# 概述

REST（REpresentational Stat Transfer 表述性状态转移）架构是基于互联网的软件开发及集成架构，我们一直认为HTTP是用来连接系统的，REST更近一步，不仅仅将HTTP作为一种便于使用的数据通道，而是拥抱WEB本身正常工作的方式，将其作为系统协作集成的基础。

ROA全称是REST OPEN API，它不同于一般纯粹的Web Service技术框架（例如：CXF、Jersey、RESTEasy等等），ROA致力于构建服务开放平台（OPEN API）的平台。我们可以基于ROA去搭建类似于豆瓣开放API（http://developers.douban.com）、淘宝开放平台（http://open.taobao.com）等应用。ROA充分的学习借鉴了当前大型互联网站点的设计思路，汲取它们的架构经验，对于OPEN API平台的很多应用层领域问题给出了相应的解决方案，开发者可以直接使用这些解决方案，也可以在其基础上进行个性化修改定制。

该平台软件作为个人成果，遵循GNU License。

## 将WEB作为建造分布式系统的平台

REST的目标用一句话表述：基于WEB的架构来建造分布式系统，使用WEB的主要应用协议HTTP（超文本转移协议）来实现系统，而HTTP协议则利用了REST架构的原则。

## ROA功能架构

Jersey、CXF、RESTEasy是纯技术的Web Service框架，而在ROA中，Web Service只是作为核心，它更多的提供了开发服务平台的很多问题域的解决方案，例如：应用认证、会话管理、安全控制、异常模型、访问超时限制等方面。



从上图中我们可以清晰的看到，ROA提供的大部分功能是偏向于“应用层”的，传统的Web Service框架是不会提供这些“应用层”问题的。但是，在实际的开发中，这些应用层面的问题

## ROA技术实现

ROA在技术实现上依赖于Spring Framework，充分借鉴了Spring MVC框架的实现理念和实现技术，ROAServlet类似于Spring MVC的DispatherServlet，是ROA的主要Servlet实现，负责拦截HTTP服务请求转由ROA框架进行处理。

# Request模型

## 传统Web Service的Request模型

Request模型设计的好坏将直接影响到整个ROA框架的调用效率，设计良好的Request模型可以让复用调用成为一件异常简单的事情。另外，它还能使得服务接口清晰化，降低开发者理解服务的难度。我们先来了解一下传统的Web Service的Request模型。

### SOAP Request模型

Web Service基本上都是使用HTTP协议进行交互，服务的响应报文一般会支持XML和JSON两种格式，但是Web Service的Request模型却会由于实现不同而会差异很大。

传统的Web Service框架都会支持SOAP格式的Request报文，任何服务都对应一组SOAP Request/Response报文，服务的调用及报文的解析都比较麻烦。举例来说，即时我们要调用一个城市的当前天气预报的简单请求，该服务只有一个city参数，但在SOAP协议中，我们必须将其封装成一个复杂的SOAP Request报文才可以。一般情况下，如果开发者不基于CXF、Axis等框架，很难使用SOAP来进行Web Service搭建。

### REST Request模型

开发者总是希望能够随时随地方便的访问服务和资源，比如通过[http://api.city.org/weather/{city}](http://api.city.org/weather/%7bcity%7d) 这种URI的方式来访问服务获得响应。把服务看成一种普通资源，通过一个URI（不一定是唯一，尽管多个URI能够标示同一个资源，但是Web无法提供任何方式来计算两个不同的URI是否实际上指向同一资源）进行定位和调用。

URI将表述与它们在Web上的资源联系、连接、关联关系。这种在资源和它的表述之间的分离，促进了后端系统和消费应用程序之间的松耦合。它对可伸缩性也有帮助，因为表述是可以被缓存和复制的。

REST Web Service充分挖掘了HTTP通讯协议的内涵，除了URI方式来定位资源/服务，HTTP所支持的动词集合（GET、POST、PUT、DELETE、OPTIONS、HEAD、TRACE、CONNECT、PATCH）构成了一个足够通用的协议，可以支持广泛的解决方案。

豆瓣网（http://douban.com）的API就是采用了标准的REST Web Service来开发的，我们来看一个获取图书信息的API：

|  |
| --- |
| [http://api.douban.com/book/subject/isbn/{isbnID}](http://api.douban.com/book/subject/isbn/%7bisbnID%7d) |

该服务使用了HTTP的GET方法进行调用，如果我们在浏览器中输入一下代码：

|  |
| --- |
| <http://api.douban.com/book/subject/isbn/9787807095101> |

以上请求将获得一个关于图书《不是为了快乐》信息的Response报文（XML格式），我们可以再浏览器中看，也可以写一段代码来消费这个Response报文，完成你自己应用需要做的工作。这种调用方式对于开发者来说非常友好，它通过URI进行，和访问网页无异。我一直将豆瓣看做一个技术驱动型的公司，如果要学习REST OPEN API设计的话，豆瓣的API是一个非常优秀的学习案例，我们简单看几个例子：

* GET [http://api.douban.com/book/subject/{subjectID}](http://api.douban.com/book/subject/%7bsubjectID%7d)：获取某本书的信息，GET是表示使用HTTP Request方法；
* GET [http://api.douban.com/people/{userID}](http://api.douban.com/people/%7buserID%7d)：获取某个用户的信息；
* DELETE [http://api.douban.com/review/{reviewID}](http://api.douban.com/review/%7breviewID%7d)：删除某篇评论；

因为HTTP并非仅仅是另一种消息协议，它是一种实现了一些非常特殊的应用语义的协议。HTTP的动词（特别是GET）支持缓存，它将减少延迟，为巨大的吞吐量提供良好的水平伸缩，保证大型应用的稳定性。

采用REST Request模型发布的服务接口变得非常清晰，调用也相对简单，REST已经模糊了服务和网页资源的界限，也模糊了不同应用实现方式的差异。Simple is Good，从这个意义上来说，REST明显优于SOAP，开发者也慢慢的偏向于REST架构。另外，传统上分布式系统开发将重点放在以特定应用的接口和交互协议的形式来暴露定制的行为，与之相反，Web则将重点放在少量广为人知的网络动作（HTTP动词集合）以及公共的表述格式。

按照Leonard·Richardson的REST的成熟度模型来看，REST分为三个层次：URI、HTTP、Hypermedia，处于前两个级别的REST架构对HTTP的请求方式也显得非常的刻板和教条：新增、更改、删除、获取资源的服务分别对应POST、PUT、DELETE、GET。



## ROA Request模型

ROA Request模型直接参考了Taobao Open Platform的思想，所有服务的URI都是相同的。Request Parameter分为System Parameter和Business Parameter两个部分，System Parameter是所有Open API都拥有的，而Business Parameter由具体服务API定义。

### 统一服务URI

采用ROA的Service Open API，所有的服务都使用统一的URI，ROA通过Method参数将Request路由到指定的Service方法中去完成服务受理。ROA平台通过ROAServlet的<servlet-mapping>来对统一服务URI进行设置。

|  |
| --- |
| <servlet>  <servlet-name>roa</servlet-name>  <servlet-class>  com.wondersgroup.roa.servlet.ROAServlet  </servlet-class>  <load-on-startup>2</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>roa</servlet-name>  <url-pattern>/openapi</url-pattern>  </servlet-mapping> |

服务平台最终的URL为：<开放平台根URI>/<ROAServlet的映射URI>。举例来说，服务器URI为api.wondersgroup.com，而ROAServlet的映射URI为/openapi，则服务统一URI为：<http://api.wondersgroup.com/openapi>。

### System Parameter

System Parameter是由Open API定义的一组参数，每个服务都拥有这些参数，用以传送ROA框架级别的参数信息。如我们前面提到的method就是一个系统级参数，使用该参数指定服务的名称。ROA共有7个系统级参数，在表中说明：

|  |  |  |
| --- | --- | --- |
| 参数名称 | 是否必须 | 说明 |
| apiKey | 是 | 为了保护平台数据，防止API被滥用或恶意使用，系统要求每个API的使用者申请一个API Key，而每个API Key唯一标识一个API使用者。API Key对应一个密钥secret，要基于ROA平台开发应用，必须事先通过申请获取apiKey/secret后，才能进行应用的开发。 |
| sessionId | 否 | 会话ID，一般是一个36位的UUID，在登录服务平台后获取； |
| method | 是 | 服务方法名，一般采用“名词+动词”的结构定义。如user.get、user.create等； |
| format | 否 | 通信报文格式，可选值为xml和json，默认为xml。 |
| locale | 否 | 本地化类型，默认为zh\_CN。 |
| sign | 是 | 签名串，请求参数的签名，服务平台通过它验证请求数据的合法性。 |
|  |  |  |

默认情况下，系统级参数名是固定的，一般情况下，并不需要对调整它。如果希望使用自行定义的参数名称，可以使用<roa:sysparams/>进行定义，如下所示：

|  |
| --- |
| <roa:sysparams  format-param-name="messageFormat"  apikey-param-name="api\_key"/> |

### Business Parameter

业务级参数，是由业务逻辑需要自行定义的，每个服务API都可以定义若干个自己的业务级参数。ROA根据参数名和ROARequest类属性名相等的契约，将业务级参数绑定到ROARequest中。

如下所示LogonRequest定义了两个userName和password两个属性，ROA就会将HTTP请求参数值绑定到LogonRequest对象的同名属性中。

|  |
| --- |
| import com.wondersgroup.roa.AbstractROARequest;  import com. wondersgroup.roa.annotation.IgnoreSign;  import javax.validation.constraints.Pattern;  public class LogonRequest extends AbstractROARequest{    @Pattern(regexp = "\\w{4,30}")  private String userName;  @IgnoreSign  @Pattern(regexp = "\\w{6,30}")  private String password;    //get/setter  } |

## Parameter数据的绑定和校验

### Parameter数据绑定

当客户端调用服务平台某个服务时，其实质是向服务平台的URI发送若干个请求参数（包括系统级和业务级的参数）。ROA框架在接收到这些请求参数后，就会将其绑定到ROARequest请求对象中，服务方法可通过这个ROARequest对象获取请求参数信息，进而执行相应的服务API并返回响应结果。下图描述了请求参数的转换过程：



首先，客户端的服务请求通过HTTP报文发送给服务端的Servlet服务器（即HTTP服务器），Servlet服务器将HTTP报文转换成一个HttpServletRequest对象。然后通过ROAServlet转交给ROA框架，ROA框架将HttpServletRequest转换成一个ROARequestContext对象。接着，ServiceRouter将ROARequestContext传给ServiceMethodAdapter，ServiceMethodAdapter在内部将ROARequestContext转换成ROARequest对象，输送给最终的服务方法。

从上面的数据转换过程中，我们知道每当客户端发起一个服务调用时，ROA都会在内部创建一个ROARequestContext实例，它包含了所有的请求数据信息。下面，我们来了解一下ROARequestContext接口的方法：

|  |  |
| --- | --- |
| 方法名 | 说明 |
| String getApiKey() | 获取apiKey系统级参数的值。ROARequestContext为每个系统级参数都分配了一个对应的接口方法，如String getMethod()、String getSessionId()等； |
| HttpAction getHttpAction() | 获取HTTP请求方法，HttpAction是一个枚举，仅有三个枚举值，即GET、DELETE和POST。这也说明，ROA仅支持GET、DELETE和POST三个HTTP请求方法； |
| String getIp() | 获取请求来源的IP地址。由于在集群环境下，请求通过前端的负载均衡器再传给后端集群的某个具体服务节点。因此，直接使用ServletRequest#getRemoteAddr()返回的值将是前端负载均衡服务器的IP，在此ROA使用了一些技巧，以保证后端服务获取的IP是客户端的IP。 |
| Object getRawRequestObject() | 获取原请求对象，即服务请求对应的HttpServletRequest对象； |
| Map getAllParams() | 获取服务请求所对应的所有请求参数。可以通过String getParamValue(String paramName)获取某个具体参数的值； |
| ROAContext getROAContext() | 获取ROA框架上下文的信息。ROAContext之于ROA框架相当于ServletContext之于Servlet容器，它包含了很多ROA框架的运行期信息，所有ROA服务方法都注册在ROAContext 中。 |

### Parameter数据校验

## XML和JSON参数绑定

如果某个请求参数的值是一个XML或JSON串，能否正确地进行绑定呢？ROA框架支持将XML或JSON格式的参数值透明地绑定到ROARequest的复合属性中。

我们通过常用的UserService#addUser(CreateUserRequest request)来说明XML/JSON参数值绑定的内容。CreateUserRequest拥有一个Address的业务级参数，如下所示：

|  |
| --- |
| import javax.validation.Valid;  …  public class CreateUserRequest extends AbstractROARequest {  @Pattern(regexp = "\\w{4,30}")  private String userName;  @Valid  // 可绑定XML或JSON的复合属性，必须打上@Valid注解进行数据校验  private Address address;  } |

Address是一个复合对象属性，它的类结构对应XML的结构：

|  |
| --- |
| import javax.validation.constraints.Pattern;  import javax.xml.bind.annotation.\*;  import java.util.List;  //使用JSR-222注解，定义了基于属性名进行数据绑定的规则  @XmlAccessorType(XmlAccessType.FIELD)  @XmlRootElement(name = "address")  public class Address {    //使用JSR-222定义数据绑规则，使用JSR-303注解定义数据校验规则  @XmlAttribute  @Pattern(regexp = "\\w{4,30}")  private String zoneCode;  @XmlAttribute  private String doorCode;    //使用JSR-222注解指定列表数据的绑定规则  @XmlElementWrapper(name = "streets")  @XmlElement(name = "street")  private List<Street> streets;  } |

JSR-222（JAXB）标准规范，已经作为XML数据绑定官方标准添加到JDK 6.0核心库中。因此，我们直接使用JSR-222注解定义XML数据的绑定规则。官方标准的JAXB库只支持XML数据的绑定，很多开源进行了扩展，支持JSON数据的绑定，ROA使用Jackson项目完成JSON数据的绑定。

请求数据绑定时一般都需要进行数据校验，因此您还需要使用JSR-303的注解定义数据校验规则。通过JSR-222和JSR-303注解的使用，ROA很轻巧地解决了请求数据绑定和数据校验的问题。

## 自定义数据转换器

对于复合结构的参数，我们推荐使用XML或JSON的格式指定参数内容。除此以外，ROA允许用户通过注册自定义转换器来支持自定义格式的参数。Spring Framework 3.0后新增了一个类型转换的核心框架，可以实现任意两个类型对象数据的转换，即org.springframework.core.convert.ConversionService。

FormattingConversionService扩展于ConversionService，添加了格式化数据的功能。Spring Framework的数据类型转换体系是高度可扩展的，ROA就是基于Spring Framework的类型转换体系实施参数数据绑定的工作。

在Spring的类型转换服务体系中，转换器是由Converter<S, T>接口定义，它仅能实现单向转换，即从S到T的转换。但是ROA有时需要双向转换功能：在服务端将参数绑定到ROARequest时，将S转换成T，而在客户端将ROARequest流化成请求报文时，需要将T转换成S。因此，ROA对Converter<S, T>接口进行了扩展，定义了一个可以实现双向转换的接口，如下所示：

|  |
| --- |
| import org.springframework.core.convert.converter.Converter;  public interface ROAConverter<S, T> extends Converter<S, T> {  S unconvert(T target);    Class<S> getSourceClass();  Class<T> getTargetClass();  } |

Converter<S, T>接口定义了一个T convert(S source)的方法，ROAConverter<S, T>新增了一个S unconvert(T target)的方法，这样就可以实现S和T两者的双向转换了。开发一个类型转换器是件轻松的事情，仅需扩展ROAConverter<S, T>接口并实现S和T相互转换的逻辑即可。我们尝试定义了一个可实现格式化电话号码和Telephone对象的双向转换器：

|  |
| --- |
| import com.wondersgroup.roa.request.ROAConverter;  import org.springframework.core.convert.converter.Converter;  import org.springframework.util.StringUtils;  public class TelephoneConverter implements ROAConverter<String, Telephone> {  //将格式化字符串转换为Telephone  @Override  public Telephone convert(String source) {  if (StringUtils.hasText(source)) {  String zoneCode =  source.substring(0, source.indexOf("-"));  String telephoneCode =  source.substring(source.indexOf("-") + 1);  Telephone telephone = new Telephone();  telephone.setZoneCode(zoneCode);  telephone.setTelephoneCode(telephoneCode);  return telephone;  }  else {  return null;  }  }  //将Telephone转换为格式化字符串  @Override  public String unconvert(Telephone target) {  StringBuilder sb = new StringBuilder();  sb.append(target.getZoneCode());  sb.append("-");  sb.append(target.getTelephoneCode());  return null;  }  @Override  public Class<String> getSourceClass() {  return String.class;  }  @Override  public Class<Telephone> getTargetClass() {  return Telephone.class;  }  } |

接下来的工作是如何将TelephoneConverter注册到ROA中，以便ROA在进行参数数据绑定时利用这个转换器。

ROA的<roa:annotation-driven/>拥有一个formatting-conversion-service属性，可以通过该属性指定一个Spring Framework的FormattingConversionService。在FormattingConversionService中即可注册自定义的Converter，如下所示：

|  |
| --- |
| <roa:annotation-driven  formatting-conversion-service="conversionService"/>  <bean id="conversionService"  class="org.springframework.format.support.FormattingConversionServiceFactoryBean">  <property name="converters">  <set>  <!--将xxxx-yyy格式化串转换为Telephone对象-->  <bean class="com.wondersgroup.roa.sample.request.TelephoneConverter"/>  </set>  </property>  </bean> |

CreateUserRequest中拥有一个Telephone的属性：

|  |
| --- |
| public class CreateUserRequest extends AbstractROARequest {  @Pattern(regexp = "\\w{4,30}")  private String userName;    @IgnoreSign  @Pattern(regexp = "\\w{6,30}")  private String password;    private Telephone telephone;  …  } |

Telephone类拥有zoneCode和telephoneCode两个属性，ROA在处理Telephone类型的数据绑定时，将自动调用TelephoneConverter进行数据转换。

UserServiceClient#testCustomConverter()演示了客户端使用TelephoneConverter的方法：

|  |
| --- |
| @Test  public void testCustomConverter() {  //注册转换器，实现Telephone到格式化字符串的转换  roaClient.addROAConvertor(new TelephoneConverter());  CreateUserRequest request = new CreateUserRequest();  request.setUserName("tomson");  request.setSalary(2500L);  Telephone telephone = new Telephone();  telephone.setZoneCode("0592");  telephone.setTelephoneCode("12345678");  CompositeResponse response = roaClient.buildClientRequest()  .post(request, CreateUserResponse.class, "user.add", "1.0");  assertNotNull(response);  assertTrue(response.isSuccessful());  assertTrue(response.getSuccessResponse()  instanceof CreateUserResponse);  } |

ROAClient注册了一个TelephoneConverter实现，当调用post()发送服务请求时，TelephoneConverter就会自动将Telephone对象转换成一个xxx-yyy的格式化串，并以请求报文的方式发送给服务端。而服务端则会利用注册在ConversionService中的TelephoneConverter，将xxx-yyy格式的电话号转为Telephone对象。

## Request Service映射

Spring MVC通过@RequestMapping注解实现HTTP请求到处理方法的映射。类似的，ROA使用@ServiceMethod注解实现HTTP Request到服务处理方法的映射。@ServiceMethod只能对Bean的方法进行标注，且该方法的签名是受限的：拥有一个ROARequest的入参和一个返回对象。

@ServiceMethod的method和version属性值是必须的，method代码服务方法名，而version表示版本号。如代码清单1的getSession()服务方法对应的注解是@ServiceMethod(method= "user.getSession", version = "1.0")，它对应如下的服务请求：

|  |
| --- |
| http://<serverUrl>/<roaServletUri>?method=user.getSession&v=1.0&... |

来看一个具体的例子：

|  |
| --- |
| //服务类必须是一个Spring Framework的Bean  @Service  public class UserService {    //服务方法对应如下的HTTP请求，?method==user.add&v=1.0&...  @ServiceMethod(method = "user.add", version = "1.0")  public Object addUser(CreateUserRequest request) {  ...  }  } |

服务开放平台一旦将服务发布出去后，其内部实现可以不断优化和调整，但是服务接口必须保证不变，否则基于服务开发的第三方应用的运行稳定性就得不到保障。如果要调整服务接口定义，必须升级版本，这也是ROA为什么要求方法名一定要和版本同时提供的原因。

一个服务方法可以同时存在多个版本，客户端可以调用指定版本的服务。来看几个不同版本的服务及对应的客户端调用参数：

@ServiceMethod(method="user.add",version="2.0") == method=user.add&v=2.0；

@ServiceMethod(method="user.add",version="3.0") == method=user.add&v=3.0；

@ServiceMethod(method="user.get",version= "1.5") == method=user.get&v=1.5；

@ServiceMethod除了method和version属性外，还拥有多个其它的属性，分别说明如下：

|  |  |
| --- | --- |
| 名称 | 说明 |
| group | 服务分组名。服务的分组没有特殊的意义，您可以为服务定义一个分组，以便在事件监听器、服务拦截器中利用分组信息进行特殊的控制。默认的分组为ServiceMethodDefinition.DEFAULT\_GROUP； |
| groupTitle | 服务分组标识； |
| tags | tags的类型是一个String[]，您可以给服务打上一个或多个TAG，以便在事件处理监听器、服务拦截器利用该信息进行特殊的处理； |
| title | 服务的标识； |
| httpAction | 服务允许的HTTP请求方法，可选值在HttpAction枚举中定义，即GET、POST和DELETE等，如果不指定则不限制； |
| needInSession | 表示该服务方法是否需要工作在会话环境中，默认所有的服务方法必须工作于会话环境中，也即请求的sessionId不能为空。如果某个方法不需要工作于会话环境中（如登录的服务方法、获取应用最新版本的服务方法），则必须显式设置：needInSession = NeedInSessionType.NO； |
| ignoreSign | 表示该服务方法是否要进行请求数据签名验证，默认为需要。如果不需要，可以设置：ignoreSign=IgnoreSignType.NO。正式环境务必开启请求签名验证的功能，这样才能对客户端请求的合法性进行校验； |
| timeout | 服务超时时间，单位为秒。如果服务方法执行时间超过timeout后，ROA将直接中断服务并返回错误的报文。 |

@ServiceMethod拥有众多的可设置属性，它们都和ROA具体的领域性问题相关联，因此，在这里只要知道method和version的属性就可以了，后面会对其它的属性进行深入的讲解。

如果一个服务类中拥有多个服务方法，而它们拥有一些共同的属性，如group、version等，能否在某个地方统一定义呢？答案是肯定的，ROA为复用服务方法元数据信息提供了一个类级别的@ServiceMethodBean。@ServiceMethodBean拥有一套和@ServiceMethod类似的属性，其属性值会被同一服务类中所有的@ServiceMethod继承。

@ServiceMethodBean类本身已经标注了Spring Framework的@Service，所以标注了@ServiceMethodBean的服务类就相当于打上的@Service，可以被Spring Framework的Bean扫描器扫描到。

下面的例子拥有两个服务方法，它们的version都是1.0：

|  |
| --- |
| @ServiceMethodBean(version = "1.0")  public class UserService {    @ServiceMethod(method = "user.add")  public Object addUser(CreateUserRequest request) {  ...  }  @ServiceMethod(method = "user.get", httpAction = HttpAction.GET)  public Object getUser(CreateUserRequest request) {  ...  }  } |

ROA框架在启动时，将创建代表ROA框架上下文的ROAContext实例，同时扫描Spring Framework容器中所有的Bean，将标注了@ServiceMethod的Bean方法注册到ROAContext的服务方法注册表中。这样，ServiceRouter就可根据ROAContext中的服务方法注册表进行请求服务的路由了。

# 应用授权与验证

## ApiKey

当用户需要访问某个应用系统前，应用系统一般都需要对该用户进行身份认证。常见的身份认证方法是让用户输入“用户/密码”，当通过验证后，允许进入系统，否则阻止用户登录系统。和应用系统类似，服务开放平台也需要对接入的应用进行身份认证，以确保服务只向合法授权的客户端应用开放。一般的做法是：服务开放平台通过一个应用申请流程向通过审核的开发者分配一个唯一的应用键和应用密钥（即apiKey/secret）。应用键是公开的，而应用密钥是保密的，只有开发者自己知道。

开发者开发的应用在访问开放平台的服务时，都必须带上这个apiKey，以亮明自己的身份。此外，还必须通过应用密钥对请求数据进行签名，开放平台通过验证服务请求的签名判断客户端应用的合法性。也就是说，开放平台通过apiKey/secret的机制对应用进行身份认证。

## ApiSecretManager

由于ROA需要在服务端采用相同的算法计算请求参数的签名，并和客户端传送过来的签名进行比较，如果两者相等，便认为当前交互的客户端是合法的客户端，反之则认为是一个非法的客户端。

因此，服务端必须知道应用键及其应用密钥的信息，这样才能顺利完成服务端签名验证的工作，ROA通过com.wondersgroup.roa.security.ApiSecretManager接口访问应用键/密钥。您可以采用适合的方式保存应用键/密钥，如保存在数据库、LDAP、文件系统等地方，然后编写一个访问应用键/密钥的ApiSecretManager实现类就可以了。

ApiSecretManager定义了两个接口方法：

|  |  |
| --- | --- |
| 方法名称 | 说明 |
| boolean isValidApiKey(String apiKey) | 判断apiKey是否是合法的应用键； |
| String getSecret(String appKey) | 根据appKey获取对应的应用密钥。 |

ROA默认提供了一个基于文件存储的FileBaseApiSecretManager实现类，FileBaseApiSecretManager默认使用读取类路径下的roa.appSecret.properties属性文件，获取应用键/密钥，属性文件中应用键/密钥采用如下方式保存：

|  |
| --- |
| 00001=abcdeabcdeabcdeabcdeabcde  00002=abcdeabcdeabcdeabcdeaaaaa |

属性名对应应用键，属性值对应应用密钥。如果属性文件放置在其它地方，则可以通过apiSecretFile属性指定位置，apiSecretFile支持“classpath:”等Spring Framework资源类型的前缀。由于大型服务平台一般是分布式的，所以将应用键/密钥保存在系统文件中并不是个好主意。如果开发者希望提供自定义的ApiSecretManager，可通过<roa:annotation-driven/>的api-secret-manager属性进行配置：

|  |
| --- |
| <roa:annotation-driven api-secret-manager="apiSecretManager"/>  <bean id="apiSecretManager"  class="com.wondersgroup.roa.sample.SampleApiSecretManager"/> |

这样，ROA就会使用SampleAppSecretManager取代默认FileBaseAppSecretManager进行应用键/密钥的读取工作了。

## 签名算法

ROA的签名算法直接参考了TOP的签名算法，该签名算法描述如下：

* 所有请求参数按参数名升序排序；
* 按请求参数名及参数值相互连接组成一个字符串：<paramName1> <paramValue1> <paramName2> <paramValue2>…；
* 将应用密钥分别添加到以上请求参数串的头部和尾部：<secret><请求参数字符串><secret>；
* 对该字符串进行SHA1运算，得到一个二进制数组；
* 将该二进制数组转换为十六进制的字符串，该字符串即是这些请求参数对应的签名；
* 该签名值使用sign系统级参数一起和其它请求参数一起发送给服务开放平台。

假设，user.create的服务有三个业务级参数，分别为userName、age及sex；这些业务级参数和系统级参数的值如下表所示：

|  |  |  |  |
| --- | --- | --- | --- |
| System Parameter | Reference | Business Parameter | Reference |
| apiKey | 000001 | userName | Tomson |
| sessionId | AAAA | age | 24 |
| method | user.create | sex | M |
| v | 1.0 |  |  |
| format | xml |  |  |
| locale | Zh\_CN |  |  |

根据ROA的签名算法，首先按字母顺序将所有参数名和参数值拼装成一个字符串：

|  |
| --- |
| age24apiKey000001formatxmllocalezh\_CNmethoduser.createsessionIdAAAAsex1userNametomsonv1.0 |

假设，apiKey为000001的secret（应用密钥）是“abcdef”，则将“abcdef”分别添加到以上请求参数串的头部和尾部，得到：

|  |
| --- |
| abcdefage24appKey000001formatxmllocalezh\_CNmethoduser.createsessionIdAAAAsex1userNametomsonv1.0abcdef |

对以上字符串进行SHA1签名运算，将签名值转换为十六进制的编码串，得到：

|  |
| --- |
| 8625FD7EEAE1E68203B48C64DE495792BF59E833 |

最后，客户端即可使用如下的URL请求串对user.create服务方法发起请求：

|  |
| --- |
| http://<serverUrl>/<roaServletUri>?appKey=000001&method=user.create&…&sign=8625FD7EEAE1E68203B48C64DE495792BF59E833 |

## 签名功能控制

默认情况下，ROA会对每个服务请求进行签名验证，如果签名验证报错，将直接驳回请求并回报相应的错误信息。ROA允许服务平台开发者开启或关闭签名验证的功能，ROA提供了三个级别的控制：

* 平台级：开启或关闭服务平台所有服务的签名验证功能；
* 服务级：在平台级签名验证功能开启的情况下，可以关闭某个具体服务的签名验证；
* 参数级：我们知道ROA的签名算法要求把所有的参数拼装成一个字符串，如果有些参数值很大（如上传文件的文件内容），签名算法将需要构造一个很大的字符串，占用很大的内存。从安全上来说，仅需对一些关键的参数进行签名就可以了，并非一定要对所有的参数进行签名。有鉴于此，ROA在服务签名时允许忽略某些参数，提供参数级的签名控制。

### 平台级别控制

通过<roa:annotation-driven/>的sign-enable属性即可开启或关闭服务平台签名验证功能：

|  |
| --- |
| <roa:annotation-driven sign-enable="false"/> |

我们强烈建议在生产环境下开启服务签名验证的功能，以保证服务平台的安全性，免受恶意客户终端的攻击。

### 服务级别控制

在平台级签名功能开启的情况下，ROA还允许关闭某个服务的签名验证功能。通过将@ServiceMethod的ignoreSign属性设置为IgnoreSignType.YES即可：

|  |
| --- |
| @ServiceMethod(method = "user.add", version = "5.0",  ignoreSign = IgnoreSignType.YES)  public Object addUser5(CreateUserRequest request) {  CreateUserResponse response = new CreateUserResponse();  response.setCreateTime("20120101010102");  response.setUserId("4");  return response;  } |

这样，客户端在访问user.add#5.0的服务方法时，就不必提供请求参数的签名信息了。

### 参数级别控制

在定义服务方法的ROARequest类时，只要在ROARequest的某些属性上标注了@IgnoreSign，这些属性所对应的请求参数就可以排除在签名参数列表之外了。来看一个例子：

|  |
| --- |
| public class LogonRequest extends AbstractROARequest{  @Pattern(regexp = "\\w{4,30}")  private String userName;    @IgnoreSign  @Pattern(regexp = "\\w{6,30}")  private String password;  ...  } |

LogonRequest的password属性所对应的请求参数将不会纳入到签名算法的参数列表中。使用这种办法，可以在具体的ROARequest类中将某些属性对应的请求参数排除在签名算法之外。如果希望某一类型的属性统一忽略签名，有没有简单的方法呢？ROA提供了一种非常便捷的方法，即在属性类定义处使用@IgnoreSign注解即可。如用于保存上传文件的UploadFile类就标注了@IgnoreSign注解：

|  |
| --- |
| @IgnoreSign  public class UploadFile {  private String fileType;  private byte[] content;  } |

这样，UploadFile作为任何ROARequest类的属性都将排除在签名算法的参数列表之外。

# Error Model

对于服务开放平台来说，不管发生了什么错误，都必须返回相应的错误报文，以便客户端应用能够根据错误报文做出相应的响应。每个服务都可能存在各种错误，如服务参数不合法、访问权限不足、版本不正确、访问超限等等。如果需要开发者自行设计这套错误模型并处理所有这些错误，那将是一项艰巨的工程。

ROA参考TOP建立了一个完整的错误模型，该错误模型拥有强大的表达能力和扩展性，很多错误自动于ROA负责处理，对于业务性的错误，开发者按错误模型构造即可。ROA将开发者从服务错误处理的技术细节中解放出来，从而可以将精力集中于具体的业务逻辑的处理上。

ROA的错误模型分为主错误和子错误两个层级，每个错误报文都会对应一个主错误和若干个子错误，主错误描述错误的类型，子错误说明错误的原因。子错误根据责任归属可以划分为ISP（Internet Service Provider）和ISV（Independent Software Vendors）两种类型的错误。如下图所示：



# Service Session管理

## Session管理

一个客户端应用开发出来后，可以有很多具体的使用者。ROA使用应用键/密钥可以定位到一个具体的客户端应用，但却无法定位到客户端应用当前的使用者。服务开放平台必须开放一个用户登录的服务，在登录成功后分配一个sessionId。这样，应用用户后续对服务平台的服务调用都附上这个sessionId，服务端就可根据这个sessionId判断请求用户的身份了。

ROA作为一个独立的框架，本身不提供具体的用户登录服务，但是它提供了一种管理用户会话的机制，以便管理用户会话并进行用户会话的评论。ROA在com.wondersgroup.roa.session包中定义了两个用于会话管理的接口：

|  |  |
| --- | --- |
| 名称 | 说明 |
| Session | 会话对象，该接口没有定义任何的方法，是一个标签接口； |
| SessionManager | 会话管理器，该接口拥有3个方法，分别是：   * public addSession(String sessionId, Session session) * public Session getSession(String sessionId) * public void removeSession(String sessionId)。 |

在实际应用中，Session一般会拥有用户相关的数据，如用户信息、用户权限等。由于服务平台一般都是工作于分布式环境中，所以一般不适合直接使用Web服务器的HTTPSession来管理ROA的会话，应当使用数据库或集中式缓存服务器等设施来管理会话。

创建会话、删除会话一般对应用户登录、用户退出两个服务，这两个具体的服务由服务开放平台开发，在服务方法中利用SessionManager提供的会话管理功能注册和删除会话，以便让ROA获知会话的状态。

ROARequestContext中提供了三个管理会话的方法，如下：

|  |  |
| --- | --- |
| 方法名称 | 说明 |
| void addSession(String sessionId, Session session) | 将会话添加到会话管理器中； |
| Session getSession(String sessionId)： | 根据sessionId获取会话对象； |
| void removeSession(String sessionId)： | 根据sessionId从会话管理器中移除会话。 |

## 注册SessionManager

系统默认实现了一个简单的DefaultSessionManager，它提供了一个基于一个Map管理的SampleSessionManager会话管理器，这里我们通过SampleSessionManager演示会话管理器的过程。在实际应用中，我们会提供使用基于集中式缓存（如memcached）和基于NO-SQL数据库（redis）等技术来管理会话。

当我们有了SessionManager的实现以后，还是基于Spring Framework IoC方式通过<roa:annotation-driven/>属性来装配到ROA中去。

|  |
| --- |
| <roa:annotation-driven session-manager="defaultSessionManager"/>  <bean id="defaultSessionManager"  class="com.wondersgroup.roa.session.impl.DefaultSessionManager"/> |

## 开发Login和Logout服务

基于ROA的服务开放平台应当至少包括两个服务，即登录和退出平台的服务。这两个服务方法也应该最先开发出来。下面是简单的登录和退出的服务方法实现：

|  |
| --- |
| //指定该服务方法无须会话  @ServiceMethod(method = "user.logon",version = "1.0",needInSession = NeedInSessionType.NO)  public Object logon(LogonRequest request) {  //创建会话对象  SimpleSession session = new SimpleSession();  session.setAttribute("userName",request.getUserName());  request.getROARequestContext().addSession("mockSessionId1", session);  LogonResponse logonResponse = new LogonResponse();  logonResponse.setSessionId("mockSessionId1");  return logonResponse;  }  @ServiceMethod(method = "user.logout",version = "1.0")  public Object logout(ROARequest request) {  //删除会话  request.getROARequestContext().removeSession();  LogoutResponse response = new LogoutResponse();  response.setSuccessful(true);  return response;  } |

简单来说，登录服务是用于创建会话生成sessionId给客户端的，因此登录服务的方法是工作在非会话环境中的。在默认情况下，所有的服务方法都必须工作于会话环境中，也即每次对服务的请求都必须提供sessionId，否则，ROA就会驳回请求，返回错误的报文。如果希望ROA不对服务方法进行会话校验，需要显式将@ServiceMethod的needInSession属性设为NeedInSessionType.NO。