**Jersey**

**简明手册**

**V3**

**张镇**

**目 录**

[1 前言 1](#_Toc490813097)

[1.1 RESTful Web服务 1](#_Toc490813098)

[1.2 Jersey是什么 2](#_Toc490813099)

[2 环境及部署 2](#_Toc490813100)

[2.1 创建Maven工程，获取所需jar包 2](#_Toc490813101)

[2.2 配置Jersey 6](#_Toc490813102)

[2.3 与Spring集成 7](#_Toc490813103)

[2.3.1 通过Maven获取Spring集成所需Jar包 7](#_Toc490813104)

[2.3.2 修改ApplicationContext.xml 11](#_Toc490813105)

[3 说明 12](#_Toc490813106)

[3.1 Jersey对JSON的支持 12](#_Toc490813107)

[3.1.1 基于POJO 12](#_Toc490813108)

[3.1.2 基于JAXB 15](#_Toc490813109)

[3.1.3 基于底层JSONObject/JSONArray 16](#_Toc490813110)

[3.1.4 总结 19](#_Toc490813111)

[3.2 Jersey对HTTPS的支持 19](#_Toc490813112)

[4 示例代码 28](#_Toc490813113)

[4.1 Jersey常用注解解释 28](#_Toc490813114)

[4.1.1 @Path 30](#_Toc490813115)

[4.1.2 @PathParam 30](#_Toc490813116)

[4.1.3 @GET 30](#_Toc490813117)

[4.1.4 @Consumes 30](#_Toc490813118)

[4.1.5 @Produces 30](#_Toc490813119)

[4.1.6 @QueryParam 31](#_Toc490813120)

[4.1.7 @Context 31](#_Toc490813121)

[4.1.8 @Singleton和@PerRequest 31](#_Toc490813122)

[4.2 资源加载器代码 31](#_Toc490813123)

[4.3 值对象代码 32](#_Toc490813124)

[4.4 基础接口服务代码 33](#_Toc490813125)

[4.5 与Spring集成的接口服务代码 35](#_Toc490813126)

[4.6 调用接口 37](#_Toc490813127)

[4.7 Clien代码 37](#_Toc490813128)

[5 测试 38](#_Toc490813129)

[6 RESTful API 设计指南 40](#_Toc490813130)

[6.1 协议 40](#_Toc490813131)

[6.2 域名 40](#_Toc490813132)

[6.3 版本（Versioning） 40](#_Toc490813133)

[6.4 路径（Endpoint） 41](#_Toc490813134)

[6.5 HTTP动词 41](#_Toc490813135)

[6.6 过滤信息（Filtering） 42](#_Toc490813136)

[6.7 状态码（Status Codes） 43](#_Toc490813137)

[6.8 错误处理（Error handling） 44](#_Toc490813138)

[6.9 返回结果 44](#_Toc490813139)

[6.10 Hypermedia API 45](#_Toc490813140)

[6.11 其他 46](#_Toc490813141)

# 前言

## RESTful Web服务

REST 在 2000 年由 Roy Fielding 在博士论文中提出，他是 HTTP 规范 1.0 和 1.1 版的首席作者之一。

首先，“REST”是 Representational State Transfer的缩写，如果一个架构符合REST原则，它就是RESTful架构。

在REST中的一切都被认为是一种资源。所谓“资源”，就是网络上的一个实体，或者说是网络上的一个具体信息。它可以是一段文本、一张图片、一首歌曲、一种服务，总之就是一个具体的实在。你可以用一个URI（统一资源定位符）指向它，每种资源对应一个特定的URI。要获取这个资源，访问它的URI就可以，因此URI就成了每一个资源的地址或独一无二的识别符。

所谓“上网”，就是与互联网上一系列的“资源”互动，调用它的URI。我们把“资源”具体呈现出来的形式，叫做它的“表现层”（Representation）。比如，文本可以用txt格式表现，也可以用HTML格式、XML格式、JSON格式表现，甚至可以采用二进制格式；图片可以用JPG格式表现，也可以用PNG格式表现。

URI只代表资源的实体，不代表它的 形式。URI只代表资源的 位置。它的具体表现形式，应该在HTTP请求的头信息中用 Accept和 Content-Type字段指定，这两个字段才是对“表现层”的描述。

客户端和服务器的一个互动过程。在这个过程中，势必涉及到数据和状态的变化。

互联网通信协议HTTP协议，是一个“无状态协议”。这意味着，所有的状态都保存在服务器端。因此，如果客户端想要操作服务器，必须通过某种手段，让服务器端发生“状态转化”（State Transfer）。而这种转化是建立在表现层之上的，所以就是“表现层状态转化”。

客户端用到的手段，只能是HTTP协议。具体来说就是HTTP协议里面，四个表示操作方式的动词：GET、POST、PUT、DELETE。它们分别对应四种基本操作：GET用来获取资源，POST用来新建资源，PUT用来新建或更新资源，DELETE用来删除资源。

通常，RESTful Web 服务应该定义以下方面：

* Web 服务的基/根 URI，比如 <http://hostname/>{appcontext}/resources。
* 支持 MIME 类型的响应数据，包括 JSON/XML/ATOM 等等。
* 服务支持的操作集合（例如 POST、GET、PUT 或 DELETE）。

**HTTP 方法操作映射表**

|  |  |  |
| --- | --- | --- |
| **GET** | 查看资源 | 查看ID为1234的资源 |
| **POST** | 创建一个资源 | 创建一个子资源 |
| **PUT** | 创建或更新一个资源 | 创建或更新ID为1234的资源 |
| **DELETE** | 删除资源 | 删除ID为1234的资源 |

## Jersey是什么

Jersey RESTful 框架是开源的RESTful框架, 实现了 JAX-RS规范。它扩展了JAX-RS 参考实现， 提供了更多的特性和工具， 可以进一步地简化 RESTful service 和 client 开发。

尽管相对年轻，它已经是一个产品级的 RESTful service和 client框架。

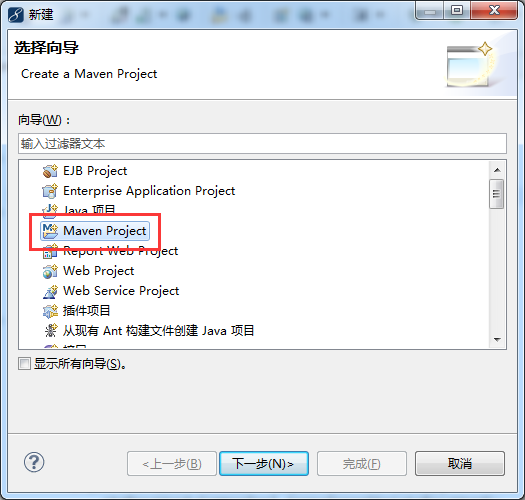
Jersey通过注解的方式，定义接口的访问URI、接收参数，同时支持将接收的json或xml转化为对应的对象，也可以自动将返回的对象转化为json或xml格式，大大简化了接口的开发和调试。

通过简单的配置，Jersey可以将编写好的Server端代码发布为符合RESTful风格的接口，client端以http方式访问，即可进行相应的交互操作。

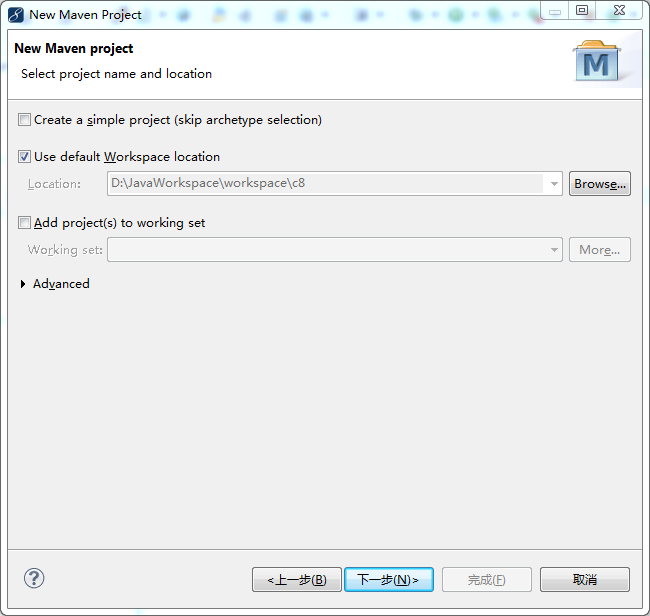
# 环境及部署

## 创建Maven工程，获取所需jar包

进入Eclipse，创建一个新项目，选择创建“Maven Project”。

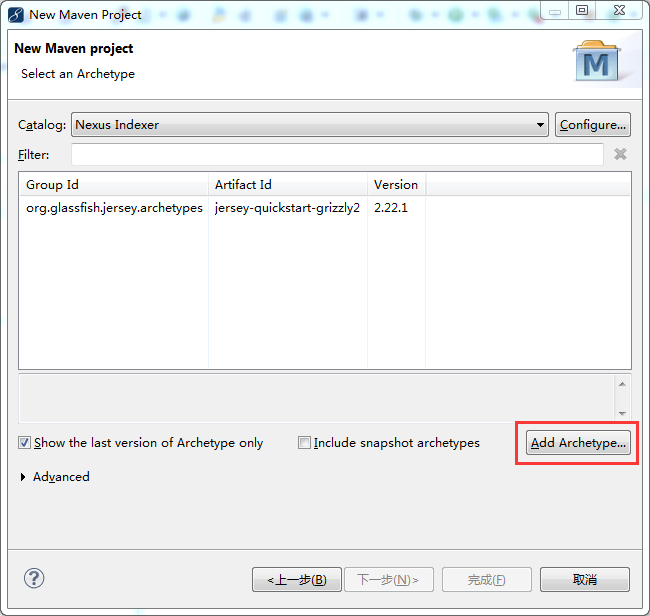


此页面中，选择在当前工作空间中创建项目。默认选择就好，不用改，直接点“下一步”。



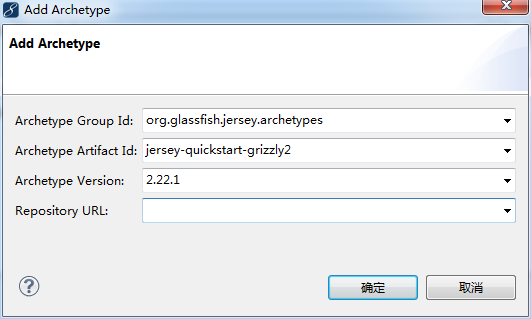
此界面中，可以直接在Filter中输入“jersey”，如果在列表中有对应的结果，则直接选择即可。

如果没有任何结果，需要点击右下角的“Add Archetype”按钮。



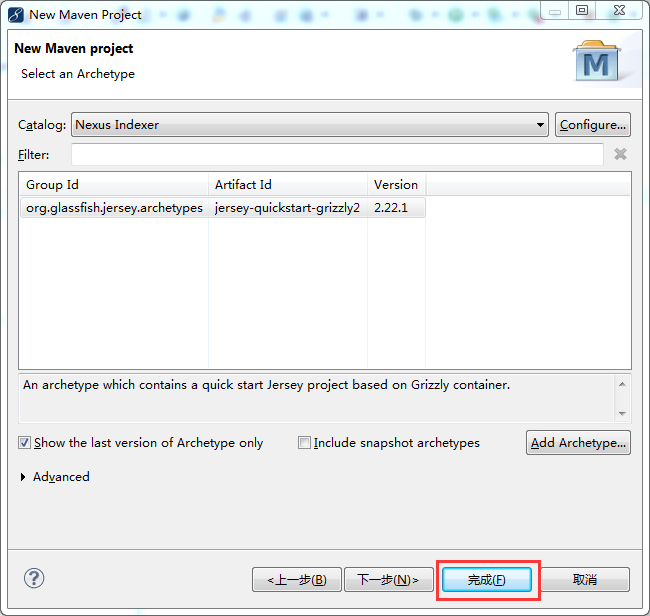
此页面中，按照如下内容输入：

* Archetype Group Id: org.glassfish.jersey.archetypes
* Archetype Artifact Id: jersey-quickstart-grizzly2
* Archetype Version: 2.22.1（这个版本，要去Jersey官网上去查一下。因为我后面用的是与Spring3集成，从兼容稳定性考虑，并未采用最新版，目前最新版为2.6）



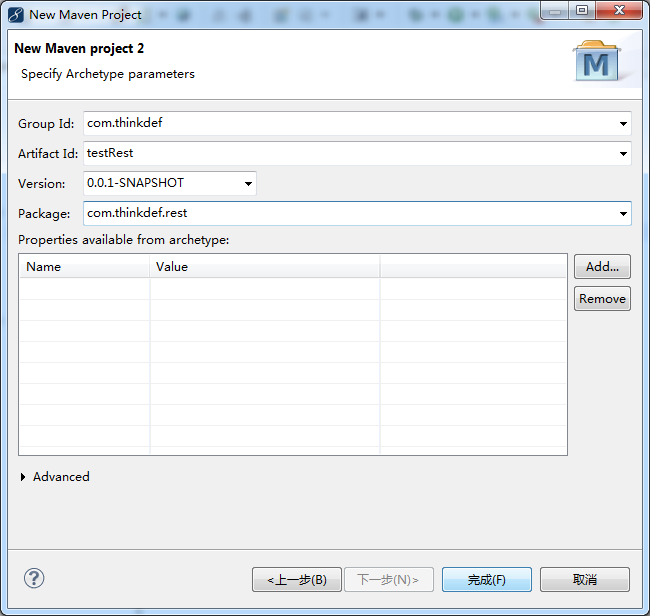
输入完毕，点击“确定”按钮，系统返回添加Maven项目界面。

选择添加的Maven包，点击此界面“下一步”按钮。



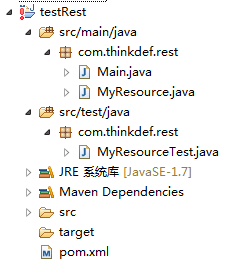
界面中按照如下内容添加：

* Group Id： 此处输入组ID，即代码包的命名空间。本例中输入“com.thinkdef”。
* Artifact Id： 此处输入工程的名称。本例中输入“testRest”。
* Package： 上面两项输入完毕后，系统会自动拼装包名。此处将包名最后一级的“testRest”修改为“rest”。本例中输入“com.thinkdef.rest”



点击“完成”按钮，系统自动创建项目，并自动从互联网中获取所需的jar包。

创建完毕的工程文件夹如下图所示。



点击打开“Maven Dependencies”，可以看到通过Maven获取到的此项目中需要使用的所有Jar文件。



将以上jar包拷贝出来，复制到所需的项目中；或使用Eclipse的“库管理”功能，将其加入到项目的构建路径中。

注意：部分公用jar包可能与项目中已有jar冲突，建议保留最新版，去除重复或冲突的jar包。

## 配置Jersey

如果需要Jersey生效，必须在web.xml中为它配置对应的servlet。

打开“web.xml”文件，将以下内容加入其中（放在已有的servlet配置的后面即可）。

<!-- Jersey REST 设置-->

<servlet>

<servlet-name>JerseyServlet</servlet-name>

<servlet-class>

org.glassfish.jersey.servlet.ServletContainer

</servlet-class>

<!-- rest服务路径 -->

<init-param>

<param-name>javax.ws.rs.Application</param-name>

<param-value>com.thinkdef.core.api.ApplicationAPI</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>JerseyServlet</servlet-name>

<url-pattern>/api/\*</url-pattern>

</servlet-mapping>

其中：

* Jersey支持两种资源加载模式，推荐采用“自定义资源加载器”方式，即如：“<param-value>com.thinkdef.core.api.ApplicationAPI</param-value>”，此配置中的“com.thinkdef.core.api.ApplicationAPI”为资源加载器，后面的“示例代码”中会对其进行介绍。
* 默认资源加载器（不建议使用）。在“param-name”中配置“jersey.config.server.provider.packages”，在“param-value”中配置资源类所在的命名空间（例如“com.cisdi.jersey.api”）即可，系统会自动扫描对应的命名空间，将获取到的所有资源类自动进行发布。
* “<url-pattern>/api/\*</url-pattern>”为服务访问的url路径。此配置即表示，后续所有接口访问url均如：<http://localhost:port/api/XXXX>

## 与Spring集成

### 通过Maven获取Spring集成所需Jar包

既然要通过Maven获取相应jar包，那么就要修改pom.xml。

在Eclipse中打开刚刚创建的Jersey项目的pom.xml文件，根据实际项目中使用的工具包的情况，将以下内容酌情添加进去。

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<!-- Jersey版本号，根据实际选用版本情况修改 -->

<jersey.version>2.22.1</jersey.version>

<!-- Jersey-Spring版本号，应与Jersey版本对应，根据实际选用版本情况修改 -->

<jersey-spring.version>2.22.1</jersey-spring.version>

<!-- Spring版本号，根据实际选用版本情况修改 -->

<spring.version>4.2.7.RELEASE</spring.version>

<!-- Servlet版本号，无需修改 -->

<servlet-api-version>3.1.0</servlet-api-version>

<!-- logback版本号，无需修改 -->

<logback.version>1.1.1</logback.version>

<!-- jcloverslf4j版本号，无需修改 -->

<jcloverslf4j.version>1.7.6</jcloverslf4j.version>

</properties>

<dependencies>

<!-- Jersey 配置 -->

<dependency>

<groupId>org.glassfish.jersey.ext</groupId>

<artifactId>jersey-spring3</artifactId>

<version>${jersey-spring.version}</version>

<exclusions>

<exclusion>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

</exclusion>

<exclusion>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

</exclusion>

<exclusion>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.glassfish.jersey.containers</groupId>

<artifactId>jersey-container-servlet</artifactId>

<version>${jersey-spring.version}</version>

</dependency>

<dependency>

<groupId>org.glassfish.jersey.containers</groupId>

<artifactId>jersey-container-servlet-core</artifactId>

<version>${jersey-spring.version}</version>

</dependency>

<dependency>

<groupId>org.glassfish.jersey.media</groupId>

<artifactId>jersey-media-json-jackson</artifactId>

<version>${jersey.version}</version>

</dependency>

<dependency>

<groupId>org.glassfish.jersey.media</groupId>

<artifactId>jersey-media-multipart</artifactId>

<version>${jersey.version}</version>

</dependency>

<dependency>

<groupId>org.glassfish.jersey.ext</groupId>

<artifactId>jersey-entity-filtering</artifactId>

<version>${jersey.version}</version>

</dependency>

<!-- Spring4.2 配置 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

<version>${spring.version}</version>

<scope>test</scope>

</dependency>

<!-- Servlet 配置 -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>${servlet-api-version}</version>

<scope>provided</scope>

</dependency>

<!-- Logback 配置 -->

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>${logback.version}</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>jcl-over-slf4j</artifactId>

<version>${jcloverslf4j.version}</version>

</dependency>

<!-- junit 配置 -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

</dependencies>

保存后，Eclipse会自动从网上自动获取对应的jar包。更新完毕后，将所需jar包复制出来，以供后续使用。

Jar包可以复制到已有项目的lib目录中，也可以通过“库”的方式加入到现有项目中。

### 修改ApplicationContext.xml

首先确认配置文件的头部是否配置了正确的schema。

本例中使用Spring 3.1.0，所以文件头的配置信息如下：

<?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:p=*"http://www.springframework.org/schema/p"*

xmlns:context=*"http://www.springframework.org/schema/context"* xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.0.xsd*

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

*http://www.springframework.org/schema/context/spring-context-3.0.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-3.0.xsd"*

default-lazy-init=*"true"* >

如果文件头配置无误，在文件头的beans标签内，加入如下配置：

<!-- 使用Annotation(注解)自动注册Bean(自注入) ,并检查@Required,@Autowired的属性已被注入 -->

<context:component-scan base-package=*"com.thinkdef.core.api.\*"*></context:component-scan>

此处的“base-package”可以配置多个命名空间，以“;”分隔。

此配置表明，系统会自动扫描*com.thinkdef.core.api*下的类，如果包含@Required、@Autowired注解，则自动获取代码中对应的依赖注入的接口，并完成接口对象的注入。

一般情况下，此处配置的命名空间中，放置的是需要进行逻辑处理的接口服务，其中会调用Service层的方法，完成相应的功能。

# 说明

## Jersey对JSON的支持

Jersey提供3种基本方式来使用JSON格式

无论使用何种方式，在原有包的基础上，都需要在客户端和服务端Maven配置文件中添加jersey-json包以支持JSON格式

<dependency>

<groupId>com.sun.jersey</groupId>

<artifactId>jersey-json</artifactId>

<version>1.18</version>

</dependency>

### 基于POJO

Request类和Response类（服务端和客户端都需要）都是基本的POJO：

**package** com.thinkdef.core.api;

**public** **class** Request {

**private** String query;

**public** String getQuery() {

**return** query;

}

**public** **void** setQuery(String query) {

**this**.query = query;

}

}

**package** com.thinkdef.core.api;

**public** **class** Response {

**private** **int** respCode;

**private** String respDesc;

**public** **int** getRespCode() {

**return** respCode;

}

**public** **void** setRespCode(**int** respCode) {

**this**.respCode = respCode;

}

**public** String getRespDesc() {

**return** respDesc;

}

**public** **void** setRespDesc(String respDesc) {

**this**.respDesc = respDesc;

}

}

服务端代码：

**package** com.thinkdef.core.api;

**import** java.io.IOException;

**import** java.net.URI;

**import** javax.ws.rs.Consumes;

**import** javax.ws.rs.POST;

**import** javax.ws.rs.Path;

**import** javax.ws.rs.Produces;

**import** javax.ws.rs.core.MediaType;

**import** javax.ws.rs.core.UriBuilder;

**import** org.glassfish.grizzly.http.server.HttpServer;

**import** com.sun.jersey.api.container.grizzly2.GrizzlyServerFactory;

**import** com.sun.jersey.api.core.PackagesResourceConfig;

**import** com.sun.jersey.api.core.ResourceConfig;

**import** com.sun.jersey.api.json.JSONConfiguration;

@Path("query")

**public** **class** MyResource {

@POST

@Consumes(MediaType.*APPLICATION\_JSON*)

@Produces(MediaType.*APPLICATION\_JSON*)

**public** Response query(Request req) {

System.*out*.println(req.getQuery());

Response resp = **new** Response();

resp.setRespCode(0);

resp.setRespDesc(req.getQuery());

**return** resp;

}

**public** **static** **void** main(String[] args) {

URI uri = UriBuilder.*fromUri*("http://127.0.0.1").port(10000).build();

ResourceConfig rc = **new** PackagesResourceConfig("com.thinkdef.core.api");

// 使用Jersey对POJO的支持，必须设置为true

rc.getFeatures().put(JSONConfiguration.FEATURE\_POJO\_MAPPING, **true**);

**try** {

HttpServer server = GrizzlyServerFactory.createHttpServer(uri, rc);

server.start();

} **catch** (IllegalArgumentException e) {

e.printStackTrace();

} **catch** (NullPointerException e) {

e.printStackTrace();

} **catch** (IOException e) {

e.printStackTrace();

}

**try** {

Thread.*sleep*(1000 \* 1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

客户端代码：

**package** com.thinkdef.core.api;

**import** javax.ws.rs.core.MediaType;

**import** com.sun.jersey.api.client.Client;

**import** com.sun.jersey.api.client.ClientResponse;

**import** com.sun.jersey.api.client.WebResource;

**import** com.sun.jersey.api.client.config.ClientConfig;

**import** com.sun.jersey.api.client.config.DefaultClientConfig;

**import** com.sun.jersey.api.json.JSONConfiguration;

**public** **class** JerseyClient {

**public** **static** **void** main(String[] args) {

ClientConfig cc = **new** DefaultClientConfig();

// 使用Jersey对POJO的支持，必须设置为true

cc.getFeatures().put(JSONConfiguration.FEATURE\_POJO\_MAPPING,

Boolean.*TRUE*);

Client client = Client.create(cc);

WebResource resource = client.resource("http://127.0.0.1:10000/query");

Request req = **new** Request();

req.setQuery("name");

ClientResponse response = resource.accept(MediaType.*APPLICATION\_JSON*)

.type(MediaType.*APPLICATION\_JSON*)

.post(ClientResponse.**class**, req);

Response resp = response.getEntity(Response.**class**);

System.*out*.println(resp.getRespCode() + " " + resp.getRespDesc());

}

}

### 基于JAXB

使用JAXB的优点在于，无论使用XML格式还是JSON格式数据，都可以使用统一的[Java](http://lib.csdn.net/base/java)模型

缺点很难找到一个合适的方式来生成特殊的JSON格式，这也是Jersey提供很多控制选项的原因

将Request类和Response类进行修改：

**package** com.thinkdef.core.api;

**import** javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement

**public** **class** Request {

**private** String query;

**public** String getQuery() {

**return** query;

}

**public** **void** setQuery(String query) {

**this**.query = query;

}

}

**package** com.thinkdef.core.api;

**import** javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement

**public** **class** Response {

**private** **int** respCode;

**private** String respDesc;

**public** **int** getRespCode() {

**return** respCode;

}

**public** **void** setRespCode(**int** respCode) {

**this**.respCode = respCode;

}

**public** String getRespDesc() {

**return** respDesc;

}

**public** **void** setRespDesc(String respDesc) {

**this**.respDesc = respDesc;

}

}

服务端代码去掉下面的配置

//   rc.getFeatures().put(JSONConfiguration.FEATURE\_POJO\_MAPPING, true);

客户端代码去掉下面的配置

//   cc.getFeatures().put(JSONConfiguration.FEATURE\_POJO\_MAPPING, Boolean.TRUE);

Jersey提供很多控制选项以便更精细的控制JSON的解析、组装过程，但是就我个人来看，JAXB提供的标签足够使用了

### 基于底层JSONObject/JSONArray

最大的优势在于可以完全控制JSON的解析、组装过程，相应的，在处理数据对象时也要更复杂

服务端代码如下：

**package** com.thinkdef.core.api;

**import** java.io.IOException;

**import** java.net.URI;

**import** javax.ws.rs.Consumes;

**import** javax.ws.rs.POST;

**import** javax.ws.rs.Path;

**import** javax.ws.rs.Produces;

**import** javax.ws.rs.core.MediaType;

**import** javax.ws.rs.core.UriBuilder;

**import** org.codehaus.jettison.json.JSONException;

**import** org.codehaus.jettison.json.JSONObject;

**import** org.glassfish.grizzly.http.server.HttpServer;

**import** com.sun.jersey.api.container.grizzly2.GrizzlyServerFactory;

**import** com.sun.jersey.api.core.PackagesResourceConfig;

**import** com.sun.jersey.api.core.ResourceConfig;

@Path("query")

**public** **class** MyResource {

@POST

@Consumes(MediaType.*APPLICATION\_JSON*)

@Produces(MediaType.*APPLICATION\_JSON*)

**public** JSONObject query(JSONObject query) {

// {"query":"name"}

System.*out*.println(query.toString());

JSONObject resp = **new** JSONObject();

**try** {

resp.put("respCode", 0);

resp.put("respDesc", query.get("query"));

} **catch** (JSONException e) {

e.printStackTrace();

}

**return** resp;

}

**public** **static** **void** main(String[] args) {

URI uri = UriBuilder.*fromUri*("http://127.0.0.1").port(10000).build();

ResourceConfig rc = **new** PackagesResourceConfig("com.thinkdef.core.api");

**try** {

HttpServer server = GrizzlyServerFactory.createHttpServer(uri, rc);

server.start();

} **catch** (IllegalArgumentException e) {

e.printStackTrace();

} **catch** (NullPointerException e) {

e.printStackTrace();

} **catch** (IOException e) {

e.printStackTrace();

}

**try** {

Thread.*sleep*(1000 \* 1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

客户端代码如下：

**package** com.thinkdef.core.api;

**import** javax.ws.rs.core.MediaType;

**import** org.codehaus.jettison.json.JSONException;

**import** org.codehaus.jettison.json.JSONObject;

**import** com.sun.jersey.api.client.Client;

**import** com.sun.jersey.api.client.ClientResponse;

**import** com.sun.jersey.api.client.WebResource;

**import** com.sun.jersey.api.client.config.ClientConfig;

**import** com.sun.jersey.api.client.config.DefaultClientConfig;

**public** **class** JerseyClient {

**public** **static** **void** main(String[] args) {

ClientConfig cc = **new** DefaultClientConfig();

Client client = Client.create(cc);

WebResource resource = client.resource("http://127.0.0.1:10000/query");

JSONObject req = **new** JSONObject();

**try** {

req.put("query", "name");

} **catch** (JSONException e) {

e.printStackTrace();

}

ClientResponse response = resource.accept(MediaType.*APPLICATION\_JSON*)

.type(MediaType.*APPLICATION\_JSON*)

.post(ClientResponse.**class**, req);

JSONObject resp = response.getEntity(JSONObject.**class**);

// {"respCode":0,"respDesc":"name"}

System.*out*.println(resp.toString());

}

}

与JAXB相比，结果是相同的，但是处理过程（主要是组装JSON对象）要复杂

### 总结

**对于上面3种方式，均可使用String类代替Request类、Response类或JSONObject类，Jersey会自动将对象转换为JSON串**

当然，如果客户端修改为String，服务端也要相应的修改为String类型

修改客户端代码：

**public** **class** JerseyClient {

**public** **static** **void** main(String[] args) {

ClientConfig cc = **new** DefaultClientConfig();

Client client = Client.create(cc);

WebResource resource = client.resource("http://127.0.0.1:10000/query");

JSONObject req = **new** JSONObject();

**try** {

req.put("query", "name");

} **catch** (JSONException e) {

e.printStackTrace();

}

String response = resource.accept(MediaType.APPLICATION\_JSON)

.type(MediaType.APPLICATION\_JSON)

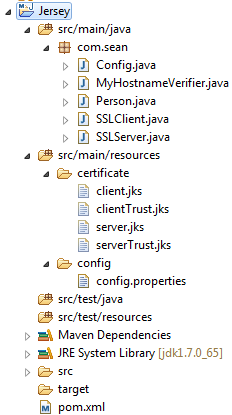
.post(String.**class**, req.toString());

}

}

## Jersey对HTTPS的支持

代码结构如下：



Maven配置文件：

<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>JERSEY</groupId>

<artifactId>JERSEY</artifactId>

<version>1.0</version>

<dependencies>

<dependency>

<groupId>com.sun.jersey</groupId>

<artifactId>jersey-client</artifactId>

<version>1.18</version>

</dependency>

<dependency>

<groupId>com.sun.jersey</groupId>

<artifactId>jersey-grizzly2</artifactId>

<version>1.18</version>

</dependency>

<dependency>

<groupId>com.sun.jersey</groupId>

<artifactId>jersey-json</artifactId>

<version>1.18</version>

</dependency>

<dependency>

<groupId>org.glassfish.jersey.media</groupId>

<artifactId>jersey-media-json-jackson</artifactId>

<version>2.15</version>

</dependency>

</dependencies>

</project>

Person类是基本的JAXB：

**package** com.thinkdef.core.api;

**import** java.util.List;

**import** javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement

**public** **class** Person {

**private** String name;

**private** List<String> addresses;

**public** Person() {

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** List<String> getAddresses() {

**return** addresses;

}

**public** **void** setAddresses(List<String> addresses) {

**this**.addresses = addresses;

}

}

客户端代码：

**package** com.thinkdef.core.api;

**import** java.net.URI;

**import** javax.net.ssl.SSLContext;

**import** javax.ws.rs.core.MediaType;

**import** javax.ws.rs.core.UriBuilder;

**import** org.glassfish.jersey.SslConfigurator;

**import** com.sun.jersey.api.client.Client;

**import** com.sun.jersey.api.client.ClientResponse;

**import** com.sun.jersey.api.client.WebResource;

**import** com.sun.jersey.api.client.config.ClientConfig;

**import** com.sun.jersey.api.client.config.DefaultClientConfig;

**import** com.sun.jersey.client.urlconnection.HTTPSProperties;

**public** **class** SSLClient {

**public** **static** **void** main(String[] args) {

**int** authType = Integer.*valueOf*(

Config.getConfig().getProperty("authority")).intValue();

SslConfigurator sslConfig = SslConfigurator.*newInstance*();

**if** (authType == 1) {

sslConfig

.trustStoreFile(

Config.getConfig().getProperty("clientTrustCer"))

.trustStorePassword(

Config.getConfig().getProperty("clientTrustCerPwd"));

} **else** **if** (authType == 2) {

sslConfig

.keyStoreFile(Config.getConfig().getProperty("clientCer"))

.keyStorePassword(

Config.getConfig().getProperty("clientCerPwd"))

.keyPassword(Config.getConfig().getProperty("clientKeyPwd"))

.trustStoreFile(

Config.getConfig().getProperty("clientTrustCer"))

.trustStorePassword(

Config.getConfig().getProperty("clientTrustCerPwd"));

}

sslConfig.securityProtocol(Config.getConfig().getProperty("protocol"));

SSLContext sslContext = sslConfig.createSSLContext();

ClientConfig cc = **new** DefaultClientConfig();

cc.getProperties().put(HTTPSProperties.PROPERTY\_HTTPS\_PROPERTIES,

**new** HTTPSProperties(**new** MyHostnameVerifier(), sslContext));

Client client = Client.create(cc);

URI uri = UriBuilder.*fromUri*("https://127.0.0.1/queryAddress")

.port(10000).build();

WebResource resource = client.resource(uri);

Person person = **new** Person();

person.setName("sean");

ClientResponse response = resource.accept(MediaType.*APPLICATION\_XML*)

.type(MediaType.*APPLICATION\_XML*)

.post(ClientResponse.**class**, person);

String addresses = response.getEntity(String.**class**);

System.*out*.println(addresses);

}

}

SSL握手过程中，会对请求IP或请求域名进行校验，如果在证书信息中无法找到相关请求IP或请求域名则会报错（javax[.NET](http://lib.csdn.net/base/dotnet).ssl.SSLHandshakeException: [Java](http://lib.csdn.net/base/java).security.cert.CertificateException: No subject alternative names present）

这里实现自己的校验逻辑（如果请求的IP为127.0.0.1或请求的域名为localhost，则直接通过校验）以覆盖默认逻辑

**package** com.thinkdef.core.api;

**import** javax.net.ssl.HostnameVerifier;

**import** javax.net.ssl.SSLSession;

**public** **class** MyHostnameVerifier **implements** HostnameVerifier {

@Override

**public** **boolean** verify(String hostname, SSLSession session) {

**if** ("127.0.0.1".equals(hostname) || "localhost".equals(hostname))

**return** **true**;

**else**

**return** **false**;

}

}

服务端代码：

**package** com.thinkdef.core.api;

**import** java.io.IOException;

**import** java.net.URI;

**import** java.util.ArrayList;

**import** java.util.List;

**import** javax.net.ssl.SSLContext;

**import** javax.ws.rs.Consumes;

**import** javax.ws.rs.POST;

**import** javax.ws.rs.Path;

**import** javax.ws.rs.Produces;

**import** javax.ws.rs.core.MediaType;

**import** javax.ws.rs.core.UriBuilder;

**import** org.glassfish.grizzly.http.server.HttpHandler;

**import** org.glassfish.grizzly.http.server.HttpServer;

**import** org.glassfish.grizzly.ssl.SSLEngineConfigurator;

**import** org.glassfish.jersey.SslConfigurator;

**import** com.sun.jersey.api.container.ContainerFactory;

**import** com.sun.jersey.api.container.grizzly2.GrizzlyServerFactory;

**import** com.sun.jersey.api.core.PackagesResourceConfig;

**import** com.sun.jersey.api.core.ResourceConfig;

@Path("queryAddress")

**public** **class** SSLServer {

@POST

@Consumes(MediaType.*APPLICATION\_XML*)

@Produces(MediaType.*APPLICATION\_XML*)

**public** Person queryAddress(String name) {

System.*out*.println(name);

Person person = **new** Person();

List<String> addresses = **new** ArrayList<String>();

addresses.add("address1");

addresses.add("address2");

person.setAddresses(addresses);

**return** person;

}

**public** **static** **void** main(String[] args) {

Integer authType = Integer.*valueOf*(

Config.getConfig().getProperty("authority")).intValue();

SslConfigurator sslConfig = SslConfigurator.*newInstance*();

**if** (authType == 1) {

sslConfig

.keyStoreFile(Config.getConfig().getProperty("serverCer"))

.keyStorePassword(

Config.getConfig().getProperty("serverCerPwd"))

.keyPassword(Config.getConfig().getProperty("serverKeyPwd"));

} **else** **if** (authType == 2) {

sslConfig

.keyStoreFile(Config.getConfig().getProperty("serverCer"))

.keyStorePassword(

Config.getConfig().getProperty("serverCerPwd"))

.keyPassword(Config.getConfig().getProperty("serverKeyPwd"))

.trustStoreFile(

Config.getConfig().getProperty("serverTrustCer"))

.trustStorePassword(

Config.getConfig().getProperty("serverTrustCerPwd"));

}

sslConfig.securityProtocol(Config.getConfig().getProperty("protocol"));

SSLContext sslContext = sslConfig.createSSLContext();

SSLEngineConfigurator sslEngineConfig = **new** SSLEngineConfigurator(

sslContext);

// 默认情况下是客户端模式，如果忘记修改模式

// 会抛出异常

// javax.net.ssl.SSLProtocolException: Handshake message sequence

// violation, 1]

sslEngineConfig.setClientMode(**false**);

**if** (authType == 1)

sslEngineConfig.setWantClientAuth(**true**);

**else** **if** (authType == 2)

sslEngineConfig.setNeedClientAuth(**true**);

ResourceConfig rc = **new** PackagesResourceConfig("com.thinkdef.core.api");

HttpHandler handler = ContainerFactory.createContainer(

HttpHandler.**class**, rc);

URI uri = UriBuilder.*fromUri*("https://127.0.0.1/").port(10000).build();

**try** {

HttpServer server = GrizzlyServerFactory.createHttpServer(uri,

handler, **true**, sslEngineConfig);

server.start();

} **catch** (IllegalArgumentException e) {

e.printStackTrace();

} **catch** (NullPointerException e) {

e.printStackTrace();

} **catch** (IOException e) {

e.printStackTrace();

}

**try** {

Thread.*sleep*(1000 \* 1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

配置文件类：

**package** com.thinkdef.core.api;

**import** java.io.File;

**import** java.io.FileInputStream;

**import** java.io.InputStream;

**import** java.util.Properties;

**public** **class** Config {

**private** **static** Properties *config*;

**public** **static** Properties getConfig() {

**try** {

**if** (**null** == *config*) {

File configFile = **new** File(

"src/main/resources/config/config.properties");

**if** (configFile.exists() && configFile.isFile()

&& configFile.canRead()) {

InputStream input = **new** FileInputStream(configFile);

*config* = **new** Properties();

*config*.load(input);

}

}

} **catch** (Exception e) {

// default set

*config* = **new** Properties();

*config*.setProperty("authority", String.*valueOf*(1));

*config*.setProperty("protocol", "SSL");

*config*.setProperty("serverCer",

"src/main/resources/certificate/server.jks");

*config*.setProperty("serverCerPwd", "1234sp");

*config*.setProperty("serverKeyPwd", "1234kp");

*config*.setProperty("serverTrustCer",

"src/main/resources/certificate/serverTrust.jks");

*config*.setProperty("serverTrustCerPwd", "1234sp");

*config*.setProperty("clientCer",

"src/main/resources/certificate/client.jks");

*config*.setProperty("clientCerPwd", "1234sp");

*config*.setProperty("clientKeyPwd", "1234kp");

*config*.setProperty("clientTrustCer",

"src/main/resources/certificate/clientTrust.jks");

*config*.setProperty("clientTrustCerPwd", "1234sp");

}

**return** *config*;

}

}

配置文件config.properties：

#1：单向认证，只有服务器端需证明其身份

#2：双向认证，服务器端和客户端都需证明其身份

authority=2

#通信协议

protocol=SSL

#服务端证书信息

serverCer=src/main/resources/certificate/server.jks

#keystore的storepass

serverCerPwd=1234sp

#keystore的keypass

serverKeyPwd=1234kp

#服务端证书信息

serverTrustCer=src/main/resources/certificate/serverTrust.jks

serverTrustCerPwd=1234sp

#客户端证书信息

clientCer=src/main/resources/certificate/client.jks

clientCerPwd=1234sp

clientKeyPwd=1234kp

clientTrustCer=src/main/resources/certificate/clientTrust.jks

clientTrustCerPwd=1234sp

服务端运行结果：

三月 03, 2015 3:30:54 下午 com.sun.jersey.api.core.PackagesResourceConfig init

INFO: Scanning for root resource and provider classes in the packages:

  com.thinkdef.core.api

三月 03, 2015 3:30:54 下午 com.sun.jersey.api.core.ScanningResourceConfig logClasses

INFO: Root resource classes found:

  class com.thinkdef.core.api.SSLServer

三月 03, 2015 3:30:54 下午 com.sun.jersey.api.core.ScanningResourceConfig init

INFO: No provider classes found.

三月 03, 2015 3:30:54 下午 com.sun.jersey.server.impl.application.WebApplicationImpl \_initiate

INFO: Initiating Jersey application, version 'Jersey: 1.18 11/22/2013 01:21 AM'

三月 03, 2015 3:30:55 下午 org.glassfish.grizzly.http.server.NetworkListener start

INFO: Started listener bound to [127.0.0.1:10000]

三月 03, 2015 3:30:55 下午 org.glassfish.grizzly.http.server.HttpServer start

INFO: [HttpServer] Started.

<?xml version="1.0" encoding="UTF-8" standalone="yes"?><person><name>sean</name></person>

客户端运行结果

<?xml version="1.0" encoding="UTF-8" standalone="yes"?><person><addresses>address1</addresses><addresses>address2</addresses></person>

# 示例代码

本例，模仿对用户进行增删改查的操作。以下代码均以此为例。

## Jersey常用注解解释

|  |  |  |
| --- | --- | --- |
| **Annotation** | **作用** | **说明** |
| @GET | 查询请求 | 相当于数据库的查询数据操作 |
| @POST | 插入请求 | 相当于数据库的插入数据操作 |
| @PUT | 更新请求 | 相当于数据库的更新数据操作 |
| @DELETE | 删除请求 | 相当于数据的删除数据操作 |
| @Path | uri路径 | 定义资源的访问路径，client通过这个路径访问资源。比如：@Path("user") |
| @Produces | 指定返回MIME格式 | 资源按照那种数据格式返回，可取的值有：MediaType.APPLICATION\_XXX。比如：@Produces(MediaType.APPLICATION\_XML) |
| @Consumes | 接受指定的MIME格式 | 只有符合这个参数设置的请求再能访问到这个资源。比如@Consumes("application/x-www-form-urlencoded") |
| @PathParam | uri路径参数 | 写在方法的参数中，获得请求路径参数。比如：@PathParam("username")  String userName |
| @QueryParam | uri路径请求参数 | 写在方法的参数中，获得请求路径附带的参数。比如：@QueryParam("desc")  String desc |
| @DefaultValue | 设置@QueryParam参数的默认值 | 如果@QueryParam没有接收到值，就使用默认值。比如：@DefaultValue("description") @QueryParam("desc") String desc |
| @FormParam | form传递的参数 | 接受form传递过来的参数。比如：@FormParam("name")  String userName |
| @BeanParam | 通过Bena的形式传递参数 | 接受client传递的bean类型的参数，同时这个bean可以在属性上配置@FormParam用以解决client的属性名称和bean的属性名称不一致的问题。比如：@BeanParam  User user |
| @Context | 获得一些系统环境信息 | 通过@Context可以获得以下信息：UriInfo、ServletConfig、ServletContext、HttpServletRequest、HttpServletResponse和HttpHeaders等 |
| @XmlRootElement | 将bean转换为xml | 如果要讲bean以xml或json的格式返回，必须要这个注解。比如：  @XmlRootElement  public class User｛...｝ |
| @XmlElements |  |  |
| @XmlElement |  |  |

### @Path

用来为资源类或方法定义URI，当然除了静态URI也支持动态URI。

@Path("service")

**public** **class** MyResource {

@Path("{sub\_path}")

@GET

**public** String getResource(@PathParam("sub\_path") String resourceName) {

......

如果此时客户端请求的URI为http://127.0.0.1:10000/service/sean，则sub\_path的值为sean。

### @PathParam

用来将请求URI的一部分作为方法参数传入方法中。对URI的动态部分，可以自定义校验正则表达式，如果请求参数校验失败，容器返回404 Not Found。

@Path("{sub\_path:[A-Z]\*}")

### @GET

表明被注解的方法响应HTTP GET请求，**@POST**、**@PUT**和**@DELETE**同理。

### @Consumes

定义请求的媒体类型，如果不指定，则容器默认可接受任意媒体类型，容器负责确认被调用的方法可接受HTTP请求的媒体类型，否则返回415 Unsupported Media Type。方法级注解将覆盖类级注解。

### @Produces

定义响应媒体类型，如果不指定，则容器默认可接受任意媒体类型，容器负责确认被调用的方法可返回HTTP请求可以接受媒体类型，否则返回406 Not Acceptable。方法级注解将覆盖类级注解。

### @QueryParam

**public** String getResource(

@DefaultValue("Just a test!") @QueryParam("desc") String description) {

......

}

如果请求URI中包含desc参数，例如：http://127.0.0.1:10000/service/sean?desc=123456，则desc参数的值将会赋给方法的参数description，否则方法参数description的值将为**@DefaultValue**注解定义的默认值。

### @Context

将信息注入请求或响应相关的类，可注入的类有：Application，UriInfo，Request，HttpHeaders和SecurityContext。

### @Singleton和@PerRequest

默认情况下，资源类的生命周期是per-request，也就是系统会为每个匹配资源类URI的请求创建一个实例，这样的效率很低，可以对资源类使用**@Singleton**注解，这样在应用范围内，只会创建资源类的一个实例。

## 资源加载器代码

以下代码中，过滤器、数据转换器、日志，一定要配置！否则后面调用的时候会很麻烦。

register(Login.class)为注册自己的接口服务类。可以根据需要注册任意多个。

**package** com.thinkdef.core.api;

**import** org.glassfish.jersey.filter.LoggingFilter;

**import** org.glassfish.jersey.server.ResourceConfig;

**import** com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider;

**import** com.thinkdef.core.api.authorization.token.TokenFilter;

**import** com.thinkdef.core.api.resource.Login;

**public** **class** ApplicationAPI **extends** ResourceConfig{

**public** ApplicationAPI(){

// 注册过滤器

register(RestWebServiceInterceptor.**class**);

// 注册数据转换器

register(JacksonJsonProvider.**class**);

// 注册日志

register(LoggingFilter.**class**);

register(Login.**class**);

}

}

## 值对象代码

本例中，值对象为“用户”对象，包含用户必须的属性字段。

**package** com.thinkdef.core.api.demo;

**import** javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement

**public** **class** User {

**private** Integer id;

**private** String name;

**private** String password;

**public** String getPassword() {

**return** password;

}

**public** **void** setPassword(String password) {

**this**.password = password;

}

**private** **int** age;

**public** User(){

}

**public** User(String id, String name, String age){

**this**.id=Integer.*parseInt*(id);

**this**.name=name;

**this**.age=Integer.*parseInt*(age);

}

**public** Integer getId() {

**return** id;

}

**public** **void** setId(Integer id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

}

## 基础接口服务代码

以下代码，通过Jersey直接调用对象默认的方法，模拟业务处理过程。

**package** com.thinkdef.core.api.demo;

**import** javax.ws.rs.Consumes;

**import** javax.ws.rs.GET;

**import** javax.ws.rs.POST;

**import** javax.ws.rs.Path;

**import** javax.ws.rs.PathParam;

**import** javax.ws.rs.Produces;

**import** javax.ws.rs.QueryParam;

**import** javax.ws.rs.core.MediaType;

@Path("/demo")

**public** **class** RestfulDemo {

/\*\*

\* 访问http://localhost:8080/api/demo，页面打出hello world.

\* 访问http://localhost:8080/api/demo/zhangsan ,页面打出hello zhangsan

\* 访问http://localhost:8080/api/demo/get?username=zhangsan

\* ,页面返回User对象的json格式

\* 访问http://localhost:8080/api/demo/update，post提交user对象的json格式

\* ，返回更新user对象的json格式。

\*/

@GET

@Produces(MediaType.*TEXT\_PLAIN*)

**public** String sayHello() {

**return** "Hello World!";

}

@GET

@Path("/{param}")

@Produces("text/plain;charset=UTF-8")

**public** String sayHelloToUTF8(@PathParam("param") String username) {

**return** "Hello " + username;

}

@GET

@Path("/get")

@Produces(MediaType.*APPLICATION\_JSON*)

**public** User sayHelloToJson(@QueryParam("username") String username) {

User user = **new** User();

user.setId(1);

user.setName(username);

**return** user;

}

@POST

@Path("/update")

@Consumes(MediaType.*APPLICATION\_JSON*)

@Produces(MediaType.*APPLICATION\_JSON*)

**public** User updateUser(User user) {

user.setId(2);

user.setName("update name:" + user.getName());

**return** user;

}

}

## 与Spring集成的接口服务代码

以下代码，通过Jersey的注解加入与Spring集成后注入的Service，可以调用系统中已有的API。

**package** com.thinkdef.core.api.resource;

**import** java.text.SimpleDateFormat;

**import** javax.ws.rs.Consumes;

**import** javax.ws.rs.GET;

**import** javax.ws.rs.POST;

**import** javax.ws.rs.Path;

**import** javax.ws.rs.Produces;

**import** javax.ws.rs.core.MediaType;

**import** net.sf.json.JSONObject;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Component;

**import** com.thinkdef.core.api.authorization.token.JwtUtils;

**import** com.upsoft.common.utils.CommonTool;

**import** com.upsoft.common.utils.EncryptUtil;

**import** com.upsoft.interfaces.core.system.IUserService;

**import** com.upsoft.upcore.dto.User;

@Component

@Path("/login")

**public** **class** Login {

//注入用户Service

@Autowired

**private** IUserService userService;

@GET

@Produces(MediaType.*TEXT\_PLAIN*)

**public** String sayHello() {

**return** "This is a Login Check API!";

}

@SuppressWarnings("unchecked")

@POST

@Path("/authentication")

@Consumes(MediaType.*APPLICATION\_JSON*)

@Produces(MediaType.*APPLICATION\_JSON*)

**public** JSONObject checkUser(User userInput){

JSONObject joReturn= **new** JSONObject();

String strUserName=userInput.getUser\_name();

String strPassWord=userInput.getPass\_word();

//替换掉特殊字符

strUserName=CommonTool.*transErrorStr*(strUserName);

**if**(strUserName!=**null**&&!"".equals(strUserName)){

strPassWord=CommonTool.*transErrorStr*(strPassWord);

**if**(strPassWord!=**null**&&!"".equals(strPassWord)){

**try** {

strPassWord = EncryptUtil.*EncoderByMd5*(strPassWord);

} **catch** (Exception e) {

e.printStackTrace();

}

}

//将登录名和进行MD5加密之后的密码传入数据库进行查找，是否有符合条件的记录。

user = userService.checkLogin(strUserName, strPassWord);

//如果有符合条件的记录，即用户输入的登录名和密码是对应的，合法的。

**if** (user != **null**) {

joReturn.put("flag", "success");

joReturn.put("message", "登录成功");

joReturn.put("user\_name", user.getUser\_name());

}

}**else**{

joReturn.put("flag", "fail");

joReturn.put("message", "登录失败");

}

**return** joReturn;

}

}

本例中，通过“@Consumes(MediaType.APPLICATION\_JSON)”注解，定义接收参数为json，调用时可以传入符合接口对象定义的json串（包含user对象字段的值），系统会自动解析并拼装为对应的user对象，供代码获取、处理。

同理，通过“@Produces(MediaType.APPLICATION\_JSON)”注解，定义返回参数为json，返回的数据对象，Jersey会自动将其处理为对应格式的json串。

## 调用接口

以“基础接口服务代码”为例：

* 访问http://localhost:8080/api/demo，页面打出hello world.
* 访问http://localhost:8080/api/demo/zhangsan ,页面打出hello zhangsan
* 访问http://localhost:8080/api/demo/get?username=zhangsan ,页面返回User对象的json格式
* 访问http://localhost:8080/api/demo/update，post提交user对象的json格式，返回更新user对象的json格式。

## Clien代码

Jersey还提供了一个客户端库，帮助您与服务器通讯并对 RESTful 服务进行单元测试。该库是一个一般实现，可以整合任何 HTTP/HTTPS-based Web 服务。

**注意：此Client包最高支持1.19.4，V2及以上版本已经无此类和方法！！**

客户端的核心类是 WebResource 类。您可以使用该类根据根 URI 构建一个请求 URL，然后发送请求并获取响应。以下代码展示了如何创建 WebResource 实例。注意 WebResource 是一个大对象，因此只创建一次。

**package** com.thinkdef.core.api.demo;

**import** javax.ws.rs.core.MediaType;

**import** com.sun.jersey.api.client.Client;

**import** com.sun.jersey.api.client.WebResource;

**public** **class** ContactClient {

**public** **static** **void** main(String[] args) {

Client c = Client.*create*();

WebResource r = c.resource("http://localhost:8080/api/login");

String jsonRes = r.accept(MediaType.*APPLICATION\_JSON*).get(String.**class**);

System.*out*.println(jsonRes);

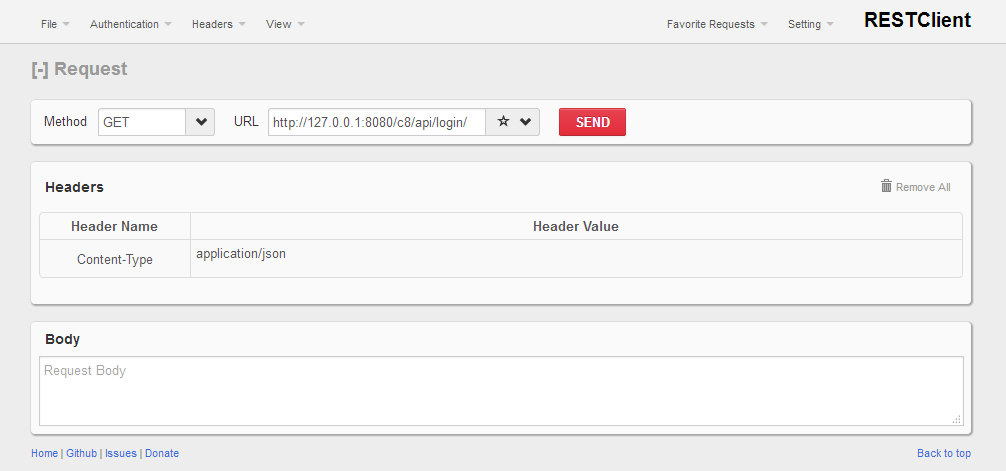
}

}

# 测试

因为RESTful的接口一般都通过HTTP协议访问，所以我们可以直接使用浏览器访问接口。

为了更加方便的对接口进行测试，可以使用Firefox浏览器，安装“RESTClient”插件，对接口进行测试。



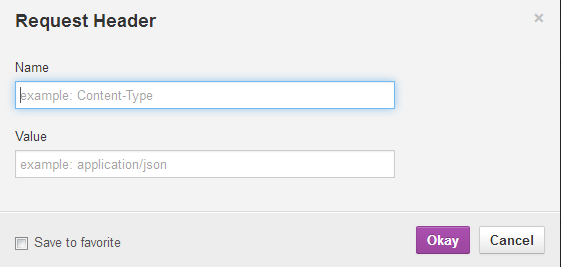
以下，对此工具的使用进行分解说明。

选择请求的方式，GET、POST、PUT、DELETE等。

接口的访问地址。

如果传入的参数是json或xml，需要在请求头中设置数据类型。

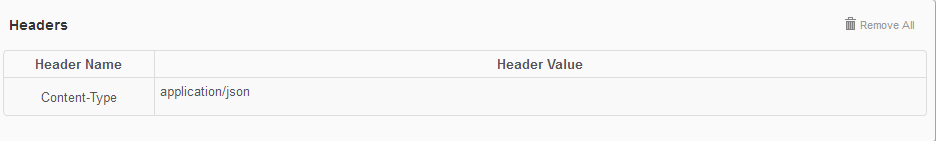
点击“headers”-“Custom Headers”菜单，系统显示如下界面。



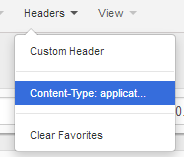
Name输入请求头名称。本例中输入默认的“Content-Type”。

Value输入请求的数据类型。本例中根据需要，输入为“application/json”。

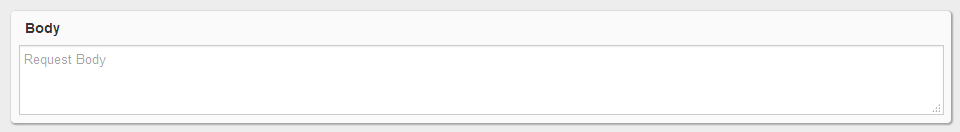
输入完毕，点击“Okay”按钮，插件即将设置保存。可以在请求头列表中看到我们添加的内容。



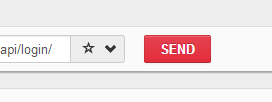
同时，在“Headers”菜单中也保存了此请求头。



在“body”文本框为请求内容，可以输入需要传入的参数值。如果需要传入的是json或xml，可以直接输入在这个文本框中。



全部设置、输入完毕后，点击“SEND”按钮，插件即会自动访问对应接口，并接收返回的信息。



# RESTful API 设计指南

## 协议

API与用户的通信协议，总是使用[HTTPs协议](http://www.ruanyifeng.com/blog/2014/02/ssl_tls.html)。

**建议：推荐采用。**

因为REST风格的接口通过请求直接将数据通过URL或者POST方式传递给服务端，如果只是是使用HTTP协议，在传输过程中的安全性较低。采用HTTPs协议，使用SSL对传输内容进行加密，可以保证在传输过程中的安全性。

## 域名

应该尽量将API部署在专用域名之下。

**https://api.example.com**

如果确定API很简单，不会有进一步扩展，可以考虑放在主域名下。

**https://www.example.com/api/**

**建议：采用。**

## 版本（Versioning）

应该将API的版本号放入URL。

**https://api.example.com/v1/**

或

**https://www.example.com/api/v1/**

另一种做法是，将版本号放在HTTP头信息中，但不如放入URL方便和直观。[Github](https://developer.github.com/v3/media/#request-specific-version)采用这种做法。

**建议：采用。**

此处的“版本”是API的版本，非软件产品的版本。由于RESTful API一般为互联网类线上系统使用，调用API的前端页面、异构系统有很多，在做API升级的时候必须考虑到已有功能、集成接口的兼容性。在URL中加入API版本编号的方式，对不同版本的接口加以区分，可以有效的防止接口升级时带来的不稳定。可以让老功能用老API，新开发用新API。

## 路径（Endpoint）

路径又称"终点"（endpoint），表示API的具体网址。

在RESTful架构中，每个网址代表一种资源（resource），所以网址中不能有动词，只能有名词，而且所用的名词往往与数据库的表格名对应。一般来说，数据库中的表都是同种记录的"集合"（collection），所以API中的名词也应该使用复数。

举例来说，有一个API提供动物园（zoo）的信息，还包括各种动物和雇员的信息，则它的路径应该设计成下面这样。

* https://api.example.com/v1/zoos
* https://api.example.com/v1/animals
* https://api.example.com/v1/employees

**建议：部分采用。**

由于标准RESTful请求传值的局限性，采用“终点=名词”的方式设计API会导致后续很多问题。

所以，建议采用“路径名=名词”的方式，即资源的名称为API URL中的路径名，在路径名之后再创建具体的动作对应的API。

例如：https://api.example.com/v1/animals/getAnimalsInfo

## HTTP动词

对于资源的具体操作类型，由HTTP动词表示。

常用的HTTP动词有下面五个（括号里是对应的SQL命令）。

* GET（SELECT）：从服务器取出资源（一项或多项）。
* POST（CREATE）：在服务器新建一个资源。
* PUT（UPDATE）：在服务器更新资源（客户端提供改变后的完整资源）。
* PATCH（UPDATE）：在服务器更新资源（客户端提供改变的属性）。
* DELETE（DELETE）：从服务器删除资源。

还有两个不常用的HTTP动词。

* HEAD：获取资源的元数据。
* OPTIONS：获取信息，关于资源的哪些属性是客户端可以改变的。

下面是一些例子。

* GET /zoos：列出所有动物园
* POST /zoos：新建一个动物园
* GET /zoos/ID：获取某个指定动物园的信息
* PUT /zoos/ID：更新某个指定动物园的信息（提供该动物园的全部信息）
* PATCH /zoos/ID：更新某个指定动物园的信息（提供该动物园的部分信息）
* DELETE /zoos/ID：删除某个动物园
* GET /zoos/ID/animals：列出某个指定动物园的所有动物
* DELETE /zoos/ID/animals/ID：删除某个指定动物园的指定动物

**建议：不采用。**

由于标准RESTful请求传值的局限性，需要在URL中拼装各类参数。如果遇到复杂条件会遇到URL超长的问题；中文字符、特殊字符会遇到乱码的问题。

所以，建议全部接口采用**POST**方式调用和传值。

## 过滤信息（Filtering）

如果记录数量很多，服务器不可能都将它们返回给用户。API应该提供参数，过滤返回结果。

下面是一些常见的参数。

* ?limit=10：指定返回记录的数量
* ?offset=10：指定返回记录的开始位置。
* ?page=2&per\_page=100：指定第几页，以及每页的记录数。
* ?sortby=name&order=asc：指定返回结果按照哪个属性排序，以及排序顺序。
* ?animal\_type\_id=1：指定筛选条件

参数的设计允许存在冗余，即允许API路径和URL参数偶尔有重复。比如，GET /zoo/ID/animals 与 GET /animals?zoo\_id=ID 的含义是相同的。

**建议：不采用。**

由于此种方式传值，需要在URL中拼装各类参数。如果遇到复杂条件会遇到URL超长的问题；中文字符、特殊字符会遇到乱码的问题。而且将各类操作参数放在URL中进行显性传值并不安全。

所以，建议全部接口采用**POST**方式调用接口，将需要传递的参数值拼装为JSON放置在BODY中进行提交。

例如：

**{**

**"limit": "10",**

**"page": "2",**

**"pre\_page": "100",**

**"sortby ": "name",**

**"order": "asc"**

**}**

## 状态码（Status Codes）

服务器向用户返回的状态码和提示信息，常见的有以下一些（方括号中是该状态码对应的HTTP动词）。

* 200 OK - [GET]：服务器成功返回用户请求的数据，该操作是幂等的（Idempotent）。
* 201 CREATED - [POST/PUT/PATCH]：用户新建或修改数据成功。
* 202 Accepted - [\*]：表示一个请求已经进入后台排队（异步任务）
* 204 NO CONTENT - [DELETE]：用户删除数据成功。
* 400 INVALID REQUEST - [POST/PUT/PATCH]：用户发出的请求有错误，服务器没有进行新建或修改数据的操作，该操作是幂等的。
* 401 Unauthorized - [\*]：表示用户没有权限（令牌、用户名、密码错误）。
* 403 Forbidden - [\*] 表示用户得到授权（与401错误相对），但是访问是被禁止的。
* 404 NOT FOUND - [\*]：用户发出的请求针对的是不存在的记录，服务器没有进行操作，该操作是幂等的。
* 406 Not Acceptable - [GET]：用户请求的格式不可得（比如用户请求JSON格式，但是只有XML格式）。
* 410 Gone -[GET]：用户请求的资源被永久删除，且不会再得到的。
* 422 Unprocesable entity - [POST/PUT/PATCH] 当创建一个对象时，发生一个验证错误。
* 500 INTERNAL SERVER ERROR - [\*]：服务器发生错误，用户将无法判断发出的请求是否成功。
* 状态码的完全列表参见[这里](http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html)。

**建议：需掌握。**

## 错误处理（Error handling）

如果状态码是4xx，就应该向用户返回出错信息。一般来说，返回的信息中将error作为键名，出错信息作为键值即可。

**{**

**"error": "Invalid API key"**

**}**

**建议：不推荐。**

此种方式只能处理错误信息，其他类型的状态难以兼顾。

建议采用一个值标记返回状态，另一个值表示返回信息的方式，可以兼容多种情况。

例如：

**{**

**"flag": "ERROR",**

**"message": "Invalid API key"**

**}**

## 返回结果

针对不同操作，服务器向用户返回的结果应该符合以下规范。

* GET /collection：返回资源对象的列表（数组）
* GET /collection/resource：返回单个资源对象
* POST /collection：返回新生成的资源对象
* PUT /collection/resource：返回完整的资源对象
* PATCH /collection/resource：返回完整的资源对象
* DELETE /collection/resource：返回一个空文档

**建议：不采用。**

建议全部接口采用**POST**方式调用接口，将需要传递的参数值拼装为JSON放置在BODY中进行提交，接口返回相应的请求结果。

## Hypermedia API

RESTful API最好做到Hypermedia，即返回结果中提供链接，连向其他API方法，使得用户不查文档，也知道下一步应该做什么。

比如，当用户向api.example.com的根目录发出请求，会得到这样一个文档。

**{"link": {**

**"rel": "collection** [**https://www.example.com/zoos**](https://www.example.com/zoos)**",**

**"href": "**[**https://api.example.com/zoos**](https://api.example.com/zoos)**",**

**"title": "List of zoos",**

**"type": "application/vnd.yourformat+json"**

**}}**

上面代码表示，文档中有一个link属性，用户读取这个属性就知道下一步该调用什么API了。rel表示这个API与当前网址的关系（collection关系，并给出该collection的网址），href表示API的路径，title表示API的标题，type表示返回类型。

Hypermedia API的设计被称为[HATEOAS](http://en.wikipedia.org/wiki/HATEOAS)。Github的API就是这种设计，访问[api.github.com](https://api.github.com/)会得到一个所有可用API的网址列表。

**{**

**"current\_user\_url": "**[**https://api.github.com/user**](https://api.github.com/user)**",**

**"authorizations\_url": "**[**https://api.github.com/authorizations**](https://api.github.com/authorizations)**",**

**// ...**

**}**

从上面可以看到，如果想获取当前用户的信息，应该去访问[api.github.com/user](https://api.github.com/user)，然后就得到了下面结果。

**{**

**"message": "Requires authentication",**

**"documentation\_url": "**[**https://developer.github.com/v3**](https://developer.github.com/v3)**"**

**}**

上面代码表示，服务器给出了提示信息，以及文档的网址。

**建议：推荐遵守。**

## 其他

（1）API的身份认证应该使用[OAuth 2.0](http://www.ruanyifeng.com/blog/2014/05/oauth_2_0.html)框架。

（2）服务器返回的数据格式，应该尽量使用JSON，避免使用XML。

**建议：推荐遵守。**