Webx框架指南

Michael Zhou

Webx框架指南

Michael Zhou

出版日期 2010-11-13

引言

1. 阅读向导

注意

- 如果你希望马上尝试用webx来搭建应用，请转至第 IV 部分 “Webx应用实作”。
- 如果你想了解webx的整体设计思想，请转至第 I 部分 “Webx框架概览”。
- 如果你想进一步了解webx的每个具体服务，请转至第 II 部分 “Webx基础设施服务”以及第 III 部分 “Webx应用支持服务”。
- 如果你想了解一些常用的开发工具，请转至第 V 部分 “辅助工具”。

2. Webx是什么?

Webx是一套基于Java Servlet API的通用Web框架。它在Alibaba集团内部被广泛使用。从2010年底，向社会开放源码。

3. Webx的历史

- 2001年，阿里巴巴内部开始使用Java Servlet作为WEB服务器端的技术，以取代原先的Apache HTTPD server和mod_perl的组合。
- 2002年，选择Jakarta Turbine作为WEB框架，并开始在此之上进行扩展。
- 2003年，经过大约一年的扩展，框架开始成熟。我们私下称这个经过改进的Turbine框架为Webx 1.0。
- 2004年，借着淘宝网的第一次改版，我们正式推出了Webx 2.0。由于Turbine开源项目发展过于缓慢，我们不得不放弃它。Webx 2.0是从零开始完全重写的，仅管它仍然延续了Turbine的使用风格。
- 2004年11月，Webx 2.0和Spring框架整合。
- 从那以后，Webx 2.0一直在进化，但没有作根本性的改动。
- 2010年，Webx 3.0发布。Webx 3.0抛弃了Webx 2.0中过时的、从Turbine中发展而来的Service框架，直接采用Spring作为其基础，并对Spring作了重大改进。Webx 3.0完全兼容Webx 2.0的代码，只需要修改配置文件就可完成升级。
- 2010年底，Webx 3.0开源。

4. 为什么要用Webx而不是其它的开源框架?

现在有很多Java的Web框架可供选择，并且它们也都是免费的。例如，

- Struts - http://struts.apache.org/
- Webwork - http://www.opensymphony.com/webwork/
- Tapestry - http://tapestry.apache.org/
- Spring MVC - http://www.springsource.org/
- ……

以上框架都是非常优秀的。说实话，如果阿里巴巴网站在2001年开始，就有这么多可选择的话，无论选择哪一个都不会有问题。因为这些年来，所有的开源Web框架都在互相学习、并趋于相似。Webx也不例外，它吸收了其它框架的很多想法。因此，当你使用Webx的时候，你会觉得在很多方面，它和其它开源的框架非常类似。

我并不是说所有的框架都一样好，而是说只要假以时日，所有的框架在发展过程中，必然会积聚好的方面，淘汰坏的方面，从而变得足够好。从这个角度看，的确没有特别明显的理由来选择Webx，但也没有明显的理由不选择Webx。

另一方面，由于每一种框架采用不同的设计，必然会有各自的优势。Webx也是如此 —— 它在某些方面有一些独到的设计，超越了同类框架。Webx有哪些优势呢?

5. Webx的优势

5.1. 成熟可靠性

这个优势主要是针对阿里巴巴及属下网站而言。因为Webx在阿里巴巴和淘宝用了很多年。对于这种超大访问量的电子商务网站，Webx经受了考验，被证明是成熟可靠的。

5.2. 开放和扩展性

- 对Spring的直接支持 —— Spring是当今主流的轻量级框架。Webx 3.0和Spring MVC一样，完全建立在Spring框架之上，故可运用Spring的所有特性。
- 扩展性 —— Webx 3.0对Spring做了扩展，使Spring Bean不再是“bean”，而是升级成“组件”。一个组件可以扩展另一个组件，也可以被其它组件扩展。这种机制造就了Webx的非常好的扩展性，且比未经扩展的Spring更易使用。
- 开放性 —— Webx被设计成多个层次，层次间的分界线很清晰。每个层次都足够开放和易于扩展。你可以使用全部的Webx，也可以仅仅使用到Webx的任何一个层次。

6. Webx还缺少什么?

和目前快速发展的开源框架相比，Webx似乎不够时髦，因为它还缺少对很多流行功能的直接支持 —— 并非不支持，而是没有方便的方法来直接完成。例如:

- 目前Webx只支持服务端的表单验证，而没有直接支持客户端的JS验证。
- 目前Webx没有直接支持AJAX编程。
- 目前Webx没有直接支持REST编程。

- 目前Webx没有直接支持Web Flow。

凡是缺少的功能，我们将在未来的版本中陆续加上。

部分 I. Webx框架概览

第 1 章 Webx总体介绍

本章概要地介绍了Webx框架的整体结构和设计。如果你想了解更多，请参考其它详细文档。

1.1. 设计理念

1.1.1. 框架的本质

图 1.1. 框架结构的建筑

应用框架（Application Framework），让人联想到建筑的框架（Frame Structure）。

- 建筑框架确定了整个建筑的结构；应用框架确定了应用的结构。
- 建筑框架允许你在不改变结构的基础上，自由改变其内容。例如，你可以用墙体随意分隔房间。应用框架允许你在不改变整体结构的基础上，自由扩展功能。

可以这样说，框架的本质就是“扩展”。维基百科这样定义描写“软件框架”，它说一个软件框架必须符合如下要素：

Inversion of Control 反转控制	应用的流程不是由应用控制的，而是由框架控制的。
Default Behavior 默认行为	框架会定义一系列默认的行为。
Extensibility 扩展性	应用可以扩展框架的功能，也可以修改框架的默认行为。
Non-modifiable Framework Code 框架本身不可更改	框架在被扩展时，自身的代码无须被改变。

在一个框架中，实现丰富的功能固然重要，然而更重要的是：*建立良好的扩展机制*。我们知道，Webx目前虽然欠缺一些流行的功能。然而Webx却有一个良好的扩展机制，来支持开发者增加新的功能。

1.1.2. 基础框架

纵观开源的Web框架，做得比较好的框架，都有一个共性 —— 它们并不是简单地实现Web应用所需要的功能（诸如Action、模板、表单验证等），而是把框架建立在另一个基础框架之

上。这个基础框架的作用是：组装模块；提供扩展机制。建立在这种基础上的Web框架有很好的适应性和扩展性，可以应对Web应用不断变化和发展的需求。

- 早期的Turbine，建立在Service框架之上。
- Webwork，建立在Xwork框架之上。
- Tapestry，建立在HiveMind或Tapestry IOC框架之上。
- 早期的Struts 1.x由于没有一个轻量框架作为基础，因此很难扩展。而Struts 2.x使用了Webwork和Xwork，因此适用能力大为提高。
- Spring MVC，建立在Spring框架之上。

一个Web框架的好坏，往往不是由它所实现的具体功能的好坏决定的，而是由其所用的基础框架的好坏决定的。

Webx建立在SpringExt的基础上 —— SpringExt是对Spring的扩展。Spring是当今主流的轻量级框架。SpringExt没有损失任何Spring的功能，但它能够提供比Spring自身更强大的扩展能力。

1.1.3. 层次化

设计良好的模块，应该是层次化的。

例如，模块B扩展了模块A，同时被模块C扩展。这样就形成了A、B、C三个层次。

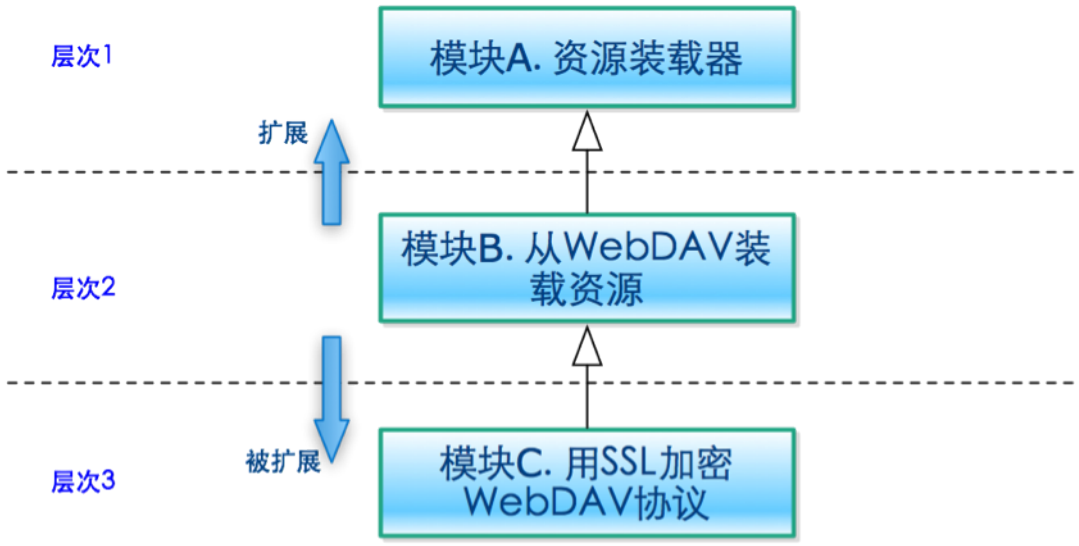


图 1.2. 模块的层次

如图所示，层次之间有如下的关系：

- 上层定义规则，下层定义细节；（上层、下层也可称为内层、外层）
- 上层是抽象的，下层是具体的；
- 越上层，越稳定（越少改变）；越下层，越易变。
- 依赖倒转（Dependency Inversion）。下层（具体）依赖上层（抽象），而不是上层依赖下层。

- 下层扩展上层时，不需要修改到上层的任何代码和配置。即符合开闭原则（Open-Closed Principle简称OCP – Open for extension, Closed for modification）。
- 每一层均可被替换。

层次化的设计，使软件中的每一个部分都可被增强或替换。

*层次化不是自然而然的，而是需要精心的设计。*设计一个层次化的组件，可以从下面几方面来考虑：

- 切分功能。每个组件专心做一件事。
- 分析哪些会改变，哪些不会改变。不变部分固化在组件中，可能会改变的部分抽象成接口，以便扩展。
- 考虑默认值和默认扩展。默认值和默认扩展应该是最安全、最常用的选择。对于默认值和默认扩展，用户在使用时不需要额外的配置。

Webx鼓励层次化的模块设计，而SpringExt提供了创建和配置层次化组件的机制。

1.2. Webx的层次

1.2.1. 三个大层次

很多用过Webx框架的人说起Webx，就想到：Webx如何处理页面、如何验证表单、如何渲染模板等等功能。事实上，这些只不过是Webx最外层、最易变、非本质的功能。

Webx框架不仅鼓励层次化设计，它本身也是层次化的。*你既可以使用全部的Webx框架，也可以只使用部分的Webx框架。*大体上，Webx框架可以划分成三个大层次，如图所示。

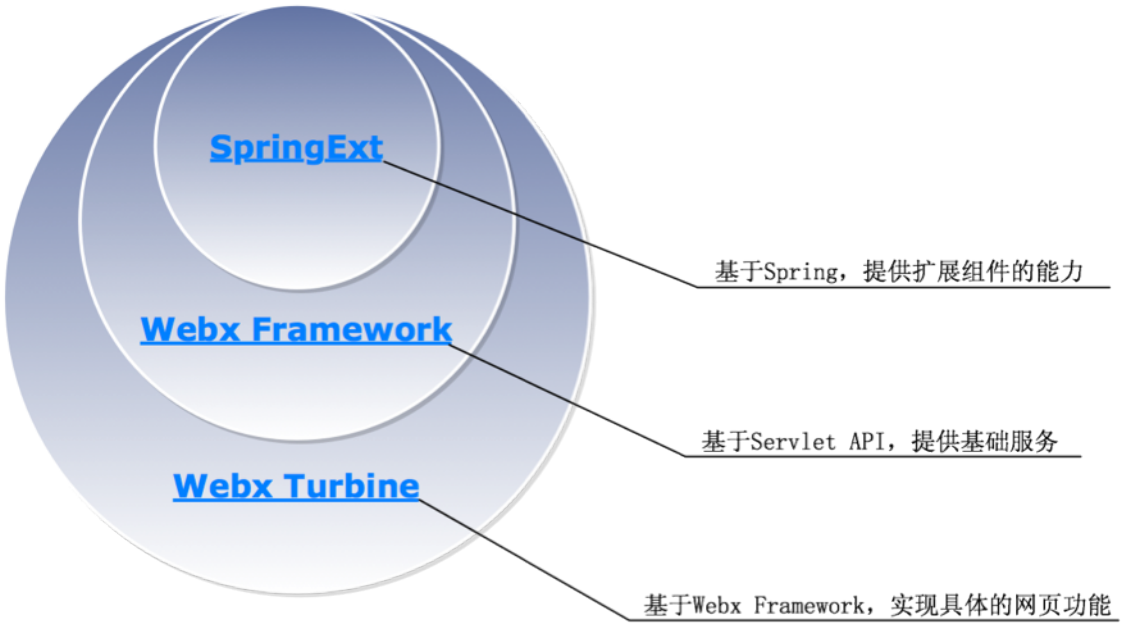


图 1.3. Webx的层次

1. SpringExt：基于Spring，提供扩展组件的能力。它是整个框架的基础。

2. Webx Framework：基于Servlet API，提供基础的服务，例如：初始化Spring、初始化日志、接收请求、错误处理、开发模式等。Webx Framework只和servlet及spring相关 —— 它不关心Web框架中常见的一些服务，例如Action处理、表单处理、模板渲染等。因此，事实上，你可以用Webx Framework来创建多种风格的Web框架。

3. Webx Turbine：基于Webx Framework，实现具体的网页功能，例如：Action处理、表单处理、模板渲染等。

1.2.2. 剪裁和定制Webx

并非所有的开发者都需要使用Webx的全部。下面列举几种情形。

1.2.2.1. 级别一：仅使用SpringExt，适用于非Web应用、单元测试

对于非Web应用和单元测试，但却想拥有Spring和SpringExt的功能，可以直接创建SpringExt容器：

例 1.1. 直接创建SpringExt容器

```
import java.io.File;
import org.springframework.core.io.FileSystemResource;
import com.alibaba.citrus.springext.support.context.XmlApplicationContext; ❶

...

XmlApplicationContext parentContext = new XmlApplicationContext(
        new FileSystemResource(new File(srcdir, "parent.xml")));  ❷

XmlApplicationContext context = new XmlApplicationContext(
        new FileSystemResource(new File(srcdir, "app.xml")), parentContext); ❸

Object mybean = context.getBean("mybean");
```

❶ 请注意代码所使用的`ApplicationContext`实现类为SpringExt扩展的类型（`c.a.c.springext.s.c.XmlApplicationContext`）。通过这个实现类，你除了可以使用原来Spring的所有功能以外，还可以使用SpringExt的所有功能，包括：Schema、Configuration Points和Contributions、ResourceLoadingService等。

❷ 这行代码从一个配置文件中创建容器。

❸ 这行代码创建了一个子容器。多个子容器和父容器之间可组成一个树状级联的容器结构。在子容器中可以访问到所有父容器中的beans和服务，但反过来是不成立的。

1.2.2.2. 级别二：仅使用SpringExt及Web组件，在此基础上运行Spring MVC、Struts等非webx框架

非webx框架也可以使用SpringExt的全部功能。

例 1.2. 修改/WEB-INF/web.xml，让非webx框架支持SpringExt

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/j2ee  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd
    ">

    <!-- 初始化日志系统，装载/WEB-INF/log4j.xml或/WEB-INF/logback.xml -->
    <listener>
        <listener-class>com.alibaba.citrus.logconfig.LogConfiguratorListener</listener-class>
    </listener>

    <!-- 初始化Spring容器，装载/WEB-INF/webx.xml, /WEB-INF/webx-*.xml -->
    <listener>
        <listener-class>com.alibaba.citrus.webx.context.WebxContextLoaderListener</listener-class> ❶
    </listener>

    <!-- 下面配置：Spring MVC、Struts的filter、servlet、mapping... -->
    ……

</web-app>
```

❶ 这里使用了webx的WebxContextLoaderListener来初始化spring容器，而不是用spring原生的ContextLoaderListener。不用担心，前者完全兼容后者。事实上前者是从后者派生而来的。

1.2.2.3. 级别三：仅使用Webx Framework，创造新的Web框架

也许你想做一个新的Web框架 —— 因为你并不想使用Webx Turbine中提供的页面处理的方案，但你仍然可以使用Webx Framework所提供的服务，例如：错误处理、开发模式等。

例 1.3. 修改/WEB-INF/webx.xml，以创建新的WEB框架

```
<webx-configuration xmlns="http://www.alibaba.com/schema/services">
    <components defaultControllerClass="com.myframework.MyController"> ❶
        <rootController class="com.myframework.MyRootController" /> ❷
    </components>
</webx-configuration>
```

❶ MyController扩展了AbstractWebxController。

❷ MyRootController扩展了AbstractWebxRootController。

这个方案非常适合作为一个新Web框架的起点 —— 免去了创建Servlet/Filter、初始化Spring容器、处理request/response等繁杂事务，并且完全支持SpringExt的所有功能，此外还包含了错误处理、开发模式等Webx Framework中的一切便利。

另一种以Webx Framework为基础的创建新框架的方法，是从pipeline入手。*通过pipeline，理论上可以实现任何框架的功能*。

1.2.2.4. 级别四：使用整个Webx框架，定制Turbine

假如你想使用几乎大部分Webx的功能，但希望对少数步骤进行改进，你可以修改pipeline。

Webx Turbine本身定义了一套pipeline的实现，但是你完全可以去修改它：插入一些步骤、删除一些步骤、修改一些步骤 —— 所有都取决于你。

最常见的一个需求，是在Webx Turbine中添加权限验证 —— 只需要插入一个步骤就可以做到了。

1.3. 本章总结

Webx框架是一个稳定、强大的Web框架。倒不是说它实现了所有的功能，而是它建立在SpringExt的基础上，具有超强的扩展能力。你可以使用全部的Webx，也可以使用部分Webx。你也可以比较容易地用SpringExt做出自己的可扩展组件。

第 2 章 SpringExt

Webx是一套基于Java Servlet API的通用Web框架。Webx致力于提供一套极具扩展性的机制，来满足Web应用不断变化和发展的需求。而SpringExt正是这种扩展性的基石。SpringExt扩展了Spring，在Spring的基础上提供了一种扩展功能的新方法。

本章将告诉你SpringExt是什么？它能做什么？本章不会涉及太深的细节，如果你想了解更多，请参考其它文档。

2.1. 用SpringExt装配服务

在Webx中有一个非常有用的ResourceLoadingService。现在我们以这个服务为例，来说明SpringExt的用途。

ResourceLoadingService是一个可以从各种输入源中（例如从File System、Classpath、Webapp中）查找和读取资源文件的服务。有点像Linux的文件系统——你可以在一个统一的树形目录结构中，定位（mount）任意文件系统，而应用程序不需要关心它所访问的资源文件属于哪个具体的文件系统。

ResourceLoadingService的结构如图所示。这是一个既简单又典型的面向对象的设计。

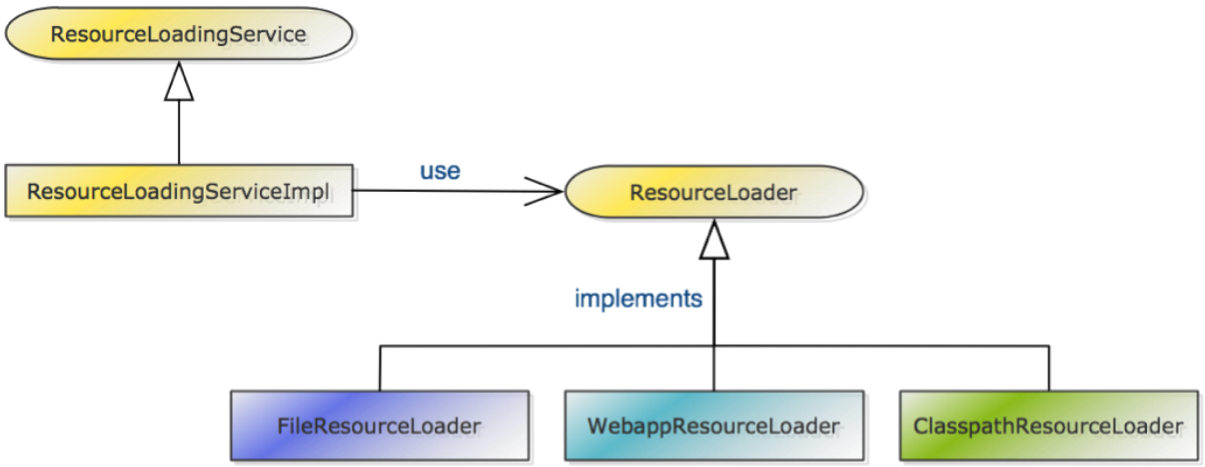


图 2.1. Resource Loading服务的设计

下面我们尝试在Spring容器中装配ResourceLoadingService服务。为了更好地说明问题，下文所述的Spring配置是被简化的，未必和ResourceLoadingService的真实代码相吻合。

2.1.1. Spring Beans

在Spring 2.0以前，你只能装配beans，就像下面这样：

例 2.1. 用Spring Beans装配Resource Loading服务

```
<bean id="resourceLoadingService" class="com.alibaba...ResourceLoadingServiceImpl">
    <property name="mappings">
        <map>
            <entry key="/file" value-ref="fileLoader" />
            <entry key="/webroot" value-ref="webappLoader" />
        </map>
    </property>
</bean>

<bean id="fileLoader" class="com.alibaba...FileResourceLoader">
    <property name="basedir" value="${user.home}" />
</bean>

<bean id="webappLoader" class=" com.alibaba...WebappResourceLoader" />
```

以上是一个典型的Spring beans的配置方案。这种方案简单易行，很好地体现了Spring的基础理念：IoC（Inversion of Control，依赖反转）。ResourceLoadingServiceImpl并不依赖FileResourceLoader和WebappResourceLoader，它只依赖它们的接口ResourceLoader。至于如何创建FileResourceLoader、WebappResourceLoader、需要提供哪些参数，这种琐事全由spring包办。

然而，其实spring本身并不了解如何创建ResourceLoader的对象、需要用哪些参数、如何装配和注入等知识。这些知识全靠应用程序的装配者（assembler）通过上述spring的配置文件来告诉spring的。也就是说，尽管ResourceLoaderServiceImpl类的作者不需要关心这些琐事，但还是有人得关心。

为了说明问题，我先定义两个角色：“服务提供者”和“服务使用者”（即“装配者”）。在上面的例子中，ResourceLoadingService的作者就是服务的提供者，使用ResourceLoadingService的人，当然就是服务使用者。服务使用者利用spring把ResourceLoadingService和ResourceLoader等其它服务装配在一起，使它们可以协同工作。当然这两个角色有时会是同一个人，但多数情况下会是两个人。因此有必要把这两个角色的职责区分清楚，才能合作。

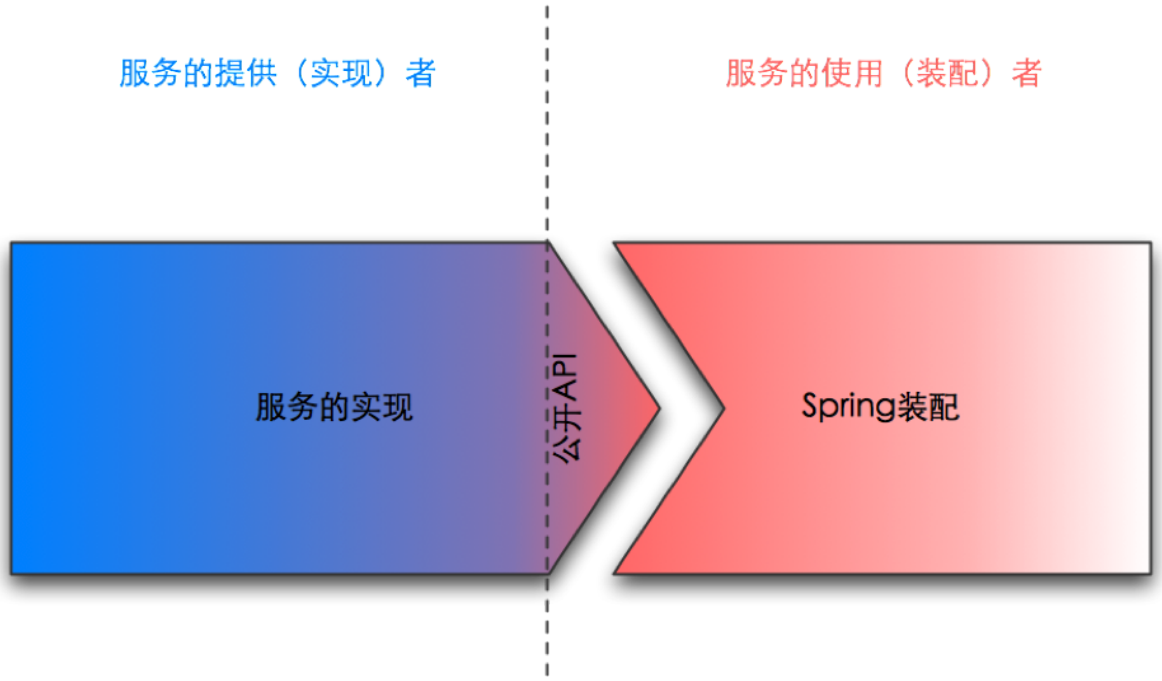


图 2.2. 服务提供者和使用者的关系

如图所示。虚线左边代表“服务提供者”的职责，虚线右边代表“服务使用者”（即“装配者”）的职责。

从图中可以看到，Spring的配置文件会依赖于服务实现类的公开API。装配者除非查看源代码（如ResourceLoadingServiceImpl的源码）或者API文档才能精确地获知这些API的细节。这有什么问题呢?

- 没有检验机制，错误必须等到运行时才会被发现。装配者仅从spring配置文件中，无法直观地了解这个配置文件有没有写对？例如：应该从constructor args注入却配成了从properties注入；写错了property的名称；注入了错误的类型等等。
- 无法了解更多约束条件。即使装配者查看API源码，也未必能了解到某些约束条件，例如：哪些properties是必须填写的，哪些是可选的，哪些是互斥的?
- 当服务的实现被改变时，Spring配置文件可能会失败。因为Spring配置文件是直接依赖于服务的实现，而不是接口的。接口相对稳定，而实现是可被改变的。另一方面，这个问题也会阻碍服务提供者改进他们的服务实现。

难怪有人诟病Spring说它只不过是用XML来写程序代码而已。

2.1.2. Spring Schema

这种情况直到Spring 2.0发布以后，开始有所改观。因为Spring 2.0支持用XML Schema来定义配置文件。同样的功能，用Spring Schema来定义，可能变成下面的样子:

例 2.2. 用Spring Schema装配Resource Loading服务

```
<resource-loading id="resourceLoadingService"
                  xmlns="http://www.alibaba.com/schema/services/resource-loading">
    <resource pattern="/file">
        <file-loader basedir="${user.home}" />
    </resource>
    <resource pattern="/webroot">
        <webapp-loader />
    </resource>
</resource-loading>
```

怎么样？这个配置文件是不是简单很多呢？和直接使用Spring Beans配置相比，这种方式有如下优点：

- 很明显，这个配置文件比起前面的Spring Beans风格的配置文件简单易读得多。因为在这个spring配置文件里，它所用的“语言”是“领域相关”的，也就是说，和ResourceLoadingService所提供的服务内容相关，而不是使用像bean、property这样的编程术语。这样自然易读得多。

- 它是可验证的。你不需要等到运行时就能验证其正确性。任何一个支持XML Schema的标准XML编辑器，包括Eclipse自带的XML编辑器，都可以告诉你配置的对错。

- 包含更多约束条件。例如，XML Schema可以告诉你，哪些参数是可选的，哪些是必须填的；参数的类型是什么等等。

- *服务的实现细节对装配者隐藏*。当服务实现改变时，只要XML Schema是不变的，那么Spring的配置就不会受到影响。

以上优点中，最后一点是最重要优点。通过Spring Schema来定义配置文件，装配者无须再了解诸如“ResourceLoadingService的实现类是什么”、“需要什么参数”等细节。那么Spring是如何得知这些内容呢?

奥秘在于所有的schema都会有一个“解释器”和它对应（即BeanDefinitionParser）。这个解释器负责将符合schema定义的XML配置，转换成Spring能解读的beans定义。*解释器是由服务的开发者来提供的* —— 在本例中，ResourceLoadingService的开发者会提供这个解释器。

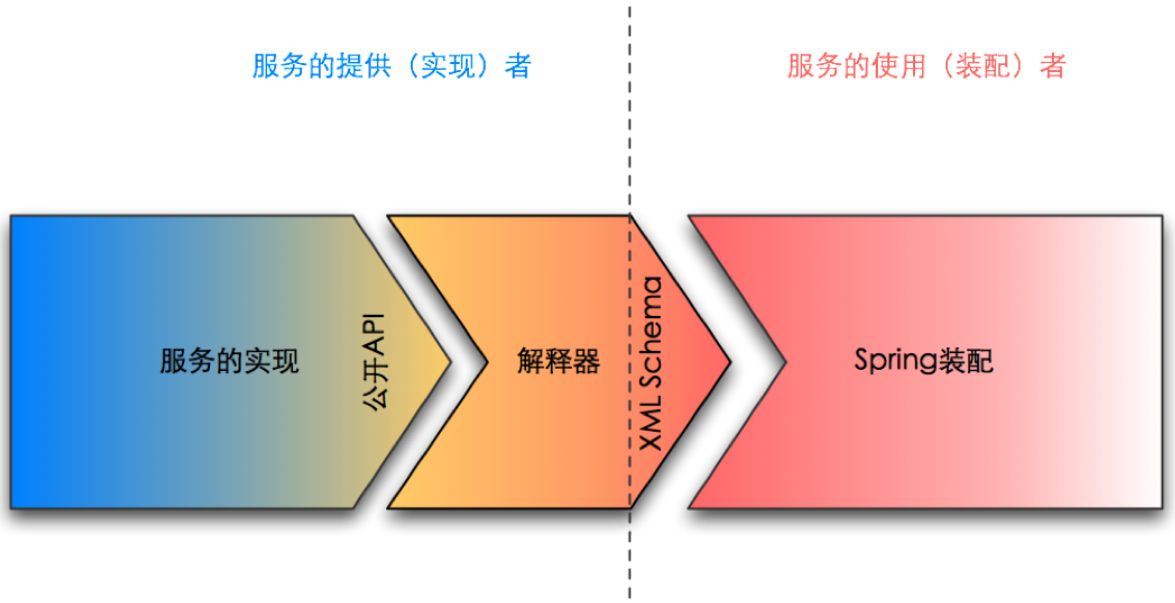


图 2.3. 用Schema改善服务角色之间的关系

如图所示，虚线右侧的装配者，不再需要了解服务具体实现类的API，它只要遵循标准的XML Schema定义来书写spring配置文件，就可以得到正确的配置。这样一来，虚线左侧的服务提供者就有自由可以改变服务的实现类，他相信只要服务的接口和XML Schema不改变，服务的使用者就不会受影响。

将和具体实现相关的工作，例如提供类名、property名称和类型等工作，交还给服务的提供者，使服务的使用者（即装配者）可以用它所能理解的语言来装配服务，这是Spring Schema所带来的核心价值。

然而，Spring Schema有一个问题 —— 它是不可扩展的。

仍以ResourceLoadingService为例。仅管在API层面， ResourceLoadingService支持任何对ResourceLoader接口的扩展，例如，你可以添加一种新的DatabaseResourceLoader，以便读取数据库中的资源。但在Spring配置文件上，你却无法自由地添加新的元素。比如：

例 2.3. 尝试在Spring Schema所装配的Resource Loading服务中，添加新的装载器

```
<resource-loading id="resourceLoadingService"
                  xmlns="http://www.alibaba.com/schema/services/resource-loading">
    <resource pattern="/file">
        <file-loader basedir="${user.home}" />
    </resource>
    <resource pattern="/webroot">
        <webapp-loader />
    </resource>
    <resource pattern="/db">
        <database-loader connection="jdbc:mysql:mydb" /> ❶
    </resource>
</resource-loading>
```

❶ 装配者希望在这里添加一种新的装载器：`database-loader`。然而，如果在设计`<resource-loading>`的schema时，并没有预先考虑到`database-loader`这种情况，那么这段配置就会报错。

使用Spring Schema时，装配者无法自主地往Spring配置文件中增加新的Resource Loader类型，除非通知服务提供者去修改`<resource-loading>`的schema —— 然而这违反了面向对象设计中的基本原则 —— OCP（Open Closed Principle）。OCP原则是面向对象设计的强大之源。它使得我们可以轻易地添加新的功能，却不需要改动老的代码；它使设计良好的代码成果可以被叠加和组合，以便实现更复杂的功能。

从本质意义来讲，Schema是API的另一种表现形式。你可以把Schema看作一种接口，而接口的实质是服务的提供者与使用者之间的合约（contract）。可惜的是，我们只能在传统API层面来贯彻OCP原则，却无法在Schema上同样遵循它。我们无法做到不修改老的schema，就添加新的元素 —— 这导致Spring Schema的作用被大大削弱。

2.1.3. SpringExt Schema

SpringExt改进了Spring，使得Spring Schema可以被扩展。下面的例子对例 2.2 “用Spring Schema装配Resource Loading服务”作了少许修改，使之能被扩展。

例 2.4. 用SpringExt Schema装配Resource Loading服务

```
<resource-loading id="resourceLoadingService"
                  xmlns="http://www.alibaba.com/schema/services"
                  xmlns:loaders="http://www.alibaba.com/schema/services/resource-loading/loaders"> ❶
    <resource pattern="/file">
        <loaders:file-loader basedir="${user.home}" /> ❷
    </resource>
    <resource pattern="/webroot">
        <loaders:webapp-loader /> ❸
    </resource>
</resource-loading>
```

❶ 重新定义namespaces —— 将`ResourceLoader`和`<resource-loading>`所属的namespace分离。

❷❸ 将`file-loader`和`webapp-loader`放在`loaders`名字空间中，表示它们是Resource Loaders的扩展。

上面的配置文件和前例中使用Spring Schema的配置文件差别很小。没错，*SpringExt Schema和Spring Schema是完全兼容的*！唯一的差别是，我们把`ResourceLoader`和`<resource-loading>`所属的namespace分开了，然后将`ResourceLoader`的配置放在专属的namespace“loaders”中。例如：`<loaders:file-loader>`。这样一来，我们就有办法在不修改`<resource-loading>`的schema的前提下，添加新的ResourceLoader实现。例如我们要添加一种新的ResourceLoader扩展 —— `DatabaseResourceLoader`，只需要做以下两件事：

1. 将包含`DatabaseResourceLoader`所在的jar包添加到项目的依赖中。如果你是用maven来管理项目，那么意味着你需要修改一下项目的`pom.xml`。

2. 在spring配置文件中添加如下行：

例 2.5. 在SpringExt Schema所装配的Resource Loading服务中，添加新的装载器

```
<resource-loading id="resourceLoadingService"
                  xmlns="http://www.alibaba.com/schema/services"
                  xmlns:loaders="http://www.alibaba.com/schema/services/resource-loading/loaders">
    <resource pattern="/file">
        <loaders:file-loader basedir="${user.home}" />
    </resource>
    <resource pattern="/webroot">
        <loaders:webapp-loader />
    </resource>
    <resource pattern="/db">
        <loaders:database-loader connection="jdbc:mysql:mydb" /> ❶
    </resource>
</resource-loading>
```

❶ 添加一个新的loader，而无须改变<resource-loading>的schema。

完美！你无须通知ResourceLoadingService的作者去修改它的schema，一种全新的ResourceLoader扩展就这样被注入到ResourceLoadingService中。正如同你在程序代码里，无须通知ResourceLoadingService的作者去修改它的实现类，就可以创建一种新的、可被ResourceLoadingService调用的ResourceLoader实现类。这意味着，我们在Spring配置文件的层面上，也满足了OCP原则。

2.2. SpringExt原理

2.2.1. XML Schema中的秘密

下面这段配置是例 2.5 “在SpringExt Schema所装配的Resource Loading服务中，添加新的装载器”的spring配置文件的片段。

```
<resource-loading>
    ...
    <resource pattern="/db">
        <loaders:database-loader connection="jdbc:mysql:mydb" />
    </resource>
</resource-loading>
```

其中，<resource-loading>是由resource-loading.xsd这个schema来定义的。而在开发resource-loading服务的时候，database-loader这种新的扩展还不存在 —— 也就是说，*resource-loading.xsd对于database-loader一无所知*。可为什么以上配置能通过XML Schema的验证呢？我们只需要查看一下resource-loading.xsd就可以知道答案了：

例 2.6. Schema片段：<resource-loading>中如何定义loaders

```
<xsd:element name="resource" type="ResourceLoadingServiceResourceType">
<xsd:complexType name="ResourceLoadingServiceResourceType">
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:any namespace="http://www.alibaba.com/schema/services/resource-loading/loaders" /> ❶
    </xsd:choice>
    <xsd:attribute name="pattern" type="xsd:string" use="required" />
</xsd:complexType>
```

❶ 这里运用了XML Schema中的<xsd:any>定义，相当于说：<resource> element下面，可以跟任意多个<loaders:*> elements。

<xsd:any>定义只关心namespace，不关心element的名称，自然可以接受未知的<database-loader> element，前提是<database-loader>的namespace是“http://www.alibaba.com/schema/services/resource-loading/loaders”。

在这段配置中，<loaders:database-loader>标签通知SpringExt：将database-loader的实现注入到resource-loading的服务中。这种对应关系是如何建立起来的呢?

在XML里，loaders前缀代表namespace：“http://www.alibaba.com/schema/services/resource-loading/loaders”；但对SpringExt而言，它还代表一个更重要的意义：*扩展点*，或称为*ConfigurationPoint*。ConfigurationPoint将namespace和可扩展的ResourceLoader接口关联起来。

在XML里，database-loader代表一个XML element；但对SpringExt而言，它还代表一个更重要的意义：*捐献*，或称为*Contribution*。Contribution将element和对ResourceLoader接口的具体扩展关联起来。

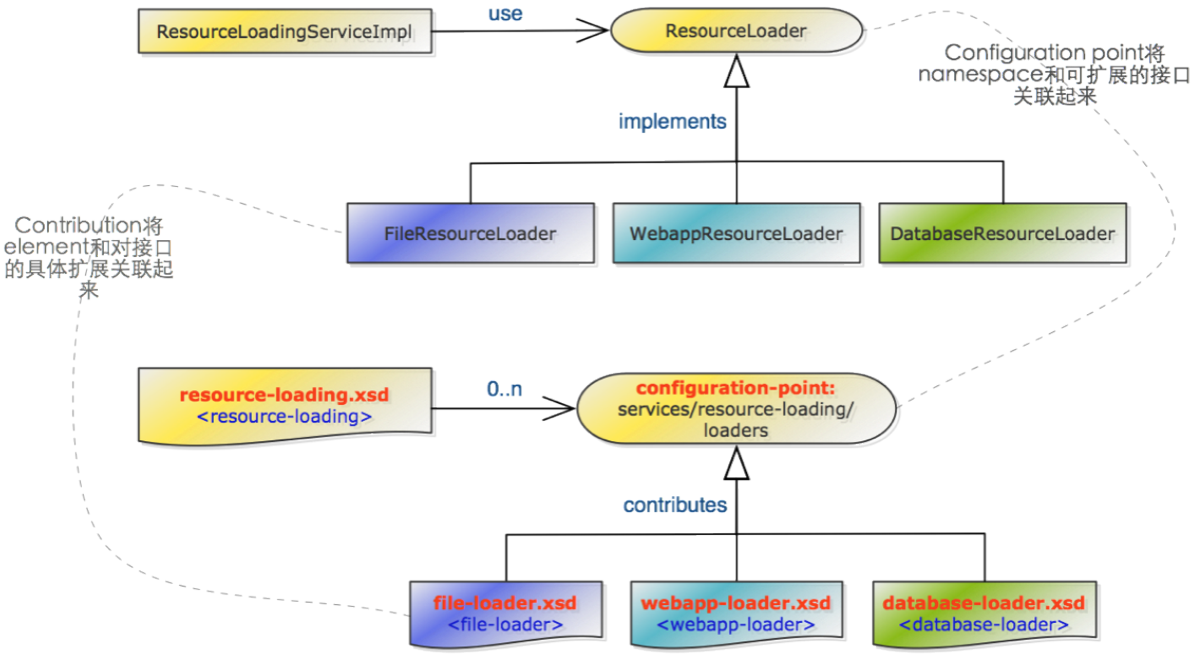


图 2.4. SpringExt的概念：扩展点和捐献

2.2.2. 扩展点，Configuration Point

SpringExt用“扩展点，Configuration Point”来代表一个可被扩展的接口。每个扩展点都：

- 对应一个唯一的名称，例如：*services/resource-loading/loaders*。
- 对应一个唯一的namespace，例如：http://www.alibaba.com/schema/*services/resource-loading/loaders*。
- 对应一个唯一的schema，例如：*services-resource-loading-loaders*.xsd。

2.2.3. 捐献，Contribution

SpringExt把每一个对扩展点的具体扩展称作“捐献，Contriubtion”。每个捐献都：

- 在对同一扩展点的所有捐献中，拥有一个唯一的名字，例如：*file-loader*，*webapp-loader*，*database-loader*等。
- 对应一个唯一的schema，例如：
  - services/resource-loading/loaders/*file-loader*.xsd
  - services/resource-loading/loaders/*webapp-loader*.xsd
  - services/resource-loading/loaders/*database-loader*.xsd

2.2.4. 组件和包

在前面的例子中，resource-loading服务调用了loaders扩展点，而file-loader、webapp-loader等则扩展了loaders扩展点。然而事实上，resource-loading服务本身也是对另一个扩展点“services”的扩展。services扩展点是Webx内部定义了一个*顶级扩展点*。

在SpringExt中，一个模块既可以成为别的模块的扩展，也可以被别的模块来扩展。这样的模块被称为“*组件*”。

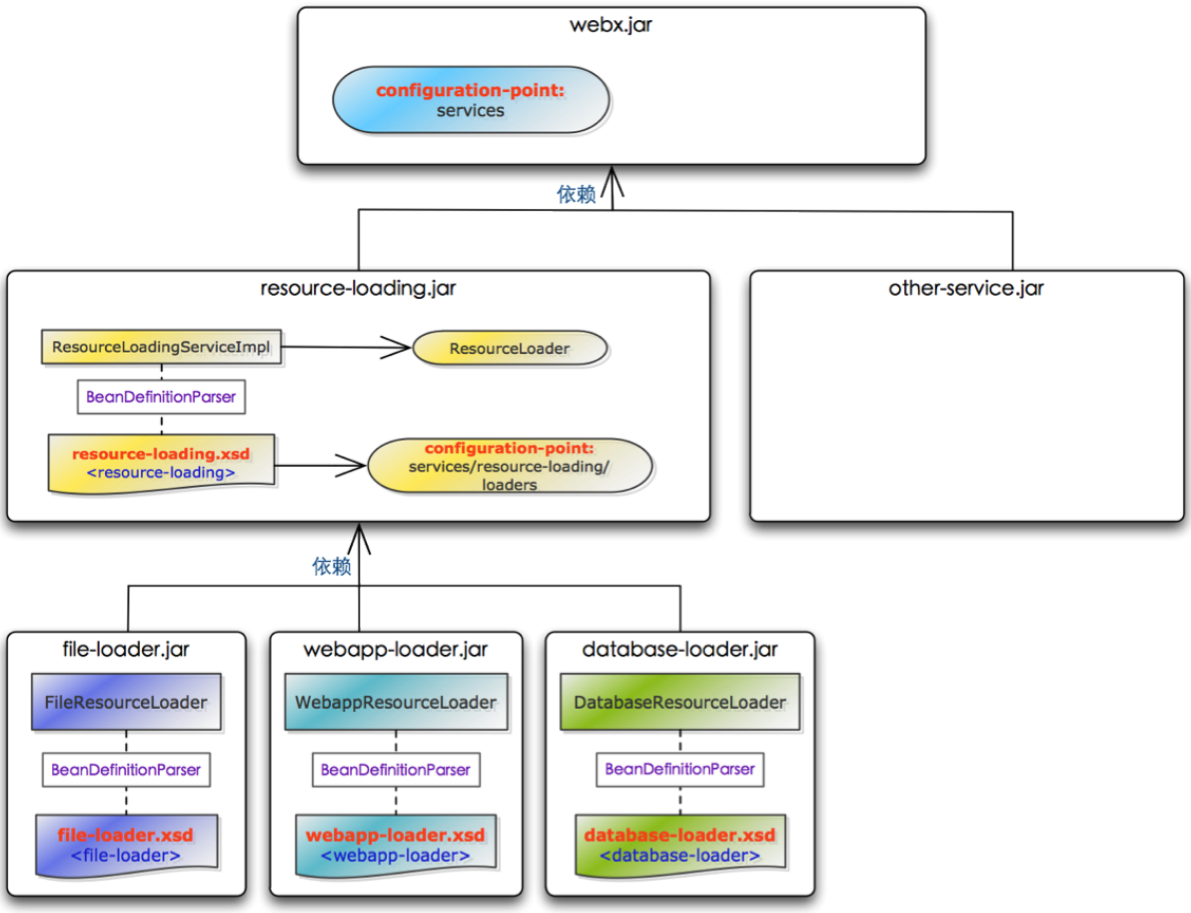


图 2.5. 组件

如图所示，resource-loading组件既扩展了services扩展点，又可被其它组件所扩展。

当你需要增加一种新的扩展时，你不需要改动原有包（例如`resource-loadings.jar`）中的任何内容，你只需要将新的扩展所在的jar包（例如`database-loader.jar`）加入到依赖表中即可。假如你使用maven来管理项目，意味着你需要修改项目的`pom.xml`描述文件，以便加入新的扩展包。

2.2.5. 取得Schemas

最后剩下的一个问题是，如何找到Schemas？为了找到schema，我们必须在Spring配置文件中指定Schema的位置。

例 2.7. 在XML中指定Schema Location

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans:beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:services="http://www.alibaba.com/schema/services"
    xmlns:loaders="http://www.alibaba.com/schema/services/resource-loading/loaders"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="
        http://www.alibaba.com/schema/services
            http://localhost:8080/schema/services.xsd ❶
        http://www.alibaba.com/schema/services/resource-loading/loaders
            http://localhost:8080/schema/services-resource-loading-loaders.xsd ❷
        http://www.springframework.org/schema/beans
            http://localhost:8080/schema/www.springframework.org/schema/beans/spring-beans.xsd ❸
    ">
    ...
    </beans:beans>
```

❶❷ 指定schema的位置。
❸

这里看起来有一点奇怪，因为它把schema的位置（`xsi:schemaLocation`）指向了一台本地服务器：`localhost:8080`。为什么这样做呢？要回答这个问题，先要搞清楚另一个问题：*有哪些部件需要用到schema?*

2.2.5.1. XML编辑器需要读取schemas

XML编辑器通过访问schema，可以实现两大功能：

- 语法提示的功能。

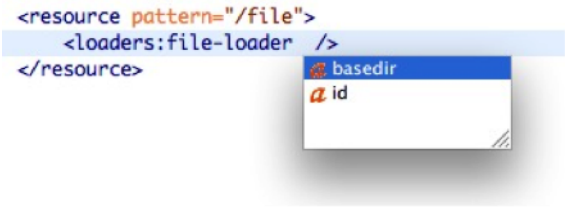


图 2.6. Eclipse XML编辑器弹出的语法提示

- 验证spring配置文件的正确性。

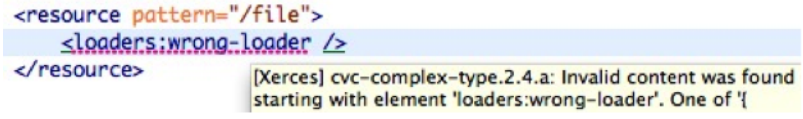


图 2.7. Eclipse XML编辑器验证spring配置文件时，显示的错误信息

XML编辑器取得schema内容的途径有两条，一条途径是访问schemaLocation所指示的网址。因此，

- 假如你声明的schemaLocation为：*http://www.alibaba.com/*schema/services.xsd，那么XML编辑器就会尝试访问www.alibaba.com服务器。
- 假如你声明的schemaLocation为：*http://www.springframework.org/*schema/beans/spring-beans.xsd，那么XML编辑器就会尝试访问www.springframework.org服务器。

然而，在外部服务器（例如www.alibaba.com和www.springframework.org）上维护一套schema是很困难的，因为：

- 你未必拥有外部服务器的控制权；
- 你很难让外部服务器上的schema和你的组件版本保持一致；
- 当你无法连接外部服务器的时候（例如离线状态），会导致XML编辑器无法帮你验证spring配置文件的正确性，也无法帮你弹出语法提示。

XML编辑器取得schema内容的另一条途径是将所有的schema转换成静态文件，然后定义一个标准的XML Catalog来访问这些schema文件。然而这种方法的难点类似于将schema存放在外部服务器上 —— 你很难让这些静态文件和你的组件版本保持一致。

SpringExt提供了一个解决方案，可以完全解决上述问题。你可以使用SpringExt所提供的maven插件，在localhost本机上启动一个监听8080端口的Schema Server，通过它就可以访问到所有的schema：

```
mvn springext:run
```

上述命令执行以后，打开浏览器，输入网址http://localhost:8080/schema就可以看到类似下面的内容：

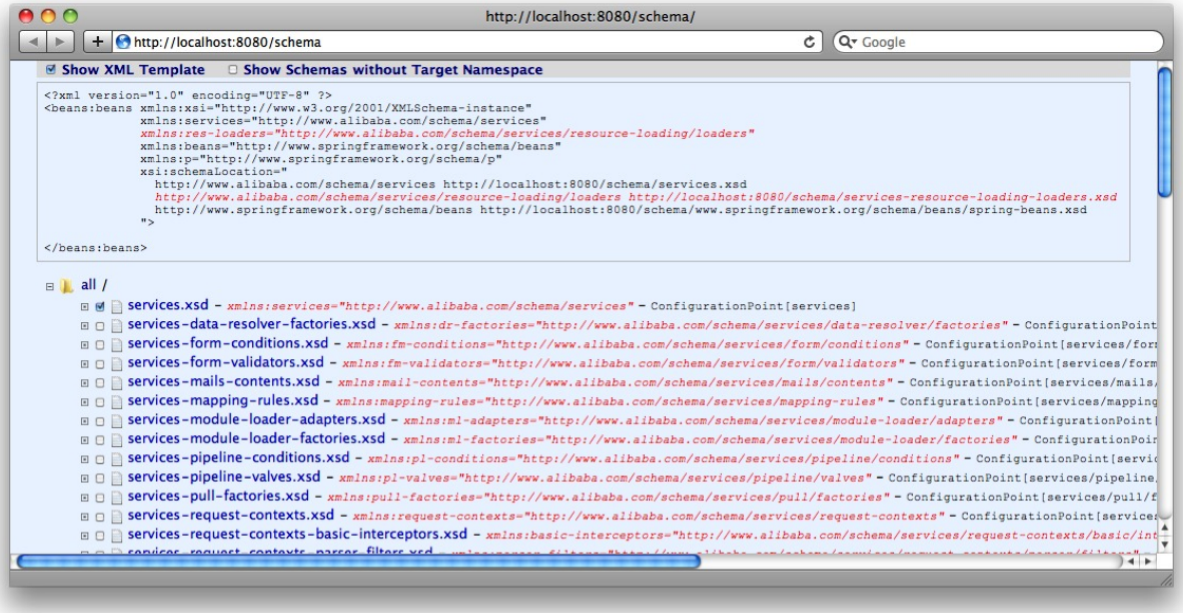


图 2.8. 用SpringExt maven插件罗列schemas

这就是为什么例 2.7 “在XML中指定Schema Location”中，把`schemaLocation`指向`localhost:8080`的原因。只有这样，才能让任何普通的XML编辑器不需要任何特殊的设置，就可以读到正确的schema。

2.2.5.2. SpringExt需要读取schemas

当SpringExt在初始化容器时，需要读取schema以验证spring配置文件。

请记住，*SpringExt永远不需要通过访问网络来访问schemas*。事实上，即使你把例 2.7 “在XML中指定Schema Location”中的schema的网址改成指向“外部服务器”的链接，SpringExt也不会真的去访问它们。例如：

- 将：`http://localhost:8080/schema/`*services.xsd*

  改成：`http://www.alibaba.com/schema/`*services.xsd*

- 将：`http://localhost:8080/schema/`*services-resource-loading-loaders.xsd*

  改成：`http://www.alibaba.com/schema/`*services-resource-loading-loaders.xsd*

- 将：`http://localhost:8080/schema/`*www.springframework.org/schema/beans/spring-beans.xsd*

  改成：`http://`*www.springframework.org/schema/beans/spring-beans.xsd*（这个就是spring原来的schema网址了）

以上修改在任何时候都不会影响Spring的正常启动。Spring是通过一种SpringExt定制的`EntityResolver`来访问schemas的。SpringExt其实只关注例子中*加亮*部分的schema网址，而忽略前面部分。

然而，如前所述，上面两种网址对于普通的XML编辑器来说是有差别的。因此，*SpringExt推荐总是以“`http://localhost:8080/schema`”作为你的`schemaLocation`网址的前缀*。下面的图总结了SpringExt是如何取得schemas的。

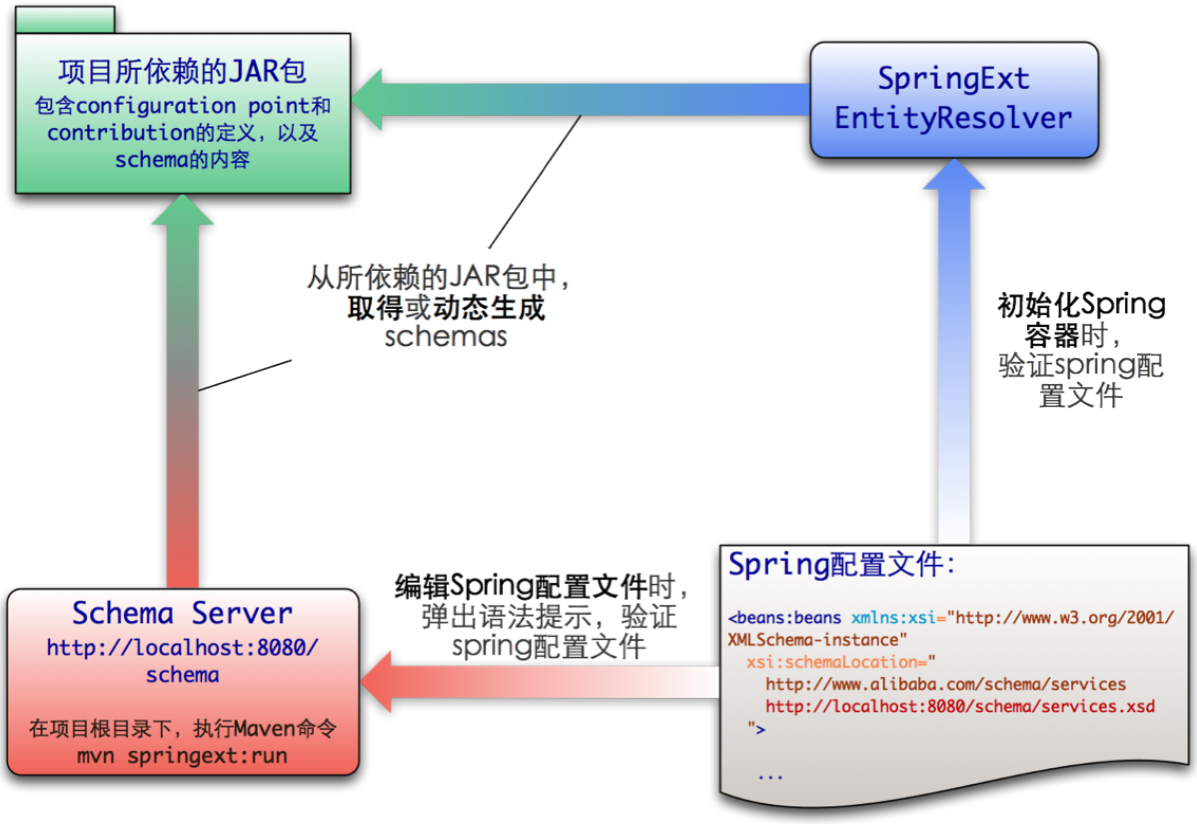


图 2.9. SpringExt如何取得schemas

2.3. SpringExt其它特性

SpringExt实际上是一个增强了的Spring的`ApplicationContext`容器。除了提供前面所说的Schema扩展机制以外，SpringExt还提供了一个增强的Resource Loading机制。前文例子中所说的Resource Loading服务是Webx中的真实功能，而且它能完全取代Spring原有的`ResourceLoader`功能 —— 也就是说，*应用程序并不需要直接调用ResourceLoading服务，它们可以直接使用Spring本身的ResourceLoader功能，其背后的ResourceLoading机制就会默默地工作*。

如果不加额外的配置，SpringExt context所用的`ResourceLoader`实现和Spring自带的完全相同。然而，你只要添加类似下面的配置，Spring的`ResourceLoader`就会被增强：

例 2.8. 配置Webx **`resource-loading`**服务

```
<services:resource-loading xmlns="http://www.alibaba.com/schema/services">
    <resource-alias pattern="/" name="/webroot" />

    <resource-alias pattern="/myapp" name="/webroot/WEB-INF" />

    <resource pattern="/webroot" internal="true">
        <res-loaders:webapp-loader />
    </resource>
    <resource pattern="/classpath" internal="true">
        <res-loaders:classpath-loader />
    </resource>
</services:resource-loading>
```

一种典型的Resource Loading服务的用途是读取CMS生成的模板。假设模板引擎从装载模板/templates，默认情况下，/templates就在webapp的根目录下。但是有一部分模板/templates/cms是由外部的内容管理系统（CMS）生成的，这些模板文件并不在webapp目录下。对此，我们只需要下面的配置：

例 2.9. 配置CMS目录

```
<resource pattern="/templates/cms">
    <res-loaders:file-loader basedir="${cms.dir}" />
</resource>
```

这样，在模板引擎浑然不知的情况下，我们就把/templates/cms目录指向webapp外部的一个文件系统目录，而保持/templates下其它模板的位置不变。

2.4. 本章总结

至此，我们简单领略了SpringExt所带来的好处和便利。SpringExt完全兼容Spring原来schema的概念和风格，但是却可以让schema像程序代码一样被扩展。Webx完全建立在SpringExt的基础上。这个基础决定了Webx是一个高度可扩展的框架，其配置虽然灵活，却又不失方便和直观。

第 3 章 Webx Framework

Webx是一套基于Java Servlet API的通用Web框架。整个Webx框架分成三个层次，本章将简单介绍其第二个层次：Webx Framework。事实上，这是第一个真正涉足WEB技术的层次。前一个层次SpringExt只是提供了一个通用的扩展机制。

Webx Framework负责完成一系列基础性的任务，如下表所示：

表 3.1. Webx Framework的任务

系统初始化	响应请求
初始化Spring容器	增强request/response/session的功能
初始化日志系统	提供pipeline流程处理机制
	异常处理
	开发模式

本章不会涉及太深的细节，如果你想了解更多，请参考其它文档。

3.1. Webx的初始化

3.1.1. 初始化级联的Spring容器

Webx Framework将负责创建一组级联的Spring容器结构。Webx所创建的Spring容器*完全兼容于Spring MVC所创建的容器*，可被所有使用Spring框架作为基础的WEB框架所使用。

例 3.1. 初始化Spring容器 - **/WEB-INF/web.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/j2ee  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd
    ">
    ...
    <listener>
        <listener-class>com.alibaba.citrus.webx.context.WebxContextLoaderListener</listener-class> ❶
    </listener>
    ...
</web-app>
```

❶ Webx利用`WebxContextLoaderListener`来初始化Spring，用来取代Spring的`ContextLoaderListener`。事实上，前者是从后者派生的。

Webx Framework将会自动搜索`/WEB-INF`目录下的XML配置文件，并创建下面这种级联的spring容器。

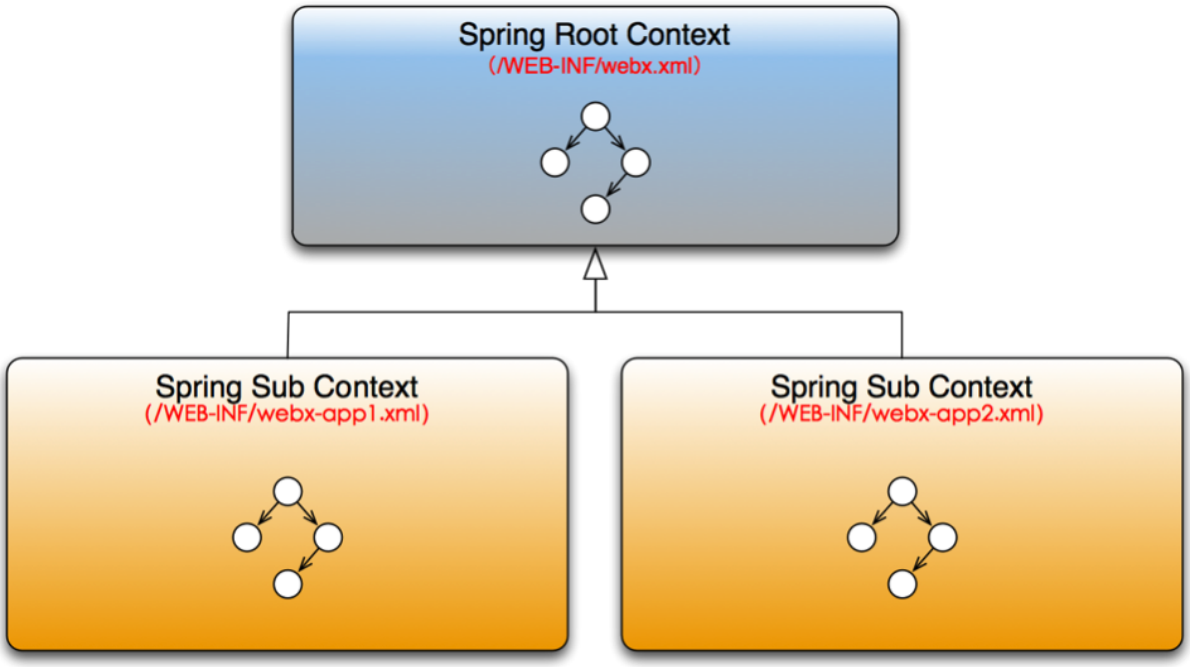


图 3.1. 级联的Spring容器

如图所示。Webx Framework将一个WEB应用分解成多个小应用模块：`app1`、`app2`，当然名字可以任意取。

- 每个小应用模块独享一个Spring Sub Context子容器。两个子容器之间的beans无法互相注入。
- 所有小应用模块共享一个Spring Root Context根容器。根容器中的bean可被注入到子容器的bean中；反之不可以。

将一个大的应用分解成若干个小应用模块，并使它们的配置文件相对独立，这是一种很不错的开发实践。然而，如果你的应用确实很简单，你不希望把你的应用分成多个小应用模块，那么，你还是需要配置至少一个小应用模块（子容器）。

3.1.2. 初始化日志系统

每个现代的WEB应用，都需要日志系统。流行的日志系统包括Log4j、Logback。

Webx Framework使用SLF4J作为它的日志框架。因此Webx Framework理论上支持所有日志系统。然而目前为止，它只包含了log4j和logback这两种日志系统的初始化模块（如有需要，可以扩充）。初始化日志系统很简单。

例 3.2. 初始化日志系统 - **/WEB-INF/web.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/j2ee  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd
    ">
    ...
    <listener>
        <listener-class>com.alibaba.citrus.logconfig.LogConfiguratorListener</listener-class> ❶
    </listener>
    ...
</web-app>
```

❶ Webx利用`LogConfiguratorListener`来初始化日志系统。

`LogConfiguratorListener`会根据你当前应用所依赖的日志系统（通常配置在maven project中），来自动选择合适的日志配置文件。

- 假设你的应用依赖了logback的jar包，那么listener就会查找`/WEB-INF/logback.xml`，并用它来初始化logback；
- 如果你的应用依赖了log4j的jar包，那么listener也会很聪明地查找`/WEB-INF/log4j.xml`配置文件。
- 假如以上配置文件不存在，listener会使用默认的配置 —— 把日志打印在控制台上。
- Listener支持对配置文件中的placeholders进行替换。
- Listener支持同时初始化多种日志系统。

注意

有关日志系统的使用方法，另有文档详细讲述。

3.2. Webx响应请求

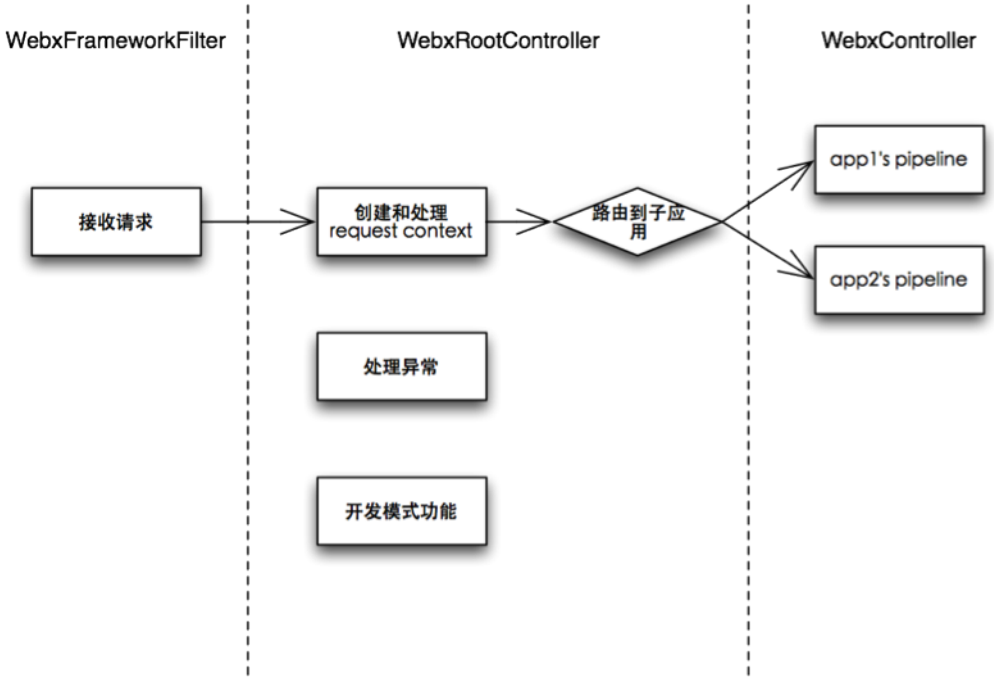


图 3.2. Webx Framework**如何响应请求**

当Webx Framework接收到一个来自WEB的请求以后，实际上它主要做了两件事：

1. 首先，它会增强request、response、session的功能，并把它们打包成更易使用的`RequestContext`对象。

2. 其次，它会调用相应子应用的pipeline，用它来做进一步的处理。

3. 假如在上面的过程中出现异常，则会触发Webx Framework处理异常的过程。

此外，Webx Framework还提供了一组辅助开发的功能，例如查看环境变量，查看schema等。这些功能只在开发模式生效，生产模式下自动关闭。

3.2.1. 增强request、response、session的功能

Webx Framework提供了一个request contexts服务。Request contexts服务利用`HttpServletRequestWrapper`和`HttpServletResponseWrapper`对request和response对象进行包装，以实现新的功能。

一个基本的request contexts的配置看起来是下面的样子：

例 3.3. 配置request contexts服务

```
<services:request-contexts xmlns="http://www.alibaba.com/schema/services/request-contexts">
    <basic />
    <buffered />
    <lazy-commit />
    <parser />
    <set-locale defaultLocale="zh_CN" defaultCharset="UTF-8" />
    ...
</services:request-contexts>

<services:upload sizeMax="5M" />
```

Request contexts所有的功能都是可配置、可扩展的 —— 它是基于SpringExt的扩展机制。

Request contexts所增加的功能对于所有的基于标准Servlet API的应用都是透明的 —— 这些应用根本不需要知道这些扩展的存在。例如，假如你在request contexts服务中配置了增强的session框架，那么所有通过标准的Servlet API取得session的应用，都将获得新功能：

例 3.4. 取得增强的session对象

```
HttpSession session = request.getSession();
```

再比如，只要你配置了upload服务，那么下面的调用将同样适用于`multipart/form-data`类型的请求（Servlet API本身是不支持upload表单的）：

例 3.5. 取得upload表单的参数

```
String value = request.getParameter("myparam");
```

注意

有关Request Contexts的原理和使用方法的详情，请参阅第 6 章 *Filter、Request Contexts和Pipeline*。

3.2.1.1. Request contexts中可用的功能

表 3.2. 可用的`RequestContext`扩展

名称	说明
`<basic>`	对输入、输出的数据进行安全检查，排除可能的攻击。例如：XSS过滤、CRLF换行回车过滤等。
`<buffered>`	对写入response中的数据进行缓存，以便于实现嵌套的页面。
`<lazy-commit>`	延迟提交response，用来支持基于cookie的session。
`<parser>`	解析用户提交的参数，无论是普通的请求，还是multipart/form-data这样的用于上传文件的请求。
`<set-locale>`	设置当前请求的区域（locale）、编码字符集（charset）。
`<rewrite>`	改写URL及参数，类似于Apache HTTPD Server中的rewrite模块。
`<session>`	增强的Session框架，可将session中的对象保存到cookie、数据库或其它存储中。

注意

有关以上所有Request Contexts的详情，请参阅第 7 章 *Request Contexts功能指南*和第 8 章 *Request Context之Session指南*。

3.2.1.2. 注入特殊对象

在Webx中，你可以这样做，例如：

例 3.6. 注入request、response、session

```
public class LoginAction {
    @Autowired
    private HttpServletRequest request;

    @Autowired
    private HttpServletResponse response;

    @Autowired
    private HttpSession session;
    ...
}
```

在这个例子中，`LoginAction`类可以是一个singleton。一般来说，你*不能把request scope的对象，注入到`singleton` scope的对象中*。但你*可以把`HttpServletRequest`、`HttpServletResponse`和`HttpSession`对象注入到singleton对象中*。为什么呢？原来，Request contexts服务对这几个常用对象进行了特殊处理，将它们转化成了singleton对象。

如果没有这个功能，那么我们就不得不将上例中的`LoginAction`配置成`request` scope。这增加了系统的复杂性，也成倍地降低了性能。而将`LoginAction`设置成singleton，只需要在系统启动时初始化一次，以后就可以快速引用它。

3.2.2. Pipeline流程机制

Webx Framework赋予开发者极大的自由，来定制处理请求的流程。这种机制就是pipeline。

Pipeline的意思是管道，管道中有许多阀门（Valve），阀门可以控制水流的走向。Webx Framework中的pipeline可以控制处理请求的流程的走向。如图所示。

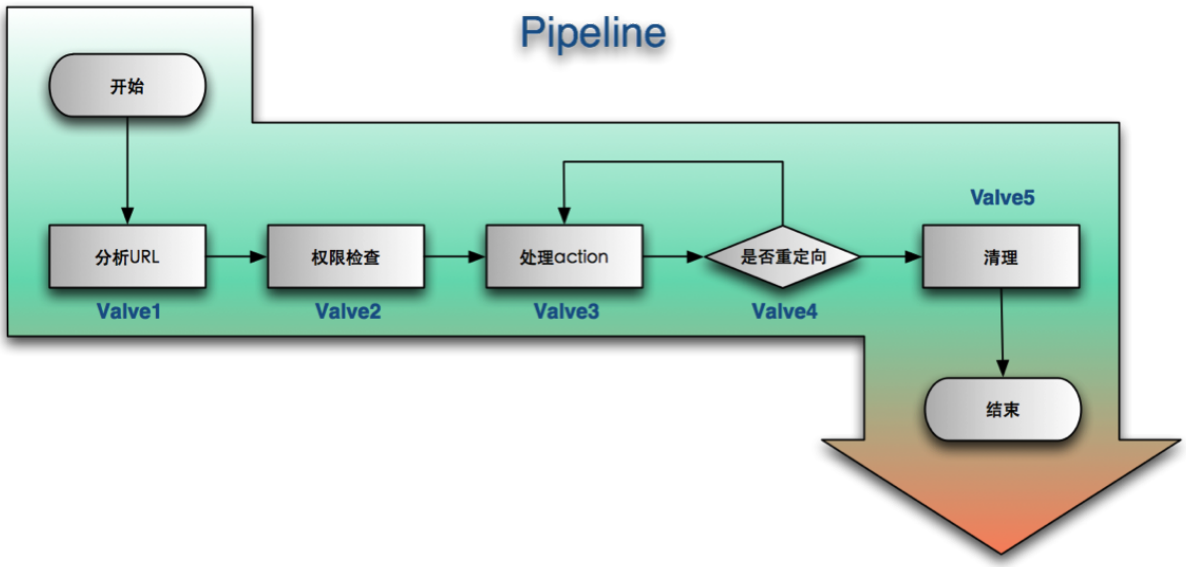


图 3.3. Pipeline工作原理示意

Webx Framework并没有规定管道的内容 —— 定制管道是应用开发者的自由。然而Webx Framework提供了一系列通用valves，你可以使用它们：

表 3.3. 通用valves

分类	Valves	说明
循环	`<while>`	有条件循环
	`<loop>`	无条件循环
选择分支	`<if>`	单分支
	`<choose><when><otherwise>`	多分支
中断	`<break>`	无条件中断
	`<break-if>` `<break-unless>`	有条件中断
	`<exit>`	无条件退出整个pipeline（结束所有的嵌套层次）
异常捕获	`<try-catch-finally>`	类似Java中的try-catch-finally结构
嵌套	`<sub-pipeline>`	创建嵌套的子pipeline。

注意

有关Pipeline的原理和使用方法的详情，请参阅第 6 章 *Filter、Request Contexts和Pipeline*。

3.2.3. 异常处理机制

当应用发生异常时，Webx Framework可以处理这些异常。

表 3.4. Webx如何处理异常

条件		处理
开发模式		展示详细出错信息。
生产模式	假如存在exception pipeline	用exception pipeline来处理异常；
	不存在exception pipeline	显示web.xml中定义的默认错误页面。

3.2.4. 开发模式工具

Webx Framework提供了一个开关，可以让应用运行于“*生产模式（Production Mode）*”或是“*开发模式（Development Mode）*”。

例 3.7. 配置运行模式

```
<services:webx-configuration>
    <services:productionMode>${productionMode:true}</services:productionMode> ❶
</services:webx-configuration>
```

❶ 使用这行配置，并且在启动应用服务器时指定参数“`-DproductionMode=false`”，就会让Webx以开发模式启动。

在开发模式下，会有一系列不同于生产模式的行为。

- 不同的主页 —— 在开发模式的主页中，可以查看和查询系统内部的信息。

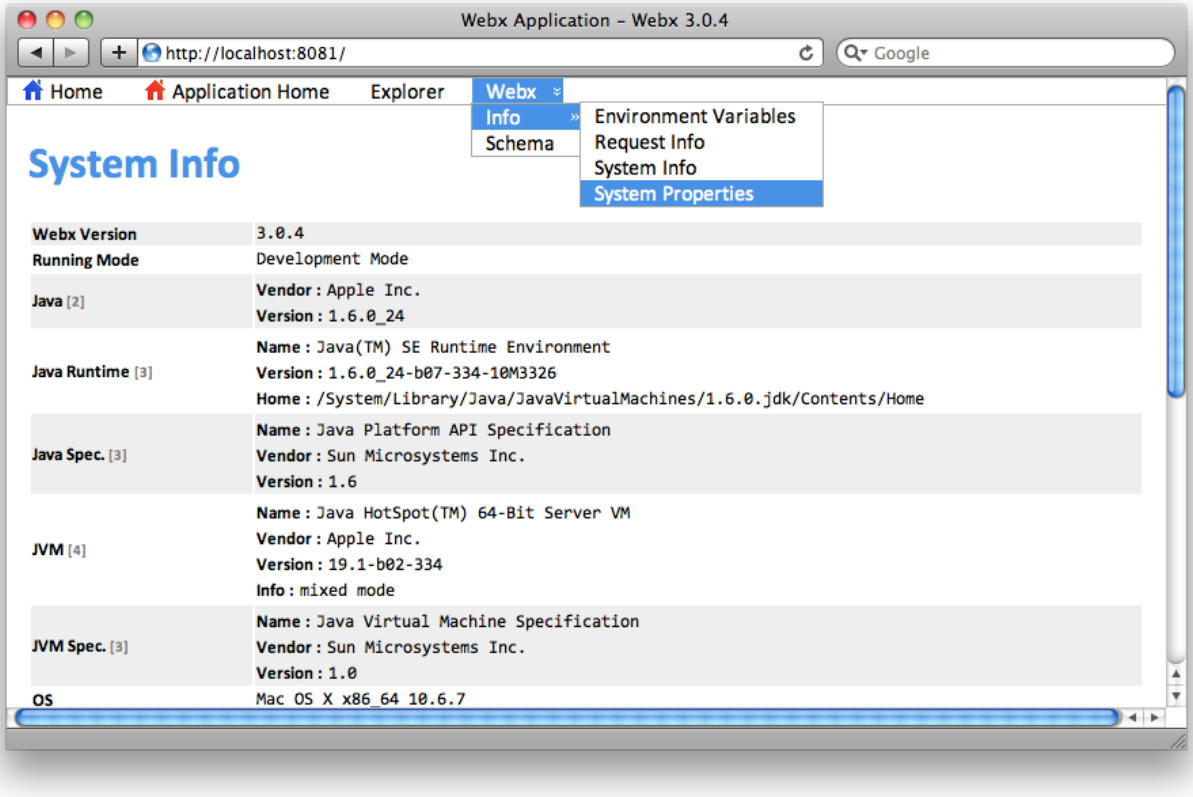


图 3.4. 开发模式的主页

- 不同的详细出错页面。

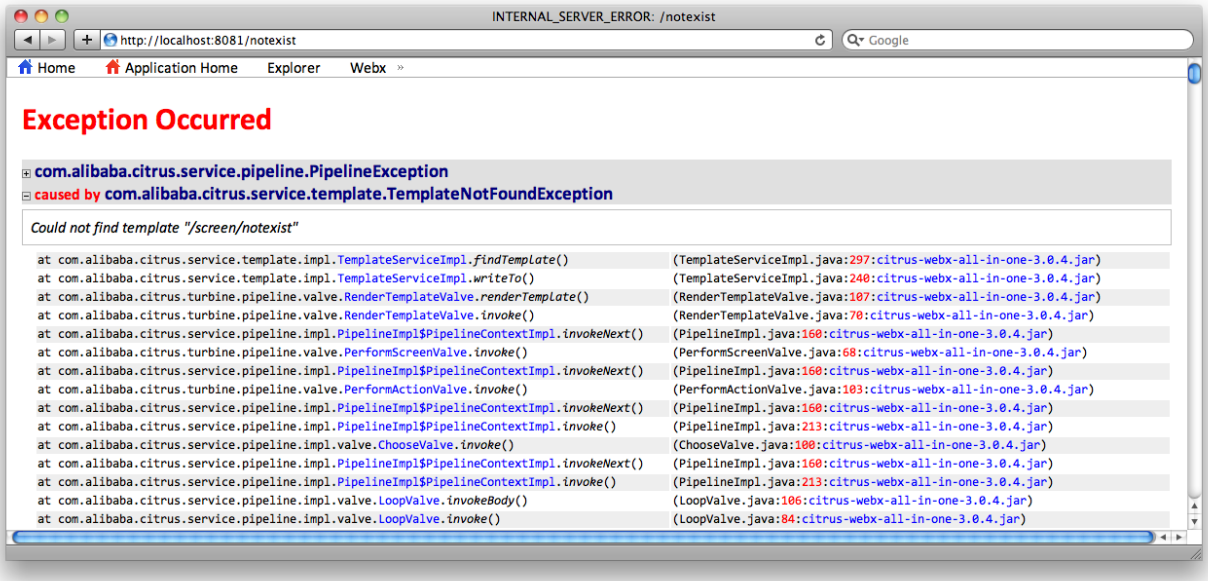


图 3.5. 开发模式的详细出错页面

- 开发模式下，可展示所有可用的schemas。

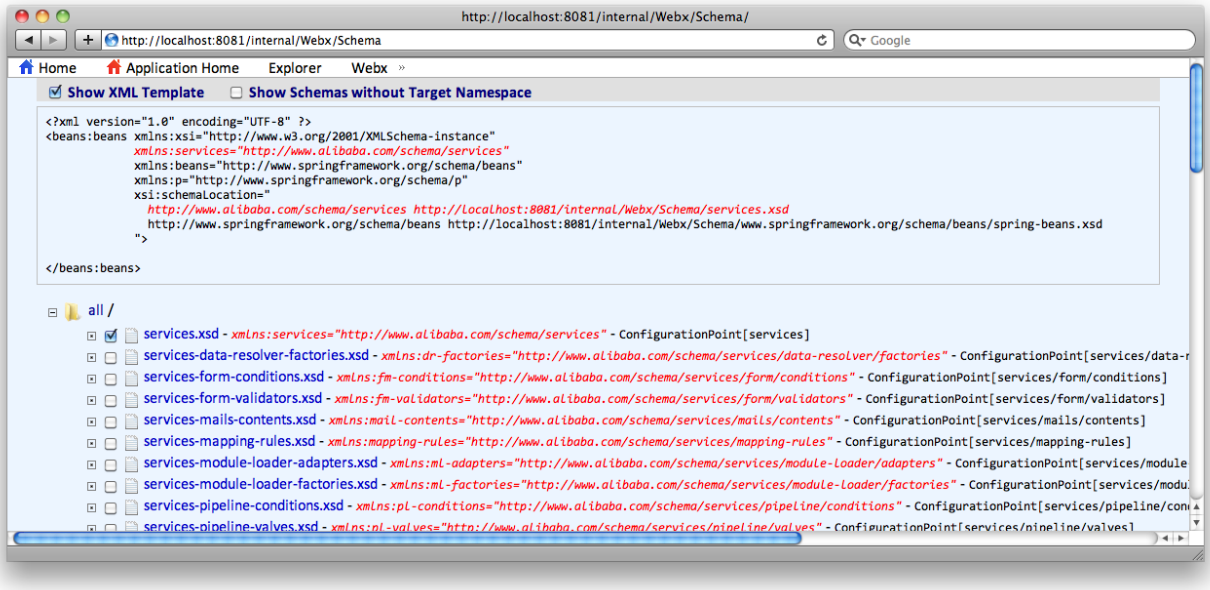


图 3.6. 开发模式下展示所有可用的schemas

- 开发模式下，可以查阅容器内部的信息。

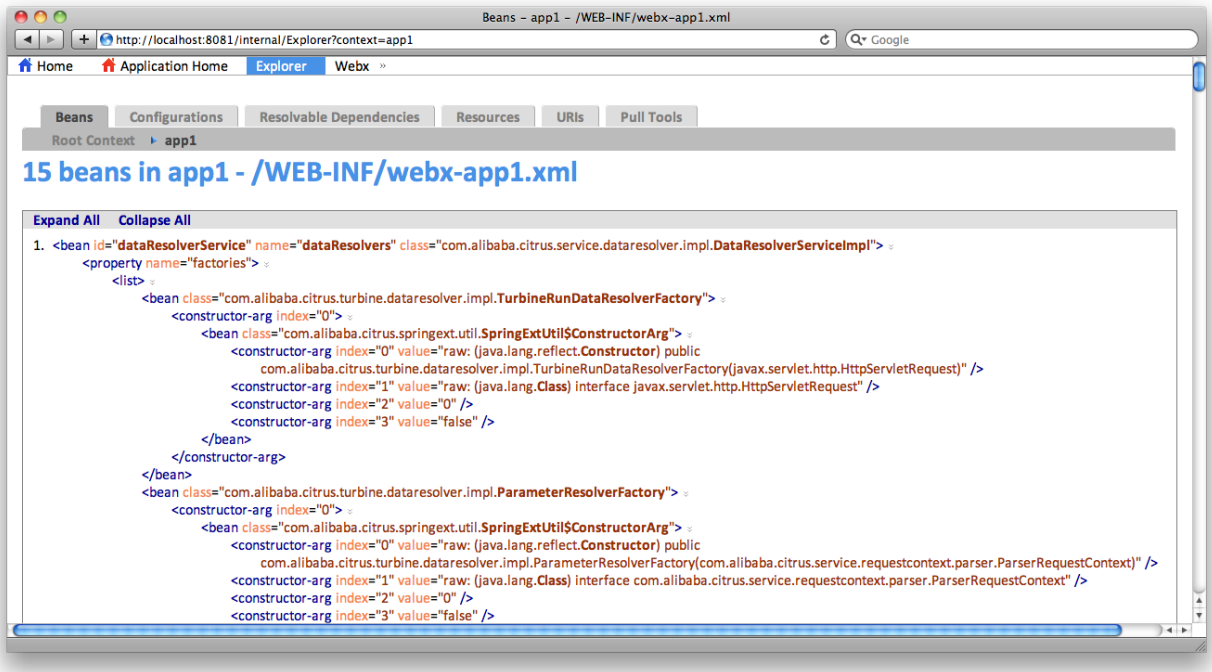


图 3.7. 开发模式下查阅容器内部的信息

可供查阅的信息包括：

表 3.5. 开发模式中可供查阅的容器信息

名称	说明
Beans	查看各Spring容器中的全部bean的定义。 这个工具有助于开发者理解用schema所定义的services和spring beans之间的联系。
Configurations	查看用来创建各Spring容器的配置文件。 这个工具会以树状和语法高亮显示配置文件以及所有被import的配置文件的内容。 不同于Beans工具，Configurations工具只忠实地展现配置文件的内容。而Beans工具展现的是真实的Beans结构。
Resolvable Dependencies	查看所有由框架置入到容器中的对象，例如：`HttpServletRequest`对象。这些对象不需要在配置文件中定义，就可被注入到应用中。
Resources	跟踪Resources的装载过程，显示Resources的树状结构。 这个工具有助于开发者理解`ResourceLoadingService`的工作原理。
URIs	查看所有的URI brokers。
Pull Tools	查看所有模板中可用的pull tools。

事实上，Webx Framework提供了一套专用的内部框架，使你可以往开发模式中添加更多的开发工具。例如，创建下面的功能并非难事：

- 查看session对象。
- 提供各种编码、解码的工具，以方便开发、调试应用。例如：将UTF-8编码的字符串转换成GBK编码；或者将字符串进行URL escape编码、解码等。

Webx Framework提供了一个接口：`ProductionModeAware`。Spring context中的beans，如果实现了这个接口，就可以感知当前系统的运行模式，从而根据不同的模式选择不同的行为——例如：在生产模式中打开cache，在开发模式中关闭cache。

例 3.8. 利用`ProductionModeAware`接口感知运行模式，并自动开关cache

```
public class ModuleLoaderServiceImpl implements ProductionModeAware {  ❶
    public void setProductionMode(boolean productionMode) { ❷
        this.productionMode = productionMode;
    }

    @Override
    protected void init() {
        ……
        if (cacheEnabled == null) {
            cacheEnabled = productionMode; ❸
        }
        ……
    }
}
```

❶❷ 实现`ProductionModeAware`接口。

❸ 根据当前运行模式自动开关cache。

3.2.5. 响应和处理请求的更多细节

当一个HTTP请求到达时，首先由WebxFrameworkFilter接手这个请求：

例 3.9. 配置**WebxFrameworkFilter** - **/WEB-INF/web.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/j2ee  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd
    ">
    ...
    <filter>
        <filter-name>webx</filter-name>
        <filter-class>com.alibaba.citrus.webx.servlet.WebxFrameworkFilter</filter-class> ❶
        <init-param>
            <param-name>excludes</param-name>
            <param-value><!-- 需要被“排除”的URL路径，以逗号分隔，前缀!表示“包含”。例如/static, *.jpg, !/
uploads/*.jpg --></param-value>  ❷
        </init-param>
        <init-param>
            <param-name>passthru</param-name>
            <param-value><!-- 需要被“略过”的URL路径，以逗号分隔，前缀!表示“不要略过”。例如/myservlet, *.jsp
 --></param-value>  ❸
        </init-param>
    </filter>

    <filter-mapping>
        <filter-name>webx</filter-name>
        <url-pattern>/*</url-pattern>  ❹
    </filter-mapping>
    ...
</web-app>
```

❶ 定义WebxFrameworkFilter。

❷ 可选的参数：“排除”指定名称的path，以逗号分隔，例如：/static, *.jpg。如果路径以!开始，表示“不排除”特殊目录。例如：*.jpg, !/uploads/*.jpg表示排除所有JPG图像文件，但不排除/uploads目录下的JPG图像文件。

❸ 可选的参数：“略过”指定名称的path，以逗号分隔，例如：/myservlet, *.jsp。和excludes参数一样，也支持!前缀，表示“不要略过”特殊目录。

❹ 匹配所有的path。

为什么使用filter而不是servlet呢？传统的WEB框架的控制器一般都是用servlet实现的。原因是：

- Filter可以“返还控制” —— 上面的配置文件直接把“/*”映射到webx filter中，这意味着webx接管了这个应用的所有请求。静态页面和资源怎么办？没关系，如果webx发现这个请求不应该由webx来处理，就会把控制“返还”给原来的控制器 —— 可能是另一个filter、servlet或者返回给servlet引擎，以默认的方式来处理。而Servlet是不具备“返还控制”的机制的。

- Servlet/Filter mapping的局限性 —— 标准的servlet引擎将URL映射到filter或servlet时，只支持前缀映射和后缀映射两种方式，非常局限。而实际情况往往复杂得多。Webx建议将所有请求都映射给webx来处理，让webx对请求做更灵活的映射。

如果你的web.xml中还有一些其它的servlet mappings，为了避免和Webx的URL起冲突，你可以把这些mapping加在excludes或passthru参数里。这样，WebxFrameworkFilter就会排除或略过指定的URL。例如：

```
        <init-param>
            <param-name>excludes</param-name>
            <param-value>/static, *.jpg, !/uploads/*.jpg</param-value>
        </init-param>
        <init-param>
            <param-name>passthru</param-name>
            <param-value>/myservlet, *.jsp</param-value>
        </init-param>
```

“passthru略过”和“excludes排除”的区别在于，如果一个servlet或filter接手被webx passthru的请求时，它们还是可以访问到webx的部分服务，包括：

- RequestContext服务，例如：解析参数、解析upload请求、重写请求、设置字符集编码和区域、基于cookie的session等。
- 开发模式及工具。
- 异常处理。
- 共享webx的spring容器。

也就是说，对于一个被passthru的请求，webx的行为更像是一个普通的filter。而“排除”则不同，如果一个请求被“排除”，webx将会立即放弃控制，将请求交还给服务器。接手控制的servlet或filter将无法访问webx一切的服务。

下图是WebxFrameworkFilter处理一个WEB请求的过程。

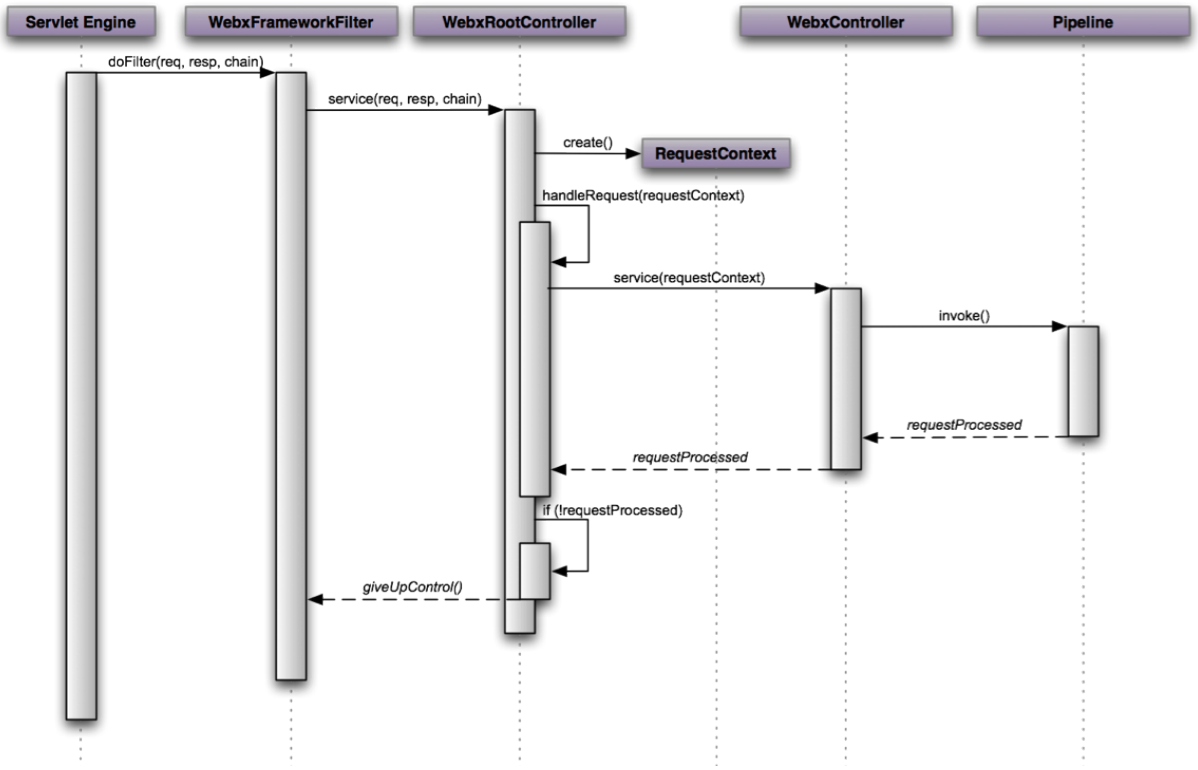


图 3.8. **WebxFrameworkFilter处理请求的详细过程**

如图所示，WebxFrameworkFilter接到请求以后，就会调用WebxRootController。从这里开始，进入Spring的世界 —— 此后所有的对

象：WebxRootController、WebxController、RequestContext、Pipeline等，全部是通过SpringExt配置在Spring Context中的。

WebxRootController对象存在于root context中，它被所有子应用所共享。它会创建RequestContext实例 —— 从而增强request、response、session的功能。接下来，WebxController对象会被调用。

WebxController对象是由每个子应用独享的，子应用app1和app2可以有不同的WebxController实现。默认的实现，会调用pipeline。

Pipeline也是由各子应用自己来配置的。假如pipeline碰到无法处理的请求，如静态页面、图片等，pipeline应当执行<exit/> valve强制退出。然后WebxRootController就会“放弃控制” —— 这意味着request将被返还给/WEB-INF/web.xml中定义的servlet、filter或者返还给servlet engine本身来处理。

3.3. 定制Webx Framework

3.3.1. 定制WebxRootController

WebxRootController是被所有子应用所共享的逻辑。 假如你想创建一种新的WEB框架，可以自己定义一个新的WebxRootController的实现。这个方案非常适合作为一个新Web框架的起点。

例 3.10. 自定义WebxRootController

```
<webx-configuration xmlns="http://www.alibaba.com/schema/services">
    <components>
        <rootController class="com.myframework.MyRootController" /> ❶
    </components>
</webx-configuration>
```

❶ 创建自己的WebxRootController。最简便的方法是：扩展AbstractWebxRootController，免去了创建Servlet/Filter、初始化Spring容器、处理request、response等繁杂事务，并且完全支持SpringExt的所有功能，此外还包含了错误处理、开发模式等Webx Framework中的一切便利。。

3.3.2. 定制WebxController

WebxController是用来控制子应用的。每个子应用可以拥有不同的WebxController实现。

Webx Framework默认的WebxController是调用pipeline。假如你不想用pipeline，而希望实现自己的针对子应用的逻辑，那么最简单的方法就是实现自己的WebxController或者扩展AbstractWebxController。

例 3.11. 自定义WebxController

```
<webx-configuration xmlns="http://www.alibaba.com/schema/services">
    <components defaultControllerClass="com.myframework.MyController"> ❶
        <component name="app1">
            <controller class="com.myframework.MyController" /> ❷
        </component>
    </components>
</webx-configuration>
```

❶ 指定默认的WebxController实现类。

❷ 对特定子应用明确指定`WebxController`实现类。

3.4. 本章总结

Webx Framework提供了一个可剪裁、可扩展的处理WEB请求基本框架。它所提供的基本功能事实上是每个WEB框架都需要用到的。Webx Framework为进一步实现WEB框架提供了坚实的基础。

第 4 章 Webx Turbine

Webx是一套基于Java Servlet API的通用Web框架。整个Webx框架分成三个层次，本章将简单介绍其第三个层次：Webx Turbine。Webx Turbine建立在Webx Framework的基础上，实现了页面渲染、布局、数据验证、数据提交等一系列工作。

Webx Turbine之所以叫这个名字，是因为Webx最早的版本，是从Apache Turbine项目上发展而来的。到现在，Turbine的代码已经荡然无存，然而Turbine中的一些风格和想法依赖保存在Webx框架中。

4.1. 设计理念

Webx Turbine所遵循下面的设计理念包括：

- 页面驱动
- 约定胜于配置

4.1.1. 页面驱动

创建一个WEB应用，一般会经历三个阶段：产品设计、用户界面设计、功能实现。分别由产品设计师、用户界面设计师和程序员协作完成。如下图所示。

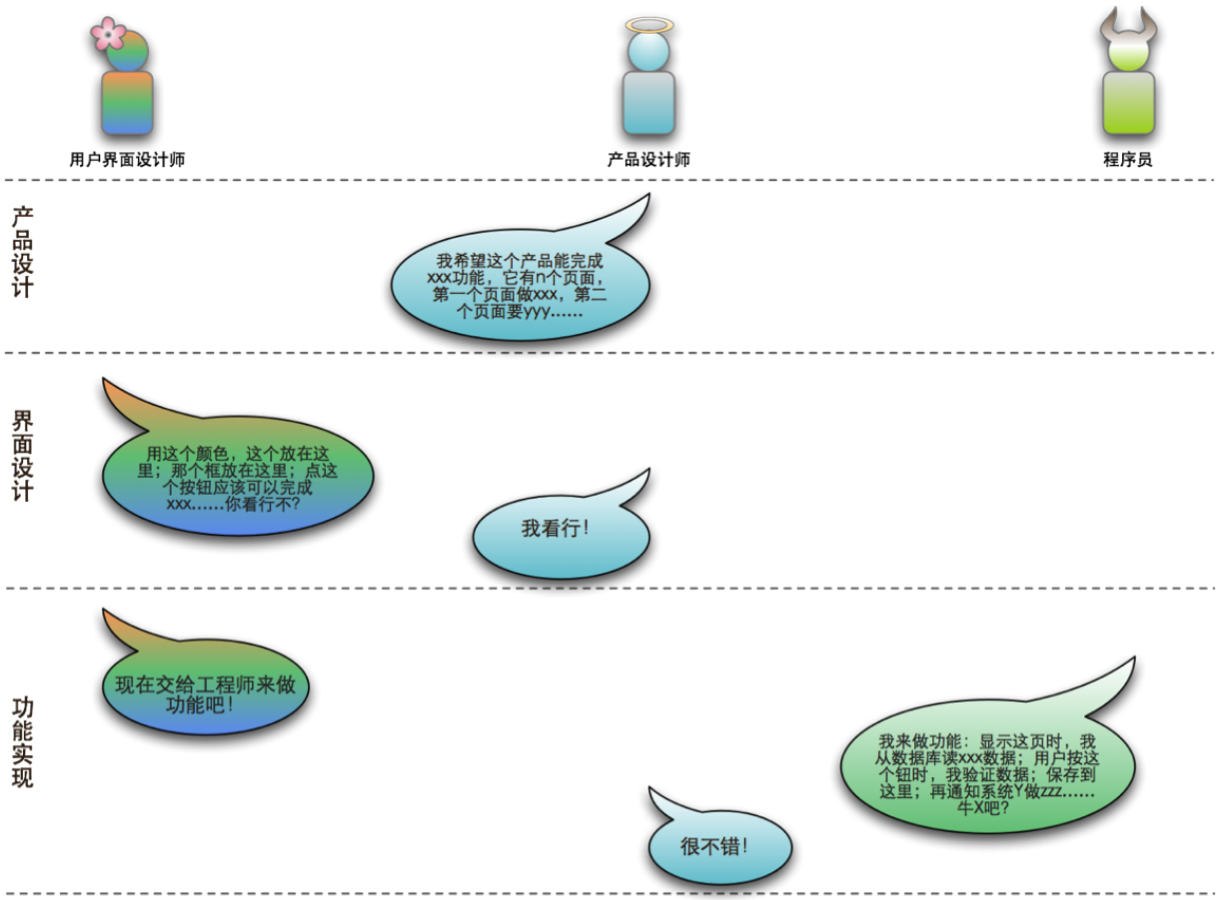


图 4.1. 协作图：创建一个WEB应用

通常，界面设计师只完成纯静态页面的设计，需要由程序员来把静态页面转换、分解成模板，才能在最终的WEB应用中被使用。为什么不让界面设计师直接创建模板呢？这样一定可以提高很多效率。然而在一般的WEB框架中，由于模板不能独立于程序元素（如action）而存在，因此在程序员介入以前，界面设计师是没有办法展示模板的效果的。

Webx Turbine推崇页面驱动的理念。它的意思是，在程序员介入以前，让界面设计师可以直接创建模板，并展示模板的效果。页面驱动的反面，是程序驱动，或者是Action驱动 —— 这是多数WEB框架的模式。

页面驱动不止提高了开发的效率，也使界面设计师在早期阶段，就可以利用框架所提供的工具，做一些以前做不到的事，例如：页面跳转、简单的表单验证、字符串操作等。这些工具是通过Webx Turbine中的一个服务来完成的：pull tools。Pull tools服务预先准备了很多模板中可用的工具，让模板可以“按需”取得这些对象 —— 这就是pull这个单词的意思。

4.1.2. 约定胜于配置

Webx Turbine的另一个理念，是约定胜于配置。“约定”即规则。规则是预先定义的，工程师只需要按着规则来做事，就不需要额外的“配置”。对比其它一些框架 —— 往往每增加一个页面，都需要在配置文件中增加若干行内容。

Webx Turbine的规则主要是指一系列映射规则。

表 4.1. Webx Turbine映射规则

映射规则	说明
将URL映射成target	target是一个抽象的概念，指明当前请求要完成的任务。Target由pipeline来解释，它可能被解释成模板名，也可能被解释成别的东西。
将target转换成模板名	模板用来展现页面的内容。Velocity、Freemarker、JSP都可以作为模板的格式，但在Webx建议使用velocity模板。
将target转换成layout布局	你可以为一组页面选择相同的布局（菜单、导航栏、版权信息等），为另一组页面选择另一种布局。
将target转换成module	在Webx Turbine中，module是指screen、action、control等，大致相当于其它框架中的action或者controller。

工程师只需要根据上述规则，将模板放在指定的目录、按照预定的方式命名module（也就是screen、action、control等），就不再需要额外的配置。

4.2. 页面布局

Webx Turbine的页面，由以下几个部分组成：

图 4.2. Webx Turbine页面的构成

其中：

- Screen，代表页面的主体。
- Layout，代表页面的布局。
- Control，代表嵌在screen和layout中的页面片段。

4.3. 处理页面的基本流程

Webx Turbine的处理流程被定义在pipeline中。Webx Framework没有规定Pipeline的内容，但Webx Turbine却定义了一系列valves。下面是一个Webx Turbine推荐的pipeline配置：

例 4.1. Webx Turbine推荐的pipeline配置 - **pipeline.xml**

```
<services:pipeline xmlns="http://www.alibaba.com/schema/services/pipeline/valves">

    <!-- 初始化turbine rundata，并在pipelineContext中设置可能会用到的对象(如rundata、utils)，以便valve取得。 -->
    <prepareForTurbine />

    <!-- 设置日志系统的上下文，支持把当前请求的详情打印在日志中。 -->
    <setLoggingContext />

    <!-- 分析URL，取得target。 -->
    <analyzeURL homepage="homepage" />

    <!-- 检查csrf token，防止csrf攻击和重复提交。假如request和session中的token不匹配，则出错，或显示expired页面。 -->
    <checkCsrfToken />

    <loop>
        <choose>
            <when>
                <!-- 执行带模板的screen，默认有layout。 -->
                <pl-conditions:target-extension-condition extension="null, vm, jsp" />
                <performAction />
                <performTemplateScreen />
                <renderTemplate />
            </when>
            <when>
                <!-- 执行不带模板的screen，默认无layout。 -->
                <pl-conditions:target-extension-condition extension="do" />
                <performAction />
                <performScreen />
            </when>
            <otherwise>
                <!-- 将控制交还给servlet engine。 -->
                <exit />
            </otherwise>
        </choose>

        <!-- 假如rundata.setRedirectTarget()被设置，则循环，否则退出循环。 -->
        <breakUnlessTargetRedirected />
    </loop>

</services:pipeline>
```

假设用户以URL：`http://localhost:8081/`来访问Webx应用。域名和端口不重要，取决于应用服务器的配置，这里假设为`localhost:8081`。Webx Framework的处理流程，从`WebxFrameworkFilter`接收请求，并且一路顺利到达pipeline。然后Pipeline开始依次执行它的valves。（下面的描述略过一些相对次要的步骤。）

1. **<analyzeURL> - 分析URL**

   分析URL的目的是取得target。由于用户访问的URL中并没有提供path信息，通常被理解为：用户想要访问“主页”。AnalyzeURL valve提供了一个可选的参数“homepage”，即是在这种情况下起作用 —— http://localhost:8081/对应的target为“homepage”。

   需要注意的是，*target不代表模板名，也不代表类名*。Target只是一个抽象的概念 —— 当前页面需要达成的目标。Target可能被后续的valves*解释成*模板名、类名或者其它东西。

2. **进入<choose> - 多重分支**

   很明显，“homepage”满足了第一个<when>所附带的条件：<target-extension-condition extension="null, vm, jsp">，意思是target的后缀不存在（null）或为“jsp”或为“vm”。

3. **<performAction> - 执行action**

   和其它框架中的action概念不同，在Webx Turbine中，action是用来处理用户提交的表单的。

   因为本次请求未提供action参数，所以跳过该步骤。

4. **<performTemplateScreen> - 查找并执行screen。**

   这里要用到一个规则：target映射成screen module类名的规则。

   假设target为xxx/yyy/zzz，那么Webx Turbine会依次查找下面的screen模块：

   - screen.xxx.yyy.Zzz，
   - screen.xxx.yyy.Default，
   - screen.xxx.Default，
   - screen.Default。

   本次请求的target为homepage，因此它会尝试查找screen.Homepage和screen.Default这两个类。

   如果找到screen类，Webx Turbine就会执行它。Screen类的功能，通常是读取数据库，然后把模板所需要的对象放到context中。

   如果找不到，也没关系 —— 这就是“页面优先”：像homepage这样的主页，通常没有业务逻辑，因此不需要screen类，只需要有模板就可以了。

5. **<renderTemplate> - 渲染模板**

   这里用到两个规则：target映射成screen template，以及target映射成layout template。

   假设target为xxx/yyy/zzz，那么Webx Turbine会查找下面的screen模板：/templates/screen/xxx/yyy/zzz。Screen模板如果未找到，就会报404 Not Found错误。找到screen模板以后，Webx Turbine还会试着查找下面的layout模板：

   - /templates/layout/xxx/yyy/zzz

- `/templates/layout/xxx/yyy/default`
- `/templates/layout/xxx/default`
- `/templates/layout/default`

Layout模板如果找不到，就直接渲染screen模板；如果存在，则把渲染screen模板后的结果，嵌入到layout模板中。

Layout模板和screen模板中，都可以调用control。每个页面只有一个screen，却可以有任意多个controls。

6. **`<breakUnlessTargetRedirected>` - 内部重定向**

   在screen和action中，可以进行“内部重定向”。内部重定向实质上就是由`<breakUnlessTargetRedirected>`实施的 —— 如果没有重定向标记，就退出；否则循环到`<loop>`标签。

   和外部重定向不同，外部重定向是向浏览器返回一个`302`或`303` response，其中包含`Location` header，浏览器看到这样的response以后，就会发出第二个请求。而内部重定向发生在pipeline内部，浏览器并不了解内部重定向。

4.4. 依赖注入

4.4.1. Spring原生注入手段

依赖注入是Spring的重要特性，Webx既然建立在Spring基础上，当然支持Spring原有的依赖注入手段，例如，你可以在Screen/control/action module类中这样写：

例 4.2. **通过`@Autowired`** annotation**注入**

```
public class LoginAction {
    @Autowired
    private UserManager userManager; ❶
    ...
}
```

❶ `UserManager`是在spring context中配置的bean。

在使用Spring原生注入手段时，需要注意beans的scope。你只能注入相同scope或较大的scope中的bean。例如，screen/action/control的scope为singleton，因此用`@Autowired`注入时，只能注入singleton的对象，不能注入诸如request、session等较小的scope对象。

4.4.2. 注入request、response和session对象

在Webx Framework中，你可以这样做：

例 4.3. 注入request、response和session对象

```
public class LoginAction {
    @Autowired
    private HttpServletRequest request;

    @Autowired
    private HttpServletResponse response;

    @Autowired
    private HttpSession session;
    ...
}
```

前面我们刚讲过，你*不能把request scope的对象，注入到singleton scope的对象中*。但在Webx中，你*可以将HttpServletRequest、HttpServletResponse和HttpSession对象注入到singleton对象中*。为什么呢？原来，<request-contexts>对这几个常用对象进行了特殊处理，将它们转化成了singleton对象。

4.4.3. 参数注入

有一些对象，是无法通过Spring的bean来注入的，例如：用户提交的参数、表单等。好在Webx Turbine提供了一种可扩展的机制（DataResolver service），通过它，我们可以在screen/control/action的方法中注入任意对象。

表 4.2. 参数注入

功能	代码示例	适用于module类型
注入一个query参数	`void doGetInt(@Param("aaa") int i)`	screen、action、control
将query参数注入bean properties	`void doSetData(@Params MyData data)`	screen、action、control
注入框架对象	`void doGetNavigator(Navigator nav)`	screen、action
	`void doGetContext(Context context)`	screen、action、control
	`void execute(ControlParameters params)`	control
注入context和control参数	`void execute(@ContextValue("myvalue") int value)`	screen、action、control
注入表单对象	`void doGetGroup(@FormGroup("myGroup1") Group group)`	action
	`void doGetGroups(@FormGroups("myGroup1") Group[] groups)`	action
将表单值注入bean properties	`void doGetGroupsBeans(@FormGroups("myGroup1") MyData[] data)`	action

4.5. 定制Webx Turbine

通过改进pipeline中的valves，我们很容易改变webx turbine的行为。

最常见的一种需求，是要对页面进行授权 —— 只有符合条件的用户才能访问相应的页面。在pipeline中，很容易添加这样的逻辑：

例 4.4. 改进pipeline，增加页面授权功能 - **pipeline.xml**

```
<services:pipeline xmlns="http://www.alibaba.com/schema/services/pipeline/valves">
    <prepareForTurbine />
    <setLoggingContext />
    <analyzeURL homepage="homepage" />
    <checkCsrfToken />
    <valve class="com.mycompany.auth.PageAuthorizationValve" /> ❶
    ...
</services:pipeline>
```

❶ 插入用于验证权限的valve。

事实上，你甚至可以重写整个pipeline，以实现另一种风格的WEB框架。

4.6. 本章总结

Webx Turbine建立在pipeline的基础上，基于页面驱动和约定胜于配置的理念，定义了一组处理页面的流程。Webx Turbine的灵活性在于，你可以轻易定制pipeline，以改变它的任何一个方面。

部分 II. Webx基础设施服务

第 5 章 Resource Loading服务指南

Webx框架中，包含了一套用来查找和装载资源的服务 —— Resource Loading服务。

Resource Loading服务从Spring ResourceLoader机制中扩展而来，并且和Spring框架融为一体。因此，你不需要写特别的Java代码，就可以让所有利用Spring ResourceLoader机制的代码，直接享用Webx所提供的新的Resource Loading机制。

5.1. 资源概述

5.1.1. 什么是资源?

在一个稍具规模的应用程序中，经常要做的一件事，就是查找资源、读取资源的内容。这里所谓的“资源”，是指*存放在某一介质中，可以被程序利用的文件、数据*。例如，基于Java的WEB应用中，常用到下面的资源：

- 配置文件：*.xml、*.properties等。
- Java类文件：*.class。
- JSP页面、Velocity模板文件：*.jsp、*.vm等。
- 图片、CSS、JavaScript文件：*.jpg、*.css、*.js等。

5.1.2. 如何表示资源?

在Java中，有多种形式可以表示一个资源：

表 5.1. 资源的表示

可表示资源的对象	说明
java.io.File	可代表文件系统中的文件或目录。例如： • 文件系统中的文件：“c:\config.sys”。 • 文件系统中的目录：“c:\windows\”。
java.net.URL	统一资源定位符。例如： • 文件系统中的文件：c:\config.sys，可以表示成URL：“file:///c:/config.sys”。 • 文件系统中的目录：c:\windows\，可以表示成URL：“file:///c:/windows/”。 • 远程WEB服务器上的文件：“http://www.springframework.org/schema/beans.xml”。 • Jar包中的某个文件，可以表示成URL：“jar:file:///c:/my.jar!/my/file.txt”。

java.io.InputStream	输入流对象，可用来直接访问资源的内容。例如：
	• 文件系统中的文件：c:\config.sys，可以用下面的代码来转换成输入流：
	`new FileInputStream("c:\\config.sys");`
	• 远程WEB服务器上的文件，可以用下面的代码来转换成输入流：
	`new URL("http://www.springframework.org/schema/beans.xml").openStream();`
	• Jar包中的某个文件，可以用下面的代码来转换成输入流：
	`new URL("jar:file:///c:/my.jar!/my/file.txt").openStream();`

然而，并不是所有的资源，都可以表现成上述所有的形式。比如，

- Windows文件系统中的目录，无法表现为输入流。
- 而远程WEB服务器上的文件无法转换成File对象。
- 多数资源都可以表现成URL形式。但也有例外，例如，如果把数据库中的数据看作资源，那么一般来说这种资源无法表示成URL。

5.1.3. 如何访问资源?

不同类型的资源，需要用不同的方法来访问。

访问CLASSPATH中的资源

将资源放在CLASSPATH是最简单的做法。我们只要把所需的资源文件打包到Jar文件中，或是在运行java时，用-classpath参数中指定的路径中。接下来我们就可以用下面的代码来访问这些资源：

例 5.1. 访问CLASSPATH中的资源

```
URL resourceURL = getClassLoader().getResource("java/lang/String.class"); // 取得URL
InputStream resourceContent = getClassLoader().getResourceAsStream("java/lang/String.class"); // 取得输入流
```

访问文件系统中的资源

下面的代码从文件资源中读取信息：

例 5.2. 访问文件系统中的资源

```
File resourceFile = new File("c:\\test.txt"); // 取得File
InputStream resourceContent = new FileInputStream(resourceFile); // 取得输入流
```

访问Web应用中的资源

Web应用既可以打包成war文件，也可以展开到任意目录中。因此Web应用中的资源（JSP、模板、图片、Java类、配置文件）不总是可以用文件的方式存取。虽然Servlet API提供了`ServletContext.getRealPath()`方法，用来取得某个资源的实际文件路径，但该方法很可能返回`null` —— 这取决于应用服务器的实现，以及Web应用的布署方式。更好的获取WEB应用资源的方法如下：

例 5.3. 访问Web应用中的资源

```
URL resourceURL = servletContext.getResource("/WEB-INF/web.xml"); // 取得URL
InputStream resourceContent = servletContext.getResourceAsStream("/WEB-INF/web.xml"); // 取得输入流
```

访问Jar/Zip文件中的资源

下面的代码读取被打包在Jar文件中的资源信息：

例 5.4. 访问Jar/Zip文件中的资源

```
URL jarURL = new File(System.getProperty("java.home") + "/lib/rt.jar").toURI().toURL();
URL resourceURL = new URL("jar:" + jarURL + "!/java/lang/String.class"); // 取得URL
InputStream resourceContent = resourceURL.openStream(); // 取得输入流
```

访问其它资源

还可以想到一些访问资源的方法，例如从数据库中取得资源数据。

5.1.4. 如何遍历资源?

有时候，我们不知道资源的路径，但希望能找出所有符合条件的资源，这个操作叫作遍历。例如，找出所有符合pattern “`/WEB-INF/webx-*.xml`”的配置文件。

遍历文件系统

例 5.5. 遍历文件系统

```
File parentResource = new File("c:\\windows");
File[] subResources = parentResource.listFiles();
```

遍历WEB应用中的资源

例 5.6. 遍历WEB应用中的资源

```
Set<String> subResources = servletContext.getResourcePaths("/WEB-INF/");
```

遍历Jar/zip文件中的资源

例 5.7. 遍历Jar/zip文件中的资源

```
File jar = new File("myfile.jar");
ZipInputStream zis = new ZipInputStream(new FileInputStream(jar));

try {
    for (ZipEntry entry = zis.getNextEntry(); entry != null; entry = zis.getNextEntry()) {
        // visit entry
    }
} finally {
    zis.close();
}
```

并非所有类型的资源都支持遍历操作。通常遍历操作会涉及比较复杂的递归算法。

5.1.5. 有什么问题?

应用程序访问资源时，有什么问题呢?

首先，资源表现形式的多样性，给应用程序的接口设计带来一点麻烦。假如，我写一个`ConfigReader`类，用来读各种配置文件。那么我可能需要在接口中列出所有的资源的形式:

例 5.8. 用来读取配置文件的接口

```
public interface ConfigReader {
    Object readConfig(File configFile);
    Object readConfig(URL configURL);
    Object readConfig(InputStream configStream);
}
```

特别是当一个通用的框架，如Spring和Webx，需要在对象之间传递各种形式的资源的时候，这种多样性将导致很大的编程困难。

其次，有这么多种查找资源和遍历资源的方法，使我们的应用程序和资源所在的环境高度耦合。这种耦合会妨碍代码的可移植性和可测试性。

比如，我希望在非WEB的环境下测试一个模块，但这个模块因为要存取Web应用下的资源，而引用了`ServletContext`对象。在测试环境中并不存在`ServletContext`而导致该模块难以被测试。再比如，我希望测试的一个模块，引用了classpath下的某个配置文件（这也是一种耦合）。而我希望用另一个专为该测试打造的配置文件来代替这个文件。由于原配置文件是在classpath中，因此是难以替换的。

对于不打算重用的应用程序来说，这个问题还不太严重：大不了我预先设定好，就从这个地方，以固定的方式存取资源。然而就算这样，也是挺麻烦的。有的人喜欢把资源放在某个子目录下，有的人喜欢把资源放在CLASSPATH下，又有人总是通过`ServletContext`来存取Web应用下的资源。当你要把这些不同人写的模块整合起来时，你会发现很难管理。

一种可能发生的情形是，因为某些原因，环境发生改变，导致资源的位置、存取方式不得不跟着改变。比如将老系统升级为新系统。但一些不得不继续使用的老代码，由于引用了旧环境的资源而不能工作 —— 除非你去修改这些代码。有时修改老代码是很危险的，可能导致不可预知的错误。又比如，由于存储硬件的改变或管理的需要，我们需要将部分资源移到另一个地方（我们曾经将Web页面模板中的某个子目录，移动到一个新的地方，因为这些模板必须由新的CMS系统自动生成）。想要不影响现有代码来完成这些事，是很困难的。

5.2. Spring的ResourceLoader机制

Spring内置了一套ResourceLoader机制，很好地解决了访问资源的大部分问题。

5.2.1. Resource接口

Spring将所有形式的资源表现概括成一个Resource接口。如下所示（下面的接口定义是被简化的，有意省略了一些东西，以便突出重点）：

例 5.9. Spring的Resource接口（简化）

```
public interface Resource {
    InputStream getInputStream();
    URL getURL();
    File getFile();
    boolean exists();
}
```

Resource接口向应用程序屏蔽了资源表现形式的多样性。于是，前面例子中的ConfigReader就可以被简化成下面的样子：

例 5.10. 用来读取配置文件的接口（简化后）

```
public interface ConfigReader {
    Object readConfig(Resource configResource);
}
```

事实上，Spring正是利用Resource接口来初始化它的ApplicationContext的：

例 5.11. Spring用Resource接口来代表用来初始化ApplicationContext的配置文件

```
public abstract class AbstractXmlApplicationContext extends ... {
    ...
    protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) {
        Resource[] configResources = getConfigResources();
        ...
    }

    protected Resource[] getConfigResources();
}
```

5.2.2. ResourceLoader和ResourcePatternResolver接口

Spring不仅可以通过ResourceLoader接口来取得单一的资源对象，还可以通过ResourcePatternResolver遍历并取得多个符合指定pattern的资源对象。这个设计向应用程序屏蔽了查找和遍历资源的复杂性。

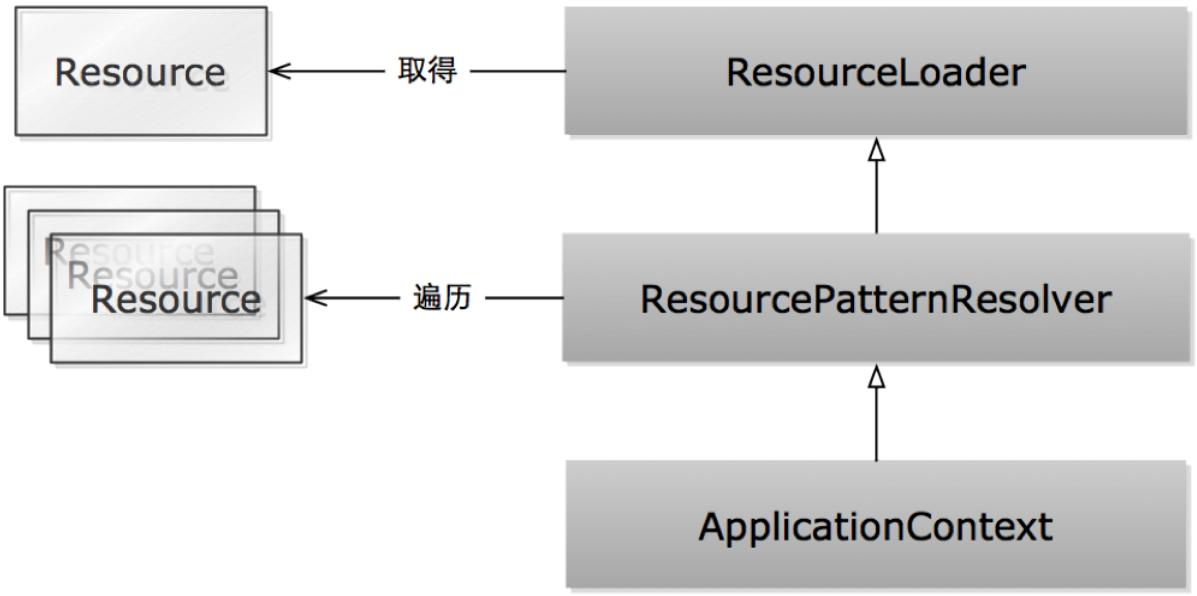


图 5.1. **ResourceLoader和ResourcePatternResolver接口**

5.2.3. 在代码中取得资源

5.2.3.1. 通过ResourceLoader取得资源

例 5.12. **通过ResourceLoader取得资源**

```
public class MyBean implements ResourceLoaderAware ❶ {
    private ResourceLoader loader;

    public void setResourceLoader(ResourceLoader loader) ❷ {
        this.loader = loader;
    }

    public void func() {
        Resource resource = loader.getResource("myFile.xml"); ❸
        ...
    }
}
```

❶❷ 实现了ResourceLoaderAware接口。要取得资源，必须要拿到ResourceLoader对象。而通过ResourceLoaderAware接口拿到ResourceLoader是最简单的方法。

❸ 调用所取得的ResourceLoader来取得资源。

5.2.3.2. 直接注入资源

另一种更简便的方法是，将资源直接“注入”到bean中 —— 你不需要手工调用ResourceLoader来取得资源的方式来设置资源。例如：

例 5.13. 直接注入资源

```
public class MyBean {
    private URL resource;

    public void setLocation(URL resource) ❶ {
        this.resource = resource;
    }

    ……
}
```

Spring配置文件可以这样写：

```
<bean id="myBean" class="MyBean">
    <property name="location" value="myFile.xml" />
</bean>
```

❶ 此处注入资源的URL

这样，Spring就会把适当的`myFile.xml`所对应的资源注入到`myBean`对象中。此外，Spring会自动把`Resource`对象转换成`URL`、`File`等普通对象。在上面的例子中，`MyBean`并不依赖于`Resource`接口，只依赖于`URL`类。

将代码稍作修改，就可以注入一组资源：

例 5.14. 注入一组资源

```
    public void setLocations(URL[] resources) ❶ {
        this.resources = resources;
    }
```

配置文件：

```
    <property name="locations" value="WEB-INF/webx-*.xml" />
```

❶ 此处注入资源的URL的数组。

上例中，可以直接得到所有符合pattern “`WEB-INF/webx-*.xml`”的配置文件。显然这是通过`ResourcePatternResolver`取得的。

5.2.4. Spring如何装载资源?

Spring是如何装载资源文件的呢？Spring装载资源的方案是由`ApplicationContext`决定的。不同的`ApplicationContext`类，实现了不同的资源装载方案。

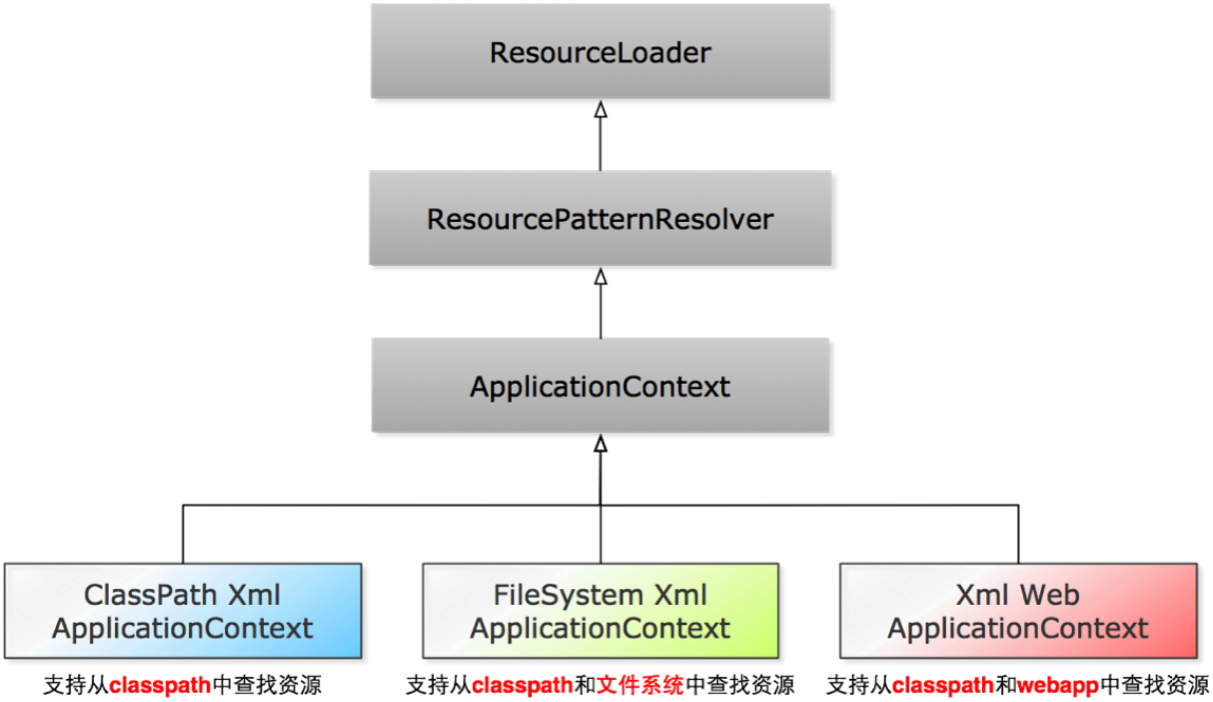


图 5.2. Spring **ApplicationContext实现了资源装载的具体方案**

5.2.4.1. `ClassPathXmlApplicationContext`

`ClassPathXmlApplicationContext`支持从classpath中装载资源。

例 5.15. **`ClassPathXmlApplicationContext` - 从classpath中装载资源**

假如我以下面的方式启动Spring，那么系统将支持从*classpath中*装载资源。

```
ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
```

`ClassPathXmlApplicationContext`装载资源文件`myFile.xml`的逻辑，相当于如下代码：

```
URL resource = getClassLoader().getResource("myFile.xml");
```

5.2.4.2. `FileSystemXmlApplicationContext`

`FileSystemXmlApplicationContext`支持从文件系统中装载资源。

例 5.16. **`FileSystemXmlApplicationContext` - 从文件系统中装载资源**

假如我以下面的方式启动Spring，那么系统将支持从*文件系统中*装载资源。

```
ApplicationContext context = new FileSystemXmlApplicationContext("beans.xml");
```

`FileSystemXmlApplicationContext`装载资源文件`myFile.xml`的逻辑，相当于如下代码：

```
File resource = new File("myFile.xml");
```

5.2.4.3. XmlWebApplicationContext

XmlWebApplicationContext支持从webapp上下文中（也就是ServletContext对象中）装载资源。

例 5.17. XmlWebApplicationContext - 从Web应用的根目录中装载资源

假如我以下面的方式启动Spring，那么系统将支持从*Web应用的根目录中*装载资源。

```
ApplicationContext context = new XmlWebApplicationContext();

context.setConfigLocation("/WEB-INF/beans.xml");
context.setServletContext(servletContext);
context.refresh();
```

也可以让ContextLoaderListener来创建XmlWebApplicationContext，只需要在/WEB-INF/web.xml中添加如下配置：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/beans.xml</param-value>
</context-param>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

XmlWebApplicationContext装载资源文件myFile.xml的逻辑，相当于如下代码：

```
URL resource = servletContext.getResource("myFile.xml");
```

5.2.4.4. Classpath和Classpath*前缀

除了用ClassPathXmlApplicationContext以外，事实上所有的Spring ApplicationContext实现也都支持装载classpath中的资源。可以用下面两种方法：

表 5.2. Spring ApplicationContext装载classpath资源的方法

方法	说明
使用classpath:前缀	例如：“classpath:myFile.xml” —— 在classpath中装载资源myFile.xml。
使用classpath*:前缀	例如：“classpath*:/META-INF/my*.xml” —— 在classpath中装载所有符合pattern的资源。

5.2.5. Spring ResourceLoader的缺点

鱼和熊掌不可得兼

Spring ResourceLoader是由ApplicationContext来实现的。而你一次只能选择一种ApplicationContext的实现 —— 如果你选择了XmlWebApplicationContext，你就放弃了FileSystemXmlApplicationContext；反之亦然。

在WEB应用中，由于Spring使用了XmlWebApplicationContext，因此你就无法装载文件系统下的资源。

不透明性

你必须用“绝对路径”来引用Spring中的资源。

假如你使用`FileSystemXmlApplicationContext`来访问资源，你必须使用绝对路径来访问文件或目录资源。这妨碍了应用程序在不同系统中布署的自由。因为在不同的系统中，例如Windows和Linux，文件的绝对路径是不同的。为了系统管理的需要，有时也需要将文件或目录放在不同于开发环境的地方。

即便是访问WEB应用下的资源，或者是classpath下的资源，你也必须明确指出它们的位置，例如：`WEB-INF/myFile.xml`、`classpath:myFile.xml`等。如果我希望把`classpath:myFile.xml`挪到另一个物理位置，就必须修改所有的引用。

无扩展性

我无法在Spring `ResourceLoader`机制中增加一种新的装载资源的方法。例如，我希望把资源文件保存在数据库中，并用`ResourceLoader`来取得它。用Spring很难做到这点。

5.3. Resource Loading服务

5.3.1. 替换Spring `ResourceLoader`

Webx Resource Loading服务可作为Spring `ResourceLoader`机制的替代品（Drop-in Replacement，投入既替换）：

- 当你不使用它时，Spring原有的`ResourceLoader`功能不受影响；
- 当你在spring配置文件中添加Resource Loading服务时，`ResourceLoader`即被切换到新的机制。新的机制可兼容原有的Spring配置和代码，但支持更多的资源装载方式，以及更多的功能，如资源重命名、资源重定向等。

你只需要在配置文件中增加以下内容，就可以将Spring `ResourceLoader`机制替换成Webx的Resource Loading服务：

例 5.18. Resource Loading服务的基本配置（`/WEB-INF/webx.xml`）

```
<resource-loading
        xmlns="http://www.alibaba.com/schema/services"
        xmlns:res-loaders="http://www.alibaba.com/schema/services/resource-loading/loaders">

    <resource-alias pattern="/" name="/webroot" /> ❶

    <resource pattern="/webroot" internal="true"> ❷
        <res-loaders:webapp-loader /> ❸
    </resource>
    <resource pattern="/classpath" internal="true"> ❹
        <res-loaders:classpath-loader /> ❺
    </resource>

</ resource-loading>
```

关于这段配置的具体含义，请参见本章其它小节：

❶ 请参见：第 5.3.3 节 “重命名资源”。

❷❹ 请参见：第 5.3.2 节 “定义新资源”。

❸ 请参见：第 5.4.2 节 “`WebappResourceLoader`”。

❺ 请参见：第 5.4.3 节 “`ClasspathResourceLoader`”。

这段配置使得Resource Loading服务的行为和原来的Spring `ResourceLoader`完全兼容：

- 仍然支持`classpath:`和`classpath*:`前缀所定义的资源。
- 如不加前缀，则代表访问WEB应用根目录下的文件。例如：
  - `/myFile.xml`代表着Web应用根目录下的`/myFile.xml`。
  - `/WEB-INF/myFile.xml`代表着Web应用根目录下的`/WEB-INF/myFile.xml`。

加上这段配置以后，虽然功能和原来相比并没有变化，然而它已经准备好向系统中添加新的资源装载的功能了。

5.3.2. 定义新资源

定义一种新资源，需要回答两个问题：

1. 资源的名称是什么？

2. 资源在哪里（或如何装载资源）？

下面的例子定义了一种新的资源，它的名称是“`/jdk/*`”，通过“`file-loader`”从文件系统`${java.home}`文件夹中装载。

例 5.19. 定义新资源：`/jdk/*`

```
<resource-loading
        xmlns="http://www.alibaba.com/schema/services"
        xmlns:res-loaders="http://www.alibaba.com/schema/services/resource-loading/loaders">
    ...

    <resource pattern="/jdk">  ❶
        <res-loaders:file-loader basedir="${java.home}" />  ❷
    </resource>

</resource-loading>
```

❶ 定义新资源，资源名以`/jdk`为前缀。

❷ `<file-loader>`表示从文件系统中装载资源。详见：第 5.4.1 节 “FileResourceLoader”。

`${java.home}`是Java提供的system property，它的值指向当前Java运行环境的根目录。

前文讲过，Spring可以直接把资源注入到对象中。使用Resource Loading服务以后，你仍然可以这样做。下面的配置把JDK目录下的`tools.jar`文件（如果存在的话）的URL注入到`myBean`中：

例 5.20. 注入`JAVA_HOME/lib/tools.jar`

```
<bean id="myBean" class="MyBean">
    <property name="location" value="/jdk/lib/tools.jar" />
</bean>
```

5.3.3. 重命名资源

重命名资源是指对于即有的资源，改变其名字。

为什么需要修改资源的名字？理由是：*取消资源名称和环境的关联性*。有一些资源的名称，具有明显的环境相关性，比如：

- `classpath:myFile.xml`或者`/classpath/myFile.xml` —— 从资源的名称就可以看出，这些资源是从classpath中装载的。
- `/WEB-INF/myFile.xml`或者`/webroot/WEB-INF/myFile.xml` —— 从资源的名称可以看出，这些资源是从web应用中装载的。

使用和环境相关的资源名称有什么问题？问题就是，当环境改变时，应用代码会受到影响。最常见的一种状况是：单元测试时，用于测试的资源文件往往被放在专供测试的目录中，这些目录和应用运行时的环境是不同的 —— 你可能希望将`classpath:myFile.xml`或`/WEB-INF/myFile.xml`改成`/src/test/config/myFile.xml`。

对资源重命名就可以解决这类问题：

- 将`classpath:myFile.xml`或者`/WEB-INF/myFile.xml`重命名成：`myapp/conf/myFile.xml`。
- 在测试环境中，将`myapp/conf/myFile.xml`名称指向另一个物理地址`src/test/config/myFile.xml`。

重命名资源是通过alias别名实现的：

例 5.21. 重命名资源

```
<resource-loading
        xmlns="http://www.alibaba.com/schema/services"
        xmlns:res-loaders="http://www.alibaba.com/schema/services/resource-loading/loaders">
    ...

    <resource-alias pattern="/myapp/conf" name="/webroot/WEB-INF" /> ❶

    <resource pattern="/webroot" internal="true"> ❷
        <res-loaders:webapp-loader /> ❸
    </resource>

</resource-loading>
```

❶ 定义了一个资源的别名：`/myapp/conf`。

当你查找`/myapp/conf/myFile.xml`时，Resource Loading服务实际上会去找`/webroot/WEB-INF/myFile.xml`。而`/webroot/*`则是由 ❷ 所定义的资源。

❷ 定义以`/webroot`为前缀的新资源。

其中，attribute *`internal=true`*是一个可选项，当它的值为`true`时，代表它所修饰的资源是不能被外界所直接访问的。例如，你想直接在`myBean`中注入`/webroot/WEB-INF/myFile.xml`是不行的。把`internal`选项设成`true`，可以让强制用户转向新的资源名称。`Internal`参数的默认值为`false`，意味着，新旧两种名称同时可用。

❸ <webapp-loader>表示从Web应用中装载资源。详见：第 5.4.2 节 “WebappResourceLoader” 。

5.3.4. 重定向资源

重定向资源的意思是，将部分资源名称，指向另外的地址。

一个常见的需求是这样的：通常我们会把页面模板保存在WEB应用的/templates目录下。但是有一批模板是由外部的CMS系统生成的，这些模板文件不可能和WEB应用打包在一起，而是存放在某个外部的目录下的。我们希望用/templates/cms来引用这些模板。

由于/templates/cms只不过是/templates的子目录，所以如果没有Resource Loading服务所提供的重定向功能，是不可能实现上述功能的。用Resource Loading服务重定向的配置如下：

例 5.22. 重定向资源

```
<resource-loading
        xmlns="http://www.alibaba.com/schema/services"
        xmlns:res-loaders="http://www.alibaba.com/schema/services/resource-loading/loaders">
    ...

    <resource-alias pattern="/templates" name="/webroot/templates" /> ❶

    <resource pattern="/templates/cms"> ❷
        <res-loaders:file-loader basedir="${cms_root}" />
    </resource>

    <resource pattern="/webroot" internal="true">
        <res-loaders:webapp-loader />
    </resource>

    ...
</resource-loading>
```

❶ 定义了一个资源的别名：/templates，指向internal资源：/webroot/templates。

❷ 将/templates的子目录/templates/cms重定向到某个外部的文件目录${cms_root}中。

其中cms_root是启动服务器时所指定的system property（-Dcms_root=...）或者spring所定义的placeholder。

通过上述配置，可以达到如下效果：

表 5.3. 访问/templates目录下的资源

资源名	如何装载?
/templates/xxx.vm	不受重定向影响。访问/webroot/templates/xxx.vm，继而通过webapp-loader访问Web应用根目录下的/templates/xxx.vm文件。
/templates/cms/yyy.vm	*被重定向*。通过file-loader访问${cms_root}目录下的文件：${cms_root}/yyy.vm。
/templates/subdir/zzz.vm	不受重定向影响。访问/webroot/templates/subdir/zzz.vm，继而通过webapp-loader访问Web应用根目录下的/templates/subdir/zzz.vm文件。

最重要的是，*访问/templates目录的应用程序并不知道这个资源重定向的存在*，当cms所对应的实际目录被改变时，应用程序也不会受到任何影响 —— 这个正是Resource Loading服务的“魔法” 。

5.3.5. 匹配资源

无论是定义新资源（<resource>）或是重命名资源（资源别名、<resource-alias>），都需要指定一个pattern attribute来匹配资源的名称。

5.3.5.1. 匹配绝对路径和相对路径

资源或资源别名的pattern支持对绝对路径和相对路径的匹配：

表 5.4. 资源或别名的pattern格式

pattern类型	格式	说明
匹配绝对路径	/absolute/path	*以/开头的pattern*代表一个绝对路径的匹配。 例如：pattern="/absolute/path"可以匹配资源名/abslute/path/xxx/yyy，但不能匹配资源名/xxx/abslute/path/yyy。
匹配相对路径	relative/path	*不以/开头的pattern*代表一个相对路径的匹配。 例如：pattern="relative/path"可以匹配资源名/relative/path/xxx/yyy，也可以匹配资源名/xxx/relative/path/yyy。

5.3.5.2. 匹配通配符

表 5.5. 通配符格式

格式	说明
星号 *	匹配0-n个字符，但不包括“/”。即，“*”只匹配一级目录或文件中的零个或多个字符。
双星号 **	匹配0-n个字符，包括“/”。即，“**”匹配多级目录或文件。
问号 ?	匹配0-1个字符，但不包括“/”。即，“?”匹配一级目录或文件中的零个或一个字符。

所有被通配符匹配的内容，将被按顺序赋给变量“$1”、“$2”、“$3”、“$4”、……。这些变量可以在其它地方被引用。

通配符匹配的名称既可以是绝对路径，也可以是相对路径。把相对路径和通配符结合起来的最常见用法，就是匹配文件名后缀，例如：pattern="*.xml"。

下面是一些使用通配符的例子：

例 5.23. 用通配符来匹配资源名称或资源别名

重命名WEB-INF及其子目录下的所有的xml文件

例如，将/myapp/conf/my/file.xml转换成/webroot/WEB-INF/my/file.xml。

```
<resource-alias pattern="/myapp/conf/**/*.xml" name="/webroot/WEB-INF/$1/$2.xml" />
```

修改文件名后缀

例如，将/myapp/conf/myfile.conf转换成/webroot/WEB-INF/myfile.xml。

```
<resource-alias pattern="/myapp/conf/*.conf" name="/WEB-INF/$1.xml"/>
```

按首字母划分子目录

将a开头的文件名放到a子目录下，b开头的文件名放到b子目录下，以此类推。

例如，将/profiles/myname转换成文件路径${profile_root}/m/myname；将/profiles/othername转换成文件路径${profile_root}/o/othername。

```
<resource pattern="/profiles/?*">
    <res-loaders:file-loader basedir="${profile_root}">
        <res-loaders:path>$1/$1$2</res-loaders:path>
    </res-loaders:file-loader>
</resource>
```

5.3.6. 在多个ResourceLoader中查找

假如，在我的Web应用中，我有一些配置文件放在/WEB-INF目录中，另外一部分配置放在classpath中。我可以这样做：

例 5.24. 在多个ResourceLoader中查找

```
<resource-loading
        xmlns="http://www.alibaba.com/schema/services"
        xmlns:res-loaders="http://www.alibaba.com/schema/services/resource-loading/loaders">
    ...

    <resource pattern="/myapp/conf">
        <res-loaders:super-loader name="/webroot/WEB-INF" /> ❶
        <res-loaders:super-loader name="/classpath" /> ❷
    </resource>

    <resource pattern="/webroot" internal="true"> ❸
        <res-loaders:webapp-loader />
    </resource>
    <resource pattern="/classpath" internal="true"> ❹
        <res-loaders:classpath-loader />
    </resource>

    ...
</resource-loading>
```

❶❷ 依次尝试两个loaders。

❸ 定义internal资源/webroot/*，从Web应用中装载资源。详见第 5.4.2 节 “WebappResourceLoader” 。

❹ 定义internal资源/classpath/*，从classpath中装载资源。详见第 5.4.3 节 “ClasspathResourceLoader” 。

Resource Loading服务根据上面的配置，会这样查找资源“/myapp/conf/myFile.xml”：

1. 先查找：/webroot/WEB-INF/myFile.xml，如果找不到，

2. 则再查找：/classpath/myFile.xml，如果找不到，则放弃。

在上例中，<super-loader>（详见第 5.4.4 节 “SuperResourceLoader”）是一种特殊的ResourceLoader，它等同于<resource-alias>。下面的两种写法是完全等同的：

例 5.25. <super-loader>和等效的<resource-alias>

```
<resource pattern="/myapp/conf">
    <res-loaders:super-loader name="/webroot/WEB-INF" />
</resource>

<resource-alias pattern="/myapp/conf " name="/webroot/WEB-INF" />
```

但是用<resource-alias>没有办法实现上面所述的多重查找的功能。

5.3.7. 装载parent容器中的资源

在Webx中，Spring容器被安排成级联的结构。

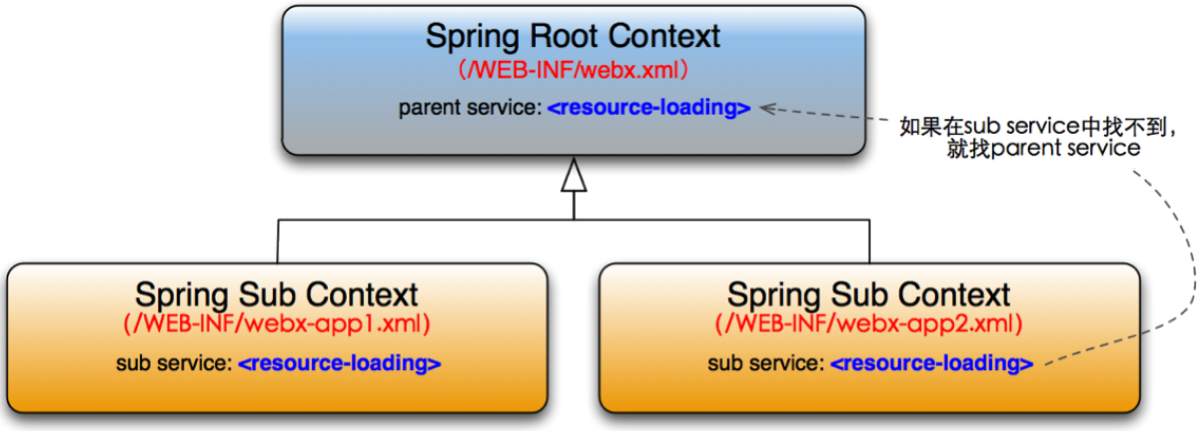


图 5.3. Spring容器的级联结构

如图所示，每个Spring容器都可以配置自己的Resource Loading服务。当调用子容器的Resource Loading服务时，遵循这样的逻辑：

1. 先在子容器的Resource Loading服务中查找资源，如果找不到，

2. 则再到parent容器的Resource Loading服务中查找，如果找不到，则放弃。

运用这种级联装载资源的方法，子应用可以把共享的资源定义在root context中，而把自己独享的资源定义在自己的容器当中。

前文所述的<super-loader>也支持级联装载资源。<super-loader>会先在当前容器的Resource Loading服务中查找，如果找不到，就到parent容器的Resource Loading服务中查找。利用<super-loader>，你甚至可以改变资源搜索的顺序。例如，你可以命令Resource Loading服务先查找parent容器中的Resource Loading服务，再查找当前容器中的ResourceLoaders：

例 5.26. 利用<super-loader>改变资源搜索的顺序

```
<resource pattern="...">
    <res-loaders:super-loader /> ❶
    <res-loaders:file-loader /> ❷
</resource>
```

❶ 先找parent容器中的Resource Loading服务。

❷ 再找当前容器中的ResourceLoaders。

5.3.8. 修改资源文件的内容

Resource Loading服务支持内容过滤 —— 你可以在获取资源以前读取甚至修改资源文件的内容。一种常见的情形是，将XML格式的资源文件用XSLT转换格式：

例 5.27. 将XML格式的资源文件用XSLT转换格式

```
<resource-loading
        xmlns="http://www.alibaba.com/schema/services"
        xmlns:res-loaders="http://www.alibaba.com/schema/services/resource-loading/loaders">
        xmlns:res-filters="http://www.alibaba.com/schema/services/resource-loading/filters">
    ...

    <resource-filters pattern="test-*.xml">
        <res-filters:xslt-filter xslt="/stylesheet.for.test/test.xsl" saveTo="/tempdir" /> ❶
    </resource-filters>

    <resource pattern="/tempdir"> ❷
        <loaders:file-loader basedir="${project.home}/target/test" />
    </resource>

</resource-loading>
```

❶ 将*所有目录下*（因为是相对路径）的名称为`test-*.xml`文件，用指定的XSL文件进行转换。

这里引进了一种新的扩展点：`ResourceFilter`。`ResourceFilter`可以在应用获取资源之前，取得控制，以便对资源做一点事。

`<xslt-filter>`是对`ResourceFilter`的扩展，它能够把XML资源用指定的xsl文件转换成新的格式。假如指定了`saveTo`参数，就可以把转换的结果保存下来，避免每次访问都重新转换。

❷ 此处定义`tempdir`目录资源，以便保存xslt转换的结果。

注意

`<xslt-filter>`的参数`xslt`所指向的xsl文件，以及参数`saveTo`所指向的目录，它们本身也是由Resource Loading服务装载的。

有哪些情况需要这种内容过滤的功能呢?

- 单元测试 —— 我们可能需要对单元测试的资源文件进行特殊的转换。
- 高速缓存 —— 有一些`ResourceLoader`可能会有性能的开销，例如：从数据库中装载资源。利用`ResourceFilter`功能，就可以把装载的资源缓存在高速cache中，以提高系统的性能。

5.3.9. 直接使用`ResourceLoadingService`

前面所讲的Resource Loading服务的用法，对应用程序而言，是完全透明的。也就是说，应用程序并不需要关心Resource Loading服务的存在，而是按照Spring `ResourceLoader`的老用法，就可以工作。

但是你也可以直接注入`ResourceLoadingService`对象，以取得更多的功能。

例 5.28. 注入`ResourceLoadingService`对象

```
public class MyClass {
    @Autowired
    private ResourceLoadingService resourceLoadingService;
}
```

下面列举了可通过`ResourceLoadingService`接口实现的功能。

取得资源

例 5.29. 通过ResourceLoadingService接口取得资源

```
Resource resource = resourceLoadingService.getResource("/myapp/conf/myFile.xml"); ❶
Resource resource = resourceLoadingService.getResource("/myapp/conf/myFile.xml",
                                                       ResourceLoadingService.FOR_CREATE); ❷
```

❶ 和Spring不同的是，如果你直接调用ResourceLoadingService取得资源，当资源文件不存在时，你会得到一个ResourceNotFoundException。而Spring无论如何都会取得Resource对象，但随后你需要调用Resource.exists()方法来判断资源存在于否。

❷ ResourceLoadingService.getResource()方法还支持一个选项：FOR_CREATE。如果提供了这个选项，那么对于某些类型的资源（如文件系统的资源），即使文件或目录不存在，仍然会返回结果。这样，你就可以创建这个文件或目录 —— 这就是FOR_CREATE参数的意思。

取得特定类型的资源

例 5.30. 通过ResourceLoadingService接口取得特定类型的资源

```
// 取得资源文件
File file = resourceLoadingService.getResourceAsFile("/myapp/conf/myFile.xml");

// 取得资源URL
URL url = resourceLoadingService.getResourceAsURL("/myapp/conf/myFile.xml");

// 取得资源输入流
InputStream stream = resourceLoadingService.getResourceAsStream("/myapp/conf/myFile.xml");
```

判断资源存在于否

例 5.31. 通过ResourceLoadingService接口判断资源存在于否

```
if (resourceLoadingService.exists("/myapp/conf/myFile.xml")) {
    ...
}
```

列举子资源

例 5.32. 通过ResourceLoadingService接口列举子资源

```
String[] resourceNames = resourceLoadingService.list("/myapp/conf");
Resource[] resources = resourceLoadingService.listResources("/myapp/conf");
```

相当于列出当前目录下的所有子目录和文件。

不是所有的ResourceLoader都支持这个操作 —— FileResourceLoader和WebappResourceLoader支持列举子资源，ClasspathResourceLoader则不支持。

跟踪取得资源的过程

例 5.33. 通过ResourceLoadingService接口跟踪取得资源的过程

```
ResourceTrace trace = resourceLoadingService.trace("/myapp/conf/webx.xml");

for (ResourceTraceElement element : trace) {
    System.out.println(element);
}
```

这是用来方便调试的功能。有点像Throwable.getStackTrace()方法，可以得到每一个方法调用的历史记录 —— ResourceLoadingService.trace()方法可以将取得资源的步骤记录下来。上面代码会在console中输出类似下面的内容：

```
"/myapp/conf/webx.xml" matched [resource-alias pattern="/myapp/conf"], at "resources.xml",
 beanName="resourceLoadingService"
"/webroot/WEB-INF/webx.xml" matched [resource pattern="/webroot"], at "resources.xml",
 beanName="resourceLoadingService"
```

列出所有可用的资源定义和别名的pattern

例 5.34. 通过ResourceLoadingService接口列出所有可用的资源定义和别名的pattern

```
String[] patterns = resourceLoadingService.getPatterns(true);
```

5.3.10. 在非Web环境中使用Resource Loading服务

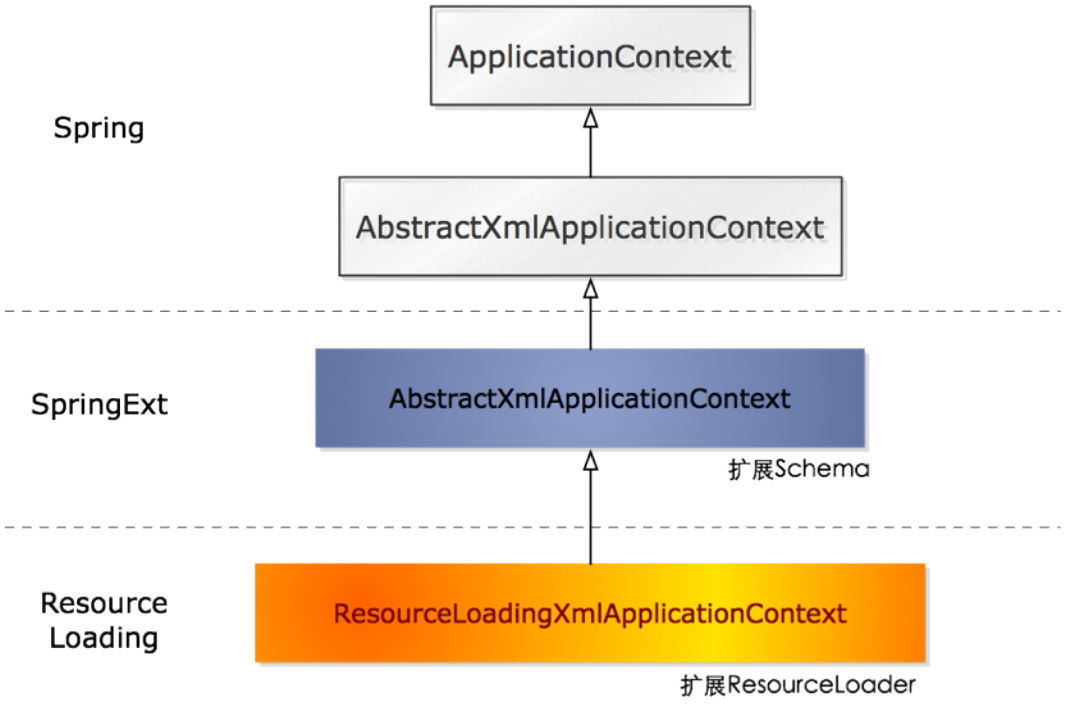


图 5.4. 在非Web环境中使用的ResourceLoadingXmlApplicationContext

在非Web环境中使用Resource Loading服务的最好方法，是创建ResourceLoadingXmlApplicationContext作为Spring容器。

例 5.35. 创建ResourceLoadingXmlApplicationContext容器

```
ApplicationContext context = new ResourceLoadingXmlApplicationContext(
                                                    new FileSystemResource("beans.xml"));
```

只要beans.xml中包含<resource-loading>的配置，就会自动启用Resource Loading服务，并取代Spring原来的ResourceLoader机制。

5.4. ResourceLoader参考

Resource Loading服务的核心是ResourceLoader。和Spring ResourceLoader不同，Resource Loading服务的ResourceLoader是可扩展的轻量级对象，担负着装载某一种类型的资源的具体任务。例如FileResourceLoader负责装载文件系统的资源；WebappResourceLoader负责装载WEB应用中的资源等等。

当你需要新的资源装载方式时，你所要做的，就是实现一种新的ResourceLoader。例如，你想从数据库中装载资源，那么就可以实现一个DatabaseResourceLoader。

5.4.1. FileResourceLoader

FileResourceLoader的功能是：从文件系统中装载资源。

基本用法

例 5.36. FileResourceLoader的基本用法

```
<resource pattern="/my/virtual">
    <res-loaders:file-loader />
</resource>
```

这样，file-loader会从哪里装载资源呢?

答案是：*从当前配置文件所在的目录中装载*。假如上述资源配置所在的配置文件是c:/myapp/conf/resources.xml，那么file-loader就会从c:/myapp/conf/myFile.xml文件中装载/my/virtual/myFile.xml资源。

这样做的思路源自于Apache的一个项目：Ant。Ant是一个广为使用的build工具。每一个Ant项目，都有一个build.xml脚本，在里面定义了很多target，诸如编译项目、打包等。通常我们都会把build.xml这个文件放在项目的根目录中，然后build.xml中的命令全是使用相对于build.xml所在的项目根目录计算出来的相对路径。例如：

例 5.37. Ant脚本（build.xml）

```
<project basedir=".">
    ...
    <target ...>
        <copy todir="bin">
            <fileset dir="src"/>
        </copy>
    </target>
    ...
</project>
```

在上面的Ant脚本中，bin、src目录全是相对于build.xml所在目录的相对目录。这样做的好处是，当你把项目移到不同的环境中，你也无需改变配置文件和脚本。

FileResourceLoader采用了和Ant完全类似的想法。

指定basedir

例 5.38. 在FileResourceLoader中指定basedir

```
<resource pattern="/my/virtual">
    <res-loaders:file-loader basedir="${my.basedir}" />
</resource>
```

FileResourceLoader当然也支持指定basedir根目录。这样，它就会从指定的basedir的子目录中查找资源。

一般来说，我们需要利用Spring Property Placeholder来设置basedir。在上面的例子中，我们可以在系统启动时，指定JVM参数：-Dmy.basedir=c:/mydata。在不同的系统环境中，必须指定正确的basedir，否则，<file-loader>有可能找不到资源。

搜索多个路径

例 5.39. 在FileResourceLoader中指定多个搜索路径

```
<resource pattern="/my/virtual">
    <res-loaders:file-loader basedir="...">
        <res-loaders:path>relativePathToBasedir</res-loaders:path> ❶
        <res-loaders:path type="absolute">c:/absolutePath</res-loaders:path> ❷
    </res-loaders:file-loader>
</resource>
```

❶ 搜索路径默认为相对路径，相对于指定的basedir。如果basedir未指定，则相对于当前resource-loading所在的配置文件的路径。

❷ 搜索路径也可以是绝对路径。

FileResourceLoader支持搜索多个路径，类似于操作系统在PATH环境变量所指定的路径中，搜索可执行文件；也类似于Java在CLASSPATH参数所指定的路径中，搜索classes。

5.4.2. WebappResourceLoader

WebappResourceLoader的功能是：从当前WEB应用中装载资源，也就是从ServletContext对象中装载资源。

例 5.40. 配置WebappResourceLoader

```
<resource pattern="/my/virtual">
    <res-loaders:webapp-loader />
</resource>
```

5.4.3. ClasspathResourceLoader

ClasspathResourceLoader的功能是：从classpath中装载资源，也就是从当前的ClassLoader对象中装载资源。

例 5.41. 配置ClasspathResourceLoader

```
<resource pattern="/my/virtual">
    <res-loaders:classpath-loader />
</resource>
```

5.4.4. SuperResourceLoader

SuperResourceLoader的功能是：调用Resource Loading服务来取得资源。它有点像Java里面的super操作符。

取得新名字所代表的资源

例 5.42. 用SuperResourceLoader取得新名字所代表的资源

```
<resource pattern="/my/virtual">
    <res-loaders:super-loader basedir="/webroot/WEB-INF" />
</resource>
```

这个操作类似于<resource-alias>。

如果在当前context的Resource Loading服务中找不到资源，它会前往parent context中查找。

在parent context中查找资源

例 5.43. 用SuperResourceLoader查找parent context中的资源

```
<resource pattern="/my/virtual">
    <res-loaders:super-loader />
</resource>
```

如果你不指定name参数，那么SuperResourceLoader会直接去parent context中查找资源，而不会在当前context的Resource Loading服务中找。

5.4.5. 关于ResourceLoader的其它考虑

以上所有的ResourceLoader都被设计成可以在任何环境中工作，即使当前环境不适用，也不会报错。

WebappResourceLoader可以兼容非WEB环境

在非WEB环境中，例如单元测试环境、你直接通过XmlApplicationContext创建的Spring环境，WebappResourceLoader也不会出错 —— 只不过它找不到任何资源而已。

SuperResourceLoader可以工作于非级联的环境

也就是说，即使parent context不存在，或者parent context中没有配置Resource Loading服务，SuperResourceLoader也是可以工作的。

这样，同一套资源配置文件，可以被用于所有环境。

5.5. 本章总结

Resource Loading服务提供了一套高度可扩展的、强大的资源装载机制。这套机制和Spring ResourceLoader无缝连接。使用它并不需要特殊的技能，只要掌握Spring的风格即可。

第 6 章 Filter、Request Contexts和Pipeline

Filter是Servlet规范2.3版及更新版所支持的一种机制。和Servlet/JSP不同，Filter自己往往不会直接产生response，相反，它提供了一种“符加”的功能，可以作用在任何一个servlet、JSP以及其它filter之上。然而，在实际的应用中，我们发现filter有很多不足之处。

Webx框架提供了两种机制（Request Contexts和Pipeline）来作为filter机制的补充。在大多数情况下，它们都可以实现类似filter的功能，但比filter更容易扩展、更容易配置、也更轻量。Webx并没有打算完全替代filter，相反它还是可以和任何filter搭配使用。

本章先简略介绍filter的功能和不足，再向你介绍Request Contexts和Pipeline的工作原理，及使用方法。

6.1. Filter

6.1.1. Filter的用途

Filter这种机制常被用来实现下面的功能：

页面授权	根据登录用户的权限，阻止或许可用户访问特定的页面。
日志和审计	记录和检查用户访问WEB应用的情况。
图片转换	改变图片的格式、精度、尺寸等。
页面压缩	压缩页面内容，加快下载速度。
本地化	显示本地语言和风格的页面。
XSLT转换	对XML内容进行XSLT转换，使之适用于多种客户端。
高速缓存	高速缓存页面，提高响应速度。

当然还有更多种的应用，我们不可能一一列举。

Filter的通用性很好。任何filter均独立于其它filter和servlet，因此它可以和任意其它filter和servlet组合搭配。下面是一段配置示例 —— 通过`SetLoggingContextFilter`，日志系统可以记录当前请求的信息，例如：URL、referrer URL、query string等。

例 6.1. Filter配置示例（**/WEB-INF/web.xml**）

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/j2ee  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd
    ">
    <filter>
        <filter-name>mdc</filter-name>
        <filter-class>com.alibaba.citrus.webx.servlet.SetLoggingContextFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>mdc</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    ...
</web-app>
```

6.1.2. Filter工作原理

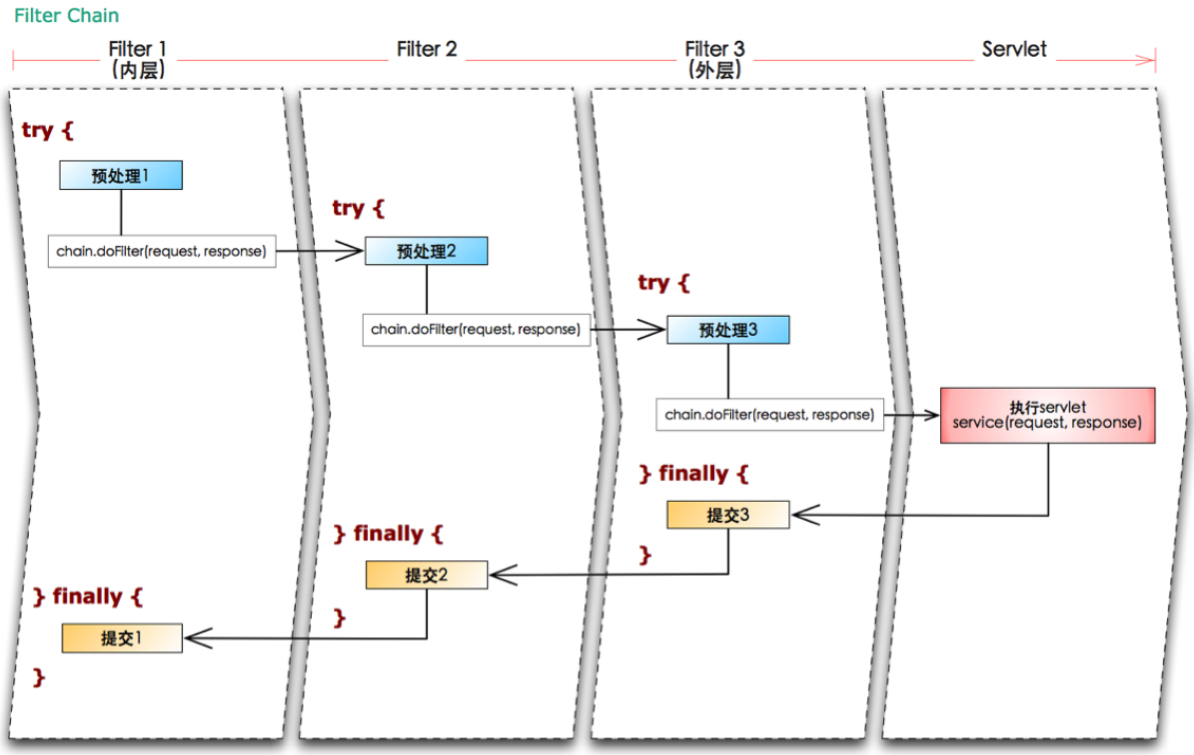


图 6.1. Filter Chain

如图所示。多个filter和至多一个servlet被串联成一个链，被称为Filter Chain。执行的时候，引擎将控制权交给链条中的头一个filter（如果有的话）。然后，就像*击鼓传花*一样，控制权被依次传递给filter chain中的下一个filter或servlet。每一个得到控制权的filter可以做下面的事：

- 继续传递控制权或立即终止filter chain。
  - Filter可将控制权传递给链条中的下一个filter或者最终的servlet。
  - Filter也可以不将控制权传递给下一个filter或servlet，这样便中止了整个filter chain的执行。

- 预处理。在传递控制权给下一个filter或servlet之前，filter可以预先做一些事情：
  - 设置request、response中的参数，例如：character encoding、content type等。
  - 将`HttpServletRequestWrapper`传递给链条中的下一位，filter可以通过wrapper改变request中的任意值。
  - 将`HttpServletResponseWrapper`传递给链条中的下一位，filter可以通过wrapper来拦截后续filter或servlet对response的修改。
- 提交。在控制权从filter chain中返回以后，filter还可以做一些后续提交的操作。
  - 例如，将response中拦截而来的数据，压缩或转换格式，并发送给客户端或filter chain的上一级。
  - 通过`try`、`catch`还可以捕获filter chain下一级所有的异常，并做处理。

6.1.3. Filter的限制

Filter是很有用的。作为servlet的补充，filter也是很成功的。但是filter并没有被设计用来完成一切事情。事实上，filter的设计限制了filter的用途。每个filter具有下面的限制：

- Filter可以访问和修改数据。但它只能访问和修改`HttpServletRequest`、`HttpServletResponse`、`ServletContext`等容器级的对象，而不能（或很难）访问应用程序中的状态。所以filter无法实现和应用逻辑密切相关的功能。
- Filter可以影响执行流程。但它不能改变filter chain的结构和顺序。Filter chain的结构和顺序是由`web.xml`中定义的。当filter得到控制权以后，它只能选择继续下去，或者立即结束，而没法进行循环、分支、条件判断等更复杂的控制。因此，filter只能用来实现粗粒度的流程控制功能（例如，当用户未获授权时，停止执行filter chain），难以应付更细致的应用程序内的控制需求。
- Filter与其它filter和servlet之间，除了request和response对象以外，无法共享其它的状态。这既是优点又是缺点。优点是使filter更独立、更通用；缺点是filter与其它filter、servlet之间难以协作，有时甚至会引起无谓的性能损失。

6.1.4. Webx对filter功能的补充

综上所述，一个filter常常做的两件事是：

- 改变request/response对象（通过`HttpServletRequestWrapper`和`HttpServletResponseWrapper`）；
- 改变应用执行的流程。

其实，大部分filter只做其中一件事。例如：

- 页面压缩filter仅仅改变response，并不改变应用的流程。
- 页面授权filter根据当前请求用户的身份，判定他是否有权限访问当前页面。这个filter会影响应用流程，却不会去改变request和response。

当然也有例外。有一些filter不做上面两件事中任何一件。例如，日志filter仅仅读取request对象并记录日志而已，既不改变request/response，也不影响应用的流程。还有一些filter同时做上面两件事。比如高速缓存页面的filter不仅要修改response，而且当cache被命中时，不再执行下一步的流程，而是直接返回cache中的内容，以提高性能。

Webx框架提供了两个服务，正好吻合了上述两个最常用的filter的功能。

Request Contexts服务	该服务负责访问和修改request和response，但不负责改变应用执行的流程。
Pipeline服务	提供应用执行的流程，但不关心request和response。

虽然这两个服务看起来和filter的功能类似，但是它们远比filter要强大和方便 —— 它们克服了上述filter的几个限制：

- 和Filter不同，Request Contexts和Pipeline服务可以访问应用内部的状态和资源，效率更高，功能更强。
- 和Filter不同，Pipeline服务可以定义灵活（但仍然简单）地控制应用的流程 。Pipeline不仅可以控制流程的中断或继续，还可以实现子流程、循环、条件转移、异常处理等更精细的流程控制。Pipeline服务甚至可以运用在非WEB的环境中。
- 和Filter不同，Request Contexts服务中的每一个环节（Request Context）之间并非完全独立、互不干涉的。每个request context可以访问它所依赖的其它request context中的状态。

6.2. Request Contexts服务

6.2.1. Request Contexts工作原理

Request Context，顾名思义，就是一个请求的上下文。事实上，你可以把Request Context看作是`HttpServletRequest`和`HttpServletResponse`这两个对象的总和。除此之外，多个Request Context可以被串接起来，被称为Request Context Chain，类似于filter chain。

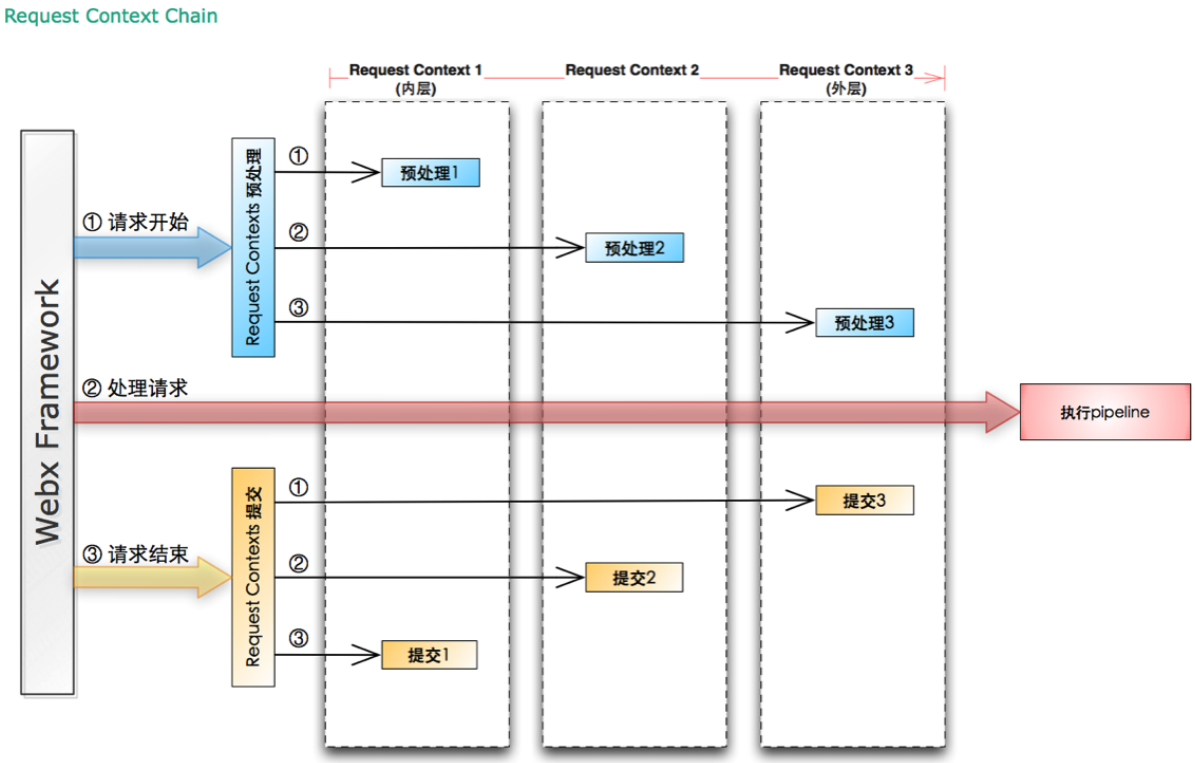


图 6.2. Request Context Chain

如上图所示，每一个Request Context都可以包括两个基本的操作：“预处理”和“提交”。

- 在一个请求开始的时候，每个Request Context的“预处理”过程被依次调用。最内层的（即最先的）Request Context最先被调用，最外层的（即最后的）Request Context最后被调用；
- 在一个请求结束的时候，每个Request Context的“提交”过程被依次调用。和“预处理”的顺序相反，最外层的（即最后的）Request Context最先被调用，最内层的（即最先的）Request Context最后被调用。

Request Context在预处理的时候，可以利用`HttpServletRequestWrapper`和`HttpServletResponseWrapper`来包装和修改request和response —— 这一点和filter相同。每一层Request Context，都会增加一个新的特性。最先的Request Context成为最内层的包装，最后的Request Context成为最外层的包装。如下图所示。

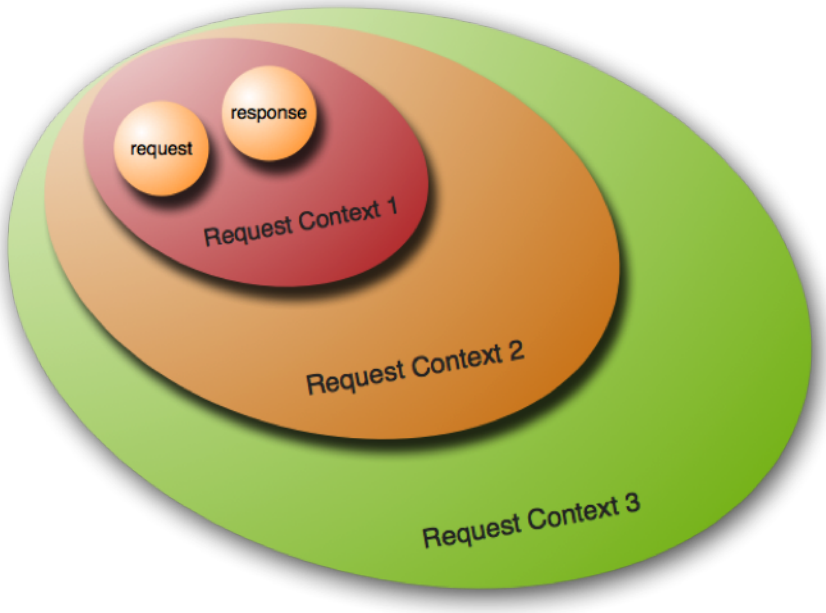


图 6.3. Request Contexts的嵌套

和filter原理中的图进行对比，你会发现，尽管Request Contexts和Filter的执行方案有明显的不同，但是Request Contexts预处理和提交的顺序是和filter chain完全一致的。预处理时，由内层执行到外层；提交时，反过来由外层执行到内层。不同的是，filter能够决定是否继续传递控制权给filter chain中的下一位，而Request Context则没有这个权利。

6.2.2. Request Contexts的用途

Webx目前提供了以下几种request context的实现，每个都有独特的功能。

表 6.1. Request Contexts的功能

名称	功能
`<basic>`	提供基础安全特性，例如：过滤response headers、cookies，限制cookie的大小等。

名称	功能
<buffered>	缓存response中的内容。
<lazy-commit>	延迟提交response。
<parser>	解析参数，支持multipart/form-data（即上传文件请求）。
<rewrite>	重写请求的URL和参数。
<session>	一套可扩展的session框架，重新实现了HttpSession接口。
<set-locale>	设置locale区域和charset字符集编码。

注意

本章对以上所有的request contexts的功能和用法不作具体的介绍，详情请参阅*第 7 章 Request Contexts功能指南*和*第 8 章 Request Context之Session指南*。

需要特别指出的是，你还可以扩展出更多的Request Context，以实现新的功能。

6.2.3. Request Contexts的使用

6.2.3.1. 配置

除了下面例子所示的一段配置之外，你不需要做太多的事，就可以使用Request Contexts。因为Request Contexts对于应用来说是透明的 —— 多数应用只需要依赖于`HttpServletRequest`和`HttpServletResponse`就可以了。

例 6.2. Request Context的配置（`/WEB-INF/webx.xml`）

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans:beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:services="http://www.alibaba.com/schema/services"
    xmlns:request-contexts="http://www.alibaba.com/schema/services/request-contexts"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="
        http://www.alibaba.com/schema/services
                http://localhost:8080/schema/services.xsd
        http://www.alibaba.com/schema/services/request-contexts
                http://localhost:8080/schema/services-request-contexts.xsd
        http://www.springframework.org/schema/beans
                http://localhost:8080/schema/www.springframework.org/schema/beans/spring-beans.xsd
    ">
    ...

    <services:request-contexts xmlns="http://www.alibaba.com/schema/services/request-contexts">
        <basic />
        <buffered />
        <lazy-commit />
        <parser />
        <set-locale defaultLocale="zh_CN" defaultCharset="UTF-8" />
        <!-- Optional -
        <session />
        <rewrite />
        -->
    </services:request-contexts>

    <services:upload sizeMax="5M" />

</beans:beans>
```

由于使用了SpringExt的schema机制，所以在支持schema的XML编辑器的帮助下，很容易书写和验证Request Contexts的配置。

6.2.3.2. 排序

Request Contexts之间，有时会有依赖关系，所以Request Contexts出现的先后顺序是非常重要的。例如，

- `<session>`提供了基于cookie的session支持。然而cookie属于response header。一旦response被提交，header就无法再修改了。因此`<session>`依赖于`<lazy-commit>`，以阻止response过早提交。也就是说，`<lazy-commit>`必须排在`<session>`之前。
- `<rewrite>`需要访问参数，而参数是能过`<parser>`解析的，所以`<parser>`要排在`<rewrite>`之前。

类似的约束还有很多。如果把Request Contexts的顺序排错，可能会导致某项功能错误或失效。然而，对于一般的应用开发者而言，这些约束往往是神秘的、并非显而易见的，需要经过细致地分析才能了解它们。

好在Request Contexts内部提供了一个机制，可以根据预定义的约束条件，对所有的Request Contexts进行自动排序。和Filter不同，应用开发者不需要在意Request Contexts在配置文件中的排列顺序，就可以保证所有的Request Contexts能够正常工作。下面的两种配置文件是等效的：

例 6.3. Request Contexts等效配置1

```
<services:request-contexts>
    <basic />
    <buffered />
    <lazy-commit />
    <parser />
    <set-locale />
    <session />
    <rewrite />
</services:request-contexts>
```

例 6.4. Request Contexts等效配置2

```
<services:request-contexts>
    <rewrite />
    <session />
    <set-locale />
    <parser />
    <lazy-commit />
    <buffered />
    <basic />
</services:request-contexts>
```

6.2.3.3. 访问特定的Request Context

一般来说，Request Contexts对于应用程序是透明的 —— 也就是说，应用程序最多只需要访问Servlet API中的接口：`HttpServletRequest`和`HttpServletResponse`即可，就好像Request Contexts不存在一样。

比如，Request Context `<parser>`能够解析`multipart/form-data`类型的请求（即上传图片请求）。但你不需要用另一个API来访问请求中的普通数据，你只需要用`HttpServletRequest`中定义的方法就可以访问，仿佛这是一个普通的请求：

例 6.5. 访问任意类型的请求中的参数

```
String value = request.getParameter("myparam");
```

再比如，Request Context `<session>`重新实现了`HttpSession`的接口，但是应用程序并不需要关心这些，他们还是和原来一样访问session：

例 6.6. 访问session

```
HttpSession session = request.getSession();

String value = (String) session.getAttribute("myattr");
session.setAttribute("myattr", newValue);
```

然而，有一些功能在原有的Servlet API中是不存在的。对于这一类功能，你必须访问特定的`RequestContext`接口，才能使用它们。例如，你只能用另一个API才能读取用户上传的文件。下面的代码可以用来取得上传文件的信息：

例 6.7. 访问特定的RequestContext接口

```
ParserRequestContext parserRequestContext =
                RequestContextUtil.findRequestContext(request, ParserRequestContext.class);

ParameterParser params = parserRequestContext.getParameters();

FileItem myfile = params.getFileItem("myfile");

String filename = myfile.getName();
InputStream istream = myfile.getInputStream();
```

另外有一些功能，使用Request Context接口比原来的Servlet API接口更方便。例如，原来的`request.getParameter()`方法只能取得字符串的参数值，但是利用`ParserRequestContext`所提供的接口，就可以直接取得其它类型的值：

例 6.8. 通过ParserRequestContext接口访问参数比HttpServletRequest更方便

```
ParameterParser params = parserRequestContext.getParameters();

String stringValue = params.getString("myparam"); // 取得字符串值，默认为null
int intValue = params.getInt("myparam"); // 取得整数值，默认为0
boolean booleanValue = params.getBoolean("myparam", true); // 取得boolean值，指定默认值为true
```

6.2.3.4. 注入request作用域的对象

Spring最强大的功能是依赖注入。但是依赖注入有一个限制：小作用域的对象不能被注入到大作用域的对象。你不能够把request和session作用域的对象注入到singleton对象中。前者在每次WEB请求时，均会创建新的实例，每个线程独享这个request/session作用域的对象；后者是在Spring初始化或第一次使用时被创建，然后被所有的线程共享。假如你把某个request/session作用域的对象意外注入到singleton对象中，将可能产生致命的应用错误，甚至导致数据库的错乱。

表 6.2. Webx中的重要对象及其作用域

对象类型	作用域
`ServletContext`	Singleton scope
`HttpServletRequest`	Request scope
`HttpServletResponse`	Request scope
`HttpSession`	Session scope
所有`RequestContext`对象， 如：`ParserRequestContext`、`SessionRequestContext`等	Request scope

在一般的情况下，对于一个singleton对象而言，例如，Webx中的action module、pipeline valve对象等，下面的代码是错误的：

例 6.9. 在action（singleton对象）中注入request scope的对象

```
public class MyAction {
    @Autowired
    private HttpServletRequest request;

    @Autowired
    private HttpServletResponse response;

    @Autowired
    private ParserRequestContext parser;
}
```

因为你不能把一个短期的对象如request、response和request context注入到`MyAction`这个singleton对象。然而，*在Webx中，这样做是可以的*！奥秘在于Request Contexts服务对上表所列的这些短期对象作了特殊的处理，使它们可以被注入到singleton对象中。事实上，被注入的只是一个“空壳”，真正的对象是在被访问到的时候才会从线程中取得的。

Webx鼓励应用程序使用singleton作用域的对象，不仅更简单，也更高效。经过上述技术处理以后，singleton对象访问request作用域对象的方法被大大简化了。

6.3. Pipeline服务

6.3.1. Pipeline工作原理

Pipeline的意思是管道，管道中有许多阀门（Valve），阀门可以控制水流的走向。在Webx中，pipeline的作用就是控制应用程序流程的走向。

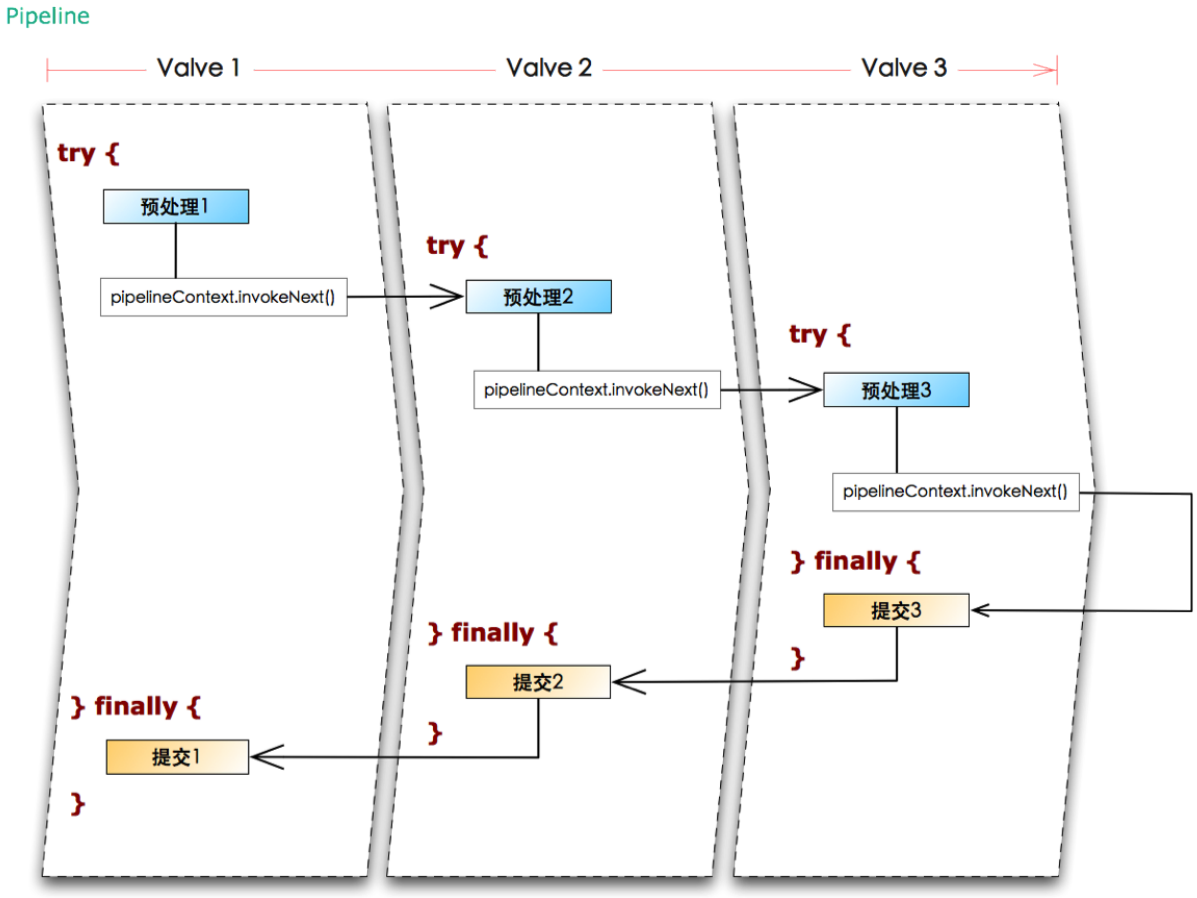


图 6.4. Pipeline和Valves

Pipeline的设计和filter非常相似，也是击鼓传花式的流程控制。但是有几点不同：

- Pipeline只能控制流程，不能改变request和response。
- Pipeline是轻量级组件，它甚至不依赖于WEB环境。Pipeline既可以在程序中直接装配，也可以由spring和schema来配置。

- Pipeline支持更复杂的流程结构，例如：子流程、条件分支、循环等。

6.3.2. Pipeline的用途

Pipeline可以说是Webx框架的核心功能之一。利用pipeline，你可以定制一个请求处理过程的每一步。

例 6.10. 一个典型的Webx应用的pipeline配置文件（**pipeline.xml**）

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans:beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:services="http://www.alibaba.com/schema/services"
    xmlns:pl-conditions="http://www.alibaba.com/schema/services/pipeline/conditions"
    xmlns:pl-valves="http://www.alibaba.com/schema/services/pipeline/valves"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="
        http://www.alibaba.com/schema/services
                http://localhost:8080/schema/services.xsd
        http://www.alibaba.com/schema/services/pipeline/conditions
                http://localhost:8080/schema/services-pipeline-conditions.xsd
        http://www.alibaba.com/schema/services/pipeline/valves
                http://localhost:8080/schema/services-pipeline-valves.xsd
        http://www.springframework.org/schema/beans
                http://localhost:8080/schema/www.springframework.org/schema/beans/spring-beans.xsd
    ">

    <services:pipeline xmlns="http://www.alibaba.com/schema/services/pipeline/valves">

        <!-- 初始化turbine rundata，并在pipelineContext中设置可能会用到的对象(如rundata、utils)，以便valve取得。
 -->
        <prepareForTurbine />

        <!-- 设置日志系统的上下文，支持把当前请求的详情打印在日志中。 -->
        <setLoggingContext />

        <!-- 分析URL，取得target。 -->
        <analyzeURL homepage="homepage" />

        <!-- 检查csrf token，防止csrf攻击和重复提交。 -->
        <checkCsrfToken />

        <loop>
            <choose>
                <when>
                    <!-- 执行带模板的screen，默认有layout。 -->
                    <pl-conditions:target-extension-condition extension="null, vm, jsp" />
                    <performAction />
                    <performTemplateScreen />
                    <renderTemplate />
                </when>
                <when>
                    <!-- 执行不带模板的screen，默认无layout。 -->
                    <pl-conditions:target-extension-condition extension="do" />
                    <performAction />
                    <performScreen />
                </when>
                <otherwise>
                    <!-- 将控制交还给servlet engine。 -->
                    <exit />
                </otherwise>
            </choose>

            <!-- 假如rundata.setRedirectTarget()被设置，则循环，否则退出循环。 -->
            <breakUnlessTargetRedirected />
        </loop>

    </services:pipeline>

</beans:beans>
```

6.3.3. Pipeline的使用

6.3.3.1. 创建一个valve

例 6.11. 一个简单的valve实现

```
public class MyValve implements Valve {
    public void invoke(PipelineContext pipelineContext) throws Exception {
        System.out.println("valve started.");

        pipelineContext.invokeNext(); // 调用后序valves

        System.out.println("valve ended.");
    }
}
```

配置（pipeline.xml）

```
<services:pipeline xmlns="http://www.alibaba.com/schema/services/pipeline/valves">
    ...
    <valve class="com.alibaba.myapp.pipeline.MyValve" />
    ...
</services:pipeline>
```

上面的代码和配置创建了一个基本的valve —— 事实上，它只是打印了一些消息，然后把控制权传递给后序的valves。

6.3.3.2. 执行一个pipeline

例 6.12. 在代码中执行pipeline

```
@Autowired
private Pipeline myPipeline;

public void invokePipeline() {
    PipelineInvocationHandle invocation = myPipeline.newInvocation();

    invocation.invoke();

    System.out.println(invocation.isFinished());
    System.out.println(invocation.isBroken());
}
```

从spring容器中取得一个pipeline对象以后（一般是通过注入取得），我们就可以执行它。上面代码中，`PipelineInvocationHandle`对象代表此次执行pipeline的状态。Pipeline执行结束以后，访问`invocation`对象就可以了解到pipeline的执行情况 —— 正常结束还是被中断?

Pipeline对象是线程安全的，可被所有线程所共享。但`PipelineInvocationHandle`对象不是线程安全的，每次执行pipeline时，均需要取得新的`invocation`对象。

6.3.3.3. 调用子流程

Pipeline支持子流程。事实上，子流程不过是另一个pipeline对象而已。

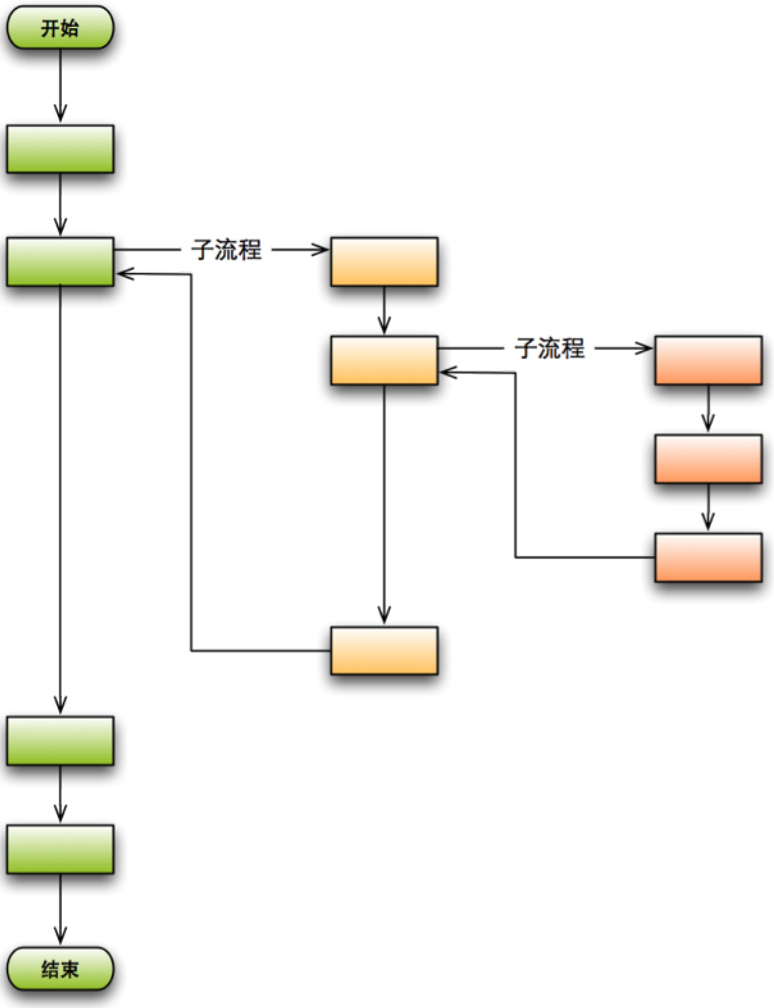


图 6.5. Pipeline和子流程

子流程是从valve中发起的。下面的Valve代码启动了一个子流程。

例 6.13. 在valve中发起一个子流程

```
public class MyNestableValve implements Valve {
    private Pipeline subPipeline;

    public void setSubPipeline(Pipeline subPipeline) {
        this.subPipeline = subPipeline;
    }

    public void invoke(PipelineContext pipelineContext) throws Exception {
        // 发起子流程，以当前流程的pipelineContext为参数
        PipelineInvocationHandle subInvocation = subPipeline.newInvocation(pipelineContext);

        subInvocation.invoke();

        System.out.println(subInvocation.isFinished());
        System.out.println(subInvocation.isBroken());

        pipelineContext.invokeNext(); // 别忘了调用后序的valves
    }
}
```

配置文件（pipeline.xml）

```
<services:pipeline xmlns="http://www.alibaba.com/schema/services/pipeline/valves">
    ...
    <valve class="com.alibaba.myapp.pipeline.MyNestableValve" p:subPipeline-ref="subPipeline" />
    ...
</services:pipeline>
```

6.3.3.4. 中断一个pipeline

Pipeline可以被中断。当有多级子pipeline时，你可以中断到任何一级pipeline。

例 6.14. 中断一个pipeline

```
pipelineContext.breakPipeline(0); // level=0，中断当前pipeline
pipelineContext.breakPipeline(1); // level=1，中断上一级pipeline

pipelineContext.breakPipeline("label"); // 中断到指定label的上级pipeline
// 以上调用相当于：
pipelineContext.breakPipeline(pipelineContext.findLabel("label"));

pipelineContext.breakPipeline(Pipeline.TOP_LABEL); // 终止所有pipelines
```

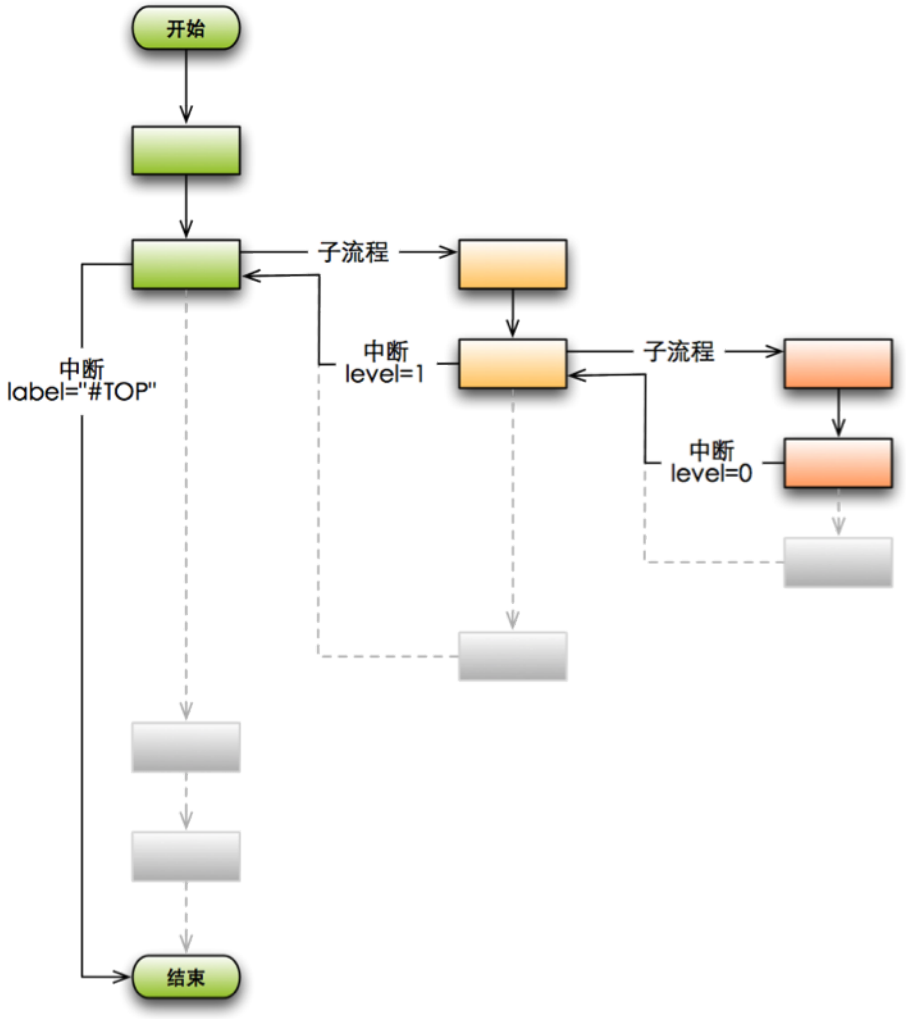


图 6.6. 中断一个pipeline

6.3.3.5. 条件分支、循环

条件分支和循环其实只不过是子流程的运用而已：

条件分支	根据一定的条件，来决定是否要执行子流程、执行哪一个子流程（多条件分支）。
循环	多次执行子流程。

下面的valve将子流程执行了至多10遍。如果子流程内部中断了流程，则循环终止。

例 6.15. 将子流程循环执行10次

```
public class Loop10 implements Valve {
    private Pipeline loopBody;

    public void setLoopBody(Pipeline loopBody) {
        this.loopBody = loopBody;
    }

    public void invoke(PipelineContext pipelineContext) throws Exception {
        PipelineInvocationHandle handle = loopBody.newInvocation(pipelineContext);

        for (int i = 0; i < 10 && !handle.isBroken(); i++) {
            handle.invoke();
        }

        pipelineContext.invokeNext();
    }
}
```

6.3.3.6. 存取pipeline的状态

当一个pipeline在运行时，你可以通过`PipelineContext`取得一些上下文信息：

例 6.16. 在valve中存取pipeline的状态

```
pipelineContext.index(); // 当前valve在pipeline中的序号
pipelineContext.level(); // 当前pipeline在所有子pipeline中的级别
pipelineContext.isBroken(); // 当前pipeline是否已经被中断
pipelineContext.isFinished(); // 当前pipeline的所有valves是否已经执行完

// 存取任意数据
pipelineContext.getAttribute(key);
pipelineContext.setAttribute(key, value);
```

6.3.3.7. 现成可用的valves

一般情况下，你并不需要写前面例子中的代码，因为Webx已经为你提供了一系列现成的valves来实现同样的功能。

无条件循环 - `<loop>`

例 6.17. 无条件循环

```
<services:pipeline>
    <loop loopCounterName="count" maxLoopCount="10"> ❶
        <valve />
        <break-if test="..." /> ❷
    </loop>
</services:pipeline>
```

❶ 定义循环变量`loopCounterName`，这个变量值将被保存在`PipelineContext`中，且可被其它的valve所访问。

❶ 定义`maxLoopCount=10`最大循环圈数，以避免循环失控。

❷ 无条件循环一定要和`<break>`、`<break-if>`或`<break-unless>`等valve相配合。

条件循环 - <while>

例 6.18. 条件循环

```
<services:pipeline>
    <while loopCounterName="count" test="count &lt;= 2"> ❶
        <valve />
    </while>

    <while maxLoopCount="10"> ❷
        <conditions:condition class="..." /> ❸
        <valve />
    </while>
</services:pipeline>
```

❶ 定义循环变量loopCounterName，这个变量值将被保存在PipelineContext中，且可被其它的valve所访问。

❶ 通过判断循环变量“count <= 2”，循环2次。

❷ 定义maxLoopCount=10，以避免循环失控。

❸ 可以自定义任意条件。

单条件分支 - <if>

例 6.19. 单条件分支

```
<services:pipeline>
    <if test="1 == 2"> ❶
        <valve />
    </if>

    <if>
        <conditions:condition class="..." /> ❷
        <valve />
    </if>
</services:pipeline>
```

❶ JEXL条件表达式。

❷ 自定义任意条件。

多条件分支 - <choose><when><otherwise>

例 6.20. 多条件分支

```
<services:pipeline>
    <choose>
        <when test="1 == 2"> ❶
            <valve />
        </when>
        <when> ❷
            <conditions:condition class="..." />
            <valve />
        </when>
        <otherwise> ❸
            <valve />
        </otherwise>
    </choose>
</services:pipeline>
```

❶ 条件分支1，用JEXL表达式来判断。

❷ 条件分支2，用任意条件判断。

❸ 分支3，当所有条件均不符合时，选择该分支。

无条件中断 - <break>

例 6.21. 无条件中断

```
<services:pipeline>
    <loop> ❶
        <valve />
        <break /> ❷
        <valve />
    </loop>

    <loop> ❸
        <valve />
        <loop>
            <break levels="1" /> ❹
        </loop>
        <valve />
    </loop>

    <loop label="MY_LOOP"> ❺
        <valve />
        <loop>
            <break toLabel="MY_LOOP" /> ❻
        </loop>
        <valve />
    </loop>
</services:pipeline>
```

❷ 无条件中止当前的pipeline（即loop循环❶）。

❹ 无条件中止上一层（`levels=1`）的pipeline（即loop循环❸）。

❻ 无条件中止指定label的pipeline（即loop循环❺）。

有条件中断 - <break-if>、<break-unless>

有条件中断是<break>和<if>的组合。

例 6.22. 有条件中断

```
<services:pipeline>
    <loop loopCounterName="count">
        <valve />
        <break-if test="count &gt; 2" /> ❶
        <valve />
    </loop>

    <loop label="MY_LOOP">
        <valve />
        <break-if toLabel="MY_LOOP"> ❷
            <conditions:condition class="..." /> ❸
        </break-if>
        <valve />
    </loop>

    <loop loopCounterName="count">
        <valve />
        <break-unless test="count &lt;= 2" /> ❹
        <valve />
    </loop>
</services:pipeline>
```

❶ 当count>2时中断。

❷ <break-if>和<break-unless>均支持和<break>类似的其它选项：levels和toLabel。

❸ 和<if>类似，也支持任意condition。

❹ <break-unless>和<break-if>的条件相反：*除非*count<=2，否则中断。

无条件退出整个pipeline - <exit>

退出整个pipeline，意思是结束所有的嵌套层次。

例 6.23. 无条件退出整个pipeline

```
<services:pipeline>
    <loop>
        <valve />
        <loop>
            <exit />
        </loop>
        <valve />
    </loop>
</services:pipeline>
```

对于Webx而言，<exit>还有一层特殊的含义：放弃WebxFrameworkFilter的控制权，把它交还给servlet engine。以URL http://localhost:8081/myapp/myimage.jpg为例，把控制权交还给servlet engine，意味着让servlet engine去显示myapp应用目录下的静态图片：myimage.jpg。

异常捕获和finally处理 - <try-catch-finally>

类似Java中的try/catch/finally结构。

例 6.24. 异常捕获和finally处理

```
<services:pipeline>
    <try-catch-finally>
        <try>
            <valve />
        </try>
        <catch exceptionName="myexception"> ❶
            <valve />
        </catch>
        <finally>
            <valve />
        </finally>
    </try-catch-finally>
</services:pipeline>
```

❶ <catch>标签可以将捕获的异常以指定名称保存在PipelineContext中，以便其它valve取得。

创建子流程 - <sub-pipeline>

单纯使用这个valve，对执行结果不会有任何影响。但可用来对较长的pipeline进行分段管理。

例 6.25. 创建子流程

```
<services:pipeline>
    <valve />
    <sub-pipeline label="mylabel">
        <valve />
    </sub-pipeline>
    <valve />
</services:pipeline>
```

6.3.3.8. 条件

在前文所述的各种条件valve（例如<if>、<when>、<while>、<break-if>、<break-unless>等）中，都用到一个共同的对象：condition。Condition是一个简单的接口。

例 6.26. Condition接口

```
public interface Condition {
    /**
     * 如满足条件，则返回<code>true</code>。
     */
    boolean isSatisfied(PipelineStates pipelineStates);
}
```

为了方便起见，Webx默认提供了一个JexlCondtion。

例 6.27. 使用JexlCondition

```
<if>
    <conditions:jexl-condition expr="loopCount == 2" />
    <break />
</if>
```

以上配置可以简化为：

```
<if test="loopCount == 2">
    <break />
</if>
```

JEXL表达式是Apache的一个小项目，表达式语法详见：http://commons.apache.org/jexl/reference/syntax.html。在JEXL表达式中，你可以使用`pipelineContext.getAttribute()`所能取得的所有状态值。例如，loop循环时，如果你设置了`loopCounterName`，那么循环计数器就可以被JEXL表达式所访问。

除此之外，Webx还提供了三个组合式的条件。

`<all-of>`

要求所有条件均满足，相当于Java中的&&操作符。

例 6.28. 组合式的条件：`<all-of>`

```
<all-of>
    <condition1 />
    <condition2 />
    <condition3 />
</all-of>
```

`<any-of>`

只要求任一条件满足，相当于Java中的||操作符。

例 6.29. 组合式的条件：`<any-of>`

```
<any-of>
    <condition1 />
    <condition2 />
    <condition3 />
</any-of>
```

`<none-of>`

要求所有条件均不满足，相当于Java中的!操作符。

例 6.30. 组合式的条件：`<none-of>`

```
<none-of>
    <condition1 />
    <condition2 />
    <condition3 />
</none-of>
```

这三个组合式条件可以互相组合，以构成任意复杂的条件判断语句。

6.4. 本章总结

Request Contexts和Pipeline是Webx框架中的两个核心服务。它们分别从两个方面实现了原本需要由Filter来实现的功能 —— Request Contexts提供了包装和修改request/response的机制，而pipeline则提供了流程控制的能力。Request contexts和pipeline组合起来的功能比servlet filter机制更加强大。因为它们是基于Spring的轻量组件，其性能、配置的方便性、扩展性都优于filter。

当然，Request Contexts和Pipeline并不想取代filter。在好几种场合，filter仍然是唯一的选择：

- 如果你既想要修改request/response，又想要控制流程；
- 如果你希望独立于任何框架。

但在你接到一个需求，正打算用filter来实现之前，请考虑一下，是否可以采用Webx所提供的这两种机制来取代。倘若可行，必然会带来更多的好处。

第 7 章 Request Contexts功能指南

在第 6 章 *Filter、Request Contexts和Pipeline*中，我们已经介绍了Request Contexts服务的作用和原理。本章我们将介绍除了session机制以外，每一个可用的Request Context的功能和用法。由于Session机制比较复杂，所以我们另辟单独的一章（第 8 章 *Request Context之Session指南*）来解释它。

本章涉及的内容包括：

名称	接口	功能
<basic>	BasicRequestContext	提供基础安全特性，例如：过滤response headers、cookies，限制cookie的大小等。
<set-locale>	SetLocaleRequestContext	设置locale区域和charset字符集编码。
<parser>	ParserRequestContext	解析参数，支持multipart/form-data（即上传文件请求）。

名称	接口	功能
<buffered>	BufferedRequestContext	缓存response中的内容。
<lazy-commit>	LazyCommitRequestContext	延迟提交response。
<rewrite>	RewriteRequestContext	重写请求的URL和参数。

7.1. <basic> - 提供基础特性

7.1.1. 拦截器接口

BasicRequestContext提供了一组interceptors拦截器接口，通过它们，你可以拦截并干预一些事件。

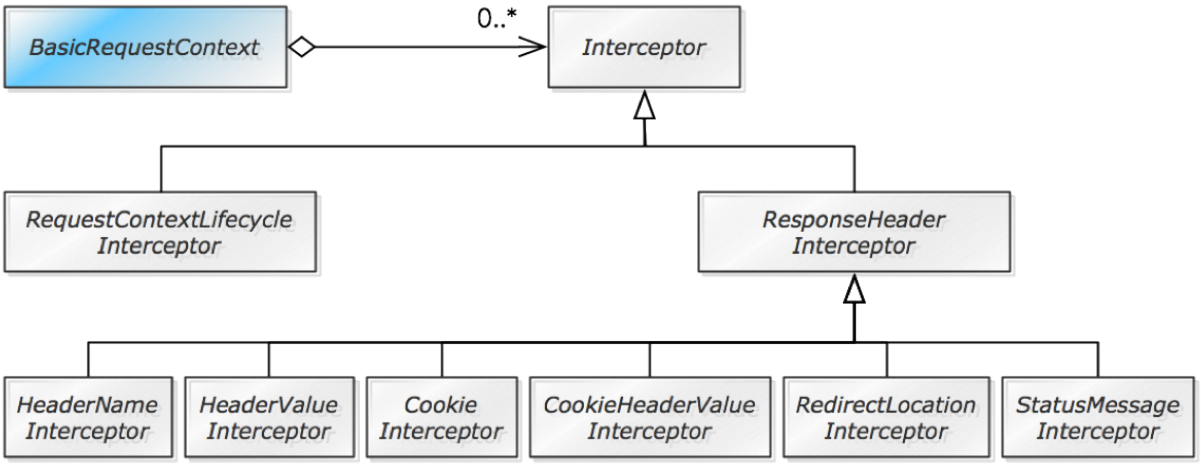


图 7.1. BasicRequestContext所提供的拦截器

你可以在<basic>中指定上图所示的任何一个Interceptor接口，以便干预特定的事件：

表 7.1. BasicRequestContext所提供的拦截器

拦载器接口	说明
RequestContextLifecycleInterceptor	拦截“预处理（prepare）”和“提交（commit）”事件。
ResponseHeaderInterceptor	拦截所有对response header的修改。
➥ HeaderNameInterceptor	拦截所有对header的修改、添加操作。可修改header name，或拒绝对header的修改。
➥ HeaderValueInterceptor	拦截所有对header的修改、添加操作。可修改header value，或拒绝对header的修改。
➥ CookieInterceptor	拦截所有对cookie的添加操作。可修改或拒绝cookie对象。需要注意的是，有两种方法可以添加cookie：通过cookie对象，或者直接写response header。对于后者，需要使用CookieHeaderValueInterceptor才能拦截得到。
➥ CookieHeaderValueInterceptor	拦截所有通过添加header来创建cookie的操作。可修改或拒绝该cookie。
➥ RedirectLocaitonInterceptor	拦截所有外部重定向的操作。可修改或拒绝重定向URL。
➥ StatusMessageInterceptor	拦截所有设置status message的操作。可以修改或拒绝该message。

通过下面的配置，就可以指定任意多个interceptor的实现。

例 7.1. 配置interceptors（**/WEB-INF/webx.xml**）

```
<services:request-contexts xmlns="http://www.alibaba.com/schema/services/request-contexts">
    <basic>
        <request-contexts:interceptors
            xmlns="http://www.alibaba.com/schema/services/request-contexts/basic/interceptors">
            <interceptor class="...Interceptor1" />
            <interceptor class="...Interceptor2" />
        </request-contexts:interceptors>
    </basic>
    ...
</services:request-contexts>
```

7.1.2. 默认拦截器

即使你不加说明，`BasicRequestContext`也总是会启用一个默认的interceptor实现：`ResponseHeaderSecurityFilter`。这个类实现了下列功能：

- 避免header name和value中出现CRLF字符 —— 在header中嵌入CRLF（回车换行）字符是一种常见的攻击手段。攻击者嵌入CRLF以后，使服务器对HTTP请求发生错误判断，从而执行攻击者的恶意代码。事实上，现在的servlet引擎如tomcat已经可以防御这种攻击。但作为框架，并不能依赖于特定的servlet引擎，所以加上这个额外的安全检查，确保万无一失。
- 将status message用HTML entity编码重写 —— 通常status message会被显示在HTML页面中。攻击者可以利用这一点在页面中嵌入恶意代码。将status message以HTML entity编码重写以后，就可以避免这个问题。
- 限制cookie的总大小 —— 过大的cookie可能使WEB服务器拒绝响应请求。攻击者同样可以利用这一点使用户无法正常访问网站。限制cookie的总大小可以部分地解决这种危机。

如果需要，你可以对`ResponseHeaderSecurityFilter`指定一些参数。

例 7.2. 配置**ResponseHeaderSecurityFilter**（**/WEB-INF/webx.xml**）

```
<request-contexts:interceptors
            xmlns="http://www.alibaba.com/schema/services/request-contexts/basic/interceptors">
    <interceptor class="...Interceptor1" />
    <interceptor class="...Interceptor2" />
    <response-header-security-filter maxSetCookieSize="5K" />
</request-contexts:interceptors>
```

7.2. **<set-locale>** -设置locale区域和charset字符集编码

区域和编码问题（尤其是后者）是每个WEB应用都必须处理好的基本问题。它虽然本身并不复杂，但是在现实开发中，由于涉及面很广，一旦发生问题（例如乱码）经常让人手足无措。`<set-locale>`提供了一个机制，确保Web应用能够设置正确的区域和编码。

7.2.1. Locale基础

Locale是国际化的基础。

一个locale的格式是：`language_country_variant`，例如：`zh_CN`、`zh_TW`、`en_US`、`es_ES_Traditional_WIN`等。

Java和框架根据不同的locale，可以取得不同的文本、对象。下面的Java代码根据不同的locale，取得不同语言版本的文字：

例 7.3. 利用ResourceBundle和locale取得国际化字符

```
Locale.setDefault(Locale.US);

String s1 = getResourceBundle(Locale.CHINA).getString("happy"); // 快乐
String s2 = getResourceBundle(Locale.TAIWAN).getString("happy"); // 快樂
String s3 = getResourceBundle(Locale.US).getString("happy"); // happy
...
ResourceBundle getResourceBundle(Locale locale) {
    return ResourceBundle.getBundle("ApplicationResources", locale);
}
```

其中所用到的ResourceBundle文件定义如下：

ApplicationResources.properties	happy = happy
ApplicationResources_zh_CN.properties	happy = \u5FEB\u4E50
ApplicationResources_zh_TW.properties	happy = \u5FEB\u6A02

7.2.2. Charset编码基础

Charset全称Character Encoding或字符集编码。Charset是将字符（characters）转换成字节（bytes）或者将字节转换成字符的算法。Java内部采用unicode来表示一个字符。将unicode字符转换成字节的过程，称为“*编码*”；将字节恢复成unicode字符的过程，称为“*解码*”。

浏览器发送给WEB应用的request参数，是以字节流的方式来表示的。Request参数必须经过解码才能被Java程序所解读。用来解码request参数的charset被称为“*输入字符集编码（Input Charset）*”；

WEB应用返回给浏览器的response响应内容必须编码成字节流，才能被浏览器或客户端解读。用来编码response内容的charset被称为“*输出字符集编码（Output Charset）*”。

一般情况下，input charset和output charset是相同的。因为浏览器发送表单数据时，总是采用当前页面的charset来编码的。例如，有一个表单页面，它的“`contentType=text/html; charset=GBK`”，那么用户填完全表单并提交时，浏览器会以GBK来编码用户所输入的表单数据。如果input charset和output charset不相同，服务器就不能正确解码浏览器根据output charset所发回给WEB应用的表单数据。

然而有一些例外情况下面，输入和输出的charset可能会不同：

- 通过Java Script发送的表单，总是用UTF-8编码的。这意味着你必须用UTF-8作为input charset方能正确解码参数。这样，除非output charset也是UTF-8，否则两者就是不同的。
- 应用间互相用HTTP访问时，可能采用不同的编码。例如，应用A以UTF-8访问应用B，而应用B是以GBK作为input/output charset的。此时会产生参数解码的错误。
- 直接在浏览器地址栏里输入包含参数的URL，根据不同的浏览器和操作系统的设置，会有不同的结果：
  - 例如，中文Windows中，无论ie还是firefox，经试验，默认都以GBK来编码参数。IE对直接输入的参数，连URL encoding也没做。

- 而在mac系统中，无论safari还是firefox，经试验，默认都是以UTF-8来编码参数。

框架必须要能够应付上面各种不确定的charset编码。

7.2.3. Locale和charset的关系

Locale和charset是相对独立的两个参数，但是又有一定的关系。

Locale决定了要显示的文字的语言，而charset则将这种语言的文字编码成bytes或从bytes解码成文字。因此，charset必须能够涵盖locale所代表的语言文字，如果不能，则可能出现乱码。下表列举了一些locale和charset的组合：

表 7.2. Locale和Charset的关系

Locale	英文字符集	中文字符集			全字符集	
	ISO-8859-1	GB2312	Big5	GBK	GB18030	UTF-8
en_US（美国英文）	✓	✓	✓	✓	✓	✓
zh_CN（简体中文）		✓		✓	✓	✓
zh_TW、zh_HK（台湾中文、香港中文）			✓	✓	✓	✓

在所有charset中，有几个“全能”编码：

UTF-8

涵盖了unicode中的所有字符。然而用UTF-8来编码中文为主的页面时，每个中文会占用3个字节。建议以非中文为主的页面采用UTF-8编码。

GB18030

中文国际标准，和UTF-8一样，涵盖了unicode中的所有字符。用GB18030来编码中文为主的页面时有一定优势，因为绝大多数常用中文仅占用2个字节，比UTF-8短1/3。然而GB18030在非中文的操作系统中，有可能不能识别，其通用性不如UTF-8好。因此仅建议以中文为主的页面采用GB18030编码。

GBK

严格说，GBK不是全能编码（例如对很多西欧字符就支持不好），也不是国际标准。但它支持的字符数量接近于GB18030。

7.2.4. 设置locale和charset

在Servlet API中，以下API是和locale和charset有关的。

表 7.3. 和locale、charset相关的servlet API

HttpServletRequest		
.getCharacterEncoding()	读取输入编码	
.setCharacterEncoding(charset)	设置输入编码	• 必须在第一次调用request.getParameter()和request.getParameterMap()前设置，否则无效。 • 如果不设置，则默认以ISO-8859-1来解码参数。

HttpServletRequest		
		• 一般只影响POST请求参数的解码，但这里有一些复杂性，参见第 7.3 节 “<parser> - 解析参数” 。
.getLocale()	取得Accept-Language中浏览器首选的locale	
.getLocales()	取得所有Accept-Language中所指定的locales	
HttpServletResponse		
.getCharacterEncoding()	取得输出编码	
.setCharacterEncoding(charset)	设置输出编码	• Since Servlet 2.4
.getContentType()	取得content type	• Since Servlet 2.4
.setContentType(contentType)	设置content type	• Content type中可能包含charset定义，例如：`text/html; charset=GBK`
.getLocale()	取得输出locale	
.setLocale(locale)	设置输出locale	• 必须在response被commit之前调用，否则无效。 • 它同时也会设置charset，除非content type已经被设置过，并用包含了charset的定义。

设置locale和charset是一件看起来容易，做起来不容易的事：

- 输入编码必须*在第一个读取request参数的调用之前*设置好，否则就无效。只有把<set-locale>作为Request Contexts服务的一环，才有可能确保读取request参数之前，设置好输入编码。
- 在Servlet 2.3之前，设置输出参数的唯一方法，是通过设置带有charset定义的content type。这一点在Servlet 2.4以后得到改进，添加了独立的设置输出编码的方法。<set-locale>弥补了Servlet 2.3和Servlet 2.4之间的差异，使WEB应用在所有的环境下，都可以独立设置content type和charset。

7.2.5. 使用方法

7.2.5.1. 使用默认值

例 7.4. 设置默认的locale和charset

```
<services:request-contexts xmlns="http://www.alibaba.com/schema/services/request-contexts">
    <set-locale defaultLocale="zh_CN" defaultCharset="GB18030" />
    ...
</services:request-contexts>
```

上面的配置，将WEB应用的输入charset、输出charset均设置成GB18030，将输出locale设置成zh_CN。

7.2.5.2. 临时覆盖默认的charset

前面讲到在一些情况下面，服务器所收到的参数（表单数据）不是用应用默认的charset来编码的。例如Java Script总是以UTF-8来提交表单；系统间通过HTTP协议通信；或者用户直接在浏览器地址栏中输入参数。

如何应付这些不确定的charset呢？<set-locale>提供的方法是，在URL中指定输入编码，并覆盖默认值。

假设当前应用的默认值是defaultLocale=zh_CN、defaultCharset=GB18030，那么下面的请求将使用默认的GB18030来解码参数，并用默认的GB18030来输出页面：

```
http://localhost:8081/myapp/myform
```

假如你希望改用UTF-8来解码参数，那么可以使用下面的URL来覆盖默认值：

例 7.5. 在URL中覆盖默认的input charset

```
http://localhost:8081/myapp/myform?_input_charset=UTF-8
```

这样，Webx将采用UTF-8来解码参数，但*仍然使用默认的GB18030来输出页面*。

需要注意的是，对于POST请求，你必须把_input_charset这个特殊的参数写在URL中，而不能写成普通的表单字段，例如：

例 7.6. 在POST表单中覆盖默认的input charset

```
<form action="http://localhost:8081/myapp/myform?_input_charset=UTF-8" method="POST"> ❶
    <input type="hidden" name="param1" value="value1"/>
    <input type="hidden" name="param2" value="value2"/>
</form>
```

❶ 必须把_input_charset这个特殊的参数写在URL中，即便是POST类型的表单。

在写AJAX Java Script代码时，也要注意：

例 7.7. 在AJAX代码中覆盖默认的input charset

```
var xhreq = new XMLHttpRequest();
xhreq.open("post", "/myapp/myform?_input_charset=UTF-8", true); ❶
...
xhreq.send("a=1&b=2");
```

❶ 必须把_input_charset这个特殊的参数写在URL中。

此外，<set-locale>也提供了临时覆盖输出编码的方法：

例 7.8. 在URL中覆盖默认的output charset

```
http://localhost:8081/myapp/myform?_output_charset=UTF-8
```

临时覆盖的输入、输出编码只会影响当前请求，它不会被记住。当一个不带有覆盖参数的请求进来时，将仍然按照默认值来设置输入、输出编码。

7.2.5.3. 持久覆盖默认的locale和charset

还有一种需求，就是多语言网页的支持。用户可以选择自己的语言：简体中文、繁体中文等。一旦用户作出选择，那么后续的网页将全部以用户所选择的语言和编码来显示。<set-

locale>直接支持这个功能。只要你按下面的URL访问页面，用户的语言和编码即被切换成简体中文和UTF-8编码。

例 7.9. 持久覆盖默认的locale和charset

```
http://localhost:8081/myapp?_lang=zh_CN:UTF-8
```

参数值_lang=zh_CN:UTF-8将被保存在session中，后续的请求不需要再次指定_lang参数。用户所作出的选择将一直持续在整个session中，直到session被作废。

需要说明的是，假如我们采用了<session> request context来取代原来的session机制，那么该参数实际的保存位置将取决于session框架的设置 —— 例如：你可以把参数值保存在某个cookie中。然而，<set-locale>并不需要关心于session的实现细节或是用来保存参数的cookie的细节。

7.2.5.4. <set-locale>的影响力

<set-locale>所设置的*输出locale*和*输出charset*值将会被保存在当前线程中，从而对整个线程产生影响。

表 7.4. 被<set-locale>影响的API

API	说明
LocaleUtil.getContext().getLocale()	可以通过这两个方法取得当前线程的输出locale和charset。Webx框架中凡是要用到默认locale和charset的地方，都会从这里去取得值。
LocaleUtil.getContext().getCharset()	
StringEscapeUtil.escapeURL()	Webx调用这两个方法进行URL编码、解码时，不需要指定charset（不同于JDK的URLEncoder/URLDecoder）。这两个函数将从LocaleUtil.getContext().getCharset()中取得当前线程的输出charset。
StringEscapeUtil.unescapeURL()	
TemplateService	TemplateService如果指定了searchLocalizedTemplates=true参数，那么它会利用当前线程的locale来搜索本地化的模板，例如： screen/myTemplate_zh_CN.vm

7.2.5.5. <set-locale>的配置参数

例 7.10. <set-locale>的配置参数

```
<set-locale defaultLocale="..."
            defaultCharset="..."
            inputCharsetParam="_input_charset"
            outputCharsetParam="_output_charset"
            paramKey="_lang"
            sessionKey="_lang" />
```

表 7.5. <set-locale>配置参数说明

参数名	说明
defaultLocale	默认locale。
defaultCharset	默认charset。
inputCharsetParam	用来临时改变输入charset的参数名，支持多个名称，以“|”分隔，例如“_input_charset|ie”。默认值为“_input_charset”。
outputCharsetParam	用来临时改变输出charset的参数名，支持多个名称，以“|”分隔，例如“_output_charset|oe”。默认为“_output_charset”。

参数名	说明
paramKey	用来持久改变输出locale和charset的参数名，默认为“_lang”。
sessionKey	用来在session中保存用户所选择的locale和charset的key，默认为“_lang”。

7.3. <parser> - 解析参数

7.3.1. 基本使用方法

7.3.1.1. 基本配置

例 7.11. <parser>基本配置

```
<services:request-contexts xmlns="http://www.alibaba.com/schema/services/request-contexts">
    <parser />
    ...
</services:request-contexts>

<services:upload sizeMax="5M" fileSizeMax="2M" />
```

绝大多数情况，你只需要上面的配置就足够了 —— <parser>会自动解析所有类型的请求，包括：

- GET请求
- 普通的POST请求（Content Type：application/x-www-form-urlencoded）
- 可上传文件的POST请求（Content Type：multipart/form-data）

7.3.1.2. 通过HttpServletRequest接口访问参数

<parser>对于大部分应用是透明的。也就是说，你不需要知道<parser>的存在，就可以访问所有的参数，包括访问multipart/form-data请求的参数。

例 7.12. 通过HttpServletRequest接口访问参数

```
@Autowired
HttpServletRequest request;

...
String s = request.getParameter("myparam");
```

7.3.1.3. 通过ParserRequestContext接口访问参数

你也可以选择使用ParserRequestContext接口。

例 7.13. 通过ParserRequestContext接口访问参数

```
@Autowired
ParserRequestContext parser;

...
String s = parser.getParameters().getString("myparam");
```

和HttpServletRequest接口相比，ParserRequestContext提供了如下便利：

直接取得指定类型的参数，例如：直接取得int、boolean值等。

例 7.14. 直接取得指定类型的参数

```
// myparam=true, myparam=false
parser.getParameters().getBoolean("myparam");

// myparam=123
parser.getParameters().getInt("myparam");
```

如果参数值未提供，或者值为空，则返回指定默认值。

例 7.15. 取得参数的默认值

```
parser.getParameters().getBoolean("myparam", false);
parser.getParameters().getString("myparam", "no_value");
parser.getParameters().getInt("myparam", -1);
```

取得上传文件的FileItem对象（这是Apache Jakarta 项目commons-fileupload所定义的接口）。

例 7.16. 取得FileItem上传文件

```
FileItem fileItem = parser.getParameters().getFileItem("myfile");
FileItem[] fileItems = parser.getParameters().getFileItems("myfile");
```

ParserRequestContext还提供了比较方便的访问cookie值的方法。

例 7.17. 访问cookie值

```
parser.getCookies().getString("mycookie");
```

7.3.2. 上传文件

用于上传文件的请求是一种叫作multipart/form-data的特殊请求，它的格式类似于富文本电子邮件的样子。下面HTML创建了一个支持上传文件的表单：

例 7.18. 创建multipart/form-data表单

```
<form action="..." method="post" enctype="multipart/form-data">
    <input type="file" name="myfile" value="" />
    ...
</form>
```

提示：不是只有需要上传文件时，才可以用multipart/form-data表单。*假如你的表单中包含富文本字段（即字段的内容是以 HTML或类似的技术描述的），特别是当字段的内容比较长的时候，用multipart/form-data比用普通的表单更高效，生成的HTTP请求也更短。*

只要upload服务存在，那么<parser>就可以解析multipart/form-data（即上传文件）的请求。Upload服务扩展于Apache Jakarta的一个项目：commons-fileupload。

7.3.2.1. 配置Upload服务

例 7.19. Upload服务的配置参数

```
<services:upload sizeMax="5M"
                 fileSizeMax="2M"
                 repository="/tmp"
                 sizeThreshold="10K"
                 keepFormFieldInMemory="true" />
```

各参数的说明如下：

表 7.6. Upload服务配置参数说明

参数名称	说明
sizeMax	HTTP请求的最大尺寸（字节，支持K/M/G），超过此尺寸的请求将被抛弃。值-1表示没有限制。
fileSizeMax	单个文件允许的最大尺寸（字节，支持K/M/G），超过此尺寸的文件将被抛弃。值-1表示没有限制。
repository	暂存上传文件的目录。 注意，这个目录是用Spring ResourceLoader装载的，而不是一个物理路径。关于ResourceLoader，详见ResourceLoading服务的文档。
sizeThreshold	将文件放在内存中的阈值（字节，支持K/M/G），小于此值的文件被保存在内存中。
keepFormFieldInMemory	是否将普通的form field保持在内存里？ 默认为false，但当sizeThreshold为0时，默认为true。

注意

当上传文件的请求的总尺寸超过sizeMax的值时，整个请求将被抛弃 —— 这意味着你不可能读到请求中的其它任何参数。而当某个上传文件的尺寸超出fileSizeMax的限制，但请求的总尺寸仍然在sizeMax的范围内时，只有超出该尺寸的单个上传文件被抛弃，而你还是可以读到其余的参数。

假如有多个upload服务（当然这种情况极少），你也可以明确指定<parser>使用哪个upload服务：

例 7.20. 明确指定upload服务

```
<parser uploadServiceRef="myUpload" />
```

7.3.2.2. 手工解析上传请求

在默认情况下，当<parser>收到一个上传文件的请求时，会立即解析并取得所有的参数和文件。然而你可以延迟这个过程，在需要的时候，再手工解析上传请求。

例 7.21. 手工解析upload请求

首先，你需要关闭自动上传

```
<parser autoUpload="false">
```

可选参数autoUpload默认值为true，当你把它改成false时，就可以实现延迟手工解析请求。在你需要解析请求时，只需要调用下面的语句即可：

```
parser.getParameters().parseUpload();
```

手工调用parseUpload可以指定和默认不同的参数：

```
UploadParameters params = new UploadParameters();

params.applyDefaultValues();
params.setSizeMax(new HumanReadableSize("10M"));
params.setFileSizeMax(new HumanReadableSize("1M"));
params.setRepository(new File("mydir"));

parser.getParameters().parseUpload(params);
```

7.3.3. 高级选项

7.3.3.1. 参数名称大小写转换

在默认情况下，假设有一个参数名为：myProductId，那么你可以使用下列*任意一种方法*来访问到它：

例 7.22. 取得参数myProductId的值的方法

```
request.getParameter("MyProductId");
request.getParameter("myProductId");
request.getParameter("my_product_id");
request.getParameter("MY_PRODUCT_ID");
request.getParameter("MY_productID");
```

假如你不希望具备这种灵活性，则需要修改配置以关闭大小写转换功能：

例 7.23. 关闭大小写转换功能

```
<parser caseFolding="none">
```

7.3.3.2. 参数值去空白

在默认情况下，假设有一个参数：id=" 123 "（两端有空白字符），那么<parser>会把它转化成"123"（两端没有空白字符）。 假如你不希望<parser>做这件事，则需要修改配置：

例 7.24. 关闭参数值去空白功能

```
<parser trimming="false">
```

这样，所有的参数值将会保持原状，不会被去除空白。

7.3.3.3. 参数值entity解码

浏览器在提交表单时，如果发现被提交的字符不能以当前的charset来编码，浏览器就会把该字符转换成&#unicode;这样的形式。例如，假设一个表单页面的content type为：*text/html; charset=ISO-8859-1*。在这个页面的输入框中输入汉字“你好”，然后提交。你会发现，提交的汉字变成了这个样子：param=*"你好"*。

在默认情况下，<parser>会对上述参数进行entity解码，使之恢复成“你好”。但是，其它的entity如“<”、“&”等并不会被转换。 如果你不希望<parser>还原上述内容，则需要修改配置：

例 7.25. 关闭参数值entity解码功能

```
<parser unescapeParameters="false">
```

7.3.3.4. 取得任意类型的参数值

前面提到，ParserRequestContext支持直接取得boolean、int等类型的参数值。事实上，它还支持取得任意类型的参数值 —— 只要Spring中有相应的PropertyEditor支持即可。

假设MyEnum是一个enum类型，这是Spring原生支持的一种类型。你可以用下面的代码来取得它：

例 7.26. 将参数值转换成enum类型

```
MyEnum myEnum = params.getObjectOfType("myparam", MyEnum.class);
```

但是，下面的语句就不是那么顺利了 —— 因为Spring不知道怎么把一个参数值，例如：“1975-12-15”，转换成java.util.Date类型。

例 7.27. 将参数值转换成java.util.Date类型

```
Date birthday = params.getObjectOfType("birthday", Date.class);
```

好在<parser>提供了一种扩展机制，可以添加新的类型转换机制。对于Date类型，你只需要添加下面的配置，就可以被支持了。

```
<parser>
    <property-editor-registrar
        class="com.alibaba.citrus.service.configuration.support.CustomDateRegistrar"
        p:format="yyyy-MM-dd" p:locale="zh_CN" p:timeZone="GMT+8" /> ❶
</parser>
```

❶ PropertyEditorRegistrar是Spring提供的一种类型注册机制，其细节详见Spring的文档。

另一个问题是，如果类型转换失败怎么办？<parser>支持两种方法。默认情况下，类型转换失败会“保持安静”（不抛异常），然后返回默认值。但你也可以选择让类型转换失败的异常被抛出来，以便应用程序处理。

例 7.28. 设置“非安静”模式：当类型转换失败时，抛出异常

```
<parser converterQuiet="false">
```

程序里这样写：

```
MyEnum myEnum = null;

try {
    myEnum = params.getObjectOfType("myparam", MyEnum.class);
} catch (TypeMismatchException e) {
    ...
}
```

7.3.3.5. 解析GET请求的参数

GET请求是最简单的请求方式。它的参数以URL编码的方式包含在URL中。当你在浏览器地址栏中敲入“http://localhost:8081/user/login.htm?name=%E5%90%8D%E5%AD%97&password=password”这样一个址址的时候，浏览器就会向localhost:8081服务器出如下HTTP请求：

```
GET /user/login.htm?name=%E5%90%8D%E5%AD%97&password=password HTTP/1.1
Host: localhost:8081
```

GET请求中的参数是以application/x-www-form-urlencoded方式和特定的charset编码的。假如用来编码URL参数的charset与应用的默认charset不同，那么你必须通过特殊的参数来指定charset（参见第 7.2 节 “<set-locale> -设置locale区域和charset字符集编码”）：

```
GET /user/login.htm?_input_charset=UTF-8&name=%E5%90%8D%E5%AD%97&password=password HTTP/1.1
```

可是，上面的请求在不同的Servlet引擎中，会产生不确定的结果。这是怎么回事呢?

原来，尽管<set-locale>会调用request.setCharacterEncoding(charset)这个方法来设置input charset编码，然而根据Servlet API的规范，这个设定只能对request content生效，而不对URL生效。换句话说，request.setCharacterEncoding(charset)方法只能用来解析POST请求的参数，而不是GET请求的参数。

那么，应该怎样处理GET请求的参数呢？根据URL规范，URL中非US-ASCII的字符必须进行基于UTF-8的URL编码。然而实际上，从浏览器到服务器，没有人完全遵守这些规范，于是便造成了一些混乱。目前应用服务器端，我们所遇到的，有下面几种不同的解码方案：

表 7.7. 服务器对参数进行解码的逻辑

服务器	解码的逻辑
Tomcat 4	• 根据request.setCharacterEncoding(charset)所设置的值来解码GET参数； • 如果未特别指定charset，则默认采用ISO-8859-1来解码参数。
Tomcat 5及更新版 以及搭载Tomcat 5以上版本的JBoss	• 如果Tomcat配置文件conf/server.xml中设置了：<Connector useBodyEncodingForURI="true">那么根据request.setCharacterEncoding(charset)所设置的值来解码GET参数。 • 如未设置useBodyEncodingForURI，或其值为false，则根据conf/server.xml中的配置<Connector URIEncoding="xxx">所指定的编码，来解码GET请求的参数。 • 如未配置URIEncoding，默认采用ISO-8859-1。
Jetty Server	• Jetty总是以UTF-8来解码GET请求的参数。

综上所述，所有的应用服务器对于POST请求的参数的处理方法是没有差别的，然而对于GET请求的参数处理方法各有不同。

如果不加任何特别的设置，Tomcat最新版是以ISO-8859-1来解码GET请求的参数，而Jetty却是以UTF-8来解码的。因此，无论你以哪一种charset来编码GET请求的参数，都不可能在所有服务器上取得相同的结果 —— 除非修改服务器的配置，但这是一件既麻烦又容易出错的事情。为了使应用程序对服务器的配置依赖较少，且可以灵活地处理GET请求的解码，<parser>对GET请求进行了手工解码，从而解决了应用服务器解码的不确定性。

<parser>完全解决了上面的问题。*依据默认值，<parser>会以<set-locale>中设定的input charset为准，来解码所有类型的请求*，包括GET和POST请求，以及multipart/form-data（上传文件）类型的请求。

然而<parser>仍保留了一些可选方案，以备不时之需。

保留Servlet引擎的解码机制

例 7.29. 使用Servlet引擎原来的解码机制

```
<parser useServletEngineParser="true" />
```

这个选项在用HttpUnit进行单元测试时非常有用。因为HttpUnit单元测试工具并没有完全遵循Servlet API的规范 —— 目前版本的HttpUnit不能正确取得query string，从而导致<parser>解析GET参数错误。

使用固定的charset来解码GET请求

例 7.30. 使用固定的charset来解码GET请求

```
<parser URIEncoding="UTF-8" useBodyEncodingForURI="false" />
```

上面的配置强制所有的GET请求均使用UTF-8作为固定的charset编码。这段逻辑和tomcat的完全相同，但你却不需要去修改tomcat的conf/server.xml就可以实现上面的逻辑。 事实上，使用固定的charset来解码GET请求的参数是符合Servlet API规范以及URL的规范的。而根据情况设置charset是一种对现实的妥协。然而*你有选择的自由 —— 无论你选择何种风格，<parser>都支持你*。

7.3.3.6. 过滤参数

出于安全的考虑，<parser>还支持对输入参数进行过滤。请看示例：

例 7.31. 配置过滤参数

```
<parser>
    <filters>
        <parser-filters:uploaded-file-whitelist extensions="jpg, gif, png" />
    </filters>
</parser>
```

上面的配置将会禁止文件名后缀不在列表中的文件被上传到服务器上。如果做得更好一点，你甚至可以对上传文件进行病毒扫描。

目前，<parser>支持两种过滤器接口：ParameterValueFilter和UploadedFileFilter。前者用来对普通的参数值进行过滤（例如排除可能造成攻击的HTML代码）；后者用来对上传文件的file item对象进行过滤，就像刚才的uploaded-file-whitelist的例子。

7.4. <buffered> - 缓存response中的内容

7.4.1. 实现原理

Webx Turbine支持用layout/screen/control等部件共同购成一个页面。其中，每个layout可包含一个screen和多个control，每个screen可包含多个control，每个control还可以再包含其它的control。Screen和control的内容都可以用程序代码直接生成：

例 7.32. 在Screen中直接输出页面内容

```
public class MyScreenOrControl {
    @Autowired
    private HttpServletResponse response;

    public void execute() throws IOException {
        PrintWriter out = response.getWriter();

        out.println("<p>hello world</p>");
    }
}
```

上面的代码是非常直观、易理解的。事实上，如果你写一个简单的servlet来生成页面，代码也是和上面的类似。

但是，在简单的代码后面有一个玄机 —— 那就是这段代码可被用于生成嵌套的页面部件，它所生成的内容可被上一层嵌套的部件所利用。例如，一个screen中包含了一个control，那么screen可以获得它所调用的control的完整的渲染内容。

这个玄机就是靠<buffered>来实现的。<buffered>改变了response的输出流，包括output stream（二进制流）和writer（文本流），使写到输出流中的内容被暂存在内存中。当需要时，可以取得缓存中的所有内容。

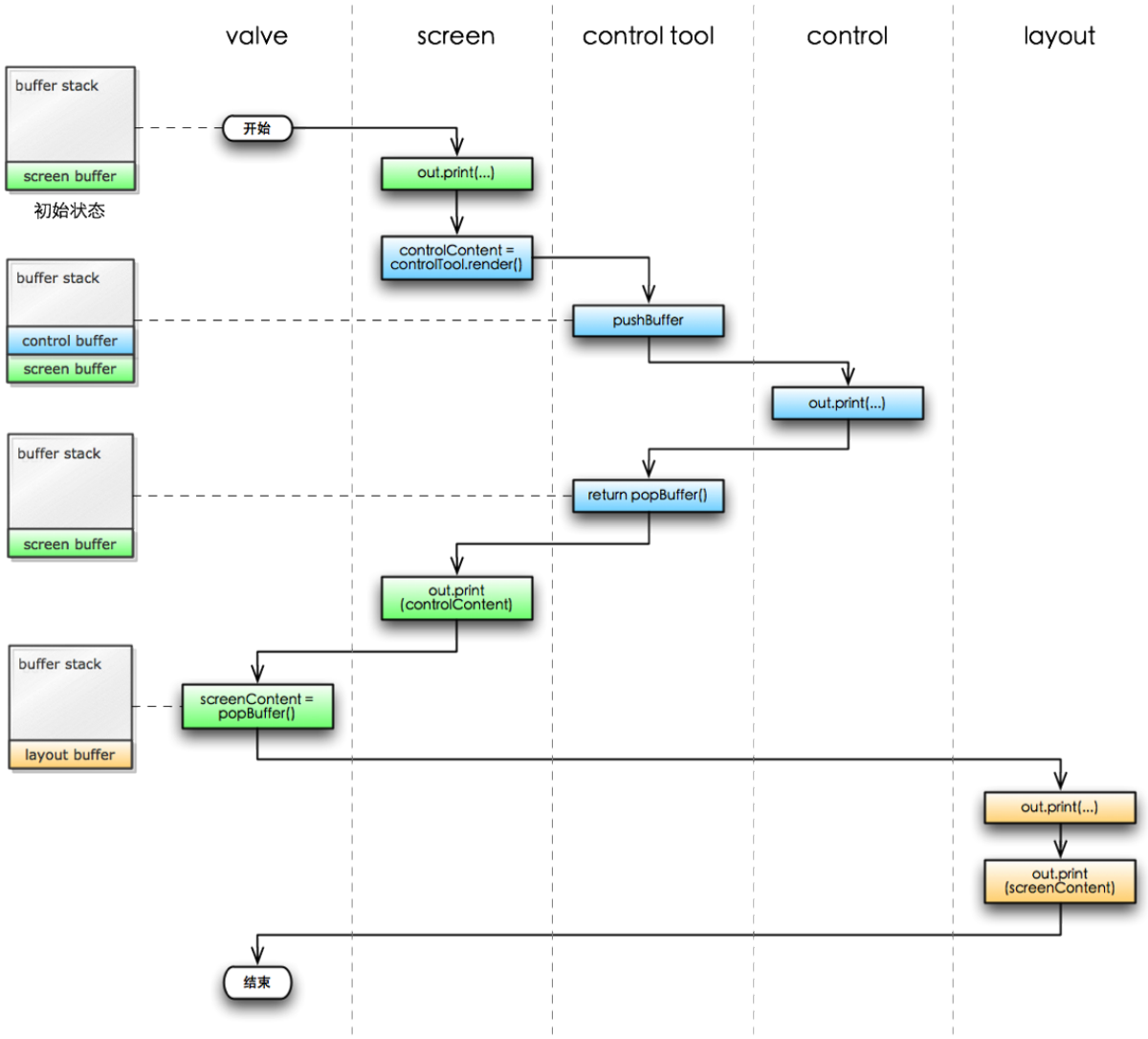


图 7.2. Webx利用<buffered>机制生成嵌套式页面的过程

如图所示。BufferedRequestContext主要包括了两条用来操作buffer栈的指令：push和pop。

- 每次push就会在栈顶创建一个新的buffer。

- 每次pop就会弹出栈顶buffer，并返回其内容。当最后一个buffer被弹出时，就会自动push一个新的buffer，从而确保任何时候栈都非空。
- 所有写入`response.getWriter()`和`response.getOutputStream()`输出流的数据，将被保存在栈顶的buffer中。
- Push和pop必须成对出现。如果在commit时发现栈内有两个或两个以上的buffer存在，说明有push/pop未匹配，则报错。
- Commit时，将仅存的栈顶buffer提交给浏览器。

`<buffered>`还有一个重要的作用，就是可以用来支持基于cookie的session机制（参见：*第 8 章 Request Context之Session指南*）。因为cookie是response header的一部分，根据HTTP协议，headers出现在content的前面。一旦content开始向浏览器输出，headers就不可能再被改变了。这会导致基于cookie的session无法保存的问题。`<buffered>`将所有的输出内容缓存在内存中，从而避免了response过早地提交给浏览器，也就解决了cookie无法保存的问题。

7.4.2. 使用方法

7.4.2.1. 配置

`<buffered>`的配置比较简单，没有任何额外的参数。只要像下面这样写就可以了：

例 7.33. 配置`<buffered>`（`/WEB-INF/webx.xml`）

```
<services:request-contexts xmlns="http://www.alibaba.com/schema/services/request-contexts">
    <buffered />
    ...
</services:request-contexts>
```

7.4.2.2. 操作buffer栈

例 7.34. 操作buffer栈

```
@Autowired
BufferedRequestContext buffered;

@Autowired
HttpServletResponse response;

...

PrintWriter out = response.getWriter();

buffered.pushBuffer(); // 创建新buffer，并压入栈顶
out.print("world");  // 在新buffer中写入

String content = buffered.popCharBuffer(); // 弹出顶层buffer

out.print("hello, ");
out.print(content); // 写入较低层的buffer
```

需要注意的是，response中有两种输出流：二进制流`response.getOutputStream()`和文本流`response.getWriter()`。与之对应的，`BufferedRequestContext`也会创建两种类型的buffer。这两种buffer类型是互斥的：

- 假如你的应用使用了`response.getWriter()`，那么，你必须使用`buffered.popCharBuffer()`以取得文本buffer的内容；
- 假如你的应用使用了`response.getOutputStream()`，那么，你必须使用`buffered.popByteBuffer()`以取得二进制buffer的内容。
- 如果用错，则抛`IllegalStateException`。

7.4.2.3. 关闭buffer机制

Buffer机制会延迟服务器对用户的响应。在大部分情况下，这不会造成明显的问题。但在某些情况下会产生严重的问题。此时，你需要把buffer机制关闭。

例如，动态生成excel文件、PDF文件以及图片文件。这样的需求有如下特点：

- 数据量大 —— 有可能达到几兆。如果把这样大的数据放在内存中，势必导致服务器性能的下降。
- 没有layout/screen/control这样的嵌套页面的需求，因此不需要buffer这样的机制来帮倒忙。
- 无状态，不需要修改session，因此也不需要buffer机制来帮助延迟提交。反过来，对于这样的大文件，提交越早越好 —— 甚至可以在文档还未完全生成的时候，就开始向用户浏览器输出，边生成边下载，从而节省大量的下载时间。

下面的程序代码模拟了一种情况 —— 生成一个120M的PDF文件。每生成1M内容，就故意暂停半秒。这样一来，120M的文件需要大约一分钟才能生成完毕。

例 7.35. 模拟生成PDF文档，关闭buffer以提高性能

```
public class MyDocument {
    @Autowired
    private BufferedRequestContext buffered;

    @Autowired
    private HttpServletResponse response;

    public void execute() throws Exception {
        buffered.setBuffering(false);

        response.setContentType("application/pdf");
        response.setHeader("Content-Disposition", "attachment; filename=\"mydocument.pdf\"");

        OutputStream out = response.getOutputStream();

        for (int m = 0; m < 120; m++) {
            for (int k = 0; k < 1024; k++) {
                for (int b = 0; b < 1024; b++) {
                    out.write((byte) b);
                }
            }

            // 每生成1M，暂停半秒
            Thread.sleep(500);
        }
    }
}
```

把上述类代码，放在screen目录中。然后访问URL：`http://localhost:8081/myapp/my_document.do`，就可以启动下载。

假如不关闭buffer机制，从用户点击下载，到浏览器提示保存文件，中间会相隔一分钟。这种用户体验是不可接受的。更糟糕的是，文件会占用至少120M的服务器内存，这也是几乎不可接受的。关闭buffer机制以后，以上两个问题就没有了：

- 用户点击下载链接，浏览器立即提示保存文件。
- 边下载边生成数据，生成数据的时间是一分钟，下载所需的时间也是一分钟左右。
- 生成的数据立即输出，不会占用过多的内存。

7.5. `<lazy-commit>` - 延迟提交response

7.5.1. 什么是提交

当浏览器向服务器发出请求，服务器就会返回一个response响应。每个response分成两部分：headers和content。下面是一个HTTP响应的例子：

例 7.36. HTTP请求的headers和content

```
HTTP/1.0 200 OK
Date: Sat, 08 Jan 2011 23:19:52 GMT
Server: Apache/2.0.63 (Unix)
...

<html>...
```

在服务器应用响应request的全过程中，都可以向浏览器输出response的内容。然而，已经输出到浏览器上的内容，是不可更改的；还没有输出的内容，还有改变的余地。这个输出的过程，被称为提交（commit）。

Servlet API中有一个方法，可以判定当前的response是否已经被提交。

例 7.37. 判断response是否已经被提交

```
if (response.isCommitted()) {
    ...
}
```

在Servlet API中，有下列操作可能导致response被提交：

- `response.sendError()`
- `response.sendRedirect()`
- `response.flushBuffer()`
- `response.setContentLength()` 或者`response.setHeader("Content-Length", length)`
- response输出流被写入并达到内部buffer的最大值（例如：8KB）

7.5.2. 实现原理

当response被提交以后，一切headers都不可再改变。这对于某些应用（例如cookie-based session）的实现是一个问题。

<lazy-commit>通过拦截response中的某些方法，来将可能导致提交的操作延迟到请求处理结束的时候，也就是request context本身被提交的时候。

<lazy-commit>必须和<buffered>配合，才能完全实现延迟提交。如前所述，<buffered>将所有的输出暂存在内存里，从而避免了因输出流达到内部buffer的最大值（例如：8KB）而引起的提交。

7.5.3. 使用方法

7.5.3.1. 配置

<lazy-commit>的配置比较简单，没有任何额外的参数。只要像下面这样写就可以了：

例 7.38. 配置<lazy-commit>（/WEB-INF/webx.xml）

```
<services:request-contexts xmlns="http://www.alibaba.com/schema/services/request-contexts">
    <lazy-commit />
    ...
</services:request-contexts>
```

7.5.3.2. 取得当前response的状态

通过LazyCommitRequestContext接口，你可以访问当前response的一些状态：

表 7.8. 通过LazyCommitRequestContext访问response状态

LazyCommitRequestContext方法名	说明
isError()	判断当前请求是否已出错
getErrorStatus()	如果sendError()方法曾被调用，则该方法返回一个error状态值。
getErrorMessage()	如果sendError()方法曾被调用，则该方法返回一个error信息。
isRedirected()	判断当前请求是否已被重定向。
getRedirectLocation()	取得重定向的URI。
getStatus()	取得最近设置的HTTP status

7.6. <rewrite> -重写请求的URL和参数

7.6.1. 概述

<rewrite>的功能和设计完全类似于Apache HTTPD Server所提供的mod_rewrite模块。它可以根据规则，在运行时修改URL和参数。

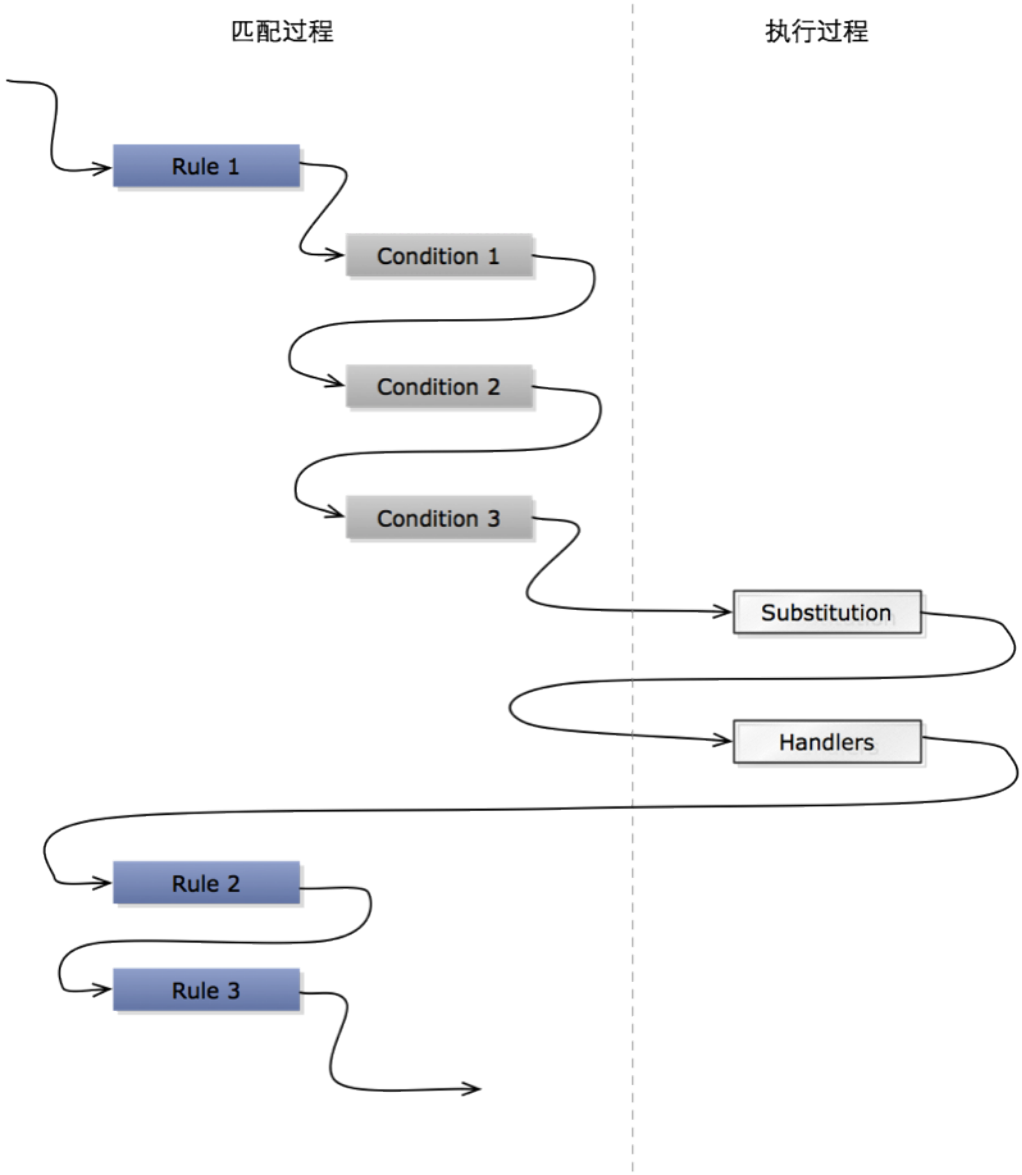


图 7.3. Rewrite工作原理

当一个请求进入<rewrite>以后，它的处理过程如上图所示。过程可分为两个大的步骤，即：匹配和执行。

- 匹配

  1. 取得URL中的path路径。
  2. 用所取得的path，依次匹配rule1、rule2、rule3中的pattern，直到找到第一个匹配。
  3. 假如rule中包含conditions，则测试conditions。如果condtions不满足，则当前的rule匹配失败，回到第2步，继续匹配下一个rules。
  4. 假如rule不包含conditions，或者conditions被满足，则当前的rule匹配成功，进入“执行”阶段。

- 执行

  1. 执行substitution替换。这可能导致path和参数的改变。
  2. 执行所有的handlers。这为编程者提供了更灵活的手段来改变request中的数据。
  3. 根据substitution中的指示，结束<rewrite>的执行、或者回到匹配阶段，用新的path和参数继续匹配后续的rules。
  4. <rewrite>结束时，根据substitution中的指示，改写request或者重定向到新的URL。

下面是一个<rewrite>配置的模板：

例 7.39. 配置<rewrite>（/WEB-INF/webx.xml）

```
<services:request-contexts xmlns="http://www.alibaba.com/schema/services/request-contexts">
    <rewrite>

        <!-- rule 1 -->
        <rule pattern="...">
            <condition test="..." pattern="..." flags="..." />
            <condition test="..." pattern="..." flags="..." />
            <substitution uri="..." flags="...">
                <parameter key="..." value="..." />
                <parameter key="..." value="..." />
                <parameter key="..." value="..." />
            </substitution>
            <handlers>
                <rewrite-handlers:handler class="..." />
            </handlers>
        </rule>

        <!-- rule 2 -->
        <rule pattern="...">
        </rule>

        <!-- rule 3 -->
        <rule pattern="...">
        </rule>

    </rewrite>
    ...
</services:request-contexts>
```

7.6.2. 取得路径

和Apache mod_rewrite不同，用来匹配rules的路径并不是URL的整个路径，而是由servletPath + pathInfo两部分组成，其中并不包含contextPath。

这是因为<rewrite>是属于WEB应用的，它只能匹配当前应用中的路径。在基于servlet的WEB应用中，一个完整的URL路径是由contextPath + servletPath + pathInfo三部分组成的。其中contextPath是用来区分应用的，所以对<rewrite>没有意义。

例如，URL是http://localhost:8081/myapp*/myservlet/path/path*，那么<rewrite>用来匹配rules的路径是：*/myservlet/path/path*。

7.6.3. 匹配rules

下面是一个简单的rule。

例 7.40. 匹配规则的配置

```
<rule pattern="/test1/hello\.htm">
    ...
</rule>
```

其中，rule pattern是一个正则表达式。特别需要注意的是，*这个正则表达式是部分匹配的*。如上例pattern可以匹配下面的路径：

- */test1/hello.htm*
- /mypath*/test1/hello.htm*
- /mypath*/test1/hello.htm*/mypath

如果你希望匹配整个path，请使用正则表达式的“^”和“$”标记。例如：

例 7.41. 匹配整个path

```
<rule pattern="^/test1/hello\.htm$">
```

部分匹配的正则表达式为你提供了较灵活的匹配能力，例如，下面的rule可以用来匹配所有以jpg为后缀的URL。

例 7.42. 后缀匹配

```
<rule pattern="\.jpg$">
```

此外，rules pattern还支持否定的pattern —— 即在正常的pattern前加上“!”即可。例如下面的rule匹配所有不以jpg为后缀的URL：

例 7.43. 否定匹配

```
<rule pattern="!\.jpg$">
```

7.6.4. 匹配conditions

每个rule都可以包含多个额外的conditions。Conditions提供了除path匹配以外的其它条件。下面是condition配置的基本格式：

例 7.44. 配置conditions

```
<rule pattern="/path">
    <condition test="..." pattern="..." flags="..." />
    <condition test="..." pattern="..." flags="..." />
    <condition test="..." pattern="..." flags="..." />
    ...
</rule>
```

每个condition由两个主要的参数：测试表达式和pattern。测试表达式中可以使用下面的变量：

表 7.9. Condition变量

客户端信息		
%{REMOTE_HOST}	客户端主机名。	相当于request.getRemoteHost()
%{REMOTE_ADDR}	客户端地址。	相当于request.getRemoteAddr()
%{REMOTE_USER}	用户名。	相当于request.getRemoteUser()
%{AUTH_TYPE}	验证用户的方法。例如BASIC、FORM、CLIENT_CERT、DIGEST等。	相当于request.getAuthType()

服务端信息		
%{SERVER_NAME}	服务器主机名。	相当于request.getServerName()
%{SERVER_PORT}	服务器端口。	相当于request.getServerPort()
%{SERVER_PROTOCOL}	服务器协议。	相当于request.getProtocol()
请求信息		
%{REQUEST_METHOD}	HTTP方法名。例如GET、POST等。	相当于request.getMethod()
%{REQUEST_URI}	所请求的URI，不包括主机名、端口和参数。	相当于request.getRequestURI()
%{QUERY_STRING}	参数和值。注意，对于POST请求取得QUERY_STRING，可能会影响性能。	相当于request.getQueryString()
%{QUERY:param}	取得参数值。无论哪一种类型的请求（GET/POST/上传文件），都可以取得参数值。	相当于request.getParameter("param")
HTTP headers		
%{HTTP_USER_AGENT}	浏览器名称。	相当于request.getHeader("User-Agent")
%{HTTP_REFERER}	前一个URL。	相当于request.getHeader("Referer")
%{HTTP_HOST}	HTTP请求中的主机名，一般代表虚拟主机。	相当于request.getHeader("Host")
%{HTTP_ACCEPT}	浏览器可以接受的文档类型。	相当于request.getHeader("Accept")
%{HTTP_COOKIE}	浏览器发送过来的cookie。	相当于request.getHeader("Cookie")

Condition pattern和rule pattern类似，也是*部分匹配的正则表达式，并且支持否定的pattern*。举例说明：

例 7.45. Condition patterns

```
<rule pattern="/path"> ❶
    <condition test="%{SERVER_NAME}:%{SERVER_PORT}" pattern="www.(\w+).com:8080" /> ❷
    <condition test="%{QUERY:x}" pattern="!1" /> ❸
    <condition test="%{QUERY:y}" pattern="2" /> ❹
</rule>
```

上面的rule匹配符合以下条件的请求：

❶ 匹配路径/path。

❷ 服务器名为www.*.com，端口为8080。

❸ 并且参数x!=1

❹ 并且参数y=2

默认情况下，必须所有的conditions条件都符合，rule才会继续执行下去。但是condition还支持一个选项：OR或者ornext。如果condtion带有这个选项，只要符合当前condition*或者*后续的conditions，rule就会执行下去。例如：

例 7.46. 部分匹配conditions

```
<rule pattern="/path"> ❶
    <condition test="%{QUERY:x}" pattern="1" flags="OR" /> ❷
    <condition test="%{QUERY:y}" pattern="2" flags="ornext" /> ❸
    <condition test="%{QUERY:z}" pattern="3" /> ❹
</rule>
```

上例中，“OR”和“ornext”代表完全一样的意思。这个rule匹配符合以下条件的请求：

❶ 匹配路径/path。

❷ 参数x=1，

❸ *或者*y=2，

❹ *或者*z=3。

7.6.5. 替换路径

当路径匹配，并且conditions也匹配（如果有的话），那么<rewrite>就会执行所匹配的rule。

例 7.47. 替换路径

```
<rule pattern="/test1/hello\.htm">
    <substitution uri="/test1/new_hello\.htm" />
</rule>
```

上例中的rule将执行下面的替换（别忘了，rule支持部分匹配，只有匹配的部分被替换）：

- 将*/test1/hello.htm*替换成*/test1/new_hello.htm*。
- 将/mypath*/test1/hello.htm*替换成/mypath*/test1/new_hello.htm*。
- 将/mypath*/test1/hello.htm*/mypath替换成/mypath/test1/new_hello.htm/mypath。

路径替换时，还支持正则表达式变量。例如：

例 7.48. 用正则表达式变量替换路径

```
<rule pattern="/(\w+)\.htm">
    <condition test="%{SERVER_NAME}" pattern="(\w+).blogs.com" />
    <substitution uri="/%1/new_$1\.htm" />
</rule>
```

需要注意的是，rule pattern中的匹配项，是用“$1”、“$2”、“$3”表示的；而condition pattern中的匹配项，是用“%1”、“%2”、“%3”表示的。只有最后一个被匹配的condition中的匹配项，才被保留用于替换。

上面的rule将执行下面的替换：将http://*myname*.blogs.com*/hello.htm*替换成同服务器上的路径：*/myname/new_hello.htm*。

7.6.6. 替换参数

<rewrite>不仅可以替换路径，还可以替换参数。

例 7.49. 替换参数

```
<rule pattern="/hello.(\w+)">
    <condition test="%{SERVER_NAME}" pattern="www.(\w+).com" />
    <substitution>
        <parameter key="ext" value="$1" />
        <parameter key="host" value="%1" />
        <parameter key="count">
            <value>1</value>
            <value>2</value>
            <value>3</value>
        </parameter>
    </substitution>
</rule>
```

替换参数和替换路径类似，也可以指定rule和condition pattern中的匹配项。参数支持多值，例如上例中的`count`参数。 上面的例子将执行以下替换行为：

- 对于请求：`http://www.myserver.com/hello.htm`，不改变其路径，只改变其参数：
  - 创建单值参数：ext=*htm*（从rule pattern中取得`$1`）
  - 创建单值参数：host=*myserver*（从condition pattern中取得`%1`）
  - 创建多值参数：count=*[1, 2, 3]*
  - *删除其它所有参数*。

如果你想保留原来所有参数，只是修改或添加一些参数，可以指定`QSA`或`qsappend`选项。

例 7.50. 保留原来的参数

```
<substitution flags="QSA">
    ...
</substitution>
```

7.6.7. 后续操作

当一个rule和其中的conditions被匹配时，`<rewrite>`就会执行这个rule。执行的结果通常是改变请求的路径或参数。当一个rule执行完毕以后，接下来做什么呢？有几种可能的情况。

7.6.7.1. 继续匹配剩余的rules

例 7.51. 默认后续操作：继续匹配剩余的rules

```
<rule pattern="...">
    <substitution uri="..." />
</rule>
<rule pattern="...">
    <substitution uri="..." />
</rule>
```

上面第一个rule执行完以后，`<rewrite>`会用改变过的路径和参数去继续匹配余下的规则。这是默认情况。

7.6.7.2. 停止匹配

例 7.52. 后续操作：停止匹配

```
<rule pattern="...">
    <substitution uri="..." flags="L" />
</rule>
<rule pattern="...">
    <substitution uri="..." />
</rule>
```

当在substitution中指定`L`或者`last`选项时，rule匹配会到此中止。后续的rules不会再被匹配。

7.6.7.3. 串接rules

例 7.53. 后续操作：串接rules

```
<rule pattern="^/common-prefix">
    <substitution flags="C" />
</rule>
<rule pattern="\.jpg">
    <substitution uri="..." />
</rule>
<rule pattern="\.htm">
    <substitution uri="..." />
</rule>
```

当在substitution中指定C或者chain选项时，假如当前rule匹配，则会像默认情况一样继续匹配剩余的rules；否则，就像last选项一样立即中止匹配。

串接rules在下面的情况下非常有用：即对一个路径进行匹配多个patterns。例如上面的例子中，第一个rule限定了路径前缀必须是“/common-prefix”，接下来的rules在此基础上继续判断：后缀是“jpg”还是“htm”？

7.6.8. 重定向

例 7.54. 重定向

永久重定向，status code=301

```
<rule pattern="^/hello1\.htm">
    <substitution uri="/new_hello.htm" flags="L,R=301" />
</rule>
```

临时重定向，status code=302，不保留参数

```
<rule pattern="^/hello2\.htm">
    <substitution uri="/new_hello.htm" flags="L,R" />
</rule>
```

临时重定向，status code=302，保留参数

```
<rule pattern="^/hello3\.htm">
    <substitution uri="/new_hello.htm" flags="L,R,QSA" />
</rule>
```

绝对URL重定向，status code=302

```
<rule pattern="^/hello4\.htm">
    <substitution uri="http://www.other-site.com/new_hello.htm" flags="L,R" />
</rule>
```

当在substitution中指定R或者redirect的时候，<rewrite>会返回“重定向”的响应。 重定向有两种：301永久重定向，和302临时重定向。默认是302临时重定向，但你可以指定301来产生一个永久的重定向。

通常，R标记会和L标记一起使用，使<rewrite>立即结束。

重定向和QSA标记一起使用时，可以将当前请求的所有参数附加到重定向请求中。不过这里需要注意的是，假如当前请求是一个post请求，那么将参数附加到新的URL中，可能会导致URL过长而重定向失败的问题。

重定向可以指向另一个不同域名的网站 —— 反过来说，*假如你希望rewrite到另一个网站，那么你必须指定重定向的选项才行*。

7.6.9. 自定义处理器

例 7.55. 自定义处理器

```
<rule pattern="...">
    <handlers>
        <rewrite-handlers:handler class="..." />
        <rewrite-handlers:handler class="..." />
    </handlers>
</rule>
```

有时候，基于正则表达式替换的substitution不能满足较复杂的需求，好在`<rewrite>`还提供了另一种机制：自定义处理器。

当rule和conditions被匹配的时候，所有的handlers将被执行。Webx提供了一个handler参考实现：

例 7.56. 自定处理器参考实现：规格化路径

```
<rule pattern="...">
    <handlers>
        <rewrite-handlers:handler
            class="com.alibaba.citrus.service.requestcontext.rewrite.support.UrlNormalizer"
        />
    </handlers>
</rule>
```

7.7. 本章总结

本文详细介绍了Request Contexts的功能。

Request Contexts服务是Webx框架的核心功能之一。它看似简单，但却提供了很多有用功能。相对于其它框架中的解决方案，RequestContexts显得更加优雅，因为其中大部分功能对应用程序是透明的 —— 应用程序不需要知道它们的存在，就可以享受它们所提供的功能。

第 8 章 Request Context之Session指南

Webx实现了一套session框架。Session框架建立在request contexts机制之上。建议你先阅读第 6 章 *Filter、Request Contexts和Pipeline*和第 7 章 *Request Contexts功能指南*，以便了解request contexts是怎么回事。

8.1. Session概述

8.1.1. 什么是Session

HTTP协议是无状态的，但通过session机制，就能把无状态的变成有状态的。Session的功能就是保存HTTP请求之间的状态数据。有了session的支持，就很容易实现诸如用户登录、购物车等网站功能。在Servlet API中，有一个`HttpSession`的接口。你可以这样使用它：

例 8.1. 在Java代码中访问session

在一个请求中，保存session的状态

```
// 取得session对象
HttpSession session = request.getSession();

// 在session中保存用户状态
session.setAttribute("loginId", "myName");
```

在另一个请求中，取出session的状态：

```
// 得到"myName"
String myName = (String) session.getAttribute("loginId");
```

8.1.2. Session数据存在哪?

Session的状态数据是怎样保存的呢?

8.1.2.1. 保存在应用服务器的内存中

一般的做法，是将session对象保存在内存里。同一时间，会有很多session被保存在服务器的内存里。由于内存是有限的，较好的服务器会把session对象的数据交换到文件中，以确保内存中的session数目保持在一个合理的范围内。

为了提高系统扩展性和可用性，我们会使用集群技术 —— 就是一组独立的机器共同运行同一个应用。对用户来讲，集群相当于一台“大型服务器”。而实际上，同一用户的两次请求可能被分配到两台不同的服务器上来处理。这样一来，怎样保证两次请求中存取的session值一致呢?

一种方法是使用session复制：当session的值被改变时，将它复制到其它机器上。这个方案又有两种具体的实现，一种是广播的方式。这种方式下，任何一台服务器都保存着所有服务器所接受到的session对象。服务器之间随时保持着同步，因而所有服务器都是等同的。可想而知，当访问量增大的时候，这种方式花费在广播session上的带宽有多大，而且随着机器增加，网络负担成指数级上升，不具备高度可扩展性。

另一种方法是TCP-Ring的方式，也就是把集群中所有的服务器看成一个环，A→B→C→D→A，首尾相接。把A的session复制到B，B的session复制到C，……，以此类推，最后一台服务器的session复制到A。这样，万一A宕机，还有B可以顶上来，用户的session数据不会轻易丢失。但这种方案也有缺点：一是配置复杂；二是每增添/减少一台机器时，ring都需要重新调整，这将成为性能瓶颈；三是要求前端的Load Balancer具有相当强的智能，才能将用户请求分发到正确的机器上。

8.1.2.2. 保存在单一数据源中

也可以将session保存在单一的数据源中，这个数据源可被集群中所有的机器所共享。这样一来，就不存在复制的问题了。

然而单一数据源的性能成了问题。每个用户请求，都需要访问后端的数据源（很可能是数据库）来存取用户的数据。

这种方案的第二个问题是：缺少应用服务厂商的支持 —— 很少有应用服务器直接支持这种方案。更不用说数据源有很多种（MySQL、Oracle、Hsqldb等各种数据库、专用的session server等）了。

第三个问题是：数据源成了系统的瓶颈，一但这个数据源崩溃，所有的应用都不可能正常运行了。

8.1.2.3. 保存在客户端

把session保存在客户端。这样一来，由于不需要在服务器上保存数据，每台服务器就变得独立，能够做到线性可扩展和极高的可用性。

具体怎么做呢？目前可用的方法，恐怕就是保存在cookie中了。但需要提醒的是，cookie具有有以下限制，因此不可无节制使用该方案：

- Cookie数量和长度的限制。每个domain最多只能有20条cookie，每个cookie长度不能超过4KB，否则会被截掉。
- 安全性问题。如果cookie被人拦截了，那人就可以取得所有的session信息。即使加密也与事无补，因为拦截者并不需要知道cookie的意义，他只要原样转发cookie就可以达到目的了。
- 有些状态不可能保存在客户端。例如，为了防止重复提交表单，我们需要在服务器端保存一个计数器。如果我们把这个计数器保存在客户端，那么它起不到任何作用。

虽然有上述缺点，但是对于其优点（极高的扩展性和可用性）来说，就显得微不足道。我们可以用下面的方法来回避上述的缺点：

- 通过良好的编程，控制保存在cookie中的session对象的大小。
- 通过加密和安全传输技术（SSL），减少cookie被破解的可能性。
- 只在cookie中存放不敏感数据，即使被盗也不会有重大损失。
- 控制cookie的生命期，使之不会永远有效。偷盗者很可能拿到一个过期的cookie。

8.1.2.4. 将客户端、服务器端组合的方案

任何一种session方案都有其优缺点。最好的方法是把它们结合起来。这样就可以弥补各自的缺点。

将大部分session数据保存在cookie中，将小部分关键和涉及安全的数据保存在服务器上。由于我们只把少量关键的信息保存在服务端，因而服务器的压力不会非常大。

在服务器上，单一的数据源比复制session的方案，更简单可靠。我们可以使用数据库来保存这部分session，也可以使用更廉价、更简单的存储，例如Berkeley DB就是一种不错的服务器存储方案。将session数据保存在cookie和Berkeley DB（或其它类似存储技术）中，就可以解决我们的绝大部分问题。

8.1.3. 创建通用的session框架

多数应用服务器并没有留出足够的余地，来让你自定义session的存储方案。纵使某个应用服务器提供了对外扩展的接口，可以自定义session的方案，我们也不大可能使用它。为什么呢？因为我们希望保留选择应用服务器软件的自由。

因此，最好的方案，不是在应用服务器上增加什么新功能，而是在WEB应用框架上做手术。一但我们在WEB应用框架中实现了这种灵活的session框架，那么我们的应用可以跑在任何标准的JavaEE应用服务器上。

除此之外，一个好的session框架还应该做到对应用程序透明。具体表现在：

- 使用标准的`HttpSession`接口，而不是增加新的API。这样任何WEB应用，都可以轻易在两种不同的session机制之间切换。
- 应用程序不需要知道session中的对象是被保存到了cookie中还是别的什么地方。
- Session框架可以把同一个session中的不同的对象分别保存到不同的地方去，应用程序同样不需要关心这些。例如，把一般信息放到cookie中，关键信息放到Berkeley DB中。甚至同是cookie，也有持久和临时之分，有生命期长短之分。

Webx实现了这种session框架，把它建立在Request Contexts的基础上。

8.2. Session框架

8.2.1. 最简配置

例 8.2. Session框架基本配置（/WEB-INF/webx.xml）

```
<services:request-contexts xmlns="http://www.alibaba.com/schema/services/request-contexts">
    <buffered />
    <lazy-commit />
    ...
    <session>
        <stores>
            <session-stores:simple-memory-store id="simple" /> ❶
        </stores>
        <store-mappings>
            <match name="*" store="simple" /> ❷
        </store-mappings>
    </session>
</services:request-contexts>
```

以上的配置，创建了一个最基本的session实现：将所有数据（❷ name=*）保存在内存里（❶ simple-memory-store）。

警告

最简配置只能用于开发，*请不要将上述配置用在生产环境*。因为simple-memory-store只是将数据保存在内存里。在生产环境中，内存有被耗尽的可能。这段配置也不支持服务器集群。

8.2.2. Session ID

Session ID唯一标识了一个session对象。把session ID保存在cookie里是最方便的。这样，凡是cookie值相同的所有的请求，就被看作是在同一个session中的请求。在servlet中，还可以把session ID编码到URL中。Session框架既支持把session ID保存在cookie中，也支持把session ID编码到URL中。

完整的session ID配置如下：

例 8.3. Session ID的配置

```
<session>
    <id cookieEnabled="true" urlEncodeEnabled="false">
        <cookie name="JSESSIONID" domain="" maxAge="0" path="/" httpOnly="true" secure="false" />
        <url-encode name="JSESSIONID" />
        <session-idgens:uuid-generator />
    </id>
</session>
```

上面这段配置包含了关于Session ID的所有配置以及默认值。如果不指定上述参数，则系统将使用默认值，其效果等同于上述配置。

表 8.1. Session ID的配置说明

配置<session><id> —— 将Session ID保存于何处?	
cookieEnabled	是否把session ID保存在cookie中，如若不是，则只能保存的URL中。

配置<session><id> —— 将Session ID保存于何处?	
	默认为开启：true。
urlEncodeEnabled	是否支持把session ID编码在URL中。如果为true开启，应用必须调用response.encodeURL()或response.encodeRedirectURL()来将JSESSIONID编码到URL中。 默认为关闭：false。
配置<session><id><cookie> —— 将Session ID存放于cookie的设置	
name	Session ID cookie的名称。 默认为JSESSIONID。
domain	Session ID cookie的domain。 默认为空，表示根据当前请求自动设置domain。这意味着浏览器认为你的cookie属于当前域名。如果你的应用包含多个子域名，例如：www.alibaba.com、china.alibaba.com，而你又希望它们能共享session的话，请把域名设置成“alibaba.com”。
maxAge	Session ID cookie的最长存活时间（秒）。 默认为0，表示临时cookie，随浏览器的关闭而消失。
path	Session ID cookie的path。 默认为/，表示根路径。
httpOnly	在session ID cookie上设置HttpOnly标记。 在IE6及更新版本中，可以缓解XSS攻击的危险。默认为true。
secure	在session ID cookie上设置Secure标记。 这样，只有在https请求中才可访问该cookie。默认为false。
配置<session><id><url-encode> —— 将Session ID编码到URL的设置	
name	指定在URL中表示session ID的名字，默认也是JSESSIONID。 此时，如果urlEncodeEnabled为true的话，调用： response.encodeURL("http://localhost:8080/test.jsp?id=1") 将得到类似这样的结果： http://localhost:8080/test.jsp;JSESSIONID=xxxyyyzzz?id=1
配置<session><id><session-idgens:*> —— 如何生成session ID?	
uuid-generator	以UUID作为新session ID的生成算法。 这是默认的session ID生成算法。

为了达到最大的兼容性，我们分两种情况来处理JSESSIONID：

1. 当一个新session到达时，假如cookie或URL中已然包含了JSESSIONID，那么我们将直接利用这个值。为什么这样做呢？因为这个JSESSIONID可能是由同一域名下的另一个不相关应用生成的。如果我们不由分说地将这个cookie覆盖掉，那么另一个应用的session就会丢失。

2. 多数情况下，对于一个新session，应该是不包含JSESSIONID的。这时，我们需要利用SessionIDGenerator来生成一个唯一的字符串，作为JSESSIONID的值。SessionIDGenerator的默认实现UUIDGenerator。

8.2.3. Session的生命期

所谓生命期，就是session从创建到失效的整个过程。其状态变迁如下图所示：

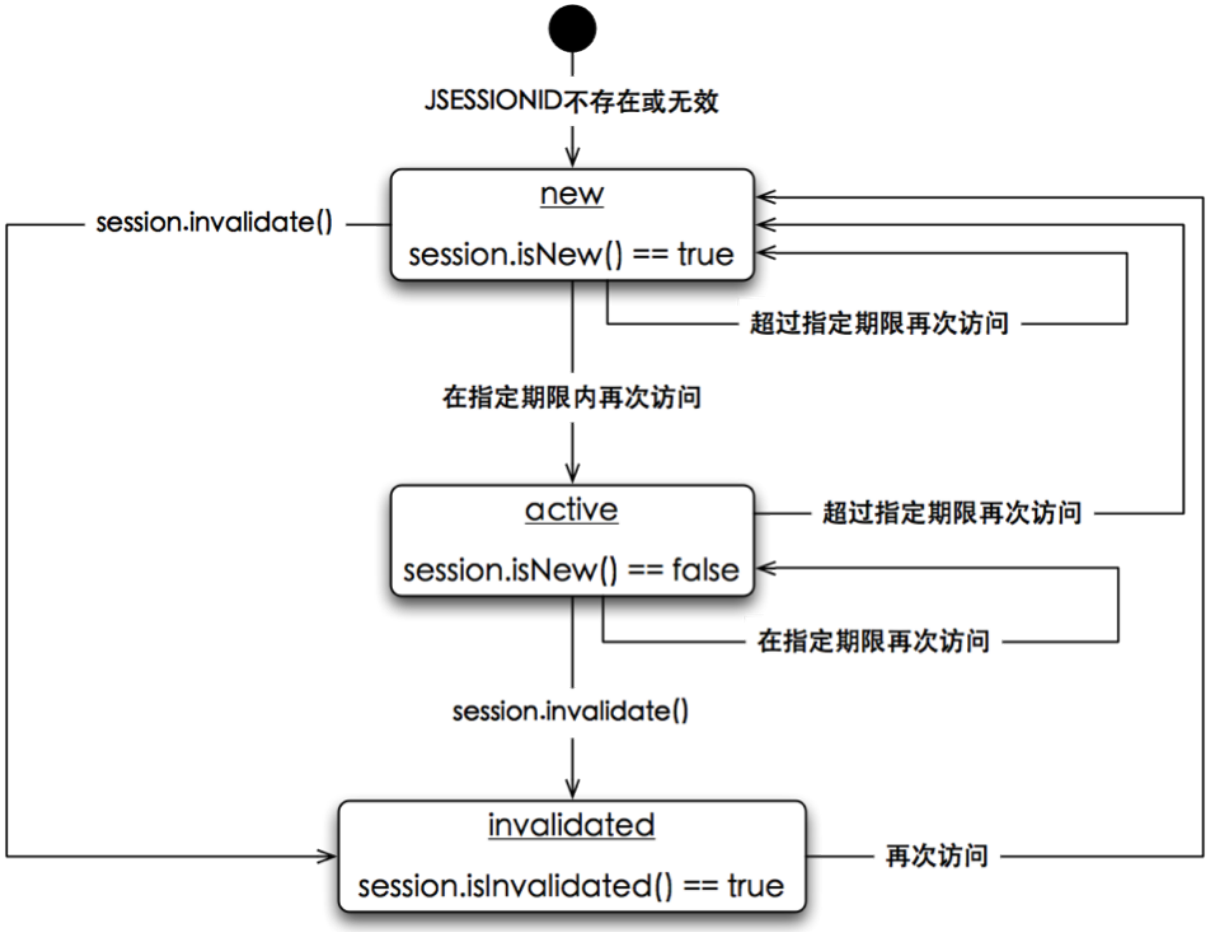


图 8.1. Session生命期

总结一下，其实很简单：

1. 第一次打开浏览器时，`JSESSIONID`还不存在，或者存在由同一域名下的其它应用所设置的无效的`JSESSIONID`。这种情况下，`session.isNew()`返回`true`。

2. 随后，只要在规定的时间间隔内，以及cookie过期之前，每一次访问系统，都会使session得到更新。此时`session.isNew()`总是返回`false`。Session中的数据得到保持。

3. 如果用户有一段时间不访问系统了，超过指定的时间，那么系统会清除所有的session内容，并将session看作是新的session。

4. 用户可以调用`session.invalidate()`方法，直接清除所有的session内容。此后所有试图`session.getAttribute()`或`session.setAttribute()`等操作，都会失败，得到`IllegalStateException`异常，直到下一个请求到来。

在session框架中，有一个重要的特殊对象，用来保存session生命期的状态。这个对象叫作session model。它被当作一个普通的对象存放在session中，但是通过`HttpSession`接口不能直接看到它。

关于session生命期的完整配置如下：

例 8.4. 关于Session生命期的配置

```
<session maxInactiveInterval="0" keepInTouch="false" forceExpirationPeriod="14400"
         modelKey="SESSION_MODEL">
    ...
</session>
```

参数的意思是：

表 8.2. Session生命期的配置参数

参数名	说明
maxInactiveInterval	指定session不活动而失效的期限，单位是秒。 默认为0，也就是永不失效（除非cookie失效）。例如，设置3600秒，表示用户离开浏览器1小时以后再回来，session将重新开始，老数据将被丢弃。
keepInTouch	是否每次都touch session（即更新最近访问时间）。 如果是false，那么只在session值有改变时touch。当将session model保存在cookie中时，设为false可以减少网络流量。但如果session值长期不改变，由于最近访问时间一直无法更新，将会使session超过maxInactiveInterval所设定的秒数而失效。 默认为false。
forceExpirationPeriod	指定session强制作废期限，单位是秒。 无论用户活动与否，从session创建之时算起，超过这个期限，session将被强制作废。这是一个安全选项：万一cookie被盗，过了这个期限的话，那么无论如何，被盗的cookie就没有用了。 默认为0，表示无期限。
modelKey	指定用于保存session状态的对象的名称。 默认为"SESSION_MODEL"。一般没必要修改这个值。

8.2.4. Session Store

Session Store是session框架中最核心的部分。Session框架最强大的部分就在于此。我们可以定义很多个session stores，让不同的session对象分别存放到不同的Session Store中。前面提到有一个特殊的对象“SESSION_MODEL”也必须保存在一个session store中。

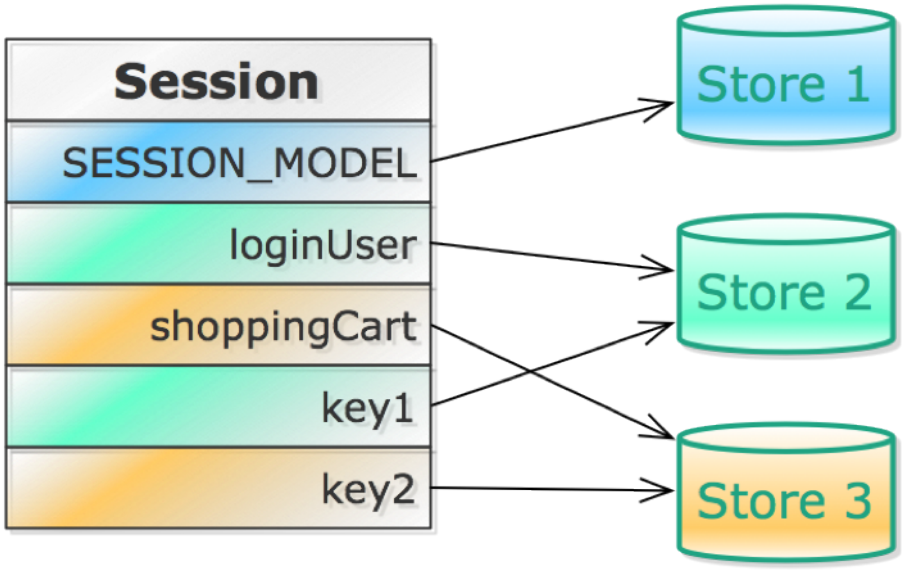


图 8.2. Session和Stores

类似于Servlet的配置，Session store的配置也包含两部分内容：session store的定义，和session store的映射（mapping）。

例 8.5. Session Store的配置

```
<session>
    <stores>
        <session-stores:store id="store1" /> ❶
        <session-stores:store id="store2" /> ❷
        <session-stores:store id="store3" /> ❸
    </stores>
    <store-mappings>
        <match name="*" store="store1" /> ❹
        <match name="loginName" store="store2" /> ❺
        <matchRegex pattern="key.*" store="store3" /> ❻
    </store-mappings>
</session>
```

❶❷❸ 定义session stores：你可以配置任意多个session store，只要ID不重复。此处，`store1`、`store2`和`store3`分别是三个session store的名称。

❹❺ 映射session stores：`match`标签用来精确匹配attribute name。一个特别的值是“*”，它代表默认匹配所有的names。

本例中， 如果调用`session.setAttribute("loginName", user.getId())`，那么这个值将被保存到`store2`里；如果调用`session.setAttribute("other", value)`将被默认匹配到`store1`中。

❻ 映射session stores：`matchRegexp`标签用正则表达式来匹配attribute names。

本例中， `key_a`、`key_b`等值都将被保存到`store3`里。

需要注意以下几点：

- *在整个session配置中，只能有一个store拥有默认的匹配*。

- 假如有多个`match`或`matchRegex`同时匹配某个attribute name，那么遵循以下匹配顺序：

  1. *精确的匹配*最优先。
  2. 正则表达式的匹配遵循最大匹配的原则，假如有两个以上的正则表达式被同时匹配，*长度较长的匹配*胜出。
  3. *默认匹配**总是在所有的匹配都失败以后才会被激活。

- 必须有一个session store能够用来存放session model。
  - 你可以用`<match name="*">`来匹配session model；
  - 也可以用精确匹配：`<match name="SESSION_MODEL" />`。其中session model的名字是必须和前述modelKey配置的值相同，其默认值为“`SESSION_MODEL`”。

8.2.5. Session Model

Session Model是用来记录当前session的生命期数据的，例如：session的创建时间、最近更新时间等。默认情况下，

- 当需要保存session数据时，`SessionModel`对象将被转换成一个JSON字符串（如下所示），然后这个字符串将被保存在某个session store中：

```
{id:"SESSION_ID",ct:创建时间,ac:最近访问时间,mx:最长不活动时间}
```

- 需要读取时，先从store中读到上述格式的字符串数据，然后再把它解码成真正的`SessionModel`对象。

以上转换过程是通过一个`SessionModelEncoder`接口来实现的。为了提供更好的移植性，Session框架可同时支持多个`SessionModelEncoder`的实现。配置如下：

例 8.6. Session Model编码器的配置

```
<session>
    <session-model-encoders>
        <model-encoders:default-session-model-encoder />
        <model-encoders:model-encoder class="..." />
        <model-encoders:model-encoder class="..." />
    </session-model-encoders>
</session>
```

在上面的例子中，提供了三个`SessionModelEncoder`的实现。第一个是默认的实现，第二、第三个是任意实现。

- 当从store取得`SessionModel`对象时，框架将依次尝试所有的encoder，直到解码成功为止。
- 当将`SessionModel`对象保存到store之前，框架将使用第一个encoder来编码对象。

当你从不同的`SessionModel`编码方案中移植的时候，上述多encoders共存的方案可以实现平滑的过渡。

8.2.6. Session Interceptor

Session Interceptor拦截器的作用是拦截特定的事件，甚至干预该事件的执行结果。目前有两种拦截器接口：

表 8.3. Session Interceptor拦截器接口

接口	功能
`SessionLifecycleListener`	监听以下session生命期事件： • Session被创建 • Session被访问 • Session被作废
`SessionAttributeInterceptor`	拦截以下session读写事件： • `onRead` – 拦截`session.getAttribute()`方法，可以修改所读取的数据。 • `onWrite` – 拦截`session.setAttribute()`方法，可以修改所写到store中的数据。

Session框架自身已经提供了两个有用的拦截器：

表 8.4. Session Interceptor的实现

名称	说明
`<lifecycle-logger>`	监听session生命期事件，并记录日志。
`<attribute-whitelist>`	控制session中的attributes，只允许白名单中所定义的attribute名称和类型被写入到或读出于session store中。 这个功能对于cookie store是很有用的。因为cookie有长度的限制，所以需要用白名单来限制写入到cookie中的数据数量和类型。

你可以同时配置多种拦截器，如下所示。

例 8.7. 配置session interceptors

```
<session>
  <request-contexts:interceptors
        xmlns="http://www.alibaba.com/schema/services/request-contexts/session/interceptors">

    <lifecycle-logger />

    <attribute-whitelist>
      <attribute name="_csrf_token" />
      <attribute name="_lang" />
      <attribute name="loginUser" type="com.alibaba...MyUser" />
      <attribute name="shoppingCart" type="com.alibaba....ShoppingCart" />
    </attribute-whitelist>

    <interceptor class="..." />

  </request-contexts:interceptors>
</session>
```

8.3. Cookie Store

Cookie Store的作用，是将session对象保存在客户端cookie中。Cookie Store减轻了服务器维护session数据的压力，从而提高了应用的扩展性和可用性。

另一方面，在现实应用中，很多地方都会直接读写cookie。读写cookie是一件麻烦的事，因为你必须要设置很多参数：`domain`、`path`、`httpOnly`...等很多参数。而操作`HttpSession`是一件相对简单的事。因此，*webx主张把一切对cookie的读写，都转换成对session的读写*。

8.3.1. 多值Cookie Store

8.3.1.1. 最简配置

例 8.8. 最基本的cookie配置（**/WEB-INF/webx.xml**）

```
<services:request-contexts xmlns="http://www.alibaba.com/schema/services/request-contexts">
    <buffered /> ❶
    <lazy-commit /> ❷
    ...
    <session>
        <stores>
            <session-stores:cookie-store id="temporaryClientStore">
                <session-stores:cookie name="tmp" />
            </session-stores:cookie-store>
        </stores>
        <store-mappings>
            <match name="*" store="temporaryClientStore" />
        </store-mappings>
    </session>
</services:request-contexts>
```

上面的配置创建了一个“临时”cookie（即随着浏览器关闭而清除），来作为默认的session对象的存储。

Cookie Store依赖其它两个Request Contexts：❶ <buffered> 和❷ <lazy-commit>。没有它们，就不能实现基于cookie的session。为什么呢？这要从HTTP协议谈起。下面是一个标准的HTTP响应的文本。无论你的服务器使用了何种平台（Apache HTTPD Server、Java Servlet/JSP、Microsoft IIS，……），只要你通过浏览器来访问，必须返回类似下面的HTTP响应：

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=AywiPrQKPEzfF9OZ; Path=/
Content-Type: text/html;charset=GBK
Content-Language: zh-CN
Content-Length: 48
Date: Mon, 06 Nov 2006 07:59:38 GMT


<html>
<body>
……
```

我们注意到，HTTP响应分为Header和Content两部分。从“`HTTP/1.1 200 OK`”开始，到“`<html>`”之前，都是HTTP *Header*，后面则为HTTP *Content*。而cookie是在header中指定的。一但应用服务器开始向浏览器输出content，那就再也没有机会修改header了。问题就出在这里。作为session的cookie可以在应用程序的任何时间被修改，甚至可能在content开始输出之后被修改。但是此后修改的session将不能被保存到cookie中。

Java Servlet API的术语称“应用服务器开始输出content”为“response被提交”。你可以通过`response.isCommitted()`方法来判断这一点。那么，哪些操作会导致response被提交呢？

- 向`response.getWriter()`或`getOutputStream()`所返回的流中输出，累计达到服务器所设定的一个chunk的大小，通常为`8K`。
- 用户程序或系统调用`response.flushBuffer()`。
- 用户程序或系统调用`response.sendError()`转到错误页面。
- 用户程序或系统调用`response.sendRedirect()`重定向。

只要避免上述情形的出现，就可以确保cookie可以被随时写入。前两个Request Contexts —— <buffered>和<lazy-commit>正好解决了上面的问题。第一个<buffered>将所有的输出到response.getWriter()或getOutputStream()的内容缓存在内存里，直到最后一刻才真正输出到浏览器；第二个<lazy-commit>拦截了response对象中引起提交的方法，将它们延迟到最后才执行。这样就保证了在cookie被完整写入之前，response绝不会被任何因素提交。

此外，<buffered>不是专为session框架而设计的。Webx的页面布局系统也依赖于这个Request Context。

8.3.1.2. Cookie的参数

例 8.9. Cookie的参数

```
<session-stores:cookie-store id="temporaryClientStore"
        maxLength="3896" maxCount="5" checksum="false">

    <session-stores:cookie name="tmp" domain="" path="/" maxAge="0" httpOnly="true"
        secure="false" survivesInInvalidating="false" />

</session-stores:cookie-store>
```

上例中列出了所有关于cookie的参数，解释如下：

表 8.5. Cookie的参数

参数名称	说明	
name	指定cookie的名称。 假设名称为“tmp”，那么将生成tmp0、tmp1、tmp2等cookie。 多个cookie stores的cookie名称不能重复。	
domain	指定cookie的域名。	这几个参数的默认值，均和Session ID cookie的设置相同。因此，*一般不需要特别设置它们*。
path	指定cookie的路径。	
maxAge	指定cookie的过期时间，单位是秒。 如果值为0，意味着cookie持续到浏览器被关闭（或称临时cookie）。 有效值必须大于0，否则均被认为是临时cookie。	
httpOnly	在cookie上设置HttpOnly标记。 在IE6及更新版本中，可以缓解XSS攻击的危险。	
secure	在cookie上设置Secure标记。 这样，只有在https请求中才可访问该cookie。	
survivesInInvalidating	这是一个特殊的设置。如果它被设置成true，那么当session被作废（invalidate）时，这个cookie store中的对象会幸存下来，并带入下一个新的session中。 如果这个值为true，必须同时设置一个大于0的maxAge。 这个设置有什么用呢？比如，我们希望在cookie中记录最近登录的用户名，以方便用户再次登录。可以把这个用户名记录在一个cookie store中，并设置survivesInInvalidating=true。即使用户退出登录，或当前session过期，新的session仍然可以读到这个store中所保存的对象。	
maxLength	指定每个cookie的最大长度。默认为3896，约3.8K。 Cookie store会把所有对象序列化到cookie中。但是cookie的长度是不能超过4K的。如果cookie的长度超过这个设定，就把数据分发到新的cookie中去。因此每个cookie store实际可能产生好几个cookie。	

参数名称	说明
	假设cookie name为`tmp`，那么所生成的cookie的名称将分别为：`tmp0`、`tmp1`、`tmp2`，以此类推。
`maxCount`	指定cookie的最大个数。默认为`5`。 因此，实际cookie store可生成的cookie总长度为：`maxLength * maxCount`。如果超过这个长度，cookie store将会在日志里面发出警告（`WARN`级别），并忽略store中的所有对象。
`checksum`	是否创建概要cookie。默认为`false`。 有时由于域名、路径等设置的问题，会导致cookie紊乱。例如：发现同名的cookie、cookie缺失等错误。这些问题很难跟踪。概要cookie就是为检查这类问题提供一个线索。如果把这个开关打开，将会产生一个概要性的cookie。假如cookie name为`tmp`，那么概要cookie的名字将是`tmpsum`。概要cookie会指出当前store共有几个cookie，每个cookie的前缀等内容。当cookie的总数和内容与概要cookie不符时，系统将会在日志中提出详细的警告信息（`DEBUG`级别）。 请尽量不要在生产系统中使用这个功能。

8.3.1.3. Session Encoders

Session里保存的是Java对象，而cookie中只能保存字符串。如何把Java对象转换成合法的cookie字符串（或者将字符串恢复成对象）呢？这就是Session Encoder所要完成的任务。

例 8.10. 配置Session Encoders

```
<session-stores:cookie-store>
    ...
    <session-stores:encoders>
        <session-encoders:encoder class="..." />
        <session-encoders:encoder class="..." />
        <session-encoders:encoder class="..." />
    </session-stores:encoders>
</session-stores:cookie-store>
```

和`SessionModelEncoder`类似，session框架也支持多个session encoders同时存在。

- 保存session数据时，session框架将使用第一个encoder来将对象转换成cookie可接受的字符串；
- 读取session数据时，session框架将依次尝试所有的encoders，直到解码成功为止。

这种编码、解码方案可让使用不同session encoders的系统之间共享cookie数据，也有利于平滑迁移系统。

Session框架提供了一种encoder的实现，编码的基本过程为：序列化、加密（可选）、压缩、Base64编码、URL encoding编码。

例 8.11. 配置Session Encoders的几种方案

- 基本配置：用hessian算法（默认）来序列化，不加密。

```
<session-stores:cookie-store>
    <session-stores:encoders>
        <session-encoders:serialization-encoder /> ❶
    </session-stores:encoders>
</session-stores:cookie-store>
```

❶ 这是默认实现。

- 用aes算法加密。AES算法可支持128、192、256位的密钥，默认为keySize=128。

```
<session-stores:cookie-store>
    <session-stores:encoders>
        <session-encoders:serialization-encoder>
            <session-serializers:hessian-serializer /> ❶
            <session-encrypters:aes-encrypter key="0123456789abcdef" /> ❷
        </session-encoders:serialization-encoder>
    </session-stores:encoders>
</session-stores:cookie-store>
```

❶ 也可以明确指定hession序列化。

❷ 添加AES加密算法，并提供密钥。

- 改用java原生的序列化算法。使用hessian算法（默认）可大幅缩短序列化的长度，但使用java原生的序列化算法，具有最好的可移植性。

```
<session-stores:cookie-store>
    <session-stores:encoders>
        <session-encoders:serialization-encoder>
            <session-serializers:java-serializer /> ❶
        </session-encoders:serialization-encoder>
    </session-stores:encoders>
</session-stores:cookie-store>
```

❶ 指定java序列化。

8.3.2. 单值Cookie Store

前面所描述的cookie store，是在一组cookie（如tmp0, tmp1, ...）中保存一组attributes的名称和对象。它所创建的cookie值，只有session框架自己才能解读。

假如有一些非webx的代码想要共享保存在cookie中的session数据，例如，Java Script代码、其它未使用webx框架的应用，希望能读取session数据，应该怎么办呢？Session框架提供了一种相对简单的“单值cookie store”可用来解决这个问题。顾名思义，单值cookie store就是在一个cookie中仅保存一个值或对象。

8.3.2.1. 最简配置

例 8.12. 单值cookie store基本配置

```
<session>
    <stores>
        ...
        <stores:single-valued-cookie-store id="loginNameCookie"> ❶
            <stores:cookie name="login" /> ❷
        </stores:single-valued-cookie-store>
    </stores>
    <store-mappings>
        ...
        <match name="loginName" store="loginNameCookie" /> ❸
    </store-mappings>
</session>
```

❶ 单值cookie store的ID是loginNameCookie。

❷ Cookie的名称是login。

❸ Session attribute的名称是`loginName`，attribute名称和cookie名称不必相同。

根据上面的配置，下面程序会生成cookie：`login=myname`。

例 8.13. 访问单值cookie的代码

```
session.setAttribute("LoginName", "myname");
```

需要注意的是，上述最简配置，只能用来存取字符串值。如果需要存取其它类型的对象，则需要配置Session Value Encoder。详见第 8.3.2.3 节 “Session Value Encoders”。

8.3.2.2. Cookie的参数

例 8.14. 单值cookie的参数配置

```
<session-stores:single-valued-cookie-store id="loginNameCookie">

    <session-stores:cookie name="login" domain="" path="/" maxAge="0" httpOnly="true"
        secure="false" survivesInvalidating="false" />

</session-stores:single-valued-cookie-store>
```

单值cookie的参数设置完全类似于普通cookie store的设置。唯一的差别是，单值cookie只生成一个cookie，而普通的cookie store则可能生成多个相关的cookies。

表 8.6. Cookie的参数

参数名称	说明	
name	指定cookie的名称。	
domain	指定cookie的域名。	这几个参数的默认值，均和Session ID cookie的设置相同。因此，*一般不需要特别设置它们*。
path	指定cookie的路径。	
maxAge	指定cookie的过期时间，单位是秒。 如果值为`0`，意味着cookie持续到浏览器被关闭（或称临时cookie）。 有效值必须大于`0`，否则均被认为是临时cookie。	
httpOnly	在cookie上设置`HttpOnly`标记。 在IE6及更新版本中，可以缓解XSS攻击的危险。	
secure	在cookie上设置`Secure`标记。 这样，只有在https请求中才可访问该cookie。	
survivesInInvalidating	这是一个特殊的设置。如果它被设置成`true`，那么当session被作废（invalidate）时，这个cookie store中的对象会幸存下来，并带入下一个新的session中。 如果这个值为`true`，必须同时设置一个大于`0`的`maxAge`。 这个设置有什么用呢？比如，我们希望在cookie中记录最近登录的用户名，以方便用户再次登录。可以把这个用户名记录在一个cookie store中，并设置`survivesInInvalidating=true`。即使用户退出登录，或当前session过期，新的session仍然可以读到这个store中所保存的对象。	

8.3.2.3. Session Value Encoders

单值cookie store可以保存任意的Java对象，只要这个Java对象能够被转换成字符串，以及从字符串中恢复。将Java对象转换成字符串，以及从字符串中恢复，就是Session Value Encoder的任务。和前面所说的Session Encoder不同，Session Value Encoder只转换session attribute的值，而Session Encoder需要转换一组session attributes的key-values。

例 8.15. Session Value Encoders的配置

```
<session-stores:single-valued-cookie-store>
    ...
    <session-stores:encoders>
        <session-value-encoders:encoder class="..." />
        <session-value-encoders:encoder class="..." />
        <session-value-encoders:encoder class="..." />
    </session-stores:encoders>
</session-stores:single-valued-cookie-store>
```

和SessionModelEncoder以及SessionEncoder类似，session框架也支持多个session value encoders同时存在。

- 保存session数据时，session框架将使用第一个encoder来将对象转换成cookie可接受的字符串；
- 读取session数据时，session框架将依次尝试所有的encoders，直到解码成功为止。

这种编码、解码方案可让使用不同session value encoders的系统之间共享cookie数据，也有利于平滑迁移系统。

目前有两种基本的session value encoders实现。<simple-value-encoder>和<mapped-values-encoder>。下面举例说明。

例 8.16. 配置Session Value Encoders的几种方案

- 编码字符串值，以指定的charset对字符串进行URL encoding。

```
<session-stores:encoders>
    <session-value-encoders:simple-value-encoder charset="GBK" /> ❶
</session-stores:encoders>
```

❶ 如不指定charset参数，默认charset为“UTF-8”。

- 编码指定类型的值，该值具有默认的PropertyEditor，可以转换成String，或从String中恢复。

```
<session-stores:encoders>
    <session-value-encoders:simple-value-encoder type="com.alibaba...MyEnum" /> ❶
</session-stores:encoders>
```

❶ Spring直接支持将Enum类型的值转成String类型，或反之。

- 编码指定类型的值，注册相应的registrar来进行类型转换。

```
<session-stores:encoders>
    <session-value-encoders:simple-value-encoder type="java.util.Date">
        <session-value-encoders:property-editor-registrar
            class="com.alibaba.citrus.service.configuration.support.CustomDateRegistrar"
            p:timeZone="GMT+8" p:format="yyyy-MM-dd" /> ❶
    </session-value-encoders:simple-value-encoder>
</session-stores:encoders>
```

❶ 注册registrar，将Date类型按格式yyyy-MM-dd转成字符串，或从该格式的字符串中恢复。

- 在上面例子的基础上，可增加encrypter，对value进行加密。

```
<session-stores:encoders>
    <session-value-encoders:simple-value-encoder type="java.util.Date">
        <session-value-encoders:property-editor-registrar
            class="com.alibaba.citrus.service.configuration.support.CustomDateRegistrar"
            p:timeZone="GMT+8" p:format="yyyy-MM-dd" />

        <session-encrypters:aes-encrypter key="0123456789abcdef" /> ❶

    </session-value-encoders:simple-value-encoder>
</session-stores:encoders>
```

❶ 用AES和指定密钥进行加密。

- `<mapped-values-encoder>`和`<simple-value-encoder>`类似，差别在于，前者只接受`java.util.Map`数据类型，并将其编码成“`key:value&key:value`”的格式。下面的例子可接受`Map<String, Date>`类型的数据：

```
<session-stores:encoders>
    <session-value-encoders:mapped-values-encoder valueType="com.alibaba...MyEnum" /> ❶
</session-stores:encoders>
```

❶ 注意此处所指定的类型为map中的value的类型。

当你用下面的代码，可设置cookie值“`key1:value1&key2:value2`”：

```
Map<String, MyEnum> mappedValue = new HashMap<String, MyEnum>();

mappedValue.put("key1", MyEnum.value1);
mappedValue.put("key2", MyEnum.value2);

session.setAttribute("cookie", mappedValue); ❶
```

❶ 将整个map作为session attribute的值，其中，map的value类型必须符合配置文件中指定的类型。

- 类似的，你同样可以对`<mapped-values-encoder>`指定registrar和encrypter，不再赘述。

8.4. 其它Session Store

8.4.1. Simple Memory Store

`SimpleMemoryStore`是最简单的session store。它将所有的session对象都保存在内存里面。这种store不支持多台机器的session同步，而且也不关心内存是否被用尽。因此这种简单的store一般只应使用于测试环境。

例 8.17. 配置simple memory store

```
<stores>
    <session-stores:simple-memory-store id="simple" />
</stores>
```

警告

鉴于`simple-memory-store`的实现的简单性，*请不要将它应用在生产环境*。

8.5. 本章总结

Session是个难题，特别是对于要求高扩展性和高可用性的网站来说。

我们在标准的Java Servlet API的基础之上，实现了一套全新的session框架。在此基础上可以进一步实现多种session的技术，例如：基于cookie的session、基于数据库的session、基于Berkeley DB的session、基于内存的session，甚至也可以实现基于TCP-ring的session等等。最重要的是，我们能把这些技术结合起来，使每种技术的优点能够互补，缺点可以被避免。

所有这一切，对应用程序是完全透明的 —— 应用程序不用知道session是如何实现的、它们的对象被保存到哪个session store中等问题 —— session框架可以妥善地处理好这一切。

部分 III. Webx应用支持服务

第 9 章 表单验证服务指南

9.1. 表单概述

9.1.1. 什么是表单验证

在WEB应用中，表单验证是非常重要的一环。表单验证，顾名思义，就是确保用户所填写的数据符合应用的要求。例如下面这个“注册新帐户”的表单：

图 9.1. 一个典型的表单页面

在这个表单中，各字段需要符合以下规则：

表 9.1. 注册新帐户的规则

字段名	规则
用户名	必须由字母、数字、下划线构成。 · 用户名的

字段名	规则
	长度必须在某个范围内，例如，4-10个字符。
密码	长度必须在某个范围内，例如，4-10个字符。
	密码和用户名不能相同，以保证基本的安全性。
确认密码（再输一遍密码）	必须和密码相同，确保用户没有打字错误。

从技术上讲，表单验证完全可以用手工书写代码的方式来实现。但是这样做既无趣，又容易出错，而且难以维护 —— 特别是当你需要修改验证规则时。因此，几乎所有的WEB框架都提供了表单验证的功能，使你能方便、快速地书写或修改表单验证的规则。

9.1.2. 表单验证的形式

验证WEB页面中的表单有如下几种形式：

9.1.2.1. 服务端批量验证

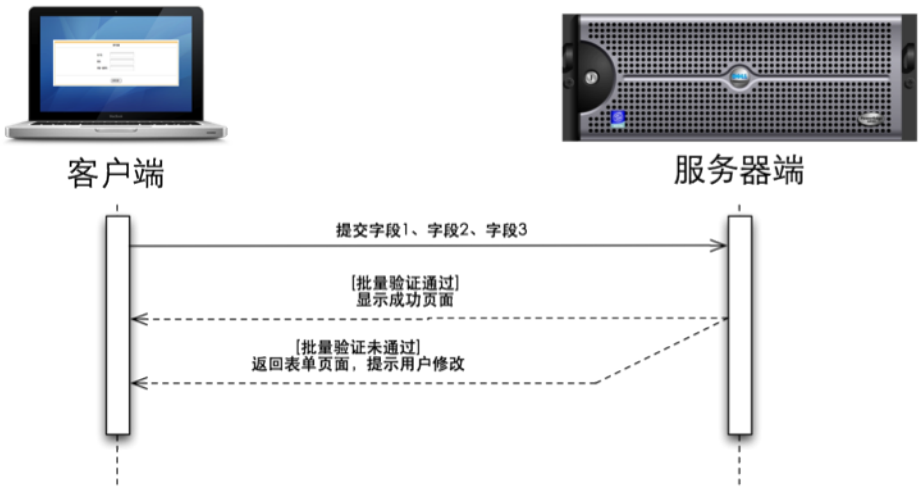


图 9.2. 服务端批量验证

服务端批量验证是最传统验证形式，它将所有表单字段一次性提交给服务器来验证。服务器对所有表单进行批量的验证后，根据验证的结果跳转到不同的结果页面。

9.1.2.2. 客户端验证

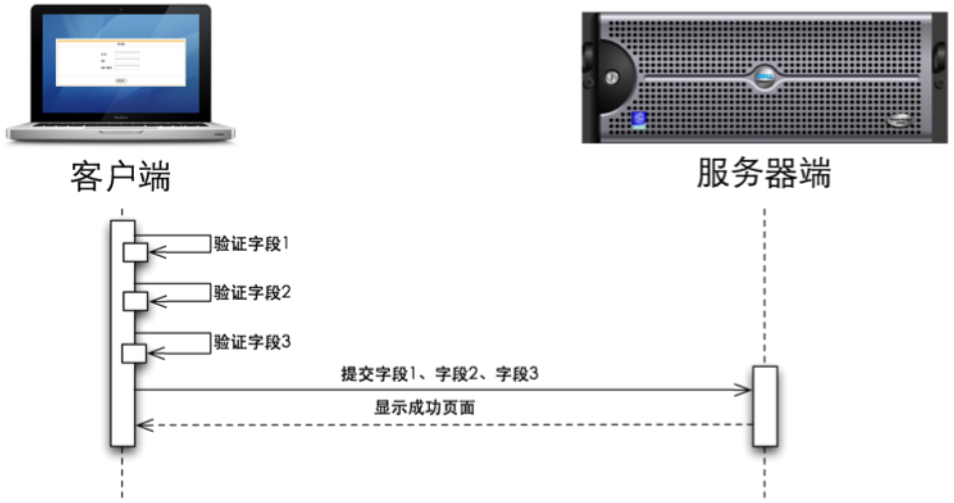


图 9.3. 客户端验证

客户端验证是利用Java Script对用户输入的数据进行逐个验证。

9.1.2.3. 服务端异步验证

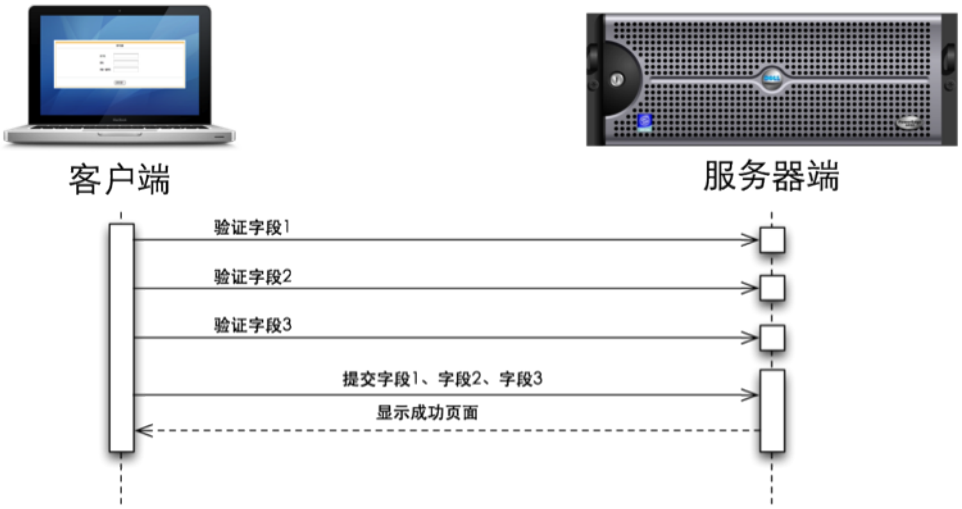


图 9.4. 服务端异步验证

服务器异步验证是利用Java Script发出异步AJAX请求，来要求服务器验证单个或多个字段。如果网络延迟不明显，那么服务器异步验证给用户的体验类似于客户端验证。

9.1.2.4. 混合式验证

以上几种验证手段各有优缺点：

表 9.2. 各种表单验证的优缺点比较

验证形式	功能性	网络负荷	用户体验	简单性	可靠性
服务端批量验证	*强*。 由于验证逻辑存在于服务器上，可访问服务器的一切资源，功能最强。	高。 当用户填错任意一个字段时，所有的字段都必须在浏览器和服务器之间来回传输一次。所以它会给网络传输带来较高的负荷。	差。 由于网络负荷较高，造成的响应迟缓。此外，验证失败时必须整个页面被刷新。这些会给用户带来不好的体验。	*简单*。 它的实现比较简单。	*可靠*。 相对于其它几种方式，服务端批量验证也是最可靠的方式。因为Java Script可能会失效（因为浏览器不支持、Java Script被关闭、网站受攻击等原因），但服务器批量验证总不会失效。
客户端验证	弱。 由于验证逻辑存在于用户浏览器上，不能访问服务器资源，因此有一些功能无法实现，例如：检查验证码、确认注册用户ID未被占用等。	*无*。 在验证时，不需要网络通信，不存在网络负荷。	*好*。 响应速度极快，用户体验最好。	复杂。 因为需要JS编程。	不可靠。 由于下列原因，Java Script可能会失效，使得客户端验证被跳过： • 浏览器不支持 • Java Script被关闭 • 网站受攻击
服务端异步验证	*强*。 由于验证逻辑存在于服务器上，可访问服务器的一切资源，功能也很强。	低。 每次验证，只需要发送当前被验证字段的数据即可，网络负荷较小。	较好。 由于网络负荷小，用户响应远快于服务端批量验证，用户体验好。		

没有一种验证方法是完美的。但把它们结合起来就可以克服各自的缺点，达到较完美的境地：

- 对所有字段做服务器端批量验证，即便Java Script失效，服务器验证可作为最后的防线。
- 只要有可能，就对字段做客户端验证，确保最迅速的响应和较好的用户体验。
- 对于必须访问服务器资源的验证逻辑，例如检查验证码、确认注册帐户ID未被占用等，采用服务器异步验证，提高用户体验。

以上混合形式的验证无疑是好的，但是它的实现也比较复杂。

目前Webx所提供的表单验证服务并没有实现客户端验证和服务端异步验证。这些功能将在后续版本中实现。在现阶段中，应用开发者必须手工编码Java Script来实现客户端验证和服务端异步验证。

9.2. 设计

9.2.1. 验证逻辑与表现逻辑分离

很容易想到的一种表单验证的实现，就是将表单验证的逻辑内嵌在页面模板中。例如，某些WEB框架实现了一些用来验证表单的JSP tags。类似下面的样子：

例 9.1. 将验证逻辑内嵌在页面模板中

```
<input type="text" name="loginId" value="${loginId}" />
<form:required value="${loginId}">
  <strong>Login ID is required.</strong>
</form:required>
<form:regexp value="${loginId}" pattern="^\w+$">
  <strong>Login ID is invalid.</strong>
</form:regexp>
```

将验证逻辑内嵌在页面模板中最大的问题是，验证逻辑和页面的表现逻辑完全混合在一起。当你需要修改验证规则时，你必须找出所有的页面，从复杂的HTML代码中，一个不漏地找到并修改它们。这是一件费时费力的工作，而且很容易出错。另一方面，嵌在页面模板中的验证规则是不能被多个页面共享和复用的。

Webx表单验证服务主张验证逻辑和页面表现逻辑完全分离。所有的验证规则都写在一个单独的配置文件中 —— 页面模板是不需要关心这些验证规则的。当你需要修改验证规则时，只需要修改独立的配置文件就可以了，并不用修改页面模板。

9.2.2. 验证逻辑和应用代码分离

另一种容易想到的方法，是把表单验证的逻辑写在Java代码中。例如，在Java代码中直接调用验证逻辑。更高级一点，也许可以通过annotation机制在Java代码中定义验证逻辑，像下面的样子：

例 9.2. 将验证逻辑内嵌在Java代码中

```
public class LoginAction {
    @Required
    @Regexp("^\\w+$")
    private String loginId;
    …
}
```

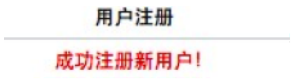
这样做的问题是，当你需要修改验证规则时，你必须一个不漏地找到所有定义annotations的那些代码，并修改它们。另一方面，annotation机制不容易扩展，很难方便地增加新的验证方案。

Webx表单验证服务主张*验证逻辑和应用代码完全分离*。所有的验证规则都写在一个单独的配置文件中 —— 应用程序的代码是不需要关心这些验证规则的。当你需要修改验证规则时，只需要修改独立的配置文件就可以了，并不需要修改程序代码。

9.2.3. 表单验证的流程

表 9.3. 一个基本的表单验证流程

步骤	客户端浏览器	WEB服务器	页面效果
1.	请求表单页面 →		
		← 返回空白表单	

步骤	客户端浏览器	WEB服务器	页面效果
2.	用户填写表单，并提交 →		
		← 验证表单数据，如果验证有错，则返回包含错误信息的表单页面，并提示出错信息。	
3.	用户修改表单，并再次提交（重复该步骤直至验证成功） →		 成功注册新用户!
		← 验证表单数据，如果验证通过，则转至下一个页面。通常是显示成功信息。	

9.3. 使用表单验证服务

Webx表单验证服务可用来支持以下几种类型的表单需求：

表 9.4. 几种表单需求

需求名称	说明
创建新数据	也就是让用户在一个空白的表单上填写数据，并验证之。例如，注册一个新帐户。
修改老数据	也就是让用户在已填有数据的表单上进行修改，并验证之。例如，修改帐户信息。
批量创建、修改数据	也就是在一个表单中，一次性创建、修改多个数据对象。例如，管理员批量审核帐户。

9.3.1. 创建新数据

下面的例子实现了“注册一个新帐户”的功能。

为了实现表单验证的功能，需要由三个部分配合起来工作：

表 9.5. 验证表单所需的部件

部件名称	说明
验证规则	也就是form service的配置文件。
页面模板	通过`$form`工具，生成表单页面。
Java代码	接收表单数据，并作后续处理。

下面逐个介绍。

9.3.1.1. 定义验证规则

表单验证服务是一个基于Spring和Spring Ext的服务，可利用Schema来配置。示例如下：

例 9.3. 表单验证规则示例

```
<beans:beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:services="http://www.alibaba.com/schema/services"
    xmlns:fm-conditions="http://www.alibaba.com/schema/services/form/conditions"
    xmlns:fm-validators="http://www.alibaba.com/schema/services/form/validators"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="
        http://www.alibaba.com/schema/services
            http://localhost:8080/schema/services.xsd
        http://www.alibaba.com/schema/services/form/conditions
            http://localhost:8080/schema/services-form-conditions.xsd
        http://www.alibaba.com/schema/services/form/validators
            http://localhost:8080/schema/services-form-validators.xsd
        http://www.springframework.org/schema/beans
            http://localhost:8080/schema/www.springframework.org/schema/beans/spring-beans.xsd
    ">

    <services:form xmlns="http://www.alibaba.com/schema/services/form/validators"> ❶
        <services:group name="register"> ❷

            <services:field name="userId" displayName="登录名"> ❸
                <required-validator> ❹
                    <message>必须填写 ${displayName}</message> ❺
                </required-validator>
                <regexp-validator pattern="^[A-Za-z_][A-Za-z_0-9]*$">
                    <message>${displayName} 必须由字母、数字、下划线构成</message>
                </regexp-validator>
                <string-length-validator minLength="4" maxLength="10">
                    <message>${displayName} 最少必须由${minLength}个字组成，最多不能超过${maxLength}个字</
message>
                </string-length-validator>
            </services:field>

            <services:field name="password" displayName="密码">
                <required-validator>
                    <message>必须填写 ${displayName}</message>
                </required-validator>
                <string-length-validator minLength="4" maxLength="10">
                    <message>${displayName} 最少必须由${minLength}个字组成，最多不能超过${maxLength}个字</
message>
                </string-length-validator>
                <string-compare-validator notEqualTo="userId">
                    <message>${displayName} 不能与 ${userId.displayName} 相同</message>
                </string-compare-validator>
            </services:field>

            <services:field name="passwordConfirm" displayName="密码验证">
                <required-validator>
                    <message>必须填写 ${displayName}</message>
                </required-validator>
                <string-compare-validator equalTo="password">
                    <message>${displayName} 必须和 ${password.displayName} 相同</message>
                </string-compare-validator>
            </services:field>

        </services:group>

    </services:form>

</beans:beans>
```

❶ <form>代表表单验证服务的配置。从这里开始定义表单验证的规则。

❷ 可以定义多个groups，每个group有一个唯一的名称，例如：“register”。每个group代表了一组需要验证的字段（field）。

❸ 每个field有一个在组中唯一的名称，例如：“userId”、“password”等。

❹ 每个field又包含了多个验证规则（validator）。

❺ 每个验证规则都包含了一段文字描述（message），如果用户填写的数据没有通过当前的规则的验证，那么用户将会看到这段文字描述，以解释出错的原因。

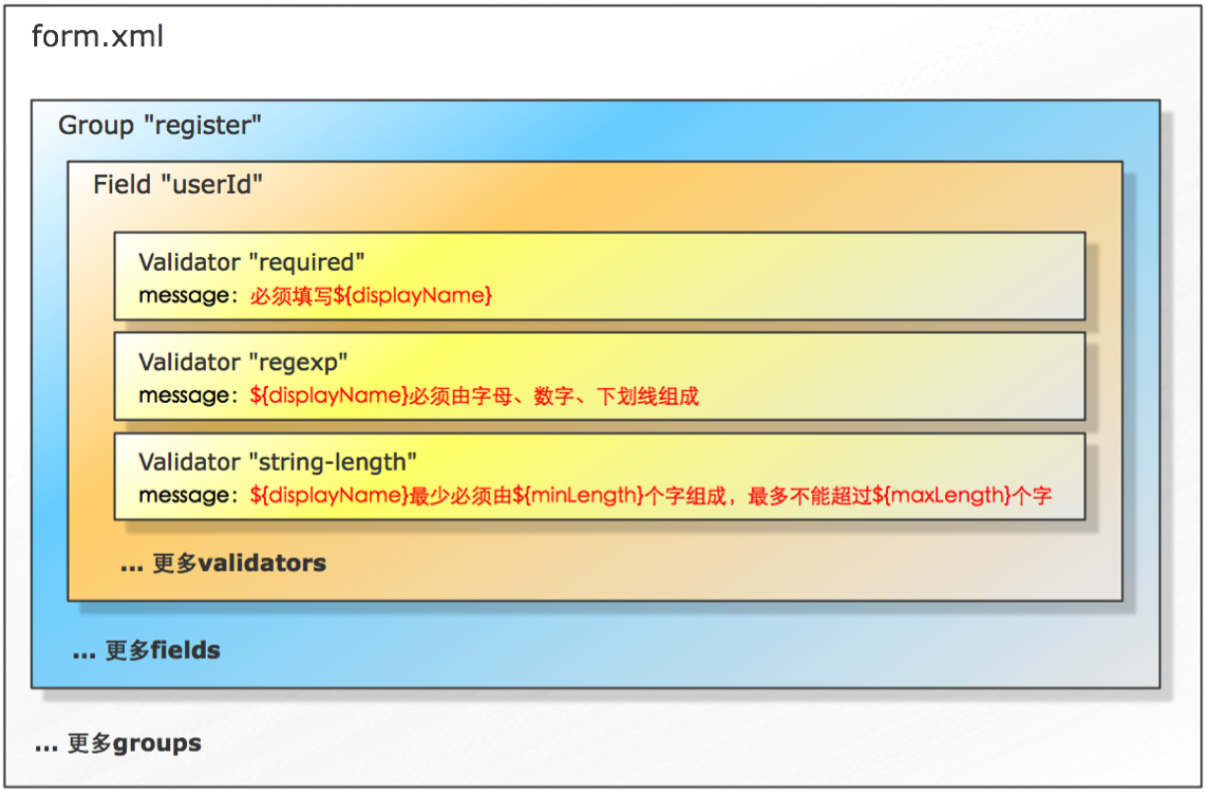


图 9.5. 表单验证配置文件的结构

9.3.1.2. 创建表单页面

创建表单页面需要使用一个pull tool工具，配置如下：

例 9.4. 表单验证pull tool的配置

```
<services:pull xmlns="http://www.alibaba.com/schema/services/pull/factories">
    <form-tool />
    ...
</services:pull>
```

上面的配置定义了一个$form工具。现在你可以在模板中直接使用它。

例 9.5. 表单验证的页面模板示例

```
#macro (registerMessage $field) ❶
    #if (!$field.valid) $field.message #end
#end

<form action="" method="post"> ❷
  <input type="hidden" name="action" value="UserAccountAction"/> ❸

  #set ($group = $form.register.defaultInstance) ❹

  <p>用户注册</p>

  <dl>
    <dt>用户名</dt>
    <dd>
        <div>
            <input type="text" name="$group.userId.key" value="$!group.userId.value"/> ❺
        </div>
        <div class="errorMessage">
            #registerMessage ($group.userId) ❻
        </div>
    </dd>

    <dt>密码</dt>
    <dd>
        <div>
            <input type="password" name="$group.password.key" value="$!group.password.value"/> ❼
        </div>
        <div class="errorMessage">
            #registerMessage ($group.password) ❽
        </div>
    </dd>

    <dt>再输一遍密码</dt>
    <dd>
        <div>
            <input type="password" name="$group.passwordConfirm.key" value="$!
group.passwordConfirm.value"/> ❾
        </div>
        <div class="errorMessage">
            #registerMessage ($group.passwordConfirm) ❿
        </div>
    </dd>
  </dl>

  <p>
      <input type="submit" name="event_submit_do_register" value="立即注册！"/> ⓫
  </p>

</form>
```

❷ HTML form的action值为空，意思是把表单提交给当前页面。

这样，当用户填写表单有错时，应用会停留在当前表单页面，将表单数据连同错误提示一起显示给用户，要求用户修改。如果表单验证通过，应用必须通过重定向操作来转向下一个页面。

❹ 创建一个`register` group的实例。

❺❼❾ 利用新创建的group对象来生成表单字段，包括生成字段的名称`$group.field.key`，以及字段的值为`$!group.field.value`。

❶ 定义velocity宏：仅当field验证通过时（即`$group.field.valid=true`），才显示错误信息。

对于空白表单和通过验证的字段而言，`$group.field.valid`为`true`。

❻❽❿ 如果验证失败的话，显示验证出错消息。这里通过前面所定义的velocity宏来简化代码。

❸ 根据这参数，表单将会被交给`UserAccountAction`来处理。Action的职责是调用表单验证过程。假如验证通过，就保存数据，并重定向到下一个页面。

⓫ 根据这个参数，表单被提交以后，系统会调用当前action（即`UserAccountAction`）的`doRegister()`方法。每个action类中，可以包含多个处理数据的动作，例如`doCreate`、`doUpdate`、`doDelete`等。

上面的Velocity页面模板演示了怎样利用表单验证服务创建一个帐户注册的HTML表单。关键技术解释如下：

创建group实例

`$form.register.defaultInstance`将会对`register` group创建一个默认的实例。绝大多数情况下，只需要创建唯一的default instance就足够了。但后面我们会讲到创建多实例的例子。

所创建的group instance（如`register`）必须先在规则配置文件中被定义。

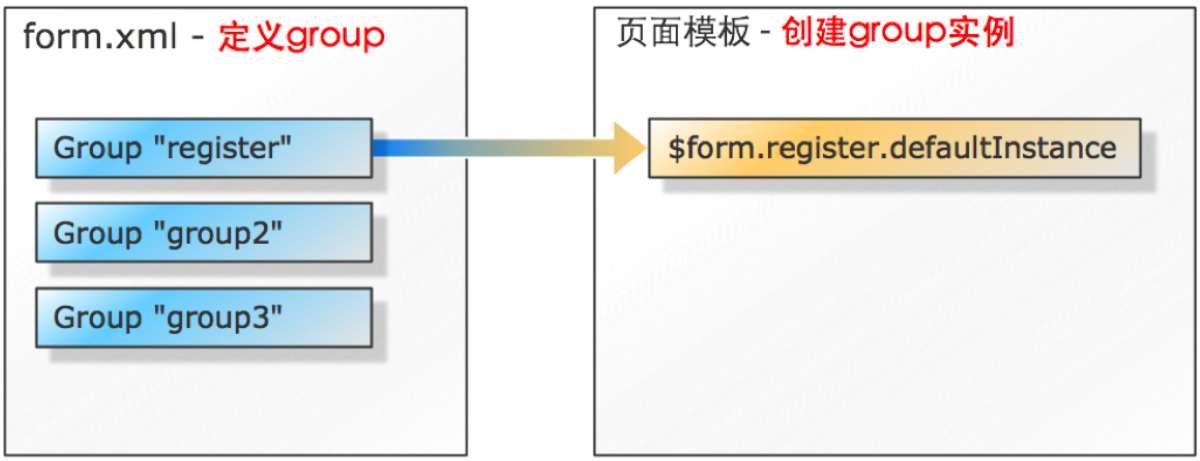


图 9.6. 创建一个group实例

生成表单字段

一个表单字段包含名称和值两个部分。

字段的名称为`$group.field.key`。表单验证服务会自动生成一个字段名。这个字段名被设计成仅供系统内部解读的，而不是让外界的系统或人来解读的。它看起来是这个样子的：“`_fm.r._0.p`”。*外界的系统不应该依赖于这个名称*。

字段的值为`$!group.field.value`。它的初始值（即用户填写数据之前）是`null`。但你也可以在配置文件中为它指定一个默认的初始值，例如：

例 9.6. 在表单验证规则中添加默认值

```
<services:field name="myfield" defaultValue="mydefault" ...>
```

因为值可能是null，因此在velocity中，需要以“$!”来标记它 —— Velocity认为null是一个错误，除非你以$!来标记它，告诉velocity忽略它。

需要注意的是，*默认值只会影响field的初始值*。一旦用户填写并提交了表单，那么$group.field.value的值将保持用户所填写的值不变 —— 不论验证失败或成功。

页面展现

一般来说，你需要定义CSS风格以便让表单的field和错误信息能以适当的格式来显示给用户。展现效果可能是像这个样子：

图 9.7. 在页面中显示表单验证错误信息

表单系统不应该干预页面的具体展现方法，以下内容均和表单系统无关。例如：

- Field展现的方式：textbox、checkbox、hidden field?
- 错误信息的颜色、显示位置。

9.3.1.3. 创建Java代码（action）

用户提交表单后，由服务器端的Java代码读取并验证用户的数据。

在Webx中，这个功能通常由action来完成。前文已经提到，在HTML表单中，设置action字段，以及event_submit_do_register提交按钮，就可以让Webx框架调用UserAccountAction.doRegister()方法。

下面是UserAccountAction类的实现代码：

例 9.7. 创建用于处理提交数据的action代码

```
public class UserAccountAction {
    @Autowired
    private FormService formService; ❶

    public void doRegister(Navigator nav) throws Exception {
        Form form = formService.getForm(); ❷

        if (form.isValid()) { ❸
            Group group = form.getGroup("register"); ❹

            MyUser user = new MyUser(); ❺
            group.setProperties(user); ❻
            save(user);

            // 跳转到注册成功页面
            nav.redirectTo("registerSuccess"); ❼
        }
    }
}
```

❶ 注入form服务。

❷ 取得form对象，form对象中包含若干groups。

❸ 仅当表单验证成功时，才执行下去。

❹ 取得group对象。Group对象的名称必须和配置文件以及模板中的group名称相同。

❺❻ 将group中的数据灌入bean中。

❼ 处理完数据以后，利用Webx navigation接口跳转到“注册成功”页面。

例子中的`MyUser`对象是一个简单的Java Bean：

例 9.8. 被灌入group数据的Java Bean

```
public static class MyUser {
    private String userId;
    private String password;

    public String getUserId() {
        return userId;
    }

    public void setUserId(String userId) {
        this.userId = userId;
    }

    public String getPassword() {
        return password;
    }

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

`Group.setProperties()`方法将fields的值映射到同名的Java Bean properties中。然而这个对应关系是可以改变的，后文会再次讲到该问题。

是不是有点复杂？事实上，*上面的代码可以通过Webx的参数注入机制加以简化*。下面的代码可以完成完全相同的功能，但是代码却短得多。然而，*理解前面的较复杂代码，将有助于你理解下面的简化代码*。

例 9.9. 创建用于处理提交数据的action代码（Annotations简化版）

```
public class UserAccountAction {
    public void doRegister(@FormGroup("register") MyUser user, Navigator nav) throws Exception {
        save(user);
        nav.redirectTo("registerSuccess");
    }
}
```

在这个简化版代码中，@FormGroup注解完成了前面复杂代码中的大部分功能，包括：

- 验证表单，如果*失败则不执行action*，否则执行`doRegister`方法。
- 取得form和`register` group对象，并将group中的数据注入到`MyUser`对象中。

9.3.2. 修改老数据

在前面的例子中，我们利用表单创建了一个新数据 —— 注册新帐户。它是从一个空白表单开始的，也就是说，在用户填写表单之前，表单是没有内容的，或只包含默认值的。另一种常见情况是修改老数据。例如“修改帐户资料”。和创建新数据的例子不同，在用户填写表单之前，表单里已经包含了从数据库中取得的老数据。

在创建新数据的模板和代码中，稍微添加一点东西，就可以实现修改老数据的功能。

9.3.2.1. 用screen来读取数据

修改老数据的第一步，是要取得老的数据。例如，取得要修改的帐户信息。在Webx中，这个任务是由screen来完成的：

例 9.10. 用screen取得表单验证的初始数据

```
public class UserAccount {
    @Autowired
    private UserManager userManager; ❶

    public void execute(Context context) throws Exception {
        User user = userManager.getUser(getCurrentUser().getId());
        context.put("user", user); ❷
    }
}
```

❶ `UserManager`是一个业务接口。通过它，可以从数据库中取得当前登录帐户的信息。

❷ 随后，screen代码把所取得的user对象放到context中，这样，就可以在模板中用`$user`来引用它。

9.3.2.2. 表单页面

例 9.11. 用来修改数据的页面模板

```
#set ($group = $form.userAccount.defaultInstance)

$group.mapTo($user) ❶
...
<input type="hidden" name="$group.userId.key" value="$!group.userId.value"/> ❷
...

<input type="text" name="$group.lastName.key" value="$!group.lastName.value"/>
...
#userAccountMessage ($group.lastName)
...
<input type="submit" name="event_submit_do_update" value="修改"/> ❸
```

在前面“创建新数据”的页面上，加上和修改一点内容：

❶ mapTo的功能是填充表单。

这行代码的意思是：用screen中所取得的user对象的值来填充表单，作为表单的初始值。和Group.setProperties()方法相反，mapTo将Java Bean properties的值映射到同名的fields中。

❷ 保存主键。

和创建新数据不同，在修改老数据时，一般需要在表单中包含主键。这个主键（user id）在数据库中唯一识别这一数据对象（user）。

应该避免用户改变主键。最简便的方法，就是用hidden字段来保存主键。

❸ 这个submit按钮将引导webx执行UserAccountAction.doUpdate方法。

需要注意的是，调用mapTo在下列情况下是无效的：

- 当$user对象不存在（值为null）时，mapTo不做任何事。这样，*你就可以让“创建新帐户”和“修改帐户信息”共用同一个模板*。在新表单中，由于$user不存在，所以mapTo失效；而在更新表单中，就可以从$user中取得初始的数据。
- 当用户提交表单以后，mapTo不做任何事。因为mapTo只会影响表单的初始数据。一旦用户修改并提交数据以后，mapTo就不会改变用户所修改的数据。

9.3.2.3. 用action来处理数据

修改老数据的action代码和创建新数据的action代码几乎相同，而且它们可以共享同一个UserAccountAction类：

例 9.12. 用来保存提交数据的action

```
public class UserAccountAction {
    public void doRegister(...) throws Exception {
        ...
    }

    public void doUpdate(@FormGroup("userAccount") MyUser user, ❶
                         Navigator nav) throws Exception {
        save(user);
        nav.redirectTo("updateSuccess");
    }
}
```

❶ 通过annotation取得的MyUser对象中，包含了通过hidden字段传过来的user id，以及其它所有字段的值。

9.3.3. 批量创建或修改数据

有时，我们需要在一个表单页面中批量创建或修改一批数据。例如，后台管理界面中，管理员可以一次审核10个帐户的信息。每个帐户的信息格式都是相同的：姓名、性别、年龄、地址等。表单验证服务完全支持这样的表单。

9.3.3.1. 用screen来读取批量数据

假如你希望做的是批量修改数据，很显然，你需要在screen代码中取得所有需要修改的数据。

例 9.13. 批量读取数据的screen

```
public class BatchUserAccount {
    @Autowired
    private UserManager userManager;

    public void execute(Context context) throws Exception {
        List<User> users = userManager.getUsers(getIds()); ❶
        context.put("users", users);
    }
}
```

❶ 和修改单个数据的screen代码不同的是，你需要一次性读取多个数据对象，并置入到context中。

9.3.3.2. 表单页面

例 9.14. 批量创建、修改数据的表单页面模板

```
<form action="" method="post">
  <input type="hidden" name="action" value="UserAccountAction"/>

  #foreach($user in $users) ❶
    #set ($group = $form.userAccount.getInstance($user.id)) ❷

    $group.mapTo($user)

    ...
    <input type="hidden" name="$group.userId.key" value="$!group.userId.value"/>

    ...
    <input type="text" name="$group.lastName.key" value="$!group.lastName.value"/>
    ...
    #userAccountMessage ($group.lastName)
    ...

  #end
  ...
  <input type="submit" name="event_submit_do_batch_edit" value="批量修改"/> ❸
</form>
```

❶ 为了批量创建、修改数据，需要在表单页面中利用foreach循环来遍历数据对象。其中，$users是由screen放入context中的对象列表。

❷ 对每个数据对象创建一个group实例。

❸ 指定action事件。这个submit按钮将引导webx执行UserAccountAction.doBatchEdit方法。

在前面的例子中，我们一直使用$form.xyz.defaultInstance来创建默认的group实例。而这里，我们改变了用法：$form.userAccount.getInstance($user.id)。每次调用该方法，就对一个group生成了一个实例（instance）。

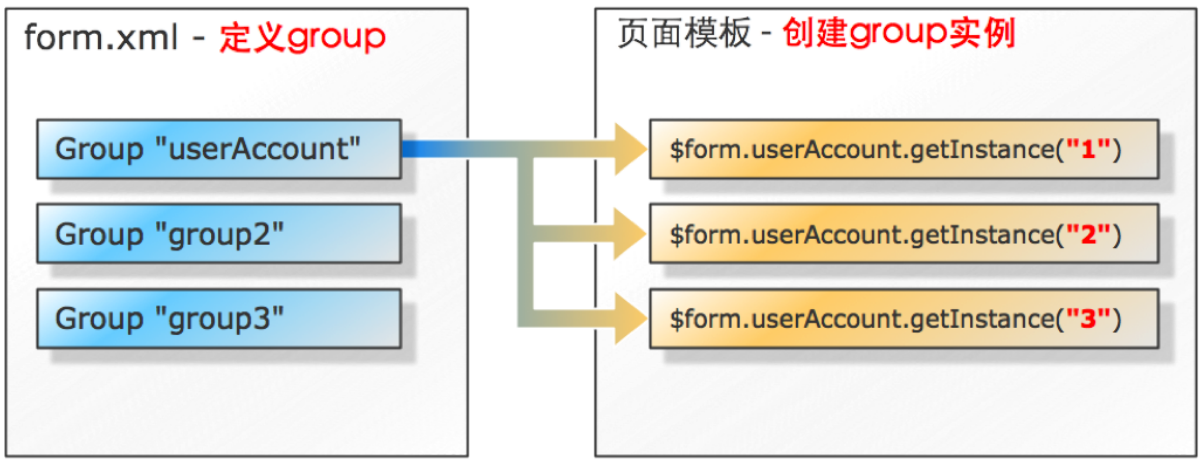


图 9.8. 创建多个group实例

每个instance必须以不同的id来区分。最简单的方法，就是采用数据对象的唯一id来作为group instance的id。在这个例子中，我们采用$user的唯一id（$user.id）来区分group instances。

前文讲过，default instance的field key是这个样子的：“_fm.r._0.p”。类似的，通过getInstance("myid")方法所取得的group中的field key是这样的：“_fm.u.myid.n”。很明显，form service就是依赖于field key中所包含的group instance id来区分同一group的不同instances的。

因为field key将作为HTML的一部分，所以group instance的id必须为满足下面的条件：*只包含英文字母、数字、下划线、短横线的字符串*。

页面的其它部分和创建、修改单个数据的代码完全相同。只不过它们被循环生成了多次。 最后的结果是类似下面的样子：

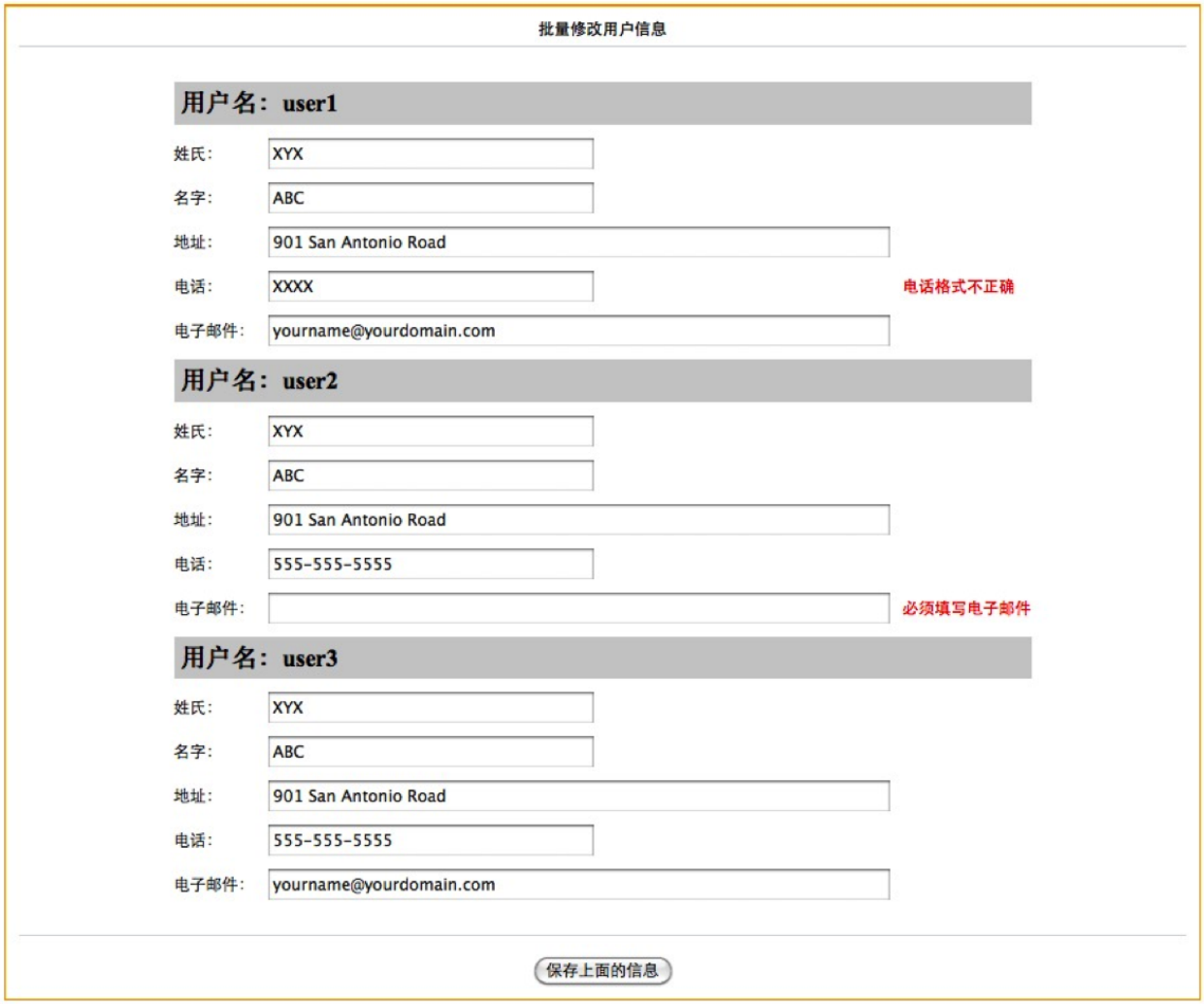


图 9.9. 批量修改数据的页面示例

9.3.3.3. 用action来处理数据

和前面的例子类似，我们先用传统的方法来写action以便阐明原理，再用annotation来简化action代码。

例 9.15. 用来批量处理数据的action

```
public class UserAccountAction {
    @Autowired
    private FormService formService;

    public void doBatchEdit(Navigator nav) throws Exception {
        Form form = formService.getForm();

        if (form.isValid()) {
            Collection<Group> groups = form.getGroups("userAccount"); ❶

            for (Group group : groups) {
                MyUser user = new MyUser();

                group.setProperties(user);
                save(user);
            }

            nav.redirectTo("success");
        }
    }
}
```

❶ 通过这个方法，可以取得所有名称为“userAccount”的group instances，包括：user1、user2、……。

取得group实例，除了例子中的form.getGroups(groupName)这种形式以外，还有以下几种变化：

- 取得所有的group instances，无论其名称是什么：Collection<Group> groups = form.getGroups();
- 取得指定group名称和instance key的group instances：Group user1Group = form.getGroup("userAccount", "user1");

下面是一个简化版的action，实现完全相同的功能。

例 9.16. 用来批量处理数据的action（Annotation简化版）

```
public class UserAccountAction {
    public void doBatchEdit(@FormGroup("userAccount") MyUser[] users,
                            Navigator nav) throws Exception {
        for (MyUser user : users) {
            save(user);
        }

        nav.redirectTo("success");
    }
}
```

9.4. 表单验证服务详解

9.4.1. 配置详解

9.4.1.1. 基本配置

例 9.17. 表单验证服务的基本配置

```
<services:form xmlns="http://www.alibaba.com/schema/services/form/validators"> ❶

    <services:group name="group1"> ❷
        <services:field name="field1"> ❸
            <validator /> ❹
            <validator />
            ...
        </services:field>

        <services:field name="field2" />
        ...
    </services:group>

    <services:group name="group2">
        ...
    </services:group>

    ...
</services:form>
```

❶ 开始配置表单验证服务。

❷ 每个表单验证服务可包含多个groups。

❸ 每个group可包含多个fields。

❹ 每个field可包含多个validators。

9.4.1.2. Post Only参数

例 9.18. 配置Post Only参数

```
<services:form postOnlyByDefault="true"> ❶
    <services:group name="group1" postOnly="true" /> ❷
</services:form>
```

❶ 如果不指定，`postOnlyByDefault`的默认值为`true`。

❷ 如果不指定，那么`postOnly`的值取决于`postOnlyByDefault`。这意味着如果什么也不设置，所有`postOnly`的实际值均为`true`。

如果一个group被设置成`postOnly=true`，那么，这个group将不接受通过GET方法提交的数据，只允许通过POST方式提交数据。这样可以略略增加系统的安全性，增加CSRF攻击的难度。

9.4.1.3. Trimming参数

例 9.19. 配置Trimming参数

```
<services:form>
    <services:group name="group1" trimmingByDefault="true"> ❶
        <services:field name="field1" trimming="true" /> ❷
    </services:group>
</services:form>
```

❶ 如果不指定，`trimmingByDefault`的默认值为`true`。

❷ 如果不指定，`trimming`的值取决于`trimmingByDefault`。这意味着如果什么也不设置，所有`trimming`的实际值均为`true`。

用户所提交的字符串数据中，两端的空白往往是无意义的。这些空白可能会影响验证规则的准确性。

如果设置了`trimming=true`参数，那么表单系统可以自动剪除字段值两端的空白字符，例如把“ my name ”（两端有空白）转变成“my name”（两端无空白）。

9.4.1.4. Display Name参数

例 9.20. 配置Display Name参数

```
<services:field name="field1" displayName="我的字段"> ❶
    <required-validator>
        <message>必须填写${displayName}</message> ❷
    </required-validator>
</services:field>
```

❶ 如果未指定`displayName`，那么其默认为field名称。也就是的“`field1`”。

❷ 在validator message中，可以引用`${displayName}`。这样做的好处是，validator message可以被复制给其它的field，而不是需要更动其信息内容。

Display Name是对当前field的一个描述信息。

9.4.1.5. 类型转换

例 9.21. 类型转换的配置

```
<services:form converterQuiet="true"> ❶
    <services:property-editor-registrar
        class="com.alibaba.citrus.service.configuration.support.CustomDateRegistrar"
        p:format="yyyy-MM-dd" /> ❷
</services:form>
```

❶ 如果`converterQuiet=true`，那么类型转换失败时，将取得默认值。否则，抛出异常。`converterQuiet`的默认值为`true`。

❷ 类型转换采用Spring Property Editor机制。你可以通过注册新的registrar来增加新的类型转换方法。这段配置增加了一种将日期转成字符串的方式（用`yyyy-MM-dd`格式）。

下面的操作将用到类型转换：

表 9.6. 何时用到类型转换?

操作	说明
`Group.setProperties(bean)`	将Group中的所有fields值注入bean properties。

操作	说明
`Group.mapTo(bean)`	用bean properties中的值初始化group fields。

9.4.1.6. 国际化

表单验证失败时，将在页面上显示错误信息。有两种方法可以定义错误信息：

- 将错误信息直接定义在配置文件中。前文的例子所用的都是这种方案。
- 将错误信息定义在Spring Message Source中，从而支持国际化。

为了将使用国际化（多语言）的错误信息，首先需要定义Spring Message Source。

例 9.22. 在Spring Message Source中定义错误信息

```
<bean id="messageSource"
       xmlns="http://www.springframework.org/schema/beans"
       class="org.springframework.context.support.ReloadableResourceBundleMessageSource"
       p:defaultEncoding="GB18030">
    <property name="basenames">
        <list>
            <value>form_msgs</value> ❶
        </list>
    </property>
</bean>
```

❶ 这段配置告诉spring去读取`form_msgs`开头的resource bundle文件，例如：

- form_msgs.properties
- form_msgs_zh_CN.properties
- form.msgs_zh_TW.properties

请注意，Spring是从`ResourceLoader`中读取resource bundle文件的。因此，你可能需要配置Resource Loading以帮助spring找到这些消息文件。关于资源装载，请参见第 5 章 *Resource Loading服务指南*。

使用message source以后，你可以省略validator中的`message`标签，但是*每个validator必须指定`id`*。表单系统将会从message source中查找指定的key：“`form.<GroupName>.<FieldName>.<ValidatorID>`”。

例 9.23. 配置validator ID

```
<services:form>
    <services:group name="register">
        <services:field name="userId">
            <required-validator id="required" /> ❶
        </services:field>
    </services:group>
</services:form>
```

❶ 指定了validator ID为`required`，根据格式“`form.<GroupName>.<FieldName>.<ValidatorID>`”，当前validator message的key为：“`form.register.userId.required`”。

假设message source定义文件及内容如下：

表 9.7. Message Source的内容

文件名	内容
form_msgs_zh_CN.properties	form.register.userId.required = 必须填写用户名
form_msgs.properties	form.register.userId.required = User ID is required

对于以上的message source内容，在中文环境中（locale=zh_CN），将显示错误信息“必须填写用户名”；而在英文环境中（locale=en_US），将显示默认的错误信息“User ID is required”。

系统的当前locale是由SetLocaleRequestContext来决定的。关于SetLocaleRequestContext的设定和使用，请参见第 7 章 *Request Contexts功能指南*。

此外，你还可以可以改变message source中key的前缀。

例 9.24. 改变message source key的前缀

```
<services:form messageCodePrefix="myform">
    ...
</services:form>
```

上面的配置将指导表单系统在message source中查找指定的key：“*myform*.GroupName.FieldName.ValidatorID”。

9.4.1.7. 切分表单服务

在实际的应用中，有时一个表单规则的配置文件会很长。将一个长文件切分成几个较短的文件，更有利于管理。表单验证服务支持导入多个form表单服务，从而实现分割较长配置文件的功能。

例 9.25. 切分表单服务

主配置文件：form.xml：

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans:beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:services="http://www.alibaba.com/schema/services"
    xmlns:beans="http://www.springframework.org/schema/beans">

    <beans:import resource="inc/form_part1.xml" /> ❶
    <beans:import resource="inc/form_part2.xml" /> ❷

    <services:form xmlns="http://www.alibaba.com/schema/services/form/validators" primary="true"> ❸
        <services:import form="part1" /> ❹
        <services:import form="part2" /> ❺

        ...
    </services:form>

</beans:beans>
```

❶❷ 导入包含着子表单服务的spring配置。

❸ 定义主表单服务时，*必须指定primary="true"*。否则spring将无法区分主从表单服务，从而导致注入FormService时失败。

❹❺ 导入指定ID的子表单服务。

子表单服务的配置文件：inc/form_part1.xml和inc/form_part2.xml：

```
<!-- inc/form_part1.xml -->
<beans:beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:services="http://www.alibaba.com/schema/services"
    xmlns:beans="http://www.springframework.org/schema/beans">

    <services:form id="part1"> ❶
        ...
    </services:form>

</beans:beans>

<!-- inc/form_part2.xml -->
<beans:beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:services="http://www.alibaba.com/schema/services"
    xmlns:beans="http://www.springframework.org/schema/beans">

    <services:form id="part2"> ❷
        ...
    </services:form>

</beans:beans>
```

❶❷ *子表单服务必须指定ID。*

导入子表单服务，意味着将子表单服务中的所有groups导入到主表单的空间。主表单中的groups将会覆盖被导入子表单中的groups。也就是说，假如主表单中存在一个group，它的名字和被导入的子表单中的group同名，那么子表单中的group将被忽略。

9.4.1.8. Group的继承和导入

在实际应用中，你会发现有一些groups的定义很相似。继承和导入的目的是让这些相似的groups之间可以共享共同的参数、字段和验证规则，避免重复定义。

下面是group继承的用法：

例 9.26. 继承一个group

```
<services:form>
    <services:group name="baseGroup">
        <services:field name="field1" >
            <validator1 />
            <validator2 />
        </services:field>
        <services:field name="field2" />
        <services:field name="field3" />
    </services:group>

    <services:group name="subGroup" extends="baseGroup">
        <services:field name="field1">
            <validator3 />
        </services:field>
        <services:field name="field4" />
    </services:group>

</services:form>
```

这段配置中，subGroup继承了baseGroup。其效果是：

- baseGroup的参数（postOnly、trimmingByDefault）被subGroup继承，除非subGroup明确指定了该参数。

- baseGroup中fields被subGroup继承。具体来说：
  - baseGroup中不同名的fields被直接添加到subGroup中。
  - baseGroup中同名的fields被subGroup中的继承。具体来说：
    - baseGroup field的参数（name、displayName、defaultValue、trimming、propertyName）被subGroup field继承，除非subGroup field明确指定了该参数。
    - baseGroup field中的validators被全部添加到subGroup field中。

因此，上面配置所定义的subGroup和下面配置中的完全等效：

例 9.27. Group继承的效果

```
<services:form>
    <services:group name="subGroup">
        <services:field name="field1">
            <validator1 /><!-- 来自baseGroup -->
            <validator2 /><!-- 来自baseGroup -->
            <validator3 />
        </services:field>
        <services:field name="field2" /><!-- 来自baseGroup -->
        <services:field name="field3" /><!-- 来自baseGroup -->
        <services:field name="field4" />
    </services:group>
</services:form>
```

另一种和继承类似的功能是导入：

例 9.28. 导入groups和fields

```
<services:form>
    <services:group name="group1">
        <services:field name="field1" />
        <services:field name="field2" />
        <services:field name="field3" />
    </services:group>

    <services:group name="group2">
        <services:import group="group1" /> ❶
        <services:field name="field4" />
    </services:group>

    <services:group name="group3">
        <services:import group="group1" field="field1" /> ❷
       <services:field name="field5" />
    </services:group>

</services:form>
```

❶ 导入group1中的全部fields。导入以后，group2拥有的fields包括：field1、field2、field3和field4。其中，field1、field2、field3均来自于group1。

❷ 导入group1中的一个field。导入以后，group3拥有的fields包括：field1和field5。其中，field1来自于group1。

导入有两种形式，导入全部fields和导入一个field。

导入和继承都可以使group共享一些内容，但是，

- 继承只能有一个base group，而导入可以有多次import；
- 继承会合并同名的fields，而导入时则禁止同名fields的。在上例中，假如`group2`已经有了`field1`，那么再次导入`group1`的`field1`将会报错。

9.4.1.9. 设置默认值

例 9.29. 设置表单字段的默认值

```
<services:field name="field1" defaultValue="defaultValue" />
```

当表单被创建时，所有的字段值默认都是空的 —— 除非你指定了`defaultValue`。需要注意的是，*默认值只影响初始表单。对于用户已经提交数据的表单不起作用*。

假如一个field需要多个值，例如多选的checkbox，那么它可以设置一个具有多值的默认值。方法是：用逗号分隔多值。像下面的样子：

例 9.30. 设置多个值作为表单字段的默认值

```
<services:field name="field1" defaultValue="defaultValue1, defaultValue2, defaultValue3" />
```

9.4.1.10. Fields和properties

Fields和properties是两个重要词汇。

Fields

Fields是指HTML表单中的form fields，例如一个文本框textbox、复选框checkbox、hidden字段等。

Fields也是表单验证的基本单元。

Properties

Properties是指Java bean中的数据成员，例如：`setName(String)`方法定义了一个property：`name`。

而在表单验证服务中，

- `Group.setProperties(bean)`方法将fields中的值注入到bean的properties中。
- `Group.mapTo(bean)`方法将bean properties的值设置成fields的初始值。

一般情况下，field的名称就是property的名称。然而有一些情况下，property名称和field名称会有出入。这时可以这样设置：

例 9.31. 设置不同的property和field名称

```
<services:field name="homeAddress" propertyName="home.address" />
```

其中，“`homeAddress`”为field名称。如果不指定`propertyName`的话，表单系统认为property名称也是“`homeAddress`”。然而在这里，指定了property名称为“`home.address`” —— spring支持这种多级的property。在上面的配置中，

- 当做`Group.setProperties()`时，会执行`bean.getHome().setAddress(value)`；

- 而做`Group.mapTo()`时，会执行`bean.getHome().getAddress()`。

9.4.1.11. Validator messages

每一个validator都可以附带一段message文字，这段文字会在validator验证失败时显示给用户看。配置validator message可以有下面两种写法：

例 9.32. Validator message的两种写法

```
<string-length-validator minLength="4" maxLength="10">
    <message>登录名最少必须由4个字组成，最多不能超过10个字</message>
</string-length-validator>
```

或者，你可以这样写：

```
<string-length-validator minLength="4" maxLength="10">
    <message>${displayName} 最少必须由${minLength}个字组成，最多不能超过${maxLength}个字</message>
</string-length-validator>
```

第二种形式使用了替换变量，例如：`${displayName}`等。这种方法有较多好处：

- 易复制 —— 假如有多个fields中都包含`string-length-validator`，由于每个fields的名称（`displayName`）、validator的参数（`minLength`、`maxLength`）都不同，第一种形式是不可复制的，而第二种形式是通用的、可复制的。
- 易维护 —— 当validator的参数被修改时，例如`minLength`被修改，对于第一种形式，你必须同时修改message字符串 —— 这很可能被忘记；而第二种形式就不需要修改message字符串。

事实上，validator message是一个JEXL表达式，其格式详见：http://commons.apache.org/jexl/reference/syntax.html。 下面列出了在message中可用的变量和工具。

表 9.8. Validator message中可用的变量和工具

分类	可用的变量和工具	
当前Validator中的所有properties	不同的validator有不同的properties。 例如，对于`<string-length-validator minLength="4" maxLength="10">`， 可取得的变量包括`${minLength}`、`${maxLength}`	
当前Field对象中的所有properties	`${displayName}`	Field显示名
	`${defaultValue}`	默认值（`String`）
	`${defaultValues}`	默认值数组（`String[]`）
	`${value}`	当前field的值（`Object`）
	`${values}`	当前field的一组值（`Object[]`）
	特定类型的值，例如：`${booleanValue}`、`${intValue}`等。	
当前Group中的其它Field对象	例如：`${userId}`，`${password}`等。 如果想取得其值，必须这样写：`${userId.value}` 也可取得其它Field properties，例如：`${userId.displayName}`	
当前的Group对象	`${group}`	
当前的Form对象	`${form}`	
System properties	所有从`System.getProperties()`中取得的值，	

分类	可用的变量和工具
	例如：${user.dir}、${java.home}等
小工具	${utils}，其中包含了很多静态方法。 例如：${utils.repeat("a", 10)}将会生成10个“a”。 详见com.alibaba.citrus.util.Utils类的API文档。

9.4.2. Validators

每个field都可以包含多个validators。Validators是用来验证当前field的正确性的。表单验证系统提供了很多常用的validators。然而，如果不够用，你还可以随时扩展出新的validators。

9.4.2.1. 验证必选项：<required-validator>

例 9.33. 配置<required-validator>

```
<services:field name="field1" displayName="我的字段">
    <required-validator>
        <message>必须填写${displayName}</message>
    </required-validator>
</services:field>
```

这是最常用的一个验证器。它的功能是确保用户填写了字段，即确保字段值非空。

需要注意的是，必选项验证器会受到trimming参数的影响。假如trimming=true（默认值），而用户输入了一组空白字符，那么仍然被认为“字段值为空”；反之，如果trimming=false，当用户输入空白时，会被认为“字段值非空”。

除了必选项验证器以外，其它*绝大多数的验证器并不会判断字段值是否为空*。例如：

例 9.34. 绝大多数的验证器并不会判断字段值是否为空

```
<services:field name="userId" displayName="登录名">
    <regexp-validator pattern="^[A-Za-z_][A-Za-z_0-9]*$">
        <message>${displayName} 必须由字母、数字、下划线构成</message>
    </regexp-validator>
</services:field>
```

在这个例子中，*即便用户什么也没填，<regexp-validator>也会通过验证*。换言之，<regexp-validator>只负责当字段值非空时，检查其是否符合特写的格式。

因此通常需要将必选项验证器和其它验证器配合起来验证。如下例：

例 9.35. 将必选项验证器和其它验证器配合起来验证

```
<services:field name="userId" displayName="登录名">
    <required-validator>
        <message>必须填写 ${displayName}</message>
    </required-validator>
    <regexp-validator pattern="^[A-Za-z_][A-Za-z_0-9]*$">
        <message>${displayName} 必须由字母、数字、下划线构成</message>
    </regexp-validator>
</services:field>
```

这样配置以后，就可以确保：

- 字段值非空；

- 字段值符合特定格式。

9.4.2.2. 验证字符串长度：<string-length-validator>

例 9.36. 配置<string-length-validator>

```
<services:field name="field1" displayName="我的字段">
    <string-length-validator minLength="4" maxLength="10">
        <message>${displayName} 最少必须由${minLength}个字组成，最多不能超过${maxLength}个字</message>
    </string-length-validator>
</services:field>
```

该验证器确保用户输入的字段值的字符数在确定的范围内。

可用的参数：

表 9.9. <string-length-validator>的可用参数：

参数名	说明
minLength	代表最少字符数，如不设置minLength代表最代表不设下限。
maxLength	代表最多字符数，如果不设置maxLength则代表不设上限，允许任意多个字符。

9.4.2.3. 验证字符串字节长度：<string-byte-length-validator>

例 9.37. <string-byte-length-validator>的配置

```
<services:field name="field1" displayName="我的字段">
    <string-byte-length-validator minLength="4" maxLength="10" charset="UTF-8">
        <message>${displayName} 最少必须由${minLength}个字节组成，最多不能超过${maxLength}个字节</message>
    </string-byte-length-validator>
</services:field>
```

该验证器也是用来验证字符值的长度的。但是和前面所讲的<string-length-validator>不同的是，<string-byte-length-validator>会将字符串先转换成字节串，然后判断这个字节串的长度。

为什么需要判定字节的长度呢？因为在数据库中，常用字节长度而不是字符长度来表示一个字段的长度的。例如：varchar(20)代表该数据库字段可接受的字节长度为20字节，超过部分将被截断。20字节可填入：

- 20个英文字母、数字，
- 或者6个UTF-8编码的汉字，
- 或者10个GBK编码的汉字。

可见20字节所能容纳的字符数是不确定的，取决于字符的类型（中文、英文、数字等），以及字符集编码的类型（ISO-8859-1、UTF-8、GBK等）。

可用的参数：

表 9.10. <string-byte-length-validator>的可用参数：

参数名	说明
minLength	代表最少字节数，如不设置minLength代表最代表不设下限。
maxLength	代表最多字节数，如果不设置maxLength则代表不设上限，允许任意多个字符。
charset	用来将字符串转换成字节，如不设置，则取当前线程的上下文编码。

9.4.2.4. 比较字符串：<string-compare-validator>

例 9.38. <string-compare-validator>的配置

```
<services:field name="field1" displayName="我的字段">
    <string-compare-validator notEqualTo="field2">
        <message>${displayName} 不能与 ${field2.displayName} 相同</message>
    </string-compare-validator>
</services:field>
```

该验证器将当前字段的值和另一个字段比较。

可用的参数：

表 9.11. <string-compare-validator>的可用参数：

参数名	说明
equalTo	确保当前字段值和指定的另一字段的值相同。
notEqualTo	确保当前字段值和指定的另一字段的值不相同。
ignoreCase	如果为true，则比较时忽略大小写。默认为false，即比较大小写。

9.4.2.5. 用正则表达式验证：<regexp-validator>

例 9.39. <regexp-validator>的配置

```
<services:field name="field1" displayName="我的字段">
    <regexp-validator pattern="^[A-Za-z_][A-Za-z_0-9]*$">
        <message>${displayName} 必须由字母、数字、下划线构成</message>
    </regexp-validator>
</services:field>
```

该验证器用正则表达式来验证字符串的格式。

可用的参数：

表 9.12. <regexp-validator>的可用参数：

参数名	说明
pattern	用来匹配字符串的正则表达式。需要注意的是： • 表达式为部分匹配，例如：表达式“abc”可以匹配用户输入“xabcy”。如果期望匹配完整字符串，必须使用“^”和“$”标识符。例如表达式“^abc$”只能匹配“abc”而不能匹配“xabcy”。 • 表达式支持否定匹配，例如：表达式“!abc”可以匹配所有不包含“abc”的字符串。

9.4.2.6. 验证电子邮件地址：<mail-address-validator>

例 9.40. <mail-address-validator>的配置

```
<services:field name="field1" displayName="我的字段">
    <mail-address-validator>
        <message>${displayName} 必须是合法电子邮件地址</message>
    </mail-address-validator>
</services:field>
```

该验证器用来确保用户输入了合法的电子邮件地址。

事实上，该验证器使用了一个比较宽松的正则表达式来验证电子邮件地址：“^\S+@[^\.]\S*$”。假如你觉得这个正则表达式不足以验证你所需要的邮件地址，你可以利用<regexp-validator>和自定义的正则表达式来直接验证。

9.4.2.7. 验证数字：<number-validator>

例 9.41. **<number-validator>的配置**

```
<services:field name="field1" displayName="我的字段1">
    <number-validator>
        <message>${displayName} 必须是数字</message>
    </number-validator>
</services:field>
<services:field name="field2" displayName="我的字段2">
    <number-validator numberType="int" lessThan="100">
        <message>${displayName} 必须是小于${lessThan}的整数</message>
    </number-validator>
</services:field>
```

该验证器用来确保用户输入了合法的数字。数字的合法性包括：格式的合法和范围的合法。

可用的参数：

表 9.13. **<number-validator>的可用参数**：

参数名	说明
numberType	数字的类型。可用的类型为：int、long、float、double、bigDecimal。如不设置，默认值为int。
equalTo	可选数字范围：要求数字*等于*指定值。
notEqualTo	可选数字范围：要求数字*不等于*指定值。
lessThan	可选数字范围：要求数字*小于*指定值。
lessThanOrEqualTo	可选数字范围：要求数字*小于或等于*指定值。
greaterThan	可选数字范围：要求数字*大于*指定值。
greaterThanOrEqualTo	可选数字范围：要求数字*大于或等于*指定值。

9.4.2.8. 比较数字：<number-compare-validator>

例 9.42. **<number-compare-validator>的配置**

```
<services:field name="field1" displayName="我的字段">
    <number-compare-validator greaterThanOrEqualTo="field2" lessThan="field3">
        <message>${displayName} 必须是位于 ${field2.displayName} 和 ${field3.displayName}之间的数字</
message>
    </number-compare-validator>
</services:field>
```

该验证器将当前字段的值和另一个字段比较。

可用的参数：

表 9.14. **<number-compare-validator>的可用参数**：

参数名	说明
numberType	数字的类型。可用的类型为：int、long、float、double、bigDecimal。如不设置，默认值为int。
equalTo	可选数字范围：要求数字*等于*指定字段的值。

参数名	说明
notEqualTo	可选数字范围：要求数字*不等于*指定字段的值。
lessThan	可选数字范围：要求数字*小于*指定字段的值。
lessThanOrEqualTo	可选数字范围：要求数字*小于或等于*指定字段的值。
greaterThan	可选数字范围：要求数字*大于*指定字段的值。
greaterThanOrEqualTo	可选数字范围：要求数字*大于或等于*指定字段的值。

9.4.2.9. 验证日期：<date-validator>

例 9.43. **<date-validator>的配置**

```
<services:field name="field1" displayName="我的字段">
    <date-validator format="yyyy-MM-dd">
        <message>${displayName} 必须是日期，格式为 ${format}</message>
    </date-validator>
</services:field>
```

该验证器用来确保用户输入正确的日期格式，也可以限定日期的范围。

可用的参数：

表 9.15. **<date-validator>的可用参数**：

参数名	说明
format	日期的格式，如不指定，默认为yyyy-MM-dd。
minDate	可选的日期范围：最早的日期。该日期格式也是用format参数来表示的。
maxDate	可选的日期范围：最晚的日期。该日期格式也是用format参数来表示的。

9.4.2.10. 验证上传文件：<uploaded-file-validator>

例 9.44. **<uploaded-file-validator>的配置**

```
<services:field name="picture" displayName="产品图片">
    <uploaded-file-validator extension="jpg, gif, png">
        <message>${displayName}不是合法的图片文件</message>
    </uploaded-file-validator>
    <uploaded-file-validator maxSize="100K">
        <message>${displayName}不能超过${maxSize}字节</message>
    </uploaded-file-validator>
</services:field>
```

该验证器用来验证用户上传文件的大小、类型等信息。

可用的参数：

表 9.16. **<date-validator>的可用参数**：

参数名	说明
minSize	最小文件尺寸。可使用K/M等单位，例如：10K、1M等。
maxSize	最大文件尺寸。可使用K/M等单位，例如：10K、1M等。
extension	允许的文件名后缀，多个后缀以逗号分隔。例如：gif、jpg、png。 注意，文件名是由浏览器传递给服务器的，因此*验证器并不能保证保证文件确实是文件名后缀声明的格式*。 例如，xxx.jpg有可能是一个exe可执行文件。
contentType	允许的文件类型，多个类型以逗号分隔。

参数名	说明
	例如：image/gif, image/jpeg, image/pjpeg, image/jpg, image/png。 注意，contentType是由浏览器传递给服务器的，因此*验证器并不能保证文件确实是浏览器所声明的格式*。 其次，有些浏览器不会传送contentType。因此*推荐使用extension文件名后缀验证*，来取代contentType验证。

注意：（请补充阅读*第 7 章 Request Contexts功能指南*）

- 上传文件验证只能检查浏览器所声称的文件名后缀和类型，并不能保证文件名后缀和类型属实。如果你希望进一步检查文件的内容，可以结合request-contexts/parser/filter的功能。
- 假如设置了request-contexts/parser/uploaded-file-whitelist，那么不符合要求的文件会在进入表单验证之前被删除。因此上传文件验证器的extension参数必须存在于uploaded-file-whitelist.extensions列表当中。
- Upload服务可限制请求的总尺寸，大于该尺寸的请求会被全部忽略，以保证服务器的安全性——这意味着对于这类超大请求，你根本读不到这个请求中的所有参数，当然也不可能执行到表单验证的阶段。
- Upload服务可以限制每个上传文件的尺寸，大于指定尺寸的文件会被删除。但只要请求的总尺寸还是在许可范围内，那么除了被删文件以外，其它的参数和文件还是可以被取得的。因此，上传文件验证器的maxSize必须小于upload服务中设置的单个文件的最大尺寸才有意义。

9.4.2.11. 验证CSRF token

CSRF是跨站请求伪造（Cross-site request forgery）的意思，它是一种常见的WEB网站攻击方法。攻击者通过各种方法伪造一个请求，模仿用户提交表单的行为，从而达到修改用户的数据，或者执行特定任务的目的。为了假冒用户的身份，CSRF攻击常常和XSS攻击配合起来做，但也可以通过其它手段，例如诱使用户点击一个包含攻击的链接。

通过CSRF token，可以确保该请求确实是用户本人填写表单并提交的，而不是第三者伪造的，从而避免CSRF攻击。CSRF token验证器是用来确保表单中包含了CSRF token。

CSRF token的验证是每个表单都需要的安全功能，所以通常可利用group的继承功能来定义CSRF token验证器。

例 9.45. **<csrf-validator>的配置**

```
<services:group name="csrfTokenCheckGroup">
    <services:field name="csrfToken">
        <csrf-validator>
            <message>您提交的表单已过期</message>
        </csrf-validator>
    </services:field>
</services:group>

<services:group name="group1" extends="csrfTokenCheckGroup">
    ...
</services:group>
```

除了表单验证以外，实现CSRF验证还需要其它几个步骤：

- 定义pull tool。

例 9.46. 定义CSRF pull tool

```
<services:pull xmlns="http://www.alibaba.com/schema/services/pull/factories">
    ...
    <csrfToken /> ❶
    ...
</services:pull>
```

❶ 定义了pull tool以后，可以在模板中以`$csrfToken`来引用它。

- 在每一个表单中创建一个保存着CSRF token的hidden字段。

例 9.47. 在模板中插入包含CSRF token的hidden字段

```
<form action="" method="post">
  $csrfToken.hiddenField ❶
  <input type="hidden" name="action" value="LoginAction"/>
  ...
</form>
```

❶ 调用`$csrfToken.hiddenField`以后将创建一个包含CSRF long live token的hidden字段，等同于调用`$csrfToken.longLiveHiddenField`。

有两种CSRF token，你也可以用下面两种方法来创建它们：

- 创建unique token：`$csrfToken.uniqueHiddenField`。这种类型的token不仅能防止CSRF攻击，还能防止重复提交表单。
- 创建long live token：`$csrfToken.longLiveHiddenField`。这种类型的token只能防止CSRF攻击，不能防止重复提交表单。

- 在pipeline中验证token。

例 9.48.

```
<services:pipeline xmlns="http://www.alibaba.com/schema/services/pipeline/valves">
    ...
    <checkCsrfToken /> ❶
    ...
</services:pipeline>
```

❶ 此处可以指定一个`tokenKey`参数。如果不指定，将使用默认的token key：`_csrf_token`。

- 在表单验证中指定`postOnly=true`（默认值），有助于提高CSRF攻击的难度。

例 9.49. 配置Post Only参数

```
<services:form postOnlyByDefault="true">
</services:form>
```

9.4.2.12. Custom Error – 由action来验证数据

有一些情况下，由validator来验证数据并不方便。例如，当我们注册帐户时，即便用户名的格式完全正确（由字母和数字构成，并在指定的字数范围之内），也有可能注册不成功的。原因是

当前用户名已经被其它用户注册使用了。而判断用户名是否可用，最简单的办法是在action中通过访问数据库来确定。

Custom Error“验证器”就是用来满足这个需求。

例 9.50. <custom-error>的配置

```
<services:group name="register">
    <services:field name="userId" displayName="登录名">
        <required-validator>
            <message>必须填写 ${displayName}</message>
        </required-validator>
        <regexp-validator pattern="^[A-Za-z_][A-Za-z_0-9]*$">
            <message>${displayName} 必须由字母、数字、下划线构成</message>
        </regexp-validator>
        <string-length-validator minLength="4" maxLength="10">
            <message>${displayName} 最少必须由${minLength}个字组成，最多不能超过${maxLength}个字</message>
        </string-length-validator>
        <custom-error id="duplicatedUserId"> ❶
            <message>登录名“${userId}”已经被人注掉了，请尝试另一个名字</message>
        </custom-error>
    </services:field>
    ...
</services:group>
```

❶ Custom Error“验证器”不做任何验证 —— 它把验证的责任交给action来做。但是除此以外，它和其它验证器完全相同 —— 你可以设置message，甚至可以用message source来实现国际化的错误提示。你不需要把错误提示写在代码中，或者启用另一种错误提示方案。

对于custom error，需要在action中有相应的支持，否则不会自动生效：

例 9.51. 用来生成custom error的action代码

```
public void doRegister(@FormField(name = "userId", group = "register") CustomErrors err, ❶
                       ...) throws Exception {
    try {
        ...
    } catch (DuplicatedUserException e) {
        Map<String, Object> params = createHashMap();
        params.put("userId", user.getUserId());

        err.setMessage("duplicatedUserId", params); ❷
    }
}
```

❶ 注入CustomErrors接口。和注入Field的方法相同，通过@FormField注解指明CustomErrors所在的 group名称以及field名称。

❷ 调用CustomeErrors.setMessage()方法。其中，“duplicatedUserId”就是配置文件中<custom-error>的id。

第二个参数params是可选的。它是一个Map，其中的所有值，都可以在<custom-error>的message中访问到。例如，这里指定的userId参数值，就可以被message表达式${userId}所访问。

9.4.2.13. 条件验证

条件验证就是让某些validator仅在条件满足时才验证。条件验证有两种，单分支和多分支验证。

例 9.52. 单分支条件验证

```
<services:field name="other" displayName="其它建议">
    <if test="commentCode.value == 'other'">
        <required-validator>
            <message>必须填写 ${displayName}</message>
        </required-validator>
    </if>
</services:field>
```

例 9.53. 多分支条件验证

```
<services:field name="field1" displayName="我的字段">
    <choose>
        <when test="expr1">
            <validator />
        </when>
        <when test="expr2">
            <validator />
            <validator />
        </when>
        <otherwise>
            <validator />
        </otherwise>
    </choose>
</services:field>
```

在上面的配置示例中，<if>和<when>都支持的test参数，其内容为JEXL表达式。其格式详见：http://commons.apache.org/jexl/reference/syntax.html。JEXL表达式中可用的变量同Validator messages中可用的变量，请参见：第 9.4.1.11 节 “Validator messages”。除此之外，条件分支还支持任意自定义的条件，方法是：

例 9.54. 在条件验证中自定义条件

```
<if xmlns:fm-conditions="http://www.alibaba.com/schema/services/form/conditions">
    <fm-conditions:condition class="xxx" /> ❶
</if>
...
<when xmlns:fm-conditions="http://www.alibaba.com/schema/services/form/conditions">
    <fm-conditions:condition class="xxx" /> ❷
</when>
```

❶❷ 实现类只需要实现Condition接口就可以了。

9.4.2.14. 多值验证

HTML表单字段均支持多值。比如：

例 9.55. 具有多值的HTML表单字段

```
<p>你喜欢吃哪些食物？</p>
<input type="checkbox" name="poll" value="italian" /> 意大利菜
<input type="checkbox" name="poll" value="french" /> 法国菜
<input type="checkbox" name="poll" value="chinese" /> 中国菜
```

当用户选择了多个复选框并提交以后，在表单系统中体现为数组：

```
field.getName();   // "poll"
field.getValues(); // "italian", "french", "chinese"
```

不仅仅是复选框，任何其它类型的输入框（textbox、hidden field、file upload等）都支持多值。然而前面所说的所有validator只对field中的第一个值进行验证。假如我希望对多个值同时进行验证，该怎么办呢？表单验证服务提供了一组用于多值验证的validators。

9.4.2.14.1. 验证值的数量

例 9.56. **<multi-values-count-validator>的配置**

```
<services:field name="poll" displayName="调查">
    <multi-values-count-validator minCount="1" maxCount="3"> ❶
        <message>至少选择${minCount}项，最多选择${maxCount}项</message>
    </multi-values-count-validator>
</services:field>
```

❶ 对用户提交的值的数量进行验证，迫使用户选择1-3项他喜欢的食物。

9.4.2.14.2. 要求所有值均通过验证

例 9.57. **<all-of-values>的配置**

```
<all-of-values>
    <message>${allMessages}</message>
    <validator />
    <validator />
</all-of-values>
```

只有当前字段的所有值都符合要求，<all-of-values>验证才会通过。其message支持${allMessages}，它是一个List列表，可以用来显示所有未通过验证的validators的消息。

9.4.2.14.3. 要求任意一个值通过验证

例 9.58. **<any-of-values>的配置**

```
<any-of-values>
    <message>至少有一个${displayName}要符合要求</message>
    <validator />
    <validator />
</any-of-values>
```

只要当前字段有一个值通过验证，<any-of-values>验证就会通过。其message支持${valueIndex}代表被验证通过的值的序号；支持${allMessages}，它是一个List列表，可以用来显示所有未通过验证的validators的消息。

9.4.2.14.4. 要求任意一个值都不通过验证

例 9.59. **<none-of-values>的配置**

```
<none-of-values>
    <message>所有${displayName}都不能符合要求</message>
    <validator />
    <validator />
</none-of-values>
```

只要当前字段有一个值通过验证，<none-of-values>验证就会失败。

9.4.2.15. 组合验证

组合验证就是将validators组合起来，类似于Java中的and（&&）、or（||）、not（!）等操作符的功能。

9.4.2.15.1. 要求所有validators通过验证

例 9.60. **<all-of>的配置**

```
<all-of>
    <validator />
    <validator />
</all-of>
```

只要有一个validator通不过验证，就失败。<all-of>不需要设置message，它的message就是第一个没有通过验证的validator的message。

9.4.2.15.2. 要求任意一个validators通过验证

例 9.61. **<any-of>的配置**

```
<any-of>
    <message>${allMessages}</message>
    <validator />
    <validator />
</any-of>
```

只要有一个validator通过验证，<any-of>验证就会通过。其message支持${allMessages}，它是一个List列表，可以用来可以用来显示所有未通过验证的validators的消息。

9.4.2.15.3. 要求任何一个validators都通不过验证

例 9.62. **<none-of>的配置**

```
<none-of>
    <message>${displayName}不符合要求</message>
    <validator />
    <validator />
</none-of>
```

只要有一个validator通过验证，<none-of>验证就会失败。

9.4.3. Form Tool

Form Tool是一个pull tool工具，配置如下：

例 9.63. Form Tool的配置

```
<services:pull xmlns="http://www.alibaba.com/schema/services/pull/factories">
    <form-tool />
    ...
</services:pull>
```

上面的配置定义了一个$form工具。可以在模板中直接使用它。下页简单介绍在模板中，$form工具的用法。

9.4.3.1. Form API

表 9.17. 有关Form的API

API用法	说明
#if ($form.valid) ... #end	判断当前form是否验证为合法，或者未经过验证。

API用法	说明
#set ($group = $form.group1.defaultInstance)	取得group1的默认实例，如果不存在，则创建之。
#set ($group = $form.group1.getInstance("id"))	取得group1的指定id的实例，如果不存在，则创建之。
#set ($group = $form.group1.getInstance("id", false))	取得group1的指定id的实例，如果不存在，则返回null。
#foreach ($group in $form.groups) ... #end	遍历当前form中所有group实例。
#foreach ($group in $form.getGroups("group1")) … #end	遍历当前form中所有名为group1的实例。

9.4.3.2. Group API

表 9.18. 有关Group的API

API用法	说明
#if ($group.valid) … #end	判断当前group是否验证为合法，或者未经过验证（即初始表单）
#if ($group.validated) ... #end	判断当前group是否经过验证（初始表单为未经过验证的表单）
$group.field1	取得field1
#foreach ($field in $group.fields) ... #end	遍历当前group中所有的fields
$group.mapTo($bean)	将bean中的properties设置成group的初始值。 该操作只对初始表单有效。如果bean为null则忽略该操作。

9.4.3.3. Field API

创建一个HTML表单字段

例 9.64. 创建一个HTML表单字段

```
$field.displayName
<input type="text" name="$field.key" value="$!field.value" />
```

其中，displayName来自于配置文件。将displayName显示在页面中的好处是，确保页面与出错信息的措辞一致。

判断验证合法性，并显示错误消息

例 9.65. 判断验证合法性，并显示错误消息

```
#if (!$field.valid)
  <div class="error">$field.message</div>
#end
```

取得多值

例 9.66. 取得多值

```
#foreach ($value in $field.values) ... #end
```

创建checkbox和radiobox的默认值

例 9.67. 创建checkbox和radiobox的默认值

```
<input type="hidden" name="$field.absentKey" value="$value" />
```

或者简化为：

```
$field.getAbsentHiddenField($value)
```

Checkbox和radiobox有一个特性，当用户没有选中它们时，它们是没有值的（就像不存在一样）。这点对于表单验证会带来不便。

表单服务支持一种特殊的**absentKey**。通过它，可以为checkbox/radiobox设置默认值。这样，当用户没有选中任何checkbox或radiobox时，这个值就成为field的值。

创建附件

Field可以带一个附件。附件是一个对象，被序列化保存在hidden字段中。在下一次请求的时候，附件可以被恢复成对象。通过附件，可以让应用程序在表单中携带一些附加信息。下页的代码会生成一个hidden字段，将**$obj**序列化保存在其中：

例 9.68. 创建附件

```
$field.setAttachment($obj)
$field.attachmentHiddenField
```

当你要取得它时，只要这样：

```
#set ($obj = $field.attachment)
```

判断是否有附件：

```
#if ($field.hasAttachment()) … #end
```

清除附件：

```
$field.cleanAttachment()
```

9.4.4. Field keys的格式

表单验证服务所生成的field key是这样的：“**_fm.r._0.p**”。它是由几部分组成：

表 9.19. **压缩格式的**field key（**以_fm.r._0.p为例**）**的组成**

名称	说明
_fm	固定的前缀。它是单词“form”的缩写。表单系统依此来识别该字段为需要验证的表单字段。
r	被压缩的group名称。
_0	代表group instance的唯一ID。_0是默认的ID。
p	被压缩的field名称。

在以上例子中，field keys被压缩了。压缩以后的field keys更短，但同时也比较难以阅读。由于在多数情况下，我们并不需要去理解field keys的含义，所以这样做并没有问题。但有一种情况，我们需要对表单进行单元测试或者集成测试。这种压缩的格式会为测试带来一定困难。为此，表单服务提供了另一种非压缩的格式可供使用。非压缩的格式和压缩格式类似，只不过其group和field的名称是完整的。例如，非压缩版的“**_fm.register._0.password**”和压缩版的**_fm.r._0.p**是等效的。

表 9.20. **非压缩格式的**field key（**以_fm.register._0.password为例**）**的组成**

名称	说明
_fm	固定的前缀，是单词“form”的缩写。表单系统依此来识别该字段为需要验证的表单字段。

名称	说明
register	完整的group名称。大小写不敏感，以下写法完全等效：register、Register、rEgiSter。
_0	代表group instance的唯一ID。_0是默认的ID。
password	完整的field名称。大小写不敏感，以下写法完全等效：password、Password、paSswoRd。

浏览器或单元测试的代码在提交表单数据时，可以混合使用压缩和非压缩的格式。但是在默认情况下，表单系统只会生成压缩格式的field keys。如果你希望表单系统生成非压缩的格式，你可以在配置文件中这样写：

例 9.69. 让表单系统生成非压缩的格式的field keys

```
<services:form fieldKeyFormat="uncompressed"> ❶
    ...
</services:form>
```

❶ 如果不指定，其默认值为compressed。

注意

当表单系统被配置成fieldKeyFormat="uncompressed"时，系统就*不支持*压缩格式的field keys了。

当表单系统被配置成fieldKeyFormat="compressed"时，系统就*同时支持*压缩格式和非压缩格式的field keys。

9.4.5. 外部验证

表单验证服务是被设计成供一个*应用的内部*使用的服务。它所生成的压缩格式的field key，例如“_fm.r._0.p”，是不稳定的。它和配置文件中的group、field的名称、排列顺序有关，可能随着配置的变化而变化。即便是非压缩的格式，例如“_fm.register._0.password”，也会因配置文件中group、field命名的改变而改变。如果需要让外界系统来提交并验证表单，最好提供一个相对稳定的接口。所以外界系统最好不要依赖于这些内部的field keys。

如果真的需要让外界系统来提交并验证表单，可以做一个screen来转发这个请求。Screen的代码像这个样子：

例 9.70. 转发外部表单请求

```
public class RemoteRegister {
    public void execute(ParameterParser params, Form form, Navigator nav) throws Exception {
        Group group = form.getGroup("register"); ❶

        group.init(); ❷
        group.getField("userId").setValue(params.getString("userId")); ❸
        group.getField("password").setValue(params.getString("password")); ❹
        group.getField("passwordConfirm").setValue(params.getString("password")); ❺
        group.validate(); ❻

        nav.forwardTo("register")
            .withParameter("action", "registerAction")
            .withParameter("eventSubmitDoRegister", "yes"); ❼
    }
}
```

❶ 创建register group的实例。

❷❸❹❺ 将request parameters中的参数设置到form group中。需要注意的是，request参数名和field名称不必相同。

❻ 验证表单。

❼ 内部重定向到register页面，并指明action参数。

只需要访问下面的URL就可以实现从系统外部注册帐户的功能：

```
http://localhost:8081/myapp/remote_register.do?userId=xxx&password=yyy
```

9.5. 本章总结

表单服务是一个比较复杂但也相当强大的服务。虽然目前它还不支持客户端验证和服务端异步验证功能，但下一步会加上这些功能。

表单服务最重要的设计思想是：将验证规则与页面以及业务逻辑完全分离，使验证规则的扩展和维护变得非常容易。

部分 IV. Webx应用实作

第 10 章 创建第一个Webx应用

本章将帮助你快速创建一个可运行的Webx应用。你可以把它作为你的Webx新项目的开端。

10.1. 准备工作

请耐心，准备工作会花掉你少许时间。但是磨刀不误砍柴工，做好准备工作将为你将来的开发节省大量时间。

10.1.1. 安装JDK

Webx需要JDK 5.0以上的版本。请从这里下载并安装它：http://www.oracle.com/technetwork/java/javase/。

10.1.2. 安装和配置maven

Webx需要maven 2或更高版本。请从这里下载并安装它：http://maven.apache.org/。

注意

你不需要对maven进行特殊的配置，因为运行Webx应用所需要的所有包都存放在全世界共享的中心Maven仓库（Central Maven Repository）中。Maven将从那里自动获取所有的jar包、源代码和javadoc。

你可以从这里查询到所有和Webx有关的发布包：http://search.maven.org/#search%7Cga%7C1%7Ccom.alibaba.citrus。

10.1.3. 安装集成开发环境

很难想像不用集成开发环境（IDE）来帮助开发Java应用会变成怎样。

如果你使用Eclipse（从这里下载：http://www.eclipse.org/），建议安装如下插件：

- Maven eclipse插件：http://eclipse.org/m2e/
- Git eclipse插件：http://eclipse.org/egit/

10.2. 创建应用

请打开命令行工具（Windows cmd或Unix/Linux bash），输入如下命令：

例 10.1. 从archetype创建Webx应用

```
mvn archetype:generate \
 -DgroupId=com.alibaba.webx \❶
 -DartifactId=tutorial1 \❷
 -Dversion=1.0-SNAPSHOT \❸
 -Dpackage=com.alibaba.webx.tutorial1 \❹
 -DarchetypeArtifactId=archetype-webx-quickstart \
 -DarchetypeGroupId=com.alibaba.citrus.sample \
 -DarchetypeVersion=1.2 \
 -DinteractiveMode=false
```

由于Windows下不支持命令换行，请改用*非换行版*:

```
mvn archetype:generate -DgroupId=com.alibaba.webx -DartifactId=tutorial1 -Dversion=1.0-SNAPSHOT
 -Dpackage=com.alibaba.webx.tutorial1 -DarchetypeArtifactId=archetype-webx-quickstart -
DarchetypeGroupId=com.alibaba.citrus.sample -DarchetypeVersion=1.2 -DinteractiveMode=false
```

命令执行完后，你会看见一个新目录：`tutorial1`。它就是我们刚刚创建的新项目。项目的各项参数如下所示：

❶ 项目组（`groupId`）：`com.alibaba.webx`。

❷ 项目名称（`artifactId`）：`tutorial1`。

❸ 项目版本（`version`）：`1.0-SNAPSHOT`。

❹ 项目中Java类的包名（`package`）：`com.alibaba.webx.tutorial1`。

注意

你完全可以根据你的需要来调整上述命令中的参数，改用其它的`groupId`、`artifactId`、`version`以及`package`。

10.3. 运行应用

进入刚创建的`tutorial1`目录，在此目录下执行maven命令：

例 10.2. 启动Jetty服务器

```
mvn jetty:run
```

这条命令会启动Jetty Server，默认的端口是`8081`。请在浏览器地址栏输入地址，或直接点击这个链接：http://localhost:8081/。你应该可以看到类似下面的结果：

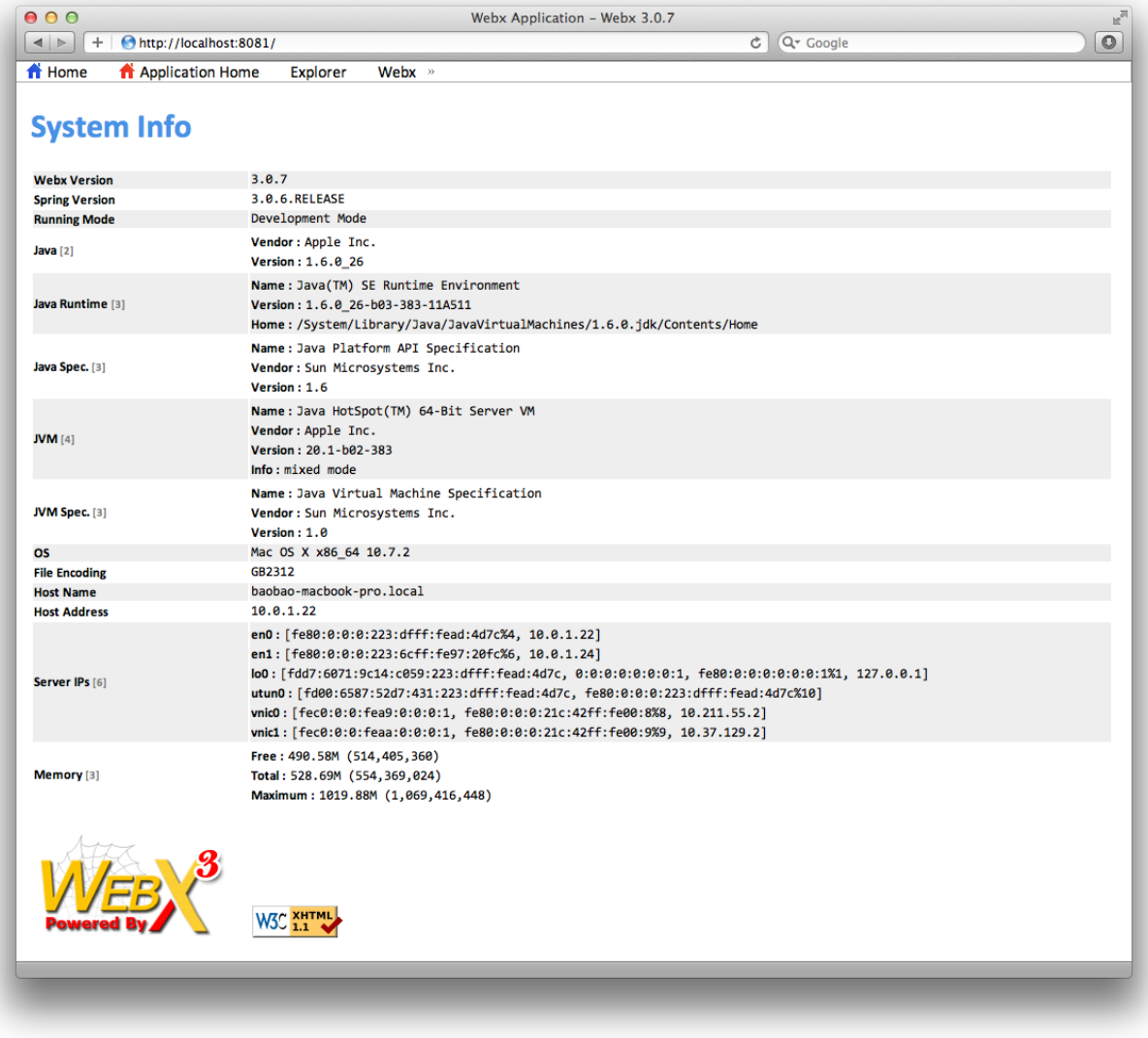
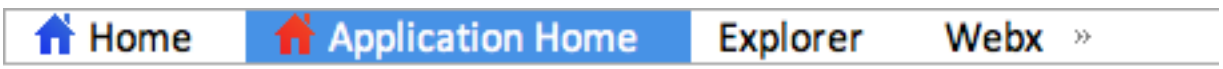


图 10.1. Webx开发者首页

这是一个“开发者首页”。它不是真正的应用程序首页，而是一个*专为开发者准备的*首页。这个页面显示了一些诸如Webx版本、Java版本、OS类型、IP地址、内存等信息。

请点击页面顶部的菜单中的Application Home，



这样就可以进入*真正的应用程序首页*：

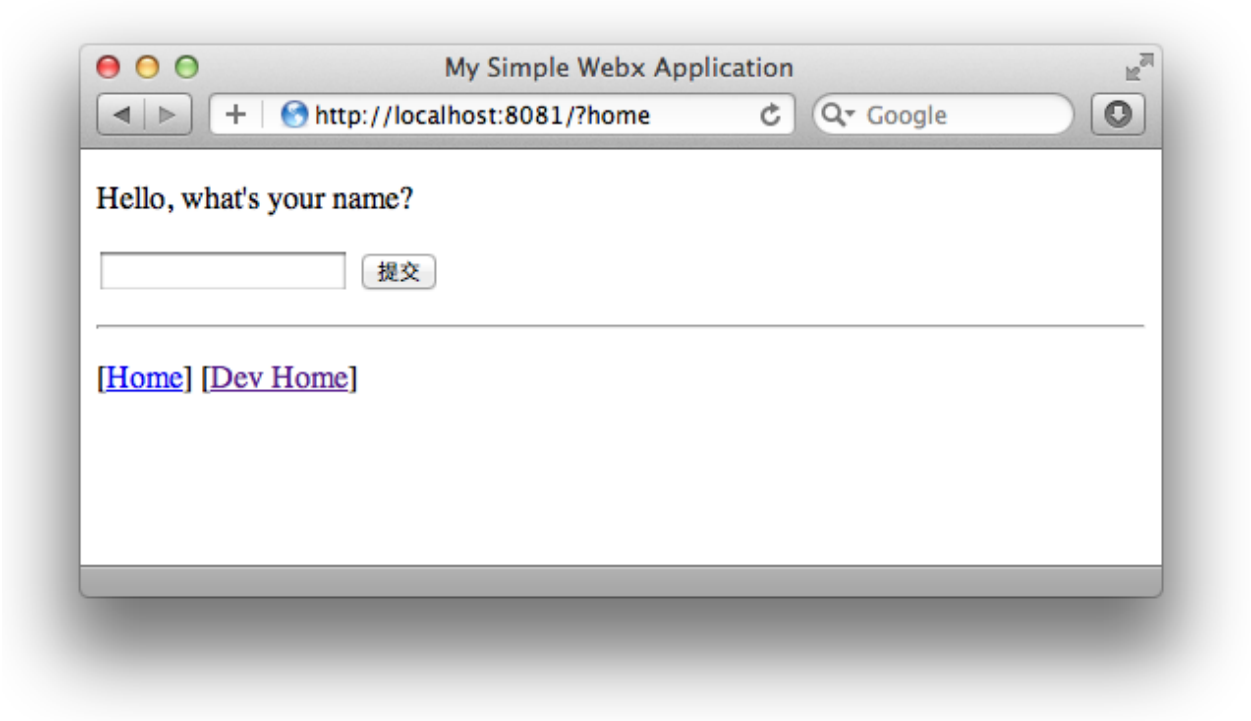


图 10.2. Webx应用首页

10.4. 提问和解答

10.4.1. 在生产环境的应用上，也会出现前述的“开发者首页”吗?

不会的。

事实上，之所以刚才的运行会产生“开发者首页”，是因为在jetty启动时，定义了一个启动参数。请打开`pom.xml`看一下：

例 10.3. 定义“开发者模式”的启动参数

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    ...
    <build>
        <plugins>
            ...
            <plugin>
                <groupId>org.mortbay.jetty</groupId>
                <artifactId>maven-jetty-plugin</artifactId>
                <configuration>
                    <contextPath>/</contextPath>
                    <connectors>
                        <connector implementation="org.mortbay.jetty.nio.SelectChannelConnector">
                            <port>8081</port>
                            <maxIdleTime>60000</maxIdleTime>
                        </connector>
                    </connectors>
                    <requestLog implementation="org.mortbay.jetty.NCSARequestLog">
                        <filename>target/access.log</filename>
                        <retainDays>90</retainDays>
                        <append>false</append>
                        <extended>false</extended>
                        <logTimeZone>GMT+8:00</logTimeZone>
                    </requestLog>
                    <systemProperties>
                        <systemProperty>
                            <name>productionMode</name> ❶
                            <value>false</value>
                        </systemProperty>
                    </systemProperties>
                </configuration>
            </plugin>
            ...
        </plugins>
    </build>
</project>
```

❶ 设定JVM启动参数：`productionMode=false`。

如果不加特别设置，系统的默认状态是“生产模式”，即`productionMode`默认为`true`。因此，生产环境的服务器总是运行在“生产模式”而不是“开发模式”下的。而“开发者首页”只会出现在“开发模式”下面。

你可以用下面的命令来覆盖`pom.xml`的设置：

例 10.4. 在命令行上指定JVM参数

```
mvn jetty:run -DproductionMode=true
```

然后在浏览器地址栏输入地址，或直接点击这个链接：http://localhost:8081/，你将直接被带入应用首页，而不是“开发者首页”。

10.4.2. “开发模式”是什么意思?

开发模式是为了方便应用开发，Webx所提供的额外功能。具体请见：第 3.2.4 节 “开发模式工具”。

10.4.3. 所生成的应用中包含了什么?

这个应用程序包含了几个Webx应用的常见元素：

- 一个欢迎页面（index screen）
- 一个页面布局（layout）
- 一个表单验证（form）
- 一个action，用来处理用户提交的数据
- Logback日志被打印在屏幕上

第 11 章 Webx日志系统的配置

日志系统是一个应用中必备的部分，提供了查看错误信息、了解系统状态的最直接手段。

本章介绍了基于Webx框架的应用如何配置、使用日志系统的方法。

11.1. 名词解释

11.1.1. 日志系统（Logging System）

表 11.1. 日志系统

名称	说明
Log4j	http://logging.apache.org/log4j/ 较早出现的比较成功的日志系统是Log4j。 Log4j开创的日志系统模型（Logger/Appender/Level）行之有效，并一直延用至今。
JUL（java.util.logging.*）	http://download.oracle.com/javase/6/docs/technotes/guides/logging/overview.html JDK1.4是第一个自带日志系统的JDK，简称（JUL）。 JUL并没有明显的优势来战胜Log4j，反而造成了标准的混乱 —— 采用不同日志系统的应用程序无法和谐共存。
Logback	http://logback.qos.ch/ 是较新的日志系统。 它是Log4j的作者吸取多年的经验教训以后重新做出的一套系统。它的使用更方便，功能更强，而且性能也更高。 Logback不能单独使用，必须配合日志框架SLF4J来使用。

11.1.2. 日志框架（Logging Framework）

JUL诞生以后，为了克服多种日志系统并存所带来的混乱，就出现了“日志框架”。日志框架本身不提供记录日志的功能，它只提供了日志调用的接口。日志框架依赖于实际的日志系统如Log4j或JUL来产生真实的日志。

使用日志框架的好处是：应用的部署者可以决定使用哪一种日志系统（Log4j还是JUL），或者在多种日志系统之间切换，而不需要更改应用的代码。

表 11.2. 日志框架

名称	说明
JCL（Jakarta Commons Logging）	http://commons.apache.org/logging/ 这是目前最流行的一个日志框架，由Apache Jakarta社区提供。 Spring框架、许多老应用都依赖于JCL。
SLF4J	http://www.slf4j.org/ 这是一个最新的日志框架，由Log4j的作者推出。 SLF4J提供了新的API，特别用来配合Logback的新功能。但SLF4J同样兼容Log4j。

由于Log4j原作者的感召力，SLF4J和Logback很快就流行起来。Webx的新版本也决定使用SLF4J作为其日志框架；并推荐Logback作为日志系统，但同时支持Log4J。

11.2. 在Maven中组装日志系统

要在应用中使用日志系统，必须把正确的jar包组装起来。本章假设你的应用是用maven构建的。

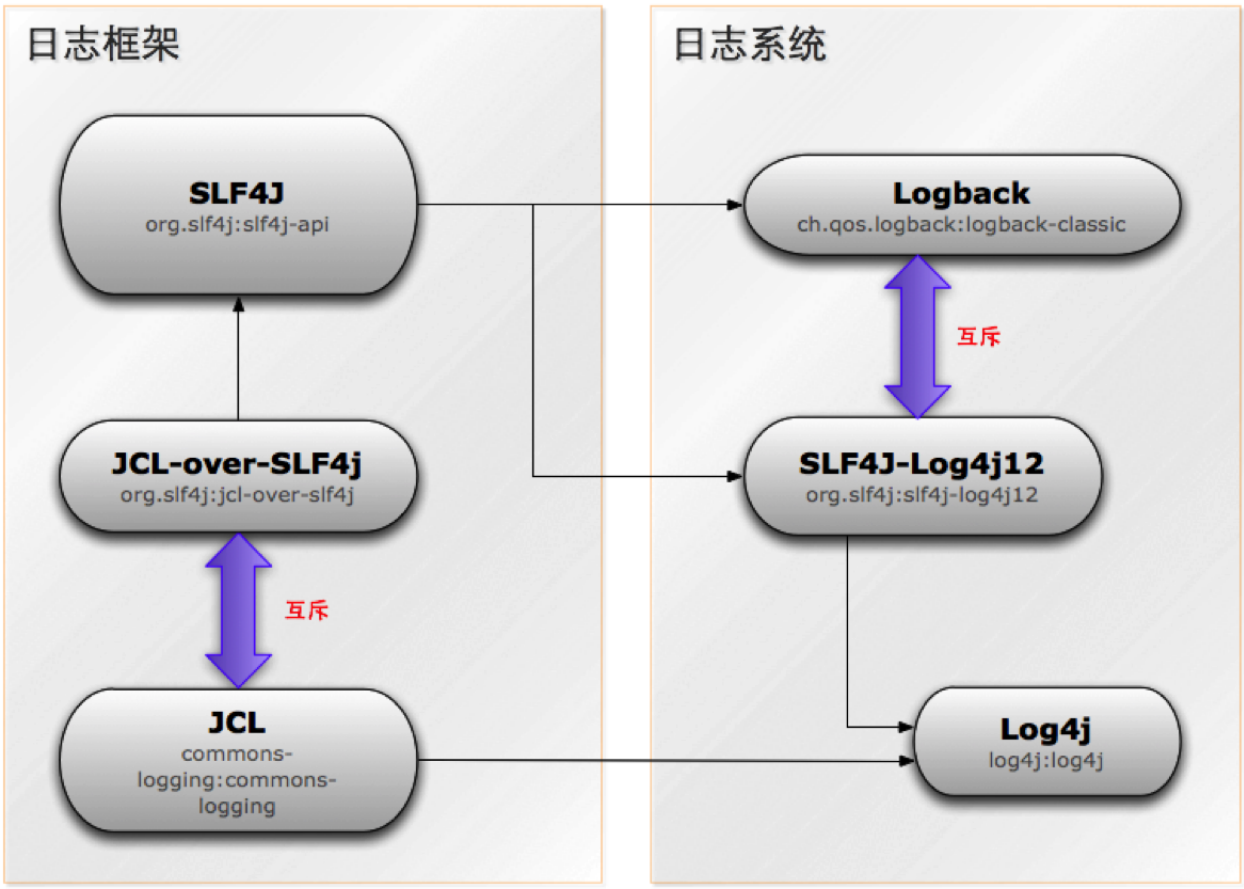


图 11.1. 日志系统的组成

如图所示，

- 由于JCL-over-SLF4J和原来的JCL具有完全相同的API，因此两者是不能共存的。
- Logback和slf4j-log4j12也不能并存，否则SLF4J会迷惑并产生不确定的结果。

组装完整的日志系统将涉及如下部件：

表 11.3. 日志系统的组成

类别	组件名称	说明
日志框架	SLF4J	Webx框架以及所有新应用，直接依赖于SLF4J。
	JCL	Spring框架、许多以前的老应用，都使用JCL来输出日志。 好在SLF4J提供了一个“桥接”包：JCL-over-SLF4J，它重写了JCL的API，并将所有日志输出转向SLF4J。这样就避免了两套日志框架并存的问题。
日志系统	Logback	Webx推荐使用logback来取代log4j。 Logback可直接被SLF4J识别并使用。
	Log4j	由于客观原因，有些系统暂时不能升级到Logback。 好在SLF4J仍然支持Log4j。Log4j需要一个适配器slf4j-log4j12才能被SLF4J识别并使用。

11.2.1. 在Maven中配置logback作为日志系统

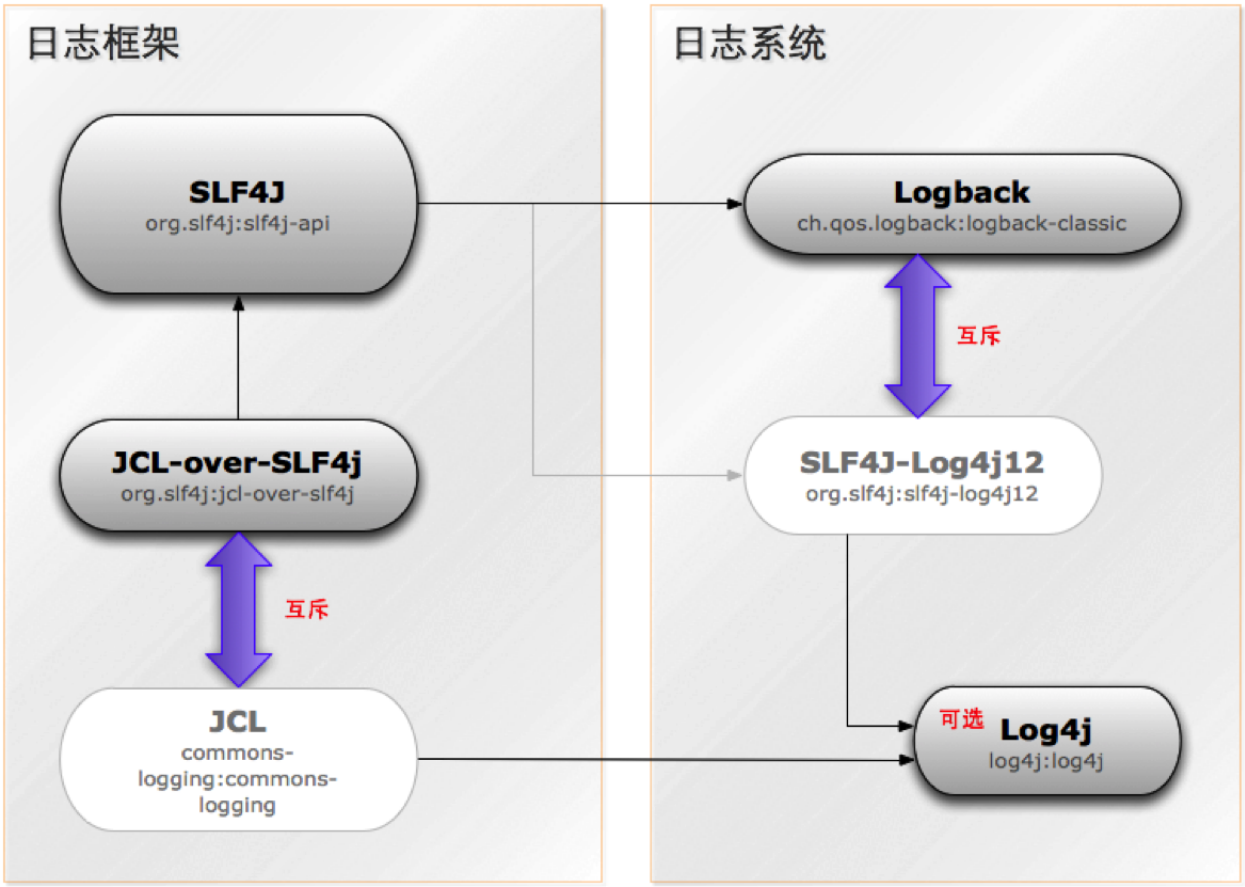


图 11.2. 以logback作为日志系统

例 11.1. 配置pom.xml以使用logback

```
<dependencies>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>jcl-over-slf4j</artifactId>
    </dependency>
    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
    </dependency>
</dependencies>

<dependencyManagement> ❶
    <dependencies>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.6.6</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>jcl-over-slf4j</artifactId>
            <version>1.6.6</version>
        </dependency>
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
            <version>1.0.6</version>
            <scope>runtime</scope> ❷
        </dependency>
        <dependency>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
            <version>1.1.1</version>
            <scope>provided</scope> ❸
        </dependency>
    </dependencies>
</dependencyManagement>
```

❶ 把所依赖jar包的版本定义在<dependencyManagement>中，而不是<dependencies>中。因为前者可影响间接依赖，后者只能影响直接依赖。

如果你的项目指定了parent pom，那么*建议把<dependencyManagement>放在parent pom中*，以便多个子项目共享配置。

❷ 将logback日志系统的依赖设定为<scope>runtime</scope>，因为应用程序永远不需要直接调用日志系统，而是通过SLF4J或JCL这样的日志框架来调用它们。

❸ 由于和jcl-over-slf4j存在冲突，因此JCL（commons-logging）是必须被排除的。由于maven目前缺少这样一个功能：它不能全局地排除一个jar包依赖，所以建议将commons-logging设置成<scope>provided</scope>，这样在最终的依赖关系中，将不会包含commons-logging包。

将commons-logging设置成<scope>provided</scope>可以用来排除commons-logging，然而这样做有一个缺点 —— 你无法从单元测试中将commons-logging排除。假如这个影响了你的单元测试的话，请使用另一种方案：

例 11.2. 另一种排除commons-logging的方法

```
        <dependency>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
            <version>99.0-does-not-exist</version> ❶
        </dependency>
```

❶ “<version>99.0-does-not-exist</version>”是一个特殊的版本，这个版本的jar包里空无一物。这样就可以“欺骗”maven使用这个空的jar包来取代commons-logging，达到排除它的目的。

最后，你需要在项目文件夹下面，执行一下命令：“mvn dependency:tree”，确保没有jar包直接或间接依赖了slf4j-log4j12。如果有的话，你可以用下面的配置来排除掉：

例 11.3. 排除间接依赖的slf4j-log4j12

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>yourGroupId</groupId>
            <artifactId>yourArtifactId</artifactId>
            <version>yourVersion</version>
            <exclusions>
                <exclusion>
                    <groupId>org.slf4j</groupId>
                    <artifactId>slf4j-log4j12</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
    </dependencies>
</dependencyManagement>
```

事实上，如果有其它的jar包依赖slf4j-log4j12，这本身就是有错误的。因为应用不应该直接依赖于这些包中的API —— 它们只应该依赖于日志框架API。任何应用都应该把下列和日志系统相关的依赖（如：slf4j-log4j12、logback-classic）设置成<scope>runtime</scope>的。

11.2.2. 在Maven中配置log4j作为日志系统

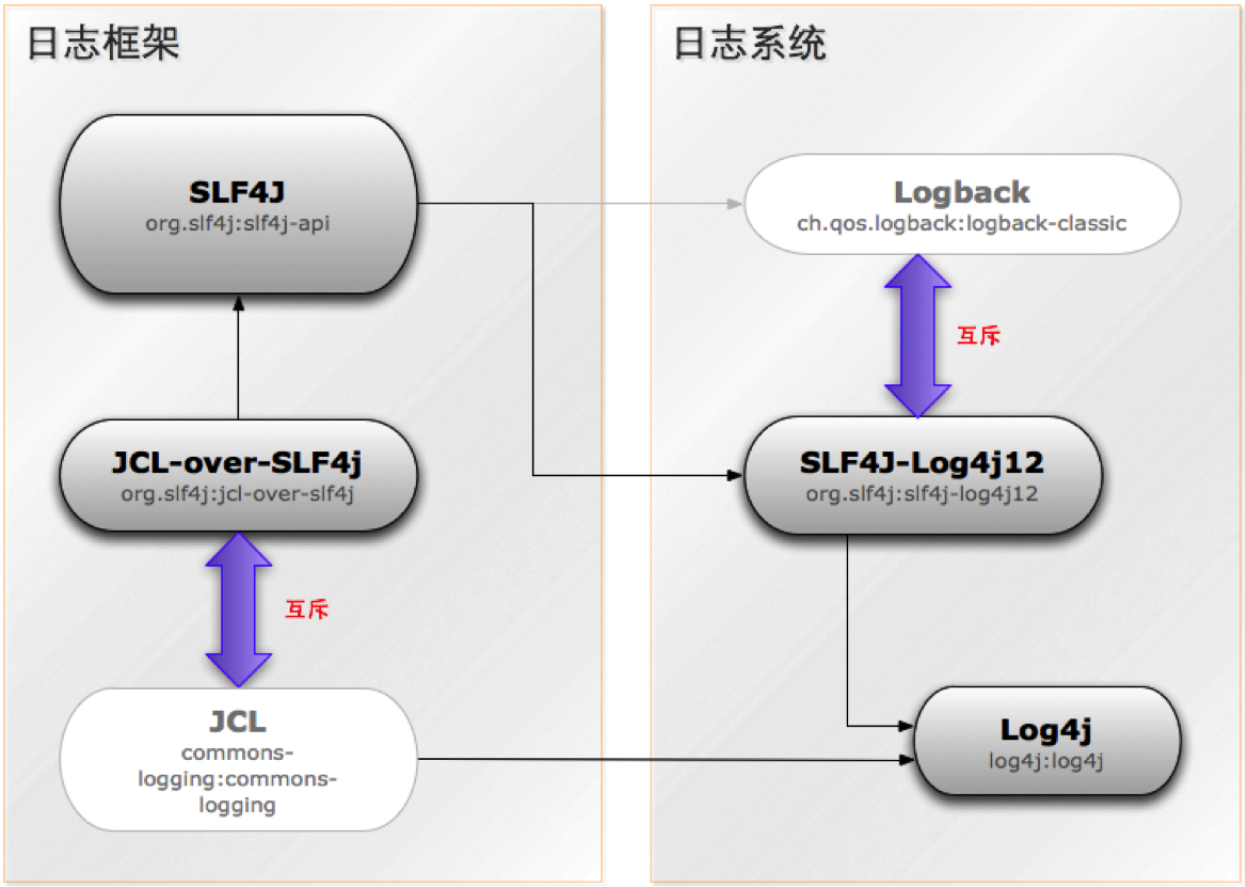


图 11.3. 以log4j作为日志系统

例 11.4. 配置pom.xml以使用log4j

```
<dependencies>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>jcl-over-slf4j</artifactId>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
    </dependency>
</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.6.6</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>jcl-over-slf4j</artifactId>
            <version>1.6.6</version>
        </dependency>
        <dependency>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
            <version>1.1.1</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>1.6.6</version>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.17</version>
            <scope>runtime</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

配置log4j的方法和关注要点和logback相似，请参见例 11.1 “配置pom.xml以使用logback”。除此以外，你需要在项目文件夹下面，执行一下命令：“mvn dependency:tree”，确保没有jar包间接依赖了logback-classic。如果有的话，你可以用下面的配置来排除掉：

例 11.5. 排除间接依赖的logback-classic

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>yourGroupId</groupId>
            <artifactId>yourArtifactId</artifactId>
            <version>yourVersion</version>
            <exclusions>
                <exclusion>
                    <groupId>ch.qos.logback</groupId>
                    <artifactId>logback-classic</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
    </dependencies>
</dependencyManagement>
```

事实上，如果有其它的jar包依赖logback-classic，这本身就是有错误的。因为应用不应该直接依赖于这些包中的API —— 它们只应该依赖于日志框架API。任何应用都应该把下列和日志系统相关的依赖（如：slf4j-log4j12、logback-classic）设置成`<scope>runtime</scope>`的。

11.3. 在WEB应用中配置日志系统

11.3.1. 设置WEB应用

例 11.6. 设置/WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/j2ee  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd
    ">

    <context-param>
        <param-name>loggingRoot</param-name> ❶
        <param-value>/tmp/logs</param-value>
    </context-param>
    <context-param>
        <param-name>loggingLevel</param-name> ❷
        <param-value>INFO</param-value>
    </context-param>
    <context-param>
        <param-name>loggingCharset</param-name> ❸
        <param-value>UTF-8</param-value>
    </context-param>
    <context-param>
        <param-name>log...</param-name> ❹
        <param-value>...</param-value>
    </context-param>

    <listener>
        <listener-class>com.alibaba.citrus.logconfig.LogConfiguratorListener</listener-class> ❺
    </listener>

    <filter>
        <filter-name>mdc</filter-name>
        <filter-class>com.alibaba.citrus.webx.servlet.SetLoggingContextFilter</filter-class> ❻
    </filter>

    <filter-mapping>
        <filter-name>mdc</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

</web-app>
```

❶❷❸❹ 指定日志系统的参数。

❺ 在WEB应用启动的时候，这个listener会被激活，并初始化日志系统。

❻ 将当前请求的信息放到日志系统的MDC中（Mapped Diagnostic Context）。

11.3.1.1. 日志系统的参数

可以在/WEB-INF/web.xml配置文件中集中定义日志系统的参数。

表 11.4. 可配置的日志参数

参数名称	说明
loggingRoot	指定保存日志文件的根目录。如不指定，默认为：“${user.home}/logs”。
loggingLevel	指定日志级别，低于指定级别的日志将不被输出。如不指定，默认为“INFO”。

参数名称	说明
loggingCharset	指定用来生成日志文件的字符集编码。如不指定，默认为当前操作系统的默认字符集编码。
log*	名称以“log”开头的任意<context-param>参数，都将被用作日志系统的参数。

日志系统的参数可被替换到log4j或logback的配置中去，例如，在logback的配置文件中，你可以指定${loggingRoot}来取得存放日志文件的根目录。

将日志参数配置在/WEB-INF/web.xml中，有如下优点：

- 使一套配置参数可同时应用于任意日志系统，包括logback和log4j。
- 便于管理。通常，我们可以利用maven的filtering机制，或者autoconfig插件来生成/WEB-INF/web.xml文件，以便定制上述参数。

11.3.1.2. 自动识别并初始化日志系统

LogConfiguratorListener负责在系统启动的时候初始化日志系统。LogConfiguratorListener会根据下面所列的条件，来自动识别出当前的日志系统，并正确地配置它：

- 假如你在maven的pom.xml中指定log4j为日志系统，那么该listener就会试图用/WEB-INF/log4j.xml来初始化日志系统。
- 假如你在maven的pom.xml中指定logback为日志系统，那么该listener就会试图用/WEB-INF/logback.xml来初始化日志系统。
- 假如你在maven的pom.xml中未指定任何日志系统（不存在logback-classic或slf4j-log4j12），那么listener会报错并失败，整个WEB应用会退出，服务器报告应用启动失败。
- 假如/WEB-INF/logback.xml（或/WEB-INF/log4j.xml）不存在，那么listener会用默认的配置文件来初始化日志。默认的配置会：
  - 把WARN级别以上的日志打印在STDERR中，
  - 把WARN级别以下的日志打印在STDOUT中。

11.3.1.3. 初始化MDC

SetLoggingContextFilter将当前请求的信息放到日志系统的MDC中（Mapped Diagnostic Context）。这样，日志系统就可以打印出诸如下面所示的日志信息：

例 11.7. 利用MDC输出的日志

```
30377 [2010-06-02 15:24:29] - GET❶ /wrongpage.htm❷ [ip=127.0.0.1❸ , ref=http://localhost:8081/index❹
 , ua=Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.6; zh-CN; rv:1.9.2.3) Gecko/20100401 Firefox/3.6.3❺ ,
 sid=scnd32ei11a7❻ ] ……
```

这段日志信息中包含了如下信息：

❶ 用户请求的类型：GET。

❷ 请求的URI：/wrongpage.htm。

❸ 用户IP：127.0.0.1。

❹ 上一个页面的链接referrer：`http://localhost:8081/index`

❺ 用户的浏览器：Mac版的mozilla浏览器。

❻ Session ID：`scnd32ei11a7`。

用户客户端的详细信息，对于发现和追踪错误非常有帮助。

SetLoggingContextFilter是一个可选的filter—— 即使没有它，Webx的`<setLoggingContext />` valve也会做同样的事情。但是把这些信息放在filter中，有利于及早记录用户的信息。

11.3.2. 定制/WEB-INF/logback.xml（或/WEB-INF/log4j.xml）

11.3.2.1. 可用的参数

在日志配置文件中，你可以使用以下参数：

表 11.5. 日志配置文件中可用的参数

在/WEB-INF/web.xml中定义的所有日志参数	
${loggingRoot}	代表保存日志文件的根目录。
${loggingCharset}	代表用来生成日志文件的字符集编码。
${loggingLevel}	代表日志级别，低于指定级别的日志将不被输出。
${log*}	自定义参数，其中“*”代表任意名称。
由系统自动取得的参数	
${loggingHost}	代表当前的服务器名称
${loggingAddress}	代表当前的服务器IP地址

11.3.2.2. 可用的MDC参数

在appender pattern中，你可以使用以下MDC参数：

表 11.6. 日志配置文件中可用的MDC参数

参数名	说明
%X{productionMode}	系统运行的模式。如果系统以开发模式运行，将会显示Development Mode；否则将会显示Production Mode。在生产环境中启动开发模式会引起严重的性能和安全问题。将系统运行的模式打印在日志中，可以作为一种提醒。
%X{method}	请求类型，如：GET或POST
%X{requestURL}	完整的URL，如：http://localhost/test
%X{requestURLWithQueryString}	完整的URL，以及query string，如：http://localhost/test?id=1
%X{requestURI}	不包括host信息的URI，如：/test
%X{requestURIWithQueryString}	不包括host信息的URI，以及query string，如：/test?id=1
%X{queryString}	URL参数，如：id=1
%X{remoteAddr}	客户端地址
%X{remoteHost}	客户端域名
%X{userAgent}	客户端浏览器信息
%X{referrer}	上一个页面的URL
%X{cookies}	所有cookies的名称，如：[cookie1, cookie2]

参数名	说明
%X{cookie.*}	特定cookie的值，如：%X{cookie.JSESSIONID}，将显示当前session的ID
%X{*}	其它由应用程序或框架置入MDC的参数

11.3.2.3. Logback配置示例

例 11.8. Logback配置示例（**/WEB-INF/logback.xml**）

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="false">
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <target>System.out</target>
        <encoding>${loggingCharset}</encoding>
        <layout class="ch.qos.logback.classic.PatternLayout">
            <pattern><![CDATA[
%n%-4r [%d{yyyy-MM-dd HH:mm:ss}] %X{productionMode} - %X{method} %X{requestURIWithQueryString} [ip=
%X{remoteAddr}, ref=%X{referrer}, ua=%X{userAgent}, sid=%X{cookie.JSESSIONID}]%n  %-5level %logger{35} -
 %m%n
            ]]></pattern>
        </layout>
        <filter class="com.alibaba.citrus.logconfig.logback.LevelRangeFilter">
            <levelMax>INFO</levelMax>
        </filter>
    </appender>

    <appender name="STDERR" class="ch.qos.logback.core.ConsoleAppender">
        <target>System.err</target>
        <encoding>${loggingCharset}</encoding>
        <layout class="ch.qos.logback.classic.PatternLayout">
            <pattern><![CDATA[
%n%-4r [%d{yyyy-MM-dd HH:mm:ss}] %X{productionMode} - %X{method} %X{requestURIWithQueryString} [ip=
%X{remoteAddr}, ref=%X{referrer}, ua=%X{userAgent}, sid=%X{cookie.JSESSIONID}]%n  %-5level %logger{35} -
 %m%n
            ]]></pattern>
        </layout>
        <filter class="com.alibaba.citrus.logconfig.logback.LevelRangeFilter">
            <levelMin>WARN</levelMin>
        </filter>
    </appender>

    <appender name="PROJECT" class="ch.qos.logback.core.FileAppender">
        <file>${loggingRoot}/${localHost}/petstore.log</file>
        <encoding>${loggingCharset}</encoding>
        <append>false</append>
        <layout class="ch.qos.logback.classic.PatternLayout">
            <pattern><![CDATA[
%n%-4r [%d{yyyy-MM-dd HH:mm:ss}] %X{productionMode} - %X{method} %X{requestURIWithQueryString} [ip=
%X{remoteAddr}, ref=%X{referrer}, ua=%X{userAgent}, sid=%X{cookie.JSESSIONID}]%n  %-5level %logger{35} -
 %m%n
            ]]></pattern>
        </layout>
    </appender>

    <root>
        <level value="${loggingLevel}" />
        <appender-ref ref="STDERR" />
        <appender-ref ref="STDOUT" />
        <appender-ref ref="PROJECT" />
    </root>
</configuration>
```

更详细配置方法请参考logback官方文档：http://logback.qos.ch/manual/configuration.html。

请特别留意示例中参数的写法，如“${loggingRoot}”；以及appender pattern中MDC参数的的写法，如：“%X{method}”、“%X{requestURIWithQueryString}”等。

11.3.2.4. Log4j配置示例

例 11.9. Log4j配置示例（/WEB-INF/log4j.xml）

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
    <appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">
        <param name="target" value="System.out" />
        <param name="encoding" value="${loggingCharset}" />
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern"
                value="%n%-4r [%d{yyyy-MM-dd HH:mm:ss}] %X{productionMode} - %X{method}
 %X{requestURIWithQueryString} [ip=%X{remoteAddr}, ref=%X{referrer}, ua=%X{userAgent}, sid=
%X{cookie.JSESSIONID}]%n  %-5level %logger{35} - %m%n"
             />
        </layout>
        <filter class="org.apache.log4j.varia.LevelRangeFilter">
            <param name="levelMax" value="INFO" />
        </filter>
    </appender>

    <appender name="STDERR" class="org.apache.log4j.ConsoleAppender">
        <param name="target" value="System.err" />
        <param name="encoding" value="${loggingCharset}" />
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern"
                value="%n%-4r [%d{yyyy-MM-dd HH:mm:ss}] %X{productionMode} - %X{method}
 %X{requestURIWithQueryString} [ip=%X{remoteAddr}, ref=%X{referrer}, ua=%X{userAgent}, sid=
%X{cookie.JSESSIONID}]%n  %-5level %logger{35} - %m%n"
             />
        </layout>
        <filter class="org.apache.log4j.varia.LevelRangeFilter">
            <param name="levelMin" value="WARN" />
        </filter>
    </appender>

    <appender name="PROJECT" class="org.apache.log4j.FileAppender">
        <param name="file" value="${loggingRoot}/${localHost}/myapp.log" />
        <param name="encoding" value="${loggingCharset}" />
        <param name="append" value="true" />
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern"
                value="%n%-4r [%d{yyyy-MM-dd HH:mm:ss}] %X{productionMode} - %X{method}
 %X{requestURIWithQueryString} [ip=%X{remoteAddr}, ref=%X{referrer}, ua=%X{userAgent}, sid=
%X{cookie.JSESSIONID}]%n  %-5level %logger{35} - %m%n"
             />
        </layout>
    </appender>

    <root>
        <level value="${loggingLevel}" />
        <appender-ref ref="STDOUT" />
        <appender-ref ref="STDERR" />
        <appender-ref ref="PROJECT" />
    </root>
</log4j:configuration>
```

更详细配置方法请参考log4j官方文档：http://logging.apache.org/log4j/1.2/manual.html。

请特别留意示例中参数的写法，如“`${loggingRoot}`”；以及appender pattern中MDC参数的的写法，如：“`%X{method}`”、“`%X{requestURIWithQueryString}`”等。

11.3.3. 同时初始化多个日志系统

在某些遗留系统中，有些代码直接用到了Log4j API（例如Log4j Appender）。假如，我们仍然希望SLF4J以logback作为日志系统，但是保持这些老代码继续不变地使用log4j来记录日志。这样我们就需要同时初始化logback和log4j。

例 11.10. 同时初始化Logback和Log4j

首先，你需要确保在`pom.xml`中，同时包含log4j和logback-classic这两个依赖，但是请*一定不要包含slf4j-log4j12*这个包，因为它会和logback-classic起冲突。

下面的配置在例 11.1 “配置`pom.xml`以使用logback”基础上，添加了log4j的依赖：

```
<dependencies>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>jcl-over-slf4j</artifactId>
    </dependency>
    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
    </dependency>
</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.6.6</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>jcl-over-slf4j</artifactId>
            <version>1.6.6</version>
        </dependency>
        <dependency>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
            <version>1.1.1</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
            <version>1.0.6</version>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.17</version>
            <scope>runtime</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

然后，你需要在/WEB-INF/web.xml中增加logSystem参数。

下面的配置在例 11.6 “设置/WEB-INF/web.xml”基础上，添加了所需的参数：

```
<web-app>

    <context-param>
        <param-name>logSystem</param-name>
        <param-value>log4j, logback</param-value>
    </context-param>
    ...

</web-app>
```

以上这段/WEB-INF/web.xml的配置，告诉LogConfiguratorListener同时初始化两个日志系统：log4j和logback。它们的配置文件分别是：/WEB-INF/log4j.xml和/WEB-INF/logback.xml。假如文件不存在也没关系，LogConfiguratorListener会用系统默认的配置文件来初始化它们。

11.4. 常见错误及解决

11.4.1. 查错技巧

11.4.1.1. 检查提示信息

分析错误前，*先检查一下日志系统输出的提示信息，往往可以节省很多时间*。
当LogConfiguratorListener启动时，将会在STDERR中打印信息，像下面这个样子：

例 11.11. 日志初始化时的提示信息（STDERR）

```
2010-06-02 16:57:28.021:INFO:/:Initializing log4j system ❶
INFO: configuring "log4j" using file:/Users/…/WEB-INF/log4j.xml ❷
 - with property localAddress = 10.16.58.5 ❸
 - with property localHost = baobao-macbook-pro.local ❹
 - with property loggingCharset = UTF-8 ❺
 - with property loggingLevel = warn ❻
 - with property loggingRoot = /tmp/logs ❼
```

通过这些信息，你可以检查如下内容：

❶ 是否选择了正确的日志系统，如：log4j或logback，抑或两样都有。

❷ 是否选择了正确的日志配置文件，如：/WEB-INF/log4j.xml。

❸❹ 日志文件的参数，如根目录、字符集编码、日志级别等信息。
❺❻
❼

11.4.1.2. 查看class真实归属的jar包位置

有时，因为各种原因导致应用找到了错误的jar包，从而产生神秘的错误。例如，你以为你使用了SLF4J的最新版，然而在服务器上存在一个SLF4J的老版本，并且其class loader优先级比新版本更高。在这种情况下，应用会引用高优先级class loader中的老版本的class。这可能导致错误。

发现这类错误的有效的方法，是在应用程序的任意点设置断点（利用eclipse远程调试功能），当系统停留在断点处时，执行如下的java代码，查看其值：

例 11.12. 查看class真实归属的jar包位置

```
getClass().getClassLoader().getResource(getClass().getName().replace('.', '/') + ".class")
```

另外，Webx开发模式所提供的详细出错页面中，也会列出stacktrace中每一个class的真实jar包位置。

11.4.2. 异常信息：No log system exists

报这个错的原因可能是：

- 不存在slf4j-log4j12、logback-classic等任何一个日志系统的实现。
- Slf4j的版本和日志系统的版本不匹配，例如，slf4j为1.4.3版，而slf4j-log4j12为1.6.6版。

解决方法：

- 用`mvn dependency:tree`命令查看所有的依赖包，排除以上错误。
- 查看服务器环境（如jboss），查看是不是存在不正确版本的jar包，被优先于应用jar包而加载了。参见第 11.4.1.2 节 “查看class真实归属的jar包位置”。

11.4.3. 异常信息：`NoSuchMethodError`: `org.slf4j.MDC.getCopyOfContextMap()`

报这个错的原因是：

- SLF4J的版本过老。MDC.getCopyOfContextMap()方法是从SLF4J 1.5.1时加入的，假如你的SLF4J是之前的版本，就会报错。

解决方法：

- 用`mvn dependency:tree`查看所有的依赖包，排除以上错误。
- 查看服务器环境（如jboss），查看是不是存在不正确版本的jar包，被优先于应用jar包而加载了。

11.4.4. `STDERR`输出：Class path contains multiple SLF4J bindings

SLF4J在`STDERR`报如下错误：

例 11.13. Class path contains multiple SLF4J bindings

```
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/…/WEB-INF/lib/logback-classic-0.9.18.jar!/org/slf4j/impl/
StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/…/WEB-INF/lib/slf4j-log4j12-1.5.11.jar!/org/slf4j/impl/
StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
```

报这个错的原因是：

- classpath中存在多个日志系统，使SLF4J无所适从。

解决方法：

- SLF4J已经列出了classpath中所有的日志系统的位置。根据这些信息，你可以调整应用的依赖，或者整理服务器的环境，使之只剩下一个日志系统。

11.4.5. 看不到日志输出

原因可能是日志的配置文件可能有错。

解决方法：

- 首先，查看LogConfiguratorListener输出到STDERR中的信息（参见第 11.4.1.1 节 “检查提示信息”），确定系统：
  - 选择了正确的日志系统；
  - 选择了正确的配置文件；
  - 设置了正确的参数（loggingRoot、loggingLevel等）。

注意

在JBOSS环境中，STDOUT和STDERR会被重定向到Log4j中，然后被输出到一个文件中，通常是log/server.log。你必须从这个日志文件中查看LogConfiguratorListener的输出。

- 假如以上信息均正确，查看日志配置文件/WEB-INF/log4j.xml或/WEB-INF/logback.xml，是否引用了正确的参数，例如：${loggingRoot}、${loggingLevel}等。
- 检查文件系统权限，确保应用有权限创建和修改日志文件。
- 假设你使用log4j作为日志系统，以jboss作为应用服务器。在JBOSS环境中，当log4j被初始化后，STDOUT和STDERR可能会被重新配置到不同的appender中。原先用来记录STDOUT和STDERR的日志文件log/server.log将不会再被使用。建议你设置/WEB-INF/log4j.xml，增加如下内容：

例 11.14. 在log4j中配置jboss服务器日志

```
<appender name="JBOSS_APPENDER" class="org.apache.log4j.FileAppender">
    <param name="file" value="${loggingRootJboss}/server.log" />
    <param name="encoding" value="${loggingCharset}" />
    <param name="append" value="true" />
    <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern"
            value="%n%-4r [%d{yyyy-MM-dd HH:mm:ss}] %X{productionMode} - %X{method}
 %X{requestURIWithQueryString} [ip=%X{remoteAddr}, ref=%X{referrer}, ua=%X{userAgent}, sid=
%X{cookie.JSESSIONID}]%n  %-5level %logger{35} - %m%n"
         />
    </layout>
</appender>

<logger name="STDOUT">
    <appender-ref ref="JBOSS_APPENDER" />
</logger>
<logger name="STDERR">
    <appender-ref ref="JBOSS_APPENDER" />
</logger>
```

这里用到了一个新的变量：${loggingRootJboss}，你需要把它定义在/WEB-INF/web.xml中。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
    ...
    <context-param>
        <param-name>loggingRootJboss</param-name>
        <param-value>${jboss}/log</param-value>
    </context-param>
    ...
</web-app>
```

如果你使用logback作为日志系统，则不需要作如上配置。

11.5. 本章总结

`LogConfiguratorListener`目前只提供了logback和log4j的支持，尽管支持一种新的日志系统是非常容易的，但现在看来，这两种日志系统已经足够我们使用了。

`LogConfiguratorListener`以SLF4J为基础。SLF4J还提供了更多的功能：

- 除了log4j和logback以外，SLF4J还支持几种其它的日志系统；
- 除了jcl-over-slf4j以外，SLF4J还提供了几种对其它legacy日志系统的桥接功能。

详情请见SLF4J的文档：http://www.slf4j.org/docs.html。

部分 V. 辅助工具

第 12 章 AutoConfig工具使用指南

12.1. 需求分析

在一个应用中，我们总是会遇到一些参数，例如：

- 数据库服务器IP地址、端口、用户名；
- 用来保存上传资料的目录。
- 一些参数，诸如是否打开cache、加密所用的密钥名称等等。

这些参数有一个共性，那就是：*它们和应用的逻辑无关，只和当前环境、当前系统用户相关*。以下场景很常见：

- 在开发、测试、发布阶段，使用不同的数据库服务器；
- 在开发阶段，使用Windows的A开发者将用户上传的文件存放在`d:\my_upload`目录中，而使用Linux的B开发者将同样的文件存放在`/home/myname/my_upload`目录中。
- 在开发阶段设置`cache=off`，在生产环境中设置`cache=on`。

很明显，*这些参数不适合被“硬编码”在配置文件或代码中*。因为每一个从源码库中取得它们的人，都有可能需要修改它们，使之与自己的环境相匹配。

12.1.1. 解决方案

12.1.1.1. 运行时替换的placeholders

很多框架支持在运行时刻替换配置文件中的placeholder占位符。例如， Webx/Spring就有这个功能。

例 12.1. 在Webx中定义placeholders

```
<services:property-placeholder />

<services:webx-configuration>
    <services:productionMode>${productionMode:true}</services:productionMode>
</services:webx-configuration>
```

在上面这个例子中，你可以在启动应用时，加上JVM参数：“-DproductionMode=false|true”来告诉系统用哪一种模式来工作。如果不指定，则取默认值“true”。

运行时替换placeholder是一种非常实用的技术，它有如下优缺点：

表 12.1. 运行时替换placeholders的优缺点

优点	缺点
• 配置文件是静态的、不变的。即使采用不同的参数值，你也不需要更改配置文件本身。 • 你可以随时改变参数的值，只需要启动时指定不同的JVM参数、或指定不同的properties文件即可。 • 这种配置对于应用程序各组件是透明的 —— 应用程序不需要做特别的编程，即可使用placeholders。	• 并非所有框架都支持这种技术。 • 支持该技术的框架各有不同的用法。例如：Spring和Log4j都支持placeholder替换，然则它们的做法是完全不同的。Spring通过PropertyPlaceholderConfigurer类来配置，而Log4j则需要在DomConfigurator中把参数传进去。

12.1.1.2. 中心配置服务器（Config Server）

这也是一种运行时技术。它可以在运行时刻，将应用所需的参数推送到应用中。

它有如下优缺点：

表 12.2. 中心配置服务器的优缺点

优点	缺点
• 它可以集中管理所有应用的配置，避免可能的错误； • 它可以在运行时改变参数的值，并推送到所有应用中。参数的更改可立即生效。	• 需要一套独立的服务器系统。性能、可用性（availability）都是必须考虑的问题。 • 对应用不是透明的，有一定的侵入性。应用程序必须主动来配合该技术。因此，该技术不可能适用于所有情况，特别对于第三方提供的代码，很难使用该技术。 • 为了连接到中心配置服务器，你仍然需要配置适当的IP、端口等参数。你需要用其它技术来处理这些参数（例如placeholders）。

12.1.1.3. Maven Filtering机制

Maven提供了一种过滤机制，可以在资源文件被复制到目标目录的同时，替换其中的placeholders。

例 12.2. 配置Maven Filtering机制

假设你的项目目录结构如下：

```
web-project
  |  pom.xml
  |
  └─src
      └─main
          ├─java
          ├─resources
          └─webapp
              └─WEB-INF
                      web.xml
```

在pom.xml中这样写：

```
<build>
    <filters>
        <filter>${user.home}/antx.properties</filter> ❶
    </filters>
    <resources>
        <resource>
            <directory>src/main/resources</directory> ❷
            <includes>
                <include>**.xml</include>
            </includes>
            <filtering>true</filtering>
        </resource>
        <resource>
            <directory>src/main/resources</directory>
            <excludes>
                <exclude>**.xml</exclude>
            </excludes>
        </resource>
    </resources>
    <plugins>
        <plugin>
            <artifactId>maven-war-plugin</artifactId>
            <configuration>
                <webResources>
                    <resource>
                        <directory>src/main/webapp</directory> ❸
                        <includes>
                            <include>WEB-INF/**.xml</include>
                        </includes>
                        <filtering>true</filtering>
                    </resource>
                    <resource>
                        <directory>src/main/webapp</directory>
                        <excludes>
                            <include>WEB-INF/**.xml</include>
                        </excludes>
                    </resource>
                </webResources>
            </configuration>
        </plugin>
    </plugins>
</build>
```

这段pom定义告诉maven：

❶ 用指定的properties文件（`${user.home}/antx.properties`）中的值，替换文件中的placeholders。

❷ 过滤`src/main/resources/`目录中的所有xml文件，替换其中的placeholders。

❸ 过滤`src/webapp/WEB-INF/`目录中的所有xml文件，替换其中的placeholders。

如果上述xml文件中，包含“`${xxx.yyy.zzz}`”这样的placeholders，将被替换成properties文件中的相应值。

和运行时替换placeholders方案相比，Maven Filtering是一个build时进行的过程。它的优缺点是：

表 12.3. Maven Filtering机制的优缺点

优点	缺点
• Maven filtering机制和应用所采用的技术、框架完全无关，对应用完全透明，通用性好。	• Maven filtering机制在build时刻永久性改变被过滤的配置文件的内容，build结束以后无法更改。这将导致

优点	缺点
	一个问题：如果要改变配置文件的参数，必须获取源码并重新build。 • 缺少验证机制。当某个placeholder拼写错误；当properties中的值写错；当某配置文件中新增了一个placeholder，而你的properties文件中没有对应的值时，maven不会提醒你。而这些错误往往被拖延到应用程序运行时才会被报告出来。

12.1.1.4. AutoConfig机制

AutoConfig是一种类似于Maven Filtering的*build时刻的工具*。这意味着该机制与应用所采用的技术、框架完全无关，对应用完全透明，具有良好的通用性。同时，AutoConfig与运行时的配置技术并不冲突。它可以和运行时替换的placeholders以及中心配置服务器完美并存，互为补充。

AutoConfig书写placeholder的方法和Maven Filtering机制完全相同。换言之，Maven Filtering的配置文件模板（前例中的`/WEB-INF/**.xml`）可以不加修改地用在AutoConfig中。

然而，autoconfig成功克服了Maven Filtering的主要问题。

表 12.4. Maven Filtering和AutoConfig的比较

问题	Maven Filtering	AutoConfig
如何修改配置文件的参数?	Maven Filtering必须获得源码并重新build;	而AutoConfig不需要提取源码，也不需要重新build，即可改变*目标文件*中所有配置文件中placeholders的值。
如何确保placeholder替换的正确性?	Maven Filtering不能验证placeholder值的缺失和错误;	但AutoConfig可以对placeholder及其值进行检查。

接下来，我们将将详细介绍AutoConfig的使用方法。

12.2. AutoConfig的设计

很多人会把AutoConfig看作Maven Filtering机制的简单替代品。事实上，这两者的设计初衷有很大的区别。

12.2.1. 角色与职责

为了把事情说清楚，我们必须要定义两种角色：*开发者（Developer）*和*布署者（Deployer）*。

表 12.5. 角色和职责

角色名称	职责
开发者	• 定义应用所需要的properties，及其限定条件； • 提供包含placeholders的配置文件模板。
布署者	• 根据所定义的properties，提供符合限定条件的属性值。 • 调用AutoConfig来生成目标配置文件。

例如，一个宠物店（petstore）的WEB应用中需要指定一个用来上传文件的目录。于是，

表 12.6. Petstore应用中的角色和职责

开发者	布署者
开发者定义了一个property：`petstore.upload_dir`， 限定条件为：“合法的文件系统的目录名”。	布署者取得petstore的二进制发布包，通过AutoConfig了解到，应用需要一个名为`petstore.upload_dir`目录名。 布署者便指定一个目录给petstore，该目录名的具体值可能因不同的系统而异。 AutoConfig会检验该值是否符合限定条件（是否为合法目录名），如果检验通过，就生成配置文件，并将其中的`${petstore.upload_dir}`替换成该目录名。

需要注意的是，一个“物理人”所对应的“角色”不是一成不变的。例如：某“开发者”需要试运行应用，此时，他就变成“布署者”。

12.2.2. 分享二进制目标文件

假设现在有两个team要互相合作，team A的开发者创建了project A，而team B的开发者创建了project B。假定project B依赖于project A。如果我们利用maven这样的build工具，那么最显而易见的合作方案是这样的：

- Team A发布一个project A的版本到maven repository中。
- Team B从maven repository中取得project A的二进制目标文件。

这种方案有很多好处，

- 每个team都可以独立控制自己发布版本的节奏；
- Team之间的关系较松散，唯一的关系纽带就是maven repository。
- Team之间不需要共享源码。

然而，假如project A中有一些配置文件中的placeholders需要被替换，如果使用Maven Filtering机制，就会出现问题。因为Maven Filtering只能在project A被build时替换其中的placeholders，一旦project A被发布到repository中，team B的人将无法修改任何project A中的配置参数。除非team B的人取得project A的源码，并重新build。这将带来很大的负担。

AutoConfig解决了这个问题。因为当team B的人从maven repository中取得project A的二进制包时，仍然有机会修改其配置文件里的placeholders。Team B的人甚至不需要了解project A里配置文件的任何细节，AutoConfig会自动发现所有的properties定义，并提示编辑。

12.2.3. 布署二进制目标文件

布署应用的人（即布署者、deployer）也从中受益。因为deployer不再需要亲手去build源代码，而是从maven repository中取得二进制目标文件即可。

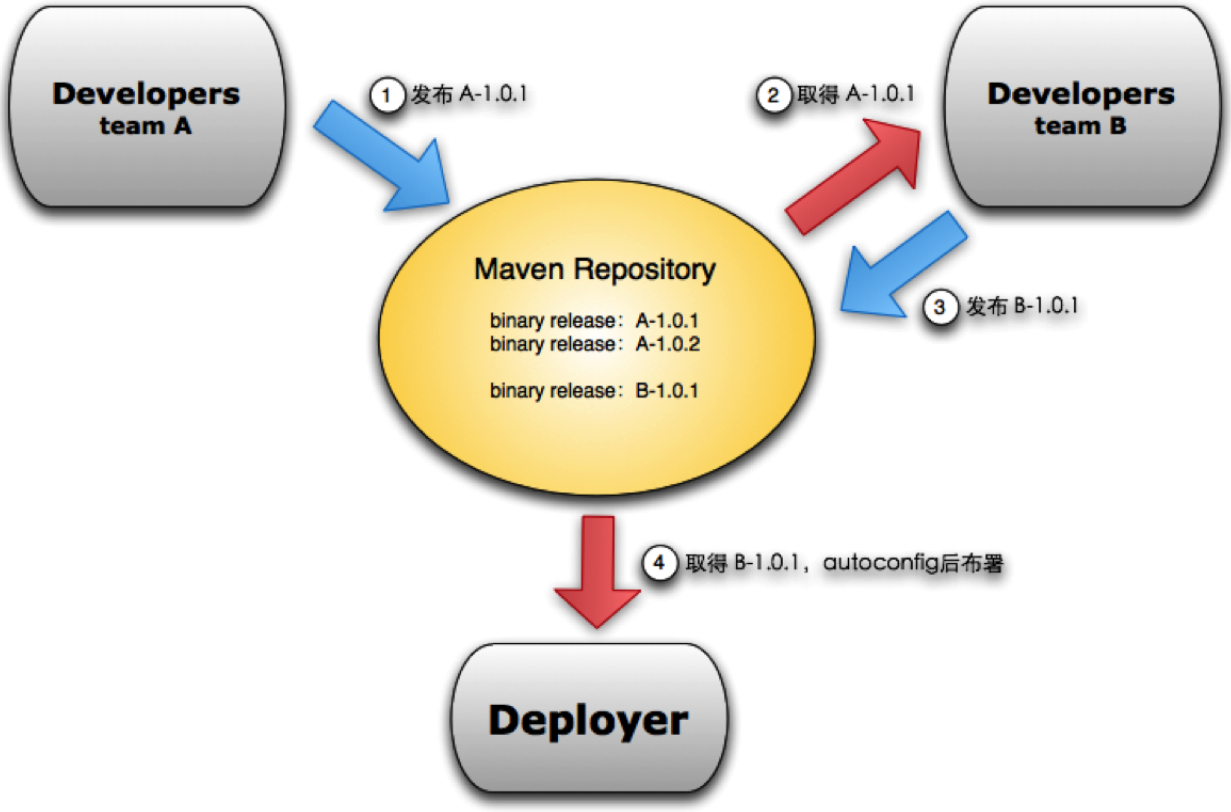


图 12.1. 多团队多角色的合作

从这个意义上讲，AutoConfig不应当被看成是一个build时的简单配置工具，而是一个“*软件安装工具*”。如同我们安装一个Windows软件 —— 我们当然不需要从源码开始build它们，而是执行其安装程序，设定一些参数诸如安装目录、文档目录、可选项等。安装程序就会自动把软件设置好，确保软件可正确运行于当前的Windows环境中。

12.2.4. AutoConfig特性列表

为了满足前面所说的目的，我们将AutoConfig设计成下面的样子：

表 12.7. AutoConfig Features

名称	描述
两种用法	• 既可独立使用（支持Windows和Unix-like平台）。 • 也可以作为maven插件来使用。
对目标文件而不是源文件进行配置	• 可对同一个目标文件反复配置。 • 配置时不依赖于项目源文件。 • 支持嵌套包文件，例如：ear包含war，war又包含jar。 • 高性能，特别对于嵌套的包文件。
验证和编辑properties	• 自动发现保存于war包、jar包、ear包中的properties定义。 • 验证properties的正确性。 • 交互式编辑properties。 • 当配置文件中出现未定义的placeholders时，提示报错。

12.3. AutoConfig的使用 —— 开发者指南

12.3.1. 建立AutoConfig目录结构

和Maven Filtering不同的是，AutoConfig是针对目标文件的配置工具。因此AutoConfig关心的目录结构是*目标文件的目录结构*。不同的build工具，创建同一目标目录结构所需要的源文件的目录结构会各有不同。本文仅以maven标准目录结构为例，来说明源文件的目录结构编排。

12.3.1.1. WAR包的目录结构

这里所说的war包，可以是一个以zip方式打包的文件，也可以是一个展开的目录。下面以maven标准目录为例，说明项目源文件和目标文件的目录结构的对比：

例 12.3. WAR包的源文件和目标文件目录结构

```
war-project（源目录结构）                -> war-project.war（目标目录结构）
 │  pom.xml
 │
 └─src
     └─main
         ├─java
         ├─resources                     -> /WEB-INF/classes
         │      file1.xml                       file1.xml
         │      file2.xml                       file2.xml
         │
         └─webapp                        -> /
             ├─META-INF                  -> /META-INF
             │  └─autoconf               -> /META-INF/autoconf ❶
             │        auto-config.xml            auto-config.xml ❷
             │
             └─WEB-INF                   -> /WEB-INF
                   web.xml                      web.xml
                   file3.xml                    file3.xml
```

❶ `/META-INF/autoconf`目录用来存放AutoConfig的描述文件，以及可选的模板文件。

❷ `auto-config.xml`是用来指导AutoConfig行为的关键描述文件。

创建war包的AutoConfig机制，关键在于创建war目标文件中的`/META-INF/autoconf/auto-config.xml`描述文件。该描述文件对应的maven项目源文件为：`/src/main/webapp/META-INF/autoconf/auto-config.xml`。

12.3.1.2. JAR包的目录结构

这里所说的jar包，可以是一个以zip方式打包的文件，也可以是一个展开的目录。下面以maven标准目录为例，说明项目源文件和目标文件的目录结构的对比：

例 12.4. JAR包的源文件和目标文件目录结构

```
jar-project（源目录结构）                -> jar-project.jar（目标目录结构）
  │  pom.xml
  │
  └─src
      └─main
          ├─java
          └─resources                    -> /
              │  file1.xml                      file1.xml
              │  file2.xml                      file2.xml
              │
              └─META-INF                 -> /META-INF
                  └─autoconf             -> /META-INF/autoconf ❶
                        auto-config.xml          auto-config.xml ❷
```

❶ /META-INF/autoconf目录用来存放AutoConfig的描述文件，以及可选的模板文件。

❷ auto-config.xml是用来指导AutoConfig行为的关键描述文件。

创建jar包的AutoConfig机制，关键在于创建jar目标文件中的/META-INF/autoconf/auto-config.xml描述文件。该描述文件对应的maven项目源文件为：/src/main/resources/META-INF/autoconf/auto-config.xml。

12.3.1.3. 普通目录

AutoConfig也支持对普通文件目录进行配置。

例 12.5. 对普通的目录执行AutoConfig

```
directory
  │  file1.xml
  │  file2.xml
  │
  └─conf ❶
      auto-config.xml ❷
```

❶ 默认情况下，AutoConfig在/conf目录中寻找AutoConfig的描述文件，以及可选的模板文件。

❷ auto-config.xml是用来指导AutoConfig行为的关键描述文件。

12.3.2. 建立auto-config.xml描述文件

AutoConfig系统的核心就是auto-config.xml描述文件。该描述文件中包含两部分内容：

1. 定义properties：properties的名称、描述、默认值、约束条件等信息；

2. 指定包含placeholders的模板文件。

下面是auto-config.xml文件的样子：（以petstore应用为例）

例 12.6. AutoConfig描述文件示例

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
    <group>

        <property name="petstore.work"
                  description="应用程序的工作目录" /> ❶

        <property name="petstore.loggingRoot"
                  defaultValue="${petstore.work}/logs"
                  description="日志文件目录" /> ❷

        <property name="petstore.upload"
                  defaultValue="${petstore.work}/upload"
                  description="上传文件的目录" /> ❸

        <property name="petstore.loggingLevel"
                  defaultValue="warn"
                  description="日志文件级别"> ❹

            <validator name="choice"
                       choice="trace, debug, info, warn, error" /> ❺

        </property>

    </group>
    <script>
        <generate template="WEB-INF/web.xml" /> ❻
        <generate template="WEB-INF/common/resources.xml" />
    </script>
</config>
```

❶❷❸❹ 定义properties

❺ 定义property的验证规则（可选）

❻ 生成配置文件的指令。

12.3.2.1. 定义properties

定义一个property的完整格式如下：

例 12.7. 定义一个property

```
<property
    name="..."
    [defaultValue="..."]
    [description="..."]
    [required="true|false"]
>
    <validator name="..." />
    <validator name="..." />
    ...
</property>
```

可用的property参数包括：

表 12.8. 定义property时可用的参数

参数名	说明
name	Property名称。

参数名	说明
defaultValue（可选）	默认值。默认值中可包含对其它property的引用，如${petstore.work}/logs。
description（可选）	对字段的描述，这个描述会显示给deployer，这对他理解该property非常重要。
required（可选）	是否“必填”，默认为true。如果deployer未提供必填项的值，就会报错。

12.3.2.2. 定义property的验证规则

目前，有以下几种验证器：

表 12.9. 可用的property验证规则

验证规则	说明
`<validator name="boolean" />`	Property值必须为true或false。
`<validator name="choice" choice="trace, debug, info, warn, error" />`	Property值必须为choice所定义的值之一。
`<validator name="email" />`	Property值必须为合法的email格式。
`<validator name="fileExist" [file="WEB-INF/web.xml"] />`	Property值必须为某个存在的文件或目录。 如果指定了file，那就意味着property值所指的目录下，必须存在file所指的文件或子目录。
`<validator name="hostExist" />`	Property值必须为合法的IP地址，或者可以解析得到的域名。
`<validator name="keyword" />`	Property值必须为字母、数字、下划线的组合。
`<validator name="number" />`	Property值必须为数字的组合。
`<validator name="regexp" regexp="..." [mode="exact|prefix|contain"] />`	Property值必须符合regexp所指的正则表达式。 其中，mode为匹配的方法： • 完全匹配exact • 前缀匹配prefix • 包含contain 如未指定mode，默认mode为contain。
`<validator name="url" [checkHostExist="false"] [protocols="http, https"] [endsWithSlash="true"] />`	Property值必须是合法URL。 假如指定了checkHostExist=true，那么还会检查域名或IP的正确性； 假如指定了protocols，那么URL的协议必须为其中之一； 假如指定了endsWithSlash=true，那么URL必须以/结尾。

12.3.2.3. 生成配置文件的指令

描述文件中，每个<generate>标签指定了一个包含placeholders的配置文件模板，具体格式为：

例 12.8. 生成配置文件的指令

```
<generate
    template="..."
    [destfile="..."]
    [charset="..."]
    [outputCharset="..."]
>
```

下面是参数的说明：

表 12.10. 生成配置文件的指令参数

参数名	说明
template	需要配置的模板名。 模板名为相对路径，相对于当前jar/war/ear包的根目录。
destfile（可选）	目标文件。 如不指定，表示目标文件和模板文件相同。
charset（可选）	模板的字符集编码。 XML文件不需要指定charset，因为AutoConfig可以自动取得XML文件的字符集编码； 对其它文件必须指定charset。
outputCharset（可选）	目标文件的输出字符集编码。 如不指定，表示和模板charset相同。

12.3.3. 建立模板文件

12.3.3.1. 模板文件的位置

定义完auto-config.xml描述文件以后，就可以创建模板了。模板放在哪里呢？举例说明。

例 12.9. 模板文件的位置

假设在一个典型的WEB应用中，你的auto-config.xml中包含指定了如下模板：

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
    <group>
        ...
    </group>
    <script>
        <generate template="WEB-INF/classes/file1.xml" />
        <generate template="WEB-INF/classes/file2.xml" />
        <generate template="WEB-INF/file3.xml" />
    </script>
</config>
```

那么，你可以把file1.xml、file2.xml、file3.xml放在下面的位置：

```
war-project（源目录结构）                -> war-project.war（目标目录结构）
  |  pom.xml
  |
  └─src
      └─main
          ├─java
          ├─resources                    -> /WEB-INF/classes
          |      file1.xml                       file1.xml - 建议放在这里
          |      file2.xml                       file2.xml - 建议放在这里
          |
          └─webapp
              ├─META-INF
              |   └─autoconf
              |       |  auto-config.xml
              |       |
              |       └─WEB-INF          -> /WEB-INF
              |           |  file3.xml             file3.xml - 也可以放在这里
              |           |
              |           └─classes      -> /WEB-INF/classes
              |                 file1.xml        file1.xml - 也可以放在这里
              |                 file2.xml        file2.xml - 也可以放在这里
              |
              └─WEB-INF                  -> /WEB-INF
                    file3.xml                    file3.xml - 建议放在这里
```

AutoConfig的寻找模板的逻辑是：

- 如果在`auto-config.xml`所在的目录下发现模板文件，就使用它；
- 否则在包的根目录中查找模板文件；如果两处均未找到，则报错。

12.3.3.2. 模板的写法

书写模板是很简单的事，你只要：

- 把需要配置的点替换成placeholder：“`${property.name}`”。当然，你得确保property.name被定义在`auto-config.xml`中。
- 假如模板中包含*不希望被替换的*运行时的placeholder“*`${...}`*”，需要更改成“*`${D}`*`{...}`”。

例 12.10. 模板示例

```
...
<context-param>
    <param-name>loggingRoot</param-name>
    <param-value>${petstore.loggingRoot}</param-value>
</context-param>
<context-param>
    <param-name>loggingLevel</param-name>
    <param-value>${petstore.loggingLevel}</param-value>
</context-param>
...
${D}{runtime.placeholder}
```

此外，AutoConfig模板其实是由Velocity模板引擎来渲染的。因此，所有的placeholder必须能够通过velocity的语法。

例 12.11. 使用不符合velocity语法的placeholders

例如，下面的placeholder被velocity看作非法：

```
${my.property.2}
```

解决的办法是，改写成如下样式：

```
${my_property_2}
```

12.4. AutoConfig的使用 —— 布署者指南

布署者有两种方法可以使用AutoConfig：

- 在命令行上直接运行。
- 在maven中调用AutoConfig plugin。

12.4.1. 在命令行中使用AutoConfig

12.4.1.1. 取得可执行文件

AutoConfig提供了Windows以及Unix-like（Linux、Mac OS等）等平台上均可使用的native可执行程序。可执行程序文件被发布在Maven repository中。

如果你已经配置好了maven，那么可以让maven来帮你下载目标文件。

例 12.12. 让maven帮忙下载AutoConfig可执行文件

请创建一个临时文件：`pom.xml`。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <parent>
        <groupId>com.alibaba.citrus.tool</groupId>
        <artifactId>antx-parent</artifactId>
        <version>1.0.10</version> ❶
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>temp</artifactId>
</project>
```

❶ 文件中的parent pom的版本号（1.0.10）决定了你要取得的AutoConfig的版本号。

然后在命令行上执行如下命令：

```
mvn dependency:copy
```

这样就取得了两个文件：

- `autoconfig-1.0.10.tgz`
- `autoexpand-1.0.10.tgz` - AutoExpand是另一个小工具。它是用来展开war、jar、ear包的。关于AutoExpand的详情，请见第 13 章 *AutoExpand工具使用指南*。

你也可以直接去maven repository中手工下载以上两个包：

- http://code.taobao.org/mvn/repository/com/alibaba/citrus/tool/antx-autoconfig/1.0.10/antx-autoconfig-1.0.10.tgz

- http://code.taobao.org/mvn/repository/com/alibaba/citrus/tool/antx-autoexpand/1.0.10/antx-autoexpand-1.0.10.tgz

取得压缩包以后，可以用下面的命令来展开并安装工具。

表 12.11. 展开并安装工具

Unix-like系统	Windows系统

`tar zxvf autoconfig-1.0.10.tgz`
`tar zxvf autoexpand-1.0.10.tgz`
`cp autoconfig /usr/local/bin`
`cp autoexpand /usr/local/bin`
	`tar zxvf autoconfig-1.0.10.tgz`
	`tar zxvf autoexpand-1.0.10.tgz`
	`copy autoconfig.exe c:\windows\system32`
	`copy autoexpand.exe c:\windows\system32`

12.4.1.2. 执行AutoConfig命令

取得可执行文件以后，就可以试用一下：在命令行上输入autoconfig。不带参数的autoconfig命令会显示出如下帮助信息。

例 12.13. AutoConfig的帮助信息

```
$ autoconfig
Detected system charset encoding: UTF-8
If your can't read the following text, specify correct one like this:
  autoconfig -c mycharset

使用方法：autoconfig [可选参数] [目录名|包文件名]

可选参数：
 -c,--charset                输入/输出编码字符集
 -d,--include-descriptors
                             包含哪些配置描述文件，例如：conf/auto-config.xml，可使用*、**、?通配符，如有多项，用
逗号分隔
 -D,--exclude-descriptors    排除哪些配置描述文件，可使用*、**、?通配符，如有多项，用逗号分隔
 -g,--gui                    图形用户界面（交互模式）
 -h,--help                   显示帮助信息
 -i,--interactive            交互模式：auto|on|off，默认为auto，无参数表示on
 -I,--non-interactive        非交互模式，相当于--interactive=off
 -n,--shared-props-name      共享的属性文件的名称
 -o,--output                 输出文件名或目录名
 -P,--exclude-packages       排除哪些打包文件，可使用*、**、?通配符，如有多项，用逗号分隔
 -p,--include-packages
                             包含哪些打包文件，例如：target/*.war，可使用*、**、?通配符，如有多项，用逗号分隔
 -s,--shared-props           共享的属性文件URL列表，以逗号分隔
 -T,--type                   文件类型，例如：war, jar, ear等
 -t,--text                   文本用户界面（交互模式）
 -u,--userprop               用户属性文件
 -v,--verbose                显示更多信息

总耗费时间：546毫秒
```

最简单的AutoConfig命令如下：

例 12.14. 最简单的AutoConfig命令

```
autoconfig petstore.war
```

无论`petstore.war`是一个zip包还是目录，AutoConfig都会正确地生成其中的配置文件。

12.4.2. 在maven中使用AutoConfig

AutoConfig也可以通过maven plugin来执行。

这种方式使用方式，方便了开发者试运行并测试应用程序。开发者可以在build项目的同时，把AutoConfig也配置好。然而对于非开发的应用测试人员、发布应用的系统管理员来说，最好的方法是使用独立可执行的AutoConfig来配置应用的二进制目标文件。

为了使用maven插件，你需要修改项目的`pom.xml`来设定它。请注意，一般来说，不要在parent `pom.xml`中设定AutoConfig，因为这个设置会作用在每个子项目上，导致不必要的AutoConfig执行。只在生成最终目标文件的子项目`pom.xml`中设定AutoConfig就可以了。例如，对于一个web项目，你可以在生成war包的子项目上设置AutoConfig plugin。

例 12.15. 在pom.xml中设定AutoConfig plugin

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    ...
    <properties>
        ...
        <!-- 定义autoconfig的版本，建议将此行写在parent pom.xml中。 -->
        <autoconfig-plugin-version>1.0.10</autoconfig-plugin-version>
    </properties>
    ...
    <build>
        <plugins>
            <plugin>
                <groupId>com.alibaba.citrus.tool</groupId>
                <artifactId>maven-autoconfig-plugin</artifactId>
                <version>${autoconfig-plugin-version}</version>
                <configuration>
                    <!-- 要进行AutoConfig的目标文件，默认为${project.artifact.file}。
                    <dest>${project.artifact.file}</dest>
                    -->
                    <!-- 配置后，是否展开目标文件，默认为false，不展开。
                    <exploding>true</exploding>
                    -->
                    <!-- 展开到指定目录，默认为${project.build.directory}/${project.build.finalName}。
                    <explodedDirectory>
                        ${project.build.directory}/${project.build.finalName}
                    </explodedDirectory>
                    -->
                </configuration>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>autoconfig</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

这样，每次执行`mvn package`或者`mvn install`时，都会激活AutoConfig，对package目标文件进行配置。

想要避免AutoConfig，只需要一个额外的命令行参数：

例 12.16. 避免执行AutoConfig

```
mvn install -Dautoconfig.skip
```

12.4.3. 运行并观察AutoConfig的结果

第一次执行AutoConfig，无论通过何种方式（独立命令行或maven插件），AutoConfig都会提示你修改user properties文件，以提供所需要的properties值。AutoConfig提供了一套基于文本的交互式界面来编辑这些properties。

例 12.17. 交互式编辑properties

```
┌──────────────────────────────────────────────────────────────
│
│ 您的配置文件需要被更新：
│
│ file:/.../antx.properties
│
│ 这个文件包括了您个人的特殊设置，
│ 包括服务器端口、您的邮件地址等内容。
│
└──────────────────────────────────────────────────────────────

如果不更新此文件，可能会导致配置文件的内容不完整。
您需要现在更新此文件吗？[Yes][No] y
```

当你通过交互式界面填写了所有properties的值，并通过了AutoConfig的验证以后，AutoConfig就开始生成配置文件：

```
即将保存到文件"file:/.../antx.properties"中，确定？[Yes][No] y

┌──────────────────────────────────────────────────────────────
│ 保存文件 file:/.../antx.properties...
│ ------------------------------------------------------------------------------
│petstore.loggingLevel   = warn
│petstore.loggingRoot    = ${petstore.work}/logs
│petstore.upload         = ${petstore.work}/upload
│petstore.work           = /tmp
└──────────────────────────────────────────────────────────────
 已保存至文件：file:/.../antx.properties
Loading file:/.../antx.properties
<jar:file:/.../Work/my/apps/petstore-webx3/target/petstore.war!/>
    Generating WEB-INF/web.xml [UTF-8] => WEB-INF/web.xml [UTF-8]

<jar:file:/.../Work/my/apps/petstore-webx3/target/petstore.war!/>
    Generating WEB-INF/common/resources.xml [UTF-8] => WEB-INF/common/resources.xml [UTF-8]

<jar:file:/.../Work/my/apps/petstore-webx3/target/petstore.war!/>
    Generating log file: META-INF/autoconf/auto-config.xml.log

Expanding: /.../Work/my/apps/petstore-webx3/target/petstore.war
       To: /.../Work/my/apps/petstore-webx3/target/petstore
done.
```

假如发现模板中某个placeholder，并未在`auto-config.xml`中定义，就会出现以下错误：

```
ERROR - Undefined placeholders found in template:
- Template:   META-INF/autoconf/WEB-INF/web.xml
- Descriptor: META-INF/autoconf/auto-config.xml
- Base URL:   file:/.../Work/my/apps/petstore-webx3/target/petstore/
----------------------------------------------------------------
-> petstore.loggingRoot
----------------------------------------------------------------
```

出现错误以后，Maven会报错，并停止build过程。假如你不希望maven停止，可以用下面的命令来执行maven：

例 12.18. 避免maven因为placeholder未定义而停止

```
mvn ... -Dautoconfig.strict=false
```

AutoConfig会生成一个日志文件，就在`auto-config.xml`所在的目录下，名字为：`auto-config.xml.log`。

例 12.19. AutoConfig所生成的日志文件

```
Last Configured at: Fri Jun 18 13:54:22 CST 2010

Base URL: file:/.../Work/my/apps/petstore-webx3/target/petstore/
Descriptor: META-INF/autoconf/auto-config.xml

Generating META-INF/autoconf/WEB-INF/web.xml [UTF-8] => WEB-INF/web.xml [UTF-8]
Generating META-INF/autoconf/WEB-INF/common/resources.xml [UTF-8] => WEB-INF/common/resources.xml [UTF-8]
```

最后，让我们查看一下AutoConfig所生成的文件，其中所有的placeholders应当被替换成你所提供的值了。

例 12.20. AutoConfig生成的结果

```
...
<context-param>
    <param-name>loggingRoot</param-name>
    <param-value>/tmp/logs</param-value>
</context-param>
<context-param>
    <param-name>loggingLevel</param-name>
    <param-value>warn</param-value>
</context-param>
...
${runtime.placeholder}
```

12.4.4. 共享properties文件

当需要配置的内容越来越多时，即使使用AutoConfig这样的机制，也会变得不胜其烦。

假如你的项目包含了好几个模块，而你只负责其中的一个模块。一般来说，你对其它模块的配置是什么并不清楚，事实上你也懒得去关心。但是你为了运行这个项目，你不得不去配置这些模块。假如模块A就是一个你不想关心的模块，但为了运行它，你需要告诉模块A一些参数：数据库连接的参数、域名、端口、文件目录、搜索引擎……可你并不清楚这些参数应该取什么值。

好在AutoConfig提供了一个共享properties文件的方法。

12.4.4.1. 共享的properties文件

你可以创建一系列文件：`module-a-db.properites`，`module-a-searchengine.properties`等等。每个文件中都包含了某个运行环境中的关于module A模块的配置参数。

现在，你可以不关心module A了！你只要使用下面的命令：

例 12.21. 指定共享的properties文件

```
autoconfig -s module-a-db.properties,module-a-searchengine.properties❶ ……
```

❶ `-s`参数代表“共享的properties文件”。

同时，你的antx.properties也被简化了，因为这里只会保存你定义的配置项，而不会包含共享的配置项。

12.4.4.2. 共享整个目录

假如共享的文件很多的话，AutoConfig还有一个贴心的功能，你可以把这些文件按目录来组织：

例 12.22. 按目录组织要被共享的properties文件

```
shared-properties/
  ├─test/                                  // 测试环境的共享配置
  │    module-a-db.properties
  │    module-a-searchengine.properties
  │    module-b.properties
  └─prod/                                  // 生产环境的共享配置
       module-a-db.properties
       module-a-searchengine.properties
       module-b.properties
```

然后，你可以直接在AutoConfig中引用目录：

例 12.23. 共享指定目录中的所有properties文件

```
autoconfig -s shared-propertes/test/ ……
```

AutoConfig就会为你装载这个目录下的所有共享配置文件。（注意，*目录必须以斜杠“/”结尾*）

12.4.4.3. 将共享目录放在http、https或ssh服务器上

AutoConfig还支持从http、https或ssh服务器上取得共享配置文件，只需要将前面例子中的文件名改成http或ssh的URI就可以了：

例 12.24. 共享远程服务器上的properties文件或目录

```
autoconfig -s http://share.alibaba.com/shared-propertes/test/❶  ……
autoconfig -s http://myname@share.alibaba.com/shared-propertes/test/❷  ……
autoconfig -s https://share.alibaba.com/shared-propertes/test/❸  ……
autoconfig -s https://myname@share.alibaba.com/shared-propertes/test/❹  ……
autoconfig -s ssh://myname@share.alibaba.com/shared-propertes/test/❺  ……
```

❶ 共享远程http服务器上的properties文件。

❷ 共享远程http服务器上的properties文件，指定登录用户名。

❸ 共享远程https服务器上的properties文件。

❹ 共享远程https服务器上的properties文件，指定登录用户名。

❺ 共享远程ssh服务器上的properties文件，必须指定用户名。

由于Subversion、Git服务器是支持HTTP/HTTPS协议的，因此将properties文件存放在Subversion或Git服务器上，也是一个极好的办法。由于采用了Subversion或Git，properties文件的版本管理问题也被一举解决了。

需要注意的是，访问http和ssh有可能需要验证用户和密码。当需要验证时，AutoConfig会提示你输入用户名和密码。输入以后，密码将被保存在*$HOME/passwd.autoconfig*文件中，以后就不需要重复提问了。

12.4.4.4. 在多种配置项中切换

当你使用前文所述的`autoconfig -s`命令来生成`antx.properties`文件时，你会发现`antx.properties`中增加了几行特别的内容：

例 12.25. 包含共享文件、目录信息的`antx.properties`文件

```
antx.properties.default  = http://share.alibaba.com/shared-propertes/test/
```

如果你在`-s`参数中指定了多项共享properties文件或目录，那么`antx.properties`中将会这样：

```
antx.properties.default.1  = http://share.alibaba.com/shared-propertes/test/
antx.properties.default.2  = file:/shared-properties/test/my-1.properites
antx.properties.default.3  = file:/shared-properties/test/my-2.properites
```

事实上，AutoConfig还支持多组共享配置，请试用下面的命令：

例 12.26. 使用多组共享配置

```
autoconfig -s http://share.alibaba.com/shared-propertes/test/ -n test❶  ……
```

❶ 为当前共享配置定义一个名字，以后可以用这个名字来简化命令。

这时，antx.properties就会是这个样子：

```
antx.properties = test
antx.properties.test = http://share.alibaba.com/shared-propertes/test/
```

再执行：

```
autoconfig -s http://share.alibaba.com/shared-propertes/prod/ -n prod ……
```

antx.properties就会变成这个样子：

```
antx.properties = prod
antx.properties.test = http://share.alibaba.com/shared-propertes/test/
antx.properties.prod = http://share.alibaba.com/shared-propertes/prod/
```

以后再执行，就不需要再指定`-s`参数了，只需用`-n`参数选择一组共享properties文件即可。例如：

```
autoconfig -n prod❶  ……                    // 使用prod生产环境的参数
autoconfig -n test❷  ……                    // 使用test测试环境的参数
autoconfig ❸  ……                           // 不指定，则使用最近一次所选择的共享文件
```

12.4.5. AutoConfig常用命令

下面罗列了AutoConfig的常用的命令及参数：

12.4.5.1. 指定交互式界面的charset

一般不需要特别指定charset，除非AutoConfig自动识别系统编码出错，导致显示乱码。

运行AutoConfig独立可执行程序	`autoconfig ... -c GBK`

运行AutoConfig maven插件	`mvn ... -Dautoconfig.charset=GBK`

12.4.5.2. 指定交互模式

默认情况下，交互模式为自动（auto）。仅当user properties中的值不满足auto-config.xml中的定义时，才会交互式地引导用户提供properties值。

但你可以强制打开交互模式：

运行AutoConfig独立可执行程序	`autoconfig ... -i` `autoconfig ... -i on`
运行AutoConfig maven插件	`mvn ... -Dautoconfig.interactive` `mvn ... -Dautoconfig.interactive=true`

或强制关闭交互模式：

运行AutoConfig独立可执行程序	`autoconfig ... -I` `autoconfig ... -i off`
运行AutoConfig maven插件	`mvn ... -Dautoconfig.interactive=false`

12.4.5.3. 指定user properties

默认情况下，AutoConfig会按下列顺序查找user properties：

1. 当前目录/antx.properties
2. 当前用户HOME目录/antx.properties

但你可以指定一个自己的properties文件，用下面的命令：

运行AutoConfig独立可执行程序	`autoconfig ... -u my.props`
运行AutoConfig maven插件	`mvn ... -Dautoconfig.userProperties=my.props`

12.4.5.4. 显示详细的信息

默认情况下，AutoConfig只输出重要的信息，但有时你想了解更多内部的情况，只需要用下面的命令：

运行AutoConfig独立可执行程序	`autoconfig ... -v`
运行AutoConfig maven插件	不适用

12.4.5.5. 指定输出文件

默认情况下，AutoConfig所生成的配置文件以及日志信息会直接输出到当前包文件或目录中。例如以下命令会改变petstore.war的内容：

```
autoconfig petstore.war
```

但你可以指定另一个输出文件或目录，这样，原来的文件或目录就不会被修改：

运行AutoConfig独立可执行程序	`autoconfig petstore.war -o petstore-configured.war`
运行AutoConfig maven插件	不适用

12.4.5.6. 避免执行AutoConfig

将AutoConfig和maven package phase绑定以后，每次build都会激活AutoConfig。假如你想跳过这一步，只需要下面的命令：

运行AutoConfig独立可执行程序	不适用
运行AutoConfig maven插件	`mvn ... -Dautoconfig.skip`

12.4.5.7. 避免中断maven build

默认情况下，假如发现有未定义的placeholders，AutoConfig会报错并中止maven的执行。假如你不想中断maven build，可以这样做：

运行AutoConfig独立可执行程序	不适用
运行AutoConfig maven插件	`mvn ... -Dautoconfig.strict=false`

12.5. 本章总结

AutoConfig是一个简单而有用的小工具，弥补了Maven Filtering及类似机制的不足。但它还有不少改进的余地。

- 界面不够直观。如果能够通过GUI或WEB界面来配置，就更好了。
- Properties validator目前不易扩展。
- 缺少集成环境的支持。

第 13 章 AutoExpand工具使用指南

13.1. AutoExpand工具简介

AutoExpand是一个小工具，可以快速地把应用包展开到目录中。

13.1.1. Java、JavaEE打包的格式

Java和JavaEE的应用通常被打成一个ZIP格式的包。

表 13.1. 标准的Java、JavaEE包的格式

包类型	说明
jar	Java ARchive。共享类库，EJB，独立应用。
war	Web application ARchive。WEB应用。
ear	Enterprise ARchive。企业级应用。
rar	Resource Adapter Archive。

其中jar和war包是当今最常用的格式。其次还有ear。

- Ear包中可以包含多个jar包（包括ejb-jar包）、rar包、war包。
- War包中可以包含多个jar包。

例 13.1. 包的内部格式

下面是一个典型的ear包的内部格式：

```
myapp.ear
  |   foo-ejb.jar             // 内嵌ejb-jar包。
  |   bar.jar                 // 内嵌普通jar包。
  |   myweb.war               // 内嵌war包。
  |
  └─META-INF
        application.xml       // EAR描述文件。
```

下面是一个典型的war包的内部格式：

```
myweb.war
  |  index.jsp
  |
  ├─images
  ├─META-INF
  └─WEB-INF
     |  web.xml             // WAR描述文件。
     |
     ├─classes
     |     foo.class        // Java类文件。
     |
     └─lib
           bar.jar          // 内嵌jar包。
           baz.jar
```

13.1.2. 应用布署的方式

多数应用服务器都支持两种类型的布署方式：以包的形式布署，或者以展开目录的形式布署。

表 13.2. 布署应用的方法

方法	用途
以包的形式布署	你可以把WEB应用的war包直接发布在应用服务器上，应用服务器不需要把包打开就能运行它。 发布一个应用包（只有一个文件），对于deployer来说是比较方便的。通常他只需要把这个包往指定的应用服务器目录一丢，就可以把应用跑起来。 但是以包形式发布的应用不太好调试，因为你为了修改包中的任何一个文件，都必须重新打包。这是很费时的工作。
以展开目录的形式布署	以展开目录的形式布署的应用，更适合于开发阶段。这样，开发者可以方便地替换应用中的任何一个文件，而不需要重新打包。这节省了很多时间。

13.1.3. AutoExpand的用武之地

AutoExpand的功能就是把包文件展开到文件夹中。

事实上，对于多数情况来说，用AutoExpand来展开一个包，和直接使用jar命令来展开包是没有差别的。例如，展开一个war包，可以用下面两种方法：

例 13.2. 用AutoExpand展开一个war包

```
$ autoexpand myweb.war
Detected system charset encoding: UTF-8
If your can't read the following text, specify correct one like this:
  autoexpand -c mycharset

Expanding: /.../myweb.war
       To: /.../myweb
done.

总耗费时间：762毫秒
```

例 13.3. 用jar命令展开一个war包

```
$ mkdir myweb
$ cd myweb
$ jar xvf ../myweb.war
  创建：META-INF/
  解压 META-INF/MANIFEST.MF
  ...
```

但是使用AutoExpand有如下好处：

- 可展开嵌套的包，例如：一个ear中包含war包，用AutoExpand可以一举把它们同时展开。
- 支持更多选项，例如：更新、覆盖、删除多余文件等。
- 比jar命令速度更快。

13.2. AutoExpand的使用

13.2.1. 取得AutoExpand

请参考第 12.4.1.1 节 “取得可执行文件”。

13.2.2. 执行AutoExpand

直接输入autoexpand可得到如下帮助信息。

例 13.4. AutoExpand的帮助信息

```
$ autoexpand
Detected system charset encoding: UTF-8
If your can't read the following text, specify correct one like this:
  autoexpand -c mycharset

使用方法：antxexpand [可选参数] 文件名 [目标目录]

可选参数：
 -c,--charset                输入/输出编码字符集
 -e,--expand-ejb-jar         是否展开ejb-jar（yes|no），默认为no
 -h,--help                   显示帮助信息
 -k,--keep-redundant-files   如果目标目录中有多余的文件，是否保持而不删除，默认为no
 -o,--overwrite              如果目标目录中的文件比zip文件中的项要新，是否覆盖之，默认为no
 -r,--expand-rar             是否展开rar（yes|no），默认为yes
 -v,--verbose                显示更多信息
 -w,--expand-war             是否展开war（yes|no），默认为yes

总耗费时间：203毫秒
```

最简单的AutoExpand命令如下：

例 13.5. 最简单的AutoExpand命令

```
autoexpand myweb.war
```

这条命令将myweb.war展开到myweb目录中。

你也可以指定一个输出目录：

例 13.6. 执行AutoExpand命令：指定输出目录

```
autoexpand myweb.war myweb-expanded.war
```

这条命令将myweb.war展开到myweb-expanded.war目录中。*在目录名中指定后缀（如.war）是一个好主意*。这样，同一个名称（*.war）既可作为目录名，也可作为包名。你可以在目录和包文件之间自由地切换，而不需要改动服务器的脚本或配置。

你可以用一条命令展开嵌套的包。例如：

例 13.7. 执行AutoExpand命令：展开嵌套的包

```
autoexpand myapp.ear myapp-expanded.ear
```

这条命令可以一举把ear以及ear中的所有war都展开。

例 13.8. 展开ear以及ear中的所有war

```
myapp-expanded.ear
|  foo-ejb.jar
|  bar.jar
|
├─myweb.war        // 展开嵌套的war
|   |  index.jsp
|   |
|   ├─images
|   ├─META-INF
|   └─WEB-INF
|       |  web.xml
|       |
|       ├─classes
|       |     foo.class
|       |
|       └─lib
|             bar.jar
|             baz.jar
|
└─META-INF
      application.xml
```

13.2.3. AutoExpand和AutoConfig的合作

你可以让AutoConfig maven插件在配置完应用以后，将应用展开到指定的目录。

例 13.9. AutoConfig完成后，再用AutoExpand展开

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    ...
    <properties>
        ...
        <!-- 定义autoconfig的版本，建议将此行写在parent pom.xml中。 -->
        <autoconfig-plugin-version>1.0.10</autoconfig-plugin-version>
    </properties>
    ...
    <build>
        <plugins>
            <plugin>
                <groupId>com.alibaba.citrus.tool</groupId>
                <artifactId>maven-autoconfig-plugin</artifactId>
                <version>${autoconfig-plugin-version}</version>
                <configuration>
                    ...
                    <!-- 配置后，是否展开目标文件，默认为false，不展开。 -->
                    <exploding>true</exploding>
                    <!-- 展开到指定目录，默认为${project.build.directory}/${project.build.finalName}。 -->
                    <explodedDirectory>
                        ${project.build.directory}/${project.build.finalName}
                    </explodedDirectory>
                </configuration>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>autoconfig</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

关于AutoConfig详情，请看第 12 章 *AutoConfig工具使用指南*。

13.3. AutoExpand的参数

AutoExpand可用的参数包括：

表 13.3. AutoExpand命令的参数

参数名	说明
-w 或 --expand-war	是否展开war（yes或no），默认为yes。
-r 或 --expand-rar	是否展开rar（yes或no），默认为yes。
-e 或 --expand-ejb-jar	是否展开ejb-jar（yes或no），默认为no。 展开ejb-jar需要读取/META-INF/application.xml文件，因此会降低展开的速度。
-o 或 --overwrite	如果目标目录中的文件比包文件中的项要新，是否覆盖之，默认为no。 指定该参数可强制覆盖文件。 如果你不信任文件中的时间戳，就可以指定这个参数。
-k 或 --keep-redundant-files	如果目标目录存在多余的文件（也就是包里不存在的文件），AutoExpand会将它们删除，以确保展开后的目录内容与包的内容完全一致，不多不少。

参数名	说明
	指定这个参数可以避免AutoExpand删除多余的文件。
-v 或 --verbose	显示更多信息。
-c 或 --charset	如果AutoExpand显示的信息是乱码，请通过这个参数指定正确的字符集编码。

13.4. 本章总结

AutoExpand虽然小，但很好用，可以作为jar命令部分功能的替代品。此外，AutoConfig也利用AutoExpand来展开经过配置的包文件。