改善可扩展性与简约性的解决方案

RESTful Web Services Cookbook

中文版

Subbu Allamaraju 著

丁雪丰 常可 译 李锟 审校

O'REILLY® | YAHOO! PRESS

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

RESTful Web Services Cookbook 中文版

Subbu Allamaraju 著

丁雪丰 常可 译

李锟 审校

電子工業出版社
Publishing House of Electronics Industry
北京·BEIJING

本书是《RESTful Web Services Cookbook》的中文翻译版。

本书从实践出发，涉及设计 RESTful Web 服务的各个方面，通过问题描述、解决方案、问题讨论的形式在 14 个章节中详细讨论了统一接口、资源、表述、URI、链接、请求、缓存、安全等诸多内容。无论读者是否设计过 RESTful Web 服务，具体使用哪种语言，都能在阅读过程中有所收获。本书也可作为手册，根据具体问题描述在书中查找解决办法。

版权贸易登记号　图字：01-2011-4234

图书在版编目（CIP）数据

RESTful Web Services Cookbook 中文版/(美)阿拉马拉尤（Allamaraju,S.）著；丁雪丰，常可译．

北京：电子工业出版社，2011.9

书名原文：RESTful Web Services Cookbook

ISBN 978-7-121-14390-8

Ⅰ. ①R… Ⅱ. ①阿… ②丁… ③常… Ⅲ. ①互联网络 - 网络服务器 - 程序设计 Ⅳ. ①TP368.5

中国版本图书馆 CIP 数据核字（2011）第 168493 号

策划编辑：　张春雨

责任编辑：　刘　舫

封面设计：　Karen Montgmery　张　健

印　　刷：　北京中新伟业印刷有限公司

装　　订：

出版发行：　电子工业出版社

北京市海淀区万寿路 173 信箱　　邮编：100036

开　　本：　787×980　1/16　　印张：20.25　　字数：356 千字

印　　次：　2011 年 9 月第 1 次印刷

印　　数：　4000 册　　定价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

O'Reilly Media, Inc.介绍

O' Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O' Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些"细微的信号"来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O' Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O' Reilly 为软件开发人员带来革命性的"动物书"；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O' Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O' Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O' Reilly 的产品都反映了公司不可动摇的理念—— 信息是激发创新的力量。

业界评论

"O' Reilly Radar 博客有口皆碑。"

—— Wired

"O' Reilly 凭借一系列(真希望当初我也想到了)非凡想法建立了数百万美元的业务。"

—— Business 2.0

"O' Reilly Conference 是聚集关键思想领袖的绝对典范。"

—— CRN

"一本 O' Reilly 的书就代表一个有用、有前途、需要学习的主题。"

—— Irish Times

"Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：'如果你在路上遇到岔路口，走小路（岔路）。' 回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。"

—— Linux Journal

推荐序

从 3 万英尺到 3 英尺

REST（表述性状态转移）是一种与 DO（分布式对象）、RPC（远程过程调用）齐名的分布式应用架构风格（architectural style）。REST 诞生于 Roy Fielding（HTTP 1.1 协议的主要设计者）在 2000 年所著的博士论文"Architectural Styles and the Design of Network-based Software Architectures"（中文版名为《架构风格与基于网络的软件架构设计》）。REST 正是 Web 自身的架构风格，它是理解 Web 架构的关键所在，也是 HTTP 1.1 协议的设计原理，因此 REST 在 Web 开发领域的重要性是可想而知的。

客观来说，对于一线开发者来说，REST 的概念是相当抽象的。与 REST 相比，一线开发者对于 DO 和 RPC 这两种架构风格理解起来要容易很多。而且一线开发者更习惯于通过具体的代码例子来学习，而不是通过抽象的概念。这个问题造成了对 REST 的大量误解，在其诞生之后的很多年中，REST 一直给人以一种“不接地气”或者“阳春白雪”的感觉。

在 2005 年之后，REST 的受关注程度因为很多方面的因素被放大了：

- Web 2.0 的兴起
- AJAX 的兴起
- Ruby on Rails 的兴起
- 云计算的兴起
- 移动互联网的兴起

REST 这种架构风格从小众走向大众，是近几年可以看到的一个明显趋势。

今天的 Web 早已不再仅仅是一个内容发布的平台，它已经发展成为了一个全球范围的计算平台。Web 的消费者从以人类用户为主，发展到了人类用户（Web 应用）和程序用户（Web 服务）并重。而开发 Web 服务，最佳选择就是 RESTful Web Services（REST 风格的 Web 服务），基于 SOAP/WSDL 的旧式 Web Services 已经很少有人使用。

有人说，软件架构师应该从 3 万英尺的高度来思考软件系统的整体架构设计。那么，当整体架构设计（RESTful Web Services）基本确定之后，一线开发者所处的高度，相当于是 3 英尺，可以说立即陷入了刺刀见红的肉搏战之中。业界迫切期待一些实战性强的、面向一线开发者的 REST 开发图书。在这类图书中，质量最高的要算是 O’Reilly 公司的 REST 开发系列图书，从 2007 年出版的《RESTful Web Services》开始。

《RESTful Web Services Cookbook》可以看做是《RESTful Web Services》一书的姊妹篇。《RESTful Web Services》这本书虽然不错，但是更多地还是为架构师而写。对于一线开发者来说，这本书缺乏很多开发细节（“魔鬼藏在细节中”），因此显得实战性不足。《RESTful Web Services Cookbook》恰好弥补了《RESTful Web Services》在实战性方面的不足。

设计良好的 REST 开发框架，自从 2005 年 Ruby on Rails 1.2 版开始逐渐出现。REST 开发的代码例子越来越多，Web 开发者社区还总结出了很多最佳实践。这些代码例子和最佳实践，需要有人加以收集和整理。《RESTful Web Services Cookbook》恰逢其时，为此做出了重要贡献。本书可以说是 Web 服务开发者的一件百宝箱，其中的内容几乎涵盖了开发 RESTful Web Services 的所有方面。笔者对于这本书有一种爱不释手的感觉，并且将其列为案头的常备图书之一。

通常对于新知识的认识过程是 What、Why、How，《RESTful Web Services Cookbook》这本书并没有涉及到 REST 架构风格的 What 和 Why，全部内容讲的都是 How。本书假设读者已经深入理解了 REST 架构风格，确信 REST 能够为当前开发的项目带来巨大好处，已经准备挽起袖子干活了。如果读者想要深入理解 REST 架构风格的 What 和 Why，可以参考 O'Reilly 公司的另外两本书：《RESTful Web Services》和《REST in Practice》。

作为专业的 Web 开发者，让我们所开发的 Web 服务拥抱 Web 的架构风格，融入 Web，成为 Web 的一部分!

李锟

2011 年 8 月 13 日 于上海

译者序

有人说计算机搞的是科学，也有人说计算机搞的是工程，于是大学里的计算机系通常叫“计算机科学与工程系”。两种说法究竟孰对孰错，我们不去深究，但请允许我做一个也许不怎么恰当的对比：

- 1905 年，Albert Einstein 提出了具有划时代意义的相对论，100 年过去了，绝大多数人只是知道世上有这么一个伟大的理论，真正理解它的人却寥寥无几。
- 2000 年，Roy Fielding在他的博士论文[注 1]中提出了“表述性状态转移”（REST），10 年过去了，很多开发者都知道REST，但真的能把它说明白的同样没几个。

两者的境遇很相似，物理学家总数就不多，理解相对论的人少也还说得过去，可为什么说很多开发者都不理解 REST 呢？以 Fielding 博士设计的 HTTP 协议为例，大家都把它当做一种传输协议，但 HTTP 其实是为 REST 而生的，它能够表达状态和状态转移，这就是它位于应用层而非传输层的原因，所以说 HTTP 中的 Transfer 被翻译成“转移”更为恰当。

如果说是Rails让大家开始真正关注REST，那么开放平台的兴起则让REST越来越多地出现在舞台上。各种基于HTTP的服务都宣称自己是REST风格的，曾经有段时间，不挂个REST的牌子出门都不好意思和人打招呼，哪怕自己是挂羊头卖狗肉也得和REST扯上关系。最后，Fielding博士非常失望，只能亲自撰写文章[注 2]告诉大家——你们搞错了，我设计的REST并非如此。

那么，真正的 REST 服务究竟是怎么样的呢？如果您也曾经读过那篇论文，或者是尝试读过，一定会发现要读懂它真得花一番功夫。有没有人可以用通俗易懂的方式指导大家设计并实现 REST 服务呢？雅虎的资深架构师 Subbu Allamaraju 做到了，本书涉及了设计 RESTful Web 服务的方方面面，总结了他多年的设计经验，书中没有枯燥冗长的理论说明，而是通过大量生动的范例来说明那些最佳实践，“问题描述”、“解决方案”和“问题讨论”这样的安排也让阅读更有针对性。无论您使用的是什么语言，都可以选择本书作为设计服务的参考，原因有两个，一是设计好的服务的原则是不随语言而变化的，二是本书的范例全部都是 HTTP 报文，无论使用何种语言、何种框架，最终都会变成 HTTP 报文。因此，没有什么理由可以让我们拒绝它。

注 1： 论文标题为“Architectural Styles and the Design of Network-based Software Architectures”，http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm。2007 年，李锟等人将该论文翻译为中文，发布于 http://www.redsaga.com/opendoc/REST_cn.pdf。

注 2： 文章标题为“REST APIs must be hypertext-driven”，http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven。

本书的翻译过程有些纠结，但收获也很多，至少让我对REST有了更清晰的认识。感谢李锟把本书介绍给了我，并建议我来主导全书的翻译，我们做了很多深入的沟通，探讨了很多实际的问题[注3]。在我快要抓狂的时候，常可加入了进来，他为读者能早日见到本书做出了很多贡献。同样也要感谢唐力群与郑佰云之前的协助，还有博文视点的多位编辑，正是有了这么多人的努力，才有了大家现在看到的这本书，希望它能给大家带来一些实实在在的帮助。如果您有什么意见或建议，或发现了书中翻译的错误，欢迎通过各种渠道（比如我的新浪微博@DigitalSonic）告诉我们。

丁雪丰

2011 年 6 月

注 3： 我们甚至还讨论过 Hypertext Transfer Protocol 该如何翻译。李锟建议翻译为“超文本转移协议”，要纠正之前错误的认识，而我则认为对于约定俗成的名字应该保持原样，并加以说明。最后争论不下，决定书中的 Hypertext Transfer Protocol 不做翻译，单独出现的 transfer 则明确翻译为“转移”。

目录

第 1 章

使用统一接口

HTTP是一种应用层协议，它定义了客户端与服务器之间的转移[注1]操作的表述形式。在此协议中，诸如GET，POST，PUT和DELETE之类的方法是对资源的操作。有了它，您就无须创造createOrder，getStatus，updateStatus等应用程序特定的操作了。能从HTTP基础设施中获得多少收益，主要取决于您把它当做应用层协议用得有多好。然而，包括SOAP和一些Ajax Web框架在内的不少技术都将HTTP作为一种传输信息的协议，这种用法很难充分利用HTTP层的基础设施。本章包含以下内容，着重介绍了将HTTP用做应用协议的几个方面：

1.1 节，“如何保持交互的可见性”

可见性是 HTTP 的关键特征之一，可通过本节了解如何保持可见性。

1.2 节，“何时使用 GET 方法”

通过本节了解何时使用 GET 方法。

1.3 节，“何时使用 POST 方法”

通过本节学习何时使用 POST 方法。

1.4 节，“如何使用 POST 方法创建资源”

通过本节了解如何使用 POST 方法创建新的资源。

1.5 节，“何时使用 PUT 方法创建新资源”

您可以使用 POST 方法或 PUT 方法创建新资源。本节将讨论什么情况下更适合使用 PUT 方法。

注 1： 本书将严格区分单独使用的“书将严格区分单独使与“书将严格区分单独使用，前者译为“转移”，后者译为“传输”。

1.1 如何保持交互的可见性

作为应用协议，HTTP 的设计目标是在客户端和服务器之间保持对库、服务器、代理、缓存和其他工具的可见性。可见性是 HTTP 的一个核心特征。按 Roy Fielding 的定义，可见性是“一个组件能够对其他两个组件之间的交互进行监视或仲裁的能力。”当协议是可见的时，缓存、代理、防火墙等组件就可以监视甚至参与其中。

问题描述

您想知道可见性的含义，以及如何保持 HTTP 请求和响应的可见性。

解决方案

一旦您识别并设计资源，就可以使用 GET 方法获取资源的表述，使用 PUT 方法更新资源，使用 DELETE 方法删除资源，以及使用 POST 方法执行各种不安全和非幂等的操作。可以添加适当的 HTTP 标头来描述请求和响应。

问题讨论

以下特性完全取决于保持请求和响应的可见性：

缓存

缓存响应内容，并在资源修改时使缓存自动失效。

乐观并发控制

检测并发写入，并在操作过期的表述时防止资源发生变更。

内容协商

在给定资源的多个可用表述中，选择合适的表述。

安全性和幂等性

确保客户端可以重复或重试特定的 HTTP 请求。

当一个 Web 服务无法保持可见性时，以上这些功能将无法正常工作。例如，当服务器对 HTTP 的使用方式阻碍乐观并发时，您可能要被迫自己实现应用特定的并发控制机制。

保持可见性让您可以使用现有的 HTTP 软件和基础设施来实现之前必须自己实现的功能。

HTTP 通过以下途径来实现可见性：

- HTTP 的交互是无状态的，任何 HTTP 中介都可以推断出给定请求和响应的意义，而无须关联过去或将来的请求和响应。
- HTTP 使用一个统一接口，包括有 OPTIONS，GET，HEAD，POST，DELETE 和 TRACE 方法。接口中的每一个方法操作一个且仅有一个资源。每个方法的语法和含义不会因应用程序或资源的不同而发生改变。这就是为什么 HTTP 以统一接口而闻名于世了。
- HTTP 使用一种与 MIME 类似的信封格式进行表述编码。这种格式明确区分标头和内容。标头是可见的，除了创建、处理消息的部分，软件的其他部分都可以不用关心消息的内容。

考虑一个更新资源的 HTTP 请求：

```
# 请求
PUT /movie/gone_with_the_wind HTTP/1.1 ❶
Host: www.example.org ❷
Content-Type: application/x-www-form-urlencoded

summary=...&rating=5&... ❸

# 响应
HTTP/1.1 200 OK ❹
Content-Type: text/html;charset=UTF-8 ❺
Content-Length: ...
<html> ❻
 ...
</html>
```

❶ 请求行包含 HTTP 方法、资源路径和 HTTP 版本

❷ 请求的表述形式标头

❸ 请求的表述内容

❹ 响应状态行包含 HTTP 版本、状态码和状态消息

❺ 响应的表述形式标头

❻ 响应的表述内容

这个例子的请求是一个 HTTP 消息。消息的第一行描述了客户端所使用的协议和方法，接下来的两行是请求头。简单查看这三行信息，任何理解 HTTP 的软件都可以明白请求的意图以及如何解析消息体。响应也是同样的，响应的第一行表示 HTTP 版本、状态码和状态消息，接下来的两行告诉 HTTP 软件如何解释消息。

对于 RESTful Web 服务，您的主要目标必定是尽最大可能保持可见性。保持可见性非常简单，使用 HTTP 方法时，其语义要与 HTTP 所规定的语义保持一致，并添加适当的标头来描述请求和响应。

保持可见性的另一方面是使用适当的状态码和状态消息，以便代理、缓存和客户端可以决定请求的结果。状态码是一个整数，状态消息是文本。

在某些情况下，您可能需要权衡其他特性，如网络效率、客户端的便利性以及分离关注点，为此放弃可见性。当您进行这种权衡时，应仔细分析对缓存、幂等性、安全性等特性的影响。

1.2 何时使用 GET 方法

Web 基础设施严重依赖于 GET 方法的幂等性和安全性。客户端期望能够重复发起 GET 请求，而不必担心造成副作用。缓存依赖于不需访问源服务器便能提供已缓存表述的能力。

问题描述

您想知道何时应该与何时不应该使用 GET 请求，以及 GET 请求使用不当的潜在后果。

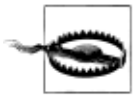
解决方案

使用 GET 方法进行安全与幂等的信息获取。

问题讨论

每个 HTTP 方法都具有特定的语义。正如 1.1 节所讨论的，GET 的目的是得到一个资源的表述，PUT 用于建立或更新一个资源，DELETE 用于删除一个资源，POST 用于创建多个新资源或对资源进行多种其他变更。

在所有上述方法中，GET 被滥用的情况最少，因为 GET 既安全又幂等。



不要把 GET 方法用于不安全或非幂等操作。因为这样做可能会造成永久性的、意想不到的、不符合需要的资源改变。

大部分对 GET 的滥用都是将它用在不安全操作上，以下是一些例子：

```
# 将页面存为书签
GET /bookmarks/add_bookmark?href=http%3A%2F%2F
    www.example.org%2F2009%2F10%2F10%2Fnotes.html HTTP/1.1
Host: www.example.org

# 向购物车添加内容
GET /add_cart?pid=1234 HTTP/1.1
Host: www.example.org

# 发送消息
GET /messages/send?message=I%20am%20reading HTTP/1.1
Host: www.example.org

# 删除便条
GET /notes/delete?id=1234 HTTP/1.1
Host: www.example.org
```

对于服务器来说，所有这些操作都是不安全和非幂等的。但对于那些基于 HTTP 的软件，这些操作都是安全和幂等的。这种差异的后果严重依赖于应用程序。例如，一个工具在服务器上通过定期提交一个 GET 请求来执行健康检查，如果使用上面第 4 个 URI，则将删除一条记录。

如果这些操作必须要使用 GET 方法，特别警惕以下几点：

- 添加 Cache-Control: no-cache 头来确保响应不被缓存。
- 确保由此产生的任何副作用都是良性的，不会改变关键业务数据。
- 在服务器实现方面，将这些操作实现成可重复执行的（例如，幂等的）。

上述要点可以帮助减少某些错误的操作（不是全部）导致的后果，但最佳的措施是避免 GET 方法的滥用。

1.3 何时使用 POST 方法

本节简要介绍 POST 的不同应用场合。

问题描述

您想知道 POST 方法的潜在应用场合。

解决方案

在以下场合中使用 POST 方法：

- 创建新的资源，把资源作为一个工厂，详见 1.4 节。
- 通过一个控制器资源来修改一个或多个资源。
- 执行需要大数据输入的查询，详见 5.3 节。
- 在其他 HTTP 方法看上去不合适时，执行不安全或非幂等的操作。

问题讨论

在 HTTP 协议中，POST 方法的语义是最通用的，HTTP 规范定义这个方法可以应用于以下场合：

- 对已存在资源做注解。
- 向公告板、新闻组、邮件列表或类似的文章群组发送消息。
- 提供数据块，例如，作为表单提交到数据处理器处理后的结果。
- 通过追加操作扩充数据库。

所有这些操作都是不安全和非幂等的，所有基于 HTTP 的工具都会这样对待 POST，例如：

- 缓存不会缓存这一方法的响应。
- 网络爬虫和类似的工具不会自动发起 POST 请求。
- 大部分通用的 HTTP 工具不会自动重复提交 POST 请求。

这些处理方式给服务器留下了巨大的空间，将POST作为针对多种操作的一个通用方法，包括穿隧（tunneling）[注2]，考虑以下例子：

```
# 一条穿隧了HTTP POST的XML RPC消息
POST /RPC2 HTTP/1.1
Host: www.example.org
Content-Type: text/xml;charset=UTF-8
```

注 2： 关于穿隧，请参考 Wikipedia，http://en.wikipedia.org/wiki/Tunneling_protocol。

```
<methodCall>
  <methodName>messages.delete</methodName>
  <params>
    <param>
      <value><int>1234</int></value>
    </param>
  </params>
</methodCall>
```

这是一个XML-RPC（http://www.xmlrpc.com/）通过POST方法穿隧操作的例子。另一个知名的例子是HTTP上的SOAP：

```
# 一条穿隧了HTTP POST的SOAP消息
POST /Messages HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=UTF-8

<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

  <soap:Body xmlns:ns="http://www.example.org/messages">
    <ns:DeleteMessage>
      <ns:MessageId>1234</ns:MessageId>
    </ns:DeleteMessage>
  </soap:Body>
</soap:Envelope>
```

这两个例子都是对 POST 方法的误用。例如，DELETE 方法在此更为适用：

```
# 使用 DELETE
DELETE /message/1234 HTTP/1.1
Host: www.example.org
```

当应用程序操作和 HTTP 协议上没有这样的直接映射关系时，使用 POST 相对其他 HTTP 方法而言，产生严重后果的概率更小。

另外，以下情况必须使用 POST，即使 GET 才是正确的方法：

- 浏览器等 HTML 客户端在发起请求获取相关资源时，将页面的 URI 作为 Referer 头。这可能会把包含在 URI 中的敏感信息泄露给外部服务器。

  这种情况下，使用传输层安全协议（Transport Layer Security，TLS，SSL 的接班人），或者，如果不能加密 URI 中的敏感信息，考虑使用 POST 处理 HTML 文档。

- 正如将在 5.3 节讨论的，当客户端提交的查询包含太多参数时，POST 可能是唯一的选择。

但即使是以上情况，POST 也应当是最后的选择。

1.4 如何使用 POST 方法创建资源

POST 方法的应用场合之一是创建新资源，该协议类似于使用“工厂方法模式”创建新对象。

问题描述

您想知道如何创建新资源，请求中需要包含什么内容，以及响应中应该包括什么内容。

解决方案

将一个已存在的资源标识为创建新资源的工厂。虽然您可以把任意资源用做工厂，但常见的做法是使用一个集合资源（详见 2.3 节）。

让客户端向工厂资源提交附有需要创建资源的表述的 POST 请求。通过可选支持的 Slug 头，客户端可以向服务器建议一个名字，作为被创建资源的 URI 的一部分。

资源创建之后，返回响应码 201（Created），并在 Location 头中包含新创建资源的 URI。

如果响应正文包含了新创建资源的完整表述，那么在 Content-Location 头中包含新创建资源的 URI。

问题讨论

考虑一个为用户创建“地址”资源的例子，您可以把“用户”资源作为一个创建新“地址”的工厂：

```
# 请求
POST /user/smith HTTP/1.1 ❶
Host: www.example.org
Content-Type: application/xml;charset=UTF-8
Slug: Home Address ❷

<address>
  <street>1, Main Street</street>
  <city>Some City</city>
</address>

# 响应
HTTP/1.1 201 Created
Location: http://www.example.org/user/smith/address/home_address ❸
Content-Location: http://www.example.org/user/smith/address/home_address ❹
Content-Type: application/xml;charset=UTF-8
```

```
<address>
  <id>urn:example:user:smith:address:1</id>
  <atom:link rel="self"
      href="http://www.example.org/user/smith/ address/home_
      address"/>
  <street>1, Main Street</stret>
  <city>Some City</city>
</address>
```

❶ 使用“用户”资源作为创建“家庭地址”资源的工厂

❷ 对新资源 URI 命名的建议

❸ 新创建资源的 URI

❹ 响应中表述的 URI

在这个例子里，请求包含了需要创建的新资源中的数据，并在 Slug 头中包含了新资源 URI 的建议名称。请注意，Slug 头是由 AtomPub（RFC5023）规定的，它只是来自客户端的建议，服务器端并不一定要遵循它。可以阅读第 6 章详细了解 AtomPub。

响应中的状态码 201 表明服务器已创建了一个新资源，并在 Location 响应头中为其指定了 URI 中为 http://www.example.org/user/smith/address/home_address。Content- Location 头告诉客户端表述内容也可以通过这一 URL 获取。

虽然使用了 Content-Location 头信息，您也可以包含新创建资源的 Last-Modified 和 ETag 头信息。

1.5 何时使用 PUT 方法创建新资源

您可以使用 HTTP POST 方法或 HTTP PUT 方法创建新资源。本节讨论何时使用 PUT 方法创建新的资源。

问题描述

您想知道何时使用 PUT 方法创建新资源。

解决方案

只有在客户端可以决定资源的 URI 时才使用 PUT 方法创建新资源，否则，使用 POST。

问题讨论

下面是一个客户端使用 PUT 方法创建新资源的例子：

```
# 请求
PUT /user/smith/address/home_address HTTP/1.1 ❶
Host: www.example.org
Content-Type: application/xml;charset=UTF-8

<address>
  <street>1, Main Street</street>
  <city>Some City</city>
</address>

# 响应
HTTP/1.1 201 Created
Location: http://www.example.org/user/smith/address/home_address
Content-Location: http://www.example.org/user/smith/address/home_address
Content-Type: application/xml;charset=UTF-8

<address>
  <id>urn:example:user:smith:address:1</id>
  <atom:link rel="self" href="http://www.example.org/user/smith/address/
     home_address"/>
  <street>1, Main Street</street>
  <city>Some City</city>
</address>
```

❶ 客户端使用 PUT 方法创建新的资源

仅在客户端可以控制 URI 的构成时，才使用 PUT 方法创建新资源。举个例子，一台存储服务器可能为每个客户端分配一个根 URI，并让客户端把根 URI 作为文件系统中的根目录，以便创建新资源。如果不能控制 URI，请使用 POST 方法。

当使用 POST 方法创建新资源时，服务器会决定新创建资源的 URI，这样可以保证 URI 的命名规则符合任意网络安全级别的配置。在为新资源生成 URI 时，您仍然可以让服务器使用表述中的信息（如 Slug 头）。

当您支持使用 PUT 创建新资源时，客户端必须可以为资源指定 URI。使用 PUT 方法创建新资源时，要考虑以下要点：

- 为了让客户端可以指定 URI，服务器需要向客户端解释 URI 在服务器中是如何组织的，什么样的 URI 是合法的，以及什么样的 URI 是非法的。

- 您还要顾及所有设置于服务器端的基于 URI 模式的安全和过滤规则，这时您可能希望客户端在创建新资源时使用范围更小的 URI。

通常情况下，任何使用 PUT 方法创建的资源，同样都可以在资源工厂中使用 POST 方法来创建。使用工厂资源可以给服务器更多的控制权，而不需要解释其命名规则。一个例外是服务器为客户端提供类文件系统的接口来管理文档。WebDAV 就是这样一个例子。

第 2 章

识别资源

开发 RESTful Web 服务的首要步骤之一就是设计资源模型。资源模型对所有客户端用来与服务器交互的资源加以识别和分类。在设计 RESTful Web 服务的所有工作，如资源的识别、媒体类型和格式的选择以及统一接口的应用中，资源的识别是最灵活的部分。

由于HTTP的可见性（详见 1.1 节），您可以使用诸如Firebug （http://getfirebug.com）、Yahoo!YSlow(http://developer.yahoo.com/yslow/)或Resource Expert Droid(http://redbot.org) 这样的工具来验证服务器是否提供了正确的HTTP响应。但您无法对资源进行同样的验证。因为，资源模型并无所谓正确与否。重要的是您能否正确地使用HTTP的统一接口来实现您的Web服务。本章将通过以下内容帮助您在大部分情况下识别资源：

2.1 节，“如何从领域名词中识别资源”

使用本节的内容从领域实体中识别初始的资源集合。

2.2 节，“如何选择资源粒度”

使用本节的内容帮助确定资源粒度。

2.3 节，“如何将资源组织为集合”

当您有多个同类型的资源时，可以将这些资源组合为集合资源。

2.4 节，“何时将资源合并为复合资源”

根据客户端的使用模式，可以将资源合并为复合资源。

设计资源模型通常是一个迭代的过程。在开发 Web 服务时，需要考虑后端的设计约束和来自其他用例的客户需求，再来回顾这些内容，迭代式地改善资源的设计。

2.1 如何从领域名词中识别资源

面向对象设计和数据库建模技术都把领域实体作为设计的基础。您可以使用同样的技术来识别资源。但要当心，正如您将在本章后面看到的那样，本节的内容过于简单，在某些情况下，可能会产生误导。

问题描述

您想要从用例和 Web 服务的描述中识别资源。

解决方案

分析您的用例，找到可以用“创建”、“读取”、“更新”或“删除”动作来操作的领域名词。将每个领域名词都标识为资源。使用 POST，GET，PUT 和 DELETE 方法分别为每个资源实现“创建”、“读取”、“更新”和“删除”操作。

问题讨论

设想一个管理照片的 Web 服务。客户端可以上传新照片、替换现有的照片、查看或删除照片。在这个例子中，“照片”是一个应用领域中的实体。客户端可以对这个实体执行的动作包括“创建一张新照片”、“替换现有的照片”、“查看照片”和“删除照片”。

您可以运用本节的内容，将“照片”识别为资源，这样客户端就可以像下面这样，使用 HTTP 的统一接口来操作这些照片：

- GET 方法获得每张照片的表述。
- PUT 方法更新照片。
- DELETE 方法删除照片。
- POST 方法创建一张新照片。

本节容易给人留下一个印象，即 REST 只适合 CRUD 风格（创建、读取、更新、删除）的应用。如果您仅限于通过领域名词来识别资源，可能会发现，那些一成不变的 HTTP 方法存在局限性。在大多数应用程序中，CRUD 操作只是接口的一部分。考虑以下例子：

- 寻找从西雅图到旧金山的交通指示。
- 生成随机数，或把英里转换为千米。
- 为客户端提供一个方法，在一个请求中，获取最小属性集的用户档案，列出最近 10 张

用户上传的照片，以及用户感兴趣的 10 条新闻。

- 批准购买软件的申请。
- 将钱从一个银行账户转到另一个银行账户。
- 合并两个地址簿。

所有这些用例中，您都可以很容易地指出名词。但在每种情况下，如果将这些名词识别为资源，会发现相应的动作没有办法映射到 GET，POST，PUT 和 DELETE 这样的 HTTP 方法上。需要额外的资源来处理这些用例。请参考本章的其他内容来确定这些额外的资源。

2.2 如何选择资源粒度

直接将领域实体映射为资源可能导致资源效率低下且难以使用，您可以使用本节中讨论的标准来确定合适的资源粒度。

问题描述

您想知道确定合适资源粒度的标准。

解决方案

可以通过网络效率、表述的多少以及客户端的易用程度来帮助确定资源的粒度。

问题讨论

看看您的应用场合，会发现多个不同粒度下的名词。例如，在一个社交网络中，交互发生在“用户”上下文中。每个用户的数据可能包括活动流、好友列表、关注者列表、共享链接等。在这样一个应用程序中，您是将用户建模成一个粗粒度的资源，封装所有这些用户数据，还是应该采用较小的粒度，提供活动流、好友、关注者等资源？答案取决于 Web 服务需要为哪些典型的客户端提供服务。前一种方法，客户端需要处理的信息量可能过大，而后一种方法可能更加灵活。如果大部分客户端是将用户数据下载并存储到本地计算机中，然后使用富用户界面进行展现，那么提供包含所有数据的用户资源是有意义的。

再举一个更简单的例子，一个带地址的用户。您可能想使用一个代理 HTTP 缓存，在其内存中保存所有用户的表述，这样客户端可以快速访问这些内容。在这个例子中，包含有地址的用户资源的表述可能太大了，不适合放入缓存。虽然减小了粒度使客户端与服务器的交互更加频繁了，但将每个用户的地址作为单独的资源更有意义。

同样的，直接把应用程序的数据库表或对象模型映射为资源，也不一定会产生最好的结果。

一些因素会影响数据库表和对象模型的设计，例如领域建模、需要高效数据访问和处理。而另一方面，HTTP 客户端要通过网络使用 HTTP 的统一接口获取资源。因此，应该以适合客户端使用模式的方式来设计资源，而不是基于现有的数据库或对象模型。

那么，您应当如何确定哪些名词是候选资源呢？如何确定合适的粒度？回答这些问题最好的方式是从客户端的角度思考问题。之前的第一个例子表明，粗粒度设计便于富客户端应用程序，而在第二个例子中，更精细的资源粒度可以更好地满足缓存的要求。因此，应从客户端和网络的角度确定资源的粒度。下列因素可能会进一步影响资源粒度：

- 可缓存性
- 修改频率
- 可变性

仔细设计资源粒度，以确保使用更多缓存，减小修改频率，或将不可变数据从使用缓存较少、修改频率更高或可变数据分离出来，这样可以改善客户端和服务器端的效率。

2.3 如何将资源组织为集合

将资源组织为集合，可以让客户端及服务器将一组资源视为一个资源来引用，在集合上执行查询，甚至将集合作为工厂来创建新资源。

问题描述

您想知道如何以最佳的方式将那些有共性的资源组织为集合。

解决方案

基于应用程序特有的条件来识别相似的资源。常见的例子有，共享同一数据库 Schema 的资源、有相同特性（attribute）或属性（property）的资源或客户端看起来是相似的资源。

为每组集合设计一个表述，这样它便可以包含集合中所有成员或某些成员的信息了（详见 3.7 节）。

问题讨论

一旦将多个相似的资源归集到一个集合资源下，您便可以像一个整体那样来进行引用，就像下面的例子所展示的那样。例如，您可以提交一个 GET 请求来获取整个集合，而不是一个个地获取单个资源。

设想一个社交网络，所有的用户记录共享同一个数据库 Schema。网络中的每个用户都有一个好友列表和关注者列表。好友和关注者是同一数据库中的其他用户。用户是基于个人兴趣进行分类的，例如跑步、骑自行车、游泳、徒步旅行等。在这个例子里，您可以识别出以下集合，它们的成员都是用户资源：

- 用户资源集合
- 任意给定用户的好友集合
- 给定用户的关注者集合
- 拥有相同兴趣的用户集合

以下是针对一个用户集合的 GET 请求的响应示例：

```
# 请求
GET /users HTTP/1.1
Host: www.example.org

# 响应
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

<users xmlns:atom="http://www.w3.org/2005/Atom"> ❶
  <atom:link rel="self" href="http://www.example.org/users"/>
  <user> ❷
    <id>urn:example:user:001</id>
    <atom:link rel="self" href="http://www.example.org/user/user001"/>
    <name>John Doe</name>
    <email>john.doe@example.org</email>
  </user>
  <user>
    <id>urn:example:user:002</id>
    <atom:link rel="self" href="http://www.example.org/user/user002"/>
    <name>Jane Doe</name>
    <email>jane.doe@example.org</email>
  </user>
  ...
</users>
```

❶ 一个集合资源

❷ 集合中的一个成员

请注意，集合并不一定都要是分层的。一个给定的资源可以是多个集合资源的一部分。例如，

一个用户资源可能是“用户集合”、“好友集合”、“关注者集合”和“徒步旅行集合”这多个集合的一部分。以下是一个用户的好友集合：

```
# 请求
GET /user/user001/friends HTTP/1.1
Host: www.example.org

# 响应
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

<users xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="self" href="http://www.example.org/user/user001/friends"/>
  <user>
    <id>urn:example:user:002</id>
    <atom:link rel="self" href="http://www.example.org/user/user002"/>
    <name>Jane Doe</name>
    <email>jane.doe@example.org</email>
  </user>
  ...
</users>
```

您可以像下面这样来使用集合资源：

- 获取集合的分页视图，例如浏览一个用户的好友集合，一次 10 条（详见 3.7 节）。
- 搜索集合成员，或者获取集合的一个过滤视图。例如，可以查询游泳者的好友（详见 5.2 节）。
- 将集合作为一个工厂，通过向集合资源提交 HTTP POST 请求来创建新成员资源。
- 一次向多个资源执行相同操作。

2.4 何时将资源合并为复合资源

在访问像 http://www.yahoo.com 或 http://www.msn.com这样的网站主页时，您会注意到这些页面会从多个信息源（例如新闻、电子邮件、天气预报、娱乐信息、财经信息等）聚合信息。如果将单个信息源视为一个资源，为每个主页提供服务就是将这些不同的资源合并为一个单独资源的结果，这个资源的表述形式是一个HTML页面。这样的Web页面就是复合资源（composite resource），也就是说，它们可以对其他资源进行合并。本节将使用相同的技术来识别复合资源。

问题描述

您想知道如何提供一个状态是由两个或更多资源的状态组合而成的资源。

解决方案

基于客户端的使用模式、性能和延时要求，确定一些新的资源，它们通过聚合其他资源来减少客户端与服务器的交互。

问题讨论

复合资源会从其他资源那里组合信息。设想在企业应用程序中为每个客户提供一个快照页，该页将显示客户信息，例如姓名、联系方式、该客户最近的购买订单汇总以及所有待决定的报价请求。通过学习到目前为止本章中讨论的内容，您可以识别出以下资源：

- 带有姓名、联系方式和其他详细信息的客户资源
- 每个客户的购买订单集合
- 每个客户的待决定报价集合

有了这些资源，您可以发起以下 GET 请求，并使用其响应来构建一个客户的快照页：

```
# 获取客户数据
GET /customer/1234 HTTP/1.1
Host: www.example.org

# 获取最近10个购买订单
GET /orders?customerid=1234&sortby=date_desc&limit=10 HTTP/1.1
Host: www.example.org

# 获取最近10个待决定报价
GET         /quotes?customerid=1234&sortby=date_desc&status=pending&limit=10
HTTP/1.1
Host: www.example.org
```

尽管这一系列 GET 请求能被服务器所接受，但它们太过频繁。对客户端而言，如果能只发送一条单一的网络请求来获取呈现页面所需的所有数据，可能会更高效一些。

针对客户快照页，可以设计一个“客户快照”复合资源，其中囊括了客户端展现页面所需的所有信息。分配这样一个 URI 形式，http://www.example.org/customer/1234/snapshot，其中，1234 是标识一个客户的标识符。下面是使用该资源的一个范例：

```
# 请求
```

```
GET /customer/1234/snapshot HTTP/1.1
Host: www.example.org

# 响应
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

<snapshot xmlns:atom="http://www.w3.org/2005/Atom">
  <!-- 客户信息 -->
  <customer>
    <id>1234</id>
    <atom:link rel="self" href="http://www.example.org/customer/1234">
    <name>...</name>
    <address>...</address>
  </customer>
  <!-- 客户最近的10个订单 -->
  <orders>
    <atom:link rel="http://www.example.org/rels/orders/recent"
         href="http://www.example.org/orders?customerid=1234&sortby
            =date_desc&"/>
    <order>
      <id>...</id>
      ...
    </order>
    ...
  </orders>
  <!-- 客户最近的10个待决定报价 -->
  <quotes>
    <atom:link rel="http://www.example.org/rels/quotes/recent"
         href="http://www.example.org/quotes?customerid=1234&sortby
            =date_desc&"/>
    ...
  </quotes>
</snapshot>
```

这个响应是一组表述的聚合，客户端可以提交三个 GET 请求来获得它们。

复合资源降低了统一接口的可见性，因为它们的表述中包含了和其他资源相重叠的数据。因此，在提供复合资源前，请考虑以下几点：

- 如果在应用程序中对复合资源的请求很少，那么它可能不是一个好的选择。依赖缓存代理，从缓存中获取这些资源，也许能让客户端受益匪浅。
- 另一个因素是网络开销——客户端与服务器之间的网络开销，服务器和后端服务或它所依赖的数据存储之间的网络开销。如果后者开销很大，那获取大量数据并在服务器上将

它们组合成复合资源可能会增加客户端的延时，降低服务器的吞吐量。

- 在本例中，想要改善延时，可以在客户端和服务器之间增加一个缓存层，并避免复合资源。进行一些负载测试来验证复合资源是否能起到改善作用。

最后，为每个客户端创建特定目标的复合资源并非是注重实效的做法。选择对您的 Web 服务最重要的客户端，设计复合资源来满足它们的需要。

第 3 章

设计表述

客户端所关心的资源是一个抽象的实体，它是用 URI 来标识的。另一方面，表述是具体而真实的，您在客户端和服务器上针对它编写代码，进行操作。

回想一下 1.1 节，HTTP 在请求和响应中为表述提供了一种包装格式。设计表述涉及(a)使用 HTTP 提供的格式包含正确的标头，(b)当表述有正文时，为正文选择合适的媒体类型并设计一种格式。本章的内容涵盖了表述设计的多个方面：

3.1 节，“如何使用实体头来注解表述”

使用本节的内容来决定发送表述时包含哪个实体头（entity header）。

3.2 节，“如何解释实体头”

使用本节的内容来决定如何从接收到的表述中解释出实体头。

3.3 节，“如何避免字符编码不匹配”

通过本节的内容了解一些与字符编码不匹配相关的预防措施。

3.4 节，“如何选择表述格式和媒体类型”

使用本节找到选择表述格式和媒体类型的判断标准。

3.5 节，“如何设计 XML 表述”

使用本节的内容来决定 XML 格式表述的基本要素。

3.6 节，“如何设计 JSON 表述”

通过本节了解如何设计 JSON 格式的表述。

3.7 节，“如何设计集合表述”

通过本节了解设计集合表述的一些惯例。

3.8 节，“何时以及如何提供 HTML 表述”

当您希望开发者或最终用户浏览某些资源时，那些资源要支持 HTML 格式。

3.9 节，“如何返回错误”

错误也是表述，只是它们反映了资源的错误状态。通过本节了解如何返回错误响应。

3.10 节，“如何在客户端处理错误”

通过本节了解如何在客户端处理错误。

3.1 如何使用实体头来注解表述

表述不仅仅是以某种格式序列化后的数据，它是一连串字节加上用于描述那些字节的元数据。在 HTTP 中，表述元数据是由使用实体头的名值对（name-value pair）来实现的。这些实体头和应用数据本身一样重要。它们可保证可见性、可发现性、通过代理路由、缓存、乐观并发性，以及以应用协议的方式正确进行 HTTP 操作。

问题描述

您想知道在对服务器的请求或对客户端的响应中应该发送哪些 HTTP 头。

解决方案

使用以下标头来注解包含消息正文的表述：

- Content-Type，用于描述表述类型，包含 charset 参数或其他针对该媒体类型而定义的参数。
- Content-Length，用于指定表述正文的字节大小。
- Content-Language，如果您以某种语言对表述进行本地化，用该标头来指定语言。
- Content-MD5，工具/软件在处理或存储表述时可能存在错误，需要提供一致性校验，用该标头来包含一个表述正文的 MD5 摘要。请注意，TCP 使用 checksum 在传输层提供一致性校验。
- Content-Encoding，当您使用 gzip，compress 或 deflate 对表述正文进行编码时，使用该标头。
- Last-Modified，用来说明服务器修改表述或资源的最后时间。

问题讨论

HTTP 的设计是这样的，发送方可以用一系列名为实体头的标头来描述表述正文（也称为实体正文或消息正文）。有了这些标头，接收方可以在无须查看正文的情况下决定如何处理正文。它们还可以将解析正文所需要提前了解及猜测的内容减到最小程度。

下面是一个经过注解的表述：

```
Content-Type: application/xml;charset=UTF-8
Content-Language: en-US
Content-MD5: bbdc7bbb8ea5a689666e33ac922c0f83
```

```
Last-Modified: Sun, 29 Mar 2009 04:51:38 GMT

<user xmlns:atom="http://www.w3.org/2005/Atom">
  <id>user001</id>
  <atom:link rel="self" href="http://example.org/user/user001"/>
  <name>John Doe</name>
  <email>john@example.org</email>
</user>
```

让我们仔细看看每个标头。

Content-Type

这个标头描述了表述的“类型”，就是通常所说的 *media-type* 或 *MIME* 类型，例如 text/html，image/png，application/xml 和 text/plain。这些都是用来编码表述正文的格式的标识符。概括来说，格式就是将信息编码进某些媒体（例如文件、磁盘或网络）的方法。XML、JSON、文本、CSV、PDF 等都是格式。媒体类型标识了所使用的格式，描述了如何解释表述正文的语义。application/xml，application/ json，text/plain，text/csv，application/pdf 等都是媒体类型。

这个标头告诉接收方如何解析数据。举例来说，如果标头的值是 application/xml 或其他以+xml 结尾的值，就可以用 XML 解析器来解析消息。如果是 application/json，就可以用 JSON 解析器。没有此标头时，就只能猜测正文的格式了。

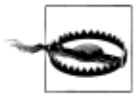
Content-Length

这个标头最早是在HTTP 1.0 中被引入的，接收方用它来判断自己是否从连接中读取了正确的字节数。要发送该标头，发送方需要在写正文前计算出表述的大小。HTTP 1.1 支持一种更有效的机制，名为分块转移编码（chuncked transfer encoding）[注1]，这让Content-Length头变得有点多余。下面是一个使用分块编码的表述：

```
HTTP/1.1 200 OK
Last-Modified: Thu, 02 Apr 2009 02:32:28 GMT
Content-Type: application/xml;charset=UTF-8
Transfer-Encoding: chunked

FF
[放置一些字节]
58
[放置一些字节]
0
```

注 1： chuncked transfer encoding，资料中译做“分块传输编码”，在 HTTP 中似乎应译为分块转移编码更为合适。详见维基百科关于这个编码的说明，http://en.wikipedia.org/wiki/Chunked_transfer_encoding。

如果客户端不支持 HTTP 1.1，请包含 Content-Length。



对于 POST 和 PUT 请求，就算使用了 Transfer-Encoding: chuncked，也要在客户端应用程序的请求中包含 Content-Length 头。因为有些代理会拒绝没有包含这两个头的 POST 和 PUT 请求。

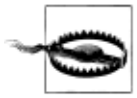
Content-Language

当表述针对某种语言做了本地化之后，请使用该标头，它的值是两个字母的 RFC 5646 语言标签，还可以在后面带上连字符（-）和任意两个字母的国家代码。下面是一个范例：

```
# 响应
HTTP/1.1 200 OK
Content-Language: kr

<address type="work">
   <street-address>강남구 삼성동 144-19,20 번지 JS 타워</street-address>
  <locality>서울특별시</locality>
  <postal-code>135-090</postal-code>
  <country-name>대한민국</country-name>
  <country-code>KR</country-code>
</address>
```

Content-MD5

接收方可以使用该标头来验证实体正文的完整性。该标头的值是表述正文的 MD5 摘要，在进行内容编码（gzip，compress 等）之后，转移编码（即 chunked）之前计算摘要值。



因为这个标头不能保证消息没有被篡改，所以不要将它作为一种安全手段。修改了正文的人同样可以修改标头的值。

在通过非可靠网络发送或接受大的表述时，这个标头非常有用。表述的发送方包含了 Content-MD5 头之后，接收方可以在解析前先验证消息的完整性。

Content-Encoding

这个标头说明了表述正文所使用的压缩类型，它的值可以是类似 gzip，compress 或 deflate 这样的字符串。下面是一个 gzip 编码的表述：

```
Content-Type: application/xml;charset=UTF-8
Content-Language: en-US
```

```
Content-MD5: b7c50feb215b112d3335ad0bd3dd88c1
Content-Encoding: gzip
Last-Modified: Sun, 29 Mar 2009 04:51:38 GMT

... gzip编码后的字节 ...
```

接收方在解析正文前需要先解压消息。客户端可以用 Accept-Encoding 头来注明自己偏好的 Content-Encoding。然而，并没有一个标准的方式让客户端了解到服务器是否可以处理用给定编码压缩过的表述。

除非您事先知道目标服务器支持某个特定的编码方法，否则应该避免在 HTTP 请求中使用这个标头。

Last-Modified

这个标头只用在响应上，它的值是一个时间戳，表示服务器最后修改资源表述的时间。

3.2 如何解释实体头

当服务器或客户端接收到表述时，在处理请求前正确地解释实体头是很重要的。本节讨论了如何从所包含的标头中解释表述。

问题描述

您想知道如何解释表述中所包含的实体头，以及如何用那些标头来处理表述。

解决方案

Content-Type

当您收到一个不带 Content-Type 的表述时，避免猜测表述的类型。当客户端发送不带该标头的请求时，返回错误码 400（Bad Request）。当您从服务器接收到一个不带该标头的响应时，将其视为不正确的响应。

Content-Length

在没有确定接收到的表述不带 Transfer-Encoding: chunked 前，不要检查 Content-Length 头是否存在。

Content-Encoding

让您的网络库代码来解压那些压缩过的表述。

Content-Language

如果存在该标头，读取并存储它的值，记录下所使用的语言。

问题讨论

大多数情况下，客户端应用程序只需检查 Content-Type 头和字符编码，以此决定如何解析表述的正文。客户端 HTTP 库必须要能透明地处理 Content-Encoding。

一些应用程序假设 Content-Length 头总是会出现在表述里，并拒绝那些不包含该标头的表述。这个假设并不正确。如果您的代码在处理请求或响应前必须确定消息长度，遵循 RFC 2616 的 4.4 节中概述的过程。

一定要基于 Content-Type，Content-Language 和 Content-Encoding 头的值来处理响应的表述。举例来说，仅仅因为客户端发送了 Accept:application/json 头或资源 URI 以.json 结尾，不能假定响应是 JSON 格式的。

3.3 如何避免字符编码不匹配

表述的发送方和接收方之间的字符编码不匹配通常会造成数据损坏和解析错误。

问题描述

您想知道如何确保表述中的字符能被接收方正确解释。

解决方案

在发送表述时，如果媒体类型允许使用 charset 参数，则包含一个带字符编码值的 charset 参数，该参数值将被用于将字符转为字节。

当您接收到一个表述，其中带有支持 charset 参数的媒体类型，在从表述正文的字节中构造字符流时，使用指定的编码。如果忽略了发送方提供的 charset 值，使用其他的值，那么应用程序很可能会把字符给解释错。

如果收到一个不带 charset 参数的 XML，JSON 或 HTML 表述，让您的 XML，JSON 或 HTML 解

析器通过检查头几个字节来确定字符集，检查的依据就是那些格式的规范中分别给出的算法。

问题讨论

诸如 application/xml，text/html，application/atom+xml 和 text/csv 这样的文本和 XML 媒体类型允许您指定字符编码，通过 Content-Type 头中的 charset 参数，使用该字符编码将字符转换为实体正文中的字节。下面是一个例子：

```
Content-Type: application/xml;charset=UTF-8
```

JSON 媒体类型 application/json 不指定 charset 参数，而是使用 UTF-8 作为默认编码。RFC 4627 规定了确定 JSON 格式数据字符编码的方式。

由于字符编码不匹配造成的错误很难发现。举例来说，当发送方使用 UTF-8 将文本编码为字节，而接收方使用 Windows-1252 编码来解码那些字节时，只要发送方使用的字符在两种编码中拥有相同的代码，您就不会察觉任何问题。例如“Hello World”这个短语在两端都很正常，但“2 €s for an espresso?”这个短语会变成“2 ?Ks for an espresso?”，原因就是编码的不同。

这种不匹配被称为*Mojibake*。http://en.wikipedia.org/wiki/Mojibake中有更多的例子。

另一个引入字符编码不匹配的常见途径是在 XML 表述的 Content-Type 头中给定了一个编码，正文却又给了另一个编码，就像下面这个例子：

```
Content-Type: application/xml; charset=UTF-8 ❶

<?xml version="1.0" encoding="ISO-8859-1"?> ❷
<user> ... </user>
```

❶ 在 Content-Type 头中声明了 UTF-8

❷ 在 XML 文档的开头中声明了 ISO-8859-1

在这种情况下，如果忽略 charset 参数（UTF-8）中提供给 XML 解析器的编码，解析器会尝试根据 XML 的开头决定字符编码，它会找到 ISO-8859-1。这会导致接收方错误地解释正文中的字符。

还要避免针对 XML 格式的表述使用 text/xml 媒体类型。text/xml 的默认字符是 us-ascii，而

application/xml 使用 UTF-8。

3.4 如何选择表述格式和媒体类型

在设计 RESTful Web 服务时，这可能是您脑中冒出来的第一个问题。然而，没有哪种格式能满足所有类型的资源和表述。针对所有表述都选择同一种格式（例如 JSON 或 XML）会降低 HTTP 提供的灵活性。

问题描述

您想知道如何为表述选择格式和媒体类型。

解决方案

保持灵活的媒体类型和格式，每个资源都要可以满足多种应用程序用例和客户端需求。

确定是否有一个标准格式和媒体类型能匹配您的用例。开始查找的最佳位置是Internet Assigned Numbers Authority（IANA，http://www.iana.org/assignments/media-types/）媒体类型登记处。

如果没有标准媒体类型和格式，使用诸如 XML（application/xml），Atom Syndication Format（application/atom+xml）或 JSON（application/json）之类的可扩展格式。

使用 image/png 这样的图片格式或者 application/vnd.ms-excel 或 application/pdf 这样的富文本格式来提供额外的数据表述。使用此类格式时，考虑添加 Content- Disposition 头，比如 Content-Disposition: attachment; filename=<status.xls>提示了文件名，客户端可以用这个文件名将表述保存到文件系统中。

表述要尽量选择众所周知的媒体类型。如果您正设计一个新的媒体类型，按照 RFC 4288 中列出的步骤，将类型和媒体类型注册到 IANA。

问题讨论

HTTP 消息格式允许针对请求和响应使用不同的媒体类型和格式。某些资源可能要求使用 XML 格式的表述，另一些可能要用 HTML 表述，同时其他的资源则要求使用 PDF 格式的表述。类似的，某些资源可以处理 application/x-www-form- urlencoded 的请求但响应中返回 XML 格式的表述。为此类灵活性预留空间是设计表述的重要部分。举例来说，一个管理客户账号的系统可能会需要提供多种媒体类型和格式。

- 每个客户账号有一个 XML 格式的表述

- 一个所有新客户的 Atom Feed
- 用电子表格展现客户趋势
- 用 HTML 页面来展示每个客户的摘要

在格式和媒体类型选择阶段，根据经验来看，最好是依据用例和客户端的类型来做出选择。为此，最好不要选择那种严格要求所有资源都使用一种或两种格式、对使用其他格式完全没有灵活性的开发框架。

使用标准或知名的媒体类型

在为表述选择格式和媒体类型时，先检查一下，是否有标准或知名的格式和媒体类型能匹配您的用例。IANA 媒体类型登记处罗列了主要的媒体类型，例如 text、application，还有子类型，例如 plain，html 和 xml，还提供了对媒体类型和基本格式的额外参考。在 *http://www.iana.org/assignments/media-types/application/*，您会找到 RFC 4627 定义的 application/json 媒体类型。如果决定把 JSON 作为表述的格式，这就是了解该格式语义的文档。表 3-1 列出了一些常用的标准或知名的媒体类型。

表 3-1：知名/标准媒体类型

媒体类型	格式	参考规范
application/xml	通用 XML 格式	RFC 3023
application/*+xml	使用 XML 格式的特殊用途媒体类型	RFC 3023
application/atom+xml	用于 Atom 文档的 XML 格式	RFC 4287 及 RFC 5023
application/json	通用 JSON 格式	RFC 4627
application/javascript	JavaScript，用于可以处理 JavaScript 的客户端	RFC 4329
ın/x-www-ncoded	查询字符串格式	HTML 4.01
application/pdf	PDF	RFC 3778
text/html	多种版本的 HTML	RFC 2854
text/csv	逗号分隔的值，是一种通用格式	RFC 4180

在这张表里，第一列是媒体类型，第二列是该类型所使用的格式。该表中的通用格式没有特定于应用程序的语义。例如，相比一个购买订单资源的 XML 格式表述，针对客户账户资源的 XML 格式表述会有很不同的语义。在本例中，是由服务器来定义这些表述中不同 XML 元素的语义的。

```
# 一个客户表述
Content-Type: application/xml;charset=UTF-8
```

```
<customer>
  <id>urn:example:customer:cust001</id>
  ...
</customer>

# 一个订单的表述
Content-Type: application/xml;charset=UTF-8
<po>
  <id>urn:example:po:po001</id>
  ...
</po>
```

另一方面，诸如 Atom，PNG，HTML 和 PDF 之类的专门格式则拥有具体的语义，这些语义是由所对应的 RFC 或表 3-1 中的其他文档来规定的。以下面的客户 HTML 表述为例：

```
Content-Type: text/html;charset=UTF-8

<html>
<head>
    <title>Customer Xyz</title>
</head>
<body>
...
</body>
</html>
```

HTML 规范描述了这个表述的语义。如果决定使用 XML 或 JSON 这样的通用格式，应该尽可能详细地用文档来说明表述的语义。

引入新的格式和媒体类型

您可以设计全新的文本或二进制格式，带上特定于应用程序的编码、解码规则，并为那些格式分配新的媒体类型。举例来说，可以为针对客户账户资源的 XML 格式分配媒体类型 application/vnd.example.customer+xml。这里的 vnd 表示"vendor"，表示这是一个特定于厂商/实现的媒体类型：

```
# 一个客户表述
Content-Type: application/vnd.example.customer+xml;charset=UTF-8

<customer>
  <id>urn:example:customer:cust001</id>
  ...
```

```
</customer>
```

在这个例子里，通过查看 Content-Type 头，无须解析 XML，任何识别该媒体类型的软件都能识别出这是一个客户账户的表述。下面这两个内容也许会对引入此类新媒体类型有所启发：

新格式

某些情况下，您的应用程序数据可能会很特殊，明显区别于现有的相关媒体类型。相关的例子包括新的音频、视频以及用于编码数据的文档格式或二进制格式。

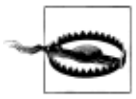
可见性

正如之前的例子，只要媒体类型被广泛支持，特定于应用程序的媒体类型可以提升可见性。如果您选择创建自己的媒体类型，请考虑以下的指导方针：

- 如果媒体类型是基于 XML 的，使用以+xml 结尾的子类型。
- 如果媒体类型是私有的，使用以 vnd.开头的子类型。例如，可以使用诸如 application/vnd.example.org.user+xml 之类的媒体类型。这是一些特定于应用程序的媒体类型所使用的另一个惯例。

如果媒体类型是公共的，按照 RFC 4288 向 IANA 注册您的媒体类型。

请注意，没有被广泛认可的新媒体类型可能会降低与客户端和某些工具的互操作性，这些工具包括代理、日志文件分析器、监控软件等。



除非希望被广泛使用，否则应该避免引入新的特定于应用程序的媒体类型。此类媒体类型的扩散可能会阻碍互操作性。

尽管自定义的媒体类型能改善协议级可见性，但现有的用于监控、过滤、路由 HTTP 流量的协议级工具可能不太关注，甚至于不关注媒体类型。因此，没有必要仅仅为了协议层面的可见性而使用自定义媒体类型。

3.5 如何设计 XML 表述

对于那些特定于应用程序的表述，例如客户档案或购买订单，在表述中包含应用程序数据是理所应当的事情。此外，为了让 Web 服务里的表述互相一致，提升其可用性，有必要在每

个表述中附加特定的详细信息。

问题描述

您想知道在 XML 格式的表述中要包含什么数据。

解决方案

在每个表述中 ,包含一个指向资源本身的 self 链接(即一个带有 self 链接关系类型的链接) , 对于那些组成资源的应用程序领域实体，在表述中要包含它们的标识符。

如果表述中的某个部分包含自然语言文本，请添加 xml:lang 属性，表示元素的内容用的是本地化语言。

问题讨论

在所有的表述中都应该包含诸如标识符和链接这样的常用元素 ,这样客户端和服务器能更方便地处理请求并生成响应。举例来说，self 链接可以让客户端了解表述的 URI，客户端可以用它作为资源的标识符。

当响应包含所请求 URI 上的资源表述时，self 链接和请求 URI 是一样的；或者当响应中的表述与请求 URI 上的资源不一致时，self 链接和 Content-Location 头是一样的。举例来说，在下面的第一个请求中，请求 URI 和表述中响应资源的地址是一致的。第二个请求中，Content-Location 提供了资源的 URI：

```
# 请求
GET /user/smith/address/0 HTTP/1.1
Host: www.example.org

# 响应
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

<address>
  <id>urn:example:user:smith:address:0</id>
  <atom:link rel="self" href="http://www.example.org/user/smith/address/0"/>
  <street>1, Olympia Dr</street>
  <city>Some City</city>
</address>

# 用于创建资源的第一个请求
POST /user/smith HTTP/1.1
```

```
Host: www.example.org
Content-Type: application/xml;charset=UTF-8

<address>
  <street>1, Main Street</stret>
  <city>Some City</city>
</address>

# 响应
HTTP/1.1 201 Created
Location: http://www.example.org/user/smith/address/1
Content-Location: http://www.example.org/user/smith/address/1
Content-Type: application/xml;charset=UTF-8

<address>
  <id>urn:example:user:smith:address:1</id>
  <atom:link rel="self" href="http://www.example.org/user/smith/address/1"/>
  <street>1, Main Street</street>
  <city>Some City</city>
</address>
```

当用来处理表述正文的代码不能访问请求 URI 或响应头的时候，在表述正文中包含 self 链接就很有用了。

对于那些包含多种本地化语言数据的表述，光用 Content-Language 头是不够的，要直接在表述正文中包含语言标签。详见下面这个例子，摘自 XML 1.0 规范：

```
# 响应
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8
Content-Language: en

<content>
  <text>The quick brown fox jumps over the lazy dog.</text> ❶
  <text xml:lang="en-GB">What colour is it?</text> ❷
  <text xml:lang="en-US">What color is it?</text> ❸
  <text xml:lang="de"> ❹
    <p>Habe nun, ach! Philosophie,</p>
    <p>Juristerei, und Medizin</p>
    <p>und leider auch Theologie</p>
    <p>durchaus studiert mit hei逻m Bem黨'n.</p>
  </text>
</content>
```

❶ 默认语言的文本，由 Content-Language 头指定

❷ en-GB 语言的文本

❸ en-US 语言的文本

❹ de 语言下所有子元素语言的文本

3.6 如何设计 JSON 表述

JSON 是一种基于 JavaScript 的数据格式。和 XML 一样，这是一种有通用目的、易于人们阅读、可扩展的格式。在 JavaScript 和 PHP 这样的语言里，解析 JSON 要比解析 XML 来得简单。大多数基于浏览器的客户端使用的 Web 服务更倾向于使用 JSON 作为表述格式。

问题描述

您想知道在 JSON 格式的表述中要包含什么数据。

解决方案

在每个表述中，包含一个指向该资源的 self 链接，对于那些组成资源的应用程序领域实体，在表述中包含它们的标识符。

如果表述中的对象是本地化的，添加一个属性来表示本地化内容的语言。

问题讨论

本节中的方法和处理 XML 的方法（详见 3.5 节）类似。下面是一个“人员”资源的表述：

```
{
  "name" : "John",
  "id" : "urn:example:user:1234",
  "link" :{
     "rel : "self",
     "href" : "http://www.example.org/person/john"
  },
  "address" :{
    "id" : "urn:example:address:4567",
    "link" :{
       "rel : "self",
       "href" : "http://www.example.org/person/john/address"
    }
    ...
    }
```

```
}
```

当 Content-Language 头不足以描述表述的本地化语言时，可添加一个属性来表示语言，就像下面这样：

```
{
  "content" :{
    "text" : [{
        "value" : "The quick brown fox jumps over the lazy dog."
      },
      {
        "lang" : "en-GB",
        "value" : "What colour is it"
      },
      {
        "lang" : "en-US",
        "value" : "What color is it"
      }
    ]
  }
}
```

3.7 如何设计集合表述

客户端会在集合成员中进行迭代。由于某些集合会包含大量成员资源，客户端需要对集合进行分页或滚动显示。

问题描述

您想知道在集合资源的表述中要包含什么内容。

解决方案

在每个集合表述中包含以下内容：

- 一个指向集合资源的 self 链接。
- 如果集合是分页的，并且还有下一页，要有一个指向下一页的链接。
- 如果集合是分页的，并且还有上一页，要有一个指向上一页的链接。
- 一个集合大小的指示符。

问题讨论

集合资源和其他资源差不多，只是在某些情况下它会包含大量成员。当服务器仅返回集合成

员的一个子集时，也应该提供一些链接，允许客户端对所有成员进行分页。下面是一个包含多篇文章的集合资源：

```
# 请求
GET /articles?contains=cycling&start=10 HTTP/1.1
Host: www.example.org

# 响应
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8
Content-Language: en

<articles total="1921" xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="self"
    href="http://www.example.org/articles?contains=cycling&start=10"/> ❶
  <atom:link rel="prev"
    href="http://www.example.org/articles?contains=cycling"/> ❷
  <atom:link rel="next"
    href="http://www.example.org/articles?contains=cycling&start=20"/> ❸
  <article>
    <atom:link rel="self"
      href="http://www.nytimes.com/2009/07/15/sports/cycling/15tour.html"/>
    <title>For Italian, Yellow Jersey Is Fun While It Lasts</title>
    <body>...</body>
  </article>
  <article>
    <atom:link rel="alternate"
      href="http://www.nytimes.com/2009/07/27/sports/cycling/27tour.html"/>
    <title>Contador Wins, but Armstrong Has Other Victory</title>
    <body>...</body>
  </article>
  ...
</articles>
```

❶ 指向集合本身的链接。

❷ 指向上一页的链接。

❸ 指向下一页的链接。

这个表述是对大量新闻文章的搜索结果，其中有三个链接——带有 self 关系类型的链接用于获得表述本身，带有 prev 关系类型的链接用于获得前 10 篇文章，另一个带有 next 关系类型的链接用于获得后 10 篇文章。客户端可以使用这些链接访问整个集合。

total 属性给了客户端一个指示符，说明集合中成员的数量。

虽然集合的大小在构建用户界面时很有用，但应该避免计算集合的准确大小。这对于 Web 服务的计算、易变性（volatile）甚至安全性来说，可能代价会很大。通常给出一个提示就足够了。

在 HTTP 的层面上，每一页都是不同的资源。这是因为本例结果的每一页都有一个不同的 URI，例如 http://www.example.org/books?contains=cycling 和 http://www.example.org/books?contains=cycling&start=10 。

3.8 何时以及如何提供 HTML 表述

HTML 是一种流行的超媒体格式，有浏览器可以作为通用客户端，用户可以与 HTML 表述进行交互，无须在浏览器中实现应用程序特定的逻辑。而且，还可以使用 JavaScript 和 HTML 解析器从 HTML 中提取或推断数据。本节会讨论 HTML 表述的利与弊，以及什么时候 HTML 是合适的选择。

问题描述

您想知道，是否必须同 XML 或 JSON 格式的表述一起设计 HTML 表述，如果是的话该怎么做。

解决方案

对于那些希望能被最终用户使用的资源，应该为它们提供HTML表述。避免为机器客户端设计HTML表述。要让Web爬虫和此类软件能够正常运作，可以使用微格式或RDFa[注3]在标记中注解数据。

问题讨论

HTML 是一种被广泛认同并支持的格式，支持它的客户端软件有浏览器、HTML 解析器、著作工具（authoring tool）和生成工具。它还是自描述的，允许用户使用任意兼容 HTML 的客户端与服务器进行交互。这让 HTML 成为了一种符合人类习惯的格式。举例来说，考虑以下 XML 格式的资源表述：

```
<person xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="self" href="http://example.org/person/john"/>
  <id>urn:example:user:1234</id>
  <name>John</name>
  <address>
     <atom:link rel="self" href="http://example.org/person/john"/>
      <id>usr:example:address:4567</id>
```

```
    <street>1 Main Street</street>
    <city>Seattle</city>
    <state>WA</state>
  </address>
</person>
```

您可以为相同资源设计一个等价的 HTML 表述（为了简单起见，省略了 CSS 样式）：

```
<html>
  <head>
    <title>John</title>
    <link rel="self" href="http://example.org/person/john"/>
  </head>
  <body>
    <h1>John</h1>
    <div>
      <div>1 Main Street</div>
      <div>Seattle</div>
      <div>WA</div>
    </div>
  </body>
</html>
```

以HTML文档的形式提供部分或全部表述时，考虑用微格式或RDFa来注解HTML。这么做可以让Web爬虫和同类软件从HTML文档中提取信息，而无须依赖文档的结构。上述范例用hcard微格式（http://microformats.org/wiki/hcard）注解后的HTML表述是这样的：

```
<html>
  <head>
    <title>John</title>
  </head>
  <body>
    <h1 class="fn">John</h1>
    <div class="vcard">
      <div class="adr">
        <div class="street-address">1 Main Street</div>
        <div class="locality">Seattle</div>
        <div><abbr class="region" title="Washington">WA</abbr></div>
      </div>
    </div>
  </body>
</html>
```

微格式通过 HTML class 属性来注解多个 HTML 元素，这样 HTML 的客户端就可以知道那些元素的语义了。hcard 微格式是 vcard 格式（RFC 2426）到 HTML 的一个映射。vcard 格式是一种用于表示地址的可互操作的标准。hcard 微格式中规定了几个 CSS 类名。上面的例子中，fn 是名称，adr 是地址，street-address 是街道名称，locality 是位置，而 region 是类似州这样的地

区。

任何可以处理微格式的 HTML 解析器都能从这个 HTML 文档中找到地址。添加这个格式并不会影响在浏览器中呈现的文档，因为微格式是用 class 属性来扩展 HTML 的。

您也可以用类似的方式来使用 RDFa：

```
<html>
  <head>
    <title>John</title>
  </head>
  <body>
    <div xmlns:v="http://www.w3.org/2001/vcard-rdf/3.0#"
        about="http://example.org/person/john">
      <h1 property="v:FN" href="http://example.org/person/john">John</h1>
      <div role="v:ADR">
        <div property="v:Street">1 Main Street</div>
        <div property="v:Locality">Seattle</div>
        <div><abbr property="v:Region" title="Washington">WA</abbr></div>
      </div>
    </div>
  </body>
</html>
```

唯一的区别在于这个例子中使用 RDFa 和 vcard 格式来标注 HTML 元素。某些搜索引擎会使用这些注解从 HTML 文档中解读信息语义。

请注意，RDFa 只是针对 XHTML 1.1 规定的。但是，目前所有的浏览器都支持用于 HTML 文档的 RDFa。

3.9 如何返回错误

HTTP 基于表述的交换，对于错误来说也是如此。当服务器发生错误时，都会返回一个反映错误状态的表述，无论这个错误是由客户端提交的请求导致的，还是服务器自己的原因。这其中包括了响应状态码、响应头和一段包含错误描述的正文。

问题描述

您想知道如何给客户端返回错误。

解决方案

对于那些由客户端的输入所造成的错误，返回带 4xx 状态码的表述。对那些由于服务器实现

或其当前状态造成的错误，则返回带 5xx 状态码的表述。这两种情况下，都要包含一个 Date 头，它是表示错误发生时间的日期-时间值。

除非请求的方法是 HEAD，否则都应该在表述中包含一段正文，使用内容协商（详见第 7 章）或适合阅读的 HTML 或纯文本对其进行格式化和本地化。

如果能以独立的、适合阅读的文档形式来提供纠正或调试错误的信息，就包含一个指向该文档的链接，可以使用 Link 头，也可以使用正文中的链接。

如果为了后期追踪或分析，在服务器上记录了错误日志，应该提供一个可以找到该错误的标识符或链接。举例来说，客户端在向服务器团队报告错误时可以把错误码一起告诉他们。

响应正文要具有描述性，但不应该包含诸如错误堆栈、数据库连接错误之类的详细信息。如果可以的话，说明客户端可以采取的后续措施，纠正错误或者帮助服务器调试并修复错误。

问题讨论

HTTP 1.1 定义了两类错误码，一类介于 400 到 417，另一类介于 500 到 505。一些 Web 服务犯的一个常见错误是返回了表示成功的状态码（200 到 206 与 300 到 307 的状态码），但在消息体里却有错误描述。

```
# 避免返回成功状态码却在正文中包含错误。
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

<error>
  <message>Account limit exceeded.</message>
</error>
```

这样 HTTP 软件就无法检查错误了。例如，缓存会把它当成响应来缓存，并将它提供给后续的客户端，哪怕那些客户端能发起成功请求。

由客户端输入导致的错误：4xx

下面给出了在服务器端应用程序代码中可能会生成的错误码（它们不是 Web/应用服务器自动生成的）：

400（Bad Request）

当服务器由于语法错误无法解读请求时，返回该错误码。

HTTP 1.1 中仅定义了一种可以返回该错误的情况，就是请求中没有包含 Host 头。

401（Unauthorized）

当客户端无权访问资源，但在身份验证后可以获得访问权限时返回该错误码。如果服务器就算是在身份验证后也不允许客户访问资源，那应该返回 403（Forbidden）错误码。

返回该错误码时，应该包含一个带有身份验证方法的 WWW-Authenticate 头。通常使用的方法是 Basic 和 Digest。

403（Forbidden）

当服务器不让客户端获得资源的访问权限，就算通过身份验证也没用时，返回该错误码。

举例来说，您可以在用户已经经过身份验证但不允许请求这个资源时返回该错误码。

404（Not Found）

当没有找到资源时返回该错误码。如果可能的话，在消息体中说明原因。

405（Not Allowed）

当资源不允许使用某个 HTTP 方法时返回该错误码。

返回一个 Allow 头，其中带有该资源的有效 HTTP 方法。

406（Not Acceptable）

详见本书完整版 7.7 节。

409（Conflict）

当请求与资源的当前状态有冲突时返回该错误码。还会包含一段正文来解释原因。

410（Gone）

资源以前存在，但今后不会再存在时，返回该错误码。

除非记录了被删除的资源，否则不能返回这个错误码。如果没有在服务器端记录被删除的资源，应该用 404（Not Found）取而代之。

412（Precondition Failed）

详见本书完整版 10.4 节。

413（Request Entity Too Large）

当 POST 或 PUT 请求的消息体过大时返回该错误码。

如果可能，在正文中说明允许哪些内容，提供一个备选方案。

415（Unsupported Media Type）

当客户端用一种服务器不理解的格式来发送消息体时返回该错误。

由服务器端导致的错误：5xx

下面给出了一些错误码，当服务器错误造成请求失败时可能会生成这些错误码：

500（Internal Server Error）

由于某些实现上的问题，您的代码在服务器端失败时返回该错误码是最好的选择。

503（Service Unavailable）

服务器在某个特定间隔或一段不确定的时间内无法完成请求时，返回该错误码。

抛出该错误的两个常见场合是后端服务器失败（例如数据库连接失败）或者是客户端请求达到了某个服务器设定的频率上限。

如果可能的话，包含一个带日期或秒数的 Retry-After 响应头，用它的值来做提示。

HTTP 状态码是标准化的，但状态消息却不是。那些都是 HTTP 1.1 使用的消息。服务器可以自由使用特定于应用程序的错误消息字符串。

针对错误的消息正文

对于所有的错误，都要在错误响应中包含一段正文，当 HTTP 方法是 HEAD 时除外。其中，包含以下部分或全部内容：

- 一段描述错误情况的摘要。

- 如果可以的话，提供一段更长的描述，说明如何改正错误。
- 一个针对该错误的标识符。
- 一个用于更深入了解错误情况的链接，带有一些提示，说明如何解决这个错误。

下面是一个例子。当客户端发送请求进行转账操作时发生了这个错误：

```
# 响应
HTTP/1.1 409 Conflict
Content-Type: application/xml;charset=UTF-8
Content-Language: en
Date: Wed, 14 Oct 2009 10:16:54 GMT
Link: <http://www.example.org/errors/limits.html>;rel="help"

<error xmlns:atom="http://www.w3.org/2005/Atom">
  <message>Account limit exceeded. We cannot complete the transfer due to
  insufficient funds in your accounts</message>
  <error-id>321-553-495</error-id>
  <account-from>urn:example:account:1234</account-from>
  <account-to>urn:example:account:5678</account-to>
  <atom:link href="http://example.org/account/1234"
          rel="http://example.org/rels/transfer/from"/>
  <atom:link href="http://example.org/account/5678"
           rel="http://example.org/rels/transfer/to"/>
</error>
```

3.10 如何在客户端处理错误

在实现客户端时，有两类错误是客户端需要处理的。第一类是网络层面的失败，第二类是服务器返回的 HTTP 错误。编程库会处理前者，把它们包装成特定于编程语言的异常处理。后者是特定于应用程序的，要求显式地编写代码来做处理。

问题描述

您想知道如何理解服务器返回的错误。

解决方案

查看下面列出的条目，寻找针对每个错误码的合适动作：

400（Bad Request）

查看错误表述的正文，了解问题的根本原因。

401（Unauthorized）

如果客户端是面向用户的，提示用户提供身份信息。其他情况下，获取必要的安全身份信息。用带有 Authorization 头的请求进行重试，其中包含身份信息。

403（Forbidden）

这个错误意味着禁止客户端用这个请求方法来访问资源。不要重复引起该错误的请求。

404（Not Found）

资源已经不存在了。如果在客户端保存了资源的数据，清除数据，或者将之标记为已删除。

405（Not Allowed）

查看 Allow 头来寻找适用于该资源的方法，然后做适当的代码变更，只用那些方法来访问资源。

406（Not Acceptable）

详见本书完整版 7.7 节。

409（Conflict）

查看 PUT 的表述正文中列出的冲突。

410（Gone）

将之等同于 404（Not Found）。

412（Precondition Failed）

详见本书完整版 10.4 节。

413（Request Entity Too Large）

在错误的正文里寻找关于有效长度的提示。

415（Unsupported Media Type）

查看表述正文，了解请求支持的媒体类型。

500（Internal Server Error）

记录该错误日志，随后通知服务器开发者。

503（Service Unavailable）

如果响应中有 Retry-After 头，在到达该时间前不要重试。这可能是整个服务器的错误，因此在客户端中要实现适当的补偿逻辑，在一段时间内避免向服务器发送请求。

问题讨论

明确地处理各种错误码可以让客户端变得更坚固。特别要留心那些会把网络层面的失败和 HTTP 错误转换为异常或错误类的 HTTP 客户端库程序。此类错误需要特别对待。

HTTP 状态码是可扩展的，服务器可以引入新状态码。如果客户端不理解 Xmn 状态码（其中的 X 是 2，3，4 或 5），那么应该将其视为 X00。例如，如果服务器返回 599，客户端不明白它是什么意思，就把它当做 500。对状态码 245 的处理方式也是类似的。

不要把 HTTP 错误视为 I/O 或网络异常，把它们当做一等应用程序对象。

第 4 章

设计 URI

URI 是跨越 Web 的资源描述符，一个 URI 由以下内容组成——协议（例如 http 和 https）、主机（例如 www.example.org）、端口号，后面紧跟一段或多段路径（例如/user/1234），还有查询字符串。在本章中，我们将关注于为 RESTful Web 服务设计 URI。

4.1 如何设计 URI

URI 是模糊的资源标识符，在大多数情况下，客户端并不需要关心服务器是如何设计 URI 的。但是，在设计 URI 时遵循常用惯例会有不少优势：

- 遵循惯例的 URI 一般容易调试和管理。
- 服务器可以集中代码，以便从请求 URI 中提取数据。
- 可以避免花费宝贵的设计与实现的时间来发明处理 URI 的新惯例和规则。
- 通过跨域、子域和路径来对服务器的 URI 进行分区，这为您带来了负载分配（distribution）、监控、路由和安全方面的操作灵活性。

问题描述

您想知道为资源设计 URI 的最佳实践。

解决方案

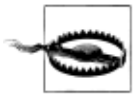
针对本地化、分布式、强化多种监控及安全策略等方面的需求，可以使用域及子域对资源进行合理的分组或划分。

- 在 URI 的路径部分使用斜杠分隔符（/）来表示资源之间的层次关系。
- 在 URI 的路径部分使用逗号（,）和分号（;）来表示非层次元素。
- 使用连字符（-）和下划线（_）来改善长路径中名称的可读性。
- 在 URI 的查询部分使用“与”符号（&）来分隔参数。

- 在 URI 中避免出现文件扩展名（例如.*php*，.*aspx* 和.*jsp*）。

问题讨论

URI 设计仅仅是实现 RESTful 应用程序的一个方面。在设计 URI 时还有一些需要考虑的惯例。



正如 URI 设计是 Web 服务成功的重要因素一样，将 URI 设计上花费的时间控制到最少也很重要，取而代之的是将注意力放到 URI 的一致性上。

域和子域

从逻辑上将 URI 分成域和子域可以为服务器管理提供很多操作方面的优势。在划分 URI 时，保证子域使用合理的名称。例如，服务器可以像下面这样，通过不同子域提供本地化的表述：

```
http://en.example.org/book/1234
http://da.example.org/book/1234
http://fr.example.org/book/1234
```

另一个例子根据客户端的类型进行划分。

```
http://www.example.org/book/1234
http://api.example.org/book/1234
```

在这个例子中，服务器提供了两个子域，一个针对浏览器，另一个针对自定义的客户端。这样的划分可以让服务器针对 HTML 和非 HTML 表述分配不同的硬件，应用不同的路由、监控或安全策略。

斜杠分隔符

根据惯例，斜杠（/）用于表示层次关系。这并不是一条硬性规定，但大多数用户在阅读 URI 时会遵循它。事实上，斜杠是 RFC 3986 中唯一提到的符号，一般用于表示层次关系。例如，下面所有 URI 路径中的斜杠都表示层次关系：

```
http://www.example.org/messages/msg123
http://www.example.org/customer/orders/order1
http://www.example.org/earth/north-america/canada/manitoba
```

一些 Web 服务可能会在结尾使用斜杠来表示集合资源。在使用这种方式时要格外小心，因为一些开发框架会错误地删除这些斜杠，或者在 URI 正规化时追加斜杠。

下划线和连字符

如果想让 URI 更易于人类阅读和解释，可以使用下划线（_）或连字符（-）：

```
http://www.example.org/blog/this-is-my-first-post
```

```
http://www.example.org/my_photos/our_summer_vacation/first_day/setting_up_camp/
```

两者并没有优劣之分，为了一致性，选择其中一个就一直使用下去。

与符号

在 URI 的查询部分可以使用与符号（&）来分隔参数：

```
http://www.example.org/print?draftmode&landscape
http://www.example.org/search?word=Antarctica&limit=30
```

在上面的第一个 URI 中，参数是 draftmode 和 landscape。第二个 URI 中的参数是 word=Antarctica 和 limit=30。

逗号和分号

使用逗号（,）和分号（;）来表示 URI 中的非层次部分。分号通常用于表示矩阵参数（matrix parameters）：

```
http://www.example.org/co-ordinates;w=39.001409,z=-84.578201
http://www.example.org/axis;x=0,y=9
```

这些符号在 URI 的路径和查询部分中是合法的，但并非所有的代码库都会将逗号和分号识别为分隔符，也许会需要一些自定义代码来提取这些参数。

句号

句号（.）除了用在域名里，还可以在 URI 中分隔文档和文件扩展名：

```
http://www.example.org/my-photos/flowers.png
http://www.example.org/index.html
http://www.example.org/api/recent-messages.xml
http://www.example.org/blog/this.is.my.next.post.html
```

最后一个例子是合法的，但是可能会造成一些混乱。一些代码库使用句号来开始 URI 中的文件扩展名，带有多个句号的 URI 会返回意想不到的结果，也可能造成解析错误。

除了历史遗留原因，不要在URI中使用句号。客户端应该使用表述的媒体类型来感知如何处理该表述。根据扩展名来“检测”（sniffing）媒体类型会造成安全隐患。举例来说，由于Internet Explorer 的媒体类型检测实现原因（http://msdn.microsoft.com/en-us/library/ms775148(VS.85).aspx），多个不同版本的Internet Explorer都存在安全隐患。

特定于实现的文件扩展名

考虑下列 URI：

```
http://www.example.org/report-summary.xml
http://www.example.org/report-summary.jsp
http://www.example.org/report-summary.aspx
```

所有这三个 URI，数据都是相同的，表述格式也可能是一样的，但文件扩展名表示了生成该资源表述所使用的技术。如果使用的技术改变了，那么 URI 也将会改变。

空格和大写字母

空格是合法字符，根据 RFC 3986，空格应该被编码为%20。但是，application/x-www-form-urlencoded 媒体类型（HTML 表单元素所使用的类型）会将空格编码为加号（+）。考虑下面的 HTML：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html lang="en">
  <head>
    <title>Search</title>
  </head>
  <body>
    <form method="GET" action="http://www.example.org/search"
      enc-type="application/x-www-form-urlencoded">
      <label for="phrase">Enter a search phrase</label>
      <input type="text" name="phrase" value=""/>
      <input type="submit" value="Search"/>
    </form>
  </body>
</html>
```

当用户提交了一个搜索短语"Hadron Supercollider"，最终的 URI（使用 application/x-www-form-urlencoded 规则）会是这样的：

```
http://www.example.org/search?phrase=Hadron+Supercollider
```

那些不知道 URI 是如何生成的代码会使用 RFC 3986 来解释 URI，搜索短语会被当成“搜索短语会被当成使用 example.org。

对于那些尚未准备接受 application/x-www-form-urlencoded 媒体类型编码的 URI 的 Web 服务，这种不一致性会造成编码错误。这并不是常见浏览器才有的问题，在一些代码库中也会遇到。

URI 中的大写字母也有可能出问题。RFC 3986 中将 URI 定义为除了协议和主机，其他部分均为大小写敏感的。例如，http://www.example.org/my-folder/doc.txt 和 HTTP://WWW.EXAMPLE.ORG/my-folder/doc.txt 是一样的，但是和 http://www.example.org/My-Folder/doc.txt 却不一样。然而，当资源来自文件系统时，这些 URI 对基

于 Windows 的 Web 服务器来说是一样的。这一大小写不敏感特性并不影响 URI 的查询部分。

出于这些原因，应尽量避免在 URI 中使用大写字符。

第 5 章

查询

查询信息是 HTTP GET 方法的一种常见应用。查询通常涉及三个组成部分，即过滤（filtering）、排序（sorting）和投影（projection）。过滤是基于一些过滤条件选择实体的一个子集的过程。排序会影响服务器是如何排列响应中结果的。投影是选择实体中的哪些字段将被包含到结果中的过程。例如，发送到电影服务器的查询请求可能会涉及按类型过滤电影，然后根据上映日期倒序排序，最后在返回客户端的响应中只选择标题、年份以及每部电影的简单介绍。

只要关注过 URI 和表述，查询设计还是相对比较简单的。客户端负责运行查询，服务器的职责包括设计 URI 来支持过滤、排序和投影，设计表述，设置合适的缓存头。本章将涉及查询设计中那些协议可见（protocol-visible）的方面，包括如下内容：

5.1 节，“如何针对查询设计 URI 对

本节展示了如何针对查询来设计 URI。

5.2 节，“如何设计查询响应”

本节展示了如何以集合元素的表述形式对查询结果进行建模。

5.3 节，“如何支持有大量输入的查询请求”

通过本节了解如何处理有大量输入的查询。

5.4 节，“如何存储查询”

使用本节中的内容来实现存储查询。

5.1 如何针对查询设计 URI

问题描述

您想知道如何设计 URI 来支持查询。

解决方案

使用查询参数让客户端来指定过滤器条件、排序字段和投影。将查询参数作为带有合理默认值的可选参数。为了支持常用查询，可以使用预定义的命名查询，编写文档说明每个参数的用途。

问题讨论

使用查询参数来设计查询是一种常用惯例，根据自己的用例，可能需要支持以下一种或全部情况的查询参数：

- 从可用资源中选择数据
- 指定排序条件
- 罗列要包含在响应中的资源的字段

举个例子，考虑一个标识书评的 URI。

```
http://www.example.org/book/978-0374292881/reviews
```

当客户端提交 GET 请求时，服务器返回一组书评。针对这个 URI，服务器也许会应用默认查询，该查询等价于：

```
http://www.example.org/book/978-0374292881/reviews?sortbyDesc=created&limit=5
```

该 URI 中包含一个查询，返回最新的 5 条书评，按照评论创建日期倒序排列。针对这个例子有多种可能的查询：

```
# 选择作者中包含"择作者中包的所有书评
http://www.example.org/book/978-0374292881/reviews?author=Jane

# 选择所有五星级书评
http://www.example.org/book/978-0374292881/reviews?rating=5

# 选择所有2009年8月15日后发布的书评
http://www.example.org/book/978-0374292881/reviews?after=2009-08-15

# 选择所有2009年8月15日后发布的书评，结果按发布日期升序排列
http://www.example.org/book/978-0374292881/reviews?after=2009-08-15&sortbyAsc=date
```

所有这些 URI 都包含过滤器和排序条件，将之作为查询条件。您还可以进一步细化该查询的输出，比方说，只返回书评的标题：

```
http://www.example.org/book/978-0374292881/reviews?after=2009-08-15&
    sortbyAsc=date&fields=title
```

在这个 URI 中，fields 参数用于指定投影。除此之外，如果大多数客户端只需要按创建日期倒序排列的书评摘要，服务器可以预定义一个查询，包含一个投影，内容是标题、评级和每个书评的链接：

```
http://www.example.org/book/978-0374292881/reviews?after=2009-08-15&
    view=summary
```

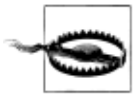
view 参数的值是一个预定义的查询。预定义查询让您有机会可以去优化常用查询的服务器实现，提供更快的响应时间。举例来说，在本例中，服务器可以在内存中缓存最流行的书评摘要。

考虑设计针对常用查询的预定义查询。

最后，您可以扩展查询，让客户端执行特定查询（ad hoc query）。以下是一些例子：

```
# 获取所有标题包含"取所有标，发行于2000年后，有至少100条评论的电影，
# 按年份排序
http://www.example.org/movies$contains('war')$compare(year>2000)
     $compare(count(comments)>100)?$sortby=year

# 使用query参数的值作为SQL中WHERE的子句
http://www.example.org/movies?query=
     '.title%20like%20'war'%20and%20year%20%3E%202000%20order%20by%20year'

# 使用XPath表达式来选择电影标题
http://www.example.org/movies[year>2000&genre='war']/title
```

这样的查询对客户端来说很灵活，客户端可以将服务器视为数据库。但是，这削弱了服务器优化数据存储和后端缓存的能力，从而降低了性能。这些查询也可能会造成 URI 和数据存储方式的紧耦合。



避免那些使用通用查询语言（例如 SQL 或 XPath）的特定查询。

一些服务器在查询上使用 HTTP 范围（range）请求。例如：

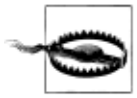
```
GET /book/978-0374292881/reviews HTTP/1.1
Host: www.example.org
```

```
Range: query:after=2009-08-15&sortbyAsc=date

# 请求
GET /report/June2009 HTTP/1.1
Host: www.example.org
Range: xpath://title
```

但是，HTTP 中除了字节范围之外并没有定义其他范围请求，就像下面的例子：

```
# 获取部分表述的请求
GET /docs/reportsJune2009.pdf HTTP/1.1
Host: www.example.org
Accept: application/pdf
Range: bytes=10241-20480

# 响应
HTTP/1.1 206 Partial Content
Content-Type: application/pdf
Content-Range: bytes=102341-20480
...
```

在不是字节范围的情况下，缓存可能会忽略范围请求。相反，查询参数则更容易实现与支持。



在实现查询时避免范围请求。

5.2 如何设计查询响应

本节讨论如何将集合用做资源来实现查询。

问题描述

您想知道如何设计查询响应的表述。

解决方案

将查询响应的表述设计为集合资源。可以通过 3.7 节了解如何设计集合表述。设置合理的过期缓存头。

如果查询没有匹配到任何资源，返回一个空集合。

问题讨论

集合是一种很方便的查询表述建模方法。以下是一个查询的结果，获取最多 5 篇发布于 2009 年 8 月 15 日后书评，按书评创建日期倒序排列：

```
# 请求
GET/book/978-0374292881/reviews?after=2009-08-15&sortbyDesc=created&limit=5
HTTP/1.1
Host: www.example.org

# 响应
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8
Cache-Control: max-age=86400
Content-Language: en

<reviews total="23" ❶  xmlns:atom="http://www.w3.org/2005/Atom"
   xml:base="http://www.example.org/978-0374292881">
   <atom:link rel="self"
      href="/book/reviews?after=2009-08-15&sortbyDesc=created&limit=5"/> ❷
         <atom:link rel="next"
      href="/book/reviews?after=2009-08-15&sortbyDesc=created&limit=5&start
=5"/> ❸
      <review>
      <atom:link rel="self" href="/book/review/03213"/>
      <created>2007-08-02</created>
      <title>Oversimplified?</title>
      <body>...</body>
      </review>

      <!-四篇书评 -->
      ...
</reviews>
```

❶ 查询匹配到的书评总数

❷ 返回头 5 篇匹配该查询的书评的 URI 链接

❸ 返回后 5 篇匹配该查询的书评的 URI 链接

该表述中包含 5 篇书评和一个指向后 5 篇书评的链接，正如 3.7 节中设计的那样。您可以让客户端细化该查询的输出，只返回所有书评的链接：

```
# 请求
```

```
GET
/book/978-0374292881/reviews?after=2009-08-15&sortbyDesc=created&limit=5&
    fields=link HTTP/1.1
Host: www.example.org

# 响应
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8
Content-Language: en

<reviews total="23" xmlns:atom="http://www.w3.org/2005/Atom"
  xml:base="http://www.example.org/978-0374292881">
  <atom:link rel="self"
     href="/book/reviews?after=2009-08-15&sortbyAsc=date"/>
  <atom:link rel="next" href="/book/reviews?after=2009-08-15&sortbyAsc=
           date&next=5"/>
  <atom:link rel="http://www.example.org/rels/review"
     href="/book/review/ 03213"/>
  <atom:link rel="http://www.example.org/rels/review"
     href="/book/review/03493"/>
  <atom:link rel="http://www.example.org/rels/review"
     href="/book/review/04501"/>
  <atom:link rel="http://www.example.org/rels/review"
     href="/book/review/04731"/>
  <atom:link rel="http://www.example.org/rels/review"
     href="/book/review/04934"/>
</reviews>
```

在该表述中，服务器使用了一个扩展链接关系类型（extended link relation type）将链接的URI 标识为书评，客户端可以使用这个 URI 来获取书评的表述。

查询参数的每次交换和组合都会产生不同的 URI。这可能会降低缓存性能，因为从协议层上来看每个 URI 对应于一个不同的资源。为了尽可能地减少 URI 的数量，可以考虑上一节中提到的预定义查询。

5.3 如何支持有大量输入的查询请求

尽管 HTTP 没有限制 URI 的长度，但它的一些实现对此却有限制。像 Internet Explorer 这样的浏览器将 URI 的长度限制为 2,083 个字符。Apache Web 服务器默认将请求行（即 GET/jobs?params.....HTTP/1.1）的长度限制为 8,190 字节（详见 Apache 的 LimitRequestLine 指令文档）。Microsoft 的 Internet Information Services（IIS）中用于表示请求行和 HTTP 头的累计字节数的默认值为 16,384（详见 IIS 的 MaxClientRequestBuffer 的文档）。Squid 将 URI 限制在 8,192 字节。这些限制通常是出于安全原因，例如避免缓冲溢出，它们也阻止了用户将大量

过滤器条件编码到 URI 中。

问题描述

您想知道如何支持那些涉及大量查询参数的查询。当这些参数包含在 URI 中时，这会导致 URI 超过多种 HTTP 层软件设定的长度限制。

解决方案

使用 HTTP POST 来支持大查询。

问题讨论

使用 POST 来处理查询削弱了 HTTP 的统一接口，根据定义，GET 才是用于安全、幂等地获取信息的。然而在遇到实际限制时，这种权衡也是必不可少的。例如，服务器可以让客户端基于工作地点、资格限定、经验要求、工作类型、关键字、公司名称等条件来搜索职位，这个条件列表太长了。当客户端通过查询参数将这些条件编码进 URI 时，URI 或请求行的长度就可能超过之前提到的限制。此时，可以使用 POST 来支持这种查询：

```
# 请求
POST /jobs HTTP/1.1

Host: www.example.org
Content-Type: application/x-www-form-urlencoded

keywords=web,ajax,php&industry=software&experience=5&...
```

该查询被编码为一个 application/x-www-form-urlencoded 字符串放置于请求内容中。服务器返回一个搜索结果的表述：

```
# 响应
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

<postings                              xmlns:atom="http://www.w3.org/2005/Atom"
xml:base="http://www.example.org">
  <posting>
    <atom:link rel="self" href="/job/499"/>
    ...
  </posting>
  <posting>
    <atom:link rel="self" href="/job/1863"/>
    ...
  </posting>
```

```
  ...
</postings>
```

考虑到这个操作是安全且幂等的，使用 POST 方法是对 HTTP 统一接口的一种误用，其后果是丧失了缓存能力。

添加 Cache-Control 或 Expires 头都无济于事，因为缓存把 POST 方法的响应当成是不可缓存的。

另一个限制是分页时的延时，要浏览搜索结果，客户端需要重复 POST 请求：

```
# 获取从10开始的结果的请求
POST /jobs HTTP/1.1
Host: www.example.org
Content-Type: application/x-www-form-urlencoded

start=10&amp;keywords=web,ajax,php&amp;industry=software&amp;experience=5&a
mp;

# 响应
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

<postings                              xmlns:atom="http://www.w3.org/2005/Atom"
xml:base="http://www.example.org">
  <posting>
    <atom:link rel="self" href="/job/5323"/>
    ...
  </posting>
  <posting>
    <atom:link rel="self" href="/job/435"/>
    ...
  </posting>
  ...
</postings>
```

因为这些结果是无法缓存的，客户端用户界面来回地浏览结果都会导致服务器（而非缓存）去响应请求。这为客户端带来了额外的延时，同时降低了服务器的可伸缩性。如果您的 Web 服务中频繁需要此类查询，请使用 5.4 节中的内容在服务器上存储查询。

5.4 如何存储查询

存储查询可以让那些使用 POST 方式发送的查询变得可以缓存。5.3 节讲述了使用 POST 来处理有大量参数的查询。本节会向您展示如何在服务器上存储这些查询，以便客户端可以用 GET 方式来执行被存储的查询。

问题描述

您想知道如何存储大查询请求，以便客户端可以用 GET 来执行它们。

解决方案

当客户端用 POST 发起一个查询请求时，创建一个新资源，它的状态中包含查询条件。返回一个带 Location 头的响应码 201（Created），Location 指向创建的资源。实现一个针对新资源的 GET 请求，返回查询结果。

如果同一客户端或另一个客户端用 POST 发起了相同查询请求，找到匹配该请求的资源，客户端被重定向到该资源的 URI 上。

问题讨论

通过在数据存储中永久保存查询条件，并为存储的查询分配一个 URI，您可以将查询结果变为可缓存的。客户端能用该 URI 来重复查询。通过将基于 POST 的查询转变为对资源的 GET 请求，缓存能为客户端提供缓存后的查询结果表述。

作为查询请求的响应，服务器保存了该查询，并为它分配了 URI http://www.example.org/query/1：

```
# 请求
POST /jobs HTTP/1.1
Host: www.example.org
Content-Type: application/x-www-form-urlencoded

keywords=web,ajax,php&amp;industry=software&amp;experience=5&amp;...

# 响应
HTTP/1.1 201 Created
Content-Type: application/xml;charset=UTF-8
Location: http://www.example.org/query/1
Content-Length: 0
```

客户端可以使用所创建的资源来获取查询结果：

```
# 请求
```

```
GET /query/1 HTTP/1.1
Host: www.example.org

# 响应
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8
Date: Wed, 28 Oct 2009 07:22:34 GMT
Cache-Control: max-age:3600
Expires: Wed, 28 Oct 2009 08:22:34 GMT

<postings                              xmlns:atom="http://www.w3.org/2005/Atom"
xml:base="http://www.example.org">
  <posting>
    <atom:link rel="self" href="/job/499"/>
    ...
  </posting>
  <posting>
    <atom:link rel="self" href="/job/863"/>
    ...
  </posting>
  ...
</postings>
```

此外，由于查询是经过缓存的，服务器可以通过 GET 来支持查询结果分页，而不用像 5.3 节里那样通过 POST 来实现：

```
# 请求
GET /query/1?start=10 HTTP/1.1
Host: www.example.org
```

存储查询弥补了使用 POST 方式处理查询的一些局限。缺点是不得不将查询永久保存为一个资源。此外，如果查询的数量很大，服务器最终有可能积聚大量不频繁使用的查询，需要频繁清理这些查询。

还要注意，将查询变为可缓存的并不能保证将来结果一定是来自于缓存。如果此类查询的数量很多，每个 URI 对应缓存中不同的响应副本，缓存命中率会很低。缓存很快会被填满，它会废弃那些不太使用的 URI。

关于作者

Subbu Allamaraju 是 Yahoo!的架构师，曾负责开发设计 RESTful Web 服务的标准及实践，目前负责为某些面向开发者的平台做架构设计。在此之前，他在 BEA Systems, Inc.开发 Web 服务及基于 Java 的软件，并参与制定 JCP 和 OASIS 标准。Subbu 参与了 4 本 J2EE 书籍的编写，均由 Wrox 发行出版。想对他有更深入的了解，请访问 *http://www.subbu.org*。

封面介绍

本书的封面动物是一只刺背鳄蜥或楔齿蜥，通常也被称为喙头蜥（tuatara），分布于新西兰；"tuatara"是新西兰毛利语，意思是"背上的山峰"（指它们那尖锐带刺的脊骨）。"刺背鳄蜥"其实是一个误称；虽然喙头蜥和普通的蜥蜴很像，但它们在解剖学上却很不一样，而且多在夜间活动，喜欢凉爽的天气。喙头蜥曾在 1831 年被大英博物馆错误分类为蜥蜴，后由动物学家 Albert Günther 于 1867 年重新分类为喙头目（Rhynchocephalia），该目中还有很多知名的中生代化石级物种。事实上，一些科学家把喙头蜥称为"活化石"，因为它们是喙头目中唯一仍然存活的。

喙头蜥发育非常缓慢--13-20 岁前都不算成熟，它们会一直长到大约 30 岁。人们相信野生的喙头蜥可以存活 80 年甚至更久。它们的平均长度是 20-31 英尺，体重 1-3 磅。喙头蜥可以有灰色、橄榄色或砖红色，它们的颜色会在其生命中发生变化。成年喙头蜥至少每年蜕一次皮。其他的生理特征包括双颞窝头骨（两面各有一个颞窝）、没有外耳、楔齿结构（牙齿紧紧咬合在额骨上--这又是另一个区别于蜥蜴的地方）、第三只眼。这第三只眼长在头顶（成年喙头蜥的第三只眼长在皮肤下），虽然不是用来看东西的，但却拥有视网膜、晶体和神经末梢，可以感光，有人认为它也可以帮助喙头蜥感应时间或季节。

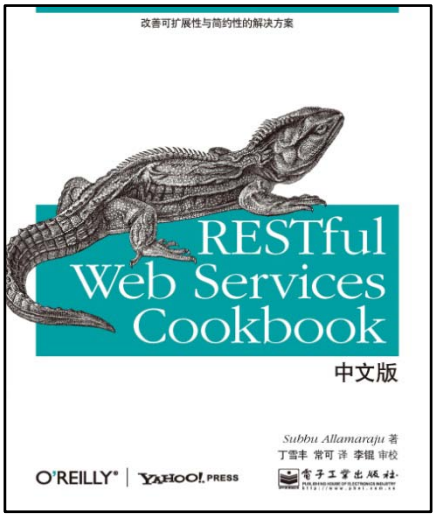
尽管喙头蜥濒临灭绝，但在新西兰仍然可以经常看到它们的身影。在 2006 年 10 月以前，新西兰的 5 分钱硬币上还印有喙头蜥，不过现在这种硬币已经不再流通了。喙头蜥频繁出现在毛利文化中，他们将其尊称为*ariki*（神的样子）。根据当地传说，喙头蜥是Whiro（掌管死亡和灾难的神）的使者，毛利女人不能吃它们。它们还代表禁忌（tapu），神圣不可侵犯的界线，跨越之后会有很严重的后果。毛利女人会在自己生殖器附近纹上蜥蜴或喙头蜥以象征禁忌。如今，喙头蜥被视为去往赋予人们生命的心理和精神国度途中的*taonga*（珍宝）和*kaitiaki*（守卫者）[注 1]。

封面图片取自 Wood 的《Animate Creation》。

注 1： taonga 和 kaitiaki 都是毛利语。

免费在线版本

（非印刷免费在线版）



了解本书更多信息请登录本迷你书的官方网站

InfoQ 中文站出品

本书由 InfoQ 中文站免费发放，如果您从其他渠道获取本书，请注册 InfoQ 中文站以支持作者和出版商，并免费下载更多 InfoQ 企业软件开发系列图书。

本迷你书主页为

http://www.infoq.com/cn/minibooks/restful-web-services-cookbook-cn