**Spring MVC与JAX-RS比较与分析**

****

**导言**

过去几年，REST逐渐成为影响Web框架、Web协议与Web应用设计的重要概念。如果你还不了解REST，那[这个简短的介绍](http://www.infoq.com/articles/rest-introduction)将有助你快速掌握REST，此外还可以点击[这里](http://www.infoq.com/rest)了解关于REST的更多信息。

现在有越来越多的公司希望能以简单而又贴合Web架构本身的方式公开Web API，因此REST变得越来越重要也就不足为奇了。使用Ajax进行通信的富浏览器端也在朝这个目标不断迈进。这个架构原则提升了万维网的可伸缩性，无论何种应用都能从该原则中受益无穷。

[JAX-RS](http://jcp.org/en/jsr/detail?id=311)（JSR 311）指的是Java API for RESTful Web Services，[Roy Fielding](http://www.ics.uci.edu/%7Efielding/)也参与了JAX-RS的制订，他在自己的博士[论文](http://www.ics.uci.edu/%7Efielding/pubs/dissertation/top.htm)中定义了REST。对于那些想要构建RESTful Web Services的开发者来说，JAX-RS给出了不同于JAX-WS（JSR-224）的另一种解决方案。目前共有4种JAX-RS实现，所有这些实现都支持Spring，[Jersey](http://wikis.sun.com/display/Jersey/Main)则是JAX-RS的参考实现，也是本文所用的实现。

如果你使用Spring进行开发，那可能想知道（或者有人曾问过你）Spring MVC与JAX-RS有何异同点？更进一步，如果你手头有一个Spring MVC应用，使用了控制类继承（SimpleFormController等），你可能还意识不到现在的Spring MVC对REST广泛的支持。

本文将介绍Spring 3中的REST特性并与JAX-RS进行对比，希望能帮助你理顺这两种编程模型之间的异同点。

开始前，有必要指出JAX-RS的目标是Web Services开发（这与HTML Web应用不同）而Spring MVC的目标则是Web应用开发。Spring 3为Web应用与Web Services增加了广泛的REST支持，但本文则关注于与Web Services开发相关的特性。我觉得这种方式更有助于在JAX-RS的上下文中讨论Spring MVC。

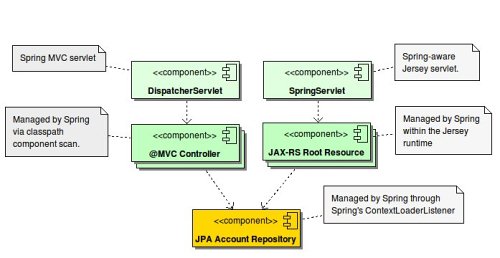
要说明的第二点是我们将要讨论的REST特性是Spring Framework的一部分，也是现有的Spring MVC编程模型的延续，因此，并没有所谓的“Spring REST framework”这种概念，有的只是Spring和Spring MVC。这意味着如果你有一个Spring应用的话，你既可以使用Spring MVC创建HTML Web层，也可以创建RESTful Web Services层。

**关于文中的代码片段**

文中的代码片段假想了一个简单的领域模型：两个JPA注解实体，分别是Account和Portfolio，其中一个Account对应多个Portfolio。持久层使用Spring配置，包含了一个JPA仓储实现，用于获取和持久化实体实例。Jersey和Spring MVC用于构建Web Services层，通过调用底层的Spring托管应用来服务客户端请求。

**引导程序与Web层包装**

我们会在Spring MVC和JAX-RS中都使用Spring实现依赖注入。Spring MVC DispatcherServlet和Jersey SpringServlet会把请求代理给Spring管理的REST层组件（控制器或资源），后者会由业务或持久层组件包装起来，如下图所示：



Jersey和Spring MVC都使用Spring的ContextLoaderListener加载业务与持久层组件，比如JpaAccountRepository：

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>

classpath:META-INF/spring/module-config.xml

</param-value>

</context-param>

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

ContextLoaderListener可用于任何Web或REST框架环境中。

**在Jersey中创建Spring管理的JAX-RS资源**

Jersey支持在REST层中使用Spring，两个简单的步骤就能搞定（事实上有3步，还需要将构建依赖加到maven artifact com.sun.jersey.contribs:jersey-spring中）。

步骤一：将如下配置片段加到web.xml中以保证Spring能够创建JAX-RS根资源：

<servlet>

<servlet-name>Jersey Web Application</servlet-name>

<servlet-class>

com.sun.jersey.spi.spring.container.servlet.SpringServlet

</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>Jersey Web Application</servlet-name>

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

</servlet-mapping>

步骤二：使用Spring和JAX-RS注解声明根JAX-RS资源类：

@Path("/accounts/")

@Component

@Scope("prototype")

public class AccountResource {

@Context

UriInfo uriInfo;

@Autowired

private AccountRepository accountRepository;

}

如下是对这些注解的说明：

@Component将AccountResource声明为Spring bean。

@Scope声明了一个prototype Spring bean，这样每次使用时都会实例化（比如每次请求时）。

@Autowired指定了一个AccountRepository引用，Spring会提供该引用。

@Path是个JAX-RS注解，它将AccountResource声明为“根”JAX-RS资源。

@Context也是一个JAX-RS注解，要求注入特定于请求的UriInfo对象。

JAX-RS有“根”资源（标记为@Path）和子资源的概念。在上面的示例中，AccountResource就是个根资源，它会处理以“/accounts/”开头的路径。AccountResource中的方法如getAccount()只需声明针对类型级别的相对路径即可。

@Path("/accounts/")

@Component

@Scope("prototype")

public class AccountResource {

@GET

@Path("{username}")

public Account getAccount(@PathParam("username") String username) {

}

}

访问路径“/accounts/{username}”（其中的username是路径参数，可以是某个账户的用户名）的请求将由getAccount()方法处理。

根资源由JAX-RS运行时（在本示例中是Spring）实例化，子资源则由应用本身实例化。比如说，对于“/accounts/{username}/portfolios/{portfolioName}”这样的请求，AccountResource（由路径的第一部分“/accounts”标识）会创建一个子资源实例，请求会被代理给该实例：

@Path("/accounts/")

@Component

@Scope("prototype")

public class AccountResource {

@Path("{username}/portfolios/")

public PortfolioResource getPortfolioResource(@PathParam("username") String username) {

return new PortfolioResource(accountRepository, username, uriInfo);

}

}

PortfolioResource本身的声明并没有使用注解，因此其所有的依赖都是由父资源传递过来的：

public class PortfolioResource {

private AccountRepository accountRepository;

private String username;

private UriInfo uriInfo;

public PortfolioResource(AccountRepository accountRepository, String username, UriInfo uriInfo) {

this.accountRepository = accountRepository;

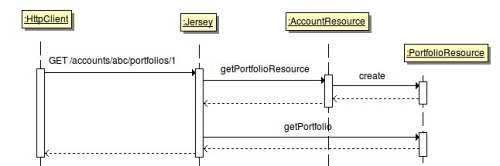
this.username = username;

this.uriInfo = uriInfo;

}

}

JAX-RS中的根与子资源创建了一个处理链，它会调用多个资源：



请记住，资源类是Web Services层组件，应当关注于Web Services相关的处理，比如输入转换、准备响应、设定响应代码等等。此外，将Web Services逻辑与业务逻辑分隔开来的实践需要将业务逻辑包装到单独的方法中以作为事务边界。

**创建Spring MVC @Controller类**

对于Spring MVC来说，我们需要创建DispatcherServlet，同时将contextConfigLocation参数指定为Spring MVC配置：

<servlet>

<servlet-name>Spring MVC Dispatcher Servlet</servlet-name>

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

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>

/WEB-INF/spring/\*.xml

</param-value>

</init-param>

</servlet>

要想在Spring MVC（@MVC）中使用基于注解的编程模型还需要少量的配置。下面的component-scan元素会告诉Spring去哪里寻找@Controller注解类。

<context:component-scan base-package="org.springframework.samples.stocks" />

接下来，我们声明了AccountController，如下代码所示：

@Controller

@RequestMapping("/accounts")

public class AccountController {

@Autowired

private AccountRepository accountRepository;

}

@RequestMapping注解会将该控制器映射到所有以“/accounts”开头的请求上。AccountController中的方法如getAccount()只需声明针对“/accounts”的相对地址即可。

@RequestMapping(value = "/{username}", method = GET)

public Account getAccount(@PathVariable String username) {

}

Spring MVC则没有根资源与子资源的概念，这样每个控制器都是由Spring而非应用来管理的：

@Controller

@RequestMapping("/accounts/{username}/portfolios")

public class PortfolioController {

@Autowired

private AccountRepository accountRepository;

}

对“/accounts/{username}/portfolios”的请求会被直接代理给PortfolioController，AccountController则完全不会参与其中。需要注意的是，该请求也可以直接由AccountController处理，这样就不需要PortfolioController了。

**Web层组件范围**

在JAX-RS中，AccountResource是通过前请求（per-request）语义声明的，这也是JAX-RS默认的推荐设置。这么做可以将特定于请求的数据注入并存储到资源类本身当中，这适用于由JAX-RS所管理的根级别资源。子资源由应用实例化，并不会直接从这种方法中获益。

在Spring MVC中，控制器永远都是单例的，他们将特定于请求的数据作为方法参数。JAX-RS也可以这么做，以单例的方式创建资源。

**将请求映射到方法上**

接下来，我们看看Spring MVC和JAX-RS如何将请求映射到方法上。@Path和@RequestMapping都可以从URL中抽取出路径变量：

@Path("/accounts/{username}")

@RequestMapping("/accounts/{username}")

这两个框架也都可以使用正则表达式抽取路径变量：

@Path("/accounts/{username:.\*}")

@RequestMapping("/accounts/{username:.\*}"

Spring MVC的@RequestMapping可以根据查询参数的有无来匹配请求：

@RequestMapping(parameters="foo")

@RequestMapping(parameters="!foo")

或是根据查询参数值进行匹配：

@RequestMapping(parameters="foo=123")

@RequestMapping还可以根据头信息的有无来匹配请求：

@RequestMapping(headers="Foo-Header")

@RequestMapping(headers="!Foo-Header")

或是根据头信息的值进行匹配：

@RequestMapping(headers="content-type=text/\*")

**处理请求数据**

HTTP请求中包含着应用需要提取和处理的数据，如HTTP头、cookie、查询字符串参数、表单参数以及请求体（XML、JSON等）中所包含的大量数据。在RESTful应用中，URL本身也可以带有重要的信息，如通过路径参数指定需要访问哪个资源、通过文件扩展名（.html, .pdf）指定需要何种内容类型等。HttpServletRequest提供了处理这一切的所有底层访问机制，但直接使用HttpServletRequest实在是太乏味了。

**请求参数、Cookies和HTTP头**

Spring MVC和JAX-RS拥有能够抽取这种HTTP请求值的注解：

@GET @Path

public void foo(@QueryParam("q") String q, @FormParam("f") String f, @CookieParam("c") String c,

@HeaderParam("h") String h, @MatrixParam("m") m) {

// JAX-RS

}

@RequestMapping(method=GET)

public void foo(@RequestParam("q") String q, @CookieValue("c") String c, @RequestHeader("h") String h) {

// Spring MVC

}

上面的注解非常像，区别在于JAX-RS支持矩阵参数（matrix parameters）的抽取，拥有单独的注解来处理查询字符串和表单参数。矩阵参数并不常见，他们类似于查询字符串参数，但却使用了特殊的路径片段（比如GET /images;name=foo;type=gif）。稍后将介绍表单参数。

假如使用了前请求范围声明资源，那么JAX-RS可以在属性和setters方法上使用上述注解。

Spring MVC有个特性能让我们少敲几个字符，如果注解名与Java参数名相同，那么就可以省略掉上面的注解名了。比如说，名为“q”的请求参数要求方法参数也得为“q”：

public void foo(@RequestParam String q, @CookieValue c, @RequestHeader h) {

}

这对于那些在参数中使用了注解而导致方法签名变长的情况来说实在是太方便了。请记住，这个特性要求代码使用调试符号进行编译。

**类型转换与HTTP请求值的格式化**

HTTP请求值（头、cookies和参数）是不变的字符串并且需要解析。

JAX-RS通过寻找valueOf()方法或是在客户化的目标类型中接收字符串的构造方法来解析请求数据。JAX-RS支持如下类型的注解方法参数，包括路径变量、请求参数、HTTP头值和cookies：

* 原生类型。
* 拥有接收单个字符串参数的构造方法的类型。
* 拥有一个接收单个字符串参数的名为valueOf的静态方法的类型。
* List<T>、Set<T>或是SortedSet<T>，其中的T满足上面2个或3个要求。

Spring 3支持上面所有要求。除此之外，Spring 3提供了一种全新的[类型转换与格式化机制](http://blog.springsource.com/2009/11/17/spring-3-type-conversion-and-validation/)，并且可以使用注解实现。

**表单数据**

如前所述，JAX-RS处理查询字符串参数和表单参数的方式是不同的。虽然Spring MVC只有一个@RequestParam，但它还提供了一种Spring MVC用户很熟悉的数据绑定机制来处理表单输入。

比如说，如果一个表单提交了3个数据，那么一种可能的处理方式就是声明一个带有3个参数的方法：

@RequestMapping(method=POST)

public void foo(@RequestParam String name, @RequestParam creditCardNumber, @RequestParam expirationDate) {

Credit card = new CreditCard();

card.setName(name);

card.setCreditCardNumber(creditCardNumber);

card.setExpirationDate(expirationDate);

}

然而，随着表单数据量的增加，这种处理方式就会变得不切实际。借助于数据绑定，Spring MVC可以创建、组装并传递包含有嵌套数据（账单地址、邮件地址等）、任意结构的表单对象。

@RequestMapping(method=POST)

public void foo(CreditCard creditCard) {

// POST /creditcard/1

// name=Bond

// creditCardNumber=1234123412341234

// expiration=12-12-2012

}

要想与Web浏览器协同工作，表单处理是个重要环节。另一方面，Web Services客户端一般会在请求体中提交XML或JSON格式的数据。

**处理请求体中的数据**

无论是Spring MVC还是JAX-RS都能够自动处理请求体中的数据：

@POST

public Response createAccount(Account account) {

// JAX\_RS

}

@RequestMapping(method=POST)

public void createAccount(@RequestBody Account account) {

// Spring MVC

}

**JAX-RS中的请求体数据**

在JAX-RS中，类型MessageBodyReader的实体供应者负责转换请求体数据。JAX-RS的实现需要拥有一个JAXB MessageBodyReader，这可以使用具有注解@Provider的客户化MessageBodyReader实现。

**Spring MVC中的请求体数据**

在Spring MVC中，如果想通过请求体数据初始化方法参数，那可以将@RequestBody注解加到该方法参数前，这与之前介绍的表单参数初始化正好相反。

在Spring MVC中，HttpMessageConverter类负责转换请求体数据，Spring MVC提供了一个开箱即用的[Spring OXM](http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/ch14.html) HttpMessageConverter。它支持JAXB、Castor、JiBX、XMLBeans和XStream，此外还有一个用于[处理JSON](http://blog.springsource.com/2010/01/25/ajax-simplifications-in-spring-3-0/)的[Jackson](http://jackson.codehaus.org/) HttpMessageConverter。

HttpMessageConverter会注册到AnnotationMethodHandlerAdapter上，后者会将到来的请求映射到Spring MVC @Controllers上。下面是其配置：

<bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter" >

<property name="messageConverters" ref="marshallingConverter"/>

</bean>

<bean id="marshallingConverter" class="org.springframework.http.converter.xml.MarshallingHttpMessageConverter">

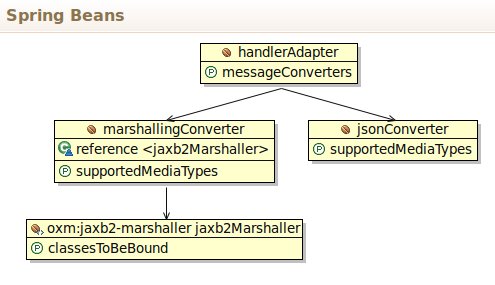
<constructor-arg ref="jaxb2Marshaller"/>

<property name="supportedMediaTypes" value="application/vnd.stocks+xml"/>

</bean>

<oxm:jaxb2-marshaller id="jaxb2Marshaller"/>

下图阐述了该配置：



Spring 3新增的[mvc客户化命名空间](http://blog.springsource.com/2009/12/21/mvc-simplifications-in-spring-3-0/)将上述配置自动化了，只需增加如下配置片段即可：

<mvc:annotation-driven />

如果JAXB位于类路径上，它会注册一个用于读写XML的转换器；如果[Jackson](http://jackson.codehaus.org/)位于类路径上，它会注册一个用于读写JSON的转换器。

**准备响应**

典型的响应需要准备响应代码、设定HTTP响应头、将数据放到响应体当中，还需要处理异常。

**使用JAX-RS设定响应体数据**

在JAX-RS中，要想将数据加到响应体中，只需要从资源方法中返回对象即可：

@GET

@Path("{username}")

public Account getAccount(@PathParam("username") String username) {

return accountRepository.findAccountByUsername(username);

}

JAX-RS会寻找类型MessageBodyWriter的实体供应者，它能将对象转换为所需的内容类型。JAX-RS实现需要具备一个JAXB MessageBodyWriter，这可以使用具有注解@Provider的客户化MessageBodyWriter实现。

**使用Spring MVC设定响应体数据**

在Spring MVC中，响应是通过一个视图解析过程来实现的，这样就可以从一系列视图技术中选择了。但在与Web Services客户端交互时，更加合理的方式则是舍弃视图解析过程，转而使用方法所返回的对象：

@RequestMapping(value="/{username}", method=GET)

public @ResponseBody Account getAccount(@PathVariable String username) {

return accountRepository.findAccountByUsername(username);

}

如果对控制器方法或其返回类型应用注解@ResponseBody，那么就会使用HttpMessageConverter处理返回值，然后用该返回值设定响应体。用于请求体参数的HttpMessageConverter集合也用于响应体，因此无需再做任何配置。

**状态代码与响应头**

JAX-RS使用一个链式API来构建响应：

@PUT @Path("{username}")

public Response updateAccount(Account account) {

// ...

return Response.noContent().build(); // 204 (No Content)

}

这可以与UriBuilder联合使用来为Location响应头创建实体链接：

@POST

public Response createAccount(Account account) {

// ...

URI accountLocation = uriInfo.getAbsolutePathBuilder().path(account.getUsername()).build();

return Response.created(accountLocation).build();

}

上面代码中所用的uriInfo要么被注入到根资源（使用了@Context）中，要么是从父资源传递给子资源。它可以附加到当前请求的路径之后。

Spring MVC提供了一个注解来设定响应代码：

@RequestMapping(method=PUT)

@ResponseStatus(HttpStatus.NO\_CONTENT)

public void updateAccount(@RequestBody Account account) {

// ...

}

可以直接使用HttpServletResponse对象设定Location头：

@RequestMapping(method=POST)

@ResponseStatus(CREATED)

public void createAccount(@RequestBody Account account, HttpServletRequest request,

HttpServletResponse response) {

// ...

String requestUrl = request.getRequestURL().toString();

URI uri = new UriTemplate("{requestUrl}/{username}").expand(requestUrl, account.getUsername());

response.setHeader("Location", uri.toASCIIString());

}

**异常处理**

JAX-RS允许资源方法抛出WebApplicationException类型的异常，该异常会包含一个响应。下面的示例代码将一个JPA NoResultException转换为特定于Jersey的NotFoundException，这会导致一个404的错误：

@GET

@Path("{username}")

public Account getAccount(@PathParam("username") String username) {

try {

return accountRepository.findAccountByUsername(username);

} catch (NoResultException e) {

throw new NotFoundException();

}

}

WebApplicationException实例会封装必要的逻辑来生成特定的响应，但每个独立的资源类方法中都需要捕获异常。

Spring MVC支持定义控制器级别的方法来处理异常：

@Controller

@RequestMapping("/accounts")

public class AccountController {

@ResponseStatus(NOT\_FOUND)

@ExceptionHandler({NoResultException.class})

public void handle() {

// ...

}

}

如果任何控制器方法抛出了JPA的NoResultException异常，上面的处理器方法就会得到调用并处理该异常，然后返回一个404错误。这样，每个控制器就都能处理异常了，好象来自同一个地方一样。

**总结**

希望本文有助于你理解Spring MVC构建RESTful Web Services的方式及其与JAX-RS编程模型之间的异同点。

如果你是个Spring MVC用户，那么你可能用它开发过HTML Web应用了。REST概念适用于Web Services和Web应用，尤其是富客户端交互上更是如此。除了本文介绍的特性之外，Spring 3还增加了对RESTful Web应用的支持。这是部分新特性的列表：用于从URL模板构建URL的新的JSP客户化标签、基于HTTP PUT和DELETE模拟表单提交的Servlet过滤器、根据内容类型自动选择视图的ContentTypeNegotiatingViewResolver、新的视图实现等等。此外，Spring文档也改进颇多。

**关于作者**

Rossen Stoyanchev是SpringSource的高级咨询师。在其职业生涯中，他做过贸易应用、记账系统和电子商务等Web应用。在SpringSource中，Rossen专注于Web技术，包括咨询、培训和“Rich-Web Development With Spring”课程的内容开发，该课程旨在帮助受训者成为认证的Spring Web应用开发者。

**查看英文原文：**[A Comparison of Spring MVC and JAX-RS](http://www.infoq.com/articles/springmvc_jsx-rs)