Web 服务精髓

Ethan Cerami 著
陈逸 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

O'Reilly & Associates, Inc. 授权中国电力出版社出版

中国电力出版社

图书在版编目（CIP）数据

Web 服务精髓 /（美）塞拉米（Cerami, E.）著；陈逸译．- 北京：中国电力出版社，2002

书名原文：Web Services Essentials

ISBN 7-5083-1302-X

I. W... II. ①塞 ... ②陈 ... III. 因特网－程序设计 IV. TP393.4

中国版本图书馆 CIP 数据核字（2002）第 099022 号

北京市版权局著作权合同登记

图字：01-2002-2783 号

书　　名 / Web 服务精髓
书　　号 / ISBN 7-5083-1302-X
责任编辑 / 夏平
封面设计 / Ellie Volckhausen，张健
出版发行 / 中国电力出版社（www.infopower.com.cn）
地　　址 / 北京三里河路 6 号（邮政编码 100044）
经　　销 / 全国新华书店
印　　刷 / 北京市地矿印刷厂
开　　本 / 787 毫米×1092 毫米　16 开本　20.5 印张　300 千字
版　　次 / 2003 年 4 月第一版　2003 年 4 月第一次印刷
印　　数 / 0001-5000 册
定　　价 / 35.00 元（册）

作者简介

Ethan Cerami 是 Mount Sinai 医学院计算生物医学研究所的一名软件工程师。他还是纽约大学计算机科学系的一名助教。

封面介绍

本书封面上的动物是一只大螯虾（spiny lobster，又名 rock lobster）。在世界范围内，大螯虾共有 45 种，大小从 2 磅到 26 磅不等。大螯虾有着布满棘的外壳和长长的触角。但与美洲龙虾不同的是，它们没有大前螯，并且尾部比美洲龙虾粗。大螯虾橙色或褐色的外壳上布满绿色、蓝色和黄色的鲜艳斑点。大螯虾栖居于热带和亚热带水域的多岩礁浅水地带，在南半球的冷水中也有分布。白天它们常潜伏在岩礁缝隙里，夜里出来觅食。大螯虾食性较广，能吃贝类、蟹、小鱼、海胆，有时也吃藻类和海草。它们出生后 7~10 年性成熟，寿命长达 30 多年。

大螯虾虽然独居，但它们的社会性比美洲龙虾强，经常共居于珊瑚礁上的洞穴中。渔民和潜水者发现了一个奇怪的现象，上千甚至上万只大螯虾经常在十月或十一月的风暴后集体大迁徙，这种现象被称为“ 螯虾大行军 ”。迁徙时，大螯虾排成一路纵队，由浅水区游向深水区。虽然大螯虾是夜间活动的生物，但是在迁徙时，它们还是经常在大白天行进。到目前为止，还没有关于这种现象的科学解释。

前言

Web服务为建立分布式Web应用提供了一个新的、不断发展的典范。本书集中介绍了Web服务的精髓，涵盖了四种主要技术：XML-RPC、SOAP、WSDL和UDDI，对每个技术都做了高水平的概述。本书还描述了相关的API，并讨论了每种技术的实现条件。本书提供了大量实例，读者立即就可以亲身体验Web服务的实际运行。

本书的读者

本书适合刚开始接触Web服务的开发者。本书致力于描绘出一幅宏观展望，使你能够理解Web服务的范围和程度，同时还提供了足够的具体细节和样例代码，使你能够学会编写自己的Web服务。

当面临专用系统还是开源实现的选择时，我们往往选择开源实现；当面临编程语言的选择时，我们会选择Java。要充分利用本书，你应该具有丰富的Java编程经验。如果需要复习Java，请参考这几本书：

- 《Learning Java》，Patrick Niemeyer 和 Jonathan Knudsen 编著（O'Reilly & Associates 公司出版）（译注 1）
- 《Java in a Nutshell》（第四版），David Flanagan 编著（O'Reilly）

对 XML（eXtensible Markup Language，可扩展标记语言）有个基本的理解也很重要。要牢固掌握 XML，可以参考这几本书：

- 《Learning XML》，Erik T. Ray 编著（O'Reilly）
- 《XML in a Nutshell: A Desktop Quick Reference》，Elliotte Rusty Harold 和 W. Scott Means 编著（O'Reilly）

本书的组织结构

本书分为五个部分。第一部分是Web服务的总体介绍；第二部分至第五部分集中介绍核心的 Web 服务技术，包括 XML-RPC、SOAP、WSDL 和 UDDI。本书最后给出了普通 Web 服务术语表。

第一部分 Web 服务简介

第一章介绍 Web 服务、Web 服务体系结构和 Web 服务协议栈，还简要介绍了当前 W3C（World Wide Web Consortium，万维网联盟）在实现 Web 服务标准化方面的工作。

第二部分 XML-RPC

第二章全面介绍了 XML-RPC，包括 XML-RPC 技术概览以及对 XML-RPC 数据

译注 1： 本书及下面提到的三本书的中文版分别为《Java 语言入门》、《Java 技术手册》、《XML 入门》及《XML 技术手册》，均已由中国电力出版社引进出版。详情请访问：*http://www.infopower.com.cn*。

类型、请求和响应的详细解释。本章还包括用Java和Perl编写的XML-RPC样例代码。

第三部分 SOAP

第三章全面阐述了SOAP，包括对SOAP规范、通过HTTP实现SOAP和W3C针对SOAP标准化工作的介绍。

第四章提供了如何使用Apache SOAP（SOAP规范的开源Java实现）的快速入门指南。本章详细解释了如何安装和部署SOAP服务，以及如何编写基本的服务和客户端代码。

第五章是有关Apache SOAP编程的深层次指南，包括如何使用数组、JavaBeans™和直接XML文档的总体介绍。本章还讨论了如何处理SOAP故障和如何维持会话状态。

第四部分 WSDL

第六章全面介绍了WSDL，包括规范本身的概览，并提供了大量的WSDL样例，还介绍了WSDL调用工具。

第五部分 UDDI

第七章全面介绍了UDDI，包括UDDI数据模型的概览，以及搜索已有数据和发布新数据的指南。

第八章提供了有关UDDI查询API的快速参考。

第九章介绍了一个开源Java实现 —— UDDI4J。样例代码说明了如何搜索和发布UDDI数据。本章还完整地介绍了UDDI4J API。

排版约定

本书使用了如下英文字体约定：

斜体（*italic*）用于：

- 路径名、文件名、函数名和程序名
- Internet 地址，如域名和 URL
- 新定义的术语

等宽字体（`constant width`）用于：

- 命令行和必须逐字输入的选项
- 程序中的名称和关键词，包括方法名、变量名、类名、值名和 XML-RPC 首部
- XML 元素标签

等宽黑体（constant width bold） 用于表示在程序代码行中强调的部分

等宽斜体（*`constant width italic`*）用于在程序代码中表示可代替的参数

建议与评论

本书的内容都经过测试，尽管我们做了最大的努力，但错误和疏忽仍然是在所难免的。如果你发现有什么错误，或者是对将来的版本有什么建议，请通过下面的地址告诉我们：

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

你还可以给我们发电子消息。要加入邮件列表或索要书目，请发电子邮件到：

info@oreilly.com

要询问技术问题或对本书发表评论，请发电子邮件到：

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

本书还有一个专门的网站，我们将在这个网站上列出例子、勘误表以及再版计划。可以通过下面的网址访问这个网站：

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

要了解本书和其他书的信息，请访问 O'Reilly 网站：

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

致谢

写一本 O'Reilly 出版的书一直是我的一个梦想。当然，不是单凭我个人的能力就可以实现这个梦想的，所以我要感谢那些帮助我实现梦想的人们。

首先，我要感谢本书的编辑 —— 来自 O'Reilly 的 Simon St.Laurent。他是第一个让我关注 Web 服务的人，从最初阶段到最后一轮校稿，每一步他都给予了耐心的指导。他还撰写了第二章。我还要感谢所有对本书初稿提出宝贵意见的技术审稿人，他们是 Leigh Dodds、Timothy J. Ewald、Martin Gudgin、Simon Horrell 和 Tim O'Reilly。Mind Electric 公司的首席执行官 Graham Glass 解答了我提出

的大量有关 GLUE 平台和 WSDL 的一般性问题。XMethods 公司的创办人之一 Tony Hong 帮助我解决了有关 SOAP 互操作性的问题，并允许再版 XMethods eBay Price Watcher Service 的 WSDL 文件。本书制作编辑 Claire Cloutier 的出色工作保证了本书拥有良好的组织结构并能按时出版。

其次，我要感谢我在 Winstar Communications 的老板 Gary Lazarus。他允许我为完成本书而灵活安排时间。为此，我永远感激他。

再者，我要感谢我的朋友和我的家人。是的，我指的就是你们，你们养育了我，并帮助我保持和谐的生活。谢谢你们。

然后，我要感谢我的岳父 Ed Orsenigo，您的勇气和坚毅时刻鼓舞着我们。

最后，我要感谢我的妻子 Amy。在我撰写本书期间，我们竟然能挤出时间共结连理。2001 年 9 月 1 日是我一生中最幸福的日子。谢谢你，Amy，谢谢你对我的支持和鼓励，谢谢你为周围的人所带来的快乐。

目录

第二部分 XML-RPC

第三部分 SOAP

第一部分

Web服务简介

第一章

简介

今天，万维网的主要作用是交互式访问文档和应用程序。这种访问几乎全部是人工操作，一般通过 Web 浏览器、音频播放器或其他交互式前端系统实现。如果 Web 扩展到能支持应用程序间的交流（即从程序到程序），其功能和应用范围就会大幅度提高。

— W3C XML 协议工作组（W3C XML Protocol Working Group）宪章

欢迎来到 Web 服务世界！本章将介绍基本的 Web 服务术语和结构，从而为你理解 Web 服务打好基础。我们将以回答如下最常见的问题的形式展开讨论：

- Web 服务到底是什么？
- 什么是 Web 服务协议栈？
- XML 消息接发、服务描述、服务发现分别指的什么？
- XML-PRC、SOAP、WSDL 和 UDDI 指的是什么？这些技术是如何互相补充和协同工作的？
- Web 服务存在哪些独特的安全问题？
- 现在有什么标准？

Web服务简介

Web服务是一类可以从Internet上获取的服务的总称，它使用标准的XML消息接发系统，并且不受任何操作系统和编程语言的约束（见图1-1）。

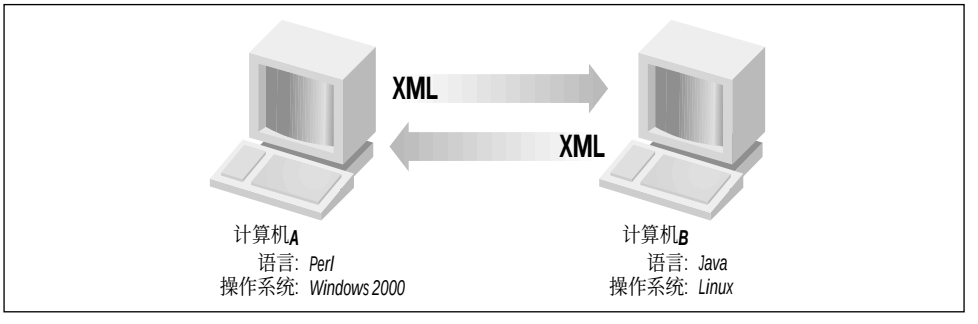

图1-1：基本的Web服务

XML消息接发有多种方法。本章后面将介绍的XML-RPC（XML Remote Procedure Call，XML远程过程调用）和SOAP都可以用于XML消息接发；也可以单独使用HTTP GET/POST并传送任意的XML文档。这两种方法都很有用（见图1-2）。

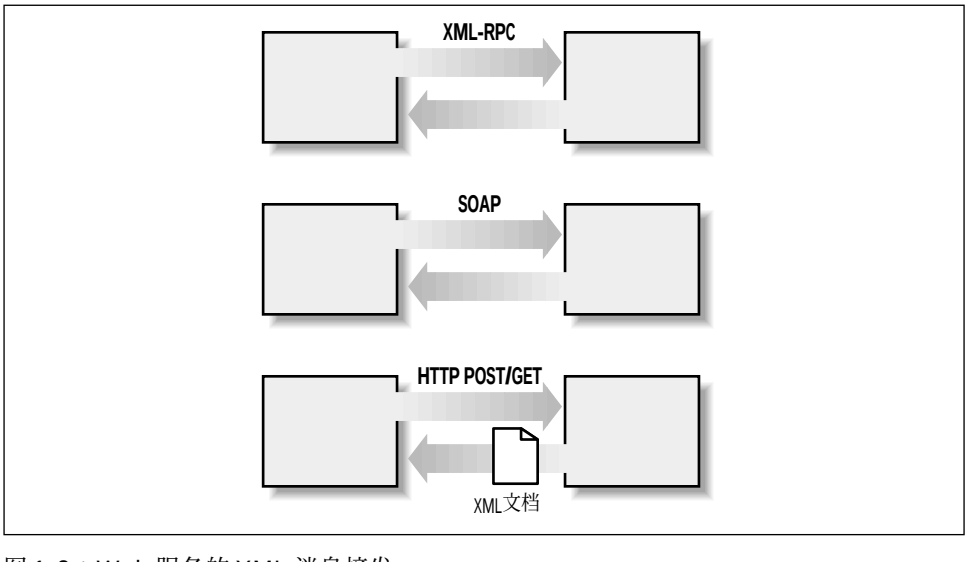

图1-2：Web服务的XML消息接发

虽然不是必需的，但 Web 服务还可以有两个附加的（也许是所期望的）性质：

Web服务应是自描述的。发布一个新的Web服务时，你还应该发布这个服务的公共接口。服务至少应该包含可供阅读的文档，以便其他开发者能更容易地链接你的服务。如果你创建了SOAP服务，那么在理想状态下，你还应该包含一个按照普通XML语法编写的公共接口，可以用XML语法标明所有的公共方法、方法参数和返回值。

Web服务应具有可发现性。如果你创建了Web服务，那么应该有相对简单的机制来发布你的服务。同样，还应该有方便相关用户找到你的服务及其公共接口的某种简单机制。严格的机制应该是通过完全的分布系统或更合乎逻辑的集中注册系统。

总地来说，完整的 Web 服务应该是满足以下条件的服务：

- 可以从 Internet 或内联网获取；
- 使用标准的 XML 消息接发系统；
- 不受任何操作系统或编程语言约束；
- 使用普通的 XML 语法，可以自描述；
- 用简单的查找机制就可以发现。

今天的 Web：以人为中心的 Web

为了更具体地描述Web服务，我们来看一个基本的电子商务功能。Widgets 公司在其网站上出售零件，并为顾客提供提交订单和查询订单状态的服务。

要查询订单状态，顾客需要通过 Web 浏览器登录公司网站，接收 HTML 网页形式的结果（见图 1-3）。

这个基本模型说明了一个以人为中心（human-centric）的Web，人在其中扮演了启动大部分 Web 请求的主要角色。它代表了当今大部分 Web 操作的主要模型。

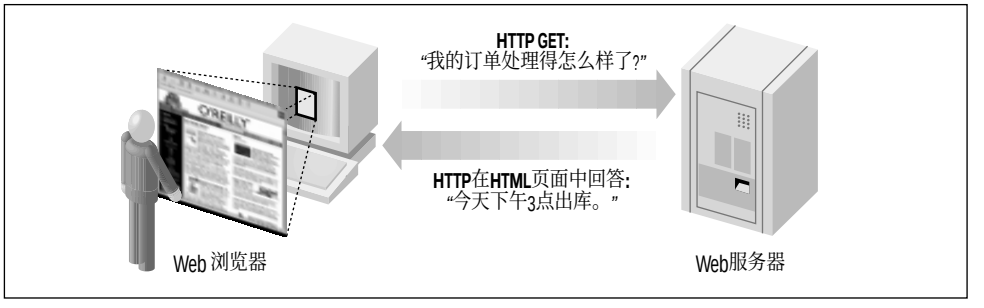

图 1-3：以人为中心的 Web

Web 服务：以应用程序为中心的 Web

有了 Web 服务，我们就可从以人为中心的 Web 迈入以应用程序为中心的（application-centric）Web。但这并不是说人将完全不起作用，这仅仅意味着应用程序间的直接会话变得和 Web 浏览器与服务器间的会话一样容易。

譬如，我们可以将订单状态应用程序转变成Web服务，应用程序和客户就能连接到该服务，并直接使用它的功能。商品目录应用程序可以查询 Widgets 公司的所有订单状态，商品目录系统可以处理、操作数据，并将数据加入到总供应链管理软件中（见图 1-4）。

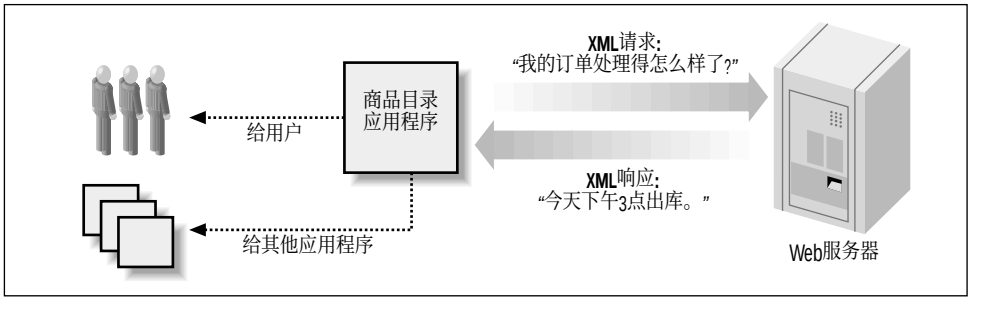

图 1-4：以应用程序为中心的 Web

事实证明，以应用程序为中心的Web在信用卡验证、包裹跟踪、公文跟踪、购物追踪、货币兑换和语言翻译等许多领域都非常有用。以应用程序为中心的Web还

可以用于个人信息的集中储存，例如，微软推荐的 .NET MyServices 项目，其作用是汇集日历、电子邮件、信用卡信息，并提供共享这些数据的 Web 服务。

Web 服务和语义 Web

Web 的原创者 Tim Berners-Lee 最近提出“语义 Web（Semantic Web）”这一概念。语义 Web 以应用程序为中心，很多想法都与 Web 服务一致。实际上，在关于 Web 服务的第一届 W3C 会议上，Berners-Lee 就提出 Web 服务实际上是语义 Web 的一个实现。要总体了解语义 Web，请参阅 Berners-Lee 在《Scientific American》上发表的文章：*http://www.sciam.com/2001/0501issue/0501berners-lee.html*。

Web 服务的梦想：自动化的 Web

以应用程序为中心的 Web 并不是一个新概念。多年来，开发者已经创建了主要供其他应用程序使用的 CGI 程序和 Java servlet，如一些公司开发的信用卡服务、搜索系统和新闻检索系统。

与 Web 服务的关键区别是这些系统大部分由专门的解决方案构成。我们有望实现 Web 服务的标准化，这样就能减少应用程序连接时的障碍。

注意：Web 服务结构为使表现形式彻底地独立于内容提供了一个有趣的选择。例如，一个网站可以只包含用 SOAP 或 XML-RPC 向逻辑地址传递参数的容器页。这样，改变表现形式将变得更加容易，人和计算机还可以“共享”同一 Web 服务。

从长远来看，Web 服务还为自动化 Web（automated Web）带来了希望。如果 Web 服务能很容易地被发现，能自描述，并遵循共同的标准，那么我们就有可能实现自动应用程序集成。某些业内人士称之为“即时（just-in-time）”应用程序集成。

譬如，MegaElectric（ME）想从 Widgets 公司购买零件，还想把订单状态无缝集成到统一的商品目录系统中。总有一天，ME 可以买到自动处理整个过程的软件。它可能用下面的过程（见图 1-5）：

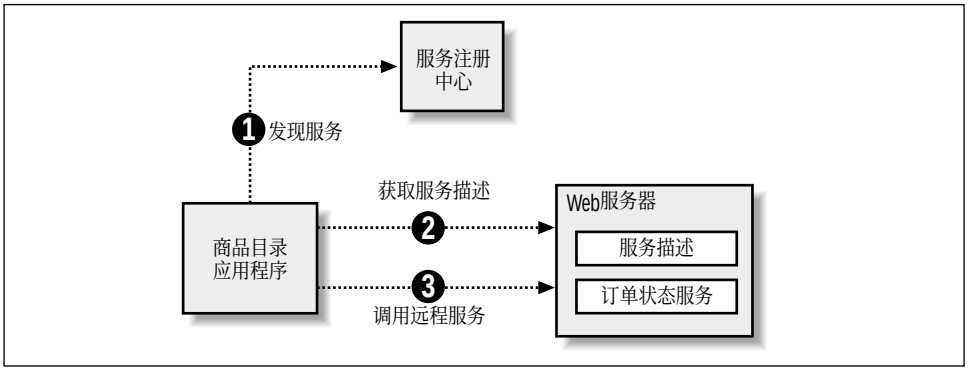

图 1-5：自动化 Web

1. 商品目录应用程序激活，连接到集中式 Web 服务目录，询问“ Widgets 公司提供订单状态服务吗? ”，服务目录返回 Widgets 公司的信息，并包含一个指向服务描述的指针。
2. 商品目录应用程序连接到 Widgets 公司，查找服务描述。
3. 服务描述文件包括了连接到指定服务的全部细节。这样，商品目录应用程序就可以自动调用订单状态服务了。

用现在的Web服务技术能自动完成这一过程吗？不能，现在只有部分过程可以自动完成。譬如，在第九章我们将会看到，创建用来查询服务注册中心的Java 程序是可能的，但是理解结果和选择真正使用哪个服务还需要人的参与。根据服务描述自动调用服务也是可能实现的，如在第六章我们将会看到，现在就有许多运行良好的自动调用工具。

即使所有这些步骤都能实现自动化，但是现在还没有实现自动化业务关系的机制。譬如，现在的服务描述不包括价格保证、送货安排和没有送货的合法处理。即使有了服务描述，也不能认为服务就没有故障，或者在任何时候都可以获得服务。

这些问题不易解决，也不易实现自动化，所以，完全自动化的Web 服务和“ 即时 ”应用程序集成也许永远不可能实现。然而，当前的Web 服务技术使我们更接近于这种理想，并让部分过程自动化。

行业前景

现在有许多有竞争力的Web服务框架和建议，三个主要的竞争者是微软的.NET、IBM 的 Web Services 和 Sun 的 Open Net Environment（ONE）。虽然各个框架有其自身的侧重点，但是它们有着共同的基本Web服务定义和前景。而且，所有的框架都共用一套技术，主要是 SOAP、WSDL 和 UDDI。

本书的重点并不在于某个特定实现或框架，而是集中讨论共同的定义和技术。这就有可能使你透过销售欺诈，直接理解和评价当前的竞争者。

Web 服务体系结构

观察Web服务体系结构的方法有两种：查看每个Web服务参与者扮演的角色；查看新兴的 Web 服务协议栈。

Web 服务角色

在 Web 服务体系结构中共有三个主要的角色：

服务提供者
: Web 服务的提供者，其任务是实现服务并在 Internet 上提供这个服务。

服务请求者
: Web服务的消费者。Web 服务请求者通过打开一个网络连接并发送XML 请求来使用已有的 Web 服务。

服务注册中心
: 这是一个逻辑上集中式的服务目录。注册中心提供一个集中的地方，供开发者发布新服务或寻找已有服务。因此，它是公司及其服务的集中交换场所。

图 1-6 显示了主要的 Web 服务角色以及它们如何相互作用。

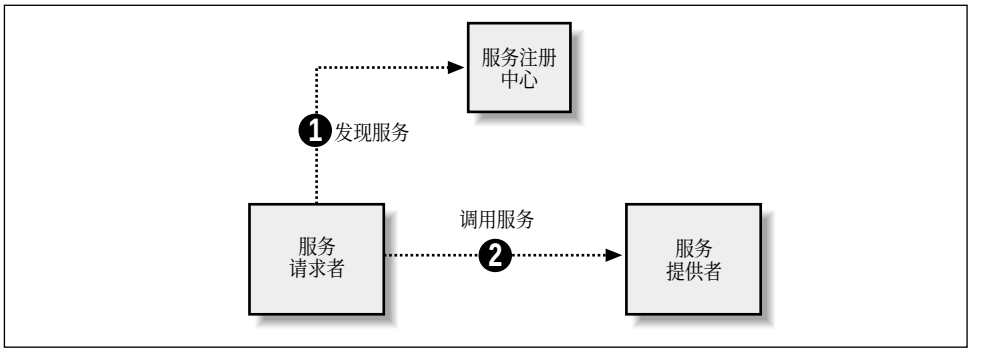

图 1-6：Web 服务角色

Web 服务协议栈

观察 Web 服务体系结构的第二种方法是查看新兴的 Web 服务协议栈。栈一直在不断发展，不过现在它由四个主要的层组成。下面是对每个层的简短描述：

服务传输

该层负责在应用程序间传输消息。目前，这一层包括超文本传输协议（HTTP），简单邮件传输协议（Simple Mail Transfer Protocol，SMTP），文件传输协议（FTP）和一些新的协议，如块可扩展交换协议（Blocks Extensible Exchange Protocol，BEEP）。

XML 消息接发

该层负责用普通的XML格式编码消息，使服务器和客户端都能理解消息。目前，这一层包括 XML-RPC 和 SOAP。

服务描述

该层负责描述Web服务的公共接口。目前，这一层通过WSDL（Web Service Description Language，Web 服务描述语言）处理。

服务发现

该层负责将服务集中到一个共同的注册中心，并提供容易使用的发布和查找功能。目前，这一层通过 UDDI（Universal Description, Discovery, and Integration，通用描述、发现和集成）处理。

随着 Web 服务的发展，可能会加入新的层，每个层也可能加入新的技术。图 1-7 概括了当前的 Web 服务协议栈。本书将对每一层进行详细的讲解。

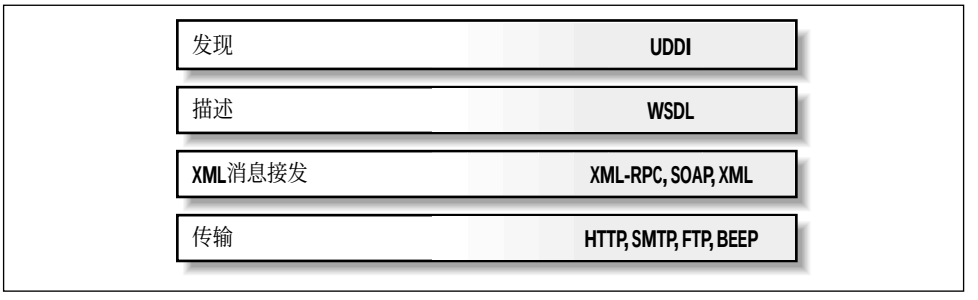

图 1-7：Web 服务协议栈

体系结构快照：IBM Web 服务浏览器

要想深入了解 Web 服务协议栈到底如何工作，请尝试使用 IBM 的 Web 服务浏览器（Web Services Browser）。利用这个浏览器，你可以搜索已有服务，查看服务描述，并自动调用这些服务，不必写任何代码就能看到Web服务协议栈中的每一层。

打开浏览器，访问 *http://demo.alphaworks.ibm.com/browser/*，你将会看到如图 1-8 所示的窗口。

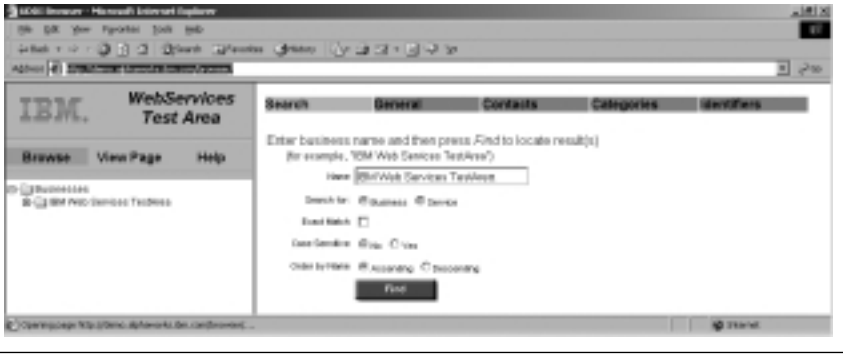

图 1-8：IBM 的 Web 服务浏览器

在左边的窗格中，你可以从集中式注册中心搜索已有的Web服务（注册中心其实使用的是UDDI，但我现在不做详细介绍）。在Search框中输入“IBM Web Services”，然后单击Search，IBM就会为你搜索集中式目录，并在左边窗格中显示相匹配的结果。选择最后一个名为IBM Web Services TestArea的文件夹，你将看到一个可用Web服务的列表（见图1-9）。

图1-9：Web服务搜索结果

单击GetWeatherService，右边的窗格将显示有关Web服务的特定细节（见图1-10），包括标明真正连接到服务的URL的绑定点（binding point），说明如何连接服务的服务描述文件（为WSDL文件，我在这里也不做详细介绍了）。

单击左边窗格中的View Page链接，右边的窗格将显示天气服务的简单用户界面，选择州和城市，IBM将自动调用服务，并显示当前相应的天气情况（见图1-11）。

如果你又选取了另外一个服务，这个服务就会被加到右边窗格的下面，如图1-12就同时显示了股票报价服务和天气服务。

图 1-10：IBM 天气服务的细节

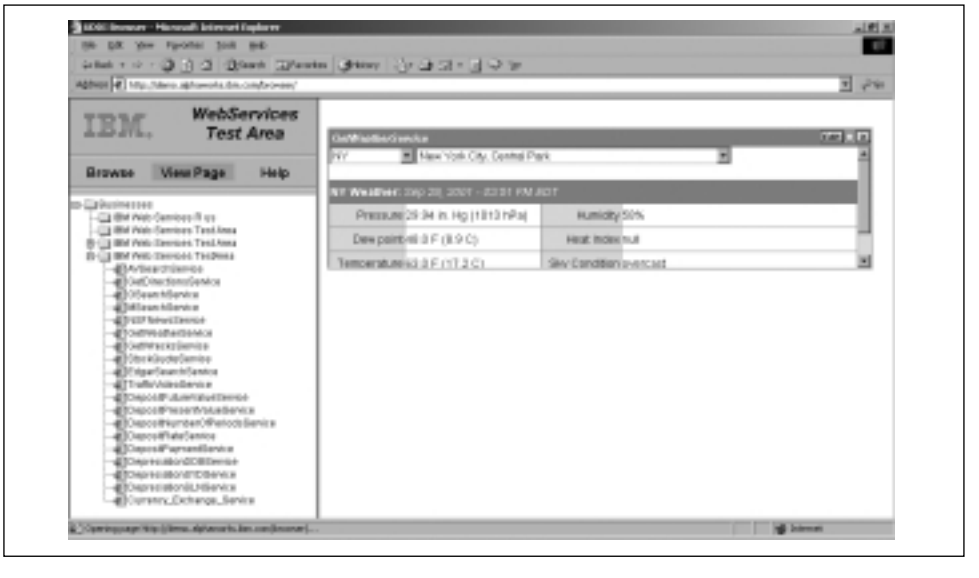

图 1-11：IBM 天气服务调用

图 1-12：一个页面上的多个 Web 服务

IBM 浏览器非常好地说明了工作状态下的 Web 服务，并强调了协议栈中的主要层。它还很好地展示了“即时”应用程序集成的潜能。每个服务基本上充当一个独立的构件，你可以在同一页继续堆叠更多的服务。这个方法最大的好处是，你不用写任何一行代码就能达到目的。

XML 消息接发

近年来，XML在计算科学领域蓬勃发展。因为它使不同的计算机系统能更容易地共享数据，而不必受操作系统和编程语言的限制，所以很快得到大家认可。有许多 XML 工具，包括供几乎各种操作系统和 Java、Perl、Python、C#、C、C++ 和 Ruby 等各种编程语言使用的解析器、编辑器。当开发者决定建立一个 Web 服务消息接发系统时，自然会选择 XML。XML 消息接发有两个主要的竞争者：XML-RPC 和 SOAP。下面的章节将分别描述这两个协议。

XML-RPC

XML-RPC 是一个用 XML 消息执行 RPC 的简单协议。服务请求使用 XML 来编

码，并通过HTTP POST发送；XML响应被嵌入HTTP响应主体。因为XML-RPC独立于平台，所以不同的应用程序之间可以通信，如Java客户端与Perl服务器之间可使用 XML-RPC。

要想深入了解 XML-RPC，请看一个简单的天气服务。服务要求输入一个邮政编码并返回该地区的当前温度。下面是天气服务的 XML-RPC 请求的例子（省略了 HTTP 首部）：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<methodCall>
   <methodName>weather.getWeather</methodName>
   <params>
      <param><value>10016</value></param>
   </params>
</methodCall>
```

服务请求由一个指定了方法名和所有方法参数的简单 `methodCall` 元素组成。

下面是来自天气服务的 XML-RPC 响应的例子：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<methodResponse>
   <params>
       <param>
          <value><int>65</int></value>
       </param>
   </params>
</methodResponse>
```

响应由一个指定了返回值的简单 `methodResponse` 元素组成。在这个例子中，返回值被指定为整型。

XML-RPC 是启动 Web 服务最容易的方法，在很多方面要比 SOAP 更简单易用。但是，不同于SOAP的是，XML-RPC没有相应的服务描述语法，这妨碍了XML-RPC 服务的自动调用 —— 实现“即时”应用程序集成的一个关键因素。第二章将详细介绍 XML-RPC。

SOAP

SOAP是在计算机之间交换信息的、基于XML的协议。虽然SOAP能用于各种消息接发系统，也可以通过各种传输协议进行传递，但是，SOAP主要是侧重于通过HTTP传输RPC。和XML-RPC一样，SOAP独立于平台，因此实现了不同应用程序间的通信。

为了更深入地理解SOAP，让我们再看一下简单的天气服务。下面是一个SOAP服务请求的例子（省略了HTTP首部）：

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://www.w3.org/2001/09/soap-envelope/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
      <ns1:getWeather
         xmlns:ns1="urn:examples:weatherservice"
         SOAP-ENV:encodingStyle="http://www.w3.org/2001/09/soap-encoding/">
         <zipcode xsi:type="xsd:string">10016</zipcode>
      </ns1:getWeather>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

正如你所看到的，SOAP请求只是比XML-RPC复杂一点。它利用了XML名称空间和XML模式（XML Schema）。但是，和在XML-RPC中一样，SOAP请求主体指定了一个方法名和一组参数。

下面是来自天气服务的一个SOAP响应的例子：

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://www.w3.org/2001/09/soap-envelope/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
      <ns1:getWeatherResponse
         xmlns:ns1="urn:examples:weatherservice"
         SOAP-ENV:encodingStyle="http://www.w3.org/2001/09/soap-encoding/">
         <return xsi:type="xsd:int">65</return>
      </ns1:getWeatherResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

响应给出单个整型返回值。第三章将会讨论 SOAP 的全部细节。

服务描述：WSDL

WSDL 目前代表 Web 服务协议栈中的服务描述层。简而言之，WSDL 是为 Web 服务指定公共接口的 XML 语法。公共接口包括所有共用功能的信息、所有 XML 消息的数据类型信息、所用特定传输协议的绑定信息和定位特定服务的地址信息。WSDL 不一定要连接到特定的 XML 消息接发系统，但它确实内置了用来描述 SOAP 服务的扩展名。

例 1-1 为一个 WSDL 文件的例子，描述了我们前面查看的天气服务的公共接口。显然，浏览这个例子时，我们要考虑许多细节。但我们现在集中考虑两点。首先，`message` 元素指定了在计算机之间传递的每个 XML 消息。在本例中为 `getWeatherRequest` 和 `getWeatherResponse`。其次，`service` 元素规定可以通过 SOAP 从 *http://localhost:8080/soap/servlet/rpcrouter* 获得服务。

例 1-1：WeatherService.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="WeatherService"
   targetNamespace="http://www.ecerami.com/wsdl/WeatherService.wsdl"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://www.ecerami.com/wsdl/WeatherService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <message name="getWeatherRequest">
      <part name="zipcode" type="xsd:string"/>
   </message>
   <message name="getWeatherResponse">
      <part name="temperature" type="xsd:int"/>
   </message>

   <portType name="Weather_PortType">
      <operation name="getWeather">
         <input message="tns:getWeatherRequest"/>
         <output message="tns:getWeatherResponse"/>
      </operation>
   </portType>
```

```
   <binding name="Weather_Binding" type="tns:Weather_PortType">
      <soap:binding style="rpc"
         transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="getWeather">
         <soap:operation soapAction=""/>
         <input>
            <soap:body
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               namespace="urn:examples:weatherservice"
               use="encoded"/>
         </input>
         <output>
            <soap:body
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               namespace="urn:examples:weatherservice"
               use="encoded"/>
         </output>
      </operation>
   </binding>

   <service name="Weather_Service">
      <documentation>WSDL File for Weather Service</documentation>
      <port binding="tns:Weather_Binding" name="Weather_Port">
         <soap:address
            location="http://localhost:8080/soap/servlet/rpcrouter"/>
      </port>
   </service>
</definitions>
```

客户端可以用 WSDL 定位 Web 服务，并调用任何有效的公共功能。通过使用可识别WSDL的工具，这个过程可以达到完全自动化，使得应用程序只需写少量代码或完全不写代码就可以轻易地集成新服务。譬如，IBM 新近发布了 IBM Web 服务调用框架（Web Services Invocation Framework，WSIF）。你可以用 WSIF 指定*WeatherService.wsdl*文件，并自动调用所描述的服务。例如，下面的命令行：

```
java clients.DynamicInvoker http://localhost:8080/wsdl/WeatherService.wsdl
    getWeather 10016
```

将产生如下输出：

```
Reading WSDL document from 'http://localhost:8080/wsdl/WeatherService.wsdl'
Preparing WSIF dynamic invocation
Executing operation getWeather
Result:
```

```
temperature=65

Done!
```

第六章将介绍 WSDL 和 WSDL 调用工具。

服务发现：UDDI

UDDI 当前代表 Web 服务协议栈中的服务发现层。UDDI 由微软、IBM 和 Ariba 提出，表示发布和查找业务和 Web 服务的技术规范。

UDDI 的核心由两部分组成。首先，UDDI 是建立分布式业务和 Web 服务目录的技术规范。数据以特定的 XML 格式存储。UDDI 规范包括搜索已有数据和发布新数据的 API。其次，UDDI 业务注册中心（UDDI Business Registry）是对 UDDI 规范的完全实现。微软和 IBM 于 2001 年 5 月建立了 UDDI 业务注册中心，现在任何人都可以用它搜索现有的 UDDI 数据，任何公司都可以注册自己及其服务。

UDDI 中的数据主要分为三类：

白页（white page）
: 该类包括某公司的一般信息，例如业务名称、业务描述和联系信息等。

黄页（yellow page）
: 该类包括公司或其所提供服务的一般分类数据，如行业、产品或根据标准分类法确定的地域代码。

绿页（green page）
: 该类包括有关 Web 服务的技术信息（指向外部规范的指针和调用 Web 服务的地址）。

图 1-13 为微软 UDDI 网站的一个快照，你可以在此站点发布自己的服务或搜索已有服务。

在第七章可以找到关于 UDDI 的全部细节。

图 1-13：微软 UDDI 站点：搜索 XMethods 公司

服务传输

Web 服务协议栈的底层是服务传输。这一层负责在两台计算机间真正传输 XML 消息。

HTTP

HTTP 是当前服务传输最流行的方法，它简单、稳定、使用广泛。另外，大部分防火墙都允许 HTTP 通过。也允许 XML-RPC 或 SOAP 消息伪装成 HTTP 消息。如果你想方便地集成远程应用程序，这当然是很好的选择，但它也会引起若干安全问题（见本章后面的“安全性考虑”一节）。

虽然 HTTP 能达到这个目的，但有些评论家还是认为 HTTP 并非 Web 服务的理想方法。具体来说，HTTP 最初是为远程文档检索而设计的，现在又用于支持 RPC，而 RPC 要求比文档检索更高的效率和可靠性。

注意：有些开发者认为，HTTP作为消息接发的基础已经足够了，而HTTP以上的层既是一个解决方案，也是一个麻烦。这些问题被称为代表状态传输（Representational State Transfer，REST）。有关这方面的问题，请访问*http://internet.conveyor.com/RESTwiki/moin.cgi*。

BEEP

最有可能取代HTTP的是BEEP（Blocks Extensible Exchange Protocol，块扩展交换协议），它是建立新协议的最优方法的一个新的IETF框架。具体地讲，BEEP层直接置于TCP之上，包括内置的信息交换协议、身份验证、安全和错误处理等若干功能。用BEEP可以为即时消息接发、文件传输、内容联合和网络管理等各种应用程序创建新协议。

SOAP不受特定传输协议的约束，你可以通过HTTP、SMTP或FTP使用SOAP。一个最有前途的想法是通过BEEP使用SOAP，与HTTP相比，这样做在性能上有几点优势。虽然BEEP需要启动信息交换，但在此之后，协议只需为每个消息分配30个字节，这使它比HTTP更有效率（注1）。另外，BEEP支持一个连接的多个数据通道，从而进一步提高了效率。

通过BEEP使用SOAP的最新建议可以从*http://beepcore.org/beepcore/docs/beep-soap.jsp* 找到。

注意：另一个可能取代HTTP的是HTTP-R（Reliable HTTP，可靠的HTTP）。IBM正在开发HTTP-R，并计划提交给Internet IETF（Internet Engineering Task Force，Internet工程任务组）。HTTP-R通过增强HTTP来保证消息的可靠性。例如，HTTP-R保证消息只传送一次，不能传送时产生报告。这对电子商务服务非常关键，如电子订购系统和商品目录管理。你可以从*http://www-106.ibm.com/developerworks/webservices/library/ws-phtt/*获得由IBM提供的HTTP-R入门资料。

注1：每个HTTP消息的开销取决于许多因素，包括所请求的URL，所使用的客户端类型，以及与HTTP响应一同返回的服务器信息类型。因此，对于每个消息，典型的浏览器和SOAP请求的开销范围约为100~300字节。

安全性考虑

Web服务的安全性至关重要。但是，XML-RPC和SOAP规范都不能提供明确的安全性或身份验证要求。而且，虽然Web服务社区提出了大量安全框架和协议，但是还没有在综合安全包上达成一致。

大致存在三种特殊的安全问题：机密性、身份验证和网络安全。

机密性

当客户端向服务器发送了一个XML请求之后，我们能保证通信的机密性吗？

幸好XML-RPC和SOAP都主要在HTTP的顶部运行，这样，XML通信就可以通过SSL（Secure Sockets Layer，安全套接字层）加密。SSL是经过了验证的技术，并被广泛使用，因此是加密消息非常可行的选择。

但是，Web服务的一个关键要素是单个Web服务可能由一系列应用程序组成。譬如，一个大型服务可能将其他三个应用程序的服务连接到一起。在这种情况下，在服务路径的每个节点上消息都需要加密，每个节点都可能是一个链上潜在的弱链接，这样，SSL就显得力不从心了。目前还没有公认的解决方案。但是，W3C XML加密标准可能是一个解决方案。这一标准为加密、解密整个或部分XML文档提供了一个框架，并且有可能得到广泛的行业支持。有关XML加密标准的信息可以从 *http://www.w3.org/Encryption/* 获得。

身份验证

当客户端连接到Web服务上时，我们如何鉴别用户呢？用户有权使用该服务吗？

一个解决方案是使用HTTP身份验证。HTTP内置有Basic和Digest身份验证支持，可以像现在保护HTML文档一样保护服务。但是，大部分安全专家都认为HTTP身份验证是一个相对较差的选择。

和加密一样，关于强大的身份验证方案还没有明确的一致意见，但有几个框架正在考虑之中。第一个是SOAP-DISG（SOAP Security Extensions: Digital Signature，SOAP安全扩展：数字签名）。DSIG使用公共密钥密码技术对SOAP消息进行数字签名，这使得客户端或服务器能够验证另一方的身份。DSIG已被提交到W3C，你可以从*http://www.w3.org/TR/SOAP-dsig/*获得。其次，OASIS（Organization for the Advancement of Structured Information Standards, 结构化信息标准推进组织）正在研究SAML（Security Assertion Markup Language，安全声明标记语言）。SAML是为业务伙伴间交换身份验证和授权信息而设计的。你可以从*http://www.oasis-open.org/committees/security/*在线获取相关信息。

在有关方面的努力下，几个公司已经推出XKMS（XML Key Management Services，XML密钥管理服务）。XKMS为分配和管理公共密钥和证书定义了一系列的服务。该协议本身建立在SOAP和WSDL的基础上，因此是一个非常好的Web服务例子，你可以从*http://www.w3.org/ TR/xkms/*在线获取XKMS规范。

网络安全

2000年6月，著名计算机专家Bruce Schneier坦率地提出“SOAP将为安全脆弱性开辟一条全新的大道”。Schneier的基本观点是HTTP是为文档检索而设计的，通过SOAP扩展HTTP将使远程客户端能够调用防火墙显式禁止的命令和过程。

你可能会说，CGI应用程序和servlet也会引发同样的安全脆弱性问题，毕竟它们也能使远程应用程序调用命令和过程。然而，当SOAP被越来越被广泛地使用时，Schneier的观点也就变得越来越具有说服力。现在还没有解决这个问题的简便方法，仍然存在许多争论。如果你现在真想过滤掉SOAP或XML-RPC消息，一个可能的方法是过滤掉所有的HTTP POST请求，这些请求都将其内容设定为`text/xml`格式（两种规范都有此要求）。另一个方法是过滤掉`SOAPAction` HTTP首部属性（请参阅第三章）。防火墙厂商也正在开发完全用于过滤Web服务通信的工具。

总体观察

在了解了Web服务协议栈的每一层之后，下一个重要的步骤是了解这几个层是如何形成一个整体的。我们可以从服务请求者或服务提供者两种角度来讨论。本节将从这两个方面加以分析，并着眼于每一方面的典型开发计划。

从服务请求者的角度

服务请求者是 Web 服务的消费者。下面是 Web 服务请求者的典型开发计划：

1. 首先，你必须找到与你的应用程序相关的服务。这一步一般要搜索合作伙伴和服务的 UDDI 业务目录（UDDI Business Directory）。
2. 一旦找到所需要的服务，下一步就要找出服务描述。如果是一个SOAP服务，你就可能发现一个WSDL文档；如果是一个XML-RPC服务，你就可能找到可阅读的集成指导。
3. 第三，你必须创建一个客户端应用程序。譬如，你可能会用你选择的语言创建一个 XML-RPC 或 SOAP 客户端。如果服务有一个WSDL 文件，那么你还可以用 WSDL 调用工具自动创建客户端代码。
4. 最后，运行你的客户端应用程序，真正调用 Web 服务。

图 1-14 是从 Web 服务请求者角度的快照。

图 1-14：开发 Web 服务：服务请求者角度

从服务提供者的角度

服务提供者提供一个或多个Web服务。下面是一个典型的服务提供者开发计划：

1. 首先，你必须开发服务的核心功能。这往往是最艰难的部分，因为你的应用程序可能会连接到数据库、Enterprise JavaBeans™（EJB）、COM对象或继承应用程序。
2. 其次，你必须开发你的核心功能的服务包装，可以是XML-RPC服务包装，也可以是SOAP服务包装。这一般相对比较简单，因为你仅仅是让已有的功能包括在一个更大的框架中。
3. 下一步，你应该提供一个服务描述。如果你正在创建SOAP应用程序，你就应该创建一个WSDL文件；如果你正在创建XML-RPC服务，你就应该考虑创建一些可阅读的指导。
4. 接着，你需要部署服务。根据需要，你可以安装和运行独立的服务器或集成现有的Web服务器。
5. 最后，你需要发布你的新服务及其规范。一般要将数据发布到全局UDDI目录或为你所在公司指定的私人UDDI目录。

图1-15提供了从服务提供者角度的快照。

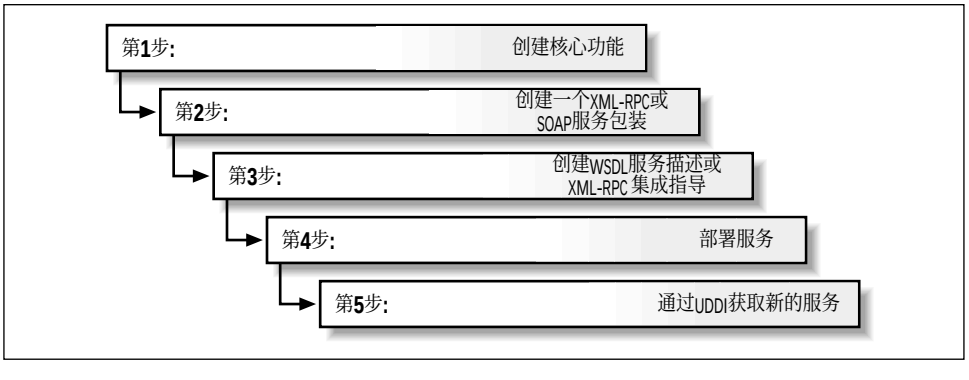

图1-15：开发Web服务：服务提供者角度

标准和一致

Web服务还处于发展初期，但它们势必会引起分布式应用程序开发领域的一场革命。但是，Web服务要取得长期成功，一个关键因素是标准化和保持这些标准的一致性。目前，本书中描述的Web服务技术都不能与W3C或IETF保持一致。尽管SOAP和WSDL已被提交到W3C，但它们还没有成为正式的推荐标准；XML-RPC没有提交到任何标准化组织；UDDI目前属于行业合作的范围，要几经反复才能被提交到标准化组织。

2000年9月，W3C成立了XML协议小组，这标志着W3C第一次正式进军Web服务领域。它的首要任务是为SOAP建立正式的推荐标准，并且目前已经最终确定了SOAP 1.2规范。2002年1月，W3C将XML协议小组纳入更为一般化的Web Services Activity，其中增加了Web服务体系结构工作组和Web服务描述工作组。

注意：有关W3C Web Services Activity的信息可以从 *http://www.w3.org/2002/ws/* 获得。

大多数进入Web服务的新手一开始都会被众多的建议标准和标准间复杂的关系吓倒。标准化Web服务协议栈的每一层将是一个挑战，而保证各层形成一个整体并让开发者有一致的感觉将是更重大的挑战。

第二部分

XML-RPC

第二章

XML-RPC 精髓

XML-RPC为通过网络的方法或函数调用提供了基于XML和HTTP的机制。XML-RPC 为连接不同的系统和发布机器可读的信息提供了一组简单但非常有用的工具。本章将对 XML-RPC 做全面的综述，包括以下几个方面的内容：

- 介绍主要概念和 XML-RPC 的历史；
- 通过了解 XML-RPC 在代码粘贴和信息发布方面的作用来探索其应用方案；
- 对 XML-RPC 做技术概览，包括对 XML-RPC 的数据类型、请求和响应的详细说明；
- 举一个用 XML-RPC 连接 Java 和 Perl 程序的例子。

XML-RPC 概览

XML-RPC允许程序通过网络调用函数或过程。它只用少量的XML词汇描述请求和响应的属性，用 HTTP 协议将信息从客户端传递给服务器。客户端在 XML 请求中指定过程名和参数，服务器在XML 响应中返回一个错误或响应。XML-RPC 参数是一组简单的类型和内容，最复杂的类型是结构和数组。XML-RPC 没有对象的概念，对由其他 XML 词汇组成的信息，它也不提供包容机制。尽管存在这些限制，XML-RPC 仍能胜任多种任务。

XML-RPC诞生于1998年初，由UserLand Software公司推出，并首次用于它们的 Frontier产品中。此后XML-RPC基本保持稳定（注1）。你可以从*http://www.xmlrpc.com/spec*获得其详细说明，还可从*http://www.xmlrpc.com/directory/1568/implementations*获得一个实现列表（在编写本书时其中有55个实现，并有各种语言的版本）。

为什么选择XML-RPC？

在这样一个好像被对象占据的编程领域里，XML-RPC对许多应用程序来说似乎太受限制了。尽管如此，当开发人员需要集成不同类型的系统时，XML-RPC固有的简洁性就使其优势显现出来。XML-RPC对数据类型的选择自由度相对较小，但对那些想用各种编程语言都能使用的格式来表达信息的开发者来说，XML-RPC提供的自由空间足够了。

XML-RPC主要用于两个有时会相互重叠的领域。建立分布式系统的系统集成员和程序员经常用XML-RPC作为粘贴码来连接内部网络中的不同部分。使用XML-RPC，开发人员就可把精力用于考虑系统接口，而不是考虑连接那些接口的协议上。建立公共服务的开发者也可以使用XML-RPC，他们可以用它定义接口，还可以用他们选择的语言实现它。一旦这种服务在Web上发布，任何具有XML-RPC能力的客户端都可连上它，开发者还可以用它建立自己的应用程序。

场景1：用XML-RPC粘贴代码

当分布式系统（有意或无意间）变得越来越普遍时，开发者不得不越来越频繁地提出集成的问题。当一些组织想把它们的信息管理合理化并减少重复时，原来运行自身界面的系统就必须与其他系统一起工作了。这通常意味着Unix系统需要与Windows系统对话，Windows需要与Linux对话，而Linux又要与主机对话。为了让不同的系统能相互对话，众多程序员花费大量的时间来建立定制协议和格式。

注1： 要了解有关XML-RPC早期历史的信息，可浏览*http://davenet.userland.com/1999/01/29/ microsoftXmlRpc*。其中介绍了UserLand和微软公司所起的作用。“我们正在与微软研究的规范快照”成为XML-RPC，而规范的其余内容演变成SOAP。

程序开发者可以用 XML-RPC 来连接在不同系统和环境下运行的程序，而不是建立那种需要大量测试、存档和调试的定制系统。用这种方法，开发者可以使用已有的 API，并在必要时向那些 API 增加连接。有些问题用一个过程就可以解决，有些问题则需要更复杂的交互，但是整个方法很像开发另一组接口。在粘贴代码的情况下，客户端和服务器的区别不是很明显，客户端和服务器这两个词只分别代表发送请求和响应的程序。如果允许程序在接收和发送请求时都使用 XML-RPC，那么同一程序可能既有客户端实现，又有服务器实现。

场景 2：用 XML-RPC 发布服务

XML-RPC为信息提供了一个对机器可读的接口，可以用于向Web世界发布信息。XML-RPC 这种作用的基本结构非常像传统的向人进行的Web发布那样，带有基本相同的安全和体系结构问题，但是其信息接受者可以是能理解 XML-RPC 接口的任何客户端。在 Web 发布时，XML-RPC 发布意味着开发者控制了服务器，但不一定控制了客户端。

譬如，O'Reilly Network的Meerkat标题行组件不仅提供了一个可阅读的接口（在 *http://meerkat.oreillynet.com*），还提供了一个面向 Web 世界的 XML-RPC 接口（文档在 *http://www.oreillynet.com/pub/a/rss/2000/11/14/meerkat_xmlrpc.html*）。普通读者可以用这个基于表单的接口查询标题，而需要用其他形式表现标题信息的开发者也可以使用 XML-RPC。这样做可以很容易地将内容与外观呈现分离，而仍然可以在以 Web 为中心的环境中运行。

XML-RPC 技术概览

XML-RPC 由三个较小的部分组成：

XML-RPC 数据模型
: 一组用于传送参数、返回值和故障（错误信息）的数据类型

XML-RPC 请求结构
: 一个包含方法和参数信息的 HTTP POST 请求

XML-RPC 响应结构

一个包含返回值或故障信息的 HTTP 响应

请求和响应结构都使用这些数据结构。三者结合就定义了一个完整的远程过程调用（Remote Procedure Call）。

注意：不了解本章后面介绍的标记细节就使用XML-RPC是完全可能的。但是，即使你不想触及这些细节，为了理解通过网络传送的信息的性质，你也应该阅读下面几节。

XML-RPC 数据模型

XML-RPC规范定义了六种基本数据类型和两种表现为类型组合的复合数据类型。虽然这是比许多编程语言所提供的类型更严格的一组数据类型，但它足以表示多种信息，而且看起来已达到多种程序－程序通信的最低平均标准。

所有的基本类型都用简单的 XML 元素来表示，元素的内容就是它的值。如定义一个字符串，其值为“ Hello World! ”你只要写：

```
<string>Hello World!</string>
```

表 2-1 列出了 XML-RPC 的基本数据类型。

表 2-1：XML-RPC 的基本数据类型

类型	值	例子
`int` 或 `i4`	32 位整数，取值在 −2 147 483 648 和 2 147 483 647 之间	`<int>27</int>` `<i4>27</i4>`
`double`	64 位浮点数	`<double>27.31415</double>` `<double>-1.1465</double>`
`Boolean`	true (1)或 false(0)	`<boolean>1</boolean>` `<boolean>0</boolean>`

表 2-1：XML-RPC 的基本数据类型（续）

类型	值	例子
`string`	ASCII 文本，不过许多实现支持 Unicode	`<string>Hello</string>` `<string>bonkers! @</string>`
`dateTime.iso8601`	ISO8601 格式的日期：*CCYYMMDD*T*HH:MM:SS*	`<dateTime.iso8601>20021125T02:20:04 </dateTime.iso8601>` `<dateTime.iso8601>20020104T17:27:30 </dateTime.iso8601>`
`base64`	以 Base 64 编码的二进制信息，与在 RFC 2045 中定义的相同	`<base64>SGVsbG8sIFdvcmxkIQ== </base64>`

注意：要想了解Base 64编码的更多信息，请参阅RFC 2045的第6.8节“ Multipurpose Internet Mail Extensions（MIME）Part One: Format of Internet Message Bodies ”，你可以从 *http://www.ietf.org/rfc/rfc2045.txt* 获得这个文档。Base 64并不是一种有效的编码格式，但它确实简化了将二进制信息加入到 XML 文档的过程。最好少用它。

基本数据类型总是包含在`value`元素中，`value`元素可包含字符串（也只有字符串），但不包含`string`元素。基本类型可以组合成两个更复杂的数据类型：数组和结构。数组表示的是有序信息，而结构表示名称 / 值对，很像散列表、关联数组或属性。

`array`元素表示数组，包含一个有一组值的`data`元素。和其他数据类型一样，`array`元素必须包含在`value`元素中。例如，下面的数组包含四个字符串：

```
<value>
  <array>
    <data>
      <value><string>This </string></value>
      <value><string>is </string></value>
      <value><string>an </string></value>
      <value><string>array.</string></value>
    </data>
  </array>
</value>
```

下面的数组包含四个整数：

```
<value>
  <array>
    <data>
      <value><int>7</int></value>
      <value><int>1247</int></value>
      <value><int>-91</int></value>
      <value><int>42</int></value>
    </data>
  </array>
</value>
```

数组还可以包含不同类型的值，如下面的数组：

```
<value>
  <array>
    <data>
      <value><boolean>1</boolean></value>
      <value><string>Chaotic collection, eh?</string></value>
      <value><int>-91</int></value>
      <value><double>42.14159265</double></value>
    </data>
  </array>
</value>
```

创建一个多维数组非常简单，只需在数组内加入数组：

```
<value>
  <array>
    <data>
      <value>
        <array>
          <data>
            <value><int>10</int></value>
            <value><int>20</int></value>
            <value><int>30</int></value>
          </data>
        </array>
      </value>
      <value>
        <array>
          <data>
            <value><int>15</int></value>
            <value><int>25</int></value>
            <value><int>35</int></value>
```

```
            </data>
          </array>
        </value>
      </data>
    </array>
  </value>
```

虽然有大量标记，但是 XML-RPC 开发者并不需要直接处理这些标记。

警告：XML-RPC 无法保证数组的值有一致的数量或类型。如果你的应用程序要求一致性，那么你需要确保所编写的代码可以始终如一地产生数量和类型都正确的输出值。

结构含有以名称标识的无序内容。名称是字符串，不过你不必将它们包含在 `string` 元素中。每个 `struct` 元素含有一组 `member` 元素，而每个 `member` 元素含有一个 `name` 元素和一个 `value` 元素。成员的顺序并不重要。虽然规范不要求名称惟一，但为了一致性，你要保证它们是惟一的。一个简单的结构应像这样：

```
<value>
  <struct>
    <member>
      <name>givenName</name>
      <value><string>Joseph</string></value>
    </member>
    <member>
      <name>familyName</name>
      <value><string>DiNardo</string></value>
    </member>
    <member>
      <name>age</name>
      <value><int>27</int></value>
    </member>
  </struct>
</value>
```

结构还可嵌套其他结构，甚至数组。如下面的结构包含一个字符串、一个结构和一个数组：

```
<value>
  <struct>
    <member>
      <name>name</name>
      <value><string>a</string></value>
```

```
        </member>
        <member>
          <name>attributes</name>
          <value><struct>
            <member>
              <name>href</name>
              <value><string>http://example.com</string></value>
            </member>
            <member>
              <name>target</name>
              <value><string>_top</string></value>
            </member>
        </struct></value>
        </member>
        <member>
          <name>contents</name>
          <value><array>
            <data>
              <value><string>This </string></value>
              <value><string>is </string></value>
              <value><string>an example.</string></value>
            </data>
          </array></value>
        </member>
      </struct>
    </value>
```

数组也可以包含结构。有时还可以用这些复杂类型表示对象结构，但有时你会发现用 SOAP 做这种类型的复杂转化更容易。

XML-RPC 请求结构

XML-RPC 请求是 XML 内容和 HTTP 首部相结合的产物。XML 内容使用数据输入结构传送参数，并包含用来标识将要调用过程的附加信息，而 HTTP 首部为通过 Web 传送请求提供包装。

每个请求都只包含一个 XML 文档，其根元素是 `methodCall` 元素。每个 `methodCall` 元素包含一个 `methodName` 元素和一个 `params` 元素。`methodName` 元素标识将调用的过程名，而 `params` 元素包含一组参数及相应的值。每个 `params` 元素包含一组 `param` 元素，而每个 `param` 元素又包含一个 `value` 元素。

譬如，要向一个名为`circleArea`的方法传送带有一个`Double`参数（代表半径）的请求时，XML-RPC 请求应该是这样的：

```
<?xml version="1.0"?>
<methodCall>
  <methodName>circleArea</methodName>
  <params>
    <param>
      <value><double>2.41</double></value>
    </param>
  </params>
</methodCall>
```

要向`sortArray`过程传送一组数组，请求应是这样的：

```
<?xml version="1.0"?>
<methodCall>
  <methodName>sortArray</methodName>
  <params>
    <param>
      <value>
        <array>
          <data>
            <value><int>10</int></value>
            <value><int>20</int></value>
            <value><int>30</int></value>
          </data>
        </array>
      </value>
    </param>
    <param>
      <value>
        <array>
          <data>
            <value><string>A</string></value>
            <value><string>C</string></value>
            <value><string>B</string></value>
          </data>
        </array>
      </value>
    </param>
  </params>
</methodCall>
```

这些请求的 HTTP 首部将反映出发送者和内容，基本模板是这样的：

```
POST /target HTTP/1.0
User-Agent: Identifier
Host: host.making.request
Content-Type: text/xml
Content-Length: length of request in bytes
```

斜体字信息根据不同的客户或请求可以发生改变。譬如，如果从正在监听*/xmlrpc*的 XML-RPC 服务器可以获得 `circleArea` 方法，那么请求应是这样的：

```
POST /xmlrpc HTTP/1.0
User-Agent: myXMLRPCClient/1.0
Host: 192.168.124.2
Content-Type: text/xml
Content-Length: 169
```

组合到一起，则整个请求如下所示：

```
POST /xmlrpc HTTP/1.0
User-Agent: myXMLRPCClient/1.0
Host: 192.168.124.2
Content-Type: text/xml
Content-Length: 169

<?xml version="1.0"?>
<methodCall>
  <methodName>circleArea</methodName>
  <params>
    <param>
      <value><double>2.41</double></value>
    </param>
  </params>
</methodCall>
```

这是带有精心建造的有效负载的普通 HTTP 请求。

注意：`User-Agent` 首部通常会反映出用于组合请求的 XML-RPC 库，而不是用于调用的特定程序。这与浏览器有一点不同，在浏览器世界中，使用 `User-Agent` 首部的“浏览器探查法”将标明产生请求的特定程序，如 Linux 的 Opera 6.0。

XML-RPC 响应结构

响应和请求非常相似，只是有一些额外的改变。如果响应成功 —— 即找到并正确执行了过程，而且返回了结果，那么 XML-RPC 响应就非常像请求，只不过 `methodResponse` 元素将取代 `methodCall` 元素，并且没有 `methodName` 元素：

```
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><double>18.24668429131</double></value>
    </param>
  </params>
</methodResponse>
```

无论是否使用了封闭的 `params` 元素，XML-RPC 响应都只包含一个参数。这个参数可以是数组或结构，因此可能会返回多个值。即使你的方法未被设计成有返回值（如 C、C++ 或 Java 中的 `void` 方法），但它还是会返回一些东西。避开这个限制的典型方法是返回一个“ 成功值 ” —— 可以是一个设为 true（1）的布尔值。

如果处理 XML-RPC 请求时出现了问题，`methodResponse` 元素就会包含一个 `fault` 元素，而不是一个 `params` 元素。和 `params` 一样，`fault` 元素只有一个值。那个值只是用来标明有错，而不是包含对请求的响应。一个故障响应可以是这样的：

```
<?xml version="1.0"?>
<methodResponse>
  <fault>
    <value><string>No such method!</string></value>
  </fault>
</methodResponse>
```

这个响应也可以是这样的：

```
<?xml version="1.0"?>
<methodResponse>
  <fault>
    <value>
      <struct>
        <member>
```

```
            <name>code</name>
            <value><int>26</int>
          </member>
          <member>
            <name>message</name>
            <value><string>No such method!</string></value>
          </member>
        </struct>
      </value>
    </fault>
</methodResponse>
```

XML-RPC 并没有标准化错误代码，因此要想知道特定的程序包如何处理故障，你需要查看其文档。

与请求一样，响应被封装在 HTTP 中，并有 HTTP 首部。无论消息中是否含有故障信息，所有的 XML-RPC 响应都使用 `200 OK` 响应代码。首部使用与请求首部相似的普通结构，一个典型的首部如下所示：

```
HTTP/1.1 200 OK
Date: Sat, 06 Oct 2001 23:20:04 GMT
Server: Apache.1.3.12 (Unix)
Connection: close
Content-Type: text/xml
Content-Length: 124
```

XML-RPC只需HTTP 1.0支持，但与HTTP 1.1兼容。`Server`首部表明处理XML-RPC 实现请求的 Web 服务器的类型。首部有时会反映出处理某个请求的 XML-RPC 服务器实现，有时也不反映。`Content-Type` 一定要设为 `text/xml`，`Content-Length`首部指定了响应的长度（以字节为单位）。带有首部和响应有效负载的完整响应可以是这样的：

```
HTTP/1.1 200 OK
Date: Sat, 06 Oct 2001 23:20:04 GMT
Server: Apache.1.3.12 (Unix)
Connection: close
Content-Type: text/xml
Content-Length: 124

<?xml version="1.0"?>
<methodResponse>
```

```
  <params>
    <param>
      <value><double>18.24668429131</double></value>
    </param>
  </params>
</methodResponse>
```

在响应从 XML-RPC 服务器传送到 XML-RPC 客户端之后，连接就会关闭。接下来的请求需要作为独立的连接来发送。

用 XML-RPC 开发服务

在应用程序中使用 XML-RPC 一般需要加入一个 XML-RPC 库，并通过这个库调用函数。创建一个能顺利处理 XML-RPC 请求的函数，只能用 XML-RPC 支持的基本类型编写代码。很少需要改变代码编写风格。添加 XML-RPC 支持需要编写把代码和库连接起来的包装代码，这一般没什么困难。

注意：随着 XML-RPC 变得越来越普及，一些环境正在构建 XML-RPC。当 Perl 和 Python 社团正在讨论相似的集成问题时，UserLand Frontier 已经做了多年了。

为了说明XMP-RPC, 我们将建立一个使用Java处理XML-RPC消息, 并使用Java、Perl 调用其过程的服务器。虽然这个例子很简单，但它说明了使用 XML-RPC 时在程序之间建立通信所必需的连接。

对话的 Java 端使用 Apache XML 项目的 Apache XML-RPC，可以从 *http://xml.apache.org/xmlrpc/* 中得到它。Apache 程序包包括一些使 XML-RPC 与 Java 更易集成的关键项：

- 向 XML-RPC 服务器添加方法的自动注册过程。
- 可以降低建立 servlet 的必要性，只使用 XML-RPC 的内置服务器。
- 使调用远程方法变得非常简单的客户端程序包。

这个例子将使用一个用内置的Apache程序包服务器注册的过程以及用来测试这个过程的客户端。

注意：要想进一步了解数据类型的细节以及为 XML-RPC 处理建立 servlet 的信息等有关 Apache XML-RPC 程序包的信息，请参阅 Simon St.Laurent、Edd Dumbill 和 Joe Johnston 编著的《Programming Web Services with XML-RPC》（由 O'Reilly 出版）的第三章，可从*http://www.oreilly.com/catalog/ progxmlrpc /chapter/ch03.html*在线获得。

我们要测试的过程将会返回圆的面积，它是在一个名为`AreaHandler`的类中定义的，如例 2-1 所示。

例 2-1：一个简单的 Java 过程

```
package com.ecerami.xmlrpc;

public class AreaHandler {

     public double circleArea(double radius) {
          double value=(radius*radius*Math.PI);
          return value;
     }

}
```

`AreaHandler`类的`circleArea`方法用一个`double`值来表示半径，并返回一个`double`值以表示相应的面积。`AreaHandler`类中没有一点 XML-RPC 特有的东西。

通过 XML-RPC 使用`circleArea`方法需要两个步骤：一是该方法必须用 XML-RPC 程序包注册，二是某种服务器必须使这个程序包可以通过 HTTP 进行访问。例 2-2 中的`AreaServer`类完成了这两个步骤。

例 2-2：建立一个 Java XML-RPC 服务器

```
package com.ecerami.xmlrpc;

import java.io.IOException;
import org.apache.xmlrpc.WebServer;
import org.apache.xmlrpc.XmlRpc;
```

```
public class AreaServer {

     public static void main(String[] args) {
       if (args.length < 1) {
          System.out.println("Usage: java AreaServer [port]");
          System.exit(-1);
         }

       try {
         startServer(args);
       } catch (IOException e) {
           System.out.println("Could not start server: " +
               e.getMessage());
       }
     }

     public static void startServer(String[] args) throws IOException {
     // 启动服务器，使用内置版本
     System.out.println("Attempting to start XML-RPC Server...");
     WebServer server = new WebServer(Integer.parseInt(args[0]));

     System.out.println("Started successfully.");

     // 注册AreaHandler类为area
     server.addHandler("area", new AreaHandler());
     System.out.println("Registered AreaHandler class to area.");

     System.out.println("Now accepting requests. (Halt program to stop.)");

     }
}
```

`main`方法检测到指定服务器运行端口的一个命令行参数，并把这个信息传送给启动内置服务器的`startServer`。一旦服务器启动（被创建时就开始运行），程序就调用`addHandler`方法注册一个名为`area`的`AreaHandler`类的实例。类`org.apache.xmlrpc.XmlRpc`处理所有的方法签名细节，这使得用两行关键代码来启动一个XML-RPC服务成为可能。要启动服务器，只需从命令行执行`com.ecerami.xmlrpc.AreaServer`，指定一个端口。

```
C:\ora\xmlrpc\java>java com.ecerami.xmlrpc.AreaServer 8899
Attempting to start XML-RPC Server...
Started successfully.
Registered AreaHandler class to area.
Now accepting requests. (Halt program to stop.)
```

一旦启动AreaServer, 例2-3中的AreaClient类就会从命令行中检测AreaServer。AreaClient类还使用XML-RPC库, 这样只需几行代码（在areaCircle方法中）就可以真正执行调用。

例 2-3：检测 XML-RPC 服务器的一个 Java 客户端

```
package com.ecerami.xmlrpc;

import java.io.IOException;
import java.util.Vector;
import org.apache.xmlrpc.XmlRpc;
import org.apache.xmlrpc.XmlRpcClient;
import org.apache.xmlrpc.XmlRpcException;

public class AreaClient {

     public static void main(String args[]) {
         if (args.length < 1) {
             System.out.println(
                 "Usage: java AreaClient [radius]");
             System.exit(-1);
         }
         AreaClient client = new AreaClient();
         double radius = Double.parseDouble(args[0]);

         try {
              double area = client.areaCircle(radius);
              // 报告结果
              System.out.println("The area of the circle would be: " + area);

         } catch (IOException e) {
              System.out.println("IO Exception: " + e.getMessage());
         } catch (XmlRpcException e) {
              System.out.println("Exception within XML-RPC: " + e.getMessage());
         }
     }

     public double areaCircle (double radius)
       throws IOException, XmlRpcException {

             // 创建客户端, 标识服务器
             XmlRpcClient client =
                 new XmlRpcClient("http://localhost:8899/");

             // 利用用户输入创建请求参数
             Vector params = new Vector();
             params.addElement(new Double (radius));
```

```
            // 发送一个请求
            Object result = client.execute("area.circleArea", params);

            String resultStr = result.toString();
            double area = Double.parseDouble(resultStr);
            return area;
    }
}
```

`main` 方法解析命令行，并向用户报告结果，而 `areaCircle` 方法处理所有与 XML-RPC 服务的交互操作。服务器连续运行，而客户端要得到一个具体的结果只需运行一次。同样的请求可以重复使用或被修改，但每个请求都是一个独立的事件。在这个应用程序中，我们只需请求一次，从命令行获取一个值作为参数。客户端构造函数以 URL 为参数，标识出它将向哪个服务器发送请求。

发送请求也需要一些附加的、在创建服务器时不必要的安装工作。服务器可以靠方法签名找出哪个参数是哪个方法的，但客户端没有任何这方面的信息。Apache 实现在 `Vector` 对象中带有参数，`Vector` 对象需要使用 Java 包装类（像 `double` 原始类型需要 `Double` 对象一样）。`Vector` 一经构造，就会和过程调用的方法名一起被返回给 `execute` 方法。在本例中，方法名是 `area.circleArea`，从而反映了 `AreaHandler` 类在服务器上是用 `area` 这一名称注册的，并包括了一个名为 `circleArea` 的方法。

调用 `execute` 方法时，客户端向其构造函数中指定的服务器发出一个 XML-RPC 请求。在本例中，请求调用第一个参数 `area.circleArea` 指定的方法，以参数的形式传递第二个参数的内容，结果产生如下 HTTP 响应：

```
POST / HTTP/1.1
Content-Length: 175
Content-Type: text/xml
User-Agent: Java1.3.0
Host: localhost:8899
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive

<?xml version="1.0" encoding="ISO-8859-1"?>
<methodCall><methodName>area.circleArea</methodName>
<params>
<param><value><double>3.0</double></value></param>
```

```
</params>
</methodCall>
```

服务器用methodResponse响应，execute函数把它作为一个Object报告。虽然XML-RPC响应会提供这个Object的类型信息，基本内容也会与这个类型一致，但是Object是一个和execute函数一般返回的值一样的特殊类型，同时仍满足Java严格的类型检测要求。

这些工作产生的结果看起来非常简单：

```
C:\ora\xmlrpc\java>java com.ecerami.xmlrpc.AreaClient 3
The area of the circle would be: 28.274333882308138

C:\ora\xmlrpc\java>java com.ecerami.xmlrpc.AreaClient 4
The area of the circle would be: 50.26548245743669
```

但是，用XML-RPC将Java程序和Java程序连接起来并不令人兴奋。如果访问Java方法的惟一公共途径是通过XML-RPC，那么使用XML-RPC连接Java程序相当方便有效。然而，许多XML-RPC的潜力在于连接其他环境。为了说明这种方法在多种环境下都能起作用，我们将建立一个调用同样函数的Perl客户端。

Perl客户端将使用Frontier::RPC模块，这是由Ken MacLeod创建的一个XML-RPC实现（当MacLeod创建这个库时，XML-RPC还只是UserLand Frontier的一部分）。Frontier::RPC模块的客户端部分叫做Frontier::Client。

注意：Frontier::RPC和它使用的所有模块都可从位于*http://www.cpan.org*的CPAN获得。

除了Perl的灵活性允许我们跳过打包参数之外，XML-RPC调用的Perl版本和其Java版本在逻辑上非常相似。例2-4所示的程序从命令行接受一个radius值，创建一个新的XML-RPC连接，并以double类型向area.circleArea方法传递radius值，然后程序打印出结果。

例2-4：XML-RPC的一个Perl客户端

```
use Frontier::Client;
```

```
$radius=@ARGV[0];

print "for radius: ", $radius, "\n";

my $client=Frontier::Client->new(url=>"http://127.0.0.1:8899");

print " The area of the circle would be: ", $client->call('area.circleArea',
     Frontier::RPC2::Double->new($radius)), "\n";
```

此过程调用最为微妙的部分是转换过程需要确保数字被翻译成double类型。没有`Frontier::RPC2::Double->new($radius)`，Frontier::RPC模块就会把radius解释成字符串或整数。为了将Perl不严格的类型值映射为XML-RPC所要求的明确类型，Frontier::RPC提供了一组解释Perl值的模块。当用于命令行时，Perl过程调用产生与Java客户端产生的结果非常相似的结果：

```
C:\ora\xmlrpc\perl>perl circle.pl 3
for radius: 3
The area of the circle would be: 28.274333882308138

C:\ora\xmlrpc\perl>perl circle.pl 4
for radius: 4
The area of the circle would be: 50.26548245743669
```

注意：要想进一步了解XML-RPC的Java和Perl实现，以及Python、PHP和ASP（Active Server Pages）实现，请参阅《Programming Web Services with XML-RPC》（由O'Reilly出版）一书。

超越简单调用

XML-RPC是一个能力有限的简单概念。在许多方面，这些限制成为XML-RPC最引人注目的特点，因为它大大降低了实现协议和检测其互操作性的困难。虽然XML-RPC很简单，但简单工具有创造力的应用可以创建复杂而功能强大的体系结构。当各种各样的系统需要通信时，XML-RPC可能是最合适的最低标准。

有些应用只需要基本的功能，如前面描述的库式函数。通过组合使用数组和结构来传递复杂的信息，XML-RPC可以支持的开发比前面的例子演示的丰富得多。

计算圆的面积可能并不令人兴奋，但是处理矩阵和大量字符串却是立竿见影。XML-RPC 本身并不支持状态管理，但是应用程序可以使用参数来维持一个请求－响应循环中的多个对话，就像 Web 开发者用 cookie 跟踪长期对话一样。

服务器可以使用 XML-RPC 来传递客户端请求的信息，从而提供一个拥有大量信息的窗口。O'Reilly Network 的 Meerkat 用这种方法使用 XML-RPC，它让客户端指定需要通过 XML-RPC 过程接收的信息。在客户端需要向服务器传递信息时，XML-RPC 对日志式的操作和客户端需要设定服务器程序属性的操作都很有用。接口的丰富程度由开发者决定，当然这种可能性肯定存在。

第三部分

SOAP

第三章

SOAP 精髓

SOAP 是一种基于 XML 的、用于在计算机之间交换信息的协议。SOAP 能应用于各种消息接发系统，并能通过各种传输协议进行传递，但最初其侧重点是通过 HTTP 传输的远程过程调用。因此，SOAP 能使客户应用程序很容易地连接远程服务并调用远程方法。譬如（后面将详述），客户应用程序可以通过定位正确的 SOAP 服务和调用正确的方法，立即增加语言翻译功能。

包括 CORBA、DCOM 和 Java RMI 等在内的其他框架提供了与 SOAP 类似的功能，但是 SOAP 消息完全用 XML 编写，因此独具平台和语言的独立性。譬如，在 Linux 上运行的 SOAP Java 客户端或在 Solaris 上运行的 Perl 客户端可以连接到在 Windows 2000 上运行的 Microsoft SOAP 服务器。

因此，SOAP 在 Web 服务体系结构中充当基石，使各种应用能很容易地交换服务和数据。

虽然SOAP还处于发展初期，但已得到广泛的行业支持。现在已经有大量的SOAP实现，如面向 Java、COM、Perl、C# 和 Python 的实现，同时，Web 上成百上千的 SOAP 服务正蓬勃发展。

本章将介绍 SOAP 精髓，具体包括如下几个方面：

- 对 SOAP 协议做一个简短的回顾，并给出一个 SOAP 对话范例；
- 详述 SOAP XML 消息规范；
- 总结 SOAP 的编码规则，包括简单类型、数组和结构规则；
- 介绍通过 HTTP 使用 SOAP 的细节；
- 概述 W3C 在 SOAP 方面的相关工作；
- 总结 4 个主要的 SOAP 实现，讨论 SOAP 的互操作性问题。

SOAP 101

SOAP 规范主要定义了三个部分：

SOAP 信封规范

SOAP XML 信封（SOAP XML Envelope）对在计算机间传递的数据如何封装定义了具体的规则。这包括应用特定的数据，如要调用的方法名、方法参数或返回值；还包括谁将处理封装内容，失败时如何编码错误消息等信息。

数据编码规则

为了交换数据，计算机必须在编码特定数据类型的规则上达成一致。譬如，两台处理股票报价的计算机需要共同的编码浮点数据类型的规则；同样，两台处理多重股票报价的计算机需要共同的编码数组的规则。因此，SOAP 必须有一套自己的编码数据类型的约定。大部分约定都基于W3C XML Schema 规范。

RPC 协定

SOAP 能用于单向和双向等各种消息接发系统。SOAP 为双向消息接发定义了一个简单的协定来进行远程过程调用和响应，这使得客户端应用可以指定远程方法名，获取任意多个参数并接收来自服务器的响应。

为了解 SOAP 协议的特性，我们从一个 SOAP 对话例子入手。XMethods.net 提供了简单的天气服务，它根据邮政分区列出各地当前的温度（如图 3-1 所示）。服务方法 `getTemp` 需要获取一个邮政编码字符串并返回一个浮点值。

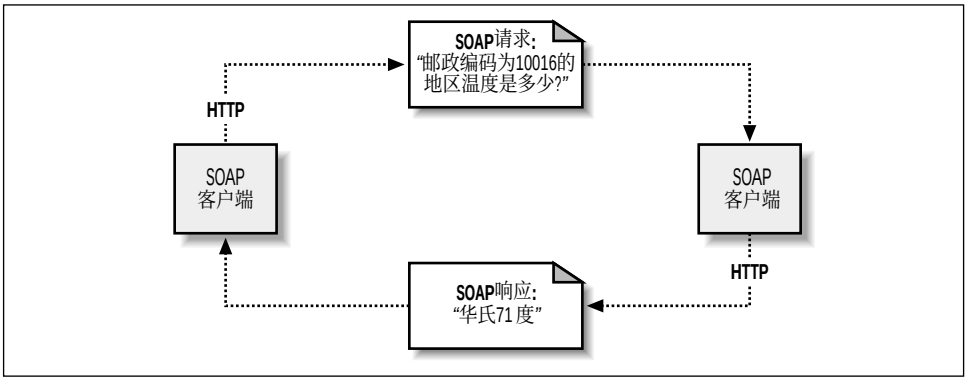

图 3-1：运转中的 SOAP：连接到 XMethods 天气服务

SOAP 请求

客户端请求必须包含要调用的方法的名称和所有必需的参数。下面是一个发送到 XMethods 的客户端请求的例子：

```
<?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:getTemp
      xmlns:ns1="urn:xmethods-Temperature"
      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
         <zipcode xsi:type="xsd:string">10016</zipcode>
      </ns1:getTemp>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

这里有几个重要的元素值得注意。首先，请求包含一个强制性 `Envelope` 元素，这个元素又包含一个强制性 `Body` 元素。

其次，请求共定义了四个 XML 名称空间。名称空间用于区分 XML 元素和属性，并经常用于引用外部模式。在我们的 SOAP 请求范例中，我们用名称空间来区分与 SOAP 信封（*http://schemas.xmlsoap.org/soap/envelope/*）相关的标识符，通过 XML Schema （*http://www.w3.org/2001/XMLSchema-instance* 和 *http://*

www.w3.org/2001/XMLSchema）编码的数据，以及 XMethods（*urn:xmethods-Temperature*）特有的应用标识符。这使应用程序模块化了，同时也为以后修改规范提供了最大的灵活性。

`Body` 元素封装 SOAP 消息的主要“有效负载”。惟一的元素是 `getTemp`，它被绑定到 XMethods 名称空间并与远程方法名相对应。方法的各个参数就像一个子元素。在这个例子中，我们只用到一个邮政编码元素，它被赋值为 XML Schema `xsd:string` 数据类型，并被设为 `10016`。如果还需要参数，则每个参数都有自己的数据类型。

SOAP 响应

下面是从 XMethods 发生的 SOAP 响应：

```
<?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:getTempResponse
      xmlns:ns1="urn:xmethods-Temperature"
      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
         <return xsi:type="xsd:float">71.0</return>
      </ns1:getTempResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

和请求一样，响应包含 `Envelope` 和 `Body` 元素，以及相同的四个 XML 名称空间。但是，这里的 `Body` 元素包含一个 `getTempResponse` 元素，与前面的请求相对应。响应元素用一个返回元素来指明是 `xsd:float` 数据类型。在撰写本文时，邮政编码 10016 所对应地区的温度是 71 华氏度。

SOAP 消息

如果急于开始编写自己的 SOAP 应用，你可以跳到本章后面的“SOAP 实现”一节。否则，我们接着研究 SOAP 规范本身。

源自客户端的请求或来自服务器的响应是一个单向消息，其正式名称为SOAP消息（SOAP message）。每个SOAP消息都有一个强制性Envelope元素、一个可选的Header元素和一个强制性Body元素（见图3-2）。每个元素都有一组相关的规则，理解这些规则将有助于调试你的SOAP应用程序。

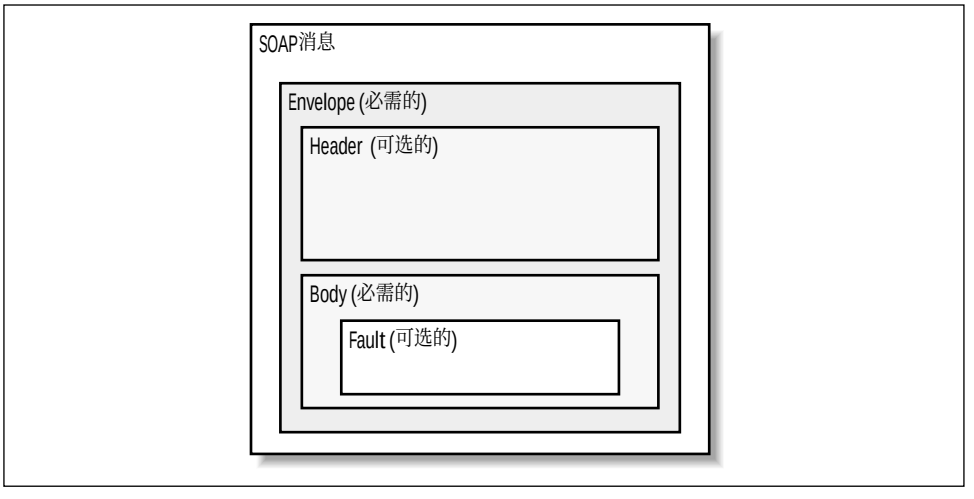

图3-2：XML SOAP消息的主要元素

信封

每个SOAP消息都有一个根Envelope元素。与HTTP和XML等其他规范不同的是，SOAP没有定义基于主版本号和次版本号（如HTTP 1.0和HTTP 1.1）的传统版本模型，而是用XML名称空间区分版本。版本必须在Envelope元素中引用，如：

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

SOAP 1.1名称空间的URI是*http://schemas.xmlsoap.org/soap/envelope/*，而SOAP 1.2名称空间的URI是*http://www.w3.org/2001/09/soap-envelope*（注1）。如果Envelope出现在其他名称空间中，则被视为版本错误。

注1： SOAP 1.2信封名称空间的确切值可能会改变，以反映SOAP 1.2发布的最后日期。*http://www.w3.org/2001/09/soap-envelope*反映的是从2001年9月以来的规范。

首部

可选的Header元素为定义更多的应用级需求提供了灵活的框架。例如，Header元素可用于为密码保护服务指定数字签名，还可用于为pay-per-use SOAP服务指定账号。许多现有的SOAP服务还没有使用Header元素，但是，当SOAP服务走向成熟时，Header框架将为身份验证、事务管理和支付授权提供一个开放机制。

Header元素的细节有意被设计为可修改，以便为应用提供者提供最大的灵活性。但是，SOAP协议还是规定了两个首部属性：

Actor属性

SOAP协议将消息路径（message path）定义成一系列SOAP服务节点。每个中间节点都可以执行一定的处理，并将消息传送到路径中的下一个节点。通过设置Actor属性，客户端能指定SOAP首部的接收者。

MustUnderstand属性

该属性表明Header元素是可选的还是强制性的。如要设为真（注2），则接收者一定要根据它定义的语义来理解和处理Header的属性，或返回一个故障信息（MustUnderstand错误编码见表3-2）。

下面是一个Header的例子：

```
<SOAP-ENV:Header>
   <ns1:PaymentAccount xmlns:ns1="urn:ecerami" SOAP-ENV:
        mustUnderstand="true">
      orsenigo473
   </ns1:PaymentAccount >
</SOAP-ENV:Header>
```

Header指定了一个SOAP服务器必须理解和处理的支付账号。

注2： SOAP 1.1采用整数值1/0来描述MustUnderstand属性；而SOAP 1.2则采用布尔值真/1，假/0。

主体

Body元素对所有SOAP消息都是强制性的。正如我们在前面所看到的那样，典型的 Body 元素都包含 RPC 请求和响应。

故障

当出现错误时，Body元素将包含一个 Fault 元素。表 3-1 中定义了故障子元素，包括 faultCode、faultString、faultActor 和 detail 元素；表 3-2 中定义了 SOAP 预定义故障代码。下面是 Fault 范例。客户端请求一个名为 Validate-CreditCard 的方法，但是服务器不支持该方法，这就会发生客户端请求错误，服务器就会返回如下的 SOAP 响应：

```
<?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/1999/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema">
   <SOAP-ENV:Body>
      <SOAP-ENV:Fault>
         <faultcode xsi:type="xsd:string">SOAP-ENV:Client</faultcode>
         <faultstring xsi:type="xsd:string">
            Failed to locate method (ValidateCreditCard) in class
            (examplesCreditCard) at /usr/local/ActivePerl-5.6/lib/
            site_perl/5.6.0/SOAP/Lite.pm line 1555.
         </faultstring>
      </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

表 3-1：SOAP 故障子元素

元素名称	描述
faultCode	用于标明一类错误的文本代码。参阅表 3-2 中的一组预定义故障代码
faultString	对错误的一个可阅读的解释

表 3-1：SOAP 故障子元素（续）

元素名称	描述
`faultActor`	标明是谁引起故障的一个文本字符串。如果 SOAP 消息经过了 SOAP 消息路径中的几个节点，客户端要知道是哪个节点导致了错误时，这个元素就显得非常重要。中间节点一定要包含 `faultActor` 元素
`detail`	一个用于携带应用程序特有的错误消息的元素。`detail`元素可以包含称为细节条目的子元素

表 3-2：SOAP 故障代码

名称	描述
SOAP-ENV:VersionMismatch	标明SOAP 的`Envelope`元素包含一个无效的名称空间，表示版本不匹配
SOAP-ENV:MustUnderstand	标明接收端不能正确处理一个其`mustUnderstand`属性被设为 `true` 的`Header` 元素。这可以确保`mustUnderstand`元素不会被无声地忽略掉
SOAP-ENV:Client	标明客户端请求包含一个错误信息。例如，客户端指定了一个不存在的方法名，或者向方法提供了不正确的参数
SOAP-ENV:Server	标明服务器不能处理客户端的请求。例如，用于提供产品数据的服务器可能不能连接到数据库

SOAP 编码

SOAP 有一套内置的编码数据类型的规则，这使得 SOAP 消息能表明特定的数据类型，如整型、浮点型、双精度型或数组。多数情况下，编码规则由你选用的SOAP 工具包直接实现，因此你是见不到规则的。但它对于理解SOAP编码基础很有用，特别是当你截取SOAP消息和试图调试应用程序时。另外值得注意的是，当W3C 规范鼓励使用 SOAP 编码规则时，那些规则其实并非必需的，这使得你在必要时可以使用不同的编码模式。

探索SOAP编码规则时一定要注意的是，XML 1.0规范没有包含编码数据类型的规则，因此，最初的SOAP规范不得不定义它自己的数据编码规则。继SOAP规范初稿之后，W3C 又发布了 XML Schema规范。XML Schema规范的第二部分Datatypes规范为XML文档内的编码数据类型提供了标准框架，因此SOAP规范采用XML Schema约定。但是，即使最新的SOAP规范采用所有由XML Schema定义的内置类型，它还是保留了它自己的定义数组和引用等不符合XML Schema标准结构的约定。本章后面的“复合类型”一节将详细讨论数组。

SOAP数据类型分为两大类：标量类型和复合类型。标量类型仅包含一个值，如姓氏、单价或产品描述。复合类型包含多个值，如订购单或一系列股票报价。复合类型进一步分为数组和结构。数组包含多个值，每个值按顺序被指定。结构也含有多个值，但每个值由存取方法名指定。

SOAP消息的编码形式通过`SOAP-ENV:encodingStyle`属性设定。要使用SOAP 1.1编码，请使用值`http://schemas.xmlsoap.org/soap/encoding/`。要使用SOAP 1.2编码，请使用值`http://www.w3.org/2001/09/soap-encoding`。

标量类型

至于标量类型，SOAP采用了由XML Schema规范指定的所有内置简单类型，包括字符串型、浮点型、双精度型和整型。表3-3摘自XML Schema中的Part 0：Primer（*http://www.w3.org/TR/2000/WD-xmlschema-0-20000407/*），列出了主要的简单类型。

表3-3：XML Schema主要的内置简单类型列表

简单类型	例子
string	Web服务
Boolean	true, false, 1, 0
float	-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN
double	-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN
decimal	-1.23, 0, 123.4, 1000.00

表 3-3：XML Schema 主要的内置简单类型列表（续）

简单类型	例子
binary	100010
integer	-126789, -1, 0, 1, 126789
nonPositiveInteger	-126789, -1, 0
negativeInteger	-126789, -1
long	-1, 12678967543233
int	-1, 126789675
short	-1, 12678
byte	-1, 126
nonNegativeInteger	0, 1, 126789
unsignedLong	0, 12678967543233
unsignedInt	0, 1267896754
unsignedShort	0, 12678
unsignedByte	0, 126
positiveInteger	1, 126789
date	1999-05-31
time	13:20:00.000, 13:20:00.000-05:00

例如，下面是一个 SOAP 的双精度数据类型响应：

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://www.w3.org/2001/09/soap-envelope"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
      <ns1:getPriceResponse
         xmlns:ns1="urn:examples:priceservice"
         SOAP-ENV:encodingStyle="http://www.w3.org/2001/09/soap-encoding">
             <return xsi:type="xsd:double">54.99</return>
      </ns1:getPriceResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

从中可以看到，`xsi:type` 属性被设置为 `xsd:double`，表明返回一个双精度值。

SOAP规范为标明特定的XML元素的数据类型提供了几个选择。第一个选择是为每个元素指定一个 xsi:type 属性；第二个选择是在一个外部 XML Schema，或者甚至在一个能直接阅读的文档中存放数据类型信息。SOAP 工具包会随着这种要求的实现方式的不同而不同，如 Apache SOAP 工具包自动为每个元素包含一个 xsi:type 属性，但是 Microsoft SOAP 工具包会忽略 xsi:type 属性，而假定一个外部 XML Schema 定义。本章的例子来自 Apache SOAP，因此使用了 xsi:type 属性。要想了解更详细的信息，请参阅本章后面的“ SOAP 互操作性 ”一节。

复合类型

SOAP 数组有一套特殊的规则，要求 SOAP 数组不但要指定元素类型，还要指定数组大小。SOAP 还支持多维数组，但并非所有 SOAP 实现都支持多维调用（如想了解详细信息，请参阅你的 SOAP 工具包）。

创建一个数组时，必须将 xsi:type 指定为 Array。数组还必须包含一个 arrayType 属性，指定所包含元素的数据类型和数组的维数需要该属性。譬如，属性 arrayType="xsd:double[10]" 指定一个含 10 个双精度值的数组，而属性 arrayType="xsd:string[5,5]" 指定一个二维的字符串数组。

下面就是含有一个双精度值数组的 SOAP 响应的例子：

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://www.w3.org/2001/09/soap-envelope"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
      <ns1:getPriceListResponse
         xmlns:ns1="urn:examples:pricelistservice"
         SOAP-ENV:encodingStyle="http://www.w3.org/2001/09/soap-encoding">
          <return
            xmlns:ns2="http://www.w3.org/2001/09/soap-encoding"
            xsi:type="ns2:Array" ns2:arrayType="xsd:double[2]">
            <item xsi:type="xsd:double">54.99</item>
            <item xsi:type="xsd:double">19.99</item>
         </return>
```

```
      </ns1:getPriceListResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

请注意，`arrayType` 被设定为 `xsd:double[2]`，数组中的每个元素都被指定为一个 `item` 元素。

与数组相比，结构也含有多个值，但用一个惟一的存取方法元素来指定每个元素。譬如，考虑一个产品目录中的某项时，结构必须包含产品 SKU、产品名称、描述和价格。从下面的例子可以看出这种结构如何在 SOAP 消息中表示。

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://www.w3.org/2001/09/soap-envelope"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
      <ns1:getProductResponse
         xmlns:ns1="urn:examples:productservice"
         SOAP-ENV:encodingStyle="http://www.w3.org/2001/09/soap-encoding">
            <return xmlns:ns2="urn:examples" xsi:type="ns2:product">
            <name xsi:type="xsd:string">Red Hat Linux</name>
            <price xsi:type="xsd:double">54.99</price>
            <description xsi:type="xsd:string">
               Red Hat Linux Operating System
            </description>
            <SKU xsi:type="xsd:string">A358185</SKU>
         </return>
      </ns1:getProductResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

结构中的每个元素都被指定了一个惟一的存取方法名。譬如，上面的消息中包含 4 个存取方法元素：`name`、`price`、`description` 和 `SKU`。每个元素都有自己的数据类型，如 `name` 为字符串型，`price` 为双精度型。

注意：本节中所有的SOAP消息例子（包括数组、结构和直接XML文档）都是用Apache SOAP工具包创建的。如想了解完整的内容，请参阅第五章。

直接编码

前面已经提到，不必使用 SOAP 编码形式。事实上，有时要完全避开 SOAP 的编码规则，而直接将完整的 XML 文档（或仅仅是一部分）植入 SOAP 消息中。我们称这种做法为直接 XML 编码，你必须指定一种直接 XML 编码形式。在 Apache SOAP 中，直接 XML 形式用名称空间 *http://xml.apache.org/xml-soap/literalxml* 指定。

譬如，下面为编码产品信息的第二种方法，它不是将产品编码为 SOAP 结构，而是将数据编码成直接 XML 文档。

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://www.w3.org/2001/09/soap-envelope"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
      <ns1:getProductResponse
         xmlns:ns1="urn:examples:XMLproductservice"
           SOAP-ENV:encodingStyle=
             "http://xml.apache.org/xml-soap/literalxml">
         <return>
            <product sku="A358185">
            <name>Red Hat Linux</name>
            <description>Red Hat Linux Operating System</description>
            <price>54.99</price></product>
         </return>
      </ns1:getProductResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

注意：有关 SOAP 编码规则的进一步讨论，请参阅 James Snell、Doug Tidwell 和 Pavel Kulchenko 合著的《Programming Web Services with SOAP》（由 O'Reilly 出版）。

SOAP 通过 HTTP 传输

SOAP 没有绑定到任何一种传输协议上，因此，它可以通过 SMTP、FTP、IBM MQSeries 或 Microsoft Message Queuing（MSMQ）传输。但是，SOAP 规范只包含有关 HTTP 的细节，所以 HTTP 仍是流行的 SOAP 传输协议。

自然，SOAP 请求能通过 HTTP 请求发送，并在 HTTP 响应内容中返回 SOAP 响应。虽然 SOAP 请求可以通过 HTTP GET 发送，但规范只包含 HTTP POST 的细节（之所以只提到 HTTP POST，是因为多数服务器限制 GET 请求字符）。另外，HTTP 请求和响应都要求将其内容设为 `text/xml` 类型。

另一个要求是，客户端必须指定一个 `SOAPAction` 首部。它是用来指定请求内容的一个服务器专用 URI。这使得我们实际上不用查看 SOAP 消息的有效负载，就可以快速确定 SOAP 请求的性质。在实际应用中，防火墙经常把首部作为一种阻挡 SOAP 请求或快速地将 SOAP 消息分配到相应的 SOAP 服务器的机制。

SOAP 规范要求客户端必须提供一个 `SOAPAction` 首部，但 `SOAPAction` 首部的实际值依赖于 SOAP 服务器的实现。譬如，要想获得由 XMethods 提供的 AltaVista BabelFish Translation 服务，就一定要将 `urn:xmethodsBabelFish#BabelFish` 指定为 `SOAPAction` 首部。即使服务器不需要完整的 `SOAPAction` 首部，客户端也必须指定一个空字符串（`""`），或者一个 null 值。例如：

```
SOAPAction: ""
SOAPAction:
```

下面是通过 HTTP 向 XMethods Babelfish Translation 服务发送请求的样例。

```
POST /perl/soaplite.cgi HTTP/1.0
Host: services.xmethods.com
Content-Type: text/xml; charset=utf-8
Content-Length: 538
SOAPAction: "urn:xmethodsBabelFish#BabelFish"

<?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/1999/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema">
   <SOAP-ENV:Body>
      <ns1:BabelFish
      xmlns:ns1="urn:xmethodsBabelFish"
      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
         <translationmode xsi:type="xsd:string">en_fr</translationmode>
         <sourcedata xsi:type="xsd:string">Hello, world!</sourcedata>
      </ns1:BabelFish>
```

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

请注意内容类型和SOAPAction首部。我们还看到BabelFish方法需要两个字符串型参数。en_fr 翻译模式为英文译为法文。

下面是来自 XMethods 的响应：

```
HTTP/1.1 200 OK
Date: Sat, 09 Jun 2001 15:01:55 GMT
Server: Apache/1.3.14 (Unix) tomcat/1.0 PHP/4.0.1pl2
SOAPServer: SOAP::Lite/Perl/0.50
Cache-Control: s-maxage=60, proxy-revalidate
Content-Length: 539
Content-Type: text/xml

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema">
   <SOAP-ENV:Body>
      <namesp1:BabelFishResponse xmlns:namesp1="urn:xmethodsBabelFish">
         <return xsi:type="xsd:string">Bonjour, monde!</return>
      </namesp1:BabelFishResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

通过 HTTP 传递的 SOAP 响应需要沿用相同的 HTTP 状态代码，如 200 OK 状态代码表示成功的响应，500 Internal Server Error 状态代码表示存在服务器错误并且 SOAP 响应包含一个 Fault 元素。

注意：包括 W3C XML 协议工作组（W3C XML Protocol Working Group）成员在内的许多人都认为SOAPAction首部的意义和作用非常模糊。因此，在SOAP 1.2中，SOAPAction状态由必需改为可选。未来的规范虽然反对使用SOAPAction首部，但仍会保留它以确保向下兼容。

SOAP 和 W3C

SOAP 1.1 最初被提交到 W3C 的时间是 2000 年 5 月。正式提交的成员既有微软、IBM、Ariba 等大公司，也有 UserLand Software 和 DevelopMentor 等小公司。

2000 年 9 月，W3C 成立了新的 XML 协议工作组，其任务是制定出一个用作信息交换的 XML 协议，并以正式的 W3C 建议推荐该协议。

2001 年 7 月，XML 协议工作组发布了 SOAP 1.2 的“工作草案”。在 W3C 内部，该文档被列入正式工作日程，这意味着在最后定稿前它将被更新多次。然而，到目前为止，SOAP 1.2 并未根本脱离 SOAP 1.1，其主要任务是澄清 SOAP 1.1 规范中的模糊条文。因此，大部分开发者应该可以相对平稳地从 SOAP 1.1 过渡到 SOAP 1.2。

注意：W3C 已经将 SOAP 1.2 分为两部分：第一部分描述 SOAP 消息接发框架和信封规范；第二部分描述 SOAP 编码规则、SOAP-RPC 协定和 HTTP 绑定细节。

一旦定稿，SOAP将进入正式的W3C建议状态。值得注意的是，即使到那时，SOAP 也并没有得到 W3C 的正式承诺。SOAP 从 1.1 版就处于“Note”状态，表明它在 W3C 成员内部就有争议。

注意：SOAP 最初代表简单对象存取协议（Simple Object Access Protocol）。W3C 始终对这种定义不满意，主要是因为规范中其实并未强制使用对象。另一方面，W3C 又不想定义一个如 XML Protocol 或 XML-P 之类的新名词，主要是因为 SOAP 这一术语已在开发人员中根深蒂固。因此，在这种奇特的背景下，SOAP 名称仍然保留，但 W3C 说 SOAP 不再代表什么了。

要想获得 XML 协议工作组的最新情况，请访问 *http://www.w3.org/2000/xp/Group/*。工作组还在 *xml-dist-app@w3.org* 提供了公共电子邮件讨论列表。

注意：SOAP 1.1 可在 *http://www.w3.org/TR/SOAP/* 上在线获得，SOAP 1.2 的工作草案可在 *http://www.w3.org/TR/soap12/* 上获得。请注意，W3C 还提供了脱离 SOAP 核心规范的“SOAP Messages with Attachments（带附件的 SOAP 消息）”提交报告。这一规范使 SOAP 消息可以包含二进制的附件，如图像和声音文件。要想全面了解，请参阅 *http://www.w3.org/TR/SOAP-attachments* 上的 W3C Note。

SOAP 实现

Internet 上有大量免费的 SOAP 实现，到编写本书时为止，SOAPWare.org 实际上已经引用了共 65 条实现。下面是最流行、引用最为广泛的四条实现。

Apache SOAP（http://xml.apache.org/soap/）
: SOAP 协议的开源 Java 实现，基于 IBM SOAP4J 实现。

Microsoft SOAP ToolKit 2.0（http://msdn.microsoft.com/soap/default.asp）
: 面向 C#、C++、Visual Basic 或其他遵从 COM 语言的 SOAP 协议的 COM 实现。

面向 Perl 的 SOAP::Lite（http://www.soaplite.com/）
: SOAP 协议的 Perl 实现，由 Paul Kulchenko 编写，包括对 WSDL 和 UDDI 的支持。

Mind Electric 公司的 GLUE（http://www.themindelectric.com）
: SOAP 协议的 Java 实现，包括对 WSDL 和 UDDI 的支持。

第四章和第五章提供了 Apache SOAP 的完整信息，第六章还将简短讨论 SOAP::Lite 和 GLUE。要想获得更完整的列表，或想找到适合你所选择语言或操作平台的 SOAP 实现，请访问 *http://www.soapware.org/directory/4/implementations*。

SOAP 互操作性

SOAP 是专门为解决操作平台和语言互操作性问题而设计的。具有讽刺意味的是，SOAP 自身也存在互操作性问题，但这只是暂时性的。譬如，在编写本书时，

Apache SOAP、面向 Perl 的 SOAP::Lite 以及 Microsoft SOAP ToolKit 之间都存在互操作性问题。Apache SOAP 要求所有的参数都通过 `xsi:type` 属性输入，而 Microsoft SOAP ToolKit 并不要求这么做；各个实现都不同程度地强制使用 `mustUnderstand` 属性，Microsoft SOAP 支持多维数组，而 Apache SOAP 和面向 Perl 的 SOAP::Lite 仅支持一维数组。

互操作性问题源于两个主要问题。首先，也是最主要的，SOAP 还处于不成熟期。SOAP 于 2000 年 5 月被提交到 W3C，至今还没有得到 W3C 的正式建议；其次是现在已经有大量的 SOAP 实现，确保它们之间的互操作还需付出很大努力。

所有这些互操作性问题都有望很快得以解决。实现自身会走向成熟，SOAP 规范也会在 W3C 的努力下趋于成熟。要想获得 SOAP 互操作性的更多信息，请访问 Microsoft Interoperability Site（*http://www.mssoapinterop.org*），或访问 XMetheods Interoperability Lab（*http://www.xmethods.net/ilab/*）。这两个站点都提供了一套互操作性测试软件，还包含大部分的主流 SOAP 实现的最新结果。

第四章

Apache SOAP 快速入门

Apache SOAP是SOAP规范的开源Java实现。IBM将IBM SOAP4J捐献给Apache开源社区，而Apache最初的代码就建立在SOAP4J基础之上。和所有其他Apache项目类似，Apache SOAP源代码对商业和非商业应用都是免费的。该源代码极易获得，并且一组活跃的程序员正为未来版本增加新的功能。

本章为使用 Apache SOAP 提供了一个快速入门介绍，目的是让你获得最重要的精髓部分，以便你开始编写代码并着手工作。本章包括以下四个主题：

- 在开源 Jakarta Tomcat 服务器上使用 Apache SOAP 的安装指南；
- 为说明基本的 SOAP RPC 编码，给出一个“ Hello, SOAP! ”客户端 / 服务器应用程序；
- 对部署以及通过基于Web的管理工具和通过命令行`ServiceManagerClient`工具管理 SOAP 服务做一个总结；
- 介绍用 `TcpTunnelGui` 工具查看动态 SOAP 对话的技巧。

安装 Apache SOAP

如果你仅仅是创建SOAP客户端，那么你只需下载正确的文件组，并将相关的JAR

文件并入你的CLASSPATH（详细信息见后）。但如果你要创建SOAP服务，你就需要获得正确的文件和一个Java servlet引擎。例如在BEA Weblogic Application Server、IBM WebSphere或者Allaire JRun上安装Apache SOAP。本节包含在开源Apache Jakarta Tomcat 3.2服务器上安装Apache SOAP的具体指南。Tomcat是免费的，而且容易安装，几分钟后就可运行。

注意：因为名为`rpcrouter`的Apache SOAP服务其实只是一个已被配置成可以接收SOAP请求的Java servlet，所以需要一个Java servlet引擎来安装SOAP服务。

下载所需的Java文件

要在Jakarta Tomcat上安装Apache SOAP，你需要下载5个发布文件。先下载Tomcat和Apache SOAP发布版：

- Apache Jakarta Tomcat: *http://jakarta.apache.org/tomcat/*
- Apache SOAP: *http://xml.apache.org/soap/*

然后下载Xerces Java Parser、Java Mail和JavaBeans Activation Framework发布版：

- Xerces Java Parser（1.1.2版或更高版本）：*http://xml.apache.org/xerces-j/index.html*
- Java Mail: *http://java.sun.com/products/javamail/*
- JavaBeans Activation Framework: *http://java.sun.com/products/javabeans/glasgow/jaf.html*

安装Tomcat CLASSPATH

下一步我们来设置Jakarta Tomcat的CLASSPATH。注意一定要包含如下JAR文件和目录：

- *soap.jar*
- *xerces.jar*
- *mail.jar*
- *activation.jar*
- SOAP 应用类文件目录

要设置 Tomcat CLASSPATH，请编辑服务器启动文件（在 Windows 系统中，文件名为 *tomcat.bat*；在 Unix 系统中，文件名为 *tomcat.sh*）。例如，在 *tomcat.bat* 文件中题为“ Set Up the Runtime Classpath ”的部分，我加入以下几行：

```
echo Adding xerces.jar to beginning of CLASSPATH
set CP=c:\web_services\lib\xerces.jar;%CP%
echo Adding soap.jar to CLASSPATH
set CP=%CP%;C:\web_services\lib\soap.jar
echo Adding mail.jar and activation.jar to CLASSPATH
set CP=%CP%;c:\web_services\lib\mail.jar;c:\web_services\lib\activation.jar
echo Adding SOAP Examples Directory
set CP=%CP%;c:\web_services\examples\classes
```

注意，虽然 Tomcat 发布版有自己的 XML 解析程序，但内置的解析程序并不熟悉名称空间，所以在 Apache SOAP 中不能运行。因此，你需要在 CLASSPATH 的最前面添加 *xerces.jar*，以强制 Tomcat 使用 Xerces（1.1.2 或更高版本）。如：

```
set CP=c:\web_services\lib\xerces.jar;%CP%
```

配置 Tomcat

最后一步，用 Tomcat 注册 Apache SOAP 服务。你只需将下面几行加入 Tomcat 配置文件（*conf/server.xml*）即可:

```
<Context path="/soap" docBase="C:\web_services\soap-2_2\webapps\soap"
         debug="1" reloadable="true">
</Context>
```

要确保正确设置了 `docBase`，以反映出你的本地安装。

启动 Tomcat

现在你可以启动 Tomcat 服务器了。在 Windows 系统中，运行 *startup.bat* 文件；在 Unix 系统中，运行 *startup.sh* 文件。如果你运行的是 Windows 系统，你将会看到如图 4-1 所示的启动窗口。默认状态下，Tomcat 会从 8080 端口启动，你将会在启动窗口中看到“ Starting HttpConnectionHandler on 8080 ”的字样。

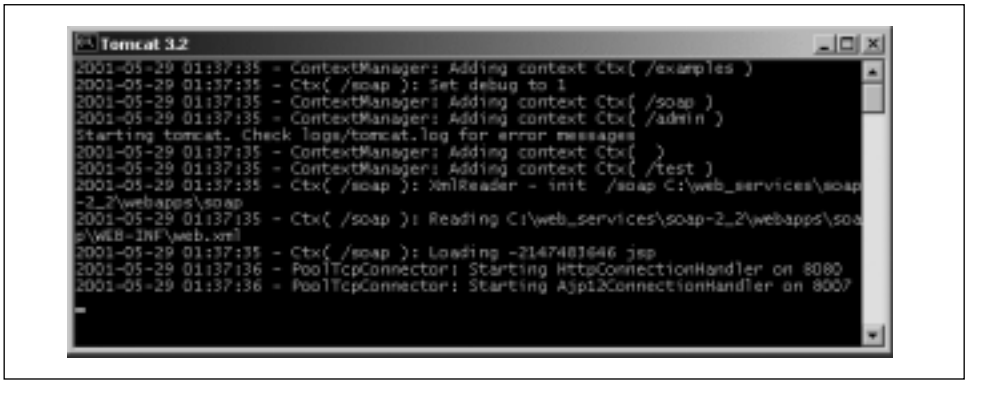

图 4-1：启动 Tomcat

运行 SOAP Administrator

Tomcat 运行时，你可以访问 SOAP Administrator。打开浏览器窗口，进入 *http://localhost:8080/soap*。如果看到如图 4-2 所示的欢迎界面，就说明安装全部正确，你可以开始部署自己的 SOAP 服务了。

Hello,SOAP!

现在你已经安装好了Apache SOAP, 让我们来实际使用一下, 构造第一个Apache SOAP 应用“ Hello,SOAP! ”。“ Hello,SOAP! ”服务提供一个 `sayHello()`方法，它接受一个名字参数，并返回一个个性化的问候。我们将从探索图 4-3 所示的一般体系结构开始。图中包含了负责处理数据的所有元素，并描绘了“ Hello,SOAP! ”对话的步骤。

图 4-2：Apache SOAP 欢迎界面

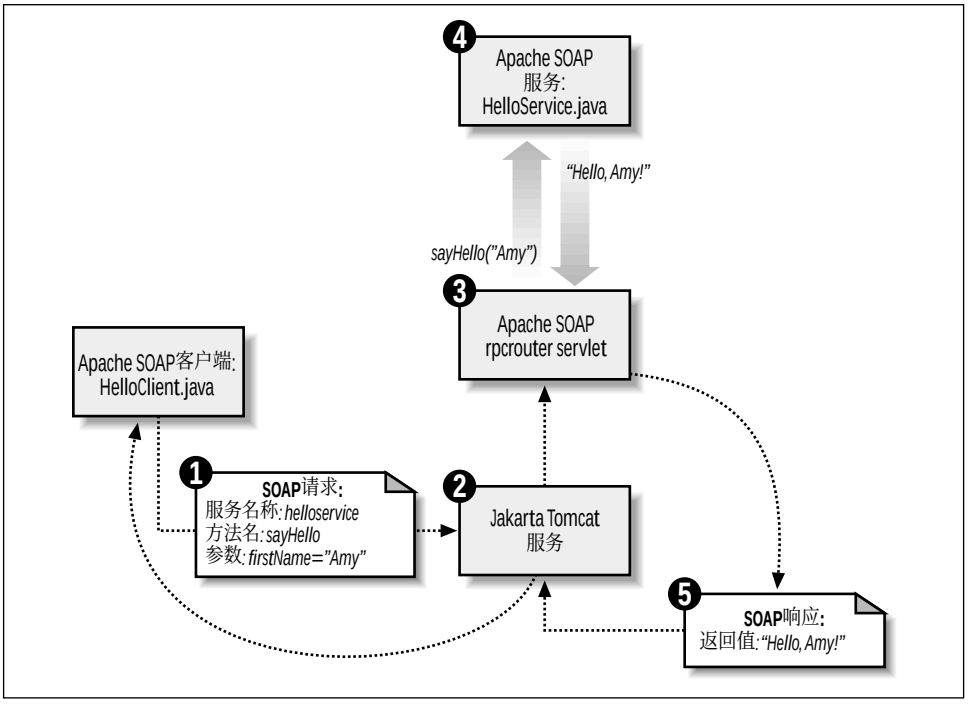

图 4-3：Apache SOAP 结构

1. Apache SOAP 客户端产生一个 SOAP 请求，并通过 HTTP POST 发送。客户端请求指定*helloservice*服务和`sayHello`方法，还包含一个名为*firstName*的参数。
2. Jakarta Tomcat 服务器接收请求，并转发给 Apache `rpcrouter` servlet。
3. `rpcrouter`查找请求的*helloservice*，生成一个`HelloService`对象实例，并调用 `sayHello()`方法。
4. `HelloService`对象提取`firstName`参数（如“ Amy ”）并返回一句问候（如“ Hello, Amy! ”）。
5. `rpcrouter` 捕获问候，把结果包入 SOAP 响应，并将响应返回客户端。

服务代码

我们首先查看服务代码（见例4-1）。服务代码最突出的一个特点是它只是一般的 Java 类。不需要导入任何 Apache SOAP 特定的库，也不用实现任何 SOAP 特定的接口，你只需提供你的服务所支持的方法。在例4-1中，我们提供了`sayHello()`方法的一个实现，它带有一个 `String`参数 `firstName`，并返回一个问候。这就是“ Hello, SOAP! ”的全部。

例 4-1：HelloService.java

```
package com.ecerami.soap;

/**
 * "Hello, SOAP!" SOAP 服务
 * 对任何客户端应用提供一个个性化的问候
 */
public class HelloService {

  /**
  * 问候客户端
  */
  public String sayHello (String firstName) {
    return new String ("Hello, "+firstName+"!");
  }
}
```

客户端代码

与服务代码不同，编写客户端代码需要连接 Apache SOAP API。但是，不管有多复杂，客户端代码一般按以下五步进行：

1. 创建一个RPC `Call`对象。`Call`对象封装调用远程SOAP服务的所有细节，如SOAP服务名和将调用的方法名。
2. 建立一组将传送到远程服务的参数。为传送原始数据类型、字符串、向量和数组等多种数据类型，Apache SOAP 提供了内置的支持。我们在第五章还会看到，Apache SOAP 还支持传送 JavaBean 和直接 XML 文档。
3. 调用远程方法。客户端在幕后把相关的数据包装进SOAP请求，发送到SOAP服务器，并接收和解析 SOAP 响应。
4. 检查 SOAP 响应中的错误。
5. 从 SOAP 响应中提取返回值。

例 4-2 列出了“ Hello, SOAP! ”应用的全部客户端代码。

例 4-2：HelloClient.java

```
package com.ecerami.soap;

/**
 * "Hello, SOAP!" SOAP客户端
 * 用法: java HelloClient first_name
*/
import java.net.*;
import java.util.Vector;
import org.apache.soap.SOAPException;
import org.apache.soap.Fault;
import org.apache.soap.Constants;
import org.apache.soap.rpc.Call;
import org.apache.soap.rpc.Parameter;
import org.apache.soap.rpc.Response;

public class HelloClient {

  /**
   * 静态main方法
   */
```

```
public static void main (String[] args) {
  String firstName = args[0];
  System.out.println ("Hello SOAP Client");
  HelloClient helloClient = new HelloClient();
  try {
    String greeting = helloClient.getGreeting(firstName);
    System.out.print (greeting);
    } catch (SOAPException e) {
    String faultCode = e.getFaultCode();
    String faultMsg = e.getMessage();
    System.err.println ("SOAPException Thrown (details below):");
    System.err.println ("FaultCode: "+faultCode);
    System.err.println ("FaultMessage: "+faultMsg);
} catch (MalformedURLException e) {
    System.err.println (e);
  }
}

/**
 * getGreeting方法
*/
public String getGreeting (String firstName)
  throws SOAPException, MalformedURLException {

  // 创建SOAP RPC Call对象
      Call call = new Call ();

  // 设置编码形式为标准SOAP编码
  call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

  // 设置对象URI和方法名
       call.setTargetObjectURI ("urn:examples:helloservice");
   call.setMethodName ("sayHello");

  // 设置方法参数
      Parameter param = new Parameter("firstName", String.class,
    firstName, Constants.NS_URI_SOAP_ENC);

   Vector paramList = new Vector ();
   paramList.addElement (param);
    call.setParams (paramList);

  // 设置Web服务的URL
  URL url = new URL ("http://localhost:8080/soap/servlet/rpcrouter");

  // 调用服务
       Response resp = call.invoke (url, "");
```

```
    // 检查故障
        if (!resp.generatedFault()) {
     // 提取返回值
       Parameter result = resp.getReturnValue ();
       String greeting = (String) result.getValue();
      return greeting;
    }
    else {
      // 提取故障代码和字符串
           Fault f = resp.getFault();
       String faultCode = f.getFaultCode();
       String faultString = f.getFaultString();
      System.err.println("Fault Occurred (details follow):");
      System.err.println("Fault Code: "+faultCode);
      System.err.println("Fault String: "+faultString);
      return new String ("Fault Occurred. No greeting for you!");
    }
  }
}
```

代码需要一个表示名的命令行参数。例如，命令行

```
java com.ecerami.soap.HelloClient Amy
```

将产生如下输出：

```
Hello SOAP Client
Hello, Amy!
```

大量 SOAP 特定的代码在 `getGreeting()`方法中产生，因此我们从那里开始代码分析。

RPC Call 对象

要生成一个 SOAP 请求，首先应实例化 `org.apache.soap.rpc.Call`对象：

```
Call call = new Call ();
```

`Call`对象封装SOAP请求的所有细节，例如我们设定的SOAP编码形式。默认的 SOAP 编码形式使用 `Constants.NS_URI_SOAP_ENC`：

```
call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
```

其他编码形式将在第五章讨论。Call对象还封装了所期望SOAP服务的URI和要调用的方法名：

```
call.setTargetObjectURI ("urn:examples:helloservice");
call.setMethodName ("sayHello");
```

API: org.apache.soap.rpc.Call

`void setEncodingStyleURI(String encodingStyleURI)`

为在 SOAP 信封内传递的参数设置编码形式 URI。取以下参数：

`encodingStyleURI`

编码形式 URI。使用 `Constants.NS_URI_SOAP_ENC` 为默认的编码形式（*http://schemas.xmlsoap.org/soap/encoding/*）。使用 `Constants.NS_URI_LITERAL_XML` 传递直接 XML 文档。

`void setTargetObjectURI(String targetObjectURI)`

设置目标对象 URI。取以下参数：

`targetObjectURI`

远程服务的URI。这通常是要调用服务的URN, 例如*urn:examples:helloservice*。

`void setMethodName(String methodName)`

设置远程方法名。取以下参数：

`methodName`

远程方法的名称，例如 `sayHello`。

`void setParams(Vector params)`

设置将从客户端传递至服务器的参数向量。取以下参数：

`params`

参数向量。向量必须由`org.apache.soap.rpc.Parameter`对象组成。

`Response invoke(URL url, String SOAPActionURI) throws SOAPException`

调用远程方法。在幕后，该方法将连接到指定的服务，发送SOAP请求，获取并解析SOAP响应。当发生致命错误时，包括网络连接失败或违反了SOAP协议，该方法将抛出一个`SOAPException`异常。取以下参数：

`url`

SOAP 服务器的绝对 URL。

SOAPActionURI
可选的 SOAPAction HTTP 首部。SOAPAction 一般用来表明 SOAP 服务的 URI。空字符串（""）表明 SOAP 目标在 HTTP 请求 URI 中指定。

设定参数

要给远程方法传送数据，你必须创建一个或多个参数。每个参数都要生成一个 org.apache.soap.rpc.Parameter 对象的实例。Parameter 构造函数需要下面四个参数：

- 参数名
- 类的类型，如 String.class、Integer.class 或 Double.class
- 参数值
- 编码形式。如果设为 null，则参数将使用 Class 对象的编码形式。

API: org.apache.soap.rpc.Parameter

Parameter（String name, Class type, Object value, String encodingStyleURI）
构建一个新的 Parameter 对象。取以下参数：

name
参数名。

type
Java 类的类型，例如 String.class、Double.class 或 String[].class。

value
参数值。

encodingStyleURI
参数的编码形式 URI。如果设为 null，则默认为指定给 Call 对象的形式。

public java.lang.Object getValue()
获取 Parameter 对象的值。

例如，我们增加一个 String 参数：

```
Parameter param = new Parameter("firstName", String.class,
   firstName, Constants.NS_URI_SOAP_ENC);
```

传送原始数据类型（如 int、double 或 float）也遵循相同的过程，惟一的区别是必须指定对象的包装，如 Integer、Double 或 Float。例如，以下代码创建了一个 Double 参数：

```
discountParam = new Parameter ("discount", Double.class,
   discount, Constants.NS_URI_SOAP_ENC);
```

每个参数都被添加到一个Vector对象中，全部Vector对象又通过setParams()方法被传送到 Call 对象中：

```
Vector paramList = new Vector ();
paramList.addElement (param);
call.setParams (paramList);
```

创建参数时，每个参数都有一个名 / 值对，但是参数的顺序极为关键。接收到方法调用后，rpcrouter将按接收顺序依次打开参数，并试图找到相匹配的方法签名。例如，带两个参数 String firstName 和 int age 的 SOAP 请求试图找到 methodName (String, int)。如果客户端颠倒了参数的顺序，就会产生“ no signature match（找不到相匹配的签名）”错误。

远程服务调用

Call 对象一旦被设定，我们就可以通过 invoke( )方法执行远程服务了。invoke()方法有两个参数：

SOAP 服务器的 URL

对于 Apache 发布版，这是连接到 rpcrouter servlet 的绝对 URL，如 *http://localhost:8080/soap/servlet/rpcrouter*。

SOAPAction 首部

根据SOAP规范，通过HTTP发送SOAP请求的客户端应用必须有一个HTTP首部 —— SOAPAction首部。SOAPAction一般用来标明SOAP服务的URI，

> 但是，空字符串（""）表示SOAP目标在HTTP请求URI中已经指定。Apache SOAP服务器实现要求你指定一个SOAPAction首部，但它会忽略其真实值，因此你可以放心地使用空字符串或 null 值。

我们的客户端代码指定了localhost Apache 服务器和一个空字符串SOAPAction首部：

```
URL url = new URL ("http://localhost:8080/soap/servlet/rpcrouter");
Response resp = call.invoke (url, "");
```

如果一切进展顺利，invoke()方法将返回一个org.apache.soap.rpc.Response对象。Response对象将按照服务器SOAP响应封装所有数据，包括所有返回参数和错误情况。

检查错误

SOAP 赋予的分布式计算天生就易受多种故障的破坏，如：

- SOAP 服务器在碰到大量的事务处理时可能会跟不上，甚至死机。
- SOAP 服务器可能不能完成请求的服务。
- SOAP 客户端可能不能打开网络连接。
- SOAP 客户端可能与 SOAP 服务器不兼容。

SOAP 错误分为两类：SOAPException 和 SOAP 故障。SOAPException 指网络连接时的致命错误或违背了 SOAP 协议。例如，当 SOAP 服务器返回一个 SOAP 响应时没有包含所需的 Body 元素，客户端检测到违反了协议，立即抛出一个 SOAPException。而 SOAP 故障是指应用层上的错误。例如，当客户端请求一个不存在的服务或方法时，SOAP 服务器会产生一个故障并将它传递回客户端。当远程服务方法不能完成执行时，也会触发一个故障。

SOAPException 由 Call.invoke()方法抛出，而 SOAP 故障被嵌入 Response 对象，需要显式地提取。

API: org.apache.soap.SOAPException

`String getFaultCode()`
返回 SOAP 故障代码，标识出错误的主要原因。返回值 `SOAP-ENV:Client`表明错误由客户端引起 ;返回值`SOAP-ENV:Server`表明错误由服务器引起。

`String getMessage()`
返回对错误的可读解释。

API: org.apache.soap.Fault

`String getFaultCode()`
返回 SOAP 故障代码，标识出错误的主要原因。返回值 `SOAP-ENV:Client`表明错误由客户端引起 ;返回值`SOAP-ENV:Server`表明错误由服务器引起。

`String getFaultString()`
返回对错误的可读解释。

`HelloClient.java`包含捕获`SOAPException`和 SOAP故障的代码，如`main()`方法捕获 `SOAPException`并显示错误的原因。

```
catch (SOAPException e) {
     String faultCode = e.getFaultCode();
     String faultMsg = e.getMessage();
     System.err.println ("SOAPException Thrown (details below):");
     System.err.println ("FaultCode: "+faultCode);
     System.err.println ("FaultMessage: "+faultMsg);
}
```

此故障代码指出了错误的起源。`SOAP-ENV:Client` 的返回值表示客户端引起错误，而 `SOAP-ENV:Server` 的返回值表示服务器引起错误。

`HelloClient.java`还通过检查`Response.generatedFault()`方法检测SOAP故障。如果这个方法的返回值为真，则代码提取 `Fault` 对象并查询其细节。

```
Fault f = resp.getFault();
String faultCode = f.getFaultCode();
```

```
String faultString = f.getFaultString();
System.err.println("Fault Occurred (details follow):");
System.err.println("Fault Code: "+faultCode);
System.err.println("Fault String: "+faultString);
```

提取返回值

最后一步是提取返回值。为此，我们可以调用`Response.getReturnValue()`方法，再调用`Parameter.getValue()`，然后传送给预定的类：

```
if (!resp.generatedFault()) {
  // 提取返回值
  Parameter result = resp.getReturnValue();
  String greeting = (String) result.getValue();
  return greeting;
}
```

API: org.apache.soap.rpc.Response

`boolean generatedFault()`
表明是否有故障发生。如果该方法返回`true`，就用`getFault()`获取内嵌的故障。

`Fault getFault()`
获取内嵌的`Fault`对象。

`Parameter getReturnValue()`
获取返回参数。

部署 SOAP 服务

部署新的 SOAP 服务的方法有两种：使用基于 Web 的管理器和使用命令行工具。

基于 Web 的管理器

要使用基于 Web 的管理器，需要打开一个新的浏览器，链接到 *http://localhost:8080/soap*，单击 Run the Admin Client 链接，随即出现如图 4-4 所示的窗口，它提供了三个基本工具：

List

获取所有已部署服务的完整列表

Deploy

部署一个新的 SOAP 服务

UnDeploy

取消一个已有的 SOAP 服务的部署

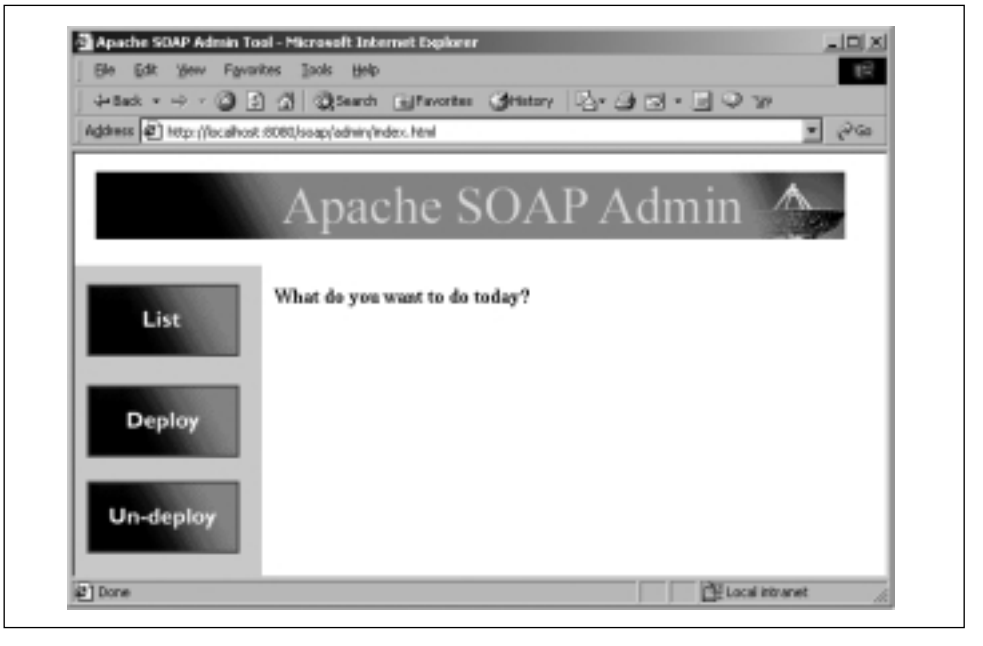

图 4-4：Apache SOAP 管理客户端

要部署 `HelloService` 类，请单击 Deploy 按钮。

Deploy a Service 页包含 6 个用于部署 Web 服务的字段（见图 4-5）。我们集中讨论最重要的字段：

ID

该字段设置 SOAP 服务的名称。建议你命名 SOAP 服务时使用 URN 语法（要想了解 URN 的详细内容，请参阅后面的选读部分）。在第一个例子中，我们把 ID 设为 `urn:examples:helloservice`。

统一资源名（URN）

统一资源名（Uniform Resource Name，URN）是长期有效并与位置无关的通用资源标识符（Uniform Resource Identifier，URI）。在 IETF RFC 2141 中详细描述了正式 URN 语法：

```
<URN> ::= "urn:" <NID> ":" <NSS>
```

其中 `<NID>` 是名称空间标识符，`<NSS>` 名称空间特定的字符串。例如，`urn:isbn:0596000588`指的是由O'Reilly公司出版的《XML in a Nutshell》。

图 4-5：Apache SOAP：Deploy a Service 页

Scope

每次调用SOAP服务时，都由一个特定的服务器对象处理请求。`Scope`定义了这个实例化对象的生命期。`Request`表示这个对象在一个SOAP请求/响应周期内存在。`Session`表示对象在客户端与服务器之间的整个会话期都存在，因此在多个请求/响应对话中都存在。`Application`表示只有一个对象

创建了实例，这个对象会处理后面的所有请求。在第一个例子中，我们把 Scope 设为 Request。

Methods

这个字段包含一个服务支持的所有方法的完整列表。方法间以空格字符隔开。HelloService 类只支持一个方法：sayHello。

Java provider

Java 提供者是将要处理后面 SOAP 请求的 Java 服务类的全称。在“Hello, World!”中，Java 提供者被设为 com.ecerami.soap.HelloService，注意这个类一定要能通过 Tomcat CLASSPATH 获得。Static 字段表示指定的方法是否为静态，如果设为 Yes，则对象不能创建实例。sayHello()方法不是静态的，因此我们将 Static 字段一直设为默认的 No。

一旦设定好这四个字段，请单击 Deploy 按钮。要想证实你的服务真地部署好了，就单击 List 按钮，你将会看到显示的服务（见图 4-6）。

图 4-6：Apache SOAP：Service Listing 页

ServiceManagerClient 命令行工具

要部署新的 SOAP 服务，还可以使用 Apache 命令行工具：`ServiceManager-Client`。此命令行工具用法如下：

```
Usage: java org.apache.soap.server.ServiceManagerClient
[-auth username:password] url operation arguments
```

它支持下列操作：

`list`
: 提供一个已有 SOAP 服务的完整列表

`deploy deployment-descriptior-file.xml`
: 部署在部署描述文件中指定的 SOAP 服务

`query servicename`
: 显示指定服务的部署描述

`undeploy service-name`
: 取消指定服务的部署

要部署新的 SOAP 服务，你必须指定一个部署描述文件。部署描述文件包含所部署服务的所有信息，包括服务 URN、服务方法列表、作用域和 Java 提供者。例如，下面就是“ Hello, SOAP! ”服务的部署描述文件。

```
<isd:service
   xmlns:isd="http://xml.apache.org/xml-soap/deployment"
   id="urn:examples:helloservice" checkMustUnderstands="false">
   <isd:provider type="java" scope="Request" methods="sayHello">
      <isd:java class="com.ecerami.soap.HelloService" static="false"/>
   </isd:provider>
</isd:service>
```

注意，在部署描述文件中指定的元素和属性与 Web 管理工具的 HTML 表单字段存在一一对应关系：

Service 元素
: 为 SOAP 服务指定 URN

Provider 元素

指定服务提供者的类型、作用域、方法和 Java 类

要通过命令行工具部署“ Hello, SOAP! ”服务，请使用如下命令：

```
java org.apache.soap.server.ServiceManagerClient http://localhost:8080/soap/
    servlet/rpcrouter deploy helloservice.xml
```

要验证服务是否确实已注册，请使用 `list` 操作：

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

你应该看到如下输出：

```
Deployed Services:
urn:examples:helloservice
```

要检索一个已有服务的部署描述文件，请使用 `query` 操作，如下面的命令：

```
java org.apache.soap.server.ServiceManagerClient http://localhost:8080/soap/
    servlet/rpcrouter query urn:examples:helloservice
```

将显示 `helloservice` 的部署描述文件。

TcpTunnelGui 工具

查看客户端和服务器间的实际 SOAP 对话一般非常有用。这有助于理解 SOAP 协议的复杂性和调试运行中的应用程序。另外对你有帮助的是，Apache SOAP 发布版包含一个便利的 `TcpTunnelGui` 工具。这个工具需要三个命令行参数：

`listenport`

`TcpTunnelGui` 工具将截获和显示所有传向 `listenport` 的消息。

`tunnelhost`

隧道主机名。对于本地安装，将此参数设为 `localhost`。

`tunnelport`

`TcpTunnelGui` 工具将截获所有消息，并将它们转发给 `tunnelport`。对于 Jakarta Tomcat，将此参数设为 `8080`。

例如，下面的命令行将截获所有发往端口 8070 的消息，并将它们转发到 `localhost`，端口 8080：

```
java org.apache.soap.util.net.TcpTunnelGui 8070 localhost 8080
```

要查看真实的 SOAP 对话，你必须修改客户端代码，使用端口 8070。例如：

```
URL url = new URL ("http://localhost:8070/soap/servlet/rpcrouter");
Response resp = call.invoke (url, "");
```

图 4-7 提供了“ Hello，SOAP! ”对话样例。来自客户端的消息显示在左边，而来自服务器的消息显示在右边。

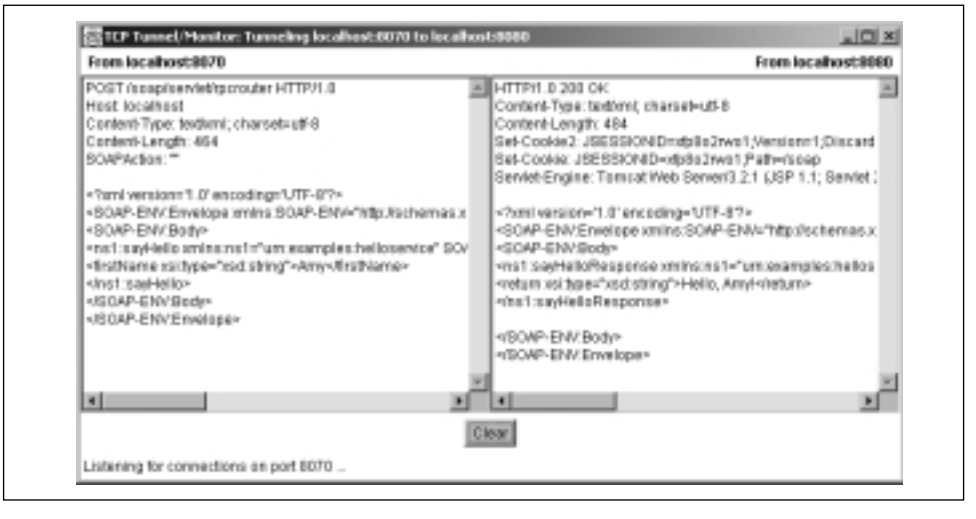

图 4-7：运行中的 `TcpTunnelGui` 工具

下面是一个请求消息样例的全文：

```
POST /soap/servlet/rpcrouter HTTP/1.0
Host: localhost
```

```
Content-Type: text/xml; charset=utf-8
Content-Length: 464
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/1999/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema">
   <SOAP-ENV:Body>
      <ns1:sayHello
         xmlns:ns1="urn:examples:helloservice"
         SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
         <firstName xsi:type="xsd:string">Amy</firstName>
      </ns1:sayHello>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

下面是一个响应消息样例的全文：

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 484
Set-Cookie2: JSESSIONID=810j57jod1;Version=1;Discard;Path="/soap"
Set-Cookie: JSESSIONID=810j57jod1;Path=/soap
Servlet-Engine: Tomcat Web Server/3.2.1
(JSP 1.1; Servlet 2.2; Java 1.3.0; Windows 2000 5.0 x86;
     java.vendor=Sun Microsystems Inc.)

<?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/1999/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema">
   <SOAP-ENV:Body>
      <ns1:sayHelloResponse
         xmlns:ns1="urn:examples:helloservice"
         SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
         <return xsi:type="xsd:string">Hello, Amy!</return>
      </ns1:sayHelloResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Web 资源

下面是一些提供了更多有关 Apache SOAP 信息的 Web 资源：

- Apache SOAP 网站 *http://xml.apache.org/soap/*。
- 邮件列表档案：Apache SOAP 维护着一个“ soap-user ”邮件列表。完整的档案可以从 *http://marc.theaimsgroup.com/?l=soap-user* 获得。

第五章

Apache SOAP 编程

在对 Apache SOAP 有了基本了解之后，我们将进一步探讨以下几个问题：

- 在客户端和服务器之间传送数组；
- 在客户端和服务器之间传送 JavaBean；
- 注册新的类型映射；
- 使用直接 XML 文档；
- 处理 SOAP 故障和异常；
- 维持会话状态。

本章将围绕五个SOAP应用的例子展开讨论，借助于这几个相关的例子可以解释上面列出的问题。一个虚构的电子商务公司在Web上出售软件，前四个例子将说明这个公司提供的服务。最后一个应用例子提供了一个简单的计数服务，以解释如何建立正式的SOAP服务。本章的每个应用都包含完整的客户端和服务代码以及对有关 API 的描述。

注意：本章中的所有例子都使用 HTTP 传输 SOAP 消息，尽管如此，Apache SOAP 也可以用 SMTP 传输 SOAP 消息。

使用数组

第四章中介绍的“ Hello, SOAP! ”应用解释了如何传送字符串和原始数据类型，SOAP的更高境界就是使用数组。幸运的是，Apache SOAP内置了对数组的支持，这使得传送数组变得相对容易一些。

为了说明基本概念，我们将建立一个简单的电子商务产品目录。客户端可以连接到目录服务，发送一组SKU（stockkeeping unit，库存保管单元）。目录服务查找每个SKU，并返回一组当前价格。客户端在幕后向服务器传送一个字符串数组，服务器返回一个双精度数组。

服务代码

我们先看一下服务代码（见例5-1）。`PriceListService`构造函数为两个当前商品创建一个商品散列表。为保持代码简洁，将价格直接写入代码。`getPriceList()`方法需要一个字符串型SKU数组，并产生相应的双精度型数组。我们假定客户端一直请求当前有效的SKU。

例5-1：PriceListService.java

```
package com.ecerami.soap;

import java.util.Hashtable;

/**
 * SOAP服务样例
 * 为指定的SKU列表提供一组价格
 */
public class PriceListService {
  protected Hashtable products; // 产品“数据库”

  /**
   * 零参数构造函数
   * 将两个样例产品载入产品数据库
   */
  public PriceListService () {
    products = new Hashtable();
    // Red Hat Linux
    products.put("A358185", new Double (54.99));
    // McAfee PGP Personal Privacy
```

```
    products.put("A358565", new Double (19.99));
  }

  /**
  * 为指定的SKU提供价格列表
  * 我们假定客户端总是指定有效的当前SKU
  */
  public double[] getPriceList (String sku[]) {
    double prices[] = new double [sku.length];
    for (int i=0; i<sku.length; i++) {
      Double price = (Double) products.get(sku[i]);
      prices[i] = price.doubleValue();
    }
    return prices;
  }
}
```

客户端代码

本节后面的例5-2为客户端代码。在调用客户端应用时，你可以在命令行指定任意数量的SKU。例如下面的命令行：

```
java com.ecerami.soap.PriceListClient A358185 A358565
```

将产生如下输出：

```
Price List Checker: SOAP Client
SKU: A358185 --> 54.99
SKU: A358565 --> 19.99
```

查看客户端代码时，首先要注意`TargetObjectURI`和方法名：

```
call.setTargetObjectURI ("urn:examples:pricelistservice");
call.setMethodName ("getPriceList");
```

这里我们假定`pricelistservice`已经用Web管理工具部署好了。你也可以用命令行工具和如下所示的部署描述文件进行部署：

```
<isd:service
   xmlns:isd="http://xml.apache.org/xml-soap/deployment"
   id="urn:examples:pricelistservice" checkMustUnderstands="false">
   <isd:provider type="java" scope="Request" methods="getPriceList">
```

```
        <isd:java class="com.ecerami.soap.PriceListService" static="false"/>
      </isd:provider>
    </isd:service>
```

要想从客户端向服务器传送数组，必须创建一个新的Parameter对象。重要的区别在于你必须为 Parameter 构造函数指定一个数组类，如 String[].class 或 Double[].class。例如，我们的新客户端创建了一个字符串型 SKU 数组参数：

```
Parameter param = new Parameter("sku", String[].class,
    skus, Constants.NS_URI_SOAP_ENC);
```

然后向参数 Vector 加入此数组参数，Vector 被传递给 Call 对象，和我们的“ Hello, SOAP! ”应用一样。

要从 Response对象中提取数组，只需转换成适当的数组类型。例如PriceList-Client 代码期望一个双精度数组：

```
Parameter result = resp.getReturnValue ();
double priceList[] = (double []) result.getValue();
```

总地来说，传送数组其实并无特别之处。但是，请注意当前 Apache SOAP 版本只支持一维数组。不久的将来一维以上数组将得到支持。请到 Apache SOAP 网站上查看当前版本的注解。

例 5-2：PriceListClient.java

```
package com.ecerami.soap;

/**
 * SOAP 客户端样例
 * 为指定的 SKU 获取价格列表
 * 用法: java PriceClient sku#1 sku#2 sku#N
*/
import java.net.*;
import java.util.Vector;
import org.apache.soap.*;
import org.apache.soap.rpc.*;

public class PriceListClient {

  /**
```

```
 * 静态main方法
 */
public static void main (String[] args) {
  System.out.println ("Price List Checker: SOAP Client");
  String skus[] = new String [args.length];
  for (int i=0; i<args.length; i++)
    skus[i] = new String (args[i]);
  PriceListClient priceListClient = new PriceListClient();
  try {
    double price[] = priceListClient.getPriceList(skus);
    for (int i=0; i<price.length; i++) {
      System.out.print ("SKU: "+skus[i]);
      System.out.println (" --> "+price[i]);
    }
  } catch (SOAPException e) {
    System.err.println (e);
  } catch (MalformedURLException e) {
    System.err.println (e);
  }
}

/**
 * getPriceList方法
*/
public double[] getPriceList (String skus[])
  throws SOAPException, MalformedURLException {
  Parameter skuParam;

  // 创建SOAP RPC Call对象
  Call call = new Call ();

  // 设置编码形式为标准SOAP编码
  call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

  // 设置对象URI和方法名
       call.setTargetObjectURI ("urn:examples:pricelistservice");
   call.setMethodName ("getPriceList");

  // 设置方法参数
      Vector paramList = new Vector ();
   Parameter param = new Parameter("sku", String[].class,
      skus, Constants.NS_URI_SOAP_ENC);
   paramList.addElement (param);
   call.setParams (paramList);

  // 设置Web服务的URL
  URL url = new URL ("http://localhost:8080/soap/servlet/rpcrouter");
```

```
    // 调用服务
    Response resp = call.invoke (url, null);

    // 检查是否成功
    if (!resp.generatedFault()) {
      // 提取返回值
            Parameter result = resp.getReturnValue ();
       double priceList[] = (double []) result.getValue();
      return priceList;
    }
    // 检查故障
    else {
      // 提取故障代码和字符串
      Fault f = resp.getFault();
      String faultCode = f.getFaultCode();
      String faultString = f.getFaultString();
      System.err.println("Fault Occurred (details follow):");
      System.err.println("Fault Code: "+faultCode);
      System.err.println("Fault String: "+faultString);
      return null;
    }
  }
}
```

下面的 SOAP 请求（不含 HTTP 首部）样例全文供参考。请注意数组编码：

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://www.w3.org/2001/09/soap-envelope"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:getPriceList
   xmlns:ns1="urn:examples:pricelistservice"
   SOAP-ENV:encodingStyle="http://www.w3.org/2001/09/soap-encoding">
<sku
   xmlns:ns2="http://www.w3.org/2001/09/soap-encoding"
   xsi:type="ns2:Array" ns2:arrayType="xsd:string[2]">
   <item xsi:type="xsd:string">A358185</item>
   <item xsi:type="xsd:string">A358565</item>
</sku>
</ns1:getPriceList>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

下面是一个完整的 SOAP 响应：

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://www.w3.org/2001/09/soap-envelope"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
      <ns1:getPriceListResponse
         xmlns:ns1="urn:examples:pricelistservice"
         SOAP-ENV:encodingStyle="http://www.w3.org/2001/09/soap-encoding">
         <return
            xmlns:ns2="http://www.w3.org/2001/09/soap-encoding"
            xsi:type="ns2:Array" ns2:arrayType="xsd:double[2]">
            <item xsi:type="xsd:double">54.99</item>
            <item xsi:type="xsd:double">19.99</item>
         </return>
      </ns1:getPriceListResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

使用 JavaBean

迄今为止，你只能使用字符串、原始数据类型和数组。幸好 Apache SOAP 还支持 JavaBean 和直接 XML 文档。根据正式的 JavaSoft 文档，JavaBean 是一个可重复使用的软件组件，可用任何构建工具进行可视化操作。但是，更一般地说，JavaBean 指的是遵从 JavaBean 命名约定的任何一个 Java 类。此约定要求所有可存取的属性都可通过 `get/set` 方法获得。如 `Color` 属性一定要有一个相应的 `getColor()/setColor()`方法对。`boolean` 属性例外，它需要的是 `is/get` 命名约定。JavaBean 约定还要求提供一个零参数构造函数。

通过使用 Java 反射，可视工具可确定可用的 Bean 属性，并通过易用文本框和单选按钮获取这些属性。在同样的行，Apache SOAP 内置的 `BeanSerializer` 类可以用反射将任何 JavaBean 转换为 XML 元素，或接受 XML 元素并自动建立一个相应的 JavaBean。这要求客户端和服务代码都能访问 JavaBean 的类文件。SOAP 调用其实并不下载 JavaBean 的代码，只是获得 JavaBean 的状态。理解这些工作机制需要更详细地了解 Java 向 XML 的转换和 SOAP 服务部署选项。

ProductBean

为了说明最重要的有关 JavaBean 的概念，我们的第二个 SOAP 例子能检索到 JavaBean 的完整状态。我们现在需要的不仅是产品价格，而且要知道产品名称、描述和价格，并且要将全部数据封装在一个 JavaBean 中。如下面的命令行：

```
java com.ecerami.soap.ProductClient A358565
```

此程序产生如下输出：

```
Product Checker: SOAP Client
SKU: A358565
Name: McAfee PGP
Description: McAfee PGP Personal Privacy
Price: 19.99
```

第一步是创建一个名为ProductBean的产品JavaBean（参见例5-3）。ProductBean有四个属性：名称、描述、价格和SKU。每个属性都有get/set方法，我们还提供一个零参数构造函数，使之成为有效的JavaBean。

例 5-3：ProductBean.java

```
package com.ecerami.soap;

/**
 * 一个产品Bean
 * 封装关于某产品的数据
*/

public class ProductBean {
  private String name;        // 产品名称
  private String description; // 产品描述
  private double price;       // 产品价格
  private String sku;         // 产品SKU

  /**
   * 零参数构造函数
   */
  public ProductBean () { }

  /**
   * 带有全部参数的构造函数
   */
```

```
  public ProductBean (String name, String description, double price,
    String sku) {
    this.name = name;
    this.description = description;
    this.price = price;
    this.sku = sku;
  }

  // 设置方法
  public void setName (String name) {
    this.name = name;
  }

  public void setDescription (String description) {
    this.description = description;
  }

  public void setPrice (double price) {
    this.price = price;
  }

  public void setSKU (String sku) {
    this.sku = sku;
  }

  // 获取方法
  public String getName () { return name; }
  public String getDescription () { return description; }
  public double getPrice () { return price; }
  public String getSKU () { return sku; }
}
```

服务代码

接下来需要编写服务代码（见例5-4）。`ProductService`构造函数创建了两个样例`ProductBean`并把它们载入产品散列表中。`getProduct()`方法搜索特定的SKU字符串, 返回相应的`ProductBean`。我们再一次假定用户总是指定一个有效的当前SKU。

例5-4：ProductService.java

```
package com.ecerami.soap;
```

```
import java.util.Hashtable;

/**
 * SOAP 服务样例
 * 为所请求的库存保管单元（SKU）提供产品信息
 */
public class ProductService {
 protected Hashtable products; // 产品“数据库”

 /**
  * 构造函数
  * 把两个样例产品载入产品数据库
  */
 public ProductService () {
   products = new Hashtable();
   ProductBean product1 = new ProductBean
     ("Red Hat Linux", "Red Hat Linux Operating System",
     54.99, "A358185");
   ProductBean product2 = new ProductBean
     ("McAfee PGP", "McAfee PGP Personal Privacy",
     19.99, "A358565");
   products.put(product1.getSKU(), product1);
   products.put(product2.getSKU(), product2);
 }

 /**
 * 为所请求的 SKU 提供产品信息
 * 我们假定客户端总是指定有效的当前 SKU
 */
 public ProductBean getProduct (String sku) {
   ProductBean product = (ProductBean) products.get(sku);
   return product;
 }
}
```

可以按前一个例子的方法配置 `ProductService`，但是，你要为类型映射注册表多填写几个字段。Apache SOAP 类型映射注册表提供了一种向 Java 类映射 XML Schema 数据类型或相反的方法。默认情况下，注册表预生成了基本数据类型，包括原始数据类型、字符串、向量、日期和数组。如果要传送新的数据类型，你需要显式注册这个新类型，并注明用哪个 Java 类串行化或解串行化这个新类型。

要注册新的映射，滚动到 Deploy a Service 页面的底部（见图 5-1）。

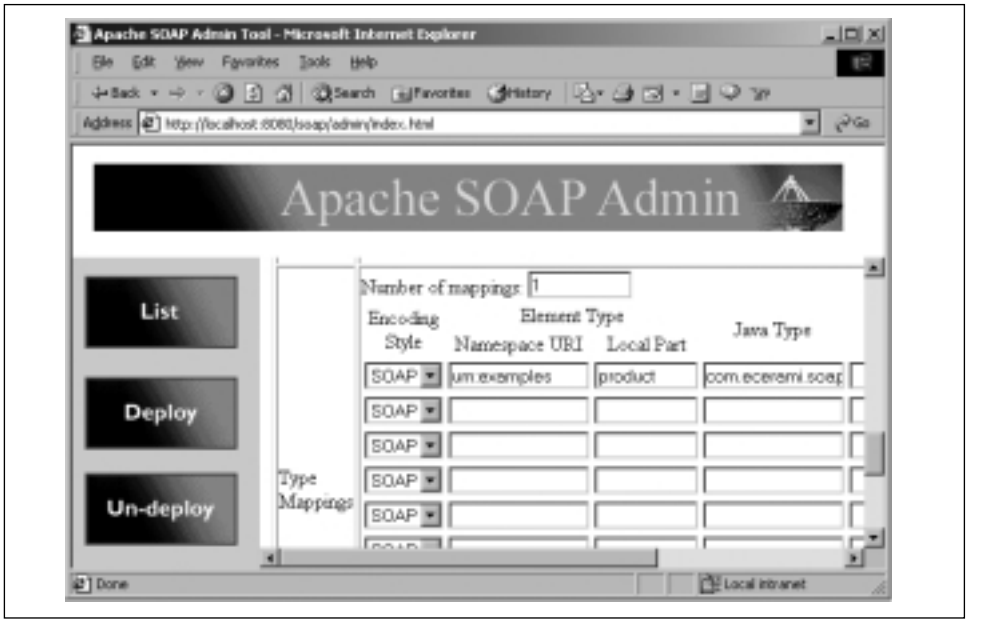

图 5-1：用新类型映射项部署 SOAP 服务

在 Number of mappings 的后面填上新映射总数。在表单相应的框中填上每个映射的正确数据。也可以用命令行工具部署服务。要在部署描述文件中设置类型映射，请使用 `isd:mappings` 元素。`isd:mappings` 元素需要一个包含若干属性的 `isd:map` 元素，属性又正好与 Web 管理工具对应。属性如下表所述：

Web 管理字段	`isd:map` 属性	描述
编码形式	`encodingStyle`	这是新数据类型的编码形式，如： `encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"` 对于 JavaBean，使用默认的 SOAP 编码形式
元素类型：名称空间 URI	`xmlns:x`	如果严格遵守 URN 语法，那么这就是 URN 名称空间的标识符。例如，当使用 `urn:examples:productservice` 时，这个字段应被设为 `urn:examples`

Web 管理字段	isd:map 属性	描述
元素类型：本地	qname	这是新数据类型的名称。名称应该能足以传达所封装数据的描述信息。例如，将这个字段设为 product
Java 类型	javaType	这是新 Java 类的完全限定名。在我们的例子中，它就是 ProductBean:com.ecerami.soap.ProductBean
Java 到 XML 串行化器	java2XMLClassName	这是负责将你的 Java 类转变成 XML 的 Java 类的完全限定名。在前面我们就注意到，Apache SOAP 含有一个能串行化任意 JavaBean 的内置 BeanSerializer，完全限定名是 org.apache.soap.encoding.soapenc.BeanSerializer
XML 到 Java 解串行化器	xml2JavaClassName	这是负责获得 XML 元素和重建 Java 对象的 Java 类的完全限定名。Apache BeanSerializer还能解串行化，因此，对于 JavaBean，最后的两个字段经常设为相同的值

如果用命令行工具部署 SOAP 服务，那么下面是完整的部署描述文件：

```
<isd:service
   xmlns:isd="http://xml.apache.org/xml-soap/deployment"
   id="urn:examples:productservice" checkMustUnderstands="false">
   <isd:provider type="java" scope="Request" methods="getProduct">
      <isd:java class="com.ecerami.soap.ProductService" static="false"/>
   </isd:provider>
       <isd:mappings defaultRegistryClass="">
     <isd:map
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       xmlns:x="urn:examples"
       qname="x:product"
       javaType="com.ecerami.soap.ProductBean"
       xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
       java2XMLClassName=
           "org.apache.soap.encoding.soapenc.BeanSerializer"/>
   </isd:mappings>
</isd:service>
```

如果用Web管理工具部署服务，那么需要验证新映射是否已经注册。单击List服务，再单击`productservice`链接。在服务细节页面的底部，你会看到新类型映射（见图5-2）。

图5-2：新注册的类型映射

客户端代码

现在，我们可以用与在服务器代码中注册新类型映射相同的方法在客户端代码中注册新的类型映射。客户端代码稍复杂一些，但要记住在服务器代码和在客户端代码中注册新映射是一样的。惟一的区别是在服务器代码中，可以通过简单的Web部署工具很容易地增加新映射，而在客户端代码中添加新映射则需要了解更多的SOAP API基本知识。

本节后面的例5-5给出了完整的客户端代码。设定了`TargetObjectURI`和方法名之后，我们开始注册新类型映射。

首先，创建一个新 BeanSerializer 对象的实例：

```
BeanSerializer bSerializer = new BeanSerializer();
```

这与我们部署服务器代码时用的类相同。这个类用于 Java 到 XML 的串行化和 XML 到 Java 的解串行化。

其次，需要创建一个新的 SOAP 映射注册表：

```
SOAPMappingRegistry registry = new SOAPMappingRegistry();
```

它创建了一个默认的注册表对象，该对象已用字符串、原始数据类型和数组的映射预生成。

第三，必须创建一个Qualified Namespace对象。这个对象必须与名称空间URI和服务部署专用的本地字段相匹配。在我们的产品参数例子中，这些字段确实匹配：

```
QName qname = new QName ("urn:examples", "product");
```

第四，必须调用 mapTypes()方法来为 ProductBean 注册新映射：

```
registry.mapTypes (Constants.NS_URI_SOAP_ENC, qname,
   com.ecerami.soap.ProductBean.class, bSerializer,
   bSerializer);
```

mapTypes()方法的参数与 Deploy a Service 页上的编码类型、限定名、Java 到 XML 串行化器及 XML 到 Java 解串行化器等字段正好相对应。

注册新类型的最后一步是将我们的新SOAP映射注册表与当前Call对象相关联：

```
call.setSOAPMappingRegistry(registry);
```

一旦注册成功，所有与urn:examples:product相对应的XML元素都会被自动重建为相应的 ProductBean。通过适当的转换，就可从 Response 对象中检索 ProductBean 了：

```
    Parameter result = resp.getReturnValue ();
    ProductBean product = (ProductBean) result.getValue();
```

例 5-5：ProductClient.java

```
package com.ecerami.soap;

/**
 * SOAP 客户端样例
 * 为指定的库存保管单元（SKU）获取产品信息
 * 用法: java ProductClient sku#
*/
import java.net.*;
import java.util.Vector;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.soapenc.BeanSerializer;
import org.apache.soap.encoding.SOAPMappingRegistry;
import org.apache.soap.util.xml.QName;

public class ProductClient {

  /**
   * 静态 main 方法
   */
  public static void main (String[] args) {
    String sku = args[0];
    System.out.println ("Product Checker: SOAP Client");
    ProductClient productClient = new ProductClient();
    try {
      ProductBean product = productClient.getProduct (sku);
      System.out.println ("SKU: "+product.getSKU());
      System.out.println ("Name: "+product.getName());
      System.out.println ("Description: "+product.getDescription());
      System.out.println ("Price: "+product.getPrice());
    } catch (SOAPException e) {
      System.err.println (e);
    } catch (MalformedURLException e) {
      System.err.println (e);
    }
  }

  /**
   * getProduct 方法
  */
  public ProductBean getProduct (String sku)
    throws SOAPException, MalformedURLException {
    Parameter skuParam;
```

```
// 创建SOAP RPC Call对象
Call call = new Call ();

// 设置编码形式为标准SOAP编码
call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

// 设置对象URI和方法名
call.setTargetObjectURI ("urn:examples:productservice");
call.setMethodName ("getProduct");

// 为ProductBean添加JavaXML映射
// 首先, 创建一个BeanSerializer
BeanSerializer bSerializer = new BeanSerializer();

// 其次, 获取当前SOAPMappingRegisty
// 该对象被预注册为基本映射
SOAPMappingRegistry registry = new SOAPMappingRegistry();

// 第三, 为ProductBean创建一个新的限定名称空间
QName qname = new QName ("urn:examples", "product");

// 第四, 为ProductBean注册新映射
registry.mapTypes (Constants.NS_URI_SOAP_ENC, qname,
   com.ecerami.soap.ProductBean.class, bSerializer,
   bSerializer);

// 第五, 为Call对象设置MappingRegistry
call.setSOAPMappingRegistry(registry);

// 设置方法参数
Vector paramList = new Vector ();
skuParam = new Parameter("sku", String.class,
  sku, Constants.NS_URI_SOAP_ENC);
paramList.addElement (skuParam);
call.setParams (paramList);

// 设置Web服务的URL
URL url = new URL ("http://localhost:8080/soap/servlet/rpcrouter");

// 调用服务
Response resp = call.invoke (url, null);

// 检查是否成功
if (!resp.generatedFault()) {
  // 提取返加值
     Parameter result = resp.getReturnValue ();
   ProductBean product = (ProductBean) result.getValue();
```

```
      return product;
    }
    // 检查故障
    else {
      // 提取故障代码和字符串
      Fault f = resp.getFault();
      String faultCode = f.getFaultCode();
      String faultString = f.getFaultString();
      System.err.println("Fault Occurred (details follow):");
      System.err.println("Fault Code: "+faultCode);
      System.err.println("Fault String: "+faultString);
      return null;
    }
  }
}
```

下面的 SOAP 请求（不含 HTTP 首部）样例全文供参考：

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://www.w3.org/2001/09/soap-envelope"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
      <ns1:getProduct
         xmlns:ns1="urn:examples:productservice"
         SOAP-ENV:encodingStyle="http://www.w3.org/2001/09/soap-encoding">
         <sku xsi:type="xsd:string">A358185</sku>
      </ns1:getProduct>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

下面是 SOAP 响应样例：

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://www.w3.org/2001/09/soap-envelope"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
      <ns1:getProductResponse
         xmlns:ns1="urn:examples:productservice"
         SOAP-ENV:encodingStyle="http://www.w3.org/2001/09/soap-encoding">
         <return xmlns:ns2="urn:examples" xsi:type="ns2:product">
            <name xsi:type="xsd:string">Red Hat Linux</name>
            <price xsi:type="xsd:double">54.99</price>
```

```
            <description xsi:type="xsd:string">
               Red Hat Linux Operating System
            </description>
            <SKU xsi:type="xsd:string">A358185</SKU>
         </return>
      </ns1:getProductResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

API: org.apache.soap.rpc.Call

`void setSOAPMappingRegistry(SOAPMappingRegistry smr)`

设置SOAP映射注册表。当需要串行化/解串行化你自己的Java类时这会很有用。取以下参数：

`smr`

SOAP 映射表。

使用直接 XML 文档

除了JavaBean以外，Apache SOAP还支持直接XML文档传送。我们的第三个例子将解释这一主要概念。客户端代码将通过用SKU属性发送产品元素来发送产品查询，如：

```
<product sku="A358185"/>
```

服务器将用一个完整的产品 XML 文档来响应，如下面的命令行：

```
java com.ecerami.soap.ProductXMLClient A358185
```

将产生如下 XML 响应：

```
<product sku="A358185">
   <name>Red Hat Linux</name>
   <description>Red Hat Linux Operating System</description>
   <price>54.99</price>
</product>
```

API: org.apache.soap.encoding.SOAPMappingRegistry

`SOAPMappingRegistry()`

构建一个新的`SOAPMappingRegistry`对象。该对象将被预置成基本数据类型的映射，包括原始数据类型、字符串、向量、日期和数组。

`void mapTypes(String encodingStyleURI, QName elementType, Class javaType, Serializer s, Deserializer ds) ()`

注册新类型映射。取以下参数：

`encodingStyleURI`

新数据类型的编码形式。要使用默认的SOAP编码形式，请使用`Constants.NS_URI_SOAP_ENC`。

`name`

新数据类型的限定名称。

`javaType`

Java类的类型，例如，`ProductBean.class`。

`s`

该Java类负责将Java类串行化为XML。对于JavaBean使用`BeanSerializer`。

`ds`

该Java类负责将XML解串行化为Java类。对于JavaBean使用`BeanSerializer`。

使用直接XML文档需要具备有关XML DOM（Document Object Model，文档对象模式）API的知识。但是，即使不熟悉DOM API，你也可以顺利读懂示例代码的一般流程。

服务代码

例5-6列出了完整的服务代码。注意，`ProductXMLService`类由前面样例中的`ProductService`扩展而成，因此使用了相同的产品散列表。还要注意新的`getProduct()`方法接受一个DOM `Element`，并返回一个DOM `Element`。在这个方法里，我们先提取SKU属性代码：

```
String sku = request.getAttribute("sku");
```

接着我们查找产品散列表。如果找到相匹配的，我们就通过DOM API建立一个完整的XML文档。为方便起见，可以用Apache实用方法来检索DocumentBuilder对象：

```
DocumentBuilder docBuilder = XMLParserUtils.getXMLDocBuilder();
Document doc = docBuilder.newDocument();
```

我们可以用手头上的文档继续增加适当的XML元素层次。例如，下面的代码用相应的文本子元素创建了一个产品名元素：

```
Text productNameText = doc.createTextNode(product.getName());
Element nameNode = doc.createElement("name");
nameNode.appendChild(productNameText);
```

例5-6：ProductXMLService.java

```
package com.ecerami.soap;

import java.util.Hashtable;
import org.w3c.dom.*;
import javax.xml.parsers.DocumentBuilder;
import org.apache.soap.util.xml.XMLParserUtils;

/**
 * SOAP服务样例
 * 为所请求的SKU提供产品名称
 * 信息作为直接XML文档被传递
 */
public class ProductXMLService extends ProductService{

  /**
  * 为所请求的XML文档提供产品信息
  */
     public Element getProduct (Element request)
    throws ProductNotFoundException {
    // 提取SKU属性
         String sku = request.getAttribute("sku");
    ProductBean product = (ProductBean) products.get(sku);

    // 创建XML文档以保存产品数据
        DocumentBuilder docBuilder = XMLParserUtils.getXMLDocBuilder();
    Document doc = docBuilder.newDocument();
```

```
    // 创建产品名称元素
    Text productNameText = doc.createTextNode(product.getName());
    Element nameNode = doc.createElement("name");
    nameNode.appendChild(productNameText);

    // 创建产品描述元素
    Text productDescriptionText =
      doc.createTextNode(product.getDescription());
    Element descriptionNode = doc.createElement("description");
    descriptionNode.appendChild(productDescriptionText);

    // 创建产品价格元素
    Text productPriceText = doc.createTextNode(
      Double.toString(product.getPrice()));
    Element priceNode = doc.createElement("price");
    priceNode.appendChild(productPriceText);

    // 创建根产品元素
    Element productNode = doc.createElement("product");
    productNode.setAttribute("sku", sku);
    productNode.appendChild(nameNode);
    productNode.appendChild(descriptionNode);
    productNode.appendChild(priceNode);
    return productNode;
  }
}
```

客户端代码

例 5-7 列出了完整的客户端代码。幸运的是，大部分代码都与前面三个例子中的代码相似。与在服务代码中一样，我们将用 DOM API 建立一个新的 XML 文档。

首先，把编码形式设为 XML 直接量：

```
call.setEncodingStyleURI(Constants.NS_URI_LITERAL_XML);
```

这使整个 XML 文档能在 SOAP 请求中传送。

接着，创建一个新的 XML 文档。我们用编写服务器代码时采用的技术来创建一个产品元素（如 `<product sku="A358185"/>`）。使用这个方便的元素，我们将创建一个新的 `Parameter` 对象：

```
skuParam = new Parameter("productNode", org.w3c.dom.Element.class,
    productNode, Constants.NS_URI_LITERAL_XML);
```

注意，我们再一次将编码形式设为 XML 直接量。

像平常一样，最后一步是正确转换返回值。在这个例子里，我们将转换成 DOM `Element` 类：

```
Parameter result = resp.getReturnValue ();
Element xmlResult = (Element) result.getValue();
```

现在，我们可以用 Apache 方便的 `DOM2Writer` 工具输出 `Element` 字符串了：

```
DOM2Writer domWriter = new DOM2Writer();
System.out.println ("Server Response: ");
System.out.println (domWriter.nodeToString(product));
```

例 5-7：ProductXMLClient.java

```
package com.ecerami.soap;

/**
 * SOAP 客户端样例
 * 为指定的 SKU 获取产品信息
 * 数据作为直接 XML 文档被返回
 * 用法: java ProductXMLClient sku#
*/
import java.net.*;
import java.util.Vector;
import org.w3c.dom.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import javax.xml.parsers.DocumentBuilder;
import org.apache.soap.util.xml.XMLParserUtils;
import org.apache.soap.util.xml.DOM2Writer;

public class ProductXMLClient {

  /**
   * 静态 main 方法
   */
  public static void main (String[] args) {
    String sku = args[0];
    System.out.println ("XML Product Checker: SOAP Client");
    ProductXMLClient productXMLClient = new ProductXMLClient();
```

```
    try {
            Element product = productXMLClient.getProduct (sku);
      DOM2Writer domWriter = new DOM2Writer();
        System.out.println ("Server Response: ");
        System.out.println (domWriter.nodeToString(product));
    } catch (SOAPException e) {
      System.err.println (e);
    } catch (MalformedURLException e) {
      System.err.println (e);
    }
  }

  /**
   * getProduct 方法
  */
  public Element getProduct (String sku)
    throws SOAPException, MalformedURLException {
    Parameter skuParam;

    // 创建 SOAP RPC Call 对象
    Call call = new Call ();

    // 设置编码形式为 XML 直接量
    call.setEncodingStyleURI(Constants.NS_URI_LITERAL_XML);

    // 设置对象 URI 和方法名
    call.setTargetObjectURI ("urn:examples:XMLproductservice");
    call.setMethodName ("getProduct");

    // 设置方法参数
    Vector paramList = new Vector ();

    // 创建 XML 文档以保存 SKU
    DocumentBuilder docBuilder = XMLParserUtils.getXMLDocBuilder();
    Document doc = docBuilder.newDocument();

    // 用 SKU 属性创建产品元素
    Element productNode = doc.createElement("product");
    productNode.setAttribute("sku", sku);

    skuParam = new Parameter("productNode", org.w3c.dom.Element.class,
      productNode, Constants.NS_URI_LITERAL_XML);
    paramList.addElement (skuParam);
    call.setParams (paramList);

    // 设置 Web 服务的 URL
    URL url = new URL ("http://localhost:8080/soap/servlet/rpcrouter");
```

```
    // 调用服务
    Response resp = call.invoke (url, null);

    // 检查是否成功
    if (!resp.generatedFault()) {
      // 提取返回值
      Parameter result = resp.getReturnValue ();
      Element xmlResult = (Element) result.getValue();
      return xmlResult;
    }
    // 检查故障
    else {
      // 提取故障代码和字符串
      Fault f = resp.getFault();
      String faultCode = f.getFaultCode();
      String faultString = f.getFaultString();
      System.err.println("Fault Occurred (details follow):");
      System.err.println("Fault Code: "+faultCode);
      System.err.println("Fault String: "+faultString);
      return null;
     }
   }
}
```

下面的完整 SOAP 请求代码供参考：

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://www.w3.org/2001/09/soap-envelope"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
      <ns1:getProduct
         xmlns:ns1="urn:examples:XMLproductservice"
         SOAP-ENV:encodingStyle=
          "http://xml.apache.org/xml-soap/literalxml">
         <productNode>
            <product sku="A358185"/>
         </productNode>
      </ns1:getProduct>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

下面是一个 SOAP 响应样例：

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

```
        xmlns:SOAP-ENV="http://www.w3.org/2001/09/soap-envelope"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <SOAP-ENV:Body>
           <ns1:getProductResponse
              xmlns:ns1="urn:examples:XMLproductservice"
              SOAP-ENV:encodingStyle=
                 "http://xml.apache.org/xml-soap/literalxml">
              <return>
                 <product sku="A358185">
                 <name>Red Hat Linux</name>
                 <description>Red Hat Linux Operating System</description>
                 <price>54.99</price></product>
              </return>
           </ns1:getProductResponse>
        </SOAP-ENV:Body>
     </SOAP-ENV:Envelope>
```

处理 SOAP 故障

正如我们在第四章中说明的那样, SOAP故障表明应用层上出现了错误。例如, 请求一个不存在的服务或方法名将会触发故障。服务对象也会触发故障, 它提供了一个将错误回传给客户端的方法。将异常和错误回传给客户端对建立健壮的应用非常关键。例如, 在已经建立的产品目录应用中, 我们一直假定用户总是传送有效的当前 SKU。当用户请求一个不存在的产品的 SKU 时, 将会发生什么事呢？为了探索这个问题, 从而说明几个故障处理选择, 让我们查看 `ProductService` 样例的一个更新版本。

服务代码

例 5-8 提供了一个完整的服务代码。`ProductService2` 类是由原来的 `ProductService`类扩展而成的, 因此使用了相同的产品散列表。新的`getProductInfo()`方法接收一个 `String` SKU 参数并检查产品散列表。

例 5-8：ProductService2.java

```
package com.ecerami.soap;

import java.util.Hashtable;
```

```
/**
 * SOAP 服务样例
 * 为所请求的 SKU 提供产品信息
 */
public class ProductService2 extends ProductService {

  /**
   * 为所请求的 SKU 提供产品信息
   * 如果未找到 SKU, 则方法抛出一个 ProductNotFoundException 异常
   */
  public ProductBean getProductInfo (String sku)
    throws ProductNotFoundException {
    ProductBean product = (ProductBean) products.get(sku);
        if (product==null)
      throw new ProductNotFoundException ("SKU Not Found: "+sku);
    return product;
  }
}
```

如果找到了相匹配的SKU, `getProductInfo()`方法将返回一个正确的`ProductBean`;否则抛出一个`ProductNotFoundException`。例5-9提供了`ProductNotFoundException`的代码：

例 5-9：ProductNotFoundException.java

```
package com.ecerami.soap;

import org.apache.soap.Fault;

/**
 * ProductNotFoundException
 * 封装与获取指定 SKU 的产品 / 价格
 * 相关的所有异常
*/
public class ProductNotFoundException extends Exception {
  private Fault fault;

  public ProductNotFoundException (String faultString) {
    super (faultString);
  }

  public ProductNotFoundException (String faultString, Fault fault) {
    super (faultString);
    this.fault = fault;
  }
```

```
  public Fault getFault () { return fault; }
}
```

客户端代码

例 5-10 提供了修改后的客户端代码。`getProduct()`方法的大部分与原来的`ProductClient`代码一样，因此这里只列出那些说明故障处理新能力的部分。首先，如果检测到一个故障，客户端代码便提取`Fault`对象，将其嵌入一个`ProductNotFoundException`:

```
Fault fault = resp.getFault();
String faultString = fault.getFaultString();
throw new ProductNotFoundException (faultString, fault);
```

这使得我们今后可以查看`Fault`对象。接着，`main()`方法纳入一个新的`printFaultDetails()`方法，用来输出所有被嵌入`Response`对象的故障细节。但是，查看这个方法前，我们将发送一个不存在的SKU以检测该代码。我们试一下SKU编号 Z358185。如果你用的是 `TcpTunnelGui` 工具，那么你将会看到如下来自 SOAP 服务器的响应：

```
<?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/1999/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <SOAP-ENV:Body>
      <SOAP-ENV:Fault>
         <faultcode>SOAP-ENV:Server</faultcode>
          <faultstring>
             Exception from service object: SKU Not Found: Z358185
          </faultstring>
          <faultactor>/soap/servlet/rpcrouter</faultactor>
      </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

我们看到，并不直接将`ProductNotFoundException`传回客户端，而是由`rpcrouter`捕获这个异常，产生一个 SOAP `Fault`元素，并将内嵌的`ProductNotFoundException`消息插入`faultstring`子元素。在客户端，我们捕获故障，并抛出我们自己的`ProductNotFoundException`。

SOAP规范还允许SOAP响应包含故障细节，故障细节可以提供详尽的错误描述，因此有助于调试程序。要访问故障细节列表，使用Fault getDetailEntries()方法，它将返回一个DOM Elements向量。于是你可以查询每个元素的名称和值：

```
for (int i=0; i< detailEntries.size(); i++) {
    Element detail = (Element) detailEntries.elementAt(i);
    String name = detail.getNodeName();
    String value = DOMUtils.getChildCharacterData(detail);
    System.err.println (name);
    System.err.println (value);
}
```

一般情况下，SOAP响应不包含故障细节，以上方法也不会打印任何东西。但是，Apache SOAP确实提供了两个可以提供有用的故障细节的内置故障监听器。第一个故障监听器DOMFaultListener将所有关于异常的堆栈记录插入服务对象；第二个故障监听器ExceptionFaultListener将插入所抛出异常的名字。但是，不能通过Web管理工具设置默认的监听器，因此，你必须使用命令行工具和isd:faultListener元素。例如，向部署描述文件中加入如下行将增加DomFault-Listener：

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

设置好DomFaultListener，服务器会产生如下错误：

```
<?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/1999/XMLSchema-instance" xmlns:xsd="http://www.
    w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<SOAP-ENV:Fault>
<faultcode>SOAP-ENV:Server</faultcode>
<faultstring>Exception from service object: SKU Not Found: Z358185</
    faultstring>
<faultactor>/soap/servlet/rpcrouter</faultactor>
<detail>
<stackTrace>com.ecerami.soap.ProductNotFoundException: SKU Not Found:
    Z358185
```

```
        at com.ecerami.soap.ProductService2.getProductInfo(ProductService2.java:
          26)
        at java.lang.reflect.Method.invoke(Native Method)
        at org.apache.soap.server.RPCRouter.invoke(RPCRouter.java:146)
        at org.apache.soap.providers.RPCJavaProvider.invoke(RPCJavaProvider.
          java:129)
        at org.apache.soap.server.http.RPCRouterServlet.doPost(RPCRouterServlet.
          java:286)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:760)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
        at org.apache.tomcat.core.ServletWrapper.doService(ServletWrapper.java:
          404)
        at org.apache.tomcat.core.Handler.service(Handler.java:286)
        at org.apache.tomcat.core.ServletWrapper.service(ServletWrapper.java:
          372)
        at org.apache.tomcat.core.ContextManager.internalService(ContextManager.
          java:797)
        at org.apache.tomcat.core.ContextManager.service(ContextManager.java:
          743)
        at org.apache.tomcat.service.http.HttpConnectionHandler.
           processConnection(HttpConnectionHandler.java:210)
        at org.apache.tomcat.service.TcpWorkerThread.runIt(PoolTcpEndpoint.java:
          416)
        at org.apache.tomcat.util.ThreadPool$ControlRunnable.run(ThreadPool.
          java:498)
        at java.lang.Thread.run(Thread.java:484)
    </stackTrace>
    </detail>
    </SOAP-ENV:Fault>
    </SOAP-ENV:Body>
    </SOAP-ENV:Envelope>
```

你会发现，以上错误现在包含`detail`元素，并且它包含一个`stackTrace`元素。用例5-10所示的代码编写的`printFaultDetails()`方法能检测到一个细节条目，并能打印出整个堆栈记录。

例 5-10：ProductClient2.java

```
package com.ecerami.soap;

/**
 * SOAP 客户端样例
 * 为指定的 SKU 获取产品信息
 * 用法: java ProductClient sku#
*/
import java.net.*;
```

```
import java.util.Vector;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.soapenc.BeanSerializer;
import org.apache.soap.encoding.SOAPMappingRegistry;
import org.apache.soap.util.xml.QName;
import org.w3c.dom.Element;
import org.apache.soap.util.xml.DOMUtils;

public class ProductClient2 {

  /**
   * 静态main方法
   */
  public static void main (String[] args) {
    String sku = args[0];
    System.out.println ("Product Checker: SOAP Client");
    ProductClient2 productClient2 = new ProductClient2();
    try {
      ProductBean product = productClient2.getProduct (sku);
      System.out.println ("SKU: "+product.getSKU());
      System.out.println ("Name: "+product.getName());
      System.out.println ("Description: "+product.getDescription());
      System.out.println ("Price: "+product.getPrice());
    } catch (ProductNotFoundException e) {
      System.err.println (e);
      printFaultDetails (e.getFault());
    } catch (SOAPException e) {
      System.err.println (e);
    } catch (MalformedURLException e) {
      System.err.println (e);
    }
  }

  /**
  * 提取并打印故障细节
  */
    public static void printFaultDetails (Fault fault) {
    //提取细节条目
    Vector detailEntries = fault.getDetailEntries();
    if (detailEntries != null) {
      //打印每个细节条目
      for (int i=0; i< detailEntries.size(); i++) {
        Element detail = (Element) detailEntries.elementAt(i);
        String name = detail.getNodeName();
        String value = DOMUtils.getChildCharacterData(detail);
        System.err.println (name);
        System.err.println (value);
```

```
      }
  }
  }

  /**
   * getProduct 方法
  */
  public ProductBean getProduct (String sku)
  throws SOAPException,MalformedURLException,ProductNotFoundException {
  Parameter skuParam;

  ... 同ProductClient.java

  // 检查是否成功
  if (!resp.generatedFault()) {
    // 提取返回值
    Parameter result = resp.getReturnValue ();
    ProductBean product = (ProductBean) result.getValue();
    return product;
  }
  // 检查故障
  else {
    // 提取故障代码和字符串
    Fault fault = resp.getFault();
    String faultString = fault.getFaultString();
    throw new ProductNotFoundException (faultString, fault);
    }
  }
}
```

维持会话状态

我们的最后一个话题是维持会话状态。回顾第四章，每个部署的SOAP服务都有一个相关的`Scope`属性。每当调用一个服务时，`rpcrouter` servlet将调用远程服务对象，`Scope`属性会定义这个远程对象的生命期。

Request

表明这个对象在一个SOAP请求/响应周期内存在。

Session

表明`rpcrouter`将为每个客户端创建一个对象实例，并在多个请求/响应对话中保留这些对象。

Application

表明只创建了一个对象实例，这个对象将会处理后面所有的请求。

为具体说明Scope属性，我们的最后一个例子展示了一个会话计数服务。服务器代码保留内存中的当前计数器，并将值返回给客户端。当把Scope设为Session时，rpcrouter将为每个客户端创建一个新的服务对象实例。因此，服务器能为每个客户端维持单独的会话计数。当把Scope设为Application时，rpcrouter将只为一个服务对象创建实例，因此，服务器计算来自所有客户端的总请求数。

服务代码

例 5-11 提供了完整的服务代码。代码维持一个名为 counter 的实例变量。对每个客户端请求，counter 都会增加。

例 5-11：CounterService.java

```
package com.ecerami.soap;

/**
 * SOAP服务样例
 * 说明Session 及 Application Scope
 *
 * 当此服务被部署成Scope="Session"时，
 * 服务器将为每个客户端创建
 * CounterService的一个实例。
 * 然后CounterService将维持每个会话的请求总数。
 *
 * 当此服务被部署成Scope="Application"时，
 * 服务器将只创建CounterService的一个实例。
 * 然后CounterService将维持所有会话的请求总数。
 *
 */
public class CounterService {
  private int counter; // 请求数目

  /**
   * 构造函数
   */
  public CounterService () { counter = 0; }

  /**
```

```
  * 返回请求数目
  */
  public int getCounter () { return ++counter; }
}
```

客户端代码

例 5-12 提供了完整的客户端代码。客户端代码的目的是要调用远程 `getCounter()`方法并检索当前计数值。

例 5-12：CounterClient.java

```
package com.ecerami.soap;

/**
 * SOAP 客户端样例
 * 从 CounterService 获取当前计数器值
 * 说明 Session 及 Application Scope
*/
import java.util.*;
import java.net.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;

public class CounterClient {
    private Call call; // 可重用的 Call 对象

  /**
   * 静态 main 方法
   */
  public static void main (String[] args) {
    System.out.println ("Session/Application Counter: SOAP Client");
    CounterClient counterClient = new CounterClient();
    counterClient.process();
  }

  /**
   * 构造函数
   * 创建可重用的 Call 对象
   */
    public CounterClient () {
    call = new Call();
 }

  /**
```

```
 * 开始计数
 */
public void process () {
  try {
    for (int i=0; i<5; i++) {
      int counter = getCounter ();
      System.out.println ("Counter: "+counter);
    }
  } catch (CounterException e) {
    System.err.println (e);
  } catch (SOAPException e) {
    System.err.println (e);
  } catch (MalformedURLException e) {
    System.err.println (e);
  }
}

/**
 * getCounter方法
*/
public int getCounter ()
  throws SOAPException, MalformedURLException,
      CounterException {

  // 设置编码形式为标准SOAP编码
  call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

  // 设置对象URI和方法名
  call.setTargetObjectURI ("urn:examples:counterservice");
  call.setMethodName ("getCounter");

  // 设置Web服务的URL
  URL url = new URL ("http://localhost:8080/soap/servlet/rpcrouter");

  // 调用服务
  Response resp = call.invoke (url, null);

  // 检查是否成功
  if (!resp.generatedFault()) {
    // 提取返回值
    Parameter result = resp.getReturnValue ();
    Integer counter = (Integer) result.getValue();
    return counter.intValue();
  }
  // 检查故障
  else {
      // 提取故障代码和字符串
```

```
        Fault f = resp.getFault();
        String faultCode = f.getFaultCode();
        String faultString = f.getFaultString();
        throw new CounterException (faultCode+": "+faultString);
    }
  }

  /**
  * CounterException
  * 封装与获取application/session计数器相关的所有异常
  */
  class CounterException extends Exception {
    private String msg;

    public CounterException (String msg) {
      super(msg);
    }
  }
}
```

Apache SOAP 在幕后用 cookie 区分客户端请求。例如，向 `CounterService` 的第一个请求产生如下 HTTP 响应：

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 477
Set-Cookie2: JSESSIONID=tfr2ps35b1;Version=1;Discard;Path="/soap"
Set-Cookie: JSESSIONID=tfr2ps35b1;Path=/soap
Servlet-Engine: Tomcat Web Server/3.2.1 (JSP 1.1; Servlet 2.2; Java 1.3.0;
    Windows 2000 5.0 x86; java.vendor=Sun Microsystems Inc.)
<?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/1999/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema">
   <SOAP-ENV:Body>
      <ns1:getCounterResponse
         xmlns:ns1="urn:examples:counterservice"
         SOAP-ENV:encodingStyle=
            "http://schemas.xmlsoap.org/soap/encoding/">
         <return xsi:type="xsd:int">1</return>
      </ns1:getCounterResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

HTTP 响应的第五行和第六行设置了一个名为 JSESSIONID 的 Jakarta Tomcat 会话 cookie。客户端可以使用这个 cookie 在多个对服务器的调用中维持会话状态。

默认状态下，SOAP Call 对象可识别 cookie，并自动将其送回服务器。惟一的要求是在同一会话的多个请求中，你要重用同一个 Call 对象。例如，我们的客户端代码在构造函数中创建了一个可重用的 Call 对象：

```
public CounterClient () {
    call = new Call();
}
```

为了维持会话，代码只是重用同一个 Call 对象。Call 对象自动返回所有会话 cookie，服务器可以为每个客户端维持独立的计数器。

第四部分

WSDL

第六章

WSDL 精髓

WSDL 是一种规范，它定义了如何用共同的 XML 语法描述 Web 服务。WSDL 描述了四种关键的数据：

- 描述所有公用函数的接口信息；
- 所有消息请求和消息响应的数据类型信息；
- 所使用的传输协议的绑定信息；
- 用来定位指定服务的地址信息。

总之，WSDL 在服务请求者和服务提供者之间提供一个协议，这一点与 Java 接口在客户代码和真正的 Java 对象间提供的协议非常相似。关键的区别是 WSDL 独立于平台和语言，主要（虽然不是专门）用于描述 SOAP 服务。

客户端可以用 WSDL 找到 Web 服务，并调用其任何公用函数。你还可以用可识别 WSDL 的工具自动完成这个过程，使应用程序只需少量甚至不需手工编码就可以容易地连接新服务。WSDL 为描述服务提供了一种共同的语言，并为自动连接服务提供了一个平台，因此，它是 Web 服务结构中的基石。

本章将讨论 WSDL 的各个方面，具体包括以下几个主题：

- WSDL 规范概述，详细解释主要的 WSDL 元素；
- 引导你入门的两个 WSDL 基本例子；
- WSDL 调用工具简介，包括 IBM WSIF（Web Services Invocation Framework，Web 服务调用框架）、SOAP::Lite 以及 The Mind Electric 公司的 GLUE 平台；
- 讨论从已有 SOAP 服务如何自动生成 WSDL 文件；
- 在 WSDL 中使用 XML Schema 类型的综述，包括数组和复杂类型的使用。

WSDL 规范

WSDL 是描述 Web 服务的 XML 语法。这个规范本身分为六个主要的元素：

`definitions`

`definitions` 元素必须是所有 WSDL 文档的根元素。它定义 Web 服务的名称，声明文档其他部分使用的多个名称空间，并包含这里描述的所有服务元素。

`types`

`types` 元素描述在客户端和服务器之间使用的所有数据类型。虽然 WSDL 没有专门被绑定到某个特定的类型系统上，但它以 XML Schema 规范作为其默认的选择。如果服务只用到诸如字符串型或整型等 XML Schema 内置的简单类型，它就不需要 `types` 元素。在本章的最后将全面讨论 `types` 元素和 XML Schema。

`message`

`message` 元素描述一个单向消息，无论是单一的消息请求还是单一的消息响应，它都描述。`message` 元素定义消息名称，它可以包含零个或更多的引用消息参数或消息返回值的消息 `part` 元素。

`portType`

`portType` 元素结合多个 `message` 元素，形成一个完整的单向或往返操作。例如，`portType` 可以将一个请求消息和一个响应消息结合在一起，形成一

个在SOAP服务中普遍使用的单一的请求/响应操作。注意，一个portType可以（并且经常）定义多个操作。

binding

binding元素描述了在Internet上实现服务的具体细节。WSDL包含定义SOAP服务的内置扩展，因此，SOAP特有的信息会转到这里。

service

service元素定义调用指定服务的地址。一般包含调用SOAP服务的URL。

图6-1简洁地表示出WSDL规范，可以帮助你清晰地了解各个元素的意义。在阅读本章剩余的部分时，你可能会回过头来参照这个图。

图6-1：简洁的WSDL规范

除了六个主要元素以外，WSDL规范还定义了下面的实用元素：

documentation

documentation元素用于提供一个可阅读的文档，可以将它包含在任何其他WSDL元素中。

import

import元素用于导入其他WSDL文档或XML Schema，实现了更多WSDL文档的模块化。譬如，为了使同一个服务在两个物理地址上都可获得，两个

WSDL 文档可以导入相同的基本元素，同时还可以包含它们自己的 service 元素。注意，目前并非所有的 WSDL 工具都支持这种导入功能。

注意：WSDL 不是 W3C 的正式建议，在 W3C 中没有正式地位。WSDL 版本 1.1 于 2001 年 3 月被提交到 W3C，当时的提交者包括 IBM、微软、Ariba 和其余六个公司。新成立的 W3C Web Services Activity 的 Web 服务描述工作组极有可能考虑 WSDL，并将决定是否将其提升到正式的建议地位。WSDL 版本 1.1 规范可以从 *http://www.w3.org/TR/wsdl* 在线获得。

基本的 WSDL 例子：HelloService.wsdl

为了使前面描述的 WSDL 概念尽可能具体，我们来看看第一个 WSDL 文件样例。

例 6-1 为一个文档样例 *HelloService.wsdl*，它描述了第四章介绍的 HelloService。

也许你还记得，HelloService 服务提供了一个名为 *sayHello* 的公用函数。这个函数只需一个字符串型参数，并返回一个字符串型问候。如果你传送一个参数 world，服务就会返回一个问候：“ Hello world! ”。

例 6-1：HelloService.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloService"
   targetNamespace="http://www.ecerami.com/wsdl/HelloService.wsdl"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://www.ecerami.com/wsdl/HelloService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <message name="SayHelloRequest">
      <part name="firstName" type="xsd:string"/>
   </message>
   <message name="SayHelloResponse">
      <part name="greeting" type="xsd:string"/>
   </message>

   <portType name="Hello_PortType">
      <operation name="sayHello">
```

```
         <input message="tns:SayHelloRequest"/>
         <output message="tns:SayHelloResponse"/>
      </operation>
   </portType>

   <binding name="Hello_Binding" type="tns:Hello_PortType">
      <soap:binding style="rpc"
         transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="sayHello">
         <soap:operation soapAction="sayHello"/>
         <input>
            <soap:body
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               namespace="urn:examples:helloservice"
               use="encoded"/>
         </input>
         <output>
            <soap:body
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               namespace="urn:examples:helloservice"
               use="encoded"/>
         </output>
      </operation>
   </binding>

   <service name="Hello_Service">
      <documentation>WSDL File for HelloService</documentation>
      <port binding="tns:Hello_Binding" name="Hello_Port">
         <soap:address
            location="http://localhost:8080/soap/servlet/rpcrouter"/>
      </port>
   </service>
</definitions>
```

下一节我们将讨论 WSDL 元素。当讨论每个元素时，你可能需要参照图 6-2，它总结了例 6-1 最重要的方面。

definitions

`definitions`元素将这个文档指定为*HelloService*，还指定了这个文档其余部分将用到的大量名称空间。

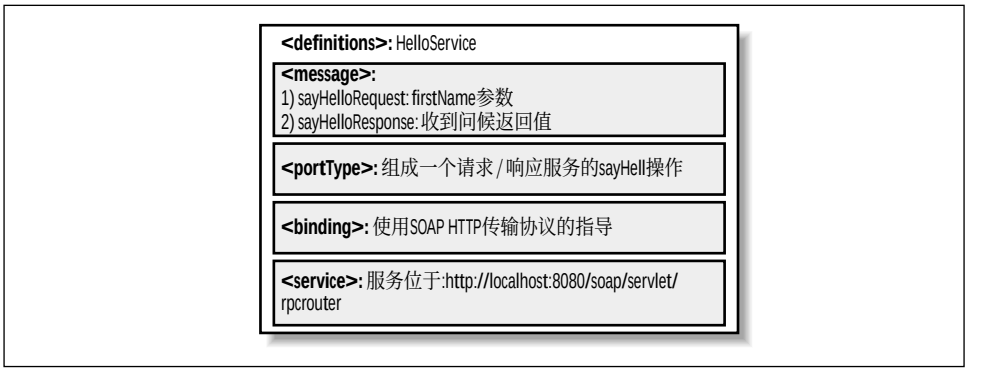

图 6-2：HelloService.wsdl 鸟瞰图

```
<definitions name="HelloService"
   targetNamespace="http://www.ecerami.com/wsdl/HelloService.wsdl"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://www.ecerami.com/wsdl/HelloService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

名称空间对于区分元素非常重要，它使文档可以引用 WSDL 规范、SOAP 规范和 XML Schema 规范等多个外部规范。

`definitions`元素还指定了一个`targetNamespace`属性。`targetNamespace`属性是一个 XML Schema 约定，使得 WSDL 文档可以引用自身。在例 6-1 中，我们指定 `targetNamespace` 为 *http://www.ecerami.com/wsdl/HelloService.wsdl*。注意，名称空间规范并不要求文档就在这个地方，重要的是你必须指定惟一的、不同于其他名称空间的值。

最后，`definitions` 元素指定了一个默认的名称空间： *xmlns=http://schemas.xmlsoap.org/wsdl/*。`message`和`portType`等没有名称空间前缀的所有元素都被认为属于默认的 WSDL 名称空间。

message

本例中定义了两个 `message`元素：第一个表示请求消息 *SayHelloRequest*，第二个表示响应消息 *SayHelloResponse*。

```
<message name="SayHelloRequest">
   <part name="firstName" type="xsd:string"/>
</message>
<message name="SayHelloResponse">
   <part name="greeting" type="xsd:string"/>
</message>
```

每个消息都含有一个part元素。在请求消息中，part指定函数参数，在本例中，我们指定了一个firstName参数；在响应中，part指定函数返回值，在本例中，我们指定了一个greeting返回值。

part元素的type属性指定一个XML Schema数据类型。type属性的值必须被指定为XML Schema QName，这意味着属性的值一定要符合名称空间的要求。譬如，firstName的type属性被设为xsd:string。xsd前缀引用了在前面的definitions元素中定义的XML Schema的名称空间。

如果函数需要多个参数或返回多个值，那么你可以指定多个part元素。

portType

portType元素定义了一个名为*sayHello*的单一操作。这个操作本身由单一的input消息（*SayHelloRequest*）和单一的output消息（*SayHelloResponse*）组成：

```
<portType name="Hello_PortType">
   <operation name="sayHello">
      <input message="tns:SayHelloRequest"/>
      <output message="tns:SayHelloResponse"/>
   </operation>
</portType>
```

message属性与前面定义的type属性非常相似，也要被指定为XML Schema QName，这意味着属性的值一定要符合名称空间的要求。譬如，input元素指定tns:SayHelloRequest的一个message属性，tns前缀引用了前面在definitions元素中定义的targetNamespace。

WSDL支持四种基本操作模式：

单向（one-way）

服务接收一个消息，于是操作有了单一的 `input` 元素。

请求－响应（request-response）

服务接收一个消息，并发出一个响应，于是操作有了一个 `input` 元素，接着又有了一个 `output` 元素（在前面的例 6-1 中已经说明）。为了封装错误，还可以指定一个 `fault` 元素。

征求－响应（solicit-response）

服务发出一个消息，并接收一个响应，于是操作有了一个 `output` 元素，接着又有了一个 `input` 元素。为了封装错误，还可以指定一个 `fault` 元素。

通知（notification）

服务发出一个消息，于是操作有了单一的 `output` 元素。

图 6-3 也展示了这些操作模式。在 SOAP 服务中，请求－响应模式应用最为普遍。

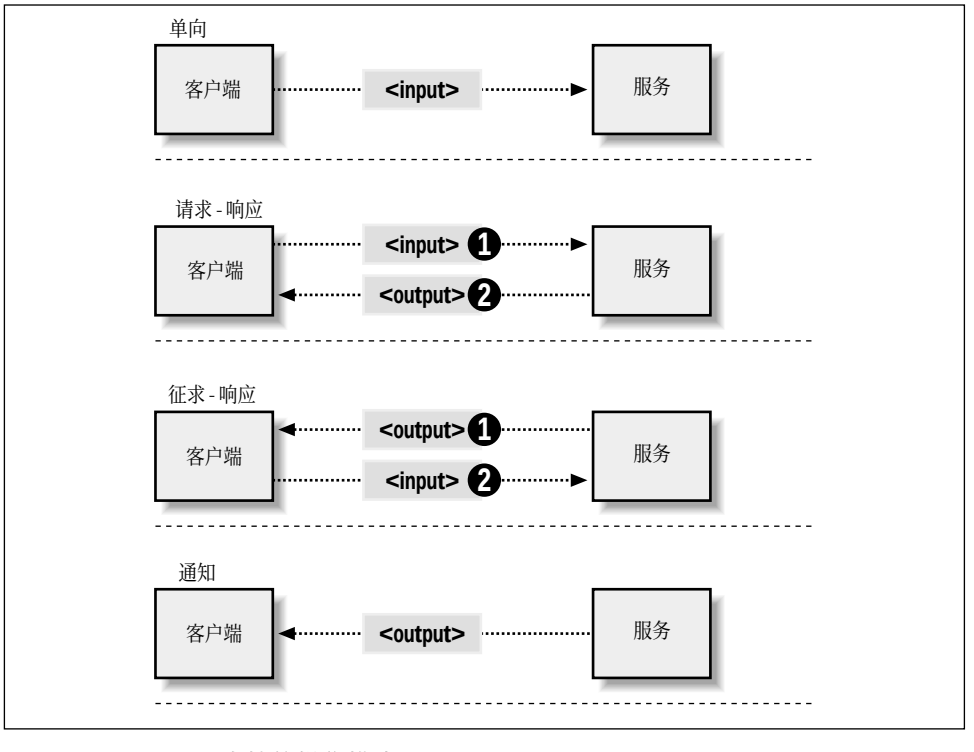

图 6-3：WSDL 1.1 支持的操作模式

binding

binding 元素提供有关 portType 操作如何通过 Internet 实际传递的细节。绑定可以通过 HTTP GET、HTTP POST 或 SOAP 等多个传输协议实现。实际上你可以为单个 portType 指定多个绑定。

binding 元素自身指定 name 和 type 属性：

```
<binding name="Hello_Binding" type="tns:Hello_PortType">
```

type 属性引用在文档前面定义的 portType。在我们的例子中，binding 元素引用在文档前面定义的 tns:Hello_PortType，于是绑定元素发出“ I will provide specific details on how the *sayHello* operation will be transported over the Internet（我将提供如何通过 Internet 传输 *sayHello* 操作的具体细节）”。

SOAP 绑定

WSDL 1.1 包含内置的 SOAP 1.1 扩展，让你可以指定 SOAP 首部、SOAP 编码形式和 SOAPAction HTTP 首部等 SOAP 特有的细节。SOAP 的扩展元素包括：

soap:binding
: 这个元素表明绑定可以通过 SOAP 实现。style 属性表明 SOAP 消息格式的总类型。rpc 的 style 值指定一个 RPC 格式，这意味着 SOAP 请求主体将包含一个标明函数名的 XML 包装元素。函数参数随后被嵌入包装元素。同样，SOAP 响应主体也会包含一个反映函数请求的 XML 包装元素。返回值随后被嵌入响应包装元素中。

  document 的 style 值指定一个 XML 文档调用格式，这意味着请求和响应消息将只由 XML 文档构成。文档格式比 rpc 格式更普通，因而不需要使用包装元素。（要想得到更多细节，请看马上将出现的注释。）

  transport 属性标明 SOAP 消息的传输，http://schemas.xmlsoap.org/soap/http 值表示 SOAP HTTP 传输，而 http://schemas.xmlsoap.org/soap/smtp 值表示 SOAP SMTP 传输。

`soap:operation`

这个元素表示特定操作与特定SOAP实现相互绑定。`soapAction`属性指定用`SOAPAction` HTTP首部标识服务。（有关`SOAPAction`首部的信息，请参阅第三章。）

`soap:body`

这个元素使你可以指定输入、输出消息的细节。在例子HelloWorld中，`body`元素指定SOAP编码形式和与指定服务相关的名称空间URN。

注意：选择`rpc`格式还是`document`格式，意见不一。这个话题在WSDL新闻组（*http://groups.yahoo.com/group/wsdl*）内曾有过激烈的讨论。因为并非所有可识别WSDL的工具都能区分这两种格式，所以这个争论会更激烈。因为`rpc`格式与前面章节中给出的SOAP例子更为一致，所以，在本章中我一直在所有的例子中使用`rpc`格式。请注意，大部分Microsoft .NET WSDL文件使用的却是`document`格式。

---

service

`service`元素指定服务地址。因为本例是一个SOAP服务，所以我们使用`soap:address`元素为Apache SOAP `rpcrouter` servlet指定本地主机地址：`http://localhost:8080/soap/servlet/rpcrouter`。

注意，服务元素包含一个`documentation`元素以提供可阅读的文档。

WSDL调用工具之一

有了如图6-1所示的WSDL文件，你就可以手动创建一个SOAP客户端来调用服务了。更好的选择则是通过WSDL调用工具自动调用服务（见图6-4）。

目前已有许多WSDL调用工具。本节只对三个调用工具做一个简短介绍。

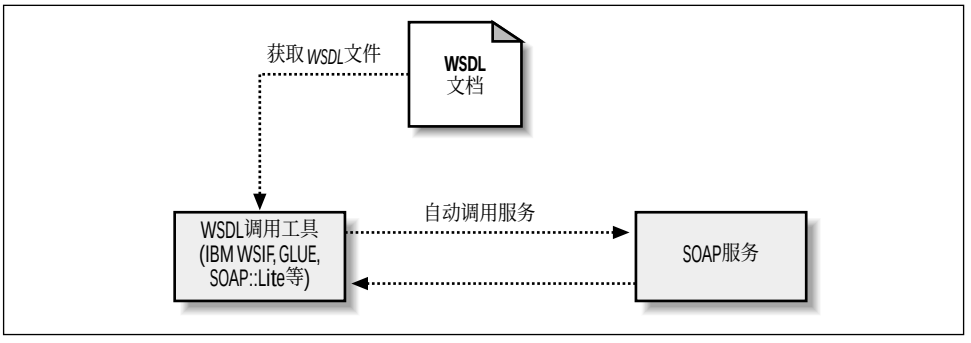

图 6-4：WSDL 调用工具

GLUE

The Mind Electric 公司提供了一个名为 GLUE 的完整 Web 服务平台（可以从 *http://www.themindelectric.com/* 获得）。这个平台本身广泛支持 SOAP、WSDL 和 UDDI。本章后面还将讨论其支持复杂数据类型等高级功能。

现在，你可以试一下 GLUE 的 `invoke` 命令行工具。下面是命令行用法:

```
usage: invoke URL method arg1 arg2 arg3...
```

譬如，要调用 HelloService，就要先确保你的 Apache Tomcat 服务器正在运行，并将 *HelloService.wsdl* 文件放在公用目录里。接着，发出如下命令：

```
invoke http://localhost:8080/wsdl/HelloService.wsdl sayHello World
```

一经调用, GLUE将立即下载指定的WSDL文件, 调用*sayHello* 方法, 并将`World` 作为参数传递。GLUE 将自动显示服务器响应：

```
Output:  result = Hello, World!
```

有关 GLUE 所有的内容就是这些!

GLUE 还支持优秀的记录功能，这样，你就可以方便地浏览所有的 SOAP 消息。

要激活记录功能，应设置`electric.logging`系统属性。最简单的方法是修改*invoke.bat*文件。原文件是这样的：

```
call java electric.glue.tools.Invoke %1 %2 %3 %4 %5 %6 %7 %8 %9
```

修改文件中传递到Java解释器的`-D`选项，使文件包含记录属性：

```
call java -Delectric.logging="SOAP" electric.glue.tools.Invoke %1 %2 %3 %4
    %5 %6 %7 %8 %9
```

现在调用HelloService时，GLUE会产生下面的输出：

```
LOG.SOAP: request to http://207.237.201.187:8080/soap/servlet/rpcrouter
<?xml version='1.0' encoding='UTF-8'?>
<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:sayHello xmlns:n='urn:examples:helloservice'>
         <firstName xsi:type='xsd:string'>World</firstName>
      </n:sayHello>
   </soap:Body>
</soap:Envelope>

LOG.SOAP: response from http://207.237.201.187:8080/soap/servlet/rpcrouter
<?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/1999/XMLSchema-instance'
   xmlns:xsd='http://www.w3.org/1999/XMLSchema'>
   <SOAP-ENV:Body>
      <ns1:sayHelloResponse
         xmlns:ns1='urn:examples:helloservice'
         SOAP-ENV:encodingStyle=
            'http://schemas.xmlsoap.org/soap/encoding/'>
         <return xsi:type='xsd:string'>Hello, World!</return>
      </ns1:sayHelloResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

result = Hello, World!
```

要看到更多的HTTP信息，只需将`electric.logging`设为`SOAP,HTTP`。

面向 Perl 的 SOAP::Lite

由Paul Kulchenko编写的面向Perl的SOAP::Lite也在一定程度上支持WSDL。你可以从 *http://www.soaplite.com* 获得这个程序包。

例 6-2 提供了一个调用 HelloService 的完整 Perl 程序。

例 6-2：Hello_Service.pl

```
use SOAP::Lite;

print "Connecting to Hello Service...\n";
print SOAP::Lite
   -> service('http://localhost:8080/wsdl/HelloService.wsdl')
   -> sayHello ('World');
```

此程序产生如下输出：

```
Connecting to Hello Service...
Hello, World!
```

WSIF

IBM 最近发布了 WSIF（Web 服务调用框架），你可以从 *http://www.alphaworks.ibm.com/tech/wsif* 获得这个包。

与 GLUE 非常相似，WSIF 也为自动调用 WSDL 服务提供了一个简单的命令行方法，如下面的命令:

```
java clients.DynamicInvoker http://localhost:8080/wsdl/HelloService.wsdl
    sayHello World
```

将产生如下输出:

```
Reading WSDL document from 'http://localhost:8080/wsdl/HelloService.wsdl'
Preparing WSIF dynamic invocation
Executing operation sayHello
Result:
greeting=Hello, World!

Done!
```

基本的 WSDL 例子: XMethods eBay Price Watcher Service

在理解复杂的 WSDL 例子前，让我们来看一个相对简单的例子。例 6-3 提供了 XMethods eBay Price Watcher Service 的一个 WSDL 文件。这个服务获得一个已有的 eBay 拍卖 ID，并返回当前报价。

例 6-3：eBayWatcherService.wsdl（引用此文件得到了 XMethods 公司的许可）

```
<?xml version="1.0"?>
<definitions name="eBayWatcherService"
   targetNamespace=
      "http://www.xmethods.net/sd/eBayWatcherService.wsdl"
   xmlns:tns="http://www.xmethods.net/sd/eBayWatcherService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns="http://schemas.xmlsoap.org/wsdl/">

   <message name="getCurrentPriceRequest">
      <part name="auction_id" type = "xsd:string"/>
   </message>
   <message name="getCurrentPriceResponse">
      <part name="return" type = "xsd:float"/>
   </message>

   <portType name="eBayWatcherPortType">
      <operation name="getCurrentPrice">
         <input
            message="tns:getCurrentPriceRequest"
            name="getCurrentPrice"/>
         <output
            message="tns:getCurrentPriceResponse"
            name="getCurrentPriceResponse"/>
      </operation>
   </portType>

   <binding name="eBayWatcherBinding" type="tns:eBayWatcherPortType">
      <soap:binding
         style="rpc"
         transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="getCurrentPrice">
         <soap:operation soapAction=""/>
         <input name="getCurrentPrice">
            <soap:body
```

```
                use="encoded"
                namespace="urn:xmethods-EbayWatcher"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
          </input>
          <output name="getCurrentPriceResponse">
             <soap:body
                use="encoded"
                namespace="urn:xmethods-EbayWatcher"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
          </output>
       </operation>
    </binding>

    <service name="eBayWatcherService">
       <documentation>
          Checks current high bid for an eBay auction
       </documentation>
       <port name="eBayWatcherPort" binding="tns:eBayWatcherBinding">
          <soap:address
           location="http://services.xmethods.net:80/soap/servlet/rpcrouter"/>
       </port>
    </service>
</definitions>
```

下面概述了主要的 WSDL 元素：

`messages`

本例中定义了两个消息：`getCurrentPriceRequest`和`getCurrentPriceResponse`。请求消息含有单一的字符串型参数，响应消息则含有单一的浮点型参数。

`portType`

`portType`定义了单个操作`getCurrentPrice`。我们又一次看到了请求/响应操作模式。

`binding`

`binding`元素指定传输格式为 HTTP SOAP。`soapAction`属性为一个空字符串（`""`）。

`service`

这个元素指定服务可从*http://services.xmethods.net:80/soap/servlet/rpcrouter*获得。

要想获得 eBay 观察家服务，你可以使用前面定义的任何 WSDL 调用工具。例如向 GLUE 发出如下调用：

```
invoke http://www.xmethods.net/sd/2001/EBayWatcherService.wsdl
    getCurrentPrice 1271062297
```

将检索到 Handspring Visor Deluxe 的当前报价：

```
result = 103.5
```

注意： XMethods 网站（*http://www.xmethods.net/*）提供了大量的 SOAP 和 .NET 服务样例。这些服务几乎都包含 WSDL 文件，因此提供了一个进一步学习 WSDL 的绝好机会。当你浏览 XMethods 目录时，可尝试用刚才描述的任何 WSDL 调用工具连接某个服务，你将会吃惊地发现，连接和调用新的服务竟然如此简单。

WSDL 调用工具之二

在前面讨论的WSDL调用工具主要是编程和命令行调用工具，我们现在讨论完全由基于 Web 的接口驱动的更简单的工具。

GLUE 控制台

GLUE 平台除了支持一些命令行工具之外，还支持非常直接的、用于部署新服务和连接已有服务的 Web 接口。

要启动一个 GLUE 控制台，只需输入：

```
console
```

这个命令会自动从默认端口 8100 启动 GLUE 控制台。打开 Web 浏览器，你将会看到 GLUE 控制台主页（见图 6-5）。

在题为 WSDL 的文本框中，你可以输入任何 WSDL 文件的 URL。如 eBay Price Watcher Service的URL：*http://www.xmethods.net/sd/2001/EBayWatcherService.wsdl*。

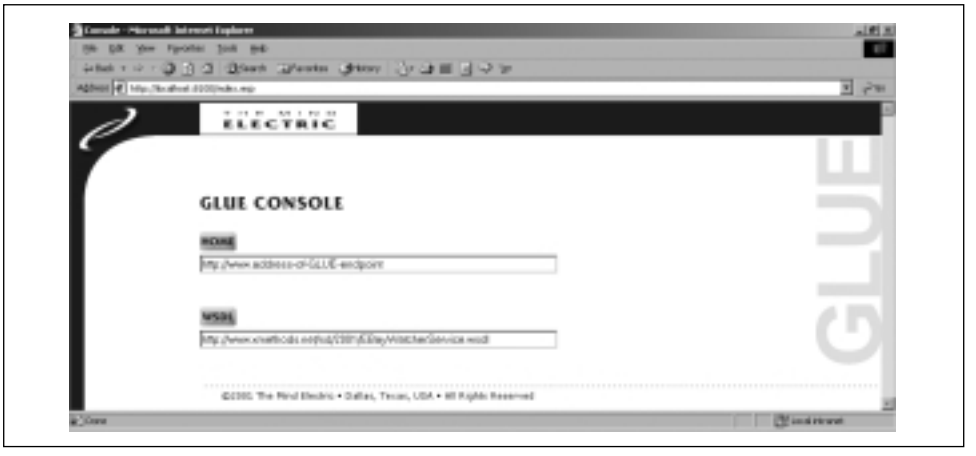

图 6-5：GLUE 控制台：主页

单击 WSDL 按钮，你将会看到 Web Service 总览页（见图 6-6）。这个网页有对指定服务（提取自 WSDL `document` 元素）和一组公共操作的描述。在 eBay 服务这个例子中，你可以看到一个 `getCurrentPrice` 方法。

图 6-6：GLUE 控制台：eBay Price Watcher Service 的 Web Service 总览页

单击 `getCurrentPrice` 方法，你将会看到 Web Method 总览页（见图 6-7）。这个网页中有一个文本框，在这个框里你可以指定拍卖 ID。

图 6-7：GLUE 控制台：getCurrentPrice 的 Web Method 总览页

输入一个拍卖 ID，然后按 SEND 按钮，GLUE 将自动调用远程方法，并在页面底部显示结果。图 6-8 显示了 Handspring Visor Deluxe 的当前标价。注意，自从用 GLUE 命令行工具调用服务以来，标价已上涨了 10 美元。

图 6-8：GLUE 控制台：调用 getCurrentPrice 方法（调用的结果显示在窗口底部）

SOAPClient.com

如果你想试试与 GLUE 相似的、基于 Web 的接口，而又不想费力下载 GLUE 程序包，则可以考虑能从 SOAPClient.com 获得的 Generic SOAP Client。

图 6-9 所示为 Generic SOAP Client 的窗口。可以采用与 GLUE 控制台相似的方法，为窗口中的一个 WSDL 文件指定地址。

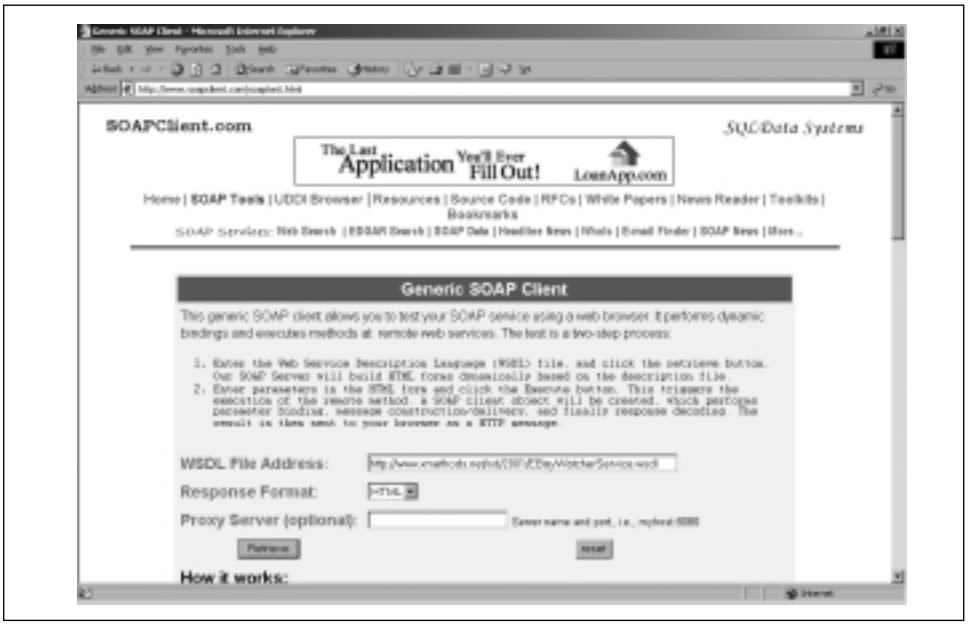

图 6-9：Generic SOAP Client，可以从 SOAPClient.com 获得

指定相同的 eBay Price Watcher Service WSDL 文件，SOAP Client 就会显示一个用于输入拍卖 ID 的文本框（见图 6-10）。

图 6-11 显示了 eBay 服务调用的结果，Handspring Visor 的标价又上涨了 4 美元。

图 6-10：Generic SOAP Client：显示有关 XMethods eBay Price Watcher Service 的信息

图 6-11：Generic SOAP Client：从 XMethods eBay Price Watcher Service 发出的响应

自动生成 WSDL 文件

WSDL 最为优秀的一面是，你几乎不用从零开始创建 WSDL 文件。现在有许多将已有服务转变成 WSDL 描述的工具，你可以选择原样使用那些 WSDL 文件或用

你喜欢的文本编辑器手动调整后再使用那些WSDL文件。在接下来的讨论中，我们将探索GLUE提供的WSDL生成工具。

注意：如果你从零开始创建WSDL文件，或者修改通过某工具生成的WSDL文件，那么验证一下你的最终WSDL文档是一个好主意。你可以从*http://pocketsoap.com/wsdl/*下载一个WSDL验证器。这个程序包要求你拥有XSLT引擎和zvonSchematron（*http://www.zvon.org/*），但安装只需几分钟。一经安装，验证器就不会让你失望，它能创建出格式非常漂亮的HTML报告，其中标明了WSDL错误和警告。

GLUE java2wsdl工具

GLUE平台含有一个将Java服务转变成WSDL描述的`java2wsdl`命令行工具。命令行的用法如下：

```
usage: java2wsdl <arguments>

其中有效参数为:
  classname                Java类名
  -d directory             输出目录
  -e url                   服务终止点
  -g                       包括GET/POST绑定
  -m map-file              读映射指令
  -n namespace             服务的名称空间
  -r description           服务描述
  -s                       包括SOAP绑定
  -x command-file          要执行的命令文件
```

可以从*http://www.themindelectric.com/products/glue/releases/GLUE-1.1/docs/guide/index.html*在线获得的GLUE用户指南中提供了有关每个参数的完整信息。现在，我们主要讨论最基本的参数。

我们来看例6-4中的`PriceService`类，这个服务提供一个`getPrice()`方法。

例6-4：PriceService.java

```
package com.ecerami.soap.examples;

import java.util.Hashtable;
/**
```

```
 * SOAP 服务样例
 * 为所请求的库存保管单元（SKU）提供当前报价
 */
public class PriceService {
  protected Hashtable products;

  /**
   * 零参数构造函数
   * 把两个样例产品载入产品数据库
   */
  public PriceService () {
    products = new Hashtable();
    //  Red Hat Linux
    products.put("A358185", new Double (54.99));
    //  McAfee PGP Personal Privacy
    products.put("A358565", new Double (19.99));
  }

  /**
  *  为所请求的 SKU 提供当前报价
  *  在实际安装时，该方法将连接到
  *  一个报价数据库。如果未找到 SKU，
  *  则方法抛出一个 PriceException 异常。
  */
  public double getPrice (String sku)
    throws ProductNotFoundException {
    Double price = (Double) products.get(sku);
    if (price == null) {
      throw new ProductNotFoundException ("SKU: "+sku+" not found");
    }
    return price.doubleValue();
  }
}
```

要为这个类生成一个 WSDL 文件，请运行如下命令：

```
java2wsdl com.ecerami.soap.examples.PriceService -s -e http://localhost:
 8080/soap/servlet/rpcrouter -n urn:examples:priceservice
```

-s 选项指导 GLUE 创建一个 SOAP 绑定，-e 选项指定我们服务的地址，-n 选项指定服务的名称空间 URN。GLUE 将生成一个 *PriceService.wsdl* 文件（见例 6-5）。

注意：如果你通过 Java 接口定义服务，并将源文件放在 CLASSPATH 中，GLUE 就会提取你的 Javadoc 注解，并将其转换成 WSDL `documentation` 元素。

例 6-5：PriceService.wsdl（由 GLUE 自动生成）

```
<?xml version='1.0' encoding='UTF-8'?>
<!-- 由 GLUE 生成 -->
<definitions name='com.ecerami.soap.examples.PriceService'
   targetNamespace='http://www.themindelectric.com/wsdl/com.ecerami.soap.
       examples.PriceService/'

   xmlns:tns='http://www.themindelectric.com/wsdl/com.ecerami.soap.
       examples.PriceService/'
   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/'>
  <message name='getPrice0SoapIn'>
    <part name='sku' type='xsd:string'/>
  </message>
  <message name='getPrice0SoapOut'>
    <part name='Result' type='xsd:double'/>
  </message>
  <portType name='com.ecerami.soap.examples.PriceServiceSoap'>
    <operation name='getPrice' parameterOrder='sku'>
      <input name='getPrice0SoapIn' message='tns:getPrice0SoapIn'/>
      <output name='getPrice0SoapOut' message='tns:getPrice0SoapOut'/>
    </operation>
  </portType>
  <binding name='com.ecerami.soap.examples.PriceServiceSoap'
      type='tns:com.ecerami.soap.examples.PriceServiceSoap'>
    <soap:binding style='rpc'
      transport='http://schemas.xmlsoap.org/soap/http'/>
    <operation name='getPrice'>
      <soap:operation soapAction='getPrice' style='rpc'/>
      <input name='getPrice0SoapIn'>
        <soap:body use='encoded'
         namespace='urn:examples:priceservice'
         encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'/>
      </input>
      <output name='getPrice0SoapOut'>
        <soap:body use='encoded'
        namespace='urn:examples:priceservice'
        encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'/>
      </output>
    </operation>
  </binding>
  <service name='com.ecerami.soap.examples.PriceService'>
```

```
    <port name='com.ecerami.soap.examples.PriceServiceSoap'
      binding='tns:com.ecerami.soap.examples.PriceServiceSoap'>
      <soap:address location='http://207.237.201.187:8080
          /soap/servlet/ rpcrouter'/>
    </port>
  </service>
</definitions>
```

然后你可以用 SOAP::Lite 调用这个服务：

```
use SOAP::Lite;

print "Connecting to Price Service...\n";
print SOAP::Lite
   -> service('http://localhost:8080/wsdl/PriceService.wsdl')
   -> getPrice ('A358185');
```

这个例子向我们展现了Web服务互操性方面的美好前景。我们有一个由GLUE生成的 WSDL 文件、一个运行 Java 的服务器和一个运行 Perl 的客户端，它们彼此紧密配合。

```
Connecting to Price Service...
54.99
```

警告：IBM Web 服务工具包（IBM Web Services Toolkit）（可以从 *http://www.alphaworks.ibm.com/tech/webservicestoolkit* 获得）提供了一个名为 `wsdlgen` 的 WSDL 生成工具。这个工具可以利用已有的 Java 类、Enterprise JavaBean 和微软的 COM 对象，自动生成相应的 WSDL 文件。但是，当本书付印时，`wsdlgen` 工具创建的还是以 W3C XML Schema 1999 版为基础的文件，因此，生成的 WSDL 文件与 SOAP::Lite 和 GLUE 等其他 WSDL 调用工具不兼容。如果你选择使用 IBM 工具，那么一定要手动更新你的 WSDL 文件，使其反映最新版的 XML Schema（*http://www.w3.org/2001/XMLSchema*）。

XML Schema 的数据类型定义

为了使 SOAP 客户端与 SOAP 服务器有效地通信，客户端和服务器在数据类型系统上应达成一致。XML 1.0 默认状态下不提供数据类型系统，而各种编程语言都提供某种基本的工具来定义整型、浮点型、双精度型和字符串型等数据类型。因

此，建立Web服务最大的挑战是建立一个能在各种操作系统上被各种编程语言使用的共同的数据类型系统。

WSDL 并不打算为 XML 数据类型定义创建一个标准。事实上，WSDL 的设计意图是要实现最大的灵活性，因此不受任何一种数据类型系统的约束。但是，默认状态下，WSDL 遵守 W3C XML Schema 规范，而 XML Schema 规范是目前使用最广泛的数据定义规范。

对 XML Schema 懂得越多，对 WSDL 文件的理解也就越透彻。深入讨论 XML Schema 已超出本章的范围，但是两个关键的因素需要在这里说明。

第一，XML Schema 规范包含编码大多数数据类型的基本数据类型系统，这个类型系统包括一长串内置的简单类型，如字符串、浮点型、双精度型、整型、时间和日期。表 6-1 摘自 XML Schema Part 0: Primer（*http://www.w3org/TR/2000/WD=xmlschema=0=20000407/*）。如果你的应用程序只使用这些简单类型，就没必要包含WSDL `types`元素，生成的WSDL文件也非常简单。如前面两个WSDL文件只使用了字符串和浮点型数据。

表 6-1：XML Schema 主要内置简单类型列表

简单类型	例子
string	Web Services
Boolean	true, false, 1, 0
float	-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN
double	-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN
decimal	-1.23, 0, 123.4, 1000.00
binary	100010
integer	-126789, -1, 0, 1, 126789
nonPositiveInteger	-126789, -1, 0
negativeInteger	-126789, -1
long	-1, 12678967543233
int	-1, 126789675

表6-1：XML Schema主要内置简单类型列表（续）

简单类型	例子
short	-1, 12678
byte	-1, 126
nonNegativeInteger	0, 1, 126789
unsignedLong	0, 12678967543233
unsignedInt	0, 1267896754
unsignedShort	0, 12678
unsignedByte	0, 126
positiveInteger	1, 126789
date	1999-05-31
time	13:20:00.000, 13:20:00.000-05:00

第二，XML Schema规范提供了创建新数据类型的工具。如果你要创建一个Schema没有定义的数据类型，那么这个工具就非常重要了。例如，一个服务可能会返回一个浮点型数组或更复杂的、包含某股票价格的高低和交易量的股票报价对象。当你的服务超出简单XML Schema数据类型时，你必须在WSDL的`types`元素中声明新的数据类型。

在本章接下来的两节中，我们将给出两个具体使用MXL Schema创建新数据类型的例子。第一个例子关注数组，而第二个例子以更为复杂的、封装产品信息的数据类型为重点。

数组

本节后面的例6-6就是一个说明如何使用数组的WSDL文件样例，这是我们在第五章中创建的Price List Service。这个服务有一个名为`getPriceList`的公共方法，它需要一个字符串型SKU值的数组，并返回一个双精度型价格值的数组。

现在，WSDL文件含有一个`types`元素，在这个元素里，我们定义了两个新的复合类型。XML Schema定义了简单类型和复合类型，简单类型没有子元素和属性，

而复合类型有子元素和属性。我们已经在WSDL文件中声明了复合类型，因为一个数组可能会有多个元素，因此每个元素对应数组中的一个值。

XML Schema要求所创建的新类型必须以已有的数据类型为基础。已有数据类型通过base属性指定。修改基本类型的方法有两个：extension和restriction，你可以选择其中的一个来修改基本数据类型。扩展只是指新数据类型将包括基本类型的所有属性以及另外的一些功能。限制是指新数据类型有基本数据类型的所有属性，但对数据可能有更多的限制。

在例6-6中，我们将通过限制创建两个复合类型，如：

```
<complexType name="ArrayOfString">
   <complexContent>
      <restriction base="soapenc:Array">
         <attribute ref="soapenc:arrayType"
            wsdl:arrayType="string[]"/>
      </restriction>
   </complexContent>
</complexType>
```

例6-6：PriceListService.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="PriceListService"
   targetNamespace="http://www.ecerami.com/wsdl/PriceListService.wsdl"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://www.ecerami.com/wsdl/PriceListService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsd1="http://www.ecerami.com/schema">

    <types>
      <schema xmlns="http://www.w3.org/2001/XMLSchema"
          targetNamespace="http://www.ecerami.com/schema"
           xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
           xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">

        <complexType name="ArrayOfString">
           <complexContent>
              <restriction base="soapenc:Array">
                 <attribute ref="soapenc:arrayType"
                 wsdl:arrayType="string[]"/>
              </restriction>
```

```
            </complexContent>
      </complexType>
      <complexType name="ArrayOfDouble">
         <complexContent>
            <restriction base="soapenc:Array">
               <attribute ref="soapenc:arrayType"
               wsdl:arrayType="double[]"/>
            </restriction>
         </complexContent>
      </complexType>
   </schema>
</types>

<message name="PriceListRequest">
   <part name="sku_list" type="xsd1:ArrayOfString"/>
</message>

<message name="PriceListResponse">
   <part name="price_list" type="xsd1:ArrayOfDouble"/>
</message>

<portType name="PriceList_PortType">
   <operation name="getPriceList">
      <input message="tns:PriceListRequest"/>
      <output message="tns:PriceListResponse"/>
   </operation>
</portType>

<binding name="PriceList_Binding" type="tns:PriceList_PortType">
<soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
   <operation name="getPriceList">
   <soap:operation soapAction="urn:examples:pricelistservice"/>
      <input>
         <soap:body
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="urn:examples:pricelistservice"
            use="encoded"/>
      </input>
      <output>
         <soap:body
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="urn:examples:pricelistservice" use="encoded"/>
      </output>
   </operation>
</binding>

<service name="PriceList_Service">
   <port name="PriceList_Port" binding="tns:PriceList_Binding">
```

```
        <soap:address location="http://localhost:8080/soap/servlet/rpcrouter"/>
      </port>
   </service>
</definitions>
```

WSDL 规范要求数组以 SOAP 1.1 编码模式为基础，还要求数组使用名称 `ArrayOfXXX`，这里 `XXX` 是数组项的类型。因此，上面的例子创建了一个名为 `ArrayOfString`的新类型，这个新类型以SOAP数组数据类型为基础，但它只许含有字符串值。同样，`ArrayOfDouble`数据类型创建了一个只含双精度值的新数组类型。

使用 WSDL `types` 元素时，你需要非常了解 XML 名称空间问题。首先，注意根 `schema` 元素一定要包含 SOAP 编码规范（*http://schemas.xmlsoap.org/soap/encoding/*）的名称空间声明。这是因为我们的新数据类型扩展了由 SOAP 指定的数组定义。

其次，根 `schema` 元素一定要指定一个 `targetNamespace` 属性。任何新定义的元素，例如我们的新数组数据类型，将属于指定的 `targetNamespace`。为了在文档后面引用数据类型，你必须回过头引用相同的 `targetNamespace`。因此，我们的 `definitions` 元素包含一个新的名称空间声明：

```
xmlns:xsd1="http://www.ecerami.com/schema">
```

`xsd1` 与 `targetNamespace` 相匹配，因此我们可以在文档后面引用新数据类型。例如 `message` 元素引用 `xsd1:ArrayOfString` 数据类型：

```
<message name="PriceListRequest">
       <part name="sku_list" type="xsd1:ArrayOfString"/>
</message>
```

注意：要想得到精彩、简洁的 W3C Schema 复合类型及通过扩展和限制而得到派生物的综述，请参阅 Donald Smith 撰写的关于“理解 W3C Schema 复合类型”的文章。这篇文章可从 *http://www.xml.com/pub/a/2001/08/22/easyschema.html* 在线获得。

自动调用数组服务

一旦超出基本数据类型的范围，本章前面描述的简单WSDL调用方法用起来就不那么容易了。例如，你不能仅仅打开GLUE控制台，传递一个字符串数组，就指望会收到一个双精度数组。还需要额外的工作，手动编写一些代码是不可避免的。但是，额外工作只是少量的，接下来的讨论将集中在GLUE平台上，这是因为它是处理复合数据类型最好的平台。其他像IBM的Web服务工具包等工具也提供类似的功能。

在开始之前，你应该熟悉GLUE `wsdl2java`命令行工具。为了自动连接指定的服务，这个工具接受一个WSDL文件，并生成一组Java类文件。你可以自己编写调用指定服务的Java类。但你的代码最好尽可能简单，所有SOAP特定的细节应完全隐藏起来（见图6-12）。

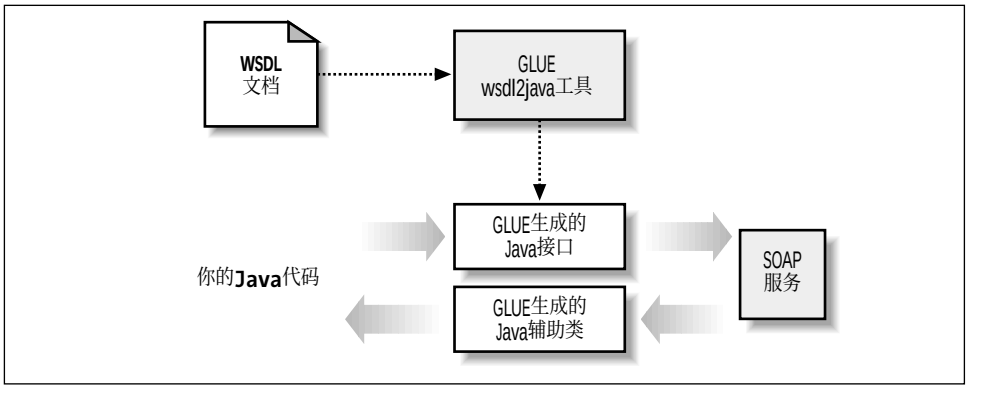

图6-12：GLUE wsdl2java工具和GLUE体系结构

下面是`wsdl2java`命令行的用法：

```
usage: wsdl2java <arguments>

其中有效参数为:
  http://host:port/filename      WSDL的URL
  -c                             检测到的异常
  -d directory                   文件的输出目录
  -l user password realm         登录凭证
  -m map-file                    读映射指令
  -p package                     设置默认的包
```

```
-v                              详细信息
-x command-file                 要执行的命令文件
```

你可以从*http://www.themindelectric.com/products/glue/releases/GLUE-1.1/docs/guide/index. html*在线获得GLUE用户指南, 里面有每个参数的完整信息。现在, 我们重点介绍最基本的参数。譬如, 要为*PriceListService.wsdl*文件生成一个Java类文件, 首先应确保这个WSDL文件可以从网站公共获得, 或通过Tomcat等Web服务器从本地获得。然后发出如下命令：

```
wsdl2java.bat http://localhost:8080/wsdl/PriceListService.wsdl -p com.
   ecerami.wsdl.glue
```

第一个参数指定WSDL文件的位置；第二个参数指定生成的文件应存放在`com.ecerami.wsdl.glue`包中。

GLUE将自动下载指定的WSDL文件，并生成两个Java类文件：

```
write file IPriceList_Service.java
write file PriceList_ServiceHelper.java
```

例6-7为第一个文件*IPriceList_Service.java*。这个文件提供了一个接口，该接口反映了通过WSDL文件暴露的公共方法。特别是这个接口显示了一个`getPriceList()`方法，这个方法接收一个字符串型数组，并返回一个双精度型数组。

例6-7：IPriceList_Service.java

```
// 由GLUE生成

package com.ecerami.wsdl.glue;

public interface IPriceList_Service
  {
  double[] getPriceList( String[] sku_list );
  }
```

例6-8为第二个文件*PriceList_ServiceHelper.java*。这是一个GLUE辅助文件, 能动态地绑定到由WSDL文件指定的服务。要获得这个服务, 只需调用静态`bind()`方法。

例 6-8：PriceList_ServiceHelper.java

```
// 由GLUE生成

package com.ecerami.wsdl.glue;

import electric.registry.Registry;
import electric.registry.RegistryException;

public class PriceList_ServiceHelper
  {
  public static IPriceList_Service bind() throws RegistryException
    {
    return bind( "http://localhost:8080/wsdl/PriceListService.wsdl" );
    }

  public static IPriceList_Service bind( String url )
    throws RegistryException
    {
    return (IPriceList_Service)
       Registry.bind( url, IPriceList_Service.class );
    }
  }
```

在 GLUE 生成了接口和辅助文件后，你就需要编写自己的类来实际调用这个服务。例 6-9 为调用 Price List Service 的应用程序样例。代码先调用 `PriceList_ServiceHelper.bind()`，这个方法返回一个 `IPriceList_Service` 对象。接下来的所有代码都将 Price List Service 当成本地对象，对开发者来说所有 SOAP 特有的细节都被隐藏了。

下面是 `Invoke_PriceList` 应用程序的输出样例：

```
    Product Catalog
    SKU:  A358185 --> Price:  54.99
    SKU:  A358565 --> Price:  19.99
```

例 6-9：Invoke_PriceList.java

```
package com.ecerami.wsdl;

import com.ecerami.wsdl.glue.*;

/**
 * SOAP调用程序。 使用PriceListServiceHelper来调用
 * SOAP服务。 PriceListServiceHelper和IPriceListService
```

```
 * 由 GLUE 自动生成
*/
public class Invoke_PriceList {

  /**
   *  通过 SOAP 获得产品列表
   */
  public double[] getPrices (String skus[]) throws Exception {
          IPriceList_Service priceListService = PriceList_ServiceHelper.bind();
     double[] prices = priceListService.getPriceList(skus);
    return prices;
  }

  /**
   *  main 方法
   */
  public static void main (String[] args) throws Exception {
    Invoke_PriceList invoker = new Invoke_PriceList();
    System.out.println ("Product Catalog");
    String skus[] = {"A358185", "A358565" };
    double[] prices = invoker.getPrices (skus);
    for (int i=0; i<prices.length; i++) {
      System.out.print ("SKU:  "+skus[i]);
      System.out.println (" --> Price:  "+prices[i]);
    }
  }
}
```

复合类型

我们的最后一个话题是复合数据类型的使用。譬如，一个家庭监视服务提供了有关你的家庭的简洁数据更新，返回的数据可以包括多个数据元素，如当前温度、安全状态、车库门关没关等。把这些数据编码成WSDL需要更多的XML Schema知识，我们也会从中更一步地体会到，对 XML Schema 知道得越多，对复杂的 WSDL 文件理解得就越透彻。

要了解复合类型，请看例 6-10 的 WSDL 文件。这个文件描述了我们在第五章创建的产品服务。

例 6-10：ProductService.wsdl

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

```
targetNamespace="http://www.ecerami.com/wsdl/ProductService.wsdl"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://www.ecerami.com/wsdl/ProductService.wsdl"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsd1="http://www.ecerami.com/schema">

 <types>
   <xsd:schema
       targetNamespace="http://www.ecerami.com/schema"
       xmlns="http://www.w3.org/2001/XMLSchema">
       <xsd:complexType name="product">
          <xsd:sequence>
             <xsd:element name="name" type="xsd:string"/>
             <xsd:element name="description" type="xsd:string"/>
             <xsd:element name="price" type="xsd:double"/>
             <xsd:element name="SKU" type="xsd:string"/>
          </xsd:sequence>
       </xsd:complexType>
    </xsd:schema>
 </types>

 <message name="getProductRequest">
   <part name="sku" type="xsd:string"/>
 </message>

 <message name="getProductResponse">
   <part name="product" type="xsd1:product"/>
 </message>

 <portType name="Product_PortType">
   <operation name="getProduct">
     <input message="tns:getProductRequest"/>
     <output message="tns:getProductResponse"/>
   </operation>
 </portType>

 <binding name="Product_Binding" type="tns:Product_PortType">
   <soap:binding style="rpc"
       transport="http://schemas.xmlsoap.org/soap/http"/>
   <operation name="getProduct">
     <soap:operation soapAction="urn:examples:productservice"/>
     <input>
       <soap:body
           encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
           namespace="urn:examples:productservice"
           use="encoded"/>
```

```
      </input>
      <output>
        <soap:body
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="urn:examples:productservice" use="encoded"/>
      </output>
    </operation>
  </binding>

  <service name="Product_Service">
    <port name="Product_Port" binding="tns:Product_Binding">
      <soap:address location="http://localhost:8080/soap/servlet/rpcrouter"/>
    </port>
  </service>
</definitions>
```

例6-10中的服务描述了一个`getProduct`操作，这个操作返回一个用于封装产品名称、描述、价格和SKU数等产品信息的复合类型*product*。

新的*product*类型的定义方式与前一个例子中定义数组时所用的方式非常相似。主要的区别是我们现在用的是`sequence`元素。`sequence`元素指定一组子元素，并要求这些子元素以指定的顺序出现。XML Schema还使你可以通过`minOccurs`和`maxOccurs`属性指定集的容量。如果这些属性没有设定（如本例），则默认值为1，意思是每个子元素刚好出现一次。

每一个子元素还可以有各自的数据类型，你将会看到，在我们的这个例子中，字符串数据类型和双精度数据类型混合匹配使用。

自动调用复合数据类型服务

要自动调用Product Service，我们再用一次GLUE `wsdl2java`工具。这一次，GLUE将生成一个Java接口类和一个Java辅助类，以及两个处理新的复合类型的文件。

例如，以下命令：

```
wsdl2java.bat http://localhost:8080/wsdl/ProductService.wsdl -p com.ecerami.
    wsdl.glue
```

将产生如下输出：

```
    write file IProduct_Service.java
    write file Product_ServiceHelper.java
    write file product.java
    write file Product_Service.map
```

我们对输出的前两个文件已经比较熟悉了，第一个文件是反映服务的Java接口，第二个文件是动态绑定到指定服务的辅助文件（见例6-11和例6-12）。

例6-11：IProduct_Service.java

```
// 由GLUE生成

package com.ecerami.wsdl.glue;

public interface IProduct_Service
  {
  product getProduct( String sku );
  }
```

例6-12：Product_ServiceHelper.java

```
// 由GLUE生成

package com.ecerami.wsdl.glue;

import electric.registry.Registry;
import electric.registry.RegistryException;

public class Product_ServiceHelper
  {
  public static IProduct_Service bind() throws RegistryException
    {
    return bind( "http://localhost:8080/wsdl/ProductService.wsdl" );
    }

  public static IProduct_Service bind( String url )
   throws RegistryException
    {
    return (IProduct_Service)
      Registry.bind( url, IProduct_Service.class );
    }
  }
```

输出的第三个文件是*product.java*，表示一个用来封装产品数据的简单容器类（见例 6-13）。GLUE 自身就带有 WSDL 文件中定义的所有复合类型，并为每个子元素创建了一个容器类，随后每个子元素被转换成容易访问的公共变量。如 product类有四个与我们的新复合数据类型相对应的公共变量：name、description、price和SKU。还要注意到公共变量与 WSDL 文件中指定的 XML Schema 类型相匹配，如 name 被声明为字符串型，而 price 被声明为双精度型。

例 6-13：product.java

```
// 由GLUE生成

package com.ecerami.wsdl.glue;

public class product
  {
  public java.lang.String name;

  public java.lang.String description;

  public double price;

  public java.lang.String SKU;
  }
```

GLUE 生成的最后一个文件是 Java 到 XML Schema 的映射文件（见例 6-14）。这个文件本身非常简洁，负责将 Java 类型转变为 XML Schema 类型或反之（见图 6-13）。根 complexType 元素表明，product 类型元素应该被转化成 com.ecerami.wsdl.glue 中的 product 类。在根复合类型中，XML Schema 类型和公共 Java 变量间存在一一对应的关系。例如，元素 name 被映射为 product.name 变量，指定其类型为字符串。同样，元素 price 对应 product.price 变量，其类型被指定为双精度型。

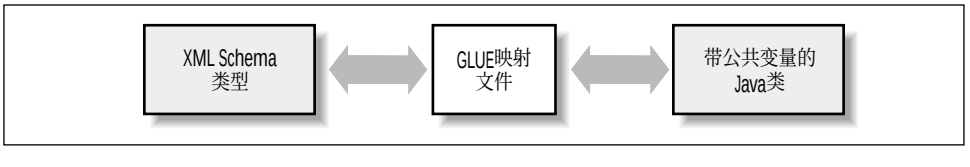

图 6-13：GLUE Java 到 XML Schema 的映射文件

例 6-14：Product_Service.map

```
<?xml version='1.0' encoding='UTF-8'?>
<!-- 由 GLUE 生成 -->
<mappings xmlns='http://www.themindelectric.com/schema/'>
 <schema
  xmlns='http://www.w3.org/2001/XMLSchema'
  targetNamespace='http://www.ecerami.com/schema'
  xmlns:electric='http://www.themindelectric.com/schema/'>
  <complexType name='product' electric:class='com.ecerami.wsdl.glue.product'>
   <sequence>
    <element name='name' electric:field='name' type='string'/>
    <element name='description'
       electric:field='description' type='string'/>
    <element name='price' electric:field='price' type='double'/>
    <element name='SKU' electric:field='SKU' type='string'/>
    </sequence>
  </complexType>
 </schema>
</mappings>
```

要调用 Product Service，你必须首先通过 GLUE `Mappings` 类显式地加载映射文件：

```
Mappings.readMappings("Product_Service.map");
```

接着你就可以像前面的例子那样访问服务了。例6-15 是完整的调用程序。下面是一些输出样例：

```
Product Service
Name:  Red Hat Linux
Description:  Red Hat Linux Operating System
Price:  54.99
```

例 6-15：Invoke_Product.java

```
package com.ecerami.wsdl;

import java.io.IOException;
import electric.xml.io.Mappings;
import electric.xml.ParseException;
import electric.registry.RegistryException;
import com.ecerami.wsdl.glue.*;

/**
 * SOAP 调用程序。用 Product_ServiceHelper 来调用产品
```

```
 * SOAP服务。  所有其他.java文件都由GLUE自动生成
 * by GLUE.
*/
public class Invoke_Product {

  /**
   * 通过SOAP服务获得产品
   */
  public product getProduct (String sku) throws Exception {
    //  加载Java到XML的映射或反之
    Mappings.readMappings("Product_Service.map");
    //  调用服务
    IProduct_Service service = Product_ServiceHelper.bind();
    product prod = service.getProduct(sku);
    return prod;
  }

  /**
   * main方法
   */
  public static void main (String[] args) throws Exception {
    Invoke_Product invoker = new Invoke_Product();
    System.out.println ("Product Service");
    product prod = invoker.getProduct("A358185");
    System.out.println ("Name:  "+prod.name);
    System.out.println ("Description:  "+prod.description);
    System.out.println ("Price:  "+prod.price);
 }
}
```

虽然这个文件的代码非常少，但它能做非常实际的工作。请务必查看 The Mind Electric公司的网站（*http://themindelectric.com/*）以获取GLUE产品的最新发布。

第五部分

UDDI

第七章

UDDI 精髓

UDDI 是一个描述、发现和连接 Web 服务的技术规范，因此是蓬勃发展的 Web 服务协议栈中最关键的部分。有了 UDDI，公司不但可以发布 Web 服务，还能查找 Web 服务。本章将全面介绍 UDDI，具体包括以下几个主题：

- UDDI 的主要概念和发展历史；
- UDDI 的主要作用，如它在供应链管理领域的潜在影响；
- UDDI 的各个技术方面，包括对 UDDI 数据模型的详细解释；
- 如何通过基于 Web 的接口查找 UDDI，如何使用 UDDI 可编程 API；
- 如何向 UDDI 发布新公司和服务；
- 面向 Java、Perl 和微软 COM 的流行 UDDI 实现。

UDDI 简介

UDDI 核心由两个部分组成。第一，UDDI 是一个建立业务和 Web 服务的分布式目录的技术规范。数据存储在特定的 XML 格式中，UDDI 规范包括搜索已有数据和发布新数据的 API 细节。第二，UDDI 业务注册中心（Business Registry）（经常被称为 UDDI“ 服务群（cloud services）”）是 UDDI 规范的一个完全操作实现。

2001年5月由微软和IBM发起的UDDI注册中心已经实现了任何人都可以搜索已有UDDI数据，也实现了任何公司都可以注册自身及其服务。

UDDI中捕获的数据主要分为三类：

白页

白页包括某个公司的一般信息，如业务名称、业务描述、联系信息、地址和电话号码。还包括惟一的业务标识符，如Dun & Bradstreet D-U-N-S® Number。

黄页

黄页包括有关公司或公司提供的服务的一般分类信息，如行业、产品或根据标准分类法确定的地理代码。

绿页

绿页包括Web服务的技术信息。一般包括指向外部规范的指针和调用Web服务的地址。UDDI并不限于描述基于SOAP的服务，它还可以用来描述任何其他服务，从单个网页或邮件地址直到SOAP、CORBA和Java RMI服务。

UDDI的简短历史

2000年9月，微软、IBM和Ariba宣布了UDDI 1.0。此后，UDDI的建设者发展到280多家公司（你可以从*http://www.uddi.org/community.html*获得完整的UDDI成员清单）。

2001年5月，微软和IBM推出了第一个UDDI操作入口站点，这标志着UDDI注册中心的诞生。2001年6月，UDDI推出了2.0版本，这个版本包括以下新特点：

- 加强了对国际化的支持。如业务可以用多种语言描述自身及其服务。
- 加强了对描述复杂组织的支持。如业务可以发布业务单位、部门或分公司，并将它们绑定在同一保护伞下。
- 改进了一组搜索选项。

到本书撰稿时为止，微软和IBM站点实现了1.0规范，并且在不远的将来会支持2.0。根据最初的计划，UDDI小组将发布三个UDDI版本，并将规范转变为一个合适的标准。

为什么选择UDDI？

UDDI初看起来好像非常简单，但它包含一些容易忽视的微妙之处。让我们先来看一看UDDI以后在某个行业的影响。

为了使概念尽可能具体，我们以半导体行业为例。目前，信息技术、电子元件和半导体制造行业大约有400家公司加入了一个名为RosettaNet的联盟。这个联盟专门为电子业务和供应链管理建立标准处理和接口。它主要的成就之一是建立了PIP（Partner Interface Processes，合作伙伴接口处理）。PIP是一个基于XML的接口，实现了两个贸易伙伴之间交换数据。现在已经有许多PIP，如：

PIP2A2
: 使一个合作伙伴可以向另一个伙伴查询产品信息。

PIP3A2
: 使合作伙伴可以查询某个产品的价格和是否有货。

PIP3A4
: 使合作伙伴可以提交一个电子购买订单，并得到订单认可。

PIP3B4
: 使合作伙伴可以查询某次运输的状态。

通过加快采用PIP的步伐，RosettaNet将会大幅度提高合作伙伴间的互操作性，使供应链管理变得更加灵活。这样，整体效率就提高了，成本也随之下降，从而激励各参与公司遵循PIP标准。

注意：你可以从 *http://www.rosettanet.org* 获得更多有关RosettaNet的信息。

2001年4月，在UDDI注册的PIP有83个。让我们来看看两个虚构的公司，看看它们在不久的将来如何支配UDDI。

场景一：向UDDI发布

首先，我们来看看Acme Parts。Acme Parts是一个普通的半导体小部件供应商，已经加入了RosettaNet联盟。最近它更新了它的电子业务服务，以遵循RosettaNet规范的一部分。例如，Acme可以让合作伙伴查找某个产品的信息、产品能否获得以及产品价格，合作伙伴还可以提交电子购买订单，并经常追踪订单状态。

Acme渴望加入现有的供应链，所以它在UDDI内注册了自己，并且注册了它的每个电子业务服务。它还为每个服务注明了实现的技术规范。例如，Acme Parts: Submit Purchase Order注明要遵循RosettaNet PIP3A4。

通过在UDDI内注册，Acme宣传了其服务，并使顾客容易发现它使用的技术标准。可以想像，将来Acme可以购买到新的可识别UDDI的电子商务软件，这种软件熟悉UDDI协议，能自动注册公司及其服务。

场景二：搜索UDDI

接着，我们来看看United Semiconductor。United Semiconductor是Acme Parts和其他许多供应商的客户。为了降低成本，它想把它所有的供应商集成为一个相干系统。

United有两个选择。首先，我们假设United和Acme已经正式确立了合作伙伴关系。这样，United就可以在UDDI业务注册中心查找Acme，确定从Acme那里可以得到哪些服务，这些服务是否遵循了相同的RosettaNet标准。United还可确定每个服务准确的绑定点，如可以从*http://www.acmeparts.com/services/po*获得Acme Parts: Submit Purchase Order服务。因此，United有了所有必需的信息，可以将Acme无缝地加入到它的供应链中，并且立即就可以开始提交电子购买订单。

第二个选择是，United还可以查找其他供应商。或者为了提交电子购买订单，它可以查找所有实现了特定 PIP 的公司。

UDDI 技术回顾

UDDI 的技术体系结构由三个部分构成：

UDDI 数据模型

UDDI 数据模型是一个描述业务和 Web 服务的 XML Schema。本章后面“ UDDI 数据模型 ”一节将详细介绍这个数据模型。

UDDI API

这是一个用于搜索和发布 UDDI 数据的、基于 SOAP 的 API。

UDDI 服务群

这是一个提供UDDI规范实现和根据预定基础使所有数据同步的操作入口站点。

UDDI 服务群现在由微软和 IBM 提供。Ariba 最初也打算提供一个操作入口，但最终还是退出了约定。不久的将来，会有更多来自其他公司的操作入口加入进来，如 Hewlett-Packard 公司。（要想得到最新的操作入口站点列表，请访问 *http://www.uddi.org/register.html*。）

当前的服务群提供一个逻辑上集中、但物理上分散的目录。这意味着提交到一个根节点的数据会在其他所有根节点上复制。目前，每 24 小时复制一次数据。

建立私人注册中心也是有可能的。如一个大公司为了注册它所有的内部 Web 服务，有可能建立它自己的UDDI注册中心。因为这种注册中心不能自动与根UDDI节点同步，所以不被当做 UDDI 群的一部分。

UDDI 数据模型

UDDI 包含一个描述以下四种核心信息的 XML Schema：

- businessEntity
- businessService
- bindingTemplate
- tModel

后面几节将描述这几种核心数据元素。图7-1说明了它们之间的从属关系。(此图根据“UDDI Data Structure Reference V1.0”中的图1制作，你可以通过访问*http://www.uddi.org/ pubs/DataStructure-V1.00-Open-20000930_2.doc*得到。)你可能急于试试基于Web的UDDI接口，但是，花点时间理解这些核心元素将会对你理解基于Web的接口和可编程API有莫大的帮助。

注意：真正的UDDI XML Schema可从以下URL：*http://www.uddi.org/schema/2001/uddi_v1.xsd*（UDDI 1.0）和*http://www.uddi.org/schema/uddi_v2.xsd*（UDDI 2.0）在线得获。

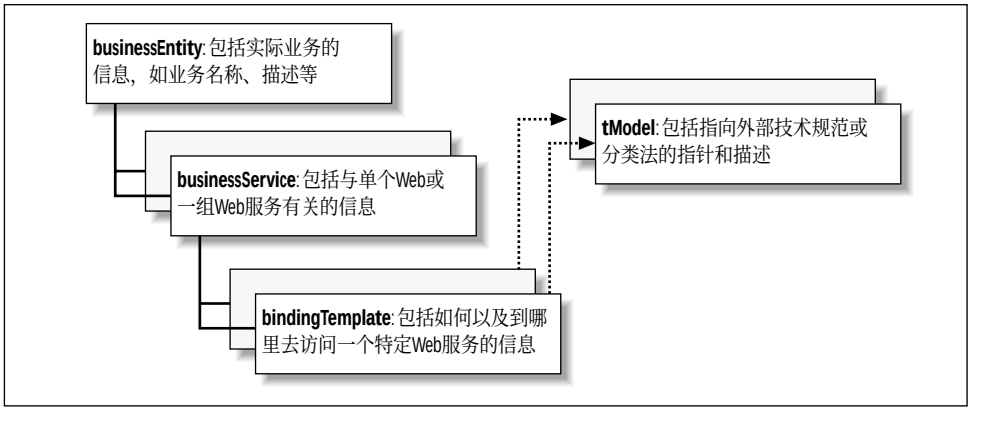

图7-1：UDDI数据模型

businessEntity

businessEntity元素包含有关实际业务的信息：业务名称、描述、地址、联系信息。例如，下面就是从微软businessEntity记录中摘出的一部分：

```
<businessEntity
   businessKey="0076b468-eb27-42e5-ac09-9955cff462a3"
   operator="Microsoft Corporation" authorizedName="Martin Kohlleppel">
   <name>Microsoft Corporation</name>
   <description xml:lang="en">Empowering people through great software
   - any time, any place and on any device is Microsoft's vision. As the
   worldwide leader in software for personal and business computing, we
   strive to produce innovative products and services that meet our
   customer's...

   </description>
   <contacts>
      <contact useType="Corporate Addresses and telephone">
         <description xml:lang="en">Corporate Mailing Addresses</
             description>
         <personName />
         <phone useType="Corporate Headquarters">(425) 882-8080</phone>
         <address sortCode="~" useType="Corporate Headquarters">
         <addressLine>Microsoft Corporation</addressLine>
            <addressLine>One Microsoft Way</addressLine>
            <addressLine>Redmond, WA 98052-6399</addressLine>
            <addressLine>USA</addressLine>
         </address>
      </contact>
      <contact useType="Technical Contact - Corporate UD">
         <description xml:lang="en">World Wide Operations</description>
            <personName>Martin Kohlleppel</personName>
            <email>martink@microsoft.com</email>
      </contact>
   </contacts>
   <identifierBag>
       <keyedReference
          tModelKey="uuid:8609c81e-ee1f-4d5a-b202-3eb13ad01823"
          keyName="D-U-N-S" keyValue="08-146-6849" />
   </identifierBag>
   <categoryBag>
       <keyedReference
          tModelKey="uuid:c0b9fe13-179f-413d-8a5b-5004db8e5bb2"
          keyName="NAICS: Software Publisher" keyValue="51121" />
   </categoryBag>
</businessEntity>
```

注册时，每个业务都会得到一个惟一的businessKey值，如微软的businessKey值就是0076b468-eb27-42e5-ac09-9955cff462a3。我们马上将会发现，这个键将业务与它发布的服务绑定起来。

除了基本的联系信息以外，businessEntity记录还可以包含业务标识符和业务种类。业务标识符可以是标识公司的任何惟一的值，例如，UDDI现在就不但可以接收Dun & Bradstreet D-U-N-S® Number，还可以接收Thomas Registry Supplier ID（具体细节，请参照表7-1和表7-2）。

表7-1：Dun & Bradstreet D-U-N-S® Number

名称	dnb-com:D-U-N-S
描述	Dun & Bradstreet D-U-N-S® Number
UUID	uuid:8609C81E-EE1F-4D5A-B202-3EB13AD01823
细节	Dun & Bradstreet D-U-N-S®（数据通用编号系统）Number是一个九位数的标识号码，用于标识业务和从属业务。现在有超过62 000 000个D-U-N-S号码。你可以从*http:// www.dnb.com/english/duns/*获得相关信息

表7-2：Thomas Register Supplier ID

名称	thomasregister-com:supplierID
描述	Thomas Register Supplier ID
UUID	uuid:B1B1BAF5-2329-43E6-AE13-BA8E97195039
细节	American Manufacturers的Thomas Register为超过168 000个美国和加拿大的公司提供惟一供应商ID。你可以从*http://www.thomasregister.com/*获得相关信息

从我们的第一个例子中你就可以发现，微软的businessEntity记录包含微软D&B D-U-N-S® Number。注意，为了包含多个值，businessEntity元素含有一个名为identifierBag的元素。这里的“bag”表明这是一个含有多个值的通用容器，使得公司可以注册多个业务标识符。

业务还可注册多个业务种类，包括行业、产品、服务或根据标准分类系统确定的地域代码。目前，UDDI预置了如下三个业务种类：

NAICS

北美行业分类系统（North American Industry Classification System, NAICS）提供行业分类（具体细节请参照表7-3）。

UNSPSC

通用标准产品和服务分类（Universal Standard Products and Service Classification，UNSPSC）提供产品和服务分类（具体细节请参照表 7-4）。

ISO 3166

国际标准化组织（International Organization for Standardization，ISO）提供一个世界地理标准分类法 ISO 3166（具体细节请参照表 7-5）。

从我们的第一例子中可以发现，微软记录包含一个 NAICS 分类“ NAICS: Software Publisher ”。

表 7-3：NAICS

名称	ntis-gov:naics:1997
描述	业务分类法：NAICS（1997 发布版）
UUID	uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2
细节	NAICS 为超过 19 000 个行业提供了一个六位数的行业代码。为了给北美自由贸易协会（North American Free Trade Association，NAFTA）提供一个标准的统计报告系统，加拿大、墨西哥和美国政府联合创建了这个分类系统。从 1997 年起，NAICS 取代了原来的标准工业分类（Standard Industry Classification，SIC）。你可以从 *http://www.naics.com/* 获得更多的有关信息。要查找你的 NAICS 代码，请访问 *http://www.naics.com/search.htm*

表 7-4：UNSPSC

名称	unspsc-org:unspsc:3-1
描述	产品分类法：UNSPSC （版本 3.1）
UUID	uuid:DB77450D-9FA8-45D4-A7BC-04411D14E384
细节	UNSPSC 为将产品和服务分类提供了标准代码。这个标准于 1998 年开发出来，现在由非营利性质的电子商业代码管理协会（Electronic Commerce Code Management Association，ECCMA）维护。UNSPSC 涵盖 54 个行业，包括针对每个可以想到的产品和服务的 12 000 多个代码。如代码 50131601 代表“ Fresh eggs（新鲜鸡蛋）”，代码 50131602 代表“ Egg substitutes（鸡蛋替代品）”。你可以从 *http://www.unspsc.org* 获得更多的有关信息

表 7-5：ISO 3166

名称	uddi-org:iso-ch:3166:1999
描述	UDDI 地理分类法 , ISO 3166
UUID	uuid:61668105-B6B6-425C-914B-409FB252C36D
细节	ISO 3166 由 237 个国家代码组成，由 ISO 维护。如中国的代码是 CN，美国的代码是 US。通过使用 ISO 3166，在 UDDI 注册的公司可以标识总部的地理位置或主要的业务地理区域。ISO 3166 还用于 Internet 顶级域国家代码。你可以从 *http://www.din.de/gremien/nas/nabd/iso3166ma/index.html* 获得更多的信息

businessService

`businessService`元素包含单个Web服务或一组相关Web服务的信息，包括名称、描述和`bindingTemplate`的一个选项列表（下一节将会介绍）。下面就是XMethods.net Delayed Stock Quote Service的一个`businessService`记录样例：

```
<businessService
   serviceKey="d5921160-3e16-11d5-98bf-002035229c64"
   businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
   <name>XMethods Delayed Stock Quotes</name>
   <description xml:lang="en">20-minute delayed stock quotes</description>
   <bindingTemplates>
      <bindingTemplate
        serviceKey="d5921160-3e16-11d5-98bf-002035229c64"
        bindingKey="d594a970-3e16-11d5-98bf-002035229c64">
      <description xml:lang="en">
         SOAP binding for delayed stock quotes service
      </description>
      <accessPoint URLType="http">
         http://services.xmethods.net:80/soap
      </accessPoint>
      <tModelInstanceDetails>
         <tModelInstanceInfo
         tModelKey="uuid:0e727db0-3e14-11d5-98bf-002035229c64" />
      </tModelInstanceDetails>
      </bindingTemplate>
   </bindingTemplates>
</businessService>
```

和`businessEntity`一样，每个`businessService`都有惟一的`serviceKey`。

bindingTemplate

`bindingTemplate` 元素包含有关从哪里以及如何获取一个特定服务的信息。例如，从上面的 XMethods 记录我们可以发现，Stock Quote Service 可以通过位于 *http://services.xmethods.net:80/soap* 的 SOAP 获得。UDDI 绑定不仅可以指向基于 HTTP 的服务，还可指向基于电子邮件、基于传真和基于 FTP 的服务，甚至还可以指向基于电话的服务（具体细节请参照表 7-6）。

表 7-6：UDDI 绑定选项

名称	描述	UUID	细节
uddi-org:smtp	基于电子邮件的服务	uuid:93335D49-3EFB-48A0-ACEA-EA102B60DDC6	标识一个通过 SMTP 电子邮件调用的服务。这个选项可以是某个人的电子邮件地址或一个基于 SMTP 的 SOAP 服务
uddi-org:fax	基于传真的服务	uuid:1A2B00BE-6E2C-42F5-875B-56F32686E0E7	标识一个通过传真传送调用的服务
uddi-org:ftp	基于 FTP 的服务	uuid:1A2B00BE-6E2C-42F5-875B-56F32686E0E7	标识一个通过 FTP 调用的服务
uddi-org:telephone	基于电话的服务	uuid:38E12427-5536-4260-A6F9-B5B530E63A07	标识一个通过电话调用的服务，包括语音交互和（或）按钮音交互
uddi-org:http	基于 HTTP 的服务	uuid:68DE9E80-AD09-469D-8A37-088422BFBC36	标识一个通过 HTTP 协议调用的 Web 服务。可以引用一个简单网页或较复杂的基于 HTTP 的 SOAP 应用程序
uddi-org:homepage	HTTP Web 的主页 URL	uuid:4CEC1CEF-1F68-4B23-8CB7-8BAA763AEB89	标识一个 Web 主页

tModel

tModel是最后一个核心数据类型，但也可能是最难掌握的一种类型。tModel代表技术模型，主要用于提供指向外部技术规范的指针。例如，XMethods Stock Quote Service的bindingTemplate提供有关从何处获取SOAP绑定的信息，但不提供有关如何连接这个服务的信息。tModel元素提供了一个指向外部技术规范的指针，从而填补了这一空白。例如，下面是XMethods Stock Quote绑定所引用的tModel：

```
<tModel
   tModelKey="uuid:0e727db0-3e14-11d5-98bf-002035229c64"
   operator="www.ibm.com/services/uddi" authorizedName="0100001QS1">
   <name>XMethods Simple Stock Quote</name>
   <description xml:lang="en">Simple stock quote interface</description>
   <overviewDoc>
       <description xml:lang="en">wsdl link</description>
       <overviewURL>
          http://www.xmethods.net/tmodels/SimpleStockQuote.wsdl
       </overviewURL>
   </overviewDoc>
   <categoryBag>
       <keyedReference
          tModelKey="uuid:c1acf26d-9672-4404-9d70-39b756e62ab4"
          keyName="uddi-org:types" keyValue="wsdlSpec" />
   </categoryBag>
</tModel>
```

该文档提供了一个指向外部技术规范的指针。在这个记录中，XMethods遵循用WSDL指定SOAP接口的极好习惯，提供了一个指向实际WSDL文件的指针。不一定总是指定一个WSDL文件，如你可以指定一个提供有关如何连接服务的详细指导的通用网页。

注意：tModel和*service type*（服务类型）两个术语经常是可以相互替代的。如微软的UDDI站点实现了“通过名称查找服务类型”，这相当于通过名称查找tModel。

---

tModel非常重要，因为你可以通过它识别服务实现的技术规范。更重要的是，当两个公司引用相同的tModel时，你可以确信它们实现的是相同的技术规范。

最后要注意一点，tModel的作用不限于Web服务技术规范。事实上，只要有必要，它就可以指向任何一个外部规范。如前面描述的所有业务标识符和分类都被注册成 tModel。特别是，D&B D-U-N-S® Number引用一个Dun & Bradstreet创建的外部标准，因此被注册成惟一的 tModel（uuid:8609C81E-EE1F-4D5A-B202-3EB13AD01823）。

tModel还可以引用其他 tModel，如XMethods tModel引用 uddi-org:types的tModel。类型tModel提供一个将规范分类的机制，在我们的例子中，XMethods被划分为 wsdlSpec种类。

搜索 UDDI

在牢固掌握了UDDI数据模型之后，我们现在开始学习搜索UDDI业务注册中心的机制。UDDI的初学者可能要从基于Web的UDDI接口开始。下一节我们将了解微软UDDI操作入口站点搜索样例的全过程，我们还会通过可编程UDDI API进行同样的搜索过程。

基于 Web 的搜索

搜索UDDI有三种选择：

- 现在，Internet Explorer支持通过实名关键词系统（Real Names Keyword System）解析UDDI名称。只要在Internet Explorer的地址栏输入“ uddi ”和待搜索的公司的名称，就可搜索到相应的UDDI，例如，输入“ uddi ariba ”。
- 你还可以直接登录微软UDDI网站 *http://uddi.microsoft.com*。
- 或者直接登录IBM UDDI网站 *http://www-3.ibm.com/services/uddi/*。

图7-2显示的是微软UDDI网站界面。你可以在左上角的Search框中输入业务名进行搜索。

图 7-2：微软 UDDI 主页（注意左上角的 Search 框）

为了研究 UDDI 的核心数据类型，我们来看一个共同的搜索路径。我们先搜索某个业务，选择一个业务服务，自顶向下搜索（drill down）一个绑定模板，最后获取 `tModel` 技术规范。搜索路径可概括为：

```
BusinessEntity
  → BusinessService
      → BindingTemplate
         → tModel record
```

首先，在 Search 框中输入“ Xmethods ”。图 7-3 显示的是一组相匹配的结果。

单击 XMethods 链接，完整的 XMethods `businessEntity` 记录就会显示出来，如图 7-4 所示。

下拉页面的滚动条，你将会看到 XMethods 提供的所有服务的完整列表。单击 XMethods Delayed Stock Quotes 链接，你就会看到如图 7-5 所示的完整的 `businessService` 记录。

图 7-3：搜索 XMethods

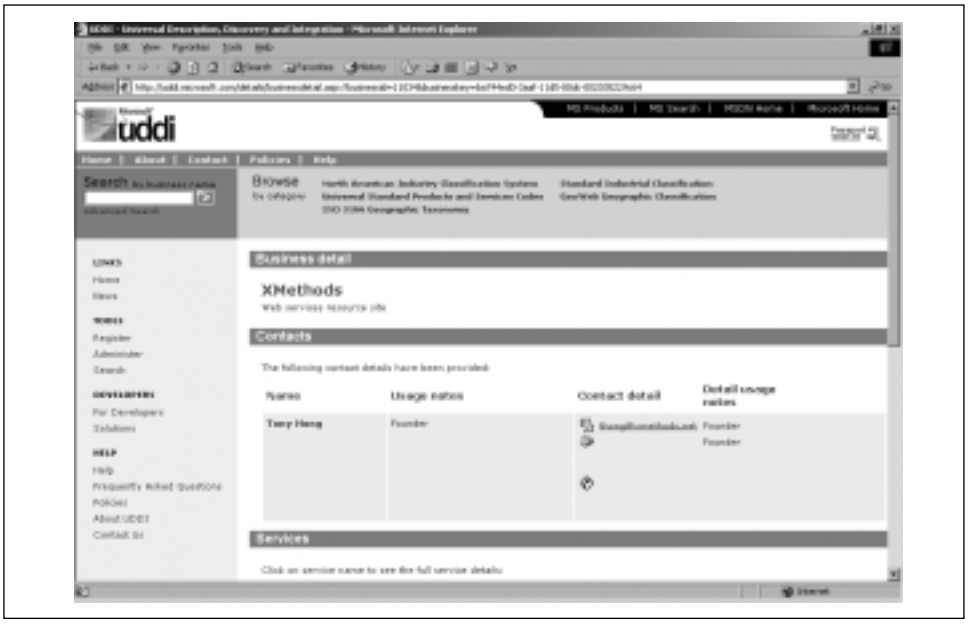

图 7-4：XMethods businessEntity 记录

图7-5显示了一个bindingTemplate，我们可以看到获取Stock Quote Service的URL。单击绑定下面的Details链接，你就会看到如图7-6所示的有关bindingTemplate的更多细节。

图7-5：XMethods Delayed Stock Quotes businessService记录

在Specification signature下面，我们可以看到，绑定引用了一个名为XMethods Simple Stock Quote的tModel。单击tModel记录，我们将看到如图7-7所示的结果。

你现在可以从图7-7获取XMethods WSDL规范文件。

注意：除了使用微软的UDDI浏览器以外，你还可以试试Soapclient.com（*http://www.soapclient.com/uddisearch.html*）提供的UDDI浏览器。有了这个服务，你可以搜索微软和IBM网站，直观地将所有方法向下考察到tModel记录。

图 7-6：The XMethods bindingTemplate 记录

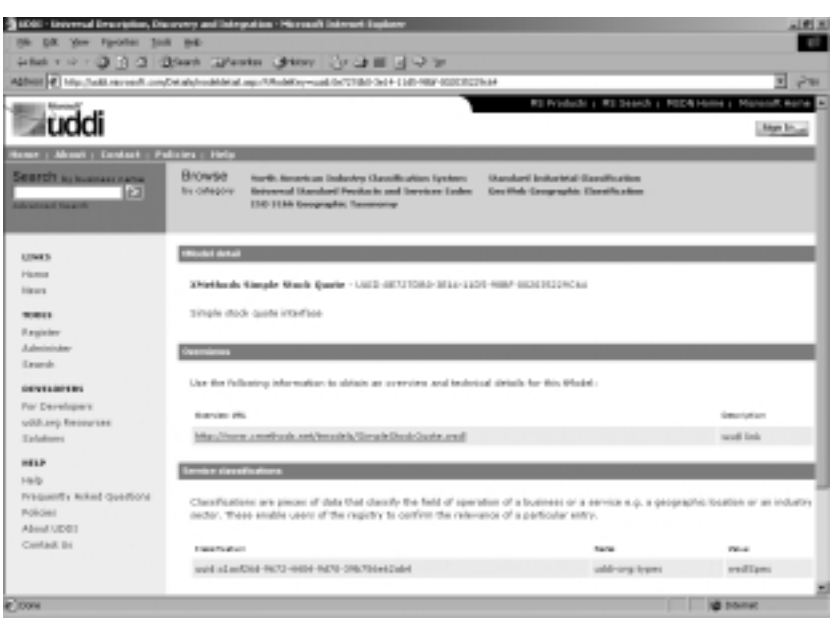

图 7-7：“ XMethods Simple Stock Quote ”tModel 记录

需要特别注意的是，微软高级搜索（Microsoft Advanced Search）实现了多条件搜索。用于搜索的条件有 NAICS 代码、UNSPSC 代码和 ISO 3166 国家代码。图 7-8 为用 NAICS 代码查找软件出版商的一个搜索样例。

图 7-8：高级搜索：根据 NAICS 代码搜索

如果你想从不同的角度浏览 UDDI 注册中心，那么可以试试 Antarcti.ca（*http://uddi.antarcti.ca*）提供的 Visual UDDI Map。这个可视化映射使你可以按种类向下搜索。图 7-9 和图 7-10 展示的是根据 NAICS 代码向下搜索。

UDDI 查询 API

UDDI API 是一个基于 SOAP 的协议，用于连接 UDDI 业务注册中心。API 广义地被分为两大类：查询 API（Inquiry API）和发布者 API（Publisher API）。前者提供搜索和获取功能，而后者提供插入和更新功能。

你可以从表 7-7 整体了解主要的 UDDI 查询函数。查询函数被进一步分为两组：*find_xxx* 和 *get_xxx*，前者提供一般的搜索功能，而后者根据惟一的键值检索完整的记录。

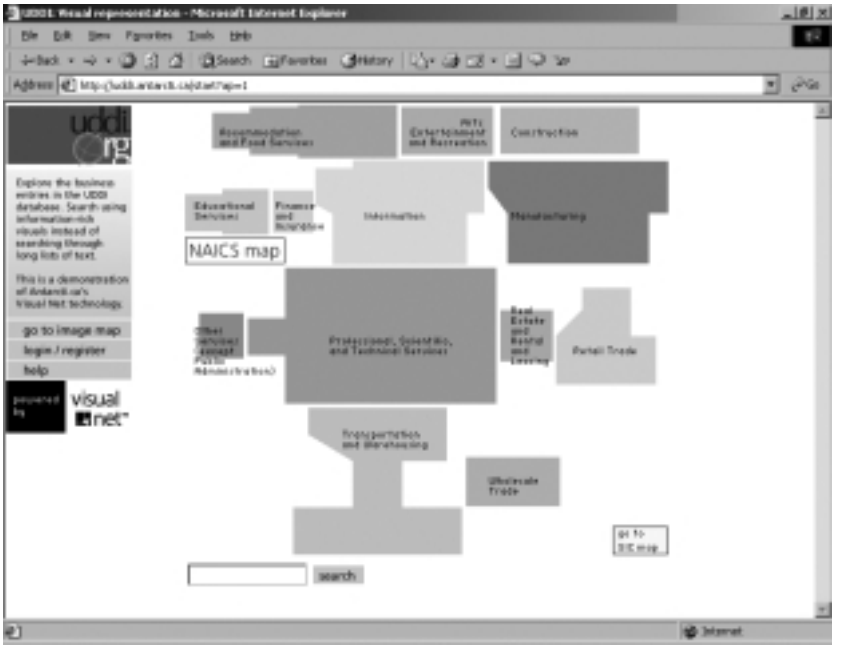

图 7-9：Antarcti.ca 提供的 Visual UDDI Map —— NAICS 窗口

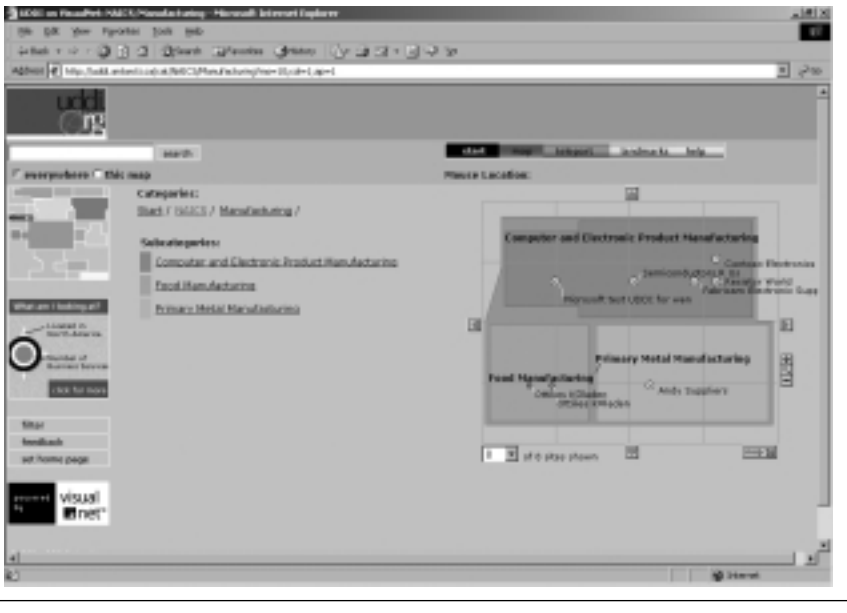

图 7-10：Antarcti.ca 提供的 Visual UDDI Map —— NAICS Manufacturing 窗口

表 7-7：UDDI 查询 API 的主要函数

函数名	描述
find_xxx 函数	
find_binding	搜索与指定服务相关的绑定
find_business	搜索符合指定条件的业务
find_service	搜索与指定业务相关的服务
find_tModel	搜索符合指定条件的 `tModel`
get_xxx 函数	
get_bindingDetail	获取一个完整的 `bindingTemplate` 记录
get_businessDetail	获取一个完整的 `businessEntity` 记录
get_serviceDetail	获取一个完整的 `businessService` 记录
get_tModelDetail	获取一个完整的 `tModel` 记录

下面的 URL 提供一个查询接口：

- 微软：*http://uddi.microsoft.com/inquire*
- IBM：*http://www-3.ibm.com/services/uddi/inquiryapi*

要发送查询函数，你可以向其中的一个 URL 发送 SOAP 请求。你还可以试试一个 UDDI 实现，本章后面将描述这些实现。

要想学习 UDDI API 的机制，最简单的方法是建立一个基于 Web 的、用于直接提交 UDDI 查询的接口。为方便建立接口，我已经在 *http://ecerami.com/uddi/* 建立了一个 UDDI 测试台。图 7-11 为这个测试台的界面。你可以在右边的文本框中输入任何 UDDI 函数。单击 Submit 后，你的查询就被发送到微软网站，并显示出 XML 结果。如果你不想输入查询的全部内容，则可以选择一个预定义查询的链接。前面的表 7-7 中所描述的每个查询函数都可以使用预定义查询。

有了查询函数表，让我们再回到基于 Web 的搜索接口中所描述的搜索路途，图 7-11 显示的是搜索接口。下面的每个例子都有一个请求和响应样例。为了使例子更简洁，我们略去了 SOAP 特定的细节。

图 7-11：UDDI 测试台：试验 UDDI 查询样例的简单接口

find_business 函数

我们先来看 *find_business* 函数。第八章提供了所有查询函数的完整的 API 规范。下面是搜索 XMethods 的 *find_business* 查询的一个样例：

```
<find_business generic="1.0" xmlns="urn:uddi-org:api">
   <name>XMethods</name>
</find_business>
```

注意，请求用 `generic` 属性指定了 UDDI 1.0 版本。在 *find_business* 函数里，我们指定了一个 `name` 元素，并将它的值设为 `XMethods`。默认状态下，UDDI 忽略大小写，从左到右逐词搜索。UDDI 用字符 `%` 作为通配符，如 `%data%` 将找出所有含“ data ”的公司。

微软返回如下响应：

```
<businessList generic="1.0" operator="Microsoft Corporation"
   truncated="false" xmlns="urn:uddi-org:api">
```

```
      <businessInfos>
         <businessInfo businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
            <name>XMethods</name>
            <description xml:lang="en">Web services resource site</description>
            <serviceInfos>
               <serviceInfo
                  serviceKey="d5b180a0-4342-11d5-bd6c-002035229c64"
                  businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
                  <name>XMethods Barnes and Noble Quote</name>
               </serviceInfo>
               <serviceInfo
                  serviceKey="ed85f000-4345-11d5-bd6c-002035229c64"
                  businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
                  <name>XMethods Pacific Bell SMS Service</name>
               </serviceInfo>
               <serviceInfo
                  serviceKey="d5921160-3e16-11d5-98bf-002035229c64"
                  businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
                  <name>XMethods Delayed Stock Quotes</name>
               </serviceInfo>
               <serviceInfo
                  serviceKey="618167a0-3e64-11d5-98bf-002035229c64"
                  businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
                  <name>XMethods Currency Exchange Rates</name>
               </serviceInfo>
            </serviceInfos>
         </businessInfo>
      </businessInfos>
   </businessList>
```

该响应包含一个根 businessList 元素以及每个相匹配公司的 businessInfo 元素。如果 UDDI 操作入口只返回相匹配结果的部分列表，则 businessList 元素的 truncated 属性就会被设成 true。如果没有找到相匹配的公司，就将返回带零个子元素的 businessList 元素。在我们的例子中，微软找到了一个与 XMethods 相匹配的结果，并标记了其惟一的 businessKey。

get_businessDetail 函数

既然已获得惟一的 businessKey，我们就可以再次向微软查询完整的 businessEntity 记录。这一步通过 *get_businessDetail* 函数完成。

```
   <get_businessDetail generic="1.0" xmlns="urn:uddi-org:api">
      <businessKey>ba744ed0-3aaf-11d5-80dc-002035229c64</businessKey>
   </get_businessDetail>
```

这个方法获取每个指定 businessKey 的完整的 businessEntity 记录（如果愿意，你可以指定多个 businessKey）。

微软做出如下响应：

```
<businessDetail generic="1.0" operator="Microsoft Corporation"
   truncated="false" xmlns="urn:uddi-org:api">
   <businessEntity
      businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64"
      operator="www.ibm.com/services/uddi" authorizedName="0100001QS1">
      <discoveryURLs>
         <discoveryURL useType="businessEntity">
            http://www.ibm.com/services/uddi/uddiget?
            businessKey=BA744ED0-3AAF-11D5-80DC-002035229C64
            </discoveryURL>
         </discoveryURLs>
         <name>XMethods</name>
         <description xml:lang="en">Web services resource site</description>
         <contacts>
           <contact useType="Founder">
             <description xml:lang="en" />
             <personName>Tony Hong</personName>
             <phone useType="Founder" />
             <email useType="Founder">thong@xmethods.net</email>
             <address>
               <addressLine />
               <addressLine />
               <addressLine />
               <addressLine />
               <addressLine />
             </address>
           </contact>
         </contacts>
         <businessServices>
           [...有关业务服务的信息...]
         </businessServices>
      </businessEntity>
</businessDetail>
```

该响应包含一个根 businessDetail 元素和每个相匹配业务的 businessEntity 元素。如果没有相匹配的结果，将返回一个 E_invalidKeyPassed 错误（后面将简单地讨论错误处理过程）。

find_service 函数

在获得 businessKey 之后，我们就还可以查询所有与之相关的服务。这一步通过 *find_ service* 函数完成：

```
<find_service generic="1.0" xmlns="urn:uddi-org:api"
   businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
</find_service>
```

微软做出如下响应：

```
<serviceList generic="1.0" operator="Microsoft Corporation"
   truncated="false" xmlns="urn:uddi-org:api">
   <serviceInfos>
      <serviceInfo
         serviceKey="618167a0-3e64-11d5-98bf-002035229c64"
         businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
         <name>XMethods Currency Exchange Rates</name>
      </serviceInfo>
      <serviceInfo
         serviceKey="d5921160-3e16-11d5-98bf-002035229c64"
         businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
         <name>XMethods Delayed Stock Quotes</name>
      </serviceInfo>
      <serviceInfo
         serviceKey="ed85f000-4345-11d5-bd6c-002035229c64"
         businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
         <name>XMethods Pacific Bell SMS Service</name>
      </serviceInfo>
      <serviceInfo
         serviceKey="d5b180a0-4342-11d5-bd6c-002035229c64"
         businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
         <name>XMethods Barnes and Noble Quote</name>
      </serviceInfo>
   </serviceInfos>
</serviceList>
```

该响应包含一个根serviceList元素和每个相匹配公司的serviceInfo元素。如果没有相匹配的结果，将返回带零个元素的 serviceList 元素。

如果想缩小搜索范围，你可以指定一个可选的 name 元素。例如，下面的查询搜索含有“ Quote ”的所有 XMethods 服务。

```
<find_service generic="1.0" xmlns="urn:uddi-org:api"
   businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
   <name>%Quote%</name>
</find_service>
```

get_serviceDetail 函数

有了惟一的 `serviceKey`，我们就可以获取完整的 `businessService` 记录。这一步通过 *get_serviceDetail* 函数完成。例如，下面的查询获取 XMethods Delayed Stock Quotes Service：

```
<get_serviceDetail generic="1.0" xmlns="urn:uddi-org:api">
    <serviceKey>d5921160-3e16-11d5-98bf-002035229c64</serviceKey>
</get_serviceDetail>
```

微软做出如下响应：

```
<serviceDetail generic="1.0" operator="Microsoft Corporation"
   truncated="false" xmlns="urn:uddi-org:api">
   <businessService
      serviceKey="d5921160-3e16-11d5-98bf-002035229c64"
      businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
      <name>XMethods Delayed Stock Quotes</name>
      <description xml:lang="en">
         20-minute delayed stock quotes
      </description>
      <bindingTemplates>
         <bindingTemplate
            serviceKey="d5921160-3e16-11d5-98bf-002035229c64"
            bindingKey="d594a970-3e16-11d5-98bf-002035229c64">
            <description xml:lang="en">
               SOAP binding for delayed stock quotes service
            </description>
            <accessPoint URLType="http">
               http://services.xmethods.net:80/soap
            </accessPoint>
            <tModelInstanceDetails>
               <tModelInstanceInfo
                  tModelKey="uuid:0e727db0-3e14-11d5-98bf-002035229c64" />
            </tModelInstanceDetails>
         </bindingTemplate>
      </bindingTemplates>
   </businessService>
</serviceDetail>
```

该响应包含一个根serviceDetail元素和每个相匹配服务的businessService元素。如果没有相匹配的结果，将返回一个E_invalidKeyPassed错误。

get_bindingDetail 函数

你可能已经注意到，*get_serviceDetail* 函数返回与所有 bindingTemplate 相关的完整细节（见前一节的响应样例）。如果只想得到绑定细节，那么你可以使用 *get_bindingDetail* 函数。例如，下面的查询获取 XMethods Stock Quote 服务的 SOAP 绑定：

```
<get_bindingDetail generic="1.0" xmlns="urn:uddi-org:api">
   <bindingKey>d594a970-3e16-11d5-98bf-002035229c64</bindingKey>
</get_bindingDetail>
```

你可以在这里指定一个或多个惟一的bindingKey值，微软返回完整的binding-Template 记录：

```
<bindingDetail generic="1.0" operator="Microsoft Corporation"
   truncated="false" xmlns="urn:uddi-org:api">
   <bindingTemplate
      serviceKey="d5921160-3e16-11d5-98bf-002035229c64"
      bindingKey="d594a970-3e16-11d5-98bf-002035229c64">
      <description xml:lang="en">
         SOAP binding for delayed stock quotes service
      </description>
      <accessPoint URLType="http">
         http://services.xmethods.net:80/soap
      </accessPoint>
      <tModelInstanceDetails>
         <tModelInstanceInfo
            tModelKey="uuid:0e727db0-3e14-11d5-98bf-002035229c64" />
      </tModelInstanceDetails>
   </bindingTemplate>
</bindingDetail>
```

通过可编程 API 查询一个动态绑定表明UDDI 的一种潜力。我们来看一个可识别 UDDI 的软件应用，这个应用尝试连接一个 Web 服务。如果访问失败，这个软件就会向 UDDI 查询更新的绑定信息，并试图在另一个新的访问点重新连接。这种方法在灾难恢复或连接备份系统时尤其有用。

get_tModelDetail 函数

最后，给定 tModelKey，我们可以获取完整的 tModel 记录。这一步通过 *get_tModelDetail* 函数实现。例如，下面的查询获取由 XMethods SOAP bindingTemplate 引用的 tModel 记录：

```
<get_tModelDetail generic="1.0" xmlns="urn:uddi-org:api">
   <tModelKey>uuid:0e727db0-3e14-11d5-98bf-002035229c64</tModelKey>
</get_tModelDetail>
```

微软做出如下响应：

```
<tModelDetail generic="1.0" operator="Microsoft Corporation"
   truncated="false" xmlns="urn:uddi-org:api">
   <tModel
      tModelKey="uuid:0e727db0-3e14-11d5-98bf-002035229c64"
      operator="www.ibm.com/services/uddi" authorizedName="0100001QS1">
      <name>XMethods Simple Stock Quote</name>
      <description xml:lang="en">Simple stock quote interface</description>
      <overviewDoc>
         <description xml:lang="en">wsdl link</description>
         <overviewURL>
            http://www.xmethods.net/tmodels/SimpleStockQuote.wsdl
         </overviewURL>
      </overviewDoc>
      <categoryBag>
         <keyedReference
            tModelKey="uuid:c1acf26d-9672-4404-9d70-39b756e62ab4"
            keyName="uddi-org:types" keyValue="wsdlSpec" />
      </categoryBag>
   </tModel>
</tModelDetail>
```

该响应包含一个根 tModelDetail 元素和每个相匹配 tModel 的 tModel 元素。如果没有相匹配的结果，将返回一个 E_invalidKeyPassed 错误。

错误处理

发生错误时，UDDI 操作入口将返回一个处理报告（disposition report）。处理报告包括有关错误原因的具体细节。例如，下面的查询是无效的，因为它没有引用所需的 name 元素，而是引用了一个非法的 company 元素。

```
<find_business generic="1.0" xmlns="urn:uddi-org:api">
   <company>XMethods</company>
</find_business>
```

微软以返回一个完整的处理报告作为响应：

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <soap:Fault>
         <faultcode>soap:Client</faultcode>
         <faultstring>System.Web.Services.Protocols.SoapException ---&gt;
         System.Xml.Schema.XmlSchemaException: Element 'urn:uddi-org:api:
         find_business' has invalid content. Expected 'urn:uddi-org:api:
         findQualifiers urn:uddi-org:api:name urn:uddi-org:api:identifierBag
         urn:uddi-org:api:categoryBag urn:uddi-org:api:tModelBag
         urn:uddi-org:api:discoveryURLs'. An error occurred at (4, 2).
         at System.Xml.XmlValidatingReader.InternalValidationCallback
         (Object sender, ValidationEventArgs e)
         [Full Stack Trace here...]
         <detail>
            <dispositionReport
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema" generic="1.0"
               operator="Microsoft Corporation" xmlns="urn:uddi-org:api">
               <result errno="10500">
                  <errInfo errCode="E_fatalError">
                     Element 'urn:uddi-org:api:find_business' has
                     invalid content. Expected 'urn:uddi-org:api:
                     findQualifiers urn:uddi-org:api:name urn:uddi-org:
                     api:identifierBag urn:uddi-org:api:categoryBag
                     urn:uddi-org:api:tModelBag urn:uddi-org:api:
                     discoveryURLs'. An error occurred at (4, 2).
                  </errInfo>
               </result>
            </dispositionReport>
         </detail>
      </soap:Fault>
   </soap:Body>
</soap:Envelope>
```

UDDI 操作入口用 SOAP `Fault` 元素返回错误，并将处理报告插入 SOAP 的 `detail` 元素。这遵循的是一般的 SOAP 错误处理模式（有关的细节请参阅第三章）。

向 UDDI 发布

介绍完搜索 UDDI 的细节后，下面就该向 UDDI 发布了。你现在又可以选择使用基于 Web 的接口或使用可编程 API。微软和 IBM 都提供有测试注册中心，你可以在这种注册中心试着发布新数据。在向真正的服务器发布数据前，强烈建议你先用这些测试注册中心试一试。

基于 Web 的发布

要找到基于 Web 的测试注册中心，请使用如下 URL：

- 微软：*https://test.uddi.microsoft.com*
- IBM：*https://www-3.ibm.com/services/uddi/testregistry*

要找到产品注册中心，请使用在本章的“搜索 UDDI”一节中引用的 URL。

安全和用户身份验证

所有向 UDDI 注册中心的插入和更新都需要经过用户身份验证，并且必须通过 SSL 传输。但是，根据 UDDI 规范，每个操作入口站点都可自由实现自己的身份验证方案。正因为如此，每个业务都必须选择一个操作入口站点来发布所有的交易。从此站点发出的数据都会传播到其他根节点，但是，所有的更新和插入都必须在最初选定的站点完成。

微软通过 Microsoft Passport 提供身份验证。图 7-12 为一个登录窗口样例。发布 UDDI 数据时，你可以注册一个新的 Passport 账号或者输入一个已有 Passport 账号的用户名和密码。

发布业务实体

在下面的样例中，我们又回到了在本章前面讨论过的 Acme Parts 的情况。Acme Parts 发布了 Acme Parts: Submit Purchase Order Service。这个服务遵循 RosettaNet PIP3A4。Acme 想在 UDDI 注册它的业务和新服务。

图 7-12：Microsoft Passport 登录窗口

如果通过 Microsoft Passport 登录，你将会看到普通 UDDI 管理窗口。图 7-13 是一个窗口样例。要想加入一个新的业务，请单击 Add a New Business 链接。

图 7-14 显示了发布新业务实体的初始字段。填入 Acme Parts 的名称和描述，单击 Save 按钮。

图 7-15 是 Edit Business 页面。从这个页面，你可以加入联系信息、业务标识符和业务分类。

例如，单击 Add a classification，你就可以向下搜索到你特定的业务分类。图 7-16 是一个样例窗口。

微软为核心 UDDI 分类 NAICS、UNSPSC 和 ISO 3166 提供了向下搜索支持。它还支持另外两个分类：标准行业分类（Standard Industry Classification，SIC）和微软 GeoWeb 地理分类。图 7-17 是一个 NAICS 向下搜索功能样例的窗口快照。

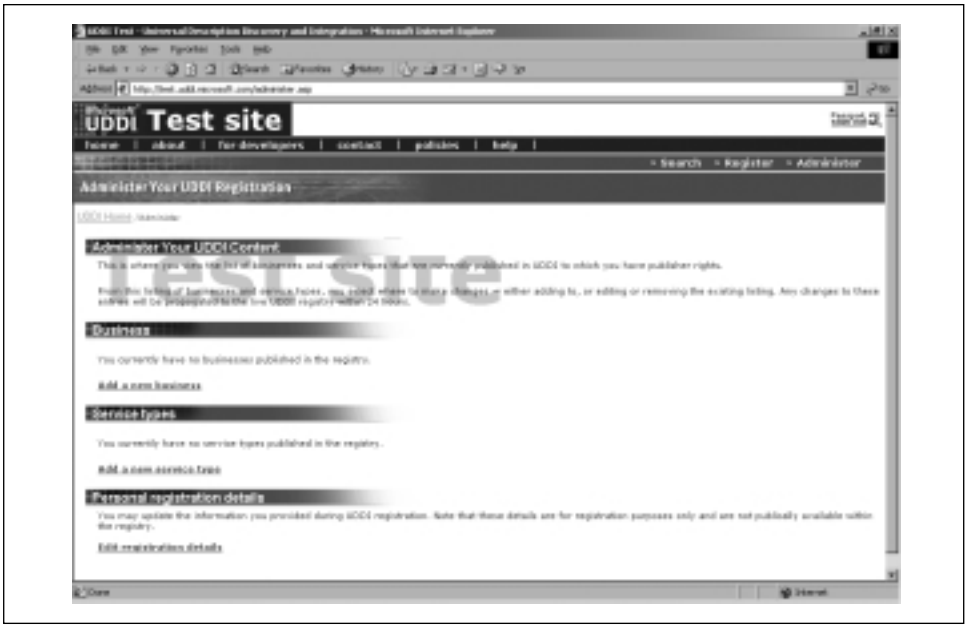

图 7-13：UDDI 管理窗口

图 7-14：增加一个新业务

在增加了联系方式、标识或分类后，一定要选择Publish 按钮，否则，你的数据不会被保存到注册中心。

图 7-15：Edit Business 页面

图 7-16：增加一个业务分类

发布业务服务

要增加一个业务服务，请返回到管理主页。现在你就可以看到你发布的业务实体了（见图 7-18）。要增加服务，请选择 Edit business，并单击 Add a Service。

图 7-17：增加 NAICS 代码

图 7-18：UDDI 管理页面显示最新发布的业务实体

图 7-19 显示的是发布新服务必需的初始字段。输入服务名称和描述，然后单击 Continue 按钮。

图 7-19：增加新服务

图 7-20 显示的是 Edit Service 网页。在此网页上单击 Define new binding。

图 7-21 显示的是创建新绑定必需的初始字段。你必须为每个绑定输入一个访问点、一个 URL 类型（HTTP、HTTPS、FTP、mailto、传真、电话或其他类型），你还可以输入一个描述。

对于 Acme Parts，我输入的是一个虚构的 SOAP 绑定接口 URL。单击 Continue 按钮，Edit Bindings 页面将被刷新，增加了一个新的规范签名部分，具体细节请参照图 7-22。

最后一步是加入你的规范签名。不幸的是，在标记 `tModel` 记录方面各站点并不完全一致 —— 在微软站点，服务类型和规范签名两个术语都指的是 `tModel` 记录。对于 Acme Parts，我们应该引用 RosettaNet PIP3A4。首先，单击 Add Specification Signature 链接。

图 7-20：Edit Service 页面

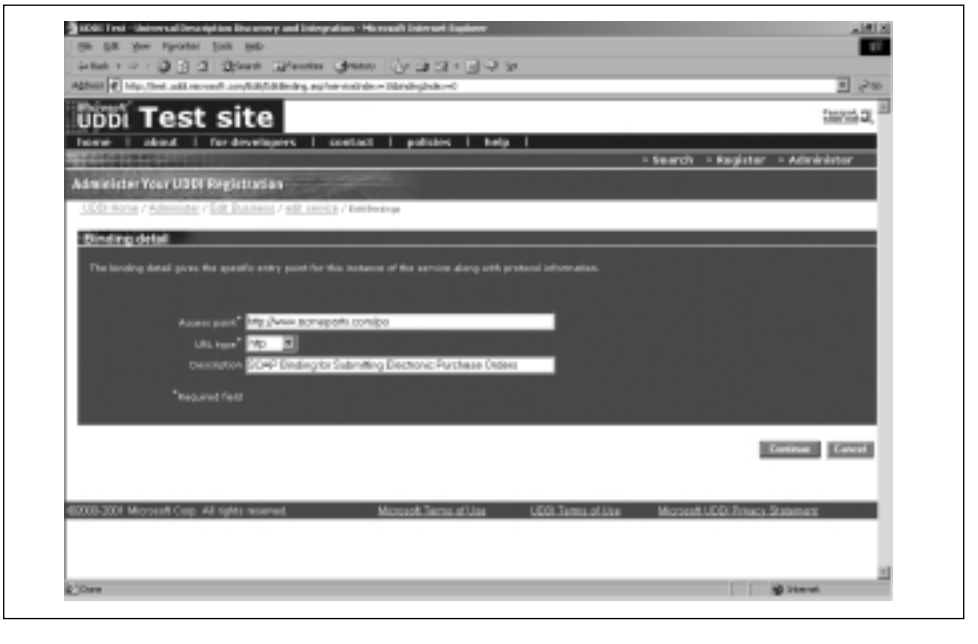

图 7-21：Edit Bindings 页面

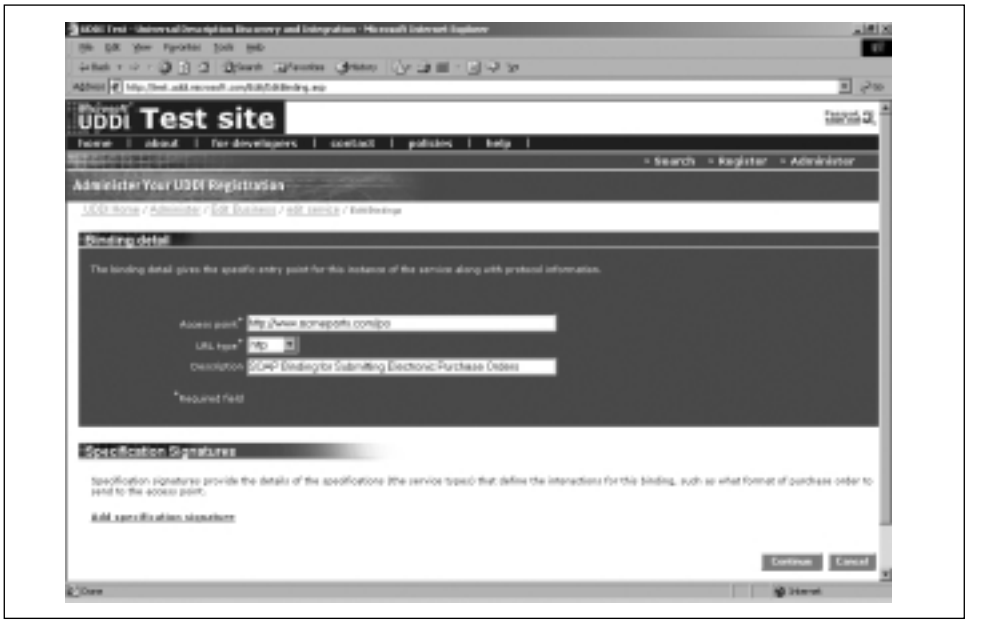

图 7-22：续 Edit Bindings 页面

你将会看到一个Find对话框，提示搜索所有已注册的`tModel`记录，具体细节请参照图 7-23。输入“ RosettaNet ”并单击Continue，你就会看到如图 7-24 所示的相匹配结果的列表。

图 7-23：搜索预先注册的 tModel 记录

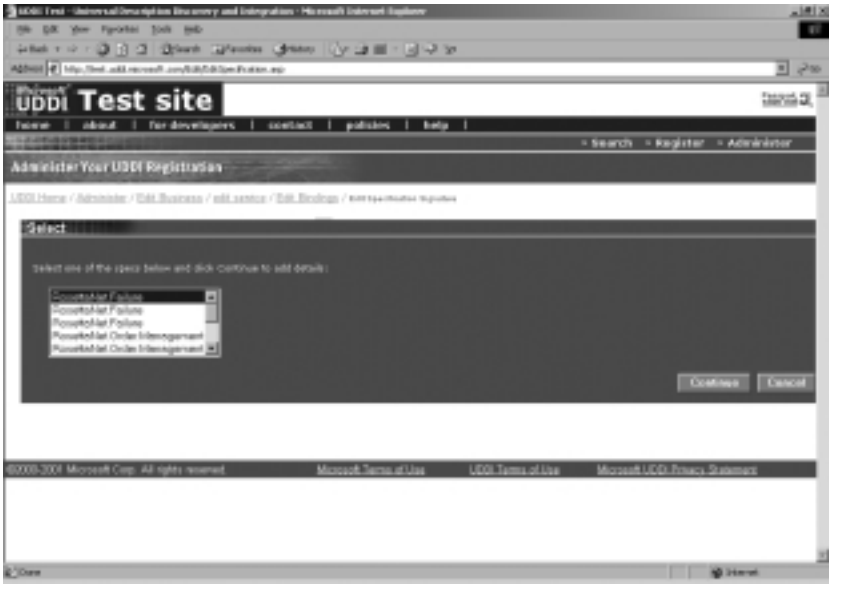

图 7-24：与“ RosettaNet ”相匹配的 tModel 搜索结果

不幸的是，RosettaNet 并没有在测试注册中心注册其所有的 PIP。如果这是一个产品注册中心，你可以下拉滚动条，为 PIP3A4 找到严格匹配的结果。现在，你可以选择最相近的结果，先单击第一个名为RosettaNet.Order.Management的项，然后单击 Continue，你就会看到如图 7-25 所示的 Edit Specification Signature 页面。

这个页面上的所有字段都是可选的，但是，如果你要为 `tModel` 记录增加更多的细节，就必须在这里完成。单击Continue按钮几次，直到你看到Publish按钮。和增加新业务记录一样，只有当你真正选定发布新记录后，注册中心才会保存你的记录。

发布 tModel 记录

发布新服务最困难的一点是确定服务采用的技术规范。用处最大的Web服务是那些采用公共规范的服务。例如，你的电子商务服务会实现 RosettaNet 标准或 OAGIS 标准（见本章后面的选读部分“ 开放应用程序集团 ”）。随着 Web 服务走

图 7-25：Edit Specification Signature 页面

向成熟，为各种应用程序定义接口的标准会越来越多。总有一天，金融交易、天气信息、体育快报、头条新闻发布和在线拍卖都有统一的标准。

发布Web服务时，第一步是确定是否有适合你的应用程序的标准，以及使用这些标准是否有意义。如果没有合适的标准，或者你想发布自己的规范时，你就需要注册自己的`tModel`记录。例如，Acme Parts可能想提供一个Product Availability Service（产品可获得性服务），而 RosettaNet 并没有包括这个实现，所以 Acme Parts 决定发布一个描述其接口的 WSDL 文件。

要增加一个新的服务类型，请回到主管理页面，选择 Add a New Service Type，你将会看到如图 7-26 所示的所需字段。

对于Acme Parts，我们已经增加了一个指向WSDL文件样例的引用，这个WSDL文件描述的是 SOAP 绑定。

图 7-26：增加新服务类型

开放应用程序集团

开放应用程序集团（Open Applications Group，OAG）是一个非营利联盟，主要从事电子商务集成。与 RosettaNet 非常相似，OAG 也发布了自己的规范 —— 开放应用程序集团集成规范（Open Applications Group Integration Specification，OAGIS）。例如，一个 OAGIS 接口可以包括以下几部分：

get_catalog
: 实现两个商业伙伴同步对话。

get_pricelist
: 实现一个商业伙伴向另一个伙伴请求当前价格列表。

get_prodavail
: 实现商业伙伴查询产品能否获得。

sync_exchangerate
: 使两个伙伴实现汇率同步。

和 RosettaNet 一样，OAGIS 规范已在 UDDI 注册。你还可以从 *http://www.openapplications.org* 获得更多的信息。

UDDI 的发布 API

向 UDDI 发布经常是通过基于 Web 的接口实现的。但是，在某些情况下，你可能要直接访问发布 API。例如，你可能正在开发自动注册新 Web 服务的 Web 服务软件，或者在灾难恢复时，你需要一个自动更新访问 URL 的绑定。

发布请求必须被发送到一个安全的 URL，并且这个 URL 要与查询 API 的 URL 截然不同。要向测试注册中心发布，请使用如下 URL：

- 微软：*https://test.uddi.microsoft.com/publish*
- IBM：*https://www.ibm.com/services/uddi/testregistry/protect/publish*

要向产品注册中心发布，请使用如下 URL：

- 微软：*https://uddi.microsoft.com/publish*
- IBM：*https://www.ibm.com/services/uddi/protect/publish*

表 7-8 列出了主要的 UDDI 发布函数。发布 API 分为三个主要的子类别：用户身份验证、保存数据和删除数据。在下一节中，我们将看到每个子类别的例子。

表 7-8：UDDI 的发布 API 的主要函数

函数名	描述
身份验证函数	
get_authToken	从操作入口站点请求一个身份验证标记。后面的 *save_xxx* 和 *delete_xxx* 函数都需要这个标记
discard_authToken	请求放弃某个身份验证标记，使之无效
save_xxx 函数	
save_binding	插入或更新一个 `bindingTemplate` 记录
save_business	插入或更新一个 `businessEntity` 记录
save_service	插入或更新一个 `businessService` 记录
save_tModel	插入或更新一个 `tModel` 记录

表 7-8：UDDI 的发布 API 的主要函数（续）

函数名	描述
delete_xxx 函数	
delete_binding	删除由 bindingKey 指定的 bindingTemplate 记录
delete_business	删除由 bindingKey 指定的 businessEntity 记录
delete_service	删除由 servicekey 指定的 businessService 记录
delete_tModel	隐藏由 tModelkey 指定的 tModel 记录。隐藏的 tModel 记录仍然能被另一个 UDDI 记录（如 bindingTemplate 记录）引用，但 *find_tModel* 函数的搜索结果不包括它。不能删除 tModel 记录

用户身份验证

只有通过身份验证的用户才能发布 UDDI 数据。像前面提到的那样，每个操作入口站点都可以自由实现自己的身份验证方案。

在发送保存或删除请求时，UDDI API 要求这个请求包含一个身份验证标记。要获得一个标记，你必须先发出一个 *get_authToken* 请求。这个函数要求你指定一个用户 ID 和密码。如下面的请求试图对 Acme Parts 的一个用户进行身份验证：

```
<get_authToken generic="1.0" xmlns="urn:uddi-org:api"
   userID="boss@acmeparts.com" cred="theBoss ">
</get_authToken>
```

如果不能识别用户，操作入口站点将返回一个 E_unknownUser 错误；否则就返回一个身份验证标记。例如，微软返回如下响应：

```
<authToken generic="1.0" xmlns="urn:uddi-org:api"
operator="http://uddi.microsoft.com">
   <authInfo>1BAAAAAAAHmykB2ylo*pV*pnrFoS4a*IblrgZSlpa
   jYC853wq9HIfsbaozLxYpG2Bo;1AAAAAAAAAKZi8QkOJ8BJfnMc
   *HeQCtOTHvu3TDkPEogcauDpvtHyxQGczEE0cj9bdl7C48RRyrK
   H7ReaF8OHivQEMltSEhgD8RNhmt0rHFdZoWkANFe*uSLmab4VvA
   FKLHFouvDh3MJ*9VK9YMLl4dg$$
   </authInfo>
</authToken>
```

接下来的对保存或删除数据的所有调用都要求使用这个标记。完成发布请求后，你可以选择调用 *discard_authToken* 函数放弃此标记。如：

```
<discard_authToken generic="1.0" xmlns="urn:uddi-org:api">
   <authInfo>1BAAAAAAAHmykB2ylo*pV*pnrFoS4a*IblrgZSlpa
   jYC853wq9HIfsbaozLxYpG2Bo;1AAAAAAAAAKZi8QkOJ8BJfnMc
   *HeQCtOTHvu3TDkPEogcauDpvtHyxQGczEE0cj9bdl7C48RRyrK
   H7ReaF8OHivQEMltSEhgD8RNhmt0rHFdZoWkANFe*uSLmab4VvA
   FKLHFouvDh3MJ*9VK9YMLl4dg$$
   </authInfo>
</discard_authToken>
```

如果发布成功，操作入口站点就返回一个 `E_success` 状态代码。如：

```
<dispositionReport generic="1.0"
   operator="Microsoft Corporation" xmlns="urn:uddi-org:api">
   <result errno="0">
      <errInfo errCode="E_success"></errInfo>
   </result>
</dispositionReport>
```

如果你确实想放弃身份验证标记，那么你可以总是调用 *get_authToken* 函数以获得一个新的标记。

保存 UDDI 数据

发布 API 使你可以插入新数据或更新已有的数据。例如，假设我们注册了我们的业务，现在想更新业务的描述。要保存一般的业务实体数据，请使用 *save_business* 函数：

```
<save_business generic="1.0" xmlns="urn:uddi-org:api">
   <authInfo>1BAAAAAAAHmykB2ylo*pV*pnrFoS4a*IblrgZSlpa
   jYC853wq9HIfsbaozLxYpG2Bo;1AAAAAAAAAKZi8QkOJ8BJfnMc
   *HeQCtOTHvu3TDkPEogcauDpvtHyxQGczEE0cj9bdl7C48RRyrK
   H7ReaF8OHivQEMltSEhgD8RNhmt0rHFdZoWkANFe*uSLmab4VvA
   FKLHFouvDh3MJ*9VK9YMLl4dg$$
   </authInfo>
   <businessEntity
      businessKey="03754729-3D3C-48E0-854A-1C1FD576CA5B">
      <name>Acme Parts</name>
      <description xml:lang="en">
          Supplier of fine semiconductor parts and Integrated Circuits
```

```
      </description>
   </businessEntity>
</save_business>
```

注意本例中身份验证标记的使用和更新的业务描述。一般来说，惟一键的值（本例中为businessKey）确定这是一个更新还是一个插入。如果惟一键被指定为空字符串（如business Key=""），则为插入一个新记录；否则为更新由惟一键表示的记录。

在响应中，UDDI 操作入口将回显这个新保存的数据。例如，微软返回：

```
<businessDetail generic="1.0" operator="Microsoft Corporation"
truncated="false" xmlns="urn:uddi-org:api">
   <businessEntity
      authorizedName="Ethan Cerami"
      businessKey="03754729-3D3C-48E0-854A-1C1FD576CA5B"
      operator="Microsoft Corporation">
      <discoveryURLs>
         <discoveryURL useType="businessEntity">
            http://test.uddi.microsoft.com/discovery?businessKey=0375
            4729-3D3C-48E0-854A-1C1FD576CA5B
         </discoveryURL>
      </discoveryURLs>
      <name>Acme Parts</name>
      <description xml:lang="en">
         Supplier of fine semiconductor parts and Integrated Circuits
      </description>
   </businessEntity>
</businessDetail>
```

要插入一个全新的记录，可以考虑使用 *save_tModel* 函数：

```
<save_tModel generic="1.0" xmlns="urn:uddi-org:api">
   <authInfo>1BAAAAAAAHmykB2ylo*pV*pnrFoS4a*IblrgZSlpa
   jYC853wq9HIfsbaozLxYpG2Bo;1AAAAAAAAAKZi8QkOJ8BJfnMc
   *HeQCtOTHvu3TDkPEogcauDpvtHyxQGczEE0cj9bdl7C48RRyrK
   H7ReaF8OHivQEMltSEhgD8RNhmt0rHFdZoWkANFe*uSLmab4VvA
   FKLHFouvDh3MJ*9VK9YMLl4dg$$
   </authInfo>
   <tModel tModelKey="">
      <name>Price Query Interface</name>
      <description xml:lang="en">
         SOAP Interface for querying product prices
      </description>
```

```
         <overviewDoc>
            <description xml:lang="en">
               WSDL File
            </description>
            <overviewURL>
               http://www.acmeparts.com/services/query_price.wsdl
            </overviewURL>
         </overviewDoc>
      </tModel>
   </save_tModel>
```

在这里，我们为 Acme Price Query 接口注册了一个新的 `tModel` 记录。因为 `tModelKey` 被设为空字符串，所以 UDDI 操作入口将它看成插入新记录。微软将返回如下响应：

```
<tModelDetail generic="1.0" operator="Microsoft Corporation"
truncated="false" xmlns="urn:uddi-org:api">
   <tModel authorizedName="Ethan Cerami" operator="Microsoft Corporation"
   tModelKey="uuid:01EBBD03-324D-4D7C-97EA-79B9C396D6EA">
      <name>Price Query Interface</name>
      <description xml:lang="en">
         SOAP Interface for querying product prices
      </description>
      <overviewDoc>
         <description xml:lang="en">WSDL File</description>
         <overviewURL>
            http://www.acmeparts.com/services/query_price.wsdl
         </overviewURL>
      </overviewDoc>
   </tModel>
</tModelDetail>
```

请注意，`tModel` 记录分配到一个自己的惟一 ID。

删除或隐藏 UDDI 数据

删除数据一般非常直接，只需指定惟一键的值。例如，下面的请求隐藏了新创建的 `tModel` 记录：

```
<delete_tModel generic="1.0" xmlns="urn:uddi-org:api">
   <authInfo>1BAAAAAAAHmykB2ylo*pV*pnrFoS4a*IblrgZSlpa
   jYC853wq9HIfsbaozLxYpG2Bo;1AAAAAAAAAKZi8QkOJ8BJfnMc
```

```
        *HeQCtOTHvu3TDkPEogcauDpvtHyxQGczEE0cj9bdl7C48RRyrK
        H7ReaF8OHivQEMltSEhgD8RNhmt0rHFdZoWkANFe*uSLmab4VvA
        FKLHFouvDh3MJ*9VK9YMLl4dg$$</authInfo>
        <tModelKey>uuid:01EBBD03-324D-4D7C-97EA-79B9C396D6EA"</tModelKey>
    </delete_tModel>
```

被隐藏的tModel记录仍可以被其他UDDI记录（如bindingTemplate记录）引用，但是，*find_tModel*函数生成的搜索结果不包括它。所以，不同于businessEntity、businessService或bindingTemplate记录，tModel记录不能真正地被删除。

UDDI 实现

现在有许多UDDI实现，这些实现使搜索或发布UDDI数据更容易，而不会陷入复杂的UDDI API中。这里是主要UDDI实现的摘要。

Java

Java有两个UDDI实现：

UDDI4J（面向Java的UDDI）（http://oss.software.ibm.com/developerworks/projects/uddi4j/）

UDDI4J最初由IBM创建。2001年1月，IBM将此代码转变为它自己的开源站点。第九章提供了具体信息。

jUDDI（http://www.juddi.org/）

jUDDI是UDDI注册中心的一个开源Java实现，是用于访问UDDI服务的一个工具包。jUDDI最初由Bowstreet公司开发，现在运行在SourceForge开源开发站点上。

微软 COM

Microsoft.com有一个UDDI实现。

UDDI 软件开发工具包（Software Development Kit, SDK）（http://uddi.microsoft.com/ developer/）

微软的 UDDI 软件开发工具包提供一个基于 COM 的 API，用于访问 UDDI 服务。

Perl

有一个面向 Perl 的 UDDI 实现。

SOAP::Lite（http://www.soaplite.com）

SOAP::Lite 为查询和发布提供一个基本的 UDDI 客户端。

Web 资源

下面的 Web 资源对你了解更多的 UDDI 细节将非常有用。

http://www.uddi.org/

UDDI 的官方站点。这个站点包括一个有用的 UDDI 技术白页，一组已加入 UDDI的组织、新闻发布、一个所有技术规范的完整文档和一个正式的UDDI FAQ。

http://groups.yahoo.com/group/uddi-technical

UDDI 技术新闻组。这个新闻组提供一个有关 UDDI 技术问题和 UDDI API 的活跃论坛。要加入一般的 UDDI 新闻组，请访问 *http://groups.yahoo.com/group/uddi-general*。

http://www.uddicentral.com

一个致力于发布 UDDI 信息和新闻的网站。

第八章

UDDI 查询 API 快速参考

本章针对 UDDI 查询 API 提供了一个快速参考。查询 API 使你可以搜索已有数据或从 UDDI 操作入口站点检索特定的记录。要全面了解 API 和 UDDI 数据模型，请参阅第七章。

表 8-1 简单概括了每个 UDDI 查询函数。在本章的后面，我们将详细介绍每个函数，并提供每个函数的如下信息：

- 简单描述
- UDDI 1.0 和 2.0 语法
- 所有函数参数的描述
- 一组可能的错误值

如有可能，我们还会为每个函数提供一个或多个 UDDI 1.0 例子，并且每个例子都与微软 UDDI 操作入口站点核对过。本章最后将简单回顾 UDDI 查找限定符，通过这些限定符可以给出更精确的搜索条件。

表 8-1：UDDI 查询 API

查询 API	描述	UDDI 版本
find_binding	搜索与指定服务相关的模板绑定	1.0，2.0
find_business	搜索与指定条件相匹配的业务	1.0，2.0
find_relatedBusinesses	通过 *uddi-org:relationships* 模型发现相关的业务	2.0
find_service	搜索与指定业务相关的服务	1.0，2.0
find_tModel	搜索与指定条件相匹配的 `tModel` 记录	1.0，2.0
get_bindingDetail	获取每个指定 `bindingKey` 的完整 `bindingTemplate`	1.0，2.0
get_businessDetail	获取每个指定 `businessKey` 的完整 `businessEntity`	1.0，2.0
get_businessDetailExt	获取每个指定 `businessKey` 的扩展 `businessEntity`	1.0，2.0
get_serviceDetail	获取每个指定 `serviceKey` 的 `businessService` 记录	1.0，2.0
get_tModelDetail	获取每个指定 `tModelKey` 的 `tModel` 记录	1.0，2.0

UDDI 查询 API

find_bindings

*find_bindings*函数搜索与指定的服务和指定的`tModel`记录相关的模板绑定记录。响应包含一个根`bindingDetail`元素，以及每个相匹配绑定的`bindingTemplate`元素。如果 UDDI 操作入口只返回部分匹配结果，则 `bindingDetail` 元素的`truncated`属性将被设为`true`；如果没有找到匹配的结果，则将返回一个带零个子元素的 `bindingDetail` 元素。

1.0 和 2.0 版本

1.0 语法:

```
<find_binding serviceKey="uuid_key" generic="1.0"
   [maxRows="nn"] xmlns="urn:uddi-org:api">
   [<findQualifiers/>]
   <tModelBag/>
</find_binding>
```

2.0 语法:

```
<find_binding serviceKey="uuid_key" [maxRows="nn"] generic="2.0"
   xmlns="urn:uddi-org:api_v2">
   [<findQualifiers/>]
   <tModelBag/>
</find_binding>
```

参数:

`serviceKey`

必需的 `uuid_key` 属性，指定相关的 `businessService`。

`maxRows`

可选属性，指定将返回的最大行数。如果超出了 `maxRows`，则 `bindingDetail` 元素的 `truncated` 属性将被设为 `true`。

`findQualifiers`

可选元素，覆盖默认的搜索功能。要获取更多信息，请参阅本章后面的“查找限定符”一节。

`tModelBag`

必需的 `uuid_key` 元素，用来指定 `tModel` 记录。如果指定了多个 `tModel` 记录，则将通过逻辑 AND 进行搜索。

例子:

下面的UDDI 1.0 例子搜索所有与XMethods Delayed Quote Service相关的SOAP绑定。`serviceKey d5921160-3e16-11d5-98bf-002035229c64`指定了XMethods Stock Quote Service，`tModel` 记录 `uuid:0e727db0-3e14-11d5-98bf-002035229c64` 引用了 SOAP 接口的 WSDL 规范。

```
<find_binding serviceKey="d5921160-3e16-11d5-98bf-002035229c64" generic="1.0"
   xmlns="urn:uddi-org:api">
   <tModelBag>
      <tModelKey>
         uuid:0e727db0-3e14-11d5-98bf-002035229c64
      </tModelKey>
   </tModelBag>
</find_binding>
```

下面是对该查询的响应：

```
<bindingDetail generic="1.0" operator="Microsoft Corporation"
   truncated="false" xmlns="urn:uddi-org:api">
   <bindingTemplate serviceKey="d5921160-3e16-11d5-98bf-002035229c64"
      bindingKey="d594a970-3e16-11d5-98bf-002035229c64">
      <description xml:lang="en">
         SOAP binding for delayed stock quotes service
      </description>
      <accessPoint URLType="http">
         http://services.xmethods.net:80/soap
      </accessPoint>
      <tModelInstanceDetails>
         <tModelInstanceInfo
            tModelKey="uuid:0e727db0-3e14-11d5-98bf-002035229c64" />
      </tModelInstanceDetails>
   </bindingTemplate>
</bindingDetail>
```

错误:

`E_invalidKeyPassed`

指定了一个无效的 `serviceKey` 属性。

`E_tooManyOptions`

指定的搜索项过多（只对 UDDI 1.0）。

`E_unsupported`

不支持指定的 `findQualifier`。

find_business

*find_business*函数搜索与指定条件相匹配的业务。响应包含一个根`businessList`元素，以及每个相匹配公司的一个`businessInfo`元素。如果 UDDI 操作入口只

返回部分匹配结果，则businessList元素的truncated属性将被设为true；如果没有找到相匹配的结果，则将返回一个带零个子元素的 businessList 元素。

1.0 和 2.0 版本

1.0 语法:

```
<find_business generic="1.0" [maxRows="nn"]
   xmlns="urn:uddi-org:api">
   [<findQualifiers/>]
   [<name/>]
   [<identifierBag/>]
   [<categoryBag/>]
   [<tModelBag/>]
   [<discoveryURLs>]
</find_business>
```

2.0 语法:

```
<find_business generic="2.0" [maxRows="nn"]
    xmlns="urn:uddi-org:api_v2">
    [<findQualifiers/>]
    [<name/> [<name/>]...]
    [<discoveryURLs/>]
    [<identifierBag/>]
    [<categoryBag/>]
    [<tModelBag/>]
</find_business>
```

参数:

maxRows

可选属性，指定将返回的最大行数。如果超出了maxRows，则businessList元素的 truncated 属性将被设为 true。

findQualifiers

可选元素，覆盖默认的搜索功能。例如，查找限定符exactNameMatch将与业务名称严格匹配。要获取更多细节，请参阅本章后面的“ 查找限定符 ”一节。

name

业务的全称或部分名称。UDDI 2.0 让你可以指定多达五个业务名称。默认

操作是进行严格的自左至右的逐词搜索。用%作为通配符，例如%data%将查找出所有含“ data ”的公司。

discoveryURLs

可选元素，通过发现URL 搜索。如果指定了多个discoveryURL，则将通过逻辑 OR 进行搜索。

identifierBag

可选元素，通过标识符搜索。例如，你可以用 Dun & Bradstreet D-U-N-S® Number 搜索。如果指定了多个标识符，则通过逻辑 OR 搜索。

categoryBag

可选元素，通过类别搜索。例如，你可以根据 NAICS 代码搜索。如果指定了多个类别，则将通过逻辑 AND 搜索。

tModelBag

可选元素，通过 tModel记录搜索。如果指定了多个tModel，则将通过逻辑 AND 搜索。

例子:

下面为三个 1.0 版的查询例子：

- 搜索所有以“ XMethods ”开头的业务：

```
<find_business generic="1.0" xmlns="urn:uddi-org:api">
   <name>XMethods</name>
</find_business>
```

- 搜索所有带指定 Dun & Bradstreet D-U-N-S® Number 的业务：

```
<find_business generic="1.0" xmlns="urn:uddi-org:api">
   <identifierBag>
   <keyedReference tModelKey="uuid:8609c81e-ee1f-4d5a-b202-3eb13ad01823"
      keyName="dnb-com:D-U-N-S" keyValue="04-693-3052" />
   </identifierBag>
</find_business>
```

- 搜索所有用 NAICS 代码注册广告的业务：

```
<find_business generic="1.0" xmlns="urn:uddi-org:api">
   <categoryBag>
      <keyedReference tModelKey="uuid:70a80f61-77bc-4821-a5e2-
```

```
            2a406acc35dd"
         keyName="Advertising" keyValue="7310" />
    </categoryBag>
</find_business>
```

下面是对第一个查询例子的响应（搜索“ XMethods ”）：

```
<businessList generic="1.0" operator="Microsoft Corporation"
   truncated="false" xmlns="urn:uddi-org:api">
   <businessInfos>
      <businessInfo businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
         <name>XMethods</name>
         <description xml:lang="en">Web services resource site</description>
         <serviceInfos>
            <serviceInfo serviceKey="d5b180a0-4342-11d5-bd6c-002035229c64"
               businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
               <name>XMethods Barnes and Noble Quote</name>
            </serviceInfo>
            <serviceInfo serviceKey="ed85f000-4345-11d5-bd6c-002035229c64"
               businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
               <name>XMethods Pacific Bell SMS Service</name>
            </serviceInfo>
            <serviceInfo serviceKey="d5921160-3e16-11d5-98bf-002035229c64"
                businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
               <name>XMethods Delayed Stock Quotes</name>
            </serviceInfo>
            <serviceInfo serviceKey="618167a0-3e64-11d5-98bf-002035229c64"
                businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
               <name>XMethods Currency Exchange Rates</name>
            </serviceInfo>
         </serviceInfos>
      </businessInfo>
   </businessInfos>
</businessList>
```

错误：

`E_nameTooLong`

业务名过长。

`E_unsupported`

不支持指定的 `findQualifier`。

`E_tooManyOptions`

指定的搜索项过多。

find_relatedBusinesses

*find_relatedBusinesses*函数搜索所有与指定`businessKey`相关的业务。UDDI 2.0进一步支持描述复杂组织，如业务可以发布业务单位、部门或分公司，并把它们绑定在同一把保护伞下。通过`uddi-org:relationships tModel`记录（`uuid:807A2C6A-EE22-470D-ADC7-E0424A337C03`）可以创建业务关系。UDDI 2.0支持三个关系值：

parent-child
: 用于表明父－子关系。如拥有子公司的控股公司就会选择发布父－子关系。

peer-peer
: 用于表明两个对等实体。如同一个公司的两个分公司将选择发布同辈关系。

identity
: 用于表明两个实体代表同一组织。

*find_relatedBusinesses*函数用于发现通过`uddi-org:relationships`模型相关联的业务。响应包含一个根`relatedBusinessesList`元素。如果UDDI操作入口只返回部分匹配结果，则`relatedBusinessesList`元素的`truncated`属性将被设为`true`；如果没有找到相匹配的结果，则将返回一个带零个子元素的`relatedBusinessesList`元素。

2.0 版本:

2.0 语法:

```
<find_relatedBusinesses generic="2.0" xmlns="urn:uddi-org:api_v2">
   [<findQualifiers/>]
   <businessKey/>
   [<keyedReference/>]
</find_relatedBusinesses>
```

参数:

`findQualifiers`
: 可选元素，覆盖默认的搜索功能。要获取更多细节，请参阅本章后面的“查找限定符”一节。

businessKey

指定 businessEntity 的必需 uuid_key。

keyedReference

可选元素，用于指定一个 uddi-org:relationship 值。keyedReference 需要三个属性：tModelKey、keyName 和 keyValue。例如，要获取带指定 businessKey的同辈关系的业务实体记录，应使用下面的keyedReference：

```
<keyedReference
   tModelKey="uuid:807A2C6A-EE22-470D-ADC7-E0424A337C03"
   keyName="uddi-org:relationships"
   keyValue="peer-peer" />
```

错误：

E_invalidKeyPassed

指定的 businessKey 属性无效。

E_unsupported

不支持指定的 findQualifier。

find_service

find_service 函数搜索与指定业务相关的服务。响应包含一个根 serviceList 元素和每个相匹配服务的一个 serviceInfo元素。如果UDDI操作入口只返回部分匹配结果，则 serviceList 元素的 truncated属性将被设为 true；如果没有找到相匹配的结果，则将返回一个带零个子元素的 serviceList 元素。

1.0 和 2.0 版本

1.0 语法：

```
<find_service businessKey="uuid_key" generic="1.0" [maxRows="nn"]
   xmlns="urn:uddi-org:api">
   [<findQualifiers/>]
   [<name/>]
   [<categoryBag/>]
   [<tModelBag/>]
</find_service>
```

2.0 语法:

```
<find_service businessKey="uuid_key" generic="2.0" [maxRows="nn"]
   xmlns="urn:uddi-org:api_v2">
   [<findQualifiers/>]
   [<name/>
   [<name/>]...]
   [<categoryBag/>]
   [<tModelBag/>]
</find_service>
```

参数:

`businessKey`

必需的 `uuid_key` 属性，指定相关的 `businessEntity`。

`maxRows`

可选属性，指定返回的最大行数。如果超过 `maxRows`值，那么 `serviceList` 元素的 `truncated` 属性将被设为 `true`。

`findQualifiers`

可选元素，覆盖默认的搜索功能。例如，查找限定符 `exactNameMatch` 将与服务名称严格匹配。要获取更多细节，请参阅本章后面的“ 查找限定符 ”一节。

`name`

服务的全称或部分名称。UDDI 2.0 让你可以指定多达 5 个服务名称。默认操作是进行严格的自左至右的逐词搜索。用 `%` 作为能配符，如 `%quote%` 将查找出所有含“ quote ”的服务。

`categoryBag`

可选元素，根据类别搜索。如果指定了多个类别，则将通过逻辑 AND 搜索。

`tModelBag`

可选元素，根据 `tModel` 记录搜索。如果指定了多个 `tModel`，那么将通过逻辑 AND 搜索。

例子:

下面是 1.0 版查询的两个例子：

- 搜索由 XMethods 提供的所有服务：

```
<find_service generic="1.0" xmlns="urn:uddi-org:api"
   businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
</find_service>
```

- 搜索由 XMethods 提供的所有含有“ quote ”的服务：

```
<find_service generic="1.0" xmlns="urn:uddi-org:api"
   businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
   <name>%Quote%</name>
</find_service>
```

下面是对第二个查询例子的响应：

```
<serviceList generic="1.0" operator="Microsoft Corporation"
   truncated="false" xmlns="urn:uddi-org:api">
   <serviceInfos>
      <serviceInfo serviceKey="d5921160-3e16-11d5-98bf-002035229c64"
         businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
         <name>XMethods Delayed Stock Quotes</name>
      </serviceInfo>
      <serviceInfo serviceKey="d5b180a0-4342-11d5-bd6c-002035229c64"
         businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
         <name>XMethods Barnes and Noble Quote</name>
      </serviceInfo>
   </serviceInfos>
</serviceList>
```

错误:

`E_invalidKeyPassed`

指定的 `businessKey` 属性无效。

`E_nameTooLong`

服务名过长。

`E_tooManyOptions`

指定的搜索项过多 （只对 UDDI 1.0 ）。

`E_unsupported`

不支持指定的 `findQualifier`。

find_tModel

find_tModel 函数搜索所有与指定条件相关的 `tModel` 记录。响应包含一个根 `tModelList` 元素，以及每个相匹配公司的一个 `tModelInfo` 元素。如果 UDDI 操作入口只返回部分匹配结果，则 `tModelList` 元素的 `truncated` 属性将被设为 `true`；如果没有找到相匹配的结果，则将返回一个带零个子元素的 `tModelList` 元素。

1.0 和 2.0 版本:

1.0 语法:

```
<find_tModel generic="1.0" [maxRows="nn"] xmlns="urn:uddi-org:api">
   [<findQualifiers/>]
   [<name/>]
   [<identifierBag/>]
   [<categoryBag/>]
</find_tModel>
```

2.0 语法:

```
<find_tModel generic="2.0" [maxRows="nn"] xmlns="urn:uddi-org:api_v2">
   [<findQualifiers/>]
   [<name/>]
   [<identifierBag/>]
   [<categoryBag/>]
</find_tModel>
```

参数:

`maxRows`

可选属性，指定将返回的最大行数。如果超过 `maxRows` 值，则 `tModelList` 元素的 `truncated` 属性将被设为 `true`。

`findQualifiers`

可选元素，覆盖默认的搜索功能。例如，查找限定符 `exactNameMatch` 将与 `tModel` 名严格匹配。要获取更多细节，请参阅本章后面的“查找限定符”一节。

`name`

`tModel`的全称或部分名称。默认操作是进行严格的自左至右的逐词搜索。用`%`作为能配符，如`%shipping%`将查找出所有含“ shipping ”的`tModel`。

`identifierBag`

可选元素，根据标识符搜索。如果指定了多个标识符，则将通过逻辑 OR 进行搜索。

`categoryBag`

可选元素，根据类别搜索。如果指定了多个类别，则将通过逻辑 AND 进行搜索。

例子:

下面是 1.0 版查询的两个例子：

- 查找所有用 OAGIS 注册的`tModel`记录：

```
<find_tModel generic="1.0" xmlns="urn:uddi-org:api">
   <name>OAGIS%</name>
</find_tModel>
```

- 查找所有用 RosettaNet 规范注册并含有“ Shipment ”的`tModel`记录：

```
<find_tModel generic="1.0" xmlns="urn:uddi-org:api">
   <name>RosettaNet%Shipment%</name>
</find_tModel>
```

下面是对第二个查询例子的响应：

```
<tModelList generic="1.0" operator="Microsoft Corporation"
   truncated="false" xmlns="urn:uddi-org:api">
   <tModelInfos>
      <tModelInfo tModelKey="uuid:e13fbf58-f69a-4c8a-a2f5-aecc5cca75e1">
         <name>Rosettanet-org:PIP4B2:NotifyOfShipmentReceipt:v1.0</name>
      </tModelInfo>
      <tModelInfo tModelKey="uuid:7b15e1a7-bbe7-4c32-8f44-64662a555b7c">
         <name>Rosettanet-org:PIP3B5:ChangeShipment:v01.00.00</name>
      </tModelInfo>
      <tModelInfo tModelKey="uuid:87c133f5-f521-420a-8316-9ff37e292796">
         <name>Rosettanet-org:PIP3B4:QueryShipmentStatus:v01.00.00</name>
      </tModelInfo>
      <tModelInfo tModelKey="uuid:cf073d7c-e297-470f-b7de-4ae18bd72ca9">
```

```
        <name>Rosettanet-org:PIP3B3:DistributeShipmentStatus:v01.00.00</name>
      </tModelInfo>
      <tModelInfo tModelKey="uuid:7c60881c-ad6c-4520-955b-49fe75b71d53">
        <name>Rosettanet-org:PIP3B2:NotifyOfAdvanceShipment:v01.00.00</name>
      </tModelInfo>
      <tModelInfo tModelKey="uuid:05b548d8-0c23-4249-8a68-b653f7c97d8a">
        <name>Rosettanet-org:PIP3B2:NotifyOfAdvanceShipment:v01.01.00</name>
      </tModelInfo>
    </tModelInfos>
  </tModelList>
```

要想得到更多有关 OAGIS 或 RosettaNet 的信息，请参阅第七章。

错误:

`E_nameTooLong`

`tModel` 名称过长。

`E_tooManyOptions`

指定的搜索项过多（只对 UDDI 1.0）。

`E_unsupported`

不支持指定的 `findQualifier`。

get_bindingDetail

*get_bindingDetail*函数获取每个指定`bindingKey`的完整`bindingTemplate`。响应包含一个根 `bindingDetail` 元素，以及每个相匹配绑定的一个 `bindingTemplate`元素。如果UDDI 操作入口只返回部分匹配结果，则 `bindingDetail` 元素的 `truncated`属性将被设为 `true`。如果没有找到相匹配的结果，则将返回一个 `E_invalidKeyPassed` 错误。

1.0 和 2.0 版本:

1.0 语法:

```
<get_bindingDetail generic="1.0" xmlns="urn:uddi-org:api">
   <bindingKey/>
   [<bindingKey/> ...]
</get_bindingDetail>
```

2.0 语法:

```
<get_bindingDetail generic="2.0" xmlns="urn:uddi-org:api_v2" >
   <bindingKey/>
   [<bindingKey/> ...]
</get_bindingDetail>
```

参数:

`bindingKey`

必需的 `uuid_key`，指定 `bindingTemplate`。

例子：

下面的 UDDI 1.0 例子获取一个与 XMethods Stock Quote Service 相关的 `bindingTemplate`：

```
<get_bindingDetail generic="1.0" xmlns="urn:uddi-org:api">
   <bindingKey>d594a970-3e16-11d5-98bf-002035229c64</bindingKey>
</get_bindingDetail>
```

下面是对这个查询的响应：

```
<bindingDetail generic="1.0" operator="Microsoft Corporation"
   truncated="false" xmlns="urn:uddi-org:api">
   <bindingTemplate serviceKey="d5921160-3e16-11d5-98bf-002035229c64"
      bindingKey="d594a970-3e16-11d5-98bf-002035229c64">
      <description xml:lang="en">
         SOAP binding for delayed stock quotes service
      </description>
      <accessPoint URLType="http">
         http://services.xmethods.net:80/soap
      </accessPoint>
      <tModelInstanceDetails>
         <tModelInstanceInfo
            tModelKey="uuid:0e727db0-3e14-11d5-98bf-002035229c64" />
      </tModelInstanceDetails>
   </bindingTemplate>
</bindingDetail>
```

错误:

`E_invalidKeyPassed`

指定的 `bindingKey` 无效。

get_businessDetail

*get_businessDetail*函数获取每个指定businessKey的完整businessEntity。响应包含一个根businessDetail元素和每个相匹配业务的一个businessEntity元素。如果UDDI操作入口只返回部分匹配结果，则businessDetail的truncated属性将被设为true。如果没有找到相匹配的结果，则将返回一个E_invalidKeyPassed错误。

1.0和2.0版本：

1.0 语法:

```
<get_businessDetail generic="1.0" xmlns="urn:uddi-org:api">
   <businessKey/>
   [<businessKey/> ...]
</get_businessDetail>
```

2.0 语法:

```
<get_businessDetail generic="2.0" xmlns="urn:uddi-org:api_v2">
   <businessKey/>
   [<businessKey/> ...]
</get_businessDetail>
```

参数:

businessKey

必需的uuid_key，指定businessEntity。你可以指定多个businessKey。

例子：

下面的UDDI 1.0例子获取XMethods公司的businessEntity记录：

```
<get_businessDetail generic="1.0" xmlns="urn:uddi-org:api">
   <businessKey>ba744ed0-3aaf-11d5-80dc-002035229c64</businessKey>
</get_businessDetail>
```

下面是对这个查询的完整响应:

```
<businessDetail generic="1.0" operator="Microsoft Corporation"
   truncated="false" xmlns="urn:uddi-org:api">
```

```
<businessEntity
   businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64"
   operator="www.ibm.com/services/uddi" authorizedName="0100001QS1">
   <discoveryURLs>
      <discoveryURL useType="businessEntity">
         http://www.ibm.com/services/uddi/uddiget?businessKey=
         BA744ED0-3AAF-11D5-80DC-002035229C64
      </discoveryURL>
   </discoveryURLs>
   <name>XMethods</name>
   <description xml:lang="en">Web services resource site</description>
      <contacts>
         <contact useType="Founder">
            <description xml:lang="en" />
            <personName>Tony Hong</personName>
            <phone useType="Founder" />
            <email useType="Founder">thong@xmethods.net</email>
            <address>
               <addressLine />
               <addressLine />
               <addressLine />
               <addressLine />
               <addressLine />
            </address>
         </contact>
      </contacts>
      <businessServices>
         <businessService
            serviceKey="d5b180a0-4342-11d5-bd6c-002035229c64"
            businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
         <name>XMethods Barnes and Noble Quote</name>
         <description xml:lang="en">
            Returns book price from Barnes and Noble online store, given
               ISBN
         </description>
         <bindingTemplates>
            <bindingTemplate
               serviceKey="d5b180a0-4342-11d5-bd6c-002035229c64"
               bindingKey="d5b61480-4342-11d5-bd6c-002035229c64">
               <description xml:lang="en">
                  SOAP Binding for tmodel:XMethods Book Quote
               </description>
               <accessPoint URLType="http">
                  http://services.xmethods.net:80/soap/servlet/rpcrouter
               </accessPoint>
               <tModelInstanceDetails>
                  <tModelInstanceInfo
                      tModelKey="uuid:26d3abd0-433d-11d5-bd6c-
                            002035229c64" />
```

```
            </tModelInstanceDetails>
         </bindingTemplate>
      </bindingTemplates>
   </businessService>
   <businessService serviceKey="ed85f000-4345-11d5-bd6c-002035229c64"
      businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
      <name>XMethods Pacific Bell SMS Service</name>
      <description xml:lang="en">
         Sends a text message to a subscriber on the PacBell SMS
            network
      </description>
      <bindingTemplates>
         <bindingTemplate
            serviceKey="ed85f000-4345-11d5-bd6c-002035229c64"
            bindingKey="ed8d1bf0-4345-11d5-bd6c-002035229c64">
            <description xml:lang="en">
               SOAP Binding for tmodel:XMethods SMS
            </description>
            <accessPoint URLType="http">
               http://services.xmethods.net:80/perl/soaplite.cgi
            </accessPoint>
            <tModelInstanceDetails>
               <tModelInstanceInfo
                  tModelKey="uuid:33f24880-433d-11d5-bd6c-
                     002035229c64" />
            </tModelInstanceDetails>
         </bindingTemplate>
      </bindingTemplates>
   </businessService>
   <businessService
         serviceKey="d5921160-3e16-11d5-98bf-002035229c64"
         businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
         <name>XMethods Delayed Stock Quotes</name>
         <description xml:lang="en">
            20-minute delayed stock quotes
         </description>
         <bindingTemplates>
            <bindingTemplate
               serviceKey="d5921160-3e16-11d5-98bf-002035229c64"
               bindingKey="d594a970-3e16-11d5-98bf-002035229c64">
               <description xml:lang="en">
                  SOAP binding for delayed stock quotes service
               </description>
               <accessPoint URLType="http">
                  http://services.xmethods.net:80/soap
               </accessPoint>
               <tModelInstanceDetails>
                  <tModelInstanceInfo
```

```
                        tModelKey="uuid:0e727db0-3e14-11d5-98bf-
                           002035229c64" />
                     </tModelInstanceDetails>
                  </bindingTemplate>
               </bindingTemplates>
            </businessService>
            <businessService
               serviceKey="618167a0-3e64-11d5-98bf-002035229c64"
               businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
               <name>XMethods Currency Exchange Rates</name>
               <description xml:lang="en">
                  Returns exchange rates between 2 countries' currencies
               </description>
               <bindingTemplates>
                  <bindingTemplate
                     serviceKey="618167a0-3e64-11d5-98bf-002035229c64"
                     bindingKey="618474e0-3e64-11d5-98bf-002035229c64">
                     <description xml:lang="en">
                        SOAP binding for currency exchange rates service
                     </description>
                     <accessPoint URLType="http">
                        http://services.xmethods.net:80/soap
                     </accessPoint>
                     <tModelInstanceDetails>
                        <tModelInstanceInfo
                           tModelKey="uuid:e092f730-3e63-11d5-98bf-
                              002035229c64" />
                     </tModelInstanceDetails>
                  </bindingTemplate>
               </bindingTemplates>
            </businessService>
         </businessServices>
      </businessEntity>
   </businessDetail>
```

错误：

`E_invalidKeyPassed`

指定的 `businessKey` 无效。

get_businessDetailExt

*get_businessDetailExt*函数获取每个指定`businessKey`的扩展`businessEntity`。响应包含一个根 `businessDetailExt` 元素，以及每个相匹配业务的一个

businessEntityExt 元素。如果操作入口只返回部分匹配结果，则 businessDetailExt 元素的 truncated 属性将被设为 true。如果没有找到相匹配的结果，则将返回一个 E_invalidKeyPassed 错误。查询不属于 UDDI 服务群并可能含有额外的业务注册信息的外部 UDDI 注册中心时，这个函数很有用。当查询 UDDI 操作入口站点时，这个方法返回的结果与 *get_businessDetail* 函数完全相同。

1.0 和 2.0 版本：

1.0 语法:

```
<get_businessDetailExt generic="1.0" xmlns="urn:uddi-org:api" >
      <businessKey/>
      [<businessKey/> ...]
</get_businessDetailExt>
```

2.0 语法:

```
<get_businessDetailExt generic="2.0" xmlns="urn:uddi-org:api_v2" >
   <businessKey/>
   [<businessKey/> ...]
</get_businessDetailExt>
```

参数:

businessKey

必需的 uuid_key，指定 businessEntity。你可以指定多个 businessKey。

例子：

下面的 UDDI 1.0 例子获取 XMethods 公司的外部 businessEntity 记录：

```
<get_businessDetailExt generic="1.0" xmlns="urn:uddi-org:api">
   <businessKey>ba744ed0-3aaf-11d5-80dc-002035229c64</businessKey>
</get_businessDetailExt>
```

当查询UDDI操作入口站点时，将返回与本章前面给出的*get_businessDetail*例子完全一样的结果。

错误:

E_invalidKeyPassed

指定的 businessKey 无效。

E_unsupported

查询不被支持。如果发生这种情况，则使用 get_businessDetail 查询。

get_serviceDetail

get_serviceDetail 函数获取每个指定 serviceKey 的 businessService 记录。响应包含一个根 serviceDetail 元素，以及每个相匹配服务的一个 businessService 元素。如果操作入口只返回部分匹配结果，则 serviceDetail 元素的 truncated 属性将被设为 true。如果没有找到相匹配的结果，则将返回一个 E_invalidKeyPassed 错误。

1.0 和 2.0 版本：

1.0 语法:

```
<get_serviceDetail generic="1.0" xmlns="urn:uddi-org:api" >
   <serviceKey/>
   [<serviceKey/> ...]
</get_serviceDetail>
```

2.0 语法:

```
<get_serviceDetail generic="2.0" xmlns="urn:uddi-org:api_v2">
   <serviceKey/>
   [<serviceKey/> ...]
</get_serviceDetail>
```

参数:

serviceKey

必需的 uuid_key，指定 serviceDetail。你可以指定多个 serviceKey。

例子：

下面的 UDDI 1.0 例子获取 Xmethods Stock Quote 服务的细节：

```
<get_serviceDetail generic="1.0" xmlns="urn:uddi-org:api">
   <serviceKey>d5921160-3e16-11d5-98bf-002035229c64</serviceKey>
</get_serviceDetail>
```

下面是对查询的完整响应：

```
<serviceDetail generic="1.0" operator="Microsoft Corporation"
   truncated="false" xmlns="urn:uddi-org:api">
   <businessService
      serviceKey="d5921160-3e16-11d5-98bf-002035229c64"
      businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
      <name>XMethods Delayed Stock Quotes</name>
      <description xml:lang="en">
         20-minute delayed stock quotes
      </description>
      <bindingTemplates>
         <bindingTemplate
            serviceKey="d5921160-3e16-11d5-98bf-002035229c64"
            bindingKey="d594a970-3e16-11d5-98bf-002035229c64">
            <description xml:lang="en">
               SOAP binding for delayed stock quotes service
            </description>
            <accessPoint URLType="http">
               http://services.xmethods.net:80/soap
            </accessPoint>
            <tModelInstanceDetails>
               <tModelInstanceInfo
                  tModelKey="uuid:0e727db0-3e14-11d5-98bf-002035229c64" />
            </tModelInstanceDetails>
         </bindingTemplate>
      </bindingTemplates>
   </businessService>
</serviceDetail>
```

错误：

`E_invalidKeyPassed`

指定的 `serviceKey` 无效。

get_tModelDetail

get_tModelDetail 函数获取每个指定 tModelKey 的 tModel 记录。响应包含一个根 tModelDetail 元素，以及每个相匹配 tModel 的一个 tModel 元素。如果 UDDI 操作入口只返回部分相匹配的结果，则 tModelDetail 元素的 truncated 属性将被设为 true。如果没有找到相匹配的结果，则将返回一个 E_invalidKeyPassed 错误。

1.0 和 2.0 版本：

1.0 语法:

```
<get_tModelDetail generic="1.0" xmlns="urn:uddi-org:api">
   <tModelKey/>
   [<tModelKey/> ...]
</get_tModelDetail>
```

2.0 语法:

```
<get_tModelDetail generic="2.0" xmlns="urn:uddi-org:api_v2">
   <tModelKey/>
   [<tModelKey/> ...]
</get_tModelDetail>
```

参数:

tModelKey

必需的 uuid_key，指定 tModel。你可以指定多个 tModelKey。

例子：

下面的 UDDI 1.0 例子获取“ Query Shipment Status ”的 RosettaNet PIP（Partner Interface Process，合作伙伴接口处理）的 tModel 记录：

```
<get_tModelDetail generic="1.0" xmlns="urn:uddi-org:api">
   <tModelKey>uuid:87c133f5-f521-420a-8316-9ff37e292796</tModelKey>
</get_tModelDetail>
```

下面是对这个查询的完整响应：

```
<tModelDetail generic="1.0" operator="Microsoft Corporation"
   truncated="false" xmlns="urn:uddi-org:api">
   <tModel tModelKey="uuid:87c133f5-f521-420a-8316-9ff37e292796"
      operator="Microsoft Corporation" authorizedName="Suhayl Masud">
      <name>Rosettanet-org:PIP3B4:QueryShipmentStatus:v01.00.00</name>
      <description xml:lang="en">
         Enables in-transit information users to query shipment status,
         and allows transport service providers to respond with shipment
         status notifications
      </description>
      <overviewDoc>
         <description xml:lang="en">
            This is the compressed file that contains the specification
            in a word document, the html message guideline document and
            the xml dtds
         </description>
         <overviewURL>
            http://www.rosettanet.org/rosettanet/Doc/0/
            CG0J24SOSBA13FCD0282FOPTD8/3B4_QueryShipment
            Status_R01_00_00.zip
         </overviewURL>
      </overviewDoc>
   </tModel>
</tModelDetail>
```

错误：

`E_invalidKeyPassed`

指定的 `tModelKey` 无效。

`E_keyRetired`

指定的 `tModelKey` 不再处于活动状态。

查找限定符

每个查找函数都可选择一组 UDDI 查找限定符，从而能给出更精确的搜索条件。表 8-2 提供了一组最常用的 UDDI 查找限定符。

表 8-2：最常用的 UDDI 查找限定符

查找限定符	描述
exactNameMatch	要求严格的名称匹配
caseSensitiveMatch	要求搜索区分大小写
sortByNameAsc	按名称字母升序排列结果
sortByNameDesc	按名称字母降序排列结果
sortByDateAsc	按最后更新时间的升序排列结果
sortByDateDesc	按最后更新时间的降序排列结果
soundex	要求按发音搜索指定名称（只针对 UDDI 2.0）

下面的例子说明了 `exactNameMatch` 限定符的用法：

```
<find_business generic="1.0" xmlns="urn:uddi-org:api">
   <findQualifiers>
      <findQualifier>exactNameMatch</findQualifier>
   </findQualifiers>
   <name>Ariba, Inc</name>
</find_business>
```

UDDI 的搜索工具非常齐全，并且在以后的版本中会得到进一步提高。

第九章

UDDI 4J

UDDI Java 版 （UDDI4J） 是一个获取和发布 UDDI 数据的 Java 客户端工具包，最初由 IBM 创建。2001 年 1 月，IBM 将此代码转到它自己的 developerWorks 开源站点（*http://www-124.ibm.com/developerworks/oss/*）。

本章将全面介绍 UDDI4J，具体包括以下几个方面：

- 使用 UDDI4J API 的各个技术方面；
- 三个应用程序样例，其中两个是使用 UDDI 查询 API 的应用程序，一个是使用 UDDI 发布 API 的应用程序；
- 完整的 UDDI4J API 快速参考。

准备工作

在开始之前，你首先必须下载 UDDI4J 发布版。你可以从 *http://oss.software.ibm.com /developerworks/projects/uddi4j/* 获得这个发布版。

你还需要以下软件：

- 完整的 Apache SOAP 发布版，包括 Xerces XML 解析器和 JavaMail API 等必备软件。完整的细节请参阅第三章。
- JSSE（Java Secure Socket Extension，Java 安全套接字扩展）。如果你计划发布 UDDI 数据，JSSE 就是必不可少的，你可以从 *http://java.sun.com/products/jsse/index.html* 获得它。

技术回顾

UDDI4J API 提供了 UDDI 数据模型和 UDDI 编程 API 的一个直接映射（具体细节请参阅第七章）。最重要的类是 `UDDIProxy` 类，这个类提供了集中访问包含在 UDDI 1.0 编程 API 中的所有查询和发布函数。要运行查询函数，你必须通过 `setInquiryURL()` 方法为 UDDI 操作入口站点提供一个查询 URL。要运行发布函数，你必须通过 `setPublishURL()` 方法指定一个 HTTPS URL。

每个 UDDI 编程 API 函数都可以通过 `UDDIProxy` 类直接获得。例如，`UDDIProxy` 类有 `find_business()`、`get_businessDetail()`、`get_TModelDetail()` 等函数的查询方法，它还有 `save_business()`、`save_service()`、`delete_business()` 等函数的发布方法。

根据方法调用，`UDDIProxy` 对象将产生一个 SOAP 请求，并向指定的 UDDI 操作入口站点发送请求，使响应数据可以通过中间对象获得。这些中间对象大部分都可以从 `com.ibm.uddi.datatype` 包中获得。如 `com.ibm.uddi.datatype.business` 包就包含了提取业务名称、描述和联系信息等业务实体信息的类。

如果发生错误，`UDDIProxy` 类将抛出一个 `UDDIException` 异常。根据错误类型，`UDDIException` 将会包含一个含有有关错误原因的细节信息的 `DispositionReport` 对象。

发现和获取 UDDI 数据

有了 UDDI，你就可以用一些稍有差别的 API 来查找有关业务和业务实体的信息。

搜索业务

我们的第一个例子运行UDDIProxy find_business()方法, 并打印出相匹配的结果。程序需要一个命令行参数，你可以用这个参数指定业务名称。如下面的命令行：

```
java com.ecerami.uddi.findBusiness Micro
```

产生如下输出：

```
Searching for Businesses: Micro
Microtrack, Inc.: f53480ab-be29-4090-9239-f4c4a7cf71c6
Microform Reading Room: 622a4879-dcaa-4bab-9aec-6e6bfb858067
Micro Informatica LLC: dce959cf-200d-4d9e-beee-ede770299212
Microsoft Corporation: 0076b468-eb27-42e5-ac09-9955cff462a3
Micromotor: 11bb5410-61d7-11d5-b286-002035229c64
MicroVideo Learning Systems: 8995b9f7-0043-4eb0-adaf-2aa81ad387e4
MicroApplications, Inc.: a23c901e-834c-4b8a-bf38-3f96fedc349a
MicroLink LLC: fb5783d6-4ba4-4bce-b181-4d3cd9f35e3d
MicroMain Corporation: 1e6c5410-00e7-4aee-acfb-fb59f3896322
Micro Motion Inc.: d4e4b830-f19e-4edf-9f44-8936e53d9a33
```

例 9-1 列出了完整的代码。首先要注意 findBusinessByName()方法创建了一个新的 UDDIProxy 对象，指定了微软 UDDI 站点的查询 URL（要得到查询和发布 URL 的完整列表，请参阅第七章）。

接着，代码调用 UDDI Proxy 的 find_business()方法：

```
BusinessList businessList = proxy.find_business(businessName, null, 0);
```

这个方法需要三个参数：业务名称、FindQualifier 对象和返回的最大记录数。在我们的例子中，因为没有查找限定符，所以传送一个空值（要想获得有关查找限定符的使用信息，请参阅本章后面的“ FindQualifiers ”一节）。我们还为最大记录数指定了一个 0 值。0 是一个保留值，表示对返回的行数无限制。

find_business()方法返回一个 BusinessList 对象。接着，代码搜索 BusinessList 对象以获得相匹配的业务。要获得有关搜索业务数据层的完整细节，请参考快速参考 API 中的 com.ibm.uddi.datatype.business 包。

当发生 UDDIException 异常时，代码就试图提取 DispositionReport 对象并显示错误发生的原因。

例 9-1：findBusiness.java

```
package com.ecerami.uddi;

import java.util.*;
import com.ibm.uddi.UDDIException;
import com.ibm.uddi.client.UDDIProxy;
import com.ibm.uddi.response.*;
import java.net.MalformedURLException;
import org.apache.soap.SOAPException;

/**
* UDDI 样例程序：搜索所有匹配
* 第一个命令行参数的公司
* 例子用法: java findBusiness XMethods
*/
public class findBusiness {

  /**
   * main 方法
  */
  public static void main (String args[]) {
    findBusiness inquiry = new findBusiness();

    try {
      // 搜索指定的业务名称
      String businessName = args[0];
      System.out.println ("Searching for Businesses: "+businessName);
      Vector businessInfoVector = inquiry.findBusinessByName (businessName);

      // 打印每个匹配业务的名称和业务键
      for (int i=0; i<businessInfoVector.size(); i++) {
          BusinessInfo businessInfo =
            (BusinessInfo) businessInfoVector.elementAt(i);
          String name = businessInfo.getNameString();
          String businessKey = businessInfo.getBusinessKey();
          System.out.println (name+": "+businessKey);
      }
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (SOAPException e) {
      e.printStackTrace();
```

```
    } catch (UDDIException e) {
      // 提取UDDI处理报告
      DispositionReport dr = e.getDispositionReport();
      if (dr!=null) {
        System.out.println("UDDIException faultCode:" +
          e.getFaultCode() +
          "\n errno:" + dr.getErrno() +
          "\n errCode:" + dr.getErrCode() +
          "\n errInfoText:" + dr.getErrInfoText());
      }
      e.printStackTrace();
    }
  }

  /**
   * 根据名称查找业务
   * @param businessName Business Name Target
   * @return Vector of BusinessInfo objects
  */
  public Vector findBusinessByName (String businessName)
    throws MalformedURLException, SOAPException, UDDIException {
    // 创建UDDI Proxy对象
    UDDIProxy proxy = new UDDIProxy();

    // 指向微软查询URL
    proxy.setInquiryURL("http://uddi.microsoft.com/inquire");

    // 查找匹配的业务
    BusinessList businessList = proxy.find_business(businessName, null, 0);

    // 处理UDDI响应
    BusinessInfos businessInfos = businessList.getBusinessInfos();
    Vector businessInfoVector = businessInfos.getBusinessInfoVector();
    return businessInfoVector;
    }
}
```

FindQualifiers

你可以使用 `FindQualifiers`对象来更精确地控制搜索条件。如下面的代码搜索严格匹配的名称：

```
    Vector fqs = new Vector();
    FindQualifiers findQualifiers = new FindQualifiers ();
```

```
FindQualifier fq = new FindQualifier(FindQualifier.exactNameMatch);
fqs.addElement(fq);
findQualifiers.setFindQualifierVector(fqs);
BusinessList bl = proxy.find_business("Microsoft Corporation",
    findQualifiers, 0);
```

要获得更为完整的细节，请参阅快速参考 API 中的 `com.ibm.uddi.util` 包。

IdentifierBag

你可以使用 `IdentifierBag`对象来执行由一个或多个UDDI标识符索引的搜索。如下面的代码试图找到与指定 Dun & Bradstreet D-U-N-S® Number 相关的业务记录：

```
Vector keyedReferenceVector = new Vector();
KeyedReference keyedRef = new KeyedReference
   ("dnb-com:D-U-N-S", "04-693-3052");
keyedRef.setTModelKey ("uuid:8609c81e-ee1f-4d5a-b202-3eb13ad01823");
keyedReferenceVector.addElement (keyedRef);
IdentifierBag idBag = new IdentifierBag ();
idBag.setKeyedReferenceVector(keyedReferenceVector);
BusinessList bl = proxy.find_business(idBag, null, 0);
```

有关 UDDI 标识符的细节请参阅第七章。

CategoryBag

你可以使用 `CategoryBag` 对象来执行由一个或多个 UDDI 分类索引的搜索。如下面的代码试图搜索用软件发布者的 NAICS 代码注册的所有业务:

```
keyedReferenceVector = new Vector();
keyedRef = new KeyedReference ("ntis-gov:naics:1997", "51121");
keyedRef.setTModelKey ("uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2");
keyedReferenceVector.addElement (keyedRef);
CategoryBag categoryBag = new CategoryBag ();
categoryBag.setKeyedReferenceVector(keyedReferenceVector);
BusinessList bl = proxy.find_business(categoryBag, null, 0);
```

有关预注册 UDDI 分类法和分类的细节，请参阅第七章。

获取 businessEntity 记录

我们的第二个例子运行 UDDI Proxy 的 get_businessDetail()方法，并打印出完整 businessEntity 记录的某些部分。这个程序需要一个命令行参数，你可以用这个参数为业务指定 businessKey。如下面的命令行：

```
java com.ecerami.uddi.getBusinessDetail 0076b468-eb27-42e5-ac09-9955cff462a3
```

产生如下结果：

```
Business Name: Microsoft Corporation
Description: Empowering people through great software - any time, any place
and on any device is Microsoft's vision. As the worldwide leader in software
for personal and business computing, we strive to produce innovative
products and services that meet our customer's
Contact: Corporate Mailing Addresses
Address: Microsoft Corporation
Address: One Microsoft Way
Address: Redmond, WA 98052-6399
Address: USA
Contact: World Wide Operations
Email: martink@microsoft.com
```

例 9-2 显示了完整的代码。注意， get_businessDetail()方法返回一个 BusinessDetail 对象。你可以漫游 BusinessDetail 对象来显示你想要的记录的任何部分。在这个例子中，代码显示业务名称、描述和联系信息。要获得有关漫游 BusinessDetail 对象的具体信息，请参阅快速参考 API 中的 com.ibm.uddi.datatype.business 包（曾在第八章讨论过）。

例 9-2：getBusinessDetail.java

```
package com.ecerami.uddi;

/**
 * UDDI 样例程序：获取由第一个命令行参数
 * 指定的 businessEntity
 * 例子用法
 * java getBusinessDetail ba744ed0-3aaf-11d5-80dc-002035229c64
 */
import java.util.*;
import com.ibm.uddi.client.UDDIProxy;
import com.ibm.uddi.UDDIException;
```

```
import com.ibm.uddi.response.DispositionReport;
import com.ibm.uddi.response.BusinessDetail;
import com.ibm.uddi.datatype.business.*;
import java.net.MalformedURLException;
import org.apache.soap.SOAPException;

public class getBusinessDetail {

  /**
  * main方法
  */
  public static void main (String args[]) {
    try {
      getBusinessDetail inquiry = new getBusinessDetail();
      BusinessDetail businessDetail = inquiry.getBusinessDetail (args[0]);
      inquiry.print_businessDetail (businessDetail);
    } catch (MalformedURLException e) {
      e.printStackTrace();
    } catch (SOAPException e) {
      e.printStackTrace();
    } catch (UDDIException e) {
      // 提取UDDI处理报告
      DispositionReport dr = e.getDispositionReport();
      if (dr!=null) {
        System.out.println("UDDIException faultCode:" +
          e.getFaultCode() +
          "\n errno:" + dr.getErrno() +
          "\n errCode:" + dr.getErrCode() +
          "\n errInfoText:" + dr.getErrInfoText());
      }
      e.printStackTrace();
    }
  }

  /**
   * 获取业务细节记录
   * @param businessKey UDDI Business Key
   * @return UDDI Business Detail record
   */
  public BusinessDetail getBusinessDetail (String businessKey)
    throws MalformedURLException, SOAPException, UDDIException {
    // 创建UDDIProxy对象
    UDDIProxy proxy = new UDDIProxy();
    // 指向微软查询URL
    proxy.setInquiryURL("http://uddi.microsoft.com/inquire");

    // 获取BusinessDetail记录
    BusinessDetail businessDetail = proxy.get_businessDetail(businessKey);
```

```
  return businessDetail;
}

/**
 * 打印业务实体数据
 * @param businessDetail UDDI Business Detail Record
 */
private void print_businessDetail (BusinessDetail businessDetail) {
  Vector businessEntityVector = businessDetail.getBusinessEntityVector();
  for (int i = 0; i < businessEntityVector.size(); i++) {
      BusinessEntity businessEntity =
        (BusinessEntity) businessEntityVector.elementAt(i);
      String name = businessEntity.getNameString();
      String description = businessEntity.getDefaultDescriptionString();
      System.out.println ("Business Name: "+name);
      System.out.println ("Description: "+description);
      Contacts contacts = businessEntity.getContacts();
      print_contacts (contacts);
    }
}

/**
 * 打印联系信息数据
 * @param contacts UDDI Contacts Information
 */
private void print_contacts (Contacts contacts) {
  Vector contactVector = contacts.getContactVector();
  for (int j=0; j< contactVector.size(); j++) {
    Contact contact = (Contact) contactVector.elementAt (j);
    String description = contact.getDefaultDescriptionString();
    Vector addressVector = contact.getAddressVector();
    Vector emailVector = contact.getEmailVector();
    System.out.println ("Contact: "+ description);
    print_addressVector (addressVector);
    print_emailVector (emailVector);
  }
}

/**
 * 打印地址数据
 * @param addressVector Vector of UDDI Address Records
 */
private void print_addressVector (Vector addressVector) {
  for (int i=0; i< addressVector.size(); i++) {
    Address address = (Address) addressVector.elementAt(i);
    Vector addressLines = address.getAddressLineVector();
    for (int j=0; j<addressLines.size(); j++) {
      AddressLine addressLine = (AddressLine) addressLines.elementAt(j);
```

```
        String addressText = addressLine.getText();
        System.out.println("Address: "+addressText);
      }
    }
  }

  /**
   * 打印电子邮件数据
   * @param emailVector Vector of UDDI Email Objects
   */
  private void print_emailVector (Vector emailVector) {
    for (int i=0; i< emailVector.size(); i++) {
      Email email = (Email) emailVector.elementAt(i);
      String emailText = email.getText();
      System.out.println ("Email: "+emailText);
    }
  }
}
```

发布 UDDI 数据

UDDI发布请求只能通过SSL连接发送，并且只限于授权用户。要指定一个HTTPS URL，请使用`setPublishURL()`方法。要获得用户授权，你必须使用UDDI Proxy的`get_authToken()`方法。这个方法要求一个用户名和密码，返回一个身份验证标记，这个标记必须被传送到接下来的所有发布方法。

我们的最后一个例子说明了如何保存一个新的`businessEntity`记录，它还说明了获得用户身份验证标记的一般技术。代码为 Acme Parts 创建一个新的`businessEntity`记录，然后确保通过提取和显示这个新的`businessKey`来发布记录。

这个程序需要两个命令行参数，在这些参数中，你可以指定一个用户名和密码。例如以下命令行：

```
java com.ecerami.uddi.saveBusiness ethan@ecerami.com oreilly
```

产生如下输出：

```
Saving New Business: Acme Parts
Authentication Token:
1IbuzUmG2DuPzkcoUd1I6iy*BqWwxPhuFWf2!ggowk*6Kiznlu4sjQeJT
```

```
OnuL2c1Smm8m28aogU!I6ZL73yROf3Q$$;1IbuzUmG2AoqvPmRavvTeO1UsaVLQk0AmGYVyyxwk3
   2Kb4
c3jMROMo4Rlp7kaQYRCXu9!95wQ8FZTIjE!47uKC9NPCCryf0Yh!IltwfYMCaPKTq*R0rTWMrzcg
   UWEt
!g859iKxlE*w7QPIx8n8aYWs8WCkAn7UHwyb
Published Business Key: 3deafa60-0023-4220-b3e0-0c87cb34526a
```

例9-3包括一个用于验证用户身份的UDDIUtil类。注意，这个类指定了一个SSL提供者：

```
System.setProperty("java.protocol.handler.pkgs",
   "com.sun.net.ssl.internal.www.protocol");
Security.addProvider (new com.sun.net.ssl.internal.ssl.Provider());
```

这两行对于通过SSL传输信息是必需的。

请注意，UDDI Proxy的get_authToken()方法需要一个用户名和密码，并返回一个AuthToken对象。于是，AuthToken对象可以用于接下来的所有发布请求。

例9-3提供了代码的其余部分。在代码中，我们保存了一个带新名称、描述和联系信息的新BusinessEntity对象。特别要注意BusinessEntity对象的实例化：

```
BusinessEntity businessEntity = new BusinessEntity("", "Acme Parts");
```

代码向构造函数传送一个空的businessKey字符串，表明这是一个新记录。UDDI Proxy的save_business()方法回应一个BusinessDetail对象，使我们可以提取新分配的businessKey值。

例9-3：saveBusiness.java

```
package com.ecerami.uddi;

/**
 * UDDI程序：发布一个新的UDDI BusinessEntity记录
 * 在命令行中指定用户名和密码
 * 例子用法：
 * java saveBusiness ethan@ecerami.com password
 */
import com.ibm.uddi.*;
import com.ibm.uddi.client.*;
import com.ibm.uddi.datatype.business.*;
import com.ibm.uddi.response.*;
```

```
import java.util.Vector;
import java.net.MalformedURLException;
import org.apache.soap.SOAPException;

public class saveBusiness {
  private AuthToken token;
  private UDDIProxy proxy;

  /**
  * main方法
  */
  public static void main (String args[]) {
    saveBusiness publish = null;
    try {
        System.out.println("Saving New Business: Acme Parts");
        AuthToken token = UDDIUtil.get_authentication_token(args[0], args[1]);
        System.out.println("Authentication Token: "+token.getAuthInfoString());
        publish = new saveBusiness(token);
        String businessKey = publish.save_business();
        System.out.println("Published Business Key: "+businessKey);
    } catch (MalformedURLException e) {
        e.printStackTrace();
    } catch (SOAPException e) {
        e.printStackTrace();
    } catch (UDDIException e) {
        DispositionReport dr = e.getDispositionReport();
        UDDIUtil.printDispositionReport (dr);
        e.printStackTrace();
    }
  }

  /**
   * 构造函数
   * @param token UDDI Authentication Token
   */
  public saveBusiness (AuthToken token) {
    this.token = token;
  }

  /**
   * 保存新的业务实体
  */
  public String save_business ()
    throws MalformedURLException, SOAPException, UDDIException {
    String businessKey = null;
```

```
    // 指向微软测试发布 URL(SSL)
    proxy = new UDDIProxy();
    proxy.setPublishURL("https://test.uddi.microsoft.com/publish");

    // 创建业务实体记录样例
    BusinessEntity businessEntity = create_business ();
    Vector businessEntityVector = new Vector();
    businessEntityVector.addElement(businessEntity);

    // 发布新的业务记录
    BusinessDetail businessDetail =
      proxy.save_business(token.getAuthInfoString(), businessEntityVector);

    // 通过提取新的业务键来验证发布
    Vector businessEntities = businessDetail.getBusinessEntityVector();
    if (businessEntities.size() > 0) {
      BusinessEntity returnedBusinessEntity =
        (BusinessEntity)(businessEntities.elementAt(0));
      businessKey = returnedBusinessEntity.getBusinessKey();
    }
    return businessKey;
  }

  /**
   * 创建新的业务实体记录样例
  */
  private BusinessEntity create_business() {
    Vector businessEntities = new Vector();
    BusinessEntity businessEntity = new BusinessEntity("", "Acme Parts");

    // 设置业务描述
    businessEntity.setDefaultDescriptionString
      ("Maker of fine semiconductor parts");

    // 设置联系名字和电子邮件
    Contact contact = new Contact ("Ethan Cerami");
    Email email = new Email("cerami@cs.nyu.edu");
    Vector emailVector = new Vector();
    emailVector.addElement(email);
    contact.setEmailVector(emailVector);
    Contacts contacts = new Contacts();
    Vector contactVector = new Vector();
    contactVector.addElement(contact);
    contacts.setContactVector(contactVector);
    businessEntity.setContacts(contacts);
    return businessEntity;
  }
}
```

UDDI4J 快速参考 API

本节为UDDI4J API的快速参考。除了com.ibm.uddi.request以外，API中的其他所有包都涉及到了。com.ibm.uddi.request这个客户端应用程序一般不直接使用。

com.ibm.uddi 包

com.ibm.uddi.UDDIElement

这是所有 UDDI 元素的基类。

概要

```
public abstract class UDDIElement extends Object {
  // 构造函数
     public UDDIElement();
     public UDDIElement(Element el) throws UDDIException;
  // 字段摘要
     protected Element base;
     public static String GENERIC;
     public static String XMLNS;
  // 公共方法
     public NodeList getChildElementsByTagName(Element el, String tag);
     abstract public void saveToXML(Element base);
  // 保护方法
     protected String getText(Node el);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement
```

com.ibm.uddi.UDDIException

这个类封装与UDDI相关的错误。UDDIException对象可能包括一个包含有关错误原因的具体信息的 DispositionReport 对象。

概要

```
public class UDDIException extends Exception {
  // 构造函数
     public UDDIException();
     public UDDIException(Element el, boolean createDispositionReport);
  // 公共方法
     public String getDetail();
     public Element getDetailElement();
     public DispositionReport getDispositionReport();
     public String getFaultActor();
     public String getFaultCode();
     public String getFaultString();
     public boolean isValidElement(Element el);
     public String toString();
  // 保护方法
     protected String getText(Node el);
}
```

层次

```
java.lang.Object → java.lang.Throwable → java.lang.Exception → com.ibm.
uddi.UDDIException
```

com.ibm.uddi.VectorNodeList

这个实用类提供了`org.w3c.dom.NodeList`接口的一个实现, 但它一般不直接用于 UDDI 客户端代码。

概要

```
public class VectorNodeList extends Object implements NodeList {
  // 构造函数
     public VectorNodeList(Vector v);
  // 公共方法
     public int getLength();
     public Vector getVector();
     public Node item(int index);
}
```

层次

```
java.lang.Object → com.ibm.uddi.VectorNodeList
```

com.ibm.uddi.client 包

com.ibm.uddi.client.UDDIProxy

这个类提供对所有由UDDI 1.0编程API提供的查询和发布函数的集中式访问。要运行查询函数，你必须通过setInquiryURL()方法设定查询URL；要运行发布函数，你必须通过setPublishURL()方法设定一个HTTPS URL。所有的发布请求，如save_xxx()和delete_xxx()，都需要一个身份验证标记。要获得这个标记，请使用get_authToken()方法，并指定一个用户名和密码。

概要

```
public class UDDIProxy extends Object {
  // 构造函数
    public UDDIProxy();
    public UDDIProxy(URL inquiryURL, URL publishURL,
    SOAPTransport transport);
  // 公共方法
    public DispositionReport delete_binding(String authInfo,
      String bindingKey) throws UDDIException, SOAPException;
    public DispositionReport delete_binding(String authInfo,
      Vector bindingKeyStrings) throws UDDIException, SOAPException;
    public DispositionReport delete_business(String authInfo,
      String businessKey) throws UDDIException, SOAPException;
    public DispositionReport delete_business(String authInfo,
      Vector businessKeyStrings) throws UDDIException, SOAPException;
    public DispositionReport delete_service(String authInfo,
      String serviceKey) throws UDDIException, SOAPException;
    public DispositionReport delete_service(String authInfo,
      Vector serviceKeyStrings) throws UDDIException, SOAPException;
    public DispositionReport delete_tModel(String authInfo,
      String tModelKey) throws UDDIException, SOAPException;
    public DispositionReport delete_tModel(String authInfo,
      Vector tModelKeyStrings) throws UDDIException, SOAPException;
    public DispositionReport discard_authToken(AuthInfo authInfo)
      throws UDDIException, SOAPException;
    public BindingDetail find_binding(FindQualifiers findQualifiers,
      String serviceKey, TModelBag tmodelbag, int maxRows)
      throws UDDIException, SOAPException;
    public BusinessList find_business(CategoryBag bag, FindQualifiers
      findQualifiers, int maxRows) throws UDDIException, SOAPException;
    public BusinessList find_business(DiscoveryURLs bag, FindQualifiers
      findQualifiers, int maxRows) throws UDDIException, SOAPException;
```

```
public BusinessList find_business(IdentifierBag bag, FindQualifiers
  findQualifiers, int maxRows) throws UDDIException, SOAPException;
public BusinessList find_business(String name, FindQualifiers
  findQualifiers, int maxRows) throws UDDIException, SOAPException;
public BusinessList find_business(TModelBag bag, FindQualifiers
  findQualifiers, int maxRows) throws UDDIException, SOAPException;
public ServiceList find_service(String businessKey, CategoryBag bag,
  FindQualifiers findQualifiers, int maxRows) throws UDDIException,
  SOAPException;
public ServiceList find_service(String businessKey, String name,
  FindQualifiers findQualifiers, int maxRows) throws UDDIException,
  SOAPException;
public ServiceList find_service(String businessKey, TModelBag bag,
  FindQualifiers findQualifiers, int maxRows) throws UDDIException,
  SOAPException;
public TModelList find_tModel(CategoryBag bag, FindQualifiers
  findQualifiers, int maxRows) throws UDDIException, SOAPException;
public TModelList find_tModel(IdentifierBag identifierBag,
  FindQualifiers findQualifiers, int maxRows) throws UDDIException,
  SOAPException;
public TModelList find_tModel(String name, FindQualifiers
  findQualifiers,int maxRows) throws UDDIException, SOAPException;
public AuthToken get_authToken(String userid, String cred)
  throws UDDIException, SOAPException;
public BindingDetail get_bindingDetail(String bindingKey)
  throws UDDIException, SOAPException;
public BindingDetail get_bindingDetail(Vector bindingKeyStrings)
  throws UDDIException, SOAPException;
public BusinessDetail get_businessDetail(String businessKey)
  throws UDDIException, SOAPException;
public BusinessDetail get_businessDetail(Vector businessKeyStrings)
  throws UDDIException, SOAPException;
public BusinessDetailExt get_businessDetailExt(String businessKey)
  throws UDDIException, SOAPException;
public BusinessDetailExt get_businessDetailExt(Vector
  businessKeyStrings) throws UDDIException, SOAPException;
public RegisteredInfo get_registeredInfo(String authInfo)
  throws UDDIException, SOAPException;
public ServiceDetail get_serviceDetail(String serviceKey)
  throws UDDIException, SOAPException;
public ServiceDetail get_serviceDetail(Vector serviceKeyStrings)
  throws UDDIException, SOAPException;
public TModelDetail get_tModelDetail(String tModelKey)
  throws UDDIException, SOAPException;
public TModelDetail get_tModelDetail(Vector tModelKeyStrings)
  throws UDDIException, SOAPException;
public BindingDetail save_binding(String authInfo,
  Vector bindingTemplates) throws UDDIException, SOAPException;
```

```
    public BusinessDetail save_business(String authInfo,
      UploadRegister[] uploadRegisters) throws UDDIException,
      SOAPException;
    public BusinessDetail save_business(String authInfo,
      Vector businessEntities) throws UDDIException, SOAPException;
    public ServiceDetail save_service(String authInfo,
      Vector businessServices) throws UDDIException, SOAPException;
    public TModelDetail save_tModel(String authInfo, UploadRegister[]
      uploadRegisters) throws UDDIException, SOAPException;
    public TModelDetail save_tModel(String authInfo, Vector tModels)
      throws UDDIException, SOAPException;
    public Element send(Element el, boolean inquiry)
      throws SOAPException;
    public Element send(UDDIElement el, boolean inquiry)
      throws SOAPException;
    public void setInquiryURL(String url) throws MalformedURLException;
    public void setPublishURL(String url) throws MalformedURLException;
    public void setTransport(SOAPTransport transport);
    public DispositionReport validate_categorization(String tModelKey,
      String keyValueString, BusinessEntity businessEntity)
      throws UDDIException, SOAPException;
    public DispositionReport validate_categorization(String tModelKey,
      String keyValueString, BusinessService businessService)
      throws UDDIException, SOAPException;
    public DispositionReport validate_categorization(String tModelKey,
      String keyValueString, TModel tModel) throws UDDIException,
      SOAPException;
}
```

层次

```
java.lang.Object → com.ibm.uddi.client.UDDIProxy
```

com.ibm.uddi.datatype 包

com.ibm.uddi.datatype.Description

这个类封装了一个与多个UDDI数据类型对象相关的原文描述，其中包括BindingTemplate、BusinessEntity、BusinessService和TModel。

概要

```
public class Description extends UDDIElement {
  // 构造函数
```

```
    public  Description();
    public Description(String value);
    public Description(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getLang();
    public String getText();
    public void saveToXML(Element parent);
    public void setLang(String s);
    public void setText(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.datatype.
Description
```

com.ibm.uddi.datatype.Name

这个类封装了一个与多个UDDI数据类型对象有关的原文名称，其中包括`BusinessEntity`、`BusinessService`和`TModel`。

概要

```
public class Name extends UDDIElement {
  // 构造函数
    public Name();
    public Name(String value);
    public Name(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getText();
    public void saveToXML(Element parent);
    public void setText(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement →
com.ibm.uddi.datatype.Name
```

com.ibm.uddi.datatype.OverviewDoc

这个类封装了一个与多个数据类型对象有关的外部综述文档，包括 `TModel` 和 `InstanceDetails`。

概要

```
public class OverviewDoc extends UDDIElement {
  // 构造函数
    public OverviewDoc();
    public OverviewDoc(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getDefaultDescriptionString();
    public Vector getDescriptionVector();
    public OverviewURL getOverviewURL();
    public String getOverviewURLString();
    public void saveToXML(Element parent);
    public void setDefaultDescriptionString(String s);
    public void setDescriptionVector(Vector s);
    public void setOverviewURL(OverviewURL s);
    public void setOverviewURL(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.datatype.
OverviewDoc
```

com.ibm.uddi.datatype.OverviewURL

这个类封装了外部综述文档的 URL。

概要

```
public class OverviewURL extends UDDIElement {
  // 构造函数
    public OverviewURL();
    public OverviewURL(String value);
    public OverviewURL(Element base) throws UDDIException;
```

```
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getText();
    public void saveToXML(Element parent);
    public void setText(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.datatype.
OverviewURL
```

com.ibm.uddi.datatype.binding 包

com.ibm.uddi.datatype.binding.AccessPoint

这个类封装了一个 UDDI accessPoint 记录的信息。

概要

```
public class AccessPoint extends UDDIElement {
  // 构造函数
    public AccessPoint();
    public AccessPoint(String value, String URLType);
    public AccessPoint(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getText();
    public String getURLType();
    public void saveToXML(Element parent);
    public void setText(String s);
    public void setURLType(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.datatype.
binding.AccessPoint
```

com.ibm.uddi.datatype.binding.BindingTemplate

这个类封装了一个 bindingTemplate 记录的信息。

概要

```
public class BindingTemplate extends UDDIElement {
  // 构造函数
    public BindingTemplate();
    public BindingTemplate(String bindingKey, TModelInstanceDetails
      tModelInstanceDetails);
    public BindingTemplate(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public AccessPoint getAccessPoint();
    public String getBindingKey();
    public String getDefaultDescriptionString();
    public Vector getDescriptionVector();
    public HostingRedirector getHostingRedirector();
    public String getServiceKey();
    public TModelInstanceDetails getTModelInstanceDetails();
    public void saveToXML(Element parent);
    public void setAccessPoint(AccessPoint s);
    public void setBindingKey(String s);
    public void setDefaultDescriptionString(String s);
    public void setDescriptionVector(Vector s);
    public void setHostingRedirector(HostingRedirector s);
    public void setServiceKey(String s);
    public void setTModelInstanceDetails(TModelInstanceDetails s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.datatype.
binding.BindingTemplate
```

com.ibm.uddi.datatype.binding.BindingTemplates

这个类封装了多个 BindingTemplate 对象。

概要

```
public class BindingTemplates extends UDDIElement {
  // 构造函数
    public BindingTemplates();
    public BindingTemplates(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public Vector getBindingTemplateVector();
    public void saveToXML(Element parent);
    public void setBindingTemplateVector(Vector s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.datatype.
binding.BindingTemplates
```

com.ibm.uddi.datatype.binding.HostingRedirector

这个类封装了一个 UDDI `hostingRedirector` 记录的信息。

概要

```
public class HostingRedirector extends UDDIElement {
  // 构造函数
    public HostingRedirector();
    public HostingRedirector(String bindingKey);
    public HostingRedirector(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getBindingKey();
    public void saveToXML(Element parent);
    public void setBindingKey(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.datatype.
binding.HostingRedirector
```

com.ibm.uddi.datatype.binding.InstanceDetails

这个类封装了有关 bindingTemplate 记录的实例细节的信息。

概要

```
public class InstanceDetails extends UDDIElement {
  // 构造函数
    public  InstanceDetails();
    public InstanceDetails(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public  String  getDefaultDescriptionString();
    public  Vector  getDescriptionVector();
    public  InstanceParms  getInstanceParms();
    public  OverviewDoc getOverviewDoc();
    public void saveToXML(Element parent);
    public  void  setDefaultDescriptionString(String  s);
    public  void  setDescriptionVector(Vector  s);
    public  void  setInstanceParms(InstanceParms s);
    public  void  setOverviewDoc(OverviewDoc s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.datatype.
binding.InstanceDetails
```

com.ibm.uddi.datatype.binding.InstanceParms

这个类封装了有关设定 bindingTemplate 记录文件或实例参数的信息。

概要

```
public class InstanceParms extends UDDIElement {
  // 构造函数
    public  InstanceParms();
    public  InstanceParms(String value);
    public  InstanceParms(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
```

```
    public static final String UDDI_TAG;
  // 公共方法
    public String getText();
    public void saveToXML(Element parent);
    public void setText(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.datatype.
binding.InstanceParms
```

com.ibm.uddi.datatype.binding.TModelInstanceDetails

这个类封装了多个 `TModelInstanceInfo` 对象。

概要

```
public class TModelInstanceDetails extends UDDIElement {
  // 构造函数
    public  TModelInstanceDetails();
    public TModelInstanceDetails(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public Vector getTModelInstanceInfoVector();
    public void saveToXML(Element parent);
    public void setTModelInstanceInfoVector(Vector s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.datatype.
binding.TModelInstanceDetails
```

com.ibm.uddi.datatype.binding.TModelInstanceInfo

这个类封装了有关一个 UDDI `tModel` 记录的实例细节的信息。

概要

```
public class TModelInstanceInfo extends UDDIElement {
  // 构造函数
    public TModelInstanceInfo();
    public TModelInstanceInfo(String tModelKey);
    public TModelInstanceInfo(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getDefaultDescriptionString();
    public Vector getDescriptionVector();
    public InstanceDetails getInstanceDetails();
    public String getTModelKey();
    public void saveToXML(Element parent);
    public void setDefaultDescriptionString(String s);
    public void setDescriptionVector(Vector s);
    public void setInstanceDetails(InstanceDetails s);
    public void setTModelKey(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.datatype.
binding.TModelInstanceInfo
```

com.ibm.uddi.datatype.business 包

com.ibm.uddi.datatype.business.Address

这个类封装了业务地址信息。也可以参考`com.ibm.uddi.datatype.business.AddressLine`类，这个类将在下一节讨论。

概要

```
public class Address extends UDDIElement {
  // 构造函数
    public Address();
    public Address(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
```

```
  // 公共方法
    public Vector getAddressLineStrings();
    public Vector getAddressLineVector();
    public String getSortCode();
    public String getUseType();
    public void saveToXML(Element parent);
    public void setAddressLineStrings(Vector s);
    public void setAddressLineVector(Vector s);
    public void setSortCode(String s);
    public void setUseType(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.datatype.
business.Address
```

com.ibm.uddi.datatype.business.AddressLine

这个类封装了业务地址中的一行。

概要

```
public class AddressLine extends UDDIElement {
  // 构造函数
    public AddressLine();
    public AddressLine(String value);
    public AddressLine(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getText();
    public void saveToXML(Element parent);
    public void setText(String s);
}
```

层次

```
java.lang.Object →com.ibm.uddi.UDDIElement →com.ibm.uddi.datatype.
business.AddressLine
```

com.ibm.uddi.datatype.business.BusinessEntity

这个类封装了UDDI `businessEntity`记录的所有信息，包括业务名称、描述、联系方式、`businessKey`和任何业务标识符或业务分类法类别。

概要

```
public class BusinessEntity extends UDDIElement {
  // 构造函数
    public  BusinessEntity();
    public BusinessEntity(String businessKey, String name);
    public BusinessEntity(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getAuthorizedName();
    public String getBusinessKey();
    public BusinessServices getBusinessServices();
    public CategoryBag getCategoryBag();
    public Contacts getContacts();
    public String getDefaultDescriptionString();
    public Vector getDescriptionVector();
    public DiscoveryURLs getDiscoveryURLs();
    public IdentifierBag getIdentifierBag();
    public Name getName();
    public String getNameString();
    public String getOperator();
    public void saveToXML(Element parent);
    public void setAuthorizedName(String s);
    public void setBusinessKey(String s);
    public void setBusinessServices(BusinessServices s);
    public void setCategoryBag(CategoryBag s);
    public void setContacts(Contacts s);
    public void setDefaultDescriptionString(String s);
    public void setDescriptionVector(Vector s);
    public void setDiscoveryURLs(DiscoveryURLs s);
    public void setIdentifierBag(IdentifierBag s);
    public void setName(Name s);
    public void setName(String s);
    public void setOperator(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.datatype.
business.BusinessEntity
```

com.ibm.uddi.datatype.business.Contact

这个类封装了业务联系方式信息，包括人员名称、一个或多个业务地址、电子邮件地址和电话号码。

概要

```
public class Contact extends UDDIElement {
  // 构造函数
    public Contact();
    public Contact(String personName);
    public Contact(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public Vector getAddressVector();
    public String getDefaultDescriptionString();
    public Vector getDescriptionVector();
    public Vector getEmailVector();
    public PersonName getPersonName();
    public String getPersonNameString();
    public Vector getPhoneVector();
    public String getUseType();
    public void saveToXML(Element parent);
    public void setAddressVector(Vector s);
    public void setDefaultDescriptionString(String s);
    public void setDescriptionVector(Vector s);
    public void setEmailVector(Vector s);
    public void setPersonName(PersonName s);
    public void setPersonName(String s);
    public void setPhoneVector(Vector s);
    public void setUseType(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.datatype.
business.Contact
```

com.ibm.uddi.datatype.business.Contacts

这个类封装了多个 `Contact` 对象。

概要

```
public class Contacts extends UDDIElement {
  // 构造函数
    public Contacts();
    public Contacts(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public Vector getContactVector();
    public void saveToXML(Element parent);
    public void setContactVector(Vector s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.datatype.
business.Contacts
```

com.ibm.uddi.datatype.business.Email

这个类封装了一个电子邮件地址。

概要

```
public class Email extends UDDIElement {
  // 构造函数
    public Email();
    public Email(String value);
    public Email(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getText();
    public String getUseType();
    public void saveToXML(Element parent);
    public void setText(String s);
    public void setUseType(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.datatype.
business.Email
```

com.ibm.uddi.datatype.business.PersonName

这个类封装了一个人员名称。

概要

```
public class PersonName extends UDDIElement {
  // 构造函数
    public PersonName();
    public PersonName(String value);
    public PersonName(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getText();
    public void saveToXML(Element parent);
    public void setText(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.datatype.
business.PersonName
```

com.ibm.uddi.datatype.business.Phone

这个类封装了一个电话号码。

概要

```
public class Phone extends UDDIElement {
  // 构造函数
    public Phone();
    public Phone(String value);
    public Phone(Element base) throws UDDIException;
```

```
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getText();
    public String getUseType();
    public void saveToXML(Element parent);
    public void setText(String s);
    public void setUseType(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.datatype.
business.Phone
```

com.ibm.uddi.datatype.service 包

com.ibm.uddi.datatype.service.BusinessService

这个类封装了 UDDI businessService 记录的所有信息。

概要

```
public class BusinessService extends UDDIElement {
  // 构造函数
    public  BusinessService();
    public BusinessService(String serviceKey, String name,
       BindingTemplates bindingTemplates);
    public BusinessService(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public BindingTemplates getBindingTemplates();
    public String getBusinessKey();
    public CategoryBag getCategoryBag();
    public String getDefaultDescriptionString();
    public Vector getDescriptionVector();
    public Name getName();
    public String getNameString();
```

```
    public String getServiceKey();
    public void saveToXML(Element parent);
    public void setBindingTemplates(BindingTemplates s);
    public void setBusinessKey(String s);
    public void setCategoryBag(CategoryBag s);
    public void setDefaultDescriptionString(String s);
    public void setDescriptionVector(Vector s);
    public void setName(Name s);
    public void setName(String s);
    public void setServiceKey(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.datatype.
service.BusinessService
```

com.ibm.uddi.datatype.service.BusinessServices

这个类封装了多个 `BusinessService` 对象。

概要

```
public class BusinessServices extends UDDIElement {
  // 构造函数
    public BusinessServices();
    public BusinessServices(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public Vector getBusinessServiceVector();
    public void saveToXML(Element parent);
    public void setBusinessServiceVector(Vector s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.datatype.
service.BusinessServices
```

com.ibm.uddi.datatype.tmodel 包

com.ibm.uddi.datatype.tmodel.TModel

这个类封装了 UDDI `tModel` 记录的所有信息。

概要

```
public class TModel extends UDDIElement {
  // 构造函数
    public TModel();
    public TModel(String tModelKey, String name);
    public TModel(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String NAICS_TMODEL_KEY;
    public static final String UDDI_TAG;
    public static final String UNSPSC_TMODEL_KEY;
  // 公共方法
    public String getAuthorizedName();
    public CategoryBag getCategoryBag();
    public String getDefaultDescriptionString();
    public Vector getDescriptionVector();
    public IdentifierBag getIdentifierBag();
    public Name getName();
    public String getNameString();
    public String getOperator();
    public OverviewDoc getOverviewDoc();
    public String getTModelKey();
    public void saveToXML(Element parent);
    public void setAuthorizedName(String s);
    public void setCategoryBag(CategoryBag s);
    public void setDefaultDescriptionString(String s);
    public void setDescriptionVector(Vector s);
    public void setIdentifierBag(IdentifierBag s);
    public void setName(Name s);
    public void setName(String s);
    public void setOperator(String s);
    public void setOverviewDoc(OverviewDoc s);
    public void setTModelKey(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.datatype.
tmodel.TModel
```

com.ibm.uddi.response 包

com.ibm.uddi.response.AuthToken

这个类封装了对一个get_authToken()查询的UDDI操作入口响应。用getAuthInfoString()方法获取实际的身份验证标记值。

概要

```
public class AuthToken extends UDDIElement {
  // 构造函数
    public AuthToken();
    public AuthToken(String operator, String authInfo);
    public AuthToken(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public AuthInfo getAuthInfo();
    public String getAuthInfoString();
    public String getOperator();
    public void saveToXML(Element parent);
    public void setAuthInfo(AuthInfo s);
    public void setAuthInfo(String s);
    public void setOperator(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.response.
AuthToken
```

com.ibm.uddi.response.BindingDetail

这个类封装了对一个get_bindingDetail()查询、一个find_binding()查询或一个save_binding()发布请求的UDDI操作入口响应。用getBindingTemplateVector()方法提取BindingTemplate对象的一个Vector。

概要

```
public class BindingDetail extends UDDIElement {
  // 构造函数
```

```
    public BindingDetail();
    public BindingDetail(String operator);
    public BindingDetail(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public Vector getBindingTemplateVector();
    public String getOperator();
    public String getTruncated();
    public boolean getTruncatedBoolean();
    public void saveToXML(Element parent);
    public void setBindingTemplateVector(Vector s);
    public void setOperator(String s);
    public void setTruncated(boolean s);
    public void setTruncated(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.response.
BindingDetail
```

com.ibm.uddi.response.BusinessDetail

这个类封装了对一个`get_businessDetail()`查询或一个`save_business()`发布请求的 UDDI 操作入口响应。用`getBusinessEntityVector()`方法提取`BusinessEntity`对象的一个`Vector`。

概要

```
public class BusinessDetail extends UDDIElement {
  // 构造函数
    public BusinessDetail();
    public BusinessDetail(String operator);
    public BusinessDetail(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public Vector getBusinessEntityVector();
    public String getOperator();
    public String getTruncated();
    public boolean getTruncatedBoolean();
```

```
    public void saveToXML(Element parent);
    public void setBusinessEntityVector(Vector s);
    public void setOperator(String s);
    public void setTruncated(boolean s);
    public void setTruncated(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.response.
BusinessDetail
```

com.ibm.uddi.response.BusinessDetailExt

这个类封装了对一个get_businessDetailExt()查询的UDDI操作入口响应。用getBusinessEntityExtVector()方法提取BusinessEntityExt对象的一个Vector。

概要

```
public class BusinessDetailExt extends UDDIElement {
  // 构造函数
    public BusinessDetailExt();
    public BusinessDetailExt(String operator, Vector businessEntityExt);
    public BusinessDetailExt(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public Vector getBusinessEntityExtVector();
    public String getOperator();
    public String getTruncated();
    public boolean getTruncatedBoolean();
    public void saveToXML(Element parent);
    public void setBusinessEntityExtVector(Vector s);
    public void setOperator(String s);
    public void setTruncated(boolean s);
    public void setTruncated(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.response.
BusinessDetailExt
```

com.ibm.uddi.response.BusinessEntityExt

这个类封装了 UDDI businessEntityExt 记录的信息。

概要

```
public class BusinessEntityExt extends UDDIElement {
  // 构造函数
    public  BusinessEntityExt();
    public BusinessEntityExt(BusinessEntity businessEntity);
    public BusinessEntityExt(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public BusinessEntity getBusinessEntity();
    public void saveToXML(Element parent);
    public void setBusinessEntity(BusinessEntity s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.response.
BusinessEntityExt
```

com.ibm.uddi.response.BusinessInfo

这个类封装了UDDI businessEntity记录的简短信息，包括businessKey、业务名称和描述。用 getServiceInfos()获取由此业务提供的一组服务。

概要

```
public class BusinessInfo extends UDDIElement {
  // 构造函数
    public  BusinessInfo();
    public BusinessInfo(String businessKey, String name, ServiceInfos
        serviceInfos);
    public BusinessInfo(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getBusinessKey();
```

```
    public String getDefaultDescriptionString();
    public Vector getDescriptionVector();
    public Name getName();
    public String getNameString();
    public ServiceInfos getServiceInfos();
    public void saveToXML(Element parent);
    public void setBusinessKey(String s);
    public void setDefaultDescriptionString(String s);
    public void setDescriptionVector(Vector s);
    public void setName(Name s);
    public void setName(String s);
    public void setServiceInfos(ServiceInfos s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.response.
BusinessInfo
```

com.ibm.uddi.response.BusinessInfos

这个类封装了多个`BusinessInfo`对象。用`getBusinessInfoVector()`方法获取`BusinessInfo`对象的一个`Vector`。

概要

```
public class BusinessInfos extends UDDIElement {
  // 构造函数
    public BusinessInfos();
    public BusinessInfos(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public Vector getBusinessInfoVector();
    public void saveToXML(Element parent);
    public void setBusinessInfoVector(Vector s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.response.
BusinessInfos
```

com.ibm.uddi.response.BusinessList

这个类封装了对一个 `find_business()` 查询的 UDDI 操作入口响应。用 `getBusinessInfos()`方法获取一个 `BusinessInfo` 对象。

概要

```
public class BusinessList extends UDDIElement {
  // 构造函数
    public BusinessList();
    public BusinessList(String operator, BusinessInfos businessInfos);
    public BusinessList(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public BusinessInfos getBusinessInfos();
    public String getOperator();
    public String getTruncated();
    public boolean getTruncatedBoolean();
    public void saveToXML(Element parent);
    public void setBusinessInfos(BusinessInfos s);
    public void setOperator(String s);
    public void setTruncated(boolean s);
    public void setTruncated(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.response.
BusinessList
```

com.ibm.uddi.response.DispositionReport

这个类封装了一个UDDI处理报告。根据具体的API调用，`DispositionReport` 可能包括错误信息或一个成功标志。例如，调用`get_businessDetail()`查询失败时会触发带有一个嵌入式 `DispositionReport` 对象的 `UDDIException`。对 `delete_business()`的调用将直接返回一个`DispositionReport`对象；如果删除成功，则成功标志将被设为 `true`。用 `getErrCode()`来获取 UDDI 错误代码（如 `E_invalidKeyPassed`，`E_fatalError`）；用 `getErrno()`获取 UDDI 错误

代码编号（如 10210, 10500）；或者用 getErroInfoText()获取对错误的更完整描述。用 success()方法获取成功标志。用 getGeneric()获取 UDDI 操作入口站点的 UDDI 版本号。

概要

```
public class DispositionReport extends UDDIElement {
  // 构造函数
    public DispositionReport(Element el) throws UDDIException;
  // 字段摘要
    public static final String E_accountLimitExceeded;
    public static final String E_authTokenExpired;
    public static final String E_authTokenRequired;
    public static final String E_categorizationNotAllowed;
    public static final String E_invalidCategory;
    public static final String E_invalidKeyPassed;
    public static final String E_invalidURLPassed;
    public static final String E_keyRetired;
    public static final String E_operatorMismatch;
    public static final String E_userMismatch;
    public static String UDDI_TAG;
  // 公共方法
    public String getErrCode();
    public String getErrInfoText();
    public int getErrno();
    public String getGeneric();
    public String getOperator();
    public boolean isValidElement(Element el);
    public void saveToXML(Element el);
    public boolean success();
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.response.
DispositionReport
```

com.ibm.uddi.response.ErrInfo

这个类封装了有关 UDDI 处理报告的信息。UDDI 客户端代码一般不直接使用这个类。

概要

```
public class ErrInfo extends UDDIElement {
  // 构造函数
    public ErrInfo();
    public ErrInfo(String value, String errCode);
    public ErrInfo(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getErrCode();
    public String getText();
    public void saveToXML(Element parent);
    public void setErrCode(String s);
    public void setText(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.response.
ErrInfo
```

com.ibm.uddi.response.RegisteredInfo

这个类封装了对一个`get_registeredInfo()`查询的UDDI操作入口响应。这个对象包括由指定的身份验证标记控制的一组`businessEntity`键和`tModel`键。

概要

```
public class RegisteredInfo extends UDDIElement {
  // 构造函数
    public RegisteredInfo();
    public RegisteredInfo(String operator, BusinessInfos businessInfos,
        TModelInfos tModelInfos);
    public RegisteredInfo(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public BusinessInfos getBusinessInfos();
    public String getOperator();
    public TModelInfos getTModelInfos();
    public String getTruncated();
```

```
    public boolean getTruncatedBoolean();
    public void saveToXML(Element parent);
    public void setBusinessInfos(BusinessInfos s);
    public void setOperator(String s);
    public void setTModelInfos(TModelInfos s);
    public void setTruncated(boolean s);
    public void setTruncated(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.response.
RegisteredInfo
```

com.ibm.uddi.response.Result

这个类封装了有关 UDDI 处理报告的信息。UDDI 客户端代码一般不直接使用这个类。

概要

```
public class Result extends UDDIElement {
  // 构造函数
    public Result();
    public Result(String errno);
    public Result(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public ErrInfo getErrInfo();
    public String getErrno();
    public String getKeyType();
    public void saveToXML(Element parent);
    public void setErrInfo(ErrInfo s);
    public void setErrno(String s);
    public void setKeyType(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.response.Result
```

com.ibm.uddi.response.ServiceDetail

这个类封装了对一个get_serviceDetail()查询或一个save_service()发布请求的UDDI操作入口响应。用getBusinessServiceVector()方法获取BusinessService对象的一个Vector。

概要

```
public class ServiceDetail extends UDDIElement {
  // 构造函数
    public ServiceDetail();
    public ServiceDetail(String operator);
    public ServiceDetail(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public Vector getBusinessServiceVector();
    public String getOperator();
    public String getTruncated();
    public boolean getTruncatedBoolean();
    public void saveToXML(Element parent);
    public void setBusinessServiceVector(Vector s);
    public void setOperator(String s);
    public void setTruncated(boolean s);
    public void setTruncated(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.response.
ServiceDetail
```

com.ibm.uddi.response.ServiceInfo

这个类封装了UDDI businessService记录的简短信息，包括服务名称、serviceKey和相关的businessKeys。

概要

```
public class ServiceInfo extends UDDIElement {
  // 构造函数
```

```
    public  ServiceInfo();
    public ServiceInfo(String serviceKey, String name);
    public ServiceInfo(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getBusinessKey();
    public Name getName();
    public String getNameString();
    public String getServiceKey();
    public void saveToXML(Element parent);
    public void setBusinessKey(String s);
    public void setName(Name s);
    public void setName(String s);
    public void setServiceKey(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.response.
ServiceInfo
```

com.ibm.uddi.response.ServiceInfos

这个类封装了多个`ServiceInfo`对象。用`getServiceInfoVector()`方法获取`ServiceInfo`对象的一个`Vector`。

概要

```
public class ServiceInfos extends UDDIElement {
  // 构造函数
    public  ServiceInfos();
    public ServiceInfos(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public Vector getServiceInfoVector();
    public void saveToXML(Element parent);
    public void setServiceInfoVector(Vector s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.response.
ServiceInfos
```

com.ibm.uddi.response.ServiceList

这个类封装了对一个`find_service()`查询的UDDI操作入口响应。用`getServiceInfos()`方法获取一个`ServiceInfos`对象。

概要

```
public class ServiceList extends UDDIElement {
  // 构造函数
    public ServiceList();
    public ServiceList(String operator, ServiceInfos serviceInfos);
    public ServiceList(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getOperator();
    public ServiceInfos getServiceInfos();
    public String getTruncated();
    public boolean getTruncatedBoolean();
    public void saveToXML(Element parent);
    public void setOperator(String s);
    public void setServiceInfos(ServiceInfos s);
    public void setTruncated(boolean s);
    public void setTruncated(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.response.
ServiceList
```

com.ibm.uddi.response.TModelDetail

这个类封装了对一个`get_tModelDetail()`查询或一个`save_tModel()`发布请求的UDDI操作入口响应。用`getTModelVector()`方法获取`TModel`对象的一个`Vector`。

概要

```
public class TModelDetail extends UDDIElement {
  // 构造函数
    public TModelDetail();
    public TModelDetail(String operator, Vector tModel);
    public TModelDetail(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getOperator();
    public Vector getTModelVector();
    public String getTruncated();
    public boolean getTruncatedBoolean();
    public void saveToXML(Element parent);
    public void setOperator(String s);
    public void setTModelVector(Vector s);
    public void setTruncated(boolean s);
    public void setTruncated(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.response.
TModelDetail
```

com.ibm.uddi.response.TModelInfo

这个类封装了一个UDDI `tModel`记录的简短信息，包括`tModel`名称和`tModelKey`。

概要

```
public class TModelInfo extends UDDIElement {
  // 构造函数
    public TModelInfo();
    public TModelInfo(String tModelKey, String name);
    public TModelInfo(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public Name getName();
    public String getNameString();
```

```
    public String getTModelKey();
    public void saveToXML(Element parent);
    public void setName(Name s);
    public void setName(String s);
    public void setTModelKey(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.response.
TModelInfo
```

com.ibm.uddi.response.TModelInfos

这个类封装了多个 `TModelInfo` 对象。用 `getTModelInfoVector()`方法获取 `TModelInfo` 对象的一个 `Vector`。

概要

```
public class TModelInfos extends UDDIElement {
  // 构造函数
    public TModelInfos();
    public TModelInfos(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public Vector getTModelInfoVector();
    public void saveToXML(Element parent);
    public void setTModelInfoVector(Vector s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.response.
TModelInfos
```

com.ibm.uddi.response.TModelList

这个类封装了对一个 `find_tModel()`查询的 UDDI 操作入口响应。用 `get-TModelInfos()`方法获取一个 `TModelInfo` 对象。

概要

```
public class TModelList extends UDDIElement {
  // 构造函数
    public TModelList();
    public TModelList(String operator, TModelInfos tModelInfos);
    public TModelList(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getOperator();
    public TModelInfos getTModelInfos();
    public String getTruncated();
    public boolean getTruncatedBoolean();
    public void saveToXML(Element parent);
    public void setOperator(String s);
    public void setTModelInfos(TModelInfos s);
    public void setTruncated(boolean s);
    public void setTruncated(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.response.
TModelList
```

com.ibm.uddi.util 包

com.ibm.uddi.util.AuthInfo

这个类封装了UDDI身份验证标记。用`getText()`方法获取实际的身份验证标记值。

概要

```
public class AuthInfo extends UDDIElement {
  // 构造函数
    public AuthInfo();
    public AuthInfo(String value);
    public AuthInfo(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
```

```
  // 公共方法
    public String getText();
    public void saveToXML(Element parent);
    public void setText(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.util.AuthInfo
```

com.ibm.uddi.util.BindingKey

这个类封装了关于一个UDDI BindingKey的信息，但是客户端代码一般不直接使用这个类。

概要

```
public class BindingKey extends UDDIElement {
  // 构造函数
    public BindingKey();
    public BindingKey(String value);
    public BindingKey(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getText();
    public void saveToXML(Element parent);
    public void setText(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.util.BindingKey
```

com.ibm.uddi.util.BusinessKey

这个类封装了有关一个UDDI BusinessKey的信息，但是客户端代码一般不直接使用这个类。

概要

```
public class BusinessKey extends UDDIElement {
  // 构造函数
    public BusinessKey();
    public BusinessKey(String value);
    public BusinessKey(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getText();
    public void saveToXML(Element parent);
    public void setText(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.util.BusinessKey
```

com.ibm.uddi.util.CategoryBag

这个类封装了多个 `KeyedReference` 对象，每个对象表示一个 UDDI 类别。如 `CategoryBag` 对象就可能含有一个或多个 NAICS 代码。这个类经常作为参数被传递给任何 `find_xxx()` 函数。

概要

```
public class CategoryBag extends UDDIElement {
  // 构造函数
    public CategoryBag();
    public CategoryBag(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public Vector getKeyedReferenceVector();
    public void saveToXML(Element parent);
    public void setKeyedReferenceVector(Vector s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.util.
CategoryBag
```

com.ibm.uddi.util.DiscoveryURL

这个类封装了一个 UDDI `DiscoveryURL` 的信息。

概要

```
public class DiscoveryURL extends UDDIElement {
  // 构造函数
    public DiscoveryURL();
    public DiscoveryURL(String value, String useType);
    public DiscoveryURL(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getText();
    public String getUseType();
    public void saveToXML(Element parent);
    public void setText(String s);
    public void setUseType(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.util.
DiscoveryURL
```

com.ibm.uddi.util.DiscoveryURLs

这个类封装了多个 `DiscoveryURL` 对象。

概要

```
public class DiscoveryURLs extends UDDIElement {
  // 构造函数
    public DiscoveryURLs();
    public DiscoveryURLs(Vector discoveryURL);
    public DiscoveryURLs(Element base) throws UDDIException;
```

```
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public Vector getDiscoveryURLVector();
    public void saveToXML(Element parent);
    public void setDiscoveryURLVector(Vector s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.util.
DiscoveryURLs
```

com.ibm.uddi.util.FindQualifier

这个类封装了单个UDDI `FindQualifier`，用于指定更精确的搜索条件。例如，被设为`exactNameMatch`的`FindQualifier`要求只能从`find_xxx()`函数调用返回那些名称完全匹配的结果。

概要

```
public class FindQualifier extends UDDIElement {
  // 构造函数
     public FindQualifier();
    public FindQualifier(String value);
    public FindQualifier(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
     public static final String caseSensitiveMatch;
    public static final String exactNameMatch;
    public static final String sortByDateAsc;
    public static final String sortByDateDesc;
    public static final String sortByNameAsc;
    public static final String sortByNameDesc;
   public static final String UDDI_TAG;
  // 公共方法
    public String getText();
   public void saveToXML(Element parent);
    public void setText(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.util.
FindQualifier
```

com.ibm.uddi.util.FindQualifiers

这个类封装了多个FindQualifier对象。该类经常作为任何find_xxx()函数调用的参数被传送。

概要

```
public class FindQualifiers extends UDDIElement {
  // 构造函数
    public FindQualifiers();
    public FindQualifiers(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public Vector getFindQualifierVector();
    public void saveToXML(Element parent);
    public void setFindQualifierVector(Vector s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.util.
FindQualifiers
```

com.ibm.uddi.util.IdentifierBag

这个类封装了多个KeyedReference对象，每个对象表示一个UDDI标识符。例如，IdentifierBag对象可能含有一个或多个Dun & Bradstreet D-U-N-S® Number。这个类经常作为任何find_xxx()函数调用的参数被传送。

概要

```
public class IdentifierBag extends UDDIElement {
  // 构造函数
    public IdentifierBag();
```

```
    public IdentifierBag(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public Vector getKeyedReferenceVector();
    public void saveToXML(Element parent);
    public void setKeyedReferenceVector(Vector s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.util.
IdentifierBag
```

com.ibm.uddi.util.KeyedReference

这个类封装了单个UDDI `keyedReference`记录。更多细节请参阅`CategoryBag`和`IdentifierBag`。

概要

```
public class KeyedReference extends UDDIElement {
  // 构造函数
    public KeyedReference();
    public KeyedReference(String keyName, String keyValue);
    public KeyedReference(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getKeyName();
    public String getKeyValue();
    public String getTModelKey();
    public void saveToXML(Element parent);
    public void setKeyName(String s);
    public void setKeyValue(String s);
    public void setTModelKey(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.util.
KeyedReference
```

com.ibm.uddi.util.KeyValue

这个类封装了有关一个 UDDI KeyValue 的信息，但客户端代码一般不直接使用这个类。

概要

```
public class KeyValue extends UDDIElement {
  // 构造函数
    public KeyValue();
    public KeyValue(String value);
    public KeyValue(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getText();
    public void saveToXML(Element parent);
    public void setText(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.util.KeyValue
```

com.ibm.uddi.util.ServiceKey

这个类封装了有关一个 UDDI ServiceKey 的信息，但客户端代码一般不直接使用这个类。

概要

```
public class ServiceKey extends UDDIElement {
  // 构造函数
    public ServiceKey();
    public ServiceKey(String value);
    public ServiceKey(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
     public String getText();
    public void saveToXML(Element parent);
```

```
        public void setText(String s);
    }
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.util.ServiceKey
```

com.ibm.uddi.util.TModelBag

这个类封装了多个 TModelKey 对象。

概要

```
public class TModelBag extends UDDIElement {
  // 构造函数
    public TModelBag();
    public TModelBag(Vector tModelKeyStrings);
    public TModelBag(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public Vector getTModelKeyStrings();
    public Vector getTModelKeyVector();
    public void saveToXML(Element parent);
    public void setTModelKeyStrings(Vector s);
    public void setTModelKeyVector(Vector s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.util.TModelBag
```

com.ibm.uddi.util.TModelKey

这个类封装了一个 UDDI tModelKey。

概要

```
public class TModelKey extends UDDIElement {
  // 构造函数
    public TModelKey();
```

```
    public TModelKey(String value);
    public TModelKey(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getText();
    public void saveToXML(Element parent);
    public void setText(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.util.TModelKey
```

com.ibm.uddi.util.UploadRegister

这个类封装了一个UDDI `UploadRegister`。`UploadRegister`表示某完整XML文档的URL，可以被传送给几个`save_xxx()`函数。例如，`UploadRegister`可能表示一个UDDI `businessEntity`记录的URL，要发布这个记录，请使用`save_business(String authInfo,UploadRegister[] uploadRegisters)`方法。注意，并非所有的UDDI操作入口站点都支持`UploadRegister`工具。

概要

```
public class UploadRegister extends UDDIElement {
  // 构造函数
    public UploadRegister();
    public UploadRegister(String value);
    public UploadRegister(Element base) throws UDDIException;
  // 字段摘要
    protected Element base;
    public static final String UDDI_TAG;
  // 公共方法
    public String getText();
    public void saveToXML(Element parent);
    public void setText(String s);
}
```

层次

```
java.lang.Object → com.ibm.uddi.UDDIElement → com.ibm.uddi.util.UploadRegister
```

词汇表

Apache SOAP

SOAP 协议的开源 Java 实现。由 Apache Software Foundation 提供。

BEEP

块可扩展交换协议(Blocks Extensible Exchange Protocol)，由 Marshall Rose 开发，现在是一个官方 IETF 规范。BEEP 是建立新的应用程序协议时最好的实践框架。它在 TCP 上直接分层，包括若干内置功能，如初始握手协议、身份验证、安全和错误处理。BEEP 目前还没有被广泛部署，但它有可能取代 HTTP 成为一个可行的远程过程调用的协议。也可参阅 *HTTP*。

binding element（binding 元素）

WSDL `binding` 元素为 Web 服务的 XML 消息接发层指定实现细节。WSDL 包括用于定义 SOAP 服务的内置绑定扩展。

bindingTemplate

一个 UDDI XML 元素，包含有关从哪里和如何访问指定 Web 服务的信息。还可参阅 *businessService* 和 *tModel*。

businessEntity

一个 UDDI XML 元素，包含有关已注册业务的信息，如业务名称、描述、地址和联系信息。

businessService

一个 UDDI XML 元素，包含有关已注册的 Web 服务或一组相关 Web 服务的信息。包括名称、描述和 `bindingTemplate` 的一个可选列表。也可参阅 *bindingTemplate*。

deployment descriptor file（部署描述文件）

一个 XML 文件，Apache SOAP 用它来定义和部署一个指定的 SOAP 服务。它包含服务 URN、一组服务方法、应用程序作用域、Java 提供者和 Java 到 XML 的类型映射。

deserialize（解串行化）
在网络连接上接收数据和重建指定的变量或对象的一种普通技术。如Java SOAP客户端可以通过网络连接接收XML消息，并用这些消息重建正确的Java对象。还可参阅*serialize*、*XML data type*和*type mapping registry*。

disposition report（处理报告）
一个UDDI XML元素，表示一个UDDI操作是成功还是失败。失败时，处理报告将会包含一个可供阅读的错误原因解释。

DNS
域名系统（Domain Name System）。将域名转换成IP地址的一个分布式系统。还可参阅*IP address*。

GLUE
由The Mind Electric公司建立的Web服务平台，支持SOAP、WSDL和UDDI。

green page（绿页）
UDDI中使用的一种通用数据种类，用来指定有关Web服务的技术信息。一般包括一个指向外部规范的指针和一个调用Web服务的地址。还可参阅*bindingTemplate*和*tModel*。

HTTP
超文本传输协议（Hypertext Transfer Protocol）。HTTP是Web浏览器和Web服务器间交换数据的主要协议。HTTP的设计初衷是获取远程文档，但是现在SOAP和XML-RPC用它调用远程过程。还可参阅*BEEP*。

IETF
Internet工程任务组（Internet Engineering Task Force）。IETF制订了Internet协议的主要标准，包括HTTP和BEEP。

IP
Internet协议（Internet Protocol），是通过Internet传递数据包的主要协议。还可参阅*TCP*。

IP address（IP地址）
一个惟一的32位地址，用来标识Internet上的一台机器。

ISO 3166
由国际标准化组织（International Organization for Standardization，ISO）维护的一个国家代码标准分类法。如中国的代码是CN，美国的代码是US。使用ISO 3166，在UDDI注册的公司可以标识总部的地理位置或其业务的主要地理区域。ISO 3166还用于顶级Internet域名国家代码。

MIME
多用途Internet邮件扩展（Multi-Purpose Internet Mail Extensions）。MIME是传输和附加图像、声音、视频等二元数据的标准技术。

mustUnderstand
一个SOAP `Header`属性，用来表明一个指定的首部是强制的还是可选的。如果被设为`true`，则接收者一定要理解并处理这个`Header`属性；否则，它必须忽略这个消息并返回一个`Fault`。还可参阅*SOAP Header*。

NAICS
北美工业分类系统（North American Industry Classification System）。NAICS为超过19000个行业提供了一

个六位数的行业代码。从1997年起，NAICS就取代了原来的标准工业分类（Standard Industry Classification，SIC）。NAICS在UDDI中作为一个标准分类法，为业务和业务服务分类。

port（端口）

一个逻辑上的连接点，TCP/IP服务器在这里监听客户端请求。HTTP使用端口80。

portType

WSDL `portType`元素结合了多个`message`元素，形成一个单向或往返操作。例如，`portType`可以结合一个请求消息和一个响应消息，以形成一个单一的请求/响应操作，就像在SOAP服务中常用的那样。

remote procedure call（远程过程调用）

一种通用技术，应用程序可以使用这种技术通过网络连接到另一个应用程序，调用它的函数，并接收调用的结果。远程过程调用（RPC）广泛应用于分布式应用程序框架中，如CORBA、分布式COM、Java RMI、SOAP和XML-RPC。

rpcrouter

接收SOAP请求，并将它们发送到合适的服务类加以处理的Apache SOAP servlet。

SAML

安全断言标记语言（Security Assertion Markup Language），由结构化信息标准推进组织（Organization for the Advancement of Structured Information Standards，OASIS）开发。SAML有助于业务合作者间的身份验证和授权信息交换。

Semantic Web（语义Web）

Tim Berners-Lee首次使用这个术语，他创立了万维网（World Wide Web）。在语义Web描绘的广阔世界里，应用程序可以很容易地使用和理解Web，就像人们现在浏览Web一样。Berners-Lee还指出，Web服务是他的语义Web梦想的一个重要实现。

serialize（串行化）

一种将变量或数组转换成标准格式以便在网络上传输的通用技术。例如，Java SOAP客户端将Java对象串行化成一种标准的XML格式，并在网络上传输这种XML。还可参阅*deserialize*、*XML data type*和*type mapping registry*。

service description（服务描述）

Web服务协议栈中的一个层，负责描述连接到一个特定Web服务的公共接口。还可参阅*WSDL*。

service provider（服务提供者）

在Web服务体系结构中，服务提供者是实现Web服务并使他人能在Internet上访问Web服务的任意一台主机，也就是传统的客户端/服务器体系结构中的一台服务器。

service registry（服务注册中心）

在Web服务体系结构中，服务注册中心是一个逻辑上集中的服务目录。开发者可以连接到服务注册中心，发布新的服务或查找已有服务。还可参阅*UDDI*。

service requestor（服务请求者）
在Web服务体系结构中，服务请求者是Web服务的消费者。请求者通过打开一个网络连接并发送XML请求来使用已有的Web服务，这与传统的客户端/服务器体系结构中的客户端一样。

service type（服务类型）
参阅 *tModel*。

SOAP
SOAP是一个基于XML的协议，用于在计算机之间交换信息。虽然SOAP可用于各种消息接发系统，并可以通过各种传输协议进行传递，但其侧重点是通过HTTP传输的远程过程调用（RPC）。与XML-RPC相同，SOAP独立于平台。因此，它能通过网络连接使各种不同的应用程序相互通信。

SOAP Body
SOAP Body元素封装了SOAP消息的主要“有效负载”。有效负载包括有关远程过程调用的细节，包括要调用的方法名、方法参数或返回值。Body元素还可以包含可选的、用来指定错误情况的Fault元素。

SOAP Envelope
SOAP XML Envelope元素封装了一个通过SOAP传输的XML消息。Envelope指定SOAP的版本，由一个可选的SOAP Header和一个必需的SOAP Body组成。还可参阅 *SOAP Header* 和 *SOAP Body*。

SOAP Header
可选的SOAP Header元素提供了一个灵活的框架，用于为某个SOAP消息指定更多的应用程序级属性。Header框架可用于各种应用程序，包括用户身份验证、事务管理或支付授权。还可参阅 *SOAP Envelope* 和 *mustUnderstand*。

SOAP::Lite for Perl（面向Perl的SOAP::Lite）
用于Perl的SOAP库。包括对SOAP、XML-RPC、WSDL和UDDI的支持。

SOAPAction Header
用于表明SOAP消息意图的HTTP Header。有些SOAP服务器要求客户端指定一个完整的SOAPAction值，如SOAPAction: "urn:xmethods-BabelFish#BabelFish"；而包括Apache SOAP在内的其他SOAP服务器只需要客户端指定一个空白SOAPAction（如SOAPAction: ""）。在SOAP 1.1下一定要有SOAPAction Header，而在SOAP 1.2下可有可无。

SOAP-DSIG
SOAP安全扩展：数字签名（SOAP Security Extensions: Digital Signature）。SOAP-DSIG使用公共密钥加密技术实现SOPA消息的数字签名。这种方法使客户端或服务器能够验证另一方的身份。SOAP-DSIG已被提交到W3C。

socket（套接字）
一种抽象。它通过将程序员与底层网络协议的细节隔离开而使网络编程更方便。

targetNamespace
XML Schema的一种约定，使XML文档能够引用自身。任何新定义的元素都将属于某个指定的targetNamespace。还可参阅 *XML Schema*。

TCP

传输控制协议(Transmission Control Protocol)。TCP主要负责将消息分解成单个的IP数据包，并在终点将这些数据包重新组织成消息。还可参阅*IP*。

TcpTunnelGui

与 Apache SOAP 捆绑在一起的一个工具，使你可以很容易地截获和察看 SOAP 请求和响应。它是调试 SOAP 应用程序的一个极佳工具。

tModel

技术模型，是一个UDDI XML元素，用于提供指向外部技术规范的指针。也被称为服务类型。

type mapping registry（类型映射注册表）

在Apache SOAP中，类型映射注册表将XML元素映射为Java类或反之。默认状态下，注册表预置了基本数据类型，如字符串、向量、日期和数组。如果你要传送新的数据类型，那么需要显式注册你的新类型，并指明哪个 Java 类负责串行化和解串行化这个新类型。还可参阅*serialize*和*deserialize*。

UDDI

通用描述、发现和集成（Universal Description, Discovery, and Integration）。UDDI目前代表Web服务协议栈中的发现层。UDDI最初由微软、IBM和Ariba创建，是发布和查找业务和 Web 服务的一个技术规范。还可参阅 *UDDI cloud services*。

UDDI Business Registry（UDDI业务注册中心）

参阅 *UDDI cloud services*。

UDDI cloud services（UDDI服务群）

也称为UDDI业务注册中心，是UDDI规范的一个完全可操作的实现。2001年5月，微软和IBM发起了UDDI服务群，现在已经实现了任何人都可以搜索已有的UDDI数据或发布新业务和服务数据。

UDDI4J

IBM 开发的一个开源 UDDI 库。

UNSPSC

统一标准产品和服务分类(Universal Standard Products and Service Classification)。UNSPSC 为产品和服务分类提供标准代码。UNSPSC于1998年被开发出来，现在由非营利的电子商业代码管理协会（Electronic Commerce Code Management Association，ECCMA）维护。它为54 个行业的产品和服务提供了超过12000 个代码。UNSPSC 在 UDDI 中是一个将业务和业务服务分类的标准分类法。

URN

统一资源名称（Uniform Resource Name)。URN是一个长期有效并与位置无关的统一资源标识符（Uniform Resource Identifier，URI)。如`urn:isbn:0596000588`指的是《XML in a Nutshell》一书（由O'Reilly出版)。URN经常被用来标识SOAP服务。

W3C

万维网联盟（World Wide Web Consortium)。W3C 制订了包括HTML、XML、XML Schema、SOAP

和XML加密等在内的Web协议和规范的主要标准。还可参阅*W3C XML Protocol Group*。

W3C XML Protocol Group（W3C XML协议工作组）

W3C XML协议工作组成立于2000年9月，致力于Web服务协议的标准化。其首要目标是建立一个官方的SOAP规范。

W3C Web Services Activity

W3C Web Services Activity成立于2002年1月，包括W3C XML协议工作组以及体系结构和描述（Architecture and Description）工作组。

web service（Web服务）

Web服务是指可以从Internet上获得、使用标准XML消息接发系统、不受操作系统和编程语言约束的任何服务。虽然并非必需，但Web服务应该还能够通过共同的XML格式自描述并通过简单查找机制被发现。

web service protocol stack（Web服务协议栈）

一个新兴的、用于建立和描述Web服务的协议栈。当前的Web服务栈由四层组成：服务传输（HTTP、FTP、BEEP等）、XML消息接发（XML-RPC、SOAP）、服务描述（WSDL）和服务发现（UDDI）。

white page（白页）

UDDI中用于指定业务名称、业务描述和地址等业务信息的通用数据种类。还可参阅*businessEntity*。

WSDL

Web服务描述语言（Web Services Description Language，WSDL）。目前WSDL代表Web服务协议栈的服务描述层。WSDL是为Web服务指定公共接口的一种XML语法。这个公共接口包括所有公用函数的信息、所有XML消息的数据类型信息、有关要使用特定传输协议的绑定信息和用于查找指定服务的地址信息。WSDL不一定要绑定到某个XML消息接发系统，但它确实包含了描述SOAP服务的内置扩展。

WSIF

Web服务调用框架（Web Services Invocation Framework）。WSIF是IBM创建的一个框架，它使程序员不用编写任何SOAP特有的代码就可以调用SOPA服务，还实现了根据WSDL文件自动调用SOAP服务。

XKMS

XML密钥管理服务（XML Key Management Services）是分配和管理公共密钥和证书的一个推荐Web服务规范。XKMS已被提交到W3C。

XML

可扩展标记语言（eXtensible Markup Language）是W3C的一个官方建议。XML是组织和共享数据的一个灵活框架，在Web服务协议栈的XML消息接发、服务描述和服务发现层中广泛使用。

XML data type（XML数据类型）

表明可能放置到某个XML元素中的数据类型。XML Schema包含对字符

串、整型、浮点型和双精度型等基本数据类型的内置支持。还可参阅*XML Schema* 和 *type mapping registry*。

XML Encryption Standard（XML 加密标准）
: 加密/解密整个或部分XML文档的一个 W3C 推荐框架。

XML namespace（XML 名称空间）
: 为区分名称相同的 XML 元素和属性提供了一个标准机制。SOAP 规范大量使用 XML 名称空间。

XML Schema
: 用来定义XML文档规则的一个框架。XML Schema包括为单个元素指定数据类型的能力，这种能力是远程过程调用（RPC）的一个关键因素。

XML-RPC
: 使用XML消息以通过HTTP执行RPC的一个协议。和 SOAP 一样，XML-RPC不依赖于操作平台，因此使不同的应用程序可以通过网络连接互相通信。

yellow page（黄页）
: UDDI 中为公司或公司所提供的服务分类的一种通用数据种类。数据可以包括行业、产品或根据标准分类法确定的地理代码。还可参阅 *ISO 3166*、*NAICS*、*tModel* 和 *UNSPSC*。