**PRIMETON TECHNOLOGIES, LTD.**

**普元信息技术股份有限公司**

MetaCube 6.0

**元数据服务引擎对外接口**

**总体设计方案**

文档OARP

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 分类 | 姓名 | 范围要求 |
| 1 | Owner | 吴艳伟 |  |
| 2 | 作者 |  |  |
| 3 | 审核人 |  |  |
| 4 | 审核人 |  |  |
| 5 | 审核人 |  |  |
| 6 | 批准人 |  |  |

文档修订记录

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 序号 | 版本号 | 修订日期 | 修订概述 | 修订人 | 审核人 | 批准人 | 备注 |
| 1 | 0.9 | 2015.4.3 |  | 朱祥坤 |  |  |  |
|  |  |  |  |  |  |  |  |

**目录**

[1． 引言 1](#_Toc417481102)

[1.1. 目的 1](#_Toc417481103)

[1.2. 阅读对象 1](#_Toc417481104)

[2． 总述 1](#_Toc417481105)

[2.1． 技术说明 1](#_Toc417481106)

[2.2． 第三方组件依赖 2](#_Toc417481107)

[3． 逻辑视图 3](#_Toc417481108)

[3.1． 客户端与元数据引擎的关系 3](#_Toc417481109)

[3.2． 元数据服务引擎中各模块的关系 4](#_Toc417481110)

[3.3． 元数据接口模块中组件列表 4](#_Toc417481111)

[3.4． 元数据接口模块中各组件与元数据服务引擎中各模块的关系 5](#_Toc417481112)

[4． 部署方案 6](#_Toc417481113)

[4.1. 总体部署方案 6](#_Toc417481114)

[4.2. 元数据引擎部署方案 7](#_Toc417481115)

[4.3. 负载均衡 7](#_Toc417481116)

[5． 运行视图 7](#_Toc417481117)

[5.1. 元数据接口模块设计 7](#_Toc417481118)

[5.1.1. 接口分类 7](#_Toc417481119)

[5.1.2. 调用方式 7](#_Toc417481120)

[5.1.3. 报文格式 8](#_Toc417481121)

[5.1.4. 时序图 12](#_Toc417481122)

[5.1.5. 模型 13](#_Toc417481123)

[5.1.6. 异常处理 13](#_Toc417481124)

[5.1.7. 日志记录 13](#_Toc417481125)

[5.1.8. Util 14](#_Toc417481126)

[5.2. 客户端jar包设计 15](#_Toc417481127)

[5.2.1. API 15](#_Toc417481128)

[5.2.2. 模型 15](#_Toc417481129)

[5.2.3. 调用过程 15](#_Toc417481130)

[5.2.4. 客户端jar包 17](#_Toc417481131)

[5.3. 功能扩展 18](#_Toc417481132)

[5.3.1. 数据补全 18](#_Toc417481133)

[5.3.2. 通讯安全 18](#_Toc417481134)

[5.3.3. 用户权限认证 19](#_Toc417481135)

[5.3.4. 数据脱敏 20](#_Toc417481136)

[6． 附录1 — REST架构介绍 21](#_Toc417481137)

[6.1． REST概念 21](#_Toc417481138)

[6.2． REST 架构风格定义的约束 21](#_Toc417481139)

[6.2.1. 无状态通信 21](#_Toc417481140)

[6.2.2. 缓存 22](#_Toc417481141)

[6.2.3. 统一的接口 23](#_Toc417481142)

[6.3． Restlet框架使用 25](#_Toc417481143)

[6.3.1. Resource的设计原则 25](#_Toc417481144)

[6.3.2. Resource的标准接口 25](#_Toc417481145)

# 引言

## 目的

文档阐述了元数据服务引擎接口模块的总体设计思路，对系统逻辑视图、部署视图做出说明，同时对架构风格以及相应技术规范等做出约定，帮助项目成员快速了解系统运行原理并保证产品整体质量。

## 阅读对象

本设计说明书的读者为以下人员：

* 软件设计人员，职责为：

根据总体设计说明进行详细设计

* 软件开发人员，职责为：

根据详细设计的内容编写各种具体实现的代码。

* 软件单元测试人员，职责为：

根据详细设计中的设计及相关单元测试内容，进行单元测试，包括编写各种代码和人工进行的单元测试。

* 项目评审人员，职责为：

检查当前说明书是否能够正确的开发软件，与需求有无偏差和遗漏。

* 系统集成人员，职责为：

根据当前详细设计中的部署设计，能够成功的集成系统。

# 总述

元数据服务引擎以HTTP + JSON的方式对外提供服务。客户端通过发送HTTP请求调用元数据服务引擎的接口，并通过JSON传输参数。元数据服务引擎对接收到的客户端请求进行处理，然后使用JSON返回响应结果。客户端与引擎之间的请求与响应都是通过JSON进行数据传输。

## 技术说明

元数据服务引擎使用Restlet框架来暴露HTTP接口，使用FastJson工具来实现POJO的序列化与反序列化。为了更方便地使用Restlet，我们将之与Spring集成在一块。

## 第三方组件依赖

要依赖的第三方组件包括：Restlet 2.2.3、Spring 2.5.6、FastJson 1.2.5，具体jar包列表如下：

spring.jar

org.restlet.jar

org.restlet.ext.servlet.jar

org.restlet.ext.spring.jar

fastjson-1.2.5.jar

cglib-nodep-2.1\_3.jar

commons-logging-1.1.1.jar

对于其他依赖的组件要求：

1. 必须使用成熟的商业或开源产品

2. 如果开源，必须具有可商用的License

由于这次使用了第三方中间件的不同版本，需要对升级或者版本统一进行评估。

# 逻辑视图

## 客户端与元数据引擎的关系



图3-1. 客户端与元数据引擎关系图

图3-1描述了客户端与元数据服务引擎及元数据采集引擎的逻辑关系。元数据服务引擎对外暴露HTTP + JSON的接口供客户端调用。客户端在Service层中调用元数据服务引擎，元数据服务引擎得到采集元数据的命令之后通过RMI方式调用元数据采集引擎，然后元数据采集引擎通过JDBC的方式从数据中心将元数据采集入库。

客户端调用元数据服务引擎时传输的数据以及元数据服务引擎返回的数据都是JSON格式。而在这过程中POJO的序列化与反序列化操作都通过元数据服务引擎所提供的jar包实现，内部技术细节对客户端是透明的。

## 元数据服务引擎中各模块的关系



图3-2. 元数据服务引擎中各模块关系图

图3-2描述了元数据服务引擎中各模块的关系，元数据产品是由各个模块组成的，而本工程也是以一个独立的模块与其他模块共同组成元数据产品。所以在开发本工程时按照一个独立的Java Project开发即可。

元数据服务引擎接口模块只是用于对外暴露HTTP + JSON接口，所有功能实现都是通过对元数据现有模块的调用来实现的，所以接口模块与元数据服务引擎中其他模块之间存在依赖关系，这些依赖关系详情请见3.4章节“元数据接口模块中各组件与元数据服务引擎中各模块的关系”。

在部署时将本工程加入到com.primeton.dgs.web工程中即可。

## 元数据接口模块中组件列表

元数据服务引擎中所有需要对外提供服务的模块都在本接口模块中对应于一个组件，本期接口模块先将部分模块包装为REST接口，具体的组件名称及其含义请见表3-1.

|  |  |
| --- | --- |
| 组件名称 | 含义 |
| com.primeton.dgs.workspace.rest.metadata | 元数据管理组件，包括元数据维护以及元数据依赖关系查询 |
| com.primeton.dgs.workspace.rest.metadata.statistics | 元数据统计信息查询组件 |
| com.primeton.dgs.workspace.rest.metadata.history | 元数据历史信息查询组件 |
| com.primeton.dgs.workspace.rest.datasource | 元数据数据源管理组件 |
| com.primeton.dgs.workspace.rest.extractor | 元数据采集任务管理组件 |
| com.primeton.dgs.workspace.rest.common | 元数据接口通用组件 |
| com.primeton.dgs.workspace.rest.util | 元数据接口工具组件 |

表3-1. 元数据接口模块中各组件列表

## 元数据接口模块中各组件与元数据服务引擎中各模块的关系

元数据服务引擎中所有需要对外提供服务的模块都在本接口模块中对应于一个组件，这些组件与相应的元数据服务引擎模块之间存在依赖关系，接口模块的各组件需要调用它所依赖的元数据模块对外提供服务。对于这些依赖的元数据模块的调用都是调用它们的Service而不是Command。

图3-3描述了元数据接口模块中各组件与元数据引擎中各模块之间的依赖关系，左侧部分是元数据接口模块中的组件名称，右侧部分是元数据引擎中的模块名称。

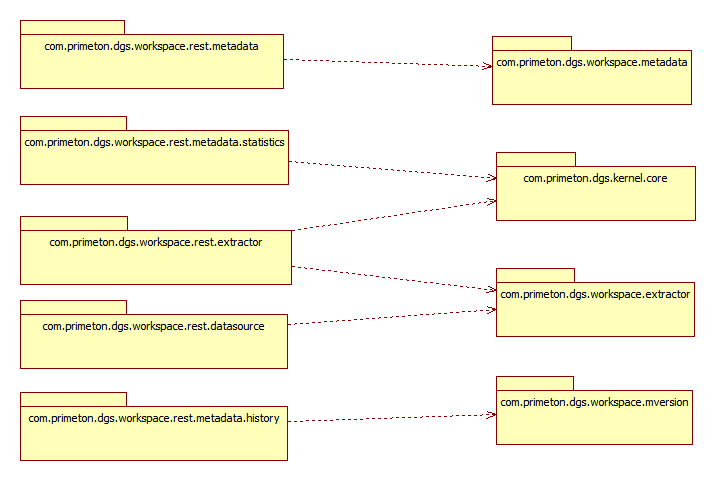


图3-3 元数据接口模块中各组件与元数据引擎中各模块的关系

# 部署方案

## 总体部署方案



图4-1. 客户端与元数据引擎部署方案图

图4-1描述了客户端与元数据引擎部署方案，访问请求将首先通过第一级负载均衡到达客户端。客户端与元数据服务引擎间有第二级负载均衡。如果是采集任务的调用，则元数据服务引擎将通过RMI的方式调用元数据采集引擎。

所有元数据服务引擎共用一个元数据采集引擎，也共用一个共享目录，上传的Excel模板无论是经过哪个服务引擎的处理都将保存在共享目录中为所有服务引擎所共用。

## 元数据引擎部署方案

元数据引擎包括服务引擎以及采集引擎，元数据服务引擎中包括元数据接口模块以及现有的元数据产品模块（采集模块除外），元数据采集引擎就是元数据采集模块。

元数据接口模块以及它所相关的kernel模块、workspace模块在部署时都将打包为jar包，这些jar包以及它们所依赖的第三方组件的jar包都将放在用web模块所打成的war包中。这个war包就是元数据服务引擎在部署时的外在表现形态。

元数据采集模块在部署时会打包为jar包，这些jar包以及它所依赖的第三方组件jar包会共同打包为一个普通压缩包。这个压缩包就是元数据采集引擎在部署时的外在表现形态，我们只需将其解压部署即可。

## 负载均衡

负载均衡部署可参考F5、nginx等负载均衡解决方案。

# 运行视图

## 元数据接口模块设计

### 接口分类

元数据接口模块总体上分为元数据操作接口、元数据分析接口以及元数据控制接口。元数据操作接口用于处理对元数据的维护操作，元数据分析接口用于处理对元数据的统计查询、血统分析查询、影响分析查询等各种分析查询操作，元数据控制接口用于处理对元数据系统的控制操作，比如元模型的管理、采集模板的管理以及采集任务的管理等管理操作。

### 调用方式

元数据服务引擎暴露的是HTTP接口，并提供4种标准接口，即GET、PUT、POST与DELETE方法，对应着对资源的4种操作，即获取、创建、修改和删除。我们在调用接口时需要使用目标资源的URI以及相关参数及用户登录信息声明一个客户端资源（如同一个目标资源的客户端代理），再使用这个客户端资源对这4种方法进行调用即可。

详细的调用过程及示例请见第5.2.3章节“[调用过程](#_调用过程)”。关于资源类Resource以及其中接口的设计请见第6.3.1章节“[Resource的设计原则](#_Resource的设计原则)”以及第6.3.2章节“[Resource的标准接口](#_Resource的标准接口)”。

### 报文格式

本章节主要阐述元数据服务引擎对外REST服务的发送报文和响应报文的通用格式，具体每个接口的输入参数和输出参数将在元数据服务引擎对外REST接口详细设计文档中按照接口逐个描述，此处不展开描述。

#### 发送报文

【报文样例】

|  |
| --- |
| {  "messageHead":{  "id":"",  "uri":"",  "user":{  "userName":"",  "userCode":"",  "userExtensions":[]  },  "headExtensions":[]  },  "messageValue":{  "parameters":{  },  "valueExtensions":[  "page":{  "startIndex":"",  "pageSize":"",  "totalCount":"",  "currentCount":""  }  ]  }  } |

【报文结构】

请求报文包含两个信息块，功能如下：

MessageHead（报文消息头）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 标识 | 描述 | 类型 | 必填 | 默认值 | 其他 |
| id | 请求唯一编号，用于唯一定位一次服务请求 | 字符串 | 是 | --- | 32位UUID字符串 |
| uri | 请求目标资源的URI地址 | 字符串 | 是 | --- |  |
| user | 用户信息对象 | JSON对象 | 是 | --- |  |
| headExtensions | 请求头扩展信息 | JSON对象集合 | 否 | --- |  |

User（用户信息）：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 标识 | 描述 | 类型 | 必填 | 默认值 | 其他 |
| userName | 请求用户标识 | 字符串 | 是 | --- | 用户ID信息 |
| userCode | 用户信息数字签名 | 字符串 | 否 | --- | 扩展信息，当需要对用户信息防篡改等安全操作时增加该信息。签名信息可以与扩展信息中字段组合签名。 |
| userExtensions | 用户扩展信息 | JSON对象集合 | 否 | --- |  |

MessageValue（请求消息体）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 标识 | 描述 | 类型 | 必填 | 默认值 | 其他 |
| parameters | 请求参数 | JSON对象 | 是 | --- | 请求指定资源时调用的参数清单，内容视具体服务参数列表而定。 |
| valueExtensions | 请求体扩展信息 | JSON对象集合 | 否 | --- |  |

Page（分页信息）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 标识 | 描述 | 类型 | 必填 | 默认值 | 其他 |
| startIndex | 查询起始位置索引 | 字符串 | 是 | --- |  |
| pageSize | 每一页的大小 | 字符串 | 是 | --- |  |
| totalCount | 查询结果集总数 | 字符串 | 是 | 0000 |  |
| currentCount | 查询结果当前页数量 | 字符串 | 是 | --- |  |

#### 响应报文

【报文样例】

|  |
| --- |
| {  "messageHead":{  "id":"",  "uri":"",  "code":"",  "message":{  "title":"",  "content":""  },  "headExtensions":[]  },  "messageValue":{  "results":[],  "valueExtensions":[  "page":{  "startIndex":"",  "pageSize":"",  "totalCount":"",  "currentCount":""  }  ]  }  } |

【报文结构】

请求报文包含两个信息块，功能如下：

MessageHead（报文消息头）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 标识 | 描述 | 类型 | 必填 | 默认值 | 其他 |
| id | 请求唯一编号，用于唯一定位一次服务请求 | 字符串 | 是 | --- | 32位UUID字符串 |
| uri | 请求目标资源的URI地址 | 字符串 | 是 | --- |  |
| code | 返回值状态码 | 字符串 | 是 | 0000 | 默认值为“0000”表示未处理，取值范围为0000～9999 |
| message | 操作返回结果信息对象 | JSON对象 | 是 | --- | 描述当前操作返回结果 |
| headExtensions | 请求头扩展信息 | JSON对象集合 | 否 | --- |  |

Message（返回结果信息）：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 标识 | 描述 | 类型 | 必填 | 默认值 | 其他 |
| title | 请求用户标识 | 字符串 | 是 | --- | 消息基本描述，200字符内 |
| content | 用户信息数字签名 | 字符串 | 否 | --- | 消息内容，长度不超过2000，如果发生异常，可以为异常堆栈信息，但需要注意堆栈长度 |

MessageValue（请求消息体）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 标识 | 描述 | 类型 | 必填 | 默认值 | 其他 |
| results | 返回值结果 | JSON对象或者数组 | 否 | --- | 结果视方法执行情况而定。 |
| valueExtensions | 请求体扩展信息 | JSON对象集合 | 否 | --- | 包含分页信息等非方法直接返回信息。 |

Page（分页信息）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 标识 | 描述 | 类型 | 必填 | 默认值 | 其他 |
| startIndex | 查询起始位置索引 | 字符串 | 是 | --- |  |
| pageSize | 每一页的大小 | 字符串 | 是 | --- |  |
| totalCount | 查询结果集总数 | 字符串 | 是 | 0000 |  |
| currentCount | 查询结果当前页数量 | 字符串 | 是 | --- |  |

#### 消息状态码

元数据接口中的消息状态码都采用4位阿拉伯数字定义，取值范围为0000-9999。具体分类如下：

0000：请求未处理

1000 – 1999：响应成功的状态码

2000 – 2999：程序出现异常的状态码

在这些分类中再按照不同模块进行状态码细分。

### 时序图

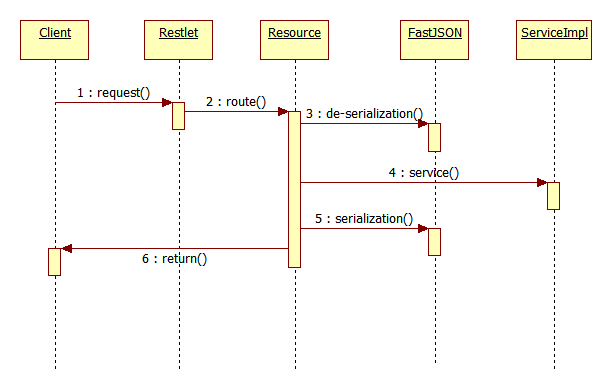


图5-1 调用时序图

|  |  |
| --- | --- |
| 步骤 | 说明 |
| 1:request | 客户端发起请求，包括GET、POST、PUT、DELETE |
| 2:route | Restlet对客户端请求进行路由分发到相应Resource |
| 3:de-serialization | 调用FastJSON将客户端请求参数进行反序列化并返回 |
| 4:service | 调用元数据实现Service进行业务操作并返回结果 |
| 5:serialization | Resource将元数据实现Service返回的处理结果进行序列化操作并返回 |
| 6:return | 元数据接口将响应结果JSON字符串返回给客户端 |

表5-7 调用时序图说明

在这个过程中，Restlet的路由器与资源Resource类都注册在Spring中，由Spring容器对其进行统一管理。这样便于请求的分发以及调用过程的处理。

在Restlet框架中根据URI进行路由分发的工作是由Router（路由器）实现的，在接收到用户请求之后，Router根据URI模板对用户请求进行委派调用。通过与Spring的集成，我们将Router的配置与管理交给Spring来托管，我们只需关注Restlet框架最核心的Resource的接口设计即可。

除了将Router交给Spring托管之外，我们把Resource也交给Spring来管理，因此我们便可通过Spring来实现Resource与其他类的调用。

### 模型

元数据接口工程中定义了一些模型，这些模型总共可以分为2类，一类是常量及通用模型，比如表示报文的Response、表示消息内容的Message、表示分页参数的Page以及表示客户端用户的User。另一类就是元数据引擎中的一些业务模型，比如元数据模型、依赖关系模型、统计信息模型、数据源模型、采集任务模型等。这些模型的具体属性将在详细说明书中进行定义。

### 异常处理

元数据接口在处理异常时的总原则如下：

1. 如果出现的异常不影响程序的运行就只将异常记录到日志文件中，不进行返回
2. 如果出现的异常影响程序的运行就将异常返回，并记录到日志文件中

在对Restlet、FastJSON的封装类中出现的异常的处理原则如下：

1. 如果异常影响程序运行就将异常封装为检查性异常并抛出，强制外层调用者捕获异常
2. 如果异常不影响程序运行就直接在封装类中将异常记录到日志文件中，不进行返回

### 日志记录

本接口中统一使用log4j来记录日志，需要记录日志的地方有：

1. 在客户端请求进入Resource的各接口时对本次客户端请求进行记录，以便后续对客户端请求进行跟踪分析
2. 如果程序出现异常，不论异常是否需要返回都要将其记录到日志文件中
3. 在接口方法中，对于必要的核心信息需要手动记录日志内容

由于元数据接口工程只是元数据系统的一个模块，因此输出日志格式统一由元数据系统来约束，本处只规定日志消息内容的格式。

本日志消息内容格式只用于约束日志记录中所描述的第1种情况，即对客户端请求的跟踪记录，另外2处如对异常信息的记录以及核心信息的记录不遵从此约束。

本日志消息内容格式为“URI|请求编码|用户名|返回状态码”，其中各段的含义如表5-8所示。

|  |  |
| --- | --- |
| 标识 | 说明 |
| URI | 请求URI |
| 请求编码 | 客户端请求编码，即客户端请求唯一标识ID |
| 用户名 | 客户端请求的用户名 |
| 返回状态码 | 元数据接口响应状态码 |

表5-8 日志消息内容格式

### Util

#### JSONUtil

本工程中对POJO的序列化与反序列化使用的是FastJSON，它有着同类工具中最高的性能。但是本工程中需要在URI中传递JSON参数，而JSON字符串中有些特殊字符是无法在URI中进行传递的，需要将其进行编码。在Server端接收到参数后需要先解码然后再将之反序列化为POJO。

本工程将原生JSON进行封装，提供JSONUtil类，定义了2个静态方法以便对POJO的序列化/反序列化以及编码/解码操作。由于是继承自FastJSON的JSON类，因此其中的方法都得以保留，在使用时直接将此类代替原生JSON类使用即可。其类图如图5-2所示：

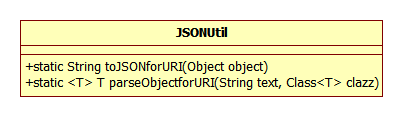


图5-2 JSONUtil类图

#### ClientResourceUtil

元数据接口工程提供了一个ClientResourceUtil类负责方便地从客户端请求中提取参数，其类图如图5-3所示：

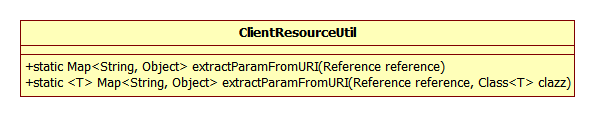


图5-3 ClientResourceUtil类图

## 客户端jar包设计

### API

#### MyClientResource类

REST架构风格是一种“客户端—服务器”架构风格，也就是说客户端发起请求，服务器处理请求并返回相应的结果。在Restlet框架中提供了ClientResource类实现客户端请求的发送，但是本工程中在向客户端发送请求时需要传递一些过滤条件以及分页查询条件所组成的参数，并且要在POST、PUT方法的消息体中传输对象序列化后的JSON字符串，为了简化这些功能的使用，将原生的ClientResource类进行了封装，提供MyClientResource类，其中包括如下6个构造方法、2个公共方法。其模型如图5-3所示：

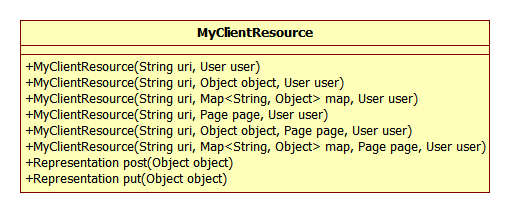


图5-3 MyClientResource模型

### 模型

元数据接口中定义的模型都会提供给客户端，这些模型的具体定义请参考《元数据服务引擎对外接口详细设计说明书》。

### 调用过程

元数据服务引擎暴露的是HTTP接口，并4种标准接口，即GET、PUT、POST与DELETE方法，对应着对资源的4种操作，即获取、创建、修改和删除。

#### 示例1：GET方法调用

GET方法通用用于对指定ID的资源的查询或者对资源列表的查询，其调用过程示例如下：

1. 定义ClientResource，这就是用于访问服务引擎的客户端资源，就像是服务引擎中资源的代理。定义实例如下：



这个例子定义了一个查询列表的ClientResource，并且使用了过滤查询以及分页查询。其中第1个参数是目标资源的URI，第二个参数是对目标资源列表查询的过滤条件所封装的对象，第三个参数是分页参数所封装的对象，第四个参数是客户端用户对象。

如果是对指定ID的资源的查询、新增、修改、删除都只需第一个与第四个参数即可。

1. 调用接口发送请求

调用GET方法：



1. 获取响应结果

对接口调用的响应结果都在Representation对象中，我们通过getText()方法便可获取其中的响应结果数据，如下所示：



#### 示例2：POST方法调用

POST方法通常用于资源的新增，其调用过程示例如下：

1. 定义ClientResource，指明我们要访问的目标资源URI并标识客户端用户



1. 调用接口发送请求

调用POST方法，比如我们使用如下方式新增一个Student对象



1. 获取响应结果

对接口调用的响应结果都在Representation对象中，我们通过getText()方法便可获取其中的响应结果数据，如下所示：



#### 示例3：PUT方法调用

PUT方法通常用于对资源的修改，其调用过程示例如下：

1. 定义ClientResource，指明我们要访问的目标资源URI并标识客户端用户。下面这个例子中指明了我们的目标资源时ID为2的Student。



1. 调用接口发送请求

调用PUT方法，比如我们使用如下方式修改ID为2的Student对象，ID在上一步中定义。



1. 获取响应结果

对接口调用的响应结果都在Representation对象中，我们通过getText()方法便可获取其中的响应结果数据，如下所示：



#### 示例4：DELETE方法调用

DELETE方法通常用于对资源的删除，其调用过程示例如下：

1. 定义ClientResource，指明我们要访问的目标资源URI并标识客户端用户。下面这个例子中指明了我们的目标资源时ID为1的Student。



1. 调用接口发送请求

调用DELETE方法，比如我们使用如下方式删除ID为1的Student对象，ID在上一步中定义。



1. 获取响应结果

对接口调用的响应结果都在Representation对象中，我们通过getText()方法便可获取其中的响应结果数据，如下所示：



### 客户端jar包

在客户端jar包中要包括JSONUtil、MyClientResource以及元数据模型，同时还要为客户端提供这个jar包所依赖的jar包，包括org.restlet.jar、fastjson-1.2.5.jar等。

## 功能扩展

### 数据补全



图5-4 数据补全时序图

DataFillingFilter接口为可选过滤器，用于对传入报文中未填而必填字段按照规则补全。如图所示，在数据解析之后服务调用前，可以通过DataFillingFilter对当前数据按照一定规则补全，如果未配置DataFillingFilter，则不会进行数据补全操作。

### 通讯安全

Http+JSON具备极为灵活的调用方式，与此同时，对服务安全也提出较高的考验。本文提供可选安全处理方案，用于解决下述几种安全问题：

1. 身份识别
2. 抵抗重放
3. 数据保密与防篡改

这里的通讯安全组件为必选组件，通过**数字签名+对称加密**方式实现。调用流程如图5-5所示：



图5-5 通讯安全时序图

元数据接口所提供的客户端jar包中要包括签名组件和加密组件，以便提供给客户端对User对象进行签名和加密处理。

### 用户权限认证

用户权限认证接口用于确认当前用户是否具备该服务接口的使用权限，该接口为可选组件，如果不配置，则默认用户可以使用当前接口。调用过程如图5-6所示：



图5-6 权限认证时序图

### 数据脱敏



图5-7 数据脱敏时序图

数据脱敏功能包含对敏感字段删除或者混淆等操作。DataDesensitizationFilter接口为可选过滤器。如上图所示，在服务调用之后可以通过DataDesensitizationFilter接口的accept()方法对返回值进行脱敏操作，如果未配置DataDesensitizationFilter，则不会进行任何脱敏处理。

# 附录1 — REST架构介绍

## REST概念

REST是Roy Fielding博士于2000年在他的博士论文中提出来的一种软件架构风格。它是一种针对网络应用的设计和开发方式，可以降低开发的复杂性，提高系统的可伸缩性。在理解 REST 相关的核心概念之前，先看看REST本身的意义。REST是Representational State Transfer（表述性状态转移）的简称，更全面的理解应该是Application States Transfer among the resource ’ s representation（资源表述中的应用状态转移）。

## REST 架构风格定义的约束

REST 是一种架构的风格，它是对分布式超媒体系统的架构元素的抽象，提供了一些核心的概念和指导思想。这种架构风格是“客户端—服务器”架构风格的一种，也就是说客户端发起请求，服务器处理请求并返回相应的结果。REST 的贡献是规定了一系列的方法论，在请求方面，规定客户端怎么发起请求，发的是什么样的请求，以什么协议发请求；在处理方面，规定服务器怎么响应请求，返回什么样的响应，系统后续应该怎么跳转等等。下面就是一些REST架构的约束。

### 无状态通信

客户端和服务器端的交互是无状态的，也就是说请求之间是相互隔离的。服务器端不保存客户端的应用上下文（context），每个从客户端来的请求都必须包括所有的必要的信息让服务器端能够完全理解请求并处理。客户端存了很多会话的信息。让我们以搜索引擎为例来看看有状态和无状态的区别。图 6-1 和图 6-2 分别给出了两个搜索引擎的有状态和无状态的交互示例。

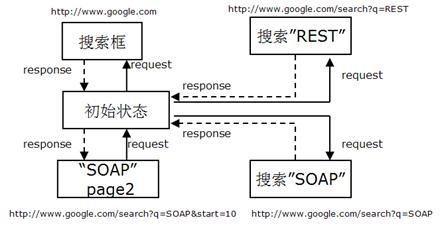


图6-1. 无状态的搜索引擎的交互示例

图 6-1 所示是无状态的搜索引擎的请求（request）和响应（response）的实例。可以清楚地看出，每个request和response都是互相独立的，相互之间没有数据的依赖。每个 request 包含服务器端响应这个request所需要的所有信息。

以搜索“SOAP”为例，用户首先发了request “http://www.google.com/search?q=SOAP”，并且得到了搜索结果，其中包含了 10个最相关的搜索结果。这个交互过程就结束了，服务器端没有保存任何和这个请求相关的信息。但是在这个返回的状态中，服务器端把下一步的可能的状态嵌在其中了。比如用户如果在这些结果没有找到自己想要的结果，他可以翻页，翻到第二页。第二页是另一个状态，这时用户点击了第二页，然后客户端又发了一个request

“http://www.google.com/search?q=SOAP&start=10”，这个request包含了所有的上下文，也就是按关键字SOAP搜索并且以第10个搜索结果开始返回。也就是说，服务器把当前的状态隐含在结果中返回，客户端保存下这些隐含的状态，而不是保存在服务器端。客户端可以根据服务器端返回来的状态构建下一个状态的请求。

无状态的好处是每个状态都有一个对应的URI，这些URI可以bookmark，可以使得用户方便地在浏览器中前进后退，可以在用户希望的任何时候访问任意多次。

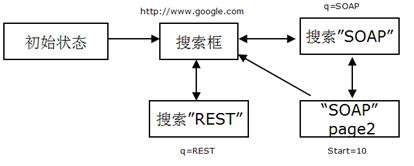


图 6-2. 有状态的搜索引擎的交互示例

图 6-2 是有状态的搜索引擎的request和response的交互示例。可以看出，request之间的时序依赖性。以搜索SOAP为例，用户在看第1-10个搜索结果然后想看11-20个结果，客户端只需要发一个“start=10”的请求而不是发“/search?q=SOAP&start=10”的请求，也就是客户端不用重复前面的request的上下文。这使得HTTP request相对简单了很多。但是它使得HTTP protocol变得复杂。服务器端和客户端需要同步会话的状态，在可靠网络上，这是一个复杂的任务。

无状态带来了一些性能的提升。在visibility（可见性）方面，对于监控系统而言，它不用去关心跨请求的数据对当前请求的影响，所以visibility有所提升。在reliability（可靠性）方面，由于客户端保存了很多会话数据，因此减少了部分服务器端的故障或者网络故障对应用的影响，因此对于用户来说，reliability有所提升。最值得一提的是scalability（可伸缩性），由于服务器端能够独立地响应每个request，而不用依赖会话历史，因此少了很多资源的管理和同步，使得多个服务器可以同时服务不同的用户的请求，获得很好的自由扩展功能。

当然，事物都有两面性，无状态带来的不足之处包括可能的网络性能的降低，因为服务器会发很多重复的数据到不同的客户端，使得客户端保存所有的会话信息；这也导致了另一个问题，就是服务器将失去对应用的一致行为的一部分控制权，而且还对客户端的实现产生依赖。

### 缓存

为了提高REST风格架构的网络性能，Roy加入了缓存的约束。缓存不是一个新的概念，HTTP协议提供机制，使得客户端可以缓存一些数据。在服务器返回的响应中，可以隐式或者显式的指明数据的缓存属性。这样，客户端可以在未来使用缓存下来的数据，减少客户端和服务器端的交互次数，从而达到提高网络性能的目的。

### 统一的接口

REST风格的架构区别于其他架构的一个最关键的指标，就是 REST 风格的架构对于任意应用，规定了统一的接口定义。接口定义包括以下4个部分：

* 资源标识符
* 通过资源的表述操作资源
* 自描述信息
* 超媒体作为应用状态的引擎

#### 资源标识符

REST架构的最核心的概念就是Resource。Resource是信息系统的一种抽象，可以是任何重要的足以把本身作为一个引用（reference）的事物。从应用的角度看，如果应用的用户需要创建一个到它的链接，获取或者缓存它的表述，或者想对它做些操作，那么都可以创建一个Resource。一个Resource可以是现实世界的事物，比如一本书、一辆车、一栋房子；也可以是一个虚拟的概念，也可以是一个算法的结果。

资源标识符是Resource的全局唯一标识，REST使用一个资源标识符来标识组件之间交互所涉及到的特定资源。如果Resource没有资源标识符，那么它不能称为Resource。在REST架构中使用URI来表示一个Resource 的资源标识符。关于 URI 的最佳实践包括URI是自描述的、有结构的，这样可以使得客户端可以自己创建一些有预测性的请求。几个比较好的 Resource URI 示例：

http://www.example.com/books/123456

http://www.example.com/softwares/im/db2/9.5

http://www.example.com/blog/2010/09/10/1

http://www.example.com/bugs/by-state/new

对于REST架构中最重要的Resource与资源标识符两者之间的关系，我们需要注意两点：

1. 两个 URI 可以指向同一个 Resource

比如我们在发布程序时可能用到如下两个URI: http://www.example.com/releases/3.1.1 和 http://www.example.com/releases/latest，在特定的时刻，他们指向的就是同一个 resource，但是背后的概念是不一样的，一个是指向一个特定的 release 版本号，一个是指向一个随着时间演进的 resource，是两个完全不同的概念。所以一个 resource 可以有一个或者多个 URI 来标识。

1. 一个 URI 不能指向不同的 Resource ，因为这和统一资源标识符的原则相违背。

#### 通过资源的表述操作资源

REST组件通过以下方式在一个资源上执行动作：使用一个表述来捕获资源的当前的或预期的状态、在组件之间传递该表述。一个表述是一个字节序列，以及描述这些字节的表述元数据。表述的其他常用但不够精确的名称包括：文档、文件、HTTP 消息实体、实例或变量，或者更直白地说它可以是JSON、XML、HTML，也可以是一个文件。一个 Resource 可以有多种表述。由于元数据引擎以HTTP + JSON的方式对外提供服务，所以我们只使用JSON这一种表述。

REST对Resource的操作定义和HTTP method 是对应起来的。HTTP协议定义GET、POST、PUT、DELETE来表示对资源的获取、创建、更新和删除。“通过资源的表述操作资源”这一约束规定了Resource的操作是通过表述来完成的，所以在Resource的表述里面，需要包含对Resource的操作怎么进行。

#### 自描述信息

REST接口的定义强调了自描述性。自描述性也是为了让客户端充分的理解当前的状态，下一步的状态怎么跳转。

前面讲了Resource的表述，在表述里面除了包含数据部分，还包括用来描述数据的元数据，或者是一些客户端需要知道的一些领域知识。通常包括翻页信息：一共有多少页，前后翻页的链接是什么；分类信息、时间信息，还有一些文本信息供用户阅读。

总之，服务器端尽可能的返回详细的信息，帮助客户端理解当前的状态，以及发起下一个请求所需要的所有信息，以达到“服务器端无状态”和客户端发起“状态跳转”的目的。

#### 超媒体作为应用状态的引擎

这个约束规定的是应用系统是通过超媒体的方式在不同的状态之间跳转。超媒体的实质是超链接，用超链接把这些相互依赖的Resource联系起来，这些超链接是由服务器端生成并且在表述里面返回来的，包括了当前的状态集合和可能的下一步的状态集合。客户端不需要任何领域知识就能够实现状态的跳转。

通过以上这些统一接口的约束，REST风格的架构使得web服务的客户端和服务器端很好地分离开来。客户端不需要关心数据的存储，使得客户端的可移植性提高了。服务器端不用关心用户的接口和用户的状态，所以服务器端将变得更加简单，而且方便的获得更好的可伸缩性。只要保持接口的定义不变，客户端和服务器端可以独立的开发和演变。

## Restlet框架使用

### Resource的设计原则

在REST架构中Resource是信息系统的一种抽象，可以是任何重要的足以把本身作为一个引用（reference）的事物。由于统一接口的约束，所有的Resource都提供了标准的接口，即GET、PUT、POST与DELETE方法，对应着对资源的4种操作，即获取、创建、修改和删除。因此对于Resource的设计就显得格外重要。我们需要针对业务需求，对Resource做出合理的规划。首先根据资源是否是单一资源，我们将Resource分为单一资源的Resource与一组资源的Resource。

#### 单一资源的Resource

在REST架构中规定了统一的接口定义，每一个Resource都只有4种方法。为了应对复杂的业务逻辑，我们把Resource分为单一资源的Resource以及一组资源的Resource。单一资源的Resource用来处理对指定ID的资源的操作，一组资源的Resource用来处理对所有资源组或基于其他条件（比如姓名、性别）过滤后的资源组的操作。

单一资源的Resource，其URI定义为”/resources/sudent/{id}”，”{id}”就是此资源的ID，这个URI就是此资源的全局唯一标识。然后在此Resource中定义GET、PUT、POST与DELETE方法来处理对此ID的资源的获取、创建、更新和删除。

对于此ID参数的获取，我们在doInit方法中完成，如下所示：

protected void doInit() throws ResourceException {

id = (String) getRequestAttributes().get("id");

}

#### 一组资源的Resource

一组资源的Resource用来处理对所有的资源组或基于其他条件（比如姓名、性别）过滤后的资源组的操作。这个Resource的URI可以定义为”/resources/sudents”，然后在其中定义GET、PUT、POST与DELETE方法来处理对此资源组的获取、创建、更新和删除。

### Resource的标准接口

在REST架构中，所有的Resource都提供了标准的接口，即GET、PUT、POST与DELETE方法，对应着对资源的4种操作，即获取、创建、修改和删除。以下是这4种方法的使用说明。

#### GET方法

GET方法通常用来表示检索一个表述，也就是对资源的描述。我们使用”@Get”注解把这个方法定义为GET方法。在GET方法使用中需要注意的是响应结果、分页查询与模糊查询。

1. 过滤查询

对资源组的查询（即获取）支持过滤查询，这些条件都通过URI进行传递，比如”/resources/sudents?fileParam={filgerParam}”。为了将数据传输格式化，本工程中需要将过滤条件封装为POJO，然后将其序列化为JSON字符串放入URI。

需要注意的是，在POJO中变量不能使用基本类型，应该使用其对应的封装类。因为基本类型是有默认值的，如果没有对其赋值，在将其序列化时会出现在JSON字符串中。定义为封装类，不对其赋值那么默认为null，这是不会出现在JSON字符串中的。

其次，JSON字符串中有些特殊字符是不能在URI中使用的，因此我们将POJO序列化后的JSON字符串进行了编码处理，再放入URI中使用。所以在Server端解析JSON时就要先解码然后再反序列化为POJO。为了方便使用，本工程将这些操作都封装在jar包，对使用者透明。

1. 响应结果

在GET方法中，单一资源的Resource返回的是空对象或一个对象，一组资源的Resource除此之外还会返回一个资源的集合。但是不管是一个对象还是集合，我们都不能将这个对象直接返回，而是将其进行序列化操作，返回JSON字符串。

REST架构风格中一个约束就是超媒体作为应用状态的引擎，所以在通过GET请求查询资源列表时不仅要返回该资源组中每个资源的属性信息，还要返回每一个资源的URI，以方便客户端的状态跳转。

1. 分页查询

元数据引擎中需要定义一个用于分页查询的POJO，在客户端查询时将此POJO序列化并编码，然后使用URI将这个参数传递给Server，格式为”/resources/sudents?page={page}”。在后台获取这个参数的方法如下所示：

Form form = getRequest().getResourceRef().getQueryAsForm();

String page = form.getFirstValue("page");

获取到这个参数之后便可对其进行解码并反序列化为POJO。

因为分页查询的参数也是将其序列化并通过URI传递的，所以在POJO中变量也不能使用基本类型，应该使用其对应的封装类。

1. 模糊查询

对于支持模糊查询的接口，要把支持模糊查询的属性规定好。

#### POST方法

POST方法通常表示创建一个资源，也能被用于调用任意过程。在使用POST方法时，我们将参数放在消息体中，而这个参数也是POJO序列化后的JSON字符串。代码如下所示：

Student student = new Student(5);

student.setName("Bob");

student.setClsId(5);

student.setSex(1);

student.setAge(12);

String str = JSON.toJSONString(student);

ClientResource client = new ClientResource("http://localhost:8080/restlet/resources/students");

Representation representation = client.post(new StringRepresentation(str));

在Server端，我们从消息体中获取参数，然后反序列化为POJO即可，代码如下所示：

@Post

public Representation addStudent(Representation entity)

throws ResourceException, IOException{

String str = entity.getText();

Student student = JSON.parseObject(str, Student.class);

return new StringRepresentation(StudentDao.addStudent(student));

}

#### PUT方法

PUT方法通常表示修改一个资源。在使用PUT方法时，也将参数放在消息体中，这个参数也是POJO序列化后的JSON字符串。如果是要修改指定ID的资源，则可访问单一资源的Resource；如果是要修改满足特定条件的资源，则访问一组资源的Resource。

PUT方法传参数的方式与POST方法类似，也是通过消息体传输。唯一不同的是POST方法访问的是一组资源的Resource，其URI为”/resources/sudents”，而PUT方法访问的是单一资源的Resource或一组资源的Resource。当修改指定ID的对象时便访问单一资源的Resource，URI为”/resources/sudent/{id}”。当基于其他条件修改对象时便访问一组资源的Resource，URI为”/resources/sudents?filterParam={filterParam}”。对于后者，URI中的参数表示修改资源的过滤条件，而消息体中传输的参数是资源的目标状态。

#### DELETTE方法

DELETE方法通常表示删除一个资源。DELETE方法不能使用消息体传递参数，而是像GET方法一样，使用URI传递参数。如果是要删除指定ID的资源，则可访问单一资源的Resource，URI为”/resources/sudent/{id}”。如果是要删除满足特定条件的资源，则访问一组资源的Resource，URI为”/resources/sudents?filterParam={filterParam}”。

#### 小结

Restlet标准接口的典型用途如表6-1所示：

|  |  |  |
| --- | --- | --- |
| 接口 | 单一资源的Resource | 一组资源的Resource |
| GET | 获取指定资源的详细信息 | 查询资源列表，包括该资源组中每个资源的URI与详细信息 |
| POST | 把指定资源当做一个资源组，并在其下创建/追加一个新的元素，使其隶属于当前资源 | 在本组资源中创建/追加一个新的资源。该操作往往返回新资源的URI |
| PUT | 替换/创建指定资源，并将其追加到相应的资源组中 | 使用给定的一组资源替换当前整组资源 |
| DELETE | 删除指定的元素 | 删除整组资源 |

表6-1. Restlet标准接口的典型用途