Web Service 那点事儿（4）—— 使用 CXF 开发 REST 服务

2017/05/30 | 分类： 基础技术 | 0 条评论 | 标签： CXF, REST, Web Service

 原文出处： 黄勇

现在您已经学会了如何使用 CXF 开发基于 SOAP 的 Web 服务，也领略了 Spring + CXF 这个强大的组合，如果您错过了这精彩的一幕，请回头看看这篇吧：

Web Service 那点事儿（2） —— 使用 CXF 开发 SOAP 服务

今天我们将视角集中在 REST 上，它是继 SOAP 以后，另一种广泛使用的 Web 服务。与 SOAP 不同，REST 并没有 WSDL 的概念，也没有叫做“信封”的东西，因为 REST 主张用一种简单粗暴的方式来表达数据，传递的数据格式可以是 JSON 格式，也可以是 XML 格式，这完全由您来决定。

REST 全称是 Representational State Transfer（表述性状态转移），它是 Roy Fielding 博士在 2000 年写的一篇关于软件架构风格的论文，此文一出，威震四方！许多知名互联网公司开始采用这种轻量级 Web 服务，大家习惯将其称为 RESTful Web Services，或简称 REST 服务。

那么 REST 到底是什么呢?

REST 本质上是使用 URL 来访问资源的一种方式。总所周知，URL 就是我们平常使用的请求地址了，其中包括两部分：请求方式 与 请求路径，比较常见的请求方式是 GET 与 POST，但在 REST 中又提出了几种其它类型的请求方式，汇总起来有六种：GET、POST、PUT、DELETE、HEAD、OPTIONS。尤其是前四种，正好与 CRUD（增删改查）四种操作相对应：GET（查）、POST（增）、PUT（改）、DELETE（删），这正是 REST 的奥妙所在！

实际上，REST 是一个“无状态”的架构模式，因为在任何时候都可以由客户端发出请求到服务端，最终返回自己想要的数据。也就是说，服务端将内部资源发布 REST 服务，客户端通过 URL 来访问这些资源，这不就是 SOA 所提倡的“面向服务”的思想吗？所以，REST 也被人们看做是一种轻量级的 SOA 实现技术，因此在企业级应用与互联网应用中都得到了广泛使用。

在 Java 的世界里，有一个名为 JAX-RS 的规范，它就是用来实现 REST 服务的，目前已经发展到了 2.0 版本，也就是 JSR-339 规范，如果您想深入研究 REST，请深入阅读此规范。

JAX-RS 规范目前有以下几种比较流行的实现技术：

- Jersey：https://jersey.java.net/
- Restlet：http://restlet.com/
- RESTEasy：http://resteasy.jboss.org/
- CXF：http://cxf.apache.org/

本文以 CXF 为例，我努力用最精炼的文字，让您快速学会如何使用 CXF 开发 REST 服务，此外还会将 Spring 与 CXF 做一个整合，让开发更加高效！

那么还等什么呢？咱们一起出发吧！

1. 使用 CXF 发布与调用 REST 服务

第一步：添加 Maven 依赖

```
<!-- lang: xml -->
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

```

```
    <groupId>demo.ws</groupId>
    <artifactId>rest_cxf</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <cxf.version>3.0.0</cxf.version>
        <jackson.version>2.4.1</jackson.version>
    </properties>

    <dependencies>
        <!-- CXF -->
        <dependency>
            <groupId>org.apache.cxf</groupId>
            <artifactId>cxf-rt-frontend-jaxrs</artifactId>
            <version>${cxf.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.cxf</groupId>
            <artifactId>cxf-rt-transports-http-jetty</artifactId>
            <version>${cxf.version}</version>
        </dependency>
        <!-- Jackson -->
        <dependency>
            <groupId>com.fasterxml.jackson.jaxrs</groupId>
            <artifactId>jackson-jaxrs-json-provider</artifactId>
            <version>${jackson.version}</version>
        </dependency>
    </dependencies>

</project>
```

以上添加了 CXF 关于 REST 的依赖包，并使用了 Jackson 来实现 JSON 数据的转换。

第二步：定义一个 REST 服务接口

```
<!-- lang: java -->
package demo.ws.rest_cxf;

import java.util.List;
import java.util.Map;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.FormParam;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

```

```
17  public interface ProductService {
18
19      @GET
20      @Path("/products")
21      @Produces(MediaType.APPLICATION_JSON)
22      List<Product> retrieveAllProducts();
23
24      @GET
25      @Path("/product/{id}")
26      @Produces(MediaType.APPLICATION_JSON)
27      Product retrieveProductById(@PathParam("id") long id);
28
29      @POST
30      @Path("/products")
31      @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
32      @Produces(MediaType.APPLICATION_JSON)
33      List<Product> retrieveProductsByName(@FormParam("name") String name);
34
35      @POST
36      @Path("/product")
37      @Consumes(MediaType.APPLICATION_JSON)
38      @Produces(MediaType.APPLICATION_JSON)
39      Product createProduct(Product product);
40
41      @PUT
42      @Path("/product/{id}")
43      @Consumes(MediaType.APPLICATION_JSON)
44      @Produces(MediaType.APPLICATION_JSON)
45      Product updateProductById(@PathParam("id") long id, Map<String, Object> fieldMap);
46
47      @DELETE
48      @Path("/product/{id}")
49      @Produces(MediaType.APPLICATION_JSON)
50      Product deleteProductById(@PathParam("id") long id);
51  }
```

以上 ProductService 接口中提供了一系列的方法，在每个方法上都使用了 JAX-RS 提供的注解，主要包括以下三类：

1. 请求方式注解，包括：@GET、@POST、@PUT、@DELETE
2. 请求路径注解，包括：@Path，其中包括一个路径参数
3. 数据格式注解，包括：@Consumes（输入）、@Produces（输出），可使用 MediaType 常量
4. 相关参数注解，包括：@PathParam（路径参数）、@FormParam（表单参数），此外还有 @QueryParam（请求参数）

针对 updateProductById 方法，简单解释一下：

该方法将被 PUT:/product/{id} 请求来调用，请求路径中的 id 参数将映射到 long id 参数上，请求体中的数据将自动转换为 JSON 格式并映射到 Map<String, Object> fieldMap 参数上，返回的 Product 类型的数据将自动转换为 JSON 格式并返回到客户端。

注意：由于 Product 类与 ProductService 接口的实现类并不是本文的重点，因此省略了，本文结尾处会给出源码链接。

第三步：使用 CXF 发布 REST 服务

```
<!-- lang: java -->
package demo.ws.rest_cxf;

import java.util.ArrayList;
import java.util.List;
import org.apache.cxf.jaxrs.JAXRSServerFactoryBean;
import org.apache.cxf.jaxrs.lifecycle.ResourceProvider;
import org.apache.cxf.jaxrs.lifecycle.SingletonResourceProvider;
import org.codehaus.jackson.jaxrs.JacksonJsonProvider;

public class Server {

    public static void main(String[] args) {
        // 添加 ResourceClass
        List<Class<?>> resourceClassList = new ArrayList<Class<?>>();
        resourceClassList.add(ProductServiceImpl.class);

        // 添加 ResourceProvider
        List<ResourceProvider> resourceProviderList = new ArrayList<ResourceProvider>();
        resourceProviderList.add(new SingletonResourceProvider(new ProductServiceImpl()));

        // 添加 Provider
        List<Object> providerList = new ArrayList<Object>();
        providerList.add(new JacksonJsonProvider());

        // 发布 REST 服务
        JAXRSServerFactoryBean factory = new JAXRSServerFactoryBean();
        factory.setAddress("http://localhost:8080/ws/rest");
        factory.setResourceClasses(resourceClassList);
        factory.setResourceProviders(resourceProviderList);
        factory.setProviders(providerList);
        factory.create();
        System.out.println("rest ws is published");
    }
}
```

CXF 提供了一个名为 org.apache.cxf.jaxrs.JAXRSServerFactoryBean 的类，专用于发布 REST 服务，只需为该类的实例对象指定四个属性即可：

1. Address：REST 基础地址
2. ResourceClasses：一个或一组相关的资源类，即接口对应的实现类（注意：REST 规范允许资源类没有接口）
3. ResourceProviders：资源类对应的 Provider，此时使用 CXF 提供的 org.apache.cxf.jaxrs.lifecycle.SingletonResourceProvider 进行装饰

4. Providers：REST 服务所需的 Provider，此时使用了 Jackson 提供的 org.codehaus.jackson.jaxrs.JacksonJsonProvider，用于实现 JSON 数据的序列化与反序列化

运行以上 Server 类，将以 standalone 方式发布 REST 服务，下面我们通过客户端来调用以发布的 REST 服务。

第四步：使用 CXF 调用 REST 服务

首先添加如下 Maven 依赖：

```
<!-- lang: xml -->
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-rs-client</artifactId>
    <version>${cxf.version}</version>
</dependency>
```

CXF 提供了三种 REST 客户端，下面将分别进行展示。

第一种：JAX-RS 1.0 时代的客户端

```
<!-- lang: java -->
package demo.ws.rest_cxf;

import java.util.ArrayList;
import java.util.List;
import org.apache.cxf.jaxrs.client.JAXRSClientFactory;
import org.codehaus.jackson.jaxrs.JacksonJsonProvider;

public class JAXRSClient {

    public static void main(String[] args) {
        String baseAddress = "http://localhost:8080/ws/rest";

        List<Object> providerList = new ArrayList<Object>();
        providerList.add(new JacksonJsonProvider());

        ProductService productService = JAXRSClientFactory.create(baseAddress, ProductService.class, providerList);
        List<Product> productList = productService.retrieveAllProducts();
        for (Product product : productList) {
            System.out.println(product);
        }
    }
}
```

本质是使用 CXF 提供的 org.apache.cxf.jaxrs.client.JAXRSClientFactory 工厂类来创建 ProductService 代理对象，通过代理对象调用目标对象上的方法。客户端同样也需要使用 Provider，此时仍然使用了 Jackson 提供的 org.codehaus.jackson.jaxrs.JacksonJsonProvider。

第二种：JAX-RS 2.0 时代的客户端

```
<!-- lang: java -->
package demo.ws.rest_cxf;

import java.util.List;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.core.MediaType;
import org.codehaus.jackson.jaxrs.JacksonJsonProvider;

public class JAXRS20Client {

    public static void main(String[] args) {
        String baseAddress = "http://localhost:8080/ws/rest";

        JacksonJsonProvider jsonProvider = new JacksonJsonProvider();

        List productList = ClientBuilder.newClient()
            .register(jsonProvider)
            .target(baseAddress)
            .path("/products")
            .request(MediaType.APPLICATION_JSON)
            .get(List.class);
        for (Object product : productList) {
            System.out.println(product);
        }
    }
}
```

在 JAX-RS 2.0 中提供了一个名为 javax.ws.rs.client.ClientBuilder 的工具类，可用于创建客户端并调用 REST 服务，显然这种方式比前一种要先进，因为在代码中不再依赖 CXF API 了。

如果想返回带有泛型的 List<Product>，那么可以使用以下代码片段：

```
<!-- lang: java -->
List<Product> productList = ClientBuilder.newClient()
    .register(jsonProvider)
    .target(baseAddress)
    .path("/products")
    .request(MediaType.APPLICATION_JSON)
    .get(new GenericType<List<Product>>() {});
for (Product product : productList) {
    System.out.println(product);
}
```

第三种：通用的 WebClient 客户端

```
<!-- lang: java -->
package demo.ws.rest_cxf;

```

```
import java.util.ArrayList;
import java.util.List;
import javax.ws.rs.core.GenericType;
import javax.ws.rs.core.MediaType;
import org.apache.cxf.jaxrs.client.WebClient;
import org.codehaus.jackson.jaxrs.JacksonJsonProvider;

public class CXFWebClient {

    public static void main(String[] args) {
        String baseAddress = "http://localhost:8080/ws/rest";

        List<Object> providerList = new ArrayList<Object>();
        providerList.add(new JacksonJsonProvider());

        List productList = WebClient.create(baseAddress, providerList)
            .path("/products")
            .accept(MediaType.APPLICATION_JSON)
            .get(List.class);
        for (Object product : productList) {
            System.out.println(product);
        }
    }
}
```

CXF 还提供了一种更为简洁的方式，使用 org.apache.cxf.jaxrs.client.WebClient 来调用 REST 服务，这种方式在代码层面上还是相当简洁的。

如果想返回带有泛型的 List<Product>，那么可以使用以下代码片段：

```
<!-- lang: java -->
List<Product> productList = WebClient.create(baseAddress, providerList)
    .path("/products")
    .accept(MediaType.APPLICATION_JSON)
    .get(new GenericType<List<Product>>() {});
for (Product product : productList) {
    System.out.println(product);
}
```

2. 使用 Spring + CXF 发布 REST 服务

第一步：添加 Maven 依赖

```
<!-- lang: xml -->
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

```

```
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>demo.ws</groupId>
    <artifactId>rest_spring_cxf</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <spring.version>4.0.6.RELEASE</spring.version>
        <cxf.version>3.0.0</cxf.version>
        <jackson.version>2.4.1</jackson.version>
    </properties>

    <dependencies>
        <!-- Spring -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <!-- CXF -->
        <dependency>
            <groupId>org.apache.cxf</groupId>
            <artifactId>cxf-rt-frontend-jaxrs</artifactId>
            <version>${cxf.version}</version>
        </dependency>
        <!-- Jackson -->
        <dependency>
            <groupId>com.fasterxml.jackson.jaxrs</groupId>
            <artifactId>jackson-jaxrs-json-provider</artifactId>
            <version>${jackson.version}</version>
        </dependency>
    </dependencies>

</project>
```

这里仅依赖 Spring Web 模块（无需 MVC 模块），此外就是 CXF 与 Jackson 了。

第二步：配置 web.xml

```
<!-- lang: xml -->
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee

http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"

         version="3.0">
```

```
    <!-- Spring -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring.xml</param-value>
    </context-param>
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <!-- CXF -->
    <servlet>
        <servlet-name>cxf</servlet-name>
        <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>cxf</servlet-name>
        <url-pattern>/ws/*</url-pattern>
    </servlet-mapping>

</web-app>
```

使用 Spring 提供的 ContextLoaderListener 去加载 Spring 配置文件 spring.xml；使用 CXF 提供的 CXFServlet 去处理前缀为 /ws/ 的 REST 请求。

第三步：将接口的实现类发布 SpringBean

```
<!-- lang: java -->
package demo.ws.rest_spring_cxf;

import org.springframework.stereotype.Component;

@Component
public class ProductServiceImpl implements ProductService {
    ...
}
```

使用 Spring 提供的 @Component 注解，将 ProductServiceImpl 发布为 Spring Bean，交给 Spring IOC 容器管理，无需再进行 Spring XML 配置。

第四步：配置 Spring

以下是 spring.xml 的配置：

```
<!-- lang: xml -->
<?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:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-4.0.xsd


http://www.springframework.org/schema/context


http://www.springframework.org/schema/context/spring-context-4.0.xsd">

    <context:component-scan base-package="demo.ws"/>

    <import resource="spring-cxf.xml"/>

</beans>
```

在以上配置中扫描 demo.ws 这个基础包路径，Spring 可访问该包中的所有 Spring Bean，比如，上面使用 @Component 注解发布的 ProductServiceImpl。此外，加载了另一个配置文件 spring-cxf.xml，其中包括了关于 CXF 的相关配置。

以下是 spring-cxf.xml 的配置：

```
<!-- lang: xml -->
<?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-4.0.xsd


http://cxf.apache.org/jaxrs


http://cxf.apache.org/schemas/jaxrs.xsd">

    <jaxrs:server address="/rest">
        <jaxrs:serviceBeans>
            <ref bean="productServiceImpl"/>
        </jaxrs:serviceBeans>
        <jaxrs:providers>
            <bean class="com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider"/>
        </jaxrs:providers>
    </jaxrs:server>

</beans>
```

使用 CXF 提供的 Spring 命名空间来配置 Service Bean（即上文提到的 Resource Class）与 Provider。注意，这里配置了一个 address 属性为“/rest”，表示 REST 请求的相对路径，与 web.xml 中配置的“/ws/*”结合起来，最终的 REST 请求根路径是“/ws/rest”，在 ProductService 接口方法上 @Path 注解所配置的路径只是一个相对路径。

第五步：调用 REST 服务

```
<!-- lang: html -->
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Demo</title>
    <link href="http://cdn.bootcss.com/bootstrap/3.1.1/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>

<div class="container">
    <div class="page-header">
        <h1>Product</h1>
    </div>
    <div class="panel panel-default">
        <div class="panel-heading">Product List</div>
        <div class="panel-body">
            <div id="product"></div>
        </div>
    </div>
</div>

<script src="http://cdn.bootcss.com/jquery/2.1.1/jquery.min.js"></script>
<script src="http://cdn.bootcss.com/bootstrap/3.1.1/js/bootstrap.min.js"></script>
<script src="http://cdn.bootcss.com/handlebars.js/1.3.0/handlebars.min.js"></script>

<script type="text/x-handlebars-template" id="product_table_template">
    {{#if data}}
        <table class="table table-hover" id="product_table">
            <thead>
                <tr>
                    <th>ID</th>
                    <th>Product Name</th>
                    <th>Price</th>
                </tr>
            </thead>
            <tbody>
                {{#data}}
                    <tr data-id="{{id}}" data-name="{{name}}">
                        <td>{{id}}</td>
                        <td>{{name}}</td>
                        <td>{{price}}</td>
                    </tr>
                {{/data}}
            </tbody>
```

```
        </table>
    {{else}}
        <div class="alert alert-warning">Can not find any data!</div>
    {{/if}}
</script>

<script>
    $(function() {
        $.ajax({
            type: 'get',
            url: 'http://localhost:8080/ws/rest/products',
            dataType: 'json',
            success: function(data) {
                var template = $("#product_table_template").html();
                var render = Handlebars.compile(template);
                var html = render({
                    data: data
                });
                $('#product').html(html);
            }
        });
    });
</script>

</body>
</html>
```

使用一个简单的 HTML 页面来调用 REST 服务，也就是说，前端发送 AJAX 请求来调用后端发布的 REST 服务。这里使用了 jQuery、Bootstrap、Handlebars.js 等技术。

3. 关于 AJAX 的跨域问题

如果服务端部署在 foo.com 域名下，而客户端部署在 bar.com 域名下，此时从 bar.com 发出一个 AJAX 的 REST 请求到 foo.com，就会出现报错：

No ‘Access-Control-Allow-Origin’ header is present on the requested resource.

要想解决以上这个 AJAX 跨域问题，有以下两种解决方案：

方案一：使用 JSONP 解决 AJAX 跨域问题

JSONP 的全称是 JSON with Padding，实际上是在需要返回的 JSON 数据外，用一个 JS 函数进行封装。

可以这样来理解，服务器返回一个 JS 函数，参数是一个 JSON 数据，例如：callback({您的 JSON 数据})，虽然 AJAX 不能跨域访问，但 JS 脚本是可以跨域执行的，因此客户端将执行这个 callback 函数，并获取其中的 JSON 数据。

如果需要返回的 JSON 数据是：

```
{"id":2,"name":"ipad mini","price":2500},{"id":1,"name":"iphone 5s","price":5000}
```

那么对应的 JSONP 格式是：

```
callback([{"id":2,"name":"ipad mini","price":2500},{"id":1,"name":"iphone 5s","price":5000}]);
```

CXF 已经提供了对 JSONP 的支持，只需要通过简单的配置即可实现。

首先，添加 Maven 依赖：

```
<!-- lang: xml -->
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-rs-extension-providers</artifactId>
    <version>${cxf.version}</version>
</dependency>
```

然后，添加 CXF 配置：

```
<!-- lang: xml -->
<jaxrs:server address="/rest">
    <jaxrs:serviceBeans>
        <ref bean="productServiceImpl"/>
    </jaxrs:serviceBeans>
    <jaxrs:providers>
        <bean class="com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider"/>
        <bean class="org.apache.cxf.jaxrs.provider.jsonp.JsonpPreStreamInterceptor"/>
    </jaxrs:providers>
    <jaxrs:inInterceptors>
        <bean class="org.apache.cxf.jaxrs.provider.jsonp.JsonpInInterceptor"/>
    </jaxrs:inInterceptors>
    <jaxrs:outInterceptors>
        <bean class="org.apache.cxf.jaxrs.provider.jsonp.JsonpPostStreamInterceptor"/>
    </jaxrs:outInterceptors>
</jaxrs:server>
```

注意：JsonpPreStreamInterceptor 一定要放在 <jaxrs:providers> 中，而不是 <jaxrs:inInterceptors> 中，这也许是 CXF 的一个 Bug，可以点击以下链接查看具体原因：

http://cxf.547215.n5.nabble.com/JSONP-is-not-works-td5739858.html

最后，使用 jQuery 发送基于 JSONP 的 AJAX 请求：

```
<!-- lang: js -->
$.ajax({
    type: 'get',
    url: 'http://localhost:8080/ws/rest/products',
    dataType: 'jsonp',
    jsonp: '_jsonp',
    jsonpCallback: 'callback',
    success: function(data) {
        var template = $("#product_table_template").html();
        var render = Handlebars.compile(template);
        var html = render({
            data: data
        });
        $('#product').html(html);
    }
});
```

以上代码中有三个选项需要加以说明：

1. dataType：必须为“jsonp”，表示返回的数据类型为 JSONP 格式
2. jsonp：表示 URL 中 JSONP 回调函数的参数名，CXF 默认接收的参数名是“_jsonp”，也可以在 JsonpInInterceptor 中配置
3. jsonpCallback：表示回调函数的名称，若未指定，则由 jQuery 自动生成

方案二：使用 CORS 解决 AJAX 跨域问题

CORS 的全称是 Cross-Origin Resource Sharing（跨域资源共享），它是 W3C 提出的一个 AJAX 跨域访问规范，可以从以下地址了解此规范：

http://www.w3.org/TR/cors/

相比 JSONP 而言，CORS 更为强大，因为它弥补了 JSONP 只能处理 GET 请求的局限性，但是只有较为先进的浏览器才能全面支持 CORS。

CXF 同样也提供了对 CORS 的支持，通过简单的配置就能实现。

首先，添加 Maven 依赖：

```
<!-- lang: xml -->
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-rs-security-cors</artifactId>
    <version>${cxf.version}</version>
</dependency>
```

然后，添加 CXF 配置：

```
<!-- lang: xml -->
<jaxrs:server address="/rest">
    <jaxrs:serviceBeans>
        <ref bean="productServiceImpl"/>
    </jaxrs:serviceBeans>
    <jaxrs:providers>
        <bean class="com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider"/>
        <bean class="org.apache.cxf.rs.security.cors.CrossOriginResourceSharingFilter">
            <property name="allowOrigins" value="http://localhost"/>
        </bean>
    </jaxrs:providers>
</jaxrs:server>
```

在 CrossOriginResourceSharingFilter 中配置 allowOrigins 属性，将其设置为客户端的域名，示例中为“http://localhost”，需根据实际情况进行设置。

最后，使用 jQuery 发送 AJAX 请求：

就像在相同域名下访问一样，无需做任何配置。

注意：在 IE8 中使用 jQuery 发送 AJAX 请求时，需要配置 $.support.cors = true，才能开启 CORS 特性。

4. 总结

本文让您学会了如何使用 CXF 发布 REST 服务，可以独立使用 CXF，也可以与 Spring 集成。此外，CXF 也提供了一些解决方案，用于实现跨域 AJAX 请求，比如：JSONP 或 CORS。CXF 3.0 以全面支持 JAX-RS 2.0 规范，有很多实用的功能需要您进一步学习，可以点击以下地址：

http://cxf.apache.org/docs/jax-rs.html

目前您所看到的 REST 请求没有任何的身份认证，这样是很不安全的，也就意味着任何人只要知道了 REST 地址就能调用。我们知道 SOAP 里有 WS-Security 规范，可以使用 WSS4J 来做 SOAP 安全，那么关于 REST 安全我们应该如何保证呢？下一篇将为您揭晓，敬请期待！

源码地址：http://git.oschina.net/huangyong/cxf_demo

相关文章

- Web Service 那点事儿（2）—— 使用 CXF 开发 SOAP 服务
- Web Service 那点事儿（3）—— SOAP 及其安全控制
- Web Service监控教程：如何识别不良部署
- Web Service入门
- 使用Java创建RESTful Web Service
- Java Web Services面试问题集锦
- 安卓开发者必备的42个链接
- 【深入Java虚拟机（8）】：Java垃圾收集机制
- Spring源码分析：非懒加载的单例Bean初始化前后的一些操作
- Java Proxy 和 CGLIB 动态代理原理

发表评论

Name*

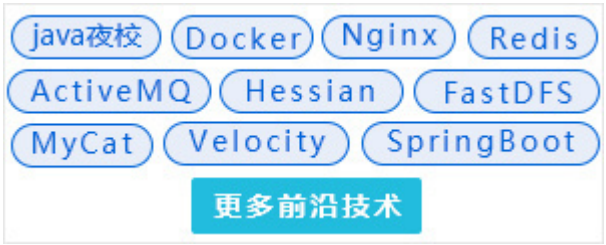
邮箱*

网站 (请以 http://开头)

(*) 表示必填项

« tomcat ssi 配置及升级导致 ssi include 错误问题解决
Spring Boot异常处理详解 »

Search



- 本周热门文章
- 本月热门
- 热门标签

0 图解 CMS 垃圾回收机制，你值得...

1 2018 年 Java 平台发布计划之新...

2 Java 异常进阶

3 通向架构师的道路（第一天）之 Apache ...

4 G1 垃圾收集器之对象分配过程

5 面试必问的 volatile，你了解多少?

6 通向架构师的道路（第二天）之 apache tom...

7 通向架构师的道路（第三天）之 apach...

8 通向架构师的道路（第四天）之 Tomc...

0 大型网站系统与 Java 中间件实践

1 深入Spring Boot：那些注入不了的 Sp...

2 谈谈 Tomcat 请求处理流程

3 深入 Spring Boot：排查 @Transactional 引起...

4 G1 垃圾收集器介绍

5 一点解决版本冲突的应急思路、怎样在所...

6 图解 CMS 垃圾回收机制，你值得...

7 高性能线程间队列 DISRUPTO...

8 谈谈 Tomcat 架构及启动过程[含...

9 代码生成利器：IDEA 强大的 Live T...

android23days Android开发 AOP ArrayList ConcurrentHashMap Eclipse GC Guava Hadoop HashMap HashSet HBase Hibernate IntelliJ io Java java8 java 8 Java9 javaee Java NIO Java乱码 Java编程入门 JDBC JDK JMX JPA Jsoup JUnit JVM Lambda log4j maven Mybatis Netty nio oracle ORM redis RESTful Scala Servlet Socket solr Spring Spring4 spring boot springboot Spring MVC SpringMVC Spring Security String synchronized TestNG ThreadLocal Tomcat volatile Zookeeper 事务 内存管理 分布式 动态代理 单例 参数太多怎么办 反射 垃圾回收 基础技术 多线程 字符串 字节码 并发 并发编程 序列化 异常 异常处理 性能 性能优化 性能调优 教程 数据结构 日志 架构 死锁 泛型 注解 测试 游戏 源码分析 算法 线程 线程池 缓存 自动化测试 虚拟机 设计模式 负载均衡 资讯 集合 面试 面试题

最新评论

关于ImportNew

ImportNew 专注于 Java 技术分享。于2012年11月11日 11:11正式上线。是的，这是一个很特别的时刻 :)

ImportNew 由两个 Java 关键字 import 和 new 组成，意指：Java 开发者学习新知识的网站。 import 可认为是学习和吸收， new 则可认为是新知识、新技术圈子和新朋友……

-

联系我们

Email：ImportNew.com@gmail.com
新浪微博：@ImportNew
推荐微信号

反馈建议：ImportNew.com@gmail.com
广告与商务合作QQ：2302462408

推荐关注

小组 — 好的话题、有启发的回复、值得信赖的圈子
头条 — 写了文章？看干货？去头条！
相亲 — 为IT单身男女服务的征婚传播平台
资源 — 优秀的工具资源导航
翻译 — 活跃 & 专业的翻译小组
博客 — 国内外的精选博客文章
设计 — UI,网页，交互和用户体验
前端 — JavaScript, HTML5, CSS
安卓 — 专注Android技术分享
iOS — 专注iOS技术分享

Java — 专注Java技术分享
Python — 专注Python技术分享