参考：http://www.ruanyifeng.com/blog/2014/05/restful\_api.html

RESTful API 设计指南

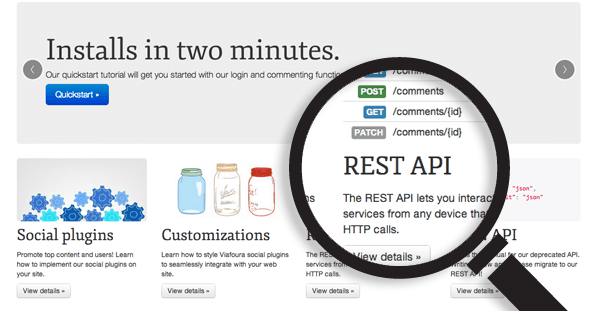
作者： [阮一峰](http://www.ruanyifeng.com/)

日期： [2014年5月22日](http://www.ruanyifeng.com/blog/2014/05/)

网络应用程序，分为前端和后端两个部分。当前的发展趋势，就是前端设备层出不穷（手机、平板、桌面电脑、其他专用设备......）。

因此，必须有一种统一的机制，方便不同的前端设备与后端进行通信。这导致API构架的流行，甚至出现["API First"](http://www.google.com.hk/search?q=API+first)的设计思想。[RESTful API](http://en.wikipedia.org/wiki/Representational_state_transfer)是目前比较成熟的一套互联网应用程序的API设计理论。我以前写过一篇[《理解RESTful架构》](http://www.ruanyifeng.com/blog/2011/09/restful.html)，探讨如何理解这个概念。

今天，我将介绍RESTful API的设计细节，探讨如何设计一套合理、好用的API。我的主要参考了两篇文章（[1](http://codeplanet.io/principles-good-restful-api-design/)，[2](https://bourgeois.me/rest/)）。



一、协议

API与用户的通信协议，总是使用[HTTPs协议](http://www.ruanyifeng.com/blog/2014/02/ssl_tls.html)。

二、域名

应该尽量将API部署在专用域名(me：即子域名)之下。

https://api.example.com

如果确定API很简单，不会有进一步扩展，可以考虑放在主域名下。

https://example.org/api/

三、版本（Versioning）

应该将API的版本号放入URL。

https://api.example.com/v1/

另一种做法是，将版本号放在HTTP头信息中，但不如放入URL方便和直观。[Github](https://developer.github.com/v3/media/#request-specific-version)采用这种做法。

四、路径（Endpoint）

路径又称"终点"（endpoint），表示API的具体网址。

在RESTful架构中，每个网址代表一种资源（resource），所以网址中不能有动词，只能有名词，而且所用的名词往往与数据库的表格名对应。一般来说，数据库中的表都是同种记录的"集合"（collection），所以API中的名词也应该使用复数。

举例来说，有一个API提供动物园（zoo）的信息，还包括各种动物和雇员的信息，则它的路径应该设计成下面这样。

* https://api.example.com/v1/zoos
* https://api.example.com/v1/animals
* https://api.example.com/v1/employees

五、HTTP动词

对于资源的具体操作类型，由HTTP动词表示。

常用的HTTP动词有下面五个（括号里是对应的SQL命令）。

* GET（SELECT）：从服务器取出资源（一项或多项）。
* POST（CREATE）：在服务器新建一个资源。
* PUT（UPDATE）：在服务器更新资源（客户端提供改变后的完整资源-完全更新）。
* PATCH（UPDATE）：在服务器更新资源（客户端提供改变的属性-部分更新）。
* DELETE（DELETE）：从服务器删除资源。

还有两个不常用的HTTP动词。

* HEAD：获取资源的元数据。
* OPTIONS：获取信息，关于资源的哪些属性是客户端可以改变的。

下面是一些例子。

* GET /zoos：列出所有动物园
* POST /zoos：新建一个动物园
* GET /zoos/ID：获取某个指定动物园的信息
* PUT /zoos/ID：更新某个指定动物园的信息（提供该动物园的全部信息）
* PATCH /zoos/ID：更新某个指定动物园的信息（提供该动物园的部分信息）
* DELETE /zoos/ID：删除某个动物园
* GET /zoos/ID/animals：列出某个指定动物园的所有动物（me-集合下资源列表）
* DELETE /zoos/ID/animals/ID：删除某个指定动物园的指定动物（me-集合下的某个资源）

六、过滤信息（Filtering）

如果记录数量很多，服务器不可能都将它们返回给用户。API应该提供参数，过滤返回结果。

下面是一些常见的参数。

* ?limit=10：指定返回记录的数量
* ?offset=10：指定返回记录的开始位置。
* ?page=2&per\_page=100：指定第几页，以及每页的记录数。
* ?sortby=name&order=asc：指定返回结果按照哪个属性排序，以及排序顺序。
* ?animal\_type\_id=1：指定筛选条件

参数的设计允许存在冗余，即允许API路径和URL参数偶尔有重复。比如，GET /zoo/ID/animals 与 GET /animals?zoo\_id=ID 的含义是相同的。

七、状态码（Status Codes）

服务器向用户返回的状态码和提示信息，常见的有以下一些（方括号中是该状态码对应的HTTP动词）。

* 200 OK - [GET]：服务器成功返回用户请求的数据，该操作是幂等的（Idempotent）。
* 201 CREATED - [POST/PUT/PATCH]：用户新建或修改数据成功。
* 202 Accepted - [\*]：表示一个请求已经进入后台排队（异步任务）
* 204 NO CONTENT - [DELETE]：用户删除数据成功。
* 400 INVALID REQUEST - [POST/PUT/PATCH]：用户发出的请求有错误，服务器没有进行新建或修改数据的操作，该操作是幂等的。
* 401 Unauthorized - [\*]：表示用户没有权限（令牌、用户名、密码错误）。
* 403 Forbidden - [\*] 表示用户得到授权（与401错误相对），但是访问是被禁止的。
* 404 NOT FOUND - [\*]：用户发出的请求针对的是不存在的记录，服务器没有进行操作，该操作是幂等的。
* 406 Not Acceptable - [GET]：用户请求的格式不可得（比如用户请求JSON格式，但是只有XML格式）。
* 410 Gone -[GET]：用户请求的资源被永久删除，且不会再得到的。
* 422 Unprocesable entity - [POST/PUT/PATCH] 当创建一个对象时，发生一个验证错误。
* 500 INTERNAL SERVER ERROR - [\*]：服务器发生错误，用户将无法判断发出的请求是否成功。

状态码的完全列表参见[这里](http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html)。

八、错误处理（Error handling）

如果状态码是4xx，就应该向用户返回出错信息。一般来说，返回的信息中将error作为键名，出错信息作为键值即可。

{

error: "Invalid API key"

}

九、返回结果

针对不同操作，服务器向用户返回的结果应该符合以下规范。

me-api路径最好对应数据库的表名，因此名字一般应该是复数

* GET /collection(s)/ID：返回资源对象的列表（数组）
* GET /collection(s)/ID/resource(s)/ID：返回单个资源对象
* POST /collection(s)：返回新生成的资源对象
* PUT /collection(s)/ID/resource(s)/ID：返回完整的资源对象
* PATCH /collection(s)/ID/resource(s)/ID：返回完整的资源对象
* DELETE /collection(s)/ID/resource(s)/ID：返回一个空文档

十、Hypermedia API

RESTful API最好做到Hypermedia，即返回结果中提供链接，连向其他API方法，使得用户不查文档，也知道下一步应该做什么。

比如，当用户向api.example.com的根目录发出请求，会得到这样一个文档。

{"link": {

"rel": "collection <https://www.example.com/zoos>",

"href": "<https://api.example.com/zoos>",

"title": "List of zoos",

"type": "application/vnd.yourformat+json"

}}

上面代码表示，文档中有一个link属性，用户读取这个属性就知道下一步该调用什么API了。rel表示这个API与当前网址的关系（collection关系，并给出该collection的网址），href表示API的路径，title表示API的标题，type表示返回类型。

Hypermedia API的设计被称为[HATEOAS](http://en.wikipedia.org/wiki/HATEOAS)。Github的API就是这种设计，访问[api.github.com](https://api.github.com/)会得到一个所有可用API的网址列表(me-网站地图：site-map，github是以json响应的形式返回网站所有的api，针对每个api有对应的功能说明，如下面的current\_user\_url为网站地图的某个链接)。

{

"current\_user\_url": "<https://api.github.com/user>",

"authorizations\_url": "<https://api.github.com/authorizations>",

// ...

}

从上面可以看到，如果想获取当前用户的信息，应该去访问[api.github.com/user](https://api.github.com/user)，然后就得到了下面结果（me-如下就是每个api说明和api地址）。

{

"message": "Requires authentication",

"documentation\_url": "<https://developer.github.com/v3>"

}

上面代码表示，服务器给出了提示信息，以及文档的网址。

十一、其他

（1）API的身份认证应该使用[OAuth 2.0](http://www.ruanyifeng.com/blog/2014/05/oauth_2_0.html)框架。

（2）服务器返回的数据格式，应该尽量使用JSON，避免使用XML。

参考: <http://www.cnblogs.com/artech/p/restful-web-api-01.html>

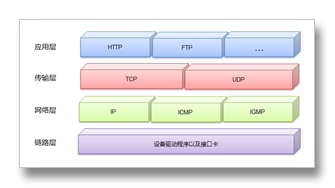
REST不是一个标准，而是一种软件应用架构风格。基于SOAP(简单对象访问协议)的Web服务采用RPC架构，如果说RPC是一种面向操作的架构风格，而REST则是一种面向资源的架构风格（rpc：面向操作的一种架构风格，它是操作的集合，实现如json-rpc；而rest ful为面向资源的架构风格，它是对资源操作的集合-主要crud）。REST是目前业界更为推崇的构建新一代Web服务（或者Web API）的架构风格（其中之一是为了解决不同类型前端请求统一服务端的请求和响应规范）。由于REST仅仅是一种架构风格，所以它是与具体的技术平台无关的，也就是说采用REST架构的应用未必一定建立在Web之上，所以在正式介绍REST之前，我们先来简单认识一下Web。

目录   
一、TCP/IP与HTTP   
二、Web资源   
      媒体类型   
      URI、URL和URN   
三、HTTP事务   
      HTTP方法   
      响应状态码   
四、HTTP报文

如果要问大家这样一个问题：“在过去半个世纪中，哪种信息技术对人类的影响最为深远？”，我想很多人的答案是Web（World Wide Web、WWW、W3或者万维网），因为它改变了我们的生活方式和思维方式。如果各位阅读过W3C介绍WWW的官方文档（“http://www.w3.org/WWW/”），应该对它的第一句话记忆犹新——“The World Wide Web (known as "WWW', "Web" or "W3") is the universe of network-accessible information, the embodiment of human knowledge”。如果将这句话翻译成简洁的中文，就是“Web是（网络）信息的来源，知识的化身”。

Web为我们提供了一种利用HTTP协议获取和操作网络资源的方式，这些将Web服务器作为宿主的资源不仅仅包含像文字和图片这些传统的信息载体，还包含音频和视频这些多媒体信息。Web的核心主要体现在三个方面，即HTTP、超文本（Hypertext）和超媒体（Hypermedia）[[1]](file:///E:\\????????????/ASP.NET%20Web%20API????????????/???1???%20ASP.NET%20Web%20API??????/" \l "_ftn1_5509)，超文本和超媒体规范了网络信息的表现形式，而HTTP则提供了网络访问的标准协议。接下来我们就以围绕着HTTP对Web作一下基本的介绍。

# 一、TCP/IP与HTTP

[](http://images.cnitblog.com/blog/19327/201401/05190925-9f6bcbb34c394a7e80fe30ca85ba0cce.png)

TCP/IP是以IP和TCP协议为核心的一整套网络协议的总称，所以有时候我们也称其为TCP/IP协议簇。毫不夸张地说，TCP/IP支撑着整个互联网，因为它就是互联网采用的网络协议。TCP/IP协议簇划分为如右图所示的4个层次[[2]](file:///E:\\????????????/ASP.NET%20Web%20API????????????/???1???%20ASP.NET%20Web%20API??????/" \l "_ftn2_5509)（应用层、传输层、网络层和链路层），构成整个协议簇的各个子协议处于相应层次中。

既然将整个协议簇命名为TCP/IP，那么IP和TCP自然就是其中最为核心的两个协议了。处于网络层的IP协议提供的IP数据报文传输是不可靠的，因为它只承诺尽可能地将数据报文发送出去，但不能保证发送的数据报文能够成功地抵达目的地。IP协议的不可靠性还体现在它不能检测数据在传输过程中是否发生了改变，也就是说数据的完整性得不到保证。IP协议是一个无连接（Connectionless）的网络协议，每次数据报文的处理对它来说均是独立的，因此IP协议也不能提供针对有序传输（数据接收的顺序与发送的顺序一致）的保证。

虽然IP协议只能提供不可靠的数据传输，同时有序传输也得不到保证，但是建立在它之上的传输层协议TCP有效地解决了这两个问题。TCP是一个基于连接的协议，数据交换双方在进行报文传输之前需要建立连接，报文传输结束之后需要关闭连接。这是一个双工（Duplex）连接，数据交换的双工均可以利用它向对方发送数据。

TCP利用“接收确认”和“超时重传”机制确保了数据能够成功抵达目的地。具体来说，接收方在成功接收到数据之后会回复一个确认消息。发送方在本地具有一个存放尚未得到确认的已发消息的缓冲区，如果发送方在一个设定的时限内没有接收到针对某个已发报文的确认消息，它会从该缓存区中选择对应的报文进行重新发送。在接收到确认之后，相应的报文会从缓存区中移除。

为了解决有序传输的问题，发送方会为每个报文进行编号，报文的序号体现了它们被发送的顺序。接收端在接收到某个报文之后，它会利用此序号判断是否具有尚未成功接收的已发报文，如果有的话，该报文会被存放到本地的缓冲区中。等到之前发送的报文全部被接收之后，接收方按照序号对接收的报文依次向上（应用层）递交，成功递交的报文会被从缓存区中移除。除了接收到“失序”的报文之外，接收方还有可能接收到重复的报文，因为没有报文均具有一个唯一的序号，如果该序号小于已经成功递交或者添加到缓存区中的报文序号，它会被认为是重复接收的报文而被丢弃。

由于每个TCP报文段都具有一个16位的检验和（Checksum），所以接收方可以根据它确认数据在传输过程中是否被篡改。除此之外，TCP还提供了“流量控制”功能避免了双方因缓存区大小不一致而导致报文丢失。具体来说，如果发送方的缓冲区大于接收方的缓存区，会导致接收方在缓冲区已满的情况下无法处理后续接收的报文，所以接收方会将自己缓存区剩余的大小及时通知给发送端，后者据此控制报文发送“流量”。

HTTP（Hypertext Transfer Protocol），全称为“超文本传输协议”，是TCP/IP协议簇的一部分。从图1-1可以看出，这是一个位于应用层的网络协议，在它之下的就是TCP协议。由于TCP协议是一个“可靠”的协议，HTTP自然也能提供可靠数据传输功能。

IP协议利用IP地址来定位数据报发送的目的地，而利用域名系统（DNS）可以实现域名与IP地址之间的转换。TCP协议利用端口号标识应用程序，所以某个应用程序在使用TCP协议进行通信的时候必须指定目标应用的IP地址（或者域名）和端口号。HTTP默认采用的端口号为80，而HTTPS（利用TLS/SSL为HTTP提供传输安全保障）的默认端口号则为443，当然在网络可达的前提下，我们可以指定任意的端口。

# 二、Web资源

这里所说的资源是一个宽泛的概念，任何寄宿于Web服务器可以利用HTTP协议获取或者操作的“事物”均可以称为资源。这也是一个抽象的概念，不仅仅是寄宿于Web服务器的某个静态物理文件可以视为Web资源，通过Web应用根据请求动态生成的数据也是Web资源。

## 媒体类型

资源实际上是一种承载着某种信息的数据，相同的信息可以采用不同形态的数据来展现，数据的“形态”主要体现为展示数据所采用的格式，比如一个数据对象可以通过XML格式来表示，也可以通过JSON格式来表示。数据的处理必须依赖于一种已知的格式，所以将Web资源的形态以一种标准化的方式固定下来显得尤为重要，这就是我们接下来着重介绍的媒体（Media Type）。

不论是通过HTTP请求从Web服务器上获取资源，还是利用请求向服务器提交资源，响应或者请求的主体（Body）除了包含承载资源本身的数据之外，其报头（Header）部分还应该包含表示数据形态的媒体类型。

媒体类型又被称为MIME（Multipurpose Internet Mail Extension）类型，MIME是一个互联网标准，它扩展了电子邮件标准，使其能够支持非ASCII字符、二进制格式附件等多种格式的邮件消息。由于MIME在电子邮件系统应用得非常好，所以被HTTP用于描述并标记多媒体内容。下面的列表给出了一种常用的媒体类型。

* text/html：HTML格式的文档。
* text/xml（application/xml）：XML格式的文本。
* text/json（application/json）： JSON格式的文本。
* image/gif（image/jpeg、image/png）：GIF（JPEG、PNG）格式的图片。
* audio/mp4（audio/mpeg、audio/vnd.wave）：MP4（MPEG、WAVE）格式的音频文件。
* video/mp4（video/mpeg、video/quicktime）：MP4（MPEG、QUICKTIME）格式的视频文件。

## URI、URL和URN

可操作的Web资源应该具有一个 唯一的标识。虽然具有很多唯一性标志符的种类可供选择（比如GUID），但是采用URI来标识Web资源已经成为了一种共识，实际上URI的全称为“统一资源标志符（Uniform Resource Identifier）”。

我想有很多人弄不清楚URI和URL之间的区别，有人甚至觉得这是同一概念的不同表述而已。一个URL肯定是一个URI，但是一个URI并不一定是一个URL，URL仅仅是URI的一种表现形式而已。两者的差异其实可以直接从其命名来区分，URI是Web资源的标志符，所以只要求它具有“标识性”即可；URL全称为“统一资源定位符（Uniform Resource Locator）”，所以除了标识性之外，它还具有定位的功能，用于描述Web资源所在的位置。

URL不仅仅用于定位目标资源所在的位置，还指名了获取资源所采用的协议，一个完整的URL包含协议名称、主机名称（IP地址或者域名）、端口号、路径和查询字符串5个部分。比如对于“ http://www.artech.com:8080/images/photo.png?size=small”这样一个URL，上述的5个部分分别是“http”、“www.artech.com”、“8080”、“/images/photo.png”和“?size=small”。

除了URL，URN也是URI的一种表现形式，URN全称“统一资源定位符（Uniform Resource Name）”。URN与资源所在的位置无关，倘若采用URN来唯一标识某个资源，在位置发生改变的时候标志符依然可以保持不变。URN一般也不会涉及到获取被标识资源采用的网络协议，所以不需要为利用不同协议访问的相同资源定义不同的标志符。

# 三、HTTP事务

虽然TCP是一种基于连接的传输层协议，并且保存双方针对同一个连接的多轮消息交换的会话状态，但是建立其上的HTTP则是一种无状态的网络协议。HTTP采用简单的“请求/响应”消息交换模式，一次HTTP事务（Transaction）始于请求的发送，止于响应的接收。针对客户端和Web服务器的多次消息交换来说，每个HTTP事务均是相互独立的。

## HTTP方法

HTTP采用简单的请求/响应模式的消息交换旨在实现针对某个Web资源的某种操作。至于针对资源的操作类型，不外乎CRUD（Create、Retrieve、Update和Delete）而已。一个HTTP请求除了利用URI标志目标资源之外，还需要通过HTTP方法（HTTP Method或者HTTP Verb）指名针对资源的操作类型。我们常用的HTTP方法 包括GET、POST、PUT、DELETE、HEAD、OPTIONS、TRACE、CONNECTION和PATCH等，我们将在《[设计篇](http://www.cnblogs.com/artech/p/restful-web-api-02.html)》以REST的视角来对它们进行详细介绍。

## 响应状态码

针对客户端向Web服务器发送的任意一个HTTP请求，不论在何种情况下得到一个响应，每个响应均具有一个由3位数字表示的状态码和相应的描述文字。不同数值的状态码体现了不同类型的响应状态，W3C对响应状态码的范围作了如下的规范。

* 100~199：信息状态码，代表请求已被接受，需要继续处理。
* 200~299：成功状态码，代表请求已成功被服务器接收、理解、并接受。
* 300~399：重定向状态码，代表需要客户端采取进一步的操作才能完成请求。
* 400~499：客户端错误状态码，代表了客户端看起来可能发生了错误，妨碍了服务器的处理。
* 500~599：服务器错误状态码，代表了服务器在处理请求的过程中有错误或者异常状态发生，也有可能是服务器意识到以当前的软硬件资源无法完成对请求的处理。

# 四、HTTP报文

客户端和Web服务器在一次HTTP事务中交换的消息被称为HTTP报头，客户端发送给服务器的请求消息被称为请求报文，服务器返回给客户端的响应消息被称为响应报头。请求报文和响应报头采用纯文本编码，由一行行简单的字符串组成。一个完整的HTTP报文由如下三个部分构成。

* 起始行：代表HTTP报文的第一行文字，请求报文利用起始行表示采用的HTTP方法、请求URI和采用的HTTP版本，而响应报文的起始行在承载着HTTP版本和响应状态码等信息。
* 报头集合：HTTP报文的起始行后面可以包含零个或者多个报头字段。每个报头表现为一个键/值对，键和值分别表示报头名称和报头的值，两者通过冒号（“：”）进行分割。HTTP报文采用一个空行作为报头集合结束的标志。
* 主体内容：代表报头集合结束标志的空行之后就是HTTP报文的主体部分了。客户端提交给服务器的数据一般置于请求报头的主体，而响应报头的主体也承载着服务器返回给客户端的数据。不论是请求报文还是响应报文，其主体部分均是可以缺省的。

接下来我们看看一个具体HTTP报文具有怎样的结构。下面这个文本片段反映的是我们通过Chrome浏览器访问微软的官网（www.microsoft. com）对应的HTTP请求，起始行体现了HTTP请求的三个基本属性，即HTTP方法（GET）、目标资源（http://www.microsoft.com/en-us/default.aspx）和协议版本（HTTP/1.1）。

1: GET http://www.microsoft.com/en-us/default.aspx HTTP/1.1

2: Host: www.microsoft.com

3: Connection: keep-alive

4: Cache-Control: max-age=0

5: User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/535.7 (KHTML, like Gecko) Chrome/16.0.912.75 Safari/535.7

6: Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

7: Accept-Encoding: gzip,deflate,sdch

8: Accept-Language: en-US,en;q=0.8

9: Accept-Charset: ISO-8859-1,utf-8;q=0.7,\*;q=0.3

10: Cookie: ...

上述这个请求报文不具有主体，所以起始行之外的所有内容均为报头集合，我们们可以根据这些报头获得主机名称、采用的缓存策略、浏览器相关信息、以及客户端支持的媒体类型（Media Type）、编码方式、语言和字符集等。

前面的HTTP请求通过浏览器发送给服务端之后会接收到具有如下结构的响应报文，我们可以此从它的起始行得到采用的HTTP版本（HTTP/1.1）和响应状态码（“200 OK”，表示请求被正常接收处理）。响应的内容被封装到响应报文的主体部分，其媒体类型的通过报头“Content-Type”表示。由于该响应报文的主体内容是一个HTML文档，所以“Content-Type”报头表示的媒体类型为“text/html”。

1: HTTP/1.1 200 OK

2: Cache-Control: no-cache

3: Pragma: no-cache

4: Content-Type: text/html; charset=utf-8

5: Content-Encoding: gzip

6: Expires: -1

7: Vary: Accept-Encoding

8: Server: Microsoft-IIS/7.5

9: X-AspNet-Version: 2.0.50727

10: VTag: 791897542300000000

11: P3P: CP="ALL IND DSP COR ADM CONo CUR CUSo IVAo IVDo PSA PSD TAI TELo OUR SAMo CNT COM INT NAV ONL PHY PRE PUR UNI"

12: X-Powered-By: ASP.NET

13: Date: Wed, 18 Jan 2012 07:06:25 GMT

14: Content-Length: 34237

15:

16: <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

17: <html>…</html>

[[1]](file:///E:\\????????????/ASP.NET%20Web%20API????????????/???1???%20ASP.NET%20Web%20API??????/" \l "_ftnref1_5509)超文本/超媒体（HyperText/HyperMedia）：超文本是一份呈现文本内容的电子文档，其核心在于可以利用内嵌的“超链接（Hyperlink）”直接访问引用的另一份文档。超媒体对超文本作了简单的扩展以呈现多媒体内容（比如图片、音频和视频等）。HTML文档是我们常见的最为典型的超文本/超媒体文件。

[[2]](file:///E:\\????????????/ASP.NET%20Web%20API????????????/???1???%20ASP.NET%20Web%20API??????/" \l "_ftnref2_5509) 除了采用这种4个层次的划分方法之外，还具有另外两种典型的划分方式。其中一种在链路层下面添加一个基于物理网络硬件的物理层，这种划分方法与此没有本质的区别。另外一种则是将TCP/IP协议簇划分为包括应用层、表示层、会话层、传输层、网络层、链路层和物理层在内的7个层次。

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

[我所理解的RESTful Web API [Web标准篇]](http://www.cnblogs.com/artech/p/restful-web-api-01.html)   
[我所理解的RESTful Web API [设计篇]](http://www.cnblogs.com/artech/p/restful-web-api-02.html)

# [我所理解的RESTful Web API [设计篇]](http://www.cnblogs.com/artech/p/restful-web-api-02.html)

《[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？

[](http://images.cnitblog.com/blog/19327/201401/06074820-3b08e822599440c28ea3f402a7fac67e.jpg)

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]](file:///E:\\????????????/ASP.NET%20Web%20API????????????/???1???%20ASP.NET%20Web%20API??????/" \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]](file:///E:\\????????????/ASP.NET%20Web%20API????????????/???1???%20ASP.NET%20Web%20API??????/" \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]](file:///E:\\????????????/ASP.NET%20Web%20API????????????/???1???%20ASP.NET%20Web%20API??????/" \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]](file:///E:\\????????????/ASP.NET%20Web%20API????????????/???1???%20ASP.NET%20Web%20API??????/" \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

[我所理解的RESTful Web API [Web标准篇]](http://www.cnblogs.com/artech/p/restful-web-api-01.html)   
[我所理解的RESTful Web API [设计篇]](http://www.cnblogs.com/artech/p/restful-web-api-02.html)