

## WebService 大讲堂之 Axis2

- <http://www.blogjava.net/nokiaguy> 原创
- Fallseir 2010-11-26 整理

## 目录

WebService 大讲堂之 Axis2 (1): 用 POJO 实现 0 配置的 WebService .....	2
一、Axis2 的下载和安装 .....	2
二、编写和发布 WebService .....	3
三、用 Java 实现调用 WebService 的客户端程序.....	5
四、用 wsdl2java 简化客户端的编写 .....	7
五、使用 C#调用 WebService.....	8
WebService 大讲堂之 Axis2 (2): 复合类型数据的传递.....	9
一、实现服务端代码 .....	9
二、实现 DataForm 类 .....	10
三、发布 WebService.....	11
四、使用 Java 编写调用 WebService 的客户端代码.....	11
五、使用 C#编写调用 WebService 的客户端代码.....	13
WebService 大讲堂之 Axis2(3): 使用 services.xml 文件发布 WebService.....	14
<b>WebService 大讲堂之 Axis2(4): 二进制文件传输 .....</b>	<b>17</b>
WebService 大讲堂之 Axis2(5): 会话 (Session) 管理.....	20
WebService 大讲堂之 Axis2(6): 跨服务会话(Session)管理 .....	22
WebService 大讲堂之 Axis2(7): 将 Spring 的装配 JavaBean 发布成 WebService .....	26
WebService 大讲堂之 Axis2(8): 异步调用 WebService.....	28
WebService 大讲堂之 Axis2(9): 编写 Axis2 模块 (Module) .....	32
WebService 大讲堂之 Axis2(10): 使用 soapmonitor 模块监视 soap 请求与响应消息 .....	37

## WebService 大讲堂之 Axis2 (1): 用 POJO 实现 0 配置的 WebService

- <http://www.blogjava.net/nokiaguy/archive/2009/nokiaguy/archive/2009/nokiaguy/archive/2009/01/02/249556.html>

本文为 <http://www.blogjava.net/nokiaguy> 原创，如需转载，请注明作者和出处，谢谢！

Axis2 是一套崭新的 WebService 引擎，该版本是对 Axis1.x 重新设计的产物。Axis2 不仅支持 SOAP1.1 和 SOAP1.2，还集成了非常流行的 REST WebService，同时还支持 Spring、JSON 等技术。这些都将在后面的系列教程中讲解。在本文中主要介绍了如何使用 Axis2 开发一个不需要任何配置文件的 WebService，并在客户端使用 Java 和 C# 调用这个 WebService。

### 一、Axis2 的下载和安装

读者可以从如下的网址下载 Axis2 的最新版本：

<http://ws.apache.org/axis2/>

在本文使用了目前 Axis2 的最新版本 1.4.1。读者可以下载如下两个 zip 包：

axis2-1.4.1-bin.zip

axis2-1.4.1-war.zip

其中 axis2-1.4.1-bin.zip 文件中包含了 Axis2 中所有的 jar 文件，axis2-1.4.1-war.zip 文件用于将 WebService 发布到 Web 容器中。

将 axis2-1.4.1-war.zip 文件解压到相应的目录，将目录中的 axis2.war 文件放到 <Tomcat 安装目录>\webapps 目录中（本文使用的 Tomcat 的版本是 6.x），并启动 Tomcat。

在浏览器地址栏中输入如下的 URL：

<http://localhost:8080/axis2/>

如果在浏览器中显示出如图 1 所示的页面，则表示 Axis2 安装成功。

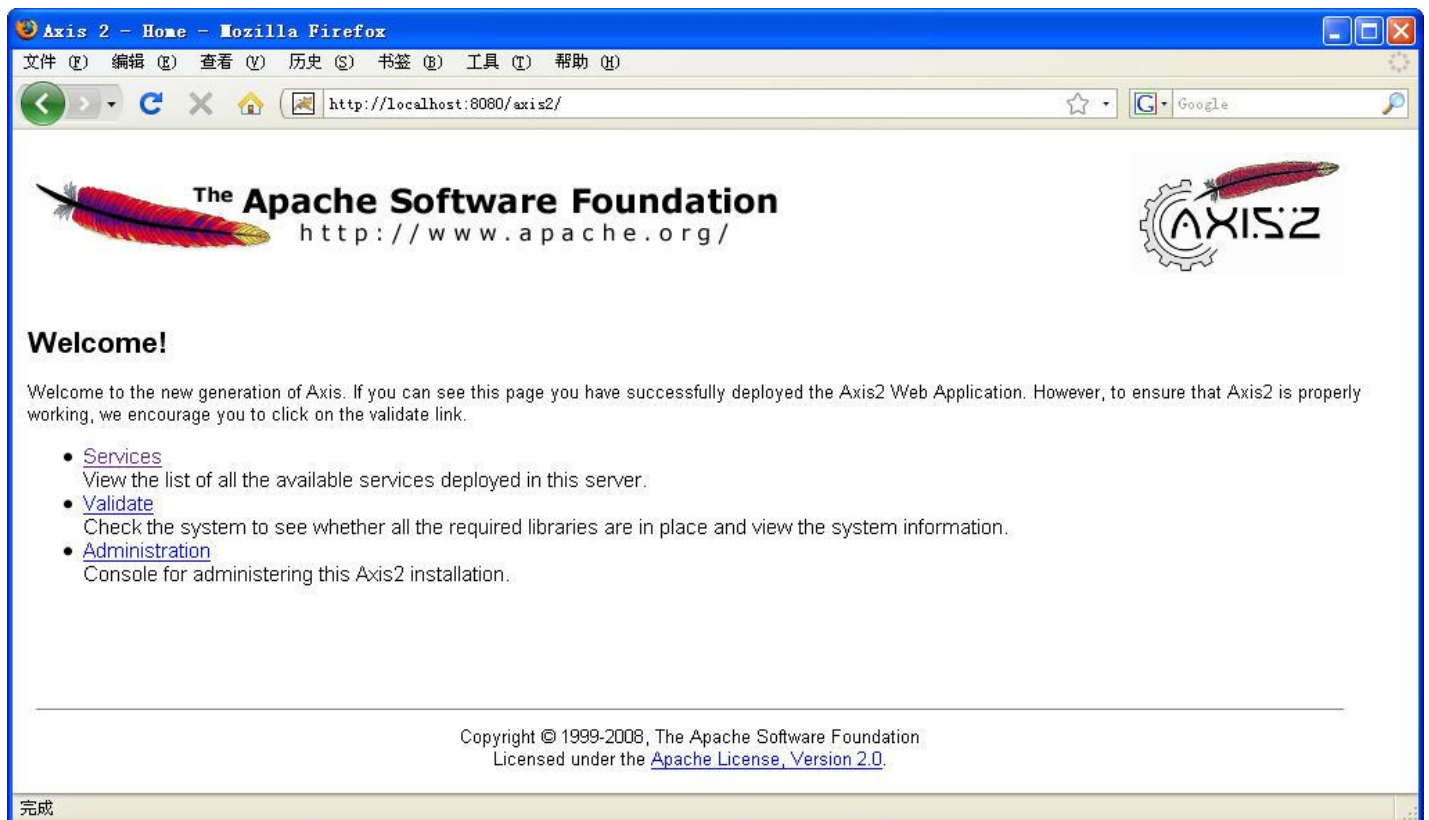


图 1

## 二、编写和发布 WebService

对于用 Java 实现的服务程序给人的印象就是需要进行大量的配置，不过这一点在 Axis2 中将被终结。在 Axis2 中不需要进行任何的配置，就可以直接将一个简单的 POJO 发布成 WebService。其中 POJO 中所有的 public 方法将被发布成 WebService 方法。

下面我们来实现一个简单的 POJO，代码如下：

```
public class SimpleService
{
    public String getGreeting(String name)
    {
        return "你好 " + name;
    }
    public int getPrice()
    {
        return new java.util.Random().nextInt(1000);
    }
}
```

在 SimpleService 类中有两个方法，由于这两个方法都是 public 方法，因此，它们都将作为 WebService 方法被发布。

编译 SimpleService 类后，将 SimpleService.class 文件放到 <Tomcat 安装目录>\webapps\axis2\WEB-INF\pojo 目录中（如果没有 pojo 目录，则建立该目录）。现在我们已经成功将 SimpleService 类发布成了 WebService。在浏览器地址栏中输入如下的 URL：

<http://localhost:8080/axis2/services/listServices>

这时当前页面将显示所有在 Axis2 中发布的 WebService，如图 2 所示。

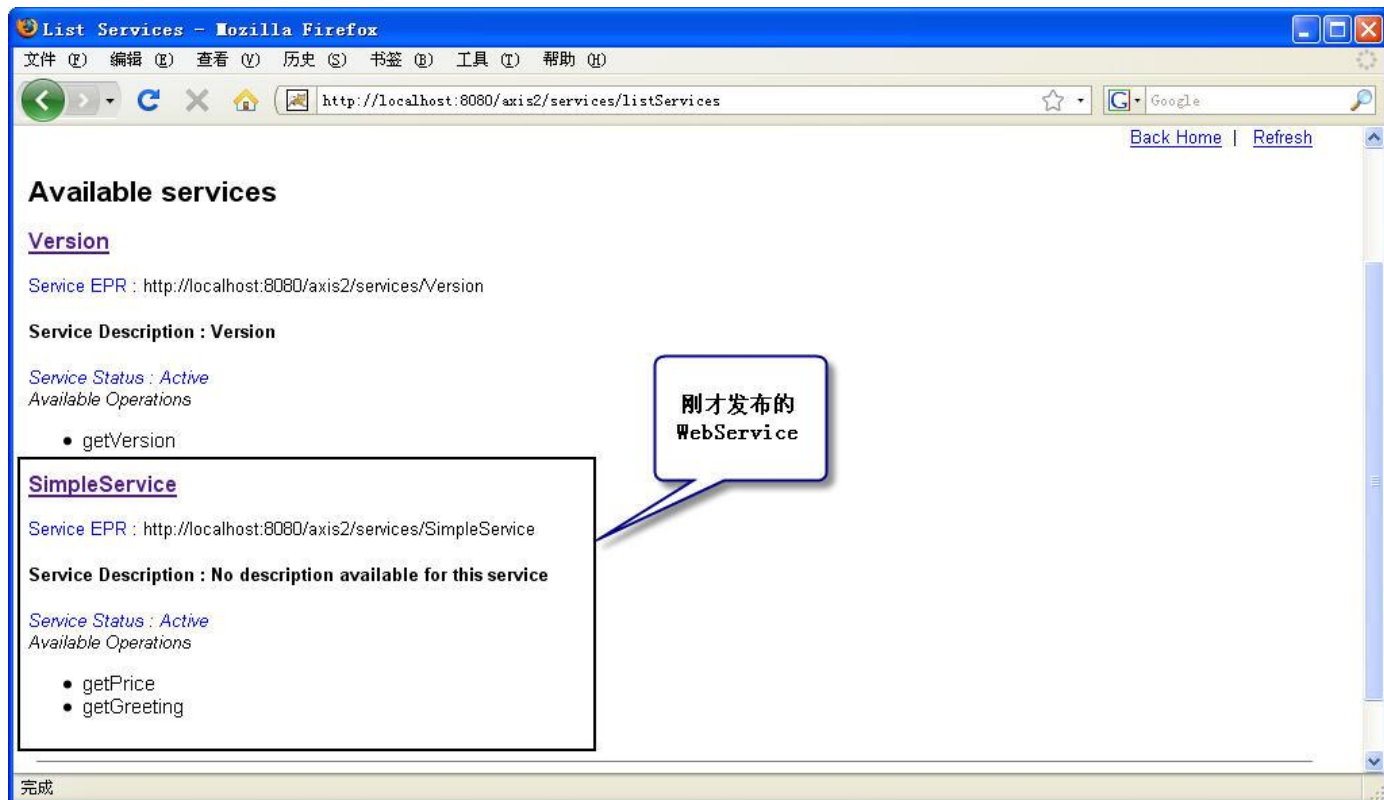


图 2

在浏览器地址栏中输入如下的两个 URL 来分别测试 `getGreeting` 和 `getPrice` 方法:

`http://localhost:8080/axis2/services/SimpleService/getGreeting?name=bill`

`http://localhost:8080/axis2/services/SimpleService/getPrice`

图 3 和图 4 分别显示了 `getGreeting` 和 `getPrice` 方法的测试结果。

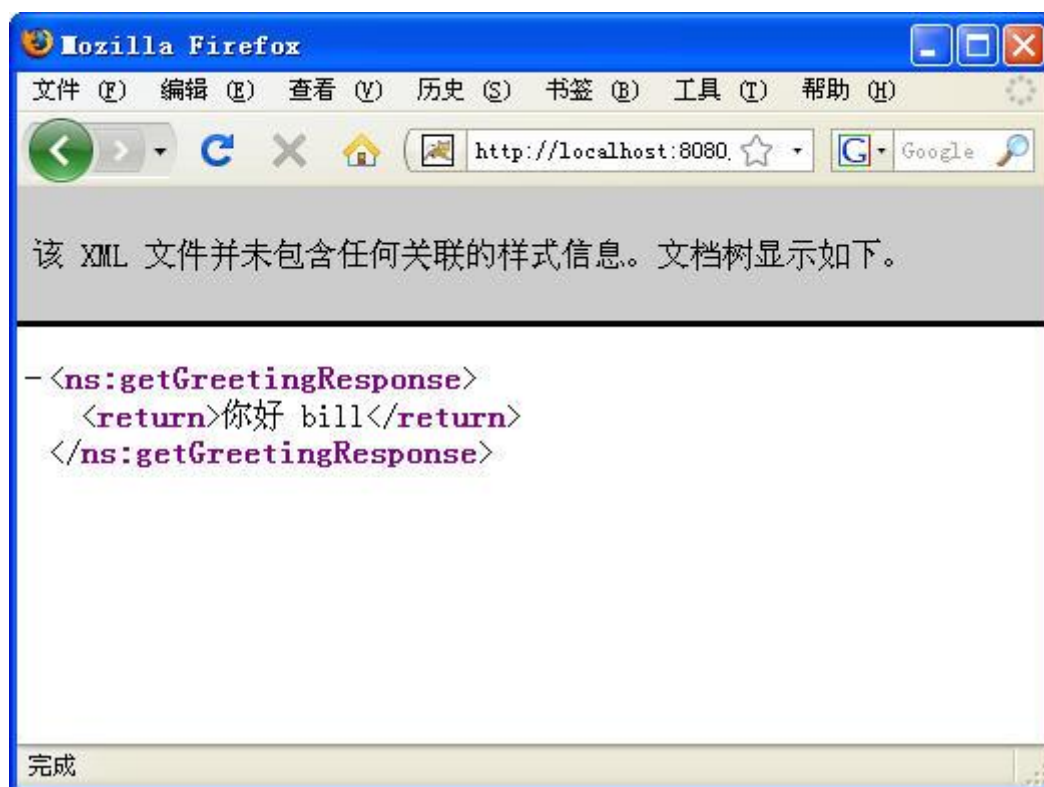


图 3 `getGreeting` 方法的测试结果

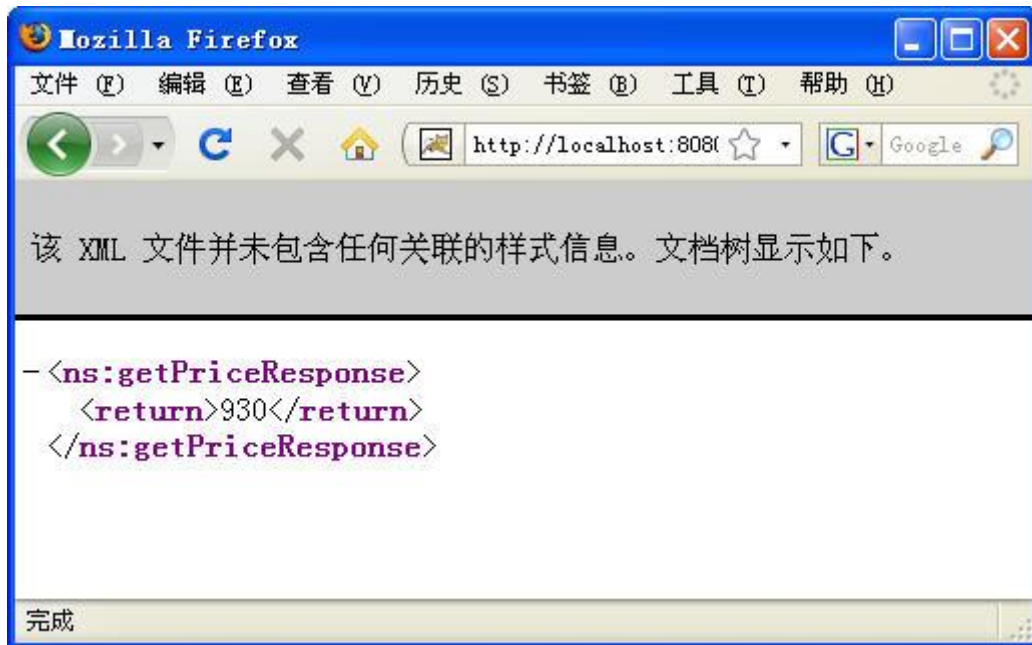


图 4 getPrice 方法的测试结果

在编写、发布和测试 0 配置的 WebService 时应注意如下几点：

1. POJO 类不能使用 package 关键字声明包。

2. Axis2 在默认情况下可以热发布 WebService，也就是说，将 WebService 的.class 文件复制到 pojo 目录中时，Tomcat 不需要重新启动就可以自动发布 WebService。如果想取消 Axis2 的热发布功能，可以打开<Tomcat 安装目录>\webapps\axis2\WEB-INF\conf\axis2.xml，找到如下的配置代码：

```
<parameter name="hotdeployment">true</parameter>
```

将 true 改为 false 即可。要注意的是，Axis2 在默认情况下虽然是热发布，但并不是热更新，也就是说，一旦成功发布了 WebService，再想更新该 WebService，就必须重启 Tomcat。这对于开发人员调试 WebService 非常不方便，因此，在开发 WebService 时，可以将 Axis2 设为热更新。在 axis2.xml 文件中找到<parameter name="hotupdate">>false</parameter>，将 false 改为 true 即可。

3. 在浏览器中测试 WebService 时，如果 WebService 方法有参数，需要使用 URL 的请求参数来指定该 WebService 方法参数的值，请求参数名与方法参数名要一致，例如，要测试 getGreeting 方法，请求参数名应为 name，如上面的 URL 所示。

4. 发布 WebService 的 pojo 目录只是默认的，如果读者想在其他的目录发布 WebService，可以打开 axis2.xml 文件，并在<axisconfig>元素中添加如下的子元素：

```
<deployer extension=".class" directory="my" class="org.apache.axis2.deployment.POJODeployer"/>
```

上面的配置允许在<Tomcat 安装目录>\webapps\axis2\WEB-INF\my 目录中发布 WebService。例如，将本例中的 SimpleService.class 复制到 my 目录中也可以成功发布（但要删除 pojo 目录中的 SimpleService.class，否则 WebService 会重名）。

### 三、用 Java 实现调用 WebService 的客户端程序

WebService 是为程序服务的，只在浏览器中访问 WebService 是没有意义的。因此，在本节使用 Java 实现了一个控制台程序来调用上一节发布的 WebService。调用 WebService 的客户端代码如下：

```

package client;

import javax.xml.namespace.QName;
import org.apache.axis2.addressing.EndpointReference;
import org.apache.axis2.client.Options;
import org.apache.axis2.rpc.client.RPCServiceClient;

public class RPCClient
{
    public static void main(String[] args) throws Exception
    {
        // 使用 RPC 方式调用 WebService
        RPCServiceClient serviceClient = new RPCServiceClient();
        Options options = serviceClient.getOptions();
        // 指定调用 WebService 的 URL
        EndpointReference targetEPR = new EndpointReference(
            "http://localhost:8080/axis2/services/SimpleService");
        options.setTo(targetEPR);
        // 指定 getGreeting 方法的参数值
        Object[] opAddEntryArgs = new Object[] {"超人"};
        // 指定 getGreeting 方法返回值的数据类型的 Class 对象
        Class[] classes = new Class[] {String.class};
        // 指定要调用的 getGreeting 方法及 WSDL 文件的命名空间
        QName opAddEntry = new QName("http://ws.apache.org/axis2", "getGreeting");
        // 调用 getGreeting 方法并输出该方法的返回值
        System.out.println(serviceClient.invokeBlocking(opAddEntry, opAddEntryArgs, classes)[0]);
        // 下面是调用 getPrice 方法的代码，这些代码与调用 getGreeting 方法的代码类似
        classes = new Class[] {int.class};
        opAddEntry = new QName("http://ws.apache.org/axis2", "getPrice");
        System.out.println(serviceClient.invokeBlocking(opAddEntry, new Object[] {}, classes)[0]);
    }
}

```

运行上面的程序后，将在控制台输出如下的信息：

```

你好 超人
443

```

在编写客户端代码时应注意如下几点：

1. 客户端代码需要引用很多 Axis2 的 jar 包，如果读者不太清楚要引用哪个 jar 包，可以在 Eclipse 的工程中引用 Axis2 发行包的 lib 目录中的所有 jar 包。
2. 在本例中使用了 RPCServiceClient 类的 invokeBlocking 方法调用了 WebService 中的方法。invokeBlocking 方法有三个参数，其中第一个参数的类型是 QName 对象，表示要调用的方法名；第二个参数表示要调用的 WebService 方法的参数值，参数类型为 Object[]；第三个参数表示 WebService 方法的返回值类型的 Class 对象，参数类型为 Class[]。当方法没有参数时，invokeBlocking 方法的第二个参数值不能是 null，而要使用 new Object[] {}。
3. 如果被调用的 WebService 方法没有返回值，应使用 RPCServiceClient 类的 invokeRobust 方法，该方法只有两个参数，它们的含义与 invokeBlocking 方法的前两个参数的含义相同。



4. 在创建 QName 对象时, QName 类的构造方法的第一个参数表示 WSDL 文件的命名空间名, 也就是 <wsdl:definitions> 元素的 targetNamespace 属性值, 下面是 SimpleService 类生成的 WSDL 文件的代码片段:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:ns1="http://org.apache.axis2/xsd"
xmlns:ns="http://ws.apache.org/axis2" xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
targetNamespace="http://ws.apache.org/axis2">
  <wsdl:types>
    ...
  </wsdl:types>
  ...
</wsdl:definitions>
```

#### 四、用 wsdl2java 简化客户端的编写

也许有很多读者会说“有没有搞错啊, 只调用两个 WebService 方法用要写这么多代码, 太麻烦了”。

不过幸好 Axis2 提供了一个 wsdl2java.bat 命令可以根据 WSDL 文件自动产生调用 WebService 的代码。wsdl2java.bat 命令可以在 <Axis2 安装目录>\bin 目录中找到。在使用 wsdl2java.bat 命令之前需要设置 AXIS2\_HOME 环境变量, 该变量值是 <Axis2 安装目录>。

在 Windows 控制台输出如下的命令行来生成调用 WebService 的代码:

```
%AXIS2_HOME%\bin\wsdl2java -uri http://localhost:8080/axis2/services/SimpleService?wsdl -p client
-s -o stub
```

其中 -url 参数指定了 wsdl 文件的路径, 可以是本地路径, 也可以是网络路径。-p 参数指定了生成的 Java 类的包名, -o 参数指定了生成的一系列文件保存的根目录。在执行完上面的命令后, 读者就会发现在当前目录下多了个 stub 目录, 在 "stub\src\client" 目录可以找到一个 SimpleServiceStub.java 文件, 该文件复杂调用 WebService, 读者可以在程序中直接使用这个类, 代码如下:

```
package client;

import javax.xml.namespace.QName;
import org.apache.axis2.addressing.EndpointReference;
import org.apache.axis2.client.Options;
import org.apache.axis2.rpc.client.RPCServiceClient;

public class StubClient
{
    public static void main(String[] args) throws Exception
    {
        SimpleServiceStub stub = new SimpleServiceStub();
```

```

SimpleServiceStub.GetGreeting gg = new SimpleServiceStub.GetGreeting();
gg.setName("比尔");
System.out.println( stub.getGreeting(gg).get_return());
System.out.println(stub.getPrice().get_return());
}
}

```

上面的代码大大简化了调用 **WebService** 的步骤，并使代码更加简洁。但要注意的是，`wsdl2java.bat` 命令生成的 **Stub** 类将 **WebService** 方法的参数都封装在了相应的类中，类名为方法名，例如，`getGreeting` 方法的参数都封装在了 `GetGreeting` 类中，要想调用 `getGreeting` 方法，必须先创建 `GetGreeting` 类的对象实例。

## 五、使用 C#调用 WebService

从理论上说，**WebService** 可以被任何支持 **SOAP** 协议的语言调用。在 **Visual Studio** 中使用 **C#**调用 **WebService** 是在所有语言中最容易实现的（**VB.net** 的调用方法类似，也同样很简单）。

新建一个 **Visual Studio** 工程，并在引用 **Web** 服务的对话框中输入如下的 **URL**，并输入 **Web** 引用名为“**WebService**”：

`http://localhost:8080/axis2/services/SimpleService?wsdl`

然后引用 **Web** 服务的对话框就会显示该 **WebService** 中的所有的方法，如图 5 所示。

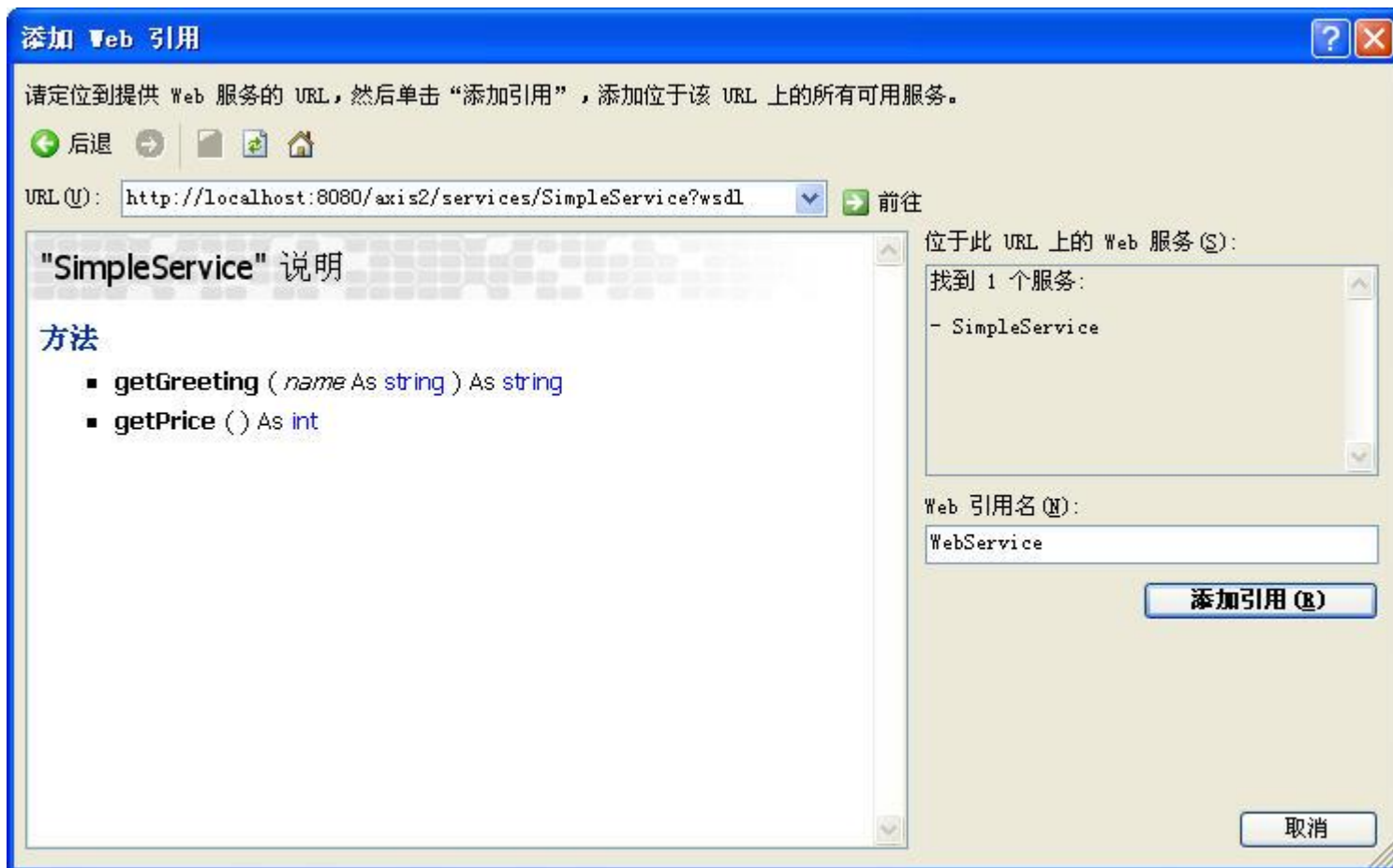


图 5

在完成上面的工作后，只需要如下三行 **C#**代码就可以调用 `getGreeting` 和 `getPrice` 方法，并显示这两个方法的返回值：



```
WebService.SimpleService simpleService = new WSC.WebService.SimpleService();
MessageBox.Show( simpleService.getGreeting("比尔"));
MessageBox.Show(simpleService.getPrice().@return.ToString());
```

在.net 解析 WSDL 文件时直接将 getGreeting 方法的参数映射为 String 类型，因此，可以直接进行传值。

从上面的调用过程可以看出，添加 Web 引用的过程就相当于在 Java 中调用 wsdl2java.bat 自动生成 stub 类的过程。只是在调用 stub 类时与 C#有一定的区别，但从总体上来说，都大大简化了调用 WebService 的过程。

## [WebService 大讲堂之 Axis2（2）：复合类型数据的传递](#)

在实际的应用中，不仅需要使用 WebService 来传递简单类型的数据，有时也需要传递更复杂的数据，这些数据可以被称为复合类型的数据。数组与类（接口）是比较常用的复合类型。在 Axis2 中可以直接使用将 WebService 方法的参数或返回值类型声明成数组或类（接口）。但要注意，在定义数组类型时只能使用一维数组，如果想传递多维数组，可以使用分隔符进行分隔，如下面的代码所示：

```
String[] strArray = new String[]{ "自行车,飞机,火箭","中国,美国,德国", "超人,蜘蛛侠,钢铁侠" } ;
```

上面的代码可以看作是一个 3\*3 的二维数组。

在传递类的对象实例时，除了直接将数组类型声明成相应的类或接口，也可以将对象实例进行序列化，也就是说，将一个对象实例转换成字节数组进行传递，然后接收方再进行反序列化，还原这个对象实例。

下面的示例代码演示了如何传递数组与类（接口）类型的数据，并演示如何使用字节数组上传图像。本示例的客户端代码使用 Java 和 C#编写。要完成这个例子需要如下几步：

### 一、实现服务端代码

ComplexTypeService 是一个 WebService 类，该类的代码如下：

```
import java.io.FileOutputStream;
import data.DataForm;

public class ComplexTypeService
{
    // 上传图像，imageByte 参数表示上传图像文件的字节，
    // length 参数表示图像文件的字节长度（该参数值可能小于 imageByte 的数组长度）
    public boolean uploadImageWithByte(byte[] imageByte, int length)
    {
        FileOutputStream fos = null;
        try
        {
            // 将上传的图像保存在 D 盘的 test1.jpg 文件中
            fos = new FileOutputStream("d:\\test1.jpg");
            // 开始写入图像文件的字节
            fos.write(imageByte, 0, length);
            fos.close();
        }
        catch (Exception e)
        {
            return false;
        }
    }
}
```

```

        finally
        {
            if (fos != null)
            {
                try
                {
                    fos.close();
                }
                catch (Exception e)
                {
                }
            }
        }
    }
    return true;
}
// 返回一维字符串数组
public String[] getArray()
{
    String[] strArray = new String[]{ "自行车", "飞机", "火箭" };
    return strArray;
}
// 返回二维字符串数组
public String[] getMDArray()
{
    String[] strArray = new String[]{ "自行车,飞机,火箭","中国,美国,德国", "超人,蜘蛛侠,钢铁侠" };
    return strArray;
}
// 返回 DataForm 类的对象实例
public DataForm getDataForm()
{
    return new DataForm();
}
// 将 DataForm 类的对象实例序列化，并返回序列化后的字节数组
public byte[] getDataFormBytes() throws Exception
{
    java.io.ByteArrayOutputStream baos = new java.io.ByteArrayOutputStream();
    java.io.ObjectOutputStream oos = new java.io.ObjectOutputStream(baos);
    oos.writeObject(new DataForm());
    return baos.toByteArray();
}
}

```

## 二、实现 DataForm 类

DataForm 是要返回的对象实例所对应的类，该类的实现代码如下：

```

package data;

public class DataForm implements java.io.Serializable
{
    private String name = "bill";
    private int age = 20;

    public String getName()
    {
        return name;
    }
    public void setName(String name)
    {
        this.name = name;
    }
    public int getAge()
    {
        return age;
    }
    public void setAge(int age)
    {
        this.age = age;
    }
}

```

### 三、发布 WebService

由于本示例的 WebService 类使用了一个 Java 类（DataForm 类），因此，在发布 WebService 之前，需要先将 DataForm.class 文件复制到<Tomcat 安装目录>\webapps\axis2\WEB-INF\classes\data 目录中，然后将 ComplexTypeService.class 文件复制到<Tomcat 安装目录>\webapps\axis2\WEB-INF\pojo 目录中，最后启动 Tomcat（如果 Tomcat 已经启动，由于增加了一个 DataForm 类，因此，需要重新启动 Tomcat）。

### 四、使用 Java 编写调用 WebService 的客户端代码

在客户端仍然使用了 RPC 的调用方式，代码如下：

```

package client;

import javax.xml.namespace.QName;
import org.apache.axis2.addressing.EndpointReference;
import org.apache.axis2.client.Options;
import org.apache.axis2.rpc.client.RPCServiceClient;

public class ComplexTypeRPCClient
{
    public static void main(String[] args) throws Exception
    {
        RPCServiceClient serviceClient = new RPCServiceClient();
    }
}

```

```

Options options = serviceClient.getOptions();
EndpointReference targetEPR = new EndpointReference(
    "http://localhost:8080/axis2/services/ComplexTypeService");
options.setTo(targetEPR);
// 下面的代码调用 uploadImageWithByte 方法上传图像文件
////////////////////////////////////
// 打开图像文件，确定图像文件的大小
java.io.File file = new java.io.File("f:\\images.jpg");
java.io.FileInputStream fis = new java.io.FileInputStream("f:\\images.jpg");
// 创建保存要上传的图像文件内容的字节数组
byte[] buffer = new byte[(int) file.length()];
// 将图像文件的内容读取 buffer 数组中
int n = fis.read(buffer);
System.out.println("文件长度: " + file.length());
Object[] opAddEntryArgs = new Object[]{ buffer, n };
Class[] classes = new Class[]{ Boolean.class };
QName opAddEntry = new QName("http://ws.apache.org/axis2","uploadImageWithByte");
fis.close();
// 开始上传图像文件，并输出 uploadImageWithByte 方法的返回传
System.out.println(serviceClient.invokeBlocking(opAddEntry,opAddEntryArgs, classes)[0]);
////////////////////////////////////

// 下面的代码调用了 getArray 方法，并返回一维 String 数组
////////////////////////////////////
opAddEntry = new QName("http://ws.apache.org/axis2", "getArray");
String[] strArray = (String[]) serviceClient.invokeBlocking(opAddEntry,
    new Object[] {}, new Class[] {String[].class })[0];
for (String s : strArray)
    System.out.print(s + " ");
System.out.println();
////////////////////////////////////

// 下面的代码调用了 getMDArray 方法，并返回一维 String 数组
////////////////////////////////////
opAddEntry = new QName("http://ws.apache.org/axis2", "getMDArray");
strArray = (String[]) serviceClient.invokeBlocking(opAddEntry, new Object[] {},
    new Class[] {String[].class })[0];
for (String s : strArray)
{
    String[] array = s.split(",");
    for(String ss: array)
        System.out.print("<" + ss + "> ");
    System.out.println();
}
System.out.println();
////////////////////////////////////

// 下面的代码调用了 getDataForm 方法，并返回 DataForm 对象实例

```

```

////////////////////////////////////
opAddEntry = new QName("http://ws.apache.org/axis2", "getDataForm");
data.DataForm df = (data.DataForm) serviceClient.invokeBlocking(opAddEntry, new Object[]{},
                                                                    new Class[]{data.DataForm.class})[0];

System.out.println(df.getAge());
////////////////////////////////////

// 下面的代码调用了 getDataFormBytes 方法，并返回字节数组，最后将返回的字节数组反序列化后，转换成
DataForm 对象实例
////////////////////////////////////
opAddEntry = new QName("http://ws.apache.org/axis2", "getDataFormBytes");
buffer = (byte[]) serviceClient.invokeBlocking(opAddEntry, new Object[]{}, new Class[]{byte[].c
lass})[0];
java.io.ObjectInputStream ois = new java.io.ObjectInputStream(
    new java.io.ByteArrayInputStream(buffer));
df = (data.DataForm) ois.readObject();
System.out.println(df.getName());
////////////////////////////////////
}
}

```

运行上面的程序，将输出如下的内容：

文件长度： 3617

true

自行车 飞机 火箭

<自行车> <飞机> <火箭>

<中国> <美国> <德国>

<超人> <蜘蛛侠> <钢铁侠>

20

bill

## 五、使用 C# 编写调用 WebService 的客户端代码

在 Visual Studio 中使用 WebService 就简单得多。假设引用 WebService 时的引用名为 complexType，则下面的代码调用了 uploadImageWithByte 方法来上传图像文件。在 Visual Studio 引用 WebService 时，uploadImageWithByte 方法多了两个 out 参数，在使用时要注意。

```

complexType.ComplexTypeService cts = new WSC.complexType.ComplexTypeService();
System.IO.FileStream fs = new System.IO.FileStream(@"f:\images.jpg", System.IO.FileMode.Open);
byte[] buffer = new byte[fs.Length];
fs.Read(buffer, 0, (int)fs.Length);
bool r;

```

```
bool rs;  
cts.uploadImageWithByte( buffer, (int)fs.Length, true, out r, out rs);
```

在获得二维数组时，可以将数据加载到 **DataGridView** 或其他类似的控件中，代码如下：

```
String[] strArray = cts.getMDArray();  
for (int i = 0; i < strArray.Length; i++)  
{  
    // 用正则表达式将带分隔符的字符串转换成 String 数组  
    String[] columns = strArray[i].Split(',');  
    // 如果 DataGridView 的表头不存在，向 DataGridView 控件添加三个带表头的列  
    if (dataGridView1.Columns.Count == 0)  
        for (int j = 0; j < columns.Length; j++)  
            dataGridView1.Columns.Add("column" + (j + 1).ToString(), "列" + (j + 1).ToString());  
    // 添加行  
    dataGridView1.Rows.Add(1);  
    for(int j = 0; j < columns.Length; j++)  
    {  
        dataGridView1.Rows[i].Cells[j].Value = columns[j];  
    }  
}
```

向 **DataGridView** 控件添加数据后的效果如图 1 所示。



图 1

对于其他的 **WebService** 方法的调用都非常简单，读者可以自己做这个实验。

要注意的是，由于 **.net** 和 **java** 序列化和反序列化的差异，通过序列化的方式传递对象实例只使用于客户端与服务端为同一种语言或技术的情况，如客户端和服务端都使用 **Java** 来编写。

如果读者要上传大文件，应尽量使用 **FTP** 的方式来传递，而只通过 **WebService** 方法来传递文件名等信息。这样有助于提高传输效率。



用 Axis2 实现 Web Service，虽然可以将 POJO 类放在 axis2\WEB-INF\pojo 目录中直接发布成 Web Service，这样做不需要进行任何配置，但这些 POJO 类不能在任何包中。这似乎有些不方便，为此，Axis2 也允许将带包的 POJO 类发布成 Web Service。

先实现一个 POJO 类，代码如下：

```
package service;

public class MyService
{
    public String getGreeting(String name)
    {
        return "您好 " + name;
    }
    public void update(String data)
    {
        System.out.println("<" + data + ">已经更新");
    }
}
```

这个类有两个方法，这两个方法都需要发布成 Web Service 方法。这种方式和直接放在.pojo 目录中的 POJO 类不同。要想将 MyService 类发布成 Web Service，需要一个 services.xml 文件，这个文件需要放在 META-INF 目录中，该文件的内容如下：

```
<service name="myService">
  <description>
    Web Service 例子
  </description>
  <parameter name="ServiceClass">
    service.MyService
  </parameter>
  <messageReceivers>
    <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
      class="org.apache.axis2.rpc.receivers.RPCMessageReceiver" />
    <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-only"
      class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver" />
  </messageReceivers>
</service>
```

其中<service>元素用于发布 Web Service，一个<service>元素只能发布一个 WebService 类，name 属性表示 WebService 名，如下面的 URL 可以获得这个 WebService 的 WSDL 内容：

<http://localhost:8080/axis2/services/myService?wsdl>

其中 name 属性名就是上面 URL 中"?"和"/"之间的部分。

<description>元素表示当前 Web Service 的描述，<parameter>元素用于设置 WebService 的参数，在这里用于设置 WebService 对应的类名。在这里最值得注意的是<messageReceivers>元素，该元素用于设置处理 WebService 方法的处理器。例如，getGreeting 方法有一个返回值，因此，需要使用可处理输入输出的 RPCMessageReceiver 类，而 update 方法没有返回值，因此，需要使用只能处理输入的 RPCInOnlyMessageReceiver 类。

使用这种方式发布 **WebService**，必须打包成 **.aar** 文件，**.aar** 文件实际上就是改变了扩展名的 **.jar** 文件。在现在建立了两个文件：**MyService.java** 和 **services.xml**。将 **MyService.java** 编译，生成 **MyService.class**。**services.xml** 和 **MyService.class** 文件的位置如下：

D:\ws\service\MyService.class

D:\ws\META-INF\services.xml

在 windows 控制台中进入 **ws** 目录，并输入如下的命令生成 **.aar** 文件（实际上，**.jar** 文件也可以发布 **webservice**，但 **axis2** 官方文档中建议使用 **.aar** 文件发布 **webservice**）：

```
jar cvf ws.aar .
```

最后将 **ws.aar** 文件复制到 <Tomcat 安装目录>\webapps\axis2\WEB-INF\services 目录中，启动 Tomcat 后，就可以调用这个 **WebService** 了。调用的方法和《**WebService 大讲堂之 Axis2（1）：用 POJO 实现 0 配置的 WebService**》所讲的方法类似。

另外 **services.xml** 文件中也可以直接指定 **WebService** 类的方法，如可以用下面的配置代码来发布 **WebService**：

```
<service name="myService">
  <description>
    Web Service 例子
  </description>
  <parameter name="ServiceClass">
    service.MyService
  </parameter>
  <operation name="getGreeting">
    <messageReceiver class="org.apache.axis2.rpc.receivers.RPCMessageReceiver" />
  </operation>
  <operation name="update">
    <messageReceiver
      class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver" />
    </operation>
</service>
```

上面的配置代码前面的部分和以前的 **services.xml** 文件的内容相同，但后面使用了 **<operation>** 元素来指定每一个 **WebService** 方法，并单独指定了处理每一个方法的处理器。对于客户端来说，调用使用这两个 **services.xml** 文件发布的 **WebService** 并没有太大区别，只是使用第二个 **services.xml** 文件发布 **WebServices** 后，在使用 **wsdl2java** 命令或使用 **C#**、**delphi** 等生成客户端的 **stub** 时，**update** 方法的 **String** 类型被封装在了 **update** 类中，在传递 **update** 方法的参数时需要建立 **update** 类的对象实例。而使用第一个 **services.xml** 文件发布的 **WebService** 在生成 **stub** 时直接可以为 **update** 方法传递 **String** 类型的参数。从这一点可以看出，这两种方法生成的 **WSDL** 有一定的区别。但实际上，如果客户端程序使用第一个 **services.xml** 文件发布的 **WebService** 生成 **stub** 类时（这时 **update** 方法的参数是 **String**），在服务端又改为第二个 **services.xml** 文件来发布 **WebService**，这时客户端并不需要再重新生成 **stub** 类，而可以直接调用 **update** 方法。也就是说，服务端使用什么样的方式发布 **WebService**，对客户端并没有影响。

如果想发布多个 **WebService**，可以使用 **<serviceGroup>** 元素，如再建立一个 **MyService1** 类，代码如下：

```
package service
public class MyService1
{
  public String getName()
```

```

{
    return "bill";
}
}

```

在 services.xml 文件中可以使用如下的配置代码来配置 MyService 和 MyService1 类：

```

<serviceGroup>
  <service name="myService">
    <description>
      Web Service 例子
    </description>
    <parameter name="ServiceClass">
      service.MyService
    </parameter>
    <messageReceivers>
      <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
        class="org.apache.axis2.rpc.receivers.RPCMessageReceiver" />
      <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-only"
        class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver" />
    </messageReceivers>
  </service>
  <service name="myService1">
    <description>
      Web Service 例子
    </description>
    <parameter name="ServiceClass">
      service.MyService1
    </parameter>
    <messageReceivers>
      <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
        class="org.apache.axis2.rpc.receivers.RPCMessageReceiver" />
      <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-only"
        class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver" />
    </messageReceivers>
  </service>
</serviceGroup>

```

## WebService 大讲堂之 Axis2(4)：二进制文件传输

在《WebService 大讲堂之 Axis2（2）：复合类型数据的传递》中讲过，如果要传递二进制文件（如图像、音频文件等），可以使用 `byte[]` 作为数据类型进行传递，然后客户端使用 RPC 方式进行调用。这样做只是其中的一种方法，除此之外，在客户端还可以使用 `wsdl2java` 命令生成相应的 `stub` 类来调用 WebService，`wsdl2java` 命令的用法详见《WebService 大讲堂之 Axis2（1）：用 POJO 实现 0 配置的 WebService》。

WebService 类中包含 `byte[]` 类型参数的方法在 `wsdl2java` 生成的 `stub` 类中对应的数据类型不再是 `byte[]` 类型，而是 `javax.activation.DataHandler`。`DataHandler` 类是专门用来映射 WebService 二进制类型的。

在 WebService 类中除了可以使用 `byte[]` 作为传输二进制的数据类型外，也可以使用 `javax.activation.DataHandler` 作为数据类型。不管是使用 `byte[]`，还是使用 `javax.activation.DataHandler` 作为

WebService 方法的数据类型，使用 `wsdl2java` 命令生成的 `stub` 类中相应方法的类型都是 `javax.activation.DataHandler`。而象使用 `.net`、`delphi` 生成的 `stub` 类的相应方法类型都是 `byte[]`。这是由于 `javax.activation.DataHandler` 类是 Java 特有的，对于其他语言和技术来说，并不认识 `javax.activation.DataHandler` 类，因此，也只有使用最原始的 `byte[]` 了。

下面是一个上传二进制文件的例子，WebService 类的代码如下：

```
package service;

import java.io.InputStream;
import java.io.OutputStream;
import java.io.FileOutputStream;
import javax.activation.DataHandler;

public class FileService
{
    // 使用 byte[] 类型参数上传二进制文件
    public boolean uploadWithByte(byte[] file, String filename)
    {
        FileOutputStream fos = null;
        try
        {
            fos = new FileOutputStream(filename);
            fos.write(file);
            fos.close();
        }
        catch (Exception e)
        {
            return false;
        }
        finally
        {
            if (fos != null)
            {
                try
                {
                    fos.close();
                }
                catch (Exception e)
                {
                }
            }
        }
        return true;
    }

    private void writeInputStreamToFile(InputStream is, OutputStream os) throws Exception
    {
        int n = 0;
        byte[] buffer = new byte[8192];
        while((n = is.read(buffer)) > 0)
```

```

    {
        os.write(buffer, 0, n);
    }
}
// 使用 DataHandler 类型参数上传文件
public boolean uploadWithDataHandler(DataHandler file, String filename)
{
    FileOutputStream fos = null;
    try
    {
        fos = new FileOutputStream(filename);
        // 可通过 DataHandler 类的 getInputStream 方法读取上传数据
        writeInputStreamToFile(file.getInputStream(), fos);
        fos.close();
    }
    catch (Exception e)
    {
        return false;
    }
    finally
    {
        if (fos != null)
        {
            try
            {
                fos.close();
            }
            catch (Exception e)
            {
            }
        }
    }
    return true;
}
}

```

上面代码在 services.xml 文件的配置代码如下：

```

<service name="fileService">
    <description>
        文件服务
    </description>
    <parameter name="ServiceClass">
        service.FileService
    </parameter>
    <messageReceivers>
        <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
            class="org.apache.axis2.rpc.receivers.RPCMessageReceiver" />
    </messageReceivers>
</service>

```

```
</messageReceivers>
</service>
```

如果使用 `wsdl2java` 命令生成调用 Java 客户端代码，则需要创建 `DataHandler` 类的对象实例，代码如下：

```
DataHandler dh = new DataHandler(new FileDataSource(imagePath));
```

`wsdl2java` 命令会为每一个方法生成一个封装方法参数的类，类名为方法名（第一个字符大写），如 `uploadWithByte` 方法生成的类名为 `UploadWithByte`。如果要设置 `file` 参数的值，可以使用 `UploadWithByte` 类的 `setFile` 方法，代码如下：

```
UploadWithByte uwb = new UploadWithByte();
uwb.setFile(dh);
```

最后是调用 `uploadWithByte` 方法，代码如下（`FileServiceStub` 为 `wsdl2java` 生成的 `stub` 类名）：

```
FileServiceStub fss = new FileServiceStub();
fss.uploadWithByte(uwb);
```

如果使用 C# 调用 `FileService`，则 `file` 参数类型均为 `byte[]`，代码如下：

```
MemoryStream ms = new MemoryStream();
Bitmap bitmap = new Bitmap(picUpdateImage.Image);
bitmap.Save(ms, System.Drawing.Imaging.ImageFormat.Jpeg);
service.fileService fs = new WSC.service.fileService();
fs.uploadWithDataHandler(ms.ToArray());
fs.uploadWithByte(ms.ToArray());
```

其中 `picUpdateImage` 为 c# 中加载图像文件的 `picturebox` 控件。

## [WebService 大讲堂之 Axis2\(5\): 会话 \(Session\) 管理](#)

**WebService** 给人最直观的感觉就是由一个个方法组成，并在客户端通过 **SOAP** 协议调用这些方法。这些方法可能有返回值，也可能没有返回值。虽然这样可以完成一些工具，但这些被调用的方法是孤立的，当一个方法被调用后，在其他的方法中无法获得这个方法调用后的状态，也就是说无法保留状态。

读者可以想象，这对于一个完整的应用程序，无法保留状态，就意味着只依靠 **WebService** 很难完成全部的工作。例如，一个完整的应用系统都需要进行登录，这在 **Web** 应用中使用 **Session** 来保存用户登录状态，而如果用 **WebService** 的方法来进行登录处理，无法保存登录状态是非常令人尴尬的。当然，这也可以通过其他的方法来解决，如在服务端使用 **static** 变量来保存用户状态，并发送一个 `id` 到客户端，通过在服务端和客户端传递这个 `id` 来取得相应的用户状态。这非常类似于 **Web** 应用中通过 **Session** 和 **Cookie** 来管理用户状态。但这就需要由开发人员做很多工作，不过幸好 **Axis2** 为我们提供了 **WebService** 状态管理的功能。

使用 **Axis2** 来管理 **WebService** 的状态基本上对于开发人员是透明的。在 **WebService** 类需要使用 `org.apache.axis2.context.MessageContext` 和 `org.apache.axis2.context.ServiceContext` 类来保存与获得保存在服务端的状态信息，这有些象使用 `HttpSession` 接口的 `getAttribute` 和 `setAttribute` 方法获得与设置 **Session** 域属性。

除此之外，还需要修改 `services.xml` 文件的内容，为 `<service>` 元素加一个 `scope` 属性，该属性有四个可取的值：`Application`, `SOAPSession`, `TransportSession`, `Request`，不过要注意一下，虽然 **Axis2** 的官方文档将这四个值的



单词首字母和缩写字母都写成了大写，但经笔者测试，必须全部小写才有效，也就是这四个值应为：**application**、**soapsession**、**transportsession**、**request**，其中 **request** 为 **scope** 属性的默认值。读者可以选择使用 **transportsession** 和 **application** 分别实现同一个 **WebService** 类和跨 **WebService** 类的会话管理。

在客户端需要使用 **setManageSession(true)** 打开 **Session** 管理功能。

综上所述，实现同一个 **WebService** 的 **Session** 管理需要如下三步：

1. 使用 **MessageContext** 和 **ServiceContext** 获得与设置 **key-value** 对。
2. 为要进行 **Session** 管理的 **WebService** 类所对应的 **<service>** 元素添加一个 **scope** 属性，并将该属性值设为 **transportsession**。
3. 在客户端使用 **setManageSession(true)** 打开 **Session** 管理功能。

下面是一个在同一个 **WebService** 类中管理 **Session** 的例子。

先建立一个 **WebService** 类，代码如下：

```
package service;
import org.apache.axis2.context.ServiceContext;
import org.apache.axis2.context.MessageContext;
public class LoginService
{
    public boolean login(String username, String password)
    {
        if("bill".equals(username) && "1234".equals(password))
        {
            // 第 1 步：设置 key-value 对
            MessageContext mc = MessageContext.getCurrentMessageContext();
            ServiceContext sc = mc.getServiceContext();
            sc.setProperty("login", "成功登录");
            return true;
        }
        else
        {
            return false;
        }
    }
    public String getLoginMsg()
    {
        // 第 1 步：获得 key-value 对中的 value
        MessageContext mc = MessageContext.getCurrentMessageContext();
        ServiceContext sc = mc.getServiceContext();
        return (String)sc.getProperty("login");
    }
}
```

在 **LoginService** 类中有两个方法：**login** 和 **getLoginMsg**，如果 **login** 方法登录成功，会将“成功登录”字符串保存在 **ServiceContext** 对象中。如果在 **login** 方法返回 **true** 后调用 **getLoginMsg** 方法，就会返回“成功登录”。

下面是 **LoginService** 类的配置代码（**services.xml**）：

```

<!-- 第 2 步: 添加 scope 属性 -->
<service name="loginService" scope="transportsession">
  <description>
    登录服务
  </description>
  <parameter name="ServiceClass">
    service.LoginService
  </parameter>
  <messageReceivers>
    <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
      class="org.apache.axis2.rpc.receivers.RPCMessageReceiver" />
  </messageReceivers>
</service>

```

使用如下的命令生成客户端使用的 stub 类:

```
%AXIS2_HOME%\bin\wsdl2java -uri http://localhost:8080/axis2/services/loginService?wsdl -p client -s -o stub
```

在 stub\src\client 目录中生成了一个 LoginServiceStub.java 类, 在该类中找到如下的构造句方法:

```

public LoginServiceStub(org.apache.axis2.context.ConfigurationContext configurationContext,
    java.lang.String targetEndpoint, boolean useSeparateListener)
    throws org.apache.axis2.AxisFault
{
    ...
    _serviceClient.getOptions().setSoapVersionURI(
        org.apache.axiom.soap.SOAP12Constants.SOAP_ENVELOPE_NAMESPACE_URI);
}

```

在该方法中最后添加如下的代码:

```

// 第 3 步: 打开客户端的 Session 管理功能
_serviceClient.getOptions().setManageSession(true);

```

下面的客户端代码使用 LoginServiceStub 对象访问了刚才建立的 WebService:

```

LoginServiceStub stub = new LoginServiceStub();
LoginServiceStub.Login login = new LoginServiceStub.Login();
login.setUsername("bill");
login.setPassword("1234");
if(stub.login(login).local_return)
{
    System.out.println(stub.getLoginMsg().local_return);
}

```

运行上面的代码后, 会输出“成功登录”信息。

## [WebService 大讲堂之 Axis2\(6\): 跨服务会话\(Session\)管理](#)

在《WebService 大讲堂之 Axis2(5): 会话(Session)管理》一文中介绍了如何使用 Axis2 来管理同一个服务的会话, 但对于一个复杂的系统, 不可能只有一个 WebService 服务, 例如, 至少会有一个管理用户的 WebService (用

户登录和注册) 以及处理业务的 **WebService**。象这种情况, 就必须在多个 **WebService** 服务之间共享会话状态, 也称为跨服务会话(**Session**)管理。实现跨服务会话管理与实现同一个服务的会话管理的步骤类似, 但仍然有一些差别, 实现跨服务会话管理的步骤如下:

实现跨服务的 **Session** 管理需要如下三步:

1. 使用 **MessageContext** 和 **ServiceGroupContext** 获得与设置 key-value 对。
2. 为要进行 **Session** 管理的 **WebService** 类所对应的<service>元素添加一个 **scope** 属性, 并将该属性值设为 **application**。
3. 在客户端使用 **setManageSession(true)**打开 **Session** 管理功能。

从上面的步骤可以看出, 实现跨服务会话管理与实现同一个服务的会话管理在前两步上存在着差异, 而第 3 步是完全一样的。下面是一个跨服务的会话管理的实例。在这个例子中有两个 **WebService** 类: **LoginService** 和 **SearchService**, 代码如下:

LoginService.java

```
package service;
import org.apache.axis2.context.MessageContext;
import org.apache.axis2.context.ServiceGroupContext;
public class LoginService
{
    public boolean login(String username, String password)
    {
        if("bill".equals(username) && "1234".equals(password))
        {
            // 第 1 步: 设置 key-value 对
            MessageContext mc = MessageContext.getCurrentMessageContext();
            ServiceGroupContext sgc = mc.getServiceGroupContext();
            sgc.setProperty("login", "成功登录");
            return true;
        }
        else
        {
            return false;
        }
    }
    public String getLoginMsg()
    {
        // 第 1 步: 获得 key-value 对中的 value
        MessageContext mc = MessageContext.getCurrentMessageContext();
        ServiceGroupContext sgc = mc.getServiceGroupContext();
        return (String)sgc.getProperty("login");
    }
}
```

**SearchService.java**

```
package service;
import org.apache.axis2.context.MessageContext;
```

```

import org.apache.axis2.context.ServiceGroupContext;
public class SearchService
{
    public String findByName(String name)
    {
        // 第 1 步: 获得 key-value 对中的 value
        MessageContext mc = MessageContext.getCurrentMessageContext();
        ServiceGroupContext sgc = mc.getServiceGroupContext();
        if (sgc.getProperty("login") != null)
            return "找到的数据<" + name + ">";
        else
            return "用户未登录";
    }
}

```

services.xml 文件中的配置代码如下:

```

<serviceGroup>
    <!-- 第 2 步: 添加 scope 属性, 并设置属性值为 application -->
    <service name="loginService" scope="application">
        <description>
            登录服务
        </description>
        <parameter name="ServiceClass">
            service.LoginService
        </parameter>
        <messageReceivers>
            <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
                class="org.apache.axis2.rpc.receivers.RPCMessageReceiver" />
        </messageReceivers>
    </service>
    <!-- 第 2 步: 添加 scope 属性, 并设置属性值为 application -->
    <service name="searchService" scope="application">
        <description>
            搜索服务
        </description>
        <parameter name="ServiceClass">
            service.SearchService
        </parameter>
        <messageReceivers>
            <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
                class="org.apache.axis2.rpc.receivers.RPCMessageReceiver" />
        </messageReceivers>
    </service>
</serviceGroup>

```

第 3 步与《WebService 大讲堂之 Axis2(5): 会话 (Session) 管理》一文中介绍的方法类似。

下面是使用两个 stub 类的对象实例访问上面实现的两个 WebService 的客户端代码:

```

LoginServiceStub stub = new LoginServiceStub();
LoginServiceStub.Login login = new LoginServiceStub.Login();
login.setUsername("bill");
login.setPassword("1234");
if(stub.login(login).local_return)
{
    System.out.println(stub.getLoginMsg().local_return);
    SearchServiceStub searchStub = new SearchServiceStub();
    SearchServiceStub.FindByName fbn = new SearchServiceStub.FindByName();
    fbn.setName("abc");
    System.out.println(searchStub.findByName(fbn).local_return);
}

```

在执行上面的代码后，将输出如下的信息：

成功登录

找到的数据<abc>

读者可以将 `scope` 属性值改成 `transportsession`，看看会输出什么！

实际上，Axis2 的会话管理也是通过 Cookie 实现的，与 Web 应用中的 Session 管理类似。如果读者使用 C# 访问支持会话（在同一个服务中的会话管理）的 WebService，需要指定一个 CookieContainer 对象，代码如下：

```

service.loginService ls = new service.loginService();
System.Net.CookieContainer cc = new System.Net.CookieContainer();
ls.CookieContainer = cc;
bool r, rs;
ls.login("bill", "1234", out @r, out rs);
if (r)
{
    MessageBox.Show(ls.getLoginMsg().@return);
}

```

如果是访问跨服务的支持会话的 WebService，则不需要指定 CookieContainer 对象，代码如下：

```

service.loginService ls = new service.loginService();
bool r, rs;
ls.login("bill", "1234", out @r, out rs);
if (r)
{
    service1.searchService ss = new service1.searchService();
    MessageBox.Show(ss.findByName("abc"));
}

```

如果读者使用 delphi（本文使用的是 delphi2009，其他的 delphi 版本请读者自行测试）调用支持会话的 WebService 时有一些差别。经笔者测试，使用 delphi 调用 WebService，将 `scope` 属性值设为 `transportsession` 和 `application` 都可以实现跨服务的会话管理，这一点和 Java 与 C# 不同，Java 和 C# 必须将 `scope` 属性值设为 `application` 才支持跨服务会话管理。在 delphi 中不需要象 C# 指定一个 CookieContainer 或其他类似的对象，而只需要象访问普通的 WebService 一样访问支持会话的 WebService 即可。

在现今的 Web 应用中经常使用 Spring 框架来装载 JavaBean。如果要想将某些在 Spring 中装配的 JavaBean 发布成 WebService, 使用 Axis2 的 Spring 感知功能是很容易做到的。

在本文的例子中, 除了<Tomcat 安装目录>\webapps\axis2 目录及该目录中的相关库外, 还需要 Spring 框架中的 spring.jar 文件, 将该文件复制到<Tomcat 安装目录>\webapps\axis2\WEB-INF\lib 目录中。

下面先建立一个 JavaBean (该 JavaBean 最终要被发布成 WebService), 代码如下:

```
package service;
import entity.Person;
public class SpringService
{
    private String name;
    private String job;
    public void setName(String name)
    {
        this.name = name;
    }
    public void setJob(String job)
    {
        this.job = job;
    }
    public Person getPerson()
    {
        Person person = new Person();
        person.setName(name);
        person.setJob(job);
        return person;
    }
    public String getGreeting(String name)
    {
        return "hello " + name;
    }
}
```

其中 Person 也是一个 JavaBean, 代码如下:

```
package entity;
public class Person
{
    private String name;
    private String job;
    public String getName()
    {
        return name;
    }
    public void setName(String name)
    {
        this.name = name;
    }
    public String getJob()
```



```

{
    return job;
}
public void setJob(String job)
{
    this.job = job;
}
}

```

将上面两个 Java 源文件编译后，放到<Tomcat 安装目录>\webapps\axis2\WEB-INF\classes 目录中。

在<Tomcat 安装目录>\webapps\axis2\WEB-INF\web.xml 文件中加入下面的内容：

```

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

```

在<Tomcat 安装目录>\webapps\axis2\WEB-INF 目录中建立一个 applicationContext.xml 文件，该文件是 Spring 框架用于装配 JavaBean 的配置文件，内容如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
        http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">
    <bean id="springService" class="service.SpringService">
        <property name="name" value="姚明" />
        <property name="job" value="职业男篮" />
    </bean>
</beans>

```

在 applicationContext.xml 文件中装配了 service.SpringService 类，并被始化了 name 和 job 属性。在配置完 SpringService 类后，就可以直接在程序中 FileSystemXmlApplicationContext 类或其他类似功能的类读取 applicationContext.xml 文件中的内容，并获得 SpringService 类的对象实例。但现在我们并不这样做，而是将 SpringService 类发布成 WebService。

在<Tomcat 安装目录>\webapps\axis2\WEB-INF\lib 目录中有一个 axis2-spring-1.4.1.jar 文件，该文件用于将被装配 JavaBean 的发布成 WebService。在 D 盘建立一个 axi2-spring-ws 目录，并在该目录中建立一个 META-INF 子目录。在 META-INF 目录中建立一个 services.xml 文件，内容如下：

```

<service name="springService">
    <description>

```

```

    Spring aware
</description>
<parameter name="ServiceObjectSupplier">
    org.apache.axis2.extensions.spring.receivers.SpringServletContextObjectSupplier
</parameter>
<parameter name="SpringBeanName">
    springService
</parameter>
<messageReceivers>
    <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
        class="org.apache.axis2.rpc.receivers.RPCMessageReceiver" />
</messageReceivers>
</service>

```

在 Windows 控制台进入 `axi2-spring-ws` 目录，并使用 `jar` 命令将 `axi2-spring-ws` 目录中的内容打包成 `axi2-spring-ws.aar`，然后将该文件复制到 `<Tomcat 安装目录>\webapps\axis2\WEB-INF\services` 目录中，启动 Tomcat 后，就可以访问该 WebService 了，访问方式与前面几篇文章的访问方式相同。获得 `wsdl` 内容的 URL 如下：

<http://localhost:8080/axis2/services/springService?wsdl>

在将 Spring 中的装配 JavaBean 发布成 WebService 需要注意以下几点：

1. 由 JavaBean 编译生成的 `.class` 文件需要放在 `WEB-INF\classes` 目录中，或打成 `.jar` 包后放在 `WEB-INF\lib` 目录中，而 `WEB-INF\services` 目录中的 `.aar` 包中不需要包含 `.class` 文件，而只需要包含一个 `META-INF` 目录，并在该目录中包含一个 `services.xml` 文件即可。
2. `services.xml` 的配置方法与前几篇文章的配置方法类似，只是并不需要使用 `ServiceClass` 参数指定要发布成 WebService 的 Java 类，而是要指定在 `applicationContext.xml` 文件中的装配 JavaBean 的名称（`SpringBeanName` 参数）。
3. 在 `services.xml` 文件中需要通过 `ServiceObjectSupplier` 参数指定 `SpringServletContextObjectSupplier` 类来获得 Spring 的 `ApplicationContext` 对象。

## [WebService 大讲堂之 Axis2\(8\)：异步调用 WebService](#)

在前面几篇文章中都是使用同步方式来调用 WebService。也就是说，如果被调用的 WebService 方法长时间不返回，客户端将一直被阻塞，直到该方法返回为止。使用同步方法来调用 WebService 虽然很直观，但当 WebService 方法由于各种原因需要很长时间才能返回的话，就会使客户端程序一直处于等待状态，这样用户是无法忍受的。

当然，我们很容易就可以想到解决问题的方法，这就是多线程。解决问题的基本方法是将访问 WebService 的任务交由一个或多个线程来完成，而主线程并不负责访问 WebService。这样即使被访问的 WebService 方法长时间不返回，客户端仍然可以做其他的工作。我们可以管这种通过多线程访问 WebService 的方式称为异步访问。

虽然直接使用多线程可以很好地解决这个问题，但比较麻烦。幸好 Axis2 的客户端提供了异步访问 WebService 的功能。

`RPCServiceClient` 类提供了一个 `invokeNonBlocking` 方法可以通过异步的方式来访问 WebService。下面先来建立一个 WebService。

`MyService` 是一个 WebService 类，代码如下：

```

package service;
public class MyService
{
    public String getName()

```

```

{
    try
    {
        System.out.println("getName 方法正在执行 ...");
        // 延迟 5 秒
        Thread.sleep(5000);
    }
    catch (Exception e)
    {
    }
    return "火星";
}
}

```

为了模拟需要一定时间才返回的 WebService 方法，在 getName 方法中使用了 sleep 方法来延迟 5 秒。下面是 MyService 类的配置代码：

```

<!-- services.xml -->
<service name="myService">
    <description>
        异步调用演示
    </description>
    <parameter name="ServiceClass">
        service.MyService
    </parameter>
    <messageReceivers>
        <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
            class="org.apache.axis2.rpc.receivers.RPCMessageReceiver" />
    </messageReceivers>
</service>

```

从上面的配置代码可以看出，MyService 的配置方式与前几章的 WebService 的配置方式完全一样，也就是说，MyService 只是一个普通的 WebService。

下面是异步调用 MyService 的 Java 客户端代码：

```

package client;

import javax.xml.namespace.QName;
import org.apache.axis2.addressing.EndpointReference;
import org.apache.axis2.client.Options;
import org.apache.axis2.context.MessageContext;
import org.apache.axis2.rpc.client.RPCServiceClient;

public class RPCAsyncClient
{
    public static void main(String[] args) throws Exception
    {
        RPCServiceClient serviceClient = new RPCServiceClient();
        Options options = serviceClient.getOptions();
        EndpointReference targetEPR = new EndpointReference(

```

```

        "http://localhost:8080/axis2/services/myService");
options.setTo(targetEPR);
Object[] opAddEntryArgs = new Object[]{};
QName opAddEntry = new QName("http://service", "getName");
serviceClient.invokeNonBlocking(opAddEntry, opAddEntryArgs,
    new org.apache.axis2.client.async.AxisCallback()
    {
        @Override
        public void onComplete()
        {
        }
        @Override
        public void onError(Exception arg0)
        {
        }
        @Override
        public void onFault(MessageContext arg0)
        {
        }
        @Override
        public void onMessage(MessageContext mc)
        {
            // 输出返回值
            System.out.println(mc.getEnvelope().getFirstElement()
                .getFirstElement().getFirstElement().getText());
        }
    });
System.out.println("异步调用！");
// 阻止程序退出
System.in.read();
}
}

```

从上面的代码可以看出，`invokeNonBlocking` 方法有三个参数，前两个参数分别指定了要调用的方法及方法参数的相关信息，而最后一个参数并不是方法返回值的类型信息，而是一个实现 `org.apache.axis2.client.async.AxisCallback` 接口的类的对象实例。在本例中隐式实现了 `AxisCallback` 接口。在 `AxisCallback` 接口中有四个方法需要实现，其中当被异步调用的方法返回时 `onMessage` 方法被调用。当运行上面的程序后，将输出如下的信息：

```

异步调用！
火星

```

虽然上面的例子可以实现异步调用，但比较麻烦。为了方便地实现异步调用，可以使用 `wsdl2java` 命令的 `-a` 参数生成可异步调用的 `Stub` 类。下面的命令可生成同步和异步调用的客户端代码（两个类），其中 `-s` 表示生成同步调用代码，`-a` 表示生成异步调用代码。

```

%AXIS2_HOME%\bin\wsdl2java -uri http://localhost:8080/axis2/services/myService?wsdl -p client -s -a -o stub

```

在执行上面的命令后，将生成两个类：`MyServiceStub` 和 `MyServiceCallbackHandler` 类，其中 `MyServiceStub` 类负责同步和异步调用 `WebService`，`MyServiceCallbackHandler` 类是一个抽象类，也是一个回调类，当使用异步方

式调用 **WebService** 方法时，如果方法返回，则 **MyServiceCallbackHandler** 类的 **receiveResultgetName** 方法被调用。下面是使用 **MyServiceStub** 类异步访问 **WebService** 的代码：

```
package client;

import client.MyServiceStub.GetNameResponse;

class MyCallback extends MyServiceCallbackHandler
{
    @Override
    public void receiveResultgetName(GetNameResponse result)
    {
        // 输出 getName 方法的返回结果
        System.out.println(result.get_return());
    }
}

public class StubClient
{
    public static void main(String[] args) throws Exception
    {
        MyServiceStub stub = new MyServiceStub();
        // 异步调用 WebService
        stub.startgetName(new MyCallback());
        System.out.println("异步调用！");
        System.in.read();
    }
}
```

执行上面的程序后，将输出如下的信息：

```
异步调用！
火星
```

在 .net 中也可以使用异步的方式来调用 **WebService**，如在 C# 中可使用如下的代码来异步调用 **getName** 方法：

```
// 回调方法
private void getNameCompletedEvent(object sender, WSC.asyn.getNameCompletedEventArgs e)
{
    listBox1.Items.Add( e.Result.@return);
}

private void button1_Click(object sender, EventArgs e)
{
    async.myService my = new WSC.asyn.myService();
    my.getNameCompleted += new WSC.asyn.getNameCompletedEventHandler(getNameCompletedEvent);
    my.getNameAsync();
    MessageBox.Show("完成调用");
}
```

其中 **async** 是引用 **MyService** 的服务名。要注意的是，在 C# 中不能在同一个 **WebService** 实例的 **getName** 方法未返回之前，再次调用该实例的 **getName** 方法，否则将抛出异常。如下面的代码会抛出一个异常：

```

async.myService my = new WSC.async.myService();
my.getNameCompleted += new WSC.async.getNameCompletedEventHandler(getNameCompletedEvent);
my.getNameAsync();
// 将抛出异常
my.getNameAsync();

```

但不同的 **WebService** 实例的方法可以在方法未返回时调用，如下面的代码是可以正常工作的：

```

asyn.myService my = new WSC.asyn.myService();
my.getNameAsync();
my.getNameCompleted += new WSC.asyn.getNameCompletedEventHandler(getNameCompletedEvent);
asyn.myService my1 = new WSC.asyn.myService();
my1.getNameCompleted += new WSC.asyn.getNameCompletedEventHandler(getNameCompletedEvent);
my1.getNameAsync();

```

## [WebService 大讲堂之 Axis2\(9\): 编写 Axis2 模块 \(Module\)](#)

**Axis2** 可以通过模块 (Module) 进行扩展。**Axis2** 模块至少需要有两个类，这两个类分别实现了 **Module** 和 **Handler** 接口。开发和使用一个 **Axis2** 模块的步骤如下：

1. 编写实现 **Module** 接口的类。**Axis2** 模块在进行初始化、销毁等动作时会调用该类中相应的方法。
2. 编写实现 **Handler** 接口的类。该类是 **Axis2** 模块的业务处理类。
3. 编写 **module.xml** 文件。该文件放在 **META-INF** 目录中，用于配置 **Axis2** 模块。
4. 在 **axis2.xml** 文件中配置 **Axis2** 模块。
5. 在 **services.xml** 文件中配置 **Axis2** 模块。每一个 **Axis2** 模块都需要使用 **<module>** 元素引用才能使用。
6. 发布 **Axis2** 模块。需要使用 **jar** 命令将 **Axis2** 模块压缩成 **.mar** 包（文件扩展名必须是 **.mar**），然后将 **.mar** 文件放在

**<Tomcat 安装目录>\webapps\axis2\WEB-INF\modules** 目录中。

先来编写一个 **WebService** 类，代码如下：

```

package service;

public class MyService
{
    public String getGreeting(String name)
    {
        return "您好 " + name;
    }
}

```

下面我们来编写一个记录请求和响应 **SOAP** 消息的 **Axis2** 模块。当客户端调用 **WebService** 方法时，该 **Axis2** 模块会将请求和响应 **SOAP** 消息输出到 **Tomcat** 控制台上。

### 第 1 步：编写 **LoggingModule** 类

**LoggingModule** 类实现了 **Module** 接口，代码如下：



```

package module;

import org.apache.axis2.AxisFault;
import org.apache.axis2.context.ConfigurationContext;
import org.apache.axis2.description.AxisDescription;
import org.apache.axis2.description.AxisModule;
import org.apache.axis2.modules.Module;
import org.apache.neethi.Assertion;
import org.apache.neethi.Policy;

public class LoggingModule implements Module
{
    // initialize the module
    public void init(ConfigurationContext configContext, AxisModule module)
        throws AxisFault
    {
        System.out.println("init");
    }
    public void engageNotify(AxisDescription axisDescription) throws AxisFault
    {
    }
    // shutdown the module
    public void shutdown(ConfigurationContext configurationContext)
        throws AxisFault
    {
        System.out.println("shutdown");
    }
    public String[] getPolicyNamespaces()
    {
        return null;
    }
    public void applyPolicy(Policy policy, AxisDescription axisDescription)
        throws AxisFault
    {
    }
    public boolean canSupportAssertion(Assertion assertion)
    {
        return true;
    }
}

```

在本例中 `LoggingModule` 类并没实现实际的功能，但该类必须存在。当 `Tomcat` 启动时会装载该 `Axis2` 模块，同时会调用 `LoggingModule` 类的 `init` 方法，并在 `Tomcat` 控制台中输出“init”。

## 第 2 步：编写 `LogHandler` 类

`LogHandler` 类实现了 `Handler` 接口，代码如下：

```

package module;

import org.apache.axis2.AxisFault;

```

```

import org.apache.axis2.context.MessageContext;
import org.apache.axis2.engine.Handler;
import org.apache.axis2.handlers.AbstractHandler;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class LogHandler extends AbstractHandler implements Handler
{
    private static final Log log = LogFactory.getLog(LogHandler.class);
    private String name;
    public String getName()
    {
        return name;
    }
    public InvocationResponse invoke(MessageContext msgContext)
        throws AxisFault
    {
        // 向 Tomcat 控制台输出请求和响应 SOAP 消息
        log.info(msgContext.getEnvelope().toString());
        return InvocationResponse.CONTINUE;
    }
    public void revoke(MessageContext msgContext)
    {
        log.info(msgContext.getEnvelope().toString());
    }
    public void setName(String name)
    {
        this.name = name;
    }
}

```

LogHandler 类的核心方法是 invoke，当使用该 Axis2 模块的 WebService 的方法被调用时，LogHandler 类的 invoke 方法被调用。

### 第 3 步：编写 module.xml 文件

在 META-INF 目录中建立一个 module.xml 文件，内容如下：

```

<module name="logging" class="module.LoggingModule">
    <InFlow>
        <handler name="InFlowLogHandler" class="module.LogHandler">
            <order phase="loggingPhase"/>
        </handler>
    </InFlow>
    <OutFlow>
        <handler name="OutFlowLogHandler" class="module.LogHandler">
            <order phase="loggingPhase"/>
        </handler>
    </OutFlow>

    <OutFaultFlow>

```

```

    <handler name="FaultOutFlowLogHandler" class="module.LogHandler">
      <order phase="loggingPhase"/>
    </handler>
  </OutFaultFlow>
  <InFaultFlow>
    <handler name="FaultInFlowLogHandler" class="module.LogHandler">
      <order phase="loggingPhase"/>
    </handler>
  </InFaultFlow>
</module>

```

#### 第 4 步：在 **axis2.xml** 文件中配置 **Axis2** 模块

打开 **axis2.xml** 文件，分别在如下四个 **<phaseOrder>** 元素中加入 **<phase name="loggingPhase"/>**：

```

<phaseOrder type="InFlow">
  ...
  <phase name="soapmonitorPhase"/>
  <phase name="loggingPhase"/>
</phaseOrder>
<phaseOrder type="OutFlow">
  ...
  <phase name="Security"/>
  <phase name="loggingPhase"/>
</phaseOrder>
<phaseOrder type="InFaultFlow">
  ...
  <phase name="soapmonitorPhase"/>
  <phase name="loggingPhase"/>
</phaseOrder>
<phaseOrder type="OutFaultFlow">
  ...
  <phase name="Security"/>
  <phase name="loggingPhase"/>
</phaseOrder>

```

#### 第 5 步：在 **services.xml** 文件中引用 **logging** 模块

**services.xml** 文件的内容如下：

```

<service name="myService">
  <description>
    使用 logging 模块
  </description>
  <!-- 引用 logging 模块 -->
  <module ref="logging"/>
  <parameter name="ServiceClass">
    service.MyService
  </parameter>
</service>

```

```
</parameter>
<messageReceivers>
  <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
    class="org.apache.axis2.rpc.receivers.RPCMessageReceiver" />
</messageReceivers>
</service>
```

## 第 6 步：发布 logging 模块

到现在为止，我们应用可以建立两个发行包：logging.mar 和 service.aar。其中 logging.mar 文件是 Axis2 模块的发行包，该包的目录结构如下：

logging.mar

module\LoggingModule.class

module\LogHandler.class

META-INF\module.xml

service.aar 文件是本例编写的 WebService 发行包，该包的目录结构如下：

service.aar

service\MyService.class

META-INF\services.xml

将 logging.mar 文件放在 <Tomcat 安装目录>\webapps\axis2\WEB-INF\modules 目录中，将 service.aar 文件放在 <Tomcat 安装目录>\webapps\axis2\WEB-INF\services 目录中。要注意的是，如果 modules 目录中包含了 modules.list 文件，Axis2 会只装载在该文件中引用的 Axis2 模块，因此，必须在该文件中引用 logging 模块，该文件的内容如下：

addressing-1.4.1.mar

soapmonitor-1.4.1.mar

ping-1.4.1.mar

mex-1.4.1.mar

axis2-scripting-1.4.1.mar

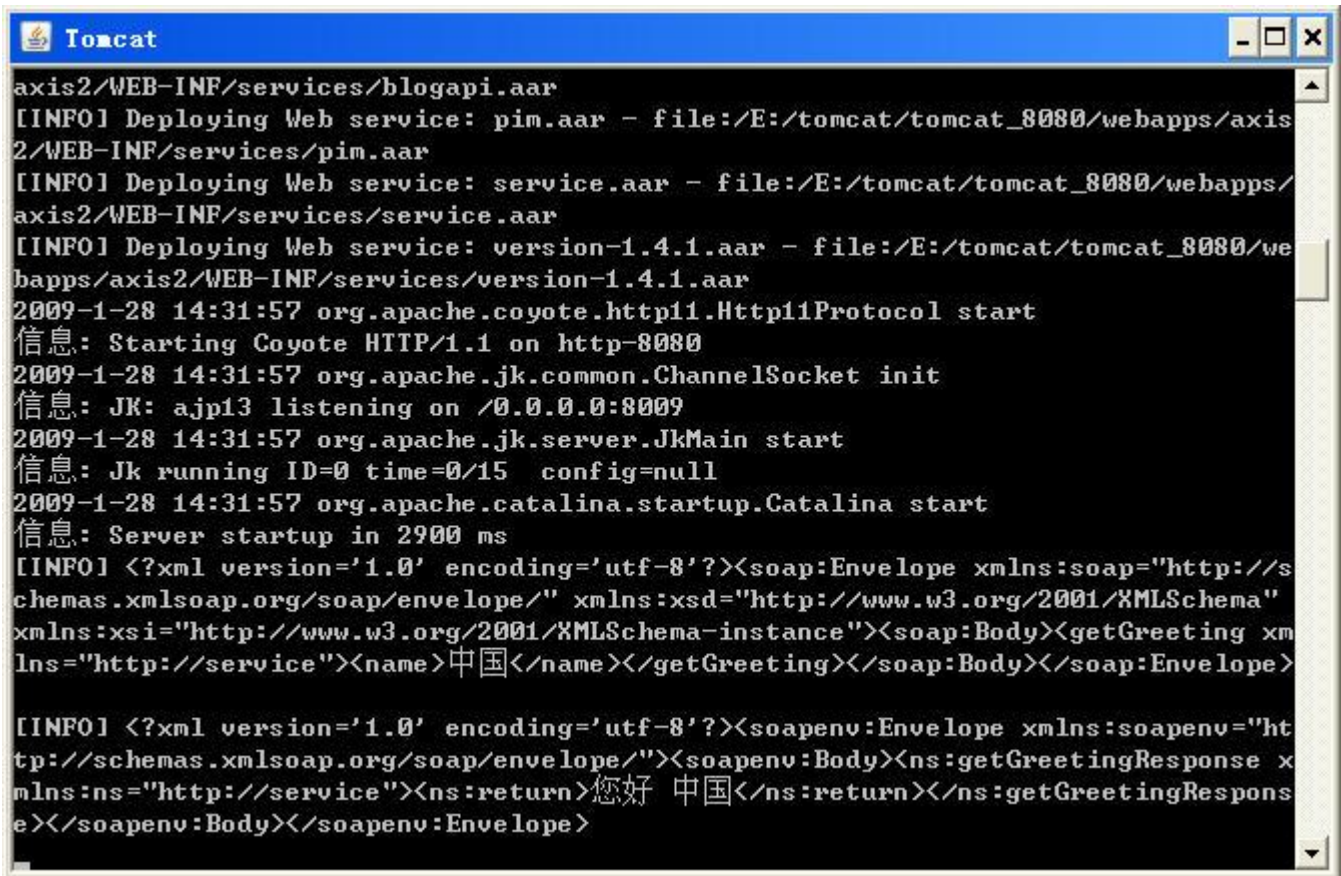
logging.mar

如果 modules 目录中不包含 modules.list 文件，则 Axis2 会装载 modules 文件中的所有 Axis2 模块。

现在启动 Tomcat，使用如下的 C# 代码调用 MyService 的 getGreeting 方法则会在 Tomcat 控制台中输出相应的请求和响应 SOAP 消息。

```
// async 是引用 MyService 的服务名
async.myService my = new WSC.asyn.myService();
MessageBox.Show(my.getGreeting("中国"));
MessageBox.Show("完成调用");
```

在执行上面的代码后，在 Tomcat 控制台中输出的信息如下图所示。



```
axis2/WEB-INF/services/blogapi.aar
[INFO] Deploying Web service: pim.aar - file:/E:/tomcat/tomcat_8080/webapps/axis2/WEB-INF/services/pim.aar
[INFO] Deploying Web service: service.aar - file:/E:/tomcat/tomcat_8080/webapps/axis2/WEB-INF/services/service.aar
[INFO] Deploying Web service: version-1.4.1.aar - file:/E:/tomcat/tomcat_8080/webapps/axis2/WEB-INF/services/version-1.4.1.aar
2009-1-28 14:31:57 org.apache.coyote.http11.Http11Protocol start
信息: Starting Coyote HTTP/1.1 on http-8080
2009-1-28 14:31:57 org.apache.jk.common.ChannelSocket init
信息: JK: ajp13 listening on /0.0.0.0:8009
2009-1-28 14:31:57 org.apache.jk.server.JkMain start
信息: Jk running ID=0 time=0/15 config=null
2009-1-28 14:31:57 org.apache.catalina.startup.Catalina start
信息: Server startup in 2900 ms
[INFO] <?xml version='1.0' encoding='utf-8'?><soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"><soap:Body><getGreeting xmlns="http://service"><name>中国</name></getGreeting></soap:Body></soap:Envelope>

[INFO] <?xml version='1.0' encoding='utf-8'?><soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"><soapenv:Body><ns:getGreetingResponse xmlns:ns="http://service"><ns:return>您好 中国</ns:return></ns:getGreetingResponse></soapenv:Body></soapenv:Envelope>
```

## [WebService 大讲堂之 Axis2\(10\): 使用 soapmonitor 模块监视 soap 请求与响应消息](#)

在 Axis2 中提供了一个 Axis2 模块（soapmonitor），该模块实现了与《WebService 大讲堂之 Axis2(9): 编写 Axis2 模块（Module）》中实现的 logging 模块相同的功能，所不同的是，logging 模块直接将 SOAP 请求与响应消息输出到 Tomcat 控制台中，而 soapmonitor 模块利用 applet 直接在页面中输出 SOAP 请求和响应消息。

下面是配置和使用 soapmonitor 模块的步骤：

### 第 1 步：部署 Applet 和 Servlet

由于 axis2 默认情况下已经自带了 soapmonitor 模块，因此，soapmonitor 模块并不需要单独安装。但 applet 所涉及到的相应的.class 文件需要安装一下。在<Tomcat 安装目录>\webapps\axis2\WEB-INF\lib 目录中找到 soapmonitor-1.4.1.jar 文件，将该文件解压。虽然 applet 并不需要 soapmonitor-1.4.1.jar 文件中所有的.class 文件，但为了方便，读者也可以直接将解压目录中的 org 目录复制到<Tomcat 安装目录>\webapps\axis2 目录中，Applet 所需的.class 文件需要放在这个目录。然后再将 org 目录复制到<Tomcat 安装目录>\webapps\axis2\WEB-INF\classes 目录中，soapmonitor 模块中的 Servlet 所对应的.class 文件需要放在这个目录。

### 第 2 步：配置 Servlet

打开<Tomcat 安装目录>\webapps\axis2\WEB-INF\web.xml 文件，在其中加入如下的内容：

```
<servlet>
  <servlet-name>SOAPMonitorService</servlet-name>
  <servlet-class>
    org.apache.axis2.soapmonitor.servlet.SOAPMonitorService
  </servlet-class>
```

```

<init-param>
  <param-name>SOAPMonitorPort</param-name>
  <param-value>5001</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>SOAPMonitorService</servlet-name>
  <url-pattern>/SOAPMonitor</url-pattern>
</servlet-mapping>

```

### 第 3 步：在 **services.xml** 文件中引用 **soapmonitor** 模块

与引用 logging 模块一样，引用 soapmonitor 模块也需要使用<module>元素，引用 soapmonitor 模块的 services.xml 文件的内容如下：

```

<service name="myService">
  <description>
    使用 logging 和 soapmonitor 模块
  </description>
  <!-- 引用 logging 模块 -->
  <module ref="logging"/>
  <!-- 引用 soapmonitor 模块 -->
  <module ref="soapmonitor"/>
  <parameter name="ServiceClass">
    service.MyService
  </parameter>
  <messageReceivers>
    <messageReceiver mep="http://www.w3.org/2004/08/wsdl/in-out"
      class="org.apache.axis2.rpc.receivers.RPCMessageReceiver" />
  </messageReceivers>
</service>

```

由于 soapmonitor 模块已经在 axis2.xml 进行配置了，因此，在本例中不需要再对 axis2.xml 文件进行配置了。

### 第 4 步：使用 **soapmonitor** 模块

启动 Tomcat 后，在浏览器中输入如下的 URL：

<http://localhost:8080/axis2/SOAPMonitor>

在浏览器中将出现 soapmonitor 所带的 Applet 的界面，当访问 MyService 的 getGreeting 方法时，在 Tomcat 控制台与 Applet 中都显示了相应的 SOAP 请求和响应消息。如图 1 和图 2 分别是调用了两次 getGreeting 方法后输出的 SOAP 请求和响应消息。



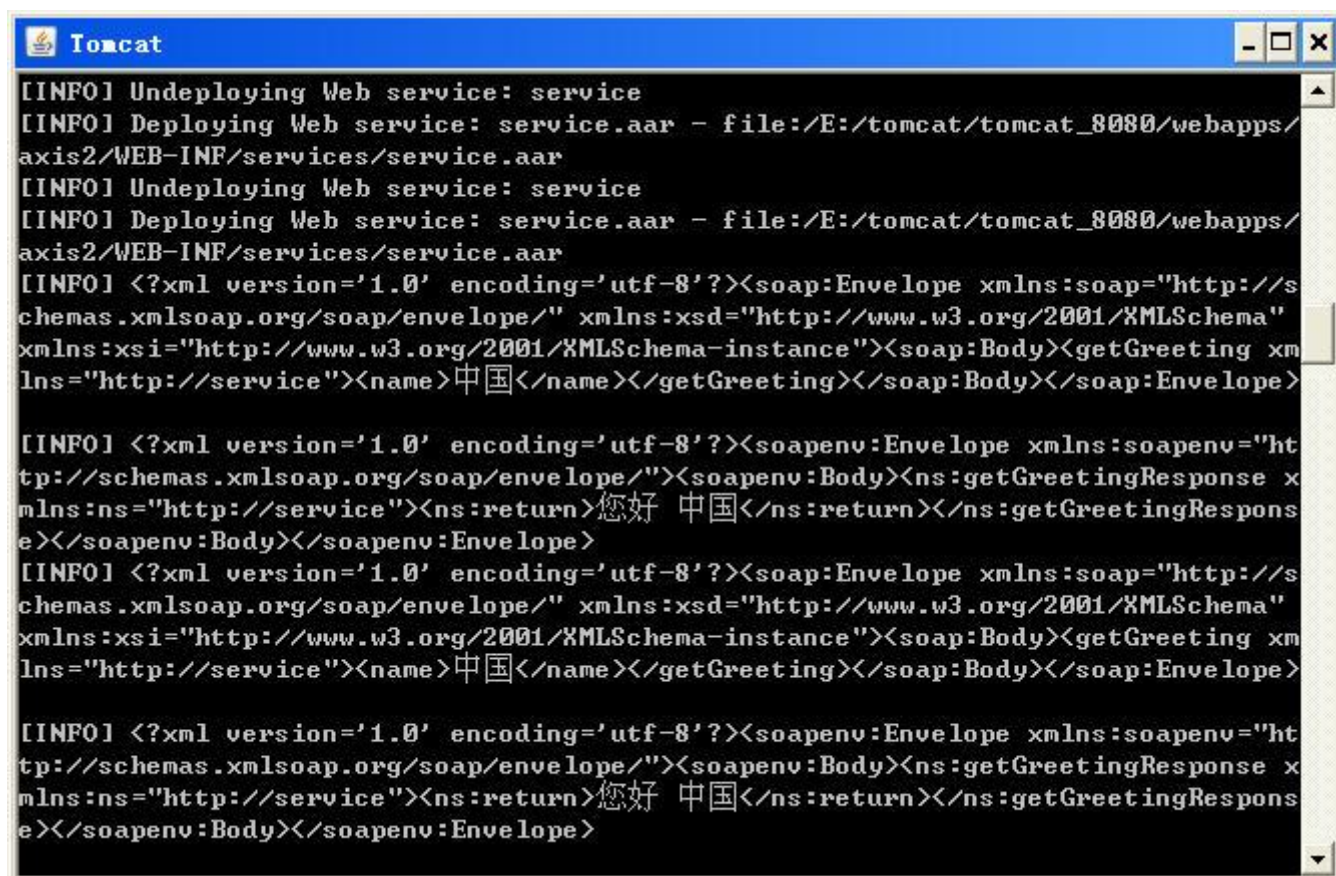


图 1

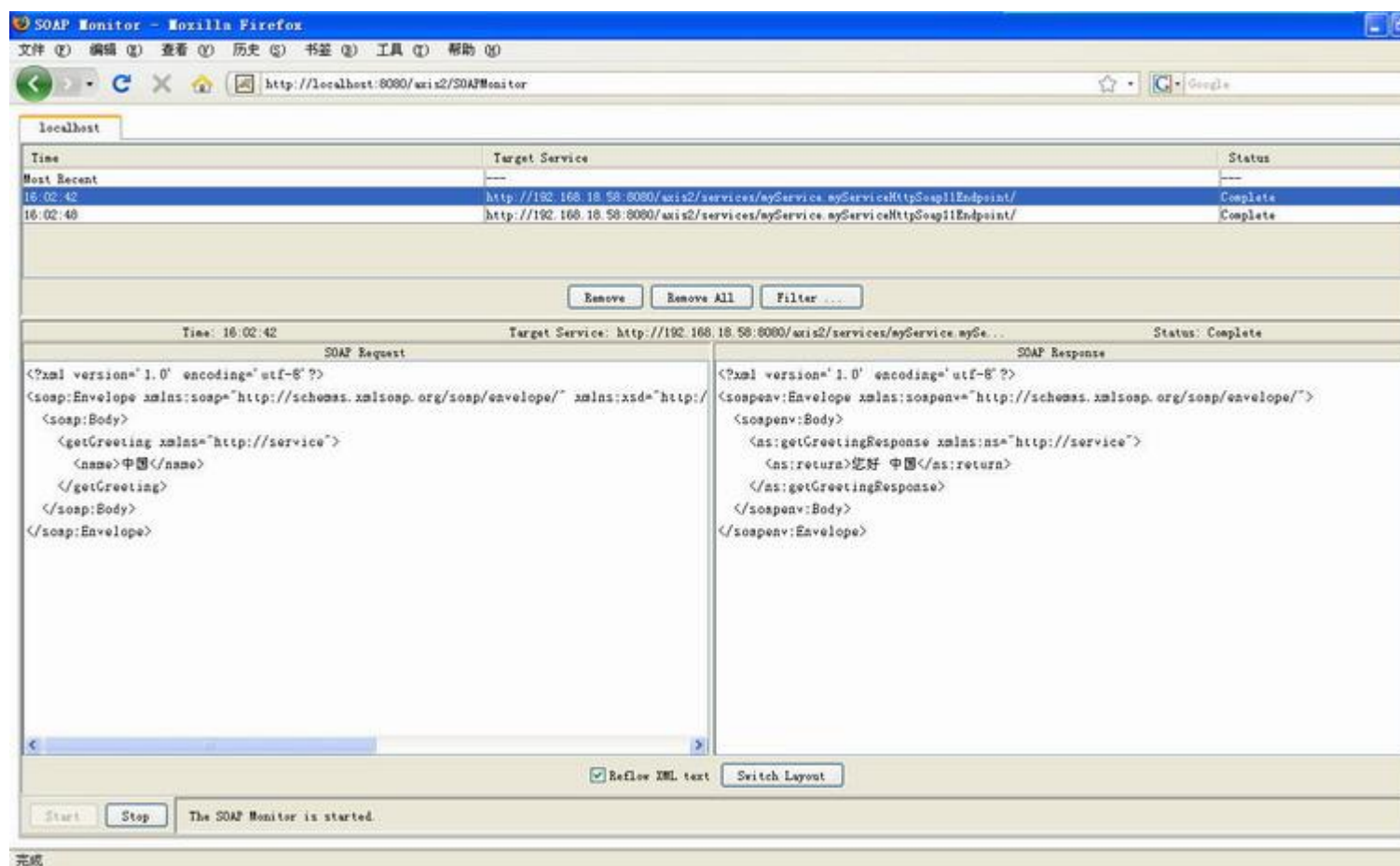




图 2

如果读者想让 `logging` 和 `soapmonitor` 模块监视部署在 Axis2 中的所有 `WebService`，可以在 `axis2.xml` 文件中使用 `<module>` 元素来引用这两个模块，代码如下：

```
<!-- 引用 logging 模块 -->  
<module ref="logging"/>  
<!-- 引用 soapmonitor 模块 -->  
<module ref="soapmonitor"/>
```