了达到这个目的，反而带来了更多的代码和沉重的维护成本。因为编写一个标签，至少需要一个 tld 文件定义、一个 Java 的实现类以及在 JSP 中声明 tld 并引用标签定义。显然，对于许多情况而言，这样所带来的代码量，反而超过了在页面上直接使用 Java 代码

- 标签库的存在为广大程序员带来了无穷无尽的学习成本。因为根据使用的框架的不同，会带来不同的标签库。再加上许多公司在标签库为基础的页面组件方面的积累，公司内部就有一套标签库，于是学习这些标签的语法已经成为了程序员的沉重负担
- 对于以标签作为基础的页面组件，在面对需求经常发生变化的页面逻辑时，显得无能为力。因为它的维护成本极高，一旦逻辑变化频繁时，这些页面组件的表现能力还不如直接通过 JavaScript 操作 HTML DOM 方便

从反方的这些观点来看，他们始终咬住一点不放：**代码的可维护性和可扩展性**。而这两点，是我们总结的最重要的最佳实践之一。同时我们也看到，这也是标签库无法避免的软肋。

面对这一问题，如果抱有偏见，那么我们一定无法从中获得有益的结论。如果客观地对标签库的本质进行分析，或许我们就能从中得到一些别样的启示。当然，同样作为一个哲学问题，笔者在这里谈的更多也是一些个人的分析观点，这是笔者的哲学选择。大家在看了这些观点之后，最重要的还是应该形成自己的哲学观点，从而在实际开发中进行合理的选择。

11.3.3 标签库的发展趋势

正反双方的激烈辩论，从效果上反而推动了视图层的发展，因为正反双方都在寻求对方的破绽，从而找出解决方案加以反驳。我们不妨来看看双方对此所做的努力，并以此分析一下标签库未来的发展趋势。

面对反方的不断挑衅，正方不紧不慢地在以下这些方面做出了改进和努力：

- **制定标准标签库 JSTL，试图统一绝大多数的标签**
- **借助 IDE 的帮助，打造更多更丰富的 Web 层组件，并加强 AJAX 等功能**
- **将成熟的 JavaScript UI 框架改造成标签的形式**
- **随着 EL 的逐步推广，在 JSP 中使用 EL 也越来越能够被大家接受**

从这些努力来看，我们能够看到一条重要的逻辑主线：统一。因为一个知识体系一旦统一，它的学习成本就会降低，受到的支持程度也自然而然就会越高。

那么反方呢？反方作为标签库的反对者，则大刀阔斧地进行改革：

- **放弃 Taglib，推崇模板技术**
- **不使用 Taglib 来构造页面组件，而更多采用 JavaScript 技术丰富页面展示**
- **直接放弃使用 HTML 为模式的页面展现方式，改用 Flex 等 Flash 表现方案**

从实际效果上来看，反方的方案则更加激进，有的干脆直接抛弃了 HTML，采取其他的浏览器表现形式。这种激进的改革，从一定程度上推动了近年来富客户端（Rich Client）

的发展，并进入了一个如火如荼的阶段。

因此，从目前的发展趋势来看，无疑反方占据了绝对的优势。这与标签库本身作为一个技术规范有着很大的关系。正因为其实现原理和实现机制过于底层，导致了它自身的发展也受到技术本身的限制。因此，笔者看来：**标签库技术在模式上无法完成突破，是阻碍标签库技术发展的主要问题**。而 Web 开发多元化的趋势，将使得标签库技术仅仅成为一个重要的技术手段尴尬地存在于整个 Web 开发中。

11.3.4　标签的分类

对标签进行分类的标准非常简单，因为不同标签在 JSP 中所起的作用完全不同。因此，标签的分类主要是根据标签的作用做出的。根据在页面的不同作用，标签主要可以分成三类：

- **逻辑控制类**——控制页面输出逻辑
- **数据输出类**——与服务器端进行数据沟通并以一定格式进行输出
- **页面组件类**——将页面元素的功能进行组件化编程

11.3.4.1　逻辑控制类标签

对于逻辑控制类的标签，几乎每个框架都会涉及。以 JSTL 为例，JSTL 大概提供了以下一些用于逻辑控制的标签：

- c:if——分支判断
- c:forEach——循环
- c:choose/c:when/c:otherwise——分支判断
- c:catch——异常处理

逻辑控制类的标签之所以在每个框架中都有所涉及的原因在于**逻辑控制是组成程序的基本要素**。我们总是存在这一种需求，在页面上进行一定的输出逻辑控制。虽然对于不同的 Web 框架，对逻辑控制类的标签的定义稍有不同，但是功能本质还是一样的。

11.3.4.2　数据输出类标签

我们同样以 JSTL 为例，JSTL 大概提供了以下几个用于数据输出的标签：

- c:out——输出表达式的值
- c:url——输出格式化 url
- c:set——设置表达式的值
- c:param——设置参数
- fmt:message——输出资源文件中的值
- fmt:formatDate——格式化输出日期

❑ fmt:formatNumber——格式化输出数字

从这些标签上来看，JSTL 所提供的数据输出标签还可以进一步细分成两类：一类是带有数据格式处理的标签（诸如 fmt 标签）；还有一类是不带有数据格式处理的标签（诸如 c 标签）。

不过我们在这里应该关注的是，数据输出类的标签与服务器端程序的通信机制。也就是当使用 c:out 标签进行数据输出时，我们所使用的表达式如何从服务器端获取数据值。这个问题实际上也是我们在本章一开始所提到的视图层的两大核心问题之一，我们称之为数据访问。在下一节中，我们将专门来讲解。

11.3.4.3 页面组件类标签

页面组件类的标签就更多了。以 Struts2 为例，我们具体可以参考在 Struts2 自带的 Reference 中所介绍的组件类的标签，其中包含许多常用的页面控件实现。

这些 UI Tag 实际上是对 HTML 自身标签的扩展和扩充。所以，这类标签如果使用得当，将为开发带来极大的便捷。但是如果使用不当，则会带来很大的学习成本和维护成本。所以对于这类标签的学习，应该保持谨慎的态度。

11.4 数据访问的哲学

数据访问，是我们对视图（View）层元素与服务器端程序进行实时交互的概括。我们之前也讲到，数据访问是视图层的两大职责之一。那么，为什么看上去简单的数据访问还存在着哲学问题呢？

因为实现视图层与服务器端程序的交互需要跨过两道坎：

❑ **从哪里去取数据**

❑ **取过来的数据如何显示**

而从技术上讲，跨过这两道坎的过程也是我们进行哲学选择的过程。接下来，我们就来重点分析一下如何跨过这两道坎。

11.4.1 不要问我从哪里来

11.4.1.1 不是问题的问题

从哪里去取数据，这似乎本来并不应该成为一个问题。因为这个问题在我们讲 Servlet 的时候就已经圆满解决了。回顾一下当时我们得出的一个重要结论：

结论 HttpServletRequest 对象主要用于处理整个 Http 生命周期中的数据。

不仅如此，当时我们还展示了 HttpServletRequest 接口的源码。在这其中，getAttribute 和 setAttribute 这一对用于数据读写的方法，就是我们在视图层进行数据访问最为常用的方法。

于是乎，凡是传统的基于 Servlet 模式的 Web 开发框架都秉承了使用 HttpServletRequest 对象作为主要的数据沟通对象的规范。这样一来，我们在视图层要获得服务器端的数据，只需要通过两个步骤就可以完成：

- **在服务器端的 Controller 中，调用 setAttribute 方法将数据写入 HttpServletRequest 对象**
- **在视图层，调用 getAttribute 方法从 HttpServletRequest 对象中读取数据**

这个过程十分简单，也符合我们对 Web 开发标准的理解。为了使整个过程看上去更加自然，在 JSP 所支持的 EL 中，我们可以直接通过表达式的编写来完成上述步骤的第二步，例如：

```
${user.name}
```

这样 JSP 中读取 HttpServletRequest 中数据的过程就无须再借助任何标签或者是 Java 硬编码来完成。

当然，建立在 EL 标准之上的 JSTL 标签，则对上述的访问规则做了进一步的扩展。它支持一个 EL 表达式在多个 Web 请求生命周期中寻找对应的数据值。例如对于上述的例子：${user.name}，JSTL 会依次在 page、request、session、application 这四个不同的生命周期中调用相应的 getAttribute 方法进行对象查找，并返回第一个成功匹配表达式计算的对象值。

由此可见，使用基于 HttpServletRequest 对象来进行数据访问，是 Servlet 规范赋予我们的最底层支持。我们可以得出结论：

结论　*传统意义上的数据访问，就是针对 HttpServletRequest 对象的读取操作。*

11.4.1.2　隐形的力量——ValueStack

数据访问的来源作为一个问题在这里被提出，主要是**因为 Struts2 的数据访问模式与传统的数据访问模式有着天壤之别。**

这种访问模式上的天壤之别，使我们不得不去分析一下 Struts2 与传统的 Web 框架产生数据访问模式差异的原因。根据我们之前的分析：在视图层要获得服务器端的数据，需要通过两个步骤来完成。其中的第一个步骤，就是在 Controller 层，通过调用 setAttribute 方法将数据写入 HttpServletRequest 对象之中。然而我们发现，这一步在 Struts2 中根本无从下手。因为 Struts2 的 Action 是一个与 Web 容器解耦合的对象，我们无法在 Action 中

直接获取 HttpServletRequest 对象进行操作！

当然，有的读者会说，这不是什么难事。我们在之前的章节中，不是已经讨论过在 Struts2 的 Action 中获取 HttpServletRequest 的方法吗？这的确是一种可行的方案，不过这与 Struts2 的核心设计理念产生了冲突。因为 Struts2 所期望的编程模式是核心处理类与 Web 容器的解耦合。

在这种情况下，Struts2 / XWork 不得不打造一个另类的数据访问模式：**通过访问数据流元素来获得所有的数据。**

在之前的章节中，我们已经对 XWork 的数据流元素有了一个综合的分析。那么，我们如何来理解上面这句“通过访问数据流元素来获得所有的数据”呢？对此，我们可以使用之前曾经使用过的一个逻辑推理的招式，整个过程如图 11-3 所示。

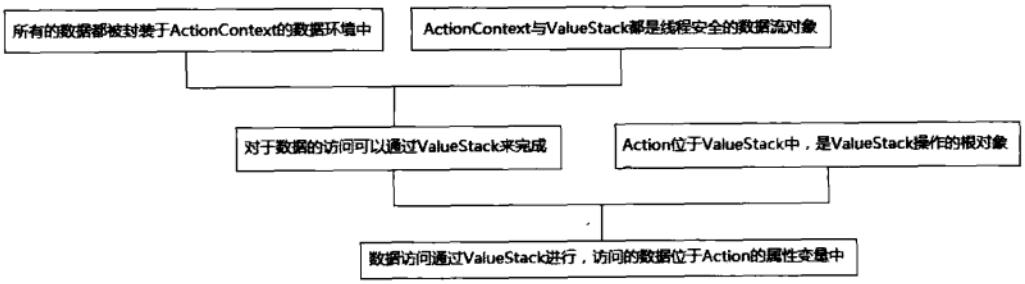


图 11-3 Struts2 数据访问推理过程

这就是我们在本节的标题中所谈到的隐形力量：ValueStack。如果把图 11-3 做一番综合性的评定，我们可以得出结论：

结论 作为数据流元素，ValueStack 是整个 Struts2 进行数据访问的实际操作句柄，而 Action 作为 ValueStack 操作的 Root 对象，其中的属性变量自然而然成为了数据访问源。

有了这一结论，整个 Struts2 的数据访问体系就豁然开朗了。因为 ValueStack 的特殊特性，我们可以将所有的数据访问都转化为对 ValueStack 的访问。有关这一点，我们可以以 JSTL 的访问逻辑进行源码级别的证明，如代码清单 11-6 所示。

代码清单 11-6 StrutsRequestWrapper.java

```
public Object getAttribute(String s) {

    if (s != null && s.startsWith("javax.servlet")) {
            return super.getAttribute(s);
    }
```

```
        ActionContext ctx = ActionContext.getContext();
        Object attribute = super.getAttribute(s);
         if (ctx != null) {
           if (attribute == null) {
               boolean alreadyIn = false;
               Boolean b = (Boolean) ctx.get("__requestWrapper.getAttribute");
               if (b != null) {
                   alreadyIn = b.booleanValue();
               }

               if (!alreadyIn && s.indexOf("#") == -1) {
                   try {
                       // 从 ValueStack 中访问数据
                       ctx.put("__requestWrapper.getAttribute", Boolean.TRUE);
                       ValueStack stack = ctx.getValueStack();
                       if (stack != null) {
                           attribute = stack.findValue(s);
                       }
                   } finally {
                       ctx.put("__requestWrapper.getAttribute", Boolean.FALSE);
                   }
               }
           }
        }
        return attribute;
    }
```

从源码中，我们可以看到 Struts2 在这里使用一个装饰模式覆盖了默认的 HttpServletRequest 的 getAttribute 的访问逻辑（这一点请具体参考第 4 章中有关装饰模式的讲解以及第 10 章中有关 Struts2 的 Http 预处理环境的讲解）。当我们在 JSP 中使用诸如 ${user.name} 的 EL 表达式进行数据访问时，这个类的代码就会被调用。我们看到，数据访问的过程被转嫁到了 ValueStack 中，这也就是 Struts2 的 Action 中的属性变量能够通过 JSTL 访问的根本原因。

如果我们把 ValueStack 的上下文环境也考虑进来，那么整个访问体系就变得异常宽泛。因为根据 OGNL 的知识，OGNL 表达式既可以对 Root 对象进行访问，也可以对上下文环境中的对象进行访问，其区别仅仅在于 OGNL 表达式的不同而已。

因此，在 Struts2 自定义的标签中，对于各种各样不同的访问表达式的问题就可以迎刃而解了。有些带有 # 符号的表达式，其表达的意思只是对 ValueStack 中上下文环境的数据的访问而已。

最后，我们来看看 ValueStack 这股隐形的力量与传统的 HttpServletRequest 的访问方式之间的核心分歧到底在哪里。

结论 对于请求－响应模式实现机制上的分歧，是造成数据访问模式差异的根本原因。

传统的 Web 框架，始终走不出 Web 容器的束缚，因而从数据访问模式上也不得不依赖 Web 对象来进行。Struts2 的 POJO 模式，天然地为数据访问构造了一个访问空间。这或许也是 Struts2 在最底层实现模式上的胜利。

11.4.2 不要问我长什么样

这一节的标题是“不要问我长什么样”，这是一个非常有趣的问题。因为既然是数据访问，在视图层怎么会产生“长什么样”的问题呢？这一点，我们可以使用一个简单的例子进行说明。

在 Java 中，表示日期有一个 java.util.Date 类。这是一个日期、时间的综合语义抽象。也就是说，在 Java 世界中要表示一个具体的时间，我们用这个类；要表示一个具体的日期，我们也用这个类。因为时间和日期在语义上表达了同样的意思。

然而，一旦时间这个语义展现在我们的面前，就成了一个异常复杂的情况。比如在中国，我们习惯使用年月日时分秒的方式来显示时间，但是在美国却是将时分秒放在前面，再辅之以月日年的形式进行显示。

在这种情况下，同样一个对象在 Java 世界中可以存活得很好，但在视图层却表现出截然不同的显示特征。这就使我们很容易想起曾经在第 6 章中谈到过的一个数据访问上的困境：

结论 数据访问的困境，主要还是因为数据模型在某些层次的展现缺乏足够的表现力。

在视图层，既没有数据类型的概念，也没有数据结构的概念。无论数据自身的结构有多复杂，一旦对外展现，都最终转化为统一的字符串形式。然而这种统一的字符串形式却有着丰富的显示特性。这个时候，我们就需要借助一些额外的手段来对数据的显示进行格式化。这就是所谓的格式化输出。

格式化输出数据的方案很多。在我们熟悉的 JSTL 标签中，fmt 标签就定义了众多的格式化标签。例如：fmt:formatDate、fmt:formatNumber 等等。而在 Freemarker 模板引擎中，则在数据访问表达式上做文章，在表达式计算的过程中完成对数据输出格式的转化。

不过无论如何，视图层在与服务器端进行数据通信的过程中，视图层对于到底取到的是什么样的数据是不知晓的。视图层应当负责对数据展示进行格式化。因此，我们对于视图层职责的认定，应该在“数据访问”这一条做一些增强：

结论 视图层不仅应负责与服务器端的数据沟通，还应该将获得的数据进行格式化输出。

11.5　小结

本章的内容不属于 Struts2 的运行主线，却对我们十分重要。因为视图层是整个 Web 开发的操作窗口，影响着 Web 交互的整个过程。

因而本章从视图表现技术入手，为大家揭示了视图层的本质和职责。并围绕视图层的两大职责，展开了对内容呈现和数据访问这两个问题的分析。本章中，读者并没有看到过多的源码解析，我们所谈论的是一些具有争议性的话题，每个话题都带有一些哲学选择的意味。读者可以细细品味框架的实现，也可以参考笔者的一些观点。

下列问题是本章的重点。大家可以温故而知新：

- Servlet 规范中的两大对象 HttpServletRequest 和 HttpServletResponse 各自有什么作用？
- 视图的本质是什么？
- 视图层有哪两大职责？
- Result 机制，从 XWork 和 Struts2 这两个不同的视角来看，能得出哪两种不同的结论？
- Result 对象在 Struts2 的运行过程中有哪些职责？
- 标签库产生的初衷是什么？
- 标签有哪些分类？
- 传统的数据访问模式与 Struts2 中的数据访问模式有什么区别？

第 12 章　三头六臂——Struts2 的扩展机制

人类创造的任何事物，都会具备人类赋予它的特性和期望值。软件程序作为人类智慧的结晶，自然也不例外。撇开软件程序自身的特性不谈，程序员对软件程序存在着 2 个最基本的期望：

- **可维护性**——程序应具备简单有效的方法对软件的功能模块进行修改和维护
- **可扩展性**——程序应具备简单有效的方法对软件的功能模块进行添加和扩展

在第 2 章中，我们曾经总结过 Web 开发中的一些最佳实践。其中有一条是这样说的：

最佳实践　*始终保证程序的可读性、可维护性和可扩展性。*

由此可见，程序的可读性、可维护性、可扩展性不仅仅是程序的基本属性，也是我们对程序进行不断重构的潜在原因，也直接促使我们使用框架进行程序的开发。而站在框架作者的角度来看，使程序保持可维护性和可扩展性是目的，使用框架进行程序开发是手段，两者不仅存在着足够的因果关系，也是相辅相成、互相促进的关系。

可读性和可维护性是我们对程序的基本要求，也是所有程序员都应该遵循的基本准则。而在此基础之上的可扩展性则对程序本身提出了更高的要求。在本章中，我们就来重点讨论一下 Struts2 中的扩展机制。

12.1　程序扩展机制的深入思考

如果我们从更加微观的角度来分析，一个很显而易见的问题会很快浮出水面：是否能够通过足够合理的设计，使得框架自身就具备可维护性和可扩展性，进而使得凡是使用框架进行编程的人，都能够直接获取这两大特性呢？答案是肯定的。为了实现软件的这一特性，一种有别于传统开发方式的软件开发模式被引入，这就是**插件（Plugin）模式**。

12.1.1　插件模式的基本概念

所谓插件，如果单单从字面上的意思去理解，即为：**可以即插即用的组件**。这里面包含了两层意思：

❑ **组件**——体现了插件的**功能性**

❑ **即插即用**——体现了插件的**灵活性**和**独立性**

要彻底理解上面所说的这两层意思，我们需要了解软件开发模式中的一些基本概念，并且深入探寻一下软件开发模式的发展过程以及其中所体现出来的优点和弊端。只有这样，我们才能真正理解插件模式对于软件开发的实际意义。

12.1.1.1　软件功能模块化

首先要引入的是**软件功能模块化**的概念。任何一款软件程序，我们都可以从软件的逻辑功能上进行模块划分。这些功能模块，有的用于软件的界面展示、有的用于业务逻辑处理、有的则负责与外部接口的交互等等。这些模块往往各司其职，在各自的领域承担着相应的职责。这些功能模块合起来，就构成了整个软件程序。这也就是我们所熟知的传统软件开发模式中模块化的概念。模块化实际上是一种“分而治之”的思想，在一定程度上解决了软件开发过程中的管理问题，也使得软件开发团队化成为一种可能。

随着开发的深入我们会发现，模块化开发存在着一些问题。因为模块的划分原则本来就是随着软件自身的需求而不断发生变化的，因而基于不断变化的需求所划分出来的模块，模块与模块之间的界定也十分模糊。所以即使模块的职责规划再清晰，模块与模块之间总也免不了形成**双向依赖**。这种依赖关系是不可避免的，然而实际上这种依赖关系却非常危险，因为当某个功能模块因为某些 Bug 造成了问题，那么与之存在依赖关系的相关模块就会被影响。而模块与模块之间这种无序的依赖关系，会造成整个程序的混乱。这也就是我们将“消除依赖”作为软件开发过程中一条重要的最佳实践的原因。当然，这也成为我们对整个软件开发模式进行重构的重要目标。这一目标，促成了我们对一种崭新的软件开发模式的思考：**是否能够通过有效的设计，将程序模块的双向依赖关系变成一种单向的集中依赖关系呢**？

12.1.1.2　核心程序与功能模块

在这种情况下，插件模式应运而生。插件模式的核心，是在模块化的概念之上，引入一个**核心程序**的概念。这个核心程序大概包含两个方面的内容：

❑ **抽取各个功能模块的公共逻辑成为一个独立的核心功能模块**

❑ **为核心功能模块添加对其他功能模块的执行调度功能**

这样一来，我们就可以顺利解决在上一章中所提到的模块与模块之间的依赖问题。因为有了核心程序，原本功能模块与功能模块之间那种杂乱无章的依赖关系就被转化为一种以核心程序为中心、功能模块围绕在核心程序周围形成的一种**星型依赖关系**。

我们可以使用工业生产中的齿轮运转过程来描述这一依赖关系，整个过程看起来就像图 12-1 所示。

图 12-1 插件模式的齿轮运转模型

从图中，我们可以发现核心程序被置于整个关系图的中心位置，成为一个最大的齿轮。而一个又一个功能模块则围绕着核心程序，以小齿轮的形式与核心程序套接在一起，以此表示功能模块对核心程序的依赖。从这个图中，我们可以找到插件模式的运行机理：

- **核心程序对功能模块的调度**——因为在大齿轮运转时，会带动小齿轮一起运转
- **各个功能模块仅仅依赖于核心程序，功能模块与功能模块之间并没有依赖**——小齿轮始终排在大齿轮周围，小齿轮之间没有交互
- **功能模块保持独立，不会相互影响**——某个小齿轮的掉落，不会影响其他的小齿轮运转

在这种情况下，插件开发模式也就初露端倪：**核心程序构成了一个基础平台，围绕在核心程序周围的程序功能模块成为一个一个插件，作为运行在基础平台之上的功能组件。**当然，插件开发模式对程序员提出了更高的编程要求。因为这样的开发模式，需要在整个软件设计上下更多的工夫，我们不仅需要对核心程序和功能模块进行逻辑抽象，还需要完成核心程序的调度功能。

插件模式的产生对于软件开发模式而言是一种进步，同时也是软件进入自我扩展和自我升级的一个重要里程碑。软件只有进入到这一阶段，我们才能说它真正走向成熟了。

12.1.2 常见的插件模式

现在，我们已经越来越多地使用基于插件模式的软件。接下来，我们将其中比较著名的几款软件进行举例说明，看看插件模式在当前软件行业中的影响力。

12.1.2.1　浏览器 —— Firefox

Firefox 浏览器是一款基于插件模式的浏览器软件。目前，基于 Firefox 浏览器的插件已经数以亿计，从各个方面对浏览器进行扩展，以改善用户的上网体验。也正因为其设计思想和用户体验方面的巨大优势，Firefox 浏览器不仅在专业人群中获得了越来越多的份额，在普通网民中也越来越受到青睐。在图 12-2 中，我们可以看到基于 Firefox 的插件已经超过 20 亿，而在使用中的插件也超过了 1 亿 5000 万个。

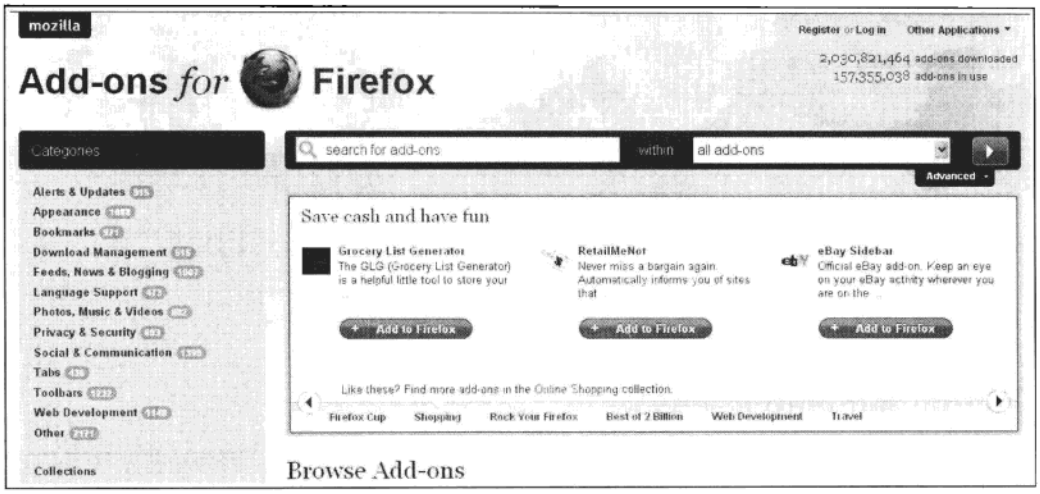


图 12-2　Firefox 插件统计图

从这幅 Firefox 插件的统计图中，我们可以看到一旦一个运行平台被搭建好，在这个运行平台之上的插件就将显示出极强的生命力。

12.1.2.2　编程 IDE——Eclipse

另外一款 Java 程序员非常熟悉的基于插件模式的软件是 Eclipse——一个服务于程序员的开发 IDE。Eclipse 很好地实现了插件的各种要素，并通过一个基础平台，将这些插件无缝整合在一起，无疑是插件模式软件中的精品。

我们可以通过许多途径获得 Eclipse 的插件。在 Eclipse 的官方网站上，就注册了众多官方插件：http://marketplace.eclipse.org/，如图 12-3 所示。

对于每一个使用过 Eclipse 进行开发的程序员来说，一定经历过由于 Eclipse 基础平台与插件的版本不匹配所带来的痛苦。这也成为一个非常值得我们思考的问题：既然插件模式是一个高级的开发模式，那么它又为什么会给程序员带来痛苦呢？

对这个问题的解答，需要从插件模式自身的特点入手，并做出深入的分析。在下一节中，我们就来重点讨论这一问题。

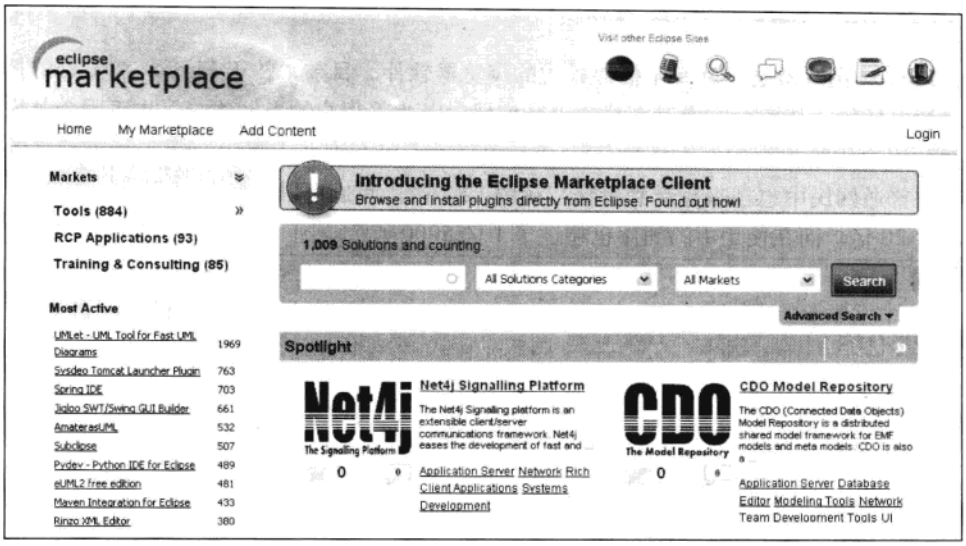


图 12-3 Eclipse 插件导航

12.1.3 插件模式的利弊分析

基于插件模式进行开发的软件，在其设计思想上无疑是先进的。因为站在软件产品的角度来说，任何一个软件产品，总是能够从本质上划分为以下两个方面：

- **软件运行环境**——可以运行基础功能模块的基础构件，俗称为程序骨架
- **功能模块**——运行在特定环境中的软件功能分类，包括用户界面、业务功能等

其中，软件运行环境作为最最基本的基础构件，需要提供的是程序的执行框架以及在这个框架运行范围内的一切要素的定义。而功能模块，往往是程序骨架中某些运行要素的具体实现，是程序运行真正的执行主体。可见，**功能模块对软件运行环境形成了高度的依赖**，这种依赖关系就是我们在插件模式的基本概念中所提到的**星型依赖关系**。

如果一个软件基于插件模式进行开发，那么它一定是基于上述两个方面进行的设计与实现。第一步，首先建立一个完整的软件运行环境，并着力打造一个通用的程序功能模块的执行模式。第二步，建立程序的功能模块，使之真正构成软件的血与肉。

这种软件开发模式从直观上看，似乎是无懈可击的。但是，正如在现实生活中，我们永远无法找到一个完美无缺的东西一样，插件模式同样存在着正反两方面需要考虑的因素。

12.1.3.1　正面因素

之前我们谈到的所有内容，几乎都可以称为插件模式的优点。因为从概念上讲，这首先是一种软件开发思路上的革命，就其本身而言就已经足够优秀了。如果进一步做一些小结，我们还能看到以下优点：

❑ 软件模块的管理变得更加容易

软件运行环境与各个功能模块之间相互独立，各个功能模块之间不形成必然的相互依赖。这就使得整个程序在逻辑上更为清晰，便于我们从各个不同的角度进行管理。

❑ 软件功能的可塑性和可扩展性更强

插件模式使得软件的功能表现变得更加灵活。不同的插件，甚至可以在一定范围内改变软件的功能行为，从而极大地增强了软件的可塑性和可扩展性。

❑ 软件的产品化和定制化成为可能

如果从产品的角度来分析，基于插件的开发，使得软件产品可以根据不同的需求进行定制化。利用插件的特性，根据不同的插件模块配置出表现不同的软件产品，是软件开发流程中的一个革命性改变。

站在这些优点之上进行的软件开发，就犹如“站在巨人的肩膀之上”。一个程序从架构角度来说始终应该以这个思路作为核心思想。

12.1.3.2　反面因素

在之前对插件模式的构成要素进行分析时，我们已经明确了一个重要的结论：

结论　*功能模块对软件运行环境形成了高度的依赖。*

对于这一个结论必须辩证地去理解。从消除功能模块与功能模块之间模糊的依赖关系的角度，这似乎体现出了插件模式的一个重要作用。然而，如果从依赖关系本身去理解，这一层依赖关系也必然成为插件模式的一个致命弱点。其具体表现为：**当软件运行环境发生变化时，会对依赖于这一运行环境的插件产生不可避免的影响**。而这样的影响，至少会表现在以下两个方面：

❑ 软件运行环境质量的优劣，将直接影响整个软件的质量

如果一个软件运行环境本身有致命的缺陷，那么运行于软件运行环境之上的插件将无能为力甚至寸步难行。

❑ 软件环境的版本升级将导致依赖于该环境的所有插件失效

也就是说，即使某个插件功能模块的描述完全相同，都不得不为了适应软件运行环境的改变，而产生多个对应的版本。这无形中增加了人为管理的成本。

对于上述的第一点，或许大家没有什么感性认识。我们可以以 Firefox 为例，当

Firefox 增加了越来越多的插件后，我们会发现，Firefox 的启动变得越来越慢。而这一点，也成为了 Firefox 底层的一个重要问题，而这一问题的解决，运行在其之上的插件却是无能为力的。

而对于上述的第二点，我们则以 Eclipse 为例，Eclipse 自身的平台至今已经做了多个版本的升级。于是，与之对应的插件也不可避免地进行了多个版本的升级支持。像 Eclipse 这样的开源项目，目前插件的数目已经可以用“万”来计数，因而为了支持不同版本的 Eclipse 而做的插件升级，将是一个非常浩大的工程。这也就是所谓的“牵一发而动全身”。

视线回到我们所讨论的 Struts2。仔细研究一下 Struts2 曾经发布的 3 个重要的里程碑版本：Struts2.0.X、Struts2.1.X 和 Struts2.2.X，我们可以发现，由于这三个版本的 Struts2 所依赖的底层的 XWork 做了重大的版本升级，同样导致了原本基于 Struts2.0.X 的插件，在 Struts2.1.X 上无法运行。广大的插件作者不得不为新的 Struts2 重新编写相应的功能插件，以适应这样的升级。

在这里，我们不得不重新搬出老生常谈的一个最佳实践了，**永远不要生搬硬套任何最佳实践，真理之锁永远只为最合适的那把钥匙开启**。所以，在判断插件模式是好是坏时，必须看看在当时的业务需求情况下这一开发模式是否合适。只有合适的才是最好的。

12.2 Struts2 的插件模式

12.2.1 深入 Struts2 插件

作为一个优秀的 Web 层框架，Struts2 本身就是依照插件模式来实现的。接下来，我们将从四个不同的方面对 Struts2 的插件模式做一些介绍。

12.2.1.1 表现形式

我们曾经在第 2 章分析框架的本质时提出一个重要的观点：

结论 *框架的本质是 JAR 包，其作用是针对 JDK 进行扩展。*

Struts2 作为一个框架，其本身也是一个 JAR 包（struts2-core-2.2.1.jar）。我们在之前的章节中已经对这个 JAR 包中的绝大多数内容做过分析。如果从插件模式的角度来看待它，这个 JAR 包实际上构建起了整个插件模式中的**核心程序**和**运行环境**。

那么插件模式中的功能组件，也就是插件，又是以什么样的形式出现呢？

结论　*Struts2 的插件以 JAR 包的形式存在。*

这个结论很明确地告诉我们，Struts2 插件的存在形式是 JAR 包。这与 Struts2 核心程序的表现形式相同。而 Struts2 对这个 JAR 包的构成要素提出了三个方面的要求：

- **插件所在的 JAR 包，必须位于 Web 应用程序的 CLASSPATH 下**
- **在 JAR 包内的根目录位置，应该存放一个名为 struts-plugin.xml 的文件**
- **struts-plugin.xml 的文件与 Struts2 的基础配置文件 struts.xml 的格式相同**

Struts2 对于插件构成要素的三方面要求，我们可以从逻辑上进行分析：其中的第一个方面和第二个方面，规定了核心程序与插件之间的通信机制；而第三个方面，规定了核心程序所支持的插件扩展内容。有关这两个角度，我们在后面的章节中还会详细展开。

12.2.1.2　安装使用

如果要获得某个插件的功能支持，我们只需要将插件所在的 JAR 文件加入到 Web 应用的 CLASSPATH 下即可。与此同时，如果插件本身还对其他的 JAR 文件形成依赖（这些依赖与 Struts2 本身无关），作为插件运行的必要依赖，也应该被加入到 CLASSPATH 中。

这就是 Struts2 中插件的安装使用方法。从中我们可以发现，Struts2 中插件的安装使用非常简单，完全符合插件的即插即用的特点。核心程序与插件之间所维系的依赖关系，仅仅取决于两个条件：

- **插件是否位于 CLASSPATH 下**
- **插件所在 JAR 文件的根目录下是否存在一个名为 struts-plugin.xml 的配置文件**

这两个条件实际上就是我们上一节中曾经提到的 Struts2 的核心程序与 Struts2 插件之间的通信机制。当一个 Struts2 的应用启动时，Struts2 的初始化主线会扫描位于 CLASSPATH 根目录下的所有 JAR 文件，如果某一个 JAR 文件满足在其根目录下有一个 struts-plugin.xml 的配置文件，Struts2 才会读取这个文件并将它作为 Struts2 基础配置的一部分加载到 Struts2 的运行环境中。

12.2.1.3　依赖关系

Struts2 中插件的依赖关系与插件模式本身所规定的星型依赖关系一致，其中包含了两层意思：

- **Struts2 的插件对 Struts2 的核心 JAR 构成依赖**

Struts2 的插件作为一个个独立的 JAR 文件，各自具备对 Struts2 的核心 JAR 的功能扩展。这种依赖关系就像是我们之前提到的插件概念中的功能模块对软件运行环境的依赖。

- **Struts2 的插件与插件之间原则上不形成相互的依赖关系**

这一点也符合插件模式本身的要求。有关这一点，我们可以在之后 Struts2 插件的不同分类上看得出来。每一种插件分别代表了不同的逻辑扩展类型，插件和插件之间完全没有逻辑功能上的交集。

从这其中的第二点，我们可以得出一个重要的推论，这个推论对我们理解 Struts2 插件的加载机制很有帮助：

结论 Struts2 的各个插件在被 Struts2 的核心程序加载时，其顺序是随机的。

这是一个自然而然得出来的推论。不过它却从另一个角度对 Struts2 的插件编写提出了要求，也是我们在研究 Struts2 插件的过程中最值得注意的一个方面。

12.2.1.4 扩展点

之前我们谈到了 Struts2 中核心程序与插件之间的通信机制。在这其中有一个起着纽带作用的配置文件：struts-plugin.xml，这个配置文件实际上在整个插件的运行机制中起着双重作用：

- **struts-plugin.xml 文件的名称和所处位置成为了核心程序与插件的通信接口**
- **struts-plugin.xml 文件的内容结构成为了插件对核心程序进行扩展的依据**

这其中第一点的通信作用，我们已经在插件的安装使用中有过详细的说明。而第二点，则是本节中我们需要详细展开的内容：Struts2 的扩展点。

有关 Struts2 的扩展点，我们在之前的源码分析中已经有过诸多涉及。而在 Struts2 的官方 Reference 中，对这些扩展点做了一些简单的总结，如图 12-4 所示。

Type	Property	Scope	Description
com.opensymphony.xwork2.ObjectFactory	struts.objectFactory	singleton	Creates actions, results, and interceptors
com.opensymphony.xwork2.ActionProxyFactory	struts.actionProxyFactory	singleton	Creates the ActionProxy
com.opensymphony.xwork2.util.ObjectTypeDeterminer	struts.objectTypeDeterminer	singleton	Determines what the key and element class of a Map or Collection should be
org.apache.struts2.dispatcher.mapper.ActionMapper	struts.mapper.class	singleton	Determines the ActionMapping from a request and a URI from an ActionMapping
org.apache.struts2.dispatcher.multipart.MultiPartRequest	struts.multipart.parser	per request	Parses a multipart request (file upload)
org.apache.struts2.views.freemarker.FreemarkerManager	struts.freemarker.manager.classname	singleton	Loads and processes Freemarker templates
org.apache.struts2.views.velocity.VelocityManager	struts.velocity.manager.classname	singleton	Loads and processes Velocity templates
com.opensymphony.xwork2.validator.ActionValidatorManager	struts.actionValidatorManager	singleton	Main interface for validation managers (regular and annotation based). Handles both the loading of configuration and the actual validation (since 2.1)
com.opensymphony.xwork2.util.ValueStackFactory	struts.valueStackFactory	singleton	Creates value stacks (since 2.1)
com.opensymphony.xwork2.reflection.ReflectionProvider	struts.reflectionProvider	singleton	Provides reflection services, key place to plug in a custom expression language (since 2.1)
com.opensymphony.xwork2.reflection.ReflectionContextFactory	struts.reflectionContextFactory	singleton	Creates reflection context maps used for reflection and expression language operations (since 2.1)
com.opensymphony.xwork2.config.PackageProvider	N/A	singleton	All beans registered as PackageProvider implementations will be automatically included in configuration building (since 2.1)
com.opensymphony.xwork2.util.PatternMatcher	struts.patternMatcher	singleton	Matches patterns, such as action names, generally used in configuration (since 2.1)
org.apache.struts2.views.dispatcher.DefaultStaticContentLoader	struts.staticContentLoader	singleton	Loads static resources (since 2.1)

图 12-4 Struts2 的扩展点

图中列举了 Struts2 或者 XWork 本身所提供的可供扩展的接口。这些接口在 Struts2 中已有默认的实现类，这些实现类定义在 Struts2 的核心 JAR 文件的 struts-default.xml 中。我们以 ObjectFactory 为例：

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

从上述代码我们可以看到，这些接口本身就带有多种实现方式。根据 XWork 容器的相关知识可知，type 和 name 共同构成了配置文件中 <bean> 节点的寻址驱动元素。Struts2 在默认情况下将读取 name 的值为 struts 或者 default 的实现类作为其默认实现类。这也是 Struts2 初始化过程中最重要的步骤之一，读者可以回顾一下我们在第 9 章中曾经给出的 BeanSelectionProvider 的实现类的源码分析。

在这里，Struts2 所给出的扩展点实际上是指我们可以在插件中重新定义这些接口的实现类，从而替换 Struts2 的默认行为。这是 Struts2 / XWork 最重要的一类扩展点。

除此之外，还可以在 struts-plugin.xml 文件中加入我们自定义的实现类、运行参数、事件映射关系等等。由于 struts-plugin.xml 与 struts.xml 的格式相同，我们甚至可以完全将它作为一个独立的 Struts2 配置来理解。这样一来，我们可以根据配置节点的不同作用，将 Struts2 中的扩展点分为三类：

- **定义新的事件映射关系对象**（Action、Interceptor、Result 等等）
- **引入新的接口定义实现或者运行参数，将其纳入 Struts2 的容器进行管理**
- **覆盖 Struts2 / XWork 的默认接口实现机制**

上述这三个方面，读者可以结合 Struts2 的配置元素定义和初始化主线来共同理解，因为所有的扩展点，最终都是通过配置元素来完成的。

12.2.2　Struts2 插件分类

Struts2 的许多重要功能都是通过插件来实现的，而这些常见的插件已经由很多前辈贡献出来，并且作为 Struts2 的分发包的一部分共同发布。所以，我们可以在 Struts2 的分发包中找到这些插件以及它们的源码。

当然，我们也可以自行编写 Struts2 插件，并将它们发布到 Struts2 的官方网站，但是 Struts2 并不对未经认证的插件提供支持，读者需要自行承担使用它们的风险。这一点在 Struts2 的官方网站上也有所说明：

> *Contributed plugins may be of varying quality. If not bundled with the official Struts 2 distribution, a plugin cannot be guaranteed to be safe. You install plugins from this space at your own risk. We do not monitor or guarantee any code posted in this space.*

与 Struts2 的分发包共同发布的插件位于 Struts2 分发包的 lib 目录下，如图 12-5 所示。当然，随着这些插件一起发布的，还有这些插件的源码。我们可以在 Struts2 分发包

的 src 目录下找到这些插件的源码。由于它们都是经过 Struts2 官方认证的插件，我们可以在应用开发过程中放心地使用它们。

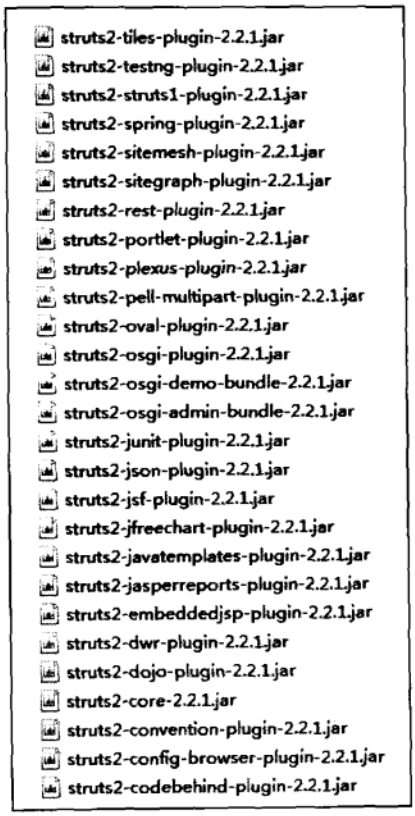


图 12-5 Struts2 自带的插件列表

Struts2 的官方网站里有一个插件中心，里面列举了发布在 Struts2 官方网站上的所有插件。我们可以在 Struts2 的首页找到这个插件中心：http://cwiki.apache.org/S2PLUGINS/home.html。如图 12-6 所示，点击其中的“Plugin Registry”按钮，我们就能进入 Struts2 的插件中心。

面对这些纷繁复杂的插件，大家可能会不知所措。所以笔者在这里对它们进行简单的分类，并且列出一些典型的插件，供给大家学习。

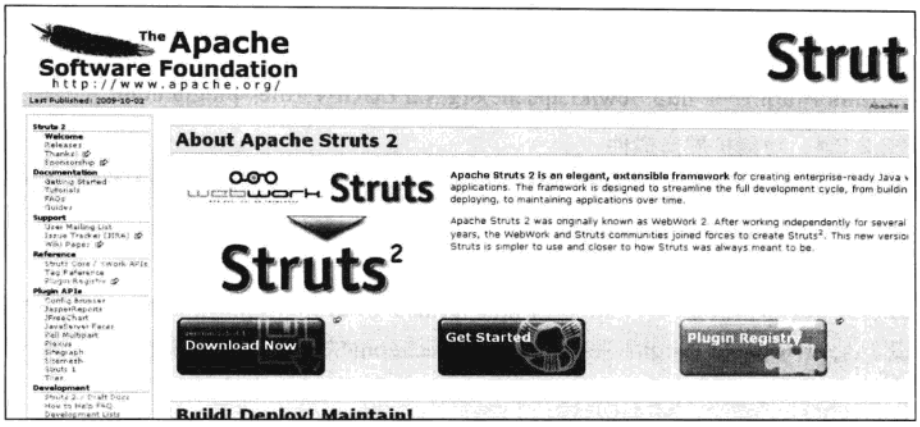


图 12-6　Struts2 的官网首页

12.2.2.1　框架整合类插件

这类插件提供了 Struts2 与其他许多开源框架的整合方式。所以，对于那些不知道 Struts2 与 Spring 如何整合、Struts2 与 DWR 如何整合的程序员，不妨先看看这些插件的说明，并试着运行一下这些插件的例子，或许会收获颇丰。

- Spring Plugin——http://cwiki.apache.org/S2PLUGINS/spring-plugin.html
- Guice Plugin——http://cwiki.apache.org/S2PLUGINS/guice-plugin.html
- JRuby Plugin:——http://cwiki.apache.org/S2PLUGINS/jruby-plugin.html

12.2.2.2　简化配置类插件

这类插件的主旨是为了简化 Struts2 原有的配置结构。Struts2 默认使用 XWork 提供的 XML 方式进行框架的基础配置。这些插件所提供的思路是使用“约定大于配置”或者 Annotation 等方式来代替 XML 配置，从而简化整个项目在配置文件上的工作量。

在这其中，某些插件还通过对 Struts2 底层的某些实现类的改写使 Struts2 支持 Restful 特性等等。不过绝大多数这类插件都会涉及 Struts2 底层的实现，是所有插件中学习成本最高的一类。在使用它们的时候需要慎重选择，因为它们有时候会直接改变 Struts2 的默认行为。

- Codebehind Plugin——http://cwiki.apache.org/S2PLUGINS/codebehind-plugin.html
- SmartURLs plugin——http://cwiki.apache.org/S2PLUGINS/smarturls-plugin.html
- Convention Plugin——http://cwiki.apache.org/S2PLUGINS/convention-plugin.html

12.2.2.3　页面装饰类插件

这类插件主要是为了整合类似 Tiles 或者 Sitemesh 这样的页面装饰框架，提供页面结

构化的功能整合支持。

- Sitemesh Plugin——http://cwiki.apache.org/S2PLUGINS/sitemesh-plugin.html
- Tiles Plugin——http://cwiki.apache.org/S2PLUGINS/tiles-plugin.html

12.2.2.4 功能扩展类插件

这类插件最多，内容也最为丰富，包含了各种各样对 Struts2 的功能扩展和功能改进。这些功能扩展类插件往往以替换 Struts2 默认的实现方式来实现。读者可以各取所需，根据不同的业务需求选择相应的插件使用。

- JFreeChart Plugin——http://cwiki.apache.org/S2PLUGINS/jfreechart-plugin.html
- JasperReports Plugin——http://cwiki.apache.org/S2PLUGINS/jasperreports-plugin.html
- Potlet Plugin——http://cwiki.apache.org/S2PLUGINS/portlet-plugin.html

12.2.3 Struts2 的插件加载机制

12.2.3.1 插件加载概述

到目前为止，我们已经对 Struts2 的插件机制有了充分的了解。作为一个 Web 层的开发框架，Struts2 实际上是由一个核心程序和一堆插件共同构成的。在这里，我们需要探讨的另外一个核心问题是，Struts2 又是通过何种机制来对所有这些插件进行加载和管理的呢？

要了解 Struts2 中插件模式的加载机制，我们可以从 Struts2 官方的 Reference 中找到一些基本的线索：

> *The framework loads its default configuration first, then any plugin configuration files found in others JARs on the classpath, and finally the "bootstrap" struts.xml.*
>
> *1) struts-default.xml (bundled in the Core JAR)*
>
> *2) struts-plugin.xml (as many as can be found in other JARs)*
>
> *3) struts.xml (provided by your application)*
>
> *Since the struts.xml file is always loaded last, it can make use of any resources provided by the plugins bundled with the distribution, or any other plugins available to an application.*

摘自 Struts2 的 Reference 的这段话，基本上囊括了 Struts2 加载和管理插件的方式。在这其中最重要的部分，当属这些配置文件在系统启动时的加载顺序：

- struts-default.xml——位于 Struts2 核心 JAR 包的根目录下——最先被加载
- struts-plugin.xml——所有插件所在 JAR 包的根目录下——紧接着被加载
- struts.xml——位于应用程序的 CLASSPATH 下——最后被加载

结合加载顺序和 Reference 文档的说明，我们可以总结出一些重要的推论：

结论　Struts2 核心包中 struts-default.xml 最先被加载，用于指定 Struts2 的一些默认行为。

struts-default.xml 这个文件我们曾经在第 3 章讲 Struts2 配置元素的时候就与之有过亲密接触。它是指定所有 Struts2 的默认行为的系统级别的配置。这些配置在 Struts2 的初始化主线运行之初被首先加载，使得 Struts2 在初始化之初就获得了基本的配置元素支持。

结论　Struts2 会搜索所有 CLASSPATH 中的 JAR 文件，查找其中的 struts-plugin.xml 文件。这些文件的加载顺序是不确定的。

这一结论从另外一个侧面证明：Struts2 的插件与插件之间并不存在依赖关系。Struts2 对于插件的管理方式是松散的，目的是为了保证插件功能的独立性。

结论　名为 struts.xml 的文件是应用程序中 Struts2 的基础配置文件，它被最后加载，指定应用程序级别的 Struts2 行为。

Struts2 对于配置文件的加载顺序，也从一个侧面体现了 Struts2 在配置文件设计上的层次性。这个层次性表现在，任何配置定义总是先整体、后局部；先系统、后应用。同时，后加载的配置定义总还有机会覆盖先加载的配置定义。这样就为一个 Web 应用提供了灵活的配置选项。

Struts2 对所有插件的加载顺序在 Struts2 的初始化主线中由一个特定的配置加载接口（ConfigurationProvider）来完成，这个配置加载接口就是 StrutsXmlConfigurationProvider。在第 9 章的初始化主线中，我们知道所有的配置加载接口都经历了先定义、后使用的过程。接下来，我们就来详细看看这两个过程。

12.2.3.2　配置加载接口的定义

首先是配置加载接口的定义过程。这个过程由 init_TraditionalXmlConfigurations 方法完成。其相关源码如代码清单 12-1 所示。

代码清单 12-1　Dispatcher.java

```
// 指定基于传统 XML 文件配置方式的配置处理类
private void init_TraditionalXmlConfigurations() {
    // 根据 web.xml 中配置的 config 值，指定依次加载的 XML 文件的名称
    String configPaths = initParams.get("config");
    if (configPaths == null) {
        // 默认的 XML 文件名称及其顺序为：struts-default.xml, struts-plugin.xml 和
        // struts.xml。这是 Struts2 在默认情况下对配置文件的加载顺序
        configPaths = DEFAULT_CONFIGURATION_PATHS;
```

```
        }
        String[] files = configPaths.split("\\s*[,]\\s*");
        for (String file : files) {
            if (file.endsWith(".xml")) {
                // Struts2 还能够支持传统的 xwork.xml，并使用 XmlConfigurationProvider
                 // 进行配置加载处理
                if ("xwork.xml".equals(file)) {
                    configurationManager.addConfigurationProvider(new
XmlConfigurationProvider(file, false));
                } else {
                    // 其余文件，将使用 StrutsXmlConfigurationProvider 进行配置加载处理
                    configurationManager.addConfigurationProvider(new
StrutsXmlConfigurationProvider(file, false, servletContext));
                }
            } else {
                throw new IllegalArgumentException("Invalid configuration file
name");
            }
        }
    }
```

这个方法中，我们看到了 Struts2 中 StrutsXmlConfigurationProvider 的定义过程。在这整个过程中，最重要的一个变量就是 configPaths，因为它指定了哪些名称的 XML 文件应被加载。

在默认情况下，Struts2 会使用 DEFAULT_CONFIGURATION_PATHS（struts-default.xml / struts-plugin.xml / struts.xml）作为默认的配置文件在 CLASSPATH 中查找。当然，我们也可以在 web.xml 中指定需要加载的 Struts2 配置文件的路径，以逗号隔开并通过初始化参数传入到初始化过程之中。

在一般情况下，我们不推荐人工指定 configPaths，因为在 Struts2 中，默认的的配置文件加载路径 DEFAULT_CONFIGURATION_PATHS 不仅包含了 Struts2 最基础的配置定义（位于 struts-default.xml 中），还包含了插件的加载（位于 struts-plugin.xml 中），同时也指定了应用程序级别的配置定义（位于 struts.xml 中）。应该说，这样的加载顺序已经十分完整有效。对 Struts2 并不十分熟悉的程序员，如果自行指定 configPaths，恐怕或多或少就会在上面 3 个方面的配置中少加载了一些配置文件而导致 Struts2 功能上的缺失。

12.2.3.3 配置加载接口的使用

StrutsXmlConfigurationProvider 的使用过程，实际上在 Struts2 的初始化主线的运行过程中，与其他配置加载接口共同被调度执行。我们知道，由于 Struts2 使用了策略模式来处理不同的配置加载接口，因而我们在这里只需要具体看看 StrutsXmlConfigurationProvider 的实现机制就可以知道配置加载接口的使用过程。其相关源码如代码清单 12-2 所示。

代码清单 12-2　StrutsXmlConfigurationProvider.java

```
// StrutsXmlConfigurationProvider 的初始化方法
public void init(Configuration configuration) {
    this.configuration = configuration;
    this.includedFileNames = configuration.getLoadedFileNames();
// 这里将调用其内部的 loadDocuments 方法，解析 XML 配置文件
    loadDocuments(configFileName);
}

// 处理 XML 配置文件
private void loadDocuments(String configFileName) {
        try {
            loadedFileUrls.clear();
            documents = loadConfigurationFiles(configFileName, null);
        } catch (ConfigurationException e) {
            throw e;
        } catch (Exception e) {
            throw new ConfigurationException("Error loading configuration
file " + configFileName, e);
        }
}

// 真正处理 XML 配置文件解析的场所，这里分成 2 个主要步骤：
// 1. XML 配置文件的寻址
// 2. XML 配置文件的解析
private List<Document> loadConfigurationFiles(String fileName, Element
includeElement) {
    List<Document> docs = new ArrayList<Document>();
    List<Document> finalDocs = new ArrayList<Document>();
    if (!includedFileNames.contains(fileName)) {
        if (LOG.isDebugEnabled()) {
            LOG.debug("Loading action configurations from: " + fileName);
        }

        includedFileNames.add(fileName);

        Iterator<URL> urls = null;
        InputStream is = null;

        IOException ioException = null;
        try {
            // 根据文件名称进行 XML 文件寻址
            urls = getConfigurationUrls(fileName);
        } catch (IOException ex) {
            ioException = ex;
        }

        // 处理 XML 文件没有找到的异常
```

```
        if (urls == null || !urls.hasNext()) {
            if (errorIfMissing) {
                throw new ConfigurationException("Could not open files of the
name " + fileName, ioException);
            } else {
                LOG.info("Unable to locate configuration files of the name
" + fileName + ", skipping");
                return docs;
            }
        }

        // 寻址完成，开始解析 XML 文件。注意，对于相同的文件名，可能找到多个 XML 文件
        URL url = null;
        while (urls.hasNext()) {
            try {
                url = urls.next();
                is = FileManager.loadFile(url);

                InputSource in = new InputSource(is);

                in.setSystemId(url.toString());

                // 依次加入每个 XML 所对应的 document 节点
                docs.add(DomHelper.parse(in, dtdMappings));
            } catch (XWorkException e) {
                if (includeElement != null) {
                    throw new ConfigurationException("Unable to load " + url,
 e, includeElement);
                } else {
                    throw new ConfigurationException("Unable to load " + url,
e);
                }
            } catch (Exception e) {
                final String s = "Caught exception while loading file " +
 fileName;
                throw new ConfigurationException(s, e, includeElement);
            } finally {
                if (is != null) {
                    try {
                        is.close();
                    } catch (IOException e) {
                        LOG.error("Unable to close input stream", e);
                    }
                }
            }
        }

        Collections.sort(docs, new Comparator<Document>() {
            public int compare(Document doc1, Document doc2) {
```

```
                return
XmlHelper.getLoadOrder(doc1).compareTo(XmlHelper.getLoadOrder(doc2));
            }
        });

        // 处理每个 XML 中的节点
        for (Document doc : docs) {
            Element rootElement = doc.getDocumentElement();
            NodeList children = rootElement.getChildNodes();
            int childSize = children.getLength();

            for (int I = 0; I < childSize; i++) {
                Node childNode = children.item(i);

                if (childNode instanceof Element) {
                    Element child = (Element) childNode;

                    final String nodeName = child.getNodeName();

                    if ("include".equals(nodeName)) {
                        String includeFileName = child.getAttribute("file");
                        if (includeFileName.indexOf('*') != -1) {
                            ClassPathFinder wildcardFinder = new
ClassPathFinder();
                            wildcardFinder.setPattern(includeFileName);
                            Vector<String> wildcardMatches =
wildcardFinder.findMatches();
                            for (String match : wildcardMatches) {

finalDocs.addAll(loadConfigurationFiles(match, child));
                            }
                        } else {

finalDocs.addAll(loadConfigurationFiles(includeFileName, child));
                        }
                    }
                }
            }

            finalDocs.add(doc);
            loadedFileUrls.add(url.toString());
        }

        if (LOG.isDebugEnabled()) {
            LOG.debug("Loaded action configuration from:" + ilename);
        }
    }
    return finalDocs;
}
```

在这里，有大量有关 XML 文件的解析部分，我们将不做详细分析。读者可以根据 Struts2 的配置文件的格式对照着查找自己感兴趣的环节自行查看源码。而对于在 XML 文件解析之前的 XML 配置文件寻址，我们有必要做一下深入的探究，其源码如代码清单 12-3 所示。

代码清单 12-3 ClassLoaderUtil.java

```
public static Iterator<URL> getResources(String resourceName, Class
callingClass, boolean aggregate) throws IOException {

    AggregateIterator<URL> iterator = new AggregateIterator<URL>();

    // 首先使用当前线程的 ContextClassLoader 在 CLASSPATH 中查找资源
    iterator.addEnumeration(Thread.currentThread()
.getContextClassLoader().getResources(resourceName));

    // 如果没有找到，继续在 Class 的 ClassLoader 中查找资源
    if (!iterator.hasNext() || aggregate) {
            iterator.addEnumeration(ClassLoaderUtil
.class.getClassLoader().getResources(resourceName));
    }

    // 如果没有找到，则在当前调用者的 ClassLoader 中继续查找
    if (!iterator.hasNext() || aggregate) {
        ClassLoader cl = callingClass.getClassLoader();

        if (cl != null) {
            iterator.addEnumeration(cl.getResources(resourceName));
        }
    }

    // 如果没有找到，会在当前文件名之前增加一个 '/' 符号，递归调用查找
    if (!iterator.hasNext() && (resourceName != null) &&
((resourceName.length() == 0) || (resourceName.charAt(0) != '/'))) {
        return getResources('/' + resourceName, callingClass, aggregate);
    }

    return iterator;
}
```

从上面的这个工具类中，我们可以看到 Struts2 在查找资源的时候非常全面。上面这个方法也可以用于查找一切 CLASSPATH 下被加载的文件资源，无论该资源是否位于 JAR 文件内，都能够被查找到。

至此为止，Struts2 中对所有插件的加载机制已经通过源码展示给读者。在这个过程中，我们可以看到 Struts2 在插件模式实现上的一些过人之处：

- Struts2 的插件被设计成独立于应用程序的 JAR 包，使得安装和使用变得极其简单
- Struts2 对于核心程序、插件以及应用程序之间的关系处理得很有条理，使得这三者之间虽然具备了丰富的层次关系，也不至于造成混乱
- Struts2 对于插件的构成要求很低，扩展点却很多，使得我们在模块化设计的过程中，有更多的余地

12.3　小结

插件模式的形成，是软件开发模式中的重要里程碑。有了插件，就相当于使得原本功能单一的软件，一下子拥有了三头六臂的功效。

在本章中，我们不仅探讨了插件模式的来龙去脉，并认真分析了 Struts2 实现插件模式的机理。最后，我们给出了 Struts2 中插件的大致分类供读者参考。对 Struts2 中的各个插件有兴趣的读者，可以直接到 Struts2 的官方网站寻找适合自己项目的插件。

本章的内容不多，却是在基础功能之上实现内容飞跃的一章。回顾本章的内容，读者是否已经了解了以下知识点呢？

- 什么是插件开发模式？
- 基于插件模式进行开发的程序，在程序的模块划分上有什么样的特殊要求？
- 插件模式有什么利弊？
- Struts2 中的插件以何种方式表现？如何进行安装使用？
- Struts2 的插件模式表现出哪两层依赖关系？
- Struts2 的插件对于 Struts2 有哪三类扩展方式？
- Struts2 的插件可以分为哪些不同的类型？
- Struts2 的插件在加载顺序上有什么特点？ Struts2 又是如何实现不同层次的配置文件的加载的？