本章内容：

- 开始
- 配置
- 深入实践
- 继续深入
- 下章预告

第十二章

SOAP

SOAP即Simple Object Access Protocol（简单对象访问协议）。不会还没听说过吧，地球上的人都知道啊。SOAP正成为Web编程世界最炙手可热的概念，而且还是席卷最新一代Web开发的Web服务热潮的组成部分。如果你听说过微软的.NET计划或P2P（peer-to-peer，对等网）革命，一定也听说了这些技术是以SOAP为基础的（即使你还不知道它）。Apache计划里有两个（而不是一个）SOAP的实现，而微软的MSDN网站上也有成百上千页专门讲SOAP（*http://msdn.microsoft.com*）。

本章中，我们解释什么是SOAP，为什么它是如此重要的Web开发模式的未来方向的一部分。这将有助于掌握基础，并为实际使用SOAP工具箱打铺垫。然后，简要浏览现有的SOAP项目，并深入探究Apache的实现。本章并不是对SOAP的完整介绍，下一章还将弥补许多缺漏。把这作为第一部分，本章中许多未决问题将在下一章中回答。

开始

我们先搞清楚SOAP是什么。这可以通过到*http://www.w3.org/TR/SOAP*阅读完整的W3C提文规范来实现，这份文档很长。但是如果除去所有表面的虚词，就能发现SOAP只是一个协议而已。它是个简单的协议（只要用就行，没必要自己写），其基本出发点是分布式架构中需要交换信息。而且，在一个可能过负荷的系统中，该出发协议是轻量级的，只需要很少的开销。最后它允许所有操作都在HTTP上进行，

这样就绕过了防火墙等棘手问题，并且可以避免使用监听千奇百怪端口号的各种套接字。了解了这一点，其他的一切都只是细节了。

当然，我们需要的正是细节，因此不会把它们放过去。SOAP 规范中有三个基本组成部分：SOAP 封装（envelope），一套编码规则，一种在请求和响应之间交互的方式。刚开始我们把 SOAP 报文想像成一封实际的信，这些包含奇怪东西的信封也带有邮资标记，上面也写满了地址。这一比喻有助于使“ 封装 ”这样的概念更好理解。图 12-1 就按这种比喻说明了 SOAP 的工作过程。

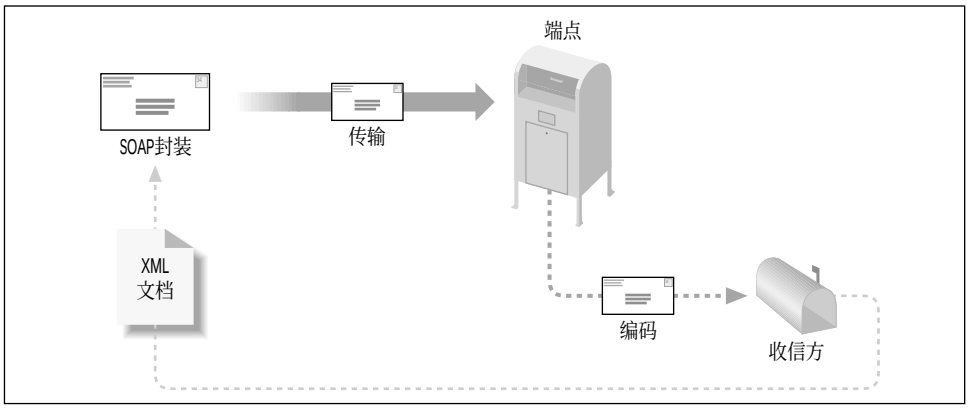

图 12-1：SOAP 报文过程

有这张图在脑子里，让我们看一看 SOAP 规范的三个组成部分。我们只是简述，而用例子来更完整地说明概念。而且，正是这三个关键的组成部分使 SOAP 如此重要和有价值。错误处理，各种编码方式的支持，自定义参数的串行化，以及 SOAP 运行于 HTTP 之上这一事实，使它在许多情况下都比其他分布式协议（注 1）更具吸引力。而且，SOAP 提供了与其他程序的高度互操作性，这一点下一章更加全面地讲解。现在，先集中在 SOAP 的基本组成部分上。

封装

SOAP 封装与真正的信封很像。它提供在 SOAP 负荷中编码的报文的信息，包括与收信人和发信人的数据，以及报文本身的细节。例如，SOAP 封装的首部可以精确

注 1： 关于 SOAP 运行于其他协议如 SMTP（甚至 Jabber）之上的研究也有很多进展。但这不是 SOAP 标准的组成部分，但可能在未来加入。所以如果有人谈论及此不用太惊讶。

规定报文的处理方式。在程序处理报文之前，程序可以确定报文的信息，包括是否能处理报文。与标准XML-RPC调用的情形不同（记得吗？XML-RPC报文、编码和其他都封装在一个XML片段中），SOAP要进行解释以确定报文的情况。普通的SOAP报文还含有编码形式（encoding style），它有助于收信人解释报文。例12-1是SOAP封装，带有指定的编码。

例12-1：SOAP封装

```
<soap:Envelope
   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
   soap:encodingStyle="http://myHost.com/encodings/secureEncoding"
>
  <soap:Body>
    <article xmlns="http://www.ibm.com/developer">
      <name>Soapbox</name>
      <url>
        http://www-106.ibm.com/developerworks/library/x-soapbx1.html
      </url>
    </article>
  </soap:Body>
</soap:Envelope>
```

可以看到编码是在封装中指定的，可被程序用来确定（用`encodingStyle`属性值）它是否能阅读`Body`元素中的报文。一定要把SOAP封装的名字空间弄对了，否则接收报文的SOAP服务器将触发版本不匹配错误，而且也无法与它们互操作。

编码

第二个SOAP组成元素是对用户定义数据类型编码的简单方式。在RPC（和XML-RPC）中，编码只用于事先定义好的数据类型集合：XML-RPC工具所支持的那些。对其他类型编码，需要修改RPC服务器和客户端。而在SOAP中，可以用XML模式容易地指定新的数据类型（用第二章中讨论过的`complexType`结构），这些新的类型可以容易地用XML表示为SOAP负荷的一部分。因为这种与XML模式的结合，我们可以在SOAP报文中编码任何数据类型，只要能用XML模式合乎逻辑地描述即可。

调用

理解SOAP调用过程的最佳方式，是将它与已经知道的技术和XML-RPC相比较。调用时，XML-RPC调用应大致如例12-2中的代码段所示。

例 12-2：XML-RPC 中的调用

```
// 指定要用的 XML 解析器
XmlRpc.setDriver("org.apache.xerces.parsers.SAXParser");

// 指定要连接的服务器
XmlRpcClient client =
    new XmlRpcClient("http://rpc.middleearth.com");

// 生成参数
Vector params = new Vector();
params.addElement(flightNumber);
params.addElement(numSeats);
params.addElement(creditCardType);
params.addElement(creditCardNum);

// 请求预订
Boolean boughtTickets =
    (Boolean)client.execute("ticketCounter.buyTickets", params);

// 处理响应
```

这是一个简单的售票程序。例 12-3 是 SOAP 中同样的调用。

例 12-3：SOAP 中的调用

```
// 生成参数
Vector params = new Vector();
params.addElement(
    new Parameter("flightNumber", Integer.class, flightNumber, null));
params.addElement(
    new Parameter("numSeats", Integer.class, numSeats, null));
params.addElement(
    new Parameter("creditCardType", String.class, creditCardType, null));
params.addElement(
    new Parameter("creditCardNumber", Long.class, creditCardNum, null));

// 创建 Call 对象
Call call = new Call();
call.setTargetObjectURI("urn:xmltoday-airline-tickets");
call.setMethodName("buyTickets");
call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
call.setParams(params);

// 调用
Response res = call.invoke(new URL("http://rpc.middleearth.com"), "");

// 处理响应
```

可以看到，`Call`对象表示的调用是驻留在内存中的。这样可以设置调用的目标、方法、编码形式、参数以及更多这里没有的东西。它比 XML-RPC 的方式更灵活，可

以显式地设置XML-RPC中隐式确定的各种参数。在本章其他部分还能看到更多调用过程，包括SOAP提供错误响应、错误层次以及调用返回结果的方式。

以上的简介已经足够了。下面介绍我们要用到的SOAP实现，并解释为什么要选用它，并看一些代码。

配置

基本概念有了，该看代码了。我们需要用到某个项目或产品，找起来比我们想像的要简单。如果需要的是基于Java的提供SOAP功能的项目，不用到别处去。有两类产品：商业的和自由的。与本书其他部分一样，我们不会使用商业产品。这不是因为它们不好（相反，有一些很棒），只是因为我认为应该让每个读者都能应用每个例子。这要求容易获得，商业产品是无法提供的。必须为此付费，要么就是下载只能用一阵子的试用版。

因此我们倾向于开源产品。在这里，我选择的是Apache SOAP，网址为*http://xml.apache.org/soap*。该项目旨在提供Java的SOAP工具箱。目前版本为2.2，可以自由下载。本章将全部使用这一版本。

其他选择

在安装和配置Apache SOAP之前，要回答几个总在脑里转的问题。不使用商业产品的原因已经清楚了。但是，有人会问了，还有一些开源或相关的选择呢，为什么不涉及呢？

IBM SOAP4J怎么样？

其他选择列表上第一个是IBM的SOAP实现：SOAP4J。这一产品实际上是当前Apache SOAP项目的基础，就像IBM XML4J融入了Apache Xerces XML解析器项目一样。估计IBM的实现还会浮出表面，封装Apache SOAP项目的实现。这与IBM的XML4J类似，它现在就是将Xerces封装而产生的。这样的产品会在开源版本上增加一些厂商支持，虽然两者代码是相同的。

微软如何呢？

毫无疑问，微软及其SOAP实现以及整个.NET计划（下一章进一步讲述）都非常重要。事实上，我很愿意花更多时间详细讲一讲微软的SOAP实现，但它只支持COM一类的对象，不支持Java。因此，这不适合本书。但是，微软（先把开发人员普遍有的对该公司的潜在抵触心理放在一边）正在Web服务方面发挥重要作用，只字不提肯定是错误的。如果要与COM或VB组件通信，强烈推荐下载微软的SOAP工具包，网址为*http://msdn.microsoft.com/library/default.asp?url=/nhp/Default.asp?contentid=28000523*，还有许多其他SOAP资源。

Axis是什么？

关注Apache的人可能听说过Apache Axis。这是下一代SOAP工具包，也是在Apache XML大旗下开发的。因为SOAP（规范而不是具体实现）都在快速而剧烈地发生变化，跟踪是很困难的。要创建既反映当前要求又能跟上新进展的SOAP实现太难了。因此，当前的Apache SOAP是比较有限的。Apache开发人员没有重新构建现有的工具包，而是另起炉灶，启动了全新的代码基础和项目 —— Axis诞生了。而且，SOAP的名字也在变化，从SOAP变成XP又变成XMLP。因此新的SOAP项目不再含有规范名，所以取名为“Axis”。当然，现在W3C也重新称规范为SOAP了（1.2版或2.0版），这就更令人迷惑了！

可以把IBM SOAP4J看作SOAP工具包的架构1。然后是Apache SOAP（本章的主题），架构2。最后Axis是下一代的架构了。此项目是由SAX驱动的，而Apache SOAP则基于DOM。而且，Axis在Apache SOAP中忽略的首部交互上提供了更易用的方式。既然Axis有这么多改进，为什么不讲呢？为时尚早呗。Axis现在的版本只是0.51。还不是beta版，甚至不是alpha版，太早了。虽然我乐于讲述所有Axis的新特性，但老板肯定不会允许我们在重要的系统中放入alpha版本之前的开源软件的，对不对？因此，我们还是把注意力放到今天就能用的产品Apache SOAP上。我保证Axis完成时，会在本书后续版本中更新这一章。但现在还是专注在可用的解决方案上吧。

安装

SOAP的安装有两个方面。一个是SOAP客户，它使用SOAP API与能接收SOAP

报文的服务器通信。另一个是 SOAP 服务器，它可以接收来自 SOAP 客户的报文。本节两方面都要讲到。

客户端

在客户端使用 SOAP，首先要从 *http://xml.apache.org/dist/soap* 下载 Apach SOAP。我下载的版本是二进制格式的 2.2 版（在 *version-2.2* 子目录中）。然后解压缩到一个目录中，比如 *javaxml2*（Windows 机器上是 *c:\javaxml2*，Mac OSX 机器上是 */javaxml2*），生成 */javaxml2/soap-2_2*。还要从 Sun 的网站 *http://java.sun.com/products/javamail/* 下载 JavaMail 包，以支持 Apache SOAP 中的 SMTP 传输协议。然后从Sun的网站*http://java.sun.com/products/beans/glasgow/jaf.html*下载JavaBeans Activation Framework（JAF）。假设已经有 Xerces 或其他 XML 解析器可用。

注意：要确保 XML 解析器是与 JAXP 兼容且支持名字空间的。如果有问题，就还是用 Xerces。使用新版本的 Xerces，1.4 以上就够了。SOAP 和 Xerces 1.3（及 1.3.1）有很多问题，尽量避免这种组合。

解压缩 JavaMail 和 JAF 包，然后将 *jar* 文件以及 *soap.jar* 库导入类路径。这些 *jar* 文件都在根目录下或相关安装目录的 *lib/* 子目录下。最后类路径应该类似于：

```
$ echo $CLASSPATH
/javaxml2/soap-2_2/lib/soap.jar:/javaxml2/lib/xerces.jar:
/javaxml2/javamail-1.2/mail.jar:/javaxml2/jaf-1.0.1/activation.jar
```

Windows 上应类似于：

```
c:\>echo %CLASSPATH%
c:\javaxml2\soap-2_2\lib\soap.jar;c:\javaxml2\lib\xerces.jar;
c:\javaxml2\javamail-1.2\mail.jar;c:\javaxml2\jaf-1.0.1\activation.jar
```

将 *java xml2/soap-2_2/* 目录加入类路径，如果要运行 SOAP 实例的话。本章将在讲述具体例子时讲具体的配置。

服务器端

要创建一套 SOAP 服务器组件，首先需要有 Servlet 引擎。与前几章中一样，本章所有例子我们用 Apache Tomcat（可在 *http://jakarta.apache.org* 下载）。然后可以将

客户端所需的一切加入服务器的类路径之中。最简单的方式是在Servlet引擎库目录中加入 *soap.jar*，*activation.jar* 和 *mail.jar*，以及解析器。Tomcat 中的库目录就是 *lib/*，其中有应该自动载入的库。如果要支持脚本（本章未讲述，但 Apache SOAP 例子中有），还应在此目录中放入 *bsf.jar*（可以 *http://oss.software.ibm.com/developerworks/projects/bsf* 获取）和 *js.jar*（从 *http://www.mozilla.org/rhino/* 获取）。

注意：如果用的是 Xerces 和 Tomcat，应该像第十章我们讲过的那样改一下名字。将 *parser.jar* 改成 *z_parser.jar*，*jaxp.jar* 改成 *z_jaxp.jar*，并且一定要在其他解析器和 JAXP 实现之前载入 *xerces.jar* 和所含 JAXP 版本。

现在重启 Servlet 引擎，就可以编写 SOAP 服务器组件了。

路由器 Servlet 和管理客户端

除了基本功能以外，Apache SOAP 中还有一个路由器 Servlet 和管理客户端。我推荐大家也一并安装，即使一时半会儿还用不上，因为可以用来测试 SOAP 的安装情况。安装过程因 Servlet 引擎的不同而异，因此这里我们只讲 Tomcat 下的安装。但是，其他几种 Servlet 引擎的安装指示可以在 *http://xml.apache.org/soap/docs/index.html* 处找到。

Tomcat 下的安装很简单，复制 *soap-2_2/webapps* 目录中的 *soap.war* 文件，粘贴到 *$TOMCAT_HOM/webapps* 目录。这就万事大吉了！要测试安装情况的话，在 Web 浏览器中输入 *http://localhost:8080/soap/servlet/rpcrouter*。结果如图 12-2 所示。

这看似有误，实则表示一切正常。将 Web 浏览器指向管理客户端，即输入 *http://localhost:8080/soap/servlet/messagerouter*，结果也一样。

测试服务器和客户端的最后一步，是要确保我们遵循了所有的安装指示。然后执行以下 Java 类，提供 RPC 路由器 Servlet 的 URL：

```
C:\>java org.apache.soap.server.ServiceManagerClient
        http://localhost:8080/soap/servlet/rpcrouter list
Deployed Services:
```

图 12-2：RPC 路由器 Servlet

获得的结果应该是空的服务列表，如上所示。如果出现了其他信息，应参考 *http://www.apache.org/soap/docs/trouble/index.html* 处的错误列表。那里有可能遇到的各种问题。我们的结果是空列表，所以安装已经完毕，可以继续往下试验本章的其他例子。

深入实践

写任何基于 SOAP 的系统都要分三步走，我们依次讲述：

- 在 SOAP-RPC 和 SOAP 消息发送（messaging）之间抉择
- 编写或获取一项 SOAP 服务
- 编写或获取一个 SOAP 客户端

第一步要决定：是在 RPC 式调用中使用 SOAP，从而在服务器上调用远程过程，还是将 SOAP 用于消息发送，从而让客户端直接向服务器发送信息。下一节我们再谈这两种过程的具体细节。做出决定后，就需要获取或编写一项服务。当然，既然我们都是 Java 的拥趸，本章将介绍如何自己编写服务。最后，还要为服务编写客户端，并静观其变。

RPC 还是消息发送？

我们的第一项任务实际上与编程无关，而取决于设计。需要在RPC和消息发送之间进行抉择。前者RPC是上一章中的主题，我们已经很熟悉了。客户端在服务器上调用远程过程，并获得某种响应。这种情况下，SOAP的角色可以比XML-RPC更灵活，错误处理和跨网络传递复杂类型的性能都要好一些。这里的概念我们应该都了解了。由于RPC系统用SOAP编写很容易，我们以此为出发点。本章讲述如何编写RPC服务，然后编写RPC客户端，再将系统运行起来。

第二种SOAP处理方式是以消息为基础的。这种方式无需调用远程过程，只是提供信息的传输。可以想像，其功能是很强大的，而且也不用要求客户端了解服务器上的特定方法。它也能更好地模拟分布式系统，通过四处传递数据分组（这里的分组只是一般意义上的，并非计算机网络术语），使不同的系统能始终了解其他系统的情况。这比RPC方式更复杂，因此我们在打好SOAP-RPC基础之后，到下一章再与其他B2B细节一起讲。

与大多数设计问题一样，如何决策还是取决于我们自己。应根据应用程序具体情况，决定SOAP的准确角色。如果有一台服务器和一组需要执行远程任务的客户端，那么RPC可能比较合适。但是，在更大的系统中，需要交换数据甚于按请求执行特定的业务功能，SOAP的消息发送功能将是更好的选择。

RPC 服务

好了，理论学习完毕，该快速向实践前进了。还记得上一章中我们需要一个类，其方法可以远程调用。

示范代码

我们从用于服务器的一些代码开始。这些代码是带有暴露给RPC客户的方法的一些类（注2）。这次我们不用上一章中的简单类，而是写一个更复杂的例子说明SOAP

注2： 可以通过Bean Scripting Framework使用脚本，但本书限于篇幅不能赘述了。请参考O'Reilly公司出版的SOAP图书，以及在线文档 *http://xml.apache.org/soap*，了解SOAP中脚本支持的更多细节。

的作用。例 12-4 是保存 CD 目录的一个类，可以用在网上音乐店程序中。这里只介绍一个基本版本，后面再逐步增加功能。

例 12-4：CDCatalog 类

```
package javaxml2;

import java.util.Hashtable;

public class CDCatalog {

    /** 按名字保存的 CD */
    private Hashtable catalog;

    public CDCatalog() {
        catalog = new Hashtable();

        // 生成目录
        catalog.put("Nickel Creek", "Nickel Creek");
        catalog.put("Let it Fall", "Sean Watkins");
        catalog.put("Aerial Boundaries", "Michael Hedges");
        catalog.put("Taproot", "Michael Hedges");
    }

    public void addCD(String title, String artist) {
        if ((title == null) || (artist == null)) {
            throw new IllegalArgumentException("Title and artist cannot be null.");
        }
        catalog.put(title, artist);
    }

    public String getArtist(String title) {
        if (title == null) {
            throw new IllegalArgumentException("Title cannot be null.");
        }

        // 返回所要求的 CD
        return (String)catalog.get(title);
    }

    public Hashtable list() {
        return catalog;
    }
}
```

这个类可以添加新的 CD，按 CD 名搜索乐手，以及获取所有 CD。请注意 `List()` 方法返回的是 `Hashtable`，其他的就没什么可讲了。Apache SOAP 提供了 Java 类型 `Hashtable` 的自动对应，与 XML-RPC 中很像。

编译这个类，请一定保证代码输入（或者下载）的正确，注意CDCatalog类与SOAP没有直接关系。这就是说，我们可以将现成的Java类直接拿来，并通过SOAP-RPC暴露它们，这大大减少了我们向基于SOAP架构转移的工作量。

部署描述代码

Java代码写完了，现在应定义一个部署描述代码（deployment descriptor）。它要负责向SOAP服务器指定几个关键要素：

- 客户端访问SOAP服务所用的URN
- 客户端可用的方法
- 自定义类的串行化和反串行化处理方法

第一个类似于URL，对于要与SOAP服务器连接的客户是必需的。第二个正是我们需要的：一系列方法，使客户端了解自己能够做什么。它也可以告诉SOAP服务器（等会就会讲到），将接受何种请求。第三个要素是告诉SOAP服务器如何处理自定义参数的一种方式，下一节给CDCatalog类添加更复杂功能时，再讨论它。

下面来看看部署描述代码的例子，并详述每一项。例12-5是正在创建的CDCatalog服务的部署描述代码。

例12-5：CDCatalog部署描述代码

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
             id="urn:cd-catalog"
>
  <isd:provider type="java"
                scope="Application"
                methods="addCD getArtist list"
  >
    <isd:java class="javaxml2.CDCatalog" static="false" />
  </isd:provider>

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

首先，其中引用了Apache SOAP部署名字空间，然后通过id属性为服务提供了一个URN。这对于不同服务而言是惟一的，可以说明服务。这里命名服务的方式够原始的，但可以完成任务。然后，通过java元素指定了要暴露的类，包括其类名（通过class属性），并且说明要暴露的方法不是静态的（通过static属性）。

接下来，指定要使用的错误监听器实现。Apache 的 SOAP 实现提供了两种监听器，我们用的是第一个 `DOMFaultListener`。它通过一个 DOM 元素返回客户响应中任何异常和错误信息。等写客户端的时候再看它，现在先别管它。另一个错误监听器实现是`org.apache.soap.server.ExceptionFaultListener`。它通过返回给客户端的一个参数暴露错误。因为很多基于 SOAP 的程序都将使用 Java 和 DOM 等 XML API，大多数情况下使用 `DOMFaultListener` 很常见。

部署服务

现在我们已经有了部署描述代码和要暴露的方法，可以部署服务了。Apache SOAP 附带有供此目的的工具，只要我们设置好。首先，需要有服务的部署描述代码，上面讲过了。其次，需要编制 SOAP 服务器可访问的服务类。最好的方式是将上一节的服务类打包成 *jar* 文档：

```
jar cvf javaxml2.jar javaxml2/CDCatalog.class
```

将 *jar* 文件放入 *lib/* 目录（或 Servlet 引擎自动载入类的其他目录）重启 Servlet 引擎。

警告：这只是生成了类文件的暂时映像。仅仅改变 *CDCatalog.java* 文件中的代码并重新编译，并不能使 Servlet 引擎获取这些变化。每次改变代码时，都要重新打包，并复制到 *lib/* 目录，以确保服务已更新。还应重启 Servlet 引擎，确保变化已经为引擎所接受。

SOAP 服务器可访问的服务类已经有了，现在可以部署服务了，使用的是 Apache SOAP 的 `org.apache.soap.server.ServiceManager` 工具类：

```
C:\javaxml2\Ch12>java org.apache.soap.server.ServiceManagerClient
    http://localhost:8080/soap/servlet/rpcrouter deploy xml\CDCatalogDD.xml
```

第一个参数是 SOAP 服务器和 RPC 路由器 Servlet，第二个参数是要采取的动作，第三个是相关的部署描述代码。完成后，看一看服务是否已经加入：

```
(gandalf)/javaxml2/Ch12$ java org.apache.soap.server.ServiceManagerClient
    http://localhost:8080/soap/servlet/rpcrouter list
Deployed Services:
        urn:cd-catalog
        urn:AddressFetcher
        urn:xml-soap-demo-calculator
```

这里列出了服务器上的所有服务。最后可以很容易地卸载服务，只要知道名字即可：

```
C:\javaxml2\Ch12>java org.apache.soap.server.ServiceManagerClient
    http://localhost:8080/soap/servlet/rpcrouter undeploy urn:cd-catalog
```

每次更新服务代码时，必须先卸载，再部署，以确保SOAP服务器运行的是最新版本。

RPC客户端

接下来该讲客户端了。为了尽量简化，我们只写两个调用SOAP-RPC的命令行程序。猜测读者们的具体业务情况是不可能的，因此我们只集中讨论SOAP细节，帮助读者与已有软件集成。只要代码的业务部分可以工作，每个SOAP-RPC调用都有几个基本步骤：

- 创建SOAP-RPC调用
- 为自定义参数设置类型映射
- 设置要使用的SOAP服务的URI
- 指定要调用的方法
- 指定所用的编码
- 添加调用参数
- 与SOAP服务连接
- 接收和解释响应

似乎多了一点，但大部分操作都是一两行方法调用代码。也就是说，与SOAP服务通信都是小菜一碟。例12-6是`CDAdder`类的代码，可以用来在目录中添加新的CD。先自己看一看代码，然后再读我的解释。

例12-6：CDAdder类

```
package javaxml2;

import java.net.URL;
import java.util.Vector;
import org.apache.soap.Constants;
import org.apache.soap.Fault;
```

```
import org.apache.soap.SOAPException;
import org.apache.soap.rpc.Call;
import org.apache.soap.rpc.Parameter;
import org.apache.soap.rpc.Response;

public class CDAdder {

    public void add(URL url, String title, String artist)
        throws SOAPException {

        System.out.println("Adding CD titled '" + title + "' by '" +
            artist + "'");

        // 生成Call对象
        Call call = new Call();
        call.setTargetObjectURI("urn:cd-catalog");
        call.setMethodName("addCD");
        call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

        // 设置参数
        Vector params = new Vector();
        params.addElement(new Parameter("title", String.class, title, null));
        params.addElement(new Parameter("artist", String.class, artist, null));
        call.setParams(params);

        // 引用
        Response response;
        response = call.invoke(url, "");

        if (!response.generatedFault()) {
            System.out.println("Successful CD Addition.");
        } else {
            Fault fault = response.getFault();
            System.out.println("Error encountered: " + fault.getFaultString());
        }
    }

    public static void main(String[] args) {
        if (args.length != 3) {
            System.out.println("Usage: java javaxml2.CDAdder [SOAP server URL] " +
                "\"[CD Title]\" \"[Artist Name]\"");
            return;
        }

        try {
            // 要连接的SOAP服务器的URL
            URL url = new URL(args[0]);

            // 获取新CD的值
            String title = args[1];
            String artist = args[2];

            // 添加CD
            CDAdder adder = new CDAdder();
```

```
            adder.add(url, title, artist);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

此程序获取要连接的SOAP服务器的URL，以及在目录中创建和添加新CD所需的信息。然后，在`add()`方法中，代码生成了SOAP `Call`对象，其中包含所有有意思的交互。SOAP服务的目标URI以及要调用的方法都在`Call`中设置，并且均与例12-5中服务的部署描述代码匹配。接下来，设置编码，应该总是常量`.NS_URI_SOAP_ENC`，除非有其他特殊编码需求。

程序生成了一个元素为SOAP `Parameter`对象的`Vector`。每个元素都表示指定方法的参数，因为`addCD()`方法的参数是两个`String`值，这非常简单。提供参数名（用于XML和调试），参数的类，以及值即可。第四个参数是可选的编码（如果参数需要特殊编码）。如果无需特殊处理，`null`值就行了。生成的`Vector`添加到`Object`对象。

Call设置好之后，使用其`invoke()`方法。返回值是一个`org.apache.soap.Response`实例，对产生的任务问题进行查询。这几乎不用解释，因此留给读者自己了。编译好客户端代码，并按本章前面所述设置类路径，此例运行结果如下：

```
C:\javaxml2\build>java javaxml2.CDAdder
   http://localhost:8080/soap/servlet/rpcrouter
   "Riding the Midnight Train" "Doc Watson"

Adding CD titled 'Riding the Midnight Train' by 'Doc Watson'
Successful CD Addition
```

例12-7是另一个简单类`CDLister`，列出了目录中的所有CD。我们不再花更多时间了，因为它与例12-6很相似，主要是按上面所述进行了增强。

例12-7：CDLister类

```
package javaxml2;

import java.net.URL;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Vector;
import org.apache.soap.Constants;
import org.apache.soap.Fault;
import org.apache.soap.SOAPException;
```

```
import org.apache.soap.rpc.Call;
import org.apache.soap.rpc.Parameter;
import org.apache.soap.rpc.Response;

public class CDLister {

    public void list(URL url) throws SOAPException {
        System.out.println("Listing current CD catalog.");

        // 生成Call对象
        Call call = new Call();
        call.setTargetObjectURI("urn:cd-catalog");
        call.setMethodName("list");
        call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

        // 无需参数

        // 调用
        Response response;
        response = call.invoke(url, "");

        if (!response.generatedFault()) {
            Parameter returnValue = response.getReturnValue();
            Hashtable catalog = (Hashtable)returnValue.getValue();
            Enumeration e = catalog.keys();
            while (e.hasMoreElements()) {
                String title = (String)e.nextElement();
                String artist = (String)catalog.get(title);
                System.out.println("  '" + title + "' by " + artist);
            }
        } else {
            Fault fault = response.getFault();
            System.out.println("Error encountered: " + fault.getFaultString());
        }
    }

    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Usage: java javaxml2.CDAdder [SOAP server URL]");
            return;
        }

        try {
            // 要连接的SOAP服务器的URL
            URL url = new URL(args[0]);

            // 列出CD
            CDLister lister = new CDLister();
            lister.list(url);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

这个类与 CDAdder 的惟一区别是 Response 对象有返回值（list()方法的 Hashtable）。这是以 Parameter 对象的形式返回的，允许客户端检查其编码并提取真正的方法返回值。完成后，客户端可以使用返回值，就像其他 Java 对象一样。此例只是遍历 CD 目录，并列出所有 CD。现在可以看看其运行情况：

```
C:\javaxml2\build>java javaxml2.CDLister
    http://localhost:8080/soap/servlet/rpcrouter
Listing current CD catalog.
  'Riding the Midnight Train' by Doc Watson
  'Taproot' by Michael Hedges
  'Nickel Creek' by Nickel Creek
  'Let it Fall' by Sean Watkins
  'Aerial Boundaries' by Michael Hedges
```

这就是 SOAP 中的基本 RPC 功能了。但我们还要继续深入，谈几个更复杂的主题。

继续深入

虽然现在 XML-RPC 能做的事，我们用 SOAP 也能完成了，但 SOAP 的能力还不止于此。正如本章开头说过的，SOAP 带来的两个重要方面就是能够容易地使用自定义参数，以及更高级的错误管理。本节中，我们就讲这两个问题。

自定义参数类型

前面 CD 目录最局限的地方（至少对目前而言）就是它只存储给定 CD 的名字和乐手。用一个对象（或对象集合）来表示带有名字、乐手、标号、歌曲列表、风格等等信息的 CD 当然更符合实际。我们用不着创建完整的结构，但是会从名字和乐手转移到有名字、乐手和标签的 CD 对象。这个对象要从客户端传到服务器，然后再传回，以说明 SOAP 处理自定义类型的方法。新的类如例 12-8 所示。

例 12-8：CD 类

```
package javaxml2;

public class CD {

    /** CD 名 */
    private String title;

    /** CD 乐手 */
    private String artist;
```

```
    /** CD的标签 */
    private String label;

    public CD() {
        // 默认构造方法
    }

    public CD(String title, String artist, String label) {
        this.title = title;
        this.artist = artist;
        this.label = label;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getArtist() {
        return artist;
    }

    public void setArtist(String artist) {
        this.artist = artist;
    }

    public String getLabel() {
        return label;
    }

    public void setLabel(String label) {
        this.label = label;
    }

    public String toString() {
        return "'" + title + "' by " + artist + ", on " +
            label;
    }
}
```

这要求`CDCatalog`类也得相应改变。例12-9所示为修改后的类，已为使用新的CD类做了改变，改变部分用黑体显示。

例12-9：更新后的CDCatalog类

```
package javaxml2;

import java.util.Hashtable;
```

```
public class CDCatalog {

    /** 按名字保存的CD */
    private Hashtable catalog;

    public CDCatalog() {
        catalog = new Hashtable();

        // 生成目录
        addCD(new CD("Nickel Creek", "Nickel Creek", "Sugar Hill"));
        addCD(new CD("Let it Fall", "Sean Watkins", "Sugar Hill"));
        addCD(new CD("Aerial Boundaries", "Michael Hedges", "Windham Hill"));
        addCD(new CD("Taproot", "Michael Hedges", "Windham Hill"));
    }

    public void addCD(CD cd) {
        if (cd == null) {
            throw new IllegalArgumentException("The CD object cannot be null.");
        }
        catalog.put(cd.getTitle(), cd);
    }

    public CD getCD(String title) {
        if (title == null) {
            throw new IllegalArgumentException("Title cannot be null.");
        }

       // 返回所请求的CD
        return (CD)catalog.get(title);
    }

    public Hashtable list() {
        return catalog;
    }
}
```

除了比较明显的之外，还把老的`getArtist(String title)`方法改成了`getCD(String title)`，使返回值变成`CD`对象。这意味着SOAP服务器需要串行化和反串行化新的类，因此客户端也要更新。我们首先看看怎样更新部署描述符，它详细地说明了与这个自定义类型相关的串行化问题。将下述几行加入`CDCatalog`的部署描述符，并将方法名改为与`CDCatalog`类对应：

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
             id="urn:cd-catalog"
>
  <isd:provider type="java"
                scope="Application"
                methods="addCD getCD list"
  >
```

```
    <isd:java class="javaxml2.CDCatalog" static="false" />
  </isd:provider>

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

  <isd:mappings>
    <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
             xmlns:x="urn:cd-catalog-demo" qname="x:cd"
             javaType="javaxml2.CD"
             java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
             xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
  </isd:mappings>
</isd:service>
```

新元素mappings指定了SOAP处理CD类这样的自定义参数的方式。首先，为每个自定义参数类型定义一个map元素。对于encodingStyle属性而言，至少到Apache SOAP 2.2，提供的值都应该是*http://schemas.xmlsoap.org/soap/encoding/*，这是目前支持的惟一编码。还需要为自定义类型提供名字空间和带名字空间前缀的类名。这里我们使用了简单的名字空间和前缀“ x ”。然后给javaType属性提供实际的Java类名，如这里的javaxml2.CD。最后再来看看java2XMLClassName和xml2JavaClassName属性，它们分别指定了从Java到XML和从XML到Java转换的类。这里使用了极方便的BeanSerializer类，也是Apache SOAP提供的。如果我们的自定义参数是JavaBean格式，这个串行化和反串行化类将免除我们自己编写之苦。还应该有一个带默认构造方法的类（请记住我们已在CD类中定义了一个空的无参数构造方法），并通过setXXX和getXXX式的方法暴露类中的数据。因此CD类完全符合需要，所以BeanSerializer工作得极好。

注意：CD类遵守JavaBean规范并非偶然。大多数数据类都很容易符合这种格式，而且我可不想自己编写串行化和反串行化类。编写它们可是件苦事（虽然不是太难，但容易弄混），我建议大家尽量在自定义参数中采用Java Bean规范。许多情况下，只需要有一个默认构造方法（无参数）在类里就行了。

现在重新生成服务的*jar*文件。然后重新部署：

```
(gandalf)/javaxml2/Ch12$ java org.apache.soap.server.ServiceManagerClient
    http://localhost:8080/soap/servlet/rpcrouter xml/CDCatalogDD.xml
```

警告：如果Servlet引擎一直在运行，且一直部署了服务，这时需要重启引擎，以激活SOAP服务的新类，并重新部署服务。

现在剩下的就是修改客户端程序，让它使用新的类和方法了。例 12-10 是更新后的客户类 CDAdder。改动部分用黑体显示。

例 12-10：更新后的 CDAdder 类

```
package javaxml2;

import java.net.URL;
import java.util.Vector;
import org.apache.soap.Constants;
import org.apache.soap.Fault;
import org.apache.soap.SOAPException;
import org.apache.soap.encoding.SOAPMappingRegistry;
import org.apache.soap.encoding.soapenc.BeanSerializer;
import org.apache.soap.rpc.Call;
import org.apache.soap.rpc.Parameter;
import org.apache.soap.rpc.Response;
import org.apache.soap.util.xml.QName;

public class CDAdder {

    public void add(URL url, String title, String artist, String label)
        throws SOAPException {

        System.out.println("Adding CD titled '" + title + "' by '" +
            artist + "', on the label " + label);
        CD cd = new CD(title, artist, label);

        // 映射此类型，使 SOAP 可以使用
        SOAPMappingRegistry registry = new SOAPMappingRegistry();
        BeanSerializer serializer = new BeanSerializer();
        registry.mapTypes(Constants.NS_URI_SOAP_ENC,
            new QName("urn:cd-catalog-demo", "cd"),
            CD.class, serializer, serializer);

        // 生成 Call 对象
        Call call = new Call();
        call.setSOAPMappingRegistry(registry);
        call.setTargetObjectURI("urn:cd-catalog");
        call.setMethodName("addCD");
        call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

        // 设置参数
        Vector params = new Vector();
        params.addElement(new Parameter("cd", CD.class, cd, null));
        call.setParams(params);

        // 调用
        Response response;
        response = call.invoke(url, "");

        if (!response.generatedFault()) {
```

```
            System.out.println("Successful CD Addition.");
        } else {
            Fault fault = response.getFault();
            System.out.println("Error encountered: " + fault.getFaultString());
        }
    }

    public static void main(String[] args) {
        if (args.length != 4) {
            System.out.println("Usage: java javaxml2.CDAdder [SOAP server URL] " +
                "\"[CD Title]\" \"[Artist Name]\" \"[CD Label]\"");
            return;
        }

        try {
            // SOAP 服务器所连接的 URL
            URL url = new URL(args[0]);

            // 获取新 CD 的值
            String title = args[1];
            String artist = args[2];
            String label = args[3];

            // 添加 CD
            CDAdder adder = new CDAdder();
            adder.add(url, title, artist, label);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

惟一值得注意的改变是对 CD 类映射的处理：

```
            //映射此类型，使 SOAP 可以使用
            SOAPMappingRegistry registry = new SOAPMappingRegistry();
            BeanSerializer serializer = new BeanSerializer();
            registry.mapTypes(Constants.NS_URI_SOAP_ENC,
                new QName("urn:cd-catalog-demo", "cd"),
                CD.class, serializer, serializer);
```

这样自定义参数才能编码并经过网络发送。我们已经讨论了怎样使用BeanSerializer类处理 JavaBean 格式的参数，如 CD 类。为了向服务器指定这一点，我们使用了部署描述符。但是，现在要让客户端知道使用串行化和反串行化类，这正是SOAPMappingRegistry类的作用。mapTypes()方法以一个编码字符串（这里使用NS_URI_SOAP_ENC 常量最好），以及特定串行化作用的参数类型的信息。首先，提供QName。这正是前面部署描述符中要使用奇怪的名字空间方式的原因。这里需要指定相同的 URN，以及元素的本地名（这里的 CD），然后这个类的 Java Class 对

象被串行化（CD.class），最后类的实例被串行化和反串行化。在BeanSerializer中，同一实例承担了两项工作。注册表中都设置好了之后，通过setSOAPMapping-Registry()方法告知Call对象。

现在可以像前面那样运行这个类，加上CD标签，一切顺利运行：

```
C:\javaxml2\build>java javaxml2.CDAdder
    http://localhost:8080/soap/servlet/rpcrouter
    "Tony Rice" "Manzanita" "Sugar Hill"
Adding CD titled 'Tony Rice' by 'Manzanita', on the label Sugar Hill
Successful CD Addition.
```

以同样方式修改CDLister类的任务留给读者做练习，可以从网上下载。

注意：有人可能会认为，由于CDLister类不能直接处理CD对象（list()方法的返回值是一个Hashtable），根本不需要改动。但是，返回的Hashtable包含CD对象的实例。如果SOAP不知道如何反串行化它们，客户端就会报错。所以，必须指定Call对象的SOAPMappingRegistry实例。

更好的错误处理

我们已经讨论了自定义对象，进行RPC调用，下面再谈一个不那么有趣的话题：错误处理。在任何网络事务中，许多东西都会出错：服务不能运行，服务器出错，对象找不到，类不见了，等等。到目前为止，我们只是用fault.getString()方法来报错。但这并不理想。我们在实际中看一看，将CDCatalog构造方法中的一行注释掉：

```
public CDCatalog() {
    //catalog = new Hashtable();

    // 生成目录
    addCD(new CD("Nickel Creek", "Nickel Creek", "Sugar Hill"));
    addCD(new CD("Let it Fall", "Sean Watkins", "Sugar Hill"));
    addCD(new CD("Aerial Boundaries", "Michael Hedges", "Windham Hill"));
    addCD(new CD("Taproot", "Michael Hedges", "Windham Hill"));
}
```

重新编译，重启Servlet引擎并重新部署。结果，在类构造方法试图向未初始化的Hashtable添加CD时，NullPointerException抛出了。运行客户端程序时，告诉我们出错了，但缺乏有用的信息：

```
(gandalf)/javaxml2/build$ java javaxml2.CDLister
    http://localhost:8080/soap/servlet/rpcrouter
Listing current CD catalog.
Error encountered: Unable to resolve target object: null
```

这对于查找问题的根源可没什么用。但是，其实在框架中能更好地进行错误处理。还记得我们指定为`faultListener`元素值的`DOMFaultListener`吗？这正是它的用武之地。发生问题时所返回的`Fault`对象包含带有详细错误信息的DOM `org.w3c.dom.Element`。首先，在客户端源代码中加一条`java.util.Iterator`的导入语句：

```
import java.net.URL;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Vector;
import org.apache.soap.Constants;
import org.apache.soap.Fault;
import org.apache.soap.SOAPException;
import org.apache.soap.encoding.SOAPMappingRegistry;
import org.apache.soap.encoding.soapenc.BeanSerializer;
import org.apache.soap.rpc.Call;
import org.apache.soap.rpc.Parameter;
import org.apache.soap.rpc.Response;
import org.apache.soap.util.xml.QName;
```

然后，对`List()`方法中的错误处理做如下改动：

```
        if (!response.generatedFault()) {
            Parameter returnValue = response.getReturnValue();
            Hashtable catalog = (Hashtable)returnValue.getValue();
            Enumeration e = catalog.keys();
            while (e.hasMoreElements()) {
                String title = (String)e.nextElement();
                CD cd = (CD)catalog.get(title);
                System.out.println("  '" + cd.getTitle() + "' by " +
cd.getArtist() +
                    " on the label " + cd.getLabel());
             }
        } else {
            Fault fault = response.getFault();
            System.out.println("Error encountered: " + fault.getFaultString());

            Vector entries = fault.getDetailEntries();
            for (Iterator i = entries.iterator(); i.hasNext(); ) {
                org.w3c.dom.Element entry = (org.w3c.dom.Element)i.next();
                System.out.println(entry.getFirstChild().getNodeValue());
             }
         }
```

通过使用getDetailEntries()方法，可以访问SOAP服务和服务器提供的有关问题的原始数据。代码遍历这些数据（通常只有一个元素，但要小心），并获取每个项中包含的DOM Element。实际上，是在处理如下XML：

```
<SOAP-ENV:Fault>
  <faultcode>SOAP-ENV:Server.BadTargetObjectURI</faultcode>
  <faultstring>Unable to resolve target object: null</faultstring>
  <stacktrace>Here's what we want!</stackTrace>
</SOAP-ENV:Fault>
```

换言之，Fault对象使我们得以访问处理错误的部分SOAP封装。而且，Apache SOAP在出错时提供Java堆栈轨迹，并提供排错所需的详细信息。通过获取stackTrace元素，并输出Element的Text节点值，客户端现在可以输出来自服务器的堆栈轨迹。改动后编译，并再次运行客户端程序，输出如下：

```
C:\javaxml2\build>java javaxml2.CDLister http://localhost:8080/soap/servlet/rpcr
outer
Listing current CD catalog.
Error encountered: Unable to resolve target object: null
java.lang.NullPointerException
        at javaxml2.CDCatalog.addCD(CDCatalog.java:24)
        at javaxml2.CDCatalog.<init>(CDCatalog.java:14)
        at java.lang.Class.newInstance0(Native Method)
        at java.lang.Class.newInstance(Class.java:237)
```

这还只是管中窥豹，但已经可以看到表明NullPointerException发生的重要信息了，甚至还有服务器类中出问题的行号。只是小小的改动，却得到了处理错误的强大得多的方式。这下我们可以对服务器类中的缺陷追本溯源了。噢，继续向前之前，别忘了将CDCatalog类改回去，可不要留下错误。

下章预告

下一章是本章主题的直接延续。XML正在成为B2B技术的基本，而SOAP是其中关键。下一章将介绍两个重要技术，UDDI和WSDL。如果对此还没有什么概念，那么你真是碰巧了。在此可以学到它们是如何共同构成Web服务的架构骨干的。准备好，让我们揭开Web服务和P2P的面纱。