**Apache CXF**

**技术预研**

**版本：v0.3 (2014-02-28)**

**版本修订**

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 日期 | 描述 | 修订者 |
| 0.1 | 2013-08-01 | 创建编写主要内容。 | 张欢 |
| 0.2 | 2013-08-05 | 编写API附录，调整文档内容结构。 | 张欢 |
| 0.3 | 2013-02-28 | 整理部分REST与CXF应用部分内容。 | 张欢 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**目录**

[1 Web Sevices 1](#_Toc363486184)

[1.1 SOAP 1](#_Toc363486185)

[1.2 WSDL 1](#_Toc363486186)

[1.3 UDDI 1](#_Toc363486187)

[1.4 JAX-WS 2](#_Toc363486188)

[1.5 JAX-RS 2](#_Toc363486189)

[1.5.1 简介 2](#_Toc363486190)

[1.5.2 实现框架 2](#_Toc363486191)

[1.5.3 相关注解 3](#_Toc363486192)

[2 Rest 3](#_Toc363486193)

[2.1 定位 3](#_Toc363486194)

[2.2 定义 3](#_Toc363486195)

[2.3 原则 3](#_Toc363486196)

[3 Apache CXF初步 4](#_Toc363486197)

[3.1 简介 4](#_Toc363486198)

[3.2 功能特性 4](#_Toc363486199)

[3.3 支持REST 5](#_Toc363486200)

[4 Apache CXF应用 5](#_Toc363486201)

[4.1 环境搭建 5](#_Toc363486202)

[4.2 使用CXF开发RESTful服务 6](#_Toc363486203)

[4.2.1 建立项目 6](#_Toc363486204)

[4.2.2 定义POJO原型 6](#_Toc363486205)

[4.2.3 配置服务资源 8](#_Toc363486206)

[4.2.4 编写服务实现类 9](#_Toc363486207)

[4.2.5 配置文件 9](#_Toc363486208)

[4.2.6 定制客户端 11](#_Toc363486209)

[4.3 验证服务资源访问 11](#_Toc363486210)

[4.3.1 通过客户端 11](#_Toc363486211)

[4.3.2 通过浏览器 12](#_Toc363486212)

[4.4 自定义JSON格式 13](#_Toc363486213)

[4.4.1 通过XML配置 13](#_Toc363486214)

[4.4.2 通过客户端 13](#_Toc363486215)

[5 附录：JAXB API 14](#_Toc363486216)

[5.1 简介 14](#_Toc363486217)

[5.2 重要注解 14](#_Toc363486218)

[5.2.1 @XmlAccessorType 14](#_Toc363486219)

[5.2.2 @XmlAccessorOrder 14](#_Toc363486220)

[5.2.3 @XmlJavaTypeAdapter 15](#_Toc363486221)

[5.2.4 @XmlElementWrapper 17](#_Toc363486222)

[5.2.5 @XmlRootElement 17](#_Toc363486223)

[5.2.6 @XmlElement 17](#_Toc363486224)

[5.2.7 @XmlTransient 18](#_Toc363486225)

[6 附录：JAX-RS API 18](#_Toc363486226)

[6.1 简介 18](#_Toc363486227)

[6.2 重要注解 18](#_Toc363486228)

[6.2.1 @Path 18](#_Toc363486229)

[6.2.2 访问方式注解 19](#_Toc363486230)

[6.2.3 @Produces 19](#_Toc363486231)

[6.2.4 @Consumes 19](#_Toc363486232)

[6.2.5 @PathParam 20](#_Toc363486233)

[6.2.6 @QueryParam 20](#_Toc363486234)

# Web Sevices

Web Service能使得运行在不同机器上的不同应用无需借助附加的、专门的第三方软件或硬件， 就可相互交换数据或集成。依据Web Service规范实施的应用之间， 无论它们所使用的语言、 平台或内部协议是什么， 都可以相互交换数据。Web Service是自描述、 自包含的可用网络模块， 可以执行具体的业务功能。Web Service也很容易部署， 因为它们基于一些常规的产业标准以及已有的一些技术，诸如XML和HTTP。Web Service减少了应用接口的花费，为整个企业甚至多个组织之间的业务流程的集成提供了一个通用机制。

JWS：由于Web Service是并不限制语言的，所以在Java语言中，有一个专有名词JWS (Java Web Service)，它是专指在Java EE中与Web Service相关的技术。

## SOAP

SOAP即简单对象访问协议(Simple Object Access Protocol)，它是用于交换XML编码信息的轻量级协议。它有三个主要方面：XML-envelope为描述信息内容和如何处理内容定义了框架，将程序对象编码成为XML对象的规则，执行远程过程调用(RPC)的约定。

SOAP可以运行在任何其他传输协议上。例如，你可以使用 SMTP，即因特网电子邮件协议来传递SOAP消息，这可是很有诱惑力的。在传输层之间的头是不同的，但XML有效负载保持相同。

Web Service 希望实现不同的系统之间能够用“软件-软件对话”的方式相互调用，打破了软件应用、网站和各种设备之间的格格不入的状态，实现“基于Web无缝集成”的目标。

## WSDL

WSDL是指Web Service描述语言。就是用机器能阅读的方式提供的一个正式描述文档而基于XML的语言，用于描述Web Service及其函数、参数和返回值。因为是基于XML的，所以WSDL既是机器可阅读的，又是人可阅读的。

## UDDI

UDDI 的目的是为电子商务建立标准；UDDI是一套基于Web的、分布式的、为Web Service提供的、信息注册中心的实现标准规范，同时也包含一组使企业能将自身提供的Web Service注册，以使别的企业能够发现的访问协议的实现标准。

## JAX-WS

JAX-WS(Java API for XML-Based Web Services)规范是一组XML Web Services的JAVA API，它允许开发者可以选择RPC-oriented或者message-oriented 来实现自己的Web Services。

在JAX-WS中，一个远程调用可以转换为一个基于XML的协议例如SOAP，在使用JAX-WS过程中，开发者不需要编写任何生成和处理SOAP消息的代码。JAX-WS会将这些API的调用转换成为对应的SOAP消息。

在服务器端，用户只需要通过Java语言定义远程调用所需要实现的接口SEI(Service Endpoint Interface)，并提供相关的实现，通过调用JAX-WS的服务发布接口就可以将其发布为WebService接口。在客户端，用户可以通过JAX-WS的API创建一个代理（用本地对象来替代远程的服务）来实现对于远程服务器端的调用。

## JAX-RS

### 简介

JAX-RS是Java EE 6引入的一个新技术。JAX-RS即Java API for RESTful Web Services，是一个Java 编程语言的应用程序接口，支持按照表述性状态转移（REST）架构风格创建Web服务。JAX-RS使用了Java SE 5引入的Java注解(Annotation)来简化Web服务的客户端和服务端的开发和部署。

### 实现框架

目前JAX-RS的核心功能类被放在javax.ws.rs.core包中，里面包含了很多底层的功能实现类和注解等。而javax.ws.rs.ext包中提供了一些API，为了对JAX-RS支持的类型（如XML和JSON）进行扩展支持。

目前实现JAX-RS的框架有：

|  |  |
| --- | --- |
| 框架 | 说明 |
| Apache CXF | 一个开源的 Services 框架，CXF 帮助您利用 Frontend 编程 API 来构建和开发 Services ，像 JAX-WS、JAX-RS。 |
| Jersey | Oracle公司推出的restful框架。可以集成至tomcat、jetty使用，支持Servlet和Spring的整合，支持jsr311的1.1和1.0版本。 |
| RESTEasy | RESTEasy是JBoss的一个开源项目，提供各种框架帮助你构建RESTful Web Services和RESTful Java应用程序。它是JAX-RS规范的一个完整实现并通过JCP认证。 |
| Restlet | 由Jerome Louvel和Dave Pawson开发，是最早的REST框架，先于JAX-RS出现。 |
| Apache Wink | 一个Apache软件基金会孵化器中的项目，其服务模块实现JAX-RS规范。 |

### 相关注解

JAX-RS在Java EE 6中的功能都存放在**javax.ws.rs**包中，其中提供了一些注解将一个服务接口、一个POJO类封装为Web资源。这些注解包括：

|  |  |
| --- | --- |
| 注解 | 说明 |
| @Path | 标注访问资源类或者方法的相对路径。 |
| @GET, @PUT, @POST, @DELETE | 标注访问方法的请求是HTTP中哪种类型。 |
| @Produces | 标注返回的MIME媒体类型，如XML或JSON。 |
| @Consumes | 标注可接受请求的MIME媒体类型。 |
| @PathParam, @QueryParam, @HeaderParam, @CookieParam, @MatrixParam, @FormParam | 分别标注方法的参数来自于HTTP请求的不同位置。 |

实际的注解使用代码，请参考后文第4章的简单示例，本文附录中也将提供API说明。

# Rest

## 定位

在各个系统交互领域，Web Services逐渐成为主流。有两种主要方式来开发Web Services：Simple Object Access Protocol (SOAP)和Representational State Transfer (REST)。

开发基于SOAP的Web Services需要很多的约束，以便客户端和服务端交互信息。例如，使用Web Service Description Language (WSDL)来描述信息。还有很多WS的标准如WS-Security。

而使用REST构架的服务被称为RESTful服务。这种架构利用简单的XML在HTTP协议上，就像网页一样发送请求，简化了基于SOAP架构开发。RESTful Web Services尤其适用于只需要提交和接受简单的XML信息。

## 定义

**REST** (REpresentational State Transfer)是指一组架构约束条件和原则，满足这些约束条件和原则的应用程序或设计就是**RESTful**。

## 原则

最重要的 REST 原则是，客户端和服务器之间的交互在请求之间是无状态的。从客户端到服务器的每个请求都必须包含理解请求所必需的信息。如果服务器在请求之间的任何时间点重启，客户端不会得到通知。此外，无状态请求可以由任何可用服务器回答，这十分适合云计算之类的环境。客户端可以缓存数据以改进性能。

另一个重要的 REST 原则是分层系统，这表示组件无法了解它与之交互的中间层以外的组件。通过将系统知识限制在单个层，可以限制整个系统的复杂性，促进了底层的独立性。

当 REST 架构的约束条件作为一个整体应用时，将生成一个可以扩展到大量客户端的应用程序。它还降低了客户端和服务器之间的交互延迟。统一界面简化了整个系统架构，改进了子系统之间交互的可见性。REST 简化了客户端和服务器的实现。

REST不是一种规范标准，而是一种SOA架构风格。它不是面向动作方法的，而是面向资源的。

# Apache CXF初步

## 简介

Apache CXF中的CXF是Celtix + XFire的缩写，其前身是Apache CeltiXfire，以下简称为CXF。官方网站地址为：<http://cxf.apache.org/>。

CXF作为Apache的一个项目，为Java EE 6中新加入的JAX-RS规范提供了框架实现。

## 功能特性

CXF支持的特性非常广泛，它在Web服务标准方面提供的支持有：

1. SOAP

2. WS-Addressing

3. WS-Policy

4. WS-ReliableMessaging

5. WS-Security

6. WS-SecurityPolicy

7. WS-SecureConversation

8. JAX-WS API，用于Web服务开发

9. WSDL优先工具

10. Java优先支持

11. JAX-RS (JSR 311 1.0) API，用于RESTfulWeb服务开发

12. JavaScript编程模型，用于客户端和服务端开发

13. Maven工具

14. 支持CORBA

15. HTTP和JMS传输层

CXF中可嵌入的开发有：

1. ServiceMix或其他JBI容器

2. Geronimo或其他Java EE容器

3. Tomcat或其他Servlet容器

4. OSGi

5. 引用OSGi远程服务实现

## 支持REST

CXF实现了JAX-RS1.0规范，并提供了很多特性帮助开发者搭建企业级的RESTful服务。下面是CXF框架提供创建RESTful服务的各种特性：

1. 集成Spring

Spring框架已经变成事实上的构建企业级JAVA应用程序集成框架。CXF提供与Spring整合，简化了配置和部署RESTful应用程序。Spring提供依赖注入促进松散耦合，提供各种服务，像声明式事务管理。所有这些Spring提供的特性都可以被开发RESTful服务的CXF框架使用。

2. 插入式数据绑定

数据库绑定就是映射HTTP请求，例如JSON或XML，到对应的JAVA对象。同样的，在发送HTTP响应之前，服务端的JAVA实现需要映射为客户端所需要的格式。通过提供数据库绑定组件，CXF在后台透明的处理映射。CXF支持各种数据绑定机制，如JAXB，JSON，XMLBean和Aegis。CXF允许指定特定的绑定机制。

3. 客户端API

JAX-RS规范并没有提供客户端调用REST服务的API。CXF提供了这种API直接调用RESTful服务，也可以使用Spring框架配置到应用程序中。

4. 安全

CXF可以使用Spring框架集成的声明式安全组件，按照应用程序的需要限制资源类和方法，而不必使用代码处理安全性问题。

5. 过滤器

过滤器用来预处理或后处理信息。CXF可以创建和配置过滤器来审核信息，记录信息的日志，还有基于应用要求修改请求或响应。

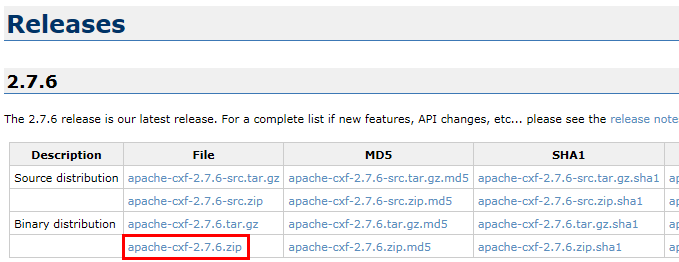
CXF也允许开发者使用JAX-WS Provider和Dispatch API来创建RESTful服务。

# Apache CXF应用

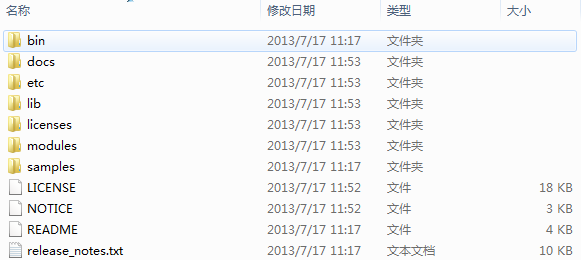
## 环境搭建

本文使用的Eclipse for Java EE版本为**3.7.2** (Indigo)；Apache CXF项目版本为**2.7.6**；Apache Tomcat版本为**7.0**。

访问<http://cxf.apache.org/download.html>下载CXF项目的资源文件，如下图所示：



下载之后解压缩ZIP文件，得到的文件内容应如下所示：



如果需要使用CXF，则只需将上图中lib目录下的所有内容复制到项目的lib目录中即可。

## 使用CXF开发RESTful服务

### 建立项目

在Eclipse中，新建一个Dynamic Web Project，项目名称为**cxf-rest**，其他配置保持默认即可。

将刚刚下载好的CXF项目中lib目录下所有资源内容复制到cxf-rest/WebContent/WEB-INF/lib目录中。Eclipse的Web App Libraries将会自动引入这些jar包。

### 定义POJO原型

在src目录下建立**com.demo.cxf.rest.entity**包，用于存放POJO类。

1. **Student**

首先在该包下建立一个**Student**类。其代码如下：

|  |
| --- |
| package com.demo.cxf.rest.entity;  import java.util.Date;  import javax.xml.bind.annotation.XmlAccessType;  import javax.xml.bind.annotation.XmlAccessorType;  import javax.xml.bind.annotation.XmlRootElement;  **@XmlRootElement(name = "Student")**  **@XmlAccessorType(XmlAccessType.FIELD)**  public class Student {  **@XmlElement(name = "StudentId")**  private long id;  **@XmlElement(name = "StudentName")**  private String name;  private Date birthday;  // setter getter  } |

这是一个Student原型类，含有id、name、birthday等基本属性。由于CXF内部使用JAXB来处理POJO和XML的映射关系，因而使用javax.xml.bind.annotation包下的注解来配置这个POJO类。

**@XmlRootElement**指定对应的XML文件中，根节点的名称为Student；**@XmlElement**用来定义XML中的子节点名称，如果没有指定，则JAXB默认使用与类属性同名的节点名称（如上述代码中的birthday属性）；**@XmlAccessorType (XmlAccessType.FIELD)**表示使用这个类中的 private 非静态字段作为 XML 的序列化的属性或者元素。

在本示例中，使用的XML相关注解，其具体原理是调用Java SE中的**JAXB**。相关的API请参考附录。

1. **UserResponse**

然后再建立一个**UserResponse**类。这个类内部包含一个Student类属性，代码如下：

|  |
| --- |
| package com.demo.cxf.rest.entity;  import javax.xml.bind.annotation.XmlAccessType;  import javax.xml.bind.annotation.XmlAccessorType;  import javax.xml.bind.annotation.XmlRootElement;  @XmlRootElement(name = "Response")  @XmlAccessorType(XmlAccessType.FIELD)  public class UserResponse {    private String status;  **private Student data;**    public UserResponse() {}  public UserResponse(String status, Student data) {  this.status = status;  this.data = data;  }  // setter getter  } |

**注意**一定要保证这样的POJO类中含有默认的无参构造方法。

### 配置服务资源

创建**com.demo.cxf.rest.service**包并定义服务接口**StudentService**，代码如下：

|  |
| --- |
| package com.demo.cxf.rest.service;  import javax.ws.rs.GET;  import javax.ws.rs.Path;  import javax.ws.rs.PathParam;  import javax.ws.rs.QueryParam;  import com.demo.cxf.rest.entity.Student;  import com.demo.cxf.rest.entity.UserResponse;  **@Path("ss")**  public interface StudentService {    **@GET**  **@Path(value = "/info")**  Student getStudent(  **@PathParam("id")**  long id,  **@PathParam("name")**  String name);    **@GET**  **@Path(value = "/info2/{name}")**  UserResponse getStudent(  **@QueryParam("name")**  String name);  } |

注意其中使用了大量位于**javax.ws.rs**包下的注解。

接口StudentService被当做是一种服务资源，可以通过**@Path**注解来配置访问该服务资源的URI地址。例如上述代码中**@Path("ss")**的含义是指：将可以通过访问<http://localhost:8080/ss/>路径获取StudentService服务资源。

该接口有两个重载方法，分别使用了**@GET**注解声明：该方法支持的HTTP访问方式为GET。同时又使用了**@Path**注解，并指定了value的值。这是配置访问该方法的URI路径，注意该路径默认是接口定义上方@Path配置路径的**子路径**。

例如，StudentService接口已经使用@Path("ss")注解声明路径为<http://localhost:8080/ss/>，而接口内部方法getStudent()又使用了**@Path(value = "/info")**配置，则该方法的URI路径为<http://localhost:8080/ss/info/>。

如果使用@Path注解指定路径时并不带有参数，如**@Path(value = "/info")**，却希望在方法中使用地址栏参数，则需要使用形如**@PathParam("id")**的注解对方法参数进行配置。如上述代码中2个参数的方法配置，此时访问<http://localhost:8080/ss/info?id=32&name=celine>地址，将可以让参数id获得数字32，让参数name获得字符串celine。**使用@PathParam注解时，参数是以键值对的形式存在的。**

而如果方法的@Path注解路径中含有参数，如**@Path(value = "/info2/{name}")**，那么该方法可以使用**@QueryParam("name")**注解将URI地址中name的部分映射到方法参数String name上。例如，访问<http://localhost:8080/ss/info2/celine>地址，那么方法的参数String name将会获得字符串celine的值。**使用@QueryParam注解时，参数是以URI地址的形式存在的。**

### 编写服务实现类

StudentService接口的实现类并不需要进行Annotation配置，实际上这种配置推荐（也是应当）只配置在接口上而不推荐（也不应该）配置在实现类上。

**com.demo.cxf.rest.service.imp.StudentServiceImp**类的代码如下：

|  |
| --- |
| package com.demo.cxf.rest.service.imp;  import java.text.SimpleDateFormat;  import com.demo.cxf.rest.entity.Student;  import com.demo.cxf.rest.entity.UserResponse;  import com.demo.cxf.rest.service.StudentService;  public class StudentServiceImp implements StudentService {  @Override  public Student getStudent(long id, String name) {  Student stu = new Student();  stu.setId(id);  stu.setName(name);  try {  stu.setBirthday(new SimpleDateFormat("yyyy-MM-dd")  .parse("1968-03-30"));  } catch (java.text.ParseException e) {  e.printStackTrace();  }  return stu;  }  @Override  public UserResponse getStudent(String name) {  Student stu = new Student();  stu.setId(1);  stu.setName(name);  try {  stu.setBirthday(new SimpleDateFormat("yyyy-MM-dd")  .parse("1968-03-30"));  } catch (java.text.ParseException e) {  e.printStackTrace();  }  return new UserResponse("ok", stu);  }  } |

其中只是模拟实现了一些最为简单的逻辑：使用HTTP的GET方式获取一个POJO（Student或UserResponse对象）的内容信息。

### 配置文件

配置文件需要有2个：一个是常规的web.xml，另一个是CXF的beans.xml。

1. **web.xml文件**

在WebContent/WEB-INF路径下建立web.xml文件，内容如下：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlnx="http://java.sun.com/xml/ns/javaee"  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"  id="WebApp\_ID"  version="3.0">    <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>/\*</url-pattern>  </servlet-mapping>  </web-app> |

可以看到其中首先指定了上下文参数contentConfigLocation，配置beans.xml文件的位置，这在Spring中是常见用法。由于CXF已经整合了Spring框架的部分功能，因而配置了Spring的上下文监听器。我们使用CXF项目自身的Servlet，让它映射到当前项目中的所有资源上（当然也可以自由配置路径筛选），并设置这个Servlet为自动启动的。

1. **beans.xml文件**

beans.xml文件类似于Spring的applicationContext.xml文件的作用，内容如下：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns: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/jaxrs  http://cxf.apache.org/schemas/jaxrs.xsd">  <import resource="classpath:META-INF/cxf/cxf.xml"/>  <import resource="classpath:META-INF/cxf/cxf-servlet.xml"/>    <bean id="rsStudentServiceImp" class="com.demo.cxf.rest.service.imp.StudentServiceImp"/>    <jaxrs:server id="test" address="/">  <jaxrs:serviceBeans>  <ref bean="rsStudentServiceImp"/>  </jaxrs:serviceBeans>  <jaxrs:extensionMappings>  <entry key="json" value="application/json"/>  <entry key="xml" value="application/xml"/>  </jaxrs:extensionMappings>  </jaxrs:server>  </beans> |

首先引入两个CXF项目的XML文件资源。**注意！**在当前的CXF版本（2.7.6）中，cxf-extension-jaxrs-binding.xml资源已经不再需要了，jar包中也不再提供，如果在此处引入该资源，将会引发错误。

下面需要定义一个bean，指定StudentService服务资源的实现类。

最后使用**<jaxrs:server>**标签进行服务资源的配置。**<jaxrs:serviceBeans>**标签指定服务资源的具体功能实现类（Bean），而**<jaxrs:extensionMappings>**标签指定支持的数据格式，如上例中的JSON格式和XML格式。

### 定制客户端

此时的项目已经可以部署到Web容器中进行访问了，但我们仍然定制一个客户端来模拟一次HTTP GET的访问。**com.demo.cxf.rest.client.Client**类的代码如下：

|  |
| --- |
| package com.demo.cxf.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 Client {  public static void main(String[] args) throws ClientProtocolException,  IOException {    HttpGet get =  new HttpGet("http://localhost:8080/cxf-rest/ss/info?id=1&name=user");  HttpClient client = new DefaultHttpClient();  get.addHeader("ACCEPT", "application/xml");  HttpResponse response = client.execute(get);  InputStream ins = response.getEntity().getContent();  byte[] b = new byte[100000];  StringBuilder sb = new StringBuilder();  while (ins.read(b) != -1) {  sb.append(new String(b, "UTF-8"));  }  System.out.println(sb.toString());  }  } |

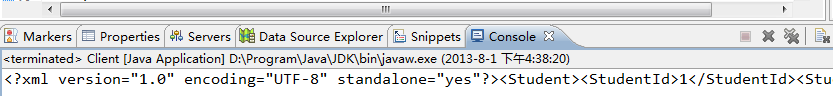
从代码中可以看到：

使用HTTP GET方式访问的地址为：<http://localhost:8080/cxf-rest/ss/info?id=1&name=user>，请求的资源类型为XML格式。

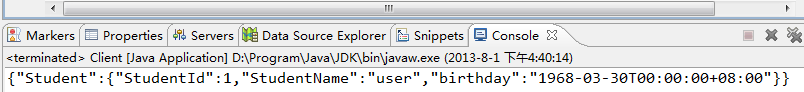
## 验证服务资源访问

### 通过客户端

将项目导出为war包并部署到Tomcat的webapps目录中，然后以Java Application方式运行刚刚写好的客户端类Client，将可以看到控制台中写出了Student对象的内容，如下图所示：

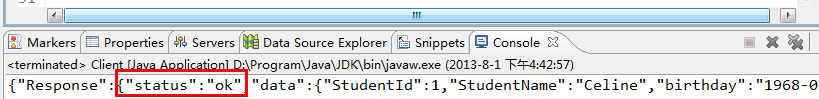


如果更改资源格式由application/**xml**变为application/**json**，则会输出JSON格式的内容，如下图所示：



注意其中的数据内容是Student类对象，而不是UserResponse类对象。因为我们访问的地址是[http://localhost:8080/cxf-rest/ss/info**?id=1&name=user**](http://localhost:8080/cxf-rest/ss/info?id=1&name=user)，该地址将会根据StudentService的配置路由到Student getStudent(long id, String name)方法上。

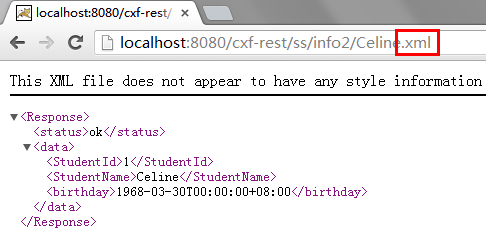
如果将访问地址修改为[http://localhost:8080/cxf-rest/ss/info2/**Celine**](http://localhost:8080/cxf-rest/ss/info2/Celine)，那么将会执行UserResponse getStudent(String name)方法上，控制台输出内容如下图所示：



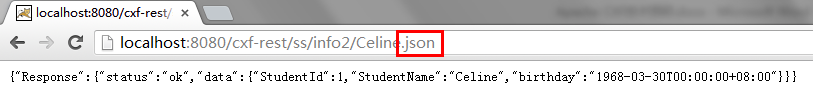
注意其中出现了只有UserResponse原型类才有的status属性。

### 通过浏览器

使用Chrome浏览器访问[http://localhost:8080/cxf-rest/ss/info2/Celine**.xml**](http://localhost:8080/cxf-rest/ss/info2/Celine.xml)地址，网页将显示返回的XML文件信息，如下图所示：



而如果将地址栏中的xml修改为json，地址为[http://localhost:8080/cxf-rest/ss/info2/Celine**.json**](http://localhost:8080/cxf-rest/ss/info2/Celine.json) ，网页内容将如下图所示，直接显示JSON内容（如果是IE浏览器，可能会提示下载JSON文件，保存后打开文件仍显示相同JSON内容）：



## 自定义JSON格式

CXF通过org.apache.cxf.jaxrs.provider.json.**JSONProvider**对JSON提供支持，默认的Provider采用Jettison进行编码或解码。无论是服务端还是客户端，CXF都可以通过配置JSONProvider来修改JSON的格式。

### 通过XML配置

例如在beans.xml文件中，可以定义这样一个bean：

|  |
| --- |
| <bean id="jsonProvider" class="**org.apache.cxf.jaxrs.provider.json.JSONProvider**">  <property name="**dropRootElement**" value="true"/>  <property name="**dropCollectionWrapperElement**" value="true"/>  <property name="**serializeAsArray**" value="true"/>  </bean> |

代码中涉及到3个属性。

**dropRootElement**属性默认值为false，这时JSON格式会将类名作为第一个节点的名称，例如：{Customer:{"id":123,"name":"John"}}；但是如果手动设置为false，那么JSON格式就会默认不显示第一个类名节点，成为：{"id":123,"name":"John"}。Google的Gson等解析方式为后者。

**dropCollectionWrapperElement**属性默认值为false，此时当遇到Collection时，JSON会在集合中将容器中类名作为一个节点，比如：{"Customer":{{"id":123,"name":"John"}}}；而设置为false，则JSON格式为{{"id":123,"name":"John"}}。

**serializeAsArray**属性默认为false，则当遇到Collection时，格式为**{**{"id":123,"name":"John"}**}**；如果设置为true，则格式为**[**{"id":123,"name":"john"}**]**，而Gson等解析为后者。

定义好这样的一个JSONProvider bean之后，使用时的配置如下所示：

|  |
| --- |
| <jaxrs:server id="customerService" address="/service1">  <jaxrs:serviceBeans>  <ref bean="customerBean" />  </jaxrs:serviceBeans>  **<jaxrs:providers>**  **<ref bean="jsonProvider"/>**  **</jaxrs:providers>**  </jaxrs:server> |

其中使用了<jaxrs:providers>标签引用了此前定义的jsonProvider的bean。

### 通过客户端

在CXF的Client端，也可以通过Java语句代码达到同样的效果：

|  |
| --- |
| JSONProvider myJsonProvider = new JSONProvider();  myJsonProvider.setDropRootElement(true);  myJsonProvider.setSupportUnwrapped(true);  myJsonProvider.setSerializeAsArray(true);  List<JSONProvider> providers = new ArrayList<JSONProvider>();  providers.add(myJsonProvider);  WebClient client =  WebClient.create("http://localhost:8080/testCXFServer", providers); |

# 附录：JAXB API

## 简介

JAXB (Java Architecture for XML Binding) 是一个业界的标准，是一项可以根据XML Schema产生Java类的技术。该过程中，JAXB也提供了将XML实例文档反向生成Java对象树的方法，并能将Java对象树的内容重新写到XML实例文档。从另一方面来讲，JAXB提供了快速而简便的方法将XML模式绑定到Java表示，从而使得Java开发者在Java应用程序中能方便地结合XML数据和处理函数。

本文中的Apache CXF在对POJO类和XML文件进行映射处理的时候，其核心正是使用了JAXB方式。

## 重要注解

### @XmlAccessorType

控制默认情况下是否对字段或 Javabean 属性进行系列化。基本用法如下：

|  |
| --- |
| **@XmlAccessorType(XmlAccessType.FIELD)**  public class Student {} |

该注解需要配合XmlAccessType一起使用，该枚举的不同取值代表不同含义，如下表所示：

|  |  |
| --- | --- |
| 取值 | 说明 |
| XmlAccessType.FIELD | JAXB 绑定类中的每个非静态、非瞬态字段将会自动绑定到 XML，除非由 XmlTransient 注释。 |
| XmlAccessType.NONE | 所有字段或属性都不能绑定到 XML，除非使用一些 JAXB 注释专门对它们进行注释。 |
| XmlAccessType.PROPERTY | JAXB 绑定类中的每个获取方法/设置方法对将会自动绑定到 XML，除非由 XmlTransient 注释。 |
| XmlAccessType. PUBLIC\_MEMBER | 每个公共获取方法/设置方法对和每个公共字段将会自动绑定到 XML，除非由 XmlTransient 注释。 |

### @XmlAccessorOrder

控制类中字段和属性的排序。基本用法如：

|  |
| --- |
| **@XmlAccessorOrder(XmlAccessOrder.UNDEFINED)**  public class Student {} |

该注解需要配合XmlAccessOrder一起使用，该枚举的不同取值代表不同含义，如下表所示：

|  |  |
| --- | --- |
| 取值 | 说明 |
| XmlAccessorOrder. ALPHABETICAL | 类中的字段和属性是按照方法 java.lang.String.compareTo(String anotherString) 确定的字母顺序进行排列的。 |
| XmlAccessorOrder. UNDEFINED | 类中字段和属性的排序是不确定的。 |

### @XmlJavaTypeAdapter

使用定制的适配器（即扩展抽象类XmlAdapter并覆盖marshal()和unmarshal()方法），以序列化Java类为XML。一些Java类型不能自然映射成XML，例如HashMap 或其他非 JavaBean 类，这个时候可以覆盖XmlAdapter来自定义转换方法。

#### 编写转换器

由于JAXB不能直接处理java.util.Map类型，所以我们要手动写一个转换器类：

|  |
| --- |
| package com.demo.cxf.rest.client;  import java.util.ArrayList;  import java.util.List;  import java.util.Map;  import javax.xml.bind.annotation.XmlAccessType;  import javax.xml.bind.annotation.XmlAccessorType;  import javax.xml.bind.annotation.XmlType;  @XmlType(name = "MapConvertor")  @XmlAccessorType(XmlAccessType.FIELD)  public class MapConvertor {  private List<MapEntry> entries = new ArrayList<MapEntry>();  public void addEntry(MapEntry entry) {  entries.add(entry);  }  public List<MapEntry> getEntries() {  return entries;  }  **public static class MapEntry {**  private String key;  private Object value;  public MapEntry() {  }  public MapEntry(Map.Entry<Object, Object> entry) {  this.key = entry.getKey().toString();  this.value = entry.getValue();  }  public MapEntry(Object key, Object value) {  this.key = key.toString();  this.value = value;  }  public String getKey() {  return key;  }  public void setKey(String key) {  this.key = key;  }  public Object getValue() {  return value;  }  public void setValue(Object value) {  this.value = value;  }  }  } |

这个转换器虽然看着代码多，实际上比较简单。只是内部自定义了一个MapEntry类来模仿Map存储key-value的功能，然后在外部类中提供一个List来存储这样的简单、纯数据（java.util.Map数据结构相对比较复杂）的若干MapEntry。

#### 定义适配器

首先认识一个抽象类：javax.xml.bind.annotation.adapters. **XmlAdapter**<ValueType,BoundType>。其中的两个泛型定义分别是：

ValueType：JAXB 无需其他操作便知道如何处理的类型。

BoundType：JAXB 不知道如何处理的一些类型。编写一个适配器，以便允许通过 ValueType 将此类型用作内存表示形式。

例如自定义的MapAdapter代码如下：

|  |
| --- |
| import java.util.HashMap;  import java.util.Map;  import javax.xml.bind.annotation.adapters.XmlAdapter;  public class MapAdapter extends XmlAdapter<MapConvertor, Map<Object, Object>> {  @Override  public MapConvertor marshal(Map<Object, Object> map) throws Exception {  MapConvertor convertor = new MapConvertor();  for (Map.Entry<Object, Object> entry : map.entrySet()) {  MapConvertor.MapEntry e = new MapConvertor.MapEntry(entry);  convertor.addEntry(e);  }  return convertor;  }  @Override  public Map<Object, Object> unmarshal(MapConvertor map) throws Exception {  Map<Object, Object> result = new HashMap<Object, Object>();  for (MapConvertor.MapEntry e : map.getEntries()) {  result.put(e.getKey(), e.getValue());  }  return result;  }  } |

其中继承了XmlAdapter抽象类，并且腹泻了**marshal()**和**unmarshal()**方法。前者将不被支持的Map类型转换为简单的MapConvertor类型；后者将简单的MapConvertor转换为原始的Map类型。

#### 注解配置

这样的一个Adapter如何被JAXB使用呢？直接在@XmlJavaTypeAdapter注解中指定value就可以了：

|  |
| --- |
| **@XmlJavaTypeAdapter(MapAdapter.class)**  Map<String, String> getMap(); |

通过这总共3个步骤，java.util.Map类型就可以被JAXB进行处理了。当然，如果用户希望进行自定义的特殊处理，可以将自己的业务逻辑写在Convertor中。

### @XmlElementWrapper

对于数组或集合（即包含多个元素的成员变量），生成一个包装该数组或集合的XML元素（称为包装器）。这个注解**只能用在数组或者集合**，不可以用于接口、包或类等。

例如在int[] names上面使用了@XmlElementWrapper注解，如下所示：

|  |
| --- |
| **@XmlElementWrapper(name=”allnames”)**  int[] names; |

那么本来names数组的XML展现如下：

|  |
| --- |
| <names> ...</names>  <names> ...</names> |

现在names数组的XML展现将变成：

|  |
| --- |
| **<allnames>**  <names> value-of-item </names>  <names> value-of-item </names>  **</allnames>** |

@ XmlElementWrapper注解有3个可选属性，说明如下所示：

|  |  |
| --- | --- |
| 属性 | 说明 |
| name | XML 包装器元素的名称。 |
| namespace | XML 包装器元素的 XML 目标名称空间。 |
| nillable | 如果为 true，则使用 xsi:nil='true' 表示集合不存在。 |

### @XmlRootElement

将类或枚举类型映射到 XML 元素。正如前文示例中所使用的情况，该注解用来声明根节点的名称，用法如下：

|  |
| --- |
| **@XmlRootElement(name = "Student")**  public class Student(){} |

如果指定了name取值，那么将按照用户自定义的名称设置根节点的名称，如果没有name取值，则JAXB会以类名称作为根节点的名称。

另外，还可以使用namespace属性来设置命名空间。

### @XmlElement

将Java类的一个属性映射到与属性同名的一个XML元素。用法如下：

|  |
| --- |
| **@XmlElement(name = "StudentId")**  private long id; |

该注解有6个可选属性，说明如下：

|  |  |
| --- | --- |
| 属性 | 说明 |
| defaultValue | 此元素的默认值。 |
| name | XML 模式元素的名称。 |
| namespace | XML 模式元素的 XML 目标名称空间。 |
| nillable | 将元素声明自定义为 nillable。 |
| required | 自定义所需的元素声明。 |
| type | 正被引用的 Java 类。 |

### @XmlTransient

有时候我们在POJO类中并不总是希望所有类属性都被映射到XML当中，那么如果我们不希望某一个属性被影射，则可以使用@XmlTransient注解。用法如下：

|  |
| --- |
| **@XmlTransient** private Date birthday; |

这样一来，在影射的XML文件中，将不存在名称为birthday的节点。该注解不含任何属性。

# 附录：JAX-RS API

## 简介

Java EE 6中加入了JAX-RS规范，其API主要分布在3个包中，它们分别是：

|  |  |
| --- | --- |
| 包路径 | 说明 |
| javax.ws.rs | 包含**高层次**的接口和注解，用于构建RESTful的服务。 |
| javax.ws.rs.core | 包含**低层次**的接口和注解，用于构建RESTful的服务。 |
| javax.ws.rs.ext | 为JAX-RS支持的类型提供扩展的API。 |

前文在1.6.3小节和4.3.2小节介绍了一些JAX-RS的常用注解及基本用法，它们主要集中在javax.ws.rs包中。这里将对JAX-RS的API进行进一步地整理归纳。

## 重要注解

### @Path

配置一个URI地址用于请求一个类或方法资源，配置的**路径是相对的**。

如果将@Path用于一个类，那么URI地址就是访问当前整体服务的地址。例如：

|  |
| --- |
| **@Path("ss")**  public interface StudentService {} |

将可以通过访问<http://localhost:8080/ss/>路径获取StudentService服务资源。

而如果将@Path用于一个方法，那么URI地址则是访问当前方法的**子路径**，这个子路径是相对于方法所在接口的“大路径”的。例如：

|  |
| --- |
| **@Path("ss")**  public interface StudentService {    **@GET**  **@Path(value = "/info")**  Student getStudent(); |

那么可以通过访问<http://localhost:8080/ss/info>来获得getStudent()方法资源。

### 访问方式注解

JAX-RS支持的HTTP访问方式类型有4种：@GET, @PUT, @POST和@DELETE。如果在一个方法上面使用**@GET**注解，如上例中代码所示，则表示getStudent()方法可以用来相应访问方式为GET的HTTP请求。

有的时候仅仅有@GET是不够的，在进行CRUD操作中：

* 若要在服务器上创建资源，应该使用 POST 方法，对应**@POST**注解；
* 若要检索某个资源，应该使用 GET方法，对应**@GET**注解；
* 若要更改资源状态或对其进行更新，应该使用 PUT方法，对应**@PUT**注解；
* 若要删除某个资源，应该使用 DELETE 方法，对应**@DELETE**注解。

### @Produces

用来配置某一个方法的返回媒体类型（**输出类型**），例如JSON或者XML。用法如下：

|  |
| --- |
| **@Produces("application/xml")**  public StreamingOutput getCustomer(int id); |

该例表示，getCustomer()方法返回的类型是XML的；如果希望返回JSON类型，可以使用：@Produce(“application/json”)。

### @Consumes

与@Produces相反，@Consumes用于配置某一个方法可以接受的媒体类型（**输入类型**）。简单用法如：

|  |
| --- |
| @PUT  @Path("{id}")  **@Consumes("application/xml")**  public void updateCustomer(@PathParam("id") int id, InputStream is) ; |

该例表示，updateCustomer()方法的参数InputStream is可以接受的数据类型是XML的。

### @PathParam

该注解用于方法中的参数。如果方法的@Path注解没有指定参数，形如@Path(value = "/info")，那么此时如果希望方法的某个参数可以根据URI地址获得参数，则可以如下配置：

|  |
| --- |
| @GET  **@Path(value = "/info")**  Student getStudent(**@PathParam("id")** long id,  **@PathParam("name")** String name); |

此时访问[http://localhost:8080/ss/info**?id=32&name=celine**](http://localhost:8080/ss/info?id=32&name=celine)地址，将可以让参数id获得数字32，让参数name获得字符串celine。使用@PathParam注解时，参数是以**键值对**的形式存在的。

### @QueryParam

该注解用于方法中的参数。如果方法的@Path注解在配置的时候指定了参数，形如@Path(value = “/info2/{name}”)，那么此时该方法的参数需要使用@QueryParam注解获得URI地址中预留位置的参数内容。例如配置如下：

|  |
| --- |
| @GET  **@Path(value = "/info2/{name}")**  UserResponse getStudent(**@QueryParam("name")** String name); |

访问[http://localhost:8080/ss/info2/**celine**](http://localhost:8080/ss/info2/celine)地址，那么方法的参数String name将会获得字符串celine的值。使用@QueryParam注解时，参数是以**URI地址**的形式存在的。

如果希望传递多个参数，则可以使用@Path配置时预留两个参数位置，如：

|  |
| --- |
| @GET  **@Path(value = "/info3/{id}/{name}")**  UserResponse getStudent(**@QueryParam("id")** long id,  **@QueryParam("name")** String name); |

访问[http://localhost:8080/ss/info3/**123**/**celine**](http://localhost:8080/ss/info3/123/celine)地址，参数id将获得取值123，参数name将获得字符串celine。