第34讲 | 基于JSON的RESTful接口协议：我不关心过程，请给我结果

2018-08-03 刘超

第34讲 | 基于JSON的RESTful接口协议：我不关心过程，请给我结果
朗读人：刘超　11′41″ | 5.36M

上一节我们讲了基于 XML 的 SOAP 协议，SOAP 的 S 是啥意思来着？是 Simple，但是好像一点儿都不简单啊！

你会发现，对于 SOAP 来讲，无论 XML 中调用的是什么函数，多是通过 HTTP 的 POST 方法发送的。但是咱们原来学 HTTP 的时候，我们知道 HTTP 除了 POST，还有 PUT、DELETE、GET 等方法，这些也可以代表一个个动作，而且基本满足增、删、查、改的需求，比如增是 POST，删是 DELETE，查是 GET，改是 PUT。

传输协议问题

对于 SOAP 来讲，比如我创建一个订单，用 POST，在 XML 里面写明动作是 CreateOrder；删除一个订单，还是用 POST，在 XML 里面写明了动作是 DeleteOrder。其实创建订单完全可以使用 POST 动作，然后在 XML 里面放一个订单的信息就可以了，而删除用 DELETE 动作，然后在 XML 里面放一个订单的 ID 就可以了。

于是上面的那个 SOAP 就变成下面这个简单的模样。

```
POST /purchaseOrder HTTP/1.1
Host: www.geektime.com
Content-Type: application/xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
 <order>
     <date>2018-07-01</date>
      <className> 趣谈网络协议 </className>
       <Author> 刘超 </Author>
       <price>68</price>
  </order>
```

而且 XML 的格式也可以改成另外一种简单的文本化的对象表示格式 JSON。

```
POST /purchaseOrder HTTP/1.1
Host: www.geektime.com
Content-Type: application/json; charset=utf-8
Content-Length: nnn

{
 "order": {
  "date": "2018-07-01",
  "className": " 趣谈网络协议 ",
  "Author": " 刘超 ",
  "price": "68"
 }
}
```

经常写 Web 应用的应该已经发现，这就是 RESTful 格式的 API 的样子。

协议约定问题

然而 RESTful 可不仅仅是指 API，而是一种架构风格，全称 Representational State Transfer，表述性状态转移，来自一篇重要的论文《架构风格与基于网络的软件架构设计》（Architectural Styles and the Design of Network-based Software Architectures）。

这篇文章从深层次，更加抽象地论证了一个互联网应用应该有的设计要点，而这些设计要点，成为后来我们能看到的所有高并发应用设计都必须要考虑的问题，再加上 REST API 比较简单直接，所以后来几乎成为互联网应用的标准接口。

因此，和 SOAP 不一样，REST 不是一种严格规定的标准，它其实是一种设计风格。如果按这种风格进行设计，RESTful 接口和 SOAP 接口都能做到，只不过后面的架构是 REST 倡导的，而 SOAP 相对比较关注前面的接口。

而且由于能够通过 WSDL 生成客户端的 Stub，因而 SOAP 常常被用于类似传统的 RPC 方式，也即调用远端和调用本地是一样的。

然而本地调用和远程跨网络调用毕竟不一样，这里的不一样还不仅仅是因为有网络而导致的客户端和服务端的分离，从而带来的网络性能问题。更重要的问题是，客户端和服务端谁来维护状态。所谓的状态就是对某个数据当前处理到什么程度了。

这里举几个例子，例如，我浏览到哪个目录了，我看到第几页了，我要买个东西，需要扣减一下库存，这些都是状态。本地调用其实没有人纠结这个问题，因为数据都在本地，谁处理都一样，而且一边处理了，另一边马上就能看到。

当有了 RPC 之后，我们本来期望对上层透明，就像上一节说的“远在天边，尽在眼前”。于是使用 RPC 的时候，对于状态的问题也没有太多的考虑。

就像 NFS 一样，客户端会告诉服务端，我要进入哪个目录，服务端必须要为某个客户端维护一个状态，就是当前这个客户端浏览到哪个目录了。例如，客户端输入 cd hello，服务端要在某个地方记住，上次浏览到 /root/liuchao 了，因而客户的这次输入，应该给它显示 /root/liuchao/hello 下面的文件列表。而如果有另一个客户端，同样输入 cd hello，服务端也在某个地方记住，上次浏览到 /var/lib，因而要给客户显示的是 /var/lib/hello。

不光 NFS，如果浏览翻页，我们经常要实现函数 next()，在一个列表中取下一页，但是这就需要服务端记住，客户端 A 上次浏览到 20～30 页了，那它调用 next()，应该显示 30～40 页，而客户端 B 上次浏览到 100～110 页了，调用 next() 应该显示 110～120 页。

上面的例子都是在 RPC 场景下，由服务端来维护状态，很多 SOAP 接口设计的时候，也常常按这种模式。这种模式原来没有问题，是因为客户端和服务端之间的比例没有失衡。因为一般不会同时有太多的客户端同时连上来，所以 NFS 还能把每个客户端的状态都记住。

公司内部使用的 ERP 系统，如果使用 SOAP 的方式实现，并且服务端为每个登录的用户维护浏览到报表那一页的状态，由于一个公司内部的人也不会太多，把 ERP 放在一个强大的物理机上，也能记得过来。

但是互联网场景下，客户端和服务端就彻底失衡了。你可以想象“双十一”，多少人同时来购物，作为服务端，它能记得过来吗？当然不可能，只好多个服务端同时提供服务，大家分担一下。但是这就存在一个问题，服务端怎么把自己记住的客户端状态告诉另一个服务端呢？或者说，你让我给你分担工作，你也要把工作的前因后果给我说清楚啊！

那服务端索性就要想了，既然这么多客户端，那大家就分分工吧。服务端就只记录资源的状态，例如文件的状态，报表的状态，库存的状态，而客户端自己维护自己的状态。比如，你访问到哪个目录了啊，报表的哪一页了啊，等等。

这样对于 API 也有影响，也就是说，当客户端维护了自己的状态，就不能这样调用服务端了。例如客户端说，我想访问当前目录下的 hello 路径。服务端说，我怎么知道你的当前路径。所以客户端要先看看自己当前路径是 /root/liuchao，然后告诉服务端说，我想访问 /root/liuchao/hello 路径。

再比如，客户端说我想访问下一页，服务端说，我怎么知道你当前访问到哪一页了。所以客户端要先看看自己访问到了 100～110 页，然后告诉服务器说，我想访问 110～120 页。

这就是服务端的无状态化。这样服务端就可以横向扩展了，一百个人一起服务，不用交接，每个人都能处理。

所谓的无状态，其实是服务端维护资源的状态，客户端维护会话的状态。对于服务端来讲，只有资源的状态改变了，客户端才调用 POST、PUT、DELETE 方法来找我；如果资源的状态没变，只是客户端的状态变了，就不用告诉我了，对于我来说都是统一的 GET。

虽然这只改进了 GET，但是已经带来了很大的进步。因为对于互联网应用，大多数是读多写少的。而且只要服务端的资源状态不变，就给了我们缓存的可能。例如可以将状态缓存到接入层，甚至缓存到 CDN 的边缘节点，这都是资源状态不变的好处。

按照这种思路，对于 API 的设计，就慢慢变成了以资源为核心，而非以过程为核心。也就是说，客户端只要告诉服务端你想让资源状态最终变成什么样就可以了，而不用告诉我过程，不用告诉我动作。

还是文件目录的例子。客户端应该访问哪个绝对路径，而非一个动作，我就要进入某个路径。再如，库存的调用，应该查看当前的库存数目，然后减去购买的数量，得到结果的库存数。这个时候应该设置为目标库存数（但是当前库存数要匹配），而非告知减去多少库存。

这种 API 的设计需要实现幂等，因为网络不稳定，就会经常出错，因而需要重试，但是一旦重试，就会存在幂等的问题，也就是同一个调用，多次调用的结果应该一样，不能一次支付调用，因为调用三次变成了支付三次。不能进入 cd a，做了三次，就变成了 cd a/a/a。也不能扣减库存，调用了三次，就扣减三次库存。

当然按照这种设计模式，无论 RESTful API 还是 SOAP API 都可以将架构实现成无状态的，面向资源的、幂等的、横向扩展的、可缓存的。

但是 SOAP 的 XML 正文中，是可以放任何动作的。例如 XML 里面可以写 < ADD >，< MINUS > 等。这就方便使用 SOAP 的人，将大量的动作放在 API 里面。

RESTful 没这么复杂，也没给客户提供这么多的可能性，正文里的 JSON 基本描述的就是资源的状态，没办法描述动作，而且能够出发的动作只有 CRUD，也即 POST、GET、PUT、DELETE，也就是对于状态的改变。

所以，从接口角度，就让你死了这条心。当然也有很多技巧的方法，在使用 RESTful API 的情况下，依然提供基于动作的有状态请求，这属于反模式了。

服务发现问题

对于 RESTful API 来讲，我们已经解决了传输协议的问题——基于 HTTP，协议约定问题——基于 JSON，最后要解决的是服务发现问题。

有个著名的基于 RESTful API 的跨系统调用框架叫 Spring Cloud。在 Spring Cloud 中有一个组件叫 Eureka。传说，阿基米德在洗澡时发现浮力原理，高兴得来不及穿上裤子，跑到街上大喊：“Eureka（我找到了）！”所以 Eureka 是用来实现注册中心的，负责维护注册的服务列表。

服务分服务提供方，它向 Eureka 做服务注册、续约和下线等操作，注册的主要数据包括服务名、机器 IP、端口号、域名等等。

另外一方是服务消费方，向 Eureka 获取服务提供方的注册信息。为了实现负载均衡和容错，服务提供方可以注册多个。

当消费方要调用服务的时候，会从注册中心读出多个服务来，那怎么调用呢？当然是 RESTful 方式了。

Spring Cloud 提供一个 RestTemplate 工具，用于将请求对象转换为 JSON，并发起 Rest 调用，RestTemplate 的调用也是分 POST、PUT、GET、 DELETE 的，当结果返回的时候，根据返回的 JSON 解析成对象。

通过这样封装，调用起来也很方便。

小结

好了，这一节就到这里了，我们来总结一下。

- SOAP 过于复杂，而且设计是面向动作的，因而往往因为架构问题导致并发量上不去。

- RESTful 不仅仅是一个 API，而且是一种架构模式，主要面向资源，提供无状态服务，有利于横向扩展应对高并发。

最后，给你留两个思考题：

1. 在讨论 RESTful 模型的时候，举了一个库存的例子，但是这种方法有很大问题，那你知道为什么要这样设计吗？
2. 基于文本的 RPC 虽然解决了二进制的问题，但是它本身也有问题，你能举出一些例子吗？

我们的专栏更新到第 34 讲，不知你掌握得如何？每节课后我留的思考题，你都有没有认真思考，并在留言区写下答案呢？我会从已发布的文章中选出一批认真留言的同学，赠送学习奖励礼券和我整理的独家网络协议知识图谱。

欢迎你留言和我讨论。趣谈网络协议，我们下期见！

精选留言

feifei 2

在讨论 RESTful 模型的时候，举了一个库存的例子，但是这种方法有很大问题，那你知道为什么要这样设计吗？

此方法的问题在于，不是解决问题，而是将数据状态进行了转移，将状态交给存储，这样业务将可以无状态化运行，这种设计可以很好的解决扩展的问题，因为无状态，可以进行负载均衡！使用集群化来解决单机的问题。

基于文本的 RPC 虽然解决了二进制的问题，但是它本身也有问题，你能举出一些例子吗?

1，效率问题，程序与文本之间转换效率低，因而不适合内部大数据交换，因为文本利用阅读，对外采用较好

2，相比于二进制rpc,传输需要的带宽更大，二进制的rpc因为可以使用专用的客户短和服务器代码，可以更好的压缩数据，以提供更大的吞吐量

2018-08-03

凡凡 0

第一个问题有点模糊，感觉这个设计没有问题，任何传输方式，一旦经过网络，都会发生很多可能性，必须做幂等处理。如果指的是服务端无状态的话，原因就是提升扩展性，应对C端用户的大规模和并发。

第二个问题，主要在于序列化和传输。序列化方面由于有格式就不如二进制紧凑，传输的数据量相对来说要大。另一个是二进制可以自定义规范，或者编码方案，传输路径上，数据被截获，也不容易解析。

2018-08-03

波 0

高并发下 尽管接口支持了幂等性 库存资源修改时需要支持cas操作

2018-08-03

_CountingStars 0

2.字段冗余

有时候soap只需要一次调用能完成的功能 restful可能需要多次api调用才能完成

不是所有动作都可以转换为资源 比如 login

2018-08-03