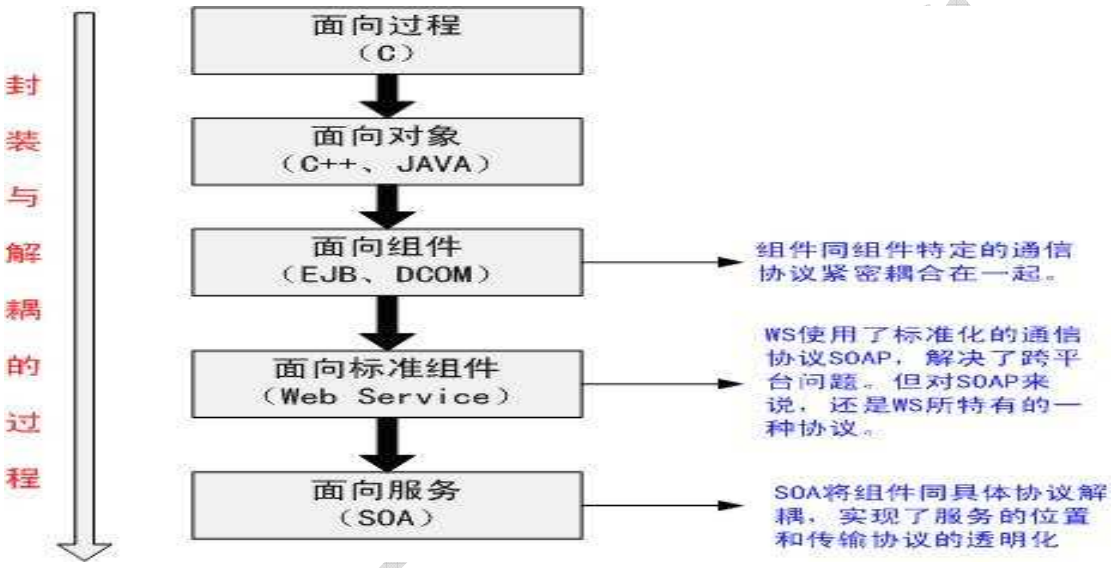
JAVA 的 WebService 支持

李海峰（QQ:61673110）-Andrew830314@163.com

SOA(Service-Oriented Architecture)面向服务架构是一种思想，它将应用程序的不同功能单元通过中立的契约（独立于硬件平台、操作系统和编程语言）联系起来，使得各种形式的功能单元更好的集成。目前来说，WebService 是 SOA 的一种较好的实现方式，WebService 采用 HTTP 作为传输协议，SOAP（Simple Object Access Protocol）作为传输消息的格式。但 WebService 并不是完全符合 SOA 的概念，因为 SOAP 协议是 WebService 的特有协议，并未符合 SOA 的传输协议透明化的要求。SOAP 是一种应用协议，早期应用于 RPC 的实现，传输协议可以依赖于 HTTP、SMTP 等。

SOA 的产生共经历了如下过程：



通常采用 SOA 的系统叫做服务总线（BUS），结构如下图所示：

JAVA 中的 Web 服务规范：

JAVA 中共有三种 WebService 规范，分别是 JAXM&SAAJ、JAX-WS（JAX-RPC）、JAX-RS。下面来分别简要的介绍一下这三个规范。

(1.)JAX-WS：

JAX-WS（Java API For XML-WebService），JDK1.6 自带的版本为 JAX-WS2.1，其底层支

持为 JAXB。早期的基于 SOAP 的 JAVA 的 Web 服务规范 JAX-RPC（Java API For XML-Remote Procedure Call）目前已经被 JAX-WS 规范取代，JAX-WS 是 JAX-RPC 的演进版本，但 JAX-WS 并不完全向后兼容 JAX-RPC，二者最大的区别就是 RPC/encoded 样式的 WSDL，JAX-WS 已经不提供这种支持。JAX-RPC 的 API 从 JAVA EE5 开始已经移除，如果你使用 J2EE1.4，其 API 位于 javax.xml.rpc.*包。
JAX-WS（JSR 224）规范的 API 位于 javax.xml.ws.*包，其中大部分都是注解，提供 API 操作 Web 服务（通常在客户端使用的较多，由于客户端可以借助 SDK 生成，因此这个包中的 API 我们较少会直接使用）。
WS-MetaData（JSR 181）是 JAX-WS 的依赖规范，其 API 位于 javax.jws.*包，使用注解配置公开的 Web 服务的相关信息和配置 SOAP 消息的相关信息。

(2.)JAXM&SAAJ：
JAXM（JAVA API For XML Message）主要定义了包含了发送和接收消息所需的 API，相当于 Web 服务的服务器端，其 API 位于 javax.messaging.*包，它是 JAVA EE 的可选包，因此你需要单独下载。
SAAJ（SOAP With Attachment API For Java，JSR 67）是与 JAXM 搭配使用的 API，为构建 SOAP 包和解析 SOAP 包提供了重要的支持，支持附件传输，它在服务器端、客户端都需要使用。这里还要提到的是 SAAJ 规范，其 API 位于 javax.xml.soap.*包。
JAXM&SAAJ 与 JAX-WS 都是基于 SOAP 的 Web 服务，相比之下 JAXM&SAAJ 暴漏了 SOAP 更多的底层细节，编码比较麻烦，而 JAX-WS 更加抽象，隐藏了更多的细节，更加面向对象，实现起来你基本上不需要关心 SOAP 的任何细节。那么如果你想控制 SOAP 消息的更多细节，可以使用 JAXM&SAAJ，目前版本为 1.3。

(3.)JAX-RS：
JAX-RS 是 JAVA 针对 REST(Representation State Transfer)风格制定的一套 Web 服务规范，由于推出的较晚，该规范（JSR 311，目前 JAX-RS 的版本为 1.0）并未随 JDK1.6 一起发行，你需要到 JCP 上单独下载 JAX-RS 规范的接口，其 API 位于 javax.ws.rs.*包。

这里的 JAX-WS 和 JAX-RS 规范我们采用 Apache CXF 作为实现，CXF 是 Objectweb Celtix 和 Codehaus XFire 合并而成。CXF 的核心是 org.apache.cxf.Bus（总线），类似于 Spring 的 ApplicationContext，Bus 由 BusFactory 创建，默认是 SpringBusFactory 类，可见默认 CXF 是依赖于 Spring 的，Bus 都有一个 ID，默认的 BUS 的 ID 是 cxf。你要注意的是 Apache CXF 2.2 的发行包中的 jar 你如果直接全部放到 lib 目录，那么你必须使用 JDK1.6，否则会报 JAX-WS 版本不一致的问题。对于 JAXM&SAAJ 规范我们采用 JDK 中自带的默认实现。

1.JAVA 的 WebService 规范 JAX-WS：
Web 服务从前面的图中不难看出自然分为 Server、Client 两部分，Server 公开 Web 服务，Client 调用 Web 服务，JAX-WS 的服务端、客户端双方传输数据使用的 SOAP 消息格式封装数据，在后面我们会看到其实 SOAP 信封内包装的就是一段 XML 代码。

I.服务端示例：
我们先看一个服务器端示例：
(1.)公开 Web 服务的接口 IHelloService：

```
package net.ilkj.soap.server;

import javax.jws.WebService;

@WebService
public interface IHelloService {

    Customer selectMaxAgeStudent(Customer c1, Customer c2);

    Customer selectMaxLongNameStudent(Customer c1, Customer c2);
}
```

我们看到这个接口很简单，仅仅是使用类级别注解@WebService 就标注了这个接口的方法将公开为 Web 服务，使用了这个注解的接口的所有方法都将公开为 Web 服务的操作，如果你想屏蔽某个方法，可以使用方法注解@Method 的 exclude=true。我们也通常把公开为 Web 服务的接口叫做 SEI（Service EndPoint Interface）服务端点接口。

(2.)实现类 HelloServiceImpl：

```
package net.ilkj.soap.server;

public class HelloServiceImpl implements IHelloService {

    @Override
    public Customer selectMaxAgeStudent(Customer c1, Customer c2) {
        if (c1.getBirthday().getTime() > c2.getBirthday().getTime())
            return c2;
        else
            return c1;
    }

    @Override
    public Customer selectMaxLongNameStudent(Customer c1, Customer c2)
    {
        if (c1.getName().length() > c2.getName().length())
            return c1;
        else
            return c2;
    }

}
```

这个实现类没有任何特殊之处，但是如果你的实现类还实现了其他的接口，那么你需要在实现类上使用@WebService 注解的 endpointInterface 属性指定那个接口是 SEI（全类名）。

(3.)Customer 类：

```
package net.ilkj.soap.server;
```

```
import java.util.Date;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "Customer")
public class Customer {
    private long id;
    private String name;
    private Date birthday;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Date getBirthday() {
        return birthday;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }
}
```

这个类是公开为 Web 服务的接口中的参数类型和返回值，因此你需要使用 JAXB 注解告诉 CXF 如何在 XML 和 Java Object 之间处理，因为前面说过 SOAP 消息格式包装的是一段 XML 代码，那么无论是服务器端还是客户端在接收到 SOAP 消息时都需要将 XML 转化为 Java Object，在发送 SOAP 消息时需要将 Java Object 转化为 XML。

(4.)发布 Web 服务：

```
package net.ilkj.soap.server;

import javax.xml.ws.Endpoint;

public class SoapServer {
```

```
    public static void main(String[] args) {
        Endpoint.publish("http://127.0.0.1:8080/helloService",
                new HelloServiceImpl());
    }

}
```

注意我们发布 Web 服务使用的是 javax.xml.ws.*包中的 EndPoint 的静态方法 publish()。

(5.)查看 WSDL：

我们访问 **http://127.0.0.1:8080/helloService?wsdl** 地址，您会看到很长的 XML 文件（由于浏览器的问题，如果你看到的是空白页面，请查看源代码），这就是 WSDL(WebService Definition Language)，对于你要访问的 Web 服务，只要在其地址后加上，就可以在浏览器中查看用于描述 Web 服务的 WSDL，这也是一种 XML，Web 服务能够被各种编程语言书写的程序访问就是通过 WSDL 这种通用的契约来完成的。

如果你已经看到 WSDL，那么表示我们的 Web 服务发布成功了。你可能会差异，我们没有借助 Tomcat 这样的 Web 服务器，直接运行一个 main 方法是怎么发布的 Web 服务呢？其实 CXF 内置了 Jetty（Servlet 容器），因此你不需要将你的程序部署到 Tomcat 等 Web 服务器也可以正常发布 Web 服务。

II.分析 WSDL 的构成：

下面我们来解释一下你所看到的 WSDL 的结构，你可以对照你所生成 WSDL（文件太长，Word 里实在放不下）。

(1.)<wsdl:definitions …

这个是 WSDL 的根元素，我们要关心的是三个属性，name 属性值为公开的 Web 服务的接口的实现类+Service（上例中为 name="**HelloServiceImplService**"，不同的 JAX-WS 实现名字是不一样的 ）；targetNamespace 指定目标名称空间，targetNamespace 的值被后面的 xmlns:tns 属性作为值，默认是使用接口实现类的包名的反缀（targetNamespace="**http://server.soap.ilkj.net/**" …

xmlns:tns="**http://server.soap.ilkj.net/**"），

你可以使用@WebService 注解的 targetNamespace 属性指定你想要的名称空间。

(2.)<wsdl:types …

这个元素会通过<xs:element … 声明几个复杂数据类型的元素。

一般首先你看到的是 Web 服务中的方法参数、返回值所涉及的所有复杂（complex）类型的元素定义<xs:element …，其中 name 属性值是这个复杂类型的 JAXB 注解的 name 属性值，type 属性是 tns:+JAXB 注解的 name 属性值的全小写形式（上例中的方法参数、返回值只涉及一个复杂类型 Customer，Customer 的@XmlRootElement 注解的 name 属性值为 Customer，因此你会看到<xs:element name="**Customer**" type="**tns:customer**" />）。

再向下你会看到 XXX 元素和 XXXResponse 元素，其中 XXX 是方法名称（你可以使用 @WebMethod 的 operationName 属性值指定 XXX 的值），XXX 是对方法参数的封装，XXXResponse 是对返回值的封装，上例中你会看到

```
<xs:element name="selectMaxAgeStudentMethod"
   type="tns:selectMaxAgeStudentMethod" />
 <xs:element name="selectMaxAgeStudentMethodResponse"
   type="tns:selectMaxAgeStudentMethodResponse" />
 <xs:element name="selectMaxLongNameStudent"
   type="tns:selectMaxLongNameStudent" />
 <xs:element name="selectMaxLongNameStudentResponse"
   type="tns:selectMaxLongNameStudentResponse" />
```

内容，

最后你会看到一组<xs:complexType …元素，这个元素通过 name 属性关联到<xs:element …，它为前面定义的元素指定封装的具体内容（通过子元素<xs:sequence …指定），上例中方法参数的复杂类型指定如下形式：

```
<xs:complexType name="selectMaxAgeStudentMethod">
    <xs:sequence>
        <xs:element minOccurs="0" name="arg0" type="tns:customer" />
        <xs:element minOccurs="0" name="arg1" type="tns:customer" />
    </xs:sequence>
</xs:complexType>
```

我们看到方法参数名称为 arg0、arg1、…，如果你想指定方法参数的名字在方法参数前使用@WebParam 的 name 属性指定值，同样，方法的返回值同样可以使用@WebResult 注解指定相关的属性值。

例如：

```
@WebResult(name = "method")
Customer selectMaxAgeStudent(@WebParam(name = "c1") Customer c1,
                             @WebParam(name = "c2") Customer c2);
```

(3.)<wsdl:message …

这个元素将输入参数（方法参数）和响应结果（方法返回值）、受检查的异常信息包装为消息。

(4.)<wsdl:portType …

这个元素指定 Web 服务的端口类型（Web 服务会被发布为 EndPoint 端点服务），它的 name 属性默认为接口名称（你可以使用@WebService 注解的 name 属性指定值）。这个元素包含了一系列的<wsdl:operation …子元素指定该端点服务包含了那些操作（方法），<wsdl:operation …的子元素<wsdl:input …、<wsdl:output …指定操作的输入输出（通过属性 message 绑定到前面声明过的消息）。

(5.)<wsdl:binding …

这个元素将前面最终的端点服务绑定到 SOAP 协议（你可以看出来 WSDL 从上到下依次有着依赖关系），其中的<soap:xxx …的 style、use 分别可以使用 SOAPBinding 注解的 style、use 属性指定值、<wsdl:operation …指定公开的操作（方法）。这部分 XML 指定最终发布的 Web 服务的 SOAP 消息封装格式、发布地址等。

(6.)<wsdl:service …

这个元素的 name 属性指定服务名称（这里与根元素的 name 属性相同），子元素<wsdl:port…的 name 属性指定 port 名称，子元素<soap:address … 的 location 属性指定 Web 服务的地址。

III.客户端调用示例：

我们从上面可以知道 Web 服务只向客户端暴漏 WSDL，那么客户端必须将 WSDL 转换为自己的编程语言书写的代码。JAX-WS 的各种实现都提供相应的工具进行 WSDL 与 JAVA 之间的互相转换，你可以在 CXF 的运行包中找到 bin 目录，其中的 wsdl2java.bat 可以将 WSDL 转换为 JAVA 类，bin 目录的各种 bat 的名字可以很容易知道其作用，但要注意 JAVA 类转换为 WSDL 最好使用前面的 URL?wsdl 的方式获得，因为这样得到的是最准确的。

你可以在命令行将当前目录切换到 CXF 的 bin 目录，然后运行 wsdl2java –h 查看这个批处理命令的各个参数的作用，常用的方式就是 **wsdljava –p 包路径 –d 目标文件夹 wsdl 的 url 地址**。现在我们将前面的 WSDL 生成客户端代码：

wsdl2java -p net.ilkj.soap.client –d E:\ http://127.0.0.1:8080/helloService?wsdl

你会在 E 盘根目录找到生成的客户端代码，然后将它复制到 Eclipse 工程即可使用。

如果你使用 MyEclipse，可以按照如下步骤从 WSDL 生成客户端代码：

New--->Other--->MyEclipse--->Web Services--->Web Services Client，然后依据设置向导即可完成，但最好还是使用 CXF 的 wsdl2java 来完成，因为 CXF2.2+版本开始支持 JAX-WS2.1 规范，而 MyEclipse 自带的好像是 XFire 的 wsdl2java，生成的客户端代码可能不是最新规范的。

我们上面的 WSDL 会生成如下所示的客户端代码：

Customer.java
HelloServiceImplService.java
IHelloService.java
ObjectFactory.java
package-info.java
SelectMaxAgeStudent.java
SelectMaxAgeStudentResponse.java
SelectMaxLongNameStudent.java
SelectMaxLongNameStudentResponse.java

其中 package-info.java、ObjectFactory.java 是 JAXB 需要的文件；HelloServiceImplService.java 继承自 javax.xml.ws.Service 类，用于提供 WSDL 的客户端视图，里面使用的是大量 javax.xml.ws.*包中的注解；剩下的类是 Web 服务的接口、方法参数、响应值的类。

在 CXF 中使用 JaxWsProxyFactoryBean 客户端代理工厂调用 Web 服务，代码如下所示：

```
package net.ilkj.soap.client;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.GregorianCalendar;
import org.apache.cxf.jaxws.JaxWsProxyFactoryBean;
import
```

```
com.sun.org.apache.xerces.internal.jaxp.datatype.XMLGregorianCalendar
Impl;

public class SoapClient {

    public static void main(String[] args) throws ParseException {
        JaxWsProxyFactoryBean soapFactoryBean = new
JaxWsProxyFactoryBean();

    soapFactoryBean.setAddress("http://127.0.0.1:8080/helloService");
        soapFactoryBean.setServiceClass(IHelloService.class);
        Object o = soapFactoryBean.create();
        IHelloService helloService = (IHelloService) o;

        Customer c1 = new Customer();
        c1.setId(1);
        c1.setName("A");
        GregorianCalendar calendar = (GregorianCalendar)
GregorianCalendar
                .getInstance();
        calendar
                .setTime(new
SimpleDateFormat("yyyy-MM-dd").parse("1989-01-28"));
        c1.setBirthday(new XMLGregorianCalendarImpl(calendar));

        Customer c2 = new Customer();
        c2.setId(2);
        c2.setName("B");
        calendar
                .setTime(new
SimpleDateFormat("yyyy-MM-dd").parse("1990-01-28"));
        c2.setBirthday(new XMLGregorianCalendarImpl(calendar));

        System.out.println(helloService.selectMaxAgeStudent(c1,
c2).getName());
    }
}
```

这里要注意的就是 Customer 的生日字段 JAX-WS 在客户端映射为了 XMLGregorianCalendar 类型。我们运行这个客户端，结果输出 A，我们的 Web 服务调用成功。你还要注意 Web 服务调用可能经常出现超时的问题，但你切不可以为只要 WSDL 可以访问，就代表 Web 服务一定可以访问，因为是否可以访问与 SEI 的实现类有关，而 WSDL 仅是 SEI 的一种 XML 表示。

IV.SOAP 消息的格式：

我们从前面了解 WebService 使用 HTTP 协议传输消息，消息格式使用 SOAP，那么在客户端和服务器端传输的 SOAP 消息是什么样子的呢？下面我们将服务端 SoapServer.java 的代码改为如下的形式：

```
package net.ilkj.soap.server;

import org.apache.cxf.interceptor.LoggingInInterceptor;
import org.apache.cxf.interceptor.LoggingOutInterceptor;
import org.apache.cxf.jaxws.JaxWsServerFactoryBean;

public class SoapServer {

    public static void main(String[] args) {
        JaxWsServerFactoryBean soapFactoryBean = new
JaxWsServerFactoryBean();
        soapFactoryBean.getInInterceptors().add(new
LoggingInInterceptor());
        soapFactoryBean.getOutInterceptors().add(new
LoggingOutInterceptor());
        // 注意这里是实现类不是接口
        soapFactoryBean.setServiceClass(HelloServiceImpl.class);

    soapFactoryBean.setAddress("http://127.0.0.1:8080/helloService");
        soapFactoryBean.create();
    }

}
```

我们注意到这里将 javax.xml.ws.EndPoint 改为 CXF 特有的 API---JaxWsServerFactoryBean，并且我们对服务端工厂 Bean 的输入拦截器集合、输出拦截器集合中分别添加了日志拦截器（拦截器是 CXF 的一项扩展功能，CXF 提供了很多拦截器实现，你也可以自己实现一种拦截器），这样可以在 Web 服务端发送和接收消息时输出信息。

现在我们再次运行服务器端和客户端，你会看到控制台输出如下信息：

```
2009-6-17 22:35:57 org.apache.cxf.interceptor.LoggingInInterceptor
logging
信息: Inbound Message
----------------------------
ID: 2
Address: /helloService
Encoding: UTF-8
Content-Type: text/xml; charset=UTF-8
Headers: {content-type=[text/xml; charset=UTF-8],
connection=[keep-alive], Host=[127.0.0.1:8080], Content-Length=[367],
SOAPAction=[""], User-Agent=[Apache CXF 2.2.2], Content-Type=[text/xml;
```

```
charset=UTF-8], Accept=[*/*], Pragma=[no-cache],
Cache-Control=[no-cache]}
Payload: <soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"><soap:Body><ns
2:selectMaxAgeStudent
xmlns:ns2="http://server.soap.ilkj.net/"><c1><birthday>1989-01-28T00:
00:00.000+08:00</birthday><id>1</id><name>A</name></c1><c2><birthday>
1990-01-28T00:00:00.000+08:00</birthday><id>2</id><name>B</name></c2>
</ns2:selectMaxAgeStudent></soap:Body></soap:Envelope>
--------------------------------------
2009-6-17 22:35:57
org.apache.cxf.interceptor.LoggingOutInterceptor$LoggingCallback
onClose
信息: Outbound Message
---------------------------
ID: 2
Encoding: UTF-8
Content-Type: text/xml
Headers: {}
Payload: <soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"><soap:Body><ns
2:selectMaxAgeStudentResponse
xmlns:ns2="http://server.soap.ilkj.net/"><return><birthday>1989-01-28
T00:00:00+08:00</birthday><id>1</id><name>A</name></return></ns2:sele
ctMaxAgeStudentResponse></soap:Body></soap:Envelope>
--------------------------------------
```

Inbound Message 输出的是服务器端接收到的 SOAP 信息，Outbound Message 输出的服务器端响应的 SOAP 信息，SOAP 的 Headers:{}的前面是 SOAP 消息的标识、编码方式、MIME 类型，Headers:{}熟悉 HTTP 应该很容易看懂这里面的消息报头的作用，Headers:{}后面的 Payload（有效负载，也叫净荷）的 XML 就是 SOAP 消息的真正内容，我们看到 SOAP 消息内容被封装为<soap:Envelope …SOAP 信封，在信封之间的内容就是 SOAP 消息正文，这个元素还有一个子元素<soap:Header …，如果你的某些注解的 header=true，那么它将被放到<soap:Header …中传输，而不是 SOAP 消息正文。

例如我们把服务端的 IHelloService 接口改为如下的形式：

```
package net.ilkj.soap.server;

import javax.jws.WebParam;
import javax.jws.WebService;

@WebService
public interface IHelloService {

    Customer selectMaxAgeStudent(
            @WebParam(name = "c1", header = true) Customer c1,
```

```
            @WebParam(name = "c2") Customer c2);

    Customer selectMaxLongNameStudent(Customer c1, Customer c2);
}
```

我们注意第一个方法的第一个参数的 header=true，也就是放在 SOAP 的消息头中传输。然后我们重新生成客户端的代码，SoapClient 的调用代码改为如下的形式：

```
package net.ilkj.soap.client;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.GregorianCalendar;
import org.apache.cxf.jaxws.JaxWsProxyFactoryBean;
import
com.sun.org.apache.xerces.internal.jaxp.datatype.XMLGregorianCalendar
Impl;

public class SoapClient {

    public static void main(String[] args) throws ParseException {
        JaxWsProxyFactoryBean soapFactoryBean = new
JaxWsProxyFactoryBean();

    soapFactoryBean.setAddress("http://127.0.0.1:8080/helloService");
        soapFactoryBean.setServiceClass(IHelloService.class);
        Object o = soapFactoryBean.create();
        IHelloService helloService = (IHelloService) o;

        Customer c1 = new Customer();
        c1.setId(1);
        c1.setName("A");
        GregorianCalendar calendar = (GregorianCalendar)
GregorianCalendar
                .getInstance();
        calendar
                .setTime(new
SimpleDateFormat("yyyy-MM-dd").parse("1989-01-28"));
        c1.setBirthday(new XMLGregorianCalendarImpl(calendar));

        Customer c2 = new Customer();
        c2.setId(2);
        c2.setName("B");
        calendar
                .setTime(new
SimpleDateFormat("yyyy-MM-dd").parse("1990-01-28"));
```

```
        c2.setBirthday(new XMLGregorianCalendarImpl(calendar));
        SelectMaxAgeStudent sms = new SelectMaxAgeStudent();
        sms.setC2(c2);
        System.out.println(helloService.selectMaxAgeStudent(sms, c1)
                .getReturn().getName());
    }
}
```

我们注意到现在客户端的 IHelloService 的第一个方法的第一个参数是 SelectMaxAgeStudent，而不是 Customer，运行之后控制台输出如下语句：

```
2009-6-17 23:02:29 org.apache.cxf.interceptor.LoggingInInterceptor
logging
信息: Inbound Message
----------------------------
ID: 2
Address: /helloService
Encoding: UTF-8
Content-Type: text/xml; charset=UTF-8
Headers: {content-type=[text/xml; charset=UTF-8],
connection=[keep-alive], Host=[127.0.0.1:8080], Content-Length=[443],
SOAPAction=[""], User-Agent=[Apache CXF 2.2.2], Content-Type=[text/xml;
charset=UTF-8], Accept=[*/*], Pragma=[no-cache],
Cache-Control=[no-cache]}
Payload: <soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"><soap:Header><
ns2:c1
xmlns:ns2="http://server.soap.ilkj.net/"><birthday>1989-01-28T00:00:0
0.000+08:00</birthday><id>1</id><name>A</name></ns2:c1></soap:Header>
<soap:Body><ns2:selectMaxAgeStudent
xmlns:ns2="http://server.soap.ilkj.net/"><c2><birthday>1990-01-28T00:
00:00.000+08:00</birthday><id>2</id><name>B</name></c2></ns2:selectMa
xAgeStudent></soap:Body></soap:Envelope>
--------------------------------------
2009-6-17 23:02:29
org.apache.cxf.interceptor.LoggingOutInterceptor$LoggingCallback
onClose
信息: Outbound Message
---------------------------
ID: 2
Encoding: UTF-8
Content-Type: text/xml
Headers: {}
Payload: <soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"><soap:Body><ns
2:selectMaxAgeStudentResponse
```

```
xmlns:ns2="http://server.soap.ilkj.net/"><return><birthday>1989-01-28
T00:00:00+08:00</birthday><id>1</id><name>A</name></return></ns2:sele
ctMaxAgeStudentResponse></soap:Body></soap:Envelope>
--------------------------------------
```

我们注意到 Inbound Message 中的 SOAP 信封将第一个方法的第一个参数放在 <soap:Header … 中传输。

V.输入输出参数与@Oneway 注解：

在 SOAP 中方法的参数是有流向的，@WebParam 注解的 mode 属性由 javax.jws.WebParam.Mode 枚举指定，表示参数的流向，默认是 IN，也就是输入参数，还可以是 OUT、INOUT 类型。

如果是 OUT、INOUT 类型的参数类型，这样的方法参数将会被当做返回值在 Web 服务调用完成后返回给你，客户端生成代码时会被转变为 javax.xml.ws.Holder<T>类型，注意不要导错包，不是 javax.xml.rpc.Holder 类型（JDK1.6 已经没有这个类型，但是 MyEclipse 中还是会有，小心导入这个已经不使用的类型）。Holder 是一个泛型类，它持有类型 T。

@javax.jws.Oneway 注解是一个标识性注解，没有任何属性，它表示公开的 Web 服务的方法没有任何返回值，不允许有 OUT 类型的参数，不允许抛出非运行时异常。如果条件不符 JAX-WS 规范要求应该报告错误，但是 CXF 的策略是如果方法存在返回值，生成客户端时将被改为 void；如果方法参数含有 OUT 类型，生成客户端时将被忽略；如果方法含有 INOUT 类型参数，生成客户端时将只作为 IN 类型参数被保留。

例：

服务端 SEI：

```
@WebService
public interface IHelloService {

    boolean selectMaxAgeStudent(@WebParam(name = "c1") Customer c1,
            @WebParam(name = "c2") Customer c2,
            @WebParam(name = "c3", mode = Mode.OUT) Holder<Customer> c3);

    Customer selectMaxLongNameStudent(Customer c1, Customer c2);
}
```

客户端调用代码：

```
public class SoapClient {

    public static void main(String[] args) throws ParseException {
        JaxWsProxyFactoryBean client = new JaxWsProxyFactoryBean();

    client.setAddress("http://127.0.0.1:335/ws/services/helloService"
);
        client.setServiceClass(IHelloService.class);
        IHelloService helloService = (IHelloService) client.create();
        Customer c1 = new Customer();
```

```
        c1.setId(1);
        c1.setName("A");
        GregorianCalendar calendar = (GregorianCalendar)
GregorianCalendar
                .getInstance();
        calendar
                .setTime(new
SimpleDateFormat("yyyy-MM-dd").parse("1989-01-28"));
        c1.setBirthday(new XMLGregorianCalendarImpl(calendar));

        Customer c2 = new Customer();
        c2.setId(2);
        c2.setName("B");
        calendar
                .setTime(new
SimpleDateFormat("yyyy-MM-dd").parse("1990-01-28"));
        c2.setBirthday(new XMLGregorianCalendarImpl(calendar));
        Holder<Customer> ch = new Holder<Customer>();
        helloService.selectMaxAgeStudent(c1, c2, ch);
        System.out.println(ch.value.name);
    }

}
```

但是 CXF 很聪明，并不是你的服务端使用 OUT、INOUT 的 Holder<T>之后，他就一定会生成客户端时原样照做，例如：上面服务端方法如果将返回值改为 void，那么 CXF 将会在发布 WSDL 时将 c3 参数作为 WebService 操作的返回值，因此客户端就会和服务端不一样了，也就是客户端会生成如下的形式：

```
public net.ilkj.soap.client.Customer selectMaxAgeStudent(
        @WebParam(name = "c1", targetNamespace = "")
        net.ilkj.soap.client.Customer c1,
        @WebParam(name = "c2", targetNamespace = "")
        net.ilkj.soap.client.Customer c2
);
```

因为这确实是一个相等的转换，对于客户端来说，并不影响调用，不过这可能也与具体的 JAX-WS 的实现有关。

VI.Web 服务上下文：

javax.xml.ws.WebServiceContext 接口用于在 Web 服务实现类中访问与服务请求有关的消息上下文和安全信息，只需要使用 javax.annotation.Resource 这个标准的注解（EJB 等 JAVA EE 规范中的一些资源都使用这个注解注入）标注即可使用这个接口。

这个接口的 getMessageContext()方法返回 javax.xml.ws.handler.MessageContext 接口，这是一个实现了 Map 接口的接口，它包含了一组属性集。

例：

```
@Resource
private WebServiceContext context;

public void selectMaxAgeStudent() {
    MessageContext mContext = context.getMessageContext();
    Set<String> set = mContext.keySet();
    for (String key : set) {
           System.out.println("***********" + key + "\t" +
                mContext.get(key));
       try {
              System.out.println("++++++++++" +
    mContext.getScope(key));
           } catch (Exception e) {
              System.out.println("++++++++++" + key + "is not exits");
           }
    }
}
```

上面的方法会打印出出消息上下文中的所有属性集及其范围，因为 getScope(String name)如果没有 name 存在会抛出异常，我们这里捕获这个异常。范围的可选值是 javax.xml.ws.handler.MessageContext.Scope 中的 APPLICATION、HANDLER 枚举，前者表示属性对于处理程序（CXF）、客户端应用程序和 SEI 都是可见的，后者只对处理程序可见。

这个接口的其他两个方法用于处理 java.security.Principal 安全主体对象，这个对象用于处理请求中的验证的角色对象，一般我们不会用到这种方式去做 Web 服务的安全管理。

VII.使用客户端视图：

在前面我们看到服务端发布 Web 服务可以使用 javax.xml.ws.Endpoint 接口发布 Web 服务，这样你可以在开发 JAX-WS 的服务端时完全避开使用底层实现的 API，统一使用标准的 JAX-WS 的接口、注解等。但是在客户端访问 Web 服务时我们使用了 CXF 的 JaxWsProxyFactoryBean 来进行操作，其实你也可以使用标准的 JAX-WS 的 API 完成客户端调用。

例：

```
QName qName = new QName("http://server.soap.ilkj.net/",
              "HelloServiceImplService");
HelloServiceImplService helloServiceImplService =
       new HelloServiceImplService(
    new URL("http://127.0.0.1:8080/ws/services/helloService?wsdl"),
              qName);
IHelloService helloService = (IHelloService) helloServiceImplService
                                        .getPort(IHelloService.class);
```

首先我们使用 Web 服务的 WSDL 中的 targetNamespace 和<wsdl:service … 中的 name 属性构建了 javax.xml.namespace.QName 接口，然后我们调用生成的客户端代码中的客户端视图类（这个类继承 javax.xml.ws.Service） HelloServiceImplService（这个类名一般与

<wsdl:service …中的 name 属性值相同）的构造方法传入 WSDL 的 URL 对象和 QName 实例，在获得客户端视图实例之后调用 **T getPort(T t)**这个泛型方法找到要使用的端点服务接口。

上面我们说的查找参数都是到 WSDL 文件中去找，其实你打开客户端视图类同样可以找到 targetNamespace、服务名称、WSDL 的 URL 地址等信息。

这里你要注意客户端视图中的相关信息其实无关紧要，例如：一个公司中的两个部门的系统 A、B 要交互，那么在开发阶段，A 公开给 B 的 Web 服务的 WSDL 的 URL、名称空间等肯定都是测试机的信息（尤其是 URL 的 IP 地址），这样你生成的客户端视图类中的 WSDL、名称空间等都是针对测试的，那么在正式上线时，你将要访问 A 的线上 Web 服务，这时你不必担心客户端视图类中的相关信息是客户机的，因为无论是你用 JaxWsProxyFactoryBean 还是上面的方式，或者是下面的 Spring 的方式，Web 服务地址等信息都是需要重新设置的，当然你如果使用默认的构造方法（譬如上面的代码 new HelloServiceImplService()是使用无参的构造方法）或者是没有重新设置相关属性，那么将使用客户端视图类中的相关信息。

VIII.JAX-WS 的异常处理：

JAX-WS 中的服务端的自定义异常使用 javax.xml.ws.WebFault 注解来完成，这样的异常会在 WSDL 文件中的<wsdl:operation …中的子元素生成<wsdl:fault … 。下面我们来看一下一个示例代码：

```
@WebFault(name = "HelloServiceException")
public class HelloServiceException extends Exception {
    /**
     *
     */
    private static final long serialVersionUID = 1562884941631450124L;

    private HelloServiceFault details;

    public HelloServiceException(String msg) {
        super(msg);
    }

    public HelloServiceException(String msg, HelloServiceFault details)
{
        super(msg);
        this.details = details;
    }

    public HelloServiceException(HelloServiceFault details) {
        super();
        this.details = details;
    }

    public HelloServiceFault getFaultInfo() {
```

```
        return details;
    }

    @XmlRootElement(name = "HelloServiceFault")
    public static class HelloServiceFault {

        private String t;

        public HelloServiceFault() {
        }

        public HelloServiceFault(String t) {
            this.t = t;
        }

        public String getT() {
            return t;
        }

        public void setT(String t) {
            this.t = t;
        }

    }
}
```

这里需要注意一下几个问题：

(1.)自定义异常必须包含一个异常消息 msg 和一个封装具体错误消息的 Bean（这里是 HelloServiceFault），这个 Bean 上必须使用 JAXB 注解，这样他可以被转换为 SOAP 消息中的 XML 内容。

(2.)自定义异常中必须有一个 getFaultInfo()的方法返回封装具体错误消息的 Bean。

IX.使用 MTOM 传输附件：

MTOM（SOAP Message Transmission Optimization Mechanism）SOAP 消息传输优化机制，可以在 SOAP 消息中发送二进制数据，与 SAAJ 传输附件不同，MTOM 需要 XOP（XML-binary Optimized Packing）来传输二进制数据。MTOM 允许将消息中包含的大型数据元素外部化，并将其作为无任何特殊编码的二进制数据随消息一起传送。MTOM 消息会打包为多部分相关 MIME 序列，放在 SOAP 消息中一起传送。因此你可以看出 MTOM 并不是将附件转为 Base64 编码，这样可以大大的提高性能，因为二进制文件转 Base64 编码会非常庞大。MTOM 已经得到了大多数厂商的支持，包括微软等，所以使用这种方式处理 SOAP 中的附件，可以获得较大的通用性。

下面我们来看一个例子：

(1.)服务端的 Customer 类：

```
import java.util.Date;
```

```java
import javax.activation.DataHandler;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlMimeType;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "Customer")
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {

    private long id;

    private String name;

    private Date birthday;

    @XmlMimeType("application/octet-stream")
    private DataHandler imageData;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Date getBirthday() {
        return birthday;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    public DataHandler getImageData() {
```

```
        return imageData;
    }

    public void setImageData(DataHandler imageData) {
        this.imageData = imageData;
    }
}
```

这里我们看到 MTOM 方式中要传输的附件必须使用 javax.activation.DataHandler 类，然后这个类型还要使用@javax.xml.binding.annotation.XmlMimeType 进行注解，标注这是一个附件类型的数据，这里我们标注 imageData 是一个二进制文件，当然你也可以使用具体的 MIME 类型，譬如：image/jpg、image/gif 等，但你要考虑客户端是否有对应的类型（因为 JAVA 语言之外的客户端的特性未必是你完全了解的），而且 javax.activation.*中的 MIME 的相关 API 可以完成 MIME 类型的自动识别。

这里你要注意的是必须在类上使用@XmlAccessorType(FIELD)注解，标注 JAXB 在进行 JAVA 对象与 XML 之间进行转换时只关注字段，而不关注属性（getXXX()方法），否则发布 Web 服务时会报出现了两个 imageData 属性的错误，不知道这是不是 CXF 或者是底层 JAXB 实现的 BUG，因为它并没有报其他的属性因为 getXXX()方法的存在而视为重复属性。这个时侯发布的 WSDL 会包含如下的内容：

```
<xs:element minOccurs="0" name="imageData"
   ns1:expectedContentTypes="application/octet-stream"
     type="xs:base64Binary"
     xmlns:ns1="http://www.w3.org/2005/05/xmlmime" />
```

接下来你要在服务端和客户端分别启用 MTOM 支持，Spring 的配置文件如下所示：

```
<jaxws:properties>
    <entry key="mtom-enabled" value="true" />
</jaxws:properties>
```

这段内容加到<jaxws:server … 、<jaxws:endpoint … 、<jaxws:client … 之间即可，也就是作为他们的子元素存在。如果你想使用 Java Code 实现，你可以在服务端、客户端获取 javax.xml.ws.soap.SoapBinding 实例，然后调用它的 setMTOMEnabled(true)方法。其实从这里你可以 JAX-WS 是天然支持 MTOM 的，只不过默认禁用了这一功能，因为在没有附件这种大量数据要传输，MTOM 的优点并不会体现出来。

我们假设服务端 SEI 的实现的一个方法如下所示：

```
public Customer selectMaxLongNameStudent(Customer c1, Customer c2) {
    Customer rs = null;
    if (c1.getName().length() > c2.getName().length())
        rs = c1;
    else
        rs = c2;
        rs.setImageData(new DataHandler(new FileDataSource(
                new File("c:"+ File.separator + "18.jpg"))));
    return rs;
}
```

我们看到 DataHandler 需要 DataSource 进行构造，这里我们用到了 javax.activation.DataSource 的一个文件实现类来实现。

客户端调用代码如下所示：

```
…… 
String attachmentMimeType = helloService.selectMaxLongNameStudent(c1,
                c2).getImageData().getDataSource().getContentType();
System.out.println(attachmentMimeType);
```

你可以看到控制台输出 image/jpeg，MTOM 传输附件成功。如果你使用了日志拦截器，你会看到服务端的控制台打印出了很多乱码，这些乱码就是传输的附件。

如果你使用的是 MyEclipse 开发，你会看到在运行客户端时，有如下错误被抛出：
java.lang.NoClassDefFoundError: com/sun/mail/util/LineInputStream
这是由于 MyEclipse 在新建 Web 工程时给你添加的 javaee.jar 中的 javax.mail.*包与 JDK1.6 自带的 javax.mail.*包版本冲突，你只需要删除 MyEclipse 中的 javaee.jar 中的 mail 包即可。其实在 JDK1.6 之后你会经常运行程序报出找不到某个类或者是某个类转型失败的异常，主要原因是 JDK1.6 中把 JAVA EE5 中的许多规范都纳入进来，而且版本也比 JAVA EE5 要高，譬如 JAX-WS2.1 就比早一些发布的 JAVA EE5 的 JAX-WS2.0 要新，这种版本不一致的问题要注意。另外的原因就是很多人根本不清楚 JAVA EE 的各项规范，构建应用总是喜欢把一堆 jar 文件都弄进来，你会发现有些人即使在 JDK1.6 环境中构建企业级应用，依然会加入 jaxb-api.jar、jaxb-impl.jar、xalan.jar、xerces.jar、jsr181.jar、… …，要知道这些规范 JDK1.6 中已经包含并附带默认实现，这样不分清楚的加进上面的 jar 文件，极易造成与 JDK1.6 自带规范、默认实现冲突的问题，所以最好详细掌握 JAVA EE 的体系结构。

关于 SOAP 消息中传输附件的规定：

在使用 MTOM 传输附件时，控制台输出的信息我们摘入主要的如下所示：

```
…
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"><soap:Body><ns
2:selectMaxLongNameStudentResponse
xmlns:ns2="http://server.soap.ilkj.net/"><return><id>2</id><name>B</n
ame><birthday>1990-01-28T00:00:00+08:00</birthday><imageData><xop:Inc
lude xmlns:xop="http://www.w3.org/2004/08/xop/include"
href="cid:507f46da-6ff9-4174-8929-5448e128359b-1@http%3A%2F%2Fcxf.apa
che.org%2F"/></imageData></return></ns2:selectMaxLongNameStudentRespo
nse></soap:Body></soap:Envelope>
--uuid:f9eb12c5-740a-4afc-8be2-c94654e787c1
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-ID:
<507f46da-6ff9-4174-8929-5448e128359b-1@http://cxf.apache.org/>
附件的二进制代码（乱码）
--uuid:f9eb12c5-740a-4afc-8be2-c94654e787c1
…
```

我们看到二进制文件使用了--uuid:***（相当于二进制文件与 SOAP 其他消息部分的分隔符，这个分隔符由 Header 中的 boundary 属性值指定，从这里你可以看到 SOAP 消息的内容和 HTTP 中的各项规定没什么不同，熟悉 HTTP 的应该知道表单提交附件也是这种形式）进行包围，在顶端的-uuid:***之后跟着附件的 MIME 头信息，这里最主要的就是 Content-ID，如果你在一个 SOAP 消息中传输了多个附件，那么这个 Content-ID 必须是唯一的。我们再看看 SOAP 消息体中的<imageData>元素中使用了如下结构：

<xop:Include xmlns:xop="http://www.w3c.org/2004/08/xop/include" href="cid:MIME 头中的 Content-ID 去除左右两侧的<、>"/>

这段代码就是与下面的二进制文件关联的所在，也是上面我们说的那个 XOP。

那么我们如果禁用 MTOM，还要传递附件，此时，附件会被编为 BASE64 码进行传递，如下所示：

```
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"><soap:Body><ns
2:selectMaxLongNameStudent
xmlns:ns2="http://server.soap.ilkj.net/"><arg0><id>1</id><name>A</nam
e><birthday>1989-01-28T00:00:00.000+08:00</birthday><imageData>BASE64
</imageData></arg0><arg1><id>2</id><name>B</name><birthday>1990-01-28
T00:00:00.000+08:00</birthday></arg1></ns2:selectMaxLongNameStudent><
/soap:Body></soap:Envelope>
```

这种方式传递附件的缺点很明显，一个 10KB 的图片的 BASE64 码在 WORD 里都可以用四篇来显示，那么大一些的附件将会使得 XML 的体积迅速膨胀，这与 MTOM 的原样传输二进制数据是没有可比性的。

2.JAVA 的 WebService 规范 JAX-RS：

REST 是一种软件架构模式，只是一种风格，不是像 SOAP 那样本身承载着一种消息协议，（两种风格的 Web 服务均采用 HTTP 做传输协议是因为 HTTP 协议能穿越防火墙，JAVA 的远程调用 RMI 等是重量级协议，不能穿越防火墙），因此你也可以叫做 REST 是基于 HTTP 协议的软件架构。REST 中重要的两个概念就是资源定位和资源操作，而 HTTP 协议恰好完整的提供了这两个要点，HTTP 协议中的 URI 可以完成资源定位，GET、POST、OPTION 等方法可以完成资源操作，因此 REST 完全依赖 HTTP 协议就可以完成 Web 服务，而不像 SOAP 协议那样只利用 HTTP 的传输特性，定位与操作由 SOAP 协议自身完成，也正是由于 SOAP 消息的存在，使得 SOAP 笨重。你也可以说 REST 充分利用了 HTTP 协议的特性，而不是像 SOAP 那样只利用了其传输这一特性（事实上大多数人提到 HTTP 协议就只会想到它能用于数据传输）。

REST 对于 HTTP 的利用分为以下两种：首先是资源定位，这就是 URI，这本身并没有什么特别的，但要注意 REST 对 HTTP 的资源定位理解更加到位，也就是你的 Web 服务的 URI 要能足够表意，例如：http://www.fetion.com.cn/fetionwap/baby/getBabyInfoById?id=1，从 URI 上可以看出这个 Web 服务定位到的资源是查询飞信 WAP 宠物的信息，依据参数 id 值查询。那么可以继续出现以下层级：

http://www.fetion.com.cn/fetionwap/baby/storeroom/getStoreRoomById?id=1

http://www.fetion.com.cn/fetionwap/baby/storeroom/chicken/getCounts?id=1

我们看到 REST 风格的 URI 的目录层级足够表意，也就是资源定位，这种定位要求 URI 是唯一的。因为 REST 流行于互联网，网上的资源应该有唯一的资源位置（例如：图片、视频）。当然，如果你的服务越复杂，URI 可能就越长，越难理解，这也算是 REST 风格的缺点。
第二种就是利用 HTTP 的 GET、POST、PUT、DELETE 四种操作外加 HEAD 请求报头完成资源操作，你可以把前四种 HTTP 的操作类比成数据库操作的 SELECT、UPDATE、INSERT、DELETE 操作，有这几种最简单的操作任意组合就可以完成各种各样的复杂操作，当然这是 REST 的理念，事实上这样创建应用有点儿牵强。

REST 是一种软件架构理念，现在被移植到 Web 服务上（因此不要提到 REST 就马上想到 WebService，JAX-RS 只是将 REST 设计风格应用到 Web 服务开发），那么在开发 Web 服务上，偏于面向资源的服务适用于 REST，偏于面向活动的服务。另外，REST 简单易用，效率高，SOAP 成熟度较高，安全性较好。
REST 提供的网络服务叫做 OpenAPI，它不仅把 HTTP 作为传输协议，也作为处理数据的工具，可以说对 HTTP 协议做了较好的诠释，充分体现了 HTTP 技术的网络能力。目前 Google、Amazon、淘宝都有基于 REST 的 OpenAPI 提供调用。

JAX-RS 的 API 在 javax.ws.rs.*包中，其中大部分也是注解。
我们先看一个较为简单的示例：
(1.)IStudentService.java：

```
package net.ilkj.rest.server;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;

@Path(value = "/student/{id}")
@Produces("application/xml")
public interface IStudentService {
    @GET
    @Path(value = "/info")
    Student getStudent(@PathParam("id") long id, @QueryParam("name")
String name);

    @GET
    @Path(value = "/info2")
    Student getStudent(@QueryParam("name") String name);
}
```

说明：
1.这个 REST 的服务接口的最终响应结果是 XML（@Produces 注解标注，这个注解可以包含一组字符串，默认值是*/*，它指定 REST 服务的响应结果的 MIME 类型，例如：application/xml、application/json、image/jpeg 等），你也可以同时返回多种类型，但具体生

成结果时使用哪种格式取决于 ContentType。CXF 默认返回的是 JSON 字符串。
2.访问方法 URI 是/student/1/info?name=Andrew-Lee、/student/1/info2?name=Fetion，由@Path 注解组合而来；
3.@QueryParam 注解用于指定将 URL 上的查询参数传递给使用这个注解的属性值；
4.@PathParam 注解用于指定将 URL 上的路径参数作为使用这个注解的属性值。
5.@GET 注解指定方法对应于 Http 的 GET 请求。JAX-RS 提供 javax.ws.rs.HttpMethod 注解允许你增加除了@GET 等之外的方法，例如：你想定义一个新的 HTTP 方法@PATCH，你可以这样像下面这样编写代码：

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@HttpMethod("PATCH")
public @interface PATCH {

}
```

但你要注意，你新增加的这个方法，Web 服务器一定要支持才可以，如果不支持，你就需要配置你的 Web 服务器，譬如上面定义的@PATCH 注解指定的 PATCH 方法在标准的 Http 方法中根本就不存在。

(2.)StudentServiceImpl.java：

```
package net.ilkj.rest.server;

import java.text.ParseException;
import java.text.SimpleDateFormat;

public class StudentServiceImpl implements IStudentService {

    public Student getStudent(long id, String name) {
        Student s = new Student();
        s.setId(id);
        s.setName(name);
        try {
            s.setBirthday(new SimpleDateFormat("yyyy-MM-dd")
                    .parse("1983-04-26"));
        } catch (ParseException e) {
            e.printStackTrace();
        }
        return s;
    }
```

```
    public Student getStudent(String name) {
        Student s = new Student();
        s.setId(1);
        s.setName(name);
        try {
            s.setBirthday(new SimpleDateFormat("yyyy-MM-dd")
                    .parse("1983-04-26"));
        } catch (ParseException e) {
            e.printStackTrace();
        }
        return s;
    }
}
```

(3.)Student.java：

```
package net.ilkj.rest.server;

import java.util.Date;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name="Student")
public class Student {
    private long id;
    private String name;
    private Date birthday;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Date getBirthday() {
        return birthday;
```

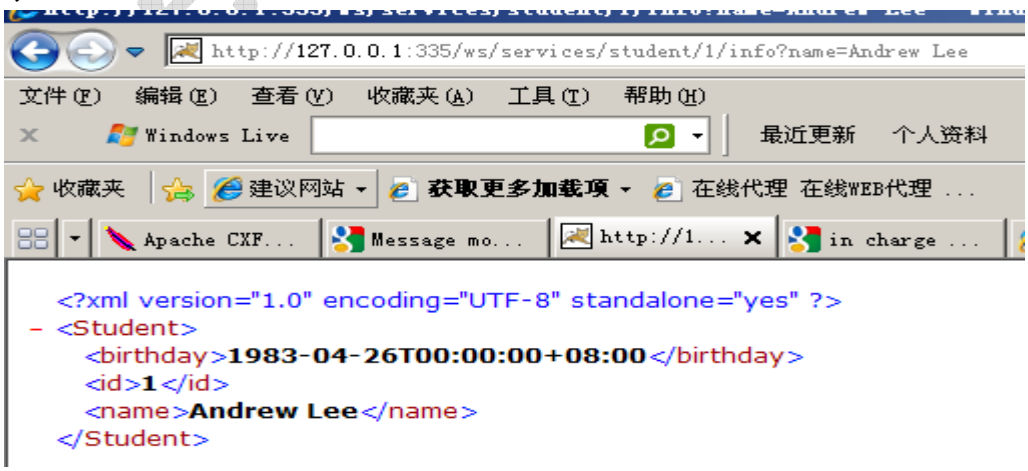
```
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }
}
```

因为最终的产出结果是 XML，因此你需要对返回值 Student 使用 JAXB 注解，这样 CXF 才会知道如何转换。

(4.)发布 REST 服务（Jetty）：

```
package net.ilkj.rest.server;

import org.apache.cxf.jaxrs.JAXRSServerFactoryBean;
import org.apache.cxf.jaxrs.lifecycle.SingletonResourceProvider;

public class RestServer {

    public static void main(String[] args) {
        JAXRSServerFactoryBean sf = new JAXRSServerFactoryBean();
        sf.setResourceClasses(StudentServiceImpl.class);
        sf.setAddress("http://localhost:335/");
        sf.create();
    }

}
```

访问 REST 风格的 Web 服务很简单，你完全不需要像 SOAP 使用 SDK 生成客户端代码，因为 REST 完全依赖 HTTP 协议，从这里可见 REST 是轻量级的。我们访问 http://127.0.0.1:335/ws/services/student/1/info?name=Andrew Lee 地址，可以看见页面如下所示：



我们看到 Web 服务访问成功，当然这个响应的 XML 客户端通常要接到并处理，所以你可以用**((HttpURLConnection)new URL("***").openConnection()).getInputStream()**获取，然后使用程序处理这个接收到的结果。为了更加好的使用 HTTP 访问 REST 服务，推荐你使用

Apache HttpComponents-Client 组件进行 HTTP 操作，因为这里使用的示例是 GET 方法的请求，你是用 URL 直接访问或者使用 java.net.URL 类来访问很容易，但是如果 Web 服务的方法使用@PUT 等注解，那么你就需要费一番头脑来在请求报头中加入要请求的方法类型等信息，这些是很繁琐的事情。HTTP-Client 的访问代码如下所示：

```
package net.ilkj.rest.client;

import java.io.IOException;
import java.io.InputStream;
import org.apache.http.HttpResponse;
import org.apache.http.client.ClientProtocolException;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.DefaultHttpClient;

public class RestClient {

        public static void main(String[] args) throws
                        ClientProtocolException,IOException {
        HttpGet get = new HttpGet(
                "http://127.0.0.1:8080/student/info2?name=Fetion");
        HttpClient httpclient = new DefaultHttpClient();
        HttpResponse response = httpclient.execute(get);
        InputStream ins = response.getEntity().getContent();
        byte[] b = new byte[1024];
        StringBuilder sb = new StringBuilder();
        while (ins.read(b) != -1) {
            sb.append(new String(b, "UTF-8"));
        }
        System.out.println(sb.toString());

    }

}
```

注意上面只是简单示例，使用的是逐个字节读入的方式，因此最好不要在输出的 XML 中出现非 ISO-8859-1 的字符。如果公开的 Web 服务是 PUT 方法，那么你可以使用 HttpPut 类来完成处理。

CXF 中的 JAX-WS 的一些设置对于 JAX-RS 也同样有效，例如日志拦截器，但不是所有的都可以使用，例如：后面讲到的 WSS4J 的拦截器就不能使用到 JAX-RS 上，因为 WS-*是 SOAPWeb 服务的相关规范。下面是服务端被访问时输出的日志信息：

```
2009-6-23 21:55:05 org.apache.cxf.interceptor.LoggingInInterceptor
logging
信息: Inbound Message
----------------------------
```

```
ID: 1
Address: /ws/services/student/info2
Encoding: UTF-8
Content-Type:
Headers: {connection=[Keep-Alive], host=[127.0.0.1:335],
user-agent=[Apache-HttpClient/4.0-beta2 (java 1.5)],
Content-Type=[null]}
Payload:
--------------------------------------
2009-6-23 21:55:06
org.apache.cxf.interceptor.LoggingOutInterceptor$LoggingCallback
onClose
信息: Outbound Message
---------------------------
ID: 1
Encoding:
Content-Type: application/xml
Headers: {Date=[Tue, 23 Jun 2009 13:55:05 GMT]}
Payload: <?xml version="1.0" encoding="UTF-8"
standalone="yes"?><Student><birthday>1983-04-26T00:00:00+08:00</birth
day><id>1</id><name>Fetion</name></Student>
--------------------------------------
```

I.JAX-RS 的方法返回值：

JAX-RS 的接口方法可以返回 javax.ws.rs.core.Response 接口或者是自定义类型（譬如上面的 Student 类型），Response 接口可以返回 Http 的响应代码、响应头或者是一种实体，javax.ws.rs.ext.MessageBodyWriter 类负责 marshall 响应实体，响应实体被直接输出（譬如上面的 Student）或者是作为 Response 的一部分。

那么与之对应的是 MessageBodyReader 负责 Unmarshall 响应实体，也就是读取实体。同样的，前面提到的@Produces 用于指示一个资源类（服务接口）或者 MessageBodyWriter 可以产出的 MIME 类型，@Consumes 用于指示资源类（服务接口）或者 MessageBodyReader 可以接受的 MIME 类型。

下面我们举一个如何返回 Response 的例子，上面的 StudentServiceImpl 代码改造之后如下所示：

(1.)服务接口：

```
@Path(value = "/student/{id}")
public interface IStudentService {
    @GET
    @Path(value = "/info")
    @Produces("application/xml")
    Response getStudent(@PathParam("id") long id,
            @QueryParam("name") String name);
```

```
    @GET
    @Path(value = "/info2")
    Response getStudent(@QueryParam("name") String name);
}
```

注意这里我们将@Produces 注解移到方法上，也就是你可以为每个服务方法单独指定输出类型。

(2.)服务实现类：

```
public class StudentServiceImpl implements IStudentService {

    public Response getStudent(long id, String name) {
        Student s = new Student();
        s.setId(id);
        s.setName(name);
        try {
            s.setBirthday(new SimpleDateFormat("yyyy-MM-dd")
                    .parse("1983-04-26"));
        } catch (ParseException e) {
            e.printStackTrace();
        }
        return Response.ok(s).build();
    }

    public Response getStudent(String name) {
        return Response.status(Response.Status.BAD_REQUEST).build();
    }
}
```

我们看到第一个方法返回的 Response 包装了一个实体，第二个方法返回 Http 的响应代码。

(3.)客户端访问代码：

```
HttpGet get = new HttpGet(

    "http://127.0.0.1:335/ws/services/student/1/info2?name=Andrew-Lee
");
HttpClient httpclient = new DefaultHttpClient();
HttpResponse response = httpclient.execute(get);
StatusLine st = response.getStatusLine();
if (st.getStatusCode() == HttpServletResponse.SC_OK) {
    InputStream ins = response.getEntity().getContent();
    byte[] b = new byte[1024];
    StringBuilder sb = new StringBuilder();
    while (ins.read(b) != -1) {
        sb.append(new String(b, "UTF-8"));
    }
```

```
    System.out.println(sb.toString());
    } else {
        System.out.println(st.getStatusCode());
}
```

这里我们只有在响应代码是 200 的时候才输出响应信息，其余都输出响应代码，这里你会看到响应代码 400 被输出，也就是服务实现类中设置的 BAD_REQUEST。

II.关于 Response 接口：

这个类有两个静态的内部类，ResponseBuilder 和 Status。

ResponseBuilder 用于创建 Response 实例，一般是 Response 先通过自己的方法（**ok()、status()、notModified()**等）获得 ResponseBuilder 实例，然后 ResponseBuilder 进行构建（设置响应头、最后修改时间、Cookie、语言、MIME 类型等），你会发现这个类的所有方法都返回这个类，这样你可以使用方法链编程调用多个方法，最后使用 ResponseBuilder 的 build()方法返回 Response 实例，也就是构建完毕。

(1.)ResponseBuilder 设置 MIME 类型使用 **type(javax.ws.rs.core.MediaType type)**方法，MediaType 类中定义了大量的 MIME 类型常量，以及几个简单的处理方法。

(2.)ResponseBuilder 中的方法 **tag(javax.ws.rs.core.EntityTag tag)**用于设置 HTTP 的响应头的 ETag，ETag 是 HTTP 中一种与 Web 资源关联的记号，具体请参考 HTTP 文档。

(3.)ResponseBuilder 中的方法 **variant(javax.ws.rs.core.Variant)**、**variants(List<Variant> list)**用于简化设置响应头。Variant 的构造方法同时接受 MediaType、Locale、字符编码三个参数。

(4.)ResponseBuilder 中的方法 cacheControl(javax.ws.rs.core.CacheControl cacheControl)方法用于设置缓存实例，CacheControl 类的方法设置缓存在客户端的存在时间。

(5.)ResponseBuilder 中的方法 cookie(NewCookie… cookies)用于设置 HTTP 的 Cookie，这里接受的是一个可变参数。

Status 用于获取响应状态码，其中定义了大量的响应代码常量，以及几个简单的处理方法。

II.JAX-RS 中的异常处理：

在 JAX-RS 中你有如下三种方式处理异常：

(1.)直接使用 Response 返回 HTTP 响应代码，例如：400、500 等表示错误的响应代码。

(2.)将异常或错误代码包装为 javax.ws.rs.WebApplicationException：

例如：**throw new WebApplicationException(new MyException("***"));**

throw new WebApplicationException(404);

这是一个运行时异常，不需要显示捕获处理。

(3.)将异常转换为响应结果：

```
public class StudentException extends RuntimeException {

    private static final long serialVersionUID = 1899623964871050278L;
```

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

import javax.ws.rs.core.Response;
import javax.ws.rs.ext.ExceptionMapper;
import javax.ws.rs.ext.Provider;

@Provider
public class StudentExceptionMapper implements
        ExceptionMapper<StudentException> {

    @Override
    public Response toResponse(StudentException se) {
        return
Response.status(Response.Status.INTERNAL_SERVER_ERROR).build();
    }

}
```

这里我们使用 javax.ws.rs.ext.ExceptionMapper<E>接口将异常映射为 Response 类型，上面我们看到将自定义的运行时异常 StudentException 映射为 500 的 Response 类型。注意这个 Mapper 必须使用@Provider 进行标注，这样你的服务实现类在抛出 StudentExcpetion 之后一律会被转换为 Response 为 500 的响应结果。

III.JAX-RS 的参数处理：

前面我们看到@QueryParam、@PathParam 注解用于获取查询参数、路径参数，另外，@MatrixParam、@FormParam、@HeadParam、@CookieParam 注解用于获取 Martix 参数、POST 方式提交的参数、HTTP 请求头中的参数、Cookie 参数。@DefaultValue 注解用于对这六个注解设置在未取到值的情况下的默认值，@Encoded 用于对禁用前四种注解的自动解码。

前面的例子都是一个一个字符串的传递，你也可以直接使用复合类型。

例：

```
@Path(value = "/student/{id}")
public interface IStudentService {

    @GET
    @Path(value = "/info/{id}/{name}")
    Response getStudent(@PathParam("") Student student);
}
```

我们看到@PathParam 的 value 属性为空字符串，这表示将从路径上将和 Student 中的属性名相同的路径参数逐个传入。我们访问 http://127.0.0.1:335/ws/services/student/1/info/2/m.j 地址，

你可以看到 Student 的 id=1，name=m.j，这里注意到如果有重复的路径参数，那么第一个起作用。

现在我们再来看一下如何使用@MatrixParam 注解，这个注解用于取 xxx;A=a;B=b;...的以;分隔的 A、B、...的值。
例：

```
@Path(value = "/student/{id}")
public interface IStudentService {

    @GET
    @Path(value = "/info2/matrix")
    Response getStudent(@MatrixParam("") Student student);
}
```

然后我们访问 http://127.0.0.1:335/ws/services/student/1/info2/matrix;id=2;name=m.j 地址即可接收到值。这个注解获取参数内部是使用 javax.ws.rs.core.PathSegment 接口的实现类（CXF 中是 org.apache.cxf.jaxrs.impl.PathSegmentImpl）来完成，用于获取 ; 分隔的路径片段中的参数。

IV.Web 服务类的生命周期：

在没有 Spring 的情况下，JAX-RS 的所有服务类都是在每次请求时重新创建，但大多数情况下，公开 Web 服务的类都是 Service 层，由于 Service 层都是无状态的，所以我们最好使用单例模式，代码如下所示：

```
JAXRSServerFactoryBean sf = new JAXRSServerFactoryBean();
sf.setResourceClasses(StudentServiceImpl.class);
sf.setResourceProvider(StudentServiceImpl.class,
          new SingletonResourceProvider(new StudentServiceImpl()));
sf.setAddress("http://localhost:8080/");
sf.create();
```

我们注意第三行使用了单例资源提供器，而不是默认的多实例资源提供器。
那么在 Spring 容器中，默认情况下由于 bean 的 scope 是 singleton，所以是单例模式的，如果你想使用多例模式，可以设置 scope 为 prototype，但一般我们没有必要这样做。

V.@Context 注解：

你可以使用@javax.ws.rs.core.Context 注解将 **UriInfo, SecurityContext, HttpHeaders, Providers, Request, ContextResolver, HttpServletRequest, HttpServletResponse, ServletContext, ServletConfig** 实例注入到服务实现类，当然其实你也可以使用标准注解@javax.annotation.Resource（但不建议这么做，否则某些情况下会出现引用错误）。
例：

```
public class StudentServiceImpl implements IStudentService {

    @Context
    private ServletContext servletContext;
```

```
    @Resource
    private MessageContext messageContext;

    public Response getStudent(long id, String name) {
        System.out.println(servletContext.getContextPath() + "\t"
                + messageContext.getUriInfo().getPath() + "\t");
        … …
    }
}
```

上面的例子我们可以输出 Web 应用上下文、访问当前方法的 REST 路径。

VI.资源路径嵌套：

首先我们看一个例子：

(1.)服务接口：

```
@Path(value = "/student/{id}/")
@Produces("application/xml")
public interface IStudentService {

    @Path(value = "info/")
    Student getStudent(@PathParam("id") long id,
            @DefaultValue("Andrew Lee") @QueryParam("name") String name);

    @GET
    @Path(value = "info/matrix")
    Response getStudent(@MatrixParam("") Student student);
}
```

(2.)Student 类：

```
@XmlRootElement(name = "Student")
public class Student {
    private long id;
    private String name;
    private Date birthday;

    … …

    @GET
    @Path("score/{name}")
    public Score getScore(@PathParam("name") String name) {
        Score score = new Score();
        score.setName(name);
        score.setNum(100);
```

```
        return score;
    }
}
```

然后我们访问 http://127.0.0.1:8080/ws/services/student/1/info/score/math?name=Andrew Lee 地址，即可看到 Score 的 Json 的内容被输出，之所以输出的不是服务接口上的制定的 XML，这是因为递归查找时会以最后找到的路径所在的类的@Produces 注解为主，而我们没有给 Student 定义产出类型，那么 CXF 输出的就是默认的 JSON 格式。这里我们要说的是 JSON 和 XML 相比的优势是在大数据量的情况下，JSON 字符串要比 XML 字符串在容量上小得多得多，传输速度肯定也会快得多得多，也就是它更适合在网络上传输，但你会发现 JSON 的字符串越多，可读性越差。
使用这种深度路径需要注意以下几点：
(1.)@GET 等 HTTPMETHOD 注解必须放在递归的最内层的方法，因此你看到在服务接口中的第一个方法上没有出现这个注解，这时会提示 CXF 第一个方法的返回值中会有需要继续递归查找的方法。
(2.)子资源（SubResource）Student 中的 getScore()方法的@Path 路径应该是一个相对路径，也就是最好不要用 / 开头。

这种子资源的路径如果每次请求时都递归查找，很消耗资源，你可以像预编译正则表达式一样，启用静态子资源解析，你可以使用如下的方式：

```
JAXRSServerFactoryBean sf = new JAXRSServerFactoryBean();
sf.setStaticSubresourceResolution(true);
```

在 Spring 配置文件中也可以如下配置：

```
<jaxrs:server staticSubresourceResolution="true">
    ......
</jaxrs:server>
```

其实这种 SubResource 并不常用，因为这样会让关系变得复杂。但这里我们要说的是一个 REST 路径的命名习惯：
(1.)最顶层的根@Path 的路径要以 / 开头；
(2.)不是根@Path 的不要以 / 开头，如果其后还有子路径被递归嵌套，那么请以 / 结尾。
遵循这个原则，你可以看到第二个方法与之前不同，info 前面的 / 被删除了，matrix 后面也没有 / 。

VII.使用 WebClient 类：
前面我们都是使用了 HTTP-Components-Client 的 API 来访问 REST 服务，这也是比较干净的方式，所谓干净就是完全依赖 HTTP 的 API（其实更加干净的方式就是使用 JAVA 自带的 HTTP 的 API 操作），但是过于干净的调用方式总会很麻烦（很明显使用 HTTP-Components-Client 要比 java.net.*下面的 API 要来得简单），其实 CXF 自带的 **org.apache.cxf.jaxrs.client.WebClient** 用起来更加简单。
例：

```
WebClient client = WebClient
```

```
    .create("http://127.0.0.1:8080/ws/services/student/1/");
Student student = client.path("info/matrix;id=2;name=m.j").accept(
                "application/xml").get(Student.class);
System.out.println(student.getName());
```

WebClient 的 API 大都返回这个类本身的实例，因此上面我们使用了方法链编程，可以连续调用 WebClient 的 API。

我们看到这种调用方式不仅更加符合 REST 风格的路径组装的方式，也可以直接把接收到的响应结果直接转换为 JAVA 对象。你可能要问客户端是如何得到 Student 类的呢？不要忘了 REST 是轻量级的，没有 SOAP 的 SDK 可以生成客户端代码，这个 Student 类实际就是根据对方公开给你的 REST 服务的返回值说明文档自己构建出来的，因为公开 REST 服务和 SOAP 服务的不同之一就是你要给出 REST 服务的返回结果的说明，假设是 XML，那么你需要给出文档结果，以及 XML 中的元素、属性的说明，通过这些返回值说明你就可以构建出客户端用户接收它的类，当然转换过程是由 JAXB 来自动完成的。具体如何公开 REST 服务可以参考阿里巴巴的 OpenAPI 文档中的例子。

3.CXF 与 Spring：

CXF 可以很好的与 Spring 整合，这样可以为我们省去很多的代码，你需要的是简单的 Spring 配置即可。

I.CXF 发布在 Web 服务器：

我们前面都是使用 CXF 自带的 Jetty 发布 Web 服务，如果我们的 Web 服务在 Web 服务器中编写，那么使用现有的 Web 服务器自然是更好的选择。

通常 CXF 在 Web 服务器发布 Web 服务都是通过 Spring 容器完成，但是 CXF 也可以完全脱离 Spring 容器独立运行在 Web 容器中，你需要书写一个类覆盖 org.apache.cxf.transport.servlet.CXFNonSpringServlet 的 loadBus 方法指定 BUS 以及发布你的 Web 服务。

我们的 Web 应用上下文是/ws。

(1.)web.xml：

```
<servlet>
        <servlet-name>CXFServlet</servlet-name>
        <servlet-class>
            net.ilkj.servlet.MyCXFNonSpringServlet
        </servlet-class>
        <init-param>
            <param-name>/helloService</param-name>
            <param-value>
                net.ilkj.soap.server.HelloServiceImpl
            </param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
<servlet-mapping>
        <servlet-name>CXFServlet</servlet-name>
        <url-pattern>/services/*</url-pattern>
```

```
</servlet-mapping>
```

你可以配置多个初始化参数，指定你要发布的Web服务实现类。

(2.)MyCXFNonSpringServlet.java：

```
package net.ilkj.servlet;

import java.util.Enumeration;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.xml.ws.Endpoint;
import org.apache.cxf.Bus;
import org.apache.cxf.BusFactory;
import org.apache.cxf.transport.servlet.CXFNonSpringServlet;

public class MyCXFNonSpringServlet extends CXFNonSpringServlet {

    /**
     *
     */
    private static final long serialVersionUID = 1930791254280865620L;

    @Override
    public void loadBus(ServletConfig servletConfig) throws
ServletException {
        super.loadBus(servletConfig);
        Bus bus = this.getBus();
        BusFactory.setDefaultBus(bus);

        // 获取在web.xml中配置的要发布的所有的Web服务实现类并发布Web服务
        Enumeration<String> enumeration = getInitParameterNames();
        while (enumeration.hasMoreElements()) {
            String key = enumeration.nextElement();
            String value = getInitParameter(key);
            try {
                Class clazz = Class.forName(value);
                try {
                    Endpoint.publish(key, clazz.newInstance());
                } catch (InstantiationException e) {
                    e.printStackTrace();
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                }
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
```

```
            }
        }
    }
}
```

为了测试结果，你可以暂时先删除 lib 目录下 Spring 的所有 jar 文件，然后我们访问 http://127.0.0.1:335/ws/services/helloService?wsdl，我们看到 Web 服务发布成功。

II.使用 Spring 发布 SOAP 方式的 Web 服务：

使用 Spring 发布 Web 服务很简单，首先将 web.xml 改为如下的形式：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/beans.xml</param-value>
</context-param>
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
<servlet>
    <servlet-name>CXFServlet</servlet-name>
    <servlet-class>
        org.apache.cxf.transport.servlet.CXFServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>CXFServlet</servlet-name>
    <url-pattern>/services/*</url-pattern>
</servlet-mapping>
```

我们看到 CXFServlet 负责截获/services/*的请求，Web 服务的 SEI 由 Spring 的上下文加载监听器来完成查找。那么下面我们来配置 Spring，beans.xml 如下所示：

```
<?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:jaxws="http://cxf.apache.org/jaxws"
    xmlns:jaxrs="http://cxf.apache.org/jaxrs"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://cxf.apache.org/jaxws
        http://cxf.apache.org/schemas/jaxws.xsd
        http://cxf.apache.org/jaxrs
```

```
        http://cxf.apache.org/schemas/jaxrs.xsd">
    <import resource="classpath:META-INF/cxf/cxf.xml" />
    <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml"
/>
    <import
resource="classpath:META-INF/cxf/cxf-extension-jaxrs-binding.xml" />
    <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />
    <jaxws:endpoint id="helloServiceWs" address="/helloService"
        implementor="#helloService" />
    <bean id="helloService"
          class="net.ilkj.soap.server.HelloServiceImpl" />
</beans>
```

我们注意到这里引入了两个新的名称空间 jaxws、jaxrs，因为 CXF 实现了 Spring 的 NamespaceHandler 接口，实现这个接口可以在 Spring 中增加额外的配置。

那么 jaxws 自然是配置 SOAP 方式的 Web 服务，你可以看到有 jaxws:server、jaxws:endpoint、jaxws:client 三个元素，jaxws:server 和 jaxws:endpoint 是等效的，都用于发布 Web 服务，出现 jaxws:endpoint 的原因是 JAX-WS 规范中使用 EndPoint 发布 Web 服务（前面使用过这种方式），CXF 为了和 JAX-WS 对应，提供了这个与 jaxws:server 功能一样的配置元素；jaxrs 是 REST 方式的 Web 服务，有 jaxrs:server、jaxrs:client 两个元素。

你也可以使用 implementorClass 属性直接指向一个类，而不是像上面那样引用一个 Bean 的名字。

我们启动 Web 服务器，访问 http://127.0.0.1:335/ws/services/helloService?wsdl 地址，如果你看到 WSDL，那么表示我们的 Web 服务发布成功。你也可以使用下面的方式发布 Web 服务：

```
<jaxws:server id="helloServiceWs" address="/helloService">
    <jaxws:serviceBean>
        <ref bean="helloService" />
    </jaxws:serviceBean>
</jaxws:server>
```

同样，你也可以使用 serviceClass 属性指向一个类，而不是使用子元素 jaxws:serviceBean 引用一个 Bean 的名字。无论是哪种方式，指向的目标都是 SEI 的实现类。

III.使用 Spring 开发 SOAP 方式的客户端：

```
<jaxws:client id="helloServiceClient"
        address="http://127.0.0.1:335/ws/services/helloService"
    serviceClass="net.ilkj.soap.client.IHelloService"/>
```

这里属性 serviceClass 指向客户端生成的接口，这里就不能指向一个 Bean 的名字，因为接口是不能被实例化的。然后你就可以向访问 Spring 容器中的 Bean 一样去访问这个 CXF 的客户端代理（JaxWsClientProxy），下面是我们在 JSP 中书写的调用代码：

```
IHelloService helloService =
          (IHelloService) WebApplicationContextUtils
                    .getWebApplicationContext(application)
                    .getBean("helloServiceClient");
out.print(helloService.selectMaxAgeStudent(c1, c2).getName());
```

从 II、III 的示例可以看出借助 Spring 在 Web 容器中发布、访问 Web 服务极其简单。
关于具体的<jaxws: … 配置项，大多数都可以从元素、属性的名字上看出其作用，具体解释请参看 http://cwiki.apache.org/CXF20DOC/jax-ws-configuration.html。

IV.使用 Spring 发布 REST 风格的 Web 服务：

Spring 发布 REST 风格的 Web 服务与 SOAP 方式的 Web 服务没有什么区别，beans.xml 代码如下所示：

```
<jaxrs:server id="studentServiceWs" address="/">
        <jaxrs:serviceBeans>
            <ref bean="studentService" />
        </jaxrs:serviceBeans>
</jaxrs:server>
<bean id="studentService"
            class="net.ilkj.rest.server.StudentServiceImpl"/>
```

我们看到发布服务时使用了 / ，也就是在 CXFServlet 拦截的/services/*之后直接跟 IStudentService 的@Path 的路径，不再额外追加其他的路径值。访问这个 Web 服务可以直接使用前面的方式访问。CXF 提供了<jaxrs:client ...标记访问 REST 服务，但这个标记需要给出 serviceClass 属性，然后内部使用 JAXRSClientFactoryBean 类来完成创建，这样就似乎必须得有服务接口才可以完成客户端调用配置，可是我一直没想通客户端怎么可能得到 REST 服务的接口呢？好在 WebClient 已经足够好用，所以我们就不必非得用这个<jaxrs:client …来完成客户端调用。

4.SOAP 的 WS-*规范：

基于 SOAP 的 Web 服务的 WS-*规范有不下几十种，但对于 JAX-WS 规范只定义了最基本的 Web 服务的规范，并未定义通用的 WS-*规范的通用接口，所以可能不同的 JAX-WS 实现提供的 WS-*规范实现的数量是不一样的。目前 CXF 支持如下几种 WS-*的规范：

(1.)WS-Addressing：实现了与底层传输协议的隔离，寻址方式采取基于消息的路由，同时也实现了对会话状态的保存机制（Web 服务 SEI 本身应该是无状态、自包含的）。

(2.)WS-Reliable Messaging：这个规范实现了可靠的消息传递，也就是保证消息从发送者正确传输到接受者。

(3.)WS-Security*和 WS-Policy、WS-Trust：这一系列规范实现了 SOAP 的服务的安全策略、信任机制。

这里我们讲一下较为常用的安全策略机制。

I.传统的用户名令牌机制：

Apache WSS4J（WebService Security For Java）实现了 JAVA 语言的 WS-Security，CXF 中使用 WSS4J 也是很容易的，WSS4J 依赖于 SAAJ。

CXF 中使用拦截器机制完成 WSS4J 功能的支持，你只需要初始化 WSS4JInInterceptor（对

应的还有一个 WSS4JOutInterceptor）实例并添加相关信息即可，CXF2.1.x 之后的版本可以完成 SAAJInInterceptor（它也有一个对应的 SAAJOutInterceptor）拦截器的自动注册，否则你需要再注册一下 SAAJ 拦截器。

下面我们看一个简单的示例：
(1.)服务器端：

```
<bean id="wss4jInInterceptor"
class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
    <constructor-arg>
        <map>
            <entry key="action" value="UsernameToken" />
            <entry key="passwordType" value="PasswordText" />
            <entry key="passwordCallbackClass"

    value="net.ilkj.soap.server.security.ServerPasswordCallbackHandle
r" />
        </map>
    </constructor-arg>
</bean>
<jaxws:server id="helloServiceWs" address="/helloService">
    <jaxws:serviceBean>
        <ref bean="helloService" />
    </jaxws:serviceBean>
    <jaxws:inInterceptors>
        <ref bean="wss4jInInterceptor" />
    </jaxws:inInterceptors>
</jaxws:server>
```

这里我们使用构造方法参数初始化了一个输入的 WSS4J 拦截器，到底有哪些参数的键值对，可以在 org.apache.ws.security.handler.WSHandlerConstants 和 org.apache.ws.security.WSConstants 中的产量列表中查找。例如：上面的第一组键值对 action 和 UsernameToken 都是 WSHandlerConstants 中的常量，表示验证机制是用户姓名令牌，也就是使用传统的用户名和密码机制。第二组的键值对分别是 WSHandlerConstants 和 WSConstants 中的常量，表示密码类型是文本，还可以是 WSConstants.PASSWORD_DIGEST（密码会被加密为 MD5）。第三组键值对的键表示服务器端验证密码的回调处理类，这个类必须 JAVA 安全认证框架中的 javax.security.auth.callback.CallbackHandler 类，你也可以用 passwordCallbackRef 指向一个 Bean 的名字。
下面我们看一下服务器端的这个密码回调处理类，它负责接收并处理客户端提交的用户名和密码，我们看到这个方法没有返回值，很显然，如果验证失败，你需要抛出异常进行表示。

```
package net.ilkj.soap.server.security;

import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
```

```
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.ws.security.WSPasswordCallback;

public class ServerPasswordCallbackHandler implements CallbackHandler {

    public final static String USER = "Fetion2";

    public final static String PASSWORD = "Fetion";

    @Override
    public void handle(Callback[] callbacks) throws IOException,
            UnsupportedCallbackException {
        WSPasswordCallback wspassCallback = (WSPasswordCallback)
callbacks[0];
        System.out.println(wspassCallback.getIdentifier() + "\t"
                + wspassCallback.getPassword());
        if (wspassCallback.getIdentifier().equals(USER)
                && wspassCallback.getPassword().equals(PASSWORD)) {
            // undo
        } else {
            throw new WSSecurityException("No Permission!");
        }
    }

}
```

(2.)客户端：

```
<bean id="wss4jOutInterceptor"
class="org.apache.cxf.ws.security.wss4j.WSS4JOutInterceptor">
    <constructor-arg>
        <map>
            <entry key="action" value="UsernameToken" />
            <entry key="user" value="Fetion" />
            <entry key="passwordType" value="PasswordText" />
            <entry key="passwordCallbackClass"

    value="net.ilkj.soap.client.security.ClientPasswordCallbackHandle
r" />
        </map>
    </constructor-arg>
</bean>
<jaxws:client id="helloServiceClient"
        address="http://127.0.0.1:335/ws/services/helloService"
        serviceClass="net.ilkj.soap.client.IHelloService">
```

```
    <jaxws:outInterceptors>
        <ref bean="wss4jOutInterceptor" />
    </jaxws:outInterceptors>
</jaxws:client>
```

我们看到这里的区别是使用了 WSS4J 的输出拦截器，因为你是要将用户名和密码输出到服务端进行验证处理。另外的不同是多个一个 user 参数初始化 WSS4J 的输出拦截器，用于初始化用户名，这是一个必选项，稍后我们将在下面的客户端密码回调处理类进行重新设定值。

```
package net.ilkj.soap.client.security;

import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.ws.security.WSPasswordCallback;

public class ClientPasswordCallbackHandler implements CallbackHandler {

    public final static String USER = "Fetion2";

    public final static String PASSWORD = "Fetion";

    @Override
    public void handle(Callback[] callbacks) throws IOException,
            UnsupportedCallbackException {
        WSPasswordCallback wspassCallback = (WSPasswordCallback)
callbacks[0];
        wspassCallback.setIdentifier(USER);
        wspassCallback.setPassword(PASSWORD);
    }

}
```

然后我们访问这个 Web 服务，从控制台查看日志，你可以看到在 SOAP 信封的 Header 中包装了<wsse:Security … 等元素，元素包括了 WS-Security 的一些信息和我们设置的用户名和密码（明文）。

下面我们看一下密码使用 MD5 密文发送的代码，beans.xml 中只需要将 passwordType 的值变为 PasswordDigest 即可，客户端回调类不需要变化，这里我们看一下服务端回调类。

```
…… 
@Override
public void handle(Callback[] callbacks) throws IOException,
            UnsupportedCallbackException {
        WSPasswordCallback wspassCallback =
```

```
                                (WSPasswordCallback) callbacks[0];
        System.out.println(wspassCallback.getIdentifier() + "\t"
                + wspassCallback.getPassword());
        if (WSConstants.PASSWORD_TEXT.
            equals(wspassCallback.getPasswordType())) {
            if (wspassCallback.getIdentifier().equals(USER)
                    && wspassCallback.getPassword().equals(PASSWORD)) {
                // undo
            } else {
                throw new WSSecurityException("No Permission!");
            }
        } else {
            System.out.println(wspassCallback.getIdentifier());
            wspassCallback.setPassword(PASSWORD);
        }

    }
```

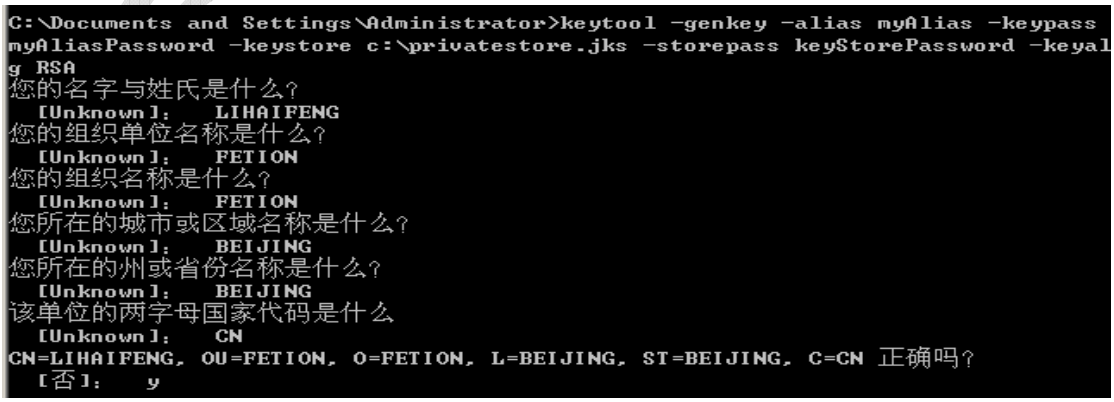
……

你可以设置断点，你会发现 Digest 密文密码情况下，WSPasswordCallback 的 passwordType 属性和 password 属性都为 null，你只能获得用户名（identifier），一般这里的逻辑是使用这个用户名到数据库中查询其密码，然后再设置到 password 属性，WSS4J 会自动比较客户端传来的值和你设置的这个值。你可能会问为什么这里 CXF 不把客户端提交的密码传入让我们在 ServerPasswordCallbackHandler 中比较呢？这是因为客户端提交过来的密码在 SOAP 消息中已经被加密为 MD5 的字符串，如果我们要在回调方法中作比较，那么第一步要做的就是把服务端准备好的密码加密为 MD5 字符串，由于 MD5 算法参数不同结果也会有差别，另外，这样的工作 CXF 替我们完成不是更简单吗？

II.数字签证方法：

除了 UsernameToken 这种传统的安全机制，常用的验证动作还有一个就是使用数字签证技术（X.509 Certificates），action 的取值为 Signature 这需要你了解 JAAS 中的相关内容，由于商业证书都是收费的，这里我们采用自签名的方法，也就是 JDK 自带的 keytool 命令。

(1.)首先我们生成密钥文件（证书）：



这个密钥的密码是 myAliasPassword，使用 RSA 算法生成密钥。

(2.)对密钥文件自签名：

```
C:\Documents and Settings\Administrator>keytool -selfcert -alias myAlias -keysto
re c:\privatestore.jks -storepass keyStorePassword -keypass myAliasPassword
```

注意各参数值的药使用上面输入的，例如：-storepass 是输入访问这个密钥文件的授权密码。

(3.)从签名后的私有密钥文件中导出公有密钥：

```
C:\Documents and Settings\Administrator>keytool -export -alias myAlias -file c:\
key.rsa -keystore c:\privatestore.jks -storepass keyStorePassword -keypass myAli
asPassword
保存在文件中的认证 <c:\key.rsa>
```

(4.)将公有密钥导入一个公有密钥文件：

```
C:\Documents and Settings\Administrator>keytool -import -alias myAlias -file c:\
key.rsa -keystore c:\publicstore.jks -storepass keyStorePassword
所有者:CN=LIHAIFENG, OU=FETION, O=FETION, L=BEIJING, ST=BEIJING, C=CN
签发人:CN=LIHAIFENG, OU=FETION, O=FETION, L=BEIJING, ST=BEIJING, C=CN
序列号:4a4c15cf
有效期: Thu Jul 02 10:05:03 CST 2009 至Wed Sep 30 10:05:03 CST 2009
证书指纹:
        MD5:9F:03:68:10:1B:85:4E:1F:10:42:48:0F:DA:3D:34:FE
        SHA1:CC:AE:7A:75:65:AF:E3:D5:9F:FF:3D:CD:13:4D:3A:EC:03:C1:52:64
        签名算法名称:SHA1withRSA
        版本: 3
信任这个认证? [否]:  y
认证已添加至keystore中
```

上面简单演示了数字签证的制作过程，下面我们用批处理的方式生成一对服务端/客户端密钥文件：

(1.)generateKeyPair.bat：

```
rem @echo off
echo alias %1
echo keypass %2
echo keystoreName %3
echo KeyStorePass %4
echo keyName %5

keytool -genkey -alias %1 -keypass %2 -keystore %3 -storepass %4    -dname "cn=%1" -keyalg RSA
keytool -selfcert -alias %1 -keystore %3 -storepass %4 -keypass %2
keytool -export -alias %1 -file %5 -keystore %3 -storepass %4
```

(2.)generateServerKey：

```
call generateKeyPair.bat apmserver apmserverpass serverStore.jks keystorePass serverKey.rsa
pause
call generateKeyPair.bat apmclient apmclientpass clientStore.jks keystorePass clientKey.rsa
pause
keytool -import -alias apmserver -file serverKey.rsa -keystore clientStore.jks -storepass keystorePass
pause
```

keytool -import -alias apmclient -file clientKey.rsa -keystore serverStore.jks -storepass keystorePass

我们运行第二个批处理文件，即可得到一对服务端/客户端密钥文件：clientStore.jks/serverStore.jks。

我们将这两个 jks 放到类路径，并新建两个属性配置文件，如下图所示：

当然实际应用中这两对配置文件肯定是分开放在服务端和客户端的。

(1.) server_sign.properties：

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=keystorePass
#org.apache.ws.security.crypto.merlin.alias.password=apmserverpass
org.apache.ws.security.crypto.merlin.keystore.alias=apmserver
org.apache.ws.security.crypto.merlin.file=serverStore.jks
```

(2.) client_sign.properties：

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=keystorePass
#org.apache.ws.security.crypto.merlin.alias.password=apmclientpass
org.apache.ws.security.crypto.merlin.keystore.alias=apmclient
org.apache.ws.security.crypto.merlin.file=clientStore.jks
```

(3.)Spring 的配置文件：

```
<bean id="wss4jInInterceptor"
class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
    <constructor-arg>
        <map>
            <entry key="action" value="Signature" />
            <entry key="user" value="apmclient" />
            <entry key="passwordCallbackClass"

    value="net.ilkj.soap.server.security.ServerPasswordCallbackHandler" />
            <entry key="signaturePropFile"
value="server_sign.properties"></entry>
```

```
            </map>
        </constructor-arg>
    </bean>
    <jaxws:server id="helloServiceWs" address="/helloService">
        <jaxws:serviceBean>
            <ref bean="helloService" />
        </jaxws:serviceBean>
        <jaxws:inInterceptors>
            <ref bean="wss4jInInterceptor" />
        </jaxws:inInterceptors>
    </jaxws:server>
    <bean id="wss4jOutInterceptor"
    class="org.apache.cxf.ws.security.wss4j.WSS4JOutInterceptor">
        <constructor-arg>
            <map>
                <entry key="action" value="Signature" />
                <entry key="user" value="apmclient" />
                <entry key="passwordCallbackClass"

        value="net.ilkj.soap.client.security.ClientPasswordCallbackHandle
    r" />
                <entry key="signaturePropFile"
    value="client_sign.properties"></entry>
            </map>
        </constructor-arg>
    </bean>
    <jaxws:client id="helloServiceClient"
            address="http://127.0.0.1:8080/ws/services/helloService"
    serviceClass="net.ilkj.soap.client.IHelloService">
        <jaxws:outInterceptors>
            <ref bean="wss4jOutInterceptor" />
        </jaxws:outInterceptors>
    </jaxws:client>
```

(4.)密码回调处理类就是把证书密码（记住不是 keyStore 的密码）设置给 WSS4J，我们这里只给出客户端处理类，服务器端的处理类就是密码换成服务端的证书密码。

```
public class ClientPasswordCallbackHandler implements CallbackHandler {

    @Override
    public void handle(Callback[] callbacks) throws IOException,
            UnsupportedCallbackException {
        WSPasswordCallback wspassCallback = (WSPasswordCallback)
callbacks[0];
        wspassCallback.setPassword("apmclientpass");
```

}

}

(5.)访问 Web 服务：

… …

helloService.selectMaxAgeStudent(c1, c2).getName()

… …

如果你使用了日志拦截器，你会看到 SOAP 信封中包含了很多<ds:… 的元素，里面封装了数字签证的相关信息。

这个例子中的服务端的密码处理类可以省去，但你要把服务端的属性配置文件中的那行注释放开，区别是密码放在回调类中比明文写在属性配置文件中安全，但是客户端只能使用密码回调类的方式设置证书密码。

由于数字签证使用的是 HTTPS 协议（SSL/TLS），你可以使用 Transport 中的 HTTP 通道的元素 http-conf:tlsClientParameters 配置 SSL/TLS 的相关信息。

III.混合验证方法：

WSS4J 支持如下几种验证模式：

XML Security

XML Signature

XML Encryption

这两种验证模式都是使用数字签证技术。

Tokens

Username Tokens

Timestamps

SAML Tokens

上面的两个示例我们分别使用了 Username Tokens、XMLSignature 验证模式，这两种也是较为常用的验证模式，但实际上 WSS4J 也支持你混合使用上述的几种验证模式，例如：你可以将 Spring 中的 WSS4J 的 Interceptor 的 action 属性指定为 Timestamp Encrypt Signature，也就是使用这三种验证模式。这也是为什么回调方法中的 WSPasswordCallback 类型是一个数组的原因，你可以依据 getPasswordType()方法返回的密码类型为不同的验证模式设置密码。

不管是哪种验证模式，如果你想使用 Java Code 方式实现，而不是 Spring 的配置文件，那么可以如下操作：

Map<String,Object> inProps= new HashMap<String,Object>();

... // configure the properties

WSS4JInInterceptor wssIn = new WSS4JInInterceptor(inProps);

然后将拦截器添加到拦截器列表即可完成操作。

5.Transport：

传输端口其实在前面的 CXFServlet 和 CXFNonSpringServlet 就是一种 ServletTransport，CXF 使用这两个传输端口发布 Web 服务。下面我们看一下 CXF 中最常需要配置的 HTTP 传输端口，另外，CXF 还支持 JMS Transport，但不常使用。

HTTP Transport：

Web 服务都是使用 HTTP 作为传输协议，这个端口用于配置服务端、客户端在调用 Web 服务时的 HTTP 的相关设置，例如：超时时间，SSL 相关设置、是否启用缓存等。

(1.)客户端调用：

```
<http-conf:conduit name="*.http-conduit">
    <http-conf:client ConnectionTimeout="5000"
            ReceiveTimeout="10000" />
</http-conf:conduit>
```

这里的*.http-conduit 是指对所有的 Web 服务调用起作用，如果你想对一部分起作用，可以使用**{targetNamespace}serviceName.http-conduit** 来设置。

http-conf:conduit 是客户端配置的顶级元素，它有如下几个子元素：

Element	Description
http-conf:client	指定 HTTP 的超时时间、是否启用持续连接、ContentType 等信息。
http-conf:authorization	指定 HTTP 基本验证方式的相关配置。.
http-conf:proxyAuthorization	指定 HTTP 基本验证方式时使用的代理服务器配置。
http-conf:tlsClientParameters	指定 SSL/TLS 连接方式的配置。
http-conf:basicAuthSupplier	指定 HTTP 基本验证方式的提供者信息。
http-conf:trustDecider	指定 HTTP 连接的信任机制配置。

上面我们配置的是客户端连接 Web 服务、接收返回值的超时时间设置，CXF 的默认设置为 30000ms 和 60000ms。

(2.)服务端调用：

```
<http-conf:destination name="*.http-destination">
    <http-conf:server ReceiveTimeout="10000" />
</http-conf:destination>
```

这里的*.http-destination 是指对所有的 Web 服务发布起作用，如果你想对一部分起作用，可以使用**{targetNamespace}serviceName.http-destination** 来设置。

http-conf:destination 是客户端配置的顶级元素，它有如下几个子元素：

Element	Description
http-conf:server	指定 HTTP 的连接设置信息。
http-conf:contextMatchStrategy	指定上下文匹配策略。

http-conf:fixedParameterOrder	指定是否固定参数拍学。

(3.)Java Code 调用：

如果你不使用 Spring，那么使用 Java Code，那么你需要借助 CXF 的 Front End 的 API 操作。

服务端：

```
import org.apache.cxf.endpoint.Server;
import org.apache.cxf.frontend.ServerFactoryBean;
import org.apache.cxf.transport.http_jetty.JettyHTTPDestination;
import org.apache.cxf.transports.http.configuration.HTTPServerPolicy;

public class SoapServer {

    public static void main(String[] args) {
        ServerFactoryBean serverFactoryBean = new ServerFactoryBean();
        serverFactoryBean

    .setAddress("http://127.0.0.1:8080/ws/services/helloService");
        serverFactoryBean.setServiceClass(HelloServiceImpl.class);
        Server server = serverFactoryBean.create();
        JettyHTTPDestination destination = (JettyHTTPDestination) server
                .getDestination();
        HTTPServerPolicy httpServerPolicy = new HTTPServerPolicy();
        httpServerPolicy.setReceiveTimeout(32000);
        destination.setServer(httpServerPolicy);
    }

}
```

客户端：

```
import java.net.MalformedURLException;
import java.net.URL;
import java.text.ParseException;
import javax.xml.namespace.QName;
import org.apache.cxf.endpoint.Client;
import org.apache.cxf.frontend.ClientProxy;
import org.apache.cxf.transport.http.HTTPConduit;
import org.apache.cxf.transports.http.configuration.HTTPClientPolicy;

public class SoapClient {

    public static void main(String[] args) throws ParseException,
            HelloException, MalformedURLException {
        QName qName = new QName("http://server.soap.ilkj.net/",
                "HelloServiceImpl");
```

```
        HelloServiceImplService helloServiceImplService = new
HelloServiceImplService(
                new
URL("http://127.0.0.1:8080/ws/services/helloService?wsdl"),
                qName);
        IHelloService helloService = helloServiceImplService
                .getPort(IHelloService.class);
        Client client = ClientProxy.getClient(helloService);
        HTTPConduit http = (HTTPConduit) client.getConduit();
        HTTPClientPolicy httpClientPolicy = new HTTPClientPolicy();
        httpClientPolicy.setConnectionTimeout(36000);
        httpClientPolicy.setReceiveTimeout(32000);
        http.setClient(httpClientPolicy);
        // helloService.***
    }

}
```

关于front-end and back-end：

Front-end和back-end是用于定义与用户相关的服务的程序接口和服务的。这里的用户可以是人也可以是程序。front-end应用程序是由应用程序用户直接参与完成的，而back-end应用程序却非直接支持front-end 服务，它们一般与需要的资源非常靠近，有能力与这些资源通信。back-end应用程序可以直接与front-end应用程序通信，或者，更普遍的是，与被称为中间程序的应用程序通信。

使用CXF的Front End的API，你可以调用更多的方法，但是你也看到操作起来也是最繁琐的，所以除非你有必要使用CXF的这么多特性，否则不要使用这种方式来操作JAX-WS。

6.CXF 的拦截器特征机制:

我们在前面看到 **CXF 通过拦截器（Interceptor）和特征（Feature）扩展自己的功能，例如：WS-Addressing 功能实用 Feature 实现，日志、WS-Security 使用 Interceptor 实现。**

我们也可以编写自己的拦截器注册到 CXF 中完成特定的功能。CXF 中的所有拦截器都要事先 org.apache.cxf.inrerceptor.Interceptor<T extends org.apache.cxf.message.Message>接口，Message 接口可以获得 SOAP 消息的相关信息。通过查看 CXF 的 API 文档，你会看到 CXF 已经实现了很多种拦截器，很多已经在发布、访问 Web 服务时已经默认添加到拦截器链。一般情况下，我们自己的拦截器只要继承 AbstractPhaseInterceptor<T extends org.apache.cxf.message.Message>类即可，这个类可以指定继承它的拦截器在什么阶段被启用，阶段属性可以通过 org.apache.cxf.phase.Phase 中的常量指定值。

例:

```
package net.ilkj.soap.server.interceptor;

import org.apache.cxf.interceptor.Fault;
import org.apache.cxf.message.Message;
```

```
import org.apache.cxf.phase.AbstractPhaseInterceptor;
import org.apache.cxf.phase.Phase;

public class HelloInInterceptor extends
AbstractPhaseInterceptor<Message> {

    public HelloInInterceptor(String phase) {
        super(phase);
    }

    public HelloInInterceptor() {
        super(Phase.RECEIVE);
    }

    @Override
    public void handleMessage(Message message) throws Fault {
        System.out.println("******************");
    }

}
```

你要注意 CXF 中的拦截器编写时不要只针对服务端或者客户端，应该是两者均可使用，另外名字要见名知意。例如：使用 In、Out 标注这是一个输入时起作用还是输出时起作用的拦截器。上面的 HelloInInterceptor 由于在构造方法中指定在接收消息阶段有效，所以即使你把它注册到 OutInterceptor 的集合中也无效。具体关于 CXF 中拦截器的内容需要时请参看 http://cwiki.apache.org/CXF20DOC/interceptors.html。

同样，我们也可以通过继承 AbstractFeature 类来实现一个新的特征，只需要覆盖 initializeProvider 方法即可。其实 Feature 就是将一组拦截器放在其中，然后一并注册使用。

例:

```
package net.ilkj.soap.server.feature;

import org.apache.cxf.Bus;
import org.apache.cxf.feature.AbstractFeature;
import org.apache.cxf.interceptor.InterceptorProvider;
import org.apache.cxf.interceptor.LoggingInInterceptor;
import org.apache.cxf.interceptor.LoggingOutInterceptor;

public class HelloFeature extends AbstractFeature {

    @Override
    protected void initializeProvider(InterceptorProvider provider, Bus
bus) {
        provider.getInInterceptors().add(new LoggingInInterceptor());
        provider.getOutInterceptors().add(new
```

```
LoggingOutInterceptor());
    }

}
```

这个特征类将日志拦截器捆绑在一起，你就可以将它注册到你要使用的地方，而不必一个一个拦截器的注册使用。

CXF 除了允许在 Spring 中的配置文件、硬编码注册拦截器和特征类，也允许你是用其自带的注解@InInterceptors、@OutInterceptors、@InFaultInterceptors、@OutFaultInterceptors、@Features 来直接注册到 SEI，但是不推荐你这么做，因为这样你的类中就耦合和 CXF 这个具体的 JAX-WS 实现的 API。

7.JAX-WS 的异步调用：

Web 服务的调用默认都是阻塞调用，但是 JAX-WS 也支持异步调用。

第一步，我们要在 wsdl2java 上做文章，你需要生成支持异步调用的客户端存根代码，那么我们需要新建文件 **async_binding.xml**，内容如下所示：

```
<bindings
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    wsdlLocation="http://127.0.0.1:8080/ws/services/helloService?wsdl"
    xmlns="http://java.sun.com/xml/ns/jaxws">
    <bindings node="wsdl:definitions">
        <enableAsyncMapping>true</enableAsyncMapping>
    </bindings>
</bindings>
```

注意红色部分是你的 WSDL 的物理位置，然后在 wsdl2java 命令中追加 **–b async_binding.xml**，注意位置关系，这里我们假设运行的 wsdl2java 命令与该文件在同一文件夹下。Wsdl2java 的-b 参数指定多个以空格分隔的 JAX-WS、JAXB 绑定文件。

一个如下所示的服务端接口：

```
@WebService
public interface IHelloService {

    Customer selectMaxAgeStudent(@WebParam(name = "c1") Customer c1,
            @WebParam(name = "c2") Customer c2);

    Customer selectMaxLongNameStudent(Customer c1, Customer c2);
}
```

生成的支持异步客户端的 Stub 接口如下所示：

```
public interface IHelloService {


    public Response<SelectMaxAgeStudentResponse>
```

```
        selectMaxAgeStudentAsync(Customer c1,Customer c2);

    public Future<?> selectMaxAgeStudentAsync(Customer c1,Customer c2,
           AsyncHandler<SelectMaxAgeStudentResponse> asyncHandler);


    public Customer selectMaxAgeStudent(Customer c1,Customer c2);

    public Response<SelectMaxLongNameStudentResponse>
          selectMaxLongNameStudentAsync(Customer arg0,Customer arg1);

    public Future<?> selectMaxLongNameStudentAsync(Customer arg0,
                          Customer arg1,
        AsyncHandler<SelectMaxLongNameStudentResponse> asyncHandler);

    public Customer selectMaxLongNameStudent(Customer arg0,
                                  Customer arg1);

}
```

这里我们为了看起来不混乱，删去了所有的标注。我们看到原有的 SEI 中的两个方法分别增加了两个以 Async 结尾的新方法（现在一共是六个方法）。这两个增加的新方法一个是保持参数不变，返回值为 javax.xml.ws.Response<原方法对应的 SOAP 响应结果类型>；另一个是增加新的 javax.xml.ws.AsyncHandler<原方法对应的 SOAP 响应结果类型>参数，返回结果变为 java.util.concurrent.Future<?>类型，熟悉 JDK1.5 并法包的应该知道这个类是干什么用的。

第二步，我们开始在客户端进行异步调用，很显然，你应该调用那两个以 Async 结尾的方法中的一个，那么为什么会有这两个异步调用方法呢？这是因为 JAX-WS 中存在两种客户端异步调用方式：

(1.)Polling：这种方式调用返回结果为 Response<T>的异步调用方法，它被称为轮询方式，所谓轮询就是不断的查询结果是否返回，有点儿类似于 Socket 监听。

(2.)Callback：这种方式调用返回结果为 Future<?>异步调用方法，它被称为回调方式，所谓回调就是在有消息返回时调用回调方法。

其实 Response 继承 Future 类，因此两种方法都是使用 JDK1.5 的异步计算类 Future 来完成的，只不过回调方法需要你再额外编写一个 AsyncHandler 的回调方法，你可以在这里干一些其他的事情。

HelloAsyncHandler 类：

```
public class HelloAsynchHandler implements
       AsyncHandler<SelectMaxAgeStudentResponse> {
    private SelectMaxAgeStudentResponse reply;
```

```
    @Override
    public void handleResponse(Response<SelectMaxAgeStudentResponse>
res) {
        try {
            System.out.println("handleResponse called");
            reply = res.get();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    public Customer getResponse() {
        return reply.getReturn();
    }
}
```

客户端调用代码：

```
JaxWsProxyFactoryBean client = … …
IHelloService helloService = (IHelloService) client.create();
Customer c1 = … …
Customer c2 = … …

Customer resp;

// Callback
HelloAsynchHandler helloAsyncHandler = new HelloAsynchHandler();
Future<?> response = helloService.selectMaxAgeStudentAsync(c1, c2,
              helloAsyncHandler);
System.out.println("Other Things...");
while (!response.isDone()) {
    Thread.sleep(100);
}
resp = helloAsyncHandler.getResponse();
System.out.println("Server responded through callback with: "
              + resp.getName());
System.out.println("----------------------------");

// polling method
Response<SelectMaxAgeStudentResponse> selectMaxAgeStudentResponse =
             helloService.selectMaxAgeStudentAsync(c1, c2);
System.out.println("Other Things...");
while (!selectMaxAgeStudentResponse.isDone()) {
    Thread.sleep(100);
}
```

```
SelectMaxAgeStudentResponse reply = selectMaxAgeStudentResponse.get();
System.out.println("Server responded through polling with: "
            + reply.getReturn().getName());
System.exit(0);
```

上面的 Other Things … 是你要操作的其他业务逻辑，通常 Future 的使用情景是将它放在代码的最开始，然后去干其他的事情，因为在 Future 的 get()方法被调用之前，它不会产生阻塞，这样你可以在方法执行的最后再去获取异步计算的结果。那么你可能会问，这样整个方法还是不能算作完全的异步调用，因为最后要返回值的时候还是会阻塞哦！其实有返回值的 Web 服务调用本就不应该使用异步调用，因为返回值通常你都是要拿回来立即使用的。这种异步调用机制适合于无返回值的 Web 服务调用，这样你就不需要再调用 Future 的 get()方法了。

如果你想有返回值的时候也进行异步调用，那么就只能使用最原始的开启新的线程的方法，也就是将 Web 服务方法的调用放在 run()方法中执行。

8.JAX-WS 的 RPC/encoded 问题：

首先我们看一下 javax.jws.soap.SOAPBinding 注解，这个注解有如下三个属性：

属性	取值 1（默认）	取值 2
style	SOAPBinding.Style.DOCUMENT	SOAPBinding.Style.RPC
use	SOAPBinding.Use.LITERAL	SOAPBinding.Use.ENCODED
parameterStyle	SOAPBinding.ParameterStyle.WRAPPED	SOAPBinding.ParameterStyle.BARE

style 表示 SOAP 消息样式，有文档和 RPC 两种样式，每一种样式可以使用字符和编码两种方式包含消息，对于参数可以使用包装模式（所有参数包装到一个 Message）和赤裸模式（所有参数独立存在）。至于它们具体的不同可以在 SEI 上使用 SOAPBinding 注解设置，然后观察生成的 WSDL 就可以看出区别。

那么对于 JAX-WS 来说，前面多次提到，它不支持 RPC/encoded，也就是你不能在 SEI 上使用如下的注解组合：

@SOAPBinding(style = SOAPBinding.Style.RPC, use = SOAPBinding.Use.ENCODED)

CXF 在发布 Web 服务时会忽略掉 ENCODED，所以你也不会看到任何错误发生（这个与 JAX-WS 的具体实现有关，其他的实现可能会抛出异常）。

但是试想一下，如果发布 Web 服务的一方是较早构建的，那么它可能公开给你的就是 JAX-RPC 的服务，一旦又是 RPC/encoded 方式的，那么 CXF 将无法访问，如果你是用 wsdl2java.bat 你将会看到 RPC/encoded not supported by JAX-WS 2.0 的错误提示。

这种情况你通常不大可能会说服对方改换 JAX-WS，那么我们就要要使用 JAX-RPC 的 Web 服务实现来访问对方的 Web 服务，这里简单说一下使用 Apache Axis 1.4 访问 RPC/encoded 的 Web 服务，注意我们使用的 Axis，不是 Axis 2，这是两个差别很大的 Web 服务实现，Axis 2 实现的是 JAX-WS、JAX-RS，而 Axis 实现的是 JAX-RPC。

我们在安装完 Axis 之后，将其自带的 lib 目录中的 jar 添加到 classpath，运行如下命令：

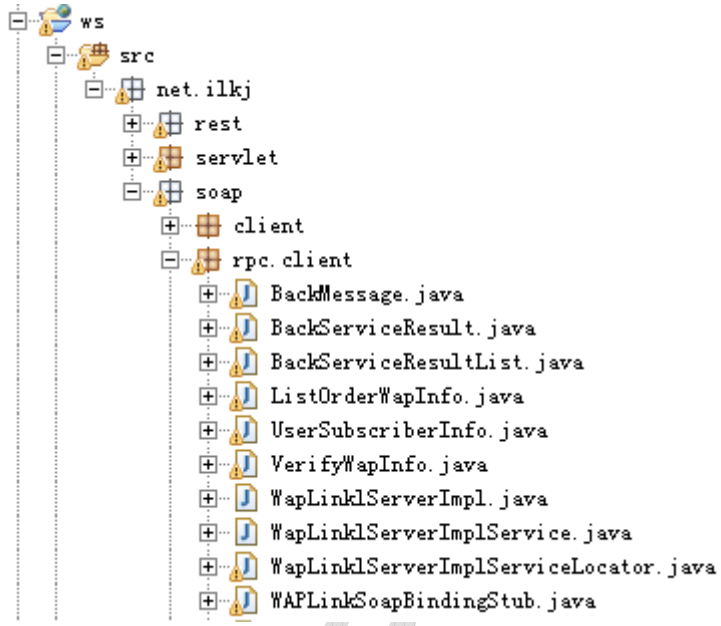
java org.apache.axis.wsdl.WSDL2Java –p 包路径 wsdl 的 URL

在得到生成的客户端代码之后，你可以在 Spring 的配置文件配置如何访问这个 JAX-RPC 的 Web 服务。

I.WSDL 样例（部分）：

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://10.0.1.72:8080/ibmp/service/WAPLink" ... …>
   <wsdl:types>
      ... ...
   <wsdl:service name="WapLinklServerImplService">
      <wsdl:port binding="impl:WAPLinkSoapBinding" name="WAPLink">
         <wsdlsoap:address location="http://10.0.1.72:8080/ibmp/service/WAPLink"/>
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

II.生成的客户端代码：



你会发现 JAX-RPC 生成的客户端没有任何注解，这是因为 JAX-RPC 是 J2EE1.4 中的 Web 规范，依赖于 JDK1.4，JAVA 从 JDK1.5 才开始支持注解，因此发布一个 JAX-RPC 的 Web 服务业显然没有它的后继版本 JAX-WS 使用注解来得简洁。

III.Spring 的配置文件：

```
<bean id="rpcClient"
    class="org.springframework.remoting.jaxrpc.JaxRpcPortProxyFactory
Bean">
    <property name="serviceInterface"
           value="net.ilkj.soap.rpc.client.WapLinklServerImpl"/>
    <property name="wsdlDocumentUrl"
          value="http://10.0.1.72:8080/ibmp/service/WAPLink?wsdl"/>
    <property name="namespaceUri"
          value="http://10.0.1.72:8080/ibmp/service/WAPLink"/>
    <property name="serviceName" value="WapLinklServerImplService"/>
    <property name="portName" value="WAPLink"/>
```

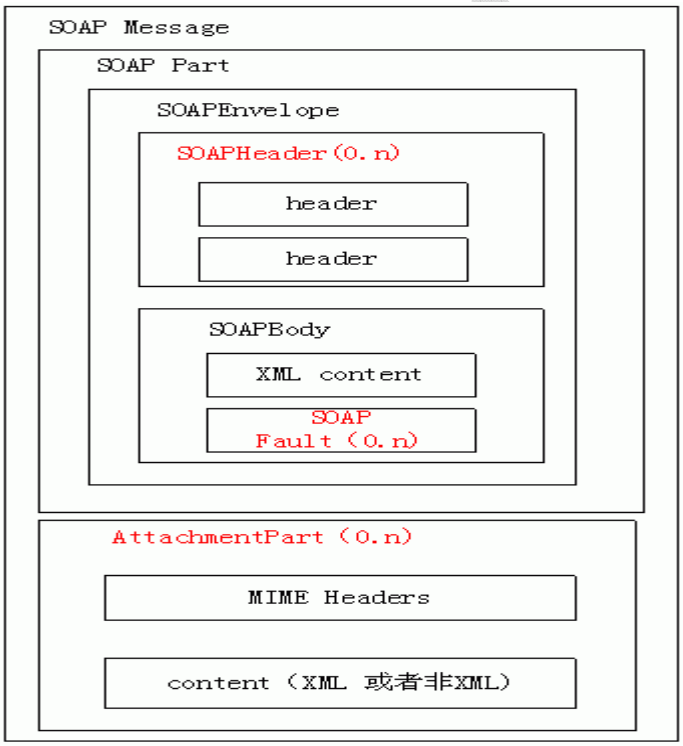
```
</bean>
```

这里使用 Spring 的 JaxRpcPortProxyFactoryBean 来访问 JAX-RPC 的 Web 服务，serviceInterface 指定客户端生成的接口，wsdlDocumentUrl 指定 WSDL 的地址，namespaceUri 指定 WSDL 中的 targetNamespace 属性值，serviceName 指定 WSDL 中的<wsdl:service … 的 name 属性值，portName 指定 WSDL 中的<wsdl:service … 的子元素<wsdl:port … 的 name 属性值。然后我们可以像使用普通的 Bean 一样在使用时将它强制转换为 serviceInterface 指定的接口类型，就可以调用 Web 服务的方法了。从这里你也可以看出 Spring 的强大易用，Spring 提供的对 RMI、JAX-RPC、JAX-WS、HTTPINVOKER、JMS 的整合调用都极为简洁。

9.JAXM 与 SAAJ：

前面我们就说过，JAXM&SAAJ、JAX-WS 都是基于 SOAP 消息的 Web 服务规范，并且前者暴漏了太多的细节，编码非常的繁琐，那么干嘛还要用它呢啊？你可以设想这样一种情况，你访问的 Web 服务传回来的 SOAP 消息中的 XML 可能无法正确解析成你的客户端对象，或者你要对 SOAP 消息中的 XML 做一些处理，那么你该怎么办呢？因为 JAX-WS 暴漏的细节极少，几乎都是自动完成的，你根本无法实现这个逻辑（或许 CXF 的拦截器可能会有提供这种打断自动处理机制，允许你在 XML 解析成 JAVA 对象之前半路插入，自己解析 XML，但这也只是 CXF 的功能，在 JAVA 面向接口的规则下，不能保证其他的 JAX-WS 实现也提供这种入口）。这个时侯 SAAJ 就派上用场了，其实 SAAJ 提供的 API 就是用于组装和解构 SOAP 消息的。

我们回忆前面介绍过 SOAP 消息的构成，可以用下图表示：



这些部件在 SAAJ 中都有对应的接口，对于客户端来说，SAAJ 的调用过程通常如下所示：

1.创建 SOAP 连接

2.创建 SOAP 消息

3.在 SOAP 消息里增加数据

4.发送消息

5.对 SOAP 应答进行处理

I.SAAJ 组装和结构 SOAP 消息：

现在我们访问前面的那个传输附件的 CXF 公开的 Web 服务接口：

(1.)服务器端：

…

```
public Customer selectMaxAgeStudent(Customer c1, Customer c2)
            throws HelloServiceException {
        try {
            System.out.println("*********************************"
                    + c1.getData().getContentType());
            // 输出接收到的附件
            InputStream ins = c1.getData().getInputStream();
            byte[] b = new byte[ins.available()];
            OutputStream ous = new FileOutputStream("c:/temp.jpg");
            while (ins.read(b) != -1) {
                ous.write(b);
            }
            ous.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        // 传递附件到客户端
        c1.setData(new DataHandler(new FileDataSource("c:/18.jpg")));
        c2.setData(new DataHandler(new FileDataSource("c:/18.jpg")));
        if (c1.getBirthday().getTime() > c2.getBirthday().getTime())
            return c2;
        else
            return c1;
}
```

…

(2.)客户端：

```
import javax.xml.soap.*;
…
// 获取SOAP连接工厂
SOAPConnectionFactory factory = SOAPConnectionFactory.newInstance();
// 从SOAP连接工厂创建SOAP连接对象
SOAPConnection connection = factory.createConnection();
// 获取消息工厂
MessageFactory mFactory = MessageFactory.newInstance();
// 从消息工厂创建SOAP消息对象
SOAPMessage message = mFactory.createMessage();
// 创建SOAPPart对象
```

```
SOAPPart part = message.getSOAPPart();
// 创建SOAP信封对象
SOAPEnvelope envelope = part.getEnvelope();
// 创建SOAPBody对象
SOAPBody body = envelope.getBody();
// 创建XML的根元素
SOAPBodyElement bodyElementRoot = body.addBodyElement(new QName(
              "http://server.soap.ilkj.net/", "selectMaxAgeStudent",
"ns2"));
// 创建Customer实例1
SOAPElement elementC1 = bodyElementRoot
              .addChildElement(new QName("c1"));
elementC1.addChildElement(new QName("birthday")).addTextNode(
              "1989-01-28T00:00:00.000+08:00");
elementC1.addChildElement(new QName("id")).addTextNode("1");
elementC1.addChildElement(new QName("name")).addTextNode("A");
// 创建附件对象
AttachmentPart attachment = message
              .createAttachmentPart(new DataHandler(new
FileDataSource(
                     "c:/18.jpg")));
// 设置Content-ID
attachment.setContentId("<" + UUID.randomUUID().toString() + ">");
attachment.setMimeHeader("Content-Transfer-Encoding", "binary");
message.addAttachmentPart(attachment);
SOAPElement elementData = elementC1.addChildElement(new QName("data"));
// 添加XOP支持
elementData.addChildElement(
       new QName("http://www.w3.org/2004/08/xop/include", "Include",
              "xop")).addAttribute(new QName("href"),"cid:"
                     + attachment.getContentId().replaceAll("<", "")
                            .replaceAll(">", ""));

// 创建Customer实例2
SOAPElement elementC2 = bodyElementRoot
              .addChildElement(new QName("c2"));
elementC2.addChildElement(new QName("birthday")).addTextNode(
              "1990-01-28T00:00:00.000+08:00");
elementC2.addChildElement(new QName("id")).addTextNode("2");
elementC2.addChildElement(new QName("name")).addTextNode("B");
AttachmentPart attachment2 = message
              .createAttachmentPart(new DataHandler(new
FileDataSource(
                     "c:/18.jpg")));
```

```
attachment2.setContentId("<" + UUID.randomUUID().toString() + ">");
message.addAttachmentPart(attachment2);
SOAPElement elementData2 = elementC2.addChildElement(new
QName("data"));
elementData2.addChildElement(
        new QName("http://www.w3.org/2004/08/xop/include", "Include",
                "xop")).addAttribute(new QName("href"),"cid:"
                    + attachment2.getContentId().replaceAll("<", "")
                            .replaceAll(">", ""));

// 访问Web服务地址
SOAPMessage reMessage = connection.call(message, new URL(
              "http://127.0.0.1:8080/ws/services/helloService"));
// 控制台输出返回的SOAP消息
reMessage.writeTo(System.out);

// 输出SOAP消息中的第一个子元素的元素名称
System.out.println("<<<<<<<<<<<<<<<<<<"
          + reMessage.getSOAPBody().getFirstChild().getLocalName());
// 输出SOAP消息中的附件
Iterator<AttachmentPart> it = reMessage.getAttachments();
while (it.hasNext()) {
    InputStream ins = it.next().getDataHandler().getInputStream();
    byte[] b = new byte[ins.available()];
    OutputStream ous = new FileOutputStream("c:/temp2.jpg");
    while (ins.read(b) != -1) {
        ous.write(b);
    }
    ous.close();
}
…
```

从上面的代码可以看出，SAAJ 完全暴漏了 SOAP 消息的处理细节，使得你可以更加自如的控制从服务端获取来的 SOAP 消息，而不必像 JAX-WS 一样完全自动处理。但是参照 WSDL 的内容，使用 SAAJ 组装对应的 SOAP 消息，确实难度比较大，尤其是对方公开的 Web 服务比较复杂的时候，你需要熟练的掌握 WSDL 的各部分内容。一个较为偷懒的办法是首先使用 JAX-WS 访问服务端，然后把 CXF 拦截器捕获的向服务端发送的 SOAP 消息复制出来，照样组装。

另外，要说明的是 SAAJ 只支持传输二进制形式的附件，不能解析传输的 BASE64 码的附件，因此，服务端的 JAX-WS 需要启用 MTOM，这样传输回来的附件 SAAJ 才会自动解析，否则你要自己将传回的 BASE 码转换成二进制文件。

SAAJ 的 API 位于 JAVA SE6 的文档中，大部分方法从名字上就可以看出是干什么用的，另外，JDK1.6 的 API 也是中文的，这里不去详细说明 SAAJ 的各个 API 的用法。

II.使用 JAXM 发布 Web 服务：

JAXM 的 API 实际上就是用来将一个 Servlet 发布成 WebService 的地址，它要求你的 Servlet 继承 javax.xml.messaging.JAXMServlet 并实现 javax.xml.messaging.ReqRespListener 接口（如果 Web 服务是单向的，也就是没有返回值给客户端，那么可以实现 javax.xml.messaging.OnewayListener 接口），其实这一个抽象类和两个接口是我们可以使用的三个 JAXM 的 API，其余的基本上都是 JAXM 底层实现需要使用的 API，我们并不需要关心。

例：

```
public class MyJAXMServlet extends JAXMServlet implements
                                                    ReqRespListener {

    /**
     *
     */
    private static final long serialVersionUID = 1870615968744239558L;

    @Override
    public SOAPMessage onMessage(SOAPMessage soapMessage) {

        // 处理参数soapMessage，这是客户端传送过来的SOAP消息

        // 返回SoapMessage，这是响应给客户端的SOAP消息
        return null;
    }

}
```

在实际应用中，你可以把你的业务层放在 onMessage()方法中调用，相当于把业务层的方法用这个 Servlet 包装成了 Web 服务，这个 Servlet 不可以使用普通的方式访问，只能使用 SoapConnection 的 call()方法调用。

实际上你可以看出来，JAXM 发布的 Web 服务比较简单，完全省略了 WSDL，这也就是说，你用这种方式发布 Web 服务，必须把要接收的 Soap 消息的内容说明发布出来（有点儿类似于 REST 风格的 OpenAPI），这样客户端才知道如何组装你想要的 SOAP 消息。从这里你也可以看出来，HTTP 协议与 SOAP 消息是基于 SOAP 的基本组成，WSDL 是完全可以没有的，WSDL 的作用是异构平台为了方便使用自己的语言特性的中间桥梁。