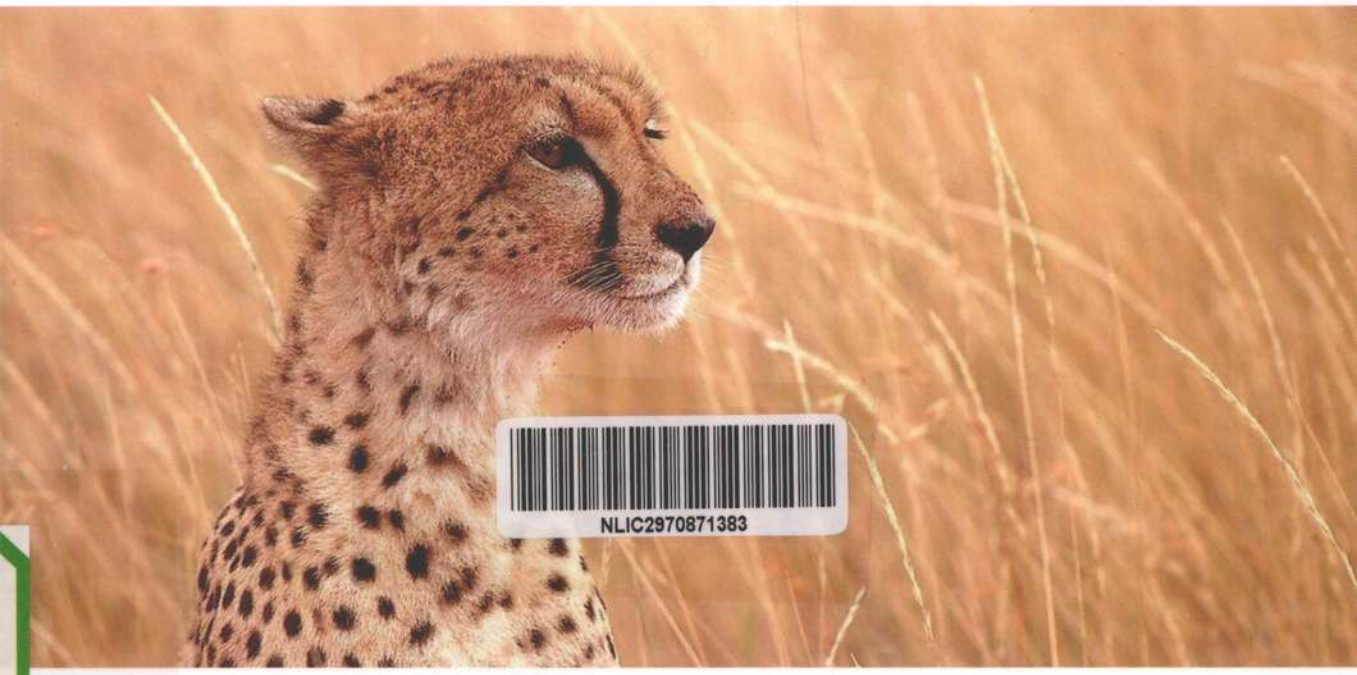
这是一本充满了各种程序代码实现方式的工具书籍
采用案例源码+解说形式全面介绍Apache CXF框架的功能

Broadview
www.broadview.com.cn

基于Apache CXF构建SOA应用

任钢 编著



NLIC2970871383

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

基于Apache CXF构建SOA应用

基于Java规范的开源Web Services框架，我最早接触的是Apache Axis，当时还编写了基于Axis框架的一个扩展框架。后来与一些公司的开发团队接触，才知道Apache还有一个开源Web Services框架，即Apache CXF。这是一个合并过来的产品。这样，国内开发人员又多了一个开源Web Services框架选择，而且这个开源框架整合了ESB和Web Services，所以我对Apache CXF未来在中国的发展前景还是充满着信心的。对于Java支持SOA，也有很长的时间了，但国内关于SOA的方案都是一些大企业的平台，有点阳春白雪的感觉。对于一些小企业，采用一种重量级的工具似乎有一些不堪重负或者得不偿失。而Apache CXF框架则是一个轻量级的开源Web Services框架。在这个框架上完全可以构筑一个企业级的SOA平台。正是在这个理念的基础上，我认真地学习了Apache CXF框架，并把在这个学习过程中的体会、经验和一些应用例子贡献给读者。

新浪微博 weibo.com
@博文视点Broadview
请访问www.broadview.com.cn/19460 下载相关资源。

上架建议：编程语言

ISBN 978-7-121-19460-3

策划编辑：张月萍
责任编辑：贾 莉
封面设计：李 玲

定价：76.00元

基于Apache CXF构建SOA应用

任钢 编著

宁波大学园区图书馆
藏书
NLIC

NLIC2970871383

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

Apache CXF 框架是一个比较有前途的开源 Web Services 框架，也是构建 SOA 架构应用的利器。本书采用案例源码和解说形式全面介绍 Apache CXF 框架的功能。

本书共 15 章，大致分为三个部分。第一部分介绍关于 SOA 和 Web Services 的初步知识，第二部分介绍 Apache CXF 框架的一些基础知识，第三部分重点介绍 Apache CXF 框架的应用，包括 Apache CXF 框架的前端（Frontends）应用、数据绑定（DataBindings）应用、传输协议（Transports）应用，并隆重推出了 Apache CXF 框架如何实现 RESTful 服务、如何支持动态语言和 WS-* 规范等，另外，还包括 Apache CXF 框架一些高级功能的用法。最后，本书还描述了 Apache CXF 的工具、配置、调试、日志、部署和发布等使用的相关内容。

本书最大的特点是实用性。对于 SOA 和 Web Services 的基本概念只是初步介绍，主要内容是基于 Apache CXF 框架的 Web Services 应用案例。对于每一个 Apache CXF 框架的功能主题，都通过一个或多个实际的案例场景来进行阐述。对于每一个案例场景，都有源代码程序例子、架构描绘和程序实现说明。笔者可以负责任地说每一个例子都经过调试并能够运行。实践也是编写本书的一个重要目的，最终目的就是让读者全方位地了解 Apache CXF 框架能实现的功能，一方面让读者理解开发者的思路，另一方面帮助读者在实际工作中应用这些方法和编程。

本书适用于软件设计师、软件开发工程师和一些正在进行 SOA 开发的开发人员，既可以作为 Apache CXF 框架的学习指南，也可以提供给软件开发工程师在设计方面进行参考。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

基于 Apache CXF 构建 SOA 应用 / 任钢编著. — 北京：电子工业出版社，2013.3
ISBN 978-7-121-19460-3

Ⅰ. ①基… Ⅱ. ①任… Ⅲ. ①互联网络—网络服务器 Ⅳ. ①TP368.5

中国版本图书馆 CIP 数据核字（2013）第 013726 号

策划编辑：张月萍
责任编辑：贾　莉
印　　刷：三河市鑫金马印装有限公司
装　　订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×1092　1/16　印张：32.75　字数：890 千字
印　　次：2013 年 3 月第 1 次印刷
印　　数：3000 册　定价：76.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件到 dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

软件架构的实现模式是一个发展的过程。从以前的面向过程、面向对象，到后来的面向构件、面向整合和面向集成，接着又进化到现在的面向服务模式。这时候，一个非常时髦的词——SOA就出现在我们的面前。

SOA是一个沉重的话题。我很早就接触了这个概念。那时既年轻也轻狂，觉得SOA无非就是那么几个已经耳熟能详的单词组合。SOA的确出现得很早，但具体落地非常艰难。仅有一些空洞的解决方案和让人发炫的理想场景。在这样高不可及的光芒下，我们只是空喊一些口号，创造着各种新鲜、时髦和美好的词汇与概念，可没有一个明确可以下手的地方。如何让SOA进入百姓家，这似乎成为了一个不可能完成的任务。

Web Services的出现，似乎给了我们黑的眼睛，让我们有了寻找光明的希望。同时，Java平台对Web Services的支持，也给了我们实现SOA的利器。

基于Java规范的开源Web Services框架，我最早接触的是Apache Axis，当时还编写了基于Axis框架的一个扩展框架。后来与一些公司的开发团队接触，才知道Apache还有一个开源Web Services框架，即Apache CXF。这是一个合并过来的产品。这样，国内开发人员又多了一个开源Web Services框架选择，而且这个开源框架整合了ESB和Web Services，所以我对Apache CXF未来在中国的发展前景还是充满着信心的。对于Java支持SOA，也有很长的时间了，但国内关于SOA的方案都是一些大企业的平台，有点阳春白雪的感觉。对于一些小企业，采用一种重量级的工具似乎有一些不堪重负或者得不偿失。而Apache CXF框架则是一个轻量级的开源Web Services框架，在这个框架上完全可以构筑一个企业级的SOA平台。正是在这个理念的基础上，我认真地学习了Apache CXF框架，并把在这个学习过程中的体会、经验和一些应用例子贡献给读者。

在本书的编写中，主要参考了Apache CXF官方网站的信息。毕竟这是关于Apache CXF最权威的官方信息来源。在参考材料中，我比较喜欢IBM的相关网站，IBM的技术网站一般都站在比较前沿的领域来讲解、讨论和分析问题，所以，书中有一些内容也是摘取或参考了IBM技术网站的思想和内容。

本书覆盖的内容较多，可以毫不夸张地说，书中的很多章节都可以独立地撰写出一本很厚的书籍。正是出于这样的考虑，笔者不能对一些技术做详细描述，有的内容也只是蜻蜓点水地简单说明一下。本书最大的特点是实用性。对于SOA和Web Services的概念，以及其中的方方面面的内容，都有很多相关的书籍来进行阐述。作为本书的基本概念，我也介绍了一些关于这方面的内容。对于每一个例子，都进行了不止一次的编写、调试和测试。我可以负责任地说每一个例子都是可以运行的。对于我每一个涉及的主题，最终都是通过一个实际的案例（包括源程序代码）

进行阐述的。这本书不是一本介绍理论的书，而是充满了各种程序代码实现方式的工具书籍。

当然，阅读本书也要具备一定的基础知识，否则，有些术语和解释还是比较难以理解的。

本书从第一次编写到最后实现出版总共花费了两年半的时间。在这个过程中我耗费了大量的时间和精力。并且这些工作都是在业余时间内完成，白天还要照常上班，只有到了晚上或者节假日才有闲暇时光。我一般难得有闲暇时间，如果有，也是打算去休息或放松，毕竟平时的工作还是比较劳累的。但我还是硬着皮头坐下来写书，写书是一件非常枯燥的任务。对于枯燥，这还不是最大的障碍。我认为最困难的是一个接着一个的技术难题。很多时候有些难题没有办法一下子解决，于是就做了一个例子又一个例子，编写了一个测试接着又一个测试，可还是不能达到自己理想的结果，沮丧、失败、自责、怀疑、困惑等等都涌上头来。在这段时间中我有几次都考虑放弃，心里总是在继续写和终止写之间徘徊不定。但总是觉得已经走了这么长的路程，不能因为一时的挫折而终止多年的辛劳，于是还是像一个孤独的苦行僧执着地编写和测试下去。很高兴的是我还是坚持下来了，在经历了无数个寂寞和孤单的夜晚，把这本书赶写出来。

在这期间，我要感谢我的家人对我的理解和支持。我的妻子和女儿总是抱怨我一天到晚总是待在电脑旁边。我要感谢我的父亲和母亲，我远离家乡，没有时间照顾他们，但他们总是支持我现在做的一切。在这期间，我的母亲永远地离开了我，我只能用这本书来纪念她。我要感谢我的好朋友江愿兵、徐宾和卢建平，他们在很多方面给了我无尽和无私的支持与鼓励。我把这里的一切都献给他们并祝愿他们好人一生平安。

由于笔者水平有限，书中覆盖的范围又比较广，涉及的概念也比较多，所以书中的错误和缺点在所难免，希望读者能给予批评和指正。我的联系方式是：rengang66@sina.com。

目　　录

第 1 章　Apache CXF 概述 1

1.1　Apache CXF 框架简介 1
1.2　Apache CXF 的基本特征 2
1.3　Apache CXF 的功能特性 2
1.3.1　支持众多标准 2
1.3.2　支持多种传输协议和协议绑定、数据绑定和数据格式 3
1.3.3　灵活部署 3
1.3.4　支持多种语言编程 3
1.3.5　支持的工具 3
1.4　Apache CXF 的历史 4

第 2 章　相关基础知识 5

2.1　SOA 基础知识 5
2.1.1　SOA 的定义、基本特征和优点 5
2.1.2　SOA 参考架构 7
2.1.3　SOA 相关技术标准 9
2.1.4　SOA 的设计原则 11
2.1.5　SOA 与 Web Services 的关系 12
2.2　Web Services 的相关规范 13
2.2.1　Web Services 简介 13
2.2.2　Web Services 架构及其 WS 规范简介 16
2.2.3　基本 Web Services 规范——WSDL、SOAP、UDDI 18
2.2.4　扩展的 WS 规范——WS-*规范 26
2.3　Java 中关于 SOA 的相关规范 30
2.3.1　JAX-RPC 规范 31
2.3.2　JAX-WS 规范 33
2.3.3　JAX-RS 规范 34
2.3.4　JAXB 规范 36

第 3 章　Apache CXF 开发环境介绍 38

3.1　Apache CXF 安装包的下载和说明 38
3.2　Apache CXF 框架支撑和运行环境 41
3.3　搭建 Apache CXF 开发环境 42

3.3.1 用 Ant 来创建项目 42
3.3.2 用 Maven 来创建项目 44
3.3.3 用 Eclipse 集成 Apache CXF 49

第 4 章 简单的 Apache CXF 例子 50

4.1 一个简单的 JAX-WS 服务程序 50
4.2 利用 Spring 创建 Web Services 服务程序 55
4.3 Servlet 容器中的 Web Services 实现 61

第 5 章 Apache CXF 的架构体系和基础 65

5.1 Apache CXF 的核心架构 65
5.2 Bus 介绍 67
5.3 消息（Messaging）和拦截器（Interceptors）组件介绍 70
5.4 前端编程模型（Frontend）组件介绍 73
5.5 服务模型（Service Model）组件说明 75
5.6 数据绑定（Data Bindings）组件 76
5.7 绑定（Bindings）组件 76
5.8 传输协议（Transport）组件 78
5.9 CXF 的注释 78
5.10 案例场景说明 82

第 6 章 CXF 的前端应用 84

6.1 CXF 的前端应用（Frontends）简介 84
6.2 基于代码优先（Java First）的 JAX-WS 前端模式实现 85
6.2.1 基于代码优先（Java First）的 Web Services 的步骤 85
6.2.2 基于代码优先（Java First）的 Web Services 的例子 95
6.3 基于 WSDL 优先（WSDL First）的 JAX-WS 前端模式实现 125
6.3.1 基于 WSDL 优先的 JAX-WS 前端模式实现的步骤 125
6.3.2 基于 WSDL 优先的 JAX-WS 前端模式实现的简单例子 130
6.3.3 基于 WSDL 优先的 JAX-WS 前端模式实现的复杂例子 145
6.4 简化前端模式（Simple Frontend） 145
6.4.1 简化前端模式（Simple Frontend）介绍 145
6.4.2 采用 Apache CXF 简化前端实现的例子 145
6.4.3 采用 Apache CXF 简化前端实现的 Servlet 例子 151
6.5 Provider/Dispatch 服务前端应用模式 154
6.5.1 Provider/Dispatch 服务前端应用模式介绍 154
6.5.2 采用 DOMSource（message）的 Provider/Dispatch 前端模式实现例子 162
6.5.3 采用 DOMSource（Payload）的 Provider/Dispatch 前端模式实现例子 168
6.5.4 采用 SOAPMessage 的 Provider/Dispatch 前端模式实现例子 170
6.6 采用 Apache CXF 的动态客户端技术 175

6.6.1 Apache CXF 的动态客户端技术介绍 175
6.6.2 Apache CXF 的动态客户端技术例子 178
6.6.3 Apache CXF 的动态客户端实现的 Servlet 例子 189

第 7 章 CXF 的数据绑定 194

7.1 数据绑定（DataBindings）介绍 194
7.2 JAXB 数据绑定 196
7.2.1 JAXB 介绍 196
7.2.2 Apache CXF 实现 JAXB 的方式 198
7.2.3 Apache CXF 实现 JAXB 数据绑定例子 199
7.3 Aegis 数据绑定 200
7.3.1 Aegis 介绍 200
7.3.2 采用简化前端、Aegis 数据绑定的例子实现 209
7.3.3 采用简化前端 Aegis 数据绑定的 Servlet 例子实现 213
7.4 MTOM 使用 217
7.4.1 MTOM 简介 217
7.4.2 CXF 实现 MTOM 的方式 218
7.4.3 CXF 实现 MTOM 的例子 221
7.4.4 CXF 实现 MTOM 的 Servlet 例子 229
7.5 XMLBeans 的使用 233
7.5.1 XMLBeans 简介 233
7.5.2 CXF 实现 XMLBeans 的方式 234
7.5.3 实现简化前端 XMLBeans 数据绑定的例子 235
7.5.4 采用简化前端 XMLBeans 数据绑定的 Servlet 例子实现 239

第 8 章 CXF 的传输 243

8.1 CXF 支持的传输协议 243
8.2 HTTP 传输协议 243
8.2.1 CXF 支持 HTTP 传输协议介绍 244
8.2.3 Spring 注入 HTTP 传输并基于 Servlet 的实现 247
8.3 JMS 传输协议 250
8.3.1 JMS 简介 250
8.3.2 在 Apache CXF 中使用 JMS 252
8.3.3 Spring 注入实现 JMS 的例子程序 258
8.3.4 Spring 注入实现 JMS 的 Servlet 例子程序 264
8.4 Local 传输协议 269
8.4.1 Apache CXF 的 Local 传输协议介绍 269
8.4.2 CXF 的 Local 配置和使用 269
8.4.3 实现简化前端 Local 传输的例子 271
8.4.4 实现 JAX-WS 规范并采用 Local 传输的例子 275

8.4.5 Spring 注入实现 JAX-WS 规范并采用 Local 传输的例子 280

第 9 章 CXF 的配置、调试和日志 285

9.1 CXF 的配置 285
9.1.1 CXF 配置概述 285
9.1.2 Bus 配置 288
9.1.3 Features 列表 290
9.1.4 JMX 管理 290
9.2 CXF 的日志管理 292
9.2.1 CXF 日志的设置 292
9.2.2 定义日志级别 294
9.2.3 使用 Log4J 日志方式 294
9.2.4 使用 SLF4J 日志方式 294
9.3 Apache CXF 的调试管理 295
9.3.1 Eclipse IDE 295
9.3.2 Tcpmon 295
9.3.3 WSMonitor 295
9.3.4 SOAP UI 295
9.3.5 Wireshark 295

第 10 章 CXF 的工具 296

10.1 Ant 工具（2.0.x 和 2.1.x） 296
10.2 在 Eclipse 的 CXF 工具 297
10.3 Java 代码生成 Web Services 297
10.4 Java 代码生成 WSDL 299
10.5 WSDL 生成 Java 代码 300
10.6 WSDL 转化为 Javascript 302
10.7 WSDL 生成服务（Service） 303
10.8 WSDL 生成 SOAP 305
10.9 WSDL 生成 XML 306
10.10 WSDL 验证器 307
10.11 XSD 生成 WSDL 307

第 11 章 CXF 实现 RESTful 服务 309

11.1 RESTful 服务介绍 309
11.1.1 RESTful 服务概述 309
11.1.2 RESTful 原则 310
11.1.3 创建基于 REST 的 Web Services 314
11.2 Apache CXF 的 RESTful 实现方式 315
11.2.1 JAX-RS 实现方式 315

11.2.2 基本特征 316
11.2.3 支持的特征 317
11.2.4 其他先进功能 320
11.3 JAX-WS Provider 和 Dispatch 实现方式 321
11.4 HTTP 绑定方式 323
11.5 CXF 实现 RESTful 服务的例子说明 326
11.5.1 CXF 采用 HttpClient 实现基本的 RESTful 应用 326
11.5.2 CXF 采用 HttpClient 在 Servlet 实现基本的 RESTful 应用 336
11.5.3 CXF 采用 WebClient 实现 RESTful 应用 340
11.5.4 CXF 采用 WebClient 在 Servlet 实现基本的 RESTful 应用 348
11.5.5 JAX-WS Provider 和 Dispatch 实现 RESTful 方式 352
11.5.6 Http_Binding 实现基于 XML 的 RESTful 方式 359
11.5.7 Http_Binding 在 Servlet 实现基于 XML 的 RESTful 方式 365
11.5.8 Http_Binding 实现基于 JSON 的 RESTful 方式 369
11.5.9 Http_Binding 在 Servlet 实现基于 JSON 的 RESTful 方式 374

第 12 章 CXF 对动态语言的支持 379

12.1 CXF 对 JavaScript 等语言的支持 379
12.1.1 用 JavaScript 来实现 Web Services 379
12.1.2 用 E4X（ECMAScript for XML）来实现 Web Services 380
12.1.3 部署 Script 服务 381
12.2 CXF 基于 JavaScript 等语言实现 Web Services 的例子 382
12.2.1 用 JavaScript 调用 CXF 的 Web Services 382

第 13 章 CXF 对 WS-* 的支持 391

13.1 Apache CXF 支持 WS-Addressing 391
13.1.1 WS-Addressing 简介 391
13.1.2 Apache CXF 的 WS-Addressing 配置 393
13.1.3 Apache CXF 的 WS-Addressing 的实现例子 396
13.2 Apache CXF 支持 WS-Policy 405
13.2.1 WS-Policy 简介 405
13.2.2 Apache CXF 使用 WS-Policy 框架 406
13.2.3 Apache CXF 的 WS-Policy 的实现例子 407
13.3 Apache CXF 支持 WS-ReliableMessaging 412
13.3.1 WS-ReliableMessaging 简介 413
13.3.2 Apache CXF 使用 WS-ReliableMessaging 的配置 414
13.3.3 Apache CXF 的 WS-ReliableMessaging 的实现例子 417
13.4 Apache CXF 支持 WS-Security 429
13.4.1 WS-Security 介绍 429
13.4.2 Apache CXF 使用 WS-Security 的配置 431

13.4.3 Apache CXF 的 WS-Security 的实现例子 437
13.5 Apache CXF 支持 WS-SecurityPolicy 450
13.5.1 WS-SecurityPolicy 简介 450
13.5.2 Apache CXF 使用 WS-SecurityPolicy 的配置 451
13.5.3 Apache CXF 的 WS-SecurityPolicy 的实现例子 453
13.6 Apache CXF 支持 WS-Trust 465
13.6.1 WS-Trust 简介 466
13.6.2 Apache CXF 使用 WS-Trust 的配置 467
13.7 Apache CXF 支持 WS-SecureConversation 470
13.7.1 WS-SecureConversation 介绍 470
13.7.2 Apache CXF 使用 WS-SecureConversation 的配置 471

第 14 章 CXF 的高级功能 472

14.1 CXF 的 Feature 功能 472
14.1.1 CXF 的 Feature 功能说明 472
14.1.2 编写和配置 CXF 的 Feature 473
14.1.3 CXF 的 Feature 列表 475
14.1.4 CXF 实现 Feature 的例子 476
14.2 CXF 的拦截器（Interceptors）和相位器（Phases） 480
14.2.1 CXF 的拦截器（Interceptors）和相位器（Phases）介绍和使用 480
14.2.2 CXF 的拦截器（Interceptors）的例子 488
14.3 CXF 的代理（invoker） 495
14.3.1 CXF 的代理（invoker）功能说明 495
14.3.2 CXF 的代理（invoker）的实现例子 496
14.4 CXF 的 MER（Multiplexed EndpointReferences） 503
14.5 CXF 的基础服务 505
14.6 CXF 的服务路由（Service Routing） 506

第 15 章 CXF 的部署和发布 510

15.1 应用服务器的具体配置指南 510
15.1.1 Tomcat 510
15.1.2 JBoss 510
15.1.3 WebLogic 511
15.1.4 WebSphere 512
15.1.5 OC4J 512
15.2 在 Spring 内嵌入 CXF 512

参考文献 514

第 1 章 Apache CXF 概述

本章内容：

1. 简介 Apache CXF。
2. Apache CXF 基本特征。
3. Apache CXF 功能特性。
4. Apache CXF 历史。

1.1 Apache CXF 框架简介

Apache CXF 框架是一个开源的 Web Services 框架。它来源于两个著名的开源项目——ObjectWeb Celtix 和 Codehaus XFire。其中，ObjectWeb Celtix 是一款著名的开源 ESB 产品，Codehaus XFire 则是闻名遐迩的 SOAP 堆栈软件。这两个开源项目在 2006 年合并后诞生了 Apache CXF 开源项目。强强联手之后生成的 Apache CXF 自然集两者的长处于一身。Apache CXF 的诞生给开源 SOA 工具领域带来很大的冲击。Apache CXF 是开源社区中继 JAX-WS R1 之后第一个通过 JAX-WS TCK 的 SOAP 堆栈，同时它还支持众多的标准、数据绑定、传输协议。

Apache CXF 提供了对 JAX-WS 规范的全面支持，一方面提供了对多种绑定（Binding）、数据绑定（DataBinding）、传输协议（Transport）以及数据格式（Format）的支持，另一方面，可以根据实际项目的需要，采用代码优先（Code First）或者 WSDL 文档优先（WSDL First）来轻松地实现 Web Services 的发布和使用。Apache CXF 利用前端编程（Frontend API）模式来构建和开发 Services ，这与 JAX-WS 类似。这些 Services 可以支持多种协议，如 SOAP、XML/HTTP、RESTful HTTP 或者 CORBA，并且可以在多种传输协议上运行，如 HTTP、JMS 或者 JBI。CXF 大大简化了 Services 的创建，同时它继承了 XFire 的传统，一样可以无缝地与 Spring 进行集成和整合。

选择 Apache CXF 框架的理由很多，主要体现在它良好的特性上。这些特性主要表现在：①简单易用，支持众多编程模型；②与 Spring 紧密集成；③提供了对 RESTful 服务的支持；④可插拔的框架结构设计，可自适应地支持其他多种数据绑定、协议标准和遗留系统；⑤具有轻量级 Web Services 框架；⑥友好的商业 Apache 许可证。

目前，Apache CXF 有很好的用户基础，这主要是因为 Apache CXF 传承了 XFire 和 Spring 的优良“基因”，这两个平台的使用者可继续沿用以前的知识。加上对 JAX-WS、数据绑定、多传输协议、多数据格式和 RESTful 的支持，Apache CXF 框架的未来一直都被看好，关注其进展是非常有意义的。

1.2 Apache CXF 的基本特征

Apache CXF 包含一个范围广泛、功能齐全的功能集合，该功能集合主要集中在以下几个方面：

- **Web Services 标准（WS-*规范）的支持**：Apache CXF 支持多种 Web Services 标准，包括 SOAP（SOAP1.1 和 SOAP1.2）规范、WSI Basic Profile、WSDL、WS-Addressing、WS-Policy、WS-ReliableMessaging、WS-Security、WS-SecurityPolicy 以及 WS-SecureConversation 等。而且现阶段对 Web Services 标准的支持还在继续扩大和完善中。
- **前端模式**：Apache CXF 支持多种前端编程模型。Apache CXF 实现了 JAX-WS 中的 API（TCK 的标准）。它还支持一种“简单前端编程模型”，即可以无注释创造客户和 SEI 终端。Apache CXF 同时支持 WSDL 优先和编码优先两种实现模式。针对 RESTful 规范，Apache CXF 还提供了对 JAX-RS（TCK 的标准）前端的支持。
- **易用性**：Apache CXF 设计明确，编码直观，易于使用。用简单的 API 调用和组合就能快速构建出编码优先的 Web Services，Maven 插件的引入使得集成工具更加容易，对 JAX-WS 的 API 支持、Spring 2.x 的 XML 支持，使配置变得更轻而易举，等等。
- **二进制和遗留协议支持**：Apache CXF 的设计采用了可插拔架构，故可以提供整合任何传输类型的架构容器，不仅支持 XML 类型的容器，也支持非 XML 类型的绑定，如 JSON 和 CORBA 等。

1.3 Apache CXF 的功能特性

Apache CXF 是一个开放源码的 Web 服务框架。该框架提供了一个基于 Web Services 标准并易于使用的编程模型。Web Services 可以使用多种不同的应用协议，诸如针对 HTTP 的应用协议就包含 SOAP、XML 和 JSON、REST 风格等。Apache CXF 框架也支持 JMS（Java 消息服务）的各种传输协议。

1.3.1 支持众多标准

1. 支持 JSR 的相关规范和标准

JAX-WS——Java API for XML-Based Web Services (JAX-WS) 2.0- JSR-224

Web Services Metadata for the Java Platform——JSR-181

JAX-RS——The Java API for RESTful Web Services - JSR-311

SAAJ——SOAP with Attachments API for Java (SAAJ) - JSR-67

2. 支持 WS-*及其相关规范

- 基本规范支持：WS-I Basic Profile 1.1。
- 服务质量规范：Web 服务可靠消息传输（WS-Reliable Messaging，WS-RM）。
- 元数据规范：WS-Policy、WSDL 1.1。
- 通信安全规范：WS-Security、WS-SecurityPolicy、WS-SecureConversation、WS-Trust（部

分支持）。

- 消息支持规范：WS-Addressing、SOAP 1.1、SOAP 1.2、MTOM（Message Transmission Optimization Mechanism）。

1.3.2　支持多种传输协议和协议绑定、数据绑定和数据格式

- 协议绑定：SOAP、REST/HTTP、纯 XML。
- 数据绑定：JAXB 2.x、Aegis、Apache XMLBeans、SDO（Service Data Objects）、JiBX（部分支持）。
- 格式：XML、JSON、FastInfoset。
- 传输协议：HTTP、Servlet、JMS 和 Local（即 JVM 内部消息通信机制），还有其他通过 Apache CXF 的 Camel 传输协议，如 SMTP/POP3、TCP 和 Jabber 等。

Apache CXF 可扩展的 API 允许开发者方便地对绑定和消息格式进行扩展，如 CORBA/IIOP 等格式支持。

1.3.3　灵活部署

- 轻量级容器：Jetty、Tomcat 或基于 Spring 的容器。
- JBI 集成：发布成为 JBI 容器中的服务引擎，这些 JBI 容器有 ServiceMix、OpenESB 或 Petals 等。
- Java EE 集成：可部署在 Java EE 应用服务器中，如 Tomcat、JBoss、Apache Geronimo、JOnAS、RedHat JBoss、OC4J、Oracle WebLogic 和 IBM WebSphere。
- 单独运行的客户机/服务器。

1.3.4　支持多种语言编程

- 完全支持 JAX-WS 2.X 客户机/服务器编程模型。
- JAX-WS 2.X 的同步、异步和单程 API。
- JAX-WS 2.X 动态调用接口（Dll）API。
- JAX-RS 的 RESTful 客户端。
- 支持包装（wrapped）和非包装（non-wrapped）风格。
- XML 消息传递 API。
- 客户端和服务器编程都支持使用 JavaScript 和 ECMAScript 4 XML(E4X)。
- 通过 Yoko 提供对 CORBA 的支持。
- 通过 ServiceMix 来支持 JBI。

1.3.5　支持的工具

- 代码生成工具：WSDL 到 Java，WSDL 到 JavaScript，Java 到 JavaScript。
- WSDL 生成工具：Java 到 WSDL，XSD 到 WSDL，IDL 到 WSDL，WSDL 到 XML。
- 增加服务端点：WSDL 到 IDL。

- 验证文件：WSDL 验证。

1.4 Apache CXF 的历史

Apache CXF 是两个著名的开源项目 ObjectWeb Celtix 和 Codehaus XFire 于 2006 年合并后的产物，Celtix 是一个开放源码并基于 Java 的企业服务总线（ESB）的项目，由 ObjectWeb 财团提供开放源码的中间件解决方案的产品。XFire 是一个基于 Java 的 SOAP 框架，是一个 Codehaus 项目的开放源码。Apache CXF 的名称来自于 Celtix 和 XFire 的组合，因此称为 Apache CXF 的产品。Apache CXF 集 ObjectWeb Celtix 和 Codehaus XFire 的长处于一身，给开源 SOA 工具领域带来不小的冲击。

官方网站：http://cxf.apache.org/index.html。

第2章 相关基础知识

本章内容：

1. SOA 基础知识。
2. Web Services 的相关规范。
3. Java 中关于 SOA 的相关规范。

2.1 SOA 基础知识

SOA 基础知识包括 SOA 的定义、基本特征、优点、参考架构、相关技术标准、设计原则等，也包括 SOA 和 Web Services 的关系。

2.1.1 SOA 的定义、基本特征和优点

SOA 有多种定义，这是一个没有正确答案的理论问题。但是其基本特征和优点基本上还是一致。

1. SOA 的定义

一般而言，SOA（Service-Oriented Architecture，面向服务架构）是一种架构模型。SOA 可以根据需求，通过网络对松散耦合的粗粒度应用组件进行分布式部署、组合和使用。服务层是 SOA 的基石。服务可以直接被应用调用，这样 SOA 有效控制了系统中与软件代理交互的人为依赖性。关于 SOA 的定义存在着多种说法。下面是一些关于 SOA 比较典型的定义说明。

SOA 的关键是“服务”的概念，W3C 将服务定义为：“服务提供者完成一组工作，为服务使用者交付所需的最终结果。最终结果通常会使使用者的状态发生变化，但也可能使提供者的状态改变，或者双方都产生变化”。

Service-architecture.com 将 SOA 定义为：“本质上是服务的集合。服务间彼此通信，这种通信可能是简单的数据传送，也可能是两个或更多的服务协调进行某些活动。服务间需要某些方法进行连接。所谓服务就是精确定义、封装完善、独立于其他服务所处环境和状态的函数。”Looselycoupled.com 将 SOA 定义为：“按需连接资源的系统。在 SOA 中，资源被作为可通过标准方式访问的独立服务，提供给网络中的其他成员。与传统的系统结构相比，SOA 规定了资源间更为灵活的松散耦合关系。”

Gartner 则将 SOA 描述为：“客户端/服务器的软件设计方法，一项应用由软件服务和软件服务使用者组成……SOA 与大多数通用的客户端/服务器模型的不同之处在于，它着重强调软件组

件的松散耦合，并使用独立的标准接口。”

IBM 定义：面向服务的体系结构是一个组件模型，它将应用程序的不同功能单元称为服务。通过这些服务之间定义良好的接口和契约联系起来。接口是采用中立的方式进行定义的，它应该独立于实现服务的硬件平台、操作系统和编程语言。这使得构建在各种各样的系统中的服务可以以统一和通用的方式进行交互。

根据以上定义，我们可以初步对面向服务的体系结构有所理解，一般认为：面向服务体系（SOA）结构是一种软件工程方法学，同时也是一种新的软件开发范型。SOA 以松散耦合的粗粒度服务为基本单元，通过开放的、标准的并独立于硬件平台、操作系统和编程语言的接口来发布服务，并通过开放式和规范性的通信协议进行通信。SOA 是一种支持多种服务方式组合、构成和跨平台、并具有一定适应性和扩展性的软件系统。

2. SOA 的基本特征

SOA 具有几个鲜明的基本特征。这些特征表示如下：

- 标准化、规范化、开放性的服务接口；
- 可复用、可重用的服务；
- 以粗粒度为主，同时结合细粒度的服务接口；
- 分级分层访问模式，并可满足安全性要求；
- 松散耦合的组合模式；
- 使用的时间性，随时可用；
- 服务接口设计管理；
- 支持各种消息模式；
- 精确定义的服务契约；
- 通过外部访问来调用 SOA。

3. SOA 的优点

了解了 SOA 的定义和基本特征，下面介绍 SOA 的优点。

（1）灵活性、开放性和可扩展性

组件化和模块化的低层服务可以采用不同组合方式创建高层服务。也就是细粒度的小服务或中服务通过集成、合并和整合来构造粗粒度的大服务，从而实现服务的复用，这些都体现了 SOA 的灵活性和开放性。同时，由于服务消费者通过服务中介来访问服务生产者，这种服务的实现方式本身也体现了 SOA 的可扩展性。

（2）明确开发人员角色

所处理的服务可以区分开发人员角色。服务之间通过服务接口或者服务描述来进行访问和调用。不同的开发人员通过服务契约进行协同，实现服务功能。服务用于解决应用间的互操作问题以及用于组合新应用或应用系统。例如，熟悉 ESB 的开发人员可以集中精力在重用访问层，协调层开发人员则无须特别了解 ESB 的实现，而可将精力放在解决核心业务问题上。

（3）支持多种访问类型

借助对 XML、Web Services 标准的支持和准确的服务接口定义，SOA 可以支持多种访问类型，包括智能手机、智能终端等新型访问渠道。

（4）更易维护

服务提供者和服务使用者是一种松散耦合关系，两者通过开放标准、规范和协议来进行交互，这些要求实现了两者之间的便捷容易的维护特性。

（5）更好的伸缩性

依靠基于服务的设计、开发和部署的架构模型，SOA 可以非常轻松并且便捷地实现伸缩性。服务提供者可以彼此独立调整，以满足服务需求。

（6）更高的可用性

该特性在服务提供者和服务使用者的松散耦合关系上得以体现。服务的使用者无须了解服务提供者的实现细节，这样服务提供者就可以在 SOA 集群环境中灵活部署，服务使用者可以被转接到可用的服务例程上。

SOA 可以认为是 B/S 模型、XML/Web Service 技术之后的正常延伸。SOA 将能够帮助设计师站在一个新的高度理解企业级架构中的各种组件的开发、部署形式。SOA 将帮助企业系统架构师以更迅速、更可靠、更具重用性架构来搭建并重构整个企业业务系统。较之以往，以 SOA 架构的系统能够更加从容淡定地面对和处理企业业务的急剧变化。

2.1.2　SOA 参考架构

SOA 具有中立的接口定义（没有强制性地绑定到特定的实现上）的特性，该特征称为服务之间的松耦合。松耦合系统的优势有两个方面，首先是它的灵活性；其次，当组成整个应用程序的每个服务的内部结构和实现逐渐地发生改变时，作为整体的服务还能够继续存在。而与此相对，紧耦合意味着应用程序的不同组件之间的接口与其功能和结构是结合得相当紧密的，因而当需要对部分或整个应用程序进行某种形式的更改时，这种改变是不可能实现的或者实现的方式非常困难，这就说明它们显得非常脆弱。

SOA 参考架构可以从业务角度了解一下。不同厂商及组织定义的 SOA 参考架构，基本上大同小异，如图 2-1 所示。SOA 参考架构本身体现了松散耦合特性，SOA 参考架构需要考虑各种应用场景，可灵活地分拆组合使用。

SOA 参考架构是一个完整的企业架构，可以覆盖整个企业范围内集成的需求。故 SOA 参考架构主要应用在企业应用集成领域，该参考架构能够以服务的方式共享和复用企业现有应用和信息资产，保护用户以前的 IT 投资，并能够以服务方式构建新的业务流程和服务对象，对组织中的业务流程进行灵活的重构和优化，增强业务的适应性和敏捷性。SOA 参考架构中的服务通过模块化的方式进行集成。下面从业务的角度，简要介绍一下 SOA 参考架构各个部分的作用与定位。

在图 2-1 中，左侧部分表示 SOA 的建模和组装，中间部分表示 SOA 的部署，右边部分表示 SOA 的自治和管理。中枢部分是企业服务总线（Enterprise Service Bus），在服务之间提供连通性支持。

本参考架构描述了企业范围内 SOA 方案所需要的关键能力。下面分别介绍三个组成部分：

1. SOA 的建模和组装

工具是集成架构的基本组件，SOA 参考架构则提供了开发服务和业务创新优化服务。开发服务用于实现新开发的组件以及重用基础架构的能力；业务创新优化服务用于从 IT 和业务两个层

面监控和管理运行情况。

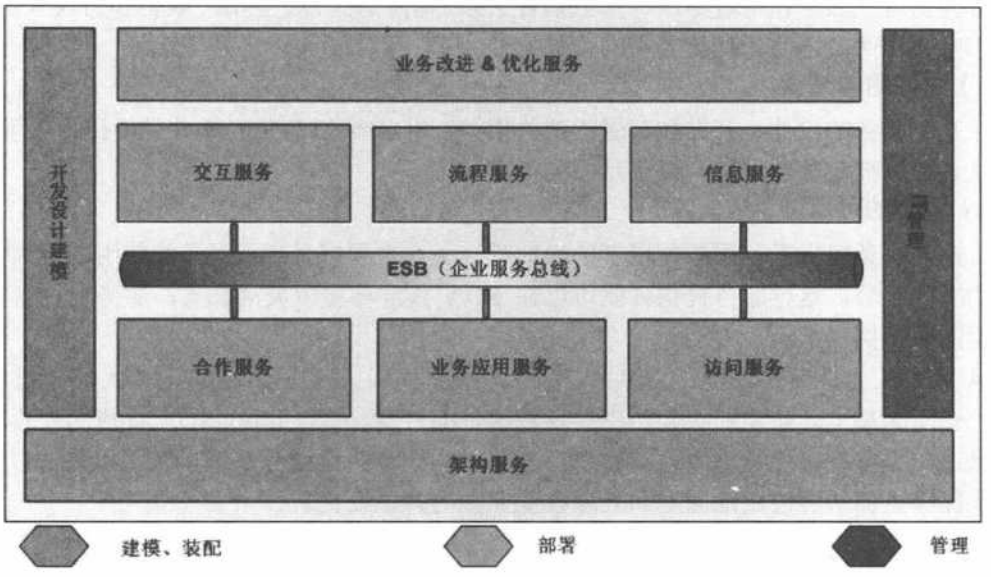


图 2-1　SOA 参考架构

2. SOA 的部署

SOA 的部署包括交互服务、流程服务、信息服务、应用服务等，通过企业服务总线来进行融会贯通。

（1）企业服务总线是 SOA 参考架构的核心。它为整个架构范围内所有服务提供相互通信的能力。其中传输服务、事件服务以及中介服务都是通过 ESB 来提供的。ESB 为服务之间的协调中介，可将组织中的所有服务，如适配器服务、信息服务、安全服务、业务服务、流程服务等，集中管理起来，暴露服务接口，并提供增值的服务注册、路由查找、服务质量保障等功能，它在 SOA 参考架构中起着核心中枢的作用。

（2）交互服务是直接面对用户的一层服务，将 IT 的功能和数据交付给最终用户，并能为用户提供友好的交互界面及门户集成的功能，满足用户特定的使用习惯。SOA 实施的效果，往往通过交互服务的易用性来实现。

（3）流程服务提供服务控制能力，将多个服务串起来实现一个业务流程。将组织中的服务进行组合编排，以满足业务流程的需要，同时，服务的组合也能够构建更大粒度的服务，形成新的业务服务。流程服务对于业务敏捷性及满足用户复杂的业务流程起着关键的作用。

（4）信息服务通过联合、复制和转换来解决基于不同实现方式的不同数据源之间的数据共享难题。SOA 解决方案中的很多服务都是由已有应用提供的，访问服务提供已有应用、打包应用程序与 ESB 之间的桥接能力，使得已有应用的功能以服务的形式对外暴露出来。在业务流程需要与外部的合作伙伴、供应商交互的情况下，伙伴服务提供一组文档、协议以及伙伴管理的能力。

（5）应用服务为新的应用组件提供运行时服务。

（6）适配器服务。适配器服务主要实现底层的适配工作，将组织现有应用系统进行封装，形成可复用的服务，来复用组织中的现有应用资产。所以，适配器服务是应用集成的基础。

3. SOA 的自治和管理

作为所有能力的基础，基础服务用于优化通过率、性能和可靠性。IT 服务管理服务包括对服务、应用和资源的管理和保护能力，如通过负载均衡来有效地分配系统计算资源。

因此 SOA 的实现可以从一个小的项目启动，在新的项目实施时，新的功能能够轻松地加到架构中，通过渐进的方式在企业范围内扩大集成的范围。

2.1.3　SOA 相关技术标准

根据 SOA 参考架构所对应的业务及构建 SOA 涉及的技术要求，需要对应的技术标准来统一不同厂家和用户的认识，构建开放的协作契约。根据技术标准在 SOA 中的角色功能，可以将 SOA 技术标准规范分为 3 大类：服务层次上的信息交互规范、基础通信标准规范、元数据标准规范。根据各种标准规范在 SOA 体系中的角色功能，可以将 SOA 协议栈分为 6 层，如图 2-2 所示。从底向上，包括传输层、消息层、描述层、管理层、服务组合层、表示层。下面简单介绍相关的技术标准及其应用场合。

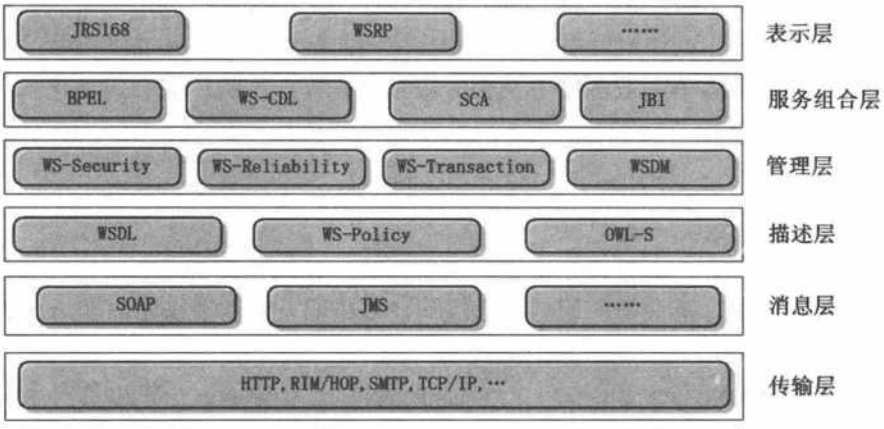


图 2-2　SOA 协议栈分层结构

（1）传输层

SOA 的传输协议包括 HTTP、RMI/IIOP（分别用于 Java 的远程对象调用和 CORBA 的远程通信）、SMTP、TCP/IP 等传统的传输协议。HTTP 协议是基于 Web 的服务最常用的协议，主要是由于 HTTP 协议便于在互联网上穿越防火墙。

（2）消息层

在消息层上，SOAP 已经是 Web Services 消息传输载体的事实上的首选协议。SOAP 协议采用 XML 描述，故语言独立、格式简单，便于理解、发布、解析和扩展。Web Services 协议栈中的许多规范都是基于 SOAP 协议进行扩展的，如 WS-Security 规范、WS-Reliability 规范、WS-Policy 规范等。在 Java EE 技术的应用中还可以选择 JMS 作为传输协议。当然，开发者还可以使用自定义或是某些厂商定义的传输协议。

（3）描述层

虽然 SOA 中的服务并不一定是 Web Services，但由于 Web Services 的语言独立性、协议的简

单性及众多厂商的支持，Web Services 已经是最常用的服务表现形式，成为了事实上的 SOA 标准 WSDL，即 Web Service Description Language（由 W3C 组织提出）。实际上，这也是一个服务语义描述。WSDL 从句法层面对 Web Services 的功能进行描述，包括 4 个不同的粒度：数据类型（Data type）、消息（Message）、方法（Operation）和访问端口（PortType）。由于此协议比较简单、语言独立，所以得到了绝大多数厂商的支持，如 IBM、Oracle 及国内中间件厂商等公司的产品，都提供了对 Web Services 及其标准的支持，而 WSDL 和 SOAP 已经成为 Web Services 最基础的标准。

（4）管理层

松耦合的分布式的 SOA 软件系统，有更多的处理服务质量方面的管理问题。例如，安全性问题，对于跨网络的多个服务访问，如何验证合法身份，如何防止饱和性攻击，消息内容如何防止篡改和窃取，多个信任域如何一次性验证身份等。为此，一些标准化组织，如 OASIS 等，讨论并推出一系列的标准，如 WS-Security 标准系列、WS-Trust 等。这些构建在 SOA 上的规范再结合传统的安全标准，如 KPI、加密算法等，在一定程度上可以用来解决 Web Services 的安全性问题；WS-Reliability、WS-Reliable Messaging 标准，在一定范围内可以用来解决消息传输的可靠性问题；WS-Transaction 系列协议，有条件地解决传统的多个 Web Services 消费的事务完整性问题。对于开发 SOA 服务容器、业务服务及企业服务总线等产品的厂商和研发机构，可以考虑并使用上述协议。目前上述部分标准和规范已经应用到某些商用产品中，如应用服务器、企业服务总线产品等。

（5）服务组合层

服务组合层的标准规范，包括两个方面的内容，第一是构建基础服务，第二是采用基础服务来进行组合、拼装、编排，形成新的服务。通过构建基础服务和将服务进行组合编排来满足用户的业务服务需要。下面列出几个稍微典型的规范。

WS-BPEL（即 Web Service Business Process Execution Language，Web 服务业务流程执行语言），是一种可执行语言，能够与各种促使业务流程自动化的软件系统相兼容。Web 服务编制，通过说明性的方式（而不是编程的方式）表达了进行 Web Services 合成的需求。此标准主要用于组织内部的业务流程管理及服务编排，目前越来越多的 BPM 产品基于此规范实现，大有代替传统工作流技术的趋势。

WS-CDL（Web Services Choreography Definition Language，Web 服务编排定义语言），定义为在多个交易服务之间建立一种形式化关系，WS-CDL 对集成的要求比较低，对于所有被集成的端点并不强求都要有独立的 Web Services 基础设施。此规范更多地用于服务组织之外的服务与流程编排。

SCA（Service Component Architecture，服务组件架构），提供了一种编程模型，可以支持基于 SOA 的应用程序实现。SCA 可以支持实现服务组件的各种技术，连接服务组件的各种存取方法。SDO（Service Data Object，服务数据对象）的产生就是为了统一和简化应用程序处理数据的方式，使用 SDO，应用编程人员可以用一致的方法操作异构数据源，包括关系型数据库、XML 数据源、Web Services 和企业信息系统。在 SOA 理想实现上，可以使用多种技术开发服务组件，通过使用 SCA 标准来组装组件及组装更大粒度的服务，而 SDO 可以作为服务及组件之间传输的

标准数据格式。

JBI（Java Business Integration，Java 业务集成），是基于 Java 语言的服务定义。JBI 定义了一个标准的体系结构，该体系结构允许第三方的组件插入到标准的基础设施上，并且即使这些组件是由不同提供商提供的，它们也可以一种可预见的和可靠的方式互操作。JBI 标准可用来实现企业服务总线及企业应用集成产品，但 JBI 仅限于 Java 技术，有很多开源的产品。

（6）表示层

表示层的标准如 JSR168 和 WSRP，主要应用在 Portal 应用系统的开发。

JSR168 遵循 Java 规范，它为创建 Portal 应用系统中的 Portlet 而建立标准的 API，JSR168 的设计目的是为实现 Portlet、基于 Java 的门户服务器和其他 Web 应用程序之间的互操作性。在开发基于 Java 技术的门户产品和 Web 页面集成时，一般情况下都考虑使用 JSR168 规范。而且，大多数的基于 Java 技术的 Portal 产品，都支持 JSR168 规范。

WSRP（Web Services for Remote Portlets，远程 Portlets 的 Web 服务），定义了如何利用基于 SOAP 规范的 Web Services 在门户应用程序中生成标记片断的规范。通过定义一组公共接口，WSRP 允许门户在它们的页面中内嵌远程运行的 Portlet，内嵌的页面并不需要门户开发人员进行任何编程。WSRP 规范是由 OASIS 组织制定，目前已得到多数厂商的支持，鉴于 WSRP 基于 Web Services 标准，而且技术相对独立，现在越来越多的 Portal 产品都支持此标准。

2.1.4　SOA 的设计原则

在软件开发的历史当中，各种设计原则也随着设计理念的不断进步，向着更优的方向发展。从 OO（Object Oriented）到 EJB（Enterprise JavaBean）再到 SOA，开发复用从面向对象、面向组件再到面向服务不断提升。一个基于 SOA 的系统由服务、发现和使用服务的应用，以及连接服务和应用的 SOA 基础设施组成。可见，对应用系统设计人员来说，最重要和最基本的概念就是服务。SOA 的设计原则和准则制定了在进行业务服务与技术服务定义时所应遵循的原则。SOA 在设计原则上，重点关心的是业务与 IT 对齐，保持灵活性、松散耦合等方面。一般来讲，SOA 包括以下设计原则。

- **一致性：**世界上有很多种实现技术可以用于创建、发布、发现和调用服务。如果 SOA 的参与者（包括服务提供者和服务的消费者）各自采用不同的技术，彼此之间又不能互相解析，必然会在开发、集成、维护等方面带来很大的问题。SOA 应该保证在所有参与者之间实现一致性，用所有参与者都能理解并能解析的方式，把整合的内容统一管理，以便减少由于这些之间的不统一而产生的开发、集成和维护等方面的工作。
- **服务粒度：**服务粒度主要是描述服务本身的功能大小或者是服务应该包含的操作数量。服务粒度的划分要考虑的因素很多，主要包括可用性、可维护性、可操作性、可扩展性和易用性等。现阶段还没有一种非常合理通用的方法、原理或模型来确定服务粒度的大小。如果粒度过粗，单个服务中必然会包含大量的操作，从而包含很多客户需要的功能，这样此服务可能会有很多的使用者，容易陷入一种“大杂烩”误区；如果粒度过细，则会导致服务总数量的急剧增长，这样管理起来非常复杂，同时还是没有达到简化服务的目的。服务粒度本身就是一个辩证的组合体，既不能太大，也不能太小，其要求是适合。

每个服务提供很少的几个方法（粒度过细），或很多操作集中在单个服务中（粒度过粗），都将对系统性能带来不好的影响。因此，服务的粒度需要精心选择。一般情况下，作为业务服务的粒度，主要还是要对业务进行核心归纳和抽象，这样才能把握好服务粒度的选择分寸。

- **自治**：所有的服务都是可以自我管理的，也就是自治的，每一个服务都独立地进行部署、版本控制和管理，单一服务的改变不应该影响到整个系统。
- **高内聚**：服务应该是高内聚的。所谓高内聚，就是服务内的功能模块都是在业务上紧密相关的，这些服务都是为了一致的主题进行合并和组装的，不相关的功能不应该放到同一个服务中。
- **松耦合**：服务之间要求最小的依赖性。所谓松耦合，就是一个服务的改变并不会影响到其他服务的变化，Web Services 要求它们之间能够相互知晓，而 SOA 可以通过使用服务总线，进一步解耦服务使用者和服务提供者，实现服务之间的透明访问。
- **服务调用**：服务平台应该建立在平台中立的技术之上，同时提供对多种可传输协议和数据格式的支持。

采用 SOA 原则进行系统分析设计，没有强制地要求必须遵循哪些原则。作为一般原则，上述内容可以作为一种参考。对 SOA 的设计原则的理解，也在随着人们对 SOA 的认识和实践过程中不断升华，不断发展变化着。

2.1.5 SOA 与 Web Services 的关系

实现 SOA 的最常见技术标准是 Web Services。不过，这并不是说明开发 SOA 的各个部分的唯一技术是 Web Services。实际上是大部分 SOA 应用都涉及集成遗留系统和旧数据，此类数据包含在使用 MQSeries 和 Common Object Request Broker Architecture （CORBA） 等技术的系统中。其中许多技术都已经针对 SOA 的规范来进行了调整，不管有无 Web Services 包装均可使用。事实上，可以仅使用 MQSeries、CORBA 甚至远程过程调用（Remote Procedure Call，RPC）技术来实现 SOA。但不得不承认，Web Services 正迅速成为用于支持 SOA 应用的事实标准。

现阶段而言，Web Services 是现在最适合实现 SOA 的技术，事实上现在 SOA 的扩展推广在很大程度上归功于 Web Services 标准的成熟和应用的普及。Web Services 标准的成熟应用为广泛地实现 SOA 架构提供了理论基础和实践环境。下面让我们看看 Web Services 中的各种协议是如何互相工作来满足 SOA 所需的特点的。

- **独立的功能实体**：通过 UDDI 的目录查找，服务提供者可以动态改变一个服务的提供方而无须影响客户端的应用程序配置。所有的访问都通过 SOAP 访问进行，只要 WSDL 接口封装良好，外界客户端是根本没有办法直接访问服务器端的数据的。
- **大数据量低频率访问**：通过使用 WSDL 和基于文本（Literal）的 SOAP 请求，开发人员可以实现能一次性接收大量数据的接口。这里需要着重指出的是 SOAP 请求分文本方式和远程调用（RPC）两种方式。
- **基于文本的消息传递**：Web Services 中大部分通信是通过 SOAP 进行的，而 SOAP 是基于 XML 的，不同版本之间可以使用不同的 DTD 或者 XML Schema 加以辨别和区分。因此只需要为不同的版本提供不同的处理就可以轻松实现版本控制。

2.2　Web Services 的相关规范

Web Services 的相关规范包括基本 Web Services 规范——WSDL、SOAP、UDDI，以及扩展的 WS 规范——WS-*规范。

2.2.1　Web Services 简介

Web Services 是用标准的、规范的 XML 概念描述的，称为 Web Services 的服务描述。这一描述囊括了与服务交互需要的全部细节，包括消息格式（详细描述操作）、传输协议和位置。该接口隐藏了实现服务的细节，允许独立于实现服务基于的硬件或软件平台和编写服务所用的编程语言使用服务。该接口允许并支持基于 Web Services 的应用程序成为松散耦合、面向组件和跨越具体技术的实现。Web Services 履行一项特定的任务或一组任务。Web Services 可以单独或同其他 Web Services 一起用于实现复杂的聚集或交互调用。

Web Services 是独立的、模块化的应用，能够通过互联网来描述、发布、定位以及调用。Web Services 是一种能够被描述并通过网络发布、发现和调用的自包含、自描述、松散耦合的软构件。在 Web Services 体系中（如图 2-3 所示），所有的应用实体都被抽象成服务。在 Web Services 的体系架构中包括三个角色：服务提供者（Service Provider）、服务请求者（Service Requestor）、服务注册器（Service Registry）。角色间主要有三个操作：发布（Publish）、查找（Find）、绑定（Bind）。

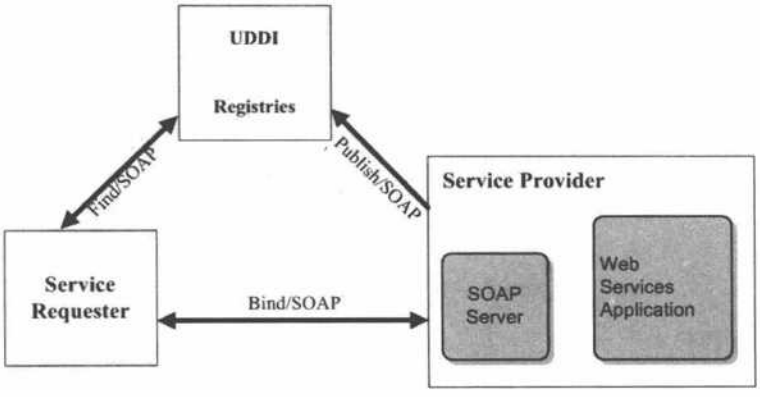


图 2-3　Web Services 结构模型

三个角色的说明：①服务提供者（Service Provider），也叫服务生产者，从业务角度看服务提供者是指服务的所有者，从体系结构上看它是指提供服务的平台；②服务请求者（Service Requester），也叫服务消费者，从业务角度看服务请求者是指需要请求特定功能的企业，从体系结构上看它是指查找和调用服务的客户端应用程序；③服务代理（Service Broker），也叫服务中介者，它是指用来存储服务描述信息的信息库（Repository），服务提供方在这里发布他们的服务，服务请求方在这里查找服务，获取服务的绑定信息。

与 Web Services 有关的操作主要有以下三种：①发布，服务提供者需要首先将服务进行一定描述并按照规范格式发布到注册服务器上，在发布操作中，服务提供者需要通过注册服务器的身

份验证和安全通道，才能对服务描述信息进行发布、修改和删除；②查找，服务请求方根据注册服务器提供的规范接口发出查询请求，以获取绑定服务所需的相关信息，在查找操作中，一般包含两种查找模式，一种是浏览模式（Browse Pattern），即服务请求方可以根据通用的分类标准来浏览或者通过一些关键字来搜索，并逐步缩小查找的范围，直到找到满足需要的服务，查找结果是一系列服务的集合，另一种是直接获取模式（Drill down Pattern），即通过唯一的关键字直接得到特定服务的描述信息，其查找结果是唯一的；③绑定，服务请求方通过分析从注册服务器中得到的服务绑定信息，包括服务的访问路径、服务调用的参数、返回结果、传输协议、安全要求等，对自己的系统进行相应配置，进而远程调用服务提供者所提供的服务。

Web Services 以技术栈的形式规范了 Web Services 体系中的各种关键技术，包括服务的描述、发布、发现以及消息的传输等。该模型应建立在现有的和新兴的标准之上，例如，HTTP、可扩展标记语言（Extensible Markup Language，XML）、简单对象访问协议（Simple Object Access Protocol，SOAP）、Web Services 描述语言（Web Services Description Language，WSDL）以及通用描述、发现和集成（Universal Description Discovery and Integration，UDDI）。Web Services 参考模型如图 2-4 所示。

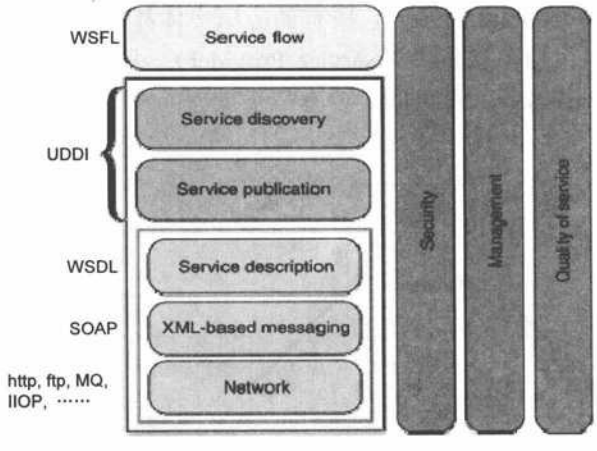


图 2-4　Web Services 参考模型

（1）XML 和 HTTP

这是 Web Services 最基本的平台。HTTP 是一个在 Internet 上广泛使用的协议，为 Web Services 组件通过 Internet 交互奠定了协议基础，并具有穿透防火墙的良好特性。XML 是一种元语言，可以用来定义和描述结构化数据，Web Services 实现的语言基础是 XML。Web Services 的其他协议规范都是以 XML 形式来描述和表达的。

（2）SOAP（Simple Object Access Protocol）

SOAP 协议，即简单对象访问协议，最先由 Microsoft 公司提交给 W3C 组织。它是 SOA 架构实现的线缆级协议，定义了服务请求者和服务提供者之间的消息传输规范。SOAP 用 XML 来格式化消息，用 HTTP 来承载消息。SOAP 包括 3 部分：定义了描述消息和如何处理消息的框架的封包（SOAP 封包）、表达应用程序定义的数据类型实例的编码规则（SOAP 编码规则）以及描述远程过程调用和应答的协定（SOAP RPC 表示）。

（3）WSDL（Web Services Description Language）

WSDL 由 Microsoft、IBM、Ariba 三家公司在 2000 年 9 月推出。它是由 Microsoft 公司的 SDL（Services Description Language）、IBM 公司的 NASSL（Network-Accessible Services Specification Language）合并后被 W3C 接纳所形成的标准。WSDL 为服务提供者提供以 XML 格式描述 Web Services 请求的标准格式，将网络服务描述为能够进行消息交换的通信端点的集合，以表达一个 Web Services 能做什么、它的位置在哪里、如何调用它等。

（4）UDDI（Universal Discovery, Description, Integration）

UDDI 规范由 Microsoft、IBM、Ariba 三家公司在 2000 年 7 月提出。它是在原有 Microsoft 提出的 DISCO（Discovery of Web Services）和 IBM 的 ADS（Advertisement and Discovery of Services）的基础上发展而来的。UDDI 是 Web Services 的信息注册规范，以便被需要该服务的用户发现和使用它。UDDI 规范描述了 Web Services 的概念，同时也定义了一种编程接口。通过 UDDI 提供的标准接口，企业可以发布自己的 Web Services 供其他企业查询和调用，也可以查询特定服务的描述信息，并动态绑定到该服务上。通过 UDDI，Web Services 可以真正实现信息的“一次注册，到处访问”。

Web Services 大多数情况下基于 XML 和 HTTP。对很多 Web Services 将基于 SOAP 消息交换作为系统间进行集成。这些消息由 XML 组成（XML 是一个基于文本的开放标准），可由来自任何应用程序（任何设计为接收此类消息的应用程序）的任何人进行访问。这就扩展了应用程序的范围，从而包含任何可通过网络对其进行访问的任何人。

但是 SOAP 和 WSDL 并不是唯一实现 Web Services 规范的技术。在 Web Services 的发展过程中，还有很多其他的相关技术，如 REST、XML-RPC 等。下面分别介绍这几种实现 Web Services 的技术。

（1）基于 SOAP 的 Web Services

基于 SOAP 的 Web Services 将要发送与下面配置信息中类似的 XML 消息。

```
<SOAPenv:Envelope
      xmlns:SOAPenv="http://schemas.xmlSOAP.org/SOAP/envelope/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<SOAPenv:Body>
         <req:getNumberOfArticles xmlns:req="http://daily-moon.com/CMS/">
         <req:category>classifieds</req:category>
         </req:getNumberOfArticles>
   </SOAPenv:Body>
</SOAPenv:Envelope>
```

这些消息将从一个系统进入另一个系统（通常通过 HTTP）。接收系统对消息进行解释，进行处理，然后发送另一个 SOAP 消息作为响应。

（2）REST（一种 Web Services 类型）

REST 是一种 Web Services 类型。REST 表现为用户可以直接访问 URL，相应的响应是与下面配置信息中类似的简单 XML 文档。

```
<currentArticles>
  <category>classifieds</category>
  <subcategory>forsale</subcategory>
  <article id="888204">
     <articleHeadline></articleHeadline>
     <articleText>30 ft ladder, only used once.
   </article>
  <article id="888242">
     <articleHeadline></articleHeadline>
     <articleText>Vintage 1963 T-Bird.  Less than 300 miles.
Driven by my daughter until I took it away.  Serious inquires only.
555-3264 after 7 PM.
</articleText>
   </article>
</currentArticles>
```

这些消息并没有特定的格式，可以是任何数据。

（3）使用 XML-RPC 之类的标准的 Web Services 类型

另一种 Web Services 类型要使用 XML-RPC 之类的标准。在这种情况下，命令将通过与下面配置信息中类似的 XML 发送到系统。

```
<?xml version="1.0"?>
<methodCall>
  <methodName>CMS.getNumberOfArticles</methodName>
  <params>
    <param>
      <value><string>classifieds</string></value>
    </param>
    <param>
      <value><string>forsale</string></value>
    </param>
  </params>
</methodCall>
```

其响应将采用类似的格式。

在某些情况下 REST 和 XML-RPC 比 SOAP 系统简单。但对于企业级应用程序，由于企业级的应用程序需要企业级的属性，如安全、互操作性，等等，这些功能在有关基于 SOAP 的 Web Services 的其他规范中进行了定义，因而，从长期来看，SOAP 更适用于企业级应用程序。

2.2.2 Web Services 架构及其 WS 规范简介

Web Services 规范通常归为两类：基本 Web Services 规范和扩展 Web Services 规范，其服务框架如图 2-5 所示。

基本的 Web Services 规范如下。

- **SOAP**：SOAP 规范是所有基于 SOAP 的 Web Services 的基础，详细说明了实际消息的格式。其中还详细说明了应用程序应如何对待消息的特定方面，从而可以创建特定类型的应用程序，使其中的消息在达到最终的目的地前在多个中间层之间进行传递。

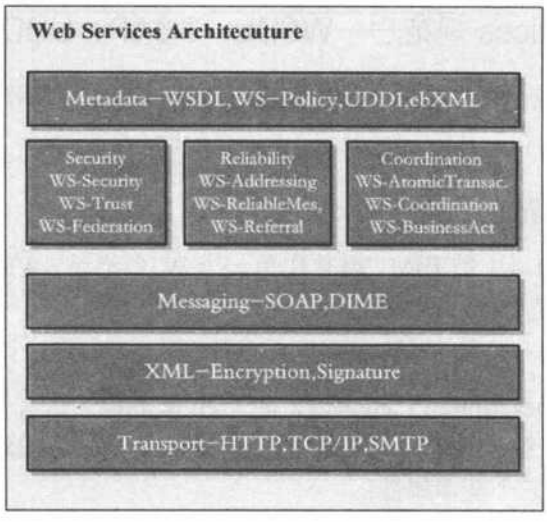


图 2-5　Web Services 规范的协议架构

- **WSDL**：Web 服务描述语言是详细说明描述基于 SOAP 的 Web Services 的标准方式的规范，包括消息应采用的形式以及应将其发送到何处。其中还详细说明了此类消息的响应。当与相应的工具结合使用时，WSDL 允许以编程方式创建对 Web 服务的调用；应用程序可以从 WSDL 文件中提取这些详细信息，并提供要使用的编程接口。
- **UDDI**：统一描述、发现和集成 （Universal Description, Discovery and Integration）是一项自最初提出后发生了一系列变化的标准。其最初的目的是为了给各公司提供在全球注册中心注册服务并在此注册中心搜索可能想使用的服务的机制。不过，由于很多公司对于将其系统对外开放的问题上都相当保守，这个目标并没有完全实现。但是，很多公司已将 UDDI 作为内部的服务及服务信息注册中心使用。另外还有数十种扩展标准，可进一步扩展基于 SOAP 的服务的用途。其中几种对企业尤为有用，如下所示。
  - **WS–Security 规范**：此规范处理加密和数字签名，允许创建特定类型的应用程序，以防止窃听消息，且能实现不可否认功能。
  - **WS–Policy 规范**：此规范对 WS-Security 规范进行了扩展，允许更具体地说明谁可以采用何种方式使用服务。
  - **WS–I 规范**：尽管 Web Services 应设计为具有互操作性，但在实际中，各规范对不同实现的解释的灵活性常常足以导致出现问题。WS-I 规范提供了一组可用于防止出现各种问题的标准和实践，并提供了标准化测试来检查问题。
  - **WS–BPEL 规范**：单个服务很好处理，但应用程序在大多数情况下较难处理。企业级计算要求至少将多个服务组合为一个完整的系统，而 WS-BPEL 规范提供了用于指定创建此类系统所必需的交互（如分支和并发处理）。

在 Web Services 中扮演重要角色的其他规范将不在本系列中讨论，其中包括 WS-ReliableMessaging 规范（允许确定检索一个且仅一个消息副本）、Web 服务资源框架（WSRF 规范）（允许使用在无状态环境中非常重要的状态）和 Web 服务分布式管理（WSDM 规范）（讨论 Web Services 的管理和使用问题）。

2.2.3 基本 Web Services 规范——WSDL、SOAP、UDDI

基本 Web Services 规范包括 WSDL 标准、SOAP 标准、UDDI 标准，下面详细描述这些标准的内容。

2.2.3.1 WSDL 规范

在 2001 年 3 月，WSDL 1.1 被 IBM、微软作为一个 W3C 记录（W3C note）提交到有关 XML 协议的 W3C XML 活动，用于描述网络服务。

1. WSDL 定义

WSDL（Web Service Description Language，即 Web 服务描述语言），是用 XML 文档来描述 Web 服务的标准，是 Web 服务的接口定义语言，由 Ariba、Intel、IBM、MS 等共同提出，通过 WSDL 可描述 Web 服务的三个基本属性：

- 服务做些什么——服务所提供的操作（方法）。
- 如何访问服务——和服务交互的数据格式以及协议。
- 服务位于何处——协议相关的地址，如 URL。

2. WSDL 简明概述

- WSDL 指网络服务描述语言，同时用于描述网络服务。
- WSDL 采用 XML 格式编写，同时也是一种 XML 文档。
- WSDL 也可用于定位网络服务。

3. WSDL 文档基本结构框架

WSDL 文档以端口集合的形式来描述 Web Services，WSDL 服务描述包含对一组操作和消息的一个抽象定义，绑定到这些操作和消息的一个具体协议，以及这个绑定的一个网络端点规范。

WSDL 文档被分为两种类型：服务接口（Service Interface）和服务实现（Service Implementation）。WSDL 文档可以分为两部分。顶部由抽象定义组成，抽象定义即服务接口（Service Interface）定义，而底部由具体描述组成。具体描述即服务实现（Service Implementation）定义。抽象部分以独立于平台和语言的方式定义 SOAP 消息，文档基本结构框架如图 2-6 所示。

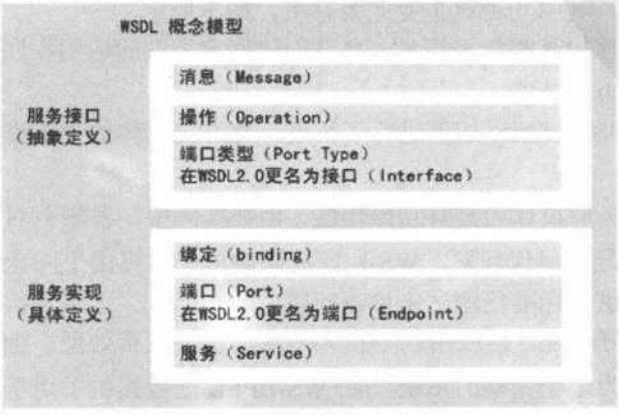


图 2-6 WSDL 文档基本结构框架

一个 WSDL 文档的主要结构示例如下：

```
<definitions>
```

```
<types>
  definition of types........
</types>

<message>
  definition of a message....
</message>

<portType>
  definition of a port.......
</portType>

<binding>
  definition of a binding....
</binding>
```

服务接口文档中的主要元素作用介绍如下：

（1）抽象定义

- Types：定义了 Web 服务使用的所有数据类型集合，可被元素的各消息部件所引用。它使用某种类型系统（一般情况下使用 XML Schema 中的类型系统）。这是独立于机器和语言的类型定义。
- Messages：通信消息数据结构的抽象类型化定义。使用 Types 所定义的类型来定义整个消息的数据结构，包括函数参数（输入与输出分开）或文档描述。
- PortTypes：引用消息部分中消息定义来描述函数签名（操作名、输入参数、输出参数）。对于某个访问入口点类型所支持操作的抽象集合。这些操作可以由一个或多个服务访问点来支持。

（2）具体定义

- Operation：对服务中所支持操作的抽象描述。一般单个 Operation 描述了一个访问入口的请求/响应消息对。
- Bindings：portType 部分的操作在此绑定实现，包含了如何将抽象接口的元素（portType）转变为具体表示的细节，具体表示也就是特定的数据格式和协议的结合，特定端口类型的具体协议和数据格式规范的绑定。
- Port：定义为协议/数据格式绑定与具体 Web 访问地址组合的单个服务访问点。
- Services：确定每一绑定的端口地址。这是一个粗糙命名的元素，代表端口的集合；相关服务访问点的集合。

可见，portType（与 message 和 type 元素的细节相结合）描述了 Web Services 是什么，binding 元素描述了如何使用 Web Services，port 及 service 元素描述了 Web Services 的位置。

2.2.3.2　SOAP 规范

2000 年 5 月，UserLand、Ariba、Commerce One、Compaq、Developmentor、HP、IBM、IONA、Lotus、Microsoft 以及 SAP 向 W3C 提交了 SOAP 因特网协议，这些公司期望此协议能够通过使用因特网标准 PortTypes（HTTP 及 XML）把图形用户界面桌面应用程序连接到强大的因特网服务器，以此来彻底变革应用程序的开发。当时这个规范是 SOAP 1.1。后 W3C 发展了 SOAP 1.2，

首个关于 SOAP 1.2 的公共工作草案由 W3C 在 2001 年 12 月发布。

SOAP 提供了一种标准的方法，使得运行在不同操作系统上并使用不同技术和编程语言的应用程序可以互相进行通信。

1. SOAP 定义

SOAP 是基于 XML 的简易协议，可使应用程序在 HTTP 之上进行信息交换。SOAP 是用于访问网络服务的协议。SOAP 的特征包括：SOAP 指简易对象访问协议，SOAP 是一种通信协议，SOAP 用于应用程序之间的通信，SOAP 是一种用于发送消息的格式，SOAP 被设计用来通过因特网进行通信，SOAP 独立于平台和语言，SOAP 基于 XML，SOAP 可绕过防火墙，SOAP 被作为 W3C 标准来发展等。

SOAP 有一些重要的语法规则，如 SOAP 消息必须用 XML 来编码、SOAP 消息必须使用 SOAP Encoding 命名空间、SOAP 消息不能包含 DTD 引用、SOAP 消息不能包含 XML 处理指令等。

2. SOAP 消息的基本结构

一条 SOAP 消息就是一个普通的 XML 文档，包含下列元素：

- 必需的 Envelope 元素，可把此 XML 文档标识为一条 SOAP 消息。
- 可选的 Header 元素，包含头部信息。
- 必需的 Body 元素，包含所有的调用和响应信息。
- 可选的 Fault 元素，提供有关在处理此消息时所发生错误的信息。

SOAP 消息的基本结构如图 2-7 所示。

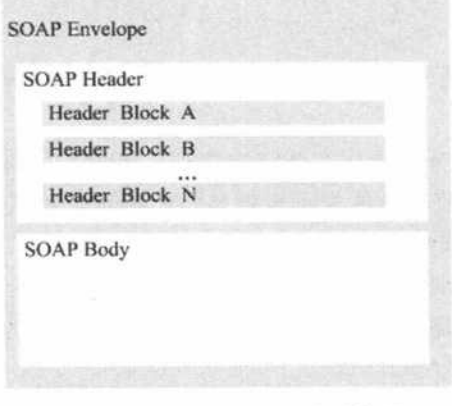


图 2-7 SOAP 消息的基本结构图

一个 SOAP 消息的文档结构如下所示。

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Header>
  ...
  ...
```

```
</soap:Header>

<soap:Body>
   ...
   ...
   <soap:Fault>
      ...
      ...
   </soap:Fault>
</soap:Body>

</soap:Envelope>
```

3. SOAP 的组成

SOAP 包括 SOAP Envelope 元素、SOAP Header 元素、SOAP Body 元素。

（1）SOAP Envelope 元素

强制使用 SOAP 的 Envelope 元素是 SOAP 消息的根元素。

Envelope 元素是 SOAP 消息的根元素。它可把 XML 文档定义为 SOAP 消息。请注意 xmlns:soap 命名空间的使用。它的值应当始终是：

```
<element attribute="attribute-value">contents</element>
```

http://www.w3.org/2001/12/soap-envelope 并且它可把封装定义为 SOAP 封装：

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap=http://www.w3.org/2001/12/soap-envelope
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  ...
  Message information goes here
  ...
```

SOAP 消息必须拥有与命名空间 "http://www.w3.org/2001/12/soap-envelope" 相关联的一个 Envelope 元素。如果使用了不同的命名空间，应用程序会发生错误，并抛弃此消息。

SOAP 的 encodingStyle 属性用于定义在文档中使用的数据类型。此属性可出现在任何 SOAP 元素中，并会被应用到元素的内容及元素的所有子元素上。SOAP 消息没有默认的编码方式。此属性语法如下：

```
soap:encodingStyle="URI"
```

实例如下所示：

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap=http://www.w3.org/2001/12/soap-envelope
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
...
Message information goes here
...
</soap:Envelope>
```

（2）SOAP Header 元素

可选的 SOAP Header 元素包含头部信息。

可选的 SOAP Header 元素可包含有关 SOAP 消息的应用程序专用信息（如认证、支付等）。如果 Header 元素被提供，则它必须是 Envelope 元素的第一个子元素。

注意：所有 Header 元素的直接子元素必须是合格的命名空间。

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap=http://www.w3.org/2001/12/soap-envelope
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Header>
<m:Trans xmlns:m=http://www.w3school.com.cn/transaction/ soap:mustUnderstand="1">
234</m:Trans>
</soap:Header>
...
...
```

上面的例子包含了一个带有“Trans”元素的头部，它的值是“234”，此元素的“mustUnderstand”属性的值是“1”。

SOAP 在默认的命名空间中（"http://www.w3.org/2001/12/soap-envelope"）定义了三个属性。这三个属性是：actor、 mustUnderstand 及 encodingStyle。这些被定义在 SOAP 头部的属性可定义容器如何对 SOAP 消息进行处理。

通过沿着消息路径经过不同的端点，SOAP 消息可从某个发送者传播到某个接收者。并非 SOAP 消息的所有部分均准备传送到 SOAP 消息的最终端点，可能会传送给消息路径上的一个或多个端点。SOAP 的 actor 属性可被用于将 Header 元素寻址到一个特定的端点，其语法如下所示：

```
soap:actor="URI"
```

实例如下所示：

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Header>
<m:Trans
xmlns:m="http://www.w3school.com.cn/transaction/"
soap:mustUnderstand="1">234</m:Trans>
</soap:Header>

...
...
```

SOAP 的 mustUnderstand 属性可用于标识标题项对于要对其进行处理的接收者来说是强制的还是可选的。若向 Header 元素的某个子元素添加了“mustUnderstand="1"”，则它可指示处理

此头部的接收者必须认可此元素。若此接收者无法认可此元素，则在处理此头部时必须失效。

其语法如下所示：

```
soap:mustUnderstand="0|1"
```

实例如下所示：

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap=http://www.w3.org/2001/12/soap-envelope
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Header>
<m:Trans xmlns:m=http://www.w3school.com.cn/transaction/ soap:mustUnderstand="1">
234
</m:Trans>
</soap:Header>
...
...
```

（3）SOAP Body 元素

强制使用的 SOAP Body 元素包含实际的 SOAP 消息。

SOAP Body 元素的直接子元素可以是合格的命名空间。SOAP 在默认的命名空间中（"http://www.w3.org/2001/12/soap-envelope"）定义了 Body 元素内部的一个元素，即 SOAP 的 Fault 元素，用于指示错误消息。

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body>
  <m:GetPrice xmlns:m="http://www.w3school.com.cn/prices">
    <m:Item>Apples</m:Item>
  </m:GetPrice>
</soap:Body>
```

上面的例子请求苹果的价格。请注意，其中的“m:GetPrice”和“Item”元素是应用程序专用的元素，它们并不是 SOAP 标准的一部分。而一个 SOAP 响应应该类似这样：

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap=http://www.w3.org/2001/12/soap-envelope
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body>
      <m:GetPriceResponse xmlns:m="http://www.w3school.com.cn/prices">
         <m:Price>1.90</m:Price>
      </m:GetPriceResponse>
</soap:Body>
</soap:Envelope>
```

4. SOAP 的优点

- **SOAP 是可扩展的。**SOAP 无须中断已有的应用程序，SOAP 客户端、服务器和协议自身

都能扩展，而且 SOAP 能极好地支持中间介质和层次化的体系结构。

- **SOAP 是简单的。**客户端发送一个请求，调用相应的对象，然后服务器返回结果。这些消息是 XML 格式的，并且封装成符合 HTTP 协议的消息。因此，它符合任何路由器、防火墙或代理服务器的要求。
- **SOAP 完全和厂商无关。**SOAP 可以相对于平台、操作系统、目标模型和编程语言独立实现。另外，传输和语言绑定以及数据编码的参数选择都是由具体的实现决定的。
- **SOAP 与编程语言无关。**SOAP 可以使用任何语言来完成，只要客户端发送正确 SOAP 请求（也就是说，传递一个合适的参数给一个实际的远端服务器）。SOAP 没有对象模型，应用程序可以捆绑在任何对象模型中。
- **SOAP 与平台无关。**SOAP 可以在任何操作系统中在无须改动的情况下正常运行。

SOAP 的相关规范可以在网站 http://www.w3.org/TR/soap12-part0/ 上获取。

2.2.3.3 UDDI 规范

UDDI 版本 1 规范于 2000 年 9 月发布，版本 2 于 2001 年 6 月发布。版本 1 打下了注册中心的基础，版本 2 添加了企业关系等功能。

UDDI 项目支持和鼓励 Web Services 之间的互操作和互利用。它是一种居于领先地位的企业之间的伙伴关系，这种关系最早是由 IBM、Ariba 和 Microsoft 建立的。UDDI 提供了一组基于标准的规范用于描述和发现服务，还提供了一组基于因特网的实现。

1. UDDI 定义

UDDI（Universal Description, Discovery and Integration）是一个主要针对 Web 服务供应商和使用者的应用。UDDI 应用中的成员可以通过 UDDI Business Registry（UBR）来操作 Web Services 的调用，UBR 是一个全球性的服务。Web Services 供应商可以在 UBR 中描述并且注册他们的服务。

用户可以在 UBR 中查找并定位那些需要的服务。UDDI 是一种根据描述文档来引导系统查找相应服务的机制。UDDI 有三种查询方式，第一是标准的“白皮书”类型的商业查询方式，第二是“黄皮书”类型的局部查找，第三是“绿皮书”类型的服务类型查找。UDDI 利用 SOAP 消息机制（标准的 XML/HTTP）来发布、编辑、浏览以及查找注册信息，并采用 XML 格式来封装各种不同类型的数据，发送到注册中心或者由注册中心来返回需要的数据。

UDDI 构建于网络传输层和基于 SOAP 的 XML 消息传输层之上。诸如 WSDL 之类的服务描述语言提供了统一的 XML 词汇供描述 Web 服务及其接口使用。这与交互式数据语言（Interactive Data Language，IDL）类似。开发人员可以通过添加分层的功能搭起整个基础，如使用 Web 服务流程语言（Web Services Flow Language，WSFL）的 Web 服务工作流描述、安全性、管理和服务质量功能，从而解决系统可靠性和可用性问题。

2. UDDI 基本概念

- **UDDI 规范：**UDDI 规范 V1 版包括两个规范文本，即 UDDI Programmer's API V1.0（UDDI 程序员 API 规范 1.0）和 UDDI Data Structure Reference V1.0（UDDI 数据结构参考 1.0）。前者定义了 UDDI Operator Site 能够支持的 API 接口，而后者描述了在 API 中 XML 描述

的数据结构的具体定义。UDDI 规范是 UDDI Operator Site 的实现蓝本，也是访问 UDDI Registry 的 Web Services 的参考规范。

- **UDDI Registry（UDDI 注册中心）**：UDDI Registry 是所有提供公共 UDDI 注册服务的站点的通称。UDDI Registry 是一个逻辑上的统一体，在物理上则是以分布式系统的架构实施的，而不同站点之间是采用 P2P（对等网络）架构实施的，因此访问其中任意一个站点就基本等于访问了 UDDI Registry。
- **UDDI Operator Site（UDDI 注册中心操作入口站点，简称 UDDI 操作入口）**：UDDI Operator Site 是 UDDI Registry 中每一个对等结点，对于 UDDI Operator Site 的查询所获得的结果是覆盖全 UDDI Registry 中的信息的，信息查询无须身份认证，而在 UDDI Operator Site 上进行信息发布则必须使用该 UDDI Operator Site 自身的用户，同时以后的更新、删除都必须通过这个 Operator Site，并使用初始发布时使用的用户进行权限认证。
- **Compatible UDDI Registry（兼容的 UDDI 注册中心）**：所有兼容 UDDI 规范但并非属于提供公共服务的 UDDI Registry 的个别 UDDI 注册中心，都称为兼容的 UDDI 注册中心。

3. UDDI 工作原理

统一描述、发现和集成协议（UDDI）规范一个由 Web Services 所构成的逻辑上的云状服务，同时也定义了一种编程接口，这种编程接口提供了描述 Web Services 的简单框架。规范包括几份相关的文档和一份 XML Schema，用来定义基于 SOAP 的注册和发现 Web Services 的协议。这些规范由来自多家业界主要公司的技术人员和管理人员花费了几个月的时间制定完成。这些公司也担负起实现第一批 UDDI 商业注册中心服务的任务，这些服务将可以被所有人所访问，同时其多个合作站点之间能够无缝地共享注册信息。

UDDI 注册中心包含了通过程序手段可以访问到的对企业和企业支持的服务所做的描述。此外，还包含对 Web Services 所支持的因行业而异的规范、分类法定义（用于对于企业和服务很重要的类别）以及标识系统（用于对于企业很重要的标识）的引用。UDDI 提供了一种编程模型和模式，它定义与注册中心通信的规则。UDDI 规范中所有 API 都用 XML 来定义，包装在 SOAP 信封中，在 HTTP 上传输。

图 2-8 说明了 UDDI 消息的传输，通过 HTTP 从客户机的 SOAP 请求传到注册中心结点，然后再反向传输。注册中心服务器的 SOAP 服务器接收 UDDI SOAP 消息并进行处理，然后把 SOAP 响应返回给客户机。就注册中心条例而言，客户机发出的要修改数据的请求必须确保是安全的、经过验证的事务。

图 2-9 说明了如何往 UDDI 注册中心送入数据，客户又如何能发现和使用这一信息。要使数据能在 UDDI 中物尽其用，需要几个步骤。例如第 1 步中，在软件公司和标准组织定义关于在 UDDI 中注册的行业或企业的规范时，开始向注册中心发布有用的信息。这些规范叫做技术模型或者更常见的说法是 tModel。其余见图中所示。

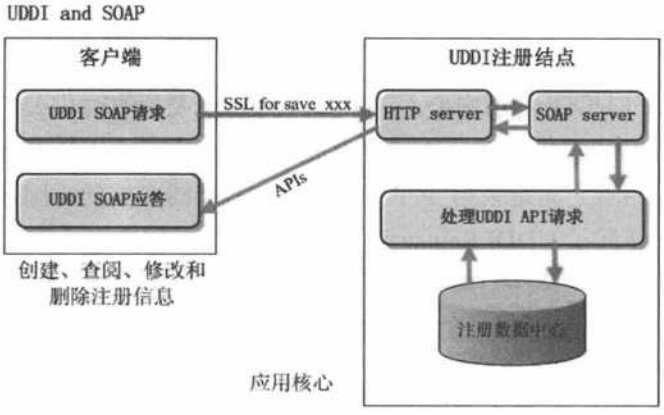


图 2-8　UDDI 消息在客户机和注册中心之间的流动

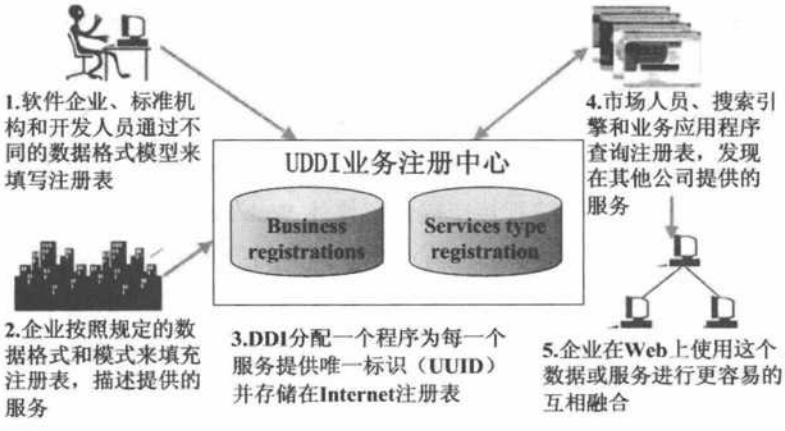


图 2-9　UDDI 工作原理

UDDI 规范的下载网址为 http://uddi.xml.org/。

2.2.4　扩展的 WS 规范——WS-*规范

扩展的 WS 规范，即指 WS-*规范，包括 WS-Policy 规范、WS-Addressing 规范、WS-ReliableMessaging 规范、WS-Security 规范、WS-SecurityPolicy 规范、WS-Trust 规范和 WS-SecureConversation 规范等。

2.2.4.1　WS–Policy 规范简介

WS-Policy（Web Services Policy Framework，Web 服务策略框架）规范提供了一种灵活、可扩展的语法，用 XML 来表示 Web Services 系统中实体的能力、要求和一般特性。WS-Policy 将策略定义为一组策略替换选项，其中每个策略替换选项又是一组策略断言。WS-Policy 规范现在已成为 Web Services 框架中一个重要的组件。

WS-Policy 定义了一个模型和一个框架，并将这些特性表示为策略。策略表示法既可以支持简单的声明式断言，也支持比较复杂的条件式断言。策略断言表示行为的个体要求、能力或其他特性。例如，开发人员可以创建一个策略断言，定义这样一个要求“必须使用受信任的 X.509 证

书签名作为 SOAP 消息主体”。WS-Policy 只是一个框架，为特定域定义一套特定的策略断言则是系统的任务。

在 WS-Policy 规范中，将策略断言分为两类，一类指传统的要求和功能，这些要求和功能最终将出现在网络中（如身份验证方案、传输协议选择）。另一类策略断言并不直接表现在网络中，但却对正确地选择和使用服务至关重要（如隐私策略、QoS 特性）。WS-Policy 只提供了一种语法，使这两种断言能够用一致的方式来组织策略。

WS-Policy 规范为服务请求者和服务提供者规范了语法和语义来描述他们的需求、首选项和性能。语法为基于策略形式表述每个领域的需求提供了一种灵活简洁的方法。领域在本文中指的是关系到服务重要性的通用描述，如下所示：

- 安全
- 隐私
- 应用程序优先权
- 用户账户优先权
- 传输控制

WS-Policy 规范还描述了这些策略的处理模型，处理模型独立于领域范畴。对于处理策略定义了 3 种操作：标准化、融合和交集。关于 WS-Policy 规范的介绍，在后面的章节还有详细内容。

2.2.4.2　WS–Addressing 规范简介

WS-Addressing（ Web 服务寻址）规范是一个 W3C 推荐标准，为 Web Services 提供一种与传输层无关的、传送寻址信息的机制。WS-Addressing 规范主要由两部分组成：传送 Web Services 端点的引用的数据结构，以及一套能够在特定的消息上关联寻址信息的消息寻址属性。

WS-Addressing 规范通过定义端点引用（EPR）和消息信息（MI）头来明确地达到这个目的。这个定义提供了单一的机制，通过这一机制，开发人员能够指定 Web Services 的位置以及服务在 SOAP 消息中使用端点引用（EPR）的方式（通过 MI 头）。但是，在消息处理模型/流中的隐式改变甚至对 SOAP 具有更加重大的影响。WS-Addressing 规范本身应该是核心 SOAP 规范的一部分。

2.2.4.3　WS–ReliableMessaging 规范简介

WS-ReliableMessaging 规范定义了一种协议和一套机制，使 Web Services 的开发人员能够确保在两个端点之间可靠地传递消息，该规范还具有各种传递保证和健壮性特征。该规范由 BEA、IBM、Microsoft 和 Tibco 联合发布。

2003 年 3 月，BEA、Microsoft、IBM 和 Tibco 软件公司共同撰写了最初的规范，在随后的两年里进行了改进和完善。2005 年 6 月，一个 WS-ReliableMessaging 版本被提交到 OASIS 的 Web Services 可靠交换（WS-RX）技术委员会，2007 年 6 月 14 日，被批准成为 OASIS 标准，2009 年 2 月 2 日，1.2 版本被批准。

在 WS-ReliableMessaging 之前，OASIS 曾提出过一个竞争的规范（WS-Reliability），该规范为富士通、日立、NEC、甲骨文和 Progress Software 所支持。这些支持者中的大多数现在也支持 WS-ReliableMessaging 规范。

WS-ReliableMessaging 规范把 Sequence 定义为一组消息的共享上下文，这组消息在源应用程序和目标应用程序之间进行交换，并有一致的传递保证要求。协议在源端点的实现至少必须能够在 Sequence 中唯一地标识并重发消息，直到目标端点确认接收到所有消息。目标端点上的实现必须能够在 Sequence 中跟踪接收到的消息并传递确认消息给源端点。

2.2.4.4 WS-Security 规范介绍

早期在因特网上传递 Web Services 领域的开发人员发现：他们必须要么提供开放的、任何人都可使用的服务，要么开发自己的服务，这些服务通常是私有的，必须定制于某个公司并且采用某种特定的安全模式。把因特网作为传输工具的采用者通常使用某种形式的注册流程（如 Google）公开因特网服务，或者仅仅向一小部分与自己有着亲密、可信关系的业务伙伴提供服务。例如，为了使用 Google 的启用 Web Services 的搜索引擎，服务请求者必须首先以基于 HTML 的形式向 Google 注册。作为注册流程的一部分，Google 向请求者发送一封带有一个安全性“令牌”的邮件。当请求者调用服务时，他们向 Google 提供这个作为 SOAP 消息的一部分的令牌来证明他们是 Google Web Services 已注册、已授权的用户。

在这些情况下，即使服务提供者使用的是 SOAP 之类的业界标准，还是需要提供有关安全性模式/流程的额外信息以使服务请求者能够使用服务。这样就会把请求者和提供者紧紧地捆绑在一起，而这种结果是双方都不喜欢的。

2002 年 4 月，IBM、Microsoft 和 Verisign 对这种需求提出了一个保护 Web Services 的业界标准方法。

WS-Security 规范以现有的密码学以及 XML 加密和签名行业标准为基础，为 Web Services 应用程序提供了一组全面的安全特性，可以通过 WS-Policy 和 WS-SecurityPolicy 来指定特定应用程序可以使用哪些特性，从而允许服务客户机自行配置以访问服务。通过跨多个平台和 Web Services 框架对这些标准的广泛支持，可以实现出色的互操作性。

WS-Security 规范覆盖以下三方面内容。

①WS-Security 规范描述通过消息完整性、消息机密性和 SOAP 消息传递的增强质量。这些机制可以用于提供多种安全性模型和加密技术。

②WS-Security 规范还提供关联安全性令牌和消息的通用机制。WS-Security 不需要特定类型的安全性令牌，它在设计上就是可扩展的（如支持多安全性令牌格式）。举例来说，客户机可能会提供身份证明和他们有特定商业认证的证明。

③WS-Security 还描述如何对二进制安全性令牌编码。此规范特别描述如何对 X.509 证书和 Kerberos 票据编码，以及如何加入难以理解的加密密钥。此规范还包括可以用于进一步描述消息中包含的凭证特征的扩展性机制。

WS-Security 规范很灵活。该规范可以被设计成用来构建多种安全性模型（包括 PKI、Kerberos 和 SSL）的基础。WS-Security 规范特别为多安全性令牌、多信任域、多签名格式和多加密技术提供支持。

WS-Security 规范提供了三种主要的机制：安全性令牌传播、消息完整性和消息机密性。这些机制本身并不提供完整的安全性解决方案。相反，WS-Security 是一种构件，它可以与其他 Web Services 扩展和更高级的特定于应用程序的协议联合使用，以适应多种安全性模型和加密技术。

这些机制可以独立使用（如传送安全性令牌），或以紧密集成的方式使用（例如，对消息签名和加密，并提供与用于签名和加密的密钥相关的安全性令牌层次结构）。

WS-Security 规范的推出紧接着 IBM 和 Microsoft 现有的 Web 服务安全性规范，包括 SOAP-SEC，Microsoft 的 WS-Security 和 WS-License，还有 IBM 的安全性令牌和加密文档。

2.2.4.5　WS-SecurityPolicy 规范简介

WS-SecurityPolicy 规范以 WS-Policy 规范为基础，采用声明方式来定义可用于指定安全相关策略的 XML 元素。声明允许 Web Services 明确地指定其策略。由 WS-SecurityPolicy 定义的声明包括以下几种。

- <SecurityToken> 允许 Web Services 指定 WS-SecurityPolicy 将接受哪些类型的安全令牌以及它所信任的令牌发行方（可选）。
- <Integrity> 允许 Web Services 指定 WS-SecurityPolicy 将接受的有关数字签名的多种选项，如用于创建消息摘要的算法。
- <Confidentiality> 允许 Web Services 指定有关应如何执行加密的选项，如所使用的加密算法。
- <Visibility> 允许 Web Services 指定消息的哪些部分必须进行解密。

WS-SecurityPolicy 规范没有定义如何准确地从设计者向任何感兴趣的客户端传输安全策略。表示策略的 XML 元素可以通过 SOAP 消息发送，也可以通过其他方式传输。但是 WS-Policy 定义了一个通用的<Policy>元素，该<Policy>元素可以包含一个或多个策略声明。以下是一个简单的示例，该示例说明了对要求使用 Kerberos 的票据进行身份验证的 Web Services。包括<SecurityToken> 声明的形式：

```
<wsp:Policy
 xmlns:wsp="http://schemas.xmlsoap.org/ws/2002/12/policy
 xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext">
   <wsse:SecurityToken wsp:Usage="wsp:Required">
      <TokenType> wsse:Kerberosv5ST</TokenType>
   </wsse:SecurityToken>
</wsp:Policy>
```

该<Policy>元素指定了在 WS-Policy 和 WS-Security 中定义的命名空间，并且只包含<SecurityToken>子元素。该子元素的使用被标记为 Required，这意味着 Web Services 需要遵守这个策略。换句话说，这个特定的 Web Services 现在只接受 Kerberos 票据作为身份验证的基础。

单个的<Policy>元素可以包含有关安全令牌、完整性以及其他方面的策略，还有几个其他选择，包括指定相关首选项的方法。例如，Web Services 可以定义一个<SecurityToken>，允许客户端使用 Kerberos 或 X.509 证书进行身份验证的<SecurityToken>，然后指出优先使用 Kerberos。但是，无论使用哪种方案，策略值都是显而易见的。通过提供一种明确的方法以便 Web Services 声明其安全要求，这样可以使客户端能够明白要访问该服务所必须做的事情。

WS-SecurityPolicy 规范的定义在 http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.3/ws-securitypolicy.html。

2.2.4.6 WS-Trust 规范简介

WS-Trust 规范定义了向安全令牌服务进行请求的形式，以及如何返回响应。

安全令牌是 WS-Security 规范的基础。例如，要获得 Kerberos 票据，可以与 Kerberos 密钥发行中心（KDC）联系；要获得 X.509 证书，则可以与证书颁发机构（CA）联系。尽管 WS-Security 规范定义了如何在标准 XML 内包装 Kerberos 票据和 X.509 证书以便在 SOAP 消息中进行传输，但它没有定义使用 SOAP 与 KDC、CA 或者能够分发安全令牌的任何其他服务进行交谈的方法。WS-Trust 规范修正了这个问题。这个规范定义的主要内容是从一些安全令牌服务（如 KDC 或 CA）请求安全令牌的机制。

WS-Trust 规范定义了向安全令牌服务进行请求的形式，以及如何返回响应。这种交互依赖于两个元素：<RequestSecurityToken>和<RequestSecurityTokenResponse>。其基本思想是发送包含<RequestSecurityToken>作为其正文的 SOAP 消息，然后获得包含<RequestSecurityTokenResponse>的 SOAP 消息以及新的安全令牌。

2.2.4.7 WS-SecureConversation 规范介绍

WS-SecureConversation 规范构建于 WS-Trust 规范基础上，使用一个 STS 来管理 SCT。一个 SCT 代表的是在消息交换所涉各方之间共享的上下文，此共享上下文内的信息允许各方使用非对称加密来确保信息的安全。此上下文向消息交换中所涉及的各方提供了一个共享的秘密值。这个共享的秘密值本身可以是在交换过程中用来对消息进行非对称加密的密匙。

WS-SecureConversation 规范在理论上可以用于多方的消息交换，但它最为常见的用法是用于一个客户机与一个服务器之间的通信。当用在这种配置中时，向客户机提供 SCT 的这个 STS 与此服务器位于一处，可在相同的端点地址访问到。这意味着服务器上的 Web Services 代码需要一种方式来辨别哪些消息是针对 STS、哪些消息是针对服务本身的，用此请求上的这个动作可服务此目的。

2.3 Java 中关于 SOA 的相关规范

Web Services 历史悠久。首先是 SOAP，可 SOAP 仅描述消息的情况，接着是 WSDL，WSDL 并不会说明如何使用 Java 编写 Web Services。在这种情况下，JAX-RPC 1.0 应运而生。经过数月使用 JAX-RPC1.0 之后，编写此规范的开发人员认识到需要对其进行一些调整，调整的结果就是 JAX-RPC 1.1。该规范使用大约一年之后，规范开发人员希望构建一个更好的版本，这就是 JAX-RPC 2.0。JAX-RPC 2.0 主要目标是与行业方向保持一致，但行业中不仅只使用 RPC Web Services，还使用面向消息的 Web Services。因此从名称中去掉了“RPC”，取而代之的是“WS”（当然表示的是 Web Services）。因此 JAX-RPC 1.1 的后续版本是 JAX-WS 2.0——Java API for XML-based Web Services。下面就分别介绍这几个关于 Java 的 Web Services 规范。

2.3.1 JAX-RPC 规范

JAX-RPC（Java API for XML-based RPC）是 Java 方面的 Web Services 的原始标准。虽然

JAX-RPC 的设计思想是可以为实际 Web Services 实现使用不同的协议实现，但在实践中，仅将其应用在基于 SOAP 协议的 Web Services。

JAX-RPC 1.0 对 Web Services 功能的认识也有一定的局限。从其名称可以看出，JAX-RPC 1.0 最初的设计目的是为了支持使用 XML 的远程过程调用（Remote Procedure Call，RPC）操作。Java 当时已经有了一项面向 Java 应用程序间的 RPC 调用的现有技术，即远程方法调用（Remote Method Invocation，RMI）。该规范编制团队选择了在现有 RMI 接口上对 JAX-RPC 进行建模。只要通过请求-响应操作使用 rpc/enc SOAP，此模型就可相当接近地进行匹配，不过映射到异步操作或其他传输的效果并不理想。2003 年年底，有关人员认识到，需要对 JAX-RPC 进行大幅修订，以处理这些问题和其他问题，因此当时的 Sun 组成了一个专家组开始进行 JAX-RPC 2.0 规范的开发。这就是后续版本 JAX-WS 2.0。

JAX-RPC 基于可扩展标记语言 XML 的远程过程调用的 Java 应用程序接口，是 Java Web Services 开发包（WSDP）的应用程序接口。JAX-RPC 为基于 SOAP 的应用程序开发提供了一个编程模型。JAX-RPC 编程模型通过抽象 SOAP 协议层的运行机制并提供 Java 和 WSDL 间的映射服务来简化开发。JAX-RPC 致力于要使应用程序或通过 Web Services 调用其他应用程序或 Web Services 变得更加容易。

Java 类或 Java 应用可以通过 JAX-RPC 包装并以 Web Services 的形式发布。JAX-RPC 提供了将 RPC 参数（in/out）编码和解码的 API，使开发人员可以方便地使用 SOAP 消息来完成 RPC 调用。同样，对于那些使用 EJB 的商业应用而言，同样可以使用 JAX-RPC 来包装成 Web Services，而 Web Services 的 WSDL 界面与原先 EJB 的方法是对应一致的。JAX-RPC 为用户包装了 Web Services 的部署和实现。对 Web Services 的开发人员而言，SOAP/WSDL 变得透明，这有利于加速 Web Services 的开发周期并缩短开发时间。

JAX-RPC 规范包括下列主题：①类型映射系统；②服务端点；③异常处理；④服务端点上下文；⑤消息处理程序；⑥服务客户机和服务上下文；⑦带附件的 SOAP；⑧运行时服务；⑨JAX-RPC 客户机调用模型。下面分别介绍这 9 个主题。

- **类型映射系统**：JAX-RPC 规范最重要的成就之一就是定义了一个标准，该标准用于将 WSDL 文档映射为其 Java 表示（服务端点、存根、绑定以及 Java 类型），反之亦然。以前这是一个主要的易引起混淆的地方，因为每个 Web Services 工具箱供应商对于相关的标准（WSDL、SOAP 和 XML）以及这些标准之间的关系都有自己的解释。因为这些标准并行发展并且尚未成熟到足可依赖的程度，所以更增加了这种混淆。
- **服务端点**：JAX-RPC 规范的服务端点指的是真正的服务实现所依赖的 Web 服务端点。那里有一个服务端点实现（具体的）类和一个服务端点接口的定义。
- **异常处理**：JAX-RPC 规范试图在应用程序级和运行系统级处理 Web Services 运行时的异常，这是基于服务端点接口的标准设计方法以及它对 wsdl.fault 元素的映射。
- **服务端点上下文**：JAX-PPC 规范允许运行时系统灵活地管理上下文信息。每一个运行时环境可以维护自己独特的上下文信息。例如，基于 Servlet 的运行时系统维护一个 ServletEndpointContext 对象。EJB 2.1 规范定义了 EJB SessionContext。

- **消息处理程序**：消息处理程序是 JAX-RPC 规范的最强大的功能。消息处理程序向 Web Services 端点（客户机和服务器）提供了附加的消息处理功能，作为对基本服务实现逻辑的扩展。处理程序可以处理加密和解密、日志记录和审计等。当前的 JAX-RPC 规范运行时系统仅仅定义了 SOAP 消息处理程序，但是 JAX-RPC 规范可以很灵活地定义其他处理程序，而且不需要任何消息处理模型。
- **服务客户机和服务上下文**：客户机 JAX-RPC 的亮点之一就在于它能把上下文信息和端点的远程方法调用关联起来。可 JAX-RPC 规范并不强求上下文信息的语义。用户可以根据 WSDL 绑定中 SOAP 头显式定义上下文信息的语义，也可以根据 WS-Security 规范之类的标准定义上下文信息的语义，还可以通过使用特定于绑定的细节（如 HTTP 请求头）来定义上下文信息的语义。这个运行时上下文信息可以由容器或者客户机来设定。容器管理的上下文管理被称为隐式上下文管理，而客户机管理被称为显式的上下文管理。在这里使用隐式一词是因为在隐式上下文管理中，客户机或服务器上都不需要进行编程来支持上下文传输，这种支持是由运行时引擎提供的。这种上下文信息的示例包括安全和事务信息。显式服务上下文的表现形式是作为追加于服务方法调用的附加参数。当从 Java 参数映射到 WSDL 时这可能会引发问题，因为这些增加的元素将映射到 WSDL 头。
- **带附件的 SOAP**：JAX-RPC 规范 API 在远程过程调用和/或返回值时支持 MIME 编码的内容的使用。这是以带附件的 SOAP 标准为基础的。带附件的 SOAP 消息是通过使用 MIME multipart/related 类型建立的。根部件是初始的 SOAP 消息，MIME 内容被作为消息的其他部件添加。这些 SOAP 部件可能包含对 MIME 部件的引用，也要注意每一个 MIME 部件包含内容的 ID 或内容定位信息来唯一地标识 MIME 部件。
- **运行时服务**：这里有几个必须由 JAX-RPC 运行时基础架构支持的确定的要求。其中比较重要的有两个，即安全性和会话管理。安全性，JAX-RPC 运行时系统必须支持最基本的 HTTP 认证。客户机发送用户名和密码到 HTTP 服务器，它们在那里被验证（HTTP 服务器被认为是 JAX-RPC 运行时系统的一部分）。所有其他的安全机制（数字证书、SSL、WS-Security 等）都是由客户机和服务器运行时系统所提供的增值功能。这些可以用 JAX-RPC 处理程序和服务传播来处理。会话管理，这个要求使客户机能参与和服务端点的会话。JAX-RPC 的运行时系统至少应该支持下面中的一个：基于 Cookie 的会话管理、URL 重写、SSL 会话等。服务器初始化会话管理进程，可以通过设置 session.maintain 属性为 true 来表明服务器就绪以支持会话。
- **JAX-RPC 客户机调用模型**：下面是三个不同的从客户机调用服务端点的模型。它们独立于任何特定于服务实现的模型。
  - **基于存根的模型（静态存根）**。开发人员有两种方法可以创建服务存根：来自 Java 服务端点接口和来自服务的 WSDL 描述。可以从 WSDL 或从服务端点接口生成一个存根类。一个生成了的存根类是实现 javax.xml.rpc.stub 接口和服务端点接口所必需的。通过设置诸如端点地址、会话、用户名、密码这样的属性，这个存根接口提供 API 以配置存根。

- **动态代理**。和上面所讨论的静态存根相反的是，运行时客户机使用 javax.xml.rpc.Service 接口来创建动态服务代理存根客户。客户机能够推算出关于 WSDL 和要调用的服务信息。它使用 ServiceFactory 类来创建服务获取代理。
- **DII（动态调用接口）**。这个软件模式省略了客户预先知道服务的确切名字和需要的参数。DII 客户机在运行时可以使用查找服务信息的服务代理程序发现这个信息。服务发现中的灵活性使运行时系统能使用服务代理程序，服务代理程序能够采用各种服务发现机制（如 ebXML、注册表、UDDI 等）。

获得 JAX-RPC 1.1 的规范、API 类和 Java doc 的下载地址：http://java.sun.com/xml/downloads/jaxrpc.html。

2.3.2　JAX-WS 规范

JAX-WS 全称为 Java API for XML-Based Web Services。该规范是一组 XML Web Services 的 Java API。JAX-WS 规范是用于简化使用 Java 构造 Web Services 和 Web Services 客户机的工作的技术。该技术提供了完整的 Web Services 堆栈，可减少开发和部署 Web Services 的工作量。JAX-WS 允许开发者可以选择 RPC-oriented 或者 message-oriented 来实现自己的 Web Services。在 JAX-WS 中，一个远程调用可以转换为一个基于 XML 的协议，如 SOAP。在使用 JAX-WS 过程中，开发者不需要编写任何生成和处理 SOAP 消息的代码。JAX-WS 在运行时实现这些 API 的调用转换成为对应的 SOAP 消息。在服务器端，用户只需要通过 Java 语言定义远程调用所需要实现的接口 SEI（Service Endpoint Interface），并提供对应的实现，通过调用 JAX-WS 的服务发布接口就可以将其发布为 Web Services 接口。在客户端，用户可以通过 JAX-WS 的 API 创建一个代理（用本地对象替代远程的服务）来实现对于远程服务器端的调用。通过 Web Services 所提供的互操作环境，开发人员可以用 JAX-WS 规范轻松实现 Java 平台与其他编程环境（.NET 等）的互操作。

JAX-WS 支持 WS-I Basic Profile 1.1，后者可确保使用 JAX-WS 堆栈开发的 Web Services 能够给基于 WS-I Basic Profile 标准使用任意语言开发的任意客户机使用。JAX-WS 还包括了 Java Architecture for XML Binding（JAXB）和 SOAP with Attachments API for Java（SAAJ）。而且，JAX-WS 提供了将传统 Java 对象（Plain Old Java Object，POJO）类转换为 Web Services 的 Annotation 库，从而加速了 Web Services 的开发工作。另外，JAX-WS 还指定了 WSDL 定义的服务到实现该服务的 Java 类之间的详细映射。采用 WSDL 定义的任意复杂类型都通过遵循 JAXB 规范定义来映射为 Java 类。JAX-WS 之前与 Java Platform, Enterprise Edition（Java EE 5）绑定。而 JAX-WS 2.0 规范是作为 Java Community Process（JCP）的 JSR 224 开发的。

JAX-WS 2.0 事实上是对 JAX-RPC 1.0 规范的扩展，也是 JAX-RPC 1.1 的后续版本。JAX-RPC 2.0 标准发布不久后便被重新命名为 JAX-WS 2.0。JAX-WS 2.0 提供给 Java 5 开发 Web Services 的最新编程标准，JAX-WS 2.0 提供了新的编程模型和增强以往的 JAX-RPC 方式下的 Web Services 开发。

和以往的 JAX-RPC 相比，JAX-WS 2.0 的主要特点如下：

- 新的 API 和新的编程模型。新的 API 主要包含在 javax.xml.ws 包中，包括服务器端和客户端的一些核心类。新的编程模型包括增强的 Handler Framework、异步调用和

Provider/Dispatch 动态编程模型。

- 采用注释来描述 Web Services，不再依赖部署过去的 Web Services 描述文件。
- 通过 JAXB 2.0 完成 XML 数据和 Java 对象的绑定，改变了 JAX-RPC 1.0 里同时存在两套 binding 机制的不便。
- MTOM/XOP（SOAP Message Transmission Optimization Mechanism/XML Binary Optimized）和 swaRef（SOAP Attachment Refenreces），解决了在 SOAP 消息里传输二进制附件的问题，同时对消息传输也进行了优化。
- 对 SOAP 1.2 提供支持。

开发 Web Services 有契约优先方法与代码优先方法，契约优先是指从 WSDL 契约着手，生成 Java 类来实现服务。代码优先是指从 Java 类着手，使用注释来生成 WSDL 文件和 Java 接口。

获得 JAX-WS 2.0 的规范、API 类和 Java doc 的下载地址：http://jcp.org/aboutJava/communityprocess/pfd/jsr224/index.html。

2.3.3 JAX-RS 规范

REST 是 Representational State Transfer 的缩写，意为“代表性状态传输”。REST 这个术语是由 Roy Fielding 在他的博士论文“Architectural Styles and the Design of Network-based Software Architectures”中提出的。REST 并不是一种标准，而是一种开发 Web 应用的架构风格，可以将其理解为一种设计模式。REST 是基于 HTTP 的 URI，以及 XML 这些现有的广泛流行的协议和标准，伴随着 REST 的广泛使用，HTTP 协议得到了更加正确的使用。

REST 在 Web 领域已经得到了广泛的接受，这是对基于 SOAP 和 WSDL 的 Web Services 的更为简单、更为容易的替代方法。接口设计方面这一转变的关键证据是主流 Web 2.0 服务提供者（包括 Yahoo、Google 和 Facebook）对 REST 的采用，这些提供者弃用或放弃了基于 SOAP 和 WSDL 的接口，而采用了更易于使用、面向资源的模型来公开其服务。

JSR-311（即 JAX-RS：Java API for RESTful Web Services）是为 Java EE 环境下的 RESTful 服务能力提供的一种规范。它能提供对传统的基于 SOAP 的 Web Services 的一种可行替代。Java EE 6 中引入对 JSR-311 的支持。

相较于基于 SOAP 和 WSDL 的 Web Services，REST 模式提供了更为简捷和轻量级的实现方案。目前，越来越多的 Web Services 开始采用 REST 风格进行设计和实现。现实中比较著名的 REST 服务包括：Google Ajax 搜索 API、Amazon Simple Storage Service（Amazon S3）等。

基于 REST 的 Web 服务遵循的一些基本的设计原则如下：

- 系统中的每一个对象或资源都可以通过一个唯一的 URI 来进行寻址，URI 的结构应该简单、可预测同时易于理解，如定义目录结构式的 URI。
- 以遵循 RFC-2616 所定义的协议方式显式地使用 HTTP 方法，建立创建、检索、更新和删除（CRUD：Create、Retrieve、Update and Delete）操作与 HTTP 方法之间的一对一映射。
  - 若要在服务器上创建资源，应该使用 POST 方法；
  - 若要在服务器上检索某个资源，应该使用 GET 方法；

  - 若要在服务器上更改资源状态或对其进行更新，应该使用 PUT 方法；
  - 若要在服务器上删除某个资源，应该使用 DELETE 方法。
- URI 所访问的所有资源都可以使用不同的形式加以表示（如 XML 或者 JSON），具体的表现形式取决于访问资源的客户端，客户端与服务提供者使用一种内容协商的机制（请求头与 MIME 类型）来选择合适的数据格式，最小化彼此之间的数据耦合。

Java EE 6 引入对 JSR-311 的支持。JSR-311 旨在定义一个统一的规范，使得 Java 程序员可以使用一套固定的接口来开发 REST 应用，避免了依赖第三方框架。同时，JAX-RS 使用 POJO 编程模型和基于标注的配置，并集成了 JAXB，从而可以有效地缩短 REST 应用的开发时间。

JAX-RS 定义的 API 位于 javax.ws.rs 包中， JAX-RS 的具体实现可以由第三方提供，如当时 Sun 的参考实现 Jersey，Apache 的 CXF 以及 JBoss 的 RESTEasy。

REST Web 服务具体实现应该遵循以下 4 个基本设计原则：

- 显式地使用 HTTP 方法。
- 无状态。
- 公开目录结构式的 URI。
- 传输 XML、JavaScript Object Notation（JSON），或同时传输这两者。

下面分别简单介绍这 4 个基本设计原则。

- **显式地使用 HTTP 方法：**基于 REST 的 Web Services 的主要特征之一是以遵循 RFC 2616 定义协议的方式显式地使用 HTTP 方法。REST 要求开发人员显式地调用 HTTP 方法，并且要求调用方式与协议定义一致。REST 基本设计原则建立了创建、读取、更新和删除（create, read, update and delete，CRUD）操作与 HTTP 方法之间的一对一映射。根据此映射：①若要在服务器上创建资源，应该使用 POST 方法；②若在服务器上检索某个资源，应该使用 GET 方法；③若要在服务器上更改资源状态或对其进行更新，应该使用 PUT 方法；④若要在服务器上删除某个资源，应该使用 DELETE 方法。
- **无状态：**REST Web Services 需要扩展以满足日益提高的性能要求、负载平衡和故障转移功能。代理和网关服务器集群通常以构建成服务拓扑的方式进行组织，从而允许根据需要将请求从一个服务器路由到另一个服务器，这样可以减少 Web Services 调用的总体响应时间。要使用中间服务器扩大规模，REST Web Services 需要发送完整、独立的请求。也就是说，发送的请求包括所有需要满足的数据，以便于中间服务器中组件能够进行转发、路由和负载平衡，而不必在请求之间在本地保存任何状态。
- **公开目录结构式的 URI：**REST Web 服务 URI 的直观性可以达到很容易猜测的程度。将 URI 看作是自身配备文档说明的接口，开发人员只需很少的解释或参考资料即可了解它指向什么，并获得相关的资源。为此，URI 的结构应该简单、可预测且易于理解。实现这种级别的可用性的方法之一是定义目录结构式的 URI。此类 URI 具有层次结构，其根为单个路径，从根开始分支的是公开服务的主要方面的子路径。根据此定义，URI 并不只是斜杠分隔的字符串，而是具有在节点上连接在一起的下级和上级分支的树。
- **传输 XML、JSON 或同时传输这两者：**资源表示形式通常反映了在客户端应用程序请求资源时的资源当前状态及其属性。这种意义上的资源表示形式只是时间上的快照。这可

以像数据库中的记录表示形式一样简单，其中包括列名称与 XML 标记之间的映射，XML 中的元素值包含行值。或者，如果系统具有数据模型，那么根据此定义，资源表示形式是系统的数据模型中的对象之一的属性快照。这些对象就是 REST Web Services 为客户端提供的资源。基于 REST 的 Web Services 设计中的最后一组约束与应用程序和服务在请求/响应有效负载或 HTTP 正文中交换的数据的格式有关。这是真正值得将一切保持简单、可读和连接在一起的方面。数据模型中的对象通常以某种方式相关，应该以在将资源传输到客户端应用程序时表示资源的方式，反映数据模型对象（资源）之间的关系。在讨论线程服务中，连接的资源表示形式的示例可能包括根讨论主题及其属性，以及指向为该主题提供的响应的嵌入链接。

2.3.4 JAXB 规范

JAXB（Java Architecture for XML Binding）规范提供了一种非常方便的方法将 XML 模式映射到 Java 代码的表示形式，从而支持数据绑定功能。JAXB 消除了将 SOAP 消息中的 XML 模式消息转换为 Java 代码的工作，因而开发人员不必全面了解 XML 和 SOAP 解析。JAXB 规范定义 Java 和 XML 模式之间的绑定。SAAJ 提供了标准的方法来处理 SOAP 消息中包含的 XML 附件。

JAXB 规范是一个采用 Java 技术，提供了一种简便、快捷的方式将 Java 类转化为 XML 模式的 XML 文档，这样可以简化 Web Services 开发。JAXB 在 Java 应用程序中利用了平台无关的 XML 数据去绑定 XML Schema 到 Java 应用程序，同时不需要开发人员具有丰富的 XML 知识。JAXB 是一种 XML 到 Java 绑定技术，支持 XML Schema 与 Java 对象之间和 XML Schema 与 Java 实例化对象之间的转化。JAXB 由一个运行时应用程序编程接口（API）和辅助工具组成，可以简化访问 XML 文档。使用 JAXB 也有助于建立一个符合和验证某一个 XML 架构（XML Schema）的 XML 文档。JAXB 提供了 xjc 模式编译器工具、schemagen 模式生成器工具，以及运行时框架。可以使用 xjc 模式编译器工具启动一个 XML 架构定义（XSD）来创建一个在 XSD 架构中定义的元素和类型映射成为一系列 JavaBeans 集合。也可以启动一系列 JavaBeans 集合设置并使用 schemagen 模式生成工具创建 XML 架构。一旦 XML 模式和 Java 类之间的存在着映射，运行时通过具有约束力的 JAXB API 的调用，XML 实例文档可以转换或被转换为 Java 实例化对象。在 XML 文档中存储的数据可以被访问，而不需要理解数据结构。最后，开发人员可以使用产生的 Java 类来组装 Web Services 的应用程序。JAXB 注释类包含了所有的运行时 JAXB 处理 XML 实例文档所需的 API 信息。JAXB 运行的 API 支持编组 JAXB 实例化对象到 XML 和反编组 XML 文档返回 JAXB 类实例化对象。另外，可以使用 JAXB 提供执行 XML 验证，这样可以加强基于 XML Schema 定义的传入和传出的 XML 文件符合度检查。JAXB 在 XML Web 服务（JAX-WS 的）2.0 工具中是默认的数据绑定技术。开发者可以在开发 JAX-WS 应用程序使用 JAXB 对象。开发者一方面可以在 Java 应用中平衡 XML 数据绑定技术来操纵 XML，另一方面也可以在 JAX-WS 开发中独立地使用 JAXB。

其中的重要定义如下。

- **解组**：把 XML 数据转化成 Java 类（或者多个类）的过程。
- **编组**：把 Java 数据转化成 XML 文档的过程（恰恰与解组相反）。
- **语义等价**：基于 XML 规则的相等。即使两个文档看起来不同，但在语义上可能是等价的。
- **往返**：从 XML 文档到 Java 代码然后再回到 XML 的整个过程。有效的往返保证输入和输出文档是相同的（语义等价）。

JAXB 开发包下载地址 http://java.sun.com/webservices/downloads/webservicespack.html。

第 3 章　Apache CXF 开发环境介绍

本章内容：

1. Apache CXF 安装包的下载和说明。
2. Apache CXF 框架支撑和运行环境。
3. 搭建 Apache CXF 开发环境。

要学习 Apache CXF 的使用，首先要下载 Apache CXF 安装包。本章说明 Apache CXF 下载的地址以及下载后解压的目录结构。使用 Apache CXF 框架进行应用程序开发，也要有其适合的开发环境。基于 Apache CXF 框架的开发环境支持 Ant、Maven 和 Eclipse 等工具。

3.1　Apache CXF 安装包的下载和说明

笔者使用的 Apache CXF 框架版本是 2.3.0，可以访问 Apache 站点（官方网站：http://cxf.apache.org/index.html）下载 Apache CXF 框架的安装包，下载时请选择“二进制发布包（Binary distribution）”，当然也可以下载相应版本的“源代码发布包（Source distribution）”。

下载完成后，将下载的文件解压缩到任意的文件夹中，如 C:/Java/CXF，在后面的章节中使用 %CXF_HOME% 表示 CXF 框架的存放目录，解压缩后形成的文件目录结构按名称排序，如图 3-1 所示。

图 3-1　Apache CXF 发布包的目录结构示意图

1. bin（目录）

bin 目录中存放的是 CXF 框架中所提供的代码生成、校验、管理控制台工具，具体描述见表 3-1。

表 3-1　Apache CXF提供的bat文件和功能

序号	名称	实现文件	描述
1	Idl to WSDL	idl2wsdl.bat	实现Idl生成WSDL文档
2	Java to JavaScript	java2js.bat	实现Java代码生成JavaScript代码
3	Java to WSDL	java2ws.bat	实现Java代码生成WSDL文档
4	CXF Management Console Tool	mc.bat	管理控制台工具
5	WSDL to Corba	wsdl2corba.bat	实现WSDL文档生成Corba代码
6	WSDL to Java	wsdl2java.bat	实现WSDL文档生成Java代码
7	WSDL to JavaScript	wsdl2js.bat	实现WSDL文档生成JavaScript代码
8	WSDL to Service	wsdl2service.bat	实现WSDL文档生成Service文档
9	WSDL to SOAP	wsdl2soap.bat	实现WSDL文档生成SOAP文档
10	WSDL to XML	wsdl2xml.bat	实现WSDL文档生成XML文档
11	WSDL Validation	wsdlvalidator.bat	WSDL文档验证
12	XSD to WSDL	xsd2wsdl.bat	实现XSD文档生成WSDL文档

2. docs（目录）

包含 Apache CXF 所有类（class）对应的 API 文档，为开发者使用 Apache CXF 框架来完成应用开发提供应有的帮助。

3. etc（目录）

包含一个基础 Service 接口暴露所需要的 web.xml 文件及其他的配置文件。

4. lib（目录）

包含 Apache CXF 及其运行时所需要的和可选的第三方支持类包（.jar 文件），可以根据不同项目所需的 Apache CXF 特性选择所需要的支持类包。如果不想一一区分的话，直接在 Web 项目中包含所有的 Apache CXF 及其运行时所需要的第三方支持类包（.jar 文件）即可。

其中，cxf-2.3.0-incubator.jar 是 Apache CXF 框架的二进制包文件，包含了全部的模块（module）；cxf-manifest-incubator.jar 是列表清单文件。

以下的 jar 包是所有 Apache CXF 项目所必需的：

- cxf.jar
- commons-logging.jar
- geronimo-activation.jar
- geronimo-annotation.jar
- geronimo-javamail.jar
- neethi.jar
- jaxb-api.jar
- jaxb-impl.jar
- stax-api.jar

- XmlSchema.jar
- wstx-asl.jar
- xml-resolver.jar

对于 Java2WSDL 和 WSDL2Java，除了必需的之外，还需要增加如下 jar 包：

- jaxb-xjc.jar
- veliocity.jar
- velocity-dep.jar

为了支持 JAX-WS，除了必需的之外，还需要增加如下 jar 包：

- jaxws-api.jar
- saaj-api.jar
- saaj-impl.jar
- asm.jar （可选的，但是可以提升包装类型的性能）

为了支持 XML 配置，除了必需的之外，还需要增加如下 jar 包：

- aopalliance.jar
- spring-beans.jar
- spring-context.jar
- spring-core.jar
- spring.web.jar

为了独立的 HTTP 服务支持，除了必需的之外，还需要增加如下 jar 包：

- geronimo-servlet.jar
- jetty.jar
- jetty-sslengine.jar
- jetty-util.jar
- sl4j.jar 或 sl4j-jdk14.jar （可选的，使用后可以提升日志功能）

为了支持 Aegis 功能，除了必需的之外，还需要增加如下 jar 包：

- jaxen.jar
- jdom.jar
- stax-utils.jar

为了支持 WS-Security，除了必需的之外，还需要增加如下 jar 包：

- bcprov-jdk14.jar
- wss4j.jar
- xalan.jar
- xmlsec.jar

为了支持 HTTP Binding，除了必需的之外，还需要增加如下 jar 包：

- jra.jar
- jettison.jar （仅为 JSON 服务所需的）

5. licenses（目录）

列出了引用第三方 jar 包的相关许可协议。

6. modules（目录）

modules 目录中包含了 Apache CXF 框架下根据不同特性而分开进行编译的二进制包文件。发布基于 Apache CXF 框架的 Web 项目时，可以选择使用该目录下的所有 .jar 文件，也可以选择 lib 目录中的 cxf-2.0.2-incubator.jar 文件。

7. samples（目录）

samples 目录中包含了所有随 Apache CXF 二进制包发布的示例，包含这些示例的源代码和相关 Web 应用配置文件，可以方便地用 Ant 来编译运行测试这些示例。通过这些源程序示例来了解 Apache CXF 开发和使用的方法，还可以通过 samples 目录和它各个子目录下的 README.txt 的文件来详细了解示例的编译与运行的步骤。

8. LICENSE 文件中包含了 Apache CXF 框架的授权协议 Apache License Version 2.0。

9. NOTICE 罗列了 Apache CXF 框架用到的相关第三方组件的授权协议及其他相关信息。

10. README 文件中包含了 Apache CXF 框架本身的一些简要说明。

11. release_notes.txt 包含了 Apache CXF 发布时的一些信息，包括运行时所需要的环境、修复 BUG 的列表等。

3.2 Apache CXF 框架支撑和运行环境

Apache CXF 框架是一种基于 Servlet 技术的 SOA 应用开发框架，要正常运行基于 Apache CXF 框架的企业应用系统，除了包含 Apache CXF 框架本身之外，还需要 JDK 和 Servlet 容器的支持。

1. JDK 版本选择、下载和安装

Apache CXF 支持非常多的特性，其中不同的特性对 JDK 版本的要求有所不同。JDK 各版本均可在甲骨文公司网站下载。如何安装 JDK 请参考甲骨文公司的相关技术文档和 JDK 的帮助文档。为了运行 Apache CXF 框架本身附带 samples 目录下的所有示例，还需要 Apache Ant 1.6.5 或以上的版本。如果要使用 Apache CXF 框架的 WS-Security 特性，还需要在安装包中添加 Bouncy Castle 组件，可以从网上下载，并增加到类路径中。

2. Servlet 容器下载和安装

Apache CXF 框架是一种基于 Servlet 技术的 SOA 应用开发框架，需要 Servlet 容器的支持。Apache CXF 框架可以在多种 Servlet 容器中运行，包括 WebSphere、WebLogic、Tomcat、Jetty 等。这里选择使用 Tomcat 6.0 和 Jetty 作为 Apache CXF 的运行容器，所有配置过程和发布步骤的说明也均是针对 Tomcat 和 Jetty 的，如果读者使用 Tomcat 之外的其他 Servlet 容器或者选择了 Tomcat 的其他版本，下面的配置过程和步骤可能需要做出调整，请读者根据实际 Servlet 容器的帮助文档进行相应调整。

Tomcat 与 Jetty 的各个版本均可从网上下载，如何正确安装 Tomcat 与 Jetty 服务器请参考相关的帮助文档。

3.3 搭建 Apache CXF 开发环境

要搭建软件开发环境必须下载并在安装之前设置环境。下面阐述了在 Windows 环境中的安装过程，也可以应用到 UNIX 的环境：

Java 5 或更高版本。Apache CXF 需要 JDK 5 或更高版本，可从以下网址下载：http://java.sun.com/j2se/1.5.0/download.jsp。

Tomcat 的 6.0 或更高版本。Tomcat 6.0，可从以下网站下载：http://tomcat.apache.org/download-60.cgi。

Apache Ant 的 1.7.1 或更高版本。Ant 将用于建立和部署代码。Ant 生成工具可从该网站下载：http://ant.apache.org/bindownload.cgi。

Maven 将用于建立和部署代码。生成工具可从该网站下载：http://maven.apache.org/。

Apache CXF 的开发包下载网址：http://cxf.apache.org/download.html。

3.3.1 用 Ant 来创建项目

一旦有了上述软件，就可以着手建立以下环境变量。

使用 Windows 操作系统的用户可以利用控制面板来设置环境变量。从控制面板选择系统属性，然后选择“高级”选项卡。在“高级”选项卡中，单击“环境变量”按钮，设置相应的环境变量。

1. 配置与检查 Ant 环境变量

在 UNIX 操作系统环境下的配置信息如下：

```
CXF_HOME=/<installation_directory>
JAVA_HOME=/<jdk_installation_directory>
ANT_HOME=/<ant_installation_directory>/apache-ant-1.6.5export

PATH=$JAVA_HOME/bin:$ANT_HOME/bin:$CXF_HOME/bin:$PATH
export CLASSPATH=.:$CXF_HOME/lib/cxf-manifest-incubator.jar:./build/classes
```

在 Windows 操作系统环境的配置信息如下：

```
set CXF_HOME=C:\<installation_directory>
set JAVA_HOME=C:\<jdk_installation_directory>
set ANT_HOME=C:\<ant_installation_directory>\apache-ant-1.6.5

set PATH=%JAVA_HOME%\bin;%ANT_HOME%\bin;%CXF_HOME%\bin;%PATH%
set CLASSPATH=.;%CXF_HOME%\lib\cxf-manifest.jar;.\build\classes
```

可以部署到 Java EE 应用服务器容器中进行测试，本示例采用 Tomcat 来进行，可以设置其 CATALINA_HOME 变量。

在 UNIX 操作系统环境下的配置信息如下：

```
CATALINA_HOME=/opt/OpenSource/Tomcat-5.5.25
export CATALINA_HOME
```

在 Windows 操作系统环境下的配置信息如下：

```
set CATALINA_HOME=C:\OpenSource\Tomcat-5.5.25
```

如果以上环境变量没有直接设置到系统的变量中，可以将其另存为一个文件，如存为：%CXF_HOME%/env.sh 或者 %CXF_HOME%/env.bat，这样在运行示例前，可以通过先运行配置文件初始化来设置环境变量。环境设置可以自动使用批处理脚本。该脚本内容如下：

```
@echo off
rem ---------------------------------------------------------------
rem CXF Environment Setup script
rem ---------------------------------------------------------------
set JAVA_HOME=C:\jdk1.5.0_12
set CATALINA_HOME=C:\Program Files\Tomcat 6.0
set ANT_HOME=C:\apache-ant-1.7.1
set CXF_HOME=C:\apache-cxf-2.2.3
set PATH=%PATH%;%JAVA_HOME%\bin;%CATALINA_HOME%\bin;%ANT_HOME%\bin;%CXF_HOME%\bin
rem ---------------------------------------------------------------
```

2. 通过 Ant 来构建和运行示例

下面采用系统自带的 java_first_pojo 例子来说明，在 Apache CXF 框架程序包的 samples 目录下，其 Ant 的 build.xml 内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="java first pojo demo" default="build" basedir=".">
   <import file="../common_build.xml"/>

   <target name="client" description="run demo client" depends="build">
      <property name="param" value=""/>
      <cxfrun classname="demo.hw.client.Client"/>
   </target>

   <target name="server" description="run demo server" depends="build">
      <cxfrun classname="demo.hw.server.Server"/>
   </target>

   <property name="cxf.war.file.name" value="helloworld"/>
   <target name="war" depends="build">
    <cxfwar filename="${cxf.war.file.name}.war"/>
   </target>

   <target name="client-servlet" description="run demo client hitting servlet"
depends="build">
      <property name="param" value=""/>
      <cxfrun classname="demo.hw.client.Client"
param1="${base.url}/helloworld/services/hello_world"
param2="${op}"
param3="${param}"/>
   </target>
</project>
```

（1）构建与启动 Server

首先，进入到 %CXF_HOME%/samples/java_first_pojo/ 目录下。在 UNIX 或 Windows 下的命令是一样的，如下所示：

```
ant server
```

可以看到如图 3-2 所示的信息。

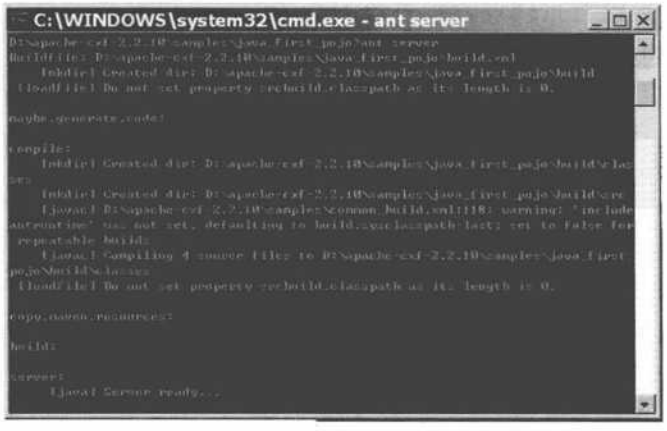


图 3-2　Server 启动完成信息

如果看不到这些信息，请检查环境变量设置是否正确。

（2）运行 Client

再打开一个命令窗口，运行：

```
ant client
```

最后可以看到如图 3-3 所示的信息。

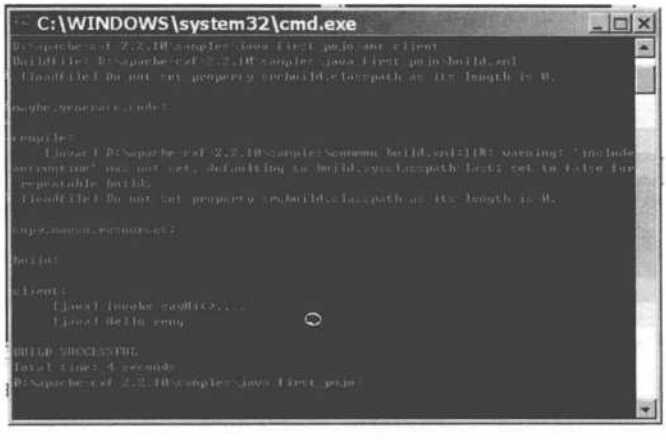


图 3-3　Client 生成的信息

如果看不到这些信息，需要再次检查环境变量设置是否正确。

读者可以在相关目录下看到形成了一个 build 目录。如果要删除已经生成的 build 目录，可以运行“ant clean”。

3.3.2　用 Maven 来创建项目

Maven 是基于项目对象模型（POM），可以通过 XML 描述信息来管理项目的构建、报告和文档的软件项目管理工具。Maven 包含了一个项目对象模型（Project Object Model）、一组标准

集合、一个项目生命周期（Project Lifecycle）、一个依赖管理系统（Dependency Management System），以及用来运行定义在生命周期阶段（phase）中插件（plugin）目标（goal）的逻辑。

在使用 Maven 时，可以用一个明确定义的项目对象模型来描述软件项目，Maven 可以应用横切的逻辑，这些逻辑来自一组共享的（或者自定义的）插件。此外，Maven 能够很方便地管理项目报告、生成站点、管理 JAR 文件，等等。

Maven 项目的主页地址为：http://maven.apache.org/。

Apache CXF 框架还支持基于 Maven 的构建和安装。Apache CXF 的安装包可以从 Maven 的中央资料库获取。

1. 配置与检查 Maven 环境变量

在 UNIX 操作系统下 Maven 配置信息如下：

```
CXF_HOME=/<installation_directory>
JAVA_HOME=/<jdk_installation_directory>
MAVEN_HOME=/<maven_installation_directory>\bin

export PATH=$JAVA_HOME/bin:$MAVEN_HOME /bin:$CXF_HOME/bin:$PATH
export CLASSPATH=.:$CXF_HOME/lib/cxf-manifest-incubator.jar:./build/classes
```

在 Windows 操作系统下 Maven 配置信息如下：

```
set CXF_HOME=/<installation_directory>
set JAVA_HOME=/<jdk_installation_directory>
set MAVEN_HOME=/<maven_installation_directory>\bin

set PATH=%JAVA_HOME%\bin;%MAVEN_HOME %\bin;%CXF_HOME%\bin;%PATH%
set CLASSPATH=.;%CXF_HOME%\lib\cxf-manifest-incubator.jar;.\build\classes
```

2. 通过 Maven 来构建和运行示例

下面采用 Apache CXF 框架系统自带的标准案例 java_first_pojo 来说明。在该目录下，Maven 的 pom.xml 内容如下：

```
<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">
   <modelVersion>4.0.0</modelVersion>
   <groupId>org.apache.cxf.samples</groupId>
   <artifactId>java_first_pojo</artifactId>
   <version>1.0</version>
   <name>Java First POJO Sample</name>
   <properties>
      <cxf.version>[2,)</cxf.version>
   </properties>
   <build>
      <sourceDirectory>src</sourceDirectory>
      <plugins>
         <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
               <source>1.5</source>
```

```
                <target>1.5</target>
            </configuration>
        </plugin>
    </plugins>
</build>
<profiles>
    <profile>
        <id>server</id>
        <build>
            <defaultGoal>test</defaultGoal>
            <plugins>
                <plugin>
                    <groupId>org.codehaus.mojo</groupId>
                    <artifactId>exec-maven-plugin</artifactId>
                    <executions>
                        <execution>
                            <phase>test</phase>
                            <goals>
                                <goal>java</goal>
                            </goals>
                            <configuration>
                                <mainClass>demo.hw.server.Server</mainClass>
                            </configuration>
                        </execution>
                    </executions>
                </plugin>
            </plugins>
        </build>
    </profile>
    <profile>
        <id>client</id>
        <build>
            <defaultGoal>test</defaultGoal>
            <plugins>
                <plugin>
                    <groupId>org.codehaus.mojo</groupId>
                    <artifactId>exec-maven-plugin</artifactId>
                    <executions>
                        <execution>
                            <phase>test</phase>
                            <goals>
                                <goal>java</goal>
                            </goals>
                            <configuration>
                                <mainClass>demo.hw.client.Client</mainClass>
                            </configuration>
                        </execution>
                    </executions>
                </plugin>
            </plugins>
        </build>
    </profile>
    <profile>
        <id>snapshots</id>
        <repositories>
```

```
            <repository>
                <id>apache-snapshots</id>
                <name>Apache SNAPSHOT Repository</name>
                <url>https://repository.apache.org/content/groups/snapshots-group/</url>
                <snapshots>
                    <enabled>true</enabled>
                </snapshots>
                <releases>
                    <enabled>false</enabled>
                </releases>
            </repository>
        </repositories>
        <pluginRepositories>
            <pluginRepository>
                <id>apache-snapshots</id>
                <name>Apache Maven Plugin Snapshots</name>
                <url>https://repository.apache.org/content/groups/snapshots-group/</url>
                <releases>
                    <enabled>false</enabled>
                </releases>
                <snapshots>
                    <enabled>true</enabled>
                </snapshots>
            </pluginRepository>
        </pluginRepositories>
    </profile>
  </profiles>
  <dependencies>
    <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-rt-frontend-jaxws</artifactId>
        <version>${cxf.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-rt-transports-http</artifactId>
        <version>${cxf.version}</version>
    </dependency>
    <!-- Jetty is needed if you're using the CXFServlet -->
    <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-rt-transports-http-jetty</artifactId>
        <version>${cxf.version}</version>
    </dependency>
  </dependencies>
</project>
```

（1）构建程序

首先，进入%CXF_HOME%/java_first_pojo/目录。在 UNIX 或 Windows 下的命令是一样的，如下：

```
mvn install
```

可以看到如图 3-4 所示的信息。

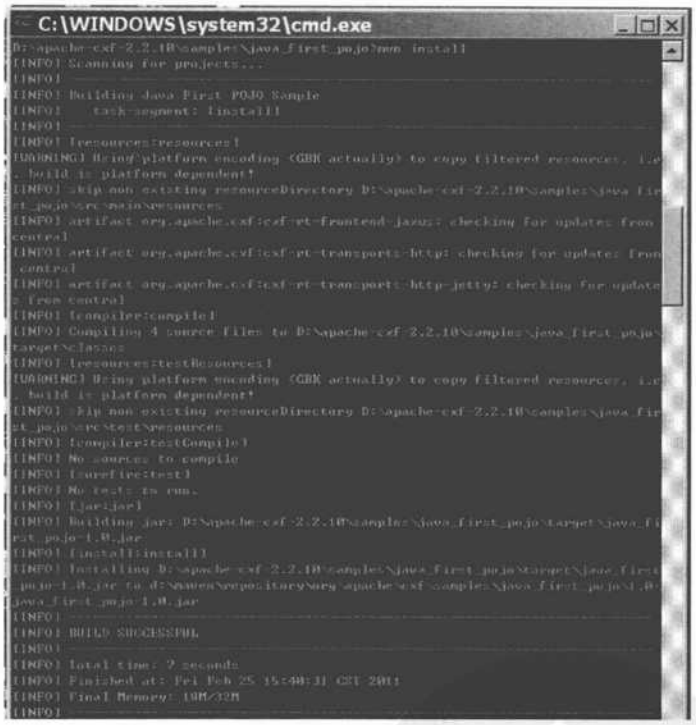


图 3-4　执行“mvn install”命令形成的反馈信息

当出现“BUILD SUCCESSFUL”提示时，说明创建成功。

（2）构建与启动 Server

运行服务器端程序：

```
mvn -Pserver
```

可以看到如图 3-5 所示的信息。

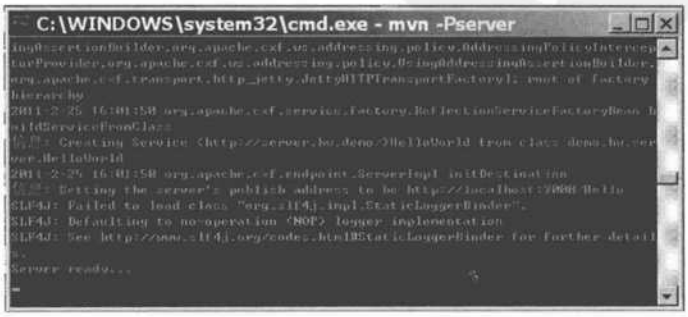


图 3-5　运行服务器形成的反馈信息

如果看不到这些信息，请检查环境变量设置是否正确。

（3）运行 Client

再打开一个命令窗口，运行：

```
mvn -Pclient
```

可以看到如图 3-6 所示的信息。

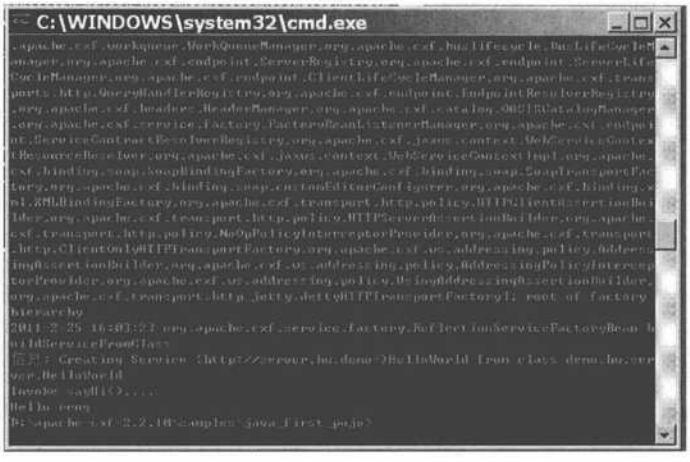


图 3-6　运行客户端程序形成的反馈信息

如果看不到这些信息，再次检查环境变量设置是否正确。

也可以在相关目录下看到形成了一个 target 目录。如果要删除已经生成的 build 目录，可以运行“mvn clean”。

3.3.3　用 Eclipse 集成 Apache CXF

后面章节的例子全部采用 Eclipse 集成 Apache CXF，因此在这里就不做专门介绍了。

第4章 简单的 Apache CXF 例子

本章内容：

1. 简单的 JAX-WS 服务程序。
2. 用 Spring 来创建 Web Services 服务程序。
3. Servlet 容器中的 Web Services 实现。

4.1 一个简单的 JAX-WS 服务程序

先编写一个比较简单的 JAX-WS 服务程序。这个例子采用 JAX-WS 内置的服务器进行发布，调用方式为前端应用（Frontends）模式。要创建这个应用，需要下面四个步骤：

（1）创建一个 JAX-WS 服务项目；
（2）编写一个对外发布的 Services 代码；
（3）对外发布 Web Services；
（4）客户端调用 Web Services。

过程实现如图 4-1 所示。

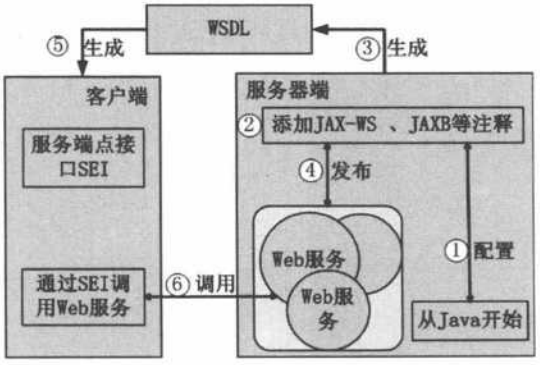


图 4-1 过程实现图

1. 创建一个 JAX-WS 服务项目

打开 Eclipse IDE 并创建一个新的 Java 项目。在 Eclipse 中选择菜单命令 File → Java Project。

创建名为“hello_world_jax_ws”的项目。

首先设置项目的 Text file encoding（字符集）为 UTF-8。

其次需要做的是把 Apache CXF 依赖的库添加到项目中：

commons-logging-1.1.1.jar
geronimo-activation_1.1_spec-1.0.2.jar (or Sun's Activation jar)
geronimo-annotation_1.0_spec-1.1.1.jar (JSR 250)
geronimo-javamail_1.4_spec-1.6.jar (or Sun's JavaMail jar)
geronimo-servlet_2.5_spec-1.2.jar (or Sun's Servlet jar)
geronimo-ws-metadata_2.0_spec-1.1.2.jar (JSR 181)
geronimo-jaxws_2.1_spec-1.0.jar (or Sun's jaxws-api-2.1.jar)
geronimo-stax-api_1.0_spec-1.0.1.jar (or other stax-api jar)
jaxb-api-2.1.jar
jaxb-impl-2.1.12.jar
jetty-6.1.21.jar
jetty-util-6.1.21.jar
neethi-2.0.4.jar
saaj-api-1.3.jar
saaj-impl-1.3.2.jar
wsdl4j-1.6.2.jar
wstx-asl-3.2.8.jar
XmlSchema-1.4.5.jar
xml-resolver-1.2.jar

Spring 的依赖库如下（这项可选）：

aopalliance-1.0.jar
spring-core-2.5.5.jar
spring-beans-2.5.5.jar
spring-context-2.5.5.jar
spring-web-2.5.5.jar

Apache CXF 包括 CXF jar：

cxf-2.2.3.jar

2. 编写服务端程序

编写一个 SEI 服务接口，定义方法 sayHello，代码如下：

```
package demo.cxf.helloworld;

import javax.jws.WebParam;
import javax.jws.WebService;
```

```
@WebService
public interface HelloWorld {
    String sayHello(@WebParam(name="text") String text);
}
```

@WebService 和@WebParam 注释这两项是必需的。

@WebService 注释说明这是一个 Web Services。@WebParam 注释说明这个参数的名称，因为 Java 接口不存储类文件的参数名称。因此，如果省略掉参数名称，系统将被命名为 arg0。

服务的实现代码如下：

```
package demo.cxf.helloworld;

import javax.jws.WebParam;
import javax.jws.WebService;

@WebService(serviceName = "HelloWorld")
public class HelloWorldImpl implements HelloWorld {

    public String sayHello(@WebParam(name="text")String text) {
       System.out.println("sayHello called");
       return "Hello " + text +",This is world.";
    }
}
```

该实现类@WebService 注释让 Apache CXF 知道开发者使用哪个接口来创建 WSDL 内容。在这种情况下简化了 HelloWorld 接口的写法。

3. 发布服务

可以通过下面的方式来发布 Web Services 服务，代码如下：

```
package demo.cxf.helloworld.server;

import javax.xml.ws.Endpoint;

import demo.cxf.helloworld.HelloWorld;
import demo.cxf.helloworld.HelloWorldImpl;

public class Server {

    public Server() {
        System.out.println("Starting Server");
        HelloWorld helloWorld = new HelloWorldImpl();
        String address = "http://localhost:9000/HelloWorld";
        Endpoint.publish(address, helloWorld);
    }

    public static void main(String[] args) throws Exception {
        new Server();
        System.out.println("Server ready...");

        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0); }

}
```

在代码实现中，客户端类和服务器类最好都实现同一个接口，这样便于管理，也方便发布。

在 IE 或其他浏览器中输入 http://localhost:9000/HelloWorld?wsdl ，即可看到如下内容，说明服务发布成功。

```
<?xml version='1.0' encoding='UTF-8'?>
<wsdl:definitions name="HelloWorld"
    targetNamespace="http://helloworld.cxf.demo/"
    xmlns:ns1="http://schemas.xmlsoap.org/soap/http"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://helloworld.cxf.demo/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <wsdl:types>
        <xs:schema elementFormDefault="unqualified"
            targetNamespace="http://helloworld.cxf.demo/" version="1.0"
            xmlns:tns="http://helloworld.cxf.demo/"
            xmlns:xs="http://www.w3.org/2001/XMLSchema">
            <xs:element name="sayHello" type="tns:sayHello" />
            <xs:element name="sayHelloResponse"
                type="tns:sayHelloResponse" />
            <xs:complexType name="sayHello">
                <xs:sequence>
                    <xs:element minOccurs="0" name="text"
                        type="xs:string" />
                </xs:sequence>
            </xs:complexType>
            <xs:complexType name="sayHelloResponse">
                <xs:sequence>
                    <xs:element minOccurs="0" name="return"
                        type="xs:string" />
                </xs:sequence>
            </xs:complexType>
        </xs:schema>
    </wsdl:types>
    <wsdl:message name="sayHelloResponse">
        <wsdl:part element="tns:sayHelloResponse" name="parameters">
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="sayHello">
        <wsdl:part element="tns:sayHello" name="parameters"></wsdl:part>
    </wsdl:message>
    <wsdl:portType name="HelloWorld">
        <wsdl:operation name="sayHello">
            <wsdl:input message="tns:sayHello" name="sayHello">
            </wsdl:input>
            <wsdl:output message="tns:sayHelloResponse"
                name="sayHelloResponse">
            </wsdl:output>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="HelloWorldSoapBinding" type="tns:HelloWorld">
        <soap:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http" />
        <wsdl:operation name="sayHello">
            <soap:operation soapAction="" style="document" />
            <wsdl:input name="sayHello">
                <soap:body use="literal" />
```

```
            </wsdl:input>
            <wsdl:output name="sayHelloResponse">
                <soap:body use="literal" />
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="HelloWorld">
        <wsdl:port binding="tns:HelloWorldSoapBinding"
            name="HelloWorldImplPort">
            <soap:address location="http://localhost:9000/HelloWorld" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

4. 访问和调用 Web Services 服务

客户端有两种方式来访问服务，第一种代码方式，采用 JAX-WS 规范，其实现代码如下：

```
package demo.cxf.helloworld.client;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.soap.SOAPBinding;

import demo.cxf.helloworld.HelloWorld;

public class Client {

    //定义服务的服务名称和服务端口
    private static final QName SERVICE_NAME = new QName("http://helloworld.cxf.demo/",
"HelloWorld");
   private static final QName PORT_NAME = new QName("http://helloworld.cxf.demo/",
"HelloWorldPort");

    public Client() {   }

    public static void main(String[] args) throws Exception{
        System.out.println("Starting Client");
        //用服务名称创建一个服务
        Service service = Service.create(SERVICE_NAME);

        // 定义 Endpoint 地址的内容
       String endpointAddress = "http://localhost:9000/HelloWorld";

        //在服务中增加端口
        service.addPort(PORT_NAME, SOAPBinding.SOAP11HTTP_BINDING, endpointAddress);

        //获取服务类的对象并处理
        HelloWorld helloWorld = service.getPort(HelloWorld.class);
        System.out.println(helloWorld.sayHello("Rodger "));
        System.exit(0);
    }
}
```

上述代码中，SERVICE_NAME 和 PORT_NAME 的 Qname 类型应定义为“http://helloworld.cxf.demo/”，原因在后面的章节中会详细说明。

调用成功提示如下：

```
Starting Client
Hello Rodger ,This is world.
```

第一种方式可以采用 Apache CXF 的实现模式，代码相对较简单，如下所示：

```
package demo.cxf.helloworld.client;

import org.apache.cxf.jaxws.JaxWsProxyFactoryBean;
import demo.cxf.helloworld.HelloWorld;

public class CXFClient {
    public CXFClient() {}

    public static void main(String[] args) {
        System.out.println("Starting Client");

        //定义 JaxWs 的代理工厂
        JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();

        //设置 JaxWs 代理工厂的属性，包括服务类和地址
        factory.setServiceClass(HelloWorld.class);
        factory.setAddress("http://localhost:9000/HelloWorld");

        //获取服务类的对象并处理
        HelloWorld helloWorld = (HelloWorld) factory.create();

        System.out.println(helloWorld.sayHello("Rodger "));
        System.exit(0);
    }
}
```

调用成功的提示与第一种方式相同。

本例代码请参见配套资源代码中“第 04 章”文件夹下的“hello_world_jax_ws”文件夹。

4.2　利用 Spring 创建 Web Services 服务程序

采用 Spring 配置，再编写一个比较简单的 JAX-WS 服务的示例程序。与上一个例子相同，这个例子采用 JAX-WS 内置的服务器进行发布，调用方式为前端应用（Frontends）模式。要创建这个应用，需要经过下面五个步骤：

（1）创建一个 JAX-WS 服务项目；

（2）编写一个对外发布的 Services 代码；

（3）定义服务器发布程序和配置文件；

（4）对外发布 Web Services；

（5）客户端调用 Web Services。

1. 创建一个 JAX-WS 服务项目

打开 Eclipse IDE 并创建一个新的 Java 项目。在 Eclipse 中选择菜单命令 File → Java Project，创建名称为“hello_world_spring”的项目。

首先设置项目的 Text file encoding（字符集）为 UTF-8。

其次需要做的是把 Apache CXF 必须依赖的库添加到项目中。

```
commons-logging-1.1.1.jar
geronimo-activation_1.1_spec-1.0.2.jar (or Sun's Activation jar)
geronimo-annotation_1.0_spec-1.1.1.jar (JSR 250)
geronimo-javamail_1.4_spec-1.6.jar (or Sun's JavaMail jar)
geronimo-servlet_2.5_spec-1.2.jar (or Sun's Servlet jar)
geronimo-ws-metadata_2.0_spec-1.1.2.jar (JSR 181)
geronimo-jaxws_2.1_spec-1.0.jar (or Sun's jaxws-api-2.1.jar)
geronimo-stax-api_1.0_spec-1.0.1.jar (or other stax-api jar)
jaxb-api-2.1.jar
jaxb-impl-2.1.12.jar
jetty-6.1.21.jar
jetty-util-6.1.21.jar
neethi-2.0.4.jar
saaj-api-1.3.jar
saaj-impl-1.3.2.jar
wsdl4j-1.6.2.jar
wstx-asl-3.2.8.jar
XmlSchema-1.4.5.jar
xml-resolver-1.2.jar
```

Spring 的依赖库如下（这项必需）：

aopalliance-1.0.jar

spring-core-2.5.5.jar

spring-beans-2.5.5.jar

spring-context-2.5.5.jar

spring-web-2.5.5.jar

Apache CXF 包括 CXF jar：

cxf-2.2.3.jar

2. 编写服务端程序

首先要写一个 SEI 服务接口，定义方法 sayHello，代码如下：

```
package demo.cxf.helloworld;

import javax.jws.WebParam;
import javax.jws.WebService;

@WebService
public interface HelloWorld {
    String sayHello(@WebParam(name="text") String text);
}
```

@WebService 和@WebParam 注释都是必需的。

@WebService 注释说明这是一个 Web Services。@WebParam 注释说明这个参数的名称，因为 Java 接口不存储类文件的参数名称，因此，如果省略掉参数名称，系统将被命名为 arg0。

服务的实现代码如下：

```
package demo.cxf.helloworld;

import javax.jws.WebParam;
import javax.jws.WebService;

@WebService(serviceName = "HelloWorld")
public class HelloWorldImpl implements HelloWorld {

    public String sayHello(@WebParam(name="text")String text) {
       System.out.println("sayHello called");
       return "Hello " + text +",This is world.";
    }
}
```

该实现类@WebService 注释让 Apache CXF 知道开发者用哪个接口来创建 WSDL 内容。这种情况下简化了 HelloWorld 接口的写法。

3. 定义服务器发布配置文件并发布服务

可以通过下面的方式来发布 Web Services 服务，代码如下：

```
package demo.cxf.helloworld.server;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Server {
    public Server() {
        System.out.println("Starting Server");
        new ClassPathXmlApplicationContext( new  String[]  {  "/demo/cxf/helloworld/
server/server-beans.xml" });
    }

    public static void main(String[] args) throws Exception {
        new Server();
        System.out.println("Server ready...");

        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

引用的 bean 配置文件——server-beans.xml，代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://cxf.apache.org/jaxws
       http://cxf.apache.org/schemas/jaxws.xsd">

    <jaxws:endpoint id="ProjectManager"
        implementor="demo.cxf.helloworld.HelloWorldImpl"
```

```
        address="http://localhost:9000/HelloWorld" />
</beans>
```

在代码实现中，客户端类和服务器类最好都实现同一个接口。这样便于管理，也方便发布。

在 IE 或其他浏览器上输入 http://localhost:9000/HelloWorld?wsdl ，即可见到如下界面，这说明服务发布成功。

```
<?xml version='1.0' encoding='UTF-8'?>
<wsdl:definitions name="HelloWorld"
    targetNamespace="http://helloworld.cxf.demo/"
    xmlns:ns1="http://schemas.xmlsoap.org/soap/http"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://helloworld.cxf.demo/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <wsdl:types>
        <xs:schema elementFormDefault="unqualified"
            targetNamespace="http://helloworld.cxf.demo/" version="1.0"
            xmlns:tns="http://helloworld.cxf.demo/"
            xmlns:xs="http://www.w3.org/2001/XMLSchema">
            <xs:element name="sayHello" type="tns:sayHello" />
            <xs:element name="sayHelloResponse"
                type="tns:sayHelloResponse" />
            <xs:complexType name="sayHello">
                <xs:sequence>
                    <xs:element minOccurs="0" name="text"
                        type="xs:string" />
                </xs:sequence>
            </xs:complexType>
            <xs:complexType name="sayHelloResponse">
                <xs:sequence>
                    <xs:element minOccurs="0" name="return"
                        type="xs:string" />
                </xs:sequence>
            </xs:complexType>
        </xs:schema>
    </wsdl:types>
    <wsdl:message name="sayHelloResponse">
        <wsdl:part element="tns:sayHelloResponse" name="parameters">
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="sayHello">
        <wsdl:part element="tns:sayHello" name="parameters"></wsdl:part>
    </wsdl:message>
    <wsdl:portType name="HelloWorld">
        <wsdl:operation name="sayHello">
            <wsdl:input message="tns:sayHello" name="sayHello">
            </wsdl:input>
            <wsdl:output message="tns:sayHelloResponse"
                name="sayHelloResponse">
            </wsdl:output>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="HelloWorldSoapBinding" type="tns:HelloWorld">
        <soap:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http" />
        <wsdl:operation name="sayHello">
            <soap:operation soapAction="" style="document" />
```

```
            <wsdl:input name="sayHello">
                <soap:body use="literal" />
            </wsdl:input>
            <wsdl:output name="sayHelloResponse">
                <soap:body use="literal" />
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="HelloWorld">
        <wsdl:port binding="tns:HelloWorldSoapBinding"
            name="HelloWorldImplPort">
            <soap:address location="http://localhost:9000/HelloWorld" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

4. 客户端访问和调用 Web Services 服务

客户端有两种方式访问服务，第一种代码方式采用 JAX-WS 规范，其实现代码如下：

```
package demo.cxf.helloworld.client;

import org.springframework.context.support.ClassPathXmlApplicationContext;
import demo.cxf.helloworld.HelloWorld;

public class Client {

    public Client() {
    }

    public static void main(String[] args) throws Exception {
        System.out.println("Starting Client");

        ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(
                new String[] { "demo/cxf/helloworld/client/client-beans.xml" });

        // 获取服务类的对象并处理
        HelloWorld helloWorld = (HelloWorld) context.getBean("client");
        System.out.println(helloWorld.sayHello("Rodger "));
        System.exit(0);
    }
}
```

引用的 bean 配置文件——client-beans.xml，代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                    http://www.springframework.org/schema/beans/spring-beans.xsd
                    http://cxf.apache.org/jaxws
                    http://cxf.apache.org/schemas/jaxws.xsd">

    <jaxws:client id="client"
        serviceClass="demo.cxf.helloworld.HelloWorld"
        address="http://localhost:9000/HelloWorld" />
</beans>
```

调用成功的标准是出现如下内容：

```
Starting Client
Hello Rodger ,This is world.
```

还有一种方式可以采用 Apache CXF 的实现模式，代码相对较简单，代码如下：

```
package demo.cxf.helloworld.client;

import org.springframework.context.support.ClassPathXmlApplicationContext;
import demo.cxf.helloworld.HelloWorld;

public class CXFClient {
    public CXFClient() {    }

    public static void main(String[] args) {
        System.out.println("Starting Client");

        ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(
                new String[] { "demo/cxf/helloworld/client/CXFClient-beans.xml" });

        // 获取服务类的对象并处理
        HelloWorld helloWorld = (HelloWorld) context.getBean("client");
        System.out.println(helloWorld.sayHello("Rodger "));
    }
}
```

引用的 bean 配置文件——CXFClient-beans.xml，代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://cxf.apache.org/jaxws
        http://cxf.apache.org/schema/jaxws.xsd">

    <bean id="client" class="demo.cxf.helloworld.client.CXFClient"
        factory-bean="clientFactory" factory-method="create" />

    <bean id="clientFactory"
        class="org.apache.cxf.jaxws.JaxWsProxyFactoryBean">
        <property name="serviceClass"
            value="demo.cxf.helloworld.HelloWorld" />
        <property name="address"
            value="http://localhost:9000/HelloWorld" />
    </bean>
</beans>
```

其实现的结果是相同的。

本例代码请参见配套资源代码中“第 4 章”文件夹下的“hello_world_spring”文件夹。

4.3 Servlet 容器中的 Web Services 实现

再编写一个在 Servlet 容器的例子程序，要创建这个应用，需要下面四个步骤：

（1）创建一个 JAX-WS 服务的 Web 项目。

（2）编写一个对外发布的 Services 代码。

（3）通过 Servlet 容器来发布 Web Services。

（4）客户端调用 Web Services。

这个例子相对应的是 Apache CXF 官方配套的 hello_world_code_first 例子。

下面分别详细描述这几个步骤。

1. 创建一个 JAX-WS 服务项目

打开 Eclipse IDE 并创建一个新的 Java 项目。在 Eclipse 中选择菜单项 File → Web Project。创建名称为 hello_world_servlet 的项目。本例子基于 Spring 来实现。采用 Tomcat 服务器来进行发布，调用方式为前端应用（Frontends）模式。

首先设置项目的 Text file encoding（字符集）为 UTF-8。

其次需要做的是把 Apache CXF 必须依存的库添加到项目中，与上例相同。一定要加入 Spring 和 Apache CXF 的依赖库。

2. 编写一个对外发布的 Services 代码

包括 SEI 服务接口和实现类，内容与第二个例子相同，这里就不列出了。

3. 通过 Servlet 容器来发布 Web Services

由于要把服务装载到 Servlet 容器，Servlet 容器采用 Tomcat。所以要创建下面的方式来发布 Web Services 服务，web.xml 文件配置代码如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
   PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>WEB-INF/server-bean.xml</param-value>
    </context-param>

    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>

    <servlet>
        <servlet-name>CXFServlet</servlet-name>
        <display-name>CXF Servlet</display-name>
        <servlet-class>
            org.apache.cxf.transport.servlet.CXFServlet
        </servlet-class>
```

```
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>CXFServlet</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

server-bean.xml 配置文件内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="
          http://cxf.apache.org/transports/http/configuration
          http://cxf.apache.org/schemas/configuration/http-conf.xsd
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd
            http://cxf.apache.org/jaxws
            http://cxf.apache.org/schemas/jaxws.xsd">

    <import resource="classpath:META-INF/cxf/cxf.xml" />
    <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
    <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />

    <jaxws:endpoint id="HelloWorld"
        implementor="demo.cxf.helloworld.HelloWorldImpl"
        address="/HelloWorld" />

</beans>
```

在代码实现中，客户端类和服务器类最好都实现同一个接口。这样便于管理，也方便发布。

把内容发布到 Tomcat 上有两种方式，一种是通过 Eclipse 直接部署到 Tomcat 上，另一种是通过 ant 部署到 Tomcat 上。部署完成后，在 IE 或其他浏览器中输入 http://localhost:18080/hello_world_servlet/HelloWorld?wsdl，即可见到如下界面，这说明服务发布成功。

```
<?xml version='1.0' encoding='UTF-8'?>
<wsdl:definitions name="HelloWorld"
    targetNamespace="http://helloworld.cxf.demo/"
    xmlns:ns1="http://cxf.apache.org/bindings/xformat"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://helloworld.cxf.demo/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <wsdl:types>
        <xs:schema elementFormDefault="unqualified"
            targetNamespace="http://helloworld.cxf.demo/" version="1.0"
            xmlns:tns="http://helloworld.cxf.demo/"
            xmlns:xs="http://www.w3.org/2001/XMLSchema">
            <xs:element name="sayHello" type="tns:sayHello" />
            <xs:element name="sayHelloResponse"
                type="tns:sayHelloResponse" />
            <xs:complexType name="sayHello">
```

```
                <xs:sequence>
                    <xs:element minOccurs="0" name="text"
                        type="xs:string" />
                </xs:sequence>
            </xs:complexType>
            <xs:complexType name="sayHelloResponse">
                <xs:sequence>
                    <xs:element minOccurs="0" name="return"
                        type="xs:string" />
                </xs:sequence>
            </xs:complexType>
        </xs:schema>
    </wsdl:types>
    <wsdl:message name="sayHelloResponse">
        <wsdl:part element="tns:sayHelloResponse" name="parameters">
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="sayHello">
        <wsdl:part element="tns:sayHello" name="parameters"></wsdl:part>
    </wsdl:message>
    <wsdl:portType name="HelloWorld">
        <wsdl:operation name="sayHello">
            <wsdl:input message="tns:sayHello" name="sayHello">
            </wsdl:input>
            <wsdl:output message="tns:sayHelloResponse"
                name="sayHelloResponse">
            </wsdl:output>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="HelloWorldSoapBinding" type="tns:HelloWorld">
        <soap:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http" />
        <wsdl:operation name="sayHello">
            <soap:operation soapAction="" style="document" />
            <wsdl:input name="sayHello">
                <soap:body use="literal" />
            </wsdl:input>
            <wsdl:output name="sayHelloResponse">
                <soap:body use="literal" />
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="HelloWorld">
        <wsdl:port binding="tns:HelloWorldSoapBinding"
            name="HelloWorldImplPort">
            <soap:address
                location="http://localhost:18080/hello_world_servlet/HelloWorld" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

注意：笔者的 Tomcat 采用的是 18080 端口。

4. 客户端访问和调用 Web Services 服务

客户端采用 Spring 集成来访问服务，其实现代码如下：

```
package demo.cxf.helloworld.client;
```

```
import org.springframework.context.support.ClassPathXmlApplicationContext;
import demo.cxf.helloworld.HelloWorld;

public class Client {

    public Client() {}

    public static void main(String[] args) throws Exception{
        System.out.println("Starting Client");

        //获得 Spring 的环境上下文
        ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(
                new String[] {"demo/cxf/helloworld/client/client-bean.xml"});

        //获得服务的 Bean 并进行处理
       HelloWorld helloWorld = (HelloWorld)context.getBean("HelloWorldClient");
       System.out.println(helloWorld.sayHello("Rodger "));
    }

}
```

配置文件 client-bean.xml 的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

    <jaxws:client id="HelloWorldClient"
serviceClass="demo.cxf.helloworld.HelloWorld"
address="http://localhost:18080/hello_world_servlet/HelloWorld" />

</beans>
```

调用成功的标准是出现如下内容：

```
Starting Client
Hello Rodger ,This is world.
```

本例代码参见配套资源代码中“第 4 章”文件夹下的“hello_world_servlet”文件夹。

第 5 章　Apache CXF 的架构体系和基础

学习 Apache CXF 的基本应用，可以不必了解 Apache CXF 的核心架构。但是对于有深度的开发要求，就必须要掌握 Apache CXF 的核心架构。Apache CXF 的核心基于 Bus 结构，同时引入了多种实现方式，如消息、拦截器、服务模型等，这样也能提升框架本身的开放性、可扩展性和可适应性。

本章内容：

1. Apache CXF 的核心架构。Apache CXF 核心架构是以 Bus 为中心，整合前端编程模型（Frontend）组件、消息（Messaging）、拦截器（Interceptors）组件、服务模型（Service Model）组件、数据绑定（Data Bindings）组件、传输协议（Transport）组件等为一个组合体，并实现动态地加载和统一其他组件的功能。
2. 分别对 Bus 组件、消息（Messaging）和拦截器（Interceptors）组件、前端编程模型（Frontend）组件、服务模型（Service Model）组件、数据绑定（Data Bindings）组件、WSDL 绑定（WSDL Bindings）组件、传输协议（Transport）组件和 Apache CXF 注释组件等进行说明。

5.1　Apache CXF 的核心架构

Apache CXF 主要是由下列组件构成：

- 总线（Bus）组件
- 前端编程模型（Frontend）组件
- 消息（Messaging）和拦截器（Interceptors）组件
- 服务模型（Service Model）组件
- 数据绑定（Data Bindings）组件
- WSDL 绑定（WSDL Bindings）组件
- 传输协议（Transport）组件
- Apache CXF 注释组件

其架构组件如图 5-1 所示。

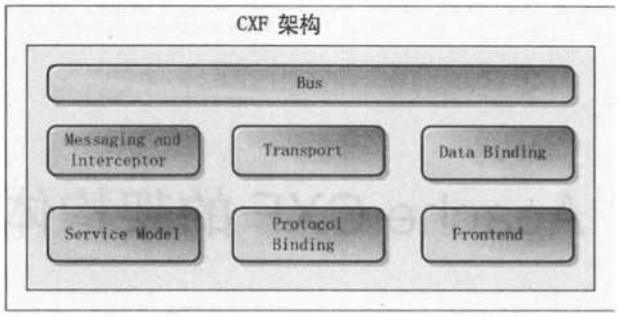


图 5-1　Apache CXF 架构组件图

表 5-1 分别描述各组件的作用。

表 5-1　Apache CXF的组件和描述

序号	组件名称	描述
1	总线（Bus）组件	Bus是CXF架构的主干，为共享资源提供了一个可配置的场所，作用类似于Spring的ApplicationContext。这些共享资源包括WSDL管理器、绑定工厂等。通过对Bus进行扩展，可以方便地容纳自己的资源，或替换现有的资源。默认Bus实现基于Spring架构。通过依赖注入，在运行时将组件串联起来。BusFactory负责Bus的创建。默认的BusFactory是SpringBusFactory，对应于默认Bus实现。在构造过程中，SpringBusFactory会搜索META-INF/cxf（包含在 CXF的Jar中）下的所有bean配置文件，根据这些配置文件构建一个ApplicationContext。开发者也可提供自己的配置文件来定制Bus
2	前端编程模型（Frontend）	前端编程为CXF提供创建服务的编程模型，当前主要的前端就是JAX-WS
3	消息（Messaging）和拦截器（Interceptors）	CXF建立在一个通用的消息层之上，主要是由消息、拦截器和拦截器链（InterceptorChain）组成。CXF以消息处理为中心的，熟悉 JSP/Servlet的开发者可以将拦截器视为CXF架构中的“Filter”，拦截器链也与“FilterChain”类似。通过拦截器，开发者可以方便地在消息传递、处理的整个过程中对CXF进行扩展。拦截器的方法主要有两个：handleMessage和handleFault，分别对应消息处理和错误处理。在开发拦截器的时候需要注意两点：拦截器不是线程安全的，不建议在拦截器中定义实例变量并使用它（这一点跟JSP/Servlet中对于Filter的处理是一样的）；不要调用下一个拦截器的handleMessage或handleFault，这个工作由InterceptorChain来完成
4	服务模型（Service Model）	CXF的服务通过服务模型来表示。它主要有两部分：ServiceInfo和服务本身。ServiceInfo作用类似WSDL，包含接口信息、绑定、端点（EndPoint）等信息；服务则包含了ServiceInfo、数据绑定、拦截器和服务属性等信息。可使用Java类和WSDL来创建服务。一般是由前端负责服务的创建，它通过ServiceFactory来完成
5	绑定（Bindings）	绑定提供了在传输过程中映射的具体格式和协议方法，主要的两个类是Binding和BindingFactory。BindingFactory负责创建Binding
6	传输协议（Transport）	为了在开发过程中对绑定和前端屏蔽内部的传输细节，CXF提供了自己的传输协议抽象。其中主要有两个对象：Conduit和Destination。前者是消息发送的基础，后者则对应消息接收。开发者还可以给Conduit和Destination注册MessageObserver，以便在消息发送和接收时获得通知

5.2　Bus 介绍

Bus（总线）是 Apache CXF 最基本的也是最重要的组件之一。Apache CXF 对外提供的功能和资源都是直接或间接通过调用 Bus（总线）来实现。Bus 是 Apache CXF 资源的管理者，也是调度者。通过 Bus 的调度和关联，开发人员可以在 Apache CXF 运行环境中共享资源。这种共享资源包括：WSDL 管理、数据绑定管理等。同时对于开发人员，它是一个不可视的后台指挥部。开发人员调用 Apache CXF 框架进行业务处理，最终都转化到对 Bus 的调用。用形象的比喻，我们用电脑，一般都知道有内存、CPU、硬盘、显示器、键盘和网卡，等等，而对于主板和数据总线，却很少了解。如果说 Apache CXF 是一台电脑的话，那么 Bus 就类似于电脑的主板和数据总线。但是 Bus 总线不是一个静态的组件，它可以根据开发人员的特殊需求和环境条件很容易地进行扩展。这个道理与电脑中的主板类似。我们知道，主板的一个特点就是采用了开放式结构。主板上有很多个插槽和扩展插槽，供计算机外围设备的控制卡（适配器）插接。这样可以集成多个厂家的各种类型插件。Bus 也是通过各种接口和外挂去支持多种数据绑定、实现多种协议交换、支持多种协议的。所以说，Bus（总线）是 Apache CXF 的核心和基础，在整个 CXF 框架中扮演着举足轻重的角色。

Bus 支持的组件如图 5-2 所示，其最重要的部分是 Bus 核心组件，包括 Bus 接口、BusFactory 抽象类、BusException 类和基于 Bus 处理的通用组件。

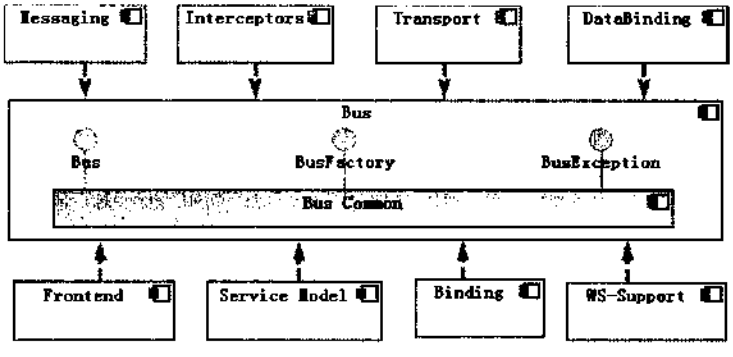


图 5-2　Bus 组件结构图

Apache CXF 使用 Spring 的依赖注入来实现配置，在运行时装载 Spring 的配置文件并把各个组件关联组合到一起。默认 Bus 实现基于 Spring 模式，采用插件方式来装配其他组件，如 Messaging 组件、Interceptors 组件、Transport 组件、Data Bindings 组件、Frontend 组件、Service Model 组件、Binding 组件和 WS-Support 组件等。其中，Interceptor 组件与 Bus 核心组件关系更加密切，主要是 Bus 就是基于拦截器原理来实现。关于这些组件在后面还会详细进行描述和说明。

在应用程序的类路径：META-INF/cxf 目录下有 bean 配置文件，在应用程序中 bean 的配置文件内容包括：

- META-INF/cxf/cxf.xml（仅仅配置 CXF 的核心内容）
- META-INF/cxf/cxf-extension.xml（配置 CXF 的扩展内容，如 cxf-extension-corba、cxf-extension-jaxws、cxf-extension-jms 等）

- META-INF/cxf/cxf-property-editors.xml（如 cxf-rt-transports-http）。

Bus 的类结构如图 5-3 所示。

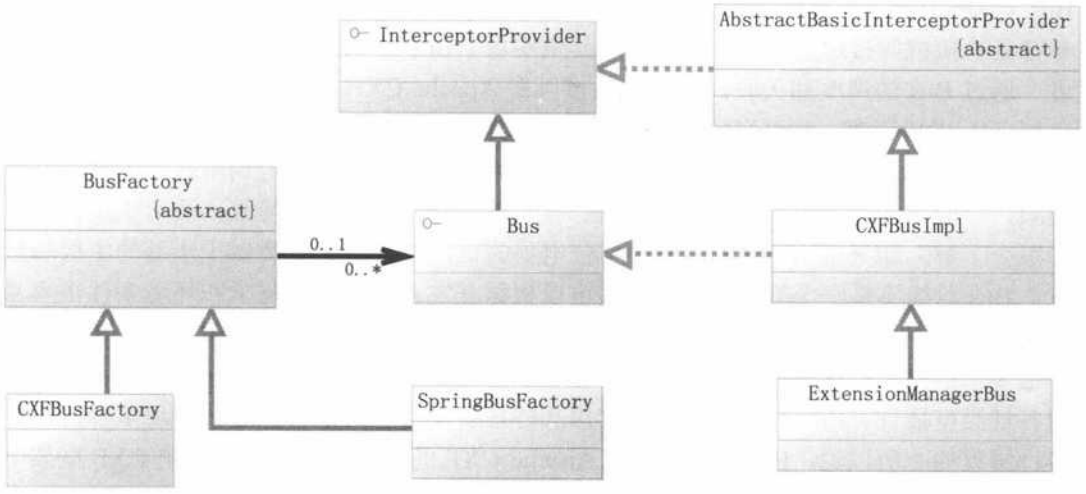


图 5-3　Bus 组件核心类图

从设计模式上看，Bus 组件框架采用工厂模式和桥梁模式。org.apache.cxf.interceptor.InterceptorProvider 接口是最顶层的接口。InterceptorProvider 接口有四个基本的方法，如下所示：

```
public abstract List getInInterceptors();
public abstract List getOutInterceptors();
public abstract List getInFaultInterceptors();
public abstract List getOutFaultInterceptors();
```

org.apache.cxf.Bus 继承 org.apache.cxf.interceptor.InterceptorProvider 接口，同样也是一个高层次接口。Bus 接口通过工厂模式由 org.apache.cxf.bus.spring.SpringBusFactory 来创建，Bus 接口的实现类是 org.apache.cxf.bus.CXFBusImpl。SpringBusFactory 继承抽象类 BusFactory，采用 Spring 的依赖注入方式来进行定义，具体表现在 cxf.xml 定义文件有一行配置信息，即<bean id="cxf" class="org.apache.cxf.bus.CXFBusImpl"/>。

当然，Bus 组件很复杂，图 5-3 只是简单地展示了一下，即抽象类 AbstractBasicInterceptorProvider 实现 InterceptorProvider 接口，而 CXFBusImpl 继承 AbstractBasicInterceptorProvider 类。Bus 也具有扩展性，这样增加一个 ExtensionManagerBus 类来继承 CXFBusImpl 类，可以实现 Bus 的扩展性和适应性。

当然，开发人员在使用 Apache CXF 进行二次开发的过程中很少直接对 Bus 进行调用。但是开发人员调用的组件最终都转化到了 Bus 组件的方法和内容。Apache CXF 提供了对多种 Data Bindings、Binding、Transport 可插拔的配置，只需要在类路径中添加相关的 Binding 和 Transport 包，Apache CXF 的 Bus 会自动加载这些模块，并调用相关接口创建对应的 Data Bindings、Binding 和 Transport。

分析一下 cxf.xml 的定义文件。cxf.xml 的代码如下：

```
<bean id="cxf" class="org.apache.cxf.bus.CXFBusImpl"/>

<bean id="org.apache.cxf.resource.ResourceManager" class="org.apache.cxf.bus. resource.
ResourceManagerImpl">
      <property name="resolvers"> …… </property>
      <property name="bus" ref="cxf"/>
   </bean>

<bean  id="org.apache.cxf.binding.BindingFactoryManager"
class="org.apache.cxf.binding.BindingFactoryManagerImpl">
      <property name="mapProvider"> …… </property>
      <property name="bus" ref="cxf"/>
</bean>

<bean  id="org.apache.cxf.transport.DestinationFactoryManager"
class="org.apache.cxf.transport.DestinationFactoryManagerImpl">
      <property name="mapProvider"> …… </property>
      <property name="bus" ref="cxf"/>
   </bean>

<bean id="org.apache.cxf.transport.ConduitInitiatorManager"
  class="org.apache.cxf.transport.ConduitInitiatorManagerImpl">
      <property name="mapProvider"> …… </property>
      <property name="bus" ref="cxf"/>
</bean>

<bean id="org.apache.cxf.wsdl.WSDLManager" class="org.apache.cxf.wsdl11.WSDLManagerImpl">
      <property name="bus" ref="cxf"/>
</bean>

<bean  id="org.apache.cxf.workqueue.WorkQueueManager"
    class="org.apache.cxf.workqueue.WorkQueueManagerImpl">
      <property name="bus" ref="cxf"/>
</bean>

<bean id="org.apache.cxf.buslifecycle.BusLifeCycleManager"
 class="org.apache.cxf.buslifecycle.CXFBusLifeCycleManager">
      <property name="bus" ref="cxf"/>
</bean>

<bean   id="org.apache.cxf.endpoint.ServerRegistry"
class="org.apache.cxf.endpoint.ServerRegistryImpl">
      <property name="bus" ref="cxf"/>
</bean>

<bean id="org.apache.cxf.transports.http.QueryHandlerRegistry"
 class="org.apache.cxf.transport.http.QueryHandlerRegistryImpl">
      <property name="bus" ref="cxf"/>
      <property name="queryHandlers"> …… </property>
</bean>

<bean  id="org.apache.cxf.endpoint.EndpointResolverRegistry"
class="org.apache.cxf.endpoint.EndpointResolverRegistryImpl">
      <property name="bus" ref="cxf"/>
</bean>

<bean  id="org.apache.cxf.headers.HeaderManager"
class="org.apache.cxf.headers.HeaderManagerImpl">
      <property name="bus" ref="cxf"/>
```

```
</bean>

<bean  id="org.apache.cxf.catalog.OASISCatalogManager"
 class="org.apache.cxf.catalog.OASISCatalogManager">
       <property name="bus" ref="cxf"/>
   </bean>

<bean  id="org.apache.cxf.endpoint.ServiceContractResolverRegistry"
class="org.apache.cxf.endpoint.ServiceContractResolverRegistryImpl">
       <property name="bus" ref="cxf"/>
  </bean>
```

上面的配置文件说明，org.apache.cxf.bus.CXFBusImpl 是 Bus 的一个默认实现类。其注入到 DestinationFactoryManager、ConduitFactoryManager、BindingFactoryManager 等中。其主要职责是提供不同的扩展访问。根据该 Bus 的执行情况，Bus 也可能负责 Apache CXF 内部的联系。

针对具体的某个实现，如 JMS 的实现，首先载入 cxf-extension-jms.xml 配置文件，内容如下：

```
<bean class="org.apache.cxf.transport.jms.JMSTransportFactory"
     id="org.apache.cxf.transport.jms.JMSTransportFactory"     lazy-init="true">
      <property name="bus" ref="cxf"/>
      <property name="transportIds">…</property>
   </bean>
```

在该配置文件中，JMSTransportFactory 是处理 JMS 的工厂类，其注入 cxf，即 org.apache.cxf.bus.CXFBusImpl，实现 JMSTransportFactory 与 Bus 的关联。

5.3 消息（Messaging）和拦截器（Interceptors）组件介绍

Apache CXF 框架是建立在通用消息层上的应用框架，其构成包括消息（Messaging）组件，拦截器（Interceptors）组件和相位（Phase）组件。拦截器组件包括拦截器和拦截器链。拦截器是基本的功能单位。Apache CXF 通过划分处理和发送消息的方式来构建一个非常灵活的架构。它可以在处理过程中重新配置任何节点。这也给予了 Apache CXF 暂停及恢复拦截器链条的能力。

Apache CXF 的消息（Messaging）和拦截器（Interceptors）组件如图 5-4 所示。

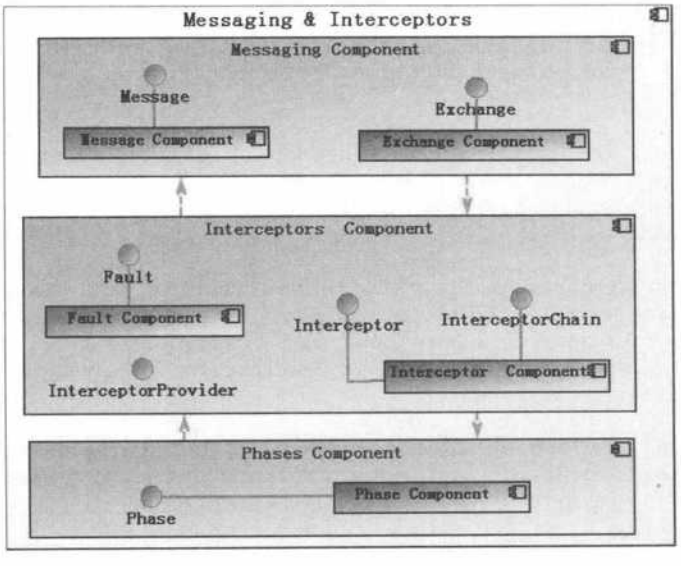


图 5-4 Apache CXF 的 Messaging 和 Interceptors 组件图

在 Apache CXF 的 Messaging 组件中，主要涉及的接口有 Message 接口和 Exchange 接口。Message 接口是 Apache CXF 传递信息的基础，所有的操作都是通过转化成一定的消息格式来进行传输的。Message 接口与 Interceptor 接口结合起来执行处理操作，Message 接口中定义了很多常量，这些都是与 Message 操作有关的信息。除了消息概念，这里还有一个交换器（Exchange）的概念。交换器（Exchange）不仅进行消息的转换，还可以转化特定的属性。例如，交换器（Exchange）实现类建立 In/Out，InFault/OutFault Message 之间的联系。Exchange 实现类还包含了与消息传输有关的 Conduit 和 Destination 的信息，同时也可以配置和 Session 有关的其他信息，以及 OneWay 的消息。消息组件对于开发人员是不可见的，但是开发人员定义的数据绑定、协议绑定，最终转化为消息组件的处理机制。

Apache CXF 的 Messaging、Phase 和 Interceptors 类结构如图 5-5 所示。

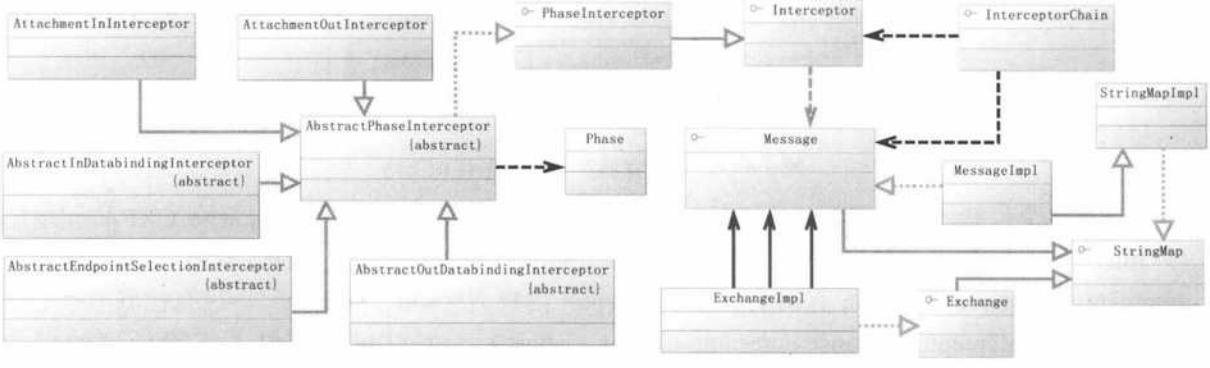


图 5-5　Apache CXF 的 Messaging、Phase 和 Interceptors 类图

拦截器（Interceptor）和拦截器链（InterceptorChain）是同时出现的。可拦截器链（InterceptorChain）依赖拦截器（Interceptor）接口。同时，拦截器（Interceptor）接口和拦截器链（InterceptorChain）又依赖消息（Message）接口。消息（Message）接口的实现类是 MessageImpl 类。ExchangeImpl 类是 Exchange 接口的实现类，但是其与消息（Message）接口关联。AbstractPhaseInterceptor 抽象类是 Phase 组件的核心，AbstractPhaseInterceptor 抽象类实现 PhaseInterceptor 接口，同时关联 Phase 接口，这样就与 Phase 接口的实现联系起来。其他的 AttachmentOutInterceptor 类、AttachmentInInterceptor 类、AbstractInDatabindingInterceptor 类、AbstractEndpointSelectionInterceptor 类和 AbstractOutDatabindingInterceptor 类都是继承 Phase 组件的核心 AbstractPhaseInterceptor 抽象类。

拦截器（Interceptors）组件包括拦截器和拦截器链。Apache CXF 的拦截器可以在很大程度上扩展 CXF 功能。通过开发人员的定制化 Interceptors，可以改动请求和响应的一些消息处理。这样，对于开发人员，可以在不修改核心模块的情况下，动态添加很多功能。同时，Apache CXF 通过 Interceptors 对消息执行特殊处理，实现了很多主要功能模块，例如，日志记录、SOAP 消息处理、消息的压缩处理，等等。

Apache CXF 框架实现拦截器（Interceptors）的原理是采用动态代理方式。其动态代理策略的实现模式如图 5-6 所示。当开发人员希望在一个关注点插入自己的处理程序，可以使用拦截器（Interceptors）绑定到需要它的方法，实际上就是在目标对象生成前创建了一个动态代理。该代

理应有一个实现方法，该方法与原始目标对象拥有相同的接口。当调用者调用代理时，开发人员可以在调用真实方法前，轻松添加需要的定制特性。通过使用拦截，可以更加高效地添加类似安全、声明性事务和远程控制这些自定义服务到对象方法。Interceptor 主要定义两个方法，一个用于处理消息 handleMessage，一个用于处理错误 handleFault。在实行具体的 Interceptor 的这两个方法中，千万别调用 Interceptor 内部的成员变量。这是由于 Interceptor 是面向消息来进行处理的，每个 Interceptor 都有可能运行在不同的线程中，如果调用了 Interceptor 中的内部成员变量，就有在 Interceptor 中造成临界资源的访问的情况，而这时的 Interceptor 也就不是线程安全的 Interceptor 了。

拦截器（Interceptors）对于开发人员是可见的，并且开发人员通过实现拦截器（Interceptors）的接口来完成业务需要的功能。

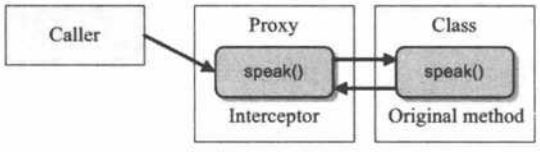


图 5-6　轻量级容器拦截的动态代理策略

当有多个拦截器时，需要将许多的 Interceptor 组合在一起运用。因此 Apache CXF 框架增加了 InterceptorChain。这是采用的是 GoF23 种设计模式的责任链模式。InterceptorChain 拥有多个 Interceptor。InterceptorChain 有调配安置 Interceptor 的权力（add、remove），也有控制消息处理的权力（doInterceptor、pause、resume、reset、abort），同时也有交付不正确处理的权力（{getset}FaultObserver）。更有意思的是为灵活控制 Interceptor 的处理消息顺序（doInterceptStartingAt、doInterceptorStartingAfter）。拦截器链（InterceptorChain）对于开发人员是可见的，并且开发人员也是通过添加拦截器（Interceptors）到拦截器链（InterceptorChain）来完成业务需要的功能。其实现过程分为客户端和服务器端，客户端实现过程如图 5-7 所示。

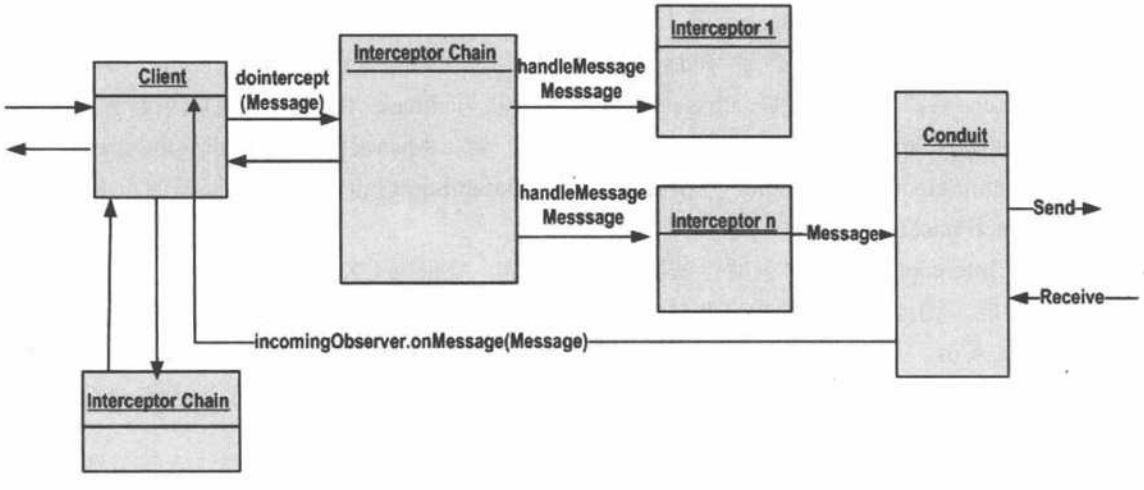


图 5-7　Apache CXF 拦截器客户端实现过程

其服务器端的实现如图 5-8 所示。

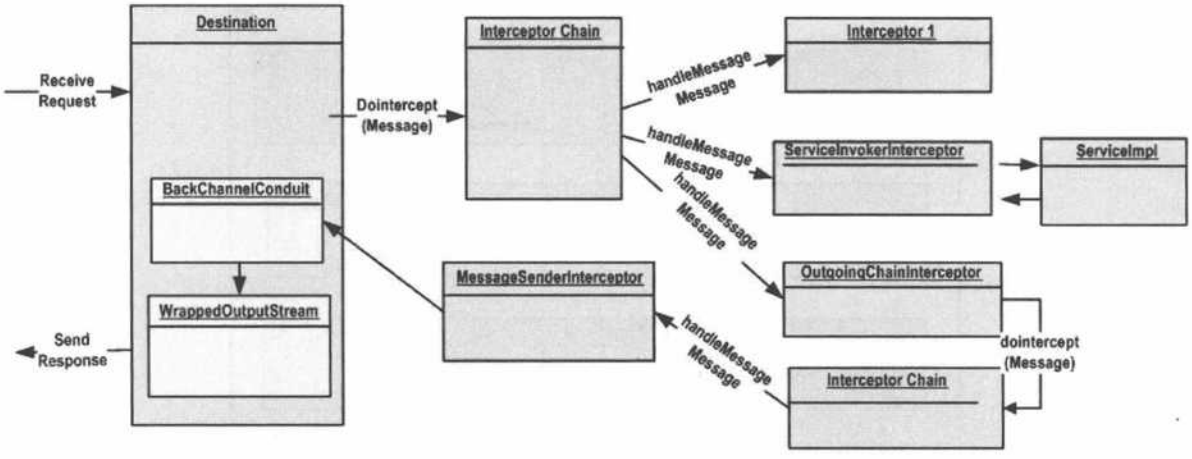


图 5-8　Apache CXF 拦截器服务器端实现过程

InterceptorProvider 接口是核心接口的基础。甚至 Bus 接口都是继承 InterceptorProvider 接口。InterceptorProvider 接口可以覆盖所有的 Interceptor 引用，包括 Bus。InterceptorProvider 接口在抽象方法 getInInterceptors 可以获得 In 的 Interceptor，抽象 getOutInterceptors 方法可以获得 Out 的 Interceptor，抽象 getInFaultInterceptors 方法可以获得 InFault 的 Interceptor，抽象 getOutFaultInterceptors 方法可以获得 OutFault 的 Interceptor。这些 Interceptor 都是开发者添加的 Interceptor，而 InterceptorChain 会根据开发者添加的 Interceptor 对象，组建内部的 Interceptor 实例，完成具体的功能任务。

为了简化配置 Interceptor 的复杂操作，在这里配置了 Feature，通过 Feature 我们可以向 Bus、Client、Endpoint 配置不同功能的 Interceptor 组。这样可以缩减配置文件的大小。Feature 对于开发人员是可见的，并且开发人员可以整合 Feature 来完成业务需要的功能。

5.4　前端编程模型（Frontend）组件介绍

前端编程模型提供一种用于 Apache CXF 的服务器端和客户端进行交互的模型。前端编程模型的设计规范主要遵循 JAX-WS 规范和 JAX-RS 规范。这样可以实现 JAX-WS 组件或 JAX-RS 组件与 Apache CXF 其余组成部分的分离——这样能达到更好的耦合。开发人员通过在添加到服务端点和客户端端点的拦截器来实现 Apache CXF 内部提供的功能。前端编程模型是一种可见性模式，即开发人员可以直接接触到该模块里面的接口和类。

Apache CXF 前端应用模式的组件如图 5-9 所示。

Apache CXF 支持四种前端应用模式，一种是 JAX-WS 前端模式，一种是 JAX-RS 前端模式，一种是 JavaScript 语言，还有一种是简化前端模式（simple frontend）。

在结构上，JAX-WS 和 JAX-RS 都通过提供 FactoryBean 来进行前端和后端的交互实现。但是 JAX-WS 和 JAX-RS 提供 FactoryBean 部件依赖于 Frontend 组件的 FactoryBean 部件，而 Frontend 组件的 FactoryBean 部件又依赖于 Endpoint 组件的由 Client、Endpoint 和 Server 接口实现的组件，且 Endpoint 组件最终依赖于 Interceptors 组件。Interceptors 组件包括 Interceptor 接口、Fault 接口和 InterceptorChain 接口等。

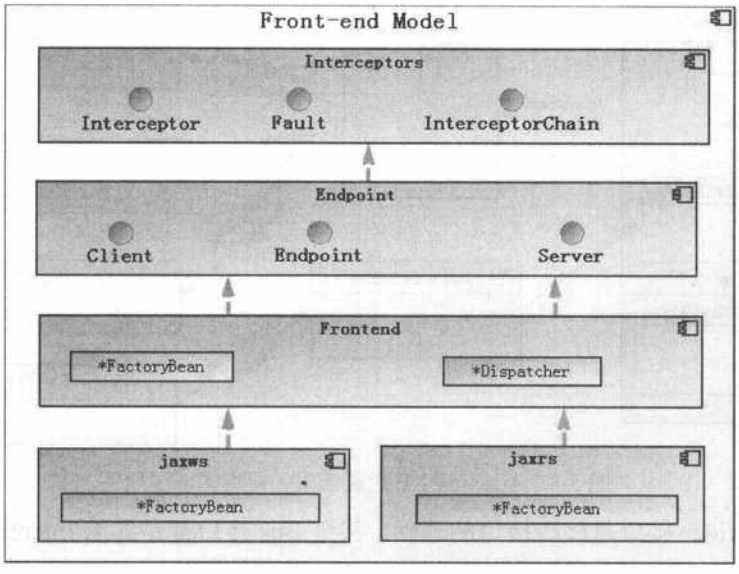


图 5-9　CXF 前端应用模式组件图

Apache CXF 前端应用模式的类结构如图 5-10 所示。

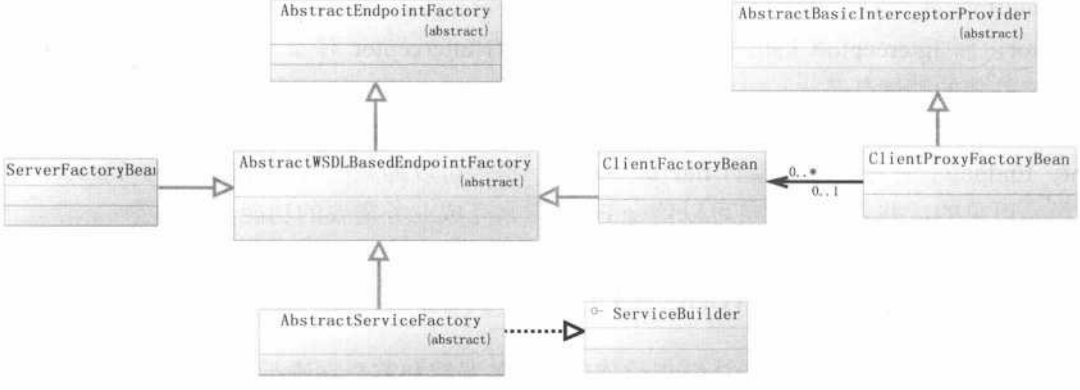


图 5-10　Apache CXF 前端应用模式类图

图 5-10 中，AbstractEndpointFactory 是祖先抽象类，而抽象类 AbstractWSDLBasedEndpointFactory 是 Apache CXF 前端应用模式的核心。ServerFactoryBean 类、AbstractServiceFactory 类和 ClientFactoryBean 类都继承 AbstractWSDLBasedEndpointFactory 抽象类。而 ClientProxyFactoryBean 类是继承并关联 ClientFactoryBean 类，其实就是 ClientFactoryBean 实例化对象的代理。

ServerFactoryBean 类主要用于服务器端的服务器端点。AbstractServiceFactory 类主要用于简化前端模式(simple frontend)的实现。而最常用的是 ClientFactoryBean 类和 ClientProxyFactoryBean 类，这两者的实现都基于 ClientFactoryBean 类，该类将创建一个客户端用于实现特定的服务类，代码如下：

```
ClientProxyFactoryBean factory = new ClientProxyFactoryBean();
 factory.setServiceClass(YourServiceInterface.class);
 YourServiceInterface client = (YourServiceInterface) factory.create();
 Client cxfClient = ClientProxy.getClient(client);
```

5.5　服务模型（Service Model）组件说明

服务模型对于开发人员是不可见的，主要用于内部对象之间的相互处理。服务模型是 Apache CXF 中 Service 的代表。它由两部分组成：第一部分是 ServiceInfo，其中包含该服务的 WSDL 模型以及操作、绑定和端点；第二部分就是 Service 本身，它包含了 ServiceInfo、数据绑定、服务拦截器、服务属性，等等。

Service（服务）可以由不同来源来构造，如类、WSDL 1.1 和 WSDL 2.0 等。前端编程模型通常通过 ServiceFactory 来创建一个服务。编程模型构建服务模型并配置服务拦截器、数据绑定等。

Apache CXF 的 Service Model 组件如图 5-11 所示。

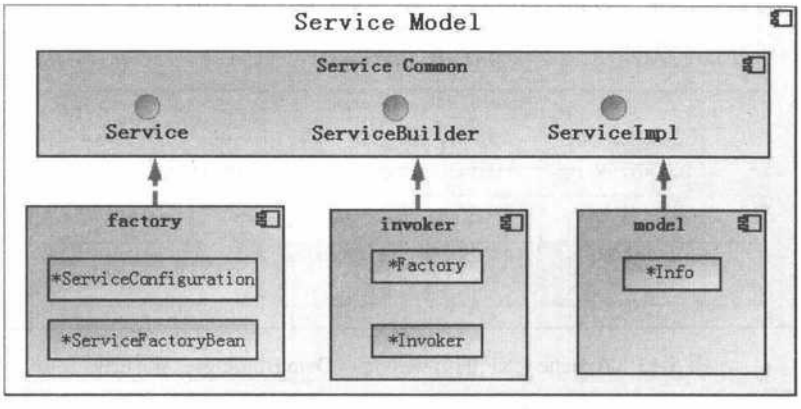


图 5-11　Apache CXF 的 Service Model 组件图

在结构上，Apache CXF 框架服务模型主要是由 Service 组件接口和三个实现组件组成的。Service 组件接口主要包括 Service 接口、ServiceBuilder 接口和 ServiceImpl 类实现。三个实现组件主要是 factory 组件、invoker 组件、model 组件。

其中值得一提的是 model 组件。model 组件包括一个顶级对象 ServiceInfo 组件。ServiceInfo 组件包括 InterfaceInfo 组件、BindingInfo 组件和 EndpointInfo 组件。InterfaceInfo 组件、BindingInfo 组件和 EndpointInfo 组件也有自己的下属组件。其结构如图 5-12 所示。

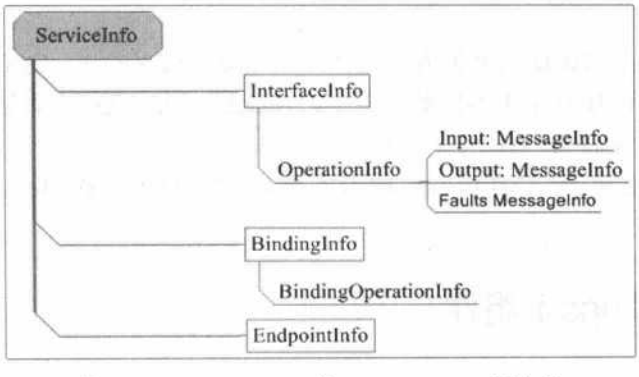


图 5-12　Apache CXF 的 Service Model 层次图

从图 5-12 中可以看到。Apache CXF 的 Service Model 核心是 ServiceInfo 接口，该接口下面有三个比较重要的接口，分别是 InterfaceInfo 接口、BindingInfo 接口、EndpointInfo 接口。

这种服务模型可以应用到动态客户端的应用中。

5.6 数据绑定（Data Bindings）组件

数据绑定（Data Bindings）实现 XML 和 Java 之间的映射。数据绑定（Data Bindings）通过生成 XML schema 的方式把 XML 转换为数据或者数据转换为 XML，并支持 wsdl2java 代码生成器。不是所有的数据绑定都支持解析、转换、生成等功能。但最低限度，数据绑定必须提供数据转换。数据绑定（Data Bindings）对于开发人员是可见的。

Apache CXF 的数据绑定（Data Bindings）组件如图 5-13 所示。

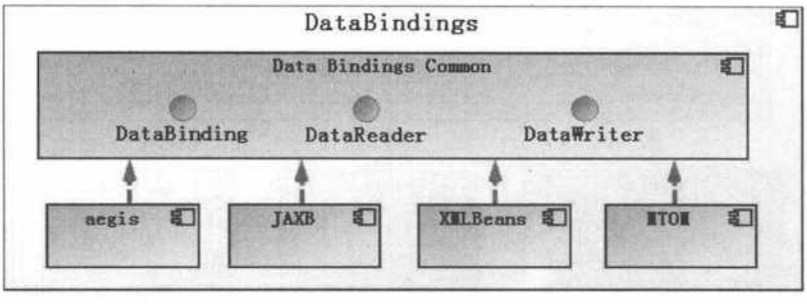


图 5-13　Apache CXF 的数据绑定（Data Bindings）组件图

在结构上，数据绑定（Data Bindings）组件主要是由数据绑定（Data Bindings）接口和四个实现组件组成。数据绑定（Data Bindings）接口主要包括 DataBinding 接口、DataReader 接口和 DataWriter 接口。DataReader 接口负责数据的读取，DataWriter 接口负责数据的编写。DataBinding 接口负责把 DataReader 接口和 DataWriter 接口集合起来使用。四个实现组件主要是 Aegis 组件、JAXB 组件、XMLBeans 组件和 MTOM 组件。这些组件分别实现不同的 DataBinding 接口、DataReader 接口和 DataWriter 接口。这种设计模式采用的是 GoF23 种设计模式中的策略模式。同时这种框架还可以继续扩展，现在 Apache CXF 框架已经加入了 SDO、JiBX 等数据绑定。

Aegis Databinding 是 Xfire 特有的数据绑定模式。Apache CXF 为了兼容 Xfire 的功能，而对该模块也采用了集成方式。

JAXB 组件是按照 JAXB 规范来集成的组件。在 Apache CXF 中，JAXB 是默认的数据绑定，Apache CXF 将采用 JAXB 规范来实现数据绑定。Apache CXF 支持多种版本的 JAXB 规范，如 Apache CXF 2.3 版本就支持 JAXB 2.0 和 JAXB 2.1。

XMLBeans 组件是基于 XMLBeans 规范来形成的组件。XMLBeans 是一种高级易用的 XML-Java 绑定技术。

5.7 绑定（Bindings）组件

绑定（Bindings）在顶层传输协议上提供了具体格式和协议的映射。

在结构上，一个绑定包含三个主要部分，Binding、BindingFactory 和 BindingFactoryManager。BindingFactoryManager 是 BindingFactory 的容器，而 BindingFactory 又是 Binding 的工厂类。在实际运行过程中，BindingFactory 通过服务模式的 BindingInfo 来构建一个绑定。该绑定实现了 createMessage 方法来定制化拦截器，而拦截器即实现了针对一个绑定消息的具体功能。

Apache CXF 框架的绑定（Bindings）组件如图 5-14 所示。

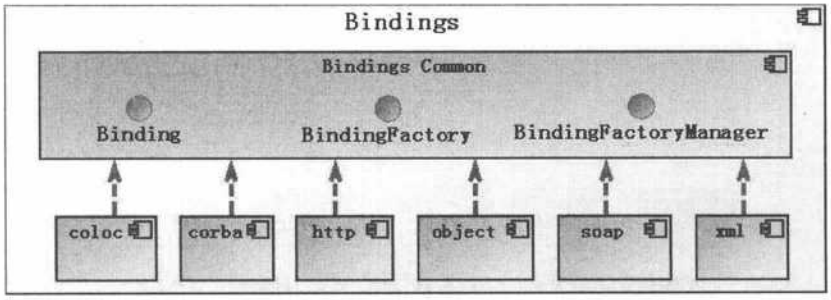


图 5-14　Apache CXF 的绑定（Bindings）组件图

绑定（Bindings）组件主要是由绑定（Bindings）接口和六个实现组件。绑定（Bindings）接口主要包括 Binding 接口、BindingFactory 接口和 BindingFactoryManager 接口。六个实现组件主要是 coloc Bindings 组件、CORBA Bindings 组件、http Bindings 组件、object Bindings 组件、SOAP Bindings 组件和 XML Bindings 组件，这些组件分别实现不同的 Bindings 接口。这种设计模式采用的是 GoF23 种设计模式中的策略模式。

绑定（Bindings）组件包括下列组件：

- coloc Bindings 组件，这是 CXF 一个特殊的协议实现内容。
- CORBA Bindings 组件主要绑定 CORBA 对象。
- http Bindings 组件主要绑定 http 对象。
- object Bindings 组件主要绑定一个任意数据对象，一般情况下是一个 JavaBean。
- SOAP Bindings 组件主要绑定 SOAP 对象。
- XML Bindings 组件主要绑定一个纯的 XML 对象。

其中 SOAP 绑定有自己的消息类 SoapMessage。它增加了一些功能，如保持当前 Soap 版本和消息头。SOAP 绑定还增加了一个特殊类型的拦截器，称为 SoapInterceptor，SOAP 绑定的拦截器支持 Soap11 和 Soap12。该 SoapInterceptor 添加了两个新方法：

```
Set<URI> getRoles();
Set<QName> getUnderstoodHeaders();
```

通过调用 getRoles 和 getUnderstoodHeaders 方法通知 SOAP 拦截器去理解特定的信息头和角色。

有许多拦截器被设计来处理 SOAP 消息：

- StaxInInterceptor：从输入的 InputStream 创建从一个 XMLStreamReader
- ReadHeadersInterceptor：读入头到 SoapMessage 中。
- MustUnderstandInterceptor：检查了所有的 mustUnderstand 属性对所有 SoapInterceptor 的 getUnderstoodHeaders 头的方法。
- SoapOutInterceptor：SOAP 消息的输出拦截器。

5.8 传输协议（Transport）组件

为了在开发过程中对绑定和前端屏蔽内部的传输细节，Apache CXF 提供了自己的传输协议抽象。

传输协议的组件结构如图 5-15 所示。

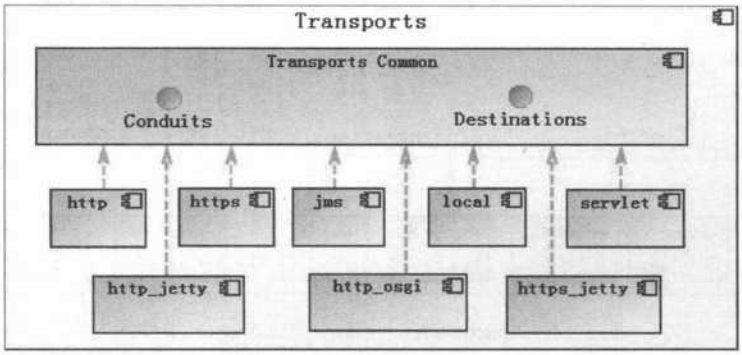


图 5-15　Apache CXF 的传输（Transport）组件图

Apache CXF 框架包括其本身的传输抽象层以达到针对前端层和绑定层隐藏具体传输细节，具体实现包括 http、https、jms、local servlet、http_jetty、http_osgi、https_jetty 等内容。

在结构上，Apache CXF 的传输协议（Transport）组件包含两个主要部分，Conduits 组件和 Destinations 组件。Apache CXF 的传输（Transport）组件主要是由 Transport 接口和 8 个实现组件组成。传输（Transport）接口主要包括 Conduits 接口和 Destinations 接口。8 个实现组件主要是 http Transport 组件、http_jetty Transport 组件、http_osgi Transport 组件、https Transport 组件、https_jetty Transport 组件、jms Transport 组件、servlet Transport 组件和 local Transport 组件等。这些组件分别实现不同的 Transport 接口。这种设计模式采用的是 GoF23 种设计模式中的策略模式。其结构是这样的。

- http Transport 组件主要支持 http 传输协议。
- http_jetty Transport 组件主要支持 http_jetty 传输协议。
- http_osgi Transport 组件主要支持 http_osgi 传输协议。
- https Transport 组件主要支持 https 传输协议。
- https_jetty Transport 组件主要支持 https_jetty 传输协议。
- jms Transport 组件主要支持 jms 传输协议。
- servlet Transport 组件主要支持 servlet 传输协议。
- local Transport 组件主要支持 local 传输协议。

5.9 CXF 的注释

Apache CXF 提供了几种自定义的注释，这样的注释可以使 Apache CXF 在运行时进行灵活的配置和定制化。Apache CXF 的注释主要如下：

- org.apache.cxf.feature.Features

- org.apache.cxf.annotations.WSDLDocumentation
- org.apache.cxf.annotations.WSDLDocumentationCollection
- org.apache.cxf.annotations.SchemaValidation
- org.apache.cxf.annotations.DataBinding
- org.apache.cxf.annotations.Logging (since 2.3)
- org.apache.cxf.annotations.GZIP (since 2.3)
- org.apache.cxf.annotations.FastInfoset (since 2.3)

1. @Features 注解的使用

使用@Features 注解是通过实现 org.apache.cxf.feature.Features 接口来实现的。@Features 注解用于添加 Features 功能。功能清单 FeaturesList 给开发者提供“开箱即用”的 Features 功能列表，但开发人员可以很容易地创建自己的 Features 功能。在许多情况下，然而，那些 Features 功能都有用于提供更多的配置和更灵活的控制的注解。

2. Interceptor 注解的使用

Interceptor 注解包括 org.apache.cxf.interceptor 包下 InInterceptors 接口、OutInterceptors 接口、OutFaultInterceptors 接口和 InFaultInterceptors 接口的实现，分别对应的是@InInterceptors、@OutInterceptors、@OutFaultInterceptors 和@InFaultInterceptors。

Interceptor 注解主要是在不同的拦截器链条中添加拦截器以便用于处理消息。

3. @WSDLDocumentation 注解和@WSDLDocumentationCollection 注解的使用

@WSDLDocumentation 注解是 org.apache.cxf.annotations 包下 WSDLDocumentation 接口的实现，而@WSDLDocumentationCollection 注解是 org.apache.cxf.annotations 包 WSDLDocumentationCollection 接口的实现。

对于“Java 代码优先”的场景，其中的 WSDL 注解是从 Java 接口/代码派生出来的，这些注解对于生成的 WSDL 文档中可以在不同位置上添加 wsd:documentation 元素。其示例代码如下：

```
@WebService
@WSDLDocumentationCollection(
    {
      @WSDLDocumentation("My portType documentation"),
      @WSDLDocumentation(value = "My top level documentation",
placement = WSDLDocumentation.Placement.TOP),
      @WSDLDocumentation(value = "My binding doc",placement = WSDLDocumentation.
Placement.BINDING)
    }
)
public interface MyService {
    @WSDLDocumentation("The docs for echoString")
    String echoString(String s);
}
```

4. @SchemaValidation 注解的使用

@SchemaValidation 注解是 org.apache.cxf.annotations 包下 SchemaValidation 接口的实现。默认情况下，出于性能的原因，Apache CXF 不验证消息是否符合 schema 格式。如果要验证 XML 是否符合 schema 格式，可以打开消息的 Schema 验证器。通过验证处理，可以更容易确定消息和

schema 格式不匹配的问题。

5. @DataBinding 注解的使用

@DataBinding 注解是 org.apache.cxf.annotations 包下 DataBinding 接口的实现。

@DataBinding 注解设置数据绑定类关联该服务。默认情况下，Apache CXF 只针对 JAXB 数据绑定。但是，Apache CXF 支持多种数据绑定，如 XMLBeans、Aegis、SDO，甚至更多的数据绑定。该注解主要用于选择绑定类的配置说明。例如，下面的程序代码，Apache CXF 的数据绑定是 SDO。

```
@DataBinding(org.apache.cxf.sdo.SDODataBinding.class)
public interface MyService {
   public commonj.sdo.DataObject echoStruct(
      commonj.sdo.DataObject struct
   );
}
```

6. @Logging 注解的使用

@Logging 注解是 org.apache.cxf.annotations 包下 Logging 接口的实现，其设置值和说明见表 5-2。

表 5-2 @Logging注解的设置值和说明

设置值	说明
limit	设置日志集的限制大小，超过后消息在日志截断。默认是64K
inLocation	设置日志记录收到的消息。可用<stderr>、<stdout>、<logger>或file: URL。默认是<logger>
outLocation	设置日志记录传出的消息。可用<stderr>、<stdout>、<logger>或file: URL。默认是<logger>

下面是一个@Logging 注解的例子。

```
@Logging(limit=16000, inLocation="<stdout>")
public interface MyService {
   String echoString(String s);
}
```

7. @GZIP 注解的使用

@GZIP 注解是 org.apache.cxf.annotations 包下 GZIP 接口的实现。其设置值和说明见表 5-3。

表 5-3 @GZIP注解的设置值和说明

设置值	说明
threshold	根据该消息判断是否进行gzip压缩

gzip 可进行压缩处理。从一个客户端的初始请求后将不会被 gzip 压缩。但是如果服务器支持，客户端的初始请求将会增加一个 Accept 头，响应将会被 gzip 压缩，同时后续请求也将被 gzip 压缩。

8. @FastInfoset 注解的使用

@FastInfoset 注解是 org.apache.cxf.annotations 包下 FastInfoset 接口的实现。其设置值和说明见表 5-4。

表 5-4　@FastInfoset注解的设置值和说明

设置值	说明
force	强调采用fastinfoset模式而不是谈判（negotiating）模式。默认为false

fastinfoset 是一种增强模式。从客户端的初始请求不会采用 fastinfoset 模式，但如果在请求中增加一个 Accept 头，若服务器支持 fastinfoset 模式，那么服务器会采用 fastinfoset 模式来响应。其后的请求也会采用 fastinfoset。

9. @EndpointProperty 注解和@EndpointProperties 注解的使用

@EndpointProperty 注解是 org.apache.cxf.annotations 包下 EndpointProperty 接口的实现。@EndpointProperties 注解是 org.apache.cxf.annotations 包下 EndpointProperties 接口的实现。

添加一个属性到端点上，如 WS-Security 等，这样可以通过端点来进行属性的配置。传统上，这些设置将通过 Spring 配置<jaxws:endpoint>元素下的<jaxws:properties>元素来实现，但这些注解可以让这些属性配置到代码中。

下面是一个@ EndpointProperty 注解和@ EndpointProperties 注解的例子代码。

```
@WebService
@EndpointProperties(
    {
      @EndpointProperty(name = "my.property", value="some value"),
      @EndpointProperty(name = "my.other.property", value="some other value"),
    })
public interface MyService {
   String echoString(String s);
}
```

10. @Policy 注解和@Policies 注解的使用

@Policy注解是org.apache.cxf.annotations包下Policy接口的实现。@Policies注解是org.apache.cxf.annotations 包下 Policies 接口的实现。

用于附加 WS-Policy 片段到一个服务或操作。该策略支持的属性见表 5-5。

表 5-5　WS-Policy策略的属性

属性名称	属性描述
uri	需要得到包含Policy定义文件的位置
includeInWSDL	当生成一个WSDL，Policy是否包含在WSDL文件中。默认值是true
placement	在指定位置上放置政策
faultClass	如果位置是一个FAULT，这将指定政策适用于的故障（fault）

下面是@ Policy 注解和@ Policies 注解的示例代码。

```
Policies({
   @Policy(uri = "annotationpolicies/TestInterfacePolicy.xml"),
   @Policy(uri = "annotationpolicies/TestImplPolicy.xml",
         placement = Policy.Placement.SERVICE_PORT),
   @Policy(uri = "annotationpolicies/TestPortTypePolicy.xml",
         placement = Policy.Placement.PORT_TYPE)
}
)
```

```
@WebService
public static interface TestInterface {
    @Policies({
        @Policy(uri = "annotationpolicies/TestOperationPolicy.xml"),
        @Policy(uri = "annotationpolicies/TestOperationInputPolicy.xml",
                placement = Policy.Placement.BINDING_OPERATION_INPUT),
        @Policy(uri = "annotationpolicies/TestOperationOutputPolicy.xml",
                placement = Policy.Placement.BINDING_OPERATION_OUTPUT),
        @Policy(uri = "annotationpolicies/TestOperationPTPolicy.xml",
                placement = Policy.Placement.PORT_TYPE_OPERATION),
        @Policy(uri = "annotationpolicies/TestOperationPTInputPolicy.xml",
                placement = Policy.Placement.PORT_TYPE_OPERATION_INPUT),
        @Policy(uri = "annotationpolicies/TestOperationPTOutputPolicy.xml",
                placement = Policy.Placement.PORT_TYPE_OPERATION_OUTPUT)
        }
    )
    int echoInt(int i);
}
```

5.10 案例场景说明

一切的应用都要来源于实践。本书采用统一例子来说明 Apache CXF 的应用。为什么要采用一个统一的例子呢？这有几个原因：第一：仅让读者理解一个案例的原理，不会让读者花费更多的时间去掌握不同业务。读者的注意力主要还是集中在 SOA 的技术实现；第二，通过不同 SOA 的技术实现，了解这些技术之间的差别；第三，越简单越好，统一的案例是一个非常简单的同时又是大家非常熟悉的场景。

在后面章节中将按照各个 SOA 技术详细介绍 Apache CXF 的应用。在应用介绍过程中，将采用案例模式来说明。在不特别说明的情况下，案例场景是在后台有一个项目数据集，初始化时候默认有两个项目，然后采用 SOA 客户端通过 SOA 方式来访问后台服务器，获得后台服务器上项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作，其用例实现如图 5-16 所示。

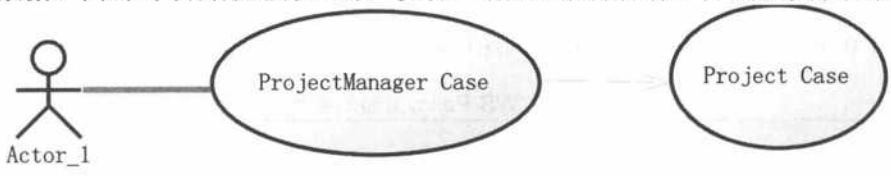


图 5-16 案例的用例图

在图 5-16 中，用例图有三个实体，Actor 是外部访问者，ProjectManager 用例是一个管理的用例，而 Project 用例是用于实现 ProjectManager 用例的组成部分。

本案例是一个项目管理方面的内容，共有 4 个对象。首先有接口类，再有业务逻辑类，最后还有实体类，分别是两个接口和两个实现类。两个接口分别是 ProjectManager 接口和 Projec 接口。两个实现类分别是 ProjectManagerImpl 实现类和 ProjectImpl 实现类。ProjectImpl 是一个实体 Bean，而 ProjectManagerImpl 实现对 Project 的管理，包括对 Project 实体接口的新增、修改、删除和查询等功能的管理类。实际上，ProjectManager 接口和 ProjectManagerImpl 就是一个逻辑管理类，而 Project 接口和 ProjectImpl 实现类就是实体类，其类结构如图 5-17 所示。

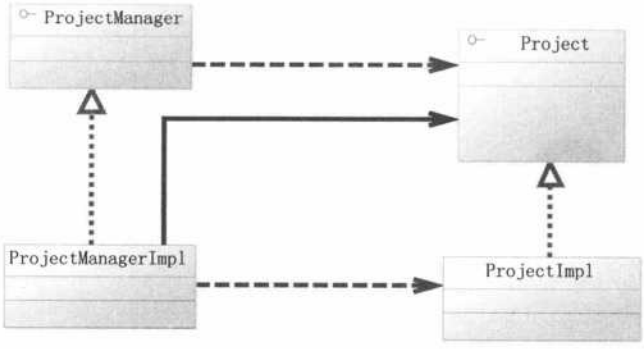


图 5-17 案例的类结构图

当然，这个类图在有些案例中会有一点突变，但是这并不影响其核心实现方式，即外部通过 PowerManager 来进行业务处理，其中 Project 是一个实体。

当然，案例也可以按照鲁棒图方式，其业务的实现如图 5-18 所示。

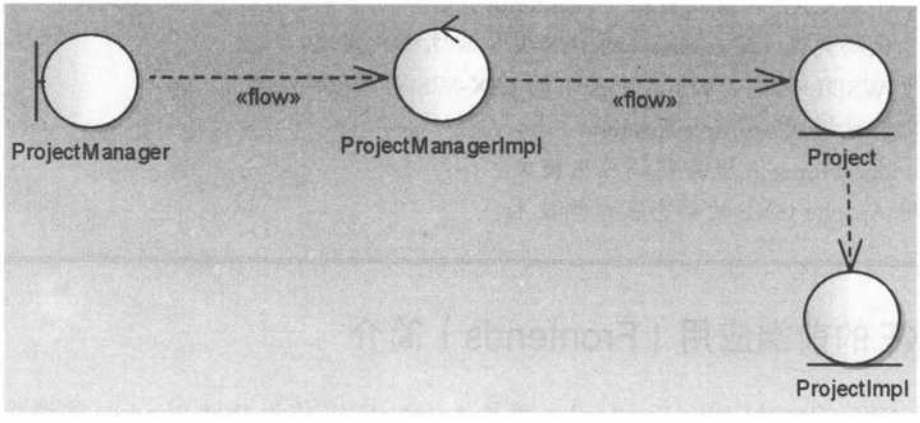


图 5-18　案例的鲁棒图

可以理解 ProjectManager 接口是一个边界类，而 ProjectManagerIml 是一个逻辑类，Project 和 ProjectImpl 是实体类。

对于每个案例，基本上按照案例说明的方式包括三个部分，第一部分是应用场景说明和应用程序架构图示，第二部分实现步骤说明和图示，第三部分是代码结构图示和源代码。

第一部分是应用场景说明和应用程序架构图示，其中应用场景说明主要讲述这个应用要达到一个怎样的目的。我们案例一般情况下都是前端访问后面的项目信息，主要做一些 CRUD（创建、读取、修改和删除）操作。应用程序架构图示主要用几张图来说明这个应用案例的整体架构，对于基于 Web Services 应用程序，重点是三个地方，第一是服务器端，第二是客户端，第三是两者之间的通信。

第二部分实现步骤说明和图示。这部分说明建立这个应用程序要考虑的步骤，一般会用一个流程图来进行图示，这样也方便读者理解。

第三部分是代码结构图示和源代码。首先会用一张应用程序的目录图片来图示案例程序的代码结构。接着会对几个重要的源代码文件进行列表说明。对于源代码文件的说明是案例中内容最多的地方，这里有很多细节，并对这些细节进行详细说明和解释。

第 6 章　CXF 的前端应用

本章主要说明 Apache CXF 的前端应用。首先介绍 Apache CXF 的前端应用（Frontends）的技术实现方式，其次分别说明 5 种 Apache CXF 前端应用的实现方式。

本章内容：

1. 简单介绍 CXF 的前端应用（Frontends）。
2. 基于代码优先（Java First）的 JAX-WS 前端模式实现。
3. 基于 WSDL 优先（WSDL First）的 JAX-WS 前端模式实现。
4. 简化前端模式（simple frontend）。
5. Provider/Dispatch 服务前端应用模式。
6. 采用 Apache CXF 的动态客户端技术。

6.1　CXF 的前端应用（Frontends）简介

Apache CXF 的前端应用（Frontends）就是 Apache CXF 作为 Web Services 的消费者，通过采用客户端模式来调用后台的 Web Services 服务。

Apache CXF 的前端应用（Frontends）有 5 种实现模式，第一种是 JAX-WS 前端模式，第二种是 JAX-RS 前端模式（与支持 Restful 的 Web Services 是一致的），第三种采用动态客户端技术，第四种是 Provider 和 Dispatch 接口方式。第五种是简化前端模式（simple frontend），其中 JAX-WS 前端模式和简化前端模式都是支持 SOAP 协议的 Web Services。

JAX-WS 前端模式也有两种实现方式，一种是代码优先（Java First），一种是协议优先（WSDL First）。基于代码优先（Java First）的编程方式就是编写 Java 程序代码，然后发布 Java 代码来产生客户端和服务器端的 Web Services。这里要用到 CXF JAX-WS 的 Java2WSDL，该程序可以把 Java 对象转化为 WSDL 和 XSD，并以此为基础形成 Web Services。基于协议优先（WSDL First）的编程方式就是首先生成 WSDL 文档，然后通过 WSDL 文档来产生客户端和服务器端的 Java 代码，然后发布这些生成的客户端和服务器端的 Web Services。这种模式的应用场景需要开发人员对 WSDL、XML、XSD 等非常熟悉，同时外部程序已经提供了 WSDL。在这样的环境条件下，把 WSDL 转化为 Java 对象。

下面就分别介绍这 5 种实现模式的原理和实现过程。

6.2　基于代码优先（Java First）的 JAX-WS 前端模式实现

首先说明基于代码优先（Java First）的 Web Services 的方式和步骤，然后通过几个例子程序来进行详细说明。

6.2.1　基于代码优先（Java First）的 Web Services 的步骤

使用 Java 代码来创建基于 JAX-WS 的 Web Services，通常都有下面的几个步骤。

（1）创建服务端点接口（SEI），定义公开为 Web Services 服务的方法。

可以直接从一个 Java 类或一个接口开始。一般而言，接口比 Java 类要好一点，这样可以实现接口和具体实现类耦合开来。

（2）为 Java 类或接口添加所需的注释。

针对要提供的外部 Web Services 服务，进行代码注释，同时也可以添加一些其他方面的注释，如数据绑定、服务名称、服务端口，等等。

（3）编写发布这个 Java 程序的 Web Services 并发布服务。

（4）编写 Web Services 的客户端应用程序。

下面详细介绍这 4 个步骤，具体实现流程如图 6-1 所示。

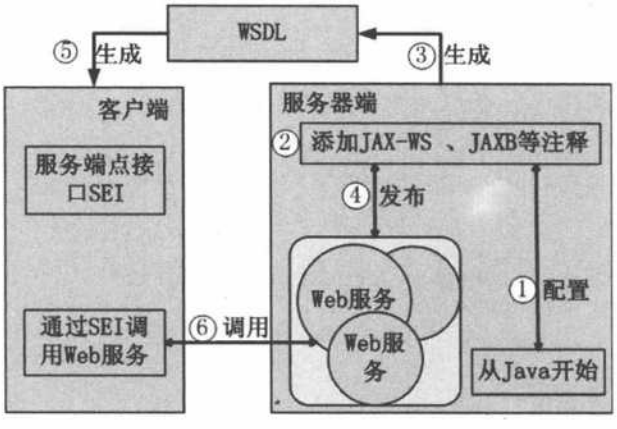


图 6-1　基于代码优先（Java First）开发 Web Services 的步骤图

6.2.1.1　创建 SEI

服务端点接口（Service Endpoint Interface，SEI）是一堆 Java 代码，SEI 是一个标准的 Java 接口。同时，它定义了一系列接口方法。它也可以定义一些成员字段和常量。该代码共享于客户端和服务端，用于实现两者请求和响应的关系，即客户端通过 SEI 来提交请求，而服务端通过 SEI 来反馈响应。当从 Java 编程开始，开发人员第一步是创建一个 SEI。

在一个 SEI 中，定义方法是为了映射到一个公开操作的服务。SEI 对应 wsdl:portType 元素。SEI 的方法对应 wsdl:portType 元素中的 wsdl:operation 元素。

提示：JAX-WS 定义了一个注释，可以指定一些属于服务但不公开的方法。然而，最好的做法是把这些方法排除在 SEI 之外。

下面例子显示了一个简单 SEI，用于处理后台项目信息的服务。

例 1

```
public interface ProjectManager {
    public  String getProjectName();
    public String getProjectNameFromID(String projectID);
    public String greeting() ;
……
}
```

随后一个步骤就是实现接口。由于 SEI 是一个标准的 Java 接口，实现 SEI 的类也是一个标准的 Java 类。如果启动一个 Java 类，需要修改代码来实现接口，即实现类需要实现 SEI。

下面例子显示了实现上面例子接口的类。

例 2

```
package demo.cxf.webservices;

public class ProjectManagerImpl implements ProjectManager {
    private Project project;

    private Map<String,Project> projectMap;

    public ProjectManagerImpl() {
        project = new ProjectImpl();
        project.setProjectID("Default projectID");
        project.setProjectName("Default projectName");
        project.setProjectDescribe("Default projectDescribe");

        projectMap = new HashMap<String,Project>();
        projectMap.put(project.getProjectID(), project);
    }

    public  String getProjectName(){
        System.out.println("getProjectName() be called");
        return project.getProjectName();
    }

    public String greeting() {
        System.out.println("greeting() be called");
       return "Hello, This is ProjectManager." ;
}
……
}
```

6.2.1.2 代码注释

JAX-WS 要依赖 Java 5 的注释功能。JAX-WS 注释用于指定 SEI 映射到一个服务定义的元数据。注释提供的内容如下：

- 定义服务的目的地命名空间。
- 定义用于请求（request）消息的类名称。
- 定义用于响应（response）消息的类名称。
- 说明操作是否单向操作。
- 说明服务的数据绑定类型。

- 说明可使用的任何自定义异常类名称。
- 定义该类型下服务使用的命名空间。

提示：大多数明智处理是采用默认注释，不需要做出具体说明。然而，通过手工注释可以为发布的服务提供更多信息。

建立一个 SEI，必须同时在 SEI 的实现类上添加@WebService 注释。

1. @ WebService 注释

@WebService 注释是 javax.jws.WebService 接口的定义，这是一个接口或类，目的是说明该接口是作为服务来使用的。@WebService 属性说明见表 6-1。

表 6-1 @WebService的属性和说明

属性	说明
name	指定服务接口的名称。此属性映射到wsdl:portType元素的属性名称，而wsdl:portType是在WSDL协议中定义的服务接口。该属性默认值是实现类名称后面追加PortType
targetNamespace	指定服务定义的目标命名空间。如果未指定此属性，目标命名空间来自于包名称
serviceName	指定发布的服务名称。此属性映射到wsdl:service元素的属性名称，而wsdl:service定义公布的服务。该属性默认值是该服务实现类的名称。注意：在SEI是不容许的
wsdlLocation	指定该服务WSDL协议保存的URI。该属性默认值是该服务部署的URI
endpointInterface	指定实现类中SEI的全名。此属性仅用于当服务属性是一个实现类时。注意：在SEI是不容许的
portName	指定该服务发布端点的名称。此属性映射到wsdl:port元素的属性名称，它指定一个公布的服务端点的详细信息。该属性默认值是实现类名称后面追加Port。注意：在SEI是不容许的

提示：开发过程中不需要提供@WebService 注释中属性的任何值。不过，实际编程中建议提供尽可能多的信息。

（1）注释 SEI

SEI 要求添加@WebService 注释。由于 SEI 是协议（contract）定义的服务，应该在@WebService 注释指定更多关于 Web Services 的细节信息。

下面例子显示了用@WebService 注释来定义例子 1 中的接口。

例 3

```
package com.iona.demo;

import javax.jws.WebResult;
import javax.jws.WebService;

@WebService(targetNamespace = "http://webservices.cxf.demo/", name = "ProjectManager",
 wsdlLocation="http://somewhere.com/ ProjectManagerService?wsdl")
public  String getProjectName();
    public String getProjectNameFromID(String projectID);
    public String greeting() ;
……
}
```

在例 3 中@WebService 注释执行以下操作：

- 指定 wsdl:portType 元素的 name 属性值是 ProjectManager，这也是定义的服务接口。
- 指定服务的目标命名空间是 http://webservices.cxf.demo/ 。
- 指定该服务将使用预定义的 WSDL 协议，地址为：http://somewhere.com/ProjectManagerService?wsdl。

（2）注释 SEI 服务实现

除了使用@WebService 注释 SEI，开发者还必须使用@WebService 注释来注释服务实现类。当添加注释到服务实现类时，开发者只需要指定 endpointInterface 属性，如例 4 所示，endpointInterface 属性要设置 SEI 全名。

例 4

```
package org.apache.cxf;

import javax.jws.*;

@WebService(serviceName = "ProjectManagerImplService",
portName = "ProjectManagerImplPort",
targetNamespace = "http://webservices.cxf.demo/",
wsdlLocation="http://somewhere.com/ ProjectManagerService?wsdl",
endpointInterface = "demo.cxf.webservices.ProjectManager")
public class ProjectManagerImpl implements ProjectManager {
private static final Logger LOG = Logger.getLogger(ProjectManagerImpl.
class.getName());
……

}
```

在例 4 中@WebService 注释执行以下操作：

- 指定 serviceName 属性值是 ProjectManagerImplService。
- 指定 portName 属性值是 ProjectManagerImplPort。
- 指定服务的目标命名空间是 http://webservices.cxf.demo/ 。
- 指定该服务将使用预定义的 WSDL 协议，地址为：http://somewhere.com/ProjectManagerService?wsdl。
- 指定 SEI 全名是 demo.cxf.webservices.ProjectManager，即 endpointInterface 属性。

（3）可选择的注释

虽然@WebService 注释足够使一个 Java 接口或 Java 类成为一个服务，但@WebService 注释不提供有关服务的其他信息，例如，如何暴露一个端点。JAX-WS 编程模型使用可选注释来增加其他服务功能，如绑定等。可以添加这些注释到 Web Services 的 SEI 中。

提示：在 SEI 中，如果开发者提供越多的细节，就越容易实现其他更多的复杂功能定义，同时这还将生成更好的 WSDL 协议。

（4）用注释定义绑定属性

若服务使用的是 SOAP 绑定，开发者可以使用 JAX-WS 注释来指定绑定属性的数量。这些

JAX-WS 注释的属性可直接映射到 WSDL 协议中，指定一个服务属性。

2. @SOAPBinding 注释

@SOAPBinding 注释由 javax.jws.soap.SOAPBinding 接口来定义。它提供有关 SOAP 绑定的相关详细信息。如果@SOAPBinding 注释没有指定，服务一般使用 doc/literal 的 SOAP 绑定。

开发人员可以在 SEI 和 SEI 的任何一种方法上放置@SOAPBinding 注释。当一个方法被@SOAPBinding 注释后，该方法的@ SOAPBinding 注释将被设置为优先处理。

表 6-2 显示@SOAPBinding 注释的属性说明。

表 6-2 @SOAPBinding注释属性和说明

属性	属性值	说明
style	Style.DOCUMENT (default) Style.RPC	指定SOAP消息的样式。如果指定为RPC风格，SOAP主体中的每个消息部分都是一个参数或返回值，并会出现在soap:body元素的一个包装元素内。在包装元素内的消息部分对应的操作参数必须与出现在操作的参数顺序相同。如果指定DOCUMENT风格，在SOAP主体内容必须是一个有效的XML文件，但对XML文件的形式没有严格的限制
use	Use.LITERAL (default) Use.ENCODED	指定SOAP消息中的数据如何被传输
parameterStyle	ParameterStyle.BARE ParameterStyle.WRAPPED (default)	指定WSDL协议消息的方法参数是如何被映射到SOAP消息体上的。BARE参数样式意味着每个参数将作为根消息的子元素被放置到消息体中。WRAPPED参数样式意味着所有的输入参数被包装成一个请求消息的单个元素，而输出参数都被包装成一个响应消息的单个元素。如果设置成为RPC样式必须使用WRAPPED参数样式

例 5 显示 SEI 使用 rpc/literal 的 SOAP 消息。

例 5

```
import javax.jws.*;
import javax.jws.soap.*;
import javax.jws.soap.SOAPBinding.*;

@WebService(name="ProjectManager")
@SOAPBinding(style=Style.RPC, use=Use.LITERAL)
public interface ProjectManager{
  ...
}
```

当运行时，Java 定义的方法映射到 XML 操作定义方法，例如：

在 XML 中是如何实现交换信息的。

消息是否可以作为一路消息来进行优化。

定义消息的命名空间的位置。

3. @WebMethod 注释

@WebMethod 注释是由 javax.jws.WebMethod 接口定义的。它被放置在 SEI 的方法上。

@WebMethod 注释提供了通常在 wsdl:operation 元素描述相关的操作方法。

表 6-3 描述了@WebMethod 注释的属性。

表 6-3 @ WebMethod注释的属性和说明

属性	说明
operationName	指定相关wsdl:operation元素名称的属性值。默认值是方法名称
action	指定由方法生成soap:operation元素内soapAction的属性值。默认值是一个空字符串
exclude	如果该方法被排除在指定的服务接口。默认为false

4. @RequestWrapper 注释

@RequestWrapper 注释由 javax.xml.ws.RequestWrapper 接口来定义。@RequestWrapper 注释放置在了 SEI 的方法上。顾名思义，@RequestWrapper 指定的实现包装 Bean 的 Java 类请求的方法参数，该方法参数是由一个远程调用发送请求消息中包含的参数。@RequestWrapper 还可用于在运行时进行请求消息编组和解组过程中指定元素的名称和命名空间。

表 6-4 描述了@RequestWrapper 注释的属性。

表 6-4 @RequestWrapper注释属性和说明

属性	描述
localName	指定请求消息XML表示形式的包装元素的本地名称。默认值是方法或@ WebMethod注释operationName属性值
targetNamespace	指定在XML包装元素定义下的命名空间。默认值是SEI的目标命名空间
className	指定实现包装元素的Java类的全名

提示：只有 className 属性是必需的，其他属性都是可选项。

5. @ResponseWrapper 注释

@ResponseWrapper 注释由 javax.xml.ws.ResponseWrapper 接口来定义。它放置在 SEI 方法上。顾名思义，@ResponseWrapper 指定实现包装 Bean 的 Java 类响应的方法参数，该方法参数由一个远程调用发送响应消息中包含的参数。它还可用于在运行时进行响应消息编组和解组过程中指定元素的名称和命名空间。

表 6-5 描述@ResponseWrapper 注释属性。

表 6-5 @ResponseWrapper注释属性和说明

属性	描述
localName	指定在响应消息XML表示形式的包装元素的本地名称。默认值是方法或@WebMethod注释operationName属性值
targetNamespace	指定在XML包装元素定义下的命名空间。默认值是SEI的目标命名空间
className	指定实现包装元素的Java类全名

提示：只有 className 属性是必需的，其他属性都是可选项。

6. @WebFault 注释

@WebFault 注释由 javax.xml.ws.WebFault 接口定义，这是 SEI 抛出的异常。@WebFault 注释

用来映射 Java 异常到 wsdl:fault 元素。此信息用于编组异常到一个代理，该代理可以由服务的创建者和消费者来进行意外处理。

表 6-6 描述@WebFault 注释属性。

表 6-6　@WebFault注释属性和说明

属性	说明
name	指定异常故障元素的本地名称
targetNamespace	指定异常故障元素定义下的命名空间。默认值是SEI的目标命名空间
faultName	指定实现异常的Java类的全名

提示：name 属性是必需的，其他属性都是可选项。

faultName 由 Sun 的 Java EE 5 SDK 的 beanName 在 http://java.sun.com/javaee/5/docs/api/javax/xml/ws/WebFault.html 上定义。

7. @Oneway 注释

@Oneway 注释由 javax.jws.Oneway 接口定义。@Oneway 注释放置在 SEI 不需要的服务响应的方法。@Oneway 注释说明在此方法执行过程中不需要等待，不保留任何资源来处理响应，这样的优势在于可以进行性能优化。

例 6 显示 SEI 及其方法被注释的方式。

例 6

```
package org.apache.cxf;

import javax.jws.*;
import javax.xml.ws.*;

@WebService(name="quoteReporter")
public interface QuoteReporter
{
  @WebMethod(operationName="getStockQuote")
  @RequestWrapper(targetNamespace="http://demo.iona.com/types",
                  className="java.lang.String")
  @ResponseWrapper(targetNamespace="http://demo.iona.com/types",
                   className="org.eric.demo.Quote")
  public Quote getQuote(String ticker);
}
```

SEI 方法参数映射到 wsdl:message 元素和 wsdl:part 元素。JAX-WS 提供注释让开发人员描述 wsdl:part，该 wsdl:part 为方法生成参数。

8. @WebParam 注释

@WebParam 注释由 javax.jws.WebParam 接口定义，放置在 SEI 定义的方法参数。如果 @WebParam 参数放置在 SOAP 标头或者 wsdl:part 生成的其他属性，@WebParam 注释允许开发人员指定参数的方向。

表 6-7 描述@WebParam 注释属性。

表 6-7 @WebParam注释属性、属性值和说明

属性名称	属性值	说明
name		指定参数的名称，因为它在WSDL中。对于RPC绑定，这是在wsdl:part代理参数。对于文件绑定，这是代表参数的XML元素的本地名称。按照JAX-WS规格定义，默认为arg*N*，其中*N*是从零开始的参数指数（即arg0、arg1等代替）
targetNamespace		指定参数的命名空间。只有在使用文件绑定时，参数映射到XML元素。默认值是使用该服务的命名空间
mode	Mode.IN (default) Mode.OUT Mode.INOUT	指定参数的方向
header	false (default) true	指定参数是否作为SOAP头的一部分
partName		当绑定文档时，指定作为参数的wsdl:part元素的名称属性值

9. @WebResult 注释

@WebResult 注释由 javax.jws.WebResult 接口定义。它被放置在 SEI 定义的方法。@WebResult 注解可以指定生成的 wsdl:part 属性，wsdl:part 部分是该方法的返回值生成的。

表 6-8 描述@WebResult 注释属性。

表 6-8 @WebResult注释属性和描述

属性	描述
name	指定返回值的名称，因为它在WSDL中。为RPC绑定，这是在wsdl:part代表返回值。对于文件绑定，这是代表返回的XML值的元素的本地名称。默认值是返回值
targetNamespace	指定返回值的命名空间。只有在使用文件绑定时，其参数才映射到XML元素。默认值是使用该服务的命名空间
header	指定返回值是否作为SOAP头的一部分
partName	当绑定文档时，指定作为参数的wsdl:part元素的名称属性值

例 7 充分显示了 SEI 的注释。

例 7

```
package org.apache.cxf;

import javax.jws.*;
import javax.xml.ws.*;
import javax.jws.soap.*;
import javax.jws.soap.SOAPBinding.*;
import javax.jws.WebParam.*;

@WebService(name=" ProjectManager")
@SOAPBinding(style=Style.RPC, use=Use.LITERAL)
public interface ProjectManager {

@WebMethod(operationName=" updateProjectName ")
@ResponseWrapper(localName = "getProjectNameFromIDResponse",
targetNamespace = "http://webservices.cxf.demo/",
className = "demo.cxf.webservices.GetProjectNameFromIDResponse")
@RequestWrapper(localName = "getProjectNameFromID",
targetNamespace = "http://webservices.cxf.demo/",
```

```
className = "demo.cxf.webservices.GetProjectNameFromID")
@WebResult(name = "return", targetNamespace = "")
 public java.lang.String getProjectNameFromID(
       @WebParam(name = "arg0", targetNamespace = "")
       java.lang.String arg0
   );
}
```

一旦注释了代码，就可以使用 java2wsdl 命令为服务生成一个 WSDL 协议。

6.2.1.3　发布 Web Services 服务

发布 Web Services 服务，就是在服务器上注册 Service 组件。Web Services 客户端可以通过 HTTP 等协议访问 Service 组件暴露出来的服务方法。

下面的例子演示了 Web Services 服务器的启动：

```
package demo.cxf.frontends.javafirst.server;

import javax.xml.ws.Endpoint;
import demo.cxf.webservices.*;

public class Server {

    public Server() {
        System.out.println("Starting Server");
        ProjectManager projectManager = new ProjectManagerImpl();
        String address = "http://localhost:9000/ProjectManager";
        Endpoint.publish(address, projectManager);
    }

    public static void main(String[] args) throws Exception {
        new Server();
        System.out.println("Server ready...");

        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

发布类的静态方法提供了一种可以方便地发布和测试的 JAX-WS 服务的方法。该方法使用的端点 URL 地址和一个 projectManager 对象作为参数。ProjectManager 服务将公布端点地址。发布在该方法创建一个轻量级的 Web 服务器的 URL（http://localhost:9000/ProjectManager）和部署到该位置的服务。轻量级的 Web Services 服务器在运行 5 分钟后，会自动退出应用状态。

通过以下网址在 Web 浏览器上可以查看该服务的 WSDL 内容。

```
http://localhost:9000/ProjectManager?wsdl
```

6.2.1.4　开发 Web Services 客户端程序

开发一个客户端类，将查找 ProjectManager 服务，并调用其 getDefaultProject 方法。下面的代码说明消费者的服务组件：

```
package demo.cxf.frontends.javafirst.client;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.soap.SOAPBinding;

import demo.cxf.webservices.Project;
import demo.cxf.webservices.ProjectImpl;
import demo.cxf.webservices.ProjectManager;

public class Client {

    private static final QName SERVICE_NAME = new QName(
            "http://webservices.cxf.demo/", "ProjectManager");
    private static final QName PORT_NAME = new QName(
            "http://webservices.cxf.demo/", "ProjectManagerPort");
    private static String endpointAddress = "http://localhost:9000/ProjectManager";

    public Client() {}

    public static void main(String[] args) throws Exception {
        System.out.println("Starting Client");

        //用服务名称创建一个服务
        Service projectService = Service.create(SERVICE_NAME);

        //用服务端口名称、绑定方式和 Endpoint 地址来创建服务器端口
        projectService.addPort(PORT_NAME, SOAPBinding.SOAP11HTTP_BINDING,
                endpointAddress);

       //获得 SEI 接口
        ProjectManager projectManager = projectService.getPort(ProjectManager.class);
        System.out.println(projectManager.getProjectName());

        Project defaultProject = projectManager.getDefaultProject();
        System.out.println("defaultProjectID is " + defaultProject.getProjectID()
                + " . " + "defaultProjectName is " + defaultProject.getProjectName());

        Project project = new ProjectImpl();
        project.setProjectID("New ProjectID");
        project.setProjectName("New ProjectName");

        projectManager.setDefaultProject(project);
        Project newProject = projectManager.getDefaultProject();
        System.out.println("NewProjectID is " + newProject.getProjectID()
                + " . " + "NewProjectName is " + newProject.getProjectName());

        projectManager.registerProject(project);
        System.out.println(projectManager.getProjectNameByID("Default projectID"));
        System.out.println(projectManager.getProjectNameByID("New ProjectID"));
        System.exit(0);
    }
}
```

Web Services 客户端程序实现的功能包括：

（1）程序首先创建 WSDL 的网址。该 WSDL URL 是“http://localhost:9000/ ProjectManager?wsdl”，标志着该网址的 WSDL 文档位置。

（2）程序创建一个 Service 对象。Service 对象使用静态 create 方法。该方法传入 WSDL URL 和服务名称作为参数。服务名称 projectService 是一个 QName，并映射到 WSDL 文档<wsdl:service>元素，该<wsdl:service>元素定义服务端点。

（3）使用 Service 对象，通过调用的 getPort 方法获得 SEI 的代理组件。该 getPort 方法传入端口名称和 SEI 类作为参数。端口名称 PORT_NAME 是一个 QName 并映射到 WSDL 文档的<wsdl:port>元素，而 SEI 类是 ProjectManager。

（4）该代理组件调用服务方法 getDefaultProject。调用该方法之前，将 Project 对象作为返回值。该方法在服务器上调用并返回。

6.2.2　基于代码优先（Java First）的 Web Services 的例子

基于代码优先（Java First）的 Web Services 有 5 个例子。这些例子都基于 JAX-WS 规范来实现 Web Services 的调用。

第一个例子是采用 POJO 的前端实现的例子，主要是采用 POJO 对象作为中间传输对象，Web Services 服务器用内嵌服务器。第二个例子是采用 POJO 并基于 Servlet 容器的前端实现的例子，主要采用 POJO 对象作为中间传输对象，但 Web Services 服务器用外置的 Servlet 服务器。第三个例子是采用 Map 的前端实现，主要采用 HashMap 对象作为中间传输对象，Web Services 服务器用内嵌服务器。第四个例子是采用 Map 并基于 Servlet 容器的前端实现，主要采用 HashMap 对象作为中间传输对象，但 Web Services 服务器用外置的 Servlet 服务器。第五个例子是基于定制注释的 JAX-WS 的例子。

6.2.2.1　采用 POJO 并基于 JAX-WS 服务的前端实现

本程序演示 Apache CXF 遵循 JAX-WS 规范的客户端和服务器端调用，客户端采用 Apache CXF 的代码优先（Java First）前端应用模式。采用 POJO 对象来作为中间传输对象。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时候默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，都是使用 POJO 对象实现客户端和服务器端的数据交换。后台服务端为遵循 JAX-WS 规范的 Endpoint 来实现。前端访问采用三种方式，分别是：① 采用 JAX-WS 规范的 Service 方式；② 采用 Apache CXF 本身的客户端来实现；③ 采用 Spring IoC 注入的客户端来实现。

采用 JAX-WS 规范的 Service 方式，实现的结构如图 6-2 所示。

其中：① 表示客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。客户端和服务端的 SEI 都是 ProjectManager 接口。

还有一种方式是采用 Apache CXF 本身的客户端来实现，实现如图 6-3 所示。

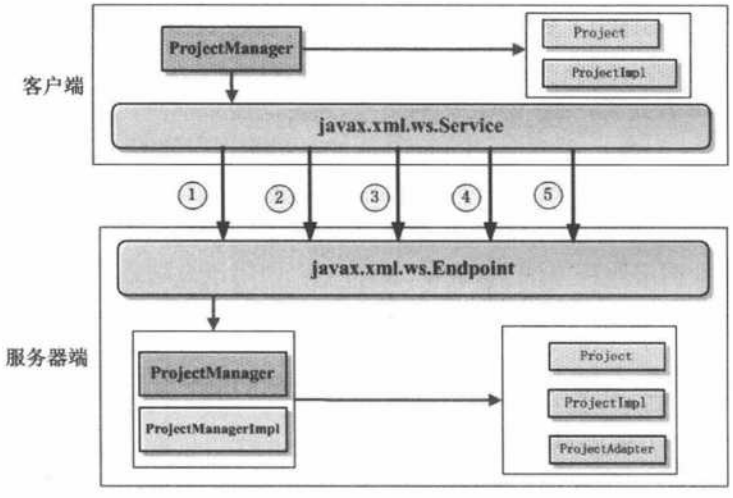


图 6-2　基于 POJO 实现 JAX-WS 前端应用模式图（1）

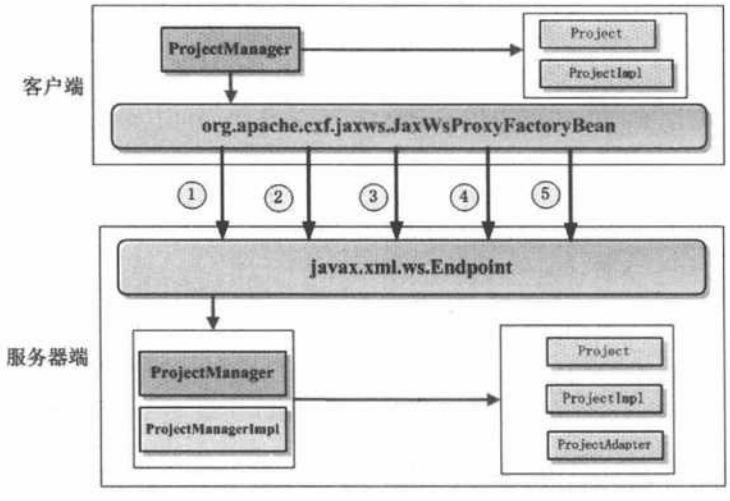


图 6-3　基于 POJO 实现 Apache CXF 前端应用模式图（2）

其中：① 表示客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。客户端和服务端的 SEI 都是 ProjectManager 接口。

采用 Spring IoC 注入的客户端来实现，实现如图 6-4 所示。

其中：① 表示客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。客户端和服务端的 SEI 都是 ProjectManager 接口。

2. 应用程序案例实现步骤

要创建 POJO 并基于 JAX-WS 服务的前端实现 Web Services 程序的流程步骤如图 6-5 所示。

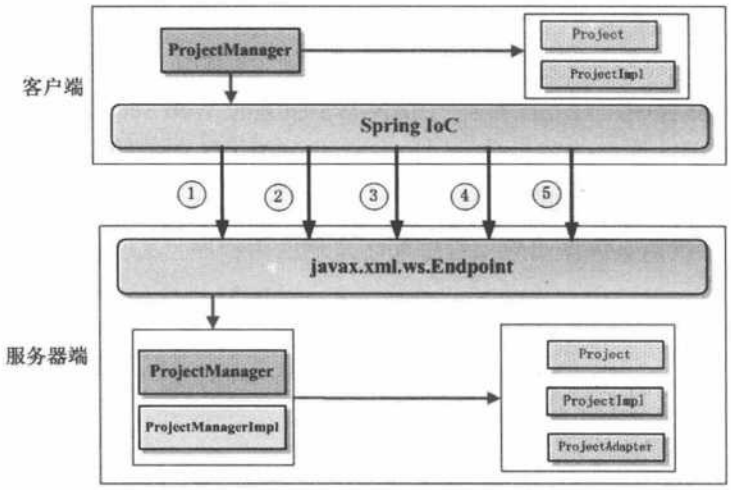


图 6-4　基于 POJO 实现 Apache CXF 前端应用模式图（3）

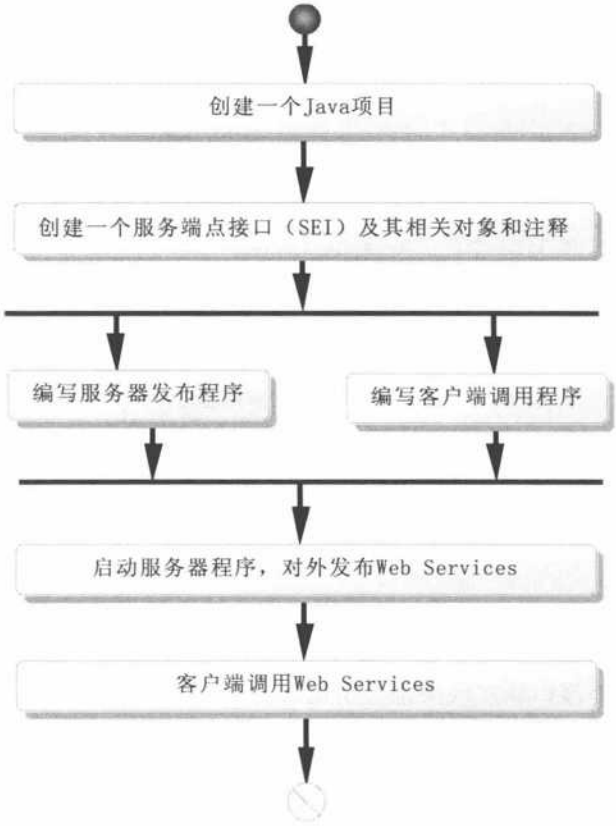


图 6-5　应用程序案例实现步骤流程图

流程图说明，要实现下面一些步骤：

（1）创建一个 Java 项目。

（2）创建一个基于 JAX-WS 服务的服务端点接口（SEI）及其相关对象和注释。

（3）编写发布这个服务端点接口（SEI）为 Web Services 的服务端程序，该服务器启动程序

支持 JAX-WS 服务。

（4）编写基于 JAX-WS 服务的 Web Services 的客户端应用程序，采用 POJO 作为数据传输对象。

（5）启动服务器程序，对外发布基于 JAX-WS 服务的 Web Services。

（6）基于 JAX-WS 服务的客户端调用后台基于 JAX-WS 服务的 Web Services。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 6-6 所示。

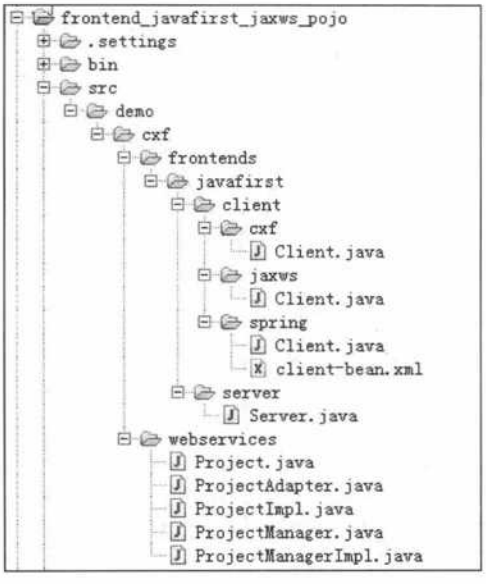


图 6-6　frontend_javafirst_jaxws_pojo 程序结构

包目录包括三个部分，第一个包目录 demo.cxf.frontends.javafirst.server 是服务器程序目录，第二个包目录 demo.cxf.frontends.javafirst.client 是客户端程序目录。第三个包目录 demo.cxf.webservices 是服务器端点接口 SEI 和相关对象等公用部分，服务器端和客户端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口，服务器端和客户端都必须引用，但服务器端和客户端对公用部分的引用是不同的。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口，而服务器端程序是接口和实现类都要引用。

下面分别介绍三部分：①服务器端点接口 SEI 和相关对象；②服务器端发布程序；③客户端调用程序。

（1）服务器端点接口（SEI）和相关对象

首先要介绍的是服务器端点接口（SEI）。

服务器端点接口（SEI）名称为 ProjectManager，其 Java 代码如下：

```
package demo.cxf.webservices;

import javax.jws.WebService;
import javax.jws.WebParam;
```

```
@WebService
public interface ProjectManager {
    public String greeting() ;
    public String greetingWithParameter(@WebParam(name = "greet")String greet);
    public String getProjectName();
    public String getProjectNameByID(String projectID);
    public void updateProjectName (@WebParam(name = "projectNam")String projectName);
    public void setDefaultProject(Project project);
    public void registerProject(Project project);
    public void removeProject(String id);
    public Project getDefaultProject();
    public void getAllProject();
}
```

在上述的代码中，本案例程序创建了一个名为 ProjectManager 的服务器端点接口（SEI）。这个 ProjectManager 接口实际上就是一个 POJO 接口。服务上有注释，表明这是一个 Web Services 接口。在这个接口中定义了 getProjectName、greeting 等多个方法。通过这些方法传入一个 Project 对象的参数。

下面是实现这个接口的 Java 实现类 ProjectManagerImpl，代码如下。

```
package demo.cxf.webservices;

import java.util.HashMap;
import java.util.Map;

import javax.jws.WebService;

@WebService
public class ProjectManagerImpl implements ProjectManager {

    private Project project;

    private Map<String, Project> projectMap;

    public ProjectManagerImpl() {
        System.out.println("----ProjectManagerImpl Construct----");
        initProject();
    }

    public String getProjectName() {
        System.out.println("----invoking getProjectName() ----");
        return project.getProjectName();
    }

    public String greeting() {
        System.out.println("----invoking greeting() ----");
        return "Hello, This is ProjectManager.";
    }

    public void updateProjectName(String projectName) {
        System.out.println("----invoking updateProjectName----");
        project.setProjectName(projectName);
    }

    public String greetingWithParameter(String greet) {
        System.out.println("----invoking greetingWithParameter be called");
```

```
        return "Hello, " + greet + " This is ProjectManager.";
    }

    public void setDefaultProject(Project project) {
        System.out.println("----invoking setDefaultProject----");
        if (project == null)
            return;
        this.project = project;
    }

    public Project getDefaultProject() {
        System.out.println("----invoking getDefaultProject----");
        return project;
    }

    public void registerProject(Project project) {
        System.out.println("----invoking registerProject----");
        if (projectMap.containsKey(project.getProjectID()))
            return;
        projectMap.put(project.getProjectID(), project);
    }

    public String getProjectNameByID(String projectID) {
        System.out.println("----invoking getProjectNameByID----");
        if (!projectMap.containsKey(projectID))
            return null;
        return projectMap.get(projectID).getProjectName();
    }

    public Project getProjectByID(String projectID) {
        System.out.println("----invoking getProjectByID ----");
        if (!projectMap.containsKey(projectID))
            return null;
        return projectMap.get(projectID);
    }

    public void removeProject(String id) {
        System.out.println("----invoking removeProject, Project ID is: " + id);
        if (!projectMap.containsKey(id))
            return;
        projectMap.remove(id);
    }

    public Map<String, Project> getProjectMap() {
        System.out.println("getProjectMap() be called");
        return projectMap;
    }

    public void getAllProject(){
        System.out.println("----invoking getAllProject ---- ");
        for (Map.Entry<String, Project> currentProject : projectMap.entrySet()) {
            System.out.println("  " + currentProject.getKey() + ": "
                    + currentProject.getValue().getProjectName());
        }
    }

    private void initProject() {
        project = new ProjectImpl();
        project.setProjectID("default projectID");
```

```
        project.setProjectName("default projectName");
        project.setProjectDescribe("default projectDescribe");

        projectMap = new HashMap<String, Project>();

        Project project2 = new ProjectImpl();
        project2.setProjectID("projectID2");
        project2.setProjectName("projectName2");
        project2.setProjectDescribe("projectDescribe2");

        projectMap.put(project.getProjectID(), project);
        projectMap.put(project2.getProjectID(), project2);
    }

}
```

实现类 ProjectManagerImpl 首先要标识这个 Web Services 服务类，然后才实现接口的所有方法。

在服务中的 ProjectManager 接口和 ProjectManagerImpl 实现类都使用对象 Project 接口和 ProjectImpl 类。ProjectImpl 类实现了 Project 接口。为了向系统通知这两者的关系，需要加入一个进行解析的对象类 ProjectAdapter，该类继承 XmlAdapter 对象，并实现 marshal 和 unmarshal 两个方法。

Project 接口的代码如下：

```
package demo.cxf.webservices;

import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;

@XmlJavaTypeAdapter(ProjectAdapter.class)
public interface Project {
    public String getProjectName() ;
    public void setProjectName(String projectName) ;
    public String getProjectID() ;
    public void setProjectID(String projectID) ;
    public String getProjectDescribe() ;
    public void setProjectDescribe(String projectDescribe) ;
}
```

ProjectImpl 实现类的程序代码如下：

```
package demo.cxf.webservices;

import javax.xml.bind.annotation.XmlType;

@XmlType(name = "Project")
public class ProjectImpl implements Project {

    private String projectName ;
    private String ProjectID;
    private String ProjectDescribe;

    public ProjectImpl() {}

    public String getProjectName() {return projectName;}
    public void setProjectName(String projectName) {this.projectName = projectName;}
```

```
    public String getProjectID() { return ProjectID;}
    public void setProjectID(String projectID) {   ProjectID = projectID;}
    public String getProjectDescribe() {return ProjectDescribe;}
    public  void  setProjectDescribe(String  projectDescribe)  {ProjectDescribe  =
projectDescribe;}
}
```

ProjectAdapter 类代码如下：

```
package demo.cxf.webservices;

import javax.xml.bind.annotation.adapters.XmlAdapter;

public class ProjectAdapter extends XmlAdapter<ProjectImpl, Project>{

    public ProjectImpl marshal(Project p) throws Exception {
       if (p instanceof ProjectImpl) {
          return (ProjectImpl)p;
       }
       return new ProjectImpl();
    }

    public Project unmarshal(ProjectImpl p) throws Exception {
       return p;
    }
}
```

（2）服务器端应用程序

服务器端应用程序主要包括 Project 组件和服务器启动类。Project 组件包括 WebService 包下的几个 Project 文件和 Server.java 文件。其中 Server.java 是服务器启动程序文件。

编写发布这个 Java 程序为 Web Services 的服务端程序。创建了 SEI，要把这个 SEI 发布出去，还需要一个部署程序，即服务端的发布程序，本案例采用 JAX-WS 规范来编写服务端程序。创建一个 Server 类，启动这个服务，接着就可以发布 SEI 服务了。

```
package demo.cxf.frontends.javafirst.server;

import javax.xml.ws.Endpoint;
import demo.cxf.webservices.*;

public class Server {

    public Server() {
        System.out.println("Starting Server");
        ProjectManager projectManager = new ProjectManagerImpl();
        String address = "http://localhost:9000/ProjectManager";
        Endpoint.publish(address, projectManager);
    }

    public static void main(String[] args) throws Exception {
        new Server();
        System.out.println("Server ready...");

        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

在本案例中，服务器启动的服务地址是 http://localhost:9000/ProjectManager。服务器引入了 SEI 接口——ProjectManager 接口。这样 ProjectManager 的 Web Services 就暴露到了外部。可以在浏览器上输入 http://localhost:9000/ProjectManager 来查看这个服务的内容，如下所示：

```
<?xml version='1.0' encoding='UTF-8'?>
<wsdl:definitions name="ProjectManagerImplService"
    targetNamespace="http://webservices.cxf.demo/"
    xmlns:ns1="http://schemas.xmlsoap.org/soap/http"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://webservices.cxf.demo/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <wsdl:types>
    ....
    </wsdl:service>
</wsdl:definitions>
```

（3）客户端应用程序

编写 Web Services 的客户端应用程序。客户端的核心实现部分主要有三种方式，第一种是基于 JAX-WS 的实现模式，第二种是采用 Apache CXF 模式，第三种采用 Spring IoC 注入模式。

第一种基于 JAX-WS 的实现模式是图 6-6 中的 demo.cxf.frontends.javafirst.client.jaxws 下的 Client.java 文件。

JAX-WS 实现模式的客户端程序代码如下：

```
package demo.cxf.frontends.javafirst.client;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.soap.SOAPBinding;

import demo.cxf.webservices.Project;
import demo.cxf.webservices.ProjectImpl;
import demo.cxf.webservices.ProjectManager;

public class Client {

    private static final QName SERVICE_NAME = new QName("http://webservices.cxf.demo/",
"ProjectManager");
    private static final QName PORT_NAME = new QName("http://webservices.cxf.demo/",
"ProjectManagerPort");
    private static String endpointAddress = "http://localhost:9000/ProjectManager";


    public Client() {   }

    public static void main(String[] args) throws Exception {
        System.out.println("Starting Client");

        //用服务名称创建一个服务
        Service service = Service.create(SERVICE_NAME);

        //用服务端口名称、绑定方式和 Endpoint 地址来创建服务器端口
        service.addPort(PORT_NAME, SOAPBinding.SOAP11HTTP_BINDING, endpointAddress);
```

```
        //获得 SEI 接口
        ProjectManager projectManager = service.getPort(ProjectManager.class);
        System.out.println(projectManager.getProjectName());

        //项目的基本情况
        getGreet(projectManager);
        greetingWithParameter(projectManager);
        getProjectName(projectManager);
        getDefaultProject(projectManager);
        getAllProject(projectManager);

        //修改默认项目，查看项目的基本情况
        setDefaultProject(projectManager);

        //增加一个项目，查看项目的基本情况
        registerProject(projectManager);

        //删除一个项目，查看项目的基本情况
        removeProject(projectManager);
        System.exit(0);
    }

    //调用 SEI 的 getGreet 方法
    private static void getGreet(ProjectManager projectManager)throws Exception {
        System.out.println(projectManager.greeting());
    }

    //调用 SEI 的 greetingWithParameter 方法
    private static void greetingWithParameter(ProjectManager projectManager)throws
Exception {
        System.out.println(projectManager.greetingWithParameter("greet"));
    }

    //调用 SEI 的 getProjectName 方法
    private static void getProjectName(ProjectManager projectManager)throws Exception {
        System.out.println(projectManager.getProjectName());
    }

    //调用 SEI 的 getDefaultProject 方法
    private static void getDefaultProject(ProjectManager projectManager)throws
Exception {
        Project defaultProject = projectManager.getDefaultProject();
        System.out.println("defaultProjectID is " + defaultProject.getProjectID()
                + " , " + "defaultProjectName is " + defaultProject.getProjectName());
    }

    //调用 SEI 的 setDefaultProject 方法
    private static void setDefaultProject(ProjectManager projectManager)throws
Exception {
        Project project = new ProjectImpl();
        project.setProjectID("New Default ProjectID");
        project.setProjectName("New Default ProjectName");

        projectManager.setDefaultProject(project);
        Project newProject = projectManager.getDefaultProject();
        System.out.println("DefaultProjectID is " + newProject.getProjectID()
                + " , " + "DefaultProjectName is " + newProject.getProjectName());
```

```
        projectManager.getAllProject();
    }

    //调用 SEI 的 registerProject 方法
    private  static  void  registerProject(ProjectManager  projectManager)throws
Exception {
        Project project = new ProjectImpl();
        project.setProjectID("New ProjectID");
        project.setProjectName("New ProjectName");

        projectManager.registerProject(project);
        System.out.println(projectManager.getProjectNameByID("New ProjectID"));
        System.out.println(projectManager.getProjectNameByID("New ProjectID"));
        projectManager.getAllProject();
    }

    //调用 SEI 的 removeProject 方法
    private static void removeProject(ProjectManager projectManager)throws Exception {
        projectManager.removeProject("New ProjectID");
        projectManager.getAllProject();
    }

    //调用 SEI 的 getAllProject 方法
    private static void getAllProject(ProjectManager projectManager) throws Exception{
        projectManager.getAllProject();
    }
}
```

客户端首先定义了两个 QName 类型变量——SERVICE_NAME 和 PORT_NAME。SERVICE_NAME 表明 Web Services 的服务名称。PORT_NAME 表明 Web Services 的服务端口名称。我们在定义 SEI 时都采用默认方式，所以 SERVICE_NAME 值是“ProjectManager”，而 PORT_NAME 值是“ProjectManagerPort”。http://webservices.cxf.demo/表明了这个服务的命名空间。我们在定义 SEI 中没有指定命名空间，系统就按照程序的包名的反向方式自动生成命名空间。本例中 ProjectManager 接口的包名是“demo.cxf.webservices”，所以 Web Services 默认的命名空间就是 http://webservices.cxf.demo/。

本例首先用服务名称来创建一个 javax.xml.ws.Service 实现类，接着通过 addPort 方法传入服务的端口类、绑定方式和 Web Services 的地址，最后通过 getPort 方法来获得服务器的 SEI 接口。这就是我们需要的 Web Services 服务接口 ProjectManager。后面的工作就是基于这个 ProjectManager 的 SEI 来实现各种业务操作。

测试编写好的程序代码，首先要启动服务器程序，即在 Eclipse 上运行 Server.java，然后运行客户端程序，其客户端程序输出结果如下：

```
default projectName
Hello, This is ProjectManager.
Hello, greet This is ProjectManager.
default projectName
defaultProjectID is default projectID . defaultProjectName is default projectName
DefaultProjectID is New Default ProjectID . DefaultProjectName is New Default
ProjectName
New ProjectName
New ProjectName
```

其服务器端输出结果如下：

```
----invoking getProjectName() ----
----invoking greeting() ----
----invoking greetingWithParameter be called
----invoking getProjectName() ----
----invoking getDefaultProject----
----invoking getAllProject ----
  default projectID: default projectName
  projectID2: projectName2
----invoking setDefaultProject----
----invoking getDefaultProject----
----invoking getAllProject ----
  default projectID: default projectName
  projectID2: projectName2
----invoking registerProject----
----invoking getProjectNameByID----
----invoking getProjectNameByID----
----invoking getAllProject ----
  default projectID: default projectName
  New ProjectID: New ProjectName
  projectID2: projectName2
----invoking removeProject, Project ID is: New ProjectID
----invoking getAllProject ----
  default projectID: default projectName
  projectID2: projectName2
```

上述的所有调用都基于 JAX-WS 规范。

第二种采用 Apache CXF 的实现模式是图 6-6 中的 demo.cxf.frontends.javafirst.client.cxf 下的 Client.java 文件。

Apache CXF 实现模式的客户端程序代码如下：

```
package demo.cxf.frontends.javafirst.client;

import org.apache.cxf.jaxws.JaxWsProxyFactoryBean;
import demo.cxf.webservices.Project;
import demo.cxf.webservices.ProjectImpl;
import demo.cxf.webservices.ProjectManager;

public class CXFClient {
    public CXFClient() {    }

    public static void main(String[] args) throws Exception {
        System.out.println("Starting Client");

        //定义 JaxWs 的代理工厂
        JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();

        //设置 JaxWs 代理工厂的属性，包括服务类和地址
        factory.setServiceClass(ProjectManager.class);
        factory.setAddress("http://localhost:9000/ProjectManager");

        //获取服务类的对象并处理
        ProjectManager projectManager = (ProjectManager) factory.create();

        //项目的基本情况
```

```
        getGreet(projectManager);
        greetingWithParameter(projectManager);
        getProjectName(projectManager);
        getDefaultProject(projectManager);
        getAllProject(projectManager);

        //修改默认项目，查看项目的基本情况
        setDefaultProject(projectManager);

        //增加一个项目，查看项目的基本情况
        registerProject(projectManager);

        //删除一个项目，查看项目的基本情况
        removeProject(projectManager);
        System.exit(0);
    }

    //调用 SEI 的 getGreet 方法
    private static void getGreet(ProjectManager projectManager)throws Exception {
        System.out.println(projectManager.greeting());
    }

    //调用 SEI 的 greetingWithParameter 方法
    private static void greetingWithParameter(ProjectManager projectManager)throws
Exception {
        System.out.println(projectManager.greetingWithParameter("greet"));
    }

    //调用 SEI 的 getProjectName 方法
    private static void getProjectName(ProjectManager projectManager)throws Exception {
        System.out.println(projectManager.getProjectName());
    }

    //调用 SEI 的 getDefaultProject 方法
    private static void getDefaultProject(ProjectManager projectManager)throws
Exception {
        Project defaultProject = projectManager.getDefaultProject();
        System.out.println("defaultProjectID is " + defaultProject.getProjectID()
                + " . " + "defaultProjectName is " + defaultProject.getProjectName());
    }

    //调用 SEI 的 setDefaultProject 方法
    private static void setDefaultProject(ProjectManager projectManager)throws
Exception {
        Project project = new ProjectImpl();
        project.setProjectID("New Default ProjectID");
        project.setProjectName("New Default ProjectName");

        projectManager.setDefaultProject(project);
        Project newProject = projectManager.getDefaultProject();
        System.out.println("DefaultProjectID is " + newProject.getProjectID()
                + " . " + "DefaultProjectName is " + newProject.getProjectName());
        projectManager.getAllProject();
    }

    //调用 SEI 的 registerProject 方法
    private static void registerProject(ProjectManager projectManager)throws
Exception {
```

```
        Project project = new ProjectImpl();
        project.setProjectID("New ProjectID");
        project.setProjectName("New ProjectName");

        projectManager.registerProject(project);
        System.out.println(projectManager.getProjectNameByID("New ProjectID"));
        System.out.println(projectManager.getProjectNameByID("New ProjectID"));
        projectManager.getAllProject();
    }

    //调用 SEI 的 removeProject 方法
    private static void removeProject(ProjectManager projectManager)throws Exception {
        projectManager.removeProject("New ProjectID");
        projectManager.getAllProject();
    }

    //调用 SEI 的 getAllProject 方法
    private static void getAllProject(ProjectManager projectManager) throws Exception{
        projectManager.getAllProject();
    }
}
```

客户端首先实例化一个 JaxWsProxyFactoryBean 对象。设置 JaxWs 代理工厂的属性，包括 SEI 和地址，然后调用 SEI 的方法。

运行客户端程序，其客户端程序和服务器端输出结果与前面介绍的相同。

第三种采用 Spring IoC 注入模式的实现模式是图 6-6 中的 demo.cxf.frontends.javafirst.client.spring 下的 Client.java 文件。

Spring IoC 注入实现模式的客户端程序代码如下：

```
package demo.cxf.frontends.javafirst.client.spring;

import org.springframework.context.support.ClassPathXmlApplicationContext;
……

public class Client {
    public Client() {}

    public static void main(String[] args) throws Exception {
        ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(
                new String[] { "demo/cxf/frontends/javafirst/client/spring/client-bean.xml" });

        // 获得 SEI 接口
        ProjectManager projectManager = (ProjectManager) context.getBean
("ProjectManagerClient");

        // 项目的基本情况
        getGreet(projectManager);
        greetingWithParameter(projectManager);
        ……

        System.exit(0);
    }

    // 调用 SEI 的 getGreet 方法
    private static void getGreet(ProjectManager projectManager)
```

```
            throws Exception {
        System.out.println(projectManager.greeting());
    }
    ……
}
```

客户端首先采用 Spring 获得 Bean，Bean 的配置文件 client-bean.xml 如下：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                    http://www.springframework.org/schema/beans/spring-beans.xsd
                    http://cxf.apache.org/jaxws
                    http://cxf.apache.org/schemas/jaxws.xsd">

    <jaxws:client id="ProjectManagerClient"
        serviceClass="demo.cxf.webservices.ProjectManager"
        address="http://localhost:9000/ProjectManager" />
</beans>
```

设置 JaxWs 代理工厂的属性，包括 SEI 和地址，然后调用 SEI 的方法。

运行客户端程序，其客户端程序和服务器端输出结果与前面的相同。

4. 其他内容

本例代码请参见配套资源代码中“第 06 章\frontend_javafirst_jaxws_pojo”文件夹。

6.2.2.2　采用 POJO 并基于 Servlet 容器的前端实现

上述采用 POJO 的例子主要是基于后台启动的 Web Services 服务，也可以把这个服务委托给 Servlet 容器来进行管理。本例的 Servlet 容器选择 Tomcat。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，都是使用 POJO 对象实现客户端和服务器端的数据交换。后台服务端采用遵循 JAX-WS 规范的 Server 来实现。前端访问采用三种方式，分别是：① 采用 JAX-WS 规范的 Service 方式；② 采用 Apache CXF 本身的客户端来实现；③ 采用 Spring IoC 注入的客户端来实现。

第一种方式是基于 POJO 实现 JAX-WS 前端，Servlet 后端应用模式，如图 6-7 所示。

其中：① 表示标准 JAX-WS 客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。客户端和服务端的 SEI 都是 ProjectManager 接口。

第二种方式是采用 Apache CXF 本身的客户端来实现，如图 6-8 所示。

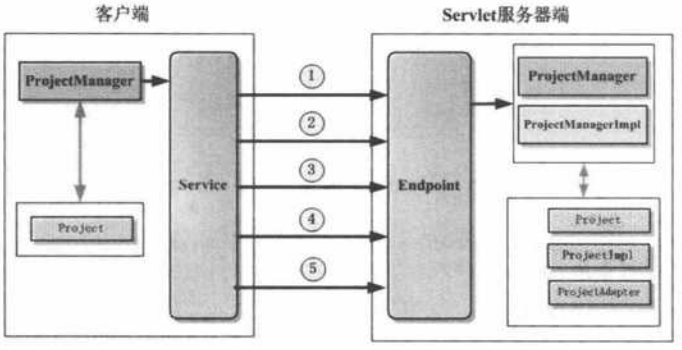


图 6-7　基于 POJO 实现 JAX-WS 前端，Servlet 后端应用模式图

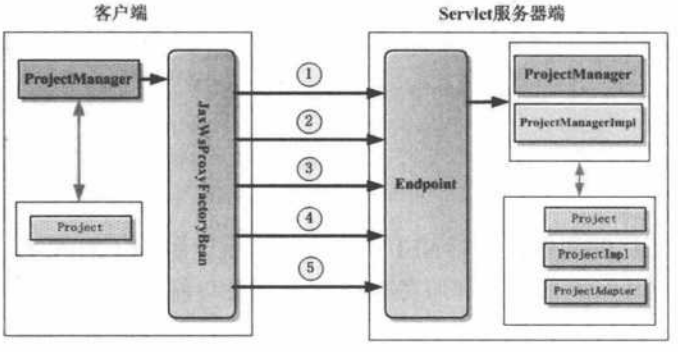


图 6-8　基于 Apache CXF 实现 JAX-WS 前端，Servlet 后端应用模式图

其中：① 表示 Apache CXF 客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。在客户端和服务端的 SEI 都是 ProjectManager 接口。

第三种方式是采用 Spring IoC 注入的客户端来实现，如图 6-9 所示。

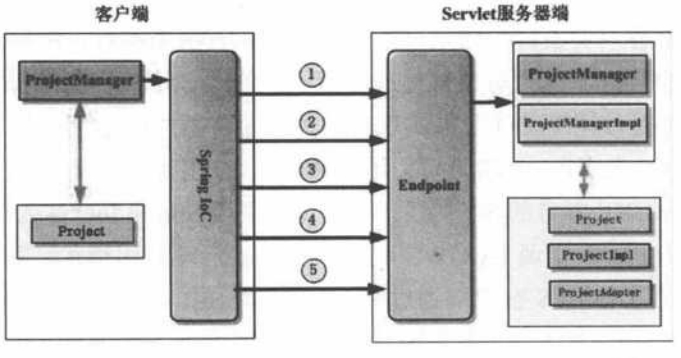


图 6-9　基于 Spring IoC 注入的客户端，Servlet 后端应用模式图

其中：① 表示 Spring IoC 注入的客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目。⑤ 表示客户端通过 SEI 查看后台的全部项目情况。在客户端和服务端的 SEI 都是 ProjectManager 接口。

2. 应用程序案例实现步骤

要创建 POJO 并基于 Servlet 容器的前端实现 Web Services 程序的流程步骤如图 6-10 所示。

（1）创建一个 Web 服务项目。

（2）创建一个服务端点接口（SEI）及其相关对象和注释。

（3）编写 web.xml 和 server-bean.xml。

（4）发布配置文件到 Servlet 服务器。

（5）编写 Web Services 的客户端应用程序。

（6）启动 Servlet 服务器程序，对外发布 Web Services。

（7）客户端调用 Web Services。

其中第 1 步和第 3 步与上述采用自带的 HTTP 服务器类似。区别主要是第 2 步，该步骤不是采用自带的服务，而是放置到 Servlet 容器中。我们采用与 Spring 框架集成的方式来实现。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 6-11 所示。

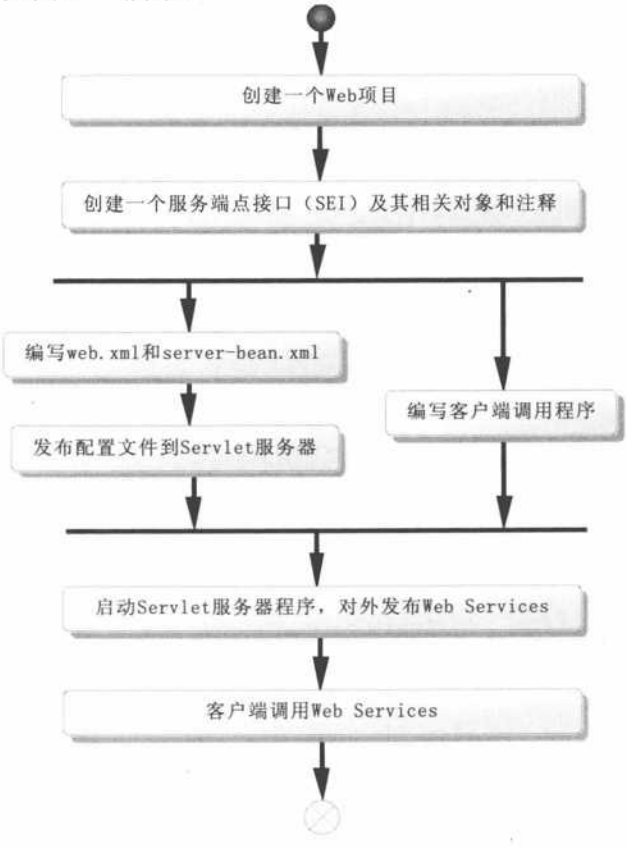


图 6-10　应用程序案例实现步骤流程图

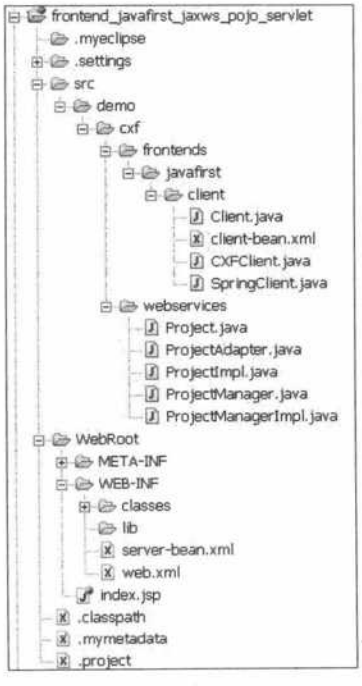


图 6-11　frontend_javafirst_jaxws_pojo_servlet 程序结构

包目录包括三个部分，第一个包目录 WebRoot 目录，是服务器程序目录，包括 web.xml 和 server-bean.xml 文件。第二个包目录 demo.cxf.frontends.javafirst.client，是客户端程序目录。第三

个包目录 demo.cxf.webservices，是 SEI 和数据对象等公用部分，客户端和服务器端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口。由于本程序直接发布到 Servlet 容器，所以没有单独编写服务器发布程序，直接把配置文件和编译好的服务器端点接口（SEI）组件等部署到 Servlet 容器即可。

下面分别介绍两部分：①服务器端配置文件；②客户端调用程序。

（1）服务器端配置文件

服务器端配置文件包括 web.xml 文件和 server-bean.xml 文件。

首先要的定义 web.xml 文件的内容如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
   PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>WEB-INF/server-bean.xml</param-value>
    </context-param>

    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>

    <servlet>
        <servlet-name>CXFServlet</servlet-name>
        <display-name>CXF Servlet</display-name>
        <servlet-class>
            org.apache.cxf.transport.servlet.CXFServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>CXFServlet</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

其中，当 Servlet 启动，将读取 WEB-INF 目录下的 server-bean.xml 文件，server-bean.xml 文件的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="
          http://cxf.apache.org/transports/http/configuration
```

```
        http://cxf.apache.org/schemas/configuration/http-conf.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
          http://cxf.apache.org/jaxws
          http://cxf.apache.org/schemas/jaxws.xsd">

    <import resource="classpath:META-INF/cxf/cxf.xml" />
    <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
    <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />
    <jaxws:endpoint id="ProjectManager"
        implementor="demo.cxf.webservices.ProjectManagerImpl"
        address="/ProjectManager" />
</beans>
```

服务端的代码基本上不做修改，把服务端程序发送到 Tomcat 中。

当 Servlet 容器启动，调用 web.xml 进行初始化，若有 listener 选项，调用其内容，为 org.springframework.web.context.ContextLoaderListener，关于 Spring 的配置在这里就不做详细介绍了。然后调用 server-bean.xml 文件，从 jaxws:endpoint 元素中了解后台的 SEI 是 demo.cxf.webservices.ProjectManagerImpl，公开 Web Services 服务。

（2）客户端应用程序

编写 Web Services 的客户端应用程序。客户端应用程序有三种方式：（1）采用 JAX-WS 规范的前端客户端；（2）Apache CXF 的前端客户端；（3）Spring IoC 注入的前端客户端。下面分别进行说明。

采用 JAX-WS 规范的前端客户端，只需修改后台服务的位置，实现的程序代码如下：

```
……
    private static String endpointAddress =
 "http://localhost:18080/frontend_javafirst_jaxws_pojo_servlet/ProjectManager";
    ……

    public static void main(String[] args) throws Exception {
……
        //用服务端口名称、绑定方式和 Endpoint 地址来创建服务器端口
        service.addPort(PORT_NAME, SOAPBinding.SOAP11HTTP_BINDING,
                endpointAddress);
      ……
```

注意：笔者的 Tomcat 采用的是 18080 端口。

Apache CXF 的前端客户端修改与基于 JAX-WS 规范类似，代码如下：

```
……

    public static void main(String[] args) throws Exception {
……
        //设置 JaxWs 代理工厂的属性，包括服务类和地址
        factory.setServiceClass(ProjectManager.class);
        factory.setAddress("http://localhost:18080/frontend_javafirst_jaxws_pojo_
servlet/ProjectManager");
      ……
```

采用 Spring IoC 注入的客户端的修改只需修改配置文件即可，配置文件如下：

```
……
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                    http://www.springframework.org/schema/beans/spring-beans.xsd
                    http://cxf.apache.org/jaxws
                    http://cxf.apache.org/schemas/jaxws.xsd">

    <jaxws:client id="ProjectManagerClient"
        serviceClass="demo.cxf.webservices.ProjectManager"
        address="http://localhost:18080/frontend_javafirst_jaxws_pojo_servlet/
ProjectManager" />
</beans>
……
```

运行客户端程序，其客户端程序输出结果与“采用 POJO 并基于 JAX-WS 服务的前端实现”的输出是相同。

4. 其他内容

本例代码参见配套资源代码中“第 06 章\frontend_javafirst_jaxws_pojo_servlet”文件夹。

6.2.2.3 采用 Map 并基于 JAX-WS 规范的前端实现

上述例子都是处理 POJO 对象，如果希望用到 Java 的 Map 这种数据类型，那么还要做一些必要的增量修改。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，都是使用 POJO 对象实现客户端和服务器端的数据交换。后台服务端为遵循 JAX-WS 规范的 Endpoint 来实现。前端访问采用 JAX-WS 规范的 Service 方式。实现的结构如图 6-12 所示。

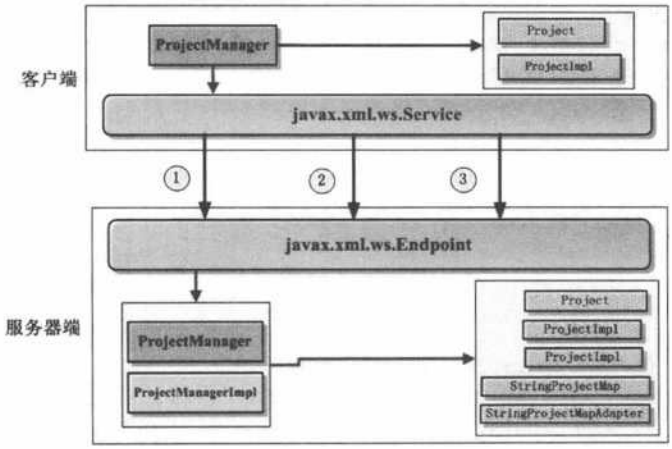


图 6-12　基于 Map 实现 JAX-WS 前端应用模式图

其中：① 表示客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目。

2. 应用程序案例实现步骤

要创建 Map 并基于 JAX-WS 服务的前端实现 Web Services 程序的步骤如图 6-13 所示。

（1）创建一个 Java 项目。

（2）创建一个服务端点接口（SEI）及其相关对象（包括 HashMap）与注释。在这里引入了对 HashMap 对象的支持。

（3）编写发布这个 Java 程序为 Web Services 的服务端程序。

（4）编写 Web Services 的客户端应用程序。

（5）启动服务器程序，对外发布 Web Services。

（6）客户端调用 Web Services。

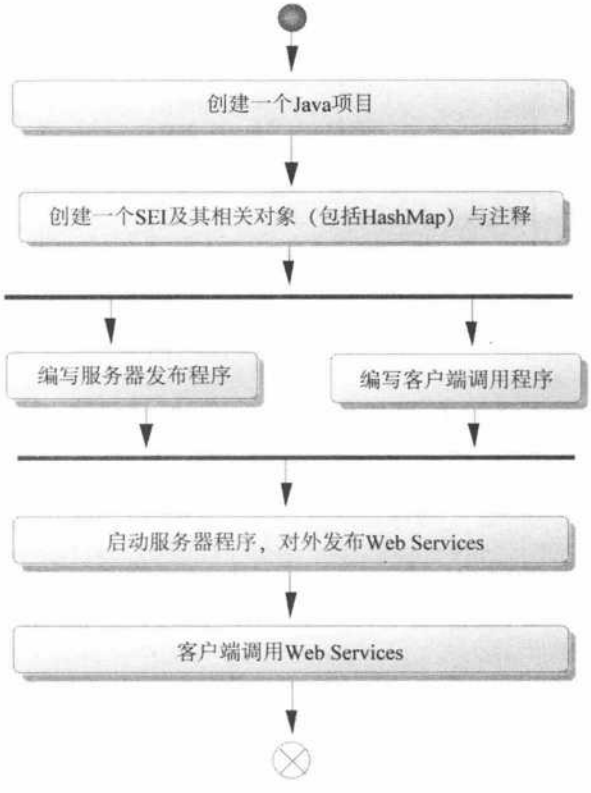


图 6-13　应用程序案例实现流程图

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 6-14 所示。本案例对比采用 POJO 并基于 JAX-WS 服务的前端实现例子，相同部分就不再赘述。

包目录包括三个部分，第一个包目录 demo.cxf.frontends.javafirst.server 是服务器程序目录，第二个包目录 demo.cxf.frontends.javafirst.client 是客户端程序目录。第三个包目录 demo.cxf.webservices 是服务器端点接口 SEI 和相关数据对象等公用部分，客户端和服务器端都有可能引用其内容。

客户端程序一般只引用 ProjectManager 和 Project 接口，而服务器端程序则是接口和实现类都要引用。

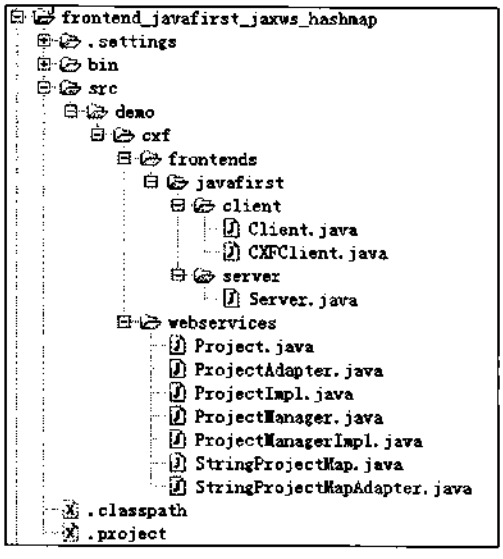


图 6-14　frontend_javafirst_jaxws_hashmap 程序结构

下面分别介绍这三部分：①服务器端点接口 SEI 和相关对象；②服务器端发布程序；③客户端调用程序。

（1）服务器端点接口（SEI）和相关对象

首先要介绍的是服务器端点接口（SEI）。

创建服务器端点接口（SEI），该接口名称为 ProjectManager，其 Java 代码如下：

```
package demo.cxf.webservices;

import javax.jws.WebService;
import javax.jws.WebParam;

@WebService
public interface ProjectManager {
    public String getProjectName();
    …
    @XmlJavaTypeAdapter(StringProjectMapAdapter.class)
    public Map<String,Project> getProjectMap();

}
```

在上述的程序代码中，增加了一个方法 Map<String,Project> getProjectMap，这样可以支持 Map 的数据类型。

下面是这个接口的 Java 实现类 ProjectManagerImpl，代码如下：

```
package demo.cxf.webservices;

import java.util.HashMap;
import java.util.Map;
```

```
import javax.jws.WebService;

@WebService
public class ProjectManagerImpl implements ProjectManager {
    private Project project;
    private Map<String,Project> projectMap;

    public ProjectManagerImpl() {
        project = new ProjectImpl();
        project.setProjectID("Default projectID");
        project.setProjectName("Default projectName");
        project.setProjectDescribe("Default projectDescribe");

        projectMap = new HashMap<String,Project>();
        projectMap.put(project.getProjectID(), project);
    }

    ……

    public Map<String,Project> getProjectMap(){
        System.out.println("getProjectMap() be called");
        return projectMap;
    }

}
```

实现类 ProjectManagerImpl 首先要增加了一个实现方法 Map<String,Project> getProjectMap。另外，还要加入两个类，分别是 StringProjectMap 类和 StringProjectMapAdapter 类。其中，StringProjectMap 类的代码如下：

```
package demo.cxf.webservices;

import java.util.ArrayList;
import java.util.List;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;

@XmlType(name = "StringProjectMap")
@XmlAccessorType(XmlAccessType.FIELD)
public class StringProjectMap {
    @XmlElement(nillable = false, name = "entry")
    List<StringProjectEntry> entries = new ArrayList<StringProjectEntry>();

    public List<StringProjectEntry> getEntries() {return entries; }

    @XmlAccessorType(XmlAccessType.FIELD)
    @XmlType(name = "IdentifiedProject")
    static class StringProjectEntry {
        //Map keys cannot be null
        @XmlElement(required = true, nillable = false)
        String id;

        Project project;
        public void setId(String i) {id = i; }
        public String getId() {return id; }
```

```
        public void setProject(Project p) {project = p;   }
        public Project getProject() {return project; }
    }
}
```

上述代码主要告诉 Apache CXF 的前端框架有一种新的数据类型 StringProjectMap。

StringProjectMapAdapter 类的代码如下：

```
package demo.cxf.webservices;

import java.util.LinkedHashMap;
import java.util.Map;
import javax.xml.bind.annotation.adapters.XmlAdapter;

public   class   StringProjectMapAdapter   extends   XmlAdapter<StringProjectMap,
Map<String, Project>> {
    public StringProjectMap marshal(Map<String, Project> v) throws Exception {
    StringProjectMap map = new StringProjectMap();
        for (Map.Entry<String, Project> e : v.entrySet()) {
        StringProjectMap.StringProjectEntry iue = new StringProjectMap.StringProjectEntry();
            iue.setProject(e.getValue());
            iue.setId(e.getKey());
            map.getEntries().add(iue);
        }
        return map;
    }

    public Map<String, Project> unmarshal(StringProjectMap v) throws Exception {
        Map<String, Project> map = new LinkedHashMap<String, Project>();
        for (StringProjectMap.StringProjectEntry e : v.getEntries()) {
            map.put(e.getId(), e.getProject());
        }
        return map;
    }
}
```

上述代码主要实现新数据类型 StringProjectMap 的适配处理。

（2）服务端发布程序

编写发布这个 Java 程序为 Web Services 的服务端程序。具体内容与前面介绍的 POJO 基本一样。

（3）客户端调用程序

编写 Web Services 的客户端应用程序，代码如下：

```
package demo.cxf.frontends.javafirst.client;

import java.util.Map;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.soap.SOAPBinding;

import demo.cxf.webservices.Project;
import demo.cxf.webservices.ProjectImpl;
import demo.cxf.webservices.ProjectManager;
```

```
public class Client {
    private static final QName SERVICE_NAME = new QName(
            "http://webservices.cxf.demo/", "ProjectManager");
    private static final QName PORT_NAME = new QName(
            "http://webservices.cxf.demo/", "ProjectManagerPort");

    public Client() {   }

    public static void main(String[] args) throws Exception {
        System.out.println("Starting Client");
        Service service = Service.create(SERVICE_NAME);
        String endpointAddress = "http://localhost:9000/ProjectManager";
        service.addPort(PORT_NAME, SOAPBinding.SOAP11HTTP_BINDING, endpointAddress);
        ProjectManager projectManager = service.getPort(ProjectManager.class);
       ……

    }

private static void getProjectMap(ProjectManager projectManager){
        Map<String, Project> projectMap = projectManager.getProjectMap();
        for (Map.Entry<String, Project> currentProject : projectMap.entrySet()) {
            System.out.println("  " + currentProject.getKey() + ": "
                    + currentProject.getValue().getProjectName());
        }
    }
}
```

由于已经有对 Map 数据类型的支持，这样就可以直接进行调用。

运行客户端程序，其输出结果如下：

```
default projectID: default projectName
  projectID2: projectName2
----------setDefaultProject-------
NewProjectID is New default ProjectID . NewProjectName is New default ProjectName
  default projectID: default projectName
  projectID2: projectName2
----------registerProject-------
  default projectID: default projectName
  New ProjectID: New ProjectName
  projectID2: projectName2
----------removeProject-------
  default projectID: default projectName
  projectID2: projectName2
```

本例采用的 HashMap 是<String,POJO>模式，做一些修改后可以支持 HashMap <String,POJO>。当然，读者也可以依此延伸到 List 等多种 Java 数据类型。

4. 其他内容

本例代码参见配套资源代码中“第 06 章\frontend_javafirst_jaxws_hashmap”文件夹。

6.2.2.4　采用 Map 并基于 Servlet 容器的前端实现

上述介绍的采用 Map 的例子主要是基于后台启动的 Web Services 服务，也可以把这个服务委托给 Servlet 容器来进行管理。本例采用 Tomcat 来处理。

要创建 Map 并基于 Servlet 容器的前端实现 Web Services 程序的步骤如图 6-15 所示。

（1）创建一个 Web 服务项目。

（2）创建一个服务端点接口（SEI）及其相关对象与注释，在这里引入了对 HashMap 对象的支持。

（3）编写 web.xml 和 server-bean.xml。

（4）发布配置文件到 Servlet 服务器。

（5）编写 Web Services 的客户端应用程序。

（6）启动 Servlet 服务器程序，对外发布 Web Services。

（7）客户端调用 Web Services。

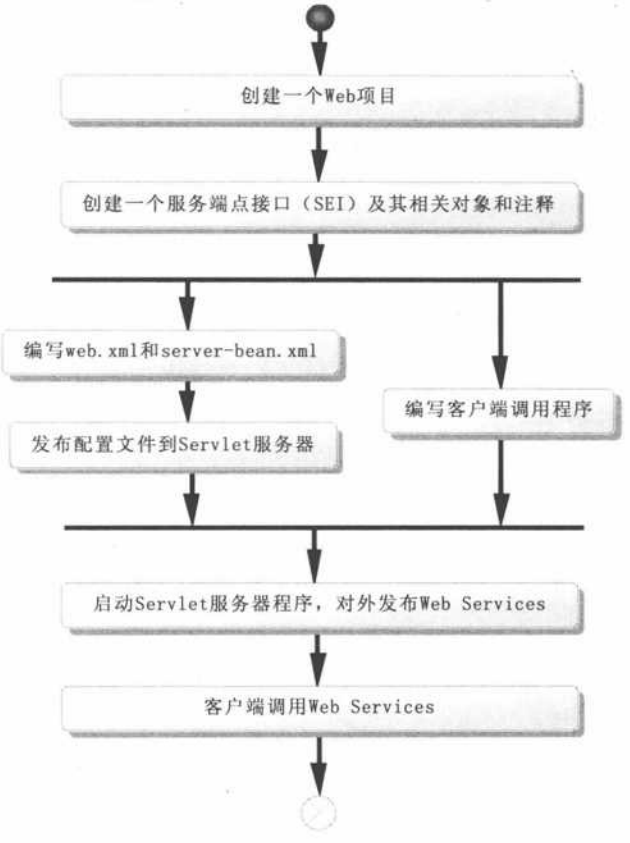


图 6-15　应用程序案例实现流程图

其中第 1 步和第 3 步与采用自带的 HTTP 服务器类似。区别主要是第 2 步，该步不是采用自带的服务，而是放置到 Servlet 容器中。本案例程序采用与 Spring 框架集成的方式来实现。

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 6-16 所示。

包目录包括三个部分，第一个包目录 WebRoot 目录，这是服务器程序目录，包括 web.xml 和 server-bean.xml 文件，第二个包目录 demo.cxf.frontends.javafirst.client，这是客户端程序目录，第三个包目录 demo.cxf.webservices，这是 SEI 和数据对象等公用部分，客户端和服务器端都可能引用其内容。

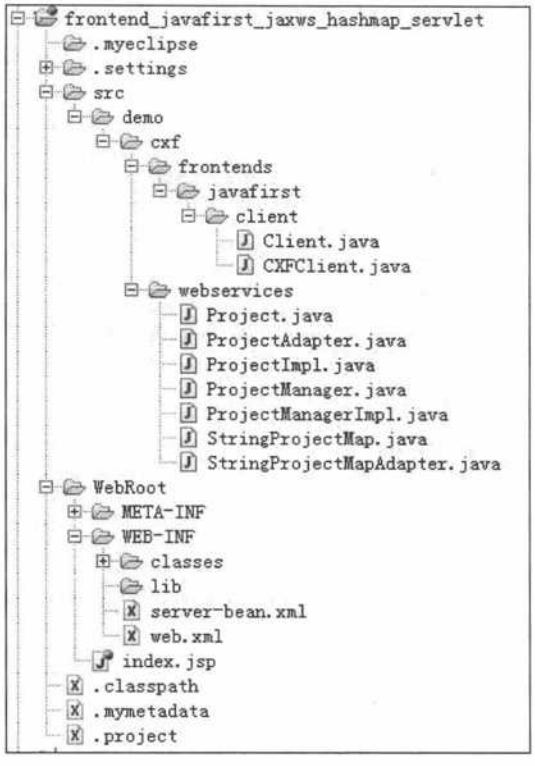


图 6-16　frontend_javafirst_jaxws_hashmap_servlet 程序结构

5. 其他内容

本例代码参见配套资源代码中“第 06 章\frontend_javafirst_jaxws_hashmap_servlet”文件夹。

6.2.2.5　基于定制注释的 JAX-WS 的例子

下面介绍一个例子，在这里例子中，直接实现首先编写 Java 程序代码，然后通过注释，把 Java 应用程序转化为 Web Services 程序。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，都使用 POJO 对象实现客户端和服务器端的数据交换。后台服务端为遵循 JAX-WS 规范的 Endpoint 实现。前端访问采用 JAX-WS 规范的 Service 方式。

采用 JAX-WS 规范的 Service 方式，实现的结构如图 6-17 所示。

2. 应用程序案例实现步骤

要创建 POJO 并基于 JAX-WS 服务的前端实现 Web Services 程序的流程如图 6-18 所示。

流程图说明如下：

（1）创建一个 Java 项目。

（2）创建一个服务端点接口（SEI）及其相关对象和注释。

（3）编写发布这个 Java 程序为 Web Services 的服务端程序。

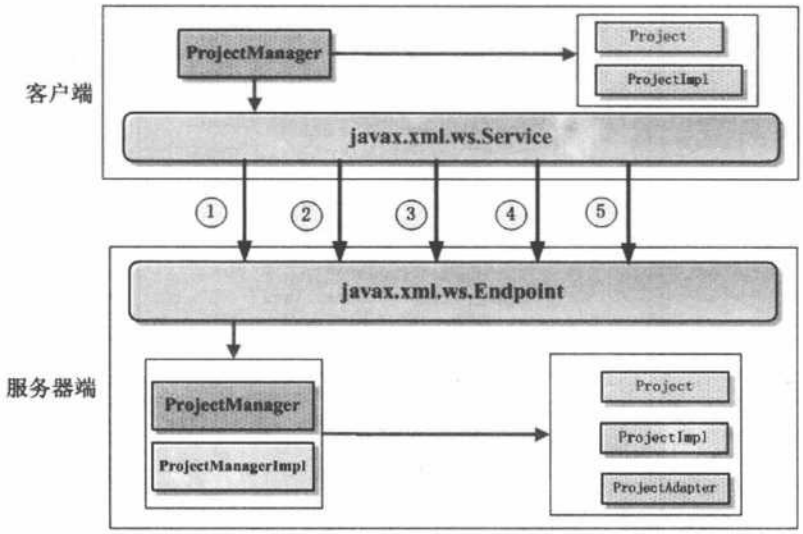


图 6-17　基于 POJO 实现 JAX-WS 前端应用模式图

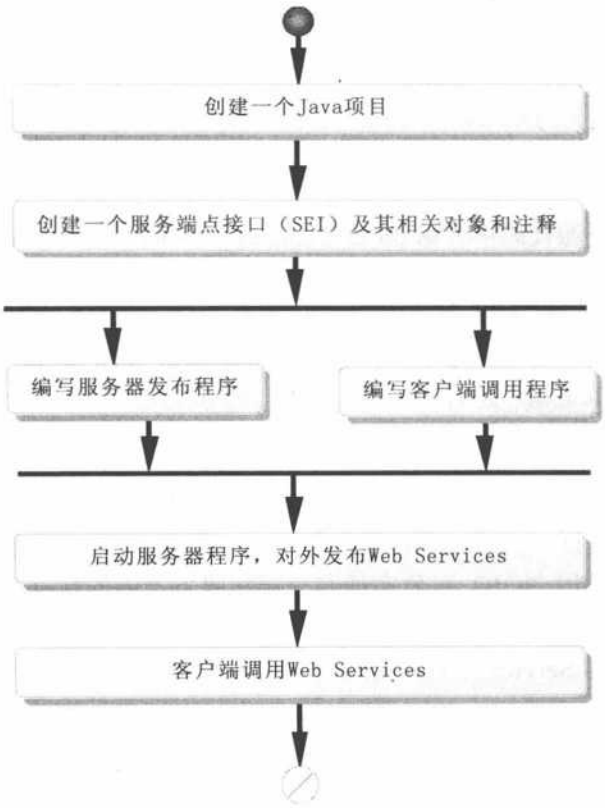


图 6-18　应用程序案例实现流程图

（4）编写 Web Services 的客户端应用程序。

（5）启动服务器程序，对外发布 Web Services。

（6）客户端调用 Web Services。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 6-19 所示。

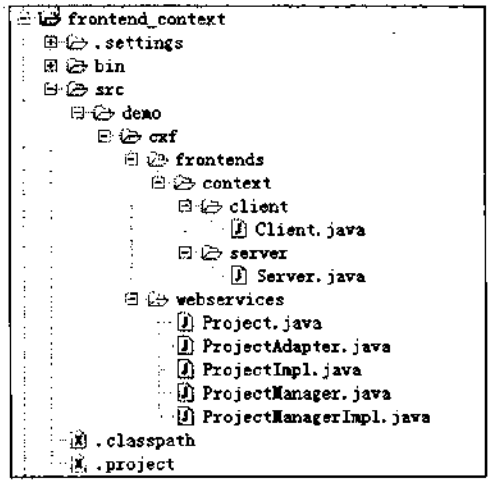


图 6-19　frontend_context 程序结构

包目录包括三个部分，第一个包目录 demo.cxf.frontends.context.server 是服务器程序目录，第二个包目录 demo.cxf.frontends.context.client 是客户端程序目录，第三个包目录 demo.cxf.webservices 是服务器端点接口 SEI 和相关对象等公用部分，服务器端和客户端都可能引用其内容。

服务器端点接口（SEI）是公用部分的接口，服务器端和客户端都必须引用。但服务器端和客户端都对公用部分的引用是不同的。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口，而服务器端程序是接口和实现类都要引用。

下面分别介绍这三部分：①服务器端点接口 SEI 和相关对象；②服务器端发布程序；③客户端调用程序。

（1）服务器端点接口（SEI）和相关对象

创建服务器端点接口（SEI）。该接口名称为 ProjectManager。下面是这个接口的 Java 实现类 ProjectManagerImpl，代码如下：

```
package demo.cxf.webservices;

import java.util.HashMap;
import java.util.Map;

import javax.jws.WebService;

@WebService(
        serviceName = "ProjectManager",
        portName = "ProjectManagerPort",
        targetNamespace = "http://webservices.cxf.demo/",
        endpointInterface = "demo.cxf.webservices.ProjectManager")
public class ProjectManagerImpl implements ProjectManager {
```

```
    private Project project;
    private Map<String,Project> projectMap;

……

}
```

实现类 ProjectManagerImpl 首先要标识这样一个 Web Services 服务类，然后才实现接口的所有方法。这里的主要区别是加入了定制化注释，主要是重命名了服务名称、服务端口等。

（2）服务器端发布程序

创建了 SEI，要将这个 SEI 发布出去，还需要一个部署程序，即服务端的发布程序，本案例采用 JAX-WS 规范来编写服务端程序。创建一个 Server 类，启动这个服务，就可以发布 SEI 服务了。

（3）客户端调用程序

编写 Web Services 的客户端应用程序。有两种实现方式，第一种是基于 JAX-WS 的实现模式。第二种是采用 Apache CXF 模式。

JAX-WS 实现模式的代码如下：

```
package demo.cxf.frontends.context.client;

import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;

import demo.cxf.webservices.ProjectManager;
import demo.cxf.webservices.Project;
import demo.cxf.webservices.ProjectImpl;

public class Client {

    private static final QName SERVICE_NAME =
new QName("http://webservices.cxf.demo/", "ProjectManagerService");
   private static final QName PORT_NAME = new QName("http://webservices.cxf.demo/",
"ProjectManagerPort");

   private static final String WSDL_LOCATION = "http://localhost:9000/ProjectManager?
wsdl";

    public Client() {}

    public static void main(String[] args) throws Exception{
        System.out.println("Starting Client");
        URL wsdlURL = new URL(WSDL_LOCATION);
        Service service = Service.create(wsdlURL,SERVICE_NAME);
        ProjectManager projectManager = service.getPort(PORT_NAME,ProjectManager.class);

       System.out.println(projectManager.getProjectName());

       Project defaultProject = projectManager.getDefaultProject();
        System.out.println("defaultProjectID is " + defaultProject.getProjectID()
                + " . " + "defaultProjectName is " + defaultProject.getProjectName());

       Project project = new ProjectImpl();
       project.setProjectName("new projectName");

        projectManager.setDefaultProject(project);
```

```
        Project newProject = projectManager.getDefaultProject();
        System.out.println("NewProjectID is " + newProject.getProjectID()
                + " . " + "NewProjectName is " + newProject.getProjectName());

        projectManager.registerProject(project);
        System.out.println(projectManager.getProjectNameByID("Default projectID"));
        System.out.println(projectManager.getProjectNameByID("New ProjectID"));

        System.exit(0);
    }
}
```

运行客户端程序，其输出结果与“采用 POJO 并基于 JAX-WS 服务的前端实现”的输出结果相同。

上述的所有调用都是基于 JAX-WS 规范。我们也可以采用 Apache CXF 的前端编程模式来实现调用服务，在这里就不再赘述了。

4. 其他内容

本例代码请参见配套资源代码中“第 06 章\frontend_context”文件夹。

这个程序可以移植到 Servlet 容器上，只是后台服务部署需要做一些修改，详细代码参见配套资源代码中“第 06 章\frontend_context_servlet”文件夹。

6.3　基于 WSDL 优先（WSDL First）的 JAX-WS 前端模式实现

本节介绍基于 WSDL 优先（WSDL First）的 JAX-WS 前端模式实现。

6.3.1　基于 WSDL 优先的 JAX-WS 前端模式实现的步骤

基于 WSDL 优先编程主要是通过已经存在的 WSDL 文档，然后通过 CXF wsdl2java 工具生成相关的客户端和服务器端的程序代码。

要建立一个基于 JAX-WS 协议的 Web 服务，步骤如下：

（1）获得一个 WSDL 文档。

（2）生成的 Web Services 组件。

（3）发布 Web Services。

（4）开发一个 Web Services 客户端。

具体实现流程如图 6-20 所示。

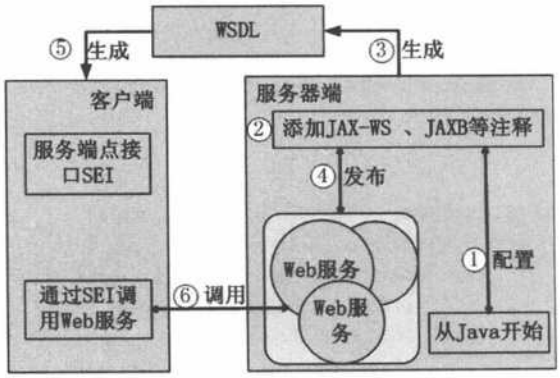


图 6-20　基于 WSDL 优先（WSDL First）开发 Web Services 的步骤图

第 1 步：获得一个 WSDL 文档

首先要获得一个 WSDL 文档，下面是一个 WSDL 文档的例子：

```
<?xml version="1.0" encoding="UTF-8" ?>
-
<wsdl:definitions name="ProjectManager"
    targetNamespace="http://webservices.cxf.demo/"
    xmlns:ns1="http://schemas.xmlsoap.org/soap/http"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://webservices.cxf.demo/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    -
    <wsdl:types>        -
        <xs:schema elementFormDefault="unqualified"
            targetNamespace="http://webservices.cxf.demo/" version="1.0"
            xmlns:tns="http://webservices.cxf.demo/"
            xmlns:xs="http://www.w3.org/2001/XMLSchema">
            <xs:element name="getAllProject" type="tns:getAllProject" />
            <xs:element name="getAllProjectResponse"
                type="tns:getAllProjectResponse" />
            ……
            <xs:element name="updateProjectNameResponse"
                type="tns:updateProjectNameResponse" />

            <xs:complexType name="getDefaultProjectResponse">
                <xs:sequence>
                    <xs:element minOccurs="0" name="return"
                        type="tns:Project" />
                </xs:sequence>
            </xs:complexType>
            ……

        </xs:schema>
    </wsdl:types>
    -
    <wsdl:message name="greeting">
        <wsdl:part element="tns:greeting" name="parameters" />
    </wsdl:message> -
    ……
    -
    <wsdl:portType name="ProjectManager">
        <wsdl:operation name="getDefaultProject">
            <wsdl:input message="tns:getDefaultProject"
                name="getDefaultProject" />
            <wsdl:output message="tns:getDefaultProjectResponse"
                name="getDefaultProjectResponse" />
        </wsdl:operation>
        -
        ……
```

```
</wsdl:portType>
-
<wsdl:binding name="ProjectManagerSoapBinding"
    type="tns:ProjectManager">
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http" />
    -
    <wsdl:operation name="getDefaultProject">
        <soap:operation soapAction="" style="document" />
        -
        <wsdl:input name="getDefaultProject">
            <soap:body use="literal" />
        </wsdl:input>
        -
        <wsdl:output name="getDefaultProjectResponse">
            <soap:body use="literal" />
        </wsdl:output>
    </wsdl:operation>
    -
    ……

</wsdl:binding>
-
<wsdl:service name="ProjectManager">
    -
    <wsdl:port binding="tns:ProjectManagerSoapBinding"
        name="ProjectManagerPort">
        <soap:address
            location="http://localhost:9000/ProjectManager" />
    </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

第 2 步：生成服务器端程序代码

通过 WSDL 文档创建一个 Web Services。其中，WSDL 文档的元素对应到 Java 组件的映射如表 6-9 所示。

表 6-9　WSDL文档元素对应到Java组件的映射表

WSDL文档元素	Java组件
<wsdl:definitions>	元素的targetNamespace属性表示Java 的包（package）
<wsdl:portType>	Java 服务器端点接口（SEI）
<wsdl:portType>	元素的<wsdl:operation>子元素表示Java方法
<wsdl:service>	Web服务类
<wsdl:message>	Web服务操作参数

Apache CXF 提供了多种工具可以自动生成 Web Services 的 Java 组件代码。使用 WSDL2Java

工具通过 WSDL 文件来创建一个服务的程序代码。可以用下面的命令实现这种模式：

```
wsdl2java -ant -impl -server -d <outputdir> mywsdl.wsdl
```

表 6-10 是 WSDL2Java 命令工具的选项说明。

表 6-10 WSDL2Java命令工具参数选项和说明

工具选项	说明
- ant:	此选项将生成Ant的构建文件build.xml，可用于生成代码
- impl:	此选项将产生一个服务实现类
- server:	此选项将生成一个服务器组件，可以用于发布并启动服务
-d	此选项可以用来直接生成代码到一个特定的文件夹

使用现有的 OrderProcess.wsdl 并将其转换成一个 JAX-WS 的服务类。

```
wsdl2java.bat -d src -ant -impl -server OrderProcess.wsdl
```

此命令将生成五种类型的文件：

- JAXB 的输入和输出信息类。这个工具生成各自的 Java 输入和输出信息组件。输入和输出消息的架构遵循 WSDL 中定义的格式。对于 OrderProcess.wsdl，工具会根据 XML 元素中映射 processOrder 和 processOrderResponse 元素产生 ProcessOrder 类来作为输入级和输出级 ProcessOrderResponse 类。工具还根据 WSDL 文件中<xs:complexType name="order">定义来生成 Order 类。
- Web Services 接口。这个 Web Services 接口包含 Web Services 方法。该服务接口的 WSDL 生成订单处理 OrderProcess.java。
- Web Services 实现类，这个类提供了一个示例实现 Web Services 接口。如果要添加实现代码，可修改这个类。该服务实现了订单流程 WSDL 生成 OrderProcessImpl.java。
- 独立服务器类，这个类提供了一个独立的实用程序来发布和测试基于 JAX-WS 规范的 Web Services 使用。OrderProcess.wsdl 文件内部嵌入式字符内容 server.The 可生成一个 OrderProcess_OrderProcessPort_Server.java。
- Build 文件。生成文件还可以使用独立服务器类来发布 Web 服务。

所有这些组件都在 demo.order 包。WSDL2Java 工具把<wsdl:definitions>元素的 targetNamespace 属性映射成为应用程序的包名称。所以上述的<wsdl:definitions>元素的 targetNamespace 属性为???，形成的程序代码中的包名称也是???。

下面的代码显示了生成的 ProcessOrder 类：

```
package demo.order;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
```

```
@XmlType(name = "processOrder", propOrder = { "arg0" })
public class ProcessOrder {
    protected Order arg0;


    public Order getArg0() {
return arg0;
}

    public void setArg0(Order value) {
        this.arg0 = value;
    }
}
```

a. 服务接口

OrderProcess 接口是一个服务器端点接口，其定义了一个接口方法，下面的代码显示了生成的 OrderProcess 接口：

```
@WebService(targetNamespace = "http://order.demo/",
 name = "OrderProcess")
@XmlSeeAlso( { ObjectFactory.class })
public interface OrderProcess {
    @ResponseWrapper(localName = "processOrderResponse",
targetNamespace = "http://order.demo/",
className = "demo.order.ProcessOrderResponse")
    @RequestWrapper(localName = "processOrder",
targetNamespace = "http://order.demo/",
className = "demo.order.ProcessOrder")
    @WebResult(name = "return", targetNamespace = "")
    @WebMethod
    public java.lang.String processOrder(
            @WebParam(name = "arg0", targetNamespace = "")
            demo.order.Order arg0);
}
```

b. 服务实现类

OrderProcessImpl 类实现 OrderProcess 接口，其形成的代码如下：

```
@javax.jws.WebService(serviceName = "OrderProcessService",
portName = "OrderProcessPort",
targetNamespace = "http://order.demo/",
wsdlLocation = "file:OrderProcess.wsdl",
endpointInterface = "demo.order.OrderProcess")

public class OrderProcessImpl implements OrderProcess{
    private static final Logger LOG = Logger.getLogger(OrderProcessImpl.class
            .getName());

    public java.lang.String processOrder(demo.order.Order arg0) {
        LOG.info("Executing operation processOrder");
```

```
        System.out.println(arg0);
        try {
            java.lang.String _return = "ORD1234";
            return _return;
        } catch (Exception ex) {
            ex.printStackTrace();
            throw new RuntimeException(ex);
        }
    }
}
```

这个类生成的@WebService 注释了所有相关的属性。它实现了生成 OrderProcess 接口功能。SEI 定义一个空的 processOrder 服务方法来接受执行。开发人员需要编写这个空方法的程序代码来完成方法的执行任务。

c. 独立服务器类

生成的 OrderProcess_OrderProcessPort_Server 类表示一个独立的服务器。其来源于<wsdl:portType>元素和<wsdl:port>元素的名称，用来发布在指定端点地址的服务并可以启动服务器，程序代码如下：

```
public class OrderProcess_OrderProcessPort_Server {
    protected OrderProcess_OrderProcessPort_Server() throws Exception {
        System.out.println("Starting Server");
        Object implementor = new OrderProcessImpl();
        String address = "http://localhost:8080/OrderProcess";
        Endpoint.publish(address, implementor);
    }

    public static void main(String args[]) throws Exception {
        new OrderProcess_OrderProcessPort_Server();
        System.out.println("Server ready...");
    }
}
```

6.3.2 基于 WSDL 优先的 JAX-WS 前端模式实现的简单例子

本程序演示 Apache CXF 遵循 JAX-WS 规范的客户端和服务器端调用，客户端采用 Apache CXF 的 WSDL 优先（WSDL First）前端应用模式。采用 POJO 对象作为中间传输对象。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，使用 POJO 对象实现客户端和服务器端的数据交换。后台服务端为遵循 JAX-WS 规范的 Endpoint 来实现。通过 WSDL 文档来生成程序的基本模板，前端访问采用两种方式，其中一种是 JAX-WS 规范的 Service 方式，实现的结构如图 6-21 所示。

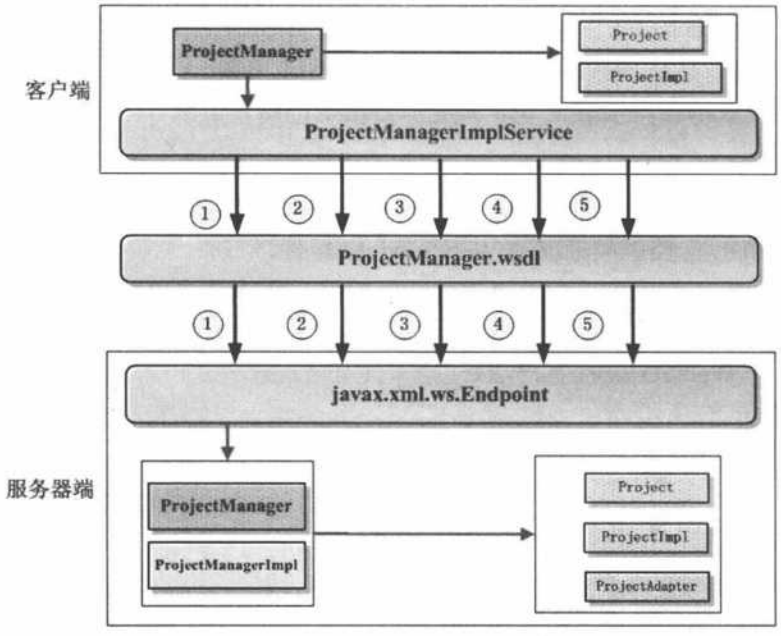


图 6-21　文档优先（WSDL First）基于 JAX-WS 规范的 Web Services 开发

其中：① 表示客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示 JAX-WS 规范客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。客户端和服务端的 SEI 都是 ProjectManager 接口。

另一种方式是采用 Apache CXF 本身的客户端来实现，实现结构如图 6-22 所示。

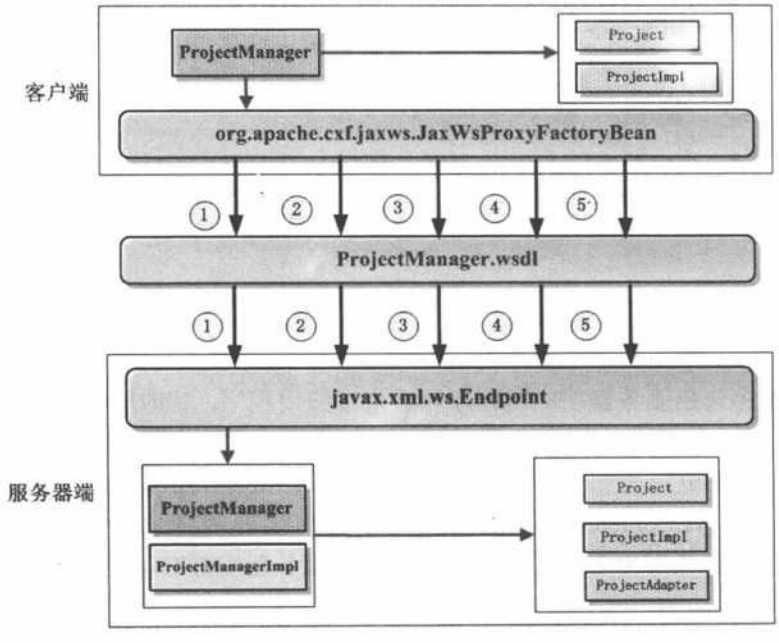


图 6-22　文档优先（WSDL First）基于 Apache CXF 规范的 Web Services 开发

其中：① 表示 Apache CXF 客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。客户端和服务端的 SEI 都是 ProjectManager 接口。

2. 应用程序案例实现步骤

（1）通过 WSDL 文档生成标准的 Web Services 组件。

（2）根据应用情况修改标准的 Web Services 组件，形成具体应用的 Web Services 组件。

（3）发布 Web Services。

（4）开发一个 Web Services 客户端。

开发过程如图 6-23 所示。

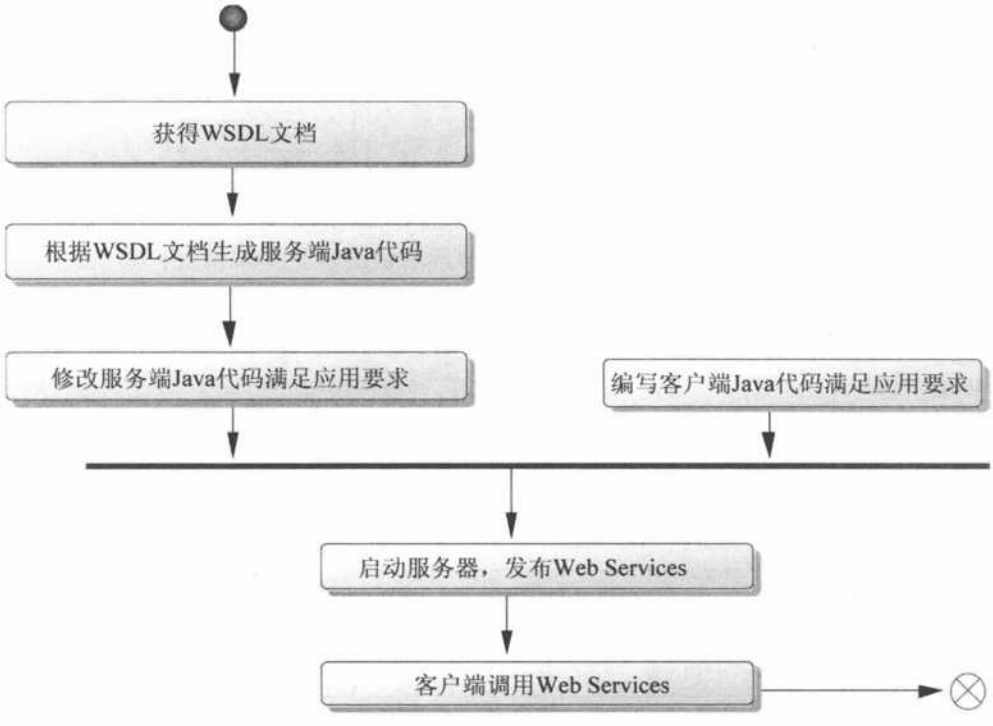


图 6-23　案例开发流程图

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 6-24 所示。

包目录包括三个部分，第一个包目录 demo.cxf.frontends.javafirst.server 是服务器程序目录，第二个包目录 demo.cxf.frontends.javafirst.client 是客户端程序目录，第三个包目录 demo.cxf.webservices 是 SEI 和相关对象等公用部分，客户端和服务器端都有可能引用其内容。

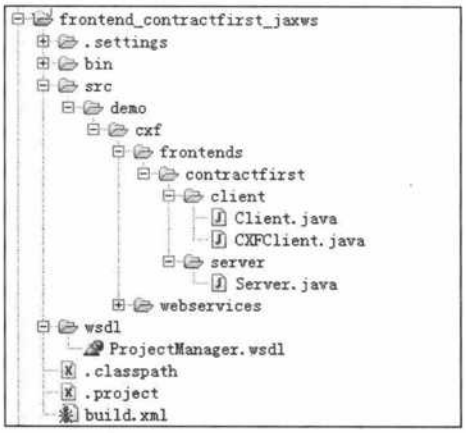


图 6-24　frontend_contractfirst_jaxws 程序结构

用下列命令生成一些 Web Services 组件，生成服务器端程序代码，通过 WSDL 文档创建一个 Web Services。

```
wsdl2java -ant  -impl -server -d outputDir wsdl\ProjectManager.wsdl
```

其中 WSDL 文档元素生成到对应到 Java 组件的类列表如图 6-25 所示。

图 6-25　WSDL 文档生成的类列表

生成规则是一个对象生成一个类、一个方法生成一个类。

其中较重要的有：Project.java、ProjectManager.java、ProjectManagerImpl.java 和 ProjectManagerImplService.java。在这里，Project 已经是一个类而不是一个接口。ProjectManager

接口是 SEI，而 ProjectManagerImpl 是实现 SEI 的实现类。ProjectManagerImplService 则是调用服务，继承 javax.xml.ws.Service 类。

生成了服务器发布程序 ProjectManager_ProjectManagerImplPort_Server.java。这个程序可以直接使用，不用修改，代码如下：

```
package demo.cxf.webservices;

import javax.xml.ws.Endpoint;

public class ProjectManager_ProjectManagerImplPort_Server{
   protected ProjectManager_ProjectManagerImplPort_Server() throws Exception {
      System.out.println("Starting Server");
      Object implementor = new ProjectManagerImpl();
      String address = "http://localhost:9000/ProjectManager";
      Endpoint.publish(address, implementor);
   }

   public static void main(String args[]) throws Exception {
      new ProjectManager_ProjectManagerImplPort_Server();
      System.out.println("Server ready...");
      Thread.sleep(5 * 60 * 1000);
      System.out.println("Server exiting");
      System.exit(0);
   }
}
```

生成的 SEI 是 ProjectManager.java 文件，由于这个是接口，也可以直接应用，不用修改，代码如下：

```
package demo.cxf.webservices;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.bind.annotation.XmlSeeAlso;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.ResponseWrapper;

@WebService(targetNamespace = "http://webservices.cxf.demo/", name = "ProjectManager")
@XmlSeeAlso({ObjectFactory.class})
public interface ProjectManager {
@ResponseWrapper(localName = "getDefaultProjectResponse",
targetNamespace = "http://webservices.cxf.demo/",
className = "demo.cxf.webservices.GetDefaultProjectResponse")
@RequestWrapper(localName = "getDefaultProject",
targetNamespace = "http://webservices.cxf.demo/",
className = "demo.cxf.webservices.GetDefaultProject")
   @WebResult(name = "return", targetNamespace = "")
   @WebMethod
   public demo.cxf.webservices.Project getDefaultProject();

@ResponseWrapper(localName = "removeProjectResponse",
targetNamespace = "http://webservices.cxf.demo/",
className = "demo.cxf.webservices.RemoveProjectResponse")
@RequestWrapper(localName = "removeProject",
```

```
targetNamespace = "http://webservices.cxf.demo/",
className = "demo.cxf.webservices.RemoveProject")
    @WebMethod
    public void removeProject(
        @WebParam(name = "id", targetNamespace = "")
        java.lang.String id
    );

@ResponseWrapper(localName = "greetingResponse",
targetNamespace = "http://webservices.cxf.demo/",
className = "demo.cxf.webservices.GreetingResponse")
@RequestWrapper(localName = "greeting",
targetNamespace = "http://webservices.cxf.demo/",
className = "demo.cxf.webservices.Greeting")
    @WebResult(name = "return", targetNamespace = "")
    @WebMethod
    public java.lang.String greeting();

@ResponseWrapper(localName = "updateProjectNameResponse",
targetNamespace = "http://webservices.cxf.demo/",
className = "demo.cxf.webservices.UpdateProjectNameResponse")
@RequestWrapper(localName = "updateProjectName",
targetNamespace = "http://webservices.cxf.demo/",
className = "demo.cxf.webservices.UpdateProjectName")
    @WebMethod
    public void updateProjectName(
        @WebParam(name = "projectName", targetNamespace = "")
        java.lang.String projectName
    );

@ResponseWrapper(localName = "getProjectNameByIDResponse",
targetNamespace = "http://webservices.cxf.demo/",
className = "demo.cxf.webservices.GetProjectNameByIDResponse")
@RequestWrapper(localName = "getProjectNameByID",
targetNamespace = "http://webservices.cxf.demo/",
className = "demo.cxf.webservices.GetProjectNameByID")
    @WebResult(name = "return", targetNamespace = "")
    @WebMethod
    public java.lang.String getProjectNameByID(
        @WebParam(name = "arg0", targetNamespace = "")
        java.lang.String arg0
    );

@ResponseWrapper(localName = "getProjectNameResponse",
targetNamespace = "http://webservices.cxf.demo/",
className = "demo.cxf.webservices.GetProjectNameResponse")
@RequestWrapper(localName = "getProjectName",
targetNamespace = "http://webservices.cxf.demo/",
className = "demo.cxf.webservices.GetProjectName")
    @WebResult(name = "return", targetNamespace = "")
    @WebMethod
    public java.lang.String getProjectName();

@ResponseWrapper(localName = "registerProjectResponse",
targetNamespace = "http://webservices.cxf.demo/",
className = "demo.cxf.webservices.RegisterProjectResponse")
@RequestWrapper(localName = "registerProject",
targetNamespace = "http://webservices.cxf.demo/",
className = "demo.cxf.webservices.RegisterProject")
```

```
    @WebMethod
    public void registerProject(
        @WebParam(name = "arg0", targetNamespace = "")
        demo.cxf.webservices.Project arg0
    );

@ResponseWrapper(localName = "setDefaultProjectResponse",
targetNamespace = "http://webservices.cxf.demo/",
className = "demo.cxf.webservices.SetDefaultProjectResponse")
@RequestWrapper(localName = "setDefaultProject",
targetNamespace = "http://webservices.cxf.demo/",
className = "demo.cxf.webservices.SetDefaultProject")
    @WebMethod
    public void setDefaultProject(
        @WebParam(name = "arg0", targetNamespace = "")
        demo.cxf.webservices.Project arg0
    );

@ResponseWrapper(localName = "getAllProjectResponse",
targetNamespace = "http://webservices.cxf.demo/",
className = "demo.cxf.webservices.GetAllProjectResponse")
@RequestWrapper(localName = "getAllProject",
targetNamespace = "http://webservices.cxf.demo/",
className = "demo.cxf.webservices.GetAllProject")
    @WebMethod
    public void getAllProject();

@ResponseWrapper(localName = "greetingWithParameterResponse",
targetNamespace = "http://webservices.cxf.demo/",
className = "demo.cxf.webservices.GreetingWithParameterResponse")
@RequestWrapper(localName = "greetingWithParameter",
targetNamespace = "http://webservices.cxf.demo/",
className = "demo.cxf.webservices.GreetingWithParameter")
    @WebResult(name = "return", targetNamespace = "")
    @WebMethod
    public java.lang.String greetingWithParameter(
        @WebParam(name = "greet", targetNamespace = "")
        java.lang.String greet
    );
}
```

ProjectManagerImplService.java 是生成的 Service 组件，代码如下：

```
package demo.cxf.webservices;

import java.net.MalformedURLException;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.WebEndpoint;
import javax.xml.ws.WebServiceClient;
import javax.xml.ws.WebServiceFeature;
import javax.xml.ws.Service;

@WebServiceClient(name = "ProjectManagerImplService",
                  //wsdlLocation = "file:wsdl/ProjectManager.wsdl",
                  targetNamespace = "http://webservices.cxf.demo/")
public class ProjectManagerImplService extends Service {

    public final static URL WSDL_LOCATION;
```

```
public final static QName SERVICE =
new QName("http://webservices.cxf.demo/", "ProjectManagerImplService");
public final static QName ProjectManagerImplPort =
new QName("http://webservices.cxf.demo/", "ProjectManagerImplPort");
    static {
        URL url = null;
        try {
            url = new URL("file:wsdl/ProjectManager.wsdl");
        } catch (MalformedURLException e) {
            System.err.println("Can not initialize the default wsdl from file:wsdl/
ProjectManager.wsdl");
            e.printStackTrace();
        }
        WSDL_LOCATION = url;
    }

    public ProjectManagerImplService(URL wsdlLocation) {
        super(wsdlLocation, SERVICE);
    }

    public ProjectManagerImplService(URL wsdlLocation, QName serviceName) {
        super(wsdlLocation, serviceName);
    }

    public ProjectManagerImplService() {
        super(WSDL_LOCATION, SERVICE);
    }

    @WebEndpoint(name = "ProjectManagerImplPort")
    public ProjectManager getProjectManagerImplPort() {
        return super.getPort(ProjectManagerImplPort, ProjectManager.class);
    }

    @WebEndpoint(name = "ProjectManagerImplPort")
    public ProjectManager getProjectManagerImplPort(WebServiceFeature... features) {
        return super.getPort(ProjectManagerImplPort, ProjectManager.class, features);
    }

}
```

唯一要进行修改的 ProjectManagerImpl.java 文件。这是 SEI 的实现类，过程中只生成了标准程序的模板，所以还需要按照具体应用进行修改。修改后的程序代码如下：

```
/**
 * Please modify this class to meet your needs
 * This class is not complete
 */

package demo.cxf.webservices;

import java.util.HashMap;
import java.util.Map;
import java.util.logging.Logger;

@javax.jws.WebService(  serviceName = "ProjectManagerImplService",
                        portName = "ProjectManagerImplPort",
                        targetNamespace = "http://webservices.cxf.demo/",
                        wsdlLocation = "file:wsdl/ProjectManager.wsdl",
```

```
                    endpointInterface = "demo.cxf.webservices.ProjectManager")

public class ProjectManagerImpl implements ProjectManager {
    private static final Logger LOG = Logger.getLogger(ProjectManagerImpl.class.getName());
    private Project project;
    private Map<String, Project> projectMap;

    public ProjectManagerImpl() {
        System.out.println("----ProjectManagerImpl Construct----");
        initProject();
    }

    public demo.cxf.webservices.Project getDefaultProject() {
        LOG.info("Executing operation getDefaultProject");
        try {
         demo.cxf.webservices.Project _return = project;
            return _return;
        } catch (Exception ex) {
            ex.printStackTrace();
            throw new RuntimeException(ex);
        }
    }

    public void removeProject(java.lang.String id) {
        LOG.info("Executing operation removeProject");
        System.out.println(id);
        try {
             if (!projectMap.containsKey(id))  return;
         projectMap.remove(id);
        } catch (Exception ex) {
            ex.printStackTrace();
            throw new RuntimeException(ex);
        }
    }

    public java.lang.String greeting() {
        LOG.info("Executing operation greeting");
        try {
         //修改默认代码
            java.lang.String _return = "this is greeting.";
            return _return;
        } catch (Exception ex) {
            ex.printStackTrace();
            throw new RuntimeException(ex);
        }
    }

    public void updateProjectName(java.lang.String projectName) {
        LOG.info("Executing operation updateProjectName");
        System.out.println(projectName);
        try {
         //修改默认代码
         project.setProjectName(projectName);
        } catch (Exception ex) {
            ex.printStackTrace();
            throw new RuntimeException(ex);
        }
    }
```

```
    public java.lang.String getProjectNameByID(java.lang.String arg0) {
        LOG.info("Executing operation getProjectNameByID");
        System.out.println(arg0);
        try {
         //修改默认代码
         if (!projectMap.containsKey(arg0))
             return null;
         java.lang.String _return = projectMap.get(arg0).getProjectName();
            return _return;
        } catch (Exception ex) {
            ex.printStackTrace();
            throw new RuntimeException(ex);
        }
}

    public java.lang.String getProjectName() {
        LOG.info("Executing operation getProjectName");
        try {
         //修改默认代码
            java.lang.String _return = project.getProjectName();
            return _return;
        } catch (Exception ex) {
            ex.printStackTrace();
            throw new RuntimeException(ex);
        }
    }

    public void registerProject(demo.cxf.webservices.Project arg0) {
        LOG.info("Executing operation registerProject");
        System.out.println(arg0);
        try {
         //修改默认代码
         if (projectMap.containsKey(arg0.getProjectID()))   return;
         projectMap.put(arg0.getProjectID(), arg0);
        } catch (Exception ex) {
            ex.printStackTrace();
            throw new RuntimeException(ex);
        }
    }

    public void setDefaultProject(demo.cxf.webservices.Project arg0) {
        LOG.info("Executing operation setDefaultProject");
        System.out.println(arg0);
        try {
         //修改默认代码
         if (arg0 == null) return;
         this.project = arg0;
        } catch (Exception ex) {
            ex.printStackTrace();
            throw new RuntimeException(ex);
        }
    }

    public void getAllProject() {
        LOG.info("Executing operation getAllProject");
        try {
         //修改默认代码
         for (Map.Entry<String, Project> currentProject : projectMap.entrySet()) {
```

```
            System.out.println("  " + currentProject.getKey() + ": "
                    + currentProject.getValue().getProjectName());
        }
      } catch (Exception ex) {
         ex.printStackTrace();
         throw new RuntimeException(ex);
      }
   }

   public java.lang.String greetingWithParameter(java.lang.String greet) {
      LOG.info("Executing operation greetingWithParameter");
      System.out.println(greet);
      try {
       java.lang.String _return = "Hello, " + greet + " This is ProjectManager.";;
         return _return;
      } catch (Exception ex) {
         ex.printStackTrace();
         throw new RuntimeException(ex);
      }
   }

   private void initProject() {
       project = new Project();
       project.setProjectID("default projectID");
       project.setProjectName("default projectName");
       project.setProjectDescribe("default projectDescribe");

       projectMap = new HashMap<String, Project>();

       Project project2 = new Project();
       project2.setProjectID("projectID2");
       project2.setProjectName("projectName2");
       project2.setProjectDescribe("projectDescribe2");

       projectMap.put(project.getProjectID(), project);
       projectMap.put(project2.getProjectID(), project2);
   }
}
```

粗体字表示的是要修改的部分。

4. 开发一个 Web Service 客户端

开发一个 JAX-WS 的客户端程序，代码如下：

```
package demo.cxf.frontends.contractfirst.client;

import java.io.File;
import java.net.URL;

import demo.cxf.webservices.Project;
import demo.cxf.webservices.ProjectManager;
import demo.cxf.webservices.ProjectManagerImplService;

public class Client {

    public Client() {}

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

```
        File wsdlFile = new File("wsdl/ProjectManager.wsdl");
        URL wsdlURL;
        if (wsdlFile.exists()) {
            wsdlURL = wsdlFile.toURL();
        } else {
            wsdlURL = new URL(args[0]);
        }

        // 用服务名称创建一个服务
        ProjectManagerImplService service = new ProjectManagerImplService(wsdlURL);

        // 获得 SEI 接口
        ProjectManager projectManager = service.getProjectManagerImplPort();

        //项目的基本情况
        getGreet(projectManager);
        greetingWithParameter(projectManager);
        getProjectName(projectManager);
        getDefaultProject(projectManager);
        getAllProject(projectManager);

        //修改默认项目，查看项目的基本情况
        setDefaultProject(projectManager);

        //增加一个项目，查看项目的基本情况
        registerProject(projectManager);

        //删除一个项目，查看项目的基本情况
        removeProject(projectManager);

        System.exit(0);
    }

    //调用 SEI 的 getGreet 方法
    private static void getGreet(ProjectManager projectManager)throws Exception {
        System.out.println(projectManager.greeting());
    }

    //调用 SEI 的 greetingWithParameter 方法
    private static void greetingWithParameter(ProjectManager projectManager)throws
Exception {
        System.out.println(projectManager.greetingWithParameter("greet"));
    }

    //调用 SEI 的 getProjectName 方法
    private static void getProjectName(ProjectManager projectManager)throws Exception {
        System.out.println(projectManager.getProjectName());
    }

    //调用 SEI 的 getDefaultProject 方法
    private  static  void  getDefaultProject(ProjectManager  projectManager)throws
Exception {
        Project defaultProject = projectManager.getDefaultProject();
        System.out.println("defaultProjectID is " + defaultProject.getProjectID()
                + " . " + "defaultProjectName is " + defaultProject.getProjectName());

    }
```

```
    //调用 SEI 的 setDefaultProject 方法
    private static void setDefaultProject(ProjectManager projectManager)throws
Exception {
        Project project = new Project();
        project.setProjectID("New Default ProjectID");
        project.setProjectName("New Default ProjectName");

        projectManager.setDefaultProject(project);
        Project newProject = projectManager.getDefaultProject();
        System.out.println("DefaultProjectID is " + newProject.getProjectID()
                + " . " + "DefaultProjectName is " + newProject.getProjectName());
        projectManager.getAllProject();
    }

    //调用 SEI 的 registerProject 方法
    private static void registerProject(ProjectManager projectManager)throws
Exception {
        Project project = new Project();
        project.setProjectID("New ProjectID");
        project.setProjectName("New ProjectName");

        projectManager.registerProject(project);
        System.out.println(projectManager.getProjectNameByID("New ProjectID"));
        System.out.println(projectManager.getProjectNameByID("New ProjectID"));
        projectManager.getAllProject();
    }

    //调用 SEI 的 removeProject 方法
    private static void removeProject(ProjectManager projectManager)throws Exception {
        projectManager.removeProject("New ProjectID");
        projectManager.getAllProject();
    }

    //调用 SEI 的 getAllProject 方法
    private static void getAllProject(ProjectManager projectManager) throws Exception{
        projectManager.getAllProject();
    }
}
```

开发一个 Apache CXF 的客户端程序，代码如下：

```
package demo.cxf.frontends.contractfirst.client;

import org.apache.cxf.jaxws.JaxWsProxyFactoryBean;
import demo.cxf.webservices.Project;
import demo.cxf.webservices.ProjectManager;

public class CXFClient {

    public CXFClient() {}

    public static void main(String[] args) throws Exception {
        System.out.println("Starting Client");

        //定义 JaxWs 的代理工厂
        JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();

        //设置 JaxWs 代理工厂的属性，包括服务类和地址
```

```
        factory.setServiceClass(ProjectManager.class);
        factory.setAddress("http://localhost:9000/ProjectManager");

        //获取服务类的对象并处理
        ProjectManager projectManager = (ProjectManager) factory.create();

        //项目的基本情况
        getGreet(projectManager);
        greetingWithParameter(projectManager);
        getProjectName(projectManager);
        getDefaultProject(projectManager);
        getAllProject(projectManager);

        //修改默认项目，查看项目的基本情况
        setDefaultProject(projectManager);

        //增加一个项目，查看项目的基本情况
        registerProject(projectManager);

        //删除一个项目，查看项目的基本情况
        removeProject(projectManager);
        System.exit(0);
    }

    //调用 SEI 的 getGreet 方法
    private static void getGreet(ProjectManager projectManager)throws Exception {
        System.out.println(projectManager.greeting());
    }

    //调用 SEI 的 greetingWithParameter 方法
    private static void greetingWithParameter(ProjectManager projectManager)throws
Exception {
        System.out.println(projectManager.greetingWithParameter("greet"));
    }

    //调用 SEI 的 getProjectName 方法
    private static void getProjectName(ProjectManager projectManager)throws
Exception {
        System.out.println(projectManager.getProjectName());
    }

    //调用 SEI 的 getDefaultProject 方法
    private static void getDefaultProject(ProjectManager projectManager)throws Exception {
        Project defaultProject = projectManager.getDefaultProject();
        System.out.println("defaultProjectID is " + defaultProject.getProjectID()
                + " . " + "defaultProjectName is " + defaultProject.getProjectName());
    }

    //调用 SEI 的 setDefaultProject 方法
    private static void setDefaultProject(ProjectManager projectManager)throws Exception {
        Project project = new Project();
        project.setProjectID("New Default ProjectID");
        project.setProjectName("New Default ProjectName");

        projectManager.setDefaultProject(project);
        Project newProject = projectManager.getDefaultProject();
        System.out.println("DefaultProjectID is " + newProject.getProjectID()
                + " . " + "DefaultProjectName is " + newProject.getProjectName());
```

```
        projectManager.getAllProject();
    }

    //调用 SEI 的 registerProject 方法
    private static void registerProject(ProjectManager projectManager)throws Exception {
        Project project = new Project();
        project.setProjectID("New ProjectID");
        project.setProjectName("New ProjectName");

        projectManager.registerProject(project);
        System.out.println(projectManager.getProjectNameByID("New ProjectID"));
        System.out.println(projectManager.getProjectNameByID("New    ProjectID"));

        projectManager.getAllProject();
    }

    //调用 SEI 的 removeProject 方法
    private static void removeProject(ProjectManager projectManager)throws Exception {
        projectManager.removeProject("New ProjectID");
        projectManager.getAllProject();
    }

    //调用 SEI 的 getAllProject 方法
    private static void getAllProject(ProjectManager projectManager) throws Exception{
        projectManager.getAllProject();
    }
}
```

运行客户端程序，其输出结果如下：

```
Starting Server
----ProjectManagerImpl Construct----

Server ready...
greet

  default projectID: default projectName
  projectID2: projectName2

demo.cxf.webservices.Project@1e3cc77
  default projectID: default projectName
  projectID2: projectName2

demo.cxf.webservices.Project@1302b69

New ProjectID

New ProjectID

  default projectID: default projectName
  New ProjectID: New ProjectName
  projectID2: projectName2

New ProjectID

  projectID2: projectName2
```

5. 其他内容

本例代码请参见配套资源代码中“第 06 章\frontend_contractfirst_jaxws”文件夹。

6.3.3　基于 WSDL 优先的 JAX-WS 前端模式实现的复杂例子

本程序演示 Apache CXF 框架遵循 JAX-WS 规范的客户端和服务器端调用，客户端采用 Apache CXF 的 WSDL 优先（WSDL First）前端应用模式，采用 POJO 对象作为中间传输对象。由于这个例子的基本技术与“基于 WSDL 优先的 JAX-WS 前端模式实现的简单例子”相同，在这里就不再赘述，有兴趣的读者可以参看具体的例子程序。

本例代码请参见配套资源代码中“第 06 章\frontend_contractfirst_jaxws_hashmap_servlet”文件夹。

6.4　简化前端模式（Simple Frontend）

Apache CXF 包括一种简单前端模式（Simple Frontend），即通过反射来创建服务。

6.4.1　简化前端模式（Simple Frontend）介绍

简化前端模式（Simple Frontend），与 JAX-WS 实现模式不同，简化前端模式在开发和部署 Web Services 时不提供任何正式的规范或标准。简化前端模式采用简单工厂模式创建服务。工厂组件采用 Java 反射 API 去动态地创建服务组件和客户端组件。这种模式易于使用，不需要任何工具来构建服务。

表 6-11 解释了 JAX-WS 和简化前端模式的区别。

表 6-11　JAX-WS和简化前端模式的区别

JAX-WS	简化前端模式
JAX-WS基于Sun的Java API规范来规定开发和部署Web Services	基于Java反射机制来开发和部署Web Services
支持Java 5注释	不支持注释

默认情况下简化前端模式使用 JAXB 数据绑定。

简化前端模式的服务端使用 ServerFactoryBean 对象，客户端采用 ClientProxyFactoryBean 对象。ServerFactoryBean 产生一个服务器实例，它需要一个服务类和地址公布于服务。通过创建一个 Server，可以提供服务给外部世界。

6.4.2　采用 Apache CXF 简化前端实现的例子

本程序演示 Apache CXF 自身简化模式的客户端和服务器端调用，客户端采用 Apache CXF 的代码优先（Java First）前端应用模式，采用 POJO 对象作为中间传输对象。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，都是使用 POJO 对象实现客户端和服务器端的数据交换。前端客户端采用

Apache CXF 自身规范的 ClientProxyFactoryBean 对象来实现，后台服务端为遵循 Apache CXF 自身规范的 ServerFactoryBean 对象来实现，其实现结构如图 6-26 所示。

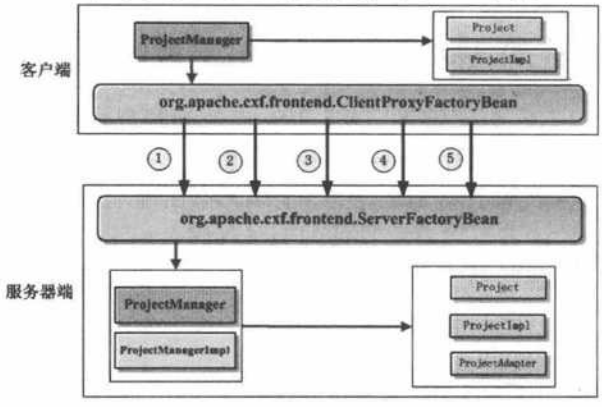


图 6-26　采用 Apache CXF 简化前端实现图

其中：① 表示客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。客户端和服务端的 SEI 都是 ProjectManager 接口。

在编写本程序例子，主要与“采用 POJO 并基于 JAX-WS 服务的前端实现”进行比较。

2. 应用程序案例实现步骤

创建 Apache CXF 简化前端实现 Web Services 程序的流程如图 6-27 所示。

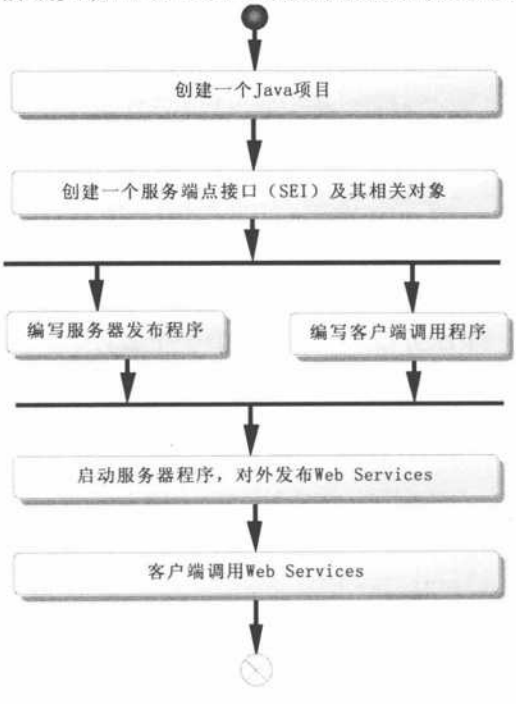


图 6-27　应用程序案例实现流程图

步骤说明如下：

（1）创建一个 Java 项目。

（2）创建一个服务端点接口（SEI）及其相关对象。

（3）编写发布这个 Java 程序为 Web Services 的服务端程序。

（4）编写 Web Services 的客户端应用程序。

（5）启动服务器程序，对外发布 Web Services。

（6）客户端调用 Web Services。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 6-28 所示。

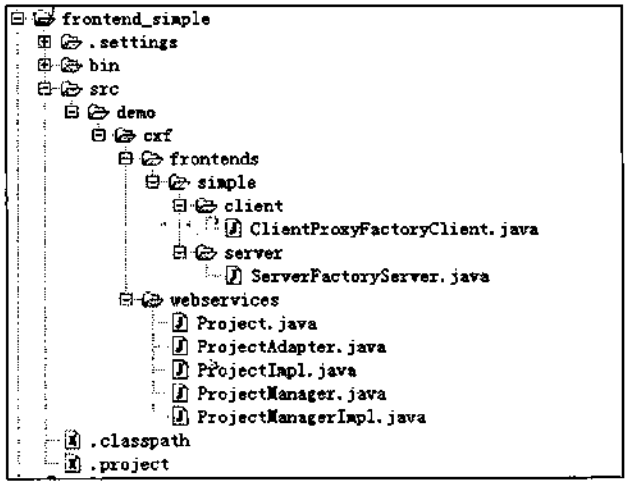


图 6-28　frontend_simple 程序结构

包目录包括三个部分，第一个包目录 demo.cxf.frontends.simple.server 是服务器程序目录，第二个包目录 demo.cxf.frontends.simple.client 是客户端程序目录。第三个包目录 demo.cxf.webservices 是 SEI 和相关对象等公用部分，客户端和服务器端都可能引用其内容。

服务器端点接口（SEI）是公用部分的接口，服务器端和客户端都必须引用。但服务器端和客户端都对公用部分的引用是不同的。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口，而服务器端程序对于接口和实现类都要引用。

与采用 POJO 并基于 JAX-WS 服务的前端实现相比，客户端和服务端的调用对象不同，SEI 也有一点点差别，其余的内容全部相同。下面只列出有区别的三个文件的代码，分别是服务器端点接口 SEI、服务器端发布程序和客户端调用程序。

（1）服务器端点接口（SEI）和相关对象

首先要介绍的是服务器端点接口（SEI）。

第一个文件是 SEI，即 ProjectManager.java。基于 JAX-WS 服务的 SEI 必须有 Java 注释，而本案例程序的代码不需要 Java 注释，代码如下：

```
package demo.cxf.webservices;

public interface ProjectManager {
    public String greeting() ;
    public String greetingWithParameter(String greet);
    public String getProjectName();
    public String getProjectNameByID(String projectID);
    public void updateProjectName (String projectName);
    public void setDefaultProject(Project project);
    public void registerProject(Project project);
    public void removeProject(String id);
    public Project getDefaultProject();
    public void getAllProject();
}
```

（2）服务器的发布程序

第二个文件是服务器的发布代码，采用实例化 ServerFactoryBean 对象来生成一个服务器实例，代码如下：

```
package demo.cxf.frontends.simple.server;

import org.apache.cxf.frontend.ServerFactoryBean;
import demo.cxf.webservices.*;

public class ServerFactoryServer {
    public ServerFactoryServer() {
        ProjectManager projectManager = new ProjectManagerImpl();
        ServerFactoryBean svrFactory = new ServerFactoryBean();
       svrFactory.setServiceClass(ProjectManager.class);
       svrFactory.setAddress("http://localhost:9000/ProjectManager");
       svrFactory.setServiceBean(projectManager);
       svrFactory.create();
    }

    public static void main(String[] args) throws Exception{
        new ServerFactoryServer();
       System.out.println("ServerFactoryBean Server ready...");

       Thread.sleep(1 * 60 * 1000);
       System.out.println("ServerFactoryBean Server exiting.");
       System.exit(0);
    }
}
```

服务器端要实例化 ServerFactoryBean 对象，然后对实例化的 ServerFactoryBean 对象进行赋值，最后调用 ServerFactoryBean 实例化对象的 create 方法形成一个对外发布的 Web Services 服务。

（3）客户端调用程序

第三个文件是客户端调用服务的代码，采用实例化 ClientProxyFactoryBean 对象来生成一个客户端代理实例，代码如下：

```
package demo.cxf.frontends.simple.client;

import org.apache.cxf.frontend.ClientProxyFactoryBean;

import demo.cxf.webservices.ProjectManager;
```

```
import demo.cxf.webservices.Project;
import demo.cxf.webservices.ProjectImpl;

public class ClientProxyFactoryClient {

    public ClientProxyFactoryClient() {}

    public static void main(String[] args) throws Exception {
        ClientProxyFactoryBean factory = new ClientProxyFactoryBean();
        factory.setServiceClass(ProjectManager.class);
        if (args != null && args.length > 0 && !"".equals(args[0])) {
            factory.setAddress(args[0]);
        } else {
            factory.setAddress("http://localhost:9000/ProjectManager");
        }
        ProjectManager projectManager = (ProjectManager) factory.create();


        //项目的基本情况
        getGreet(projectManager);
        greetingWithParameter(projectManager);
        getProjectName(projectManager);
        getDefaultProject(projectManager);
        getAllProject(projectManager);

        //修改默认项目，查看项目的基本情况
        setDefaultProject(projectManager);

        //增加一个项目，查看项目的基本情况
        registerProject(projectManager);

        //删除一个项目，查看项目的基本情况
        removeProject(projectManager);

        System.exit(0);
    }

    //调用 SEI 的 getGreet 方法
    private static void getGreet(ProjectManager projectManager)throws Exception {
        System.out.println(projectManager.greeting());
    }

    //调用 SEI 的 greetingWithParameter 方法
    private static void greetingWithParameter(ProjectManager projectManager)throws
Exception {
        System.out.println(projectManager.greetingWithParameter("greet"));
    }

    //调用 SEI 的 getProjectName 方法
    private static void getProjectName(ProjectManager projectManager)throws
Exception {
        System.out.println(projectManager.getProjectName());
    }

    //调用 SEI 的 getDefaultProject 方法
    private static void getDefaultProject(ProjectManager projectManager)throws
Exception {
        Project defaultProject = projectManager.getDefaultProject();
```

```
        System.out.println("defaultProjectID is " + defaultProject.getProjectID()
                + " . " + "defaultProjectName is " + defaultProject.getProjectName());

    }

    //调用 SEI 的 setDefaultProject 方法
    private  static  void  setDefaultProject(ProjectManager  projectManager)throws
Exception {
        Project project = new ProjectImpl();
        project.setProjectID("New Default ProjectID");
        project.setProjectName("New Default ProjectName");

        projectManager.setDefaultProject(project);
        Project newProject = projectManager.getDefaultProject();
        System.out.println("DefaultProjectID is " + newProject.getProjectID()
                + " . " + "DefaultProjectName is " + newProject.getProjectName());
        projectManager.getAllProject();
    }

    //调用 SEI 的 registerProject 方法
    private static void registerProject(ProjectManager projectManager)throws Exception {
        Project project = new ProjectImpl();
        project.setProjectID("New ProjectID");
        project.setProjectName("New ProjectName");

        projectManager.registerProject(project);
        System.out.println(projectManager.getProjectNameByID("New ProjectID"));
        System.out.println(projectManager.getProjectNameByID("New ProjectID"));
        projectManager.getAllProject();
    }

    //调用 SEI 的 removeProject 方法
    private static void removeProject(ProjectManager projectManager)throws Exception {
        projectManager.removeProject("New ProjectID");
        projectManager.getAllProject();
    }

    //调用 SEI 的 getAllProject 方法
    private static void getAllProject(ProjectManager projectManager) throws Exception{
        projectManager.getAllProject();
    }

}
```

客户端首先要实例化一个 ClientProxyFactoryBean 对象，然后通过 ClientProxyFactoryBean 对象的 create 方法来创建一个 SEI，这样就可以采用 SEI 来访问 Web Services 服务了。

4. 测试和运行结果

测试编写好的程序代码，首先要启动服务器程序，即在 Eclipse 上运行 Server.java，然后在 Eclipse 上运行客户端程序，程序的运行结果与采用 POJO 并基于 JAX-WS 服务的前端实现是一样的。

5. 其他内容

本例代码请参见配套资源代码中“第 06 章\frontend_simple”文件夹。

6.4.3　采用 Apache CXF 简化前端实现的 Servlet 例子

上述采用 POJO 的例子主要是基于后台启动的 Web Services 服务，也可以把这个服务委托给 Servlet 容器来进行管理。本例采用 Tomcat 来处理。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这些过程中，都使用 POJO 对象来实现客户端和服务器端的数据交换。后台服务端为遵循 JAX-WS 规范的 Endpoint 来实现。前端访问采用 Apache CXF 的简化客户端模式，实现结构如图 6-29 所示。

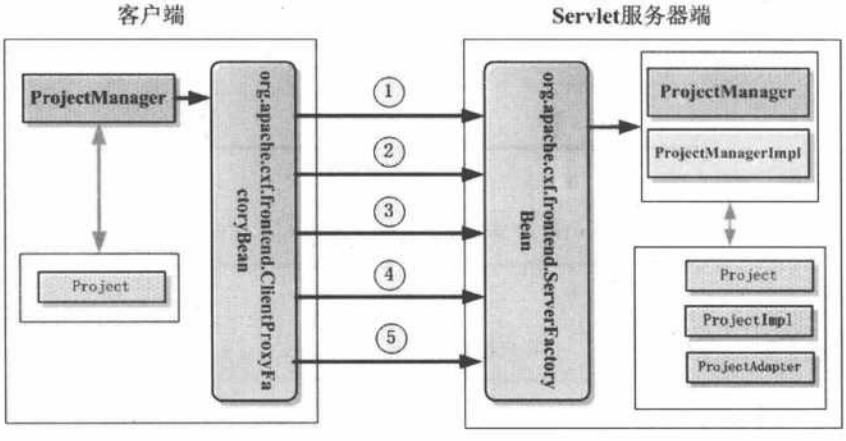


图 6-29　采用 Apache CXF 简化前端，Servlet 后端应用模式图

其中：① 表示 Apache CXF 客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。客户端和服务端的 SEI 都是 ProjectManager 接口。

2. 应用程序案例实现步骤

要创建基于 Servlet 的 Apache CXF 简化前端实现的 Web Services 程序实现，流程如图 6-30 所示。

步骤说明如下：

（1）创建一个 Web 服务项目。

（2）创建一个服务端点接口（SEI）及其相关对象和注释。

（3）编写 web.xml 和 server- bean.xml。

（4）发布配置文件到 Servlet 服务器。

（5）编写 Web Services 的客户端应用程序。

（6）启动 Servlet 服务器程序，对外发布 Web Services。

（7）客户端调用 Web Services。

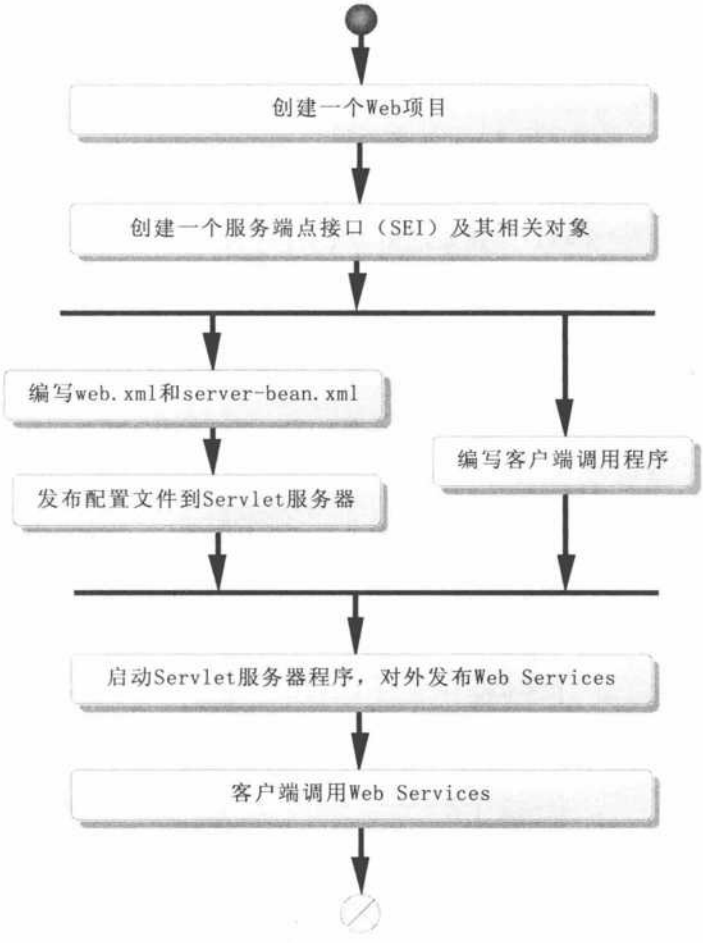


图 6-30　应用程序案例实现流程图

其中，第 1 步和第 3 步与上述采用自带的 HTTP 服务器是类似的。区别主要在第 2 步，该步不是采用自带的服务，而是放置到 Servlet 容器中。我们采用与 Spring 框架集成的方式来实现。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 6-31 所示。

包目录包括三个部分，第一个包目录 WebRoot 目录是服务器程序目录，包括 web.xml 和 server-bean.xml 文件；第二个包目录 demo.cxf.frontends.simple.client 是客户端程序目录；第三个包目录 demo.cxf.webservices 是 SEI 和数据对象等公用部分，客户端和服务器端都可能引用其内容。

服务器端点接口（SEI）是公用部分的接口。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口。由于本程序是直接发布到 Servlet 容器，所以没有单独编写服务器发布程序，直接把配置文件和编译好的服务器端点接口（SEI）组件等部署到 Servlet 容器即可。

下面分别介绍两部分：①服务器端配置文件；②客户端调用程序。

图 6-31　frontend_simple_servlet 程序结构

（1）服务器端配置文件

服务器端配置文件包括 web.xml 文件和 server-bean.xml 文件。

web.xml 文件的定义与采用 POJO 并基于 Servlet 容器的前端实现案例相同，这里不再赘述。其中，Servlet 启动时，将读取 WEB-INF 目录下的 server-bean.xml 文件，server-bean.xml 文件的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:simple="http://cxf.apache.org/simple"
    xmlns:soap="http://cxf.apache.org/bindings/soap"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
        http://cxf.apache.org/bindings/soap
        http://cxf.apache.org/schemas/configuration/soap.xsd
        http://cxf.apache.org/simple
        http://cxf.apache.org/schemas/simple.xsd">

    <import resource="classpath:META-INF/cxf/cxf.xml" />
    <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
    <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />

    <simple:server id="ProjectManager"
        serviceClass="demo.cxf.webservices.ProjectManager"
        address="/ProjectManager">
        <simple:serviceBean>
            <bean class="demo.cxf.webservices.ProjectManagerImpl" />
        </simple:serviceBean>
    </simple:server>
</beans>
```

服务端的程序代码基本不做修改，把服务端程序发送到 Tomcat 中。

当 Servlet 容器启动，调用 web.xml 进行初始化，若有 listener 选项，调用其内容，为 org.springframework.web.context.ContextLoaderListener，然后调用 server-bean.xml 文件，从 jaxws:endpoint 元素中了解到后台的 SEI 是 demo.cxf.webservices.ProjectManagerImpl，公开 Web Services 服务。

采用 JAX-RS 规范的前端客户端，只需修改后台服务的位置即可，代码如下：

```
……
    public class ClientProxyFactoryClient {
……

      public static void main(String[] args) throws Exception {
           ClientProxyFactoryBean factory = new ClientProxyFactoryBean();
           factory.setServiceClass(ProjectManager.class);
           if (args != null && args.length > 0 && !"".equals(args[0])) {
                factory.setAddress(args[0]);
} else {
    factory.setAddress("http://localhost:18080/frontend_simple_servlet/ProjectManager");
        }
        ProjectManager projectManager = (ProjectManager) factory.create();

      ……
}
```

注意：笔者的 Tomcat 采用的是 18080 端口。

（2）客户端程序

Apache CXF 的前端客户端修改与基于 JAX-WS 规范相同。

4. 测试和运行结果

测试编写好的程序代码，首先要启动 Tomcat 服务器程序，然后在 Eclipse 上运行客户端程序，其客户端程序和服务器端输出结果与采用 POJO 并基于 JAX-WS 服务的前端实现是一样的。

5. 其他内容

本例代码请参见配套资源代码中“第 06 章\frontend_simple_servlet”文件夹。

6.5 Provider/Dispatch 服务前端应用模式

通过 SEI 的调用都是高层次上的 Web Services，主要针对服务方法进行调用。但在有些应用场景下，开发人员需要对服务内的 XML 进行更深层次的解析和处理，这就需要采用新的工具和方法。

JAX-WS API 的 Provider 和 Dispatch 接口用于对底层 XML 进行操作。

6.5.1 Provider/Dispatch 服务前端应用模式介绍

当 Web Services 客户端和 Web Services 服务端之间通过 XML 来传输时，信息非常大，同时，开发人员也不想在 Java 对象和 XML 消息之间的转换上开销大。在这种情况下，使用 Provider 和 Dispatch 就非常有优势。作为 Web Services 的组成部分，开发人员可以直接处理 XML，或者采用

更有效的方式来解析 XML，从而不必把这些工作全部都提交给 Web Services 框架去处理。

JAX-WS 规范中的 javax.xml.ws.Provider 接口提供了一个创建和执行服务 Provider 接口的功能，该功能可以处理 XML 消息。Dispatcher 客户端可以通过 XML 消息格式来提交请求，同时遵照相同格式来获得响应。Provider 实现类作为一个服务器端点在服务器上进行发布。另一方面，JAX-WS 规范的 javax.xml.ws.Dispatch 接口用于处理 XML 消息和采用 XML 格式发送响应到服务 Provider。

由于 Provider 服务和 Dispatch 服务都能解析原始 XML 文件，开发人员需要了解消息传递模式和服务对象的类型。这样可以对交换信息属性有更好的理解。后面将介绍开发 Provider 服务和 Dispatch 服务的实现。本节将按照以下顺序来进行说明：了解消息传递模式，理解服务对象的类型，实现 Provider 服务，实现 Dispatch 服务，发布 Provider 服务。

6.5.1.1　JAX-WS Dispatch API

JAX-WS 规范 Dispatch API 介绍包括概述、数据类型和使用方法。

1. JAX-WS Dispatch 概述

Dispatch 对象有两种使用模式：

- 消息模式（Message mode）
- 负载模式（Payload mode）

在开发过程中，开发人员可指定 Dispatch 实例化对象的使用模式并使用特定的用户级代码。

消息模式（Message mode）：在消息模式下，一个 Dispatch 实例化对象与一个复杂消息一块工作。该复杂消息包括具有绑定的消息头定义和包装体。例如，消费者调用一个服务，该服务需要有一个 SOAP 消息来提供给 Dispatch 对象的 invoke 方法调用并返回一个指定的 SOAP 消息。消费者代码负责完成和读取 SOAP 消息报头和 SOAP 消息内封装的信息。如果使用 JAXB 对象，消息模式不会被处理。当创建 Dispatch 对象时，可以通过在程序代码中设置 java.xml.ws.Service.Mode.MESSAGE 属性来指定调用消息模式。

负载模式（Payload mode）：负载模式，也称为消息有效负载模式，Dispatch 实例化对象只与一个消息有效负载协同工作。例如，在负载 Dispatch 对象的模式下，工作只有一个 SOAP 消息体。绑定层处理任何绑定的包装和标头。当被一个 invoke 方法调用时，其返回值中绑定层包装和包头已经被分离开了，仅仅消息的正文被留下。当创建 Dispatch 对象时，可以通过指定 java.xml.ws.Service.Mode.PAYLOAD 属性来指定调用有效载荷模式。

2. Dispatch 对象的数据类型

Dispatch 对象因为是更低层次的对象，因此不能使用 JAXB 等高层次的 API 来优化类型。Dispatch 对象有以下几种对象：

- javax.xml.transform.Source
- javax.xml.soap.SOAPMessage
- javax.activation.DataSource
- JAXB

下面分别介绍 javax.xml.transform.Source 对象、javax.xml.soap.SOAPMessage 对象、javax.

activation.DataSource 对象和 JAXB 对象。

（1）javax.xml.transform.Source 对象

Dispatch 对象可以接受和返回从 javax.xml.transform.Source 接口继承的对象。javax.xml.transform.Source 接口对象是具有支持 XML 文档的低层次对象。每个 javax.xml.transform.Source 接口对象的实现提供了访问存储的方法和操作 XML 文档的内容。实现 javax.xml.transform.Source 接口的对象包括 DOMSource、SAXSource、StreamSource 等。

（2）javax.xml.soap.SOAPMessage 对象

Dispatch 对象可以使用 javax.xml.soap.SOAPMessage 对象，当下列条件为真时：①Dispatch 对象使用 SOAP 绑定；②Dispatch 对象使用消息模式。

（3）javax.activation.DataSource 对象

当下列条件为真时，Dispatch 对象可以使用对象来实现 javax.activation.DataSource 接口：①Dispatch 对象使用 HTTP 绑定；②Dispatch 对象采用消息模式。

DataSource 对象提供一种与 MIME 类型协同工作的机制，该机制可以来源于一系列数据源，如网址、数据文件和字节数组。

（4）使用 JAXB 对象

虽然 Dispatch 对象是低层次的 API，但允许开发者使用原始消息来调用 JAXB 对象。若要使用 JAXB 对象，Dispatch 对象必须通过 JAXBContext 知道如何编组和反编组 JAXB 对象。一旦 Dispatch 对象被创建，JAXBContext 就将产生。

开发人员可以通过用带参数的 invoke 方法来调用 JAXBContext 对象，全面掌握和了解 JAXB 对象。开发者还可以通过 JAXBContext 对象的返回消息对象来加深对 JAXB 对象的理解。

3. Dispatch 对象的使用

要使用 Dispatch 对象来调用远程服务，需要执行下列操作：

（1）创建 Dispatch 对象。

（2）构建请求消息。

（3）调用适当的 invoke()方法。

（4）解析响应消息。

要创建 Dispatch 对象需要执行下列操作：

（1）基于 wsdl:service 元素定义来创建一个服务对象代表，表明 Dispatch 对象将会被调用。

（2）使用服务对象的 createDispatch 方法来创建 Dispatch 对象。

```
public Dispatch<T> createDispatch(QName portName, java.lang.Class<T> type,
Service.Mode mode)
throws WebServiceException;
```

如果开发人员使用 JAXB 对象，则 createDispatch 方法用法如下：

```
public Dispatch<T> createDispatch(QName portName, javax.xml.bind.JAXBContext
context, Service.Mode mode)
throws WebServiceException;
```

表 6-12 描述了 createDispatch 方法的参数。

表 6-12　createDispatch方法的参数和描述

参数	描述
portName	指定wsdl:port元素的QName，该元素代表了Dispatch对象将调用的服务提供者
type	指定Dispatch对象使用的对象数据类型
mode	指定Dispatch对象使用的调度模式

下面代码显示了在负载模式下，创建一个 Dispatch 对象与 DOMSource 对象协同工作。

```
package com.iona.demo;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class Client{
public static void main(String args[])  {
   QName serviceName = new QName("http://org.apache.cxf", "stockQuoteReporter");
   Service s = Service.create(serviceName);

   QName portName = new QName("http://org.apache.cxf", "stockQuoteReporterPort");
   Dispatch<DOMSource>  dispatch  =  createDispatch(portName,  DOMSource.class,
Service.Mode.PAYLOAD);
   ...
```

当与 Dispatch 对象一起工作时，消息必须从头开始。开发人员应当负责确保信息传递给匹配的 Dispatch 对象。这需要服务提供者知道精确的知识信息是什么。如果存在精确的知识信息，服务提供者需要头信息。这些信息可以提供一个 WSDL 文档或 XML Schema 文档定义的消息。虽然服务提供者有很大差异，但应遵循以下几个原则：

- 该请求的根元素对应于 wsdl:operation 元素的 name 属性值，其操作被调用。
- 该请求的根元素受到命名空间的限定。
- 如果该服务被调用，使用 RPC/文字消息，命名空间不会限定请求中的顶级元素。
- 如果该服务被调用，使用 RPC /文字信息的顶级元素都不能为空。
- 如果该服务被调用，使用文档/文字消息，该消息的架构定义确定是否有命名空间限定的元素。

消费者产生一个响应同步调用，使用 Dispatch 对象的 invoke 方法，代码如下：

```
T invoke(T msg) throws WebServiceException;
```

当 Dispatch 对象被创建，invoke 方法传递的响应和请求将被定义。例如，开发人员使用 createDispatch(portName, SOAPMessage.class, Service.Mode.MESSAGE)方法来创建一个 Dispatch 对象，响应和请求都是 SOAPMessage 对象。

下面的代码使用 DOMSource 对象实现同步调用远程服务。

```
// Creating a DOMSource Object for the request
DocumentBuilder db = DocumentBuilderFactory.newDocumentBuilder();
Document requestDoc = db.newDocument();
Element  root  =  requestDoc.createElementNS("http://org.apache.cxf/stockExample",
"getStockPrice");
```

```
root.setNodeValue("DOW");
DOMSource request = new DOMSource(requestDoc);

// Dispatch disp created previously
DOMSource response = disp.invoke(request);
```

Dispatch 对象也支持异步调用。Dispatch 对象可使用投票方法和回调方法。

当使用投票方式，invokeAsync 方法返回一个 Response<t>对象，可以定期轮询，主要用于查看响应是否有反馈。

```
Response <T> invokeAsync(T msg) throws WebServiceException;
```

当使用回调方式，invokeAsync 方法接受一个 AsyncHandler 实现对象的返回值，并处理返回的响应。

```
Future<?> invokeAsync(T msg, AsyncHandler<T> handler)
throws WebServiceException;
```

当一个请求不会获得一个响应时，可使用 Dispatch 对象的 invokeOneWay 方法来进行远程调用，代码如下：

```
void invokeOneWay(T msg) throws WebServiceException;
```

当 Dispatch 对象被创建，用于封装请求的对象类型被确定。例如，Dispatch 对象使用 createDispatch（portName，DOMSource.class，Service.Mode.PAYLOAD）方法，请求将会被打包成一个 DOMSource 对象。

下面的代码演示使用 JAXB 对象调用远程服务。

```
// Creating a JAXBContext and an Unmarshaller for the request
JAXBContext jbc = JAXBContext.newInstance("org.apache.cxf.StockExample");
Unmarshaller u = jbc.createUnmarshaller();

// Read the request from disk
File rf = new File("request.xml");
GetStockPrice request = (GetStockPrice)u.unmarshal(rf);

// Dispatch disp created previously
disp.invokeOneWay(request);
```

6.5.1.2 Provider 服务

JAX-WS 规范 Provider 服务包括概述、数据类型和使用方法。

1. JAX-WS Provider 概述

实现 Provider 接口的对象有两种模式：

- 消息模式（Message mode）
- 有效载荷模式（Payload mode）

开发人员指定的模式决定了实现传递的详细信息。

在消息模式下，一个 Provider 实例化对象与一个复杂消息一起工作。该复杂消息包括具有绑定的消息头定义和包装信息。例如，消费者调用一个服务，该服务需要有一个 SOAP 消息来提供

Provider 实例化对象的 invoke 方法并返回一个指定的 SOAP 消息。所用 Provider 实例化对象的返回值都被定义为 SOAP 消息。

当创建 Dispatch 对象时，可以通过设置 java.xml.ws.Service.Mode.MESSAGE 属性来指定调用消息模式。

开发者通过设定 javax.xml.ws.ServiceMode 注释的值为 java.xml.ws.Service.Mode.MESSAGE 来指定一个 Provider 实现方式，代码如下所示：

```
@WebServiceProvider
@ServiceMode(value=Service.Mode.MESSAGE)
public class stockQuoteProvider implements Provider<SOAPMessage>
{
  ...
}
```

在负载模式下，Provider 实施仅仅提供了有且只有一个消息的负载。例如，在负载工作模式下，一个 Provider 实现对象只能在一个 SOAP 消息体内。绑定层处理全部绑定的封装体和消息头。

开发者通过设定 javax.xml.ws.ServiceMode 注释的值为 java.xml.ws.Service.Mode.PAYLOAD 来指定一个 Provider 实现方式，代码如下所示：

```
@WebServiceProvider
@ServiceMode(value=Service.Mode.PAYLOAD)
public class stockQuoteProvider implements Provider<DOMSource>
{
  ...
}
```

如果不提供@ ServiceMode 注释，该 Provider 的实例化将默认为负载模式。

2. JAX-WS Provider 数据类型

因为 Provider 对象是更低层次的对象，因此不能使用 JAXB 等高层次的 API 来优化类型。Provider 对象有以下几种对象：

- javax.xml.transform.Source
- javax.xml.soap.SOAPMessage
- javax.activation.DataSource

下面分别介绍 javax.xml.transform.Source 对象、javax.xml.soap.SOAPMessage 对象和 javax.activation.DataSource 对象。

（1）javax.xml.transform.Source 对象

Provider 对象可以接受和返回一个或多个实现 javax.xml.transform.Source 接口的对象。javax.xml. transform.Source 对象是支持 XML 文档的低层次对象。每个 javax.xml.transform.Source 对象的实现提供了访问存储的方法和操作 XML 文档的内容。以下 DOMSource 类和 SAXSource 类都实现 javax.xml.transform.Source 接口。

DOMSource 类作为一个文档对象模型（DOM）树的 XML 消息。 XML 消息存储为一个节点对象集，可以通过 getNode 方法访问。节点可以用 setNode 方法被更新或被添加到 DOM 树上。

SAXSource 类持有作为一种对象的简单 API（SAX）的 XML 消息。SAX Source 对象包含一个 InputSource 对象，它包含了原始数据和一个 XmlReader 对象的解析原始数据。StreamSource

持有数据流的 XML 消息。该数据流可以被任何其他数据流操纵。

当使用 Source 对象，开发人员负责确保所有必需的、具有约束力的具体包装被添加到消息。例如，当一个服务交互了期待的 SOAP 消息，开发人员必须确保所需的 SOAP 信封添加到发出的请求上，保证 SOAP 封装的内容是正确的。

（2）SOAPMessage 对象

当下列条件为真，Provider 接口实现可以使用 javax.xml.soap.SOAPMessage 对象：

- Provider 的实现模式采用 SOAP 绑定。
- Provider 的实现模式采用消息模式。

SOAPMessage 对象，顾名思义，拥有一个 SOAP 消息。它们包含一个 SOAPPart 对象和 0 个或多个 AttachmentPart 对象。该 SOAPPart 对象包含 SOAP 指定的 SOAP 消息部分，如 SOAP 封装的特定部分、任何 SOAP 头和 SOAP 消息体。该 AttachmentPart 对象包含了作为附件的二进制数据。

（3）DataSource 对象

当下列条件为真，Provider 接口实现可以使用一个实现 javax.activation.DataSource 接口的对象：

- Provider 实现模式采用 HTTP 绑定。
- Provider 实现模式采用消息模式。

DataSource 对象提供一种与 MIME 类型协同工作的机制，该机制可以来源于一系列数据源，如网址、数据文件和字节数组。

3. 实现 Provider 接口

一般来说，Provider 接口还是较容易实现的。因为，它只有一个方法——invoke。此外，它有三个简单的要求：①Provider 接口实现必须有一个@WebServiceProvider 注释；②Provider 接口实现必须有一个默认的构造函数；③Provider 接口必须有实现版本的方法。

换句话说，开发人员不能实现一个 Provider<T>接口，而必须实现一个使用的具体数据类型接口的版本。例如，开发人员可以编写代码来实现一个 Provider<SAXSource>实例对象。

Provider 接口的复杂实现是处理请求消息和创建适当的响应。

不像高层次 SEI 服务的实现，Provider 接口实现对象接收原始 XML 数据的要求，并使用原始 XML 数据作为响应。这就要求开发人员对服务的消息构成细节有深入的了解。这些细节通常可以在 WSDL 描述文档服务中找到。

WS-I 基本概要，包括提供有关服务所使用的信息准则：

- 请求的根元素是基于 wsdl:operation 元素下 name 属性值对应于操作被调用的结果。
- 所有消息的根元素将被命名空间限定。
- 如果服务使用 rpc/literal 消息，消息的顶层元素不会被命名空间限定。
- 如果服务使用 rpc/literal 消息，顶层元素都不能为空。
- 如果服务使用 rpc/literal 消息，该消息的 schema 定义确定是否有元素的命名空间被限定。

如果服务使用 doc/literal 消息，请求的根元素基于被 wsdl:operation 元素所引用的 wsdl:part 元素下 name 属性的值。

高层元素的子元素可能被命名空间限定，所以确定使用时，需要检查它们的 schema 定义。

Provider 接口只有一个方法，即 invoke 方法，该方法需要编写代码来实现功能。invoke 方法

接收传入的请求，该请求被封装成由 Provider 接口实现的对象类型。invoke 方法同时返回传出的响应，该响应被封装成由 Provider 接口实现的对象类型。例如，一个 Provider<SOAPMessage>接口的实现对象将需要一个 SOAPMessage 对象作为参数来实现请求和响应。

消息传递模式中，使用的 Provider 实现必须确定请求和响应有绑定特定封装和消息头。使用消息模式的实现接受请求消息中所有绑定特定封装和消息头。这些请求消息还必须添加响应消息的特定封装和消息头。使用负荷模式的实现仅仅接受请求的正文。使用负荷模式实现的 XML 文档将被放置到请求消息的正文中。

下面显示了在消息模式下，一个 Provider 接口与 SOAPMessage 对象的实现。

```
import javax.xml.ws.Provider;
import javax.xml.ws.Service;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceProvider;

@WebServiceProvider(portName="stockQuoteReporterPort"
serviceName="stockQuoteReporter")
@ServiceMode(value="Service.Mode.MESSAGE")
public class  stockQuoteReporterProvider implements Provider<SOAPMessage>
{
public stockQuoteReporterProvider()  { }

public SOAPMessage invoke(SOAPMessage request)  {
  SOAPBody requestBody = request.getSOAPBody();
  if(requestBody.getElementName.getLocalName.equals("getStockPrice"))  {
    MessageFactory mf = MessageFactory.newInstance();
    SOAPFactory sf = SOAPFactory.newInstance();

    SOAPMessage response = mf.createMessage();
    SOAPBody respBody = response.getSOAPBody();
    Name bodyName = sf.createName("getStockPriceResponse");
    respBody.addBodyElement(bodyName);
    SOAPElement respContent = respBody.addChildElement("price");
    respContent.setValue("123.00");
    response.saveChanges();
    return response;
    }
    ...
  }
}
```

该代码执行以下操作：

（1）指定程序后面的类对象，实现一个提供服务的对象，其实现的 wsdl:service 命名为 stockQuoteReporter，wsdl:port 元素被命名为 stockQuoteReporterPort。

（2）在消息模式下指定 Provider 实现。

（3）提供所需的默认的公共构造函数。

（4）提供 invoke 方法，可以返回一个实现 SOAPMessage 接口的对象。

（5）传入的 SOAP 消息体抽象出请求消息。

（6）检查请求的消息的根元素，以确定如何处理该请求。

（7）创建一个需要使用响应的工厂。

（8）创建一个 SOAP 消息的响应。

（9）作为一个 SOAPMessage 对象返回一个响应。

下面的例子显示了在负载模式下使用 DOMSource 对象作为参数的 Provider 接口实现情况，其实现的接口类是 stockQuoteReporterProvider，代码如下：

```
import javax.xml.ws.Provider;
import javax.xml.ws.Service;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceProvider;

@WebServiceProvider(portName="stockQuoteReporterPort"
serviceName="stockQuoteReporter")
@ServiceMode(value="Service.Mode.PAYLOAD")
public class  stockQuoteReporterProvider implements Provider<DOMSource>
public stockQuoteReporterProvider() {
  }

public DOMSource invoke(DOMSource request) {
    DOMSource response = new DOMSource();
    ...
    return response;
  }
}
```

该代码执行以下操作：

（1）指定本类实现的是一个 Provider 接口，其实现映射 wsdl:service，故其名为 stockQuoteReporter，其映射端口由 wsdl:port 元素定义，所以被命名为 stockQuoteReporterPort。

（2）指定采用负载模式来实现 Provider 接口。

（3）提供所需的默认的公共构造函数。

（4）提供 invoke 方法的实现，可以使用一个 DOMSource 并返回一个 DOMSource 对象。

6.5.2 采用 DOMSource（message）的 Provider/Dispatch 前端模式实现例子

本程序演示 Apache CXF 采用 Provider/Dispatch 前端模式实现客户端和服务器端调用，客户端采用 Dispatch 的 DOMSource（message）前端应用模式。后台采用 Provider 方式，主要处理大型的、复杂的 XML 文件。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个存储信息的 XML 文件，通过客户端访问服务器，获得后台的 XML 文件的数据。在这个过程中，都是使用 XML 文件实现客户端和服务器端的数据交换。前台客户端采用遵循 JAX-WS 规范的 Service 对象，后台服务端也由遵循 JAX-WS 规范的 Endpoint 对象来实现，实现结构如图 6-32 所示。

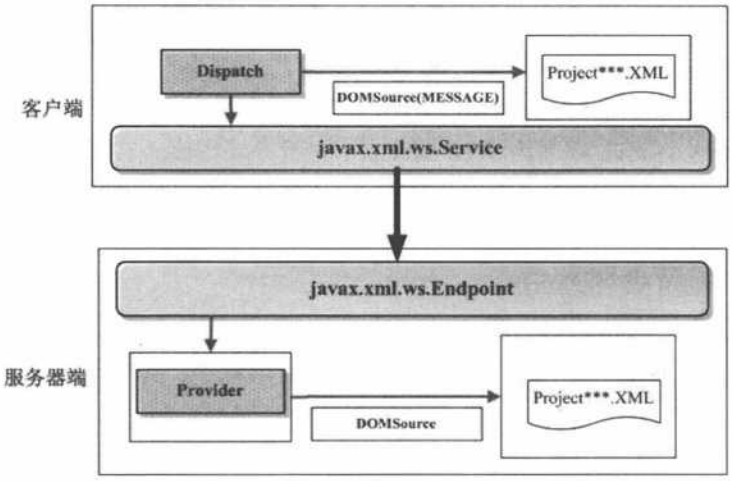


图 6-32　采用 Apache CXF 简化前端实现图

图中的粗箭头表示 JAX-WS 客户端 Service 通过 Endpoint 获得 Provider，访问后台的 XML 文件，然后返回给前端的 Dispatch 进行处理。

2. 应用程序案例实现步骤

要创建 DOMSource（message）的 Provider/Dispatch 前端模式实现 Web Services 程序的流程如图 6-33 所示。

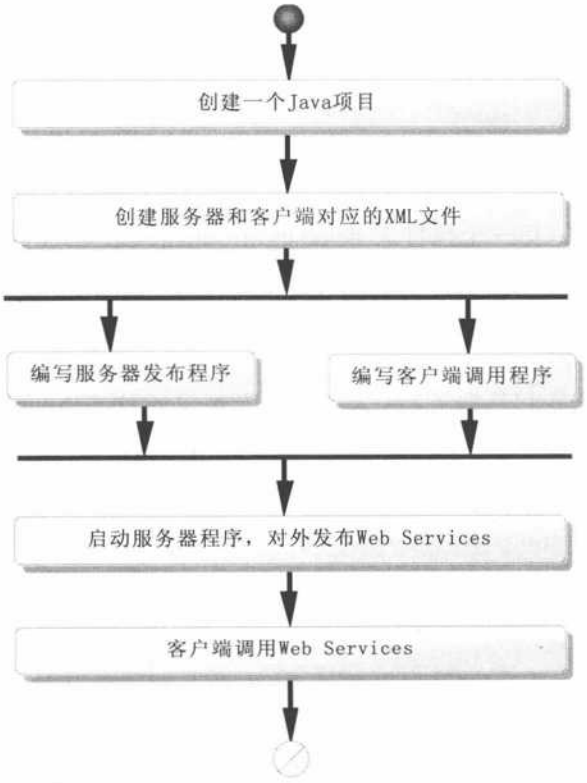


图 6-33　应用程序案例实现流程图

步骤说明如下：

（1）创建一个 Java 项目。

（2）创建服务器和客户端对应的 XML 文件。

（3）编写发布这个 Java 程序为 Web Services 的服务端程序。

（4）编写 Web Services 的客户端应用程序。

（5）启动服务器程序，对外发布 Web Services。

（6）客户端调用 Web Services。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 6-34 所示。

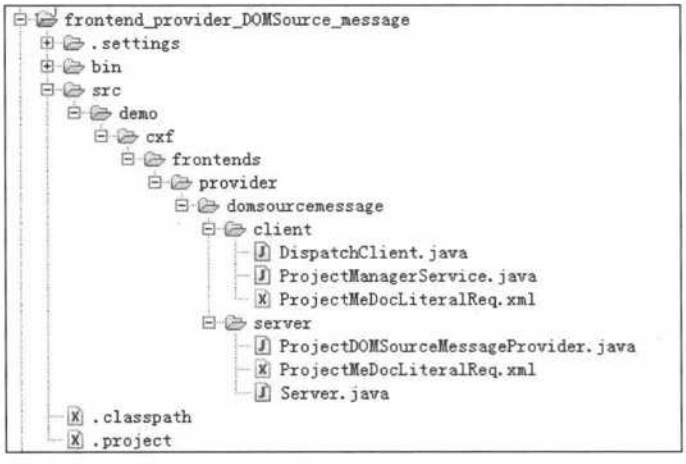


图 6-34　frontend_provider_DOMSource_message 程序结构

包目录包括两个部分，第一个包目录 demo.cxf.frontends.provider.domsourcemessage.server 是服务器程序目录，第二个包目录 demo.cxf.frontends.provider.domsourcemessage.client 是客户端程序目录。注意，在这里并没有服务器端点接口 SEI。

（1）服务器端应用程序

服务器端应用程序主要包括服务器启动组件、Provider<DOMSource>组件和要访问的 XML 文件。

服务器启动组件为 Server.java 文件，代码如下：

```
package demo.cxf.frontends.provider.domsourcemessage.server;
import javax.xml.ws.Endpoint;
public class Server {
    protected Server() throws Exception {
        System.out.println("Starting SoapService");
        Object implementor = new ProjectDOMSourceMessageProvider();
        String address = "http://localhost:9000/ProjectManager";
        Endpoint.publish(address, implementor);
    }

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

```
        new Server();
        System.out.println("Server ready...");

        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

启动服务器，实例化 Provider<DOMSource>对象，然后通过遵循 JAX-WS 规范的 Endpoint 发布 Web Services。Provider<DOMSource>组件只有一个文件——ProjectDOMSourceMessageProvider.java，代码如下：

```
package demo.cxf.frontends.provider.domsourcemessage.server;

import java.io.InputStream;
import javax.xml.soap.MessageFactory;
import javax.xml.soap.SOAPMessage;
import javax.xml.transform.dom.DOMSource;
import javax.xml.ws.Provider;
import javax.xml.ws.Service;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceProvider;

@WebServiceProvider(portName = "ProjectManagerPort", serviceName = "ProjectManagerService",
        targetNamespace = "http://demo.cxf.webservices/projectmanger")
@ServiceMode(value = Service.Mode.MESSAGE)
public class ProjectDOMSourceMessageProvider implements Provider<DOMSource> {

    public ProjectDOMSourceMessageProvider() {}

    public DOMSource invoke(DOMSource request) {
        DOMSource response = new DOMSource();
        try {
            MessageFactory factory = MessageFactory.newInstance();
            SOAPMessage soapReq = factory.createMessage();
            soapReq.getSOAPPart().setContent(request);

            System.out.println("Incoming Client Request as a DOMSource data in MESSAGE
Mode");
            soapReq.writeTo(System.out);
            System.out.println("\n");

            InputStream is = getClass().getResourceAsStream("ProjectMeDocLiteralReq.xml");
            SOAPMessage greetMeResponse =  factory.createMessage(null, is);
            is.close();

            response.setNode(greetMeResponse.getSOAPPart());
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        return response;
    }
}
```

其服务器上的 XML 文件 ProjectMeDocLiteralReq.xml 的内容如下：

```
<?xml version="1.0" encoding="utf-8" ?>

<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <SOAP-ENV:Body>
        <ns4:projectNameResponse
            xmlns:ns4="http://demo.cxf.webservices/projectmanger">
            <ns4:responseType>Hi this is DOMSource message xml file projectName
</ns4:responseType>
        </ns4:projectNameResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

（2）客户端应用程序

客户端应用程序主要由遵循 JAX-WS 规范的 Service 组件、DispatchClient 组件和一个 XML 文件组成。

DispatchClient 组件只有一个文件——DispatchClient.java，代码如下：

```
package demo.cxf.frontends.provider.domsourcemessage.client;

import java.io.File;
import java.io.InputStream;
import java.net.URL;

import javax.xml.namespace.QName;
import javax.xml.soap.MessageFactory;
import javax.xml.soap.SOAPMessage;
import javax.xml.transform.dom.DOMSource;
import javax.xml.ws.Dispatch;
import javax.xml.ws.Service.Mode;

public class DispatchClient {
    private static final String WSDLFile = "http://localhost:9000/ProjectManager?wsdl";
    private static final QName ServiceName = new QName(
            "http://demo.cxf.webservices/projectmanger", "ProjectManagerService");
    private static final QName PortName = new QName(
            "http://demo.cxf.webservices/projectmanger", "ProjectManagerPort");

    public DispatchClient() {}

    public static void main(String[] args) throws Exception {
        URL wsdlURL;
        if (args.length == 0) {
            wsdlURL = new URL(WSDLFile);
        } else {
            File wsdlFile = new File(args[0]);
            if (wsdlFile.exists()) {wsdlURL = wsdlFile.toURL();
            } else {wsdlURL = new URL(args[0]);    }
        }

        MessageFactory factory = MessageFactory.newInstance();

        ProjectManagerService service = new ProjectManagerService(wsdlURL,ServiceName);
        InputStream is1 = DispatchClient.class
                .getResourceAsStream("ProjectMeDocLiteralReq.xml");
```

```
        if (is1 == null) {
            System.err.println("Failed to create input stream from file "
                    + "ProjectMeDocLiteralReq.xml, please check");
            System.exit(-1);
        }

        SOAPMessage soapReq = factory.createMessage(null, is1);
        DOMSource domReqMessage = new DOMSource(soapReq.getSOAPPart());

        Dispatch<DOMSource> dispDOMSrcMessage = service.createDispatch(PortName,
              DOMSource.class, Mode.MESSAGE);

        System.out.println("Invoking  server  through  Dispatch  interface  using
DOMSource in MESSAGE Mode");
       DOMSource domRespMessage = dispDOMSrcMessage.invoke(domReqMessage);
       System.out.println("Response from server: "
                          +
domRespMessage.getNode().getLastChild().getTextContent());
    }
}
```

DispatchClient 组件调用了遵循 JAX-WS 规范的 Service 组件。

Service 组件是一个继承 javax.xml.ws.Service 的类——ProjectManagerService 类，用该类来对服务器进行访问，代码如下：

```
package demo.cxf.frontends.provider.domsourcemessage.client;

import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.WebServiceClient;
import javax.xml.ws.Service;

@WebServiceClient(name = "ProjectManagerService" )
public class ProjectManagerService extends Service {

    public ProjectManagerService(URL wsdlLocation) {
       super(wsdlLocation, null);
    }

    public ProjectManagerService(URL wsdlLocation, QName serviceName) {
       super(wsdlLocation, serviceName);
    }

    public ProjectManagerService() {
       super(null, null);
    }
}
```

XML 文件，要实现与后台服务器中的 XML 文件格式匹配，代码如下：

```
<?xml version="1.0" encoding="utf-8" ?>

<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <SOAP-ENV:Body>
```

```
        <ns4:projectname
            xmlns:ns4="http://demo.cxf.webservices/projectmanger">
            <ns4:requestType>Gil</ns4:requestType>
        </ns4:projectname>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

4. 测试和运行结果

测试编写好的程序代码。首先要启动服务器程序，即在 Eclipse 上运行 Server.java，然后运行客户端程序，客户端程序输出结果如下：

```
Invoking server through Dispatch interface using DOMSource in MESSAGE Mode
Response from server:
            Hi this is DOMSource message xml file projectName
```

服务器端输出结果如下：

```
Incoming Client Request as a DOMSource data in MESSAGE Mode
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
mlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <SOAP-ENV:Header
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"/><SOAP-ENV:Body
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
        <ns4:projectname xmlns:ns4="http://demo.cxf.webservices/projectmanger">
            <ns4:requestType
xmlns:ns4="http://demo.cxf.webservices/projectmanger">Gil</ns4:requestType>
        </ns4:projectname>
    </SOAP-ENV:Body></SOAP-ENV:Envelope>
```

5. 其他内容

本例代码请参见配套资源代码中“第 06 章\frontend_provider_DOMSource_message”文件夹。

6.5.3 采用 DOMSource（Payload）的 Provider/Dispatch 前端模式实现例子

本程序演示 Apache CXF 采用 Provider/Dispatch 前端模式实现客户端和服务器端调用，客户端采用 Dispatch 的 DOMSource（Payload）前端应用模式。后台采用 Provider 方式，主要处理大型的、复杂的 XML 文件。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个存储信息的 XML 文件，通过客户端访问服务器，获得后台的 XML 文件的数据。在这个过程中，都使用 XML 文件实现客户端和服务器端的数据交换。前台客户端采用 JAX-WS 规范的 Service 对象，后台服务端由遵循 JAX-WS 规范的 Endpoint 对象来实现。实现结构如图 6-35 所示。

其中：图中的粗箭头表示 JAX-WS 客户端 Service 通过 Endpoint 获得 Provider，访问后台的 XML 文件，然后返回给前端的 Dispatch 进行处理。

2. 应用程序案例实现步骤

与 DOMSource（message）的 Provider/Dispatch 前端模式实现的流程图完全相同。

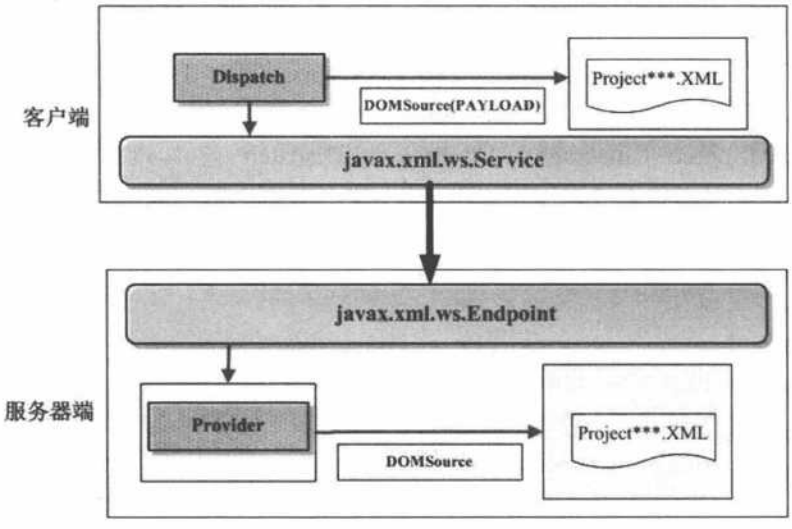


图 6-35　采用 Apache CXF 简化前端实现图

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 6-36 所示。

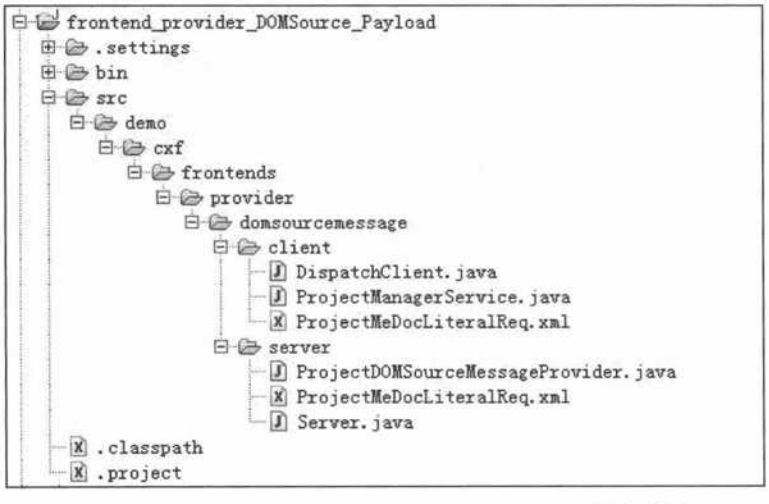


图 6-36　frontend_provider_DOMSource_Payload 程序结构

包目录包括两个部分，第一个包目录 demo.cxf.frontends.provider.domsourcemessage.server 是服务器程序目录，第二个包目录 demo.cxf.frontends.provider.domsourcemessage.client 是客户端程序目录。注意，在这里并没有服务器端点接口 SEI。

（1）服务器端应用程序

服务器端应用程序主要包括服务器启动组件、Provider<DOMSource>组件和要访问的 XML 文件。这三部分与采用 DOMSource（message）的 Provider/Dispatch 前端模式实现完全相同，这里就不再说明。

（2）客户端应用程序

客户端应用程序主要由遵循 JAX-WS 规范的 Service 组件、DispatchClient 组件和一个 XML 文件实现。

与采用 DOMSource（message）的 Provider/Dispatch 前端模式实现相比，不同的是 DispatchClient.java，代码如下（见代码加粗部分）：

```
package demo.cxf.frontends.provider.domsourcemessage.client;

import java.io.File;
import java.io.InputStream;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.soap.MessageFactory;
import javax.xml.soap.SOAPMessage;
import javax.xml.transform.dom.DOMSource;
import javax.xml.ws.Dispatch;
import javax.xml.ws.Service.Mode;

public class DispatchClient {

    private static final String WSDLFile = "http://localhost:9000/ProjectManager?wsdl";
    ……

    public DispatchClient() {}

    public static void main(String[] args) throws Exception {
        URL wsdlURL;
        ……

        Dispatch<DOMSource> dispDOMSrcMessage = service.createDispatch(PortName,
              DOMSource.class, Mode.PAYLOAD);

        ……
    }
}
```

4. 测试和运行结果

测试编写好的程序代码，其客户端程序和服务器端程序输出结果与采用 DOMSource（message）的 Provider/Dispatch 前端模式实现完全相同，这里就不再说明。

5. 其他内容

本例代码请参见配套资源代码中“第 06 章\frontend_provider_DOMSource_Payload”文件夹。

6.5.4 采用 SOAPMessage 的 Provider/Dispatch 前端模式实现例子

本程序演示 Apache CXF 采用 Provider/Dispatch 前端模式实现客户端和服务器端调用，客户端采用 Dispatch 的 SOAPMessage 前端应用模式。后台采用 Provider 方式，主要处理大型的、复杂的 XML 文件。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个存储信息的 XML 文件，通过客户端访问服务器，获得后台的 XML

文件的数据。在这个过程中，都使用 XML 文件实现客户端和服务器端的数据交换。前台客户端采用 JAX-WS 规范的 Service 对象，后台服务器端为遵循 JAX-WS 规范的 Endpoint 对象来实现。实现结构如图 6-37 所示。

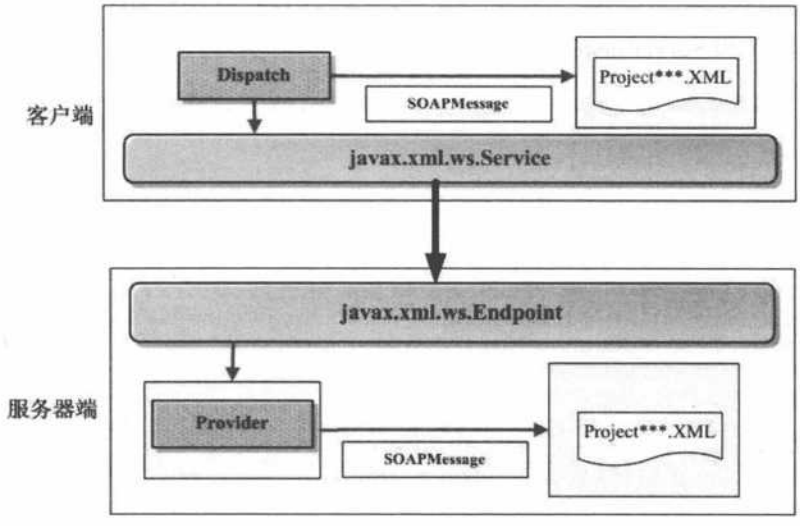


图 6-37　采用 Apache CXF 简化前端实现图

其中：图中的粗箭头表示 JAX-WS 客户端 Service 通过 Endpoint 获得 Provider，访问后台的 XML 文件，然后返回给前端的 Dispatch 进行处理。

2. 应用程序案例实现步骤

本流程与 DOMSource（message）的 Provider/Dispatch 前端模式实现的流程完全相同。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 6-38 所示。

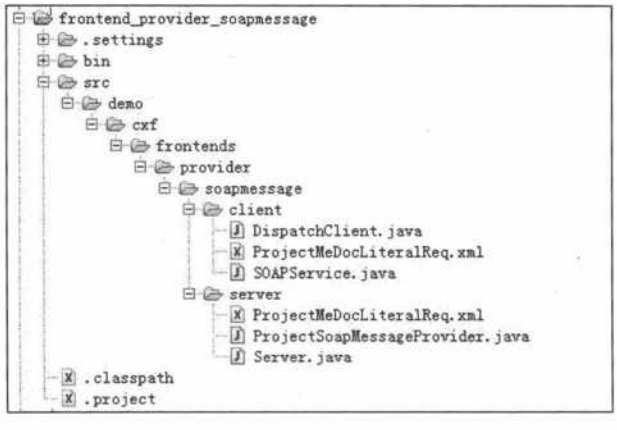


图 6-38　frontend_provider_soapmessage 程序结构

包目录包括两个部分，第一个包目录 demo.cxf.frontends.provider.soapmessage.server 是服务器程序目录，第二个包目录 demo.cxf.frontends.provider.soapmessage.client 是客户端程序目录。注意，

在这里并没有服务器端点接口 SEI。

（1）服务器端应用程序

服务器端应用程序主要包括服务器启动组件、Provider<SOAPMessage>组件和要访问的 XML 文件。

服务器启动组件为 Server.java 文件，代码如下：

```
package demo.cxf.frontends.provider.soapmessage.server;

import javax.xml.ws.Endpoint;

public class Server {
    protected Server() throws Exception {
        System.out.println("Starting SoapService");
        Object implementor = new ProjectSoapMessageProvider();
        String address = "http://localhost:9000/SoapContext/SoapPort";
        Endpoint.publish(address, implementor);
    }

    public static void main(String args[]) throws Exception {
        new Server();
        System.out.println("Server ready...");

        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

启动服务器，实例化 Provider<SOAPMessage>对象，然后通过遵循 JAX-WS 规范的 Endpoint 发布 Web Services。

Provider<SOAPMessage>组件只有一个文件，即 ProjectSoapMessageProvider.java，代码如下：

```
package demo.cxf.frontends.provider.soapmessage.server;

import java.io.InputStream;

import javax.xml.soap.MessageFactory;
import javax.xml.soap.SOAPMessage;
import javax.xml.ws.Provider;
import javax.xml.ws.Service;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceProvider;


@WebServiceProvider(portName = "SoapPort", serviceName = "SOAPService",
        targetNamespace = "http://demo.cxf.org/projectmanger_soap_http")
        //wsdlLocation = "wsdl/projectmanger.wsdl")
@ServiceMode(value = Service.Mode.MESSAGE)
public class ProjectSoapMessageProvider implements Provider<SOAPMessage> {

    public ProjectSoapMessageProvider() {
        // TODO Auto-generated constructor stub
    }

    public SOAPMessage invoke(SOAPMessage request) {
```

```
        SOAPMessage response = null;
        try {
            System.out.println("Incoming Client Request as a SOAPMessage");
            MessageFactory factory = MessageFactory.newInstance();
            InputStream is = getClass().getResourceAsStream(
                    "ProjectMeDocLiteralReq.xml");
            response = factory.createMessage(null, is);
            is.close();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        return response;
    }
}
```

服务器上的 XML 文件 ProjectMeDocLiteralReq.xml 内容如下：

```
<?xml version="1.0" encoding="utf-8" ?>

<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <SOAP-ENV:Body>
        <ns4:projectNameResponse
            xmlns:ns4="http://demo.cxf.org/projectmanger_soap_http">
            <ns4:responseType>Hi this is xml file projectName</ns4:responseType>
        </ns4:projectNameResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

（2）客户端应用程序

客户端应用程序主要由遵循 JAX-WS 规范的 Service 组件、DispatchClient 组件和一个 XML 文件组成。

DispatchClient 组件只有一个文件——DispatchClient.java，代码如下：

```
package demo.cxf.frontends.provider.soapmessage.client;

import java.io.File;
import java.io.InputStream;
import java.net.URL;

import javax.xml.namespace.QName;
import javax.xml.soap.MessageFactory;
import javax.xml.soap.SOAPMessage;
import javax.xml.ws.Dispatch;
import javax.xml.ws.Service.Mode;

public class DispatchClient {
    public static final String WSDLFile = "http://localhost:9000/SoapContext/
SoapPort?wsdl";

    public DispatchClient() {}

    public static void main(String[] args) throws Exception {
        URL wsdlURL;
```

```
        if (args.length == 0) {wsdlURL = new URL(WSDLFile);
        } else {
            File wsdlFile = new File(args[0]);
            if (wsdlFile.exists()) {
                wsdlURL = wsdlFile.toURL();
            } else {
                wsdlURL = new URL(args[0]);
            }
        }

        MessageFactory factory = MessageFactory.newInstance();
        QName serviceName = new QName("http://demo.cxf.org/projectmanger_soap_http",
"SOAPService");
        QName portName = new QName("http://demo.cxf.org/projectmanger_soap_http",
"SoapPort");

        SOAPService service = new SOAPService(wsdlURL, serviceName);
        InputStream is1 = DispatchClient.class.getResourceAsStream
("ProjectMeDocLiteralReq.xml");
        if (is1 == null) {
            System.err.println("Failed to create input stream from file "
                    + "GreetMeDocLiteralReq1.xml, please check");
            System.exit(-1);
        }
        SOAPMessage soapReq1 = factory.createMessage(null, is1);

        Dispatch<SOAPMessage> dispSOAPMsg = service.createDispatch(portName,
                SOAPMessage.class, Mode.MESSAGE);

        System.out.println("Invoking server through Dispatch interface using SOAPMessage");
        SOAPMessage soapResp = dispSOAPMsg.invoke(soapReq1);
        System.out.println("Response from server: "
                + soapResp.getSOAPBody().getTextContent());
    }
}
```

DispatchClient 组件调用遵循 JAX-WS 规范的 Service 组件。

Service 组件是一个继承 javax.xml.ws.Service 的类——SOAPService 类，用该类来对服务器进行访问，代码如下：

```
package demo.cxf.frontends.provider.soapmessage.client;

import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.WebServiceClient;
import javax.xml.ws.Service;

@WebServiceClient(name = "SOAPService" )
public class SOAPService extends Service {
    public SOAPService(URL wsdlLocation) {
        super(wsdlLocation, null);
    }

    public SOAPService(URL wsdlLocation, QName serviceName) {
        super(wsdlLocation, serviceName);
```

```
    }

    public SOAPService() {
        super(null, null);
    }
}
```

最后是 XML 文件，主要说明该 XML 必须与后台服务器中的 XML 文件格式要匹配，代码如下：

```
<?xml version="1.0" encoding="utf-8" ?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <SOAP-ENV:Body>
        <ns4:projectname
            xmlns:ns4="http://demo.cxf.org/projectmanger_soap_http">
            <ns4:requestType>Gil</ns4:requestType>
        </ns4:projectname>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

4. 测试和运行结果

测试编写好的程序代码，首先要启动服务器程序，即在 Eclipse 上运行 Server.java，然后运行客户端程序，其客户端程序输出结果如下：

```
Invoking server through Dispatch interface using SOAPMessage
Response from server:
            Hi this is xml file projectName
```

其服务器端输出结果如下：

```
Incoming Client Request as a SOAPMessage
```

5. 其他内容

本例代码请参见配套资源代码中“第 06 章\frontend_provider_soapmessage”文件夹。

6.6　采用 Apache CXF 的动态客户端技术

Web Services 客户端通常使用服务接口来调用服务方法。直至现在，客户端应用程序都使用服务端点接口（SEI）或服务端点接口代理接口来调用服务方法。在有些情况下，需要提供一个在运行时动态生成的客户端。当开发人员不希望创造和维护一个客户端需要额外的开销时，动态客户端通常是一种比较适合的解决办法。由于动态客户端动态地检查 WSDL，并在 WSDL 定义基础上动态地创建客户端输入和输出对象，它也可以验证 WSDL 文件和输入/输出消息格式，但实际上动态客户端并没有调用 Web 服务作为单元测试环境的一部分。

6.6.1　Apache CXF 的动态客户端技术介绍

Apache CXF 提供一个替换方式来创建 Web Services 动态客户端。它通过使用

JaxWsDynamicClientFactory 工厂类来阐明动态的概念。 Apache CXF 还提供了一个不支持 JAX-WS 的类 DynamicClientFactory。如果服务组件不使用开发 JAX-WS 的任意 API 方法，这种隐式调用是非常有用的。本节的重点是 JAX-WS 版本的动态客户端工厂类。

1. 创建一个简单的动态客户端程序

如何创建一个动态的客户端？开发者需要一个现有的 WSDL 文件来生成一个动态的客户端。该 JaxWsDynamicClientFactory 工厂类会在运行时和内存中利用这个 WSDL 文档并创建和维护 SEI 和数据类。作为说明这项工作的一部分，我们用 Order 流程实现 WSDL 来阐述，代码如下：

```
package demo.order.client;

import org.apache.cxf.jaxws.endpoint.dynamic.JaxWsDynamicClientFactory;
import org.apache.cxf.endpoint.Client;
import java.lang.reflect.Method;

public class OrderProcessJaxWsDynamicClient {
    public OrderProcessJaxWsDynamicClient() {
    }

    public static void main(String str[]) throws Exception {
        JaxWsDynamicClientFactory dcf = JaxWsDynamicClientFactory.newInstance();
        Client client = dcf
                .createClient("http://localhost:8080/OrderProcess?wsdl");
        Object order = Thread.currentThread().getContextClassLoader()
                .loadClass("demo.order.Order").newInstance();
        Method m1 = order.getClass().getMethod("setCustomerID", String.class);
        Method m2 = order.getClass().getMethod("setItemID", String.class);
        Method m3 = order.getClass().getMethod("setQty", Integer.class);
        Method m4 = order.getClass().getMethod("setPrice", Double.class);
        m1.invoke(order, "C001");
        m2.invoke(order, "I001");
        m3.invoke(order, 100);
        m4.invoke(order, 200.00);
        Object[] response = client.invoke("processOrder", order);
        System.out.println("Response is " + response[0]);
    }
}
```

该 OrderProcessJaxWsDynamicClient 执行以下内容：

（1）创建一个 JaxWsDynamicClientFactory 工厂类，本实例是通过调用静态的 newInstance 方法来实现的。

（2）在得到该实例后，开发人员调用这个实例的 createClient 方法来创建动态的客户端组件。此方法检索从 URL http://localhost:8080/OrderProcess?wsdl WDSL 文件。请注意，该 URL 是发布在 http://localhost:8080/OrderProcess 的 OrderProcess 服务。此方法返回一个 org.apache.cxf.endpoint.ClientImpl 类的对象。

（3）使用上下文类加载器类，动态地生成一个 Order 实例。还需要调用订购 processOrder 方法。

（4）下一组代码填充订购 Bean。在这个过程中使用 Java 反射。由于这是一个动态生成客户端，也没有 SEI 可以直接提供给客户，开发人员必须使用 Java 反射 API 来调用服务和数据的方法。在客户端的调用对象的方法将调用 processOrder 服务方法，该方法包含一个参数 Bean。

（5）检索和打印输出在控制台上。我们的目标是表示为一个简单的字符串，其中包含订单 ID。

2. 用 Apache CXF 的服务模式建立动态客户端

上述代码采用了 Java 的反射机制。Apache CXF 也可以采用其他方式创建动态客户端，例如，采用 Apache CXF Service model 创建动态的 Web Services 客户端。

Apache CXF Service model 的结构如图 6-39 所示。

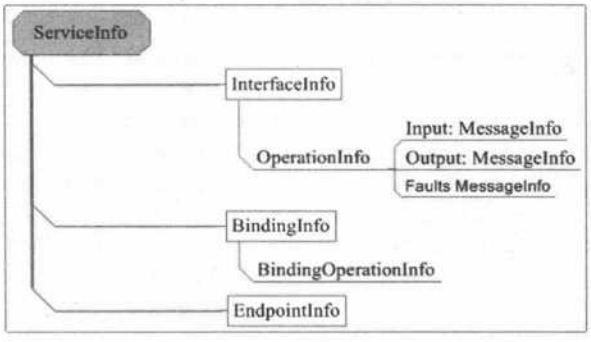


图 6-39　Apache CXF Service model 的结构图

可以把前面用反射机制形成的代码改成使用 Service model 的实现形式，代码如下：

```
import java.beans.PropertyDescriptor;
import java.util.List;
import javax.xml.namespace.QName;
import org.apache.cxf.endpoint.Client;
import org.apache.cxf.endpoint.Endpoint;
import org.apache.cxf.jaxws.endpoint.dynamic.JaxWsDynamicClientFactory;
import org.apache.cxf.service.model.BindingInfo;
import org.apache.cxf.service.model.BindingMessageInfo;
import org.apache.cxf.service.model.BindingOperationInfo;
import org.apache.cxf.service.model.MessagePartInfo;
import org.apache.cxf.service.model.ServiceInfo;

public class OrderProcessJaxWsDynClient {

    public OrderProcessJaxWsDynClient() {
    }

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

        JaxWsDynamicClientFactory dcf = JaxWsDynamicClientFactory.newInstance();
        Client client = dcf
                .createClient("http://localhost:8080/OrderProcess?wsdl");
        Endpoint endpoint = client.getEndpoint();//
        // Make use of CXF service model to introspect the existing WSDL
        ServiceInfo serviceInfo = endpoint.getService().getServiceInfos()
                .get(0);
        QName bindingName = new QName("http://order.demo/",
                "OrderProcessServiceSoapBinding");
```

```
        BindingInfo binding = serviceInfo.getBinding(bindingName);
        QName opName = new QName("http://order.demo/", "processOrder");
        BindingOperationInfo boi = binding.getOperation(opName);
        // Operation name is processOrder
        BindingMessageInfo inputMessageInfo = null;
        if (!boi.isUnwrapped()) {
            // OrderProcess uses document literal wrapped style.
            inputMessageInfo = boi.getWrappedOperation().getInput();
        } else {
            inputMessageInfo = boi.getUnwrappedOperation().getInput();
        }
        List<MessagePartInfo> parts = inputMessageInfo.getMessageParts();
        MessagePartInfo partInfo = parts.get(0); // Input class isOrder
        // List<MessagePartInfo> parts = inputMessageInfo.getMessageParts();
        // MessagePartInfo partInfo = parts.get(0); // Input class is Order
        // Get the input class Order
        Class<?> orderClass = partInfo.getTypeClass();
        Object orderObject = orderClass.newInstance();
        // Populate the Order bean
        // Set customer ID, item ID, price and quantity
        PropertyDescriptor custProperty = new PropertyDescriptor("customerID",
                orderClass);
        custProperty.getWriteMethod().invoke(orderObject, "C001");
        PropertyDescriptor itemProperty = new PropertyDescriptor("itemID",
                orderClass);
        itemProperty.getWriteMethod().invoke(orderObject, "I001");
        PropertyDescriptor priceProperty = new PropertyDescriptor("price",
                orderClass);
        priceProperty.getWriteMethod().invoke(orderObject,
                Double.valueOf(100.00));
        PropertyDescriptor qtyProperty = new PropertyDescriptor("qty",
                orderClass);
        qtyProperty.getWriteMethod().invoke(orderObject, Integer.valueOf(20));
        // Invoke the processOrder() method and print the result
        // The response class is String
        Object[] result = client.invoke(opName, orderObject);
        System.out.println("The order ID is " + result[0]);
    }
}
```

正如读者所看到的，Order 对象作为输入参数调用 processOrder 方法，PropertyDescriptor 用于调用 Order 类的 set 和 get 方法。尤其重要的是所有的调用都是动态的，没有一行硬编码，包括参数的传入和结果的输出。基于服务模式的动态客户端可用于对 Web Services 有许多复杂依赖关系的应用程序。

6.6.2 Apache CXF 的动态客户端技术例子

本程序演示 Apache CXF 采用动态客户端来实现客户端和服务器端调用，客户端采用 Apache CXF 的代码优先（Java First）前端应用模式。采用 POJO 对象来作为中间传输对象。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，都使用 POJO 对象实现客户端和服务器端的数据交换。

后台服务端采用两种方式，分别是：① 遵循 JAX-WS 规范的 Endpoint 实现；② 采用 Apache CXF 本身的 JaxWsServerFactoryBean 服务端来实现。

前端访问采用两种方式，分别是：① 采用 Apache CXF 本身的 DynamicClientFactory 客户端来实现；② 采用 Apache CXF 本身的 JaxWsDynamicClientFactory 客户端来实现。

这样的组合方式有四种，分别为：① 服务器端为遵循 JAX-WS 规范的 Endpoint 来实现，客户端为用 Apache CXF 本身的 DynamicClientFactory 客户端来实现；② 服务器端为遵循 JAX-WS 规范的 Endpoint 来实现，客户端为 Apache CXF 本身的 JaxWsDynamicClientFactory 客户端来实现；③ 服务器端为 Apache CXF 本身的 JaxWsServerFactoryBean 服务端来实现，客户端为用 Apache CXF 本身的 DynamicClientFactory 客户端来实现；④ 服务器端为 Apache CXF 本身的 JaxWsServerFactoryBean 服务端来实现，客户端为 Apache CXF 本身的 JaxWsDynamicClientFactory 客户端来实现。

第一种方式的实现结构如图 6-40 所示。

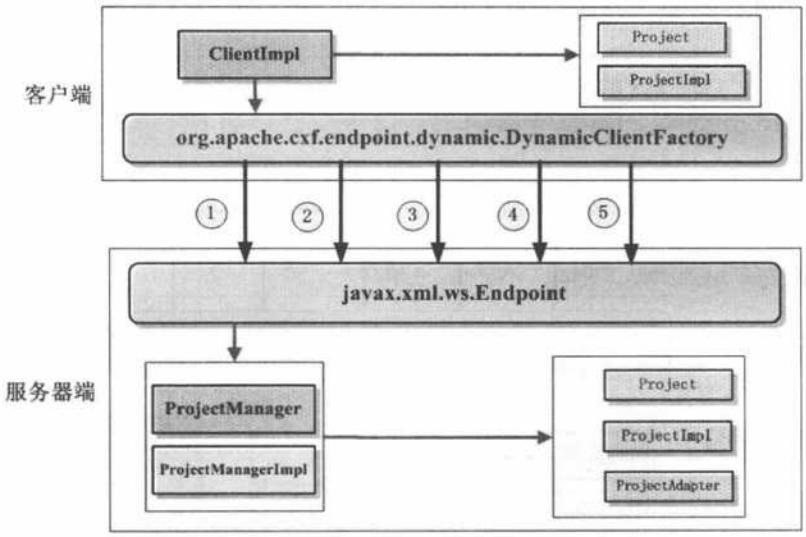


图 6-40　基于 DynamicClientFactory 前端，JAX-WS 服务端实现动态客户端应用模式

其中：① 表示 DynamicClientFactory 客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。客户端和服务端的 SEI 都是 ProjectManager 接口。

第二种方式的实现结构如图 6-41 所示。

其中：① 表示 JaxWsDynamicClientFactory 客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表

示客户端通过 SEI 删除一个项目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。在客户端和服务端的 SEI 都是 ProjectManager 接口。

第三种方式的实现结构如图 6-42 所示。

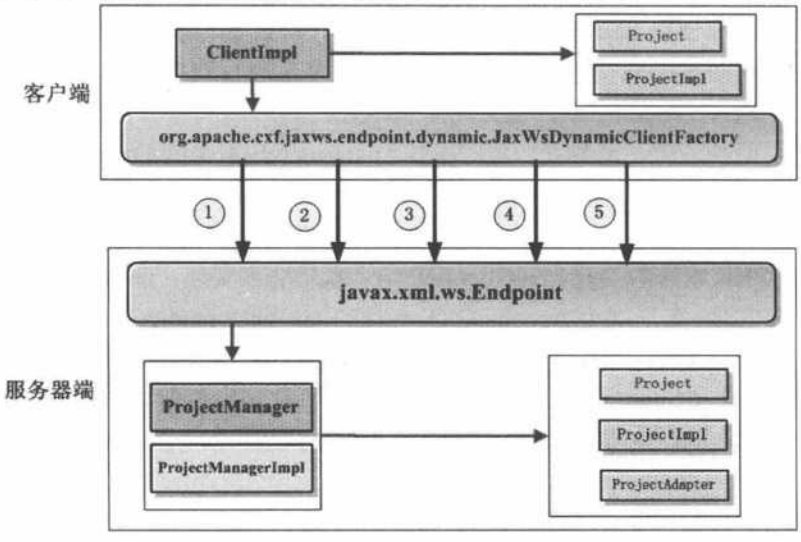


图 6-41　基于 JaxWsDynamicClientFactory 前端，JAX-WS 服务端实现动态客户端应用模式

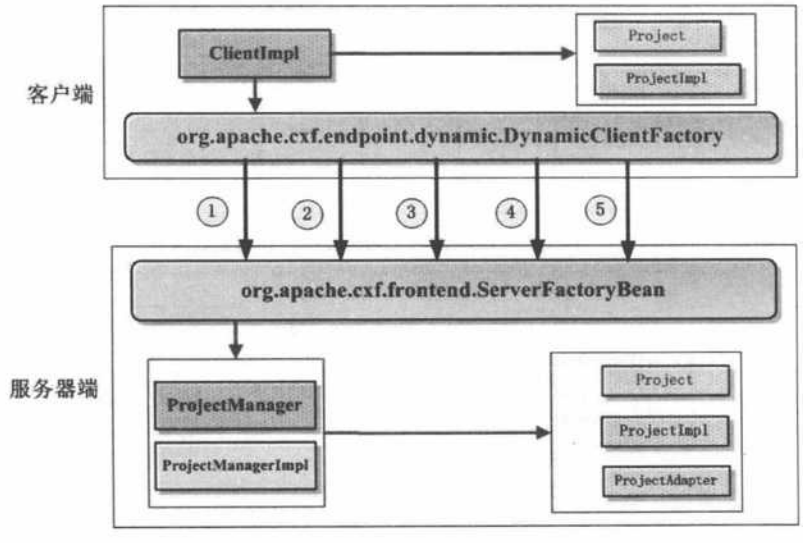


图 6-42　基于 DynamicClientFactory 前端，JaxWsServerFactoryBean 服务端实现动态客户端应用模式

其中：① 表示 DynamicClientFactory 客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。在客户端和服务端的 SEI 都是 ProjectManager 接口。

第四种方式的实现结构如图 6-43 所示。

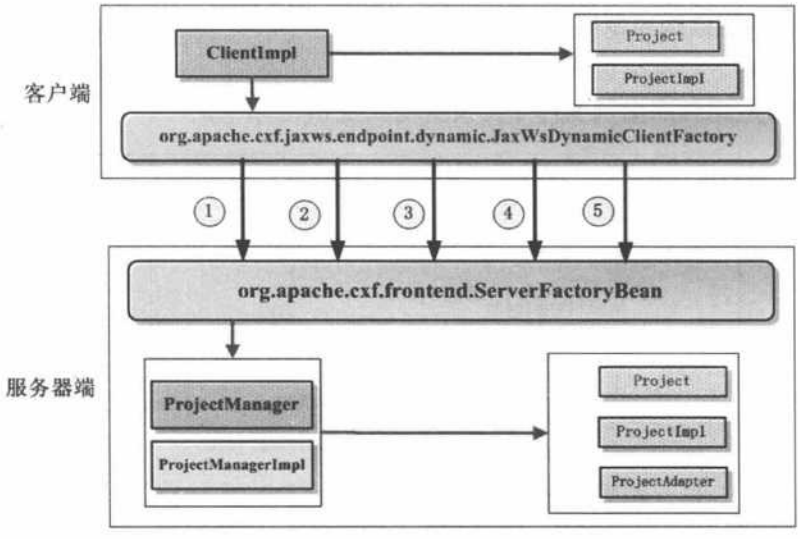


图 6-43　基于 JaxWsDynamicClientFactory 前端，JaxWsServerFactoryBean 服务端实现动态客户端应用模式

其中：① 表示 JaxWsDynamicClientFactory 客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。在客户端和服务端的 SEI 都是 ProjectManager 接口。

2. 应用程序案例实现步骤

要创建动态客户端实现 Web Services 程序的流程如图 6-44 所示。

步骤说明如下：

（1）创建一个 Java 项目。

（2）获得 WSDL 文件。

（3）由 WSDL 文件形成服务端点接口（SEI）及其相关对象和注释。

（4）编写发布这个 Java 程序为 Web Services 的服务端程序。

（5）编写 Web Services 的客户端应用程序。

（6）启动服务器程序，对外发布 Web Services。

（7）客户端调用 Web Services。

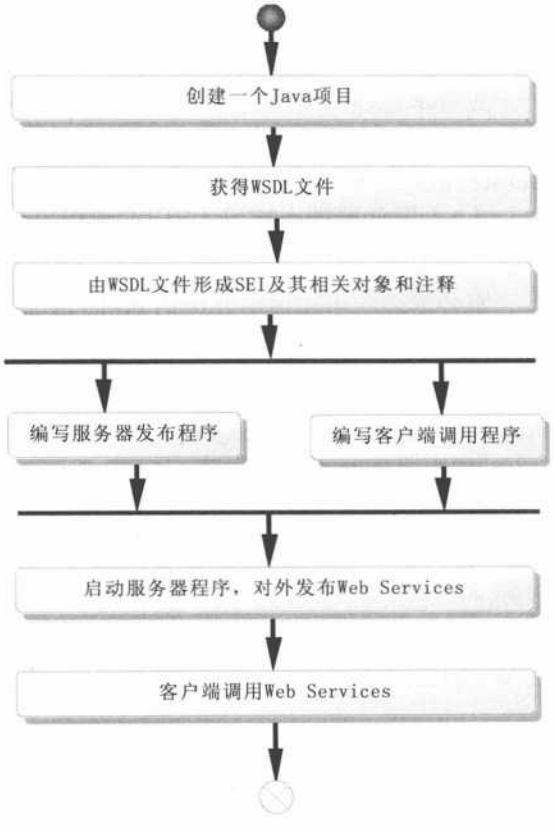


图 6-44　应用程序案例实现步骤流程图

3. 应用程序案例结构和源码说明

前面介绍的四种方式都可以实现上述功能。为了方便介绍，这里只说明服务器端应用程序和客户端应用程序。应用程序案例的程序目录结构如图 6-45 所示。

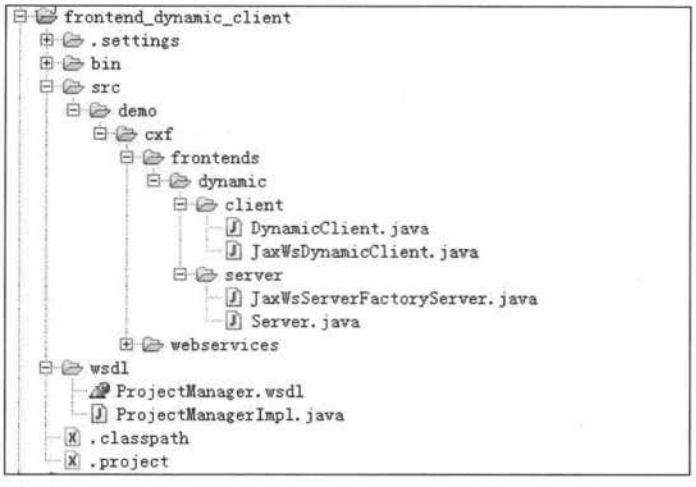


图 6-45　frontend_dynamic_client 程序结构

包目录包括三个部分，第一个包目录 demo.cxf.frontends.dynamic.server 是服务器程序目录，第二个包目录 demo.cxf.frontends.dynamic.client 是客户端程序目录，第三个包目录 demo.cxf.webservices 是 SEI 和相关对象等公用部分，客户端和服务器端都可能引用其内容。

（1）服务器端点接口（SEI）和相关对象

首先要介绍的是服务器端点接口（SEI）。

服务器端应用程序主要包括 ProjectManager 组件和服务器启动类。

创建服务器端点接口（SEI）和组件，Project 组件包括 WebService 包下的十几个 Project 文件，主要是 ObjectFactory.java、Project.java、ProjectManager.java、ProjectManagerImpl.java、ProjectManagerImplService.java 等。其他的类文件都是由 ProjectManager 接口上方法映射出来的。

（2）服务器端发布程序

创建了 SEI，要把这个 SEI 发布出去，还需要一个部署程序，即服务器端的发布程序，本案例采用 JAX-WS 规范来编写服务器端程序。创建一个 Server 类，启动这个服务，然后发布 SEI 服务。

服务器启动类有基于 JAX-WS 规范的 Server.java 文件和基于 Apache CXF 本身实现的 JaxWsServerFactoryServer.java 文件。

```
package demo.cxf.frontends.dynamic.server;

import javax.xml.ws.Endpoint;
import demo.cxf.webservices.*;

public class Server {

    public Server() {
        System.out.println("Starting Server");
        ProjectManager projectManager = new ProjectManagerImpl();
```

```
        String address = "http://localhost:9000/ProjectManager";
        Endpoint.publish(address, projectManager);
    }

    public static void main(String[] args) throws Exception {
        new Server();
        System.out.println("Server ready...");

        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

在本案例中，服务器启动的服务地址是 http://localhost:9000/ProjectManager。服务器引入了 SEI 接口——ProjectManager 接口。这样，ProjectManager 的 Web Services 就暴露到了外部。可以在浏览器上输入 http://localhost:9000/ProjectManager 来查看这个服务的内容。

还有一种方式可以发布服务，即采用 Apache CXF 本身的服务发布类来实现，代码如下：

```
package demo.cxf.frontends.dynamic.server;

import org.apache.cxf.jaxws.JaxWsServerFactoryBean;
import demo.cxf.webservices.*;

public class JaxWsServerFactoryServer {

    protected JaxWsServerFactoryServer() throws Exception{
        System.out.println("Starting Server");
        ProjectManagerImpl projectManager = new ProjectManagerImpl();
       JaxWsServerFactoryBean svrFactory = new JaxWsServerFactoryBean();
       svrFactory.setServiceClass(ProjectManagerImpl.class);
       svrFactory.setAddress("http://localhost:9000/ProjectManager");
       svrFactory.setServiceBean(projectManager);
       svrFactory.create();
    }

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

        new JaxWsServerFactoryServer();
       System.out.println("JaxWsServerFactoryBean Server ready...");

       Thread.sleep(5 * 60 * 1000);
       System.out.println("JaxWsServerFactoryBean Server exiting");
       System.exit(0);
    }
}
```

（3）客户端应用程序

编写 Web Services 的客户端应用程序。有两种实现方式，第一种是采用 Apache CXF 本身的 DynamicClientFactory 客户端来实现，第二种是采用 Apache CXF 本身的 JaxWsDynamicClientFactory 客户端来实现。

采用 Apache CXF 本身的 DynamicClientFactory 客户端程序代码如下：

```
package demo.cxf.frontends.dynamic.client;
```

```
import java.lang.reflect.Method;
import java.net.URL;

import org.apache.cxf.endpoint.Client;
import org.apache.cxf.endpoint.ClientImpl;
import org.apache.cxf.endpoint.dynamic.DynamicClientFactory;

import demo.cxf.webservices.Project;

public class DynamicClient {

    public DynamicClient() {}

    public static void main(String[] args) throws Exception{
        URL wsdlURL = new URL("http://localhost:9000/ProjectManager?wsdl");
        DynamicClientFactory dcf = DynamicClientFactory.newInstance();
        //Client client = dcf.createClient("http://localhost:9000/ProjectManager?
wsdl", SERVICE_NAME);
        Client client = dcf.createClient(wsdlURL);

        ClientImpl projectManager = (ClientImpl) client;


     //项目的基本情况
        getGreet(projectManager);
        greetingWithParameter(projectManager);
        getProjectName(projectManager);
        getDefaultProject(projectManager);
        getAllProject(projectManager);

        //修改默认项目，查看项目的基本情况
        setDefaultProject(projectManager);

        //增加一个项目，查看项目的基本情况
        registerProject(projectManager);

        //删除一个项目，查看项目的基本情况
        removeProject(projectManager);

        System.exit(0);
    }

    //调用 SEI 的 getGreet 方法
    private static void getGreet(ClientImpl projectManager)throws Exception {
        Object[] response = projectManager.invoke("greeting");
        System.out.println("Response is " + response[0]);
    }

    //调用 SEI 的 greetingWithParameter 方法
    private static void greetingWithParameter(ClientImpl projectManager)throws
Exception {
        Object[] response = projectManager.invoke("greetingWithParameter","greet");
        System.out.println("Response is " + response[0]);
    }

    //调用 SEI 的 getProjectName 方法
    private static void getProjectName(ClientImpl projectManager)throws Exception {
        Object[] response = projectManager.invoke("getProjectName");
```

```
        System.out.println("default project is " + response[0]);
    }

    //调用 SEI 的 getDefaultProject 方法
    private static void getDefaultProject(ClientImpl projectManager)throws Exception {
        Object[] response = projectManager.invoke("getDefaultProject");
        Project defaultProject = (Project)response[0];
        System.out.println("defaultProjectID is " + defaultProject.getProjectID()
                + " . " + "defaultProjectName is " + defaultProject.getProjectName());
    }

    //调用 SEI 的 setDefaultProject 方法
    private static void setDefaultProject(ClientImpl projectManager)throws Exception {
        Object project =
Thread.currentThread().getContextClassLoader().loadClass("demo.cxf.webservices.
Project").newInstance();
        Method m1 = project.getClass().getMethod("setProjectID", String.class);
        Method m2 = project.getClass().getMethod("setProjectName", String.class);

        m1.invoke(project, "this is Dynamic projectID");
        m2.invoke(project, "this is Dynamic projectName");
        projectManager.invoke("setDefaultProject", project);

        getDefaultProject(projectManager);
        projectManager.invoke("getAllProject");
    }

    //调用 SEI 的 registerProject 方法
    private static void registerProject(ClientImpl projectManager)throws Exception {

        Object project =
 Thread.currentThread().getContextClassLoader().loadClass("demo.cxf.webservices.
Project").newInstance();
        Method m1 = project.getClass().getMethod("setProjectID", String.class);
        Method m2 = project.getClass().getMethod("setProjectName", String.class);

        m1.invoke(project, "New ProjectID");
        m2.invoke(project, "New ProjectName");
        projectManager.invoke("registerProject",project);

        System.out.println(projectManager.invoke("getProjectNameByID","New ProjectID"));
        projectManager.invoke("getAllProject");
    }

    //调用 SEI 的 removeProject 方法
    private static void removeProject(ClientImpl projectManager)throws Exception {
        projectManager.invoke("removeProject","New ProjectID");
        projectManager.invoke("getAllProject");
    }

    //调用 SEI 的 getAllProject 方法
    private static void getAllProject(ClientImpl projectManager) throws Exception{
        projectManager.invoke("getAllProject");
    }

}
```

本例首先创建一个实现 javax.xml.ws.Service 接口的服务，接着通过 addPort 方法来传入服务

的端口类、绑定方式和 Web Services 的地址。最后通过 getPort 方法来获得服务器的 SEI 接口。这里需要 Web Services 服务接口 ProjectManager，后面的工作基于这个 ProjectManager 的 SEI 来实现。

上述的所有调用都基于 Apache CXF 的 DynamicClientFactory 规范。下面采用 Apache CXF 的前端编程模式来实现调用服务。

采用 Apache CXF 本身的 JaxWsDynamicClientFactory 客户端程序代码如下：

```
package demo.cxf.frontends.dynamic.client;

import java.lang.reflect.Method;
import java.net.URL;
import javax.xml.namespace.QName;

import org.apache.cxf.jaxws.endpoint.dynamic.JaxWsDynamicClientFactory;
import org.apache.cxf.endpoint.Client;
import org.apache.cxf.endpoint.ClientImpl;

import demo.cxf.webservices.Project;

public class JaxWsDynamicClient {

    private static final QName SERVICE_NAME   =
new QName("http://webservices.cxf.demo/", "ProjectManagerImplService");

    public static void main(String[] args) throws Exception {
        URL wsdlURL;
        wsdlURL = new URL("http://localhost:9000/ProjectManager?wsdl");

        JaxWsDynamicClientFactory factory = JaxWsDynamicClientFactory.newInstance();

        Client client = factory.createClient(wsdlURL,SERVICE_NAME);
        ClientImpl projectManager = (ClientImpl) client;

        //项目的基本情况
        getGreet(projectManager);
        greetingWithParameter(projectManager);
        getProjectName(projectManager);
        getDefaultProject(projectManager);
        getAllProject(projectManager);

        //修改默认项目，查看项目的基本情况
        setDefaultProject(projectManager);

        //增加一个项目，查看项目的基本情况
        registerProject(projectManager);

        //删除一个项目，查看项目的基本情况
        removeProject(projectManager);

        System.exit(0);
    }

    //调用 SEI 的 getGreet 方法
    private static void getGreet(ClientImpl projectManager)throws Exception {
        Object[] response = projectManager.invoke("greeting");
```

```
        System.out.println("Response is " + response[0]);
    }

    //调用 SEI 的 greetingWithParameter 方法
    private static void greetingWithParameter(ClientImpl projectManager)throws Exception {
        Object[] response = projectManager.invoke("greetingWithParameter","greet");
        System.out.println("Response is " + response[0]);
    }

    //调用 SEI 的 getProjectName 方法
    private static void getProjectName(ClientImpl projectManager)throws Exception {
        Object[] response = projectManager.invoke("getProjectName");
        System.out.println("default project is " + response[0]);
    }

    //调用 SEI 的 getDefaultProject 方法
    private static void getDefaultProject(ClientImpl projectManager)throws Exception {
        Object[] response = projectManager.invoke("getDefaultProject");
        Project defaultProject = (Project)response[0];
        System.out.println("defaultProjectID is " + defaultProject.getProjectID()
                + " . " + "defaultProjectName is " + defaultProject.getProjectName());
    }

    //调用 SEI 的 setDefaultProject 方法
    private static void setDefaultProject(ClientImpl projectManager)throws Exception
{

        Object project = Thread.currentThread().getContextClassLoader().loadClass
("demo.cxf.webservices. Project").newInstance();
        Method m1 = project.getClass().getMethod("setProjectID", String.class);
        Method m2 = project.getClass().getMethod("setProjectName", String.class);
        m1.invoke(project, "this is Dynamic projectID");
        m2.invoke(project, "this is Dynamic projectName");
        projectManager.invoke("setDefaultProject", project);

        getDefaultProject(projectManager);
        projectManager.invoke("getAllProject");
    }

    //调用 SEI 的 registerProject 方法
    private static void registerProject(ClientImpl projectManager)throws Exception {

        Object project =
 Thread.currentThread().getContextClassLoader().loadClass("demo.cxf.webservices.
Project").newInstance();
        Method m1 = project.getClass().getMethod("setProjectID", String.class);
        Method m2 = project.getClass().getMethod("setProjectName", String.class);
        m1.invoke(project, "New ProjectID");
        m2.invoke(project, "New ProjectName");
        projectManager.invoke("registerProject",project);

        System.out.println(projectManager.invoke("getProjectNameByID","New
ProjectID"));
        projectManager.invoke("getAllProject");
    }

    //调用 SEI 的 removeProject 方法
    private static void removeProject(ClientImpl projectManager)throws Exception {
```

```
        projectManager.invoke("removeProject","New ProjectID");
        projectManager.invoke("getAllProject");
    }

    //调用 SEI 的 getAllProject 方法
    private static void getAllProject(ClientImpl projectManager) throws Exception{
        projectManager.invoke("getAllProject");
    }
}
```

客户端首先实例化一个 JaxWsProxyFactoryBean 对象。设置 JaxWs 代理工厂的属性，包括 SEI 和地址，然后调用 SEI 的方法。

4. 测试和运行结果

测试编写好的程序代码，首先要启动服务器程序，即在 Eclipse 上运行 Server.java，然后在 Eclipse 上运行客户端程序，两种方式的客户端程序输出结果如下：

```
default projectName
Hello, This is ProjectManager.
Hello, greet This is ProjectManager.
default projectName
defaultProjectID is default projectID . defaultProjectName is default projectName
DefaultProjectID is New Default ProjectID . DefaultProjectName is New Default
ProjectName
New ProjectName
New ProjectName
```

其服务器端输出结果如下：

```
----invoking getProjectName() ----
----invoking greeting() ----
----invoking greetingWithParameter be called
----invoking getProjectName() ----
----invoking getDefaultProject----
----invoking getAllProject ----
  default projectID: default projectName
  projectID2: projectName2
----invoking setDefaultProject----
----invoking getDefaultProject----
----invoking getAllProject ----
  default projectID: default projectName
  projectID2: projectName2
----invoking registerProject----
----invoking getProjectNameByID----
----invoking getProjectNameByID----
----invoking getAllProject ----
  default projectID: default projectName
  New ProjectID: New ProjectName
  projectID2: projectName2
----invoking removeProject, Project ID is: New ProjectID
----invoking getAllProject ----
  default projectID: default projectName
  projectID2: projectName2
```

可以看出，达到了预期的测试效果。

5. 其他内容

本例代码请参见配套资源代码中“第 06 章\frontend_dynamic_client”文件夹。

6.6.3 Apache CXF 的动态客户端实现的 Servlet 例子

上述采用 POJO 的例子主要是基于后台启动的 Web Services 服务，也可以把这个服务委托给 Servlet 容器来进行管理。本例采用开源服务器 Tomcat 来做 Servlet 容器。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，都使用 POJO 对象实现客户端和服务器端的数据交换。后台服务端为遵循 JAX-WS 规范的 Endpoint 来实现。前端访问采用两种方式，分别是：① 采用 Apache CXF 本身的 DynamicClientFactory 客户端来实现；② 采用 Apache CXF 本身的 JaxWsDynamicClientFactory 客户端来实现。

采用 Apache CXF 本身的 DynamicClientFactory 客户端来实现的结构如图 6-46 所示。服务器端是在 Servlet 容器中。

其中：① 表示 DynamicClientFactory 客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。在客户端和服务端的 SEI 都是 ProjectManager 接口。

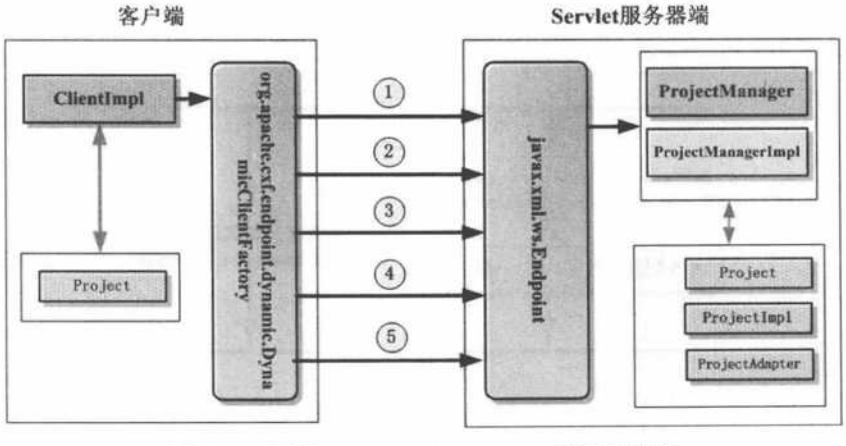


图 6-46　采用 DynamicClientFactory 客户端实现

第二种方式是采用 Apache CXF 本身的 JaxWsDynamicClientFactory 客户端来实现，实现的结构如图 6-47 所示。服务器端是在 Servlet 容器中。

其中：① 表示 JaxWsDynamicClientFactory 客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。在客户端和服务端的 SEI 都是 ProjectManager 接口。

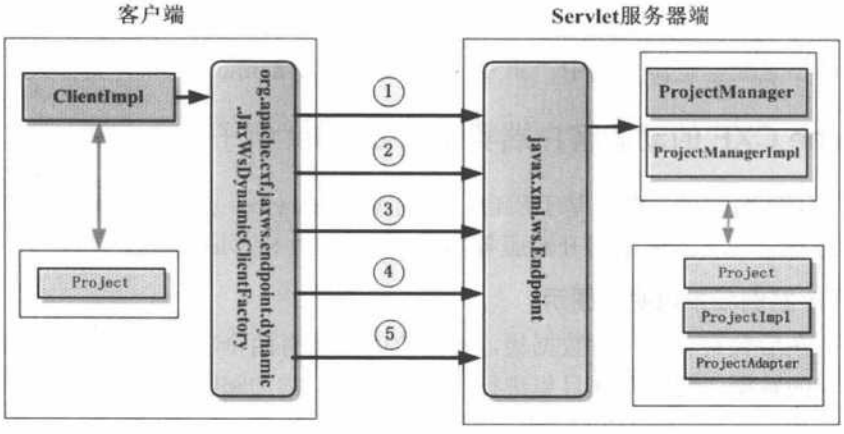


图 6-47　采用 JaxWsDynamicClientFactory 客户端实现

2. 应用程序案例实现步骤

应用程序案例的实现流程如图 6-48 所示。

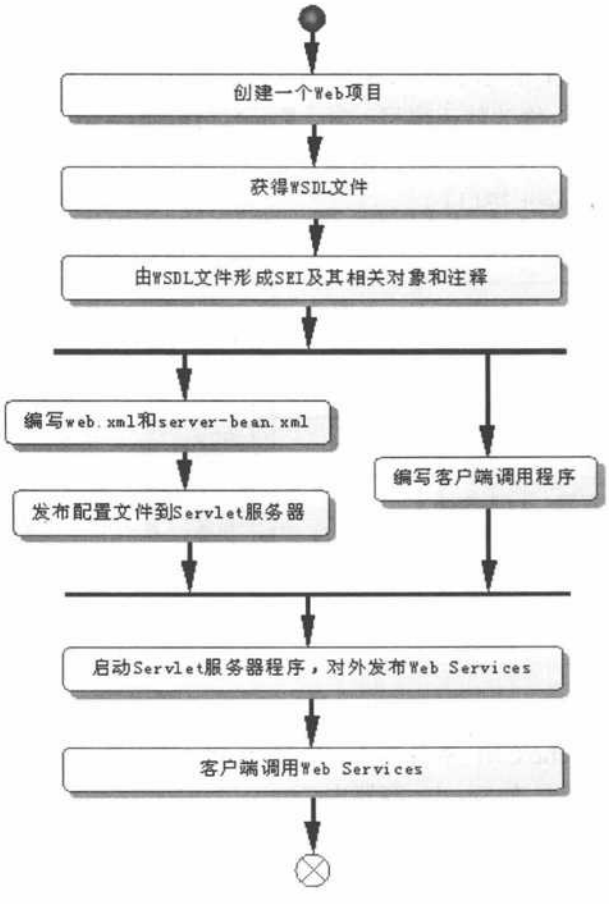


图 6-48　应用程序案例实现流程

步骤说明如下：

（1）创建一个 Web 服务项目。

（2）创建一个服务端点接口（SEI）及其相关对象和注释。

（3）编写 web.xml 和 server-bean.xml。

（4）发布配置文件到 Servlet 服务器。

（5）编写 Web Services 的客户端应用程序。

（6）启动 Servlet 服务器程序，对外发布 Web Services。

（7）客户端调用 Web Services。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 6-49 所示。

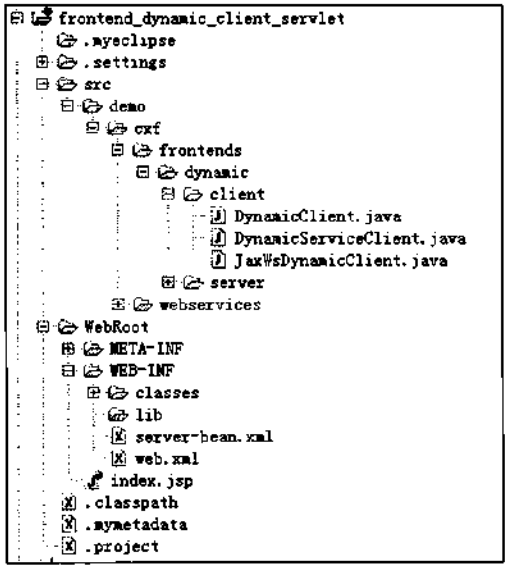


图 6-49　frontend_dynamic_client_servlet 程序结构

包目录包括三个部分，第一个包目录 WebRoot 目录是服务器程序目录，包括 web.xml 和 server-bean.xml 文件，第二个包目录 demo.cxf.frontends.dynamic.client 是客户端程序目录。第三个包目录 demo.cxf.webservices 是 SEI 和数据对象等公用部分，客户端和服务器端都可能引用其内容。

（1）服务器端配置文件

服务器端配置文件包括 web.xml 文件和 server-bean.xml 文件。

首先要定义的 web.xml 文件如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
```

```
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>WEB-INF/server-bean.xml</param-value>
    </context-param>

    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>

    <servlet>
        <servlet-name>CXFServlet</servlet-name>
        <display-name>CXF Servlet</display-name>
        <servlet-class>
            org.apache.cxf.transport.servlet.CXFServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>CXFServlet</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

其中，当 Servlet 启动时，将读取 WEB-INF 目录下的 server-bean.xml 文件，server-bean.xml 文件的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="
          http://cxf.apache.org/transports/http/configuration
          http://cxf.apache.org/schemas/configuration/http-conf.xsd
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd
            http://cxf.apache.org/jaxws
            http://cxf.apache.org/schemas/jaxws.xsd">

    <import resource="classpath:META-INF/cxf/cxf.xml" />
    <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
    <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />
    <jaxws:endpoint id="ProjectManager"
        implementor="demo.cxf.webservices.ProjectManagerImpl"
        address="/ProjectManager" />
</beans>
```

服务器端的程序代码基本上不做修改，把服务器端程序发送到 Tomcat 中。

当 Servlet 容器启动，调用 web.xml 进行初始化，当有 listener 选项时，调用其内容，为 org.springframework.web.context.ContextLoaderListener，然后调用 server-bean.xml 文件，从 jaxws:endpoint 元素中了解到后台的 SEI 是 demo.cxf.webservices.ProjectManagerImpl。公开 Web Services 服务。

（2）客户端调用程序

客户端调用程序有两种方式，一种是采用 JAX-WS 规范的前端客户端，还有一种是 Apache CXF 自定义的前端客户端。

采用 JAX-RS 规范的前端客户端，只需修改后台服务的位置即可，代码如下：

```
……
    public static void main(String[] args) throws Exception {
……
        //用服务端口名称、绑定方式和 Endpoint 地址来创建服务器端口
URL wsdlURL =
 new
URL("http://localhost:18080/frontend_dynamic_client_servlet/ProjectManager?wsdl");
        …
```

注意：笔者的 Tomcat 采用的是 18080 端口。

Apache CXF 的前端客户端修改与基于 JAX-WS 规范相同，代码如下：

```
……
    public static void main(String[] args) throws Exception {
……
        URL wsdlURL = new URL(
    "http://localhost:18080/frontend_dynamic_client_servlet/ProjectManager?wsdl");
      …
```

4. 测试和运行结果

客户端程序和服务器端输出结果与上述的“Apache CXF 的动态客户端技术”的例子的输出结果相同。

5. 其他内容

本例代码请参见配套资源代码中“第 06 章\frontend_dynamic_client_servlet”文件夹。

第 7 章　CXF 的数据绑定

本章主要讲解 Apache CXF 框架的数据绑定（DataBindings）的相关内容，包括 Apache CXF 框架如何支持 JAXB 数据绑定、Aegis 数据绑定、MTOM 数据绑定、XMLBeans 数据绑定等。针对每种数据绑定，首先简介该数据绑定的内容，其次说明 Apache CXF 框架是如何支持该数据绑定的，最后展示示例。

本章内容：

1. 数据绑定（DataBindings）基本概念介绍。
2. JAXB 数据绑定。
3. Aegis 数据绑定。
4. MTOM 使用。
5. XMLBeans 的使用。

7.1　数据绑定（DataBindings）介绍

数据绑定（DataBindings）主要是在 Java 平台应用程序中操作 XML。数据绑定为 Java 开发和使用 XML 提供了一种简单而直接的方法。在实际应用中，基于数据绑定的 Java 应用程序可以在很大程度上忽略 XML 文档的实际结构，而直接使用 XML 文档的数据内容。数据绑定在一般情况下，对于那些将 XML 用于数据交换的 Java 应用程序是比较理想的。XML 和 Java 之间的绑定和转化如图 7-1 所示。

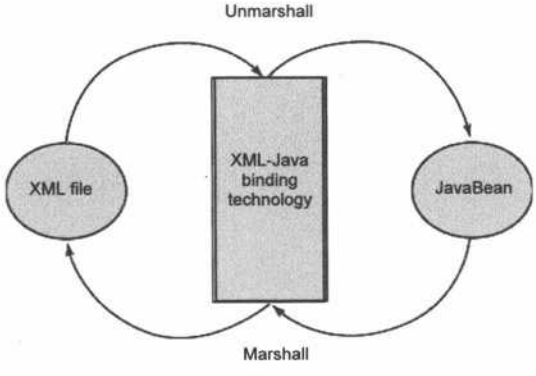


图 7-1　XML-Java 绑定

除了简化编程之外，数据绑定还提供了其他一些好处。由于数据绑定对许多文档细节进行了抽象，因此在内存中处理文档性能会好。数据绑定通常所需要的内存比文档模型方法（如 DOM 或 JDOM）要少。由于不需要遍历文档结构以获取数据，因此用数据绑定方法访问程序内的数据要比用文档模型方法快。最后，在输入时，一些特殊类型的数据（如数字和日期）可以被转换成内部表示，而不是保留为文本形式，这使应用程序可以更有效更合理地使用数据值。

如果数据绑定有这么多的好处，那么何时使用文档模型方法呢？以下两种主要情形下需要用文档模型方法而不是数据绑定：

- 当应用程序确实要关注文档结构的细节时。例如，开发人员正在编写一个 XML 文档编辑器，则会希望采用文档模型，而不是使用数据绑定。
- 当正在处理的文档不需要遵循固定的结构时。例如，对于实现常规的 XML 文档数据库，数据绑定不是一种很好的方法。

关于数据绑定，有几个术语要简单地了解一下。

文法（Grammar）是用于定义一系列 XML 文档结构的一套规则。其中一类文法是 XML 规范所规定的文档类型定义（Document Type Definition，DTD）格式。另一类日渐普及的文法是 XML Schema 规范所定义的 W3C XML Schema（Schema）格式。文法定义了哪些元素和属性可以出现在文档中，以及在文档中元素是如何嵌套的（通常包括嵌套元素的次序和数目）。一些类型的文法（如 Schema）还可以更进一步，使字符数据内容与特定数据类型甚至正则表达式相匹配。

编组（Marshalling）是把内存中的对象生成 XML 表示的过程。与 Java 对象序列化一样，这种表示需要包含所有依赖的对象：主对象、引用的对象和这些对象引用的对象，等等。因此在 Java 和 XML 环境中，编组就是把一些 Java 对象转化成一个（或多个） XML 文档。在数据库环境中，则是把 Java 表示的数据存入数据库。显然，编组的秘密在于把 Java 实例中的面向对象结构转化成适用于 XML 的扁平结构，或者 RDBMS 中的关系结构。

解组（Unmarshalling）是把数据从存储媒介转换到内存中的过程，正好与编组是相反的过程，在内存中根据 XML 表示构建一个对象，需要把 XML 文档解组到 Java VM 中。这里的复杂性不在扁平数据中，因为这不是必需的，而在于从正确的数据到正确的 Java 代码变量的映射。如果映射是错误的，就不可能正确地访问数据。当然，如果再尝试重新编组，还会造成更大的问题，并且问题传播的速度会很快。

往返（Round-tripping）可能是最重要也最容易误解的数据绑定术语。往返用于描述从存储媒介到内存，然后又回到存储媒介的完整循环。在 XML 和 Java 技术环境中，这就意味着从 XML 文档到 Java 的实例变量，然后再回到 XML 文档。正确的往返要求，如果中间没有修改数据，XML 输入和 XML 输出应该是等同的。

Apache CXF 框架支持多种数据绑定。默认情况下是 JAXB 数据绑定，也支持 Apache CXF 框架独有的 Aegis 数据绑定。Aegis 数据绑定可以使用任何数据对象转变为 schema 内容的 Java 数据类型，包括 Maps、Lists 和一些不能采用注释方式的 Java 对象。此外，Apache CXF 框架也支持一种 SourceDataBinding 数据绑定。这种 SourceDataBinding 能够读取和写入数据源对象。Apache CXF 框架也增加了对 XMLBeans 数据绑定的支持。同时 Apache CXF 框架还增加了对 SDO 的数据绑定。

Apache CXF 支持的数据绑定类型如下。

- Aegis (2.1)
- Aegis Databinding (2.0.x)
- JAXB
- MTOM Attachments with JAXB
- XMLBeans
- SDO

7.2 JAXB 数据绑定

JAXB 是一种常用的数据绑定规范。在 Apache CXF 应用中，JAXB 是默认的数据绑定。如果在应用程序的 Spring 配置或者在编程中没有指定的其他数据绑定，Apache CXF 应用将采用 JAXB 规范来实现数据绑定。Apache CXF 框架支持多种版本的 JAXB 规范，如 Apache CXF 2.3 版本就支持 JAXB 2.0 和 JAXB 2.1。

JAXB 通过 Java 注释和类路径的文件建立 XML 文件和 Java 之间的映射。JAXB 支持代码优先和 WSDL 文档优先这两种编程方式。同时，JAXB 还支持基于 WSDL 文档优先的动态客户端技术。这要求应用程序在运行时动态创建一个客户端代理，这要用到 Apache CXF 的 DynamicClientFactory 类。Apache CXF 框架遵循 JAXB 规范说明。如果读者希望对这方面有更多的理解，请阅读 JAXB 规范指南。

7.2.1 JAXB 介绍

JAXB（Java Architecture for XML Binding）是一个业界的标准。这是一项可以根据 XML Schema 产生 Java 类的技术。JAXB 采用 Java 技术，提供了一种简便、快捷的方式将 Java 类转化为 XML 模式，这样可以简化 Web Services 开发。该过程中，JAXB 也提供了将 XML 实例文档反向生成 Java 对象树的方法，并能将 Java 对象树的内容重新写到 XML 实例文档。从另一方面来讲，JAXB 提供了快速而简便的方法将 XML 模式绑定到 Java 表示，从而使得 Java 开发者在 Java 应用程序中能方便地结合 XML 数据和处理函数。

JAXB 在 Java 应用程序中利用了平台无关的 XML 数据去绑定 XML Schema 到 Java 应用程序，同时不需要开发人员具有丰富的 XML 知识。JAXB 是一种 XML 到 Java 的绑定技术，支持 schema 与 Java 对象之间和 XML Schema 与 Java 实例化对象之间的转化。JAXB 由一个运行时应用程序编程接口（API）和辅助工具组成，可以简化访问 XML 文档。使用 JAXB 也有助于建立一个符合和验证某一个 XML 架构（XML Schema）的 XML 文档。

JAXB 提供了 xjc 模式编译器工具\schemagen 模式生成器工具以及运行时框架。可以使用 xjc 模式编译器工具启动一个 XML 架构定义（XSD）来创建一个在 XSD 架构中定义的元素和类型，并映射成为一系列 JavaBeans 集合。也可以启动一系列 JavaBeans 集合设置并使用 schemagen 模式来生成工具创建 XML 架构。一旦 XML 模式和 Java 类之间存在着映射，通过具有约束力的 JAXB 运行环境下 API 的使用，XML 实例文档可以转化或被转化为 Java 实例化对象。在 XML 文档中存储的数据可以被访问，而不需要理解数据结构。最后，开发人员可以使用产生的 Java 类来组装 Web Services 的应用程序。

JAXB 注释类和说明包含了所有的运行时 JAXB 处理 XML 实例文档所需的 API 信息。JAXB 运行 API 支持编组 JAXB 实例化对象到 XML 和反编组 XML 文档返回 JAXB 类实例化对象。另外，开发人员可以使用 JAXB 提供执行 XML 验证，这样可以加强传入和传出的 XML 文件符合 XML Schema 定义的约束。JAXB 在 XML Web Services（JAX-WS 的 2.0）工具中是默认的数据绑定技术。开发人员可以在开发 JAX-WS 应用程序时使用 JAXB 对象。当希望在 Java 应用中调用和协调 XML 数据绑定技术来操纵 XML，也可以在 JAX-WS 开发中独立地使用 JAXB。

图 7-2 说明了 JAXB 的体系结构。

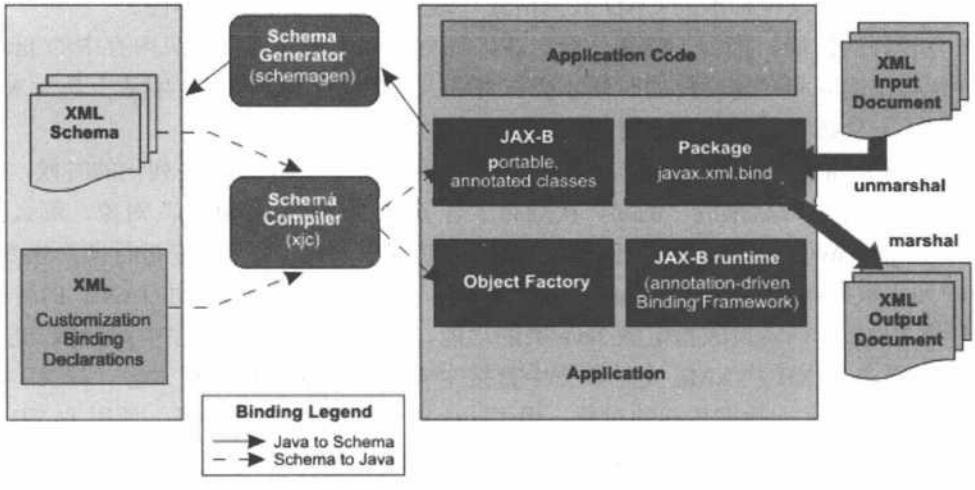


图 7-2　JAXB 的体系结构

这意味着开发人员不需要处理甚至不需要知道 XML 编程技巧，就能在 Java 应用程序中利用核心 XML 数据的灵活性。而且，可以充分利用 XML 的优势而不用依赖于复杂的 XML 处理模型，如 SAX 或 DOM。JAXB 隐藏了许多 XML 细节并且取消了 SAX 和 DOM 中没用的关系——生成的 JAXB 类仅描述原始模型中定义的关系。其结果是结合了 Java 代码的高度可移植和 XML 数据的高度可移植。这些代码可用来创建灵活、轻便的应用程序和 Web Services。

通过 JAXB 对 XML 文档进行的操作主要包括：将符合 XML Schema 规定的 XML 文档解析生成一组相应的 Java 对象；对这些对象进行操作（修改、增加和删除对象的属性，等等）；将这些对象的内容保存到这个 XML 文档中。

JAXB 的应用过程一般来说包含以下几个步骤：

（1）根据应用程序所要操作的 XML 数据格式，撰写相应的 XML Schema。

（2）使用 JAXB 所带的编译工具（Binding Compiler），将这个 XML Schema 文件作为输入，产生一系列相关的 Java Class 和 Interface。

（3）在使用 JAXB 编译工具的时候，可以有选择性地提供一个配置文件，来控制 JAXB 编译工具的一些高级属性。

这些 Java Class 和 Interface 是应用程序操纵 XML 数据的主要接口和方法。

使用 XML 和 JAXB 的优势主要有下面 4 点。

（1）简单易用。由于使用 JAXB 时，可以隐藏对 XML 文档的操作。采用 Java 开发的应用程序不使用 SAX 或 DOM 接口编程去操纵 XML 文档，这样可以使程序量降低、错误

率减少，有助于程序质量的提高。

（2）维护性好。当系统需要升级或更新时，如果只是数据模型发生了改变，开发人员只需要重新定义 XML Schema，然后让 JAXB 重新生成对 XML 文档进行操纵的类，这样就不用定制地修改源程序代码，可以让应用程序具有很好的维护性。

（3）数据兼容性好。当应用程序要进行版本更新时，往往要考虑兼容性问题。由于使用了 XML 作为数据交换的格式，只需要提供 XSLT 的模板就能将原有的数据格式自动转换成新版本的格式。除此以外，开发人员可以将应用程序中的 XML 表示的图形数据文件任意转换成其他标准的 CAD 数据格式，这些都是非常好的兼容性。

（4）有较好的性能。因为 JAXB 运行时在大部分时间中都是直接操纵内存中的 Java 对象，只有在读取和存储文件的时候才会与 XML 文件进行 I/O 操作，实质上是为 XML 文件做了一个缓存，因此运行时的性能很好。

与直接使用 Java 对象序列化相比，JAXB 也有局限性。在使用 Java 序列化的时候，Java 对象和 XML 文档是直接对应起来的。例如，在画布上有一个“java.awt.Button”对象，那么当进行序列化的时候，“java.awt.Button”的所有数据被直接存储到序列化文件中，在进行图形恢复的时候，文件中的序列化数据直接就转换成“java.awt.Button”对象了。然而在使用 JAXB 的时候，由于 JAXB 没有将 XML 文档转换成指定的 Java 类的功能，在 XML 和应用程序中间会多一层。例如，开发人员不能要求 JAXB 从 XML 文档数据中直接生成“java.awt.Button”对象，JAXB 在大多数环境中只能生成中间进行数据传递的对象。用“java.awt.Button”作为例子，使用 JAXB，开发人员只能根据 XML 中的数据生成 JAXB 特有的“MyButton”对象，再通过“MyButton”对象中的数据去初始化“java.awt.Button”对象。这种间接的数据传递所带来的负面影响是使用 JAXB 的系统会产生大量传递数据的中间对象。虽然说，由于 Java 虚拟机自动回收内存的特点，大量中间对象的产生有可能在长时间运行的 Server 应用中降低系统的性能，但是这种数据传递的方法在许多系统设计中都是可行的。

JAXB 开发包下载地址 http://java.sun.com/webservices/downloads/webservicespack.html。

7.2.2 Apache CXF 实现 JAXB 的方式

Apache CXF 框架支持两种方式来配置 JAXB。JAXB 明确了应用程序修改操作的两个属性集：上下文属性和编组性质。Apache CXF 框架允许应用程序添加这些属性。不过要注意，在某些情况下，Apache CXF 会自己使用这些属性。

1. JAXB 属性

开发人员可以通过 JAXBDataBinding 类的两个属性添加内容到 JAXBDataBinding 对象中。在 JAXBDataBinding 类的“contextProperties”和“marshallerProperties”的属性内部都有 Map<String, Object>变量。无论开发人员在 Map<String, Object>中传入什么参数，Apache CXF 将传递给 JAXB。配置文件如下：

```
<jaxws:server id="bookServer"
    serviceClass="org.apache.cxf.anonymous_complex_type.AnonymousComplexTypeImpl"
    address="http://localhost:8080/act"
    bus="cxf">
    <jaxws:invoker>
```

```
    <bean class="org.apache.cxf.service.invoker.BeanInvoker">
      <constructor-arg>
        <bean class="org.apache.cxf.anonymous_complex_type.AnonymousComplexTypeImpl"/>
      </constructor-arg>
    </bean>
  </jaxws:invoker>
   <jaxws:dataBinding>
    <bean class="org.apache.cxf.jaxb.JAXBDataBinding">
      <property name="contextProperties">
       <map>
           <entry>
               <key><value>com.sun.xml.bind.defaultNamespaceRemap</value></key>
               <value>uri:ultima:thule</value>
           </entry>
           </map>
      </property>
    </bean>
    </jaxws:dataBinding>
 </jaxws:server>
```

2. 命名空间前缀管理

JAXB 的参考实现允许应用程序提供一个对象来实现命名空间 URI 映射到 URI 前缀。开发人员可以创建这样一个对象，并通过编组属性来实现这种方式。然而，Apache CXF 框架提供了一个更容易的过程。该 JAXBDataBinding 的 namespaceMap 属性接受一个 Map<String, String>变量。把它看成一个从命名空间 URI 映射到空间映射前缀的 Map。如果程序加载了这个 Map，Apache CXF 框架将为属性建立必要的编组。命名空间 URI 配置文件如下：

```
<jaxws:server id="bookServer"
    serviceClass="org.apache.cxf.anonymous_complex_type.AnonymousComplexTypeImpl"
address="http://localhost:8080/act"
bus="cxf">
   <jaxws:invoker>
    <bean class="org.apache.cxf.service.invoker.BeanInvoker">
      <constructor-arg>
        <bean class="org.apache.cxf.anonymous_complex_type.AnonymousComplexTypeImpl"/>
      </constructor-arg>
    </bean>
  </jaxws:invoker>
   <jaxws:dataBinding>
    <bean class="org.apache.cxf.jaxb.JAXBDataBinding">
      <property name="namespaceMap">
       <map>
           <entry>
               <key><value>http://cxf.apache.org/anonymous_complex_type/</value></key>
               <value>BeepBeep</value>
           </entry>
           </map>
      </property>
    </bean>
    </jaxws:dataBinding>
 </jaxws:server>
```

7.2.3　Apache CXF 实现 JAXB 数据绑定例子

在上述的所有例子中，只要不指定具体的数据绑定模式，都采用的是 JAXB 数据绑定。在这

里就不进行例子的说明了。

7.3 Aegis 数据绑定

Aegis 是 Apache CXF 框架特有的数据绑定方式。本节首先介绍 Aegis 的基础知识，然后说明 Apache CXF 框架如何实现 Aegis 方式，最后通过两个案例程序来进一步说明。

7.3.1 Aegis 介绍

Aegis 是一种数据绑定方式，即可以从 Java 对象映射到基于 XML Schema 描述的 XML 文档。同时，Aegis 也可以由 XML Schema 描述的 XML 文档映射到 Java 对象。Aegis 给开发人员提供了一个更底层操作的映射，同时允许开发人员进行细节控制和定制化。

Aegis 是 XFire 的一个组成部分，它与 XFire 一块移植到 Apache CXF 框架平台中。

1. Aegis 概述

Apache CXF 框架的 Aegis 可以独立地作为 Java 对象和 XML 的映射机制。本节仅描述在 Apache CXF 框架内部如何使用 Aegis。

在某些应用场合下，Aegis 相比较 JAXB 而言是有一些优势的。Aegis 使用更少的配置但能产生更多的、更自然的 XML 映射。例如，Aegis 的默认设置“nillable”，可以在一个位置上为整个服务统一属性，而不必注释每一个元素。Aegis 最大的优势在于定制化 XML 与对象之间的映射而不需要用@annotations 方式来注释 Java 代码。这样可以耦合数据类和 Web Services 绑定类加载的依赖关系。

Aegis 是一种快速的基于 StAX 的数据绑定。可以尽量地简化基于代码优先的 Web Service 开发。Aegis 能够处理大多数的 Java 类并且能为这些类创造出智能 Schemas。

Aegis 的特点包括：

- 不需要注释来暴露实现类。
- 支持接口。
- 支持 collections 数据类型，如 Map 等。
- 支持更广范围的数据类型。

2. Aegis 的基本用法

可以在任何 Web Services 配置中使用 Aegis 数据绑定。用 Aegis 配置的 Web Services 将产生一个有效的 WSDL 描述，并且 Aegis 也可以用来配置任何客户端。可以使用 JAXB 或.NET 语言又或一个脚本语言来实现 Aegis 服务。

通过使用与服务器端同样的 Java 类和配置环境，也可以在客户端使用 Aegis。不过，并不是所有的实际应用都将 Aegis 作为一个客户端来使用，因为 Aegis 缺乏一个 wsdl2java 工具。

在客户端上使用 Aegis 有可能带有严重的兼容性风险。由于没有指定 WSDL 文档，在代码中的微小改动都会导致客户端和服务器不相容。如果在客户端使用 Aegis，应确保双方使用完全相同的 Apache CXF 版本。如果不能做到这一点，应该考虑使用 wsdl2java 生成客户端 JAX-WS/JAXB 代码。

每个 Apache CXF 服务端和客户端使用一个前端：JAX-WS 或简化模式。无论是在 Spring 还是通过 Java 代码，这些 Apache CXF 应用都有一个位置来配置数据绑定。

例如，下面的例子是一个以 Aegis 作为数据绑定的简单前端服务。

```
<simple:server id="pojoservice" serviceClass="demo.hw.server.HelloWorld" address="/
hello_world">
    <simple:serviceBean>
        <bean class="demo.hw.server.HelloWorldImpl" />
    </simple:serviceBean>
   <simple:dataBinding>
      <bean class="org.apache.cxf.aegis.databinding.AegisDatabinding" />
   </simple:dataBinding>
  </simple:server>
 </bean>
```

AegisDatabinding 是实现 Apache CXF 集成一个数据绑定到 Aegis 的实现类。

请注意，AegisDatabinding 类像所有的数据绑定一样都不会被重用。上面的例子使用了一个匿名嵌套 Bean 绑定。如果开发人员为第一个类进行了数据绑定，一定要使用 scope ='prototype' 这样的方式进行说明。

3. Aegis 的操作

Aegis 是如何工作的？Aegis 为每个服务提供了从 Java 类型（Class<?> objects）映射到 XML Schema 类型的维护。Aegis 采用映射方式来读取和编写 XML。既包含一个简单的服务，也覆盖所有的 Java 类型或 Java 内置类型，还包括预先定义的映射到 XML Schema 或有属性的 Bean 类（递归）等简单其他类型。

让开发人员开始一个从 Java 到 XML 映射的序列化。给定一个 Java 对象，Aegis 判断该 Java 对象是否有一个映射。默认情况下，Aegis 有一系列 Java 基本类型映射为 XML Schema 中定义的基本类型，加上其他一些特殊项目设置。这些映射都由 Java 类来执行。Aegis 既可以把对象转化为 XML，也可以把 XML 转化为对象。特殊情况下 Aegis 也可以把 DataSource 或 DataHandler 映射到 MTOM 的附件。

如何发现一个类型在 Aegis 没有映射？默认情况下，Aegis 调用类型创造器创造一个新的映射。类型创造器使用多种机制来创建 Java 对象到 XML Schema 的映射。这些机制包括反射、注释和 XML 类型映射文件。作为映射工作的一部分，Aegis 指派一个基于 Java 包的命名空间 URI。（注：在这时不支持 elementForm='unqualified'）这些映射被一个通用的映射类执行并保存下来。

如何进行反序列化过程？在反序列化情况下（JAXB 称这个过程为解组），Aegis 发布一个 XML 元素并创建一个 Java 对象。Aegis 实现从 Java 类型映射到 XML Schema 类型的维护。默认情况下，当一个 XML 实例文档没有说明一个给定元素的类型信息时，Aegis 怎样才能确定 Java 类型？在 Apache CXF 框架的外部，应用程序需要告诉 Aegis 这个 XML 文档根元素申请的预期类型。在 Apache CXF 内部，Aegis 可以通过服务信息来获取部分信息。服务的 WSDL 信息配置提供足够的信息来组合每个部分的 XML 架构类型。一旦前端已确定了类型，WSDL 信息配置可以调用带有 QName 类型的 Aegis 以及 Aegis 的映射。

因为 Aegis 在服务组件创建了类型映射。Aegis 不能动态创建或选择一个基于 XML Schema 的 Java 类，因此无法从 XML 中启动类型创建者。

4. 使用对服务接口不可见的 Java 类

许多 Web Service 开发人员需要使用一些不能直接通过反射服务接口来获得的类型。下面是开发人员使用属性和参数类型的一些例子。

- 声明一个基类，但输入一个子类并且继承该基类。
- 声明一个原始 Collection 类，如 Set、List 或 Map，并发送任意对象作为主键值和属性值。
- 声明为基本异常的“throws”类型，然后抛出其继承出来的其他异常类。
- 声明一个接口或抽象类型。

Aegis 可以处理上面所有的一切。除了接口外，这里涉及两种机制：根类列表和 xsi:type 属性。

正如上文所述，Aegis 可以写任何“anything”，但它只能读取这种映射类型的对象。开发人员必须给 Aegis 所有类型的列表，在该列表中表明可见的服务，必须指示 Aegis 在发送时需要的 xsi:type 属性。

接口和抽象类是有区别的。显然，Aegis 不能实例（通过运行“new”）一个接口。因此，知道一个特定的 XML Schema 类型映射到一个接口是没有足够信息的。为了能够读取 XML 元素对应的接口，Aegis 要知道一个“代理类”实现接口。开发人员必须给 Aegis 一个接口类型映射到代理类的名称。

Aegis 核心是 AegisContext 类。每个 AegisDatabinding 对象都有一个 AegisContext，但是在 AegisContext 之间无法共享数据绑定。

默认情况下，AegisDatabinding 将按照默认属性创建自己的 AegisContext。要配置额外的类型，以及控制其他选择，开发人员必须定制化地创建自己的 AegisContext 并指定一些属性，然后把 AegisContext 对象传递到 AegisDatabinding 对象。

要使用附加类或接口，需要在 AegisContext 中设置两个或三个属性。

- rootClasses 是一组 Java 的 Class<?>对象。它们已经被添加到已知 Aegis 类型的列表。Aegis 将为每个类型创建映射。为方便起见，可以使用 Spring 的 rootClassNames 属性。这是一个字符串列表，其中包含类名。
- writeXsiTypes 是一个布尔类型。将其设置为 true 可以用于发送 xsi:type 属性。
- beanImplementationMap 是从 Class<?>到类名的映射。使用该属性可以指定接口（或抽象类）的代理类。

5. 全局类型创建选项

这里有一些默认类型映射的全局属性选项。开发人员可以通过创建一个 org.apache.cxf.aegis.type.TypeCreationOptions 并传递给 AegisContext 对象来控制和使用这些属性。

这里有 4 个属性，其中两个是更常用的属性。

defaultNillable 是定义在 xsd:sequences 中 xsd:element 项内 nillable 属性的默认值属性。默认情况下，该值是 true。因为任何 Java 的引用都可以为空。然而，nillable='true'在一些 wsdl2java 工具处理起来会更麻烦，因此许多开发人员更喜欢默认设置为 nillable='false'。

defaultMinOccurs 是定义 xsd:sequences 在创建 Java 数组中 xsd:element 项的 minOccurs 的默认值属性。结合 nillable，开发人员经常需要调整这个值从 0 到 1 获得更多有用的数组映射。

defaultExtensibleElements 导致每个序列结束一个 xsd:any。这里的想法是 Schema 的演变过程，一个从旧版本产生的 Java 客户端数据不会接受一个较新版本服务端产生的更多元素。故开发人员

要谨慎使用此功能。Web Services 的版本管理是一个非常复杂的话题，xsd:any 可能带来意想不到的后果。

defaultExtensibleAttributes 允许每个元素是任何属性。默认情况下，Aegis 不可以映射任何属性或参数到属性。

请注意，这些都是默认的类型创造者的选择。如果开发人员采取了创建一个定制的类型创建者的方式，第一步就是选择或取消这些选项。

下面是采用 Java 编程来设置这些选项的简单例子。在 Spring 中也会有类似的功能。

```
TypeCreationOptions tOpts = new TypeCreationOptions();
tOpts.setDefaultMinOccurs(1);
tOpts.setDefaultNillable(false);
AegisDatabinding aDB = new AegisDatabinding();
aDB.getAegisContext().setTypeCreationOptions(tOpts);
sFactory.getServiceFactory().setDataBinding(aDB);
```

6. Bean 类型映射的细节控制

现在已经开始从使用 Aegis 默认配置到更详细地了解其控制细节过程。下一阶段的细节是通过 XML 映射文件和注释来定制化类型和行为。

（1）XML 映射文件

XML 映射文件是 Aegis 功能的一个重大特征。映射文件允许开发人员定义映射过程的细节，而不必修改任何 Java 类型或维护一个保持了很多类型映射指令的中心文件。

Aegis XML 映射适用于服务和 Bean 类。所谓 Bean 类，是指遵循 beans 模式规范，同时在 Web Service 中使用的 Java 类。

总的来说，前端映射服务到 XML Schema，填写 XML Schema 元素和类型与组件，然后从 schema 项映射到 Java 类。不过，Aegis 也通过 XML 配置来提供实现的方法和参数，这应该是前端处理的工作。这种操作模式也非常适用于简单前端，因为简单前端模式没有其他方法来控制这些映射。Aegis 服务配置最好在简单前端上使用。

对于 Bean 和服务定制，Aegis 通过类装载器查找到相关的配置文件进行定制。如果开发者的类是 my.hovercraft.is.full.of.Eels，Aegis 将会在所有的类路径中搜索 /my/hovercraft/is/full/of/Eels.aegis.xml 文件。换句话说，如果 Eels.class 是放在一个 JAR 文件或一个目录下面，Eels.aegis.xml 与类文件在同一个目录下。另一方面，Aegis 类装载器可以在一个完全不同的 JAR 或 tree 为类创建 XML 文件，只要 XML 文件最终在相同的逻辑位置。换句话说，开发人员甚至不需要知道来源，这给开发人员提供了非常大的便利。

（2）Bean 映射

下面例子文件是一个很简单的映射。这需要属性' horse '，重命名为' Feathers '，使之成为一种属性，而不是元素。

```
<mappings>
  <mapping name="">
    <property name="horse" mappedName="Feathers" style="attribute"/>
  </mapping>
</mappings>
```

1）名称和命名空间

开发人员还可以指定 bean 自身的 QName。下面的映射导致类有一个 QName（urn:north-pole:operations）的雇员信息。

```
<mappings xmlns:np="urn:north-pole:operations">
  <mapping name="np:Employee">
  </mapping>
</mappings>
```

请注意，映射元素上的命名空间，前缀用于指定元素 QName 的名称/主题属性。

这将导致如下面的一个映射：

```
<np:Employee xmlns:np="urn:north-pole:operations">
  <np:Name>Santa Claus</np:Name>
  <np:Title>Chief Present Officer (CPO)</np:Title>
</np:Employee>
```

2）忽略属性

如果开发人员不希望序列化某些属性，可以通过设置属性 ignore 来忽略它：

```
<mappings>
  <mapping>
    <property name="propertyName" ignore="true"/>
  </mapping>
</mappings>
```

3）minOccurs 和 Nillable 属性

默认 Aegis 映射是假设任何 Java 对象都可以为空，即相应 Schema 元素中 minOccurs 应该是 0 和 nillable = true。属性由映射来控制。

```
<mappings>
  <mapping>
    <property name='everpresentProperty' minOccurs='1' nillable='false'/>
 </mapping>
<mappings>
```

4）替代类型绑定

开发人员希望指定自己类型映射的属性。这也无须创建任何 Java 的定制类型，就能通过 XML 映射文件来配置。默认情况下，如果 Aegis 映射一个属性为日期类型，那么可以使用 XML 属性类型 xsd:dateTime。下面是一个使用 xsd:date 的例子。

```
<mappings xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <mapping>
        <property name="birthDate"
            type="org.apache.cxf.aegis.type.basic.DateType"
            typeName="xsd:date"
            />
    </mapping>
</mappings>
```

5）集合类

Aegis 将“raw”的集合类型映射为一个 xsd:any 类型。如果希望 WSDL 去显示某些特定类型

的集合，最简单的办法是使用 Java 泛型，而不是原始类型。如果由于某种原因而不能这样做，可以使用 componentType 和 keyType 属性去指定 Java 类。

6）不同服务的多个映射

在同一类型的不同服务下该如何定义不同的映射行为呢？文件的 mapping 元素接受 uri 属性。每个 AegisContext 都有一个 mappingNamespaceURI 属性。如果在 aegis.xml 文件映射一个 uri 属性，该 uri 属性必须符合现有服务的 URI。

7）服务和参数

对于服务而言，映射文件定义操作、参数和属性。

下面的例子指定 getUnannotatedStrings 返回一个命名为 UnannotatedStringCollection 的元素。这是一个 String 值的基本集合。接着，例子指定了 getValues 第一个参数也是一个 String 值的基本集合。

```
<mappings>
    <mapping>
        <method name="getUnannotatedStrings">
            <return-type name="UnannotatedStringCollection"
               componentType="java.lang. String"/>
        </method>
       <method name="getValues">
        <parameter index="0" componentType="java.lang.String"/>
       </method>
    </mapping>
</mappings>
```

8）注释方式

与 JAXB 的一样，Aegis 也支持一些 Java 注释来控制映射过程。这些属性都仿照 JAXB 属性。Aegis 在包 org.apache.cxf.aegis.type.java5 中定义这些属性。属性列表如下：

- XmlAttribute
- XmlType
- XmlElement
- XmlReturnType
- XmlIgnore

此外，Aegis 遵循 JAXB 注释，其内容如下：

- javax.jws.WebParam
- javax.jws.WebResult
- javax.xml.bind.annotation.XmlAttribute
- javax.xml.bind.annotation.XmlElement
- javax.xml.bind.annotation.XmlSchema
- javax.xml.bind.annotation.XmlType
- javax.xml.bind.annotation.XmlTransient

注意，Aegis 不处理 package-info.java 类，所以 XmlSchema 的应用可以用于处理这些类。

7. 创建自己的类型映射

如果想完全控制 Java 和 XML 之间的映射，必须创建自己的类型映射。要做到这一点，开发人员应该创建一个类，继承 org.apache.cxf.aegis.type.Type，然后必须注册为服务类型，即可实现映射。

要注册类型映射，有两种选择。

如果开发人员只是在应用程序中的服务中添加自定义映射类型，最简单是从 AegisContext 检索 AegisContext 类型映射，并注册自定义映射类型作为 Class<?>来映射到一个自定义映射对象。

如果开发人员想完全控制这一进程，可以创建自己的类型映射。默认类型映射类是标准的 map 类型。开发人员可以使用这些，或者可以创建自己的类型映射并实现。设置喜欢的类型映射，并在服务初始化之前，安装在应用程序的上下文。

8. 自定义类型创建

如果开发人员希望在 Aegis 构建 Java 的新映射类型和新类型？可以创建自定义的 TypeCreator，要么放在类型创造者的前面，要么更换整个标准列表。

类型创造者与类型映射是一样的，开发人员可以在一个 DefaultTypeMapping 实例化对象中调用 setTypeCreator 来引入新的类型。

9. 使用 Aegis

大多数使用 Aegis 的开发人员都只使用简单前端。如果曾经在另一些 Web Services 框架下使用 Aegis 的开发人员，则必须修改所有使用 Aegis 或一些其他绑定的代码，所以使用基于 Aegis 数据绑定的 JAX-WS 前端没有太大的优势。开发人员也可能希望采用 Aegis 做一个专业化的前端，如 JAX-RS。

（1）Spring 配置

大多数使用 Aegis 的开发人员只使用简单前端。

```
<simple:server id="pojoservice" serviceClass="demo.hw.server.HelloWorld" address="/
hello_world">
    <simple:serviceBean>
        <bean class="demo.hw.server.HelloWorldImpl" />
    </simple:serviceBean>
        <simple:dataBinding>
      <bean class="org.apache.cxf.aegis.databinding.AegisDatabinding" />
    </simple:dataBinding>
<!-- Use this property only for XFire compatibility -- this version for 2.0.x ...
              <property name="serviceConfigurations">
                  <list>
                    <bean
class="org.apache.cxf.jaxws.support.JaxWsServiceConfiguration"/>
                    <bean
class="org.apache.cxf.aegis.databinding.AegisServiceConfiguration"/>
                    <bean
class="org.apache.cxf.service.factory.DefaultServiceConfiguration"/>
                 </list>
              </property>
-->
  </simple:server>
 </bean>
```

开发人员还可以在 JAX-WS 上使用 Aegis。这里有一个 Spring 配置的例子。

```
<bean id="aegisBean" class="org.apache.cxf.aegis.databinding.AegisDatabinding" scope=
"prototype"/>
 <bean id="jaxws-and-aegis-service-factory"
   class="org.apache.cxf.jaxws.support.JaxWsServiceFactoryBean"
   scope="prototype">
       <property name="dataBinding" ref="aegisBean"/>
<!-- Use this property only for XFire compatibility -- this version for 2.0.x ...
              <property name="serviceConfigurations">
                  <list>
                    <bean class="org.apache.cxf.jaxws.support.JaxWsServiceConfiguration"/>
                    <bean class="org.apache.cxf.aegis.databinding.AegisServiceConfiguration"/>
                    <bean class="org.apache.cxf.service.factory.DefaultServiceConfiguration"/>
                  </list>
              </property>
-->
<!-- Use this property only for XFire compatibility -- this version for 2.1
              <property name="serviceConfigurations">
                  <list>
                    <bean class="org.apache.cxf.jaxws.support.JaxWsServiceConfiguration"/>
                    <bean class="org.apache.cxf.aegis.databinding.
XFireCompatibilityServiceConfiguration"/>
                    <bean class="org.apache.cxf.service.factory.DefaultServiceConfiguration"/>
                  </list>
              </property>
-->
 </bean>

 <jaxws:endpoint id="my_service_endpoint" implementor="#my-service" address="/MyIndex">
  <jaxws:serviceFactory>
   <ref bean='jaxws-and-aegis-service-factory' />
  </jaxws:serviceFactory>
 </jaxws:endpoint>
```

(2) Java 配置

下面是一个使用 Java 配置的简单前端例子。

```
import org.apache.cxf.aegis.databinding.AegisDatabinding;
import org.apache.cxf.frontend.ServerFactoryBean;

ServerFactoryBean sf = new ServerFactoryBean();
sf.setServiceClass(serviceClass);
sf.setAddress("http://myhost/service");
sf.getServiceFactory().setDataBinding(new AegisDatabinding());
sf.create();
```

同样，这是一个客户端使用简单前端的例子。请注意 WSDL 的用法。如果没有 WSDL，简单的前端将试着通过 Java 反射来获得参数的名称。与往常一样，不带参数名字的 Java 接口，参数名称就默认为“arg0”、“arg1”等。

```
import org.apache.cxf.aegis.databinding.AegisDatabinding;
import org.apache.cxf.frontend.ClientProxyFactoryBean;

ClientProxyFactoryBean factory = new ClientProxyFactoryBean();
factory.setServiceClass(serviceInterfaceClassObject);
factory.setAddress("http://myhost/service");
factory.setWsdlLocation("http://myhost/service?wsdl");
```

```
factory.getServiceFactory().setDataBinding(new AegisDatabinding());
MyService client = (MyService) factory.create();
```

（3）XFire 兼容性

XFire 有几种从 Java 包名生成命名空间 URI 的不同约定。这些约定增加了一个额外的“/”字符。

如果开发人员需要使用 XFire 来进行互操作，要给此行为增加兼容性。需要启用加入此兼容性。

这是一种额外服务配置：XFireCompatibilityServiceConfiguration（在 2.0.x 版本，AegisServiceConfiguration）。

上面的 Spring 例子显示了通过注释方式来展现配置的内容。对于在 Java 客户端，还需要添加服务配置到服务工厂，其代码如下：

```
import org.apache.cxf.aegis.databinding.AegisServiceConfiguration;
// for 2.0.x
sf.getServiceFactory().getServiceConfigurations().add(0, new AegisServiceConfiguration());
// for 2.1
sf.getServiceFactory().getServiceConfigurations().add(0, new
XFireCompatibilityServiceConfiguration());
```

这将更改 Apache CXF 框架生成的默认命名空间，这样与 XFire 的生成代码方式相同。

10. Class <->XML 映射概述

Aegis 有一个灵活的映射系统，让开发人员可以控制 bean 如何运作。默认情况下，POJO 是通过它们的名字和命名空间来进行序列化的。如果开发人员在“org.codehaus.xfire”包中有一个“Employee”类。被序列化后其命名空间为 http://xfire.codehaus.org，同时其本地名称是“Employee”。

例如，Java 类的代码例子如下：

```
public class Employee
{
  private String name;
  private String title;

  public String getName() { return name; }
  public void setName(String name) { this.name = name; }

  public String getTitle() { return title; }
  public void setTitle(String title) { this.title = title; }
}
```

这样的语句在 XML 将转化为下面内容：

```
<Employee xmlns="http://xfire.codehaus.org">
  <name>Santa Claus</name>
  <title>Chief Present Officer (CPO)</title>
</Employee>
```

在 XML Schema 这将成为一个复杂类型：

```
<xsd:complexType name="Employee">
  <xsd:sequence>
```

```
    <xsd:element name="name" type="xsd:string" minOccurs="0" maxOccurs="1/>
    <xsd:element name="title" type="xsd:string" minOccurs="0" maxOccurs="1/>
  </xsd:sequence>
</xsd:complexType>
```

Aegis 支持的类型如下。

- 基本类型：int、double、float、long、byte[]、short、String、BigDecima。
- Arrays。
- Collections——包括 Maps。
- 日期：java.util.Date、java.util.Calendar、java.sql.Timestamp、java.sql.Date、java.sql.Time。
- XML：org.w3c.dom.Docmument、org.jdom.Element、XMLStreamReader。
- 复杂类型，上述各种类型的复合。

如果在 Java Bean 的构造函数进行属性定义，确保其默认的构造函数（即不带参数）存在。（Aegis 需要实例化的 Java 类有一个无参数构造函数。）

11. 在 Java 中默认设置 minOccurs 和 nillable 参数

如果开发人员需要更多属性，也许是大部分属性或全部属性，那 minOccurs 属性值不是 0、或 nillable 属性值不是 false。开发人员可以通过编写 Java 代码在任何地方改变 Aegis 的默认值。

下面是一个例子：它从服务工厂中提取绑定供应商，并更改配置参数。

```
ServerFactoryBean sf = new ServerFactoryBean();
sf.setServiceClass(serviceClass);
sf.setAddress("http://myhost/service");
AegisDatabinding db = new AegisDatabinding();
sf.getServiceFactory().setDataBinding(db);

DefaultTypeMappingRegistry tmr = (DefaultTypeMappingRegistry)db.getTypeMappingRegistry();
// here we disuade XFire from its rather annoying tendency to assume that, just because
// anything in Java can be null, that we want to advertise all that nullity all over.
Configuration configuration = tmr.getConfiguration();
configuration.setDefaultMinOccurs(1);
configuration.setDefaultNillable(false);

sf.create();
```

7.3.2　采用简化前端、Aegis 数据绑定的例子实现

本程序演示 Apache CXF 框架采用 Aegis 数据绑定来实现客户端和服务器端的相互调用，客户端采用 Apache CXF 的代码优先（Java First）前端应用模式。采用 POJO 对象来作为中间传输对象。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，都使用 POJO 对象实现客户端和服务器端的数据交换。后台服务端由遵循 JAX-WS 规范的 Endpoint 来实现。前端访问采用 JAX-WS 规范的 Service 方式，实现如图 7-3 所示。

其中：① 表示客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI

修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目。⑤ 表示客户端通过 SEI 查看后台的全部项目情况。在客户端和服务端的 SEI 都是 ProjectManager 接口。

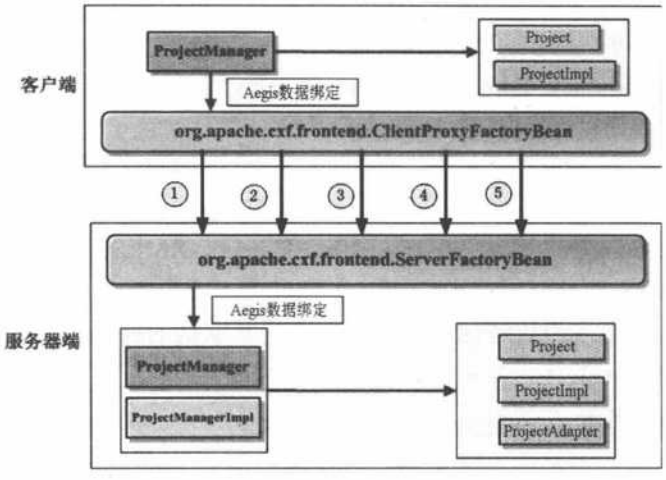


图 7-3　JAX-WS 规范简化前端 Aegis 数据绑定实现结构图

在编写本程序例子时，主要与第 6 章“采用 POJO 并基于 JAX-WS 服务的前端实现”进行对比。

2. 应用程序案例实现步骤

要创建 Aegis 数据绑定的前端实现 Web Services 程序的流程如图 7-4 所示。

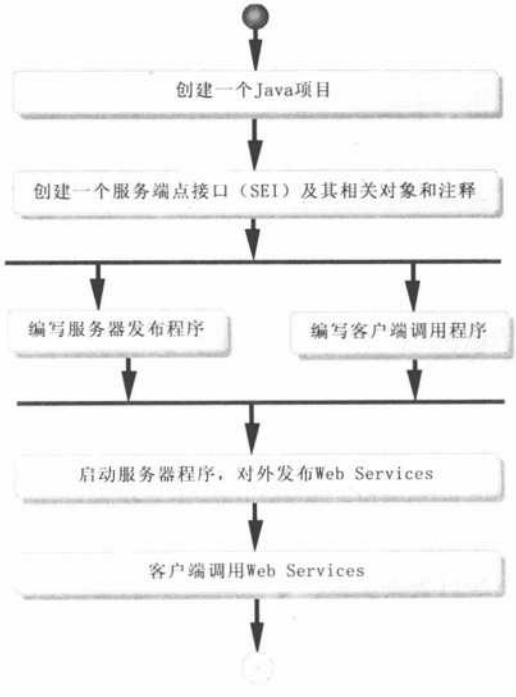


图 7-4　应用程序案例实现流程图

流程图说明，步骤如下：

（1）创建一个 Java 项目。

（2）创建一个服务端点接口（SEI）及其相关对象和注释。

（3）编写发布这个 Java 程序为 Web Services 的服务端程序。

（4）编写 Web Services 的客户端应用程序。

（5）启动服务器程序，对外发布 Web Services。

（6）客户端调用 Web Services。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 7-5 所示。

包目录包括 3 个部分，第一个包目录 demo.cxf.databing.aegis.server 是服务器程序目录。第二个包目录 demo.cxf.databing.aegis.client 是客户端程序目录。第三个包目录 demo.cxf. webservices 是 SEI 和相关对象等公用部分，客户端和服务器端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口，服务器端和客户端都必须引用。但服务器端和客户端对公用部分的引用是不同的。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口，而服务器端程序对于接口和实现类都要引用。

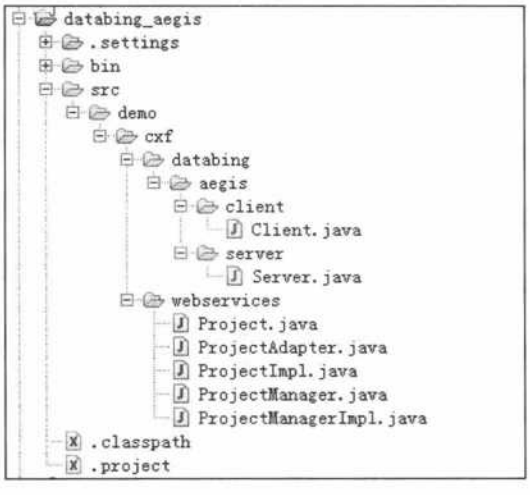


图 7-5　databing_aegis 程序结构

与第 6 章中“采用 POJO 并基于 JAX-WS 服务的前端实现”相比，只有客户端和服务端的数据绑定方式不同，其余的内容全部相同。“采用 POJO 并基于 JAX-WS 服务的前端实现”采用的是默认的 JAXB 数据绑定，而本程序采用的是 Aegis 数据绑定，下面只列出有区别的两个文件的代码。

（1）服务器端发布程序

第一个文件是服务器发布程序，即 Server.java，代码如下：

```
package demo.cxf.databing.aegis.server;

import org.apache.cxf.aegis.databinding.AegisDatabinding;
import org.apache.cxf.frontend.ServerFactoryBean;
```

```
import demo.cxf.webservices.*;

public class Server {

    public Server() {
        System.out.println("Starting Server");
        ProjectManager projectManager = new ProjectManagerImpl();
        String address = "http://localhost:9000/ProjectManager";
        ServerFactoryBean svrFactory = new ServerFactoryBean();
        svrFactory.setServiceClass(ProjectManager.class);
        svrFactory.setAddress(address);
        svrFactory.setServiceBean(projectManager);
        // 设置数据的绑定模式为 AegisDatabinding
        svrFactory.getServiceFactory().setDataBinding(new AegisDatabinding());
        svrFactory.create();
    }

    public static void main(String[] args) throws Exception {
        new Server();
        System.out.println("Server ready...");

        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

其中黑体字说明了服务器采用 Aegis 数据绑定来发布程序。

（2）客户端访问程序

第二个文件是客户端访问程序，即 Client.java，代码如下：

```
package demo.cxf.databing.aegis.client;

import org.apache.cxf.aegis.databinding.AegisDatabinding;
import org.apache.cxf.frontend.ClientProxyFactoryBean;
import demo.cxf.webservices.Project;
import demo.cxf.webservices.ProjectImpl;
import demo.cxf.webservices.ProjectManager;

public class Client {
    public Client() {   }

    public static void main(String[] args) throws Exception {
        System.out.println("Starting Client");

        //定义 ClientProxy 的代理类
        ClientProxyFactoryBean factory = new ClientProxyFactoryBean();

        //设置 JaxWs 代理工厂的属性，包括服务类和地址
        factory.setServiceClass(ProjectManager.class);
        factory.setAddress("http://localhost:9000/ProjectManager");
        factory.getServiceFactory().setDataBinding(new AegisDatabinding());

        //获取服务类的对象并处理
        ProjectManager projectManager = (ProjectManager) factory.create();
        //System.out.println(projectManager.getProjectName());
```

```
        //项目的基本情况
        getGreet(projectManager);
        greetingWithParameter(projectManager);
        getProjectName(projectManager);
        getDefaultProject(projectManager);
        getAllProject(projectManager);

        //修改默认项目，查看项目的基本情况
        setDefaultProject(projectManager);

        //增加一个项目，查看项目的基本情况
        registerProject(projectManager);

        //删除一个项目，查看项目的基本情况
        removeProject(projectManager);
        System.exit(0);
    }

    //调用 SEI 的 getGreet 方法
    private static void getGreet(ProjectManager projectManager)throws Exception {
        System.out.println(projectManager.greeting());
    }

    ……
}
```

其中黑体字说明了客户端采用 Aegis 数据绑定来访问程序。

4. 测试和运行结果

运行客户端程序，其客户端程序和服务器端输出结果与采用 POJO 并基于 JAX-WS 服务的前端实现案例相同。

5. 其他内容

上述的所有调用都基于 JAX-WS 规范。

本例代码请参见配套资源代码中“第 07 章\databing_aegis”文件夹。

7.3.3　采用简化前端 Aegis 数据绑定的 Servlet 例子实现

本程序演示 Apache CXF 框架采用 Aegis 数据绑定来实现客户端和服务器端的相互调用，客户端采用 Apache CXF 的代码优先（Java First）前端应用模式，服务器端采用 Servlet 容器，采用 POJO 对象来作为中间传输对象。本例 Servlet 容器采用 Tomcat6。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，都使用 POJO 对象实现客户端和服务器端的数据交换。后台服务端为遵循 JAX-WS 规范的 Server 来实现。前端访问采用 JAX-WS 规范的 Service 方式，实现的结构如图 7-6 所示。

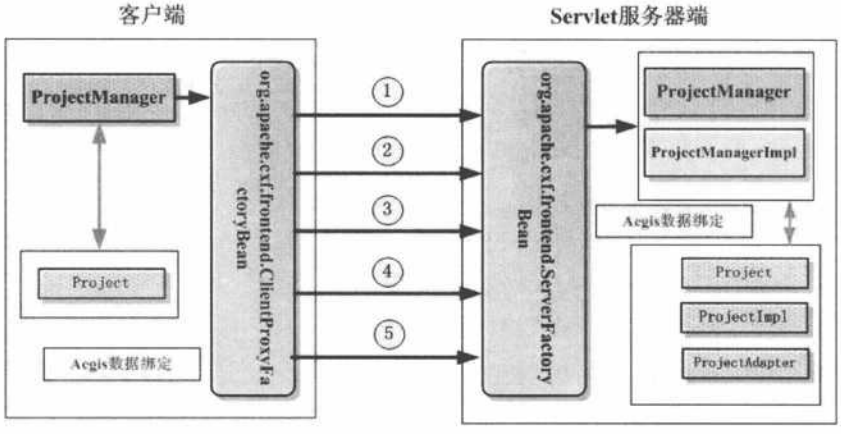


图 7-6　JAX-WS 规范简化前端 Aegis 数据绑定 Servlet 实现结构图

2. 应用程序案例实现步骤

要创建Aegis数据绑定并基于Servlet容器的前端实现Web Services程序的流程步骤如图7-7所示。

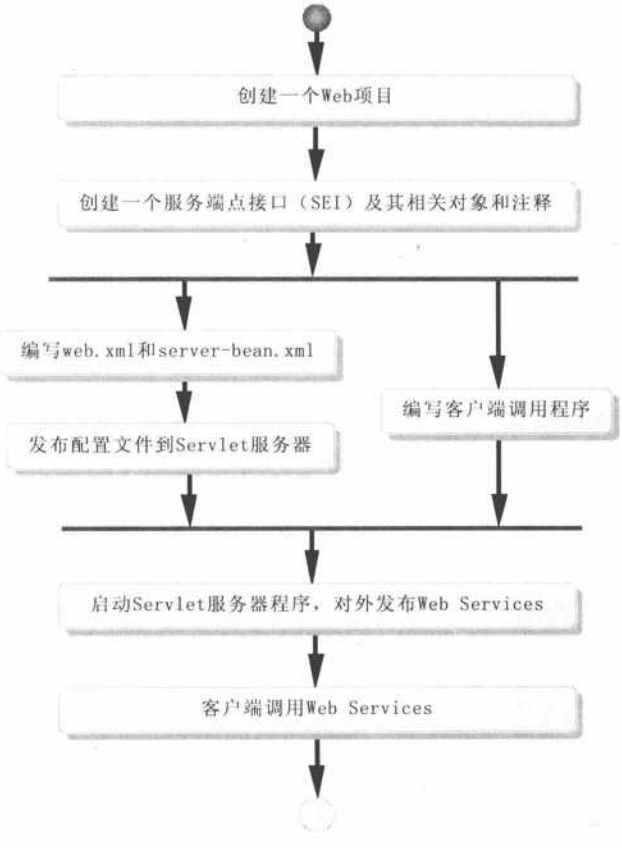


图 7-7　应用程序案例实现流程图

流程图说明，步骤如下：

（1）创建一个 Web 服务项目。

（2）创建一个服务端点接口（SEI）及其相关对象和注释。

（3）编写发布这个 Java 程序为 Web Services 的服务端程序。

（4）编写 web.xml 和 server-bean.xml。

（5）发布配置文件到 Servlet 服务器。

（6）启动 Servlet 服务器程序，对外发布 Web Services。

（7）客户端调用 Web Services。

其中第 1 步和第 3 步与上述采用自带的 HTTP 服务器类似。主要是第 2 步。该步骤不是采用自带的服务，而是放置到 Servlet 容器中。我们采用与 Spring 框架集成的方式来实现。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 7-8 所示。

包目录包括三个部分，第一个包目录 WebRoot，是服务器程序目录，包括 web.xml 和 server-bean.xml 文件；第二个包目录 demo.cxf.databing.aegis.client，是客户端程序目录；第三个包目录 demo.cxf.webservices，是 SEI 和数据对象等公用部分，客户端和服务器端都有可能引用其内容。

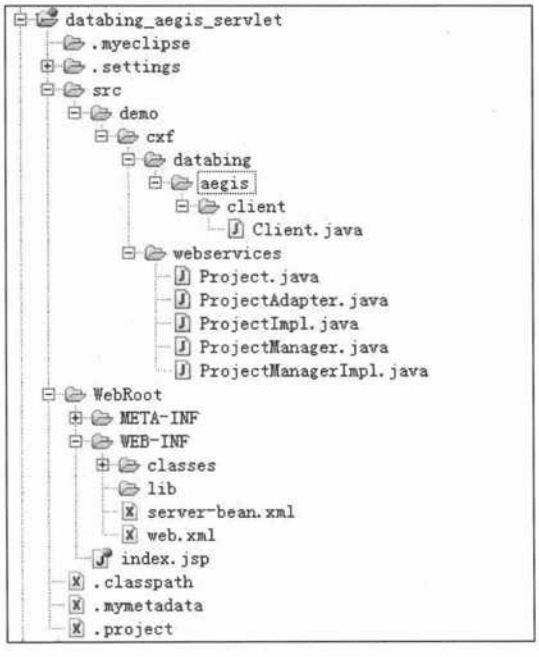


图 7-8　frontend_simple_servlet 程序结构

服务器端点接口（SEI）是公用部分的接口。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口。由于本程序直接发布到 Servlet 容器，所以没有单独编写服务器发布程序，直接把配置文件和编译好的服务器端点接口（SEI）组件等部署到 Servlet 容器即可。

下面分别介绍两部分：①服务器端配置文件；②客户端调用程序。

（1）服务器端配置文件

服务器端配置文件包括 web.xml 文件和 server-bean.xml 文件。

web.xml 文件的定义与第 6 章“采用 POJO 并基于 Servlet 容器的前端实现”的案例相同，这里就不再列出了。其中，当 Servlet 启动，将读取 WEB-INF 目录下的 server-bean.xml 文件，server-bean.xml 文件的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:simple="http://cxf.apache.org/simple"
    xmlns:soap="http://cxf.apache.org/bindings/soap"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
        http://cxf.apache.org/bindings/soap
        http://cxf.apache.org/schemas/configuration/soap.xsd
        http://cxf.apache.org/simple
        http://cxf.apache.org/schemas/simple.xsd">

    <import resource="classpath:META-INF/cxf/cxf.xml" />
    <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
    <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />

    <simple:server id="ProjectManager"
        serviceClass="demo.cxf.webservices.ProjectManager"
        address="/ProjectManager">
        <simple:serviceBean>
            <bean class="demo.cxf.webservices.ProjectManagerImpl" />
        </simple:serviceBean>
        <simple:dataBinding>
            <bean
                class="org.apache.cxf.aegis.databinding.AegisDatabinding" />
        </simple:dataBinding>

    </simple:server>
</beans>
```

把服务端程序发送到 Tomcat 中。

当 Servlet 容器启动，调用 web.xml 进行初始化，若配置文件有 listener 选项，调用其内容，为 org.springframework.web.context.ContextLoaderListener，关于 Spring 的配置在这里就不做详细说明了。然后调用 server-bean.xml 文件，从 jaxws:endpoint 元素中了解到后台的 SEI 是 demo.cxf.webservices.ProjectManagerImpl，同时采用的数据绑定为 org.apache.cxf.aegis.databinding.AegisDatabinding。公开 Web Services 服务。

（2）客户端访问程序

采用 JAX-RS 规范的前端客户端，只需修改后台服务的位置即可，代码如下：

```
    ……
public class Client {
    ……
    public static void main(String[] args) throws Exception {

        System.out.println("Starting Client");

        //定义 ClientProxy 的代理类
        ClientProxyFactoryBean factory = new ClientProxyFactoryBean();
```

```
            //设置 JaxWs 代理工厂的属性，包括服务类和地址
            factory.setServiceClass(ProjectManager.class);
factory.setAddress("http://localhost:18080/databing_aegis_servlet/ProjectManager");
factory.getServiceFactory().setDataBinding(new AegisDatabinding());

//获取服务类的对象并处理
ProjectManager projectManager = (ProjectManager) factory.create();
            ……
    }
    ……
}
```

注意：笔者的 Tomcat 采用的是 18080 端口。

4. 测试和运行结果

运行客户端程序，其客户端程序输出结果与第 6 章“采用 POJO 并基于 JAX-WS 服务的前端实现”案例相同。

5. 其他内容

上述的所有调用都基于 JAX-WS 规范。

本例代码请参见配套资源代码中的“第 07 章\databing_aegis_servlet”文件夹。

7.4 MTOM 使用

本节主要介绍 MTOM 的基础知识，然后说明 Apache CXF 框架如何实现 MTOM 方式，最后通过两个例子来进行进一步说明。

7.4.1 MTOM 简介

MTOM（Message Transmission Optimization Mechanism），中文含义是消息传输优化机制，这是 W3C 的消息传输优化机制规范，即通过 Web Services 方法来有效地发送二进制数据。一般情况下消息传输优化机制 （MTOM）消息编码与 WSHttpBinding 同时使用。换句话，MTOM 是一种机制，用原始字节的形式传输包含 SOAP 消息并且文件内容较大的二进制附件，这样可以使所传输的消息较小。

所以说，MTOM 是一个标准，该标准能方便和有效地给 Web Services 应用提供二进制数据传输。很多框架都支持 MTOM，如 Axis2、JAX-WS RI、JBoss WS、XFire、Microsoft's WCF，等等。

为了简单性和通用性，XML 被设计成以文本格式来表示数据。使用 SOAP 进行传递的数据首先被序列化，也就是将数据转换成字符串在 XML 文档中传送。在目的地，字符串再被反序列化，即再被转换成表示原来的值的数据类型。把二进制数据放入 SOAP 报文的最简单的方法，就是使用 Base64 编码的方式对其进行编码，以实现数据的序列化和反序列化。假如二进制文件是 XML 文档的一部分，这需要 Base64 编码。Base64 是一种很常见的编码规范，其作用是将二进制序列转换为可读的 ASCII 字符序列，常用在需用通过文本协议传输二进制数据的情况下，如 HTTP 和 SMTP 协议。Base64 编码基本原理是把三个 8bit 的字节转换为四个 6bit 的字节，然后

为 6bit 再添加两位高位 0，组成四个 8bit 的字节，不满四个字节的以‘=’填充。因为 Base64 将输入的数据编码成只含有 {'A'-'Z', 'a'-'z', '0'-'9', '+', '/'} 这 64 个字符的字符串，所以称之为 Base64 编码。可以看出，转换后的字符串理论上要比原来的长 1/3。在 W3C 的 XML Schema Part 2: Datatypes 规范中，定义了 base64Binary 类型，SOAP 报文中使用 Base64 编码的二进制内容可以定义为该类型。

当 MTOM 作为一种服务启用时，MTOM 携带二进制数据并且创建一个附件。

启用 MTOM 是一个相当简单的过程。首先，必须标注 schema 模式类型或 POJO，也可以让 JAXB 知道某一领域可以优化成 MTOM。其次，需要告诉 Apache CXF 框架希望启用 MTOM。

MTOM 不但被 JAXB 支持，同时也被 Aegis 支持。

7.4.2 CXF 实现 MTOM 的方式

一般情况下，实现 MTOM 有以下几个步骤。

（1）配置文件。

（2）注释 POJO。

（3）启动 MTOM 属性。

（4）调用 DataHandler 对象实现。

现在我们来分别介绍这些步骤。假设应用程序有这样一个图片模式类型：

```
<schema targetNamespace="http://pictures.com"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <element name="Picture">
    <complexType>
      <sequence>
        <element name="Title" type="xsd:string"/>
        <element name="ImageData" type="xsd:base64Binary"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

在上述情况下，开发者希望 ImageData 元素能作为附件来进行数据传送。要做到这一点，只需要添加一个 xmime:expectedContentTypes 注释即可，配置文件的代码如下所示：

```
<schema targetNamespace="http://pictures.com"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xmime="http://www.w3.org/2005/05/xmlmime">
  <element name="Picture">
    <complexType>
      <sequence>
        <element name="Title" type="xsd:string"/>
        <element name="ImageData" type="xsd:base64Binary"
          xmime:expectedContentTypes="application/octet-stream"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

这等于告诉 Apache CXF 框架，ImageData 区域的数据不仅是为 base64Binary 元素创建一个

byte []数组，也可以是任何内容类型。系统将创建一个 DataHandler，而不仅只用于流数据。

在基于代码优先的编程模式下，需要在 POJO 上添加注释表明 imageData 字段是一个 MTOM 的内容。其程序代码如下所示：

```
@XmlType
public class Picture {
  private String title;

  @XmlMimeType("application/octet-stream")
  private DataHandler imageData;

  public String getTitle() { return title; }
  public void setTitle(String title) { this.title = title; }

  public DataHandler getImageData() { return imageData; }
  public void setImageData(DataHandler imageData) { this.imageData = imageData; }
}
```

如果使用 JAX -WS 来发布服务端点，可以按照下列代码来启用 MTOM：

```
import javax.xml.ws.Endpoint;
import javax.xml.ws.soap.SOAPBinding;

Endpoint ep = Endpoint.publish("http://localhost/myService", new MyService());
SOAPBinding binding = (SOAPBinding) ep.getBinding();
binding.setMTOMEnabled(true);
```

如果使用 XML 来发布服务端点，配置代码如下：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://cxf.apache.org/jaxws
http://cxf.apache.org/schema/jaxws.xsd">

  <jaxws:endpoint  id="helloWorld"  implementor="demo.spring.HelloWorldImpl"
      address="http://localhost/HelloWorld">
    <jaxws:properties> <entry key="mtom-enabled" value="true"/> </jaxws:properties>
  </jaxws:endpoint>

</beans>
```

如果在简化前端模式下编写程序代码，可以在代码中设置 ServerFactoryBean 或 ClientProxyFactoryBean 的 MTOM 功能的属性，代码如下：

```
Map<String,Object> props = new HashMap<String, Object>();
props.put("mtom-enabled", Boolean.TRUE); // Boolean.TRUE or "true" will work as the
property value here

ClientProxyFactoryBean pf = new ClientProxyFactoryBean();
pf.setPropertyies(props);
....
YourClient client = (YourClient) pf.create();
```

```
ServerFactoryBean sf = new ServerFactoryBean();
sf.setPropertyies(props);
...
sf.create();
```

同样，简化前端模式也可以使用 XML 配置，其服务器配置文件内容如下：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:simple="http://cxf.apache.org/simple"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://cxf.apache.org/simple http://cxf.apache.org/schema/simple.xsd">

  <simple:server
    id="helloWorld"
    serviceClass="demo.spring.HelloWorldImpl"
    address="http://localhost/HelloWorld">
    <simple:properties>
      <entry key="mtom-enabled" value="true"/>
    </simple:properties>
  </simple:server>
</beans>
```

其客户端配置文件如下：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:simple="http://cxf.apache.org/simple"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://cxf.apache.org/simple http://cxf.apache.org/schema/simple.xsd">

  <simple:client
    id="helloWorldClient"
    serviceClass="demo.spring.HelloWorldImpl"
    address="http://localhost/HelloWorld">
    <simple:properties>
      <entry key="mtom-enabled" value="true"/>
    </simple:properties>
  </simple:client>
</beans>
```

一旦完成了上面的配置，就可以开始编写业务逻辑了。可以非常方便地创造 DataHandler 对象，其创建代码如下：

```
Picture picture = ...;
DataHandler handler = picture.getImageData();
InputStream is = handler.getInputStream();
```

还有许多其他方法可以创建 DataHandler。可以使用 FileDataSource、ByteArrayDataSource 等数据源来创建，也可以使用开发人员自己编写的数据源来创建：

```
DataSource source = new ByteArrayDataSource(new byte[] {...}, "content/type");
```

```
DataSource source = new FileDataSource(new File("my/file"));

Picture picture = new Picture();
picture.setImageData(new DataHandler(source));
```

7.4.3　CXF 实现 MTOM 的例子

本程序演示 Apache CXF 框架采用 MTOM 数据绑定来实现客户端和服务器端的相互调用，客户端采用 Apache CXF 的代码优先（Java First）前端应用模式。采用 POJO 对象作为中间传输对象。

1. 应用程序案例场景说明和图示

应用场景：首先是要有一个图片文件，通过 Web Services 把客户端图片发送到服务器。再通过 Web Services 获得服务器的图片。这样可以实现通过 Web Services 来进行二进制数据的传递。

在这个过程中，都是使用 POJO 对象实现客户端和服务器端的数据交换。后台服务端为遵循 JAX-WS 规范的 Endpoint 来实现。前端访问 JAX-WS 规范的 Service 方式，实现的结构如图 7-9 所示。

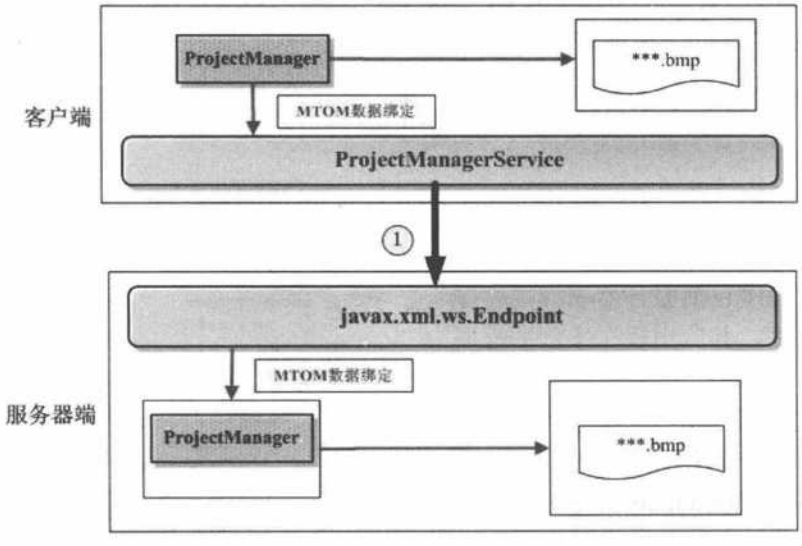


图 7-9　JAX-WS 规范简化前端 MTOM 数据绑定实现结构图

2. 应用程序案例实现步骤

要创建 MTOM 数据绑定的前端实现 Web Services 程序的流程步骤如图 7-10 所示。

流程图说明，需要实现下面的步骤：

（1）创建一个 Java 项目。

（2）创建一个 SEI 及其相关对象和注释，增加类型处理程序。

（3）编写发布这个 Java 程序为 Web Services 的服务端程序。

（4）编写 Web Services 的客户端应用程序。

（5）启动服务器程序，对外发布 Web Services。

（6）客户端调用 Web Services。

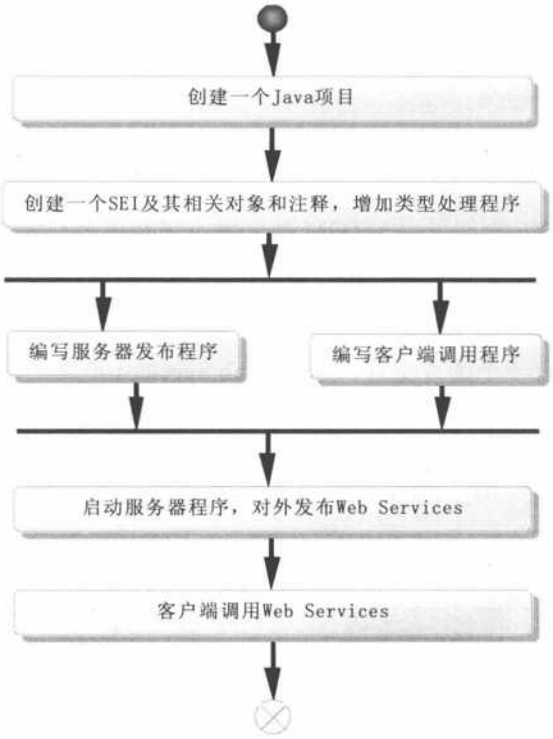


图 7-10　应用程序案例实现步骤流程图

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 7-11 所示。

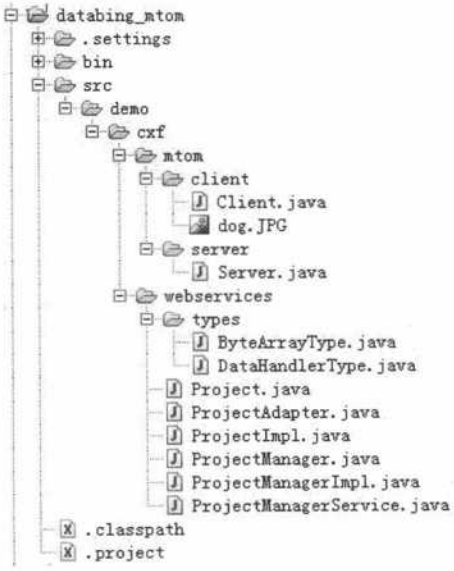


图 7-11　databing_mtom 程序结构

包目录包括三个部分，第一个包目录 demo.cxf.mtom.server 是服务器程序目录；第二个包目录 demo.cxf.mtom.client 是客户端程序目录；第三个包目录 demo.cxf. webservices 是 SEI 和相关对象等公用部分，客户端和服务器端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口，服务器端和客户端都必须引用。但服务器端和客户端对公用部分的引用是不同的。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口，而服务器端程序对于接口和实现类都要引用。

与第 6 章“采用 POJO 并基于 JAX-WS 服务的前端实现”案例相比，只有客户端和服务端的数据绑定方式不同，其余的内容全部相同。第 6 章“采用 POJO 并基于 JAX-WS 服务的前端实现”案例采用的是默认的 JAXB 数据绑定，而

本程序采用的是 MTOM 数据绑定，下面只列出有区别的三部分：①服务器端点接口 SEI 和相关对象；②服务器端发布程序；③客户端调用程序。

（1）服务器端点接口（SEI）和相关对象

首先要介绍的是服务器端点接口（SEI）。

创建服务器端点接口（SEI）。该接口名称为 ProjectManager，其 Java 代码如下：

```
package demo.cxf.webservices;

import javax.activation.DataHandler;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.WebParam;
import javax.xml.ws.Holder;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.ResponseWrapper;

@WebService
public interface ProjectManager {
    ……
    @ResponseWrapper(localName = "doByteArrayResponse",
targetNamespace = "http://webservices.cxf.demo/types",
className = "demo.cxf.webservices.types.ByteArrayType")
    @RequestWrapper(localName = "doByteArray",
targetNamespace = "http://webservices.cxf.demo/types",
className = "demo.cxf.webservices.types.ByteArrayType")
    @WebMethod
    public void doByteArray(
            @WebParam(mode = WebParam.Mode.INOUT, name = "name",
targetNamespace = "http://webservices.cxf.demo/types")
            Holder<java.lang.String> name,
            @WebParam(mode = WebParam.Mode.INOUT, name = "attachinfo",
targetNamespace = "http://webservices.cxf.demo/types")
            Holder<byte[]> attachinfo);

    @ResponseWrapper(localName = "doDataHandlerResponse",
TargetNamespace = "http://webservices.cxf.demo/types",
className = "demo.cxf.webservices.types.DataHandlerType")
    @RequestWrapper(localName = "doDataHandler",
targetNamespace = "http://webservices.cxf.demo/types",
className = "demo.cxf.webservices.types.DataHandlerType")
    @WebMethod
    public void doDataHandler(
            @WebParam(mode = WebParam.Mode.INOUT, name = "name",
targetNamespace = "http://webservices.cxf.demo/types")
            Holder<java.lang.String> name,
            @WebParam(mode = WebParam.Mode.INOUT, name = "attachinfo",
targetNamespace = "http://webservices.cxf.demo/types")
            Holder<DataHandler> attachinfo);

}
```

在上述的程序代码中，注释信息表明了这需要 MTOM 的支持。

实现这个接口的 Java 实现类 ProjectManagerImpl 的代码如下：

```
package demo.cxf.webservices;
```

```
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.HashMap;

import javax.activation.DataHandler;
import javax.jws.WebService;
import javax.mail.util.ByteArrayDataSource;
import javax.xml.ws.Holder;

@WebService(serviceName = "ProjectManagerService",
        portName = "ProjectManagerPort",
        endpointInterface = "demo.cxf.webservices.ProjectManager",
        targetNamespace = "http://webservices.cxf.demo/")

public class ProjectManagerImpl implements ProjectManager {

    ……

    public void doByteArray(Holder<String> name, Holder<byte[]> attachinfo) {
        System.out.println("从客户端获取图像");
        System.out.println("图形数据大小是 " + attachinfo.value.length);
        name.value = "Hello " + name.value;
    }

    public void doDataHandler(Holder<String> name,
            Holder<DataHandler> attachinfo) {
        try {
            System.out.println("通过 mtom 方式从客户端获取图像");
            InputStream mtomIn = attachinfo.value.getInputStream();
            ByteArrayOutputStream out = new ByteArrayOutputStream();
            copy(mtomIn, out);
            System.out.println("图形数据大小是 " + out.size());
            name.value = "你好 " + name.value;
            mtomIn.close();
            attachinfo.value = new DataHandler(new ByteArrayDataSource(out
                    .toByteArray(), attachinfo.value.getContentType()));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static int copy(final InputStream input, final OutputStream output)
            throws IOException {
        final byte[] buffer = new byte[4096];
        int n = 0;
        n = input.read(buffer);
        int total = 0;
        while (-1 != n) {
            output.write(buffer, 0, n);
            total += n;
            n = input.read(buffer);
        }
        return total;
    }
}
```

实现类 ProjectManagerImpl 首先要标识一个 Web Services 服务类。然后才实现接口的所有方法。在 ProjectManagerImpl 中，增加了两种数据类型，要把这两种数据类型通知到 Apache CXF 框架，所以还要添加两个类。

首先是处理字节数组的新类型。其实现代码如下：

```
package demo.cxf.webservices.types;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "ByteArrayType", propOrder = {"name", "attachinfo"})
public class ByteArrayType {

    @XmlElement(required = true)
    protected String name;
    @XmlElement(required = true)
    protected byte[] attachinfo;

    public String getName() {return name; }
    public void setName(String value) {this.name = value; }
    public byte[] getAttachinfo() {return attachinfo; }
    public void setAttachinfo(byte[] value) { this.attachinfo = ((byte[]) value); }
}
```

其次是处理 DataHandler 的新类型。其实现代码如下：

```
package demo.cxf.webservices.types;

import javax.activation.DataHandler;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlMimeType;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "DataHandlerType", propOrder = { "name",  "attachinfo"})
public class DataHandlerType {
    @XmlElement(required = true)
    protected String name;
    @XmlElement(required = true)
    @XmlMimeType("application/octet-stream")
    protected DataHandler attachinfo;
    public String getName() { return name; }
    public void setName(String value) {this.name = value; }
    public DataHandler getAttachinfo() { return attachinfo; }
    public void setAttachinfo(DataHandler value) {this.attachinfo = value; }
}
```

由于采用 ProjectManagerService 来创建服务，所以 ProjectManagerService 代码如下：

```
package demo.cxf.webservices;
```

```
import java.net.MalformedURLException;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.WebEndpoint;
import javax.xml.ws.WebServiceClient;
import javax.xml.ws.WebServiceFeature;
import javax.xml.ws.Service;

@WebServiceClient(name = "ProjectManagerService",
                  targetNamespace = "http://webservices.cxf.demo/")
public class ProjectManagerService extends Service {
   public final static URL WSDL_LOCATION;
public final static QName ProjectManagerSERVICE =
new QName("http://webservices.cxf.demo/", "ProjectManagerService");
public final static QName ProjectManagerPort =
new QName("http://webservices.cxf.demo/", "ProjectManagerPort");
   static {
      URL url = null;
      try {
         //url = new URL("file:/D:/apache-cxf-2.2.10/samples/mtom/wsdl/mtom_xop.wsdl");
       url = new URL("");
      } catch (MalformedURLException e) {
         System.err.println("Can not initialize the default wsdl");
         // e.printStackTrace();
      }
      WSDL_LOCATION = url;
   }

   public ProjectManagerService(URL wsdlLocation) {super(wsdlLocation, ProjectManagerSERVICE); }

   public ProjectManagerService(URL wsdlLocation, QName serviceName) {
      super(wsdlLocation, serviceName);
   }

   public ProjectManagerService() {super(WSDL_LOCATION, ProjectManagerSERVICE); }

   @WebEndpoint(name = "ProjectManagerPort")
   public ProjectManager getTestMtomPort() {
      return super.getPort(ProjectManagerPort, ProjectManager.class);
   }

   @WebEndpoint(name = "ProjectManagerPort")
   public ProjectManager getTestMtomPort(WebServiceFeature... features) {
      return super.getPort(ProjectManagerPort, ProjectManager.class, features);
   }

}
```

（2）服务器端发布程序

编写发布这个 Java 程序为 Web Services 的服务端程序。创建 SEI，要把这个 SEI 发布出去，还需要一个部署程序，即服务器端的发布程序，本案例采用 JAX-WS 规范来编写服务器端程序。创建一个 Server 类，启动这个服务，接着就可以发布 SEI 服务。

```
package demo.cxf.mtom.server;

import javax.xml.ws.Binding;
import javax.xml.ws.Endpoint;
```

```
import javax.xml.ws.soap.SOAPBinding;

import demo.cxf.webservices.*;

public class Server {

    protected Server() throws Exception {
        System.out.println("Starting Server");
        ProjectManager implementor = new ProjectManagerImpl();
        String address = "http://localhost:9000/ProjectManager";
        Endpoint ep = Endpoint.publish(address, implementor);
        Binding binding = ep.getBinding();
        ((SOAPBinding) binding).setMTOMEnabled(true);
    }

    public static void main(String[] args) throws Exception {
        new Server();
        System.out.println("Server ready...");

        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

其中黑体字表明加载 MTOM 附件。

在本案例中，服务器启动的服务地址是 http://localhost:9000/ProjectManager。服务器引入了 SEI 接口——ProjectManager 接口。这样 ProjectManager 的 Web Services 就暴露到了外部。可以在浏览器上输入 http://localhost:9000/ProjectManager 来查看这个服务的内容。

（3）客户端调用程序

编写 Web Services 的客户端应用程序。客户端调用服务的实现模式，其代码如下：

```
package demo.cxf.mtom.client;

import java.awt.Image;
import java.io.ByteArrayInputStream;
import java.io.File;
import java.io.InputStream;
import java.net.URI;
import java.net.URL;

import javax.activation.DataHandler;
import javax.imageio.ImageIO;
import javax.xml.namespace.QName;
import javax.xml.ws.Binding;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.Holder;
import javax.xml.ws.soap.SOAPBinding;

import demo.cxf.webservices.ProjectManagerService;
import demo.cxf.webservices.ProjectManager;

public class Client {
    public static final String WSDLFile = "http://localhost:9000/ProjectManager?wsdl";

    private static final QName SERVICE_NAME = new QName(
```

```
            "http://webservices.cxf.demo/", "ProjectManagerService");

    private static final QName PORT_NAME = new QName(
            "http://webservices.cxf.demo/", "ProjectManagerPort");

    public Client() {}

    public static void main(String[] args) throws Exception {
        URL wsdlURL;
        if (args.length == 0) {
            wsdlURL = new URL(WSDLFile);
        } else {
            File wsdlFile = new File(args[0]);
            if (wsdlFile.exists()) {
                wsdlURL = wsdlFile.toURL();
            } else {
                wsdlURL = new URL(args[0]);
            }
        }
        System.out.println(wsdlURL);

        ProjectManagerService pmService = new ProjectManagerService(wsdlURL, SERVICE_
NAME);
        ProjectManager port = (ProjectManager) pmService.getPort(PORT_NAME,ProjectManager.
class);
        Binding binding = ((BindingProvider) port).getBinding();
        ((SOAPBinding) binding).setMTOMEnabled(true);

        URL fileURL = Client.class.getResource("dog.JPG");
        File aFile = new File(new URI(fileURL.toString()));
        long fileSize = aFile.length();
        System.out.println("dog.JPG 图像文件大小是: " + fileSize);

        System.out.println("\n 开始使用 basic byte array 来进行 MTOM 测试 :");
        Holder<String> name = new Holder<String>("测试数组");
        Holder<byte[]> param = new Holder<byte[]>();
        param.value = new byte[(int) fileSize];
        InputStream in = fileURL.openStream();
        int len = in.read(param.value);
        while (len < fileSize) {
            len += in.read(param.value, len, (int) (fileSize - len));
        }
        System.out.println("--发送 dog.JPG 图形到服务器");
        System.out.println("--发送的名称是 " + name.value);

        port.doByteArray(name, param);

        System.out.println("--从服务器接受字节，返回的字节大小是 "+ param.value.length);
        System.out.println("--返回的字符名称是 " + name.value);

        Image image = ImageIO.read(new ByteArrayInputStream(param.value));
        System.out.println("--成功地通过字节数组下载图片，哈希码是"+ image.hashCode());
        System.out.println("成功地运行 MTOM/byte array 演示.");

        System.out.println("\n 开始使用 DataHandler 来进行 MTOM 测试:");
        name.value = "Dog";
        Holder<DataHandler> handler = new Holder<DataHandler>();
```

```
        handler.value = new DataHandler(fileURL);

        System.out.println("--发送 dog.JPG 图片到服务器");
        System.out.println("--发送的字符名称是 " + name.value);

        port.doDataHandler(name, handler);

        InputStream mtomIn = handler.value.getInputStream();
        fileSize = 0;
        for (int i = mtomIn.read(); i != -1; i = mtomIn.read()) {
            fileSize++;
        }
        System.out.println("--通过 DataHandler 方式从服务器接受文件，"+ "返回的文件大小是 "
+ fileSize);
        System.out.println("--返回的字符名称是 " + name.value);

        System.out.println("成功地运行 MTOM/DataHandler 演示.");
        System.exit(0);
    }

}
```

4. 测试和运行结果

运行客户端程序，其输出结果如下：

```
dog.JPG 图像文件大小是: 484323

开始使用 basic byte array 来进行 MTOM 测试 :
--发送 dog.JPG 图形到服务器
--发送的名称是 测试数组
--从服务器接受字节，返回的字节大小是 484323
--返回的字符名称是 Hello 测试数组
--成功地通过字节数组下载图片，哈希码是 6417481
成功地运行 MTOM/byte array 演示.

开始使用 DataHandler 来进行 MTOM 测试:
--发送 dog.JPG 图片到服务器
--发送的字符名称是 Dog
--通过 DataHandler 方式从服务器接受文件，返回的文件大小是 484323
--返回的字符名称是 你好 Dog
成功地运行 MTOM/DataHandler 演示.
```

5. 其他内容

上述的所有调用都是基于 JAX-WS 规范。

本例代码参见配套资源代码中“第 07 章\databing_mtom”文件夹。

7.4.4 CXF 实现 MTOM 的 Servlet 例子

本程序演示 Apache CXF 采用 MTOM 数据绑定来实现客户端和服务器端的相互调用，客户端采用 Apache CXF 的代码优先(Java First)前端应用模式。服务器端采用 Servlet 容器。采用 POJO 对象作为中间传输对象。本例 Servlet 容器采用 Tomcat 6。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的大文本附件数据。

在这个过程中，都使用 POJO 对象实现客户端和服务器端的数据交换。后台服务器端为遵循 JAX-WS 规范的 Server 来实现。前端访问采用 JAX-WS 规范的 Service 方式，实现的结构如图 7-12 所示。

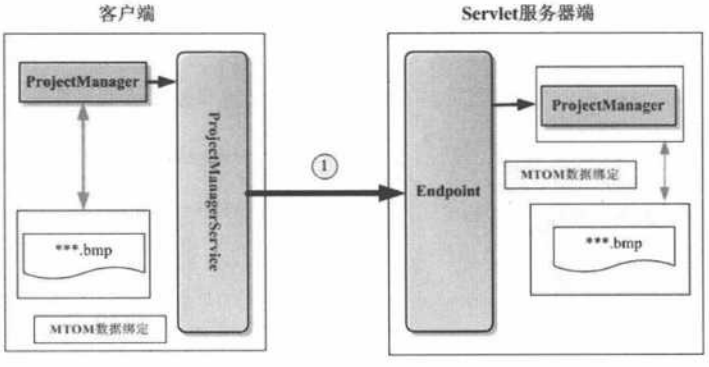


图 7-12　JAX-WS 规范简化前端 MTOM 数据绑定 Servlet 实现结构图

2. 应用程序案例实现步骤

要创建 MTOM 数据绑定并基于 Servlet 容器的前端实现 Web Services 程序的流程步骤如图 7-13 所示。

流程图说明，要实现以下步骤：

（1）创建一个 Web 服务项目。

（2）创建一个 SEI 及其相关对象和注释，增加类型处理程序。

（3）编写发布这个 Java 程序为 Web Services 的服务端程序。

（4）编写 web.xml 和 server-bean.xml。

（5）发布配置文件到 Servlet 服务器。

（6）启动 Servlet 服务器程序，对外发布 Web Services。

（7）客户端调用 Web Services。

其中第 1 步和第 3 步与采用自带的 HTTP 服务器类似。主要是第 2 步，该步骤不是采用自带的服务，而是放置到 Servlet 容器中。我们采用与 Spring 集成方式来实现。

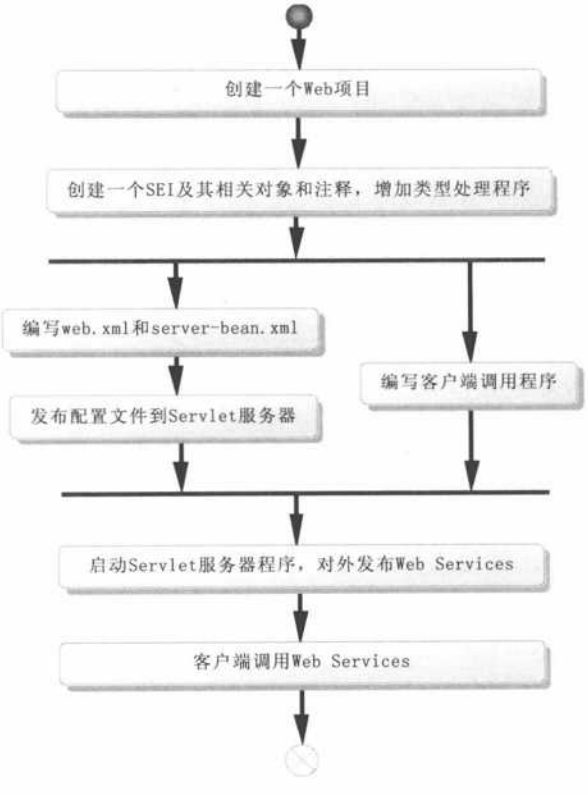


图 7-13　应用程序案例实现步骤流程图

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的目录结构如图 7-14 所示。

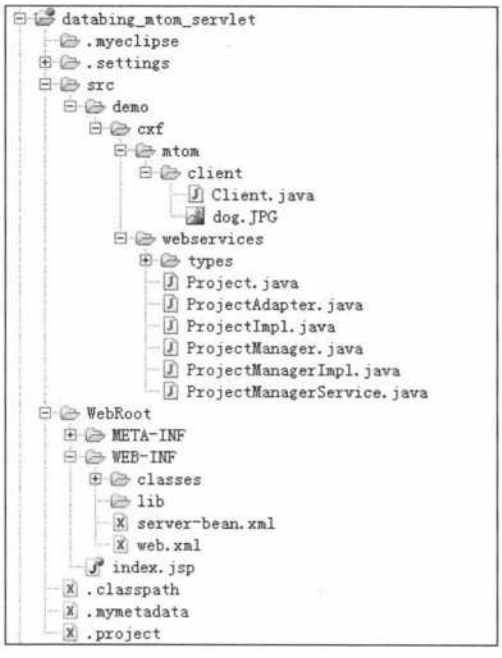


图 7-14　databing_mtom_servlet 程序结构

包目录包括三个部分，第一个包目录 WebRoot 目录是服务器程序目录，包括 web.xml 和 server-bean.xml 文件；第二个包目录 demo.cxf.mtom.client 是客户端程序目录；第三个包目录 demo.cxf.webservices 是 SEI 和数据对象等公用部分，客户端和服务器端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口。由于本程序直接发布到 Servlet 容器，所以没有单独编写服务器发布程序，直接把配置文件和编译好的服务器端点接口（SEI）组件等部署到 Servlet 容器即可。

下面分别介绍两部分：①服务器端配置文件；②客户端调用程序。

（1）服务器端配置文件

服务器端配置文件包括 web.xml 文件和 server-bean.xml 文件。

web.xml 文件的定义与第 6 章“采用 POJO 并基于 Servlet 容器的前端实现”案例相同，这里就不再列出了。其中，当 Servlet 启动，将读取 WEB-INF 目录下的 server-bean.xml 文件。server-bean.xml 文件的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="
          http://cxf.apache.org/transports/http/configuration
          http://cxf.apache.org/schemas/configuration/http-conf.xsd
```

```
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
          http://cxf.apache.org/jaxws
          http://cxf.apache.org/schemas/jaxws.xsd">

    <import resource="classpath:META-INF/cxf/cxf.xml" />
    <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
    <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />
    <jaxws:endpoint id="ProjectManager"
        implementor="demo.cxf.webservices.ProjectManagerImpl"
        address="/ProjectManager">
        <jaxws:properties>
            <entry key="mtom-enabled" value="true" />
        </jaxws:properties>
    </jaxws:endpoint>
</beans>
```

把服务端程序发送到 Tomcat 中。

当 Servlet 容器启动，调用 web.xml 进行初始化，若有 listener 选项，调用其内容，为 org.springframework.web.context.ContextLoaderListener，关于 Spring 的配置在这里就不做详细说明。然后就调用 server-bean.xml 文件，从 jaxws:endpoint 元素中了解到后台的 SEI 是 demo.cxf.webservices.ProjectManagerImpl，同时采用的数据绑定为 entry key="mtom-enabled" value="true"。公开 Web Services 服务。

（2）客户端调用程序

采用 JAX-RS 规范的前端客户端，只需修改后台服务的位置即可，代码如下：

```
    ......

public class Client {

    public static final String WSDLFile =
"http://localhost:18080/databing_mtom_servlet/ProjectManager?wsdl";
    private static final QName SERVICE_NAME = new QName(
            "http://webservices.cxf.demo/", "ProjectManagerService");
......
    public static void main(String[] args) throws Exception {
        ......
    }
}
```

注意：笔者的 Tomcat 采用的是 18080 端口。

4. 测试和运行结果

运行客户端程序，其输出结果与“CXF 实现 MTOM 例子”相同。

5. 其他内容

上述的所有调用都基于 JAX-WS 规范。

本例代码参见配套资源代码中“第 07 章\databing_mtom_servlet”文件夹。

7.5　XMLBeans 的使用

XMLBeans 是另一种 XML 模式映射到 Java 对象的技术。这是一种高级易用的 XML-Java 绑定技术，该技术允许开发人员像访问任何 Java 对象或 JavaBean 一样访问 XML 文件。Apache CXF 框架也支持对 XMLBeans 的使用。

7.5.1　XMLBeans 简介

XMLBean 以 XML Schema 为基础建立的、访问 XML 的一种解决方法，该技术提供了访问和处理 XML 数据和文档，具有既可以完全地访问 XML 内容、又不丢失 XML Schema 信息的强大功能。使用 XMLBeans 可以采用面向对象的观点来对待和处理 XML 数据与文档，同时又可以忠实于该 XML 数据对应的 XML 结构和 XML Schema。

XMLBeans 由 BEA 引入并且作为其 WebLogic Workshop 产品的一部分。2003 年 9 月，BEA 将 Page Flow、XMLBeans 和其他技术捐赠给开放源码社区。两个随之启动的项目是 Apache Beehive 和 Apache XMLBeans。

可以将 XMLBeans 看作对象关系映射器（ORM），但这是针对 XML 文件而非关系数据库而言的。XMLBeans 允许访问任何 Java 对象或 JavaBean，像访问 XML 文件一样。也就是说，XMLBeans 是一个 XML-Java 绑定工具。

XMLBeans 的优点主要体现在下面四点：

（1）XMLBeans 基于标记流，因此可以轻松地使用指针在 XML 数据和文档之间导航。指针接口适用于所有 XML 数据和文档。

（2）对于有 XML Schema 的 XML 数据和文档，XMLBeans 将生成这些 XML 数据和文档的 Java 类代码。

（3）开发者可以使用这些 Java 代码轻松地读/写 XML 数据和文档，并且被强制地执行 XML Schema 中规定的一些约束。

（4）生成的 Java 类代码都能够忠实地描述原始 XML 数据和文档的内容，因为 Java 类代码都是基于受保护的、最基本的 XML 表现。

所以使用 XMLBeans 来访问 XML 数据和文档的好处是显而易见的，这些好处包括：

（1）完全使用面向对象的观点来看待和处理数据与文档。

（2）开发者不再需要编写大量的代码来访问 XML 数据和文档。

（3）可以使用 XML Schema 中规定对于数据的约束条件，而不需要自己去编写实现这些约束的代码。

（4）访问其中的某个数据项时，不需要解析所有的 XML 数据和文档。

在底层，XML 文件始终与 bean 表示保持同步。如果具有符合某个 XML Schema 的文件，则可以使用 getter 和 setter 操作该文件，正如在 JavaBean 中所做的一样。XML 文档被当作一级数据对象对待（这意味着它们可以被无限制地使用），以类似 JavaBean 的方式被访问。

至少有两点是 XMLBeans 区别其他 XML-Java 绑定工具的：

（1）对 XML Schema 的全面支持。

（2）对 XML 文档中信息的忠实描述。

XMLBeans 提供了三个主要的 API。

- **XmlObject**：从 XML Schema 生成的 Java 类全都继承 XmlObject。这些生成类充当了强类型的所有要素和 XML 定义的二传手。复杂类型主要是 XmlObjects。例如，getCustomer 可能会返回一个 CustomerType（这是一个 XmlObject）。简单类型则转化为简单的 getter 和 setter 方法，返回正确的 Java 类型。例如，getName 可能会返回一个 String。
- **XmlCursor**：在任何 XmlObject 中，开发人员可以得到一个 XmlCursor 对象。这提供了高效、低成本的 XML Infoset 访问级别。游标表示 XML 实例的位置。开发者可以在任何级别的粒度——从个别到令牌字符，都需要实例化 XmlCursor 对象。
- **SchemaType**：XMLBeans 提供了一个完整的 XML Schema 对象模型，开发者可以利用 XMLBeans 来反映基础架构的元信息。例如，开发者可能要生成一个基于 XML Schema 的 XML 实例样品或者找到一个元素的枚举，以便在开发过程中可以显示这些变量。

上述的内容都基于性能方面。非正式的基准和用户反馈表明，XMLBeans 的确是非常快。

XMLBeans 的官方网站是：http://xmlbeans.apache.org/index.html。

7.5.2 CXF 实现 XMLBeans 的方式

Apache CXF 2.1 版本中增加了对 XMLBeans 的支持。

1. 采用 Spring 的配置方式

采用 Spring 编程，在服务器端，要对配置文件做如下的配置：

```
<jaxws:server serviceClass="demo.hw.server.HelloWorld" address="/hello_world">
   <jaxws:dataBinding>
      <bean class="org.apache.cxf.xmlbeans.XmlBeansDataBinding" />
   </jaxws:dataBinding>
</jaxws:server>
```

或采用下面的方式来进行配置：

```
<jaxws:endpoint id="helloWorld" implementor="demo.spring.HelloWorldImpl"
address="http://localhost/HelloWorld">
   <jaxws:dataBinding>
      <bean class="org.apache.cxf.xmlbeans.XmlBeansDataBinding" />
   </jaxws:dataBinding>
</jaxws:endpoint>
```

在客户端也要做一些定义，配置信息如下：

```
<jaxws:client id="helloClient" serviceClass="demo.spring.HelloWorld"
address="http://localhost:9002/HelloWorld">
   <jaxws:dataBinding>
      <bean class="org.apache.cxf.xmlbeans.XmlBeansDataBinding" />
   </jaxws:dataBinding>
<jaxws:client>
```

2. 采用 FactoryBeans 编程方式

如果不采用 Spring 配置方式而使用编程方式，可采用数据绑定的 ClientProxyFactoryBean（或者其子类）和 ServerFactoryBean（或者其子类），实现代码如下：

```
factory.getServiceFactory().setDataBinding(new
org.apache.cxf.xmlbeans.XmlBeansDataBinding());
```

7.5.3　实现简化前端 XMLBeans 数据绑定的例子

本程序演示 Apache CXF 框架采用 XMLBeans 数据绑定来实现客户端和服务器端的相互调用，客户端采用 Apache CXF 框架的代码优先（Java First）前端应用模式。在这种模式下，不能采用 POJO 对象作为中间传输对象。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询和删除操作。

在这个过程中，前端客户端采用 Apache CXF 框架自身规范的 ClientProxyFactoryBean 对象实现，后台服务器端为遵循 Apache CXF 框架自身规范的 ServerFactoryBean 对象来实现。采用 XMLBeans 数据绑定方式，实现的结构如图 7-15 所示。

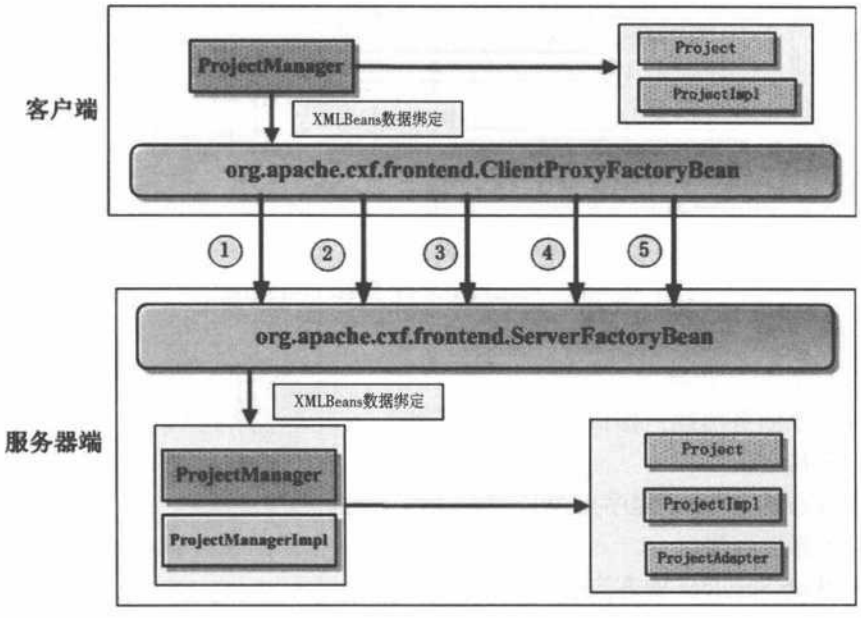


图 7-15　JAX-WS 规范简化前端 XMLBeans 数据绑定实现结构图

其中：①表示客户端通过 Service 获得 SEI，访问后台的项目基本情况；②表示客户端通过 SEI 修改默认项目；③表示客户端通过 SEI 增加一个项目；④表示客户端通过 SEI 删除一个项目。⑤表示客户端通过 SEI 查看后台的全部项目情况。在客户端和服务端的 SEI 都是 ProjectManager 接口。

在编写本程序例子时，主要针对第 6 章“采用 POJO 并基于 JAX-WS 服务的前端实现”案例进行比较。

2. 应用程序案例实现步骤

要创建 XMLBeans 数据绑定的前端实现 Web Services 程序的流程步骤如图 7-16 所示。

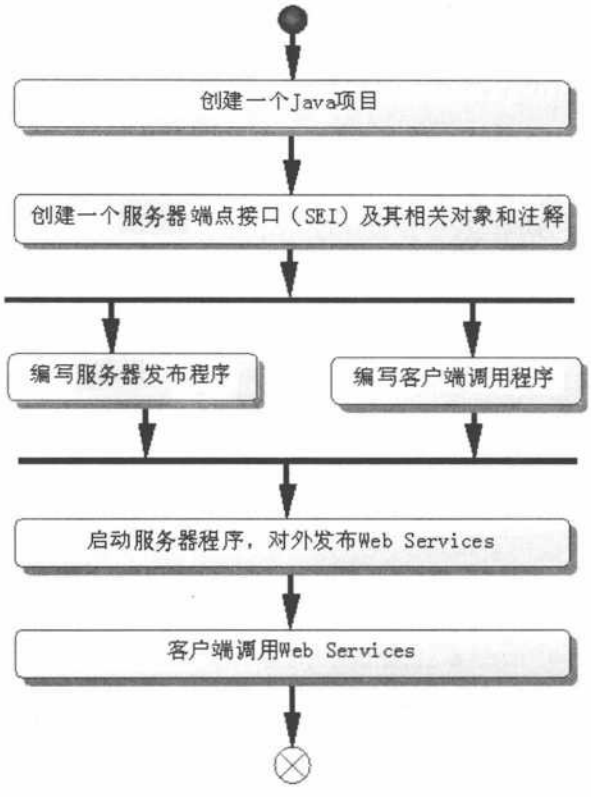


图 7-16　应用程序案例实现步骤流程图

流程图说明，需要实现以下步骤：

（1）创建一个 Java 项目。

（2）创建一个服务器端点接口（SEI）及其相关对象和注释。

（3）编写发布这个 Java 程序为 Web Services 的服务器端程序。

（4）编写 Web Services 的客户端应用程序。

（5）启动服务器程序，对外发布 Web Services。

（6）客户端调用 Web Services。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 7-17 所示。

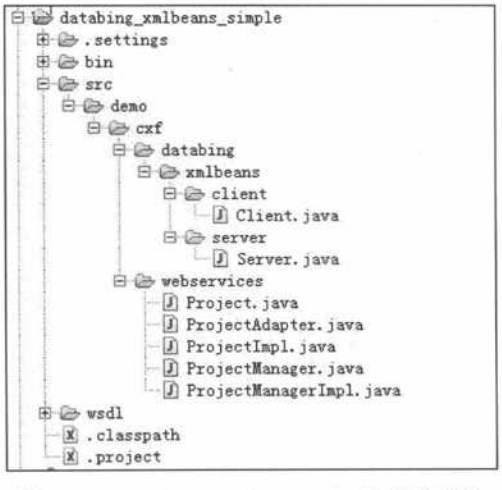


图 7-17　databing_xmlbeans_simple 程序结构

包目录包括三个部分，第一个包目录 demo.cxf.databing.xmlbeans.server 是服务器程序目录；第二个包目录 demo.cxf.databing.xmlbeans.client 是客户端程序目录；第三个包目录 demo.cxf.webservices 是 SEI 和相关对象等公用部分，客户端和服务器端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口，服务器端和客户端都必须引用。但服务器端和客户端对公用部分的引用是不同的。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口，而服务器端程序对于接口和实现类都要引用。

与第 6 章“采用 POJO 并基于 JAX-WS 服务的前端实现”案例相比，只有客户端和服务器端的数据绑定方式不同，其余的内容全部相同。第 6 章“采用 POJO 并基于 JAX-WS 服务的前端实现”案例采用默认的 JAXB 数据绑定，而本程序采用的是 XMLBeans 数据绑定，下面分别介绍有区别的三部分：①服务器端点接口 SEI 和相关对象；②服务器端发布程序；③客户端调用程序。

（1）服务器端点接口（SEI）

首先要介绍的是服务器端点接口（SEI），即 ProjectManager.java 文件。由于 XMLBeans 数据绑定不支持 POJO 对象，所以关于这方面的方法都删除，代码如下：

```
package demo.cxf.webservices;

import javax.jws.WebService;
import javax.jws.WebParam;

@WebService
public interface ProjectManager {
    public String greeting() ;
    public String greetingWithParameter(@WebParam(name = "greet")String greet);
    public String getProjectName();
    public String getProjectNameByID(String projectID);
    public void updateProjectName (@WebParam(name = "projectName")String projectName);
    public void removeProject(@WebParam(name = "id") String id);
    public void getAllProject();
}
```

（2）服务器发布程序

第一个文件是服务器发布程序，即 Server.java，代码如下：

```
package demo.cxf.databing.aegis.server;

import org.apache.cxf.aegis.databinding.AegisDatabinding;
import org.apache.cxf.frontend.ServerFactoryBean;
import demo.cxf.webservices.*;

public class Server {

    public Server() {
        System.out.println("Starting Server");
        ProjectManager projectManager = new ProjectManagerImpl();
        String address = "http://localhost:9000/ProjectManager";
        ServerFactoryBean svrFactory = new ServerFactoryBean();
        svrFactory.setServiceClass(ProjectManager.class);
        svrFactory.setAddress(address);
        svrFactory.setServiceBean(projectManager);
        // 设置数据的绑定模式为 XmlBeansDataBinding
        svrFactory.getServiceFactory().setDataBinding(new XmlBeansDataBinding());
        svrFactory.create();
    }

    public static void main(String[] args) throws Exception {
        new Server();
```

```
        System.out.println("Server ready...");

        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

其中黑体字说明了服务器采用 XMLBeans 数据绑定来发布程序。

（3）客户端调用程序

第二个文件是客户端访问程序，即 Client.java，代码如下：

```
package demo.cxf.databing.aegis.client;

import org.apache.cxf.aegis.databinding.AegisDatabinding;
import org.apache.cxf.frontend.ClientProxyFactoryBean;
import demo.cxf.webservices.Project;
import demo.cxf.webservices.ProjectImpl;
import demo.cxf.webservices.ProjectManager;

public class Client {
    public Client() {   }

    public static void main(String[] args) throws Exception {
        System.out.println("Starting Client");

        //定义 ClientProxy 的代理类
        ClientProxyFactoryBean factory = new ClientProxyFactoryBean();

        //设置 JaxWs 代理工厂的属性，包括服务类和地址
        factory.setServiceClass(ProjectManager.class);
        factory.setAddress("http://localhost:9000/ProjectManager");
        factory.getServiceFactory().setDataBinding(new XmlBeansDataBinding());

        //获取服务器类的对象并处理
        ProjectManager projectManager = (ProjectManager) factory.create();
        //System.out.println(projectManager.getProjectName());

        //项目的基本情况
        getGreet(projectManager);
        greetingWithParameter(projectManager);
        getProjectName(projectManager);
        getAllProject(projectManager);

        //删除一个项目，查看项目的基本情况
        removeProject(projectManager);
        System.exit(0);
    }

    //调用 SEI 的 getGreet 方法
    private static void getGreet(ProjectManager projectManager)throws Exception {
        System.out.println(projectManager.greeting());
    }

    ……
}
```

其中黑体字说明了客户端采用 XMLBeans 数据绑定来访问程序。相应删除了采用 POJO 作为

数据对象进行传递的。

4. 测试和运行结果

运行客户端程序，其客户端程序和服务器端输出结果与第 6 章“采用 POJO 并基于 JAX-WS 服务的前端实现”案例相同。

5. 其他内容

上述的所有调用都基于 JAX-WS 规范。

本例代码参见配套资源代码中“第 07 章\databing_xmlbeans_simple”文件夹。

7.5.4 采用简化前端 XMLBeans 数据绑定的 Servlet 例子实现

本程序演示 Apache CXF 采用 XMLBeans 数据绑定来实现客户端和服务器端的相互调用，客户端采用 Apache CXF 的代码优先（Java First）前端应用模式。服务器端采用 Servlet 容器。本例 Servlet 容器采用 Tomcat 6。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询和删除操作。

在这个过程中，前端客户端采用 Apache CXF 框架自身规范的 ClientProxyFactoryBean 对象实现，后台服务端为遵循 Apache CXF 框架自身规范的 ServerFactoryBean 对象来实现。实现的结构如图 7-18 所示。

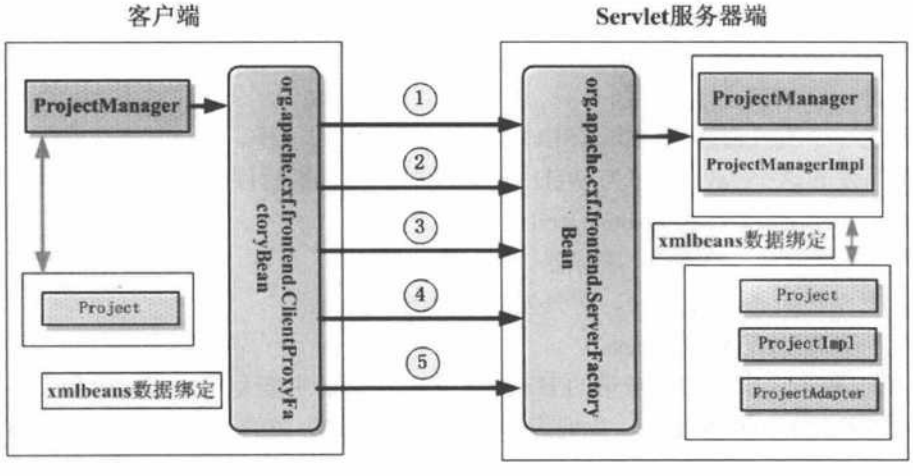


图 7-18　JAX-WS 规范简化前端 XMLBeans 数据绑定 Servlet 实现结构图

2. 应用程序案例实现步骤

要创建 XMLBeans 数据绑定并基于 Servlet 容器的前端实现 Web Services 程序的流程步骤如图 7-19 所示。

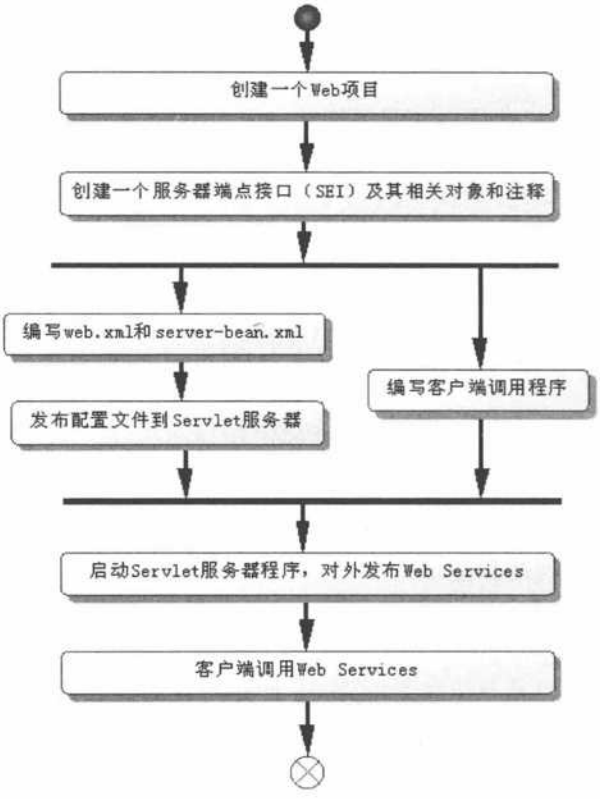


图 7-19　应用程序案例实现步骤流程图

流程图说明，需要实现以下步骤：

（1）创建一个 Web 服务项目。

（2）创建一个服务器端点接口（SEI）及其相关对象和注释。

（3）编写发布这个 Java 程序为 Web Services 的服务器端程序。

（4）编写 web.xml 和 server-bean.xml。

（5）发布配置文件到 Servlet 服务器。

（6）启动 Servlet 服务器程序，对外发布 Web Services。

（7）客户端调用 Web Services。

其中第 1 步和第 3 步与采用自带的 HTTP 服务器类似。主要是第 2 步，该步不是采用自带的服务，而是放置到 Servlet 容器中。我们采用与 Spring 框架集成的方式来实现。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 7-20 所示。

包目录包括三个部分，第一个包目录 WebRoot 目录是服务器程序目录，包括 web.xml 和 server-bean.xml 文件；第二个包目录 demo.cxf.databing.xmlbeans.client 是客户端程序目录；第三个包目录 demo.cxf.webservices 是 SEI 和数据对象等公用部分，客户端和服务器端都有可能引用其内容。

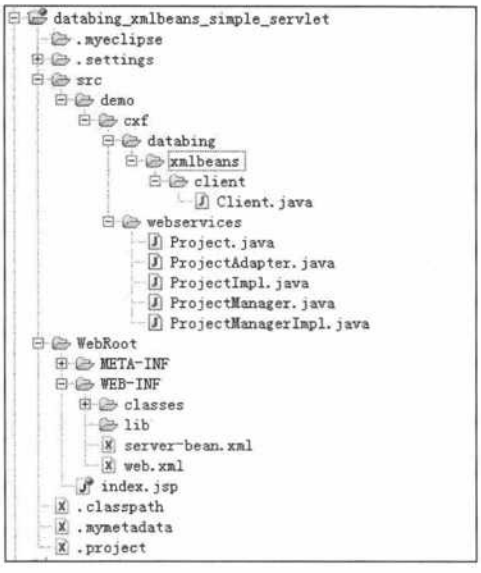


图 7-20　databing_xmlbeans_simple_servlet 程序结构

服务器端点接口（SEI）是公用部分的接口。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口。由于本程序直接发布到 Servlet 容器，所以没有单独编写服务器发布程序，直接把配置文件和编译好的服务器端点接口（SEI）组件等部署到 Servlet 容器即可。

下面分别介绍：①服务器端配置文件；②客户端调用程序。

（1）服务器端配置文件

服务器端配置文件包括 web.xml 文件和 server-bean.xml 文件。

web.xml 文件的定义与第 6 章“采用 POJO 并基于 Servlet 容器的前端实现”案例相同，这里就不再列出了。其中，当 Servlet 启动，将读取 WEB-INF 目录下的 server-bean.xml 文件，server-bean.xml 文件的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:simple="http://cxf.apache.org/simple"
    xmlns:soap="http://cxf.apache.org/bindings/soap"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
        http://cxf.apache.org/bindings/soap
        http://cxf.apache.org/schemas/configuration/soap.xsd
        http://cxf.apache.org/simple
        http://cxf.apache.org/schemas/simple.xsd">

    <import resource="classpath:META-INF/cxf/cxf.xml" />
    <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
    <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />

    <simple:server id="ProjectManager"
        serviceClass="demo.cxf.webservices.ProjectManager"
        address="/ProjectManager">
        <simple:serviceBean>
            <bean class="demo.cxf.webservices.ProjectManagerImpl" />
```

```
        </simple:serviceBean>
        <simple:dataBinding>
            <bean
                class=" org.apache.cxf.xmlbeans.XmlBeansDataBinding" />
        </simple:dataBinding>
    </simple:server>
</beans>
```

把服务器端程序发送到 Tomcat 中。

当 Servlet 容器启动，调用 web.xml 进行初始化，若有 listener 选项，调用其内容，为 org.springframework.web.context.ContextLoaderListener，关于 Spring 的配置在这里就不做详细说明了。然后调用 server-bean.xml 文件，从 jaxws:endpoint 元素中了解到后台的 SEI 是 demo.cxf.webservices.ProjectManagerImpl，同时采用的数据绑定为 org.apache.cxf.xmlbeans.XmlBeansDataBinding。公开 Web Services 服务。

（2）客户端调用程序

采用 JAX-WS 规范的前端客户端，只需修改后台服务的位置即可，代码如下：

```
      ……
public class Client {
      ……
       public static void main(String[] args) throws Exception {
           System.out.println("Starting Client");
           //定义 ClientProxy 的代理类
           ClientProxyFactoryBean factory = new ClientProxyFactoryBean();
           //设置 JaxWs 代理工厂的属性，包括服务器类和地址
           factory.setServiceClass(ProjectManager.class);
factory.setAddress("http://localhost:18080/databing_xmlbeans_simple_servlet/
ProjectManager");
factory.getServiceFactory().setDataBinding(new AegisDatabinding());

//获取服务类的对象并处理
ProjectManager projectManager = (ProjectManager) factory.create();
           ……
    }
    ……
}
```

注意：笔者的 Tomcat 采用的是 18080 端口。

4. 测试和运行结果

运行客户端程序，其客户端程序输出结果与第 6 章“采用 POJO 并基于 JAX-WS 服务的前端实现”案例相同。

5. 其他内容

上述的所有调用基于 JAX-WS 规范。

本例代码参见配套资源代码中“第 07 章\databing_xmlbeans_simple_servlet”文件夹。

第8章 CXF的传输

Apache CXF 支持多种传输协议，包括 HTTP 协议、HTTPS 协议、JMS 协议和本地协议。本章主要介绍 Apache CXF 对 HTTP、JMS 协议和本地协议的内容与实现。针对每个传输协议，首先是简介该传输协议内容，其次说明 Apache CXF 框架是如何支持该传输协议的，最后用示例来进行说明。关于 Apache CXF 对 HTTPS 协议的支持将在后面章节中介绍。

本章内容：

1. Apache CXF 支持的传输协议
2. HTTP 传输协议
3. JMS 传输协议
4. 本地传输协议

8.1 CXF 支持的传输协议

Apache CXF 框架支持下列传输协议：

- HTTP 协议
- HTTPS 协议
- JMS 协议
- 本地协议

HTTP、HTTPS 和 JMS 是 TCP 的远程路由协议，而本地协议主要在 JVM 内部进行信息传递。传输主要是信息的路由。

在 Apache CXF 框架中，通过特定的传输协议，Web Services 消息在不同的服务器端点之间路由。传输时所提供的细节定义一个端点。一个端点是一个服务的物理表现。简单地说，这是一个实例化的服务。端点的定义是具有约束力的细节和传输的细节。传输细节通常被称为网络细节。终结点是在 WSDL 协议<wsdl:service>元素定义的一部分。

8.2 HTTP 传输协议

HTTP 是一个标准的网络传输协议。HTTP 传输协议被广泛地应用在 Web Services，它已经成为最常用的和最标准的服务端点通信通道。本节先介绍一下 Apache CXF 是如何实现 HTTP 传输

协议的。最后通过一个例子程序来进行进一步说明。

8.2.1 CXF 支持 HTTP 传输协议介绍

Apache CXF 对 HTTP 传输协议的支持有两种方式。第一种是 HTTP 协议上面的 SOAP 协议，第二种就是简单的 HTTP 协议。

先对 SOAP 协议做一个简单介绍。SOAP 协议包括 SOAP 1.1 协议和 SOAP 1.2 协议。

1. SOAP 1.1 协议

可以通过<soap:binding> 元素来定义 SOAP 1.1 协议。<soap:binding>元素是<wsdl:binding>元素的子元素。当使用<soap:binding>表示，标志着这项服务是绑定到 SOAP 版本 1.1 协议，即该消息将遵循 SOAP 1.1 的格式。该<soap:binding>元素中的传输属性可以定义一个指定的传输协议。在这种情况下，它将会使用 HTTP。其 URI 的表现形式：

```
http://schemas.xmlsoap.org/soap/http
```

下面的 WSDL 代码说明 SOAP 1.1 与 HTTP 传输绑定：

```
<wsdl:binding name="OrderProcessServiceSoapBinding" type="tns:OrderProcess">
<soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
. . .
</wsdl:binding>
```

下面的 WSDL 代码说明端点地址使用 SOAP 1.1：

```
<wsdl:service name="OrderProcessService
   <wsdl:port binding="tns:OrderProcessServiceSoapBinding" name="OrderProcessPort">
         <soap:address location="http://localhost:8080/OrderProcess">
   </wsdl:port>
</wsdl:service>
```

需要指定<soap:address>元素来发送 SOAP 1.1 消息。<soap:address>元素是<wsdl:port>元素的直接子元素。而<wsdl:port>元素是<wsdl:service>元素的一部分。该<soap:address>元素需要一个属性来指定位置，即指定端点的地址。

2. SOAP 1.2 协议

可以通过<soap12:binding>元素来定义 SOAP 1.2 协议。<soap12:binding>元素是<wsdl:binding>元素的子元素。当使用<soap12:binding>表示，标志着这项服务是绑定到 SOAP 版本 1.2 协议，即该消息将遵循 SOAP 1.2 的格式。该<soap12:binding>元素中的传输属性可以定义一个指定的传输协议。在这种情况下，它将会使用 HTTP 传输协议。其 URI 的表现形式：

```
http://schemas.xmlsoap.org/soap/http
```

下面的 WSDL 代码说明 SOAP 1.2 与 HTTP 传输绑定：

```
<wsdl:binding name="OrderProcessServiceSoapBinding" type="tns:OrderProcess">
<soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"
/>
. . .
```

```
</wsdl:binding>
```

下面的 WSDL 代码说明端点地址使用 SOAP 1.2：

```
<wsdl:service name="OrderProcessService">
  <wsdl:port binding="tns:OrderProcessServiceSoapBinding"  name="OrderProcessImplPort">
       <soap12:address location="http://localhost:8080/OrderProcess">
  </wsdl:port>
</wsdl:service>
```

需要指定<soap12:address>元素来发送 SOAP 1.2 消息。<soap12:address>元素是<wsdl:port>元素的直接子元素，而<wsdl:port>元素是<wsdl:service>元素的一部分。该<soap12:address>元素需要一个属性来指定位置，即指定端点的地址。

3. 独立 HTTP 协议

Web Services 通常遵循 SOAP 协议的消息格式。但可以根据应用程序的要求选择使用 HTTP 协议格式来发送消息。独立 HTTP 传输协议可以用 HTTP 消息格式来发送 Web Services。独立 HTTP 传输协议使用了 HTTP 的 GET 和 POST 方法实现消费者和服务端点之间的请求和响应。

下面 WSDL 文档显示独立 HTTP 绑定：

```
<binding name="OrderProcessServiceHttpBinding"  type="OrderProcess">
<http:binding verb="GET"/> <operation name="processOrder">
    <http:operation location="processOrder"/>
        <input>
            <http:urlEncoded/>
        </input>
        <output>
            <mime:content type="text/html"/>
        </output>
    </operation>
</binding>
```

前面的代码介绍 HTTP 绑定，这意味着输入和输出消息将采用 HTTP 协议的格式，作为一个 GET 请求发送的信息。输入的信息具有价值 http:urlEncoded，意味着该消息参数将采取一个名称=值对的形式。输出消息或将字符串返回类型为 HTML 格式。

还可以使用对立 HTTP 协议绑定指定服务端点地址。

```
<wsdl:service name="OrderProcessService">
    <wsdl:port binding="tns:OrderProcessServiceHttpBinding"  name="OrderProcessPort">
         <http:address location="http://localhost:8080/OrderProcess">
    </wsdl:port>
</wsdl:service>
```

需要指定<http:address>元素用 HTTP 格式发送消息。<http:address>元素是<wsdl:port>元素的直接子元素，而<wsdl:port>是<wsdl:service>元素的组成部分。该<http:address>元素有一个指定位置的属性，即指定端点地址。

4. HTTP 管道（HTTP Conduit）

HTTP 管道是一个特别的通道，在该通道中可以使用特定 HTTP 相关属性去影响端点之间的信息交换。通常可以指定 HTTP 连接属性，如是否允许分块、连接超时等。HTTP 管道总是由客

户端或服务端的消费者来定义。下面例子代码说明了 HTTP 管道的使用：

```
<beans …
xmlns:http-conf=http://cxf.apache.org/transports/http/configuration
…
<http-conf:conduit name="{http://order.demo} OrderProcessImplPort.http-conduit">
<http-conf:client Connection="Keep-Alive" AllowChunking="false" />
<http-conf:tlsClientParameters secureSocketProtocol="SSL">
</http-conf:tlsClientParameters>
</http-conf:conduit>
…
</beans>
```

HTTP 管道中的属性列表和说明如表 8-1 所示。

表 8-1　HTTP管道中的属性列表和说明

属性	说明
ConnectionTimeout	表示时间在毫秒级别。客户端尝试发送新连接请求之前的等待时间。默认值是30 000。该请求值为0表示永远不会超时
AllowChunking	它表明请求是否可以在块中发送到服务器。默认为true
CacheControl	它传输关于缓存行为的指令，当该信息请求传递给服务器
ContentType	它表明MIME数据是发送到服务器

HTTP 目的地的代码如下：

```
<beans ...
...
<http-conf:destination
name="{http://order.demo}OrderProcessImplPort.http-destination">
    <http-conf:server HonorKeepAlive="true" />
</http-conf:destination>
</beans>
```

HTTP 目的地的属性列表和说明如表 8-2 所示。

表 8-2　HTTP目的地的属性列表和说明

属性	说明
ReceiveTimeout	表示时间在毫秒级别。服务器尝试连接之前收到超时请求。默认值是30 000。0意味着服务器将永远不会超时
HonorKeepAlive	它表明响应被发送后，是否接受一个保持连接客户端的请求。默认值是false
CacheControl	它传输关于缓存行为的指令，当该信息请求传递给客户端
ContentType	它表明MIME数据是发送到客户端

Apache CXF 框架在创建服务中包括两个 HTTP 选项。一个 HTTP 选项是开发者把 Servlet 传输部署在 Servlet 容器中，还有一个 HTTP 选项是独立的基于 Jetty 的 HTTP 服务器。

- 客户端 HTTP 传输（包括 SSL 支持）
- Jetty 配置
- 服务器 HTTP 传输
- Servlet 的传输

- 独立 HTTP 传输

8.2.3　Spring 注入 HTTP 传输并基于 Servlet 的实现

本程序演示 Apache CXF 框架遵循 JAX-WS 规范的客户端和服务器端调用，客户端采用 Apache CXF 的代码优先（Java First）前端应用模式。采用 POJO 对象作为中间传输对象。本案例程序采用与 Spring 框架集成的方式来实现，即 Spring 框架注入客户端，同时在服务器上也是采用 Spring 框架注入来实现。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，都使用 POJO 对象实现客户端和服务器端的数据交换。后台服务端为遵循 JAX-WS 规范的 Endpoint 实现。前端访问采用 JAX-WS 规范的 Service 方式，实现的结构如图 8-1 所示。

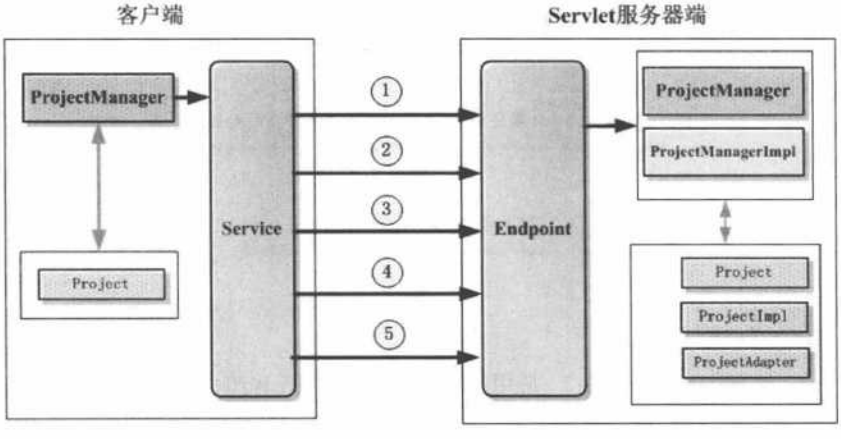


图 8-1　基于 HTTP 实现 JAX-WS 前端应用模式图

其中：① 表示客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。在客户端和服务端的 SEI 都是 ProjectManager 接口。

2. 应用程序案例实现步骤

要创建HTTP传输并基于Servlet容器的前端实现Web Services程序的流程步骤如图8-2所示。流程图说明，要实现以下步骤：

（1）创建一个 Web 服务项目。

（2）创建一个服务器端点接口（SEI）及其相关对象和注释。

（3）编写发布这个 Java 程序为 Web Services 的服务器端程序。

（4）编写 web.xml 和 server-bean.xml。

（5）发布配置文件到 Servlet 服务器。

（6）启动 Servlet 服务器程序，对外发布 Web Services。

（7）客户端调用 Web Services。

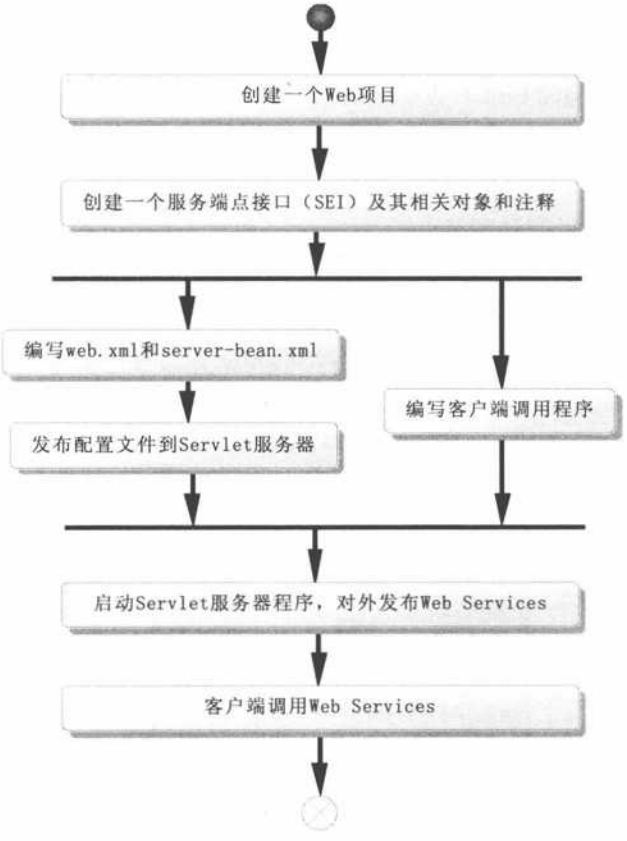


图 8-2　应用程序案例实现步骤流程图

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 8-3 所示。

包目录包括三个部分，第一个包目录 WebRoot 目录是服务器程序目录，包括 web.xml 和 server-bean.xml 文件；第二个包目录 demo.cxf.transport.http.client 是客户端程序目录；第三个包目录 demo.cxf.webservices 是 SEI 和数据对象等公用部分，客户端和服务器端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口。由于本服务器程序直接发布到

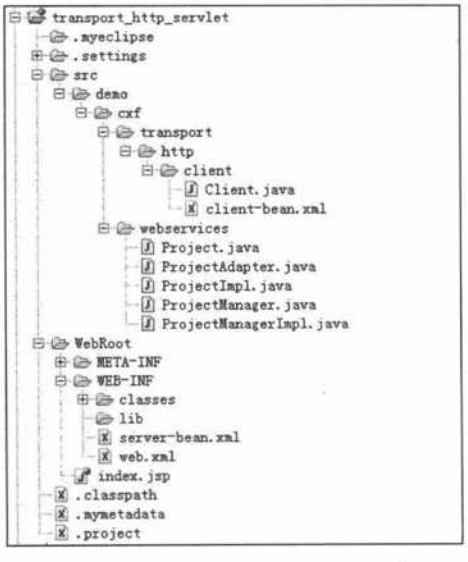


图 8-3　transport_http_servlet 程序结构

Servlet 容器，所以没有单独编写服务器发布程序，直接把配置文件和编译好的服务器端点接口（SEI）组件等部署到 Servlet 容器即可。

本案例与第 6 章“采用 POJO 并基于 Servlet 容器的前端实现”案例相比较，Servlet 服务端配置是完全相同。只是客户端采用了 Spring 注入方式来实现，就这么一点差别。下面只介绍有区别的客户端调用程序中的两部分客户端代码。

客户端程序包括程序文件 Client.java 和配置文件 client-bean.xml。

客户端 Client.java 实现模式的代码如下：

```
package demo.cxf.transport.http.client;

import org.springframework.context.support.ClassPathXmlApplicationContext;
import demo.cxf.webservices.Project;
import demo.cxf.webservices.ProjectImpl;
import demo.cxf.webservices.ProjectManager;

public class Client {
    public Client() {}

    public static void main(String[] args) throws Exception {
        ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(
                new String[] {"demo/cxf/transport/http/client/client-bean.xml"});

        ProjectManager projectManager = (ProjectManager) context.getBean("ProjectManagerClient");

        //项目的基本情况
        getGreet(projectManager);
        greetingWithParameter(projectManager);
        getProjectName(projectManager);
        getDefaultProject(projectManager);
        getAllProject(projectManager);

        //修改默认项目，查看项目的基本情况
        setDefaultProject(projectManager);

        //增加一个项目，查看项目的基本情况
        registerProject(projectManager);

        //删除一个项目，查看项目的基本情况
        removeProject(projectManager);

        System.exit(0);
    }

    //调用 SEI 的 getGreet 方法
    private static void getGreet(ProjectManager projectManager)throws Exception {
        System.out.println(projectManager.greeting());
    }
    ……
}
```

客户端首先定义了两个 QName 类型变量——SERVICE_NAME 和 PORT_NAME。SERVICE_NAME 表明 Web Services 的服务名称。PORT_NAME 表明 Web Services 的服务端口名称。我们在定义 SEI 时都采用默认方式，所以 SERVICE_NAME 值是“ProjectManager”，而 PORT_NAME

值是“ProjectManagerPort”。

应用程序的 Spring 配置文件是 client-bean.xml，代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/jaxws
http://cxf.apache.org/schemas/jaxws.xsd">

    <jaxws:client id="ProjectManagerClient"
serviceClass="demo.cxf.webservices.ProjectManager"
address="http://localhost:18080/transport_http_servlet/ProjectManager" />
</beans>
```

4. 测试和运行结果

运行客户端程序，其客户端程序和服务器端输出结果与第 6 章“采用 POJO 并基于 JAX-WS 服务的前端实现”案例相同。

5. 其他内容

上述的所有调用都基于 JAX-WS 规范。

本例代码参见配套资源代码中“第 08 章\transport_http_servlet”文件夹。

8.3 JMS 传输协议

本节先介绍 JMS 传输协议的基础知识，然后说明 Apache CXF 框架是如何实现 JMS 的。最后通过两个例子来进行进一步说明。

8.3.1 JMS 简介

JMS（Java Message Service）：是面向消息中间件相互通信的应用程序接口（API）。JMS 既支持点对点的域，又支持发布/订阅（publish/subscribe）类型的域，并且提供对下列类型的支持：经认可的消息传递、事务型消息的传递、一致性消息和具有持久性的订阅者支持。JMS 还提供了另一种方式来集成当前的应用与旧的后台系统。JMS 实现原理如图 8-4 所示。

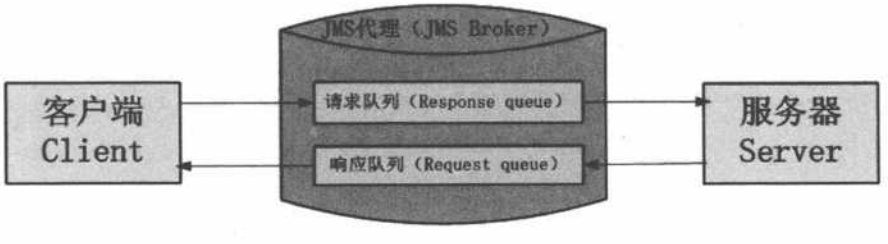


图 8-4 JMS 实现原理

Java Message Service 是由 Sun Microsystems 开发的。JMS 为 Java 程序提供了一种访问企

业消息系统的方法。企业消息系统，即面向消息的中间件（MOM），提供了以松散耦合的灵活方式集成应用程序的一种机制。它们提供了基于存储和转发的应用程序之间的异步数据发送，即应用程序彼此不直接通信，而是与作为中介的 MOM 通信。MOM 提供了有保证的消息发送，应用程序开发人员无须了解远程过程调用（PRC）和网络/通信协议的细节。

JMS 规范 1.1 声称：JMS 是一组接口和相关语义，它定义了 JMS 客户如何访问企业消息产品的功能。

JMS 体系结构，JMS 发送和接收消息的基础是连接，它负责分配 JVM 之外的资源。通常 JMS 提供者至少为 P2P 事务实现一个 QueueConnection 连接，至少为 P/S 事务实现一个 TopicConnection 连接。这些连接提供一个 Session，它是管理消息发送和接收的结构。

P2P 事务管理的基本结构是 QueueSender 和 QueueReceiver。用于 P/S 事务管理的基本结构是 TopicSubscriber 和 TopicPublisher。Topic 和 Queue 对象封装了每条目标消息和源消息的特定信息。这一层次结构如图 8-5 所示。

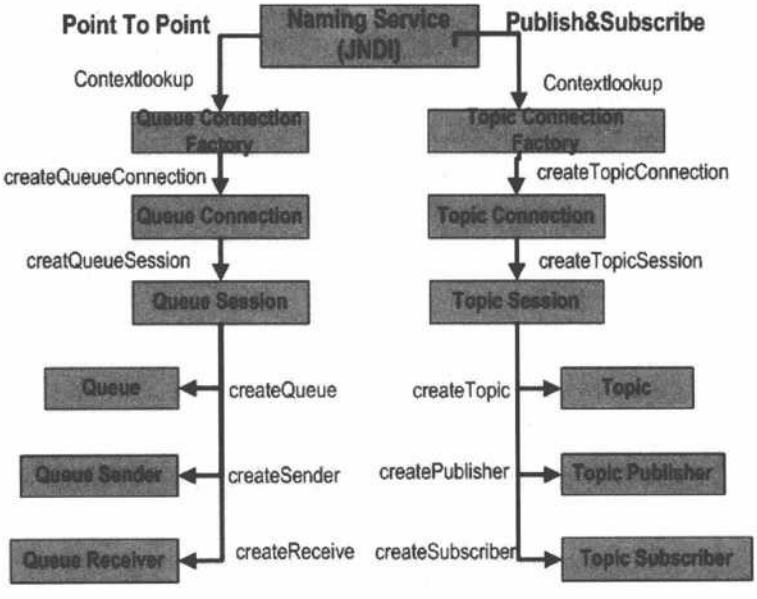


图 8-5　JMS 类层次结构

其他特定于应用程序服务器的结构（如请求 / 响应支持类）和特性可以在 JMS 标准中找到。JMS 应用程序由以下元素组成。

- JMS 客户机。用 JMS API 发送和接收消息的 Java 程序。
- 非 JMS 客户机。由于考虑在传统程序中，通常会涉及 JMS 应用程序的一部分，所以在规划时非 JMS 客户机必须优先考虑 JMS 应用的存在。
- 消息。就 JMS 应用程序设计而言，通过 JMS 和非 JMS 客户机所交换的消息格式和内容是完整的。
- JMS 提供者。如前所述，JMS 定义了一组接口，JMS 提供者必须提供特定于其 MOM 产

品的具体实现。

- 管理对象。消息系统提供者的管理员创建并独立于提供者专有技术的对象。

开发一个典型的 JMS 程序需有以下步骤才能产生和使用消息：

（1）通过 JNDI 查询 ConnectionFactory。

（2）通过 JNDI 查询一个或者多个 Destination。

（3）用 ConnectionFactory 创建一个 Connection。

（4）用 Connection 创建一个或者多个 Session。

（5）用 Session 和 Destination 创建所需要的 MessageProducer 和 MessageConsumer。

（6）启动 Connection。

这时，消息就可以开始流动，应用程序可以根据需要接收、处理和发送消息。

如果要了解关于 JMS 更详细的信息，可以参阅 The Java Message Service specification, version 1.1。

8.3.2 在 Apache CXF 中使用 JMS

Apache CXF 框架提供传输插件来促使端点使用 Java 消息服务（JMS）队列和主题。Apache CXF 框架的 JMS 传输插件使用 Java 命名和目录接口(JNDI)来查找和获取 JMS 引用。一旦 Apache CXF 框架已经建立一个连接到一个 JMS 提供者，无论是作为一个 JMS 的 JMS ObjectMessage 或 TextMessage，Apache CXF 框架都支持消息包装的传递。

使用新 JMSConfigFeature 可以更方便地配置。Apache CXF 框架的 2.0.9 版本和 2.1.3 版本开始提供一个新的更简单、更灵活的配置方式。基于 SOAP 的 JMS 传输超过 JMS 1.0。从 Apache CXF 框架的 2.3 版本开始，Apache CXF 框架做了一些基于 SOAP over JMS 的 JMS 传输规范支持的改善。

1. JMS 命名空间

（1）WSDL 命名空间

在命名空间 http://cxf.apache.org/transports/jms 定义一个 JMS 端点的 WSDL 扩展定义。为了使用 JMS 扩展，开发人员需要添加下面的命名空间来定义内容。JMS 扩展命名空间的代码如下：

```
xmlns:jms="http://cxf.apache.org/transports/jms"
```

（2）配置命名空间

为了使用 JMS 配置属性，开发人员需要添加下面的 bean 配置元素来显示应用程序的路径。

```
xmlns:jms="http://cxf.apache.org/transports/jms"
```

2. 基本端点配置

JMS 端点需要知道一些关于如何建立一个正确的连接目的地的基本信息。这些信息可以通过 WSDL 或 XML 配置两种方式之一获取。下面的配置中所描述的元素都可以用于客户端和服务器端目的地。

（1）使用 WSDL

JMS 目的地信息通过 jms:address 和其子元素提供，jms:address 元素属性指定需要确定的 JMS

代理和目标信息。jms:JMSNamingProperties 元素指定 Java 用于连接到 JNDI 服务的属性。

（2）元素的地址

对于一个 JMS 端点的基本配置是使用一个 jms:address 元素来作为服务端口的子元素。jms:address 元素使用下面描述的属性来配置 JMS 代理连接。jms:address 元素的属性如表 8-3 所示。

表 8-3　jms:address元素的属性和说明

属性	说明
destinationStyle	指定JMS目的地是一个JMS队列或者是一个JMS主题
jndiConnectionFactoryName	指定绑定到JMS连接工厂的JNDI名称，以供连接到JMS目的地时使用
jndiDestinationName	指定绑定到JMS连接工厂的JNDI名称，以供请求被发送
jndiReplyDestinationName	指定绑定到JMS连接工厂的JNDI名称，以供答复被发送。这个属性允许开发人员用自定义的目的地来进行答复
connectionUserName	当连接到JMS broker时，指定使用的用户名
connectionPassword	当连接到JMS broker时，指定使用的密码

（3）JMSNamingProperties 元素

为了增加 JNDI 提供者和 JMS 的互操作性，jms:address 元素有一个子元素 jms:JMSNamingProperties，允许开发人员通过指定特定值来达到连接到 JNDI 提供者的属性。jms:JMSNamingProperties 元素有两个属性：name 和 value。name 属性指定设置的属性名称。value 属性指定 jms:JMSNamingProperties 元素的属性数值。在 jms:JMSNamingProperties 元素也可以用于提供者特定的属性规范。

以下是 JNDI 常见可设置的属性列表：

- java.naming.factory.initial
- java.naming.provider.url
- java.naming.factory.object
- java.naming.factory.state
- java.naming.factory.url.pkgs
- java.naming.dns.url
- java.naming.authoritative
- java.naming.batchsize
- java.naming.referral
- java.naming.security.protocol
- java.naming.security.authentication
- java.naming.security.principal
- java.naming.security.credentials
- java.naming.language
- java.naming.applet

关于这些属性的详细资料，可参考 JNDI 提供的文件或阅读 Java API 参考资料。

（4）使用命名答复目的地

默认情况下，Apache CXF 端点使用 JMS 创造一个来回发送答复的临时队列。开发者可以通过修改 jndiReplyDestinationName 属性来更改此项设置的行为。客户端的端点将监听指定的目的

地的答复，并在所有的传出请求覆盖的 ReplyTo 领域指定属性值。如果没有在请求中指定任何目的地，服务器端点将使用 jndiReplyDestinationName 属性值作为放置位置的答复。

下面的示例显示了一个 JMS 规范的 WSDL 端口例子。JMS WSDL 端口代码定义如下：

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
      <jms:address jndiConnectionFactoryName="ConnectionFactory"
               jndiDestinationName="dynamicQueues/test.cxf.jmstransport">
         <jms:JMSNamingProperty name="java.naming.factory.initial"
                        value="org.apache.activemq.jndi.ActiveMQInitialContextFactory" />
         <jms:JMSNamingProperty name="java.naming.provider.url"
                         value="tcp://localhost:61616" />
    </jms:address>
  </port>
</service>
```

3. 使用配置

除了使用 WSDL 文件来指定一个 JMS 端点的连接信息，还可以用 XML 文件来进行配置。针对端点内容，配置文件中的信息将覆盖 WSDL 文件的信息。

（1）配置元素

可以使用下列配置元素之一来配置一个 JMS 端点。

jms:conduit：在该元素中包含一个消费者的端点配置。它有一个属性、名称，存在认同价值的格式，如下所示。

```
{WSDLNamespace}WSDLPortName.jms-conduit
```

jms:destination：该元素包含一个提供者的端点配置。它有一个属性、名称，存在认同价值的格式，如下所示。

```
{WSDLNamespace}WSDLPortName.jms-destination
```

（2）地址元素

JMS 连接信息通过一个指定的 jms:address 基本配置添加子元素。在配置文件中使用 jms:address 元素与 WSDL 文件中使用的元素是相同的。jms:address 元素列出的属性是 WSDL 文件中地址元素的属性表。如在 WSDL 文件中的 jms:address 元素，jms:address 配置元素也有一个 jms:JMSNamingProperties 的子元素，该子元素用于指定其他信息连接到一个 JNDI 提供者。

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:ct="http://cxf.apache.org/configuration/types"
      xmlns:jms="http://cxf.apache.org/transports/jms"
      xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.
springframework.org/schema/beans/spring-beans.xsd
            http://cxf.apache.org/jaxws
http://cxf.apache.org/schemas/jaxws.xsd
http://cxf.apache.org/transports/jms
http://cxf.apache.org/schemas/configuration/jms.xsd">

<jms:conduit name="{http://cxf.apache.org/jms_endpt}HelloWorldJMSPort.jms-conduit">
     <jms:address destinationStyle="queue"
            jndiConnectionFactoryName="myConnectionFactory"
            jndiDestinationName="myDestination"
```

```
            jndiReplyDestinationName="myReplyDestination"
            connectionUserName="testUser"
            connectionPassword="testPassword">
        <jms:JMSNamingProperty name="java.naming.factory.initial"
                value="org.apache.cxf.transport.jms.MyInitialContextFactory"/>
        <jms:JMSNamingProperty name="java.naming.provider.url"
                    value="tcp://localhost:61616"/>
    </jms:address>
</jms:conduit>
</beans>
```

4. 消费者端点配置

JMS 消费者端点指定它们所使用的消息类型。JMS 消费者端点可以使用一个 JMS ObjectMessage 或 JMS TextMessage。当使用 ObjectMessage 时，消费者端点把一个 byte []作为存储数据的方法以及从 JMS 消息体中检索数据。当消息发送，消息数据，包括任何格式化的消息，被打包成一个 byte []和 JMS 消息放置到 BUS 总线上。当接收到消息，消费者端点将尝试解组 JMS 体中存储的数据，就好像它是一个 byte []包装一样。

当使用一个 TextMessage 时，消费者端点使用字符串类型方式来存储和检索来自 JMS 消息体的数据。当消息被发送，消息信息，包括任何形式的特定信息，将转换成一个字符串放置到 JMS 消息体中。当消费者收到消息，如果该 JMS 消息体被封装成一个字符串，消费者端点将尝试解组存储在 JMS 消息体的数据。

当本机 JMS 应用程序与 Apache CXF 消费者交互时，JMS 应用程序负责解释消息和格式化消息。

消费者端点都可以通过 WSDL 和 XML 文件来进行配置。

5. 使用配置

JMS 服务器端点有大量可以在合同中配置的行为，包括：

- 消息是如何互相关联的。
- 持久订阅的使用。
- 如果服务使用本地 JMS，如何进行交易。
- 端点使用的消息选择器。

服务器端点可以配置两种方法之一：

- 配置方式进行配置
- 采用 WSDL 文件方式进行配置

（1）指定的消息类型

开发者可以使用一个 jms:runtimePolicy 元素的一个属性来指定消费者支持的消息类型。

messageType ——指定如何将消息数据打包为 JMS 消息。文本类型数据将封装为一个 TextMessage，二进制数据将封装为一个 ObjectMessage。

被 JMS 消费者端点接收的消息类型配置使用可选的 jms:client 元素。jms:client 元素是 WSDL 端口元素的子元素。

使用 jms:destination 元素，开发人员可以配置服务器端点。可以使用 jms:runtimePolicy 元素来指定服务器端点的行为，jms:runtimePolicy 元素具有的属性见表 8-4。

表 8-4 jms:runtimePolicy元素的属性

属性	描述
useMessageIDAsCorrealationID	指定JMS broker是否将使用消息ID去关联消息。默认为false
durableSubscriberName	指定一个名称，用于注册一个持久订阅
messageSelector	指定一个使用选择器的消息字符串值。有更多用于指定消息的语法信息
selectors	参照JMS 1.1的规范
transactional	指定本地JMS broker是否将创建消息过程中的事务。默认为false

下面的例子显示了配置 JMS 消费者端点的配置项：

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:ct="http://cxf.apache.org/configuration/types"
      xmlns:jms="http://cxf.apache.org/transports/jms"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd"
http://cxf.apache.org/jaxws
http://cxf.apache.org/schemas/jaxws.xsd
http://cxf.apache.org/transports/jms
http://cxf.apache.org/schemas/configuration/jms.xsd">
...
<jms:conduit
name="{http://cxf.apache.org/jms_endpt}HelloWorldJMSPort.jms-conduit">
       <jms:address ... >
...
</jms:address>
<jms:runtimePolicy messageType="binary"/>
...
</jms:conduit>
...
</beans>
```

jms:conduit 的 id 在{WSDLNamespace} WSDLPortName.jms-conduit 的前面。这些信息提供给 Apache CXF 框架，以便 Apache CXF 框架能够使用服务器端点来联合配置。

（2）使用 WSDL

服务器端点行为使用可选的 jms:server 元素来进行配置。jms:server 元素是 WSDL 端口元素的子元素，其属性见表 8-5。

表 8-5 jms:server元素中WSDL端口元素的子元素和属性

属性	描述
useMessageIDAsCorrealationID	指定JMS broker是否将使用消息ID去关联消息。默认为false
durableSubscriberName	指定一个名称，用于注册一个持久订阅
messageSelector	指定一个使用选择器的消息字符串值。有更多用于指定消息的语法信息
selectors	参照JMS 1.1的规范
transactional	指定本地JMS broker是否将创建消息过程中的事务。默认为false。目前还不支持

（3）JMS 运行配置

除了配置 JMS 端点外部可视的内容外，开发人员还可以配置其内部运行时的行为。有三种类

型的运行配置。

6. 会话池配置（常见于服务业和消费者）

使用 jms:sessionPoolConfig 元素来配置一个端点的 JMS 会话池。此属性允许开发人员设置 JMS 会话的数量。端点被保证维持 JMS 会话池的数量不能超过高位，也不能低于低位。

表 8-6 列出 jms:sessionPool 元素的属性。

表 8-6　jms:sessionPool元素的属性和描述

属性	描述
lowWaterMark	指定端点JMS会话池的最低数量。默认为20
highWaterMark	指定端点JMS会话池的最高数量。默认为500

下面的例子显示了如何配置 Apache CXF 框架中 JMS 服务器端点的会话池：

```
<jms:destination
name="{http://cxf.apache.org/jms_endpit}HelloWorldJMSPort.jms-destination">
...
  <jms:sessionPool lowWaterMark="10" highWaterMark="5000" />
</jms:destination>
```

在 jms:sessionPool 元素内也可以用 jms:conduit.。

消费者特定的运行配置，消费者 JMS 配置允许开发人员指定两个运行时的行为：

- 消费者会等待响应的毫秒数。
- 在 JMS 代理删除前，请求存在的毫秒数。

使用 jms:clientConfig 元素来设置 JMS 消费者运行时的行为。jms:clientConfig 元素的属性在表 8-7 中列出。

表 8-7　jms:clientConfig元素的属性和描述

属性	描述
clientReceiveTimeout	指定的时间量，以ms为单位，即端点将等待超时之前的响应，并发出一个异常。默认值是2000
messageTimeToLive	指定的时间量，以ms为单位，一个请求能保持没有接受的时间，同时JMS broker不能将其删除。默认值是0，指定该消息不能被删除

下面的示例显示了一个配置片段，设置消费者端点请求的生命周期为 500ms，其超时值是 500ms。

```
<jms:conduit name="{http://cxf.apache.org/jms_endpt}HelloWorldJMSPort.jms-conduit">
...
  <jms:clientConfig clientReceiveTimeout="500"
      messageTimeToLive="500" />
</jms:conduit>
```

服务器指定运行时的配置。JMS 服务器配置允许开发人员指定运行时的行为：

- 在 JMS 代理删除前，响应信息能保持不接受的时间量。
- 当创建和访问持久订阅时所使用的客户端标识符。

jms:serverConfig 元素用于指定服务运行时的配置。表 8-8 列出该元素的属性，指定配置值可

以在运行时控制服务的行为。

表 8-8 jms:serverConfig元素的属性和描述

属性	描述
messageTimeToLive	指定的时间量，以ms为单位，一个响应可以保持未读的时间，同时JMS broker不允许删除。默认为0，指定的消息可以永远保持
durableSubscriptionClientId	指定客户端标识符的端点，可用于创建和访问持久订阅

下面的示例显示了一段配置代码，设置服务器端点反应寿命为 500ms，其持久订阅时所使用的客户端标识符 id 是“jms-test-id”。

```
<jms:destination id="{http://cxf.apache.org/jms_endpt}HelloWorldJMSPort.jms-destination">
  <jms:address ... >
    ...
  </jms:address>
  <jms:serverConfig messageTimeToLive="500"
                durableSubscriptionClientId="jms-test-id" />
</jms:destination>
```

8.3.3 Spring 注入实现 JMS 的例子程序

本程序演示 Apache CXF 遵循 JAX-WS 规范并采用 JMS 传输协议实现客户端和服务器端调用，客户端采用 Apache CXF 的代码优先（Java First）前端应用模式。客户端和服务器端都采用 Spring IoC 注入来实现。本案例程序选择 activeMQ 这个开源的 JMS 消息组件，关于 activeMQ 的介绍详见网站“http://activemq.apache.org/”。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，后台服务器端为遵循 JAX-WS 规范的 Endpoint 来实现，采用 Spring IoC 注入方式。采用 JMS 传输协议方式。前端访问直接采用 Spring IoC 注入方式，实现的结构如图 8-6 所示。

其中：①表示客户端通过 Service 获得 SEI，访问后台的项目基本情况；②表示客户端通过 SEI 修改默认项目；③表示客户端通过 SEI 增加一个项目；④表示客户端通过 SEI 删除一个项目；⑤表示客户端通过 SEI 查看后台的全部项目情况。在客户端和服务端的 SEI 都是 ProjectManager 接口。

由于采用了 activeMQ 的一个开源 JMS 消息组件。所以程序要引入 activemq-all-5.3.1.jar 包。

2. 应用程序案例实现步骤

要创建 JMS 应用并实现 Web Services 程序的流程步骤如图 8-7 所示。

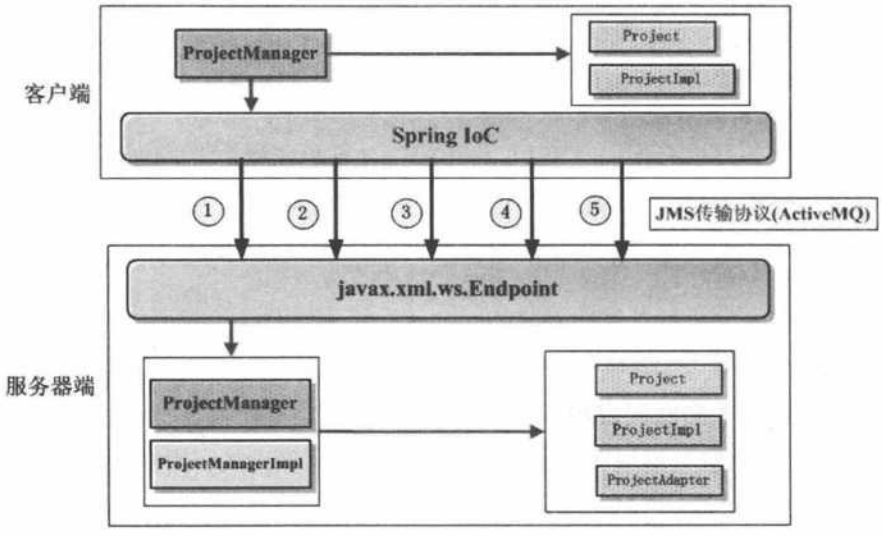


图 8-6　基于 JMS 实现 JAX-WS 前端应用模式图

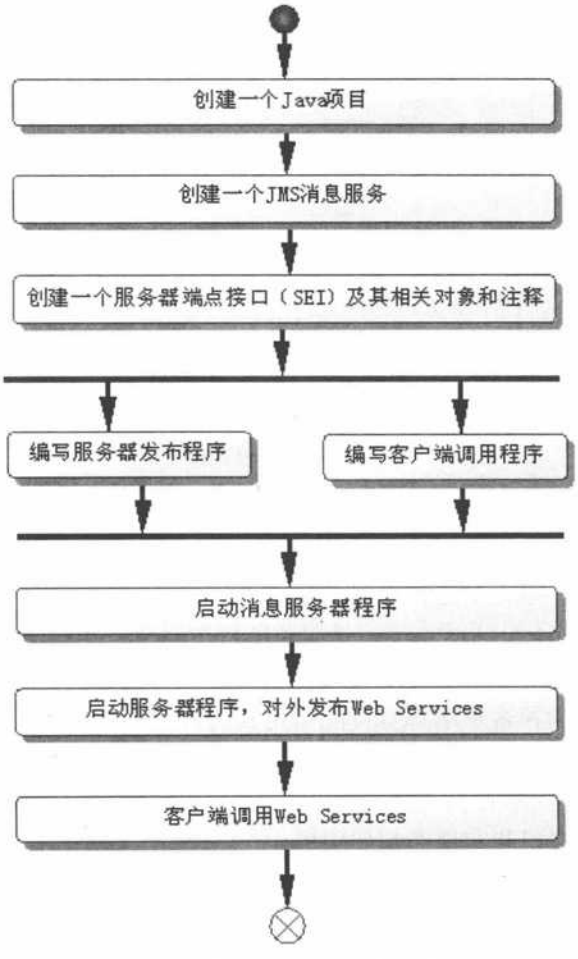


图 8-7　应用程序案例实现步骤流程图

流程图说明，要实现以下步骤：

（1）创建一个 Java 项目。

（2）创建一个 JMS 消息服务。

（3）创建一个服务器端点接口（SEI）及其相关对象和注释。

（4）编写发布这个 Java 程序为 Web Services 的服务器端程序。

（5）编写 Web Services 的客户端应用程序。

（6）启动消息服务程序。

（7）启动服务器程序，对外发布 Web Services。

（8）客户端调用 Web Services。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 8-8 所示。

图 8-8　transport_jms_spring 程序结构

包目录包括四个部分，第一个包目录 demo.cxf.transport.jms.broker 是消息服务器程序目录；第二个包目录 demo.cxf.transport.jms.server 是服务器程序目录；第三个包目录 demo.cxf.transport.jms.client 是客户端程序目录；第四个包目录 demo.cxf.webservices 是服务器端点接口 SEI 和相关对象等公用部分，服务器端和客户端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口，服务器端和客户端都必须引用。但服务器端和客户端对公用部分的引用是不同的。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口，而服务器端程序是接口和实现类都要引用。

整个应用程序包括了消息组件程序、服务器端应用程序和客户端应用程序。下面分别介绍四部分：①消息组件程序；②服务器端点接口 SEI 和相关对象；③服务器端发布程序；④客户端调用程序。

（1）消息组件程序

消息组件程序只有一个文件，即 MessageBroker.java，其代码如下：

```
package demo.cxf.transport.jms.broker;

import org.apache.activemq.broker.BrokerService;
import org.apache.activemq.store.memory.MemoryPersistenceAdapter;

public final class MessageBroker {
    private MessageBroker() {}

    public static void main(String[] args) throws Exception {
        BrokerService broker = new BrokerService();
        broker.setPersistenceAdapter(new MemoryPersistenceAdapter());
        broker.addConnector("tcp://localhost:61616");
        broker.start();
        System.out.println("JMS broker ready ...");
        Thread.sleep(5 * 60 * 1000);
        System.out.println("JMS broker exiting");
        broker.stop();
        System.exit(0);
    }
}
```

（2）服务器端点接口（SEI）和相关对象

服务器端应用程序主要包括 Project 组件和服务器启动类。Project 组件是标准的后面组件，与其他例子完全相同，这里就不再列出程序代码了。

（3）服务器发布程序

本例服务器端应用程序采用 Spring IoC 注入方式实现，下面只列出有区别的两个文件的代码。

第一个文件是服务器发布程序，即 Server.java，代码如下：

```
package demo.cxf.transport.jms.server;

import javax.xml.ws.Endpoint;

import org.apache.cxf.Bus;
import org.apache.cxf.bus.spring.SpringBusFactory;
import demo.cxf.webservices.*;

public class Server {
    public Server() {
        System.out.println("Starting Server");
        ProjectManager projectManager = new ProjectManagerImpl();
        String address = "http://localhost:9000/ProjectManager";
        SpringBusFactory factory = new SpringBusFactory();
        Bus bus = factory.createBus("demo/cxf/transport/jms/server/server-bean.xml");
        factory.setDefaultBus(bus);
        Endpoint.publish(address, projectManager);
    }

    public static void main(String[] args) throws Exception {
        new Server();
        System.out.println("Server ready...");

        Thread.sleep(5 * 60 * 1000);
```

```
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

服务器注入的 Bean 配置文件——server-bean.xml：

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:simple="http://cxf.apache.org/simple"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xmlns:soap="http://cxf.apache.org/bindings/soap"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://cxf.apache.org/bindings/soap
        http://cxf.apache.org/schemas/configuration/soap.xsd
        http://cxf.apache.org/simple
        http://cxf.apache.org/schemas/simple.xsd
        http://cxf.apache.org/jaxws
        http://cxf.apache.org/schemas/jaxws.xsd">

    <import resource="classpath:META-INF/cxf/cxf.xml" />
    <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
    <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />
    <import resource="classpath:META-INF/cxf/cxf-extension-jms.xml" />

    <jaxws:endpoint id="ProjectManager"
        implementor="demo.cxf.webservices.ProjectManagerImpl" address="jms://" >
        <jaxws:features>
            <bean class="org.apache.cxf.transport.jms.JMSConfigFeature" p:jmsConfig-ref=
"jmsConfig" />
        </jaxws:features>
    </jaxws:endpoint>

    <bean id="jmsConfig" class="org.apache.cxf.transport.jms.JMSConfiguration"
        p:connectionFactory-ref="jmsConnectionFactory"
        p:targetDestination="test.cxf.jmstransport.queue" />

    <bean id="jmsConnectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="tcp://localhost:61616" />
    </bean>
</beans>
```

（4）客户端调用程序

客户端应用程序主要包括 SEI 接口和客户端访问类。Project 组件是标准的后面组件，与其他例子完全相同，这里就不再列出程序代码了。

本案例客户端应用程序采用 Spring IoC 注入方式实现，下面只列出有区别的两个文件的代码。

客户端程序 Client.java 的代码如下：

```
package demo.cxf.transport.jms.client;

import org.springframework.context.support.ClassPathXmlApplicationContext;
```

```
……

public class Client {
    public Client() {   }

    public static void main(String[] args) throws Exception {
        ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(
                new String[] { "demo/cxf/transport/jms/client/client-bean.xml" });

        ProjectManager projectManager = (ProjectManager) context.getBean("ProjectManagerClient");

        // 项目的基本情况
        getGreet(projectManager);
        greetingWithParameter(projectManager);
        ……
        System.exit(0);
    }

    // 调用 SEI 的 getGreet 方法
    private static void getGreet(ProjectManager projectManager)
            throws Exception {
        System.out.println(projectManager.greeting());
    }
    ……
}
```

客户端注入的 Bean 配置文件——client-bean.xml：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                    http://www.springframework.org/schema/beans/spring-beans.xsd
                    http://cxf.apache.org/jaxws
                    http://cxf.apache.org/schemas/jaxws.xsd">

    <jaxws:client id="ProjectManagerClient"
        serviceClass="demo.cxf.webservices.ProjectManager"
        address="jms://" >
        <jaxws:features>
            <bean class="org.apache.cxf.transport.jms.JMSConfigFeature"
                p:jmsConfig-ref="jmsConfig" />
        </jaxws:features>
    </jaxws:client>

    <bean id="jmsConfig"
        class="org.apache.cxf.transport.jms.JMSConfiguration"
        p:connectionFactory-ref="jmsConnectionFactory"
        p:targetDestination="test.cxf.jmstransport.queue" />

    <bean id="jmsConnectionFactory"
        class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL" value="tcp://localhost:61616" />
    </bean>
</beans>
```

4. 测试和运行结果

运行本程序的过程分为三个步骤：

第 1 步是启动 JMS 消息组件程序，即运行 MessageBroker.java 文件程序。

第 2 步是启动 Web Services 服务器组件程序，即运行 Server.java 文件程序。

第 3 步是运行客户端程序，即运行 Client.java 文件程序。

运行客户端程序，其客户端程序和服务器端输出结果与第 6 章“采用 POJO 并基于 JAX-WS 服务的前端实现”案例的相同。

5. 其他内容

上述的所有调用都基于 JAX-WS 规范。

本例代码参见配套资源代码中“第 08 章\transport_jms_spring”文件夹。

8.3.4 Spring 注入实现 JMS 的 Servlet 例子程序

本程序演示 Apache CXF 遵循 JAX-WS 规范并采用 JMS 传输协议实现客户端和服务器端调用，客户端采用 Apache CXF 的代码优先（Java First）前端应用模式。在客户端和服务器端都采用 Spring IoC 注入来实现。同样，本案例程序选择 activeMQ 这个开源的 JMS 消息组件。服务器端采用 Servlet 容器。采用 POJO 对象作为中间传输对象。本例 Servlet 容器采用 Tomcat 6。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，后台服务器端为遵循 JAX-WS 规范的 Endpoint 来实现，采用 Spring IoC 注入方式。采用 JMS 传输协议方式。前端访问直接采用 Spring IoC 注入方式，实现的结构如图 8-9 所示。

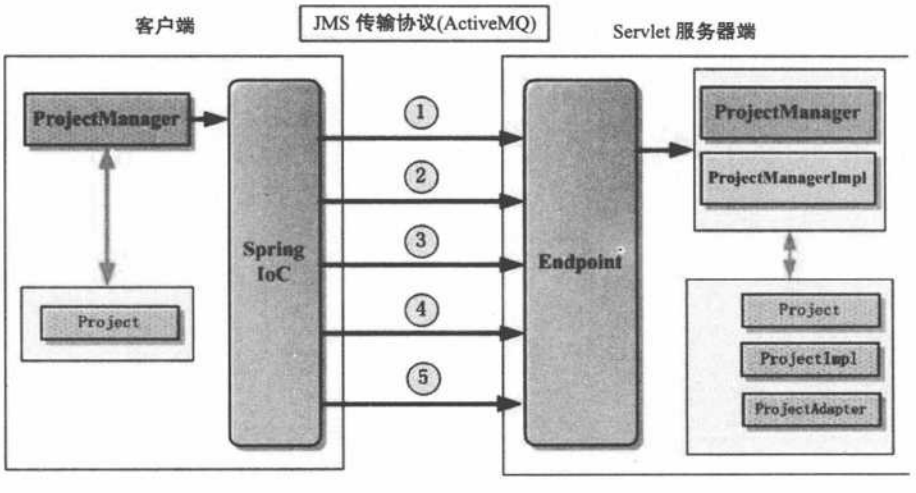


图 8-9 基于 JMS 实现 Servlet 前端应用模式图

其中：①表示客户端通过 Service 获得 SEI，访问后台的项目基本情况；②表示客户端通过 SEI 修改默认项目；③表示客户端通过 SEI 增加一个项目；④表示客户端通过 SEI 删除一个项目；⑤表示客户端通过 SEI 查看后台的全部项目情况。在客户端和服务端的 SEI 都是 ProjectManager

接口。

由于采用了 activeMQ，所以程序要引入 activemq-all-5.3.1.jar 包。

2. 应用程序案例实现步骤

要创建 JMS 应用并基于 Servlet 容器的前端实现 Web Services 程序的流程步骤如图 8-10 所示。

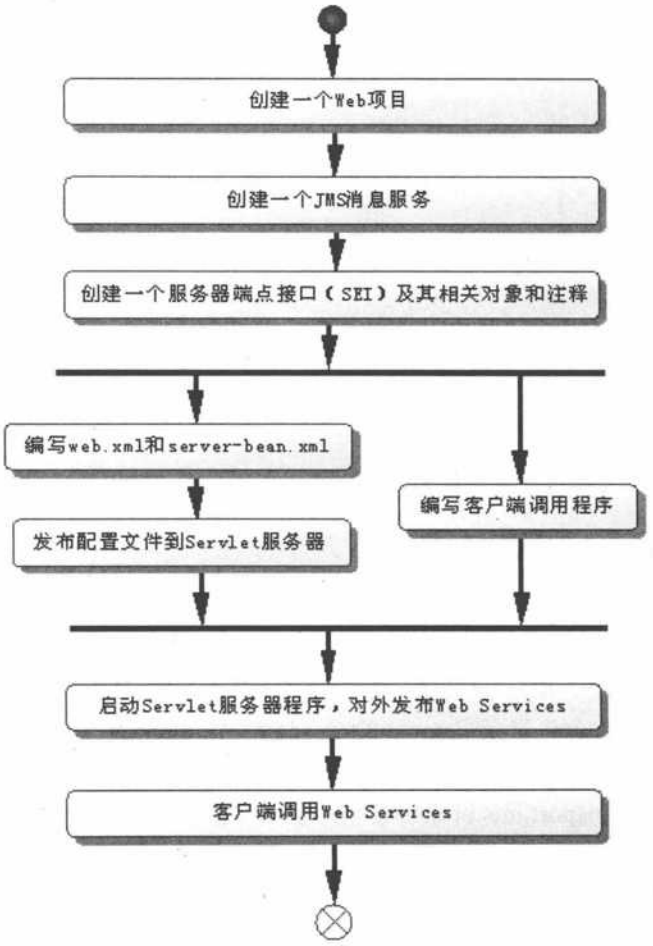


图 8-10　应用程序案例实现步骤流程图

流程图说明，要实现以下步骤：

（1）创建一个 Web 服务项目。

（2）创建一个 JMS 消息服务。

（3）创建一个服务器端点接口（SEI）及其相关对象和注释。

（4）编写发布这个 Java 程序为 Web Services 的服务器端程序。

（5）编写 web.xml 和 server-bean.xml。

（6）发布配置文件到 Servlet 服务器。

（7）启动 Servlet 服务器程序，对外发布 Web Services。

（8）客户端调用 Web Services。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 8-11 所示。

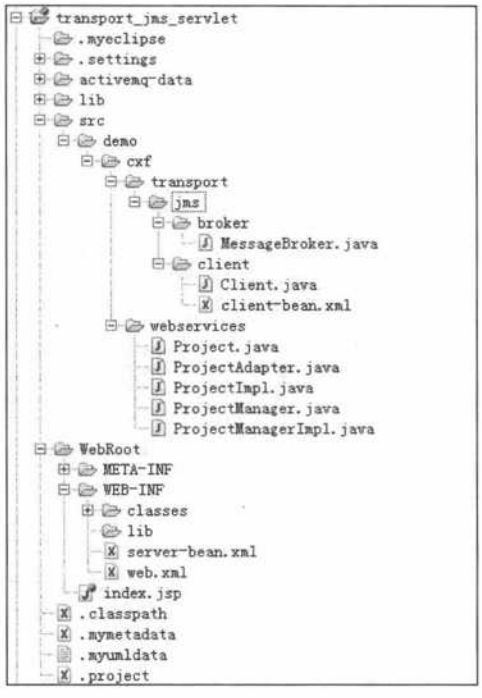


图 8-11　transport_jms_servlet 程序结构

包目录包括四个部分，第一个包目录 WebRoot 是服务器程序目录，包括 web.xml 和 server-bean.xml 文件；第二个包目录 demo.cxf.transport.jms.client 是客户端程序目录；第三个包目录 demo.cxf.webservices 是 SEI 和数据对象等公用部分，客户端和服务器端都有可能引用其内容；第四个包目录 demo.cxf.transport.jms.broker 是消息服务器程序。

服务器端点接口（SEI）是公用部分的接口。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口。由于本程序直接发布到 Servlet 容器，所以没有单独编写服务器发布程序，直接把配置文件和编译好的服务器端点接口（SEI）组件等部署到 Servlet 容器即可。

下面分别介绍三部分：①消息组件程序；②服务器端配置文件；③客户端调用程序。

（1）消息组件程序

我们选择 activemq 这个开源的消息组件。启动消息组件。MessageBroker 的程序代码如下：

```
package demo.cxf.transport.jms.broker;

import org.apache.activemq.broker.BrokerService;
import org.apache.activemq.store.memory.MemoryPersistenceAdapter;

public final class MessageBroker {
    private MessageBroker() {}

    public static void main(String[] args) throws Exception {
        BrokerService broker = new BrokerService();
```

```
        broker.setPersistenceAdapter(new MemoryPersistenceAdapter());
        broker.addConnector("tcp://localhost:61616");
        broker.start();
        System.out.println("JMS broker ready ...");
        Thread.sleep(5 * 60 * 1000);
        System.out.println("JMS broker exiting");
        broker.stop();
        System.exit(0);
    }
}
```

（2）服务器端配置文件

服务器端配置文件包括 web.xml 文件和 server-bean.xml 文件。首先要定义的 web.xml 文件与之前完全相同，这里就不列出详细代码了。

Servlet 启动，将读取 WEB-INF 目录下的 server-bean.xml 文件，server-bean.xml 的配置文件如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xmlns:jms="http://cxf.apache.org/transports/jms"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation=" http://www.springframework.org/schema/beans
                         http://www.springframework.org/schema/beans/spring-beans.xsd
                         http://cxf.apache.org/jaxws
                         http://cxf.apache.org/schemas/jaxws.xsd ">
    <import resource="classpath:META-INF/cxf/cxf.xml" />
    <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
    <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />
    <import resource="classpath:META-INF/cxf/cxf-extension-jms.xml" />

    <jaxws:endpoint id="ProjectManager"
implementor="demo.cxf.webservices.ProjectManagerImpl" address="jms://" >
        <jaxws:features>
            <bean class="org.apache.cxf.transport.jms.JMSConfigFeature" p:jmsConfig-ref=
"jmsConfig" />
        </jaxws:features>
    </jaxws:endpoint>

    <bean id="jmsConfig" class="org.apache.cxf.transport.jms.JMSConfiguration"
        p:connectionFactory-ref="jmsConnectionFactory"
        p:targetDestination="test.cxf.jmstransport.queue" />

    <bean id="jmsConnectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="tcp://localhost:61616" />
    </bean>
</beans>
```

（3）客户端调用程序

开发一个 Web Service 客户端，客户端程序代码如下：

```
package demo.cxf.transport.jms.client;

import org.springframework.context.support.ClassPathXmlApplicationContext;
……
```

```
public class Client {
    public Client() {   }

    public static void main(String[] args) throws Exception {
        ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(
                new String[] { "demo/cxf/transport/jms/client/client-bean.xml" });

        ProjectManager projectManager = (ProjectManager) context.getBean
("ProjectManagerClient");

        // 项目的基本情况
        getGreet(projectManager);
        greetingWithParameter(projectManager);
        ……
        System.exit(0);
    }

    // 调用 SEI 的 getGreet 方法
    private static void getGreet(ProjectManager projectManager)
            throws Exception {
        System.out.println(projectManager.greeting());
    }
    ……
}
```

首先对 ClassPathXmlApplicationContext 对象通过客户端配置文件进行实例化，然后调用这个实例化对象的 getBean 方法，返回 SEI。后面的事情就比较容易解决了。

客户端程序的配置文件如下：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://cxf.apache.org/jaxws
      http://cxf.apache.org/schemas/jaxws.xsd">

    <jaxws:client id="ProjectManagerClient"
        serviceClass="demo.cxf.webservices.ProjectManager" address="jms://">
        <jaxws:features>
            <bean class="org.apache.cxf.transport.jms.JMSConfigFeature"
                p:jmsConfig-ref="jmsConfig" />
        </jaxws:features>
    </jaxws:client>
    <bean id="jmsConfig"
        class="org.apache.cxf.transport.jms.JMSConfiguration"
        p:connectionFactory-ref="jmsConnectionFactory"
        p:targetDestination="test.cxf.jmstransport.queue" />
    <bean id="jmsConnectionFactory"
        class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL" value="tcp://localhost:61616" />
    </bean>
</beans>
```

说明如下：主要是描述 jaxws:client 的 Bean 定义，该 Bean 是 demo.cxf.webservices.

ProjectManager。其 features 属性是 org.apache.cxf.transport.jms.JMSConfigFeature。配置文件类是 org.apache.cxf.transport.jms.JMSConfiguration，而 org.apache.cxf.transport.jms.JMSConfiguration 的连接 Bean 是 org.apache.activemq.ActiveMQConnectionFactory。

4. 测试和运行结果

运行本程序的过程分为三个阶段：

第 1 步是启动 JMS 消息组件程序，即运行 MessageBroker.java 文件程序。

第 2 步是启动 Tomcat 服务组件程序。

第 3 步是运行客户端程序，即运行 Client.java 文件程序。

运行客户端程序，其客户端程序和服务器端输出结果与前面相同。

5. 其他内容

上述的所有调用都基于 JAX-WS 规范。

本例代码参见配套资源代码中“第 08 章\transport_jms_servlet”文件夹。

8.4 Local 传输协议

本节先介绍 Local 传输协议的基础知识，然后说明 Apache CXF 框架如何实现 Local 传输协议。最后通过三个例子程序来进行进一步说明。

8.4.1 Apache CXF 的 Local 传输协议介绍

Apache CXF 框架支持 Local 传输协议。开发人员可以利用 Local 传输协议在一个单独的 JVM 内发送消息或信息。该消息首先被序列化，并通过管道模式从一个服务器端点到另一个服务器端点。结构如图 8-12 所示。

Local 传输协议采用的链接为 local://{端点名称（endpoint_name）}，其中端点名称可以支持任何字符集。这个情形看起来有点类似本地 EJB 调用。开发人员只需定义 local://就可以实现本地传输协议。

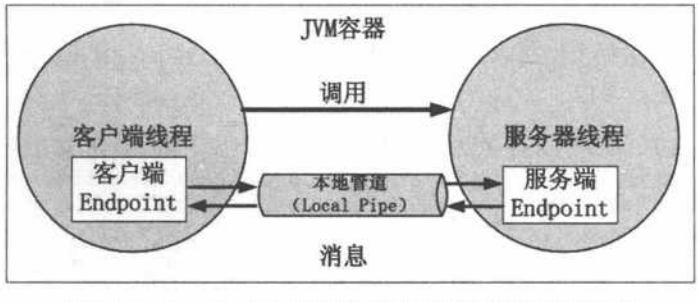


图 8-12　Apache CXF 支持本地传输协议的原理结构图

8.4.2 CXF 的 Local 配置和使用

Apache CXF 的 Local 配置和使用包括基于 JAX-WS 的使用和简化模式（Simple Frontend）的使用。

1. 基于 JAX-WS 的使用

此代码显示了如何发布一个本地服务器端点：

```
import javax.xml.ws.Endpoint;

HelloServiceImpl serverImpl = new HelloServiceImpl();
Endpoint.publish("local://hello", serverImpl);
```

或使用 XML，代码如下：

```
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:simple="http://cxf.apache.org/simple"
     xmlns:soap="http://cxf.apache.org/bindings/soap"
     xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://cxf.apache.org/bindings/soap
http://cxf.apache.org/schemas/configuration/soap.xsd
http://cxf.apache.org/simple
http://cxf.apache.org/schemas/simple.xsd">

  <import resource="classpath:META-INF/cxf/cxf.xml"/>
  <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml"/>

  <bean class="org.apache.cxf.transport.local.LocalTransportFactory" lazy-init="false">
     <property name="transportIds">
        <list>
           <value>http://cxf.apache.org/transports/local</value>
           <value>http://schemas.xmlsoap.org/soap/http</value>
           <value>http://schemas.xmlsoap.org/wsdl/soap/http</value>
        </list>
     </property>
  </bean>

<jaxws:endpoint name="helloWorld" address="local://hello"
implementor="org.example.HelloServiceImpl"/>
</beans>
```

2. 采用简化模式（Simple Frontend）

使用本地传输协议之前，开发人员需要在总线的 LocalTransportFactory 注册默认 SOAP 的 transportURI，代码如下：

```
Bus bus = BusFactory.getDefaultBus();
LocalTransportFactory localTransport = new LocalTransportFactory();
dfm.registerDestinationFactory("http://schemas.xmlsoap.org/soap/http",
localTransport);
dfm.registerDestinationFactory("http://schemas.xmlsoap.org/wsdl/soap/http",
localTransport);
dfm.registerDestinationFactory("http://cxf.apache.org/bindings/xformat",
localTransport);
dfm.registerDestinationFactory("http://cxf.apache.org/transports/local",
localTransport);

ConduitInitiatorManager extension = bus.getExtension(ConduitInitiatorManager.class);
extension.registerConduitInitiator("http://cxf.apache.org/transports/local",
localTransport);
```

```
extension.registerConduitInitiator("http://schemas.xmlsoap.org/wsdl/soap/http",
localTransport);
extension.registerConduitInitiator("http://schemas.xmlsoap.org/soap/http",
localTransport);
extension.registerConduitInitiator("http://cxf.apache.org/bindings/xformat",
localTransport);
```

也可以通过使用 local://地址的方式获得服务器和客户端工厂的 Bean，其服务器端代码如下：

```
import org.apache.cxf.frontend.ServerFactoryBean;

ServerFactoryBean sf = new ServerFactoryBean();
sf.setAddress("local://hello");
sf.setServiceBean(new HelloServiceImpl());
sf.setServiceClass(HelloService.class); // Optionally specify the service interface
sf.create();
```

其客户端代码如下：

```
import org.apache.cxf.frontend.ClientProxyFactoryBean;

ClientProxyFactoryBean cf = new ClientProxyFactoryBean();
cf.setAddress("local://hello");
cf.setServiceClass(HelloService.class); // Optionally specify the service interface
HelloService h = (HelloService) cf.create();
```

8.4.3　实现简化前端 Local 传输的例子

本程序演示 Apache CXF 框架采用 Local 传输来实现客户端和服务器端的相互调用，客户端采用 Apache CXF 的代码优先（Java First）前端应用模式。在这种模式下，采用 POJO 对象作为中间传输对象。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询和删除操作。

在这个过程中，前端客户端采用 Apache CXF 自身规范的 ClientProxyFactoryBean 对象来实现，后台服务端为遵循 Apache CXF 自身规范的 ServerFactoryBean 对象来实现。采用本地传输方式，实现的结构如图 8-13 所示。

其中：① 表示客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。在客户端和服务端的 SEI 都是 ProjectManager 接口。

在编写本程序例子时，主要针对第 6 章“采用 Apache CXF 简化前端实现”案例进行比较。

2. 应用程序案例实现步骤

要创建 Local 通信的前端实现 Web Services 程序的流程步骤如图 8-14 所示。

流程图说明，要实现以下步骤：

（1）创建一个 Java 项目。

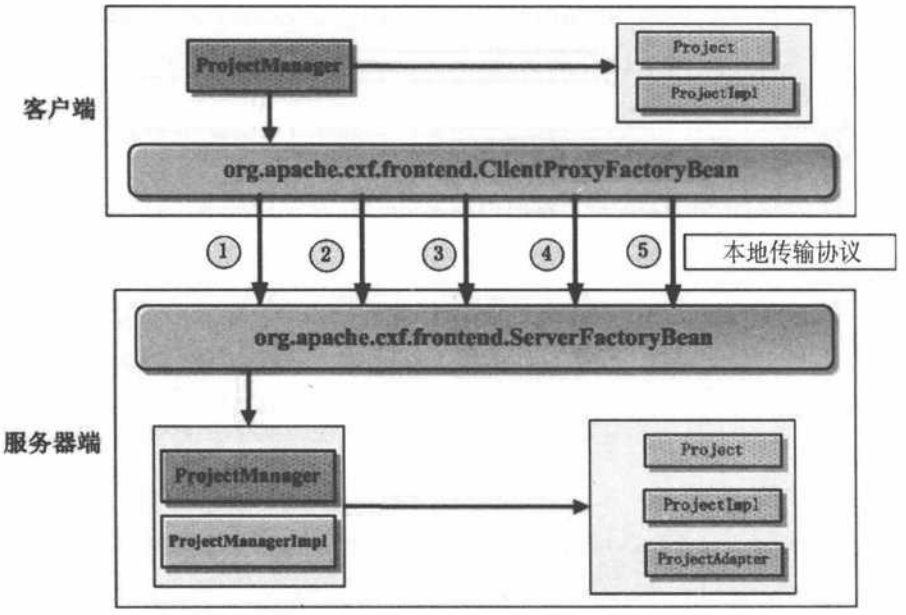


图 8-13　基于 Local 实现简化前端应用模式图

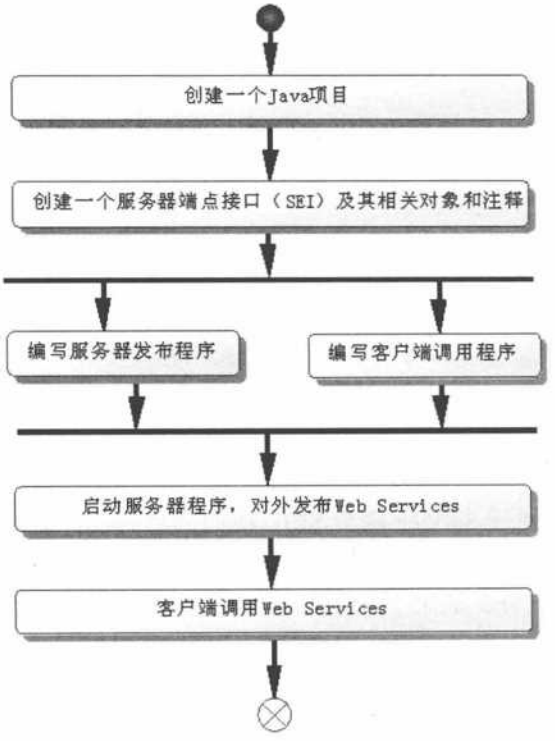


图 8-14　应用程序案例实现步骤流程图

（2）创建一个服务器端点接口（SEI）及其相关对象和注释。

（3）编写发布这个 Java 程序为 Web Services 的服务器端程序。

（4）编写 Web Services 的客户端应用程序。

（5）启动服务器程序，对外发布 Web Services。

（6）客户端调用 Web Services。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 8-15 所示。

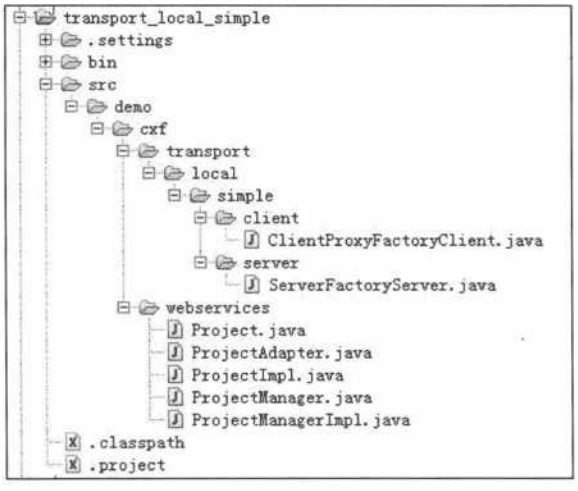


图 8-15　transport_local_simple 程序结构

包目录包括三个部分，第一个包目录 demo.cxf.transport.local.simple.server 是服务器程序目录。第二个包目录 demo.cxf.transport.local.simple.client 是客户端程序目录。第三个包目录 demo.cxf.webservices 是服务器端点接口 SEI 和相关对象等公用部分，服务器端和客户端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口，服务器端和客户端都必须引用。但服务器端和客户端对公用部分的引用是不同的。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口。而服务器端程序是接口和实现类都要引用。

本案例程序与第 6 章“采用 Apache CXF 简化前端实现”相比，只有客户端和服务器端的传输协议不同，其余的内容全部相同。第 6 章“采用 Apache CXF 简化前端实现”采用的是 HTTP 传输协议，而本程序采用的是本地传输协议，下面只列出有区别的两个文件的代码。

下面分别介绍两部分：①服务器端发布程序；②客户端调用程序。

（1）服务器发布程序

第一个文件是服务器发布程序，即 ServerFactoryServer.java，代码如下：

```
package demo.cxf.transport.local.simple.server;

import org.apache.cxf.frontend.ServerFactoryBean;
import demo.cxf.webservices.*;

public class ServerFactoryServer {
    public ServerFactoryServer() {
        ProjectManager projectManager = new ProjectManagerImpl();
        ServerFactoryBean svrFactory = new ServerFactoryBean();
```

```
        svrFactory.setAddress("local://ProjectManager");
        svrFactory.setServiceClass(ProjectManager.class);
        svrFactory.setServiceBean(projectManager);
        svrFactory.create();
    }

    public static void main(String[] args) throws Exception {
        new ServerFactoryServer();
        System.out.println("ServerFactoryBean Server ready...");

        if (!"inProcess".equals(args[0])) {
            Thread.sleep(5 * 60 * 1000);
            System.out.println("Server exiting");
            System.exit(0);
        }
    }
}
```

其中黑体字说明了服务器采用的是 Local 传输协议来发布程序。

（2）客户端调用程序

第二个文件是客户端访问程序，即 ClientProxyFactoryClient.java，代码如下：

```
package demo.cxf.transport.local.simple.client;

import org.apache.cxf.frontend.ClientProxyFactoryBean;
……
import demo.cxf.transport.local.simple.server.ServerFactoryServer;

public class ClientProxyFactoryClient {
    public ClientProxyFactoryClient() {}

    public static void main(String[] args) throws Exception {
        ServerFactoryServer.main(new String[] { "inProcess" });
        ClientProxyFactoryBean factory = new ClientProxyFactoryBean();
        if (args != null && args.length > 0 && !"".equals(args[0])) {
            factory.setAddress(args[0]);
        } else {
            factory.setAddress("local://ProjectManager");
        }
        factory.setServiceClass(ProjectManager.class);
        ProjectManager projectManager = (ProjectManager) factory.create();

        // 项目的基本情况
        getGreet(projectManager);
        greetingWithParameter(projectManager);
        ……
        System.exit(0);
    }

    // 调用 SEI 的 getGreet 方法
    private static void getGreet(ProjectManager projectManager)
            throws Exception {
        System.out.println(projectManager.greeting());
    }
    ……
}
```

其中黑体字说明了客户端采用的是 Local 传输协议来访问程序。

4. 测试和运行结果

由于在客户端已经启动了服务器程序，所以直接运行客户端程序，其客户端程序和服务器端输出结果与第 6 章“采用 Apache CXF 简化前端实现”案例相同。

5. 其他内容

上述的所有调用都基于 Apache CXF 自身 Web Services 规范。

本例代码参见配套资源代码中“第 08 章\transport_local_simple”文件夹。

8.4.4　实现 JAX-WS 规范并采用 Local 传输的例子

本程序演示 Apache CXF 遵循 JAX-WS 规范并采用 Local 传输的客户端和服务器端调用，客户端采用 Apache CXF 的代码优先（Java First）前端应用模式。采用 POJO 对象作为中间传输对象。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，都使用 POJO 对象实现客户端和服务器端的数据交换。后台服务端为遵循 JAX-WS 规范的 Endpoint 实现。采用本地传输方式。前端访问采用两种方式，一种是 JAX-WS 规范的 Service 方式，实现的结构如图 8-16 所示。

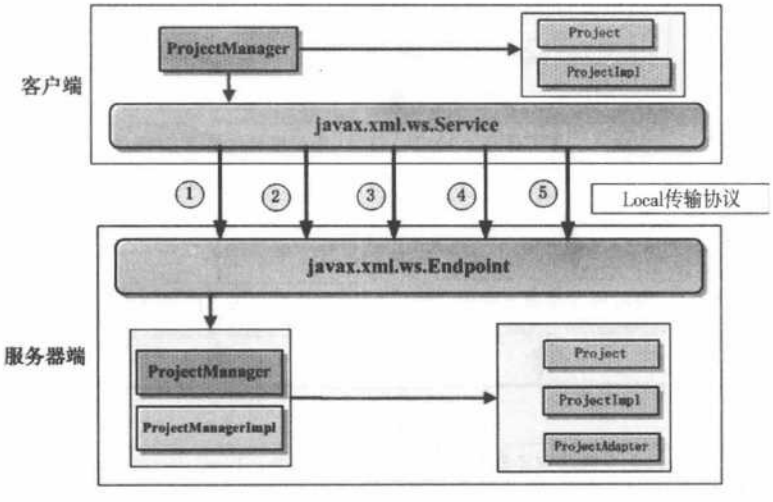


图 8-16　基于本地实现 JAX-WS 前端应用模式图

还有一种方式是采用 Apache CXF 本身的客户端来实现，实现的结构如图 8-17 所示。

其中：①表示客户端通过 Service 获得 SEI，访问后台的项目基本情况；②表示客户端通过 SEI 修改默认项目；③表示客户端通过 SEI 增加一个项目；④表示客户端通过 SEI 删除一个项目；⑤表示客户端通过 SEI 查看后台的全部项目情况。在客户端和服务器端的 SEI 都是 ProjectManager 接口。

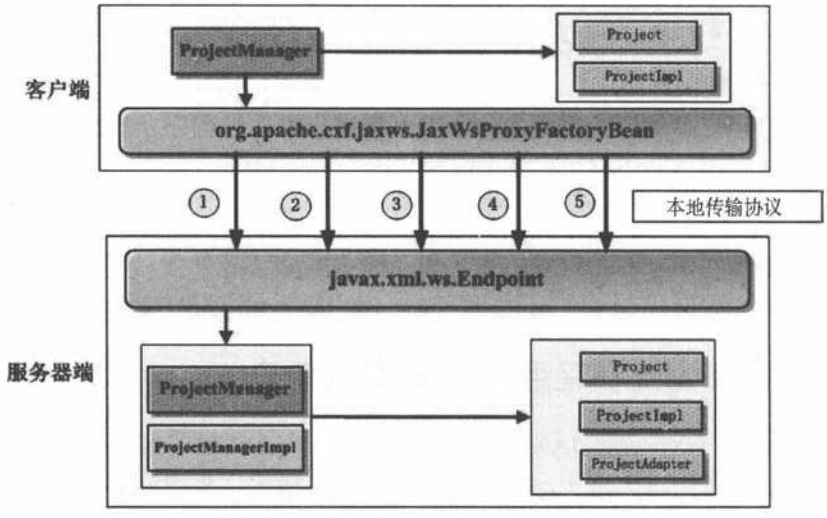


图 8-17　基于 POJO 实现 Apache CXF 前端应用模式图

在编写本程序例子时，主要针对第 6 章“采用 POJO 并基于 JAX-WS 服务的前端实现”案例进行比较。

2. 应用程序案例实现步骤

要创建 Local 通信并基于 JAX-WS 服务的前端实现 Web Services 程序的流程步骤如图 8-18 所示。

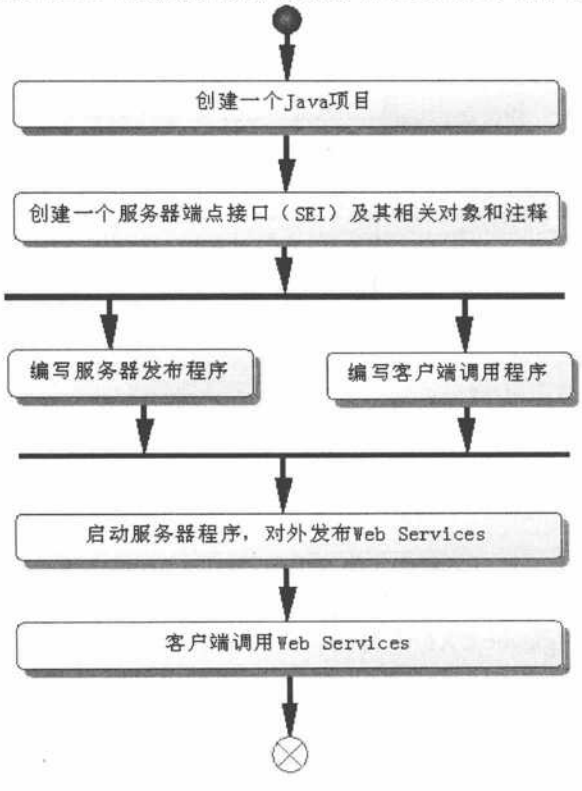


图 8-18　应用程序案例实现步骤流程图

流程图说明，要实现以下步骤：

（1）创建一个 Java 项目。

（2）创建一个服务器端点接口（SEI）及其相关对象和注释。

（3）编写发布这个 Java 程序为 Web Services 的服务器端程序。

（4）编写 Web Services 的客户端应用程序。

（5）启动服务器程序，对外发布 Web Services。

（6）客户端调用 Web Services。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 8-19 所示。

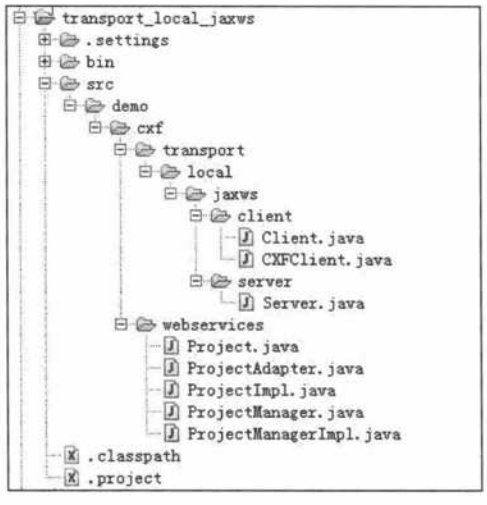


图 8-19　transport_local_jaxws 程序结构

包目录包括三个部分，第一个包目录 demo.cxf.transport.local.jaxws.server 是服务器程序目录。第二个包目录 demo.cxf.transport.local.jaxws.client 是客户端程序目录。第三个包目录 demo.cxf.webservices 是服务器端点接口 SEI 和相关对象等公用部分，服务器端和客户端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口，服务器端和客户端都必须引用。但服务器端和客户端对公用部分的引用是不同的。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口。而服务器端程序是接口和实现类都要引用。

与第 6 章“采用 POJO 并基于 JAX-WS 服务的前端实现”案例相比，只有客户端和服务端的传输协议不同，其余的内容全部相同。第 6 章“采用 POJO 并基于 JAX-WS 服务的前端实现”案例采用的是 HTTP 传输协议，而本程序采用的是 Local 传输协议，下面只列出有区别的两个文件的代码。

下面分别介绍两部分：①服务器端发布程序；②客户端调用程序。

（1）服务器端发布程序

第一个文件是服务器发布程序，即 Server.java，代码如下：

```
package demo.cxf.transport.local.jaxws.server;

import javax.xml.ws.Endpoint;
import demo.cxf.webservices.*;

public class Server {

    public Server() {
        System.out.println("Starting Server");
        ProjectManager projectManager = new ProjectManagerImpl();
        String address = "local://ProjectManager";
```

```
        Endpoint.publish(address, projectManager);
    }

    public static void main(String[] args) throws Exception {
        new Server();
        System.out.println("Server ready...");

        if (!"inProcess".equals(args[0])) {
            Thread.sleep(5 * 60 * 1000);
            System.out.println("Server exiting");
            System.exit(0);
        }
    }
}
```

（2）客户端调用程序

客户端调用程序有两种实现方式，第一种是基于 JAX-WS 的客户端实现模式，第二种是采用 Apache CXF 客户端实现模式。

JAX-WS 实现模式的客户端程序 Client.java 的代码如下：

```
package demo.cxf.transport.local.jaxws.client;

import javax.xml.namespace.QName;
……
import demo.cxf.transport.local.jaxws.server.Server;

public class Client {
    private static final QName SERVICE_NAME = new QName(
            "http://webservices.cxf.demo/", "ProjectManager");
    private static final QName PORT_NAME = new QName(
            "http://webservices.cxf.demo/", "ProjectManagerPort");
    private static String endpointAddress = "local://ProjectManager";

    public Client() {}

    public static void main(String[] args) throws Exception {
        Server.main(new String[] { "inProcess" });
        //用服务名称创建一个服务
        Service service = Service.create(SERVICE_NAME);
        //用服务端口名称、绑定方式和 Endpoint 地址来创建服务器端口
        service.addPort(PORT_NAME, SOAPBinding.SOAP11HTTP_BINDING, endpointAddress);

       //获得 SEI 接口
        ProjectManager projectManager = service.getPort(ProjectManager.class);
        System.out.println(projectManager.getProjectName());

        //项目的基本情况
        getGreet(projectManager);
        greetingWithParameter(projectManager);
        ……
        System.exit(0);
    }

    //调用 SEI 的 getGreet 方法
    private static void getGreet(ProjectManager projectManager)throws Exception {
```

```
        System.out.println(projectManager.greeting());
    }
    ……
}
```

其中黑体字说明了客户端采用的是本地传输协议来访问程序。

上述的所有调用都基于 JAX-WS 规范。下面我们采用 Apache CXF 框架的前端编程模式来实现调用服务。

Apache CXF 实现模式的客户端程序 CXFClient.java 的代码如下：

```
package demo.cxf.transport.local.jaxws.client;

import org.apache.cxf.jaxws.JaxWsProxyFactoryBean;
……
import demo.cxf.transport.local.jaxws.server.Server;

public class CXFClient {
    public CXFClient() {}

    public static void main(String[] args) throws Exception {
        Server.main(new String[] { "inProcess" });
        // 定义 JaxWs 的代理工厂
        JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();

        // 设置 JaxWs 代理工厂的属性，包括服务类和地址
        factory.setServiceClass(ProjectManager.class);
        factory.setAddress("local://ProjectManager");

        // 获取服务类的对象并处理
        ProjectManager projectManager = (ProjectManager) factory.create();

        // 项目的基本情况
        getGreet(projectManager);
        greetingWithParameter(projectManager);
        getProjectName(projectManager);
        ……
        System.exit(0);
    }

    // 调用 SEI 的 getGreet 方法
    private static void getGreet(ProjectManager projectManager)throws Exception {
        System.out.println(projectManager.greeting());
    }
    ……
}
```

其中黑体字说明了客户端采用的是本地传输协议来访问程序。

4. 测试和运行结果

由于在客户端已经启动了服务器程序，所以直接运行客户端程序，其客户端程序和服务器端输出结果与前面相同。

5. 其他内容

上述的调用有基于 JAX-WS 规范，也有基于 Apache CXF 自身 Web Services 规范。

本例代码参见配套资源代码中“第 08 章\transport_local_jaxws”文件夹。

8.4.5 Spring 注入实现 JAX-WS 规范并采用 Local 传输的例子

本程序演示 Apache CXF 遵循 JAX-WS 规范并采用 Local 传输的客户端和服务器端调用，客户端采用 Apache CXF 的代码优先（Java First）前端应用模式。在客户端和服务器端都采用 Spring IoC 注入来实现。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，后台服务器端为遵循JAX-WS规范的 Endpoint 来实现，采用 Spring IoC 注入方式。采用本地传输方式。前端访问直接采用 Spring IoC 注入方式，实现的结构如图 8-20 所示。

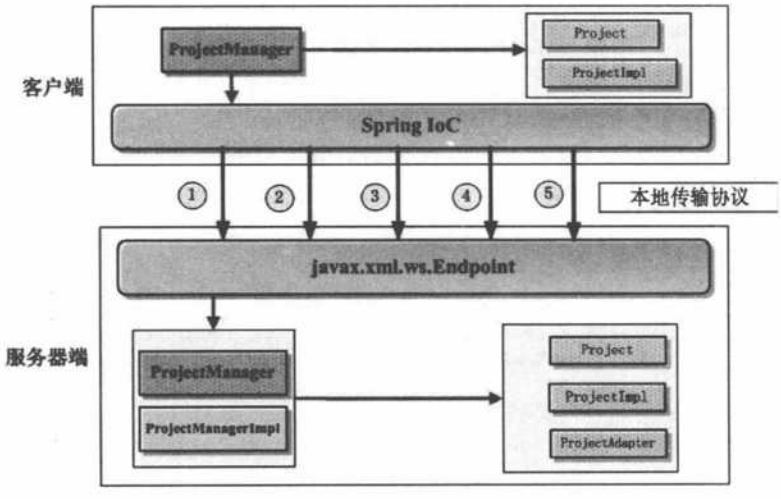


图 8-20 基于 Local 采用 Spring 注入实现应用模式图

其中：①表示客户端通过 Service 获得 SEI，访问后台的项目基本情况；②表示客户端通过 SEI 修改默认项目；③表示客户端通过 SEI 增加一个项目；④表示客户端通过 SEI 删除一个项目；⑤表示客户端通过 SEI 查看后台的全部项目情况。在客户端和服务端的 SEI 都是 ProjectManager 接口。

在编写本程序例子时，主要针对实现 JAX-WS 规范并采用 Local 传输的例子进行比较。

2. 应用程序案例实现步骤

要创建采用 Spring 注入的 Local 实现 Web Services 程序的流程步骤如图 8-21 所示。

流程图说明，要实现以下步骤：

（1）创建一个 Java 项目。

（2）创建一个服务器端点接口（SEI）及其相关对象和注释。

（3）编写发布这个 Java 程序为 Web Services 的服务器端程序和 Spring 注入配置文件。

（4）编写 Web Services 的客户端应用程序和 Spring 注入配置文件。

（5）启动服务器程序，对外发布 Web Services。

（6）客户端调用 Web Services。

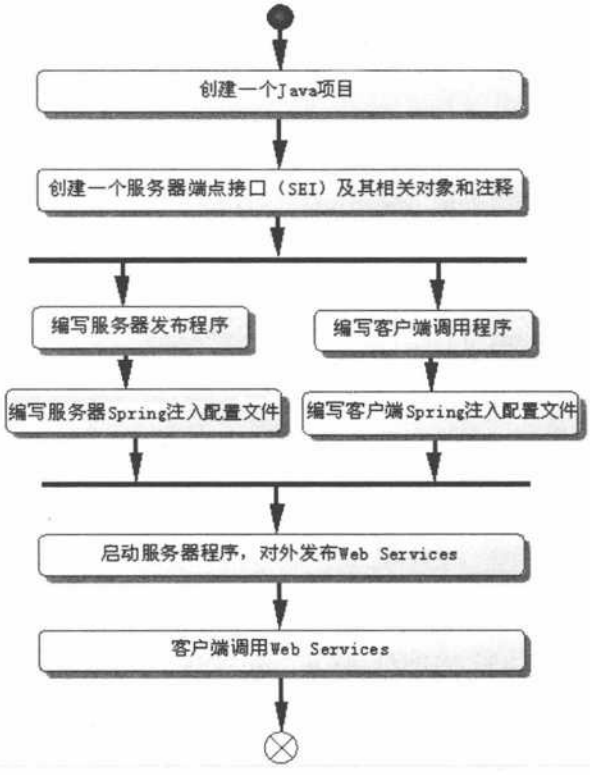


图 8-21　应用程序案例实现步骤流程图

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 8-22 所示。

包目录包括三个部分，第一个包目录 demo.cxf.transport.local.server 是服务器程序目录。第二个包目录 demo.cxf.transport.local.client 是客户端程序目录。第三个包目录 demo.cxf.webservices 是服务器端点接口 SEI 和相关对象等公用部分，服务器端和客户端都有可能引用其内容。

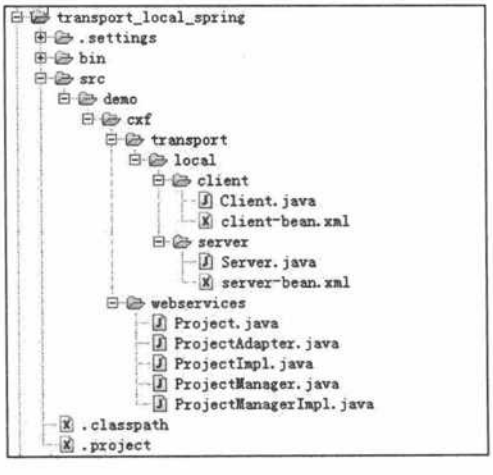


图 8-22　transport_local_spring 程序结构

服务器端点接口（SEI）是公用部分的接口，服务器端和客户端都必须引用。但服务器端和客户端对公用部分的引用是不同的。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口。而服务器端程序是接口和实现类都要引用。

与本章“实现 JAX-WS 规范并采用本地传输的例子”实现相比，只有客户端和服务器端的实现方式不同，其余的内容全部相同。本章“实现 JAX-WS 规范并采用本地传输的例子”直接

调用来实现，而本案例采用 Spring IoC 注入方式实现，下面只列出有区别的两个文件的代码。

下面分别介绍：①服务器端发布程序；②客户端调用程序。

（1）服务器端发布程序

第一个文件是服务器发布程序，即 Server.java，代码如下：

```
package demo.cxf.transport.local.server;

import javax.xml.ws.Endpoint;
import org.apache.cxf.Bus;
import org.apache.cxf.bus.spring.SpringBusFactory;
import demo.cxf.webservices.*;

public class Server {
    public Server() {
        ProjectManager projectManager = new ProjectManagerImpl();
        String address = "local://ProjectManager";
        SpringBusFactory factory = new SpringBusFactory();
        Bus bus = factory.createBus("demo/cxf/transport/local/server/server-bean.xml");
        factory.setDefaultBus(bus);
        Endpoint.publish(address, projectManager);
    }

    public static void main(String[] args) throws Exception {
        new Server();
        System.out.println("Server ready...");

        if (!"inProcess".equals(args[0])) {
            Thread.sleep(5 * 60 * 1000);
            System.out.println("Server exiting");
            System.exit(0);
        }
    }
}
```

服务器注入的 Bean 配置文件——server-bean.xml：

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:simple="http://cxf.apache.org/simple"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xmlns:soap="http://cxf.apache.org/bindings/soap"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://cxf.apache.org/bindings/soap
        http://cxf.apache.org/schemas/configuration/soap.xsd
        http://cxf.apache.org/simple
        http://cxf.apache.org/schemas/simple.xsd
        http://cxf.apache.org/jaxws
        http://cxf.apache.org/schemas/jaxws.xsd">

    <import resource="classpath:META-INF/cxf/cxf.xml" />
    <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />

    <bean class="org.apache.cxf.transport.local.LocalTransportFactory"
```

```
        lazy-init="false">
        <property name="transportIds">
            <list>
                <value>http://cxf.apache.org/transports/local</value>
                <value>http://schemas.xmlsoap.org/soap/http</value>
                <value>http://schemas.xmlsoap.org/wsdl/soap/http</value>
            </list>
        </property>
    </bean>

    <jaxws:endpoint id="ProjectManager" implementor="demo.cxf.webservices.ProjectManagerImpl" />
</beans>
```

（2）客户端调用程序

在 JAX-WS 实现模式下，客户端程序 Client.java 的代码如下：

```
package demo.cxf.transport.local.client;

import org.springframework.context.support.ClassPathXmlApplicationContext;
……
import demo.cxf.transport.local.server.Server;

public class Client {
    public Client() {}

    public static void main(String[] args) throws Exception {
        Server.main(new String[] { "inProcess" });
        ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(
                new String[] { "demo/cxf/transport/local/client/client-bean.xml" });
        ProjectManager projectManager = (ProjectManager) context.getBean
("ProjectManagerClient");

        // 项目的基本情况
        getGreet(projectManager);
        greetingWithParameter(projectManager);
        ……
        System.exit(0);
    }

    // 调用 SEI 的 getGreet 方法
    private static void getGreet(ProjectManager projectManager)
            throws Exception {
        System.out.println(projectManager.greeting());
    }
    ……
}
```

客户端注入的 Bean 配置文件——client-bean.xml：

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                    http://www.springframework.org/schema/beans/spring-beans.xsd
                    http://cxf.apache.org/jaxws
                    http://cxf.apache.org/schemas/jaxws.xsd">
    <jaxws:client id="ProjectManagerClient"
        serviceClass="demo.cxf.webservices.ProjectManager"
```

```
        address="local://ProjectManager" />
</beans>
```

4. 测试和运行结果

由于在客户端已经启动了服务器程序，所以直接运行客户端程序，其客户端程序和服务器端输出结果与本章“实现 JAX-WS 规范并采用 Local 传输的例子”输出相同。

5. 其他内容

上述的所有调用都基于 JAX-WS 规范。

本例代码参见配套资源代码中“第 08 章\transport_local_spring”文件夹。

第9章 CXF的配置、调试和日志

本章主要说明Apache CXF框架的配置、日志管理和调试管理。首先介绍Apache CXF框架的配置，其次描述Apache CXF框架的日志管理，最后简单说明一下Apache CXF框架的调试管理。

本章内容：

1. CXF的配置。
2. CXF的日志管理。
3. CXF的调试管理。

9.1 CXF的配置

Apache CXF框架的配置管理包括配置文件、Bus配置、Feature列表和JMX管理。

9.1.1 CXF配置概述

Apache CXF框架配置主要说明Apache CXF的配置文件、文件类型和配置文件里的内容，包括如何使用配置文件。

1. CXF提供的配置文件

Apache CXF框架在Web Services应用程序启动过程中，可以在程序的服务器端和客户端自动地搜寻开发人员配置的XML文件。一般情况下，Apache CXF框架会先寻找一个默认的基本配置目录，正常情况下是“/cxf.xml”路径。同时，如果应用程序运行在一个Servlet容器里，配置文件一般都会放置在Web应用程序下/WEB-INF/classes文件夹中。

如果开发人员希望修改默认位置，可用一个命令行来指定文件的默认属性：-Dcxf.config.file=some_other_config.xml。自定义配置文件也将放置在此类文件目录中。

如果用URL作为配置位置，要求采用如下的指定模式：-Dcxf.config.file.url=config_file_url。

Apache CXF配置文件与Spring配置文件的XSD格式是一样的，因此所有的配置文件都按照下面的XML格式来表示：

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="
```

```
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

<!-- Configuration goes here! -->

</beans>
```

如果不熟悉 Spring，同时也不希望更多地了解 Spring，上面 XML 格式的实现也是非常容易的。实际上，在配置文件中，仅仅<beans>元素是与 Spring 相关的。故只需简单地创建这个配置文件并放置在应用程序的 classpath 中，同时添加一个需要配置的组件内容即可实现配置。

2. 配置文件的类型

Apache CXF 框架的配置文件可以是客户端配置文件，也可以是服务器配置文件。下面分别介绍这两种配置文件。

（1）客户端配置文件

Web Services 客户端配置 cxf.xml 文件（或相关的其他名称的文件）的主要功能是实现 Web Services 客户端的特定功能。例如，下面客户端的 cxf.xml 配置文件说明，在特定的请求和响应服务中关闭分块转让编码：

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
      xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration
         http://cxf.apache.org/schemas/configuration/http-conf.xsd
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd">

  <http-conf:conduit
         name="{urn:ebay:apis:eBLBaseComponents}Shopping.http-conduit">
     <http-conf:client AllowChunking="false"/>
  </http-conf:conduit>
</beans>
```

（2）服务器配置文件

通常，Web Services 服务端的 cxf.xml 配置文件主要是以 Apache CXF 框架总线为主来进行配置的，因为 Apache CXF 总线可以创建所有服务（Service）和端点（Endpoint）。端点（Endpoint）配置主要是通过 WEB-INF/cxf-servlet.xml 下一个专有文件或一个 web.xml 文件（Web 应用程序部署描述符）中 Spring 应用上下文指定文件进行。cxf-servlet.xml 文件装载比较慢，因为加载 cxf-servlet.xml 前要装入所有可能的 CXF 的端点模块；而采用 Spring 应用的配置文件装载起来比较快，因为 Spring 只允许装载指定的并且是必需的 CXF 模块。

如果要引入一个 cxf-servlet.xml 的配置文件，开发人员需要在 web.xml 部署描述符中的 init-param 元素录入一个装载 CXF 的文件名称。（注意，如果使用的配置文件名称是“cxf-servlet.xml”，那就没有必要使用 init-param 元素。因为 cxf-servlet.xml 是 CXF 搜寻的默认文件名称。）

如果对于一个基于 Spring 应用上下文端点的应用引入配置文件。开发人员可以看到 web.xml 文件显式地引用 beans.xml 应用程序上下文的文件，该文件通过 context-param 元素（和

ContextLoaderListener 对象）来进行配置明细。Spring 应用程序上下文需要手动引入三个 CXF 的配置模块文件。

3. 配置文件要求配置的内容和如何配置

如果开发人员希望改变 Apache CXF 框架的默认行为，可以使用一些特定的功能或微调一个组件的行为。只需提供一个 Spring 配置文件而无须编写一行代码。

在某些情况下，扩展 WSDL 文档实现可以达到同一目的。下面情况也是有可能的：可以针对 wsdl:port 元素来添加 Apache CXF 框架的特定扩展功能，并采用这种方式来调整指定的传输协议。或者可以使用 WS-Policy 元素来迫使应用程序使用 WS-Addressing 规范。

实现特殊的目标，最通用的方法是使用 Spring 配置文件：通过在 Spring 配置文件上进行简单的配置就可以实现或启用一些 Feature 功能。

（1）可以使用任何通用的 Spring bean 元素或 CXF 自定义类元素来设置运行组件的属性。

（2）当运行时进行动态修改（实际上是以修改程序代码的方式来实现）。

（3）下面的例子说明了 Spring 配置文件的内容，其配置的目的是实现日志记录和输出故障信息。

使用普通的 Spring bean 元素来启用消息日志记录，下面是配置文件的配置代码内容：

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

   <bean id="logInbound" class="org.apache.cxf.interceptor.LoggingInInterceptor"/>
   <bean id="logOutbound"
     class="org.apache.cxf.interceptor.LoggingOutInterceptor"/>

   <bean id="cxf" class="org.apache.cxf.bus.CXFBusImpl">
      <property name="inInterceptors">
         <list>
            <ref bean="logInbound"/>
         </list>
      </property>
      <property name="outInterceptors">
         <list>
            <ref bean="logOutbound"/>
         </list>
      </property>
      <property name="outFaultInterceptors">
         <list>
            <ref bean="logOutbound"/>
         </list>
      </property>
   </bean>
</beans>
```

在这个例子中，指定由类 org.apache.cxf.bus.CXFBusImplApache 来实现 CXF 的 Bus 组件，它的 ID 是“cxf”，这是默认的。如果开发人员希望 Bus 组件通过日志记录拦截器来处理所有客户端和服务器端点的出站和入站拦截器链，必须重新循环 Bus 组件的这个行动。

使用 Apache CXF 框架自定义 bean 元素来启用消息日志记录，其配置文件如下：

```
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:cxf="http://cxf.apache.org/core"
     xsi:schemaLocation="
http://cxf.apache.org/core
http://cxf.apache.org/schemas/core.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

   <bean id="logInbound" class="org.apache.cxf.interceptor.LoggingInInterceptor"/>
   <bean id="logOutbound" class="org.apache.cxf.interceptor.LoggingOutInterceptor"/>

   <cxf:bus>
      <cxf:inInterceptors>
         <ref bean="logInbound"/>
      </cxf:inInterceptors>
      <cxf:outInterceptors>
         <ref bean="logOutbound"/>
      </cxf:outInterceptors>
      <cxf:inFaultInterceptors>
         <ref bean="logOutbound"/>
      </cxf:inFaultInterceptors>
   </cxf:bus>
</beans>
```

在这里，不需要指定 bus 实现类。inInterceptors、outInterceptors 和 inFaultInterceptors 子元素都是类型列表。所有这些信息都是嵌入在基础 schema 和 bean 定义提供给<cxf:bus> 元素来调用。请注意必须指定<beans>元素的 schemaLocation 属性模式位置，使用 Spring 模式可以验证配置文件。下一个例子是一个更简单的配置。

可使用 Feature 功能中设置日志的功能，下面的配置是启用消息日志记录：

```
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:cxf="http://cxf.apache.org/core"
     xsi:schemaLocation="
http://cxf.apache.org/core
http://cxf.apache.org/schemas/core.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

   <cxf:bus>
      <cxf:features>
         <cxf:logging/>
      </cxf:features>
   </cxf:bus>
</beans>
```

9.1.2 Bus 配置

总线是 Apache CXF 的架构骨干。总线管理可以扩展一些功能并充当一个拦截器提供者。装载在总线上的拦截器将被添加到各自的入站消息拦截器链、出站消息拦截器链或故障拦截器链。

这些拦截器链都是被所有客户端和服务器端点上总线在其范围内创建的。在默认情况下，应用程序都不通过编写拦截器对象来实现，而是通过配置文件的配置或使用编写 Java 代码来添加拦截器，如下所示。

注意：这些都是具体配置的端点。

1. 使用配置文件

在下面例子中，在 Spring 配置文件简单地定义 Bus 元素，并根据需要添加子元素，例如：

```
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:cxf="http://cxf.apache.org/core"
     xsi:schemaLocation="
        http://cxf.apache.org/core
        http://cxf.apache.org/schemas/core.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

   <bean id="logOutbound"
     class="org.apache.cxf.interceptor.LoggingOutInterceptor"/>

   <cxf:bus>
      <cxf:outInterceptors>
         <ref bean="logOutbound"/>
      </cxf:outInterceptors>
   </cxf:bus>
</beans>
```

该总线的配置属性由 schema 中的 http://cxf.apache.org/schemas/core.xsd 来进行定义。

确定命名空间<beans>元素中的 schemaLocation 属性架构下的 URI 对，如 http://cxf.apache.org/core 等。

该<bus>元素支持的子元素如表 9-1 所示。

表 9-1　<bus>元素中子元素的名称和说明

名称	说明
inInterceptors	贡献到入站消息拦截器链的拦截器。一个<bean>或<ref>的列表
inFaultInterceptors	贡献到入站故障拦截器链的拦截器。一个<bean>或<ref>的列表
outInterceptors	贡献到出站消息拦截器链的拦截器。一个<bean>或<ref>的列表
outFaultInterceptors	贡献到出站故障拦截器链的拦截器。一个<bean>或<ref>的列表
features	总线上的功能（feature）应用。一个<bean>或<ref>的列表。这里可以了解到这些应用的功能（feature）

2. 配置使用 Java 代码

虽然通常不太方便，但拦截器也可以通过 Java 代码被添加到总线。给出一个 EndpointImpl 对象，拦截器就能加入到总线中。其实现的例子代码如下：

```
import javax.xml.ws.Endpoint;
import org.apache.cxf.interceptor.LoggingInInterceptor;
import org.apache.cxf.interceptor.LoggingOutInterceptor;
import org.apache.cxf.jaxws.EndpointImpl;
```

```
Object implementor = new GreeterImpl();
EndpointImpl ep = (EndpointImpl) Endpoint.publish("http://localhost/service",
implementor);

ep.getServiceFactory().getBus().getInInterceptors().add(new
LoggingInInterceptor());
ep.getServiceFactory().getBus().getOutInterceptors().add(new
LoggingOutInterceptor());
```

9.1.3 Features 列表

很多元素可以作为 Features 元素的子元素方式出现，并采用 http://org.apache.cxf/core 和 http://org.apache.cxf/jaxws 作为命名空间。

关于这部分内容详见第十四章的“14.1.3 CXF 的 Feature 列表”。

9.1.4 JMX 管理

JMX（Java Management Extensions）是一个为应用程序植入管理功能的框架。JMX 是一套标准的代理和服务，实际上，用户可以在任何 Java 应用程序中使用这些代理和服务来实现管理。这里介绍 Apache CXF 框架如何支持 JMX 功能。

1. JMX 介绍

JMX 是一种应用编程接口。JMX 也是一种可扩充对象和方法的集合体，可用于跨越一系列不同的异构操作系统平台、系统体系结构和网络传输协议，灵活地开发无缝集成的系统。JMX 服务管理应用提供了用户界面指导、Java 类和开发集成系统、网络及网络管理应用的规范。

管理对象是 JMX 应用程序的核心。JMX 结构包括：支持 Java 的 Web 浏览器用户接口、管理运行模块 ARM（Admin Runtime Module）和应用。这三个部件之间通过 RMI（Remote Method Invocation）进行通信。这里需要说明的是，RMI 可以实现：在一个 Java 虚拟机（JVM）上运行的程序可以调用远程服务器上另一个 JVM 的对象。

用户接口用来发布管理操作，这些操作可以间接地通过浏览器或通过单独的应用程序来激发。管理运行模块用来给应用提供实例化的管理对象。它包括 Agent 对象接口、通知接口和被管数据接口。应用指的是那些被管设备单元。

2. 配置 JMX 集成

为实现 Apache CXF 框架与 JMX 集成，注册一个继承 Apache CXF 框架中 Bus 组件的 InstrumentationManager 类。使用 Spring 配置 XML 文件，下面 XML 代码可以说明 Apache CXF 与 JMX 集成：

```
<bean id="org.apache.cxf.management.InstrumentationManager"
  class="org.apache.cxf.management.jmx.InstrumentationManagerImpl">
  <property name="bus" ref="cxf" />
  <property name="enabled" value="true" />
</bean>
```

默认 InstrumentationManager 接受的配置选项如表 9-2 所示。

表 9-2　默认InstrumentationManager接受的配置选项

名称	值	默认值
enabled	是否启用JMX集成	false
bus	CXF bus实例对象注册JMX扩展器	None
server	一个可选的参考。一个MBeanServer实例化对象注册MBean。如果不提供，一个MBeanServer使用“usePlatformMbeanServer”和/或“serverName”选项来解决问题	None
usePlatformMbeanServer	如果是true并且没有提供MBeanServer，JMX 扩展器在MBean服务器上注册MBeans参考平台	false
serverName	如果被设置，usePlatformMbeanServer是false，并且没有提供MBeanServer，JMX扩展器用该名称的MBean server来注册MBean	None
createMBServerConnectorFactory	如果是true，MBeanServer创建MBean连接器	true
threaded Determines	如果MBean连接器由这个线程或独立的线程来完成创建。当createMBServerConnectorFactory是true，这些线程才会相互关联起来	false
daemon Determines	如果MBean连接器创建线程被标记为一个守护线程。当createMBServerConnectorFactory是true，这些线程才会相互关联起来	false
JMXServiceURL	该连接器的网址在MBeanServer上创建。当createMBServerConnectorFactory是true，连接器和网址才会相互关联起来	service:jmx:rmi:///jndi/rmi://localhost:9913/jmxrmi

为了避免任何不必要的运行开销，在默认情况下测量响应时间的性能计数器是被禁用的。需要进一步配置才能启用该计数器。

实例配置，加入下面的 XML 信息到 CXF 的 Spring 上下文，即启用 JMX 集成：

```
<bean id="org.apache.cxf.management.InstrumentationManager"
  class="org.apache.cxf.management.jmx.InstrumentationManagerImpl">
  <property name="bus" ref="cxf" />
  <property name="enabled" value="true" />
  <property                          name="JMXServiceURL
value="service:jmx:rmi:///jndi/rmi://localhost:9914/jmxrmi" />
</bean>
```

3. 用 ServiceMix 4 MBeanServer 来配置 CXF

如果在 ServiceMix4 容器中嵌入一个 CXF 服务，配置与上面略有不同。开发人员不想启动一个新的 MBeanServer，同时又不希望创建附加连接器作为两者的容器。开发人员可以通过 OSGI 框架来引用容器的 MBeanServer 和注入这些引用到 JMX 集成延伸。不要忘记添加 Spring 的 OSGI 的命名空间和 schemaLocation 到 CXF 的配置文件。

```
<!-- OSGi namespace and schemaLocation required -->
<beans ...
      xmlns:osgi="http://www.springframework.org/schema/osgi"
      ...
      xsi:schemaLocation="...
```

```
          http://www.springframework.org/schema/osgi
http://www.springframework.org/schema/osgi/spring-osgi.xsd">

...

<!-- Grab a reference to the current MBeanServer -->
<osgi:reference id="mbeanServer" interface="javax.management.MBeanServer" cardinality="0..1"/>

<bean id="org.apache.cxf.management.InstrumentationManager"
  class="org.apache.cxf.management.jmx.InstrumentationManagerImpl">
     <property name="bus" ref="cxf" />
     <property name="enabled" value="true" />
      <!-- Unless you really want to open an additional connector, set this to false -->
      <property name="createMBServerConnectorFactory " value="false" />

      <!-- Inject the reference to the MBeanServer -->
      <property name="server" ref="mbeanServer" />
</bean>
```

4. 如何获得 Request/Response 的处理时间?

Apache CXF 管理模块还提供了一个功能，即可以让 Apache CXF 在 Bus 组件运行时获得服务的统计数字。

下面是配置代码，可添加到 Spring 上下文文件，这样就启动了时间响应的收集器：

```
<!-- Wiring the counter repository -->
    <bean id="CounterRepository" class="org.apache.cxf.management.counters.CounterRepository">
       <property name="bus" ref="cxf" />
    </bean>
```

该 CounterRepository 收集以下数据信息：调用应用程序错误、检查应用程序错误、未检查的应用程序错误、运行时错误、逻辑运行错误、总处理时间、最大处理时间和最小处理时间。

9.2 CXF 的日志管理

Apache CXF 框架的日志管理包括日志的设置、定义日志的级别并使用 Log4j 和 SLF4J 日志。

9.2.1 CXF 日志的设置

Apache CXF 框架在客户端和服务器端都用 Java SE 日志来记录 SOAP 请求和响应。通过定义使用输入/输出的拦截器可以激活日志记录，这些拦截器可拦截到客户端和服务器端的请求和响应服务。这些拦截器可以通过编程方式指定（通过 Java 代码或注释），也可以通过配置文件来定义。

建议采用配置文件方式来管理日志。配置方式与编程方式相比的优点如下：

（1）可以任意修改对日志记录的要求而无须重新编译代码。

（2）由于没有在代码中添加 Apache CXF 特定的 API，这有助于保持与其他基于 JAX-WS 规范的 Web Services 兼容互通。

对于应用程序的配置，服务器端点也可以直接添加日志记录拦截器实例化对象，其代码如下：

```
import javax.xml.ws.Endpoint;
import org.apache.cxf.interceptor.LoggingInInterceptor;
```

```
import org.apache.cxf.interceptor.LoggingOutInterceptor;
import org.apache.cxf.jaxws.EndpointImpl;

Object implementor = new GreeterImpl();
EndpointImpl ep = (EndpointImpl) Endpoint.publish("http://localhost/service",
implementor);

ep.getServer().getEndpoint().getInInterceptors().add(new LoggingInInterceptor());
ep.getServer().getEndpoint().getOutInterceptors().add(new LoggingOutInterceptor());
```

对于运行在 CXFServlet 的 Web Services，下面的注释可用在 SEI 接口或 SEI 实现类。如果这些代码放置在 SEI，这可以同时激活客户端和服务器的日志记录。如果这些代码放置在 SEI 的实现类，它只记录服务器端的日志。

```
import org.apache.cxf.feature.Features;

@javax.jws.WebService(portName = "MyWebServicePort", serviceName = "MyWebService", ...)
@Features(features = "org.apache.cxf.feature.LoggingFeature")
public class MyWebServicePortTypeImpl implements MyWebServicePortType {
```

实现相同功能的代码如下：

```
import org.apache.cxf.interceptor.InInterceptors;
import org.apache.cxf.interceptor.OutInterceptors;

@javax.jws.WebService(portName = "WebServicePort", serviceName = "WebServiceService", ...)
@InInterceptors(interceptors = "org.apache.cxf.interceptor.LoggingInInterceptor")
@OutInterceptors(interceptors = "org.apache.cxf.interceptor.LoggingOutInterceptor")
public class WebServicePortTypeImpl implements WebServicePortType {
```

对应的客户端应用程序的日志记录，下面的代码可作为一个例子：

```
import org.apache.cxf.endpoint.Client;
import org.apache.cxf.frontend.ClientProxy;
import org.apache.cxf.interceptor.LoggingInInterceptor;
import org.apache.cxf.interceptor.LoggingOutInterceptor;

public class WSClient {
   public static void main (String[] args) {
      MyService ws = new MyService();
      MyPortType port = ws.getPort();

      Client client = ClientProxy.getClient(port);
      client.getInInterceptors().add(new LoggingInInterceptor());
      client.getOutInterceptors().add(new LoggingOutInterceptor());

      // make WS calls...
```

9.2.2 定义日志级别

在 Apache CXF 软件包发布的文件中，有一个/etc 目录，该目录下有一个 Java SE logging.properties 文件的例子。开发人员可以用该文件配置日志记录。例如，开发人员希望控制台日志记录从 WARNING 级别变成 FINE 级别，只需在 logging.properties 属性文件修改这两个属性，配置如下：

```
.level= FINE
java.util.logging.ConsoleHandler.level = FINE
```

一旦这样做了，开发人员需要设置- Djava.util.logging.config.file 属性去定位 logging.properties 文件的位置。作为一个例子，下面的 Ant 目标具有此属性设置：

```
<target name="runClient">
  <java classname="client.WSClient" fork="true">
    <classpath>
      <pathelement location="${build.classes.dir}"/>
      <fileset dir="${env.CXF_HOME}/lib">
        <include name="*.jar"/>
      </fileset>
    </classpath>
    <jvmarg
value="-Djava.util.logging.config.file=/usr/myclientapp/logging.properties"/>
  </java>
</target>
```

9.2.3 使用 Log4J 日志方式

如上所述，Apache CXF 框架默认情况下使用 java.util.logging 包。但框架也可以把 Apache CXF 日志转化为 Log4J 日志。这通过配置文件来实现。有两种方式进行转化，下面就这种配置进行详细说明。

1. 添加下面的系统属性到 Apache CXF 初始化的类路径：

```
-Dorg.apache.cxf.Logger=org.apache.cxf.common.logging.Log4jLogger
```

2. 添加 META-INF/cxf/org.apache.cxf.Logger 文件到类路径，确保该文件包含以下内容：

```
org.apache.cxf.common.logging.Log4jLogger
```

9.2.4 使用 SLF4J 日志方式

如上所述，Apache CXF 默认情况下使用 java.util.logging 包。但开发者也可以把 Apache CXF 日志转化为 SLF4J 日志。这也是通过配置文件来实现。有两种方式进行转化，下面配置进行详细说明。

1. 添加下面系统属性到 Apache CXF 初始化的类路径：

```
-Dorg.apache.cxf.Logger=org.apache.cxf.common.logging.Slf4jLogger
```

2. 添加 META-INF/cxf/org.apache.cxf.Logger 文件到类路径，确保该文件包含以下内容：

```
org.apache.cxf.common.logging.Slf4jLogger
```

9.3 Apache CXF 的调试管理

简单介绍一下 Apache CXF 的调试工具。

9.3.1 Eclipse IDE

在 Eclipse 开发工具上可以调试 Web Services 的很多内容。这种方式主要是进行源代码的跟踪和调试。也可以通过其他工具来查看和获得网络流量、SOAP 请求和 SOAP 响应等内容。

9.3.2 Tcpmon

采用 Tcpmon 工具，开发人员可以轻松地查看在客户端和服务器端之间频繁交换的信息。

9.3.3 WSMonitor

WSMonitor 是一个类似 Tcpmon 的工具，但是 WSMonitor 比 Tcpmon 更轻量级而且提供更多的功能。

9.3.4 SOAP UI

SOAP UI 也可以用于调试。除了查看信息，SOAP UI 还可以发送消息和测试服务负载。SOAP UI 也有支持 Eclipse、IDEA 和 NetBeans 插件。

9.3.5 Wireshark

Wireshark，一个网络数据包分析仪，可用于跟踪 SOAP 消息路由。Wireshark 也可以帮助开发人员知道在服务器上造成 Apache CXF 客户端无法正常运行的 HTML 错误信息，让开发人员看到 non-SOAP 错误消息。

第 10 章　CXF 的工具

本章主要介绍 Apache CXF 的工具。首先是应用 Apache CXF 框架进行二次开发的工具，这里主要介绍 Ant，其后重点介绍 Apache CXF 框架的生成工具，即通过一些命令行实现基本功能。

本章内容：

1. 应用 Apache CXF 的开发工具。
2. Apache CXF 的生成工具。

10.1　Ant 工具（2.0.x 和 2.1.x）

Apache CXF 2.1 版本及其以前版本都提供了一个对 Ant 实现的 WSDL2Java 和 java2ws 命令。但 Apache CXF 的 2.2 版本将不会支持这些命令，因为这种使用方式太简单了。

在 Apache CXF 中使用 Ant 需要一个单独的 JAR 文件，该文件存于 modules/integration/cxf-anttasks-2.1.jar。

下面例子是使用 Ant 的 build 文件代码：

```
<project name="frontend_javafirst_jaxws" default="build" basedir=".">

   <import file="common_build.xml"/>

   <target name="client" description="run demo client" depends="build">
      <property name="param" value=""/>
      <cxfrun classname="demo.cxf.frontends.javafirst.client.Client"
         param1=""
         param2="${op}"
         param3="${param}"/>
   </target>

   <target name="server" description="run demo server" depends="build">
      <cxfrun classname="demo.cxf.frontends.javafirst.server.Server"/>
   </target>

   <property name="cxf.war.file.name" value="helloworld"/>
   <target name="war" depends="build">
      <cxfwar filename="${cxf.war.file.name}.war"/>
   </target>

   <target name="client-servlet" description="run demo client hitting servlet"
```

```
depends="build">
        <property name="param" value=""/>
        <cxfrun    classname="demo.cxf.frontends.javafirst.client.Client"    param1=
"${base.url}/helloworld/services/hello_world?wsdl" param2="${op}" param3="${param}"/>
    </target>
</project>
```

可以直接使用 Ant <java/>任务，这种方式可以通过命令行 WSDL2Java 和 java2ws（Apache CXF 的 2.1 版本）或 java2wsdl（Apache CXF 的 2.0.x 版本）来完成。

10.2　在 Eclipse 的 CXF 工具

JAX-WS 工具是基于 Eclipse 的 Webtools 框架工具，在该平台上可以开发 JAX-WS 服务。

10.3　Java 代码生成 Web Services

通过 Java 代码生成 Web Services 主要是通过工具把 Java 代码转换为 Web Services 的文件——WSDL 文件。java2ws 使用 Web 服务器端点实现（SEI）类及其相关联的类对象来生成一个 WSDL 文件、包装 bean 和用于启动 Web Services 的服务器端代码和客户端代码。采用的命令工具名称为 java2ws，需要在 Windows 的 DOS 窗口中执行命令。

本节介绍 java2ws 命令的调用方式、参数列表、参数说明以及一些例子与其说明。

1. java2ws 调用方式

命令 java2ws 的调用方式和参数列表如下。

```
java2ws -databinding <jaxb or aegis> -frontend <jaxws or simple> -wsdl -wrapperbean
-client -server -ant
         -o <output-file> -d <resource-directory> -classdir <compile-classes-directory>
-cp <class-path>
         -soap12 -t <target-namespace> -beans <ppathname of the bean definition file>*
         -servicename <service-name> -portname <port-name> -createxsdimports -h -v
-verbose -quiet {classname}
```

2. java2ws 参数说明

表 10-1 描述了 java2ws 命令各个参数的使用和功能。

表 10-1　java2ws命令参数选项和功能

参数选项	说明
-?	显示此工具的联机帮助，与-help和-h参数功能一样
-help	显示此工具的联机帮助，与-?和-h参数功能一样
-h	显示此工具的联机帮助，与-?和-help参数功能一样
-o	指定生成的WSDL文件名称
-databinding	指定的数据绑定类型（如Aegis或JAXB）。JAXWS前端应用模式的默认值是JAXB数据绑定，而简单前端应用模式默认值是Aegis数据绑定
-frontend	指定前端应用。java2ws工具支持JAXWS和简单前端应用
-wsdl	指定生成的WSDL文件

（续表）

参数选项	说明
-wrapperbean	指定要生成的包装Bean类源代码和异常Bean类源代码
-client	指定要生成客户端源代码
-server	指定要生成的服务器端源代码
-ant	指定要生成一个用于Ant 的build.xml脚本文件
-cp	指定SEI和type类的文件搜索路径和zip/jar文件
-soap12	指定生成的WSDL将包含一个SOAP 1.2绑定
-t	指定在生成的WSDL文件中使用目的命名空间（namespace）
-servicename	指定生成的服务元素name属性的数值
-v	显示该工具的版本号
-verbose	标识在代码生成过程中要求显示反馈信息
-quiet	标识在代码生成过程中要求不显示反馈信息
-s	生成的源文件（包装Bean、异常Bean、客户端或服务器端代码）的目录位置
-classdir	定义生成源文件的编译目录。如果没有指定，源文件将不会被编译
-portname	指定生成的WSDL文件中使用的端口名称
-createxsdimports	以独立文件的输出模式并用imports来导入而不是使用内联来加载文件到WSDL
-d	定义资源文件的放置目录。在默认情况下，WSDL文件将放置在这个目录
classname	指定SEI类的名称

3. 例子及其说明

下面用五个例子来说明 java2ws 的使用。

java2ws 调用的例子 1 代码如下：

```
java2ws -wsdl -d ./resource org.apache.hello_world_soap_http.Greeter
```

使用 org.apache.hello_world_soap_http.Greeter 来生成 WSDL 文件，并把生成的文件放置到当前的 resource 文件目录下。

java2ws 调用的例子 2 代码如下：

```
java2ws -cp ./tmp org.apache.hello_world_soap_http.Greeter -wsdl
```

使用 org.apache.hello_world_soap_http.Greeter 来生成 WSDL 文件。在/tmp 目录下搜索 SEI 类、类型类和 zip/jar 文件。

java2ws 调用的例子 3 代码如下：

```
java2ws -o hello.wsdl -wsdl org.apache.hello_world_soap_http.Greeter
```

使用 org.apache.hello_world_soap_http.Greeter 来生成 WSDL 文件。生成的文件名字为 hello.wsdl。

java2ws 调用的例子 4 代码如下：

```
java2ws -client -server -s ./src org.apache.hello_world_soap_http.Greeter
```

使用 org.apache.hello_world_soap_http.Greeter 来生成客户端源代码和服务器端源代码。生成

的文件放置到当前的/src 目录。

java2ws 调用的例子 5 代码如下：

```
java2ws -wrapperbean -classdir ./classes org.apache.hello_world_soap_http.Greeter
```

使用 org.apache.hello_world_soap_http.Greeter 来生成包装 Bean 类源代码和异常 Bean 类源代码。对生成的源代码文件进行编译并把编译文件放置到当前的/classes 目录。

10.4　Java 代码生成 WSDL

java2wsdl 使用编译的 Web 服务器端点实现类（SEI）和相关联的类生成一个 WSDL 文件。Apache CXF 框架的后期版本已经不再支持这种调用了。

本节介绍 java2wsdl 命令的调用方式、参数列表、参数说明以及一些例子与其说明。

1. java2wsdl 调用方式

命令 java2wsdl 的调用方式和参数列表如下。

```
java2wsdl [-?|-help|-h][-o <output-file>][-cp <class-path>][-soap12][-t <target-namespace>]
[-servicenam   <seservice-name>]   [-v][-verbose|-quiet][-s   <source-directory>]
[-classdir <compile-classes-directory>][-portname <port-name>][-createxsdimports]
[-d <output-directory>] { classname }
```

2. java2wsdl 参数说明

表 10-2 描述 java2wsdl 命令各个参数的使用和功能。

表 10-2　java2wsdl命令参数选项和功能

参数选项	说明
-?	显示此工具的联机帮助
-help	显示此工具的联机帮助
-h	显示此工具的联机帮助
-o	指定生成的WSDL文件名称
-cp	指定的SEI和类型类的文件搜索路径和zip/jar文件
-soap12	指定生成的WSDL将包含一个SOAP 1.2绑定
-t	指定生成的WSDL文件使用的目标命名空间（namespace）
-servicename	指定生成服务元素的name属性的数值
-v	显示该工具的版本号
-verbose	标识在代码生成过程中要求显示反馈信息
-quiet	标识在代码生成过程中要求不显示反馈信息
-s	生成源文件的放置目录位置，主要是JAXWS模式，命令将生成包装Bean和异常Bean
-classdir	定义生成源文件的编译目录。如果没有指定，源文件将不会被编译
-portname	指定在生成的WSDL文件中使用的端口名称
-createxsdimports	以独立的文件的输出模式并用imports来导入而不是使用内联来加载它们到WSDL
-d	输出文件的放置目录
classname	指定的SEI类的名称。注意这是编译后的类，而不是源文件

3. java2wsdl 例子及其说明

下面用四个例子来说明 java2wsdl 的使用。

java2wsdl 调用的例子 1 代码如下：

```
java2wsdl org.apache.hello_world_soap_http.Greeter
```

使用 org.apache.hello_world_soap_http.Greeter 来生成 WSDL 文件。

java2wsdl 调用的例子 2 代码如下：

```
java2wsdl -cp ./tmp org.apache.hello_world_soap_http.Greeter
```

使用 org.apache.hello_world_soap_http.Greeter 来生成 WSDL 文件，在/tmp 目录下搜索 SEI 类、类型类和 zip/jar 文件。

java2wsdl 调用的例子 3 代码如下：

```
java2wsdl -o hello.wsdl org.apache.hello_world_soap_http.Greeter
```

使用 org.apache.hello_world_soap_http.Greeter 来生成 WSDL 文件，生成的文件名字为 hello.wsdl。

java2wsdl 调用的例子 4 代码如下：

```
java2wsdl -o hello.wsdl -t http://cxf.apache.org org.apache.hello_world_soap_
http.Greeter
```

使用 org.apache.hello_world_soap_http.Greeter 来生成 WSDL 文件，生成的文件名字为 hello.wsdl。hello.wsdl 的命名空间为“http://cxf.apache.org”。

10.5 WSDL 生成 Java 代码

wsdl2java 命令的使用需要一个 WSDL 文件并基于这个文件生成一个有完整注释的 Java 源码程序，该程序能提供一项 Web Services 服务。该 WSDL 文档必须有一个有效的 portType 元素，但 WSDL 文档并不需要包含一个绑定元素或服务元素。使用可选项参数，可以定制化生成的代码。此外，wsdl2java 命令可以生成一个基于 Ant 工具的生成文件，以便生成应用程序。本命令需要在 Windows 的 DOS 窗口中实现。

本节介绍 wsdl2java 命令的调用方式、参数列表、参数说明以及一些例子与其说明。

1. wsdl2java 调用方式

命令 wsdl2java 的调用方式和参数列表如下。

```
wsdl2java -fe <frontend name>* -db <data binding name>* -wv <[wsdl version]>*
-p <[wsdl namespace =]Package Name>* -sn <service-name> -b <binding-name>*
-catalog <catalog-file-name> -d <output-directory> -compile
-classdir <compile-classes-directory> -impl -server -client -all
-autoNameResolution -defaultValues<=class name for DefaultValueProvider> -ant
-nexclude <schema namespace [= java packagename]>* -exsh <(true, false)>
-dns <(true, false)> -dex <(true, false)> -validate -keep
-wsdlLocation <wsdllocation attribute> -xjc<xjc arguments>
 -noAddressBinding -h -v -verbose -quiet <wsdlurl>
```

2. wsdl2java 参数说明

表 10-3 描述 wsdl2java 命令各个参数的使用和功能。

表 10-3　wsdl2java命令参数选项和功能

参数选项	说明
-?	显示此工具的联机帮助
-help	显示此工具的联机帮助
-h	显示此工具的联机帮助
-fe	指定的前端名称。默认是JAXWS。目前也只支持JAXWS前端
-db	指定数据绑定的名称。默认是JAXB。目前，支持JAXB数据绑定和XMLBeans数据绑定。CXF 2.3版本将支持SDO（静态的和动态SDO）数据绑定
-wv wsdl-version	指定的WSDL版本。默认是WSDL 1.1。目前仅支持WSDL 1.1版本
-p [wsdl-namespace=] PackageName	指定0或更多，生成代码使用的包名。可以选择指定的WSDL命名空间与包名称映射
-sn service-name	WSDL服务名称使用的生成代码
-b binding-name	指定JAXWS绑定文件、JAXB绑定文件或XMLBeans上下文文件。使用多个-b标志来指定绑定多种文件
-catalog catalog-file-name	指定目录文件来映射导入的wsdl/schema
-d output-directory	指定生成的代码文件被写入的目录
-compile	编译生成的Java文件
-classdir	定义生成源文件的编译目录。如果没有指定，源文件将不会被编译
-client	生成客户端主线的源代码
-server	生成服务器端主线的源代码
-impl	生成实现对象的源代码
-all	生成所有开始点代码：类型、服务代理、服务接口、服务器主线、客户主线、实现对象，一个基于Ant工具的build.xml文件
-ant	生成的一个Ant工具的build.xml文件
-autoNameResolution	不需要使用定制化约束而自动解决命名冲突
-defaultValues=[DefaultValueProvider impl]	指定implement执行和客户端生成的默认值。需要定制化一个默认的驱动，默认提供驱动是RandomValueProvider
-nexclude	当生成代码时，schema-namespace [=java-packagename]忽略指定的WSDL架构命名空间。这个选项可以使用多次。此外，可以选择被排斥在命名空间（被描述类型的使用）的Java包名
-exsh (true/false)	启用或禁用隐性SOAP头（即SOAP头在wsdl:binding中定义，而不是在wsdl:portType section定义），默认值是false
-dns (true/false)	启用或禁用加载默认的命名空间名称映射包。默认值是true。同时http://www.w3.org/2005/08/addressing=org.apache.cxf.ws.addressing命名空间包映射将被启用
-dex (true/false)	启用或禁用默认加载不包括命名空间的映射。默认值是true
-validate	启用在生成WSDL代码前进行验证操作
-keep	指定的代码生成器不会覆盖已经存在的文件。这需要开发人员负责解决由此产生的编制问题
-wsdlLocation	指定@WebServiceClient注释的wsdlLocation属性值

（续表）

参数选项	说明
-xjc<xjc args>	指定一个逗号分隔的列表。当使用JAXB的绑定参数时，直接传递给XJC处理器。使用 - xjc – X可以保持一个可用的XJC插件列表
-noAddressBinding	与CXF 2.0兼容，此标志要求代码生成器生成WS-Addressing类型的老版本CXF属性，而不是生成兼容WS-Addressing类型的JAX-WS 2.1属性
-v	显示该工具的版本号
-verbose	标识在代码生成过程中要求显示反馈信息
-quiet	标识在代码生成过程中要求不显示反馈信息
wsdlfile	生成代码使用WSDL文件的路径和名字

3. wsdl2java 工具例子及其说明

下面用三个例子来说明 wsdl2java 的使用。

wsdl2java 调用的例子 1 代码如下：

```
wsdl2java HelloWorld.wsdl
```

使用名字为 HelloWorld.wsdl 的 WSDL 文件生成 Java 类。

wsdl2java 调用的例子 2 代码如下：

```
wsdl2java -p com.iona.greeting Greeting.wsdl
```

使用名字为 HelloWorld.wsdl 的 WSDL 文件生成名字为 com.iona.greeting 的 Java 类。

wsdl2java 调用的例子 3 代码如下：

```
wsdl2java -client HelloWorld.wsdl
```

使用名字为 HelloWorld.wsdl 的 WSDL 文件生成客户端主线的开始点代码。

10.6 WSDL 转化为 Javascript

wsdl2js 命令通过一个 WSDL 文档来生成一个提供 Web Services 的 JavaScript 代码。该 WSDL 文档必须有一个有效的 portType 元素，但该 WSDL 文档并不需要包含一个绑定元素或服务元素。采用的命令工具名称为 wsdl2js，需要在 Windows 的 DOS 窗口中实现。

本节介绍 wsdl2js 命令的调用方式、参数列表、参数说明以及一些例子与其说明。

1. wsdl2js 调用方式

命令 wsdl2js 的调用方式和参数列表如下。

```
wsdl2js  -? -help -h  -wv <wsdlVersion> -p <wsdlNamespace=jsPrefix> -catalog
<catalogUrl> -d <outDir> -validate -v -verbose -quiet {wsdlUrl}
```

2. wsdl2js 参数说明

表 10-4 描述 wsdl2js 命令各个参数的使用和功能。

表 10-4　wsdl2js命令参数选项和功能

参数选项	说明
-?	显示此工具的联机帮助
-help	显示此工具的联机帮助
-h	显示此工具的联机帮助
-wv	指定该命令希望的WSDL版本。默认是WSDL 1.1。该命令还支持WSDL 1.2
-p	指定WSDL文件的命名空间和生成的JavaScript使用的前缀之间的映射。这个选项可以多次使用
-catalog	指定一个XML目录的URL，该URL用于解决引入schema和WSDL文档
-d	指定生成的代码文件被写入的目录
-validate	启用在生成WSDL代码前进行验证操作
-v	显示该命令的版本号
-verbose	标识在代码生成过程中要求显示反馈信息
-quiet	标识在代码生成过程中要求不显示反馈信息
wsdlUrl	指定生成代码的WSDL文件的位置

3. wsdl2js 例子及其说明

下面用三个例子来说明 wsdl2js 的使用。

wsdl2js 调用的例子 1 代码如下：

```
wsdl2js hello_world.js
```

使用名字为 HelloWorld.wsdl 的 WSDL 文件生成 Java 类。

wsdl2js 调用的例子 2 代码如下：

```
wsdl2js -wv 1.2 hello_world12.wsdl
```

使用名字为 HelloWorld.wsdl 的 WSDL 文件生成 JavaScript 代码，同时要求支持 WSDL 1.2。

wsdl2js 调用的例子 3 代码如下：

```
wsdl2js -d javascript hello_world.wsdl
```

使用名字为 HelloWorld.wsdl 的 WSDL 文件生成 JavaScript 代码，同时把生成的文件放置到当前的 JavaScript 文件目录下。

10.7　WSDL 生成服务（Service）

wsdl2service 命令创建一个新的 WSDL 文件，该文件包含从现有的 WSDL 文档中绑定 HTTP 服务或 JMS 服务的定义。采用的命令工具名称为 wsdl2service，需要在 Windows 的 DOS 窗口中实现。

本节介绍 wsdl2service 命令的调用方式、参数列表、参数说明以及一些例子与其说明。

1. wsdl2service 调用方式

命令 wsdl2service 的调用方式和参数列表如下。

```
wsdl2service -transport <soap/xml/jms> -e <service-name>
            -p <port-name> -n <attribute-binding-name>
```

```
            [-a <address>] -jds <jms-destionation-Style(queue/topic)>
            -jpu <jms-jndi-provider-URL> -jcf <jms-initial-context-factory>
            -jfn <jms-jndi-connection-factory-name>
            -jdn <jms-jndi-destination-name>  -jmt <jms-message-type(text/binary)>
            -jmc <use-messageid-as-correlationid(true/false)>
            -jsn <durable-subscriber-name> -o <output-file>
-d <output-directory> -h -v -verbose -quiet <wsdlurl>
```

2. wsdl2service 参数说明

表 10-5 描述 wsdl2service 命令各个参数的使用和功能。

表 10-5 wsdl2service命令参数选项和功能

参数选项	说明
-?	显示此工具的联机帮助
-help	显示此工具的联机帮助
-h	显示此工具的联机帮助
-transport <soap/xml/jms>	指定生成Web Services所使用的传输类型。有SOAP、XML、JMS三种类型
-e <service-name>	指定生成Web Services元素name属性值
-p <port-name>	指定生成Web Services端口name属性值。可以指定多个端口元素，但是要对应多个单独的名称空间
-n <attribute-binding-name>	指定生成Web Services所使用的绑定类型
-a <address>	指定端口地址元素地址使用的值
-jds <jms-destionation-Style(queue/topic)>	JMS目标类型（队列/主题）
-jpu <jms-jndi-provider-URL>	JMS上JNDI提供的网址
-jcf <jms-initial-context-factory>	JMS初始上下文工厂
-jfn <jms-jndi-connection-factory-name>	JMS中JNDI连接工厂的名称
-jdn <jms-jndi-destination-name>	JMS中JNDI目的地名称
-jmt <jms-message-type(text/binary)>	JMS消息类型（text/binary）
-jmc <use-messageid-as-correlationid(true/false)>	使用MessageID作为CorrelationID（true/false）
-jsn <durable-subscriber-name>	持久化注册名称
-o <output-file>	指定生成WSDL文件的名称
-d <output-directory>	指定生成WSDL文件的放置目录
-v	显示该工具的版本号
-verbose	标识在代码生成过程中要求显示反馈信息
-quiet	标识在代码生成过程中要求不显示反馈信息
wsdlurl	生成代码WSDL文件的路径和文件名

3. wsdl2service 例子及其说明

下面用一个例子来说明 wsdl2service 的使用。

```
wsdl2service -transport soap -e servicename -p portname -n bindingname HelloWorld.wsdl
```

使用名字为 HelloWorld.wsdl 的 WSDL 文件生成服务。该服务基于 SOAP 协议，采用 servicename 名称，采用 portname 作为端口，采用 bindingname 做绑定。

10.8　WSDL 生成 SOAP

wsdl2soap 命令将产生一个实现 SOAP 绑定的新 WSDL 文件。该文件是基于一个有 portType 元素的 WSDL 文件。采用的命令工具名称为 wsdl2soap，需要在 Windows 的 DOS 窗口中实现。

本节介绍 wsdl2soap 命令的调用方式、参数列表、参数说明以及一些例子与其说明。

1. wsdl2soap 调用方式

命令 wsdl2soap 的调用方式和参数列表如下。

```
wsdl2soap [[-?] | [-help] | [-h]] [-i <port-type-name>] [-b <binding-name>]
[-soap12] [-d <output-directory>] [-o <output-file>] [-n <soap-body-namespace>]
[-style (document/rpc)] [-use (literal/encoded)] [-v] [[-verbose] | [-quiet]] wsdlurl
```

2. wsdl2soap 参数说明

表 10-6 描述 wsdl2soap 命令各个参数的使用和功能。

表 10-6　wsdl2soap命令参数选项和功能

参数选项	说明
-?	显示此工具的联机帮助
-help	显示此工具的联机帮助
-h	显示此工具的联机帮助
-i	指定生成绑定的portType元素
-b	指定生成SOAP绑定的名称
-soap12	指定生成的绑定将使用SOAP 1.2
-d	指定生成WSDL文件放置的目录
-o	指定生成WSDL文件的名称
-n	当类型是RPC时，指定SOAP body命名空间
-style (document/rpc)	指定SOAP绑定的编码方式（文档模式或RPC）。默认为文档模式
-use (literal/encoded)	指定SOAP绑定类型（encoded或literal）。默认为literal
-v	显示该工具的版本号
-verbose	标识在代码生成过程中要求显示反馈信息
-quiet	标识在代码生成过程中要求不显示反馈信息
wsdlurl	生成代码的WSDL文件的路径和文件名

3. wsdl2soap 例子及其说明

下面用一个例子来说明 wsdl2soap 的使用。

```
wsdl2soap -i GreetPortType -n http://apache.org/hello_world_doc_lit -o Greeting.wsdl
-style rpc -use literal TestGreeting.wsdl
```

使用名字为 TestGreeting.wsdl 的 WSDL 文件生成一个新的基于 SOAP 的 WSDL 文件。其绑定的 portType 元素为 GreetPortType。SOAP body 命名空间是 http://apache.org/hello_world_doc_lit。生成 WSDL 文件的名称是 Greeting.wsdl。SOAP 绑定的编码方式是 RPC。SOAP 绑定类型是 literal。

10.9 WSDL 生成 XML

wsdl2xml 命令将产生一个实现 XML 绑定的新 WSDL 文件。该文件基于一个有 portType 元素的 WSDL 文件。采用的命令工具名称为 wsdl2xml，需要在 Windows 的 DOS 窗口中实现。

本节介绍 wsdl2xml 命令的调用方式、参数列表、参数说明以及一些例子与其说明。

1. wsdl2xml 调用方式

命令 wsdl2xml 的调用方式和参数列表如下。

```
wsdl2xml [[-?] | [-help] | [-h]] [-i <port-type-name>] [-b <binding-name>] [-d
<output-directory>] [-o <output-file>] [-a <address>] [-e <service-name>] [-p
<port-name>] [-v] [[-verbose] | [-quiet]] wsdlurl
```

2. wsdl2xml 参数说明

表 10-7 描述 wsdl2xml 命令各个参数的使用和功能。

表 10-7 wsdl2xml命令参数选项和功能

参数选项	说明
-?	显示此工具的联机帮助
-help	显示此工具的联机帮助
-h	显示此工具的联机帮助
-i	指定将生成的绑定的portType元素
-b	指定生成XML绑定的名称
-d	指定生成WSDL文件的放置目录
-o	指定生成WSDL文件的名称
-p	指定生成端口元素name属性的值，用空格分开
-s	指定生成服务的名称
-a	指定端口地址元素的值，用空格分开
-v	显示该工具的版本号
-verbose	标识在代码生成过程中要求显示反馈信息
-quiet	标识在代码生成过程中要求不显示反馈信息
wsdlurl	包含了portType元素定义的WSDL文件的路径和文件名

3. wsdl2xml 例子及其说明

下面用一个例子来说明 wsdl2xml 的使用。

```
wsdl2xml -i GreetPortType -a address TestGreeting.wsdl
```

使用名字为 TestGreeting.wsdl 的 WSDL 文件生成一个新的基于 XML 的 WSDL 文件。其绑定的 portType 元素为 GreetPortType。端口地址元素的值为 address。

10.10　WSDL 验证器

该 wsdlvalidator 命令验证一个 WSDL 文件或 WSDL URL 格式是否正确并与 schema 模式是否一致。采用的命令工具名称为 wsdlvalidator，需要在 Windows 的 DOS 窗口中实现。

本节介绍 wsdlvalidator 命令的调用方式、参数列表、参数说明以及一些例子与其说明。

1. wsdlvalidator 调用方式

命令 wsdlvalidator 的调用方式和参数列表如下。

```
wsdlvalidator -d <schema-directory> -s <schema-url>* -w -h -v -verbose -quiet <wsdlurl>
```

2. wsdlvalidator 参数说明

表 10-8 描述 wsdlvalidator 命令各个参数的使用和功能。

表 10-8　wsdlvalidator命令参数选项和功能

参数选项	说明
-?	显示此工具的联机帮助
-help	显示此工具的联机帮助
-h	显示此工具的联机帮助
-d <schema-directory>	指定搜索schema的目录
-s <schema-url>	指定一个用户特定schema模式的URL，该URL包含验证。此选项可以多次出现
-v	显示该工具的版本号
-verbose	标识在代码生成过程中要求显示反馈信息
-quiet	标识在代码生成过程中要求不显示反馈信息
-w	禁止警告，如果希望验证WSDL（无服务/绑定）的逻辑性，必须选择这个选项
wsdlurl	生成代码WSDL文件的路径和文件名

3. wsdlvalidator 例子及其说明

下面用两个例子来说明 wsdlvalidator 的使用。

wsdlvalidator 调用的例子 1 代码如下：

```
wsdlvalidator HelloWorld.wsdl
```

验证 HelloWorld.wsdl 格式是否正确并与 schema 模式是否一致。

wsdlvalidator 调用的例子 2 代码如下：

```
wsdlvalidator -s http://cxf.apache.org/schemas -s http://mycompany/schemas Greeting.wsdl
```

验证 Greeting.wsdl 格式是否正确并与 schema 模式是否一致。指定一个用户特定 schema 模式的网址是 http://cxf.apache.org/schemas 和 http://mycompany/schemas。

10.11　XSD 生成 WSDL

xsd2wsdl 命令用 XSD 模式文件生成 WSDL 文件。采用的命令工具名称为 xsd2wsdl，需要在

Windows 的 DOS 窗口中实现。

本节介绍 xsd2wsdl 命令的调用方式、参数列表、参数说明以及一些例子与其说明。

1. xsd2wsdl 调用方式

命令 xsd2wsdl 的调用方式和参数列表如下。

```
xsd2wsdl [-?|-help|-h][-o <output-file>][-n <wsdl-name>][-t <target-namespace>][-v]
[-verbose|-quiet][-d <output-directory>] { xsdurl }
```

2. xsd2wsdl 参数说明

表 10-9 描述 xsd2wsdl 命令各个参数的使用和功能。

表 10-9 xsd2wsdl命令参数选项和功能

参数选项	说明
-?	显示此工具的联机帮助
-help	显示此工具的联机帮助
-h	显示此工具的联机帮助
-o	指定生成WSDL文件的名称
-n	生成WSDL的名称
-t	指定生成WSDL文件使用的目标命名空间
-v	显示该工具的版本号
-verbose	标识在代码生成过程中要求显示反馈信息
-quiet	标识在代码生成过程中要求不显示反馈信息
-d	输出文件的放置目录
xsdurl	指定XSD网址

3. xsd2wsdl 例子及其说明

下面用一个例子来说明 xsd2wsdl 的使用。

```
xsd2wsdl -t http://org.apache/invoice -n Invoice Invoice.xsd
```

使用名字为 Invoice.xsd 的 XSD 文件生成一个 WSDL 文件。新生成 WSDL 文件的目标命名空间是 http://org.apache/invoice，WSDL 名字是 Invoice。

第 11 章 CXF 实现 RESTful 服务

REST（REpresentation State Transfer），即代表性状态传输，描述了一种应用程序的架构样式。这个单词首次出现在 2000 年 Roy Fielding 的博士论文中，他是 HTTP 规范的主要编写者之一。REST 表示为一组架构约束条件和原则。满足这些约束条件和原则的应用程序或设计就是 RESTful。RESTful 提供了一种对传统模式（基于 SOAP 的 Web Services 的解决方案）的替代方案。

本章内容：

1. 介绍 RESTful 服务。
2. Apache CXF 实现 RESTful 方式。
3. Apache CXF 实现 RESTful 服务的例子说明。

11.1 RESTful 服务介绍

代表性状态传输（Representational State Transfer，REST）在 Web 领域已经得到了广泛的接受。REST 是基于 SOAP 和 Web Services 描述语言（Web Services Description Language，WSDL）的更为简单的替代品。这一转变的关键证据是主流 Web 2.0 服务提供者（包括 Yahoo、Google 和 Facebook）对 REST 的采纳，这些提供者弃用或放弃了基于 SOAP 和 WSDL 的接口，而采用了更易于使用、面向资源的模型来公开其服务。

11.1.1 RESTful 服务概述

REST 这个术语是由 Roy Fielding 在他的博士论文《Architectural Styles and the Design of Network-based Software Architectures》中提出的。REST 并非标准，而是一种开发 Web 应用的架构风格，可以将其理解为一种设计模式。REST 基于 HTTP、URI 以及 XML 这些现有的广泛流行的协议和标准。伴随着 REST 的发展，HTTP 协议得到了更加正确的使用。在 Roy Fielding 的论文中，他将 REST 作为目前 Web 体系结构的一种基础概念进行了详细介绍。他为 REST 提出了下列标准：

- 为现代 Web 体系结构进行建模的一组约束。
- REST 原则已应用于 HTTP 和 URI 规范。
- 在 HTTP 的发展过程中是可见的。

REST 不是一种协议，而是一种体系结构风格，这是非常重要的区别。

相比较于基于 SOAP 和 WSDL 的 Web Services，REST 模式提供了更为简洁的实现方案。目前，越来越多的 Web Services 开始采用 REST 风格设计和实现，真实世界中比较著名的 REST 服务包括：Google Ajax 搜索 API、Amazon Simple Storage Service（Amazon S3）等。

基于 REST 的 Web Services 遵循以下一些基本的设计原则。

- 系统中的每一个对象或资源都可以通过一个唯一的 URI 来进行寻址，URI 的结构应该简单、可预测且易于理解，比如定义目录结构式的 URI。
- 以遵循 RFC-2616 所定义协议的方式显式地使用 HTTP 方法，建立创建、检索、更新和删除（CRUD：Create, Retrieve, Update and Delete）操作与 HTTP 方法之间的一对一映射：
  - 若要在服务器上创建资源，应该使用 POST 方法；
  - 若要检索某个资源，应该使用 GET 方法；
  - 若要更改资源状态或对其进行更新，应该使用 PUT 方法；
  - 若要删除某个资源，应该使用 DELETE 方法。
- URI 所访问的每个资源都可以使用不同的形式加以表示（比如 XML 或者 JSON），具体的表现形式取决于访问资源的客户端，客户端与服务提供者使用一种内容协商的机制（请求头与 MIME 类型）来选择合适的数据格式，最小化彼此之间的数据耦合。

11.1.2 RESTful 原则

在 Web 应用程序中最重要的 REST 原则是，客户端和服务器之间的交互在请求之间是无状态的。从客户端到服务器的每个请求都必须包含理解请求所必需的全部信息。如果服务器在请求之间的任何时间点重启，客户端就不会得到通知。此外，无状态请求可以由任何可用服务器回答，这十分适合云计算之类的环境。客户端可以缓存数据以改进性能。

在服务器端，应用程序状态和功能可以分为各种资源。资源是一个向客户端公开的概念实体。资源包含多种概念，如应用程序对象、数据库记录、算法，等等。每个资源都使用 URI（Universal Resource Identifier）得到一个且唯一的地址。所有资源都共享统一的界面，以便在客户端和服务器之间传输状态。访问资源使用标准的 HTTP 方法，比如 GET、PUT、POST 和 DELETE。Hypermedia 是应用程序状态的引擎，资源表示通过超链接互联。

REST 另一个重要的原则是分层系统，这表示组件无法了解它与之交互的中间层以外的组件。通过将系统知识限制在单个层，可以限制整个系统的复杂性，促进了底层的独立性。

当 REST 架构的约束条件作为一个整体应用时，将生成一个可以扩展到大量客户端的应用程序。它还降低了客户端和服务器之间的交互延迟。统一界面简化了整个系统架构，改进了子系统之间交互的可见性。REST 简化了客户端和服务器的实现。

由于轻量级以及通过 HTTP 直接传输数据的特性，Web Services 的 RESTful 方法已经成为现阶段最常见的替代方法。RESTful 方法可以使用各种语言（比如 Java 程序、Perl、Ruby、Python、PHP 和 JavaScript，包括 Ajax）实现客户端。基于 RESTful 的 Web Services 通常可以通过自动客户端或代表用户的应用程序来进行访问。但是，这种简便性的服务让用户能够直接交互，使用他们的 Web 浏览器构建一个 GET URL 并读取返回的内容。

在 REST 样式的 Web Services 中，每个资源都有一个地址。资源本身就是方法调用的目标，

方法列表对所有资源都是一样的。这些方法都是标准方法，包括 HTTP GET、POST、PUT、DELETE，还可能包括 HEADER 和 OPTIONS。

在 RPC 样式的架构中，其关注点在于方法，而在 REST 样式的架构中，关注点在于资源——将使用标准方法检索并操作信息（使用表示的形式）。资源表现形式在表示形式中使用超链接进行互联。

Leonard Richardson 和 Sam Ruby 在他们的著作《RESTful Web Services》中引入了术语 REST-RPC 混合架构。REST-RPC 混合 Web Services 不使用信封包装方法、参数和数据，而是直接通过 HTTP 传输数据，这与 REST 样式的 Web Services 是类似的。但是它不使用标准的 HTTP 方法操作资源。它在 HTTP 请求的 URI 部分存储方法信息。好几个知名的 Web Services，比如 Yahoo 的 Flickr API 和 del.icio.us API 都使用这种混合架构。

REST 的具体实现应该遵循四个基本设计原则：

- 显式地使用 HTTP 方法。
- 无状态。
- 公开目录结构式的 URI。
- 传输 XML、JavaScript Object Notation（JSON），或同时传输这两者。

下面将详述这四个原则并解释其技术原理。

1. 显式地使用 HTTP 方法

基于 REST 的 Web Services 的主要特征之一是遵循 RFC 2616 定义的并且能显式地使用 HTTP 方法的协议方式。例如，HTTP GET 被定义为数据产生方法，意思是客户端应用程序可用于检索资源并从 Web Services 获取数据，或者执行某个查询并预期查找某一组匹配资源然后对该资源进行响应。

REST 要求开发人员显式地使用 HTTP 方法，并且其使用方式与协议定义一致。这个基本 REST 设计原则支持创建、读取、更新和删除（create, read, update and delete，CRUD）操作与 HTTP 方法之间的一对一映射。根据此映射：

- 若要在服务器上创建资源，应该使用 POST 方法。
- 若要检索某个资源，应该使用 GET 方法。
- 若要更改资源状态或对其进行更新，应该使用 PUT 方法。
- 若要删除某个资源，应该使用 DELETE 方法。

许多 Web API 的使用中，一个设计缺陷在于将 HTTP 方法用于非预期的用途。例如，HTTP GET 请求中的请求 URI 通常标识一个特定的资源。请求 URI 中的查询字符串包括一组参数，这些参数定义服务器用于查找一组匹配资源的搜索。至少在 HTTP/1.1 RFC 是这样描述 GET 方法的。但是在许多情况下，不优雅的 Web API 使用 HTTP GET 来触发服务器上的事务性操作——例如，向数据库添加记录。在这些情况下，GET 请求 URI 属于不正确使用，或者至少不是以基于 REST 的方式使用。如果 Web API 使用 GET 调用远程过程，则应该类似如下：

```
GET /adduser?name=Robert HTTP/1.1
```

这是非常不优雅的设计，因为上面的 Web 方法支持通过 HTTP GET 进行状态更改操作。换句话说，该 HTTP GET 请求具有副作用。如果处理成功，则该请求的结果是向基础数据存储区添

加一个新用户——在此例中为 Robert。这里的问题主要在语义上。Web Services 旨在通过检索与请求 URI 中的路径（或查询条件）匹配的资源，并在响应中返回这些资源或其表示形式，从而响应 HTTP GET 请求，而不是向数据库添加记录。从该协议方法预期用途的角度看，然后再从与 HTTP/1.1 兼容的 Web Services 的角度看，这种方式使用 GET 是不一致的。

除了语义之外，GET 还有其他问题，为了触发数据库中的记录的删除、修改或添加，或者以某种方式更改服务器端状态，GET 请求 Web 缓存工具（爬网程序）和搜索引擎简单地通过对某个链接进行爬网处理，从而意外地对服务器端进行了更改。克服此常见问题的简单方法是将请求 URI 上的参数名称和值转移到 XML 标记中。这样产生的标记是要创建的实体 XML 表示形式，可以在 HTTP POST 的正文中进行发送，此 HTTP POST 的请求 URI 是该实体的预期父实体。

2. 无状态

REST Web Services 需要满足日益提高的性能要求。REST 具有负载平衡和故障转移功能、代理和网关的服务器集群，这些通常以形成服务拓扑的方式进行组织，从而允许根据需要将请求从一个服务器路由到另一个服务器，以减少 Web Services 调用的总体响应时间。要使用中间服务器扩大规模，REST Web Services 需要发送完整、独立的请求。也就是说，发送的请求包括所有需要满足的数据，以便中间服务器中的组件能够进行转发、路由和负载平衡，而不需要在请求之间在本地保存任何状态。

完整、独立的请求不要求服务器在处理请求时检索任何类型的应用程序上下文或状态。REST Web Services 应用程序（或客户端）在 HTTP Header 和请求正文中包括服务器端组件生成响应所需要的所有参数、上下文和数据。这种意义上的无状态可以改进 Web Services 性能，并简化服务器端组件的设计和实现，因为服务器上没有状态，从而消除了与外部应用程序同步会话数据的需要。

图 11-1 演示了一个有状态的服务，某个应用程序可能向其请求多页结果集中的下一个页面，并假设该服务跟踪应用程序在结果集中导航的离开位置。在这个有状态的设计中，该服务递增并在某个位置存储 previousPage 变量，以便能够响应下一个页面的请求。

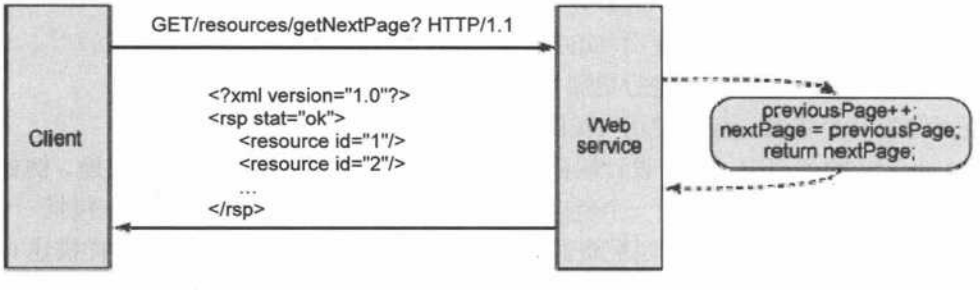


图 11-1 有状态的设计

类似如此的有状态的服务变得复杂化了。在 Java Platform, Enterprise Edition（Java EE）环境中，有状态的服务需要大量的预先考虑，以高效地存储会话数据和支持整个 Java EE 容器集群中的会话数据同步。在此类环境中，存在一个 Servlet/JavaServer Pages（JSP）和 Enterprise JavaBeans（EJB）开发人员非常熟悉的问题，他们经常在会话复制过程中艰难地查找引发 java.io.NotSerializableException 的根源。无论该异常是由 Servlet 容器在 HttpSession 复制过程

中引发的，还是由 EJB 容器在有状态的 EJB 复制过程中引发的。这都是个问题，会耗费开发人员的时间，尝试在构成服务器状态并且有时非常复杂的对象图表中查明没有实现 Serializable 的对象。此外，会话同步增加了开销，从而影响服务器性能。

另一方面，无状态的服务器端组件不那么复杂，很容易跨进行负载平衡的服务器进行设计、编写和分布。无状态的服务不仅性能更好，而且还将大部分状态维护职责转移给客户端应用程序。在基于 REST 的 Web Services 中，服务器负责生成响应，并提供使客户端能够独自维护应用程序状态的接口。例如，在针对多页结果集的请求中，客户端应该包括要检索的实际页编号，而不是简单地要求检索下一页（参见图 11-2）。

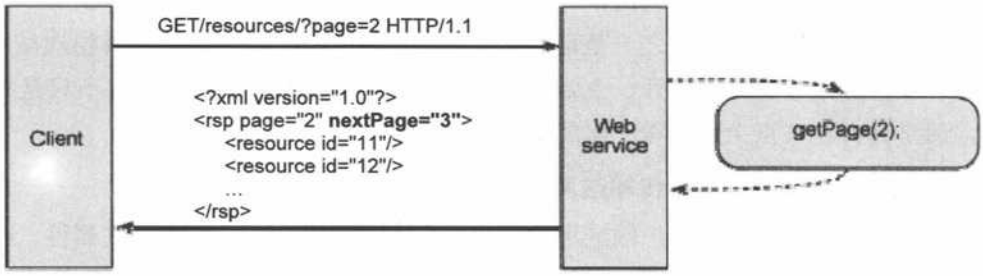


图 11-2　无状态的设计

无状态 Web Services 生成的响应链接到结果集中下一个页编号，并允许客户端完成所需的相关工作以便保留此值。可以将基于 REST 的 Web Services 设计在这个方面大致划分为以下两组职责，以阐明如何维护无状态的服务。

（1）服务器

生成响应，其中包括指向其他资源的链接，以使得应用程序可以在相关资源之间导航。此类响应嵌入了链接。类似的，如果请求针对父容器或子容器资源，则基于 REST 的典型响应还可能包括指向父资源的子资源或从属资源的链接，以便这些资源保持连接在一起。

生成响应，其中指明了是否可缓存，以通过减少针对重复资源的请求数量或通过完全消除某些请求来改进性能。服务器通过包括 Cache-Control 和 Last-Modified（日期值）HTTP 响应 Header 实现此目的。

（2）客户端应用程序

使用 Cache-Control 响应 Header 确定是否缓存资源（创建资源的本地副本）。客户端还读取 Last-Modified 响应 Header，并在 If-Modified-Since Header 中发回日期值，以向服务器询问资源是否已更改。这称为条件 GET（Conditional GET），两个 Header 同时进行，因为服务器的响应为标准 304 代码（Not Modified），如果请求的资源自从该时间以后尚未更改，则省略实际的资源。 HTTP 响应代码 304 意味着客户端可以安全地将资源表示形式在本地副本的缓存作为最新版本使用，从而实际上跳过了后续 GET 请求，直到资源更改为止。

发送可独立于其他请求得到服务的完整请求。这要求客户端充分利用 Web Services 接口指定的 HTTP Header，并在请求正文中发送完整的资源表示形式。客户端发送的请求极少对先前的请求、某个会话在服务器上的存在性、服务器向请求添加上下文的能力或请求之间保留的应用程序状态做出假设。

客户端应用程序与服务之间的这种协作对于基于REST的 Web Services中的无状态性极为重

要。它通过节省带宽和最小化服务器端应用程序状态改进了性能。

3. 公开目录结构式的 URI

从对资源寻址的客户端应用程序角度看，URI 决定了 REST Web Services 将具有的直观程度，以及服务是否将以设计人员能够按照预测的方式被使用。基于 REST 的 Web Services 第三个特征完全与 URI 相关。

REST Web Services URI 的直观性应该达到很容易猜测的程度。将 URI 看作是自身配备文档说明的接口，开发人员只需很少（如果有的话）的解释或参考资料即可了解它指向什么，并获得相关的资源。为此，URI 的结构应该简单、可预测且易于理解。

实现这种级别的可用性的方法之一是定义目录结构式的 URI。此类 URI 具有层次结构，其根为单个路径，从根开始分支的是公开、主要的服务子路径。根据此定义，URI 并不只是斜杠分隔的字符串，而是具有在节点上连接在一起的下级和上级分支树。

4. 传输 XML、JSON 或同时传输这两者

资源表示形式通常反映了在客户端应用程序请求资源时的资源当前状态及其属性。这种意义上的资源表示形式只是时间上的快照。 这可以像数据库中的记录表示形式一样简单，其中包括列名称与 XML 标记之间的映射，XML 的元素值包含行值。如果系统具有数据模型，那么根据此定义，资源表示形式是系统数据模型中对象之一的属性快照。这些对象就是希望开发者的 REST Web Services 为客户端提供的资源。

基于 REST 的 Web Services 设计中的最后一组约束与应用程序和服务在请求/响应有效负载或 HTTP 正文中交换的数据格式有关。这是真正值得将一切保持简单、可读和连接在一起的方面。

数据模型的对象通常以某种方式相关，应该以在将资源传输到客户端应用程序时表示资源的方式，反映数据模型对象（资源）之间的关系。 在讨论线程服务中，连接的资源表示形式的示例可能包括根讨论主题及其属性，以及指向为该主题提供响应的嵌入链接。

11.1.3 创建基于 REST 的 Web Services

要创建基于 REST 的 Web Services，开发者首先需要确定希望作为 Web Services 进行公开的所有资源。一些资源的示例包括雇员列表、雇员详细信息、订购单，等等。在 REST 中，每种资源都是通过唯一的统一资源标识符（Uniform Resource Identifier，URI）来标识的。开发或设计人员需要为每种资源确定唯一的 URI。例如，雇员列表可以标识如下：http://www.employee-details.com/employees-list。雇员详细信息可以使用如下所示的 URI 进行标识：http://www.employee-details.com/employees/01234。

使用 HTTP 操作 GET、PUT、POST 和 DELETE 来检索和修改资源。在用户的资源表示中提供一些超链接，以提供更多的相关信息。为这些资源的请求和响应数据指定格式，这需要 PUT 和 POST 操作。图 11-3 的关系图显示了与一个基于 REST 的 Web Services 之间的交互。

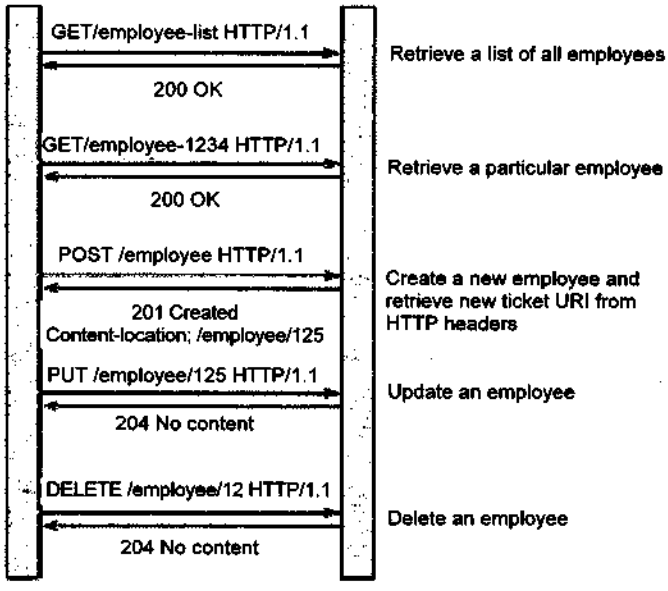


图 11-3　与一个基于 REST 的 Web Services 之间的交互

11.2　Apache CXF 的 RESTful 实现方式

目前，Apache CXF 框架有三种不同的方式支持 REST 风格的 Web Services。分别是 JAX-RS 方式、JAX-WS Provider & Dispatch 方式和 HTTP 绑定方式。

JAX-RS 方式：Apache CXF 框架实现了对 JAX-RS 的 1.0（执行的 JSR-311）的支持，这是一个 Java 的 REST 风格的 Web Services API。这提供了一个更加标准的方式来创建基于 Java 的 REST 风格的 Web Services。

JAX-WS Provider & Dispatch：这也可能通过 JAX-WS Provider 和 Dispatch 的 API 来创建 REST 风格的 Web Services。

HTTP 绑定方式：HTTP 绑定提供了一种创建资源并映射到服务上操作的灵活方式。目前可以通过注释或基于映射定义来实现。不过在将来的版本中 HTTP 绑定将逐渐被 Apache CXF 废弃。

11.2.1　JAX-RS 实现方式

JAX-RS （JSR-311）是为 Java EE 环境下 RESTful 服务能力提供的一种规范。JAX-RS 提供对传统的基于 SOAP 的 Web Services 一种可行的替代方案。同时，JAX-RS 也具有一个基于 Java 编程语言的 API，它提供了根据代表性状态传输（REST）体系结构风格来创建 Web Services 的支持。Apache CXF 框架支持 JAX-RS（符合 JSR-311），该规范定义了 Java 开发 RESTful 服务的方式和方法。

11.2.2 基本特征

1. JAX-RS 基本知识

JAX-RS 规范引入根资源、资源的方法、子资源、子资源定位器、消息正文的阅读器（MessageBodyReader）和消息正文写作器（MessageBodyWriter）的内容。

2. 支持数据绑定

JAX-RS 规范的 MessageBodyReader 和 MessageBodyWriter 实现了在不同数据格式上读取和写入的数据绑定。符合 JAX-RS 规范实现支持的数据格式包括：支持 JAXB 注释的 Bean 类，JAXP 源对象，InputStreams 等。此外，Apache CXF 框架的 JAX-RS 也可以兼容现在 Apache CXF 支持的 JAXB、XBeans、Aegis 和 SDO 等数据绑定。

3. 客户端 API

Apache CXF 框架的 JAX-RS 提供了复杂的 REST 客户端，支持三种模式：基于代理的模式、以 HTTP 为中心的模式和以 XML 为中心的模式。

4. Multiparts 支持

Multiparts 可以映射为 StreamSource、InputStream、数据源或自定义 Java 类型并被多种方式来处理。Apache CXF 核心运行包提供了基于 Apache CXF 框架的 JAX-RS 绑定来处理附件。

5. 对 XSLT 和 XPath 的支持

Apache CXF 框架的 JAX-RS 规范首先要支持 XSLT 和 XPath。XSLT 和 XPath 技术在处理复杂数据或检索复杂的 XML 文档方面具有强大功能。

6. 复杂的搜索查询

使用查询参数“beans”提供了一种方法来捕捉搜索。可以通过枚举，如名称/值来完成表达的查询要求，如“?name=CXF&version=2.3”可捕捉到一个包含 setname 方法和 setVersion 方法的 Bean。这个“template”Bean 可在代码上用于比较所获取的本地数据。

Apache CXF 框架的 JAX-RS 也支持另一种高级搜索查询，如自由项查询语言（FIQL）。

7. 调试

针对不同的 HTTP Accept 或 Accept 语言值和请求方法，使用浏览器来测试一个给定的 HTTP 资源会做出不同的反应。例如，资源类支持一个“/resource”的 URI ，然后可以使用下面查询代码之一来测试资源类：

```
> GET /resource.xml
> GET /resource.en
```

运行时将用适当的标头值来取代“.xml”或“.en”。因为它知道给定的 URI 后缀类型或语言值，需要做一些配置方面的事情。下面是一个 Spring 的配置例子。

```
<jaxrs:server id="customerService" address="/">
    <jaxrs:serviceBeans>
      <bean class="org.apache.cxf.jaxrs.systests.CustomerService" />
    </jaxrs:serviceBeans>
    <jaxrs:extensionMappings>
```

```
    <entry key="json" value="application/json"/>
    <entry key="xml" value="application/xml"/>
  </jaxrs:extensionMappings>
  <jaxrs:languageMappings/>
</jaxrs:server>
```

Apache CXF 框架也支持用_type 来定义参数的查询，如下：

```
{{ > GET /resource?_type=xml}}
```

重写请求方法也很简单：

```
> GET /resource?_method=POST
```

另外，可以指定 HTTP 头的 X-HTTP-Method-Override：

```
> POST /books
> X-HTTP-Method-Override : PATCH
```

例如，目前不支持以 HTTP 为中心的客户端 API，除了那些支持 HTTP 使用的 HttpURLConnection 方法。当需要时，X-HTTP-Method-Override 可以设置去覆盖这种限制。

8. 日志

现有 Apache CXF 框架中许多功能都可以在 jaxrs:server 或 jaxrs:client 上应用。例如，要启用日志记录的请求和响应，只需执行：

```
<beans xmlns:cxf="http://cxf.apache.org/core"
  xsi:schemaLocation="http://cxf.apache.org/core
    http://cxf.apache.org/schemas/core.xsd">
<beans>
<jaxrs:server>
<jaxrs:features>
          <cxf:logging/>
</jaxrs:features>
<jaxrs:server>
</beans>
```

9. 过滤器、拦截器和调用器（Invokers）

在 Apache CXF 框架的 JAX-RS 过滤器和拦截器的帮助下，很有可能是拦截或修改呼入和呼出调用。此外，定制化调用也提供了在调用服务 Bean 之前可以直接拦截调用的选项。

11.2.3 支持的特征

支持的特征包括安全的 JAX-RS 服务、SecurityManager 和 IllegalAccessExceptions、重定向、MVC 支持、服务列表和 WADL 支持、配置 JAX-RS 的服务、匹配请求 URI 集成、JAX-WS 和 JAX-RS、JAX-RS 和 Spring AOP、与分布式 OSGi 集成等内容。

1. 安全的 JAX-RS 服务

Apache CXF 框架有一个标准的样例\jax_rs\basic_https 显示了如何使用 HTTPS 通信。

采用像 Tomcat 的容器和 Spring 安全框架来进行用户认证处理。有时开发者可能想要做的就是自定义身份验证。最简单的方式，这是注册一个自定义调用或 RequestHandler 过滤器，将提取

用户名和密码，代码如下：

```
public class AuthenticationHandler implements RequestHandler {

    public Response handleRequest(Message m, ClassResourceInfo resourceClass) {
        AuthorizationPolicy policy = (AuthorizationPolicy)m.get(AuthorizationPolicy.class);
        policy.getUserName();
        policy.getPassword();
        return null;
    }
}
```

Apache CXF 框架有一个标准样例\jax_rs\spring_security 演示如何通过 Spring 来提供安全认证和安全授权。

2. SecurityManager 和 IllegalAccessExceptions

如果安装了 java.lang.SecurityManager，那么开发者可能需要用“suppressAccessChecks”许可来配置信任的 JAX-RS 以达到 JAX-RS 上下文或参数域的成功。例如，开发者可能需要修改 Tomcat 的 catalina.policy 下列权限：

```
grant codeBase "file:${catalina.home}/webapps/yourwebapp/lib/cxf.jar" {
    permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
};
```

3. 重定向

通过配置 CXFServlet 或使用 Apache CXF 框架的 JAX-RS RequestDispatcherProvider，Apache CXF 可以重定向请求或响应到其他 Servlet 资源。

4. MVC 支持

对于 XSLT 的支持，可以通过 XSLTJaxbProvider 生成复杂的（X）HTML 表现层。

对于 JSP 的支持。随着 RequestDispatcherProvider 的说明，JAX-RS 服务响应可能被重定向到 JSP 页面做进一步处理。请看这个示例，beans.xml 文件。除了“resourcePath”和“dispatcherName”属性，可以设置一个“scope”属性，有两个可能的值，“request”和“session”，但“request”是默认值。它影响的方式，JSP 代码可以检索通过 RequestDispatcherProvider 给它的参数。如果它是一个“request”范围，则所有的参数都设置为当前 HTTP 请求属性。如果它是一个“session”范围，则所有的参数都设置为当前 HTTP 会话属性。

RequestDispatcherProvider 设置下列参数：

- JAX-RS 方法响应对象。此参数的名称要么是一个简单类名，要么是复杂类名。对象（小写）或从 beanNames 地图检索属性值使用完全限定类名来对应这个简单类对象名称。
- 所有的路径、查询和矩阵参数已在方法执行期间初始化“absolute.path”、“base.path”和“relative.path”。

5. 服务列表和 WADL 支持

Apache CXF 框架的 JAX-RS 支持 WADL。Apache CXF 框架的 JAX-RS 服务端点可以列在服务器列表页面，用户可以检查 WADL 文件。

6. 配置 JAX-RS 的服务

JAX-RS 框架的服务可以通过 Spring 编程方式配置，也可以使用 CXFNonSpringJAXRSServlet 来进行配置。

7. 匹配请求 URI

这里有大量的变量。假设有一个 Web 应用程序，称为“rest”。CXFServlet 的 url-pattern 是“/test/*”。最后，jaxrs:server 的地址是“/bar”。

请求如/rest/test/bar 或/rest/test/bar/baz 将交付给特定 jaxrs:server 端点下的某一个资源类。对于前者请求要被处理，与@Path ("/")应提供资源类，在后一种情况，至少@Path ("/")或更具体的@Path（“/baz”）。

同样的请求可以在 CXFServlet 表示为“/*”和 jaxrs:server 用“/test/bar”。

当 CXFServlet 和 jaxrs:server 使用“/”，那么它是一个根资源类，应提供至少一个@Path “/test/bar”的上述要求进行匹配。

一般来说这种方式比较好，指定 URI 目录下属性，更有可能改变 CXFServlets 或 jaxrs:server。

8. 集成 JAX-WS 和 JAX-RS

Apache CXF 框架的 JAX-RS 尽力使 SOAP 开发人员更方便地使用 JAX-RS 的体验，并在同一个服务 bean 时结合双方需要的 JAX-WS 和 JAX-RS。

9. JAX-RS 和 Spring AOP

Apache CXF 框架的 JAX-RS 采用基于 Spring 的 AOP 拦截器。例如：

```
<beans xsi:schemaLocation=" http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/aop
  http://www.springframework.org/schema/aop/spring-aop.xsd
  http://cxf.apache.org/jaxrs
  http://cxf.apache.org/schemas/jaxrs.xsd">
  <import resource="classpath:META-INF/cxf/cxf.xml"/>
  <import resource="classpath:META-INF/cxf/cxf-extension-jaxrs-binding.xml"/>
  <import resource="classpath:META-INF/cxf/cxf-servlet.xml"/>

  <jaxrs:server id="bookservice" address="/">
    <jaxrs:serviceBeans>
        <ref bean="bookstore"/>
        <ref bean="bookstoreInterface"/>
      </jaxrs:serviceBeans>
   </jaxrs:server>
   <bean id="bookstore" class="org.apache.cxf.systest.jaxrs.BookStore"/>
   <bean id="bookstoreInterface" class="org.apache.cxf.systest.jaxrs.BookStoreWithInterface"/>

   <aop:config>
    <aop:aspect id="loggingAspect" ref="simpleLogger">
        <aop:before method="logBefore"
           pointcut="execution(*
org.apache.cxf.systest.jaxrs.BookStore*.*(..))"/>
        <aop:after-returning method="logAfter"
           pointcut="execution(*
org.apache.cxf.systest.jaxrs.BookStore*.*(..))"/>
      </aop:aspect>
```

```
  </aop:config>
  <bean id="simpleLogger"
       class="org.apache.cxf.systest.jaxrs.SimpleLoggingAspect"/>
</beans>
```

请注意，有些 AOP 配置应用于两个 JAX-RS 的资源类。默认情况下 Spring 使用 JDK 动态代理类，如果一个类要被代理实现，至少有一个接口或有其他方式 CGLIB 代理。

例如，下面是 org.apache.cxf.systest.jaxrs.BookStoreWithInterface 的代码例子：

```
public interface BookInterface {
   @GET
   @Path("/thosebooks/{bookId}/")
   @Produces("application/xml")
   Book getThatBook(Long id) throws BookNotFoundFault;
}

public class BookStoreWithInterface extends BookStoreStorage
   implements BookInterface {

   public Book getThatBook(@PathParam("bookId") Long id)
      throws BookNotFoundFault {
      return doGetBook(id);
   }

   @Path("/thebook")
   public Book getTheBook(@PathParam("bookId") Long id)
      throws BookNotFoundFault {
      return doGetBook(id);
   }
}
```

在这种情况下，Spring 将使用 JDK 动态代理类包装一个 BookStoreWithInterface。因此，它一个重要的方法需要被调用，如 getThatBook (...)会是接口的一部分。

另一种方法，getTheBook 方法不能在 JAX-RS 运行时被分发，因为它不可能通过一个 JDK 代理来发现。如果此方法还需要调用，则此方法应该要么加入接口或者必须显式启用 CGLIB 代理。例如：

```
<aop:config proxy-target-class="true"/>
```

10. 与分布式 OSGi 集成

分布式 OSGi RI（DOSGi RI）是 Apache CXF 项目的一个子项目。关于 DOSGi 如何注册成为 Java 接口可以去了解关于任务和消费远程服务的相关知识。DOSGi 单一部署和多节点分布包含所有的 OSGi 包，Apache CXF 的端点可以被成功地发布。

Apache CXF 框架的 JAX-RS 规范实现已经集成 DOSGi RI 1.1-SNAPSHOT。这使得它有可能暴露 Java 接口作为 RESTful 服务和使用基于代理的客户端 API 来消费服务。

11.2.4 其他先进功能

Apache CXF 框架的 JAX-RS 提供更多的扩展功能，如支持 JMS 传输、单向调用（HTTP 和 JMS），悬浮调用（HTTP 和 JMS）、使用现有 REST-aware 代码去扩展用户应用模式。

11.3　JAX-WS Provider 和 Dispatch 实现方式

Apache CXF 框架也支持使用 JAX-WS 的 Provider/Dispatch 和 XML 绑定来创建 REST 风格的 Web Services。在 Apache CXF 的标准样例 samples\restful_dispatch 中有相应的例子程序。

例子 samples\restful_dispatch 可以分为服务器端和客户端程序。

1. 服务器端的实现和功能

服务器的代码演示了如何通过 JAX-WS 的 Provider 接口构建一个 RESTful 端点。

服务器端的 Server.java 负责发布遵循 JAX-WS 规范的 Web Services，采用 javax.xml.ws.Endpoint 方式发布，在生成 Endpoint 时调用如下代码：

```
Endpoint e = Endpoint.create(HTTPBinding.HTTP_BINDING, new RestSourcePayloadProvider());
```

把 RestSourcePayloadProvider 进行了实例化，并用 HTTPBinding 绑定方式加载到 Endpoint 对象中。

RestSourcePayloadProvider 是实现 javax.xml.ws.Provider 接口的实现类。其主要作用是对本地 XML 的读取，然后转换为 javax.xml.transform.dom.DOMSource 格式，代码如下：

```
private DOMSource createDOMSource(String fileName) {
        DocumentBuilderFactory factory;
        DocumentBuilder builder;
        Document document = null;
        DOMSource response = null;

        try {
            factory = DocumentBuilderFactory.newInstance();
            builder = factory.newDocumentBuilder();
            InputStream greetMeResponse = getClass().getResourceAsStream(fileName);

            document = builder.parse(greetMeResponse);
            response = new DOMSource(document);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return response;
    }
```

然后采用 Response 方式以 DOMSource 对象返回，所以服务端和客户端采用的传输对象是 javax.xml.transform.dom.DOMSource 格式。

对 REST 服务器提供以下服务：一个 RESTful 客户服务提供网址 http://localhost:9000/customerservice/customers。用户访问该网址查询或更新客户信息。在 Server.java 的代码为：

```
String address = "http://localhost:9000/customerservice/customer";
e.publish(address);
```

2. 客户端的实现和功能

客户端代码演示了如何基于 JAX-WS 的 Dispatch 方式，通过 HTTP POST 命令来发送 XML 数据，以及如何使用 URL.openStream 方法来发送 HTTP 的 GET。

首先介绍一下通过 HTTP GET 请求模式获取服务器的数据。

客户端 HTTP GET 的代码如下：

```
  String endpointAddress = "http://localhost:9000/customerservice/customer";

URL url = new URL(endpointAddress); //定义 URL 地址
InputStream in = url.openStream();//获取 URL 地址放回的数据流
StreamSource source = new StreamSource(in); //转换数据流为 javax.xml.transform.stream.
//StreamSource 对象
  printSource(source);// 列出这个 StreamSource 的内容
```

获取了 StreamSource 后，接着在客户端列出这个 StreamSource 的内容，代码如下：

```
private static void printSource(Source source) {
      try {
          ByteArrayOutputStream bos = new ByteArrayOutputStream();
          StreamResult sr = new StreamResult(bos);
          Transformer trans = TransformerFactory.newInstance().newTransformer();
          Properties oprops = new Properties();
          oprops.put(OutputKeys.OMIT_XML_DECLARATION, "yes");
          trans.setOutputProperties(oprops);
          trans.transform(source, sr);

          System.out.println(bos.toString());//列出 ByteArrayOutputStream 的内容
          bos.close();
          System.out.println();
      } catch (Exception e) {
          e.printStackTrace();
      }
  }
```

具体 HTTP GET 请求的内容和返回的结果如下：

一个 HTTP GET 请求到网址 http://localhost:9000/customerservice/customer 返回一个客户名单的超链接，返回的 XML 文档如下。

```
<Customers>
  <Customer href="http://localhost/customerservice/customer?id=1234">
      <id>1234</id>
  </Customer>
  <Customer href="http://localhost/customerservice/customer?id=1235">
      <id>1235</id>
  </Customer>
  <Customer href="http://localhost/customerservice/customer?id=1236">
      <id>1236</id>
  </Customer>
</Customers>
```

一个 HTTP GET 请求到 URL http://localhost:9000/customerservice/customers?id=1234 返回一个客户的实例，其编号为 1234。返回的 XML 文档如下。

```
<Customer>
  <id>1234</id>
  <name>John</name>
  <phoneNumber>123456</phoneNumber>
</Customer>
```

下面介绍一下通过 HTTP 的 POST 请求模式来修改服务器的数据。

HTTP 的 POST 请求实现主要涉及 javax.xml.ws.Service、javax.xml.ws.Dispatch 和 javax.xml.transform.dom.DOMSource 等几个对象。其过程描述如下。

（1）首先实例化一个 javax.xml.ws.Service，代码如下：

```
QName serviceName = new QName("http://apache.org/hello_world_xml_http/wrapped",
"cutomerservice");
  QName portName = new QName("http://apache.org/hello_world_xml_http/wrapped",
"RestProviderPort");
  String endpointAddress = "http://localhost:9000/customerservice/customer";
……
Service service = Service.create(serviceName);
service.addPort(portName, HTTPBinding.HTTP_BINDING, endpointAddress);
……
```

（2）通过 javax.xml.ws.Service 对象创建出一个 javax.xml.ws.Dispatch 实例化对象，代码如下：

```
Dispatch<DOMSource> dispatcher =
   service.createDispatch(portName,DOMSource.class, Service.Mode.PAYLOAD);
Map<String, Object> requestContext = dispatcher.getRequestContext();
……
```

（3）创建一个 javax.xml.transform.dom.DOMSource 对象。当然，要通过本地的 XML 文件来生成，代码如下：

```
……
Client client = new Client();
   InputStream is = client.getClass().getResourceAsStream("CustomerJohnReq.xml");
   Document doc = XMLUtils.parse(is);
DOMSource reqMsg = new DOMSource(doc);
……
```

（4）对服务器提交 HTTP POST 请求，传入 DOMSource 对象参数，代码如下：

```
requestContext.put(MessageContext.HTTP_REQUEST_METHOD, "POST");
   System.out.println("Invoking through HTTP POST to update customer using JAX-WS
Dispatch");
   DOMSource result = dispatcher.invoke(reqMsg);
   printSource(result); //列出这个 DOMSource 的内容
```

一个 HTTP POST 请求的数据到网址 http://localhost:9000/customerservice/customers：

```
<Customer>
  <id>1234</id>
  <name>John</name>
  <phoneNumber>234567</phoneNumber>
</Customer>
```

使用提供的数据来更新客户信息为 1234。

11.4　HTTP 绑定方式

HTTP 绑定提供了一种创建资源并映射到服务上操作的灵活方式。目前可以通过注释或基于

映射定义来实现。不过在 Apache CXF 将来的版本中 HTTP 绑定将被废弃，在这里也是简单地做一个介绍。在 samples\ restful_http_binding 中有相应的例子程序。

例子 samples\ restful_http_binding 可以分为服务器端和客户端程序。

1. 服务器端的实现和功能

服务器端演示了三种不同的端点实现，分别是 RESTful XML 端点、RESTful JSON 端点和 SOAP 端点。后面增加了静态的 HTTP 文件发布。

服务器端的 Server.java 负责发布遵循 JAX-WS 规范的 Web Services，采用 JaxWsServerFactoryBean 方式发布。

SEI 的代码如下：

```
package demo.restful.server;

import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;

import org.codehaus.jra.Delete;
import org.codehaus.jra.Get;
import org.codehaus.jra.HttpResource;
import org.codehaus.jra.Post;
import org.codehaus.jra.Put;

@WebService(targetNamespace = "http://demo.restful.server")
public interface CustomerService {
    @Get
    @HttpResource(location = "/customers")
    @WebResult(name = "Customers")
    Customers getCustomers();

    @Get
    @HttpResource(location = "/customers/{id}")
    Customer getCustomer(@WebParam(name = "GetCustomer")
                         GetCustomer getCustomer) throws CustomerNotFoundFault;

    @Put
    @HttpResource(location = "/customers/{id}")
    void updateCustomer(@WebParam(name = "Customer") Customer c);

    @Post
    @HttpResource(location = "/customers")
    long addCustomer(@WebParam(name = "Customer") Customer c);

    @Delete
    @HttpResource(location = "/deleteCustomer/{id}")
    void deleteCustomer(@WebParam(name = "DeleteCustomer")
        DeleteCustomer deleteCustomer) throws CustomerNotFoundFault;
}
```

都是采用@HttpResource 方式来定制化注释信息。

第一种是 RESTful 的 XML 服务器端点，其发布如下代码：

```
private static void createRestService(Object serviceObj) {
        // 创建服务器的 factory bean
```

```
        JaxWsServerFactoryBean sf = new JaxWsServerFactoryBean();
        sf.setServiceClass(CustomerService.class);
        // 使用了解 Java Rest 注释的 HTTP Binding
        sf.setBindingId(HttpBindingFactory.HTTP_BINDING_ID);
        sf.setAddress("http://localhost:8080/xml/");
        sf.getServiceFactory().setInvoker(new BeanInvoker(serviceObj));

      // 关闭“wrapped”属性，意思是不会产生有“wrapped”属性的 XML 文档，可以是自定义的 XML 内容
        sf.getServiceFactory().setWrapped(false);
        sf.create();
    }
```

第二种是 RESTful 的 JSON 服务器端点，其发布如下代码：

```
private static void createJsonRestService(Object serviceObj) {
        // 创建服务器的 factory bean
        JaxWsServerFactoryBean sf = new JaxWsServerFactoryBean();
        sf.setServiceClass(CustomerService.class);
        // 使用了解 Java Rest 注释的 HTTP Binding
        sf.setBindingId(HttpBindingFactory.HTTP_BINDING_ID);
        sf.setAddress("http://localhost:8080/json");
        sf.getServiceFactory().setInvoker(new BeanInvoker(serviceObj));

        //关闭“wrapped”属性，意思是不会产生有“wrapped”属性的 JSON 文档，可以是自定义的 JSON 内容
        sf.getServiceFactory().setWrapped(false);

        //告诉 CXF 对 JSON 端点使用不同的 Content-Type。这也许是“application/json”
        //但“text/plain”允许我们在 Web 浏览器查看更容易
        Map<String, Object> properties = new HashMap<String, Object>();
        properties.put("Content-Type", "text/plain");

        // 创建 JSON StAX 实现
        Map<String, String> nstojns = new HashMap<String, String>();
        nstojns.put("http://demo.restful.server", "acme");

        MappedXMLInputFactory xif = new MappedXMLInputFactory(nstojns);
        properties.put(XMLInputFactory.class.getName(), xif);

        MappedXMLOutputFactory xof = new MappedXMLOutputFactory(nstojns);
        properties.put(XMLOutputFactory.class.getName(), xof);

        sf.setProperties(properties);
        sf.create();
    }
```

第三种是 SOAP 服务器端点，其发布如下代码：

```
private static void createSoapService(Object serviceObj) {
        JaxWsServerFactoryBean sf = new JaxWsServerFactoryBean();
        sf.setServiceClass(CustomerService.class);
        sf.setAddress("http://localhost:8080/soap");
        sf.getServiceFactory().setInvoker(new BeanInvoker(serviceObj));
        sf.getServiceFactory().setWrapped(false);
        sf.create();
    }
```

第四种是静态的 HTTP 文件发布，其实现如下代码：

```
private static void serveHTML() throws Exception {
        Bus bus = BusFactory.getDefaultBus();
        DestinationFactoryManager dfm = bus.getExtension(DestinationFactoryManager.class);
        DestinationFactory df = dfm.getDestinationFactory("http://cxf.apache.org/
transports/http/configuration");

        EndpointInfo ei = new EndpointInfo();
        ei.setAddress("http://localhost:8080/test.html");

        Destination d = df.getDestination(ei);
        d.setMessageObserver(new MessageObserver() {
            public void onMessage(Message message) {
                try {
                    // HTTP seems to need this right now...
                    ExchangeImpl ex = new ExchangeImpl();
                    ex.setInMessage(message);
                    Conduit backChannel = message.getDestination().getBackChannel
(message, null, null);

                    MessageImpl res = new MessageImpl();
                    res.put(Message.CONTENT_TYPE, "text/html");
                    backChannel.prepare(res);

                    OutputStream out = res.getContent(OutputStream.class);
                    FileInputStream is = new FileInputStream("test.html");
                    IOUtils.copy(is, out, 2048);

                    out.flush();
                    out.close();
                    is.close();
                    backChannel.close(res);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
    }
```

2. 客户端的实现和功能

客户端的实现很简单，就是用 URL 去访问，代码如下：

```
URL url = new URL("http://localhost:8080/xml/customers");
  InputStream in = url.openStream();
  System.out.println(getStringFromInputStream(in));
```

11.5 CXF 实现 RESTful 服务的例子说明

Apache CXF 实现 RESTful 服务总共有九个例子。其实现的场景覆盖 HttpClient 实现、WebClient 实现、JAX-WS Provider 和 Dispatch 实现、HTTP 绑定实现。

11.5.1 CXF 采用 HttpClient 实现基本的 RESTful 应用

本程序演示 Apache CXF 框架采用 HttpClient 方式实现 RESTful 的服务器端，同时客户端进

行调用，返回 Respone。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，都采用 RESTful 的方式来进行处理。前端访问采用 HTTP 方式，后台服务器为遵循 JAX-RS 规范的 RESTful 来实现资源。实现的结构如图 11-4 所示。

其中：①表示客户端通过 HTTP 访问后台的项目基本情况；②表示客户端通过 HTTP 修改默认项目；③表示客户端通过 HTTP 增加一个项目；④表示客户端通过 HTTP 删除一个项目；⑤表示客户端通过 http 查看后台的全部项目情况。

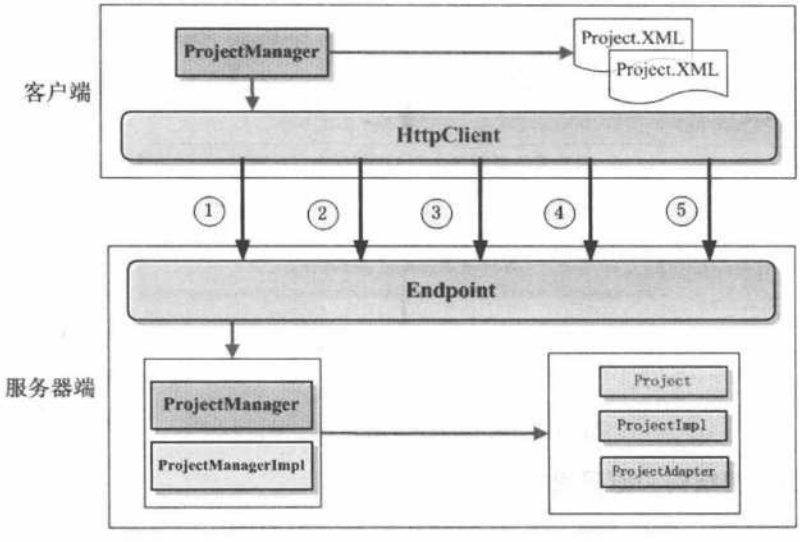


图 11-4　基于 HttpClient 实现 RESTful 应用结构图

直接通过 HTTP 访问页面方式来获得相关的信息。这里使用了 org.apache.commons.httpclient 组件。所以程序要引入 commons-httpclient-3.1.jar 包。

2. 应用程序案例实现步骤

在设计 REST 的 Web Services 时，可以采用以下四个步骤：①决定资源及其描述性 URL；②为每个 URL 的通信选择一种数据格式；③指定每个资源上的方法；④指定返回的数据和状态码。

要创建 HttpClient 并基于 JAX-RS 服务的前端实现 Web Services 程序的流程步骤如图 11-5 所示。

流程图说明，要实现以下步骤：

（1）创建一个 Java 项目。

（2）创建一个基于 JAX-RS 服务的服务端点接口（SEI）及其相关对象和注释。

（3）编写发布这个 Java 程序为 Web Services 的服务器端程序，支持 JAX-RS 服务。

（4）采用 HttpClient 并基于 JAX-RS 服务的前端客户端应用程序。

（5）启动服务器程序，对外发布基于 JAX-RS 服务的 Web Services。

（6）基于 JAX-RS 服务的前端客户端调用后台基于 JAX-RS 服务的 Web Services。

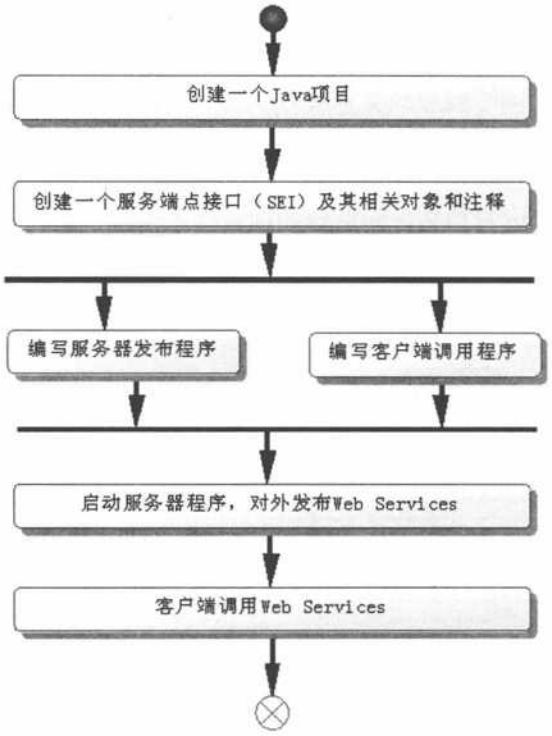


图 11-5　应用程序案例实现步骤流程图

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 11-6 所示。

图 11-6　jaxrs_httpclient 程序目录结构

包目录包括三个部分，第一个包目录 demo.cxf.frontends.javafirst.server 是服务器程序目录。第二

个包目录 demo.cxf.frontends.javafirst.client 是客户端程序目录。第三个包目录 demo.cxf.webservices 是服务器端点接口 SEI 和相关对象等公用部分，服务器端和客户端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口，服务器端和客户端都必须引用。但服务器端和客户端对公用部分的引用是不同的。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口。而服务器端程序是接口和实现类都要引用。

下面分别介绍三部分：①服务器端点接口 SEI 和相关对象；②服务器端发布程序；③客户端调用程序。

（1）服务器端点接口（SEI）和相关对象

首先要介绍的是服务器端点接口（SEI）。

服务器端对于 Project 资源类（UserResourc）的实现，其核心的功能是响应有关用户资源的 HTTP GET/PUT/DELETE 请求，而这些请求响应逻辑正对应了 UserRestHelper 类中关于用户资源类的 HTTP 请求。

Project 组件包括 WebService 包下的 Project 几个文件。

ProjectManager 的程序代码如下：

```
package demo.cxf.webservices;

import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.core.Response;

@Path("/ProjectManager/")
public interface ProjectManager {

    @GET
    @Path("/getProjectName/")
    public  Response getProjectName();

    @GET
    @Path("/getProjectNameFromID/{projectID}/")
    public Response getProjectNameFromID(@PathParam("projectID") String  projectID);

    @GET
    @Path("/greeting/")
    public Response greeting() ;

    @GET
    @Path("/greetingWithParameter/{greet}/")
    public Response greetingWithParameter(@PathParam("greet") String greet);

    @GET
    @Path("/updateProjectName/{projectName}/")
    public Response updateProjectName (@PathParam("projectName") String projectName);

    @PUT
    @Path("/setDefaultProject/")
    public Response setDefaultProject( Project project);
```

```
    @POST
    @Path("/registerProject/")
    public Response registerProject(Project project);

    @DELETE
    @Path("/removeProject/{id}/")
    public Response removeProject(@PathParam("id") String id);

    @GET
    @Path("/getDefaultProject/")
    //@Produces("application/xml")
    public Project getDefaultProject();

    @GET
    @Path("/getAllProject/")
    public Response getAllProject() ;

}
```

当然这只是一个接口，还有一个实现类 ProjectManagerImpl，代码如下：

```
package demo.cxf.webservices;

import java.util.HashMap;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Response;
import javax.xml.bind.annotation.XmlRootElement;

@Path("/ProjectManager/")
//@Produces("application/xml")
public class ProjectManagerImpl implements ProjectManager {
    private Project project;
    private HashMap<String, Project> projectMap;

    public ProjectManagerImpl() {
        System.out.println("----ProjectManagerImpl Construct----");
        initProject();
    }

    public Response greeting() {
        return Response.ok("Hello, This is ProjectManager.").build();
    }

    public Response greetingWithParameter(String greet) {
        return Response.ok("Hello, " + greet + " This is ProjectManager.")
                .build();
    }

    public Response getProjectName() {
        System.out.println("----invoking getProjectName ");
        Response response;
        if (project.getProjectName() == null) {
            response = Response.notModified().build();
        } else {
            response = Response.ok(project.getProjectName()).build();
        }
        return response;
    }
```

```
    public Response getProjectNameFromID(String projectID) {
        System.out.println("----invoking getProjectNameFromID, projectID is "
                + projectID);
        Response response;
        if (projectID == null)
            return Response.notModified().build();
        if (!projectMap.containsKey(projectID)) {
            response = Response.notModified().build();
        } else {
            response = Response.ok(projectMap.get(projectID).getProjectName())
                    .build();
        }
        return response;
    }

    public Response updateProjectName(String projectName) {
        Response response;
        System.out.println("----invoking updateProjectName, Project name is: "+projectName);
        project.setProjectName(projectName);
        response = Response.ok(project.getProjectName()).build();
        return response;
    }

    public Response setDefaultProject(Project project) {
        Response response;
        if (project == null)
            Response.notModified().build();
        System.out.println("----invoking setDefaultProject, Project name is: "+project.
getProjectName());
        this.project = project;
        response = Response.ok(this.project.getProjectName()).build();
        return response;
    }

    public Response registerProject(Project project) {
        Response response;
        if (project == null) return Response.notModified().build();
        System.out.println("----invoking registerProject, Project ID is: "
                + project.getProjectID());
        if (projectMap.containsKey(project.getProjectID()))
            return Response.notModified().build();
        projectMap.put(project.getProjectID(), project);
        //System.out.println(projectMap.size());
        response = Response.ok(project.getProjectName()).build();
        return response;
    }

    public Response removeProject(String id) {
        Response response;
        System.out.println("----invoking removeProject, Project ID is: "
                + id);
        if (!projectMap.containsKey(id))
            return Response.notModified().build();
        projectMap.remove(id);
        response = Response.ok("delete is ok.").build();
        return response;
    }
```

```
    // @Produces("application/xml")
    public Project getDefaultProject() {
        System.out.println("----invoking getDefaultProject ");
        return project;
    }

    public Response getAllProject() {
        Response response;
        System.out.println("----invoking getAllProject ");
        response = Response.ok(projectMap.values().toString()).build();
        return response;
    }

    private void initProject() {
        project = new ProjectImpl();
        project.setProjectID("default projectID");
        project.setProjectName("default projectName");
        project.setProjectDescribe("default projectDescribe");

        projectMap = new HashMap<String, Project>();

        Project project2 = new ProjectImpl();
        project2.setProjectID("projectID2");
        project2.setProjectName("projectName2");
        project2.setProjectDescribe("projectDescribe2");

        projectMap.put("1", project);
        projectMap.put("2", project2);
    }
}
```

（2）服务器端发布程序

服务器端发布程序是 Server.java 文件，其作用就是启动服务器。服务器端代码如下：

```
package demo.cxf.frontends.javafirst.server;

import org.apache.cxf.jaxrs.JAXRSServerFactoryBean;
import org.apache.cxf.jaxrs.lifecycle.SingletonResourceProvider;
import demo.cxf.webservices.ProjectManager;
import demo.cxf.webservices.ProjectManagerImpl;

public class Server {

    public Server() {
        System.out.println("Starting Server");

        ProjectManager projectManager = new ProjectManagerImpl();
        JAXRSServerFactoryBean sf = new JAXRSServerFactoryBean();
       sf.setResourceClasses(ProjectManager.class);
       sf.setResourceProvider(ProjectManager.class, new SingletonResourceProvider
(projectManager));
       sf.setAddress("http://localhost:9000/");
       sf.create();
    }

    public static void main(String[] args) throws Exception {
        new Server();
        System.out.println("Server ready...");
```

```
        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

采用了 JAXRSServerFactoryBean 对象。首先实例化给对象，然后载入 Project 资源类并把服务发布出来。

（3）客户端调用程序

客户端的核心实现部分主要由四部分组成：使用 HTTP PUT 增加、修改用户资源；使用 HTTP GET 得到某一具体用户资源；使用 HTTP DELETE 删除用户资源；使用 HTTP GET 得到用户列表资源。

客户端程序文件包括 CXFClient.java、add_project.xml 和 update_project.xml，其中 CXFClient.java 是实现客户端调用的主程序。add_project.xml 和 update_project.xml 是用于分别进行增加和修改 Bean 的 XML 文件。

客户端实现查询服务器的项目基本情况的代码如下：

```
//不传入参数和传入参数两种方式来获得后台的信息
    private static void getGreetResponse()throws Exception {
        URL url = null;
        InputStream in = null;

        System.out.println("Sent HTTP GET request to greeting info.");
        url = new URL("http://localhost:9000/ProjectManager/greeting");
        in = url.openStream();
        System.out.println(getStringFromInputStream(in));

        System.out.println("\n");
        System.out.println("Sent HTTP GET request to greetingWithParameter.");
        url = new URL("http://localhost:9000/ProjectManager/greetingWithParameter/
greetisgood");
        in = url.openStream();
        System.out.println(getStringFromInputStream(in));

    }

    //无参数获取后台项目的细节数据
    private static void getDefaultProjectName() throws Exception{
        URL url = null;
        InputStream in = null;
        System.out.println("\n");
        System.out.println("Sent HTTP GET request to getProjectName.");
        url = new URL("http://localhost:9000/ProjectManager/getProjectName");
        in = url.openStream();
        System.out.println(getStringFromInputStream(in));
    }

    //通过参数来获取后台项目的细节数据
    private static void getProjectNameFromID() throws Exception{
        URL url = null;
        InputStream in = null;
        System.out.println("\n");
        System.out.println("Sent HTTP GET request to getProjectNameFromID.");
```

```
        url = new URL("http://localhost:9000/ProjectManager/getProjectNameFromID/2");
        in = url.openStream();
        System.out.println(getStringFromInputStream(in));
    }
```

客户端实现修改默认项目的代码如下：

```
//通过本地 XML 文件形成 Bean，然后作为输入参数修改后台的默认项目数据
   private static void updateProject(CXFClient client) throws Exception{
    System.out.println("\n");
        System.out.println("Sent HTTP PUT request to update project.");
        String inputFile = client.getClass().getResource("update_project.xml").getFile();
        File input = new File(inputFile);
        PutMethod put = new PutMethod( "http://localhost:9000/ProjectManager/
setDefaultProject");
        RequestEntity entity = new FileRequestEntity(input,"text/xml; charset=UTF-8");
        put.setRequestEntity(entity);
        HttpClient httpclient = new HttpClient();

        try {
            int result = httpclient.executeMethod(put);
            System.out.println("Response status code: " + result);
            System.out.println("Response body: "+ put.getResponseBodyAsString());
        } finally {
            // 释放当前正在处理的连接池
            put.releaseConnection();
        }
    }
```

在上述代码中，其项目是通过读取本地的 XML 文件来形成一个实体 Bean，通过 Bean 来修改后台的项目库。

客户端实现增加一个项目的代码如下：

```
//通过本地 XML 文件形成 Bean，然后作为输入参数在后台增加一个项目
    private static void addProject(CXFClient client) throws Exception{
        System.out.println("\n");
        System.out.println("Sent HTTP POST request to add project");

        String inputFile = client.getClass().getResource("add_project.xml").getFile();
        File input = new File(inputFile);
        PostMethod post = new PostMethod("http://localhost:9000/ProjectManager/
registerProject");
        post.addRequestHeader("Accept", "text/xml");

        FileRequestEntity entity = new FileRequestEntity(input, "text/xml; charset=ISO-8859-1");
        post.setRequestEntity(entity);
        HttpClient httpclient = new HttpClient();

        try {
            int result = httpclient.executeMethod(post);
            System.out.println("Response status code: " + result);
            System.out.println("Response body: "+ post.getResponseBodyAsString());
        } finally {
            // 释放当前正在处理的连接池
            post.releaseConnection();
        }
    }
```

在上述代码中，其项目是通过读取本地的 XML 文件来形成一个实体 Bean，通过 Bean 来增加后台的项目库。

客户端实现删除一个项目的代码如下：

```
//通过输入项目 ID 为参数来删除后台的一个项目
    private static void deleteProject() throws Exception{
        System.out.println("\n");
        System.out.println("Sent HTTP POST request to delete project");
        DeleteMethod delete = new DeleteMethod(
                "http://localhost:9000/ProjectManager/removeProject/10");
        //delete.addRequestHeader("Accept", "text/xml");

        HttpClient httpclient = new HttpClient();

        try {
            int result = httpclient.executeMethod(delete);
            System.out.println("Response status code: " + result);
            System.out.println("Response body: "
                    + delete.getResponseBodyAsString());
        } finally {
            // 释放当前正在处理的连接池
            delete.releaseConnection();
        }
    }
```

在删除过程中，直接说明项目的 ID，然后删除后台的项目库。

客户端实现查看后台的全部项目情况的代码如下：

```
//查看后台所有的项目信息，格式是采用 Bean 的内部编码
    private static void getAllProject() throws Exception {
        System.out.println("\n");
        System.out.println("Sent HTTP GET request to getAllProject.");
        URL url = new URL("http://localhost:9000/ProjectManager/getAllProject");
        InputStream in = url.openStream();
        System.out.println(getStringFromInputStream(in));
    }
```

在删除过程中，直接说明项目的 ID，然后删除后台的项目库。

使用标准的 HTTP 状态作为状态码。成功的操作都会返回 200。如果系统无法根据资源的 ID 找到它，就会返回 404。系统故障导致的任何错误都会返回 500。

4. 测试和运行结果

测试编写好的程序代码，首先要启动服务器程序，即在 Eclipse 上运行 Server.java。然后运行客户端程序，客户端程序输出结果如下：

```
Sent HTTP GET request to greeting info.
Hello, This is ProjectManager.

Sent HTTP GET request to greetingWithParameter.
Hello, greetisgood This is ProjectManager.

Sent HTTP GET request to getProjectName.
default projectName
```

```
Sent HTTP GET request to getProjectNameFromID.
projectName2


Sent HTTP GET request to getAllProject.
[demo.cxf.webservices.ProjectImpl@10d9151,
demo.cxf.webservices.ProjectImpl@1bf68a9]


Sent HTTP PUT request to update project.
Response status code: 200
Response body: XmlProjectName2


Sent HTTP GET request to getProjectName.
XmlProjectName2


Sent HTTP GET request to getAllProject.
[demo.cxf.webservices.ProjectImpl@10d9151,
demo.cxf.webservices.ProjectImpl@1bf68a9]


Sent HTTP POST request to add project
Response status code: 200
Response body: XmlProjectName10


Sent HTTP GET request to getAllProject.
[demo.cxf.webservices.ProjectImpl@cd83d8,
demo.cxf.webservices.ProjectImpl@10d9151,
demo.cxf.webservices.ProjectImpl@1bf68a9]


Sent HTTP POST request to delete project
Response status code: 200
Response body: delete is ok.


Sent HTTP GET request to getAllProject.
[demo.cxf.webservices.ProjectImpl@10d9151,
demo.cxf.webservices.ProjectImpl@1bf68a9]
```

5. 其他内容

上述的所有调用都基于 JAX-RS 规范。

本例代码参见配套资源代码中“第 11 章\jaxrs_httpclient”文件夹。

11.5.2 CXF 采用 HttpClient 在 Servlet 实现基本的 RESTful 应用

本程序演示 Apache CXF 框架采用 HttpClient 来实现客户端和 Servlet 服务器端的 RESTful 调用，客户端采用 Apache CXF 的代码优先（Java First）前端应用模式。服务器端采用 Servlet 容器。本例 Servlet 容器采用 Tomcat 6。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询和删除操作。

在这个过程中，前端客户端采用 Apache CXF 框架自身规范的 HttpClient 对象来实现，后台服务端为遵循 JAX-RS 规范的 Servlet 容器来实现。实现的结构如图 11-7 所示。

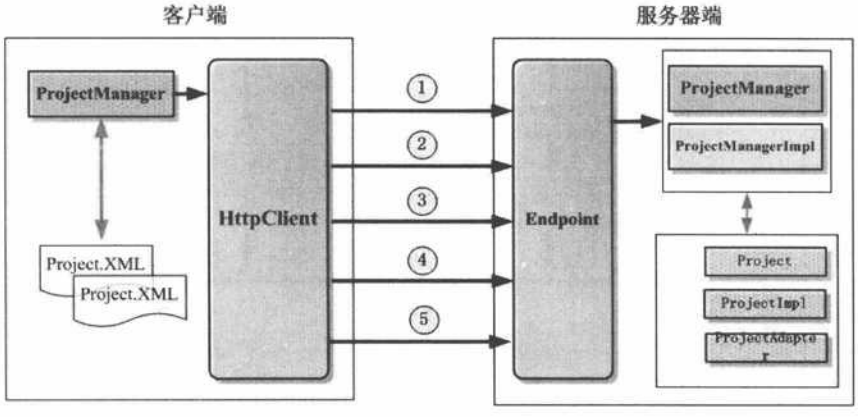


图 11-7　基于 HttpClient 实现 Servlet 的 RESTful 应用结构图

其中：① 表示客户端通过 HTTP 访问后台的项目基本情况；② 表示客户端通过 HTTP 修改默认项目；③ 表示客户端通过 HTTP 增加一个项目；④ 表示客户端通过 HTTP 删除一个项目；⑤ 表示客户端通过 HTTP 查看后台的全部项目情况。

2. 应用程序案例实现步骤

要基于 JAX-RS 服务创建 HttpClient，并采用 Servlet 容器的前端实现 Web Services 程序的步骤流程如图 11-8 所示。

流程图说明，要实现以下步骤：

（1）创建一个 Web 服务项目。

（2）创建一个基于 JAX-RS 服务的服务端点接口（SEI）及其相关对象和注释。

（3）采用 HttpClient 方式，编写基于 JAX-RS 服务的 Web Services 的客户端程序。

（4）编写基于 JAX-RS 服务的 web.xml 和 server-bean.xml。

（5）发布配置文件到 Servlet 服务器。

（6）启动 Servlet 服务器程序，对外发布基于 JAX-RS 服务的 Web Services。

（7）基于 HttpClient 方式的客户端调用后台基于 JAX-RS 服务的 Web Services。

其中第 1 步和第 3 步与前面采用自带的 HTTP 服务器相似。主要是第 2 步，该步骤不是采用自带的服务，而是放置到 Servlet 容器中。我们采用与 Spring 集成方式来实现。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 11-9 所示。

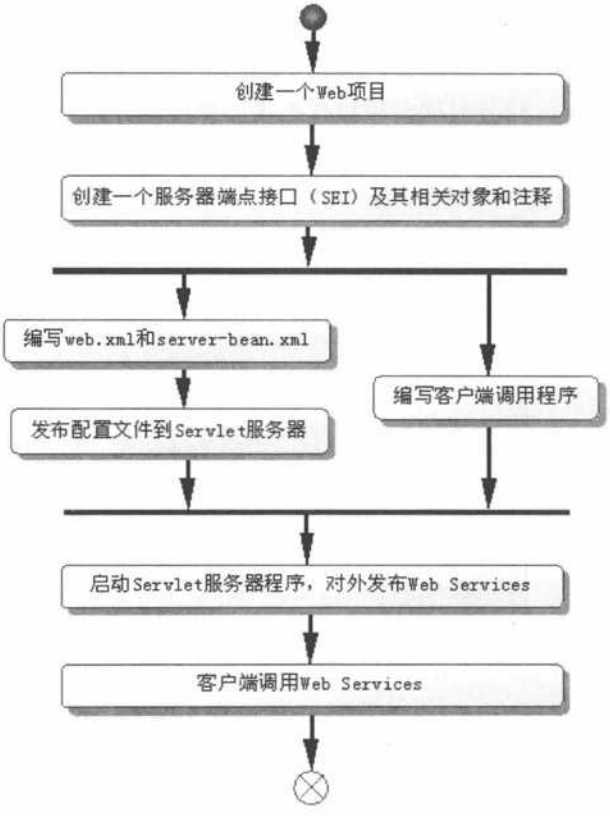


图 11-8　应用程序案例实现步骤流程图

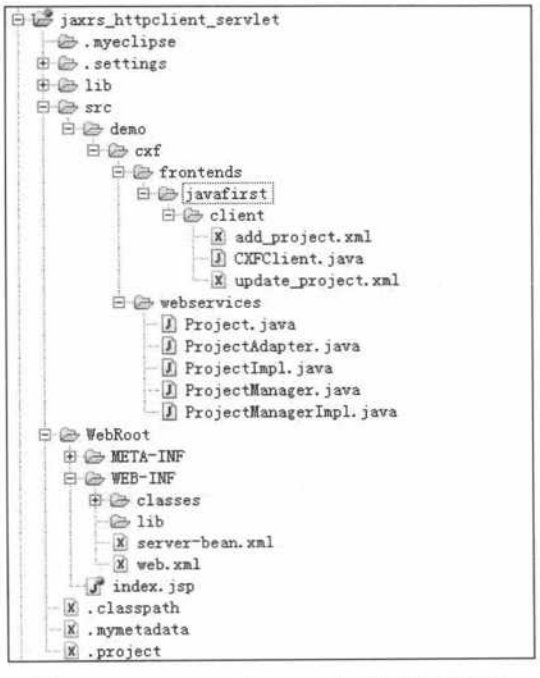


图 11-9　jaxrs_httpclient_servlet 程序目录结构

包目录包括三个部分，第一个包目录 WebRoot 目录是服务器程序目录，包括 web.xml 和 server-bean.xml 文件。第二个包目录 demo.cxf.frontends.javafirst.client 是客户端程序目录。第三个包目录 demo.cxf.webservices 是 SEI 和数据对象等公用部分，客户端和服务器端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口。由于本程序直接发布到 Servlet 容器，所以没有单独编写服务器发布程序，直接把配置文件和编译好的服务器端点接口（SEI）组件等部署到 Servlet 容器即可。

下面分别介绍两部分：①服务器端配置文件；②客户端调用程序。

（1）服务器端配置文件

服务器端配置文件包括 web.xml 文件和 server-bean.xml 文件。

web.xml 文件的定义与第 6 章“采用 POJO 并基于 Servlet 容器的前端实现”案例相同，这里就不再列出了。其中，Servlet 启动，将读取 WEB-INF 目录下的 server-bean.xml 文件，server-bean.xml 文件的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxrs="http://cxf.apache.org/jaxrs"
    xmlns:cxf="http://cxf.apache.org/core"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://cxf.apache.org/jaxrs
        http://cxf.apache.org/schemas/jaxrs.xsd
        http://cxf.apache.org/core
        http://cxf.apache.org/schemas/core.xsd">

    <import resource="classpath:META-INF/cxf/cxf.xml" />
    <import
        resource="classpath:META-INF/cxf/cxf-extension-jaxrs-binding.xml" />
    <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />

    <jaxrs:server id="ProjectManager"
        serviceClass="demo.cxf.webservices.ProjectManagerImpl" address="/">
    </jaxrs:server>

</beans>
```

把服务端程序发送到 Tomcat 中。

当 Servlet 容器启动，调用 web.xml 进行初始化，若有 listener 选项，调用其内容，为 org.springframework.web.context.ContextLoaderListener，关于 Spring 的配置在这里就不做详细说明了。然后就调用 server-bean.xml 文件，从 jaxrs:server 元素中了解到后台的 SEI 是 demo.cxf.webservices.ProjectManagerImpl，公开 Web Services 服务。

（2）客户端调用程序

采用 JAX-RS 规范的前端客户端，只需修改后台服务的位置即可，代码如下：

```
    ……
public class Client {
```

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

        System.out.println("Starting Client");



      private static void getGreetResponse()throws Exception {
        URL url = null;
InputStream in = null;
System.out.println("Sent HTTP GET request to greeting info.");
url = new URL("http://localhost:18080/jaxrs_httpclient_servlet/ProjectManager/greeting");
in = url.openStream();
System.out.println(getStringFromInputStream(in));

        System.out.println("Sent HTTP GET request to greetingWithParameter.");
url = new URL(
    "http://localhost:18080/jaxrs_httpclient_servlet/ProjectManager/
greetingWithParameter/greetisgood");
        in = url.openStream();
        System.out.println(getStringFromInputStream(in));
    }
        ……
    }
    ……
}
```

注意：笔者的 Tomcat 采用的是 18080 端口。

4. 测试和运行结果

测试编写好的程序代码，首先要启动 Tomcat 服务器程序。然后在 Eclipse 上运行客户端程序，其客户端程序和服务器端输出结果与“CXF 采用 HttpClient 实现基本的 RESTful 应用”案例运行的结果相同。

5. 其他内容

上述的所有调用都基于 JAX-RS 规范。

本例代码参见配套资源代码中“第 11 章\jaxrs_httpclient_servlet”文件夹。

11.5.3 CXF 采用 WebClient 实现 RESTful 应用

本程序演示 RESTful 的服务器端，同时客户端采用 WebClient 来进行调用，返回 Bean。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，都是采用 RESTful 的方式来进行处理。前端访问采用 WebClient 方式，后台服务器为遵循 JAX-RS 规范的 RESTful 来实现资源。WebClient 是 Apache CXF 框架的 JAX-RS 规范前端实现。实现的结构如图 11-10 所示。

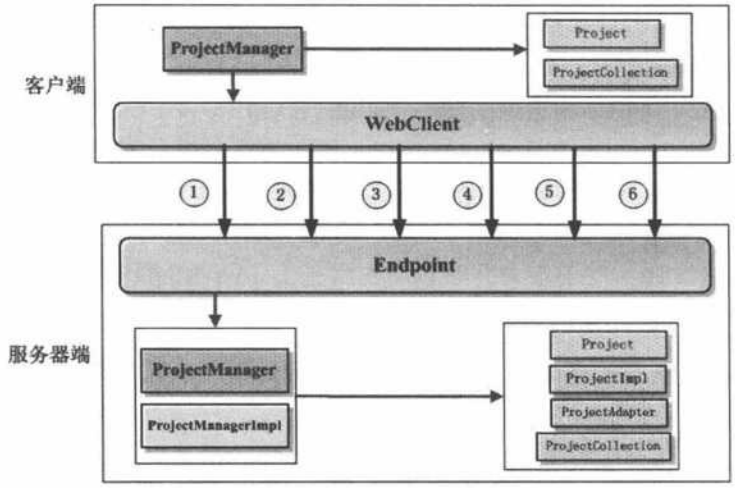


图 11-10　基于 WebClient 实现 RESTful 应用结构图

其中：① 表示客户端通过 WebClient 访问后台的基本情况；② 表示客户端通过 WebClient 访问后台的项目基本情况；③表示客户端通过 WebClient 修改默认项目；④表示客户端通过 WebClient 增加一个项目；⑤ 表示客户端通过 WebClient 删除一个项目；⑥表示客户端通过 WebClient 查看后台的全部项目情况。

2. 应用程序案例实现步骤

创建 WebClient 并基于 JAX-RS 规范的前端实现 Web Services 程序的步骤流程如图 11-11 所示。

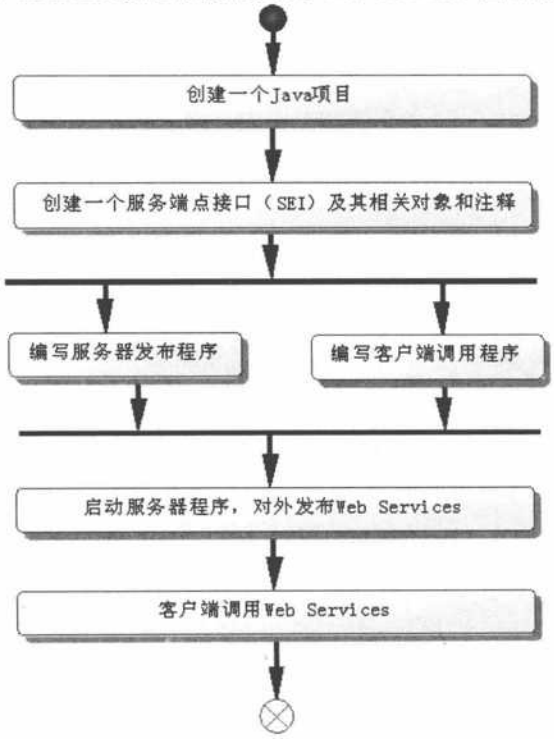


图 11-11　应用程序案例实现步骤流程图

流程图说明，要实现以下步骤：

（1）创建一个 Java 项目。

（2）创建一个基于 JAX-RS 服务的服务端点接口（SEI）及其相关对象和注释。

（3）编写发布这个 Java 程序为 Web Services 的服务端程序，支持 JAX-RS 服务。

（4）采用 WebClient 并基于 JAX-RS 服务的前端客户端应用程序。

（5）启动服务器程序，对外发布基于 JAX-RS 服务的 Web Services。

（6）基于 JAX-RS 服务的前端客户端调用后台基于 JAX-RS 服务的 Web Services。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 11-12 所示。

包目录包括三个部分，第一个包目录 demo.cxf.frontends.javafirst.server 是服务器程序目录。第二个包目录 demo.cxf.frontends.javafirst.client 是客户端程序目录。第三个包目录 demo.cxf.webservices 是服务器端点接口 SEI 和相关对象等公用部分，服务器端和客户端都有可能引用其内容。

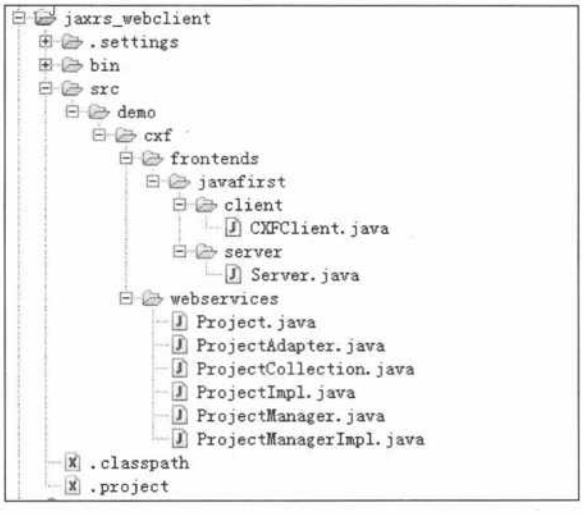


图 11-12　jaxrs_webclient 程序目录结构

服务器端点接口（SEI）是公用部分的接口，服务器端和客户端都必须引用。但服务器端和客户端对公用部分的引用是不同的。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口。而服务器端程序是接口和实现类都要引用。

下面分别介绍三部分：①服务器端点接口 SEI 和相关对象；②服务器端发布程序；③客户端调用程序。

（1）服务器端点接口（SEI）和相关对象

首先要介绍的是服务器端点接口（SEI）。Project 组件包括 WebService 包下的 Project 几个文件。

ProjectManager 的程序代码如下：

```
package demo.cxf.webservices;
```

```
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;

@Path("/ProjectManager/")
//@Produces("application/xml")
public interface ProjectManager {

    @GET
    @Path("/greeting/")
    public String greeting() ;

    @GET
    @Path("/greetingWithParameter/{greet}/")
    public String greetingWithParameter(@PathParam("greet") String greet);

    @GET
    @Path("/getProjectName/")
    public  String getProjectName();

    @GET
    @Path("/getProjectNameFromID/{projectID}/")
    public String getProjectNameFromID(@PathParam("projectID") String  projectID);


    @GET
    @Path("/getDefaultProject/")
    //@Produces("application/xml")
    public Project getDefaultProject();

    @GET
    @Path("/getProjectFromID/{projectID}/")
    //@Produces("application/xml")
    public  Project  getProjectFromID(@PathParam("projectID")  String   projectID);


    @GET
    @Path("/updateProjectName/{projectName}/")
    public void updateProjectName (@PathParam("projectName") String projectName);


    @PUT
    @Path("/setDefaultProject/")
    public void setDefaultProject( Project project);

    @POST
    @Path("/registerProject/")
    public void registerProject(Project project);

    @DELETE
    @Path("/removeProject/{id}/")
    //public void removeProject(@PathParam("id") Project project);
    public void removeProject(@PathParam("id") String id);

    @GET
    @Path("/getProjectCollection/")
```

```
    public ProjectCollection getProjectCollection();

    @GET
    @Path("/getAllProject/")
    public String getAllProject() ;

}
```

ProjectManager 接口内容基本与 HttpClient 实现基本的 RESTful 应用是一样的。但是对于返回值，不是采用的 javax.ws.rs.core.Response。还有一个实现类 ProjectManagerImpl 类。代码基本与 HttpClient 实现一样，这里省略。

在这里增加了一个 ProjectCollection 的 Bean，主要是针对前端有返回整个项目的需求。

ProjectCollection 类代码如下：

```
package demo.cxf.webservices;

import java.util.Collection;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "ProjectCollection")
public class ProjectCollection {

    private String projectMapId;
    private String projectMapName;
    private Collection<Project> projects;

    public String getProjectMapId() {return projectMapId; }
    public void setProjectMapId(String projectMapId) {this.projectMapId = projectMapId;}
    public String getProjectMapName() {return projectMapName;}
    public void setProjectMapName(String projectMapName) {this.projectMapName =
projectMapName;}
    public Collection<Project> getProjects() {return projects;}
    public void setProjects(Collection<Project> projects) {this.projects = projects;}
}
```

实际上 ProjectCollection 就是一个项目集的概念，可以获得项目组中的全部或多个项目。

（2）服务器端发布程序

服务器端发布程序就是 Server.java 文件。其作用就是启动服务器。服务器端代码如下：

```
package demo.cxf.frontends.javafirst.server;

import org.apache.cxf.jaxrs.JAXRSServerFactoryBean;
import org.apache.cxf.jaxrs.lifecycle.SingletonResourceProvider;
import demo.cxf.webservices.ProjectManager;
import demo.cxf.webservices.ProjectManagerImpl;

public class Server {

    public Server() {
        System.out.println("Starting Server");

        ProjectManager projectManager = new ProjectManagerImpl();
        JAXRSServerFactoryBean sf = new JAXRSServerFactoryBean();
       sf.setResourceClasses(ProjectManager.class);
       sf.setResourceProvider(ProjectManager.class, new SingletonResourceProvider
(projectManager));
```

```
        sf.setAddress("http://localhost:9000/");
        sf.create();
    }

    public static void main(String[] args) throws Exception {
        new Server();
        System.out.println("Server ready...");
        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

ProjectManager 接口内容基本与 HttpClient 实现基本的 RESTful 应用的内容是一样的。这里省略。

（3）客户端调用程序

客户端的核心实现部分主要由六部分组成：使用 HTTP PUT 增加、修改用户资源；使用 HTTP GET 得到某一具体用户资源；使用 HTTP DELETE 删除用户资源；使用 HTTP GET 得到用户列表资源。

客户端程序文件只有 CXFClient.java， CXFClient.java 是实现客户端调用的主程序。

客户端实现查询服务器的基本情况的代码如下：

```
//不带参数获取后台信息
    private static void greeting(){
        // Service instance
        System.out.println("String details from REST service.");
        WebClient client = WebClient.create("http://localhost:9000/");
        System.out.println(client.path("ProjectManager/greeting/").accept
("application/xml").get(String.class));
    }

    //带参数获取后台信息
    private static void greetingWithParameter(){
        // Service instance
        System.out.println("String details from REST service.");
        WebClient client = WebClient.create("http://localhost:9000/");
        System.out.println(client.path("ProjectManager/greetingWithParameter/
greetisgood").
accept("application/xml").get(String.class));
    }
```

客户端实现查询服务器的项目基本情况的代码如下：

```
// 不带参数获取后台的默认项目信息
    private static void getProjectName() {
        // Service instance
        System.out.println("String details from REST service.");
        WebClient client = WebClient.create("http://localhost:9000/");
        System.out.println(client.path("ProjectManager/getProjectName/")
                .accept("application/xml").get(String.class));
    }

    // 不带参数获取后台的默认项目 Bean
    private static void getDefaultProject() {
```

```
        System.out.println("\n");
        System.out.println("get Default project details from REST service.");
        WebClient getProjectClient = WebClient.create("http://localhost:9000/");
        Project project = getProjectClient.path(
                "ProjectManager/getDefaultProject").accept("application/xml")
                .get(Project.class);
        System.out.print("Project Id " + project.getProjectID() + " ; ");
        System.out.println("Project Name " + project.getProjectName());
    }

    // 带参数获取后台的默认项目 Bean
    private static void getProjectFromID() {
        System.out.println("\n");
        System.out.println("get Project details From ID through REST service.");
        WebClient getProjectClient = WebClient.create("http://localhost:9000/");
        Project project = getProjectClient.path(
                "ProjectManager/getProjectFromID/2").accept("application/xml")
                .get(Project.class);
        System.out.print("Project Id " + project.getProjectID() + " ; ");
        System.out.println("Project Name " + project.getProjectName());

    }
```

客户端实现修改默认项目代码如下：

```
// 修改后台的默认项目 Bean
    private static void updateProject() {
        System.out.println("\n");
        System.out.println("update Project from REST service.");
        Project thisProject = new ProjectImpl();
        thisProject.setProjectID("projectID1");
        thisProject.setProjectName("projectName1");
        thisProject.setProjectDescribe("projectDescribe1");

        WebClient updateProjectClient = WebClient.create("http://localhost:9000/");
        updateProjectClient.path("ProjectManager/setDefaultProject").accept(
                "application/xml").put(thisProject);
    }
```

在上述代码中，其项目是创建一个实体 Bean，通过 Bean 来修改后台的项目库。

客户端实现增加一个项目的代码如下：

```
// 增加一个项目到后台的项目组
    private static void addProject() {
        System.out.println("\n");
        System.out.println("add project from REST service.");
        Project thisProject = new ProjectImpl();
        thisProject.setProjectID("projectID10");
        thisProject.setProjectName("projectName10");
        thisProject.setProjectDescribe("projectDescribe10");

        WebClient addProjectClient = WebClient.create("http://localhost:9000/");
        addProjectClient.path("ProjectManager/registerProject").accept(
                "application/xml").post(thisProject);
    }
```

在上述代码中，其项目是通过创建一个实体 Bean，通过 Bean 来增加后台的项目库。

客户端实现删除一个项目的代码如下：

```
/// 从后台的项目组中删除一个项目
    private static void deleteProject() {
        System.out.println("\n");
        System.out.println("delete project from REST service.");
        WebClient deleteProjectClient = WebClient
                .create("http://localhost:9000/");
        deleteProjectClient.path("ProjectManager/removeProject/projectID10")
                .accept("application/xml").delete();
    }
```

在删除过程中，直接说明项目的 ID，然后删除后台的项目库。

客户端实现查看后台的全部项目情况的代码如下：

```
// 获取后台的项目组的全部项目并展示
    private static void getProjectCollection() {
        Iterator<Project> iterator = null;
        ProjectCollection projectCollection = null;
        // System.out.println("\n");
        System.out.println("get projectCollection from REST service.");
        WebClient projectCollectionClient = WebClient
                .create("http://localhost:9000/");
        projectCollection = projectCollectionClient.path(
                "ProjectManager/getProjectCollection")
                .accept("application/xml").get(ProjectCollection.class);

        iterator = projectCollection.getProjects().iterator();
        while (iterator.hasNext()) {
            Project project1 = iterator.next();
            System.out.print("ProjectID is " + project1.getProjectID() + ";");
            System.out.println("Project name is " + project1.getProjectName());
        }
    }
```

在删除过程中，直接说明项目的 ID，通过执行删除代码，就能删除后台的项目库中项目 ID 的内容。

使用标准的 HTTP 状态作为状态码。成功的操作都会返回 200。如果系统无法根据资源的 ID 找到它，就会返回 404。任何由系统故障导致的错误都会返回 500。

4. 测试和运行结果

测试编写好的程序代码，首先要启动服务器程序，即在 Eclipse 上运行 Server.java，然后运行客户端程序，客户端程序输出结果如下：

```
Hello, This is ProjectManager.
String details from REST service.
Hello, greetisgood This is ProjectManager.
String details from REST service.
projectName1

get Default project details from REST service.
Project Id projectID1 ; Project Name projectName1
```

```
get Project details From ID through REST service.
Project Id projectID2 ; Project Name projectName2

update Project from REST service.

get Default project details from REST service.
Project Id projectID1 ; Project Name projectName1
get projectCollection from REST service.
ProjectID is projectID2;Project name is projectName2
ProjectID is default projectID;Project name is default projectName

add project from REST service.
get projectCollection from REST service.
ProjectID is projectID10;Project name is projectName10
ProjectID is projectID2;Project name is projectName2
ProjectID is default projectID;Project name is default projectName

delete project from REST service.
get projectCollection from REST service.
ProjectID is projectID2;Project name is projectName2
ProjectID is default projectID;Project name is default projectName
```

5. 其他内容

上述的所有调用基于 JAX-RS 规范。

本例代码参见配套资源代码中“第 11 章\jaxrs_webclient”文件夹。

11.5.4 CXF 采用 WebClient 在 Servlet 实现基本的 RESTful 应用

本程序演示 Apache CXF 框架采用 WebClient 来实现客户端和 Servlet 服务器端的 RESTful 调用，客户端采用 Apache CXF 的代码优先（Java First）前端应用模式。服务器端采用 Servlet 容器。本例 Servlet 容器采用 Tomcat 6。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询和删除操作。

在这个过程中，前端客户端采用 Apache CXF 自身规范的 WebClient 对象来实现，后台服务端为遵循 JAX-RS 规范的 Servlet 容器来实现。实现的结构如图 11-13 所示。

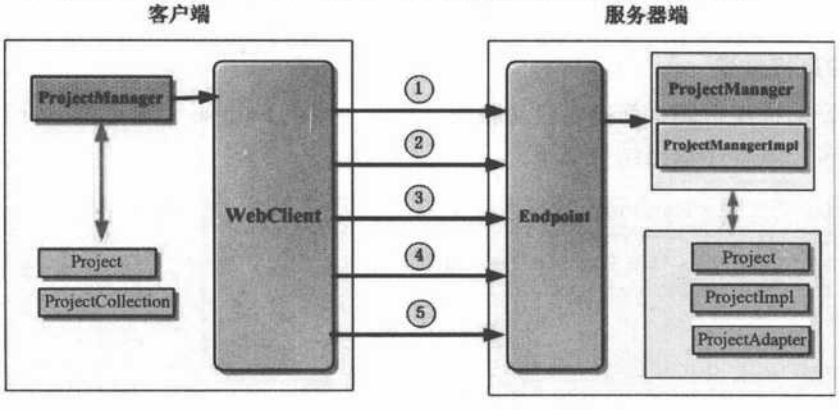


图 11-13　基于 WebClient 实现 Servlet 的 RESTful 应用结构图

其中：① 表示客户端通过 WebClient 访问后台的基本情况；② 表示客户端通过 WebClient 访问后台的项目基本情况；③表示客户端通过 WebClient 修改默认项目；④表示客户端通过 WebClient 增加一个项目；⑤ 表示客户端通过 WebClient 删除一个项目。

2. 应用程序案例实现步骤

要基于 JAX-RS 服务创建 WebClient 并在 Servlet 容器的前端实现 Web Services 程序的步骤流程如图 11-14 所示。

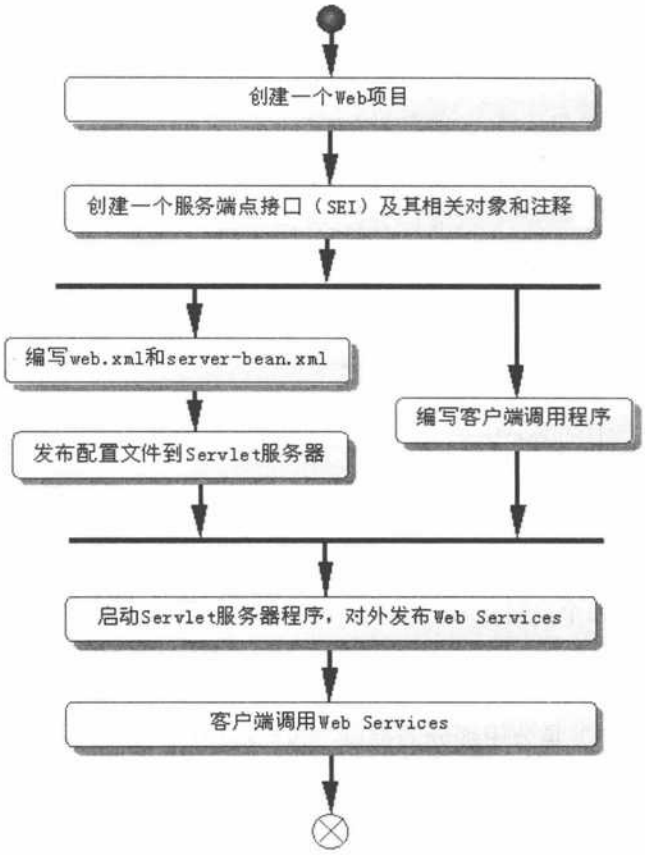


图 11-14　应用程序案例实现步骤流程图

流程图说明，要实现以下步骤：

（1）创建一个 Web 服务项目。

（2）创建一个基于 JAX-RS 服务的服务端点接口（SEI）及其相关对象和注释。

（3）采用 WebClient 方式，编写基于 JAX-RS 服务的 Web Services 的客户端程序。

（4）编写基于 JAX-RS 服务的 web.xml 和 server-bean.xml。

（5）发布配置文件到 Servlet 服务器。

（6）启动 Servlet 服务器程序，对外发布基于 JAX-RS 服务的 Web Services。

（7）基于 HttpClient 方式的客户端调用后台基于 JAX-RS 服务的 Web Services。

其中第 1 步和第 3 步与前面采用自带的 HTTP 服务器类似。主要是第 2 步，该步骤不是采用自带的服务，而是放置到 Servlet 容器中。我们采用与 Spring 集成方式来实现。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 11-15 所示。

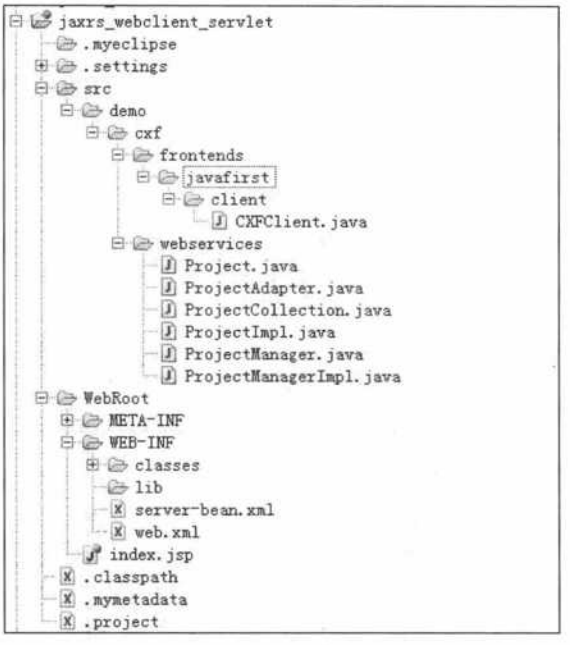


图 11-15　jaxrs_webclient_servlet 程序目录结构

包目录包括三个部分，第一个包目录 WebRoot 是服务器程序目录，包括 web.xml 和 server-bean.xml 文件。第二个包目录 demo.cxf.frontends.javafirst.client 是客户端程序目录。第三个包目录 demo.cxf.webservices 是 SEI 和数据对象等公用部分，客户端和服务器端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口。由于本程序直接发布到 Servlet 容器，所以没有单独编写服务器发布程序，直接把配置文件和编译好的服务器端点接口（SEI）组件等部署到 Servlet 容器即可。

下面分别介绍两部分：①服务器端配置文件；②客户端调用程序。

（1）服务器端配置文件

服务器端配置文件包括 web.xml 文件和 server-bean.xml 文件。

web.xml 文件的定义与第 6 章“采用 POJO 并基于 Servlet 容器的前端实现”案例相同，这里就不再列出了。其中，当 Servlet 启动，将读取 WEB-INF 目录下的 server-bean.xml 文件。server-bean.xml 文件的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxrs="http://cxf.apache.org/jaxrs"
    xmlns:cxf="http://cxf.apache.org/core"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
```

```
        http://cxf.apache.org/jaxrs
        http://cxf.apache.org/schemas/jaxrs.xsd
        http://cxf.apache.org/core
        http://cxf.apache.org/schemas/core.xsd">

    <import resource="classpath:META-INF/cxf/cxf.xml" />
    <import
        resource="classpath:META-INF/cxf/cxf-extension-jaxrs-binding.xml" />
    <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />

    <jaxrs:server id="ProjectManager"
        serviceClass="demo.cxf.webservices.ProjectManagerImpl" address="/">
    </jaxrs:server>
</beans>
```

把服务端程序发送到 Tomcat 中。这里采用了 jaxrs 节点。

当 Servlet 容器启动，调用 web.xml 进行初始化，若有 listener 选项，调用其内容，为 org.springframework.web.context.ContextLoaderListener。然后就调用 server-bean.xml 文件，从 jaxrs:server 元素中了解到后台的 SEI 是 demo.cxf.webservices.ProjectManagerImpl，公开 Web Services 服务。

（2）客户端调用程序

采用 JAX-RS 规范的前端客户端，只需修改后台服务的位置即可，代码如下：

```
      ……
public class CXFClient {

private static String webAddress = "http://localhost:18080/jaxrs_webclient_servlet/";

      public CXFClient() { }
……
// 不带参数获取后台信息
private static void greeting() {
        // Service instance
        System.out.println("String details from REST service.");
        WebClient client = WebClient.create(webAddress);
        System.out.println(client.path("ProjectManager/greeting/").accept(
                "application/xml").get(String.class));
    }
……
}
```

注意：笔者的 Tomcat 采用的是 18080 端口。

4. 测试和运行结果

测试编写好的程序代码，首先要启动 Tomcat 服务器程序。然后在 Eclipse 上运行客户端程序，其客户端程序和服务器端输出结果与前面的相同。

5. 其他内容

上述的所有调用都基于 JAX-RS 规范。

本例代码参见配套资源代码中“第 11 章\jaxrs_webclient_servlet”文件夹。

11.5.5 JAX-WS Provider 和 Dispatch 实现 RESTful 方式

本程序演示 Apache CXF 框架采用 Provider/Dispatch 前端模式实现客户端和 RESTful 服务器端调用，客户端采用 Dispatch 的 Service.Mode.PAYLOAD 前端应用模式。后台采用 Provider 方式，主要处理大型的、复杂的 XML 文件。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个存储信息的 XML 文件，然后通过客户端访问服务器，获得后台的 XML 文件的数据。在这个过程中，都使用 XML 文件实现客户端和服务器端的数据交换。前台客户端采用 JAX-WS 规范的 Service 对象，后台服务器端为遵循 JAX-WS 规范的 Endpoint 对象来实现。实现的结构如图 11-16 所示。

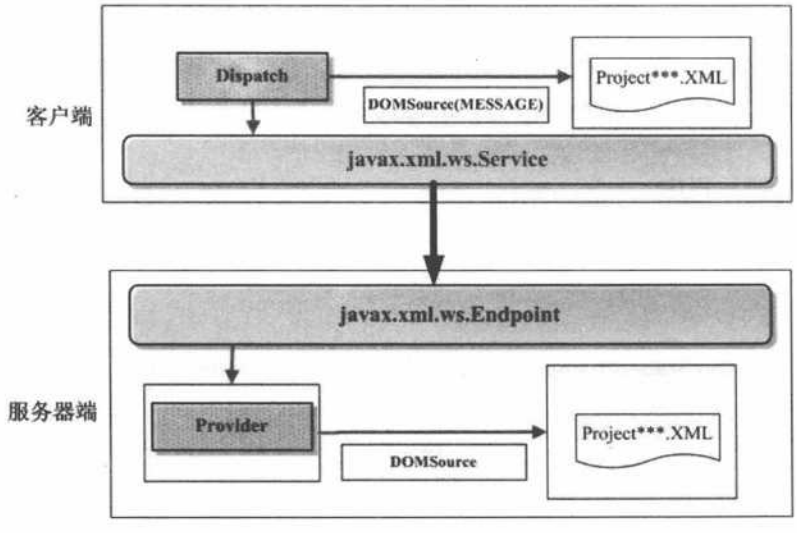


图 11-16　采用 Provider 和 Dispatch 实现 RESTful 应用结构图

图中粗箭头表示客户端 Service 通过 Endpoint 获得 Provider，访问后台的 XML 文件，然后返回给前端的 Dispatch 进行处理。

2. 应用程序案例实现步骤

要创建 RESTful 方式的 Provider/Dispatch 前端模式实现 Web Services 程序的步骤流程如图 11-17 所示。

流程图说明，要实现以下步骤：

（1）创建一个 Java 项目。

（2）创建服务器和客户端对应的 XML 文件。

（3）编写发布这个 Java 程序为 Web Services 的服务器端程序。

（4）编写 Web Services 的客户端应用程序。

（5）启动服务器程序，对外发布 Web Services。

（6）客户端调用 Web Services。

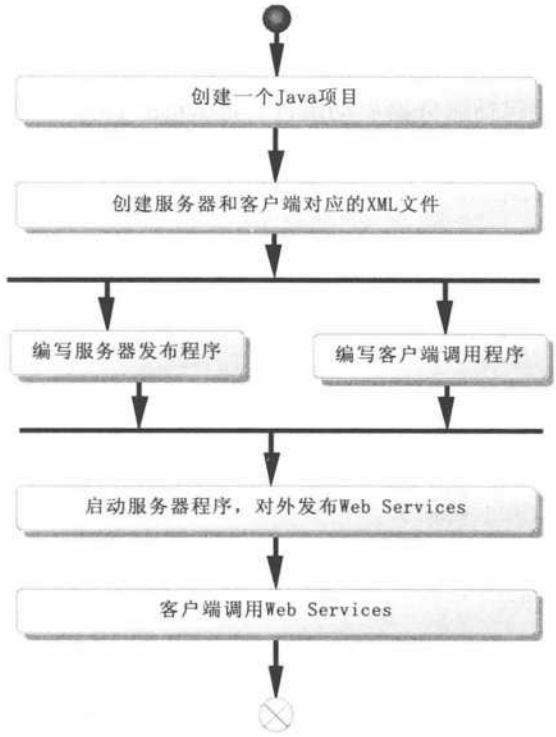


图 11-17　应用程序案例实现步骤流程图

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 11-18 所示。

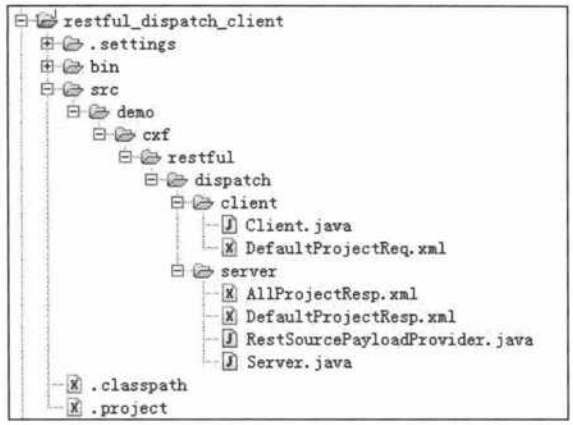


图 11-18　restful_dispatch_client 程序结构

包目录包括两个部分，第一个包目录 demo.cxf.restful.dispatch.server 是服务器程序目录。第二个包目录 demo.cxf.restful.dispatch.client 是客户端程序目录。注意，在这里并没有服务器端点接口 SEI。

本案例程序只包括服务器端应用程序和客户端应用程序两个部分。下面分别介绍两部分：

①服务器端发布程序；②客户端调用程序。

（1）服务器端发布程序

服务器端应用程序主要包括服务器启动组件、Provider<DOMSource>组件和要访问的 XML 文件。

服务器启动组件为 Server.java 文件，代码如下：

```
package demo.cxf.restful.dispatch.server;
import javax.xml.ws.Endpoint;
import javax.xml.ws.http.HTTPBinding;

public class Server {
   protected Server() throws Exception {
      System.out.println("Starting Server");
      Endpoint endpoint = Endpoint.create(HTTPBinding.HTTP_BINDING, new
RestSourcePayloadProvider());
      String address = "http://localhost:9000/ProjectManager/project";
      endpoint.publish(address);
   }

   public static void main(String args[]) throws Exception {
      new Server();
      System.out.println("Server ready...");

      Thread.sleep(5 * 60 * 1000);
      System.out.println("Server exiting");
      System.exit(0);
   }
}
```

启动服务器，实例化 Provider<DOMSource>对象，然后通过遵循 JAX-WS 规范的 Endpoint 发布 Web Services。

Provider<DOMSource>组件只有一个文件——RestSourcePayloadProvider.java，代码如下：

```
package demo.cxf.restful.dispatch.server;

import java.io.InputStream;

import javax.annotation.Resource;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.ws.Provider;
import javax.xml.ws.Service;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.WebServiceProvider;
import javax.xml.ws.handler.MessageContext;

import org.w3c.dom.Document;

import org.apache.cxf.message.Message;

@WebServiceProvider()
@ServiceMode(value = Service.Mode.PAYLOAD)
public class RestSourcePayloadProvider implements Provider<DOMSource> {
```

```
@Resource
protected WebServiceContext wsContext;

public RestSourcePayloadProvider() {
}

public DOMSource invoke(DOMSource request) {
    MessageContext mc = wsContext.getMessageContext();
    String path = (String)mc.get(Message.PATH_INFO);
    String query = (String)mc.get(Message.QUERY_STRING);
    String httpMethod = (String)mc.get(Message.HTTP_REQUEST_METHOD);

    System.out.println("--path--- " + path);
    System.out.println("--query--- " + query);
    System.out.println("--httpMethod--- " + httpMethod);

    if (httpMethod.equalsIgnoreCase("POST")) {
        // TBD: parse query info from DOMSource
        System.out.println("---Invoking updateProject---");
        return updateProject(request);
    } else if (httpMethod.equalsIgnoreCase("GET")) {
        if (path.equals("/ProjectManager/project") && query == null) {
            System.out.println("---Invoking getAllProject---");
            return getAllProjects();
        } else if (path.equals("/ProjectManager/project") && query != null) {
            System.out.println("---Invoking getProject---");
            return getProject(query);
        }
    }
    return null;
}

private DOMSource getAllProjects() {
    return createDOMSource("AllProjectResp.xml");
}

private DOMSource getProject(String ProjectID) {
    return createDOMSource("DefaultProjectResp.xml");
}

private DOMSource updateProject(DOMSource request) {
    // TBD: returned update customer info
    return createDOMSource("DefaultProjectResp.xml");
}

private DOMSource createDOMSource(String fileName) {
    DocumentBuilderFactory factory;
    DocumentBuilder builder;
    Document document = null;
    DOMSource response = null;

    try {
        factory = DocumentBuilderFactory.newInstance();
        //factory.setValidating(true);
        builder = factory.newDocumentBuilder();
        InputStream greetMeResponse = getClass().getResourceAsStream(fileName);

        document = builder.parse(greetMeResponse);
```

```
            response = new DOMSource(document);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return response;
    }
}
```

其服务器上的 XML 文件有 DefaultProjectResp.xml 和 AllProjectResp.xml，其中 DefaultProjectResp.xml 内容如下：

```
<?xml version="1.0" encoding="utf-8" ?>
<Project>
    <name>default project name</name>
    <id>default ID</id>
</Project>
```

AllProjectResp.xml 内容如下：

```
<?xml version="1.0" encoding="utf-8" ?>
<Projects>
    <Project href="http://localhost/projectservice/project?id=1234">
        <id>1234</id>
    </Project>
    <Project href="http://localhost/projectservice/project?id=1235">
        <id>1235</id>
    </Project>
    <Project href="http://localhost/projectservice/project?id=1236">
        <id>1236</id>
    </Project>
</Projects>
```

（2）客户端调用程序

客户端应用程序主要有 DispatchClient 组件和一个 XML 文件。

DispatchClient 组件只有一个文件——Client.java，代码如下：

```
package demo.cxf.restful.dispatch.client;

import java.io.ByteArrayOutputStream;
import java.io.InputStream;
import java.net.URI;
import java.net.URL;
import java.util.Map;
import java.util.Properties;

import javax.xml.namespace.QName;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;
import javax.xml.ws.Dispatch;
import javax.xml.ws.Service;
import javax.xml.ws.handler.MessageContext;
```

```
import javax.xml.ws.http.HTTPBinding;

import org.w3c.dom.Document;
import org.apache.cxf.helpers.XMLUtils;

public final class Client {

    private Client() {
    }

    public static void main(String args[]) throws Exception {
        QName serviceName = new QName("http://apache.org/wrapped","projectservice");
        QName portName = new QName("http://apache.org/wrapped","RestProviderPort");
        String endpointAddress = "http://localhost:9000/ProjectManager/project";

        // Sent HTTP GET request to query all customer info
        URL url = new URL(endpointAddress);
        System.out.println("Invoking server through HTTP GET to query all customer info");
        InputStream in = url.openStream();
        StreamSource source = new StreamSource(in);
        printSource(source);

        // Sent HTTP GET request to query customer info
        url = new URL(endpointAddress + "?id=1234");
        System.out.println("Invoking server through HTTP GET to query customer info");
        in = url.openStream();
        source = new StreamSource(in);
        printSource(source);

        // Sent HTTP POST request to update customer info using JAX-WS Dispatch
        URI endpointURI = new URI(endpointAddress.toString());
        String path = null;
        if (endpointURI != null) {
            path = endpointURI.getPath();
        }

        Service service = Service.create(serviceName);
        service.addPort(portName, HTTPBinding.HTTP_BINDING, endpointAddress);
        Dispatch<DOMSource> dispatcher = service.createDispatch(portName,
                DOMSource.class, Service.Mode.PAYLOAD);
        Map<String, Object> requestContext = dispatcher.getRequestContext();

        Client client = new Client();
        InputStream is = client.getClass().getResourceAsStream(
                "DefaultProjectReq.xml");
        Document doc = XMLUtils.parse(is);
        DOMSource reqMsg = new DOMSource(doc);

        requestContext.put(MessageContext.HTTP_REQUEST_METHOD, "POST");
        System.out.println("Invoking through HTTP POST to update customer using JAX-WS
Dispatch");
        DOMSource result = dispatcher.invoke(reqMsg);
        printSource(result);

        System.out.println("Client Invoking succeeded!");
        System.exit(0);
    }

    private static void printSource(Source source) {
```

```
        try {
            ByteArrayOutputStream bos = new ByteArrayOutputStream();
            StreamResult sr = new StreamResult(bos);
            Transformer trans = TransformerFactory.newInstance()
                    .newTransformer();
            Properties oprops = new Properties();
            oprops.put(OutputKeys.OMIT_XML_DECLARATION, "yes");
            trans.setOutputProperties(oprops);
            trans.transform(source, sr);
            System.out.println("**** Response ******");
            System.out.println(bos.toString());
            bos.close();
            System.out.println();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

还有一个 XML 文件，主要是实现与后台服务器中的 XML 文件格式匹配，代码如下：

```
<?xml version="1.0" encoding="utf-8" ?>
<tns:Project
    xmlns:tns="http://apache.org/hello_world_soap_http/types">
    <tns:id>123456</tns:id>
</tns:Project>
```

4. 测试和运行结果

测试编写好的程序代码，首先要启动服务器程序，即在 Eclipse 上运行 Server.java。然后运行客户端程序，客户端程序输出结果如下：

```
Invoking server through HTTP GET to query all customer info
**** Response ******
<Projects>
    <Project href="http://localhost/projectservice/project?id=1234">
        <id>1234</id>
    </Project>
    <Project href="http://localhost/projectservice/project?id=1235">
        <id>1235</id>
    </Project>
    <Project href="http://localhost/projectservice/project?id=1236">
        <id>1236</id>
    </Project>
</Projects>

Invoking server through HTTP GET to query customer info
**** Response ******
<Project>
    <name>default project name</name>
    <id>default ID</id>
</Project>

Invoking through HTTP POST to update customer using JAX-WS Dispatch
**** Response ******
<Project>
    <name>default project name</name>
    <id>default ID</id>
</Project>
```

```
Client Invoking succeeded!
```

其服务器端输出结果如下：

```
--path--- /ProjectManager/project
--query--- null
--httpMethod--- GET
---Invoking getAllProject---
--path--- /ProjectManager/project
--query--- id=1234
--httpMethod--- GET
---Invoking getProject---
--path--- /ProjectManager/project
--query--- null
--httpMethod--- POST
---Invoking updateProject---
```

5. 其他内容

上述的所有调用都基于 JAX-WS 规范。注意，这里是 JAX-WS 规范。

本例代码参见配套资源代码中“第 11 章\restful_dispatch_client”文件夹。

11.5.6 Http_Binding 实现基于 XML 的 RESTful 方式

本程序演示 Apache CXF 框架采用 Http_Binding 前端模式实现基于 XML 的客户端和 RESTful 服务器端调用。有两种实现方式，一种是采用 XML 来处理，还有一种是采用 JSON 来实现。本例采用 XML 来进行处理。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询。在这个过程中，都使用 HTTP 实现客户端和服务器端的数据交换，采用 XML 数据格式进行传递。前台客户端采用标准的 java.net.URL 对象，后台服务端为遵循 Apache CXF 框架的 JAX-WS 规范的 JaxWsServerFactoryBean 对象来实现。实现的结构如图 11-19 所示。

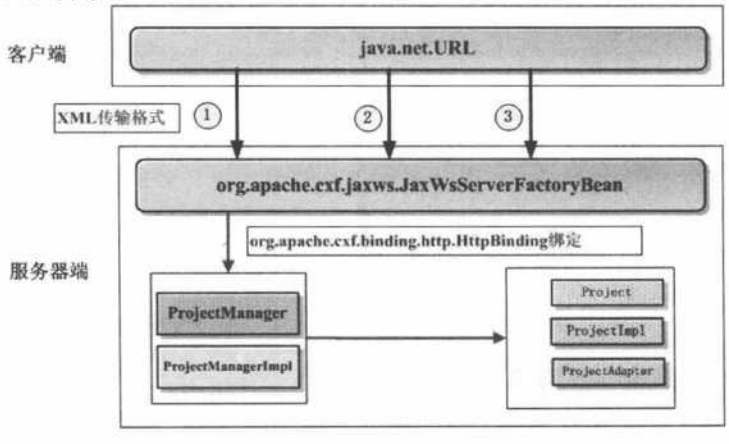


图 11-19　基于 XML 的 Http_Binding 实现 RESTful 应用结构图

其中：①表示客户端通过 URL 访问后台的基本情况；②表示客户端通过 URL 访问后台的项目基本情况；③表示客户端通过 URL 修改默认项目。

2. 应用程序案例实现步骤

要创建 Http_Binding 实现基于 XML 的 RESTful 方式的前端实现 Web Services 程序的流程步骤如图 11-20 所示。

流程图说明，要实现以下步骤：

（1）创建一个 Java 项目。

（2）创建一个基于 JAX-WS 服务的服务端点接口（SEI）及其相关对象和注释。

（3）编写发布这个 Java 程序为 Web Services 的服务端程序，支持 JAX-RS 服务。

（4）采用 HttpBinding 并基于 JAX-RS 服务的前端客户端应用程序，调用方式为 XML。

（5）启动服务器程序，对外发布基于 JAX-RS 服务的 Web Services。

（6）基于 JAX-RS 服务的前端客户端调用后台基于 JAX-RS 服务的 Web Services。

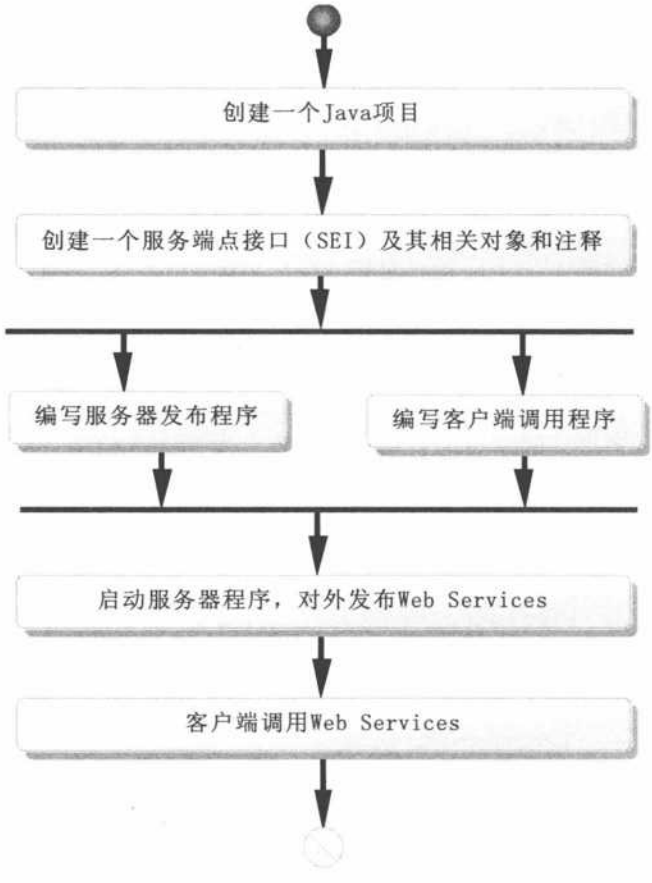


图 11-20　应用程序案例实现步骤流程图

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 11-21 所示。

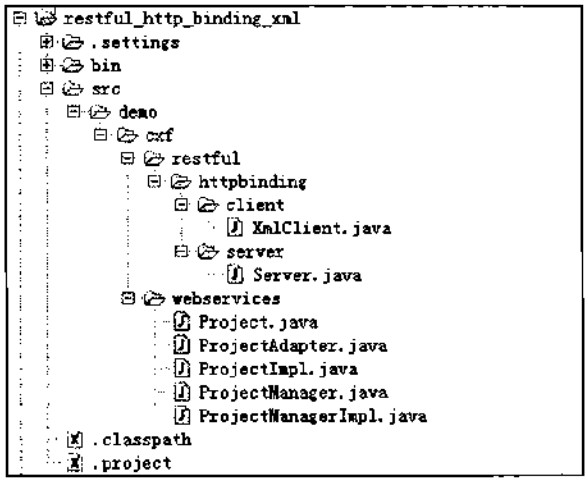


图 11-21　restful_http_binding_xml 程序结构

包目录包括三个部分，第一个包目录 demo.cxf.restful.httpbinding.server 是服务器程序目录。第二个包目录 demo.cxf.restful.httpbinding.client 是客户端程序目录。第三个包目录 demo.cxf.webservices 是服务器端点接口 SEI 和相关对象等公用部分，服务器端和客户端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口，服务器端和客户端都必须引用。但服务器端和客户端对公用部分的引用是不同的。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口。而服务器端程序是接口和实现类都要引用。

下面分别介绍三部分：①服务器端点接口 SEI 和相关对象；②服务器端发布程序；③客户端调用程序。

（1）服务器端点接口（SEI）和相关对象

首先要介绍的是服务器端点接口（SEI）。Project 组件包括 WebService 包下的 Project 几个文件。

ProjectManager 接口的代码如下：

```
package demo.cxf.webservices;

import javax.jws.WebService;
import javax.jws.WebParam;
import javax.jws.WebResult;

import org.codehaus.jra.Delete;
import org.codehaus.jra.Get;
import org.codehaus.jra.HttpResource;
import org.codehaus.jra.Post;
import org.codehaus.jra.Put;

@WebService(targetNamespace = "http://webservices.cxf.demo/", name = "ProjectManager")
public interface ProjectManager {

    @Get
    @HttpResource(location = "/getProjectName")
    @WebResult(name = "String")
```

```
    public String getProjectName();

    @Get
    @HttpResource(location = "/greeting")
    @WebResult(name = "String")
    public String greeting() ;

    @Post
    @HttpResource(location = "/greetingWithParameter/{greet}")
    //@WebResult(name = "String")
    public String greetingWithParameter(@WebParam(name = "greet") String greet);

    @Put
    @HttpResource(location = "/updateProjectName/{projectName}")
    public void updateProjectName (@WebParam(name = "projectName")String
projectName);

    @Put
    @HttpResource(location = "/setDefaultProject/{id}")
    public void setDefaultProject(@WebParam(name = "Project") Project project);

    @Post
    @HttpResource(location = "/registerProject/{id}")
    public void registerProject(@WebParam(name = "Project")Project project);

    @Get
    @HttpResource(location = "/getProjectNameByID/{projectID}")
    @WebResult(name = "String")
    public String getProjectNameByID(@WebParam(name = "projectID") String projectID);

    @Delete
    @HttpResource(location ="/removeProject/{id}/")
    public void removeProject(@WebParam(name ="id") String id);

    @Get
    @HttpResource(location = "/getDefaultProject")
    @WebResult(name = "Project")
    public Project getDefaultProject();
}
```

（2）服务器端发布程序

服务器端发布程序就是 Server.java 文件，作用是启动服务器。服务器端代码如下：

```
package demo.cxf.restful.httpbinding.server;

import org.apache.cxf.binding.http.HttpBindingFactory;
import org.apache.cxf.jaxws.JaxWsServerFactoryBean;
import org.apache.cxf.service.invoker.BeanInvoker;

import demo.cxf.webservices.ProjectManager;
import demo.cxf.webservices.ProjectManagerImpl;

public final class Server {
    private static String httpAddress = "http://localhost:9000/ProjectManager/";

    private Server() { }

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

```
    ProjectManager projectManager = new ProjectManagerImpl();
        createRestService(projectManager);
        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }

    private static void createRestService(Object serviceObj) {
    String address = httpAddress+"xml/";
        // Build up the server factory bean
        JaxWsServerFactoryBean sf = new JaxWsServerFactoryBean();
        sf.setServiceClass(ProjectManager.class);
        // Use the HTTP Binding which understands the Java Rest Annotations
        sf.setBindingId(HttpBindingFactory.HTTP_BINDING_ID);
        sf.setAddress(address);
        sf.getServiceFactory().setInvoker(new BeanInvoker(serviceObj));
        sf.getServiceFactory().setWrapped(false);
        sf.create();
    }
}
```

（3）客户端调用程序

客户端的核心实现部分主要由 XmlClient 组成。客户端程序文件只有 XmlClient.java。XmlClient.java 是实现客户端调用的主程序。客户端实现查询服务器的基本情况的代码如下：

```
package demo.cxf.restful.httpbinding.client;

import java.io.InputStream;
import java.net.URL;
import org.apache.cxf.helpers.IOUtils;

public final class XmlClient {
    private static String httpAddress = "http://localhost:9000/ProjectManager/";

    private XmlClient() { }
    public static void main(String[] args) throws Exception {
        // 了解项目一些基本情况
        greeting();
        greetingWithParameter();
        getProjectName();

        // 获得默认项目的内容
        getDefaultProject();

        // 根据参数来获得项目的内容
        getProjectFromID();
    }


    // 不带参数获取后台信息
    private static void greeting() throws Exception{
        System.out.println();
        System.out.println("Sent HTTP GET request to greeting from xml");
        String thisHttpAddress = httpAddress + "xml/greeting";
        URL url = new URL(thisHttpAddress);
        InputStream in = url.openStream();
        System.out.println(getStringFromInputStream(in));
```

```
    }

    // 带参数获取后台信息
    private static void greetingWithParameter()throws Exception {
        System.out.println();
        System.out.println("Sent HTTP GET request to greetingWithParameter from xml");
        String thisHttpAddress = httpAddress + "xml/greetingWithParameter?greet=greetisgood";
    URL url = new URL(thisHttpAddress);
        InputStream in = url.openStream();
        System.out.println(getStringFromInputStream(in));
    }

    // 不带参数获取后台的默认项目信息
    private static void getProjectName() throws Exception{
        System.out.println();
        System.out.println("Sent HTTP GET request to getProjectName from xml");
        String thisHttpAddress = httpAddress + "xml/getProjectName/";
        URL url = new URL(thisHttpAddress);
        InputStream in = url.openStream();
        System.out.println(getStringFromInputStream(in));
    }

    // 不带参数获取后台的默认项目 Bean
    private static void getDefaultProject() throws Exception{
        System.out.println();
        System.out.println("Sent HTTP GET request to getDefaultProject from xml");
        String thisHttpAddress = httpAddress + "xml/getDefaultProject/";
        URL url = new URL(thisHttpAddress);
        InputStream in = url.openStream();
        System.out.println(getStringFromInputStream(in));
    }

    // 带参数获取后台的默认项目 Bean
    private static void getProjectFromID() throws Exception{
        System.out.println();
        System.out.println("Sent HTTP GET request to getProjectFromID from xml");
        String thisHttpAddress = httpAddress + "xml/getProjectNameByID?projectID=projectID2";
        URL url = new URL(thisHttpAddress);
        InputStream in = null ;
        in = url.openStream();
        System.out.println(getStringFromInputStream(in));
    }

    private static String getStringFromInputStream(InputStream in) throws Exception
    {
        return IOUtils.toString(in);
    }

}
```

使用标准的 HTTP 状态作为状态码。成功的操作都会返回 200。如果系统无法根据资源的 ID 找到它，就会返回 404。任何系统故障导致的错误都会返回 500。

4. 测试和运行结果

测试编写好的程序代码，首先要启动服务器程序，即在 Eclipse 上运行 Server.java。然后运行客户端程序，客户端程序输出结果如下：

```
Sent HTTP GET request to greeting from xml
<ns2:String xmlns:ns2="http://webservices.cxf.demo/">Hello, This is ProjectManager.
</ns2:String>

Sent HTTP GET request to greetingWithParameter from xml
<ns2:String xmlns:ns2="http://webservices.cxf.demo/">Hello, This is ProjectManager.
</ns2:String>

Sent HTTP GET request to getProjectName from xml
<ns2:String xmlns:ns2="http://webservices.cxf.demo/">default projectName</ns2:String>

Sent HTTP GET request to getDefaultProject from xml
<Project><projectDescribe>default
projectDescribe</projectDescribe><projectID>default
projectID</projectID><projectName>default projectName</projectName></Project>

Sent HTTP GET request to getProjectFromID from xml
<ns2:String xmlns:ns2="http://webservices.cxf.demo/">default projectName</ns2:String>
```

5. 其他内容

上述的所有调用都基于 JAX-WS 规范。注意，这里是 JAX-WS 规范，不是 JAX-RS 规范。

本例代码参见配套资源代码中“第 11 章\restful_http_binding_xml”文件夹。

11.5.7　Http_Binding 在 Servlet 实现基于 XML 的 RESTful 方式

本程序演示 Apache CXF 采用 Http_Binding 来实现客户端和 Servlet 服务器端的 RESTful 调用，客户端采用 Apache CXF 的代码优先（Java First）前端应用模式。服务器端采用 Servlet 容器。本例 Servlet 容器采用 Tomcat 6。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询和修改操作。

在这个过程中，前台客户端采用标准的 java.net.URL 对象，后台服务器端为遵循 Apache CXF 框架的 JAX-WS 规范的 JaxWsServerFactoryBean 对象来实现。实现的结构如图 11-22 所示。

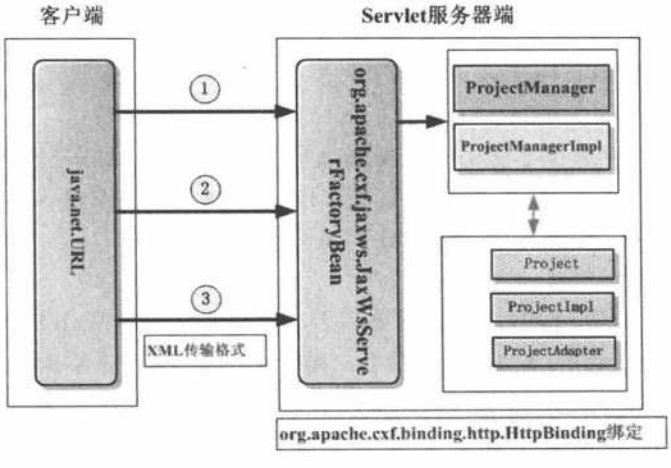


图 11-22　基于 XML 的 Http_Binding 实现 Servlet 的 RESTful 应用结构图

其中：① 表示客户端通过 URL 访问后台的基本情况；② 表示客户端通过 URL 访问后台的项目基本情况；③表示客户端通过 URL 修改默认项目。

2. 应用程序案例实现步骤

要创建Http_Binding实现基于XML的RESTful方式并在Servlet容器的前端实现Web Services程序的流程步骤如图 11-23 所示。

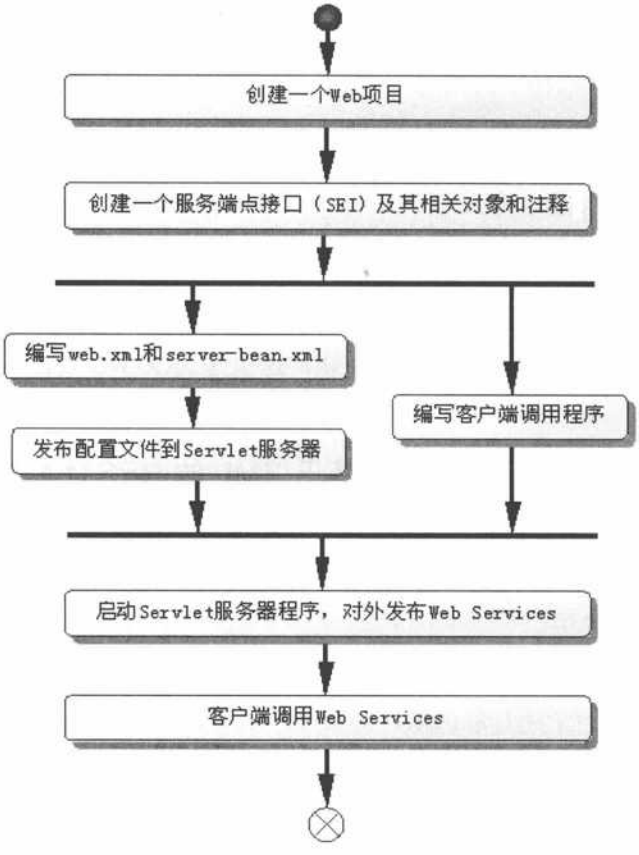


图 11-23 应用程序案例实现步骤流程图

流程图说明，要实现以下步骤：

（1）创建一个 Web 服务项目。

（2）创建一个基于 JAX-WS 服务的服务端点接口（SEI）及其相关对象和注释。

（3）采用 Http_Binding 的 XML 方式，编写基于 JAX-RS 服务的 Web Services 的客户端程序。

（4）编写基于 JAX-WS 服务的 web.xml 和 server-bean.xml。

（5）发布配置文件到 Servlet 服务器。

（6）启动 Servlet 服务器程序，对外发布基于 JAX-RS 服务的 Web Services。

（7）基于 Http_Binding 的 XML 方式的客户端调用后台基于 JAX-RS 服务的 Web Services。

其中第 1 步和第 3 步与前面采用自带的 HTTP 服务器相似。主要是第 2 步，该步骤不是采用自带的服务，而是放置到 Servlet 容器中。我们采用与 Spring 框架集成的方式来实现。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 11-24 所示。

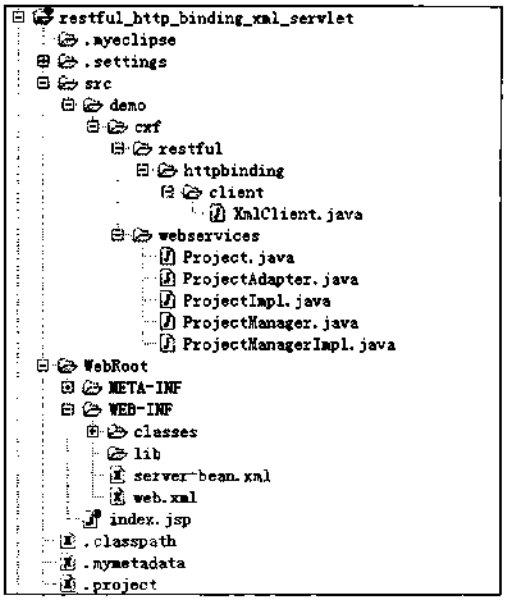


图 11-24　restful_http_binding_xml_servlet 程序结构

包目录包括三个部分，第一个包目录 WebRoot 目录是服务器程序目录，包括 web.xml 和 server-bean.xml 文件。第二个包目录 demo.cxf.restful.httpbinding.client 是客户端程序目录。第三个包目录 demo.cxf.webservices 是 SEI 和数据对象等公用部分，客户端和服务器端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口。由于本程序直接发布到 Servlet 容器，所以没有单独编写服务器发布程序，直接把配置文件和编译好的服务器端点接口（SEI）组件等部署到 Servlet 容器即可。

下面分别介绍两部分：①服务器端配置文件；②客户端调用程序。

（1）服务器端配置文件

服务器端配置文件包括 web.xml 文件和 server-bean.xml 文件。

web.xml 文件的定义与采用 POJO 并基于 Servlet 容器的前端实现案例相同，这里就不再列出了。其中，当 Servlet 启动，将读取 WEB-INF 目录下的 server-bean.xml 文件，server-bean.xml 文件的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://cxf.apache.org/jaxws
    http://cxf.apache.org/schemas/jaxws.xsd">
```

```
    <import resource="classpath:META-INF/cxf/cxf.xml" />
    <import
        resource="classpath:META-INF/cxf/cxf-extension-http-binding.xml" />
    <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />

    <jaxws:endpoint id="ProjectManager"
        implementor="demo.cxf.webservices.ProjectManagerImpl"
        address="/ProjectManager/xml"
        bindingUri="http://apache.org/cxf/binding/http">
        <jaxws:serviceFactory>
            <bean class="org.apache.cxf.jaxws.support.JaxWsServiceFactoryBean">
                <property name="wrapped" value="false" />
            </bean>
        </jaxws:serviceFactory>
    </jaxws:endpoint>

</beans>
```

将服务端程序发送到 Tomcat 中。

当 Servlet 容器启动，调用 web.xml 进行初始化，然后调用 server-bean.xml 文件，从 jaxrs:server 元素中了解到后台的 SEI 是 demo.cxf.webservices.ProjectManagerImpl，公开 Web Services 服务。

（2）客户端调用程序

采用 JAX-RS 规范的前端客户端，只需修改后台服务的位置即可，代码如下：

```
    ……
public final class XmlClient {

    private static String httpAddress =
"http://localhost:18080/restful_http_binding_xml_servlet/ProjectManager/";

    private XmlClient() { }

    public static void main(String[] args) throws Exception {
     // 了解项目一些基本情况
        greeting();
        greetingWithParameter();
        getProjectName();

        // 获得默认项目的内容
        getDefaultProject();

        // 根据参数来获得项目的内容
        getProjectFromID();

 ……

}
```

注意：笔者的 Tomcat 采用的是 18080 端口。

4. 测试和运行结果

测试编写好的程序代码，首先要启动 Tomcat 服务器程序。然后在 Eclipse 上运行客户端程序，其客户端程序和服务器端输出结果与前面的相同。

5. 其他内容

上述的所有调用都基于 JAX-WS 规范。注意，这里是 JAX-WS 规范。

本例代码参见配套资源代码中“第 11 章\restful_http_binding_xml_servlet”文件夹。

11.5.8 Http_Binding 实现基于 JSON 的 RESTful 方式

本程序演示 Apache CXF 框架采用 Http_Binding 前端模式实现基于 JSON 的客户端和 RESTful 服务器端调用。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询。在这个过程中，都使用 HTTP 实现客户端和服务器端的数据交换，采用 JSON 数据格式进行传递。前台客户端采用标准的 java.net.URL 对象，后台服务器端为遵循 Apache CXF 框架的 JAX-WS 规范的 JaxWsServerFactoryBean 对象来实现。实现的结构如图 11-25 所示。

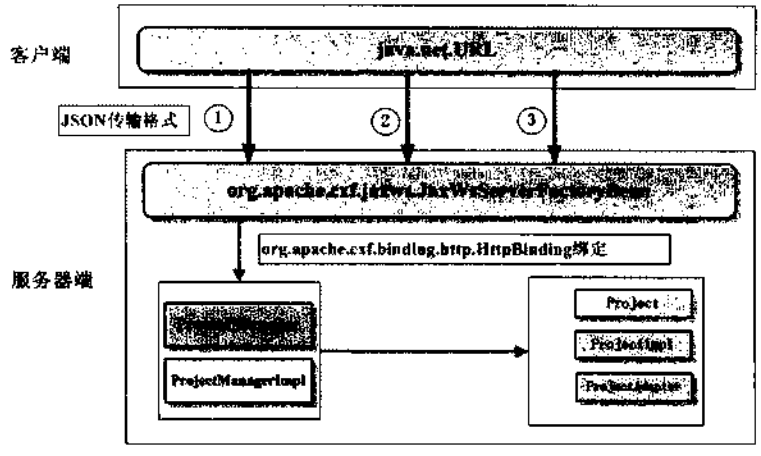


图 11-25　基于 JSON 的 Http_Binding 实现 RESTful 应用结构图

其中：① 表示客户端通过 URL 访问后台的基本情况；② 表示客户端通过 URL 访问后台的项目基本情况；③表示客户端通过 URL 修改默认项目。

2. 应用程序案例实现步骤

要创建 Http_Binding 实现基于 JSON 的 RESTful 方式的前端实现 Web Services 程序的流程步骤如图 11-26 所示。

流程图说明，要实现以下步骤：

（1）创建一个 Java 项目。

（2）创建一个基于 JAX-WS 服务的服务端点接口（SEI）及其相关对象和注释。

（3）编写发布这个 Java 程序为 Web Services 的服务器端程序，支持 JAX-RS 服务。

（4）采用 HttpBinding 并基于 JAX-RS 服务的前端客户端应用程序，调用方式为 JSON。

（5）启动服务器程序，对外发布基于 JAX-RS 服务的 Web Services。

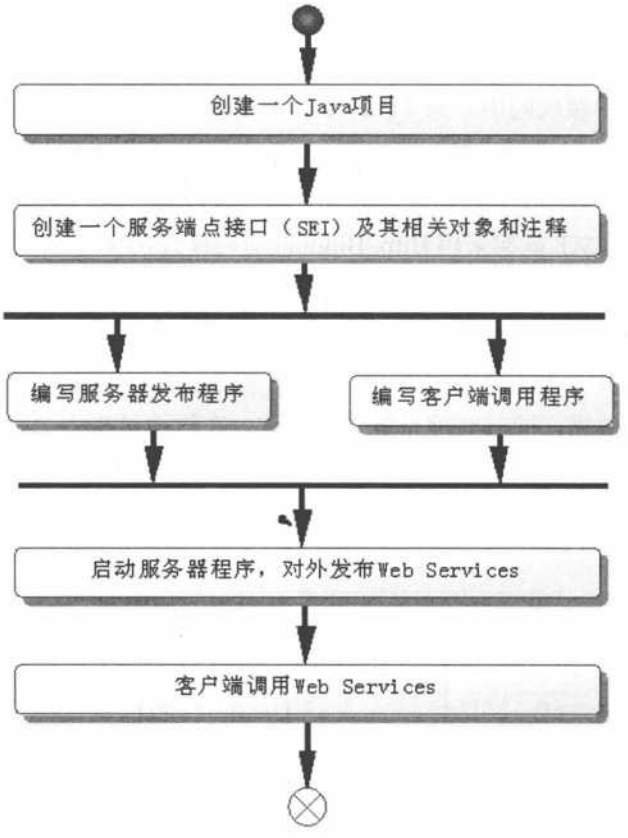


图 11-26　应用程序案例实现步骤流程图

（6）基于 JAX-RS 服务的前端客户端调用后台基于 JAX-RS 服务的 Web Services。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 11-27 所示。

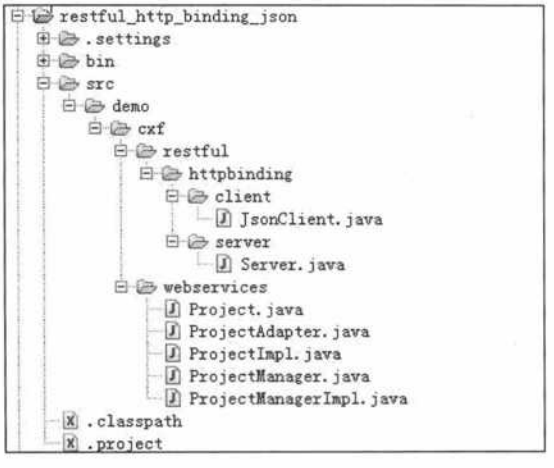


图 11-27　restful_http_binding_json 程序结构

包目录包括三个部分，第一个包目录 demo.cxf.restful.httpbinding.server 是服务器程序目录。第二个包目录 demo.cxf.restful.httpbinding.client 是客户端程序目录。第三个包目录 demo.cxf.webservices 是服务器端点接口 SEI 和相关对象等公用部分，服务器端和客户端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口，服务器端和客户端都必须引用。但服务器端和客户端对公用部分的引用是不同的。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口。而服务器端程序是接口和实现类都要引用。

下面分别介绍三部分：①服务器端点接口 SEI 和相关对象；②服务器端发布程序；③客户端调用程序。

（1）服务器端点接口（SEI）和相关对象

首先要介绍的是服务器端点接口（SEI）。

Project 组件包括 WebService 包下的 Project 几个文件。

ProjectManager 接口内容基本与 Http_Binding 实现基于 XML 的 RESTful 方式的应用的内容一样。这里就不再说明了。

（2）服务器端发布程序

服务器端发布程序就是 Server.java 文件。其作用就是启动服务器。服务器端代码如下：

```
package demo.cxf.restful.httpbinding.server;

import java.util.HashMap;
import java.util.Map;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLOutputFactory;
import org.apache.cxf.binding.http.HttpBindingFactory;
import org.apache.cxf.jaxws.JaxWsServerFactoryBean;
import org.apache.cxf.service.invoker.BeanInvoker;
import org.codehaus.jettison.mapped.MappedXMLInputFactory;
import org.codehaus.jettison.mapped.MappedXMLOutputFactory;

import demo.cxf.webservices.ProjectManager;
import demo.cxf.webservices.ProjectManagerImpl;

public final class Server {
    private static String httpAddress = "http://localhost:9000/ProjectManager/";
    private Server() {}

    public static void main(String[] args) throws Exception {
        ProjectManager projectManager = new ProjectManagerImpl();
        createJsonRestService(projectManager);
        System.out.println("Started ProjectManager!");
        System.out.println("Server ready...");
        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }

    private static void createJsonRestService(Object serviceObj) {
        String address = httpAddress + "json/";
        // Build up the server factory bean
        JaxWsServerFactoryBean sf = new JaxWsServerFactoryBean();
```

```
        sf.setServiceClass(ProjectManager.class);
        // Use the HTTP Binding which understands the Java Rest Annotations
        sf.setBindingId(HttpBindingFactory.HTTP_BINDING_ID);
        sf.setAddress(address);
        sf.getServiceFactory().setInvoker(new BeanInvoker(serviceObj));

        sf.getServiceFactory().setWrapped(false);
        Map<String, Object> properties = new HashMap<String, Object>();
        properties.put("Content-Type", "text/plain");

        Map<String, String> nstojns = new HashMap<String, String>();
        nstojns.put("http://webservices.cxf.demo/", "ProjectManager");

        MappedXMLInputFactory xif = new MappedXMLInputFactory(nstojns);
        properties.put(XMLInputFactory.class.getName(), xif);

        MappedXMLOutputFactory xof = new MappedXMLOutputFactory(nstojns);
        properties.put(XMLOutputFactory.class.getName(), xof);

        sf.setProperties(properties);
        sf.create();
    }
}
```

（3）客户端调用程序

客户端的核心实现部分主要由 XmlClient 组成。客户端程序文件只有 XmlClient.java，XmlClient.java 是实现客户端调用的主程序。客户端实现查询服务器的基本情况的代码如下：

```
package demo.cxf.restful.httpbinding.client;

import java.io.InputStream;
import java.net.URL;
import org.apache.cxf.helpers.IOUtils;

public final class JsonClient {
    private static String httpAddress = "http://localhost:9000/ProjectManager/";
    private JsonClient() {}
    public static void main(String[] args) throws Exception {
        // 了解项目一些基本情况
        greeting();
        greetingWithParameter();
        getProjectName();

        // 获得默认项目的内容
        getDefaultProject();

        // 根据参数来获得项目的内容
        getProjectFromID();
    }

    // 不带参数获取后台信息
    private static void greeting() throws Exception {
        System.out.println();
        System.out.println("Sent HTTP GET request to greeting from json");
        String thisHttpAddress = httpAddress + "json/greeting";
        URL url = new URL(thisHttpAddress);
        InputStream in = url.openStream();
```

```
        System.out.println(getStringFromInputStream(in));
    }

    // 带参数获取后台信息
    private static void greetingWithParameter() throws Exception {
        System.out.println();
        System.out.println("Sent HTTP GET request to greetingWithParameter from json");
        String thisHttpAddress = httpAddress
                + "json/greetingWithParameter?greet=greetisgood";
        URL url = new URL(thisHttpAddress);
        InputStream in = url.openStream();
        System.out.println(getStringFromInputStream(in));
    }

    // 不带参数获取后台的默认项目信息
    private static void getProjectName() throws Exception {
        System.out.println();
        System.out.println("Sent HTTP GET request to getProjectName from json");
        String thisHttpAddress = httpAddress + "json/getProjectName";
        URL url = new URL(thisHttpAddress);
        InputStream in = url.openStream();
        System.out.println(getStringFromInputStream(in));
    }

    // 不带参数获取后台的默认项目 Bean
    private static void getDefaultProject() throws Exception {
        System.out.println();
        System.out.println("Sent HTTP GET request to getDefaultProject from json");
        String thisHttpAddress = httpAddress + "json/getDefaultProject";
        URL url = new URL(thisHttpAddress);
        InputStream in = url.openStream();
        System.out.println(getStringFromInputStream(in));
    }

    // 带参数获取后台的默认项目 Bean
    private static void getProjectFromID() throws Exception {
        System.out.println();
        System.out.println("Sent HTTP GET request to getProjectFromID from json");
        String thisHttpAddress = httpAddress
                + "json/getProjectNameByID?projectID=projectID2";
        URL url = new URL(thisHttpAddress);
        InputStream in = url.openStream();
        System.out.println(getStringFromInputStream(in));
    }

    private static String getStringFromInputStream(InputStream in)
            throws Exception {
        return IOUtils.toString(in);
    }
}
```

使用标准的 HTTP 状态作为状态码。成功的操作都会返回 200。如果系统无法根据资源的 ID 找到它，就会返回 404。任何系统故障导致的错误都会返回 500。

4. 测试和运行结果

测试编写好的程序代码，首先要启动服务器程序，即在 Eclipse 上运行 Server.java。然后运行

客户端程序，客户端程序输出结果如下：

```
Sent HTTP GET request to greeting from json
{"ProjectManager.String":"Hello, This is ProjectManager."}

Sent HTTP GET request to greetingWithParameter from json
{"ProjectManager.String":"Hello, This is ProjectManager."}

Sent HTTP GET request to getProjectName from json
{"ProjectManager.String":"default projectName"}

Sent HTTP GET request to getDefaultProject from json
{"Project":{"projectDescribe":"default      projectDescribe","projectID":"default
projectID","projectName":"default projectName"}}

Sent HTTP GET request to getProjectFromID from json
{"ProjectManager.String":"default projectName"}
```

5. 其他内容

上述的所有调用都基于 JAX-WS 规范。注意，这里是 JAX-WS 规范。

本例代码参见配套资源代码中“第 11 章\restful_http_binding_json”文件夹。

11.5.9 Http_Binding 在 Servlet 实现基于 JSON 的 RESTful 方式

本程序演示 Apache CXF 框架采用 Http_Binding 来实现客户端和 Servlet 服务器端的 RESTful 调用，客户端采用 Apache CXF 的代码优先（Java First）前端应用模式。服务器端采用 Servlet 容器。本例 Servlet 容器采用 Tomcat 6。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询操作。

在这个过程中，前台客户端采用标准的 java.net.URL 对象，后台服务器端为遵循 Apache CXF 框架的 JAX-WS 规范的 JaxWsServerFactoryBean 对象来实现。实现的结构如图 11-28 所示。

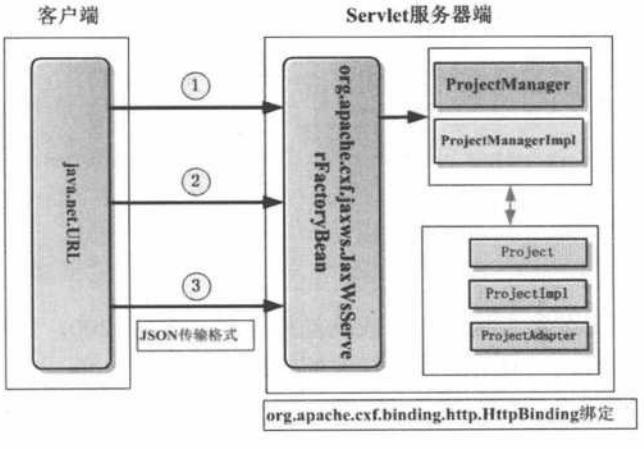


图 11-28 基于 JSON 的 Http_Binding 实现 Servlet 的 RESTful 应用结构图

其中：① 表示客户端通过 URL 访问后台的基本情况；② 表示客户端通过 URL 访问后台的项目基本情况；③表示客户端通过 URL 修改默认项目。

2. 应用程序案例实现步骤

要创建 Http_Binding 实现基于 JSON 的 RESTful 方式并在 Servlet 容器的前端实现 Web Services 程序的流程步骤如图 11-29 所示。

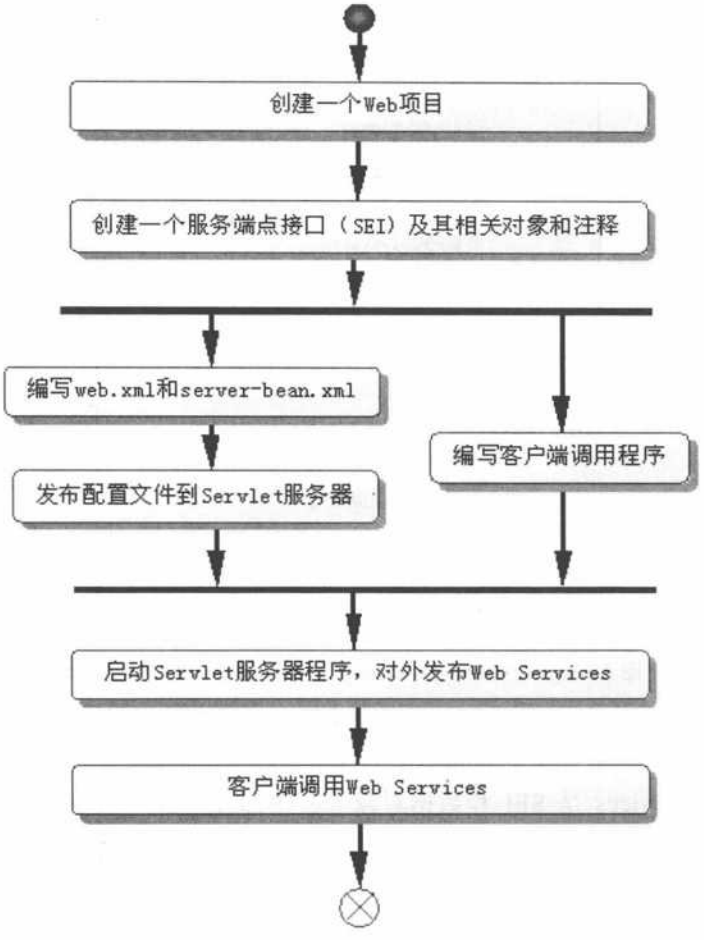


图 11-29　应用程序案例实现步骤流程图

流程图说明，要实现以下步骤：

（1）创建一个 Web 服务项目。

（2）创建一个基于 JAX-WS 服务的服务端点接口（SEI）及其相关对象和注释。

（3）采用 Http_Binding 的 JSON 方式，编写基于 JAX-RS 服务的 Web Services 的客户端程序。

（4）编写基于 JAX-RS 服务的 web.xml 和 server-bean.xml。

（5）发布配置文件到 Servlet 服务器。

（6）启动 Servlet 服务器程序，对外发布基于 JAX-RS 服务的 Web Services。

（7）基于 Http_Binding 的 JSON 方式的客户端调用后台基于 JAX-RS 服务的 Web Services。

其中第 1 步和第 3 步与前面采用自带的 HTTP 服务器类似。主要是第 2 步，该步骤不是采用自带的服务，而是放置到 Servlet 容器中。我们采用与 Spring 框架集成的方式来实现。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 11-30 所示。

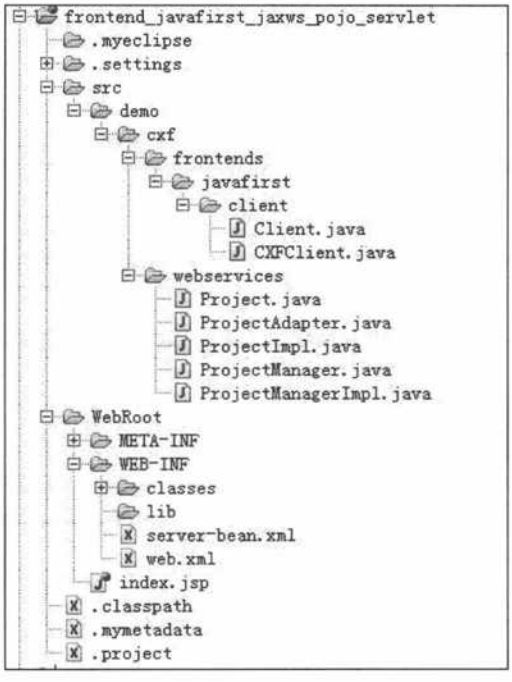


图 11-30　frontend_javafirst_jaxws_pojo_servlet 程序结构

包目录包括三个部分，第一个包目录 WebRoot 是服务器程序目录，包括 web.xml 和 server-bean.xml 文件。第二个包目录 demo.cxf.restful.httpbinding.client 是客户端程序目录。第三个包目录 demo.cxf.webservices 是 SEI 和数据对象等公用部分，客户端和服务器端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口。由于本程序直接发布到 Servlet 容器，所以没有单独编写服务器发布程序，直接把配置文件和编译好的服务器端点接口（SEI）组件等部署到 Servlet 容器即可。

下面分别介绍两部分：①服务器端配置文件；②客户端调用程序。

（1）服务器端配置文件

服务器端配置文件包括 web.xml 文件和 server-bean.xml 文件。

web.xml 文件的定义与第 6 章“采用 POJO 并基于 Servlet 容器的前端实现”的案例相同，这里就不再列出了。其中，当 Servlet 启动，将读取 WEB-INF 目录下的 server-bean.xml 文件，server-bean.xml 文件的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://cxf.apache.org/jaxws
    http://cxf.apache.org/schemas/jaxws.xsd">

    <import resource="classpath:META-INF/cxf/cxf.xml" />
    <import resource="classpath:META-INF/cxf/cxf-extension-http-binding.xml" />
    <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />

    <jaxws:endpoint id="ProjectManager"
        implementor="demo.cxf.webservices.ProjectManagerImpl"
        address="/ProjectManager/json"
        bindingUri="http://apache.org/cxf/binding/http">
        <jaxws:serviceFactory>
            <bean class="org.apache.cxf.jaxws.support.JaxWsServiceFactoryBean">
                <property name="wrapped" value="false" />
            </bean>
        </jaxws:serviceFactory>
    </jaxws:endpoint>

</beans>
```

把服务器端程序发送到 Tomcat 中。

当 Servlet 容器启动，调用 web.xml 进行初始化，若有 listener 选项，调用其内容，为 org.springframework.web.context.ContextLoaderListener，关于 Spring 的配置在这里就不做详细说明。然后就调用 server-bean.xml 文件，从 jaxrs:server 元素中了解到后台的 SEI 是 demo.cxf.webservices.ProjectManagerImpl，公开 Web Services 服务。

（2）客户端调用程序

采用 JAX-RS 规范的前端客户端，只需修改后台服务的位置即可，代码如下：

```
    ……
public final class JsonClient {

    private static String httpAddress =
"http://localhost:18080/restful_http_binding_xml_servlet/ProjectManager/";

   private XmlClient() { }

   public static void main(String[] args) throws Exception {
    // 了解项目一些基本情况
        greeting();
        greetingWithParameter();
        getProjectName();

        // 获得默认项目的内容
        getDefaultProject();

        // 根据参数来获得项目的内容
        getProjectFromID();
 ……
}
```

注意：笔者的 Tomcat 采用的是 18080 端口。

4. 测试和运行结果

测试编写好的程序代码，首先要启动 Tomcat 服务器程序。然后在 Eclipse 上运行客户端程序，其客户端程序和服务器端输出结果与前面的相同。

5. 其他内容

上述的所有调用都基于 JAX-WS 规范。注意，这里是 JAX-WS 规范。

本例代码参见配套资源代码中“第 11 章\restful_http_binding_json_servlet”文件夹。

第 12 章　CXF 对动态语言的支持

本章主要说明 Apache CXF 对动态语言（Dynamic Languages）的支持。首先介绍 Apache CXF 采用 JavaScript 来实现 Web Services，其次介绍 Apache CXF 支持 E4X (ECMAScript for XML)的实现方式。最后用例子来进行说明。

本章内容：

1. CXF 对 JavaScript 等语言的支持。
2. CXF 实现 JavaScript 等语言的例子。

12.1　CXF 对 JavaScript 等语言的支持

Apache CXF 框架对 JavaScript 等语言的支持，覆盖采用 JavaScript 来实现 Web Services，同时包括用 E4X（ECMAScript for XML）来实现 Web Services 和如何部署 Script 服务。

12.1.1　用 JavaScript 来实现 Web Services

用 JavaScript 实现 Web Services 有两个步骤，首先是定义了 JAX-WS 样式的元数据，其次是编写服务的业务逻辑代码。

1. 定义了 JAX-WS 样式的元数据

Java 使用 Java 注释来指定 JAX-WS 元数据。由于 JavaScript 不支持注释，所以采用普通的 JavaScript 变量来定义 JavaScript 实现的元数据。Apache CXF 框架把任何相同名字或用 WebServiceProvider 开头的 JavaScript 变量都作为 JAX-WS 元数据变量。

变量属性希望能定义相同的元数据，这些元数据是由 JAX-WS 的 WebServiceProvider 来进行注解的，其中包括：

- wsdlLocation 指定 WSDL 定义服务的网址。
- servicename 指定服务的名称。
- portName 指定服务的端口/接口名称。
- targetNamespace 指定服务的目的地命名空间。

JavaScript 的 WebServiceProvider 还可以指定以下可选属性：

- ServiceMode 表示定义服务是否是 SOAP 有效载荷文件或是处理 SOAP 消息全部文件。此属性模仿了 JAX-WS 的 ServiceMode 注释，默认值是 PAYLOAD。
- BindingMode 表示服务绑定 ID 网址。默认的是 SOAP 1.1/HTTP 绑定。
- EndpointAddress 表明这 URL 消费应用使用服务进行通信。该属性是可选项，但没有默认值。

下面例子显示了 JavaScript 实现服务的一个元数据描述。

```
var WebServiceProvider1 = {
   'wsdlLocation': 'file:./wsdl/hello_world.wsdl',
   'serviceName': 'SOAPService1',
   'portName': 'SoapPort1',
   'targetNamespace': 'http://apache.org/hello_world_soap_http',
};
```

2. 编写服务的业务逻辑代码

可以通过查询 WebServiceProvider 变量的调用属性来实现服务的逻辑。这个变量是一个函数，它接受一个输入参数、一个 javax.xml.transform.dom.DOMSource 节点，并返回相同类型的文档。该调用函数使用作为 Java 应用的常规 Java DOMSource 类接口来多次操纵输入或输出文件。

下面例子描述了一个 JavaScript 实现简单服务的调用属性。

```
WebServiceProvider.invoke = function(document) {
   var ns4 = "http://apache.org/hello_world_soap_http/types";
   var list = document.getElementsByTagNameNS(ns4, "requestType");
   var name = list.item(0).getFirstChild().getNodeValue();
   var newDoc = document.getImplementation().createDocument(ns4, "ns4:greetMeResponse",
null);
   var el = newDoc.createElementNS(ns4, "ns4:responseType");
   var txt = newDoc.createTextNode("Hi " + name);
   el.insertBefore(txt, null);
   newDoc.getDocumentElement().insertBefore(el, null);
   return newDoc;
}
```

12.1.2 用 E4X（ECMAScript for XML）来实现 Web Services

用 E4X 编写 Apache CXF 框架的 Web Services 与用 JavaScript 非常相似。在 E4X 中使用相同的 WebServiceProvider 变量来定义 JAX-WS 元数据。也在 WebServiceProvider 变量的调用属性中编写该服务的实现逻辑。

两种方法之间唯一的区别是操纵不同的文件类型。当使用 E4X 时，E4X 的 XML 文档既可以操作请求文档，也可以操作返回值。这些文档将依赖 E4X 内置的 XML 特性来进行操作。

下面例子描述了 E4X 实现一个简单服务的调用属性。

```
var SOAP_ENV = new Namespace('SOAP-ENV', 'http://schemas.xmlsoap.org/soap/envelope/');
var xs = new Namespace('xs', 'http://www.w3.org/2001/XMLSchema');
var xsi = new Namespace('xsi', 'http://www.w3.org/2001/XMLSchema-instance');
var ns = new Namespace('ns', 'http://apache.org/hello_world_soap_http/types');

WebServiceProvider1.invoke = function(req) {
   default xml namespace = ns;
   var name = (req..requestType)[0];
```

```
    default xml namespace = SOAP_ENV;
    var resp = <SOAP-ENV:Envelope xmlns:SOAP-ENV={SOAP_ENV} xmlns:xs={xs} xmlns:xsi=
{xsi}/>;
    resp.Body = <Body/>;
    resp.Body.ns::greetMeResponse = <ns:greetMeResponse xmlns:ns={ns}/>;
    resp.Body.ns::greetMeResponse.ns::responseType = 'Hi ' + name;
    return resp;
}
```

12.1.3　部署 Script 服务

Apache CXF 框架部署了一个基于 JavaScript 和 E4X 的 Web 服务轻量级的容器，并可以利用 Apache CXF 的插件式传输架构。

可以使用下列命令来发布服务到容器中：

```
java org.apache.cxf.js.rhino.ServerApp [ -a addressURL ] [ -b baseAddressURL ] file.js
[ file2.js file3.jsx ... ]
```

该 org.apache.cxf.js.rhino.ServerApp 类需要一个或多个 JavaScript 文件，以 js 为后缀的 JavaScript 文件或以 JSX 为后缀的 E4X 文件，并在 Apache CXF 运行时加载这些类。如果 ServerApp 在文件中定位到 JAX-WS 元数据，则 ServerApp 为每个服务创建并注册一个 JAX-WS 的 Provider<DOMSource>对象。该 Provider<DOMSource>对象代理一个请求处理并在相关文件中实现。ServerApp 也采用一个包含 JavaScript 和 E4X 文件的目录名称。这将加载包含 JAX-WS 元数据的脚本，加载所有这些文件后，就可以在每个端点发布这些服务。ServerApp 命令的参数选项和说明见表 12-1。

表 12-1　ServerApp命令的参数选项和说明

参数选项		说明
-a	addressURL	ServerApp发布指定脚本文件的服务端点应用到指定的网址
-b	baseAddressURL	指定ServerApp使用的基本网址，使用脚本文件指定的服务器端点应用来进行发布。该服务器端点应用的详细地址是附加了服务的端口名称
-v		确定ServerApp是否运行在verbose模式

可选的参数使用{{EndpointAddress}}属性来处理任何地址信息，该属性存在于 JAX-WS 元数据。

例如，开发人员使用下面例子，其含义是显示命令来部署服务，服务将部署在 http://cxf.apache.org/goodness。

```
java org.apache.cxf.js.rhino.ServerApp -a http://cxf.apache.org/goodness hello_world.jsx
```

使用一个通用的基础 URL 来部署大量的服务，可以使用上面所表示的命令。如果 hello_world.jsx 定义服务的端口名称是 HelloWorld，ServerApp 将发布服务为 http://cxf.apache.org/helloWorld。如果服务由 goodbye_moon.js 指定端口名称是 blue，ServerApp 将发布服务为 http://cxf.apache.org/blue。

```
java  org.apache.cxf.js.rhino.ServerApp  -b  http://cxf.apache.org  hello_world.jsx
goodbye_moon.js
```

12.2 CXF 基于 JavaScript 等语言实现 Web Services 的例子

这里仅仅用一个例子来说明 Apache CXF 框架是如何通过 JavaScript 实现 Web Services 的。

12.2.1 用 JavaScript 调用 CXF 的 Web Services

本程序演示 JavaScript 调用 Apache CXF 框架的 Web Services。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，前端访问采用 HTML 页面方式，然后引用 JavaScript 语言来处理，后台服务器为遵循 JAX-WS 规范的 Endpoint 来实现服务资源。实现如图 12-1 所示。

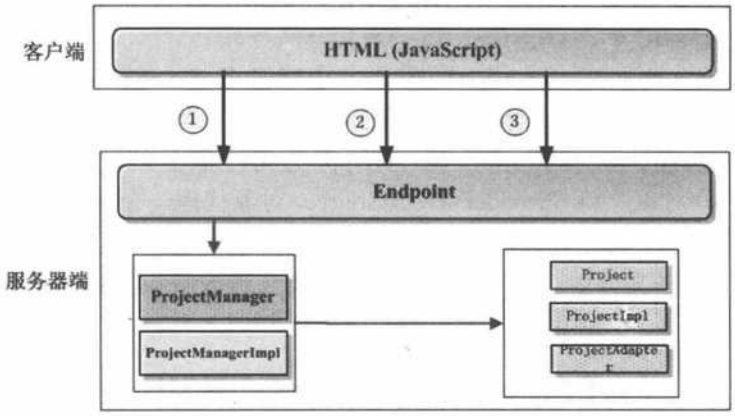


图 12-1 基于 JavaScript 实现 Web Services 应用图

其中：① 表示客户端通过 HTTP 访问后台的项目基本情况；② 表示客户端通过 HTTP 修改默认项目；③ 表示客户端通过 HTTP 增加一个项目。

2. 应用程序案例实现步骤

要创建 JavaScript 并前端实现 Web Services 程序的流程如图 12-2 所示。

流程图说明，要实现以下步骤：

（1）创建一个 Java 项目。

（2）创建一个服务器端点接口（SEI）及其相关对象和注释。

（3）编写发布这个 Java 程序为 Web Services 的服务器端程序。

（4）编写 cxf.xml 文件。

（5）编写 Web Services 的客户端应用程序，主要是在页面上编写 JavaScript 代码。

（6）启动服务器程序，对外发布 Web Services。

（7）客户端调用 Web Services。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例结构如图 12-3 所示。

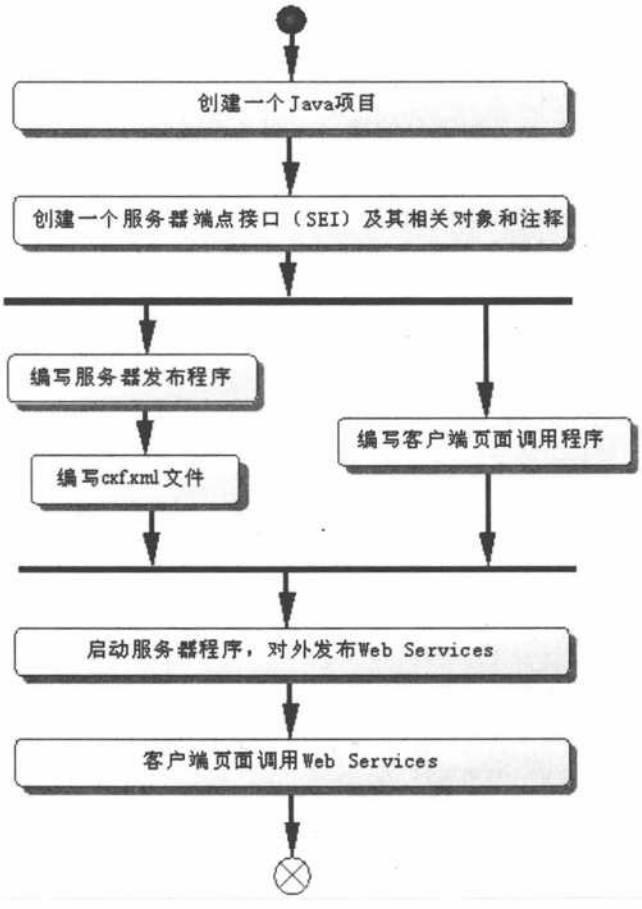


图 12-2　应用程序案例实现步骤流程图

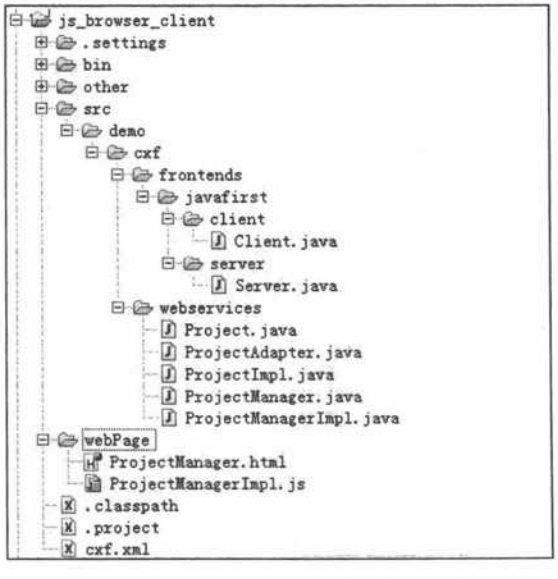


图 12-3　js_browser_client 程序结构

包目录包括三个部分，第一个包目录 demo.cxf.frontends.javafirst.server 是服务器程序目录。第二个包目录 webPage 是客户端程序目录。第三个包目录 demo.cxf.webservices 是服务器端点接口 SEI 和相关对象等公用部分，服务器端和客户端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口，服务器端和客户端都必须引用。但服务器端和客户端对公用部分的引用是不同的。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口，而服务器端程序是接口和实现类都要引用。

下面分别介绍三部分：①服务器端点接口 SEI 和相关对象；②服务器端发布程序；③客户端页面调用程序。

（1）服务器端点接口（SEI）和相关对象

首先要介绍的是服务器端点接口（SEI）。

Project 组件包括 WebService 包下的 Project 几个文件。Project 组件代码基本都是一样的，这里省略。

其中 ProjectManager.java 文件代码如下：

```
package demo.cxf.webservices;


import javax.jws.WebService;
import javax.jws.WebParam;
import javax.jws.WebMethod;
import javax.jws.WebResult;
import javax.jws.soap.SOAPBinding;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.ResponseWrapper;

@WebService(targetNamespace = "urn:webservices.cxf.demo:jsjaxws")
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
public interface ProjectManager {

    @ResponseWrapper(localName = "greetingResponse", targetNamespace = "http://
webservices.cxf.demo/")
    @RequestWrapper(localName = "greeting", targetNamespace = "http://webservices.
cxf.demo/")
    @WebResult(name = "return", targetNamespace = "")
    @WebMethod
    public String greeting() ;

    @ResponseWrapper(localName = "greetingWithParameterResponse",
targetNamespace = "http://webservices.cxf.demo/")
    @RequestWrapper(localName = "greetingWithParameter", targetNamespace = "http://
webservices.cxf.demo/")
    @WebResult(name = "return", targetNamespace = "")
    @WebMethod
    public String greetingWithParameter(@WebParam(name = "greet")String greet);

    @ResponseWrapper(localName = "getProjectNameResponse",
targetNamespace = "http://webservices.cxf.demo/")
    @RequestWrapper(localName = "getProjectName", targetNamespace = "http://
webservices. cxf.demo/")
    @WebResult(name = "return", targetNamespace = "")
    @WebMethod
    public String getProjectName();
```

```
    @WebMethod
    public String getProjectNameByID(@WebParam(name = "projectID")String projectID);

    @WebMethod
    public void updateProjectName (@WebParam(name = "projectName")String projectName);

    @WebMethod
    public void setDefaultProject(Project project);

    @WebMethod
    public void registerProject(Project project);

    @WebMethod
    public void removeProject(@WebParam(name = "id") String id);

    //@WebMethod
    //public Project getDefaultProject();

    @WebMethod
    public void getAllProject();

}
```

（2）服务器端发布程序

Server.java 是服务器启动程序文件。服务器程序文件基本都一样，这里省略。

cxf.xml 是服务器的配置文件，其代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xmlns:cxf="http://cxf.apache.org/core"
    xmlns:httpj="http://cxf.apache.org/transports/http-jetty/configuration"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/core
http://cxf.apache.org/schemas/core.xsd
http://cxf.apache.org/jaxws
http://cxf.apache.org/schemas/jaxws.xsd
http://cxf.apache.org/transports/http-jetty/configuration
http://cxf.apache.org/schemas/configuration/http-jetty.xsd">

    <httpj:engine-factory bus="cxf">
        <!-- The server runs on port 9000 -->
        <httpj:engine port="9000">
            <httpj:handlers>
                <bean
                    class="org.mortbay.jetty.handler.ContextHandler">
                    <property name="contextPath" value="/" />
                    <property name="handler">
                        <bean
                            class="org.mortbay.jetty.handler.ResourceHandler">
                            <property name="baseResource">
                                <bean
                                    class="org.mortbay.resource.FileResource">
                                    <constructor-arg
```

```
                                        value="file:webPage" />
                                    </bean>
                                </property>
                            </bean>
                        </property>
                    </bean>
                    <bean class="org.mortbay.jetty.handler.DefaultHandler" />
                </httpj:handlers>
            </httpj:engine>
        </httpj:engine-factory>

        <jaxws:endpoint
            name="{http://webservices.cxf.demo/}ProjectManagerImplService"
            createdFromAPI="true">
            <jaxws:properties>
                <entry key="schema-validation-enabled" value="true" />
            </jaxws:properties>
        </jaxws:endpoint>

    </beans>
```

配置文件的重点是黑体部分。

（3）客户端页面调用程序

客户端程序文件包括 webpage 目录下面的 ProjectManager.html 和 ProjectManagerImpl.js。ProjectManager.html 页面客户端实现查询服务器的项目基本情况的代码如下：

```
<html>
    <head>
        <title>JavaScript Client call project Sample</title>
        <!-- Generate and retrieve a JavaScript client for the server. -->
        <script type="text/javascript" src="/ProjectManager?js"></script>
        <script type="text/javascript" src="/ProjectManagerImpl.js"></script>

    </head>
    <body>
        <h1>ProjectManager Sample  </h1>
         <o:p>————————————————————</o:p>

        <form><div>
          <table width="584" height="58">
                    <tr>
                        <td width="253">
                            方法 greeting                          </td>
                        <td width="319">
                            <input type="button" value="调用" name="invokeGreeting"
                                onClick="greeting()">
                    </tr>
                    <tr>
                        <td>
                            反馈信息:
                        </td>
                        <td>
                            <span id='responseText1'>- 等待反馈 -</span>
                        </td>
```

```
            </tr>
    </table>
      </div>
    <o:p>————————————
```

```
                                反馈信息:
                            </td>
                            <td>
                                <span id='responseText3'>- 等待反馈 -</span>
                            </td>
                        </tr>
                    </table>
                </div>
        </form>
        </body>
</html>
```

展现的界面如图 12-4 所示。

图 12-4　展现的界面

客户端引用的 js 文件——ProjectManagerImpl.js 的代码如下：

```
// Create an object that talks to the server.
var server = new urn_webservices_cxf_demo_jsjaxws_ProjectManager();
// Set the URL for the service. No cross-scripting allowed.
server.url = "/ProjectManager";

var responseSpan;

// This is the function called for an error.
function onerror(error) {
    alert('error ' + error);
}

// This function is invoked from the button press to run the service.
function greeting() {
    var responseSpan = document.getElementById('responseText1');
    var greetingResponse = function(response, span) {
        span.firstChild.nodeValue = response;
    }
    var receiveFirstResponse = function(response) {
        greetingResponse(response, responseSpan);
    };
    var greetingError = function(error) {
        alert('error ' + error);
    };
```

```
    server.greeting(receiveFirstResponse, greetingError);
}

// This function is invoked from the button press to run the service.
function greetingWithParameter() {
    var responseSpan = document.getElementById('responseText2');
    var project = document.getElementById('ProjectSelect1');
    var str = project.value;
    var greetingWithParameterResponse = function(response, span) {
        span.firstChild.nodeValue = response;
    }
    var receiveFirstResponse = function(response) {
        greetingWithParameterResponse(response, responseSpan);
    };
    // alert('information ' + category);
    var greetingWithParameterError = function(error) {
        alert('error ' + error);
    };
    server.greetingWithParameter(receiveFirstResponse,
            greetingWithParameterError, str);
}

// This function is invoked from the button press to run the service.
function getProjectName() {
    var responseSpan = document.getElementById('responseText3');
    var getProjectNameResponse = function(response, span) {
        span.firstChild.nodeValue = response;
    }
    var receiveFirstResponse = function(response) {
        getProjectNameResponse(response, responseSpan);
    };
    var getProjectNameError = function(error) {
        alert('error ' + error);
    };
    alert('information getProjectName ');
    server.getProjectName(receiveFirstResponse, getProjectNameError);
}

// This function is invoked from the button press to run the service.
function getAllProject() {
    var receiveFirstResponse = function(response) {
        alert('ok ');
    };
    var getAllProjectError = function(error) {
        alert('error ' + error);
    };
    alert('information getProjectName ');
    server.getAllProject(receiveFirstResponse, getAllProjectError);
}

// This function is invoked from the button press to run the service.
function getProjectNameByID() {
    var responseSpan = document.getElementById('responseText3');
    var project = document.getElementById('ProjectSelect2');
    var str = project.value;
    var getProjectNameByIDResponse = function(response, span) {
        span.firstChild.nodeValue = response;
    }
    var receiveFirstResponse = function(response) {
```

```
        getProjectNameByIDResponse(response, responseSpan);
    };
    // alert('information ' + category);
    var getProjectNameByIDError = function(error) {
        alert('error ' + error);
    };
    server.getProjectNameByID(receiveFirstResponse,
            getProjectNameByIDError, str);
}
```

启动服务器，打开页面，调用页面按钮即可看到实现情况。

本例代码参见配套资源代码中“第 12 章\js_browser_client”文件夹。

第 13 章 CXF 对 WS-* 的支持

本章主要说明 Apache CXF 框架对 WS-* 的支持和应用。在本章中，分别介绍 Apache CXF 框架如何支持 WS-Addressing 规范、WS-Policy 规范、WS-ReliableMessaging 规范、WS-Security 规范、WS-SecurityPolicy 规范、WS-Trust 规范和 WS-SecureConversation 规范等 WS-*规范。针对每个 WS-*规范，首先简介该 WS-*规范内容，其次说明 Apache CXF 框架是如何支持它的，最后用一个示例来说明。

本章内容：

1. Apache CXF 支持 WS-Addressing 规范。
2. Apache CXF 支持 WS-Policy 规范。
3. Apache CXF 支持 WS-ReliableMessaging 规范。
4. Apache CXF 支持 WS-Security 规范。
5. Apache CXF 支持 WS-SecurityPolicy 规范。
6. Apache CXF 支持 WS-Trust 规范。
7. Apache CXF 支持 WS-SecureConversation 规范。

13.1 Apache CXF 支持 WS-Addressing

2004 年 8 月，BEA、IBM、Microsoft、SAP AG 和 Sun 公司联合宣布并向 W3C（World Wide Web Consortium）提交了一项 Web 服务规范—— WS-Addressing（Web Service Addressing）的申请案，以请求获得 W3C 组织的批准。

WS-Addressing 是针对互联网上软件服务的一项重要技术规范，它定义了端点参考和信息报头，以让系统能支持 Web 服务信息的传输。WS-Addressing 被设计位于其他规范（如 WS-ReliableMessaging 和 WS-Federafio 等）的底层。

本节简单介绍一下 WS-Addressing 的概念，Apache CXF 框架如何使用 WS-Addressing 的方法和例子。

13.1.1 WS-Addressing 简介

Web 服务寻址（WS-Addressing）是一个 W3C 推荐标准，为 Web Services 提供一种与传输层

无关的，传送寻址信息的机制。规范主要由两部分组成：传送 Web Services 端点引用的数据结构，以及一套能够在特定的消息上关联寻址信息的消息寻址属性。

WS-Addressing 规范通过定义端点引用（EPR）和消息信息（MI）头来明确达到这个目的。这个定义提供了单一的机制。通过这一机制，开发、使用人员能够指定 Web Services（或者服务的实例）的位置以及服务在 SOAP 消息中使用端点引用（EPR）的方式（通过 MI 头）。但是，正是在消息处理模型/流中的隐式改变甚至对 SOAP 都具有更加重大的影响。WS-Addressing 规范的重要性将随着时间而增加，这表明它比其他规范更重要一些，该规范本身应该是核心 SOAP 规范的一部分。

大多数的 Web Services 都通过 HTTP 协议来传输。可 SOA 的应用环境不仅仅只是 HTTP 协议，还包括 TCP、SMTP 等多种协议来传输 SOAP 消息，所以在 SOA 的背景下，Web Services 寻址方式主要面临以下三个问题。

（1）**多种协议问题**：访问 Web Services 有多种协议，而将不仅限于某个单一的传输协议（如 HTTP）。在访问 Web Services 时可能会使用 TCP、SMTP 等不同的底层协议。甚至在实际情况中，请求的 Web Services 使用一种传输协议，而答复的 Web Services 却使用另一种协议。

（2）**异步服务问题**：服务之间的交互已不再是简单的同步 request-response 方式，在实际应用中越来越多的是异步式消息交换模式，而异步式消息交换模式更加适合业务的需要，在这种情况下 response 消息如何寻址，已不再那么简单。

（3）**服务延时问题**：在现实业务中，那就是在 Web Services 的请求消息发送后，过很长一段时间才能得到回复消息（如下完订单，几个小时或几天才能得到确认消息）。这是一种广泛存在一种现象。这种情况下，如何依旧将回复消息与请求时发送的消息关联上，从而维持住 Web Services 的状态？这也是以往的寻址方式所没有考虑到的问题。

而 WS-Addressing 规范则为解决以上三个问题提供了方案。通过对 SOAP 消息的扩展，为 Web Services 的寻址问题提供更强大的支持，并且该规范也是其他各种 WS-*规范的重要基础。

为解决这些技术问题，微软、IBM、BEA、SAP 和 Sun 等 IT 公司联合推出了 WS-Addressing 协议，并提交到 W3C 组织进行标准化。W3C 组织的 WS-Addressing 工作组在标准化过程中对该规范进行了改进和补充。

WS-Addressing 规范最简单的定义就是，在 SOAP 消息的 header 中，用一系列 XML 标签定义了服务传递中间的目的地，Action、ReplyTo、Fault 处理等传输层细节。

WS-Addressing 规范是如何传输数据？WS-Addressing 协议通过在 SOAP 消息的 header 中加入数据的方式来传送数据。具体的传送方法有如下两种：Endpoint Reference 和 Message Addressing Properties。如果北京的寄信人要寄信给深圳的收信人，那么 Message Addressing Properties 就好像信封的格式（比如在信封的什么位置写邮编、在信封的什么位置写地址），而 Endpoint Reference 就表示信封上具体的邮编和地址的内容。

目前，WS-Addressing 分成三部分：

- 核心——主要包括端点引用和消息寻址属性的规范。
- SOAP 绑定——将这些属性绑定到 SOAP 协议上。
- 元数据规范——说明了在核心中定义的抽象的属性将如何使用 WSDL 来描述，如何在端

点引用中包含 WSDL 元数据，以及如何使用 WS-Policy 标明一个 Web Services 是否支持 WS-Addressing。

1. WS-Addressing 规范内容

WS-Addressing 规范是将消息路由数据包含在 SOAP 头中的一种标准方法。利用 WS-Addressing 的消息可以在标准化 SOAP 头中包含发出者和需要发送的元数据，而不依赖于网络层传输来传送路由信息。网络层传输只负责将消息发送到能够读取 WS-Addressing 元数据的分配器那里。一旦消息抵达了 URI 所制定的分配器，网络层传输的工作就完成了。通过在标准的 SOAP 头中（wsa:ReplyTo）指定应答消息应该发送到哪里的端点引用，WS-Addressing 规范可以支持异步交互方式。服务提供者使用另一个连接，将应答消息发送给 wsa:ReplyTo 指定的端点。这就将 SOAP 请求/应答消息的交互与 HTTP 请求/应答协议分离，这样，跨越任意时间的长时间运行的交互成为可能。

2. 端点引用（Endpoint Reference）

端点引用（Endpoint Reference，EPR）是一个 XML 结构，封装了对访问 Web Services 的消息寻址有用的信息。这包括了消息的目的地地址、任何其他路由消息到目的地所需的参数（称作引用参数），以及有关服务的任选元数据（如 WSDL 或 WS-Policy）。

消息寻址属性（Message Addressing Properties），表明与将消息传送到 Web Services 有关的寻址信息，包括：

- 目的地——该消息的目的地的 URI。
- 源端点——发出该消息的服务端点（EPR）。
- 应答端点——应答消息接收者的端点（EPR）。
- 故障端点——故障消息接收者的端点（EPR）。
- 动作——指示该消息的语义（可能有助于该消息的寻址）的 URI。
- 消息 ID——唯一消息标识符 URI。
- 关系——与之前消息的关系（一对 URI）。

WS-Addressing 规范的最新官方文档见如下网址：http://www.w3.org/Submission/ws-addressing/。

13.1.2　Apache CXF 的 WS-Addressing 配置

Apache CXF 框架支持 WS-Addressing 规范的 2004-08 版本和 1.0 版本。要在 Apache CXF 启用 WS-Addressing 服务，首先要启用服务的 WSAddressingFeature。

如果采用 XML 来进行配置，服务器端配置要遵循下面的语法格式：

```
<jaxws:endpoint id="{your.service.namespace}YourPortName">
  <jaxws:features>
    <wsa:addressing xmlns:wsa="http://cxf.apache.org/ws/addressing"/>
  </jaxws:features>
</jaxws:endpoint>
```

客户端配置按照下面的 XML 语法格式：

```
<jaxws: client id="{your.service.namespace}YourPortName">
  <jaxws:features>
```

```
    <wsa:addressing xmlns:wsa="http://cxf.apache.org/ws/addressing"/>
  </jaxws:features>
</jaxws:client>
```

从 API 调用的方式来看，这两者非常相似：

```
import org.apache.cxf.jaxws.EndpointImpl;
import org.apache.cxf.ws.addressing.WSAddressingFeature;

MyServiceImpl implementor = new MyServiceImpl()
EndpointImpl ep = (EndpointImpl) Endpoint.create(implementor);
ep.getFeatures().add(new WSAddressingFeature());
ep.publish("http://some/address");
```

开发人员也可以在 ClientProxyFactoryBeans 和 ServerFactoryBeans 使用 WS-Addressing 规范，或者基于 JAX-WS 版本的 JaxWsProxyFactoryBean 和 JaxWsServerFactoryBean，代码如下：

```
import org.apache.cxf.frontend.simple.ClientProxyFactoryBean;
import org.apache.cxf.ws.addressing.WSAddressingFeature;

ClientProxyFactoryBean factory = new ClientProxyFactoryBean();
factory.setServiceClass(MyService.class);
factory.setAddress("http://acme.come/some-service");
factory.getFeatures().add(new WSAddressingFeature());
MyService client = (MyService) factory.create();
```

为了使用 WS-Addressing 规范进行消息交换，下面的两项工作是必要的。

（1）地址拦截器（org.apache.cxf.ws.addressing.MAPAggregator 和 org.apache.cxf.ws.addressing.soap.MAPCodec）需要加入到入站和出站拦截器链。

（2）WS-Addressing 规范的使用有以下三种表示：

- <UsingAddressing xmlns="http://www.w3.org/2005/02/addressing/wsdl">元素附加到<wsdl:port>、<wsdl:service>或<wsdl:binding>元素。
- 消息有效策略包含一个<Addressing xmlns="http://www.w3.org/2007/02/addressing/metadata">断言，或者是<UsingAddressing>断言有以下三个命名空间：http://schemas.xmlsoap.org/ws/2004/08/addressing/policy、http://www.w3.org/2005/02/addressing/wsdl、http://www.w3 .org/2006/05/addressing/wsdl。
- 消息上下文的 org.apache.cxf.ws.addressing.using 属性设置为 Boolean.TRUE。

1. 使用寻址 Feature 功能

寻址 Feature 功能元素中定义命名空间 http://cxf.apache.org/ws/addressing。它支持两个属性，见表 13-1。

表 13-1　Feature功能元素命名空间http://cxf.apache.org/ws/addressing支持的属性

名称	数值
allowDuplicates	一个布尔值，判断是否重复容忍MessageIDs（默认是：true）

续表

名称	数值
usingAddressingAdvisory	一个布尔值，指示如果存在WSDL的 <UsingAddressing>元素，只是提供内容，即<UsingAddressing>元素的缺席并不阻止WS-A头的编码。在Java代码优先情况启用WS-Addressing特别有用，那就是不会在一定的条件下的WSDL中不会达到指定寻址

例如，在 JAX-WS 服务器端点应用寻址 Feature 功能的配置如下：

```
<beans ... xmlns:wsa="http://cxf.apache.org/ws/addressing" ...>
   <jaxws:endpoint ...>
      <jaxws:features>
         <wsa:addressing allowDuplicates="false"/>
      </jaxws:features>
   </bean>
</beans>
```

2. 手动添加寻址拦截器

org.apache.cxf.ws.addressing.MAPAggregator 和 org.apache.cxf.ws.addressing.soap.MAPCodec 必须添加到入站、出站消息和故障拦截链。

在全局层面上，可以适用于所有客户端和服务器端点，下面例子可以这样做。由于 allowDuplicates 和 usingAddressingAdvisory 本来就是 MAPAggregator 拦截器的属性，也可以通过 Spring 框架平台的语法来设置。配置文件如下：

```
<bean id="mapAggregator" class="org.apache.cxf.ws.addressing.MAPAggregator">
   <property name="allowDuplicates" value="false"/>
</bean>
<bean id="mapCodec" class="org.apache.cxf.ws.addressing.soap.MAPCodec"/>

<cxf:bus>
   <cxf:inInterceptors>
      <ref bean="mapAggregator"/>
      <ref bean="mapCodec"/>
   </cxf:inInterceptors>
   <cxf:inFaultInterceptors>
      <ref bean="mapAggregator"/>
      <ref bean="mapCodec"/>
   </cxf:inFaultInterceptors>
   <cxf:outInterceptors>
      <ref bean="mapAggregator"/>
      <ref bean="mapCodec"/>
   </cxf:outInterceptors>
   <cxf:outFaultInterceptors>
      <ref bean="mapAggregator"/>
      <ref bean="mapCodec"/>
   </cxf:outFaultInterceptors>
</cxf:bus>
```

13.1.3 Apache CXF 的 WS-Addressing 的实现例子

本程序演示 Apache CXF 框架遵循 WS-Addressing 规范的客户端和服务器端调用，客户端采用 Apache CXF 框架的代码优先（Java First）前端应用模式。采用 POJO 对象作为中间传输对象。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，都使用 POJO 对象实现客户端和服务器端的数据交换。后台服务器端为遵循 JAX-WS 规范的 Endpoint 来实现，采用 Spring IoC 注入的客户端来实现。前端访问也采用遵循 JAX-WS 规范的 Service 方式来实现。

客户机和服务器都应用到 bus 的寻址 Feature 处理功能，确保安装好 WS-Addressing 的拦截器。WS-Addressing 的拦截器包括包括逻辑拦截器（MAPAggregator）和协议拦截器（MAPCodec）。逻辑拦截器（MAPAggregator）负责聚合 WS-A MessageAddressingProperties 到当前消息。协议拦截器（MAPCodec）负责按照 SOAP 标头来编码/解码这些属性。

演示特定的 logging.properties 文件用来记录 messagesrelating 窥探到 WS-A 头和显示这些内容到控制台，在简洁 form.Normally 的 WS-Addressing 的 MessageAddressProperties 产生和传播。但是在某些情况下，应用程序可能希望参加 MAP 组合，例如，通过 RelatesTo 头来关联一序列的请求。这个演示说明了隐式和显式 MAP 的传播。

本案例程序也说明了脱钩 HTTP 传输的用法，即一个单独的服务器→客户端 HTTP 连接的方式分发响应。请注意正常的 HTTP 模式（如果响应是在后台基础客户端→服务器 HTTP 连接通道提供）当然可以在 WS-Addressing 使用过程中，在这种情况下，<wsa:ReplyTo>头被设置为一个众所周知的匿名 URI：“http://www.w3.org/2005/08/addressing/anonymous”。

在所有其他方面，本案例程序参考基本的 hello_world 例子，说明 WS-Addressing 的用法是与应用程序无关。本程序实现结构如图 13-1 所示。

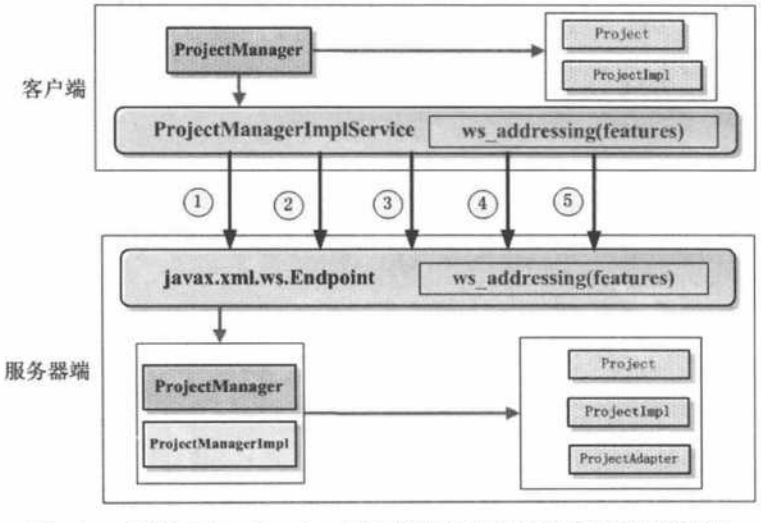


图 13-1 遵循 WS-Addressing 规范的客户端和服务器端调用模式图

其中：①表示客户端通过 Service 获得 SEI，访问后台的项目基本情况；②表示客户端通过 SEI 修改默认项目；③表示客户端通过 SEI 增加一个项目；④表示客户端通过 SEI 删除一个项目；⑤表示客户端通过 SEI 查看后台的全部项目情况。在客户端和服务端的 SEI 都是 ProjectManager 接口。

2. 应用程序案例实现步骤

创建 WS-Addressing 规范实现 Web Services 程序的流程步骤如图 13-2 所示。

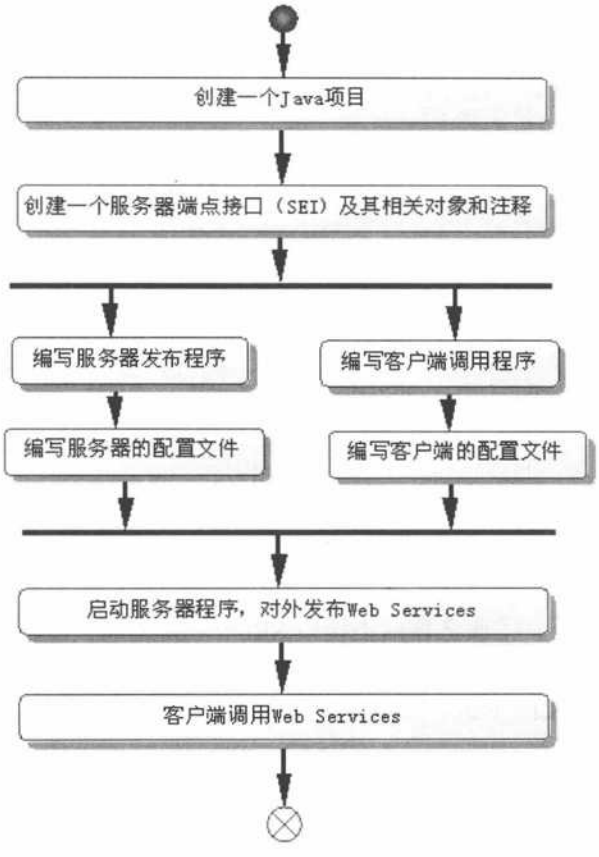


图 13-2　ws_addressing 应用程序案例实现步骤流程图

流程图说明，要实现以下步骤：

（1）创建一个 Java 项目。

（2）创建一个服务器端点接口（SEI）及其相关对象和注释。

（3）编写发布这个 Java 程序为 Web Services 的服务器端程序和服务器配置文件。

（4）编写 Web Services 的客户端应用程序和客户端配置文件。

（5）启动服务器程序，对外发布 Web Services。

（6）客户端调用 Web Services。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构

如图 13-3 所示。

包目录包括三个部分，第一个包目录 demo.cxf.ws.addressing.server 是服务器程序目录。第二个包目录 demo.cxf.ws.addressing.client 是客户端程序目录。第三个包目录 demo.cxf.webservices 是服务器端点接口 SEI 和相关对象等公用部分，服务器端和客户端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口，服务器端和客户端都必须引用。但服务器端和客户端对公用部分的引用是不同的。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口，而服务器端程序是接口和实现类都要引用。

图 13-3　ws_addressing 程序结构

下面分别介绍三部分：①服务器端点接口 SEI 和相关对象；②服务器端发布程序；③客户端调用程序。

（1）服务器端点接口（SEI）和相关对象

Project 组件包括 WebService 包下关于 Project 组件的几个文件。

首先要介绍的是服务器端点接口（SEI）。创建服务器端点接口与一般的开发类似，都是创建一个充当 SET 的 ProjectManager 接口和其实现类 ProjectManagerImpl。

SEI 文件 ProjectManager.java 代码如下：

```
package demo.cxf.webservices;
……

@WebService(targetNamespace = "http://webservices.cxf.demo/", name = "ProjectManager")
public interface ProjectManager {

public String greeting() ;
    ……
}
```

实现 SEI 的 ProjectManagerImpl.java，其代码如下：

```
package demo.cxf.webservices;
……
```

```
@javax.jws.WebService( serviceName = "ProjectManagerService",
       portName = "ProjectManagerPort",
       targetNamespace = "http://webservices.cxf.demo/")

public class ProjectManagerImpl implements ProjectManager {
    ……

    public ProjectManagerImpl() {
        System.out.println("----ProjectManagerImpl Construct----");
        initProject();
    }

    public String greeting() {
        System.out.println("----invoking greeting() ----");
        return "Hello, This is ProjectManager.";
    }
    ……
}
```

同时加入了一个继承 javax.xml.ws.Service 类的 ProjectManagerService 类。

```
package demo.cxf.webservices;

import java.net.MalformedURLException;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.WebEndpoint;
import javax.xml.ws.WebServiceClient;
import javax.xml.ws.WebServiceFeature;
import javax.xml.ws.Service;

@WebServiceClient(name = "ProjectManagerService",
              targetNamespace = "http://webservices.cxf.demo/")
public class ProjectManagerService extends Service {
   public final static URL WSDL_LOCATION;
   public final static QName SERVICE = new QName("http://webservices.cxf.demo/",
"ProjectManagerService");
public final static QName ProjectManagerPort =
new QName("http://webservices.cxf.demo/", "ProjectManagerPort");
   static {
      URL url = null;
      try {
         url = new URL("http://localhost:9000/ProjectManager?wsdl");
      } catch (MalformedURLException e) {
         System.err.println("Can not initialize the default wsdl ");
         // e.printStackTrace();
      }
      WSDL_LOCATION = url;
   }

   public ProjectManagerService(URL wsdlLocation) {super(wsdlLocation, SERVICE); }
   public  ProjectManagerService(URL  wsdlLocation,  QName  serviceName)  {super
(wsdlLocation, serviceName); }
   public ProjectManagerService() {super(WSDL_LOCATION, SERVICE); }

   @WebEndpoint(name = "ProjectManagerPort")
public ProjectManager getProjectManagerPort() {
return super.getPort(ProjectManagerPort, ProjectManager.class); }
```

```
    @WebEndpoint(name = "ProjectManagerPort")
    public ProjectManager getProjectManagerPort(WebServiceFeature... features) {
        return super.getPort(ProjectManagerPort, ProjectManager.class, features);
    }
}
```

（2）服务器端发布程序

编写发布这个 Java 程序为 Web Services 的服务器端程序。

创建了 SEI，要把这个 SEI 发布出去，还需要一个部署程序，即服务器端的发布程序，本案例采用 JAX-WS 规范来编写服务器端程序。创建一个 Server 类，启动这个服务，接着就可以发布 SEI 服务。Server.java 是服务器启动程序文件，其代码如下：

```
package demo.cxf.ws.addressing.server;

import java.net.URL;
import javax.xml.ws.Endpoint;
import org.apache.cxf.Bus;
import org.apache.cxf.bus.spring.SpringBusFactory;
import demo.cxf.webservices.*;

public class Server {
    public Server() {
        System.out.println("Starting Server");
        SpringBusFactory bf = new SpringBusFactory();
       URL busFile = Server.class.getResource("server.xml");
       Bus bus = bf.createBus(busFile.toString());
       bf.setDefaultBus(bus);

       ProjectManager projectManager = new ProjectManagerImpl();
       String address = "http://localhost:9000/ProjectManager";

        Endpoint.publish(address, projectManager);
    }

    public static void main(String[] args) throws Exception {
        new Server();
        System.out.println("Server ready...");

        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

在代码中要求装载文件 server.xml，配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:cxf="http://cxf.apache.org/core"
    xmlns:wsa="http://cxf.apache.org/ws/addressing"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://cxf.apache.org/core
        http://cxf.apache.org/schemas/core.xsd
        http://www.springframework.org/schema/beans
```

```
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <cxf:bus>
        <cxf:features>
            <wsa:addressing />
        </cxf:features>
    </cxf:bus>
</beans>
```

在本案例中，服务器启动的服务地址是 http://localhost:9000/ProjectManager。服务器引入了 SEI 接口——ProjectManager 接口。这样 ProjectManager 的 Web Services 就暴露到了外部。可以在浏览器上键入 http://localhost:9000/ProjectManager?wsdl 来查看这个服务的内容。

（3）客户端调用程序

客户端应用程序包括 Web Services 的客户端应用程序和配置文件。

Web Services 的客户端应用程序——Client.java 的代码如下：

```
package demo.cxf.ws.addressing.client;

import java.io.File;
import java.net.URL;
import java.util.Map;
import javax.xml.namespace.QName;
import javax.xml.ws.BindingProvider;

import org.apache.cxf.Bus;
import org.apache.cxf.bus.spring.SpringBusFactory;
import org.apache.cxf.ws.addressing.AddressingBuilder;
import org.apache.cxf.ws.addressing.AddressingProperties;
import org.apache.cxf.ws.addressing.AttributedURIType;
import org.apache.cxf.ws.addressing.ObjectFactory;

import static org.apache.cxf.ws.addressing.JAXWSAConstants.CLIENT_ADDRESSING_PROPERTIES;

import demo.cxf.webservices.Project;
import demo.cxf.webservices.ProjectImpl;
import demo.cxf.webservices.ProjectManager;
import demo.cxf.webservices.ProjectManagerService;

public class Client {
    private static final QName SERVICE_NAME = new QName(
            "http://webservices.cxf.demo/", "ProjectManagerService");

    private static final ObjectFactory WSA_OBJECT_FACTORY = new ObjectFactory();
    private static final String WSDLFile = "http://localhost:9000/ProjectManager?wsdl";
    private static final String USER_NAME = System.getProperty("user.name");
    public Client() {   }

    public static void main(String[] args) throws Exception {
        System.out.println("Starting Client");
        URL wsdlURL;

        if (args.length == 0) {
            wsdlURL = new URL(WSDLFile);
        } else {
            File wsdlFile = new File(args[0]);
```

```
            if (wsdlFile.exists()) {
                wsdlURL = wsdlFile.toURL();
            } else {
                wsdlURL = new URL(args[0]);
            }
        }

        SpringBusFactory bf = new SpringBusFactory();
        URL busFile = Client.class.getResource("client.xml");
        Bus bus = bf.createBus(busFile.toString());
        bf.setDefaultBus(bus);

        // 用服务名称创建一个服务
        ProjectManagerService service = new ProjectManagerService(wsdlURL,SERVICE_
NAME);

        // 获得 SEI 接口
        ProjectManager projectManager = service.getPort(ProjectManager.class);
        implicitPropagation(projectManager);
        explicitPropagation(projectManager);
        implicitPropagation(projectManager);
        System.exit(0);
    }

    //通过内在的 MessageAddressingProperties 进行一系列的 Web Services 调用
    private static void implicitPropagation(ProjectManager projectManager)
            throws Exception {
        System.out.println();
        System.out.println("Implicit MessageAddressingProperties propagation");
        System.out.println("-----------------------------------------------");

        // 项目的基本情况
        getGreet(projectManager);
        greetingWithParameter(projectManager);
        getProjectName(projectManager);
        getDefaultProject(projectManager);
        getAllProject(projectManager);

        // 修改默认项目，查看项目的基本情况
        setDefaultProject(projectManager);

        // 增加一个项目，查看项目的基本情况
        registerProject(projectManager);

        // 删除一个项目，查看项目的基本情况
        removeProject(projectManager);

    }

    //通过外在的 MessageAddressingProperties 进行一系列的 Web Services 调用
    private static void explicitPropagation(ProjectManager projectManager)
            throws Exception {
        System.out.println();
        System.out.println("Explicit MessageAddressingProperties propagation");
        System.out.println("-----------------------------------------------");

        // get Message Addressing Properties instance
        AddressingBuilder builder = AddressingBuilder.getAddressingBuilder();
```

```
        AddressingProperties maps = builder.newAddressingProperties();
        maps = setMapMessageIDProperty(maps);

        // associate MAPs with request context
        Map<String, Object> requestContext = ((BindingProvider) projectManager)
                .getRequestContext();

        // 项目的基本情况
        requestContext.put(CLIENT_ADDRESSING_PROPERTIES, maps);
        getGreet(projectManager);
        requestContext.remove(CLIENT_ADDRESSING_PROPERTIES);

        AddressingProperties maps1 = setMapMessageIDProperty(builder.newAddressingProperties());

        requestContext.put(CLIENT_ADDRESSING_PROPERTIES, maps1);
        greetingWithParameter(projectManager);
        requestContext.remove(CLIENT_ADDRESSING_PROPERTIES);

        AddressingProperties maps2 = setMapMessageIDProperty(builder.
newAddressingProperties());

        requestContext.put(CLIENT_ADDRESSING_PROPERTIES, maps2);
        getProjectName(projectManager);
        requestContext.remove(CLIENT_ADDRESSING_PROPERTIES);

        AddressingProperties maps3 = setMapMessageIDProperty(builder.
newAddressingProperties());

        requestContext.put(CLIENT_ADDRESSING_PROPERTIES, maps3);
        getDefaultProject(projectManager);
        requestContext.remove(CLIENT_ADDRESSING_PROPERTIES);

        AddressingProperties maps4 = setMapMessageIDProperty(builder.
newAddressingProperties());

        requestContext.put(CLIENT_ADDRESSING_PROPERTIES, maps4);
        getAllProject(projectManager);
        requestContext.remove(CLIENT_ADDRESSING_PROPERTIES);
    }

    // 对 AddressingProperties 加入 AttributedURIType 属性
    private static AddressingProperties setMapMessageIDProperty(
            AddressingProperties maps) {
        // set MessageID property
        AttributedURIType messageID = WSA_OBJECT_FACTORY
                .createAttributedURIType();
        messageID.setValue("urn:uuid:" + System.currentTimeMillis());
        maps.setMessageID(messageID);
        return maps;
    }

    // 调用 SEI 的 getGreet 方法
    private static void getGreet(ProjectManager projectManager)
            throws Exception {
        System.out.println(projectManager.greeting());
    }
    ……

}
```

配置文件 client.xml 的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:cxf="http://cxf.apache.org/core"
    xmlns:wsa="http://cxf.apache.org/ws/addressing"
    xmlns:http="http://cxf.apache.org/transports/http/configuration"
    xsi:schemaLocation="
        http://cxf.apache.org/core
        http://cxf.apache.org/schemas/core.xsd
        http://cxf.apache.org/transports/http/configuration
        http://cxf.apache.org/schemas/configuration/http-conf.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <http:conduit
        name="{http://webservices.cxf.demo/}ProjectManagerPort.http-conduit">
        <http:client
            DecoupledEndpoint="http://localhost:9000/ProjectManager" />
    </http:conduit>

    <cxf:bus>
        <cxf:features>
            <wsa:addressing />
        </cxf:features>
    </cxf:bus>
</beans>
```

本例首先创建一个服务名称为 javax.xml.ws.Service 的实现类，接着通过 addPort 方法来传入服务的端口类、绑定方式和 Web Services 的地址，最后通过 getPort 方法来获得服务器的 SEI 接口。这就是 Web Services 服务接口 ProjectManager。后面基于这个 ProjectManager 的 SEI 来实现各种业务操作。

4. 测试和运行结果

测试编写好的程序代码，首先要启动服务器程序，即在 Eclipse 上运行 Server.java。然后运行客户端程序，其输出结果如下：

```
Starting Client
Implicit MessageAddressingProperties propagation
-----------------------------------------------
Hello, This is ProjectManager.
Hello, greet This is ProjectManager.
default projectName
defaultProjectID is default projectID . defaultProjectName is default projectName
DefaultProjectID is New Default ProjectID . DefaultProjectName is New Default
ProjectName
New ProjectName
New ProjectName

Explicit MessageAddressingProperties propagation
-----------------------------------------------
Hello, This is ProjectManager.
Hello, greet This is ProjectManager.
New Default ProjectName
```

```
defaultProjectID is New Default ProjectID . defaultProjectName is New Default
ProjectName

Implicit MessageAddressingProperties propagation
-----------------------------------------------
Hello, This is ProjectManager.
Hello, greet This is ProjectManager.
New Default ProjectName
defaultProjectID is New Default ProjectID . defaultProjectName is New Default
ProjectName
DefaultProjectID is New Default ProjectID . DefaultProjectName is New Default
ProjectName
New ProjectName
New ProjectName
```

5. 其他内容

上述的所有调用都基于 JAX-WS 规范。

本例代码参见配套资源代码中“第 13 章\ws_addressing”文件夹。

13.2　Apache CXF 支持 WS-Policy

Web 服务策略框架规范（Web Services Policy Framework，WS-Policy）提供了一种灵活、可扩展的语法，用于表示基于 XML Web Services 的系统中实体能力、要求和一般特性。很可能大部分 Web 服务程序员都会在不同程度上受到 Web 服务策略框架规范（WS-Policy）的影响。WS-Policy 规范现在已成为 Web Services 框架的重要组件。

本节简单介绍一下 WS-Policy 规范概念，以及 Apache CXF 框架使用 WS-Policy 规范的方法和例子。

13.2.1　WS-Policy 简介

WS-Policy 规范定义了一个框架和一个模型，并将这些特性理解为策略。策略表示法既支持简单的声明式断言，也支持比较复杂的条件式断言。策略断言表示行为的个体要求、能力或其他特性。例如，可以创建一个策略断言，定义这样一个要求：必须使用受信任的 X.509 证书签名 SOAP 消息主体。WS-Policy 规范同时也只是一个框架，为特定域定义一套特定的策略断言是该系统的任务。

WS-Policy 规范为服务请求者和服务提供者定义了语法和语义来描述业务的需求、首选项和性能。语法以策略的形式表述每个领域的需求并提供了一种灵活简洁的方法。领域在本文中指的是关系到服务重要性的通用描述，如下所示：

- 安全
- 隐私
- 应用程序优先权
- 用户账户优先权
- 传输控制

WS-Policy 规范还描述了这些策略的处理模型，处理模型是独立于领域的。处理策略定义了

三种操作：标准化、融合和交集。

下面主要介绍 WS-Policy 规范的这些术语。

断言（Assertion）

断言（Assertion）是策略（Policy）的基本组成单元，每个断言都可以通过限定名（QName）来识别。断言既可以是一个简单的字符串也可以是带多个子元素和属性的复杂对象。断言表示服务行为的个性化要求、能力或其他特性。对于策略处理基础架构来说，可以将其看作一条指令。例如，断言可以声明消息是被加密的。这种断言的实际定义将包含在 WS-Security 策略域规范中。

因此，特定于某个领域的断言，其意义是不同的，但可以通过断言单独的规范来区别和识别。各特定领域断言的详细信息都非常多。但 WS-Policy 规范将每个断言看作 opaque。例如，可以创建一个断言，定义这样一个安全要求：必须使用受信任的 X.509 证书签名 SOAP 消息主体。

每个断言可以通过限定名来识别。断言可以是一个简单的字符串或带多个子元素和属性的复杂对象。但是，任何 WS-Policy 规范支持的框架处理中只涉及 XML 的根元素 QName。

替代（Alternative）

替代（Alternative）是针对断言的另一种表现形式，或者说是断言的组合形式。策略是由断言和操作符 <wsp:All>、<wsp:ExactlyOne> 的嵌套组合以及可选属性构成的。对于给定的 Web Services 调用，利用该策略语法来描述可接受的断言组合，最终形成策略处理基础架构的完整指令集。每组断言集在术语上称为替代（Alternative）。

注意，只有包含可选属性或 <wsp:ExactlyOne> 操作符的策略才有多个替代。换句话说，只包含一个断言和 <wsp:All> 操作符的策略可以解析为一个替代。

下面部分涉及到可以应用到策略上的处理类型。

标准化（Normalization）：灵活的 WS-Policy 规范支持框架语法可能会产生潜在的复杂策略。定义策略的标准格式以简化策略的操作和阐明对其的理解。正是基于标准化格式（Normalization）策略来执行策略处理的基本操作。标准化的目的是为了进行融合（Merge）和交集（Intersection）的操作，把策略简化为标准格式。

融合（Merge）：融合是将多个子策略组合在一起形成一个单一策略的过程。

交集（Intersection）:交集是用于替代两个 WS-Policy 公用部分的过程。只有双方至少对一个策略替代达成一致，才会产生交集。

在 Web Services 标准化规范中，体现这些约束的标准就只是 WS-Policy 规范，而其他的规范如 WS-Addressing、WS-RM、WS-Security 都可以看作 WS-Policy 规范的特例。在 SOA 的实践中提出了新的 Service Governance（服务管制）概念，服务管制就包含了对 Policy 的管理。Policy 管理包含两部分内容：一是要在设计和运行时规定服务策略（Service Policy）的定义、交换和强制执行的方式和方法；二是要有基于标准（比如 WS-*）的组织内部规章。

13.2.2 Apache CXF 使用 WS-Policy 框架

Apache CXF 框架的 WS-Policy 框架提供了一个允许 Apache CXF 用户和开发者使用 WS-policy 应用架构和调用框架的 API。Apache CXF 的 WS-Policy 框架由两部分组成，第一部分是一个核心运行环境（Core），第二部分是允许开发人员插入自定义断言的 API。

13.2.3　Apache CXF 的 WS-Policy 的实现例子

本程序演示 Apache CXF 框架遵循 WS-Policy 规范的客户端和服务器端调用，客户端采用 Apache CXF 的代码优先（Java First）前端应用模式。采用 POJO 对象作为中间传输对象。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，都使用 POJO 对象实现客户端和服务器端的数据交换。后台服务器端为遵循 JAX-WS 规范的 Endpoint 来实现，采用 Spring IoC 注入的客户端来实现。前端访问程序采用遵循 JAX-WS 规范的 Service 方式来实现。本程序实现的结构如图 13-4 所示。

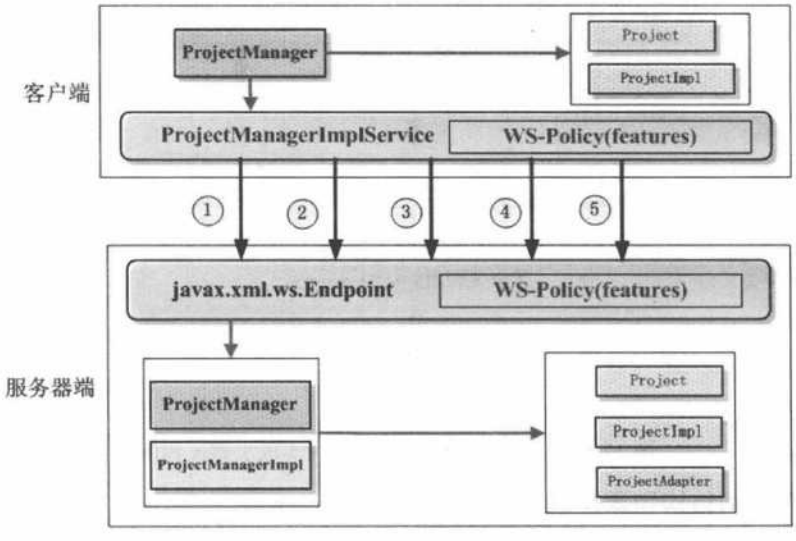


图 13-4　遵循 WS-Policy 规范的客户端和服务器端调用模式图

其中：① 表示客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。在客户端和服务端的 SEI 都是 ProjectManager 接口。

2. 应用程序案例实现步骤

要创建 WS-Policy 实现 Web Services 程序的流程步骤如图 13-5 所示。

流程图说明，要实现以下步骤：

（1）创建一个 Java 项目。

（2）创建一个服务器端点接口（SEI）及其相关对象和注释。

（3）编写发布这个 Java 程序为 Web Services 的服务器端程序和服务器配置文件。

（4）编写 Web Services 的客户端应用程序和客户端配置文件。

（5）启动服务器程序，对外发布 Web Services。

（6）客户端调用 Web Services。

图 13-5　ws_policy 应用程序案例实现步骤流程图

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 13-6 所示。

图 13-6　ws_policy 程序结构

包目录包括三个部分，第一个包目录 demo.cxf.ws.policy.server 是服务器程序目录。第二个包目录 demo.cxf.ws.policy.client 是客户端程序目录。第三个包目录 demo.cxf.webservices 是服务器端点接口 SEI 和相关对象等公用部分，服务器端和客户端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口，服务器端和客户端都必须引用。但服务器端和客户端对公用部分的引用是不同的。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口，而服务器端程序是接口和实现类都要引用。

下面分别介绍三部分：①服务器端点接口 SEI 和相关对象；②服务器端发布程序；③客户端调用程序。

（1）服务器端点接口（SEI）和相关对象

首先要介绍的是服务器端点接口（SEI）。Project 组件包括这几个 WebService 包下的 Project 几个文件和 Server.java 文件。

创建服务器端点接口与 WS-Addressing 实现例子相似，都创建一个充当 SEI 的 ProjectManager 接口和其实现类 ProjectManagerImpl。本案例的 ProjectManager 接口和其实现类 ProjectManagerImpl 与 WS-Addressing 实现例子的 ProjectManager 接口和 ProjectManagerImpl 相同。同时加入了一个继承 javax.xml.ws.Service 类的 ProjectManagerService 类，ProjectManagerService 类与 WS-Addressing 实现例子的 ProjectManagerService 类基本相同。

（2）服务器端发布程序

编写发布这个 Java 程序为 Web Services 的服务器端程序。创建了 SEI，要把这个 SEI 发布出去，还需要一个部署程序。即服务器端的发布程序，本案例采用 JAX-WS 规范来编写服务器端程序。创建一个 Server 类，启动这个服务，接着就可以发布 SEI 服务。

Server.java 是服务器启动程序文件，其代码如下：

```
package demo.cxf.ws.policy.server;

import java.net.URL;
import javax.xml.ws.Endpoint;

import org.apache.cxf.Bus;
import org.apache.cxf.bus.spring.SpringBusFactory;

import demo.cxf.webservices.*;

public class Server {

    public Server() {
        System.out.println("Starting Server");
        SpringBusFactory bf = new SpringBusFactory();
       URL busFile = Server.class.getResource("server.xml");
       Bus bus = bf.createBus(busFile.toString());
       bf.setDefaultBus(bus);

       ProjectManager projectManager = new ProjectManagerImpl();
       String address = "http://localhost:9000/ProjectManager";

        Endpoint.publish(address, projectManager);
    }

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

```
        new Server();
        System.out.println("Server ready...");

        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

在代码中要求装载文件 server.xml，配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:cxf="http://cxf.apache.org/core"
    xmlns:p="http://cxf.apache.org/policy"
    xsi:schemaLocation="
          http://cxf.apache.org/core
          http://cxf.apache.org/schemas/core.xsd
          http://cxf.apache.org/policy
          http://cxf.apache.org/schemas/policy.xsd
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd">

    <cxf:bus>
        <cxf:features>
            <p:policies />
        </cxf:features>
    </cxf:bus>

    <p:externalAttachment location="addr-external.xml" />
</beans>
```

引入的外部文件——addr-external.xml 的代码如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<attachments xmlns:wsp="http://www.w3.org/ns/ws-policy"
    xmlns:wsa="http://www.w3.org/2005/08/addressing">
    <wsp:PolicyAttachment>
        <wsp:AppliesTo>
            <wsa:EndpointReference>
                <wsa:Address>
                    http://localhost:9000/SoapContext/SoapPort
                </wsa:Address>
            </wsa:EndpointReference>
        </wsp:AppliesTo>
        <wsp:Policy>
            <wsam:Addressing
                xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
                <wsp:Policy />
            </wsam:Addressing>
        </wsp:Policy>
    </wsp:PolicyAttachment>
</attachments>
```

在本案例中，服务器启动的服务地址是 http://localhost:9000/ProjectManager。服务器引入了

SEI 接口——ProjectManager 接口。这样 ProjectManager 的 Web Services 就暴露到了外部。可以在浏览器上输入 http://localhost:9000/ProjectManager?wsdl 来查看这个服务的内容。

（3）客户端调用程序

客户端应用程序包括 Web Services 的客户端应用程序和配置文件。

Web Services 的客户端应用程序——Client.java 的代码如下：

```
package demo.cxf.ws.policy.client;

import java.net.URL;

import org.apache.cxf.Bus;
import org.apache.cxf.bus.spring.SpringBusFactory;
……

public class Client {

    public Client() {}

    public static void main(String[] args) throws Exception {
        System.out.println("Starting Client");

        SpringBusFactory bf = new SpringBusFactory();
        URL busFile = Client.class.getResource("client.xml");
        Bus bus = bf.createBus(busFile.toString());
        bf.setDefaultBus(bus);

        ProjectManagerService service = new ProjectManagerService();
        ProjectManager projectManager = service.getProjectManagerPort();

        // 项目的基本情况
        getGreet(projectManager);
        ……
        System.exit(0);

    }

    // 调用 SEI 的 getGreet 方法
    private static void getGreet(ProjectManager projectManager)
            throws Exception {
        System.out.println(projectManager.greeting());
    }

    ……

}
```

配置文件 client.xml 的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:cxf="http://cxf.apache.org/core"
    xmlns:p="http://cxf.apache.org/policy"
    xmlns:http="http://cxf.apache.org/transports/http/configuration"
    xsi:schemaLocation="
```

```
        http://cxf.apache.org/transports/http/configuration
        http://cxf.apache.org/schemas/configuration/http-conf.xsd
        http://cxf.apache.org/core
        http://cxf.apache.org/schemas/core.xsd
        http://cxf.apache.org/policy
        http://cxf.apache.org/schemas/policy.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <http:conduit
        name="{http://webservices.cxf.demo/}ProjectManagerPort.http-conduit">
        <http:client
            DecoupledEndpoint="http://localhost:9990/decoupled_endpoint" />
    </http:conduit>

    <cxf:bus>
        <cxf:features>
            <p:policies />
        </cxf:features>
    </cxf:bus>

</beans>
```

本例首先创建一个服务名称为 javax.xml.ws.Service 的实现类，接着通过 addPort 方法来传入服务的端口类、绑定方式和 Web Services 的地址。最后通过 getPort 方法来获得服务器的 SEI 接口。这就是 Web Services 服务接口 ProjectManager。后面就基于这个 ProjectManager 的 SEI 来实现各种业务操作。

4. 测试和运行结果

测试编写好的程序代码，首先要启动服务器程序，即在 Eclipse 上运行 Server.java，然后运行客户端程序，其输出结果如下：

```
Starting Client
Hello, This is ProjectManager.
Hello, greet This is ProjectManager.
default projectName
defaultProjectID is default projectID . defaultProjectName is default projectName
DefaultProjectID is New Default ProjectID . DefaultProjectName is New Default
ProjectName
New ProjectName
New ProjectName
```

5. 其他内容

上述的所有调用都基于 JAX-WS 规范。

本例代码参见配套资源代码中“第 13 章\ws_policy”文件夹。

13.3 Apache CXF 支持 WS-ReliableMessaging

本节简单介绍一下 WS-ReliableMessaging 规范概念，以及 Apache CXF 框架使用 WS-ReliableMessaging 规范的方法和例子。

13.3.1　WS-ReliableMessaging 简介

WS-ReliableMessaging 规范（有时也简化为 WS-RM 规范）定义了一种协议和一套机制，使 Web Services 的开发者能够确保在两个端点之间可靠地传递消息，该规范还具有各种传递保证和健壮性特征。该规范由 BEA、IBM、Microsoft 和 Tibco 联合发布。

WS-ReliableMessaging 规范把 Sequence 定义为一组消息的共享上下文，这组消息在源应用程序和目标应用程序之间进行交换，并有相同的传递保证要求。协议在源端点的实现至少必须能够在 Sequence 中唯一地标识并重发消息，直到目标端点确认接收到所有消息。目标端点上的实现必须能够在 Sequence 中跟踪接收到的消息并传递确认消息给源端点。

WS-ReliableMessaging 规范模型如图 13-7 所示。

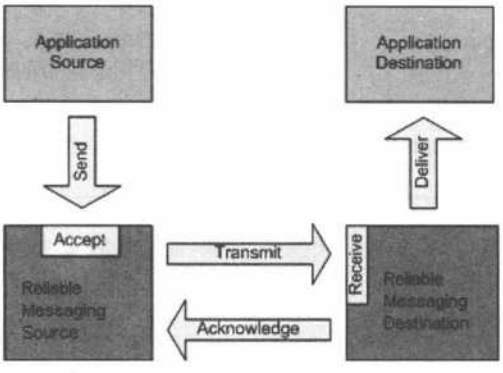


图 13-7　WS-ReliableMessaging 规范模型

为了实现性能、健壮性和完整性这些目标，采用了一种经证实并被广泛接受的模式。该模式在每个通信端点上引入将被称为端点管理器的消息传递中间件组件。这些端点管理器可以接受来自一个或多个应用程序的消息传递请求，为消息到达目的地提供可靠的传递，并将接收到的消息传递到合适的应用程序。

端点管理器能够简化管理消息和 Sequence 状态持久性的复杂程度。它可以与其他的资源管理器一起参与资源管理活动以实现更好的健壮性和完整性特征，同时降低应用程序代码的复杂性。端点管理器可以作为应用程序软件本身的一个组件（例如，作为专门的 JAX-RPC 处理程序）来实现。或者，也可以作为通过远程 API（如 JMS）来访问的一个分布式组件来实现。

WS-ReliableMessaging 协议定义和支持了集中递送保障，包括：

- 至少一次（AtLeastOnce）——每一条消息将被递送到目标应用至少一次。如果一条消息无法被递送，可靠消息源或可靠消息目标至少有一个应当报错。例如，目标应用可能得到重复的信息。
- 至多一次（AtMostOnce）——每一条消息将被递送到目标应用不超过一次。消息可以没有被递送到目标应用，但目标应用决不能收到重复的消息。
- 正好一次（ExactlyOnce）——每一条消息将被递送到应用一次且仅一次。如果消息无法被递送，可靠消息源或可靠消息目标至少有一个应当报错。目标应用将不会受到重复的消息。
- 按序（InOrder）——消息将按照从源应用发送到可靠消息源的顺序，从可靠消息目标被递送到目标应用。这一保障包括了以上所有保障。

WS-ReliableMessaging 规范的定义见 http://specs.xmlsoap.org/ws/2005/02/rm/ws-reliablemessaging.pdf。其他相关资料可以参看网站：http://schemas.xmlsoap.org/ws/2005/02/rm/。

13.3.2 Apache CXF 使用 WS-ReliableMessaging 的配置

Apache CXF 基于拦截器来实现 WS-ReliableMessaging 规范。Apache CXF 实现 WS-ReliableMessaging 规范的拦截器由四个组成，如表 13-2 所示。

表 13-2 WS-ReliableMessaging规范的四个拦截器功能

拦截器	功能
org.apache.cxf.ws.rm.RMOutInterceptor	负责发送CreateSequence请求和等待CreateSequenceResponse响应，并为应用程序消息来聚集这序列属性（ID和消息号）
org.apache.cxf.ws.rm.RMInInterceptor	拦截和处理RM协议消息（这本不是应用程序级的工作），以及应用程序消息附带的SequenceAcknowledgments
org.apache.cxf.ws.rm.soap.RMSoapInterceptor	编码和解码RM头
org.apache.cxf.ws.rm.soap.RetransmissionInterceptor	负责创造一个应用消息的副本，该副本为以后重新发送提供信息

为了使两个端点之间实现可靠的消息交换，需要确保 RM 拦截器和 WS-Reliable 消息拦截器被添加到 Apache CXF 框架的拦截链中。Apache CXF 框架实现 WS-ReliableMessaging 规范有两种方式：

- 使用 RMAssertion 和 Apache CXF 框架的 WS-Policy 框架。
- 用可靠性消息功能（Reliable Messaging Feature）。

13.3.2.1 使用 RMAssertion 和 CXF 的 WS-Policy 的框架

当 RMAssertion 元素附加到 wsdl:service 元素（或任何其他 WSDL 元素，根据 WS-Policy 规则，这是策略或 PolicyReference 元素的分子附着点），同时 Apache CXF 框架的 WS-Policy 框架已经启用，RM 拦截器将被自动添加到它们各自的策略框架拦截链。断言属性将控制来源和目的地的行为。

例如，在服务器端启用了 WS-Policy 框架，实现代码在配置文件中如下所示：

```
<jaxws:endpoint ...>
   <jaxws:features>
      <p:policies/>
   </jaxws:features>
</jaxws:endpoint>
```

或者 WSDL 的配置代码如下所示：

```
<wsp:Policy wsu:Id="RM" xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
utility-1.0.xsd">
   <wsam:Addressing xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
      <wsp:Policy/>
   </wsam:Addressing>
   <wsrmp:RMAssertion
```

```
xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
        <wsrmp:BaseRetransmissionInterval Milliseconds="10000"/>
    </wsrmp:RMAssertion>
</wsp:Policy>
...
<wsdl:service name="ReliableGreeterService">
    <wsdl:port binding="tns:GreeterSOAPBinding" name="GreeterPort">
        <soap:address location="http://localhost:9020/SoapContext/GreeterPort"/>
        <wsp:PolicyReference URI="#RM" xmlns:wsp="http://www.w3.org/2006/07/ ws-policy"/>
    </wsdl:port>
</wsdl:service>
```

当然，也可以在服务器端点上指定 Features 子元素，做如下的配置：

```
<wsp:Policy wsu:Id="="RM" xmlns:wsp="http://www.w3.org/2006/07/ws-policy" ...>
</wsp:Policy>

<jaxws:endpoint ...>
    <jaxws:features>
    <p:policies>
        <wsp:PolicyReference URI="#RM" xmlns:wsp="http://www.w3.org/2006/07/ws-policy"/>
    </p:policies>
</jaxws:features>
```

13.3.2.2　用可靠性消息功能（Reliable Messaging Feature）

如果不使用 WS-Policy 框架，或想要做更多的配置，如序列终止策略或其他参数的持久化存储，也可以使用 ReliableMessaging 功能。ReliableMessaging 功能在命名空间 http://cxf.apache.org/ws/rm/manager（包括 http:// cxf.apache.org/schemas/wsrm-manager.xsd 和 http://cxf.apache.org/schemas/wsrm-manager-types.xsd）进行定义并支持以下子元素，如表 13-3 所示。

表 13-3　ReliableMessaging支持的子元素

名称	数值
RMAssertion	RMAssertion类型元素
deliveryAssurance	DeliveryAssuranceType类型元素描述应用有所保证
sourcePolicy	SourcePolicyType类型，允许配置RM源的细节
destinationPolicy	一个DestinationPolicyType类型允许配置RM目的地的细节
store	存储是否使用（默认值：null）。这必须是一个jdbcStore类型（在同一命名空间），或一个bean或对一个实现RMStore接口的引用

例如，在 Bus 一级的应用功能，配置文件内容如下：

```
<cxf:bus>
    <cxf:features>
        <wsa:addressing/>
        <wsrm-mgr:reliableMessaging>
            <wsrm-policy:RMAssertion>
                <wsrm-policy:BaseRetransmissionInterval Milliseconds="4000"/>
                <wsrm-policy:AcknowledgementInterval Milliseconds="2000"/>
            </wsrm-policy:RMAssertion>
            <wsrm-mgr:sourcePolicy>
```

```
            <wsrm-mgr:sequenceTerminationPolicy maxLength="5"/>
        </wsrm-mgr:sourcePolicy>
        <wsrm-mgr:destinationPolicy acceptOffers="false">
        <wsrm:store>
          <ref bean="myStore"/>
        </wsrm:store>
      </wsrm-mgr:reliableMessaging>
   </cxf:features>
</cxf:bus>
```

1. 配置可靠消息存储体

为了使用持久性，开发者必须指定实施对象的 RM 持久性存储体。当然开发者也可以开发自己的持久性存储体，或者使用 Apache CXF 基于 JDBC 的存储体（org.apache.cxf.ws.rm.persistence.jdbc. RMTxStore）。还可以配置使用自定义 jdbcStore 类（命名空间 http://cxf.apache.org/ws/rm/manager）。jdbcStore 类支持的属性如表 13-4 所示。

表 13-4　jdbcStore类支持的属性、数据类型和默认值

属性名称	数据类型	默认值
driverClassName	String	org.apache.derby.jdbc.EmbeddedDriver
userName	String	null
passWord	String	null
url	String	jdbc:derby:rmdb;create=true

代码例子如下：

```
<wsrm-mgr:jdbcStore id="myStore"
   driverClassName="org.apache.derby.jdbc.ClientDriver"/>
   url="jdbc:derby://localhost:1527/rmdb;create=true"/>
   password="password"/>
```

2. 手动添加可靠消息拦截器

为确保可靠的消息拦截器添加到适当的拦截器链，需要对所有客户端和服务端点做手工处理的配置代码如下：

```
<bean id="rmLogicalOut" class="org.apache.cxf.ws.rm.RMOutInterceptor">
      <property name="bus" ref="cxf"/>
</bean>

 <bean id="rmLogicalIn" class="org.apache.cxf.ws.rm.RMInInterceptor">
      <property name="bus" ref="cxf"/>
</bean>

 <bean id="rmCodec" class="org.apache.cxf.ws.rm.soap.RMSoapInterceptor"/>

   <cxf:bus ...>
      <cxf:inInterceptors>
         <ref bean="rmLogicalIn"/>
         <ref bean="rmCodec"/>
      </cxf:inInterceptors>
      <cxf:inFaultInterceptors>
```

```
        <ref bean="rmLogicalIn"/>
        <ref bean="rmCodec"/>
    </cxf:inFaultInterceptors>
    <cxf:outInterceptors>
        <ref bean="rmLogicalOut"/>
        <ref bean="rmCodec"/>
    </cxf:outInterceptors>
    <cxf:outFaultInterceptors>
        <ref bean="rmLogicalOut"/>
        <ref bean="rmCodec"/>
    <cxf:outFaultInterceptors>
</cxf:bus>
```

RetransmissionIntercepor 通过 RMOutInterceptor 会自动加入，所以不必出现在配置文件中。

3. 手工地配置可靠消息传送管理器（Reliable Messaging Manager）

要配置 RM 管理器属性，可以使用 rmManager 元素（参考 http://cxf.apache.org/ws/rm/manager 命名空间）。该元素支持作为 ReliableMessaging 功能元素下相同的子元素。例如，如果不使用这个功能，可以指定 sequence 最大长度为 5，配置信息内容如下：

```
<wsrm-mgr:rmManager xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager">
    <wsrm-mgr:sourcePolicy>
        <wsrm-mgr:sequenceTerminationPolicy maxLength="5"/>
    </wsrm-mgr:sourcePolicy>
</wsrm-mgr:rmManager>
```

13.3.3　Apache CXF 的 WS-ReliableMessaging 的实现例子

本程序演示 Apache CXF 框架遵循 WS-ReliableMessaging 规范的客户端和服务器端调用，客户端采用 Apache CXF 框架的代码优先（Java First）前端应用模式。采用 POJO 对象作为中间传输对象。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，都使用 POJO 对象实现客户端和服务器端的数据交换。后台服务端为遵循 JAX-WS 规范的 Endpoint 来实现，采用 Spring IoC 注入的客户端来实现。前端访问程序采用 JAX-WS 规范的 Service 方式来实现。本程序实现的结构如图 13-8 所示。

其中：① 表示客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。在客户端和服务端的 SEI 都是 ProjectManager 接口。

2. 应用程序案例实现步骤

创建 WS-ReliableMessaging 实现 Web Services 程序的流程步骤如图 13-9 所示。

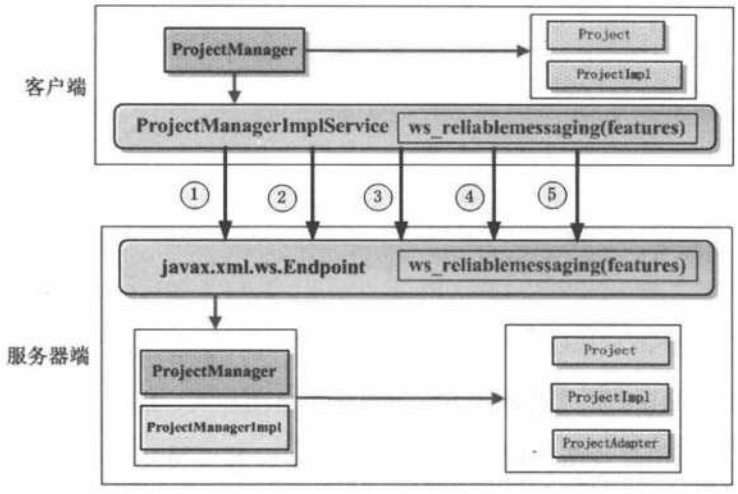


图 13-8　遵循 WS-ReliableMessaging 规范的客户端和服务器端调用模式图

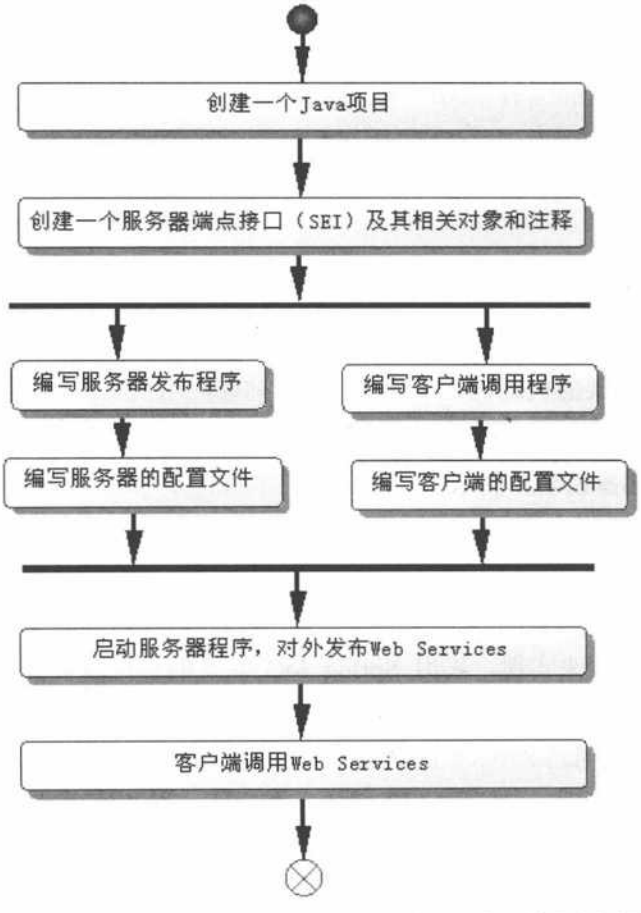


图 13-9　ws_reliablemessaging 应用程序案例实现步骤流程图

流程图说明，要实现以下步骤：

（1）创建一个 Java 项目。

（2）创建一个服务器端点接口（SEI）及其相关对象和注释。

（3）编写发布这个 Java 程序为 Web Services 的服务器端程序和服务器配置文件。

（4）编写 Web Services 的客户端应用程序和客户端配置文件。

（5）启动服务器程序，对外发布 Web Services。

（6）客户端调用 Web Services。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 13-10 所示。

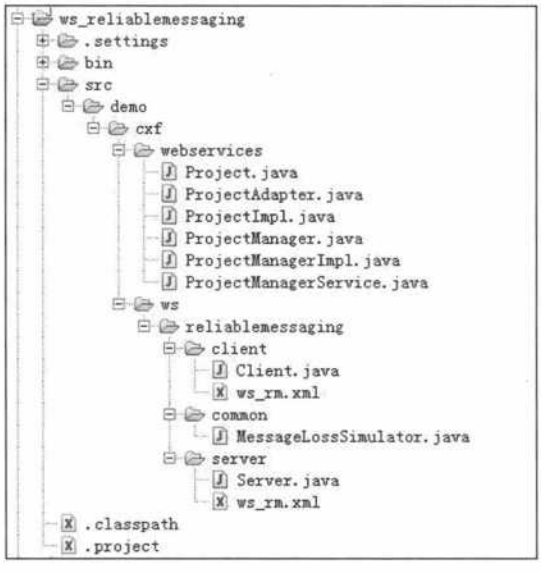


图 13-10　ws_reliablemessaging 程序结构

包目录包括三个部分，第一个包目录 demo.cxf.ws.reliablemessaging.server 是服务器程序目录。第二个包录目 demo.cxf.ws.reliablemessaging.client 是客户端程序目录。第三个包目录 demo.cxf.webservices 是服务器端点接口 SEI 和相关对象等公用部分，服务器端和客户端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口，服务器端和客户端都必须引用。但服务器端和客户端对公用部分的引用是不同的。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口，而服务器端程序是接口和实现类都要引用。

下面分别介绍三部分：①服务器端点接口 SEI 和相关对象；②服务器端发布程序；③客户端调用程序。

（1）服务器端点接口（SEI）和相关对象

首先要介绍的是服务器端点接口（SEI）。Project 组件包括这几个 WebService 包下的 Project 几个文件。创建服务器端点接口与 WS-Addressing 的实现例子相似，都创建一个充当 SEI 的 ProjectManager 接口和其实现类 ProjectManagerImpl。本案例的 ProjectManager 接口和其实现类 ProjectManagerImpl 与 WS-Addressing 实现例子的 ProjectManager 接口和 ProjectManagerImpl 相同。同时加入了一个继承 javax.xml.ws.Service 类的 ProjectManagerService 类，ProjectManagerService 类与 WS-Addressing 实现例子的 ProjectManagerService 类基本相同。

（2）服务器端发布程序

编写发布这个 Java 程序为 Web Services 的服务器端程序。创建了 SEI，接着要把这个 SEI 发布出去，还需要一个部署程序，即服务器端的发布程序，本案例采用 JAX-WS 规范来编写服务端程序。创建一个 Server 类，启动这个服务，接着就可以发布 SEI 服务。

Server.java 是服务器启动程序文件，其代码如下：

```
package demo.cxf.ws.policy.server;
```

```
import java.net.URL;
import javax.xml.ws.Endpoint;

import org.apache.cxf.Bus;
import org.apache.cxf.bus.spring.SpringBusFactory;

import demo.cxf.webservices.*;

public class Server {

    public Server() {
        System.out.println("Starting Server");
        SpringBusFactory bf = new SpringBusFactory();
       URL busFile = Server.class.getResource("server.xml");
       Bus bus = bf.createBus(busFile.toString());
       bf.setDefaultBus(bus);

       ProjectManager projectManager = new ProjectManagerImpl();
       String address = "http://localhost:9000/ProjectManager";

        Endpoint.publish(address, projectManager);
    }

    public static void main(String[] args) throws Exception {
        new Server();
        System.out.println("Server ready...");

        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

在代码中要求装载文件 ws_rm.xml，配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:cxf="http://cxf.apache.org/core"
    xmlns:wsa="http://cxf.apache.org/ws/addressing"
    xmlns:http="http://cxf.apache.org/transports/http/configuration"
    xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
    xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
    xsi:schemaLocation="
        http://cxf.apache.org/core
        http://cxf.apache.org/schemas/core.xsd
        http://cxf.apache.org/transports/http/configuration
        http://cxf.apache.org/schemas/configuration/http-conf.xsd
        http://schemas.xmlsoap.org/ws/2005/02/rm/policy
        http://schemas.xmlsoap.org/ws/2005/02/rm/wsrm-policy.xsd
        http://cxf.apache.org/ws/rm/manager
        http://cxf.apache.org/schemas/configuration/wsrm-manager.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <cxf:bus>
        <cxf:features>
            <cxf:logging />
            <wsa:addressing />
```

```
            <wsrm-mgr:reliableMessaging>
                <wsrm-policy:RMAssertion>
                    <wsrm-policy:BaseRetransmissionInterval
                        Milliseconds="4000" />
                    <wsrm-policy:AcknowledgementInterval
                        Milliseconds="2000" />
                </wsrm-policy:RMAssertion>
                <wsrm-mgr:destinationPolicy>
                    <wsrm-mgr:acksPolicy intraMessageThreshold="0" />
                </wsrm-mgr:destinationPolicy>
            </wsrm-mgr:reliableMessaging>
        </cxf:features>
    </cxf:bus>

</beans>
```

在本案例中，服务器启动的服务地址是 http://localhost:9000/ProjectManager。服务器引入了 SEI 接口——ProjectManager 接口。这样 ProjectManager 的 Web Services 就暴露到了外部。可以在浏览器上输入 http://localhost:9000/ProjectManager?wsdl 来查看这个服务的内容。

（3）客户端调用程序

客户端应用程序包括 Web Services 的客户端应用程序和配置文件。

Web Services 的客户端应用程序——Client.java 的代码如下：

```
package demo.cxf.ws.reliablemessaging.client;

import java.lang.reflect.UndeclaredThrowableException;
import java.net.URL;

import org.apache.cxf.Bus;
import org.apache.cxf.bus.spring.SpringBusFactory;

import demo.cxf.webservices.Project;
import demo.cxf.webservices.ProjectImpl;
import demo.cxf.webservices.ProjectManager;
import demo.cxf.webservices.ProjectManagerService;
import demo.cxf.ws.reliablemessaging.common.MessageLossSimulator;

public class Client {

    private static final String USER_NAME = System.getProperty("user.name");

    public Client() {
    }

    public static void main(String[] args) throws Exception {
        System.out.println("Starting Client");

        try {
            SpringBusFactory bf = new SpringBusFactory();
            URL busFile = Client.class.getResource("ws_rm.xml");
            Bus bus = bf.createBus(busFile.toString());
            bf.setDefaultBus(bus);

            bus.getOutInterceptors().add(new MessageLossSimulator());

            ProjectManagerService service = new ProjectManagerService();
```

```
            ProjectManager projectManager = service.getProjectManagerPort();

            //项目的基本情况
            getGreet(projectManager);
            greetingWithParameter(projectManager);
            ……

            // allow aynchronous resends to occur
            Thread.sleep(30 * 1000);

            bus.shutdown(true);

        } catch (UndeclaredThrowableException ex) {
            ex.getUndeclaredThrowable().printStackTrace();
        } catch (Exception ex) {
            ex.printStackTrace();
        } finally {
            System.exit(0);
        }
    }

    //调用 SEI 的 getGreet 方法
    private static void getGreet(ProjectManager projectManager)throws Exception {
        System.out.println(projectManager.greeting());
    }
    ……
}
```

配置文件 ws_rm.xml 的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:cxf="http://cxf.apache.org/core"
    xmlns:wsa="http://cxf.apache.org/ws/addressing"
    xmlns:http="http://cxf.apache.org/transports/http/configuration"
    xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
    xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
    xsi:schemaLocation="
        http://cxf.apache.org/core
        http://cxf.apache.org/schemas/core.xsd
        http://cxf.apache.org/transports/http/configuration
        http://cxf.apache.org/schemas/configuration/http-conf.xsd
        http://schemas.xmlsoap.org/ws/2005/02/rm/policy
        http://schemas.xmlsoap.org/ws/2005/02/rm/wsrm-policy.xsd
        http://cxf.apache.org/ws/rm/manager
        http://cxf.apache.org/schemas/configuration/wsrm-manager.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <cxf:bus>
        <cxf:features>
            <cxf:logging />
            <wsa:addressing />
            <wsrm-mgr:reliableMessaging>
                <wsrm-policy:RMAssertion>
                    <wsrm-policy:BaseRetransmissionInterval
```

```
                    Milliseconds="4000" />
                <wsrm-policy:AcknowledgementInterval
                    Milliseconds="2000" />
            </wsrm-policy:RMAssertion>
            <wsrm-mgr:destinationPolicy>
                <wsrm-mgr:acksPolicy intraMessageThreshold="0" />
            </wsrm-mgr:destinationPolicy>
        </wsrm-mgr:reliableMessaging>
    </cxf:features>
</cxf:bus>

<http:conduit
    name="{http://webservices.cxf.demo}ProjectManagerPort.http-conduit">
    <http:client
        DecoupledEndpoint="http://localhost:9990/decoupled_endpoint" />
</http:conduit>

</beans>
```

还有一个实现类 MessageLossSimulator.java，代码如下：

```
package demo.cxf.ws.reliablemessaging.common;

import java.io.IOException;
import java.io.OutputStream;
import java.math.BigInteger;
import java.util.ListIterator;
import java.util.logging.Level;
import java.util.logging.Logger;

import org.apache.cxf.interceptor.Fault;
import org.apache.cxf.interceptor.InterceptorChain;
import org.apache.cxf.interceptor.MessageSenderInterceptor;
import org.apache.cxf.io.AbstractWrappedOutputStream;
import org.apache.cxf.message.Message;
import org.apache.cxf.phase.AbstractPhaseInterceptor;
import org.apache.cxf.phase.Phase;
import org.apache.cxf.phase.PhaseInterceptor;
import org.apache.cxf.ws.addressing.AddressingProperties;
import org.apache.cxf.ws.rm.RMContextUtils;

public class MessageLossSimulator extends AbstractPhaseInterceptor<Message> {

    private static final Logger LOG = Logger.getLogger(MessageLossSimulator.class.
getName());
    private int appMessageCount;

    public MessageLossSimulator() {
        super(Phase.PREPARE_SEND);
        addBefore(MessageSenderInterceptor.class.getName());
    }

    public void handleMessage(Message message) throws Fault {
        AddressingProperties maps = RMContextUtils.retrieveMAPs(message, false,true);
        RMContextUtils.ensureExposedVersion(maps);
        String action = null;
        if (maps != null && null != maps.getAction()) {
            action = maps.getAction().getValue();
        }
```

```
        if (RMContextUtils.isRMProtocolMessage(action)) {
            return;
        }
        appMessageCount++;
        // do not discard odd-numbered messages
        if (0 != (appMessageCount % 2)) {
            return;
        }

        // discard even-numbered message
        InterceptorChain chain = message.getInterceptorChain();
        ListIterator it = chain.getIterator();
        while (it.hasNext()) {
            PhaseInterceptor pi = (PhaseInterceptor) it.next();
            if (MessageSenderInterceptor.class.getName().equals(pi.getId())) {
                chain.remove(pi);
                LOG.info("Removed MessageSenderInterceptor from interceptor chain.");
                break;
            }
        }
        message.setContent(OutputStream.class,new WrappedOutputStream(message));

        message.getInterceptorChain().add(
                        new AbstractPhaseInterceptor<Message>(Phase.PREPARE_SEND_ENDING) {
                            public void handleMessage(Message message) throws Fault {
                                try {
                                    message.getContent(OutputStream.class).close();
                                } catch (IOException e) {
                                    throw new Fault(e);
                                }
                            }
                        });
    }

    private class WrappedOutputStream extends AbstractWrappedOutputStream {
        private Message outMessage;

        public WrappedOutputStream(Message m) {
            this.outMessage = m;
        }

        @Override
        protected void onFirstWrite() throws IOException {
            if (LOG.isLoggable(Level.FINE)) {
                BigInteger nr = RMContextUtils.retrieveRMProperties(outMessage,
                        true).getSequence().getMessageNumber();
                LOG.fine("Losing message " + nr);
            }
            wrappedStream = new DummyOutputStream();
        }
    }

    private class DummyOutputStream extends OutputStream {

        @Override
        public void write(int b) throws IOException {
        }
    }
}
```

本例首先创建一个服务名称为 javax.xml.ws.Service 的实现类，接着通过 addPort 方法来传入服务的端口类、绑定方式和 Web Services 的地址。最后通过 getPort 方法来获得服务器的 SEI 接口。这就是 Web Services 服务接口 ProjectManager。后面基于这个 ProjectManager 的 SEI 来实现各种业务操作。

4. 测试和运行结果

测试编写好的程序代码，首先要启动服务器程序，即在 Eclipse 上运行 Server.java。然后运行客户端程序，客户端程序输出结果如下：

```
Starting Client

ID: 1
Address: http://localhost:9000/ProjectManager
Encoding: UTF-8
Content-Type: text/xml
Headers: {SOAPAction=["http://schemas.xmlsoap.org/ws/2005/02/rm/CreateSequence"],
Accept=[*/*]}
Payload: <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header><Action
xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">http://schemas.xmlsoap.o
rg/ws/2005/02/rm/CreateSequence</Action><MessageID
xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">urn:uuid:ab97fc1e-29e0-4
7bb-9998-98a147124a9a</MessageID><To
xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">http://localhost:9000/Pr
ojectManager</To><ReplyTo
xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing"><Address>http://schemas.
xmlsoap.org/ws/2004/08/addressing/role/anonymous</Address></ReplyTo></soap:Header>
<soap:Body><CreateSequence xmlns="http://schemas.xmlsoap.org/ws/2005/02/rm" xmlns:ns2=
"http://schemas.xmlsoap.org/ws/2004/08/addressing"><AcksTo><ns2:Address>http://sc
hemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous</ns2:Address></AcksTo><Exp
ires>PT0S</Expires><Offer><Identifier>urn:uuid:db8e22ab-d99d-4c58-a246-4b8316f874
ef</Identifier><Expires>PT0S</Expires></Offer></CreateSequence></soap:Body></soap
:Envelope>
---------------------------------------
2011-2-26 23:51:19 org.apache.cxf.interceptor.LoggingInInterceptor logging
信息: Inbound Message
----------------------------
ID: 1
Response-Code: 200
Encoding: UTF-8
Content-Type: text/xml; charset=utf-8
Headers: {Content-Length=[1013], Server=[Jetty(6.1.21)], content-type=[text/xml;
charset=utf-8]}
Payload: <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header><Action
xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">http://schemas.xmlsoap.o
rg/ws/2005/02/rm/CreateSequenceResponse</Action><MessageID
xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">urn:uuid:5c867625-5646-4
5da-9fa5-c6d7a529c9bd</MessageID><To xmlns="http://schemas.xmlsoap.org/ws/2004/08/
addressing">http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous</To><
RelatesTo
xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">urn:uuid:ab97fc1e-29e0-4
7bb-9998-98a147124a9a</RelatesTo></soap:Header><soap:Body><CreateSequenceResponse
xmlns="http://schemas.xmlsoap.org/ws/2005/02/rm"
xmlns:ns2="http://schemas.xmlsoap.org/ws/2004/08/addressing"><Identifier>urn:uuid
```

```
:377a8b65-5aff-4c22-914a-822ac60c6568</Identifier><Expires>P0Y0M0DT0H0M0.0S</Expi
res><Accept><AcksTo><ns2:Address>http://localhost:9000/ProjectManager</ns2:Addres
s></AcksTo></Accept></CreateSequenceResponse></soap:Body></soap:Envelope>
--------------------------------------
2011-2-26 23:51:19 org.apache.cxf.ws.rm.soap.RMSoapInterceptor updateServiceModelInfo
信息: Updating service model info in exchange
2011-2-26  23:51:19  org.apache.cxf.interceptor.LoggingOutInterceptor$LoggingCallback
onClose
信息: Outbound Message
---------------------------
ID: 2
Address: http://localhost:9000/ProjectManager
Encoding: UTF-8
Content-Type: text/xml
Headers: {SOAPAction=[""], Accept=[*/*]}
Payload:  <soap:Envelope  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header><Action
xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">http://webservices.cxf.d
emo/ProjectManager/greeting</Action><MessageID
xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">urn:uuid:d6160c8d-8876-4
b97-9d6b-aa58d13843f4</MessageID><To
xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">http://localhost:9000/Pr
ojectManager</To><ReplyTo
xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing"><Address>http://schemas.
xmlsoap.org/ws/2004/08/addressing/role/anonymous</Address></ReplyTo><wsrm:Sequenc
e xmlns:ns2="http://schemas.xmlsoap.org/ws/2004/08/addressing" xmlns:wsrm="http://
schemas.xmlsoap.org/ws/2005/02/rm"><wsrm:Identifier>urn:uuid:377a8b65-5aff-4c22-9
14a-822ac60c6568</wsrm:Identifier><wsrm:MessageNumber>1</wsrm:MessageNumber></wsr
m:Sequence></soap:Header><soap:Body><ns2:greeting  xmlns:ns2="http://webservices.
cxf.demo/"/></soap:Body></soap:Envelope>
--------------------------------------
2011-2-26 23:51:19 org.apache.cxf.interceptor.LoggingInInterceptor logging
信息: Inbound Message
----------------------------
ID: 2
Response-Code: 200
Encoding: UTF-8
Content-Type: text/xml; charset=utf-8
Headers: {Content-Length=[1049], Server=[Jetty(6.1.21)], content-type=[text/xml;
charset=utf-8]}
Payload:  <soap:Envelope  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header><Action    xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">
http://webservices.cxf.demo/ProjectManager/greetingResponse</Action><MessageID xmlns=
"http://schemas.xmlsoap.org/ws/2004/08/addressing">urn:uuid:af6dd111-1bc3-4031-ad
b7-5c8a659a37e4</MessageID><To xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">
http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous</To><RelatesTo
xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">urn:uuid:d6160c8d-8876-4
b97-9d6b-aa58d13843f4</RelatesTo><wsrm:Sequence xmlns:ns2="http://schemas.xmlsoap.
org/ws/2004/08/addressing" xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm">
<wsrm:Identifier>urn:uuid:db8e22ab-d99d-4c58-a246-4b8316f874ef</wsrm:Identifier>
<wsrm:MessageNumber>1</wsrm:MessageNumber></wsrm:Sequence></soap:Header><soap:Bod
y><ns2:greetingResponse   xmlns:ns2="http://webservices.cxf.demo/"><return>Hello,
This is ProjectManager.</return></ns2:greetingResponse></soap:Body></soap:Envelope>
--------------------------------------
Hello, This is ProjectManager.
2011-2-26 23:51:19 demo.cxf.ws.reliablemessaging.common.MessageLossSimulator handleMessage
信息: Removed MessageSenderInterceptor from interceptor chain.
2011-2-26  23:51:19  org.apache.cxf.interceptor.LoggingOutInterceptor$LoggingCallback
```

```
onClose
信息: Outbound Message
---------------------------
ID: 3
Address: http://localhost:9000/ProjectManager
Encoding: UTF-8
Content-Type: text/xml
Headers: {SOAPAction=[""]}
Payload: <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"><soap:
Header><Action  xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">http://
webservices.cxf.demo/ProjectManager/greetingWithParameter</Action><MessageID
xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">urn:uuid:46723f73-df69-4
b1e-acb8-6d16a2bb102b</MessageID><To  xmlns="http://schemas.xmlsoap.org/ws/2004/08/
addressing">http://localhost:9000/ProjectManager</To><ReplyTo xmlns="http://schemas.
xmlsoap.org/ws/2004/08/addressing"><Address>http://schemas.xmlsoap.org/ws/2004/08
/addressing/role/anonymous</Address></ReplyTo><wsrm:Sequence   xmlns:ns2="http://
schemas.xmlsoap.org/ws/2004/08/addressing"  xmlns:wsrm="http://schemas.xmlsoap.org/
ws/2005/02/rm"><wsrm:Identifier>urn:uuid:377a8b65-5aff-4c22-914a-822ac60c6568</ws
rm:Identifier><wsrm:MessageNumber>2</wsrm:MessageNumber></wsrm:Sequence></soap:He
ader><soap:Body><ns2:greetingWithParameter xmlns:ns2="http://webservices.cxf.demo/">
<greet>greet</greet></ns2:greetingWithParameter></soap:Body></soap:Envelope>
……
……
```

服务器端程序输出结果如下：

```
ID: 1
Address: http://localhost:9000/ProjectManager
Encoding: UTF-8
Content-Type: text/xml
Headers: {SOAPAction=["http://schemas.xmlsoap.org/ws/2005/02/rm/CreateSequence"],
Accept=[*/*]}
Payload:  <soap:Envelope  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header><Action    xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">
http://schemas.xmlsoap.org/ws/2005/02/rm/CreateSequence</Action><MessageID xmlns=
"http://schemas.xmlsoap.org/ws/2004/08/addressing">urn:uuid:9852e657-4f35-4956-99
e3-612d744f58ae</MessageID><To xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">
http://localhost:9000/ProjectManager</To><ReplyTo xmlns="http://schemas. xmlsoap.org/
ws/2004/08/addressing"><Address>http://schemas.xmlsoap.org/ws/2004/08/addressing/
role/anonymous</Address></ReplyTo></soap:Header><soap:Body><CreateSequence xmlns=
"http://schemas.xmlsoap.org/ws/2005/02/rm" xmlns:ns2="http://schemas.xmlsoap.org/
ws/2004/08/addressing"><AcksTo><ns2:Address>http://schemas.xmlsoap.org/ws/2004/08
/addressing/role/anonymous</ns2:Address></AcksTo><Expires>PT0S</Expires><Offer><I
dentifier>urn:uuid:a56ca435-d515-47a2-9f0e-fe6e76852b55</Identifier><Expires>PT0S
</Expires></Offer></CreateSequence></soap:Body></soap:Envelope>
---------------------------------------
2011-2-26 23:52:23 org.apache.cxf.interceptor.LoggingInInterceptor logging
信息: Inbound Message
---------------------------
ID: 1
Response-Code: 200
Encoding: UTF-8
Content-Type: text/xml; charset=utf-8
Headers: {Content-Length=[1013], Server=[Jetty(6.1.21)], content-type=[text/xml;
charset=utf-8]}
Payload:  <soap:Envelope  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header><Action xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">http://
```

```
schemas.xmlsoap.org/ws/2005/02/rm/CreateSequenceResponse</Action><MessageID  xmlns=
"http://schemas.xmlsoap.org/ws/2004/08/addressing">urn:uuid:82a58d6a-e16c-4c75-82
49-989f73718b31</MessageID><To xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">
http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous</To><RelatesTo
xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">urn:uuid:9852e657-4f35-4
956-99e3-612d744f58ae</RelatesTo></soap:Header><soap:Body><CreateSequenceResponse
xmlns="http://schemas.xmlsoap.org/ws/2005/02/rm"  xmlns:ns2="http://schemas.xmlsoap.org/
ws/2004/08/addressing"><Identifier>urn:uuid:1181919f-eaa0-4db6-865c-e730d62fa027<
/Identifier><Expires>P0Y0M0DT0H0M0.0S</Expires><Accept><AcksTo><ns2:Address>http:
//localhost:9000/ProjectManager</ns2:Address></AcksTo></Accept></CreateSequenceRe
sponse></soap:Body></soap:Envelope>
--------------------------------------
2011-2-26 23:52:23 org.apache.cxf.ws.rm.soap.RMSoapInterceptor updateServiceModelInfo
信息: Updating service model info in exchange
2011-2-26 23:52:23 org.apache.cxf.interceptor.LoggingOutInterceptor$LoggingCallback
onClose
信息: Outbound Message
----------------------------
ID: 2
Address: http://localhost:9000/ProjectManager
Encoding: UTF-8
Content-Type: text/xml
Headers: {SOAPAction=[""], Accept=[*/*]}
Payload:  <soap:Envelope  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header><Action    xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">
http://webservices.cxf.demo/ProjectManager/greeting</Action><MessageID
xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">urn:uuid:6ecc61de-63da-4
672-8e51-80eff3838959</MessageID><To xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">
http://localhost:9000/ProjectManager</To><ReplyTo  xmlns="http://schemas.xmlsoap.org/
ws/2004/08/addressing"><Address>http://schemas.xmlsoap.org/ws/2004/08/addressing/
role/anonymous</Address></ReplyTo><wsrm:Sequence xmlns:ns2="http://schemas.xmlsoap.
org/ws/2004/08/addressing"  xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm">
<wsrm:Identifier>urn:uuid:1181919f-eaa0-4db6-865c-e730d62fa027</wsrm:Identifier>
<wsrm:MessageNumber>1</wsrm:MessageNumber></wsrm:Sequence></soap:Header><soap:Body>
<ns2:greeting xmlns:ns2="http://webservices.cxf.demo/"/></soap:Body></soap:Envelope>
--------------------------------------
2011-2-26 23:52:23 org.apache.cxf.interceptor.LoggingInInterceptor logging
信息: Inbound Message
----------------------------
ID: 2
Response-Code: 200
Encoding: UTF-8
Content-Type: text/xml; charset=utf-8
Headers: {Content-Length=[1049], Server=[Jetty(6.1.21)], content-type=[text/xml;
charset=utf-8]}
Payload: <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"><soap:
Header><Action    xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">http://
webservices.cxf.demo/ProjectManager/greetingResponse</Action><MessageID
xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">urn:uuid:9c70a2e6-432d-4
23e-b0e5-b46d8628283e</MessageID><To  xmlns="http://schemas.xmlsoap.org/ws/2004/08/
addressing">http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous</To><
RelatesTo xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">urn:uuid:6ecc61de-
63da-4672-8e51-80eff3838959</RelatesTo><wsrm:Sequence  xmlns:ns2="http://schemas.
xmlsoap.org/ws/2004/08/addressing" xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/
rm"><wsrm:Identifier>urn:uuid:a56ca435-d515-47a2-9f0e-fe6e76852b55</wsrm:Identifi
er><wsrm:MessageNumber>1</wsrm:MessageNumber></wsrm:Sequence></soap:Header><soap:
Body><ns2:greetingResponse  xmlns:ns2="http://webservices.cxf.demo/"><return>Hello,
This is ProjectManager.</return></ns2:greetingResponse></soap:Body></soap:Envelope>
--------------------------------------
```

```
Hello, This is ProjectManager.
2011-2-26 23:52:23 demo.cxf.ws.reliablemessaging.common.MessageLossSimulator handleMessage
信息: Removed MessageSenderInterceptor from interceptor chain.
2011-2-26 23:52:23 org.apache.cxf.interceptor.LoggingOutInterceptor$LoggingCallback
onClose
信息: Outbound Message
---------------------------
ID: 3
Address: http://localhost:9000/ProjectManager
Encoding: UTF-8
Content-Type: text/xml
Headers: {SOAPAction=[""]}
Payload:  <soap:Envelope  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header><Action xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">
http://webservices.cxf.demo/ProjectManager/greetingWithParameter</Action><Message
ID  xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">urn:uuid:3250b5c4-5bfa-
4695-b487-302b3cd342f9</MessageID><To xmlns="http://schemas.xmlsoap.org/ws/2004/08/
addressing">http://localhost:9000/ProjectManager</To><ReplyTo  xmlns="http://schemas.
xmlsoap.org/ws/2004/08/addressing"><Address>http://schemas.xmlsoap.org/ws/2004/08/
addressing/role/anonymous</Address></ReplyTo><wsrm:Sequence  xmlns:ns2="http://
schemas.xmlsoap.org/ws/2004/08/addressing"  xmlns:wsrm="http://schemas.xmlsoap.org/
ws/2005/02/rm"><wsrm:Identifier>urn:uuid:1181919f-eaa0-4db6-865c-e730d62fa027</ws
rm:Identifier><wsrm:MessageNumber>2</wsrm:MessageNumber></wsrm:Sequence></soap:He
ader><soap:Body><ns2:greetingWithParameter  xmlns:ns2="http://webservices.cxf.demo/">
<greet>greet</greet></ns2:greetingWithParameter></soap:Body></soap:Envelope>
……
……
```

5. 其他内容

上述的所有调用都基于 JAX-WS 规范。

本例代码参见配套资源代码中“第 13 章\ws_reliablemessaging”文件夹。

13.4　Apache CXF 支持 WS-Security

WS-Security 规范提出了一套标准的 SOAP 扩展，可以在构建安全的 Web Services 以实现完整性和机密性时使用。其全称扩展为“Web 服务安全性语言（Web Services Security Language）”或“WS-Security”。

本节简单介绍一下 WS-Security 的概念，以及 Apache CXF 使用 WS-Security 的方法和例子。

13.4.1　WS-Security 介绍

早期在因特网上传递 Web Services 领域的开发者发现：他们要么提供开放的、任何人都可使用的服务，要么开发自己的服务，这些服务通常是完全私有的，非常特定于某个公司并且采用某种安全模式。这是两种极端的模式。如把因特网作为传输工具的采用者通常使用某种形式的注册流程（如 Google）公开因特网服务，或者仅仅向一小部分与自己有着亲密、可信关系的业务伙伴提供服务。例如，为了使用 Google 的启用 Web Services 的搜索引擎，服务请求者必须首先用基于 HTML 的形式向 Google 注册。作为注册流程的一部分，Google 向请求者发送一封带有一个安全性“令牌”的邮件。当请求者调用服务时，他们向 Google 提供这个作为 SOAP 消息的一部分的令牌来证明他们是 Google Web Services 已注册的、已授权的用户。

在这些情况下，即使服务提供者使用的是 SOAP 之类的业界标准，还是需要提供有关安全性模式/流程的额外信息以使服务请求者能够使用服务。这样就会把请求者和提供者紧紧地捆绑在一起，而这种结果是双方都不愿意的。

2002 年 4 月，IBM、Microsoft 和 Verisign 对这种需求提出了一个保护 Web Services 的业界标准方法。

WS-Security 规范以现有的密码学以及 XML 加密和签名行业标准为基础，为 Web Services 应用程序提供了一组全面的安全特性，可以通过 WS-Policy 规范和 WS-SecurityPolicy 规范来指定特定应用程序可以使用哪些特性，从而允许服务客户机自行配置以便访问服务。通过跨多个平台和 Web Services 框架对这些标准的广泛支持，可以实现良好的互操作性。

为什么需要 WS-Security 规范？随着客户应用程序的逐步发展，确定了一套有关安全性的非功能性需求，包括下面三点：

（1）客户与其业务伙伴之间在因特网上传播的通信不应该被第三方看到。

（2）客户必须能够确定消息来自哪个人并能够验证发送方就是那个发送方声称的发送方。

（3）客户必须能够确定被传送的数据没有被篡改。

使用 HTTPS/SSL 传输安全性可以解决非功能性需求。但这个应用程序将是个点到点应用程序，不涉及第三方服务提供者或者中介者，所以目前有一种思路，即使用密码术加密整个 SOAP 消息或加密 SOAP 消息的一部分。不过这种思路还未实现。

使用数字签名和数字证书可以解决非功能性需求。当使用数字证书方法时，Web Services 请求者必须有一个由可信认证中心签署的数字证书。请求者将使用这个证书来表明他们的身份，并对 SOAP 消息进行数字签名，这样就可以验证请求者的身份和消息的完整性。

一旦客户的系统接收到消息，就要对消息做时间戳记并进行日志记录。此时，数字签名会得到验证。验证过程将确保消息来自发送方，并且由于消息内容是在发送方的站点上签署的，所以还要验证消息内容在传输过程中没有被篡改。

WS-Security 规范覆盖到下面三方面的内容。（1）WS-Security 规范描述通过提供消息完整性、消息机密性和单独消息认证来实现对 SOAP 消息传递增强的质量保护。这些机制可以用于提供多种安全性模型和加密技术。（2）WS-Security 规范还提供关联安全性令牌和消息的通用机制。WS-Security 规范不需要特定类型的安全性令牌。它在设计上就是可扩展的（例如，支持多安全性令牌格式）。举例来说，客户机可能会提供身份证明和他们有特定商业认证的证明。（3）WS-Security 规范还描述如何对二进制安全性令牌进行编码。此规范特别描述如何对 X.509 证书和 Kerberos 票据编码以及如何加入难以理解的加密密钥。该规范还包括可以进一步描述消息中包含凭证特征的扩展性机制。

WS-Security 规范很灵活，它被设计成用来构建多种安全性模型（包括 PKI、Kerberos 和 SSL）的基础。WS-Security 规范特别为多安全性令牌、多信任域、多签名格式和多加密技术提供支持。

WS-Security 规范提供了三种主要的机制：安全性令牌传播、消息完整性和消息机密性。这些机制本身并不提供完整的安全性解决方案。相反，WS-Security 规范是一种构件，它可以与其他 Web Services 扩展和更高级的特定于应用程序的协议联合使用，以适应多种安全性模型和加密技术。这些机制可以独立使用（例如，传送安全性令牌），或以紧密集成的方式使用（例如，对

消息签名和加密，并提供与用于签名和加密的密钥相关的安全性令牌层次结构）。

WS-Security 规范紧跟着 IBM 和 Microsoft 现有的 Web 服务安全性规范，包括 SOAP-SEC、Microsoft 的 WS-Security 和 WS-License，还有 IBM 的安全性令牌和加密文档。

13.4.2　Apache CXF 使用 WS-Security 的配置

目前，Apache CXF 框架实现 WS-Security 规范要结合 WSS4J。若要使用集成，开发者需要配置 WSS4J 的拦截器，并将这些拦截器添加到应用系统的服务端或客户端。

13.4.2.1　配置 WSS4J 拦截器

在 Apache CXF 应用中，为实现服务器或客户端的 WS-Security，需要设置 WSS4J 拦截器。设置 WSS4J 拦截器的方式要么通过编写程序代码，要么通过 Spring 配置。下面会针对如何配置并适用于特定安全行动的拦截器进行详细描述。

重要提示：（1）如果使用的是 CXF 2.0.x 版本，开发者必须添加 SAAJ（输入/输出）拦截器（在 Apache CXF 的 2.1 版本后会自动完成该功能），因为 2.0.x 版本要为每个响应/请求创建 DOM。WS-Security 需要 DOM 库的支持。（2）该 Web Services 提供者不需要基于 WS-Security 拦截器来输入或输出。例如，若只是需要对传入消息签名，Web Services 提供者只需要一个输入的 WSS4J 拦截器，同时 SOAP 客户端只需要一个输出的 WSS4J 拦截器。

13.4.2.2　编写程序代码来实现拦截器

在服务器端，需要在 Apache CXF 服务器端点上添加拦截器。如果使用 JAX-WS 的 API 来发布服务，其代码如下：

```
import org.apache.cxf.endpoint.Endpoint;
import org.apache.cxf.jaxws.EndpointImpl;

EndpointImpl jaxWsEndpoint = (EndpointImpl) Endpoint.publish("http://host/service",
myServiceImpl);
Endpoint cxfEndpoint = jaxWsEndpoint.getServer().getEndpoint();
```

如果使用 JAX-WS 的 ServerFactoryBean，可以简单地通过服务器对象来访问服务，其代码如下：

```
import org.apache.cxf.endpoint.Endpoint;
import org.apache.cxf.endpoint.Server;
import org.apache.cxf.frontend.ServerFactoryBean;

ServerFactoryBean factory = ...;
...
Server server = factory.create();
Endpoint cxfEndpoint = server.getEndpoint();
```

在客户端，开发者可以通过 ClientProxy 对象来获取对 CXF 服务器端点的使用引用：

```
GreeterService gs = new GreeterService();
Greeter greeter = gs.getGreeterPort();
...
org.apache.cxf.endpoint.Client client = org.apache.cxf.frontend.ClientProxy.
```

```
getClient(greeter);
org.apache.cxf.endpoint.Endpoint cxfEndpoint = client.getEndpoint();
```

下面代码已经添加拦截器：

```
import org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor;
import org.apache.cxf.ws.security.wss4j.WSS4JOutInterceptor;
...

Map<String,Object> inProps= new HashMap<String,Object>();
... // how to configure the properties is outlined below;

WSS4JInInterceptor wssIn = new WSS4JInInterceptor(inProps);
cxfEndpoint.getInInterceptors().add(wssIn);
cxfEndpoint.getInInterceptors().add(new SAAJInInterceptor()); // 2.0.x only; not
needed in 2.1+

Map<String,Object> outProps = new HashMap<String,Object>();
... // how to configure the properties is outlined below;

WSS4JOutInterceptor wssOut = new WSS4JOutInterceptor(outProps);
cxfEndpoint.getOutInterceptors().add(wssOut);
cxfEndpoint.getOutInterceptors().add(new SAAJOutInterceptor()); // 2.0.x only; not
needed in 2.1+
```

13.4.2.3 Spring 配置

如果采用 Spring 框架来构建端点（例如，Web Services 运行在如 Tomcat 的 Servlet 容器），可以轻松地完成上述的定义，配置文件如下：

```
<import resource="classpath:META-INF/cxf/cxf.xml" />
<import resource="classpath*:META-INF/cxf/cxf-extension-*.xml" />

<jaxws:endpoint id="myService"
  implementor="com.acme.MyServiceImpl"
  address="http://localhost:9001/MyService">

  <bean id="myPasswordCallback"
    class="com.mycompany.webservice.ServerPasswordCallback"/>

  <jaxws:inInterceptors>
    <!-- SAAJ Interceptor needs to be explicitly declared only in CXF 2.0.x -->
    <bean class="org.apache.cxf.binding.soap.saaj.SAAJInInterceptor"/>
    <bean class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
      <constructor-arg>
        <map>
          <entry key="action" value="UsernameToken"/>
          <entry key="passwordType" value="PasswordDigest"/>
          <entry key="signaturePropFile" value="..."/>
          <entry key="passwordCallbackRef">
            <ref bean="myPasswordCallback"/>
          </entry>
          ...
        </map>
      </constructor-arg>
    </bean>
```

```
  </jaxws:inInterceptors>
</jaxws:endpoint>
```

主要是看 bean class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor"元素的配置。该元素下的子元素 constructor-arg 映射相应的 WSHandlerConstants.XXXXX 和 WSConstants.XXXX 常量并对应 WSS4J 的 WSHandlerConstants 类和 WSConstants 类。因此，通过查看 WSHandlerConstants，可以看到 WSHandlerConstants.USERNAME_TOKEN 值将需要“UserNameToken”，而不是在 Spring 中配置。

如果希望避免在文本上查找 WSHandlerConstants.XXXXX 和 WSConstants.XXXX 常量，可以使用 Spring 的 util 命名空间在 Spring 定义文档中引用静态变量，代码如下所示：

```
<beans
  ...
  xmlns:util="http://www.springframework.org/schema/util"
  ...
  xsi:schemaLocation="
        ...
        http://www.springframework.org/schema/util  http://www.springframework.org/
schema/util/spring-util.xsd">
  ...

  <bean class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
    <constructor-arg>
      <map>
        <entry value="UsernameToken">
          <key>
            <util:constant  static-field="org.apache.ws.security.handler.WSHandlerConstants.
ACTION"/>
          </key>
        </entry>
        ...
      </map>
    </constructor-arg>
  </bean>
  ...
```

1. 其他配置选项

虽然 Apache CXF 框架的 WSS4J 拦截器可以支持 WSHandlerConstants.XXXXX 和 WSConstants.XXXX 等标准配置属性，Apache CXF 框架也提供了访问一些额外的、低级别的配置能力，如 WSS4J 和其他一些安全相关的拦截器。

2. 验证签名和消息内容加密

Apache CXF 框架允许一个 CryptoCoverageChecker 拦截器验证签名和加密而不需要迁移到基于配置上的 WS-SecurityPolicy。拦截器可以支持元素和内容级别。使用 API 来配置这个拦截器，请按照下面的代码例子来进行编写：

```
import org.apache.cxf.ws.security.wss4j.CryptoCoverageChecker;
import org.apache.cxf.ws.security.wss4j.CryptoCoverageChecker.XPathExpression;
import org.apache.cxf.ws.security.wss4j.CryptoCoverageUtil.CoverageScope;
import org.apache.cxf.ws.security.wss4j.CryptoCoverageUtil.CoverageType;
```

```
Map<String, String> prefixes = new HashMap<String, String>();
        prefixes.put("ser", "http://www.sdj.pl");
        prefixes.put("soap", "http://schemas.xmlsoap.org/soap/envelope/");

List<XPathExpression> xpaths = Arrays.asList(
    new XPathExpression("//ser:Header", CoverageType.SIGNED, CoverageScope.ELEMENT),
    new XPathExpression("//soap:Body", CoverageType.ENCRYPTED, CoverageScope.CONTENT));

CryptoCoverageChecker checker = new CryptoCoverageChecker(prefixes, xpaths);
```

还可以在 Spring 配置文件中按照 bean 的常规格式来定义拦截器。

经过上述拦截器配置后，只需添加配置的拦截器到客户端或服务器的拦截器链，与上述所示的 WSS4J 拦截器相同。如果要加强任何 XPath 的覆盖面，保证要拒绝包含有拦截器链中的 WSS4JinInterceptor 拦截器或者该拦截器的所有请求。

3. 定制处理器

作为 Apache CXF 框架，开发者可以在 WSS4JInInterceptor 定义定制的 WSS4J 处理器配置。要启用这个配置选项，配置文件提供了一个 non-WSS4J 定义属性——wss4j.processor.map 到 WSS4JInInterceptor。同样的配置也可以通过 API 来完成。key 值是一个 WS-S 头元素的 XML 限定名来处理特定处理器的执行情况。key 项值可以是一个字符串，表示一个实例化的处理器类名称、对象实现处理器名称，或者是一个定义 WS-S 头元素禁用处理的 null 值。下面是示例代码。

```
<bean class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
  <constructor-arg>
    <map>
      ...
      <!-- This reconfigures the processor implementation that WSS4j uses to
             process a WS-S Signature element. -->
      <entry key="wss4j.processor.map">
        <map key-type="javax.xml.namespace.QName">
          <entry value="my.class">
            <key>
              <bean class="javax.xml.namespace.QName">
                <constructor-arg value="http://www.w3.org/2000/09/xmldsig#"/>
                <constructor-arg value="Signature"/>
              </bean>
            </key>
          </entry>
        </map>
      </entry>
      ...
    </map>
  </constructor-arg>
</bean>
```

4. 定制操作

在 Apache CXF 框架中，可以在 WSS4JOutInterceptor 配置中定义定制的 WSS4J 活动。要启动这个配置选项，提供了一个 non-WSS4J 定义属性——wss4j.action.map 到 WSS4JOutInterceptor。同样配置可以通过 API 来完成。key 值是一个整数，表示 WSS4J 行动标识符。key 项值可以是一个字符串，表示一个实例化活动的类或一个实现活动的对象。此配置选项允许开发者重写内置的行

动实现或添加开发者自己的。下面是示例代码。

```
<bean class="org.apache.cxf.ws.security.wss4j.WSS4JOutInterceptor">
  <constructor-arg>
    <map>
      ...
      <!-- Redefines the action for SAMLTokenSigned to use a custom implementation. -->
      <entry key="wss4j.action.map">
        <map key-type="java.lang.Integer" value-type="java.lang.Object">
          <entry key="0x10" value-ref="mySamlTokenSignedAction"/>
        </map>
      </entry>     ...
    </map>
  </constructor-arg>
</bean>
```

13.4.2.4　配置 WS-Security 操作

采用用户名令牌认证的实现。WS-Security 规范支持许多指定令牌方式。其中之一是 UserNameToken 头。它采用一种标准方式进行一个用户名和密码的通信或密码转化到另一个端点。当使用 UsernameToken，一定要检查 OASIS 用户名称令牌配置文件规范，这也是出于安全性的考虑。请注意，无论是在 Apache CXF 框架中或在 WSS4J 中实现，nonce 支持为防止重复攻击而采用的防守规范。

对于服务器端，开发者需要设置 WSS4JInInterceptor 的下列属性：

```
inProps.put(WSHandlerConstants.ACTION, WSHandlerConstants.USERNAME_TOKEN);
// Password type : plain text
inProps.put(WSHandlerConstants.PASSWORD_TYPE, WSConstants.PW_TEXT);
// for hashed password use:
//properties.put(WSHandlerConstants.PASSWORD_TYPE, WSConstants.PW_DIGEST);
// Callback used to retrieve password for given user.
inProps.put(WSHandlerConstants.PW_CALLBACK_CLASS,
ServerPasswordHandler.class.getName());
```

该密码回调类允许开发人员去检索一个给定的用户使用密码，以便于 WS-Security 可以决定他们是否授权密码。这里是一个小例子：

```
import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.ws.security.WSPasswordCallback;

public class ServerPasswordCallback implements CallbackHandler {

    public void handle(Callback[] callbacks) throws IOException, UnsupportedCallbackException {

        WSPasswordCallback pc = (WSPasswordCallback) callbacks[0];

        if (pc.getIdentifer().equals("joe")) {
            // set the password on the callback. This will be compared to the
            // password which was sent from the client.
            pc.setPassword("password");
        }
```

```
    }
}
```

请注意，对于一个纯文本密码（或任何其他未知输入密码），密码验证委托给回调类。看 WSS4Jproject 的文档，见 org.apache.ws.security.processor.UsernameTokenProcessor 的 handleUsernameToken 方法。在这种情况下，ServerPasswordCallback 对象的实现代码如下：

```
public class ServerPasswordCallback implements CallbackHandler {

    public void handle(Callback[] callbacks) throws IOException, UnsupportedCallbackException {
        WSPasswordCallback pc = (WSPasswordCallback) callbacks[0];

        if (pc.getIdentifer().equals("joe") {
            if (!pc.getPassword().equals("password")) {
                throw new IOException("wrong password");
            }
        }
    }
}
```

在客户端，需要配置 WSS4J 即将消失的属性：

```
outProps.put(WSHandlerConstants.ACTION, WSHandlerConstants.USERNAME_TOKEN);
// Specify our username
outProps.put(WSHandlerConstants.USER, "joe");
// Password type : plain text
outProps.put(WSHandlerConstants.PASSWORD_TYPE, WSConstants.PW_TEXT);
// for hashed password use:
//properties.put(WSHandlerConstants.PASSWORD_TYPE, WSConstants.PW_DIGEST);
// Callback used to retrieve password for given user.
outProps.put(WSHandlerConstants.PW_CALLBACK_CLASS,
ClientPasswordHandler.class.getName());
```

再次使用密码回调，除去这一次不是在服务器端指定的，开发人员指定希望发送消息的密码，这样就不必在应用系统的配置文件中存储密码。

```
import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.ws.security.WSPasswordCallback;

public class ClientPasswordCallback implements CallbackHandler {
    public void handle(Callback[] callbacks) throws IOException, UnsupportedCallbackException {
        WSPasswordCallback pc = (WSPasswordCallback) callbacks[0];
        // set the password for our message.
        pc.setPassword("password");
    }
}
```

在多用户采用不同密码的情况下，使用 WSPasswordCallback 的 getIdentifer 方法来获取当前 SOAP 请求的用户名。

13.4.3　Apache CXF 的 WS-Security 的实现例子

本程序演示 Apache CXF 框架遵循 WS-Security 规范的客户端和服务器端调用，客户端采用 Apache CXF 框架的代码优先（Java First）前端应用模式。采用 POJO 对象作为中间传输对象。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，都使用 POJO 对象实现客户端和服务器端的数据交换。后台服务器端为遵循 JAX-WS 规范的 Endpoint 来实现，采用 Spring IoC 注入的客户端来实现。前端访问程序采用遵循 JAX-WS 规范的 Service 方式来实现。

WS-Security 配置如下：在客户端和服务器端点加入 WSS4JInterceptors。服务器和客户端都配置了传出和传入拦截器。一些行动如时间戳（Timestamp）、用户名令牌（UsernameToken）、签名（Signature）、加密（Encryption）等都将被记录下来。这些都是通过在拦截器适当地配置属性实现的。日志记录功能用来记录入站和出站的 SOAP 消息，并显示这些信息到控制台。本程序实现的结构如图 13-11 所示。

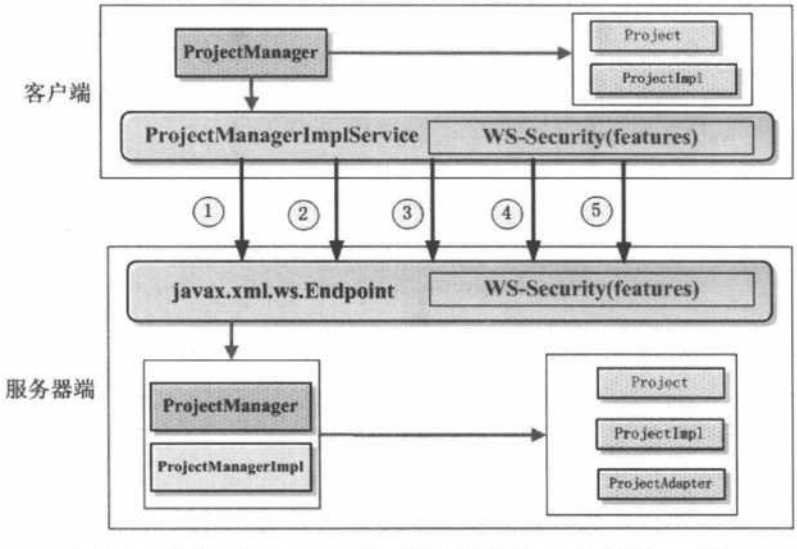


图 13-11　遵循 WS-Security 规范的客户端和服务器端调用模式图

其中：① 表示客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。在客户端和服务端的 SEI 都是 ProjectManager 接口。

2. 应用程序案例实现步骤

创建 WS-Security 实现 Web Services 程序的流程步骤如图 13-12 所示。

流程图说明，要实现以下步骤：

（1）创建一个 Java 项目。

（2）创建一个服务端点接口（SEI）及其相关对象和注释。

（3）配置签名文件和加密文件。

（4）编写发布这个 Java 程序为 Web Services 的服务器端程序和服务器配置文件。

（5）编写 Web Services 的客户端应用程序和客户端配置文件。

（6）启动服务器程序，对外发布 Web Services。

（7）客户端调用 Web Services。

图 13-12　ws_security_sign_enc 应用程序案例实现步骤流程图

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 13-13 所示。

包目录包括三个部分，第一个包目录 demo.cxf.ws.security.server 是服务器程序目录。第二个包目录 demo.cxf.ws.security.client 是客户端程序目录。第三个包目录 demo.cxf.webservices 是服务器端点接口 SEI 和相关对象等公用部分，服务器端和客户端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口，服务器端和客户端都必须引用。但服务器端和客户端对公用部分的引用是不同的。客户端程序一般只引用接口，如 ProjectManager 和 Project

接口，而服务器端程序是接口和实现类都要引用。

本案例还有一些目录也是程序的组成部分，主要是签名目录和加密目录，内容如图 13-14 所示。

图 13-13　ws_security_sign_enc 程序结构

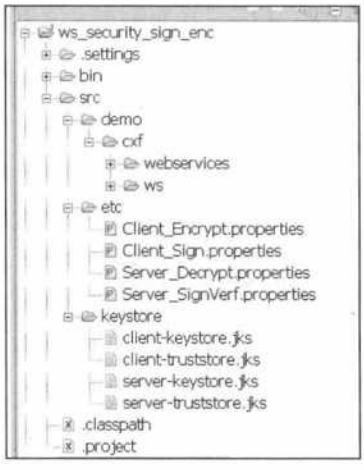


图 13-14　WS-Security 的 etc 目录和 keystore 目录

用于 WS-Security 还有两个目录，即 etc 目录和 keystore 目录。etc 目录是属性配置文件的目录，而加密文件在 keystore 目录下。可以看看 etc 目录配置的属性文件，如看签名属性——Client_Sign.properties，其内容如下：

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=storepassword
org.apache.ws.security.crypto.merlin.keystore.alias=clientx509v1
org.apache.ws.security.crypto.merlin.file=keystore/client-keystore.jks
```

该 Client_Sign.properties 属性文件说明采用的加密驱动器是 org.apache.ws.security.components.crypto.Merlin。merlin.keystore.type 的方式是 jks。merlin.keystore.password 的口令 storepassword。用户的别名是 clientx509v1。签名文件是当前程序目录下的 eystore/client-keystore.jks 文件。

本案例应用程序包括服务器端应用程序、客户端应用程序和 WS-Security 文件。下面分别介绍三部分：①服务器端点接口 SEI 和相关对象；②服务器端发布程序；③客户端调用程序。

（1）服务器端点接口（SEI）和相关对象

首先要介绍的是服务器端点接口（SEI）。服务器端应用程序主要包括 Project 组件、服务器启动类和 WS-Security 相关文件。

Project 组件包括这几个 WebService 包下的 Project 几个文件和 Server.java 文件。

创建服务器端点接口与 WS-Addressing 实现例子基本相似，都是创建一个充当 SEI 的 ProjectManager 接口和其实现类 ProjectManagerImpl。本案例的 ProjectManager 接口和其实现类

ProjectManagerImpl 与 WS-Addressing 实现例子的 ProjectManager 接口和 ProjectManagerImpl 相同。同时加入了一个继承 javax.xml.ws.Service 类的 ProjectManagerService 类，ProjectManagerService 类与 WS-Addressing 实现例子的 ProjectManagerService 类基本相同。

（2）服务器端发布程序

编写发布这个 Java 程序为 Web Services 的服务器端程序。创建了 SEI，要把这个 SEI 发布出去，还需要一个部署程序，即服务器端的发布程序。本案例采用 JAX-WS 规范来编写服务器端程序。创建一个 Server 类，启动这个服务，接着就可以发布 SEI 服务。

其中 Server.java 是服务器启动程序文件，其代码如下：

```
package demo.cxf.ws.security.server;

import java.net.URL;
import javax.xml.ws.Endpoint;
import java.util.HashMap;
import java.util.Map;

import org.apache.cxf.Bus;
import org.apache.cxf.bus.spring.SpringBusFactory;
import org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor;
import org.apache.cxf.ws.security.wss4j.WSS4JOutInterceptor;

import demo.cxf.webservices.*;

public class Server {
   private static final String WSU_NS
      =
"http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.
xsd";

   protected Server() throws Exception {
      System.out.println("Starting Server");
      ProjectManager projectManager = new ProjectManagerImpl();
      String address = "http://localhost:9000/ProjectManager";
      Endpoint e = Endpoint.publish(address, projectManager);
   }

   public static void main(String args[]) throws Exception {
      SpringBusFactory bf = new SpringBusFactory();
      URL busFile = Server.class.getResource("wssec-server.xml");
      Bus bus = bf.createBus(busFile.toString());

      Map<String, Object> outProps = new HashMap<String, Object>();
      outProps.put("action", "UsernameToken Timestamp Signature Encrypt");

      outProps.put("passwordType", "PasswordText");
      outProps.put("user", "serverx509v1");
      outProps.put("passwordCallbackClass", "demo.cxf.ws.security.server.UTPasswordCallback");

      //If you are using the patch WSS-194, then uncomment below two lines and
      //comment the above "user" prop line.
      //outProps.put("user", "Alice");
      //outProps.put("signatureUser", "serverx509v1");

      outProps.put("encryptionUser", "clientx509v1");
      outProps.put("encryptionPropFile", "etc/Server_SignVerf.properties");
```

```
        outProps.put("encryptionKeyIdentifier", "IssuerSerial");
        outProps.put("encryptionParts", "{Element}{" + WSU_NS + "}Timestamp;"
                    + "{Content}{http://schemas.xmlsoap.org/soap/envelope/}Body");

        outProps.put("signaturePropFile", "etc/Server_Decrypt.properties");
        outProps.put("signatureKeyIdentifier", "DirectReference");
        outProps.put("signatureParts", "{Element}{" + WSU_NS + "}Timestamp;"
                    + "{Element}{http://schemas.xmlsoap.org/soap/envelope/}Body");

        bus.getOutInterceptors().add(new WSS4JOutInterceptor(outProps));

        Map<String, Object> inProps = new HashMap<String, Object>();

        inProps.put("action", "UsernameToken Timestamp Signature Encrypt");
        inProps.put("passwordType", "PasswordDigest");
        inProps.put("passwordCallbackClass", "demo.cxf.ws.security.server.UTPasswordCallback");

        inProps.put("decryptionPropFile", "etc/Server_Decrypt.properties");
        inProps.put("encryptionKeyIdentifier", "IssuerSerial");

        inProps.put("signaturePropFile", "etc/Server_SignVerf.properties");
        inProps.put("signatureKeyIdentifier", "DirectReference");

        bus.getInInterceptors().add(new WSS4JInInterceptor(inProps));

        bf.setDefaultBus(bus);

        new Server();
        System.out.println("Server ready...");

        Thread.sleep(5 * 60 * 1000);

        bus.shutdown(true);
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

在代码中要求装载文件 wssec-server.xml，配置代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:cxf="http://cxf.apache.org/core"
    xmlns:wsa="http://cxf.apache.org/ws/addressing"
    xmlns:http="http://cxf.apache.org/transports/http/configuration"
    xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
    xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
    xsi:schemaLocation="
        http://cxf.apache.org/core
        http://cxf.apache.org/schemas/core.xsd
        http://cxf.apache.org/transports/http/configuration
        http://cxf.apache.org/schemas/configuration/http-conf.xsd
        http://schemas.xmlsoap.org/ws/2005/02/rm/policy
        http://schemas.xmlsoap.org/ws/2005/02/rm/wsrm-policy.xsd
        http://cxf.apache.org/ws/rm/manager
        http://cxf.apache.org/schemas/configuration/wsrm-manager.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
```

```
    <cxf:bus>
        <cxf:features>
            <cxf:logging />
            <wsa:addressing />
        </cxf:features>
    </cxf:bus>

    <http:conduit
        name="{{http://webservices.cxf.demo/}ProjectManagerPort.http-conduit">
        <http:client
            DecoupledEndpoint="http://localhost:9990/ProjectManager" />
    </http:conduit>
</beans>
```

在本案例中，引入了一个处理用户口令的类——UTPasswordCallback.java，代码如下：

```
package demo.cxf.ws.security.server;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;

import org.apache.ws.security.WSPasswordCallback;

public class UTPasswordCallback implements CallbackHandler {
   private Map<String, String> passwords = new HashMap<String, String>();

   public UTPasswordCallback() {
      passwords.put("Alice", "ecilA");
      passwords.put("abcd", "dcba");
      passwords.put("clientx509v1", "storepassword");
      passwords.put("serverx509v1", "storepassword");
   }

   public void handle(Callback[] callbacks) throws IOException, UnsupportedCallbackException {
      for (int i = 0; i < callbacks.length; i++) {
         WSPasswordCallback pc = (WSPasswordCallback)callbacks[i];

         String pass = passwords.get(pc.getIdentifier());
         if (pass != null) {
            pc.setPassword(pass);
            return;
         }
      }
      throw new IOException();
   }

   public void setAliasPassword(String alias, String password) {
      passwords.put(alias, password);
   }
}
```

在本案例中，服务器启动的服务地址是 http://localhost:9000/ProjectManager。服务器引入了

SEI 接口——ProjectManager 接口。这样 ProjectManager 的 Web Services 就暴露到了外部。可以在浏览器上输入 http://localhost:9000/ProjectManager?wsdl 来查看这个服务的内容。

（3）客户端调用程序

客户端应用程序包括 Web Services 的客户端应用程序和配置文件。

Web Services 的客户端应用程序——Client.java 的代码如下：

```
package demo.cxf.ws.security.client;

import java.lang.reflect.UndeclaredThrowableException;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;

import org.apache.cxf.Bus;
import org.apache.cxf.bus.spring.SpringBusFactory;
import org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor;
import org.apache.cxf.ws.security.wss4j.WSS4JOutInterceptor;

import demo.cxf.webservices.Project;
import demo.cxf.webservices.ProjectImpl;
import demo.cxf.webservices.ProjectManager;
import demo.cxf.webservices.ProjectManagerService;

public class Client {

    private static final String USER_NAME = System.getProperty("user.name");
    private static final String WSU_NS =
 "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd";

    private Client() {
    }

    public static void main(String args[]) throws Exception {
        try {
            SpringBusFactory bf = new SpringBusFactory();
            URL busFile = Client.class.getResource("wssec-client.xml");
            Bus bus = bf.createBus(busFile.toString());
            bf.setDefaultBus(bus);

            Map<String, Object> outProps = new HashMap<String, Object>();
            outProps.put("action", "UsernameToken Timestamp Signature Encrypt");

            outProps.put("passwordType", "PasswordDigest");
            outProps.put("user", "clientx509v1");

            // If you are using the patch WSS-194, then uncomment below two
            // lines and
            // comment the above "user" prop line.
            // outProps.put("user", "abcd");
            // outProps.put("signatureUser", "clientx509v1");

            outProps.put("passwordCallbackClass",
                    "demo.cxf.ws.security.client.UTPasswordCallback");

            outProps.put("encryptionUser", "serverx509v1");
            outProps.put("encryptionPropFile", "etc/Client_Encrypt.properties");
```

```
            outProps.put("encryptionKeyIdentifier", "IssuerSerial");
            outProps    .put(   "encryptionParts","{Element}{" + WSU_NS    +
"}Timestamp;"   + "{Content}{http://schemas.xmlsoap.org/soap/envelope/}Body");

            outProps.put("signaturePropFile", "etc/Client_Sign.properties");
            outProps.put("signatureKeyIdentifier", "DirectReference");
            outProps.put("signatureParts","{Element}{" + WSU_NS+ "}Timestamp;"
                    + "{Element}{http://schemas.xmlsoap.org/soap/envelope/}Body");

            bus.getOutInterceptors().add(new WSS4JOutInterceptor(outProps));

            Map<String, Object> inProps = new HashMap<String, Object>();

            inProps.put("action", "UsernameToken Timestamp Signature Encrypt");
            inProps.put("passwordType", "PasswordText");
            inProps.put("passwordCallbackClass",
                    "demo.cxf.ws.security.client.UTPasswordCallback");

            inProps.put("decryptionPropFile", "etc/Client_Sign.properties");
            inProps.put("encryptionKeyIdentifier", "IssuerSerial");

            inProps.put("signaturePropFile", "etc/Client_Encrypt.properties");
            inProps.put("signatureKeyIdentifier", "DirectReference");

            bus.getInInterceptors().add(new WSS4JInInterceptor(inProps));

            ProjectManagerService service = new ProjectManagerService();
            ProjectManager projectManager = service.getProjectManagerPort();

            // 项目的基本情况
            getGreet(projectManager);
            greetingWithParameter(projectManager);
            ……

            // allow aynchronous resends to occur
            Thread.sleep(30 * 1000);

            bus.shutdown(true);

        } catch (UndeclaredThrowableException ex) {
            ex.getUndeclaredThrowable().printStackTrace();
        } catch (Exception ex) {
            ex.printStackTrace();
        } finally {
            System.exit(0);
        }
    }

    // 调用 SEI 的 getGreet 方法
    private static void getGreet(ProjectManager projectManager)
            throws Exception {
        System.out.println(projectManager.greeting());
    }
    ……
}
```

配置文件 wssec-client.xml 的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:cxf="http://cxf.apache.org/core"
      xmlns:wsa="http://cxf.apache.org/ws/addressing"
      xmlns:http="http://cxf.apache.org/transports/http/configuration"
      xmlns:wsrm-policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
      xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
      xsi:schemaLocation="
         http://cxf.apache.org/core
         http://cxf.apache.org/schemas/core.xsd
         http://cxf.apache.org/transports/http/configuration
         http://cxf.apache.org/schemas/configuration/http-conf.xsd
         http://schemas.xmlsoap.org/ws/2005/02/rm/policy
         http://schemas.xmlsoap.org/ws/2005/02/rm/wsrm-policy.xsd
         http://cxf.apache.org/ws/rm/manager
         http://cxf.apache.org/schemas/configuration/wsrm-manager.xsd
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd">

   <cxf:bus>
      <cxf:features>
         <cxf:logging/>
         <wsa:addressing/>
      </cxf:features>
   </cxf:bus>

   <http:conduit
name="{http://webservices.cxf.demo/}ProjectManagerPort.http-conduit">
     <http:client DecoupledEndpoint="http://localhost:9990/ProjectManager"/>
   </http:conduit>

</beans>
```

还有一个处理口令的类 UTPasswordCallback.java，与服务器的 UTPasswordCallback.java 代码相同，这里省略。

本例首先创建一个服务名称为 javax.xml.ws.Service 的实现类，接着通过 addPort 方法来传入服务器的端口类、绑定方式和 Web Services 地址。最后通过 getPort 方法来获得服务器的 SEI 接口。这就是 Web Services 服务器接口 ProjectManager。后面就基于这个 ProjectManager 的 SEI 来实现各种业务操作。

4. 测试和运行结果

运行客户端程序，其输出结果如下：

```
Starting Client
……
ID: 22
Address: /ProjectManager
Response-Code: 200
Encoding: UTF-8
Content-Type: text/xml; charset=UTF-8
Headers: {Host=[localhost:9990], User-Agent=[Apache CXF 2.2.10], connection=[keep-alive],
transfer-encoding=[chunked], Pragma=[no-cache], Content-Type=[text/xml; charset=
UTF-8], content-type=[text/xml; charset=UTF-8], Cache-Control=[no-cache], Accept=[*/*]}
Payload:  <soap:Envelope  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
```

```
xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"><soap:Header><wsse:Security
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-se
cext-1.0.xsd" soap:mustUnderstand="1"><xenc:EncryptedKey xmlns:xenc="http://www.w3.
org/2001/04/xmlenc#" Id="EncKeyId-3ACF25D599055D57CD1298773936875110"><xenc:EncryptionMethod
Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5" /><ds:KeyInfo xmlns:ds="http://
www.w3.org/2000/09/xmldsig#">
<wsse:SecurityTokenReference  xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-secext-1.0.xsd"><ds:X509Data>
<ds:X509IssuerSerial>
<ds:X509IssuerName>CN=Mayank Mishra,OU=Dev,O=Apache,L=INDORE,ST=MP,C=IN</ds:X509IssuerName>
<ds:X509SerialNumber>1245002984</ds:X509SerialNumber>
</ds:X509IssuerSerial>
</ds:X509Data></wsse:SecurityTokenReference>
</ds:KeyInfo><xenc:CipherData><xenc:CipherValue>TCsRDMolC6brOtHmT6dOOo1JeVp3KydNc
ee1W1Nbf9kL/QGq6L59TXMATM+5Gu0ChCr2LLc8Im2PQAhXeC//ykYk8leeshuS59Adcgybnv3vLB+4Su
ysxjdn+xS8QPAq6/MkZopB3v4IecmWL5ensA1BffDTWh3yZMNGDx2z2D8=</xenc:CipherValue></xe
nc:CipherData><xenc:ReferenceList><xenc:DataReference  URI="#EncDataId-131"  />
<xenc:DataReference URI="#EncDataId-132" /></xenc:ReferenceList></xenc:EncryptedKey>
<wsse:BinarySecurityToken  xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-
200401-wss-wssecurity-secext-1.0.xsd"  xmlns:wsu="http://docs.oasis-open.org/wss/
2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" EncodingType="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#Base64Binary"
ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-pro
file-1.0#X509v3" wsu:Id="CertId-3ACF25D599055D57CD1298773936859106">
MIICNjCCAZ8CBEo1PQcwDQYJKoZIhvcNAQEEBQAwYjELMAkGA1UEBhMCSU4xCzAJBgNVBAgTAk1QMQ8wD
QYDVQQHEwZJTkRPUkUxDzANBgNVBAoTBkFwYWNoZTEMMAoGA1UECxMDRGV2MRYwFAYDVQQDEw1NYXlhbm
sgTWlzaHJhMB4XDTA5MDYxNDE4MTAxNVoXDTE5MDYxMjE4MTAxNVowYjELMAkGA1UEBhMCSU4xCzAJBgN
VBAgTAk1QMQ8wDQYDVQQHEwZJTkRPUkUxDzANBgNVBAoTBkFwYWNoZTEMMAoGA1UECxMDRGV2MRYwFAYD
VQQDEw1NYXlhbmsgTWlzaHJhMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDAiXozxqFqRmn5d7JK7
+iOL0n0IvoxsphntXk1OEzalwFITzOr3IbnwDWDI5FIUidq/QmpqijEQmCX2Ne4CTTxDVeSNvxWLc3Kjm
sECDwrKulmireRVdg/T0K8JZ0ezZxP35+dM0o3Evq5ocekJJ84Em46WrLH7DR2yA1DryowXQIDAQABMA0
GCSqGSIb3DQEBBAUAA4GBAEpO4eEh//9weRuBlr8H8/oJSNtt9DywNwIqBtn+7i7hWX6pz+UflsH+B+hY
1HZDKPMyl8Wxh6jduRdRRh2Hgi+0yUDSmWxJXgF5NnwaKf2EN53opm/S4X/ZjbO31Y8vNWHFMUhAFFxgE
2bV3LaN3Lwi6muBpdx6KVS8hvN/SzBr</wsse:BinarySecurityToken><ds:Signature
xmlns:ds="http://www.w3.org/2000/09/xmldsig#" Id="Signature-129">
<ds:SignedInfo>
<ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
<ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
<ds:Reference URI="#Timestamp-128">
<ds:Transforms>
<ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
</ds:Transforms>
<ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
<ds:DigestValue>DhWi7GGBK2rzZXty/cg7+kPAlBo=</ds:DigestValue>
</ds:Reference>
<ds:Reference URI="#id-130">
<ds:Transforms>
<ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
</ds:Transforms>
<ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
<ds:DigestValue>UmdYN32xlTKMswHy38dZ0G9w+/Y=</ds:DigestValue>
</ds:Reference>
</ds:SignedInfo>
<ds:SignatureValue>
mjsfzdaa58TaRW9vMPYiTXgeHA7/cevGVaNmDjWfjhx2vqXShYHokJKypIjm+XQgxJm3u00fM0J8
+2oRmzru9v3RgaaJeknLb4xTfP0i1nTvCOGzXH/1TYXCjvD9h6NWuaAZ7VEGE47611zIEGJn/ydk
qgFYiGvRzzftcBH8ki8=
</ds:SignatureValue>
<ds:KeyInfo Id="KeyId-3ACF25D599055D57CD1298773936859107">
```

```
<wsse:SecurityTokenReference
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-se
cext-1.0.xsd" xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-utility-1.0.xsd"    wsu:Id="STRId-3ACF25D599055D57CD1298773936859108">
<wsse:Reference  xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
wss-wssecurity-secext-1.0.xsd"   URI="#CertId-3ACF25D599055D57CD1298773936859106"
ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-pro
file-1.0#X509v3" /></wsse:SecurityTokenReference>
</ds:KeyInfo>
</ds:Signature><xenc:EncryptedData xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
Id="EncDataId-131" Type="http://www.w3.org/2001/04/xmlenc#Element"><xenc:EncryptionMethod
Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"  /><ds:KeyInfo  xmlns:ds=
"http://www.w3.org/2000/09/xmldsig#">
<wsse:SecurityTokenReference  xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-secext-1.0.xsd"><wsse:Reference xmlns:wsse=
"http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
URI="#EncKeyId-3ACF25D599055D57CD1298773936875110" /></wsse:SecurityTokenReference>
</ds:KeyInfo><xenc:CipherData><xenc:CipherValue>hiKllwIvQOpzS765F8Dn0cVbFkFAQTTF6
NDOBk4s4UuxzS9Y106cuo6P+Sbx/rjvWIOwDpTHqoWE
I3s61Egxe6UP/gMQoHPS/sXSPS8xOsvd5VBY3FRX6GpRtlY1wuFIWxn2LU32QgK9TicN7stRCLvF
2bdBfd0d9ISuWNwlSQBfCKUp2Vsu8155yCHw+59qfdHWnQguvyT03Tth6AX8B1bcfijySzgD153c
6kO3GAAb/7cX+uK99cNCgcsakZRRPM6j2xR3vf0Shv0PAEfdzxpY1lVFLLRx4h/7O91pKJN6q4Js
x22M4pN5p/28unSKplk+rehxlIU/pGLxrgC/6OvGTyoAh6soDYny4utR18Ww2RTwiZc0/DSo+OgZ
xBPHk9tyeNQea64TeQO1Uzv7uviUvfxnn3ZMUR1ja/UzvC7KIATDFtsiwPl2Kx7t1IbF7l2+X7Fg
bIuWa/hl2Zlfg+9IDuYLMYPqdDgQ3wFbzwjXpIVDKdacOl547sjEx8E+sqAykIlXDc/rOQWg1ftv
biA74rhk9lLGZO/Bpfn8cL3Hofd+T1j7I8G/GHNOdQDr/ciEurw/d7r4o4ueygVeQnmdykUWtpW+
5rLrnTZ1XRP+mp1mqqj7W/l84dK3rrDF</xenc:CipherValue></xenc:CipherData></xenc:Encry
ptedData><wsse:UsernameToken  xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-secext-1.0.xsd"  xmlns:wsu="http://docs.oasis-open.org/
wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"  wsu:Id="UsernameToken-127">
<wsse:Username>serverx509v1</wsse:Username><wsse:Password  Type="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordText">st
orepassword</wsse:Password></wsse:UsernameToken></wsse:Security><Action xmlns=
"http://www.w3.org/2005/08/addressing">http://webservices.cxf.demo/ProjectManager
/getProjectNameByIDResponse</Action><MessageID  xmlns="http://www.w3.org/2005/08/
addressing">urn:uuid:47e66391-b76c-41ee-a8c5-d68cf2d67293</MessageID><To xmlns=
"http://www.w3.org/2005/08/addressing">http://localhost:9990/ProjectManager</To><
RelatesTo  xmlns="http://www.w3.org/2005/08/addressing">urn:uuid:24f11411-13ad-4489-
b793-204f21fd7865</RelatesTo></soap:Header><soap:Body xmlns:wsu="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"  wsu:Id="id-130">
<xenc:EncryptedData xmlns:xenc="http://www.w3.org/2001/04/xmlenc#" Id="EncDataId-132"
Type="http://www.w3.org/2001/04/xmlenc#Content"><xenc:EncryptionMethod
Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"  /><ds:KeyInfo  xmlns:ds=
"http://www.w3.org/2000/09/xmldsig#">
<wsse:SecurityTokenReference  xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-secext-1.0.xsd"><wsse:Reference   xmlns:wsse="http://
docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"  URI=
"#EncKeyId-3ACF25D599055D57CD1298773936875110" /></wsse:SecurityTokenReference>
</ds:KeyInfo><xenc:CipherData><xenc:CipherValue>qfvKYEcWQ+ZsU408Z2GcVroDmt2f5Uqlr
B6CLL5hnFw5CxsWgW/+ZFP3n267Lso0sGOOXqX9RcDP
0hRc0IyxsOAY4dwL/AHJQo/nKqZoBI+2ZJafjEvNez9lTBQicWNQ41yBTkQfLLy+FQybkICtkQq1
fdE7sbOA82Ab6tGU9xnLSIixRk54+lGvsaZznFsaWFxy2uvi7lzztJk89U+c9A==</xenc:CipherValu
e></xenc:CipherData></xenc:EncryptedData></soap:Body></soap:Envelope>
--------------------------------------
New ProjectName
……
```

服务器端程序输出结果如下：

```
……
ID: 13
Response-Code: 202
Encoding: UTF-8
Content-Type: text/xml
Headers: {}
Payload:  <soap:Envelope  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"><soap:Header><wsse:Security
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-se
cext-1.0.xsd" soap:mustUnderstand="1"><xenc:EncryptedKey xmlns:xenc=
"http://www.w3.org/2001/04/xmlenc#" Id="EncKeyId-3ACF25D599055D57CD1298773937187125">
<xenc:EncryptionMethod  Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"  />
<ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
<wsse:SecurityTokenReference  xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-secext-1.0.xsd"><ds:X509Data>
<ds:X509IssuerSerial>
<ds:X509IssuerName>CN=Mayank Mishra,OU=Dev,O=Apache,L=INDORE,ST=MP,C=IN</ds:X509IssuerName>
<ds:X509SerialNumber>1245002984</ds:X509SerialNumber>
</ds:X509IssuerSerial>
</ds:X509Data></wsse:SecurityTokenReference>
</ds:KeyInfo><xenc:CipherData><xenc:CipherValue>CPjS2oLzGSvFh9gyT2gT7b2PhnDUO+DY1
reTbGyP14WABGAtxkeLwCFQpg6WqZajDSnj6wbJvKiEEEohPkBwM8rrhxX5Px1kNbJMEUOvc6T89zLgms
L5p3/5+ZSf6GDkLa26zgyVvpKcLo3xgRGfbOo/cOIVOl4KaLk38j1XQwg=</xenc:CipherValue></xe
nc:CipherData><xenc:ReferenceList><xenc:DataReference URI="#EncDataId-149" />
<xenc:DataReference URI="#EncDataId-150" /></xenc:ReferenceList></xenc:EncryptedKey>
<wsse:BinarySecurityToken  xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-
200401-wss-wssecurity-secext-1.0.xsd"  xmlns:wsu="http://docs.oasis-open.org/wss/
2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" EncodingType="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#Base64Binary"
ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-pro
file-1.0#X509v3" wsu:Id="CertId-3ACF25D599055D57CD1298773937171121">
MIICNjCCAZ8CBEo1PQcwDQYJKoZIhvcNAQEEBQAwYjELMAkGA1UEBhMCSU4xCzAJBgNVBAgTAk1QMQ8wD
QYDVQQHEwZJTkRPUkUxDzANBgNVBAoTBkFwYWNoZTEMMAoGA1UECxMDRGV2MRYwFAYDVQQDEw1NYXlhbm
sgTWlzaHJhMB4XDTA5MDYxNDE4MTAxNVoXDTE5MDYxMjE4MTAxNVowYjELMAkGA1UEBhMCSU4xCzAJBgN
VBAgTAk1QMQ8wDQYDVQQHEwZJTkRPUkUxDzANBgNVBAoTBkFwYWNoZTEMMAoGA1UECxMDRGV2MRYwFAYD
VQQDEw1NYXlhbmsgTWlzaHJhMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDAiXozxqFqRmn5d7JK7
+iOL0n0IvoxsphntXk1OEzalwFITzOr3IbnwDWDI5FIUidq/QmpqijEQmCX2Ne4CTTxDVeSNvxWLc3Kjm
sECDwrKulmireRVdg/T0K8JZ0ezZxP35+dM0o3Evq5ocekJJ84Em46WrLH7DR2yA1DryowXQIDAQABMA0
GCSqGSIb3DQEBBAUAA4GBAEpO4eEh//9weRuBlr8H8/oJSNtt9DywNwIqBtn+7i7hWX6pz+UflsH+B+hY
1HZDKPMyl8Wxh6jduRdRRh2Hgi+0yUDSmWxJXgF5NnwaKf2EN53opm/S4X/ZjbO31Y8vNWHFMUhAFFxgE
2bV3LaN3Lwi6muBpdx6KVS8hvN/SzBr</wsse:BinarySecurityToken><ds:Signature
xmlns:ds="http://www.w3.org/2000/09/xmldsig#" Id="Signature-147">
<ds:SignedInfo>
<ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
<ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
<ds:Reference URI="#Timestamp-146">
<ds:Transforms>
<ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
</ds:Transforms>
<ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
<ds:DigestValue>Q8yOTNDKgfmkuKT5f5VSrN9RWrA=</ds:DigestValue>
</ds:Reference>
<ds:Reference URI="#id-148">
<ds:Transforms>
<ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
</ds:Transforms>
<ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
<ds:DigestValue>L0ZFOrEtoWK0TcM2BOIncsyIU0c=</ds:DigestValue>
```

```
</ds:Reference>
</ds:SignedInfo>
<ds:SignatureValue>
I83n6l6XPJljurwh/qqziDtJbHPEcRRpnKHAtCEBrWCKt3JCM/Bb0wQfXPyQetO/SaUq6ayXgNG6
Ix9oPIggmsG07gYhcOrUAesYa7Dr8AoOLnNnMrXylaU0qUzJ5fN5uDob/MADH7sBIpEIam74GR5/
nAqVqXHxeUjLjmONxlc=
</ds:SignatureValue>
<ds:KeyInfo Id="KeyId-3ACF25D599055D57CD1298773937171122">
<wsse:SecurityTokenReference  xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-secext-1.0.xsd" xmlns:wsu="http://docs.oasis-open.
org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
wsu:Id="STRId-3ACF25D599055D57CD1298773937171123"><wsse:Reference
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-se
cext-1.0.xsd" URI="#CertId-3ACF25D599055D57CD1298773937171121" ValueType="http://
docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3"
/></wsse:SecurityTokenReference>
</ds:KeyInfo>
</ds:Signature><xenc:EncryptedData xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
Id="EncDataId-149" Type="http://www.w3.org/2001/04/xmlenc#Element"><xenc:EncryptionMethod
Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"  /><ds:KeyInfo  xmlns:ds=
"http://www.w3.org/2000/09/xmldsig#">
<wsse:SecurityTokenReference  xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-secext-1.0.xsd"><wsse:Reference
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-se
cext-1.0.xsd" URI="#EncKeyId-3ACF25D599055D57CD1298773937187125" />
</wsse:SecurityTokenReference>
</ds:KeyInfo><xenc:CipherData><xenc:CipherValue>pgIRg4LCAmi7LPWIIootfsEGR+peNTcHq
GlsV7RDGUTfi2pZ2a0GnHJqz8jWDVlAjOxc6Bg9x1/6
9shlyg+ElvzZA29lG/31S4tWCHzN50JvFN2eP/vtQAeBEZ7UsmyIbkQ44nJxMx09X0CKbJDh60uu
lWDqM5WAE8KQqFWe25juvYUZ3xHU06Ajl6EkxS3Dwn49yQrNrtr5iEiECCkPBVExhSo4xOQ3XZG2
Xv52vBZaa/wKtXcySfDskcgTa8CEY3GhTmV0ikVfAZhThBdyYFvHCpxUZEWLOny/bsjPzG9ePrX3
tb6MlAkVlUEIJvwG8YLQmH/aefUD7LOXvoBRnIaFAPmNLbeb42RpiSJbGj8x95SlvOJgoJ8iIqfd
JXic+DGQrGNMMXsWkkubsNEuSX+2H0X+IP30DClk/XrTIsPBFAeiAWz9HdlBGUyU/J5NJ37Xhc8/
5Wi72MfMspsZcoB4d8YZmM6OfuNdRCmaJ8EGXWVW2hiNf5C0CIaeHTXQrX6asZuRIWL/9OvGGGKh
DZ/FDhJ4YNUe5JAoWDAcH92ohvKyTWjr06a1XPguNFQ5qZaSWW2lmrkchPqm7UuklfIcQWY2aH41
7CdVGLadSPGWBruQZOQ3TIXJAi8zc7Yx</xenc:CipherValue></xenc:CipherData></xenc:Encry
ptedData><wsse:UsernameToken
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-se
cext-1.0.xsd"  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-utility-1.0.xsd"  wsu:Id="UsernameToken-145"><wsse:Username>serverx509v1
</wsse:Username><wsse:Password  Type="http://docs.oasis-open.org/wss/2004/01/oasis-
200401-wss-username-token-profile-1.0#PasswordText">storepassword</wsse:Password>
</wsse:UsernameToken></wsse:Security><MessageID xmlns="http://www.w3.org/2005/08/
addressing">urn:uuid:f0bd2c36-3ea8-480b-bb71-c623332e9292</MessageID><To xmlns=
"http://www.w3.org/2005/08/addressing">http://www.w3.org/2005/08/addressing/anony
mous</To><ReplyTo xmlns="http://www.w3.org/2005/08/addressing"><Address>
http://www.w3.org/2005/08/addressing/none</Address></ReplyTo></soap:Header><soap:
Body  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
utility-1.0.xsd"  wsu:Id="id-148"><xenc:EncryptedData  xmlns:xenc="http://www.w3.
org/2001/04/xmlenc#" Id="EncDataId-150" Type="http://www.w3.org/2001/04/xmlenc#Content">
<xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc" />
<ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
<wsse:SecurityTokenReference  xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-secext-1.0.xsd"><wsse:Reference xmlns:wsse=
"http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
URI="#EncKeyId-3ACF25D599055D57CD1298773937187125" /></wsse:SecurityTokenReference>
</ds:KeyInfo><xenc:CipherData><xenc:CipherValue>n0qyLuH7lppXY1EsZG9Pxz+DdNKq3Kh3s
X37Ywtdn9E=</xenc:CipherValue></xenc:CipherData></xenc:EncryptedData></soap:Body>
</soap:Envelope>
```

```
---------------------------------------
----invoking removeProject, Project ID is: New ProjectID
……
```

5. 其他内容

上述的所有调用都基于 JAX-WS 规范。

本例代码参见配套资源代码中“第 13 章\ws_security_sign_enc”文件夹。

13.5 Apache CXF 支持 WS-SecurityPolicy

本节简单介绍一下 WS-SecurityPolicy 规范概念，以及 Apache CXF 框架使用 WS-SecurityPolicy 规范的方法和例子。

13.5.1 WS-SecurityPolicy 简介

WS-SecurityPolicy 规范以 WS-Policy 规范为基础，定义了可用于指定安全相关策略的 XML 元素。由于这些元素允许 Web Services 明确地指定其策略，因此它们被称为声明。由 WS-SecurityPolicy 规范定义的声明包括以下几种。

- <SecurityToken> 允许 Web Services 指定 WS-SecurityPolicy 规范将接受哪些类型的安全令牌以及它所信任的令牌发行方（可选）。
- <Integrity> 允许 Web Services 指定 WS-SecurityPolicy 规范将接受的有关数字签名的多种选项，例如，用于创建消息摘要的算法。
- <Confidentiality> 允许 Web Services 指定如何执行加密的选项，例如，使用的加密算法。
- <Visibility> 允许 Web Services 指定消息的哪些部分必须进行解密。

WS-SecurityPolicy 规范没有定义如何准确地从设计者向任何感兴趣的客户端传输安全策略。表示策略的 XML 元素可以通过 SOAP 消息发送，也可以通过其他方式传输。但是 WS-Policy 规范定义了一个通用的<Policy>元素，它可以包含一个或多个策略声明。以下是一个简单的示例，它说明了对于要求使用 Kerberos 票据进行身份验证的 Web Services。包括<SecurityToken> 声明的形式：

```
<wsp:Policy
 xmlns:wsp="http://schemas.xmlsoap.org/ws/2002/12/policy
 xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext">
   <wsse:SecurityToken wsp:Usage="wsp:Required">
      <TokenType> wsse:Kerberosv5ST</TokenType>
   </wsse:SecurityToken>
</wsp:Policy>
```

该<Policy>元素指定了在 WS-Policy 规范和 WS-Security 规范中定义的命名空间，并且只包含<SecurityToken>子元素。该子元素的使用被标记为 Required，这意味着 Web Services 需要遵守这个策略。换句话说，正如元素所指示的那样，这个特定的 Web Services 现在只接受 Kerberos 票据作为身份验证的基础。

单个的<Policy>元素可以包含有关安全令牌、完整性以及其他方面的策略，还有几个其他选择，包括指定相关首选项的方法。例如，Web Services 可以定义一个<SecurityToken>，允许客户

端使用 Kerberos 或 X.509 证书进行<SecurityToken>的身份验证，然后指出优先使用 Kerberos。但是，无论使用哪种方案，策略值都是显而易见的。通过提供一种明确的方法以便 Web Services 声明其安全要求，它们使客户端能够明白要访问该服务必须做的事情。

WS-SecurityPolicy 规范的定义可以在 http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-os.html 查看。

13.5.2　Apache CXF 使用 WS-SecurityPolicy 的配置

Apache CXF 框架介绍了使用 WS-SecurityPolicy 规范来配置 WSS4J 而不是在 WS-Security 规范页面上进行说明。然而，WS-Security 页面内容仍然适用并且很重要。WS-SecurityPolicy 规范只是提供了一个更方便、更标准的方式来配置和控制安全要求。随着 WSDL 文件的安全要求作为 WS-Policy 的配置信息块，其他工具如.NET，可以很容易地知道如何配置与 Apache CXF 服务进行互操作。

注意： 在这一点上，WS-SecurityPolicy 仅支持适用于“WSDL 优先”的应用情景。在 WS-SecurityPolicy 配置信息块只能从 WSDL 中拉出。在未来，Apache 计划实现代码优先的功能，但现阶段只有 WSDL 优先是可以使用的。

1. 启用 WS-SecurityPolicy

在 Apache CXF 框架中，如果 cxf-rt-ws-policy 和 cxf-rt-ws-security 已经放置到了类路径下，那 WS-SecurityPolicy 就自动启用了。由于整个安全运行是在策略的推动下，唯一的要求就是策略和安全策略引擎可用。

如果开发者使用的是全部的“bundle” jar，所有的安全和策略的内容都已经包括在内了。

2. 策略介绍

随着 WS-SecurityPolicy 绑定和操作引用 WSDL 中 WS-Policy 部分，该部分描述了与该服务交互的基本安全要求。WS-SecurityPolicy 规范允许指定非对称/对称密钥、加密传输、数据头和信息体进行加密或签名、是否签署然后加密或加密然后签署、是否包括时间戳、是否使用派生密钥，等等。基本上，WS-SecurityPolicy 规范描述了 WSDL 内部服务的交互中，什么活动是安全所必需的。

但是，WS-SecurityPolicy 配置信息块并不包括“一切”，也就是需要一个在运行时能够创建信息和任务的功能。WS-SecurityPolicy 配置信息块默认情况下不描述这些内容，如密钥存储位置、用户名和密码等。这些需要在运行时增加 WS-SecurityPolicy 配置信息块来进行配置。

3. 配置额外的性能

针对 Apache CXF 框架中，有几个额外的属性需要设置，以提供额外的运行信息。

4. 通过 Spring 框架来进行配置

Spring 框架很容易为客户端或服务端点属性配置该属性——使用 SOAP 客户端，后端的 Web Services 提供者。

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
  http://cxf.apache.org/jaxws
  http://cxf.apache.org/schemas/jaxws.xsd">

  <jaxws:client name="{http://cxf.apache.org}MyPortName" createdFromAPI="true">
    <jaxws:properties>
      <entry key="ws-security.callback-handler" value="interop.client.
KeystorePasswordCallback"/>
      <entry key="ws-security.signature.properties" value="etc/client.properties"/>
      <entry key="ws-security.encryption.properties" value="etc/service.properties"/>
      <entry key="ws-security.encryption.username" value="servicekeyalias"/>
    </jaxws:properties>
  </jaxws:client>

</beans>
```

对于上面的 jaxws:client 的 name 属性，是使用 WSDL 命名空间对应 WSDL 服务所需的 wsdl:port 元素的名称属性，示例如下：

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jaxws="http://cxf.apache.org/jaxws"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
  http://cxf.apache.org/jaxws
  http://cxf.apache.org/schemas/jaxws.xsd">

  <jaxws:endpoint
    id="MyService"
    address="https://localhost:9001/MyService"
    serviceName="interop:MyService"
    endpointName="interop:MyServiceEndpoint"
    implementor="com.foo.MyService">

    <jaxws:properties>
      <entry key="ws-security.callback-handler" value="interop.client.UTPasswordCallback"/>
      <entry key="ws-security.signature.properties" value="etc/keystore.properties"/>
      <entry key="ws-security.encryption.properties" value="etc/truststore.properties"/>
      <entry key="ws-security.encryption.username" value="useReqSigCert"/>
    </jaxws:properties>

  </jaxws:endpoint>
</beans>
```

5. 通过 API 来配置

配置客户端属性包括设置客户端的 RequestContext 的属性：

```
Map<String, Object> ctx = ((BindingProvider)port).getRequestContext();
ctx.put("ws-security.encryption.properties", properties);
port.echoString("hello");
```

13.5.3　Apache CXF 的 WS-SecurityPolicy 的实现例子

本程序演示 Apache CXF 遵循 WS-SecurityPolicy 规范的客户端和服务器端调用，客户端采用 Apache CXF 的代码优先（Java First）前端应用模式。采用 POJO 对象作为中间传输对象。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，都使用 POJO 对象实现客户端和服务器端的数据交换。后台服务器端为遵循 JAX-WS 规范的 Endpoint 来实现，采用 Spring IoC 注入的客户端来实现。前端访问程序采用遵循 JAX-WS 规范的 Service 方式来实现。通过在 WSDL 文件中加入 WS-SecurityPolicies，然后在服务器和客户端实现 WS-SecurityPolicy 功能。本程序实现的结构如图 13-15 所示。

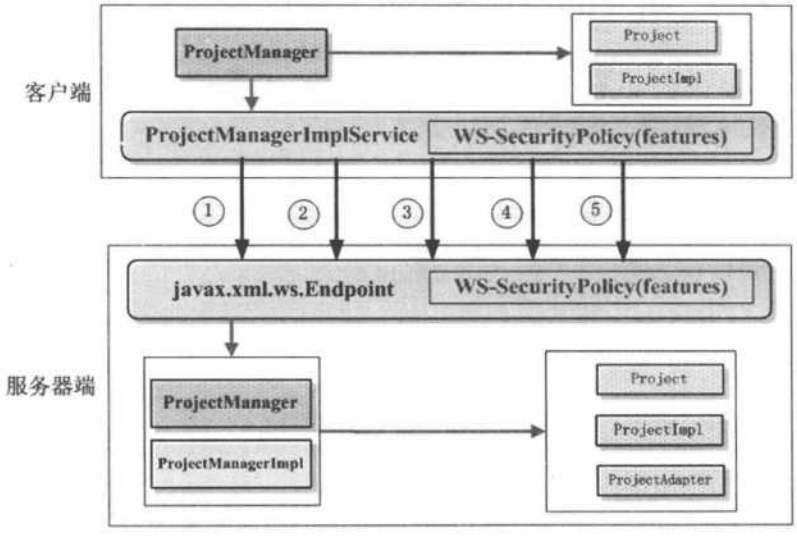


图 13-15　遵循 WS-SecurityPolicies 规范的客户端和服务器端调用模式图

其中：① 表示客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。在客户端和服务端的 SEI 都是 ProjectManager 接口。

2. 应用程序案例实现步骤

创建 WS-SecurityPolicies 实现 Web Services 程序的流程步骤如图 13-16 所示。

流程图说明，要实现以下步骤：

（1）创建一个 Java 项目。

（2）创建一个服务器端点接口（SEI）及其相关对象和注释。

（3）配置签名文件和加密文件。

（4）编写发布这个 Java 程序为 Web Services 的服务器端程序和服务器配置文件。

（5）编写 Web Services 的客户端应用程序和客户端配置文件。

（6）启动服务器程序，对外发布 Web Services。

（7）客户端调用 Web Services。

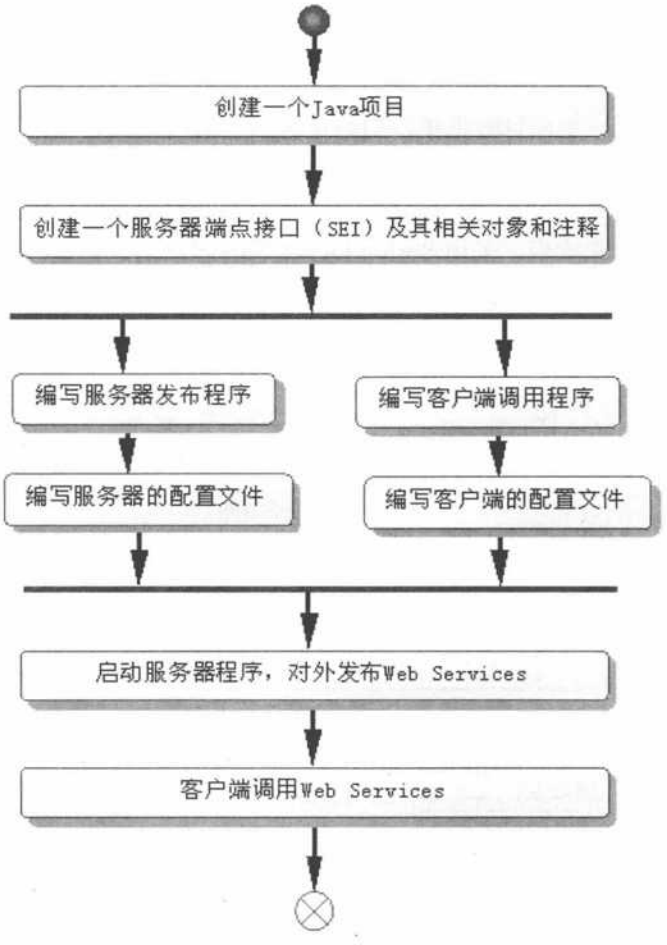


图 13-16　ws_security_ut_policy 应用程序案例实现步骤流程图

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 13-17 所示。

包目录包括三个部分，第一个包目录 demo.cxf.ws.securitypolicy.server 是服务器程序目录。第二个包目录 demo.cxf.ws.securitypolicy.client 是客户端程序目录。第三个包目录 demo.cxf.webservices 是服务器端点接口 SEI 和相关对象等公用部分，服务器端和客户端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口，服务器端和客户端都必须引用。但服务器端和客户端对公用部分的引用是不同的。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口，而服务器端程序是接口和实现类都要引用。

在 WS-SecurityPolicy 例子程序还有两个目录，即 certs 目录和 wsdl 目录。etc 目录是属性配置文件的目录，而加密文件在 keystore 目录下，其结构如图 13-18 所示。

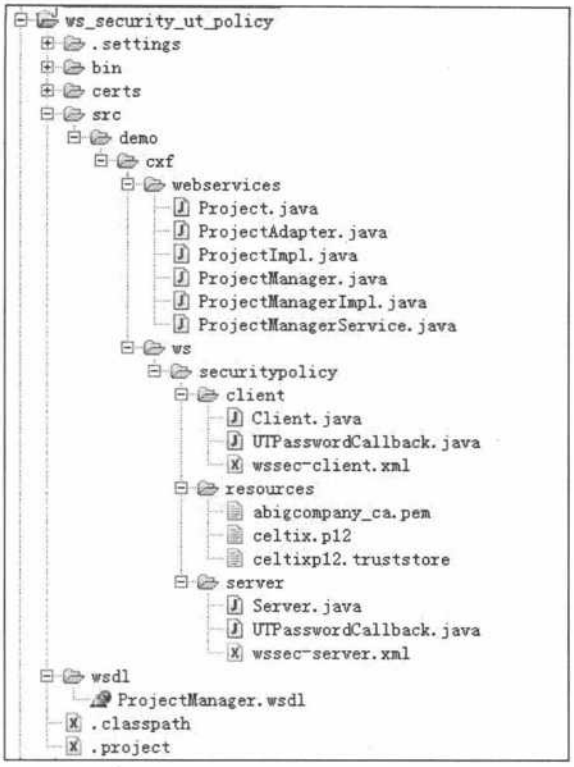


图 13-17　ws_security_ut_policy 程序结构

图 13-18　WS-Security 的 certs 目录和 wsdl 目录

下面分别介绍三部分：①服务器端点接口 SEI 和相关对象；②服务器端发布程序；③客户端调用程序。

（1）服务器端点接口（SEI）和相关对象

首先要介绍的是服务器端点接口（SEI）。

服务器端应用程序主要包括 Project 组件和 WS-SecurityPolicy 相关文件。

Project 组件包括这几个 WebService 包下的 Project 几个文件和 Server.java 文件。

创建服务器端点接口与 WS-Addressing 实现例子基本相似，都是创建一个充当 SEI 的 ProjectManager 接口和其实现类 ProjectManagerImpl。本案例的 ProjectManager 接口和其实现类 ProjectManagerImpl 与 WS-Addressing 实现例子的 ProjectManager 接口和 ProjectManagerImpl 相同。同时加入了一个继承 javax.xml.ws.Service 类的 ProjectManagerService 类，ProjectManagerService 类与 WS-Addressing 实现例子的 ProjectManagerService 类基本相同。

（2）服务器端发布程序

编写发布这个 Java 程序为 Web Services 的服务器端程序。创建了 SEI，要把这个 SEI 发布出去，还需要一个部署程序，即服务器端的发布程序，本案例采用 JAX-WS 规范来编写服务器端程序。创建一个 Server 类，启动这个服务，接着就可以发布 SEI 服务。其中 Server.java 是服务器启动程序文件。

```
package demo.cxf.ws.securitypolicy.server;

import java.net.URL;
```

```
import org.apache.cxf.Bus;
import org.apache.cxf.bus.spring.SpringBusFactory;

public class Server {
    protected Server() throws Exception {
        System.out.println("Starting Server");
        SpringBusFactory bf = new SpringBusFactory();
        URL busFile = Server.class.getResource("wssec-server.xml");
        Bus bus = bf.createBus(busFile.toString());
        bf.setDefaultBus(bus);
    }

    public static void main(String args[]) throws Exception {
        System.out.println();
        new Server();
        System.out.println("Server ready...");

        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

在代码中要求装载文件 wssec-server.xml，配置代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:cxf="http://cxf.apache.org/core"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:sec="http://cxf.apache.org/configuration/security"
    xmlns:http="http://cxf.apache.org/transports/http/configuration"
    xmlns:httpj="http://cxf.apache.org/transports/http-jetty/configuration"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="
          http://cxf.apache.org/core
          http://cxf.apache.org/schemas/core.xsd
          http://cxf.apache.org/configuration/security
          http://cxf.apache.org/schemas/configuration/security.xsd
          http://cxf.apache.org/jaxws
          http://cxf.apache.org/schemas/jaxws.xsd
          http://cxf.apache.org/transports/http/configuration
          http://cxf.apache.org/schemas/configuration/http-conf.xsd
          http://cxf.apache.org/transports/http-jetty/configuration
          http://cxf.apache.org/schemas/configuration/http-jetty.xsd
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <cxf:bus>
        <cxf:features>
            <cxf:logging />
        </cxf:features>
    </cxf:bus>

    <jaxws:endpoint id="server" endpointName="s:ProjectManagerPort"
        serviceName="s:ProjectManagerService"
        implementor="demo.cxf.webservices.ProjectManagerImpl"
        address="https://localhost:9001/ProjectManager"
        wsdlLocation="wsdl/ProjectManager.wsdl"
```

```
        depends-on="ClientAuthHttpsSettings"
        xmlns:s="http://webservices.cxf.demo/">


        <jaxws:properties>
            <entry key="ws-security.callback-handler"
                value="demo.cxf.ws.securitypolicy.server.UTPasswordCallback" />
        </jaxws:properties>
    </jaxws:endpoint>

    <httpj:engine-factory id="ClientAuthHttpsSettings">
        <httpj:engine port="9001">
            <httpj:tlsServerParameters>
                <sec:keyManagers keyPassword="password">
                    <sec:keyStore type="JKS" password="password"
                        file="certs/cherry.jks" />
                </sec:keyManagers>
                <sec:trustManagers>
                    <sec:keyStore type="JKS" password="password"
                        file="certs/truststore.jks" />
                </sec:trustManagers>
                <sec:cipherSuitesFilter>
                    <sec:include>.*_EXPORT_.*</sec:include>
                    <sec:include>.*_EXPORT1024_.*</sec:include>
                    <sec:include>.*_WITH_DES_.*</sec:include>
                    <sec:include>.*_WITH_NULL_.*</sec:include>
                    <sec:exclude>.*_DH_anon_.*</sec:exclude>
                </sec:cipherSuitesFilter>
                <sec:clientAuthentication want="true" required="true" />
            </httpj:tlsServerParameters>
        </httpj:engine>
    </httpj:engine-factory>
</beans>
```

在本案例中，引入了一个处理用户口令的类——UTPasswordCallback.java，代码如下：

```
package demo.cxf.ws.securitypolicy.server;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;

import org.apache.ws.security.WSPasswordCallback;

public class UTPasswordCallback implements CallbackHandler {

    private Map<String, String> passwords =
        new HashMap<String, String>();

    public UTPasswordCallback() {
        passwords.put("Alice", "ecilA");
        passwords.put("abcd", "dcba");
    }

    public void handle(Callback[] callbacks) throws IOException, UnsupportedCallbackException {
```

```
        for (int i = 0; i < callbacks.length; i++) {
            WSPasswordCallback pc = (WSPasswordCallback)callbacks[i];

            String pass = passwords.get(pc.getIdentifier());
            if (pass != null) {
                pc.setPassword(pass);
                return;
            }
        }
        throw new IOException();
    }

    public void setAliasPassword(String alias, String password) {
        passwords.put(alias, password);
    }
}
```

在本案例中，引入了一个 WSDL 文件——ProjectManager.wsdl，代码如下：

```
<?xml version='1.0' encoding='UTF-8'?>
<wsdl:definitions name="ProjectManagerImplService"
    targetNamespace="http://webservices.cxf.demo/"
    xmlns:ns1="http://schemas.xmlsoap.org/soap/http"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://webservices.cxf.demo/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <wsdl:types>
        <xs:schema elementFormDefault="unqualified"
            targetNamespace="http://webservices.cxf.demo/" version="1.0"
            xmlns:tns="http://webservices.cxf.demo/"
            xmlns:xs="http://www.w3.org/2001/XMLSchema">
            ……
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="greetingWithParameter">
            <soap:operation soapAction="" style="document" />
            <wsdl:input name="greetingWithParameter">
                <soap:body use="literal" />
            </wsdl:input>
            <wsdl:output name="greetingWithParameterResponse">
                <soap:body use="literal" />
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>

    <wsdl:service name="ProjectManagerService">
        <wsdl:port binding="tns:ProjectManagerImplServiceSoapBinding"
            name="ProjectManagerPort">
            <wsp:PolicyReference
                xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
                URI="#UP_policy" />
            <soap:address
                location="https://localhost:9001/ProjectManager" />
        </wsdl:port>
    </wsdl:service>

    <wsp:Policy wsu:Id="UP_policy"
        xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
```

```
        xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-utility-1.0.xsd">
        <sp:TransportBinding
            xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
            <wsp:Policy>
                <sp:TransportToken>
                    <wsp:Policy>
                        <sp:HttpsToken />
                    </wsp:Policy>
                </sp:TransportToken>
                <sp:Layout>
                    <wsp:Policy>
                        <sp:Strict />
                    </wsp:Policy>
                </sp:Layout>
                <sp:AlgorithmSuite>
                    <wsp:Policy>
                        <sp:Basic128 />
                    </wsp:Policy>
                </sp:AlgorithmSuite>
            </wsp:Policy>
        </sp:TransportBinding>
        <sp:SupportingTokens
            xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
            <wsp:Policy>
                <sp:UsernameToken
                    xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"
                    sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/
securitypolicy/IncludeToken/Always">
                    <wsp:Policy>
                        <sp:WssUsernameToken11 />
                    </wsp:Policy>
                </sp:UsernameToken>
            </wsp:Policy>
        </sp:SupportingTokens>
    </wsp:Policy>

</wsdl:definitions>
```

在本案例中，服务器启动的服务地址是 https://localhost:9001/ProjectManager。服务器引入了 SEI 接口——ProjectManager 接口。这样 ProjectManager 的 Web Services 就暴露到了外部。可以在浏览器上输入 https://localhost:9001/ProjectManager?wsdl 来查看这个服务的内容。

（3）客户端调用程序

客户端应用程序包括 Web Services 的客户端应用程序和配置文件。

Web Services 的客户端应用程序——Client.java 的代码如下：

```
package demo.cxf.ws.securitypolicy.client;

import java.io.File;
import java.net.URL;
import org.apache.cxf.Bus;
import org.apache.cxf.bus.spring.SpringBusFactory;
import demo.cxf.webservices.Project;
import demo.cxf.webservices.ProjectImpl;
import demo.cxf.webservices.ProjectManager;
import demo.cxf.webservices.ProjectManagerService;
```

```
public class Client {

    private Client() {}

    public static void main(String args[]) throws Exception {
        URL wsdlURL;
        File wsdlFile = new File("wsdl/ProjectManager.wsdl");

        if (wsdlFile.exists()) {
            wsdlURL = wsdlFile.toURL();
        } else {
            wsdlURL = new URL(args[0]);
        }

        SpringBusFactory bf = new SpringBusFactory();
        URL busFile = Client.class.getResource("wssec-client.xml");

        Bus bus = bf.createBus(busFile.toString());
        bf.setDefaultBus(bus);

        ProjectManagerService service = new ProjectManagerService();
        ProjectManager projectManager = service.getProjectManagerPort();

        try {
            // 项目的基本情况
            getGreet(projectManager);
            greetingWithParameter(projectManager);
            getProjectName(projectManager);
            getDefaultProject(projectManager);
            getAllProject(projectManager);

            // 修改默认项目，查看项目的基本情况
            setDefaultProject(projectManager);

            // 增加一个项目，查看项目的基本情况
            registerProject(projectManager);

            // 删除一个项目，查看项目的基本情况
            removeProject(projectManager);

        } catch (Exception e) {
            System.out.println("Invocation failed with the following: "    +
e.getCause());
            System.out.println();
        }
        System.exit(0);
    }

    // 调用 SEI 的 getGreet 方法
    private static void getGreet(ProjectManager projectManager)
            throws Exception {
        System.out.println(projectManager.greeting());
    }
    ……
}
```

配置文件 wssec-client.xml 的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:cxf="http://cxf.apache.org/core"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:sec="http://cxf.apache.org/configuration/security"
    xmlns:http="http://cxf.apache.org/transports/http/configuration"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="
         http://cxf.apache.org/core
         http://cxf.apache.org/schemas/core.xsd
         http://cxf.apache.org/configuration/security
         http://cxf.apache.org/schemas/configuration/security.xsd
         http://cxf.apache.org/jaxws
         http://cxf.apache.org/schemas/jaxws.xsd
         http://cxf.apache.org/transports/http/configuration
         http://cxf.apache.org/schemas/configuration/http-conf.xsd
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd">

    <cxf:bus>
        <cxf:features>
            <cxf:logging />
        </cxf:features>
    </cxf:bus>

    <http:conduit
        name="{http://webservices.cxf.demo/}ProjectManagerPort.http-conduit">
        <http:tlsClientParameters disableCNCheck="true">
            <sec:trustManagers>
                <sec:keyStore type="JKS" password="password"
                    file="certs/truststore.jks" />
            </sec:trustManagers>
            <sec:keyManagers keyPassword="password">
                <sec:keyStore type="JKS" password="password"
                    file="certs/wibble.jks" />
            </sec:keyManagers>
            <sec:cipherSuitesFilter>
                <!-- these filters ensure that a ciphersuite with
                    export-suitable or null encryption is used,
                    but exclude anonymous Diffie-Hellman key change as
                    this is vulnerable to man-in-the-middle attacks -->
                <sec:include>.*_EXPORT_.*</sec:include>
                <sec:include>.*_EXPORT1024_.*</sec:include>
                <sec:include>.*_WITH_DES_.*</sec:include>
                <sec:include>.*_WITH_NULL_.*</sec:include>
                <sec:exclude>.*_DH_anon_.*</sec:exclude>
            </sec:cipherSuitesFilter>
        </http:tlsClientParameters>
    </http:conduit>


    <jaxws:client
        name="{http://webservices.cxf.demo/}ProjectManagerPort"
        createdFromAPI="true">
        <jaxws:properties>
            <entry key="ws-security.username" value="Alice" />
            <entry key="ws-security.callback-handler"
                value="demo.cxf.ws.securitypolicy.client.UTPasswordCallback" />
```

```
        </jaxws:properties>
    </jaxws:client>

</beans>
```

还有一个处理口令的类 UTPasswordCallback.java，与服务器的 UTPasswordCallback.java 代码相同，这里省略。

客户端也引入了一个 WSDL 文件——ProjectManager.wsdl，与服务器的 ProjectManager.wsdl 代码相同，这里省略。

本例首先创建一个服务名称为 javax.xml.ws.Service 的实现类，接着通过 addPort 方法来传入服务的端口类、绑定方式和 Web Services 地址。最后通过 getPort 方法来获得服务的 SEI 接口。这就是 Web Services 服务器接口 ProjectManager。后面就基于这个 ProjectManager 的 SEI 来实现各种业务操作。

4. 测试和运行结果

运行客户端程序，其输出结果如下：

```
Starting Client
……
ID: 1
Response-Code: 200
Encoding: UTF-8
Content-Type: text/xml; charset=utf-8
Headers: {Content-Length=[257], Server=[Jetty(6.1.21)], content-type=[text/xml;
charset=utf-8]}
Payload:  <soap:Envelope  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header /><soap:Body><ns2:greetingResponse xmlns:ns2="http://webservices.cxf.demo/">
<return>Hello, This is ProjectManager.</return></ns2:greetingResponse></soap:Body>
</soap:Envelope>
--------------------------------------
Hello, This is ProjectManager.
2011-2-27 14:56:48 org.apache.cxf.interceptor.LoggingOutInterceptor$LoggingCallback
onClose
信息: Outbound Message
---------------------------
ID: 2
Address: https://localhost:9001/ProjectManager
Encoding: UTF-8
Content-Type: text/xml
Headers: {SOAPAction=[""], Accept=[*/*]}
Payload:  <soap:Envelope  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header><wsse:Security
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-se
cext-1.0.xsd" soap:mustUnderstand="1"><wsse:UsernameToken xmlns:wsse="http://docs.
oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd" xmlns:wsu=
"http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.x
sd"   wsu:Id="UsernameToken-2"><wsse:Username>Alice</wsse:Username><wsse:Password
Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-prof
ile-1.0#PasswordText">ecilA</wsse:Password></wsse:UsernameToken></wsse:Security><
/soap:Header><soap:Body><ns2:greetingWithParameter xmlns:ns2="http://webservices.
cxf.demo/"> <greet>greet</greet></ns2:greetingWithParameter></soap:Body></soap:Envelope>
--------------------------------------
2011-2-27 14:56:48 org.apache.cxf.interceptor.LoggingInInterceptor logging
信息: Inbound Message
```

```
---------------------------
ID: 2
Response-Code: 200
Encoding: UTF-8
Content-Type: text/xml; charset=utf-8
Headers: {Content-Length=[289], Server=[Jetty(6.1.21)], content-type=[text/xml;
charset=utf-8]}
Payload: <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header /><soap:Body><ns2:greetingWithParameterResponse xmlns:ns2="http://webservices.
cxf.demo/"><return>Hello, greet This is ProjectManager.</return>
</ns2: greetingWithParameterResponse></soap:Body></soap:Envelope>
--------------------------------------
Hello, greet This is ProjectManager.
2011-2-27 14:56:48 org.apache.cxf.interceptor.LoggingOutInterceptor$LoggingCallback
onClose
信息: Outbound Message
---------------------------
ID: 3
Address: https://localhost:9001/ProjectManager
Encoding: UTF-8
Content-Type: text/xml
Headers: {SOAPAction=[""], Accept=[*/*]}
Payload: <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header><wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-secext-1.0.xsd" soap:mustUnderstand="1"><wsse:UsernameToken
xmlns:wsse="http://docs.oasis- open.org/wss/2004/01/oasis-200401-wss-wssecurity-
secext-1.0.xsd" xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
wss-wssecurity-utility-1.0.xsd" wsu:Id="UsernameToken-3"><wsse:Username>Alice</wsse:
Username><wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
wss-username-token-profile-1.0#PasswordText">ecilA</wsse:Password></wsse:Username
Token></wsse:Security></soap:Header><soap:Body><ns2:getProjectName xmlns:ns2=
"http://webservices.cxf.demo/" /></soap:Body></soap:Envelope>
--------------------------------------
2011-2-27 14:56:48 org.apache.cxf.interceptor.LoggingInInterceptor logging
信息: Inbound Message
---------------------------
ID: 3
Response-Code: 200
Encoding: UTF-8
Content-Type: text/xml; charset=utf-8
Headers: {Content-Length=[258], Server=[Jetty(6.1.21)], content-type=[text/xml;
charset=utf-8]}
Payload: <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header /><soap:Body><ns2:getProjectNameResponse xmlns:ns2="http://webservices.
cxf.demo/"><return>default
projectName</return></ns2:getProjectNameResponse></soap:Body></soap:Envelope>
--------------------------------------
default projectName
……
```

服务器端程序输出结果如下：

```
……
ID: 1
Address: /ProjectManager
Encoding: UTF-8
Content-Type: text/xml; charset=UTF-8
```

```
Headers: {Content-Length=[774], Host=[localhost:9001], User-Agent=[Apache CXF
2.2.10], connection=[keep-alive], SOAPAction=[""], Pragma=[no-cache], Content-Type=
[text/xml; charset=UTF-8], content-type=[text/xml; charset=UTF-8], Cache-Control=
[no-cache], Accept=[*/*]}
Payload: <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header><wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-secext-1.0.xsd" soap:mustUnderstand="1">
<wsse:UsernameToken xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
wss-wssecurity-secext-1.0.xsd" xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-utility-1.0.xsd" wsu:Id="UsernameToken-1"><wsse:Username>
Alice</wsse:Username><wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-username-token-profile-1.0#PasswordText">ecilA</wsse:Password></
wsse:UsernameToken></wsse:Security></soap:Header><soap:Body><ns2:greeting xmlns:ns2=
"http://webservices.cxf.demo/" /></soap:Body></soap:Envelope>
--------------------------------------
----invoking greeting() ----
2011-2-27 14:56:48 org.apache.cxf.interceptor.LoggingOutInterceptor$LoggingCallback
onClose
信息: Outbound Message
---------------------------
ID: 1
Encoding: UTF-8
Content-Type: text/xml
Headers: {}
Payload: <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header /><soap:Body><ns2:greetingResponse xmlns:ns2="http://webservices.cxf.demo/">
<return>Hello, This is ProjectManager.</return></ns2:greetingResponse></soap:Body>
</soap:Envelope>
--------------------------------------
2011-2-27 14:56:48 org.apache.cxf.interceptor.LoggingInInterceptor logging
信息: Inbound Message
---------------------------
ID: 2
Address: /ProjectManager
Encoding: UTF-8
Content-Type: text/xml; charset=UTF-8
Headers: {Content-Length=[833], Host=[localhost:9001], User-Agent=[Apache CXF
2.2.10], connection=[keep-alive], SOAPAction=[""], Pragma=[no-cache], Content-Type=
[text/xml; charset=UTF-8], content-type=[text/xml; charset=UTF-8], Cache-Control=
[no-cache], Accept=[*/*]}
Payload: <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header><wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-secext-1.0.xsd" soap:mustUnderstand="1"><wsse:UsernameToken
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-se
cext-1.0.xsd" xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-utility-1.0.xsd" wsu:Id="UsernameToken-2"><wsse:Username>Alice</wsse:Username>
<wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-
token-profile-1.0#PasswordText">ecilA</wsse:Password></wsse:UsernameToken></wsse:
Security></soap:Header><soap:Body><ns2:greetingWithParameter xmlns:ns2="http://
webservices.cxf.demo/"><greet>greet</greet></ns2:greetingWithParameter></soap:Bod
y></soap:Envelope>
--------------------------------------
----invoking greetingWithParameter be called
2011-2-27 14:56:48 org.apache.cxf.interceptor.LoggingOutInterceptor$LoggingCallback
onClose
```

```
信息: Outbound Message
---------------------------
ID: 2
Encoding: UTF-8
Content-Type: text/xml
Headers: {}
Payload: <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header />><soap:Body><ns2:greetingWithParameterResponse xmlns:ns2="http://
webservices.cxf.demo/"><return>Hello, greet This is ProjectManager.</return>
</ns2:greetingWithParameterResponse></soap:Body></soap:Envelope>
--------------------------------------
2011-2-27 14:56:48 org.apache.cxf.interceptor.LoggingInInterceptor logging
信息: Inbound Message
----------------------------
ID: 3
Address: /ProjectManager
Encoding: UTF-8
Content-Type: text/xml; charset=UTF-8
Headers: {Content-Length=[780], Host=[localhost:9001], User-Agent=[Apache CXF
2.2.10], connection=[keep-alive], SOAPAction=[""], Pragma=[no-cache], Content-Type=
[text/xml; charset=UTF-8], content-type=[text/xml; charset=UTF-8], Cache-Control=
[no-cache], Accept=[*/*]}
Payload: <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header><wsse:Security
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-se
cext-1.0.xsd" soap:mustUnderstand="1"><wsse:UsernameToken xmlns:wsse="http://
docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-
1.0.xsd" wsu:Id="UsernameToken-3"><wsse:Username>Alice</wsse:Username><wsse:Password
Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-prof
ile-1.0#PasswordText">ecilA</wsse:Password></wsse:UsernameToken></wsse:Security><
/soap:Header><soap:Body><ns2:getProjectName xmlns:ns2="http://webservices.cxf.demo/" />
</soap:Body></soap:Envelope>
--------------------------------------
----invoking getProjectName() ----
……
```

5. 其他内容

上述的所有调用都基于 JAX-WS 规范。

本例代码参见配套资源代码中“第 13 章\ws_security_ut_policy”文件夹。

13.6 Apache CXF 支持 WS-Trust

WS-Trust 规范是 WS-*规范族中的一员，也是 OASIS（Organization for the Advancement of Structured Information Standards）其中的一项标准。WS-Trust 规范对 WS-Security 规范提供了一些扩展，专门处理有关安全 tokens 的发布、整新和验证，确保各方参与者的互操作处在一个可信任的安全数据交换环境中。通过使用 WS-Trust 规范中定义的这些扩展规范，可以确保工作在 Web 服务框架中的 Web 应用之间的通信是安全的。

WS-Trust 规范发起于一些有代表性的公司，最后于 2007 年 3 月被 OASIS 核准通过并作为一

项标准发布。

本节简单介绍一下 WS-Trust 规范的概念，以及 Apache CXF 框架使用 WS-Trust 规范的方法。

13.6.1 WS-Trust 简介

WS-Trust 规范定义了对安全令牌服务进行请求的形式，以及如何返回响应。

安全令牌是 WS-Security 规范的基础。例如，要获得 Kerberos 票据，可以与 Kerberos 密钥发行中心（KDC）联系；要获得 X.509 证书，则可以与证书颁发机构（CA）联系。尽管 WS-Security 规范定义了如何在标准 XML 内包装 Kerberos 票据和 X.509 证书以便在 SOAP 消息中进行传输，但 WS-Security 规范没有定义使用 SOAP 与 KDC、CA 或者能够分发安全令牌的任何其他服务进行交谈的方法。WS-Trust 规范修正了这个问题。这个规范定义的主要内容是从一些安全令牌服务（如 KDC 或 CA）请求安全令牌的机制。假设客户端希望调用 Web Services A 和 Web Services B，并进一步假设 Web Services A 需要 Kerberos 票据进行身份验证，而 Web Services B 需要数字签名以及相匹配的 X.509 证书。如图 13-19 所示，客户端可能会首先向安全令牌服务 X（实际上是 KDC）发出 SOAP 请求，以请求包装在<BinarySecurityToken>内部的 Kerberos 票据。然后调用 Web Services A，将最新获得的令牌传递给 SOAP 消息的<Security>元素。接下来，客户端可以向安全令牌服务 Y（即 CA）发出 SOAP 请求，以请求 X.509 证书。在证书返回后，客户端就可以调用 Web Services B，以同时传递这个证书和匹配的数字签名。

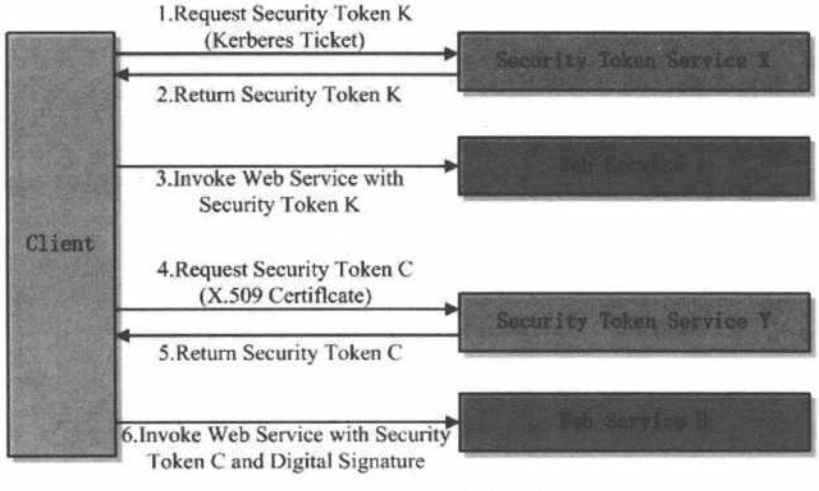


图 13-19　WS-Trust 的交互过程

WS-Trust 规范定义了向安全令牌服务进行请求的形式，以及如何返回响应。这种交互依赖于两个元素：<RequestSecurityToken>和<RequestSecurityTokenResponse>。其基本思想是发送包含<RequestSecurityToken>作为其正文的 SOAP 消息，然后获得包含 <RequestSecurityTokenResponse>的 SOAP 消息以及新的安全令牌。

WS-Trust 规范综合了两个相关函数。第一个函数是为了支持对安全令牌的处理，具体而言就是发出、更换以及取消安全令牌。第二个函数是为了支持中介信任关系。这两个函数看起来虽然不同，但它们之间是相互关联的，都要求安全令牌必须是可信的，并且这种信任也必须要以某

种形式的令牌表示。WS-Trust 规范的核心是一组用来发出、更换、取消以及验证安全令牌的消息。这些消息可以由客户机通过调用一种称为 Security Token Service（STS）的特定类型的 SOAP Web Services 进行交换。它们还可以采用其他的方式传递（比如以对另外一个服务的请求的安全头部的形式）。

下面简单介绍一下 STS 的基础知识。STS 是一种 Web Services，可实现一个由 WS-Trust 规范定义的简单接口。这个接口允许客户机使用安全令牌提交对几种类型操作的请求。由于一个 STS 就是一种 Web Services，因此 WS-Security 规范可被直接用在请求和响应消息内，而 WS-SecurityPolicy 规范则可被用来指定由客户机提供的凭证类型或需要在消息上进行其他的安全性处理。

13.6.2　Apache CXF 使用 WS-Trust 的配置

在有了 Apache CXF 的支持后，WS-Trust 建立了 WS-SecurityPolicy 去实现处理 IssuedToken 策略断言，该断言可以在 WS-SecurityPolicy 信息块中找到。

注意：因为 WS-IssuedToken 支持是建立在 WS-Security Policy 的基础上，目前仅适用于“wsdl 优先”的项目。

WS-Trust 规范扩展了 WS-Security 规范，允许发行、更新和安全令牌验证。WS-Trust 规范所做的大部分工作都是围绕着“安全令牌服务”或 STS 的。STS 用于创建消息处理的服务以获得安全令牌。执行 STS 是用于交流服务的 SAML 令牌。当使用“直线”的 WS-Security 规范，客户端和服务器必须事先交换密钥。如果客户端和服务器在同一个安全域，这通常不是一个问题，但对于较大的、复杂的应用跨越多个域，它可能是一个负担。此外，如果多种服务需要相同的安全凭据，更新所有的服务时，可以通过一个重大行动来改变这些凭据。

WS-Trust 规范通过使用安全令牌来解决问题，该安全令牌来源于一个值得信赖的安全令牌服务。客户端可以通过以策略为基础的 STS 和由 STS 定义的需求来进行验证本身，执行 STS 然后提供一个安全令牌（例如：一个 SAML 令牌），客户端可用于与目标服务进行交流，这项服务可以验证令牌，以确保它真的来自受信任的 STS。

当 Apache CXF 框架的 WS-SecurityPolicy 在运行时遇到 IssuedToken 断言，运行时要求 org.apache.cxf.ws.security.trust.STSClient 的实例化对象来与 STS 进行交流，以获得所需的令牌。由于 STSClient 是一个 WS-SecurityPolicy 客户端，它需要配置项来创建安全 SOAP 消息以便于与 STS 进行交流。

以下是几种配置 STSClient 的方法。

1. 直接配一个 STS 的 bean 属性：

在这种情况下，直接创建一个 STSClient 对象作为客户端对象的属性。该 wsdlLocation、服务/端点名称等都在该客户端上进行配置。配置信息如下：

```
<jaxws:client name="{http://cxf.apache.org/}MyService" createdFromAPI="true">
   <jaxws:properties>
      <entry key="ws-security.sts.client">
         <!-- direct STSClient config and creation -->
         <bean class="org.apache.cxf.ws.security.trust.STSClient">
```

```
            <constructor-arg ref="cxf"/>
            <property name="wsdlLocation" value="target/wsdl/trust.wsdl"/>
            <property name="serviceName" value="{http://cxf.apache.org/
securitytokenservice}SecurityTokenService"/>
            <property name="endpointName" value="{http://cxf.apache.org/
securitytokenservice}SecurityTokenEndpoint"/>
            <property name="properties">
                <map>
                    <entry key="ws-security.username" value="alice"/>
                    <entry key="ws-security.callback-handler" value="client.
MyCallbackHandler"/>
                    <entry key="ws-security.signature.properties" value="clientKeystore.
properties"/>
                    <entry key="ws-security.encryption.properties" value="clientKeystore.
properties"/>
                    <entry key="ws-security.encryption.username" value="mystskey"/>
                </map>
            </property>
        </bean>
    </entry>
  </jaxws:properties>
</jaxws:client>
```

上面例子显示了如何配置 STS，通过使用 UserNameToken 配置文件来验证客户端。这是假设在 clientKeystore.properties 内的密钥库包含了 STS 的客户端私钥和公钥（以上为 mystskey）。如果没有，针对签名和加密性能属性单独创建一个属性文件，并分别指向密钥库和信任库。

记住 jaxws:client 的 createdFromAPI 属性需要设置为 true（如上所示）。如果开发者通过编写程序代码来创建 CXF 的客户端——需要调用 Endpoint.publish 或 Service.getPort。

这也适用于“代码优先”的情况：

```
STSClient sts = new STSClient(...);
sts.setXXXX(....)
.....
((BindingProvider)port).getRequestContext().put("ws-security.sts.client", sts);
```

clientKeystore.properties 格式的例子：

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=KeystorePasswordHere
org.apache.ws.security.crypto.merlin.keystore.alias=ClientKeyAlias
org.apache.ws.security.crypto.merlin.file=NameOfKeystore.jks
```

2. 基于端点名称的间接配置

如果在运行时没有找到直接在客户端上配置的 STSClient，这会用附加了.sts-client 的 STSClient 对象来检查配置文件。例如，在客户端的端点名称为“{http://cxf.apache.org/}TestEndpoint”，那么可以被配置为：

```
<bean name="{http://cxf.apache.org/}TestEndpoint.sts-client"
   class="org.apache.cxf.ws.security.trust.STSClient" abstract="true">
   <property name="wsdlLocation" value="WSDL/wsdl/trust.wsdl"/>
   <property name="serviceName" value="{http://cxf.apache.org/securitytokenservice}
SecurityTokenService"/>
```

```
    <property name="endpointName" value="{http://cxf.apache.org/securitytokenservice}
SecurityTokenEndpoint"/>
    <property name="properties">
        <map>
            <entry key="ws-security.sts.token.properties" value="etc/bob.properties"/>
            <entry key="ws-security.callback-handler" value="interop.client.
KeystorePasswordCallback"/>
            <entry key="ws-security.signature.properties" value="etc/alice.properties"/>
            <entry key="ws-security.encryption.properties" value="etc/bob.properties"/>
        </map>
    </property>
</bean>
```

这个例子的配置属性演示了使用 X.509 令牌来进行 STS 验证的过程。abstract="true"设置表示 bean 在实际需要时缓建 STSClient 对象。当这种情况发生后，在 CXF 运行过程中将用这个 bean 配置值实例化一个新 STSClient。

3. 默认配置

如果 STSClient 不能从上面的方法发现，它试图找到另外的间接配置之一，但具有名称“default.sts-client”。这可以用来配置多个服务的 sts-clients。

Apache CXF 框架支持 WS-Trust 1.4t。Apache CXF 提供有限的 WS-Trust 1.4 支持。目前支持的功能如下。

ActAs (2.2.10)能力允许发起人请求一个安全令牌。这种能力成为重要的复合服务，该复合服务遵循 JAX-WS 规范并采用自定义的 Interceptor 的应用模式。在这种情况下，依赖方（即间接服务请求的最终目的地）可能需要请求真正来源的信息。ActAs 能力允许一个中介去请求可以传达这一消息的令牌。

下面的示例代码演示如何使用一个拦截器来动态设置在 STS RST 的 ActAs 内容。该值可能是一个包含格式良好的 XML 或 DOM 元素的字符串变量。内容将逐字地被添加到 RST。请注意，这个拦截器是适用于有担保的客户或发起人，而不是对应 STSClient 的拦截器链。

```
public class ActAsOutInterceptor extends AbstractPhaseInterceptor<Message> {
    ActAsOutInterceptor () {
        // This can be in any stage before the WS-SP interceptors
        // setup the STS client and issued token interceptor.
        super(Phase.SETUP);
    }

    @Override
    public void handleMessage(Message message) throws Fault {
        message.put(SecurityConstants.STS_TOKEN_ACT_AS, ...);

    }
}
```

另外，ActAs 内容可能会被直接在 STS 中进行设置，如下所示：

```
<bean name="{http://cxf.apache.org/}TestEndpoint.sts-client"
    class="org.apache.cxf.ws.security.trust.STSClient" abstract="true">
    <property name="wsdlLocation" value="WSDL/wsdl/trust.wsdl"/>
    <property name="serviceName" value="{http://cxf.apache.org/securitytokenservice}
SecurityTokenService"/>
    <property name="endpointName" value="{http://cxf.apache.org/securitytokenservice}
```

```
SecurityTokenEndpoint"/>
    <property name="actAs" value="..."/>
    <property name="properties">
        <map>
            <entry key="ws-security.sts.token.properties" value="etc/bob.properties"/>
            <entry key="ws-security.callback-handler" value="interop.client.
KeystorePasswordCallback"/>
            <entry key="ws-security.signature.properties" value="etc/alice.properties"/>
            <entry key="ws-security.encryption.properties" value="etc/bob.properties"/>
        </map>
    </property>
</bean>
```

13.7 Apache CXF 支持 WS-SecureConversation

WS-SecureConversation 规范是在 WS-Security 规范和 WS-Trust 规范之上构建的一种规范，用于支持涉及多个消息的安全交换。由于这所使用的上下文内容都是针对运行中的消息交换，因此 WS-SecureConversation 规范有可能比 WS-Security 规范实现更高的效率。WS-SecureConversation 策略包括一个<sp:SecureConversationToken>元素，用于描述消息交换将使用的安全令牌，以及提供应用于 token-exchange 消息的安全性选项。

本节简单介绍一下 WS-SecureConversation 规范的概念，以及 Apache CXF 框架使用 WS-SecureConversation 规范的方法。

13.7.1 WS-SecureConversation 介绍

WS-SecureConversation 规范构建于 WS-Trust 规范基础上，使用一个 STS 来管理 SCT。一个 SCT 代表在消息交换所涉各方之间共享的上下文，此共享上下文内的信息允许各方使用非对称加密来确保信息的安全。特别的，此上下文向消息交换中所涉的各方提供了一个共享的秘密值。这个共享的秘密值本身可以是在交换过程中用来对消息进行非对称加密的密钥。

WS-SecureConversation 规范在理论上可以用于多方的消息交换，但它最为常见的用法是在一个客户机与一个服务器之间的通信上。当用在这种配置中时，向客户机提供 SCT 的这个 STS 与此服务器位于一处，可在相同的端点地址访问到。这意味着服务器上的 Web Services 代码需要一种方式来辨别哪些消息针对 STS、哪些消息针对服务本身，用此请求上的这个动作可服务此目的。图 13-20 展示了这种位于一处的 STS 配置以及消息交换。

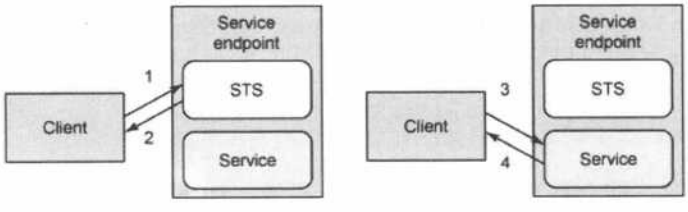


图 13-20 与服务器位于一处的 WS-SecureConversation STS

当客户机想要开始与服务器交换消息时，它首先会联系此 STS 并建立上下文。如图 13-18 所示，消息 1 指定了动作 http://schemas.xmlsoap.org/ws/2005/02/trust/RST/SCT。响应消息 2 则向客户机提供了此 SCT。上下文在消息 3 中被引用，由它指定与实际的服务应用程序相关的任何动作。在此时间区间内，这个 SCT 在自此客户机至此服务器的任何连续消息中均有效。消息 3 和消息 4 使

用了基于共享秘密的对称加密，客户机和服务器之间的所有后续消息也是如此。服务器应用程序使用由此客户机提供的上下文引用来直接访问来自这个上下文（由 STS 保存的）的共享秘密。

13.7.2　Apache CXF 使用 WS-SecureConversation 的配置

在 Apache CXF 框架的支持下，WS-SecureConversation 规范在 WS-SecurityPolicy 规范实现下建立了处理 SecureConverstationToken 策略，可以在 WS-SecurityPolicy 片段发现断言。

注意： 由于 WS-SecureConversation 规范建立在 WS-SecurityPolicy 的基础上，目前仅适用于“wsdl 优先”的项目。

WS-Security 规范的问题之一，是为所有通信使用强大的加密密钥，这样会造成通信性能大幅度下降。WS-SecureConversation 规范在一开始帮助去重新激活客户端和服务器端使用的加密在将来还可以重新使用，这有一个问题，如果客户端需要发送多个请求到服务器，那只需要客户端发送一个请求，然后其他的请求都会被丢弃。WS-SecureConversation 规范实际上减慢了谈判的要求和延长了请求/响应到服务器。

在 WS-SecureConversation 规范中有两个安全政策将是有效的：

“outer”策略，介绍与实际端点交互的安全要求。这将在某处包含 SecureConversationToken。“bootstrap”策略保持在 SecureConverstationToken 内。这项策略在当客户端谈判 SecureConversation 的关键值时才能启动作用。

像直接配置 WS-SecurityPolicy 规范那样为 WS-SecureConversation 规范的有效工作配置 WS-SecurityPolicy 属性。唯一的区别是，需要有一种方式来指定哪些属性是用于 SecureConversationToken 的“bootstrap”策略，哪些是用于实际服务策略。要做到这一点，属性为 SecureConversationToken 的“bootstrap”策略被附加了“.sct”后缀，例如：

```
<jaxws:client name="{http://InteropBaseAddress/interop}XDC-SEES_IPingService"
createdFromAPI= "true">
   <jaxws:properties>
      <!-- properties for the external policy -->
      <entry key="ws-security.username" value="abcd"/>

      <!-- properties for the SecureConversationToken bootstrap policy -->
      <entry key="ws-security.username.sct" value="efgh"/>
      <entry key="ws-security.callback-handler.sct" value="interop.client.
KeystorePasswordCallback"/>
      <entry key="ws-security.encryption.properties.sct" value="etc/bob.properties"/>
   </jaxws:properties>
</jaxws:client>
```

通过 Java API 时，使用代码类似于以下内容：

```
org.apache.cxf.endpoint.Client client;
client.getRequestContext().put("ws-security.username.sct", username);
client.getRequestContext().put("ws-security.password.sct", password);
```

注意： 在 WS-SecureConversation 规范最常见的使用下，开发者不需要配置任何服务策略。所有“硬”的东西都用于“bootstrap”策略并通过服务策略来提供新密钥。这样可以使自己的服务器沟通尽可能简单有效。

第 14 章 CXF 的高级功能

本章主要说明 Apache CXF 框架的高级功能。首先介绍 Apache CXF 框架的 Feature 功能，然后分别说明 Apache CXF 框架的拦截器（Interceptors）、相位器（Phases）、代理（invoker）等。最后说明 Apache CXF 框架的 MER（Multiplexed EndpointReferences）、基础服务和服务路由（Service Routing）。

本章内容：

1. CXF 的 Feature 功能介绍。
2. CXF 的拦截器（Interceptors）和相位器（Phases）。
3. CXF 的代理（invoker）。
4. CXF 的 MER（Multiplexed EndpointReferences）。
5. CXF 的基础服务。
6. CXF 的服务路由（Service Routing）。

14.1 CXF 的 Feature 功能

本节主要说明 Apache CXF 框架的 Feature（功能）内容，编写和配置 Apache CXF 框架的 Feature 功能，以及 Apache CXF 框架的 Feature 列表和实现例子。

14.1.1 CXF 的 Feature 功能说明

实现 Apache CXF 框架的 Feature（功能）通常通过添加功能来定制服务器端、客户端和总线（Bus）。例如，增加一个 LoggingFeature 功能来记录服务器端和客户端的每一个消息。

默认情况下，Apache CXF 框架所有的初始化方法都委托给 Feature 对象的 initializeProvider（InterceptorProvider）方法，如果开发人员希望增加一些新的功能，只需简单地增加这些功能拦截器到服务器端、客户端和总线（Bus）。下面是 AbstractFeature 的代码。

```
public abstract class AbstractFeature {
   public void initialize(Server server, Bus bus) {
      initializeProvider(server.getEndpoint(), bus);
   }

   public void initialize(Client client, Bus bus) {
```

```
        initializeProvider(client, bus);
    }

    public void initialize(Bus bus) {
        initializeProvider(bus, bus);
    }

    protected void initializeProvider(InterceptorProvider provider, Bus bus) {
        // you could customized the interceptors in the provider here
    }

    /**
     * Convenience method to extract a feature by type from an active list.
     *
     * @param features the given feature list
     * @param type the feature type required
     * @return the feature of the specified type if active
     */
    public static <T> T getActive(List<AbstractFeature> features,
                                  Class<T> type) {
        T active = null;
        if (features != null) {
            for (AbstractFeature feature : features) {
                if (type.isInstance(feature)) {
                    active = type.cast(feature);
                    break;
                }
            }
        }
        return active;
    }
}
```

从上面的代码可以看到抽象类 AbstractFeature 构造的时候要执行 initializeProvider 方法。所以，当对 Feature 有特殊处理的时候，把要操作的内容添加到 initializeProvider 方法即可。

14.1.2　编写和配置 CXF 的 Feature

Apache CXF 框架提供多种 Feature 配置以实现不同的功能，如日志记录、故障转移、策略、寻址、信息可靠性等。从表 14-1 和表 14-2 中可以看到 Feature 列表显示的功能。

编写 Feature 功能是一件既简单又容易的事情。如果开发人员要增加一个新的 Feature 功能。那么首先创建一个 Feature 类，该类要继承 AbstractFeature 抽象类并实现 initializeProvider 接口。同时在新建 Feature 类中写功能扩展代码来定制客户端或服务器配置拦截器。下面就是一个增加日志 Feature 功能的例子。

```
public class LoggingFeature extends AbstractFeature {
    private static final int DEFAULT_LIMIT = 100 * 1024;
    private static final LoggingInInterceptor IN = new LoggingInInterceptor
(DEFAULT_LIMIT);
    private static final LoggingOutInterceptor OUT = new LoggingOutInterceptor
(DEFAULT_LIMIT);

    int limit = DEFAULT_LIMIT;
```

```
    @Override
    protected void initializeProvider(InterceptorProvider provider, Bus bus) {
        if (limit == DEFAULT_LIMIT) {
            provider.getInInterceptors().add(IN);
            provider.getInFaultInterceptors().add(IN);
            provider.getOutInterceptors().add(OUT);
            provider.getOutFaultInterceptors().add(OUT);
        } else {
            LoggingInInterceptor in = new LoggingInInterceptor(limit);
            LoggingOutInterceptor out = new LoggingOutInterceptor(limit);
            provider.getInInterceptors().add(in);
            provider.getInFaultInterceptors().add(in);
            provider.getOutInterceptors().add(out);
            provider.getOutFaultInterceptors().add(out);
        }
    }

    public void setLimit(int lim) { limit = lim; }

    public int getLimit() {
        return limit;
    }
}
```

也可以在编写代码过程中动态添加 Feature 功能。主要是在 initializeProvider 方法上编写添加 Feature 功能的代码，上例主要是加入日志功能。

要在服务器和客户端添加 Feature 功能，需要使用服务类的 Feature 注释，如下代码所示：

```
@org.apache.cxf.feature.Features (features =
"org.apache.cxf.jaxws.service.AnnotationFeature")
    public class HelloServiceImpl implements HelloService {
        public String sayHi() {
            return "HI";
        }
    }
```

开发人员通过 ServerFactoryBean 实例化对象在服务器端添加 Feature 功能，同时通过使用 ClientFactoryBean 实例化对象在客户端添加 Feature 功能，如下代码所示：

```
import org.apache.cxf.frontend.ServerFactoryBean;
import org.apache.cxf.frontend.ClientFactoryBean;
...
ServerFactoryBean serverFactoryBean = new ServerFactoryBean();
MyFeature myFeature = new MyFeature();
// added my feature to the serverFactoryBean
serverFactoryBean.setFeatures(new ArrayList().add(myFeature));
...

ClientFactoryBean clientFactoryBean = new ClientFactoryBean();
clientFactoryBean.setFeatures(new ArrayList().add(myFeature));
...
```

上面的代码是直接在程序代码中添加 Feature 功能，另外，也可以通过配置文件来添加 Feature 功能。

下面用一些例子来说明如何使用配置文件增加 Feature 功能。配置文件如下：

```
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:cxf="http://cxf.apache.org/core"
     xmlns:jaxws="http://cxf.apache.org/jaxws"
     xsi:schemaLocation="
http://cxf.apache.org/core
http://cxf.apache.org/schemas/core.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://cxf.apache.org/jaxws
http://cxf.apache.org/schemas/jaxws.xsd">
   <!-- adding the feature to the bus-->
   <cxf:bus>
      <cxf:features>
         <cxf:logging/>
      </cxf:features>
   </cxf:bus>

   <bean id="myfeature" class="com.example.Myfeature"/>

   <!-- adding the feature to the client -->
   <jaxws:client id="client"
   serviceClass="org.apache.hello_world_soap_http.Greeter"
   wsdlLocation="wsdl/hello_world.wsdl">
   <jaxws:features>
       <bean ref="myfeature" />
   <jaxws:features>
   </jaxws:client>

   <!-- adding the feature to the server -->
<jaxws:server id="server"
serviceBean="org.apache.hello_world_soap_http.GreetImpl"
address="http://localhost:8080/simpleWithAddress">
     <jaxws:features>
       <bean class="com.example.Myfeature"/>
     </jaxws:features>
   </jaxws:server>

</beans>
```

可以在 FeaturesList 功能列表中找到更多的关于 Apache CXF 提供 Feature 功能的信息。

14.1.3　CXF 的 Feature 列表

在配置文件中，http://org.apache.cxf/core 和 http://org.apache.cxf/jaxws 命名空间下的 Feature 元素还包括一些子元素。不同的配置元素能实现不同的功能。这些 Feature 功能名称、实现的类以及这些类所在的包如表 14-1 所示。

表 14-1　Feature功能名称、实现类和所存在的包

Feature功能名称	实现类	所存在的包
logging	org.apache.cxf.feature.LoggingFeature	cxf-rt-core
failover	org.apache.cxf.clustering.FailoverFeature	cxf-rt-core
bean	org.apache.cxf.feature.FastInfosetFeature	cxf-rt-core

（续表）

Feature功能名称	实现类	所存在的包
bean	org.apache.cxf.transport.http.gzip.GZIPFeature	cxf-rt-transport-http
enableColoc	org.apache.cxf.binding.coloc.feature.ColocFeature	cxf-rt-bindings-coloc
bean	org.apache.cxf.databinding.stax.StaxDataBindingFeature	cxf-rt-core
bean	org.apache.cxf.management.interceptor.ResponseTimeFeature	cxt-rt-management
addressing	org.apache.cxf.ws.addressing.WSAddressingFeature	cxf-rt-ws-addr
policies	org.apache.cxf.ws.policy.WSPolicyFeature	cxf-rt-ws-policy
reliableMessaging	org.apache.cxf.ws.rm.feature.RMFeature	cxf-rt-ws-rm

这些 Feature 功能名称、实现功能描述以及命名空间如表 14-2 所示。

表 14-2　Feature功能名称、实现功能描述以及命名空间

Feature功能名称	实现功能描述	命名空间
logging	出站、入站和故障消息内容的日志	http://cxf.apache.org/core
failover	该Feature功能允许客户端从最初的目标端点故障转移到另一个目标，兼容端点的目标服务	http://cxf.apache.org/clustering
bean	启用FastInfoset来编码消息。默认FastInfoset作为服务器使用第一条消息的一部分，但是“force=true”属性可以强制执行，就算是第一条消息也不可避免	
bean	允许使用gzip来压缩传输过来的消息	
enableColoc		http://cxf.apache.org/binding/coloc
bean		http://springframework.org
bean		http://springframework.org
addressing	启用和控制WS-Addressing的使用	http://cxf.apache.org/ws/addressing
policies	启用和控制的WS-Policy框架的使用。也可作为一个策略（Policy）和策略引用（PolicyReference）元素的附件点	http://cxf.apache.org/policy-config
reliableMessaging	启用和控制WS-RM的使用	http://cxf.apache.org/ws/rm/manager

14.1.4　CXF 实现 Feature 的例子

本程序演示遵循 Apache CXF 框架 API 规范并采用具有 GZIPFeature 功能的客户端和服务器端的调用，客户端采用 Apache CXF 框架的代码优先（Java First）前端应用模式，通过 Spring IoC 注入来实现。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，后台服务端由遵循 Apache CXF 框架 API 规范的 ServerFactoryBean 实现。使用 GZIPFeature 功能。前端访问直接采用 Spring IoC 注入方式，实现的结构如图 14-1 所示。

其中：① 表示客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项

目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。在客户端和服务器端的 SEI 都是 ProjectManager 接口。

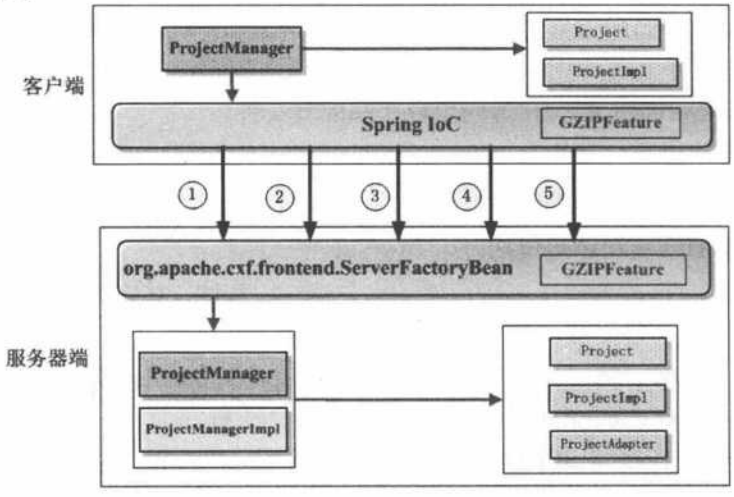


图 14-1　遵循 Apache CXF 框架并采用具有 GZIPFeature 功能的应用模式

2. 应用程序案例实现步骤

创建遵循 Apache CXF API 规范并实现 GZIPFeature 功能程序的流程如图 14-2 所示。

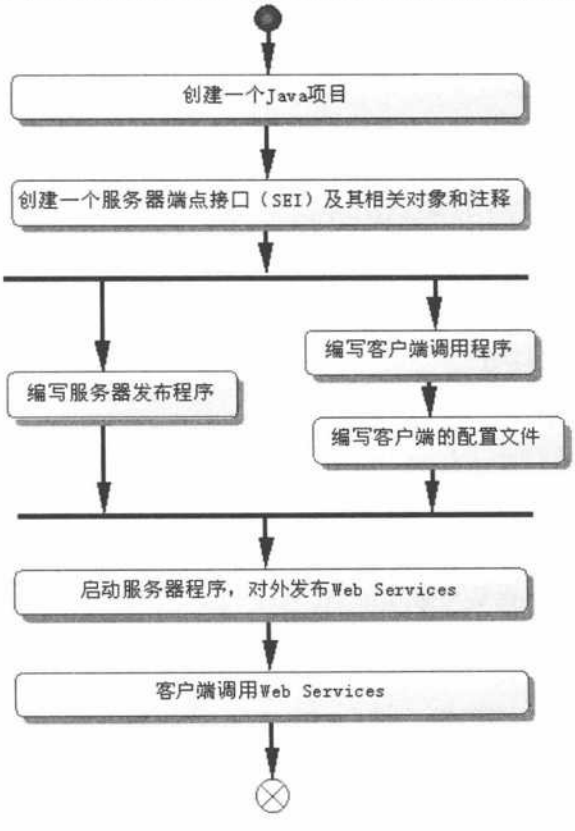


图 14-2　应用程序案例实现步骤流程图

流程图说明，要实现以下步骤：

（1）创建一个 Java 项目。

（2）创建一个服务器端点接口（SEI）及其相关对象和注释。

（3）编写发布这个 Java 程序为 Web Services 的服务端程序。

（4）编写 Web Services 的客户端应用程序和配置文件。

（5）启动服务器程序，对外发布 Web Services。

（6）客户端调用 Web Services。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 14-3 所示。

包目录包括三个部分，第一个包目录 demo.cxf.integration.feature.server 是服务器程序目录，第二个包目录 demo.cxf.integration.feature.client 是客户端程序目录，第三个包目录 demo.cxf.webservices 是服务器端点接口 SEI 和相关对象等公用部分。服务器端和客户端都有可能引用第三个包目录内容。

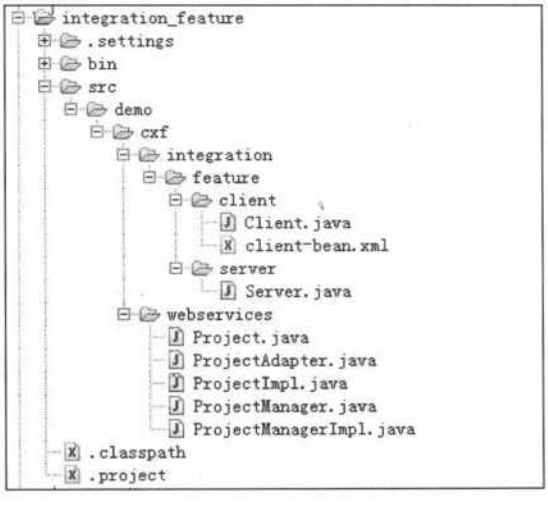


图 14-3　integration_feature 程序结构

服务器端点接口（SEI）是公用部分的接口，服务器端和客户端都必须引用。但服务器端和客户端对公用部分的引用是不同的。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口，而服务器端程序是接口和实现类都要引用。

在编写本程序例子时，主要是与第 6 章中“采用 POJO 并基于 JAX-WS 服务的前端实现例子”进行比较，本案例增加 GZIPFeature 功能实现，下面只列出有区别的两个文件的代码。

（1）服务器端发布程序

第一个文件是服务器端发布程序，即 Server.java，代码如下：

```
package demo.cxf.integration.feature.server;

import javax.xml.ws.Endpoint;
import org.apache.cxf.jaxws.JaxWsServerFactoryBean;
```

```
import org.apache.cxf.transport.http.gzip.GZIPFeature;

import demo.cxf.webservices.*;

public class Server {

    public Server() {
        System.out.println("Starting Server");
        ProjectManager projectManager = new ProjectManagerImpl();
        GZIPFeature gzip = new GZIPFeature();
        gzip.setThreshold(1);
        JaxWsServerFactoryBean server = new JaxWsServerFactoryBean();
        server.setServiceBean(projectManager);
        String address = "http://localhost:9000/ProjectManager";
        server.setAddress(address);
        server.getFeatures().add(gzip);
        server.create();

    }

    public static void main(String[] args) throws Exception {
        new Server();
        System.out.println("Server ready...");

        Thread.sleep(5 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);

    }
}
```

上面代码中加粗部分是实现 GZIPFeature 功能的程序代码。

在本案例中，服务器启动的服务地址是 http://localhost:9000/ProjectManager。服务器引入了 SEI 接口——ProjectManager 接口。这样 ProjectManager 的 Web Services 就暴露给了外部，可以在浏览器上输入 http://localhost:9000/ProjectManager?wsdl 来查看这个服务的内容。

（2）客户端调用程序

客户端程序代码是基本相同的。只有客户端的 Spring 配置文件不同，即 client-bean.xml，代码如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                    http://www.springframework.org/schema/beans/spring-beans.xsd
                    http://cxf.apache.org/transports/http/configuration
                    http://cxf.apache.org/schemas/configuration/http-conf.xsd
                    http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

    <http-conf:conduit name="*.http-conduit">
        <http-conf:client AcceptEncoding="gzip" />
    </http-conf:conduit>

    <jaxws:client id="ProjectManagerClient"
```

```
        serviceClass="demo.cxf.webservices.ProjectManager"
        address="http://localhost:9000/ProjectManager">
        <jaxws:features>
            <bean
                class="org.apache.cxf.transport.http.gzip.GZIPFeature">
                <property name="threshold" value="1" />
            </bean>
        </jaxws:features>
    </jaxws:client>
</beans>
```

加粗部分是配置信息，主要是配置 GZIPFeature 类进入系统中。

4. 测试和运行结果

测试编写好的程序代码，首先要启动服务器程序，即在 Eclipse 上运行 Server.java，然后运行客户端程序，测试结果与第 6 章中“采用 POJO 并基于 JAX-WS 服务的前端实现例子”的测试结果类似。

5. 其他内容

上述的所有调用都基于 JAX-WS 规范。

本例代码参见配套资源代码中“第 14 章\integration_feature”文件夹。

14.2 CXF 的拦截器（Interceptors）和相位器（Phases）

本节主要介绍 Apache CXF 框架的拦截器（Interceptors）和相位器（Phases）的基本概念和使用，并给出一个例子进行说明。

14.2.1 CXF 的拦截器（Interceptors）和相位器（Phases）介绍和使用

拦截器（Interceptors）是 Apache CXF 内部的基本处理单元。当服务被调用时，首先创建一个拦截器链（InterceptorChain）。在拦截器链（InterceptorChain）上，每个拦截器都有机会处理拦截的消息，包括读消息、修改消息、增加消息头、验证消息，等等。

拦截器可以应用在客户端和服务器端。当 Apache CXF 客户端调用 Apache CXF 服务器端时，对于 Apache CXF 客户端拦截器而言，这是一个传出拦截器链，但对于 Apache CXF 服务器端而言，这是一个传入拦截器链。当服务器发送响应返回给客户端，对于 Apache CXF 服务器端的拦截器，这是一个传出拦截器链，而对于 Apache CXF 客户端而言，这是一个传入拦截器链。在 SOAPFaults 情况下，Apache CXF 框架中的 Web Services 将创建一个独立的传出错误处理拦截器链，而客户端将创建一个独立的传入错误处理拦截器链。

Apache CXF 内置的拦截器包括：

- SoapActionInterceptor ——处理 SOAPAction 消息头。如果有操作被设置，就将被执行。
- StaxInInterceptor ——从传输的输入流中创建一个 Stax XMLStreamReader。
- Attachment(In/Out)Interceptor ——转化多部分/相关消息到一系列的附件。

拦截器链（InterceptorChain）被划分为几个阶段（Phases）。每个拦截器的阶段（Phases）定义一般在拦截器的构造方法中。每个阶段（Phases）可能包含多个拦截器。传入的拦截器链，将

有几个阶段，其阶段信息如表 14-3 所示。

表 14-3　传入拦截器链的阶段和功能

Phase（阶段）	功能
RECEIVE	传输级处理
(PRE/USER/POST)_STREAM	数据流级处理/转换
READ	这通常会发生在读取信息头的阶段
(PRE/USER/POST)_PROTOCOL	协议处理，如JAX-WS的SOAP处理程序
UNMARSHAL	请求的解组
(PRE/USER/POST)_LOGICAL	解组请求的处理
PRE_INVOKE	预调用操作
INVOKE	服务调用
POST_INVOKE	即将离任服务的调用链

即将离任的拦截器链也有几个阶段，其内容如表 14-4 所示。

表 14-4　离任拦截器链的阶段和功能

Phase（阶段）	功能描述
SETUP	为以下几个阶段进行任何设置
(PRE/USER/POST)_LOGICAL	编组对象的处理
PREPARE_SEND	连接的开放
PRE_STREAM	
PRE_PROTOCOL	杂项协议行动
WRITE	协议消息的写法，如SOAP信封
MARSHAL	对象的编组
(USER/POST)_PROTOCOL	协议消息的处理
(USER/POST)_STREAM	字节层次消息的处理
SEND	最后消息的发送和传输流关闭

14.2.1.1　CXF 的 InterceptorProviders 使用

Apache CXF 内部几个不同的组件可以对一个 InterceptorChain（拦截器链）提供多种 Interceptors（拦截器）。InterceptorProvider 接口的方法如下：

```
public interface InterceptorProvider {
    List<Interceptor> getInInterceptors();
    List<Interceptor> getOutInterceptors();
    List<Interceptor> getOutFaultInterceptors();
    List<Interceptor> getInFaultInterceptors();
}
```

对一个拦截器链要添加拦截器，只需把拦截器提供到 InterceptorProvider 即可。

```
MyInterceptor interceptor = new MyInterceptor();
provider.getInInterceptors().add(interceptor);
```

在 Apache CXF 框架内的 InterceptorProvider 有：

- 客户端（Client）
- 服务器端点（Endpoint）
- 服务（Service）
- 总线（Bus）
- 绑定（Binding）

14.2.1.2 编写和配置一个拦截器（Interceptors）

在 Apache CXF 发布程序中有一个例子程序称为 configuration_interceptor。该例子程序说明如何开发一个用户拦截器并配置拦截器到拦截器链中。

1. 编写一个拦截器（Interceptors）

编写一个拦截器是非常简单的任务。开发人员编写的拦截器要么直接继承 AbstractPhaseInterceptor，要么是继承 AbstractPhaseInterceptor 的子类，如 AbstractSoapInterceptor 等。从 AbstractPhaseInterceptor 继承的拦截器可以访问消息接口的方法。例如，Apache CXF 的 AttachmentInInterceptor 用于转化多部分/相关消息到一系列的附件。实现代码如下所示：

```
import java.io.IOException;

import org.apache.cxf.attachment.AttachmentDeserializer;
import org.apache.cxf.message.Message;
import org.apache.cxf.phase.AbstractPhaseInterceptor;
import org.apache.cxf.phase.Phase;

public class AttachmentInInterceptor extends AbstractPhaseInterceptor<Message> {
   public AttachmentInInterceptor() {
      super(Phase.RECEIVE);
   }

   public void handleMessage(Message message) {
      String contentType = (String) message.get(Message.CONTENT_TYPE);
      if (contentType != null && contentType.toLowerCase().indexOf("multipart/
related") != -1) {
         AttachmentDeserializer ad = new AttachmentDeserializer(message);
         try {
            ad.initializeAttachments();
         } catch (IOException e) {
            throw new Fault(e);
         }
      }
   }

   public void handleFault(Message messageParam) {
   }
}
```

继承 AbstractPhaseInterceptor 子类的拦截器可以获得比直接从 AbstractPhaseInterceptor 继承的拦截器更具体的消息接口信息。AbstractSoapInterceptor 类是 AbstractPhaseInterceptor 类下其中一个子类。从 AbstractSoapInterceptor 类继承的拦截器可以访问 SOAP 信息头和 SoapMessage 类的版

本信息。例如，Apache CXF 框架的 SoapActionInInterceptor 类用于解析 SOAP 的操作，代码实现如下：

```
import java.util.Collection;
import java.util.List;
import java.util.Map;

import org.apache.cxf.binding.soap.Soap11;
import org.apache.cxf.binding.soap.Soap12;
import org.apache.cxf.binding.soap.SoapMessage;
import org.apache.cxf.binding.soap.model.SoapOperationInfo;
import org.apache.cxf.endpoint.Endpoint;
import org.apache.cxf.helpers.CastUtils;
import org.apache.cxf.interceptor.Fault;
import org.apache.cxf.message.Exchange;
import org.apache.cxf.message.Message;
import org.apache.cxf.phase.Phase;
import org.apache.cxf.service.model.BindingOperationInfo;
import org.apache.cxf.service.model.OperationInfo;

public class SoapActionInInterceptor extends AbstractSoapInterceptor {

    public SoapActionInInterceptor() {
        super(Phase.READ);
        addAfter(ReadHeadersInterceptor.class.getName());
        addAfter(EndpointSelectionInterceptor.class.getName());
    }

    public void handleMessage(SoapMessage message) throws Fault {
        if (message.getVersion() instanceof Soap11) {
            Map<String, List<String>>
headers = CastUtils.cast((Map)message.get(Message.PROTOCOL_HEADERS));
            if (headers != null) {
                List<String> sa = headers.get("SOAPAction");
                if (sa != null && sa.size() > 0) {
                    String action = sa.get(0);
                    if (action.startsWith("\"")) {
                        action = action.substring(1, action.length() - 1);
                    }
                    getAndSetOperation(message, action);
                }
            }
        } else if (message.getVersion() instanceof Soap12) {
            ……
        }
    }

    private void getAndSetOperation(SoapMessage message, String action) {
        if ("".equals(action)) {
            return;
        }

        Exchange ex = message.getExchange();
        Endpoint ep = ex.get(Endpoint.class);

        BindingOperationInfo bindingOp = null;

        Collection<BindingOperationInfo> bops = ep.getBinding().getBindingInfo().
```

```
getOperations();
        for (BindingOperationInfo boi : bops) {
            SoapOperationInfo soi = (SoapOperationInfo) boi.getExtensor(SoapOperationInfo.class);
            if (soi != null && soi.getAction().equals(action)) {
                if (bindingOp != null) {
                    //more than one op with the same action, will need to parse normally
                    return;
                }
                bindingOp = boi;
            }
        }
        if (bindingOp != null) {
            ex.put(BindingOperationInfo.class, bindingOp);
            ex.put(OperationInfo.class, bindingOp.getOperationInfo());
        }
    }
}
```

请注意需要将拦截器的阶段考虑进来，这些代码在拦截器的构造方法中进行，代码如下：

```
public class MyInterceptor extends AbstractSoapInterceptor {
  public MyInterceptor() {
    super(Phase.USER_PROTOCOL);
  }
  ...
}
```

还可以表示本拦截器在同一个阶段运行在其他拦截器的前面或后面，代码如下：

```
public class MyInterceptor extends AbstractSoapInterceptor {
  public MyInterceptor() {
    super(Phase.USER_PROTOCOL);

    // MyInterceptor needs to run after SomeOtherInterceptor
    getAfter().add(SomeOtherInterceptor.class.getName());

    // MyInterceptor needs to run before YetAnotherInterceptor
    getBefore().add(YetAnotherInterceptor.class.getName());
  }
  ...
}
```

可以通过编程方式和配置方式来添加自定义拦截器到拦截器链。

2. 通过编程来添加拦截器

要添加拦截器到服务器，首先要访问服务器对象：

```
import org.apache.cxf.endpoint.Server;
import org.apache.cxf.frontend.ServerFactoryBean;
...

MyInterceptor myInterceptor = new MyInterceptor();

Server server = serverFactoryBean.create();
server.getEndpoint().getInInterceptor().add(myInterceptor);
```

在客户端的程序代码也非常相似：

```
import org.apache.cxf.endpoint.Client;
import org.apache.cxf.frontend.ClientProxy;
...

MyInterceptor myInterceptor = new MyInterceptor();
FooService client = ... ; // created from ClientProxyFactoryBean or generated JAX-WS
client

//You could also call clientProxyFactroyBean.getInInterceptor().add(myInterceptor)
to add the interceptor

Client cxfClient = ClientProxy.getClient(client);
cxfClient.getInInterceptor().add(myInterceptor);

// then you can call the service
client.doSomething();
```

还可以使用注释通过 SEI 或服务类来加入拦截器。当 Apache CXF 创建了服务器或客户端，Apache CXF 的拦截器根据注释将被添加到系统中，代码如下：

```
@org.apache.cxf.interceptor.InInterceptors (interceptors = {"com.example.Test1Interceptor" })
@org.apache.cxf.interceptor.InFaultInterceptors (interceptors = {"com.example.Test2Interceptor" })
@org.apache.cxf.interceptor.OutInterceptors (interceptors = {"com.example.Test1Interceptor" })
@org.apache.cxf.interceptor.InFaultInterceptors  (interceptors  =  {"com.example.
Test2Interceptor", "com.example.Test3Intercetpor" })
@WebService(endpointInterface = "org.apache.cxf.javascript.fortest.SimpleDocLitBare",
          targetNamespace = "uri:org.apache.cxf.javascript.fortest")
public class SayHiImplementation implements SayHi {
  public long sayHi(long arg) {
     return arg;
  }
  ...
}
```

3. 通过配置来添加拦截器

该配置文件说明如何在配置文件中添加拦截器。添加 MyInterceptor 到总线（Bus）的代码如下：

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:cxf="http://cxf.apache.org/core"
      xsi:schemaLocation="
http://www.springframework.org/schema/beans  http://www.springframework.org/schema/
beans/spring-beans.xsd
http://cxf.apache.org/core http://cxf.apache.org/schemas/core.xsd">

   <bean id="MyInterceptor" class="demo.interceptor.MyInterceptor"/>

   <!-- We are adding the interceptors to the bus as we will have only one endpoint/
service/bus. -->

   <cxf:bus>
      <cxf:inInterceptors>
         <ref bean="MyInterceptor"/>
      </cxf:inInterceptors>
      <cxf:outInterceptors>
```

```
            <ref bean="MyInterceptor"/>
        </cxf:outInterceptors>
    </cxf:bus>
</beans>
```

对于嵌入式 Jetty-based 的 Web Services，配置文件通过定义-Dcxf.config.file=server.xml 选项来开始启动服务。

添加 MyInterceptor 到客户端程序中：

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:http="http://cxf.apache.org/transports/http/configuration"
      xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration
http://cxf.apache.org/schemas/configuration/http-conf.xsd
         http://www.springframework.org/schema/beans http://www.springframework.
org/schema/beans/spring-beans.xsd">

    <http:conduit
name="{http://apache.org/hello_world_soap_http}SoapPort9001.http-conduit">
      <http:client DecoupledEndpoint="http://localhost:9990/decoupled_endpoint"/>
    </http:conduit>

    <bean id="MyInterceptor" class="demo.interceptor.MyInterceptor"/>

    <!-- We are adding the interceptors to the bus as we will have only one
endpoint/service/bus. -->

    <bean id="cxf" class="org.apache.cxf.bus.CXFBusImpl">
        <property name="inInterceptors">
            <ref bean="MyInterceptor"/>
        </property>
        <property name="outInterceptors">
            <ref bean="MyInterceptor"/>
        </property>
    </bean>
</beans>
```

要定义客户端的配置文件，使用- Dcxf.config.file = client.xml 选项来启动客户端。

4. Apache CXF 贡献拦截器（Interceptors）

在 Apache CXF 框架中，所有消息处理的功能都是通过拦截器实现的。因此，当调试消息流时，开发人员会在拦截器链遇到大量的拦截器。下面是一些常用的拦截器和功能列表。

（1）默认 JAX-WS 的传入拦截器链（服务器）

- AttachmentInInterceptor：针对 MIME 的边界解析 MIME 头，找到“根”部分重置根部分的输入流，并存储其他部分内容到附件集合。
- StaxInInterceptor：创建从上一个 XMLStreamReader 消息传输的 InputStream。
- ReadHeadersInterceptor：解析 SOAP 头并将 SOAP 头存储在消息中。
- SoapActionInInterceptor：解析“soapaction”消息头和查找消息头，如果一个独特的应用可为完成该行动，则调用该行动。
- MustUnderstandInterceptor：检查 MustUnderstand 标头，并根据需要来进行处理。
- SOAPHandlerInterceptor：根据 JAX-WS 规范的 SOAP 来进行处理。

- LogicalHandlerInInterceptor：根据 JAX-WS 规范的逻辑处理（LogicalHandler）来进行处理。
- CheckFaultInterceptor：中止当前的拦截器并调用故障处理拦截器链，默认设置是 Check。
- URIMappingInterceptor：处理 HTTP GET，抽象化操作信息并设置消息是相同的。
- DocLiteralnInterceptor：检查 SOAP 主体中的第一个元素，确定适当的行动（假如在 SOAPAction 中没有找到），并调用数据绑定来读取数据。
- SoapHeaderInterceptor：完成 SOAP 消息头的数据绑定并映射到参数头。
- WrapperClassInInterceptor：针对打包 doc/lit，DocLiteralInInterceptor 可能是在单个的 JAXB 类中阅读。这个拦截器带出各个部件 bean，该 bean 构造需要调用服务的 Object[]。
- SwAInInterceptor：针对 SOAP 附件，找到适当的附件并在参数列表中分配正确的位置。
- HolderInInterceptor：用于 OUT 和 IN/OUT 参数，JAX-WS 需要创建一个 Holder 对象。这个拦截器创建一些 Holder 对象组并放到参数列表。
- ServiceInvokerInInterceptor：表示确实调用该服务。

（2）默认传出拦截器链（服务器）

- HolderOutInterceptor：用于 IN/OUT 的输出参数，拉出 JAX-WS 的 Holder 对象（在 HolderInInterceptor 创建）的值，并将这些值添加到输出消息列表中。
- SwAOutInterceptor：如果 OUT 组件是 SOAP 附件，就会从列表中取出，并等待以后再使用。
- WrapperClassOutInterceptor：doc/lit 打包，获取剩下的部分并创建一个包装的 JAXB 类来表示整个消息。
- SoapHeaderOutFilterInterceptor：删除标记头。
- SoapActionOutInterceptor：设置 SOAP 的行动。
- MessageSenderInterceptor：返回一个目标对象并设置其输出流、消息头等，为输出传输做准备。
- SoapPreProtocolOutInterceptor：这个拦截器负责建立 SOAP 版本和消息头。所以，它可以提供任何预先协议。
- AttachmentOutInterceptor：如果服务使用附件（如 MTOM 启用），该拦截器设置必需的附件 marshallers 和 mime 组合。
- StaxOutInterceptor：在消息中从 OutputStream 中创建 XMLStreamWriter。
- SoapHandlerInterceptor：JAX-WS 的 SOAPHandler 处理。
- SoapOutInterceptor：SoapOutInterceptor 在 soap:envelope 写入开始元素，并在消息的其他头文件块写入结束元素。也可以在 soap:body 添加开始元素。
- LogicalHandlerOutInterceptor：JAX-WS 的逻辑处理块。
- WrapperOutInterceptor：如果是采用包装 doc/lit 和不使用包装 bean，或是采用 RPC 的 lit，输出包装（wrapper）元素到数据流中。
- BareOutInterceptor：使用数据绑定来写出参数。
- SoapOutInterceptor：通过$SoapOutEndingInterceptor 关闭 soap:body 和 soap:envelope。
- StaxOutInterceptor：通过$StaxOutEndingInterceptor 刷新 stax 流。
- MessageSenderEnding：关闭交换，让传输知道完成了一切操作，应该刷新到客户端。

14.2.2 CXF 的拦截器（Interceptors）的例子

本程序演示 Apache CXF 遵循 JAX-WS 规范并采用自定义的 Interceptor 的客户端和服务器端调用，客户端采用 Apache CXF 的代码优先（Java First）前端应用模式，通过 Spring IoC 注入来实现本案例程序。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，后台服务器端为遵循 JAX-WS 规范的 Endpoint 来实现，使用自定义的 Interceptor 功能。前端访问直接采用 Spring IoC 注入方式，实现的结构如图 14-4 所示。

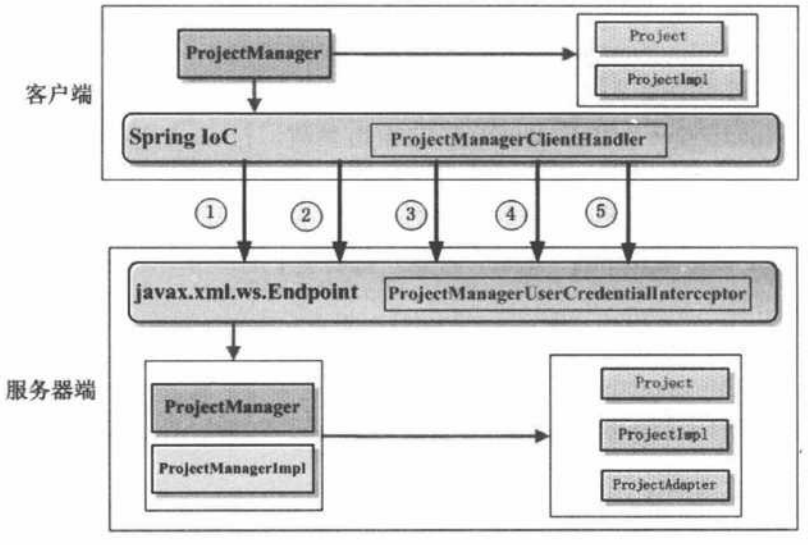


图 14-4　遵循 JAX-WS 规范并采用自定义的 Interceptor 的应用模式

其中：① 表示客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过 SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。在客户端和服务端的 SEI 都是 ProjectManager 接口。

2. 应用程序案例实现步骤

本应用程序案例实现步骤与 Apache CXF 实现 Feature 的例子的实现步骤完全一样。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 14-5 所示。

包目录包括三个部分，第一个包目录 demo.cxf.integration.interceptor.server 是服务器程序目录，第二个包目录 demo.cxf.integration.interceptor.client 是客户端程序目录，第三个包目录 demo.cxf.webservices 是服务器端点接口 SEI 和相关对象等公用部分，服务器端和客户端都有可能引用其内容。

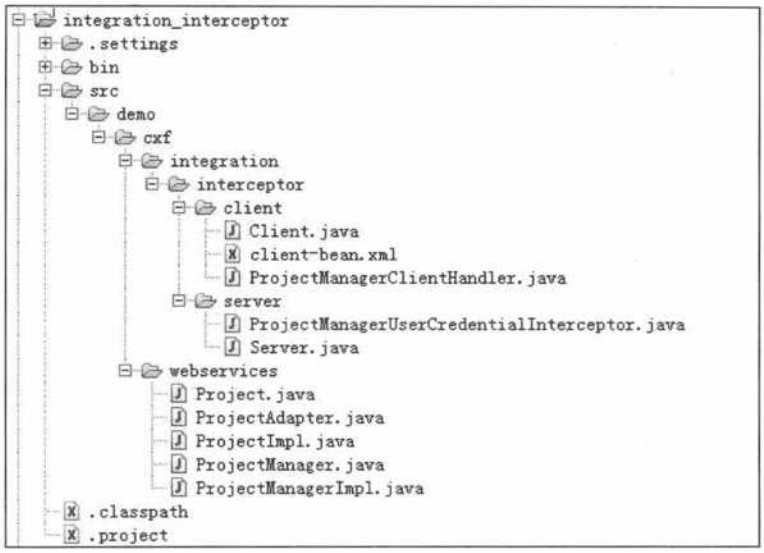


图 14-5　integration_interceptor 程序结构

服务器端点接口（SEI）是公用部分的接口，服务器端和客户端都必须引用，但服务器端和客户端都对公用部分的引用是不同的。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口，而服务器端程序是接口和实现类都要引用。

在编写本程序例子时，主要与第 6 章中“采用 POJO 并基于 JAX-WS 服务的前端实现例子”进行比较，本案例增加 Interceptor 功能实现，下面只列出有区别的两个文件的代码。

（1）服务器端发布程序

服务器端应用程序主要包括 Project 组件。与采用 POJO 并基于 JAX-WS 服务的前端实现例子相比，只有服务器自定义拦截器文件和 Project 组件中的 ProjectManagerImpl 是不同的。

第一个文件是服务器自定义拦截器文件，即 ProjectManagerUserCredentialInterceptor，代码如下：

```
package demo.cxf.integration.interceptor.server;

import javax.xml.namespace.QName;

import org.apache.cxf.binding.soap.SoapMessage;
import org.apache.cxf.binding.soap.interceptor.AbstractSoapInterceptor;
import org.apache.cxf.headers.Header;
import org.apache.cxf.interceptor.Fault;
import org.apache.cxf.phase.Phase;
import org.w3c.dom.Element;
import org.w3c.dom.Node;

public class ProjectManagerUserCredentialInterceptor extends AbstractSoapInterceptor {
    private String userName;
    private String password;

    public ProjectManagerUserCredentialInterceptor() {
        super(Phase.PRE_INVOKE);
    }
```

```
    public void handleMessage(SoapMessage message) throws Fault {
        System.out.println("ProjectManagerUserCredentialInterceptor handleMessage
invoked");
        QName qnameCredentials = new QName("ProjectManagerCredentials");

        // Get header based on QNAME
        if (message.hasHeader(qnameCredentials )) {
            Header header = message.getHeader(qnameCredentials);

            Element elementOrderCredential= (Element) header.getObject();
            Node nodeUser = elementOrderCredential.getFirstChild();
            Node nodePassword = elementOrderCredential.getLastChild();

            if (nodeUser != null) {userName = nodeUser.getTextContent();}
            if (nodePassword != null) {password = nodePassword.getTextContent();}
        }

        System.out.println("userName reterived from SOAP Header is " + userName);
        System.out.println("password reterived from SOAP Header is " + password);

        // Perform dummy validation for John
        if ("Rodger".equalsIgnoreCase(userName) && "password".equalsIgnoreCase(password)) {
            System.out.println("Authentication successful for John");
        } else {
            throw new RuntimeException("Invalid user or password");
        }
    }

    public String getUserName() {return userName;  }
    public void setUserName(String userName) {this.userName = userName;}
    public String getPassword() {  return password;}
    public void setPassword(String password) { this.password = password;}
}
```

Project 组件中的 ProjectManagerImpl.java 是不同的，增加了拦截器注释，代码如下：

```
……
@org.apache.cxf.interceptor.InInterceptors
(interceptors = {"demo.cxf.integration.interceptor.server.
ProjectManagerUserCredentialInterceptor" })
@WebService
public class ProjectManagerImpl implements ProjectManager {
    ……
    }
```

（2）客户端调用程序

客户端应用程序主要包括 Project 组件、客户端程序和客户端自定义拦截器。

第一个文件是客户端的客户端自定义拦截器，即 ProjectManagerClientHandler.java，代码如下：

```
package demo.cxf.integration.interceptor.client;

import javax.xml.namespace.QName;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.apache.cxf.binding.soap.SoapMessage;
```

```
import org.apache.cxf.binding.soap.interceptor.AbstractSoapInterceptor;
import org.apache.cxf.binding.soap.interceptor.SoapPreProtocolOutInterceptor;
import org.apache.cxf.headers.Header;
import org.apache.cxf.interceptor.Fault;
import org.apache.cxf.phase.Phase;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class ProjectManagerClientHandler extends AbstractSoapInterceptor {

    public String userName;
    public String password;

    public ProjectManagerClientHandler() {
        super(Phase.WRITE);
        addAfter(SoapPreProtocolOutInterceptor.class.getName());
    }

    public void handleMessage(SoapMessage message) throws Fault {
            System.out.println("ProjectManagerClientHandler handleMessage invoked");
            DocumentBuilder builder = null;
            try {
                builder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
            } catch (ParserConfigurationException e) {
                e.printStackTrace();
            }
            Document doc = builder.newDocument();
            Element elementCredentials = doc.createElement("ProjectManagerCredentials");
            Element elementUser = doc.createElement("username");
            elementUser.setTextContent(getUserName());
            Element elementPassword = doc.createElement("password");
            elementPassword.setTextContent(getPassword());
            elementCredentials.appendChild(elementUser);
            elementCredentials.appendChild(elementPassword);

            // Create Header object
            QName qnameCredentials =  new QName("ProjectManagerCredentials");
            Header header = new Header(qnameCredentials, elementCredentials);
            message.getHeaders().add(header);
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }
}
```

客户端程序，即 Client.java，代码如下：

```
package demo.cxf.integration.interceptor.client;

import org.apache.cxf.frontend.ClientProxy;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import demo.cxf.webservices.Project;
import demo.cxf.webservices.ProjectImpl;
import demo.cxf.webservices.ProjectManager;

public class Client {
    public Client() {}

    public static void main(String[] args) throws Exception {
        System.out.println("Starting Client");

        ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(
                new String[] { "demo/cxf/integration/interceptor/client/client-bean.
xml" });

        // 获得 SEI 接口
        ProjectManager projectManager = (ProjectManager) context.getBean
("ProjectManagerClient");

        ProjectManagerClientHandler clientInterceptor = new ProjectManagerClientHandler();
        clientInterceptor.setUserName("Rodger");
        clientInterceptor.setPassword("password");
        org.apache.cxf.endpoint.Client cxfClient = ClientProxy.getClient(projectManager);
        cxfClient.getOutInterceptors().add(clientInterceptor);

        //项目的基本情况
        getGreet(projectManager);
        greetingWithParameter(projectManager);
        .......
        System.exit(0);
    }

    //调用 SEI 的 getGreet 方法
    private static void getGreet(ProjectManager projectManager)throws Exception {
        System.out.println(projectManager.greeting());
    }
    ......
}
```

本例首先创建一个服务名称为 javax.xml.ws.Service 的实现类，接着通过 addPort 方法来传入服务的端口类、绑定方式和 Web Services 的地址。最后通过 getPort 方法来获得服务器的 SEI 接口。这就是我们需要的 Web Services 服务接口 ProjectManager。首先要实例化 ProjectManagerClientHandler 对象，这是继承 AbstractSoapInterceptor 抽象类的对象。其次要获得 org.apache.cxf.endpoint 对象。最后调用 org.apache.cxf.endpoint 对象的拦截器链的 add 方法，这样把 ProjectManagerClientHandler 对象加上即可。

后面的工作就是基于这个 ProjectManager 的 SEI 来实现各种业务操作。

4. 测试和运行结果

测试编写好的程序代码，首先要启动服务器程序，即在 Eclipse 上运行 Server.java。然后运行客户端程序。其客户端程序输出结果如下：

```
ProjectManagerClientHandler handleMessage invoked
Hello, This is ProjectManager.
ProjectManagerClientHandler handleMessage invoked
Hello, greet This is ProjectManager.
ProjectManagerClientHandler handleMessage invoked
New ProjectName
ProjectManagerClientHandler handleMessage invoked
defaultProjectID is New ProjectID . defaultProjectName is New ProjectName
ProjectManagerClientHandler handleMessage invoked
ProjectManagerClientHandler handleMessage invoked
ProjectManagerClientHandler handleMessage invoked
DefaultProjectID is New Default ProjectID . DefaultProjectName is New Default
ProjectName
ProjectManagerClientHandler handleMessage invoked
ProjectManagerClientHandler handleMessage invoked
ProjectManagerClientHandler handleMessage invoked
New ProjectName
ProjectManagerClientHandler handleMessage invoked
New ProjectName
ProjectManagerClientHandler handleMessage invoked
ProjectManagerClientHandler handleMessage invoked
ProjectManagerClientHandler handleMessage invoked
```

其服务器端输出结果如下：

```
ProjectManagerUserCredentialInterceptor handleMessage invoked
userName reterived from SOAP Header is Rodger
password reterived from SOAP Header is password
Authentication successful for John
----invoking greeting() ----
ProjectManagerUserCredentialInterceptor handleMessage invoked
userName reterived from SOAP Header is Rodger
password reterived from SOAP Header is password
Authentication successful for John
----invoking greetingWithParameter be called
ProjectManagerUserCredentialInterceptor handleMessage invoked
userName reterived from SOAP Header is Rodger
password reterived from SOAP Header is password
Authentication successful for John
----invoking getProjectName() ----
ProjectManagerUserCredentialInterceptor handleMessage invoked
userName reterived from SOAP Header is Rodger
password reterived from SOAP Header is password
Authentication successful for John
----invoking getDefaultProject----
ProjectManagerUserCredentialInterceptor handleMessage invoked
userName reterived from SOAP Header is Rodger
password reterived from SOAP Header is password
Authentication successful for John
----invoking getAllProject ----
  default projectID: default projectName
  projectID2: projectName2
ProjectManagerUserCredentialInterceptor handleMessage invoked
```

```
userName reterived from SOAP Header is Rodger
password reterived from SOAP Header is password
Authentication successful for John
----invoking setDefaultProject----
ProjectManagerUserCredentialInterceptor handleMessage invoked
userName reterived from SOAP Header is Rodger
password reterived from SOAP Header is password
Authentication successful for John
----invoking getDefaultProject----
ProjectManagerUserCredentialInterceptor handleMessage invoked
userName reterived from SOAP Header is Rodger
password reterived from SOAP Header is password
Authentication successful for John
----invoking getAllProject ----
  default projectID: default projectName
  projectID2: projectName2
ProjectManagerUserCredentialInterceptor handleMessage invoked
userName reterived from SOAP Header is Rodger
password reterived from SOAP Header is password
Authentication successful for John
----invoking registerProject----
ProjectManagerUserCredentialInterceptor handleMessage invoked
userName reterived from SOAP Header is Rodger
password reterived from SOAP Header is password
Authentication successful for John
----invoking getProjectNameByID----
ProjectManagerUserCredentialInterceptor handleMessage invoked
userName reterived from SOAP Header is Rodger
password reterived from SOAP Header is password
Authentication successful for John
----invoking getProjectNameByID----
ProjectManagerUserCredentialInterceptor handleMessage invoked
userName reterived from SOAP Header is Rodger
password reterived from SOAP Header is password
Authentication successful for John
----invoking getAllProject ----
  default projectID: default projectName
  New ProjectID: New ProjectName
  projectID2: projectName2
ProjectManagerUserCredentialInterceptor handleMessage invoked
userName reterived from SOAP Header is Rodger
password reterived from SOAP Header is password
Authentication successful for John
----invoking removeProject, Project ID is: New ProjectID
ProjectManagerUserCredentialInterceptor handleMessage invoked
userName reterived from SOAP Header is Rodger
password reterived from SOAP Header is password
Authentication successful for John
----invoking getAllProject ----
  default projectID: default projectName
  projectID2: projectName2
```

5. 其他内容

上述的所有调用都基于 JAX-WS 规范。

本例代码参见配套资源代码中“第 14 章\integration_interceptor”文件夹。

14.3　CXF 的代理（invoker）

本节介绍 Apache CXF 框架的代理（invoker）功能说明和例子。

14.3.1　CXF 的代理（invoker）功能说明

invoker 允许开发人员定义一个特定的执行方法。这个功能特别有用，针对用户的底层服务对象不是普通的 JavaBeans，而是需要通过一个自定义工厂来进行创建或使用的 JavaBeans。

Apache CXF 框架提供了大量的 invoker 处理简单业务。其中最为理想的是为服务提供单个对象的使用。在这种情况下，单个实例化对象被创建并提供给所有的服务来调用。所提供的 BeanInvoker 涵盖此功能，例子代码如下：

```
Service service = ...;
service.setInvoker(new BeanInvoker(new MyCustomBean(someParams)));
```

可以通过两种方式访问基本服务对象。如果用 ServerFactoryBean 实例化对象来创建服务，这将产生一个服务器对象，可用于获取服务，代码如下：

```
ServerFactoryBean factory = new ServerFactoryBean();
....
Server server = factory.create()
Service service = server.getEndpoint().getService();
```

如果已经创建一个 JAX-WS 规范的服务端点对象，可以按照下面的代码方式来访问服务：

```
EndpointImpl endpoint = (EndpointImpl) Endpoint.publish("http://host/service", new
MyService());
....
Server server = endpoint.getServer();
Service service = server.getEndpoint().getService();
```

下面的例子说明了 invoker 如何让 Apache CXF 框架调用远程无状态会话 Bean 来开放 Web Services。由于要调用其他方法，这个 invoker 将创建一个无状态会话实例化对象。同样的技术可以给需要进行自定义创建/查找的任何对象作为服务进行调用。

invoker 代码的执行情况如下：

```
public class EJBInvoker extends AbstractInvoker{
  private EJBHome home;
  private Method createMethod;
  private static final Object[] EMPTY_OBJECT = new Object[0];

  public EJBInvoker(EJBHome home)
  {
    this.home = home;
    try
    {
      if(!home.getEJBMetaData().isSession()
|| !home.getEJBMetaData().isStatelessSession()){
        throw new IllegalArgumentException("home must be for a stateless session
bean");
      }
```

```
      createMethod = home.getClass().getMethod("create", new Class[0]);
    }
    catch(Exception ex) {
      throw new IllegalArgumentException("Unable to initialize invoker: " + ex);
    }
  }

  public Object getServiceObject(final Exchange context)
  {
    return createMethod.invoke(home, EMPTY_OBJECT);
  }
}
```

invoker 一旦定义，就需要在服务上注册。一旦一个处理进入了持久服务对象，上面例子中的 invoker 就可以按照下面方式进行绑定使用。

```
Service ejbService = ....;
ejbService.setInvoker(new EJBInvoker(ejbHome));
```

如果使用的是一个 EJB3 容器，可以使用下面的方式来进行调用，这是一个简化版本的代码。

```
public class EJB3Invoker extends AbstractInvoker {
    private Object ejb;

    public EJB3Invoker(String jndiName) throws NamingException{
        ejb = new InitialContext().lookup(jndiName);
    }

    public Object getServiceObject(final Exchange context) {
        return ejb;
    }
}
```

除了提供本身的 invoker 进行调用，Apache CXF 框架还可以给其他服务提供执行体。执行体是一种控制服务调度的方式，下面是例子代码。

```
Service service = ....; // look up the service from CXF, or create it
service.setExecutor(new MyExecutor());
```

14.3.2 CXF 的代理（invoker）的实现例子

本程序演示了遵循 Apache CXF API 规范并采用自定义的 invoker 的客户端和服务器端调用，客户端采用 Apache CXF 的代码优先（Java First）前端应用模式，通过 Spring IoC 注入来实现本案例程序。

1. 应用程序案例场景说明和图示

应用场景：在后台有一个项目数据集，初始化时默认有两个项目，然后通过客户端访问服务器，获得项目数据集的数据，同时对项目集进行查询、修改、增加和删除操作。

在这个过程中，后台服务器端由遵循 Apache CXF 框架 API 规范的 Endpoint 来实现。使用自定义的 invoker 代理功能。前端访问直接采用 Spring IoC 注入方式，实现的结构如图 14-6 所示。

其中：① 表示客户端通过 Service 获得 SEI，访问后台的项目基本情况；② 表示客户端通过

SEI 修改默认项目；③ 表示客户端通过 SEI 增加一个项目；④ 表示客户端通过 SEI 删除一个项目；⑤ 表示客户端通过 SEI 查看后台的全部项目情况。在客户端和服务端的 SEI 都是 ProjectManager 接口。

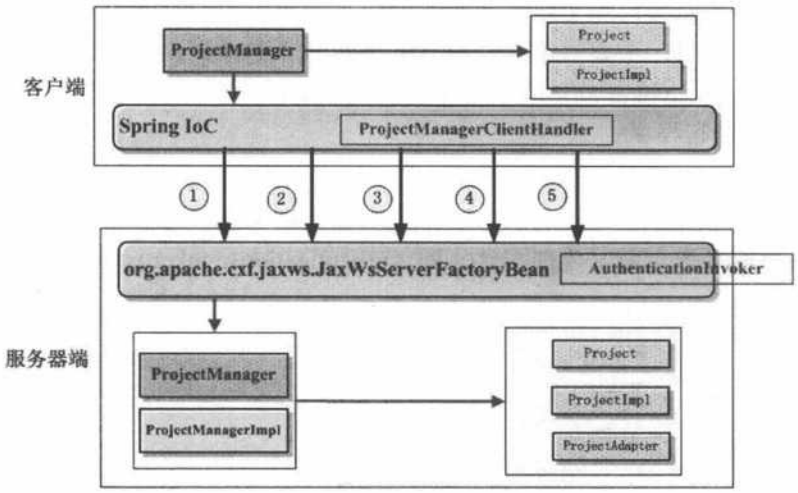


图 14-6　遵循 Apache CXF 规范并采用自定义 invoker 的应用模式

2. 应用程序案例实现步骤

本应用程序案例实现步骤与 Apache CXF 实现 Feature 的例子完全一样。

3. 应用程序案例结构和源码说明

本案例应用程序包括服务器端应用程序和客户端应用程序，其应用程序案例的程序目录结构如图 14-7 所示。

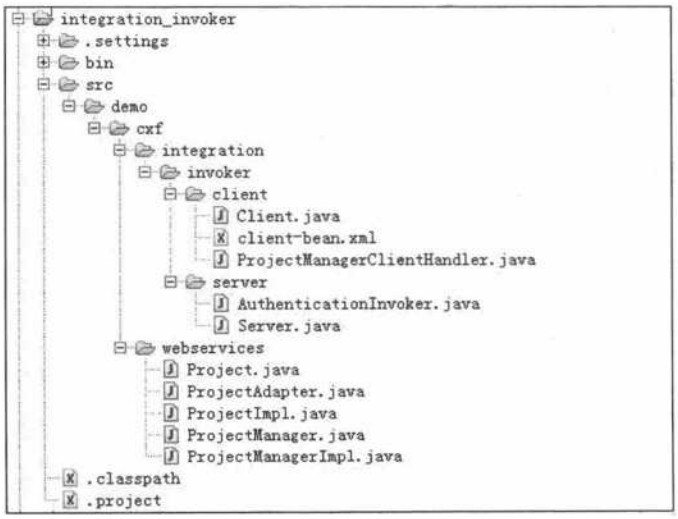


图 14-7　integration_invoker 程序结构

包目录包括三个部分，第一个包目录 demo.cxf.integration.invoker.server 是服务器程序目录，第二个包目录 demo.cxf.integration.invoker.client 是客户端程序目录，第三个包目录 demo.cxf.

webservices 是服务器端点接口 SEI 和相关对象等公用部分，服务器端和客户端都有可能引用其内容。

服务器端点接口（SEI）是公用部分的接口，服务器端和客户端都必须引用。但服务器端和客户端对公用部分的引用是不同的。客户端程序一般只引用接口，如 ProjectManager 和 Project 接口，而服务器端程序是接口和实现类都要引用。

在编写本程序例子时，主要与第 6 章中“采用 POJO 并基于 JAX-WS 服务的前端实现例子”进行比较，本案例增加 invoker 功能实现，下面只列出有区别的文件代码。

（1）服务器端发布程序

服务器端应用程序主要包括 Project 组件、服务器启动类和服务器自定义拦截器。与采用 POJO 并基于 JAX-WS 服务的前端实现例子相比，只有服务器自定义代理文件和服务器发布程序是不同的。

第一个文件是服务器自定义代理文件，即 AuthenticationInvoker.java，代码如下：

```
package demo.cxf.integration.invoker.server;

import java.util.List;

import javax.xml.namespace.QName;

import org.apache.cxf.binding.soap.SoapHeader;
import org.apache.cxf.common.util.ClassHelper;
import org.apache.cxf.headers.Header;
import org.apache.cxf.message.Exchange;
import org.apache.cxf.service.invoker.AbstractInvoker;
import org.apache.cxf.service.model.OperationInfo;
import org.w3c.dom.Element;
import org.w3c.dom.Node;

import demo.cxf.webservices.*;

public class AuthenticationInvoker  extends AbstractInvoker {

    private String userName;
    private String password;
    private Object bean;

    public AuthenticationInvoker(Object bean) {  this.bean = bean;}

    @Override
    public Object invoke(Exchange exchange, Object o) {

        // Get method and class details from the request
        OperationInfo opInfo = exchange.get(OperationInfo.class);
        String methodName = opInfo.getInputName();
        Class<?> realClass = ClassHelper.getRealClass(bean);
        QName qnameOrderCredential = new QName("ProjectManagerCredentials");

        // Perform security check only if the service class is OrderProcessImpl
        // and method name is processOrder
        if (realClass == ProjectManagerImpl.class && "greeting".equals(methodName)){
            dealDetails(exchange,qnameOrderCredential);
        }
        else if (realClass == ProjectManagerImpl.class && "greetingWithParameter".equals
(methodName)){
            dealDetails(exchange,qnameOrderCredential);
        }
```

```
        else if (realClass == ProjectManagerImpl.class && "getProjectName".equals
(methodName)){
            dealDetails(exchange,qnameOrderCredential);
        }
        else if (realClass == ProjectManagerImpl.class && "getProjectNameByID".equals
(methodName)){
            dealDetails(exchange,qnameOrderCredential);
        }
        else if (realClass == ProjectManagerImpl.class && "updateProjectName".equals
(methodName)){
            dealDetails(exchange,qnameOrderCredential);
        }
        else if (realClass == ProjectManagerImpl.class && "setDefaultProject".equals
(methodName)){
            dealDetails(exchange,qnameOrderCredential);
        }
        else if (realClass == ProjectManagerImpl.class && "registerProject".equals
(methodName)){
            dealDetails(exchange,qnameOrderCredential);
        }
        else if (realClass == ProjectManagerImpl.class && "removeProject".equals
(methodName)){
            dealDetails(exchange,qnameOrderCredential);
        }
        else if (realClass == ProjectManagerImpl.class && "getDefaultProject".equals
(methodName)){
            dealDetails(exchange,qnameOrderCredential);
        }
        else if (realClass == ProjectManagerImpl.class && "getAllProject".equals
(methodName)){
            dealDetails(exchange,qnameOrderCredential);
        }

        // Call super class invoke method
        // This will invoke processOrder method
        return super.invoke(exchange,o);
    }

    private void dealDetails(Exchange exchange,QName qnameOrderCredential){
        List list = (List) exchange.getInMessage().get(Header.HEADER_LIST);
        for (int  i = 0 ; i< list.size() ; i++  ) {
            // Get the SOAP header
            SoapHeader header = (SoapHeader) list.get(i);
            if(header.getName().equals(qnameOrderCredential)) {

                Element orderCredential= (Element) header.getObject();
                Node usernamel = orderCredential.getFirstChild();
                Node passwordel = orderCredential.getLastChild();
                if (usernamel != null) {userName = usernamel.getTextContent();}
                if (passwordel != null) {password = passwordel.getTextContent();}
            } else {
                throw new RuntimeException("Request doesn't contain OrderCredentials
namespace");
            }
        }

        System.out.println("userName retrieved from SOAP Header is " + userName);
        System.out.println("password retrieved from SOAP Header is " + password);
```

```
        // Perform dummy validation for John
        if  ("Rodger".equalsIgnoreCase(userName)  &&  "password".equalsIgnoreCase
(password)) {
            System.out.println("Authentication successful for John");
        } else {
            throw new RuntimeException("Invalid user or password");
        }
    }

    @Override
    public Object getServiceObject(Exchange arg0) {    return bean;}
    public String getUserName() {return userName;  }
    public void setUserName(String userName) {this.userName = userName;   }
    public String getPassword() {  return password;    }
    public void setPassword(String password) { this.password = password;  }
}
```

服务器端应用程序中的服务启动程序也不同，即 Server.java 是不同的，增加了代理，代码如下：

```
package demo.cxf.integration.invoker.server;

import org.apache.cxf.jaxws.JaxWsServerFactoryBean;
import demo.cxf.webservices.*;
public class Server {
    public Server() {
        System.out.println("Starting Server");
        ProjectManager projectManager = new ProjectManagerImpl();
        JaxWsServerFactoryBean server = new JaxWsServerFactoryBean();
        server.setServiceBean(projectManager);
        String address = "http://localhost:9000/ProjectManager";
        server.setAddress(address);
        server.setInvoker(new AuthenticationInvoker(projectManager));
        server.create();
    }

    public static void main(String[] args) throws Exception {
        new Server();
        ……
    }
}
```

其中粗体部分是加入代理的程序代码。

（2）客户端调用程序

客户端应用程序主要包括 Project 组件、客户端访问程序和客户端自定义代理。这里仅列出有差别的内容。

第一个文件是客户端的客户端自定义代理，即 ProjectManagerClientHandler.java，代码如下：

```
package demo.cxf.integration.invoker.client;

import javax.xml.namespace.QName;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.apache.cxf.binding.soap.SoapMessage;
import org.apache.cxf.binding.soap.interceptor.AbstractSoapInterceptor;
```

```
import org.apache.cxf.binding.soap.interceptor.SoapPreProtocolOutInterceptor;
import org.apache.cxf.headers.Header;
import org.apache.cxf.interceptor.Fault;
import org.apache.cxf.phase.Phase;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class ProjectManagerClientHandler extends AbstractSoapInterceptor {
    public String userName;
    public String password;;

    public ProjectManagerClientHandler() {
        super(Phase.WRITE);
        addAfter(SoapPreProtocolOutInterceptor.class.getName());
    }

    public void handleMessage(SoapMessage message) throws Fault {
            System.out.println("ProjectManagerClientHandler handleMessage invoked");
            DocumentBuilder builder = null;
            try {
                builder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
            } catch (ParserConfigurationException e) {
                e.printStackTrace();
                throw new Fault(e);
            }
            Document doc = builder.newDocument();
            Element credentialsel = doc.createElement("ProjectManagerCredentials");
            Element unameel = doc.createElement("username");
            unameel.setTextContent(getUserName());
            Element pwdel = doc.createElement("password");
            pwdel.setTextContent(getPassword());
            credentialsel.appendChild(unameel);
            credentialsel.appendChild(pwdel);

            //Create Header object
            QName orderqname =  new QName("ProjectManagerCredentials");

            Header header = new Header(orderqname,credentialsel);
            message.getHeaders().add(header);
    }

    public String getPassword() {  return password;}

    public void setPassword(String password) {this.password = password;}

    public String getUserName() {return userName;}

    public void setUserName(String userName) {this.userName = userName;}
}
```

客户端访问程序，即 Client.java，代码如下：

```
package demo.cxf.integration.invoker.client;


import org.apache.cxf.frontend.ClientProxy;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import demo.cxf.webservices.Project;
```

```
import demo.cxf.webservices.ProjectImpl;
import demo.cxf.webservices.ProjectManager;

public class Client {
    public Client() {   }

    public static void main(String[] args) throws Exception {
        System.out.println("Starting Client");

        ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(
                new String[] { "demo/cxf/integration/invoker/client/client-bean.xml" });

        // 获得 SEI 接口
        ProjectManager projectManager = (ProjectManager) context.getBean
("ProjectManagerClient");

        ProjectManagerClientHandler clientInterceptor = new ProjectManagerClientHandler();
       clientInterceptor.setUserName("Rodger");
       clientInterceptor.setPassword("password");
       org.apache.cxf.endpoint.Client cxfClient = ClientProxy.getClient(projectManager);
       cxfClient.getOutInterceptors().add(clientInterceptor);


        //项目的基本情况
        getGreet(projectManager);
        greetingWithParameter(projectManager);
        ……
        System.exit(0);
    }

    //调用 SEI 的 getGreet 方法
    private static void getGreet(ProjectManager projectManager)throws Exception {
        System.out.println(projectManager.greeting());
    }
    ……
}
```

本例首先创建一个服务名称为 javax.xml.ws.Service 的实现类，接着通过 addPort 方法来传入服务的端口类、绑定方式和 Web Services 的地址。最后通过 getPort 方法来获得服务器的 SEI 接口。这就是我们需要的 Web Services 服务接口 ProjectManager。首先要实例化 ProjectManagerClientHandler 对象，这是继承 AbstractSoapInterceptor 抽象类的对象，其次要获得 org.apache.cxf.endpoint 对象，最后调用 org.apache.cxf.endpoint 对象的拦截器链的 add 方法，这样把 ProjectManagerClientHandler 对象加上即可。

后面的工作就是基于这个 ProjectManager 的 SEI 实现各种业务操作。

4. 测试和运行结果

测试编写好的程序代码，首先要启动服务器程序，即在 Eclipse 上运行 Server.java，然后运行客户端程序。其客户端程序输出结果如下：

```
ProjectManagerClientHandler handleMessage invoked
Hello, This is ProjectManager.
ProjectManagerClientHandler handleMessage invoked
Hello, greet This is ProjectManager.
ProjectManagerClientHandler handleMessage invoked
```

```
default projectName
ProjectManagerClientHandler handleMessage invoked
defaultProjectID is default projectID . defaultProjectName is default projectName
ProjectManagerClientHandler handleMessage invoked
ProjectManagerClientHandler handleMessage invoked
ProjectManagerClientHandler handleMessage invoked
DefaultProjectID is New Default ProjectID . DefaultProjectName is New Default
ProjectName
ProjectManagerClientHandler handleMessage invoked
ProjectManagerClientHandler handleMessage invoked
ProjectManagerClientHandler handleMessage invoked
New ProjectName
ProjectManagerClientHandler handleMessage invoked
New ProjectName
ProjectManagerClientHandler handleMessage invoked
ProjectManagerClientHandler handleMessage invoked
ProjectManagerClientHandler handleMessage invoked
```

其服务器端输出结果如下：

```
----invoking greeting() ----
----invoking greetingWithParameter be called
----invoking getProjectName() ----
----invoking getDefaultProject----
----invoking getAllProject ----
  default projectID: default projectName
  projectID2: projectName2
----invoking setDefaultProject----
----invoking getDefaultProject----
----invoking getAllProject ----
  default projectID: default projectName
  projectID2: projectName2
----invoking registerProject----
----invoking getProjectNameByID----
----invoking getProjectNameByID----
----invoking getAllProject ----
  default projectID: default projectName
  New ProjectID: New ProjectName
  projectID2: projectName2
----invoking removeProject, Project ID is: New ProjectID
----invoking getAllProject ----
  default projectID: default projectName
  projectID2: projectName2
```

5. 其他内容

上述的所有调用都基于 JAX-WS 规范。

本例代码参见配套资源代码中“第 14 章\integration_invoker”文件夹。

14.4　CXF 的 MER（Multiplexed EndpointReferences）

Apache CXF 的 MER（复用 EndpointReferences）支持单个的、无状态的服务实例表示或模拟多个服务实例。单个服务并不维护本身状态，而当使用当前目标的 EPR（EndpointReference）来识别实例数据的属性时，另外附带一个 EPR 状态。这是一种应用程序级别的复用形式。

为了说明这一点，考虑一个简单的 Web Services 调用 Number，表示一个整数，该方法支持

单一操作 isEven，当输入数字为偶数则返回 true。返回端点引用（EPR）号码通过一个操作创建（int）一个 NumberFactory 服务。

对于 NumberFactory.create 方法而言，一个典型的实现将实例化对象并发布为每个请求的整数值端点，相应的 EPR 返回给调用者。

下面是方法 NumberFactory.create 的核心代码：

```
EndpointReferenceType create(int val) {
  synchronized (this) {
    if (!servicePublished) {
      publishSingleInstanceServant(SERVICE_QNAME);
      servicePublished = true;
    }
  }
  String state = String.valueOf(val);
  String portName = null; // unless there are multiple ports in the service
  return EndpointReferenceUtils.getEndpointReferenceWithId(SERVICE_QNAME,
          portName, state, BusFactory.getDefaultBus());
}
```

下面是方法 Number.isEven 的核心代码：

```
@Resource
   protected WebServiceContext wsContext;
   public boolean isEven() {
      MessageContext messageContext = wsContext.getMessageContext();
      String state = EndpointReferenceUtils.getCurrentEndpointReferenceId(messageContext);
      int val = Integer.parseInt(state);
      return valIsEven(val);
   }
```

在 cxf.xml 中设置 multiplexWithAddress 属性为“true”，可以启用 HTTP。

```
<bean name="<port name>.http-destination" abstract="true">
  <property name="multiplexWithAddress" value="true"/>
</bean>

<!-- or for all http ports (using a wildcard) -->
<bean name="*" abstract="true" class="org.apache.cxf.transport.http_jetty.
JettyHTTPDestination">
  <property name="multiplexWithAddress" value="true"/>
</bean>
```

例子 JAX-WS 客户端代码使用 EPR 访问 Number 实例化对象的代码如下：

```
NumberFactoryService service = new NumberFactoryService();
   NumberFactory factory = service.getNumberFactoryPort();
   EndpointReferenceType epr = factory.create("20");
   NumberService numService = new NumberService();
   ServiceImpl serviceImpl = ServiceDelegateAccessor.get(numService);
   Number num = (Number)serviceImpl.getPort(epr, Number.class);
   ...
```

14.5　CXF 的基础服务

Apache CXF 框架有 4 个基础类来代表服务，明细如表 14-5 所示。

表 14-5　Apache CXF中 4 个基础类和服务功能

类	功能
Service	该服务包含了服务模式，Service是一个Service本身及其方法的抽象代表。同时也包含其他信息，如绑定信息、调用者、执行者、服务属性和拦截器，等等
Endpoint	Endpoint代表一个可以接收消息的端点。其主要职能是特定于该端点（例如，SOAP的拦截器）进行拦截，并监听可能发生的故障。客户端和服务器端都使用Endpoint类
Client	Client是一个管理客户端上消息流的客户端实现
Server	Server表示在Apache CXF框架内的服务器端点。Server提供了启动或停止服务器的方法，并实现访问Endpoint

Apache CXF 框架还包括一些工厂类来协助客户端和服务器的创建，如表 14-6 所示。

表 14-6　Apache CXF基础工厂类、功能及其实现

类	功能	实现
AbstractServiceFactoryBean	AbstractServiceFactoryBean可以采用WSDL模式或编程模式来创建Web Services。它同时还可以设置诸如数据绑定或拦截器等的属性。AbstractServiceFactoryBean隐藏在最常见的Client/ServerFactoryBean实现的背后	ReflectionServiceFactoryBean, JaxWsServiceFactoryBean
ServerFactoryBean	创建一个服务器的端口	ServerFactoryBean, JaxWsServerFactoryBean
ClientFactoryBean	创建一个客户端的端口	ClientFactoryBean, JaxWsClientFactoryBean
ClientProxyFactoryBean	围绕着客户端创建一个Java代理	ClientProxyFactoryBean, JaxWsProxyFactoryBean

在 Apache CXF 框架内基础服务工厂 bean 是 ReflectionServiceFactoryBean。它提供了一种方法映射一个类到服务。如果开发人员对如何控制映射产生兴趣，可以提供并扩展自己的服务配置。大多数服务的配置都是这样实现的。例如，在 JAX-WS 前端模式应用中，JaxWsServiceConfiguration 通过检查 JAX-WS 注释来控制如何映射服务。

开发人员可以添加自己的服务配置，代码如下：

```
MyServiceConfiguration config = new AbstractServiceConfiguration() {
... // your implementation
};
ReflectionServiceFactoryBean serviceFactory = new ReflectionServiceFactoryBean();
serviceFactory.getServiceConfigurations().add(0, config);

ServerFactoryBean svrFactory = new ServerFactoryBean();
svrFactory.setServiceFactory(serviceFactory);
...
Server svr = svrFactory.create();
```

14.6 CXF 的服务路由（Service Routing）

Bus 是 Apache CXF 架构的主干，为共享资源提供了一个可配置的场所，其作用类似于 Spring 的 ApplicationContext。这些共享资源包括 WSDL 管理器、绑定工厂等。通过对 Bus 进行扩展，可以方便地载入自己的资源，或替换现有的资源。默认 Bus 实现基于 Spring，通过依赖注入，将运行时组件串起来。Bus 的创建由 BusFactory 负责，默认是 SpringBusFactory，对应于默认 Bus 实现。在构造过程中，SpringBusFactory 会搜索 META-INF/cxf（就包含在 CXF 的 Jar 中）下的所有配置文件。

在开发过程中可能有这样的情况发生，即开发人员新开发一个新版本的 Web Services，但是仍然要保持旧版本 Web Services 的运行。这里说明如何开发一个简单的 Web Services 路由器来适应不同版本的服务调用。

对于更复杂的路由要求，不妨看看 Apache Camel 是如何全力支持 EIP 模式的。

一个常见的做法是在 Web Services 使用的 XML 文件中明确划定 XML 命名空间兼容的版本。例如：

```
<wsdl:types>
      <schema
          targetNamespace="http://apache.org/2007/03/21/hello_world_xml_http/mixed/types"
          xmlns="http://www.w3.org/2001/XMLSchema">
          <element name="sayHi">
              <complexType />
          </element>
          ……
      </schema>
  </wsdl:types>
```

在许多可能实现的服务路由的不同方案中，一个简单的方法（“简单”是因为开发人员必须写上一定量的程序代码量，但需要确实在一定程度上熟悉 Apache CXF 框架的内部结构）是通过编写 Apache CXF 的拦截器来充当一个路由器。

首先，需要有一个中介服务来注册一个虚拟的拦截器。这个中介拦截器处于拦截器链的开始阶段，确保中介拦截器是第一个被消息管道调用的拦截器。当中介拦截扫描传入的消息，schema 命名空间就根据用户编程策略引导到所定义的消息端点。

下面让我们看看程序代码。

例 1：服务器——这个服务器有三个端点，一个端点是虚拟服务，另外两个端点是 Greeter 服务的不同版本。

```
import javax.xml.ws.Endpoint;

import org.apache.cxf.jaxws.EndpointImpl;
import org.apache.cxf.testutil.common.AbstractBusTestServerBase;
import org.apache.hello_world_mixedstyle.GreeterImplMixedStyle;

public class Server extends AbstractBusTestServerBase {
    protected void run() {
        //implementor1 and implementor2 are published using local transport
        Object implementor1 = new GreeterImplMixedStyle();
        String address1 = "local://SoapContext/version1/SoapPort";
```

```
        Endpoint.publish(address1, implementor1);

        Object implementor2 = new GreeterImplMixedStyle();
        String address2 = "local://SoapContext/version2/SoapPort";
        Endpoint.publish(address2, implementor2);

        //A dummy service that acts as a routing mediator
        Object implementor = new GreeterImplMixedStyle();
        String address = "http://localhost:9027/SoapContext/SoapPort";
        javax.xml.ws.Endpoint jaxwsEndpoint = Endpoint.publish(address, implementor);

        //Register a MediatorInInterceptor on this dummy service
        EndpointImpl jaxwsEndpointImpl = (EndpointImpl)jaxwsEndpoint;
        org.apache.cxf.endpoint.Server server = jaxwsEndpointImpl.getServer();
        org.apache.cxf.endpoint.Endpoint endpoint = server.getEndpoint();
        endpoint.getInInterceptors().add(new MediatorInInterceptor());
    }

    public static void main(String[] args) {
        try {
            Server s = new Server();
            s.start();
        } catch (Exception ex) {
            ex.printStackTrace();
            System.exit(-1);
        } finally {
            System.out.println("done!");
        }
    }
}
```

例 2：中介拦截器。

```
import java.io.BufferedInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.List;

import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamConstants;
import javax.xml.stream.XMLStreamException;
import javax.xml.stream.XMLStreamReader;

import org.apache.cxf.Bus;
import org.apache.cxf.binding.soap.SoapMessage;
import org.apache.cxf.binding.soap.SoapVersion;
import org.apache.cxf.binding.soap.SoapVersionFactory;
import org.apache.cxf.bus.CXFBusFactory;
import org.apache.cxf.endpoint.Server;
import org.apache.cxf.endpoint.ServerRegistry;
import org.apache.cxf.interceptor.InterceptorChain;
import org.apache.cxf.interceptor.StaxInInterceptor;
import org.apache.cxf.message.Message;
import org.apache.cxf.phase.AbstractPhaseInterceptor;
import org.apache.cxf.phase.Phase;
import org.apache.cxf.staxutils.DepthXMLStreamReader;
import org.apache.cxf.staxutils.StaxUtils;
import org.apache.cxf.transport.MessageObserver;
```

```
public class MediatorInInterceptor extends AbstractPhaseInterceptor<SoapMessage> {
   public MediatorInInterceptor() {
      super();
      setPhase(Phase.POST_STREAM);
      addBefore(StaxInInterceptor.class.getName());
   }

   public void handleMessage(SoapMessage message) {
      String schemaNamespace = "";
      InterceptorChain chain = message.getInterceptorChain();

      //scan the incoming message for its schema namespace
      try {
         //create a buffered stream so that we get back the original stream after scaning
         InputStream is = message.getContent(InputStream.class);
         BufferedInputStream pis = new BufferedInputStream(is);
         pis.mark(pis.available());
         message.setContent(InputStream.class, pis);

         //TODO: process attachements

         String encoding = (String)message.get(Message.ENCODING);
         XMLStreamReader reader = XMLInputFactory.newInstance().createXMLStreamReader
(pis, encoding);
         DepthXMLStreamReader xmlReader = new DepthXMLStreamReader(reader);

         if (xmlReader.nextTag() == XMLStreamConstants.START_ELEMENT) {
            String ns = xmlReader.getNamespaceURI();
            SoapVersion soapVersion = SoapVersionFactory.getInstance().getSoapVersion(ns);
            //advance just past header
            StaxUtils.toNextTag(xmlReader, soapVersion.getBody());
            //past body.
            xmlReader.nextTag();
         }

         schemaNamespace = xmlReader.getName().getNamespaceURI();
         pis.reset();
      } catch (IOException e) {
         e.printStackTrace();
      } catch (XMLStreamException e) {
         e.printStackTrace();
      }

      //Look up for all available endpoints registered on the bus
      Bus bus = CXFBusFactory.getDefaultBus();
      ServerRegistry serverRegistry = bus.getExtension(ServerRegistry.class);
      List<Server> servers = serverRegistry.getServers();

      //if the incoming message has a namespace contained "2007/03/21", we redirect
//the message
      //to the new version of service on endpoint "local://localhost:9027/
//SoapContext/version2/SoapPort"
      Server targetServer = null;
      for (Server server : servers) {
         targetServer = server;
         String address = server.getEndpoint().getEndpointInfo().getAddress();
         if (schemaNamespace.indexOf("2007/03/21") != -1) {
            if (address.indexOf("version2") != -1) {
```

```
                break;
            }
        } else if (address.indexOf("version1") != -1) {
            break;
        }
    }

    //Redirect the request
    MessageObserver mo = targetServer.getMessageObserver();
    mo.onMessage(message);

    //Now the response has been put in the message, abort the chain
    chain.abort();
  }
}
```

这里要注意两件事情：

该 MediatorInInterceptor 是 SOAP 绑定，开发人员可以编写一个类似 XML 绑定的拦截器。

开发人员在拦截器末尾处调用 chain.abort 方法来停止虚拟服务的所有下一步操作。

在这个例子中，implementor1 和 implementor2 利用当地传输协议，即使用类似“local://SoapContext/version1/SoapPort”的表示。该 MediatorInInterceptor 查找 ServerRegistry，找到被服务器托管的端点以便重新调度。

该 MediatorInInterceptor 在 StaxInInterceptor 之前被设置为 POST_STREAM 阶段，这样可以确保它是第一个被调用的拦截器。

第 15 章　CXF 的部署和发布

本章主要说明 Apache CXF 框架的部署和发布。首先介绍 Apache CXF 框架在应用服务器的具体配置，其次讲述如何在 Spring 内嵌入 Apache CXF 框架。

本章内容：

1. 应用服务器的具体配置指南，主要针对的服务器有 Tomcat、JBoss、WebLogic、Websphere 和 OC4J 等。
2. 讲述如何在 Spring 内嵌入 Apache CXF。

15.1　应用服务器的具体配置指南

Apache CXF 框架支持的应用服务器包括 Tomcat、JBoss、WebLogic、Websphere 和 Oracle OC4J 等。

15.1.1　Tomcat

本书所有应用案例基本上都采用 Tomcat 服务器。对于在 Tomcat 服务器上的部署，这里就不再进行详细说明。

15.1.2　JBoss

对于 Apache CXF 框架在 JBoss 的部署，首先要把 war 打包成为 ear。

当把 war 打包成为 ear 时，需要把 JBoss 的具体文件（jboss-app.xml）添加到文件夹$EAR/META-INF 来定义类加载器的位置。

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-app>
<loader-repository>apache.cxf:loader=spring_http.ear
<loader-repository-config>java2ParentDelegation=false</loader-repository-config>
</loader-repository>
</jboss-app>
```

如果开发人员使用的 LinkageErrors 涉及 QName 类，尝试重新打包没有 javax.xml.namespace.QName 类的 stax-api.jar。（在 JBoss 4.0.5GA 以前）这一版本的冲突主要包含在 JBoss 的 lib 和 lib/endorsed 目录中。

15.1.3　WebLogic

在 WebLogic 部署 CXF WAR 文件有两种方式。

1. 把 jars 放置到 endorsed 目录下

将 geronimo-ws-metadata_2.0_spec-1.1.1.jar 放置到$Weblogic_Home/jdk_../jre/lib/endorsed 目录下。这样可以实现在 WebLogic 部署 Apache CXF 框架。（不推荐这种方式，这种部署有可能打乱应用服务器自身的环境，推荐使用下面方式，因为下一种部署影响的只是一个单独的模块。）

2. 打包 war 到 ear，用 weblogic-application.xml 文件部署 ear

在 META-INF 目录中创建一个标准 J2EE 的 application.xml 文件（例子是$CXF_HOME/samples/java_first_spring_support）。其文件内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC"-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN"
    "http://java.sun.com/dtd/application_1_3.dtd">
<application>
<display-name>spring_http</display-name>
<module>
<web>
<web-uri>spring_http.war</web-uri>
<context-root>spring</context-root>
</web>
</module>
</application>
```

在 META-INF 文件夹中创建一个 weblogic-application.xml 文件，其内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<weblogic-application xmlns="http://www.bea.com/ns/weblogic/90">

<application-param>
<param-name>webapp.encoding.default</param-name>
<param-value>UTF-8</param-value>
</application-param>

<prefer-application-packages>
<package-name>javax.jws.*</package-name>
</prefer-application-packages>

</weblogic-application>
```

在上面例子中，prefer-application-packages 元素创建了基于 WebLogic 应用服务器下的过滤类装载器。在包名内所有类名称列表都是 WebLogic 系统要搜索并要求加载的。如果某一个类的包不在此列，WebLogic 将尝试加载自身的（可能更早）的版本，所以如果部署过程中发现了任何特定类的部署错误信息，都可以在部署中增加特定类的包。

另外，开发人员也可以指定在 WebLogic 应用服务器上 application.xml 文件的其他选项，请参阅更多信息的 WebLogic 指导。

运行“jar cvf ...”命令来创建 ear，然后部署。

15.1.4 WebSphere

截止到现在为止，只有一种方式可以实现在 WebSphere 中加载 Apache CXF，只要加入 jars 到“endorsed”文件夹中，就可以让 Apache CXF 框架在 WebSphere 应用服务器容器中工作。

注意：WebSphere 应用服务器本身也提供了对 Web Services 的支持。一旦 WebSphere 启动了 Web Services 功能，那 Apache CXF 就很难工作了。所以，如果要把 Apache CXF 部署到 WebSphere，前提条件是 WebSphere 不启动 Web Services 功能，否则两者有一定的冲突。

（1）把 Apache CXF 框架的 jars 放置到 endorsed 目录下。
（2）将 wsdl4j-1.6.1.jar 放置到$WebSphere_HOME/java/jre/lib/endorsed 文件夹中。
（3）在 WebSphere 控制台，找到具体的企业应用程序，单击 Class loading and update detection。
（4）选中 Classes loaded with application class loader first 选项。
（5）选中 Class loader for each war file in application 选项。

对于 WebSphere 应用服务器，如何添加自己的类加载器？

如果把 wsdl4j-1.6.1 jar 放到$WAS_HOME/java/jre/lib/endorsed 文件夹中，所有的应用程序将依赖当前系统的 wsdl4j 版本。另一种解决方案是创建一个服务器，在开发人员的父类加载器之前进行加载，创建一个新的类加载器与开发人员的 wsdl4j 版本共享库，并添加到开发人员这个新的类加载器。wsdl4j 版本只可用于上述创建的特定服务器，不影响其他应用服务器上运行。

实现步骤：

（1）在 WAS 控制台选择 Environment > Shared Libraries。
（2）选择程序库可视的范围。
（3）单击 New，设定值如“名称= MYAPP_SHARED_LIB，类路径= PATH_TO/wsdl4j-1.6.2.jar”，保存。
（4）选择 Application servers > [your server name] > Java and Process Management > Class loader > New。
（5）选择 Classes loaded with application class loader first 和 Save。
（6）选择新的类加载器，然后单击 Shared library references。
（7）添加共享库（MYAPP_SHARED_LIB），保存并重新启动服务器。

15.1.5 OC4J

Oracle OC4J 自带有高度定制的 XML 处理包，包括 Oracle 的 SAX、StAX、JAXP、JAX-WS、SAAJ、WSDL 及其他。这些框架在 Oracle 专有的 OC4J 都已经得到了实现。这使 Oracle 的产品都具有互操作性。这种方式运行良好并非常成功，以至于它很难引进新的实现，如 Apache CXF 框架。

换一句话说，Apache CXF 框架部署在 Oracle OC4J 比较困难。

15.2 在 Spring 内嵌入 CXF

可以在现有的 Spring 应用中载入 Apache CXF 框架。因为 Apache CXF 框架所有的 XML 配

置文件都是基于 Spring 的 XML 格式文件，这两种配置方式和访问模式是相同的。

Apache CXF 框架包含覆盖各种 CXF 模块的 Spring 配置文件。只需在应用系统中引用这些内容。下面是一个例子，引用 Apache CXF 的核心组件、SOAP 绑定和 Servlet 传输。

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schema/jaxws.xsd">

<import resource="classpath:META-INF/cxf/cxf.xml" />
<import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
<import resource="classpath:META-INF/cxf/cxf-servlet.xml" />
    ...
</beans>
```

要包括所有的 Apache CXF 的模块，可以使用 cxf-all 或通配符表达式：

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xmlns:jaxws=http://cxf.apache.org/jaxws
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schema/jaxws.xsd">

<import resource="classpath:META-INF/cxf/cxf.xml" />
<import resource="classpath*:META-INF/cxf/cxf-extension-*.xml" />
      ...
</beans>
```

cxf-all 唯一不包括的是 servlet 传输。假如要加入 servlet 传输，就必须明确地指定出来。这是因为引用会告诉 Apache CXF 框架不再使用独立的 HTTP 运输，即在“classpath:META-INF/cxf/cxf-extension-http-jetty.xml”的 XML 文件。

这个列表提供了更多关于 Apache CXF 的 Spring 文件清单。需要开发人员在使用 web.xml 时，根据要求确定导入的列表或清单。

参 考 文 献

[1] Apache CXF 官方网站：http://cxf.apache.org/

[2] IBM 官方网站：https://www.ibm.com/developerworks/cn

[3] 微软官方网站：http://www.microsoft.com/zh-cn/default.aspx

[4] Naveen Balani, Rajeev Hathi. Apache CXF Web Service Development. Packt Publishing, 2010.

[5] Kent Ka Iok Tong. Developing Web Services with Apache CXF and Axis2. TipTec Development, 2010.

[6] （美）汉森（Hansen，M.D.）著，成保栋 译. 使用 Java Web 服务构建 SOA. 北京：电子工业出版社，2009.

[7] IBM Patterns: Service-Oriented Architecture and web Services. IBM RedBook. 2004.

[8] 邢少敏，周伯生. SOA 研究进展. 计算机科学，2008，35(9): 13-20.

[9] 朱律玮. SOA 参考架构. 计算机与网络，2008，34(7): 46-47.

[10] 付东普. SOA 技术标准的应用. 计算机与网络，2008，34(3): 124-125.

[11] 凌晓东. SOA 综述. 计算机应用与软件，2007，24(10): 122-124,199.

[12] 刘壮业，姚郑. 面向服务架构若干关键问题研究. 计算机工程与设计，2009，(3): 600-604.

[13] 王楠，刘心雄，陈和平. Web Service 技术研究. 计算机与数字工程，2006，34(7): 88-90,100.

[14] 聂会琴，童恒庆，李锡喆. SOAP 协议综述. 计算机与现代化，2004，(4): 39-42.

[15] 余俊隆，闵联营，龚晓秋. 基于策略的 Web 服务访问控制的研究. 计算机与数字工程，2009，37(1): 81-83,87.