《[我所理解的RESTful Web API [Web标准篇]](http://www.cnblogs.com/artech/p/restful-web-api-01.html)》Web服务已经成为了异质系统之间的互联与集成的主要手段，在过去一段不短的时间里，Web服务几乎清一水地采用SOAP来构建。构建REST风格的Web服务是最近两三年风行的潮流，所以很多人以为REST是一个事物。而事实却是：REST自其诞生之日起到现在（2014年）已经有14年了，它为什么叫这么一个“奇怪”的名字呢？

目录   
一、为什么叫这个“奇怪”的名字？二、采用URI标识资源   
二、采用URI标识资源   
三、使用“链接”关联相关的资源   
四、使用统一的接口   
五、使用标准的HTTP方法   
六、支持多种资源表示方式   
七、无状态性

# **一、为什么叫这个“奇怪”的名字？**

2000年，Roy Thomas Fielding博士在他那篇著名的博士论文《Architectural Styles and the Design of Network-based Software Architectures》中提出了几种软件应用的架构风格，REST作为其中的一种架构风格在这篇论文的第5章中进行了概括性的介绍。我个人建议本书的读者都能读读这篇论文，原文和中文译文都可以从网络上找到。

REST是“REpresentational State Transfer”的缩写，可以翻译成“表现状态转换”，但是在绝大多数场合中我们只说REST或者RESTful。为什么会起这么一个奇怪的名字呢？我们可以从上述这篇论文中找到答案。Fielding在论文中将REST定位为“分布式超媒体应用（Distributed Hypermedia System）”的架构风格，它在文中提到一个名为“HATEOAS（Hypermedia as the engine of application state）”的概念。

我们利用一个面向最终用户的Web应用来对这个概念进行简单阐述：这里所谓的应用状态（Application State）表示Web应用的客户端的状态，简单起见可以理解为会话状态。资源在浏览器中以超媒体的形式呈现，通过点击超媒体中的链接可以获取其它相关的资源或者对当前资源进行相应的处理，获取的资源或者针对资源处理的响应同样以超媒体的形式再次呈现在浏览器上。由此可见，超媒体成为了驱动客户端会话状态的转换的引擎。

借助于超媒体这种特殊的资源呈现方式，应用状态的转换体现为浏览器中呈现资源的转换。如果将超媒体进一步抽象成一般意义上的资源呈现（Representation ）方式，那么应用状态变成了可被呈现的状态（REpresentational State）。应用状态之间的转换就成了可被呈现的状态装换（REpresentational State Transfer），这就是REST。

REST在我看来是一种很笼统的概念，它代表一种架构风格。对于多个Web应用采用的架构，我们只能说其中某一个比其它的更具有REST风格，而不能简单粗暴地说：“它采用了REST架构而其它的没有”。为了将REST真正地落地，Lenoard Rechardson & Sam Ruby在《RESTful Web Services》一书中提出了一种名为“面向资源的架构（ROA： Resource Oriented Architecture）”。该书中介绍了一些采用ROA架构的Web服务应该具备的基本特征，它们可以指导我们如果构架具体的RESTful Web API。

# **二、采用URI标识资源**

SOAP Web API采用RPC风格，它采用面向功能的架构，所以我们在设计SOAP Web API的时候首相考虑的是应高提供怎样的功能（或者操作）。RESTful Web API采用面向资源的架构，所以在设计之初首先需要考虑的是有哪些资源可供操作。

资源是一个很宽泛的概念，任何寄宿于Web可供操作的“事物”均可视为资源。资源可以体现为经过持久化处理保存到磁盘上的某个文件或者数据库中某个表的某条记录，也可以是Web应用接受到请求后采用某种算法计算得出的结果。资源可以体现为一个具体的物理对象，它也可以是一个抽象的流程。

一个资源必须具有一个或者多个标识，既然我们设计的Web API，那么很自然地应该采用URI来作为资源的标识。作为资源标识的URI最好具有“可读性”，因为具有可读性的URI更容易被使用，使用者一看就知道被标识的是何种资源，比如如下一些URI就具有很好的可读性。

* http://www.artech.com/employees/c001（编号C001的员工）
* http://www.artech.com/sales/2013/12/31（2013年12月31日的销售额）
* http://www.artech.com/orders/2013/q4（2013年第4季度签订的订单）

除了必要的标志性和可选的可读性之外，标识资源的URI应该具有“可寻址性（Addressability）”。也就是说，URI不仅仅指明了被标识资源所在的位置，而且通过这个URI可以直接获取目标资源。通过前面的介绍 我们知道URI具有URL和URN两种主要的表现形式，只要前者具有可寻址性，所以我们最好采用一个URL作为资源的标识。

URI除了可以标识某个独立的资源外（比如“http://www.artech.com/employees/c001”），还可以标识一组资源的集合或者资源的容器（比如“http://www.artech.com/orders/2013/q4”）。当然，一组同类资源的集合或者存放一组同类资源的容器本身也可以视为另一种类型的复合型（Composite）资源，所以“URI总是标识某个资源”这种说法是没有问题的。

# **三、使用“链接”关联相关的资源**

在绝大多数情况下，资源并不会孤立地存在，必然与其它资源具有某种关联。既然我们推荐资源采用具有可寻址性的URL来标识，那么我们就可以利用它来将相关的资源关联起来。比如我们采用XML来表示一部电影的信息，那么我们采用如下的形式利用URL将相关的资源（导演、领衔主演、主演、编剧以及海报）关联在一起。实际上这可以视为一份超文本/超媒体文档。当用户得到这样一份文档的时候，可以利用自身的内容获得某部影片基本的信息，还可以利用相关的“链接”得到其它相关内容的详细信息。

1: <movie>

2: <name>魔鬼代言人</name>

3: <genre>剧情|悬疑|惊悚</genre>

4: <directors>

5: <add ref="http://www.artech.com/directors/taylor-hackford">泰勒.海克福德</add>

6: </directors>

7: <starring>

8: <add ref = "http://www.artech.com/actors/al-pacino">阿尔.帕西诺</add>

9: <add ref = "http://www.artech.com/actors/keanu-reeves ">基诺.李维斯</add>

10: </starring>

11: <supportingActors>

12: <add ref = "http://www.artech.com/actors/charlize-theron ">查理兹.塞隆</add>

13: <add ref = "http://www.artech.com/actors/jeffrey-jones ">杰弗瑞.琼斯</add>

14: <add ref = "http://www.artech.com/actors/connie-nielsen">康尼.尼尔森</add>

15: </supportingActors>

16: <scriptWriters>

17: <add ref = "http://www.artech.com/scriptwriters/jonathan-lemkin">乔纳森•莱姆金</add>

19: <add ref = "http://www.artech.com/scriptwriters/tony-gilroy">托尼•吉尔罗伊 </add>

20: </scriptWriters>

21: <language>英语</language>

22: <poster ref = "http://www.artech.com/images/the-devil-s-advocate"/>

23: <story>...</story>

24: </movie>

Fielding在他的论文中将REST定位为“分布式超媒体应用”的架构风格，而超媒体的核心就是利用“链接”相关的信息结成一个非线性的网，所以从一点也可以看出REST和“使用链接关联相关的资源”这个特性使吻合的。

# **四、使用统一的接口**

由于REST是面向资源的，所以一个Web API旨在实现针对单一资源的操作。我们在前面已经说个，针对资源的基本操作唯CRUD而已，这是使我们可以为Web API定义标准接口成可能。所谓的标准接口就是针对不同资源的Web API定义一致性的操作来操作它们，其接口可以采用类似于下面的模式。

1: public class ResourceService

2: {

3: public IEnumerable<Resource>[] Get();

4: public void Create(Resource resource);

5: public void Update(Resource resource);

6: public void Delete(string id);

7: }

能否采用统一接口是RESTful Web API和采用RPC风格的SOAP Web服务又一区别。如果采用RPC风格的话，我们在设计Web API的时候首先考虑的是具体哪些功能需要被提供，所以这样的Web API是一组相关功能的集合而已。

以一个具体的场景为例。现在我们需要设计一个Web API来管理用于授权的角色，它只需要提供针对角色本身的CRUD的功能以及建立/解除与用户名之间的映射关系。如果我们将其定义成针对SOAP的Web服务，其服务接口具有类似于如下的结构。

1: public class RoleService

2: {

3: public IEnumerable<string> GetAllRoles();

4: public void CreateRole(string roleName);

5: public void DeleteRole(string roleName);

6:

7: public void AddRolesInUser(string userName, string[] roleNames);

8: public void RemoveRolesFromUser(string userName, string[] roleNames);

9: }

如下我们需要将其定义成一个纯粹的RESTful的Web API，只有前面三个方法在针对角色的CRUD操作范畴之内，但是后面两个方法却可以视为针对“角色委派（Role Assignment）”对象的添加和删除操作。所以这里实际上涉及到了两种资源，即角色和角色委派。为了使Web API具有统一的接口，我们需要定义如下两个Web API。

1: public class RolesService

2: {

3: public IEnumerable<string> Get();

4: public void Create(string roleName);

5: public void Delete(string roleName);

6: }

7:

8: public class RoleAssignmentsService

9: {

10: public void Create(RoleAssignment roleName);

11: public void Delete(RoleAssignment roleName);

12: }

# **五、使用标准的HTTP方法**

由于RESTful Web API采用了同一的接口，所以其成员体现为针对同一资源的操作。对于Web来说，针对资源的操作通过HTTP方法来体现。我们应该将两者统一起来，是Web API分别针对CRUD的操作只能接受具有对应HTTP方法的请求。

我们甚至可以直接使用HTTP方法名作为Web API接口的方法名称，那么这样的Web API接口就具有类似于如下的定义。对于ASP.NET Web API来说，由于它提供了Action方法名称和HTTP方法的自动映射，所以如果我们采用这样的命名规则，就无需再为具体的Action方法设定针对HTTP方法的约束了。

1: public class ResourceService

2: {

3: public IEnumerable<Resource>[] Get();

4: public void Post(Resource resource);

5: public void Put(Resource resource);

6: public void Patch (Resource resource);

7: public void Delete(string id);

8:

9: public void Head(string id);

10: public void Options();

11: }

上面代码片断提供的7个方法涉及到了7个常用的HTTP方法，接下来我们针对资源操作的语义对它们作一个简单的介绍。首先GET、HEAD和OPTIONS这三个HTTP方法旨在发送请求以或者所需的信息。对于GET，相应所有人对它已经非常熟悉了，它用于获取所需的资源，服务器一般讲对应的资源置于响应的主体部分返回给客户端。

HEAD和OPTIONS相对少见。从资源操作的语义来讲，一个针对某个目标资源发送的HEAD请求一般不是为了获取目标资源本身的内容，而是得到描述目标资源的元数据信息。服务器一般讲对应资源的元数据置于响应的报头集合返回给客户端，这样的响应一般不具有主体部分。OPTIONS请求旨在发送一种“探测”请求以确定针对某个目标地址的请求必须具有怎样的约束（比如应该采用怎样的HTTP方法以及自定义的请求报头），然后根据其约束发送真正的请求。比如针对“跨域资源”的预检（Preflight）请求采用的HTTP方法就是OPTIONS。

至于其它4中HTTP方法（POST、PUT、PATCH和DELETE），它们旨在针对目标资源作添加、修改和删除操作。对于DELETE，它的语义很明确，就是删除一个已经存在的资源。我们着重推荐其它三个旨在完成资源的添加和修改的HTTP方法作一个简单的介绍。

通过发送POST和PUT请求均可以添加一个新的资源，但是两者的不同之处在于：对于前者，请求着一般不能确定标识添加资源最终采用的URI，即服务端最终为成功添加的资源指定URI；对于后者，最终标识添加资源的URI是可以由请求者控制的。也正是因为这个原因，如果发送PUT请求，我们一般直接将标识添加资源的URI作为请求的URI；对于POST请求来说，其URI一般是标识添加资源存放容器的URI。

比如我们分别发送PUT和POST请求以添加一个员工，标识员工的URI由其员工ID来决定。如果员工ID由客户端来指定，我们可以发送PUT请求；如果员工ID由服务端生成，我们一般发送POST请求。具体的请求与下面提供的代码片断类似，可以看出它们的URI也是不一样的。

1: PUT http://www.artech.com/employees/300357 HTTP/1.1

2: ...

3:

4: <employee>

5: <id>300357</id>

6: <name>张三</name>

7: <gender>男<gender>

8: <birthdate>1981-08-24</birthdate>

9: <department>3041</department>

10: </employee>

1: POST http://www.artech.com/employees HTTP/1.1

2: ...

3:

4: <employee>

5: <name>张三</name>

6: <gender>男<gender>

7: <birthdate>1981-08-24</birthdate>

8: <department>3041</department>

9: </employee>

POST和PUT请求一般将所加资源的内容置于请求的主体。但是对于PUT请求来说，如果添加资源的内容完全可以由其URI来提供，这样的请求可以不需要主体。比如我们通过请求添加一个用于控制权限的角色，标识添加角色的URI由其角色名称来决定，并且不需要指定除角色名称的其它信息，那么我们只要发送如下一个不含主体的PUT请求即可。

1: PUT http://www.artech.com/roles/admin HTTP/1.1

2:

3: ...

除了进行资源的添加，PUT请求还能用于资源的修改。由于请求包含提交资源的标识（可以放在URI中，也可以置于保存在主体部分的资源内容中），所以服务端能够定位到对应的资源予以修改。对于POST和PUT，也存在一种一刀切的说法：POST用于添加，PUT用于修改。我个人比较认可的是：如果PUT提供的资源不存在，则做添加操作，否则做修改。

对于发送PUT请求以修改某个存在的资源，服务器一般会将提供资源将原有资源整体“覆盖”掉。如果需要进行“局部”修改，我们推荐请求采用PATCH方法，因为从语义上讲“Patch”就是打补丁的意思。

## **安全性与幂等性**

关于HTTP请求采用的这些个方法，具有两个基本的特性，即“安全性”和“幂等性”。对于上述7种HTTP方法，GET、HEAD和OPTIONS均被认为是安全的方法，因为它们旨在实现对数据的获取，并不具有“边界效应（Side Effect[[1]](E:/????????????/ASP.NET Web API????????????/???1??? ASP.NET Web API??????/" \l "_ftn1_9470)）”。至于其它4个HTTP方法，由于它们会导致服务端资源的变化，所以被认为是不安全的方法。

幂等性（Idempotent）是一个数学上的概念，在这里表示发送一次和多次请求引起的边界效应是一致的。在网速不够快的情况下，客户端发送一个请求后不能立即得到响应，由于不能确定是否请求是否被成功提交，所以它有可能会再次发送另一个相同的请求，幂等性决定了第二个请求是否有效。

上述3种安全的HTTP方法（GET、HEAD和OPTIONS）均是幂等方法。由于DELETE和PATCH请求操作的是现有的某个资源，所以它们是幂等方法。对于PUT请求，只有在对应资源不存在的情况下服务器才会进行添加操作，否则只作修改操作，所以它也是幂等方法。至于最后一种POST，由于它总是进行添加操作，如果服务器接收到两次相同的POST操作，将导致两个相同的资源被创建，所以这是一个非幂等的方法。

当我们在设计Web API的时候，应该尽量根据请求HTTP方法的幂等型来决定处理的逻辑。由于PUT是一个幂等方法，所以携带相同资源的PUT请求不应该引起资源的状态变化，如果我们在资源上附加一个自增长的计数器表示被修改的次数，这实际上就破坏了幂等型。

不过就我个人的观点来说，在有的场合下针对幂等型要求可以不需要那么严格。举个例子，我对于我们开发的发部分应用来说，数据表基本上都有一个名为LastUpdatedTime的字段表示记录最后一次被修改的时间，因为这是为了数据安全审核（Auditing）的需要。在这种情况下，如果接收到一个基于数据修改的PUT请求，我们总是会用提交数据去覆盖现有的数据，并将当前服务端时间（客户端时间不可靠）作为字段LastUpdatedTime的值，这实际上也破坏了幂等性。

可能有人说我们可以在真正修改数据之前检查提交的数据是否与现有数据一致，但是在涉及多个表链接的时候这个“预检”操作会带来性能损失，而且针对每个字段的逐一比较也是一个很繁琐的事情，所以我们一般不作这样的预检操作。

# **六、支持多种资源表示方式**

资源和资源的表示（Representaion）是两个不同的概念，资源本身是一个抽象的概念，是看不见摸不着的，而看得见摸得着的是资源的表现。比如一个表示一个财年销售情况的资源，它既可以表示为一个列表、一个表格或者是一个图表。如果采用图表，又可以使用柱状图、K线图和饼图等，这一切都是针对同一个资源的不同表示。

我们说“调用Web API获取资源”，这句话其实是不正确的，因为我们获取的不是资源本身，仅仅是资源的某一种表示而已。对于Web来说，目前具有两种主流的数据结构，XML和JSON，它们也是资源的两种主要的呈现方式。在多语言环境下，还应该考虑描述资源采用的语言。

我们在设计Web API的时候，应该支持不同的资源表示，我们不能假定请求提供的资源一定表示成XML，也不能总是以JSON格式返回获取的资源，正确的做法是：根据请求携带的信息识别提交和希望返回的资源表示。对于请求提交的资源，我们一般利用请求的Content-Type报头携带的媒体类型来判断其采用的表示类型。对于响应资源表示类型的识别，可以采用如下两种方式。

* 让请求URI包含资源表示类型，这种方式使用的最多的是针对多语言的资源，我们一般讲表示语言（也可以包含地区）的代码作为URI的一部分，比如“http://www.artech.com/en/orders/2013”表示将2013年的订单以英文的形式返回。
* 采用“内容协商（Content Negotiation）”根据请求相关报头来判断它所希望的资源表示类型，比如“Accept”和“Accept-language”报头可以体现请求可以接受的响应媒体类型和语言。

对于上述两种资源表示识别机制，我们很多人会喜欢后者，因为第一种不够“智能”。实际上前者具有一个后者不具有的特性：“浏览器兼容型”[[2]](E:/????????????/ASP.NET Web API????????????/???1??? ASP.NET Web API??????/" \l "_ftn2_9470)。对于Web API开发来说，浏览器应该成为一种最为常用的测试工具。在不借助任何插件的情况下，我们利用浏览器访问我们在地址栏中输入的URI时对生成的请求内容不能作任何干预的，如果与资源表示相关的信息（比如语言、媒体类型）被直接包含到请求的URI中，那么所有的情况都可以利用浏览器直接测试。

有人从另一方面对“URI携带资源表示类型”作了这样的质疑：由于URI是资源的标识，那么这导致了相同的资源具有多个标识。其实这是没有问题的，URI是资源的唯一标识，但不是其“唯一的唯一标识“，相同的资源可以具有多个标识。

# **七、无状态性**

RESTful只要维护资源的状态，而不需要维护客户端的状态。对于它来说，每次请求都是全新的，它只需要针对本次请求作相应的操作，不需要将本次请求的相关信息记录下来以便用于后续来自相同客户端请求的处理。

对于上面我们介绍的RESTful的这些个特性，它们都是要求我们为了满足这些特征做点什么，唯有这个无状态却是要求我们不要做什么，因为HTTP本身就是无状态的。举个例子，一个网页通过调用Web API分页获取符合查询条件的记录。一般情况下，页面导航均具有“上一页”和“下一页”链接用于呈现当前页的前一页和后一页的记录。那么现在有两种实现方式返回上下页的记录。

* Web API不仅仅会定义根据具体页码的数据查询定义相关的操作，还会针对“上一页”和“下一页”这样的请求定义单独的操作。它自身会根据客户端的Session ID对每次数据返回的页面在本地进行保存，以便能够知道上一页和下一页具体是哪一页。
* Web API只会定义根据具体页码的数据查询定义相关的操作，当前返回数据的页码由客户端来维护。

第一种貌似很“智能”，其实就是一种画蛇添足的作法，因为它破坏了Web API的无状态性。设计无状态的Web API不仅仅使Web API自身显得简单而精炼，还因减除了针对客户端的“亲和度（Affinty）”使我们可以有效地实施负载均衡，因为只有这样集群中的每一台服务器对于每个客户端才是等效的。

[[1]](E:/????????????/ASP.NET Web API????????????/???1??? ASP.NET Web API??????/" \l "_ftnref1_9470) 大部分计算机书籍都将Side Effect翻译成“副作用”，而我们一般将“副（负）作用”理解为负面的作用，其实计算机领域Side Effect表示的作用无所谓正负，所以我们觉得还是还原其字面的含义“边界效用”。除此之外，对于GET、HEAD和OPTIONS请求来说，如果服务端需要对它们作日志、缓存甚至计数操作，严格来说这也算是一种Side Effect，但是请求的发送者不对此负责。

[[2]](E:/????????????/ASP.NET Web API????????????/???1??? ASP.NET Web API??????/" \l "_ftnref2_9470) 这里的“兼容”不是指支持由浏览器发送的请求，因为通过执行JavaScript脚本可以让作为宿主的浏览器发送任何我们希望的请求，这里的兼容体现在尽可能地支持浏览器访问我们在地址栏中输入的URI默认发送的HTTP-GET请求。

参考资料：   
[1] 《HTTP： The Definitive Guide》, By By David Gourley, Brian Totty, Marjorie Sayer, Anshu Aggarwal, Sailu Reddy   
[2] 《RESTful Web Services》, RESTful Web Services   
[3] 《A Brief Introduction to REST》，<http://www.infoq.com/articles/rest-introduction>   
[4] 《TCP/IP Illustrated (Volumn 1: The Protocol)》, by W. Richard Stevens