**Web 服务编程，REST 与 SOAP**

**为什么选择 REST**

REST 架构风格是一种全新的针对 Web 应用的开发风格，与 RPC 风格的 Web 服务（关于 RPC 风格的 Web 服务描述，请参见 [Wikipeida](http://en.wikipedia.org/wiki/Web_service) ）一样，是企业信息化的一个重要架构实践领域。本文从一个简单的应用场景出发，使用 REST 和 SOAP 两种不同的架构风格实现，通过对 REST 与 SOAP Web 服务具体对比,旨在帮助读者更深刻理解 REST 架构风格。

4http://dw1.s81c.com/developerworks/i/v17/dw-cmts-arrow.png [评论](http://www.ibm.com/developerworks/cn/webservices/0907_rest_soap/#icomments)

[李 三红](http://www.ibm.com/developerworks/cn/webservices/0907_rest_soap/#authorN10038), 高级软件工程师, IBM

2009 年 7 月 28 日

* [expand](http://www.ibm.com/developerworks/cn/webservices/0907_rest_soap/#toggle)**内容**



在 IBM Bluemix 云平台上开发并部署您的下一个应用。

[**开始您的试用**](https://developer.ibm.com/sso/bmregistration?lang=zh_CN&ca=dwchina-_-bluemix-_-0907_rest_soap-_-sidebar)

**REST 简介**

在开始我们的正式讨论之前，让我们简单看一下 REST 的定义。

REST（Representational State Transfer）是 Roy Fielding 提出的一个描述互联系统架构风格的名词。为什么称为 REST？Web 本质上由各种各样的资源组成，资源由 URI 唯一标识。浏览器（或者任何其它类似于浏览器的应用程序）将展示出该资源的一种表现方式，或者一种表现状态。如果用户在该页面中定向到指向其它资源的链接，则将访问该资源，并表现出它的状态。这意味着客户端应用程序随着每个资源表现状态的不同而发生状态转移，也即所谓 REST。

关于 REST 本身，本文就不再这里过多地讨论，读者可以参考 developerWorks 上其它介绍 REST 的文章。本文的重点在于通过 REST 与 SOAP Web 服务的对比，帮助读者更深刻理解 REST 架构风格的特点，优势。

[**回页首**](http://www.ibm.com/developerworks/cn/webservices/0907_rest_soap/#ibm-pcon)

**应用场景介绍（在线用户管理）**

本文将借助于一个应用场景，通过基于 REST 和 SOAP Web 服务的不同实现，来对两者进行对比。该应用场景的业务逻辑会尽量保持简单且易于理解，以有助于把我们的重心放在 REST 和 SOAP Web 服务技术特质对比上。

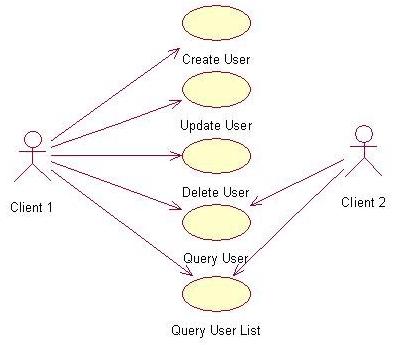
**需求描述**

这是一个在线的用户管理模块，负责用户信息的创建，修改，删除，查询。用户的信息主要包括：

* 用户名（唯一标志在系统中的用户）
* 头衔
* 公司
* EMAIL
* 描述

需求用例图如下：

**图 1. 需求用例图**



如图 1 所示，客户端 1（Client1）与客户端 2（Client2）对于信息的存取具有不同的权限，客户端 1 可以执行所有的操作，而客户端 2 只被允许执行用户查询（Query User）与用户列表查询（Query User List）。关于这一点，我们在对 REST Web 服务与 SOAP Web 服务安全控制对比时会具体谈到。下面我们将分别向您介绍如何使用 REST 和 SOAP 架构实现 Web 服务。

[**回页首**](http://www.ibm.com/developerworks/cn/webservices/0907_rest_soap/#ibm-pcon)

**使用 REST 实现 Web 服务**

本部分将基于 Restlet 框架来实现该应用。Restlet 为那些要采用 REST 结构体系来构建应用程序的 Java 开发者提供了一个具体的解决方案。关于更多的 Restlet 相关内容，本文不做深入讨论，请见参考资源列表。

**设计**

我们将采用遵循 REST 设计原则的 ROA（Resource-Oriented Architecture，面向资源的体系架构）进行设计。ROA 是什么？简单点说，ROA 是一种把实际问题转换成 REST 式 Web 服务的方法，它使得 URI、HTTP 和 XML 具有跟其他 Web 应用一样的工作方式。

在使用 ROA 进行设计时，我们需要把真实的应用需求转化成 ROA 中的资源，基本上遵循以下的步骤：

* 分析应用需求中的数据集。
* 映射数据集到 ROA 中的资源。
* 对于每一资源，命名它的 URI。
* 为每一资源设计其 Representations。
* 用 hypermedia links 表述资源间的联系。

接下来我们按照以上的步骤来设计本文的应用案例。

在线用户管理所涉及的数据集就是用户信息，如果映射到 ROA 资源，主要包括两类资源：用户及用户列表。用户资源的 URI 用http://localhost:8182/v1/users/{username} 表示，用户列表资源的 URI 用 http://localhost:8182/v1/users 表示。它们的 Representation 如下，它们都采用了如清单 1 和清单 2 所示的 XML 表述方式。

**清单 1. 用户列表资源 Representation**

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<users>

<user>

<name>tester</name>

<link>http://localhost:8182/v1/users/tester</link>

</user>

<user>

<name>tester1</name>

<link>http://localhost:8182/v1/users/tester1</link>

</user>

</users>

**清单 2. 用户资源 Representation**

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<user>

<name>tester</name>

<title>software engineer</title>

<company>IBM</company>

<email>tester@cn.ibm.com</email>

<description>testing!</description>

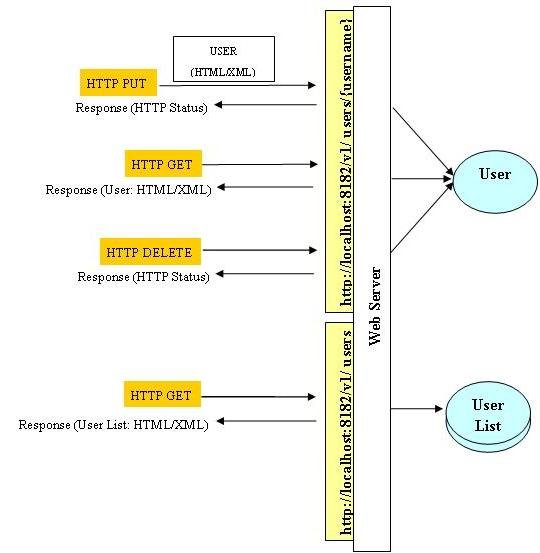
</user>

客户端通过 User List Resource 提供的 LINK 信息 ( 如 :**<link>http://localhost:8182/v1/users/tester</link>**) 获得具体的某个 USER Resource。

**Restful Web 服务架构**

首先给出 Web 服务使用 REST 风格实现的整体架构图，如下图所示：

**图 2. REST 实现架构**



接下来，我们将基于该架构，使用 Restlet 给出应用的 RESTful Web 服务实现。

下面的章节中，我们将给出 REST Web 服务实现的核心代码片段。关于完整的代码清单，读者可以通过资源列表下载。

**客户端实现**

清单 3 给出的是客户端的核心实现部分，其主要由四部分组成：使用 HTTP PUT 增加、修改用户资源，使用 HTTP GET 得到某一具体用户资源，使用 HTTP DELETE 删除用户资源，使用 HTTP GET 得到用户列表资源。而这四部分也正对应了图 2 关于架构描述的四对 HTTP 消息来回。关于 UserRestHelper 类的完整实现，请读者参见本文所附的代码示例。

**清单 3. 客户端实现**

public class UserRestHelper {

//The root URI of our ROA implementation.

public static final tring *APPLICATION\_URI* = "http://localhost:8182/v1";

//Get the URI of user resource by user name.

private static String getUserUri(String name) {

return *APPLICATION\_URI* + "/users/" + name;

}

//Get the URI of user list resource.

private static String getUsersUri() {

return *APPLICATION\_URI* + "/users";

}

//Delete user resource from server by user name.

//使用 HTTP DELETE 方法经由 URI 删除用户资源

public static void deleteFromServer(String name) {

Response response = new Client(Protocol.*HTTP*).delete(*getUserUri*(name));

……

}

//Put user resource to server.

//使用 HTTP PUT 方法经由 URI 增加或者修改用户资源

public static void putToServer(User user) {

//Fill FORM using user data.

Form form = new Form();

form.add("user[title]", user.getTitle());

form.add("user[company]", user.getCompany());

form.add("user[email]", user.getEmail());

form.add("user[description]", user.getDescription());

Response putResponse = new Client(Protocol.*HTTP*).put(

*getUserUri*(user.getName()), form.getWebRepresentation());

……

}

//Output user resource to console.

public static void printUser(String name) {

*printUserByURI*(*getUserUri*(name));

}

//Output user list resource to console.

//使用 HTTP GET 方法经由 URI 显示用户列表资源

public static void printUserList() {

Response getResponse = new Client(Protocol.*HTTP*).get(*getUsersUri*());

if (getResponse.getStatus().isSuccess()) {

DomRepresentation result = getResponse.getEntityAsDom();

//The following code line will explore this XML document and output

//each user resource to console.

……

} else {

System.*out*.println("Unexpected status:"+ getResponse.getStatus());

}

}

//Output user resource to console.

//使用 HTTP GET 方法经由 URI 显示用户资源

private static void printUserByURI(String uri) {

Response getResponse = new Client(Protocol.*HTTP*).get(uri);

if (getResponse.getStatus().isSuccess()) {

DomRepresentation result = getResponse.getEntityAsDom();

//The following code line will explore this XML document and output

//current user resource to console.

……

} else {

System.*out*.println("unexpected status:"+ getResponse.getStatus());

}

}

}

**服务器端实现**

清单 4 给出的是服务器端对于用户资源类（UserResourc）的实现，其核心的功能是响应有关用户资源的 HTTP GET/PUT/DELETE 请求，而这些请求响应逻辑正对应了 UserRestHelper 类中关于用户资源类的 HTTP 请求。

**清单 4. 服务器端实现**

public class UserResource extends Resource {

private User \_user;

private String \_userName;

public UserResource(Context context, Request request, Response response) {

//Constructor is here.

……

}

//响应 HTTP DELETE 请求逻辑

public void delete() {

// Remove the user from container.

getContainer().remove(\_userName);

getResponse().setStatus(Status.SUCCESS\_OK);

}

//This method will be called by handleGet.

public Representation getRepresentation(Variant variant) {

Representation result = null;

if (variant.getMediaType().equals(MediaType.TEXT\_XML)) {

Document doc = createDocument(this.\_user);

result = new DomRepresentation(MediaType.TEXT\_XML, doc);

}

return result;

}

//响应 HTTP PUT 请求逻辑。

public void put(Representation entity) {

if (getUser() == null) {

//The user doesn't exist, create it

setUser(new User());

getUser().setName(this.\_userName);

getResponse().setStatus(Status.SUCCESS\_CREATED);

} else {

getResponse().setStatus(Status.SUCCESS\_NO\_CONTENT);

}

//Parse the entity as a Web form.

Form form = new Form(entity);

getUser().setTitle(form.getFirstValue("user[title]"));

getUser().setCompany(form.getFirstValue("user[company]"));

getUser().setEmail(form.getFirstValue("user[email]"));

getUser().setDescription(form.getFirstValue("user[description]"));

//Put the user to the container.

getApplication().getContainer().put(\_userName, getUser());

}

//响应 HTTP GET 请求逻辑。

public void handleGet() {

super.handleGet();

if(this.\_user != null ) {

getResponse().setEntity(getRepresentation(

new Variant(MediaType.TEXT\_XML)));

getResponse().setStatus(Status.SUCCESS\_OK);

} else {

getResponse().setStatus(Status.CLIENT\_ERROR\_NOT\_FOUND);

}

}

//build XML document for user resource.

private Document createDocument(User user) {

//The following code line will create XML document according to user info.

……

}

//The remaining methods here

……

}

UserResource 类是对用户资源类的抽象，包括了对该资源的创建修改（put 方法），读取（handleGet 方法 ）和删除（delete 方法），被创建出来的 UserResource 类实例被 Restlet 框架所托管，所有操纵资源的方法会在相应的 HTTP 请求到达后被自动回调。

另外，在服务端，还需要实现代表用户列表资源的资源类 UserListResource，它的实现与 UserResource 类似，响应 HTTP GET 请求，读取当前系统内的所有用户信息，形成如清单 1 所示的用户列表资源 Representation，然后返回该结果给客户端。具体的实现请读者参见本文所附的代码示例。

[**回页首**](http://www.ibm.com/developerworks/cn/webservices/0907_rest_soap/#ibm-pcon)

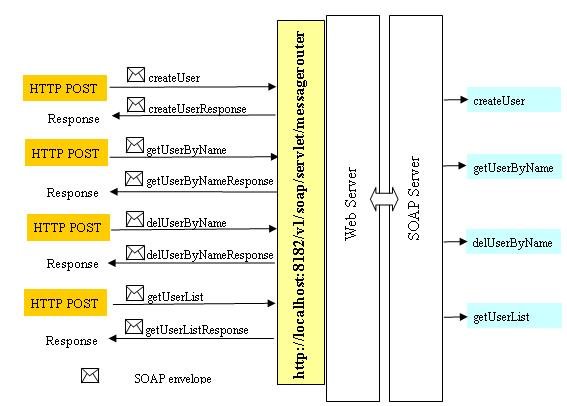
**使用 SOAP 实现 Web 服务**

本文对于 SOAP 实现，就不再像 REST 那样，具体到代码级别的实现。本节将主要通过 URI,HTTP 和 XML 来宏观上表述 SOAP Web 服务实现的技术本质，为下一节 REST Web 服务与 SOAP Web 服务的对比做铺垫。

**SOAP Web 服务架构**

同样，首先给出 SOAP 实现的整体架构图，如下图所示：

**图 3. SOAP 实现架构**



可以看到，与 REST 架构相比，SOAP 架构图明显不同的是：所有的 SOAP 消息发送都使用 HTTP POST 方法，并且所有 SOAP 消息的 URI 都是一样的，这是基于 SOAP 的 Web 服务的基本实践特征。

**获得用户信息列表**

基于 SOAP 的客户端创建如清单 5 所示的 SOAP XML 文档，它通过类 RPC 方式来获得用户列表信息。

**清单 5. getUserList SOAP 消息**

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

<soap:Body>

<p:getUserList xmlns:p="http://www.exmaple.com"/>

</soap:Body>

</soap:Envelope>

客户端将使用 HTTP 的 POST 方法，将上述的 SOAP 消息发送至 http://localhost:8182/v1/soap/servlet/messagerouter URI，SOAP SERVER 收到该 HTTP POST 请求，通过解码 SOAP 消息确定需要调用 getUserList 方法完成该 WEB 服务调用，返回如下的响应：

**清单 6. getUserListResponse 消息**

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

<soap:Body>

<p:get

UserListResponse xmlns:p="http://www.exmaple.com">

<Users>

<username>tester<username>

<username>tester1<username>

......

</Users>

<p: getUserListResponse >

</soap:Body>

</soap:Envelope>

**获得某一具体用户信息**

**清单 7. getUserByName SOAP 消息**

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

<soap:Body>

<p:getUserByName xmlns:p="http://www.exmaple.com">

<username>tester</username>

</p:getUserByName >

</soap:Body>

</soap:Envelope>

同样地，客户端将使用 HTTP 的 POST 方法，将上述的 SOAP 消息发送至 http://localhost:8182/v1/soap/servlet/messagerouterURI，SOAP SERVER 处理后返回的 Response 如下：

**清单 8. getUserByNameResponse SOAP 消息**

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

<soap:Body>

<p:getUserByNameResponse xmlns:p="http://www.exmaple.com">

<name>tester</name>

<title>software engineer</title>

<company>IBM</company>

<email>tester@cn.ibm.com</email>

<description>testing!</description>

</p:getUserByNameResponse>

</soap:Body>

</soap:Envelope>

实际上，创建新的用户，过程也比较类似，在这里，就不一一列出，因为这两个例子对于本文在选定的点上对比 REST 与 SOAP 已经足够了。

[**回页首**](http://www.ibm.com/developerworks/cn/webservices/0907_rest_soap/#ibm-pcon)

**REST 与 SOAP 比较**

本节从以下几个方面来对比上面两节给出 REST 实现与 SOAP 实现。

**接口抽象**

RESTful Web 服务使用标准的 HTTP 方法 (GET/PUT/POST/DELETE) 来抽象所有 Web 系统的服务能力，而不同的是，SOAP 应用都通过定义自己个性化的接口方法来抽象 Web 服务，这更像我们经常谈到的 RPC。例如本例中的 getUserList 与 getUserByName 方法。

RESTful Web 服务使用标准的 HTTP 方法优势，从大的方面来讲：标准化的 HTTP 操作方法，结合其他的标准化技术，如 URI，HTML，XML 等，将会极大提高系统与系统之间整合的互操作能力。尤其在 Web 应用领域，RESTful Web 服务所表达的这种抽象能力更加贴近 Web 本身的工作方式，也更加自然。

同时，使用标准 HTTP 方法实现的 RRESTful Web 服务也带来了 HTTP 方法本身的一些优势：

* ***无状态性（Stateless）***

HTTP 协议从本质上说是一种无状态的协议，客户端发出的 HTTP 请求之间可以相互隔离，不存在相互的状态依赖。基于 HTTP 的 ROA，以非常自然的方式来实现无状态服务请求处理逻辑。对于分布式的应用而言，任意给定的两个服务请求 Request 1 与 Request 2, 由于它们之间并没有相互之间的状态依赖，就不需要对它们进行相互协作处理，其结果是：Request 1 与 Request 2 可以在任何的服务器上执行，这样的应用很容易在服务器端支持负载平衡 (load-balance)。

* ***安全操作与幂指相等特性（Safety /Idempotence）***

HTTP 的 GET、HEAD 请求本质上应该是安全的调用，即：GET、HEAD 调用不会有任何的副作用，不会造成服务器端状态的改变。对于服务器来说，客户端对某一 URI 做 n 次的 GET、HAED 调用，其状态与没有做调用是一样的，不会发生任何的改变。

HTTP 的 PUT、DELTE 调用，具有幂指相等特性 , 即：客户端对某一 URI 做 n 次的 PUT、DELTE 调用，其效果与做一次的调用是一样的。HTTP 的 GET、HEAD 方法也具有幂指相等特性。

HTTP 这些标准方法在原则上保证你的分布式系统具有这些特性，以帮助构建更加健壮的分布式系统。

**安全控制**

为了说明问题，基于上面的在线用户管理系统，我们给定以下场景：

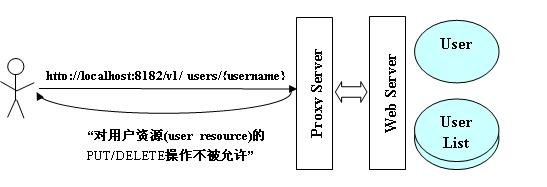
参考一开始我们给出的用例图，对于客户端 Client2，我们只希望它能以只读的方式访问 User 和 User List 资源，而 Client1 具有访问所有资源的所有权限。

如何做这样的安全控制？

通行的做法是：所有从客户端 Client2 发出的 HTTP 请求都经过代理服务器 (Proxy Server)。代理服务器制定安全策略：所有经过该代理的访问 User 和 User List 资源的请求只具有读取权限，即：允许 GET/HEAD 操作，而像具有写权限的 PUT/DELTE 是不被允许的。

如果对于 REST，我们看看这样的安全策略是如何部署的。如下图所示：

**图 4. REST 与代理服务器 (Proxy Servers)**

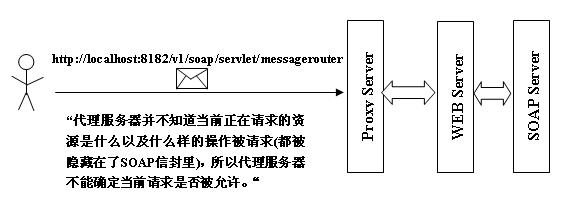


一般代理服务器的实现根据 (URI, HTTP Method) 两元组来决定 HTTP 请求的安全合法性。

当发现类似于（http://localhost:8182/v1/users/{username}，DELETE）这样的请求时，予以拒绝。

对于 SOAP，如果我们想借助于既有的代理服务器进行安全控制，会比较尴尬，如下图：

**图 5. SOAP 与代理服务器 (Proxy Servers)**



所有的 SOAP 消息经过代理服务器，只能看到（http://localhost:8182/v1/soap/servlet/messagerouter, HTTP POST）这样的信息，如果代理服务器想知道当前的 HTTP 请求具体做的是什么，必须对 SOAP 的消息体解码，这样的话，意味着要求第三方的代理服务器需要理解当前的 SOAP 消息语义，而这种 SOAP 应用与代理服务器之间的紧耦合关系是不合理的。

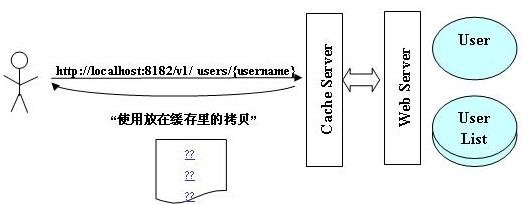
**关于缓存**

众所周知，对于基于网络的分布式应用，网络传输是一个影响应用性能的重要因素。如何使用缓存来节省网络传输带来的开销，这是每一个构建分布式网络应用的开发人员必须考虑的问题。

HTTP 协议带条件的 HTTP GET 请求 (Conditional GET) 被设计用来节省客户端与服务器之间网络传输带来的开销，这也给客户端实现 Cache 机制 ( 包括在客户端与服务器之间的任何代理 ) 提供了可能。HTTP 协议通过 HTTP HEADER 域：If-Modified-Since/Last- Modified，If-None-Match/ETag 实现带条件的 GET 请求。

REST 的应用可以充分地挖掘 HTTP 协议对缓存支持的能力。当客户端第一次发送 HTTP GET 请求给服务器获得内容后，该内容可能被缓存服务器 (Cache Server) 缓存。当下一次客户端请求同样的资源时，缓存可以直接给出响应，而不需要请求远程的服务器获得。而这一切对客户端来说都是透明的。

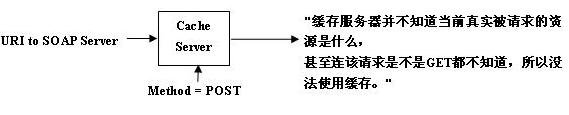
**图 6. REST 与缓存服务器 (Cache Server)**



而对于 SOAP，情况又是怎样的呢？

使用 HTTP 协议的 SOAP，由于其设计原则上并不像 REST 那样强调与 Web 的工作方式相一致，所以，基于 SOAP 应用很难充分发挥 HTTP 本身的缓存能力。

**图 7. SOAP 与缓存服务器 (Cache Server)**



两个因素决定了基于 SOAP 应用的缓存机制要远比 REST 复杂：

其一、所有经过缓存服务器的 SOAP 消息总是 HTTP POST，缓存服务器如果不解码 SOAP 消息体，没法知道该 HTTP 请求是否是想从服务器获得数据。

其二、SOAP 消息所使用的 URI 总是指向 SOAP 的服务器，如本文例子中的 http://localhost:8182/v1/soap/servlet/messagerouter，这并没有表达真实的资源 URI，其结果是缓存服务器根本不知道那个资源正在被请求，更不用谈进行缓存处理。

**关于连接性**

在一个纯的 SOAP 应用中，URI 本质上除了用来指示 SOAP 服务器外，本身没有任何意义。与 REST 的不同的是，无法通过 URI 驱动 SOAP 方法调用。例如在我们的例子中，当我们通过

getUserList SOAP 消息获得所有的用户列表后，仍然无法通过既有的信息得到某个具体的用户信息。唯一的方法只有通过 WSDL 的指示，通过调用 getUserByName 获得，getUserList 与 getUserByName 是彼此孤立的。

而对于 REST，情况是完全不同的：通过 http://localhost:8182/v1/users URI 获得用户列表，然后再通过用户列表中所提供的 LINK 属性，例如 <link>http://localhost:8182/v1/users/tester</link>获得 tester 用户的用户信息。这样的工作方式，非常类似于你在浏览器的某个页面上点击某个 hyperlink, 浏览器帮你自动定向到你想访问的页面，并不依赖任何第三方的信息。

[**回页首**](http://www.ibm.com/developerworks/cn/webservices/0907_rest_soap/#ibm-pcon)

**总结**

典型的基于 SOAP 的 Web 服务以操作为中心，每个操作接受 XML 文档作为输入，提供 XML 文档作为输出。在本质上讲，它们是 RPC 风格的。而在遵循 REST 原则的 ROA 应用中，服务是以资源为中心的，对每个资源的操作都是标准化的 HTTP 方法。

本文主要集中在以上的几个方面，对 SOAP 与 REST 进行了对比，可以看到，基于 REST 构建的系统其系统的扩展能力要强于 SOAP，这可以体现在它的统一接口抽象、代理服务器支持、缓存服务器支持等诸多方面。并且，伴随着 Web Site as Web Services 演进的趋势，基于 REST 设计和实现的简单性和强扩展性，有理由相信，REST 将会成为 Web 服务的一个重要架构实践领域。