Facets of XFire

XFire Guide V2

XFire开发指南

晁岳攀

http://smallnest.googlepages.com

XFire 野猪书

如果可以将 XFire Web Services 框架比做一颗璀璨夺目的钻石的话，那么本书将从这颗钻石的多个切面上来欣赏它闪耀的光芒。

大约是在去年，我写了一份关于 XFire 开发的电子文档，介绍了采用 XFire 框架开发 Web Services 的基本的方法。由于以前的 XFire 官方的文档不很准确，也不完善，有些实践的代码没有通过，所以在那个文档没有进行深层次的探讨。陆续有些网友提出了一些开发的问题，比如参数如果是自定义类型的问题、传送图片的问题、自动创建代码的问题，很多问题都是在实际应用 XFire 进行 Web Services 所必须面临的问题。基于此，我把实际使用 XFire 开发时的一些实践整理出来，希望能对正在采用和将要采用 XFire 框架进行开发的朋友有所帮助。

通过本书，你能了解到如何快速的采用 XFire 开发一个 Web Service，如何根据一个 WSDL 文件生成服务器端和客户端代码，如何在一个桌面应用程序嵌入 Web Services，本书也演示了采用三行代码就完成调用一个 Web Service 的过程。本书介绍了如何采用 Aegis、jaxb2、xmlbeans、Castor、JiBX、MessageBinding 等多种 Binding 方式将 XML 映射到 JAVA 对象上。最后本书介绍了 XFire 的身份验证方式、与 Spring 容器的继承以及 XFire 对 MTOM 支持。

显然，本书不是 XFire 的用户手册，也不是官方的用户指南，而是根据开发过程中总结的一些实践文档，并不能涵盖 XFire 的方方面面。XFire 最权威也最翔实的文档毫无疑问是官方的 Wiki，尤其是 Wiki 上的 User's Guide。

晁岳攀

smallnest@gmail.com

关注濒危动物，热爱大自然

世界体型最小、最罕见的「侏儒猪」一度濒临绝种，体长约 30 公分，体重不到 10 公斤。

内容目录

XFire 开发起步

本章的内容包括：

- Xfire 介绍
- 一个简单的 Web Service
- 发布一个 Web Service
- 编写客户端的代码

XFire 是下一代的 java SOAP 框架。XFire 提供了非常方便的 API，使用这些 API 可以开发面向服务(SOA)的程序。它支持各种标准，性能优良（基于低内存的 STAX 模型）。

- 支持多个重要的 Web Service 标准，包括 SOAP、WSDL、WS-I Basic Profile、WS-Addressing、WS-Security 等
- 高性能的 SOAP 栈
- 可选的绑定(binding)方式，如 POJO、XMLBeansJAXB1.1、JAXB2.0、Castor 和 JiBX 等
- 支持 JSR181 API
- 多种传输方式，如 HTTP、JMS、XMPP、In - JVM 等
- 灵活的接口
- 支持多个容器，如 Spring、Pico、Plexus、Loom
- 支持 JBI，参看 servicemix 项目(http://servicemix.org)
- 客户端和服务器代码生成

本书中对 Web 服务泛指时，通常以 Web Services（复数形式）来表示，针对某个特定的 Web 服务，以 Web Service(单数形式)来表示。

在准备第一个例子前，先把本书所用到的软件及版本说明一下。XFire 不同的版本会有些差别，建议下载最新的 XFire 版本。

操作系统

Fedora Core 6,Linux 内核版本为 2.6.18-1.2798。当然你可以选择 Windows。

开发工具

Eclipse3.2.0 + sysdeo tomcat插件

Web 服务器

tomcat 6.0.10

XFire 1.2.5

你可以到官方网站下载 xfire-distribution-1.2.5.zip，下载到本地解压后可以得到 XFire 编译打包后的文件 xfire-all-1.2.5.jar，lib 目录下为 XFire 所依赖的第三方的 Jar 文件，你开发的时候会用得着。此外还包括手册和几个例子。建议在 Eclipse 里建一个用户库，把 xfire-all-1.2.5.jar 和第三方的 Jar 文件都包含进去，因为下面几章的开发都会用到这个库。

第一个 Web Service 程序

(本书附带的源代码 facet0)

OK,可以开始编写你第一个 Web Service 程序了。

首先在 Eclipse 下建立一个 Tomcat 工程，名字叫 facet0,web 应用程序的上下文(context)为 facet。(实际开发时工程名和上下文你可以随意命名，只要保证和下面的访问的地址一致即可，下面每个章节的相关的命名都可以自己命名，不再赘述，免得唐僧)。

在 WEB-INF 文件夹下新建一个 web.xml 文件，内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
       http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <servlet>
    <servlet-name>XFireServlet</servlet-name>
```

```
    <display-name>XFire Servlet</display-name>
    <servlet-class>
        org.codehaus.xfire.transport.http.XFireConfigurableServlet
    </servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>XFireServlet</servlet-name>
    <url-pattern>/servlet/XFireServlet/*</url-pattern>
  </servlet-mapping>

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

在这个 web.xml 文件中定义了一个 XfireServlet，它负责提供 Web Services，并提供每个 Web Service 的 WSDL。如果你发布了一个 Web Service，比如叫 BookService，你就可以通过网址 http://<server_url[:port]>/<context>/ services/BookService 来访问这个 Web Service，并且通过网址 http://<server_url[:port]>/<context>/ services/BookService?WSDL 得到这个 Web Service 的 WSDL 信息。

在源文件夹 WEB-INF/src 下新建一个 package：com.googlepages.smallnest.facet,在这个 package 下新建一个接口 HelloService,这个接口定义了只定义了一个 hello 方法。这个方法要求传入一个字符串的参数，记过处理后返回另一个字符串，非常的简单。

```
package com.googlepages.smallnest.facet;

public interface HelloService
{
        public String hello(String name);
}
```

下面你的任务就是要建立一个实现类 HelloServiceImpl，实现这个接口。

```
package com.googlepages.smallnest.facet;

public class HelloServiceImpl implements HelloService
{
	public String hello(String name)
	{
		if (null == name)
		{
			return "Hello Guest";
		}
		return "Hello " + name;
	}
}
```

这个实现类实现了 hello 方法，对于传入的参数 name，如果不为空，将简单和 Hello 字符串连接后返回，如果为空则返回“Hello Guest”。

这个例子绝对够简，在加你初始学 java 的时候，接触到的第一个例子和这个基本一样。并且可以负责任的告诉你，后面的例子都和这个一样的简单。

到目前为止你所做的开发和平常的 java 开发没有什么区别，都是普普通通的接口和 java 类。最后一步就是要编写一个 services.xml 文件，这个文件定义了要发布的 web services。

新建一个 xml 文件，保存为 services.xml，保存在 WEB-INF/src/META-INF/xfire 文件夹下，eclipse 会自动将它复制到 WEB-INF/classes/META-INF/xfire 下。XFireServlet 会读取并解析这个文件。

services.xml 文件的内容如下：

```
<beans>
  <service xmlns="http://xfire.codehaus.org/config/1.0">
    <name>HelloService</name>
    <namespace>http://smallnest.googlepages.com/HelloService</namespace>
    <serviceClass>com.googlepages.smallnest.facet.HelloService</serviceClass>
    <implementationClass>com.googlepages.smallnest.facet.HelloServiceImpl</implementationCla
ss>
  </service>
```

```
</beans>
```

这个文件定义了一个 Web Service:helloService，并定义 serviceClass 和 implementationClass，分别是 helloService 的接口和实现类。

Xf 采用 XBean(http://xbean.org)来处理这个文件，XBean 允许用户可以将自定义的语法和 Spring 的定义混合在一起，例如：

```
<beans xmlns="http://xfire.codehaus.org/config/1.0">
  <service>
    <name>Echo</name>
    <serviceBean>#echoBean</serviceBean>
  </service>

  <bean id="echoBean" class="org.codehaus.xfire.services.Echo"/>
</beans>
```

当然，你也可以完全采用 Spring 的语法定义：

```
<bean name="echoService" class="org.codehaus.xfire.spring.ServiceBean">
    <property name="serviceBean" ref="echo"/>
    <property name="serviceClass" value="org.codehaus.xfire.test.Echo"/>
    <property name="inHandlers">
      <list>
        <ref bean="addressingHandler"/>
      </list>
    </property>
</bean>
<bean id="echo" class="org.codehaus.xfire.test.EchoImpl"/>
<bean id="addressingHandler" class="org.codehaus.xfire.addressing.AddressingInHandler"/>
```

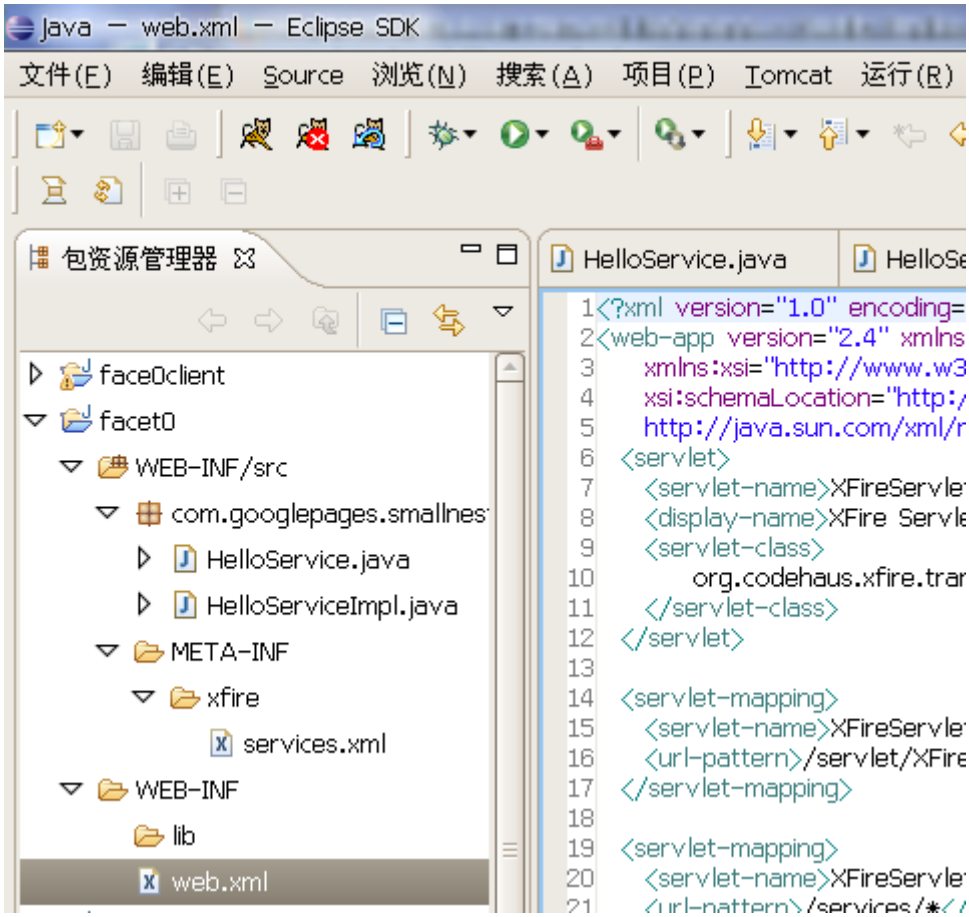
你还可以通过<import resource="classpath:org/codehaus/xfire/spring/xfire.xml"/>将一些 XFire 定义好的 Bean 引入到你的文件中,这个文件中定义了 TransportManager, ServiceRegistry以及一些简单的 ServiceFactory。

services.xml 的基本结构如下：

```
<beans xmlns="http://xfire.codehaus.org/config/1.0">
  <xfire>
    <inHandlers>
      <handler handlerClass=""> </handler>
    </inHandlers>
  </xfire>
  <service>
    <name/>
    <namespace/>
    <serviceClass/>
    <implementationClass/>
    <serviceFactory/>
    <bindingProvider/>
    <style>document|rpc|message|wrapped</style>
    <use>literal|encoded</use>
    <scope>request|session|application</scope>
    <invoker/>
    <executor/>
    <inHandlers>
      <handler handlerClass=""/></inHandlers>
    <outHandlers>
      <handler handlerClass=""/>
    </outHandlers>
    <faultHandlers>
      <handler handlerClass=""/></faultHandlers>
    <createDefaultBindings>true|false</createDefaultBindings>
    <bindings>
      <soap11Binding name="qname" transport="" allowUndefinedEndpoints="">
        <endpoints>
          <endpoint name="qname" url="" />
        </endpoints>
      </soap11Binding>
      <soap12Binding name="qname" transport="" allowUndefinedEndpoints="">
        <endpoints>
          <endpoint name="qname" url="" />
        </endpoints>
      </soap12Binding>
    </bindings>
  </service>
```

```
</beans>
```

启动 Tomcat：



一个 Web Service 就开发完成并发布了。

如果你采用 Spring 2.0+,注意不要把 xmlns="http://xfire.codehaus.org/config/1.0"设置在 root 中，而是设置在 service 节点上。

查看 Web Service

打开浏览器，访问 http://localhost:8080/facet/services，你会看到下面的界面：

这说明 HelloService 发布成功了，点击后面的蓝色的链接，就会得到这个 Web Service 的 WSDL 信息。可以保存这个信息到文档 HelloService.wsdl 中备用。

消费 Web Service

你已经成功创建并发布了一个 Web Service，恭喜你。作为 Web Service的一个完整的生产线，你已经完成了一半的工作。或许，剩下的一半的工作并不属于你的职责，抑或剩下的这一半工作才是你要做的工作：吃掉这个 Web Service。

Web Service被开发出来就是要人（其实是其它程序）用的。接下来这一节就是要带你完成消费这个 Web Service的过程。

很多情况下，比如有网友问，一个短信服务商通过 Web Service发布了一个短信接口，如何进行代码开发来使用这个短信服务？

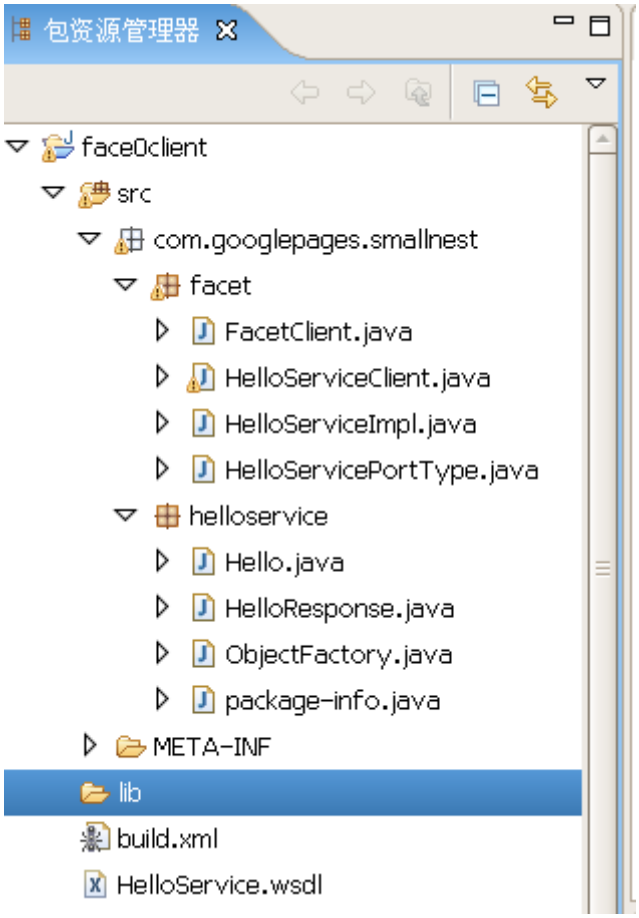
短信服务商既然提供了 Web Service，那么他就应该告诉你这个 Web Service的 WSDL，你可以通过这个 WSDL 文件生成访问这个 Web Service的代码。无论短信服务商采用何种语言，做了多少类多少代码完成这个 Web Service，跟你没有任何关系。你只要发送正确的消息给这个 Web Service，就会得到它正确的反馈文本。

在 eclipse 中新建一个 java 工程，并将 HelloService.wsdl存到这个工程的文件夹中。编写一个 build.xml 文件，采用 ant 自动生成客户端代码（同时也生成了服务器端的代码骨架，不过目前来说对我们无用，因为我们不创建服务器端代码）。

```
<?xml version="1.0" encoding="gb2312"?>
<project name="facet" default="help" basedir=".">

	<!-- ================================================================== -->
	<!-- 设置属性                                                           -->
	<!-- ================================================================== -->
	<property name="optimize" value="false" />
	<property name="debug" value="on" />
	<property name="deprecation" value="false" />

	<property name="build.lib" value="${basedir}/../lib" />
	<property name="sources" value="${basedir}/src" />
	<property name="build.classes" value="${basedir}/bin" />



	<!-- ================================================================== -->
	<!-- 设置类路径                                                         -->
```

```
	<!-- ================================================================== -->
	<path id="classpath">
		<pathelement location="${build.classes}" />
		<fileset dir="${build.lib}">
			<include name="*.jar" />
		</fileset>
	</path>

	<taskdef name="wsgen" classname="org.codehaus.xfire.gen.WsGenTask"
classpathref="classpath" />

	<!-- ================================================ -->
	<!-- 帮助                                             -->
	<!-- ================================================ -->
	<target name="help" description="显示帮助信息">
		<echo message="target                    描述" />
		<echo
message="------------------------------------------------------------------------" />
		<echo message="compile                   编译 " />
		<echo message="create_code               创建代码" />
	</target>


	<!-- ================================================ -->
	<!-- 编译代码                                         -->
	<!-- ================================================ -->
	<target name="compile" description="编译代码">
		<echo>编译程序代码</echo>
		<javac srcdir="${sources}" destdir="${build.classes}" classpathref="classpath"
debug="${debug}" optimize="${optimize}" deprecation="${deprecation}" />
	</target>

	<!-- ================================================ -->
	<!-- 创建客户端代码                                   -->
	<!-- ================================================ -->
	<target name="create_code" description="创建代码">
		<echo>创建代码</echo>
		<wsgen outputDirectory="${sources}" wsdl="${basedir}/HelloService.wsdl"
package="com.googlepages.smallnest.facet" overwrite="true" />
	</target>

</project>
```

执行 create_code target，wsgen 就会根据 wsdl 文件自动生成代码。



XFire 提供了 Eclipse 的插件，可以通过向导一步步生成所需的代码。本书中未采用插件，而是通过 Ant 来生成。

不必关心生成的代码，无论 AXIS 还是 XFire 还是 DOTNET 等，都会自动帮你创建访问 Web Services 的代码，提供给你的接口非常的简单，就象访问本地的 Java Bean 一样。

新建一个 FacetCleint 类，在这个类中你可以使用创建的代码访问 Web Service。

```
package com.googlepages.smallnest.facet;

public class FacetClient
{
```

```
        //使用创建的客户端代码访问HelloService
        public static void main(String[] args)
        {
                HelloServiceClient client = new HelloServiceClient();
                HelloServicePortType helloService = client.getHelloServiceHttpPort();

                //调用服务
                String  result = helloService.hello("Juliet");
                System.out.println("结果: " + result);

        }
}
```

真正的代码就三行，先得到一个 HelloServiceClient 对象，然后得到 HelloServicePortType 接口，就象调用本地的 Javabean 一样调用这个接口的 hello 方法，就可以得到反馈的结果。

就这么简单！

以下每个章节的 Web Service 都可以通过这种方式来进行消费。

本章中的例子传入的参数是 String 类型的，返回的结果也是 String 类型的，如果是参数或返回值自定义类型，如何实现？

2 嵌入 XFire 服务

本章的内容包括：

- 在桌面应用程序中集成 XFire

Xfire 可以部署到多种应用服务器(Web 服务器)中，包括但不限于 Tomcat、Weblogic Server 8.1、Weblogic Server 9.2、IBM Websphere 6.x、Orion 2.0.7、Resin 、OC4J 10.1.3.x等。

事实上，XFire 也可以被应用到 C/S 架构的应用程序中，可以在桌面应用程序中发布 Web Services。XFire 可以通过内置的 Jetty 作为内部的 Web 服务器进行发布 Web Services,所以确保你的类路径中包含了 Jetty。

在 eclipse 中新建一个 Java 工程，和上一章一样，这次你仍然要增加一个 HelloService 接口和实现类 HelloServiceImpl。

HelloService.java

```
package com.googlepages.smallnest.facet;

public interface HelloService
{
	public String Hello(String name);
}
```

HelloServiceImpl.java

```
package com.googlepages.smallnest.facet;
```

```
public class HelloServiceImpl implements HelloService
{

	public String Hello(String name)
	{
		if (null == name)
		{
			return "Hello Guest";
		}
		return "Hello " + name;
	}

}
```

现在，你可以新建一个 EmbeddedServer 类来模拟一个 Web 服务器。

```
package com.googlepages.smallnest.facet;

import org.codehaus.xfire.XFire;
import org.codehaus.xfire.XFireFactory;
import org.codehaus.xfire.server.http.XFireHttpServer;
import org.codehaus.xfire.service.Service;
import org.codehaus.xfire.service.binding.ObjectServiceFactory;
import org.codehaus.xfire.service.invoker.BeanInvoker;

public class EmbeddedServer
{
	XFireHttpServer server;

	public boolean start()
	{
		ObjectServiceFactory serviceFactory = new ObjectServiceFactory();
	 Service service = serviceFactory.create(HelloService.class);
	 service.setInvoker(new BeanInvoker(new HelloServiceImpl()));

	 //注册服务
	 XFire xfire = XFireFactory.newInstance().getXFire();
	 xfire.getServiceRegistry().register(service);

	 //启动服务器
```

```
        server = new XFireHttpServer();
        server.setPort(8191);
        try
              {
                    server.start();
              }
              catch (Exception e)
              {
                    return false;
              }

              return true;
       }




       public static void main(String[] args)
       {
              EmbeddedServer server = new EmbeddedServer();

              erver.start();
       }

}
```

在这个类中,start 方法启动服务，stop 方法停止服务。首先得到一个 ObjectServiceFactory 对象，通过这个对象创建一个服务 HelloService，并指定访问这个服务时实际要调用的实现类。接着注册了这个服务。启动内置的服务器，并将端口设置为 8191。运行这个程序，打开浏览器，访问 http://localhost:8191/，可以看到下面的界面。

简单客户端访问

本章演示了如何通过 Client 类来访问 Web 服务。

第一章介绍访问 HelloService 的客户端代码时，是通过 Ant 自的创建了客户端的访问代码，然后调用这些代码就可以访问 HelloService 了。在本章中，演示了另外一种方法，无须创建客户端代码，只需通过 Client 就可以调

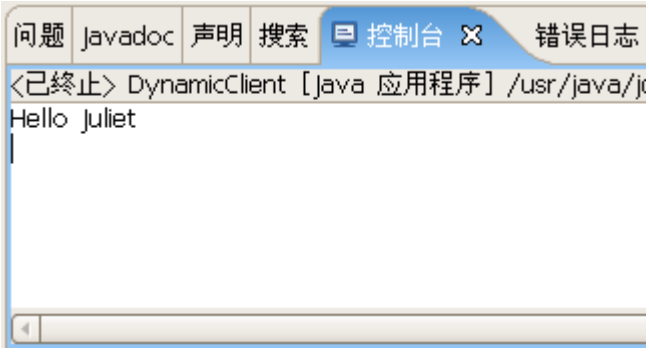
首先启动一个 HelloService 服务。我们可以采用第一章的例子，把 tomcat 启动起来，这样就可以访问 HelloService。你要做的工作就是通过 Client 类访问这个服务。

在 eclipse 中新建一个 Java 工程，并新建类 DynamicClient。

```
package com.googlepages.smallnest.facet;

import java.net.MalformedURLException;
import java.net.URL;

import org.codehaus.xfire.client.Client;

public class DynamicClient
{

	public static void main(String[] args) throws MalformedURLException, Exception
	{
		Client client = new Client(new
URL("http://localhost:8080/facet/servlet/XFireServlet/HelloService?wsdl"));
		Object[] results = client.invoke("hello", new Object[] {"Juliet"});
		System.out.println((String) results[0]);
	}

}
```

Client 只需一个参数，这个参数代表一个 Web Service WSDL 的 URL，然后调用 Client

的 Invoke 方法，这个方法的第一个参数是你要调用的方法(hello)，第二个参数代表了 hello 方法所需的参数，这样就可以得到 Web Service 的结果了。



如果传入的参数是自定义类型的，传出的参数也是复杂类型的，如何实现？

JSR181

本章的内容包括：

- JSR181 介绍
- 编写符合 JSR181 规范的 Web Services
- 参数为自定义类型
- 返回结果为集合类型
- 一种利用 header 传送验证信息的方法

JAVA 5.0 中引入了元数据(metadata)的特性，利用这一特性，可以直接为 JAVA 代码添加一些特别的注释。不再需要通过 XML 文件进行配置，减少 XML 配置文件的应用。 比如 EJB3.0、Hinernate3.0 等都引入了通过注释将 POJO 映射成 EJB 或者可持久化的对象。JSR181 制定了规范，提供了一套注释 Web Service 的方法，方便 web service 的开发。其实这也不是什么新技术，微软在 DOTNET 框架中早就提供属性的功能（和注视类似），利用属性就可以标记普通的类成为 Web Service。

JSR181 规范可以到 http://www.jcp.org 下载。

在本章的例子中，提供了一个 MemberService，你可以把它理解成一个会员注册服务。Web Service 的消费客户端可以上传一个 Member 对象，也可以查询会员列表。在调用 Web Service 服务的方法时，需要进行身份验证。这样容易理解，如果你是这个 Web Service 的提供商，你当然只希望你的合作伙伴使用，每个合作伙伴都通过用户名和密码来进行验证。

开发服务器端代码

在 eclipse 中新建一个 Tomcat 工程。新建一个 Member 类，这个类代表了一个会员，会员有姓名、电子邮箱、级别三个属性。这个类就是一个普通的 JavaBean。

```
package com.googlepages.smallnest.facet;

public class Member
{
	private String name;
	private String email1;
	private int rank;

	public void setName(String name)
	{
		this.name = name;
	}
	public String getName()
	{
		return name;
	}
	public void setEmail1(String email1)
	{
		this.email1 = email1;
	}
	public String getEmail1()
	{
		return email1;
	}
	public void setRank(int rank)
	{
		this.rank = rank;
	}
	public int getRank()
	{
		return rank;
	}
}
```

新建一个 User 类，也是一个普通的 JavaBean，用来映射 SOAP Header 中的身份验证信息。

```
package com.googlepages.smallnest.facet;

public class User
```

```
{
        private String username;
        private String password;
       public void setUsername(String username)
       {
              this.username = username;
       }
       public String getUsername()
       {
              return username;
       }
       public void setPassword(String password)
       {
              this.password = password;
       }
       public String getPassword()
       {
              return password;
       }
}
```

下面可以编写你的 MemberService类了。在这个例子中，不再使用接口和实现类的方式（当然你也可以这么做），而是直接实现一个服务类。

```
package com.google1pages.small1nest.facet;

import java.util1.ArrayList;
import java.util1.Collection;
import java.util1.List;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;

@WebService
public class MemberService
{
       private List<Member> memebers = new ArrayList<Member>();

       @WebMethod
       @WebResult(name = "Members")
```

```
        public Collection<Member> getMembers(@WebParam(name = "User", header=true)User user)
        {
                authorize(user);
                return memebers;
        }


        @WebMethod
    public boolean addMember(@WebParam(name="User", header=true) User user,
                             @WebParam(name="member") Member member)
    {
                if (!authorize(user))
                {
                        return false;
                }
                memebers.add(member);
         return true;
    }

        private boolean authorize(User user)
    {
         System.out.println(user.getUsername());
         System.out.println(user.getPassword());
         return true;
    }
}
```

如果你还不了解 JAVA 的注释功能，建议你先找些资料学习一下。WebService、WebMethod、WebResult、WebParam这些注释的名字都非常的直观，在 user 参数的 WebParam 中定义了 header=true，说明这个 user 的信息是从 SOAP Header 中得到。在执行每个 WebMethod 时，首先先对 user 进行身份验证（其实这里也没做什么验证，只是将用户名和密码打印出来）。

在 WEB-INF/src/META-INF/xfire文件夹下新建 services.xml 文件，重申一下这个文件会被 Eclipse 复制到 WEB-INF/classes/META-INF/xfire文件夹下。services.xml文件的内容如下：

```
<beans>
  <service xmlns="http://xfire.codehaus.org/config/1.0">
```

```
    <serviceClass>com.googlepages.smallnest.facet.MemberService</serviceClass>
    <serviceFactory>jsr181</serviceFactory>
  </service>
</beans>
```

如何发布这个 Web Service?如何查看这个 Web Service 的 WSDL 信息？

编写客户端代码

和第一章一样发布这个 Web Service后，将 WSDL 信息保存为 MemberService.wsdl文件备用。

客户端的开发也和第一章一样，通过 Ant 自动生成代码。

新建一个 Java 工程，将 MemberService.wsdl复制到工程的文件夹下。Ant 所需的 build.xml 文件内容如下：

```
<?xml version="1.0" encoding="gb2312"?>
<project name="facet" default="help" basedir=".">

        <!-- ================================================================== --
>
        <!-- 设置属性                                                           -->
        <!-- ================================================================== --
>
        <property name="optimize" value="false" />
        <property name="debug" value="on" />
        <property name="deprecation" value="false" />

        <property name="build.lib" value="${basedir}/../lib" />
        <property name="sources" value="${basedir}/src" />
        <property name="build.classes" value="${basedir}/bin" />
```

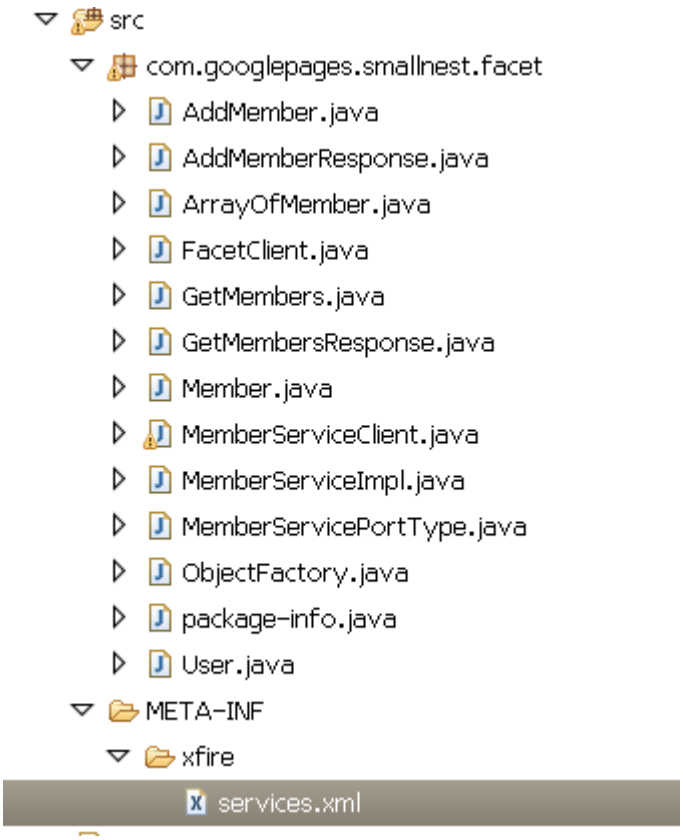
```
        <!-- ================================================================== -->
        <!-- 设置类路径                                                         -->
        <!-- ================================================================== -->
        <path id="classpath">
                <pathelement location="${build.classes}" />
                <fileset dir="${build.lib}">
                        <include name="*.jar" />
                </fileset>
        </path>

        <taskdef name="wsgen" classname="org.codehaus.xfire.gen.WsGenTask"
classpathref="classpath" />

        <!-- ================================================ -->
        <!-- 帮助                                             -->
        <!-- ================================================ -->
        <target name="help" description="显示帮助信息">
                <echo message="target                描述" />
                <echo
message="-------------------------------------------------------------------------" />
                <echo message="compile               编译 " />
                <echo message="create_code           创建代码" />
        </target>


        <!-- ================================================ -->
        <!-- 编译代码                                         -->
        <!-- ================================================ -->
        <target name="compile" description="编译代码">
                <echo>编译程序代码</echo>
                <javac srcdir="${sources}" destdir="${build.classes}" classpathref="classpath"
debug="${debug}" optimize="${optimize}" deprecation="${deprecation}" />
        </target>

        <!-- ================================================ -->
        <!-- 创建代码                                         -->
        <!-- ================================================ -->
        <target name="create_code" description="创建代码">
                <echo>创建代码</echo>
```

```
            <wsgen outputDirectory="${sources}" wsdl="${basedir}/MemberService.wsdl"
package="com.googlepages.smallnest.facet" overwrite="true" />
        </target>

</project>
```

运行 create_code target，生成了客户端和服务器端代码，这里服务器端的代码早以完成，所以不必理会，不过你可以对比一下自动代码和上一节的服务器端的代码的异同。

自动创建的代码的绑定(binding)方式为 JAXB2，实现方式为接口+实现类的方式，通过 JSR181 注释的方式声明 Web Service。

自动代码创建了下面的文件，除了 FacetClient。



新建 FacetClient 类，这个类将使用前面创建的代码访问 Web Service。

```
package com.googlepages.smallnest.facet;

import java.util.List;
```

```
import javax.xml.bind.JAXBElement;
import javax.xml.namespace.QName;

public class FacetClient
{
	public static void main(String[] args)
	{
		MemberServiceClient client = new MemberServiceClient();
		MemberServicePortType memberService = client.getMemberServiceHttpPort();

		User user = new User();
		JAXBElement<String> name = new JAXBElement<String>(new
QName("http://facet.smallnest.googlepages.com", "username"),String.class,"smallnest");
		JAXBElement<String> password = new JAXBElement<String>(new
QName("http://facet.smallnest.googlepages.com", "password"),String.class,"123456");
		user.setUsername(name);
		user.setPassword(password);


		Member oneMember = new Member();
		JAXBElement<String> memeberName = new JAXBElement<String>(new
QName("http://facet.smallnest.googlepages.com", "name"),String.class,"迈克");
		JAXBElement<String> email = new JAXBElement<String>(new
QName("http://facet.smallnest.googlepages.com", "email"),String.class,"micro@micro.com");
		oneMember.setName(memeberName);
		oneMember.setEmail(email);
		oneMember.setRank(5);

		//增加一个会员
		memberService.addMember(oneMember, user);

		//获取会员列表
		List<Member> memebers = memberService.getMembers(user).getMember();
		for (Member member : memebers)
		{
			System.out.println(member.name.getValue());
		}

	}

}
```

因为绑定方式为 JAXB2，所以产生 User 和 Memeber 的实例稍微有些繁琐，得到

memberService 实例后就可以调用它的 addMemeber方法和 getMemeber方法，getMemeber方法返回的结果是一个 List 对象，这个列表对象是 Memeber 类型的对象列表。

运行这个程序，测试一下结果。

自动创建的服务器端代码的绑定方式是 JAXB2 方式，而第一节中你自己创建的代码并未采用 JAXB2，那是什么绑定方式？你喜欢什么样的绑定方式？

Aegis 绑定

本章的内容包括：

- 两种 Web Services 开发方式
- Aegis 在 JAVA1.4 和 JAVA 5 下的配置
- 集合的映射
- 自定义类型

Aegis 是 XFire 的缺省的绑定方式,可以将 XML 映射成 POJO。开发一个 Web Sercie 有两种方式，一种是代码优先（code first），先开发你的 POJO，设置为 Web Service，然后得到它的 XML schema / WSDL,另外一种是 schema 优先(schema fisrt)，先设计（或得到）一个 XML schema/WSDL，然后再进行编码，发布 Web Service。这又分两种情况，一种是只有对象类型的 schema，可以通过 XML 映射框架生成 Java 对象代码，如 Castor、JiBX、JAXB2、XMLBeans 等，然后再进行 Web Service 的开发，一种是根据 WSDL 直接生成所需的对象和 Web Service 的骨架。Aegis 只支持代码优先的开发方式。

Bingding 这个词可能在其它地方有不同的方式，这里翻译成**绑定**，既照顾到了意思的翻译，发音也类似。这里所谓的绑定，就是 XML 文档和 POJO 对象的映射方式。通过绑定，你可以实现将自定义类型的对象映射成 XML 文档，通过 SOAP 进行传输。

Aegis 有一套可以灵活的控制 Java Bean 的映射。缺省情况下你的 POJO 将基于命名空间(namespace)和名称进行序列化（serialized）。例如你在 package com.googlepages.smallnest.facet 下有一个类 User：

```
package com.googlepages.smallnest.facet;

public class User
{
        private String username;
```

```
        private String password;
        public void setUsername(String username)
        {
                this.username = username;
        }
        public String getUsername()
        {
                return username;
        }
        public void setPassword(String password)
        {
                this.password = password;
        }
        public String getPassword()
        {
                return password;
        }
}
```

转换成 XML 方式为：

```
<User xmlns=" http://facet.smallnest.googlepages.com" >
   <name>smallnest</name>
   <password>mypass</password>
</User>
```

XML schema 为:

```
<xsd:complexType name="User">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string" minOccurs="0" maxOccurs="1/>
    <xsd:element name="password" type="xsd:string" minOccurs="0" maxOccurs="1/>
  </xsd:sequence>
</xsd:complexType>
```

Aegis 支持的类型包括：

- 基本类型：int,double,float,long,byte[],short,String,decimal
- 数组(Array)

- 集合类型(Collection)
- 日期类型:java.util.Date, java.util.Calendar, java.sql.Timestamp, java.sql.Date, java.sql.Time
- XML:org.w3c.dom.Docmument, org.jdom.Element, XMLStreamReader,XML 文本
- 以上类型组合成的自定义类型

如果你使用自定义类型，自定义类型必须有一个默认的构造方法，也就是不带参数的构造方法。

映射文件

如果你还在使用 JDK1.4 ，你可以使用映射文件控制 POJO 和 XML 的映射。比如你在 package com.googlepages.smallnest.facet 下有一个 User 需要映射，那么在这个 package 下新建 User.aegis.xml 文件，格式如下：

```
<mappings>
  <mapping uri="" name="">
    <method name="methodName">
      <return-type mappedName="" componentType=""/>
      <parameter index="" mappedName=""/>
    </method>
    <property name="" mappedName="" style="attribute|element" componentType=""/>
  </mapping>
</mappings>
```

1. 控制命名

假设你的映射控制如下：

```
<mappings xmlns:my="http://my.smallnest.googlepages.com">

  <mapping name="my:User">
    <property name="name" mappedName="Name"/>
    <property name="password" mappedName="Password"/>
  </mapping>
```

```
</mappings>
```

映射的结果如下：

```
<my:User xmlns:my="http://my.smallnest.googlepages.com">

  <my:Name>smallnest</my:Name>
  <my:Password>mypass</my:Password>
</my:User>
```

2. 忽略属性

如果你不想某个属性被序列化，只需指定 ignore 属性。

```
<mappings xmlns:my="http://my.smallnest.googlepages.com">

  <mapping name="my:User">
    <property name="name" mappedName="Name"/>
    <property name="password" ignore=" true" />
  </mapping>
</mappings>
```

3. minOccurs 和 nillable

因为 Java 对象可以被设置为 null，所以 Aegis 默认映射的属性 minOccurs 为 0，nillable 为 true，你可以在映射文件中进行修改。

```
<mappings xmlns:my="http://my.smallnest.googlepages.com">

  <mapping name="my:User">
    <property name="name" mappedName="Name"  minOccurs="1"  nillable="false"/>
    <property name="password" mappedName="Password"/>
  </mapping>
</mappings>
```

4. 从 java 代码中控制 minOccurs 和 nillable

如果你有多个映射文件，你想把这些文件中的 minOccurs 属性都设为 1,nillable 属性都设为 false，那么可以通过 Java 代码简单的控制。

```
AnnotationServiceFactory serviceFactory = new AnnotationServiceFactory();

AegisBindingProvider binder = (AegisBindingProvider)serviceFactory.getBindingProvider();
DefaultTypeMappingRegistry tmr =
(DefaultTypeMappingRegistry)binder.getTypeMappingRegistry();

Configuration configuration = tmr.getConfiguration();
configuration.setDefaultMinOccurs(1);
configuration.setDefaultNillable(false);
```

通过注释

如果你使用 JAVA 5，那么你可以通过注释而不是映射文件的方式控制 POJO 的映射。一个简单的注释例子如下：

```
import org.codehaus.xfire.aegis.type.java5.XmlElement;

@XmlType(namespace="http://facet.smallnest.googpages.com")
public class User
{
  private String name;
  private String password;

  @XmlElement(name="Name", namespace="http://facet.smallnest.googpages.com")
  public String getName() { return name; }
  public void setName(String name) { this.name = name; }

  @XmlElement(name="Password", namespace="http://facet.smallnest.googpages.com")
  public String getPassword() { return password; }
  public void setPassword(String password) { this.password = password; }
}
```

注释设置在属性的 get 方法上。

一个 User 对象序列化为：

```
<my:User xmlns:my="http://my.smallnest.googlepages.com">

  <my:Name>smallnest</my:Name>
  <my:Password>mypass</my:Password>
</my:User>
```

你也可以使用：

```
import org.codehaus.xfire.aegis.type.java5.XmlElement;

@XmlType(namespace="http://facet.smallnest.googpages.com")
public class User
{
  private String name;
  private String password;

  @XmlAttribute(name="Name", namespace="http://facet.smallnest.googpages.com")
  public String getName() { return name; }
  public void setName(String name) { this.name = name; }

  @XmlElement(name="Password", namespace="http://facet.smallnest.googpages.com")
  public String getPassword() { return password; }
  public void setPassword(String password) { this.password = password; }
}
```

一个 User 对象序列化为：

```
<my:User my:Name=" smallnest"   xmlns:my="http://my.smallnest.googlepages.com">

  <my:Password>mypass</my:Password>
</my:User>
```

你也可以指定哪个属性被忽略：

```
import org.codehaus.xfire.aegis.type.java5.Xml1Element;

@Xml1Type(namespace="http://facet.sma11nest.googpages.com")
public class User
{
  .....

  @IgnoreProperty
  public String getPassword() { return password; }
  public void setPassword(String password) { this.password = password; }
}
```

继承

当前 XFire 支持 POJO 的继承，但不支持接口的继承。

集合

很多情况下，Web Services 需要返回一个几何类型，这会带来一个问题：集合类型包含的对象类型是 Object 类型的，Aegis 如何能够聪明的得到实际的对象类型？

如果你采用 JAVA 5，这不是个问题，你可以采用集合的 Generic 形式，如 Collection<User>来明确集合包含的类型。

如果你采用 JAVA1.4，可以通过映射文件类配置。例如一个 Web Service：

```
public interface MyServicel

{
    String getFoo();
    Collection getCollection();
    void setList(int id, java.util.List);
}
```

映射文件 MyService1.agis.xml 为

```
<mappings>
```

```
    <mapping>
        <method name="getCollection">
            <return-type componentType="java.lang.String"/>
        </method>
        <method name="setList">
            <parameter index="1" componentType="java.lang.String"/>
        </method>
    </mapping>
</mappings>
```

注意 setList 只设置了第二个参数，Aegis 允许你只设置不按默认控制的属性和方法，参数的类型为 String，实际是指 List 包含的对象类型为 String。

集合可以为重载的方法设映射规则。映射规则为最匹配原则。

集合映射也可以应用在 JavaBean 上。通过<property componentType="a class" />来设置。

Map 类型也可以进行映射，需要增加 keyType 属性。

集合包含的元素如果还是集合，可以通过下面的方式。假设有下面一个 Web Service：

```
public class ListService

{
  public List getListOfListOfDoubles
  {
    List l = new ArrayList();
    List doubles = new ArrayList();

    doubles.add(new Double(1.0));

    l.add(doubles);
    return l;
  }
```

```
}
```

可以这样映射(注意返回类型的 componentType 的设置)：

```
<mappings>

  <mapping>
    <method name="getListofListofDoubles">
      <return-type componentType="#someDoubles"/>
    </method>
    <component name="someDoubles" class="java.util.List" componentType="java.lang.Double" />
  </mapping>
</mappings>
```

自定义类型

通过基本类型的组合，你可以实现基本的 JavaBean 的映射。如果还不满足你的要求，比如自定义类型的多次复合或者你自己来处理这些数据而不是加载到内存中去映射，那么你可以自己定义一种类型,比如对于一个 Web Method:

```
public ReferenceToData doSomething(…) {}
```

为 ReferenceToData 创建一个类型：

```
public class ReferenceToDataType extends Type

{
  public ReferenceToDataType() {
    setTypeClass(ReferenceToData.class);
    setSchemaType(new QName(.. the QName of the type you're returning ..));
  }

  public void writeObject(Object value, XMLStreamWriter writer, MessageContext context)
  {
    ReferenceToData data = (ReferenceToData) value;
    … do you're writing to the writer
  }

  public Object readObject( MessageReader reader, MessageContext context )
  {
```

```
    // If you're reading you can read in a reference to the data
    XMLStreamReader reader = context.getInMessage().getXMLStreamReader();

    ReferenceToData data = read(reader);
    return data;
  }

  public void writeSchema(Element schemaRoot)
  {
    // override this to write out your schema
    // if you have it in DOM form you can convert it to YOM via DOMConverter
  }
}
```

注册这个类型

```
ServiceRegistry serviceRegistry  = ....; // get this from the XFire instance

Serivce service = serviceRegistry.getService("serviceName");

TypeMapping tm = ((AegisBindingProvider) service.getBindingProvider).getTypeMapping(service);
tm.register(new ReferenceToDataType());
```

如果通过 services.xml 进行配置，需要在 services.xml 中定义：

```
<beans xmlns="http://xfire.codehaus.org/config/1.0">

  <!-- This calls initializeTypes before your service is created -->
  <bean id="TypeRegistrar" init-method="initializeTypes" class="foo.TypeRegistrar ">
    <property name="typeMappingRegistry" ref="xfire.typeMappingRegistry"/>
  </bean>

  <service id="MyService">
    <serviceClass>foo.MyService</serviceClass>
  </service>
</beans>
```

还需创建一个类：

```
public class TypeRegistrar {

  private TypeMappingRegistry typeMappingRegistry;

  public void setTypeMappingRegistry (TypeMappingRegistry typeMappingRegistry) {
    this.typeMappingRegistry= typeMappingRegistry;
  }

  public void initializeTypes() {
    TypeMapping tm = typeMappingRegistry.getDefaultTypeMapping();
    tm.register(new ReferenceToDataType());
  }
}
```

本章和下面几章很大程度上参考了 XFire 的 User' Guide，你可以看 User' Guide 文档了解更多的知识。

Xfire 提供的 Ant task:wsgen 提供两种映射方式创建代码:JAXB2 和 XMLBeans。为什么没有提供 Aegis?

JAXB2 绑定(基于代码)

本章的内容包括：

- JAXB2 基于代码的开发

JAXB 是一种将 POJO 和 XML 映射的规范，现在已经发展到 2.0 了，你可以到 https://jaxb.dev.java.net/了解 JAXB 更多的信息。XFire 支持 JAXB1.1 和 JAXB2.0 的绑定，既然 JAXB 最新版本是 2.0，本书将只介绍 JAXB2.0 的

使用 JAXB2.0 绑定可以有两种方式开发：代码优先和 schema 优先。本章先介绍代码优先方式。

还是使用会员注册 MemberService 的例子：创建三个类 MemberService、MemberServiceImpl 和 Member。你可以通过注释控制 Member 的命名空间和名称。

MemberService.java:

```
package com.googlepages.small1nest.facet;

import java.util.Collection;

import javax.jws.WebService;

@WebService
public interface MemberService
{

        public abstract Collection<Member> getMembers();

        public abstract boolean addMember(Member member);

}
```

MemberServiceImpl.java:

```
package com.googlepages.smallnest.facet;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

import javax.jws.WebService;

@WebService
public class MemberServiceImpl implements MemberService
{
        private List<Member> memebers = new ArrayList<Member>();


        public Collection<Member> getMembers()
        {
                return memebers;
        }


    public boolean addMember(Member member)
    {
                memebers.add(member);
         return true;
    }


}
```

Memeber.java:

```
package com.googlepages.smallnest.facet;

import org.codehaus.xfire.aegis.type.java5.XmlElement;
import org.codehaus.xfire.aegis.type.java5.XmlType;

@XmlType(namespace="http://smallnest.googlepages.com")
public class Member
```

```
{
        private String name;
        private String email;
        private int rank;

        @XmlElement(name="name", namespace="http://smallnest.googlepages.com")
        public void setName(String name)
        {
                this.name = name;
        }
        public String getName()
        {
                return name;
        }

        @XmlElement(name="email", namespace="http://smallnest.googlepages.com")
        public void setEmail(String email)
        {
                this.email = email;
        }
        public String getEmail()
        {
                return email;
        }

        @XmlElement(name="rank", namespace="http://smallnest.googlepages.com")
        public void setRank(int rank)
        {
                this.rank = rank;
        }
        public int getRank()
        {
                return rank;
        }
}
```

这个例子比第三章中的例子更简单，因为这里不需要进行用户的验证。其它看起来和第三章没有什么两样。不同之处在于 services.xml 文件的配置：

```
<beans>
  <service xmlns="http://xfire.codehaus.org/config/1.0">
    <name>MemberService</name>
```

```
    <namespace>http://smallnest.googlepages.com/MemberService</namespace>
    <serviceClass>com.googlepages.smallnest.facet.MemberService</serviceClass>
    <implementationClass>com.googlepages.smallnest.facet.MemberServiceImpl</implementationC
lass>
    <serviceFactory>org.codehaus.xfire.jaxb2.JaxbServiceFactory</serviceFactory>
  </service>
</beans>
```

这里指定了 serviceFactory 为 jaxb2.JaxbServicefactory。

关于继承

你可以将你的 JAXB 数据模型设计成为多态的，比如父类为 Shape，子类有 Square、Triangle、Circle 等。假如一个 Web Service 的方法返回 Shape 类型的对象，而你有时可能返回以子类的对象，你必须要让 Jaxb2 知道这些相关的类，可以通过下面的代码实现：

```
List searchPackages = new ArrayList();
searchPackages.add("com.someshape.squares");
searchPackages.add("com.someshape.triangles");
searchPackages.add("com.someshape.circles");

Service service = …;
service.setProperty(JaxbType.SEARCH_PACKAGES, searchPackages);
```

将子类所在的 package 加入到 JaxbType.SEARCH_PACKAGES属性中。或者通过 services.xml 文件进行配置：

```
<service>
…
<property name="jaxb.search.packages">
  <list>
   <entry>com.acme.square</entry>
  </list>
</property>
</service>
```

JAXB2 绑定(基于 schema)

本章的内容包括：

- JAXB2 基于 schema 的开发

上一章介绍了采用代码优先的方式进行开发。在本章中介绍如何根据一个给定的 schema 来创建 Web Service。

本章将访问一个外部的 Web Service:Weather Service,地址为 http://www.webservicex.net/WS/WSDetails.aspx?CATID=12&WSID=68。数据类型的 schema 为：

```
<s:schema elementFormDefault="qualified" targetNamespace="http://www.webservicex.net"
          xmlns:s="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://www.webservicex.net">
    <s:element name="GetWeatherByZipCode">
        <s:complexType>
            <s:sequence>
                <s:element minOccurs="0" maxOccurs="1" name="ZipCode" type="s:string"/>
            </s:sequence>
        </s:complexType>
    </s:element>
    <s:element name="GetWeatherByZipCodeResponse">
        <s:complexType>
            <s:sequence>
                <s:element minOccurs="1" maxOccurs="1" name="GetWeatherByZipCodeResult"
                           type="tns:WeatherForecastsType"/>
            </s:sequence>
        </s:complexType>
    </s:element>
    <s:complexType name="WeatherForecastsType">
        <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="Latitude" type="s:float"/>
            <s:element minOccurs="1" maxOccurs="1" name="Longitude" type="s:float"/>
            <s:element minOccurs="1" maxOccurs="1" name="AllocationFactor" type="s:float"/>
```

```
            <s:element minOccurs="0" maxOccurs="1" name="FipsCode" type="s:string"/>
            <s:element minOccurs="0" maxOccurs="1" name="PlaceName" type="s:string"/>
            <s:element minOccurs="0" maxOccurs="1" name="StateCode" type="s:string"/>
            <s:element minOccurs="0" maxOccurs="1" name="Status" type="s:string"/>
            <s:element minOccurs="0" maxOccurs="1" name="Details"
type="tns:ArrayOfWeatherData"/>
        </s:sequence>
    </s:complexType>
    <s:complexType name="ArrayOfWeatherData">
        <s:sequence>
            <s:element minOccurs="0" maxOccurs="unbounded" name="WeatherData"
type="tns:WeatherData"/>
        </s:sequence>
    </s:complexType>
    <s:complexType name="WeatherData">
        <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="Day" type="s:string"/>
            <s:element minOccurs="0" maxOccurs="1" name="WeatherImage" type="s:string"/>
            <s:element minOccurs="0" maxOccurs="1" name="MaxTemperatureF" type="s:string"/>
            <s:element minOccurs="0" maxOccurs="1" name="MinTemperatureF" type="s:string"/>
            <s:element minOccurs="0" maxOccurs="1" name="MaxTemperatureC" type="s:string"/>
            <s:element minOccurs="0" maxOccurs="1" name="MinTemperatureC" type="s:string"/>
        </s:sequence>
    </s:complexType>
</s:schema>
```

事实上，我们可以得到 Weather Service 的 WSDL，通过 wsgen 这个 Ant Task 就可以生成所需的代码。现在换一个思路，或者说换一个需求。你现在只有这个数据对象的 schema，而且不让你设计 WSDL，而是自己设计一个 Web Service，实现 GetWeatherByZipCode方法。

首先，使用 XJC 工具根据这个 schema 生成相应的 POJO 文件，你可以到网站 https://jaxb.dev.java.net/下载这个工具。

设计一个接口 WeatherService:

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

```
import net.webservicex.GetWeatherByZipCode;
import net.webservicex.GetWeatherByZipCodeResponse;

@WebService(name="WeatherServiceIntf", targetNamespace="http://www.webservicex.net")
public interface WeatherService
{
    @WebMethod
    GetWeatherByZipCodeResponse GetWeatherByZipCode(@WebParam(name="GetWeatherByZipCode")
GetWeatherByZipCode body);
}
```

用 weatherServiceImpl 类实现这个接口：

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

import net.webservicex.GetWeatherByZipCode;
import net.webservicex.GetWeatherByZipCodeResponse;
import net.webservicex.WeatherForecastsType;

/**
 * @author <a href="mailto:dan@envoisolutions.com">Dan Diephouse</a>
 */
@WebService(endpointInterface="org.codehaus.xfire.jaxb.WeatherService",
serviceName="WeatherService")
@SOAPBinding(parameterStyle=SOAPBinding.ParameterStyle.BARE)
public class WeatherServiceImpl implements WeatherService
{

    public GetWeatherByZipCodeResponse GetWeatherByZipCode(GetWeatherByZipCode body)
    {
        GetWeatherByZipCodeResponse res = new GetWeatherByZipCodeResponse();
        String zipCode = body.getZipCode();

        WeatherForecastsType weather = new WeatherForecastsType();

        weather.setLatitude(1);
        weather.setLongitude(1);
        weather.setPlaceName("Vienna, AT");
        weather.setAllocationFactor(1);
```

```
        res.setGetWeatherByZipCodeResult(weather);

        return res;
    }
}
```

新建 services.xml 文件并进行配置：

```
<beans xmlns="http://xfire.codehaus.org/config/1.0">

  <bean id="weatherService" class="org.codehaus.xfire.jaxb.WeatherServiceImpl"/>

  <service>
    <serviceBean>#weatherService</serviceBean>
    <serviceFactory>#jaxbServiceFactory</serviceFactory>
    <schemas>
      <schema>META-INF/xfire/WeatherForecast.xsd</schema>
    </schemas>
    <style>document</style>
  </service>

  <bean name="jaxbServiceFactory" class="org.codehaus.xfire.jaxb2.JaxbServiceFactory">
    <constructor-arg ref="xfire.transportManager"/>
  </bean>

</beans>
```

不要将 JAXB1.1 和 JAXB2.0 的 jar 放在一起，否则有可能导致冲突，无法使用 JAXB2.0 的类。

XMLBeans 绑定

本章的内容包括：

- 基于 XmlBeans 绑定方式的开发

XmlBeans 是 BEA 贡献给 Apache 的一个开源项目。XmlBeans 是 XFire 支持的另外一种绑定方式，而且提供了 wsgen 辅助工具，wsgen 可以从一个 WSDL 文档生成客户端代码和服务器端代码。利用 XmlBeans 可以解析客户端的请求，也可以格式化对请求的相应文档。XmlBeans 的官方网站为 http://xmlbeans.apache.org。

开发一个基于 XmlBeans 绑定的 Web Servic 需要四步：

1. 编写或者获得 schema
2. 根据 schema 获得 XMLBeans(通过 XMLBeans 提供的工具)
3. 创建你的服务
4. 注册你的服务

假设你有一个 schema：Member.xsd

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" attributeFormDefault="qualified"
elementFormDefault="qualified" targetNamespace="http://facet.smallnest.googlepages.com">
<xsd:complexType name="Member">
<xsd:sequence>
<xsd:element minOccurs="0" name="email" nillable="true" type="xsd:string"/>
<xsd:element minOccurs="0" name="name" nillable="true" type="xsd:string"/>
<xsd:element minOccurs="0" name="rank" type="xsd:int"/>
</xsd:sequence>
</xsd:complexType>
<xsd:element name="addMember">
```

```
<xsd:complexType>
<xsd:sequence>
<xsd:element maxOccurs="1" minOccurs="1" name="member" nillable="true" type="tns:Member"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:element name="addMemberResponse">
<xsd:complexType>
<xsd:sequence>
<xsd:element maxOccurs="1" minOccurs="1" name="out" type="xsd:boolean"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:element name="getMembers">
<xsd:complexType/>
</xsd:element>
<xsd:complexType name="ArrayOfMember">
<xsd:sequence>
<xsd:element maxOccurs="unbounded" minOccurs="0" name="Member" nillable="true"
type="tns:Member"/>
</xsd:sequence>
</xsd:complexType>
<xsd:element name="getMembersResponse">
<xsd:complexType>
<xsd:sequence>
<xsd:element maxOccurs="1" minOccurs="1" name="Members" nillable="true"
type="tns:ArrayOfMember"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:complexType name="User">
<xsd:sequence>
<xsd:element minOccurs="0" name="password" nillable="true" type="xsd:string"/>
<xsd:element minOccurs="0" name="username" nillable="true" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>
<xsd:element name="User" type="tns:User"/>
</xsd:schema>
```

你可以通过 XMLBeans 根据这个 schema 创建所需的对象。

通过 wsgen 根据 WSDL 创建服务器端和客户端代码。

```
<?xml version="1.0" encoding="gb2312"?>
<project name="facet" default="help" basedir=".">

	<!-- ================================================================== --
>
	<!-- 设置属性                                                           -->
	<!-- ================================================================== --
>
	<property name="optimize" value="false" />
	<property name="debug" value="on" />
	<property name="deprecation" value="false" />

	<property name="build.lib" value="${basedir}/../lib" />
	<property name="sources" value="${basedir}/src" />
	<property name="build.classes" value="${basedir}/bin" />


	<!-- ================================================================== --
>
	<!-- 设置类路径                                                         -->
	<!-- ================================================================== --
>
	<path id="classpath">
		<pathelement location="${build.classes}" />
		<fileset dir="${build.lib}">
			<include name="*.jar" />
		</fileset>
	</path>

	<taskdef name="wsgen" classname="org.codehaus.xfire.gen.WsGenTask"
classpathref="classpath" />

	<!-- ================================================= -->
	<!-- 帮助                                              -->
	<!-- ================================================= -->
	<target name="help" description="显示帮助信息">
		<echo message="target                    描述" />
		<echo
message="-------------------------------------------------------------------" />
		<echo message="compile                   编译 " />
		<echo message="create_code               创建代码" />
	</target>
```

```
        <!-- ================================================= -->
        <!-- 编译代码                                          -->
        <!-- ================================================= -->
        <target name="compile" description="编译代码">
                <echo>编译程序代码</echo>
                <javac srcdir="${sources}" destdir="${build.classes}" classpathref="classpath"
debug="${debug}" optimize="${optimize}" deprecation="${deprecation}" />
        </target>

        <!-- ================================================= -->
        <!-- 创建客户端代码                                    -->
        <!-- ================================================= -->
        <target name="create_code" description="创建代码">
                <echo>创建代码</echo>
                <wsgen outputDirectory="${sources}" wsdl="${basedir}/MemberService.wsdl"
package="com.googlepages.smallnest.facet" binding="xmlbeans" overwrite="true" />
        </target>

</project>
```

注意 wsgen task 中设置了 binding 方式为 xmlbeans，默认为 JAXB2。执行 create_code target 得到 MemberServiceClient、MemberServiceImpl 和 MemberServicePortType 三个文件。MemberServicePortType 接口的内容如下：

```
package com.googlepages.smallnest.facet;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import org.apache.xmlbeans.XmlObject;

@WebService(name = "MemberServicePortType", targetNamespace =
"http://facet.smallnest.googlepages.com")
@SOAPBinding(use = SOAPBinding.Use.LITERAL, parameterStyle =
SOAPBinding.ParameterStyle.WRAPPED)
public interface MemberServicePortType {


    @WebMethod(operationName = "addMember", action = "")
```

```
    @WebResult(name = "out", targetNamespace = "http://facet.small1nest.google1pages.com")
    public Boolean addMember(
        @WebParam(name = "member", targetNamespace =
"http://facet.small1nest.google1pages.com")
        XmlObject member,
        @WebParam(name = "User", targetNamespace = "http://facet.small1nest.google1pages.com",
header = true)
        XmlObject User);

    @WebMethod(operationName = "getMembers", action = "")
    @WebResult(name = "Members", targetNamespace = "http://facet.small1nest.google1pages.com")
    public XmlObject getMembers(
        @WebParam(name = "User", targetNamespace = "http://facet.small1nest.google1pages.com",
header = true)
        XmlObject User);

}
```

注意方法的参数和返回值都是 XmlObject 对象。在实现类中可以调用前面创建的 XMLBeans 进行业务处理。

最后需要配置 services.xml：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xfire.codehaus.org/config/1.0">
  <service>
    <serviceClass>com.google1pages.small1nest.facet.MemberServiceImpl</serviceClass>
    <wsdlURL>file:/root/research/facet07/MemberService.wsdl</wsdlURL>
    <serviceFactory>#xmlbeansServiceFactory</serviceFactory>
  </service>
<bean name="xmlbeansServiceFactory"
class="org.codehaus.xfire.xmlbeans.XmlBeansServiceFactory">
    <constructor-arg ref="xfire.transportManager"/>
  </bean>
</beans>
```

Castor 绑定

本章的内容包括：

- 基于 castor 绑定方式的开发

Castor 是另外一种比较著名的 XML-POJO 映射框架。它提供了一种灵活的接口，即可以在代码也可以通过映射文件实现 XML 和 POJO 的映射 Castor 与其它映射框架最大的不同在于一旦映射文件改变并不需要重新编译 JAVA 代码。Castor 的官方网站为 http://www.castor.org/。

Castor 支持代码优先和 schema 优先的开发方式。

值得注意的是，Caster 不能直接映射下面两种类型的 XML:

```
<person>
  <givenName>John</givenName>
  <surname>Doe</surname>
</person>

<person>
  <givenName>Jane</givenName>
  <surname>Doe</surname>
</person>
```

和

```
<persons>
  <person>
    <givenName>John</givenName>
    <surname>Doe</surname>
  </person>

  <person>
```

```
    <givenName>Jane</givenName>
    <surname>Doe</surname>
  </person>
</persons>
```

如果要映射这种类型，就需要创建一个容器：

```
public class Persons {
  private List persons = new ArrayList();

  public List getPersons() {
      return persons;
  }

  public boolean addPerson(Person person) {
      return persons.add(person);
  }

  public void setPersons(List persons) {
      this.persons = persons;
  }

  public boolean removePerson(Person person) {
      return persons.remove(person);
  }
}
```

Web Service 接口:

```
@WebService(name = "PersonService")
public interface PersonService
{
  @WebMethod(operationName = "addPersons", action = "urn:AddPersons")
  public void addPersons( @WebParam(name = "persons")Persons persons );
}
```

同时，Castor 不支持直接或者间接的对接口的映射。

Schema 优先的开发

首先将 schema 文件 Member.xsd 保存到 META-INF/schema文件夹下：

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" attributeFormDefault="qualified"
elementFormDefault="qualified" targetNamespace="http://facet.smallnest.googlepages.com">
<xsd:complexType name="Member">
<xsd:sequence>
<xsd:element minOccurs="0" name="email" nillable="true" type="xsd:string"/>
<xsd:element minOccurs="0" name="name" nillable="true" type="xsd:string"/>
<xsd:element minOccurs="0" name="rank" type="xsd:int"/>
</xsd:sequence>
</xsd:complexType>
<xsd:element name="addMember">
<xsd:complexType>
<xsd:sequence>
<xsd:element maxOccurs="1" minOccurs="1" name="member" nillable="true" type="tns:Member"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:element name="addMemberResponse">
<xsd:complexType>
<xsd:sequence>
<xsd:element maxOccurs="1" minOccurs="1" name="out" type="xsd:boolean"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:element name="getMembers">
<xsd:complexType/>
</xsd:element>
<xsd:complexType name="ArrayOfMember">
<xsd:sequence>
<xsd:element maxOccurs="unbounded" minOccurs="0" name="Member" nillable="true"
type="tns:Member"/>
</xsd:sequence>
</xsd:complexType>
<xsd:element name="getMembersResponse">
<xsd:complexType>
<xsd:sequence>
<xsd:element maxOccurs="1" minOccurs="1" name="Members" nillable="true"
type="tns:ArrayOfMember"/>
</xsd:sequence>
</xsd:complexType>
```

```
</xsd:element>
<xsd:complexType name="User">
<xsd:sequence>
<xsd:element minOccurs="0" name="password" nillable="true" type="xsd:string"/>
<xsd:element minOccurs="0" name="username" nillable="true" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>
<xsd:element name="User" type="tns:User"/>
</xsd:schema>
```

现在你可以使用 castor 提供的工具根据这个 schema 生成 POJO。castor 的工具和说明都可以在 http://www.castor.org/sourcegen.html 得到。它提供了一个 Ant Task：

```
<ant:taskdef name="castor-srcgen"
                classname="org.exolab.castor.tools.ant.taskdefs.CastorSourceGenTask"
                classpathref="castor.class.path"/>
```

你可以调用这个 task 创建代码：

```
<castor-srcgen file="${sources}/META-INF/schema/Memeber.xsd"
                        package="com.googlepages.smallnest.facet"
                        todir="${sources}"
                        types="j2"
                        warnings="false" />
```

下面你就可以实现一个 Web Service 了：

```
package com.googlepages.smallnest.facet;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

public class MemberService
{
        public GetMembersResponse getMembers()
        {
                … …
```

```
        }
}
```

services.xml 文件的配置如下：

```
<beans >

  <service xmlns="http://xfire.codehaus.org/config/1.0">
    <serviceClass>com.googlepages.smallnest.facet.MemberService</serviceBean>
    <schemas>
      <schema>META-INF/schema/Member.xsd</schema>
    </schemas>
    <style>document</style>
    <serviceFactory>#castorServiceFactory</serviceFactory>
  </service>

  <bean id="castorTypeRegistry" class="org.codehaus.xfire.castor.CastorTypeMappingRegistry"
/>

  <bean id="bindingProvider" class="org.codehaus.xfire.aegis.AegisBindingProvider">
        <constructor-arg ref="castorTypeRegistry" />
  </bean

  <bean id="castorServiceFactory"
class="org.codehaus.xfire.service.binding.ObjectServiceFactory">
    <constructor-arg index="0" ref="xfire.transportManager" />
    <constructor-arg index="1" ref="bindingProvider" />
   </bean>
</beans>
```

代码优先的开发

基于代码优先的开发方式不再过多叙述，开发流程如下：

1. 创建 POJO
2. 创建 Web Service，参数或者返回值可能包含第一步创建的 POJO

3. 为 POJO 创建合适的 schema
4. 根据 castor 创建映射文件（可以通过工具自动创建）
5. 创建 services.xml 文件

JiBX 绑定

本章的内容包括：

- JiBX 绑定

JiBX 也是一个绑定框架，能够在 XML 和 Java 类之间提供相当复杂的映射。你可以通过它提供的工具(http://jibx.sourceforge.net/jibx-tools/index.html)创建 schema 或者通过 xsd2Jibx(http://jibx.sourceforge.net/xsd2jibx/index.html)工具从 schema 创建 java 代码。据说它的性能不错。

本章的例子实现了将一个天气预报的数据的中一种温度计量单位（华氏或者摄氏）转换为另外一种计量单位。

本章假设几个先决条件：你已有一个天气预报数据的 **scheme**，一个温度转换类 TemperatureConverter,服务接口 TemperatureConversionService和实现类 TemperatureConversionServiceImpl以及服务方法中所需的数据类型 Weather。

WeatherData.xsd

```
<xsd:complexType name="WeatherData">
   <xsd:sequence>
    <xsd:element minOccurs="0" maxOccurs="1" name="Day" type="xsd:string"/>
    <xsd:element minOccurs="0" maxOccurs="1" name="WeatherImage" type="xsd:string"/>
    <xsd:element minOccurs="0" maxOccurs="1" name="MaxTemperatureF" type="xsd:string"/>
    <xsd:element minOccurs="0" maxOccurs="1" name="MinTemperatureF" type="xsd:string"/>
    <xsd:element minOccurs="0" maxOccurs="1" name="MaxTemperatureC" type="xsd:string"/>
    <xsd:element minOccurs="0" maxOccurs="1" name="MinTemperatureC" type="xsd:string"/>
   </xsd:sequence>
  </xsd:complexType>
```

TemperatureConverter.java:

```
public class TemperatureConverter {

  public static float convertToCelsius(float fahrenheit) {
    return (fahrenheit - 32) / 1.8f;
  }

  public static float convertToFahrenheit(float celsius) {
    return (celsius * 1.8f) + 32;
  }
}
```

TemperatureConversionService.java:

```
public interface TemperatureConversionService {

    Weather convertFahrenheitToCelsius(Weather fahrenheitWeather);

    Weather convertCelsiusToFahrenheit(Weather celsiusWeather);
}
```

TemperatureConversionServiceImpl.java:

```
public class TemperatureConversionServiceImpl implements TemperatureConversionService {

  public TemperatureConversionServiceImpl() {}

  public Weather convertFahrenheitToCelsius(Weather fahrenheitWeather) {
    fahrenheitWeather.setMaxTempC(
        TemperatureConverter.convertToCelsius(fahrenheitWeather.getMaxTempF()));
    fahrenheitWeather.setMinTempC(
        TemperatureConverter.convertToCelsius(fahrenheitWeather.getMinTempF()));
    return fahrenheitWeather;
  }

  public Weather convertCelsiusToFahrenheit(Weather celsiusWeather) {
    celsiusWeather.setMaxTempF(
        TemperatureConverter.convertToFahrenheit(celsiusWeather.getMaxTempC()));
    celsiusWeather.setMinTempF(
        TemperatureConverter.convertToFahrenheit(celsiusWeather.getMinTempC()));
```

```
    return celsiusWeather;
  }
}
```

Weather.java:

```
public class Weather {

  private String day;
  private String image;
  private float maxTempF;
  private float minTempF;
  private float maxTempC;
  private float minTempC;

  public Weather() {}

  public String getDay() {
    return day;
  }

  public void setDay(String day) {
    this.day = day;
  }

  public String getImage() {
    return image;
  }

  public void setImage(String image) {
    this.image = image;
  }

  public float getMaxTempC() {
    return maxTempC;
  }

  public void setMaxTempC(float maxTempC) {
    this.maxTempC = maxTempC;
  }
```

```
  public float getMaxTempF() {
    return maxTempF;
  }

  public void setMaxTempF(float maxTempF) {
    this.maxTempF = maxTempF;
  }

  public float getMinTempC() {
    return minTempC;
  }

  public void setMinTempC(float minTempC) {
    this.minTempC = minTempC;
  }

  public float getMinTempF() {
    return minTempF;
  }

  public void setMinTempF(float minTempF) {
    this.minTempF = minTempF;
  }
}
```

现在，你需要在 Weather 类和 schema 定义 WeatherData 之间定义一种映射，在 JiBX 中，这叫做绑定定义(binding definition)。因为 Weather 和 WeatherData 之间非常相近，所以绑定定义文件也比较的直观,JiBX 允许你进行复杂的映射：

```
<binding>
 <namespace uri="http://www.webservicex.net"/>
 <mapping name="WeatherData" class="org.codehaus.xfire.jibx.Weather">
   <value name="Day" set-method="setDay" get-method="getDay"/>
   <value name="WeatherImage" set-method="setImage" get-method="getImage"/>
   <value name="MaxTemperatureF" usage="optional" set-method="setMaxTempF" get-
method="getMaxTempF"/>
   <value name="MinTemperatureF" usage="optional" set-method="setMinTempF" get-
method="getMinTempF"/>
   <value name="MaxTemperatureC" usage="optional" set-method="setMaxTempC" get-
method="getMaxTempC"/>
```

```
   <value name="MinTemperatureC" usage="optional" set-method="setMinTempC" get-
method="getMinTempC"/>
 </mapping>
</binding>
```

services.xml 文件定义如下：

```
<import resource="classpath:org/codehaus/xfire/spring/xfire.xml"/> <bean
id="jibxServiceFactory" class="org.codehaus.xfire.jibx.JibxServiceFactory">
  <constructor-arg ref="xfire.transportManager"/>
</bean>

<bean id="serviceBean"
  class="org.codehaus.xfire.jibx.TemperatureConversionServiceImpl"
  singleton="true"/>

<bean id="temperatureConversionService"
  class="org.codehaus.xfire.spring.remoting.XFireExporter"
  singleton="true">
 <property name="serviceBean" ref="serviceBean"/>
 <property name="serviceFactory" ref="jibxServiceFactory"/>
 <property name="serviceClass" value="org.codehaus.xfire.jibx.TemperatureConversionService"/>
 <property name="namespace" value="http://jibx.xfire.codehaus.org/"/>
 <property name="xfire" ref="xfire"/>
 <property name="schemas">
  <list>
   <value>WeatherData.xsd</value>
  </list>
 </property>
</bean>
```

使用 Ant Task 进行绑定：

```
<taskdef name="bind" classname="org.jibx.binding.ant.CompileTask"
  classpath="/path/to/jibx-bind.jar"/>

 <target name="jibxCompile">
  <bind verbose="true" load="true" binding="jibx_bindings.xml">
   <classpathset dir="/classes"/>
  </bind>
```

```
</target>
```

客户端的 Spring 文件：

```
<import resource="classpath:org/codehaus/xfire/spring/xfire.xml"/>

  <bean name="jibxServiceFactory" class="org.codehaus.xfire.jibx.JibxServiceFactory">
    <constructor-arg ref="xfire.transportManager"/>
  </bean>

  <bean id="client" class="org.codehaus.xfire.spring.remoting.XFireClientFactoryBean">
     <property name="serviceInterface">
         <value>org.codehaus.xfire.jibx.TemperatureConversionService</value>
     </property>
     <property name="wsdlDocumentUrl">
        <value>http://yourhostand:port/jibxExample/TemperatureConversion?wsdl</value>
     </property>
     <property name="namespaceUri">
        <value>http://jibx.xfire.codehaus.com/</value>
     </property>
     <property name="serviceFactory" ref="jibxServiceFactory"/>
  </bean>
```

相对来说，JiBX 的绑定还是比较复杂的，你会在什么情况下选择 JiBX 作为你的绑定框架?

MessageBinding

本章的内容包括：

- 直接的 Message 处理方式

前面几章花了大量的篇幅介绍各种绑定方式，这是因为绑定是 XFire 非常重要的一块，出了前面提到的几种绑定方式，XFire 还提供了一种完全不同的绑定方式：MessageBinding。

为什么还要提供另外一种绑定方式呢？有时候，你可能想直接和 XML 打交道。Aegis 绑定方式允许这么做。你可以在你的 Bean 中使用 Document、Element、XmlStreamReader，但是 Aegis 假定我们处理这个 stream 并把它对应的一个对象上，所以你并能直接的使用 XmlStreamReader。

MessageBinding 可以直接从 request 得到 XmlStreamReader，并把它直接提供给你使用。你也可以用一个 XmlStreamReader 响应这个请求。

为了使用 Messagebinding，你需要把 Factory 的 style 设置为 message。

```
ObjectServiceFactory factory = new ObjectServiceFactory(new MessageBindingProvider());
factory.setStyle("message");
```

你可以声明一个服务，参数和返回值都是 XmlStreamReader：

```
public class MyService {
  public XMLStreamReader invoke(XMLStreamReader reader) {
    //处理 reader
    return responseStream;
  }
}
```

身份验证

本章的内容包括：

- 身份验证的几种方式

很多情况下你会考虑到在 Web Services 采用身份验证。作为一个 Web Services 发布商，你为你的客户提供股票分析的服务，并收取服务的费用，你当然不希望其他人能免费的使用它，所以你希望加把锁，只有验证过的用户才可以使用。提供短信服务商的用户也是如此。即使在同一个公司，你也不希望你的员工随意的访问财务部的 Web Services，获取公司员工的薪水情况（我们公司的薪水按规定保密的，不知道其它公司的是否也如此）.....

本章将介绍几种身份验证的方式。

HTTP 身份验证

安全起见，你可以在你的应用服务器(Web 服务器)中设置允许访问的 IP，或在防火墙上设置 IP 访问的策略，在最外面的大门上先加一把锁。

而且，你可以通过应用服务器(Web 服务器)配置你的 Web 应用程序进行身份验证，具体如何配置可以参考各应用服务器(Web 服务器)的配置文档，如果你采用 Apache 进行负载平衡，也可以在 Apache 中进行配置。

原谅我每次在应用服务器名词之后都加个括号。在上一版本的书中，我将 Tomcat 笼统的称之为应用服务器，有人抗议说 Tomcat 不是应用服务器，安全起见，我加个括号说明一下。

SOAP Header 中进行身份验证(JSR181 方式)

在第四章的例子中，已经演示了如何采用 JSR181 标注的方式进行身份验证，可以到看，采用标注的方式处理身份验证非常的简单实用。

SOAP Header 中进行身份验证(handler 方式)

如果在 SOAP Header 中身份验证的方式比较特殊，是自己定义的一种方式，那么可以通过 Handler 来处理（事实上 Handler 不仅仅能处理 SOAP Header，它也可以处理 SOAP Body）。

假设在 SOAP Header 中身份验证的信息如下：

```
<AuthenticationToken xmlns="http://facet.smallnest.googlepages.com.cn">
  <Username>smallnest</Username>
  <Password>mypass</Password>
</AuthenticationToken>
```

你可以写一个 Handler 来处理它：

```
import org.codehaus.xfire.MessageContext;
import org.codehaus.xfire.handler.AbstractHandler;
import org.codehaus.yom.Element;

public class AuthenticationHandler extends AbstractHandler
{
    private final static String TOKEN_NS = "http://facet.smallnest.googlepages.com.cn";

    public void invoke(MessageContext context)
        throws XFireFault
    {
        if (context.getInMessage().getHeader() == null)
        {
            throw new XfireFault(" 请求必须包含身份验证信息",
                                 XFireFault.SENDER);
        }

        Element token =
context.getInMessage().getHeader().getFirstChildElement("AuthenticationToken", TOKEN_NS);
```

```
        if (token == null)
        {
            throw new XFireFault("请求必须包含身份验证信息",
                                 XFireFault.SENDER);
        }

        String username = token.getFirstChildElement("Username", TOKEN_NS).getValue();
        String password = token.getFirstChildElement("Password", TOKEN_NS).getValue();

        try
        {
            //进行身份验证
            ....

            // 身份验证通过
            context.setProperty(User_KEY, user);
        }
        catch (Exception e)
        {
            throw new XFireFault("非法的用户名和密码", XFireFault.SENDER);
        }
    }
}
```

客户端代码需要提供身份验证的信息：

ClientAuthenticationHandler.java

```
public class ClientAuthenticationHandler extends AbstractHandler {

    private String username = null;
    private String password = null;


    public ClientAuthenticationHandler() {
    }

    public ClientAuthenticationHandler(String username,String password) {
        this.username = username;
        this.password = password;
```

```
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public void invoke(MessageContext context) throws Exception {


        final Namespace ns =
Namespace.getNamespace("http://facet.small1nest.goog1epages.com.cn");
        Element e1 = new Element("header",ns);
        context.getOutMessage().setHeader(e1);

        Element auth = new Element("AuthenticationToken", ns);
        Element username_e1 = new Element("Username",ns);
        username_e1.addContent(username);
        Element password_e1 = new Element("Password",ns);
        password_e1.addContent(password);
        auth.addContent(username_e1);
        auth.addContent(password_e1);
        e1.addContent(auth);
    }
}
```

访问代码：

```
MyXFireClient rsc = new MyXFireClient();
        MyPortType myType = rsc.getServiceHttpPort();
        XFireProxy proxy =  (XFireProxy)Proxy.getInvocationHandler(myType);
        Client client = proxy.getClient();

        client.addOutHandler(new ClientAuthenticationHandler("small1nest","mypass"));
```

WS-Security

Spring 集成

本章的内容包括：

- 通过 Spring 文件配置 Web Services
- 使用 Spring Remoting模块
- 高级配置

Xfire 很好的提供了对 Spring 的支持。你可以很方便的将一个 Spring bean 包装成一个 Web Service。可以通过创建 Spring 管理的 XFire engine、ServiceRegistry、ransportManager 和 ServiceFactory 来实现。

XBean、Spring 及其它

Xfire 使用 XBean(http://xbean.org)来管理配置文件。XBean 允许在 XML 文档中混合 Spring Bean 的定义。

下面是一个简单的配置文件：

```
<beans xmlns="http://xfire.codehaus.org/config/1.0">

  <service>
    <name>Echo</name>
    <serviceClass>org.codehaus.xfire.test.Echo</serviceClass>
  </service>

</beans>
```

可以通过 XBean 的 Classpath application context 加载它：

```
import junit.framework.TestCase;
```

```
import org.apache.xbean.spring.context.ClassPathXmlApplicationContext;
import org.codehaus.xfire.service.ServiceRegistry;

public class XBeanExampleTest
    extends TestCase
{
    public void testLoading()
        throws Exception
    {
        ClassPathXmlApplicationContext context =
            new ClassPathXmlApplicationContext(new String[] {
                "/org/codehaus/xfire/spring/examples/simple.xml",
                "/org/codehaus/xfire/spring/xfire.xml" });

        ServiceRegistry reg = (ServiceRegistry) context.getBean("xfire.serviceRegistry");
        assertTrue(reg.hasService("Echo"));
    }
}
```

下面是一个混合 Spring Bean 的例子：

```
<beans xmlns="http://xfire.codehaus.org/config/1.0">

  <service>
    <name>Echo</name>
    <serviceBean>#echoBean</serviceBean>
  </service>
  <bean id="echoBean" class="org.codehaus.xfire.services.Echo"/>
</beans>
```

为 service 设置属性：

```
<service>
  … define your normal attributes …
  <properties>
    <property key="mtom-enabled">true</property>
    <property key="myProperty">myValue</property>
  <properties>
```

```
</service>
```

你可以完全通过 Spring 声明的方式定义服务,例如：

```
<bean name="echoService" class="org.codehaus.xfire.spring.ServiceBean">
    <property name="serviceBean" ref="echo"/>
    <property name="serviceClass" value="org.codehaus.xfire.test.Echo"/>
    <property name="inHandlers">
      <list>
        <ref bean="addressingHandler"/>
      </list>
    </property>
</bean>

<bean id="echo" class="org.codehaus.xfire.test.EchoImpl"/>

<bean id="addressingHandler" class="org.codehaus.xfire.addressing.AddressingInHandler"/>
```

XFire 定义了几个标准的 Bean，如 TransportManager、 ServiceRegistry以及一些简单的 ServiceFactory。你可以通过下面的方式引入到你的 Spring 文件中：

```
  <import resource="classpath:org/codehaus/xfire/spring/xfire.xml"/>
```

设置属性通过下面的方式：

```
  <bean class="org.codehaus.xfire.spring.ServiceBean">
  … define your normal attributes …
  <property name="properties">
    <map>
      <entry>
        <key><value>mtom-enabled</value></key>
        <value>true</value>
      </entry>
      <entry>
        <key><value>myProperty</value></key>
        <value>myValue</value>
      </entry>
    </map>
```

```
  <property>
</bean>
```

你可以利用XFireSpringServlet通过HTTP的方式暴露Web Services不必使用Spring Remoting模块：

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

<web-app>
  <servlet>
    <servlet-name>XFireServlet</servlet-name>
    <display-name>XFire Servlet</display-name>
    <servlet-class>
        org.codehaus.xfire.spring.XFireSpringServlet
    </servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>XFireServlet</servlet-name>
    <url-pattern>/servlet/XFireServlet/*</url-pattern>
  </servlet-mapping>

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

Spring Remoting

首先在你的web.xml加入一个DispatcherServlet：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext.xml
    classpath:org/codehaus/xfire/spring/xfire.xml</param-value>
```

```
</context-param>

<context-param>
    <param-name>log4jConfigLocation</param-name>
    <param-value>/WEB-INF/log4j.properties</param-value>
</context-param>

<listener>
    <listener-class>org.springframework.web.util.Log4jConfigListener</listener-class>
</listener>

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

<servlet>
    <servlet-name>xfire</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>xfire</servlet-name>
    <url-pattern>/*</url-pattern>
</servlet-mapping>
```

服务接口：

```
package org.codehaus.xfire.spring.example;

public interface Echo
{
    String echo(String in);
}
```

接口实现类：

```
package org.codehaus.xfire.spring.example;

public class EchoImpl
```

```
        implements Echo
{
    public String echo(String in)
    {
        return in;
    }
}
```

下一步就是为这个 Service 定义一个 exporter，既然前面定义的 dispatcher servlet 名字叫 xfire，那么我们这就里的配置文件名就叫 xfire-servlet.xml：

```
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="urlMap">
        <map>
            <entry key="/EchoService">
                <ref bean="echo"/>
            </entry>
        </map>
    </property>
</bean>

<bean id="echo" class="org.codehaus.xfire.spring.remoting.XFireExporter">
    <property name="serviceFactory">
        <ref bean="xfire.serviceFactory"/>
    </property>
    <property name="xfire">
        <ref bean="xfire"/>
    </property>
    <property name="serviceBean">
        <ref bean="echoBean"/>
    </property>
    <property name="serviceClass">
        <value>org.codehaus.xfire.spring.example.Echo</value>
    </property>
</bean>
```

在 spring 的定义文件中，echoBean 的定义如下：

```
<bean id="echoBean" class="org.codehaus.xfire.spring.example.EchoImpl"/>
```

Spring Remoting Client

你可以在 Spring 中配置一个服务的客户端：

```
<bean id="testWebService" class="org.codehaus.xfire.spring.remoting.XFireClientFactoryBean">
    <property name="serviceClass">
      <value>org.codehaus.xfire.spring.example.Echo</value>
    </property>
    <property name="wsdlDocumentUrl">
      <value>http://localhost:8080/xfire/EchoService?WSDL</value>
    </property>
</bean>
```

高级配置

下面几个例子是 XFireExporter 实践例子。

定义 AnnotationServiceFactory（JAVA 5 环境）：

```
<bean id="xfire.annotationServiceFactory"
        class="org.codehaus.xfire.annotations.AnnotationServiceFactory">
        <constructor-arg index="0">
            <ref bean="xfire.commonsAnnotations"/>
        </constructor-arg>
        <constructor-arg index="1">
            <ref bean="xfire.transportManager"/>
        </constructor-arg>
        <constructor-arg index="2">
            <ref bean="xfire.aegisBindingProvider"/>
        </constructor-arg>
    </bean>

    <bean id="xfire.commonsAnnotations"
        class="org.codehaus.xfire.annotations.jsr181.Jsr181WebAnnotations"/>
```

定义 AnnotationServiceFactory（JAVA 1.4 环境）：

```
<bean id="xfire.annotationServiceFactory"
```

```
        class="org.codehaus.xfire.annotations.AnnotationServiceFactory">
        <constructor-arg index="0">
            <ref bean="xfire.commonsAnnotations"/>
        </constructor-arg>
        <constructor-arg index="1">
            <ref bean="xfire.transportManager"/>
        </constructor-arg>
        <constructor-arg index="2">
            <ref bean="xfire.aegisBindingProvider"/>
        </constructor-arg>
    </bean>

    <bean id="xfire.commonsAnnotations"
        class="org.codehaus.xfire.annotations.commons.CommonsWebAttributes"/>
```

输出你的通过注释定义的 Web Service:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
  <bean name="/Echo" class="org.codehaus.xfire.spring.remoting.XFireExporter">
    <property name="serviceBean" ref="echo"/>
    <property name="serviceClass"><value>org.codehaus.xfire.spring.Echo</value></property>
    <property name="serviceFactory"><ref bean="xfire.annotationServiceFactory"/>
  </bean>

  <bean id="echo" class="org.codehaus.xfire.spring.EchoImpl"/>
</beans>
```

下面再看一个 XMLBeans 和 Spring 的例子。

定义 XMLBeans 的 ServiceFactory(构造子注入法):

```
<bean id="xfire.xmlbeansServiceFactory"
       class="org.codehaus.xfire.xmlbeans.XmlBeansServiceFactory"
       singleton="true">
       <constructor-arg index="0">
```

```
            <ref bean="xfire.transportManager"/>
        </constructor-arg>
 </bean>
```

定义 XMLBeans 的 ServiceFactory(Setter 注入法):

```
<bean id="xfire.xmlbeansServiceFactory"
        class="org.codehaus.xfire.xmlbeans.XmlBeansServiceFactory"
        singleton="true">
        <property name="transportManager">
            <ref bean="xfire.transportManager"/>
        </property>
    </bean>
```

输出服务：

```
<bean name="/Echo" class="org.codehaus.xfire.spring.remoting.XFireExporter">
    <property name="serviceBean"><ref bean="echo"/></property>
    <property name="serviceClass"><value>org.codehaus.xfire.spring.Echo</value></property>
    <property name="serviceFactory"><ref bean="xfire.xmlbeansServiceFactory"/></property>
  </bean>
```

MTOM

本章的内容包括：

- 通过一个上传图片的服务演示 MTOM 的开发

MTOM 提供了在 Web Services 中处理大的二进制数据的方法。

使用 MTOM 非常简单，在代码中加入

```
service.setProperty("mtom-enabled", "true");
```

或者在 services.xml 文件中配置：

```
<service>
  … normal service definition
  <properties>
    <property key="mtom-enabled">true</property>
  </properties>
</service>
```

在客户端代码中也要进行相应的配置：

```
MyService myClient = …;
Client client = Client.getInstance(myClient);

client.setProperty("mtom-enabled", "true");
```

Aegis 支持下列类的优化：

- DataSource
- DataHandler
- byte[]

你也可以实现一个自己的类型，只需继承 AbstractXOPType。

任何在 schema 中的 base64Binary 都可使用 JAXB 优化。你也可以指定期望的 mime 类型：

```
<s:element name="GetPictureResponse">
    <s:complexType>
      <s:sequence>
         <s:element name="image" type="s:base64Binary"
xmime:expectedContentTypes="image/jpeg"/>
      </s:sequence>
    </s:complexType>
  </s:element>
```

JAXB 足够聪明，使用 Image 对象代替 byte[]。

现在给你提一个需要，创建一个 Web Service，用户可以使用它上传图片，如何做？

服务的接口如下：

```
package com.googlepages.small1nest.facet;

public interface PictureService
{
       public boolean uploadPicture(byte[] data,String name);
}
```

实现类：

```
package com.googlepages.small1nest.facet;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class PictureServiceImpl implements PictureService
{
```

```
        public boolean uploadPicture(byte[] data, String name)
        {

                try
                {
                        File image = new File(name);
                        image.createNewFile();
                        FileOutputStream out = new FileOutputStream(name);
                        out.write(data);
                        out.close();
                        System.out.println(image.getAbsolutePath());
                }
                catch (FileNotFoundException e)
                {
                        return false;
                }
                catch (IOException e)
                {
                        return false;
                }

                return true;
        }

}
```

services.xml 文件：

```
<beans>
  <service xmlns="http://xfire.codehaus.org/config/1.0">
    <name>PictureService</name>
    <namespace>http://smallnest.googlepages.com/PictureService</namespace>
    <serviceClass>com.googlepages.smallnest.facet.PictureService</serviceClass>
    <implementationClass>com.googlepages.smallnest.facet.PictureServiceImpl</implementationC
lass>
    <properties>
       <property key="mtom-enabled">true</property>
       </properties>
  </service>
</beans>
```

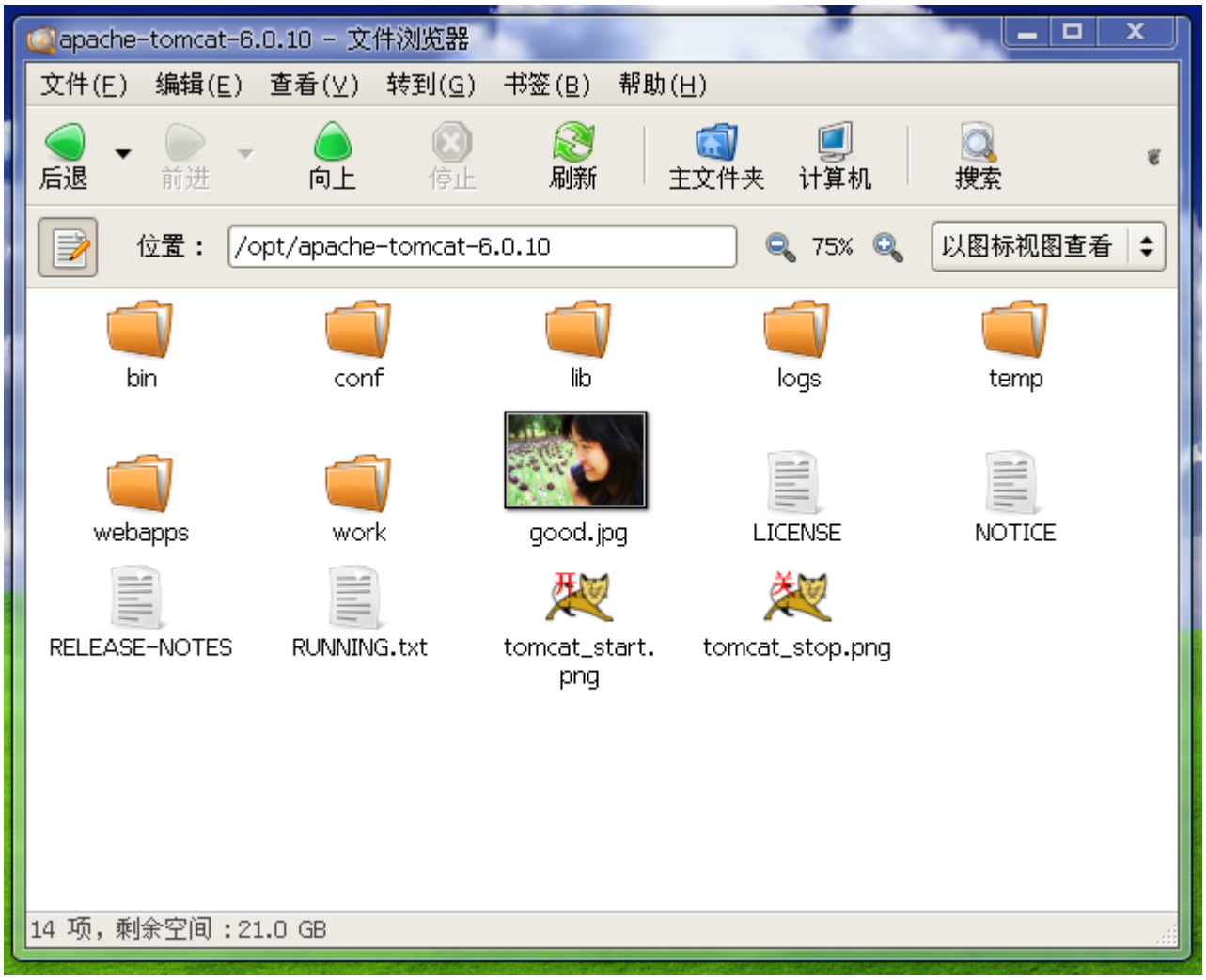
发布这个服务，等待客户端上传。

下面创建一个客户端进行测试。

在 eclipse 中创建一个 Java 工程，利用 Ant 和上面的 Web Service 的 WSDL 生成客户端的访问代码。创建 FacetClient 类：

```
package com.googlepages.smallnest.facet;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class FacetClient
{

	public static void main(String[] args)
	{
		PictureServiceClient client = new PictureServiceClient();
		PictureServicePortType pictureService = client.getPictureServiceHttpPort();
		FileInputStream in;
		try
		{
			in = new FileInputStream("girl.jpg");
			byte[] data = new byte[in.available()];
			in.read(data); //图片不大
			in.close();
			boolean b = pictureService.uploadPicture(data, "good.jpg");
			System.out.println(b);
		}
		catch (FileNotFoundException e)
		{
			e.printStackTrace();
		}
		catch (IOException e)
		{
			e.printStackTrace();
		}

	}

}
```

运行这个程序，在 Tomcat 相应的目录查看上传的结果：


apache-tomcat-6.0.10 - 文件浏览器
文件(F) 编辑(E) 查看(V) 转到(G) 书签(B) 帮助(H)
后退
前进
向上
停止
刷新
主文件夹
计算机
搜索
位置：
/opt/apache-tomcat-6.0.10
75%
以图标视图查看
bin
conf
lib
logs
temp
webapps
work
good.jpg
LICENSE
NOTICE
RELEASE-NOTES
RUNNING.txt
tomcat_start.png
tomcat_stop.png
14 项，剩余空间：21.0 GB

后记

该到了收尾的时候了,其它和 XFire 相关的问题，比如 WS - Security、比如版本控制，比如 JMS 传输，关于这些，或许将来放在 V2.1 中进行介绍。