Java™ 与 SOAP

O'Reilly & Associates 公司介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构O'Reilly & Associates公司授权中国电力出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly & Associates公司是世界上在UNIX、X、Internet和其他开放系统图书领域具有领导地位的出版公司，同时是联机出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》（被纽约公共图书馆评为二十世纪最重要的50本书之一）到GNN（最早的Internet门户和商业网站），再到WebSite（第一个桌面PC的Web服务器软件），O'Reilly & Associates一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly & Associates是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly & Associates公司具有深厚的计算机专业背景，这使得O'Reilly & Associates形成了一个非常不同于其他出版商的出版方针。O'Reilly & Associates所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly & Associates还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly & Associates依靠他们及时地推出图书。因为O'Reilly & Associates紧密地与计算机业界联系着，所以O'Reilly & Associates知道市场上真正需要什么图书。

作者简介

Robert Englander 是 MindStream 软件公司（*www.mindstrm.com*）的总裁兼首席工程师。他的工作是为软件体系结构、设计和开发提供咨询，同时也为客户的项目开发基本框架。他致力于组件体系结构和分布式系统领域的工作。

Rob 已使用 Java 和 C++ 为从小商店到大集团公司之类的客户编写了很多软件。他还在业界会议上做过多次演讲，也为杂志和期刊撰写文章。他还是O'Reilly出版公司的《Developing Java Beans》一书的作者。

封面介绍

本书封面上的动物是一种叫做 Pterois volitans 的红色火鱼。这些鱼遍布于印度洋和太平洋中100英尺以下的温暖热带水域。它们喜欢一动不动地呆在海底，通常是在暗礁下，等待猎物（主要是一些小鱼和甲壳类动物）的到来。

红火鱼是鲉科鱼家族的一员——鲉是它们背鳍上的毒刺的名字。由于对保护自己的能力充满自信，火鱼即使在遇到人类时也常常不会回避；相反，它们会将毒刺指向入侵者。它们的自信是有道理的：被它们的毒刺扎一下，对人类来说尽管一般不会致命，但却极其疼痛。

除它们所习惯的海洋环境之外，也常常可以在家庭的热带鱼水族箱中看到红火鱼。它们可以长到16英寸之长，因此往往需要一个相当大的水族箱。不过尽管它们拥有可怕的刺和食肉的天性，它们通常还是非常友善的，而且与其他的鱼类相处得很好——除非那些鱼小到可以被吃掉。

目录

前言 1

第一章 简介 7

RPC 和面向消息的分布式系统 8

自我描述数据 9

XML 10

API 规范与传输级规范 11

SOAP 概述 12

SOAP 实现 13

本书的编写方式 15

开始吧 15

第二章 SOAP 消息 17

HTTP 绑定 17

HTTP 请求 18

HTTP 响应 20

SOAP 封套 22

Envelope 元素 26

Header 元素 26
actor 属性 27
mustUnderstand 属性 28
encodingStyle 属性 30
封套版本 31
Body 元素 31
SOAP Fault 32

第三章 SOAP 数据的编码 36
模式与名称空间 36
串行化规则 39
指定数据类型 43
默认值 56
SOAP 的 root 属性 57

第四章 RPC 形式的服务 59
SOAP RPC 元素 59
一个简单的服务 66
部署服务 68
编写服务客户 79
使用请求级作用域进行部署 89
使用会话级作用域进行部署 91
传递参数 94

第五章 处理复杂数据类型 106
传递数组参数 106
返回数组 116
传递自定义类型参数 121
返回自定义类型 134

第六章 自定义串行化 142
自定义类型的编码 143

第七章 错误和异常 172
在 Apache SOAP 中抛出一个服务器端异常 172
在 Apache SOAP 中创建一个错误监听器 176
在 GLUE 中抛出和捕获异常 182

第八章 可供选择的技术 187
SOAP 消息传递 188
直接编码 200

第九章 SOAP 的互操作性与 WSDL 217
Web 服务描述语言 218
从 Apache SOAP 客户调用 GLUE 服务 229
一个使用 Apache SOAP 的代理服务 236
从 GLUE 客户调用 Apache SOAP 服务 241
访问 .NET 服务 249
编写 Apache Axis 客户 255

第十章 SOAP Header 259
Apache SOAP 的提供者和路由器 260
替换提供者和路由器类 261
一个能处理 SOAP 头的 Apache SOAP 服务 266

第十一章 JAX-RPC 与 JAXM 274
JAX-RPC 275
不使用 Ant 来完成工作 276

创建一个JAX-RPC服务 278
创建一个JAX-RPC客户应用程序 285
根据WSDL生成存根 287
动态调用接口 290
用一句话介绍JAXM 290
接下来会是什么? 291

词汇表 293

前言

SOAP（Simple Object Access Protocol，简单对象访问协议），是当前众多分布式计算技术中最新的一个。与其他分布式计算技术不同的是，它是基于XML的，这样就不需要花很多时间来重新定义计算世界。相反，SOAP规范描述了数据的内容和结构等重要方面，以及它们与RPC（Remote Procedure Call，远程过程调用）和消息传递系统（message passing system）这些众所周知的编程模型之间的关系。

这些规范存在于XML世界里。SOAP与特定的编程语言、计算平台或软件开发环境无关。有很多SOAP实现可以提供到各种编程语言（如C#、Perl和Java）的绑定。离开了这些实现，SOAP仍然是抽象的：它是一个没有被具体化的重要概念。然而，只有与特定的软件开发语言相结合，SOAP才能得以生存，本书也体现了这一点。Java天生就适合处理XML，并使它能够更好地建立SOAP服务和客户应用程序。如果你希望使用Java来编写SOAP软件，那么本书正好可以作为你的入门书籍。

本书适合的读者

本书是为那些想知道如何利用Java访问基于SOAP的Web服务以及如何使用Java

创建基于SOAP的Web服务的人编写的。本书的对象是已经熟悉Java的程序设计人员、学生和专业人士，因此本书不打算在Java语言的基本观念或语法上花费时间。如果你对Java还不是很熟悉，可以去买一本关于Java语言的书来看，如：O'Reilly出版公司的《Learning Java》或《Java in a Nutshell》（译注1）。

未来的演变

SOAP规范目前仍在演变之中。这本书是依照SOAP规范的1.1版编写的。尽管其中一些现在提到的概念和技术在未来的SOAP版本中仍将沿用，但很明显，新版本发布时肯定会带来一些很重要的新东西。同时，我们使用的Java实现也在演变。显然，这本书中的描述和例子将会随时间的推移而变得过时甚至荒废（或迟或早），这取决于Web服务技术发展的速度。事实上，预兆已经出现了：本书中的许多例子都是基于Apache SOAP第2版编写的，但它注定要被Apache SOAP第3版（也称为Axis）所取代。我们将在第九章中简要介绍现在可以得到的它的一个早期版本。尽管JAX RPC和JAXM API规范本身还在发展之中，但Axis还是被设计成支持这些规范。Sun公司已针对这些规范提供了一个测试版本的参考实现（将在第十一章中讨论）；而这个版本的出现时间比Axis的最新版本更早。同样，如果认为JAX Pack规范的定稿将会终止这一系列演变过程，那也是很愚蠢的。不过，不管怎样，当这一切不可避免地到来的时候，但愿你的知识能够使你足以跟得上这种变化的步伐。

本书的组织

本书中各章的内容循序渐进，因此最好的方案是按章节顺序依次阅读。

第一章，简介

这一章对SOAP进行了简要介绍，包括相关的技术、存在的问题以及与其他解决方案的对比等。本章还介绍了Apache SOAP和GLUE，本书会一直使用这两种SOAP实现。

译注1：本书中文版《Java技术手册》已由中国电力出版社引进出版。

第二章，SOAP消息

这一章介绍了SOAP封套，它是一个携带了客户和服务器之间SOAP事务的有效负载的结构化XML文档。本章全面讲述了SOAP封套的所有方面，包括Header、SOAP的Body元素和Fault。还介绍了SOAP HTTP绑定的一些细节。

第三章、SOAP数据的编码

这一章全面介绍了SOAP事务的数据编码，包括对数据元素进行编码和串行化的规则。它首先介绍了名称空间，然后深入探讨了简单和复杂数据类型的串行化。

第四章，RPC形式的服务

这一章深入讨论了基于SOAP的RPC形式的服务。给出了与服务方法和参数有关的广泛内容，包括服务的部署和活动机制的有关细节。

第五章，处理复杂数据类型

这一章介绍了如何建立包含使用复杂参数和返回值（比如数组和Java bean）方法的服务。它全面讲述了在客户系统和服务器系统上将这些类型映射到Java类的机制。

第六章，自定义串行化

作为对第五章的补充，这一章全面介绍了非标准自定义数据类型的使用。讲述了一些用来传递自定义数据类型参数与返回值的工具和API。本章还详细介绍了编写串行化和反串行化自定义类型Java类的方法。

第七章，错误和异常

这一章讲述了SOAP Fault（错误），以及它与Java异常的关系。它介绍了默认的机制以及生成和扩展错误内容的技术。

第八章，可供选择的技术

这一章首先介绍了SOAP的消息形式的服务，这是一项RPC模型的替代技术。它还讲述了在SOAP封套中传递直接XML的技术，最后介绍了SOAP附件。

第九章，SOAP的互操作性与WSDL

这一章介绍了如何让使用不同技术开发的SOAP客户和服务器能够正确地共同工作。它对WSDL（Web Services Description Language，Web服务描述语言）做了简要介绍。本章还介绍了分别使用Apache SOAP和GLUE建立客户和服务器的例子，以及可以访问微软.NET服务的Java客户的例子，并对Apache Axis做了简要介绍。

第十章，SOAP Header

这一章讨论了SOAP Header的使用，它为在存在于SOAP Body之外的客户与服务之间传递数据提供了一种手段。本章讲述了一个用作其他服务的消息路由器的中间服务的开发过程。为了处理SOAP Header，我们还开发了一些用于扩展Apache SOAP框架的Java类。

第十一章，JAX-RPC与JAXM

这一章讨论了一个正在形成的标准：基于XML的RPC的Java API（JAX-RPC）。它对Sun公司一个早期版本的参考实现做了简要介绍。本章包括了一个服务和一个客户的开发，还讲述了如何使用这些工具来开发代码，以访问由WSDL描述的服务。最后还介绍了JAXM。

排版约定

本书使用下列英文字体约定：

等宽字体（constant width）用于：

- Java程序中的所有代码，包括关键字、操作符、数据类型、常量、方法名称、变量名称、类名、接口名称及Java包名。
- 需要在屏幕上逐字输入的命令行和选项。
- 名称空间。

斜体（*italic*）用于：

- 路径名、文件名和Internet地址，例如域名和URL。斜体还用于表示可执行文件。

将一本书中所有的东西都精细地区分开来，这项工作就像一场痛苦的战争。但是我仍尝试着区别名称空间（使用等宽字体）和URL（使用斜体），尽管两者看起来没什么不同。同样，我还将Java方法名（使用等宽字体，以一对圆括号作为结尾）和从SOAP服务中导出的方法（使用等宽字体，但没有圆括号）区别开来。

建议与评论

请向出版商提出有关本书的一些评论和问题：

美国：

O'Reilly & Associates,Inc.
101 Morris Street
Sebastopol, CA 95472

中国：

100080 北京市海淀区知春路 49 号希格玛公寓 B 座 809 室
奥莱理软件（北京）有限公司

本书拥有相关的网页，其中列出了勘误表、示例或者任何附加信息。你可以通过这个地址访问此页面：

http://www.oreilly.com/catalog/javaxmldatabind

要发表评论或者询问有关本书的技术问题，请向这个地址发送电子邮件：

bookquestions@oreilly.com
info@mail.oreilly.com.cn

有关书籍、会议、资源中心以及 O'Reilly Network 的更多信息，请参见 O'Reilly 公司的网站：

http://www.oreilly.com
http://www.oreilly.com.cn

致谢

我的好朋友 Rinaldo DiGiorgio 直到今天还在鼓励我，让我对 Java 及其相关技术感兴趣。我认为再没有其他任何人能在Java工作上对我产生如此大的影响了。谢谢你，Rinaldo，谢谢你使我走上了一条正确的路。

非常感谢 David Askey 和 Anne Thomas Manes，他们审阅了此书并提供了许多有价值的意见。他们设法找出错误并且提供建议，没有他们的帮助本书不可能变得这么好。感谢Lorraine Pecorelli通读各章并使本书减少了词不达意的问题。还要对本书的编辑Mike Loukides表示最诚挚的谢意。在完成本书的过程中曾出现过很多困难，是Mike 勇于承担义务和忠于事业的精神才使我们的努力变成这本书。同时也要将这份功劳归功于 O'Reilly 的设计和生产人员。

最后，我要谢谢我的家人，Jessica 和 Carolyn，感谢他们的支持。这次我将不感谢我的朋友 ——他们全然没有帮忙!

第一章

简介

在软件发展的历史过程中，新方法的出现时常将已被丢弃的思想又重新使用到主流应用中。每当一个思想被重用时，它以往成功和失败的经验就成为无价之宝，能够帮助改进该思想本身并完善其实现，至少能使它更具实用性。我并不是说我们要穿新鞋走老路，相反，我们要做的是温故而知新。这样做往往会成为新思想和新技术的催化剂。

我们看到的主机和终端构成的集中计算模式，现在摇身一变，以应用服务器和瘦客户机的形式又回来了。我们曾经见过的P-Code概念现在又以Java和Visual Basic这样的解释语言的形式再次出现。软件开发领域的此消彼长就像现实宇宙的变化。经过足够久的时间之后，也许未来有一天你会发现又能用到你今天正在做的工作。

我要表达的观点是（事实上也确实如此）：好思想之所以是好思想，并不在于它是不是新出现的。重要的是要合乎时宜。正因为如此，分布式计算现在才大行其道。分布式计算的观念并不是才出现的，但是却常常被翻出来用。目前普遍使用的基础设施和技术（像Internet、Web浏览器和它们的相关协议）使我们又重新回到了分布式计算，并发展了它。这场变革的最新热点正是Web服务。

Web服务（Web service）主要是一种服务器功能，它通过发布接口的机制来提供可访问的功能。有很多种技术能够实现这样的功能，它们有一个非常重要的共

同点：它们都能够提供可以用标准协议来访问的计算服务。举例来说，你可能找到这样一个股市报价服务，它能根据某公司的股票代码返回该股票市值和交易信息。虽然这是一个非常特定的功能，但这就是Web服务的本质。它们不提供像应用服务器中常见的丰富的功能——它们提供小而集中的功能，当与其他服务联合时，这些功能就会变得很有用。你可以想像这样一个使用Web服务的在线交易应用程序，它的服务范围从股市报价到交易执行再到银行的事务处理。Web服务的优越性在于它最终能够通过结合分布在Web各处的数据位、数据块以及其他服务迅速（或至少，使用最少的开发时间）建立复杂的应用程序。Sun公司过去一直在呐喊“网络就是计算机”，而这个幻想的确正在实现。

RPC和面向消息的分布式系统

在分布式系统中，大多数组成部分都是以松散耦合的实体形式存在的，它们通过彼此通信来完成任务。分布式软件中最普遍使用的模型之一是RPC（Remote Procedure Call，远程过程调用）。RPC系统如此流行的一个理由是它与我们程序员所熟悉的函数/方法调用的语法和语义非常相似。像Java RMI、微软公司的COM以及CORBA这样的技术都使用了这种模型。当然，在把曾经熟悉的本地方法调用改到远程系统上去之前，你还会遇到很多困难，而且即使你解决了所有的问题，它仍然很可能感觉起来像本地的方法调用。通常，一旦方法调用返回，我们就不再关心它是如何发生的了（注1）。API层中为程序员所做的多数抽象化的工作在于它们在分布式系统的实现方面完成了大部分工作。

分布式计算方面流行的另一个模型是消息传递（message passing）。与RPC模型不同，消息传递没有仿效程序语言中函数调用的语法，而是改为在各方之间传递结构化的数据消息。这些消息能在某种程度上对分布式系统的节点进行去耦，而且基于消息的系统通常被证明比基于RPC的系统更具灵活性。然而，这种灵活性有时会让程序员作茧自缚。

注1：我并不是说要你无视正在调用远程系统的事实。想像一下你用数组存取器来遍历一个包含几万个元素的远程数组的结果，对每一个元素都要访问远程系统。效率之低可想而知！

看来好像结合以上两种模型会产生一个合理而且强大的新模型。你是否能想像这样一个系统：它一方面像RPC形式的调用那样简单并且为你所熟悉，一方面又具有消息型系统的灵活性？在我看来，你可能需要这样定义数据格式：既能够对自身进行描述，同时又要遵循一组管理这些描述的标准。嗯……

自我描述数据

在程序设计中，术语“自我描述数据（self-describing data）”是指数据对自身进行了描述。换一种方式表达，如果问题是“自我描述数据是什么？”，那么回答会是：“描述它本身的数据。”这并不是一个很有用的定义。但是一个设计为具有灵活性的数据格式将被许多人使用，这就促使了自我描述数据的产生。

让我们来看一个非常简单的例子。假设我们正在设计一种消息形式的股市报价分布式系统。我们可以将响应消息设计成类似下面的格式：

- 前10个字符包含股票代码，右边由空格填充。
- 接下来的10个字符表示最终价格，右边由空格填充。
- 接下来的10个字符表示交易量，右边由空格填充。
- 接下来的20个字符表示报价的时间戳。
- 接下来的10个字符表示出价（bid price）。
- 接下来的10个字符表示要价（ask price）。

你看出来了吧，这是一个固定格式的消息。它不能由它本身描述，而是由上面列表中的说明来描述。它不具有灵活性。有时不需要任何灵活性——因为它并不适合所有的情况。但是你不想为今后的改进保留一点空间吗？让我们考虑下面这种可能性：最终价格、出价和要价的值可以使用两种不同的格式。第一种格式使用标准的十进制小数，如25.5。另一种格式使用分数，这样同样的值会被编码成$25\frac{1}{2}$。自我描述格式会针对每一个价格字段所使用的格式做出规定。

现在将同样的概念应用到消息的整个结构和其各组成部分中去。这将为消息内容的全面描述带来灵活性。举例来说，你可以使股市报价响应中的某些字段变得可选。假定你在股市收市之后提出报价请求，这时也许返回的出价和要价数据已不再有用。那么我们为什么还要返回这些数据呢？自我描述格式可以描述它自身的内容，因此没有必要返回无用的数据。

为使自我描述数据变得真正有用，每个人都必须使用相同的语言来描述数据。我不是指程序设计语言；我指的是描述和表达的语言。可以说，我们需要的是一个描述和格式化消息的新方式，即新语法。这种新的语法能使用标准的规则来规范消息的格式。事实上，只有这样我们才能看见自我描述数据的真正价值：不在于它能够省去返回的无用数据，而在于所有人都使用相同的语言进行交流。如果我编写了一个使用 11 个字符来表示股票代码的应用程序，它将不能和你的使用 10 个字符的股票服务器沟通。但是如果我们使用自我描述数据进行数据交换，这个问题将基本上不复存在：每一个数据块都能够用某种标准的方式说明它自身的含义，因此这是一个更好的选择，它使得出自不同的人或组织的软件能够相互通信。

XML

其实我们每次使用 Web 浏览器时，都会看到一种自我描述数据的表现形式。HTML是使用标准数据格式的一个很好的例子，使用自我描述元素使它具有相当的灵活性。举例来说，某一段文字所使用的字体和颜色就是由这段文字本身来描述的。这样的自我描述数据通常被称为标记语言（markup language）。它的内容是用它本身的指令来“标记”的。这样做很不错，而且很显然业界对它的接受程度是不可思议的。尽管如此，对于它的设计者没有预料到的那些内容，HTML 还不具有足够的灵活性。这不是在说 HTML 的坏话；这是事实。因为 HTML 不可扩展。

XML（Extensible Markup Language，可扩展标记语言）正好是我们正在寻找的。XML 是层次化的、基于标签的语言，它与 HTML 很相似。对我们来说重要的差别是它可以充分扩展。它允许我们使用标准的方式来描述我们的程序中所专有的数据内容，这个内容设计者预先并不需要知道。举个例子，XML 允许我们创建可

以用来表示上一节中那种股市报价响应消息的内容。它会定义为了完成某种任务所需要遵从的规则，而不必规定一个特定的格式。

我不准备在介绍XML本身上花费时间。如果你对XML不是很熟悉，也许你需要在开始读第二章之前补补课。关于XML和与之相关的主题，市面上已经有很多书。其中一本比较好的入门书是Elliotte Rusty Harold和W. Scott Means所著的《XML in a Nutshell》（O'Reilly公司出版）。对于Java开发者来说，应该特别关注由Brett McLaughlin所著的《Java and XML》（O'Reilly公司出版）一书。

API规范与传输级规范

Java程序员习惯于处理API级的规范，在这个规范里，类、接口、方法等等都有针对特定需要的明确定义。这些规范被设计成与特定实现无关，而是关注于必须实现的抽象。

我们来看看JSM（Java Message Service，Java消息服务）的规范。它充分描述了应用程序用来访问MOM（Message-Oriented Middleware，面向消息的中间件）产品特性的API。标准API产生的动机很简单：如果MOM厂商采用了该API，那么在面对不同的产品提供者时程序设计者的工作就会容易得多。从理论上讲，你可以在不影响代码结果的情况下将JMS实现换成另一个。但实际上，它意味着在某种程度上对产品厂商的桎梏，使它只能在牺牲性能的前提下提供可选的API以充分发挥自己产品的能力和优势。尽管如此，API规范还是存在了很长一段时间，而且它们确实做到了大多数它们想要做的事情。

然而，API规范方法自身的问题使它在软件开发的一个重要方面——互用性方面存在缺陷。使用厂商A的JMS兼容API开发的应用程序将不能与用厂商B的JMS兼容服务器开发的应用程序进行通信。API规范仅仅涉及API，而不管参与各方之间通信的数据格式。这好像是在暗示互用性不如程序设计语法的公共性重要。是的，它是一个折衷的结果，但是它不总是对的。

另一方面，传输级规范考虑了各方之间传输的数据（“在线上”的数据）在其内容和格式上的独特性。它专注于分布式计算交互的表示法，而不是专注于API。因

此可以打保票说，如果你使用多个厂商的实现，那么你所用的API将不会是同一个。然而，你将能够让使用不同厂商的产品开发的分布式系统一起工作，这是一个好机会。如果你正在做Web服务领域内的任何工作，传输级规范方法将是你的盟友；而API规范方法却帮不了太多忙。

SOAP概述

分布式计算世界中最新的进展之一，是一个传输级规范的产生，即SOAP（Simple Object Access Protocol，简单对象访问协议）。该协议相对来说是轻量级的，它基于XML，而且是针对分布式计算环境下的信息交换而设计的。在SOAP世界中没有中心服务器这个概念；所有的节点都被视为相等甚至对等的实体。

SOAP协议由截然不同的几个部分组成。首先是封套（envelope），它用来描述消息的内容以及有关如何处理它的一些线索。第二部分由自定义数据的实例的编码规则组成。这是SOAP最关键的部分之一：这体现了它的可扩展性。最后一部分说明了封套的应用和用来描述RPC调用与响应的数据编码规则，包括作为底层传输协议的HTTP的使用。

目前，在Java中使用SOAP的最常见的用法是RPC形式的分布式计算，但是它并不是惟一的选择。任何传输都能用来传送SOAP消息，尽管HTTP是惟一一个在规范中被描述的协议。目前关于将SMTP、BEEP、JXTA和其他协议用于传送SOAP消息的有意义的话题已在讨论中。

在Java中使用SOAP

SOAP与RMI、CORBA和COM的不同点在于它更专注于内容，竭力去除自身与实现及底层传输之间的耦合性。这对于有关Java的书来说是个很有趣的概念，不是吗？毕竟，实现和传输很可能是使用Java建立的。但是SOAP并不指定是使用Java还是任何其他的实现策略。

事实上SOAP只是一个启用者，它不能做超出它的规范抽象之外的工作。为从

SOAP或任何其他的协议中获益，必须真正实现它们。Java是实现SOAP、创建使用SOAP作为传输数据格式的Web服务以及应用程序的一种很棒的技术。

SOAP实现

当我写这本书的时候已经有了几十个SOAP实现，而且新的还在不断出现。它们中有些是用Java实现的，有些则不是。有些是免费的，有些则不是。当然不可避免地有些很好、而有些则不是。在这里对它们一一进行比较，或者即使只是分出三六九等都是不切实际的。另一方面，把注意力集中在单个的实现上也是不明智的，那会表现出某种偏见，我不打算这样做。作为一个合理的折衷方案，我将选择两个有趣的Java SOAP实现并贯穿本书广泛地使用。这会给你一个机会接触不同的API和编程策略。在第九章和第十一章中，我将违反这条规则并且简要地介绍几个其他的重要SOAP技术。

Apache SOAP

Apache软件基金会（Apache Software Foundation）正在实施一项叫做Apache SOAP的项目。这是一个支持SOAP规范的Java实现，它可以在任何支持Java servlet的服务器上使用。Apache SOAP 2.2版可以从*http://xml.apache.org/soap/index.html*处下载，本书中的例子都是基于它编写的。有四个非常重要的原因使我选择了这个工具：它支持很多的规范、它有相当广泛的用户基础，它的源程序代码公开，并且是免费的。

尽管可以对Apache SOAP应用很多种服务器策略，我还是选择了Apache Tomcat（第3.3版和第4版），它可以从*http://jakarta.apache.org/tomcat/index.html*处得到。其理由与SOAP关系不大，不过它确实在Tomcat下工作得很好。是否使用Tomcat不会对本书的例子带来大的问题，因此如果你喜欢，也可以自由地选择一些其他的服务器技术。

Apache SOAP是在还没有SOAP API标准的时候开发的。SOAP规范并没有指定与某种语言绑定，因此实现者不得不提出他们自己的API。最近，Java社团已开

始为基于SOAP的消息通信和RPC API制定标准，分别是JAXM和JAX-RPC。我们将在第十一章中介绍这些API；它们将使用Axis来实现，Axis是Apache SOAP实现的下一代产品。在理想状况下，JAXM和JAX- RPC标准会及时完工，以使我能对它们做出评价，因为它们几乎肯定将代替为Apache SOAP 2.2而开发的那些API。事实上，虽然标准的API现在才刚刚具体化，但它们将最终成形，这将促使我编写本书的第二版。本书的大部分内容将会把重心集中在你现在能用来编写应用程序的那些技术上。一旦你的头脑中有了相关的概念，再转到不同的API上去将不会很难。

GLUE

GLUE是一家叫做Mind Electric的公司(*www.themindelectric.com*)开发的SOAP实现。它是完全使用Java开发的，适用于多种支持servlet的服务器，也可以使用它自己的HTTP服务器独立运行。本书中的GLUE例子是基于GLUE第2版的，可以在*http://www.themindelectric.com/product/glue/glue.html*处得到。我选择GLUE也有四个理由：它使用了一种对程序员非常友好的方式，它的API与Apache SOAP中的差别很大，它依赖WSDL，而且它对于大多数应用是免费的(注2)。

GLUE API也是专有的。我预期如果使用者需要，GLUE的下一个版本将会采用像JAXM和JAX-RPC这样的标准，当然，目前的GLUE API仍然会被保留。

其他的实现

在第九章中我们将会花一点时间来尝试一下其他的技术。微软公司的.NET是基于SOAP的Web服务领域的主要开发平台，因此我们要看看使用Java编写的应用程序如何通过SOAP与.NET服务进行通信。

Apache软件基金会现在正在开发一个称为Axis的下一代SOAP实现。虽然现在就来讨论Axis的具体细节还为时尚早，但它无疑将取代或包含Apache SOAP的

注2: 请参考GLUE的许可协议。

2.X版。我们将尝试使用Axis目前的版本编写一个简单的Axis应用程序（此外，正如我已经说过的，在第十一章我们将介绍JAXM和JAX-RPC的提议标准，Axis最终将执行这个标准）。

本书的编写方式

很显然这本书不可能包括SOAP技术的每个方面。我的目的是使你对Java软件开发环境下的SOAP技术的主要内容有一个良好的认识。你将会发现对于许多例子，我们不仅介绍了Java源代码，而且也将Java代码运行后产生的SOAP XML语句呈现给你。这会让你产生这样的感觉：不同的API都能够实现它。这样学习SOAP将使你有信心超越本书的范围去探寻我提到的和不曾提到的那些SOAP实现的特性和能力。

不谈安全

有一个特别的领域在本书中没有涉及到，那就是：安全。在谈论一个分布式计算技术时怎能不提到安全呢？答案实际上相当简单：SOAP规范不处理安全问题。目前的实现依赖于主机寄存技术的安全特性。它可能是SSL、基本的HTTP鉴别、代理鉴别或某些其他机制，所有对安全问题的处理是主机寄存技术的一个功能而不是SOAP本身的。有人预计SOAP规范的一个将来版本或一个独立的SOAP安全规范将会解决这个问题，但是现在你还不得不依赖主机寄存技术的支持。

不过，目前的规范的确提到了使用SOAP头的安全方案，你会发现有一些SOAP实现已经遵循这些方案。然而，在标准被制定之前，这种机制将可能在每种实现中都有所不同。

开始吧

如果你准备使用例子，就需要安装所有必需的软件，包括前面提到的SOAP实现JDK和许多其他的支持技术。所有这些软件包都有相当棒的文档说明，因此你在安装方面应该不会有困难。

本书安排各章的规则是，每一章都是在前面各章的基础概念上建立。我建议依次阅读各章，当然那完全由你自己决定。如果你对XML已经很了解，或者甚至对Web服务已有一些认识，那么可以跳过一小部分章节。

第二章

SOAP 消息

所有的SOAP消息都封装在一个称为封套（envelope）的XML文档中，它是保存一条SOAP消息的结构化容器。称为封套是恰当的，因为你完成某项操作所需的所有东西都将放进一个封套中，把它发送给接收者，接收者打开封套并重新构造出原始的内容，以便能完成你所请求的操作。SOAP封套的内容遵照SOAP规范（注1），允许发送方与接收方使用一种与语言无关的方法来交换信息：举例来说，发送方可以使用Python编写，而接收方可以使用Java或C#编写。因为双方在如何解释封套上是一致的，所以都不用去关心对方是如何实现的。在本章中，我们将深入讨论SOAP封套。

HTTP 绑定

SOAP规范要求SOAP实现支持使用HTTP来传输SOAP XML有效负载，因此，目前所有的SOAP实现都支持HTTP，这并不是巧合。当然，各种实现也可以自由地支持任何其他的传输方式，即使规范中没有描述它们。使用什么方式来传输SOAP有效负载都无关紧要，你可以使用SMTP、FTP、JMS，甚至是专用的方案；

注1：　该规范可以在 *http://www.w3.org/TR/SOAP* 处找到。SOAP 1.1 规范不是 W3C 标准，不过现在正在开发中的 SOAP 1.2 规范会是。

事实上，传输的替代方案时常被讨论，而且有一些已经被实现。然而，由于到目前为止HTTP还是最普遍的SOAP传输方式，因此我们将把精力集中在它上面。一旦你掌握了如何将SOAP绑定到HTTP，你就可以很容易地理解其他的传输机制。

SOAP当然可以用于像RPC这样的请求/响应消息交换，同时它还能用于固有的单向消息交换，如SMTP。到现在为止，大多数基于Java的SOAP实现都能够实现RPC形式的消息，因此我们会将主要时间花在这方面；HTTP在RPC形式的消息交换上是一个天才，因为它允许请求和响应作为单一事务过程中的整体部分发生。然而，不应该忽视单向消息传递，而且HTTP也不排斥这样的消息交换。我们将在第八章中介绍单向消息传递。

HTTP请求

我们将看到的第一个SOAP消息是一个RPC请求。虽然它相当简单，但是包含了使用HTTP传输方式的完整SOAP消息所必需的所有元素。这条消息的XML有效负载被包含在一个HTTP POST请求中。下面的消息请求服务器以摄氏温度返回服务器所在地的当前温度。将下面的代码快速浏览一遍，但是现在不必去关注太多的细节：

```
POST /LocalWeather HTTP/1.0
Host: www.mindstrm.com
Content-Type: text/xml; charset="utf-8"
Content-Length: 328
SOAPAction: "WeatherStation"

<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
    <m:GetCurrentTemperature xmlns:m="WeatherStation">
         <m:scale>Celsius</m:scale>
    </m:GetCurrentTemperature>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

这个 SOAP HTTP 请求使用了 HTTP POST 方法。虽然 SOAP 有效负载可以使用其他一些HTTP方法来传递，例如HTTP GET方法，但SOAP规范中定义的HTTP绑定要求使用 POST 方法。POST 方法还指定了正在访问的服务的名称。在这个例子的头中，我们指定将数据传给某个主机上的名为 /LocalWeather的服务，主机的位置在后面说明。这告诉服务器该如何在自己的处理空间中路由该请求。最后，我们的例子指明我们使用了 HTTP 1.0 版，虽然 SOAP 对 HTTP 的版本没有特别的要求。

Host:头字段指定了请求要发送到的服务器的地址，即*www.mindstrm.com*。下一个头字段Content-Type:告诉服务器我们正在使用text/xml媒体类型发送数据。所有的 SOAP 消息都必须使用 text/xml 来发送。本例中的内容类型（content type）还指定数据需要使用 UTF-8 字符集来编码。SOAP 标准对编码方式不做特别要求。Content-Length:告诉服务器接下来将要被 POST 的 SOAP XML 有效负载中的字符数。

到这里为止，所有的头都是适用于任何 HTTP POST 消息的标准 HTTP 头。然而接下来的这个则是SOAP特有的。SOAPAction:头字段对于使用HTTP传输的所有SOAP请求消息都是必需的（注2）。它以URI的形式告诉HTTP服务器一些信息以说明消息的意图。这些信息没有被包含到消息本身中，而是被包含在一个HTTP头字段里，因为它不要求系统首先处理XML有效负载。这意味着，如果服务器上没有能够处理请求的信息或资源，它将可以不去解析这个消息体。尽管这个字段可以包含任何格式的数据甚至可以为空，但这个字段本身必须存在。头SOAPAction: "WeatherStation"可以指明我们的请求需要一个与位于服务器所在建筑的屋顶上的气象台之间的实际连接。如果服务器知道气象台已从建筑上摔下来并被一辆过路的汽车压坏了，它就不会再费事去处理SOAP有效负载。这种事情通常不太可能发生，但是重要的是服务器能通过在SOAPAction:字段中指定的URI得知消息本身的意图，并做出相应的反应。URI并不要求是某种特定的形式，这一点也很重要。它可以是一个URL、一个名称、一个单词，甚至是一个数字，只要它对于接收消息的服务器来说有意义就行。

注 2: 在 SOAP 1.1 规范中是必需的，但预计在 SOAP 1.2 中会是可选的。

如果 SOAPAction:字段包含一个空的字符串（""），那么消息的意图实际上就由HTTP请求的URI（在本例中是/Local Weather）提供。服务器可以将这个URI解释为要访问气象台，或者其他的什么含义。如果这个字段中不包含任何数据，那么该消息中就没有关于后面消息意义和意图的任何信息了。在这种情况下，我们就要期望服务器来继续处理XML有效负载。

HTTP 响应

一个RPC形式的请求消息通常导致一个相应的HTTP响应。当然，如果服务器不能正确处理HTTP头中的信息，它会返回一个某种类型的HTTP错误。但是假定头被正确处理了，系统将返回一个预期的SOAP响应。下面是对前面的RPC形式请求例子的HTTP 响应:

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: 359

<SOAP-ENV:Envelope
     xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
     SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Body>
     <m:GetCurrentTemperatureResponse xmlns:m="WeatherStation">
           <m:temperature>26.6</m:temperature>
     </m:GetCurrentTemperatureResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

大致上，响应的HTTP头字段和请求的头字段差不多。报头第一行中的响应代码200表示服务器能够处理该SOAP XML有效负载。Content-Type:和Content-Length:字段与在前面请求消息中的含义相同。此外不再需要更多其他的头字段；HTTP POST方式生来就是一个请求/响应机制，它暗含了请求和响应之间的相关性。如果你发送一个请求，那么响应就是同一个事务处理过程的一部分。

让我们将最初的请求消息中的单位改成下面这样:

```
<m:scale>Calcium</m:scale>
```

你知道的，对吗？是的，没有Calcium这样一个用于表示温度的单位；我们构造的请求是错误的。接着，我们把这个SOAP请求发送给服务器。假如气象台不是真的从屋顶上掉下来了，服务器就会处理这个请求。就像我们预期的一样，我们的SOAP处理程序不能理解Calcium这个温度单位，它产生的响应将是下面这个错误信息：

```
HTTP/1.0 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: 525

<SOAP-ENV:Envelope
          xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   <SOAP-ENV:Body>
      <SOAP-ENV:Fault>
         <faultcode>SOAP-ENV:Client</faultcode>
         <faultstring>Client Error</faultstring>
         <detail>
            <m:weatherfaultdetails xmlns:m='WeatherStation">
               <message>No such temperature scale: Calcium</message>
               <errorcode>1234</errorcode>
            </m:weatherfaultdetails>
         </detail>
      </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

HTTP错误码500“Internal Server Error（内部服务器错误）”说明出现了错误。和前面一样，HTTP头包括了Content-Type:和Content-Length:两个字段。SOAP规范中说如果消息被收到而且是可理解的，响应发送时就应使用一个2xx状态码。当服务器不能理解消息，或者消息使用的格式不正确，丢失了某些信息，又或者由于其他原因不能够被正确处理时，服务器就必须使用HTTP错误码500（“Internal Server Error”）。HTTP头后跟一个SOAP封套，包括它的自己错误码、错误字符串和错误的详细描述。

我并不确信这是处理SOAP错误的最好方法，也不能肯定使用一个传输协议的错误来包装SOAP错误就是一个好方法。传输错误和SOAP错误之间彼此并不真正相关。毕竟，响应中还包括了一个适当的SOAP消息用来详细地描述错误。然而，公平地说，对于SOAP规范的开发者，这种错误处理程序与HTTP传递错误的方

式一致，我指的是HTML文档。我还可以指出两者的问题，但是这是规定，而且它并不妨碍对SOAP的处理。

HTTP绑定概述

现在我们要来讲述SOAP的HTTP绑定。它的好处在于，确实，HTTP是一种已确立的机制，它对于传输XML这种工作极为胜任，因此它是一个用来传输SOAP消息的好方法。

前面例子中的XML的一些细节对于你来说可能是非常简单易懂的，尽管我们还没有讲到它们。如果你了解XML，那太棒了。如果不是，那也不用害怕。讨论这些的目的是为了使你了解对于SOAP的HTTP绑定是什么样的。在下一节中我们将详细介绍SOAP封套，它是包含SOAP请求和响应的XML有效负载。

SOAP封套

SOAP封套整体上使用了XML来表示SOAP请求或响应（注3）。Envelope元素在消息体中是最上层的XML元素，它必须存在，这样消息才会被认为有效。因此Envelope基本上就代表了这个包含SOAP消息的XML文档。Envelope中还可以有一个可选的Header元素，如果使用它，它就必须是Envelope元素的第一个子元素。Envelope中一定要包含一个Body元素（注4）。如果Envelope中有一个Header元素，则Body元素要紧跟在Header之后，否则，Body元素就必须是Envelope的第一个子元素。

SOAP封套使用HTTP来包装和传输，就像使用邮政服务的纸信封。SOAP封套就相当于纸信封；SOAP头（如果存在的话）和SOAP体相当于信封里的内容；而HTTP头就相当于信封上的实际通讯地址。

注3：在携带了附件时会有一个例外的情况出现。我们将在第八章讨论这种可能性。

注4：目前已经出现了一些关于在SOAP 1.2里提供无Body元素封套的讨论。

我们可以来做进一步的类比。一个纸信封上的回复地址在SOAP程序中没有直接对应的部分；然而，在这两种情况下一个无法递送的邮件都会退回给寄件人。惟一不能对应的事物是邮票。我猜测这是这个类比有点不恰当的地方。但是，尽管也许不必为向服务器发送一个HTTP请求而付费，但是你需要对Internet服务付费。因此这就是你的邮票!

名称空间

理解XML的名称空间对于理解SOAP来说是很重要的。我不准备去详细解释XML名称空间的所有细节，这样做不仅与本书的主题没有很大的直接关系，而且会花费大量的时间。然而，我会对XML名称空间概念中与SOAP相关的部分做一个简要介绍。基本上，名称空间是用来排除XML元素与属性之间歧义性的一种XML机制。换句话说，它们帮助我们了解元素的上下文或意义。我们来看看前面那个例子的这一部分:

```
<m:GetCurrentTemperature xmlns:m="WeatherStation">
        <m:scale>Celsius</m:scale>
</m:GetCurrentTemperature>
```

XML允许我们用与程序设计语言中的变量类似的东西来表示名称空间。在上面的例子中我们建立了一个名称空间标识符并命名为m，它通过“xmlns:m="WeatherStation"”声明。通常，在SOAP消息中声明名称空间的典型语法是:

```
xmlns:id="some URI"
```

关键字xmlns告诉XML处理器我们在声明名称空间。如果定义中还包括一个限定符名称，那么我们就要同时创建一个能用来把此名称空间与其他名称空间区别开来的名称空间限定符。如果没有指定限定符名称，那么我们将定义默认的名称空间。id是我们正在声明的标识符的名称，在这里是m。最后一个部分是一个提供名称空间内容的URI。虽然这个URI通常是一个URL，但重要的是要认识到它仅仅是一个标识符；文档的接收者不会试图去下载该URI的内容（它可能根本就不存在）。我们可以使用字母m来限定与声明本身在同一个作用域内的元素。

限定XML元素意味着将一个特定的名称空间关联到该元素。在本例中，通过声明元素为m:GetCurrentTemperature，我们用名称空间WeatherStation限定了GetCurrentTemperature这个XML元素。这意味着GetCurrentTemperature与用标识符m所表示的WeatherStation名称空间相关联。

要注意名称空间标识符m能用来限定它在其中声明的元素。但是即便使用m来声明，该元素也不会有什么特别。其实我们并不需要仅仅因为该元素将这个名称空间声明作为它的一个属性而使用名称空间标识符m来限定GetCurrentTemperature元素，我们这样做是因为我们想要指明GetCurrentTemperature被定义到哪个名称空间。把这个声明放在这个元素中为标识符m创建了作用域。很明显，标识符在与它的声明处于同一层的地方使用才有效；就像我们例子中所使用的那样。因此，名称空间的作用域是限定在包含它声明的元素中的。这说明名称空间ID可以用于包含它的声明的元素、该元素的任何属性、任何子元素以及包含元素的相关属性中。某个名称空间标识符在高于该元素层次的XML元素中是不适用的。如果你返回去再看看前面的例子，这句话意味着名称空间标识符m不能在Body元素的属性中使用，因为这超出了m的作用域。这条作用域的规则与面向块（block-oriented）的编程语言（比如Java）使用的规则很相似，在那里当你进入某个括起来的代码块时名称空间将被压入堆栈，而当你从该块退出时，它将被弹出。下面的Java程序片段说明了变量var1和var2的同种类型的作用域：

```
int var1 = 10;
{
    int var2 = 2 * var1; //这是正确的，因为var1处于作用域内
}
var1 = var2; //这是无效的，因为var2超出了它的作用域
```

注意： XML支持对默认名称空间（default namespace）的使用，这将导致在没有显式指明名称空间的情况下名称空间的限定。如果不使用一个关联的名称空间标识符，默认的名称空间将通过为名为xmlns的属性赋值来声明。看看下面这个例子：

```
<GetCurrentTemperature xmlns="WeatherStation">
    <scale>Celsius</scale>
</GetCurrentTemperature>
```

在这个例子里，GetCurrentTemperature元素和scale元素都与WeatherStation名称空间相关联。

名称空间限定在需要判定某元素或某属性的真实含义时常常是必需的。考虑一下这个例子:

```
<truckmonitor>
    <scale>37F6A</scale>
    <weight>12000</weight>
    <scale>Celcius</scale>
    <temperature>25</temperature>
</truckmonitor>
```

这个 XML 片段中包含了一些在一个卡车监测站收集的数据。第一个 scale元素指定卡车称重时使用的单位。稍后，我们还有一个相同的元素名称 scale，用于描述汽车监测站所在地外部温度的单位。这样我们如何知道两个scale元素的意义呢？你可能觉得通过看看全部的结构仅仅凭着直觉就能区分开，但是这样做并不总是有效的。

当然，要避免这个问题的一个方法是使用更具有描述性的名称，或者将 XML 的结构设计得能更好地表示元素的含义。但是你不一定总能将这些事情掌握在控制之中；你也许正在根据来自不同地方的多个 XML 片段创建一个组合的文档。适当使用名称空间限定将会对解决名称冲突和歧义的问题有帮助。下面的文档与前面的那个是一样的，只是对两个 scale 元素使用了适当的限定:

```
<truckmonitor xmlns:ns1="TruckScale" xmlns:ns2="Thermometer">
    <ns1:scale>37F6A</ns1:scale>
    <weight>12000</weight>
    <ns2:scale>Celcius</ns2:scale>
    <temperature>25</temperature>
</truckmonitor>
```

第一个 scale现在与名称空间 TruckScale相关联，而第二个 scale与名称空间 Thermometer相关联。你注意到了两个名称空间是在同一个元素的属性里声明的吗？既然名称空间的声明无非是一个属性而已，而一个 XML 元素又可以有不止一个属性，这样做就完全正确。

SOAP定义了两个需要在SOAP消息里使用的名称空间。SOAP Envelope由名称空间: http://schemas.xmlsoap.org/soap/envelope限定。我们在前面的例子中是通过声明 SOAP-ENV 名称空间标识符来使用这个名称空间，并用它来限定

Envelope元素的。名称空间标识符http://schemas.xmlsoap.org/soap/encoding用于encodingStyle属性的声明，我们将在后面讨论它。要注意encodingStyle属性是使用SOAP-ENV标识符作为名称空间限定的。我们再浏览一遍它：

```
<SOAP-ENV:Envelope
          xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
          SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
         . . .
         . . .
</SOAP-ENV:Envelope>
```

Envelope 元素

Envelope是用来表示SOAP消息的XML文档中的最上层的元素。Envelope是最高层次的惟一的XML元素；SOAP消息中其余的部分都在Envelope元素里，被编码为子元素。一个SOAP消息必定有一个Envelope元素。如前面的例子所示，Envelope元素中可以有名称空间声明，由http://schemas.xmlsoap.org/soap/envelope名称空间限定。这就是为什么该元素的名称被写成SOAP-ENV:Envelope的原因。Envelope元素中通常还会声明encodingStyle属性，该属性使用声明过的名称空间标识符SOAP-ENV来限定名称空间。

Envelope的所有子元素和属性本身必须由名称空间限定。与Envelope一样，这些元素和属性也是被SOAP-ENV限定的。对于本章后面的部分和本书的其余部分，我们将使用名称空间标识符SOAP-ENV来代表http://schemas.xmlsoap.org/soap/envelope名称空间。这将使本书更容易阅读。要记住用于限定符的是名称空间本身而不是它的名称。

Header 元素

SOAP头是可选的。如果它存在，就必须以Header命名并且一定要是Envelope元素的第一个子元素。Header元素也是由SOAP-ENV标识符来限定名称空间的。

通常，Header项用来对消息中用于事务处理、鉴别或其他与处理或路由信息有关的那些元素进行编码。它是很有用的，因为，正如我们看到的，Body元素只用来对表示RPC（或其他）有效负载的信息进行编码。Header是一个扩展机制，它允许在Body元素的消息语义之外有任何类型的信息存在，而这些信息对正确处理消息是有用的甚至是必需的。

对于Header元素的直接子元素，应该限制属性的使用。规范中说应该这样做，意味着尽管可以在Header元素的深层子元素中使用属性，但对于直接子元素，规范要求接收者忽略它的属性。

这里是一个SOAP头的例子，它包含一个名为username的直接子元素。我们没有在这个元素中使用任何属性，只是对它的名称空间进行了限定。username元素标识了发出请求的用户。

```
<SOAP-ENV:Header>
    <ns1:username xmlns:ns1="JavaSoapBook">
      Jessica
    </ns1:username>
</SOAP-ENV:Header>
```

actor 属性

一个SOAP消息在被处理之前常常会经过一个或多个中间环节。举例来说，在客户应用程序和目标SOAP服务之间可能存在一个SOAP代理服务。我们将在第十章中看到一个有趣的例子，在那里我们将开发一个SOAP应用程序服务来代理另一个服务，后者用来代表客户应用程序，它实际上只是指明了最终服务的地址。因此，头可能既包含供最终目的地使用的信息，也包含仅供中间环节使用的信息。actor属性（隐式地或是显式地）指明了Header元素的接收者。

了解SOAP通过中间环节的必要条件是很重要的。从本质上说，要求任何供中间环节使用的SOAP Header元素都不被传递到路径中的下一个SOAP处理器。Header元素中描述了发送者和接收者之间的契约。然而，如果一个Header元素中包含的信息既被中间环节需要又被一个下游的服务器所需要，那么中间环节就

会在消息中插入适当的Header，然后再送往下游。事实上，如果中间环节认为需要，它可以自由地添加任何 Header 元素。

如果一个 Header 元素中没有 actor 属性，传输时就会假定该元素只供最终接收者使用。这样做等价于包含 actor 属性并将它的值设为最终目的地的 URI 地址。

我们来在前面的例子中增加一个 actor 属性。假设消息将发往一个位于 *http://www.mindstrm.com/AppServer*的中间应用程序服务器。我们希望那个应用程序服务器能记录发送请求的用户的名称，然后将该请求转送到最终的目的服务器。为此，我们把 username的 actor 属性设为 *http://www.mindstrm.com/AppServer*:

```
<SOAP-ENV:Header>
   <ns1:username
         xmlns:ns1="JavaSoapBook"
         SOAP-ENV:actor="http://www.mindstrm.com/AppServer">
         Jessica
    </ns1:username>
</SOAP-ENV:Header>
```

代理应用程序服务器看到 username元素是供自己使用的而且不必转发到最终目的地，它会适当地使用这些数据。它将从 Header 中删除 username元素，然后再将消息转发给最终目的地。正如你所看到的，actor属性是用 SOAP-ENV标识符限定的。这是因为 actor 属性是用 SOAP 定义的，并且使用了相应的名称空间 http://schemas.xmlsoap.org/soap/envelope 来指定。

mustUnderstand 属性

SOAP 中包含了可选的和强制性的头元素的概念。这并不意味着被包含的元素都是强制性的 ——那是应用程序的事情。“强制性的”意味着接收者必须理解并且正确使用由Header元素所提供的信息。这个要求允许我们应付这样的情况：如果SOAP消息的接收者不知道如何处理Header元素中提供的数据，那么它将不能完成工作。在这种情况下，可以使用 mustUnderstand 属性并将它的值设为 1。

比方说，这也许在发送应用程序升级到一个新版本的情况下是有用的。新的版本可能使用了一些新的信息，而这些信息必须被服务器处理以保证返回结果的有效性。当然你会期望服务器也进行相应的升级，但是也许那还没有发生。因为版本失配，服务器的旧版本不了解它从升级后的客户应用程序收到的新的SOAP头元素。举例来说，头元素 username 必须被接收方理解；如果不是，该消息应该被拒绝。我们可以通过在 SOAP 消息中将 mustUnderstand 属性的值设为 1 来达到这个要求（值0实质上等价于不提供该属性）。我们来修改一下前面的例子，指定接收者必须理解 username 元素：

```
<SOAP-ENV:Header>
    <ns1:username
        xmlns:ns1="JavaSoapBook"
        SOAP-ENV:actor="http://www.mindstrm.com/AppServer"
        SOAP-ENV:mustUnderstand="1">
        Jessica
    </ns1:username>
</SOAP-ENV:Header>
```

如果这个消息的接收者不理解username元素，它将用一个SOAP错误来响应。该响应可能看起来和下面差不多：

```
HTTP/1.0 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: 373

<SOAP-ENV:Envelope
          xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    <SOAP-ENV:Body>
       <SOAP-ENV:Fault>
          <faultcode>SOAP-ENV:MustUnderstand</faultcode>
          <faultstring>SOAP Must Understand Error</faultstring>
          <faultactor>http://www.mindstrm.com/AppServer</faultactor>
       </SOAP-ENV:Fault>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

该错误响应包括一个 faultcode、一个 faultstring和一个 faultactor; faultactor 指明错误发生在何处。我们将在本章后面的部分中进一步讨论错误。

encodingStyle 属性

encodingStyle属性指定用于串行化和反串行化数据元素的数据编码规则。SOAP并没有为数据串行化指定任何默认的规则，了解这一点是很重要的。然而，SOAP确实指定了SOAP实现一般都要支持的一个定义数据类型的简单方案。这是下一章的主题，因此我们在这里将不深入讨论编码规则的细节。不过，重要的是要知道如何在SOAP消息中指定用于串行化和反串行化数据元素的编码形式。

encodingStyle属性是使用SOAP-ENV名称空间标识符限定的。在下面的例子中，我们指定encodingStyle属性为Envelope元素的一部分；我们将在以后各章的例子中把这个声明改到Body元素中去。两种方式都可以，你只要知道encodingStyle属性适用于它的声明所在的那个元素以及该元素的所有子元素（也就是说，在它的作用域内）就可以了。

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    . . .
    . . .
</SOAP-ENV:Envelope>
```

可以在声明中使用多个URI。在这种情况下，第一个URI必须是最有效力的，而最后一个是最没有效力的。这里有一个例子：

```
SOAP-ENV:encodingStyle="http://www.mindstrm.com/tightEncoding
http://www.mindstrm.com/looseEncoding"
```

在这种情况下，系统将首先使用*http://www.mindstrm.com/tightEncoding*来查找编码规则（注5）。如果使用这个URI找不到编码规则，系统就会尝试*http://www.mindstrm.com/looseEncoding*。可以通过给URI指定一个空字符串值（""）来关闭当前作用域的编码形式。这个声明适用于当前元素和它的所有子元素。

注5：看起来好像系统尝试下载这个tightEncoding，特别是在这个URI碰巧是一个URL的时候。但实际情况并不是如此；系统会仅仅在它自己的表中查找看是否能理解tightEncoding，并做出相应的动作。

封套版本

SOAP规范没有定义一种计数系统来说明你正在使用SOAP封套的哪个版本。在SOAP 1.1中，所有的封套都必须与http://schemas.xmlsoap.org/soap/envelope名称空间相关联（注6），这在我们前面的所有例子中都用过。就是这么回事。目前还没有其他信息可用来为SOAP封套定版本。如果一个SOAP消息与任何其他的名称空间相关联，或如果没有声明名称空间，那么接收者将不得不返回一个SOAP版本失配的响应。下面是这种情况下的一个SOAP错误响应的例子：

```
HTTP/1.0 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: 311

<SOAP-ENV:Envelope
          xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   <SOAP-ENV:Body>
      <SOAP-ENV:Fault>
         <faultcode>SOAP-ENV:VersionMismatch</faultcode>
         <faultstring>SOAP Envelope Version Mismatch</faultstring>
      </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Body 元素

SOAP Body元素在SOAP 1.1中是强制性的。如果没有SOAP头，Body必须是Envelope中的第一个子元素；否则它必须直接跟着SOAP Header元素。Body元素也是由SOAP-ENV名称空间标识符限定的。

Body元素中包含了SOAP请求或响应。在这里你可以找一个包含方法名称及其参数的RPC形式的消息，或者是一个单向消息和它的相关部分，或者是一个错误和它的详细信息。SOAP Body元素没有被SOAP规范完全定义；我们稍后将详细地讨论这个主题。事实上，SOAP只定义了一种类型的Body：SOAP Fault。

注6： 对于SOAP 1.2，这个名称空间是http://www.w3.org/2001/12/soap-envelope。

让我们再回顾一下前面那个包含 Body 元素的封套的例子。在那种情况下，请求是一个 RPC。和其他由 SOAP 定义的元素和属性一样，我们也使用了 SOAP-ENV 名称空间标识符来限定这个 Body 元素。

```
<SOAP-ENV:Envelope
          xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
          SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
         <m:GetCurrentTemperature xmlns:m="WeatherStation">
                <m:scale>Celsius</m:scale>
         </m:GetCurrentTemperature>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP Fault

Fault是SOAP规范定义的惟一一个Body元素项。它用来向SOAP消息的发送方传回错误信息。Fault元素必须作为Body元素的直接子元素出现，而且它的出现不能超过一次。

SOAP 定义了 Fault 元素的四个子元素。不是所有的子元素都是必需的，有些只是在某些情况下才需要。这些元素将在下面各节中介绍。你可能会回想起我们在前面的例子中已经看到过所有这些元素，因此在看过这里的介绍之后你可能想返回去再看看那些例子。

faultcode 元素

faultcode元素提供了一个可以被软件处理过程识别的错误指示，它为接收这个错误的系统提供了一个适当处理错误的机会。这个代码对人来说没什么用处，那是下一节将介绍的faultstring元素的目的。faultcode元素是强制性的，而且必须作为Fault元素的子元素出现。SOAP定义了许多错误码供faultcode元素使用。这些代码与http://schemas.xmlsoap.org/soap/envelope名称空间相关联。下面是SOAP 定义的错误码的简短描述：

VersionMismatch

说明与SOAP Envelope相关联的名称空间无效。

MustUnderstand

说明有一个SOAP Header元素使用了值为1的mustUnderstand属性，但有某个属性不可理解，或者是与某个属性关联的语义不可理解。

Client

说明格式化SOAP消息出错或者是该消息没有包含适当的或者必要的信息。客户在没有对该消息做适当更改的情况下不应该再次发送该消息。在使用错误的温度单位来请求得到当地温度的那个例子中我们已经看到了这种情况。

Server

说明由于某种与接收消息的格式化和内容无关的原因，消息不能处理。我们可以认为这意味着消息在稍晚的时候不做修改就可以再次被发送。举例来说，完成请求动作所需的后端数据库服务器可能当前不在线，但可能稍后会在线。

这些错误码是可扩展的；这意味着它们能通过使用句点符号来扩展，在句点符号后跟随的名称将提供更特定的细节。例如：

```
Server.WeatherStationFailure
```

这仍会被当成一个Server错误，但它提供了更多的信息：用来完成这个请求的气象台的设备有点问题。我们可以进一步把该错误码扩展成下面这样：

```
Server.WeatherStationFailure.SmashedToPieces
```

这个错误码让我们知道气象台设备需要修理了，可能是因为从建筑物上摔落之后又被轧坏了。然而，请记住，这些代码是提供给软件处理的。扩展的错误码在预先知道可能发生的问题的情况下才被使用，它允许错误接收者采取相应的行动（我不敢肯定设备被打碎是否有资格被当成一个预知的条件）。

faultstring 元素

这个元素也是强制性的，而且必须以Fault元素的子元素的形式出现。它的目的是提供一个对人来说可读的错误描述；它被接收者处理的方式没有被设计成像faultcode元素那样。发送者将会在这个元素中放入一个合理的描述，读取者应该可以结合原始请求的上下文弄懂它的意思。

faultactor 元素

这个元素仅在某些情况下是必需的，如果存在，它也必须作为Fault的子元素出现。它是前面介绍的头属性actor的必然结果。它说明错误出现在哪个系统上。在前面，我们谈到了SOAP中间环节并使用一个代理应用服务器作为例子。在那里，代理因为它不知道如何处理username元素而产生错误。因为应用程序代理不是原始消息的最终接收者，因此它需要使用faultactor元素，以指明它自己是错误的来源。这个元素的值就是错误产生者的URI。

如果错误的来源是消息的最终目的地，Fault元素就不必包含一个faultactor元素。不过，在这种场合下也可以包含该元素。

detail 元素

如果请求消息中与Body元素有关的部分出现错误，detail元素就提供与之有关的信息。如果Body的内容(或内容有缺损)导致消息不能被适当处理，那么detail元素必须以Fault的子元素的形式出现。另一方面，如果错误与消息中的Body元素无关，就不能包含detail元素；SOAP规定如果detail元素不存在就说明错误与对Body元素的处理无关。

特别地，detail元素不能够用来进一步描述与SOAP Header元素有关的错误。在前面的例子中，因为问题与Body的内容无关，代理应用服务器就不能够使用detail元素来进一步描述错误。

另一个错误的例子

这是另一个完整的SOAP错误的例子:

```
HTTP/1.0 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: 595

<SOAP-ENV:Envelope
          xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   <SOAP-ENV:Body>
      <SOAP-ENV:Fault>
         <faultcode>SOAP-ENV:Client</faultcode>
         <faultstring>Client Error</faultstring>
         <faultactor>http://www.mindstrm.com/LocalWeather</faultactor>
         <detail>
            <m:weatherfaultdetails xmlns:m="WeatherStation">
               <message>No such temperature scale: Calcium</message>
               <errorcode>1234</errorcode>
            </m:weatherfaultdetails>
         </detail>
      </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

这里使用了与本章前面那个例子中相同的一个响应。惟一的不同是这次我们包含了faultactor元素来识别错误的来源。所以这个例子包含了每一个SOAP错误码的实例，其中detail元素对错误做了进一步描述。

到这里，我们已经介绍完了SOAP消息本身。接下来我们要看看在消息中使用的编码方式。

第三章

SOAP 数据的编码

当通过网络传输数据时，其中的数据一定要遵从底层的传输协议，而且还要采用让发送方与接收方都能够理解其含义的方式来格式化。这就是我们所提到的数据编码（data encoding）。数据的编码包含对数据结构、数据传送类型，当然还有数据值的组织。正如在Java中，被串行化的是数据，而不是行为。数据编码和串行化规则帮助SOAP事务中的各方理解消息的内容和含义。SOAP编码的模型是以XML的数据编码为基础的，但是SOAP编码约束或者改变了那些规则以适合SOAP的目的。我想你会发现使用这里介绍的编码规则将能很容易地表示绝大多数系统的数据需求。

模式与名称空间

名称空间提供了用于决定如何解释元素和属性名称的机制。因为XML允许任意的元素名称，所以它需要一种机制来指明对于给定的名称应该使用哪个字典来查找它的含义。SOAP规范的第5节中定义的编码形式在SOAP中使用得最普遍。这种编码形式，在由http://schemas.xmlsoap.org/encoding名称空间引用的模式中定义，它常被称为“SOAP第5节”。在SOAP消息中，按照惯例这个名称空间使用如SOAP-ENC或soapenc这样的名称空间限定符来引用。在我们的例子中，将使用SOAP-ENC名称空间限定符来表示这个名称空间。

在SOAP中，模式被用作数据元素定义的引用，了解这一点是很重要的。它们不被用于在标准SOAP处理过程中验证SOAP消息数据的有效性，不过你要这样做的话也没什么能够阻止你。对模式的引用常被当作名称空间使用以限定一个已串行化的数据元素。理解数据的结构和含义并据此编码，这是开发者和底层框架的事。

SOAP第5节中合并了“XML Schema第二部分：数据类型”（注1）中的所有内置数据类型。这个模式定义了你将用到的绝大部分基本数据类型，你可以直接使用或者是作为你自己类型的一部分使用。一般的用法是，声明一个名为xsd的名称空间标识符并将它与名称空间http://www.w3.org/2001/XMLSchema相关联。正如我们在上一章中所看到的，这个声明通常被作为SOAP Envelope或SOAP Body的一个属性。

另一个用于数据编码的通用名称空间标识符是xsi，它与名称空间http://www.w3.org/2001/XMLSchema/instance相关联。说明这个标识符最简单的办法是使用一个例子。假设我们想要将某个特定的XML元素的数据类型声明为XML Schema 2001中定义的浮点（float）类型。我们通过使用xsi的type属性来实现，type属性是一个实例数据类型。xsi:type属性指定了编码数据值的数据类型。这里是使用它的一个例子：

```
<xyz xsi:type="xsd:float">3.14159</xyz>
```

注意： Schema 2001在这里不是惟一能使用的。Schema 1999和Schema 2000也很常用，而且其他的还将出现。SOAP实现应该能够处理任何一种可能性，因为消息中会包含对适当模式的引用。

下面的名称空间标识符在SOAP文献和实现中被普遍使用。让我们看一看它们的声明以便了解它们：

```
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/encoding/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema/instance/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema/"
```

注1： 可以在*http://www.w3.org/TR/xmlschema-2/*处找到这个规范。

术语

为了完全了解SOAP编码，有必要介绍一些术语。这些术语在SOAP规范中描述，但我们将在此基础上做进一步说明，并给出一些使用Java和XML的例子。

术语“值（value）”用来描述XML元素中被编码的实际数据。这个数据是一连串的字符。这个字符串的解释刚好和你在通常的Java编程中使用的那些类型有关：一个数字、一个名称、一个日期或更复杂的东西。一个SOAP消息中的值总是与给定的类型相关联。我们稍后会讨论类型；现在我们只要说在SOAP中一个给定值的数据类型从来不会是未定义的就行了。在消息中总会有方法来确定一个值的正确数据类型。

SOAP对简单值与复合值做出了区分。一个简单（simple）值不包含任何命名的部分；它仅仅包含单一的数据。与编程语言中像字符串、整数和浮点数这样的类型相关联的数据值可以视为简单值。下面是一些在Java中的应用以及对应的SOAP例子：

```
int a = 10;                        <a xsi:type="xsd:int">10</a>
float x = 3.14159;                 <x xsi:type="xsd:float">3.14159</x>
java.lang.String s = "SOAP";       <s xsi:type="xsd:string">SOAP</s>
```

上面的每一个变量都包含了关联类型的简单值。数据没有被分成多个部分，而且得到该数据不需要其他名称或机制。

一个复合（compound）值包含多个彼此相关的数据。要访问该数据中的单个数据可能需要先在一系列有序的值中指定一个位置值，就像传统的数组那样。它们可以使用作为关联数组的键的值来访问，比如一个散列表。它们也可以使用构成部分的名称来访问，就像C语言里的struct。无论使用什么机制，总有办法来从一个复合值里把一个特定的简单值分离出来，这种机制称为存取器（accessor）。复合值的某个构成部分其本身也可能是一个复合值。下面是在Java和SOAP中使用复合值的一些例子：

```
int[3] iArray = {10, 20, 30};

<iArray xsi:type=SOAP-ENC:Array SOAP-ENC:arrayType="xsd:int[3]">
    <val>10</val>
```

```
    <val>20</val>
    <val>30</val>
</iArray>

class Sample
{
     public int iVal = 10;
     public java.lang.String sVal = "Ten";
}
Sample samp = new Sample();


<Sample>
    <iVal xsi:type="xsd:int">10</iVal>
    <sVal xsi:type="xsd:string">Ten</sVal>
</Sample>
```

在这个例子中，在获得Java数组变量iArray中的值时要用到顺序存取器。为了访问Java变量samp，我们可以使用命名的存取器iVal和sVal。在“复合数据类型”一节中，我们将会进一步讨论一些复合数据类型，包括数组和结构。

和大多数程序设计语言一样，在SOAP中，值与相应的数据类型相关联。我们在讨论简单数据类型和复合数据类型时已经简略地提到了这个问题；每一个这样的值都有一个相关联的数据类型。就像在Java中一样，我们先声明数据类型，然后为它们分配值。

串行化规则

SOAP消息中的XML元素不是独立的（independent）就是嵌入的（embedded）。由于XML的层次性，绝大多数元素都是作为其他元素的子元素嵌入的。不是其他任何元素子元素的那些元素被称为独立元素；它们出现在串行化过程的最高层。

SOAP消息中的所有值都被编码成某个元素的内容。数据值本身不能在某个元素之外单独出现。不过，这并不意味着每个XML元素中都会包含一个值。举例来说，像结构或数组这样的复合数据类型就包含了含有实际数据值的子元素。而定义这些复合数据值的元素就不直接包含数据。稍后我们将介绍复合数据类型。

引用

SOAP第5节还准许元素引用包含在其他元素中的值。在这种情况下，该元素中就不提供值，而是使用一个属性来指定能够找到实际数据值的元素。数据值必须包含在串行化最上层的独立元素中。

包含数据值的那个元素中必须包含一个名为id的ID类型属性。这个id属性的值就是其他元素引用该数据时要使用的名称。下面就是一个这样的元素:

```
<lastName id="name-1">Englander</lastName>
```

lastName元素自己就是一个完全有效的存取器。此外，它还有一个标识符以允许其他元素引用其数据值。

访问其他元素值的那些元素本身不包含数据值。它们是包含一个herf属性的空元素，该属性用来引用值。href属性是uri-reference类型，该类型在XML Schema规范中定义；它的值引用了包含数据值的那个元素的id属性。下面的例子中就是一个引用前面那个lastName元素值的存取器元素:

```
<surName href="#name-1"/>
```

能够引用其他元素值的这种能力是很重要的，理由有两个。首先，它使缩减SOAP消息的大小成为可能。假设你正在发送一个包含Englander家庭成员名字的消息。这个XML可能是这样的:

```
<member>
    <firstName>Rob</firstName>
    <lastName>Englander</lastName>
</member>
<member>
    <firstName>Jessica</firstName>
    <lastName>Englander</lastName>
</member>
<member>
    <firstName>Arnold</firstName>
    <lastName>Englander</lastName>
</member>
    . . .
    . . .
    . . .
```

正如你看见的，有很多重复的数据。因为许多存取器都可以指向同一个元素，所以引用允许我们将重复减到最少。在这种情况下，该值被称为多重引用（multi-reference）值。让我们使用多重引用来进行另一种尝试：

```
<surName id="Last-Name">Englander</surName>
<member>
    <firstName>Rob</firstName>
    <lastName href="#Last-Name"/>
</member>
<member>
    <firstName>Jessica</firstName>
    <lastName href="#Last-Name"/>
</member>
<member>
    <firstName>Arnold</firstName>
    <lastName href="#Last-Name"/>
</member>
    . . .
    . . .
    . . .
```

在这种情况下，家庭的每个成员都有一个lastName元素，该元素引用了surName中的数据值。当然，在这个不是很恰当的例子里我们并不能真正节省空间。然而，如果数据值非常大的话，该技术将会节省相当大的空间。而且如果被引用的值是一个复合元素的话，这种节省的效果会变得更为显著。

节省空间并不是使用多重引用变量的惟一理由。当很多对象引用同一个对象，在串行化这种对象的图或集合时，该技术同样有用。在反串行化期间需要重建对象的时候，保持它们之间的关系也很重要。我们来看一个Java的例子。在下面的代码中，有一个命名为Employee的类包含了职员的名字和姓氏、他或她的头衔以及该职员的经理（也是一位职员）。下面的代码定义了一个叫Rob的经理和三位分别叫做Ben、Andrew和Lorraine的职员，他们为Rob工作。

```
Class Employee {
    protected java.lang.String _firstName;
    protected java.lang.String _lastName;
    protected java.lang.String _title = "Worker Bee";
    protected Employee _manager;
    public Employee(java.lang.String first, java.lang.String last,
                    Employee mgr) {
```

```
        _firstName = first;
        _lastName = last;
        _manager = mgr;
    }
    public void setTitle(java.lang.String title) {
        _title = title;
    }
    public java.lang.String getTitle() {
        return _title;
    }
    public java.lang.String getFirstName() {
        return _firstName;
    }
    public java.lang.String getLastName() {
        return _lastName;
    }
    public Employee getManager() {
        return _manager;
    }
}
Employee _rob = new Employee("Rob", "Englander", null);
_rob.setTitle("Slave Driver");
Employee _ben = new Employee("Ben", "Jones", _rob);
Employee _andrew = new Employee("Andrew", "Smith", _rob);
Employee _lorraine = new Employee("Lorraine", "White", _rob);
```

图3-1显示了三个“工蜂（Worker Bee）”（Ben、Andrew和Lorraine）和“监工”经理（Rob）之间的关系。显然我们不希望重复变量_rob所代表的对象。那样做不能适当地表示这些对象之间的关系。我们希望每一个“工蜂”职员对象都引用同一个经理对象。如果你还不能明白这种区分为什么很重要，请想像一下Rob得以晋升为“高级监工”的情况。我们当然不希望对每一个职员对象都先调用getManager()，然后再调用setTitle()。我们希望一次就能够完成对_rob对象的变动。

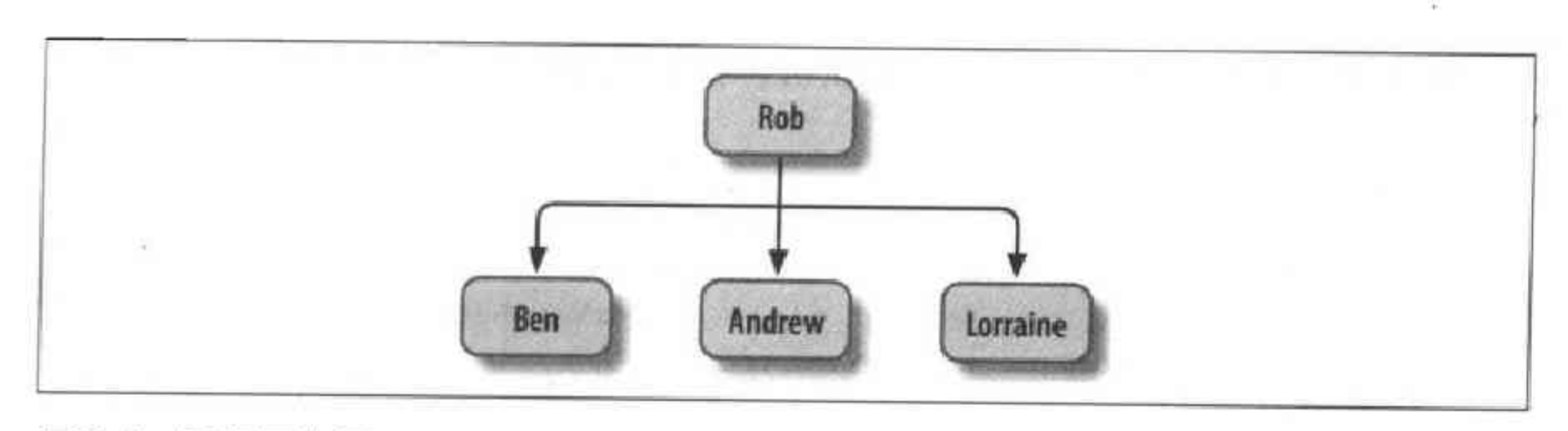

图3-1：职员层次图

在SOAP中，多重引用变量保存对象之间的这些关系，使得对象关系图在SOAP事务处理的两端都能够被适当地表示。为上面例子中这种关系建立的XML如下：

```
<rob id="Rob">
    <firstName>Rob</firstName>
    <lastName>Englander</lastName>
    <title>Slave Driver</title>
</rob>
<ben>
    <firstName>Ben</firstName>
    <lastName>Jones</lastName>
    <title>Worker Bee</title>
    <manager href="#Rob"/>
</ben>
<andrew>
    <firstName>Andrew</firstName>
    <lastName>Smith</lastName>
    <title>Worker Bee</title>
    <manager href="#Rob"/>
</andrew>
<lorraine>
    <firstName>Lorraine</firstName>
    <lastName>White</lastName>
    <title>Worker Bee</title>
    <manager href="#Rob"/>
</lorraine>
```

指定数据类型

在每一个包含数据的元素中必须同时也指定该数据的类型。指定数据类型有几种不同的方法。第一种机制是在元素中包含一个xsi:type属性。这意味着该属性名为type，它由xsi指定的名称空间所限定。分配给xsi:type的值必须是像xsd:float、xsd:string这样的有效类型标识符。在第二种机制中，值可以是数组的一个元素，该数组已经将它的成员限制为特定的类型。在这种情况下，不需要为单个值进行显式的类型声明；这将在以后我们讨论数组时再做介绍。最后，元素名本身可以被关联到某种类型，而该类型可以由相关联的XML模式决定。下面的例子来自一个XML模式，它定义了一个复合数据类型：

```
<element name="Automobile" type="Automobile"/>
<complexType name="Automobile">
```

```
        <element name="make" type="xsd:string"/>
        <element name="model" type="xsd:string"/>
        <element name="year" type="xsd:int"/>
    </complexType>
```

数据类型Automobile中包含了类型为string的make和model元素，以及一个类型为int的year元素。现在让我们来看一个基于这个模式的Automobile类型的实例：

```
<niceCar xsi:type="Automobile" >
    <model>Corvette</model>
<make>Chevrolet</make>
    <year>1999</year>
</niceCar>
```

这个XML定义了Automobile类型的一个实例：1999年雪佛兰产的Corvette型车。因为该模式已经为Automobile的每一个成员指定了数据类型，我们就不需要在例子中声明其数据类型了。

简单数据类型

SOAP使用在“XML Schema第二部分:数据类型”中定义的简单类型，在那一节中称之为“内置的数据类型”。规范的这一部分讨论了整数、浮点数和字符串等内容，以及许多由这些基本类型导出的数据类型。比如，positiveInteger是一个以int类型为基础的简单数据类型，但是它只允许使用正数值。

注意： 不要仅仅因为某个数据类型是内置的，你就断定所有的SOAP实现都会自动支持它。最聪明的做法是先查看你所使用的系统的文档。也许像xsd:int、xsd:string和xsd:float这样的类型比较可靠一些。而且我想你将发现相当多而且相当棒的新类型，但是首先你得去看看文档。

如果看一看XML Schema规范，你会找到很多的简单类型。表3-1中列出了少数几个简单类型及其相关值的例子。

表 3-1：XML Schema 定义的简单类型

类型	例子
int	-41
float	3.14159
string	"java and SOAP"
positiveInteger	100

这里是表 3-1 中的示例数据在 XML 文档中使用时的形式：

```
<iVal xsi:type="xsd:int">-41</iVal>
<fVal xsi:type="xsd:float">3.14159</fVal>
<sVal xsi:type="xsd:string">java and SOAP</sVal>
<pVal xsi:type="xsd:positiveInteger">100</pVal>
```

数据类型用xsi:type属性来显式定义，因此你可以清楚地看到它是什么类型。然而，要记住，这并不是将一个元素与一个类型相关联的惟一办法。

字符串

在字符串元素的声明上你可能会遇到两种不同的方法。第一种是xsd:string，我们在前面的一个例子中用过；第二种是 SOAP-ENC:string，这是一个基于xsd:string的类型，它允许对多重引用值使用 id 和 href 属性。这些字符串类型与像Java这样的编程语言中的字符串类型 java.lang.String不完全相同。对SOAP 中字符串类型的使用有一些限制，用来防止使用某些特殊字符，比如构成XML 标记所使用的那些字符。很明显，如果字符串的值里出现了像“<”和“>”这样的字符，就会干扰 XML 的整体结构。看看这个例子：

```
<value xsi:type="SOAP-ENC:string">embedded < is no good</value>
```

字符串中包含的“<”字符会导致问题，因为这个字符在 XML 中有特别的意义。这并不是说把这样的字符编码到字符串中是不可能的；你只需使用替代的字符或者除SOAP-ENC:string以外的其他类型即可。对括号使用转义序列也是一种解决方案，如 <和>。解决这个问题的另一个办法是使用 MIME base64 编码，因为 base64 字母表中不包括任何被禁止的字符。我们稍后会介绍 base64 编码。

枚举

枚举的值本质上是一组用于表示其他值的名称。在C语言中，enum关键字允许你使用描述性的名称来代替整数值。举例来说，你可以定义一个使用颜色名称来代替值的枚举类型，而不是使用值1、2和3来表示红色、黄色和绿色。在运行时，系统将以相关联的值代替这些名称。由于这使源代码更具有可读性并且更容易理解，所以它或多或少为编程带来了方便。

枚举类型的一个问题是，用于在内部表示它们的值不是标准化的。比如，对于Green这个名称，没有标准化的整数值与它对应。因此将枚举类型值从一个系统传输到另一个系统便存在一个固有的问题。对于应用程序来说，内部使用的实际值并不像看起来那样是惟一的。在C语言中，整数所用的值常默认为从0开始。在这种情况下，枚举类型中名称声明的顺序就决定了值的大小；第一项被定义为0，第二项是1，依次类推。然而，这种语法允许程序员覆盖这种行为而独立地定义每个项目。这样实际值就是对于两个程序来说可能不一致的编程细节。有些程序语言，比如Java，甚至不支持枚举类型（注2）。但是这并不会削弱枚举类型的用处。

SOAP通过对值名称的编码来编码枚举类型。在表示交通信号的情况中，SOAP编码将忽略内在的实际值而支持颜色的名称。这里是想像中的交通信号状态编码的XML片段：

```
<trafficSignal>
    <signalState xsi:type="xsd:string">Green</signalState>
</trafficSignal>
```

发送应用程序不对Green值的内部表示进行编码，而是使用了Green这个名称本身。接收者可以将Green解析为自己觉得适当的内部值。在模式中定义一个枚举类型也是可能的。交通信号模式的定义看起来可能是这样的：

```
<element name="TrafficSignalState" type="TrafficSignalState"/>
<simpleType name="TrafficSignalState" base="xsd:string">
```

注2： 当然，在Java中有一个Enumeration接口，这是一种用于访问集合中元素的机制。但这种功能与C风格的枚举类型无关。

```
        <enumeration value="Red"/>
        <enumeration value="Yellow"/>
        <enumeration value="Green"/>
    </simpleType>
```

使用这个模式定义，我们可以将 trafficSignal 值编码成：

```
<trafficSignal>
    <TrafficSignalState>Green</TrafficSignalState>
</trafficSignal>
```

二进制编码

对于 SOAP 规范中称为字节数组（byte array）的二进制数据，SOAP 推荐使用 MIME base64 编码算法来进行编码。既然没有数据不能表示成二进制的，这就意味着一个SOAP文档可以使用base64字符集对任何类型的数据进行编码。这将为那些对于特定协议有特别意义的替代字符带来好处。

虽然在这里详细介绍base64编码的所有细节是不合适的，但是有必要来看一个简单的例子。base64 编码使用一个由 6 位值组成的字母表。其中的每个值都有一个替换字符。表 3-2 示例了 base64 字母表。

表 3-2：base64 字符集

6 位值	base64 字符
0~25	A~Z
26~51	a~z
52~61	0~9
62	+
63	/
PAD	=

为了对二进制数据进行编码，必须先把它分解成 6 位的片段。如果数据中的字节总数不能被 6 整除，右边的空位将用零填充。base64 编码是为电子邮件而设计的，

它将每一行数据限制在76个字符。在SOAP世界中，我们没有这个限制，因此不需要根据76个字符的限制来分割我们的数据；你可以在任何觉得方便的地方插入新行。

假设我们想要对字符串“<data>”进行编码。那些括号在对字符串进行编码时是个难题，但是在base64中却工作得很好。首先，让我们看看数据中每个字节的十六进制表示：

字符	<	d	a	t	a	>
16进制值	0x3C	0x64	0x61	0x74	0x61	0x3E

下一步我们要将它们转换为二进制，并将它们分成6位的片段。我们正好得到完整的8块，因此不需要填充数据。下面是6位的二进制片段和它们的十进制值以及相应的base64字符：

6位值	001111	000110	010001	100001	011101	000110	000100	111110
十进制值	15	6	17	33	29	6	4	62
base64字符	O	F	P	h	d	F	D	+

现在我们就能使用base64来编码数据并在XML中这样表示它：

```
<value xsi:type="SOAP-ENC:base64">OFPhdFD+</value>
```

为了解码，首先需要将字符串里的每个字符转换成等价的6位值。然后再将它们重新编组成8位值，每个值表示解码后数据的一个字节。这样你就应该会得到最初的字符串：“<data>”。

base64编码对于传送任何类型的二进制数据都是有用的。然而，如果你想要使用这个编码类型来传送一个10MB的位图文件，那你可得再考虑考虑。base64会使数据的大小增加大约1/3，数据的大小在一些情况下可能是很重要的。这并不是说不能这样做，当然，它可能对你的程序正好合适。

注意：处理大数据对象时，通常使用的一项技术是发送一个引用该数据的URL，而不是直接发送数据本身。接收者使用这个URL来另行获取该数据。如果你想要使你的SOAP消息体积较小，这是一个好办法，尤其是在接收者不一定使用该数据的时候。

我们将在第八章中讨论的另外一项技术，是以MIME附件的形式传送数据。这不会减少消息的整体大小，但是它确实能够把数据放在SOAP有效负载之外，从而改善SOAP处理的性能。

多态存取器

术语"多态（polymorphism）"在面向对象的软件设计和开发中经常被使用。它的意思是一件事物（通常是一个对象）能表示为其他事物的外表。虽然这本书不是有关面向对象编程的，但还是让我们谈谈我们通常在Java中是如何看待多态的，然后再看看它在SOAP中是什么意思。

考虑一下我们早先使用的那个叫做Employee的Java类。我们对经理使用了和一般职员一样的Employee类。当然在真正的世界里，经理会有一些普通职员所没有的特征和行为（不，我并不是指个性特色的差别，尽管我们都知道这也是一种差别！）。举例来说，一个Manager类可以照料（也可以说是控制）一组职员。我们可以创建一个从Employee类派生出的Manager类，然后把经理所特有的东西加进那个新类。之所以能这样做是因为经理也是职员。换句话说，经理是职员的一种，所以一个Manager对象的实例也会具有Employee对象的形态。如果Manager实例被作为某个Employee的引用传递给某个函数，接收方法不会知道也不需要知道这个对象实际上是一个Manager实例。它把它视为一个Employee实例。这是多态的一个例子。

另一个多态的例子是微软的COM（Component Object Model，组件对象模型）编程中的variant（变体）类型。变体表示一个数据值和一个相关联的数据类型指示器。当获取数据的时候，数据类型指示器会告诉用户要使用哪一种存取器。

在SOAP中，多态的机制不像它们在Java中的那样精细，SOAP多态更接近COM的变体。如果你已经看过SOAP规范，你可能注意到了有关多态的整个一节只有几行的内容。多态在SOAP中只意味着存取器中可以包含一个类型声明。就这些。

下面的XML片段显示了一个名为quantity的存取器。在第一种情况下，我们假设quantity依据某些模式被定义为xsd:int类型。因此，没有使用类型声明。它不能是其他类型的值。在第二种情况下，quantity存取器中包含了类型声明，它允许存取器以xsd:float类型取得数据。第二种形式就是一个多态的存取器。

```
<quantity>37</quantity>
<quantity xsi:type="xsd:float">37</quantity>
```

当我们介绍异构数组和通用复合类型时还会看到多态存取器的一些重要应用。

复合数据类型

SOAP规范中说得最多的有两种复合数据类型是：结构和数组。规范中对这两种类型的定义如下：

结构（Struct）

一个复合值，它的成员值之间仅靠存取器的名称来区分，而且其中不能包含两个名称相同的存取器。

数组（Array）

一个复合值，它的成员值之间仅靠成员的顺序位置来区分。

我们在讨论SOAP编码术语的时候曾提到过这些类型的定义。这些数据结构对许多程序设计语言来说都是很普通的，而且SOAP定义它们的方式也没什么不同。

结构

很多程序语言都支持结构，尽管它们并不完全一致。举例来说，在C语言中你可以使用struct关键字声明一个复合数据结构。这个关键字本质上允许你定义一个由命名的部分组成的数据类型或实例，其中的每个部分都可以是任何一种其他的有效数据类型的实例。在C++里你能用与C语言同样的方式使用struct关键字。不过，你会更时常使用一种称为类的特别（而且功能更多）的数据结构。在Java中没有明确的结构的概念，但是和在C++中一样，我们可以使用类来实现与结构相同的功能。

这里是一个Java类，它包含的字段表示一把吉他的某些特性：

```
class Guitar {
    public java.lang.String _manufacturer;
    public java.lang.String _model;
    public int _year;
}
```

这可能不是在现实世界的Java应用程序中描述一把吉他的最好方法，但是它完全有效而且非常近似于C和C++中的struct变量。与这个类相应的XML模式片段看起来可能是这样的：

```
<element name="Guitar" type="Guitar"/>
<complexType name="Guitar">
     <element name="manufacturer" type="xsd:string"/>
     <element name="model" type="xsd:string"/>
     <element name="year" type="xsd:int"/>
</complexType>
```

这是一个使用上面模式的串行化Cuitar实例的例子：

```
<Guitar>
    <manufacturer>Epiphone</manufacturer>
    <model>Sheraton II</model>
    <year>1997</year>
</Guitar>
```

数组

SOAP数组的实例使用数据类型SOAP-ENC:Array来声明。数组的元素可以是任何有效的SOAP类型，甚至也可以是数组。数组必须含有一个SOAP-ENC:arrayType属性，它说明了其元素的数据类型和数组的维数。

让我们先来看一个数组的声明，它是一个包含5个浮点数的一维数组。实际值在这里被省略掉了（我们将会在下一个例子中看到这些值）。

```
<someNumbers xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:float[5]">
    . . .
    . . .
</someNumbers>
```

一个数组将被串行化为一系列的元素值，其中元素顺序表示了它们的顺序位置。由于数组成员是通过位置来访问的，因此数组中每一个值的实际元素名称并不重要；XML要求为数组中的每一个元素给出一个名称，但是SOAP并不在意这个名称是什么。你时常会发现一个串行化的数组的每个元素都使用了相同的名称，不过那不是要求的。SOAP规范建议使用元素的数据类型来作为元素的名称。同样，这也只是一个建议而已。如果可能我喜欢使用反映数组内容的名称。这里是一个含有数据值的数组：

```
<someNumbers xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:float[5]">
    <number>1.234</number>
    <number>2.345</number>
    <number>3.456</number>
    <number>4.567</number>
    <number>5.678</number>
</someNumbers>
```

既然元素按它们出现在数组中的顺序被串行化，你就能通过查看来确定其顺序位置。与Java和许多其他的语言一样，SOAP数组的索引从0开始。因此值4.567的顺序位置就是3（它是数组的第4个元素）。

在多维数组的情况下，元素以行优先的顺序列出。它的意思就是越靠右边的顺序位置变化得越快。图3-2是一个二维数组，它包含了从1到6的整数。数组的大小是3行2列，该图以纵向显示主维，以横向显示次维。

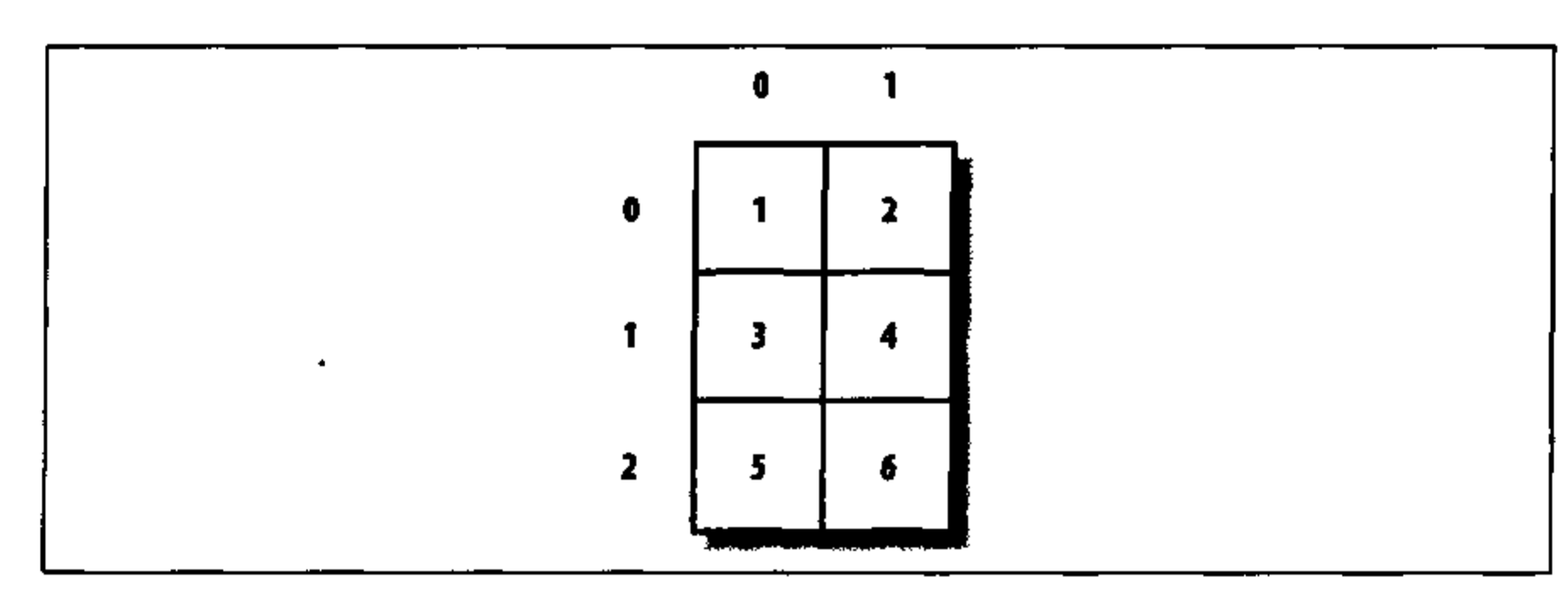

图 3-2: 一个二维数组

SOAP-ENC:arrayType允许你通过用逗号隔开每一维的办法指定多维数组中每一维的大小。图 3-2 中的数组的编码可以是这样的:

```
<values xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:int[3,2]">
    <value>1</value>
    <value>2</value>
    <value>3</value>
    <value>4</value>
    <value>5</value>
    <value>6</value>
</values>
```

到现在为止，我们所看到的数组都是同构的：数组中的每个元素都是同一种数据类型。但是对每个值类型都不同的异构数组进行编码也是可能的。这是多态存取器显得很重要的一个地方。为了创建一个异构数组，SOAP-ENC:arrayType要被赋值为 xsd:ur-type。这个类型意味着所定义的数组值的数据类型不受 SOAP-ENC:arrayType属性的限制，相反，数组里的每一个数据值在编码时都包含了数据类型。这里是一个包含不同类型值的数组:

```
<mixedNuts xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:ur-type[3]">
    <nut xsi:type="xsd:float">3.14159</nut>
    <nut xsi:type="xsd:string">SOAP</nut>
    <nut xsi:type="xsd:int">8141992</nut>
</mixedNuts>
```

SOAP也允许省略数组的维数大小。这意味着数组中特定维的大小可以通过数组中成员的数量来决定。因此 mixedNuts 例子中的第一行也可以写成下面这样:

```
<mixedNuts xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:ur-type[]">
```

通过检查这个 XML，我们能够确定这个一维数组的大小是 3。

不完全数组

在某些情况下，一个数组可以不被填满数据。换句话说，在数组中可能有一个或者多个元素是没有值的。在下面的Java代码中，数组contestants是一个大小为 10 的一维字符串数组，但是只有在位置 6、7 和 8 的元素才包含数据值:

```
java.lang.String contestants[] = new java.lang.String[10];
contestants[6] = "Rob";
contestants[7] = "Arnold";
contestants[8] = "Scott";
```

如果我们对这个数组使用标准编码，我们将需要编码它的每个元素，即使其中的大多数值都不存在。不应该使用空值来代表null值。在我们的这个字符串数组中，元素包含一个空字符串与元素值为null之间是有区别的。我们有时会想要使用空字符串来编码，因为它正确地反映了该值存在。在上面的例子中，数组contestants中除了位置为6、7和8的值以外都是不存在的；它们不包含java.lang.String的实例。我们可以通过对每一个元素使用xsi:null属性，并将它们赋值为1来编码这些null元素。这些元素将不包含任何数据：

```
<contestants xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[10]" >
    <name xsi:null="1"/>
    <name xsi:null="1"/>
    <name xsi:null="1"/>
    <name xsi:null="1"/>
    <name xsi:null="1"/>
    <name xsi:null="1"/>
    <name>Rob</name>
    <name>Arnold</name>
    <name>Scott</name>
    <name xsi:null="1"/>
</contestants>
```

这样做完全正确，但是它的效率不高。SOAP允许这样编码数组：除了包含值的元素块之外可以省略任何其他的东西。要这么做，就要在数组的声明中包含SOAP-ENC:offset属性。这个属性的值表示数组中第一个实际存在的元素的顺序位置（从零开始）。因此我们可以将数组contestants的编码改为下面这样：

```
<contestants xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[10]"
             SOAP-ENC:offset="[6]">
    <name>Rob</name>
    <name>Arnold</name>
    <name>Scott</name>
</contestants>
```

这当然比传送所有null数组值更有效率。注意SOAP-ENC:arrayType属性仍然指定数组的大小为10。这很重要，因为我们想让这个数组的接收者使用正确的数组大小。

稀疏数组

如果所传输的不完全数组中的元素是连续的，就像contestants例子里的那样，SOAP-ENC:offset属性就会工作得很好。但是这当然并不是不完全数组的惟一可行方案。假设数据值是不连续的，就像下面的代码这样，那又该怎么办？

```
java.lang.String contestants[] = new java.lang.String[10];
contestants[1] = "Rob";
contestants[5] = "Arnold";
contestants[8] = "Scott";
```

这种数组称为稀疏数组（sparse array）。SOAP为稀疏数组提供的编码方式是，允许数组里的每一个被传输的元素通过包含一个SOAP-ENC:position属性来指定它自己的顺序位置。因此我们现在可以将上面的数组编码为：

```
<contestants xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[10]">
    <name SOAP-ENC:position="[1]">Rob</name>
    <name SOAP-ENC:position="[5]">Arnold</name>
    <name SOAP-ENC:position="[8]">Scott</name>
</contestants>
```

通过部分编码并传输，稀疏数组有潜力远比编码不完全数组中所有不存在的值更有效率。不过，你应该知道，许多目前的SOAP实现还不支持这些结构。但我相信这只是时间问题，而且我希望当你读到本文时，SOAP实现工具将会支持不完全数组和稀疏数组。

普通复合类型

很多时候预先不知道串行化的复合类型值的准确格式。这可能是因为没有为数据定义模式，或者该数据来自某个通用数据库查询的结果。举例来说，假设你使用SOAP来查询一个数据库，并发送了下面这样的查询请求：

```
select * from guitars;
```

为防止你不熟悉SQL语法，我们需要解释一下，这条语句在名为guitars的表中查询所有记录，并要求返回表中所有记录的所有字段。然而，我们预先不可能知道字段的名称，或者一共有多少字段。表3-3表示了一个叫做guitars的简单

数据库表，它包含两个记录。表中有三个字段:manufacturer（制造商）、model（模型）和year（年份）。

表3-3: guitars表

制造商	模型	年份
Gibson	Les Paul	1958
Fender	American Stratocaster	2001

我们预先不知道我们的SQL查询会返回哪些字段，一个有效的SOAP响应可能包含下面这样的串行化数据:

```
<guitars>
    <guitar>
        <manufacturer xsi:type="xsd:string>Gibson</manufacturer>
        <model xsi:type="xsd:string">Les Paul</model>
        <year xsi:type="xsd:int">1958</year>
    </guitar>
    <guitar>
        <manufacturer xsi:type="xsd:string>Fender</manufacturer>
        <model xsi:type="xsd:string">American Stratocaster</model>
        <year xsi:type="xsd:int">2001</year>
    </guitar>
</guitars>
```

这个XML包含了普通的复合数据。该数据的接收者要指出如何处理它。结构和数组都可以在普通复合类型中使用。每个guitar元素的格式类似于SOAP的结构，而guitars元素则更像数组，尽管它没有使用xsi:type="SOAP-ENC:Array"来声明。每个guitar的元素中都使用了多态数据存取器，因为它们的数据类型可能不可预知。

默认值

SOAP规范简要介绍了对默认值的处理。它指出省略传输数据中的元素将暗示它使用默认值。在这方面C++语言提供了一个很好的编程方法，它允许为方法的参

数定义默认值，这就使得调用者可以省略这些参数的值。这里是C++中一个方法定义的例子：

```
int getValue (int param1, int param2 = 1)
```

在这种情况下，param2的默认值是1。所以如果调用者在调用getValue()时没有提供该参数的值，系统将使用值1。Java中不支持这种语法，但可以定义支持默认值的SOAP事务。

让我们来假设这样一个定单录入系统，它预期一个名为order的复合元素，order中的每一项包含一个数量、一个项目编号和一个描述。我们把数量的默认值设为1，因此如果某个项的定购数只有1，发送者就不必在该项中包含数量。在这种情况下，我们可以将一个定单串行化成：

```
<order>
    <item>
        <quantity>2</quantity>
        <itemNum>1020304</itemNum>
        <description>Really cool device</description>
    </item>
    <item>
        <itemNum>1020305</itemNum>
        <description>Some other device</description>
    </item>
</order>
```

我们在第二个项目中就没有包含一个数量元素。相反，我们希望接收者使用默认值。重要的是SOAP没有定义默认值，它也不要求它们。如何正确处理是发送者和接收者的事。

SOAP的root属性

链表、树以及有向图是常用的数据结构。然而，我们还没有介绍过一种用于指定这些结构中的出发点或根（root）的串行化机制。SOAP的root属性正是用于这个目的的。将一个元素的SOAP-ENC:root属性设为1表示它是链表这样的结构的根。

这里是一段单链表的Java代码，它的节点元素为整型，后面是一个包含三个节点的链表实例。节点的值分别是50、60和70，而且它们以升序放入链表，50是链表的头（或根）。

```
class Node {
    int _value;
    Node _next;
    public Node(int value, Node next) {
        _value = value;
        _next = next;
    }
}
Node n3 = new Node(70, null);
Node n2 = new Node(60, n3);
Node n1 = new Node(50, n2);
```

变量n1是链表的根。将链表串行化后可能是这样：

```
<list>
    <node>
        <value id="node-3" xsi:type="xsd:int">70</value>
        <next xsi:null="1"/>
    </node>
    <node>
        <value id="node-2" xsi:type="xsd:int">60</value>
        <next href="#node-3"/>
    </node>
    <node>
        <value SOAP-ENC:root="1" id="node-1" xsi:type="xsd:int">50</value>
        <next href="#node-2"/>
    </node>
</list>
```

我们对一个节点使用了对下一个节点的引用，而SOAP-ENC:root属性指定了该链表的根。

第四章

RPC 形式的服务

虽然SOAP没有限定使用特定形式的分布式计算，但是它常使用RPC模型。这是你所能预期的全部东西；依照SOAP规范，SOAP的设计目标之一是要封装并交换RPC调用。这种方式非常适合于Java编程，因为Java对象的方法调用可以很容易地翻译成RPC调用。在本章中我们将来看看SOAP中RPC请求和响应消息的结构。在这里我们将用Java实现几个RPC服务，并将它们分别部署到Apache SOAP和GLUE中。当然，我们也会编写一些Java代码来调用这些服务。

SOAP RPC 元素

创建一个SOAP RPC请求要使用在第二章和第三章中介绍的SOAP结构和编码。对RPC不需要其他新的XML和数据编码形式。让我们来看看在SOAP中表示一个RPC方法调用都需要些什么：

- 目标对象
- 方法名称
- 方法参数
- SOAP头数据

目标对象 URI

目标对象 URI 实质上就是我们想要使用的服务的资源地址。当你浏览 Web 的时候，你一直在使用资源地址 ——你通过指定页面的资源地址导航到一个网页。例如，要查看描述O'Reilly出版公司JavaBeans书籍的作者的网页，你需要使用URL *http://www.oreilly.com/catalog/javabeans/author.html*。协议、服务器地址和资源都写在一起。让我们把它们分离出来，这样你可以看清每个部分。服务器地址是 *www.oreilly.com*，HTML 页资源是 */catalog/javabeans/author.html*，而协议是 *http*。

这正是我们在SOAP RPC请求消息中用来指定目标对象的方式，其中只有一点不同：用目标对象名称取代了 HTML 页资源。目标对象的名称由传输协议（比如，一个 HTTP头）而不是SOAP消息本身传送。除SOAPAction属性之外，这是SOAP RPC中惟一一个在消息体之外传输的部分。在Apache SOAP服务器中，一个RPC消息的目标对象通常是 /soap/rpcrouter。这个 URI 表示在 Apache SOAP 中路由 SOAP RPC 消息的资源。对于GLUE服务器，目标资源通常是 /glue，后跟服务的名称本身。我们稍后会看到这两个服务器的 HTTP 头。

方法名称

你可能猜到了，方法名称是方法被调用时使用的名称。在第二章中我们讨论过一个在气象台提供目前温度的服务。那个服务暴露了一个叫做 getCurrentTemperature的方法。如果你习惯用Java术语来考虑，你可以将方法名称想像成一个服务对象的实例的公共方法。

在 SOAP XML 中，被调用的方法像 SOAP 的结构（注 1）那样被包装，它为每个参数包含了一个存取器。方法名称 struct 是 SOAP Body元素的直接子元素。让我们再来看看例子：

注 1： 术语“结构（struct）”是指用于表示一个服务方法以及其参数的 XML 元素的层次结构。对于 C 程序员来说，不要将它与 C 的 struct 相混淆，它们显然有着不同的含义。

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV=
       "http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle=
       "http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Body>
          <m:GetCurrentTemperature xmlns:m="WeatherStation">
                <m:scale>Celsius</m:scale>
         </m:GetCurrentTemperature>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

注意被调用的方法的名称以服务对象的名称作为名称空间限定。名称空间应该是一个URI，它用来为目标服务提供一种解释方法意义的机制。URI通常会指向描述某个方法的资源，不过在本例中这样做就行了。但是我们正在使用什么服务对象呢？它不是目标对象。它是目标对象通过一个特殊的名称知道的另一个对象。在这个例子中，服务对象的名称是WeatherStation。SOAP规范中没有描述用来将服务对象映射到它们的物理实现的机制。每个SOAP实现都有自己的做法，这一点我们将在“部署服务”一节中看到。

方法签名（method signature）也可能是方法结构的一部分，但这不是必需的。事实上，SOAP规范没有描述方法签名的格式，而本书也不会讨论签名（而且我们所讨论的服务器也不用它）。我提到方法签名仅仅是想让你在使用一个支持它的实现时不会感到困扰（注2）。

方法的参数

方法的参数被建模为存取器，而且是方法结构的直接子元素。对于所有的in和in/out参数都是如此；所有的out参数在方法调用结构中都不会出现。SOAP允许对缺少参数的方法进行处理，如同我们在第三章中看到的一样。是否处理一个缺少参数的方法由应用程序决定。如果不允许缺少参数，应用程序会用一个SOAP错误进行响应。每个存取器在方法的参数名之后命名，而且它的类型与相应的服务方法类型一致。

注2：　我怀疑你是否会在SOAP消息中遇到方法签名。这个概念好像还没有被业界接受，我所使用过的SOAP引擎都没有实现它。

前面示例的SOAP封套中的getCurrentTemperature方法有一个参数。这个参数叫做scale，它用来指定返回值所用的温度单位。这个参数是使用服务名称来进行名称空间限定的，这与用一个模式名来限定存取器不一样。这意味着scale元素要根据WeatherStation名称空间中的定义来表示。我们可以在scale中包含一个xsi:type属性，但是省略该属性也是有效的。只要类型与名称空间所指定的模式中指定的一致，你就不需要在SOAP消息中再指定类型。你甚至可以发送一个不同的类型，只要你指定了它的类型。

让我们来看一些Java代码并看看它是如何映射到SOAP RPC方法调用上去的。DowJones30类表示目前道琼斯工业指数的平均水平。DowJones30包含了一个叫做getCurrentLevel()的方法，它返回目前贸易期间的道琼斯工业指数平均水平。它还有一个方法叫做getPrice()，该方法返回道琼斯工业指数某个指标股目前的交易价格。getPrice()接受一个名为stock的参数以表示股票代码，以及一个currency字符串参数以指定返回价格时所使用的货币类型。返回的价格是一个浮点数。

```
public class DowJones30 {
   private float _level;
   public float getCurrentLevel() {
      return _level;
   }
   public float getPrice(String stock, String currency) {
      float result;
      // 确定股票的价格
      // 并用指定的货币单位返回它
      return result;
   }
}
```

现在假设我们已经部署了一个由DowJones30类所实现的名为DowJones的服务。为了调用DowJones服务的getCurrentLevel()方法，我们使用了一个像这样的SOAP封套:

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV=
       "http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle=
       "http://schemas.xmlsoap.org/soap/encoding/">
```

```
        <SOAP-ENV:Body>
            <m:getCurrentLevel xmlns:m="DowJones">
            </m:getCurrentLevel>
        </SOAP-ENV:Body>
    </SOAP-ENV:Envelope>
```

其中没有参数，因为getCurrentLevel()方法不带任何参数。现在我们来调用getPrice()方法，使用IBM作为股票代码，USD（美元）作为货币单位。这个封套看起来会是这样的：

```
    <SOAP-ENV:Envelope
        xmlns:SOAP-ENV=
            "http://schemas.xmlsoap.org/soap/envelope/"
        SOAP-ENV:encodingStyle=
            "http://schemas.xmlsoap.org/soap/encoding/">
        <SOAP-ENV:Body>
            <m:getPrice xmlns:m="DowJones">
                <m:stock>IBM</m:stock>
                <m:currency>USD</m:currency>
            </m:getPrice>
        </SOAP-ENV:Body>
    </SOAP-ENV:Envelope>
```

方法的响应

方法响应也使用SOAP结构的形式。方法的结构中包含一个返回值的存取器，随后是各out或in/out参数值的存取器，如果方法签名（注3）存在，它们的顺序将与在方法签名中的顺序一致。out和in/out值的存取器名称要与它们在方法签名中的名称相对应，但返回值的名称无关紧要。我喜欢将返回值的名称设为能反映方法名本身的值。我也趋向于使用动词－名词形式命名服务方法，这样将允许我使用名词为返回值存取器命名。举例来说，如果我的服务名称是computeSize，那么我会将返回值存取器命名为size。

现在变得较为常用的一种方式是，在被调用的方法名后面加上Response一词，并用它来命名返回值的结构。所以对于一个叫做computeSize的方法，包含结果的

注3：在前面已经提到过，SOAP规范中描述的方法签名还没有被实现。在此提到它仅仅是让你知道有这回事。

结构会被命名为computeSizeResponse。虽然这是一个常用的惯例，但是它不是要求的，而且你会发现有一些SOAP实现不这样做。

这里是调用getCurrentLevel方法可能返回的SOAP封套：

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV=
        "http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle=
        "http://schemas.xmlsoap.org/soap/encoding/"> <SOAP-ENV:Body>
          <m:getCurrentLevelResponse xmlns:m="DowJones">
                <m:currentLevel>10513.15</m:currentLevel>
          </m:getCurrentLevelResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

错误响应

如果一个SOAP RPC方法调用失败，它将返回一个SOAP错误。让我们看一个SOAP错误的例子。假设我们调用了DowJones服务的getPrice方法但是指定了一个非法的货币参数。服务可能返回一个像这样的错误：

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV=
        "http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle=
        "http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
      <SOAP-ENV:Fault>
          <faultcode>SOAP-ENV:Client</faultcode>
          <faultstring>Client Error</faultstring>
          <detail>
              <m:dowJonesfaultdetails xmlns:m="DowJones">
                  <message>Invalid Currency</message>
                  <errorcode>1234</errorcode>
              </m:dowJonesfaultdetails>
          </detail>
      </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP RPC方法调用要么失败要么成功，但是不可能同时发生。因此你无法找到一个既包含方法响应又包含错误响应的例子。

可选的头数据

有时处理RPC需要方法签名中不包含的信息。举例来说，这种情况可能是需要向服务传递一个用户名。对方法调用来说，用一个用户名作为参数通常不是很合适，但是尽管如此接收应用程序可能需要它。像这样的信息可以编码到SOAP头中，我们在第二章的“Header元素”中讨论过。还可能有特定于传输的数据；举例来说，你可能想要传递一个cookie信息作为HTTP头的一部分，用以指定当前方法调用使用的HTTP会话。不管是哪种情况，这个附加的数据都是在SOAP体的边界之外被编码的（注4）。

服务激活期

对于任何分布式系统，对象的生存期都是一个重要的部分，它有时被称为服务作用域（service scope）或服务激活期（service activation）。无论使用什么术语，它都包含了系统中各种可能类型的对象生存期模型。在基于HTTP的SOAP系统中通常有三种服务激活期模型：

请求级服务激活期

对于一个给定的服务对象，对每次请求都会创建该对象的一个新实例，这个实例将一直存在，直到请求完成。这种服务激活期用于那些不需要维护状态的服务对象。

应用程序级服务激活期

由服务对象的一个实例来处理服务应用程序整个生存期中的所有方法调用。这种激活期可以在需要维护状态的服务中或者在对象创建时使用了大量资源的情况下使用。

注4：实际上，发送特定于传输的数据，比如cookie，可能不是一个好主意，它会使你局限于某种特定的传输协议。使用其他一些与传输无关的会话数据可能会更合适。

会话级服务激活期

由服务对象的一个实例来处理一个会话生存期中的所有方法调用。如你所知，会话不必是一个标准。通过建立和传递令牌，或者使用已有的办法比如cookie，你可以实现特定于应用程序的会话。当然在这些情况下，你需要传递一些附加的头信息。

SOAP规范没有对服务对象的生存期定义任何条件和行为；SOAP实现没有被真正要求支持这里描述的三个模型。然而，这些激活期模型在SOAP之外也被用到，而且你可以预期即使不是所有现有的SOAP实现都支持它们，也有大多数支持。对于某个服务，每个实现都有它自己独特的设置激活期模型的方法，但其行为应该是一致的。你或许会发现一个服务的服务激活期模型被看成服务部署的一部分。我们将在本章后面通过实例来讨论服务部署时涉及这个主题。

一个简单的服务

让我们来设计一个叫做CallCounterService的简单服务。这个服务跟踪它所接收的方法调用的数量。它本身没什么用处，但是它给我们一个机会建立一个服务并运行它，并使我们能看到服务如何部署以及服务激活期模型是如何工作的。我们将这个例子的类放在了javasoap.book.ch4包中。请确定已在系统上为这个包创建了适当的目录，并在服务器和客户系统的类路径中包含了它。

这里是MethodCounter类的源代码，它实现了CallCounterService。注意Java类名不必与服务名一致。

```
package javasoap.book.ch4;
public class MethodCounter {
   int _counter;
   public MethodCounter() {
      _counter = 0;
   }
   public int getCount() {
      return _counter;
   }
   public boolean doSomething() {
      _counter += 1;
```

```
        return true;
    }
}
```

可以看到，这段Java代码没有什么特别的。MethodCounter包含一个叫做_counter的变量，它跟踪对象生存期中建立的方法调用的数量。这个类包含了一个叫做doSomething()的方法，该方法将_counter变量加1并返回一个表示调用是否成功的布尔值。还有一个称为getCount()的方法，它返回doSomething()被调用的次数。我们依照JavaBeans的命名惯例，为一个叫做Count的只读整型属性命名了getCount()方法。我们遵循JavaBeans惯例是有理由的。首先，只要有可能我们就应该尽量遵循标准的惯例。更重要的是，我们稍后将会看到，一些SOAP实现可以直接使用JavaBeans的属性，而不需要程序员做额外的工作。

注意我们的服务类使用了不带参数的构造函数。这很重要。Java SOAP实现常常通过调用Class.newInstance()动态地加载服务类。这个方法要求类拥有一个不带参数的构造函数。当然，如果我们没有显式地包含一个构造函数，默认的不带参数的构造函数就会自动建立。我们显式地包含构造函数是为了说明确实需要它的事实，不管它是如何被包含的。

在进一步讨论之前，让我们看一看用来封装对doSomething()方法的调用的SOAP封套：

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV=
        "http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle=
        "http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
        <m:doSomething xmlns:m="CallCounterService">
        </m:doSomething>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP Body元素的第一个子元素是被调用的方法的名字；该元素使用服务名称（而不是实现服务的类名）作为名称空间限定。既然我们的服务类的doSomething()方法不带参数，doSomething元素也就没有参数。

部署服务

服务的部署不是SOAP规范的一部分，所以每个实现都有它自己的部署程序。我们将会看到使用两个实现 ——Apache SOAP和GLUE的服务部署。

在Apache SOAP中部署

在Apache SOAP中，你必须创建一个部署描述文件（deployment descriptor），它是一个XML文件，包含关于服务以及一个实现服务的Java类的信息。CallCounterService由Java类javasoap.book.ch4.MethodCounter实现，让我们看看它的部署描述文件：

```
<isd:service
   xmlns:isd="http://xml.apache.org/xml-soap/deployment"
   id="urn:CallCounterService">

   <isd:provider type="java"
         scope="Application"
         methods="doSomething getCount">
     <isd:java class="javasoap.book.ch4.MethodCounter"
         static="false"/>
   </isd:provider>
   <isd:faultListener>
   org.apache.soap.server.DOMFaultListener
   </isd:faultListener>
   <isd:mappings>
   </isd:mappings>
</isd:service>
```

最外面的XML元素叫做service，它使用标识符isd作为名称空间限定，标识符isd表示http://xml.apache.org/xml-soap/deployment名称空间。这个service元素还有一个id属性，它用来为被部署的服务指派一个惟一的名字。我为服务名称CallCounterService添加了前缀urn。你不一定要这样做，但这是我所常用的一个做法，至少在这个例子中是这样。只要知道它不是必需的就行了。

第一子元素叫做provider，它描述了服务的实现。与service的所有子元素一样，这个元素也是使用从父元素中继承的isd标识符来作为名称空间限定的。

provider元素有三个属性。第一个属性是type，它告诉Apache SOAP这个服务正在使用什么实现类型；我们将它的值设为 java（Apache SOAP 允许除Java之外的其他服务实现，但我们将不做介绍）。scope属性指定被部署服务的服务激活期模型。这个属性可以取的值有 Application、Session和Request。这些与前面介绍的激活期模型相对应。我们现在使用Application级的服务激活期模型；我们随后将试试其他的值。provider元素的第三个属性叫做methods。这个属性列出了该服务暴露的方法的名称。由于 Apache SOAP 在进行方法调用时使用了动态的类加载和Java映射机制，因此这里的方法名要与实现服务的Java类中的方法相对应。方法名之间用空格隔开。对于本例我们将暴露 doSomething 和 getCount 方法。

provider包含一个子元素，它进一步描述了服务实现的细节。我们为type属性分配了值 java，它与子元素名 java 对应。java 元素有两个属性并且没有数据。class属性的值为实现服务的Java类的全名，在这里是MethodCounter。一个叫做 static的属性用来指出服务实现类中被暴露的方法是不是静态的。

部署描述文件的下一个元素是service的子元素 faultListener。这个元素的数据值是一个Java类的全名，它用来产生一个错误监听器对象。当一个错误状况出现时，这个对象负责创建SOAP 错误并返回它们。我们现在使用的是 org.apache.soap.server.DOMFaultListener类，它由Apache SOAP提供。在第七章中我们将进一步介绍 SOAP 错误并且编写自己的错误监听器。

最后一个元素叫做 mapping。在第六章中我们将使用这个部件来定义对自定义Java类型的串行化和反串行化进行管理的Java类。而现在我们不需要任何映射，因此在这里的映射部分中没有任何内容。

接下来，我们需要使用部署描述文件来部署我们的服务。我已将部署描述文件命名为*CallCounterService.dd*；你使用什么名字并不要紧，只要它对你有意义并且你能识别它就行。现在，如果你的服务程序还没有运行，你需要启动它。Apache SOAP包括一个Java应用程序，你可以通过命令行来让它部署一个服务。这个应用程序叫做服务管理程序客户（org.apache.soap.server.ServiceManagerClient）；它能部署、卸除和列出服务。这是一个部署服务的命令：

```
java org.apache.soap.server.ServiceManagerClient
  http://georgetown:8080/soap/servlet/rpcrouter deploy CallCounterService.dd
```

第一参数是 Apache SOAP RPC 路由器的 URL；我正在 Apache Tomcat 服务器上运行 Apache SOAP，它的服务端口为 8080。服务器是本地网络上的一个叫做 georgetown的机器。第二个参数 delpoy告诉服务管理程序客户我们正在部署一个服务。最后一个参数CallCounterService.dd是部署描述文件的名字。如果一切正常，你将看不见来自服务管理程序客户的任何输出；你将重新回到命令提示符。如果你得到任何异常或其他错误消息，请仔细查看输出。常见的问题有类的路径不对、运行命令的目录不是部署描述文件所在的目录等等。

在成功部署服务之后，就能通过再次调用服务管理程序客户来校验它。这一次我们将要求服务管理程序列出所有被部署的服务：

```
java org.apache.soap.server.ServiceManagerClient
  http://georgetown:8080/soap/servlet/rpcrouter list
```

第一个参数和前面是一样的：也是 Apache SOAP RPC 路由器的 URL。list 告诉服务管理程序客户列出当前已经部署的服务。你会看到下面这样的响应：

```
Deployed Services:
    urn:CallCounterService
```

Apache SOAP 将保存被部署的服务的信息，因此如果你稍后关闭运行着 Apache SOAP 的服务器，过一会再重新启动它的时候，该服务仍然是可用的。但是，它并不保存实现这些服务的对象的状态；因此它们又会像第一次启动一样，具有一个新的生存期。

Apache SOAP 管理工具

你可能会发现使用基于 Web 的 Apache SOAP 管理界面更为简单。这个界面可以通过一个 URL 来打开，这个 URL 包含运行 Apache SOAP 服务器的地址和端口，后跟 SOAP Admin 资源。我是在 georgetown 的 8080 端口上运行 Apache SOAP 的，所以这个 URL 就是 *http://georgetown:8080/soap/admin/index.html*。图 4-1 显示了 Apache SOAP 管理页面。

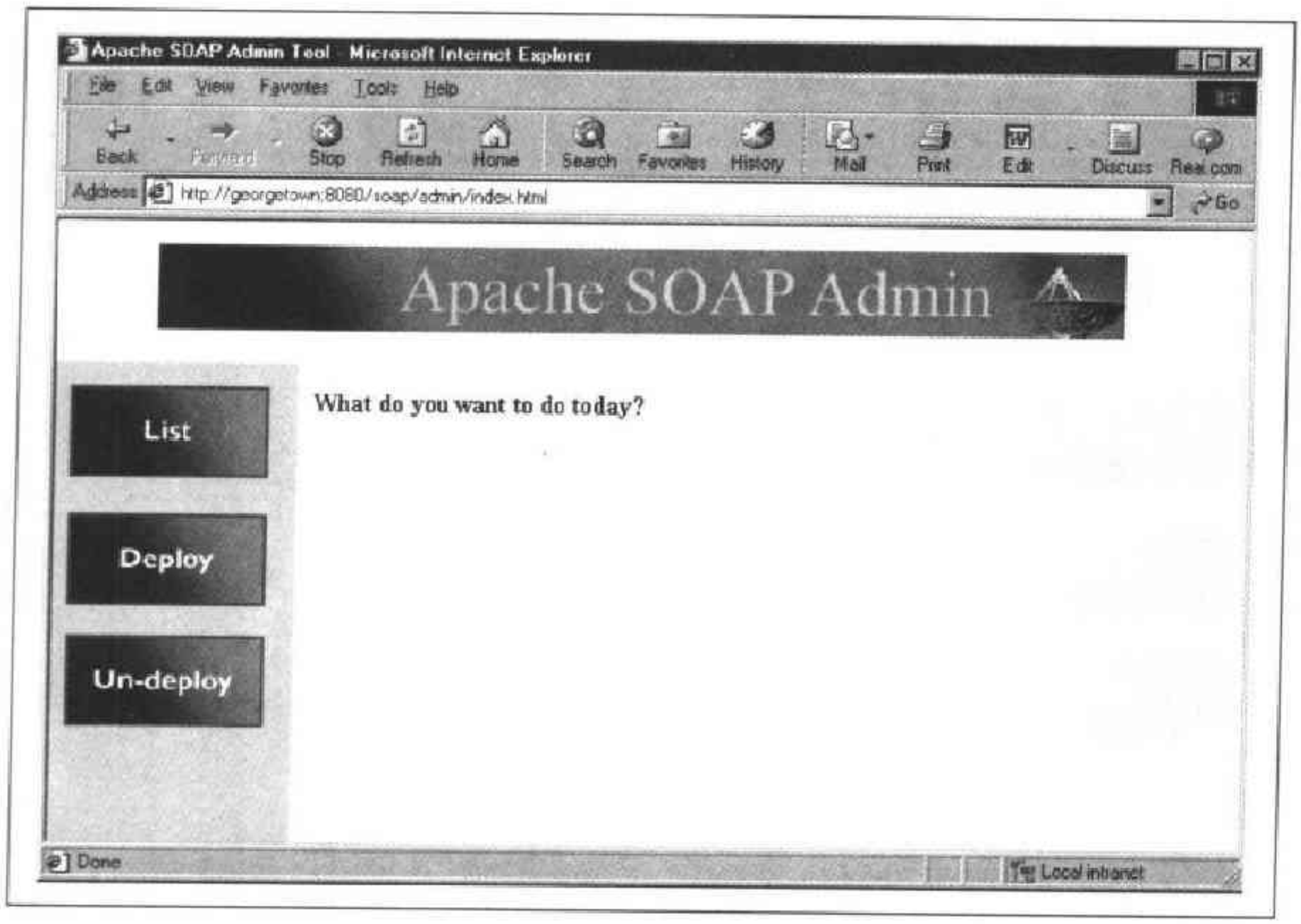

图 4-1: Apache SOAP 管理页面

List 按钮将带你到一个列出所有已部署的服务的页面。你将看见 urn:CallCounterService被列出，如图 4-2 所示。urn:CallCounterService链接会带你到一个展现我们的服务部署细节的页面，如图 4-3 所示，你能看到我们在部署描述文件中指定的各种部署细节。

你也能在这个界面中部署和卸除服务。如果你单击 Un-deploy，就会看到一个已部署的服务的列表。要卸除一个服务，只要在它的链接上单击。如果你愿意，可以现在就卸除 urn:CallCounterService。你也可以使用服务管理程序客户，通过命令行来卸除一个服务:

```
java org.apache.soap.server.ServiceManagerClient
  http://georgetown:8080/soap/servlet/rpcrouter undeploy urn:CallCounterService
```

既然现在服务已经不被部署，单击 Web 界面上的 Deploy 按钮。图 4-4 显示了服务部署的表单。部署服务的信息已被填写。

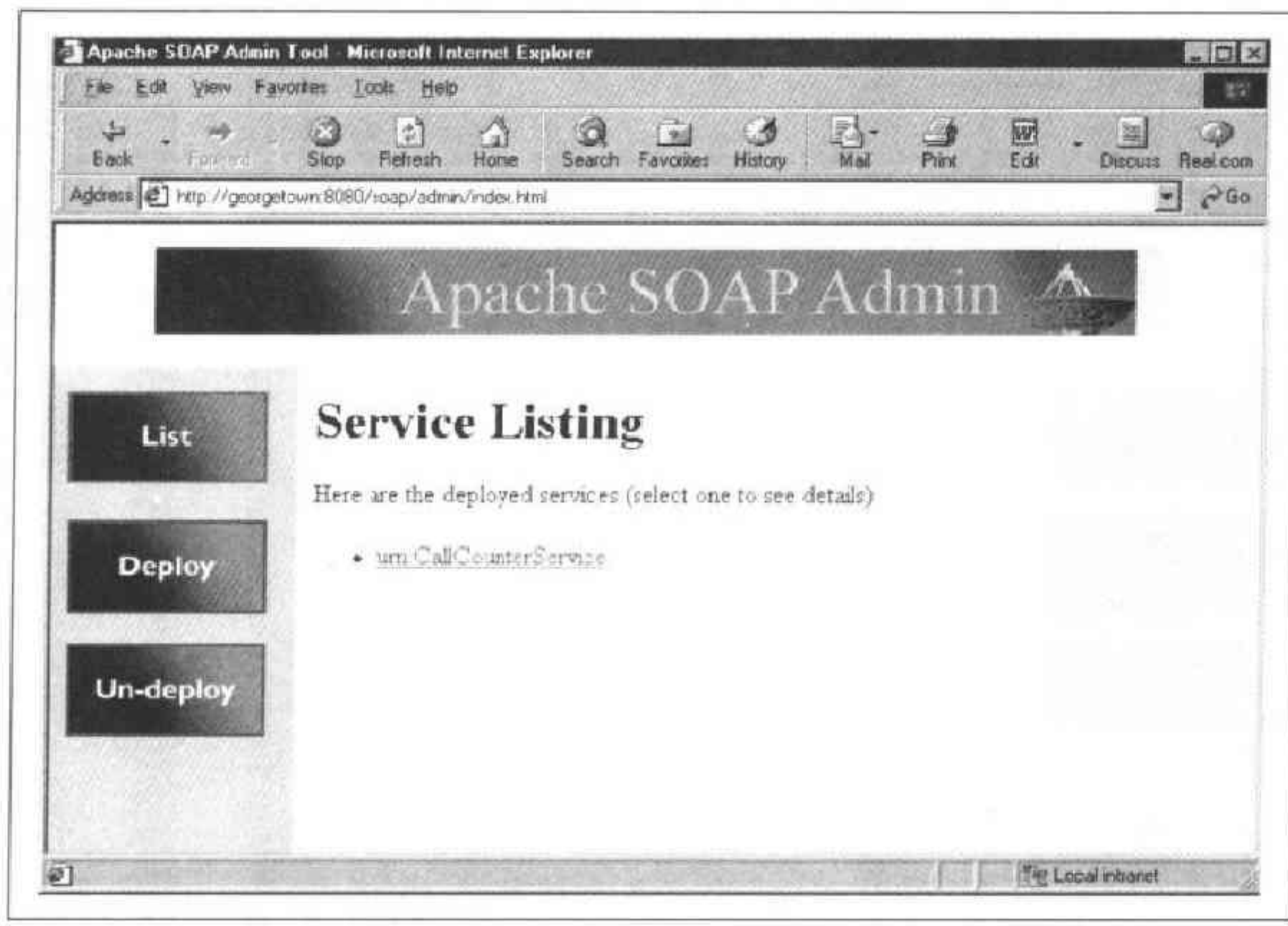

图 4-2: 当前已部署的服务

为了部署这个服务，滚动到表单的最底端，你可以看到一个标示为 Deploy 的按钮。单击该按钮，服务将会被部署。你可以使用 Web 界面去查看被部署的服务，以确保它与前面所示的一致。Apache SOAP 不会验证任何信息，因此你需要自己确认它的正确性。

这里有一个明显的安全问题。你会希望任何人都能用这些正确的网址来部署和卸除服务吗？估计不会。那你就需要使用主机寄存技术的安全功能。

在 GLUE 中部署

在 GLUE 中，我们可以使用同一个 Java 类来实现 `CallCounterService`，因为和 Apache SOAP 一样，GLUE 也不需要为服务编写任何特别的或额外的代码。然而，部署 GLUE 服务的过程和 Apache SOAP 有很大不同，而且它相对来说要简单一些。GLUE 提供了多个办法来部署服务。我们将编写一个单独的应用程序来部署

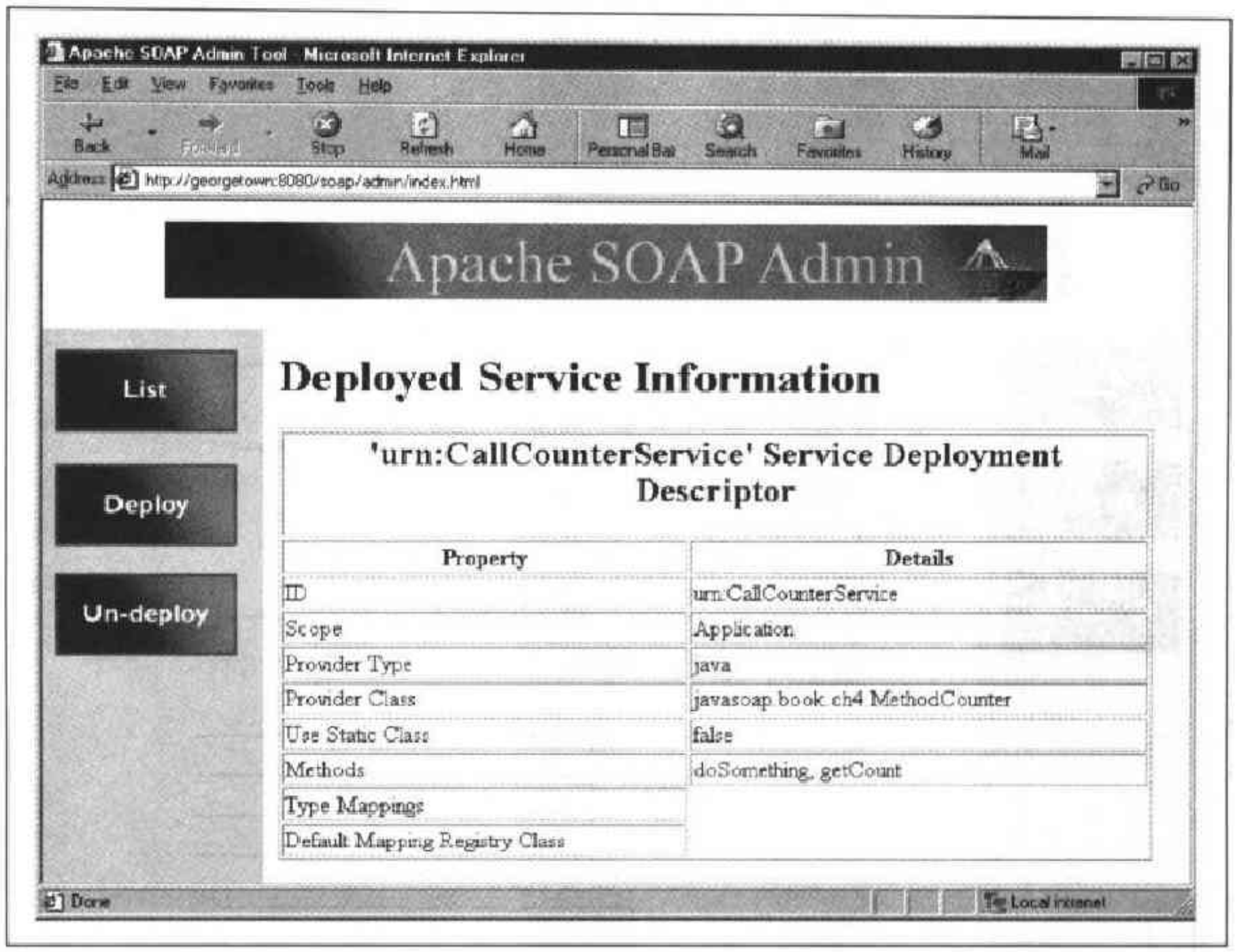

图 4-3: 服务细节信息

服务。这个应用程序将使用我们自己的代码来启动GLUE的内置HTTP服务器的一个实例，然后使用 GLUE 类和相关方法来发布服务。下面就是这个应用程序:

```
package javasoap.book.ch4;
import electric.util.Context;
import electric.registry.Registry;
import electric.server.http.HTTP;
public class CallCounterApp {
   public static void main( String[] args )
      throws Exception {

      HTTP.startup("http://georgetown:8004/glue");
      Context context = new Context();
      context.addProperty("activation", "application");
      Registry.publish( "urn:CallCounterService",
         javasoap.book.ch4.MethodCounter.class, context );
   }
}
```

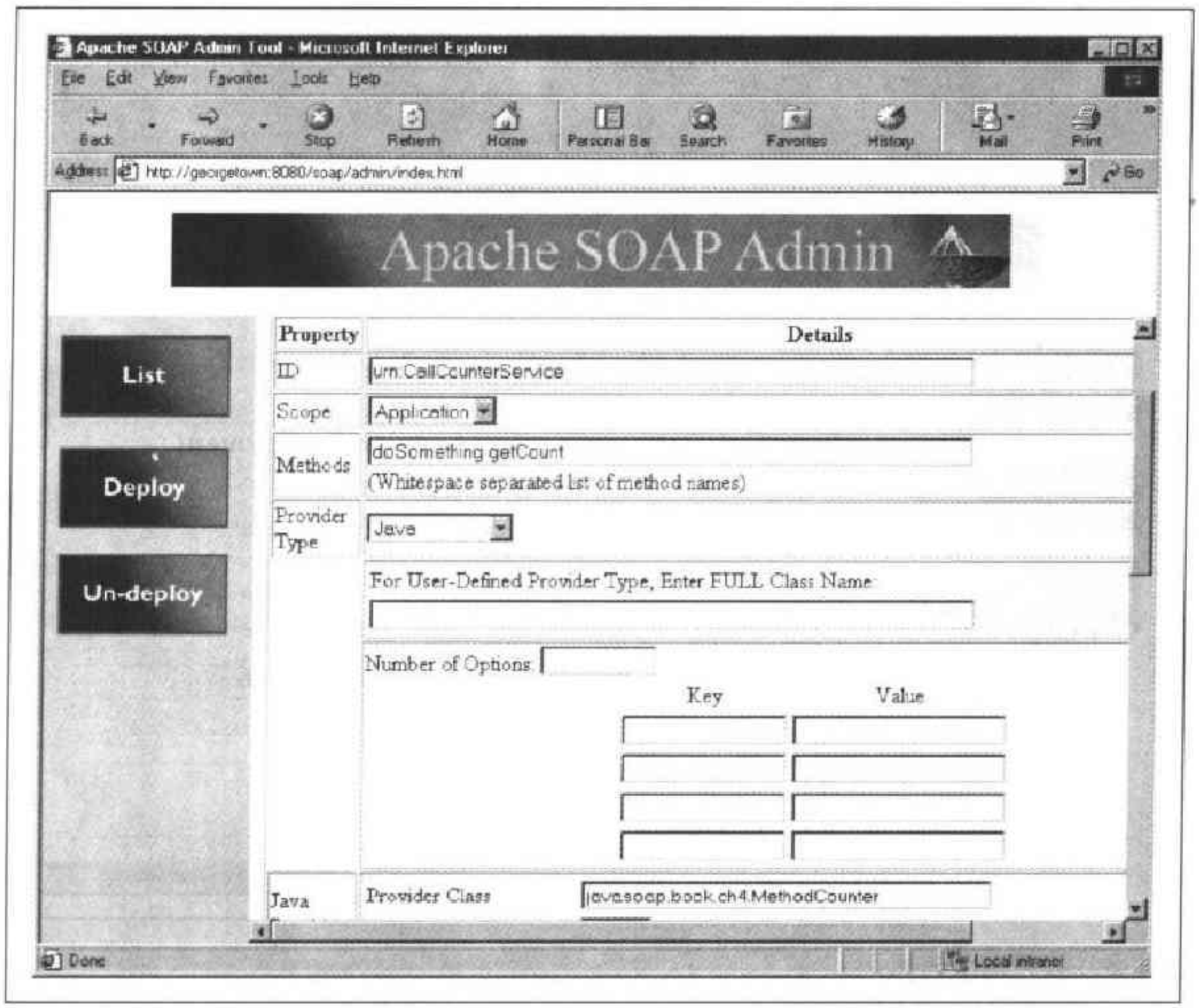

图 4-4：使用 SOAP 部署一个服务

CallCounterApp 类包含必需的静态 main()方法，因此它可以从命令行调用。HTTP.startup()方法在我的本地计算机georgetown的8004端口上启动了HTTP服务器的一个实例，用于接收和路由 SOAP 消息的资源是 /glue。把它们放到一起就得到了一个完整的URL: *http://georgetown:8004/glue*，它作为一个参数传给HTTP.startup()。

在启动了服务器之后，我们需要向它注册我们的服务。首先要指定我们要用的激活期方式。因此，我们创建了类electric.util.Context的一个实例。这个类是GLUE API的一部分，它用于容纳属性名和相关值的一个集合。为了记录激活期模型，我们调用了context对象的addProperty()方法，传给它属性名activation和与之关联的值application。最后，我们将使用GLUE的electric.reg-

istry.Registry类来发布服务。传入的第一个参数是urn:CallCounterClass，这又是要被发布的服务的名字。第二个参数是实现服务的Java类Method-Counter。context对象作为最后一个参数被传入，表明我们想要使用应用程序级服务激活期模型来发布（部署）服务。

为了启动这个应用程序，我们需要在命令行提示符后输入如下命令：

```
java javasoap.book.ch4.CallCounterApp
```

当我们启动应用程序时，会看见一些输出，告诉我们GLUE版本号和监听SOAP消息的GLUE资源的完整URL。这个URL与传给HTTP.startup()方法的地址是一样的。现在，urn:CallCounterService服务已被部署，并已准备就绪以接收SOAP RPC方法调用。

GLUE是怎么知道服务暴露了哪些方法的呢？在Apache SOAP部署描述文件中我们明确地定义了被暴露的方法；而我们在GLUE中却没有做类似的事情。相反，GLUE使用了一些技术来自动决定这些方法。因此，部署一个服务的最简单的方法是让GLUE自己来决定暴露哪些方法。在以后的章节中，我们将看到如何自己来控制要暴露的方法。对于这个例子，GLUE暴露了MethodCounter中除构造函数以外所有的公共方法。GLUE需要一个不带参数的构造函数，对此你不应该感到惊讶，这与Apache SOAP是一样的。

GLUE 控制台

GLUE包括一个基于Web的界面，它可以用来查看某个给定服务器上可用的服务，并可以编辑它们，甚至部署新的服务。GLUE控制台是一个特别的Web应用程序，可以使用GLUE安装目录下的*/bin*目录里的*console.bat*（或*console.sh*）脚本来启动它。因为console也是一个Windows命令，所以如果你正在使用微软公司Windows环境下的命令行，我建议先进入该文件的目录再运行console指令。如果你先进入了GLUE的*/bin*目录，你将能启动正确的程序。

默认情况下，GLUE控制台在本地计算机的8100端口上监听HTTP连接（也就是

http://localhost:8100)。如果要运行另外一个端口上的控制台应用程序，就要为console命令指定端口号作为参数。下面的指令启动8150端口上的控制台:

```
console 8150
```

本书中的所有例子均使用默认的8100端口。现在我们来启动控制台应用程序，运行你的Web浏览器，打开*http://localhost:8100*。你将看到如图4-5所示的GLUE控制台窗口。

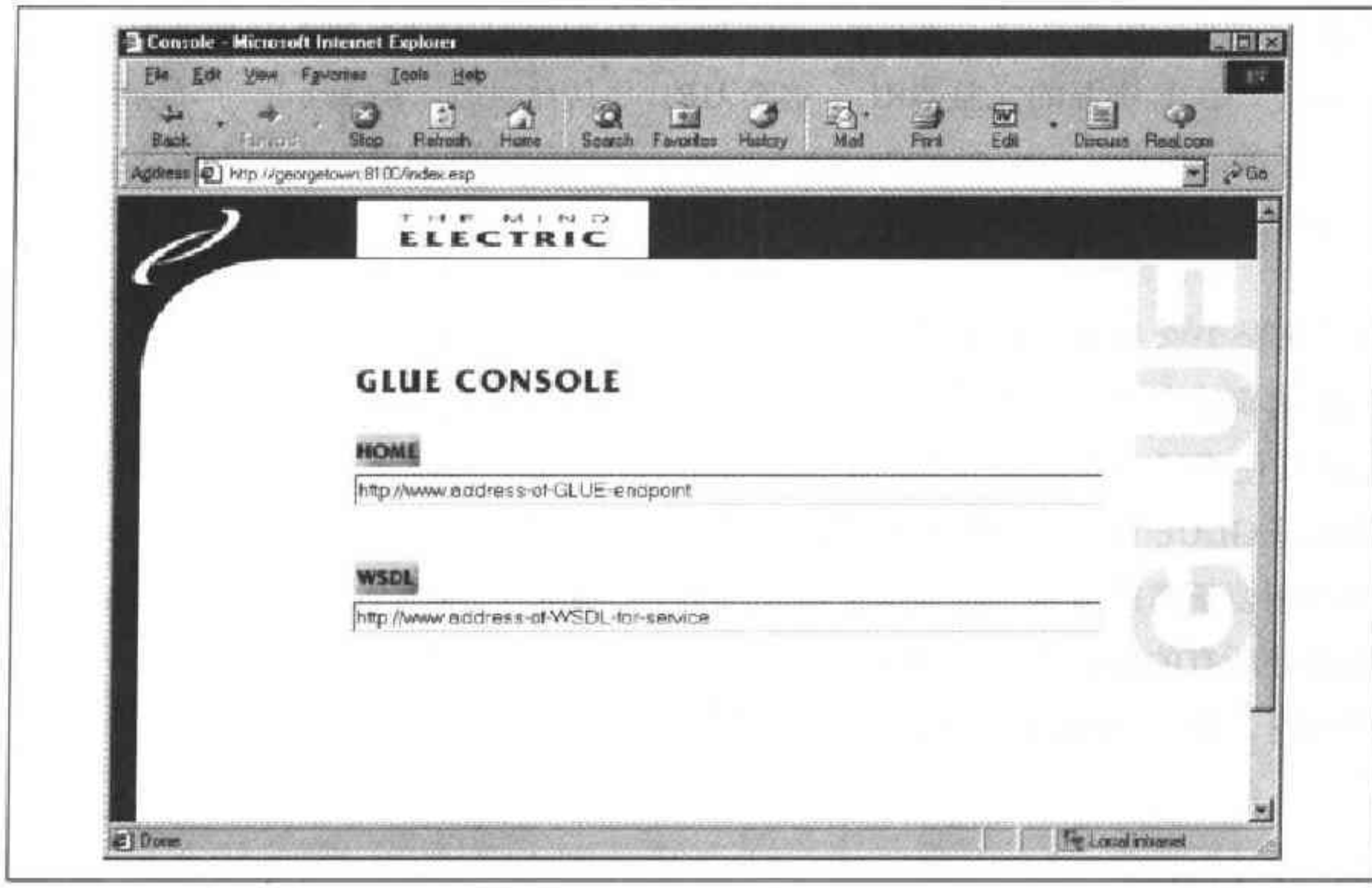

图4-5: GLUE控制台

现在，我们将只使用GLUE控制台界面上的第一栏，它被标示为HOME。记住我们已经使用*http://georgetown:8004/glue*这个URL启动了GLUE服务器的一个实例。在HOME一栏里输入该URL，单击HOME按钮，就会进入如图4-6所示的页面。这个页面包括服务器的端点URL和管理SOAP消息的资源。它还列出了目前服务器上所有可用的服务。第一个服务system/admin是由GLUE使用的特殊服务。第二个服务是urn:CallCounterService，它是我们在前面部署的。urn:CallCounterService链接将把你带到一个描述Web服务的页面，如图4-7所示。

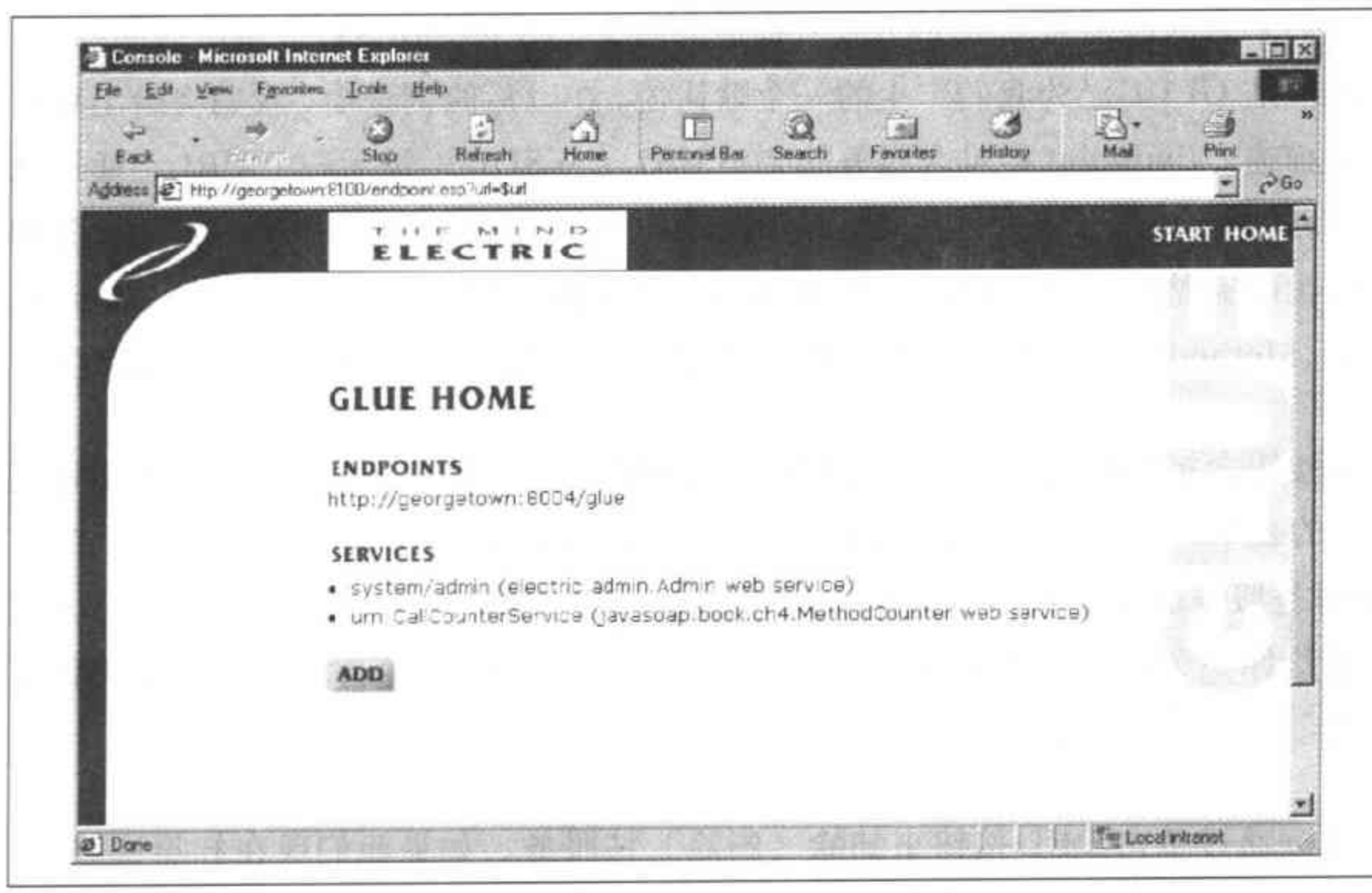

图 4-6: GLUE 主页

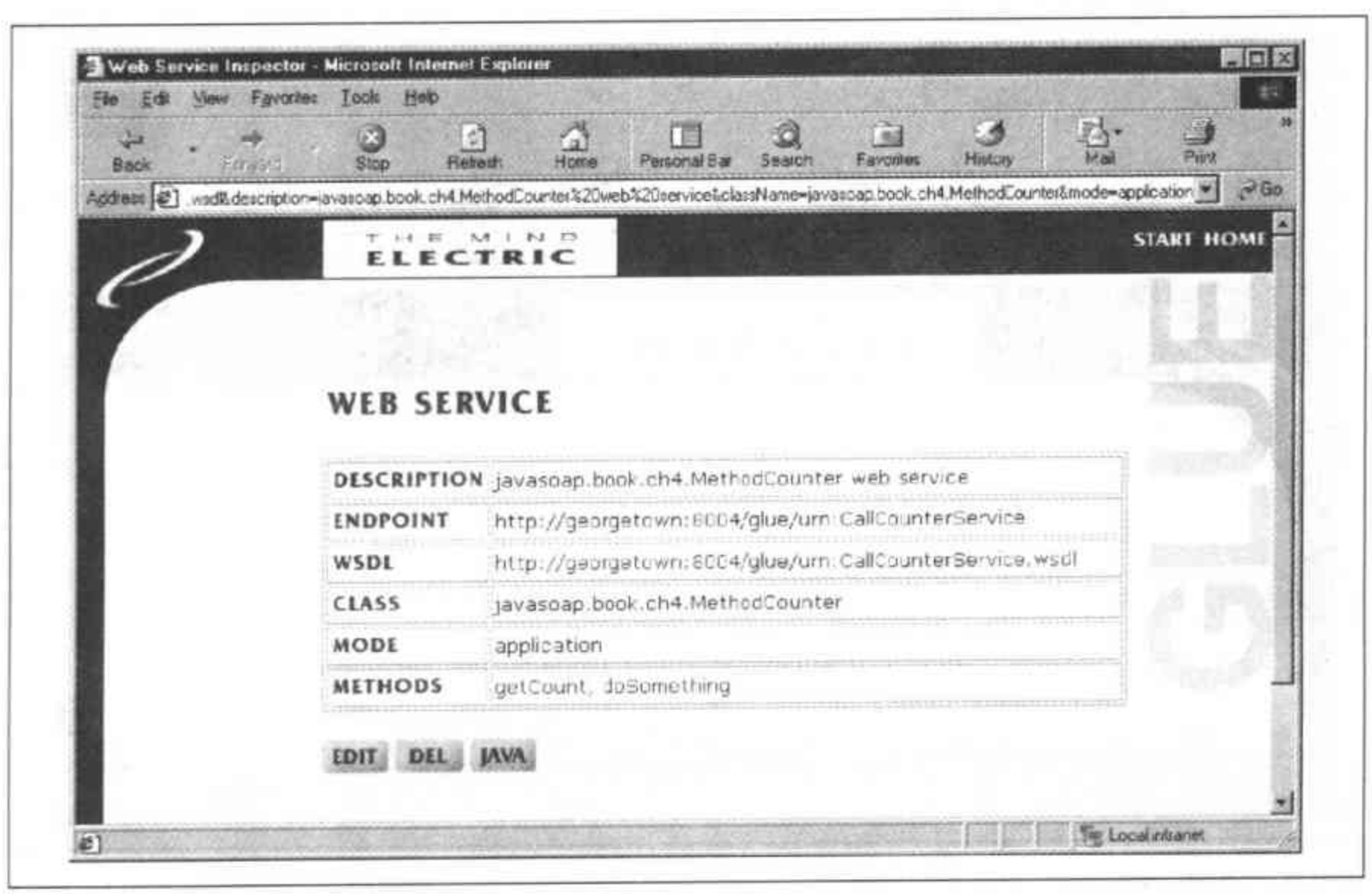

图 4-7: Web 服务信息

让我们来看看需要为一个Web服务提供的信息。Description栏提供服务的描述，它的值是 GLUE已为我们产生的一个默认值，在以后的例子中，我们将给出更有用的描述。Endpoint栏包含服务的完整URL。WSDL栏显示一个URL，通过它可以获取服务的WSDL，我们将在第九章中介绍这些。Class栏显示实现服务的Java类的完整类名称。而Mode栏显示本服务的激活期模型使用的是application。最后，Method栏包含能在这个服务上调用的方法。这些都是由GLUE自动设定的。

Edit按钮会把你带到一个允许你编辑服务的描述、Java类和模式（服务激活期模型）的表单。它也允许你修改URN（Universal Resource Name，统一资源名称）。URN是服务的实际名称，在本例中是urn:CallCounterService。单击JAVA按钮将给你一个能用来访问服务的客户端Java类的列表。我们将在“用GLUE调用服务”一节中介绍它。

你可以通过单击DEL按钮来卸除（删除）该服务。如果我们现在卸除该服务，GLUE控制台将用如图4-8所示的页面作为响应。

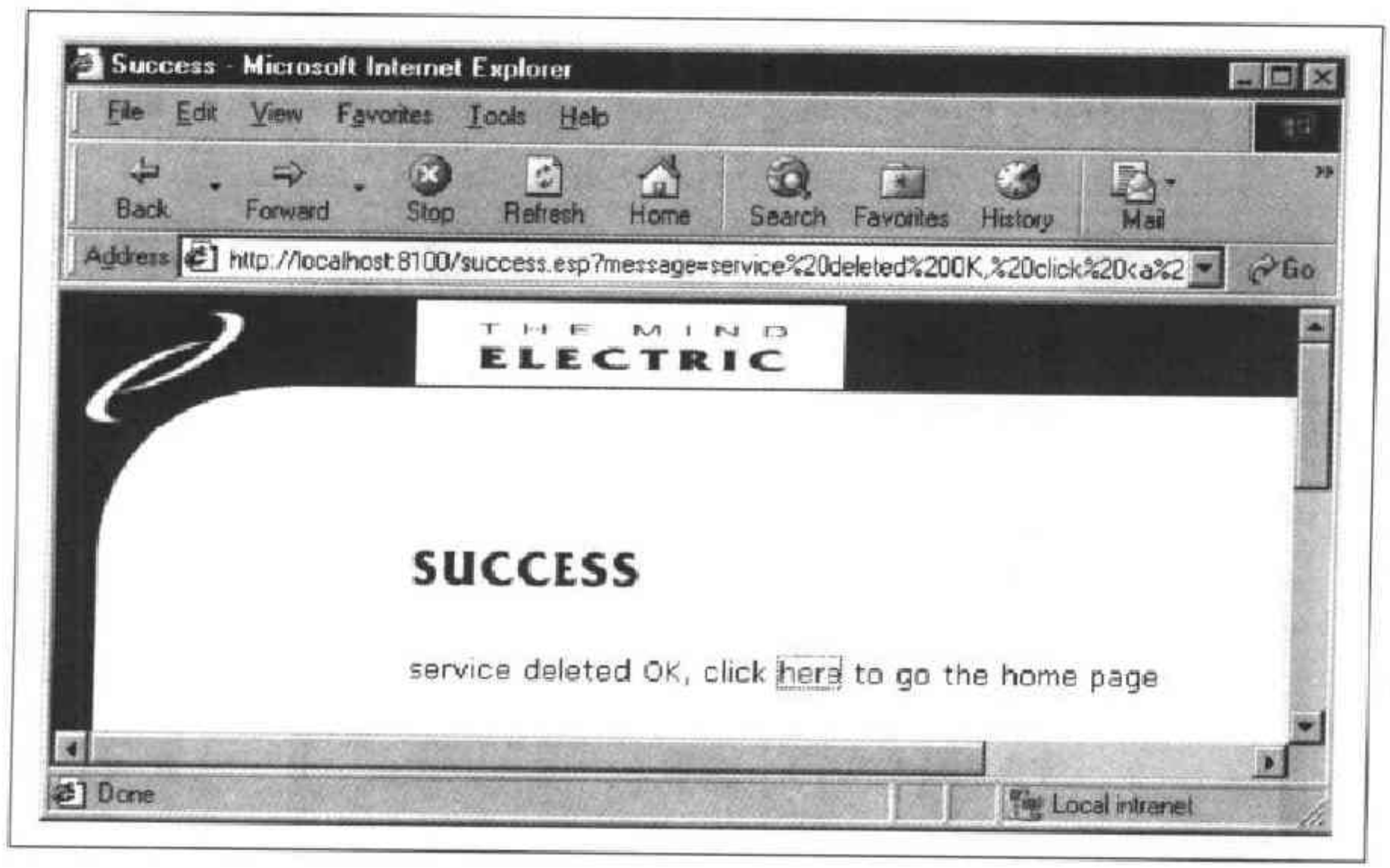

图4-8：成功的卸除

现在，单击“here”链接回到列出了服务器上可用 Web 服务的页面。它与图 4-6 看起来差不多，只不过urn:CallCounterService将不再被列出。现在单击ADD按钮，它会带你到一个表单，在那里你将被要求提供部署服务所需要的信息（图 4-9）。我已为urn:CallCounterService服务输入了信息，这次给出的是一个比较好的描述。我使用了与在 Java 应用程序中部署该服务时相同的URN、类和模式。

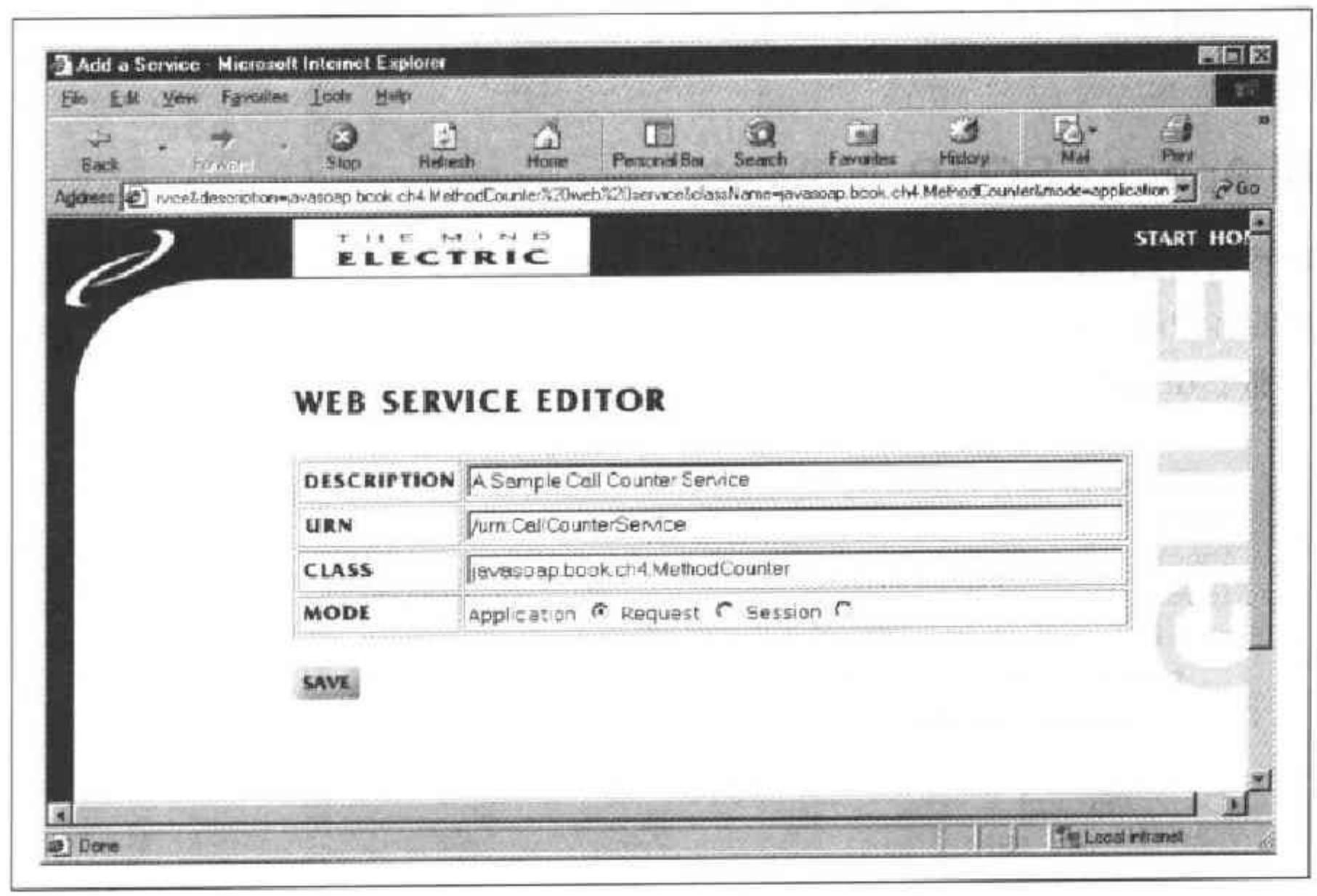

图 4-9: 通过控制台增加一个服务

按SAVE按钮部署服务。如果你提供了一个不正确的类名，GLUE控制台将给出一个指示添加失败的页面。否则你会被带到该 Web 服务的页面（图 4-7）。

编写服务客户

在以下几节里，我们将分别在 Apache SOAP 和 GLUE 中编写一些 Java 代码来调用我们已在服务中暴露的那些方法。现在，我们将不混合使用这些技术 —— 我

们将使用Apache SOAP API来调用一个Apache SOAP服务器，而使用GLUE API来调用一个 GLUE 服务器。

那么互操作性怎么样呢？SOAP作为传输级协议的最重要的方面之一就是你选择哪一个实现并不会妨碍它与其他 SOAP 实现正确的交流。然而，SOAP的互操作性方面仍旧有一些问题。有些技术比其他的好，而且在某些情况下你还需要克服一些困难才能让不同的SOAP技术彼此通信。因此让我们把这个话题推迟到第九章再去讨论，我们将在那里详细地介绍这些问题。

用 Apache SOAP 调用服务

既然现在已经让一个服务运行起来了，那么就让我们编写一些Java代码来作为服务的客户端。下面是一个简单的Java 应用程序，它调用了 Apache SOAP 服务器上那个 urn:CallCounterService服务的 getCount 方法：

```
package javasoap.book.ch4;
import java.net.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
public class GetCountApp {
   public static void main(String[] args)
      throws Exception {

      URL url =
        new URL(
          "http://georgetown:8080/soap/servlet/rpcrouter");

      Call call = new Call();
      call.setTargetObjectURI("urn:CallCounterService");
      call.setMethodName("getCount");
      try {
         Response resp = call.invoke(url, "");
         Parameter ret = resp.getReturnValue();
         Object value = ret.getValue();
         System.out.println("Result is " + value);
      }
      catch (SOAPException e) {
         System.err.println("Caught SOAPException (" +
                            e.getFaultCode() + "): " +
                            e.getMessage());
```

```
        }
    }
}
```

这个程序首先使用Apache SOAP服务器上Apache RPC路由器的URL创建了类java.net.URL的一个实例。然后它创建了类org.apache.soap.rpc.Call的一个实例，这个Java类封装了一个SOAP RPC方法调用。call对象的setTargetObjectURI()方法用来指定服务名，而setMethodName()方法指定将被调用的方法的名称。

现在，我们已经准备好要调用指定服务的方法了。既然方法调用可以抛出一个SOAPException异常，那么我们就把它放到一个try/catch块中。如果出现了什么错误，异常的错误代码和消息将被打印出来。

为了调用该方法，我们使用了call对象的invoke()方法，并以前面建立的URL作为第一个参数。第二个参数是一个SOAP行为URI，它被当作一个HTTP头项。我们没有使用SOAP行为，所以对这个参数可以使用一个空字符串。invoke()方法返回org.apache.soap.rpc.Response的一个实例；如果成功返回，这个对象中将包含来自服务器的响应。现在让我们假设调用是成功的。在调用该方法之后，我们还调用了响应对象的getReturnValue()方法。这个方法返回org.apache.soap.rpc.Parameter的一个实例，它包含服务方法调用的返回值或响应。现在我们调用刚才返回的对象的getValue()方法，该方法返回Object的一个实例；你得到的这个对象的实际类型将依赖于该方法返回值的类型。这一点很重要。Apache SOAP的RPC方法调用返回Java对象的实例，而不是简单类型的实例。比如说，被调用的服务方法返回一个int值，这个简单返回值将被作为相应Java对象类型的一个实例，在本例中是Integer。我们可以将返回值的类型转换为Integer，但是我们不这样做，我们让它作为一个Object实例并将结果用System.out.println()打印出来。这个方法隐式地调用了toString()方法，以字符串形式返回了这个整数的值。

接下来让我们运行这个应用程序：

```
java javasoap.book.ch4.GetCountApp
```

如果一切正常，你将看见下面这样的输出：

```
Result is 0
```

结果为0是因为服务对象的doSomething()方法在运行时会将计数器加1，而我们到现在为止还没有对这个方法做任何调用。很不错吧！我们已经成功地部署并在Java中调用了一个SOAP服务。让我们来看看用来调用该服务的SOAP有效负载：

```
POST /soap/servlet/rpcrouter HTTP/1.0
Host: Georgetown
Content-Type: text/xml; charset=utf-8
Content-Length: 344
SOAPAction: ""

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<SOAP-ENV:Body>
  <ns1:getCount xmlns:ns1="urn:CallCounterService">
</ns1:getCount>

</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

对于这个SOAP文档中的所有数据，你都应该很熟悉。特定于这个例子的信息都包含在SOAP体中，在那里可以看到getCount元素（用来表示被调用的方法），它被urn:CallCounterService名称空间限定，这个名称空间是被调用的方法所属的那个服务的名称。现在让我们来看服务器返回的响应：

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 468
Set-Cookie2: JSESSIONID=yqf5f6su31;Version=1;Discard;Path="/soap"
Set-Cookie: JSESSIONID=yqf5f6su31;Path=/soap
Servlet-Engine: Tomcat Web Server/3.2.3

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
```

```
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<SOAP-ENV:Body>
 <ns1:getCountResponse
  xmlns:ns1="urn:CallCounterService"
  SOAP-ENV:encodingStyle=
    "http://schemas.xmlsoap.org/soap/encoding/">
   <return xsi:type="xsd:int">0</return>
 </ns1:getCountResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

我们把HTTP头也显示出来了，这样能看到整个响应。注意服务器返回了HTTP的cookie值。虽然我们现在用不到它们，但如果我们的服务使用了会话级服务激活期，那些cookie将会是很有用的。

现在来看SOAP体。既然被调用的方法名称是getCount ，体元素的第一个子元素就是getCountResponse，它以服务名称作为名称空间限制。SOAP-ENV:encodingStyle属性被设置为 http://schemas.xmlsoap.org/soap/encoding。在getCountResponse元素里，我们会发现只有一个名为return的子元素，它包含了getCount方法的返回值。return的类型被显式地声明为xsd:int，并且其数据值为0。

让我们对doSomething方法做一些调用，以使计数器有所变化：

```
package javasoap.book.ch4;
import java.net.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
public class GetCountApp2 {
   public static void main(String[] args)
      throws Exception {

      URL url =
        new URL(
          "http://georgetown:8080/soap/servlet/rpcrouter");

      Call call = new Call();
      call.setTargetObjectURI("urn:CallCounterService");
```

```
        try {
            call.setMethodName("doSomething");
            Response resp = call.invoke(url, "");
            resp = call.invoke(url, "");
            resp = call.invoke(url, "");
            call.setMethodName("getCount");
            resp = call.invoke(url, "");
            Parameter ret = resp.getReturnValue();
            Object value = ret.getValue();
            System.out.println("Result is " + value);
        }
        catch (SOAPException e) {
            System.err.println("Caught SOAPException (" +
                               e.getFaultCode() + "): " +
                               e.getMessage());
        }
    }
}
```

在这个例子中，我们首先用setMethodName()将方法的名称设置为doSomething，然后我们运行call.invoke()三次。接着将方法名设成getCount，并再次调用服务。和上一次一样，getCount方法的结果被显示出来。在从命令行运行了这个应用程序之后，应该会看见如下的输出：

```
Result is 3
```

我们已经部署了urn:CallCounterService，它使用的是应用程序作用域，这意味着对于每一个服务请求都将使用服务类的单个实例。因此每一次doSomething方法被调用的时候，MethodCounter实例中的_counter变量都会加1。如果再一次运行应用程序，结果将会是6。第三次它将会是9。现在你看到了应用程序作用域的应用，尽管在这个例子里它显得很平常。如果想要将计数器清零，那么关闭用来运行Apache SOAP的服务器并重启它。

注意： 你可能会认为卸除服务然后再部署它的方法可以代替关闭服务器。但是，这样做不会起作用，至少在Tomcat服务器上是这样。显然实现服务类被存入了缓存并在服务重新部署时被再次使用。如果你正在使用一些其他的服务器来运行Apache SOAP，你应该自己测试一下。

用 GLUE 调用服务

现在让我们使用 GLUE API 来创建一个应用程序，并让它与我们前面建立的 GLUE 服务器通信。GLUE 采用了与 Java 客户编程不同的方式。它允许你经由本地的 Java 接口访问服务。这些接口包含与服务所暴露的方法对应的方法。在这个例子中，我们要编写一个包括 getCount()方法和 doSomething()方法的 Java 接口，就像在 MethodCounter 中那样。下面就是一个这样的接口，它被命名为 ICallCounterService（注 5）。

```
package javasoap.book.ch4;
public interface ICallCounterService {
  int getCount();
  boolean doSomething();
}
```

下一步让我们创建一个用来访问发布在 GLUE 上的服务 urn:CallCounter-Service 的应用程序。

```
package javasoap.book.ch4;
import electric.registry.Registry;
import electric.registry.RegistryException;
public class GetCountApp3 {
   public static void main(String[] args) throws Exception
   {
      try {
         ICallCounterService ctr =
           (ICallCounterService)Registry.bind(
             "http://georgetown:8004/glue/urn:CallCounterService.wsdl",
                 ICallCounterService.class);
         System.out.println("Result is " + ctr.getCount());
      }
      catch (RegistryException e)
      {
         System.out.println(e);
      }
   }
}
```

GLUE 允许我们将服务接口绑定到服务。一旦完成了绑定，我们就可以把它当作（它也确实就是）本地对象来简单地使用。因此我们惟一要做的一步就是调用

注 5： 接口名通常以一个“I”开头，但这不是惟一的惯例。

electric.registry.Registry类的静态方法bind()。第一个参数是用来描述服务的WSDL文件的URL。GLUE是基于WSDL的，而且它自动为服务产生WSDL描述（第九章将对WSDL做简要介绍）。为建立适当的URL，需要以服务器和资源地址（*http://georgetown:8004/glue*）开头，再加上服务的名称，然后再加上*.wsdl*。第二个参数是用来表示接口的Java类ICallCounterService.class。Registry.bind()方法返回值的类型将被转换为服务ICallCounterService的Java接口的一个实例。

在对Registry.bind()的调用返回之后，我们就能使用ctr变量来调用getCount()和doSomething()方法，这与它是本地Java对象没什么两样。在这个例子中，我们只是简单地打印出ctr.getCount()的调用结果。应该这样运行这个例子：

```
java javasoap.book.ch4.GetCountApp3
```

getCount()调用的结果是0，这是因为我们还没有对doSomething()做任何调用：

```
Result is 0
```

GLUE控制台能产生一些Java代码，这会使访问服务更加容易。要产生这些代码，使用你的Web浏览器回到GLUE控制台，然后进入urn:CallCounterService并单击JAVA按钮。GLUE将使用一个与我所用的不同的接口名称（IMethodCounter），但是接口的内容却是相同的。辅助类（helper class）被叫做MethodCounterHelper。它封装了对Registry.bind()的调用和用于传递正确接口的类型转换。辅助类提供了两个静态的bind()方法。一个没有带参数并自动生成了服务端点的URL。另一个bind()方法允许你指定一个其他的URL，如果你喜欢的话。这里对前面的客户应用程序做了修改，它使用了控制台生成的Java接口和辅助类。请确保你为接口和辅助类使用了适当的包名。GLUE控制台不做这个工作，因此这是你的事情（注6）。

注6：GLUE提供了一个工具wsdl2java，它可以根据WSDL文件生成Java代码，并且有一个为指定的包产生代码的选项。我们将在第九章介绍这个工具。

```
package javasoap.book.ch4;
import electric.registry.RegistryException;
public class GetCountApp4 {
   public static void main(String[] args) throws Exception
   {
      try {
       IMethodCounter ctr = MethodCounterHelper.bind();
       System.out.println("Result is " + ctr.getCount());
    }
    catch (RegistryException e)
    {
       System.out.println(e);
    }
   }
}
```

使用辅助类MethodCounterHelper可以使代码更整洁。对于这样一个小程序，它不会有很大不同，但是在一个大的应用程序中这样做就会有明显的效果。编译新的客户程序、Java 接口和辅助类，并把它们作为 javasoap.book.ch4 包的一部分。从命令行运行这个例子会得到与前面一样的结果。

GLUE控制台还提供了一个不用写任何代码就能调用服务的方法。将你的浏览器转回到 urn:CallCounterService Web服务页面。在METHODS栏中列出的方法都是超链接；单击这些链接中的一个，你会得到一个表单，从中你可以调用刚才选择的服务的方法。如果你选择了 getCount 方法，你将看到如图 4-10 所示的表单。其中有输入参数值和返回值的栏，分别标示为INPUTS和OUTPUTS。如果 getCount 方法有任何参数，就会有地方让你输入数据。不过在本例中是没有的。单击SEND按钮来调用该服务方法。当调用返回时，结果将被显示在SEND按钮下面；如果你现在就这样做，你将会看到一个0。这个Web界面为检查Web服务的功能提供了一个不必写Java代码的有用工具。

让我们修改这个GLUE例子，在调用getCount之前先调用几次 doSomething方法。这是我们客户程序的一个新版本，它使用GLUE API来进行多次调用，而且使用了由 GLUE 控制台产生的接口和辅助类:

```
package javasoap.book.ch4;
import electric.registry.RegistryException;
public class DoSomethingApp {
```

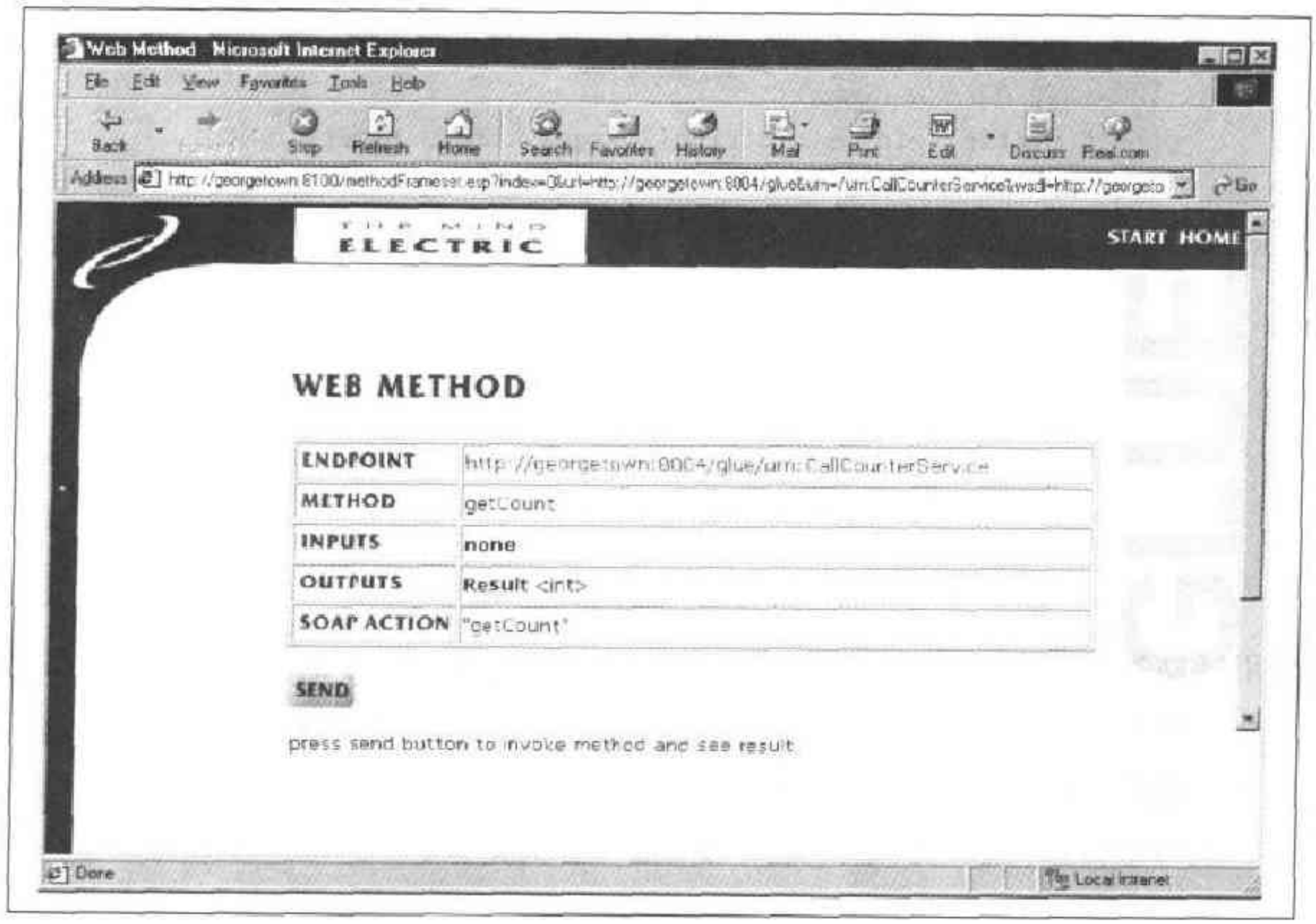

图 4-10: 在浏览器中调用 getCount

```
public static void main(String[] args) throws Exception
{
    try {
     IMethodCounter ctr = MethodCounterHelper.bind();
     ctr.doSomething();
     ctr.doSomething();
     ctr.doSomething();
     System.out.println("Result is " + ctr.getCount());
    }
    catch (RegistryException e)
    {
     System.out.println(e);
    }
  }
}
```

当你第一次运行这个客户程序的时候，会得到如下输出:

```
Result is 3
```

如果你再次运行它，结果将会是6，以此类推，这是因为服务使用了应用程序作用域（激活期模式）。要使计数器重置为0，可以停止CallCounterApp应用程序然后再重新运行它，或者在GLUE控制台中先将该服务卸除然后再把它添加回来。对于GLUE控制台，后一种方式是可行的。

使用请求级作用域进行部署

直到现在，我们所建立的所有例子在部署时都使用了应用程序级作用域（服务激活期）。在这个模式中，由实现服务类的同一个实例来处理所有的方法调用。现在我们来将服务的部署改为请求级作用域。在请求级作用域中，每产生一个方法调用，就会新建一个服务对象的实例，当这个调用完成后，该对象也将被销毁。

首先我们要通过修改部署描述文件来改造在Apache SOAP下运行的那个例子。惟一要做的改变是将provider元素的scope属性改为Request；部署描述文件中的其他部分都保持原样：

```
<isd:service
   xmlns:isd="http://xml.apache.org/xml-soap/deployment"
   id="urn:CallCounterService">

   <isd:provider type="java"
         scope="Request"
         methods="doSomething getCount">
     <isd:java class="javasoap.book.ch4.MethodCounter"
         static="false"/>
   </isd:provider>
   <isd:faultListener>
   org.apache.soap.server.DOMFaultListener
   </isd:faultListener>
   <isd:mappings>
   </isd:mappings>
</isd:service>
```

记得要先卸除urn:CallCounterService然后再重新部署它，你可以使用前面介绍的任何一种技术。你不必使用我们前面介绍的关闭服务器的方式；因为我们现在使用的是请求级作用域，所以在内存中不会再有MethodCounter的任何实例。因此你可以使用服务管理程序客户的命令行方式来卸除和部署该服务，或者使用

Apache SOAP Admin 浏览器应用程序来做这个工作。如果你决定关闭服务器然后再重新启动它，你仍然需要卸除和重新部署你的服务，因为 Apache SOAP 永久地保留了已部署的服务。

既然你已经使用请求级作用域重新部署了urn:CallCounterService，你就可以使用 Apache 版本的客户应用程序来访问该服务了。即使应用程序在调用getCount()之前调用了doSomething()方法很多次，其结果将总是0。这是请求级作用域的直接结果：每当这些方法中的任何一个被调用时，都会产生MethodCounter类的一个新实例，而且那个实例会在方法调用完成时被销毁。调用之间不保留任何状态。因此，计数器的值只能是 0。

让我们在 GLUE 下对这个服务做同样的事情。要完成这个工作最容易的办法是使用GLUE控制台。浏览urn:CallCounterService服务的Web服务页面，然后单击 EDIT 按钮。这将带你到如图 4-11 所示的 Web 服务编辑器页面。

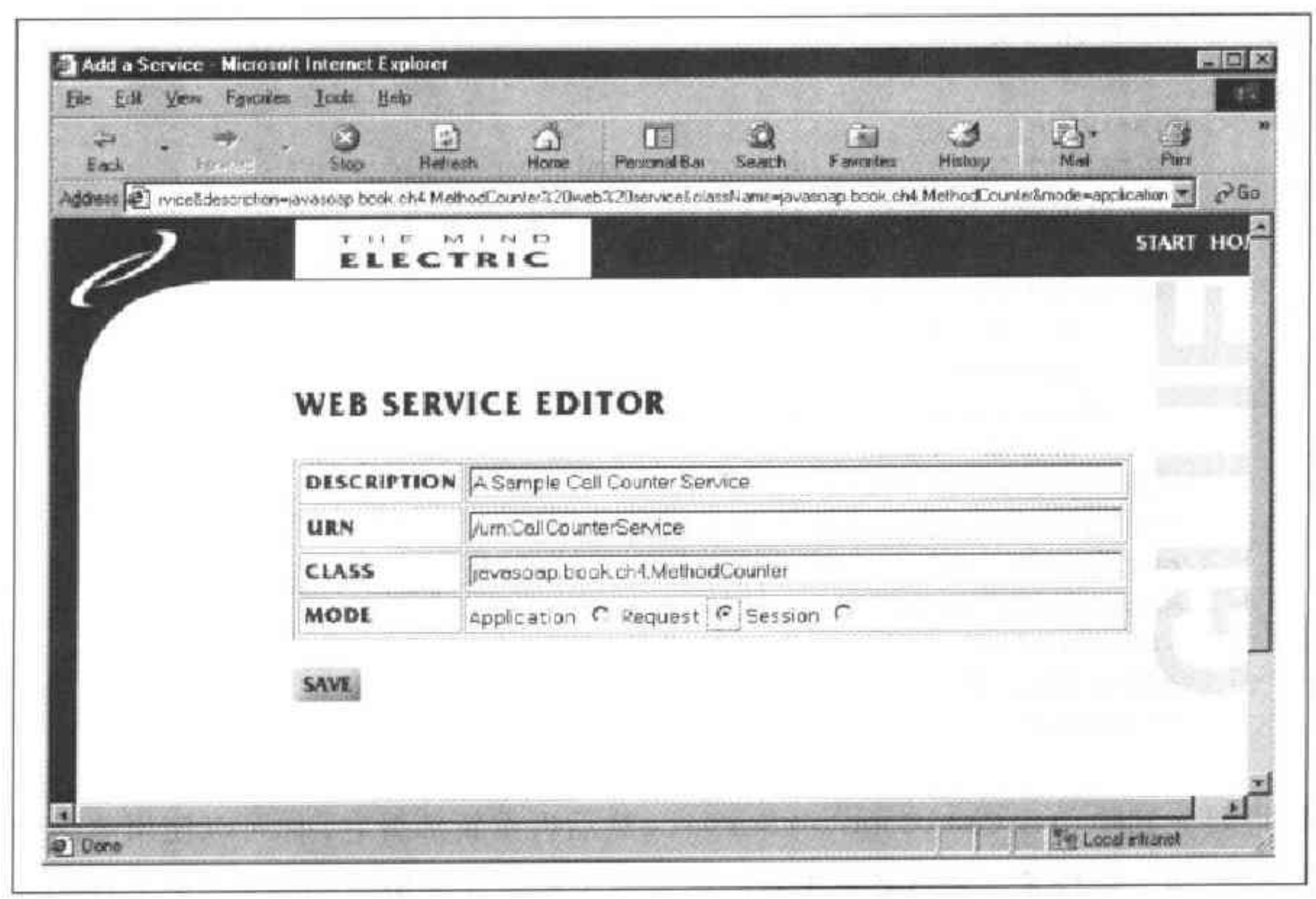

图 4-11: Web 服务编辑器

要改变激活期模式，可以在MODE栏中单击Request对应的单选按钮。现在单击SAVE按钮，此时服务就被重新部署为使用请求级作用域。

注意： 我对服务激活期模型使用了不同的术语。有时我称它为作用域，有时称它为激活期模式，其他时候我称它为服务激活期模型。它们全都意味着同一个事物，但是不同的实现使用了不同的术语，而我希望你能习惯于所有这些词。用词可能是不同的，但是它们依然都表示实现服务的对象的生存期。

你可以再次运行DoSomethingApp应用程序，以检查现在该服务是否运行于请求级作用域。你将总是得到结果0。再一次说明，对doSomething和getCount的调用都发生在MethodCounter类的彼此独立的实例上。

使用会话级作用域进行部署

Apache SOAP通过在HTTP头中传递cookie来对会话提供管理。在本章前面的部分，我们已经看到一个来自服务器的包含了Set-Cookie和Set-Cookie2头项的HTTP响应。客户应用程序在需要时可以利用这些cookie来使本次方法调用也使用前面请求的会话。 Apache SOAP API使用了一种简单的技术来实现这个目的。org.apache.soap.transport.http.SOAPHTTPConnection包含了一个只带有一个布尔类型参数的setMaintainSession()方法。当这个参数为真时，就通知连接对象保持当前的会话。当下一个方法调用发生的时候，连接对象通过了解cookie中的信息，并将它们传回到服务器来实现对会话的保持。

让我们来编辑部署描述文件，将scope属性设为Session。现在再一次运行客户程序。你将得到如下输出：

```
Result is 3
```

这个输出证明了服务对象对所有的方法调用都使用了同一个实例。如果再运行一次，仍会得到相同的结果，因为在你第一次调用doSomething()的时候会话对象的一个新实例就建立了，而且后来的每一次方法调用都使用了相同的服务对象。因为我们现在对每一个服务调用都使用了org.apache.soap.rpc.Call的同一个

实例，因此我们也使用org.apache.soap.transport.http. SOAPHTTP-Connection的同一个实例。你或许已经猜到了，org.apache.soap.transport.http.SOAPHTTPConnection的默认行为是保持会话。

使用会话级作用域的一个常见例子是购物车，这时你需要在对服务方法的各次调用之间跟踪会话。然而，你仍然会有不希望所有的调用都使用同一个会话的时候，尽管服务被部署为使用会话作用域。在下一个例子中，在第一次方法调用之后，SOAPHTTPConnection对象被Call对象返回。接下来对连接的一次setMaintain-Session()方法调用使用了false作为参数。这就阻止了连接对所有的后继服务方法调用都使用同一个会话。

```
package javasoap.book.ch4;
import java.net.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.transport.http.*;
public class SessionApp {
   public static void main(String[] args)
      throws Exception {

      URL url =
        new URL(
          "http://georgetown:8080/soap/servlet/rpcrouter");

      Call call = new Call();
      call.setTargetObjectURI("urn:CallCounterService");
      try {
         call.setMethodName("doSomething");
         Response resp = call.invoke(url, "");
         SOAPHTTPConnection st =
          (SOAPHTTPConnection)call.getSOAPTransport();
         st.setMaintainSession(false);
         resp = call.invoke(url, "");
         st.setMaintainSession(true);
         resp = call.invoke(url, "");
         call.setMethodName("getCount");
         resp = call.invoke(url, "");
         Parameter ret = resp.getReturnValue();
         Object value = ret.getValue();
         System.out.println("Result is " + value);
      }
      catch (SOAPException e) {
```

```
            System.err.println("Caught SOAPException (" +
                               e.getFaultCode() + "): " +
                               e.getMessage());
        }
    }
}
```

在对 doSomething()进行第三调用之前，我们再一次调用了 setMaintainSession()方法，这次使用 true 为参数。因此，在第三次调用 doSomething()之后，SOAPHTTPConnection 对象将取得会话 cookie。然后，当我们调用 getCount()的时候，那些 cookie 被回传至服务器，以保证相同的会话对象被再一次使用。当你运行 SessionApp 应用程序时，应该会得到如下输出：

```
Result is 1
```

如果想要为多个调用保持同一个会话，就要在作为该会话一部分的第一次调用之前建立会话。然而，这样编程显得有些臃肿，至少对于我来说是如此。当我开始使用同一个会话做调用的时候，我喜欢先创建SOAPHTTPConnection的一个实例，然后再使用它。当然，这些考虑只适用于部署为使用会话作用域的服务。如果你想做这样一个尝试，那就将 try 块里的代码改成下面这样：

```
call.setMethodName("doSomething");
Response resp = call.invoke(url, "");
resp = call.invoke(url, "");
SOAPHTTPConnection st = new SOAPHTTPConnection();
call.setSOAPTransport(st);
resp = call.invoke(url, "");
call.setMethodName("getCount");
resp = call.invoke(url, "");
Parameter ret = resp.getReturnValue();
Object value = ret.getValue();
System.out.println("Result is " + value);
```

服务器的会话生存期与具体的实现相关。你正在使用的服务器平台、它被配置的方式以及可用的配置选项都关系到会话生存期的持续时间。在大部分情况下，服务器在出现一段空闲时间之后就会终止会话。GLUE 自动管理会话，而且通过调用electric.server.http.HTTP.setSessionTimeout()让你设置会话的生存期。传给这个方法的参数是一个整型值，它包含一个会话可以持续处于不活跃状态的秒数，超过这个时间会话就会失效。

传递参数

到现在为止，本章中的例子都使用了不带参数的方法。现在我们花一小段时间来看看如何传递参数。让我们建立一个叫做urn:StockPriceService的服务，它由javasoap.book.ch4.StockPrice类实现。getPrice()方法使用股票代码和货币单位两个参数。该方法总是返回浮点值75.33——这不是一个非常有趣的股市，但至少它不会下跌。严肃点说，我们仅对如何传递参数感兴趣。访问一个数据库或者数据来源来得到股票价格这些细节的工作，在此就留给你去做吧。

```
package javasoap.book.ch4;
public class StockPrice {
   public float getPrice(String stock, String currency) {
      float result;
      // 确定股票的价格并按指定货币单位返回它
      result = (float)75.33;
      return result;
   }
}
```

我们将隐藏部署的细节，不管是Apache SOAP还是GLUE，因为我们已经在这个主题上花了很多时间了。这里是Apache SOAP中由StockPrice类实现的StockPriceService服务的部署描述文件。该服务使用应用程序级作用域，它只包含一个以指定货币单位返回股票当前价格的方法getPrice。

```
<isd:service
   xmlns:isd="http://xml.apache.org/xml-soap/deployment"
   id="urn:StockPriceService">

   <isd:provider type="java"
         scope="Application"
         methods="getPrice">
     <isd:java class="javasoap.book.ch4.StockPrice"
         static="false"/>
   </isd:provider>
   <isd:faultListener>
   org.apache.soap.server.DOMFaultListener
   </isd:faultListener>
   <isd:mappings>
   </isd:mappings>
</isd:service>
```

这个服务在Apache SOAP中可以使用基于浏览器的Admin工具或者通过将上面的部署描述文件放入一个文件然后使用服务管理程序客户来部署。我们将使用后面一种方式。包含部署描述文件的文件是*StockPriceService.dd*。通过这个文件，使用下面的指令将会在我的本地计算机georgetown上部署服务：

```
java org.apache.soap.server.ServiceManagerClient
  http://georgetown:8080/soap/servlet/rpcrouter deploy StockPriceService.dd
```

这里是一个使用Apache SOAP API来访问StockPriceService的简单应用程序：

```
package javasoap.book.ch4;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
public class StockPriceApp {
   public static void main(String[] args)
      throws Exception {

      URL url =
        new URL(
          "http://georgetown:8080/soap/servlet/rpcrouter");

      Call call = new Call();
      call.setTargetObjectURI("urn:StockPriceService");

      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      String stock = "MINDSTRM";
      String currency = "USD";
      Vector params = new Vector();
      params.addElement(new Parameter("stock",
                        String.class, stock, null));
      params.addElement(new Parameter("currency",
                        String.class, currency, null));
      call.setParams(params);

      try {
         call.setMethodName("getPrice");
         Response resp = call.invoke(url, "");
         Parameter ret = resp.getReturnValue();
         Object value = ret.getValue();
         System.out.println("Price is " + value);
      }
```

```
        catch (SOAPException e) {
            System.err.println("Caught SOAPException (" +
                               e.getFaultCode() + "): " +
                               e.getMessage());
        }
    }
}
```

这个应用程序的开头与我们在前面用 Apache SOAP API 建立的那些应用程序很相似。建立参数的代码首先调用了 Call 对象的 setEncodingStyleURI()方法。传给这个方法的参数是一个字符串，它包含了用于串行化参数的编码形式的URI。在本例中，编码形式由 Constants.NS_URI_SOAP_ENC 给出，它是一个在 org.apache.soap.Constants类中定义的常量。这个常量的值是"http://schemas.xmlsoap.org/soap/encoding/"，这是标准的SOAP编码名称空间。Call 实例中编码形式的默认值是 null，因此你一定要记得调用 setEncodingStyleURI()方法。不这样做，服务器端将不知道用来反串行化参数的编码方案。

接下来，我们建立了分别代表股票代码（注 7）和货币单位的两个 String 变量，当调用 getPrice()方法时会用到它们。现在我们来创建 java.util.Vector 的一个实例，它将包含这些参数。每个参数都存在于 org.apache.soap.rpc.Parameter的一个实例中。这个类的构造函数使用参数的名称、包含参数值的对象的类、包含参数值的对象以及参数编码形式的 URI 作为参数。对于 org.apache.soap.rpc.Parameter构造函数的第一个参数，我们传入了名称 stock，它是 StockPrice 类的 getPrice 方法的第一个参数的名称。第二个参数是 String.class，它告诉构造函数这个值是作为Java的 String实例传递的。第三个参数是一个 String变量，它包含了我们真正要传入的值："MINDSTRM"。我们传入 null 作为最后一个参数，表示编码形式 URI 不应该包含在用作串行化本参数的 XML 元素中。回忆一下，我们已经在调用 call.setEncodingStyleURI()时建立过用于整个调用过程的编程形式URI了；除非你想要为这个参数传入一个特殊的值来使用某种其他的编码形式，否则只需要传入 null。org.apache.soap.rpc.Parameter 实例被 params.addElement()方法添加到矢量中。对于 currency 也是同样的过程，也把它加入矢量 params。

注 7： MINDSTRM 不是真实的股票代码。

最后，我们将这个包含各参数的矢量通过call.setParams()方法传入Call对象。从这里开始，余下的代码都与前面的一样了。调用getPrice()服务方法，然后其结果被打印显示。这里是发送到服务器的SOAP封套:

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

<SOAP-ENV:Body>
  <ns1:getPrice
      xmlns:ns1="urn:StockPriceService"
      SOAP-ENV:encodingStyle=
         "http://schemas.xmlsoap.org/soap/encoding/">
      <stock xsi:type="xsd:string">MINDSTRM</stock>
      <currency xsi:type="xsd:string">USD</currency>
  </ns1:getPrice>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

正如我们以前曾经看到的，我们所调用的方法的名称被用作SOAP Body的子元素，名称空间由服务自身的名称限定。getPrice元素的SOAP-ENV:encodingStyle属性适用于它的全部子元素，除非那些元素包括了属于它们自己的SOAP-ENV:encodingStyle（这就是我们向org.apache.soap.rpc.Parameter的构造函数传入null的原因）。两个参数被串行化为getPrice的子元素。每一个类型都被显式地使用xsi:type属性以xsd:string设定，而参数值被串行化为与参数相关联的元素的值。

现在让我们看一看使用GLUE API来传递参数都需要做些什么。我们已经看到GLUE将Java接口绑定到服务，而且像对待其他的Java实例一样对待那些接口的方法。因此在基于GLUE的应用程序中传递参数就没什么深奥的东西了。但是为了强化你的概念，无论如何还是让我们来做一遍。

假设CallCounterApp应用程序现在还在运行中，我们就可以使用GLUE控制台来在这个GLUE服务器实例上部署新的服务。在GLUE服务器的主页上单击ADD按钮将把你带到一个空的Web服务编辑器表单。填入适当的服务细节，如图4-12所示。

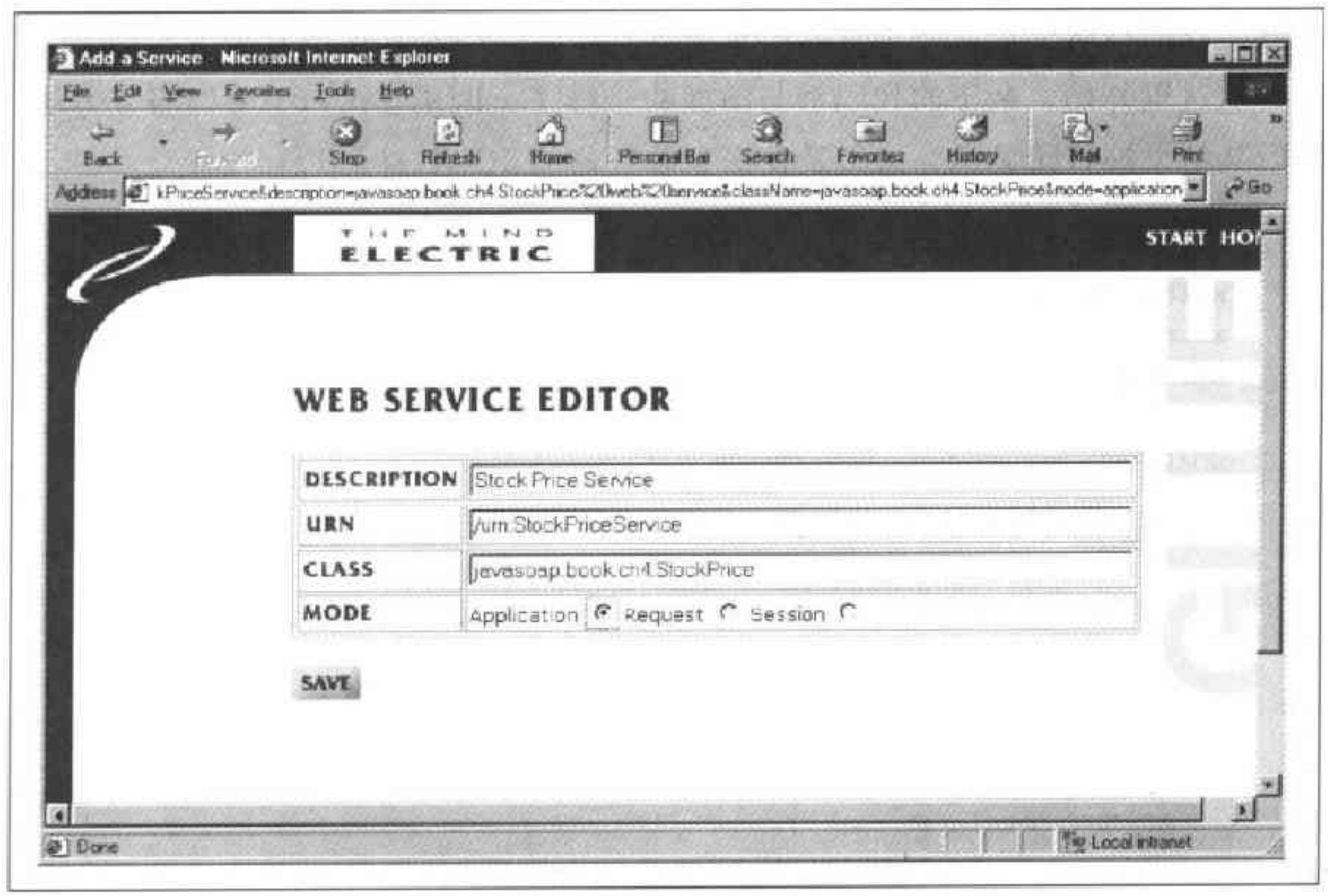

图 4-12: 用 Web 服务编辑器指定细节

在这个图中，我已经为该服务作出简短的描述，以及服务的URN (urn:StockPriceService)。实现服务的Java类是StockPrice，而且服务的激活期模式是应用程序级的。单击SAVE来部署服务。

现在我们需要一个Java接口来绑定到这个服务。如果你喜欢，可以使用GLUE控制台来生成它；对于这个例子，我们将使用接口IStockPriceService，如下所示。对于上面两种情况，要记住GLUE控制台生成接口时所使用的名称与我们手工建立接口所使用的名称是不同的。

```
package javasoap.book.ch4;
public interface IStockPriceService {
  float getPrice(String stock, String currency);
}
```

使用GLUE API来编写一个访问StockPriceService的应用程序很简单。在此我们将不使用GLUE控制台的辅助类，尽管没什么理由不让你这样做。这里是访问该服务的应用程序：

```
package javasoap.book.ch4;
import electric.registry.Registry;
import electric.registry.RegistryException;
public class StockPriceApp2 {
   public static void main(String[] args) throws Exception
   {
      try {
         IStockPriceService ctr =
           (IStockPriceService)Registry.bind(
              "http://georgetown:8004/glue/urn:StockPriceService.wsdl",
              IStockPriceService.class);
         String stock = "MINDSTRM";
         String currency = "USD";
         System.out.println("Price is " +
           ctr.getPrice(stock, currency));
      }
      catch (RegistryException e)
      {
        System.out.println(e);
      }
   }
}
```

一旦你为服务绑定了一个接口，就可以直接使用这个Java接口，就好像该服务对象在本地一样。因此我们需要做的全部事情就是调用 ctr.getPrice()，并将stock和currency变量作为参数传递给它。运行StockPriceApp2，你将得到如下的输出：

```
Price is 75.33
```

我们还没有看到过GLUE应用程序传输的SOAP消息。现在就让我们看看它与Apache SOAP传输的消息有什么不同。这里是GLUE API传输的SOAP封套：

```
<soap:Envelope
 xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
 xmlns:xsd='http://www.w3.org/2001/XMLSchema'
 xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
 xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
 soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
  <soap:Body>
    <n:getPrice
       xmlns:n='http://tempuri.org/javasoap.book.ch4.StockPrice'>
         <arg0 xsi:type='xsd:string'>MINDSTRM</arg0>
         <arg1 xsi:type='xsd:string'>USD</arg1>
```

```
      </n:getPrice>
    </soap:Body>
  </soap:Envelope>
```

GLUE产生的SOAP封套采用与Apache SOAP一样的方法建立xsi和xsd名称空间标识符。GLUE没有使用SOAP-ENV标识符，而是使用了名称soap，很难说这是一种重要的不同；它仅仅是一个名称而已。它可以是任何名称、只要分配给它的值是http://schemas.xmlsoap.org/soap/envelope。encodingStyle属性也与在Apache SOAP例子中的相同。看起来其中好像有一个额外的属性soapenc，但是在消息中没有任何地方引用它，所以那将对任何事情都不产生影响。用来表示服务方法的getPrice元素没有使用服务名称来进行名称空间限定，这与Apache SOAP的封套中的情况不同。它的名称空间在实现服务的类的名称前面加上了地址*http://tempuri.org/*，这是GLUE产生的默认名称空间。与Apache SOAP的例子不同，它的参数叫做arg0、arg1等等。既然方法的参数以它们出现的顺序被列出，因此其名称并不重要。它们的类型被显式地指定，正如它们在Apache例子中一样。

我比较喜欢使用实际的服务名称来限定服务方法的名称，而不是使用一个默认的不真实的服务名称。这是容易更正的；只要为服务器运行时使用的context对象提供一个名称空间属性就行了。我们用了同样的方法来指定该服务的激活期模式。这里是服务器应用程序（原来的CallCounterApp）的一个修改版本，只不过采用了调用context.addProperty()来指定名称空间，我们以"namespace"作为它的第一个参数，以"urn: StockPriceService"作为第二个参数。这个新的应用程序发布urn:StockPriceService，因此我们不需要使用GLUE控制台。

```
package javasoap.book.ch4;
import electric.util.Context;
import electric.registry.Registry;
import electric.server.http.HTTP;
public class StockPriceApp3 {
   public static void main( String[] args )
      throws Exception {

      HTTP.startup("http://georgetown:8004/glue");
      Context context = new Context();
      context.addProperty("activation", "application");
      context.addProperty("namespace", "urn:StockPriceService");
```

```
        Registry.publish("urn:StockPriceService",
            javasoap.book.ch4.StockPrice.class, context );
    }
}
```

如果你的服务器上仍有一个CallCounterApp的实例在运行，先终止它；因为它使用了与新应用程序相同的端口和地址。一旦启动了StockPriceApp3，SOAP消息中用于限定方法名称的名称空间将会是urn:StockPriceService。让我们看看服务器对这个请求产生的响应：

```
<soap:Envelope
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <soap:Body>
      <n:getPriceResponse xmlns:n='urn:StockPriceService'>
        <Result xsi:type='xsd:float'>75.33</Result>
      </n:getPriceResponse>
    </soap:Body>
  </soap:Envelope>
```

这个响应封套与Apache SOAP产生的封套非常相似；惟一不同的是返回值被包含在一个名为Result的元素中，而在Apache SOAP中这个元素叫做return。但是重要的是响应的结构，而你可以看到这个响应的结构与Apache SOAP服务器的响应是相同的。

使用in、out和in/out参数

到目前为止，所有的例子都假定方法的参数中包含了由客户传递到服务的值，而且服务在返回之前不改变参数的值。这是SOAP RPC服务的默认行为；在这种情况下参数是作为in参数被引用的，因为这时值被传递“进”服务。但是SOAP也支持使用其他两种参数形式。服务方法包含的参数可能不包含从客户传递到服务的数据，但是在返回时却包含了数据（要被客户读取）。这些参数被称为out参数：参数的值传递“出”服务。参数也可能是in/out的，表示一个值被传入服务，而另外一个值被传出服务。

不是所有的SOAP实现都支持使用out或者in/out参数（Apache SOAP不支持；而GLUE支持）。这对你应该是一个警告。除非你能控制服务器以及它们的所有客户，否则使用没有被广泛支持的特性不是一个好主意。你可以通过这个办法来避免使用out和in/out参数：使用包含你希望返回的值的自定义类型，然后定义你的方法让它返回一个自定义类型的实例。这就是我所使用的方式；它还能避免有关方法目的的任何混乱。另一方面，最近出现的实现都能支持in/out参数。事实上，它们中有许多工具所使用的技术都与我们正要去探究的技术相同。

尽管我已发出了反对使用它们的警告，还是让我们来看一看如何使用out和in/out参数。GLUE在使用out和in/out参数时采用了用于封装数据类型的持有者类（holder class）。Java中已有像java.lang.String这样的简单类和通用类。对于自定义数据类型，GLUE包含了一个holder工具，它可以产生持有者类（请牢记这种方法是GLUE特有的；SOAP没有对API做出规定，只规定了XML编码方式）。这里是StockPrice类的一个修改版本，它包括了两个新方法。第一方法是getPriceOut()；它返回一个布尔值来表示请求是否成功。它的前两个参数与前面那个getPrice()方法中的相同：股票代码和货币单位。第三个参数叫做price；当方法被调用时，它是一个没有值的out参数，当方法返回时，它包含了结果价格。这样getOutPrice()就使用了一个out参数来返回价格，该价格是由getPrice()方法计算得到的。price参数是electric.util.holder.floatOut的一个实例，它是一个为浮点类型out参数服务的持有者类。

另一个新方法叫做correctSymbol()，它将一个股票代码转换成适当的格式（全部换成大写字母）。它返回一个用来表示成功或失败的布尔值，但是更重要的是它只使用了一个叫做symbol的字符串类型in/out参数。当该方法被调用的时候，这个参数包含一个股票代码；当correctSymbol()返回时，它将包含被转换后的股票代码。为了支持这个行为，符号参数被声明为electric.util.holder.java.lang.StringInOut 的一个实例，这个类是一个为字符串类型in/out参数服务的持有者类。可以不对StockPriceApp3做任何修改就发布它的服务。

```
package javasoap.book.ch4;
import electric.util.holder.*;
import electric.util.holder.java.lang.*;
public class StockPrice {
```

```
    public float getPrice(String stock, String currency) {
        float result;
        // 确定股票的价格
        // 并用指定的货币单位返回它
        result = (float)75.33;
        return result;
    }
    public boolean getOutPrice(String stock, String currency,
                               floatOut price) {
        try {
            price.value = getPrice(stock, currency);
            return true;
        }
        catch (Exception e) {
            return false;
        }
    }
    public boolean correctSymbol(StringInOut symbol) {
        symbol.value = symbol.value.toUpperCase();
        return true;
    }
}
```

既然我们通过添加方法修改了服务，我们就需要相应地修改 javasoap.book.ch4.IStockPriceService 接口。这里是新的接口：

```
package javasoap.book.ch4;
import electric.util.holder.*;
import electric.util.holder.java.lang.*;
public interface IStockPriceService {
  float getPrice(String stock, String currency);
  public boolean getOutPrice(String stock, String currency,
                             floatOut price);
  public boolean correctSymbol(StringInOut symbol);
}
```

现在让我们创建一个应用程序来使用这个服务。我们将首先调用 correct-Symbol()来使用 in/out 参数将一个股票代码转换成大写字母，然后再调用 getOutPrice()来使用 out 参数得到价格：

```
package javasoap.book.ch4;
import electric.registry.Registry;
import electric.registry.RegistryException;
import electric.util.holder.*;
import electric.util.holder.java.lang.*;
```

```
public class InOutApp {
   public static void main(String[] args) throws Exception
   {
      try {
         IStockPriceService ctr =
           (IStockPriceService)Registry.bind(
             "http://georgetown:8004/glue/urn:StockPriceService.wsdl",
             IStockPriceService.class);
         StringInOut stock = new StringInOut("MiNdStRm");
         boolean ok = ctr.correctSymbol(stock);
         if (ok) {
           System.out.println("Symbol converted to: "
                        + stock.value);
         }
         String currency = "USD";
         floatOut price = new floatOut();
         ok = ctr.getOutPrice(stock.value, currency, price);
         System.out.println("Result is " + price.value);
      }
      catch (RegistryException e)
      {
        System.out.println(e);
      }
   }
}
```

股票代码变量stock使用了符号“MiNdStRm”进行实例化。我们现在将这个作为electric.util.holder.java.lang.StringInOut实例的变量传递到服务接口的correctSymbol()方法。我们期望在方法调用返回时该符号已被转换成大写字母。接下来，我们调用getOutPrice()，使用stock.value作为第一个参数。货币单位用与前面一样的方法传入，而最后一个参数使用electric.holder.floatOut的一个实例price。当方法返回时，price中将包含股票价格。运行这个例子将产生如下输出:

```
Symbol converted to: MINDSTRM
Result is 75.33
```

你可以看到，out和in/out参数都被如期处理。在结束本章之前，让我们来看一看在out和in/out参数影响下客户和服务器之间传递的SOAP封套的部分内容。对correctSymbol()的调用将产生一个与我们以前看到的有些不同的响应封套。在被返回的布尔值之后紧跟着修改过的参数值。这里是该响应的SOAP体:

```
<soap:Body>
  <n:correctSymbolResponse xmlns:n='urn:StockPriceService'>
    <Result xsi:type='xsd:boolean'>true</Result>
    <arg0 xsi:type='xsd:string'>MINDSTRM</arg0>
  </n:correctSymbolResponse>
</soap:Body>
```

包含被修改的 in/out 参数值的 XML 元素 arg0 紧接着 Result 元素出现。而且，当然，你能看到最初的值已被转换成大写字母。现在我们来看看调用 getOutPrice()时发送到服务器的封套的部分内容。只有两个参数值被编码，这是因为第三个参数是一个 out 参数，因此在方法被调用时还没有值。

```
<n:getOutPrice xmlns:n='urn:StockPriceService'>
  <arg0 xsi:type='xsd:string'>MINDSTRM</arg0>
  <arg1 xsi:type='xsd:string'>USD</arg1>
</n:getOutPrice>
```

最后，这里是对 getOutPrice()调用后返回的 SOAP 体的响应部分。price 参数的值跟在结果之后。前两个参数没有出现是因为它们两个都是 in参数，所以它们的值没有被更改。

```
<n:getOutPriceResponse xmlns:n='urn:StockPriceService'>
  <Result xsi:type='xsd:boolean'>true</Result>
  <arg2 xsi:type='xsd:float'>75.33</arg2>
</n:getOutPriceResponse>
```

在下一章中，我们将开始探讨 SOAP 中更复杂的特性和能力以及它的实现。

第五章

处理复杂数据类型

在前一章中，我们用Java创建了几个RPC形式的服务，这些服务仅仅处理了简单数据类型。简单数据类型可能在有些场合很合适，但你总会遇到需要使用更复杂的数据类型的时候，就像平时在Java编程中经常遇到的一样。在这一章中，我们来看看如何使用以数组和Java bean作为参数和返回值的方法来建立服务。由于所有的自定义数据类型都可能用到本章中讨论的复杂数据类型，因此一直到下一章结束，我们都将涉及到用户自定义的数据类型。

传递数组参数

我们来看看数组，它们也许是在程序设计中使用得最普遍的复杂数据类型了。它们无处不在，当然在SOAP中也是如此。我们在第三章中曾使用过SOAP数组，因此你可能对数组如何编码的细节已有所了解。那么现在就让我们用数组来编写Java的服务吧。

本章中我们继续使用前面使用过的股票市场的例子。一个能够返回股票信息的服务会是很有用的，它需要返回当天股票的交易量、这些股票的平均交易价格、当天价格上涨的股票数等等。可能还有其他各种情况。我们以一个返回当日股票总交易量的服务作为开始。我们将这个服务叫做urn:BasicTradingService，它有一个名为getTotalVolume的方法。下面是实现这个服务的Java类：

```
package javasoap.book.ch5;
public class BasicTradingService {

   public BasicTradingService() {}

   public int getTotalVolume(String[] stocks) {

      // 从某个数据来源得到各股票的交易量
      // 并返回其总数
      int total = 345000;
      return total;
   }
}
```

BasicTradingService类有一个getTotalVolume()方法，该方法返回交易的总数。由于我们不打算真正从数据提供者处取得输入数据，因此我们只返回一个虚构的值。这个方法接受一个名为stocks的字符串数组并返回一个整数。这个字符串数组中的值代表股票代码，在实际应用中，我们需要从数据提供者处分别取得字符串数组中各个股票的交易数量，然后返回一个总数。

Apache SOAP已经为数组提供了内置的支持，因此你不需要在服务器端做什么特别的事情。这也意味着在部署描述文件中不用添加新的内容，和前几章中介绍的一样即可。下面是服务urn:BasicTradingService的部署描述文件：

```
<isd:service
      xmlns:isd="http://xml.apache.org/xml-soap/deployment"
      id="urn:BasicTradingService">
   <isd:provider type="java"
      scope="Application"
      methods="getTotalVolume">
     <isd:java class="javasoap.book.ch5.BasicTradingService"
      static="false"/>
   </isd:provider>
   <isd:faultListener>org.apache.soap.server.DOMFaultListener
   </isd:faultListener>
   <isd:mappings>
   </isd:mappings>
</isd:service>
```

你可以使用上面这个部署描述文件来部署你的服务，或者在浏览器中使用Apache SOAP的Admin工具来部署它。

现在我们要写一个访问该服务的客户应用程序。这与前几章里的很相似，不同的仅仅是传入的参数类型，这里我们用字符串数组代替了前面的简单类型。这是它的代码:

```
package javasoap.book.ch5;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
public class VolumeClient {
   public static void main(String[] args) throws Exception {

      URL url = new URL("http://georgetown:8080/soap/servlet/rpcrouter");

      Call call = new Call();
      call.setTargetObjectURI("urn:BasicTradingService");

      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      String[] stocks = { "MINDSTRM", "MSFT", "SUN" };
      Vector params = new Vector();
      params.addElement(new Parameter("stocks",
                           String[].class, stocks, null));
      call.setParams(params);

      try {
         call.setMethodName("getTotalVolume");
         Response resp = call.invoke(url, "");
         Parameter ret = resp.getReturnValue();
         Object value = ret.getValue();
         System.out.println("Total Volume is " + value);
      }
      catch (SOAPException e) {
         System.err.println("Caught SOAPException (" +
                              e.getFaultCode() + "): " +
                              e.getMessage());
      }
   }
}
```

我们向Parameter构造函数传入的第二个参数是String[].class,这说明stocks参数的类型是字符串数组。好了，到现在为止我们已经完成了所有的工作。如果你运行这个例子，其输出应该是这样的:

```
Total Volume is 345000
```

我们现在来看看为了调用服务的getTotalVolume方法而向服务器传送的SOAP封套:

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

  <SOAP-ENV:Body>
    <ns1:getTotalVolume xmlns:ns1="urn:BasicTradingService"
      SOAP-ENV:encodingStyle=
           "http://schemas.xmlsoap.org/soap/encoding/">
         <stocks
           xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/"
           xsi:type="ns2:Array"
           ns2:arrayType="xsd:string[3]">
              <item xsi:type="xsd:string">MINDSTRM</item>
              <item xsi:type="xsd:string">MSFT</item>
              <item xsi:type="xsd:string">SUN</item>
         </stocks>
    </ns1:getTotalVolume>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

stocks元素表示我们传给服务方法的字符串数组参数。通过把xsi:type属性设置为ns2:Array来指定数组类型。

ns2名称空间标识符在前面的行中定义。接下来，为ns2:arrayType属性分配一个值xsd:string[3]。这表示该数组的大小为3，数组的每个元素都是xsd:string类型。stocks元素有三个子元素，每一个的名称都是item。记住，数组的子元素使用什么名称并不重要，因为不同的SOAP实现使用不同的模式来为这些元素命名。本例通过将xsi:type设为xsd:string显式地指定了数组中每个元素的类型。

这个例子使用的是同构数组（homogeneous array），也就是说，数组中所有的元素都是同种类型的。你或许还会遇到要使用异构数组（heterogeneous array）的情况，让我们来看看这种可能性。在Java语言中，方法常常使用数组作为参数，而在其他语言里可能使用一个变长的参数列表。举个例子，C语言中的printf()函数的参数个数就是不固定的。尽管Java不支持这种方式，但你可以通过在数组

中传递参数值来模拟这种功能。一个数组可以是任何长度的，而且其中的元素并不一定要是同一类型的。

现在我们向urn:BasicTradingService服务中添加一个用异构数组作为参数的方法executeTrade。它的参数是一个数组，数组的元素包含股票代码、需要交易的数量和一个表示是买还是卖的标志（true表示买）。返回值是一个描述交易的字符串（注1）。这里是改动后的BasicTradingService类：

```
package javasoap.book.ch5;
public class BasicTradingService {

    public BasicTradingService() {
    }
    public int getTotalVolume(String[] stocks) {

        // 从某个数据来源得到各股票的交易量
        // 并返回其总数

        int total = 345000;
        return total;
    }
    public String executeTrade(Object[] params) {
        String result;
        try {
            String stock = (String)params[0];
            Integer numShares = (Integer)params[1];
            Boolean buy = (Boolean)params[2];
            String orderType = "Buy";
            if (false == buy.booleanValue()) {
                orderType = "Sell";
            }
            result = (orderType + " " + numShares + " of " + stock);
        }
        catch (ClassCastException e) {
            result = "Bad Parameter Type Encountered";
        }
        return result;
    }
}
```

注1：当然还有其他方式来设计这样的方法接口，而且这也不是我所选择的设计。这种情况可能会要求使用自定义类或Java bean。但本例中所使用的方法演示了同构数组的使用。

executeTrade()方法只有一个参数：一个叫params的Object[]数组。这个数组中的对象必须转换成相应的类型：String、Integer和Boolean。我喜欢将类型转换放在一个try/catch块中以防调用者出错。在这种方式下，我可以在方法调用错误时做一些适当的处理，尽管有时候只是简单地返回错误描述。在第七章里，我们将介绍在这种情况下如何生成SOAP错误。executeTrade()方法将数组传递的信息生成一个描述输入参数的字符串，该字符串保存在result变量中并返回给调用者。

现在我们来修改客户应用程序，使它能够向服务的executeTrade方法传递一个适当的Object[]类型的参数。multiParams变量被声明为Object[]类型，它由一个字符串MINDSTRM、一个值为100的整数和一个值为true的布尔变量组成。既然我们使用的是Java的Object数组，我们就不再使用Java的简单类型作为数组的元素。取而代之，我们将这些简单值包装到等价的Java对象里。Parameter构造函数的第二个参数是Object[].class，它是一个对象数组的类。

```
package javasoap.book.ch5;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
public class TradingClient {
   public static void main(String[] args)
      throws Exception {

      URL url =
        new URL(
          "http://georgetown:8080/soap/servlet/rpcrouter");

      Call call = new Call();
      call.setTargetObjectURI("urn:BasicTradingService");

      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      Object[] multiParams = { "MINDSTRM", new Integer(100),
                               new Boolean(true) };
      Vector params = new Vector();
      params.addElement(new Parameter("params",
                        Object[].class, multiParams, null));
      call.setParams(params);
```

```
        try {
            call.setMethodName("executeTrade");
            Response resp = call.invoke(url, "");
            Parameter ret = resp.getReturnValue();
            Object value = ret.getValue();
            System.out.println("Trade Description: " + value);
        }
        catch (SOAPException e) {
            System.err.println("Caught SOAPException (" +
                               e.getFaultCode() + "): " +
                               e.getMessage());
        }
    }
}
```

如果一切正常，运行 executeTrade()服务方法的结果将是:

```
Trade Description: Buy 100 of MINDSTRM
```

我们可以试试改变multiParams数组中参数的顺序。在这种情况下，我们将得到一个类转换异常，而且该方法会返回一个错误字符串。

这里是正确调用 executeTrade()服务方法的 SOAP 封套:

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

  <SOAP-ENV:Body>
    <ns1:executeTrade xmlns:ns1="urn:BasicTradingService"
      SOAP-ENV:encodingStyle=
          "http://schemas.xmlsoap.org/soap/encoding/">

      <params
          xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/"
          xsi:type="ns2:Array" ns2:arrayType="xsd:anyType[3]">
        <item xsi:type="xsd:string">MINDSTRM</item>
        <item xsi:type="xsd:int">100</item>
        <item xsi:type="xsd:boolean">true</item>
      </params>
    </ns1:executeTrade>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

通过将xsi:type属性的值设置为ns2:Array来设定params元素的类型。它与同构的情况相比惟一不同的是，数组中各个元素的类型由不同的ns2:arrayType属性值来设定。值xsd:anyType[3]指定数组包含3个元素，每一个都可能是任意的有效数据类型（注2）。

现在让我们看看在GLUE中传递数组参数的方法。BasicTradingService类在GLUE中可以不做修改就部署。我们将使用一个简单的Java应用程序来启动这个服务：

```
package javasoap.book.ch5;
import electric.util.Context;
import electric.registry.Registry;
import electric.server.http.HTTP;
public class BasicTradingApp {
   public static void main( String[] args )
      throws Exception {

      HTTP.startup("http://georgetown:8004/glue");
      Context context = new Context();
      context.addProperty("activation", "application");
      context.addProperty("namespace",
                              "urn:BasicTradingService");
      Registry.publish("urn:BasicTradingService",
         javasoap.book.ch5.BasicTradingService.class, context );
   }
}
```

编译并运行这个应用程序，该服务将被部署。现在让我们使用GLUE API写一个简单的例子来访问这个服务。首先我们来看服务的接口IBasicTradingService：

```
package javasoap.book.ch5;
public interface IBasicTradingService {
  int getTotalVolume(String[] symbols);
  String executeTrade(Object[] params);
}
```

注2：在第三章里我们提到了用ur-type来表示任何可能的数据类型。在这里我们使用了anyType来达到同样的目的。2001年5月发布的XML Schema建议性标准的第0部分在第2.5.4节中做出了下面的解释："anyType表示一个叫做ur-type的抽象、ur-type是派生所有的简单类型和复杂类型的基础类型。anyType类型不以任何形式约束它的内容。"

现在我们可以编写一个绑定到该服务并调用它的两个方法的应用程序:

```
package javasoap.book.ch5;
import electric.registry.RegistryException;
import electric.registry.Registry;
public class BasicTradingClient {
   public static void main(String[] args) throws Exception
   {
      try {
        IBasicTradingService srv =
          (IBasicTradingService)Registry.bind(
            "http://georgetown:8004/glue/urn:BasicTradingService.wsdl",
            IBasicTradingService.class);
        String[] stocks = { "MINDSTRM", "MSFT", "SUN" };
        int total = srv.getTotalVolume(stocks);
        System.out.println("Total Volume is " + total);
        Object[] multiParams = { "MINDSTRM", new Integer(100),
                                 new Boolean(true) };
        String desc = srv.executeTrade(multiParams);
        System.out.println("Trade Description: " + desc);
      }
      catch (RegistryException e)
      {
         System.out.println(e);
      }
    }
  }
```

正如我们以前看到的，GLUE允许我们使用熟悉的Java编程语法，而不必考虑底层的SOAP结构。这对传递数组参数同样适用。在将接口绑定到服务之后，所有的事情都已经被我们处理得非常好了。

在本例所产生的SOAP请求封套中，你会看到一些有趣的事情。这就是getTotalVolume()方法调用的SOAP封套:

```
<soap:Envelope
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <soap:Body>
      <n:getTotalVolume xmlns:n='urn:BasicTradingService'>
        <arg0 href='#id0'/>
```

```
        </n:getTotalVolume>
        <id0 id='id0' soapenc:root='0'
          xmlns:ns2='http://www.themindelectric.com/package/java.lang/'
          xsi:type='soapenc:Array' soapenc:arrayType='xsd:string[3]'>
           <i xsi:type='xsd:string'>MINDSTRM</i>
           <i xsi:type='xsd:string'>MSFT</i>
           <i xsi:type='xsd:string'>SUN</i>
        </id0>
      </soap:Body>
  </soap:Envelope>
```

Body元素的前面部分与我们以前看到的差不多；而封套由名称空间标识符soap限定，它表示SOAP封套的名称空间。SOAP体的第一个子元素是getTotalVolume，这当然是被调用的服务方法的名称。getTotalVolume是由服务名称作为名称空间限定的。getTotalVolume的惟一子元素是arg0，表示被传递给方法的参数。但是这不是我们传递的数组；它是对该数组的引用。这是GLUE与Apache SOAP API为产生这个调用所采用的方式之间的一个明显不同。Apache SOAP把数组作为getTotalVolume的一个子元素放在封套中，而GLUE使用引用并在getTotalVolume元素结束之后才串行化该数组。于是该参数就被串行化为arg0，包含一个值为#id0的href属性。它不包含任何数据，因为这个数组存在于别处。

我们传递的数组参数紧跟在getTotalVolume元素之后。它叫做id0，由GLUE产生，不过元素的名称本身并不重要。其id属性的值被设定为id0，它与getTotalVolume元素中href属性的值一致。GLUE生成了soapenc:root属性并赋值为0，这意味着该元素不被作为某个对象图（object graph）的根（GLUE好像会自动包括这个属性）。接着我们会看到一个叫做ns2的名称空间标识符，它看起来好像代表了GLUE的内部包java.lang；然而，ns2名称空间标识符根本就没有被用到。xsi:type和soapenc:arrayType属性建立的方式与在Apache SOAP例子中的一样。最后，数组中的元素被串行化。在串行化方面，这个例子与由Apache SOAP生成的那个例子之间的惟一不同在于数组元素本身的名称。Apache SOAP将它们命名为item，而GLUE将它们命名为i。名称无关紧要；结果会是一样的。

这个例子给了我们一个机会看到两种串行化数组的方式，而这两种方式都是正确的。如果要实现互操作，那么让SOAP实现了解这些不同的形式就非常重要。这

就是我们坚持要列出例子所产生的封套内容的原因。如果你在让基于不同实现的应用程序互相正确通信的问题上遇到了困难(我是说如果),那么熟悉这些不同的形式将帮助你找到解决的办法。

让我们看看调用executeTrade服务方法所使用的SOAP封套。这个方法接受一个异构数组作为参数。它也使用了一个对独立串行化数组的引用,这次它被编码为xsd:anyType:

```
<soap:Envelope
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <soap:Body>
      <n:executeTrade xmlns:n='urn:BasicTradingService'>
        <arg0 href='#id0'/>
      </n:executeTrade>
      <id0 id='id0' soapenc:root='0'
        xmlns:ns2='http://www.themindelectric.com/package/java.lang/'
        xsi:type='soapenc:Array' soapenc:arrayType='xsd:anyType[3]'>
        <i xsi:type='xsd:string'>MINDSTRM</i>
        <i xsi:type='xsd:int'>100</i>
        <i xsi:type='xsd:boolean'>true</i>
      </id0>
    </soap:Body>
</soap:Envelope>
```

返回数组

到目前为止我们已经将数组作为传入参数使用过了。现在我们来使用一个数组作为服务方法的返回值。我们将在服务中加入一个叫做getMostActive()的方法,它返回一个String[]来包含当日成交最为活跃的那些股票的代码。这里是BasicTradingService类的新版本:

```
package javasoap.book.ch5;
public class BasicTradingService {

   public BasicTradingService() {
   }
```

```
    public String[] getMostActive() {

        // 取得交易最活跃的股票
        String[] actives = { "ABC", "DEF", "GHI", "JKL" };
        return actives;
    }
    public int getTotalVolume(String[] stocks) {

        // 从某个数据源得到股票的交易量
        // 并返回其总数
        int total = 345000;
        return total;
    }
    public String executeTrade(Object[] params) {
        String result;
        try {
            String stock = (String)params[0];
            Integer numShares = (Integer)params[1];
            Boolean buy = (Boolean)params[2];
            String orderType = "Buy";
            if (false == buy.booleanValue()) {
                orderType = "Sell";
            }
            result = (orderType + " " + numShares + " of " + stock);
        }
        catch (ClassCastException e) {
            result = "Bad Parameter Type Encountered";
        }
        return result;
    }
}
```

既然我们并不是真正从数据来源处获取数据，我们只需要将一些假的股票代码填入数组并返回它就可以了。接下来要重新部署这个服务。从一个Apache SOAP客户调用这个服务很简单。服务方法没有输入参数，因此我们仅需要建立调用然后启动它：

```
package javasoap.book.ch5;
import java.net.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
public class MostActiveClient
{
  public static void main(String[] args) throws Exception
  {
```

```
URL url = new
  URL("http://georgetown:8080/soap/servlet/rpcrouter");
Call call = new Call();
call.setTargetObjectURI("urn:BasicTradingService");
call.setMethodName("getMostActive");
Response resp;
try {
  resp = call.invoke(url, "");
  Parameter ret = resp.getReturnValue();
  String[] value = (String[])ret.getValue();
  int cnt = value.length;
  for (int i = 0; i < cnt; i++) {
     System.out.println(value[i]);
  }
}
catch (SOAPException e) {
  System.err.println("Caught SOAPException (" +
                     e.getFaultCode() + "): " +
                     e.getMessage());
}
}
}
```

我们将ret.getValue返回值的类型转换为String[]，因为这才是我们所需要的类型。在过去的例子中，我们能够让一个值保持为Object的实例不变是因为我们依赖了Object的toString()方法来显示这个值。现在我们要返回该数组，因此需要将它的值转换成适当的数组类型。接下来，我们需要得到数组的长度然后顺序打印我们所得到的数组的每一个值。如果运行了这个例子，你将会看到如下的输出：

```
ABC
DEF
GHI
JKL
```

这个方法调用所返回的SOAP封套是相当简单的：

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"> <SOAP-ENV:Body>
     <ns1:getMostActiveResponse
        xmlns:ns1="urn:BasicTradingService"
```

```
      SOAP-ENV:encodingStyle=
        "http://schemas.xmlsoap.org/soap/encoding/">

    <return
      xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/"
      xsi:type="ns2:Array" ns2:arrayType="xsd:string[4]">
        <item xsi:type="xsd:string">ABC</item>
        <item xsi:type="xsd:string">DEF</item>
        <item xsi:type="xsd:string">GHI</item>
        <item xsi:type="xsd:string">JKL</item>
    </return>
  </ns1:getMostActiveResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

为了在GLUE中部署BasicTradingService类的这一个版本，可以使用我们原来的BasicTradingApp类。我们可以修改Java接口IBasicTradingService，使它包括新的方法：

```
package javasoap.book.ch5;
public interface IBasicTradingService {
  int getTotalVolume(String[] symbols);
  String executeTrade(Object[] params);
  String[] getMostActive();
}
```

现在我们要修改应用程序BasicTradingClient来加入一个对getMostActive()方法的调用，然后在数组中迭代各个值并打印它们。在使用GLUE时我们不需要将返回值的类型转换为String[]，这是因为，与Apache SOAP的例子不同，这个接口的getMostActive()方法已经被定义为返回正确的类型了。这里是修改后的代码：

```
package javasoap.book.ch5;
import electric.registry.RegistryException;
import electric.registry.Registry;
public class BasicTradingClient {
   public static void main(String[] args) throws Exception
   {
      try {
        IBasicTradingService srv = (IBasicTradingService)Registry.bind(
          "http://georgetown:8004/glue/urn:BasicTradingService.wsdl",
          IBasicTradingService.class);
```

```
        String[] stocks = { "MINDSTRM", "MSFT", "SUN" };
        int total = srv.getTotalVolume(stocks);
        System.out.println("Total Volume is " + total);
        Object[] multiParams = { "MINDSTRM", new Integer(100),
                                 new Boolean(true) };
        String desc = srv.executeTrade(multiParams);
        System.out.println("Trade Description: " + desc);
        String[] actives = srv.getMostActive();
        int cnt = actives.length;
        for (int i = 0; i < cnt; i++) {
           System.out.println(actives[i]);
        }
    }
    catch (RegistryException e)
    {
       System.out.println(e);
    }
  }
}
```

运行这个例子会得到如下输出:

```
Total Volume is 345000
Trade Description: Buy 100 of MINDSTRM
ABC
DEF
GHI
JKL
```

GLUE对作为返回值的数组使用了与前面作为参数的数组相同的串行化技术;它为实际的返回值建立了一个独立串行化的数组,然后又引用了这个实际的数组值。调用getMostActive()方法所返回的SOAP封套是:

```
<soap:Envelope
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <soap:Body>
      <n:getMostActiveResponse
        xmlns:n='urn:BasicTradingService'>
         <Result href='#id0'/>
      </n:getMostActiveResponse>
      <id0 id='id0' soapenc:root='0'
```

```
        xmlns:ns2='http://www.themindelectric.com/package/java.lang/'
        xsi:type='soapenc:Array' soapenc:arrayType='xsd:string[4]'>
         <i xsi:type='xsd:string'>ABC</i>
         <i xsi:type='xsd:string'>DEF</i>
         <i xsi:type='xsd:string'>GHI</i>
         <i xsi:type='xsd:string'>JKL</i>
      </id0>
    </soap:Body>
</soap:Envelope>
```

传递自定义类型参数

作为Java程序员，我们当然不会把自己限制于仅仅使用标准Java软件包中的类。我们所编写的所有代码的实质部分是自定义类型，也就是我们自己建立的Java类，它们具体实现了我们所要创建的系统的功能和特性。

考虑一下要在一个Java类中描述一次股票交易所需要的所有数据。这个新类可能包含交易股票的代码、交易的数量以及一个对交易类型的描述（买还是卖）。当设计这样一个类的时候，在更大的系统上下文中去分析它是很重要的。这种分析能提供像给予类什么样的属性和行为这样的线索。我们一直在做这样的工作；这称为软件设计。其结果是产生了包含访问属性和行为的方法的Java类。由于SOAP是一个数据传送协议，所以我们对类的属性更感兴趣。这是我们想要在线路上传输的东西。

表示一个Java类的属性经常采用的方法是使用JavaBeans设计模式。这些模式为那些用于访问类的方法指定命名惯例。你可能不熟悉JavaBeans，但是我敢打赌，你肯定多次见过这个模式。这里是O'Reilly公司出版的《Developing Java Beans》一书中对属性存取器模式的描述：

> 用于保存或获取属性值的方法应遵循属性的标准设计模式。这些方法允许抛出检验过的异常，但是这是可选的。方法的声明方式如下：
>
> ```
> public void set<PropertyName>(<PropertyType> value);
> public <PropertyType> get<PropertyName>();
> ```
>
> 遵照这个模式声明的一对方法表示了一个名为<PropertyName>的<PropertyType>类型的可读写属性。如果只有get方法，就表示该属性为只

读；而如果只有set方法，则表示该属性为只写。在<PropertyType>为布尔类型时，其get方法可以使用如下的表示方式代替：

```
public boolean is<PropertyName>();
```

如果你遵照这个模式来命名属性存取器，Java的映射机制就能够在运行时确定这个存取方法（注3）。对于为了在SOAP消息中串行化数据而访问某个Java类实例的数据值的SOAP实现来说，这是一个很方便的方法。它使Apache SOAP和GLUE都能从这种映射中获益。这意味着你需要遵循一个良好的命名惯例，这样，当在SOAP中使用自定义数据时，你会受益匪浅。当然，要使用自定义数据，这还远远不够，现在让我们开始深入讨论这个问题。

我们首先来定义一次股票交易，看它到底要包含些什么内容。它需要有一个用来表示要交易的股票的代码；这个代码应该是字符串类型的元素。它还需要一个标志来说明是要买还是要卖。最后，它还需要一个包含交易数量的整数（在一个实际的应用程序中，它也可能包含各种不同的安全证书、买方和经纪人的名字以及其他许多东西。作为选择，这些额外的数据可以在其他对象中表示）。这种股票交易的XML模式片段看起来应该是这样的：

```
<element name="StockTrade" type="StockTrade"/>
<complexType name="StockTrade">
    <element name="symbol" type="xsd:string"/>
    <element name="buy" type="xsd:boolean"/>
    <element name="numshares" type="xsd:int"/>
</complexType>
```

我们来建立一个名为StockTrade_ClientSide的自定义Java类，它描述了一次股票交易。一般情况下我会将这个类命名为StockTrade，但是现在我想要表明我将在客户端使用这个类。我们会为服务器端的使用编写一个类似的类，它将使用一个对应的名字。我这样做是要说明你不必在消息事务的两端使用同一个Java类。事实上，也许使用同一个类是没有意义的，甚至常常是不可能的。

StockTrade_ClientSide有三个可读写的属性：Symbol、Buy和NumShares，它们分别表示股票代码、买卖标志和交易数量。

注3： JavaBean提供了其他办法来完成这个工作，但这超出了本书的研究范围。

```
package javasoap.book.ch5;
public class StockTrade_ClientSide {
   String   _symbol;
   boolean  _buy;
   int      _numShares;
   public StockTrade_ClientSide() {
   }
   public StockTrade_ClientSide(String symbol,
                     boolean buy, int numShares) {
      _symbol = symbol;
      _buy = buy;
      _numShares = numShares;
   }
   public String getSymbol() {
      return _symbol;
   }
   public void setSymbol(String symbol) {
      _symbol = symbol;
   }
   public boolean isBuy() {
      return _buy;
   }
   public void setBuy(boolean buy) {
      _buy = buy;
   }
   public int getNumShares() {
      return _numShares;
   }
   public void setNumShares(int numShares) {
      _numShares = numShares;
   }
}
```

现在让我们创建一个StockTrade_ServerSide类来表示服务器端的股票交易。只是为了使这个类与它在客户端的副本之间显得不同，我们删掉了带参数的构造函数。而且为了对应，我们需要改变类变量的名称并调整它们出现的顺序。

```
package javasoap.book.ch5;
public class StockTrade_ServerSide {
   int      _shares;
   boolean  _buyOrder;
   String   _stock;
   public StockTrade_ServerSide() {
   }
   public String getSymbol() {
      return _stock;
```

```
    }
    public void setSymbol(String stock) {
        _stock = stock;
    }
    public boolean isBuy() {
        return _buyOrder;
    }
    public void setBuy(boolean buyOrder) {
        _buyOrder = buyOrder;
    }
    public int getNumShares() {
        return _shares;
    }
    public void setNumShares(int shares) {
        _shares = shares;
    }
}
```

我们可以向urn:BasicTradingService服务中加入一个新的方法executeStockTrade()，它将股票交易作为参数。这个方法返回的是一个描述交易的字符串。下面是被修改后的BasicTradingService类。我们能利用这个类中已有的executeTrade()方法。在新方法executeStockTrade()中，我们用trade参数的三个属性创建了一个Object数组，并将这个数组传递给executeTrade()方法。

```
package javasoap.book.ch5;
public class BasicTradingService {

    public BasicTradingService() {
    }
    public String executeStockTrade(StockTrade_ServerSide trade) {
        Object[] params = new Object[3];
        params[0] = trade.getSymbol();
        params[1] = new Integer(trade.getNumShares());
        params[2] = new Boolean(trade.isBuy());
        return executeTrade(params);
    }
    public String[] getMostActive() {

        // 取得交易最活跃的股票
        String[] actives = { "ABC", "DEF", "GHI", "JKL" };
        return actives;
    }
    public int getTotalVolume(String[] stocks) {
```

```
        // 从某个数据来源得到各股票的交易量
        // 并返回其总数

        int total = 345000;
        return total;
    }
    public String executeTrade(Object[] params) {
        String result;
        try {
            String stock = (String)params[0];
            Integer numShares = (Integer)params[1];
            Boolean buy = (Boolean)params[2];
            String orderType = "Buy";
            if (false == buy.booleanValue()) {
                orderType = "Sell";
            }
            result = (orderType + " " + numShares + " of " + stock);
        }
        catch (ClassCastException e) {
            result = "Bad Parameter Type Encountered";
        }
        return result;
    }
}
```

为部署这个服务，我们需要把自定义类型映射到实现该类型的Java类。我们还需要给自定义类型一个名称并用一个适当的名称空间限定它。这有点像我们前面声明一个服务的步骤。映射发生在部署描述文件的 isd:sections部分中。这就是我们将用来在 Apache SOAP 中部署该服务的部署描述文件:

```
<isd:service
    xmlns:isd="http://xml.apache.org/xml-soap/deployment"
    id="urn:BasicTradingService"> <isd:provider
     type="java"
     scope="Application"
     methods="getTotalVolume getMostActive executeTrade executeStockTrade">
    <isd:java
       class="javasoap.book.ch5.BasicTradingService"
       static="false"/>
  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener
  </isd:faultListener>
  <isd:mappings>
    <isd:map
```

```
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:x="urn:BasicTradingService" qname="x:StockTrade"
        javaType="javasoap.book.ch5.StockTrade_ServerSide"
        java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
        xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
  </isd:mappings>
</isd:service>
```

让我们在继续介绍之前对映射做一个总结，这很重要。从自定义类型到Java类之间的映射与一种编码形式和一个完全限定的类型名称有关。一个Java类实现对自定义类型的描述，而工具类则用来完成串行化和反串行化工作。现在让我们来看例子中的细节。

部署描述文件中的大部分内容看起来很熟悉；它与第四章给出的很类似。不同的是我们现在的isd:mappings部分中多了一项。isd:map项描述了从股票交易类型到StockTrade_ServerSide类的映射。isd:map元素中没有数据；所有的信息都以属性形式提供。第一个属性encodingStyle指定了与串行化该类型相关的编码形式。下一个是名称空间标识符x，其值被设为urn:BasicTradingService。我使用了服务的名称作为自定义类型的名称空间限定符；然而，你不一定要这样做，而且事实上在某些情况下你也不可能这么做。举例来说，如果你的自定义类型被多个服务所使用，或者你使用的是一个由第三方定义的自定义类型，那么你肯定想要为自定义类型使用一个适当的名称空间。接下来的属性qname指定了该类型被映射到的完全限定符名。给它设定的值是x:StockTrade，这与用服务名限定的名称空间StockTrade是同一个名称空间。javaType属性指明了用于实现这个类型的服务器－本地Java类；在服务器端我们使用了javasoap.book.ch5.StockTrade_ServerSide。最后两个属性java2XMLClassName和xml2JavaClassName分别告诉Apache SOAP使用哪个本地Java类来完成串行化和反串行化工作。Apache SOAP自带了一个自定义串行化/反串行化类，它可以实现自定义XML类型与遵循JavaBeans属性存取器模式的Java类之间的转换。它能完成串行化和反串行化两种工作，这是我们为两个属性都使用了它的原因。在下一章中我们将看到如何创建自定义类型串行化器。

既然已经有了一个部署描述文件，我们就可以马上部署urn:BasicTrading-Service。一旦这样做了，我们的服务就已准备就绪，可以接受对execute-

StockTrade方法的调用了。然而，我们还需要在客户应用程序中也做一些安装工作。让我们看看客户应用程序：

```
package javasoap.book.ch5;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
public class StockTradeClient
{
  public static void main(String[] args) throws Exception
  {
    URL url = new
       URL("http://georgetown:8080/soap/servlet/rpcrouter");
    Call call = new Call();
    SOAPMappingRegistry smr = new SOAPMappingRegistry();
    call.setSOAPMappingRegistry(smr);
    call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
    call.setTargetObjectURI("urn:BasicTradingService");
    call.setMethodName("executeStockTrade");
    BeanSerializer beanSer = new BeanSerializer();
    // 映射Stock Trade类型
    smr.mapTypes(Constants.NS_URI_SOAP_ENC,
       new QName("urn:BasicTradingService", "StockTrade"),
       StockTrade_ClientSide.class, beanSer, beanSer);
    // 建立一个股票交易的实例
    StockTrade_ClientSide trade =
          new StockTrade_ClientSide("XYZ", false, 350);
    Vector params = new Vector();
    params.addElement(new Parameter("trade",
                         StockTrade_ClientSide.class, trade, null));
    call.setParams(params);
    Response resp;
    try {
      resp = call.invoke(url, "");
      Parameter ret = resp.getReturnValue();
      Object desc = ret.getValue();
      System.out.println("Trade Description: " + desc);
    }
    catch (SOAPException e) {
      System.err.println("Caught SOAPException (" +
                         e.getFaultCode() + "): " +
                         e.getMessage());
    }
```

```
    }
}
```

在建立了Call对象之后，我们又创建了org.apache.soap.encoding.SOAP-MappingRegistry的一个实例smr。这个类包含了从自定义类型到Java类的映射，并将它通过Call对象的setSOAPMappingRegistry()方法传给这个Call对象。我们在前面的例子中不需要这样做，这是因为在没有任何参数传递的情况下Call对象会为自己建立一个映射注册表。SOAPMappingRegistry()中包含了所有预先定义的映射，例如我们用于数组的那些映射。我们不久将会把我们的映射添加进去。

接下来我们调用了call.setEncodingStyleURI()方法。这个方法指定了我们为自定义类型所使用的总体编码形式。来自类org.apache.soap.Constants的常量NS_URI_SOAP_ENC表示我们使用的http://schemas.xmlsoap.org/soap/encoding/名称空间。这个编码形式的名称空间应该与我们在部署描述文件中为StockTrade类型指定的编码形式使用的名称空间相同。setTargetObjectURI()和setMethodName()与它们在前面例子中的使用方法是一样的。下一步是要创建org.apache.soap.encoding.soapenc.BeanSerializer类的一个实例；我们可以使用这个标准的串行化器是因为我们的自定义类型遵循了JavaBeans属性存取器模式。现在我们可以通过调用smr.mapTypes()来建立映射了。该方法的第一个参数是Constants.NS_URI_SOAP_ENC，指明映射要使用的编码形式是标准的SOAP编码形式。你可能会觉得很奇怪：我们已经在前面为整个调用指定过编码形式了，在这里为什么还要这样做？理由很简单：这个参数给你一个机会来覆盖这种特定映射的编码形式。不过，如果你使用null作为参数值，那么映射将对编码形式使用null名称空间，这是不正确的。我们来看这个消息的SOAP封套，你会看到尽管编码形式与总体调用的编码形式是一致的，但是对于这个串行化参数，encodingStyle属性并没有重复出现。如果某个参数的编码形式与Call使用的不同，它就会作为该参数的属性出现。

smr.mapTypes()的下一个参数是org.apache.soap.util.xml.Qname的一个实例，它表示一个被完全限定的名称（名称前面加上了一个名称空间限定符）。我们使用urn:BasicTradingService作为名称空间，StockTrade作为名称。第三个

参数是StockTrade_ClientSide.class，它是实现了自定义类型的Java类。接下来的两个参数是用于这个类型的串行化器和反串行化器的实例。因为org.apache.soap.encoding.soapenc.BeanSerializer类实现了串行化和反串行化，所以在这里我们使用了前面为它们创建的beanSer对象。

剩下的部分相当简单。我们创建了StockTrade_ClientSide的一个实例，并用带参数的构造函数设定了它的属性值。然后我们为参数建立了一个Vector，这与我们在前面的例子中所做的是一样的。如果运行这个例子，你应该会看到如下的输出：

```
Trade Description: Sell 350 of XYZ
```

让我们看看被传输的SOAP封套。封套的相关部分由trade元素开始，它表示要传递给executeStockTrade服务方法的自定义参数。分配给xsi:type的值是ns1:StockTrade。ns1是一个在父元素executeStockTrade中被声明为urn:BasicTradingService的名称空间标识符。而且StockTrade是自定义类型的名称。trade元素有三个子元素，每一个子元素对应于StockTrade自定义类型的一个属性。每个元素的名称也与该属性完全对应。这是很关键的，因为Apache SOAP要通过Java映射来寻找与Java类相关联的set方法。因此，封套中的名称一定要和类使用的名称相匹配。这些属性元素的类型都被显式地声明，而且这些类型也必须与对应的属性类型一致。

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
     <ns1:executeStockTrade xmlns:ns1="urn:BasicTradingService"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <trade xsi:type="ns1:StockTrade">
         <numShares xsi:type="xsd:int">350</numShares>
         <buy xsi:type="xsd:boolean">false</buy>
         <symbol xsi:type="xsd:string">XYZ</symbol>
      </trade>
     </ns1:executeStockTrade>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

现在让我们看看在GLUE中如何处理自定义类型，我们仍不做修改地使用BasicTradingService类。我们需要在IBasicTradingService 接口中加入executeStockTrade()方法。我们可以用前面介绍的那些方法来部署这个服务。这里是IBasicTradingService的新版本：

```
package javasoap.book.ch5;
public interface IBasicTradingService {
  int getTotalVolume(String[] symbols);
  String executeTrade(Object[] params);
  String[] getMostActive();
  String executeStockTrade(StockTrade_ClientSide trade);
}
```

由于我们不需要直接处理SOAP结构，因此使用GLUE API来编写访问服务的应用程序就变得很简单了。在这里我们也应该可以按同样的模式来做，但是有一个问题。IBasicTradingInterface接口的executeStockTrade()方法的参数是StockTrade_ClientSide的一个实例。而用于实现服务的BasicTradingService类的executeStockTrade方法使用StockTrade_ServerSide作为参数。这是设计导致的一个失配。默认情况下，GLUE会在客户端和服务器端寻找同一个类。假如我们在客户应用程序中用StockTrade_ServerSide类来取代StockTrade_ClientSide类，那么一切将会很完美。但我们不准备那样做，因此你只要相信它会工作得很好就行了（或者，你自己尝试一下当然更好）。我们将采取另外一种方式。

每当我开发分布式系统时，我会很高兴让客户端和服务器端代码共享Java接口。但是，我不喜欢共享Java类，尤其是当那些类中包含了特定于客户或服务器的代码的时候。对于这个例子，我们可以重写我们的股票交易类，删去那些客户和服务器功能代码，只剩下相关数据。我们可以让客户和服务器使用同一个包里的类，因此作为基本交易Web服务的实现者，我们应该分发这样一个包，让它包含用于开发者客户应用程序的类。这样做是可行的，但通常不会这样做。回到共享Java接口而不是类的观念上来吧，我们可以为交易数据建立一个接口，然后让服务器和客户类都实现那个接口。这是不在服务器和客户之间共享任何实际的可执行代码而是共享代码要素的一个漂亮而整洁的方法。再说一次，这个接口会存在于一

个对服务器和客户系统都可用的软件包中。GLUE声称将在它的下一个版本中支持这种机制（当读到这里时你可能已经在使用这个新的版本了）。

GLUE支持的另外一种机制也可以让你达到这个目的。这种机制不需要你修改服务器端，在客户端要做的工作也很简单。我们要为我们的客户例子创建一个新的包，这是因为在接下来的工作中我们会创建与前面使用的文件名称相同的文件。当在同一台计算机上运行客户和服务器时，新的软件包会防止我们覆盖文件。因此要注意这个例子是一个叫做javasoap.book.ch5.client的新软件包的一部分，而且我们将要创建的文件存在于这个软件包对应的目录中。

到目前为止我一直有意地避免讨论WSDL，尽管GLUE是完全建立在WSDL之上的。然而，本过程的第一个步骤就使用了GLUE的wsdl2java工具程序，它将生成基于某个WSDL文件的Java代码。那么这个WSDL是从哪里来的呢？它是由GLUE服务器自动产生的。基于GLUE API的客户端应用程序自始至终都在使用它。wsdl2java会生成绑定到服务的Java接口、一个辅助类、一个描述我们使用的自定义类型数据字段的Java数据结构类和一个映射文件。映射文件本质上是一个自定义类型的模式定义；它告诉GLUE该如何将Java数据结构类的字段映射到自定义类型的字段。GLUE使用了很多种机制来处理自定义类型映射；在本例中，映射文件显式地定义了映射，而不是依靠Java反射（reflection）来进行映射。

现在让我们来为客户应用程序生成这些文件。确认你的当前目录是javasoap.book.ch5.client软件包所在的目录，然后输入下面的命令：

```
wsdl2java http://georgetown:8004/glue/urn:BasicTradingService.wsdl
   -p javasoap.book.ch5.client
```

wsdl2java工具程序的参数是服务WSDL的完整URL，就像我们前面例子里在bing()方法中使用的一样。-p选项告诉工具程序生成代码所使用的本地软件包的名称：javasoap.book.ch5.client。wsdl2java的输出由四个文件组成。第一个是*IBasicTradingService.java*，包含用来绑定到服务的Java接口。在此之前我们一直是用手工编写这个文件的；现在让我们看看wsdl2java生成的这个文件：

```
// 由GLUE生成的
package javasoap.book.ch5.client;
```

```
public interface IBasicTradingService
  {
  String executeStockTrade( StockTrade_ServerSide arg0 );
  String[] getMostActive();
  int getTotalVolume( String[] arg0 );
  String executeTrade( Object[] arg0 );
  }
```

这个接口定义与我们自己编写的不同仅仅在于声明的包不同、方法的参数名称不同以及executeStockTrade使用了StockTrade_ServerSide而不是StockTrade_ClientSide作为参数。你应该还记得IBasicTradingService的早期版本不能工作就是因为GLUE默认对客户和服务器都使用同一个类。乍一看，这里StockTrade_ServerSide类的用法违背了我们原来说的要在客户代码与服务器代码间完全解除耦合的那个原则。但是记住软件包的声明是javasoap.book.ch5.client，所以这个Java接口并没有引用服务器所使用的那个相同的StockTrade_ServerSide类，因为它们属于不同的包。让我们看看由wsdl2java生成的*StockTrade_ServerSide.java*文件中的那个StockTrade_ServerSide类：

```
// 由GLUE生成的
package javasoap.book.ch5.client;
public class StockTrade_ServerSide
  {
  public int _shares;
  public boolean _buyOrder;
  public String _stock;
  }
```

这只是在服务器端所使用的javasoap.book.ch5包中对应类的一个镜像。然而，它的确正确地反映了数据值。我们将在客户应用程序中创建这个类的一个实例并传递给服务，因为这正是前面生成的那个IBasicTradingService接口的executeStockTrade()方法所要求的类型。

wsdl2java生成的第三个文件是*BasicTradingService.map*，它包含了前面提到的映射模式。你可以自己去看看这个映射文件的内容；它包含了定义客户端和服务器端Java之间字段映射的XML项。生成的最后一个文件是*BasicTradingServiceHelper.java*，它包含了一个完成bind()操作的辅助类。我不会用到那个辅助类，因此在这里我们将不理会它。

讲述一个过程要比实现这个过程麻烦得多。接下来让我们着手编写客户应用程序:

```
package javasoap.book.ch5.client;
import electric.registry.RegistryException;
import electric.registry.Registry;
import electric.xml.io.Mappings;
public class StockTradeClient2 {
   public static void main(String[] args) throws Exception
   {
      try {
        Mappings.readMappings("BasicTradingService.map");
        IBasicTradingService srv =
          (IBasicTradingService)Registry.bind(
          "http://georgetown:8004/glue/urn:BasicTradingService.wsdl",
          IBasicTradingService.class);
        StockTrade_ServerSide trade =
                      new StockTrade_ServerSide();
        trade._stock = "MINDSTRM";
        trade._buyOrder = true;
        trade._shares = 500;
        String desc = srv.executeStockTrade(trade);
        System.out.println("Trade Description is: " + desc);
      }
      catch (RegistryException e)
      {
         System.out.println(e);
      }
   }
}
```

在完成绑定操作之前的关键一步是要读取映射文件。这个工作通过调用electric.xml.io.Mappings类的readMappings()方法来完成,这个类是GLUE的一部分。在调用了bind()之后,我们简单地创建了StockTrade_ServerSide类的一个实例,为它的各个字段设置了值,并调用了executeStockTrade方法。运行这个应用程序的输出应该是:

```
Trade Description is: Buy 500 of MINDSTRM
```

现在让我们看看这个应用程序所产生的SOAP封套。和我们预期的一样,executeStockTrade元素使用了服务名作为名称空间限定。其子元素arg0是对一个名为id0的独立串行化元素的引用,正如GLUE为一个数组参数所产生的一样。这个例子向我们说明了GLUE为什么要产生一个名称空间限定符ns2,而它

看起来是在引用服务器端实现类所在的包。在这里该名称空间被用来限定xsi:type属性的值，这个值与实现类的名称StockTrade_ServerSide对应。id0元素的子元素中包含了自定义类型的数据字段。

```
<soap:Envelope
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <soap:Body>
      <n:executeStockTrade xmlns:n='urn:BasicTradingService'>
        <arg0 href='#id0'/>
      </n:executeStockTrade>
      <id0 id='id0' soapenc:root='0'
       xmlns:ns2='http://www.themindelectric.com/package/javasoap.book.ch5/'
       xsi:type='ns2:StockTrade_ServerSide'>
        <_shares xsi:type='xsd:int'>500</_shares>
        <_buyOrder xsi:type='xsd:boolean'>true</_buyOrder>
        <_stock xsi:type='xsd:string'>MINDSTRM</_stock>
      </id0>
    </soap:Body>
</soap:Envelope>
```

返回自定义类型

从服务方法调用返回自定义数据类型与传递自定义类型一样有用(也同样常见)。我们可以以此增强我们的股票服务，使它提供一个接受单个股票代码然后返回其当天最高价和最低价的方法。类HighLow_ServerSide和HighLow_ClientSide分别表示服务器端和客户端的最高/最低价。

```
package javasoap.book.ch5;
public class HighLow_ServerSide {
   public float _high;
   public float _low;
   public HighLow_ServerSide() {
   }
   public HighLow_ServerSide (float high, float low) {
      setHigh(high);
      setLow(low);
   }
   public float getHigh() {
```

```
        return _high;
    }
    public void setHigh(float high) {
        _high = high;
    }
    public float getLow() {
        return _low;
    }
    public void setLow(float low) {
        _low = low;
    }
}

package javasoap.book.ch5;
public class HighLow_ClientSide {
    public float _high;
    public float _low;
    public String toString() {
        return "High: " + _high +
          " Low: " + _low;
    }
    public HighLow_ClientSide() {
    }
    public float getHigh() {
        return _high;
    }
    public void setHigh(float high) {
        _high = high;
    }
    public float getLow() {
        return _low;
    }
    public void setLow(float low) {
        _low = low;
    }
}
```

为得到返回值，服务器端的类使用了带参数的构造函数；客户端的类则包含了一个toString()方法以使客户应用程序更容易显示从服务器返回的对象的内容。现在让我们在BasicTradingService类中添加一个叫做getHighLow()的新方法。这个方法接收一个字符串类型的股票代码，并返回HighLow_ServerSide类的一个实例。下面是这个包含了新方法的类，其中省略了保持不变的代码：

```
package javasoap.book.ch5;
public class BasicTradingService {
```

```
    public BasicTradingService() {
    }
    . . .
    . . .
    public HighLow_ServerSide getHighLow(String stock) {

        // 获取指定股票的最高/最低价
        return new HighLow_ServerSide((float)110.375,
                        (float)109.5);
    }
}
```

为了能在 Apache SOAP 中使用这个新的方法，我们需要使用一个修改后的部署描述文件来重新部署这个服务。我们要把 getHighLow 加入到方法的列表，而且在 mappings 部分为最高/最低价对象添加一个项目。第二个映射项定义了 HighLow 自定义类型，它的名称空间限定符使用了服务的名称 urn:BasicTradingService。这里是修改后的部署描述文件：

```
<isd:service
    xmlns:isd="http://xml.apache.org/xml-soap/deployment"
    id="urn:BasicTradingService">
  <isd:provider
     type="java"
     scope="Application"
     methods="getTotalVolume getMostActive executeTrade executeStockTrade
              getHighLow">
    <isd:java
       class="javasoap.book.ch5.BasicTradingService"
       static="false"/>
  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener
  </isd:faultListener>
  <isd:mappings>
    <isd:map
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       xmlns:x="urn:BasicTradingService" qname="x:StockTrade"
       javaType="javasoap.book.ch5.StockTrade_ServerSide"
      java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
       xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
    <isd:map
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       xmlns:x="urn:BasicTradingService" qname="x:HighLow"
       javaType="javasoap.book.ch5.HighLow_ServerSide"
      java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
```

```
        xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
    </isd:mappings>
  </isd:service>
```

现在我们创建一个客户应用程序来调用服务方法getHighLow并取得返回的最高/最低价对象。对于 Apache SOAP 客户，我们要使用 HighLow_ClientSide 类来表示返回值。下面就是这个新的应用程序:

```
package javasoap.book.ch5;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
public class HighLowClient
{
  public static void main(String[] args) throws Exception
  {
    URL url = new
       URL("http://georgetown:8080/soap/servlet/rpcrouter");
    Call call = new Call();
    SOAPMappingRegistry smr = new SOAPMappingRegistry();
    call.setTargetObjectURI("urn:BasicTradingService");
    call.setMethodName("getHighLow");
    call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
    call.setSOAPMappingRegistry(smr);
    BeanSerializer beanSer = new BeanSerializer();
    // 映射High/Low类型
    smr.mapTypes(Constants.NS_URI_SOAP_ENC,
            new QName("urn:BasicTradingService", "HighLow"),
            HighLow_ClientSide.class, beanSer, beanSer);
    String stock = "XYZ";
    Vector params = new Vector();
    params.addElement(new Parameter("stock",
                          String.class, stock, null));
    call.setParams(params);
    Response resp;
    try {
      resp = call.invoke(url, "");
      Parameter ret = resp.getReturnValue();
      HighLow_ClientSide hilo =
             (HighLow_ClientSide)ret.getValue();
      System.out.println(hilo);
    }
```

```
        catch (SOAPException e) {
          System.err.println("Caught SOAPException (" +
                             e.getFaultCode() + "): " +
                             e.getMessage());
        }
      }
    }
```

smr.MapTypes()方法将HighLow自定义类型映射到Java类HighLow_ClientSide。既然我们的类遵循了JavaBeans属性存取器模式，那么和以前一样，我们可以使用Apache的BeanSerializer来在XML与Java之间转换。我们建立了一个叫做stock的字符串参数来向getHighLow方法传递值（尽管在服务器代码中我们并没有真正用到它）。在调用了这个方法之后，我们将resp.getReturnValue()的返回值的类型转换成了HighLow_ClientSide的一个实例。然后我们把返回的参数值ret传递给System.out.println()方法来显示它。因为我们已经在HighLow_ClientSide类中实现了toString()方法，因此我们现在已完成了所有工作。当你运行这个例子的时候，会得到下面这样的输出：

```
High: 110.375 Low: 109.5
```

下面是这个方法调用所返回的SOAP封套。return元素的类型被设为HighLow，它用urnBasicTradingService作为名称空间限定。HighLow的属性被作为return元素的子元素，其类型被设为浮点数并包含了对应的值。

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"> <SOAP-ENV:Body>
    <ns1:getHighLowResponse
      xmlns:ns1="urn:BasicTradingService"
      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <return xsi:type="ns1:HighLow">
         <low xsi:type="xsd:float">109.5</low>
         <high xsi:type="xsd:float">110.375</high>
      </return>
   </ns1:getHighLowResponse>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

要使用GLUE来返回HighLow，我们将会采取与传递自定义类型时相同的步骤。我们需要再一次将客户端的程序单独地放到另一个包javasoap.book.ch5.client中去。重新启动BasicTradingApp应用程序以部署该服务。现在在javasoap.book.ch5.client包对应的目录下运行wsdl2java工具程序:

```
wsdl2java http://georgetown:8004/glue/urn: BasicTradingService.wsdl
    -p javasoap.book.ch5.client
```

下面的代码是wsdl2java所产生的新的IBasicTradingService接口。getHighLow()方法返回HighLow_ServerSide的一个实例；记住这个类是由GLUE作为javasoap.book.ch5.client包的一部分生成的，而不是给我们的服务器类BasicTradingApp使用的。

```
// 由GLUE生成的
package javasoap.book.ch5.client;
public interface IBasicTradingService
  {
  HighLow_ServerSide getHighLow( String arg0 );
  String executeStockTrade( StockTradeServer arg0 );
  String[] getMostActive();
  int getTotalVolume( String[] arg0 );
  String executeTrade( Object[] arg0 );
  }
```

下一个类HighLow_ServerSide是一个简单的Java数据结构，它反映了将要被映射到HighLow自定义类型的数据字段。我手工添加了toString()方法以使得显示结果更容易。

```
package javasoap.book.ch5.client;
public class HighLow_ServerSide {
  public float _high;
  public float _low;
  public String toString() {
     return "High: " + _high +
        " Low: " + _low;
  }
}
```

现在让我们使用GLUE创建一个客户应用程序来调用getHighLow服务方法并显示所产生的返回值的内容。不需要做很多的工作，真的；我们只需要读取映射文

件，完成绑定，然后在被绑定的接口中调用getHighLow方法。javasoap.book.ch5.client.HighLow_ServerSide返回的实例被传给System.out.println()来显示，正如在Apache SOAP例子中一样，该类的toString()方法组装了要显示的字符串。

```
package javasoap.book.ch5.client;
import electric.registry.RegistryException;
import electric.registry.Registry;
import electric.xml.io.Mappings;
public class HighLowClient2 {
   public static void main(String[] args) throws Exception
   {
      try {
        Mappings.readMappings("BasicTradingService.map");
        IBasicTradingService srv = (IBasicTradingService)Registry.bind(
          "http://georgetown:8004/glue/urn:BasicTradingService.wsdl",
          IBasicTradingService.class);
        HighLow_ServerSide hilo = srv.getHighLow("ANY");
        System.out.println(hilo);
      }
      catch (RegistryException e)
      {
         System.out.println(e);
      }
   }
}
```

这里是从服务器返回的SOAP封套。你现在应该可以看得懂它。与返回数组时的情况一样，GLUE使用了对该自定义类型的一个独立串行化实例的引用。

```
<soap:Envelope
 xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
 xmlns:xsd='http://www.w3.org/2001/XMLSchema'
 xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
 xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
 soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <soap:Body>
      <n:getHighLowResponse xmlns:n='urn:BasicTradingService'>
        <Result href='#id0'/>
      </n:getHighLowResponse>
      <id0 id='id0' soapenc:root='0'
        xmlns:ns2='http://www.themindelectric.com/package/javasoap.book.ch5/'
        xsi:type='ns2:HighLow_ServerSide'>
        <_high xsi:type='xsd:float'>110.375</_high>
```

```
            <_low xsi:type='xsd:float'>109.5</_low>
          </id0>
        </soap:Body>
      </soap:Envelope>
```

在本章中，我们深入介绍了数组和自定义数据类型的使用，介绍了在Apache SOAP和GLUE中如何支持这些结构。你可能会在它们或者别的SOAP实现中找到其他有用的数据类型。这些类型可能包括Java Vector和Hashtable类、Java集合以及其他很多已经被普遍使用的Java类。

我们看到Apache SOAP和GLUE使用了不同的方式来处理复杂类型，而且它们都工作得很不错。然而，仍然会有可能出现这样的情况：你需要使用一种自定义类型，但使用你的SOAP实现所提供的办法不能够对它进行串行化。出现这种情形多半仅仅是因为它不支持某个特定类型的数据(比如多维的或稀疏的数组)，或者是与你的应用程序有关的其他原因。我们将在下一章中解决这个问题。

第六章

自定义串行化

也许只需要使用你的SOAP实现所支持的那些数据类型，你的Web服务和客户应用程序就能工作得很好。毕竟，我们正在关注的技术已经通过对数组和自定义类型的支持为我们提供了很好的灵活性。但在另一方面，有时候你依然会需要对某些数据进行串行化，而这种串行化是无法通过前面所介绍的那些技术来简单实现的。也许你想要采用一种与现有的应用程序更一致的方式来串行化一个Java类的数据，即使该编码方式不是工业标准。或者也许你想要为某种在SOAP规范中描述了但没有被你的SOAP实现所支持的结构增加你自己的支持。

你使用的SOAP实现可能会强迫你为某种类型的对象编写自定义串行化器。举例来说，Apache提供了Bean串行化器；它可能对其他的某些特殊情况提供了串行化器，但就是没有你所需要的那一种。因此在这种情况下，你就需要针对该实现提供自定义的串行化器。

考虑将被串行化的数据的用途也是很重要的。Java中那些关于实现Serializable接口的规则在这里同样用得上。举例来说，不管是不是使用SOAP，串行化一个I/O流都是不明智的。对任何类型的分布式系统都是这样。在盲目地将它传送给另一方之前，你必须先了解数据的语义。

在本章中，我们将研究几个自定义串行化的例子，并介绍Apache SOAP和GLUE两者所使用的技术和API。与前面的几章不同，本章将不对这两种技术采用重复

的例子。到目前为止，你应该已经充分熟悉 Apache SOAP 和 GLUE，并对如何使用它们有自己的见解了。

自定义类型的编码

让我们开始吧，先使用非标准的串行化来处理一个自定义的数据类型。我们将使用 Apache SOAP 来开发这个例子．想像我们正在开发一个被当作转发性代理来使用的Web服务，它用来连接到一个专有的股市数据来源服务。我们想要将已经按照数据来源所使用的格式串行化好的数据传送给服务。是的，又是一个假想出来的例子！但是到你完成它的时候，我敢打赌你会明白这个技术的很多用处。服务被设计为允许客户应用程序将数据以认可的格式传递到下游的数据来源服务，而不必修改 Web 服务本身。因此我们的 Web 服务不需要知道被串行化数据的格式；这个格式只需要被客户应用程序和数据来源理解就行了。图 6-1 显示了这个体系结构。两个客户应用程序都使用相同的服务并调用相同的方法。然而，它们对要传输到数据来源的数据所使用的串行化格式可能相同，也可能不同。

图 6-1：转发服务

假设数据来源所接受的默认数据格式是一个线性的自描述格式。这个格式是一个任意长度的字符串，它包含了一个表示命名数据字段数量的数字和它们的字符串值。这种格式能接受 1 到 9 个字段。消息的第一个字符表示字段的数目，它可以是 1 到 9 之间的任何 ASCII 字符。余下的每个字段都由一个名称和一个值组成，名称与值之间用一个冒号隔开，而在值的后面加上另一个冒号。这里是一个包含名为SYMBOL和PRICE的两个字段的消息例子，其对应的值分别是ZZ和110.5:

```
2SYMBOL:ZZ:PRICE:110.5:
```

这是我们想要用来串行化消息的格式。但是首先，让我们创建一个新的服务，将它称为urn:DataFeedService，它包含一个叫sendMessage的方法。这个方法只接受一个消息参数，并且只返回一个消息参数。因此在我们为服务定义Java类之前，需要先定义消息的Java类。DataFeedMessage类描述了一个用于这个服务的消息。它包含了一个名/值对的列表，用来容纳那些字段名和字段的数据值。

```
package javasoap.book.ch6;
import java.util.*;
public class DataFeedMessage {
   Hashtable _fields = new Hashtable();
   public DataFeedMessage() {
   }
   public void addField(String name, String value) {
      _fields.put(name, value);
   }
   public Enumeration getFieldNames() {
      return _fields.keys();
   }
   public String getFieldValue(String name) {
      return (String)_fields.get(name);
   }
   public String toString() {
      String cnt = String.valueOf(_fields.size());
      StringBuffer msg = new StringBuffer();
      msg.append(cnt);
      msg.append(":");
      for (Enumeration e =_fields.keys(); e.hasMoreElements();) {
         String name = (String)e.nextElement();
         String value = (String)_fields.get(name);
         String section = name + ":" + value + ":";
         msg.append(section);
      }
      return msg.toString();
   }
   public void parseFormattedMessage(String msg) {
      _fields.clear();
      StringTokenizer st = new StringTokenizer(msg, ":");
      String token = st.nextToken();
      int cnt = new Integer(token).intValue();
      for (int i = 0; i < cnt; i++) {
         String name = st.nextToken();
         String value = st.nextToken();
         addField(name, value);
      }
   }
```

字段都被保存在_fields变量中，该变量是java.util.Hashtable的一个实例。addField()方法允许你通过提供名称和值作为参数来添加一个字段，然后将它们加入到一个散列表中，其中名称被作为键。getFieldNames()和getFieldValue()方法用于访问该散列表中的名称和值，它们隐藏了这个类在内部使用了java.util.Hashtable类这一事实。最后两个方法是最有意思的。toString()方法使用我们前面定义时所采用的格式，在消息中返回一个包含所有字段及其值的字符串。parseFormattedMessage()方法接受一个如前所述格式的字符串作为参数，并根据它的内容组装出包含内容的内部散列表。在串行化或反串行化我们自定义的消息类型的时候，这两个方法将会很有用。

现在我们可以着手为这个新的服务建立Java类了。我们将把这个类称为DataFeedService，该类将包含一个叫做sendMessage()的方法，这个方法接受DataFeedMessage的一个实例作为参数。同时这个方法也返回DataFeedMessage的一个实例。下面就是那个DataFeedService类：

```
package javasoap.book.ch6;
public class DataFeedService {
   public DataFeedService() {
   }
   public DataFeedMessage sendMessage(DataFeedMessage msg) {
      // 在这里我们可以通过调用msg.toString()得到格式化后的消息
      String stock = msg.getFieldValue("SYMBOL");
      String request = msg.getFieldValue("REQUEST");
      DataFeedMessage res = new DataFeedMessage();
      res.addField("SYMBOL", stock);
      if (request.equals("PRICE"))
         res.addField("PRICE", "123.5");
      return res;
   }
}
```

我们在这个例子中所做的全部工作就是从数据来源消息中获取股票代码，把它作为返回的股票代码值，同时如果REQUEST字段中有一个值是PRICE，则在返回值中添加一个价格字段。因此sendMessage()方法使用了输入参数中的一些数据来产生一个DataFeedMessage类型的返回值。

因为这个类没有遵循JavaBeans中的方法命名模式，我们就不能使用由Apache SOAP提供的BeanSerializer来串行化和反串行化我们所建立的数据来源消息

实例。因此我们需要建立自己的串行化器。一个用来串行化对象的自定义Java类必须实现org.apache.soap.util.xml.Serializer接口；为实现这个接口，我们要编写一个marshall()方法。同样，一个用来反串行化对象的Java类也必须实现org.apache.soap.util.xml.Deserializer接口并提供unmarshall()方法。只创建一个类来完成这两种功能是完全合理的，因此我们将采取这种方式。我们称自定义的串行化器类为DataFeedMessageSerializer。

```
package javasoap.book.ch6;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
import org.apache.soap.util.*;
import org.apache.soap.rpc.SOAPContext;
import org.w3c.dom.*;
import java.io.*;
import java.util.*;
public class DataFeedMessageSerializer
                  implements Serializer, Deserializer {

  public DataFeedMessageSerializer() {
  }
  public void marshall(String inScopeEncStyle, Class javaType,
                 Object src, Object context, Writer sink,
                 NSStack nsStack, XMLJavaMappingRegistry xjmr,
                 SOAPContext ctx)
        throws IllegalArgumentException, IOException  {

    if(!javaType.equals(DataFeedMessage.class)) {
      throw new IllegalArgumentException(
        "Can only serialize javasoap.book.ch6.DataFeedMessage instances");
    }

    nsStack.pushScope();

    if (src != null)
    {
      SoapEncUtils.generateStructureHeader(inScopeEncStyle,
                                           javaType, context,
                                           sink, nsStack, xjmr);
      DataFeedMessage msg = (DataFeedMessage)src;
      String data = msg.toString();
      sink.write(data);
      sink.write("</" + context + '>');
    }
```

```
    else
    {
      SoapEncUtils.generateNullStructure(inScopeEncStyle,
                                         javaType, context,
                                         sink, nsStack, xjmr);
    }
    nsStack.popScope();
  }
  public Bean unmarshall(String inScopeEncStyle, QName
                 elementType, Node src,
                 XMLJavaMappingRegistry xjmr, SOAPContext ctx)
          throws IllegalArgumentException {

      Element elem = (Element)src;
      String value = DOMUtils.getChildCharacterData(elem);

      DataFeedMessage msg = null;
      if(value!=null && !((value=value.trim()).equals("")) {
         msg = new DataFeedMessage();
         msg.parseFormattedMessage(value);
      }
      return new Bean(DataFeedMessage.class, msg);
    }
  }
```

让我们先来看看marshall()，它串行化了DataFeedMessage的一个实例。由于自定义的串行化器不能串行化任何其他对象的实例，所以它首先要校验javaType参数值是DataFeedMessage.class类型的。如果类型不吻合，marshall()将抛出一个IllegalArgumentException异常并给出适当的描述（到目前为止我们还没有讨论异常，因此你可能不知道抛出一个异常后会出现什么结果。我们将在第七章中深入研究这个问题）。随后，我们调用了nsStack.pushScope()。作为参数传递的nsStack变量表示被串行化的SOAP消息的层次化名称空间（hierarchical namespace）。这个对象是一个名称空间定义的堆栈。每当串行化器的marshall()方法被调用时，我们就向这个堆栈中压入其名称空间作用域。这项技术允许我们在正确的层次上声明名称空间。在从marshall()方法返回之前，一定要弹出堆栈。如果没有这样做，就会存在用当前层次之上的XML元素的名称空间声明覆盖我们的名称空间的危险。

既然已经压入了名称空间作用域，我们还要确认要串行化的对象不为空。如果src是空的，就要调用SoapEncUtils.generateNullStructure()方法来处理空

元素的创建。如果 s r c 不为空（这是更有趣的情况），我们就要调用 SoapEncUtils.generateStructureHeader()，完成创建一个SOAP消息的第一个步骤。这个方法产生元素标签，并写入必要的属性及其对应值。我们不必去管这些问题；我们的工作是要处理对实际数据的串行化而不是XML标签细节。好吧，这基本上是事实。我们将在最后做一些清除工作，你马上就会看到。

既然我们已经知道 src 是正确的类型，我们就将它的类型转换为 DataFeed-Message，并在 msg 变量中保存一份对它的引用。我们现在要做的是调用 msg.toString()方法，该方法会以要求的格式返回数据来源消息中的数据。这个消息以适当的格式保存在data变量中。marshall()方法使用一个称为sink的参数，该参数是 java.io.Writer 的一个实例。SOAP 封套的所有串行化工作都在此处完成。在这里，实际被串行化的数据被放在了 sink 变量中。以 data 为参数调用 sink.write()方法，就将对数据来源消息格式化后的数据写入到了 sink 里。最后一步是为这个元素写终止标签，尽管写开始元素标签的工作已经被自动完成。我们写关闭标签时需要元素的名称；这个名称可以由 context 对象得到，而该对象是作为 marshall()方法的一个参数被传递进来的。我们再一次调用了 sink.write()方法来关闭元素。因为context中实现了toString()方法来返回被串行化的元素的名称，所以我们这样做是可以的。差不多了，再调用 nsStack.popStack()方法弹出堆栈，这个方法就编写完成了。

现在让我们看看反串行化的步骤：将已编码的数据转换为某个Java 类的实例。这个过程由串行化器的 unmarshall()方法完成。反串行化需要使用 Apache SOAP 提供的一些处理 XML 的 API。我们将 src 参数转换成 org.w3c.dom.Element 的一个实例，这个类常常用来表示 XML 元素。然后，我们想要得到某元素被格式化后的数据。那是用本节开始时所介绍的格式来格式化的数据来源消息。这个工作是通过将变量 elem 传递给 org.apache.soap.util.xml.DOMUtils 类的 getChildCharacterData()方法来完成的。这个调用返回的字符串值保存在 value 变量中，它包含了格式化后的数据来源消息。当然，先确认该消息中是否真的含有数据是一个好主意，因此我们首先检查了value是否为null或者空字符串。如果包含数据，我们就建立DataFeedMessage的一个新实例并将这个数据传递给它的 parseFormattedMessage()方法。现在，我们已经成功地反串行化了

数据并用它建立了DataFeedMessage的一个实例。unmarshall()方法需要返回org.apache.soap.util.Bean的一个实例（注1）；这个类用来关联一个Java对象和它的Java类。既然你能用一个对象的getClass()方法来获知它的类，你可能会不理解为什么我们还要这样做。这是因为，如果一个对象是由某个映射类的子类实现的，那么getClass()方法的结果就不准确，如果该对象为null，则使用这个方法就是不可能的。因此我们的最后一个步骤就是创建org.apache.soap.util.Bean的一个实例，使用DataFeedMessage.class和msg分别作为它的构造函数的第一个和第二个参数。

现在我们已经为我们自定义的串行化器实现了串行化和反串行化功能。接下来我们要部署urn:DataFeedService服务。为了部署该服务，需要在部署描述文件中包含一个适当的映射项，以便我们自定义类型的实例能被映射到正确的Java类并与我们刚创建的自定义串行化器相关联。下面是这个部署描述文件：

```
<isd:service
    xmlns:isd="http://xml.apache.org/xml-soap/deployment"
    id="urn:DataFeedService">
  <isd:provider
     type="java"
     scope="Application"
     methods="sendMessage">
    <isd:java
      class="javasoap.book.ch6.DataFeedService"
      static="false"/>
  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener
  </isd:faultListener>
  <isd:mappings>
    <isd:map
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:x="urn:DataFeedService" qname="x:DataFeedMessage"
      javaType="javasoap.book.ch6.DataFeedMessage"
        java2XMLClassName="javasoap.book.ch6.DataFeedMessageSerializer"
        xml2JavaClassName="javasoap.book.ch6.DataFeedMessageSerializer"/>
  </isd:mappings>
</isd:service>
```

注1：不要因为这个类名中有Bean就被它迷惑住了。它与我们在前面章节中用到的BeanSerialier没有特别的关系；它只是来自某个其他项目的一个名称。

在映射段中有一项为DataFeedMessage，它使用服务本身的名称作为名称空间限定。我们将 java2XMLClassName 和 xml2JavaClassName 属性的值都指定为 javasoap.book.ch6.DataFeedMessageSerializer，这个值将我们自定义的串行化器类告知系统。现在可以部署服务了。

这里是一个可以访问这个服务的客户应用程序。其步骤类似于我们在前一章里使用的：我们将自定义类型映射为bean，并使用 DataFeedMessageSerializer 来串行化和反串行化该对象。

```
package javasoap.book.ch6;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.util.xml.*;
public class DataFeedClient {
   public static void main(String[] args)
      throws Exception {

      URL url =
        new URL(
          "http://georgetown:8080/soap/servlet/rpcrouter");

      Call call = new Call();

      SOAPMappingRegistry smr = new SOAPMappingRegistry();
      call.setTargetObjectURI("urn:DataFeedService");
      call.setMethodName("sendMessage");
      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      call.setSOAPMappingRegistry(smr);
      DataFeedMessageSerializer msgSer =
                  new DataFeedMessageSerializer();
      // 映射数据来源消息类型
      smr.mapTypes(Constants.NS_URI_SOAP_ENC,
            new QName("urn:DataFeedService", "DataFeedMessage"),
            DataFeedMessage.class, msgSer, msgSer);
      DataFeedMessage msg = new DataFeedMessage();
      msg.addField("SYMBOL", "XYZ");
      msg.addField("REQUEST", "PRICE");
      Vector params = new Vector();
      params.addElement(new Parameter("msg",
                  DataFeedMessage.class, msg, null));
```

```
        call.setParams(params);

        try {
            Response resp = call.invoke(url, "");
            Parameter ret = resp.getReturnValue();
            Object value = ret.getValue();
            System.out.println(value);
        }
        catch (SOAPException e) {
            System.err.println("Caught SOAPException (" +
                               e.getFaultCode() + "): " +
                               e.getMessage());
        }
    }
}
```

我们在变量msgSer中建立了DataFeedMessageSerializer的一个实例。当我们调用smr.mapTypes()来将我们的自定义类型映射到DataFeedMessage类时，我们对它的串行化器和非串行化器两个参数都使用了msgSer。因此当我们调用sendMessage服务方法时，DataFeedMessageSerializer就被用来串行化数据来源消息，以及反串行化来自服务方法的返回值。接下来，我们先创建了DataFeedMessage的一个实例，然后向该实例中添加了两个字段：SYMBOL和REQUEST，它们的值分别为XYZ和PRICE。现在我们组合服务方法的参数并调用该方法。来自sendMessage服务方法的返回值也是一个数据来源消息，因此它被反串行化到DataFeedMessage的一个实例中。因为可以用toString()方法显示它的内容，所以就不再需要对ret.getValue()的返回值进行类型转换。如果你运行这个例子，应该能得到如下输出：

```
2:SYMBOL:XYZ:PRICE:123.5:
```

这是返回的DataFeedMessage实例被格式化后的数据；服务器把传给它的股票代码连同一个值为123.5的价格字段一起放到了它的返回值里。让我们看看被发送到服务器的SOAP封套：

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
     <ns1:sendMessage
```

```
        xmlns:ns1="urn:DataFeedService"
        SOAP-ENV:encodingStyle=
          "http://schemas.xmlsoap.org/soap/encoding/">
       <msg
       xsi:type="ns1:DataFeedMessage">2:REQUEST:PRICE:SYMBOL:XYZ:
       </msg>
     </ns1:sendMessage>
   </SOAP-ENV:Body>
 </SOAP-ENV:Envelope>
```

msg元素的类型被设为DataFeedMessage，名称空间用urn:DataFeedService限定。你可以看到数据按照我们的格式被格式化了。现在让我们看看从服务方法返回的SOAP封套，它也将返回值的类型设成了DataFeedMessage，名称空间也使用urn:DataFeedService限定。因此我们已经能够使用我们自己的自定义串行化类来在客户和服务器上处理数据来源消息的编组（marshalling）和解组（unmarshalling）了。

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
     <ns1:sendMessageResponse
       xmlns:ns1="urn:DataFeedService"
       SOAP-ENV:encodingStyle=
         "http://schemas.xmlsoap.org/soap/encoding/">
         <return
         xsi:type="ns1:DataFeedMessage">2:PRICE:123.5:SYMBOL:XYZ:
         </return>
     </ns1:sendMessageResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

串行化稀疏数组

我们曾在第三章中讨论过稀疏数组。然而，Apache SOAP和GLUE都不支持它。那么现在让我们自己来做。

从头来编写稀疏数组的串行化器可能会花费许多功夫，所以我们通过修改来自Apache SOAP发布包中的org.apache.soap.encoding.soapenc. ArraySer-

ializer来完成。我们将修改ArraySerializer的代码，生成一个叫做javasoap.book.ch6.SparseArraySerializer的新类。这里是整个类的代码，但是不要被它所迷惑。因为代码的大部分都来自于ArraySerializer类。我们将只解释用来操作稀疏数组的修改部分。

```
package javasoap.book.ch6;
import org.apache.soap.encoding.soapenc.*;
import java.beans.*;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import org.w3c.dom.*;
import org.apache.soap.util.*;
import org.apache.soap.util.xml.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
public class SparseArraySerializer implements Serializer, Deserializer
{
   public SparseArraySerializer() {
   }
   public void marshall(String inScopeEncStyle,
               Class javaType, Object src,
               Object context, Writer sink, NSStack nsStack,
               XMLJavaMappingRegistry xjmr, SOAPContext ctx)
       throws IllegalArgumentException, IOException {
      nsStack.pushScope();
      String lengthStr = src != null
                       ? Array.getLength(src) + ""
                       : "";
      Class componentType = javaType.getComponentType();
      QName elementType = xjmr.queryElementType(componentType,
                                Constants.NS_URI_SOAP_ENC);
      if (src == null) {

         SoapEncUtils.generateNullArray(inScopeEncStyle,
                                        javaType,
                                        context,
                                        sink,
                                        nsStack,
                                        xjmr,
                                        elementType,
                                        lengthStr);
      }
      else {
```

```
                SoapEncUtils.generateArrayHeader(inScopeEncStyle,
                                                 javaType,
                                                 context,
                                                 sink,
                                                 nsStack,
                                                 xjmr,
                                                 elementType,
                                                 lengthStr);
            sink.write(StringUtils.lineSeparator);
            int length = Array.getLength(src);
            for (int i = 0; i < length; i++) {

              nsStack.pushScope();
              Object value = Array.get(src, i);
              // 既然这是在串行化一个稀疏矩阵,
              //我们就希望只在元素存在时才进行串行化工作
              if (value != null) {
               Class actualComponentType = value.getClass();
                // 使用临时的sink,
                // 以便我们在输出之前可以修改它
                StringWriter sw = new StringWriter();
                xjmr.marshall(inScopeEncStyle,
                         actualComponentType, value, "item",
                    sw, nsStack, ctx);

               sink.write("<item ");
                sink.write(nsStack.getPrefixFromURI(
                     Constants.NS_URI_SOAP_ENC) +
                    ":position=\"[" + i + "]\" ");
               sink.write(sw.toString().substring(6));
               sink.write(StringUtils.lineSeparator);
              }
                nsStack.popScope();
             }
             sink.write("</" + context + '>');
            }
            nsStack.popScope();
          }
          public Bean unmarshall(String inScopeEncStyle,
                         QName elementType, Node src,
                         XMLJavaMappingRegistry xjmr, SOAPContext ctx)
                    throws IllegalArgumentException {

            Element root = (Element)src;
            String name = root.getTagName();
            QName arrayItemType = new QName("", "");
            Object array = getNewArray(inScopeEncStyle, root,
```

```
                                arrayItemType, xjmr);
        if (SoapEncUtils.isNull(root)) {
            return new Bean(array.getClass(), null);
        }
        Element tempEl = DOMUtils.getFirstChildElement(root);
        while (tempEl != null) {
            String declEncStyle = DOMUtils.getAttributeNS(tempEl,
                    Constants.NS_URI_SOAP_ENV,
                    Constants.ATTR_ENCODING_STYLE);
            String actualEncStyle = declEncStyle != null
                                ? declEncStyle
                                : inScopeEncStyle;
            QName declItemType =
                    SoapEncUtils.getAttributeValue(tempEl,
                    Constants.NS_URI_CURRENT_SCHEMA_XSI,
                    Constants.ATTR_TYPE,
                    "array item", false);
            QName actualItemType = declItemType != null
                                ? declItemType
                                : arrayItemType;
            Bean itemBean = xjmr.unmarshall(actualEncStyle,
                                actualItemType,
                                tempEl, ctx);
            // 取得在数组中的位置
            String pos = DOMUtils.getAttributeNS(tempEl,
                                inScopeEncStyle, "position");

            int right = pos.indexOf(']');
            String substr = pos.substring(1, right);
            int idx = Integer.parseInt(substr);
            Array.set(array, idx, itemBean.value);
            tempEl = DOMUtils.getNextSiblingElement(tempEl);
        }
        return new Bean(array.getClass(), array);
    }
    public static Object getNewArray(String inScopeEncStyle,
                    Element arrayEl,
                    QName arrayItemType,
                    XMLJavaMappingRegistry xjmr)
            throws IllegalArgumentException {
        QName arrayTypeValue =
                SoapEncUtils.getAttributeValue(arrayEl,
                Constants.NS_URI_SOAP_ENC,
                Constants.ATTR_ARRAY_TYPE,
                "array", true);
        String arrayTypeValueNamespaceURI =
                arrayTypeValue.getNamespaceURI();
        String arrayTypeValueLocalPart =
```

```
        arrayTypeValue.getLocalPart();
int leftBracketIndex =
        arrayTypeValueLocalPart.lastIndexOf('[');
int rightBracketIndex =
        arrayTypeValueLocalPart.lastIndexOf(']');
if (leftBracketIndex == -1
  || rightBracketIndex == -1
  || rightBracketIndex < leftBracketIndex) {
    throw new IllegalArgumentException(
            "Malformed arrayTypeValue '" +
             arrayTypeValue + "'.");
}
String componentTypeName =
        arrayTypeValueLocalPart.substring(0,
        leftBracketIndex);
if (componentTypeName.endsWith("]")) {
   throw new IllegalArgumentException(
        "Arrays of arrays are not " +
        "supported '" + arrayTypeValue + "'.");
}
arrayItemType.setNamespaceURI(arrayTypeValueNamespaceURI);
arrayItemType.setLocalPart(componentTypeName);
int length = DOMUtils.countKids(arrayEl,
    Node.ELEMENT_NODE);
String lengthStr =
    arrayTypeValueLocalPart.substring(leftBracketIndex + 1,
                                      rightBracketIndex);
if (lengthStr.length() > 0) {
   if (lengthStr.indexOf(',') != -1) {
      throw new IllegalArgumentException(
           "Multi-dimensional arrays are " +
           "not supported '" + lengthStr + "'.");
   }
   try {
      int explicitLength = Integer.parseInt(lengthStr);
      length = explicitLength;
   }
   catch (NumberFormatException e) {
      throw new IllegalArgumentException(
             "Explicit array length is not a " +
             "valid integer '" + lengthStr + "'.");
   }
}
Class componentType = xjmr.queryJavaType(arrayItemType,
                inScopeEncStyle);
return Array.newInstance(componentType, length);
}
}
```

首先让我们来看marshall()方法的代码。当数组被串行化时就会调用这个方法。稀疏数组的串行化也不需要对数组的每一个元素特殊对待。惟一的变化就是对数组中的每一项增加了一个position属性来表示该元素在数组中的位置。除此之外，再没有额外的属性了，属性值也和串行化普通数组时所使用的完全一样。position元素需要用名称空间http://schemas.xmlsoap.org/soap/encoding/来限定。我们还希望串行化器跳过那些为null的数组元素，而不是将它们也串行化。

看看marshall()中调用Array.getLength()来确定数组长度的那段代码。它返回的值保存在一个整型变量length中，该变量将用于后面的循环。在这个循环中，代码对数组的每一个元素都进行了处理（注2）。我们分别为每一个元素建立了一个名为value的Object类型的引用，并检查该对象是否为null。如果它为null，则我们不管它，继续循环中的下一个元素。如果它不为null，我们就先确定这个value对象的运行时类。然后，我们需要为数组项编写元素标记，但Apache SOAP没有为串行化元素的属性提供某种机制。既然我们必须为稀疏数组的每项元素都包含一个position属性，这就出现了一个问题。不过，我们仍然需要利用Apache SOAP代码来写入元素及其值。

让我们先关注后面的部分。首先我们创建了java.io.StringWriter的一个实例sw。通常情况下，我们会将sink传递到xjmr.marshall()方法，由该方法串行化数组元素并将它写入sink。而在这里，我们没有用sink，而是传递了StringWriter的一个实例sw。这将存储数组项被串行化后的形式，我们不久将会用到它。传递给xjmr.marshall()的第四个参数是"item"，它是元素的标签名。既然我们希望接下来就将它写进sink，因此我们就可以通过使用"<item"作为参数调用sink来自己完成这个工作。在写入了元素的开始标签之后，下面我们就可以向sink中添入属性/值对了。我们需要对该属性进行名称空间限定，因此我们使用nsStatck.getPrefixFromURI()方法来查找用于SOAP编码名称空间的名称空间ID。常量Constants.NS_URI_SOAP_ENC用来表示那个名称空间，所以我们

注2：实际上，这种方式可能并不是在Java中表示稀疏数组的最好方式，因为它需要处理每一个元素。对于一个非常稀疏的巨大数组，我们可以使用更智能的表示方法来实现它。

将它作为参数传递给getPrefixFromURI()。现在我们来组建完整的属性/值对，我们以position属性作为开始，该属性使用了我们刚才得到的名称空间ID作为限定。我们知道属性名称是position，而数组项的索引与循环变量i的值对应。所有这些元素都被格式化为与它们的值相结合，并传递给sink.write()方法。由于position属性的值被写在一对方括号中，我们就要使用转义序列向字符串中写入方括号。

然后，我们需要将已被串行化的数据从临时的StringWriter写入到sink中。但是我们不必再写入元素的开始标签，因为我们已经向sink中写过了这个标签和它的属性。由于我们用了相同的标签名item，因此我们可以在StringWriter实例中跳过该数据的最初六个字符并获取其余的部分(我们想要跳过的那六个字符是左尖括号<、加上item，再加上一个尾缀空格)。因此我们调用sw.toString().substring(6)并将它传递到sink.write()。我们曾设法将属性写入到项目元素之中，而且现在我们仍然将利用Apache SOAP API来做大部分的工作。marshall()方法中代码的剩余部分向sink中添加了一个换行符和一个关闭标签，这样就完成了数组元素的串行化工作。

下面我们来看看用来反串行化稀疏数组的unmarshall()方法。在这种情况下，我们没有多少工作要做。几乎所有代码的工作都集中在确定数组的大小和类型以及定位一个适当的数组实例上了。我们仅仅想要确定position属性的值，以便能将要反串行化的项放入数组中的适当位置。可以通过调用DOMUtils.getAttributeNS来得到该属性的字符串值。我们将XML文档的元素、当前作用域的编码形式以及属性的名称传递给该方法。该元素在前面已被保存在变量tempEl中。编码形式可以在传递给unmarshall()方法的inScopeEncStyle参数中找到。而且，我们知道属性名称是position。它的属性值将被保存在一对封闭方括号里的整数中。因此我们要从方括号里得到它的子字符串，并将它传递到Integer.parseInt()方法，从而得到一个表示数组元素位置的整数值。这个值将被作为第二个参数传递到Array.set()方法，该方法会把数组元素放到数组中的指定位置。代码的其余部分已经为我们写好了。

既然我们已经得到了一个稀疏数组的串行化器，我们就需要有一个使用它的方法。毕竟，Apache SOAP默认使用ArraySerializer类来处理数组的串行化和反串

行化的工作。所以我们需要找一个方法来覆盖该行为。我倾向于修改 Apache SOAP 中的 org.apache.soap.encoding.SOAPMappingRegistry 类来迫使它使用 SparseArraySerializer 而不是 ArraySerializer。然而，使用这种方法的一个问题是当它处理不含position元素的串行化数组时，会得出无效的结果。这听起来可不太好。我们当然可以将SparseArraySerializer设计得更加巧妙以处理所有的可能性，但是那意味着一个包含更多代码的非常复杂的串行化器。如果你想要这样做，我打赌Apache的设计者会很高兴考虑将它包含进去。比较简单的方式是在要使用稀疏数组的服务的mappings段里映射稀疏数组串行化器。这意味着我们不需要去修改 Apache SOAP 的源程序代码，因为这样做的话，将来如果Apache发布的新版本与它发生冲突，它就会变成一种痛苦。因此我们将使用特定于服务的方式。

让我们向urn:DataFeedService服务中加入一个叫做echoStocks的方法。这个方法使用一个字符串数组作为参数并使用同一个数组作为返回值。这里是DataFeedService类的新版本，其中包含了 echoStocks()方法的Java 实现:

```
package javasoap.book.ch6;
public class DataFeedService {
   public DataFeedService() {
   }
   public String[] echoStocks(String[] stocks) {
      return stocks;
   }
   public DataFeedMessage sendMessage(DataFeedMessage msg) {
      // 我们可以通过调用msg.toString()
      // 来在这里显示格式化后的消息
      String stock = msg.getFieldValue("SYMBOL");
      String request = msg.getFieldValue("REQUEST");
      DataFeedMessage res = new DataFeedMessage();
      res.addField("SYMBOL", stock);
      if (request.equals("PRICE"))
         res.addField("PRICE", "123.5");
      return res;
   }
}
```

我们需要向部署描述文件中增加一个映射项以便能对稀疏数组使用稀疏数组串行化器。我们想要映射的类型是 Array，名称空间用 http://schemas.xmlsoap.

org/soap/encoding/限定。这是标准SOAP数组使用的标准SOAP编码方式。映射项覆盖了用来串行化和反串行化数组的默认类。我们已经知道SparseArraySerializer类将作为java2XMLClassName和xml2JavaClassName属性的值。它告诉系统要求对数组使用新的串行化器。现在我们需要指定一个Java类来实例化Java数组。那个类来自Java反射API，名为java.lang.reflect.Array。我们将使用这个值作为映射项中javaType属性的值。echoStocks方法名被添加到service标签的methods属性里。

然而，我们仍然需要解决一个问题。Apache SOAP Java代码不会料到你要覆盖标准的串行化器，而且也不会做出任何有关串行化器和映射注册表的假设。当我们的服务试图返回来自echoStocks的一个数组时，新的串行化器仍然不会被用到，尽管我们在映射项中已经将java2xmlClassName的值指定为了SparseArraySerializer。迂回的解决办法是编写我们自己的映射注册表，这听起来好像不太容易。我们稍后将这样做；新的注册表将被命名为javasoap.book.ch6.BetterSOAPMappingRegistry。我们需要在部署描述文件中指定这个注册表，这可以通过在isd:mappings元素里加入defaultRegistryClass属性来完成，属性的值使用新注册表的类名。这就是修改后的部署描述文件：

```
<isd:service
    xmlns:isd="http://xml.apache.org/xml-soap/deployment"
    id="urn:DataFeedService">
  <isd:provider
     type="java"
     scope="Application"
    methods="sendMessage echoStocks">
   <isd:java
      class="javasoap.book.ch6.DataFeedService"
      static="false"/>
</isd:provider>

<isd:faultListener>org.apache.soap.server.DOMFaultListener
</isd:faultListener>

<isd:mappings defaultRegistryClass="javasoap.book.ch6.BetterSOAPMappingRegistry" >
   <isd:map
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:x="http://schemas.xmlsoap.org/soap/encoding/" qname="x:Array"
      javaType="java.lang.reflect.Array"
```

```
        java2XMLClassName="javasoap.book.ch6.SparseArraySerializer"
        xml2JavaClassName="javasoap.book.ch6.SparseArraySerializer"/>
      <isd:map
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:x="urn:DataFeedService" qname="x:DataFeedMessage"
        javaType="javasoap.book.ch6.DataFeedMessage"
        java2XMLClassName="javasoap.book.ch6.DataFeedMessageSerializer"
        xml2JavaClassName="javasoap.book.ch6.DataFeedMessageSerializer"/>
    </isd:mappings>
  </isd:service>
```

我们在新注册表中要做的所有事情就是返回 SparseArraySerializer的而不是ArraySerializer的一个实例。BetterSOAPMappingRegistry扩展了由Apache SOAP 提供的 SOAPMappingRegistry 类。为了完成这个工作，我们覆盖了querySerializer方法以便它在需要的时候返回SparseArraySerializer。基类做了大部分的工作，只是在它想要返回 ArraySerializer时，我们就改为返回SparseArraySerializer。这里是新的注册表类:

```
package javasoap.book.ch6;
import org.apache.soap.util.xml.*;
import org.apache.soap.*;
import org.apache.soap.encoding.soapenc.*;
public class BetterSOAPMappingRegistry
        extends org.apache.soap.encoding.SOAPMappingRegistry {
   SparseArraySerializer sparseSer = new SparseArraySerializer();
   public BetterSOAPMappingRegistry() {
      super();
   }
   public Serializer querySerializer(Class javaType,
                String encodingStyleURI)
   throws IllegalArgumentException {

   Serializer s;
   try {
      s = super.querySerializer(javaType, encodingStyleURI);
      if (s instanceof ArraySerializer)
        return sparseSer;
   }
   catch (IllegalArgumentException e) {
      if (javaType != null
       && encodingStyleURI != null
       && encodingStyleURI.equals(Constants.NS_URI_SOAP_ENC)) {
          if (javaType.isArray()) {
            return sparseSer;
```

```
            }
        }
        throw e;
    }
    return s;
  }
}
```

如果我们已经使用了isd:service元素的defaultRegistryClass属性，为什么还要在部署描述文件使用映射项呢？是的，这是因为属性本身不起作用，而把属性组合到映射项中就可以。这个原理非常简单，而且能帮你不必深入到Apache源代码的内部。接下来我们要部署这个服务。

现在，让我们来看看在一个客户应用程序中需要做哪些事情。我们当然需要做同一件事，但将采用不同的方式。我们先指定该映射，然后使用BetterSOAP-MappingRegistry类而不是SOAPMappingRegistry类。这里是客户代码：

```
package javasoap.book.ch6;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.util.xml.*;
public class DataFeedClient2 {
   public static void main(String[] args)
      throws Exception {

      URL url =
        new URL(
          "http://georgetown:8080/soap/servlet/rpcrouter");

      Call call = new Call();

      BetterSOAPMappingRegistry smr =
                        new BetterSOAPMappingRegistry();
      call.setTargetObjectURI("urn:DataFeedService");
      call.setMethodName("echoStocks");
      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      call.setSOAPMappingRegistry(smr);
      SparseArraySerializer arraySer =
                        new SparseArraySerializer();
      // 映射该数组
```

```
        smr.mapTypes(Constants.NS_URI_SOAP_ENC,
                     new QName(Constants.NS_URI_SOAP_ENC, "Array"),
                     java.lang.reflect.Array.class, arraySer,
                     arraySer);
        String[] stocks = new String[10];
        stocks[2] = "XYZ";
        stocks[5] = "ABC";
        Vector params = new Vector();
        params.addElement(new Parameter("msg", String[].class,
                                  stocks, null));
        call.setParams(params);

        try {
            Response resp = call.invoke(url, "");
            Parameter ret = resp.getReturnValue();
            String[] value = (String[])ret.getValue();
            int cnt = value.length;
            for (int i = 0; i < cnt; i++) {
                if (value[i] != null) {
                    System.out.println("Item " + i + ": " + value[i]);
                }
            }
        }
        catch (SOAPException e) {
            System.err.println("Caught SOAPException (" +
                               e.getFaultCode() + "): " +
                               e.getMessage());
        }
    }
}
```

smr变量现在是BetterSOAPMappingRegistry的一个实例，但是从那里开始我们就像前面的例子那样使用它。我们创建了SparseArraySerializer的一个实例，将它用于映射过程；然后我们又调用了smr.mapTypes()方法，并将先前初始化的SparseArraySerializer对象作为用来串行化和反串行化的对象，将java.lang.reflect.Array.class用作实现该数组的对象的类。请注意Qname构造函数所使用的参数。第一个参数是标准的SOAP编码名称空间，第二个是"Array"。因此它会覆盖默认的映射。

现在我们产生一个大小为10的空字符串数组，并为它的第2和第5个位置上的元素赋值。数组中的其他元素为null。不要赋给它们空字符串值，因为一个空的字符串与null值是两个不同的概念（尽管在这里使用空字符串并没有什么害处，但

是如果这样做，数组里的10个项都会得到一个被串行化的元素）。我们使用了与前面一样的办法调用了echoStocks服务。返回值的类型被转换成String[]，以便我们能利用它来找到其中各元素的值。如果前面所做的全都没错，我们应该只在位置2和位置5找对象，因为服务器返回了与我们发给它的相同的数组。运行这个应用程序将产生如下输出：

```
Item 2: XYZ
Item 5: ABC
```

下面是这个例子的SOAP封套。如果没问题，它将只描述那些真正被串行化的非null元素。

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
  <ns1:echoStocks
   xmlns:ns1="urn:DataFeedService"
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
     <msg
       xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/"
       xsi:type="ns2:Array" ns2:arrayType="xsd:string[10]">
         <item ns2:position="[2]" xsi:type="xsd:string">XYZ</item>
         <item ns2:position="[5]" xsi:type="xsd:string">ABC</item>
     </msg>
 </ns1:echoStocks>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

msg元素的类型被设定为一个10个字符串的数组，与其他的数组没什么不同。每个元素的类型被显式地设为xsd:string，而且都包含了一个position属性来指示它的位置。那正是我们的目的。

串行化集合

数组不是在Java编程中惟一被普遍使用的集合类型。如果能对散列表、链表以及其他一些集合类型进行串行化，那也将很不错。GLUE为散列表提供了一个自定义实现，这很值得一看。这个例子只是GLUE所提供的许多自定义串行化机制中

的一个。它需要你实时编写自定义类型的模式，这当然意味着你需要了解模式。虽然这不是这本书的主题，但是看看其中的部分代码仍然对你有好处。

要在GLUE中编写自定义串行化器，我们首先要扩展electric.xml.io.Type类来生成一个新类，这个类是GLUE里SOAP类型的基类。我们需要做的全部事情就是覆盖方法writeSchema()、writeObject()和readObject()。这就是HashtableType类在GLUE中的实现：

```
// Mind Electric版权2001
package electric.xml.io.collections;
import java.io.*;
import java.util.*;
import electric.xml.*;
import electric.xml.io.*;
import electric.util.Value;
public class HashtableType extends Type
  {
  /**
   * @param schema
   */
  public void writeSchema( Element schema )
    {
    Element complexType = schema.addElement( "complexType" );
    complexType.setAttribute( "name", "Map" );
    Element sequence = complexType.addElement( "sequence" );
    Element element = sequence.addElement( "element" );
    element.setAttribute( "name", "item" );
    element.setAttribute( "minOccurs", "0" );
    element.setAttribute( "maxOccurs", "unbounded" );
    Element subType = element.addElement( "complexType" );
    Element subSequence = subType.addElement( "sequence" );
    Element key = subSequence.addElement( "element" );
    key.setAttribute( "name", "key" );
    key.setAttribute( "type", getName( key, Object.class ) );
    Element value = subSequence.addElement( "element" );
    value.setAttribute( "name", "value" );
    value.setAttribute( "type", getName( key, Object.class ) );
    }
 /**
  * @param writer
  * @param object
  * @throws IOException
  */
 public void writeObject( IWriter writer, Object object )
   throws IOException
```

```
{
writer.writeType( this );
Hashtable hashtable = (Hashtable) object;
for( Enumeration enum = hashtable.keys();
                          enum.hasMoreElements(); )
  {
  Object key = enum.nextElement();
  Object value = hashtable.get( key );
  IWriter itemWriter = writer.writeElement( "item" );
  itemWriter.writeObject( "key", key );
  itemWriter.writeObject( "value", value );
  }
}
/**
 * @param reader
 * @param value
 * @throws IOException
 */
public void readObject( IReader reader, Value value )
  throws IOException
  {
  Hashtable hashtable = new Hashtable();
  value.setObject( hashtable );
  IReader[] readers = reader.getReaders( "item" );
  for( int i = 0; i < readers.length; i++ )
    {
    IReader itemReader = readers[ i ];
    Object key = itemReader.readObject( "key" );
    Object theValue = itemReader.readObject( "value" );
    hashtable.put( key, theValue );
    }
  }
}
```

GLUE为这个类提供了内置的支持，因此当我们编写服务和客户应用程序代码时就不必对它进行注册。如果你想在GLUE中开发自己的自定义类型串行化器，就必须使用electric.xml.io.Mappings类里那些mapClass()方法中的一个。这些方法告诉GLUE系统该如何将你的自定义类型映射到串行化器。

writeSchema()方法中的代码为散列表写一个模式定义。我们将不去看那些用来实现这个目的的类和方法，相反，让我们看看生成的模式：

```
<complexType name="Map">
  <sequence>
```

```
      <element name="item" minOccurs="0" maxOccurs="unbounded">
        <complexType>
          <sequence>
            <element name="key" type="KEYTYPE"/>
            <element name="value" type="VALUETYPE"/>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
```

这是对一个散列表的模式定义，或者叫一个映射。记住一个映射中的每一项都包含了键/值对。它使用名称Map，被定义成一个complexTyp类型，它是一个包含了0个或更多元素的序列。每个元素也是一个复杂类型，是一个包含两个子元素key和value的序列。元素key和value的类型被设为适当的类型名；我用KEYTYPE和VALUETYPE作为占位符。在实际的代码中，那些类型将由对应的Java对象实例的类来确定。当我们运行一个使用这个自定义串行化器的例子时，就会看到这种情况。

自定义类型的内容在writeObject()方法中被串行化。该方法的第一个参数是electric.xml.io.IWriter接口的一个实例。我们用这个接口来写要被串行化的对象的内容。第二个参数是要被串行化的Java对象。这个方法首先将该对象的类型写入到一个流中，并将this传递到writer的writeType()方法。这样可以识别正被串行化为Map的对象的类型。然后该对象的类型被转换成Hashtable的一个实例以便我们能访问键/值对。我们所要做的全部工作就是使用从hashtable.keys()方法返回的Enumeration来遍历表中的元素。对于每个元素，我们都取得键和值的对象实例。为了写每一项，我们调用writer.writeElement()方法，它返回一个称为itemWriter的IWriter的新实例。这个实例通过使用itemWriter.writeObject()方法来写出item元素的键和值数据，调用方法时使用了元素名称和相关的Object实例作为参数。

readObject()方法完成反串行化工作。它的第一个参数是electric.xml.io.IReader接口的一个实例，我们用它来读取要被反串行化的内容。另一个参数是electric.util.Value的一个实例，它包含一个java.lang.Object实例和它的对应类型（java.lang.Class的一个实例）。第一步是要建立java.util.

Hashtable的一个新实例并将它作为参数传递到value.setObject()方法。接着，我们调用reader.getReaders()方法，该方法返回一个IReader实例的数组，其中的每一个元素对应于映射中的一个键/值对。对于每个IReader，我们都调用getObject()方法，以取得元素的键和值对象。然后这两个对象又被hashtable.put()方法加入到散列表中。

这就是要做的所有工作。我们现在要创建一个使用散列表的服务和客户应用程序，来看看如何注册自定义串行化器。让我们创建一个叫做urn:AnotherFeedService的新服务。这个服务像前面的数据来源服务一样，也包含了一个叫做sendMessage()的方法。但是这一次该方法的参数是一个散列表，散列表中的每个元素都包含了字段的名称和它们的相应值。这意味着新服务必须将数据格式化为本章开始时指定的格式。既然这不会对我们所做的工作带来什么影响，我们就不必去管它。我们将只关注散列表的自定义串行化。javasoap.book.ch6.AnotherFeedService类实现了该服务。

```
package javasoap.book.ch6;
import java.util.Hashtable;
public class AnotherFeedService {
   public AnotherFeedService() {
   }
   public Hashtable sendMessage(Hashtable msg) {
      String stock = (String)msg.get("SYMBOL");
      String request = (String)msg.get("REQUEST");
      Hashtable res = new Hashtable();
      res.put("SYMBOL", stock);
      if (request.equals("PRICE"))
         res.put("PRICE", "123.5");
      return res;
   }
}
```

sendMessage()方法与前面的DataFeedService类中的方法所完成的工作基本上是相同的。不同的是消息传递时使用了散列表而不是DataFeedMessage的实例。为部署服务，我们使用了FeedServiceApp类：

```
package javasoap.book.ch6;
import electric.util.Context;
import electric.registry.Registry;
```

```
import electric.server.http.HTTP;
import java.util.Hashtable;
public class FeedServiceApp {

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

      String ns = "urn:AnotherFeedService";
      HTTP.startup("http://georgetown:8004/glue");
      Context context = new Context();
      context.addProperty("activation", "application");
      context.addProperty("namespace", ns);
      Registry.publish(ns,
         javasoap.book.ch6.AnotherFeedService.class, context );
   }
}
```

既然我们使用的是一个在GLUE中已经内置了支持的自定义类型，那就让我们手工编写用来绑定到服务的接口。下面就是javasoap.book.ch6.IAnother-DataFeed接口。它包含了我们为服务定义的sendMessage()方法，该方法带有一个Hashtable类型的参数，同时也返回一个Hashtable。

```
package javasoap.book.ch6;

import electric.registry.RegistryException;
import electric.registry.Registry;
import java.util.*;

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

         IAnotherFeedService srv = (IAnotherFeedService)Registry.bind(
          "http://georgetown:8004/glue/urn:AnotherFeedService.wsdl",
           IAnotherFeedService.class);

         Hashtable msg = new Hashtable();
         msg.put("SYMBOL", "XYZ");
         msg.put("REQUEST", "PRICE");

         Hashtable result = srv.sendMessage(msg);
```

```
            for (Enumeration e = result.keys(); e.hasMoreElements(); ) {
                String key = (String)e.nextElement();
                String value = (String)result.get(key);
                System.out.println(key + ": " + value);
            }
        }
        catch (RegistryException e) {
            System.out.println(e);
        }
    }
}
```

返回的散列表里的元素被枚举，并显示了它们的键/值对。其结果是:

```
SYMBOL: XYZ
PRICE: 123.5
```

让我们来看看调用 sendMessage()时，发送到服务的 SOAP 封套。这是一个查看被串行化的散列表的机会。

```
<soap:Envelope
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <soap:Body>
      <n:sendMessageResponse xmlns:n='urn:AnotherFeedService'>
        <Result href='#id0'/>
      </n:sendMessageResponse>
      <id0 id='id0' soapenc:root='0'
            xmlns:ns2='http://xml.apache.org/xml-soap'
            xsi:type='ns2:Map'>
        <item>
          <key xsi:type='xsd:string'>PRICE</key>
          <value xsi:type='xsd:string'>123.5</value>
        </item>
        <item>
          <key xsi:type='xsd:string'>SYMBOL</key>
          <value xsi:type='xsd:string'>XYZ</value>
        </item>
      </id0>
    </soap:Body>
</soap:Envelope>
```

和我们已经见过的在GLUE中使用复杂类型的例子一样，实际的数据被单独串行化了，而参数又引用了这个数据。我们要发送的散列表可以在 id0 元素中找到。它的类型被设为Map，由http://xml.apache.org/xml-SOAP作为名称空间限定。之所以使用这个名称空间，是因为Apache组实际上已为编码一个映射类型，或者说是散列表建立了这个模式。而GLUE实现在这个方面与Apache保持了一致。每个item的子元素都包含了key和value元素，这些key和value元素又依次包含了映射项的键/值对。在这种情况下，所有的键和值都是字符串，这是因为我们在例子中使用了这种类型。

在本章中，我们看到了该如何为我们自己的自定义类型创建串行化器。使用这些技术，结合那些已内置支持的类型，事实上已经没什么SOAP数据类型不能被传输和使用了。在某些情况下，你会发现你是在头脑中用SOAP设计类，至少在头脑中会对你所选择的SOAP实现的能力和限制进行判断。是否能解决你所使用的SOAP实现的某些局限性对你的项目来说很重要，当你需要使用更高级的特性时，自定义的串行化能为你提供很多的思路。

第七章

错误和异常

软件编写的好与坏通常在于它们对错误和问题的处理方式上。在每件事都能正常工作的情况下的处理过程要比在出错情况下的处理过程简单得多。结构化的和面向对象的编程为处理出错提供了很多技术。方法常常会返回一个特殊的值，比如null或-1，来表示某种类型的错误。在Java中，方法则时常通过抛出一个异常来表示发生了某种不正常的事情。好的软件常常能够预期将要发生的错误，并准备好采取适当的行动去解决它。在SOAP的术语中，这些不正常的或者说异常的问题，通常被称为错误(fault)。每当一个服务方法不能处理输入的参数或不能正确地返回数据时，就会产生错误。为什么会发生错误？这可是一个永远也讨论不完的问题。引起错误的原因通常是为方法提供了不正确的参数值、后台的问题以及SOAP请求消息使用了不正确的格式。我相信你还能找到许多其他的原因，而且还可能被迫写过一些代码来处理过这些问题。在本章中，我们将介绍在SOAP中产生和处理错误的机制。

在Apache SOAP中抛出一个服务器端异常

一个服务产生错误的最常见的办法就是抛出一个异常。Apache SOAP中有一个机制可以用来处理实现服务的Java类中抛出的异常。异常中所包含的信息会被用来生成相应的SOAP错误，并将它返回到调用者。

SOAP错误的内容我们早在第二章中就已经介绍过。在那里我们介绍了可以在错误中使用的四种失效错误。其中一个是SOAP-ENV:Server，它表示当处理与请求消息无关的内容时出现的问题。这个错误码可以用在，比如，一个被用来正确处理消息的后台服务器不可用的情况下。我们来创建一个简单的服务，让它产生一个最简单的错误，以便我们研究Apache SOAP中所采用的基本错误机制。这里是javasoap.book.ch7.SampleFaultService类，它实现了urn:SampleFault-Service服务：

```
package javasoap.book.ch7;
import org.apache.soap.SOAPException;
public class SampleFaultService {
   public SampleFaultService() {
   }
   public int generateFault()
        throws Exception {
      // 某种不正常的事情发生了……
      throw new Exception();
   }
}
```

generateFault()方法的声明表示它可以抛出java.lang.Exception的一个实例。事实上，那是它所做的全部工作。我们模拟了当服务方法接收到一个调用却因为后台问题不能处理该调用的情况，看看这时会发生什么事情。

现在让我们来看看调用generateFault服务的客户应用程序。到目前为止，我们一直是通过调用Response对象的getReturnValue()方法来从一个服务取得返回结果。然而，这种方式假设该服务被正确调用了，它不处理有错误发生的情况。为了将错误考虑进来，我们需要通过调用Response对象的faultGenerated()方法来询问它是否出现了异常。如果它返回true，则说明有异常发生，若为false则表示没有。如果没有出现异常，我们就可以像从前一样处理它，但如果有异常，我们就要调用Response对象的getFault()方法来从Response对象中获取该异常。这个方法返回类org.apache.soap.Fault的一个实例，它是一个用来封装SOAP错误内容的类。

```
package javasoap.book.ch7;
import java.net.*;
import java.util.*;
```

```
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.util.xml.*;
public class SampleClient {
   public static void main(String[] args)
      throws Exception {

      URL url =
        new URL(
          "http://georgetown:8080/soap/servlet/rpcrouter");

      Call call = new Call();

      call.setTargetObjectURI("urn:SampleFaultService");
      call.setMethodName("generateFault");
      try {
         Response resp = call.invoke(url, "");
         if (resp.generatedFault()) {
            String code = resp.getFault().getFaultCode();
            String desc = resp.getFault().getFaultString();
            System.out.println(code + ": " + desc);          }
         else {
            Parameter ret = resp.getReturnValue();
            Object value = ret.getValue();
            System.out.println(value);
         }
      }
      catch (SOAPException e) {
         System.err.println("Caught SOAPException (" +
                            e.getFaultCode() + "): " +
                            e.getMessage());
         }
      }
   }
```

如果你运行这个例子，就会得到如下输出：

```
SOAP-ENV:Server: Exception from service object: null
```

这个例子除了从服务中抛出类java.lang.Exception的一个实例之外其他什么事情也没做，它生成了一个格式正确并且代码为SOAP-ENV:Server的错误。该错误中还有包含了错误字符串和错误施动者元素的默认值。这就是从generateFault服务方法返回的SOAP错误：

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<SOAP-ENV:Fault>
<faultcode>SOAP-ENV:Server</faultcode>
<faultstring>Exception from service object: null</faultstring>
<faultactor>/soap/servlet/rpcrouter</faultactor>
</SOAP-ENV:Fault>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

但是错误是在哪里产生的呢？答案在我们前面任何一个例子的部署描述文件中都可以找到。其中的每一个描述符中都包含了一个isd:faultListener元素项，它看起来可能如下所示：

```
<isd:faultListener>org.apache.soap.server.DOMFaultListener
</isd:faultListener>
```

这个元素指定了你的服务中用来处理错误的那个Java类。这个类处理你的服务中抛出的异常，并把它转换成一个SOAP错误。不幸的是，从一个java.lang.Exception的实例中提取出来的信息用处并不大。Apache SOAP中包括了一个org.apache.soap.SOAPException类，该类对java.lang.Exception进行了扩展，包含了几个读取和设置SOAP错误码和错误字符串的方法。这当然很有用，因此我们将服务类改为使用SOAPException：

```
package javasoap.book.ch7;
import org.apache.soap.SOAPException;
public class SampleFaultService {
   public SampleFaultService() {
   }
   public int generateFault()
        throws SOAPException {
      throw new SOAPException("SOAP-ENV:Server",
                              "Data Feed Unavailable");
   }
}
```

现在generateFault()方法抛出了SOAPException的一个实例。让我们假定错误是因为服务所用的底层数据来源不可用造成的。在此情况下，我们就需要使用SOAP-ENV:Server错误码，它用作SOAPException类构造函数的第一个参数。第

二个参数对错误进行了描述。这两个值被用来生成SOAP错误的faultcode和faultstring元素。这种方法也允许我们使用SOAP-ENV:Client来产生错误，这个错误码代表服务方法遇到了一个格式不正确的请求或者由于SOAP体中其他信息的问题导致处理不成功。回想一下前面几章的股票市场数据来源服务的例子。其中有几个例子在消息字段里使用了属性/值对。如果所要求的特定字段丢失了，比如缺少用来表示股票代码的SYMBOL字段，服务器就会产生一个使用SOAP-ENV:Client错误码的错误，就像这样：

```
throw new SOAPException("SOAP-ENV:Client", "Missing SYMBOL");
```

如果你只是要生成一个适当的错误，这样做就足够了。但是，在某些情况下，你可以在SOAP错误中使用detail元素来返回更有意义的信息。这个元素只有在描述与请求信息体有关的错误时才允许使用。因此，它不能用于描述数据来源不可用这样的错误，但是，在许多其他的错误情况下，它确实很有用（实际上，我发现这样做是目光短浅的。因为其实有很多其他位置上的信息对于与请求体无关的错误来说都是有帮助的。举例来说，假设某个数据来源不可用，但另外有一个数据来源却是可以接受请求的。这时你可能会希望在错误中包括一些必要的信息，以使得客户可以转向使用另一个可用的服务。当然，你可以将你的服务设计为能够自动选择后台资源，但那样做并不总能合乎要求）。在下一节中，我们将尝试在使用SOAP-ENV:Client错误码的错误中提供详细信息。

在Apache SOAP中创建一个错误监听器

为了在错误中使用detail元素，我们需要通过实现org.apache.soap.server.SOAPFaultListener接口创建一个新类来监听错误。这个接口定义了一个叫做'fault()的方法，每当需要产生一个错误时Apache SOAP框架就会调用该方法。惟一要传递给这个方法的的参数是org.apache.soap.server.SOAPFaultEvent的一个实例（注1），它包含org.apache.soap.Fault的一个实例和被服务方法

注1：如果你熟悉JavaBeans模式中对事件的命名方法，那你也许就会明白这种命名惯例，它表示发生的事件是一个SAOPFault事件。监听者接口使用在事件名后面加上Listener的方式来命名，而包含事件信息的对象则以在其后加上Event的方式命名。

抛出的SOAPException。错误对象中包括了几个用来获取和设置错误码、错误字符串、错误施动者以及详细信息的方法。这就是我们为提供详细信息所需要操作的对象。

但是在创建错误监听器之前，我们还需要一种机制来向它提供详细的信息数据。在我们的服务方法与错误监听器之间没有直接的沟通渠道，但是我们可以利用异常机制。既然我们已经知道由服务方法抛出的SOAPException异常可被错误监听器访问，我们就可以通过继承SOAPException来创建一个符合我们需要的新类。但在这样做之前，我们还需要一个类来装载这些信息。我们可以用一个java.util.Hashtable。用这种方法，我们可以组装出任意数量的属性/值对来描述错误的详细信息。考虑了所有这些问题之后，我们现在就来创建一个新的异常类型，它包含一个Detail属性，它是Hashtable的一个实例。

```
package javasoap.book.ch7;
import org.apache.soap.*;
import java.util.Hashtable;
public class FeedException extends SOAPException {
   Hashtable _detail;
   public FeedException(String fcode, String fstring) {
      super(fcode, fstring);
   }
   public void setDetail(Hashtable detail) {
      _detail = detail;
   }
   public Hashtable getDetail() {
      return _detail;
   }
}
```

这段代码相当简单。既然FeedException扩展了SOAPException，我们就可以在前面使用SOAPException实例的地方使用新类的实例来代替。它的构造函数使用错误码和错误字符串作为参数，并将它们传递到其父类的构造函数。Detail属性被定义为Hashtable的一个实例，我们拥有它的设置和获取存取器。

现在我们就来修改服务类，让它抛出一个用一些详细信息组装出的FeedException的实例。这就是修改后的新SampleFaultService类的代码。我们在详细信息中包含了reasonCode、reasonDescription、alternateProvider和

alternateContact字段以及它们的相应值。这些包含在一个散列表中的详细信息通过SetDetail()方法被传递到FeedException实例。我们之所以能够成功地抛出FeedException，是因为它是SOAPException的一个子类。另一方面，我们也可以修改这个方法的定义，让它直接抛出一个FeedException。

```
package javasoap.book.ch7;
import org.apache.soap.SOAPException;
import java.util.Hashtable;
public class SampleFaultService {
   public SampleFaultService() {
   }
   public int generateFault()
        throws SOAPException {
      FeedException e = new FeedException("SOAP-ENV:Server",
                              "Data Feed Unavailable");
      Hashtable detail = new Hashtable();
      detail.put("reasonCode", "199");
      detail.put("reasonDescription", "Power Outage");
      detail.put("alternateProvider", "www.mindstrm.com");
      detail.put("alternateContact", "rob@mindstrm.com");
      e.setDetail(detail);
      throw e;
   }
}
```

现在我们可以在新的错误监听类上工作了。我们称这个类为javasoap.book.ch7.FeedFaultListener。它实现了org.apache.soap.server.SOAPFaultListener接口并实现了必需的fault()方法。

```
package javasoap.book.ch7;
import org.apache.soap.*;
import org.apache.soap.server.*;
import java.util.*;
import org.w3c.dom.*;
import org.apache.soap.util.xml.*;
import javax.xml.parsers.DocumentBuilder;

public class FeedFaultListener
        implements SOAPFaultListener {
   public FeedFaultListener() {
   }
   public void fault(SOAPFaultEvent event) {
```

```
        FeedException ex = (FeedException)event.getSOAPException();
        Vector v = new Vector();
        DocumentBuilder builder =
                XMLParserUtils.getXMLDocBuilder();
        Document doc = builder.newDocument();
        Hashtable detail = ex.getDetail();
        Enumeration e = detail.keys();
        while (e.hasMoreElements()) {
            String name = (String)e.nextElement();
            String value = (String)detail.get(name);
            Element elem = doc.createElement(name);
            Text txt = doc.createTextNode("");
            txt.setData(value);
            elem.appendChild(txt);
            v.addElement(elem);
        }
        event.getFault().setDetailEntries(v);
    }
}
```

在详细介绍其逻辑之前，让我们来看看 fault()方法到底完成了什么工作。它接收一个 SOAPFaultEvent 事件，以响应服务方法抛出的 FeedException。这个事件中包含了对FeedException和将被发送到调用者的实际Fault两者的引用。因此我们需要从FeedException中将在前面加入进去的详细信息提取出来，并转换成一组 XML 元素，然后再把这些项目添加到 Fault 里，以便能将它们写入到要送回给调用者的 SOAP 文档中去。

fault()方法首先要取得服务方法抛出的FeedException。为获得这个异常，我们调用了 event 的 getSOAPException()方法。返回结果的类型被转换为 FeedException，我们可以从中找到错误的细节信息。接下来我们创建了java.util.Vector的一个实例，用于保存我们在前面创建的详细信息部分的元素。

现在我们需要为detail元素产生XML子元素，其中用来操作XML所使用的API都不是 SOAP 专用的。如果你想要了解如何在 Java 中操作 XML，请参考 Brett McLaughlin 编写的《Java and XML》一书（O'Reilly 出版）。我们在本书中没有涉及到很复杂的特性，因此你阅读本书后面的内容应该没什么困难。我们调用了XMLParserUtils.getXMLDocBuilder以得到一个DocumentBuilder接口的实例，它随后被用来创建一个新的文档。这个 Document 是一个我们用来存放细节信息项的 XML 文档。

这些细节通过使用FeedException的getDetail()方法得到。接下来，我们将要枚举散列表中的项目，取出其中用来组成细节数据的名/值对。我们对每一对数据都以其name为元素名创建了一个新的XML元素，这个工作是通过调用doc.createElement()方法来完成的。然后我们创建了一个文本节点，并将它的值设为从散列表中取回的值。现在把这个文本节点添加到该元素中，接着再把该元素加入到我们前面创建的矢量里。当我们处理了所有的名/值对之后，再通过将其作为参数传递到setDetailEntries()方法来将详细信息元素中的矢量加入到Fault对象中。

现在我们要来部署这个服务了，这一次我们使用FeedFaultListener类作为部署描述文件中isd:FaultListener元素的值：

```
<isd:faultListener>
javasoap.book.ch7.FeedFaultListener
</isd:faultListener>
```

如果你运行客户应用程序，从服务器返回的错误中将会包含一个详细信息部分。客户并不会处理这个部分，但是我们可以在服务器返回的SOAP封套中看到我们工作的结果：

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
 <SOAP-ENV:Fault>
  <faultcode>SOAP-ENV:Server</faultcode>
  <faultstring>Data Feed Unavailable</faultstring>
  <faultactor>/soap/servlet/rpcrouter</faultactor>
  <detail>
    <alternateProvider>www.mindstrm.com</alternateProvider>
    <reasonDescription>Power Outage</reasonDescription>
    <alternateContact>rob@mindstrm.com</alternateContact>
    <reasonCode>199</reasonCode>
  </detail>
 </SOAP-ENV:Fault>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

四个详细信息字段被作为了detail元素的子元素。为了使这个信息变得有用，需要让我们的客户应用程序处理它。现在我们就来这么做。下面是客户应用程序的一个修改版本，它将把我们在错误中发送的信息提取出来：

```
package javasoap.book.ch7;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.util.xml.*;
import org.w3c.dom.*;
public class SampleClient {
   public static void main(String[] args)
      throws Exception {

      URL url =
        new URL(
          "http://georgetown:8080/soap/servlet/rpcrouter");

      Call call = new Call();

      call.setTargetObjectURI("urn:SampleFaultService");
      call.setMethodName("generateFault");
      try {
         Response resp = call.invoke(url, "");
         if (resp.generatedFault()) {
            Fault fault = resp.getFault();
            String code = fault.getFaultCode();
            String desc = fault.getFaultString();
            System.out.println(code + ": " + desc);
            Vector v = fault.getDetailEntries();
            int cnt = v.size();
            for (int i = 0; i < cnt; i++) {
               Element n = (Element)v.elementAt(i);
               Node nd = n.getFirstChild();
               System.out.println(n.getNodeName() + ": " +
                                       nd.getNodeValue());
            }
         }
         else {
            Parameter ret = resp.getReturnValue();
            Object value = ret.getValue();
            System.out.println(value);
         }
```

```
        }
        catch (SOAPException e) {
            System.err.println("Caught SOAPException (" +
                               e.getFaultCode() + "): " +
                               e.getMessage());
        }
    }
}
```

如果resp.generatedFault()返回true，我们就像以前那样显示错误码和错误字符串。然后，我们调用fault.getDetailEntries()方法，它返回一个包含详细信息元素的矢量。接着我们简单地遍历这个矢量，再将得到的每一个对象的类型转换成org.w3c.dom.Element的一个实例，并对结果元素调用getFirstChild()方法。最后我们显示该元素的名称和节点数据。如果你运行该例子，就会看到如下输出：

```
SOAP-ENV:Server: Data Feed Unavailable
alternateProvider: www.mindstrm.com
reasonDescription: Power Outage
alternateContact: rob@mindstrm.com
reasonCode: 199
```

在GLUE中抛出和捕获异常

在GLUE中有关错误和异常的概念与在Apache SOAP中的很相似，尽管其中所用的某些技术是不同的。实际上，GLUE有多种方式可以用来处理异常，因此如果你对这些方式都感兴趣的话，我建议你去阅读GLUE的用户指南；我们在这里只讨论其中的一种。

让我们跳过基本的介绍直接来看一个返回包含详细信息的错误的例子。GLUE中带有自己的SOAP异常类electric.net.soap.SOAPException。它与Apache的SOAPException不同，因此不要将两者弄混淆了。GLUE的SOAPException允许你设置错误码、错误字符串、错误施动者和详细信息，它有多个构造函数，它们分别使用错误元素的不同组合作为参数。因此，你不必再去编写错误处理程序来向错误中写入详细信息了；只需要抛出一个带有适当值的SOAPException的实例

即可。现在我们来创建一个名为urn:AnotherFaultService的服务，并用AnotherFaultService类来实现它:

```
package javasoap.book.ch7;
import electric.net.soap.SOAPException;
import electric.xml.*;
public class AnotherFaultService {
   public int generateFault()
         throws SOAPException {
      Element elem = new Element("detail");
      Element sub = elem.addElement("reasonCode");
      sub.addText("199");
      sub = elem.addElement("reasonDescription");
      sub.addText("Power Outage");
      sub = elem.addElement("alternateProvider");
      sub.addText("www.mindstrm.com");
      sub = elem.addElement("alternateContact");
      sub.addText("rob@mindstrm.com");
      SOAPException e = new SOAPException(
                           "Data Feed Unavailable",
                           SOAPException.SERVER,
                           "urn:AnotherFaultService",
                           elem);
      throw e;
   }
}
```

在这个类中，我们在建立SOAPException的一个实例之前先生成了详细信息部分。GLUE带有一个简单的XML API，我们在这里用它来产生detail元素。第一步是用detail作为标签名称创建了一个electric.xml.Element元素的实例。接着，我们又用所添加元素的标签名作为参数调用elem.addElement()方法向detail元素中添加了一个新的子元素。elem.addElement()方法返回了一个元素，我们调用该元素的addText()方法设置了元素的值。一旦建立了detail元素，我们就准备来创建SOAPException了。要传递给构造函数的参数是错误字符串、错误码、错误施动者以及包含有细节信息的XML元素。这里是用来部署urn:AnotherFaultService服务的代码:

```
package javasoap.book.ch7;
import electric.util.Context;
import electric.server.http.HTTP;
import electric.registry.Registry;
```

```
public class FaultServiceApp {

      public static void main( String[] args )
                 throws Exception {
      String ns = "urn:AnotherFaultService";
      HTTP.startup("http://georgetown:8004/glue");
      Context context = new Context();
      context.addProperty("activation", "application");
      context.addProperty("namespace", ns);
      Registry.publish(ns,
         javasoap.book.ch7.AnotherFaultService.class, context );
   }
}
```

用来绑定到服务的Java接口中带有一个generateFault()方法，这个方法被声明为可以抛出electric.net.soap.SOAPException的一个实例:

```
package javasoap.book.ch7;
import electric.net.soap.SOAPException;
public interface IAnotherFaultService {
  int generateFault() throws SOAPException;
}
```

GLUE将在线处理错误的细节，并在客户端将这个错误转换回electric.net.soap.SOAPException的一个实例。这里就是调用urn:AnotherFaultService服务中generateFault()方法的客户应用程序:

```
package javasoap.book.ch7;
import electric.registry.RegistryException;
import electric.registry.Registry;
import electric.net.soap.SOAPException;
import electric.xml.*;
public class FaultServiceClient {
   public static void main(String[] args) throws Exception
   {
      try {
         IAnotherFaultService srv =
           (IAnotherFaultService)Registry.bind(
             "http://georgetown:8004/glue/urn:AnotherFaultService.wsdl",
             IAnotherFaultService.class);
         srv.generateFault();

      }
      catch (SOAPException se) {
```

```
            System.out.println(se.getSOAPCode());
            System.out.println(se.getMessage());
            System.out.println(se.getSOAPActor());
            System.out.println(se.getSOAPDetailElement());
        }
        catch (RegistryException e) {
            System.out.println(e);
        }
    }
}
```

我们在SOAPException的catch块中打印了错误的内容。错误码、错误字符串和错误施动者分别通过调用SOAPException的getSOAPCode()、getMessage()和getSOAPActor()方法取得，而细节信息则保存在调用getSOAPDetailElement()方法所返回的一个XML元素中。如果我们想要处理详细信息部分的字段，可以单独处理它的子元素及其值。但在这里，我们只是利用了detail元素的toString()方法来显示它的内容。如果你运行它，就会得到如下输出：

```
Server
Data Feed Unavailable
urn:AnotherFaultService
<detail>
  <reasonCode>199</reasonCode>
  <reasonDescription>Power Outage</reasonDescription>
  <alternateProvider>www.mindstrm.com</alternateProvider>
  <alternateContact>rob@mindstrm.com</alternateContact>
</detail>
```

其中惟一看起来不寻常的是在第一行上显示的错误码，它没有包括名称空间限定符，显然是SOAPException实现将它省掉了。这或许是个好主意，因为在你不知道它的具体值的情况下该限定符是没有意义的。看看从服务器返回的SOAP封套，你就会明白我所指的是什么：

```
<soap:Envelope
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <soap:Body>
      <soap:Fault>
```

```
        <faultcode>soap:Server</faultcode>
        <faultstring>Data Feed Unavailable</faultstring>
        <faultactor>urn:AnotherFaultService</faultactor>
        <detail>
          <reasonCode>199</reasonCode>
          <reasonDescription>Power Outage</reasonDescription>
          <alternateProvider>www.mindstrm.com</alternateProvider>
          <alternateContact>rob@mindstrm.com</alternateContact>
        </detail>
      </soap:Fault>
    </soap:Body>
  </soap:Envelope>
```

产生和处理SOAP错误是使用SOAP来编写应用程序的一个重要部分。你不能小看它，就像在Java编程中你不能忽视异常处理一样。幸运的是，处理错误的技术简单易懂，如果再做一点额外的工作，就可以无休无止地向其中加入细节信息。

最好的软件总是能够预料到可能会发生的错误。诀窍是提前设计处理它们的方法，而且不要对意外的错误感到惊讶。这不是SOAP的内容，而是软件设计的内容。如果你能正确地处理错误，你的SOAP工作就不会再有什么障碍了。

第八章

可供选择的技术

虽然SOAP最常用于RPC，但仍然有其他的可能性。SOAP还可以用来创建消息形式的服务，在这种情况下，SOAP封套被发送到服务器而不要求服务器向调用者返回任何数据。这种消息中可以包含任何对服务器有意义的有效负载——而且它不必被写成一个过程调用。这与电子邮件系统的工作方式有点类似。当你发送一个电子邮件时，你不会从服务器得到什么，也没有任何规则来限定邮件所能包含的内容。服务器接收你的消息并完成它自己的工作，并不再有任何牵涉到发送者的工作。这与RPC服务有着鲜明的不同，在RPC服务中，返回值是方法调用过程的一部分。

不是所有的SOAP实现都支持消息形式的服务，但这种形式的服务还是很有用的，值得我们进一步探讨。GLUE不支持消息服务，而Apache SOAP支持，因此我们将使用Apache SOAP来开发一些例子。

Apache SOAP为消息形式的服务提供的API是比我们用于RPC的层次更低的API。这不奇怪，因为RPC通过来回地发送消息来完成工作。RPC API建立在消息API之上，并且为使编写RPC服务和服务客户更容易而包含了一些有用的抽象。你会发现在Apache SOAP消息服务世界中没有这么方便。相反你将必须用更低层的API来直接处理封套的编组和解组、XML结构以及很多其他在PRC中

都能自动完成的工作。其好处是你能对这些过程有较多的控制，尽管在大部分情况下我还是更喜欢较好的抽象。

到目前为止使用的绝大多数例子中，我们都使用了SOAP编码形式，但它不是编码数据的惟一方式。直接编码是另一种重要而且有用的机制，尤其是在你想要使用SOAP传递已有的XML文档时。那么让我们来看一个使用直接编码的例子，看它是如何使用的。

我们也将看看带附件的SOAP。它的道理很简单：正如你能把一个文件附在一个电子邮件里，你同样能使用MIME把外部数据附在一个SOAP消息中。当需要传递的SOAP消息中有部分数据（比如图像或声音文件）没有必要编码或解码时，这种技术尤其有用。使用附件可以让你能与SOAP消息一起发送大的二进制数据，而不必把这些数据包含到SOAP封套里。对于许多系统来说，与在封套里编码数据相比，使用附件会带来更为明显的性能提升。

SOAP消息传递

消息传递服务的SOAP封套格式与RPC服务的格式不同。在消息服务模式里没有不同的方法签名、方法参数或者返回值的概念。基本上，你只需要向服务发送一些XML就可以了。然而，要想把消息发送到正确的服务和服务方法还有一些要求。SOAP Body第一个子元素的名称一定要与服务方法的名称对应，而且该元素使用的xmlns名称空间属性指定了服务的名称。这里是一个简单的SOAP消息，它将被发送到urn:WeatherDiary服务的recordTemperature方法进行处理：

```
<SOAP_ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Body>
      <recordTemperature xmlns="urn:WeatherDiary"/>
      <temperature>75.5</temperature>
      <zipcode>50328</zipcode>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

recordTemperature服务方法记录了与某个邮政编码有关的地区的目前温度。请注意在recordTemperature元素中没有数据，尽管如果它包含了数据和子元素也

是完全正确的。重点是这样做没有必要。事实上，我也可以把 temperature 和 zipcode 元素用作 recordTemperature 的子元素而不是兄弟元素。这两种方法的任何一个都行；至于 SOAP 所关心的，仅仅是一个被传递到某个方法的消息，而不是一个需要被正确处理的带参数的 RPC。如同你不久将会看到的那样，对这两种使用元素的方法，我们都会为它们编写一些代码。

一旦你成功地建立了第一个元素，这个 XML 的其余部分就由你决定了。你不必去对任何参数进行编码，也不必遵循任何严格的语法。但这并不意味着你能向终点的消息服务发送任意的消息。相反，没有了严格的语法要求，要使不在你控制之下的服务和服务客户能够相互操作就变得更加困难。你不得不确保你的消息遵照各种不同的其他服务所要求的格式。那不会使这个服务模型更糟，而是使其具有更大的灵活性，并随之带来与层次的灵活性相关的所有问题。

编写一个消息服务

编写一个消息形式的服务与编写一个 RPC 服务并不是一点共同点都没有：它也需要一个方法以便收到消息时被调用。所不同是方法的签名。在 RPC 中，我们可以自由定义服务方法的参数和它们的类型，以及返回值类型。在消息传递模型中，所有的服务方法都有相同的签名。它们都有三个参数。第一个是 org.apache.soap.Envelope 的一个实例，它封装了服务方法所收到的 SOAP 封套的内容。另外两个参数都是 org.apache.soap.rpc.SOAPContext 的实例（注 1），这是一个封装了 MIME 多段消息的类。其中一个用于输入，另一个用于输出。是的，是输出！尽管不要求 SOAP 信息服务提供响应，但在底层传输协议支持的情况下是允许它们这样做的。HTTP 协议当然允许向调用者返回某些内容，因此在 HTTP 协议上建立的消息服务是可以提供响应的。事实上，Apache SOAP 框架要求消息服务提供响应。

让我们来为 urn:WeatherDiary 服务编写一个 Java 类。我们需要一个叫 recordTemperature 的方法，以便客户应用程序能够将它们的当前温度发送给服务。这里就是实现该服务的 javasoap.book.ch8.WeatherDiary 类：

注 1： 这个类存在于 SOAP RPC 包中，这是一个有趣的现象，但是这并不表示它只能用于 RPC 形式的服务；它也许应该被包含在另一个不同的包中。

```
package javasoap.book.ch8;
import java.io.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.SOAPContext;
import javax.mail.MessagingException;
import org.w3c.dom.*;
public class WeatherDiary {
   Hashtable _diary = new Hashtable();
   public void recordTemperature (Envelope env, SOAPContext
                     reqCtx, SOAPContext resCtx)
            throws Exception {

      Vector v = env.getBody().getBodyEntries();
      int cnt = v.size();
      String zipcode = null;
      String temperature = null;
      for (int i = 0; i < cnt; i++) {
         Element e = (Element)v.elementAt(i);
         String name = e.getTagName();
         if (name.equals("zipcode")) {
            zipcode = e.getFirstChild().getNodeValue();
         }
         else if (name.equals("temperature")) {
            temperature = e.getFirstChild().getNodeValue();
         }
      }
      if (zipcode == null || temperature == null) {
         throw new IllegalArgumentException(
               "ZIPCODE and/or TEMPERATURE Not Specified");
      }
      _diary.put(zipcode, temperature);
      resCtx.setRootPart("OK", "text/xml");
   }
}
```

我们希望得到SOAP体里的元素，于是我们就以env为参数调用了getBody()方法，它返回org.apache.soap.Body的一个实例。然后我们又调用了该体对象的getBodyEntries()方法，该方法返回一个包含org.w3c.dom.Element实例的java.util.Vector。矢量中的每一个Element封装了SOAP体的一个直接子元素。如果你看一下我们将要发送到服务的SOAP封套，就会看见我们在体元素中使用了三个直接子元素：recordTemperature、zipcode和temperature。这就是我们将在urn:WeatherDiary服务的recordTemperature方法中使用的格式。

我们遍历Vector以获取体元素的每个子元素。为获知我们都得到了何种类型的元素，我们调用了getTagName()方法，并检查其返回结果看是否是zipcode或temperature元素（我们也会得到recordTemperature元素，它用来指定服务与方法名称，但是在这个例子中我们没有用到它）。当我们找到一个感兴趣的元素标签时，我们就去取它的第一个（也是惟一一个）子节点，我们将在那里得到元素值的数据。该数据被进一步保存到局部变量zipcode和temperature中。当完成循环之后，我们还要检查这两个变量是否仍为null。如果是，则表明消息里至少缺少一个必需的元素，因此我们就要抛出一个异常。这个异常会让Apache SOAP向调用者返回一个SOAP错误。否则，我们就把该数据保存到一个散列表中，用邮政编码作为键，用温度作为其值。我们将在应用程序级作用域部署这个服务，因此同一个散列表的内容可以被多个调用使用。

最后一步是要设定一个返回值。这有点让人迷惑——为什么一个消息服务还需要返回值？不过，如果你不这样做，Apache框架会抛出一个异常。我在前面说过，Apache框架坚持要求消息服务返回一个响应消息，而且消息中还要有内容。最简单的办法是调用resCtx.setRootPart()方法来向响应内容的根部分添加一条简单的文本消息，并以它作为返回值。该方法的第一个参数是要发送的文本的内容；第二个参数是数据的MIME类型，在本例中是text/xml。我在这个例子中所做的是发回一个字符串“OK”来表示数据已收到。

现在让我们来部署这个服务。下面是它的部署描述文件：

```
<isd:service
    xmlns:isd="http://xml.apache.org/xml-soap/deployment"
    id="urn:WeatherDiary" type="message">
  <isd:provider
     type="java"
     scope="Application"
     methods="recordTemperature">
    <isd:java
       class="javasoap.book.ch8.WeatherDiary"
       static="false"/>
  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener
  </isd:faultListener>
  <isd:mappings>
```

```
    </isd:mappings>
  </isd:service>
```

这个部署描述文件与我们在前面RPC形式的服务中所创建的那些部署描述文件之间的惟一区别在于对 isd:service 元素中 type 属性的使用方法不同。这个属性的值现在被设定为 "message"，表示正在被部署的服务是一个消息服务，而不是一个 RPC 服务。

现在这个服务就可以接收不同地区的客户应用程序所发来的温度了。让我们来看一个能够发送这些数据的客户应用程序。我们在SOAP封套中建立了XML语句，然后将它发送到 urn:WeatherDiary服务。

```
package javasoap.book.ch8;
import java.io.*;
import java.net.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import org.apache.soap.*;
import org.apache.soap.messaging.*;
import org.apache.soap.transport.*;
import org.apache.soap.util.xml.*;
public class WeatherClient {
   public static void main (String[] args) throws Exception {
      DocumentBuilder xdb = XMLParserUtils.getXMLDocBuilder();
      Document doc = xdb.newDocument();

      Element elem =
            doc.createElementNS(Constants.NS_URI_SOAP_ENV,
            "SOAP-ENV:Envelope");
      doc.appendChild(elem);

      Element sub =
            doc.createElementNS(Constants.NS_URI_SOAP_ENV,
            "SOAP-ENV:Body");
      elem.appendChild(sub);
      elem = sub;
      sub = doc.createElement("recordTemperature");
      sub.setAttribute("xmlns", "urn:WeatherDiary");
      elem.appendChild(sub);
      sub = doc.createElement("zipcode");
      Text txt = doc.createTextNode("");
      txt.setData("12345");
```

```
        sub.appendChild(txt);
        elem.appendChild(sub);
        sub = doc.createElement("temperature");
        txt = doc.createTextNode("");
        txt.setData("52.3");
        sub.appendChild(txt);
        elem.appendChild(sub);
        Envelope msgEnv = Envelope.unmarshall(
                            doc.getDocumentElement());
        URL url = new URL(
          "http://georgetown:8080/soap/servlet/messagerouter");
        Message msg = new Message();
        msg.send(url, "", msgEnv);

        SOAPTransport st = msg.getSOAPTransport();
        BufferedReader br = st.receive();
        String line;
        while ((line = br.readLine()) != null) {
           System.out.println(line);
        }
     }
  }
```

代码的绝大部分并不是真正与SOAP相关的；它只是建立了一个要用作SOAP封套的XML文档。然而，我认为如果使用Apache的消息形式服务，你就可以实时地建立你的SOAP封套（注2）。现在让我们来看一下它是如何工作的。

第一步是取得org.w3c.dom.Document的一个实例，该实例封装了从根元素开始的整个XML文档。我们通过调用XMLParserUtils.getXMLDocBuilder()方法来得到一个DocumentBuilder，然后调用它的newDocument()方法，于是我们就得到了一个初始的Document对象。现在我们就能处理封套元素了，我们将把它作为文档的根。doc.createElementNS()方法产生了一个带有名称空间标识符的元素。该方法的第一个参数是从Constants.NS_URI_SOAP_ENV中得到的标准SOAP封套的名称空间http://schemas.xmlsoap.org/soap/envelope/。第二个参数是一个限定了的名称，也就是一个限定符加上一个元素名，中间用冒号(:)

注2：如果你不想实时地生成你自己的封套，你可能想要看看Apache SOAP所提供的一个消息服务例子。这个例子假定封套由其他系统产生，而且还会告诉你如何将它加载到你的客户应用程序并发送到服务。

隔开。本例中我们使用了限定符SOAP-ENV，它和前面例子中的是一致的，不过这个限定符只不过是个名称空间标识符；你可以使用任何名称。这个元素本身的名称就是Envelope。既然已经生成了SOAP封套，我们就需要通过调用doc.appendChild()方法把它加入到文档中。我们在创建Body元素时使用了相同的技术，只是这一次将它添加到了Envelope元素中，而不是文档本身中，这是因为SOAP Body是SOAP Envelope元素的子元素。

既然已经建立了Envelope和Body，现在我们就要来为这四个特殊的内容创建元素，并将这些元素添加进Body里。因此我们设置elem变量指向Body元素，然后使用sub变量来建立Body元素的子元素。为调用正确的服务方法，Body的第一个子元素必须被命名为recordTemperature。我们通过将元素名作为参数传递到doc.createElement()方法得到这个元素。得到的元素被保存在sub变量中。为了能调用正确的服务，我们还要将服务的名称作为xmlns属性的值。可以通过调用sub.setAttribute()方法来完成这项工作，所用的第一个参数是属性的名称，第二个参数是属性的值。recordTemperature元素中不包含任何数据或子元素，因此我们只要用elem.appendChild(sub)方法把它加入到Body元素中就可以了。

接下来我们要建立zipcode元素，这一次又要用到doc.createElement()方法。这个元素中没有属性，但需要为邮政编码添加数据节点。我们通过调用doc.createTextNode()方法产生了org.w3c.dom.Text的一个实例。Text类用于表示文本类型的数据，用在这里非常合适。接下来我们又依次调用txt.setData()方法来设置邮政编码值，使用sub.appendChild()方法来将这个文本节点添加到zipcode元素中，并将zipcode元素添加到Body元素中。我们使用完全相同的办法创建了temperature元素，然后把它也添加到Body元素中。

我们将要以org.apache.soap.messaging.Message实例的形式将消息发送给服务器，但首先需要把这个XML文档转换成org.apache.soap.Envelope的一个实例。我们通过调用Envelope.unmarshall()方法来完成这个工作，并将它作为文档的根元素。这看起来好像会增加一些开销。可以肯定生成Envelope对象会再一次被编组以发送到服务器。不管怎样，调用这个方法的结果是返回一个叫做msgEnv的Envelope实例。接下来我们又用处理消息服务路由的SOAP路由器地址建立了java.net.URL的一个实例。对于RPC服务，我们曾向*http://georgetown:*

8080/SOAP/servlet/rpcrouter 发送过 SOAP 请求；而对于消息服务，我们则要把请求发送到 *http://georgetown:8080/SOAP/servlet/messagerouter*。现在我们可以建立 Message 的一个实例并调用它的 send()方法了。URL 是第一个参数，而封套是最后一个参数。第二个参数是一个用来表示 SOAP 施动者的空字符串；因为我们在这里没有用到 SOAP 施动者，所以用一个空字符串就够了。

由于我们已经知道所采用的服务传输协议是 HTTP，而且我们的服务也能返回一个错误或者一个简单的“OK”字符串作为响应，那么在应用程序结束之前看一下响应会是个好主意。为显示响应信息，我们首先调用了msg.getSOAPTransport()，它给我们一个发送消息所使用的传输类org.apache.soap.transport.SOAPTransport的实例。如果该传输协议支持响应，就像 HTTP 那样，则你可以调用传输类的 receive()方法，该方法返回一个 BufferedReader 对象。如果传输协议不支持响应，receive()方法就会返回 null。一旦我们获得了 BufferedReader，我们所要做的工作就是将响应中的每一行打印到控制台。如果服务能处理该消息，则惟一被显示的就是一个“OK”字符串，否则，你会看到一个完整的SOAP错误信息。

既然我们把客户也放到了同一台机器上，那就让我们来看看它所发送的SOAP封套：

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <recordTemperature xmlns="urn:WeatherDiary"/>
    <zipcode>12345</zipcode>
    <temperature>52.3</temperature>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

现在我们已经有了一个可以收集信息的服务以及一个可以发送信息的客户。但我们还没有一个可以让人获取已收集信息的机制。我们当然可以扩充前面的那个服务，向其中添加一个方法来查询与某个指定邮政编码对应的温度。但是，请等一下……这听起来是一个 RPC 形式的服务，而不是一个消息服务。对于消息服务，返回一个像“OK，我收到你的消息了”这样的通用响应是一回事，而让它根据所收到的消息内容来返回特定的结果又是另外一回事。如果你正在使用请求/响应

模型，那为什么你不使用RPC形式的服务呢？许多开发者在对每个应用程序使用何种服务这个问题上有自己强烈的意愿，但假如没有“不是所有的消息传输都支持响应”这个限制的话，对于这个例子就没有理由不使用消息服务了。如果总是使用一种程序模型，就很容易陷入思维的陷阱。某些系统能受益于消息接发模型的自由度，在那里你不用受RPC服务中传递特殊参数等问题的限制。比如说，你也许想要向一个消息服务发送一个采购定单文档，并希望接收一个包含递送时间表的确认文档。这种交换当然可以用RPC实现，但按照消息交换来考虑它可能更容易。尽管SOAP RPC并不妨碍你开发无限制的参数传递模型，在这方面消息接发模式仍然更适合你的要求。因此就让我们来扩充我们的服务，增加一个返回XML文档的getTemperature()方法。

```
package javasoap.book.ch8;
import java.io.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.SOAPContext;
import javax.mail.MessagingException;
import org.w3c.dom.*;
import org.apache.soap.util.xml.*;
public class WeatherDiary {
   Hashtable _diary = new Hashtable();
   Hashtable _times = new Hashtable();
   public void getTemperature(Envelope env, SOAPContext reqCtx,
                              SOAPContext resCtx)
         throws Exception {

      Vector v = env.getBody().getBodyEntries();
      int cnt = v.size();
      String zipcode = null;
      for (int i = 0; i < cnt; i++) {
         Element e = (Element)v.elementAt(i);
         String name = e.getTagName();
         if (name.equals("zipcode")) {
            zipcode = e.getFirstChild().getNodeValue();
         }
      }
      if (zipcode == null) {
         throw new IllegalArgumentException(
                  "ZIPCODE Not Specified");
      }
      String temperature = (String)_diary.get(zipcode);
      String rectime = (String)_times.get(zipcode);
```

```
        String response = "<WeatherDiaryResponse>" +
                          (char)10 +
                          "   <temperature>" + temperature +
                          "</temperature>" +
                          (char)10 +
                          "   <recordTime>" + rectime +
                          "</recordTime>" +
                          (char)10 +
                          "</WeatherDiaryResponse>";
        resCtx.setRootPart(response, "text/xml");
    }
    public void recordTemperature (Envelope env, SOAPContext
                    reqCtx, SOAPContext resCtx)
      throws Exception {

        Vector v = env.getBody().getBodyEntries();
        int cnt = v.size();
        String zipcode = null;
        String temperature = null;
        for (int i = 0; i < cnt; i++) {
            Element e = (Element)v.elementAt(i);
            String name = e.getTagName();
            if (name.equals("zipcode")) {
                zipcode = e.getFirstChild().getNodeValue();
            }
            else if (name.equals("temperature")) {
                temperature = e.getFirstChild().getNodeValue();
            }
        }
        if (zipcode == null || temperature == null) {
            throw new IllegalArgumentException(
                  "ZIPCODE and/or TEMPERATURE Not Specified");
        }
        _diary.put(zipcode, temperature);
        _times.put(zipcode, new Date().toString());
        resCtx.setRootPart("OK", "text/xml");
    }
}
```

为使从服务方法返回的响应变得更有趣，我们把recordTemperature()方法改为能够将记录温度数据的时间保存到另一个散列表_times中，以邮政编码为键，以当前时间为值。不过最重要的修改还是添加了getTemperature()服务方法。这个方法一开始就从SOAP体中寻找一个zipcode元素，这个元素告诉该方法被请求的是哪一个邮政编码。如果找不到zipcode元素，该方法就抛出一个异常，这

将导致一个SOAP错误。一旦有了邮政编码，我们就可以从_daily和_times散列表的记录中分别取得温度和时间了。接下来，我们在一个叫做response的String变量里创建一个XML文档。这个文档很简单。它的根元素是一个叫WeatherDiaryResponse的元素，该元素又有两个子元素temperature和recordTime。这两个子元素中包含了与指定邮政编码对应的温度和时间。response字符串被用作resCtx.setRootPart()方法的第一个参数。其结果就是我们根据收到的消息返回一个包含被请求数据的XML文档作为响应。你需要重新部署这个服务，并向部署描述文件中添加getTemperature()方法。

现在让我们来建立一个能请求温度数据的客户应用程序。这个客户应用程序的创建过程与我们在其他的例子中所做的一样。这一次，我们只包含了zipcode元素和它的数据，而且它的第一元素叫做getTemperature，用于调用正确的服务方法。下面是该客户应用程序的代码：

```
package javasoap.book.ch8;
import java.io.*;
import java.net.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import org.apache.soap.*;
import org.apache.soap.messaging.*;
import org.apache.soap.transport.*;
import org.apache.soap.util.xml.*;
public class TemperatureClient {
   public static void main (String[] args) throws Exception {
      DocumentBuilder xdb = XMLParserUtils.getXMLDocBuilder();
      Document doc = xdb.newDocument();

      Element elem =
                doc.createElementNS(Constants.NS_URI_SOAP_ENV,
                "SOAP-ENV:Envelope");
      doc.appendChild(elem);

      Element sub =
                doc.createElementNS(Constants.NS_URI_SOAP_ENV,
                "SOAP-ENV:Body");
      elem.appendChild(sub);
      elem = sub;
      sub = doc.createElement("getTemperature");
```

```
        sub.setAttribute("xmlns", "urn:WeatherDiary");
        elem.appendChild(sub);
        sub = doc.createElement("zipcode");
        Text txt = doc.createTextNode("");
        txt.setData("12345");
        sub.appendChild(txt);
        elem.appendChild(sub);
        Envelope msgEnv = Envelope.unmarshall
                            (doc.getDocumentElement ());
        URL url = new URL(
            "http://georgetown:8080/soap/servlet/messagerouter");
        Message msg = new Message ();
        msg.send (url, "", msgEnv);

        SOAPTransport st = msg.getSOAPTransport ();
        BufferedReader br = st.receive ();
        String line;
        while ((line = br.readLine ()) != null) {
            System.out.println (line);
        }
    }
}
```

在运行这个应用程序之前，你还需要向服务发送一条与邮政编码12345对应的温度数据(因为重新部署服务销毁了我们原先所保存的数据)。向服务加入一些数据之后，再运行客户，它的输出应该是这样的:

```
<WeatherDiaryResponse>
   <temperature>52.3</temperature>
   <recordTime>Wed Oct 31 14:58:48 EST 2001</recordTime>
</WeatherDiaryResponse>
```

估计你现在已经能够描述出向getTemperature()方法发送的SOAP封套是怎样的了，因此在这里我们不再讨论它。惟一值得关注的是响应的封套。

也许让输出变得更加结构化会更好，刚才我们只是简单地转储了几个XML元素而已。因此现在我们来依据该响应生成一个文档。下面的代码替代了msg.send()调用之后的所有代码。在得到了BufferedReader之后我们还需要Document-Builder的一个实例。但是，现在我们与其一点一点地创建这个文档，倒不如使用前面的响应的内容。我们首先创建了org.xml.sax.InputSource的一个实例，并将BufferedReader传递给它的构造函数。现在我们用InputSource作为参数

调用了xdb.parse()方法。它将解析响应中的XML语句，并为我们生成一个Document实例。在取得根元素之后，我们将它打印出来以便确认我们取得了预期的元素。接着我们又通过调用getElementsByTagName("*")取得了根元素下所有子元素的一个清单，注意我们使用的是通配符(*)。这个方法将得到WeatherDiaryResponse根元素下的所有子元素。最后，我们遍历该清单，并将其中的每个元素及其对应数据都打印出来。

```
SOAPTransport st = msg.getSOAPTransport ();
BufferedReader br = st.receive ();
xdb = XMLParserUtils.getXMLDocBuilder();
InputSource is = new InputSource(br);
doc = xdb.parse(is);
elem = doc.getDocumentElement();
System.out.println("Root Element Is: " +
                                elem.getTagName());

NodeList children = elem.getElementsByTagName("*");
int cnt = children.getLength();
for (int i = 0; i < cnt; i++) {
   Node n = children.item(i);
   String name = n.getNodeName();
   String val = n.getFirstChild().getNodeValue();
   System.out.println(name + ": " + val);
}
```

运行修改后的客户应用程序将产生如下的输出：

```
Root Element Is: WeatherDiaryResponse
temperature: 52.3
recordTime: Wed Oct 31 14:58:48 EST 2001
```

这样，通过增加一点XML的处理工作，我们就能从消息形式的服务得到更有意义的数据。不过这要在底层传输协议支持响应的情况下才行。

直接编码

我们现在已经知道了如何在RPC形式的服务中使用SOAP，也知道了该如何在限制较少的SOAP消息接发服务中传输XML数据。但是还有另一种可能性，就是

在RPC服务中传递XML数据。问题在于RPC服务依赖于一种在XML元素与Java类之间进行转换的类型映射机制。如果能用RPC形式的方法调用来传递参数并用标准的XML返回，那当然好了。是的，你可以这样做。为了达到这样的目的，你需要使用XML直接编码，它告诉底层的SOAP框架该数据是XML，应该保持原样。

让我们回到金融世界。我们要建立一个服务，让它用XML格式提供上市公司的有关信息。我们把这个服务叫做urn:CorporateDataService，它有一个方法叫做getDataForSymbol。这个服务是一个RPC服务，而这个方法使用了一个股票代码作为参数。它返回的数据中可以包含公司名称、公司地址、公司成立年份、员工数目以及过去52周内股票的最高和最低价格。这个服务将从另一个保存了公司信息的系统中获得数据，该信息以XML格式保存在一个数据库中。因此格式化数据就不是服务的职责，服务只需要将它直接返回给调用者即可。这是一种非常适合使用XML直接编码的场合。如果服务需要以某种方式对这个数据进行处理，它就应该被设计成返回一个可以映射成Java类的复杂数据类型。使用直接编码的另外一个理由是客户应用程序可能希望以XML格式接收数据以供后台使用。这时在服务器端对XML进行转换只会浪费时间，而且客户端还要再花时间将它转换回XML。在这种情况下，我们就要利用SOAP来为访问已有数据提供支持，并依赖已有的软件系统来对数据进行处理。

为了创建一个具有适当签名的Java方法，我们需要一个封装XML文档的类，因为这个文档就是我们想要向调用者返回的东西。org.w3c.dom.Element类能够满足我们的要求，因此我们就让服务方法返回该类的一个实例。下面这段代码中的javasoap.book.ch8.CorporateDataService类实现了这个服务。该服务打开了一个文件，文件名是传给它的股票代码再加上一个*.xml*扩展名。该文件应该被保存在和这个类相同的目录中(我们假定某个其他的系统已经产生了可供服务载入的公司数据XML文件)。我们以该文件名作为构造函数的参数创建了一个java.io.FileReader。然后我们又建立了一个DocumentBuilder实例，并调用了它的parse()方法来将该文件的内容转换成一个XML文档对象。传递到parse()方法的参数是org.xml.sax.InputSource的实例，我们在创建这个实例时使用了FileReader作为其构造函数的参数。现在我们所要做的就是通过调用

doc.getDocumentElement()方法返回其根元素。你可以依照本书中其他的例子来部署本服务，将getDataForSymbol作为该服务的惟一服务方法。不必再做其他特殊的映射工作。

```
package javasoap.book.ch8;
import java.io.*;
import java.util.*;
import org.apache.soap.*;
import org.w3c.dom.*;
import org.apache.soap.util.xml.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
public class CorporateDataService {
   public CorporateDataService() {
   }
   public Element getDataForSymbol(String symbol)
         throws Exception {
      FileReader fr = new FileReader(symbol + ".xml");
      DocumentBuilder xdb = XMLParserUtils.getXMLDocBuilder();
      Document doc = xdb.parse(new InputSource (fr));
      if (doc == null) {
         throw new SOAPException(Constants.FAULT_CODE_SERVER,
                              "Invalid Data");
      }
      return doc.getDocumentElement();
   }
}
```

让我们来看一个包含了XML例子的数据文件，我们的服务将把该XML传递给调用者：

```
<corporateData>
   <symbol>MINDSTRM</symbol>
   <name>MindStream Software, Inc.</name>
   <address>
      <address1>111 Smithtown Bypass</address1>
      <address2>Suite 208</address2>
      <city>Hauppauge</city>
      <state>NY</state>
      <zip>11788</zip>
   </address>
   <incorpYear>1998</incorpYear>
   <numEmployees>unknown</numEmployees>
   <tradeHistory>
```

```
        <yearlyHigh>75.5</yearlyHigh>
        <yearlyLow>60</yearlyLow>
    </tradeHistory>
</corporateData>
```

现在我们可以写一个调用 getDataForSymbol 方法的客户应用程序:

```
package javasoap.book.ch8;
import java.io.*;
import java.util.*;
import java.net.*;
import org.w3c.dom.*;
import org.apache.soap.util.xml.*;
import org.apache.soap.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.rpc.*;
public class GetDataClient {
   public static void main(String[] args)
         throws Exception {
      URL url = new URL(
          "http://georgetown:8080/soap/servlet/rpcrouter");
      Call call = new Call();
      call.setTargetObjectURI("urn:CorporateDataService");
      call.setMethodName("getDataForSymbol");
      call.setEncodingStyleURI(Constants.NS_URI_LITERAL_XML);
      String symbol = "MINDSTRM";
      Vector params = new Vector();
      params.addElement(new Parameter("symbol", String.class,
                  symbol, Constants.NS_URI_SOAP_ENC));
      call.setParams(params);
      Response resp;
      try {
         resp = call.invoke(url, "");
      }
      catch (SOAPException e) {
         System.out.println(e.getMessage());
         return;
      }
      if (!resp.generatedFault()) {
         Parameter ret = resp.getReturnValue();
         Element bookEl = (Element)ret.getValue();
         System.out.println(DOM2Writer.nodeToString(bookEl));
      }
   }
}
```

我们需要将编码形式设定为XML直接编码，不仅是因为我们想让服务器以一个直接XML文档来串行化响应，而且因为我们想让响应也按照这种方式来解释。为了达到这个目的，我们用Constants.NS_URI_LITERAL_XML作为参数来调用call.setEncodingStyleURI()方法。在Apache SOAP中，这将设置Body元素的编码形式。也就是说，它为XML的最高层元素设置了encodingStyle属性。然而，我们不希望所传递的参数（股票代码）也被作为XML解释，毕竟它只是一个字符串而已。我们想对这个参数使用标准的SOAP编码，因此我们也将为symbol元素设置encodingStyle属性，并将属性值设为标准SOAP编码所对应的值。为指定这个参数的编码形式，我们将Parameter构造函数的第四个参数值设定为Constants.NS_URI_SOAP_ENC。在前面所有的例子中，我们都将这个值设为null，null表示在该参数中所代表的这一层元素上不需要encodingStyle属性，这个参数所在的层默认为采用与Body元素一致的编码形式。而为symbol元素显式地设定编码形式会覆盖Body的编码形式。

现在我们将像往常一样调用这个方法，但我们要把它的返回值类型转换为Element的一个实例，因为我们预期返回的值是直接XML类型。我们将得到的结果Element传递到DOM2Writer.nodeToString()方法，以便将这个XML打印到控制台。让我们先看一看被发送到服务的SOAP封套：

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:getDataForSymbol
xmlns:ns1="urn:CorporateDataService"
SOAP-ENV:encodingStyle="http://xml.apache.org/xml-soap/literalxml">
<symbol xsi:type="xsd:string"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
MINDSTRM
</symbol>
</ns1:getDataForSymbol>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

你可以看到，SOAP Body元素的encodingStyle属性被设置成了直接编码形式。而symbol元素又覆盖了这个设置，它使用了SOAP标准编码。下面是响应的封

套。你可以看到返回值也使用了XML直接编码，同时整个响应本身就是一个XML数据。

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:getDataForSymbolResponse
xmlns:ns1="urn:CorporateDataService"
SOAP-ENV:encodingStyle="http://xml.apache.org/xml-soap/literalxml">
<return>
<corporateData>
   <symbol>MINDSTRM</symbol>
   <name>MindStream Software, Inc.</name>
   <address>
      <address1>111 Smithtown Bypass</address1>
      <address2>Suite 208</address2>
      <city>Hauppauge</city>
      <state>NY</state>
      <zip>11788</zip>
   </address>
   <incorpYear>1998</incorpYear>
   <numEmployees>unknown</numEmployees>
   <tradeHistory>
      <yearlyHigh>75.5</yearlyHigh>
      <yearlyLow>60</yearlyLow>
   </tradeHistory>
</corporateData>
</return>
</ns1:getDataForSymbolResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

如果你想用直接编码的XML来作为参数发送到一个RPC服务方法，可以使用相同的技术。只需要在调用Parameter构造函数时指定Constants.NS_URI_LITERAL_XML编码形式，同时，也没有规定说一定要对每一个参数都使用相同的编码形式；你可以根据应用程序的要求去做。

带附件的SOAP

许多SOAP实现都允许你连同数据一起发送附件形式的SOAP封套。当数据给

SOAP处理器带来很大而且不必要的负担时，使用附件是很有好处的。举个例子，考虑一下．如何将一个音频文件或者视频文件作为某个SOAP消息的一部分来发送？它们常常是很大的文件，而且不能用XML来编码。你当然可以用base64这样的二进制编码方案来把它放到SOAP封套里。这样做的确没错，但是却事倍功半。既然你的SOAP引擎不需要对它做任何事情，那为什么还要让它处理这么大的二进制数据呢？将这些数据作为附件发送就不会导致这些开销，而且还可以不必使用XML结构。

让我们来创建一个用于二手车买卖的服务。我们使用了一些方法来列出要卖的汽车、并返回可供交易的汽车的信息。我们需要一个用于描述该列表的自定义类型；这个类型封装了汽车的所有细节和汽车卖家的联系方式信息。下面是我们用来传递这个信息的javasoap.book.ch8.Listing类。它包含了一个toString()方法，因此我们可以很轻易地显示它的内容。

```
package javasoap.book.ch8;
public class Listing {
   String _make;
   String _model;
   int    _year;
   int    _miles;
   String _ownerName;
   String _ownerEmail;
   public Listing() {
   }
   public Listing(String make, String model,
     int year, int miles, String ownerName, String ownerEmail) {
      setMake(make);
      setModel(model);
      setYear(year);
      setMiles(miles);
      setOwnerName(ownerName);
      setOwnerEmail(ownerEmail);
   }
   public void setMake(String make) {
      _make = make;
   }
   public String getMake() {
      return _make;
   }
   public void setModel(String model) {
      _model = model;
```

```
    }
    public String getModel() {
        return _model;
    }
    public void setYear(int year) {
        _year = year;
    }
    public int getyear() {
        return _year;
    }
    public void setMiles(int miles) {
        _miles = miles;
    }
    public int getMiles() {
        return _miles;
    }
    public void setOwnerName(String ownerName) {
        _ownerName = ownerName;
    }
    public String getOwnerName() {
        return _ownerName;
    }
    public void setOwnerEmail(String ownerEmail) {
        _ownerEmail = ownerEmail;
    }
    public String getOwnerEmail() {
        return _ownerEmail;
    }
    public String toString() {
        String result = String.valueOf(_year) + " " +
                        _make + " " + _model +
                        " with " + String.valueOf(_miles) +
                        " miles:" +
                        "\n   Owned by " + _ownerName + "(" +
                        _ownerEmail + ")";
        return result;
    }
}
```

Listing类有两个字符串属性Make和Model，它们分别表示汽车的制造商和汽车的型号名称。Year是一个整型属性，表示汽车的生产年份，Miles整型属性则表示汽车的已行驶里程。OwnerName和OwnerEmail都是字符串类型的属性，它们分别表示汽车拥有者的名字和email地址。属性的get和set方法遵循JavaBeans模式，因此我们就不必再为Listing类编写自定义串行化器了。

现在我们要创建一个类来实现urn:UsedCarListingService服务。addListing()(注3)方法将清单发送给服务，它以一个Listing实例作为它的第一个参数，并返回一个整数来代表这个清单的ID。我们将允许调用者在调用addListing()Web服务方法时附加一个图像文件，因此该方法的第二个参数就是一个用来封装这个附件的javax.activation.DataHandler实例。Apache SOAP已为DataHandler类提供了一个串行化器，并利用了JAF(JavaBeans Activation Framework、JavaBeans活动框架)来处理MIME附件。这个框架使得在RPC服务中接收附件就像传递参数一样简单。引用和提取附件的工作都已经为你做好了，你可以通过JAF API来处理数据。search()方法有两个字符串参数，分别表示汽车的制造商和汽车的型号名称，它返回一个包含清单ID的数组。最后，getListing()方法以一个清单ID作为参数，它返回Listing的一个实例；getImage()方法也同样以一个清单ID作为参数，并返回相关的作为附件的图像文件。这里就是javasoap.book.ch8.UsedCarListingService类的代码：

```
package javasoap.book.ch8;
import java.util.*;
import java.io.*;
import javax.activation.*;
import org.apache.soap.util.mime.*;
public class UsedCarListingService {
   Hashtable _listings = new Hashtable();
   int _nextId = 100;
   public int addListing(Listing listing, DataHandler handler)
                 throws Exception {
      _listings.put(new Integer(_nextId), listing);
      int listno = _nextId;
      _nextId++;
      if (handler != null) {
         String fname = String.valueOf(listno) + ".bin";
         DataSource ds = handler.getDataSource();
         ByteArrayDataSource bsource =
               new ByteArrayDataSource(ds.getInputStream(),
                                       handler.getContentType());
         bsource.writeTo(new FileOutputStream(fname));
      }
      return listno;
   }
```

注3： 当然使用removeListing()方法也是一个好主意，但它对我们这个例子并不重要。

```
    public Integer[] search(String make, String model) {
       Vector v = findListingIds(make, model);
       int cnt = v.size();
       if (cnt == 0)
          return null;
       Integer ids[] = new Integer[cnt];
       for (int i = 0; i < cnt; i++) {
          ids[i] = (Integer)v.elementAt(i);
       }
       return ids;
    }
    protected Vector findListingIds(String make, String model) {

       Vector result = new Vector();
       for (Enumeration e = _listings.keys() ;
                                e.hasMoreElements() ;) {
          Integer i = (Integer)e.nextElement();
          Listing listing = (Listing)_listings.get(i);
          if (make.equals(listing.getMake()) &&
              model.equals(listing.getModel())) {
             result.add(i);
          }
       }

       return result;
    }
    public Listing getListing(int id) {
       Listing listing = (Listing)_listings.get(new Integer(id));
       return listing;
    }
    public DataHandler getImage(int id)
         throws Exception {
       String fname = String.valueOf(id) + ".bin";
       DataSource ds =
            new ByteArrayDataSource(new File(fname), null);
       DataHandler dh = new DataHandler(ds);
       return dh;
    }
}
```

这个服务在应用程序级作用域操作，因此只要服务一直运行，所有的清单就都是可用的。我们不需要再去以某种类型来永久保存这些清单，尽管在现实世界中你应该这样做。附件文件是以文件形式保存在磁盘上的。我们将所有的清单存放在一个叫做_listings的散列表中，其键是独一无二的清单ID。新的ID值在每次调用addListing()方法时由递增_nextId值得到。addListing()方法的前半部

分将以作为参数传入的清单的形式保存到散列表中。散列表中的键是一个基于_nextId值的Integer，_nextId是顺序递增的。下一步是要提取出附件并将它的内容保存为一个本地文件。文件名是由清单号加上.*bin*扩展名组成的。我们通过调用handler参数的getDataSource()方法取得一个对javax.activation.DataSource的引用，然后建立一个org.apache.soap.util.mime.ByteArrayDataSource。其构造函数的第一个参数是数据的java.io.InputStream，它由调用ds.getInputStream()方法得到。另一个参数是一个字符串，它包含了数据的MIME类型，通过调用handler.getContentType()方法得到。现在我们可以将附件的内容保存到文件里了，方法是调用writeTo()方法，并向该方法传递java.io.FileOutputStream的一个实例，这个实例的构造函数以我们在前面生成的那个文件名作为参数。现在既然附件中的数据已被存为本地文件，我们就可以将清单号返回给调用者了。

search()方法调用了一个叫做findListingIds()的本地方法，该方法返回一个包含Integer对象的Vector对象。请注意findListingIds()并没有被暴露为服务方法。Vector中的每一项代表了与指定制造商和型号相符的汽车清单。接下来我们建立了一个Integer[]，其内容由Vector中的各项数据组成，并将它返回给调用者。这个数组允许调用者使用清单ID来获取实际的清单和相关的图像。

getListing()方法中惟一的参数是一个整型参数：清单ID。这个ID用来从散列表_listings中获取Listing对象。Listing对象接着被返回到调用者。getImage()方法也使用清单ID来作为一个整型参数，并用它来生成一个与清单相关的图像文件的名称。这个文件名用于实例化一个java.io.File，这个对象被用作ByteArrayDataSource类构造函数的参数。因为不必指定数据的内容类型，所以它的第二个参数为null。接下来我们又用数据源建立了一个新的DataHandler，然后将这个处理程序返回给调用者。现在你可以部署这个服务了，并暴露它的addListing、search、getListing和getImage服务方法。记住要为自定义的清单类型添加一个映射项，如下所示：

```
<isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         xmlns:x="urn:UsedCarListingService" qname="x:Listing"
         javaType="javasoap.book.ch8.Listing"
java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
```

现在让我们来建立一个能向服务发送清单的客户应用程序。我们只想要验证数据的正确性，所以我们不准备发送一个很大的二进制数据文件，相反将使用一个具有可读性的名为 *data.bin* 的文件：

```
1:Rob
2:Englander
3:MindStream Software, Inc.
```

下面是客户应用程序的代码：

```
package javasoap.book.ch8;
import java.io.*;
import java.util.*;
import java.net.*;
import org.apache.soap.util.mime.*;
import org.apache.soap.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.rpc.*;
import javax.activation.*;
import javax.mail.internet.*;
import org.apache.soap.util.xml.*;
public class AddListingClient {
   public static void main(String[] args)
         throws Exception {
      URL url = new URL(
            "http://georgetown:8080/soap/servlet/rpcrouter");
      Call call = new Call();
      call.setTargetObjectURI("urn:UsedCarListingService");
      call.setMethodName("addListing");
      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      SOAPMappingRegistry smr = new SOAPMappingRegistry();
      BeanSerializer beanSer = new BeanSerializer();
      call.setSOAPMappingRegistry(smr);
      smr.mapTypes(Constants.NS_URI_SOAP_ENC,
         new QName("urn:UsedCarListingService", "Listing"),
               javasoap.book.ch8.Listing.class,
               beanSer, beanSer);
      Vector params = new Vector();
      Listing listing = new Listing("Pontiac", "Firebird", 1968,
            105000, "Rob Englander", "rob@mindstrm.com");
      params.addElement(new Parameter("listing",
               javasoap.book.ch8.Listing.class, listing, null));
      DataSource ds = new
            ByteArrayDataSource(new File("data.bin"), null);
      DataHandler dh = new DataHandler(ds);
```

```
        params.addElement(new Parameter("handler",
                  javax.activation.DataHandler.class, dh, null));
        call.setParams(params);

        Response resp;
        try {
           resp = call.invoke(url, "");
        }
        catch (SOAPException e) {
           System.out.println(e.getMessage());
           return;
        }
        if (!resp.generatedFault()) {
           Parameter ret = resp.getReturnValue();
           System.out.println("The listing number is " +
                                  ret.getValue());
        }
        else {
           Fault fault = resp.getFault();
           System.out.println ("Fault Code = " +
                   fault.getFaultCode());
           System.out.println ("      String = " +
                   fault.getFaultString());
        }
     }
  }
```

客户应用程序首先建立了一个用于调用addListing服务方法的调用。既然我们将要传递Listing的一个实例，我们就把它映射到BeanSerializer的一个实例。建立这样一个清单：1968年生产的Pontiac Firdbird，汽车已行驶105000英里，拥有者是我自己。这是调用该方法的第一个参数。接下来，我们建立了ByteArrayDataSource的一个实例。它的第一个参数是以*data.bin*文件名为基础建立的一个java.io.File。我们并不需要指定数据的内容类型，因此我们又一次为第二个参数使用了null。数据源被用于创建一个DataHandler对象，这个对象将作为服务方法调用中的第二个参数。

运行这个例子会得到如下输出：

```
The listing number is 100
```

因为该服务被部署于应用程序作用域，所以清单号会在每次运行客户应用程序的时候加1。

被发送到服务器的SOAP封套中并不包含图像文件的内容，传输使用了MIME多段消息，其中SOAP封套是第一部分，而附件文件的内容是第二部分。因此handler参数的XML元素是一个对第二部分的引用，就像这样：

```
<handler href ="cid:6889270.1004989967097.apache-soap.georgetown"/>
```

当你运行这个例子的时候，引用的值可能会不同，因为它是实时生成的，而且包含了源的名称（此处的源是geotgetown）。但重要的是，附件文件的内容并没有出现在SOAP封套中。在这个例子里，附件的内容很小，但是你可以想像一下一个10M的音频文件或者一个汽车在车赛中的视频剪辑文件。如果你监听了在这个调用中发往服务器的HTTP流，就可以看到在SOAP封套之后有下面这样的内容：

```
Content-Type: application/octet-stream
Content-Transfer-Encoding: 8bit
Content-ID: <6889270.1004989967097.apache-soap.georgetown>
Content-Length: 52

1: Rob
2: Englander
3: MindStream Software, Inc.
```

这是附件在编码后的内容。在实际应用程序中，附件可能是一个包含了已编码二进制数据的巨大的流，但在这个例子中，我们只用了一条很短的消息，以便我们能够确认附件被正确地处理了。

现在我们来写一个能够获取清单及其相应图像文件的客户程序。首先我们要调用search方法，以“Pontiac”和“Firebird”作为参数。假设我们在前面只运行了一次AddListingClient客户程序，search方法返回的整型数组中就会只包含一个值为100的元素。这是返回的第一个也是惟一的一个清单项。我们将使用这个ID来获取清单的内容及其图像文件（以附件的形式）。这里是它的代码：

```
package javasoap.book.ch8;
import java.io.*;
import java.util.*;
import java.net.*;
import org.apache.soap.util.mime.*;
import org.apache.soap.*;
```

```
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.rpc.*;
import javax.activation.*;
import javax.mail.internet.*;
import org.apache.soap.util.xml.*;
public class SearchClient {
   public static void main(String[] args)
         throws Exception {
      URL url = new URL(
          "http://georgetown:8080/soap/servlet/rpcrouter");
      Call call = new Call();
      call.setTargetObjectURI("urn:UsedCarListingService");
      call.setMethodName("search");
      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      SOAPMappingRegistry smr = new SOAPMappingRegistry();
      BeanSerializer beanSer = new BeanSerializer();
      call.setSOAPMappingRegistry(smr);
      smr.mapTypes(Constants.NS_URI_SOAP_ENC,
          new QName("urn:UsedCarListingService", "Listing"),
          javasoap.book.ch8.Listing.class, beanSer, beanSer);
      Vector params = new Vector();
      String make = "Pontiac";
      String model = "Firebird";
      params.addElement(new Parameter("make",
               java.lang.String.class, make, null));
      params.addElement(new Parameter("model",
               java.lang.String.class, model, null));
      call.setParams(params);

      Response resp;
      try {
         resp = call.invoke(url, "");
      }
      catch (SOAPException e) {
         System.out.println(e.getMessage());
         return;
      }
      if (!resp.generatedFault()) {
         Parameter ret = resp.getReturnValue();
         int[] ids = (int[])ret.getValue();
         int cnt = ids.length;
         for (int i = 0; i < cnt; i++) {
            call.setMethodName("getListing");
            params = new Vector();
            params.addElement(new Parameter("id",
                java.lang.Integer.class, new Integer(ids[i]),
                null));
```

```
            call.setParams(params);
            try {
               resp = call.invoke(url, "");
            }
            catch (SOAPException e) {
               System.out.println(e.getMessage());
               return;
            }
            ret = resp.getReturnValue();
            Listing listing = (Listing)ret.getValue();
            System.out.println(listing);
            call.setMethodName("getImage");
            try {
               resp = call.invoke(url, "");
            }
            catch (SOAPException e) {
               System.out.println(e.getMessage());
               return;
            }
            ret = resp.getReturnValue();
            DataHandler handler = (DataHandler)ret.getValue();
            DataSource ds = handler.getDataSource();
            String fname = "EX17." +
                     String.valueOf(ids[i]) + ".bin";
            ByteArrayDataSource bsource =
                  new ByteArrayDataSource(ds.getInputStream(),
                                          handler.getContentType());
            bsource.writeTo(new FileOutputStream(fname));
            System.out.println(
                  "   Image File Stored In " + fname);
         }
      }
      else {
            Fault fault = resp.getFault();
            System.out.println ("Fault Code = " +
                    fault.getFaultCode());
            System.out.println ("      String = " +
                    fault.getFaultString());
         }
      }
   }
```

这段代码中的第一个部分我们现在应该已经很熟悉了。我们建立了对search方法进行调用的call对象、映射了Listing类型，并为查找Pontiac Firdbirds车而向该方法中添加了两个字符串参数。该调用的返回值被转换成了一个int[]类型(尽管服务类返回一个Integer[]，但因为Apache SOAP框架将一个原始数据元

素的数组映射为一个Java原始类型的数组，因此我们会得到一个int[])。现在我们可以循环遍历清单ID的数组。对其中的每一个我们都将方法名称设置为getListing，并使用从ids数组中取出的清单ID来为call建立一个Parameter。现在我们来调用这个方法，这一次要把它的返回结果的类型转换为Listing的一个实例。因为Listing已经实现了一个toString()方法，我们就可以直接将Listing对象传递到System.out.println()来显示它的内容。接下来，我们又将方法名称设置为getImage()并再次调用。这一次我们又将其返回值类型转换成了一个DataHandler并取得了它的DataSource。我们使用了与UsedCarListingService类中所使用的相同技术来将附件的内容写入到一个本地文件。文件名用*EX17.xxx.bin*的形式，其中的*xxx*就是清单ID。

最后一步是显示包含清单内容的文件的名称。其结果就像下面这样：

```
1968 Pontiac Firebird with 105000 miles:
   Owned by Rob Englander(rob@mindstrm.com)
   Image File Stored In EX17.100.bin
```

如果你查看文件*EX17.100.bin*的内容，就会发现它与最初的附件内容是一致的：

```
1: Rob
2: Englander
3: MindStream Software, Inc.
```

如果想在Apache SOAP消息形式的服务中使用附件，就必须在更低的层次工作，因为它没有为此提供像RPC级API那样的良好抽象。GLUE也支持SOAP附件，并使用非常直接的API。虽然我们不准备在这里介绍它，但我想GLUE抽象是很容易理解和使用的。

本章讨论了可供选择的SOAP技术，如果加上我们已经介绍的其他各项技术，你就可以使用SOAP来应付任何类型的数据传输模型了。

第九章

SOAP的互操作性与WSDL

到目前为止，我们所有的例子都对其服务和客户使用了相同的SOAP实现。我们的Apache SOAP服务器例子使用了Apache SOAP客户应用程序来访问，而GLUE服务器例子则用GLUE客户应用程序来访问。在这些不常见的场合中，当你可以控制分布式系统中每一个节点所使用的技术时，你可以很容易地让它们全都使用相同的技术。然而，这种机会是不常有的，而且它与在计算机世界中使用Web服务的初衷不相符。因此有必要了解一下SOAP实现之间是如何进行互操作的。

现在有几十个SOAP实现可供使用，而且每天还会有其他的工具不断涌现。随着时间的推移，你或许会发现大量的现有企业级系统都在利用某种SOAP机制，以使得其本身可以通过多种传输协议来被访问。一些系统的底层将会毫无疑问地使用SOAP，因此你可以不用去理会它。同时新的基于SOAP的分布式软件框架还将不断产生。我们怎么能期待这些系统之间都有良好的互操作性呢？毕竟，软件仍然是由正在以这种或那种形式彼此竞争的那些公司和个人开发的，这就通常会导致互操作性问题。有时候因为规范本身有一节或者多节是对解释开放的，而为不同组织工作的开发者会对事物做出不同的解释，所以就会导致问题。发生了这些问题并不能说明规范本身有缺陷；它只能说明软件开发最终是一个人类事业，而人是趋向于持有不同意见的。

一个经常导致互操作性问题的原因是SOAPAction字段，SOAP用它来发送信息给一个在SOAP封套范围之外的接收者。这个字段是作为底层传输的头信息的一部分被传输的；举例来说，当SOAP消息在HTTP上传输时，这个字段就被作为一个HTTP头来传送（SOAPAction在除了HTTP之外的传输协议中如何传输现在还不清楚）。该字段的内容是一个想要为接收者提供信息的URI。这个信息可能是服务的名称空间、服务的方法名称，或者别的东西。无论它是什么，它所想要的是向接收者提供一些有关封套内容的线索，而不需要接收者解析其XML。有些SOAP实现不需要在SOAPAction字段中发送任何数据值，而其他一些工具需要这样做。了解哪些内容在消息中是必要的并在消息中包含它们，这是实现SOAP互操作性的关键。本章我们将看到目前的一些SOAP实现对SOAPAction的要求，并通过例子来讨论用来设定字段值的API。但是首先，我们将介绍WSDL（Web Services Description Language，Web服务描述语言），这是一个重要的标准，它可以帮助解决一些互操作性问题。

Web服务描述语言

避免基于SOAP的服务的互操作性问题的一个方法是使用一个结构化的语言来描述该服务、它的位置、服务方法、参数、数据类型，等等。WSDL就做了这些工作。WSDL是描述Web服务的XML语法。系统能通过查看与服务相关的WSDL文档来确定Web服务的编程接口。该文档描述了服务方法，以及它们的参数和返回类型，同时还可能包括服务的地址或端点。WSDL的最大好处之一在于，它是描述Web服务的单一的、普遍承认的标准（注1），相对于其他的技术，它使得SOAP开发者避免使用他们自己的机制来完成一个任务。

WSDL是否已经消除了与人（或机器）的解释有关的问题了？嗯，它还没有完全解决这个问题。虽然它确实意味着对服务的描述远比没有标准时明白得多，但是在另一方面，WSDL也有它自己的一个规范，而且这个规范可能会被多个组织做出不同的解释。WSDL不是完美的解决方案，但是使用它所提供的技术确实比不使用它们要好得多。

注1： 你可以在*http://www.w3c.org/TR/wsdl*找到WSDL规范。

WSDL 概述

我们不准备详细地介绍WSDL的每个方面，因为它是一个很大的主题，本身就可以写成一本书。但由于它为实现SOAP的互操作性做出了重要的贡献，因此花些时间来看看它是很重要的。有一些SOAP实现（比如GLUE）吸收WSDL作为它们自己的一部分。本书中各个部分的GLUE例子都使用了wsdl2java工具程序来生成客户端的Java代码以访问一个基于GLUE的服务。wsdl2java读取一个描述服务的WSDL文档，并根据它创建Java接口和类文件。在那些例子里，我们不必真正去编写一个WSDL文件，因为GLUE会自动生成WSDL。GLUE客户端绑定过程中也使用了服务的WSDL；那就是它产生绑定的方式，这种绑定隐藏在你用来与服务进行交互的接口之后。

根据 WSDL 规范：

> WSDL 是一种 XML 格式，它将网络服务描述为一组端点，这些端点对包含面向文档的或者面向过程的信息的消息进行操作。操作和消息被抽象地描述，然后被绑定到一个具体的网络协议和消息格式来定义端点。相关的具体端点组合成为抽象的端点（即服务）。

一个 WSDL 文档是一个或多个服务定义的集合。该文档包含一个称为definitions的XML根元素；这个元素中包含了对服务的定义。definitions元素中也可以包含一个可选的targetNamespace属性，用来指定与服务定义有关的URI。在WSDL中使用名称空间和名称空间标识符的方法与我们在SOAP封套中的用法一样，因此在文档的definitions元素层次上出现几个名称空间定义是很常见的。

WSDL 将服务定义为网络端点的集合。这些端点在 WSDL 术语中也叫做 port。WSDL 将服务 port 和相关联的 message 从服务所绑定（binding）的网络协议中分离出来。将一个binding与一个网络地址组合起来就得到了一个port，而一个 service 就是这些 port 的集合。

WSDL 规范所定义的主要文件段如下：

types

使用某些类型系统（例如 XSD）来描述数据类型的一个容器。

message

对用于通信的数据的一个抽象的、已指定类型的定义。

operation

对服务所支持的行为的抽象描述。

portType

一个或多个端点所支持的抽象的操作集合。

binding

针对某个特定的端口类型的一个具体协议和数据格式规范。

port

它是一个单一的端点，定义为一个绑定和一个网络地址的组合。

service

一些相关端点的集合。

让我们来看一个能提供基于股票代码的非实时股票信息和新闻标题查询的服务。这个服务叫做urn:CorpDataServices。它有一个方法叫做getHeadLines，该方法只有一个代表股票代码的String类型的参数，它返回一个包含与该上市公司有关的新闻标题的String[]。这个服务中还有一个叫做getQuote的方法，它也以一个股票代码为参数并返回一个叫做Quote的自定义类型实例，这个实例中包含了股票代码、上一次交易的价格、相对于开盘价的涨跌金额、当日到目前为止的交易数量以及提供这些信息的时间戳。

下面的代码实现了Quote自定义类型。这个类放在service包中。我将服务器端和客户端的类分开了，因为我不打算让服务器与客户共享某个类；因此还有一个为客户端类编写的clients包。我们将使用前面介绍过的技术来映射那些客户代码要用到的自定义类型。

```
package javasoap.book.ch9.services;
import java.io.*;
import java.util.*;
import java.net.*;
public class Quote {
   String _symbol;
   float  _lastPrice;
```

```
    float  _change;
    String _timeStamp;
    long   _volume;
    public String getSymbol() {
       return _symbol;
    }
    public void setSymbol(String symbol) {
       _symbol = symbol;
    }
    public float getLastPrice() {
       return _lastPrice;
    }
    public void setLastPrice(float lastPrice) {
       _lastPrice = lastPrice;
    }
    public float getChange() {
       return _change;
    }
    public void setChange(float change) {
       _change = change;
    }
    public String getTimeStamp() {
       return _timeStamp;
    }
    public void setTimeStamp(String timeStamp) {
       _timeStamp = timeStamp;
    }
    public long getVolume() {
       return _volume;
    }
    public void setVolume(long volume) {
       _volume = volume;
    }
}
```

这是我们现在将要用到的惟一一个自定义类型；其余的数据均使用内置的数据类型。接下来我们需要一个实现这个服务的Java类：

```
package javasoap.book.ch9.services;
import java.util.Vector;
public class CorpDataService {
   public Quote getQuote(String symbol)
            throws Exception {
      QuoteParser qp = new QuoteParser();
      return qp.getQuote(symbol);
   }
```

```
    public String[] getHeadlines(String symbol)
            throws Exception {
        HeadlineParser hp = new HeadlineParser();
        Vector v = hp.getHeadlines(symbol);
        int len = v.size();
        String[] result = new String[len];
        for (int i = 0; i < len; i++) {
            result[i] = (String)v.elementAt(i);
        }
        return result;
    }
}
```

CorpDataService类中的getQuote()方法使用了一个叫做QuoteParser的类。这个类解析一个基于Web的查询服务数据流并将结果打包为javasoap.book.ch9.services.Quote的一个实例；不过，我不准备在这里列出该类的代码，因为它是我们所用的查询系统所特有的。我已经在*http://www.mindstrm.com/soap.htm*发布了这个服务，因此你将不必再去自己运行这个服务了（稍后我会为你提供访问该服务所需要的信息）。类似地，getHeadlines()方法中使用了一个叫做HeadlineParser的类，这个类以String类型收集新闻标题，并在java.util.Vector的一个实例中返回它们，然后根据这个矢量的内容建立一个String[]，最后将这个数组返回到调用者。

我决定使用GLUE来实现这个服务。这个服务位于*http://mindstrm.com:8004/glue/urn:CorpDataServices*（注2）。这里是我用来发布这个服务的javasoap.book.ch9.services.CorpDataServices 类：

```
package javasoap.book.ch9.services;
import electric.util.Context;
import electric.server.http.HTTP;
import electric.registry.Registry;
public class CorpDataServices {

    public static void main( String[] args )
                throws Exception {
        String ns = "urn:CorpDataServices";
        HTTP.startup("http://mindstrm.com:8004/glue");
```

注2： 请确认使用的是*mindstrm.com*而不是*www.mindstrm.com*，这两个URL不被解析为同一个地址。

```
        Context context = new Context();
        context.addProperty("activation", "application");
        context.addProperty("namespace", ns);
        context.addProperty("description", "Corporate Services Demo");
        Registry.publish(ns,
           Javasoap.book.ch9.services.CorpDataService.class, context);
    }
}
```

在运行 CorpDataServices 例子之后，就可以请求得到与这个服务相关联的WSDL文档。实际上我们在前面的每个GLUE例子中都是这样做的。绑定过程需要WSDL，而WSDL是服务器自动生成的。当然，你也可以手工建立你的WSDL；我比较喜欢让系统为我做这个工作。查看一个GLUE服务的WSDL的最简单方法就是使用浏览器。urn:CorpDataServices服务的 WSDL 位于 *http://mindstrm.com:8004/glue/urn:CorpDataServices.wsdl*。有些浏览器，比如Internet Explorer，会以一种可以展开各种节点的方式来显示以使你能够看到节点下面的内容。但不管你使用的是哪一种浏览器，我们的服务应用程序所返回的WSDL都是一样的：

```
<definitions name="CorpDataService"
  targetNamespace="http://www.themindelectric.com/wsdl/CorpDataService/"
  xmlns:tns="http://www.themindelectric.com/wsdl/CorpDataService/"
  xmlns:electric="http://www.themindelectric.com/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
<types>
  <schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://www.themindelectric.com/package/java.lang/"
  targetNamespace="http://www.themirdelectric.com/package/java.lang/">
    <complexType name="ArrayOfstring">
      <complexContent>
        <restriction base="soapenc:Array">
          <attribute ref="soapenc:arrayType" wsdl:arrayType="string[]" />
        </restriction>
      </complexContent>
    </complexType>
  </schema>
  <schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:tns=
```

```
      "http://www.themindelectric.com/package/javasoap.book.ch9.services/"
    targetNamespace=
      "http://www.themindelectric.com/package/javasoap.book.ch9.services/">
    <complexType name="Quote">
      <sequence>
        <element name="_symbol" nillable="true" type="string" />
        <element name="_lastPrice" type="float" />
        <element name="_change" type="float" />
        <element name="_timeStamp" nillable="true" type="string" />
        <element name="_volume" type="long" />
      </sequence>
    </complexType>
  </schema>
</types>
<message name="getQuote0SoapIn"> <part name="arg0" type="xsd:string" />
</message>
<message name="getQuote0SoapOut"> <part name="Result"
    xmlns:ns1=
      "http://www.themindelectric.com/package/javasoap.book.ch9.services/"
    type="ns1:Quote" />
</message>
<message name="getHeadlines1SoapIn"> <part name="arg0" type="xsd:string" />
</message>
<message name="getHeadlines1SoapOut">
  <part name="Result"
    xmlns:ns1="http://www.themindelectric.com/package/java.lang/"
    type="ns1:ArrayOfstring" />
</message>
<portType name="CorpDataServiceSoap">
  <operation name="getQuote" parameterOrder="arg0">
    <input name="getQuote0SoapIn" message="tns:getQuote0SoapIn" />
    <output name="getQuote0SoapOut" message="tns:getQuote0SoapOut" />
  </operation>
  <operation name="getHeadlines" parameterOrder="arg0">
    <input name="getHeadlines1SoapIn"
           message="tns:getHeadlines1SoapIn" />
    <output name="getHeadlines1SoapOut"
           message="tns:getHeadlines1SoapOut" />
  </operation>
</portType>
<binding name="CorpDataServiceSoap" type="tns:CorpDataServiceSoap">
  <soap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http" />
  <operation name="getQuote">
    <soap:operation soapAction="getQuote" style="rpc" />
    <input name="getQuote0SoapIn">
     <soap:body use="encoded" namespace="urn:CorpDataServices"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
```

```
    </input>
    <output name="getQuote0SoapOut">
     <soap:body use="encoded" namespace="urn:CorpDataServices"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
    </output>
  </operation>
  <operation name="getHeadlines">
   <soap:operation soapAction="getHeadlines" style="rpc" />
    <input name="getHeadlines1SoapIn">
     <soap:body use="encoded" namespace="urn:CorpDataServices"
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
    </input>
    <output name="getHeadlines1SoapOut">
      <soap:body use="encoded" namespace="urn:CorpDataServices"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
     </output>
   </operation>
 </binding>
 <service name="CorpDataService"> <documentation>Corporate Services Demo
   </documentation>
   <port name="CorpDataServiceSoap" binding="tns:CorpDataServiceSoap">
     <soap:address
       location="http://mindstrm.com:8004/glue/urn:CorpDataServices" />
   </port>
 </service>
 </definitions>
```

让我们将由urn:CorpDataServices服务产生的WSDL浏览一遍。考虑到我们只使用了两个服务方法和一个自定义数据类型，它显得有点长（即使是对于极为简单的服务，其WSDL文档都会显得比较长而且难懂。这就是你最好使用一个工具来产生WSDL的原因）。图9-1显示了我们这个服务的WSDL文档的高层次的结构。某些元素之后的“(n)”表示在某种类型中可能有不止一个元素。

definitions元素是这个文档的根元素；其他所有内容都在这个元素之下。name属性代表所包含的服务定义的名称，在本例中是CorpDataService。请注意它不是服务的名称urn:CorpDataServices，而是用来描述服务定义的名称。这个名称来自于实现服务的Java类javasoap.book.ch9.services.CorpDataService的名称。targetNamespace属性声明了与这个服务定义有关的名称空间。它的默认名称通过在http://www.themindelectric.com/wsdl之后加上服务名称来自

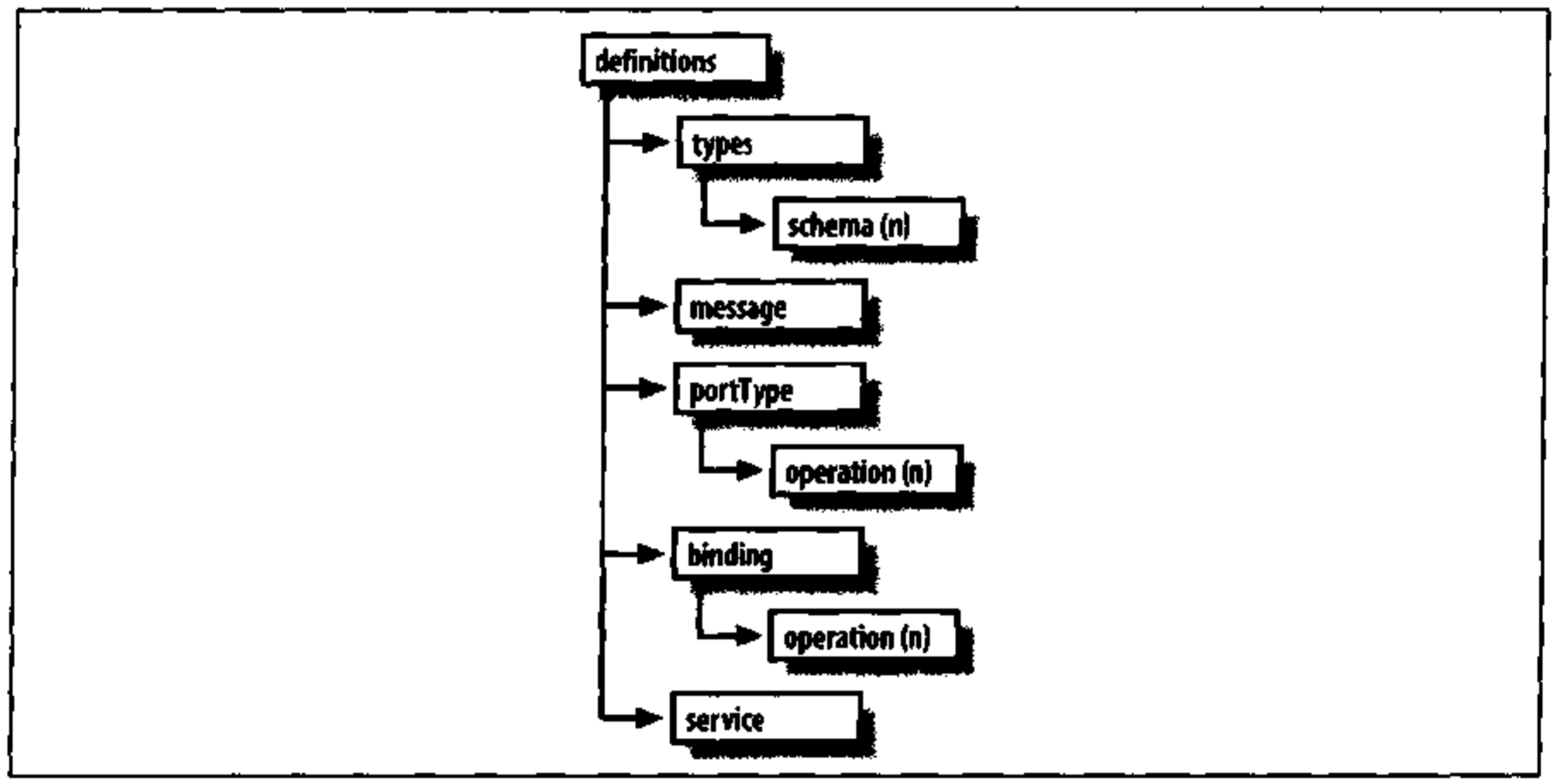

图 9-1：一个 WSDL 文档的高层次结构

动生成（注3）。definitions元素中包括了许多名称空间标识符的声明，它们全部都是自解释的。最后一个属性将整体名称空间设为了http://schemas.xmlsoap.org/wsdl，它反映了 WSDL 自己的名称空间。

下一个是types元素。这个元素定义了服务中要用到的自定义类型。每个自定义型用一个schema元素来定义，而每个schema元素都是types元素的直接子元素。第一个schema用来定义getHeadlines服务方法返回的字符串数组类型。这个字符串数组是一个complexType；请记住即使是一个数组也会被看成是复杂类型，因为它里面包含了多个值。它的name属性值为"ArrayOfstring"。在complexType中，我们可以看到一个名为complexContent的元素，它将这个类型限定为字符串数组。第二个schema元素有一个名称属性值“Quote”。这是一个用来从服务中返回股市报价数据的自定义类型，它也使用complexType元素来定义。在本例中，complexType包含了一个sequence元素，该元素又依次包含

注 3： GLUE在它在WSDL中所生成的很多名称空间定义中都使用了www.themindelectric.com。你可以覆盖这个targetNamespace，方法是：以“targetNamespace”作为属性名称，以你希望使用的值作为第二个参数，调用context.addProperty()方法。然而，这样做不会对WSDL的其他节使用的www.themindelectric.com产生影响。

了自定义的查询类型中的各个数据字段。每个字段用一个 element 标签来定义，并包含一个 name 属性以指定数据字段的名称。GLUE 使用了 javasoap.book.ch9.services.Quote 类中实际成员的名称。此外还指定了每个元素的类型。

接下来是message元素。这些元素定义了要在客户和服务器之间传递的消息。每个 message 包含一个 name 属性，它由服务方法名加上一个数字组成，加上数字是为了确保这个 name 是独一无二的。最后，如果这个 message 是一个 in 类型的参数，GLUE 就会为这个 name 再加上 SoapIn。若它是一个 out 类型的参数，则会为 name 加上 SoapOut。从 message 元素的观点来说，不存在 in/out 类型的参数；所有的消息要么被传入服务，要么被服务传出。

每个 message 中包含 0 或多个 part 元素。像我们这个例子这样的 RPC 形式的服务，对于它的传入消息，每一个 part 代表了一个方法参数。服务的 out 参数不会在一个传入消息的描述中出现，因为没有相关值会从客户传递到服务器。在这个例子中，名为 getQuote0SoapIn 的消息中只包含了一个名为 arg0 的 String 类型的 part。这个消息 part 对应于 getQuote 服务方法的第一个（也是惟一的）参数：一个包含了股票代码的字符串。GetQuoteSoapOut 消息中也只包含了一个 part，这一次叫做 Result。它是 getQuote 方法的返回值。要注意这个 Result 被定义为一个 Quote 类型的实例，并使用 http://www.themindelectric.com/package/javasoap.book.ch9.services/ 作为名称空间限定。它引用了在前面的 WSDL 文档中定义的自定义报价类型。getHeadLines 消息遵循同样的模式，因此我们不再详述。如果在这些消息中存在任何返回了值的 out 或 in/out 参数，那就要在对应的 xxxSoapOut 消息中的 Result 部分之后紧接着将它定义为 part 元素。举例来说，如果 getQuote 方法使用了一个用来代表股票代码的 in/out 参数，用作响应的 message 元素就应该是这样的：

```
<message name="getQuote0SoapOut">
 <part name="Result"
 xmlns:ns1="http://www.themindelectric.com/package/javasoap.book.ch9.services/"
    type="ns1:Quote" />
 <part name="arg0" type="xsd:string" />
</message>
```

portType元素是一个定义了一组操作及其相关消息的抽象。我们的portType中有一个值为"CorpDataServiceSoap"的name属性。在这个文件中，该name属性由在前面用过的名称之后加上"Soap"组成（译注1）。PortType元素又包含了两个operation元素，每一个都对应于一个服务方法。第一个包含了一个值为"getQuote"的name属性，这是服务的名称（译注2）。它还有一个parameterOrder属性，用来指定方法参数出现的顺序。因为getQuote方法只有一个参数，故其值为"arg0"。对于那些包含了多个参数的操作，parameterOrder属性的值将依次列出所有的参数，中间用空格隔开。每个operation都会包含单个input元素和单个output元素，它们每个又包含一个name和一个message属性。name属性中包含该消息的名称，而message属性中则包含了在前面的WSDL中定义的message元素的名称空间限定名。

WSDL文档中的下一个主要部分是binding段。这个部分将抽象的portType和一个实际的协议结合起来。binding元素的name属性同样又是CorpDataServiceSoap。type属性指定本binding中实现的端口类型；在本例中它引用了在前一个WSDL部分中定义的portType元素。在binding元素之下我们会发现另一个binding标签，不过它是用文档根元素中定义的soap名称空间标识符来进行名称空间限定的。这个元素指定了SOAP绑定信息。style属性的值为"rpc"，表明这是一个RPC形式的服务绑定。transport属性则指定了底层的传输协议；在本例中，传输协议是http://schemas.xmlsoap.org/soap/http，它表示HTTP传输协议。在soap:binding元素之后是代表在binding中出现的服务操作或者方法的operation元素。我们来看第一个，它有一个值为"getQuote"的name属性。它包含一个带有两个属性的soap:operation元素。其style属性的值也是"rpc"，代表这个绑定描述了一个RPC形式的SOAP调用。SoapAction的属性值为"getQuote"，它是该方法本身的名称。这是一个不起眼的问题，但对于SOAP的互操作性却是一个重要的部分。在调用服务时为SOAPAction头字段发送正确的值是非常必要的；如果你不注意这些，在使用某些SOAP实现时就会

译注1：对照前文，此处应该是指definitions元素的name属性值。

译注2：应为服务方法的名字，不过原文如此。

遇到问题。在这个例子中，这个属性告诉我们服务要求被调用的方法名应包含在SOAPAction头字段中。如果你提供了不正确的头，它就不能正常工作。每个operation中还包含一个input和一个output元素。这两个元素的结构是一样的，因此我们只需要介绍input就可以了。input元素有一个值为"getQuote0-SoapIn"的name属性。还记得这个名称吗？这是在WSDL中前面的部分定义过的消息的名称；现在我们要将这个消息与本次绑定的操作关联起来。还有一个soap:body元素，它用于指定这个绑定的名称空间以及所使用的编码类型。你可以看到这个例子使用了标准的SOAP编码，而其名称空间就是服务名称本身：urn:CorpDataServices。我们在CorpDataServices类中已经通过调用context.setProperty()方法设定了这个名称空间。

我们这个WSDL例子中的最后一部分是service元素。这个元素将一个绑定和一个特定的地址组合起来。它的name属性就是服务的名称，本例中是CorpDataService。随后是一个带有对本服务进行简单文字描述的"documentation"标签。这个元素下所包含的数据来自我们为发布这个服务所创建的CorpDataServices应用程序中的context.addProperty("description", "Corporate Service Demo")方法调用。如果你没有提供一个服务描述，GLUE就会根据实现服务的Java类自动生成一个描述。最后一个元素是port，我们用它来将绑定和特定的地址组合起来。它的name属性值为"CorpDataServiceSoap"。尽管这只是一个名称，而且本身并没有什么含义，但在目前阶段的WSDL中，我喜欢用这种方式来命名，因为它从字面上就能告诉我现在这个CorpDataService服务是在使用一个SOAP绑定。不管用什么方式，binding属性现在已被设为tns:CorpDataServiceSoap了，它引用了前面的binding元素，带有一个包含本服务实际地址的location属性。对于我们的服务，这个地址是*http://mindstrm.com:8004/glue/urn:CorpDataServices*，这是我们发布这个服务的URL。

从 Apache SOAP 客户调用 GLUE 服务

既然现在我们已经让这个GLUE服务运行起来了，那么就让我们通过一个用Apache SOAP编写的客户应用程序来访问它。这个服务发布了WSDL，这当然很不错，但Apache SOAP用不上这些数据。因此我们只好自己来从这个WSDL

中获取我们需要的数据信息了。在我们开始之前，先要编写一个简单的应用程序来尝试调用 urn:CorpDataServices 服务的 getHeadlines 方法。

首先我们需要确定正确的URL。在前面调用Apache SOAP服务的时候，这个URL指向的是RPC路由器 servlet。而现在，我们使用的是一个GLUE上的服务，因此这个 URL 就是 *http://mindstrm.com:8004/glue/urn:CorpDataServices*。

对纯粹的 Apache SOAP 例子来说，我们一直是调用 call.setTarget-ObjectURI()并将服务名称作为参数传递给它，而现在我们是在前面的URL中指定服务名称。这种方法是正确的，不过我们在调用 setTargetObjectURI()方法时却需要给出参数值，否则我们就会得到一个客户端错误。好在既然我们已经可以达到目的端点，GLUE 就不会理睬这个方法所使用的参数值。因此在调用 setTargetObjectURI()方法时，就可以使用任何非空字符串作为参数。我在这里仍然倾向于使用服务名称，但不管用什么参数都无关紧要。下面是 Apache2Glue 应用程序的代码:

```
package javasoap.book.ch9;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.util.xml.*;
import org.w3c.dom.*;
public class Apache2Glue {
   public static void main(String[] args)
      throws Exception {

      URL url =
         new URL("http://mindstrm.com:8004/glue/urn:CorpDataServices");

      Call call = new Call();
      call.setTargetObjectURI("XYZ");

      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      String stock = args[0];
      Vector params = new Vector();
      params.addElement(new Parameter("stock", String.class,
                                 stock, null));
      call.setParams(params);
```

```
        try {
            call.setMethodName("getHeadlines");
            Response resp = call.invoke(url, "");
            if (resp.generatedFault()) {
                Fault fault = resp.getFault();
                String code = fault.getFaultCode();
                String desc = fault.getFaultString();
                System.out.println(code + ": " + desc);
                Vector v = fault.getDetailEntries();
                int cnt = v.size();
                for (int i = 0; i < cnt; i++) {
                    Element n = (Element)v.elementAt(i);
                    Node nd = n.getFirstChild();
                    System.out.println(n.getNodeName() + ": " +
                                       nd.getNodeValue());
                }
            }
            else {
                Parameter ret = resp.getReturnValue();
                String[] value = (String[])ret.getValue();
                int cnt = value.length;
                for (int i = 0; i < cnt; i++) {
                    System.out.println(value[i]);
                }
            }
        }
        catch (SOAPException e) {
            System.err.println("Caught SOAPException (" +
                               e.getFaultCode() + "): " +
                               e.getMessage());
        }
    }
}
```

其余的代码与我们访问一个 Apache SOAP 服务时所用的是一样的。编写这样的代码并不困难。只有一个问题：要弄清服务的地址。如果你回去看一下这个服务的WSDL，你会发现答案就在眼前：service段内soap:address元素的location属性就是我们所要的URL。GLUE不关心目标对象的URI，这个现象很有趣，但它的确不会造成任何问题。

如果你用一个有效的股票代码作为命令行参数来运行Apache2Glue应用程序，它会返回一些与该股票代码相关的新闻。比如，下面的命令将查询与通用汽车有关的新闻标题：

```
java javasoap.book.ch9.clients.Apache2Glue GM
```

这里它的结果:

```
GMAC files shelf for $8.1 billion in notes
Vivendi Pays $1.5B for EchoStar Slice
Product Is King--Again
GM plans to form venture with SAIC-Wuling in China
Vivendi Pays $1.5B for EchoStar Slice
Daewoo Mtr makes first loan payment in a year-report
'Twas the Night Before Q4
Valeo Says U.S. Unit Filed for Bankruptcy
Honda's tank in the SUV wars
Daewoo Says to Resume Operations Monday
```

因为我的服务是从一个实时的数据源获取数据的,所以当你运行这个程序的时候,将得到不同的结果。

现在既然已经有了一个能工作的客户,那么就让我们来修改它,让它能够调用getQuote()服务方法。在这种情况下,我们需要处理一个自定义数据类型。因此,在继续深入之前,我们必须先创建一个报价(Quote)类,以便能够映射出服务方法的返回值。WSDL文档中为我们描述了自定义的报价类型。这里是该文档的相应部分:

```
<complexType name="Quote">
  <sequence>
    <element name="_symbol" nillable="true" type="string" />
    <element name="_lastPrice" type="float" />
    <element name="_change" type="float" />
    <element name="_timeStamp" nillable="true" type="string" />
    <element name="_volume" type="long" />
  </sequence>
</complexType>
```

没有必要编写一个自定义的串行化器来处理它;我们可以编写一个带有适当set和get方法的Java bean。它的属性名和数据类型必须与Quote类型中的各个element的name和type属性一致。对于本例而言,这个类中就必须带有一个名为_symbol的属性,因此我们就需要一个get_symbol()方法和一个set_symbol()方法。我们还加入了一个toString()方法来使得从服务取得数据之后输出该对象更方便。下面就是客户端Quote类的代码:

```
package javasoap.book.ch9.clients;
import java.io.*;
import java.util.*;
import java.net.*;
public class Quote {
   String _symbol;
   float  _lastPrice;
   float  _change;
   String _timeStamp;
   long   _volume;
   public String get_symbol() {
      return _symbol;
   }
   public void set_symbol(String symbol) {
      _symbol = symbol;
   }
   public float get_lastPrice() {
      return _lastPrice;
   }
   public void set_lastPrice(float lastPrice) {
      _lastPrice = lastPrice;
   }
   public float get_change() {
      return _change;
   }
   public void set_change(float change) {
      _change = change;
   }
   public String get_timeStamp() {
      return _timeStamp;
   }
   public void set_timeStamp(String timeStamp) {
      _timeStamp = timeStamp;
   }
   public long get_volume() {
      return _volume;
   }
   public void set_volume(long volume) {
      _volume = volume;
   }
   public String toString() {
      String result = "Symbol:     " + get_symbol() +
                      "\nLast Price: " + get_lastPrice() +
                      "\nChange:     " + get_change() +
                      "\nTime Stamp: " + get_timeStamp() +
                      "\nVolume:     " + get_volume();
      return result;
   }
}
```

现在我们准备要修改客户程序了。我们将把新的类称为Apache2Glue_v2。我们要把从getQuote方法返回的值映射到Quote类。为实现这个映射，我们需要知道该数据类型来自哪个名称空间。而WSDL又一次为我们提供了这个信息:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:tns=
    "http://www.themindelectric.com/package/javasoap.book.ch9.services/"
  targetNamespace=
    "http://www.themindelectric.com/package/javasoap.book.ch9.services/">
  <complexType name="Quote">
    <sequence>
      <element name="_symbol" nillable="true" type="string" />
      <element name="_lastPrice" type="float" />
      <element name="_change" type="float" />
      <element name="_timeStamp" nillable="true" type="string" />
      <element name="_volume" type="long" />
    </sequence>
  </complexType>
</schema>
```

该WSDL文档中为自定义报价类型定义的targetNamespace属性值为http://www.themindelectric.com/package/javasoap.book.ch9.services/。在将数据映射到我们的报价类时，我们在Qname构造函数中使用了这个名称空间，不过在这里谈这个有点为时过早。客户建立了SOAPMappingRegistry的一个实例和BeanSerializer的一个实例，就像在第五章中一样。然后我们使用了该映射注册表的mapTypes()方法来将服务方法getQuote返回的对象映射到我们的Quote对象。接下来，我们使用标准的方法建立了对该方法的调用，然后执行这个调用，并将得到的Quote对象传递给System.out.println()方法。这里是Apache2Glue_v2的代码:

```
package javasoap.book.ch9.clients;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
import org.w3c.dom.*;
public class Apache2Glue_v2 {
   public static void main(String[] args)
      throws Exception {
```

```
URL url =
    new URL("http://mindstrm.com:8004/glue/urn:CorpDataServices");
SOAPMappingRegistry smr = new SOAPMappingRegistry();
BeanSerializer beanSer = new BeanSerializer();
smr.mapTypes(Constants.NS_URI_SOAP_ENC,
  new QName(
    "http://www.themindelectric.com/package/javasoap.book.ch9.services/",
    "Quote"),
    Quote.class, beanSer, beanSer);

Call call = new Call();
call.setSOAPMappingRegistry(smr);
call.setTargetObjectURI("XYZ");
call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
String stock = args[0];
Vector params = new Vector();
params.addElement(new Parameter("stock", String.class,
                                stock, null));
call.setParams(params);

try {
   call.setMethodName("getQuote");
   Response resp = call.invoke(url, "");
   if (resp.generatedFault()) {
      Fault fault = resp.getFault();
      String code = fault.getFaultCode();
      String desc = fault.getFaultString();
      System.out.println(code + ": " + desc);
      Vector v = fault.getDetailEntries();
      int cnt = v.size();
      for (int i = 0; i < cnt; i++) {
         Element n = (Element)v.elementAt(i);
         Node nd = n.getFirstChild();
         System.out.println(n.getNodeName() + ": "
                                    + nd.getNodeValue());
      }
   }
   else {
      Parameter ret = resp.getReturnValue();
      Quote value = (Quote)ret.getValue();
      System.out.println(value);
   }
}
catch (SOAPException e) {
   System.err.println("Caught SOAPException (" +
                    e.getFaultCode() + "): " +
                    e.getMessage());
}
```

```
    }
}
```

下面是它的输出:

```
Symbol:      GM
Last Price:  47.28
Change:      -0.2
Time Stamp:  4:03PM
Volume:      3490100
```

一个使用 Apache SOAP 的代理服务

在本章的其余部分，我将继续使用 *http://mindstrm.com:8004/glue/urn:CorpDataServices* 作为服务的后台。在下面这个例子里，我们将创建一个名为 urn:QuoteProxyService的Apache SOAP服务，它能够从我们前面使用的基于GLUE的服务中获取数据。换句话说，这个新的 Apache SOAP 服务将可以作为一个代理来获取基于 GLUE 的服务的报价数据。其结构如图 9-2 所示。

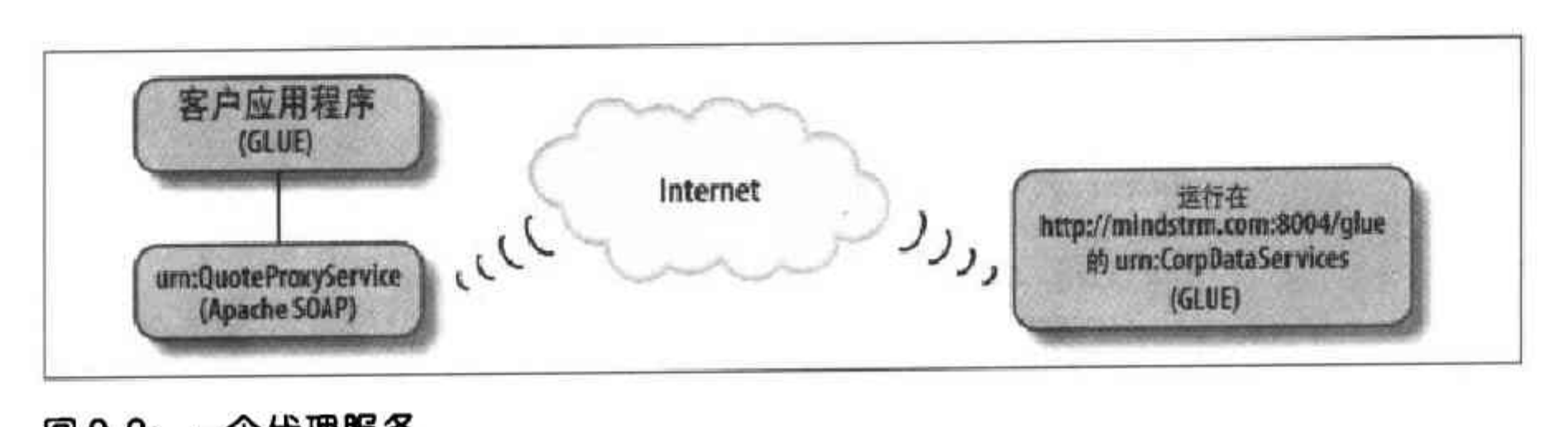

图 9-2: 一个代理服务

我们不准备简单地复制来自urn:CorpDataServices的getQuote()服务方法，而将做一些有趣的尝试。毕竟，我们可以随心所欲地设计我们想要的代理服务；它从其他服务获取数据的这一事实只是一个实现细节。因此，让我们向urn:QuoteProxyService添加一个可以在一次调用中处理多个股票代码的方法，这会使它变得更有价值。对于客户应用来说，能一次使用多个股票代码也确实更加方便。这个方法名叫getQuotes()；它接收一个字符串数组作为参数。数组中的每一个元素都是一个股票代码，而且方法的返回值也是一个报价的数组。这个新服

务将不代理 urn:CorpDataServices 服务的 getHeadlines()方法。这样做没有特别的理由；这仅仅是我的选择，我喜欢这样设计我的代理。

这一次我们又需要一个可以由代理服务返回的Java 报价类。为了与我们的包里的命名惯例一致，这个类将隶属于 javasoap.book.ch9.services包。为避免与前面的Java 报价类混淆，我们将它命名为 ProxyQuote。这个类中包含了与其他的报价类一样的信息，但其属性名和访问方法不一样。这样做的惟一原因就是为了防止混淆；名称本身是无关紧要的。这里是 ProxyQuote 类的代码:

```
package javasoap.book.ch9.services;
import java.io.*;
import java.util.*;
import java.net.*;
public class ProxyQuote {
   String _sym;
   float  _last;
   float  _diff;
   String _time;
   long   _vol;
   public String getStockSymbol() {
      return _sym;
   }
   public void setStockSymbol(String symbol) {
      _sym = symbol;
   }
   public float getLast() {
      return _last;
   }
   public void setLast(float last) {
      _last = last;
   }
   public float getDiff() {
      return _diff;
   }
   public void setDiff(float diff) {
      _diff = diff;
   }
   public String getTime() {
      return _time;
   }
   public void setTime(String time) {
      _time = time;
   }
   public long getVol() {
```

```
        return _vol;
    }
    public void setVol(long vol) {
        _vol = vol;
    }
}
```

现在我们有了一个表示urn:QuoteProxyService服务中报价数据的Java类了。但不要忘了这个服务还是urn:CorpDataService服务的一个客户。当我们为这个服务编写一个 Apache SOAP 客户程序的时候，我们仍需要一个用于映射返回报价数据的Java 类。我们借用了Quote的代码：仅将它的包名改为javasoap.book.ch9.services，删掉了toString()方法，并将它的名称改成了RemoteQuote。修改后的代码如下：

```
package javasoap.book.ch9.services;
import java.io.*;
import java.util.*;
public class RemoteQuote {
    String _symbol;
    float  _lastPrice;
    float  _change;
    String _timeStamp;
    long   _volume;
    public String get_symbol() {
        return _symbol;
    }
    public void set_symbol(String symbol) {
        _symbol = symbol;
    }
    public float get_lastPrice() {
        return _lastPrice;
    }
    public void set_lastPrice(float lastPrice) {
        _lastPrice = lastPrice;
    }
    public float get_change() {
        return _change;
    }
    public void set_change(float change) {
        _change = change;
    }
    public String get_timeStamp() {
        return _timeStamp;
    }
```

```
    public void set_timeStamp(String timeStamp) {
        _timeStamp = timeStamp;
    }
    public long get_volume() {
        return _volume;
    }
    public void set_volume(long volume) {
        _volume = volume;
    }
}
```

现在我们已经拥有了所有必要的支持类，可以为urn:QuoteProxyService编写代码了。这个Java类被命名为QuoteProxyService，而且其代码相当简单。这个服务暴露了一个叫做getQuotes()的方法，它根据股票代码数组返回任意数量的报价。这个方法返回一个ProxyQuote[]，这是一个ProxyQuote实例的数组，每一个元素对应于一个股票代码。其中的单个报价是由getStockQuote()方法取得的，这个方法以一个股票代码作为参数并返回基于这个股票代码的单个ProxyQuote。每一个报价被保存于一个本地的Vector中，它的内容将用于组成适当大小的ProxyQuote[]。最后这个数组被返回。getStockQuote()方法使用的处理过程与Apache2Glue_v2客户应用程序所用的基本相同。惟一不同的是这里没有对返回的错误元素进行处理，我们需要将这些数据排除在作为结果的RemoteQuote实例之外，以便组装出ProxyQuote实例；要记住从后端服务所返回的报价数据的类型与这个新服务将要返回给其客户的报价数据的类型是不一样的。它的确应该是这样的。对本例来说，直接返回RemoteQuote实例当然是最简单的，但在实际的服务中我们不会这样做，而且代理服务不应该以任何形式暴露后台服务的存在。这就是QuoteProxyService的代码：

```
package javasoap.book.ch9.services;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
import org.w3c.dom.*;
public class QuoteProxyService {
    public QuoteProxyService() {
    }
```

```
public ProxyQuote[] getQuotes(String[] symbols)
                        throws Exception {
   Vector v = new Vector();
   int cnt = symbols.length;
   for (int i = 0; i < cnt; i++) {
      v.add(getStockQuote(symbols[i]));
   }
   cnt = v.size();
   ProxyQuote[] quotes = new ProxyQuote[cnt];
   for (int i = 0; i < cnt; i++) {
      quotes[i] = (ProxyQuote)v.elementAt(i);
   }
   return quotes;
}
ProxyQuote getStockQuote(String symbol)
                    throws Exception {
   URL url = new URL("http://mindstrm.com:8004/glue/urn:CorpDataServices");
   SOAPMappingRegistry smr = new SOAPMappingRegistry();
   BeanSerializer beanSer = new BeanSerializer();
   smr.mapTypes(Constants.NS_URI_SOAP_ENC,
     new QName(
       "http://www.themindelectric.com/package/javasoap.book.ch9.services/",
       "Quote"),
       RemoteQuote.class, beanSer, beanSer);

   Call call = new Call();
   call.setSOAPMappingRegistry(smr);
   call.setTargetObjectURI("XYZ");
   call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
   String stock = symbol;
   Vector params = new Vector();
   params.addElement(new Parameter("stock", String.class,
                              stock, null));
   call.setParams(params);

   call.setMethodName("getQuote");
   Response resp = call.invoke(url, "");
   ProxyQuote quote = new ProxyQuote();
   if (resp.generatedFault()) {
      throw new Exception("Service Call Failed");
   }
   else {
      Parameter ret = resp.getReturnValue();
      RemoteQuote value = (RemoteQuote)ret.getValue();
      quote.setStockSymbol(value.get_symbol());
      quote.setLast(value.get_lastPrice());
      quote.setDiff(value.get_change());
```

```
            quote.setTime(value.get_timeStamp());
            quote.setVol(value.get_volume());
        }
        return quote;
    }
}
```

我们可以使用与部署本书中其他所有的 Apache SOAP 服务相同的方式来部署这个服务。我们需要为 ProxyQuote 类包含一个映射。对报价类型的完全限定名是 urn:QuoteProxyService:Quote。我们使用了与JavaBean兼容的属性命名模式，因此映射时使用了 BeanSerializer 类来转换。其部署描述文件应该是下面这样的：

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
             id="urn:QuoteProxyService">
  <isd:provider type="java"
                scope="Application"
                methods="getQuotes">
    <isd:java class="javasoap.book.ch9.services.QuoteProxyService"
    static="false"/>
  </isd:provider>
  <isd:faultListener>org.apache.soap.server.DOMFaultListener
  </isd:faultListener>
  <isd:mappings>
    <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     xmlns:x="urn:QuoteProxyService" qname="x:Quote"
     javaType="javasoap.book.ch9.services.ProxyQuote"
     java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
     xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
  </isd:mappings>
</isd:service>
```

从 GLUE 客户调用 Apache SOAP 服务

这一次，我们将编写一个 GLUE 客户程序来访问我们刚创建的 urn:QuoteProxyService 服务。我们从哪里开始呢？GLUE 要根据 WSDL 文档来动态创建服务绑定。在从 GLUE 到 GLUE 的例子里，GLUE 自动生成了用于描述服务的 WSDL 文档，但 Apache SOAP 不会做这些工作。当然我们也可以手工编写合适的 WSDL 文档……但是这样做的工作量会很大，而且最终的 WSDL 文

档正确的可能性也很小，除非你想要比别人更深地了解WSDL（其实你并不需要了解那么多）。因此，作为替代的方法，我们将使用随GLUE提供的用于生成WSDL的java2wsdl工具（注4）。这个工具并不管你的Java代码是否用于基于GLUE的服务；它只是简单地读取这个Java代码并产生WSDL。java2wsdl不能为基于Apache的服务生成一个完美的WSDL文档，但修改这个文件的工作要比完全用手工编写省事多了。现在让我们来生成WSDL:

```
java2wsdl javasoap.book.ch9.services.QuoteProxyService -n
    urn:QuoteProxyService -t urn:QuoteProxyService -s
    -e http://georgetown:8080/soap/servlet/rpcrouter
```

java2wsdl的第一个参数是实现该服务的Java类。-n选项允许我们指定名称空间；我们在这里使用了服务的名称作为名称空间。-t选项允许我们指定targetNamespace属性的值。这两个选项仅应用于所产生的WSDL文档中的definitions元素。你即将看到，我们还将手工修改其余的部分。-s选项告诉该工具要创建一个SOAP绑定。我们需要这个选项是因为我们是在为一个已知的端点（即服务被部署的位置）建立一个服务描述，因此我们也知道我们所用的传输协议（分开定义服务接口和它的绑定，这不仅是合情合理的，也是常用的方法，这样做可以使得同一个服务可以使用不同的端点和不同的协议来部署）。正因为如此，我们也使用了-e选项，它允许我们指定端点。既然我们的RPC形式的Apache SOAP服务是通过RPC路由器servlet来路由的，其端点就是*http://georgetown:8080/soap/servlet/rpcrouter*（注5）。

既然已经生成了一个WSDL文件，那就让我们完成这些必要的调整好了。下面代码中对修改的部分使用了黑体。

```
<?xml version='1.0' encoding='UTF-8'?>
<definitions
name='urn:QuoteProxyService'
targetNamespace='urn:QuoteProxyService'
```

注4: 还有一些来自不同厂商的其他工具也能完成这样的工作，比如，IBM Web服务工具包中就包含了类似的工具。

注5: 这个位置在我的本地网中。你需要在这个位置使用你自己的合适的网络地址来部署这个服务。

```
xmlns:tns='urn:QuoteProxyService'
xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
xmlns:http='http://schemas.xmlsoap.org/wsdl/http/'
xmlns:mime='http://schemas.xmlsoap.org/wsdl/mime/'
xmlns:xsd='http://www.w3.org/2001/XMLSchema'
xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
xmlns:wsdl='http://schemas.xmlsoap.org/wsdl/'
xmlns='http://schemas.xmlsoap.org/wsdl/'> <types>
    <schema xmlns='http://www.w3.org/2001/XMLSchema'
       xmlns:tns='urn:QuoteProxyService'
       targetNamespace='urn:QuoteProxyService'>
      <complexType name='Quote'>
        <sequence>
          <element name='stockSymbol' nillable='true' type='string'/>
          <element name='last' type='float'/>
          <element name='diff' type='float'/>
          <element name='time' nillable='true' type='string'/>
          <element name='vol' type='long'/>
        </sequence>
      </complexType>
      <complexType name='ArrayOfQuote'>
        <complexContent>
          <restriction base='soapenc:Array'>
            <attribute ref='soapenc:arrayType'
                 wsdl:arrayType='tns:Quote[]'/>
          </restriction>
        </complexContent>
      </complexType>
    </schema>
    <schema xmlns='http://www.w3.org/2001/XMLSchema'
       xmlns:tns='urn:QuoteProxyService'
       targetNamespace='urn:QuoteProxyService'>
      <complexType name='ArrayOfstring'>
        <complexContent>
          <restriction base='soapenc:Array'>
            <attribute ref='soapenc:arrayType' wsdl:arrayType='string[]'/>
          </restriction>
        </complexContent>
      </complexType>
    </schema>
  </types>
  <message name='getQuotes0SoapIn'>
    <part name='arg0'
       xmlns:ns1='urn:QuoteProxyService' type='ns1:ArrayOfstring'/>
  </message>
  <message name='getQuotes0SoapOut'>
    <part name='Result'
       xmlns:ns1='urn:QuoteProxyService' type='ns1:ArrayOfQuote'/>
```

```
</message>
<portType name='QuoteProxyServiceSoap'>
  <operation name='getQuotes' parameterOrder='arg0'>
    <input name='getQuotes0SoapIn' message='tns:getQuotes0SoapIn'/>
    <output name='getQuotes0SoapOut' message='tns:getQuotes0SoapOut'/>
  </operation>
</portType>
<binding name='QuoteProxyServiceSoap' type='tns:QuoteProxyServiceSoap'>
  <soap:binding style='rpc'
      transport='http://schemas.xmlsoap.org/soap/http'/>
  <operation name='getQuotes'>
    <soap:operation soapAction='getQuotes' style='rpc'/>
    <input name='getQuotes0SoapIn'>
      <soap:body use='encoded' namespace='urn:QuoteProxyService'
           encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'/>
    </input>
    <output name='getQuotes0SoapOut'>
      <soap:body use='encoded' namespace='urn:QuoteProxyService'
           encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'/>
    </output>
  </operation>
</binding>
<service name='QuoteProxyService'>
  <port name='QuoteProxyServiceSoap'
          binding='tns:QuoteProxyServiceSoap'>
    <soap:address
          location='http://georgetown:8080/soap/servlet/rpcrouter'/>
  </port>
</service>
</definitions>
```

我们需要修改除definitions元素之外的所有元素的名称空间和目标名称空间值；definitions元素中的是正确的，因为我们为java2wsdl提供了-n和-t命令行参数。这就是上面所有的namespace和targetNamespace属性值都被改动过的原因。生成的值中还包含了基于服务实现所用的Java类的名称。它们只是名称，但我仍然不想暴露这种信息。因此在文档的其他部分中，我删除了这些Java类名称前面的包名，举例来说，原来在binding元素的name属性值的前面部分就是我们的服务类所在的包名。修改name属性对互操作性来说不是必要的，但我认为这样做至少可以消除一种假象，它说明暴露Java类和包的名称并不是必需的。

对于自定义报价类型的定义也需要做一些更改。为这个类型生成的名称是ProxyQuote，它来自于在服务器端实现自定义类型的Java类名。然而，它与我们

在部署描述文件中的映射段中所用的类型名不一致。我们来看看部署描述文件中的相关部分:

```
<isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:x="urn:QuoteProxyService" qname="x:Quote"
javaType="javasoap.book.ch9.services.ProxyQuote"
java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
```

这里的名称空间是 urn:QuoteProxyService。我们已经修改了 WSDL 中的对应部分，因此现在看起来这个部分就完成了。但是，再看看 qname 属性，我们赋给它的值是 Quote（注 6），而不是 ProxyQuote。这意味着在编码响应封套时，报价数据的类型将被设为 Quote，并用 urn:QuoteProxy-Service 作为名称空间限定。java2wsdl 生成器没办法知道这些，因为这些信息存在于 Java 代码之外。

最后，我们需要更改组成自定义报价类型的数据元素的名称。java2wsdl 工具程序没有使用默认的JavaBeans属性存取器，相反它直接在实现类中查找成员变量。因此它所使用的名称是 sym、_last，等等，但 BeanSerializer 类在编码 ProxyQuote 类的实例时会使用基于 JavaBeans 命名标准的属性名。是的，绝大部分都是这样。它们实际上不会像你所期望的那样被编码。如果你看看从服务返回的 SOAP 封套，就会看到自定义报价数据类型的元素的各属性的第一个字符全都是小写字母。而根据 Java Beans 规范，读写存取器 setThing()和getThing()代表属性名是 Thing。但 BeanSerializer 会将它们编码为 thing。这个问题对 Apache 到 Apache 的例子是没有影响的，但是对 GLUE 客户就不同了。于是我们就遇到了一些由于两种SOAP实现的古怪行为而导致的小问题。为了解决这些问题，我们将这些元素的名称改成了 stockSymbol、last、diff、time 和 vol。

文档中对getQuotes服务方法所返回的报价数组也有定义。name属性的值被改成了 ArrayOfQuote，而 wsdl:arrayType 的值被改成了 tns:Quote[]。实际上我们只是在报价及相应的报价数组定义中去掉了“Proxy”而已。第二个 message 定义引用了报价数组，因此它也应该被修改来反映报价数组类型的正确名称。

注 6: 这个值实际上是 x:Quote，但为方便查询起见，我仅仅使用了这个名称的本地部分而不是完全限定名。

现在我们要来发布这个 WSDL 文档了。我的 Apache SOAP 引擎是建立在一个 Apache Tomcat服务器上的。这个服务器监听了8080端口，它在我的本地网中一个叫 georgetown的机器上运行。在我的 Apache SOAP 安装目录下的 *webapps/soap* 目录是一个用来存放 WSDL 文件的非常好的地方。这意味着我可以使用 *http://georgetown:8080/soap/QuoteProxyService.wsdl* 这个 URL 来访问到它。

现在，我们要准备写一个 GLUE 客户程序来访问 urn:QuoteProxyService 服务了。这个应用程序的所有类都应该在 javasoap.book.ch9.clients 包中。我们不打算使用wsdl2java工具程序来建立用于绑定到服务的类和接口，因为这样做我们无法控制所产生的类的名称。我们需要能够控制这些类名，是因为我们将要把所有的客户端类都放到 javasoap.book.ch9.clients 包中，而且我们已经有了一个 Quote 类；我们希望将对应于报价数据类型的本地 Java 类命名为 ProxyQuote。你可以在对应于javasoap.book.ch9.clients包的目录以外的某个目录下运行 wsdl2java；并将生成的 Quote 类及其文件名改为 ProxyQuote；修改 *QuoteProxyService.map*文件中对它的引用；再把这些文件移到我们这个包的目录下。在这里，我们仅用手工来编写这些类。这并不困难，而且它给了我们一个机会来深入了解编写 GLUE 客户应用程序的细节。

那么，我们现在就来建立一个 ProxyQuote 类来表示从服务返回的报价数据。这个类需要对报价中的每一个元素建立一个公共的数据字段，我们还添加了一个 toString()方法来使显示它更为容易。这里是 ProxyQuote 类的代码：

```
package javasoap.book.ch9.clients;
public class ProxyQuote {
   public String stockSymbol;
   public float last;
   public float diff;
   public String time;
   public long vol;
   public String toString() {
      String result = "Symbol:      " + stockSymbol +
                      "\nLast Price: " + last +
                      "\nChange:     " + diff +
                      "\nTime Stamp: " + time +
                      "\nVolume:     " + vol;
      return result;
   }
}
```

我们还需要一个映射文件来描述对从服务返回的ProxyQuote类的数据的映射。这个文件叫做 *QuoteProxyService.map*。我把它放在与 javasoap.book.ch9.clients 类相同的目录下，以便它能被自动加载。下面就是这个映射文件:

```
<?xml version='1.0' encoding='UTF-8'?>
<mappings xmlns='http://www.themindelectric.com/schema/'>
  <schema xmlns='http://www.w3.org/2001/XMLSchema'
      targetNamespace='urn:QuoteProxyService'
      xmlns:electric='http://www.themindelectric.com/schema/'>
    <complexType
          name='Quote'
          electric:class='javasoap.book.ch9.clients.ProxyQuote'>
        <sequence>
          <element name='stockSymbol' nillable='true'
                   electric:field='stockSymbol' type='string'/>
          <element name='vol'
                   electric:field='vol' type='long'/>
          <element name='last'
                   electric:field='last' type='float'/>
          <element name='diff'
                   electric:field='diff' type='float'/>
          <element name='time' nillable='true'
                   electric:field='time' type='string'/>
        </sequence>
      </complexType>
    </schema>
  </mappings>
```

要注意我们为 complexType 元素的 name 属性所赋的值是 Quote。这与我们在 Apache 服务的部署描述文件中所用的一致，而且也与我们对 WSDL 文档的修改一致。所有 element 标签中的 name 属性都与从服务器返回的报价数据中的元素名称相对应，而且其属性值也与映射的 ProxyQuote 类中的成员变量名一致。对整个文档都起作用的targetNamespace属性的值为urn:QuoteProxyService。这个值将被服务用于限定返回Quote类型的名称空间；这也和WSDL文档中的内容相一致。

这里是一个我们用来绑定到服务的Java接口。它包含了一个也是惟一的服务方法 getQuotes()。

```
package javasoap.book.ch9.clients;
public interface IQuoteProxyService {
```

```
    ProxyQuote[] getQuotes(String[] symbols);
  }
```

现在我们已经具备了编写客户应用程序所需要的所有条件。我们使用命令行参数来指定我们要报价的股票代码。我们将使用WSDL所发布的地址来绑定到服务，使用 args 作为参数调用 getQuotes()方法，并遍历返回的数组，最后显示其结果。这里是它的代码:

```
package javasoap.book.ch9.clients;
import electric.registry.Registry;
public class Glue2Apache {
   public static void main( String[] args )
            throws Exception {
      IQuoteProxyService service = (IQuoteProxyService)Registry.bind(
            "http://georgetown:8080/soap/QuoteProxyService.wsdl",
            IQuoteProxyService.class);
      ProxyQuote[] quotes = service.getQuotes(args);
      int cnt = quotes.length;
      for (int i = 0; i < cnt; i++) {
         System.out.println(quotes[i]);
         System.out.println("");
      }
   }
}
```

我用了四个股票代码来运行这个例子，这是它的命令行:

```
java javasoap.book.ch9.clients.Glue2Apache IBM MSFT GM MMM
```

下面是我所得到的输出结果:

```
Symbol:      IBM
Last Price:  123.1
Change:      -0.79
Time Stamp:  1:59PM
Volume:      3916800
Symbol:      MSFT
Last Price:  67.701
Change:      -1.789
Time Stamp:  2:04PM
Volume:      21485800
Symbol:      GM
Last Price:  47.36
Change:      -0.74
```

```
Time Stamp: 1:59PM
Volume:     1719700
Symbol:     MMM
Last Price: 118.51
Change:     -1.29
Time Stamp: 2:00PM
Volume:     953400
```

访问 .NET 服务

尽管微软的 .NET 技术并不是基于 Java 的，但是我们还是不能不提到它。如果你要与Web服务进行交互，你肯定会不止一次地遇到.NET。为了说明如何实现.NET与用Java编写的服务之间的互操作，我用C#语言编写了一个.NET Web服务，并将它发布在*http://mindstrm.com:8199/CorpDataService/Proxy.asmx*（注7）。在使用 .NET 时，需要指明你的服务是 RPC 形式的，因为这不是 .NET 的默认服务形式。为了达到这个目的，要在你的C#代码中使用`[SoapRpcService()]`声明。用来代理 `urn:CorpDataServices` 服务的 .NET 服务运行在 *http://mindstrm.com:8004/glue/urn:CorpDataServices*。图 9-3 描述了各个部分之间的关系。

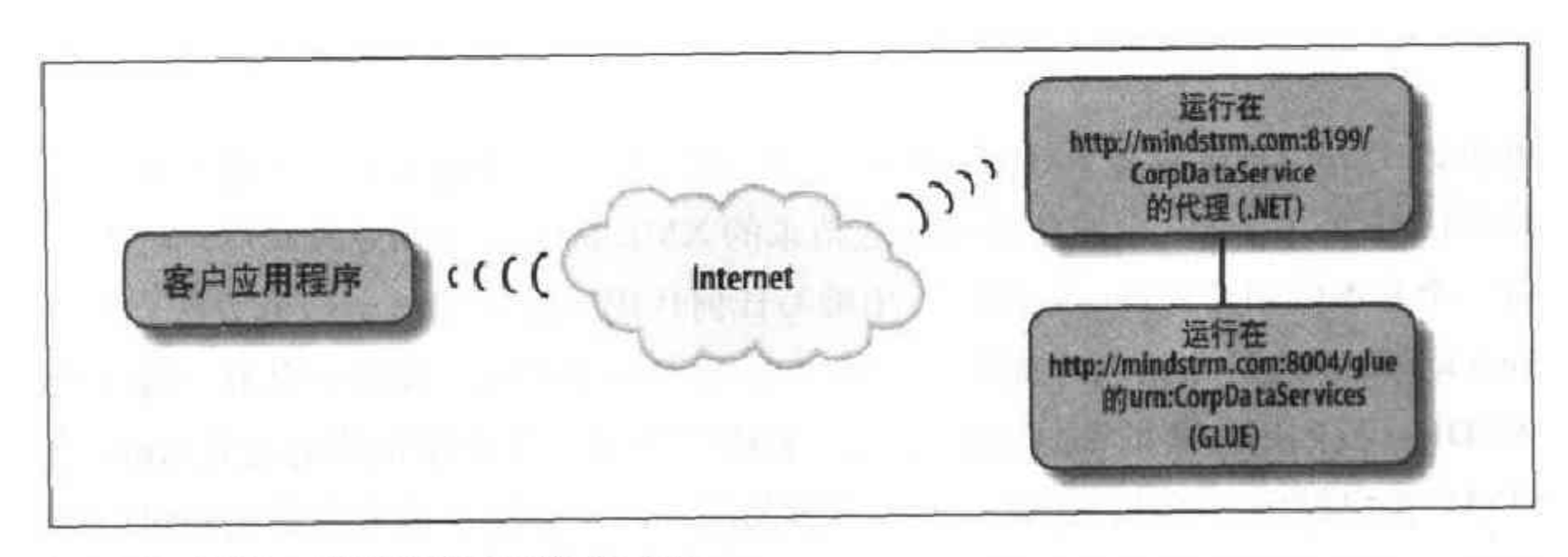

图 9-3：客户、代理和服务之间的交互

`Proxy` 服务有一个 `getHeadlines` 方法和一个 `getQuote` 方法，这与 `urn:CorpDataServices`服务一样。每个方法都使用了单一的字符串参数来表示股票代码。`getHeadlines`返回字符串数组，而`getQuote`返回一个自定义报价数据类型。你可以简单地通过访问它的URL来得到这个.NET服务的有关信息。因此在你的浏

注 7：　这个服务的代码位于 *http://www.mindstrm.com/soap.htm*。

览器中输入网址 *http://mindstrm.com:8199/ CorpDataService/Proxy.asmx*，就会得到如图 9-4 所显示的页面。

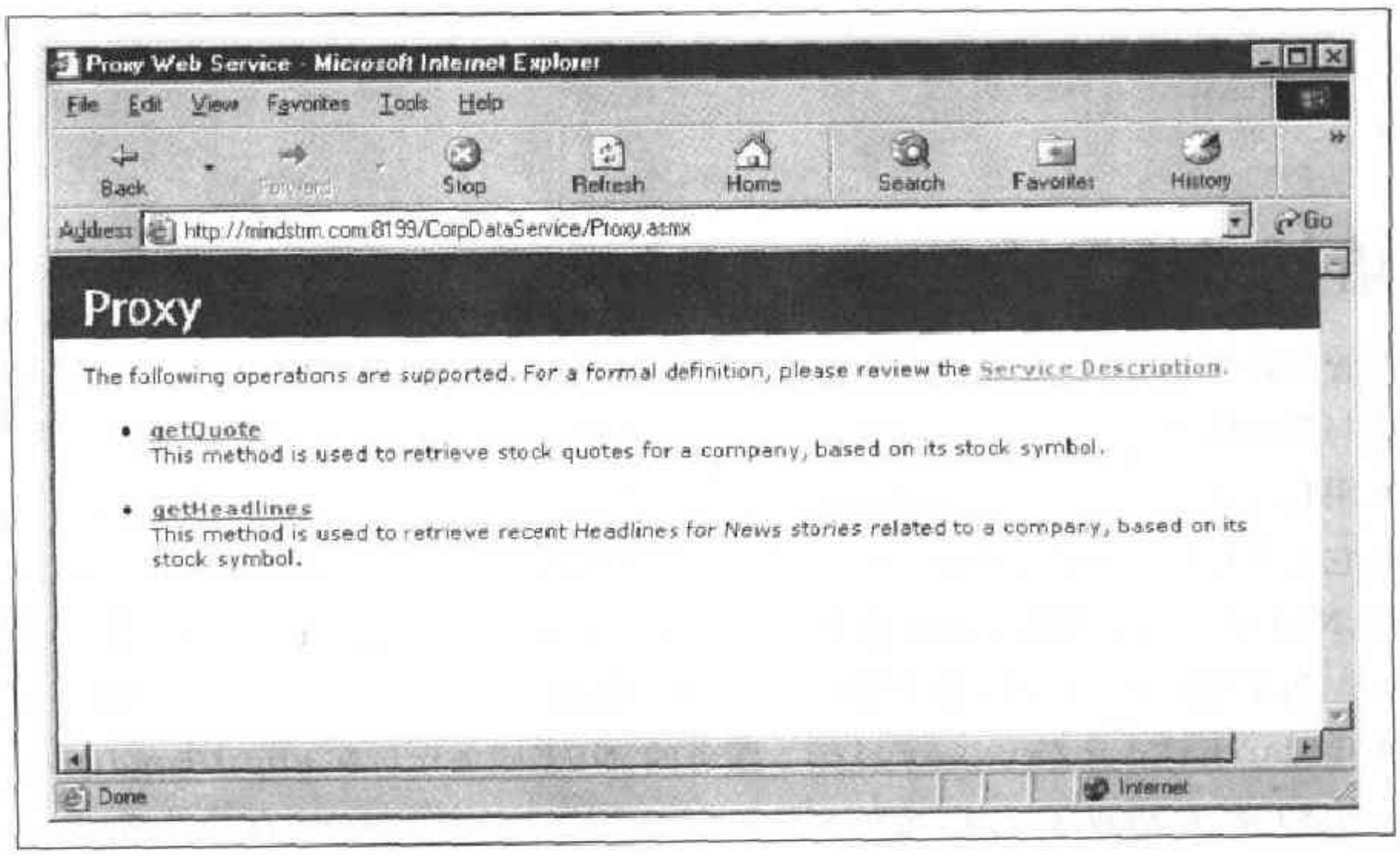

图 9-4: 一个 .NET 服务的有关信息

单击 getQuote 链接或者 getHeadlines 链接，就会链接一个新页面，在这个页面中你可以看到一些向这个服务方法发送请求的 XML 例子以及相应的响应。这里也有一个基本的用户界面，你可以不用编写任何代码就能调用这个服务方法。单击 Service Description 会为你提供这个 Proxy 服务的 WSDL。我们可以看一遍这个 WSDL，找到所需要的 SOAPAction、名称空间等，以便能正确地建立 Apache SOAP 客户程序。不过，还有一个比较容易的办法。既然单击方法的名称可以带你到一个显示在服务方法与客户之间传递的 XML 消息的页面，我们就可以让浏览器来指出需要发送什么样的消息。图 9-5 显示了该页的一部分，其中显示了调用 getQuote 时需要发送什么样的请求封套。

它差不多给了我们要访问该服务所需要的全部信息，除了它的地址以外。虽然 HTTP 头中的 Host 字段中说明了服务器名是 mindstrm.com，但它没有显示访问服务所用的端口。为了得到正确的地址，我们需要查看 WSDL，这里是这个服务的 WSDL 文档中的 service 段：

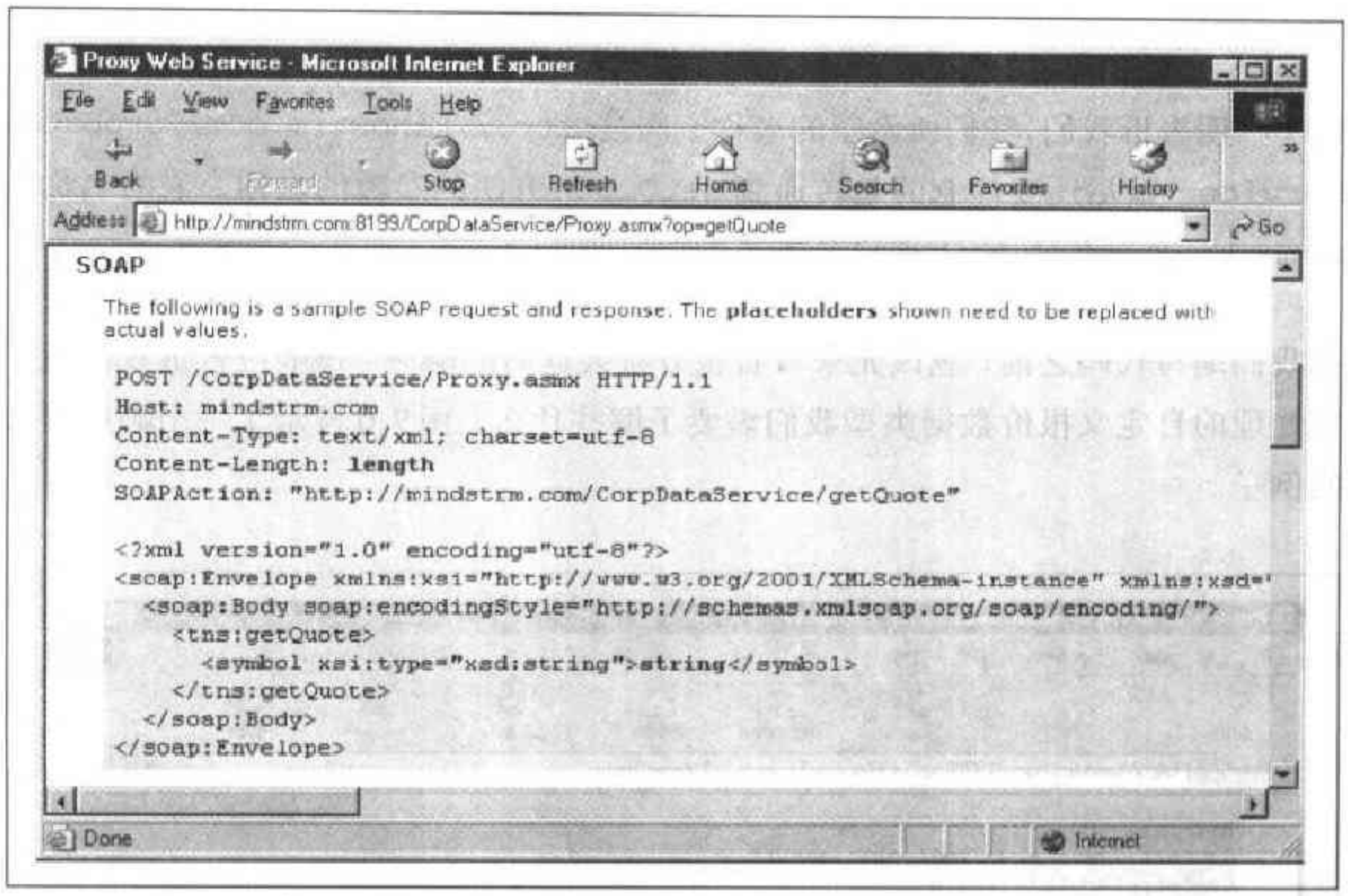

图 9-5: 发送到 getQuote 的 SOAP 文档

```
<service name="Proxy">
  <port name="ProxySoap" binding="s2:ProxySoap">
  <soap:address
    location="http://mindstrm.com:8199/CorpDataService/Proxy.asmx" />
  </port>
```

我们现在已经看到了这个 URL 是 *http://mindstrm.com:8199/CorpDataService/Proxy.asmx*。接下来我们需要 SOAPAction 头字段的值。是的，在这个例子中它很重要！如果你在这里做的不对，.NET就不会调用你的服务。因此我们需要在应用程序中使用值 http://mindstrm.com/CorpDataService/getQuote 来指定 SOAPAction。记住，这并不是一个用来定位服务或服务方法位置的地址，因此它不包含像端口号这样的细节。它是一个 URI，一个用来向 .NET 框架说明后面的 SOAP 封套与什么有关的名称。

现在我们需要确定用于限定 getQuote 元素的名称空间。getQuote 是用名称空间标识符 tns 来限定的（在图 9-5 中没有显示出来，但是如果将窗口向右滚，你就会看到它）。这里是该名称空间的声明:

```
xmlns:tns=http://mindstrm.com/CorpDataService
```

这个声明告诉我们，我们所希望的名称空间是http://mindstrm.com/CorpDataService。最后还要注意的是传递到getQuote方法的参数的名称。它的名称是symbol，这一点对.NET来说是重要的。

在我们编写代码之前，应该先来看看服务将会返回的响应。该响应告诉我们对于要处理的自定义报价数据类型我们需要了解些什么。图9-6显示了一个响应封套的例子。

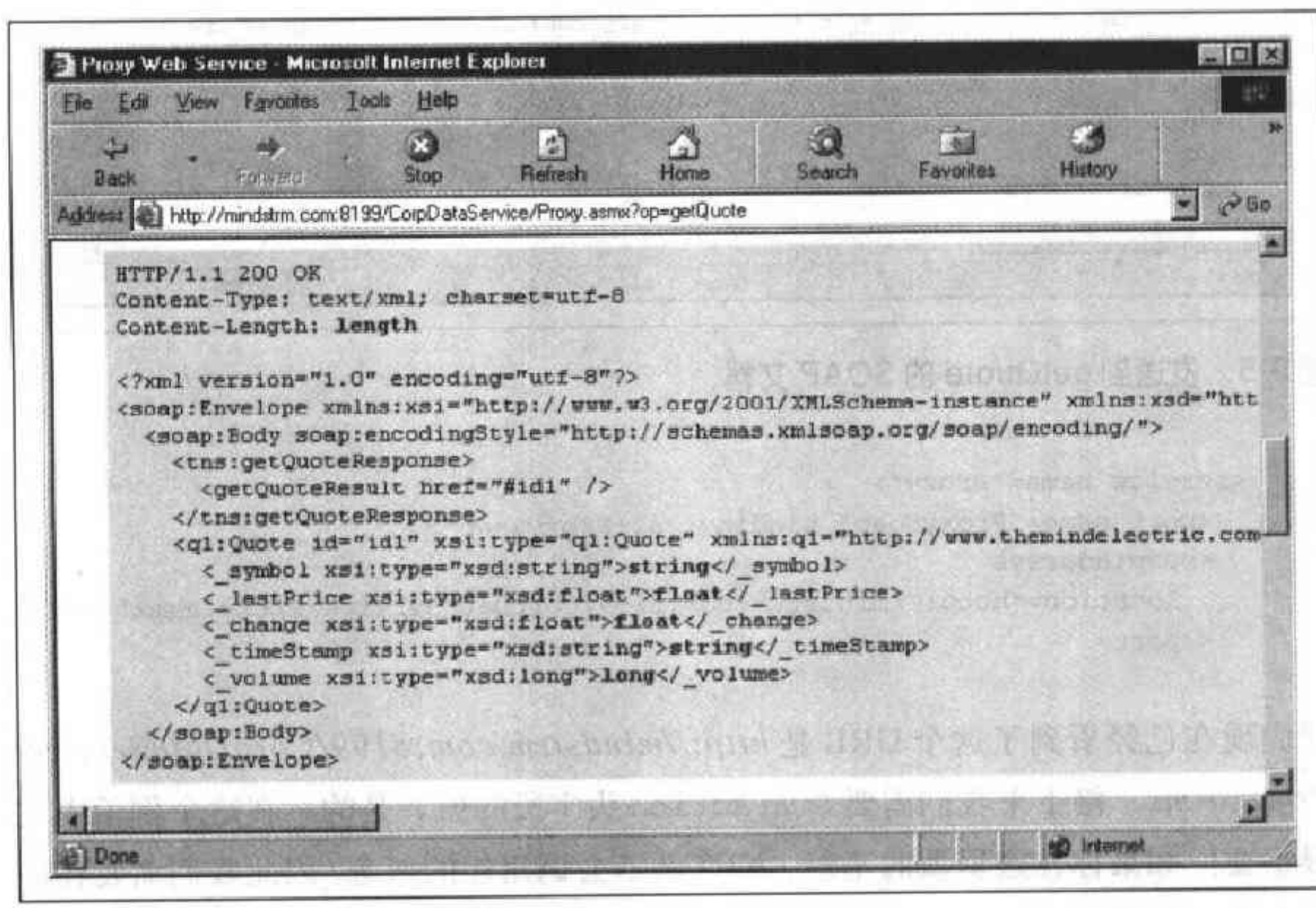

图9-6：对getQuote调用的响应

getQuoteResponse由tns限定，它的值与请求封套一样，也是http://mindstrm.com/CorpDataService(该名称空间在页面中也被遮住了)。还要注意Quote元素也使用了名称空间限定，它所用的是标识符q1。同样，它也被遮住了一部分。为此我们在这里将它的实际内容写出来：

```
xmlns:q1="http://www.themindelectric.com/package/javasoap.book.ch9.services/"
```

这个XML中的www.themindelectric.com是用来干什么的呢？在开发这个.NET服务时，我使用了.NET Visual Studio来生成了一个C#类，让它访问在GLUE中实现的urn:CorpDataServices服务。它通过请求WSDL来完成这个工作，这个WSDL是由基于GLUE的服务器自动生成的。而当该服务器产生名称空间的时候，它使用了www.themindelectric.com和Java包的名称。记住，它只是一个名称而已。在将这个类型映射到一个Java类的时候，知道这个名称会为我们带来方便。

这就是访问该.NET服务的Apache SOAP客户应用程序的代码：

```
package javasoap.book.ch9.clients;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
import org.w3c.dom.*;
public class Apache2DotNet {
   public static void main(String[] args)
      throws Exception {

      URL url =
           new URL("http://mindstrm.com:8199/CorpDataService/Proxy.asmx");
      SOAPMappingRegistry smr = new SOAPMappingRegistry();
      BeanSerializer beanSer = new BeanSerializer();
      smr.mapTypes(Constants.NS_URI_SOAP_ENC, new QName(
        "http://www.themindelectric.com/package/javasoap.book.ch9.services/",
        "Quote"),
         Quote.class, beanSer, beanSer);

      Call call = new Call();
      call.setSOAPMappingRegistry(smr);
      call.setTargetObjectURI("http://mindstrm.com/CorpDataService");
      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      String stock = args[0];
      Vector params = new Vector();
      params.addElement(new Parameter("symbol",
                                 String.class, stock, null));
      call.setParams(params);

      try {
         call.setMethodName("getQuote");
```

```
            Response resp = call.invoke(url,
                  "http://mindstrm.com/CorpDataService/getQuote");
            if (resp.generatedFault()) {
               Fault fault = resp.getFault();
               String code = fault.getFaultCode();
               String desc = fault.getFaultString();
               System.out.println(code + ": " + desc);
               Vector v = fault.getDetailEntries();
               int cnt = v.size();
               for (int i = 0; i < cnt; i++) {
                  Element n = (Element)v.elementAt(i);
                  Node nd = n.getFirstChild();
                  System.out.println(n.getNodeName() + ": " +
                                                 nd.getNodeValue());
               }
            }
            else {
               Parameter ret = resp.getReturnValue();
               Quote value = (Quote)ret.getValue();
               System.out.println(value);
            }
         }
         catch (SOAPException e) {
            System.err.println("Caught SOAPException (" +
                           e.getFaultCode() + "): " +
                           e.getMessage());
         }
      }
   }
```

Java类Apache2DotNet与我们前面开发的Apache2Glue_v2类差不多，因此我们将只介绍变化的部分，而不是重新解释整个类。有变化的部分使用了黑体字，以便你能迅速地找到它们。

访问这个服务时要用到的URL是*http://mindstrm.com:8199/CorpDataService/Proxy.asmx*。既然我们将要调用getQuote，我们就会得到一个自定义报价数据类型的实例。在图9-6中你可能已经注意到了，报价类型数据字段与urn:CorpDataServices所返回的Quote类型的数据字段是完全一致的。因此，我们在这里就可以通过映射Quote类来重用一部分代码，Quote类是我们在为访问GLUE服务而编写Apache SOAP客户程序时开发的。它说明我们将要用来限定报价类的名称空间也是一样的：http://www.themindelectric.com/package/javasoap.book.ch9.services。

接下来改动的是传递给 call.setTargetObjectURI()方法的参数。这一次我们传递的是 http://mindstrm.com/CorpDataService，它是 WSDL 中用于限定 getQuote 元素的名称空间。我们还需要将 Parameter()构造函数中指定的参数名称改为 symbol，因为这是我们在前面查看请求封套时找到的名称。

最后一个改变是我们需要指定与这个方法调用有关的 SOAPAction 值。我们使用了 call.invoke()方法的第二个参数来设定这个值。在其他的 Apache SOAP 客户程序中，我们曾将一个空字符串作为第二个参数，那是因为没必要指定一个 SOAPAction。但是，当调用.NET 服务方法时，一定要包含正确的 SOAPAction。在前面的部分，我们已经知道了 SOAPAction 的值应为 http://mindstrm.com/CorpDataService/getQuote，因此我们就将它传递给 call.invoke()方法。

代码的其他部分没有变化。如果运行 Apache2DotNet 应用程序，它将调用我们的.NET 服务的 getQuote 方法，这个方法会依次调用 urn: CorpDataServices 服务的 getQuote 方法。当我以 GM 为股票代码运行这个应用程序时，其输出看起来是这样的：

```
Symbol:      GM
Last Price:  48.05
Change:      0.35
Time Stamp:  11:06AM
Volume:      1078700
```

编写 Apache Axis 客户

Apache 小组现在正在开发一种叫做 Axis 的全新的 SOAP 实现（注 8）。现在就在 Axis 上花费大量时间似乎还太早了，因为这个项目还处于开发阶段。在它稳定下来之前还会有很多变化，而且现在它还没有为产品应用做好准备。Axis 计划遵循 JAX-RPC，这是针对 SOAP RPC 的一个刚出现的 API 标准。然而，Axis 现在毕

注 8：它的实际名称现在还在讨论之中，但到目前为止它还是叫做 Axis。本书中的这些例子基于 2001 年 11 月 5 日发布的 Axis 版本。显然，随着开发的进行，这个 API 还将不断变化。

竟已经出现了，我们可以尝试着使用它。因此，让我们来简单地看看Axis，体验一下它是如何工作的。在后面的第十一章里，我们将介绍JAX-RPC。

客户的核心是Axis2DotNet。它调用了我们前面使用的那个.NET服务的get-Headlines方法。

```
package javasoap.book.ch9.clients;
import org.apache.axis.AxisFault;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.client.Transport;
import org.apache.axis.encoding.XMLType;
import org.apache.axis.transport.http.HTTPConstants;
import org.apache.axis.utils.Options;
import java.net.URL;
import java.util.*;
public class Axis2DotNet {
   public static void main(String args[]) {
      try {
         URL url =
          new URL("http://mindstrm.com:8199/CorpDataService/Proxy.asmx");
         Service  service = new Service();
         Call call = (Call) service.createCall();
         call.setTargetEndpointAddress( url );
         call.setOperationName("getHeadlines");
         call.setProperty(Call.NAMESPACE,
                  "http://mindstrm.com/CorpDataService");
         call.setProperty(HTTPConstants.MC_HTTP_SOAPACTION,
              "http://mindstrm.com/CorpDataService/getHeadlines");
         call.addParameter("symbol", XMLType.XSD_STRING, Call.PARAM_MODE_IN);
         call.setReturnType(XMLType.SOAP_ARRAY);
         ArrayList ret = (ArrayList)call.invoke(new Object[] {args[0]});
         Object[] res = ret.toArray();
         int cnt = res.length;
         for (int i = 0; i < cnt; i++) {
            System.out.println((String)res[i]);
         }
      }
      catch (Exception e) {
         System.out.println(e);
      }
   }
}
```

Axis 看起来是在向一个更为结构化的 API 的方向发展，这相对于 Apache SOAP 实现来说是一个受欢迎的变化。这并不是说 Apache SOAP 不好；事实上它是一个很不错的工具，而且能够很好地工作。在Axis中出现的这些更为巧妙的面向对象的 API 当然是在 Apache SOAP 的经验上开发的。

现在还是让我们来看看代码吧。我们首先创建了 java.net.URL的一个实例，这与我们在 Apache SOAP 的例子中所做的一样。下一步就是要建立 org.apache.axis.client.Service的一个实例，它用作创建org.apache.axis.client.Call对象的工厂，并通过调用service.createCall()方法建立了该对象的一个实例。从这里开始，所有的工作都通过 Call 对象来完成。我们把 URL 传递给 call.setTargetEndpointAddress()，并用call.setOperationName()方法设定了方法的名称。接下来就要设置名称空间。在前一个例子中，我们已经确定了要用的名称空间是 http://mindstrm.com/CorpDataService。这个名称空间被 call.setProperty()方法设成了 Call 对象的一个属性。调用时，该方法的第一个参数通过 Call.NAMESPACE传入，第二个参数用适当的名称空间作为字符串传入。我们还用同样的方法来设置 SOAPAction属性，这一次使用常量 Call.MC_HTTP_SOAPACTION作为属性名，用 http://mindstrm.com/CorpDataService/getHeadlines 作为其值。记住，.NET 服务要求指定一个与服务方法对应的 SOAPAction 值。

下一步是用call.addParameter.()方法建立Call的参数。该方法的第一个参数是要传给服务的参数名称。这个名称在SOAP封套中被用作一个元素名。第二个参数指定它的数据类型；在这里用的是常量 XMLType.XSD_STRING，说明该参数是一个内置的字符串类型的实例。最后一个参数Call.PARAM_MODE_IN说明该参数是一个 in 参数（Axis 也支持 out 和 in/out 参数）。有趣的是参数的值并没有在 call.addParameter()方法中指定。相反，我们将一个 Object[]传递到随后的 call.invoke()方法，Object[]中的每一项对应于一个已用 call.addParameter()建立的参数值。我们这里只有一个参数：args[0]中的字符串，它包含了股票代码。因此我们用 args[0]来建立有且仅有一个子元素项的 Object[]实例。

call.invoke()方法的返回值是 java.util.ArrayList 的一个实例，它包含了与传递给服务方法的股票代码有关的新闻标题。调用返回的 ArrayList 的 toArray()方法将返回一个Object[]实例。该实例中的每一项都是一个包含了新闻标题的 java.lang.String。我们遍历这个数组并将新闻标题写到控制台上。我在命令行以 GM 为参数运行 Axis2DotNet。这里是我所得到的结果:

```
Autos: Year of Big Sales, Little Profits
Autos: Year of Zero Financing and Profits
Report: EchoStar May Invest in Spaceway
EchoStar may invest $1 billion in Spaceway-WSJ
GM-Daewoo Motor deal due by Jan 20 - creditor bank
In Detroit, a New Definition of 'Quality'
GM's bright idea; Ford's e-commerce plans take root
GM invests in Lutz's Cunningham Motors
December U.S. Auto Sales Seen Up Over '00
RESEARCH ALERT-UBS Warburg ups GM EPS view
```

看起来编写一个 .NET 服务的 Axis 客户相当容易。Axis 也能实时生成和使用 WSDL，不过我们不打算在这里介绍它。

让我们再来修改这个代码，使它可以访问运行于*http://mindstrm.com:8004/glued*的基于GLUE的服务 urn:CorpDataServices。新的类名是 Axis2Glue。我们只需要在 Axis2DotNet 类的基础上做一处改动就可以让它访问 GLUE 服务了：将 URL 改为*http://tokyo:8004/glue/urn:CorpDataServices*。其他的代码都可以保持不变。这样就行了！很显然，Axis 工作组正在努力规范化它的 API，当它成熟的时候，将会是一个非常值得关注的实现。

第十章

SOAP Header

在SOAP封套中可以包含一个Header元素，该元素中能够包含在RPC（或其他）形式调用的边界之外的数据。头是一个扩展机制，它可以携带那些存在于体消息语义之外但是又有用或者对于正确处理消息有必要的任何类型的信息。路由信息中可能包含事务标识符、鉴别密钥或者其他与消息处理及路由有关的信息。

当一个消息要被发送到中间媒介时，把路由信息放在SOAP头里非常有用，中间媒介会依次将该消息发送到目的地。在第九章中我们开发了一些代理服务，它们使用其他的服务来完成自己的任务。在那些情况下，客户应用程序并不知道它所调用的服务实际上只是一个代理。然而，代理服务也可以仅将消息转发到一个指定的端点或者使用其他的端点来完成它们的任务。我们将要开发一个这种代理服务、或称中介服务的例子。

尽管SOAP目前还没有指定任何安全机制，但Header元素仍不失为编码如用户名和密码这样的鉴别数据的一个好地方。用Header来存储鉴别数据，使你能够让服务方法只接受必要的、与完成任务有关的参数，同时也为你提供了一种向接收方传递其他相关数据的机制。将SOAP头用于鉴别的这种语义是与实现相关的，但这种机制却在某些其他的标准中占有一席之地。

GLUE 为 SOAP 头提供了相当好的支持（注 1），但 Apache SOAP 在这方面就欠缺得多了。我将为你找到一种在 Apache SOAP 的客户和服务中使用头的技术。

Apache SOAP 的提供者和路由器

Apache SOAP引擎没有为使用SOAP头提供直接的支持。不过，它提供了足够的 Java 类和功能丰富的钩（hook），这使得你很容易使用头。相关的工作大部分都在服务端完成。基本上，我们希望有一种方法能使我们在所实现的服务方法内部访问 SOAP 头。不过在深入到代码中之前，我们先来看看 Apache SOAP 处理和路由 SOAP 消息的方式。图 10-1 显示了一个 RPC 消息发送到一个 Apache SOAP 服务所经过的路径。灰色的方块代表被显示的对象是 Apache SOAP 框架的一部分，白色的则代表我们在本书中编写的那些 Java 类。

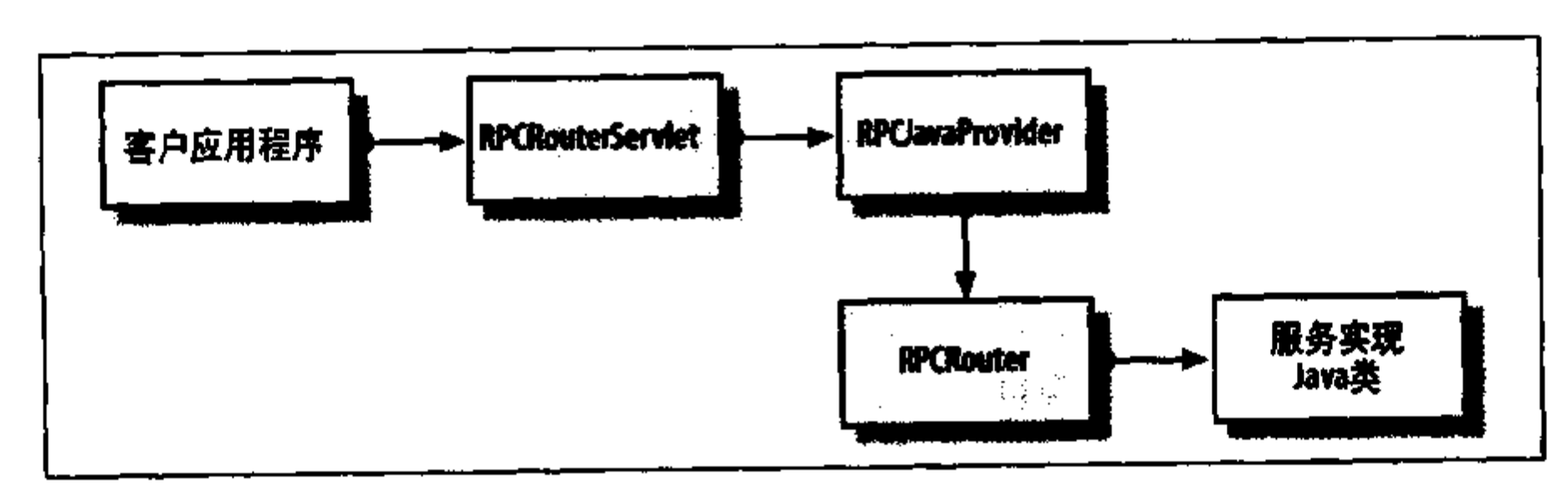

图 10-1：从客户到服务实现的路径

Apache SOAP 是基于 Java servlet 的。当发送一条 RPC 形式的 SOAP 消息时，实际上是将这条消息发送给了 `org.apache.soap.server.http.RPCRouterServlet` 类。由 servlet 为特定的调用确定适当的提供者类。到目前为止我们所有的 Apache SOAP 服务都使用了默认的 Java 提供者。下面是部署描述文件中的有关部分：

```
<isd: provider  type="java"
```

在每一个例子里，我们都设置 type 属性的值为 "java"。这仅仅是一个别名，它告知框架我们想要使用默认的 Java 提供者类 `org.apache.soap.providers.`

注 1：　如果你想要在 GLUE 中使用头，GLUE 为此提供了非常全面的文档。

RPCJavaProvider。前面那个servlet使用这个提供者类来定位和调用服务方法。默认的提供者使用org.apache.soap.server.RPCRouter来将消息路由到适当的服务实现类。路由器加载服务类并用Java反射来调用指定的方法。如果要指定某个其他的提供者，只要将该数据提供者的Java类的全名作为type属性的值即可。

替换提供者和路由器类

为了能在我们的服务方法中使用 SOAP 头，我们需要用自己设计的路由器来替换默认的那个。我们完全可以修改org.apache.soap.server.RPCRouter类并重建Apache SOAP源码树，但那样做会显得有些混乱，因为对于每个Apache SOAP引擎的新版本我们都将不得不再这样做一次。幸运的是，Apache SOAP引入了可插入式提供者（pluggable provider）的概念。这意味着你可以插入自己的提供者，以便在部署服务时使用它。要使用你自己的提供者，必须先实现你自己的提供者类。这正是我们正在寻找的钩。创建我们自己的提供者，让它使用我们自己的可以将 SOAP 头传递到我们的服务方法的路由器。比方说，如果实现了一个叫做BetterRPCJavaProvider的提供者类，我们就能通过在相应的部署描述文件中修改 isd:provider 元素来将它关联到一个服务的部署：

```
<isd:provider type="javasoap.book.ch10.services.BetterRPCJavaProvider"
```

这个元素告诉RPC路由器servlet使用我们的新提供者类而不是默认的那个。而这正是我们想要的。现在，让我们来编写自己的提供者吧。

所有的提供者都必须实现org.apache.soap.util.Provider接口，该接口包含了 locate()和invoke()方法。locate()方法定位服务部署信息，为方法调用做准备。invoke()方法调用服务方法，它假定前面对 locate()的调用是成功的。

我们在这里不必重头开始。org.apache.soap.providers.RPCJavaProvider类在定位服务方面的工作做得很完美，因此我们只需要扩展它，根据我们的要求修改 invoke()方法。这里是新提供者类 BetterRPCJavaProvider 的代码：

```
package javasoap.book.ch10.services;
```

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.server.*;
import org.apache.soap.server.http.*;
import org.apache.soap.util.*;

public class BetterRPCJavaProvider
        extends org.apache.soap.providers.RPCJavaProvider {

   public void invoke(SOAPContext reqContext, SOAPContext resContext)
            throws SOAPException {
      try {
         Response resp = BetterRPCRouter.invoke(envelope.getHeader(), dd,
               call, targetObject, reqContext, resContext);
         Envelope env = resp.buildEnvelope();
         StringWriter sw = new StringWriter();
         env.marshall(sw, call.getSOAPMappingRegistry(), resContext);
         resContext.setRootPart(sw.toString(),
                        Constants.HEADERVAL_CONTENT_TYPE_UTF8);
      }
      catch(Exception e) {
         if (e instanceof SOAPException)
            throw (SOAPException )e;

         throw new SOAPException(Constants.FAULT_CODE_SERVER, e.toString());
      }
   }
}
```

我从org.apache.soap.providers.RPCJavaProvider源代码中将invoke()方法的代码提取出来。我所做的几个改动已用黑体字标出来了。第一个改动是调用了BetterRPCRouter的静态invoke()方法，它以org.apache.soap.Header的一个实例作为第一个参数。我们现在还没有实现这个类，但不久就会实现它；这样做是因为默认路由器org.apache.soap.server.RPCRouter不允许我们向它的invoke()方法传递头。其余的参数与org.apache.soap. server.RPCRouter里的一样。扩展默认提供者类的一个好处在于它把获得SOAP封套的工作放在了locate()方法的实现过程中，locate()方法保存在envelope类变量（这是org.apache.soap.Envelope的一个实例）中。因此我们要做的全部工作就是在我们的路由器类中将envelope.getHeader()作为invoke()方法的第一个参数。

现在我们可以接着来实现自己的路由器。这一次我们同样不需要重写整个类；只要扩展org.apache.soap.server.RPCRouter就行了。因为这个基类的其余部分都工作得很好，我们就只需要实现invoke()方法。我们的invoke()方法将以一个org.apache.soap.Header作为它的第一个参数。我们将得到一个向服务方法中传递头的途径。

能够查看源代码的一个好处是，你有时能在其中发现一些平时不注意的技巧。这个代码说明了org.apache.soap.server.RPCRouter实现实际上期望你的服务方法以org.apache.soap.rpc.SOAPContext的一个实例作为它的第一个参数。在那种情况下，默认的路由器只使用了第一个位置之后的其他参数。如果你愿意，可以试试这样做。回到前面章节中的任何一个Apache SOAP服务类，将org.apache.soap.rpc.SOAPContext的一个实例插入方法签名作为第一个参数，将其他的参数顺序后移。如果你在做上述修改后重新编译并部署这个服务，所有的事情还会与从前一样；路由器考虑到了你可能加入第一个特殊参数的情况，它能够正确处理它，不过它并不强求你这样做。这个特征正好可以用来满足我们访问SOAP头的需要。问题是从org.apache.soap.rpc.SOAPContext中获取头的工作量较大。尽管如此，这个机制还是非常好地满足了我们的需要。其结果是使得这个路由器允许我们实现一个能以org.apache.soap.Header的一个实例作为第一个参数的服务方法。这里是BetterRPCRouter类的代码。再一次说明，绝大部分代码都来自其基类，修改过的部分都用黑体标出来了。

```
package javasoap.book.ch10.services;

import org.apache.soap.server.*;
import java.io.*;
import java.util.*;
import java.lang.reflect.*;
import org.w3c.dom.*;
import org.apache.soap.util.Bean;
import org.apache.soap.util.MethodUtils;
import org.apache.soap.util.IOUtils;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.util.StringUtils;

public class BetterRPCRouter extends org.apache.soap.server.RPCRouter {
```

```
public static Response invoke(Header hdr, DeploymentDescriptor dd,
                    Call call, Object targetObject, SOAPContext reqCtx,
                    SOAPContext resCtx)
        throws SOAPException {
    byte providerType = dd.getProviderType();
    Vector params = call.getParams();
    String respEncStyle = call.getEncodingStyleURI();
    Object[] args = null;
    Class[] argTypes = null;
    if (params != null) {
        int paramsCount = params.size();
        args = new Object[paramsCount];
        argTypes = new Class[paramsCount];
        for (int i = 0; i < paramsCount; i++) {
            Parameter param = (Parameter)params.elementAt(i);
            args[i] = param.getValue();
            argTypes[i] = param.getType();
            if (respEncStyle == null) {
                respEncStyle = param.getEncodingStyleURI();
            }
        }
    }

    if (respEncStyle == null) {
        respEncStyle = Constants.NS_URI_SOAP_ENC;
    }

    Bean result = null;
    try {
        if (providerType == DeploymentDescriptor.PROVIDER_JAVA ||
            providerType == DeploymentDescriptor.PROVIDER_USER_DEFINED) {
            Method m = null ;
            try {
                m = MethodUtils.getMethod (targetObject,
                        call.getMethodName(), argTypes);
            } catch(NoSuchMethodException e) {
                try {
                    int paramsCount = 0 ;
                    if (params != null) paramsCount = params.size();
                    Class[] tmpArgTypes = new Class[paramsCount+1];
                    Object[] tmpArgs = new Object[paramsCount+1];
                    for (int i = 0 ; i < paramsCount ; i++)
                        tmpArgTypes[i+1] = argTypes[i] ;

                    argTypes = tmpArgTypes;
                    argTypes[0] = Header.class;
                    m = MethodUtils.getMethod(targetObject,
                                call.getMethodName(),argTypes);
```

```
                for (int i = 0 ; i < paramsCount ; i++)
                   tmpArgs[i+1] = args[i];

                tmpArgs[0] = hdr;
                args = tmpArgs;
             }
             catch (NoSuchMethodException e2) {
                throw e;
             }
          catch (Exception e2) {
             throw e2;
          }
       }          catch (Exception e) {
             throw e;
          }

          result = new Bean(m.getReturnType(),
                            m.invoke(targetObject, args));
       }
       else {
          Class bc = Class.forName("org.apache.soap.server.InvokeBSF");
          Class[] sig = {DeploymentDescriptor.class,
                         Object.class,
                         String.class,
                         Object[].class};
          Method m = MethodUtils.getMethod(bc, "service", sig, true);
          result = (Bean) m.invoke (null,
                    new Object[] {dd, targetObject,
                                  call.getMethodName (), args});
       }
    }
    catch (InvocationTargetException e) {
       Throwable t = e.getTargetException();
       if (t instanceof SOAPException) {
          throw (SOAPException)t;
       }
       else {
          throw new SOAPException(Constants.FAULT_CODE_SERVER,
              "Exception from service object: " + t.getMessage(), t);
       }
    }
    catch (ClassNotFoundException e) {
       throw new SOAPException (Constants.FAULT_CODE_SERVER,
              "Unable to load BSF: script services " +
              "unsupported without BSF", e);
    }
    catch (Throwable t) {
       throw new SOAPException (Constants.FAULT_CODE_SERVER,
```

```
            "Exception while handling service request: " + t.getMessage(), t);
        }

        Parameter ret = null;
        if (result.type != void.class) {
            ret = new Parameter (RPCConstants.ELEM_RETURN, result.type,
                                 result.value, null);
        }

        return new Response(call.getTargetObjectURI(), call.getMethodName (),
                            ret, null, null, respEncStyle, resCtx);
        }
    }
```

invoke()方法的第一个参数是org.apache.soap.Header的一个实例，它包含了来自请求封套的SOAP头。我们使用Java反射来试图获取服务类的方法，并假定该方法的参数及其对应类型都直接与SOAP体中的一致。如果这样做不成功，代码会再次尝试，这一次它寻找一个带有所有SOAP体参数并且第一个参数是org.apache.soap.Header类型的方法。为了这样做，我们将argTypes[0]赋值为Header.class，并将tmpArgs[0]赋值为hdr（invoke()方法中的一个参数）。这些数组在发现并随后自动调用Java方法时用于Java的反射机制。正如你所看到的，代码的绝大部分都没变。它的大部分工作都是在寻找一个合适的服务类方法，处理使用了一个BSF脚本（注2）来实现服务的可能性并处理错误。

一个能处理SOAP头的Apache SOAP服务

我们现在已有了一个提供者和一个路由器，它们可以将SOAP头传递到我们的服务方法，因此我们现在就来实现一个可以使用SOAP头的服务。把我们的服务建立在第九章的urn:ProxyQuoteService上是一个不错的选择。它的实现中将后端服务的物理地址硬编码为*http://mindstrm.com:8004/glue/urn:CorpData-Services*；我们将要求客户应用程序提供该信息作为SOAP头部分。如果在头中没有路由数据，我们的代理服务将默认使用一个假的地址*http://mindstrm.com:8899/glue/*

注2：Apache SOAP允许你用BSF脚本来实现服务。如果你对这个问题感兴趣，可以参考Apache SOAP的文档。

urn:CorpDataServices。如果服务能够访问一个默认的后台服务，也能够根据SOAP头定向到一个由客户端指定的服务，你可以使用类似的技术。

让我们来设计SOAP头，让它包含单个名为targetAddress的子元素。这个元素中包含了一个由代理使用的后台服务地址。头中的元素可以使用SOAP-ENV:mustUnderstand元素并将其值设为"1"，以指定封套的接收方必须能理解头元素并能够正确地使用其中的数据。一个客户端发送的要求接收方理解targetAddress的SOAP头是这样的：

```
<SOAP-ENV:Header>
<targetAddress SOAP-ENV:mustUnderstand="1">
http://mindstrm.com:8004/glue/urn:CorpDataServices
</targetAddress>
</SOAP-ENV:Header>
```

现在来设计我们的服务，让它理解targetAddress头元素，而不管任何其他的头元素。如果有任何其他的头元素使用了一个值为"1"的mustUnderstand属性，服务将返回一个SOAP错误。这个新服务将叫做urn:DirectedQuoteProxy-Service，它由DirectedQuoteProxyService类实现：

```
package javasoap.book.ch10.services;

import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
import org.w3c.dom.*;

import javasoap.book.ch9.services.RemoteQuote;
import javasoap.book.ch9.services.ProxyQuote;

public class DirectedQuoteProxyService {

   static final String TARGETADDRESS = "targetAddress";
   static final String DEFAULTTARGET =
            "http://mindstrm.com:8899/glue/urn:CorpDataServices";
```

```
    public DirectedQuoteProxyService() {
    }

    String getTargetAddress(Header header)
                throws SOAPException  {

      String target = DEFAULTTARGET;

      Vector entries = header.getHeaderEntries();
      for (int i = 0; i < entries.size(); i++) {
        Element el = (Element)entries.elementAt(i);
        String name = el.getLocalName();
        String val = el.getAttributeNS(Constants.NS_URI_SOAP_ENV,
          Constants.ATTR_MUST_UNDERSTAND);

        if (name.equals(TARGETADDRESS)) {
          target = el.getFirstChild().getNodeValue();
        }
        else if (val != null && val.equals("1")) {
          if (!name.equals(TARGETADDRESS))
          throw new SOAPException("MustUnderstand",
            "Service Doesn't Understand Header Element: " + name);
        }
      }

      return target;
    }

    public ProxyQuote getQuote(Header hdr, String symbol)
                throws SOAPException, Exception {

      String target = getTargetAddress(hdr);

      URL url = new URL(target);

      SOAPMappingRegistry smr = new SOAPMappingRegistry();

      BeanSerializer beanSer = new BeanSerializer();

      smr.mapTypes(Constants.NS_URI_SOAP_ENC,
        new QName(
         "http://www.themindelectric.com/package/javasoap.book.ch9.services/",
         "Quote"),
         RemoteQuote.class, beanSer, beanSer);
```

```
        Call call = new Call();
        call.setSOAPMappingRegistry(smr);
        call.setTargetObjectURI("XYZ");
        call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

        String stock = symbol;

        Vector params = new Vector();
        params.addElement(new Parameter("stock", String.class, stock, null));
        call.setParams(params);

        call.setMethodName("getQuote");
        Response resp = call.invoke(url, "");
        ProxyQuote quote = new ProxyQuote();
        if (resp.generatedFault()) {
            throw new Exception("Service Call Failed");
        }
        else {
            Parameter ret = resp.getReturnValue();
            RemoteQuote value = (RemoteQuote)ret.getValue();
            quote.setStockSymbol(value.get_symbol());
            quote.setLast(value.get_lastPrice());
            quote.setDiff(value.get_change());
            quote.setTime(value.get_timeStamp());
            quote.setVol(value.get_volume());
        }

        return quote;
    }
}
```

这段代码与我们在第九章中开发的QuoteProxyClass大部分都是一样的。只是getQuote()方法现在使用了org.apache.soap.Header作为它的第一个参数，而且使用了一个包含股票代码的String作为第二个参数。在构造URL之前，还使用header作为参数调用了getTargetAddress()方法。getTargetAddress()方法在头中没有targetAddress元素的情况下返回假的默认地址，而在targetAddress元素存在时返回该元素的值。我们遍历了所有的头元素来查找targetAddress元素。在查找时，我们还要看看其他的头元素中是否也包含了值为"1"的SOAP-ENV:mustUnderstand属性，如果有，就抛出一个适当的SOAPException异常。抛出这个异常会导致向调用者返回一个相应的SOAP错误。getTargetAddress()的返回值被用来构造URL。代码的其余部分与javasoap.book.ch9.services.QuoteProxyService中是一样的。

我们现在来部署这个服务，要记住使用我们的BetterRPCJavaProvider类来作为provider元素的type属性值。它的部署描述文件应该是下面这样的：

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
             id="urn:DirectedQuoteProxyService">
  <isd:provider type="javasoap.book.ch10.services.BetterRPCJavaProvider"
                scope="Application"
                methods="getQuote">
  <isd:java class="javasoap.book.ch10.services.DirectedQuoteProxyService"
        static="false"/>
  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener
  </isd:faultListener>

  <isd:mappings>
    <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         xmlns:x="urn:QuoteProxyService" qname="x:Quote"
         javaType="javasoap.book.ch9.services.ProxyQuote"
        java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
         xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
  </isd:mappings>
</isd:service>
```

让我们来写一个能访问urn:DirectedQuoteProxyService的Apache SOAP客户应用程序。现在大部分的代码看起来都应该很熟悉了，因此我将只解释其中与构造SOAP头有关的新部分。这个类叫做HeaderClient：

```
package javasoap.book.ch10.clients;

import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
import org.w3c.dom.*;
import javax.xml.parsers.DocumentBuilder;

import javasoap.book.ch9.services.ProxyQuote;

public class HeaderClient {
   public static void main(String[] args)
```

```
throws Exception {

URL url = new URL("http://georgetown:8080/soap/servlet/rpcrouter");

SOAPMappingRegistry smr = new SOAPMappingRegistry();

BeanSerializer beanSer = new BeanSerializer();

smr.mapTypes(Constants.NS_URI_SOAP_ENC,
  new QName("urn:QuoteProxyService", "Quote"),
            ProxyQuote.class, beanSer, beanSer);

Call call = new Call();
call.setSOAPMappingRegistry(smr);
call.setTargetObjectURI("urn:DirectedQuoteProxyService");
call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

String stocks = args[0];

Header header = new Header();
String must = Constants.NS_PRE_SOAP_ENV + ":" +
                          Constants.ATTR_MUST_UNDERSTAND;

Vector entries = new Vector();
DocumentBuilder builder = XMLParserUtils.getXMLDocBuilder();
Document doc = builder.newDocument();
Element elem = doc.createElement("targetAddress");
elem.setAttribute(must, "1");
Text txt = doc.createTextNode("");
txt.setData("http://mindstrm.com:8004/glue/urn:CorpDataServices");
elem.appendChild(txt);
entries.addElement(elem);
header.setHeaderEntries(entries);
call.setHeader(header);

Vector params = new Vector();
params.addElement(new Parameter("stocks", String.class, stocks, null));
call.setParams(params);

try {
   call.setMethodName("getQuote");
   Response resp = call.invoke(url, "");

   if (resp.generatedFault()) {
      Fault fault = resp.getFault();
```

```
            String code = fault.getFaultCode();
            String desc = fault.getFaultString();
            System.out.println(code + ": " + desc);

            Vector v = fault.getDetailEntries();
            int cnt = v.size();
            for (int i = 0; i < cnt; i++) {
               Element n = (Element)v.elementAt(i);
               Node nd = n.getFirstChild();
               //System.out.println(n.getNodeName()
                     + ": " + nd.getNodeValue());
            }
         }
         else {
            Parameter ret = resp.getReturnValue();
            ProxyQuote value = (ProxyQuote)ret.getValue();
            System.out.println("Symbol:      " + value.getStockSymbol() +
                      "\nLast Price: " + value.getLast() +
                      "\nChange:     " + value.getDiff() +
                      "\nTime Stamp: " + value.getTime() +
                      "\nVolume:     " + value.getVol());
         }
      }
      catch (SOAPException e) {
         System.err.println("Caught SOAPException (" +
                        e.getFaultCode() + "): " +
                        e.getMessage());
      }
   }
}
```

我们创建了一个名为header 的org.apache.soap.Header实例，我们将用它来组建头元素。String类型的变量must中保存了mustUnderstand属性的限定名。这个名称由常量值Constants.NS_PRE_SOAP_ENV和Constants.ATTR_MUST_UNDERSTAND组成，中间用“:”隔开。这个值的结果是SOAP-ENV:mustUnderstand。现在我们将不得不在XML层次上工作，就像我们在第七章中开发自定义错误细节项时所做的那样。我们在这里采取了同样的方法，只是不需要创建SOAP-ENV:Header元素了；那是org.apache.soap.Header类的工作。我们只需要在头内部建立子项，因此我们创建了targetAddress作为文档的第一个元素。接下来我们调用了elem.setAttribute()方法来设置mustUnderstand属性，并传递给它一个must字符串变量和一个"1"值作为参数。然后我们又创建了一个文本节点，并将它的值设为http://mindstrm.com:8004/glue/urn:CorpData-

Services，这就是我们的代理服务在定向时要使用的端点。头元素项的矢量通过调用 header.setHeaderEntries()方法添加到头元素中，然后这个头被 call.setHeader()方法添加到了 call 对象中。现在，在创建 SOAP 封套时，我们建立的头将被包含在内。我在命令行用 AMX 作为参数运行了这个客户应用程序，其结果是向服务发送了如下的 SOAP 封套:

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Header>
<targetAddress SOAPENV:mustUnderstand="1">
http://tokyo:8004/glue/urn:CorpDataServices
</targetAddress>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
<ns1:getQuote
xmlns:ns1="urn:DirectedQuoteProxyService"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<stocks xsi:type="xsd:string">AMX</stocks>
</ns1:getQuote>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

第十一章

JAX-RPC 与 JAXM

对于大多数的软件技术来说，当一个技术得到某种程度的广泛应用时，标准化工作就会展开。是的，SOAP 和 Web 服务已经满足了这个要求，因此标准（以及它们所产生的缩写词）正在出现。有大量的与在 Java 中处理 XML 有关的 API 标准，它们将全部被包含在 Java 的 XML API（JAX）之中。JAX 不是一种产品——它是一个 API 定义，而且，它和与 Java 有关的众多 API 规范没什么不同。不过，因为 XML 得到了如此广泛的应用，JAX 就不得不被分成很多组成部分。其中有两个对我们来说有特别的意义：那就是JAX-RPC和JAXM。JAX-RPC是与基于XML的 RPC 有关的 Java API。我希望随着时间的推移，所有主要的 SOAP RPC 实现都能遵循这个 API。JAXM 是有关 XML 消息传递的 Java API，而且与 JAX-RPC 一样，它也有可能被广泛接受。

这两个 API 是全新的，并且还没有准备提供给产品世界使用。JAXM 与 JAX-RPC 规范还处在广泛征求意见阶段，因此在它们定稿之前还会有改动；然而，在这个阶段变化应该会相对小一些。不过，Sun 公司还是提供了一个允许你启用这些技术的参考实现。这个参考实现包含在 Java WSDP（Java Web Services Developer Pack，Java Web 服务开发包）中，它可以在 *http://java.sun.com/webservices/webservicespack.html* 得到。你可以将这个版本视为一个早期的β版本；只要你不把它作为一个产品服务器来使用并且不奢望它已经消除了所有的bug以及具备你所希望的一切特征，它将会帮你节省时间，以便在标准最终完成的时候你能尽快

地建立一个产品服务器。WSDP使用了Tomcat 4服务器，因此如果你到目前为止使用的是Tomcat和Apache SOAP，那你在做与JAX有关的工作时应该会相当轻松。即使你没有用过Tomcat，这个包中的安装说明也是相当简单的，因此你用起来应该没什么困难。

为了研究JAX-RPC，我们将会使用JAX的参考实现来建立一些例子。这些例子将是我们在本书前面部分所开发的那些例子的新版本。因为我假设在此之前你已阅读了本书前面的每一章，因此我们的进度将会很快。我们不准备完整地介绍JAX-RPC和JAXM的每个方面；而是会利用这个机会很快地浏览一下这些API是如何被构造的，以及它们相对于前面章节所介绍的Apache SOAP 2.X和GLUE API有什么区别。换句话说，我们将使用JAX来重建前面章节的某些服务和客户例子。

JAX-RPC

我将首先介绍JAX-RPC，因为我们已经在RPC世界中花费了大多数的时间。像GLUE一样，JAX-RPC对程序员隐藏了底层协议（SOAP）的细节。为了隐藏底层协议，JAX-RPC利用了大家都知道的存根（stub）和层（tie）对象。存根被客户应用程序用来访问一个远程的服务。存根看起来就像服务接口一样，但是它作为本地客户程序的一部分运行。存根使用底层的框架产生一个适当的SOAP消息并通过HTTP将它发送出去。因此从客户应用程序设计者的角度来看，如果你拥有了存根，就不再需要去处理SOAP消息的任何细节了。图11-1描述了客户代码、存根和JAX-RPC客户框架的运行时组件之间的关系。

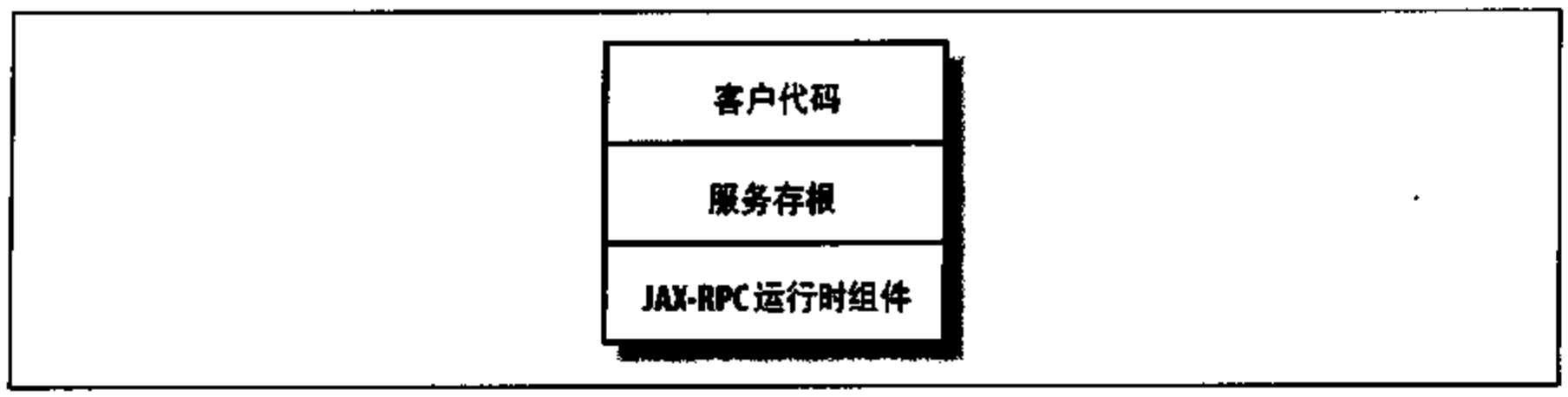

图11-1：JAX-RPC体系结构中的存根

层在概念上与存根很相似，但它存在于服务器端。层用作JAX-RPC运行时组件与服务实现之间的桥梁，它去除了你的服务代码与JAX-RPC框架之间的耦合。接收到的服务消息由JAX-RPC运行时组件处理并通过一个层对象传递到服务。服务的响应向后穿过层到达JAX-RPC运行时组件，在那里它们将被转换成SOAP响应或者SOAP错误，并通过HTTP返回。图11-2显示了服务代码、服务层和JAX-RPC运行时组件之间的关系。

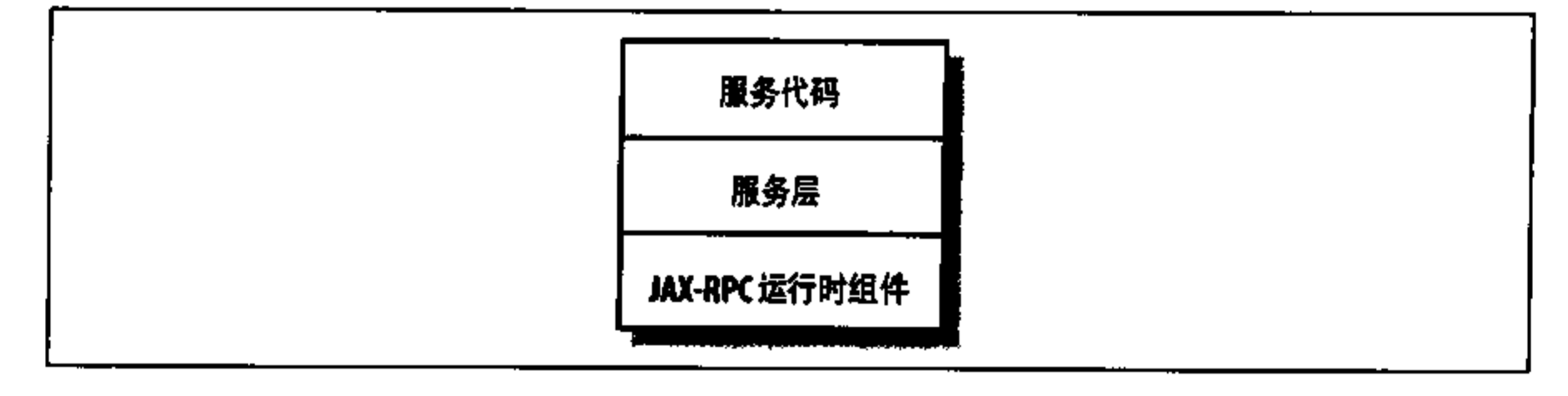

图11-2: JAX-RPC体系结构中的层

回想一下Apache SOAP 2.X和GLUE中有关RPC的机制。Apache向Java程序员暴露了很多的SOAP细节，它还要求程序员在一个具体的层次上处理SOAP。从体系结构上来讲，JAX-RPC更像GLUE，它隐藏了许多底层的细节。这是SOAP API演变的一个必然趋势，不管它们是不是专有的。

不论你的API的复杂性如何，或者你选择了哪一种分布机制，将你的软件中那些用于处理分布式计算的机制抽象化并隔离出来都是一个好习惯。这样会使你在更换机制时对其余的工作带来最小的影响。本章中的例子就使用了这种方式，尽管例子是如此的简单以致于看起来就像是用牛刀杀鸡。图11-3显示了我们将使用JAX-RPC来开发的RPC例子的总体体系结构。

不使用Ant来完成工作

Java Web服务开发包的教程很大程度上依赖于一个叫做Ant的包。所有Java源文件的编译、目录的创建、JAR文件的创建等工作都由Ant的配置文件来管理。虽然Ant相当流行，而且使用它建立在本章中这样类型的代码是一个好主意，但

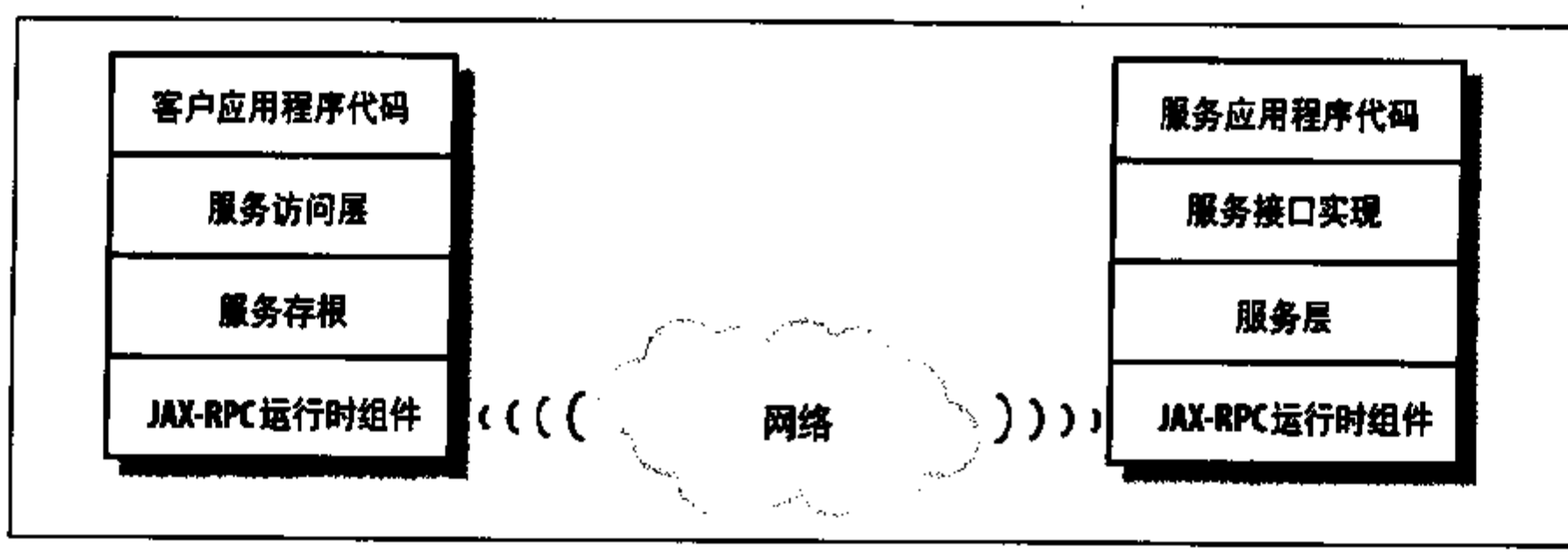

图 11-3：使用 JAX-RPC 的 RPC 体系结构

我们不准备在这里介绍它的用法（更多关于Ant的信息，请参见由Jesse Tilly和Eric Burke编写的《Ant: The Definitive Guide》一书）。相反，我会教你使用手工的步骤来完成这些例子。如果你知道如何使用Ant，就可以跳过本章中的一些细节。

开始创建时最关键的是类的路径。测试版（Early Access Release）里有10个JAR文件需要加入到你的类路径之中。这些文件位于开发包安装根目录中的common\lib目录下，不论你使用什么样的办法，请确认它们是在你的类路径之中。这就是那些JAR文件：

jaxrpc-ri.jar
jaxrpc-api.jar
activation.jar
dom4j.jar
jaxm-api.jar
jaxm-client.jar
log4j.jar
mail.jar
xalan.jar
xerces.jar

创建一个JAX-RPC服务

在第九章中，我们曾介绍了建立位于客户应用程序与其他服务之间的代理服务。在那一章中，与代理通信的后端服务位于*http://mindstrm.com:8004/glue/urn:CorpDataServices*。让我们重做这个工作，只是这一次我们将使用JAX-RPC参考实现来建立我们的代理服务（注1）。这个例子会让我们看到如何使JAX-RPC与GLUE进行互操作，而且它还给我们一个机会使用JAX-RPC来建立一个服务和一个客户。

建立这个服务的第一步是为它编写一个Java接口，在这里我将调用javasoap.book.ch11.services.IStockServiceProxy 。因为JAX-RPC使用了Java RMI包，因此这个接口一定要实现java.rmi.Remote接口。因为Java RMI提供了一个很好的分布式计算抽象，因此这个策略非常通用。接口中的方法要被设计为可以抛出RemoteException，这个异常也来自java.rmi包。我们将定义一个getStockQuote()方法，它以一个代表股票代码的字符串为参数，并返回javasoap.book.ch9.services.ProxyQuote的一个实例。没必要再去建立一个用来返回报价的新类；因为我们在第九章中已经做过了。IStockServiceProxy接口的代码看上去应该是这样的：

```
package javasoap.book.ch11.services;

import java.rmi.Remote;
import java.rmi.RemoteException;
import javasoap.book.ch9.services.ProxyQuote;

public interface IStockServiceProxy extends Remote {
    public ProxyQuote getStockQuote(String symbol) throws RemoteException;
}
```

下一步，我们要编写一个实现IStockServiceProxy 接口的Java类，我将调用javasoap.book.ch11.services.StockServiceProxyImpl。与通常的情况不一样，我将在类名中使用一个Impl后缀，不过这也是一种常见的做法，而且在这里

注1： 我假定你已经成功地下载并安装了Java Web服务开发包，而且能够正确地使用它。

它符合我的要求（通常，我仅仅从接口名称中删掉“I”字母，即把StockServiceProxy作为实现类的名称，但现在我要把这个名称用在别的地方。我想去除实际完成工作的那个类与使用JAX-RPC或其他API的类之间的耦合）。于是StockServiceProxyImpl类就是图11-3中的服务接口实现层的一部分。我们将使用另一个叫做javasoap.book.ch11.services.StockServiceProxy的类来做实际的工作。与StockServiceProxyImpl不同，这个类不仅对关于JAX-RPC的机制一无所知，也对层对象所使用的IStockServiceProxy接口一无所知。这就使得不管我们正在使用哪一种分布式计算机制，都可以不做修改地使用这个类。换句话说，StockServiceProxy是在图11-3中显示的服务应用程序代码层的一部分。我们将在StockServiceProxyImpl类的getStockQuote()方法中使用这个StockServiceProxy的实例，如下面所示：

```
package javasoap.book.ch11.services;

import javasoap.book.ch9.services.ProxyQuote;

public class StockServiceProxyImpl implements IStockServiceProxy {

    public ProxyQuote getStockQuote(String symbol) {
        StockServiceProxy sp = new StockServiceProxy();
        return sp.getStockQuote(symbol);
    }
}
```

现在我们来实现StockServiceProxy类，它的任务是调用urn:CorpDataServices服务上的GetQuote()方法。然而，我想要把与这个基于GLUE的服务通信的部分推迟一下再做介绍，因为这将与它的WSDL文件的使用有关。我们稍后将介绍它。而现在，StockServiceProxy的getStockQuote()方法将只返回一个包含了虚构数据的ProxyQuote实例。这里是它的代码：

```
package javasoap.book.ch11.services;

import javasoap.book.ch9.services.ProxyQuote;

public class StockServiceProxy {

        public ProxyQuote getStockQuote(String symbol) {
```

```
        ProxyQuote pq = new ProxyQuote();
        pq.setStockSymbol(symbol);

        // 为目前的使用设置虚构的数据
        pq.setLast((float)0.0);
        pq.setDiff((float)0.0);
        pq.setTime("NA");
        pq.setVol(0);

        return pq;
    }
}
```

这些是我们实现服务所需要的所有Java类和接口。编译它们，现在是可以为服务创建存根和层的时候了。有一个叫做xrpcc的命令行工具可以用来产生这些类。xrpcc使用一个配置文件作为输入。这个文件是一个XML文件，其中包含了关于类、名称空间、服务名称等的信息。我将它命名为*CorpDataServices.xml*。让我们来看一看它的内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns="http://java.sun.com/jax-rpc-ri/xrpcc-config">
  <rmi name="CorpDataServicesProxy"
    targetNamespace="http://mindstrm.com/wsdl"
    typeNamespace="http://mindstrm.com/types">
      <service name="CorpDataServices"
           packageName="javasoap.book.ch11.client">
        <interface name="javasoap.book.ch11.services.IStockServiceProxy"
         servantName="javasoap.book.ch11.services.StockServiceProxyImpl"/>
      </service>
  </rmi>
</configuration>
```

所有的内容都在configuration标签下面，该标签使用了xmlns属性来将全局名称空间定义为http://java.sun.com/jax-rpc-ri/xrpcc-config。既然我们正在使用一个基于RMI的接口，configuration的第一子元素就叫做rmi（注2）。它的name属性用来指定模型的名称，这个名称一般会作为生成那些文件名称的基础。你可以把它想像成服务定义的名称（它在即将生成的WSDL文件中正是起这

注2： xrpcc也允许使用WSDL文档来替代RMI。我们将在本章后面介绍这种方法。

个作用)。targetNamespace和typeNamespace属性用来声明将分别关联到目标服务和自定义类型的名称空间。在rmi标签之后是service标签，它为xrpcc提供关于用来实现服务的Java类和接口的信息。它的name属性包含服务本身的名称，而packageName属性包含用来生成客户端代码的Java包的名称。在service后面是interface标签，它告诉该工具用于实现服务接口的类。它的name属性包含了服务的Java接口的完全限定名，而servantName则包含了用于实现这个接口的Java类的完全限定名。

接下来我们为服务运行xrpcc。我的工作类路径的根目录是e:\projects\soap，因此我这样使用该命令行：

```
xrpcc -both -keep -classpath e:\projects\soap -d gen CorpDataServices.xml
```

-both选项告诉xrpcc我们想要它为客户和服务器端都生成文件。如果你有什么理由要为客户端和服务器端单独产生文件，也可以改成使用-client或-server。如果你像我这样指定了-keep选项，用于生成代码的Java源文件将不会被删除（默认情况下，xrpcc编译被产生的代码，然后删除这些源文件，因为你以后不会真的需要它们。我使用-keep是为了可以看看这些代码。看看xrpcc都产生了哪些类和方法是很有帮助的，尤其是当你还是新手的时候）。-classpath选项为查找输入类文件指定类路径；-d选项用于指定产生的文件存放的根目录。最后，命令行里的最后一个参数是配置文件名。

这里是由xrpcc产生的文件的一个列表，有组织地存放在xrpcc工作目录的子目录中。其中粗体的项目是目录的名称。为了明白起见，我没有列出那些由于-keep选项而被保留的Java源文件（事实上，我手工删除了它们以确保它们不被配置文件打包）。

gen

CorpDataServices.wsdl

CorpDataServicesProxy_Config.properties

javasoap\book\ch11\client

CorpDataServicesProxy.class

CorpDataServicesProxy_SerializerRegistry.class

CorpDataServicesProxyImpl.class

javasoap\book\ch11\services

GetStockQuote_RequestStruct.class

GetStockQuote_RequestStruct_SOAPSerializer.class

GetStockQuote_ResponseStruct.class

GetStockQuote_ResponseStruct_SOAPBuilder.class

GetStockQuote_ResponseStruct_SOAPSerializer.class

IStockServiceProxy_Stub.class

IStockServiceProxy_Tie.class

javasoap\book\ch9\services

ProxyQuote_SOAPSerializer.class

我把这些文件放在一个独立的*gen*目录中，这是因为稍后它们将被打包成一个JAR文件，而我不希望将整个javasoap.book包的路径都打进包里去。在第一次运行xrpcc时，我曾注意到存根和层的类都被产生到了同一个包中。我不喜欢那样做。当你看完本章后就会明白，我把我的客户端和服务器端的类分开存放在不同的包里了，但是xrpcc没有为我们提供这样一个选项。此外，javasoap.book.ch9.services.ProxyQuote类的串行化器类(ProxyQuote_SOAPSerializer)被放到了javasoap.book.ch9.services包里。我猜想这是有道理的，虽然这不是我希望将它存放到的地方，但这一次我没有去管它。有的人常常会在xrpcc所产生的源程序的包名称上花费很多时间，并手工重新编译它，但我不会为此操心。这仍然是一个测试版本，而且一切将会改变。只要记住这一点：你可能不会对xrpcc所产生的包的结构感到满意。

下一步，我们要建立一个部署描述文件。在部署描述文件中包含的信息用来配置Web服务器上的服务应用程序。这里就是我们这个服务的部署描述文件，它的名称要求是*web.xml*；我将它放到了*gen*目录里。其中的黑体文字表示在本例中专用的代码。

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

```
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems. Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">

<web-app>
   <display-name>CorpDataServices</display-name>
   <description>Corporate Data Services</description>
   <servlet>
      <servlet-name>JAXRPCEndpoint</servlet-name>
      <display-name>JAXRPCEndpoint</display-name>
      <description>
         Endpoint for Corporate Data Services Application
      </description>
      <servlet-class>
        com.sun.xml.rpc.server.http.JAXRPCServlet
      </servlet-class>
      <init-param>
         <param-name>configuration.file</param-name>
         <param-value>
          /WEB-INF/CorpDataServicesProxy_Config.properties
        </param-value>
      </init-param>
      <load-on-startup>0</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>JAXRPCEndpoint</servlet-name>
      <url-pattern>/jaxrpc/*</url-pattern>
   </servlet-mapping>
   <session-config>
      <session-timeout>60</session-timeout>
   </session-config>
</web-app>
```

这还不算完成。既然我们已经为我们的服务实现了Java类和接口，产生了层和存根，并建立了一个部署描述文件，那么下面的步骤就是在一个WAR文件中打包该服务。一个WAR文件是一个Web应用程序档案文件（Web application archive），也就是说，一个WAR文件中包含了根据Java servlet规范部署并运行一个Web应用程序所需要的那些文件。我们将把IStockServiceProxy、StockServiceProxyImpl和StockServiceProxy的类文件拷贝到*gen*目录中的适当目录下，再对javasoap.book.ch9.services.ProxyQuote类做同样的事情。我们需要组织我们的文件和目录，使它们成为一个适当的WAR文件。所有的工作都应从一个叫做WEB-INF（全部使用大写字母）的根目录下开始。将*web.xml*、

*Corp-DataServicesProxy_Config.properties*和*CorpDataServices.wsdl*文件移动到WEB-INF目录中。现在我们需要将Java类放进适当的目录。在*WEB-INF*目录下面创建一个叫做*classes*的目录，并使它成为*javasoap*目录的根目录。换句话说，移动*javasoap*目录和它的全部内容到*WEB-INF\classes*目录中。Ant会使管理这个过程非常简单：你只要创建一些配置文件，它会做其余的工作。但是现在你知道了Ant在幕后都会做些什么。

让我们来创建WAR文件。在*gen*目录下执行下面的指令：

```
jar cvf corpdata.war WEB-INF
```

这会把所有我们需要的东西打包到一个名为*corpdata.war*的WAR文件中。图11-4使用WinZip程序显示了该WAR文件的内容。

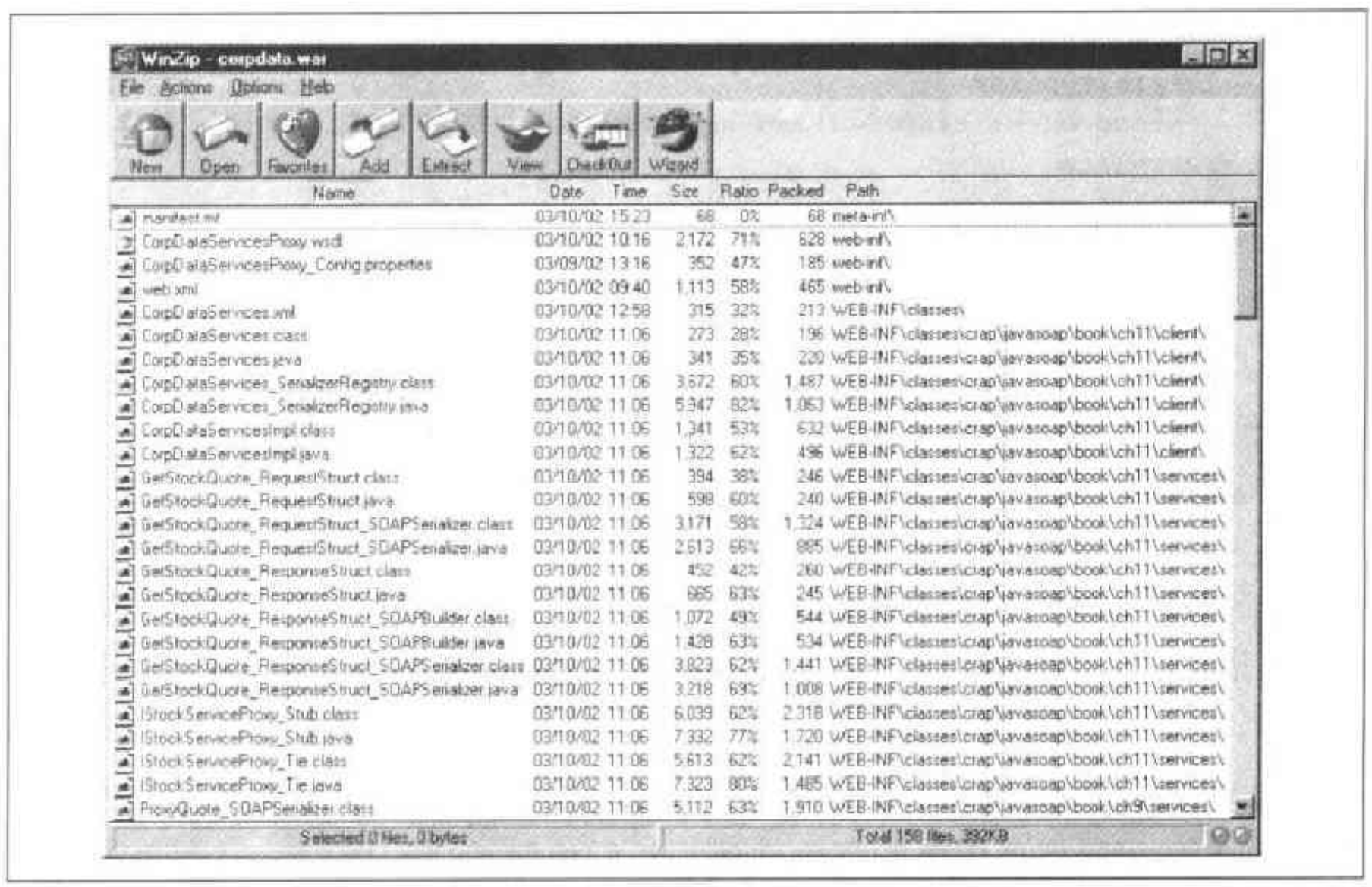

图11-4：corpdata.war的内容

现在我们完成了吗？还不完全是这样。最后一个步骤是部署服务，这很简单，仅仅需要将这个WAR文件拷贝到Tomcat 4的*webapps*目录下就行了（记住Tomcat 4是包含在Java WSDP中的）。那么我们现在就来这样做。我的WSDP安装在

e:\jwsdp-1_0-ea1，因此我把*corpdata.war*文件拷贝到了*e:\jw-sdp-1_0-ea1\webapps*目录里。如果你已经运行了Tomcat 4，退出并重新启动它；这样它将会读取*corpdata.war*文件并部署服务。可以在你的浏览器中输入*http://localhost:8080/corpdata/jaxrpc*来检查确认你的服务是否已被正确部署。如果服务被正确地部署了，你将会看到如下的内容：

```
A Web Service is installed at this URL.
It supports the following ports: "IStockServiceProxy" (http://localhost:8080/
corpdata/jaxrpc/ IStockServiceProxy)
```

好了，很容易吧！实际上，它并不像看起来那么糟糕。许多工作都是打包和部署的工作。如果你使用Ant工具而不是手工来做，一切都会比较快地完成。记住这只是参考实现的一个测试版本，因此你可以坚信在这个系统成为产品时它会有更好的部署工具。

现在既然有了一个已部署好的服务，让我们再花一点时间来仔细思考一下。开发过程是否比我们在Apache SOAP中所用的更好？看来是这样的。我们不必编写任何代码，除了实现该服务所需要的那些之外，而在Apache SOAP中，我们不得不编写与传输协议、类型映射等等有关的大量代码。特别地，请注意我们不再需要去直接处理我们自定义的ProxyQuote类了。存根和层处理了整个映射和串行化过程。

从服务部署的角度来说这与GLUE相比又怎么样呢？它们是差不多的，虽然在某些细节上存在明显的不同。身为一个开发者，我认为我当然喜欢不必对任何生成的代码做进一步处理工作的方式。为什么我必须要去处理这些类文件呢？甚至，为什么要让我意识到它们的存在？使用GLUE，其中的一些工作被动态地完成了，这使它管理起来更容易一些。你仍然需要一些客户端存根，但是服务的开发隐藏了所有事情。在我看来，那是一个优点。

创建一个JAX-RPC客户应用程序

在开始编写一个客户应用程序之前，让我们确保已将前面产生的所有类文件都拷贝到了本地计算机上的适当位置。记住我们把它们包含在一个WAR文件里了，而

且它们还存在于 *gen\WEB-INF\classes* 目录之下。但是这些位置都不在我的局部类路径中，因此现在是将它们放到适当位置的时候了，这样客户应用程序就能够访问到它们。

为这个服务编写一个客户应用程序相当简单。你所要做的全部事情就是取得一个前面生成的存根的实例，并通过这个对象调用所需要的服务方法。这里是一个叫做 javasoap.book.ch11.client.StockQuoteApp 的例子应用程序类的代码：

```
package javasoap.book.ch11.client;

import javasoap.book.ch11.services.*;
import javasoap.book.ch9.services.*;

public class StockQuoteApp {
   public static void main(String[] args) {
      try {
         CorpDataServicesProxyImpl proxy =
             new CorpDataServicesProxyImpl();
         IStockServiceProxy_Stub stub =
             (IStockServiceProxy_Stub)proxy.getIStockServiceProxy();
         stub._setTargetEndpoint(args[0]);
         ProxyQuote quote = stub.getStockQuote("IBM");
         System.out.println("Price is " + quote.getLast());
         System.out.println("Volume is " + quote.getVol());
         System.out.println("Timestamp is " + quote.getTime());
         System.out.println("Diff is " + quote.getDiff());
      } catch (Exception ex) {
         ex.printStackTrace();
      }
   }
}
```

我们创建了 CorpDataServicesProxyImpl 的一个实例，然后调用了它的 getIStockServiceProxy()方法。这样做为我们给出了存根的一个引用，即IStockServiceProxy_Stub的一个实例。我们需要通过它的_setTargetEndpoint()方法将端点的 URL 传递给该存根。这个 URL 是 *http://localhost:8080/corpdata/jaxrpc/IStockServiceProxy*，是我们在前面部署的那个服务端点的位置。现在我们可以简单地调用getStockQuote()方法，其结果是产生了一个对同名服务方法的调用。注意返回值是 javasoap.book.ch9.services.ProxyQuote的一个实例。

客户端和服务器端都自动对这个自定义类型进行了串行化处理。运行这个应用程序将产生下面的输出:

```
Price is 0.0
Volume is 0
Timestamp is NA
Diff is 0.0
```

这就是使用JAX- RPC来为一个服务开发客户和服务代码的方法。但是事情不会总在你的预料之中。我前面已经提到过，xrpcc能利用WSDL文件。那也许是一个产生访问GLUE服务的存根的不错的方式(而且或许它是为访问网络某处的任意一个服务而生成存根的最好方法)。接下来我们将试试这样做。

根据 WSDL 生成存根

根据WSDL来产生客户代码相当简单。首先我们将修改在前面用作xrpcc输入的那个*CorpDataServices.xml*文件。我们必须删去整个rmi段并以一个wsdl段取代它。修改后的文件看起来应该是这样的:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns="http://java.sun.com/jax-rpc-ri/xrpcc-config">
  <wsdl name="CorpDataServicesProxy"
        location="http://mindstrm.com:8004/glue/urn:CorpDataServices.wsdl"
        packageName="javasoap.book.ch11.glueclient">
  </wsdl>
</configuration>
```

name属性指定了模型的名称，正如它在前面的RMI例子所做的那样。location属性包含了服务的WSDL资源地址。我们将使用第九章中的那个GLUE服务；它的WSDL是*http://mindstrm.com:8004/glue/urn:CorpDataServices.wsdl*。packageName属性与以前一样，指定了生成的代码的包名。这次我们将使用javasoap.book.ch11.glueclient，因为这个代码必须能访问我们的那个基于GLUE的服务。运行下面的命令来产生客户端代码:

```
xrpcc -client -keep -classpath e:\projects\soap -d gen  CorpDataServices.xml
```

这与我们在前面所使用的指令基本上是一样的，除了-client选项以外，它用来说明我们只想生成客户端代码。完成这个过程以后，应该把这个Java类文件移动到正确的javasoap.book.ch11.glueclient目录里，以便它们能出现在类路径中。关注一下javasoap.book.ch11.glueclient.Quote类。这是一个用来表示从GLUE服务返回的数据的自定义类型；在WSDL文档中已经对它做了类型声明。很棒！我们不必再为处理这个类型做特别的事情了。

让我们编写一个叫做javasoap.book.ch11.glueclient.StockQuoteApp的客户应用程序。它看起来将和其他的StockQuoteApp差不多，但是它会和已存在的基于GLUE的服务进行会话。这里是它的代码：

```
package javasoap.book.ch11.glueclient;

public class StockQuoteApp {
   public static void main(String[] args) {
      try {
         CorpDataServiceImpl proxy =
             new CorpDataServiceImpl();
         CorpDataServiceSoap_Stub stub =
             (CorpDataServiceSoap_Stub)proxy.getCorpDataServiceSoap();
         stub._setTargetEndpoint(args[0]);
         Quote quote = stub.getQuote("IBM");
         System.out.println("Price is " + quote.get_lastPrice());
         System.out.println("Volume is " + quote.get_volume());
         System.out.println("Timestamp is " + quote.get_timeStamp());
         System.out.println("Change is " + quote.get_change());
      } catch (Exception ex) {
         ex.printStackTrace();
      }
   }
}
```

运行这个应用程序需要在命令行中指明服务端点的URL。在本例中，正确的URL是*http://mindstrm.com:8004/glue/urn:CorpDataServices*。当我运行它时，得到了如下的输出：

```
Price is 105.09
Volume is 10726400
Timestamp is Mar  8
Change is 1.38
```

可以使用这个例子的代码来使我们前面的服务与GLUE服务进行会话以取得数据，而不再像前面那样使用虚构的数据了。到这个时候，你应该已经非常清楚怎样完成这个修改了，因此我将只快速浏览一遍这段代码。我们需要修改StockServiceProxy类的getStockQuote()方法。修改后的代码看起来应该是下面这样的：

```
package javasoap.book.ch11.services;

import javasoap.book.ch9.services.ProxyQuote;
import javasoap.book.ch11.glueclient.*;

public class StockServiceProxy {

   public ProxyQuote getStockQuote(String symbol) {

      ProxyQuote pq = new ProxyQuote();
      try {
         CorpDataServiceImpl proxy =
             new CorpDataServiceImpl();
         CorpDataServiceSoap_Stub stub =
             (CorpDataServiceSoap_Stub)proxy.getCorpDataServiceSoap();
         stub._setTargetEndpoint(
             "http://mindstrm.com:8004/glue/urn:CorpDataServices");
         Quote quote = stub.getQuote(symbol);

         pq.setLast(quote.get_lastPrice());
         pq.setVol(quote.get_volume());
         pq.setTime(quote.get_timeStamp());
         pq.setDiff(quote.get_change());
      }
      catch (Exception ex) {
         ex.printStackTrace();
      }

      return pq;
   }
}
```

我们导入了javasoap.book.ch11.glueclient包；这个包中包含了用来和urn:CorpDataServices服务会话的存根代码。这次我们来硬编码传给_setTargetEndpoint()的URL。在从getQuote()调用的返回值中获得一个Quote实例之

后，我们使用它来组装 ProxyQuote的一个实例。记住，这两个服务返回的数据类型是不同的。然后我们简单地返回了 ProxyQuote实例。这是相当简单的，得谢谢WSDL。当然，我们还必须编译修改后的StockServiceProxy类，并产生和部署一个更新过的WAR文件来包含这个修改后的代码以及整个javasoap.book.ch11.glueclient软件包。

动态调用接口

有时候，对于应用程序接口的方式，你会想要得到对服务的更多控制。需要较多控制的一个理由是你的应用程序如果预先不知道服务方法和参数，它就不可能产生存根。你所调用的服务也可能没有提供用来描述它的 WSDL 文件。DII（Dynamic Invocation Interface，动态调用接口）是一个允许你在更接近于JAX-RPC运行时的层次上调用服务的API。我们不准备在DII上花费时间，但你应该知道它。当我们在第九章开发那个简单的Axis例子时，我们曾看到过它的结构。它可能与这里介绍的不完全一样，而且这些API仍在演变中，但是方法是相同的。

我将DII看成是编写与SOAP Web服务进行通信的代码的最终手段。让一个工具来根据WSDL描述产生底层的代码是最好的选择，因为WSDL是与语言和实现无关的，而且它是一个被普遍承认的标准。如果你的SOAP工具能利用WSDL文件来完成工作，那当然是最好不过了。

用一句话介绍 JAXM

JAXM为编写消息接发客户和服务所提供的API是在一个低层次上的，与我们在第八章中所见过的类似。它们要求你创建与服务器（或者中间媒介）之间的连接，并且建立用来组装消息的XML。虽然这是一项强有力的技术，但是它同时也给你带来了许多束缚。我喜欢直接使用XML的那种灵活性，但是我仍然想要RPC形式的交互的结构。如果你的服务需要设计成接受XML文件作为参数，你可能会发现直接编码在RPC的上下文里用起来非常合适。如果这还不行，那么就只能采用消息接发（或者专有的）方式了。

我建议你看一看Java WSDP中教程所带的例子。尽管我们在这里没有开发一个JAXM例子，你也有了足够的知识能够弄懂到底发生了什么。某些事情提醒了我，这本书的一个将来版本会深入地探讨JAXM、JAX-RPC和JAX包中的其他内容。

接下来会是什么？

我想你现在已经是跃跃欲试准备编写Java的Web服务了。要及时跟上发展的脚步。要掌握最新技术的资讯，包括已存在的实现的新版本。GLUE、Axis（Apache SOAP）、.NET以及许多其他工具的新版本将会使在Java中开发服务变得更加容易、更加完善，并且更为强大。最后，关注标准化的API、安全问题、服务注册表和IDE集成；这个列表的增长速度会和带宽需求增长的速度一样快！赶快投入进来吧。

词汇表

accessor
存取器

byte array
字节数组

COM（Component Object Model）
组件对象模型

data encoding
数据编码

deployment descriptor
部署描述文件

DII（Dynamic Invocation Interface）
动态调用接口

helper class
辅助类

heterogeneous array
异构数组

homogeneous array
同构数组

JAF（JavaBeans Activation Framework）
JavaBeans活动框架

JSM（Java Message Service）
Java消息服务

markup language
标记语言

marshalling
编组

message passing system
消息传递系统

method signature
方法签名

MOM（Message-Oriented Middleware）
面向消息的中间件

multi-reference
多重引用

RPC（Remote Procedure Call）
远程过程调用

schema
模式

self-describing data
自我描述数据

service activation
服务激活期

service scope
服务作用域

SOAP (Simple Object Access Protocol)
简单对象访问协议

sparse array
稀疏数组

unmarshalling
解组

URN (Universal Resource Name)
统一资源名称

Web service
Web服务

WSDP (Web Services Developer Pack)
Web服务开发包

XML (Extensible Markup Language)
可扩展标记语言