

Rop 开发手册



1.0

Copyright © 2012-2018 陈雄华

1 概述

1.1 快速认识 Rop

Rop 是 Rapid Open Platform 的简称，它不同于一般纯技术型的 Web Service 框架（如 CXF，Jersey 等），Rop 致力于构建服务开放平台的框架，您可以使用 Rop 开发类似于淘宝服务开放平台这样的服务平台。Rop 充分借鉴了当前大型网站的服务开放平台的设计思路，汲取了它们成功的实践经验，对服务开放平台的很多应用层领域问题给出了解决方案，开发者可以直接使用这些解决方案，也可以在此基础上进行个性化扩展。

Rop 功能架构

CXF 和 Jersey 是纯技术纯的 Web Service 框架，而在 Rop 中，Web Service 只是核心，它提供了开发服务平台的诸多领域问题的解决方案：如应用认证、会话管理、安全控制、错误模型、版本管理、超时限制等。

下面通过图 1 了解一下 Rop 框架的整体结构：

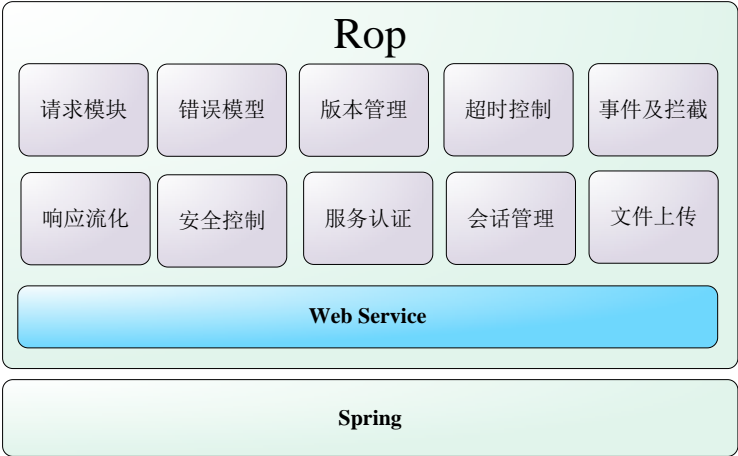


图 1 Rop 框架

从图 1 中，可以看到 Rop 所提供的大部分功能都是偏“应用层”的，传统技术型的 Web Service 框架是不会僭越到这些“应用层”的问题的。但是，在实际开发中，这些应用层的问题不但不可避免，而且非常考验开发者的设计经验，此外，这些工作还会占用较大的开发工作量。Rop 力图让开发者从这些复杂的工作中解脱出来，让他们可以真正聚焦服务平台业务逻辑的实现上。

Rop 技术架构

Rop 在技术实现上充分借鉴了 Spring MVC 的框架设计理念和实现技术，首先 RopServlet 类似于 Spring MVC 的 DispatcherServlet，是 Rop 的门面 Servlet，负责截获 HTTP 服务请求转由 Rop 框架处理。具体技术架构通过图 2 描述：

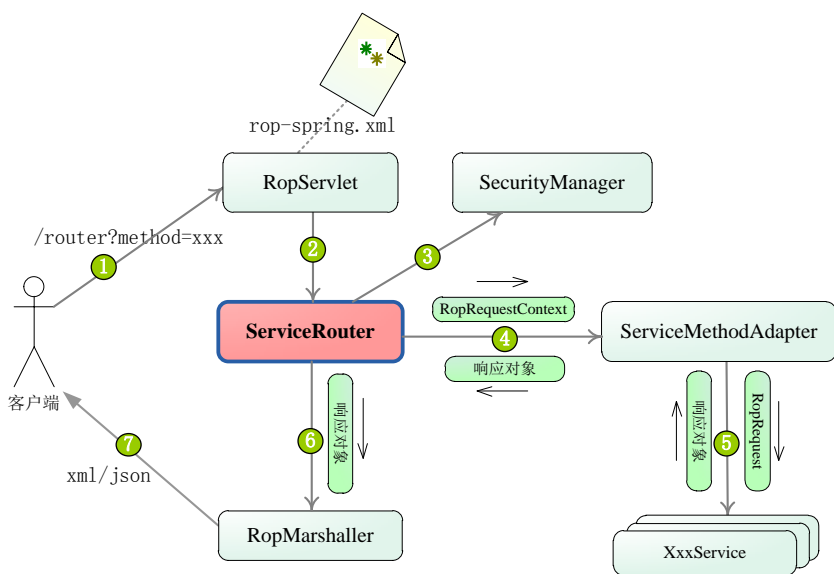


图 2 Rop 技术框架

Rop 的配置信息统一在 Spring 配置文件中通过 rop Schema 命名空间定义。ServiceRouter 是 Rop 框架的核心，它负责协调各组件的交互并最终完成服务处理的工作。RopServlet 在启动时会从 Spring 容器中搜索出 ServiceRouter 的 Bean 实例并注册之。

在服务请求到达后，RopServlet 截获请求并转交给 ServiceRouter 处理，ServiceRouter 将服务请求封装成一个的 RopRequestContext 对象，RopRequestContext 包含了服务请求的所有信息。而后，ServiceRouter 使用 SecurityManager 检查服务请求的安全性，只有通过了 SecurityManager 的安全检查，才会调用目标服务处理方法执行服务，否则将阻止请求并返回错误响应信息。

完成 SecurityManager 的安全检查后，ServiceRouter 通过 ServiceMethodAdapter 对目标的服务方法发起调用。由于具体服务方法的签名各不相同，因此必须采用反射机制进行适配调用。当返回响应对象后，ServiceRouter 使用 RopMarshaller 将响应对象编组为特定的响应报文返回给客户端。

回顾一个第 7 章的 7.1 小节，您会发现 Rop 的顶级框架接口类在 Spring MVC 中都能找到对应的对象：RopServlet 对应 DispatcherServlet，ServiceMethodAdapter 对应 HandlerAdapter，RopMarshaller 对应 ViewResolver，而 ServiceRouter 承担了 HandlerMapping 和部分 DispatcherServlet 的角色。因此，如果您在学习 Spring MVC 的框架后，理解 Rop 框架的实现原理将变得非常轻松。

1.2 使用 Rop 开发一个服务

将 Rop 项目克隆到本地

由于 Rop 托管在 github (www.github.com)中，为了获取最新的 Rop 项目，必须在您的系统中安装 Git 客户端软件，我们推荐使用 msysgit，它能够让我们在 Windows 系统中像 Linux 一样使用 Git。

从 <http://code.google.com/p/msysgit/> 下载并安装 git 客户端 msysgit，然而在开始菜单中找到并打开 Git Bash，在命令窗口运行如下命令：

```
cd /d/agileSpring/
git clone git://github.com/itstamen/rop.git
```

第一行命令将当前工作目录移到某个系统目录下，需要注意的是：由于 msysgit 是在 Windows 中模拟的 Linux 环境，所以 D:/ 对应为 /d/。第二行命令，从 github 中克隆一个 Rop 项目到您本地机中。

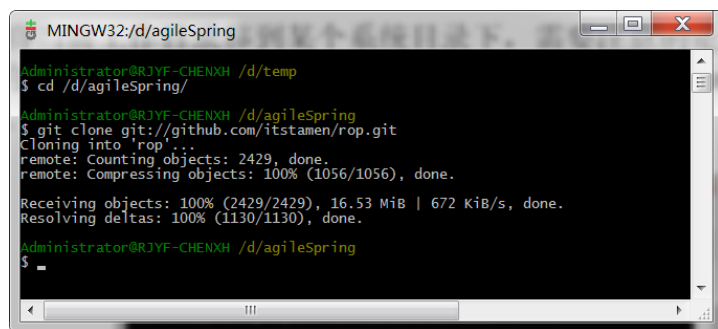


图 3 克隆 Rop 到本地系统

使用以下的 Maven 命令构建 rop 和 rop-sample 项目，打开 DOS 窗口，移到 rop 及 rop-sample 项目的 pom.xml 所在的目录，执行构建命令：

```
mvn clean install
```

还可以通过如下 Maven 命令，启动 rop-sample 项目（首先移到 rop-sample 的 pom.xml 所在的目录）：

```
mvn jetty:run
```

如果您不想下载最新的 Rop 项目，上面的过程就可以免除了，您直接在项目的 pom.xml 中引入 Rop 类包就可以了，如下所示：

```
<dependency>
  <groupId>com.bookegou</groupId>
  <artifactId>rop</artifactId>
  <version>1.0</version>
</dependency>
```

Rop 的发布包已经发布到 Maven 的核心仓库中（org.sonatype.oss），因此您可以直接使用在 pom.xml 引用即可。


开发一个服务方法

rop-sample 项目中有一个 com.rop.sample.UserService 的服务类，我们就通过这个服务类了解开发基于 Rop 的 Web Service 服务的过程。

代码清单 1 UserService.java

```
package com.rop.sample;
```

```
import com.rop.RopRequest;
import com.rop.annotation.NeedInSessionType;
import com.rop.annotation.ServiceMethod;
import com.rop.annotation.ServiceMethodBean;
import com.rop.sample.response.LogonResponse;
import com.rop.session.SimpleSession;
```

@ServiceMethodBean ①  标注 Rop 的注解, 使 UserService 成为一个 Rop 的服务 Bean

```
@ServiceMethod(method = "user.getSession", version = "1.0", | ② 使该服务方法成为一个
needInSession = NeedInSessionType.NO) Web Service 的方法。
public Object getSession(LogonRequest request) {
```

//创建一个会话

```
SimpleSession session = new SimpleSession();
session.setAttribute("userName",request.getUserName());
request.getRopRequestContext().addSession("mockSessionId1", session);
```

//返回响应

```
LogonResponse logonResponse = new LogonResponse();
logonResponse.setSessionId("mockSessionId1");
return logonResponse;
```

```
}
```

由于 ServiceMethodBean 注解已经标注了 Spring 的 @Service 注解（即 org.springframework.stereotype.Service），因此标注 @ServiceMethodBean 的类也相当于标注了 @Service，Rop 扩展 Spring @Service 的目的是为了引入新的功能特性。

在类方法处标注 @ServiceMethod，即可将该方法发布成一个 Rop 的 Web Service 服务。@ServiceMethod 注解拥有丰富的参数，method 值是必须的，它用于指定服务方法名称，version 用于指定服务方法的版本号，needInSession 用于说明该服务方法是否要工作在会话环境中。更多参数的说明，详见本章后续内容，这里只要了解这么多就可以了。

由于任何一个服务方法都是由请求/响应对构成的，所以在 Rop 的服务方法的签名是有约定的：入参必须为 RopRequest 接口或其子类，出参可以是任何标注了 JSR 222 注解的对象。这种约定在一定程度上限制了方法签名的灵活性，但是由于 Rop 强烈建议对所有请求参数都做服务端校验，因此把参数封装成一个 Java 类并在类属性中标注 JSR 303 注解，就可以在参数绑定时实施数据校验了。来看一下 LogonRequest 的类：

代码清单 2 LogonRequest.java：请求对象

```
package com.rop.sample.request;
import com.rop.AbstractRopRequest;
import com.rop.annotation.IgnoreSign;
import javax.validation.constraints.Pattern;

public class LogonRequest extends AbstractRopRequest{
```

```
@Pattern(regexp = "\\w{4,30}")
```

```
private String userName;
```

```
@IgnoreSign
```

```
@Pattern(regexp = "\\w{6,30}")
```

```
private String password;
```

```
//get/setter
```

```
}
```

Rop 采用“契约优于配置”的原则：请求参数按名称匹配的方式自动绑定到请求对象的属性上。在请求对象类中，Rop 使用 JSR 303(Bean Validation)注解描述请求参数的校验规则，如果请求参数值违反了校验规则，Rop 将驳回服务请求，直接返回相应的错误报文给客户端。

服务方法必须返回一个响应对象，Rop 框架会将其转换成响应报文。在这个例子中，如果服务正常执行，将返回一个 LogonResponse 的对象。LogonResponse 类定义如下：

代码清单 3 LogonResponse.java：请求入参对象

```
package com.rop.sample.response;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlRootElement;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(name = "logonResponse")
public class LogonResponse{

    @XmlAttribute
    private String sessionId;

    ...
}
```

Rop 使用 JSR 222(即 JAXB)注解描述响应类到响应报文的转换映射关系，响应对象最终将以响应报文的形式（XML 或 JSON）返回给客户端。

如何在 Spring 中配置 Rop

Rop 基于 Spring 框架工作，可以看成是一个 Spring 的子项目。Rop 提供了一个扩展的 Spring Schema 命令空间，使用 Rop Schema 在 Spring 配置文件配置好 Rop 环境非常方便：

代码清单 4 sampleRopApplicationContext.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"
    xmlns:rop="http://www.bookegou.com/schema/rop"
```

```
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.1.xsd
```

```
http://www.bookegou.com/schema/rop
```

```
http://www.bookegou.com/schema/rop/rop-1.0.xsd">
```

① 引入 Rop Schema 定义文件

```
<!--② 扫描Spring Bean-->
```

```
<context:component-scan base-package="com.rop.sample"/>
```

```
<!--③启动Rop框架 -->
```

```
<rop:annotation-driven/>
```

```
</beans>
```

首先，引入 Rop 的 Schema 命名空间，如①处所示。由于 Rop 的服务类必须是一个 Bean，所以需要声明 Spring 的扫描器，将标注了 Spring Bean 注解的类加载为 Spring 容器中的 Bean。由于 @ServiceMethodBean 注解本身标注了 Spring 的 @Service，所以所有标注了 @ServiceMethodBean 的类也会自动成为 Spring 的 Bean。最后，我们通过一个简单 <rop:annotation-driven/> 即可启动 Rop 框架，如③所示。

在 web.xml 配置 Rop

由于客户端需要通过 HTTP 访问 Rop 服务，因此 Rop 必须依附于一个 Web Servlet 容器。和 Spring MVC 的 DispatcherServlet 类似，Rop 提供了一个 com.rop.RopServlet，在 web.xml 中配置好 RopServlet，这样 Rop 就可以接收 HTTP 的服务请求了：

代码清单 5 sampleRopApplicationContext.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_2_5.xsd"
  version="2.5">
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      classpath*:sampleRopApplicationContext.xml
    </param-value>
  </context-param>

  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
  <listener>
    <listener-class>org.springframework.web.util.Log4jConfigListener</listener-class>
  </listener>

  <servlet>
```



```

<servlet-name>rop</servlet-name>
<servlet-class>
    com.rop.RopServlet
</servlet-class>
<load-on-startup>2</load-on-startup>
</servlet>

```

①

定义一个 *RopServlet*, 并
指定其映射的 URL

```

<servlet-mapping>
    <servlet-name>rop</servlet-name>
    <url-pattern>/router</url-pattern>
</servlet-mapping>

```

```

</web-app>

```

由于 Rop 是基于 Spring 工作的, 因此首先必须在 web.xml 中配置一个 Spring 容器, 然后再通过 RopServlet 指定 Rop 的工作的路径。



提示

值得注意的是, Rop 依赖于 Spring 框架而非 Spring MVC 框架, 因此您的 Web 应用框架是可以自由选择的, 如 Struts、JSF 等。

访问 Rop 服务

在 rop-sample 项目中, 我们已经配置 maven-jetty-plugin 插件, 因此在 IDEA 的 Maven Project 中找到 ropSample->Plugins->jetty->jetty:run, 右键菜单中运行 Run, 启动 rop-sample 项目。在控制台中, 您将可以看到 rop-sample 启动的输出信息。

下面开发一个访问 rop-sample 服务的客户端程序:

代码清单 6 UserServiceClient.java

```

package com.rop.sample;

```

```

import com.rop.client.CompositeResponse;
import com.rop.client.DefaultRopClient;
import com.rop.client.RopClient;
import com.rop.client.ClientRequest;
import com.rop.sample.request.LogonRequest;
import com.rop.sample.response.LogonResponse;
import org.testng.annotations.Test;
import static org.testng.Assert.*;

```

```

public class UserServiceClient {

```

```

    public static final String SERVER_URL = "http://localhost:8088/router";
    public static final String APP_KEY = "00001";
    public static final String APP_SECRET = "abcdeabcdeabcdeabcdeabcde";
    private RopClient ropClient =
        new DefaultRopClient(SERVER_URL, APP_KEY, APP_SECRET);

```

准备好 Rop 客户端对象, 入参
为服务的 URL, 本应用对应的
应用键和密钥。

①

```

@Test

```

```
public void createSession() {  
  
    LogonRequest ropRequest = new LogonRequest(); ②  
    ropRequest.setUserName("tomson");  
    ropRequest.setPassword("123456");  
  
    CompositeResponse response = ropClient.buildClientRequest()  
        .get(ropRequest, LogonResponse.class, "user.getSession", "1.0");③  
  
    assertNotNull(response);  
    assertTrue(response.isSuccessful());  
    assertNotNull(response.getSuccessResponse());  
    assertTrue(response.getSuccessResponse() instanceof LogonResponse);  
    assertEquals(((LogonResponse) response.getSuccessResponse()).getSessionId(),  
        "mockSessionId1");  
}  
}
```

构造请求对象

对服务发起调用并获取响应结果

大凡 Web Service 框架都会提供服务的客户端，DefaultRopClient 即是 Rop 提供了客户端，您可以非常方便地通过 DefaultRopClient 以面向对象的方式访问服务获取响应对象，无需关心服务请求和响应报文的底层细节。

运行以上测试方法，通过报文抓取工具，我们可以看到“真实”的底层通信细节，如下图 4 所示：

请求报文

响应报文

GET /
router?appKey=00001&method=user.getSession&v=1.0&format=xml&locale=zh_CN&userName=tomson&password=123456&sign=51C4A9C9970742CFD178EAEBABAC826F86365DAB HTTP/1.1

Accept: text/plain, application/json, */*
User-Agent: Java/1.6.0_29
Host: localhost:8080
Connection: keep-alive

HTTP/1.1 200 OK
Content-Type: application/xml; charset=UTF-8
Transfer-Encoding: chunked
Server: Jetty(6.1.5)

64
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<logonResponse sessionId="mockSessionId1"/>

0

图 4 服务请求和响应报文

由此可见，DefaultRopClient 会自动将 LogonRequest 请求对象拼装成一个请求 URL，同时，在接收到响应报文后，会将响应报文反编组成一个响应对象。

2 请求服务模型

2.1 传统 Web Service 请求模型

请求模型设计的好坏将直接影响服务调用的难易程度，设计良好的请求模型可以让服务调用成为随时随地、信手拈来的事。此外，还能使服务接口清晰化，降低开发者理解服务的难度。我们先来了解一下传统 Web Service 的请求模型。

SOAP 请求模型

Web Service 基本上都是使用 HTTP 传输协议进行交互的，服务的响应报文一般支持 XML 和 JSON 两种格式，但 Web Service 服务请求模型却各有千秋。

传统的 Web Service 采用 SOAP 请求报文，任何服务都对应一组 SOAP 请求/响应报文，服务的调用及报文解析都比较麻烦。举例来说，即使是调用一个诸如查看当天天气的简单服务，该服务仅有一个 city 的参数，在 SOAP 的世界里，您也必须将其封装成一个复杂的 SOAP 请求报文才行。一般情况下，不借助 CXF、Axis 这类框架你很难访问 SOAP。

REST 请求模型

但是，很多情况下，开发者往往希望自由地随时随地访问服务，比如，通过一个形如 `http://www.xxx.com/weather/{city}` 的 URL 就可以访问服务获取响应。把服务看成一个类似于文档、图片式的普通资源，通过一个唯一的 URL 进行定位和调用——这就是现在方兴未艾的 REST Web Service 的中心思想。

REST Web Service 充分挖掘了 HTTP 通讯协议的内涵，借助 HTTP 方法（如 GET、POST、PUT、DELETE 等）及合理设计的服务 URL，让 Web Service 达到不言自明的效果。

豆瓣网的 API 就是采用标准的 REST Web Service 开发的，来看一个获取图书信息的 API：

`http://api.douban.com/book/subject/isbn/{isbnID}`

该服务使用 HTTP 的 GET 方式调用，说白了就是您可以简单地在浏览器地址栏中敲入以下 URL，就可以发起服务调用：

`http://api.douban.com/book/subject/isbn/9787508630069`

以上请求将获得《史蒂夫·乔布斯传》这本书的服务响应报文，它是一个 XML 报文，您既可以在浏览器中预览，也可以写一个程序消费这个响应报文，完成您要干的事情。这种服务调用方式，对于服务调用者非常亲切，因为它和访问一个网页并无二致。如果要学习 REST Web Service 的设计，豆瓣网的 API 就是不错的学习案例，我们来欣赏一下豆瓣网其它几个 API：

- GET `http://api.douban.com/movie/subject/{subjectID}`：获取某个专题的信息，GET 表示使用 HTTP 请求方法，下同；
- GET `http://api.douban.com/people/{userID}`：获取某个用户的信息；
- GET `http://api.douban.com/people?q=douban&start-index=10&max-results=5`：搜索

用户，用户名通过 `q` 参数传递，其它两个参数是分页控制参数；

■ `DELETE http://api.douban.com/review/{reviewID}`：删除某条评论。

采用 REST 请求模型发布的服务接口很清晰化、调用也很简单，REST 服务已经模糊了服务和网页资源的界限。简单就是最好的，从这个意义上说 REST 确实优于 SOAP，开发者也纷纷用脚做出了投票，弃 SOAP 之暗而投 REST 之明。

REST 在扛起挑战 SOAP 大旗时，对 SOAP 的战斗檄文是：复杂，笨重，EJB 死灰复燃。但是，当 REST 得天下后，我们发现 REST 本身也存在一些刻板的东西。

首先，经典的 REST 对 HTTP 请求方法的使用过于教条化：新增、更改、删除、获取资源的服务分别对应 POST、PUT、DELETE 和 GET 的 HTTP 请求方法。一般的 Web 服务器和浏览器都只支持 GET 和 POST 这两种 HTTP 请求方法，所以在实际应用中，REST 希望充分挖掘 HTTP 请求方法能力的倡议遭遇了困难。

其次，REST 提倡为每个服务设计一个“达意”的 URL，让服务的 URL 望文生义。从可读性、清晰化的角度上看，REST 的这个建议是非常值得称赞的。但是，服务的消费者主体是程序，让每个服务对应不同的 URL，反而让客户端程序不好写。

综上所述，当前如日中天的 REST Web Service 自身也存在一些待改进的地方。淘宝的 TOP 的请求模型可以看成是 REST 的变体，首先，TOP 提供的所有服务的 URL 都是一样的：即为 `http://gw.api.taobao.com/router/rest`，使用 `method` 参数指定服务 API 名称，再通过其它参数指定服务的入参。由于平台所有服务的 URL 都相同，不同的服务方法通过 `method` 参数区分，反而让服务的调度变得简单了。

2.2 Rop 请求模型

Rop 请求模型的设计直接借鉴了 TOP 的思想，服务开放平台的所有服务 URL 是相同的，请求参数分为系统级参数和业务级参数两部分，系统级参数是所有服务 API 都拥有的参数，而业务级参数由具体服务 API 定义。

统一服务 URL

采用 Rop 的服务开放平台，其所有的服务都使用统一的 URL，Rop 通过 `method` 系统级参数将请求路由到指定的服务方法中完成服务受理。如何设置这个统一的服务 URL 呢？答案很简单，即是通过 RopServlet 的 `<servlet-mapping>` 进行定义，如代码清单 5 所示。

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

系统级参数

系统级参数是由开放平台定义的一组参数，每个服务都拥有这些参数，用以传送框架级的参数信息。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。Rop 共有 7 个系统级参数，在表 1 中说明：

表 1 系统级参数

参数名称	是否必须	参数说明
appKey	是	应用键，开放平台用以确定客户端应用的身份，如 000001，000002 等。应用键对应一个密钥 secret。要基于服务平台开发应用，必须事先通过申请获取 appKey/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；
v	是	服务方法的版本号，如 1.0、2.0 等。一个具体的服务方法上 method+v 两者唯一确定。因此服务平台必须保证所有服务的 method+v 的唯一性。
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。

locale、format 这两个系统级参数的功用是不言自明的，而其它的系统级参数由于涉及到服务开放平台很多的领域性问题，需要一些背景知识的铺垫，因此我们将在后续内容进行专门的介绍。

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<rop:sysparams/>进行定义，如下所示：

代码清单 7 sampleRopApplicationContext.xml：定义系统级参数名

```
<rop:sysparams
    format-param-name="messageFormat"
    appkey-param-name="app_key"/>
```

业务级参数

业务级参数，顾名思义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。Rop 根据参数名和 RopRequest 类属性名相等的契约，将业务级参数绑定到 RopRequest 中。

如代码清单 2 的 LogonRequest 定义了两个 userName 和 password 两个属性，Rop 就会将 HTTP 请求参数值绑定到 LogonRequest 对象的同名属性中。

2.3 参数数据绑定与校验

参数数据绑定

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 URL 发送若干个请求参数（包括系统级和业务级的参数）。Rop 框架在接收到这些请求参数后，就会将其绑定到 RopRequest 请求对象中，服务方法可通过这个 RopRequest 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。图 5 描述了请求参数的转换过程：

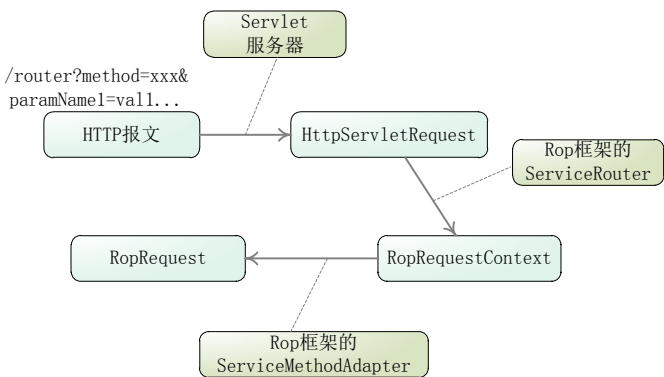


图 5 请求数据转换过程

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 RopServlet 转交给 Rop 框架，Rop 框架将 HttpServletRequest 转换成一个 RopRequestContext 对象。接着，ServiceRouter 将 RopRequestContext 传给 ServiceMethodAdapter，ServiceMethodAdapter 在内部将 RopRequestContext 转换成 RopRequest 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，Rop 都会在内部创建一个 RopRequestContext 实例，它包含了所有的请求数据信息。下面，我们来了解一下 RopRequestContext 接口的方法：

- String getAppKey(): 获取 appKey 系统级参数的值。RopRequestContext 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，仅有两个枚举值，即 GET 和 POST。这也说明，Rop 仅支持 GET 和 POST 两个 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 Rop 使用了一些技巧，以保证后端服务获取的 IP 是客户端的 IP。具体实现可以参见 com.rop.ServletRequestContextBuilder#getRemoteAddr(HttpServletRequest request)的实现；
- Object getRawRequestObject(): 获取原请求对象，即服务请求对应的 HttpServletRequest 对象；
- Map<String, String> 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；
- RopContext getRopContext(): 获取 Rop 框架上下文的信息。RopContext 之于 Rop 框架相当于 ServletContext 之于 Servlet 容器，它包含了很多 Rop 框架的运行期信息，所有 Rop 的服务方法都注册在 RopContext 中。

概括来说, `RopRequestContext` 为每个系统级参数都提供了一个方法, 如 `getAppKey()`、`getMethod()`等。对于业务级参数, 则可以使用 `RopRequestContext` 的 `getParamValue("<参数名>")`获取。此外, `RopRequestContext` 还提供了获取原始请求对象、客户端 IP 等方法。

所有服务方法的入参都是 `RopRequest` 接口或其实现类, `RopRequest` 接口仅有一个方法:

```
RopRequestContext getRopRequestContext();
```

`RopRequest` 的实现类负责定义业务级参数对应的属性, 这样, 在服务方法内部, 就可以通过 `RopRequest#getRopRequestContext()` 获取 `RopRequestContext`, 再通过 `RopRequestContext` 访问到系统级参数了。而业务级参数是可通过 `RopRequest` 实现类的属性获取。

在 1.2 小节的 `getSession()`服务方法中, 我们定义了一个 `LogonRequest` 的请求对象, 它就是一个实现了 `RopRequest` 接口的对象, 在服务方法内部, 可以通过 `LogonRequest` 访问到系统级参数、业务级参数及其它相关的信息, 如下所示:

代码清单 8 UserService.java : 访问参数

```
@ServiceMethod(method = "user.getSession",needInSession = NeedInSessionType.NO)
public Object getSession(LogonRequest request) {

    //①访问系统级参数
    String appKey = request.getRopRequestContext().getAppKey();

    //②-1 访问业务级参数: 通过类属性
    String userName1 = request.getUserName();
    //②-2 访问业务级参数: 通过RopRequestContext获取
    String userName2 = request.getRopRequestContext().getParamValue("userName");

    //③获取其它信息
    String ip = request.getRopRequestContext().getIp();
}
```

通过上面的实例, 我们可以知道通过 `RopRequest` 可以很方便地获取系统级参数、业务级参数及客户端的相关信息。

参数数据校验

由于应用客户端和服务平台都是服务报文进行通信的, 所有的请求参数都以字符串的形式传送过来。为了保证服务得到正确执行, 必须事先对请求参数进行数据合法性验证, 只有在服务请求所有参数都符合约定的情况下, 服务平台才执行具体的服务操作, 否则直接驳回请求, 返回错误的报文。

数据校验从责任主体上看, 可分为客户端校验和服务端校验两种。对于一个封闭式的应用软件来说, 由于服务端和客户端都是一体化开发的, 为了减少开发工作量, 有时仅需要进行客户端校验就可以了。但是, 服务开放平台的潜在调用者是不受限的, 应用开发者可基于服务平台开发出众多丰富多彩的应用, 在这种场景下, 服务端校验是必不可少的。

服务请求参数的校验是开放平台的一项重要的基础功能, `Rop` 和 `Spring MVC` 一样使

用 JSR 303 注解定义参数的校验规则，当请求数据违反校验规则时，直接返回对应的错误报文，只有所有请求参数都通过合法性验证后，才调用目标服务方法。

下面的 CreateUserRequest 使用了 JSR 303 注解的，来看一下具体的使用方法：

代码清单 19 CreateUserRequest.java：使用 JSR 303 对业务级参数进行校验

```
public class CreateUserRequest extends AbstractRopRequest {

    @Pattern(regexp = "\\w{4,30}")
    private String userName;

    @IgnoreSign
    @Pattern(regexp = "\\w{6,30}")
    private String password;

    @DecimalMin("1000.00")
    @DecimalMax("100000.00")
    @NumberFormat(pattern = "#,###.##")
    private long salary;

    ...
}
```

对于系统级的参数，Rop 本身会负责校验，开发者仅需关注业务级参数的校验即可。当请求的参数违反校验规则后，Rop 将把这些错误“翻译成”对应的错误报文。假设 salary 格式不对，其对应的错误报文为：

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<error code="33">
    <message>非法的参数</message>
    <solution>请查看根据服务接口对参数格式的要求</solution>
    <subErrors>
        <subError code="isv.parameters-mismatch:salary-and-yyy">
            <message>传入的参数salary和aaa不匹配</message>
        </subError>
    </subErrors>
</error>
```

关于错误处理模型及报文格式，我们将在 10.6 小节中专门讲解。

2.4 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？Rop 框架支持将 XML 或 JSON 格式的参数值透明地绑定到 RopRequest 的复合属性中。

我们通过 rop-sample 实例项目的 UserService#addUser(CreateUserRequest request)讲解 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：

代码清单 10 CreateUserRequest.java：复合属性


```
package com.rop.sample.request;
import javax.validation.Valid;
...

public class CreateUserRequest extends AbstractRopRequest {

    @Pattern(regexp = "\\w{4,30}")
    private String userName;

    ...

    @Valid
    private Address address;
}
```

① 可绑定 XML 或 JSON 的复合属性，必须打上@Valid 注解进行数据校验

Address 是一个复合对象属性，它的类结构对应 XML 的结构：

代码清单 11 CreateUserRequest.java：复合属性

```
package com.rop.sample.request;

import javax.validation.constraints.Pattern;
import javax.xml.bind.annotation.*;
import java.util.List;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(name = "address")
public class Address {

    @XmlAttribute
    @Pattern(regexp = "\\w{4,30}")
    private String zoneCode;

    @XmlAttribute
    private String doorCode;

    @XmlElementWrapper(name = "streets")
    @XmlElement(name = "street")
    private List<Street> streets;
}
```

① 使用 JSR 222 注解，定义了基于属性名进行数据绑定的规则。

② 使用 JSR 222 定义数据绑定规则，使用 JSR 303 注解定义数据校验规则。

③ 使用 JSR 222 注解指定列表数据的绑定规则

JSR 222 标准规范（也即 JAXB），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，Rop 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 注解两者珠联璧合，Rop 很完美地解决了请求数据绑定和数据校验的问题。

开发者仅需要在 CreateUserRequest 中标注上注解，无需做任何其它的开发工作，就可以绑定客户端的 XML 和 JSON 数据了。rop-sample 项目的 UserServiceRawClient 有一个

testServiceXmlRequestAttr()测试方法，它演示了 XML 参数数据绑定的场景：

代码清单 12 UserServiceRawClient.java : XML 请求参数

```
@Test
public void testServiceXmlRequestAttr() {
    RestTemplate restTemplate = new RestTemplate();
    MultiValueMap<String, String> form = new LinkedMultiValueMap<String, String>();
    form.add("method", "user.add");
    form.add("messageFormat", "xml");① ← 指定消息报文格式为 XML 格式
    ...

    form.add("address",
        "<address zoneCode=\"0001\" doorCode=\"002\">\n" + ② ← XML 格式的参数数据
        "  <streets>\n" +
        "    <street no=\"001\" name=\"street1\"/>\n" +
        "    <street no=\"002\" name=\"street2\"/>\n" +
        "  </streets>\n" +
        "</address>");

    //手工对请求参数进行签名
    String sign = RopUtils.sign(form.toSingleValueMap(), "abcdeabcdeabcdeabcdeabcde");
    form.add("sign", sign);

    //调用服务获取响应报文
    String response = restTemplate.postForObject(SERVER_URL, form, String.class);
}
```

如果有某个请求参数的内容是 XML，必须将报文格式设置成 xml，如①所示。在②处，address 的参数值即是一个 XML 格式的字符串，它将正确绑定到 CreateUserRequest 的 address 属性中。

相似的，下面的 testServiceJsonRequestAttr()测试方法则使用 JSON 格式为 address 参数提供数据：

代码清单 13 UserServiceRawClient.java : XML 请求参数

```
public void testServiceJsonRequestAttr() {
    RestTemplate restTemplate = new RestTemplate();
    MultiValueMap<String, String> form = new LinkedMultiValueMap<String, String>();
    form.add("method", "user.add");
    form.add("messageFormat", "json");① ← 指定消息格式为 JSON 格式

    form.add("address",
        "{\"zoneCode\":\"0001\", \"\n" +
        "\"doorCode\":\"002\", \"\n" +
        "\"streets\": [{\"no\":\"001\", \"name\":\"street1\"}, \n" +
        "  {\"no\":\"002\", \"name\":\"street2\"} ] }"; ② ← JSON 格式的参数数据

    String sign = RopUtils.sign(form.toSingleValueMap(), "abcdeabcdeabcdeabcdeabcde");
    form.add("sign", sign);
}
```

```
String response = restTemplate.postForObject(SERVER_URL, form, String.class);
}
```

将报文格式设置为 json，即可支持 JSON 格式参数数据的绑定。在默认情况下，Rop 不允许同时使用 XML 和 JSON，仅能两者取一：请求和响应报文要么是 XML，要么是 JSON。

2.5 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，Rop 允许您通过注册自定义转换器支持自定义格式的参数。Spring 3.0 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 的数据类型转换体系是高度可扩展的，Rop 就是基于 Spring 的类型转换体系实施参数数据绑定的工作，因此，Rop 允许开发者定义自己的类型转换器。

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 Rop 需要双向转换功能：在服务端将参数绑定到 `RopRequest` 时，将 S 转换成 T，而在客户端将 `RopRequest` 流化成请求报文时，需要将 T 转换成 S。因此，Rop 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：

```
package com.rop.request;
import org.springframework.core.convert.converter.Converter;
public interface RopConverter<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，`RopConverter<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 `RopConverter<S, T>` 接口并实现 S 和 T 相互转换的逻辑即可。`rop-sample` 中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：

代码清单 14 TelephoneConverter.java：双向类型转换器

```
package com.rop.sample.request;

import com.rop.request.RopConverter;
import org.springframework.core.convert.converter.Converter;
import org.springframework.util.StringUtils;

public class TelephoneConverter implements RopConverter<String, Telephone> {
```

```
@Override
public Telephone convert(String source) {① ← 将格式化字符串转换为 Telephone
```

```
    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;
    }
}
```

```
@Override
public String unconvert(Telephone target) {② ← 将 Telephone 转换为格式化字符串
```

```
    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 `Rop` 中，以便 `Rop` 在进行参数数据绑定时利用这个转换器。

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

代码清单 15 sampleRopApplicationContext.xml：注册自定义类型转换器

```
<rop: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.rop.sample.request.TelephoneConverter"/>
        </set>
    </property>
</bean>
```

```

    </property>
</bean>

```

CreateUserRequest 中拥有一个 Telephone 的属性：

代码清单 16 CreateUserRequest.java

```

public class CreateUserRequest extends AbstractRopRequest {

    @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，Rop 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：

代码清单 17 UserServiceClient.java:测试自定义类型转换器

```

@Test
public void testCustomConverter() {

    ropClient.addRopConvertor(new TelephoneConverter()); ①

    CreateUserRequest request = new CreateUserRequest();
    request.setUserName("tomson");
    request.setSalary(2500L);
    Telephone telephone = new Telephone();
    telephone.setZoneCode("0592");
    telephone.setTelephoneCode("12345678");

    CompositeResponse response = ropClient.buildClientRequest()
        .post(request, CreateUserResponse.class, "user.add", "1.0");

    assertNotNull(response);
    assertTrue(response.isSuccessful());
    assertTrue(response.getSuccessResponse() instanceof CreateUserResponse);
}

```

注册转换器，实现 Telephone 到格式化字符串的转换。

在①处，RopClient 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 格式的电话号码转为 Telephone 对象。

从上面的分析可知，`RopConverter#unconvert()`是服务于客户端，而`RopConverter#convert()`则服务于服务端。由于客户端和服务端位于不同的JVM中，因此必须各自独立注册`RopConverter`，关于`RopClient`的更多内容，请参见10.12小节。

2.6 请求服务映射

Spring MVC 通过 `@RequestMapping` 注解实现 HTTP 请求到处理方法的映射。类似的，`Rop` 使用 `@ServiceMethod` 注解实现 HTTP 请求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 `RopRequest` 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>/<ropServletUri>?method=user.getSession&v=1.0&...

来看一个具体的例子:

代码清单 18 UserService.java

```

@Service ①
public class UserService {

    @ServiceMethod(method = "user.add", version = "1.0") ②
    public Object addUser(CreateUserRequest request) {
        ...
    }
}

```

服务类必须是一个 Spring 的 Bean

服务方法对应如下的 HTTP 请求:
?method=user.add&v=1.0&...

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 **Rop** 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"): 对应 method=user.add&v=2.0;
- @ServiceMethod(method = "user.add", version = "3.0"): 对应 method=user.add&v=3.0;
- @ServiceMethod(method = "user.get", version = "1.5"): 对应 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：

- **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 或 POST, 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, Rop 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 Rop 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, Rop 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 的 `@Service`, 所以标注了 `@ServiceMethodBean` 的服务类就相当于打上的 `@Service`, 可以被 Spring 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:

代码清单 19 UserService.java : 使用 `@ServiceMethodBean`

```
@ServiceMethodBean(version = "1.0") ①  version 将被本类中所有@ServiceMethod继承。
public class UserService {

    @ServiceMethod(method = "user.add") ②
    public Object addUser(CreateUserRequest request) {
        ...
    }

    @ServiceMethod(method = "user.get", httpAction = HttpAction.GET)③
    public Object getUser(CreateUserRequest request) {
        ...
    }
}
```

②和③处的服务方法的 `version` 都自动设置为 1.0，如果 `UserService` 业务类方法显式指定了 `version` 属性，将会覆盖 `@ServiceMethodBean` 的设置。

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

3 应用授权及验证

3.1 应用键/应用密钥

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 `appKey/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 `appKey`，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 `appKey/secret` 的机制对应用进行身份认证。

3.2 应用键/密钥管理器

由于 Rop 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，Rop 通过 `com.rop.security.AppSecretManager` 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 `AppSecretManager` 实现类就可以了。

`AppSecretManager` 拥有两个接口方法：

- `boolean isValidAppKey(String appKey)`：判断 `appKey` 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。

Rop 默认提供了一个基于文件存储的 `FileBaseAppSecretManager` 实现类，`FileBaseAppSecretManager` 默认使用读取类路径下的 `rop.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 appSecretFile 属性指定位置，appSecretFile 支持 “classpath:” 等 Spring 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 AppSecretManager，可通过<rop:annotation-driven/>的 app-secret-manager 属性进行配置：

```
<rop:annotation-driven app-secret-manager="appSecretManager"/>
<bean id="appSecretManager" class="com.rop.sample.SampleAppSecretManager"/>
```

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

3.3 签名算法

Rop 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：

- (1) 所有请求参数按参数名升序排序；
- (2) 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>…;
- (3) 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>;
- (4) 对该字符串进行 SHA1 运算，得到一个二进制数组；
- (5) 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；
- (6) 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 3 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：

表 2 服务参数列表

系统级参数名称	参数值	业务级参数名称	参数值
appKey	000001	userName	tomson
sessionId	AAAA	age	24
method	user.create	sex	1
v	1.0		
format	xml		
locale	zh_CN		

根据 Rop 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：

age24appKey000001formatxmllocalezh_CNmethoduser.createsessionIdAAAAsex1userNametomsonv1.0

假设，appKey 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.createsessionIdAAAAsex1userN
ametomsonv1.0abcdef
```

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>/<ropServletUri>?appKey=000001&method=user.create&...
&sign=8625FD7EEAE1E68203B48C64DE495792BF59E833
```

3.4 签名功能控制

默认情况下，Rop 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。Rop 允许服务平台开发者开启或关闭签名验证的功能，Rop 提供了 3 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：在 10.4.3 小节中，我们知道 Rop 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，Rop 在服务签名时允许忽略某些参数，提供参数级的签名控制。

平台级控制

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

```
<rop: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，免受恶意客户端的攻击。

服务级控制

在平台级签名功能开启的情况下，Rop 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。

参数级控制

在定义服务方法的 `RopRequest` 类时，只要在 `RopRequest` 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 AbstractRopRequest{

    @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 `RopRequest` 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？`Rop` 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 `RopRequest` 类的属性都将排除在签名算法的参数列表之外。关于 `UploadFile` 的进一步信息，请参见 10.8 小节的内容。

4 服务会话管理

4.1 会话管理概述

一个客户端应用开发出来后，可以有很多具体的使用者。`Rop` 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。

Rop 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。Rop 在 `com.rop.session` 包中定义了两个用于会话管理的接口：

- **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 `addSession(String sessionId, Session session)`、`Session getSession(String sessionId)` 及 `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 服务器的会话管理机制（如 `HttpSession`）来管理 Rop 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 `SessionManagerRop` 提供的会话管理功能注册和删除会话，以便让 Rop 获知会话的状态。

Rop 在 `RopRequestContext` 中提供了 3 个管理会话的方法，介绍如下：

- `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。

4.2 注册会话管理器

`rop-sample` 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，您应该使用基于集中式缓存（如 `memcached`）或数据库（`redis`）等来管理会话。

代码清单 20 `SampleSessionManager`：基于 Map 的会话管理器

```
package com.rop.sample;

import com.rop.session.Session;
import com.rop.session.SessionManager;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

public class SampleSessionManager implements SessionManager{
    private final Map<String, Session> sessionCache =
        new ConcurrentHashMap<String, Session>(128, 0.75f, 32);

    @Override
    public void addSession(String sessionId, Session session) {
        sessionCache.put(sessionId, session);
    }
}
```

使用该 Map 保存 Session 对象。

①

```

@Override
public Session getSession(String sessionId) {
    return sessionCache.get(sessionId);
}

@Override
public void removeSession(String sessionId) {
    sessionCache.remove(sessionId);
}
}

```

开发出会话管理器后，如何将其装配到 Rop 中呢？答案还是通过 `<rop:annotation-driven>` 的属性：

```

<rop:annotation-driven session-manager="sampleSessionManager"/>
<bean id="sampleSessionManager"
    class="com.rop.sample.SampleSessionManager" />

```

4.3 开发登录和退出服务

基于 Rop 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 rop-sample 的登录和退出的服务方法：

代码清单 21 UserService.java:登录及退出服务方法

```

@ServiceMethod(method = "user.login",version = "1.0",
               needInSession = NeedInSessionType.NO) ①
public Object login(LogonRequest request) {

    SimpleSession session = new SimpleSession();
    session.setAttribute("userName",request.getUserName());
    request.getRopRequestContext().addSession("mockSessionId1", session);

    LogonResponse logonResponse = new LogonResponse();
    logonResponse.setSessionId("mockSessionId1");
    return logonResponse;
}

@ServiceMethod(method = "user.logout",version = "1.0")
public Object logout(RopRequest request) {
    request.getRopRequestContext().removeSession();③
    LogoutResponse response = new LogoutResponse();
    response.setSuccessful(true);
    return response;
}

```

① 指定该服务方法无须会话

② 创建会话对象

③ 删除会话

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，Rop 就会驳回请求，返回错误的报文。如果希望 Rop 不对服务方法进行会话校验，需要像①一样，显式将 @ServiceMethod 的

needInSession 属性设为 NeedInSessionType.NO。

客户端在调用登录服务的方法后，可从响应报文中获取 sessionId，这样就可以在后续的服务调带上这个 sessionId，Rop 会根据 sessionId 将对应的 Session 绑定到服务请求上下文 RopRequestContext 中。

5 错误模型

5.1 错误模型概述

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。

Rop 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 Rop 负责处理，对于业务性的错误，开发者按错误模型构造即可。Rop 将开发者从服务错误处理的荆棘丛中解放出来，从而可以将精力集中于具体的业务逻辑的处理上。

Rop 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：

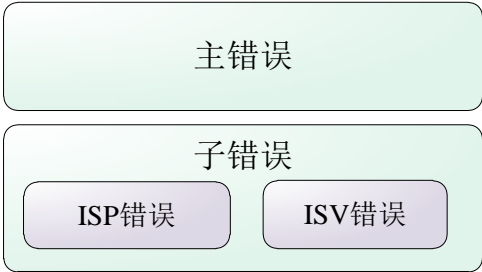


图 6 Rop 错误模型

ISP 代表平台服务提供商一方，如服务内部异常、服务端数据库资源不可用等错误，其责任归属于 ISP，这类错误称为 ISP 错误。ISP 错误的错误编码以“isp.”为前缀，如 isp.xxx-service-unavailable、isp.xxx-service-timeout 等。

ISV 代表基于平台服务开发应用的开发者一方，这类错误产生的原因是由于开发者造成的。ISV 错误的错误编码以“isv.”为前缀，如 isv.invalid-permission、isv.missing-parameter:xxx 等。

当服务发生错误时，Rop 将返回统一格式的错误报文，它由一个主错误和若干个子错误组成。下面即是一个完整的错误报文：

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<error code="33">
```

```
<message>非法的参数</message>
<solution>请查看根据服务接口对参数格式的要求</solution>
<subErrors>
  <subError code="isv.invalid-paramete:salary">
    <message>参数salary无效，格式不对、非法值、越界等</message>
  </subError>
  <subError code="isv.invalid-paramete:userName">
    <message>参数userName无效，格式不对、非法值、越界等</message>
  </subError>
</subErrors>
</error>
```

<error>代表主错误，它拥有一个对应编码，此外还包括错误消息及解决方法。subErrors 元素中包含多个子错误，详细说明是哪些参数违反数据了校验规则，子错误的编码附带了违反校验规则的参数名，对定位错误非常有帮助。

5.2 系统级主错误编码

Rop 内置定义了一套和业务无关的错误代码体系，我们称这些错误为系统级错误。Rop 在 i18n/rop/error 的国际化资源文件（目前提供了 zh_CN 和 en 两个国际化资源文件）中定义了系统级主错误及子错误。

在 Rop 中，主错误都对应一个唯一的数字编码，目前共包括 27 个主错误，通过下表进行说明：

表 3 系统级主错误编码

错误编码	错误说明	错误编码	错误说明	错误编码	错误说明
1	服务不可用	20	缺少 sessionId 参数	29	非法的版本参数
2	开发者权限不足	21	无效的 sessionId 参数	30	不支持的版本号
3	用户权限不足	22	缺少 appKey 参数	31	无效报文格式类型
4	图片上传失败	23	无效的 appKey 参数	32	缺少必选参数
5	HTTP 方法被禁止	24	缺少签名参数	33	非法的参数
6	编码错误	25	无效签名	34	用户调用服务的次数超限
7	请求被禁止	26	缺少方法名参数	35	会话调用服务的次数超限
8	服务已经作废	27	不存在的方法名	36	应用调用服务的次数超限
9	业务逻辑出错	28	缺少版本参数	37	应用调用服务的频率超限

27 个系统级错误可以划分为以下几类：

- 系统级参数错误：这类错误是由于系统级请求参数缺失或不合法引起的，20~31 都是这一类型的错误；
- 业务级参数错误：业务级参数缺失或不合法而引起的错误，如 32 和 33；
- 服务访问超限错误：客户端的服务调用超过配额，34~37 都是这一类型的错误；
- 权限不足错误：如 2 和 3 的错误都是开发者或应用用户的权限不足，造成服务无法访问的错误；
- 其它错误：以上类型之外的错误。

5.3 系统级子错误编码

子错误的编码是一个格式化的层级编码串，如 `isp.xxx-service-unavailable`、`isv.xxx-not-exist:invalid-yyy` 等，其中 `xxx` 和 `yyy` 都是变量占位符，会根据具体的错误填写相应的值。

每个子错误都对应一个主错误，而一个主错误可以对应多个子错误。子错误和主错误的映射关系在 Rop 的 `com.rop.validation.SubErrors` 中定义，`SubErrors` 类拥有一个获取子错误对应主错误的静态方法：

```
public static MainError getMainError(SubErrorType subErrorType, Locale locale)
```

`MainErrorType` 和 `SubErrorType` 的枚举类分别定义了系统级主错误和子错误的编码，主子错误的国际化信息都在 `i18n/rop/error` 的国际化资源文件中定义。

ISP 子错误编码

Rop 目前仅拥有两个 ISP 的子错误，它们对应的主错误编码为 1，即 `Service Currently Unavailable` 主错误。这两个 ISP 子错误分别是：

- `isp.xxx-service-unavailable`=调用后端服务{0}抛异常:{1}，服务不可用。\\n 异常信息:\\n{2}
- `isp.xxx-service-timeout` = 调用{0}服务超时，该服务的超时限制为{1}秒，请和服务平台提供商联系

因服务平台原因产生的任何错误，如数据库连接不上，某个服务资源不可用，内部抛出异常等，都对应 `isp.xxx-service-unavailable` 的 ISP 子错误。其中子错误代码中的 `xxx` 会被替换成具体的服务名。如 `user.get` 服务对应的子错误码为：`isp.user-get-service-unavailable`。

`isp.xxx-service-unavailable` 的错误消息是带参数的格式化串，Rop 会将引起错误的异常信息格式化到消息内容中。假设我们调用 `user.getSession` 服务方法，其内部抛出了 `IllegalArgumentException`，则客户端将得到一个如下的错误报文：

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<error code="1">
  <message>Service Currently Unavailable</message>
  <solution>Service Currently Unavailable</solution>
  <subErrors>
    <subError code="isp.user-getSession-service-unavailable">
      <message>
        call the back-end service user.getSession thrown
        exception:java.lang.IllegalArgumentException, the service is unavailable.
        exception info is :
        ...
      </message>
    </subError>
  </subErrors>
</error>
```

详细的错误信息不但方便客户端应用的开发者进行问题的定位，还可以充分利用这些

错误信息进行用户交互设计和业务逻辑的处理，有效提高服务开放平台的可编程能力。

Rop 允许设置服务执行的最大过期时间，即一个服务的执行超过指定限时后，直接返回错误报文，释放服务端资源，以平衡服务平台资源的利用。Rop 默认不设置服务过期时间，可通过<rop:annotation-driven/>的 service-timeout-seconds 属性指定服务执行最大过期时间，单位为秒：

```
<rop:annotation-driven service-timeout-seconds="10"/>
```

每个服务方法都可以定义自己的服务过期时间，以覆盖<rop:annotation-driven/>所定义的统一过期时间，来看一个例子：

代码清单 22 UserService.java:指定服务过期时间

```
@ServiceMethod(method = "user.timeout", version = "1.0", timeout = 1) ①
public Object timeoutService(CreateUserRequest request) throws Throwable {
    Thread.sleep(2000); ②
    CreateUserResponse response = new CreateUserResponse();
    response.setCreateTime("20120101010102");
    response.setUserId("2");
    return response;
}
```

在①处通过@ServiceMethod的 timeout 属性为 user.timeout 服务方法指定的过期时间为 1 秒。在②处，我们故意让该服务方法睡眠 2 秒钟，以便超过服务方法过期时间的限制。客户端调用该服务方法后，则返回如下的错误报文：

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<error code="1">
    <message>服务不可用</message>
    <solution>服务目前无法使用</solution>
    <subErrors>
        <subError code="isp.user-timeout-service-timeout">
            <message>
                调用user.timeout服务超时，该服务的超时限制为1秒，请和服务平台提供商联系。
            </message>
        </subError>
    </subErrors>
</error>
```

ISV 子错误编码

在实际应用中，开发者遇到更多的都会是 ISV 的错误。Rop 定义了 5 个 ISV 子错误，说明如下：

- isv.missing-parameter:xxx=缺少必要的参数{0}
- isv.parameters-mismatch:xxx-and-yyy=传入的参数{0}和{1}不匹配；
- isv.invalid-paramete:xxx=参数{0}无效，格式不对、非法值、越界等
- isv.invalid-permission=权限不够、非法访问
- isv.xxx-not-exist:invalid-yyy=根据{0}查询不到{1}

其中，前 3 个子错误都是由于业务级参数未能通过合法性校验而引起的。当请求参数对象的属性打上@NotNull 的 JSR 303 注解时，即说明该请求参数是必须的，如果未提供该参数就会报出 isv.missing-parameter:xxx 子错误，其中 xxx 为具体的参数名。

Rop 将请求参数绑定到请求参数对象属性时，如果发生类型不匹配的错误（如将 aaa 赋给一个整型的属性），将返回 isv.parameters-mismatch:xxx-and-yyy 的子错误码，其中 xxx 为参数名，而 yyy 为请求参数的值。违反其它校验规则的统一返回 isv.invalid-paramete:xxx 的子错误，其中 xxx 为未通过校验的参数名。

可以通过 SecurityManager 实现类开发服务权限管理逻辑，如果 SecurityManager 驳回服务的访问，则说明应用或用户的权限不足，不能访问某个受限平台服务，Rop 将返回 isv.invalid-permission 的子错误，

isv.xxx-not-exist:invalid-yyy 子错误表示不存在对应某个 ID 的对象，如根据 userId 获取用户对象时，如果查不到对应的对象，则返回该子错误。以上我们介绍的 4 个 ISV 子错误，ROP 都会自动产生，无需服务开发者关注。而 isv.xxx-not-exist:invalid-yyy 则需要服务开发者自己负责创建，可以通过 Rop 的 NotExistErrorResponse 对象产生该错误。来看一个 rop-sample 中的例子：

代码清单 23 UserService.java:模拟返回 CreateUserResponse 响应

```
@ServiceMethod(method = "user.get", version = "1.0", httpAction = HttpAction.GET)
public Object getUser(RopRequest request) throws Throwable {
    String userId = request.getRopRequestContext().getParamValue("userId");
    if("9999".equals(userId)){
        return new NotExistErrorResponse("user","userId","9999",
            request.getRopRequestContext().getLocale());
    }else{
        CreateUserResponse response = new CreateUserResponse();
        //add creaet new user here...
        response.setCreateTime("20120101010102");
        response.setUserId(userId);
        response.setFeedback("user.get");
        return response;
    }
}
```

① ← 模拟 userId 为 9999 不存在对应 User 对象的逻辑。

NotExistErrorResponse 构造函数的第 1 个入参为对象名，第 2 个入参为查询使用的属性名，第 3 个参数为查询的属性值。使用 userId 值为 9999 的请求参数调用该服务时，将返回如下的响应报文：

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<error code="33">
    <message>Invalid Arguments</message>
    <solution>
        check the Required application parameter is valid(refer the subError message)
    </solution>
    <subErrors>
        <subError code="isv.user-not-exist:invalid-userId">
            <message>cant find userId by 9999</message>
        </subError>
    </subErrors>
</error>
```

```

    </subError>
  </subErrors>
</error>

```

Rop 所定义的子错误码其实是一个模式化串, Rop 会根据具体的错误产生相应的编码。这些模式化的子错误码基本上涵盖了各种通用的错误, 从模型设计角度上看, 这个错误模型是收敛的。因此, 使用 Rop 的平台开发者无需再为设计错误码而烦心了, 直接使用这个错误模型即可, 大大降低了服务平台设计的难度。

5.4 业务级子错误编码

系统级子错误模型是业务无关的通用性错误, 在使用 Rop 开发您的服务平台时, 一定会有很多业务相关的错误。如删除一个不允许删除的业务单据, 在单据未通过审核时直接审批等等, 这些业务相关的错误, 称之为业务级子错误。

Rop 设计了一个通用的业务级子错误编码: `isv.xxx-service-error:yyy`, 其中 `xxx` 为服务方法名(服务方法名的“.”替换成“-”), 而 `yyy` 为业务错误代码, 一般由大写英文字符组成, 来看几个具体的业务级子错误编码的示例:

- `isv.order-delete-service-error:ORDER_NOT_ALLOW_DELETE;`
- `isv.order-approve-service-error:ORDER_IS_NOT_AUDIT;`
- `isv.user-password-change-service-error:PASSWORD_TOO_SIMPLE;`

由于业务级子错误是由服务平台开发者自己定义的, 因此错误码及错误信息的国际化资源也必须由服务平台开发者提供。`rop-sample` 的业务级子错误国际化资源位于 `rop-sample` 项目的 `i18n/rop/sampleRopError` 资源文件中。我们来看一下对应 `zh_CN` 的资源文件:

```
isv.user-add-service-error\USER_NAME_RESERVED= {0} 是预留的用户,请选择其它的用户名.
```

`sampleRopError_zh_CN.properties` 目前仅定义了一个业务级子错误, 可以在该文件中定义任意多个错误。属性值可以使用 `{n}` 定义变量占位符, 以便在运行其替换成具体的内容。



提 示

在属性资源文件中, 属性键名如果包含“:”或“=”的字符, 必须使用转义符, 即在其前面添加“\”字符。如果属性值一行编写不下, 可以通过“\”进行分行。具体参见 `java.util.Properties` 类的 `JavaDoc` 的说明。

必须在 `<rop:annotation-driven>` 中指定业务级子错误资源文件的地址, 以便 Rop 将其加载到框架中:

```
<rop:annotation-driven ext-error-base-name="i18n/rop/sampleRopError"/>
```

所有业务级子错误对应的主错误码均为 9 (业务逻辑出错), Rop 提供了一个用于构造业务级错误的响应类, 即 `BusinessServiceErrorResponse`, 其构造方法如下所示:

```
public BusinessServiceErrorResponse(String serviceName, String errorCode, Locale locale,
Object... params)
```

`serviceName` 为服务方法, 即 `method` 系统参数对应的值, 如“`user.get`”。`errorCode` 为

业务错误代码，如 ORDER_NOT_ALLOW_DELETE、ORDER_IS_NOT_AUDIT 等，它将用于替换 isv.xxx-service-error:yyy 中的“yyy”。最后的 params 用于替换错误消息内容的{n}中。

我们通过 rop-sample 的一个例子了解业务级子错误的具体使用方法：

代码清单 24 UserService.java:模拟返回 CreateUserResponse 响应

```
@ServiceMethod(method = "user.add", version = "1.0")
public Object addUser(CreateUserRequest request) {
    if (reservesUserNames.contains(request.getUserName())) {
        return new BusinessServiceErrorResponse(
            request.getRopRequestContext().getMethod(), "USER_NAME_RESERVED",
            request.getRopRequestContext().getLocale(), request.getUserName());
    } else {
        CreateUserResponse response = new CreateUserResponse();
        response.setCreateTime("20120101010101");
        response.setUserId("1");
        return response;
    }
}
```

①

如果新增的用户名是预留的用户名，则不能注册，返回一个业务级错误响应

user.add 是一个用于注册新用户的服 务，它需要保证新增的用户名不能使用预留的用户名，否则返回一个 isv.user-add-service-error:USER_NAME_RESERVED 的业务级错误。

假设“jhon”是预留的用户名，如果调用 user.add 服 务时，上传的 userName 参数是“jhon”，将返回如下的错误响应报文：

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<error code="9">
    <message>业务逻辑出错</message>
    <solution>请了解服务调用的前置条件，检查是否满足业务逻辑</solution>
    <subErrors>
        <subError code="isv.user-add-service-error:USER_NAME_RESERVED">
            <message>jhon 是预留的用户,请选择其它的用户名.</message>
        </subError>
    </subErrors>
</error>
```

6 响应报文控制

6.1 分体式报文模型

服务开放平台对服务的处理结果统一由响应报文告之调用终端，不管服务处理正确与否，都必须对应一个响应报文。如果响应报文既要包含正确的业务响应报文，又要包括错误的响应报文，将增加服务响应报文格式的复杂度。由于 Rop 已经建立了一个可以描述所有类型错误的错误模型，因此错误响应报文的格式是统一的。业务服务的结果响应报文是业务相关的，也就是说，每个业务服务方法的正确业务响应报文是不一样的。

基于以上的分析，Rop 将正确业务响应报文和错误处理响应报文独立开来，这样服务

平台的开发者仅需设计正确的业务响应报文即可，错误响应报文直接由 Rop 提供。如果服务正确执行，返回正确的业务响应报文，否则返回 Rop 格式的错误响应报文。

6.2 响应报文定义

Rop 的服务方法可返回任意类型的响应对象，响应类通过 JSR 222 注解进行对象流化定义。只要定义好服务的响应类，并正确标注 JSR 222 注解，业务响应报文也就定义好了。

代码清单 25 CreateUserResponse .java:服务响应的返回对象

```
package com.rop.sample.response;

import javax.xml.bind.annotation.*;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(name = "createUserResponse")
public class CreateUserResponse{

    @XmlAttribute      ② 将属性名输出流化为报文的元素
    private String userId;

    @XmlAttribute
    private String createTime;

    @XmlElement
    private Foo foo = new Foo();

    @XmlElement
    private String feedback;

    ...
}
```

1. 流化规则定义在属性成员中;
2. 根元素名为 createUserResponse

①

在上节的 addUser(CreateUserRequest request)服务方法中，如果一切服务执行正常，将返回一个 CreateUserResponse 的对象。

运行 rop-sample 项目的 UserServiceRawClient#testAddUserByVersion1()测试方法，将看到如下的响应报文：

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<createUserResponse createTime="20120101010101" userId="1">
  <foo field2="2" field1="1"/>
  <feedback>hello</feedback>
</createUserResponse>
```

观察以上的输出报文，可以看到 createTime 和 userId 以元素属性的形式出现，而 foo 和 feedback 则以子元素的形式出现。

6.3 报文输出格式

默认情况下，Rop 的响应报文是以 XML 的格式输出的。除 XML 之外，Rop 还支持 JSON 格式的响应报文。您只要将 format 的系统级参数设置为 json，就会返回 JSON 格式的响应报文了。

运行 rop-sample 项目的 UserServiceRawClient#testAddUserWithJsonFormat()测试方法，您将看到如下的输出：

```
{ "userId": "1", "createTime": "20120101010101", "foo": { "field1": "1", "field2": "2" }, "feedback": "hello" }
```

值得注意的是，当前 Rop 不支持混合使用 XML 和 JSON 的报文格式，也即请求和响应的数据格式要么采用 XML，要么采用 JSON。如果希望采用 XML 格式的请求参数值，那么响应的报文也必须是 XML。

6.4 报文的国际化支持

Rop 使用 locale 系统级参数指定本地化的信息，默认的 locale 值为 zh_CN，服务平台将根据 locale 值的不同返回相应的本地化报文信息。由于错误报文是由 Rop 框架全权负责的，因此错误报文的国际化问题，Rop 本身已经提供了解决方案。目前，Rop 为系统级错误提供了 zh_CN 和 en 两个国际化资源文件。

运行 rop-sample 的 UserServiceRawClient#testI18nErrorMessage()测试方法，该方法将发起两次对 user.add 服务方法的调用，其中请求参数都相同，唯有 locale 的值不同：第一次为 locale=en，第二次为 locale=zh_CN。第一次服务调用返回的错误响应报文为：

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<error code="33">
  <message>Invalid Arguments</message>
  <solution>
    check the Required application parameter is valid(refer the subError message)
  </solution>
  <subErrors>
    <subError code="isv.parameters-mismatch:salary-and-yyy">
      <message>
        incoming parameter salary and aaa does not match, both have a certain correspondence between
      </message>
    </subError>
  </subErrors>
</error>
```

第二次服务调用返回的错误响应报文：

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<error code="33">
  <message>非法的参数</message>
  <solution>请查看根据服务接口对参数格式的要求</solution>
  <subErrors>
    <subError code="isv.parameters-mismatch:salary-and-yyy">
```

```
<message>传入的参数salary和aaa不匹配，两者有一定的对应关系</message>
</subError>
</subErrors>
</error>
```

正确的业务响应报文由服务平台开发者负责，因此国际化的工作也由服务平台开发者负责。在服务方法内部，可以通过 `request.getRopRequestContext().getLocale()` 获取请求参数 `locale` 对应的本地化对象，进而控制响应对象的国际化输出。如果服务开放平台没有国际化的需求，可以不关注 `locale` 参数。

7 文件上传

7.1 Rop 文件上传解决思路

由于服务请求报文是一个文本，无法直接传送二进制的文件内容，因此必须采用某种转换机制将二进制的文件内容转换为字符串。**Rop** 采用如下的方式对上传文件进行编码：

```
<fileType>@<BASE64(文件内容)>
```

`<fileType>`代表文件类型，文件内容采用 **BASE64** 算法进行编码，这样二进制的文件内容就可以转换为一个字符串，两者“@”字符分隔。服务端接收到上传的文件后，即可解析出文件的类型和文件的内容。

Rop 定义了一个 `UploadFile`，代表一个上传的文件，来看一下 `UploadFile` 的定义：

代码清单 26 UploadFile.java

```
package com.rop.request;

import com.rop.annotation.IgnoreSign;
import org.springframework.util.FileCopyUtils;

import java.io.File;
import java.io.IOException;

@IgnoreSign ①
public class UploadFile {

    private String fileType;

    private byte[] content;

    public UploadFile(String fileType, byte[] content) {
        this.content = content;
        this.fileType = fileType;
    }

    public UploadFile(File file) {
        try {
```

说明签名算法将忽略所有 `UploadFile` 的参数数据的签名。

```

        this.content = FileCopyUtils.copyToByteArray(file);
        this.fileType = file.getName().substring(file.getName().lastIndexOf('.')+1);
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}

public String getFileType() {
    return fileType;
}

public byte[] getContent() {
    return content;
}
}

```

UploadFile 类定义处标注了@IgnoreSign 注解，这说明 Rop 的签名算法会忽略所有 UploadFile 属性。因此，客户端无须将上传文件的参数纳入到签名参数列表中。

Rop 定义了一个服务于 UploadFile 中的转换器：UploadFileConverter，Rop 已经在内部注册了这个转换器，因此可以在 RopRequest 实现类中直接使用 UploadFile，Rop 会自动将代表上传文件的格式化字符串参数正确绑定到 UploadFile 中。

7.2 文件上传实例

rop-sample 的 UserService 有一个用于上传用户头像的服务方法：

代码清单 27 UploadFile.java

```

@ServiceMethod(method = "user.upload.photo", version = "1.0", httpAction = HttpAction.POST)
public Object uploadPhoto(UploadUserPhotoRequest request) throws Throwable {

```

```

    String fileType = request.getPhoto().getFileType();
    int length = request.getPhoto().getContent().length;

```

① 获取上传文件的类型和文件内容

```

    ClassPathResource outFile = new ClassPathResource("/");
    FileCopyUtils.copy(request.getPhoto().getContent(),
        new File(outFile.getFile().getParent()+"/1." + fileType));

```

② 将上传的文件保存到类路径下

```

    UploadUserPhotoResponse response = new UploadUserPhotoResponse();
    response.setFileType(fileType);
    response.setLength(length);
    return response;
}

```

在①处获取文件的类型和文件的内容，在②处将其保存到服务器的类路径下，最后返回上传成功的响应。uploadPhoto()服务方法的入参是 UploadUserPhotoRequest，其代码如下所示：


```
public class UploadUserPhotoRequest extends AbstractRopRequest {
    private String userId;

    private UploadFile photo; ①
    ...
}
```

photo 属性对应的类型即是 UploadFile，Rop 会自动将格式化的 photo 字符串请求参数绑定到这个属性对象中。下面是客户端调用 user.upload.photo#1.0 服务的测试方法，它位于 UserServiceRawClient 类中：

代码清单 28 UserServiceRawClient.java：测试调用上传文件服务

```
@Test
public void testUploadUserPhoto() throws Throwable {
    RestTemplate restTemplate = new RestTemplate();
    MultiValueMap<String, String> form = new LinkedMultiValueMap<String, String>();
    form.add("method", "user.upload.photo");
    form.add("appKey", "00001");
    form.add("v", "1.0");
    form.add("sessionId", "mockSessionId1");
    form.add("locale", "en");
    form.add("userId", "1");

    String sign = RopUtils.sign(form.toSingleValueMap(), "abcdeabcdeabcdeabcdeabcde");
    form.add("sign", sign);

    ClassPathResource resource = new ClassPathResource("photo.png");
    UploadFile uploadFile = new UploadFile(resource.getFile());//①构造一个上传文件对象

    //②添加一个上传的文件，photo参数不参与签名
    form.add("photo", "png@" + Base64.encodeBase64String(uploadFile.getContent()));

    //③调用上传文件服务
    String response = restTemplate.postForObject(SERVER_URL, form, String.class);
    System.out.println("response:\n" + response);
    assertTrue(response.indexOf("png") > -1);
}
```

在①处选择一个要上传的文件，在②处将其转换为一个格式化字符串并使用 photo 传递，在③处直接调用 user.upload.photo#1.0 服务上传文件。

7.3 文件上传控制

一般情况下，服务开放平台对上传文件的类型及大小都有严格的限制，一方面保障了服务平台的安全，另一方面也可以限制了服务平台资源的占用。

默认情况下，Rop 不限制上传文件的类型且允许最大上传文件的大小为 10M。可以通过<rop:annotation-driven/>进行相应的限制：

```
<rop:annotation-driven
```

```
upload-file-max-size="10"  
upload-file-types="png,gif"/>
```

upload-file-max-size 的单位为 K，用以指定最大上传文件的大小，而 upload-file-types 用于指定允许上传的文件类型，多值用逗号分隔。如果允许上传所有文件，可以设置为“*”。按照以上的配置，允许上传的文件类型为 png 及 gif，而最大允许上传文件的大小为 10K。

8 服务安全控制

8.1 安全控制架构

服务开放平台由于服务接口是公开发布的，存在很大的安全隐患。如何保障服务访问的安全，保证客户端按规范对服务进行访问，阻止恶意用户的访问和攻击是服务开放平台一项重要的工作。可以说，如果服务安全问题解决不好，将直接导致服务开放平台的失败。

SecurityManager 可以看成是 Rop 的安全部门总管，SecurityManager 通过协调多个安全组件完成 Rop 服务访问安全的整体控制，Rop 安全部门的成员及协作关系如图 7 所示：

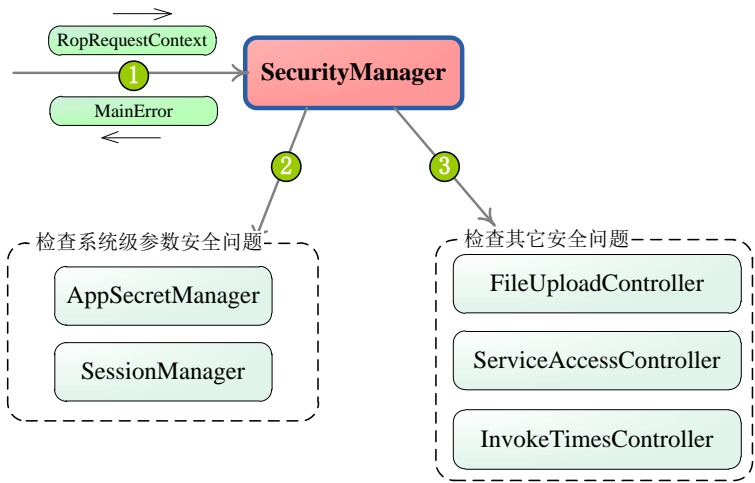


图 7 请求数据转换过程

前面我们介绍 Rop 技术架构时，SecurityManager 已经出场了。当服务请求到达后，ServiceRouter 将事先“咨询” SecurityManager：这个服务请求有没有安全问题？如果 SecurityManager 回告有安全问题（即返回的 MainError 不为空），则 ServiceRouter 驳回服务请求，返回相应的错误响应。只有 SecurityManager 回告该服务请求没有安全问题（即返回的 MainError 为 null），ServiceRouter 才调用 ServiceMethodAdapter 执行目标服务并返回正确的响应。

下面，我们来认识一下 Rop 的“安全部门”都有哪些人物：

- com.rop.security.AppSecretManager：应用键/密钥管理器，它负责检查 appKey/secret 的安全性。同时，SecurityManager 将在 AppSecretManager 的配合下完成请求数据签名的校验；
- com.rop.session.SessionManager：会话管理器，SecurityManager 利用

SessionManager 访问 Session 对象，同时完成会话安全的检验；

- **com.rop.security.FileUploadController**: 文件上传控制器，判断上传文件的类型和大小是否符合要求，上节介绍的文件上传控制其内部就是由 FileUploadController 负责的；
- **com.rop.security.ServiceAccessController**: 服务访问控制器，判断某个应用或用户是否有权访问目标服务；
- **com.rop.security.InvokeTimesController**: 判断应用/会话/用户访问服务的次数或频率是否超限。

前面 3 个安全组件，我们在前面已经学习过了，接下来，我们来了解一下后两个安全组件的具体使用。

8.2 ServiceAccessController

SecurityManager 委托 ServiceAccessController 判断应用或应用的用户有否权限访问某个服务。服务平台开发者，可以实现 ServiceAccessController 接口，定义自己的服务访问安全控制逻辑。ServiceAccessController 拥有两个接口方法：

- **boolean isAppGranted(String appKey,String method,String version)**: 是否授权应用访问某个服务；
- **boolean isUserGranted(Session session,String method,String version)**: 是否授权用户访问某个服务。

默认情况下，Rop 使用 DefaultServiceAccessController 作为开放平台的服务访问权限器，在该默认实现类中，这两个方法也都直接返回 true，表示不进行服务访问控制，即所有服务方法对任何应用及应用的用户都完全开放。

下面，我们通过 rop-sample 的 SampleServiceAccessController 了解如何开发一个服务访问控制器：

代码清单 29 SampleServiceAccessController.java:服务访问控制器

```
package com.rop.sample;

import com.rop.security. ServiceAccessController;
...

public class SampleServiceAccessController implements ServiceAccessController{

    private static final Map<String, List<String>> aclMap = new HashMap<String, List<String>>();

    static {

        ArrayList<String> serviceMethods = new ArrayList<String>();
        serviceMethods.add("user.logon");
        serviceMethods.add("user.logout");
        serviceMethods.add("user.getSession");
        aclMap.put("00003", serviceMethods);
    }
}
```

①
定义一个权限控制列表，
使 appKey 为 00003 的应用
仅能访问 3 个服务方法。

```

@Override
public boolean isAppGranted(String appKey,String method,String version) {②
    if(aclMap.containsKey(appKey)){
        List<String> serviceMethods = aclMap.get(appKey);
        return serviceMethods.contains(method);
    }else{
        return true;
    }
}

@Override
public boolean isUserGranted(Session session, String method, String version) {
    return true;
}
}

```

根据应用的服务权限控制列表进行检查。

在①中，我们通过一个 Map 模拟应用的服务访问控制列表。在实际应用中，您应当将权限控制列表保存到 LDAP 或数据库中。在②处，覆盖 isAppGranted()接口方法，根据服务访问控制列表对应用进行安全访问检查。

下面的配置片断将 SampleServiceAccessController 装配到 Rop 中：

```

<rop:annotation-driven service-access-controller="serviceAccessController"/>
<bean id="serviceAccessController" class="com.rop.sample.SampleServiceAccessController"/>

```

当未通过 SampleServiceAccessController 的安全访问的验证时，系统将返回相应的错误报文。运行 UserServiceRawClient#testViolationServiceAccessController()测试方法，将看到如下的响应报文：

```

<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<error code="2">
    <message>开发者权限不足</message>
    <solution>不足的ISV权限</solution>
    <subErrors>
        <subError code="isv.invalid-permission">
            <message>权限不够、非法访问</message>
        </subError>
    </subErrors>
</error>

```

8.3 InvokeTimesController

任何服务开放平台的计算资源都有限的，必须考虑平台服务的可用性。如果某个应用占用过多的服务资源，势必影响到其它用户的访问，从而响应服务平台整体的稳定性。因此，一般需要对接入应用进行服务资源的配额管控，当超过服务配额时，暂时或阻止服务的请求。

Rop 可以分别在应用级、会话级及用户级进行服务配额的限制，当超过服务调用次数或频率时，拒绝提供服务直接返回错误的报文。Rop 定义了一个 InvokeTimesController 接

口，开发者只要实现该接口并注册到 Rop 中就可以了。来了解一下 InvokeTimesController 接口的方法：

- `void caculateInvokeTimes(String appKey,Session session)`：计算服务调度的次数。每次服务调用后，Rop 都会调用该接口方法。开发者可以实现该方法，以记录应用、会话及用户的服务调用次数及频率的情况；
- `boolean isUserInvokeLimitExceed(String appKey,Session session)`：判断某个应用用户的服务调用次数是否已经超限。平台服务开发者需要自行根据 `caculateInvokeTimes()` 统计的结果进行判断，下同；
- `boolean isSessionInvokeLimitExceed(String appKey,String sessionId)`：判断某个应用的某个会话的服务调用次数是否已经超限；
- `boolean isAppInvokeLimitExceed(String appKey)`：判断某个应用的服务调用次数是否已经超限；
- `boolean isAppInvokeFrequencyExceed(String appKey)`：判断某个应用的服务调用频率是否已经超限。

rop-sample 也提供了一个 `SampleInvokeTimesController` 的实现类，介绍如下：

代码清单 30 SampleInvokeTimesController.java

```
package com.rop.sample;

import com.rop.security.InvokeTimesController;
import com.rop.session.Session;

import java.util.*;

public class SampleInvokeTimesController implements InvokeTimesController{

    private static Map<String,Integer> appCallLimits = new HashMap<String,Integer>();
    private static Map<String,Integer> appCallCounter = new HashMap<String,Integer>();
    static {
        appCallLimits.put("00002",10); ① ← appKey 为 00002 的应用访问服务的最大
    }                                     次数设置为 10

    @Override
    public void caculateInvokeTimes(String appKey, Session session) { ② ← 统计应用调用服务的次数
        if(!appCallCounter.containsKey(appKey)){
            appCallCounter.put(appKey,0);
        }
        appCallCounter.put(appKey,appCallCounter.get(appKey)+1);
    }

    @Override
    public boolean isUserInvokeLimitExceed(String appKey, Session session) {
        return false;
    }

    @Override
```

```

public boolean isSessionInvokeLimitExceed(String appKey, String sessionId) {
    return false;
}

@Override
public boolean isAppInvokeLimitExceed(String appKey) { ③
    return appCallLimits.containsKey(appKey) &&
        appCallCounter.get(appKey) > appCallLimits.get(appKey);
}

@Override
public boolean isAppInvokeFrequencyExceed(String appKey) {
    return false;
}
}

```

如果超过服务最多调用次数，返回 true 阻止请求。

开发好 SampleInvokeTimesController 后，通过如下配置将其安装到 Rop 中：

```

<rop:annotation-driven invoke-times-controller="invokeTimesController"/>
<bean id="invokeTimesController"
    class="com.rop.sample.SampleInvokeTimesController"/>

```

UserServiceRawClient#testViolationInvokeTimesController()演示了违反服务调用次数限制的情况，服务端将返回如下的错误报文：

```

<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<error code="36">
    <message>应用调用服务的次数超限</message>
    <solution>请检查应用授权访问服务的次数</solution>
</error>

```

9 拦截器及事件体系

9.1 拦截器

服务请求在通过 SecurityManager 的安全检查后，Rop 将依次执行 com.rop.Interceptor 拦截器的 beforeService()方法，一旦某个拦截器通过 RopRequestContext#setRopResponse()设置了一个响应对象，Rop 将终止执行链直接返回响应。在目标服务方法执行成功后，Rop 再依次执行这些拦截器的 beforeResponse()方法。其调用流程在图 8 中说明：

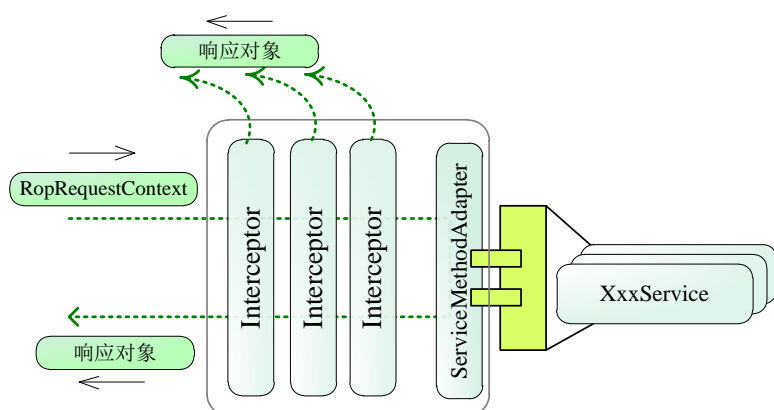


图 8 Rop 拦截器体系

通过上图,我们可以发现 Rop 的拦截器架构是完全参考了 Spring MVC 的拦截器体系设计的。我们来看一下 com.rop.Interceptor 有哪些接口方法:

- `void beforeService(RopRequestContext ropRequestContext)`: 在调用目标服务方法之前调用, 如果在该方法中, 通过 `RopRequestContext#setRopResponse()` 设置了响应对象, Rop 将中断执行链的执行直接返回响应;
- `void beforeResponse(RopRequestContext ropRequestContext)`: 在调用完目标服务方法后, 执行该拦截器方法;
- `boolean isMatch(RopRequestContext ropRequestContext)`: 可以通过该方法指定拦截器的匹配服务方法, 只有该方法返回 `true` 时, 拦截器才会实施拦截;
- `int getOrder()`: 指定拦截器的先后顺序。

rop-sample 中定义了一个 `ReservedUserNameInterceptor` 拦截器, 来看一下它的代码:

代码清单 31 ReservedUserNameInterceptor.java

```
package com.rop.sample;
```

```
import com.rop.AbstractInterceptor;
import com.rop.RopRequestContext;
import com.rop.sample.response.InterceptorResponse;
import org.springframework.stereotype.Component;
```

```
@Component
```

```
public class ReservedUserNameInterceptor extends AbstractInterceptor {
```

```
    @Override
```

```
    public void beforeService(RopRequestContext ropRequestContext) { ①
```

```
        if ("jhonson".equals(ropRequestContext.getParamValue("userName"))) {
```

```
            InterceptorResponse response = new InterceptorResponse();
```

```
            response.setTestField("the userName can't be jhonson!");
```

```
            ropRequestContext.setRopResponse(response);
```

```
        }
```

```
    }
```

当新增用户的服务使用了预留的用户名时, 阻止请求继续执行。

```

@Override
public void beforeResponse(RopRequestContext ropRequestContext) {
}

@Override
public boolean isMatch(RopRequestContext ropRequestContext) {②
    return "user.add".equals(ropRequestContext.getMethod());
}
}

```

该拦截器仅对 user.add 实施拦截。

ReservedUserNameInterceptor 的作用是确保调用“user.add”的服务时，新增用户的 userName 不使用到预留的用户名。该拦截器通过 isMatch() 限定拦截的目标，这里，仅对 user.add 的服务实施拦截。

在开发出拦截器后，还必须将拦截器注册到 Rop 中，这样拦截器才能工作。Rop 为注册拦截器提供了专门的 Schema：

```

<rop:interceptors>
    <bean class="com.rop.sample.ReservedUserNameInterceptor"/>
</rop:interceptors>

```

在<rop:interceptors/>中可以注册任意多个拦截器。

9.2 事件及监听

Rop 框架拥有一个完成的事件体系：包括框架级事件和服务级事件。框架级事件包括框架启动后事件（AfterStartedRopEvent）及框架关闭前事件（PreCloseRopEvent）。而服务级事件包括服务执行前事件（PreDoServiceEvent）和服务执行后事件（AfterDoServiceEvent）。

一般情况下，每个事件都要对应一个自己的监听器，但是 Spring 的事件体系颠覆了这种笨拙的设计模式，它只定义了一个监听器接口，通过泛型指定监听不同的事件。Rop 传承了 Spring 这一优秀的监听器设计理念，定义了一个 RopEventListener：

```

public interface RopEventListener<E extends RopEvent> extends EventListener {

    void onRopEvent(E ropEvent);

    int getOrder();
}

```

实现类仅需要通过泛型 E 指定需要监听的事件类型就可以了，getOrder() 接口方法指定监听器的执行顺序，序号越小越先执行。下面是 rop-sample 中的一个具体例子：

代码清单 32 SamplePreDoServiceEventListener.java

```

package com.rop.sample;

import com.rop.RopRequest;
import com.rop.RopRequestContext;
import com.rop.event.PreDoServiceEvent;

```



```

import com.rop.event.RopEventListener;
import com.rop.marshaller.MessageMarshallerUtils;

public class SamplePreDoServiceEventListener
    implements RopEventListener<PreDoServiceEvent> {①

    @Override
    public void onRopEvent(PreDoServiceEvent ropEvent) {
        RopRequestContext ropRequestContext = ropEvent.getRopRequestContext();
        if(ropRequestContext != null && ropRequestContext.getRopRequest() != null){
            RopRequest ropRequest = ropRequestContext.getRopRequest();
            String message = MessageMarshallerUtils.getMessage(ropRequest,
                ropRequestContext.getMessageFormat());
            System.out.println("message("+ropEvent.getServiceBeginTime()+")+"+message);
        }
    }

    @Override
    public int getOrder() {
        return 1;
    }
}

```

该监听器将对 *PreDoServiceEvent* 事件进行监听

和拦截器一样，事件监听器也必须注册到 Rop 中才会生效，其配置片断如下所示：

```

<rop:listeners>
    <bean class="com.rop.sample.SamplePostInitializeEventListener"/>
    <bean class="com.rop.sample.SamplePreDoServiceEventListener"/>
    <bean class="com.rop.sample.SampleAfterDoServiceEventListener"/>
</rop:listeners>

```

以上配置片断一共注册了 3 个事件监听器，它们分别监听不同的事件。如果多个监听器监听同一个事件，监听器的执行顺序由监听器的 `getOrder()` 决定。

Rop 采用异步的方式执行事件监听，服务执行的线程和事件监听执行线程是两个不同的线程，也就是说，事件监听器的执行不会影响服务方法的执行。所以您可以开发一个监听 *AfterDoServiceEvent* 事件的监听器，记录每个请求/响应报文的日志，由于事件监听器异步运行，它不会影响正常服务执行的效率。

值得注意的是，Rop 的拦截器是工作于服务执行线程中的，拦截器的执行效率会直接影响到服务执行的效率，因此，不宜在拦截器中做太多的事情。



实战经验

一般情况下，服务开放平台都必须记录每个服务请求的请求报文和响应报文，以便后续的审计和分析工作。如果直接在服务前后记录报文，将会影响服务执行的效率。通过编写一个 `AfterDoServiceEvent` 事件的监听器，在监听方法中记录请求/响应报文是一个非常理想的实现方案。类似的，您还可以将一些不太重要的辅助性工作安排到事件监听器中去做。

10 性能调优

服务开放平台负载能力是评价服务平台优劣的一个重要指标，服务平台的负载能力可以通过硬件架构（如采用服务器群集）及软件架构（如充分利用缓存）来提升。Rop 作为一个独立的服务开放平台框架，不限制平台开发者采用各种软件和硬件的优化方案。

对于服务平台的单个服务结点的来说，其内存资源和 CPU 能力都是有限的，如何在保证系统稳定的前提下提升吞吐率，实现单点的优化控制，这是 Rop 需要考虑的。

10.1 服务平台线程池参数调整

Rop 内部拥有一个线程池，它由 `ThreadPoolExecutor` 实现，统一使用 `ThreadPoolExecutor` 执行服务请求。可以通过合理规划 `ThreadPoolExecutor` 的参数优化 Rop 的性能。

`<rop:annotation-driven>` 提供了 4 个用于调节 `ThreadPoolExecutor` 参数的属性：

- `core-pool-size`：池中核心线程数，默认为 200；
- `max-pool-size`：池中最大线程数，默认为 500；
- `keep-alive-seconds`：如果池中当前有多于 `corePoolSize` 的线程，则这些多出的线程在空闲时间超过 `keepAliveTime` 时将会终止；
- `queue-capacity`：列队的长度。

`ThreadPoolExecutor` 将根据 `corePoolSize` 和 `maximumPoolSize` 设置的边界自动调整池大小。当新任务提交时，如果运行的线程少于 `corePoolSize`，则创建新线程来处理请求，即使其他辅助线程是空闲的。如果运行的线程多于 `corePoolSize` 而少于 `maximumPoolSize`，则仅当队列满时才创建新线程。

下面通过 `<rop:annotation-driven>` 调整 Rop 线程池的参数：

```
<rop:annotation-driven
    core-pool-size="500"
    max-pool-size="1000"
    queue-capacity="20"
    keep-alive-seconds="300"/>
```

我们并不能给出一套放之四海而皆准的标准配置，平台开发者需要根据具体的环境，

通过压力测试等工具得到最佳的参数值。

10.2 限制服务的占用时长

服务开放平台拥有众多的服务并被所有平台应用共享，如果某些服务执行占用过多时间使资源得不到快速释放，将会影响到服务平台的整体性能。所以为了保障服务平台的整体性能，应该避免服务长时间不释放的问题。

Rop 允许通过<rop:annotation-driven/>的 `service-timeout-seconds` 指定所有服务统一的过期时间，单位都为秒。还可以在服务方法定义处通过 `@ServiceMethod` 的 `timeout` 指定具体方法的过期时间。这样，如果某次服务请求占用时间过长，Rop 会强制释放请求并返回相应的错误响应。

通过合理设计服务过期时间，可以有效保障服务开放平台整体的稳定性，避免因某个异常事件的积累效应而导致整体的崩溃，提升平台的健壮性。

10.3 限制应用/用户的访问

我们一再强调服务开放平台是公共资源，它的服务对象是一个群体而不是单一的应用，因此，必须防止某个应用或用户过多地占用服务资源。这好比医院是为广大患者提供服务的，如果某个患者一天就挂了 100 个专家号，那么其它的患者势必就看不上病了，所以医院需要限制每个患者的挂号次数，这样才能保证医疗资源的合理利用。

您可以设计一个 `InvokeTimesController` 实现类，合理限制应用、会话及用户的访问次数及频度。在保障平台应用正常运行的情况下，提升服务平台的稳定性。

11 开发客户端 SDK

服务开放平台的所有服务都必须发布清晰的 API 接口，服务开放平台的 API 是基于 HTTP 协议定义，和具体开发语言无关，应用开发者完全可以采用感兴趣语言访问服务。但是平台应用开发者如果直接根据 API 协议构造请求报文访问服务，其开发难度还是很大的，同时开发效率也必将受到很大的影响。

所以基本上所有的服务开放平台都会为不同语言提供相应客户端 SDK 开发包，如 Java 版、PHP 版的 SDK 包。这样应用开发者仅需引入平台的 SDK 包，就可以通过很自然的方式访问平台服务，而无需关心请求报文发送，响应报文解析等问题，大大提高应用的开发效率。

11.1 Rop 提供了哪些支持

Rop 为了方便服务开放平台开发者打造自己平台的 SDK 包，提供了一个专门的 `com.rop.client` 包，可以很方便地使用包中的类打造服务平台 Java 版的 SDK 开发包。首先，来了解一下 `com.rop.client.RopClient`，`RopClient` 的代表服务平台的客户端调用器，它包括了若干易用的方法：

- `ClientRequest buildClientRequest()`: 构造一个服务请求对象 `ClientRequest`, 每次请求对应一个 `ClientRequest`, 可通过 `ClientRequest` 发送服务请求获取服务响应。关于 `ClientRequest`, 我们将在后面进行详细说明;
- `void setSessionId(String sessionId)`: 设置会话的 ID, 当为 `RopClient` 设置 `sessionId` 后, 后续的服务请求都会自动带上这个 `sessionId` 的系统级参数;
- `void addRopConverter(RopConverter ropConverter)`: 将 `RopConverter` 注册到 `RopClient` 中。这样, `RopClient` 就会在发送服务请求时, 自动将相应对象型的请求参数转换成 `String` 型的请求参数;
- `RopClient setAppKeyParamName(String paramName)`: 指定 `appKey` 系统级参数所采用的参数名, `RopClient` 为所有系统参数都定义了更改参数名的方法, 如 `setMethodParamName()`、`setSessionIdParamName()` 等。客户端的系统级参数名必须和服务端的保持一致, 如果服务平台通过 `<rop:sysparams/>` 更改了某个系统级参数的参数名, 则客户端也必须调用相应的 `setXxxParamName()` 方法显式指定参数名。如果服务平台采用默认的系统级参数名, 则 `RopClient` 也无需调用 `setXxxParamName()` 指定参数名。

所有的服务请求都通过 `com.rop.client.ClientRequest` 发送, `ClientRequest` 采用“链式编程”模式定义了一系列接口方法:

- `ClientRequest addParam(String paramName, Object paramValue)`: 添加一个请求参数, 如果 `paramValue` 的类型拥有对应 `RopConverter`, 则 `RopClient` 会调用对应 `RopConverter` 的 `unconvert()` 方法将 `paramValue` 转换为对应字符串。值得注意的是, 仅需通过该方法添加业务级的参数即可, 系统级参数自动由 `RopClient` 组装;
- `ClientRequest addParam(String paramName, Object paramValue, boolean needSign)`: 默认所有的请求参数都在签名, 如果某个参数不需要签名, 可以将 `needSign` 设置为 `false`;
- `ClientRequest clearParam()`: 如果要复用一个 `ClientRequest` 对象, 在完成一次服务请求后, 必须调用该方法清除 `ClientRequest` 中的请求参数列表;
- `<T> CompositeResponse post(Class<T> ropResponseClass, String methodName, String version)`: 通过 POST 发送服务请求, 调用 `methodName#version` 的服务, 通过前面的 `addParam()` 方法添加的请求参数将组装成相应的请求报文发送到服务平台中。该方法指定了响应对象的类型为 `ropResponseClass`, 如果服务调用成功, 则 `CompositeResponse` 的 `isSuccessful()` 方法将返回 `true`, 其 `T getSuccessResponse()` 方法将返回对应的业务响应对象, 这个对象的类型即通过 `ropResponseClass` 参数指定。如果服务调用失败, 则 `isSuccessful()` 方法将返回 `false`, 可以通过 `CompositeResponse` 的 `getErrorResponse()` 方法获取错误响应对象;
- `<T> CompositeResponse post(RopRequest ropRequest, Class<T> ropResponseClass, String methodName, String version)`: 您也可以通过 `RopRequest` 统一指定请求参数, `RopClient` 自动将 `RopRequest` 中的属性抽取成一个个请求参数组装成完整的请求报文。使用该方法发送的请求不关注通过 `addParam()` 添加的业务级请求参数, 仅

是关注 RopRequest 中的业务参数。

相似的，ClientRequest 还提供了和 post()相似的两个 get()方法，不再赘述。综上所述，系统级参数统一由 RopClient 负责，ClientRequest 仅需指定业务级参数即可。ClientRequest 有两种方式发送服务请求，其一是通过 addParam()一个个添加业务级参数，其二是直接在调用 post()或 get()方法时，通过 RopRequest 传入；

对于业务级参数是否纳入签名的控制，如果采用 addParam()添加参数，如果参数类定义处没有标注 @IgnoreSign，则需显式指定，即使用 addParam("param1","value1",false)添加参数。如果采用带 RopRequest 入参的 post()/get()发送请求，RopClient 会自动根据属性中的 @IgnoreSign 判断是否在纳入签名，无需显式指定。

在 10.1.2 小节的代码清单 6 中，我们已经接触到了 RopClient，RopClient 比直接使用 Spring 的 RestTemplate 访问服务要简洁得很多。rop-sample 的 UserServiceClient 提供了若干个演示 RopClient 的测试方法，我们摘取一个来学习使用 RopClient 的整体过程：

代码清单 33 UserServiceClient.java：客户端直接使用 RopClient 访问服务

```
public class UserServiceClient {

    public static final String SERVER_URL = "http://localhost:8080/router";
    public static final String APP_KEY = "00001";
    public static final String APP_SECRET = "abcdeabcdeabcdeabcdeabcde";
    private DefaultRopClient ropClient =
        new DefaultRopClient(SERVER_URL, APP_KEY, APP_SECRET);

    {
        ropClient.setFormatParamName("messageFormat");
        ropClient.addRopConverotor(new TelephoneConverter());
    }

    @Test
    public void addUser() {
        LogonRequest ropRequest = new LogonRequest();
        ropRequest.setUserName("tomson");
        ropRequest.setPassword("123456");
        CompositeResponse response = ropClient.get(ropRequest, LogonResponse.class,
            "user.getSession", "1.0");
        String sessionId = ((LogonResponse) response.getSuccessResponse()).getSessionId();
        ropClient.setSessionId(sessionId); ③ 设置 sessionId，这样后续的所有服务请求都会带上这个 sessionId

        CreateUserRequest createUserRequest = new CreateUserRequest();
        createUserRequest.setUserName("katty");
        createUserRequest.setSalary(2500L);

        response = ropClient.buildClientRequest()
            .post(createUserRequest, CreateUserResponse.class,
                "user.add", "1.0"); ③ 通过 post()调用 user.add#1.0 服务
```

构造 RopClient ①

对 ropClient 进行相关配置 ②

```

        assertNotNull(response);
        assertTrue(response.isSuccessful());
        assertTrue(response.getSuccessResponse() instanceof CreateUserResponse);
    }
}

```

使用 RopClient 调用服务一般有 6 个步骤：

- 1) 构造 RopClient 实例，指定了服务地址并设置 appKey/secret；
- 2) 对 RopClient 进行个性化定制，如指定系统级参数名，注册 RopConverter 等；
- 3) 调用获取服务端会话的服务获取 sessionId 并将其设置到 RopClient 中，这样后续的服务调用都会带上这个 sessionId；
- 4) 构造服务请求对象，指定业务级参数；
- 5) 创建一个 ClientRequest 并执行具体的服务调用；
- 6) 获取服务响应对象，执行相关的处理逻辑。

前 3 个步骤仅需执行一次，以后的服务调用仅需执行后 3 个步骤就可以了。

11.2 服务开放平台相关的 SDK 包

首先，来了解一下服务开放平台的 SDK 包含哪些内容：

- 1) 整个 Rop 的 JAR 包；
- 2) 服务开放平台所有的 RopConverter；
- 3) 服务开放平台的所有请求及响应类。

具体的开放平台可以根据通过封装 RopClient，如您可以把指定服务端 URL、指定系统参数名及注册 RopConverter 等工作固化下来，这样，就可开发出更回便捷的客户端类。来看一下 rop-sample 的 RopSampleClient：

代码清单 34 RopSampleClient.java：打造服务平台自身的客户端

```

package com.rop.sample.client;

import com.rop.AbstractRopRequest;
import com.rop.RopRequest;
import com.rop.client.CompositeResponse;
import com.rop.client.DefaultRopClient;
import com.rop.sample.request.LogonRequest;
import com.rop.sample.request.TelephoneConverter;
import com.rop.sample.response.LogonResponse;

public class RopSampleClient {

    public static final String SERVER_URL = "http://localhost:8088/router";
    private DefaultRopClient ropClient ;

    public RopSampleClient(String appKey,String secret) { ①
        ropClient = new DefaultRopClient(SERVER_URL, appKey, secret);
        ropClient.setFormatParamName("messageFormat");
        ropClient.addRopConvector(new TelephoneConverter());
    }
}

```

通过 appKey/secret 即可构造出一个完善的客户端对象。

```

}

public String logon(String userName, String password) {②
    LogonRequest ropRequest = new LogonRequest();
    ropRequest.setUserName("tomson");
    ropRequest.setPassword("123456");
    CompositeResponse response = ropClient.buildClientRequest()
        .get(ropRequest, LogonResponse.class, "user.logon", "1.0");
    String sessionId = ((LogonResponse) response.getSuccessResponse()).getSessionId();
    ropClient.setSessionId(sessionId);
    return sessionId;
}

public void logout() {
    ropClient.buildClientRequest().get(LogonResponse.class, "user.logout", "1.0");
}

public ClientRequest build(){
    return ropClient.buildClientRequest();
}
}

```

将服务平台的登录、退出等方法直接封装到 RopSampleClient 中。

这样，开发者只需指定服务平台分配的 appKey/secret，就可实现化出一个可用的 RopSampleClient。RopSampleClient 已经在内部固化了服务平台的 URL 地址，自动注册了平台相关的 RopConvertor。这样，服务平台的 URL、系统级参数名称以及使用了哪些 RopConvertor，对应用开发者来就完全透明了。调用 RopSampleClient#logon() 登录服务平台后，就可以通过 RopSampleClient#build() 获取 ClientRequest 对象进行平台服务的调用了。

最后，把 RopSampleClient、服务平台的请求/响应对象、以及平台使用到的 RopConvertor 打成一个 JAR 包，这就是服务平台所对应的 Java 版 SDK 了。

12 参考资料

1. TOP: <http://api.taobao.com/apidoc/main.htm>;
2. 豆瓣 API: <http://developers.douban.com/wiki/?title=guide>
3. Spring Framework: <http://www.springsource.org>
4. Spring MVC 设计原理: 《Spring 3.x 企业应用开发实战》