维普资讯 http://www.cqvip.com

中冗余节点已被除去，所有节点呈现出标准的树形结构。

4. 上面我们通过SQL语句已经完成了数据从二维表到树形结构逻辑上的转变。转变之后节点之间的关系可以有标准的父子-兄弟关系来描述。接下来需要进行的是该树形结构的物理文件实现和数据查询API的实现。因为表3中节点之间的父子-兄弟关系非常明确，笔者使用了经典的子女-兄弟链接法以Java语言编写程序来物理地存储上面的树形数据，实际应用中提高检索效率的效果很好。在该算法中每个存储单位由三部分组成：有效信息、最小孩子地址和紧邻兄长地址。（详见下图2）

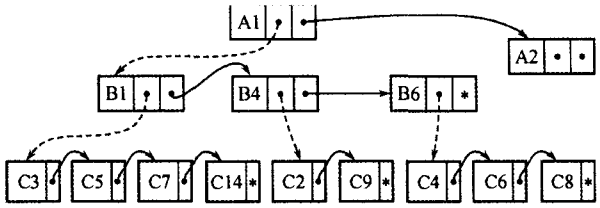

图2 子女-兄弟链接法

[注释3]：每个记录设两类指针，分别指向最左边的子女（每个记录型对应一个）和最近的兄弟。如何用程序实现子女-兄弟链接式的数据存储结构，和数据查询API，在这里就不深入讨论了，以后有机会我会对此进行专门的论述。■

参考资料

SourceForge有一个很好的Java开源项目dbsql2xml，号称能够将关系数据库转换为层次结构的XML。。虽然它只是形式上把存在数据库中的数据转换为XML形式，在逻辑结构上并没有做实质性的转换工作，但是它提供了很好的做进一步基于Java开发的基础。有兴趣研究的话请参看：

http://dbsql2xml.sourceforge.net/

作者简介

聂伟，毕业于北京理工专攻信息安全和数据库。毕业后曾从事信息管理系统（MIS）程序开发一年。后来进入微软全球技术支持中心从事数据库开发支持两年。现就职于SPX公司从事汽车电子相关的嵌入式系统和数据库系统开发。对数据库和C++/Java都有所研究，尤其擅长数据库。

■ 责任编辑:赵健平（zhaojp@csdn.net）

在了解过REST之后，你肯定很想知道这个概念在你的实际应用当中究竟能派上多大用场。我将在本文解答在进行REST研究时容易产生的十点疑惑。

REST也许适用于CRUD，但并不适用于“真的”业务逻辑

这是那些对REST的好处持怀疑态度的人最常见的反应。毕竟，要是你只能create/read/update/delete，那你如何表达更复杂的应用语义呢？我已经在本系列文章的引言部分探讨过这些被大家所关心的问题了，不过这方面绝对值得进一步讨论。

首先，HTTP动词（verbs）——GET、PUT、POST和DELETE——跟CRUD操作并不是一一对应的。例如，POST和PUT都可用于创建新资源，它们的区别在于：PUT请求是由客户端决定（被创建或更新的）资源的URI；而POST请求是向一个“集合(collection)”或“工厂(factory)”资源发出的，是由服务器来指派URI的。不过无论怎样，我们回到那个问题：如何应付更复杂的业务逻辑呢？

任何返回一个结果c的计算calc(a, b)，都可被转换为一个标识其结果的URI——例如x = calc(2,3)可被转换为http://example.com/calculation?a=2&b=3。初看，这仿佛是完全错误的REST式HTTP的用法——我们应当用URIs来标识资源（resources）而不是操作（operations），不是吗？没错，其实你就是这么做的：http://example.com/sum?augend=2&addend=3标识的是一个资源，即2加3的结果。在这一特定的（显然是精心设计的）示例中，你用GET来获取计算结果是没问题的——毕竟它是可缓存的（cacheable），你可以引用它，而且该计算多半是安全的（safe）且代价很小。

当然，在许多（即便称不上大多数）情况下，用GET来执行计算也许是会犯错的。别忘了，GET应当是一个“安全的（safe）”操作，也就是说，假如客户端只是通过发出GET请求来跟随一个链接，那么它不承担任何义务（比如因使用你的服务而向你付费）或责任。所以，在许多其他情况下，“通过POST请求向服务器提供输入数据、以便服务器新建一个资源”是更合适的做法。服务器在响应该POST请求时，可以给出结果的URI（而且有可能把你转向过去）。这个结果接下来便可被重用、被加入书签、在获取时被缓存等等。你基本上可以将这一模型推广应用到任何产生结果的操作——这涵盖几乎你能想到的所有操作。

没有正式的契约与描述语言

从RPC到CORBA，从DCOM到Web服务，我们已

维普资讯 http://www.cqvip.com

回答关于REST的十点疑问

在了解过REST之后，你肯定很想知道这个概念在介绍性的、“Hello, World”级的东西以外能派上多大用场。本文，Stefan Tilkov解答了人们——尤其是那些深谙基于SOAP/WSDL的Web服务架构手法的人——起初研究REST时容易产生的有关REST的十点疑惑。

■ 译 / 徐涵

习惯于拥有一个“列出操作、名称及输入输出参数类型”的接口描述（interface description）了。没有接口描述语言的话，REST怎么用呢？

就这一被十分频繁问到的问题，我有三点答复。

首先，假如你决定用XML（这是很普遍的做法）来配合REST式HTTP的话，那么各种现有的XML模式语言（schema languages）（如RELAX NG、Schematron等）仍旧可供你使用。可以说，一个用WSDL描述的东西，常常有95%的内容并不是跟WSDL相关、而是跟你定义的XML Schema复杂类型（complex types）相关的。WSDL所增加的，大部分是有关操作（operations）及其名称的——对于REST的统一接口（uniform interface）来说，描述这些是颇为无趣的，因为GET、PUT、POST和DELETE就是你所能使用的全部操作了。关于XML Schema的使用，这意味着，即便你依赖于一个REST式接口，你仍旧可使用你所偏爱的数据绑定工具（假如刚好你有的话）来为你偏爱的语言生成数据绑定代码。（回答还没结束，见下。）

第二，问问你自己需要描述做什么。最常见（尽管并非唯一）的用例（use case）是：描述被用来给接口生成桩（stubs）和骨架（skeletons）代码。它通常不是文档（documentation），因为WSDL格式的描述并不告诉你操作的语义——它只是告诉你操作的名称。你得通过一些人人类可读的文档来了解如何调用它。典型的REST做法是，你应提供HTML格式的文档，其中可能包含指向你的资源的直接链接（direct links）。如果你采取提供多个表示（multiple representations）的做法的话，那么你可以真正拥有自文档化的（self-documenting）资源——你只要在浏览器中对一个资源做HTTP GET请求，就可以得到一个HTML文档，其中不但包含数据，还包含你可以对它执行的操作（HTTP动词）的列表以及它接受和返回的内容类型（content types）。

最后，假如你坚持为你的REST式服务（RESTful service）使用描述语言，那么你可以使用WADL（Web Application Description Language，Web应用描述语言），或适当地使用WSDL 2.0（其制定者声称它也可以描述REST式服务）。不过，WADL和WSDL 2在描述超媒体（hypermedia）方面均无帮助——而且考虑到这是REST的核心方面之一，我不太确信它们是否充分可用。

谁真会把他们应用中如此多的实现细节暴露出来？

另一个普遍关心的问题是，资源太低层（low-level），暴露了那些不应暴露出来的实现细节。说到底，这不就把“按有意义的方式来运用资源”的担子加到客户端（用户）的身上了吗？

简单的回答是：不。一个资源的GET、PUT或其他方法的实现，可以跟一个“服务”或RPC操作的实现复杂程度相当。应用REST设计原则，并不是说你必须把下层数据模型（underlying data model）中的各项暴露出来——它只意味着，你采用以数据为中心的（data-centric）方式、而不是以操作为中心的（operation-centric）方式把业务逻辑暴露出来。

一个相关的关切是，不支持对资源的直接访问将增加安全性。这是由“通过隐匿得到安全（security by obscurity）”这条陈旧的谬论得出的结论。人们可以这样反驳：其实恰恰相反，如果你隐瞒你通过特定于应用的协议访问哪些资源，你将无法利用基础设施（infrastructure）来保护它们。通过为有意义的资源指派单独的URI，你可以利用Apache的安全规则（以及重写逻辑、日志和统计等）对不同资源采取不同处理。把这些明确化了，你的安全性将得到提升，而不是降低。

维普资讯 http://www.cqvip.com

REST只能配合HTTP使用，它不是传输协议无关的

首先，毫无疑问，HTTP不是一种传输协议（transport protocol），而是一种应用协议（application protocol）。它采用TCP作为下层传输（underlying transport），但它拥有自己的语义（否则它就没什么用处了）。仅将HTTP作为传输，是不恰当的。

第二，抽象未必总是好事。Web服务的做法，是试图把许多大不相同的技术隐藏在单个抽象层背后——但这容易引发抽象泄露（leak）。例如，通过JMS和通过HTTP请求发送消息存在着巨大的不同，试图把各种存在极大差异的技术弱化为它们的最基本共通特性是毫无益处的。打个比方，如果要创建一个通用抽象（common abstraction）、把一个关系数据库和一个文件系统隐藏在一个通用API之后，当然这可以去做，但一旦你涉及解决像查询这样的问题时，该抽象的问题就显露出来了。

最后，正如Mark Baker曾说过的："协议无关性是一个缺陷，而不是一个特性"。虽然这给人的最初感觉是比较奇怪，但你要知道，真正的协议无关性（protocol independence）是不可能实现的——你所能做的只是决定依赖于哪一种协议。依赖于一种得到了广泛采纳和官方标准化的协议（如HTTP）根本不是问题，而且比起试图取代它的某种抽象，它得到了更广泛的普及与支持。

没有实际的、明确而一致的指南教你如何设计REST式（RESTful）应用

REST式设计在许多方面均没有"官方"最佳实践和"如何按符合REST原则的方式、用HTTP解决一个特定问题"的标准方式。毋庸置疑，这是会逐渐得到改善的。尽管如此，REST具体表达了比基于WSDL/SOAP的Web服务更多的应用概念。换言之，虽然该批评对REST有很大价值，但这一批评更适用于其替换技术（它们基本上没有向你提供任何建议）。

有时，这种疑虑以"连REST专家们都无法就具体怎么做达成一致"的形式出现。但一般说来，情况并不是这样——举个例子，我比较相信我数周前在这里讲述的核心概念尚未（而且也不会）遭到REST圈内人士（假定存在这个圈子的话）的质疑，这并不是因为那是一篇特别好的文章，而是因为人们在做过更深入的了解以后便能达成许多共识。假如你有机会做个实验的话，可以试试看，是让五位SOA支持者在某方面达成一致更容易，还是让五位REST支持者在某方面达成一致更容易。根据我个人的过往经验和长期参与数个SOA与REST讨论组的经历来看，我倾向于相信后者更加容易。

REST不支持事务

"事务（transaction）"一词存在着多种不同解释，不过人们一般所说的事务，指的是数据库里的ACID这种。在一个SOA环境中——无论是否基于Web服务或HTTP——各个服务（或系统、或Web应用）的实现仍然有可能与一个支持事务的数据库进行交互：这无需很大改变，假如你不用显式创建事务的话（除非你的服务运行在一个EJB容器或其他可以为你处理事务创建的环境中）。如果你与多个资源交互，情况也一样。

如果你打算把事务组合（或者创建）为一个更大的单元，情况将有所不同。在Web服务环境中，至少有一种办法可以做到跟人们所熟知的（比如Java EE环境所支持的）两阶段提交（2PC）相似：即采用WS-Atomic Transaction（WS-AT），它是WS-Coordination标准族中的成员。本质上，WS-AT所实现的是跟XA规定的两阶段提交协议非常相似或相同的。这意味着，你的事务上下文（transaction context）将用SOAP报头来传播，而你的实现（implementation）将负责确保资源管理器进入现有事务。本质上，跟EJB开发者所熟悉的模型一样——你的分布式事务跟本地事务一样是原子性的。

关于SOA环境中的原子事务（atomic transactions），有很多看法或反对意见：

- 松耦合与事务（尤其是ACID那样的）根本格格不入。比如"跨越多个独立系统来协调提交，会在它们之间造成紧耦合"就充分说明了这一点。
- 为了进行这种协调，需要对所有服务进行中央控制——而跨越企业边界进行两阶段提交事务是不可能或基本无法做到的。
- 支持这种事务所需的基础设施（infrastructure）通常极为昂贵和复杂。

很大程度上，在SOA或REST环境中需要ACID事务，其实是一种设计异味（design smell）——你很可能已经为你的服务或资源采用了错误的模型。当然，原子事务只是一种类型的事务——除此以外还有扩展的事务模型，也许它更适合松耦合系统。不过，即便在Web服务阵营里，它们也没得到较多采纳。

REST是不可靠的

常有人指出，REST式HTTP里没有与WS-ReliableMessaging对等的特性，于是许多人便断定，REST不能应用于讲究可靠性（reliability）的场合（那就是说差不多所有跟业务场景相关的系统）。但很多时候，你不一定需要一个处理消息递送（message delivery）的基础设施组件（infrastructure component），相反，你需要知道一个消息是否已被递送。

维普资讯 http://www.cqvip.com

通常，收到一个响应消息——比方说HTTP里的200 OK——表明你知道你的通信伙伴已经收到请求。但如果你没有收到响应消息，那问题就来了：你不知道是你的请求没有到达另一端，还是已经收到了（触发了某些处理）、但响应消息丢失了。

确保请求消息抵达另一端的最简单的做法，就是把消息重发一遍——当然，仅当接受方有能力处理重复消息（比如通过忽略它们）时才可以这么做。这种能力被称作幂等性（idempotency）。HTTP确保GET、PUT和DELETE是幂等的（idempotent）——如果你的应用实现得当的话，那么客户端在没有收到响应时只需把请求重发一遍即可。但POST消息不是幂等的——至少在HTTP规范里没有保证。对此你有多种选择：要么改用PUT（如果你的语义可以映射上去的话），采用Joe Gregorio描述的一种常见的最佳实践；要么采纳一种致力于统一有关做法的提案（例如Mark Nottingham的POE（POST Once Exactly）、Yaron Goland的SOA-Rity或Bill de hora的HTTPLR）。

就我个人而言，我倾向于上述第一种做法——即把可靠性问题转嫁到应用设计方面，不过对此存在多种不同看法。

尽管这些方案均解决了相当一部分可靠性问题，但没有（或至少就我所知没有）一个能支持消息递送承诺，比如按序递送一系列HTTP请求和响应。不过值得一提的是，许多现有的SOAP/WSDL方案没有依靠WS-ReliableMessaging（或其前身）也勉强应付了。

不支持发布/订阅

本质上，REST基于的是一种客户端-服务器模型（client-server model），HTTP总把客户端和服务器称为通信端点（endpoints of communication）。客户端通过发送请求和接受响应的方式与服务器进行交互。在发布/订阅模型（publish/subscribe model）中，客户订阅特定种类的信息，然后每当有新信息出现时它就会得到通知。REST式HTTP环境怎么可能支持发布/订阅呢？

寻找理想的例子并不难，聚合（syndication）就是一个。RSS和Atom Syndication都是聚合的例子。客户端通过“向一个代表变更集合（collection of changes）的资源发出HTTP请求”来查询新信息。这搞不好会相当低效，但实际上并没有，因为GET是Web上最优化的操作。其实，你可以很容易想象，要是一个受欢迎的博客服务器主动把各个变更通知各订阅者的话，那么它应该可以在可伸缩性方面得到很大提高。轮流通知（notification by polling）具有极好的可伸缩性。

你可以将这一聚合模型（syndication model）推广应用到你的各个应用资源——例如，为用户资源或账目审计追踪记录的变更提供Atom提要（feed）。除了可以满足任意数量应用的订阅需求，你还可以用提要阅读器（feed reader）来查看这些提要（feeds），就像在浏览器里查看一个资源的HTML表示（representation）一样。

当然，在某些情况下这就不合适了。比如，对于软实时（soft real-time）需求，采用其他技术也许更为合适。但在许多情况下，由聚合模型赢得的松耦合（loose coupling）、可伸缩性（scalability）与通知（notification），整体上是相当不错的。

无异步交互

在HTTP的请求/响应模型之下，如何实现异步通信？同样地，我们应注意到人们在谈及异步性（asynchronicity）时常常指的是不同方面。有人指的是编程模型，它可以是跟线上交互（wire interactions）无关的阻塞或非阻塞模型。这不是我们所关心的。但假如你把一个请求从客户端（用户）递送到服务器端（提供者）的过程需花费数小时，这怎么办呢？用户如何知道处理有没有结束？

HTTP有一个专门的响应代码202 Accepted，它的意思是“请求已被接受处理，但处理还没有结束”。显然，这正是你所需要的。至于处理结果，有多种办法：服务器可以返回一个资源的URI，然后客户端通过向该URI发送GET请求来访问结果（尽管在专门为一个请求创建资源时采用响应代码201 Created更为恰当）。或者，客户端可以提供一个URI，并期待服务器在处理完成后把结果POST上去。

缺少工具

最后一点，人们常常抱怨缺少用于支持REST式HTTP开发的工具。正如我在第二点里提到的，在数据方面其实不是这样——你可以使用你熟悉的数据绑定与其他数据APIs，因为这与方法数量和调用它们的方式无关。至于普通的HTTP与URI支持，现在所有的编程语言、框架及工具包都能提供立即支持。最后，厂商们正在为“用它们的框架进行更便捷的REST式HTTP开发”提供越来越多的支持，例如Sun的JAX-RS（JSR 311）、微软的.NET 3.5及ADO.NET数据服务框架里对REST的支持。■

查看英文原文：http://www.infoq.com/articles/tilkov-rest-doubts。

本文经InfoQ中文站（www.infoq.com/cn）授权刊登。

作者简介

徐涵，中文W3C技术推广网站W3China（w3china.org）创始人，开放翻译计划（transwiki.org）发起人，W3C特邀专家，InfoQ中文站编辑。

■ 责任编辑：赵健平（zhaojp@csdn.net）