改善可扩展性与简约性的解决方案

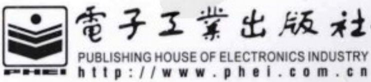
RESTful Web Services Cookbook

中文版

Subbu Allamaraju 著

丁雪丰 常可 译 李锟 审校

O'REILLY® | YAHOO! PRESS



RESTful Web Services Cookbook中文版

REST设计哲学激起了Web与企业应用开发者的无尽遐想，但用这种方式来开发真正的Web服务并非易事。本书包含了超过100个技巧，帮助您充分利用REST，HTTP和Web基础设施。无论使用何种编程语言和开发框架，您都将了解到如何为客户端/服务器应用设计满足性能、可扩展性、可靠性及安全目标的RESTful Web服务。

本书的每一节都包含一到两个问题描述，带有简单易学、步骤详尽的解决方案，还有使用HTTP请求与响应、XML、JSON和Atom片段的例子。您还将看到针对每个解决方案的实现指南，讨论其中的利弊和权衡。

- 了解如何设计满足多种应用场景的资源
- 成功设计表述与URI
- 使用链接与链接标头来实现超文本约束
- 理解何时及如何使用Atom和AtomPub
- 为了支持缓存，该做什么，不该做什么
- 了解如何实现并发控制
- 处理涉及复制、合并、事务、批处理及部分更新的高级用例
- 保护Web服务，支持OAuth

建议读者最好能对Web开发有一定认识
——包含XML，HTTP GET和POST。

“无论您打算使用何种语言构建RESTful服务，本书都是一本极佳的指南。内容丰富、考虑周到，囊括了与服务设计与实现相关的很多最佳实践。”

——Eben Hewitt
架构师
“Java SOA Cookbook”作者

Subbu Allamaraju，Yahoo!架构师。他设计了RESTful Web服务的标准及实践，目前负责开发者平台的架构工作。Subbu之前就职于BEA System, Inc.，开发Web服务及基于Java的软件。他参与了4本J2EE相关书籍的编写，均由Wrox发行出版。

O'REILLY®
oreilly.com.cn

图书分类：Web服务

策划编辑：张春雨
责任编辑：刘舫

www.phei.com.cn

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

O'Reilly Media, Inc.授权电子工业出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale in the mainland of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-121-14390-8

定价：59.00元

RESTful Web Services Cookbook 中文版

RESTful Web Services Cookbook

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

内 容 简 介

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。

978-0-596-80168-7 RESTful Web Services Cookbook © 2010 by O'Reilly Media, Inc. Simplified Chinese edition, jointly published by O'Reilly Media, Inc. and Publishing House of Electronics Industry, 2010. Authorized translation of the English edition, 2010 O'Reilly Media, Inc., the owner of all rights to publish and sell the same. All rights reserved including the rights of reproduction in whole or in part in any form.

本书中文简体版专有出版权由 O'Reilly Media, Inc.授予电子工业出版社，未经许可，不得以任何方式复制或抄袭本书的任何部分。

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：
装　　订：北京中新伟业印刷有限公司
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

目录

前言......xv

第 1 章　使用统一接口......1

1.1　如何保持交互的可见性......2
1.2　何时需要权衡可见性......4
1.3　如何维护应用程序状态......6
1.4　如何在服务器端实现安全和幂等的方法......9
1.5　如何在客户端处理安全和幂等方法......12
1.6　何时使用 GET 方法......13
1.7　何时使用 POST 方法......15
1.8　如何使用 POST 方法创建资源......17
1.9　何时使用 PUT 方法创建新资源......19
1.10　如何使用 POST 方法实现异步任务......20
1.11　如何使用 DELETE 方法实现异步删除......23
1.12　何时使用自定义 HTTP 方法......24
1.13　何时及如何使用自定义 HTTP 标头......26

第 2 章　识别资源......29

2.1　如何从领域名词中识别资源......30
2.2　如何选择资源粒度......31
2.3　如何将资源组织为集合......32

2.4 何时将资源合并为复合资源......35
2.5 如何支持计算或处理函数......37
2.6 何时及如何使用控制器来操作资源......40

第 3 章 设计表述......45

3.1 如何使用实体头来注解表述......46
3.2 如何解释实体头......50
3.3 如何避免字符编码不匹配......51
3.4 如何选择表述格式和媒体类型......52
3.5 如何设计 XML 表述......56
3.6 如何设计 JSON 表述......58
3.7 如何设计集合表述......59
3.8 如何保持同构的集合......61
3.9 如何在表述中使用可移植的数据格式......63
3.10 何时使用实体标识符......65
3.11 如何在表述中编码二进制数据......66
3.12 何时以及如何提供 HTML 表述......68
3.13 如何返回错误......70
3.14 如何在客户端处理错误......74

第 4 章 设计 URI......77

4.1 如何设计 URI......77
4.2 如何将 URI 用做模糊标识符......81
4.3 如何让客户端将 URI 视为模糊标识符......83
4.4 如何保持酷的 URI......85

第 5 章 Web 链接......88

5.1 如何在 XML 表述中使用链接......89
5.2 如何在 JSON 表述中使用链接......92

5.3 何时以及如何使用链接标头......93
5.4 如何分配链接关系类型......94
5.5 如何使用链接来管理应用程序的流程......97
5.6 如何处理临时 URI......101
5.7 何时以及如何使用 URI 模板......103
5.8 如何在客户端使用链接......105

第 6 章 Atom 和 AtomPub......108

6.1 如何利用 Atom 建模资源......109
6.2 何时使用 Atom......113
6.3 如何使用 AtomPub 服务和分类文档......117
6.4 如何针对 Feed 和 Entry 资源使用 AtomPub......119
6.5 如何使用媒体资源......122

第 7 章 内容协商......125

7.1 如何标明客户端偏好......126
7.2 如何实现媒体类型协商......128
7.3 如何实现语言协商......129
7.4 如何实现字符编码协商......131
7.5 如何支持压缩......132
7.6 何时以及如何发送 Vary 头......133
7.7 如何处理协商失败......134
7.8 如何使用代理驱动的内容协商......136
7.9 何时支持服务器驱动的协商......137

第 8 章 查询......139

8.1 如何针对查询设计 URI......139
8.2 如何设计查询响应......142
8.3 如何支持有大量输入的查询请求......144

8.4 如何存储查询 146

第 9 章 Web 缓存 149

9.1 如何设置过期缓存头 150
9.2 何时设置过期缓存头 153
9.3 何时以及如何在客户端使用过期缓存头 156
9.4 如何支持复合资源的缓存 157
9.5 如何保持新鲜且温暖的缓存 158

第 10 章 条件请求 161

10.1 如何生成 Last-Modified 和 ETag 头 163
10.2 如何在服务器端实现条件 GET 请求 164
10.3 如何从客户端提交条件 GET 和 HEAD 请求 167
10.4 如何在服务器端实现条件 PUT 请求 169
10.5 如何在服务器端实现条件 DELETE 请求 173
10.6 如何从客户端发起无条件 GET 请求 175
10.7 如何从客户端提交条件 PUT 和 DELETE 请求 176
10.8 如何使 POST 请求条件化 178
10.9 如何生成一次性 URI 181

第 11 章 其他内容 184

11.1 如何复制资源 185
11.2 如何合并资源 187
11.3 如何移动资源 189
11.4 何时使用 WebDAV 方法 191
11.5 如何支持跨服务器的操作 193
11.6 如何获取资源的快照 195
11.7 如何撤销资源更新 198
11.8 如何为部分更新提炼资源 200

11.9 如何使用 PATCH 方法......203
11.10 如何批量处理相似的资源......206
11.11 如何触发批量操作......209
11.12 何时使用 POST 来合并多个请求......211
11.13 如何支持批量请求......215
11.14 如何支持事务......217

第 12 章 安全......220

12.1 如何使用基本身份验证来验证客户端......221
12.2 如何使用摘要身份验证来验证客户端......224
12.3 如何使用三方 OAuth......226
12.4 如何使用两方 OAuth......232
12.5 如何处理 URI 中的敏感信息......235
12.6 如何维护表述的机密性与完整性......237

第 13 章 可扩展性与版本控制......239

13.1 如何维持 URI 的兼容性......240
13.2 如何维持 XML 和 JSON 表述的兼容性......242
13.3 如何扩展 Atom......245
13.4 如何维持链接的兼容性......249
13.5 如何实现支持可扩展性的客户端......250
13.6 何时需要版本化......251
13.7 如何版本化 RESTful Web 服务......252

第 14 章 服务发现......256

14.1 如何编写 RESTful Web 服务的文档......256
14.2 如何使用 OPTIONS......259

附录 A 辅助读物......261

附录 B　REST 概述 265

附录 C　HTTP 方法 268

附录 D　Atom Syndication Format 273

附录 E　链接关系类型 279

索引 287

PDG

第 1 章

使用统一接口

1

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，“何时需要权衡可见性”
　有时您可能需要对可见性进行权衡以满足应用的需要，可通过本节找到一些类似的场合。

1.3 节，“如何维护应用程序状态”
　通过本节了解管理状态的最佳方式。

1.4 节，“如何在服务器端实现安全和幂等的方法”
　保持安全性和幂等性有助于保证服务器能安全处理重复性请求。需要实现服务器时请使用本节的内容。

1.5 节，“如何在客户端处理安全和幂等方法”
　请按照本节的内容来实现客户端的安全性和幂等性原则。

1.6 节，“何时使用 GET 方法”
　通过本节了解何时使用 GET 方法。

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

1.7 节，“何时使用 POST 方法”

通过本节学习何时使用 POST 方法。

2 1.8 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。

1.9 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。

1.10 节，“如何使用 POST 方法实现异步任务”

通过本节了解如何使用 POST 方法实现异步任务。

1.11 节，“如何使用 DELETE 方法实现异步删除”

通过本节学习如何使用 DELETE 方法实现异步的资源删除。

1.12 节，“何时使用自定义 HTTP 方法”

通过本节了解为什么不推荐使用自定义的 HTTP 方法。

1.13 节，“何时及如何使用自定义 HTTP 标头”

通过本节了解何时及如何使用自定义 HTTP 标头。

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（详见附录 A），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>
```

4

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。

正如我们将在 1.2 节讨论的内容一样，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 何时需要权衡可见性

本节讨论了一些可能需要对可见性做出权衡的常见场合。

问题描述

您想知道有哪些常见场合可能需要让请求和响应降低对协议的可见性。

解决方案

当有多个共享数据的资源，或一个操作总是要修改多个资源时，请考虑降低可见性，以得到更好的信息抽象、更松散的耦合程度、更好的网络效率、更好的资源粒度，或者纯粹为了方便客户端使用。

问题讨论

可见性经常与其他架构要求相冲突，如抽象、松耦合、效率和信息粒度等。例如，考虑一个“人”资源与一个相关的“地址”资源，任何客户端都可以提交一个 GET 请求得到这两个资源的表述，但为了方便客户端，服务器端可能会在“人”的资源表述中包含“地址”资源，就像下面这样：

```
# 获取“人”的请求
GET /person/1 HTTP/1.1
Host: www.example.org
Content-Type: application/xml;charset=UTF-8

<person>
  <name>John Doe</name>
  <address type="home">
    <street>1 Main Street</street>
    <city>Bellevue</city>
    <state>WA</state>
  </address>
</person>

# 获取地址的请求
GET /person/1/address HTTP/1.1
Host: www.example.org
Content-Type: application/xml;charset=UTF-8
<address type="home">
  <street>1 Main Street</street>
  <city>Bellevue</city>
  <state>WA</state>
</address>
```

让我们假设服务器允许客户端提交 PUT 请求更新这些资源。当一个客户端修改了其中一个资源，相关资源的状态也会改变。然而，在 HTTP 层面上，这些是相互独立的资源。只有服务器才知道它们是相互依赖的。数据的重叠是降低可见性的常见原因。

降低可见性的重要影响之一是缓存（详见第 9 章）。由于这些资源在 HTTP 层面上

是相互独立的，缓存将有两个“地址”的副本：一个独立的“地址”表述，以及作为“人”的一部分的表述，这样是很低效的。而且，在缓存中一个表述失效不会影响另一个表述，这样一来陈旧的表述就保留在缓存里了。

在这个例子中，您可以包含一个指向地址的引用，以此避免包含详细地址，从而消除这些资源之间的重叠。可以使用链接（详见第 5 章）向其他资源提供引用。

尽管提供链接可以减少数据重叠，但这将迫使客户端发起更多的请求。

在这个例子中，可见性和客户端的易用性之间需要权衡，也许还包括网络效率。一个需要“人”资源的客户端可以通过单个请求得到“人”及“地址”的信息。

下面这些情况下，您可能需要为了其他好处放弃可见性：

方便客户端
: 为了方便客户端使用，服务器可能需要设计特定目标的粗粒度组件资源（例如，2.4 节）。

抽象
: 为了抽象复杂的业务操作（包括事务），服务器可能需要使用控制器资源来改变其他资源（例如，2.6 节）。这样的资源可以隐藏业务操作的实现细节。

网络效率
: 当客户端需要在短时间内连续执行几个操作时，可能需要将这些操作组合到一个批处理中，以降低网络延迟（例如，11.10 节和 11.13 节）。

这些情况中，如果只关注可见性，您的 Web 服务就不得不设计成以无重叠的独立资源形式来显示所有数据。按这种方式设计的 Web 服务，可能导致资源粒度过细，客户端和服务器之间分离关注点不够，可详见 2.6 节中的例子。其他场合包括复制或合并资源、部分更新（详见第 11 章），可能也需要权衡可见性。

假如在设计过程的早期即意识到其影响，为得到其他好处而降低可见性未必是坏事。

1.3 如何维护应用程序状态

当您在阅读关于 REST 的文章时，经常得到这样的建议——在客户端保存应用程序

状态。但什么是“应用程序状态”？应当如何在客户端保存这些状态？本节描述了保存应用程序状态的最佳实践。

问题描述

您想知道如何管理 RESTful Web 服务的状态，这样就不需要依赖于服务器内存中的会话了。

解决方案

将应用程序状态编码到 URI 里，并通过链接在表述中包含这些 URI（见第 5 章）。让客户端使用这些 URI 与资源进行交互。如果状态过大，或出于安全或隐私考虑不能在网络中传输，则可以在持久化存储（如数据库或文件系统）中存储应用程序状态，并将其状态的引用编码在 URI 中。

问题讨论

考虑一个简化的汽车保险应用，假定其中涉及两个步骤。第一步，客户端提交一个带有司机和车辆细节信息的请求，服务器返回一个一周内有效的报价。第二步，客户端提交购买保险的请求。在这个例子中，应用程序的状态就是报价。服务器需要知道从第一个步骤返回的报价，基于它才能给出第二个请求中的保单。

应用程序状态是服务器需要在每个客户端的每个请求之间维护的状态。在客户端保持这个状态并不意味着要像 ASP.NET 和 JavaServer Faces 之类的 Web 框架所做的那样，把会话状态序列化到 URI 或 HTML 表单中。

由于 HTTP 是无状态协议，每个请求与之前的请求都是独立的。然而，交互应用程序通常要求客户端执行时遵循特定顺序。这就要求服务器在协议之外暂时存储每个客户端在步骤序列里的当前位置。这里的诀窍是管理状态，这样就可以在可靠性、网络性能和可扩展性之间寻求平衡。

资源表述中的链接是保持应用程序状态的最好地方，如下所示：

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

fname=...&lname=...&..

# 响应
HTTP/1.1 200 OK
```

8

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

<quote xmlns:atom="http://www.w3.org/2005/Atom">
  <driver>
    ...
  </driver>
  <vehicle>
    ...
  </vehicle>
  <offer>
    ...
    <valid-until>2009-10-02</valid-until>
    <atom:link href="http://www.example.org/quotes/buy?quote=abc1234"
         rel="http://www.example.org/rels/quotes/buy"/> ❶
  </offer>
</html>
```

❶ 一个包含应用程序状态的链接

在这个例子中，服务器把报价保存在数据存储里，并将主键编码在URI中。当客户端使用该URI发起请求购买保险时，服务器可以通过这个主键恢复该应用程序状态。

应当选择形如数据库或文件系统的持久化存储来保存应用程序状态。使用缓存或内存会话这样的非持久化存储会降低Web服务的可靠性，例如在服务器重启时状态可能会丢失，非持久化存储解决方案也会降低服务器的可扩展性。

如果报价所需的数据量很小，服务器可以将状态编码在URI中，正如稍后代码所展示的那样。

当在数据库中保存应用程序状态时，使用数据库复制（replication）以便所有服务器实例都可以访问这些状态。如果应用程序状态不是永久的，可能需要在某些地方清理这些状态。

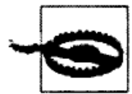
```
# 请求
GET /quotegen?fname=...&lname=...&... HTTP/1.1
Host: www.example.org

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

<quote xmlns:atom="http://www.w3.org/2005/Atom">
  <driver>
```

```
    ...
  </driver>
  <vehicle>
    ...
  </vehicle>
  <offer>
    ...
    <valid-until>2009-08-02</valid-until>
    <atom:link href="http://www.example.org/quotes/buy?fname=...&lname=...&..."
          rel="http://www.example.org/quotes/buy"/>
  </offer>
</html>
```

因为客户端需要在每个请求中发回以上这些数据，在链接中编码应用程序状态会降低网络性能。然而这样可以提高可扩展性，因为服务器不需要保存任何数据；也可以提高可靠性，因为服务器不需要使用复制。可以基于特定用例和状态数量，组合以上两种方式来管理应用程序状态，保持网络性能、可扩展性和可靠性之间的平衡。



当在链接中保存应用程序状态时，一定要加入一些检查方式（如签名）来检测和防止伪造状态。详见 12.5 节的示例。

1.4 如何在服务器端实现安全和幂等的方法

安全性和幂等性是服务器在实现某些方法时必须保证提供给客户端的。本节会讨论为什么安全性和幂等性如此重要，以及如何实现服务器端的安全性和幂等性。

问题描述

您想知道幂等性和安全性的具体含义，以及如何才能确保服务器提供的各种 HTTP 方法实现了这两个特性。

解决方案

实现 GET、OPTIONS 和 HEAD 方法时，不要引起任何副作用。当客户端重新提交一个 GET，HEAD，OPTIONS，PUT 或 DELETE 请求时，确保服务器提供同样的表述形式，并发情况除外（详第 10 章）。

10

问题讨论

安全性和幂等性是服务器要实现的 HTTP 方法特性，表 1-1 展示了哪个方法是安全的，哪个方法是幂等的。

表 1-1：HTTP 方法的安全性与幂等性

方法	安全?	幂等?
GET	Yes	Yes
HEAD	Yes	Yes
OPTIONS	Yes	Yes
PUT	No	Yes
DELETE	No	Yes
POST	No	No

实现安全方法

在 HTTP 协议中，安全方法是不会引起副作用的。客户端在向安全方法发送请求时，无须担心会引起意料之外的副作用。为了提供这一保证，可以将安全方法实现为只读操作。

安全性并不意味着服务器每次都必须返回同一结果。它只是表明客户端可以发起请求，并知道它不会改变资源的状态。举例来说，以下两个请求都是安全的：

```
# 第一次请求
GET /quote?symb=YHOO HTTP/1.1
Host: www.example.org

HTTP/1.1 200 OK
Content-Type: text/plain;charset=UTF-8

15.96

# 10 分钟后的第 2 次请求
GET /quote?symb=YHOO HTTP/1.1
Host: www.example.org

HTTP/1.1 200 OK
Content-Type: text/plain;charset=UTF-8

16.10
```

这个例子中，两次请求响应结果不同，可能是由其他客户端或后台操作引起的。

实现幂等方法

幂等性保证客户端重复发起某个请求的效果与一次请求的效果一致。幂等性大多与

11

网络或软件故障息息相关。客户端可以重复这些要求并期望同样的结果。例如，考虑一个客户端更新产品价格的案例。

```
# 请求
PUT /book/gone-with-the-wind/price/us HTTP/1.1
Host: www.example.org
Content-Type: application/x-www-form-urlencoded

val=14.95
```

现在假设因为网络故障，客户端读不到响应。因为 HTTP 协议规定 PUT 是幂等的，所以客户端可以重复提交请求：

```
# 请求
PUT /book/gone-with-the-wind/price/us HTTP/1.1
Host: www.example.org
Content-Type: application/x-www-form-urlencoded

val=14.95

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

<value>14.95</value>
```

要让这种方法正常运作，除 POST 以外的所有方法都必须是幂等的。在编程语言的术语中，幂等方法类似于“setter”。举例来说，以下代码中的 setPrice 方法，调用多次和只调用一次产生的效果是一样的：

```
class Book {
    private Price price;
    public void setPrice(Price price) {
        this.price = price;
    }
}
```

DELETE 方法的幂等性

DELETE 方法是幂等的。这意味着就算服务器在前一个请求中已经删除了资源，它也必须返回 200（OK）响应码。但实际上，要把 DELETE 实现为幂等操作，需要服务器追踪所有已经删除的资源。否则，它可能会返回一个 404（Not Found）响应码。

```
# 第 1 次请求
DELETE /book/gone-with-the-wind HTTP/1.1
Host: www.example.org

# 响应
HTTP/1.1 200 OK

# 第 2 次请求
DELETE /book/gone-with-the-wind HTTP/1.1
Host: www.example.org
```

```
# Response
HTTP/1.1 404 Not Found
Content-Type: text/html;charset=UTF-8

<html>
  ...
</html>
```

就算服务器保留了所有已删除资源的记录，安全策略也可能要求服务器对所有已不存在的资源返回一个 404(Not Found)响应码。

1.5 如何在客户端处理安全和幂等方法

问题描述

您想了解如何实现幂等和（或）安全的 HTTP 请求。

解决方案

把 GET，OPTIONS 和 HEAD 看做只读操作，需要时，可以随时发起这些请求。

在网络或软件出错时，重新提交 GET，PUT 和 DELETE 请求以做确认，提供 If-Unmodified-Since 和（或）If-Match 条件头（详见第 10 章）。

不要重复提交 POST 请求，除非客户端事先知道（例如，通过服务器的文档）服务器对任意特定资源的 POST 操作的实现是幂等的。

问题讨论

安全方法

任何客户端都可以发起任意次数的 GET，OPTIONS 和 HEAD 请求。如果服务器的实现在处理这类请求时导致了意想不到的副作用，那可以直接认定服务器的 HTTP 实现是错误的。

幂等方法

正如在 1.4 节所讨论的，幂等性保证了客户端可以在不能肯定服务器是否成功处理了请求时，重复发起这一请求。在 HTTP 中，除了 POST 以外的所有方法都是幂等的。在客户端实现上，当您在一个幂等方法上遇到软件或网络的错误时，可以实现一个重试请求的逻辑。下面是一段伪码片段：

```
try{
```

```
        // 提交一个 PUT 请求
        response = httpRequest.send("PUT", ...);
        if(response.code == 200){
            // 成功
            ...
        }
        else if(response.code >= 500){
            // 由于客户端错误造成失败
            ...
        }
        else if(response.code >= 400){
            // 由于服务器错误造成失败
            ...
        }
        ...
    }
    catch(NetworkFailure failure){
        // 重试请求
        ...
    }
```

在这个例子中，客户端实现了在网络出现故障时重复发起请求的逻辑，服务器端返回一个 4xx 或 5xx 错误的情况除外。客户端还要像往常那样继续处理 HTTP 层的错误（详见 3.14 节）。

因为 POST 不是幂等的，上面的做法不能套用在 POST 请求上，除非服务器明确声明可以这么做。10.8 节描述了一种服务器为 POST 请求提供幂等性的方法。

1.6 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。

14 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。

1.7 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

15

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 1.8 节。
- 通过一个控制器资源来修改一个或多个资源，详见 2.6 节。
- 执行需要大数据输入的查询，详见 8.3 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

<methodCall>
  <methodName>messages.delete</methodName>
  <params>
    <param>
      <value><int>1234</int></value>
    </param>
  </params>
</methodCall>注3
```

16 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：

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

注 3：　取自 RFC 2616 Sec 9.5（*http://tools.ieft.org/html/rfc2616#section-9.5*）。

- 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 8.3 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。

1.8 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

17

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

18 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——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。可以通过第 10 章了解这些头信息的详细内容。

1.9 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>
```

19

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。11.4 节中的 WebDAV 就是这样一个例子。

1.10 如何使用 POST 方法实现异步任务

HTTP 是一种同步、无状态的协议。当客户端向服务器提交一个请求时，无论成功与否，客户端都期望得到一个回答。但这并不意味着服务器必须在返回响应前结束对请求的处理。例如，在一个银行系统中，当您发起一次转账时，可能在下一个工作日前都不会进行转账，而客户端可能稍后就要检查状态。本节讨论如何使用 POST 方法处理异步请求。

问题描述

您想知道如何实现需要长时间才能完成的 POST 请求。

解决方案

在接收到 POST 请求时，创建一个新的资源，并返回状态码 202 (Accepted)，其包含新资源的表述，这个新资源的目的是让客户端可以跟踪异步任务的状态。仔细设计这个资源，让它的表述中包含请求的当前状态和诸如时间戳之类的相关信息。

当客户端向任务资源提交 GET 请求时，根据请求的当前状态，执行以下的某个动作：

执行中

返回响应码 200（OK）及包含当前状态的任务资源表述。

成功完成

返回响应码 303（See Other）及一个 Location 头，其中是展示任务输出的资源的 URL。

任务失败

返回响应码 200（OK）及告知资源创建已失败的任务资源的表述，客户端需要读取表述的内容以找到失败的原因。

问题讨论

考虑一个图片处理的 Web 服务，提供文件格式转换、光学字符识别、图像清理等服务。为了使用这个服务，客户端需要上传原始图片，根据上传图片的类型和大小，以及当前服务器的负载，服务器可能需要几秒至几个小时的时间来处理每一张图片。处理完成后，客户端应用程序可以查看或下载处理后的图片。

让我们从客户端提交 POST 请求以启动图片处理任务：

```
# 请求
POST /images/tasks HTTP/1.1
Host: www.example.org
Content-Type: multipart/related; boundary=xyz

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

...

--xyz
Content-Type: image/png

...

--xyz--
```

21

在这个例子中，客户端使用了一个分段（multipart）消息，其第一部分包含一份 XML 文档，描述了需要服务器执行的图片处理操作类型，第二部分是需要处理的图片。

在保证内容合法、所提供的图片处理请求可以被受理的前提下，服务器会创建一个新的任务资源：

```
# 响应
```

```
HTTP/1.1 202 Accepted ❶
Content-Type: application/xml;charset=UTF-8
Content-Location: http://www.example.org/images/task/1
Date: Sun, 13 Sep 2009 01:49:27 GMT
<status xmlns:atom="http://www.w3.org/2005/Atom">
  <state>pending</state>
  <atom:link href="http://www.example.org/images/task/1" rel="self"/>
  <message xml:lang="en">Your request has been accepted for processing.</message>
  <ping-after>2009-09-13T01:59:27Z</ping-after> ❷
</status>
```

❶ 响应码标识出服务器已经接受了处理请求

❷ 稍后检查状态的一个建议时间

随后，客户端可以向这个任务资源发送一个 GET 请求，如果服务器还在处理这个任务，可以返回如下响应：

```
# 请求
GET /images/task/1 HTTP/1.1
Host: www.example.org
# 响应

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

<status xmlns:atom="http://www.w3.org/2005/Atom">
  <state>pending</state>
  <atom:link href="http://www.example.org/images/task/1" rel="self"/>
  <message xml:lang="en">Your request is currently being processed.</message>
  <ping-after>2009-09-13T02:09:27Z</ping-after>
</status>
```

22

欲了解 ping-after 元素的取值原理，详见 3.9 节。

当服务器成功完成了图片处理后，它可以将客户端重定向到处理结果上。在这个例子中，处理结果是一个新的图像资源：

```
# 请求
GET /images/task/1 HTTP/1.1
Host: www.example.org

# 响应
HTTP/1.1 303 See Other ❶
Location: http://www.example.org/images/1
```

```
Content-Location: http://www.example.org/images/task/1

<status xmlns:atom="http://www.w3.org/2005/Atom">
  <state>done</state>
  <atom:link href="http://www.example.org/images/task/1" rel="self"/>
  <message xml:lang="en">Your request has been processed.</message>
</status>
```

❶ 详见处理结果的目标资源

响应码 303 仅指出处理结果存在于 Location 头标识的 URI 中，并不意味着所请求 URI（例如，http://www.example.org/images/task/1）的资源已经被移动到了新的位置。

这个表述告诉客户端它需要访问 http://www.example.org/images/1 以得到处理结果。另一方面，如果服务器处理任务失败，它可以返回如下结果：

```
# 请求
GET /images/task/1 HTTP/1.1
Host: www.example.org

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

<status xmlns:atom="http://www.w3.org/2005/Atom">
  <state>failed</state>
  <atom:link href="http://www.example.org/images/task/1" rel="self"/>
  <message xml:lang="en">Failed to complete the request.</message>
  <detail xml:lang="en">Invalid image format.</detail>
  <completed>2009-09-13T02:10:00Z</completed>
</status>
```

1.11 如何使用 DELETE 方法实现异步删除

23

本节概要性地介绍了一种将 DELETE 用于异步任务的方法，适用于资源删除需要大量时间进行后台清理和归档的情况。

问题描述

您想知道如何实现需要长时间才能完成的 DELETE 请求。

解决方案

在收到 DELETE 请求时，创建一个新资源，返回响应码 202（Accepted），响应中包

含该资源的表述，客户端可以通过该资源来追踪状态。当客户端向任务资源发起 GET 请求时，返回响应码 200（OK），以及一个表示任务当前状态的表述。

问题讨论

实现异步资源删除要比创建或更新资源更容易。以下步骤描述了一种实现方式：

1. 首先，客户端提交请求，删除一个资源。

```
DELETE /users/john HTTP/1.1
Host: www.example.org
```

2. 服务器创建一个新的资源并返回标识任务状态的表述。

```
HTTP/1.1 202 Accepted
Content-Type: application/xml;charset=UTF-8

<status xmlns:atom="http://www.w3.org/2005/Atom">
  <state>pending</state>
  <atom:link href="http://www.example.org/task/1" rel="self"/>
  <message xml:lang="en">Your request has been accepted for processing.</message>
  <created>2009-07-05T03:10:00Z</ping>
  <ping-after>2009-07-05T03:15:00Z</ping-after>
</status>
```

3. 客户端可以查询 http://www.example.org/task/1 以了解删除请求的状态。

可以使用同样的手段，通过 PUT 方法实现异步资源更新。

1.12 何时使用自定义 HTTP 方法

曾经有过数次扩展 HTTP 以加入新方法的尝试，最著名的是 WebDAV（*http://www.webdav.org*）。WebDAV 定义了几个新的 HTTP 方法，如 PROPFIND，PROPPATCH，MOVE，LOCK，UNLOCK 等，用于分布式编著和版本化文档（详见 11.4 节）。其他例子包括用于部分更新的 PATCH（详见 11.9 节）以及用于合并资源的 MERGE（*http://msdn.microsoft.com/en-us/library/cc668771.aspx*）。

问题描述

您想知道使用自定义 HTTP 方法的影响。

解决方案

避免使用非标准的自定义 HTTP 方法，因为引入新方法后，就不能依赖那些只了解标准 HTTP 方法的现有软件了。

您应该设计一个可以抽象此类操作的控制器（详见 2.6 节）资源，并使用 HTTP 的 POST 方法。

问题讨论

扩展 HTTP 方法最重要的好处是，它可以让服务器为扩展方法定义清晰的语义并保持接口一致。但是，除非得到广泛支持，否则扩展方法将会降低互操作性。

例如，WebDAV 将 MOVE 的语义定义为“逻辑上与复制一致，接着是一致性维护处理，最后进行源文件的删除，所有这三个动作是以原子操作的形式来执行的。”任何客户端都可以通过提交 OPTIONS 请求以确认一个 WebDAV 资源是否实现了 MOVE 方法。需要时，如果资源支持 MOVE 方法，客户端可以提交 MOVE 请求把资源从一个地方移动到另一个地方：

```
# 发现所支持方法的请求
OPTIONS /docs/annual_report HTTP/1.1
Host: www.example.org

# 响应
HTTP/1.1. 204 No Content
Allow: GET, PUT, DELETE, MOVE

# 移动
MOVE /docs/annual_report HTTP/1.1
Host: www.example.org
Destination: http://www.example.org/docs/annual_report_2009

# 响应
HTTP/1.1 201 Created
Location: http://www.example.org/docs/annual_report_2009
```

当然也可以遵循 WebDAV 的做法，设计一个新方法，比如 CLONE，创建一个现有的资源的副本： 25

```
# 克隆请求
CLONE /po/1234 HTTP/1.1
Host: www.example.org

# 已创建克隆
HTTP/1.1 201 Created
Location: www.example.org/po/5678
```

客户端会发现服务器支持这一方法，并提交 CLONE 请求。

实际上，代理、缓存及 HTTP 库会将这些方法认定为非幂等、不安全及不可缓存的。

换言之，它们对这样的扩展方法使用和 POST 一样的处理规则，POST 也是非幂等、不安全且在大部分情况下是不可缓存的。这是因为幂等性和安全性是服务器必须保证的。对于未知的自定义方法，代理、缓存和 HTTP 库不能假定服务器提供了这样的保证。因此，对大部分基于 HTTP 的软件和工具，自定义 HTTP 方法等同于 POST 方法。

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

id=urn:example:po:1234

# 已创建克隆
HTTP/1.1 201 Created
Location: www.example.org/po/5678
```

此外，并不是所有的 HTTP 软件（包括防火墙）都支持任意的扩展方法。因此，只有在互操作性不是主要问题时才考虑使用自定义 HTTP 方法。

优先使用 POST 而非自定义 HTTP 方法。不是每一个 HTTP 软件都可以让您使用自定义 HTTP 方法的。使用 POST 是一个更安全的选择。

1.13 何时及如何使用自定义 HTTP 标头

HTTP 服务器使用自定义的 HTTP 标头并不罕见。一些广为人知的自定义 HTTP 标头包括 X-Powered-By，X-Cache，X-Pingback，X-Forwarded-For 及 X-HTTP-Method-Override。HTTP 并不禁止这类扩展标头，但是因客户端和服务器使用自定义标头的方式不同，可能会阻碍互操作。本节讨论何时及如何使用自定义的 HTTP 标头。

26

问题讨论

您想知道使用自定义 HTTP 标头的常见惯例和最佳实践。

解决方案

应该出于传递信息的目的来使用自定义标头。实现客户端和服务器时，要让它们在没有发现需要的自定义标头时也不会失败。

避免使用自定义 HTTP 标头改变 HTTP 方法的行为。限制各种改变 POST 方法行为的

标头。

如果自定义标头中传递的信息对于请求和响应的正确分析至关重要，则应在请求正文、响应正文或请求的 URI 中包含这些信息，避免在这类情形中使用自定义标头。

问题讨论

大部分使用 WordPress 博客平台（*http://wordpress.org*）的站点会在响应中包含以下自定义的 HTTP 标头：

```
X-Powered-By: PHP/5.2.6-2ubuntu4.2
X-Pingback: http://www.example.org/xmlrpc.php
```

这些标头并非 HTTP 的一部分。第一个是 PHP 运行时生成的，WordPress 构建在 PHP 之上，它说明服务器使用了 Ubuntu 上的一个特定版本的 PHP。X-Pingback 标头包含了一个 URI，客户端可以在其他服务器引用这一资源时通过它来通知 WordPress。类似的，HTTP 缓存代理 Squid 使用 X-Cache 标头告知客户端，响应中的表述是由缓存提供的。

这类用法是信息型的，客户端收到这些标头，可以忽略它们而无须担心功能性损失。另一个常见的信息标头是 X-Forwarded-For：

```
X-Forwarded-For: 192.168.123.10, 192.168.123.14
```

这个标头的目的是将请求源传递给服务器。一些代理服务器和缓存服务器使用这个标头将请求的来源报告给服务器。在这个例子中，服务器通过 192.168.123.14 收到来自 192.168.123.10 的请求。如果请求穿透的所有代理和缓存服务器都加入了这个标头，服务器就可以得到客户端的 IP 地址。

虽然一些自定义标头的名字以 X-开头，但是还没有建立标头命名的规范。当您引入自定义标头时，建议使用类似 X-{公司名}-{标头名}这样的风格。

以下自定义 HTTP 标头不是信息型的，可能必须有这样的标头请求或响应才能得到正确的处理：


```
# 资源的一个版本号
X-Example-Version: 1.2

# 客户端的标识符
X-Example-Client-Id: 12345

# 操作
```

```
X-Example-Update-Type: Overwrite
```

请避免这种用法。这削弱了 URI 作为资源标识和 HTTP 方法作为操作的能力。

另一个常见的自定义标头是 X-HTTP-Method-Override，这一标头最早是 Google 用于 Google 数据协议（*http://code.google.com/apis/gdata/docs/2.0/basics.html*）中的。下面是一个例子：

```
# 请求
POST /user/john/address HTTP/1.1
X-HTTP-Method-Override: PUT
Content-Type: application/xml;charset=UTF-8

<address>
  <street>...</street>
  <city>...</city>
  <postal-code>...</postal-code>
</address>
```

这个例子中，客户端使用 X-HTTP-Method-Override 的 PUT 值代替了请求使用的 POST 方法的行为。这一扩展的原理是通过 POST 穿隧 PUT 方法，以使那些配置为阻止 PUT 的防火墙可以允许该请求通过。

除了使用 X-HTTP-Method-Override 覆盖 POST 方法，还可以用一个特别的资源来处理相同的请求，使用的是不带 X-HTTP-Method-Override 的 POST 方法。任何介于客户端和服务器之间的媒介（intermediary）都可能会忽略自定义标头。

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。

2.5 节，“如何支持计算或处理函数”
  使用本节的内容来识别那些实现处理函数（processing function）的资源。

2.6 节，“何时及如何使用控制器来操作资源”
  使用本节的内容来设计控制器资源以完成跨多资源的修改操作。

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）。 33

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

34 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 8.2 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（详见 11.10 节）。

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

35

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。

37

2.5 如何支持计算或处理函数

处理函数并不少见，像 Babel Fish（*http://babelfish.yahoo.com/*）、XE.com（*http://www.xe.com/*）和 Google Maps（*http://maps.google.com*）这样的网站，都会接受一些输入，通过存储在后端服务器上的数据和一些算法来处理这些输入，并返回结果。这些都是处理函数。

问题描述

您希望知道如何为诸如执行计算或验证数据之类的任务提供资源抽象。

解决方案

将处理函数视为一个资源，使用 HTTP GET 来获取表述，其中包含处理函数输出。使用查询参数来为处理函数提供输入。

问题讨论

REST 只适用于应用程序领域中的“事物”或“实体”资源，这是对 REST 架构约束最常见的理解之一。尽管在很多情况下这个观点是对的，但在涉及处理函数的场合中似乎并非如此。看看下面这些例子:

两地距离

客户端向服务器提交了两个地点的经度和纬度，服务器计算两地距离并将其返回给客户端。

行驶方向

客户端在表单中自由提交两个地点，比如"Seattle, WA"和"San Francisco, CA"，服务器返回一个列表，其中包含行驶路段和转弯方向。

验证信用卡

客户端向服务器提交信用卡信息，例如持卡者姓名、卡号和有效期，服务器返回该卡是否为有效信用卡。

这些例子都有一个共同的特点，如果使用 2.1 节中的内容，您会找到一些名词，但却无法方便地对其应用统一接口。例如，如果将每个"地点"标识为资源，您找不到与"计算距离"等效的 HTTP 操作。

38 一种解决方法是将处理函数本身视为资源。在第一个例子中，可以将距离计算器视为一个资源，而距离是它的表述。下面的请求和响应就是一个例子：

```
# 请求
GET /distance_calc?lats=47.610&lngs=-122.333&late=37.788&lnge=-122.406 HTTP/1.1
Host: www.example.org

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

<result xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="self"
    href="http://www.example.org/distance_calc?lats=47.610&
          lngs=-122.333&late=37.788&lnge=-122.406"/>
  <distance unit="miles">808.0</distance>
</result>
```

类似的，"方向定位器"、"兴趣点（point of interest[注1]）定位器"和"信用卡验证器"可以将"方向"、"兴趣点"和"验证结果"作为资源的表述：

```
# 寻找方向的请求
GET /directions?from=Seattle,WA&to=San%20Francisco HTTP/1.1
Host: www.example.org

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

注 1： Point of Interest，简称 POI，主要用于绘图、GIS 和 GPS 方面，大意就是受关注的地理信息点，详见 Wikipedia，http://en.wikipedia.org/wiki/Point_of_interest。

```
<directions>
  <step>
    ...
  </step>
  <step>
    ...
  </step>
</directions>

# 寻找兴趣点的请求
GET /poi?lat=47.610&lng=-122.333 HTTP/1.1
Host: www.example.org

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

<atom:feed xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:title>Points of Interest</title>
  <atom:link href="http://www.example.org/poi?lat=47.610&lng=-122.333/" rel="self"/>
  <atom:updated>2009-10-01T18:30:02Z</atom:updated>
  <atom:author>
    <atom:name>All Names Made Up Inc.</atom:name>
  </atom:author>
  <atom:id>urn:uuid:60a76c80-d399-11d9-b93C-0003939e0af6</atom:id>
  <atom:entry>
    <atom:id>urn:example:poi:0012</atom:id>
    <atom:title>...</atom:title>
    <atom:updated>2009-09-13T18:30:02Z</atom:updated>
    <atom:link rel="alternate" href="http://www.example.org/poi/0012.html"/>
    <atom:content type="text">...</atom:content>
  </atom:entry>
  ...
</atom:feed>

# 验证信用卡的请求（通过 HTTP 发送）
GET /validate?ccnum=1234567890123456 HTTP/1.1
Host: www.example.org

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

invalid
```

因为所有这些方法都是安全且幂等的，所以 GET 是最适合实现处理函数的 HTTP 方法。

把类似 https://www.example.org/validate 这样的 URI 用来表示“操作”，这破坏了统一接口。但是 URI 仅仅是在标识资源，就 HTTP 而言，URI 的语法并不会有什么妨碍。

2.6 何时及如何使用控制器来操作资源

就 RESTful Web 服务而言，控制器能帮助提升服务器与客户端的独立程度，改善网络效率，并让服务器以原子操作的形式来实现复杂操作。

问题描述

您想知道如何编写以原子方式修改多个资源的操作，或者无法明显映射到 PUT 或 DELETE 方法的操作。

解决方案

为每个独立的操作指定一个控制器资源。让客户端使用 HTTP 的 POST 方法提交请求，触发操作。如果操作的结果是创建一个新资源，返回响应码 201（Created），并附有 Location 头，指向新创建的资源。如果操作的结果是修改一个或多个现有资源，返回响应码 303（See Other），附有一个带 URI 的 Location 头，客户端能通过它来获取被修改资源的表述。如果服务器无法提供单个 URI 指向所有被修改的资源，则返回响应码 200（OK），在正文中附上一个表述，客户端能用它来了解操作结果。关于错误处理，请详见 3.13 节。

40

问题讨论

控制器是一种能以原子方式修改资源的资源。从您的领域模型中也许看不到对这种资源的需求，但它能帮助服务器抽象复杂的业务操作，并为客户端提供触发这些操作的途径。这能降低客户端与服务器的耦合性。

假设要为用户合并两个地址簿，移动电话上的客户端需要与服务器的地址簿同步所有联系人。一种方法是像下面这样使用 PUT 请求：

1. 向地址簿资源提交一个 GET 请求，从服务器下载完整的地址簿。
2. 加载本地联系人列表，与服务器下载的地址簿合并。
3. 向地址簿资源提交一个 PUT 请求，将整个地址簿替换为合并后的新地址簿。

这样能够完成任务，但会有一些限制。在客户端下载完整地址簿，与本地联系人列

表合并，这使得客户端无法有效利用网络。而且，一些用户在服务器上的地址簿可能会很大，并非所有字段都与客户端有关。客户端可能没有足够的计算能力来处理合并操作。更重要的是，合并地址簿内容的应用逻辑应该属于服务器，而非客户端。期望客户端来处理这个任务会造成代码重复，并且无法有效分离关注点。

下面是另一种做法：

1. 从服务器获取地址簿中的每个地址。
2. 如果该地址能与本地存储中的某个地址吻合，则进行合并，通过提交 PUT 请求更新它。
3. 如果在本地存储中有服务器上不存在的新联系人，向地址簿提交一个 POST 请求进行添加。

这个方法在网络交互上有缺陷，同样不适用于像移动电话这样的客户端的受限环境。

更有效的解决方案是使用控制器资源来解决这个问题。针对这个例子，设计一个控制器资源，允许客户端向服务器提交用于合并的地址簿。

```
# 合并地址簿的请求
POST /user/smith/address_merge HTTP/1.1
Host: www.example.org
Content-Type: text/csv;charset=UTF-8

John Doe, 1 Main Street, Seattle, WA
Jane Doe, 100 North Street, Los Angeles, CA
...

# 响应
HTTP/1.1 303 See Other
Location: http://www.example.org/user/smith/address_book
Content-Type: text/html;charset=UTF-8

<html>
  <body>
    <p>See <a href="http://www.example.org/user/smith/address_book">address
    book</a> for the merged address book.</p>
  </body>
</html>
```

41

合并后，服务器将客户端重定向到用户更新后的地址簿。如果需要，客户端可以获取一份副本。

下面是另一个例子。一个书店里，店员想要将一本书的税前价格下降 15%，并修改

税后价格来反映这个折扣。服务器可以将折扣百分比作为资源，客户端提交 PUT 请求来修改当前折扣。在同一请求中，服务器还可以修改图书总价。

```
# 更新折扣的请求
PUT /book/1234/discount HTTP/1.1
Host: www.example.org
Content-Type: application/x-www-urlencoded

value=15

# 响应
HTTP/1.1 204 No Content
```

现在，考虑一下，客户端想要提供该书的 30 天免费在线版本，并带有 15%折扣。服务器上维护了一个集合，其中包含所有正在提供 30 天免费访问的图书，客户端可以提交 POST 请求将该书添加到集合中。

```
# 向电子书列表添加图书的请求
POST /30dayebookoffers HTTP/1.1
Host: www.example.org
Content-Type: application/x-www-urlencoded

id=1234&from=2009-10-10&to=2000-11-10

# 响应
HTTP/1.1 201 Created
Location: http://www.example.org/30dayebookoffer/1234
Content-Length: 0
```

42 如果业务上要求以原子方式完成这两个修改，那么可以使用控制器资源。

```
# 添加折扣及 30 天免费访问的请求
POST /book/1234/discountebookoffer HTTP/1.1
Host: www.example.org
Content-Type: application/x-www-urlencoded

id=1234&discount=15&ebook_from=2009-10-10&ebook_to=2000-11-10

# 响应
HTTP/1.1 303 See Other
Location: http://www.example.org/book/1234
Content-Length:0

# 获取更新后的图书的请求
GET /book/1234 HTTP/1.1
Host: www.example.org

# 响应
```

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

<book xmlns:atom="http://www.w3.org/2005/Atom">
  <id>urn:example:book:1234</id>
  <atom:link rel="self" href="http://www.example.org/book/1234"/>
  ...
  <discount>15</discount>
  ...
  <atom:link rel="http://www.example.org/rels/offer"
        href="http://www.example.org/30dayebookoffer/1234"/>
  ...
</book>
```

在响应中，服务器包含了一个链接以便客户端能发现30天免费电子书优惠。如果客户端为用户提供了用户界面，那么它可以提供一个指向该电子书的链接，将用户引导过去。

从这些例子中可以发现一个关键点，即您可能会在将应用程序中的操作映射为统一接口方法时遇到困难。例如，在打折的例子中，服务器将当前折扣值作为资源，这样客户端能使用PUT请求来更新它。类似的，服务器将30天免费电子书优惠标识为一个集合，让客户端使用POST请求向集合中添加新书。但要把两个任务合并成单个请求时，并不能明确地将之映射到任意一个HTTP方法上。控制器在这些情况下是最合适的。

针对与上述例子类似的用例，不要直接在图书资源上使用POST方法，因为这会导致穿隧。当客户端针对不同动作在单一URI上使用相同方法时就会出现穿隧。例如下面这个例子：

43

```
# 添加折扣的请求
POST /book/1234 HTTP/1.1
Host: www.example.org
Content-Type: application/x-www-urlencoded

op=updateDiscount&discount=15

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

<book xmlns:atom="http://www.w3.org/2005/Atom">
  <id>urn:example:book:1234</id>
  <atom:link rel="self" href="http://www.example.org/book/1234"/>
  ...
  <discount>15</discount>
  ...
```

```
</book>

# 添加30天免费电子书的请求
POST /book/1234 HTTP/1.1
Host: www.example.org
Content-Type: application/x-www-urlencoded

op=30dayOffer&ebook_from=2009-10-10&ebook_to=2000-11-10

# 响应
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8
<book xmlns:atom="http://www.w3.org/2005/Atom">
  <id>urn:example:book:1234</id>
  <atom:link rel="self" href="http://www.example.org/book/1234"/>
  ...
  <atom:link rel="http://www.example.org/rels/offer"
        href="http://www.example.org/30dayebookoffer/1234"/>
</book>
```

在请求中，参数 op=updateDiscount 和 op=30dayOffer 用来标识操作。这就会导致穿隧。

穿隧会降低协议层面上的可见性（详见 1.1 节），因为请求的可见部分（例如请求URI、使用的 HTTP 方法、请求头和媒体类型）并不会明确地描述操作。

要不惜一切代价避免穿隧，针对每个操作使用不同的资源（例如控制器）。

第3章

设计表述

45

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，“如何保持同构的集合”
以本节的内容为指导方针，了解如何保持容易迭代的集合。

3.9 节，“如何在表述中使用可移植的数据格式”
通过本节了解表述中针对格式化数字、日期、时间、货币等内容的互操作方法。

3.10 节，“何时使用实体标识符”
虽然 URI 是资源的唯一标识符，但有时使用实体标识符能帮助改善互操作性。通过本节的内容可以了解其中的原因。

3.11 节，“如何在表述中编码二进制数据”
有时您不得不和二进制数据打交道。通过本节了解如何在表述中使用分段（multipart）媒体类型来编码二进制数据。

3.12 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。

3.13 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。

3.14 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。

47

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。

48

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
```

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>
```

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。

```
    <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。

49

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（详见第 7 章）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。我们会在第 9 章讨论这个标头。

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。

50

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。详见 7.1 节了解如何告知服务器客户端可以处理哪些表述类型。

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。

51

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 ?Ǩs 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。

52 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。

53

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
application/x-www-form-urlencoded	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。 55

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。

56 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，详见第 5 章），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（详见 3.10 节）。

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>
```

57

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 遝 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（详见 5.2 节），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（详见 3.10 节）。

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"
    }
    ...
    }
}
```

59

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。

PDG

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:

60

```
# 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。

61

3.8 如何保持同构的集合

依据使用场合，您可以通过使用相似度（using similarities）把资源组合成集合。但是，无论选择了什么条件，保持同构的（homogeneous）表述都是很重要的，这能为客户端带来一些便利。

问题描述

您想知道如何为集合设计表述格式，使集合中的成员看上去并不完全一样。

解决方案

在设计集合的表述时，让集合中的成员在结构和语法上是相似的。

问题讨论

在为集合设计表述格式时，只包含成员资源同构的部分。举例来说，如果您的产品集合可以囊括汽车、船和摩托车，那么在产品集合里只包含资源中的特定产品细节。请注意，集合是聚集某种意义上相似的资源的一种手段，当包含了那些同一集合里其他资源没有的信息时，这通常是一个糟糕的抽象。下面的例子就是一个糟糕的抽象：

```
<!-- 避免这种做法 -->
<products xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="self" href="http://www.example.org/catalog/products"/>
  <!-- 第一个成员是一辆汽车。-->
  <automobile>
    <id>9001</id>
    <atom:link rel="self" href="http://www.example.org/catalog/product/9001"/>
    <make>Smart</make>
    <model>Fortwo Convertible</model>
    <year>2009</year>
    <class classid="small">Small Car</class>
    <mpg>
       <city>33</city>
       <highway>41</highway>
    </mpg>
    <drivetrain>2WD</drivetrain>
    <list-price currency="USD">19495</list-price>
  </automobile>
  <!-- 第二个成员是一艘船！ -->
  <sailboat>
    <id>10101</id>
    <atom:link rel="self" href="http://www.example.org/catalog/product/10101"/>
    <make>Jeanneau</make>
    <model>Sunfast 3200</model>
    <year>2008</year>
    <length unit="ft">32</length>
    <hull-type>fiberglass</hull-type>
    <number-of-engines>1</number-of-engines>
    <list-price currency="USD">95995</list-price>
  </sailboat>
</products>
```

62

在这个例子中，虽然汽车和船共享了一些公共属性，但它们还包含了每个产品特有的属性。客户端应用程序在迭代这样的集合时，那些特殊的属性是没有意义的，客户端有可能无法处理此类表述。要避免这样的表述，保持集合的同构性，像下面这样：

```
<products xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="self" href="http://www.example.org/catalog/products"/>
  <product type="automobile">
```

```
    <id>9001</id>
    <atom:link rel="self" href="http://www.example.org/catalog/product/9001"/>
    <make>Smart</make>
    <model>Fortwo Convertible</model>
    <year>2009</year>
    <list-price currency="USD">19495</list-price>
  </product>
  <product type="sailboat">
    <id>10101</id>
    <atom:link rel="self" href="http://www.example.org/catalog/product/10101"/>
    <make>Jeanneau</make>
    <model>Sunfast 3200</model>
    <year>2008</year>
    <list-price currency="USD">95995</list-price>
  </product>
</products>
```

比起先前的例子，同构的形式对客户端来说更方便一些。

3.9 如何在表述中使用可移植的数据格式

有不少方法可以在表述中编码日期、时间、国家、数字和时区。举例来说，您可以用 UNIX 日期格式、UNIX 时间戳（UNIX epoch time）或 MM-DD-YYYY 和 DD-MM-YYYY 这样的纯文本来格式化日期-时间（date-time）值。大多数日期-时间格式要求客户端和服务器的时钟同步，或者依赖本地时间。由于时钟、时区甚至是夏令时的不同，都会让这些格式造成互操作性的问题。类似的，货币和数字格式在不同国家之间也不一样，就算使用了可移植的数据格式，为一个国家的客户端设计的表述，也可能无法与另一个国家的客户端或服务器进行互操作。

问题描述

63

您想知道如何为日期、时间、数字、货币等内容选择合适的格式。

解决方案

除非文本要呈现给最终用户，否则尽量避免使用特定语言、地区或国家的格式或者格式标识符。应该使用如下的可移植格式取而代之：

- 使用 W3C XML Schema 中为格式化数字（包括货币）而定义的小数、浮点数和双精度浮点数的数据类型。
- 针对国家和从属地区，使用 ISO 3166 代码。

- 针对货币，使用 ISO 4217 字母或数字代码。
- 针对表述中使用的日期、时间和日期-时间值，使用 RFC 3339 规范。
- 使用 BCP 47 语言标签来表示文本语言。
- 使用 Olson 时区数据库中的时区标识符来表示时区。

问题讨论

为数据选择可移植的格式可以消除互操作性错误。下面的范例展示了一些常用格式，请注意，您的应用程序领域中可能会有额外的数据类型，它们并未包含在下面。在发明自己的标准前，请先查看行业或领域特定的标准。

数字

XML Schema 规定的数字格式是独立于语言和国家的，因此它们是可移植的:

```
123.456    +1234.456    -1234.456    -.456,123.
```

但是，像下面这样的格式是不可移植的:

```
1,234,567    12,34,567    1,234
```

国家和地区

ISO 3166-1，即 ISO 3166 的第一部分，规定了双字母的国家代码，例如 US 是美国、DK是丹麦、IN是印度等。

ISO 3166-2，即 ISO 3166 的第二部分，规定了国家细分地区（比如州和省）的代码。例如 US-WA，US-CO，CA-BC，IN-AP 等。

64

货币

ISO 4217 为货币名称规定了三个字母的货币代码。其中的前两个字母是 ISO 3166-1 中的双字母国家代码，第三个字母通常是货币的首字母。例如 USD 是美元（U.S. dollar）、CAD是加拿大元（Canadian dollar）、DKK是丹麦克朗（Danish krone）。这些代码表示了货币的估价和兑换。在货币上使用这些代码消除了货币名称（比如 dollar）或符号（比如$）的歧义。

日期和时间

RFC 3339 是 ISO 8601 的一个子集，是一个使用格里历（Gregorian calendar）[注2]来表

注 2: 又称格列高利历，世界绝大部分通用的阳历，由教皇格列高利十三世于 1582 年倡导使用，为朱利安历法的改进版。

示日期和时间的标准。RFC 3339 格式化后的日期、时间和日期-时间值有以下特点:

- 可以用字符串排序的方法来比较两个值。
- 这种格式是易于人们阅读的。
- 日期既可以是协调世界时间(Coordinated Universal Time, UTC),也可以是 UTC 的一个偏移量,这避免了与时区和夏令时相关的问题。

下面是经过适当格式化后的日期、时间和日期-时间的例子:

```
2009-09-18Z
23:05:08Z
2009-09-18T23:05:08Z
2009-09-18T23:05:08-08:00
```

W3C XML Schema 中的 `date`, `time` 和 `dateTime` 数据类型遵循 RFC 3339, 可以使用支持这些数据类型的库来读取、解析这些值。

语言标签

BCP 是“Best Current Practice”的简称, BCP 47 目前指的是 RFC 5646 和 RFC 5645, 其中定义了诸如 HTML 的 `lang` 和 XML 的 `xml:lang` 这样的语言标签。例如, `en` 是英语、`en-CA` 是加拿大英语, 以及 `ja-JP` 是日本本土使用的日语。

时区标识符

Olso 时区数据库提供了时区名称的一个统一约定, 还包含了与时区相关的数据。该数据库说明了时区、季节变换(比如夏令时), 甚至是历史时区变换。大多数编程语言提供的时区类/辅助工具都支持该数据库, 包括 Java 的 `java.util.Timezone`、Ruby 的 `TZInfo`、Python 的 `tzinfo` 和 C#的 `System.TimeZoneInfo`。

3.10 何时使用实体标识符

对于 RESTful Web 服务而言, URI 是资源的唯一标识符。但是, 应用程序代码通常要处理领域实体的标识符。当客户端或服务器是一个更大的异构应用程序集合的一部分时, 来自资源的信息可能会跨过多个系统边界, 实体标识符可以用于交叉引用或转换数据。

问题描述

您想知道何时要包含实体标识符, 将它连同资源 URI 一起放在表述里。

解决方案

对于资源表述中的每个应用程序领域实体，都要包含 URN 格式的标识符。

问题讨论

尽管 URI 唯一标识了资源，但遇到下列情况时实体标识符还是很有用的：

- 您的客户端和服务器是一个更大的环境中的一部分，这个环境包含使用 RPC、SOAP、异步消息、存储过程甚至第三方应用程序的应用程序，实体标识符可能是那些系统中唯一的公共命名者（denominator），用于提供一致的数据标识符。
- 客户端和服务器可以存储自己的实体副本，无须从资源 URI 中进行解码或者把 URI 用做数据的键。尽管这样做不符合标准，但 URI 还是有可能会发生变化的。客户端可以使用这些标识符来交叉引用来自不同表述的多个实体。
- 并非应用程序领域中所有的实体都映射到资源上时，实体标识符可以为表述中的数据提供唯一标识符。

就算您用唯一 URI 把应用程序中所有的实体都映射到资源上，在表述中包含实体标识符也会让您的应用程序为未来与非 HTTP 的 Web 服务集成做好准备。要维护标识符的唯一性，可以考虑将它们格式化成 URN。

下面是一个例子，用户资源在数据库中的标识符是 1234，用户的地址是 4567：

```
<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 Doe</name>
  <address>
    <id>urn:example:address:4567</id>
    <street>1 Main Street</street>
    <city>Seattle</city>
    <state>WA</state>
  </address>
</person>
```

66

3.11 如何在表述中编码二进制数据

并不是每个表述都能完全依赖于 XML 和 JSON 这样的文本格式的，一些表述可能需要在文本中包含二进制数据。这样的例子包括电影编目中的视频预览，或者音乐商店中的音频试听表述的封面图片。

问题描述

您想知道如何在包含文本数据的表述中编码二进制数据吗？

解决方案

使用 multipart/mixed，multipart/related 或 multipart/alternative 这样的分段媒体类型。避免使用 Base64 编码将二进制数据与文本格式编码在一起。

问题讨论

分段消息让您能把不同格式的数据整合到一个 HTTP 消息中。一个分段消息包含多个消息部分，由边界进行分割，每个部分都可以包含不同媒体类型的消息。下面是一个例子：

```
Content-type: multipart/mixed; boundary="abcd"

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

<movie> ... </movie>
--abcd
Content-type: video/mpeg

... 这里是视频 ...

--abcd--
```

该分段消息有两个部分，一部分包含了一个 XML 文档，另一部分包含一个视频。针对此类用例，可以考虑表 3-2 中列举的分段媒体类型。

表 3-2：使用分段媒体类型

媒体类型	用法
multipart/form-data	把键值对数据与任意媒体类型的数据编码在一起。与您用 HTML 表单上传文件时的用法一致
multipart/mixed	将多个任意媒体类型的部分打包在一起。在先前的例子中，分段消息把 application/xml 表示的电影元数据和 video/mpeg 表示的视频整合进了一个 HTTP 消息中
multipart/alternative	在使用不同媒体类型发送同一资源的可选表述时，使用该媒体类型。最好的例子是以纯文本（媒体类型是 text/plain）和 HTML（媒体类型是 text/html）来发送电子邮件
multipart/related	当消息的各个部分是互相关联的，而且需要一起处理这些部分时，使用该媒体类型。第一部分是根，可以通过 Content-ID 头引用其他部分

在某些编程语言中，创建及解析分段消息十分麻烦，而且很复杂。作为变通，不在表述中包含二进制数据，而是提供一个链接，把二进制数据当做单独的资源来获取。举例来说，在前面的例子中，可以提供一个指向视频的链接。

3.12 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：

68

```
<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>
```

注 3：RDFa（Resource Description Framework in attributes）是一个 W3C 推荐标准。详见 Wikipedia：http://en.wikipedia.org/wiki/RDFa。

```
        <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>
```

69

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。

3.13 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

70

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（详见 5.3 节），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

71

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，就像第12章中讨论的那样。

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（详见14.2节）。

406（Not Acceptable）

详见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。

72

412（Precondition Failed）

详见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：

73

```
# 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>
```

在生成该消息正文时，要遵循第 7 章中讨论的内容。

3.14 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 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 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。详见 1.5 节中的例子。

PDG

第 4 章

设计 URI

75

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 节，“如何设计 URI”
通过本节了解一些常用的 URI 设计惯例。

4.2 节，“如何将 URI 用做模糊标识符”
通过本节了解一些该做的和不该做的事，以便让 URI 始终是模糊标识符（opaque identifier）。

4.3 节，“如何让客户端将 URI 视为模糊标识符”
将 URI 视为模糊标识符有助于客户端与服务器的解耦。本节展示了一些技术，服务器可以运用它们来帮助客户端将 URI 视为模糊标识符。

4.4 节，“如何保持酷的 URI”
既然 URI 是客户端与服务器之间接口的重要部分，那保持“酷”的 URI 就很重要了，所谓“酷”，指的就是 URI 的稳定性和永久性。通过本节您将了解到一些相关的实践。

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

76

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。

77

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。

78

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>
```

79

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“Hadron+Supercollider”。

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。

4.2 如何将 URI 用做模糊标识符

大多数情况下，将 URI 用做模糊标识符是很常见的事。只需保证每个资源都有一个不同的 URI 就可以了。但是，本节中的一些实践会造成 URI 重载。在这些情况下，URI 可能会成为未指定信息和动作的通用网关。这会导致错误的缓存响应，甚至会泄露那些没有经过身份验证就不该被共享的安全数据。

问题描述

您想知道如何避免出现阻碍 URI 成为唯一标识符[注1]的情况。

注 1：本章中的“唯一标识符”（unique identifier）和“模糊标识符”（opaque identifier）基本是一个意思。

PDG

解决方案

只使用 URI 来决定哪个资源来处理请求。

不要通过使用相同 URI 的 POST 请求来穿隧重复的状态变更，也不要使用自定义头来重载 URI。只为信息性目的（informational purposes）使用自定义头。

问题讨论

80

将 URI 作为唯一资源标识符是一种很直接的做法，但当重载某些 HTTP 方法或者用 URI 以外的东西来决定如何处理请求时情况就不同了。

下面是一个使用自定义 HTTP 头来决定返回结果的例子：

```
# 请求
GET /news HTTP/1.1
Host: www.example.org
X-Filter: science;sports;weather

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

... 消息正文 ...
```

在这个例子中，通过 X-Filter 头的内容重载了 URI http://www.example.org/news。如果另一个客户端发起了一个相似的请求，但自定义头里放了另一个值（例如，politics;economy;healthcare），服务器将返回另一个资源的表述。

这些实践是很容易避免的，在这个例子里，服务器应该为不同的新闻过滤器提供不同的 URI。

另一个将 URI 作为网关而非唯一标识符的常用实践，是用 POST 请求穿隧重复的状态变更。包括 ASP.NET，JavaServer Pages 和一些 Ajax 工具在内的多个 Web 框架都将其作为默认实践：

```
# 请求
POST /ajax-endpoint HTTP/1.1
Host: www.example.org

<request>
  <filter>science</filter>
  <filter>sports</filter>
  <filter>weather</filter>
</request>
```

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

... 消息正文 ...

# 请求
POST /ajax-endpoint HTTP/1.1
Host: www.example.org

<request>
  <filter>politics</filter>
  <filter>economy</filter>
  <filter>healthcare</filter>
</request>

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

... 消息正文 ...
```

81

由于人们通常是将 HTTP 作为一种传输协议，所以才有了这样的实践。只要避免这些实践，将 URI 作为唯一标识符，事情会相对容易一些。

4.3 如何让客户端将 URI 视为模糊标识符

无论如何设计 URI，重要的是让客户端尽可能地将 Web 服务 URI 视为模糊标识符。客户端应该能在无须理解服务器 URI 结构的情况下，使用服务器提供的 URI 发送额外的请求。

问题描述

您想知道如何确保客户端将 URI 视为模糊标识符。

解决方案

无论何时，尽量在运行时使用表述正文（见 5.1 节和 5.2 节）或 HTTP 标头（见 5.3 节）中的链接来提供 URI。

无法提供可能的 URI 全集时，可以考虑使用 URI 模板（见 5.7 节），或事先约定规则让客户端通过编程的方式构造 URI。

问题讨论

REST 和 HTTP 的架构约束都没有要求客户端将 URI 视为模糊标识符，但这么做可

以降低服务器与客户端的耦合度。服务器期望客户端通过返回的表述中的少量信息，或离线知识（例如文档或逆向工程）中的少量信息来构造 URI，但这会带来紧耦合性。在 Web 服务改变创建新 URI 的方式时，这种耦合将使现有客户端无法工作。

大多数情况下，创建 URI 的过程属于服务器，而非客户端。例如，一个照片分享 Web 服务，返回最近上传到服务器的照片列表。

```
<?xml version="1.0" encoding="utf-8" ?>
<photos>
  <photo>
      <id>nj1-1234</id>
      <user-id>987</user-id>
      <server-id>east-nj1</server-id>
  </photo>
   <photo>
      <id>nj4-1235</id>
      <user-id>988</user-id>
      <server-id>east-nj4</server-id>
    </photo>
  ...
</photos>
```

82

因为这个表述中没有提供 URI，该 Web 服务的客户端必须读取文档，编写客户端代码以编程方式来创建每张照片的 URI。

```
http://east-nj1.photos.example.org/987/nj1-1234
http://east-nj4.photos.example.org/988/nj4-1235
```

这些 URI 中包含了实现层面的数据，例如服务器名、照片 ID 和用户 ID。如果服务器发生了会修改所有新照片 URI 的架构变更，客户端也要对创建 URI 的方式做相应变化。

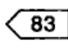
当您的 Web 服务要求客户端基于其实现细节来创建 URI，这些细节将变成 Web 服务公共接口的一部分。尽量避免或减少向客户端泄露这些实现细节。

为了将客户端从这些实现细节中解耦出来，服务器可以在表述中提供链接。

```
<?xml version="1.0" encoding="utf-8" ?>
<photos xmlns:atom="http://www.w3.org/2005/Atom">
  <photo>
    <atom:link href="http://east-nj1.photos.example.org/987/nj1-1234"
          rel="alternate"
          title="Sunset view from our backyard"/>
    <atom:link href="http://east-nj1.photos.example.org/987"
```

```
          rel="http://www.example.org/rels/owner"/>
    <id>nj1-1234</id>
  </photo>
  <photo>
    <atom:link href="http://east-nj4.photos.example.org/988/nj4-1235"
          rel="alternate"/>
    <atom:link href="http://east-nj1.photos.example.org/988"
          rel="http://www.example.org/rels/owner"/>
    <id>nj4-1235</id>
  </photo>
  ...
</photos>
```

这个表述中直接用链接将实现细节编码进 URI。其中的每张照片都有一个获取图像文件的链接和另一个获取照片拥有者资源的链接。要弄清楚哪个链接指向何处，客户端不需要知道 URI 是如何构造的，只要理解 `rel` 属性值的含义即可。


83

请注意，要求客户端将 URI 视为模糊标识符可能需要在性能上做些权衡。通常 URI 在长度上要比数据库标识符更长，因此在网络中传输 URI 会增加消息大小。要在表述中需要传递大量 URI 时可能会有些麻烦。

当 Web 服务无法在表述中为客户端提供所有可能 URI 的列表时（例如，支持特殊查询），可以使用“半模糊”的 URI 模板（详见 5.7 节）。如果您想对 URI 做数字签名以防御请求篡改（详见 12.5 节）或加密部分 URI 以保护敏感信息，那也需要放宽或忽略不透明性。出于这个目的，客户端与服务器需要事先交换 URI 签名的细节。

4.4 如何保持酷的 URI

设计 URI 时，应该让它能保持很长一段时间。客户端可以在数据库和配置文件中存储 URI，甚至将其硬编码在代码中。实际上，Web 是工作在 URI 永久性的假设下的。这个设计原则可以归诸于“酷的 URI 是不会改变的”这条公理（*http://www.w3.org/rovider/tyle/URI*）。当服务器决定改变 URI 时，客户端就无法工作了。酷的 URI 是那些永远不变的 URI。

当您的 Web 服务工作在私有网络和受控网络中时，URI 变更的影响可能是微不足道的。但是，URI 是客户端与服务器间接口的重要部分，对 URI 的变更注定会破坏它。本节展示了如何保持 URI 的永久性。

问题描述

您想知道如何支持“酷的 URI 是不会改变的”这条公理。

解决方案

基于稳定的概念、标识符和信息来设计 URI。在服务器上使用重写规则为客户端屏蔽实现级别的变更。在 URI 必须改变的情况下（例如，合并两个应用程序、较大的重新设计等），可以保留旧 URI，使用响应码 301（Moved Permanently）向客户端发出带有新 URI 的重定向请求，或者在少数情况下，对于那些不再有效的 URI 可以返回响应码 410（Gone）。

如果用于 URI 的概念或标识符出于业务、技术或安全原因不能永久固定，那 URI 也不能持久。5.6 节中有应对之策。

84

问题讨论

URI 的永久性依赖于用于创建 URI 的概念和标识符的稳定性和永久性。例如，一篇标题为“My Trip Report”的文档的 URI 是 `http://www.example.org/2009/11/my_trip_report`，只要文档发布后服务器将其标题视为不可变更的，那该 URI 就是稳定的。通常用于存储资源数据的唯一标识符能帮助设计稳定的 URI，这些标识符很少发生变化。

就算用于创建 URI 的概念或标识符发生变化，也可以通过 Web 服务器所支持的重写规则来隐藏这些变化，例如 Apache 的 mod_rewrite（*http://httpd.apache.org/ocs/.0/mod/mod_rewrite.html*）和 Internet Information Services（IIS）的 URLRewrite（*http://www.iis.net/extensions/URLRewrite*）。您可以使用这些 Web 服务器扩展来隐藏由服务器应用程序合并、修改路径等情况引起的 URI 变更。

如果无法隐藏 URI 变更，对于所有发往旧 URI 的请求都予以 301（Moved Permanently）响应并在 Location 头中放入新 URI:

```
# 请求
GET /users/1 HTTP/1.1
Host: www.example.org
Accept: application/json

# 响应
HTTP/1.1 301 Moved Permanently
Location: http://www.example2.org/users/1
```

当客户端收到响应码 301（Moved Permanently）后，应该从客户端的本地存储中删除旧 URI 的所有副本，用新 URI 来替换它们。这将减少客户端随后的重定向数量。

不要禁用客户端应用程序的重定向支持。反而应该考虑为客户端可进行的重定向设定一个合理的数量限制。并验证 Location URI 映射到了一个可信的域或 IP 地址。禁用重定向将在服务器决定修改 URI 时影响客户端的使用。

一旦设置了重定向，需要在服务器上监控旧 URI 的请求流量。在能确定主要客户端已更新了存储的链接，指向新 URI 之前，要保持旧 URI 的重定向服务。当无法监控旧 URI 时，应为旧 URI 建立并传递一个适当的终身策略。

一旦流量下降或超过事先约定的时间点，将响应码 301（Moved Permanently）转为 410（Gone）或 404（Not Found）。还要包含一个消息正文，指明如何找到新的或相关的资源。

```
# 请求
GET /users/ HTTP/1.1
Host: www.example.org
Accept: application/xml;charset=UTF-8

# 响应
HTTP/1.1 410 Gone
Content-Type: application/xml;charset=UTF-8;
Expires: Sat, 01 Jan 2011 00:00:00 GMT

<error xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="help" href="http://www.example2.org"/>
  <message xml:lang="en-US">This resource no longer exists.
  Related information may be found at http://www.example2.org</message>
</error>
```

85

请注意，上面例子中的 410（Gone）响应里用指向未来时间[注2]的 Expires 头做了标记。欲了解更多缓存响应的内容，请见第 9 章。

注 2：　原书出版时，2011 年 1 月 1 日是未来时间，中文版出版时应该已经不是了。

第 5 章

87

Web 链接

链接提供了一种方式，可以从一个资源导航到另一个资源。日常生活中有很多链接的例子。旅行者使用路牌和地图来决定走哪条路。书籍和文章使用脚注和参考文献把读者引向相关的资料。在软件方面，我们使用变量和指针在应用程序的不同环节之间建立链接。

万维网基于同样的原则。HTML 文档使用锚（anchor）和表单让用户穿梭于网页之间，利用 img，object 和 link 元素来引入相关资源。以下是一个 HTML 文档形式的资源表述的正文：

```
<html>
  <head>
    <link href="http://www.restful-webservices-cookbook.org/styles/main.css"
          rel="stylesheet" type="text/css"/>
    <link href="http://www.restful-webservices-cookbook.org/feed"
          rel="alternate feed" type="application/atom+xml"/>
  </head>
  <body>
    <p><img src="http://www.restful-webservices-cookbookorg/images/cover"
        align="left"/>Read <a href="http://www.restful-webservices-cookbook.org">
        RESTful Web Services Cookbook</a> to learn about building RESTful apps.
    </p>
  </body>
</html>
```

这个例子中的每个 link 元素都指向一个相关的资源。浏览器可以通过第一个 link 元素找到与此 HTML 文档关联的样式表。Feed 阅读器可以通过第二个 link 来获取相关的 Atom Feed。img 元素指向另一个相关的资源——图像文件，浏览器可以将它呈现在屏幕上。锚（a）元素为客户端提供了导航到另一个页面的途径。

88

本章将讨论以下内容，演示在 RESTful Web 服务中何时以及如何使用链接。

5.1 节，“如何在 XML 表述中使用链接”

通过本节了解如何将链接嵌入到通用的 XML 表述中。

5.2 节，“如何在 JSON 表述中使用链接”

通过本节了解如何将链接嵌入到 JSON 表述中。

5.3 节，“何时以及如何使用链接标头”

链接标头（link header）以一种格式独立的方式来提供链接，通过本节了解何时以及如何使用它们。

5.4 节，“如何分配链接关系类型”

缺少有含义的链接关系类型（link relation type）的链接没什么大用处。本节演示如何给链接分配关系类型。

5.5 节，“如何使用链接来管理应用程序的流程”

通过本节了解如何使用链接来管理应用程序的流程。

5.6 节，“如何处理临时 URI”

不是所有链接都是永久性的。本节给出了一些场景，其中的链接可能只存在很短时间，并演示了如何处理这些情况。

5.7 节，“何时以及如何使用 URI 模板”

通过本节了解在服务器不能构建完整 URI 的情况下如何使用 URI 模板。

5.8 节，“如何在客户端使用链接”

通过本节了解如何实现客户端来使用服务器提供的链接。

5.1 如何在 XML 表述中使用链接

HTML，XHTML 和 Atom 制定了在表述中嵌入链接的规则。理解这些格式语义的客户端可以发现表述中的链接。然而，XML 是一种通用的格式，服务器有责任设计一种方法，把链接嵌入到为客户端设计的 XML 格式的表述和文档中。客户端可以参考该设计以了解如何发现并使用包含在表述中的链接。

问题描述

您想知道如何将链接嵌入到 XML 格式的表述中。

解决方案

使用 Atom 中定义的 `link` 元素。此元素被声明在 *http://www.w3.org/2005/Atom* 命名空间中，具有以下属性：

89 `href`

包含链接的 URI。

`rel`

这个属性原本代表“关系”，标明了链接类型。

`title`（可选）

这是一个易于阅读的链接标题。如果希望最终用户使用该链接，客户端可以把它显示给用户。

`type`（可选）

这是服务器针对本链接 URI 可能返回的表述媒体类型的提示。

`hreflang`（可选）

这是对服务器针对本链接 URI 可能返回的表述内容语言的提示。

`length`（可选）

这是服务器针对本链接 URI 可能返回的表述内容长度的提示。

存在于特定元素中的链接本身并不意味着什么，除非客户端知道如何找到并使用它们。因此，要提供文档，告诉客户端如何在 XML 表述中找到链接。

问题讨论

Atom 的 `link` 元素是灵活的、可扩展的，类似于 HTML 和 XHTML 文档中的链接。下面这个例子是一个照片资源表述中的链接：

```
<photo xmlns:atom="http://www.w3.org/2005/Atom">
  ...
  <atom:link link href="http://east-nj1.photos.example.org/987/nj1-1234"
        type="image/jpeg"
        rel="alternate"
        title="Sunset view from our backyard"/>
</photo>
```

在这个表述中，链接的目的是要告诉客户端，位于 `http://east-nj1.photos.example.org/987/nj1-1234` 的资源可以作为候补表述，它提供了一个 JPEG 格式的表述。

href 和 rel 属性是所有链接属性中的必要属性。href 属性的值是客户端定位资源 URI 所必需的，rel 属性传达了链接的语义。它们回答了如下问题：

- URI 所指的资源是什么？
- 链接的意义何在？
- 客户端能在该 URI 的资源上执行什么操作？

90

- 针对那个资源的请求和响应都支持哪些表述格式？

其他属性都是可选的。在适当的时候使用它们来提供一些提示信息。

href 的值是一个绝对的 URI。也可以使用相对 URI，只要在 link 元素或者它的父元素上引入 xml:base 属性，如下所示：

```
<addresses xmlns:atom="http://www.w3.org/2005/Atom" xml:base=
  http://www.example.org">
  <atom:link rel="http://www.example.org/rels/address"
       href="/address/1">
  <atom:link rel="http://www.example.org/rels/address"
       href="/address/2">
  <atom:link rel="http://www.example.org/rels/address"
       href="/address/3">
</addresses>
```

xml:base 属性的值是一个 URI，客户端可以用它来解析链接中的相对 URI。

因为 XML 解析库不会自动根据 xml:base 属性来解析相对 URI，所以最好选择绝对 URI。

有些应用程序使用纯 URI 把资源链接在一起。下面是一些例子：

```
<!-- 避免这种风格 -->
<user>
  <uri>http://www.example.org/user/001</uri>
  <address>http://www.example.org/user/001/address/001</address>
</user>
```

要避免这种传递 URI 的方法，因为这些 URI 缺乏 Atom link 元素的灵活性和可扩展性。正如 5.4 节所讨论的，没有链接关系类型的纯 URI 不能向客户端传达 URI 的语义。

PDG

5.2 如何在 JSON 表述中使用链接

在写这本书时，JSON 表述中的链接还没有标准的使用方法。本节提出了一个从 Atom 的 link 元素定义到 JSON 的映射，保留了相同的灵活性和可扩展性。

问题描述

您想知道如何将链接纳入 JSON 格式的表述中。

91

解决方案

每个链接都使用一个 link 属性（或 links 属性，以数组形式引入多个链接），属性值是一个链接对象或链接对象数组。每个链接对象都包含 href 和 rel 属性。详见 5.1 节了解这些属性的含义。

此外，还可以使用链接关系类型作为属性名，链接的 URI 作为属性值。

问题讨论

以下是使用两种形式的链接范例:

```
{
  "link" :{
    "rel" : "alternate",
    "href" : "http://east-nj1.photos.example.org/987/nj1-1234",
  }
}

{
  "links" :[
    {
     "rel" : "alternate",
     "href" : "http://east-nj1.photos.example.org/987/nj1-1234"
    ,

    {
     "rel" : "http://www.example.org/rels/owner",
     "href" : "http://east-nj1.photos.example.org/987",
    }
  ]
}
```

上面这种形式遵循了 Atom 的 link 元素定义。下面则使用更紧凑的形式定义了相同的链接:

```
{
  "alternate" : "http://east-nj1.photos.example.org/987/nj1-1234"
  "owner" : "http://east-nj1.photos.example.org/987/nj1-1234"
}
```

不管采用哪种形式，重要的是要抓住 Atom 中 link 元素的本质。确保最起码为每个 URI 带上链接关系类型。

详见 5.4 节了解为何链接关系类型如此重要。

5.3 何时以及如何使用链接标头

链接标头提供了一种格式独立的方式来传递链接。可以通过链接标头来传递链接，无须把它嵌入到表述正文中。

问题描述

92

您想知道如何通过 HTTP 标头来传递链接。

解决方案

链接标头提供了一种传递链接的方式,即将链接作为 HTTP 标头来传递。下面是 Link 标头的格式和范例:

```
# 链接标头格式
Link: <{URI}>;rel="{relation}";type="{media type"};title="{title}"...

# 范例
Link: <http://east-nj1.photos.example.org/987/nj1-1234>;rel="alternate"
```

当想要用格式独立的方式来传递链接，或者表述格式不支持链接时，可以使用链接标头。

问题讨论

Link 标头适用于以下情况:

- 使用二进制格式的表述，如图像、富文本文档，电子表格等。
- 表述的格式无法轻易发现链接（例如，纯文本文档）。
- 您的客户端/服务器软件需要在不解析表述正文的情况下添加链接或读取链接。

下面是一个照片图像资源的例子，其中带有两个链接。前者指向同一照片资源的替

代表述，后者则提供了照片所有者的链接：

```
# 响应
HTTP/1.1 200 OK
Content-Type: image/jpeg
Link: <http://east-nj1.photos.example.org/987/nj1-1234.xml>;
      rel="alternate;type="application/xml"
Link: <http://east-nj1.photos.example.org/987>;
      rel="http://www.example.org/rels/owner"

... 字节 ..
```

链接标头的主要优点在于它独立于格式，并且在协议层是可见的。另一方面，表述内的链接是格式相关的。特别是像 XML 和 JSON 这样的通用格式，没有定义用于发现链接的处理模型。换言之，客户端需要读取服务器提供的文档，这样才能了解如何发现 XML 或 JSON 中的链接。链接标头就没有这样的限制。

93

5.4 如何分配链接关系类型

如果没有为链接中的 URI 分配有意义的语义，链接本身的作用并不是很大。链接关系类型传达了链接的角色或作用。一旦客户端和服务器对这些类型的含义达成一致，客户端就可以从链接中找到并使用 URI。因此，为链接分配具体且有意义的关系类型是非常必要的。

问题描述

您想知道针对一个链接要使用什么关系类型。

解决方案

链接关系类型的主要目的是作为与链接关联的语义的标识符。有两种为链接关系类型赋值的方法。当链接的作用与表 5-1 中所述的标准类型相匹配时，使用表中的值。附录 E 中有已注册的关系类型的完整列表。如果没有相匹配的类型，按照如下惯例定义一个扩展的链接关系类型：

- 以 URI 的形式来表示链接关系类型，例如 http://www.example.org/rels/create-po。
- 在该 URI 中提供一个 HTML 文档形式的信息资源，该 HTML 文档描述了链接关系类型的语义，其中包含如下一些细节：支持的 HTTP 方法、请求和响应所支持的表述格式，以及与使用链接时的相关业务规则等。

- 如果链接关系类型要面向大众，按照 Web Linking Internet-Draft 的 6.2 节中所列出的过程登记链接关系。

表 5-1：一些常用的已注册链接关系类型

名称	用途
self	使用这个类型来链接到资源的首选 URI
alternate	为相同资源提供另一个版本的链接时，使用这个类型
edit	使用这个类型来链接到一个客户端可以用来编辑资源的 URI 上
related	使用这个类型来链接到相关的资源上
previous 和 next	使用这些类型来链接到一系列有序资源中的上一个或下一个资源
first 和 last	使用这些类型来链接到一系列有序资源的第一个和最后一个资源，例如，到集合里的第一个和最后一个资源

问题讨论

94

为了保持高效的链接，必须为链接关系选择明确的值。如果表 5-1 中所有的类型都匹配不了您的用例，可以利用链接关系值的可扩展性，以 URI 的形式来定义特定于应用程序的链接关系值。

链接关系最先出现在 HTML 里。HTML 4.01 的 6.12 节[注1]里定义了如下链接关系：alternate、stylesheet、start、next、prev、contents、index、glossary、copyright、chapter、section、subsection、appendix、help 和 bookmark。所有这些值都不是大小写敏感的。针对每个关系，还可以使用多个值，例如 rel="alternate help"。HTML 5 规范定义了附加链接关系，例如 archives、feed 和 pingback 等。

Atom Syndication Format 也定义了链接，提供了可扩展的机制来定义扩展链接关系类型。下面这个表述的例子中，同时使用了注册的和扩展的关系类型：

```
<review xmlns="org:example:books" xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="self" href="http://example.org/book/978-0374292881/review/ 189742"/>
  <atom:link rel="first" href="http://example.org/book/978-0374292881/review/9863"/>
  <atom:link rel="last" href="http://example.org/book/978-0374292881/review/49732"/>
  <id>urn:org:example:books:189742</id>
  <author>...</author>
  <content type="text/html">&lt![CDATA[
    ...
  ]]&gt;</content>
  <atom:link rel="http://example.org/rels/book"
             href="http://example.org/book/978-0452286757"/>
  <atom:link rel="http://example.org/rels/author"
             href="http://example.org/authors/ayn_rand"/>
```

注 1： HTML 4.0 详见 http://www.w3.ong/TR/html401/。

```
  <atom:link rel="http://example.org/rels/add-review"
             href="http://example.org/book/978-0452286757/reviews"/>
</review>
```

这个表述属于 URI 为 http://www.example.org/book/978-0374292881/review/189742 的书评资源，使用了下列链接关系类型：

self
: 该链接关系类型的作用是为资源提供一个链接。客户端可以使用此链接的 URI 来获取资源。

first 和 last
: 带有此类关系类型的链接指向第一篇和最后一篇书评。

http://www.example.org/rels/book
: 这个扩展的关系类型标识了与此书评相关的书籍。

http://www.example.org/rels/author
: 这个扩展的关系类型标识了此书的作者。

http://www.example.org/rels/add-review
: 带有该关系类型的链接的作用是为该书创建新的书评。

95

始终将 URI 作为扩展的链接关系类型的值。此外，考虑在扩展链接关系的 URI 上提供 HTML 文档，用下面的信息来描述链接关系：

- 链接关系的目的。
- 使用该链接关系的资源类型，以及链接目标的资源类型。
- 针对目标 URI 的有效 HTTP 方法。
- 所期望的针对目标 URI 的请求和响应的媒体类型。

下面是一个例子：

```
<html>
  <head>
    <title>Link relation - http://www.example.org/rels/add-review</title>
  </head>
  <body>
    <h1>Link Relation: <code>http://www.example.org/rels/add-review</code></h1>

    <p>Use this link relation to add new book reviews. You may find links with this
     relation type in representations of book resources and review resources.</p>
```

```
    <p>You can use the link's URI to submit new reviews.</p>

    <p>Use a representation of media type <code>application/xml</code> and HTTP
      method

    <code>POST</code> to submit new reviews.</p>
  </body>
</html>
```

此类在线文档可以在开发时帮助发现链接关系类型。

对于所有的链接关系类型都要使用小写字符。

请注意，注册过的链接关系类型（例如 self 和 alternate）必须以非大小写敏感的方式进行比较，而扩展的关系类型在以 URI 形式做比较时必须是大小写敏感的。对于这两种方式都使用小写值可以简化提取链接的代码。

5.5 如何使用链接来管理应用程序的流程

超媒体和链接的主要应用之一就是将客户端从学习服务器用来管理应用程序流程的规则中解耦出来。服务器可以提供包含应用程序状态的链接，从而利用超媒体作为应用程序状态的引擎。

问题描述

96

您想知道如何保持客户端与实现应用程序流程的业务逻辑的解耦。

解决方案

设计每个表述时，其中包含的链接可以帮助客户端转移到各种可能的后续状态。如果服务器在从一个步骤进行到下一个步骤时需要携带状态，将状态编码到链接中，就像 1.3 节描述的那样。

问题讨论

想象一个管理员工招聘流程的 Web 服务。这一流程中有多个步骤，例如（a）输入候选人的详细信息，（b）查看介绍人意见，（c）进行背景审查，（d）录用。每一个步骤都必须按顺序进行。可以使用表 5-2 里的资源和 URI 实现此序列。这些 URI 中的标记{id}的是候选人的 ID。

表 5-2：员工招聘流程中的 URI

方法	URI	目的
POST	http://www.example.org/hires	创建一个候选人资源
POST	http://www.example.org/hires/{id}/refs	提交介绍人意见
POST	http://www.example.org/hires/{id}/bgchecks	提交背景审查结果
POST	http://www.example.org/hires/{id}/hire	发录用通知
POST	http://www.example.org/hires/{id}/no-hire	不录用

假设这个 Web 服务受以下业务规则的约束：

- 在输入候选人信息之后，开始介绍人审查。
- 在收到至少两个肯定的介绍人意见之后，开始背景审查。
- 背景没问题之后，发录用通知。

有了这些规则和 URI，就可以为员工招聘流程实现客户端了。每个步骤之后，客户端可以检查规则，看是否可以进行到下一个步骤。不过，这种做法造成了客户端和服务器业务规则的耦合，因为服务器应该管理那些规则，而非客户端。

当您的 Web 服务需要客户端了解并实现应用程序的流程规则时，就是在引入另一种客户端和服务器之间的耦合。就像构造 URI 的细节，这种流程规则也成为了 Web 服务公共接口的一部分，这样一来不破坏客户端就无法更改规则了。

97 一个更好的选择是让服务器为客户端提供“上下文“的链接，其中包含可能的下一步的 URI。当客户端发现链接时，它可以尝试转移到流程中的下一步。当表述中的链接缺失时，客户端可以认为无法进行状态转移。这避免了客户端了解和硬编码应用程序流程。

依此类推，用户与一个基于浏览器的 Web 应用程序进行交互。用户按照链接和表单操作浏览器，不用预先知道应用程序流程。链接的目的是要把同样的好处扩展到应用程序客户端。以下这个表述，是服务器创建的一个新候选人资源：

```
# 录入候选人信息的请求
POST /hires HTTP/1.1
Host: www.example.org
Content-Type: application/json

{
  "name": "Joe Prospect",
  ...
```

```
}

# 包含提交介绍人审查结果链接的响应
HTTP/1.1 201 Created
Location: http://www.example.org/hires/099
Content-Location: http://www.example.org/hires/099
Content-Type: application/json

{
  "name": "Joe Prospect",
  "id": "urn:example:hr:hiring:099",
  ...
  "link" : { ❶
    "rel" : "http://www.example.org/rels/hiring/post-ref-result",
    "href" : "http://www.example.org/hires/099/refs"
  }
}
```

❶ 提交介绍人审查结果的链接

表述中有一个用来提交介绍人审查结果的链接。由于没有其他链接，客户端还不能初始化背景审查流程或发录用通知。

输入两个肯定的介绍人审查结果之后，服务器可以返回一个表述，告诉客户端可以开始背景审查了：

```
# 输入第一个介绍人意见的请求
POST /hires/099/refs HTTP/1.1
Host: www.example.org
Content-Type: application/json

{
  "text" : "Joe is a ...",
  "by" : "...",
  "on" : "2009:10:12T16:05:00Z"
}

# 响应
HTTP/1.1 200 OK
Content-Location: http://www.example.org/hires/099
Content-Type: application/json

{
  "name": "Joe Prospect",
  "id": "urn:example:hr:hiring:099",
  ...
  "link" :{
    "rel" : "http://www.example.org/rels/hiring/post-ref-result",
```

98

```
    "href" : "http://www.example.org/hires/099/refs"
  }
}

# 输入第二个介绍人意见的请求
POST /hires/099/refs HTTP/1.1
Host: www.example.org
Content-Type: application/json

{
  "text" : "Worked with Joe, ...",
  "by" : "...",
  "on" : "2009:10:12T17:00:00Z"
}

# 响应
HTTP/1.1 200 OK
Content-Location: http://www.example.org/hires/099
Content-Type: application/json

{
  "name": "Joe Prospect",
  "id": "urn:example:hr:hiring:099",
  "refs": ...,
  ...
  "links" : [{❶
      "rel" : "http://www.example.org/rels/hiring/add-ref-result",
      "href" : "http://www.example.org/hires/099/refs"
    },
    { ❷
      "rel" : "http://www.example.org/rels/hiring/add-background-check",
      "href" : "http://www.example.org/hires/099/bgchecks"
    }]
}
```

❶ 用来提交介绍人审查结果的链接

❷ 用来提交背景审查结果的链接

99 这时，客户端可以输入更多的介绍人审查结果，或者提交背景审查结果。根据服务器的业务策略，如果结果是可以接受的，服务器会包含一个发放录用通知的链接。

```
# 提交背景审查结果的请求
POST /hires/099/bgchecks HTTP/1.1
Host: www.example.org
Content-Type: application/json
{
  "text" : "...",
```

```
    "by" : "...",
    "on" : "..."
  }

# 背景审查通过
HTTP/1.1 200 OK
Content-Location: http://www.example.org/hires/099
Content-Type: application/json;charset=UTF-8

{
  "prospect" :
  {
    "name": "Joe Prospect",
    "id": "urn:example:hr:hiring:099",
    "refs": ...,
    "link" : { ❶
      "rel" : "http://www.example.org/rels/hiring/make-offer",
      "href" : "http://www.example.org/hires/099/hire"
    }
  }
}
```

❶ 用来发放录用通知的链接

在这种方式下，服务器引导客户端走完了员工招聘流程，无须强迫客户端实现额外的逻辑。

链接的存在本身并不能将客户端从知道如何准备数据以及如何发起状态转移请求中解耦出来。正如 5.4 节讨论的那样，服务器必须建立链接关系类型，并撰写文档，说明如何找到链接以及所有扩展链接关系类型的语义。

5.6 如何处理临时 URI

正如 4.4 节所讨论的，Web 的完整性是建立在 URI 持久性的基础之上的。但是，有些情况下，URI 是临时的。例如，一个 URI 可能只能使用一次，或者可能在一段固定的时间后过期。以下是一些依赖于临时 URI 的情况：

- Web 服务向客户端提供安全令牌，就像您在访问银行收银台时可能会收到的令牌。客户端可以用此令牌在短时间内访问资源。
- 保险报价 Web 服务生成报价。每个报价都特定于某个指定客户端，72 小时内有效，之后报价就作废了，客户端必须获取一个新的令牌。 100
- 在网站上注册一个用户后，服务器通过电子邮件向用户发送一串密文，期望用户

在服务器的 HTML 表单中输入这串密文以便验证用户的电子邮件地址。

问题描述

您想知道如何支持临时 URI。

解决方案

通过链接来传递临时 URI。为那些链接分配扩展关系类型，并撰写文档说明 URI 在多长时间内有效，URI 到期以后客户端应该做什么。当客户端向一个过期的 URI 提交请求之后，返回一个适当的 4xx 错误代码，并在正文中说明客户端可以采取的各种措施。

问题讨论

当服务器提供了某个资源的 URI，默认情况下，客户端可以认为 URI 是永久性的。由于服务器不会事先生成临时 URI，它们可以在运行时使用链接来传递那些 URI。但那只是问题的一部分。另一部分是如何告知客户端，这些 URI 是临时的。用清晰的文档来说明链接关系可以解决这个问题。

举例来说，如果是让客户在报价后的两分钟内购买，服务器可以提供一个用于购买的链接。该链接包含一个加密状态，两分钟后便会失效，以此作为额外的安全措施：

```
# 请求
POST /bid/ASBV_04_10_2009_1 HTTP/1.1
Host: www.example.org
Content-Type: application/xml;charset=UTF-8

...

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

<purchase-req xmlns:atom="http://www.w3.org/2005/Atom">
  <amount currency="USD">...</amount>
  ...
  <atom:link rel="http://www.example.org/purchase-req/auth"
        href="http://www.example.org/auth/ASBV_04_10_2009_1/09_31?_k=
              a1191fd35d23"/>
</purchase-req>
```

101 在这种情况下，服务器需要撰写文档说明带有 http://www.example.org/funds-req/auth 链接关系类型的 URI 的有效期只有两分钟。如果客户端在过期后

访问该 URI，服务器会返回一个错误：

```
# 请求
POST /bid/ASBV_04_10_2009_1/09_31?_k=a1191fd35d23 HTTP/1.1
Host: www.example.org
Content-Type: application/xml;charset=UTF-8

...

HTTP/1.1 403 Forbidden
Content-Type: application/xml;charset=UTF-8
Date: Sat, 17 Oct 2009 20:16:18 GMT

<error xmlns:atom="http://www.w3.org/2005/Atom">
  <message xml:lang="en">Authorization expired. Resubmit the bid.</message>
  <atom:link rel="http://www.example.org/purchase-req/bid"
        href="http://www.example.org/auth/ASBV_04_10_2009_1?retry"/>
</error>
```

详见 5.8 节了解如何实现客户端来处理临时 URI。

5.7 何时以及如何使用 URI 模板

到目前为止，本章的内容假定服务器拥有为每个链接生成有效且完整的 URI 所需的所有信息。当情况并非如此时，服务器可以向客户端提供 URI 模板。

问题描述

您想知道如何让客户端在向服务器提交请求之前，在 URI 中包含额外的信息。

解决方案

URI 模板是一个字符串，它由互相匹配的括号（{ }）隔开的标记组成的。客户用 URI 安全的字符串代替这些标记（包括括号），把模板转换成有效的 URI。

为了使标记替换及匹配保持简单，把标记限制在 URI 的以下部分：

- 路径段，如 `http://www.example.org/segment1/{token1}/segment2`
- 查询参数值，如 `http://www.example.org/path?param1={p1}¶m2={p2}`
- 矩阵参数值，如 `http://www.example.org/path;param1={p1}; param2={p2}`

要在表述中包含一个 URI 模板，可使用下列方法：

102

- 对于 XML 系的表述，使用 `link-template` 元素，它被定义在您自己的应用程序

的 XML 命名空间中。为了一致性，在该元素上定义与 `link` 元素上相同的属性。

请注意，Atom Syndication Format 中定义的 `link` 元素的 `href` 属性不允许使用 URI 模板。

- 对于 JSON 表述，使用 link-template 或 link-templates 属性来传递 URI 模板。

问题讨论

URI 模板为服务器提供了一种方法，向客户端返回半透明的 URI，并允许客户端填充缺失的部分以生成有效的 URI。URI 模板的理念并不新鲜。WSDL 2.0 的第 6.8.1.1 节使用括号来指定替换标记。WADL 使用相同的标记在 URI 路径段中指定资源 ID。还有人尝试将 URI 模板使用的语法作为 Internet Draft[注2] 形式化下来。

下面是一个 URI 模板的例子，用于搜索人员：

```
http://www.example.org/people?k={keyword}&
    p={page-number}&r={results-per-page}
```

该模板有三个标记：一个用于搜索关键词，一个用于起始页码，一个用于每页的结果数。客户端可以用实际值代替这些标记，生成有效的 URI。

```
http://www.example.org/people/k=sports&p=1&r=10
```

由于括号的存在，在使用 URI 安全的字符替换所有的标记（包括括号）前，URI 模板不是有效的 URI。

下面的表述片段中包含 URI 模板：

```
<!-- 表述 -->
<link-template href="http://www.example.org/customers/{customer-id}"
               title="View customer detail"
               rel="http://www.example.org/rels/detail"/>
<link-template href="http://www.example.org/search/k={keyword}&p={page-number}&
               r={results-per-page}"
               title="Search results"
               rel="http://www.example.org/rels/search"/>

// JSON 表述
```

注 2：在本书编写时，该 Internet-Draft 已经过期，请访问 http://tools.ietf.org/html/draft-gregorio-uritemplate 获取更新内容。

```
"link-templates" : [{
   "rel" : "http://www.example.org/rels/detail",
   "href" : "http://www.example.org/customers/{customer-id}",
   "title" : "View customer detail"
 },
 {
   "rel" : "http://www.example.org/rels/search",
   "href" : "http://www.example.org/search/k={keyword}&p={page-number}&r=
       {results-per-page}",
   "title" : "Search results"
 }]
```


这些 link-template 元素的 href 值是 URI 模板。第一个模板要求客户端用有效值替换{customer-id}。第二个模板要求客户端替换{keyword}，{page-number}和{results-per-page}。

由于 URI 模板是半透明的，包含客户端需要替换的标记，因此您需要告诉客户端对于每个标记，哪些值是有效的。最简单的方法是为 URI 模板中使用的每个标记添加文档，如表 5-3 所示。

表 5-3：URI 模板：/search/k={keyword}&p={page-number}&r={result-per-page}

标记	作用
{keyword}	用逗号或空格分隔的搜索词汇的列表
{page-number}	搜索结果的页码，从 0 开始
{results-per-page}	每页的结果数

5.8 如何在客户端使用链接

Web 浏览器是使用链接进行浏览的客户端的最好范例。服务器在带有链接的 HTML 表单中呈现应用程序的当前状态。用户既可以立即调用链接，也可以将它们作为书签以备后用。页面中显示的 HTML 帮助用户确定特定的链接是否可以作为书签（也就是可以稍后使用），或者是否需要立即操作某些链接。您需要让客户端实现同样的运作方式。

问题描述

您想知道如何实现客户端来使用由服务器提供的链接。

解决方案

为了支持服务器提供的 URI 和 URI 模板，应该基于已知的关系类型从链接中提取 URI 和 URI 模板。这些链接与其他资源数据构成了应用程序的当前状态。

如果应用程序长时间运行，可以把 URI 和关系类型与其他表述数据一起存储起来。

104 根据链接的存在与否来决定流程，存储判断表述中是否包含给定链接方面的知识。

通过链接关系文档了解各种相关的业务规则，例如身份验证、URI 的持久性、方法和所支持的媒体类型等。

问题讨论

当服务器在表述中使用链接传递 URI 时，实现客户端来利用那些链接。这需要从链接中提取应用程序的状态（以特定的 URI 和链接关系类型进行表示），储存它们，并且根据已知关系类型的链接是否存在来制定决策。

从客户端的角度再来考虑一下 5.5 节讨论的员工雇用流程。客户端应用程序很有可能是长时间运行的。它可能是一个桌面应用程序，或一个 Web 应用程序，带有基于 Web 的用户界面。有人使用客户端做一些重要的工作，例如发送背景审查请求。每当有新的信息，用户调用客户端，驱动它流转到下一个步骤。在两个步骤之间，客户端需要将资源的状态和所有的有效转移（即链接）保存到数据库里。

举例来说，再看一下输入准员工信息的请求和服务器的响应。

```
# 输入候选人信息的请求
POST /hires HTTP/1.1
Host: www.example.org
Content-Type: application/json

{
  "name": "Joe Prospect",
  ...
}

# 包含提交介绍人审查结果链接的响应
HTTP/1.1 201 Created
Location: http://www.example.org/hires/099
Content-Location: http://www.example.org/hires/099
Content-Type: application/json

{
  "name": "Joe Prospect",
  "id": "urn:example:hr:hiring:099",
  ...
  "link" :{
    "rel" : "http://www.example.org/rels/hiring/post-ref-result",
    "href" : "http://www.example.org/hires/099/refs"
  }
}
```

客户端收到该表述时，需要提取并存储以下内容：

- 员工的详细信息，例如姓名、ID 和其他信息。
- 资源的当前状态，雇用流程已经准备进入介绍人审查步骤了。

用户进行介绍人审查并准备输入详细内容时，客户端需要识别资源的当前状态，并为用户提供用户界面。在本例中，这涉及到呈现一个输入介绍人审查结果的用户界面。

在输入两个肯定的介绍人审查结果后，资源的当前状态会变成下面这样：

```
# 响应
Content-Location: http://www.example.org/hires/099
Content-Type: application/json

{
  "name": "Joe Prospect",
  "id": "urn:example:hr:hiring:099",
  "refs": ...,
  ...
  "links" : [{
      "rel" : "http://www.example.org/rels/hiring/add-ref-result",
      "href" : "http://www.example.org/hires/099/refs"
    },
    {
      "rel" : "http://www.example.org/rels/hiring/add-background-check",
      "href" : "http://www.example.org/hires/099/bgchecks"
    }]
}
```

资源的当前状态现在包括两个链接：一个是继续补充介绍人审查结果，另一个是添加背景审查结果。客户端的用户界面中需要向用户提供一个选择，添加一个新的介绍人审查或输入背景审查结果。此时，用户可能会根据业务策略开始背景审查，并在得到结果后重新打开客户端。然后，客户端重新创建用户界面，提供同样的选择，输入介绍人审查或背景审查的结果。

从这个例子中可以看到，当服务器使用链接时，客户端需要像浏览器一样使用这些链接。虽然这看起来可能很麻烦，但是这种方法把客户端从服务器那里解耦出来，并帮助客户端和服务器各自独立进行。

第 6 章

Atom 和 AtomPub

107

Atom Syndication Format（RFC4287）和 Atom Publishing Protocol（也称为 AtomPub，RFC5023）中定义了一些资源（比如 Entry 与 Feed，还有它们的表述）以及操作这些资源的协议。Atom 的设计中使用了易于阅读的内容，比如 HTML 和纯文本。它最适合用于那些主要基于文本，供人阅读的资源，例如博客、论坛、评论系统等。AtomPub 用于描述语义，让客户端可以创建和修改 Atom 格式的资源。AtomPub 还引入了服务和分类资源来帮助应用程序进行探测。

Atom 与 AtomPub 已被用于多种应用场景。虽然 Atom 常用于输出博客 Feed，但是也可以把它扩展到用户配置、搜索结果、相册等应用数据。举例来说，Google Data Protocol API 针对一些 Google 产品扩展了 Atom。这样的用法引发了一个问题，即何时适合使用 Atom 与 AtomPub。就算您发现 Atom 与 AtomPub 不适合自己的 Web 服务，也会觉得使用链接、服务文档，以及支持媒体资源和分类是非常有用的。本章包含的内容回答了这些问题:

6.1 节，“如何利用 *Atom* 建模资源”
　　通过本节了解如何使用 Atom 来对资源和集合进行建模。

6.2 节，“何时使用 *Atom*”
　　通过本节确定 Atom 对您的资源表述来说是否是适当的格式。

6.3 节，“如何使用 *AtomPub* 服务和分类文档”
　　通过本节了解如何使用 AtomPub 服务和分类文档。

6.4 节，“如何针对 *Feed* 和 *Entry* 资源使用 *AtomPub*”
　　通过本节了解如何使用 AtomPub 管理 Atom 格式的资源。

6.5 节，“如何使用媒体资源”
　　通过本节了解如何使用 AtomPub 管理媒体资源。

6.1 如何利用 Atom 建模资源

108

使用 Atom 来表示资源的关键之一优势是互操作性。可以找到数量众多的工具（比如 Google Reader、Bloglines 和 NewsGator 这样的 Feed 阅读器，以及大多数的浏览器和电子邮件客户端）和编程类库（Apache Abdera、ROME Project, Windows Communication Foundation 等）。使用 Atom 作为资源的表述格式涉及以 Atom Entry 和 Feed 的形式对资源进行建模，以及将应用程序特定的数据字段映射到 Atom 规定的元素和属性上。

问题描述

您想知道如何让您的资源表述支持 Atom 格式。

解决方案

要使用 Atom，把资源建模为 Entry，把集合建模为 Feed。Entry 的表述是一个 XML 文档，在 Atom 中定义的 `entry` 元素是其根元素。Feed 的表述也是一个 XML 文档，`feed` 元素为根元素。一个 `feed` 文档中包含几个 `entry` 元素。这些元素都定义在 *http://www.w3.org/2005/Atom* 命名空间中。这个命名空间的常用前缀是 `atom`。附录 D 里有 Atom 格式中定义的元素的概述。

问题讨论

Atom 是一个 XML 格式，它基于两种类型的资源：Entry 文档及 Feed 文档。为了更好地说明如何使用 Atom 来设计表述，考虑一本书的 XML 格式表述：

```
# 请求
GET /books/1 HTTP/1.1
Host: www.example.org

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

<book xmlns:atom="http://www.w3.org/2005/Atom">
  <isbn>0-9767736-6-X</isbn>
  <title>Johnny Web and the Atomic Circle</title>
  <atom:link href="http://www.example.org/books/1"/>
  <date-published>2010-01-01</date-published>
  <cover-art href="http://www.example.org/books/1/cover" type="image/jpeg"/>
  <author href="http://www.example.org/books/1/authors/1>R. W. Smith</author>
  <description>
    A lively tale of a young boy who discovers a secret
    scientific fraternity with a dark past and hidden purpose.
```

```
  <description>
</book>
```

109 下面是同样的资源，表示为 Atom Entry 文档：

```
# 请求
GET/books/1 HTTP/1.1
Host: www.example.org

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

<atom:entry xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:id>urn:isbn:0-9767736-6-X</atom:id> ❶
  <atom:title>Johnny Web and the Atomic Circle</atom:title> ❷
  <atom:link href="http://www.example.org/books/1" rel="edit"/>
  <atom:link href="http://www.example.org/books/1/cover.png"
           rel="enclosure" type="image/png"/> ❸
  <atom:published>2010-01-01T00:00:00Z</atom:published> ❹
  <atom:author> ❺
    <atom:name>R. W. Smith</atom:name>
    <atom:uri>http://www.example.org/books/1/authors/1</atom:uri>
  </atom:author>
  <atom:updated>2010-12-13T18:30:02Z</atom:updated>
  <atom:content type="text"> ❻
    A lively tale of a young boy who discovers a secret scientific fraternity
    with a dark past and hidden purpose.
  </atom:content>
</atom:entry>
```

❶ 这本书的 ISBN 标识符

❷ 书名

❸ 封面的链接

❹ 出版日期

❺ 该书的作者

❻ 对该书的描述

这个表述的媒体类型是 application/atom+xml，在 IANA 注册为 Atom。

Atom 为 Feed 文档注册了媒体类型 application/atom+xml。6.4 节中讨论的 AtomPub 把 Entry 文档定义为带有 application/atom+xml;type=entry 媒体类型的表述。在 AtomPub 中，当表述是一个 Atom Entry 文档时，您可以将 application/atom+xml 或 application/atom+xml;type=entry 作为媒体类型。

这个表述中包含了几个带有应用程序数据（如作者、标题、描述、封面链接等）的托管元素。之前的两个表述都含有相同的信息，但服务器对它们的编码方式不同。一些显著的区别如下：

110

- ISBN 标识符被映射到了 atom:id 元素上。
- 封面的链接被映射到了 atom:link 元素上。
- 诸如书名、发行日期之类的元数据被映射到了相应的 Atom 元素上。
- 作者数据被映射到了 atom:author 元素上。

只要资源的应用程序语义与那些 Atom 规定的元素是匹配的，这种映射就是有意义的。

可以把这个例子延伸出去，使用 Atom Feed 文档来表示一组图书：

```
GET /books HTTP/1.1
Host: www.example.org

HTTP/1.1 200 OK
Content-Type: application/atom+xml;charset=UTF-8
<atom:feed xmlns:atom="http://www.w3.org/2005/Atom"> ❶
  <atom:title>Sci-Fi Books</title>
  <atom:link href="http://www.example.org/books" rel="self"
    hreflang="en" type="application/atom+xml"/>
  <atom:updated>2013-12-13T18:30:02Z</atom:updated>
  <atom:author>
    <atom:name>Example Inc.</atom:name>
  </atom:author>
  <atom:id>urn:uuid:5f49aa74-e920-425d-a150-8907494905e7</atom:id>
  <atom:entry> ❷
    <atom:id>urn:isbn:0-9767736-6-X</atom:id>
    <atom:title>Johnny Web and the Atomic Circle</atom:title>
    <atom:link href="http://www.example.org/books/1" rel="alternate"/>
    <atom:link href="http://www.example.org/books/1/cover.png"
               rel="enclosure" type="image/png"/>
    <atom:published>2010-01-01T00:00:00Z</atom:published>
    <atom:author>
```

```
      <atom:name>R. W. Smith</atom:name>
      <atom:uri>http://www.example.org/books/1/authors/1</atom:uri>
    </atom:author>

    <atom:updated>2010-12-13T18:30:02Z</atom:updated>
    <atom:content type="text">
      A lively tale of a young boy who discovers a secret scientific fraternity
      with a dark past and hidden purpose.
    </atom:content>
  </atom:entry>
  <atom:entry> ❸
    <atom:id>urn:isbn:0-9767736-9-X</atom:id>
    <atom:title>Johnny Web Meets the Wolfman</atom:title>
    <atom:link href="http://www.example.org/books/2" rel="alternate"/>
    <atom:link href="http://www.example.org/books/2/cover.png"
               rel="enclosure" type="image/png"/>
    <atom:published>2011-02-21T00:00:00Z</atom:published>
    <atom:author>
      <atom:name>R. W. Smith</atom:name>
      <atom:uri>http://www.example.org/books/1/authors/1</atom:uri>
    </atom:author>
    <atom:updated>2010-12-13T18:30:02Z</atom:updated>
    <atom:content type="text">
      Young Johnny goes to college and sets out to solve the mystery behind the
      strange noises coming from Professor Sirius' basement lab.
    </atom:content>
  </atom:entry>
</atom:feed>
```

111

❶ 将图书集合表示为一个 Feed

❷ 表示为 Feed 的成员 Entry 的图书

❸ 另一本书，表示为相同 Feed 中的另一个成员 Entry

在这个表述中，Feed 文档有两个 Entry，分别对应了一本书。如果有更多的 Entry，或者服务器需要对 Feed 做分页，可以使用带有 `previous`、`next`、`first` 和 `last` 关系类型的链接。这些关系是由 RFC 5005 所规定的。附录 E 中有已注册的链接关系类型的列表，3.7 节中有使用链接进行分页的例子。

任何能识别 Atom 格式的客户端软件都可以在无须自定义编码的情况下处理或显示图书的 Feed。例如，当出版商以 Atom Feed 文档的形式提供了最新图书列表时，可以使用 Feed 阅读器来订阅所有的新书更新。6.2 节中详细讨论了 Atom 的适用性。

6.2 何时使用 Atom

Atom Feed 和 Entry 的默认内容模型中包括文本、HTML 或 XHTML 内容及摘要、标识符、链接、作者、分类等。这个内容模型最适合 Feed 形式的信息片段发布及聚合。然而，由于这种格式掌握了那些有利于大多数应用程序的基本概念，它可以应用到各种场景，而不仅限于内容 Feed。本节可以帮助您确定 Atom 是否适合您的 Web 服务。

问题描述

您想知道 Atom 格式是否适合您的 Web 服务。

解决方案

当资源的信息模型或元数据能自然映射到 Atom Feed 和 Entry 的语法和语义上时，可以考虑使用 Atom。就算资源的信息模型无法映射到 Atom，也可以考虑提供 Atom Feed，其中的 Entry 包含描述资源的简短文字、HTML 或 XHTML 摘要，以及指向资源的链接。用户可以通过 Feed 阅读器这样的支持 Feed 的工具来了解这些资源。

112

问题讨论

Atom 格式针对信息片段列表有很强的语义。Atom 规定了用来标识资源、相关链接和元数据（例如，作者）的元素。然而，在 atom:content 和 atom:summary 元素中传递数据时，它的语义相对较弱。例如，考虑下面的 Atom Entry:

```
<atom:entry xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:title>Johnny Web Series Goes Anime</atom:title>
  <atom:id>urn:blog:1234</atom:id>
  <atom:link rel="self" href="http://www.example.org/blog/2009/11/01"/>
  <atom:updated>2009-11-11T11:11:11Z</atom:updated>
  <atom:author>
    <atom:name>J. W. Smith</atom:name>
  </atom:author>
  <atom:content type="xhtml">
   <div xmlns="http://www.w3.org/1999/xhtml">
     <h1>Johnny Web Series goes Anime</h1>
     <p>After months of negotiation and lots of hush-hush shuttle diplomacy,
     I am pleased to announce we&apos;ve reached a deal to bring the entire
     Johnny Web book series out as an anime television show.</p>
     <p>The first production is scheduled for early next year. &quot;Atomic
     Circle&quot; will be the book used for the first season.
     Others will follow.</p>
     <p>I&apos;ll keep you posted here on the latest developments.
     As Johnny always sez:</p>
     <blockquote>
       <p>Ursus Major!</p>
```

```
        </blockquote>
      </div>
    </atom:content>
  </atom:entry>
```

这是一个博客的例子。大量的信息以 XHTML 标记的形式包含在 atom:content 元素里。能识别 Atom 的客户端可以解释 Atom Entry 文档的内容。

现在考虑下面的表述，这是一个生产计划。这个表述依赖于 atom:entry 元素的可扩展性（详见附录 D 以了解 atom:content 的更多用法）。客户端应用程序应该使用这个生产信息来跟踪和管理电视项目的生产计划:

113

```
<atom:entry xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:title>Johnny Web Sample Production Schedule</atom:title>
  <atom:id>urn:example:sked:1111</atom:id>
  <atom:link rel="self" href="http://www.example.org/ps/1111"/>
  <atom:updated>2011-11-11T11:11:11Z</atom:updated>
  <atom:author><name>J. W. Smith</name></atom:author>
  <atom:content type="application/xml"> ❶
    <production-schedule>
      <story-development>
        <days>5</days>
        <planned-start>2012-01-01</planned-start>
      </story-development>
      <pencil-roughs>
        <days>2</days>
        <planned-start>2012-01-10</planne-start>
      </pencil-roughs>
      <layouts-and-ink>
         <days>3</days>
         <planned-start>2012-01-15</planned-start>
      </layouts-and-ink>
    </production-schedule>
  </content>
</atom:entry>
```

❶ 作为 atom:content 元素的子元素的生产计划

尽管上面的表述是一个有效的 Atom Entry，但是标准的 Atom 客户端不知道如何处理 atom:content 元素中的自定义 XML。定制的应用程序客户端可以理解内嵌的 XML 文档，但是读取这些数据会有额外的开销。对于定制客户端，仅仅为了从 atom:content 中提取额外的 XML 文档，而需要先去解析 Entry 文档，这个附加的工作对改善客户端代码效率毫无帮助。

对于那些目标是机器客户端的非 HTML 数据，Atom 格式就不那么有用了，为资源

设计一个更紧凑的 XML 表述往往更加简单:

```
<production-schedule xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="self" href="http://www.example.org/ps/1111"/>
  <story-development>
    <days>5</days>
    <planned-start>2012-01-01</planned-start>
  </story-development>
  <pencil-roughs>
    <days>2</days>
    <planned-start>2012-01-10</planne-start>
  </pencil-roughs>
  <layouts-and-ink>
    <days>3</days>
    <planned-start>2012-01-15</planned-start>
  </layouts-and-ink>
</production-schedule>
```

在选择 Atom 作为资源表述之前，请考虑以下条件（按照其重要性顺序排列）:

- 资源的数据模型和语义是否能与 Atom Feed 或 Entry 相对应
- 元数据（比如 atom:author、atom:category 和 atom:contributor）对资源是否有意义
- 与 Atom 工具的互操作性

下面是另一个例子。服务器把用户地址簿作为资源提供出来，每个地址由一个邮政地址、一个电子邮件和几个电话号码等内容组成，格式是 XHTML。 114

Atom 非常适合这个例子，因为服务器可以把每个地址的 XHTML 内容连同元数据一起表示为一个资源:

```
<atom:feed xmlns:atom="http://www.w3.org/2005/Atom"> ❶
  <atom:title>John Doe's Address Book</title>
  <atom:link href="http://www.example.org/user/001/ab" rel="self"
        hreflang="en" type="application/atom+xml"/>
  <atom:updated>2013-12-13T18:30:02Z</atom:updated>
  <atom:author>
    <atom:name>John Doe</atom:name>
  </atom:author>
  <atom:id>urn:uuid:94dcfd50-dd4b-11de-8a39-0800200c9a66</atom:id>
  <atom:entry> ❷
    <atom:id>urn:uuid:b550ca30-dd4b-11de-8a39-0800200c9a66</atom:id>
    <atom:title>John's home address</atom:title>
    <atom:link href="http://www.example.org/user/ab/1" rel="alternate"/>
    <atom:published>2009-01-05T10:00:00Z</atom:published>
```

```
    <atom:author>
      <atom:name>John</atom:name>
    </atom:author>
    <atom:updated>2009-05-10T13:30:00Z</atom:updated>
    <atom:content type="xhtml">
      <div> <!-- XHTML of the address -->  </div>
    </atom:content>
  </atom:entry>
  <atom:entry> ❸
    <atom:id>urn:uuid:b550ca30-dd4b-11de-8a39-0800200c9a66</atom:id>
    <atom:title>Jane</atom:title>
    <atom:link href="http://www.example.org/user/ab/2" rel="alternate"/>
    <atom:published>2009-01-05T10:00:00Z</atom:published>
    <atom:author>
      <atom:name>Jane Doe</atom:name>
    </atom:author>
    <atom:updated>2009-01-10T13:30:00Z</atom:updated>
    <atom:content type="xhtml">
      <div> <!-- XHTML of the address -->  </div>
    </atom:content>
  </atom:entry>
```

❶ 表示为 atom:feed 文档的通信录

❷ 表示为包含 XHTML 的 atom:entry 元素的地址

❸ 表示为包含 XHTML 的 atom:entry 元素的另一个地址

同样的地址簿，现在考虑用一个桌面地址簿应用程序来处理它。要实现这样的桌面客户端，服务器需要以适用于数据而非展现的格式来提供地址。下面是一个例子：

115

```
<atom:feed xmlns:atom="http://www.w3.org/2005/Atom"> ❶
  <atom:title>John Doe's Address Book</title>
  <atom:link href="http://www.example.org/user/001/ab" rel="self"
       hreflang="en" type="application/atom+xml"/>
  <atom:updated>2013-12-13T18:30:02Z</atom:updated>
  <atom:author>
    <atom:name>John Doe</atom:name>
  </atom:author>
  <atom:id>urn:uuid:94dcfd50-dd4b-11de-8a39-0800200c9a66</atom:id>
  <atom:entry>
    <atom:id>urn:uuid:b550ca30-dd4b-11de-8a39-0800200c9a66</atom:id>
    <atom:title>John's home address</atom:title>
    <atom:link href="http://www.example.org/user/ab/1" rel="alternate"/>
    <atom:published>2009-01-05T10:00:00Z</atom:published>
    <atom:author>
      <atom:name>John</atom:name>
    </atom:author>
```

```
    <atom:updated>2009-05-10T13:30:00Z</atom:updated>
    <atom:content type="application/xml">
      <address> ❷
        <street>...</street>
        <city>...</city>
        <postal-code>...</postal-code>
        <phone type="home">...</phone>
      </address
    </atom:content>
  </atom:entry>
  <atom:entry>
    <atom:id>urn:uuid:b550ca30-dd4b-11de-8a39-0800200c9a66</atom:id>
    <atom:title>Jane</atom:title>
    <atom:link href="http://www.example.org/user/ab/2" rel="alternate"/>
    <atom:published>2009-01-05T10:00:00Z</atom:published>
    <atom:author>
      <atom:name>Jane Doe</atom:name>
    </atom:author>
    <atom:updated>2009-01-10T13:30:00Z</atom:updated>
    <atom:content type="application/xml">
      <address> ❸
        <street>...</street>
        <city>...</city>
        <postal-code>...</postal-code>
        <phone type="home">...</phone>
      </address
    </atom:content>
  </atom:entry>
```

❶ 针对桌面客户端的地址簿 Feed

❷ 第一个地址，格式化为特定应用程序的 XML 元素

❸ 第二个地址，格式化为特定应用程序的 XML 元素

这种形式削弱了 Atom 格式的功效。

最后，考虑一个股票报价列表，其中包含每个股票的股票名称、份额，以及每小时、每天、每周、每年的最高价和最低价。如果没有用于描述份额、最高价和最低价的扩展，这样的资源数据模型是无法映射到 Atom Entry 上的。Atom 不是此类资源的最佳选择。 116

6.3 如何使用 AtomPub 服务和分类文档

AtomPub 引入了额外的资源，比如服务文档和媒体资源。服务文档帮助客户端发现由 Web 服务提供的集合。服务器可以利用媒体资源将音频和视频文件、图片或任意

文档和 Atom Entry 关联起来。6.5 节讨论了媒体资源。

问题描述

您想知道如何使用 AtomPub 服务和分类文档。

解决方案

使用一个服务文档资源将集合归集为工作区。这种资源的表述是一个以 `service` 为根元素（定义在 *http://www.w3.org/2007/app* 命名空间中）的 XML 文件。这个命名空间最常用的命名空间前缀是 `app`。该表述的媒体类型是 `application/atomsvc+xml`。

一个服务由一个或多个工作区（`app:workspace`）组成。每个工作区包含多个集合（`app:collection`），列出了所有 Feed 的 URI、接受的媒体类型（`app:accept`）和分类（`app:category`）。

分类资源罗列了集合中的资源分类。它的表述是一个以 `category` 作为根元素的 XML 文档，其中包含 `atom:category` 元素，媒体类型是 `application/atomcat+xml`。

下面的“问题讨论”部分有完整的范例。

问题讨论

服务文档的目的是让客户端找到服务器上可用的集合，工作区是一种可以简单归集相关集合的机制。一旦客户端知道了服务文档的 URI，它就可以找到该服务中所有集合的 URI。下面是一个例子：

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

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

<app:service xmlns="http://www.w3.org/2007/app" ❶
             xmlns:atom="http://www.w3.org/2005/Atom">

  <app:workspace> ❷
   <atom:title>CDs by Independent Artists and Reviews</atom:title>
   <app:collection href="http://www.example.org/cds" > ❸
     <atom:title>CDs</atom:title>
     <app:categories href="http://www.example.org/cds/categories"/>
     <app:accept>image/png</accept>
     <app:accept>image/jpeg</accept>
     <app:accept>image/gif</accept>
```

117

```
    </app:collection>
    <app:collection href="http://www.example.org/reviews"> ❹
      <atom:title>Reviews</atom:title>
    </app:collection>
  </app:workspace>
</app:service>
```

❶ 一个服务文档

❷ 一个工作区

❸ 一个 CD 集合

❹ 一个评论集合

这是一个由单一工作区组成的服务文档的表述。工作区是集合的一个逻辑分组。该工作区有两个集合：一个针对 CD，另一个针对评论。

CD 集合的 URI 是 `http://www.example.org/cds`。这个集合提供了 PNG、JPEG 和 GIF 图片，还有 Atom Entry。详见 6.5 节以了解如何管理此类媒体资源。客户端可以获取分类资源，通过它来发现这个集合提供的资源分类。

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

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

<app:categories xmlns:app="http://www.w3.org/2007/app"
                xmlns:atom="http://www.w3.org/2005/Atom"
                fixed="yes" scheme="http://www.example.org/audio">
  <atom:category term="jazz"/>
  <atom:category term="hip hop"/>
  <atom:category term="classical"/>
</app:categories>
```

工作区里的第二个集合是一个用来管理评论的评论集合。关于服务和分类文档的 Schema，详见 RFC 5023 的附录 B。

6.4 如何针对 Feed 和 Entry 资源使用 AtomPub 118

Atom Publishing Protocol（RFC 5023）是一种应用协议，用于编辑 Atom 文档。它描

述了如何创建、更新和删除 Atom Entry。它也支持编辑相关的非文本媒体，比如图像、存档文件等。如果您正在用 Atom 格式发布资源，并且那些资源是可编辑的，请考虑支持 AtomPub。

问题描述

您想知道如何使用 AtomPub 协议。

解决方案

允许客户端将 Atom Entry 文档作为 POST 请求的正文提交上来，以此创建新资源，把 Atom Feed 的 URI 当做工厂来使用。客户端随后可以使用带有 edit 关系类型的链接来修改（使用 PUT）或删除（使用 DELETE）资源。

当表述是一个 Atom Entry 文档时，给媒体类型添加一个参数 type=entry。

问题讨论

AtomPub 使用 HTTP 的子集来创建、获取、更新和删除资源。AtomPub 中的资源是 Atom Entry、Atom Feed 及媒体资源（详见 6.5 节）。AtomPub 使用 POST 创建新的资源，使用 GET 获取表述，使用 PUT 更新资源，使用 DELETE 删除资源。下面是一系列典型的操作：

```
# 创建资源的请求
POST /books HTTP/1.1 ❶
Content-Type: application/atom+xml;type=entry;charset=UTF-8

<atom:entry>
  <atom:id>urn:isbn:0-9767736-7-X</atom:id>
  <atom:title>Johnny Web Goes Out West</atom:title>
  <atom:published>2012-04-01T00:00:00Z</atom:published>
  <atom:author>
    <atom:name>R. W. Smith</atom:name>
    <atom:uri>http://example.org/books/1/authors/1</atom:uri>
  </atom:author>
  <atom:updated>2012-04-01T18:30:02Z</atom:updated>
  <atom:content type="text">
    Space hero Johnny Web tries
    to enjoy a vacation at a dude
    ranch only to be swept up in
    a criminal plot to sell unregulated
    solar power to off-worlders.
  </atom:content>
</atom:entry>

# 响应
```

119

```
HTTP/1.1 201 Created
Location: http://www.example.org/books/13

# 更新请求 ❷
PUT /books/13 HTTP/1.1
Host: www.example.org
Accept: application/atom+xml
If-Match: "h1g2f3d4s5a"

<atom:entry>
 ...

</atom:entry>

# 响应
HTTP/1.1 200 OK
Content-Type: application/atom+xml;type=entry;charset=UTF-8
Content-Length: XXX
ETag: "m1n2b3v4c5x6z"

<atom:entry>
  ...
</atom:entry>

# 删除请求 ❸
DELETE /books/13 HTTP/1.1
Host: www.example.org
If-Match: "m1n2b3v4c5x6z"

# 响应
HTTP/1.1 204 No Content
```

❶ 使用 POST 创建一个新资源。

❷ 使用 PUT 更新资源。

❸ 使用 DELETE 删除资源。

请注意，AtomPub 在 IANA 注册了 edit 关系类型。客户端可以使用带有这种关系类型的链接 URI 来获取、更新或删除资源。

AtomPub 还规定了 Slug 头。客户端可以在 POST 请求中使用这个标头来提供一个文本值，服务器可以将它作为 URI，分配给新创建的资源。详见 1.9 节的例子。

6.5 如何使用媒体资源

媒体资源是 AtomPub 引入的一种资源类型。媒体资源可以是 Atom Entry 文档之外的各种东西，可用于表示文档、图片、音频和视频文件等。由于媒体资源不是 Atom Entry 文档，AtomPub 为每个媒体资源关联了一个媒体链接资源。

120 媒体链接资源就是一个描述了媒体资源并链接到该资源的 Atom Entry。

问题描述

您想知道如何处理图像、音频/视频文件之类的媒体，把它们和 Atom Entry 关联起来。

解决方案

Atom 中的每个媒体资源都会有媒体链接 Entry。媒体资源有可能是二进制资源，可以用媒体链接 Entry 为各媒体资源提供元数据。

如果集合（即 Feed）支持媒体资源，在服务文档里列出支持的媒体类型，如 6.3 节所述。

客户端可以通过向集合提交 `POST` 请求来创建媒体资源。创建媒体资源和媒体链接资源。经由 `Location` 头返回媒体链接资源的 URI。在媒体链接资源的表述中，通过 `atom:content` 元素的 `src` 属性来提供新建的媒体资源。

如果服务器支持编辑媒体资源，在 Atom Entry 中包括一个带 `edit-media` 关系的链接。客户端可以使用这个链接来获取、更新或删除媒体资源。

问题讨论

通过使用媒体资源和与之相关的媒体链接 Entry 资源，AtomPub 支持在媒体资源（如图片、音频、视频文件等）上的读/写操作。媒体资源是真正的媒体。媒体链接 Entry 是一个 Atom Entry，它包含有关媒体资源的元数据，与其他 Atom Entry 一起出现在相关的 Atom Feed 中。

当客户端使用 AtomPub 添加了一个媒体文件时，服务器会同时创建媒体资源和媒体链接 Entry。下面是一个例子：

```
# 请求
POST /cds HTTP/1.1
Host: www.example.org
Content-Type: image/png
Slug: Epocalyptica
```

```
Content-Length: nnn

... 二进制数据 ...

# 响应
HTTP/1.1 201 Created
Content-Type: application/atom+xml;charset=UTF-8
Location: http://www.example.org/cds/112-epocalyptica ❶

Content-Location: http://www.example.org/cds/112-epocalyptica

<atom:entry xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:title>Epocalyptica</atom:title>
  <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</atom:id>
  <atom:updated>2009-04-01T04:01:00Z</atom:updated>
  <atom:author><atom:name>Jay Doe</atom:name></atom:author>
  <atom:summary type="text">Epocalyptica</atom:summary>
  <atom:content type="image/png"
                src="http://www.example.org/cds/112-epocalyptica.png"/> ❷
  <atom:link rel="self" href="http://www.example.org/cds/112-epocalyptica"/>
  <atom:link rel="edit-media"
             href="http://www.example.org/cds/112-epocalyptica.png"/> ❸
  <atom:link rel="edit"
             href="http://www.example.org/cds/112-epocalyptica"/> ❶
</atom:entry>
```

121

❶ 媒体链接资源

❷ 包含媒体元数据的媒体 Entry 资源

❸ 编辑媒体的链接

❹ 编辑媒体元数据的链接

媒体资源 URI(带 edit-media 关系类型的那个 URI)的目的是修改实际的媒体文件，正如下面的例子所示:

```
# 请求
PUT /cds/112-epocalyptica.png HTTP/1.1
Host: www.example.org
Content-Type: image/png
Content-Length: nnn

...二进制数据...

# 响应
HTTP/1.1 200 OK
```

客户端可以使用媒体链接 Entry URI（带 edit 关系的那个）来修改与媒体文件相关联的元数据。下面的例子为与媒体资源相关的媒体链接条目增加了内容摘要：

```
# 请求
PUT /cds/112-epocalyptica HTTP/1.1
Content-Type: application/atom+xml;charset=UTF-8
If-Match: "z9x8c7v6b5n4m3"

<atom:entry xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:title>Epocalyptica</atom:title>
  <atom:id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</atom:id>
  <atom:updated>2009-04-01T04:01:00Z</atom:updated>
  <atom:author><name>Jay Doe</name></atom:author>
  <atom:summary type="text">Coolest album ever</atom:summary>
  <atom:content type="image/png"
           src="http://www.example.org/cds/112-epocalyptica.png"/>
  <atom:link rel="self"
           href="http://www.example.org/cds/112-epocalyptica"/>
  <atom:link rel="edit-media"
           href="http://www.example.org/cds/112-epocalyptica.png"/>
  <atom:link rel="edit"
           href="http://www.example.org/cds/112-epocalyptica"/>
</atom:entry>

# 响应
HTTP/1.1 204 No Content
```

122

这个例子说明了如何处理任意媒体内容，以及如何管理媒体的元数据。您可以在各种涉及非文本资源及其元数据的场景下应用这种模式。这样的例子包括电影预告片、报告、演讲幻灯片、电子表格、扫描文档等。

第 7 章

内容协商

123

内容协商（Content Negotiation），有时也简写为 conneg，该过程用于当存在多个可用的资源表述形式时，为客户端选择一个最好的出来。内容协商常用于标明媒介类型的偏好，但它也可以用于标明本地化语言、字符编码以及压缩方面的偏好。

HTTP 指定了两种类型的内容协商。分别是服务器驱动（server-driven）的协商和代理驱动（agent-driven）的协商。前者使用请求头（request headers）来进行选择，后者则使用不同的 URI。

本章讨论了与内容协商相关的如下内容:

7.1，“如何标明客户端偏好（Client Preferences）”

当请求一个资源时，应如何决定包含哪些 Accept-*头及其取值。

7.2，“如何实现媒体类型协商（Media Type Negotiation）”

通过本节了解如何实现服务器，正确地解释用于媒体类型协商的那些 Accept 请求头。

7.3，“如何实现语言协商（Language Negotiation）”

通过本节了解如何使用 Accept-Language 头来实现语言协商。

7.4，“如何实现字符编码协商（Character Encoding Negotiation）”

通过本节了解如何为资源表述确定请求所用的字符编码。

7.5，“如何支持压缩”

HTTP 允许客户端通过 Accept-Encoding 请求头来标明其对压缩表述的偏好。本节将说明如何在服务器上处理这样的标头。

7.6，“何时及如何发送 Vary 头”

了解如何使用 Vary 头。

124 7.7，“如何处理协商失败”

了解当首选的表述形式不可用时，何时及如何返回错误信息。

7.8，“如何使用代理驱动的内容协商”

代理驱动的内容协商为客户端提供了另一种选择，用于请求资源的某种特定表述。通过本节了解何时以及如何使用它。

7.9，“何时支持服务器驱动的协商”

通过本节了解为多种表述提供支持的优点和缺点。

7.1 如何标明客户端偏好

实现客户端时，一件重要的事情就是该客户端要能够向服务器标明自己的偏好和能力，包括它所能够处理的表述格式，所偏好的语言，能够识别的字符编码，以及它是否支持压缩。对于响应中的表述，即便事先知道它的格式、字符编码、语言以及压缩类型，清楚明确地标明客户端的偏好以及能力将更有助于客户端适应各种变化。否则，当服务器以另外的表述格式提供资源时，您所用到的 HTTP 库的默认偏好设定可能会提示服务器，让它返回另外一种表述，这样的话，客户端就会出错。因此，最好是要求一个确定的表述，而不是采用默认表述，因为这个默认设定是会变的。

问题描述

您想知道如何允许客户端标明它自己的能力，例如所能支持的媒体类型、语言等。

解决方案

发起一个请求时，添加一个 Accept 头，其值是一个偏好媒体类型的列表，以逗号分隔。根据客户端对各种媒体类型的偏好程度，可以再设定一个参数 q。这个参数标明了客户端对那些在 Accept-*头中所列出的各个媒体类型的相对偏好程度，它常和 Accept 头一起用。如果客户端只能处理某些特定的格式，那就在 Accept 头中添加“*;q=0.0”，来向服务器标明自己无法处理那些没有列在 Accept 头里的媒体类型。

如果客户端仅能处理属于某一种字符集的字符，那就增加一个 Accept-Charset 头，其值就是所偏好的字符集。否则，就不要添加这个头。

添加 Accept-Language 头来指明表述所偏好的语言。

125 如果客户端能够解压那些采用诸如 gzip，compress 或者 deflate 编码的压缩表述，就添加一个 Accept-Encoding 头并列出所支持的编码格式，否则就略过该标头。

问题讨论

在 HTTP 中，Accept-*头存在的目的是为了让客户端可以表达其对于响应表述的偏好。服务器基于其自身的能力，会评估客户端的偏好，然后确定合适的表述来返回。因为这个过程的输出是由服务器决定的，所以称之为服务器驱动的内容协商（server-driven negotiation）。

例如，假设一个客户端具有下列偏好：

- 客户端希望表述语言是法语，但是也可以接受英语的表述。
- 客户端可以处理 Atom 格式的表述，其媒体类型是 application/atom+xml，也可以接受媒体类型为 application/xml 的 XML 格式的表述，但除此之外的格式就不能接受了。
- 客户端知道如何处理经 gzip 格式压缩的表述。

客户端可以通过以下的请求头来标明这些偏好：

```
# 请求头
Accept: application/atom+xml;q=1.0, application/xml;q=0.6, */*;q=0.0
Accept-Language: fr;q=1.0, en;q=0.5
Accept-Encoding: gzip
```

在这些标头中，分号之后的部分就是 q 参数。这个参数的值是一个浮点数，通常带一位小数，不过 HTTP 1.1 允许至多到小数点后三位。客户端可以使用该参数来标明其对各个选项的相对偏好程度，从 0.0（无法接受）到 1.0（最为理想）。例如，上面出现的 Accept 头标明了该客户端无法处理除了 Atom 和 XML 之外的表述格式。q 参数的默认值是 1.0。

并非所有的服务器都支持 q 参数。此类服务器可能会从 Accept 头中选择第一个它所支持的媒体类型。

请注意，服务器并不总是能够完全地或者正确地支持内容协商。客户端应当做好这样的准备，即所收到的表述并不符合 Accept-*头中的定义。3.2 节讨论了如何使用诸如 Content-Type 这样的实体头来确定该怎样处理响应表述。

Accept-*头，比如 Accept 和 Accept-Language，表达的是一组范围内的媒体类型、语言等。

126

反之，Content-*头表达的是一个确定的媒体类型、语言等。

7.2 如何实现媒体类型协商

无论服务器是只支持一种媒体类型，还是支持多种媒体类型，正确地解释 Accept 头对于提升互操作性而言是必不可少的。

问题描述

您想知道如何决定响应中的表述应该使用哪种媒体类型。

解决方案

如果请求中没有 Accept 头，那就使用默认格式来返回被请求资源的表述。

如果请求中含有 Accept 头，那就解析它，并按照 q 参数降序排序媒体类型的值。然后从此列表中选择一个服务器所支持的媒体类型。在响应中要包含一个 Vary 头，详见 7.6 节。

如果列表中的所有媒体类型服务器都不支持，那就使用 7.7 节里介绍的方法来确定一个合适的响应。

问题讨论

考虑以下的 Accept 头：

```
Accept: application/atom+xml;q=1.0, application/xml;q=0.6, text/html
```

这里包含了三种媒体类型，其中两种附带不同值的 q 参数，第三种则没有 q 参数值。而 q 参数的默认值是 1.0，所以上述 Accept 头其实等同于下面这个：

```
Accept: application/atom+xml;q=1.0, application/xml;q=0.6, text/html;q=1.0
```

从这个 Accept 头中，服务器端的首选应当是 application/atom+xml 或者 text/html，第二选择应当是 application/xml。如果三种都支持，那服务器端返回的表述可以是 Atom 或 HTML 中的任意一种格式。根据 Accept 头的值，这两者都可以被客户端接受：

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

... 表述 ...
```

127 尽管这个逻辑实现起来看似简单，但同样存在某些情况，可能破坏与客户端的互操作性。考虑下述情形。

一开始，假设服务器端所有的表述都只支持 application/xml 格式。由于没有其他格式的表述了，它就会忽略 Accept 头，而对所有的请求都返回 XML 格式的表述。然后，由于客户端要求，它添加对 application/json 格式的支持并决定检查 Accept 头的值。这导致了下面的情况：

```
# 原来的请求
GET /movie/gone_with_the_wind HTTP/1.1
Host: www.example.org
Accept: application/json

# 服务器只支持 XML
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

... xml ...

# 现在的相同请求
GET /movie/gone_with_the_wind HTTP/1.1
Host: www.example.org
Accept: application/json

# 响应——服务器同时支持 JSON 和 XML
HTTP/1.1 200 OK
Content-Type: application/json

... json ...
```

这里破坏了兼容性，因为曾经正常工作的客户端不再能正常运作了。当面临这样的情况时，应该使用一个新的 URI 来提供新的表述。例如：

```
# 现在的相同请求
GET /movie/gone_with_the_wind?format=json HTTP/1.1
Host: www.example.org
Accept: application/json

HTTP/1.1 200 OK
Content-Type: application/json

... json ...
```

这也称为代理驱动的协商（agent-driven negotiation），详见 7.8 节。

7.3 如何实现语言协商

HTTP 对语言协商的支持有助于为 Web 服务提供有限的本地化支持。语言选择只是

128 本地化的一方面。除了在表述中将人们阅读的文字进行翻译以外，本地化还经常包括根据区域和文化特征对信息进行相应的修改。

问题描述

您想知道在一个表述中如何确定人们阅读的文字所使用的语言。

解决方案

如果请求中不含有 Accept-Language 头，那就在返回的表述中对所有给人阅读的文字使用默认的语言。

如果请求中含有 Accept-Language 头，那就解析它，依照 q 参数对语言进行排序，选择列表里的第一个服务器所支持的语言。在响应中包含一个 Vary 头，详见 7.6 节。

如果列表里的语言服务器端一个也不支持，并且 Accept-Language 头也不包含“*;q=0.0”，那就在响应里使用默认的语言。

问题讨论

语言协商的协议与媒体协商是类似的。客户端在 Accept-Language 头中提供它所能接受的语言以及 q 参数值，以此表达它的意图，服务器依此决定在响应中使用哪种语言。

```
# 请求
GET /movie/gone_with_the_wind HTTP/1.1
Host: www.example.org
Accept-Language: en,en-US,fr;q=0.6

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

<movie>
  <title>Gone with the Wind</title>
  <year>1936</year>
  ...
</movie>
```

当不同语言的表述之间的区别仅仅是在其中的供人阅读的文字上时，这个方法就最为合适了，就像下面这个表述：

```
# 请求
GET /movie/gone_with_the_wind HTTP/1.1
```

```
Host: www.example.org
Accept-Language: en,en-US,fr;q=0.6

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

<movie>
  <title>Autant en emporte le vent</title>
  <year>1936</year>
   ...
</movie>
```

129

如果表述之间的区别更加显著，那就使用其他本地化方式，比如客户端的 IP 地址或者特定区域/语言的 URI。

7.4 如何实现字符编码协商

如果客户端要求（通过 Accept-Charset 头）文本的表述使用某个特定的字符编码，那么照此行事会增进互操作性。

问题描述

您想知道要对响应里的文本表述使用何种字符编码。

解决方案

如果请求中没有 Accept-Charset 头，那就以 UTF-8 对返回的表述进行编码。

如果请求中含有 Accept-Charset 头，那就解析它，依照 q 参数对那些字符集进行排序，然后选择服务器所支持的字符编码。

如果服务器端不支持任何所请求的字符集，并且 Accept-Charset 头中不包含“*;q=0.0”，那么就使用 UTF-8 编码返回的表述。

所有这些情况中，如果媒体类型是文本的，并且允许使用 charset 参数，就在 Content-Type 头中包含 charset 参数，以标明服务器所用的字符编码。同样，在响应中也要包含一个 Vary 头，详见 7.6 节。

问题讨论

绝大多数的平台和编程语言都支持 UTF-8。UTF-8 也是 application/xml 和

application/json 媒体类型的默认编码。除非客户端明确要求其他编码，否则应该总是使用 UTF-8。

130

避免使用 text/xml，因为它的默认编码是 US-ASCII。

7.5 如何支持压缩

服务器可以提供经过压缩的表述，其格式包括 gzip，deflate 或者 compress。在 HTTP 中，这通常被称为内容编码（content encoding）。

问题描述

您想知道何时需要压缩表述。

解决方案

如果服务器支持压缩响应内容，那就从 Accept-Encoding 头中选择一种压缩技术。在响应中同样也要包含一个 Vary 头，详见 7.6 节。如果服务器所支持的压缩编码没有一个位于标头中，那就忽略它。对于 q 参数的处理同其他 Acceopt-*的情况。

如果请求中不包含 Accept-Encoding 头，就不要对表述进行压缩。

问题讨论

客户端可能支持也可能不支持像 gzip 或 deflate 这样的内容编码。因此，重要的是，只有当客户端发送了一个 Accept-Encoding 头，并且其中含有服务器所支持的压缩格式时，才返回压缩过的响应。这里是一个范例：

```
# 请求
GET /movie/gone_with_the_wind HTTP/1.1
Host: www.example.org
Accept-Encoding: gzip

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

... gzip 压缩后的字节 ...
```

在大多数情况下，HTTP 服务器可以被配置为自动为响应运用某种给定的编码。例如，在 Apache HTTP 服务器的配置文件中加上下面的内容，就可以告诉服务器为所有的 application/xml 表述用上 deflate 编码：

```
AddOutputFilterByType DEFLATE application/xml
```

7.6 何时以及如何发送 Vary 头

131

当服务器使用内容协商来选择表述时，根据 Accept-*头的不同，同一个 URI 可以产生不同的表述。Vary 头告诉客户端服务器在选择表述时使用了哪些请求头。

问题描述

您想知道如何使用 Vary 头来告知客户端服务器是如何选择某个表述的。

解决方案

每当针对一个资源有多种形式的表述可用时，就要包含一个 Vary 头。该标头的值是一些请求头的列表，以逗号分隔，服务器在选择表述时参考了这些请求头。如果服务器还使用了除这些请求头之外的信息，例如客户端的 IP 地址、当前时间、用户个性化设置等，那就将 Vary 头的值设置为*。

问题讨论

服务器可以用 Vary 头来通知客户端服务器驱动内容协商的结果。Vary 头的值是一组请求头，而不是响应头。例如，考虑下面的请求和响应序列：

```
# 针对英语表述的请求
GET /status HTTP/1.1
Host: www.example.org
Accept-Language: en;q=1.0,*/*;q=0.0

# 响应
HTTP/1.1 200 OK
Content-Language: en
Vary: Accept-Language

...

# 针对德语表述的请求
GET /status HTTP/1.1
Host: www.example.org
Accept-Language: de;q=1.0,*/*;q=0.0
```

```
# 响应
HTTP/1.1 200 OK
Content-Language: de
Vary: Accept-Language

...

# 针对法语表述的请求
GET /status HTTP/1.1
Host: www.example.org
Accept-Language: fr;q=1.0,*/*;q=0.0

# 响应
HTTP/1.1 200 OK
Content-Language: fr
Vary: Accept-Language
```

132

尽管请求 URI 保持不变，但客户端及中间媒介可以根据 Vary 头中所列出的请求头的值来区分不同的响应。缓存把这个标头作为缓存键的一部分，来维持表述的不同副本。客户端可以使用这个信息来获知服务器用于内容协商的标准。

7.7 如何处理协商失败

对于一个给定的资源，服务器可以随意地提供任意可用的表述。然而，客户端可能会无法处理某些特定的媒体类型。除了浏览器以外，大多数 HTTP 客户端都只能处理一或两种格式。

问题讨论

您想知道当服务器无法提供客户端所偏好的表述时，是该提供一个默认的表述，还是该返回一个错误。

解决方案

如果服务器无法提供满足客户端偏好的表述，并且客户端明确包含了一个“*/*;q=0.0”，那就返回状态码 406（Not Acceptable），并在表述主体中包含表述的列表。

如果服务器不能支持所请求的 Accept-Encoding 值，就不要应用任何内容编码，直接提供表述。

问题讨论

在这个例子中，客户端只能处理 application/json 这一种媒体类型：

```
# 请求
GET /user/001/followers HTTP/1.1
Accept: application/json,*/*;q=0.0 ❶

# 响应
406 Not Acceptable ❷
Content-Type: application/json
Link: <http://www.example.org/errors/mediatypes.html>;rel="help" ❸

{
  "message" : "This server does not support JSON. See help for alternatives."
}
```

❶ 客户端仅能处理 JSON 这一种格式。

133

❷ 服务器不支持 JSON 格式。

❸ 该链接指向关于所支持格式的帮助信息。

在本例中，服务器识别出 JSON，但是无法以这种格式提供表述。由于客户端的请求中对除了 application/json 之外的媒体类型都含有 q=0.0，所以对客户端而言失败是能接受的。

请注意，服务器给出的错误信息本身使用了 JSON 格式。这是合理的，服务器完全可以使用常见的格式来实现错误信息。当然，也可以使用适合阅读的 HTML 格式。

```
# 请求
GET /user/001/followers HTTP/1.1
Accept: application/json,*/*;q=0.0

# 响应
406 Not Acceptable
Content-Type: text/html;charset=UTF-8
Link: <http://www.example.org/errors/mediatypes.html>;rel="help"

<html>
  <head>
    <title>JSON Not Supported</title>
  </head>
  <body>
    <p>This server does not support JSON. See <a
    href="http://www.example.org/errors/mediatypes.html">help</a> for
    alternatives.</p>
```

```
  </body>
</html>
```

7.8 如何使用代理驱动的内容协商

虽然服务器驱动的协商是内置于 HTTP 中的，但它也有一些局限：

- 内容协商在表述中不包含诸如货币单位、距离单位、日期格式以及其他与区域相关的可读文字的变化。举例来说，没有办法根据语言偏好确定货币和日期格式。
- 在某些情况下，由于本地化需求的复杂性，服务器可能会为不同的语言环境准备不同的资源。
- 常见的 Web 浏览器针对 Accept 头使用了大范围的媒体类型。例如，某些配置下的 Firefox 浏览器会发送 `Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8`。这就使得要在浏览器中查看经过了内容协商后的表述变得十分困难。

对于这些情况，就应该使用代理驱动的协商（agent-driven negotiation）。当客户端不能通过 Accept-*头来传达其偏好的时候，代理协商就变得有用了。

134

问题描述

您想知道如何实现代理驱动的协商。

解决方案

为每个表述都提供一个 URI。

问题讨论

代理驱动的协商其实就是为每种表述提供一个单独的 URI，使得客户端可以通过此 URI 直接访问它想要的表述。在代理驱动协商中，客户端使用服务器给出的信息来决定使用哪个 URI。如果表述存在，服务器就返回之，否则就返回一个 404(Not Found) 响应码。

因为是由客户端来决定过程的结果，所以这一技术被称为代理驱动的协商。此处的术语“代理”指的是用户代理，而最常见的用户代理就是浏览器。

尽管为所有的 Accept-*头都实现代理驱动的协商是可能的，但从实用上讲，它最常

用在媒体类型和语言方面。

对于服务器而言，有若干种方式可以为资源的每种语言和媒体类型分别指定 URI。其中一些常用的方法包括：

查询参数

将语言或者媒体类型以查询参数的形式添加在主 URI 的后面，这些参数的值使用媒体类型的简写方式。例如，`format` 参数用于支持媒体类型协商，`lang` 参数用于语言协商：

```
http://www.example.org/status?format=json
http://www.example.org/status?format=xml
http://www.example.org/status?format=csv
```

***URI* 扩展**

在主 URI 后面添加一个句点（.）以及媒体类型的简写。例如，`status.atom` 用于一个 `application/atom+xml` 的表述，`status.json` 用于 `application/json` 的表述。

子域名

创建子域名来支持特定语言的表述。例如，`en.wikipedia.org` 提供维基百科英文版条目的表述，`de.wikipedia.org` 则提供德文版条目的表述。

当使用代理驱动协商时，服务器可以选择使用 `alternate` 链接关系类型的链接来宣告替代内容，示例如下：

135

```
<status xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="self" href="http://www.example.org/status?format=xml&lang=en"/>❶
  <atom:link rel="alternate" type="application/json"
        href="http://www.example.org/status?format=json&lang=fr"/> ❷
  ...
</status>
```

❶ 指向资源的链接

❷ 指向替代内容的链接

7.9 何时支持服务器驱动的协商

内容协商并非总是合适的。尽管一些流行的 Web 服务以及 Web 服务框架支持将诸如 XML、JSON 和 Atom 等通用的格式用于每个资源，但也应该考虑在自己的 Web 服务中因支持多种格式所造成的开销。

问题描述

您想知道服务器协商是否适合自己的 Web 服务。

解决方案

仅在客户端需要时，或者每种表述都包含相同信息时，支持多种表述。如果所包含的信息不同，则应该为每个表述使用不同的 URI。

问题讨论

只有当您所用的开发框架提供支持时，内容协商才会容易实现。否则，实现、测试以及管理内容协商都将花费很多时间和精力。大多数的客户端仅能处理单独一种格式。在这样的情况下，支持多种格式就不必要了。在考虑为每种资源都支持多种表述之前，考虑下列事项：

- 在某些情况下，对于每种不同的表述格式，应用程序的流程可能会有所不同。尤其是 HTML 格式的表述。用户界面上的制约可能会需要 HTML 表述遵循一个不同于其他格式（例如 XML）的应用程序流程。这种情况下，对 HTML 和 XML 格式都提供服务器驱动的协商就是不现实的。
- 如果 `Accept` 头含有多种媒体类型及不同的 `q` 参数，那么基于此返回明确的资源并非易事。不是所有的开发框架都支持这一功能。
- 136 语言协商对于全球化服务而言可能过于简单。在某些情况下，法律和商业方面的需求会因地区而不同，此时代理驱动的协商大概才是最佳选择。
- 缓存可能无法很好地处理内容协商。一些缓存可能会忽略或限制每种资源的变体的存在数量。

综上所述，在为服务器应用程序提供对服务器协商的支持之前，要仔细地考量实际需求。

第 8 章

查询

137

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:

8.1 节，“如何针对查询设计 URI”
本节展示了如何针对查询来设计 URI。

8.2 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。

8.3 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。

8.4 节，“如何存储查询”
使用本节中的内容来实现存储查询。

8.1 如何针对查询设计 URI

138

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。可使用14.1节中的内容，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“Jane”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：

139

```
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“war”，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
```

140

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。

8.2 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。如 9.1 节中描述的那样，需要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：

141

```
# 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>
```

142

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。

8.3 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&...
```

143

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&amp;

# 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>
```

144

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 8.4 节中的内容在服务器上存储查询。

8.4 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。8.3 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
```

145

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 8.3 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。

第 9 章

Web 缓存

147

缓存是构建于 HTTP 统一接口之上的最有用的功能之一。可以利用缓存减少终端用户感知到的延时，增加可靠性，减少带宽使用和成本，降低服务器负载。缓存无处不在，可以在服务器网络里、内容分发网络（content delivery network，简称 CDN）或是客户端网络里（通常被称为转发代理[注1]）。

通常所说的缓存可以是类似 memcached（*http://memcached.org/*）这样的对象缓存，或者 Squid（*http://www.squid-cache.org/*）和 Traffic Server（*http://incubator.apache.org/projects/trafficserver.html*）这样的 HTTP 缓存。这两类缓存都可以改善性能，在整个 Web 服务的部署架构中扮演着重要角色。但两者有一个重要的区别，Squid 这样的 HTTP 缓存不需要客户端或服务器调用任何特殊的编程 API 来管理缓存中的数据，对象缓存的情况就不同了。举例来说，要使用 memcached，必须使用 memcached 的编程 API 来存储、获取并删除对象；而 HTTP 缓存则是基于与客户端和服务器相同的统一接口。因此，只要遵循规定使用 HTTP，应该可以在无须改变代码的情况下添加缓存层。

因为缓存既可以是一个 HTTP 客户端，也可以是服务器，所以在与缓存有关的讨论中，源服务器一词用于区分缓存服务器与托管代码的服务器。

以下是本章讨论的内容:

9.1 节，“如何设置过期缓存头”

过期缓存头（Expiration Caching Header）控制着是否要为客户端缓存表述，以及缓存多少时间。通过本节了解如何对这些 HTTP 头进行设置。

注 1：转发代理，即 Forward Proxy。代理服务器分为两类，通常所说的代理服务器即指转发代理，另一类是反向代理（Reverse Proxy）。

PDG

9.2 节，“何时设置过期缓存头”

只有特定的 HTTP 响应可以被缓存，通过本节了解何时设置过期缓存头。

148 9.3 节，“何时以及如何在客户端使用过期缓存头”

通过本节了解客户端应该如何处理过期缓存头。

9.4 节，“如何支持复合资源的缓存”

通过本节了解如何针对复合资源的缓存做出权衡。

9.5 节，“如何保持新鲜且温暖的缓存”

通过本节了解一些如何将缓存保持在非空及最新状态的窍门。

9.1 如何设置过期缓存头

当缓存可以在不访问源服务器时做出尽可能多的响应时，它是最高效的。设计过期缓存（Expiration Caching）就是为了降低源服务器收到的请求数量，同时减少应用程序使用的带宽。

过期缓存基于 Cache-Control 和 Expires 这两个头，它们指导客户端和缓存在一段指定的时间内保存从服务器返回的表述副本。在这个时间窗口以内，甚至超出该时间窗口，缓存可以对后续请求做出响应，无须访问源服务器。

问题描述

您想知道如何为您的资源表述开启缓存。

解决方案

基于更新的频率，决定缓存可以提供表述的时间周期。在这个时间周期内，缓存是最新的，在此之后，缓存的表述就变旧了。

在提供表述时，包含一个 Cache-Control 头，其中带有一个与新鲜寿命（freshness lifetime）相同的 max-age 值（单位为秒）。Cache-Control 是一个 HTTP 1.1 头，为了支持遗留的 HTTP 1.0 缓存，还要包含一个带过期日期时间的 Expires 头。过期时间是服务器生成表述的时间加上新鲜寿命。还要包含一个带有日期时间的 Date 头，即服务器返回响应的时间。这个标头能帮助客户端计算新鲜寿命，即为 Expires 和 Date 两值之差。

如果您决定不让缓存提供副本，则加上一个带 no-cache 值的 Cache-Control 头。这里还要加上 Pragma: no-cache 头来支持遗留的 HTTP 1.0 缓存。

下面列表展示了 Cache-Control 指令及其适用性：

public

这是默认指令。当请求是经过身份验证的，但您仍希望共享缓存提供缓存响应时，也可以使用该指令。

private

当响应专属于某个客户端或用户时，使用该指令。出现这个指令，任意客户端缓存（例如，浏览器缓存或转发代理）都可以缓存表述，但诸如服务器缓存或网络缓存之类的共享缓存则不能进行缓存。

在基于客户端或用户身份验证来提供表述的时候添加该指令。

149

no-cache 和 no-store

通过这些指令可以避免缓存存储或提供已经缓存的响应。

max-age

该指令的值即为新鲜寿命，单位为秒。

s-maxage

这个指令与 max-age 类似，但只对共享缓存有意义。在源服务器同时设置了 max-age 和 s-maxage 的时候，缓存会使用 s-maxage 头。实际上，单独设置 max-age 指令就足够了。

must-revalidate

这个指令要求缓存在提供陈旧的表述前先检查源服务器。

proxy-revalidate

这个指令与 must-revalidate 指令类似，但它只应用于共享缓存。

问题讨论

服务器可以通过两个 HTTP 头来控制过期缓存：Expires（HTTP 1.0）头和 Cache-Control（HTTP 1.1）头。下面这个例子指示缓存提供一个小时的表述：

```
# 响应
HTTP/1.1 200 OK
Date: Sun, 09 Aug 2009 00:56:14 GMT ❶
Last-Modified: Sun, 09 Aug 2009 00:56:14 GMT ❷
Expires: Sun, 09 Aug 2009 01:56:14 GMT ❸
Cache-Control: max-age=3600,must-revalidate ❹
Content-Type: application/xml; charset=UTF-8
```

```
...
```

❶ 服务器生成响应的日期时间

❷ 表示服务器最后修改表述的日期时间

❸ 过期日期时间值

❹ 过期寿命和其他指令

150 这个消息有一个 Cache-Control 头，其中包含设置为 3600 秒的 max-age 指令。还要注意 Expires 和 Date 头，两者之差是 3600 秒。因为存在 must-revalidate 指令，一小时后，缓存会向服务器发出一个条件请求，在向客户端再次提供表述前重新验证响应的内容（详见 10.3 节）。

在一小时的过期时间之内，客户端向同一资源发起另一个请求时，缓存会提供响应的副本，通过提供存储的表述副本来响应请求，而不是访问源服务器。

```
# 第一个请求
GET /person/joe HTTP/1.1
Host: www.example.org

# 第一个响应
HTTP/1.1 200 OK
Date: Sun, 09 Aug 2009 00:44:14 GMT
Last-Modified: Sun, 09 Aug 2009 00:40:14 GMT
Expires: Sun, 09 Aug 2009 01:44:14 GMT
Cache-Control: max-age=3600,must-revalidate

...

# 10 分钟后的第 2 个请求
GET /person/joe HTTP/1.1
Host: www.example.org

# 第 2 个响应——由缓存返回
HTTP/1.1 200 OK
Date: Sun, 09 Aug 2009 00:54:14 GMT
Last-Modified: Sun, 09 Aug 2009 00:40:14 GMT
Expires: Sun, 09 Aug 2009 01:44:14 GMT
Cache-Control: max-age=3600,must-revalidate
Age: 600

...
```

这个响应中的 Age 头是缓存添加的，它标明了缓存是在多久以前从源服务器上获取该表述的。看到应答中的这个 Age 头之后，客户端就能计算出它获取的是一个 10 分钟之前的副本。如果这个值大于 max-age，则说明表述已经旧了。

最佳过期缓存的关键是为资源表述计算一个合理的新鲜寿命值。如果有历史信息，例如表述的更新日志，可以参考它们来建立基线寿命。如果没有此类数据，可以先从一个合理的推测值出发，在获取更多信息后对其做出调整。通常这些信息源自于您发现某个客户端看不到最近更新的表述。

添加 no-cache 或 no-store 指令可以防止缓存提供了已经缓存的响应。如非必要，请不要使用这些指令。通常会使用一个小的 max-age 值，而非添加 no-cache 或 no-store 指令，这能帮助客户端在至少一小段时间内获取缓存的副本，并且不需要在表述的新鲜程度上做出太多妥协。

151

类似 Squid 这样的缓存还支持两个 Cache-Control 头的扩展指令——stale-if-error 和 stale-while-revalidate。服务器可以用 stale-if-error 来告诉缓存，它们可以在指定时间间隔内继续提供陈旧的响应。

```
# 响应
HTTP/1.1 200 OK
Cache-Control: max-age=3600, stale-if-error=600

...
```

在这个例子里，响应会在一小时后变旧，但如果缓存在过期后访问源服务器时发生了错误，它还可以多提供 10 分钟的陈旧响应。

下面的例子演示了 stale-while-revalidate:

```
# 响应
HTTP/1.1 200 OK
Cache-Control: max-age=3600, stale-while-revalidate=600

...
```

这个扩展允许缓存以异步的方式重新验证响应，与此同时，可以再提供最多 10 分钟的陈旧响应。在 10.3 节中可以了解更多关于重新验证的内容。

9.2 何时设置过期缓存头

不是 HTTP 中的每个请求都能缓存的。HTTP 规定了什么是能缓存的，什么是不能

缓存的，缓存也可能只实现了部分 HTTP 缓存协议。本节将告诉您什么是可以缓存的，什么是不可以缓存的。

问题描述

您想知道何时在响应中包含过期缓存头。

解决方案

为所有带成功响应码的 GET 和 HEAD 请求的响应设置过期缓存头。虽然 POST 是可缓存的，但缓存会把该方法视为不可缓存的。其他方法无须设置过期缓存头。

152 除了带 200 (OK) 响应码的成功响应，还可以考虑缓存下面这些带 3xx 和 4xx 响应码的 HTTP 头，这能减少由客户端错误带来的流量。这种做法被称为消极缓存（negative caching）。

- `300 (Multiple Choices)`

  带这个状态码的表述不太可能经常发生变化，缓存此类应答可以降低服务器负载。

- `301 (Moved Permanently)`

  当资源永久移动时，那些将 URI 存储在数据库中的客户端可能不会修改这些 URI。在这种情况下，缓存可以不用访问源服务器，直接提供重定向响应。

- `400 (Bad Request)`

  当服务器返回该状态码时，客户端按理说不该重复请求，但出于软件错误或恶意攻击等原因，有些客户端会重复发起请求。

- `403 (Forbidden)`

  如果服务器永久拒绝服务该资源，请为其添加缓存。

- `404 (Not Found)`

  该资源不存在，而且没有必要让服务器仅为失败生成表述。

- `405 (Method Not Allowed)`

  客户端可能会因为软件错误而重复发送此类请求。

- 410（Gone）

  资源不再存在了，所以缓存应该尽可能久地提供错误响应。

问题讨论

根据 HTTP 1.1，GET，HEAD 和 POST 方法的响应是可缓存的，而且缓存可以存储并提供任意 GET 和 HEAD 请求的响应，除非使用 Cache-Control: no-cache 这样的缓存指令明确禁止缓存。就算包含了缓存头，也不会缓存 POST 请求的响应。

除此之外，在客户端提交 HEAD 请求时，大多数缓存会以 GET 请求的方式将其转发到源服务器，如果响应是可缓存的，则进行保存，然后返回一个空的响应给客户端。

```
# 来自客户端的请求
HEAD /person/joe HTTP/1.1
Host: www.example.org

# 从缓存发往源服务器的请求
GET /person/joe HTTP.1.1
Host: www.example.org

# 源服务器返回给缓存的响应
HTTP/1.1 200 OK
Cache-Control: max-age=3600
Date: Sun, 09 Aug 2009 00:56:14 GMT
Last-Modified: Sun, 09 Aug 2009 00:56:14 GMT
Expires: Sun, 09 Aug 2009 01:56:14 GMT

...正文...

# 缓存发给客户端的响应
HTTP/1.1 200 OK
Cache-Control: max-age=3600
Date: Sun, 09 Aug 2009 00:56:14 GMT
Last-Modified: Sun, 09 Aug 2009 00:56:14 GMT
Expires: Sun, 09 Aug 2009 01:56:14 GMT

...
```

153

因此，如果您将 GET 的响应标记为可缓存的，那么缓存就可以响应 GET 和 HEAD 这两种请求了。

值得一提的是，在消极缓存 404（Not Found）和类似响应码时，为这些响应分配自己的（通常是短期的）新鲜寿命是很常见的做法，就算源服务器的响应没有明确的新鲜寿命时也是如此。这么做的目的一般是为了在不访问源服务器的情况下，快速

处理错误的请求。

9.3 何时以及如何在客户端使用过期缓存头

前两节从服务器的视角讨论了过期缓存头，本节将从客户端的视角来讨论它们。

问题描述

您想知道是否一定要在客户端显式地实现对缓存的支持。

解决方案

除非您是在构建一个封装在压缩包内的、需要用户安装并运行的客户端应用程序，否则应该避免在客户端应用程序中实现对过期缓存的支持。可以在客户端网络中部署一个转发代理缓存，避免在客户端代码中实现自己的缓存层。

问题讨论

通常情况下，客户端应该独立于过期缓存之外。理论上是有可能构建支持 HTTP 缓存协议的客户端应用程序的。举例来说，常见的浏览器实现了过期缓存，并将表述154存储在内存或文件系统之中。实际上，在客户端应用程序中构建并维护一个缓存是项很麻烦的工作。它涉及正确地实现 no-store，no-cache 和 must-revalidate 这样的过期指令，还要遵循 Vary 头的内容。在同一个运行时内放入缓存还会造成客户端应用程序代码与缓存代码争夺内存与 CPU 资源，这会让客户端的调优变得更加困难。况且，实现中的任何错误都会导致客户端的安全问题。

比较之下，在客户端和服务器之间放置一个转发代理缓存更容易一些。不涉及任何开发活动，您就可以坐享经过精心测试的、健壮的缓存基础设施所带来的好处，无须自己构建。这么做还为日后扩展留下了空间，您可以架设一个所有客户端共享的转发代理服务器集群。

就算您的客户端库程序支持编程式缓存，也要避免这种做法，而应该将所有的缓存活动交给转发代理。

如果客户端和服务器在同一个网络里，不一定需要转发代理。服务器上可以部署一个被所有客户端所共享的缓存。但是，如果客户端是在和一个其他网络中的第三方 Web 服务交互，那么使用转发代理可以帮助减少客户端与服务器之间的往返请求次数。

9.4 如何支持复合资源的缓存

问题描述

您想知道如何为复合资源开启过期缓存。

解决方案

基于那些对新旧程度有强烈要求的数据来制定缓存决策，根据它们的变更频率来设置过期头。

问题讨论

相比其他类型的资源，为复合资源实现缓存更为复杂，此类资源包含与其他资源重叠的数据，它们可能会有不同的过期时间。比如客户快照资源包含客户信息、详细联系信息、客户信息中的最近购买订单摘要，以及待确定的报价请求。因为服务器已经以资源的形式提供了客户信息、详细联系信息、购买订单和报价，对它们的变更都会使复合资源变旧。

155

复合资源是便利性与缓存效率之间权衡的一个好例子。对客户端而言，复合资源用起来很方便，但服务器要提供最新的资源却代价很大。本例中有三部分信息，其中每个部分的变更频率和信息来源可能都不相同：

- 客户，带有名称、联系信息和其他详细资料。这个数据不太会经常改变，将其作为资源提供出来时，服务器可以设置一个大的过期间隔（例如，几天）。
- 每个客户的购买订单集合，这个数据每天晚上都会发生变化，因为它是从另一个后端系统导进来的，服务器可以将 Last-Modified 头设为午夜时间，再将过期时间设为 24 小时。
- 每个客户的待确定报价集合。这个数据可能有严格的业务要求，服务器上的过期时间要设置为 5 分钟。

现在假定这个复合资源的使用者是一个管理客户关系的员工，他所使用的客户端要关注 5 分钟内发生的新报价请求。

如此一来，服务器的选择显而易见。既然待确认的报价信息是最关键的，那服务器应该以此为主来计算 Last-Modified 和 ETag 头：

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

```
# 响应
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8
Date: Sun, 25 Oct 2009 18:08:22 GMT
Last-Modified: Sun, 25 Oct 2009 16:54:10 GMT
Cache-Control: max-age=300
ETag: "81a540e69b4c29a80586284be0d3f296"

<snapshot xmlns:atom="http://www.w3.org/2005/Atom">
  <customer>
    <id>1234</id>
    <atom:link rel="self" href="http://www.example.org/customer/1234">
    <name>...</name>
    <address>...</address>
  </customer>
  <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>
  <quotes>
    <atom:link rel="http://www.example.org/rels/quotes/recent"
          href="http://www.example.org/quotes?customerid=1234&sortby=
          date_desc&"/>
    ...
  </quotes>
</snapshot>
```

156

本例演示了如何做出权衡。为了保证该资源的用户能在 5 分钟内看到报价，服务器为整个表述选择了一个较短的过期间隔。这种缓存的使用方法不太高效，就算那些改动不频繁的数据也会在 5 分钟后变为旧数据。一个解决办法是将复合资源打散成三个资源，但这就强迫客户端发起多次请求来获取数据。换言之，服务器需要考虑利害关系并做出权衡。当资源的表述粒度较粗时，就会发生这种情况，有可能会以独立资源的方式来提供表述中所含的数据。

9.5 如何保持新鲜且温暖的缓存

缓存支持的众多挑战之一就是保持缓存的“新鲜”（最新状态）和“温暖”（非空），就算客户端没有发起请求时也是如此。以一个照片分享服务为例，在用户上传照片

后，所有针对这些照片的缓存都是空的，因此服务器不得不生成照片的表述。类似的，当引入一个新的缓存时，它是空的，随着客户端开始发起请求后，它才会慢慢有内容。当缓存处于新鲜状态时，其中包含最新的表述。一个“温暖”的缓存能避免冷启动问题。但是，预先保持缓存的新鲜与温暖不在 HTTP 的范畴之内。本节介绍的内容只是一些提示而已。

问题描述

您想知道如何确保缓存拥有最新的表述并有效运作。

解决方案

尽可能地让过期时间与更新频率保持同步。当这一点很难做到时，实现一些后台进程来监控数据库变更，定时执行无条件 GET 请求（详见 10.6 节）来刷新缓存。在定时调度这些请求时，一定要考虑到数据库复制的延迟。

如果正在使用 Squid，可以使用 HTTP 缓存频道扩展（HTTP cache channel extension）向缓存传播资源更新。

如果想清除缓存的表述，请查看所使用的缓存的文档。举例来说，Squid 提供了一个扩展从缓存中清除表述（详见 *http://wiki.squid-cache.org/SquidFaq/OperatingSquid*）。请注意，只可以清除那些受您管辖的缓存中的表述。可能还有一些下游缓存会继续持有缓存的副本。

问题讨论

157

在 HTTP 中，当客户端提交 PUT，POST 或 DELETE 请求时，要求缓存作废表述。此后，客户端再对同一资源发起 GET 或 HEAD 请求时，缓存就能从源服务器获得一个新的表述。尽管这种做法不太高效（因为缓存无法保持非空状态），但它却保证了客户端能获得最新的表述。

在现实中，您的网络里可能有多个应用程序会读写相同的数据存储，只要其中任意应用程序的写操作绕过 HTTP 缓存和服务器，那么就会导致缓存的表述变旧。考虑下面这些例子：

- 服务器上可能会有每晚执行的定时任务，更新摘要数据表以反映日间的工作。这个更新可能直接发生在数据库层面上，不会有任何 HTTP 请求。但是，您可能缓存了这个资源，一旦定时任务执行了，存储在缓存中的应答就旧了。
- 广泛分布的数据存储在一天里可能会周期性地进行复制。数据的改变不会反映在全部的缓存里。

- 大型应用程序可能会有一个或多个客户端或服务是使用自定义协议来更新数据的。只要使用这些服务就可以在不通过 HTTP 的情况下改变数据库，此时缓存并不知道资源发生了变化。

在所有这些情况里，您都需要确保所有对后端数据存储的变更最终都可以反映在缓存里。一种解决方法是实现触发器来监控数据库变更，随后运行一个定时任务通过缓存发起无条件 GET 请求（详见 10.6 节）。在有客户端发起对这些资源的请求前，它们的表述就已经在缓存里了。举例来说，您可以每天早上执行一个这样的定时任务，在工作日开始之前，前一天的更新就能在缓存里了。

另一个解决方案是利用类似 HTTP 缓存频道这样的 Cache-Control 扩展。该扩展定义了一个“频道”，缓存服务可以订阅该频道接收资源更新的通知。在编写本书时，Squid 支持该扩展。下面这个例子中，服务器提供了一个 URI 来检查资源变更：

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

# 响应
HTTP/1.1 200 OK
Cache-Control: max-age=600,channel="http://www.example.org/channels/orders"
Date: Sun, 09 Aug 2009 00:56:14 GMT
Last-Modified: Sun, 09 Aug 2009 00:56:14 GMT
Expires: Sun, 09 Aug 2009 01:56:14 GMT
Content-Type: application/xml;charset=UTF-8

...
```

158

了解该扩展的缓存服务可以通过 http://www.example.org/channels/orders 来订阅服务器发布的事件。欲了解更多内容，请访问 *http://ietfreport.isoc.org/idref/draft-nottingham-http-cache-channels*。

第 10 章

条件请求

159

HTTP 中的条件请求（conditional request）着眼于两个问题。对于 GET 请求，条件请求帮助客户端和缓存校验被缓存的表述是否仍然是最新的。对于非安全请求，比如 PUT，POST 和 DELETE，条件请求则可以提供并发控制。

不支持条件 GET 请求会降低性能，面临并发时，如果不使 POST，PUT 和 DELETE 这些非安全请求变得条件化，则可能会影响到应用的完整性。当缺少足够的并发控制时，服务器很容易发生“更新丢失”（lost updates）和“删除过期”（stale deletes）。当客户端提交请求来修改或者删除资源时，它基于的是自己所认定的资源当前状态。但是在并发环境中，资源的当前状态不是静态的。服务器端可能有某些后台途径，或者有其他的客户端可能更新或删除资源。

并发控制可以确保在并发访问的情况下对数据做出正确的处理。实现并发控制有两种方式:

悲观并发控制

在这个模型中，客户端会得到一个锁，获得资源的当前状态，并做所需的修改，然后将锁释放。在这一过程中，服务器会禁止其他客户端得到同一资源的锁。关系型数据库就是依照这一模型运作的。

乐观并发控制

在这个模型中，客户端首先得到一个令牌，并将此令牌包含于请求中尝试进行写操作，而不是先获得锁。如果令牌仍然有效，那么操作就成功，否则失败。

作为一种无状态的应用控制，HTTP 采用的是乐观并发控制。乐观并发控制的基本流程如下:

160

1. 服务器给客户端的每个资源表述都附带一个令牌。

2. 对于资源的改变，服务器会让这些令牌也随之改变。换句话说，每当资源的状态有变化时，服务器的令牌版本也会变化。

3. 在每个需要修改或者删除资源的请求中，客户端向服务器提供令牌信息。这种包含令牌的请求也叫做“条件请求“。

4. 服务器检查客户端所提供的令牌是否仍然有效。如果不再有效，那就意味着并发失败，服务器终止该请求。

客户端、服务器以及缓存都能使用相似的技术来实现条件 GET 请求。条件 GET 请求能够延长陈旧表述的生命周期。本章涉及以下实现条件请求的内容:

10.1，“如何生成 *Last-Modified* 和 *ETag* 头”
通过本节了解如何生成 Last-Modified 和 ETag 头。

10.2，“如何在服务器端实现条件 *GET* 请求”
通过本节了解如何在服务器端支持条件 GET 请求。

10.3，“如何从客户端提交条件 *GET* 和 *HEAD* 请求”
通过本节了解如何从客户端发出条件 GET 请求。

10.4，“如何在服务器端实现条件 *PUT* 请求”
通过本节了解如何在服务器端为 PUT 请求实现乐观并发控制。

10.5，“如何在服务器端实现条件 *DELETE* 请求”
通过本节了解如何在服务器端为 DELETE 请求实现乐观并发控制。

10.6，“如何从客户端发起无条件 *GET* 请求”
通过本节了解如何获取最新的表述以用于开发和排错。

10.7，“如何从客户端提交条件 *PUT* 和 *DELET* 请求”
通过本节了解如何实现客户端，将 If-Unmodified-Since 和 If-Match 头用做令牌，进行并发控制检验。

10.8，“如何使 *POST* 请求条件化”
通过本节了解服务器是如何使用链接来为 POST 请求交换并发控制令牌的。

10.9，“如何生成一次性 *URI*”
通过本节了解如何生成一个用于修改资源状态的 URI，且只能使用一次。

10.1 如何生成 Last-Modified 和 ETag 头

161

服务器通过 Last-Modified 和 ETag 响应头来驱动条件请求。客户端使用下面的请求头来发起条件请求:

- 使用 If-Modified-Since 和 If-None-Match 来校验缓存中的表述。
- 使用 If-Unmodified-Since 和 If-Match 作为并发控制的先决条件。

如果这些标头的名字让您觉得费解，请联想一下您的“意图“。对于 GET 请求，只要表述在指定时间之后被修改过和/或表述与所提供的实体标签均不匹配，客户端的意图是获取一个最新的表述。

对于并发控制，其意图是，只有当表述自某个时间以来没有被修改和/或与所提供的实体标签相匹配时，才进行操作。

这两组请求头都允许客户端随请求提供“先决条件”。

处理条件请求的效率取决于服务器能够以多快的速度校验 Last-Modified 和 ETag 头。本节的内容将讨论如何生成这些标头。

问题描述

您想知道如何生成 Last-Modified 和 ETag 头，以便为资源表述提供缓存和条件请求的支持。

解决方案

如果您可以接触到用于保存资源的数据存储，那就修改数据存储的结构，给每个资源都包含一个修改日期的时间戳和/或一个用来跟踪其版本变化的序列值。

对于关系型数据库，当数据被修改时，可以通过触发器自动地更新这些字段。

如果您无法修改数据存储的结构，或者数据存储不允许使用时间戳或者序列值，那就根据资源数据来生成一个 ETag 头的值。将这个值和时间戳一起保存在另外的数据表或数据存储中，这样服务器就不必为了计算这些值而重新加载整个表述。

如果表述并不算很大，那么可以根据表述的正文生成一个 MD5 值，赋给 ETag。另外也可以使用数据表中的某些会随资源更新而改变的字段来计算 ETag 值。

对于资源的每个表述，记得要使用不同的 ETag 值。

162

问题讨论

大多数 Web 服务器都会自动给静态内容添加 ETag 和 Last-Modified 头。而对于应用程序资源，就需要自己编程来添加了。

在这些标头中，Last-Modified 只能精确到秒，因而只是一个弱验证。实体标签则是强验证，因为它的值可以在每次服务器修改表述时随之改变。实体标签就像是对象的哈希码。可以使用实体标签来比较资源的各个表述。

要支持条件请求，并不需要同时使用 Last-Modified 和 ETag 头，可以选择其中的一个，或者一起使用，并贯彻下去就可以了。

如果是设计一个新的 Web 服务，那就在数据存储中包含一个时间戳和一个版本计数器。大多数关系型数据库都可以通过一些触发器自动在数据发生变化时更新或增加上述字段的值。这通常是最有效的手段，因为这样服务器不必为了检查这些值而加载整个资源的数据。

某些 Web 框架会在程序代码生成表述之后自动生成 ETag 头。其原理是计算整个表述的哈希值。如果表述的数据从数据库中加载需要花一些时间，那么该功能的性能就会受到影响。

如果资源的数据遍布于多个表中，那就可能需要从这些表里选择最新的 Last-Modified 时间戳，并使用对应版本号的哈希值。

对于非关系型数据库，具体采用的方法就因实现而异了。

如果使用版本号来生成 ETag 的值，那就得确保这个值是和特定表述相关的。例如，如果资源的两个表述具有不同的媒体类型，那就结合使用媒体类型值和版本号，让 ETag 值是和特定表述相关的。

10.2 如何在服务器端实现条件 GET 请求

如果表述自最近一次发布以来未曾改变过，那么条件 GET 请求可以让服务器有机会略过响应正文。在条件请求中，客户端根据上一次请求中的 Last-Modified 和 ETag 头来发送 If-Modified 和 If-Match 头。条件请求并不会减少来自客户端的请求数量，但它们可以减少服务器向客户端发送最新表述的次数。

问题描述

163

您想知道如何实现条件 GET 请求。

解决方案

让服务器记录最后一次修改的日期时间和/或实体标签。发布表述时，将最后一次修改时间作为 Last-Modified 头的值，将实体标签作为 ETag 的值。

响应 GET 和 HEAD 请求时，如果客户端发送了 If-None-Match 头，那就将它与服务器表述的 ETag 值做比较。如果客户端发送了 If-Unmodified-Since 头，那就将它与服务器表述的最后修改时间做比较。

如果上述两个比较的结果都为假，或者客户端并未发送这两个标头，那就向客户端返回表述的最新副本，并包含新的 ETag 和/或 Last-Modified 头。否则就返回 HTTP 状态码 304（Not Modified）且不含消息正文。

问题讨论

使用条件 GET 请求来延长一个缓存副本寿命的过程，被称为“校验“（validation）。要支持这一过程，服务器必须要向客户端返回过期标头（expiration headers）和条件标头（conditional headers），通过返回状态码 304（Not Modified）来实现校验，延长被缓存响应的寿命。

下面的例子描绘了客户端如何利用缓存代理服务器来保存响应，并让缓存自动处理校验。如果您的客户端在本地保存表述的副本，并想要知道本地的版本是否还最新，详见 10.3 节。

下面的表述中包含 ETag 和 Last-Modified 头，还有过期缓存头（expiry caching headers）：

```
# 响应
HTTP/1.1 200 OK
Date: Sun, 09 Aug 2009 00:56:14 GMT
Last-Modified: Sun, 09 Aug 2009 00:56:14 GMT
Expires: Sun, 09 Aug 2009 01:56:14 GMT
Cache-Control: max-age=3600,must-revalidate
E-Tag: "3f4a74db207d0447d46710a64971e777"
Content-Type: application/xml; charset=UTF-8

...
```

在本例中，ETag 头的值是一个实体标签。如果发起两个 GET 请求，并且收到的 ETag 头的值不同，那就说明表述已经发生了改变。

164

Etag，If-Match以及 If-None-Match头的值都要写在引号内。

本例中服务器的意图是让表述在缓存中保存一小时，过期时，通过条件 GET 请求来校验表述。以下便是校验过程：

```
# 第一次请求
GET /person/joe HTTP/1.1
Host: www.example.org

# 第一次响应
HTTP/1.1 200 OK ❶
Date: Sun, 09 Aug 2009 00:44:14 GMT
Last-Modified: Sun, 09 Aug 2009 00:40:14 GMT
Expires: Sun, 09 Aug 2009 01:44:14 GMT
Cache-Control: max-age=3600,must-revalidate ❷

...

# 10 分钟后的第 2 次请求
GET /person/joe HTTP/1.1
Host: www.example.org
# 第 2 次响应——由缓存返回
HTTP/1.1 200 OK
Date: Sun, 09 Aug 2009 00:54:14 GMT
Last-Modified: Sun, 09 Aug 2009 00:40:14 GMT
Expires: Sun, 09 Aug 2009 01:44:14 GMT
Cache-Control: max-age=3600,must-revalidate
Age: 600 ❸

...
```

❶ 服务器生成的响应

❷ 响应可以被缓存 3600 秒，超时后就必须重新校验

❸ 一个产生于 10 分钟前的响应，由缓存提供

客户端在 1 小时之后（缓存过期）发出的请求会导致缓存的响应被重新校验。

```
# 1 小时后的第三次请求
GET /person/joe HTTP/1.1 ❶
Host: www.example.org

# 缓存发给源服务器的请求
GET /person/joe HTTP/1.1 ❷
Host: www.example.org
```

165

```
If-Modified-Since: Sun, 09 Aug 2009 00:40:14 GMT
If-None-Match: "3f4a74db207d0447d46710a64971e777"

# 服务器生成的响应
HTTP/1.1 304 Not Modified ❸
Date: Sun, 09 Aug 2009 01:54:14 GMT
Last-Modified: Sun, 09 Aug 2009 00:56:14 GMT
Expires: Sun, 09 Aug 2009 02:54:14 GMT
Cache-Control: max-age=3600,must-revalidate
E-Tag: "3f4a74db207d0447d46710a64971e777"
Content-Type: application/xml; charset=UTF-8

# 缓存返回的响应
HTTP/1.1 200 OK ❹
Date: Sun, 09 Aug 2009 00:54:14 GMT
Last-Modified: Sun, 09 Aug 2009 00:40:14 GMT
Expires: Sun, 09 Aug 2009 01:44:14 GMT
Cache-Control: max-age=3600,must-revalidate

...
```

❶ 发送于缓存到期之后的请求

❷ 缓存向源服务器发出请求，校验缓存中的响应

❸ 源服务器的响应，提示响应并未发生改变

❹ 缓存发送给客户端的响应，包含原缓存中的副本

服务器并未接收到第二个请求，因为当时响应还没有过期，缓存中的副本仍然有效。第三个请求则到达了服务器，目的是为了做校验。服务器所返回的第三个响应宣告表述并没有发生改变，同时还将有效期延长了一小时。

在本例中，服务器的职责是比较 If-Modified-Since 和/或 If-None-Match 头的值和当前值，返回状态码 200（OK）以及资源表述，或者返回状态码 304（Not Modified）。

10.3 如何从客户端提交条件 GET 和 HEAD 请求

如果客户端在本地存储了表述，那么可以使用条件 GET 或 HEAD 请求来确定本地所保存的表述是否仍然还是最新的。

166

问题描述

您想知道如何在客户端实现条件 GET 和 HEAD 请求。

解决方案

当服务器返回 Last-Modified 和/或 ETag 头时，将它们同表述数据一起存储。

再次对同一资源发出 GET 和 HEAD 请求时，包含下面的标头，让请求变得“条件化“：

- If-Modified-Since 头，其值就是所保存的 Last-Modified 头的值。
- If-None-Match 头，其值是所保存的 ETag 头的值。

问题讨论

支持条件请求会涉及对 Last-Modified 和 ETag 头的存储，以及在随后的请求中回放它们。考虑下面的请求：

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

# 响应
HTTP/1.1 200 OK
Date: Sun, 09 Aug 2009 02:55:46 GMT
Last-Modified: Sun, 09 Aug 2009 00:56:14 GMT
Expires: Sun, 09 Aug 2009 03:55:46 GMT
Cache-Control: max-age=3600,must-revalidate
E-Tag: "3f4a74db207d0447d46710a64971e777"
Content-Type: application/xml; charset=UTF-8

<person xmlns="org:example:people" xmlns:atom="http://www.w3.org/2005/Atom">
  <name>John Doe</name>
  <address>
    <street>1 Main Street</street>
    <city>Seattle</city>
    <atom:link rel="self" href="http://www.example.org/person/john/address"/>
    <state>WA</state>
  </address>
  <atom:link rel="self" href="http://www.example.org/person/john"/>
</person>
```

这并不是一个条件请求，因为请求中不包含 If-Modified-Since 和 If-None-Match。

如果要保存这个表述，以便客户端今后使用，那么就要保存人名、地址以及资源的 URI。同时也要包含每个表述的 Last-Modified 和/或 ETag 值，以便可以发起条件

请求。

之后，可以检查服务器是否改变了表述，只需在请求中包含 If-Modified-Since 和/或 If-None-Match 头即可。

167

```
# 请求
GET /person/joe HTTP/1.1
If-Modified-Since: Sun, 09 Aug 2009 00:56:14 GMT
If-None-Match: "3f4a74db207d0447d46710a64971e777"

# 响应
HTTP/1.1 304 Not Modified
Date: Sun, 09 Aug 2009 03:10:03 GMT
Last-Modified: Sun, 09 Aug 2009 00:56:14 GMT
Expires: Sun, 09 Aug 2009 04:10:03 GMT
Cache-Control: max-age=3600,must-revalidate
E-Tag: "3f4a74db207d0447d46710a64971e777"
```

服务器的响应表明客户端的表述副本依然是最新的，同时也将副本的过期时间延长了一小时。

除非客户端本地保存有表述的副本，否则不要发送条件请求。

10.4 如何在服务器端实现条件 PUT 请求

不严谨的实现，或者缺少并发控制的 PUT 请求可能导致“更新丢失“。本节的内容将展示如何在服务器端用 Last-Modified 和/或 ETag 头来实现条件 PUT 请求，支持乐观并发控制。

问题描述

您想知道如何为 PUT 请求实现并发控制。

解决方案

如果资源还不存在，并且服务器支持通过 PUT 来创建资源，那就在由客户端指定的 URI 处新建资源。如果服务器不支持创建资源，就返回状态码 404（Not Found）给客户端。

如果资源已经存在，那就采取下面的步骤：

- 如果客户端并未包含 If-Unmodified-Since 和/或 If-Match 头，那就返回状态码 403（Forbidden）。在响应正文中解释为何返回该状态码。
- 如果客户端所提供的 If-Unmodified-Since 或 If-Match 头与服务器表述的实际的修改时间和 ETag 值不匹配，返回错误码 412（Precondition Failed）。
- 168 如果客户端提交了一个条件 PUT 请求，并且所提供的条件也符合实际值，那就更新资源，并返回 200（OK）或者 204（No Content）。
  此外还可以选择性地包含更新过的 Last-Modified 和/或 ETag 头，这要求响应中还同时包含 Content-Location 头，其值是更新后的资源的 URI。

关于服务器需要做的各种检查，详见图 10-1。

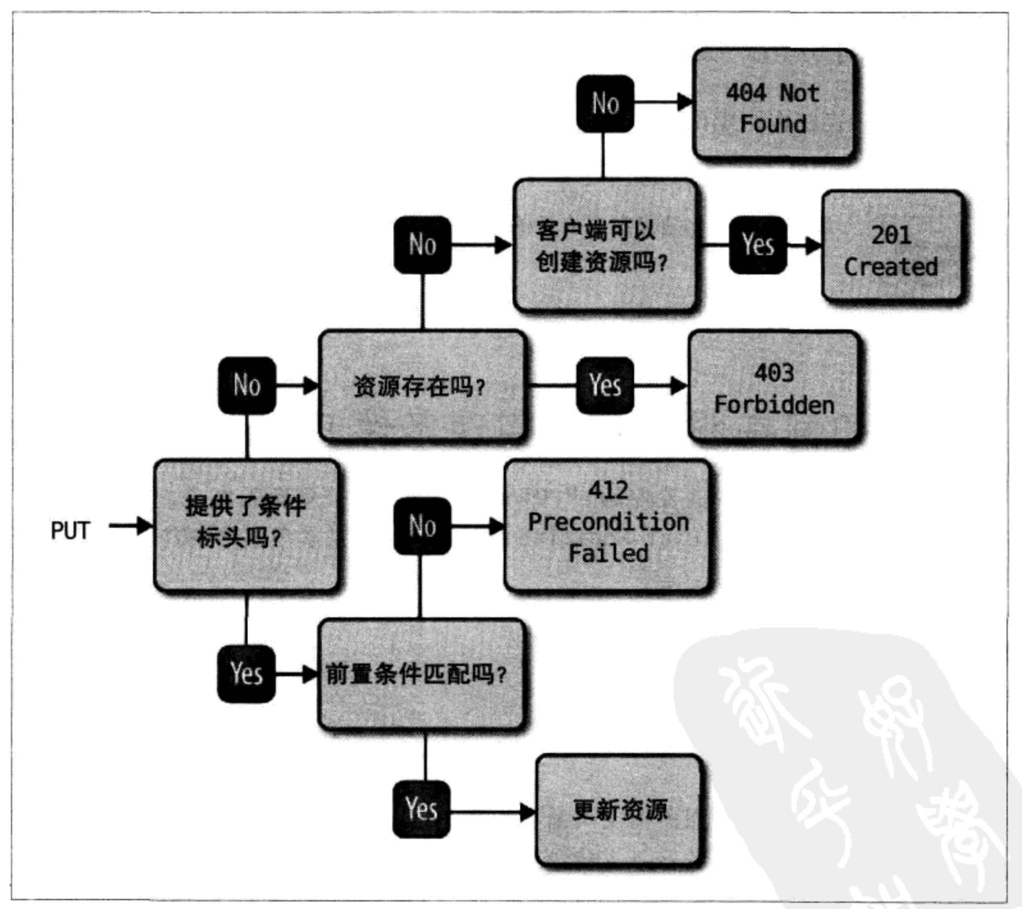


图 10-1：实现条件 PUT 请求

每当服务器向客户端返回表述时，都要记得包含 Last-Modified 和/或 ETag 头。

问题讨论

实现条件 PUT 请求的步骤和实现 GET 请求（详见 10.2 节）是相似的。关键的不同之处在于这里使用 If-Unmodified-Since 和/或 If-Match 头，而不是 If-Modified-Since 和/或 If-None-Match 头。

下面是客户端所提交的一个条件 PUT 请求：

```
# 当且仅当包含的条件标签匹配时处理该请求
PUT /reviews/notes_from_underground HTTP/1.1
Host: www.example.org
If-Unmodified-Since: Sun, 09 Aug 2009 00:56:14 GMT
If-Match: "3f4a74db207d0447d46710a64971e777"
```

169

如果服务器期待一些条件标头，但是并没有找到，那么它必须返回响应码 403（Forbidden）。如果在请求中发现了条件标头，那必须将它们同 Last-Modified 和/或 ETag 的当前值进行比较。如果比较结果相同，服务器就可以处理更新，并返回 200（OK）。否则，必须返回响应码 412（Precondition Failed）。关于这个响应码的消息示例，详见 10.7 节。

如果同时发送 Last-Modified 和 ETag，那么服务器只在 If-Unmodifed-Since 和 If-Match 头同时匹配当前值时才处理 PUT 请求。如果其中一个不匹配，就返回 412（Precondition Failed）。

下面的例子展示了为什么 PUT 请求的条件化如此重要。考虑一个 Wiki 形式的内容管理服务，其中客户端可以修改和/或删除内容。假设有两个客户端 A 和 B，按照下面的顺序修改一个资源。客户端 A 获得资源的一个表述，并开始在本地对资源进行编辑：

```
# 来自客户端 A 的请求
GET /reviews/notes_from_underground HTTP/1.1
Host: www.example.org

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

客户端 B 也获得了这个资源的表述，开始在本地做编辑修改：

```
# 来自客户端 B 的请求
GET /reviews/notes_from_underground HTTP/1.1
Host: www.example.org
```

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

客户端 B 完成了编辑并将修改通过 PUT 请求提交到服务器:

```
# 来自客户端 B 的请求
PUT /reviews/notes_from_underground HTTP/1.1 ❶
Host: www.example.org

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

...
```

170

❶ 第一次更新

几秒钟后，客户端 A 也完成了编辑，并将修改提交为另一个 PUT 请求:

```
# 来自客户端 A 的请求
PUT /reviews/notes_from_underground HTTP/1.1 ❶
Host: www.example.org

# 响应
HTTP/1.1 204 OK
Content-Type: application/xml;charset-UTF-8

...
```

❶ 第二次的更新覆盖了第一次的更新

这次操作的结果是客户端 A 覆盖了客户端 B 所做的修改。B 提交的更新完全丢失了！而 A 和 B 都无从知晓这一情况。从两个客户端的角度看，各自的 PUT 操作都成功完成。只是 B 会在今后发现它所做的更新丢失了。除非服务器显式地记录每个修改操作，否则也无法检查到这个丢失的更新。

将 PUT 请求条件化就可以防止像这样丢失更新。下面是来自客户端 B 的请求:

```
# 来自客户端 B 的请求
PUT /reviews/notes_from_underground HTTP/1.1 ❶
Host: www.example.org
If-Unmodified-Since: Sun, 09 Aug 2009 00:56:14 GMT
If-Match: "3f4a74db207d0447d46710a64971e777"
```

```
# 响应
HTTP/1.1 204 No Content
Content-Location: http://www.example.org/reviews/notes_from_underground
Content-Type: application/xml;charset-UTF-8
Last-Modified: Sun, 09 Aug 2009 01:10:14 GMT
If-Match: "5dcb920acfd4f3943dbc1672756d7f43"
```

❶ 第一次条件更新成功

响应中包含 Content-Location 头以及更新过的 Last-Modified 和 ETag 值，可供客户端在今后的请求中使用。

如果客户端 A 的请求也是条件化的，并且服务器正确检测到了冲突，那么 A 要做的更新操作就会失败：

```
# 来自客户端 A 的请求
PUT /reviews/notes_from_underground HTTP/1.1 ❶
Host: www.example.org
If-Unmodified-Since: Sun, 09 Aug 2009 00:56:14 GMT
If-Match: "3f4a74db207d0447d46710a64971e777"

# 响应
HTTP/1.1 412 Precondition Failed
Content-Type: application/xml;charset-UTF-8

<error>
  <message>The review you are trying to update has changed.</message>
  <description>You are trying to  update a resource based on stale information.
  Get a new copy of this review, resolve any differences, and retry.</description>
</error>
```

171

❶ 第二次条件更新失败

10.5 如何在服务器端实现条件 DELETE 请求

条件 DELETE 请求有助于防止客户端基于过时的信息删除资源。例如，客户端正要删除一个“未激活”的用户资源，而该用户正好被另一个客户端设置为“激活”了，这是大家都不希望看到的情况。

问题描述

您想知道如何为 DELETE 请求实现并发控制。

解决方案

如果客户端提交了非条件化的 DELETE 请求，那就返回错误码 403（Forbidden）。如果所提供的条件不匹配，就返回错误码 412（Precondition Failed）。任何一种情况下，都要在表述的正文中给出操作失败的原因。

如果客户端提交了条件 DELETE 请求，并且条件也满足，那就进行删除资源的操作。

关于服务器需要做的检查工作，详见图 10-2。

DELETE → 是否提供了条件标头？
No → 403 Forbidden
Yes → 前置条件是否匹配？
No → 412 Precondition Failed
Yes → 删除资源

图 10-2：实现条件 DELETE 请求

问题讨论

客户端想要删除一个资源时，大多会假设删除这个资源后不会导致任何问题。与此同时，另一个客户端可能正在更新同一个资源，这使第一个客户端的假设不再成立。使用条件 DELETE 可以避免这样的情形发生。当服务器将 DELETE 条件化后，它就可以检查并确认客户端所要删除的资源是否处于最新状态。这是通过比较客户端所提供的 If-Unmodified-Since 和 If-Match 头与服务器的当前值来实现的。

172

```
# 当且仅当包含的条件标签匹配时处理该请求
DELETE /reviews/notes_from_underground HTTP/1.1
Host: www.example.org
If-Unmodified-Since: Sun, 09 Aug 2009 00:56:14 GMT
If-Match: "3f4a74db207d0447d46710a64971e777"

# 响应
HTTP/1.1 204 No Content
```

10.6 如何从客户端发起无条件 GET 请求

HTTP 1.1 允许客户端修改过期缓存并请求最新的表述。通过本节的内容，您可以了解如何在收到 412（Precondition Failed）或者在成功的 PUT 或 PATCH 后，获取最新的资源表述。

问题描述

您想知道如何实现客户端，以获取服务器端的最新可用表述。

解决方案

在 GET 请求中包含 Cache-Control: no-cache 以及 Pragma: no-cache 头。

问题讨论

假设客户端通过条件 PUT 请求来更新资源。但是由于所提供的条件不匹配，服务器返回了 412（Precondition Failed）。

173

```
# 当且仅当包含的条件标签匹配时处理该请求
PUT /reviews/notes_from_underground HTTP/1.1
Host: www.example.org
If-Unmodified-Since: Sun, 09 Aug 2009 00:56:14 GMT
If-Match: "3f4a74db207d0447d46710a64971e777"
...

# 响应
HTTP/1.1 412 Precondition Failed
Content-Length:0
```

客户端可以发起一个无条件的 GET 请求，获取资源的最新表述。

```
# 请求
GET /status HTTP/1.1
Cache-Control: no-cache ❶
Pragma: no-cache

# 响应
HTTP/1.1 200 OK
Date: Sun, 09 Aug 2009 05:20:10 GMT
Last-Modified: Sun, 09 Aug 2009 05:20:10 GMT ❷
ETag: "a3d3005f4a1632c88e8889af985e6294"
Expires: Sun, 09 Aug 2009 15:56:14 GMT
Cache-Control: max-age=36000,public
Content-Type: application/xml; charset=UTF-8

...
```

❶ Cache-Control: no-cache 和 Pragma: no-cache 头使得客户端的请求变得无条件化。

❷ 服务器返回了最新的表述，附带可用的 Last-Modified 和 ETag。

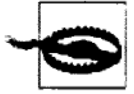
请求中的 no-cache 指令会通知中间所有的缓存都不要提供表述，直接将请求转给源服务器。

请注意，也有一些缓存会忽略 no-cache 指令。这种情况下，缓存可能会返回 Warning 头。

```
# 请求
GET /status HTTP/1.1
Cache-Control: no-cache
Pragma: no-cache

# 响应
Date: Sun, 09 Aug 2009 00:56:14 GMT
Last-Modified: Sun, 09 Aug 2009 00:56:14 GMT
Expires: Sun, 09 Aug 2009 10:56:14 GMT
Cache-Control: max-age=36000,public
Content-Type: application/xml; charset=UTF-8
Age: 1021
Warning: 110
```

174 Warning 头的值是一个整数代码，本例中的值表示这个响应是已经过期的。关于这个标头的详细信息请参见 HTTP 1.1 规范。



不到万不得已，不要使用非条件 GET 请求，它们会影响服务器的性能并导致延时。

10.7 如何从客户端提交条件 PUT 和 DELETE 请求

问题描述

您想知道如何在客户端支持对 PUT 和 DELETE 请求的并发控制。

解决方案

当客户端使用 PUT 请求来创建新资源，或者服务器在之前的 GET 或 PUT 请求中没有返回 If-Modified-Since 和/或 ETag 头时，就和往常一样发出 PUT 请求。

如果客户端有来自先前请求的 If-Modified-Since 和/或 ETag 头，那么在发出 PUT 和 DELETE 请求的同时，包含下面的标头，使请求“条件化”：

- If-Unmodified-Since 头，其值和 Last-Modified 头的值相同，它提示服务器，只有当资源在指定时间以来没有变化时才处理请求。
- If-Match 头，其值和 ETag 头的值相同，它提示服务器，只有当所提供的值匹配当前的 ETag 值时才处理请求。

如果服务器返回状态码 412（Precondition Failed），那就提交一个无条件 GET 请求（详见 10.6 节）获得最新的 Last-Modified 和 ETag 头，根据最新的表述，核对并确认对资源的更新或删除是否仍然有效，然后用这些标头重复 PUT 或 DELETE 请求。

问题讨论

这里有一个客户端发起条件 PUT 请求的范例：

```
# 请求
PUT /reviews/notes_from_underground HTTP/1.1
Host: www.example.org
If-Unmodified-Since: Sun, 09 Aug 2009 00:56:14 GMT
If-Match: "3f4a74db207d0447d46710a64971e777"
...

# 响应
HTTP/1.1 200 OK
Date: Sun, 16 Aug 2009 01:00:23 GMT
Content-Location: http://www.example.org/reviews/notes_from_underground
Last-Modified: Sun, 16 Aug 2009 01:00:23 GMT
E-Tag: "5bbae963eb30e03cf1fd218a9dc92a5b"
Content-Type: application/xml; charset=UTF-8

...
```

175

在这次请求中，If-Unmodified-Since 和客户端从上一次请求中获得的 Last-Modified 头的值相同（详见 10.3 节）。类似的，If-Match 头和 ETag 头的值也相同。如果客户端本地的表述副本里还没有这些标头，或者服务器返回了响应代码 412（Precondition Failed），那就提交一个新的无条件 GET 请求来获得表述，就

和在 10.6 节里所讲述的那样。

```
# 无条件 GET 请求
GET /reviews/notes_from_underground HTTP/1.1
Host: www.example.org
Cache-Control: no-cache
Pragma: no-cache

# 响应
HTTP/1.1 200 OK
Content-Type: application/xml;charset-UTF-8
Date: Sun, 16 Aug 2009 01:00:23 GMT
Last-Modified: Sun, 09 Aug 2009 00:55:46 GMT
ETag: "3f4a74db207d0447d46710a64971e777"

...
```

只通过 HEAD 请求来获得这些标头是不够的。对于客户端，这些标头对应着资源的当前状态。通过 HEAD 请求只能获得这些标头本身，这还不足以让客户端知道资源的当前状态，还要同时获得正文才行。

对于 DELETE 请求，客户端也必须遵循同样的过程。412（Precondition Failed）响应码意味着客户端是根据已经过时的信息在更新或删除资源。

```
# 条件请求
PUT /reviews/notes_from_underground HTTP/1.1
Host: www.example.org
If-Unmodified-Since: Sun, 09 Aug 2009 00:56:14 GMT
If-Match: "3f4a74db207d0447d46710a64971e777"

# 响应
HTTP/1.1 412 Precondition Failed
Content-Type: application/xml;charset=UTF-8

<error>
  <message>The review you are trying to update has changed.</message>
  <description>You are trying to  update a resource based on stale information.
  Get a new copy of this review, resolve any differences, and retry.</description>
</error>
```

176

10.8 如何使 POST 请求条件化

和 PUT 或 DELETE 不同，对资源的 POST 请求并不一定会导致请求 URI 的资源有任何变化。服务器可能会创建一个新资源（返回响应码 201）或者用一个不同的 URI 标识结果（返回响应码 303）。对于这些情况，客户端本地并不会存储表述和条件标

头。本节的内容将展示如何使用链接来使这些 POST 请求变得条件化且“不可重复”（即“一次性”）。

问题描述

您想实现 POST 请求，让服务器可以检查并防止客户端的重复提交。

解决方案

让客户端使用一次性的 URI，服务器为每个 POST 请求提供了一个链接。根据 10.9 节的内容来生成这些一次性的 URI。URI 中含有一个由服务器生成的令牌，只能用于一次 POST 请求。此外，还需要将所有用过的令牌保存在服务器的事务日志里。

客户端提交一个 POST 请求时，检查其中的令牌是否已经存在于事务记录里。如果存在，那就返回响应码 403（Forbidden），在正文中给出解释。如果不存在，那就处理请求并根据结果返回 201（Created）或者 303（See Other）。之后同样也在事务日志中保存令牌。

问题讨论

考虑一个银行转账的应用，其中服务器需要将指定数目的款项从一个账户转移到另一个账户。服务器可以使用一个控制器资源来实现转账：

```
# 请求
POST /transfers ❶
Host: example.org
Content-Type: application/xml;charset=UTF-8

<transfer>
  <source>urn:example:org:account:1</source>
  <target>urn:example:org:account:2</target>
  <currency>USD</currency>
  <amount>100.00</amount>
  <note>Testing transfer</note>
</transfer>

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

<transfer xmlns:atom="http://www.w3.org/2005/Atom">
  <source>urn:example:org:account:1</source>
  <target>urn:example:org:account:2</target>
  <atom:link href="http://www.example.org/transactions/1" rel="self"/>
```

177

```
  <currency>USD</currency>
  <amount>100.00</amount>
  <note>Testing transfer</note>
</transfer>
```

❶ 客户端根据它所认为的银行账户资源的当前状态提交 POST 请求

❷ 服务器无法验证客户端的请求是否是基于账户资源的当前状态

在本例中，由 `http://www.example.org/transfers` 这个 URI 所标识的资源是一个控制器资源。POST 请求的结果是修改两个账户资源，以及创建一个新资源。要让这个请求变得条件化，服务器必须使用一个令牌，其值是由两个账户资源的当前状态得来的。服务器可以根据 10.9 节的内容来生成下面这样的 URI:

```
http://www.example.org/transfers;t=e6e3c89d4dfe7f3a818734a6237ccfc5
```

和 `http://www.example.org/transfers` 不同，这个 URI 包含了一个令牌，它源自需要被修改的两个资源。客户端可以使用这个 URI 来发起转账请求。服务器会先校验 URI 中的令牌是否仍然符合账户资源的当前状态，然后再处理请求并进行转账操作。

178

```
# 请求
POST /transfer;t=e6e3c89d4dfe7f3a818734a6237ccfc5 HTTP/1.1 ❶
Host: example.org
Content-Type: application/xml;charset=UTF-8

<transfer>
  <source>urn:example:org:account:1</source>
  <target>urn:example:org:account:2</target>
  <currency>USD</currency>
  <amount>100.00</amount>
  <note>Testing transfer</note>
</transfer>

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

<transfer xmlns:atom="http://www.w3.org/2005/Atom">
  <source>urn:example:org:account:1</source>
  <target>urn:example:org:account:2</target>
  <atom:link href="http://www.example.org/transactions/1" rel="self"/>
  <currency>USD</currency>
  <amount>100.00</amount>
  <note>Testing transfer</note>
</transfer>
```

❶ 客户端提交条件 POST 请求，包含一个基于账户资源当前状态所生成的 URI

❷ 服务器可以校验客户端的请求是否确实基于账户资源的当前状态

通过检查事务日志中是否存在该令牌，服务器可以找出重复的请求。

```
# 请求
POST /transfer;t=e6e3c89d4dfe7f3a818734a6237ccfc5 HTTP/1.1 ❶
Host: example.org
Content-Type: application/xml;charset=UTF-8

<transfer>
  <source>urn:example:org:account:1</source>
  <target>urn:example:org:account:2</target>
  <currency>USD</currency>
  <amount>100.00</amount>
  <note>Testing transfer</note>
</transfer>

# 响应
HTTP/1.1 403 Forbidden ❷
Content-Type: application/xml;charset=UTF-8
Date: Sat, 17 Oct 2009 20:16:18 GMT

<error xmlns:atom="http://www.w3.org/2005/Atom">
  <message xml:lang="en">Transfer already created.</message>
</error>
```

❶ 客户端提交条件 POST 请求

❷ 服务器检查出一个冲突

这一次，服务器检查了事务日志，发现这个令牌已经被使用过了。于是它返回 403 (Forbidden) 而不是重复创建资源。这一技术有时也被称为“一次性 *POST*”。

10.9 如何生成一次性 URI

179

本节讨论如何生成可用于条件 POST 请求的 URI。

问题描述

您想生成客户端可用来发起条件 POST 请求的 URI。

解决方案

如果 URI 是用于创建新资源的，那就基于一个序列或者一个时间戳和随机数的合并

字符串来生成令牌。如果 URI 是用于修改一个或多个资源，那么就在令牌里再包含那些资源的实体标签和标识符。将令牌编码到 URI 中。

问题讨论

继续考虑 10.8 节中的银行转账例子。要请求转账，服务器需要提供一个 URI，该 URI 由账户的当前状态生成。下面是一个可以生成这类 URI 的资源：

```
# 请求
GET /transfer-token?from=urn:example:org:account:1&
  to=urn:example:org:account:2 HTTP/1.1 ❶
Host: www.example.org

# 响应
HTTP/1.1 200 OK
Cache-Control: no-cache
Link: <http://www.example.org/transfers;t=e6e3c89d4dfe7f3a818734a6237ccfc5>;
    rel="http://www.example.org/rel/transfer" ❷

<accounts> ❸
  <account>
    <id>urn:example:org:account:1</id>
    <balance>200.00</balance>
  </account>
  <account>
    <id>urn:example:org:account:2</id>
    <balance>100.00</balance>
  </account>
</accounts
```

❶ 请求获得一个 URI，以发起条件请求

❷ 条件 URI

❸ 资源的当前状态

180 请求中的资源好比是一个令牌制造机。客户端提供转账操作所涉及的账户，服务器返回一个用来发起转账的 URI。此外还包含了账户的当前状态。

客户端必须根据响应中所返回的状态来使用该链接，链接仅对账户资源的当前状态有效。

有多种途径可以生成这样的 URI。下面这个 URI 中包含的令牌值是一个 MD5 哈希值，它由一个随机数、两个账户的当前状态值及其实体标签一起组合而成，随后进行 MD5:

```
http://www.example.org/transfers;t=e6e3c89d4dfe7f3a818734a6237ccfc5
```

为了防止 URI 被篡改，还可以考虑在 URI 中包含一个数字签名。服务器不需要为今后的验证保存这个令牌，因为它可以由资源的当前实体标签重新计算得到。如果哈希值不匹配，服务器就可以认为客户端所使用的 URI 已经不再有效。这样的一次性令牌也被称为 *nonce*，nonce 的意思是"number used once"。

也可以使用一次性 URI 来检测重放攻击（replay attack）。重放攻击通过恶意实体窃听（malicious entity eavesdropping）来捕获请求和响应，并重发请求来伪装成原客户端。

如果将一次性 URI 用于这一目的，要确保所用的令牌都是随机生成的，这样恶意实体便无法猜测到未来的令牌值。

如果 URI 的目的是通过 POST 创建新的资源，那就根据当前的日期时间和随机数来生成令牌。下面的一次性 URI 用于创建一个新的地址资源。URI 中的令牌是一个随机数和当前时间的 MD5 哈希结果。

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

# 响应
HTTP/1.1 204 No Content
Cache-Control: no-cache
Link: <http://www.example.org/user/smith;t=360e22a55267f0a525b1d49ddc9eed71>;
   rel="http://www.example.org/rel/transfer" ❷
```

❶ 获得一个 URI 用来发起条件请求，创建资源

❷ 条件 URI

在本例中，并不需要提供任何输入就能生成 URI。要验证这个 URI 有没有被使用过，服务器需要有事务日志。当客户端用这个 URI 提交 POST 请求创建新资源时，首先检查事务日志中是否已经存在这个令牌。如果不存在，那就处理请求，并将令牌保存在事务日志中。

181

如果是 Web 应用程序，那也可以将令牌存在 HTML 表单的隐藏域中。这是一种常用的检测重复表单提交的模式。

第 11 章 其他内容

183

本章所着眼的这些问题通常被视为极具挑战性，或者是超越了 REST 统一接口约束的领域。本章的内容涉及到创建副本、建立快照（snapshots）、移动及合并、批量处理请求，以及对事务的支持。尽管它们不会出现在每个 RESTful Web 服务中，本章仍将向您展示怎样将 HTTP 的统一接口与资源和链接结合起来，解决上述问题。

本章所用到的关键原则是，只考虑特定的应用场合，如无必要，不把问题或者解决方案一般化。例如，假设某个用例涉及到改变一张专辑的所属类别。在服务器端的实现中，这个改动可能需要修改专辑的 URI。您可能想把这个问题抽象为将某资源从一个位置“移动”到另一个位置。但是 HTTP 中并没有 MOVE 这样的方法，您也许就会认为这个问题已经超越了 HTTP 的范畴。然而，一个像“改变所属类别”这样的问题，可以很容易地被解决（详见 11.3 节），并不需要扩展 HTTP。对于其他问题也同样如此，比如批处理、复制乃至事务。以下是本章所要讨论的内容:

11.1，“如何复制资源”
　通过本节了解如何创建一个资源的副本。

11.2，“如何合并资源”
　通过本节了解如何合并两个或更多的资源。

11.3，“如何移动资源”
　通过本节了解如何移动一个资源。

11.4，“何时使用 WebDAV 方法”
　通过本节了解什么时候可以使用 WebDAV 扩展方法。

11.5，“如何支持跨服务器的操作（Operations Across Servers）”
　通过本节了解如何支持跨越服务器边界的操作。

184
11.6，“如何获取资源的快照”
　实现一个简单的资源版本管理机制，以便客户端可以遍历对某特定资源所做的

修改。

11.7，“如何撤销资源更新”

通过本节了解如何撤销对一个资源所做的修改。

11.8，“如何为部分更新提炼资源”

通过本节了解如何提炼资源并调整其粒度，以便可以针对部分更新（partial updates）使用 PUT 方法。

11.9，“如何使用 PATCH 方法”

通过本节了解如何使用 PATCH 方法来做部分更新。

11.10，“如何批量处理相似的资源”

通过本节了解如何批量地创建、更新或者删除相似的资源。

11.11，“如何触发批量操作”

通过本节了解如何使用一个特定于应用的资源来触发批量操作。

11.12，“何时使用 POST 来合并多个请求”

通过本节了解为什么通过单个 HTTP POST 请求来穿隧多个 HTTP 请求做批处理是不值得推荐的做法。

11.13，“如何支持批量请求”

通过本节了解如何使用 POST 来进行批量处理。

11.14，“如何支持事务”

通过本节了解在设计 RESTful Web 服务时如何处理事务。

11.1 如何复制资源

本节讲述了如何在无须泄露服务器实现细节的情况下创建资源的副本。

问题描述

您想知道如何创建一个已存在资源的副本。

解决方案

设计一个控制器资源，它可以创建副本。客户端向该控制器资源发起 POST 请求复制资源。为了让这个 POST 的操作能视条件而定，使用 10.9 节中的内容，为客户端提供一个一次性的 URI。

在控制器创建了副本之后，返回响应码 201 (Created) 以及一个 Location 头，其中包含该副本的 URI。

185

问题讨论

设想一个管理相册的 Web 服务。客户端想要复制一本相册，并对之做一些修改。为了支持这个功能，服务器可以设计一个控制器资源来生成副本，并在相册的表述里包含指向此控制器资源的链接。以下是一个获取相册资源表述的请求：

```
# 请求
GET /albums/2009/10/1011 HTTP/1.1
Host: www.example.org

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

<album xmlns:atom="http://www.w3.org/2005/Atom">
  <id>urn:example:album:1011</id>
  <atom:link rel="self" href="http://www.example.org/albums/2009/10/1011"/>
  <atom:link rel="http://www.example.org/rels/duplicate"
             href="http://www.example.org/albums/2009/10/1011/duplicate;
                  t=a5d0e32ddff373df1b3351e53fc6ffb1"/> ❶
  ...
</album>
```

❶ 含有用于复制资源的 URI 的链接

控制器资源的 URI 里包含了一个令牌，这样就变成条件请求了。服务器可以使用该令牌来确保重复的 POST 不会生成额外的相册副本。

假设客户端知道 http://www.exam ple.org/rels/duplicate 这个链接关系类型的语义，它就可以使用 http://www.example.org/albums/2009/08/hiking/duplicate;t=a5d0e32ddff373df1b3351e53fc6ffb1 这个 URI 来创建相册的副本。

```
# 请求
POST /albums/2009/08/1011/duplicate;t=a5d0e32ddff373df1b3351e53fc6ffb1 HTTP/1.1 ❶
Host: www.example.org

# 响应
HTTP/1.1 201 Created
Content-Type: application/xml;charset=UTF-8
Location: http://www.example.org/2009/08/1014 ❷
Content-Location: http://www.example.org/2009/08/1014

<album xmlns:atom="http://www.w3.org/2005/Atom">
```

```
  <id>urn:example:album:1014</id>
  <atom:link rel="self" href="http://www.example.org/albums/2009/08/1014"/>
  ...
</album>
```

❶ 用于复制资源的请求

❷ 原始资源的副本

在这个实现中，服务器创建了副本并为它提供了一个新的 URI。客户端完全不需要知道服务器的实现细节。 186

11.2 如何合并资源

问题描述

您想知道如何合并两个或更多的资源。

解决方案

设计一个和具体应用相关的控制器资源用于合并资源。客户端向这个 URI 提交一个 GET 请求，查询参数中包含需要合并到这个控制器的资源的 URI 或者标识符。服务器返回 Last-Modified 头和 ETag 头,还有那些要被合并到表述正文中的资源的摘要。在实体标签中，包含一个序列号或者把时间戳和一个随机数字拼接在一起。

除了校验这个摘要，客户端还要向触发合并的 URI 发出一个 POST 请求，并提供 If-Unmodified-Since 和 If-Match 头。

合并完成之后，服务器在事务日志里保存 If-Match 头的值并返回响应码 201（Created），带上包含合并后资源 URI 的 Location 头。今后，如果客户端提交的 POST 请求具有相同的 If-Match 值，那么服务器就会返回响应码 412（Precondition Failed）。

问题讨论

一次合并操作会涉及服务器上的两个或更多资源。为了确保客户端的松耦合度，合并的细节应该完全由服务器负责。下面的例子中，客户端通过请求一个 URI 将两本相册合并成一本新的相册：

```
# 请求
GET /albums/merge?src=urn%3Aexample%3Aalbum%3A1011&
  dest=urn%3Aexample%3Aalbum%3A1012 HTTP/1.1 ❶
Host: www.example.org
```

```
# 响应
HTTP/1.1 200 OK
Cache-Control: no-cache
Content-Type: application/xml;charset=UTF-8
Last-Modified: Sun, 08 Nov 2009 04:47:03 GMT ❷
ETag: "d88a39e41c314f57917da04c920fd608"

<albums> ❸
 <album>
   <id>urn:example:album:1011</id>
   <atom:link rel="self" href="http://www.example.org/albums/2009/08/1011"/>
   ...
 </album>
 <album>
   <id>urn:example:album:1012</id>
   <atom:link rel="self" href="http://www.example.org/albums/2009/08/1012"/>
   ...
 </album>
</albums>
```

❶ 该请求用于合并资源的当前状态

❷ 表述的 Last-Modified 和 ETag

❸ 被合并资源的状态

在本例中，客户端提供了需要合并的两个资源的标识符，服务器返回这两个资源的摘要。除此之外，还可以将 URI 作为合并的参数。

这一步本质上就是为被合并资源创建了一个新的复合资源（详见 2.4 节）。

在客户端，需要进行校验，确保响应中的相册状态与客户端在发送合并请求之前保存在本地的那个状态是相同的。

```
# 请求
POST /albums/merge?src=urn%3Aexample%3Aalbum%3A1011&
  dest=urn%3Aexample%3Aalbum%3A1012 HTTP/1.1 ❶
Host: www.example.org
If-Unmodified-Since: Sun, 08 Nov 2009 04:47:03 GMT ❷
If-Match: "d88a39e41c314f57917da04c920fd608"

# 响应
HTTP/1.1 201 Created
Location: http://www.example.org/albmus/2009/08/1091
```

```
Content-Location: http://www.example.org/albmus/2009/08/1091
Content-Type: application/xml;charset=UTF-8
Last-Modified: Sun, 08 Nov 2009 05:30:10 GMT
ETag: "48be3ab269550ee00a84eb5a1a44f330"

<album>
  <id>urn:example:album:1091</id>
  <atom:link rel="self" href="http://www.example.org/albums/2009/08/1091"/>
  ...
</album>
```

❶ 用于合并资源的请求

❷ 先决条件

服务器创建了一个新的合并资源。在此过程中，根据用例要求的不同，服务器可能也会同时删除原始的资源。

11.3 如何移动资源

188

问题描述

您想知道如何移动资源。

解决方案

包含一个链接或者链接模板，它指向一个可以移动该资源的控制器。让客户端使用 POST 来提交移动请求。使用 10.8 节中介绍的技术让这个 POST 请求根据条件执行。

在请求处理完毕之后，根据结果返回响应码 201(Created)或者响应码 303(See Other)。

问题讨论

“移动”的确切含义是随具体应用而变。它可以指将一个资源复制到同一服务器上（或者另外一台服务器上）的不同位置，并删除原始资源，也可以指在不改变资源位置的情况下修改其状态。不管是哪种，为了保持松耦合度，客户端自身不应关心移动操作在服务器端的具体含义和实现细节。

还是用相册的例子。某个相册位于“friends”文件夹内，客户端想要将该相册移动到“family”文件夹下。服务器的 URI 是根据相册的分类来组织的，移动操作会导致相册 URI 的改变。

移动资源和复制资源差别并不大。在本例中，服务器可以提供一个 URI 或者 URI

模板，让客户端给出目标资源需要满足的条件:

```
# 请求
GET /albums/friends/2009/08/1011 HTTP/1.1
Host: www.example.org

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

<album xmlns:atom="http://www.w3.org/2005/Atom">
  <id>urn:example:album:1011</id>
  <atom:link rel="self" href="http://www.example.org/albums/2009/08/1011"/>
  <link-template rel="http://www.example.org/rels/move"
     href="http://www.example.org/albums/friends/2009/08/1011/move;
           t=dc4128786d463dc7e40c18457d1826fa?group={category}"/> ❶
  <category>friends</category>
  ...
</album>
```

❶ 带有 URI 模板的链接，用于改变资源的分类

189 在这个表述中，服务器为客户端提供了一个 URI 模板，让其可以指定一个分类，作为资源移动的目的地。

```
# 请求
POST /albums/friends/2009/08/1011/move;t=dc4128786d463dc7e40c18457d1826fa?
  group=family HTTP/1.1 ❶
Host: www.example.org
Content-Length:0

# 响应
HTTP/1.1 201 Created
Content-Type: application/xml;charset=UTF-8
Location: http://www.example.org/family/2009/08/1021 ❷
Content-Location: http://www.example.org/family/2009/08/1021

<album>
  <id>urn:example:album:1021</id>
  <atom:link rel="self" href="http://www.example.org/family/albums/2009/08/1021"/>
  <category>family</category>
  ...
</album>
```

❶ 修改分类的请求

❷ 新的资源

这个请求会产生一个新资源，其 URI 是 http://www.example.org/albums/family/2009/08/1021。如果客户端还试图访问原始相册，服务器可以返回响应码 410（Gone）或者 404（Not Found）。

11.4 何时使用 WebDAV 方法

WebDAV（RFC 4918）是 HTTP 的一个扩展，用于资源的分布式创作和版本控制。它扩展了 HTTP 并规定了一组 HTTP 方法和标头，来管理文件和文档。该协议包含下列扩展方法：

PROPFIND

在 WebDAV 中，文档可以具有属性，而客户端可以通过这个方法获取这些属性。

PROPPATCH

客户端可以使用这个方法设定、添加或者删除资源的属性。

MKCOL

WebDAV 允许将文档归组为集合（collection），客户端可以使用这个方法创建新的集合。

COPY

客户端可以使用这个方法在目标 URI 中创建给定资源的副本。

MOVE

这个方法和 COPY 类似，只是服务器会在操作中删除原始资源。

190

LOCK

这个方法让客户端可以锁定某个文档，启用悲观并发控制。

UNLOCK

客户端可以使用该方法解锁之前锁定的资源。

问题描述

您想知道何时可以使用 WebDAV 中规定的方法。

解决方案

当您的 Web 服务是一个内容创作应用程序，并且服务器支持 WebDAV，那么就使用 WebDAV 的扩展方法。对于其他种类的应用程序，应该避免使用 WebDAV。

问题讨论

需要强调的是，WebDAV 方法都是“以文件为中心”的操作，它们假设资源都是像文件对象一样，易于复制、覆盖和更名等。然而，在大多数 RESTful Web 服务中，资源并不是文件，以文件为中心的视角通常难以映射到应用资源和业务场合中。

WebDAV 通常被作为一个范例来展示如何扩展 HTTP 以满足特定应用领域的需求。设计 WebDAV 扩展是为了允许客户端编辑和管理位于远程服务器上的文档。下面的例子演示了如何复制一个资源：

```
# 复制资源的请求
COPY /report/working/2010.pdf HTTP/1.1
Host: www.example.org
Destination:  http://www.example.org/projections/2010.pdf

# 响应
HTTP/1.1 201 Created
```

在这个请求中，客户端通过一个 Destination 头来选择副本的目标位置。客户端也可以将该方法应用于其他 WebDAV 资源之上，比如属性（properties）和集合。客户端还可以提供一个 Overwrite 头来告诉服务器是否应当覆盖任何已经存在于目标 URI 的资源，或者指定一个 Depth 头来标明复制集合时的深度。

```
# 复制资源的请求
COPY /report/working/2010.pdf HTTP/1.1
Host: www.example.org
Destination:  http://www.example.org/projections/2010.pdf
Overwrite:F

# 响应
HTTP/1.1 201 Created
```

191

类似的，还可以使用 MOVE 方法来将资源从一个位置移动到另一个位置。

```
# 请求
MOVE /report/working/2010.pdf HTTP/1.1
Host: www.example.org
Destination: http://www.example.org/projections/2010.pdf

# 响应
HTTP/1.1 201 Created
Location: http://www.example.org/archives/this-resource
```

COPY 和 MOVE 方法都是原子性的，可以应用于跨服务器的资源。例如，实际的移动操作可能会要求先将一个或多个资源从源服务器上复制到目标服务器上，然后再删

除源服务器上的资源。只有在资源被成功复制到目标服务器之后，源服务器上的资源才可以被删除。这些方法同时还要求客户端负责处理诸如锁定、解锁甚至跨服务器复制资源之类的细节。

11.5 如何支持跨服务器的操作

本节将讨论如何支持那些跨越服务器边界的操作。例如将一份用户资料从一个应用程序迁移到另一个应用程序，将待定报价的摘要导入到客户关系管理应用程序中，或者将目前存在于草稿服务器上的文档发布到生产服务器上。类似这样的用例需要操作多个服务器中的状态。

问题描述

您想知道如何发起一个要改变位于两台或更多服务器上的资源的操作。

解决方案

让服务器彼此协作，设计并实现跨服务器的操作。这可能会要求服务器之间在数据格式、后端接口以及并发控制上达成共识，还要从同一个数据存储载入数据，将数据规范化以适应其他服务器的格式，然后存储。让其中一台服务器为客户端提供链接来触发操作。

问题讨论

当面临需要跨越服务器边界的操作时，关键就是要分离关注点。考虑下面两个 Web 服务：一个负责管理用户的联系人，运行于 http://contacts.example1.org，另一个负责管理消息，比如电子邮件、短信、语音留言等，运行于 http://messaging.example2.org。现在要让一名用户将他的联系人导入到消息 Web 服务中。要支持这一功能，先假定联系人 Web 服务包含一个链接可以将联系人导出到消息 Web 服务中。

192

以下是用户的联系人列表表述，包含可以将联系人列表导出到消息 Web 服务中去的链接：

```
# 请求
GET /user/smith/contacts HTTP/1.1
Host: contacts.example1.org

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

```
<contacts xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link     rel="self"     href="http://contacts.example2.org/user/smith/
contacts"/>
  <atom:link rel="http://contacts.example1.org/rels/export-to-messaging"
    href="http://messaging.example2.org/user/smith/import;
          t=bcb9169866c69410be37f68210a6986c"
    title="Export contacts into to messaging."/> ❶
  <contact>
    ...
  </contact>
  ...
</contact>
```

❶ 将数据导出到其他服务器的链接

如果客户端理解链接关系类型 http://contacts.example2.org/rels/export-to-messaging 的语义，它就能发起导出操作。假设该链接关系类型的文档中说明，客户端需要向链接 URI 提交一个 POST 请求，将联系人导出到消息服务中。链接的 URI 含有一个安全令牌来防止未经授权的使用。

```
# 请求
POST /user/smith/import;t=bcb9169866c69410be37f68210a6986c HTTP/1.1 ❶
Host: messaging.example2.org

# 响应
HTTP/1.1 303 See Other
Location: http://messaging.example2.org/user/smith ❷
Content-Type: application/xml;charset=UTF-8
```

❶ 导出数据的请求

❷ 提供导出结果的 URI

当客户端提交该请求时，消息服务会产生一个对联系人 Web 服务的后端请求，取得一份联系人的副本。在这一过程中，依照服务器之间的安全约定，消息 Web 服务可能会将 URI 中的令牌传给联系人 Web 服务。

193 在这个过程中，服务器负责确保联系人列表数据在消息服务器上是可用的。客户端的职责仅仅是触发这一操作。这解耦了客户端与服务器的实现细节，包括并发控制、原子性、数据格式的区别等。这一切都是服务器互相配合的结果。

如果由于技术或组织方面的问题而使得这样的配合无法实现，那么客户端就别无选择，必须从 http://contacts.example2.org/user/smith/contacts 下载所有的联系人，然后再提交给消息服务。详见 2.6 节的例子。

11.6 如何获取资源的快照

本节描述如何在每个更新之前获取一份资源快照。当客户端通过 PUT 方法更新资源时，服务器会更新资源的当前状态，这样一来客户端就没法知道在更新之前的情况了。某些时候，客户端也希望能够回溯到某一时间，浏览修改的历史记录。

例如，假设现在要设计一个 Web 服务，提供某个十字路口过去和现在的交通情况的快照。大多数客户端会想要最新的情况，不过有时也可能想浏览过去的交通情况。

或者考虑一个 Wiki 的例子。对于 Wiki 上的每个页面，服务器都需要维护一个修改栈，保存每个页面当前和过去的修改记录，以便客户端能够获取、比较并评估对页面所做的修改。

问题描述

您想知道如何保存资源的快照，以便客户端能够浏览资源之前的版本。

解决方案

每当客户端提交 PUT 请求更新资源时，在修改资源之前，先隐式地创建一个快照（资源的一份副本）。在更新后的资源的表述中，包含一个链接指向该快照。同时也在快照里包含一个链接指向更新后的资源。

当客户端提交 DELETE 请求时，就删除资源，并删除所有的快照。

问题讨论

本质上，这一节的内容是让您为资源创建一个简单的版本控制机制，并且客户端无须知道如何创建和管理资源的版本。那些不关心版本的客户端可以和往常一样使用 HTTP 的统一接口，甚至压根就不会注意到服务器维护着快照。那些确实需要关心版本信息的客户端则可以浏览过去的版本。在服务器的存储空间中，每个快照都可以是一份完整副本或者只包含与先前版本的差异列表。 194

以下是一个创建新快照的范例：

```
# 请求
PUT /trails/ColchuckLake HTTP/1.1
```

```
Host: wiki.example.org
If-Unmodified-Since: Sun, 01 Nov 2009 12:34:43 GMT
Content-Type: application/atom+xml;type=entry;charset=UTF-8

<atom:entry xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:author>
    <atom:name>Joe Hiker</atom:name>
  </atom:author>
  <atom:title>Colchuck Lake</atom:title>
  <atom:id>urn:example:wiki:trails:ColcuckLake</atom:id>
  <atom:content xml:lang="en" type="html">
    ... initial draft ...
  </atom:content>
</atom:entry>

# 响应
HTTP/1.1 204 No Content
```

作为对 PUT 请求的响应，服务器将资源复制到了 http://wiki.example.org/trails/ColchukLake/s1。

```
# 请求
GET /trails/ColchuckLake HTTP/1.1
Host: wiki.example.org

# 响应
HTTP/1.1 200 OK
Last-Modified: Sun, 01 Nov 2009 16:24:56 GMT
Content-Type: application/atom+xml;type=entry;charset=UTF-8

<atom:entry xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:author>
    <atom:name>Joe Hiker</atom:name>
  </atom:author>
  <atom:title>Colchuck Lake</atom:title>
  <atom:id>urn:example:wiki:trails:ColcuckLake</atom:id>
  <atom:link href="http://wiki.example.org/trails/ColchuckLake" rel="self"/> ❶
  <atom:link href="http://wiki.example.org/trails/ColchuckLake/s1" rel=
       "previous"/> ❷
  <atom:content xml:lang="en" atom:type="html">
    ... updated draft ...
  </atom:content>
</atom:entry>
```

❶ 资源的当前版本

❷ 快照

带有 previous 关系的链接指向资源的上一个版本。当客户端提交另一个 PUT 请求来修改资源时，服务器端可以重复上述过程，在 http://wiki.example.org/trails/ColchuckLake/s2 处创建一份新的快照。

195

```
# 更新资源的请求
PUT /trails/ColchuckLake HTTP/1.1
Host: wiki.example.org
Content-Type: application/atom+xml;type=entry;charset=UTF-8
If-Unmodified-Since: Sun, 01 Nov 2009 16:24:56 GMT

<atom:entry xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:author>
    <atom:name>Joe Hiker</atom:name>
  </atom:author>
  <atom:title>Colchuck Lake</atom:title>
  <atom:id>urn:example:wiki:trails:ColcuckLake</atom:id>
  <atom:content xml:lang="en" type="html">
    ... updated draft ...
  </atom:content>
</atom:entry>

# 响应
HTTP/1.1 204 No Content
```

这一步之后，位于 http://wiki.example.org/trails/ColchuckLake 的是资源的最新版本。它之前的版本位于 http://wiki.example.org/trails/ColchuckLake/s2，再之前的版本则位于 http://wiki.example.org/trails/ColchuckLake/s1。通过具有 previous 关系类型的链接，客户端可以回溯之前的快照。

在这个更新之后，http://wiki.example.org/trails/ColchuckLake/s2 的表述包含了它自己的快照——http://wiki.example.org/trails/ColchuckLake/s2。

```
# 请求
GET /trails/ColchuckLake/s2 HTTP/1.1
Host: wiki.example.org

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

<atom:entry xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:author>
    <atom:name>Joe Hiker</atom:name>
  </atom:author>
  <atom:title>Colchuck Lake</atom:title>
  <atom:id>urn:example:wiki:trails:ColcuckLake</atom:id>
```

```
    <atom:link href="http://wiki.example.org/trails/ColchuckLake/s2" rel="self"/> ❶
    <atom:link href="http://wiki.example.org/trails/ColchuckLake/s1" rel="previous"/> ❷
    <atom:link href="http://wiki.example.org/trails/ColchuckLake" rel="next"/> ❸
    <atom:content xml:lang="en" atom:type="html">
     ... 更新后的草稿 ...
    </atom:content>
</atom:entry>
```

196

❶ 资源的当前版本

❷ 另一份快照

❸ 更新的快照

客户端可以使用具有 previous 关系类型的链接跳转到上一个快照，或者使用具有 next 关系类型的链接跳转到下一个快照。

11.7 如何撤销资源更新

少数情况下，您可能会需要为客户端提供一个“撤销”功能。例如，客户端可能想撤销对某个报价的变更。这个问题同建立快照类似，只是增加了将最新的快照重新激活的功能。

问题描述

您想知道如何撤销对某个资源所做的修改。

解决方案

每当客户端提交 PUT 请求更新资源时，创建资源的一份快照，详见 11.6 节。

提供一个控制器资源专门用于撤销操作。要撤销一个修改，让客户端提交一个 POST 请求。将资源的当前状态记录到一个事务日志中以备审计。服务器用最新的快照恢复资源的状态，并将客户端重定向至资源的 URI。

问题讨论

考虑 11.6 节中引入的 Wiki 文档 http://wiki.example.org/trails/Colcuck Lake。在该资源的表述中，包含一个用于撤销的链接。

```
# 请求
GET /trails/ColchuckLake HTTP/1.1
Host: wiki.example.org
```

```
# 响应
HTTP/1.1 200 OK
Last-Modified: Sun, 01 Nov 2009 16:24:56 GMT
Content-Type: application/atom+xml;type=entry;charset=UTF-8

<atom:entry xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:author>
    <atom:name>Joe Hiker</atom:name>
  </atom:author>
  <atom:title>Colchuck Lake</atom:title>
  <atom:id>urn:example:wiki:trails:ColcuckLake</atom:id>
  <atom:link href="http://wiki.example.org/trails/ColchuckLake" rel="self"/>
  <atom:link href="http://wiki.example.org/trails/ColchuckLake/s1" rel="previous"/>
  <atom:link href="http://wiki.example.org/trails/ColchuckLake/undo;
           t=72f2a2342ce7dc806ae7697e138bad71"
           rel="http://wiki.example.org/rels/undo"/> ❶
  <atom:content xml:lang="en" atom:type="html">
    ... 更新后的草稿 ...
  </atom:content>
</atom:entry>
```

197

❶ 用于撤销资源当前状态的链接

客户端可以向具有 http://wiki.example.org/rels/undo 关系类型的链接提交一个"撤销"请求。该链接中的 URI 是一个控制器资源，它能够撤销一个修改。

```
# 请求
POST /trails/ColchuckLake/undo;t=72f2a2342ce7dc806ae7697e138bad71
Host: wiki.example.org
Content-Length:0

# 响应
HTTP/1.1 303 See Other
Location: http://wiki.example.org/trails/ColcuckLake
```

控制器会还原资源的当前状态，并记录下撤销请求以备审计。后续对资源的 GET 请求就将获得被还原后的状态了。

```
# 请求
GET /trails/ColchuckLake HTTP/1.1
Host: wiki.example.org

# 响应
HTTP/1.1 200 OK
Last-Modified: Sun, 01 Nov 2009 16:24:56 GMT
Content-Type: application/atom+xml;type=entry;charset=UTF-8

<atom:entry xmlns:atom="http://www.w3.org/2005/Atom">
```

```
    <atom:author>
      <atom:name>Joe Hiker</atom:name>
    </atom:author>
    <atom:title>Colchuck Lake</atom:title>
    <atom:id>urn:example:wiki:trails:ColcuckLake</atom:id>
    <atom:link href="http://wiki.example.org/trails/ColchuckLake" rel="self"/>
    <atom:content xml:lang="en" atom:type="html">
      ... 更新后的草稿 ...
    </atom:content>
  </atom:entry>
```

只需稍做扩展就能实现对重做（redo）的支持了。

198

11.8 如何为部分更新提炼资源

某些时候，很可能会遇到需要部分（partially）更新一个资源的情况。比如，资源可能很大，而需要做的更新又很少。这种情况下，如果提交 GET 请求来获得资源的全部表述，做完这个小修改，再用 PUT 请求将整个表述提交回服务器以完成对资源的更新，这个过程就显得有些浪费了。对此，首先想到的方案就是将所涉及的资源加以提炼，去除对部分更新的需求。11.9 节将介绍另一种解决方案。

问题描述

您想知道如何提炼资源，使得客户端能够部分更新资源。

解决方案

设计一个新的资源，将客户端可以修改的那部分资源封装起来。让客户端使用 PUT 请求来更新这个资源，在实际效果上就部分地更新了原始资源。

问题讨论

这个解决方案的关键优点在于，它让客户端可以通过 HTTP 的 PUT 方法来更新原始资源的一个子集。并且还有个附加的好处，即让那些子集之外的隐藏资源可以有自己的 URI，客户端可以通过这些 URI 来访问它们。

例如，考虑下面的“客户”资源的表述：

```
# 请求
GET /customers/1 HTTP/1.1
Host: www.example.org

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

```
Last-Modified: Thu, 05 Nov 2009 01:54:19 GMT
ETag: "ca87aa4ff1505934281d91f807b25b3c"

<customer xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="self" href="http://www.example.org/customers/1"/>
  <name>J. P. Goodright</name>
  <status>active</status>
  <address>
    <street>123 Main</street>
    <city>Byteville</city>
    <state>MD</state>
    <postal-code>12345</postal-code>
  </address>
  <billing-contact>
   <name>P.J. Billingsley</name>
    <email>pjbill@example.org</email>
    <voice-phone>123-456-7890</voice-phone>
    <fax-line>234-567-8901</fax-line>
  </billing-contact>
  <services>
    <preferred-shipping>two-day</preferred-shipping>
    <billing-method>net-30</billing-method>
    <minimum-order>1000</mininum-order>
    <customer-discount>10%</customer-discount>
  </services>
</customer>
```

199

假设想要更新客户资源的账单联系人（billing contact）信息。服务器端可以提炼资源，引入一个新的“billing-contact”资源，其中包括从原始资源中选取出的元素。

```
# 请求
GET /customers/1/billing-contact HTTP/1.1
Host: www.example.org

# 响应
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8
Last-Modified: Thu, 05 Nov 2009 01:54:19 GMT
ETag: "d65b17759967753e7eb37b28f1bdb1fa"

<billing-contact xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link ref="self" href="http://www.example.org/customers/1/billing-contact"/>
  <name>P.J. Billingsley</name>
  <email>pjbill@example.org</email>
  <voice-phone>123-456-7890</voice-phone>
  <fax-line>234-567-8901</fax-line>
</billing-contact>
```

然后可以提炼客户资源的表述，包含一个链接指向 billing-contact 资源。

```
# 请求
GET /customers/1 HTTP/1.1
Host: www.example.org

# 响应
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8
Last-Modified: Thu, 05 Nov 2009 01:54:19 GMT
ETag: "99d37a01bc588d8743f704eaffdb22b0"

<customer xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link rel="self" href="http://www.example.org/customers/1"/>
  <atom:link rel="related" href="http://www.example.org/customers/1/billing-contact"/>
  <name>J. P. Goodright</name>
  <status>active</status>
  <address>
    <street>123 Main</street>
    <city>Byteville</city>
    <state>MD</state>
    <postal-code>12345</postal-code>
  </address>
  <services>
    <preferred-shipping>two-day</preferred-shipping>
    <billing-method>net-30</billing-method>
    <minimum-order>1000</mininum-order>
    <customer-discount>10%</customer-discount>
  </services>
</customer>
```

200

客户端现在可以通过向 `http://www.example.org/customers/1/billing-contact` 的资源提交 PUT 请求来更新账单联系人信息了。

还可以把这个方法扩展到客户资源的其他部分。例如，下面的 PUT 请求会更新客户状态和首选发货方式：

```
# 请求
PUT /customer/1234/info HTTP/1.1
Content-Type: application/xml;charset=UTF-8
If-Match: "bfcc688bd542e17f27da0f82200c35ea"
If-Unmodified-Since: Thu, 05 Nov 2009 01:54:19 GMT

<customer-info>
  <status>premium</status>
  <preferred-shipping>next-day</preferred-shipping>
</customer-info>
```

另外，假设客户端想要删除客户的现有E-mail地址，但同时还会添加两个新的E-mail地址。这样的情况下，你可以将全部E-mail地址表述为另一个资源，并让客户端提交PUT请求更新这个资源：

```
# 请求
PUT /customer/1234/emails HTTP/1.1
Content-Type: application/xml;charset=UTF-8
If-Match: "b5b55c8a7f18dd77b4b2d94eed7f1be5"
If-Unmodified-Since: Thu, 05 Nov 2009 01:54:19 GMT

<emails>
  <email type="work">pjbill1@example.org</status>
  <email type="alt">jane1@example.org</status>
</emails>
```

这样的资源可能会显得不一致，或者“有污染”。“customer info”或“emails”资源并没有存在的理由，仅仅是为了支持对客户资源的某些部分更新。但需要记住的是，任何适用于获取和更新的内容都可以被作为资源。

11.9 如何使用PATCH方法

201

HTTP的PUT方法是用来完整更新或替换一个资源的，而PATCH方法则是用来支持部分更新的。截止本书写作时，定义这个方法的规范还没有正式发布（详见*http://tools.ietf.org/html/draft-dusseault-http-patch*）。

问题描述

您想知道如何使用PATCH方法。

解决方案

PATCH方法是一个不安全且非幂等的HTTP方法。请求的正文是一个表述，描述了一组要对资源执行的修改。当收到这个请求时，服务器会将修改应用到资源上，然后返回响应码200（OK）或者204（No Content）。

通过要求客户端提供If-Unmodified-Since和/或If-Match头，可以让该方法的实现变得条件化。如果所提供的先决条件不匹配，则返回响应码412（Precondition Failed）。

通过OPTIONS响应的Allow头来告知是否支持PATCH方法。同时还要包含一个Accept-Patch头，其值是PATCH方法所支持的媒体类型。

问题讨论

在下面的例子中，客户端提交 PATCH 请求来修改客户的姓名、添加昵称，并删除传真号码。除了请求中的表述，这个请求响应过程与使用条件化 PUT 请求的过程是类似的。

```
# 部分更新的请求
PATCH /customers/1 HTTP/1.1 ❶
Host: www.example.org
If-Match: "2a7ad61820b6ba89e6c4a119e22f7dfc" ❷
If-Unmodified-Since: Thu, 05 Nov 2009 01:00:01 GMT
Content-Type: application/xml;charset=UTF-8

<diff-customer> ❸
  <replace-name>J. P. Goodright, Jr.</replace-name>
  <add-nickname>Jimmy</add-nickname>
  <remove-fax-line/>
</diff-customer>

# 响应
HTTP/1.1 204 No Content
```

❶ 部分更新资源的请求

❷ 请求的先决条件

❸ 表述的正文，描述了需要做的修改

202 在本例中，服务器使用 XML 格式将对“customer”资源所做的修改描述为简单的命令，比如“replace name”、“add nickname”以及“remove fax line”。

要支持 PATCH 方法，服务器需要定义一个能够表达修改的表述格式。这一问题的解决方案与媒体类型有关。例如，针对 XML 文档的表达方式和针对二进制图像文件相比可能会大相径庭。此外，并不存在可以复用的 IANA 注册媒体类型和格式来描述修改。

在响应中，服务器可以包含一个 Content-Location 头，以及最新的 Last-Modified 和/或 ETag 头。否则，客户端就必须发出一个非条件化的 GET 请求（详见 10.6 节）来获取更新后的资源表述，以及最新的 ETag 和 Last-Modified 头。

由于 PATCH 方法不是幂等的，因此客户端要避免发送重复的 PATCH 请求。

除了响应码 422 (Unprocessable Entity)，PATCH 方法的响应码和其他 HTTP 方法类似。由于可能导致资源状态错误，服务器不能接受请求时，就返回响应码 422。如下例:

```
# 部分更新的请求
PATCH /customers/1 HTTP/1.1
If-Match: "2ce12fc9d303b1eee1ed3efe9713663c"
If-Unmodified-Since: xxx
Host: www.example.org
Content-Type: application/xml;charset=UTF-8

<diff-customer>
  <remove>
    <name/>
    <fax-line/>
  </remove>
</diff-customer>

# 响应
422 Unprocessable Entity
Content-Type: application/xml;charset=UTF-8

<error>
  <message xml:lang="en">Name cannot be removed.</message>
</error>
```

要避免这样的错误，可以让 PATCH 请求的表述格式只包含那些合法的修改组合，当它们被执行后不会造成资源状态不一致。

有一个方法可以确保 PATCH 请求只包含合法的修改组合，就是为每个资源设计一个专门的格式，如同前面例子中那样，并避免使用通用的 diff 格式。通用的 diff 工具包括 UNIX 中的 diff 或者微软的 XML Diff 和 Patch 工具（*http://msdn.microsoft.com/en-us/library/aa302294.aspx*）。

203

最后，在 OPTIONS 响应中添加 Accept-Patch 头，列出本方法所支持的媒体类型。

```
# 请求
OPTIONS /customers/1
Host: http://www.example.org

# 响应
HTTP/1.1 204 No Content
Allow: POST, GET, PATCH
Accept-Patch: application/xml
```

在本书写作时，PATCH 方法的规范还很新，今后可能还会有小的修改。如果对这一

问题有所顾虑，就使用 11.8 节中的方案或者用 POST 方法替代 PATCH。

```
# 使用 POST 进行部分更新的请求
POST /customers/1 HTTP/1.1
Host: www.example.org
If-Match: "2a7ad61820b6ba89e6c4a119e22f7dfc"
If-Unmodified-Since: Thu, 05 Nov 2009 01:00:01 GMT
Content-Type: application/xml;charset=UTF-8

<diff-customer>
  <replace-name>J. P. Goodright, Jr.</replace-name>
  <add-nickname>Jimmy</add-nickname>
  <remove-fax-line/>
</diff-customer>

# 响应
HTTP/1.1 303 See Other
Location: http://www.example.org/customers/1
Content-Length:0
```

无法使用 PATCH 时，使用 POST 而非 PUT 来做部分更新。在 HTTP 中，PUT 是专门用来完整更新或替换资源的。

11.10 如何批量处理相似的资源

当客户端需要针对不同的资源提交多个相似的请求时，只要在每个资源上的操作都相同，并且资源是“相似”的，那么就可以将它们合并为针对一个资源集合的单一操作。11.11 节和 11.13 节展示了解决“批量处理”相关问题的其他方案。

问题描述

204

您想知道如何同时创建、更新或者删除多个相似的资源。

解决方案

使用 POST 和集合资源来同时创建多个相似的资源。让客户端在请求中包含需要创建的资源信息。为所有这些创建的资源分配一个 URI，并用响应码 303 (See Other) 将客户端重定向到资源集合。该资源的表述中包含全部新创建资源的链接。

要批量地更新或删除多个相似资源，使用一个可以返回包括所有这些资源信息的表述的 URI。向这个 URI 提交一个 PUT 请求，附带上需要更新的资源信息，或者提交一个 DELETE 请求来删除这些资源。

在所有这些情况中都要确保请求处理的原子性。

由于批量操作可能会持续较长时间，所以要限制请求的大小，以避免服务器过载。否则，客户端可以无意或有意地导致拒绝服务攻击。

此外，对批量操作的并发性和原子性的控制也可能长时间锁定数据库，这会降低性能和可伸缩性。

问题讨论

考虑下列用例:

- 创建 10 个地址。
- 从用户的愿望单里删除 5 本书。
- 更新用户收藏夹中的所有电影。

对于这些用例，如果能够通过一个 URI 标识全部的资源，客户端就能使用合适的 HTTP 方法整体操作这个集合。例如，客户端可以提交一个 POST 请求来创建 10 个地址。

```
# 请求
POST /user/user002/addresses HTTP/1.1 ❶
Host: www.example.org
Content-Type: application/xml;charset=UTF-8

<addresses> ❷
  <address>
    ...
  </address>
  <address>
   ...
  </address>
  ...
</addresses>

# 响应
HTTP/1.1 303 See Other
Location: http://www.example.org/user/user002/addresses ❸
```

❶ 作为批量创建资源的工厂的集合

❷ 包含所有需要创建的资源数据的表述

205

❸ 跳转到集合获取结果

客户端需要进行重定向以获得所有新创建资源的 URI。

```
# 请求
GET /user/user002/addresses 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:3600
Last-Modified: Thu, 05 Nov 2009 01:54:19 GMT
ETag: "474263ccac5ad16fbe9f48bf63ef0846"

<addresses xmlns:atom="http://www.w3.org/2005/Atom">
  <address>
    <atom:link rel="self" href="http://www.example.org/user/user002/addr001"/>
    ...
  </address>
  <address>
    <atom:link rel="self" href="http://www.example.org/user/user002/addr002"/>
    ...
  </address>
  ...
</addresses>
```

上面这样的操作降低了统一接口的可见性。这个例子中，由于服务器创建了多个资源，所以它不能使用响应码 201 (Created) 并提供一个单一的 Location 头。客户端需要从集合正文中读出所创建资源的 URI。

这是客户端的方便性/网络效率以及可见性之间的一种权衡。

类似的，还可以通过对集合发起 PUT 请求批量地更新多个资源。下面的例子替换了用户收藏夹中的所有电影：

206

```
# 请求
PUT /user/user002/favmovies HTTP/1.1
Host: www.example.org
Content-Type: application/xml;charset=UTF-8
If-Match: "10895276d1cfdabdce24e6e902222198"

<favmovies>
  <favmovie>
    <id>urn:example:movie:2001</id>
    ...
  </favmovie>
  <favmovie>
```

```
    <id>urn:example:movie:2002</id>
    ...
  </favmovie>
</favmovies>

# 响应
HTTP/1.1 204 No Content
```

可以用同样方式实现批量删除。

服务器必须把批量请求实现成原子操作。如果请求中要创建 10 个地址，那么服务器应当在返回成功响应码之前完成全部这 10 个地址的创建。如果出现错误，服务器不能部分提交变更。

11.11 如何触发批量操作

客户端需要批量地进行某些操作并不是罕见的情况。比如，创建昨日销售订单的摘要，将一篇或多篇文章归档，批准一组选定的订单等，这些都涉及需要批量执行的任务。当服务器已经具备了执行批量操作所需的大部分数据时，可以使用本节讲到的内容。11.12 节和 11.13 节将讨论其他情况。

问题描述

您想知道如何设计一个应用相关的资源来触发批量操作。

解决方案

设计一个控制器资源，它可以根据客户端的输入来启动批量操作，允许客户端使用 POST 请求来开始处理。如果客户端需要跟踪该过程或者需要提交大量的数据，那就使用 1.10 节中的方案，返回响应码 202（Accepted）。否则就返回 200（OK）或者 204（No Content）。

问题讨论

本节的内容展示了如何使用“fire and forget”[注1]的策略来处理批量操作。举例来说，考虑这样一种情况，需要修正系统中所有创建于 2010 年之前的用户的地址。服务器存有大量从遗留系统中收集的地址信息，现在需要将它们重新修改为邮政服务所使用的格式。解决此问题的一种办法是让客户端遍历每个地址，进行修改，把地址标

207

注 1：　fire-and-forget，根据 Wikipedia 的解释，即指在武器发射后不需要任何外界干预就能更新自己的坐标或目标信息。详见 http://en.wikipedia.org/wiki/Fire-and-forget。

记为合法的或者需要手工修改的，然后保存，就像下面这样：

```
# 获取每个地址
GET /user/001/address HTTP/1.1
Host: www.example.org

# 在本地进行地址修正
...

# 保存更新后的地址
PUT /user/001/address HTTP/1.1
Host: www.example.org
Content-Type: application/xml;charset=UTF-8

...
```

在这个过程中，对每个地址的处理都涉及两次请求，除此之外，客户端还可以将全部的地址作为一个集合获取到本地并进行修改，然后使用 11.10 节中的方案，通过一个请求更新全部地址。

```
# 获取所有地址
GET /addresses?before=2010-01-01 HTTP/1.1
Host: www.example.org

# 对所有地址进行修正
...

# 保存更新后的地址
PUT /addresses?before=2010-01-01 HTTP/1.1
Host: www.example.org
Content-Type: application/xml;charset=UTF-8

...
```

正如在 11.10 节里讨论的那样，设计一个能在单一请求中获取大量数据，并且原子化地处理它们的资源是比较麻烦的。较为简单的替代方案是“要求”服务器来完成对地址的修改。客户端提交一个 POST 请求让服务器启动地址修改的过程。此处不再需要其他输入了，因此请求的正文是空的。

```
# 进行地址修正的请求
POST /address-correction?before=2010-01-01 HTTP/1.1 ❶
Host: www.example.org
Content-Length:0

# 响应
HTTP/1.1 202 Accepted
Content-Type: application/xml;charset=UTF-8
```

208

```
Date: Sun, 13 Sep 2009 01:49:27 GMT

<status xmlns:atom="http://www.w3.org/2005/Atom">
  <state>pending</state>
  <atom:link href="http://www.example.org/address-correction/status/1" rel="self"/>
  <message xml:lang="en">Your request has been accepted for processing.</message>
  <ping-after>2009-09-13T01:59:27Z</ping-after>
</status>
```

❶ 触发一个批量任务后便不再过问

如果不需要对过程进行跟踪，服务器返回 200（OK）或者 204（No Content）就可以了。

```
# 进行地址修正的请求
POST /address-correction?before=2010-01-01 HTTP/1.1
Host: www.example.org
# 响应
HTTP/1.1 204 No Content
```

在这个例子中，客户端只需向一个已知 URI 提交一个 POST 请求来发起批量操作。服务器会自己完成对每个地址的修改。本质上来说，客户端所做的就是“按下开关”。

11.12 何时使用 POST 来合并多个请求

将若干 HTTP 请求合并为一个请求来支持批量处理的做法并不少见。以下是该方法的常见实现：

1. 客户端将每个 HTTP 请求（包括 URI、HTTP 方法名和 HTTP 标头）序列化为一个 JSON 对象或者 XML 文档，甚至是 multipart/mixed 消息的一部分。
2. 客户端创建一个包封（envelope）格式将这些请求合并到一个消息中。
3. 客户端用 POST 将这个消息提交至服务器的一个资源上，它通常被称为批处理端点（batch end point）。
4. 服务器在接收到这个消息后，将其解封，重新构建 HTTP 请求，然后将这些请求派发至相应的 URI。或者，也可以略过 HTTP，将这些请求直接派发至能够处理它们的代码上。
5. 服务器收集每个请求的响应，然后将它们序列化为一个消息，返回给客户端。
6. 客户端解封消息并处理每个响应消息。

示例如下:

```
# 批量请求
POST /batch HTTP/1.1 ❶
Host: www.example.org
Content-Type: application/xml;charset=UTF-8

<batch-request>
  <request method="PUT" uri="http://www.example.org/req/2009/11/1/log"> ❷
    <headers>
      <header name="Content-Type" value="application/xml"/>
    </headers>
    <body>
      ...
    </body>
  </request>
  <request method="POST" uri="http://www.example.org/req/2009/11/2/reject"> ❸
    <headers>
      <header name="Content-Type" value="application/xml"/>
    </headers>
    <body>
      ...
    </body>
  </request>
  ...
</batch-request>

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

<batch-response>
  <response status="200" message="OK"> ❺
    <headers>
      <header name="Content-Type" value="application/xml"/>
    </headers>
    <body>
      ...
    </body>
  </response>
  <response status="412" message="Precondition Failed"> ❻
    <headers>
      <header name="Content-Type" value="application/xml"/>
    </headers>
    <body>
      ...
    </body>
  </response>
  ...
```

```
</batch-response>
```

❶ 发至一个网关资源的 HTTP 请求

❷ 一个序列化了的 HTTP PUT 请求，用于更新资源

❸ 一个序列化了的 HTTP POST 请求，用于创建新资源

210

❹ 来自网关资源的 HTTP 响应，其中包含了处理结果

❺ 更新资源请求的结果

❻ 创建新资源请求的结果

这就是将多个 HTTP 请求合并为一个 HTTP POST 请求的过程。

问题描述

您想知道通过单一 POST 请求来批量处理多个 HTTP 请求是否适用于您的 Web 服务吗。

解决方案

避免将多个 HTTP 请求合并为一个 POST 请求。应当使用 11.13 节中的方案，设计针对特定应用的资源来处理批量请求，而不是合并请求。

问题讨论

下面的设计问题可能会促使您使用前面的合并方法。

一台服务器负责管理采购申请。客户端提供一个用户界面用于审批和展示开放的申请，每个列表有 10 个项目。用户检查每一项并通过选择控件（比如复选框）来接受或者拒绝一个申请。某些时候，用户还可以添加一段日志做进一步说明。在这些修改都完成后，单击“提交”按钮处理所列出的 10 条申请。然后客户端以同样的形式列出下面 10 条开放申请。

服务器将每个申请作为一个资源，并使用如下 HTTP 方法:

- PUT 用于更新每个申请的日志
- POST 和一个控制器资源用于拒绝申请
- PUT 还可用于接受一个申请

由于每次要处理最多 10 个申请，自然可以先将 10 个 HTTP 请求合并到一个请求里

再提交给服务器。这么做的理由之一就是减少网络延迟。客户端只需开启一个网络连接而不是 10 个。

然而，这种普遍使用的合并方法有以下缺点:

并发性

HTTP 提供了 Last-Modified 和 ETag 头来实现乐观并发检查。将多个 HTTP 请求合并为一个请求的批量操作会让并发检查困难重重，因为服务器可能需要对批次中的每个单独任务都进行检查。

211 **原子性**

HTTP 请求是原子性的。每个请求执行一个任务，如果出错，服务器也可以保证数据的原子性和一致性。混合多个任务于单一请求中的批量操作，尤其是其中还含有互相依赖的操作时，不利于 Web 服务保持原子性以及从错误中恢复。

可见性

通过一个 HTTP 请求合并多个操作后，中间服务（intermediaries）无法对批次之中所描述的动作做出应答。此外，典型的安全措施会检查请求并防范攻击，但它们也不大可能检查到批次中的可疑请求，这可能会导致拒绝服务攻击。

错误处理

处理和报告错误对于批量操作而言更加困难。一个批量操作的结果可能是“混合”的，其中既有成功的，又有失败的。

可伸缩性

一个使用批量操作的典型理由是觉得批量请求比单独执行每个请求要具有更好的可伸缩性。如果大多数批量请求都到达一个单独的服务器，这些请求会降低服务器的响应能力。比起不支持批量处理的应用程序，一个大量使用单独服务器处理批量请求的应用程序，在性能上的表现可能会更差。

当一个 HTTP 操作需要修改多个资源时，服务器就应该创建专门的方案，就像 10.8 节所介绍的。处理批量请求的通用方案会让 HTTP 的请求和响应变得完全不透明，并且无法保证一定实现具备更好性能和更低延迟的目标。如果不能进行很好地设计和实现，用 POST 合并 HTTP 请求还有可能让服务器遭到拒绝服务攻击。

Internet-Draft“HTTP Multipart Batched Request Format”（详见附录 A）试图规范化一种 multipart/http 媒体类型，用于在分段消息的每个部分里使用 HTTP 请求或响应。使用这样的媒体类型容易遇到前面所讨论的那些限制。

11.13 如何支持批量请求

问题描述

您想知道如何批量处理 HTTP 请求。

解决方案

212

当面临需要将多个 HTTP 请求合并到一个 POST 请求里的情况时，先重新分析用例。设计一个特定的控制器资源，让它来支持这个用例，不要将问题一般化为通过 POST 合并多个请求。

问题讨论

下面是 11.12 节中引入的采购申请例子的一个替代解决方案：

```
# 批准 10 个申请
POST /approvals HTTP/1.1 ❶
Host: www.example.org
Content-Type: application/xml;charset=UTF-8

<approvals> ❷
  <approval>
    <id>001</id>
    <status>approve</status>
  </approval>
  <approval>
    <id>002</id>
    <log>Missing bids</log>
  </approval>
  <approval>
    <id>003</id>
    <status>reject</status>
    <log>Exceeds budget limit</log>
  </approval>
  ...
</approvals>

# 响应
HTTP/1.1 303 See Other ❸
Location: http://www.example.org/reqs/2009/11?page=2
```

❶ 批准申请的请求

❷ 包含申请审批数据的表述

❸ 成功响应

前面的请求消息包含同 11.12 节中的批量请求相同的信息，区别在于，这里使用了一个专门的资源来批量提交审核，而不是将 HTTP 方法组合进单独的 POST 请求里。它的结果也是一个单独的响应。如果操作成功，服务器就把客户端重定向到下一个申请页面。如果失败，则可以返回一个错误信息。

```
# 批准 10 个申请
POST /approvals HTTP/1.1
Host: www.example.org
Content-Type: application/xml;charset=UTF-8

<approvals>
  <approval>
    <id>001</id>
    <status>approve</status>
  </approval>
  <approval>
    <id>002</id>
    <log>Missing bids</log>
  </approval>
  <approval>
    <id>003</id>
    <status>reject</status>
    <log>Exceeds budget limit</log>
  </approval>
  ...
</approvals>

# 批量响应
HTTP/1.1 400 Bad Request ❶
Date: Tue, 03 Nov 2009 06:44:39 GMT
Content-Type: application/xml;charset=UTF-8
Content-Language: en
Link: <http://www.example.org/help/approvalcodes.html>;rel="help"

<error>
  <message>Authorization type is missing for requisition ID 001.</message>
</error>
```

213

❶ 一个原子的错误响应

这样的方案比通用的 HTTP 合并要更易于实现。上述请求中，处理请求的资源使用了一个特有的 URI，因此请求是可见的。而返回代码只有一个，因此响应也是可见的。

11.14 如何支持事务

关于 RESTful Web 服务的常见问题之一就是如何处理事务。通常下列场合会涉及这个问题:

- 客户端通过服务器连续进行一系列的操作。客户端想要取消整个操作，并且撤销在先前步骤中对数据所做的修改。
- 客户端同一系列服务器依次进行交互，以此实现一个应用流程，并且客户端希望可以还原任意的结果状态变化，或者永久记录下这些状态变化。

事务经常被视为是 REST 和 HTTP 所缺少的特性。例如，如果 HTTP 支持事务，网上银行服务就可以让客户端在一个事务中完成从一个账户到另一个账户的转账，从而保证原子性。这样的实现还可以改善交互的可见性，因为每个 HTTP 请求的实现都可以不修改任何相关资源。然而，在 Web 服务这样的分布式和去中心化的环境里支持事务，将会增加服务器和客户端的耦合度。并且也使得应用协议变得有状态了，因而削弱了可伸缩性。 214

问题描述

您想知道如何支持事务。

解决方案

提供一个能够对数据做原子性修改的资源。将没有提交的状态视为应用状态，用 1.3 节介绍的方案来管理它们。如果要允许客户端撤销动作，使用 PUT, DELETE 或者 POST 来做补偿性修改。

问题讨论

事务模型存在多种类型。以下是最常见的两种:

- 短期原子事务，确保原子性、一致性、隔离性和持久性（即 ACID）。
- 长期事务，其中的修改会有一段持续时间，而应用程序可以继续运行或者进行补偿动作（compensating actions）。

例如，在数据库中创建一个新用户，这可能以原子事务的形式发生，而购买一本书或者预定一个旅行行程可能就是长期事务了。

对于 HTTP 而言，每个请求都提供了对原子性修改的控制域（sphere of control）。一个 HTTP 请求要么成功要么失败，在 HTTP 中并没有部分失败一说。要支持原子

性操作，可以向资源提交修改信息，让那个资源尝试使用乐观并发控制（详见第 10 章）来提交那些修改。服务器可以提供一些保障，比如原子性、一致性、隔离性和持久性，这可以通过将所提交的修改任务转交给一个事务型的后端数据存储来完成。

举例来说，10.8 节的转账范例中使用了一个资源在单独的请求中原子化地更新两个银行账户资源。

```
# 请求
POST /transfer;t=e6e3c89d4dfe7f3a818734a6237ccfc5 HTTP/1.1
Host: example.org
Content-Type: application/xml;charset=UTF-8

<transfer>
  <source>urn:example:org:account:1</source>
  <target>urn:example:org:account:2</target>
  <currency>USD</currency>
  <amount>100.00</amount>
  <note>Testing transfer</note>
</transfer>

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

<transfer xmlns:atom="http://www.w3.org/2005/Atom">
  <source>urn:example:org:account:1</source>
  <target>urn:example:org:account:2</target>
  <atom:link href="http://www.example.org/transactions/1" rel="self"/>
  <currency>USD</currency>
  <amount>100.00</amount>
  <note>Testing transfer</note>
</transfer>
```

215

只要客户端还在关注着，这就是一个原子性的活动。客户端不用了解服务器是如何实现原子性活动的。服务器用两种方法确保并发控制：一是检查条件头 `If-Unmodified-Since` 或 `If-Match`；二是将先决条件编码到 URI 中，就像前面的例子。

另一方面，在 HTTP 中模仿像两阶段提交这样的事务协议会使 HTTP 协议变成有状态的，这可能会削弱 Web 服务的可伸缩性。

可以通过链接来引入补偿动作。例如，服务器可以在线路（itinerary）资源中提供一个链接来取消一个旅行预定。

```
# 一条旅行线路
GET /bookings/XAA55Z HTTP/1.1
Host: www.example.org

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

<itinerary xmlns:atom="http://www.w3.org/2005/Atom">
  <locator-id>XAA55Z</locator-id>
  <atom:link rel="self" href="http://www.example.org/bookings/XAA55Z"/>
  <atom:link rel="http://www.example.org/rels/cancel"
  href="http://www.example.org/bookings/XAA55Z/cancel"/>

  <!-- details of the itinerary -->
...
</itinerary>
```

客户端可以使用这个具有 http://www.example.org/rels/cancel 关系的链接来取消预约，并进行必要的补偿操作，比如取消旅行的具体路段，扣除取消费用以及退款。

使用本节介绍的方案来管理事务可以降低客户端和服务器的耦合度，保持无状态的交互，并保证每个 HTTP 请求的原子性。

第 12 章 安全

217

安全这个术语是用来描述系统不同层面和部分中的不同事物的。举例来说，一个基于 Web 的应用程序会涉及用户访问资源。要保护这样一个系统需要做到以下几点:

- 保证只有通过身份验证的用户才能访问资源。
- 从信息被收集那刻起，一直到它被存储起来，后续展现给经授权的组织和用户，都要保证它的机密性和完整性。
- 防止未授权或恶意客户端滥用资源和数据。
- 维护保密性，遵守当地法律（其中会有很多安全方面的条款）。

没有能满足所有这些需求的通用解决方案，每个应用程序都需要把细致的分析作为架构和设计的一部分，旨在覆盖所有这些安全方面。

本章将讨论 RESTful Web 服务安全相关话题的一个子集，将其中一些常见问题映射到了已有的基于 HTTP 的标准和实践上，例如身份验证、授权、机密性和完整性。

12.1 节，“如何使用基本身份验证来验证客户端”
: 通过本节了解如何使用 HTTP 基本身份验证。

12.2 节，“如何使用摘要身份验证来验证客户端”
: 通过本节了解如何使用 HTTP 摘要身份验证。

12.3 节，“如何使用三方 OAuth”
: 通过本节了解如何使用三方 OAuth（Three-Legged OAuth）[注1]协议让用户授权客户端访问他们的资源。

注 1：“三方 OAuth”（Three-Legged OAuth），“三方”即指授权过程中的消费方、服务提供方和用户；有些情况下不需要用户参与，于是就有了“两方 OAuth”（Two-Legged OAuth）。

12.4 节，“如何使用两方 OAuth”
　　通过本节了解如何使用两方 OAuth（Two-Legged OAuth）协议来验证客户端。

12.5 节，“如何处理 URI 中的敏感信息”
　　通过本节了解如何预防编码在 URI 中的状态被篡改，如何保持状态的机密性。

218

12.6 节，“如何维护表述的机密性与完整性”
　　通过本节了解如何维护表述的机密性与完整性。

12.1　如何使用基本身份验证来验证客户端

基本身份验证涉及客户端交换的标识符，以及服务器用于验证请求的一个共享 secret。

问题描述

您想知道如何实现基本身份验证。

解决方案

在服务器端，当客户端提交请求，访问一个受保护的资源时，服务器会返回响应码 `401（Authorization Required）`以及一个 `WWW-Authenticate` 头。

```
WWW-Authenticate: Basic realm="Some name"
```

在客户端，将客户端标识符（例如，如果客户端代表用户发起请求，则此处是用户名）和共享 secret（例如一个密码）以`<identifier>:<secret>`的形式连接起来，然后计算出这段文本的 Base64 编码值，将结果包含在客户端请求的 `Authorization` 头中。

```
Authorization: Basic <Base64 编码后的值
```

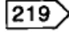
在服务器端，将文本解码，验证 secret 是否一致。

如果客户端事先知道服务器针对某个资源要求基本验证，它可以在每个请求里都带上 `Authorization` 头，以避免收到带 `WWW-Authorization` 头的 `401（Unauthorized）`响应。为方便起见，可以在服务器的文档中加入身份验证要求。

问题讨论

类似基本验证和摘要验证（详细见 12.2 节）这样的身份验证协议都使用了一种质询-应答（challenge-response）协议。当客户端访问受保护资源时，服务器使用 `WWW-Authenticate` 头来质询客户端，要求提供一个答案。对于基本验证和摘要验证，

该问题是“您是谁？”客户端使用 Authorization 头来提供答案，答案是密码的一个函数，或更笼统一点，是客户端和服务器所共享的 secret 的一个函数。



没有哪个 secret 是真正保密的，除非它是安全存储于客户端实现中的。

诸如基本身份验证和摘要身份验证这样的身份验证方案，可以用在两种场合中：客户端代表自己访问受保护资源，以及客户端代表用户访问受保护资源。

基本身份验证可以追溯到 HTTP 1.0，后来是 RFC 2617 来规定的。其中客户端用 Base64 编码共享 secret，并通过 Authorization 请求头将之提供出来。

Base64 编码是可逆的，当客户端没有用 TLS 来连接服务器时，不要使用基本身份验证。

下面是一个来自客户端的初始请求，试图访问一个要求身份验证的资源：

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

# 响应
401 Unauthorized
WWW-Authenticate: Basic realm="Photos App" ❷
Content-Type: application/xml;charset=UTF-8

<error xmlns:atom="http://www.w3.org/2005/Atom">
  <message>Unauthorized.</message>
</error>
```

❶ 一个不带身份信息的请求

❷ 一个带有质询的应答，要求使用基本身份验证提供身份信息

因为资源是受保护的，所以服务器质询客户端，要使用名为 Basic 的身份验证方案来提供它的身份信息。realm 值是一个字符串，标识服务器上的一块保护空间。

假设客户端/用户标识为 photoapp.001，共享 secret 是 basicauth。客户端计算出字符串 photoapp.001:basicauth 的 Base64 编码，用 Authorization 头发送下列请求：

```
# 请求
GET /photos HTTP/1.1
```

```
Host: www.example.org
Authorization: Basic cGhvdG9hcHAuMDAxOmJhc2ljYXV0aA== ❶

# 响应
HTTP/1.1 200 OK
Content-Type: application/xml;charset-UTF8

...
```

220

❶ 一个带身份信息的请求

服务器用 Base64 解码身份信息，检查客户端提供的共享 secret 是否匹配某个已知的值，以及是否允许客户端使用所提供的身份访问该资源。如果服务器收到一个不带 Authorization 头的请求，或者 Authorization 头中提供的身份信息不匹配，服务器会返回一个带 WWW-Authentication 响应头的错误。

既然验证的响应中可能包含敏感信息，要保证为该响应设置恰当的 Cache-Control 和 Expires 头。举例来说，如果该响应是特定于某个客户端的，使用 Cache-Control: private 来避免共享缓存为其他客户端存储或提供该响应。

```
# 请求
GET /users/admin HTTP/1.1
Host: www.example.org
Authorization: Basic cGhvdG9hcHAuMDAxOmJhc2ljYXV0aA==

# 响应
HTTP/1.1 200 OK
Cache-Control: max-age:3600,private
Vary: Authorization
Content-Type: application/xml;charset-UTF8

...
```

扩展 Authorization 头

Authorization 头是可扩展的。摘要身份验证（详见 12.2 节）和 OAuth（详见 12.3 节和 12.4 节）使用 Authorization 头来向服务器发送身份信息。此外，一些专门的身份验证技术也用它来提供身份信息。例如 Amazon 的 Simple Storage Service（S3）使用如下请求头来验证客户端：

```
Authorization: AWS AWSAccessKeyId:Signature
```

此处的 AWS 是 Amazon 使用的身份验证方案的标识符，AWSAccessKeyId 是 Amazon 分配给客户端的标识符，Signature 是请求方法、几个确定 HTTP 头和消息正文的

数字签名。客户端用 Amazon 生成并共享给该客户端的 secret 计算出签名。Amazon 用相同的数据和 secret 计算出签名，并与客户端提供的进行验证，以此来验证客户端的身份。

221

12.2 如何使用摘要身份验证来验证客户端

摘要身份验证（也是由 RFC 2617 来规定的）和基本身份验证很相似，不同之处在于客户端是发送身份信息的摘要给服务器。摘要身份验证还提供了防止重放攻击的机制。

问题描述

您想知道如何实现摘要身份验证。

解决方案

当客户端发送不带 `Authorization` 头的请求来访问受保护资源时，服务器返回响应码 `401 (Authorization Required)`，带有 `WWW-Authenticate` 头、`Digest` 身份验证方案，至少还有 `realm` 和 `nonce` 指令。*nonce* 是一个数字或者令牌，只能用一次或者有限次数。

在客户端，包含一个 `Authorization` 头，其中包括客户端或用户标识符的摘要、realm 和共享 secret。

将提供的摘要与服务器上存储的身份信息的摘要进行比对，包含一个 `Authentication-Info` 响应头，这相当于客户端的 `Authorization` 头。

默认情况下，客户端使用 MD5 来计算摘要。与基本身份验证不同的是，该技术不用交换未经加密的共享 secret。

问题讨论

下面是一个简单的服务器响应，其中有 nonce 和带 `Authorization` 头的客户端请求：

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

# 响应
401 Unauthorized
```

```
WWW-Authenticate: Digest realm="Sample app", nonce="6cf093043215da528d7b5039ed4694d3",
      qop="auth" ❷
Content-Type: application/xml;charset=UTF-8

<error xmlns:atom="http://www.w3.org/2005/Atom">
  <message>Unauthorized.</message>
</error>

# Request
GET /photos HTTP/1.1
Host: www.example.org
Authorization: Digest username="photoapp.001", realm="Sample app",
      nonce="6cf093043215da528d7b5039ed4694d3",
      uri="/photos", response="89fba5bf5e5f9dd69865258c21860956",
      cnonce="c019e396409afe784ae9f203b8dfdf7e", nc=00000001, qop="auth" ❸

# Response
HTTP/1.1 200 OK
Content-Type: application/xml;charset-UTF8

...
```

222

❶ 一个不带身份信息的请求

❷ 包含质询和 nonce 的应答

❸ 带 response 指令的请求，详见下文了解如何计算该值

与基本身份验证不同，摘要身份验证要求客户端提供身份信息的摘要。在第二步中，服务器包含以下指令：

realm

一个字符串，标识服务器上的一个受保护区域。

nonce

每个 401（Unauthorized）响应唯一生成的一个字符串，要求客户端在生成摘要时使用该值。nonce 指令晚于某些 nonce 值时，服务器可以拒绝包含此类 nonce 指令的请求，以此来防止重放攻击。

qop

对于这个指令，摘要身份验证定义了两个值：auth 和 auth-int。auth 的意思是服务器只在做客户端身份验证时使用摘要身份验证。auth-int 的意思是服务器还可以在维护请求完整性时使用该验证方案。当 qop=auth-int 时，客户端在计算摘要时要包含请求的正文。

本例中，共享 secret 是 digestauth 且客户端/用户标识为 photoapp.001，客户端使用这些信息加上请求方法、资源 URI 和 nonce 来计算摘要：

1. 将客户端/用户标识符、realm 和共享 secret 连接为<identifier>:<realm>:<secret>，计算其 MD5 值，将结果记为 A1。
2. 将请求方法和请求 URI 连接为<method>:<URI>,计算其 MD5 值,将结果记为 A2。
3. 将 A1，nonce 和 A2 连接为<A1>:<nonce>:<A2>，计算其 MD5 值。

客户端将计算的结果作为 Authorization 头中 respense 指令的值。

223

服务器提供的 nonce 是 Authorization 头中的一部分，因此客户端在没有事先通过 WWW-Authenticate 头来获得 nonce 值的情况下无法发送 Authorization 头。

使用一次性或有限次数的令牌作为 nonce，服务器可以限制重放攻击的发生。

与 10.9 节中用到的一次性令牌很相似，在请求中使用一次性或有限次数的令牌要求服务器维护一个所有已使用令牌的日志。

WWW-Authenticate 头中还可以包含其他指令，例如 domain，opaque，stale 和 algorithm 指令。对这些指令和实现细节的详细讨论超出了本书的范畴，可以参考 Chris Shiflett 的《*HTTP Developer's Handbook*》（Sams）了解摘要身份验证的更多细节。

12.3 如何使用三方 OAuth

OAuth（*http://oauth.net*）是 2007 年开发的一种委托身份验证协议。使用该协议，用户无须提供其身份信息，就可以让客户端去访问服务器上的个人数据。OAuth 的身份验证协议之所以被称为“三方”是因为协议中涉及服务提供方（即服务器）、OAuth 消费方（即客户端）和用户。

无论何时，当客户端想要访问指定用户在服务器上的资源时，OAuth 的三方协议都可适用。举例来说，Twitter，Yahoo!，Google，Netflix 等网站的用户可使用 OAuth 协议授权第三方工具访问他们的数据，这样一来这些工具就能在无须用户提供身份信息（用户名、密码之类的信息）的情况下访问他们的数据了。大多数编程语言都提供了该协议的实现。

问题描述

您想知道如何实现三方 OAuth 协议。

解决方案

图 12-1 展示了 OAuth 协议的角色。在协议开始时，服务器使用“消费方 key”作为客户端标识符，“消费方 secret”作为共享 secret。一旦用户授权客户端访问其资源后，服务器使用“访问令牌”作为授权客户端的标识符，“令牌 secret”作为共享 secret。

224

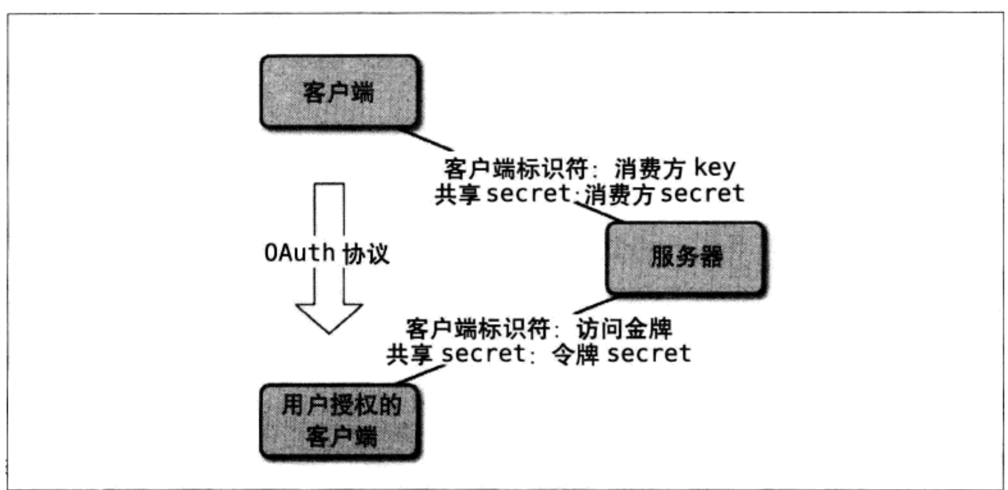


图 12-1：三方 OAuth 流的角色

OAuth 依赖服务器和客户端发布的三组令牌和 secret。

消费方 *key* 和消费方 *secret*

消费方 key 是客户端的唯一标识符。客户端使用消费方 secret 对请求进行签名获得请求令牌。

请求令牌和令牌 *secret*

请求令牌是服务器发布的一个临时的一次性标识符，用于询问用户是否授权给客户端。令牌 secret 用于签名请求以获得访问令牌。

访问令牌和令牌 *secret*

访问令牌是客户端用于访问用户资源的标识符。只要访问令牌有效，拥有令牌的客户端就能访问用户的资源。服务器可以随时取消令牌，可以是因为令牌到期，也可以是用户取消权限。secret 用于对访问受保护用户资源的请求进行签名。

使用三方 OAuth 涉及以下几个步骤，该流程的目的是获取访问令牌和 secret。服务器可以针对某一特定时段或限定访问特定用户资源来授予访问令牌。

1. 客户端提前向服务器发起请求，获取消费方 key 和消费方 secret。

2. 客户端使用消费方 key 获得请求令牌和 secret。

3. 客户端将用户引导至服务器进行授权，让客户端访问用户的资源。这个处理的结果会产生一个经授权的请求令牌。

225 4. 客户端请求服务器来提供访问令牌和 secret，它们代表了一个标识符和共享 secret，客户端用它们来代表用户访问资源。

5. 发起请求访问受保护资源时，客户端要带上一个 Authorization 头（或查询参数），其中包含消费方 key、访问令牌、签名方法和签名、时间戳、nonce 以及可选的 OAuth 协议版本。

请注意，OAuth 是一个基于 HTTP 之上的协议，服务器需要有文档向客户端说明下列 URI:

- 获取请求令牌的 URI
- 获取用户授权的 URI
- 获取访问令牌的 URI

OAuth 建议使用 POST 方式来获取请求和访问令牌。

问题讨论

考虑一下 11.1 节中介绍的相册 Web 服务。客户端需要用户的授权去复制用户的相册资源。客户端提前通过某些途径访问服务器，获得 oauth_consumer_key 和 secret。例如，服务器可以为客户端提供一个 Web 页面，注册并获取消费方 key 和消费方 secret。假定服务器分配的消费方 key 是 1191fd420e0164c2f9aeac32ed35d23，共享 secret 是 fd9b9d0f769c3bcc548496e4b5077da79c02d7be。

客户端初始化三方协议，服务器给出了如下 URI:

- 获取请求令牌的 URI（例如，https://www.example.org/oauth/ request_token）
- 获取用户授权的 URI（例如，https://www.example.org/oauth/ authorize）
- 获取访问令牌的 URI（例如，https://www.example.org/oauth/ access_token）

因为响应中涉及共享 secret 和用户授权，对这些 URI 应该考虑使用 TLS。

这些请求中涉及如下参数:

oauth_consumer_key
服务器发布给每个客户端的唯一标识符。

oauth_signature_method
: 计算签名时使用的签名方法。OAuth 将 HMAC-SHA1 和 RSA-SHA1 定义为签名方法。当客户端和服务器使用 TLS 时，您可以不用签名，把 PLAINTEXT 作为该参数的值。

226

oauth_timestamp
: 这是从 1970 年 1 月 1 日 00:00:00 GMT 开始的秒数。

oauth_nonce
: 这是一个随机字符串，在所有带相同 oauth_timestamp 的请求中，该字符串是唯一的。这个参数帮助服务器检测重放攻击。请注意，与摘要身份验证不同，OAuth 要求客户端来生成 nonce 值。

oauth_version
: 这是 OAuth 的版本，目前是 1.0。

客户端可以将这些参数作为 Authorization 头的指令来发送，它们也可以是查询参数，或者用 application/x-www-form-urlencoded 编码后放在请求正文里。在下面的例子中，所有请求都使用 Authorization 头。

第一步，客户端提交一个请求，从服务器获取请求令牌和 secret。本请求的签名是基于和消费方 key 一起获得的消费方 secret 来计算的。签名包括 oauth_consumer_key，oauth_signature_method，oauth_timestamp，oauth_nonce 和 oauth_version，必须按如下方式计算:

1. 收集 oauth_consumer_key，oauth_signature_method，oauth_timestamp，oauth_nonce 和 oauth_version 参数。

2. 百分号编码[注2]参数，先按照名称排序，然后再按值排序。

3. 将参数拼接为字符串，就用您计算 application/x-www-form- urlencoded 字符串的方式。在本例中，该字符串的值是 oauth_consumer_key= a1191fd420e0164c2f9aeac32ed35d23&oauth_nonce=109843dea839120a&oauth_signature_method=HMAC-SHA1&oauth_timestamp=1258308730&oauth_version=1.0。

4. 使用共享 secret 计算签名。在本例中，签名是 d8e19bb988110380a72f6ca33b2ba5903272fe1。

5. 对签名进行 Base64 编码，然后百分号编码结果文本。

消费方使用本签名发送一个请求去获取请求令牌。

```
# 获得请求令牌的请求
POST /request_token HTTP/1.1 ❶
Host: www.example.org
Authorization: OAuth realm="http://www.example.com/photos",
                     oauth_consumer_key=a1191fd420e0164c2f9aeac32ed35d23,
                     oauth_nonce=109843dea839120a,
                     oauth_signature=d8e19bb988110380a72f6ca33b2ba5903272fe1,
                     oauth_signature_method=HMAC-SHA1,
                     oauth_timestamp=1258308730,
                     oauth_version=1.0 ❷

Content-Length:0

# 包含请求令牌和 secret 的响应
HTTP/1.1 200 OK
Content-Type: application/x-www-form-urlencoded
oauth_token=0e713d524f290676de8aff4073b1bb52e37f065c
  &oauth_token_secret=394bc633d4c93f79aa0539fd554937760f05987c ❸
```

227

❶ 获取请求令牌和 secret 的请求

❷ 验证客户端的 Authorization 头

❸ 包含请求令牌和 secret 的响应

响应中的 oauth_token 就是请求令牌，客户端必须使用它才能获得用户的许可。客户端引导用户去访问服务器上的一个资源以获得授权。

注 2：百分号编码（percent encoding），也就是常说的 URL encoding。

```
# 获得授权的请求
GET /oauth/authorize?oauth_token=0e713d524f290676de8aff4073b1bb52e37f065c HTTP/1.1
Host: www.example.org
```

该资源的实现取决于服务器。这时，服务器需要检查用户是否通过了服务器的身份验证。服务器允许用户选择让客户端访问哪部分数据，以及访问的种类。举例来说，用户授权客户端编辑相册或创建新相册，但不能删除任意相册或照片。下一步，服务器将用户引导回客户端。如果客户端有基于 Web 的界面，服务器能通过回调 URI 将用户重定向到该界面。如果没有，服务器会要求用户手动在客户端的用户界面中输入验证码。无论哪种方式，客户端都可以获得来自服务器的验证码。

客户端使用验证码来获取访问令牌。该请求中的签名基于 oauth_consumer_key、请求令牌、验证码、oauth_signature_method、oauth_timestamp、oauth_nonce 和 oauth_version。

```
# 获得访问令牌的请求
POST /access_token HTTP/1.1 ❶
Host: www.example.org
Authorization: OAuth oauth_consumer_key="a1191fd420e0164c2f9aeac32ed35d23",
                     oauth_token="ad0d1c7a765c9e6e8b14e639c763177312d18e7e",
                     oauth_verifier="988786765423",
                     oauth_signature_method="RSA-SHA1",
                     oauth_signature="698d58fd3316304181e11c6eb8127ffea7e2df46",
                     oauth_timestamp="1258328458",
                     oauth_nonce="109843dea839120a",
                     oauth_version="1.0" ❷

Content-Length:0

HTTP/1.1 200 OK
Content-Type: application/x-www-form-urlencoded

oauth_token=8d743f1165c7030177040ec70f16df8bc6f415c7
  &oauth_token_secret=95aec3132c167ec2df818770dfbdbd0a8b2e105e ❸
```

228

❶ 获取访问令牌和 secret 的请求

❷ 验证客户端的 Authorization 头

❸ 包含访问令牌和令牌 secret 的响应

该响应中包含了访问令牌和 secret。在发布访问令牌前，服务器必须校验请求令牌和消费方 key 是否匹配。

客户端使用这些内容来构造 Authorization 头，用它发起请求为用户访问受保护资源，

该请求中的签名基于 oauth_consumer_key，access token，oauth_ signature_ method，oauth_timestamp，oauth_nonce 和 oauth_version；还包括 URI 中的所有查询参数，当请求媒体类型是 application/x-www-form-urlencoded 时，这部分则是请求正文。

```
# 请求
POST /albums/2009/08/1011/duplicate;t=a5d0e32ddff373df1b3351e53fc6ffb1 HTTP/1.1 ❶
Host: www.example.org
Authorization: OAuth oauth_consumer_key="a1191fd420e0164c2f9aeac32ed35d23",
                     oauth_token="827fa00c6f15db4063378bb988e1563e0c318dbc",
                     oauth_signature_method="RSA-SHA1",
                     oauth_signature="f863cceebb4f1fe60739b125128e7355dcbf14ea",
                     oauth_timestamp="1258328889",
                     oauth_nonce="3c93e7fdd1101e515997abf84116ef579dccce1a",
                     oauth_version="1.0" ❷
```

❶ 访问受保护资源的请求

❷ 包含使用了访问令牌和访问 secret 的 Authorization 头的请求

尽管这个流程看上去很复杂，但它的设计让客户端能在无须获得用户身份信息（例如用户名和密码）的情况下访问用户数据。此外，用户可以让服务器取消任意客户端的权限。

考虑 OAuth 时，请注意该协议并不与任何特定资源相关联。想更深入地讨论该协议的使用问题，请参考《Beginner's Guide to OAuth》（*http://hueniverse.com/oauth*）。

12.4 如何使用两方 OAuth

两方 OAuth 类似于这种情况——客户端使用基本或摘要身份验证，通过 Authorization 头向服务器提供它的身份信息，没有代理介入。

229 请注意，OAuth 规范中并没有规定这种身份验证方式，但它作为一种利用服务器对客户端进行验证的途径被广泛使用。

问题描述

您想知道如何实现两方 OAuth 验证客户端请求。

解决方案

两方 OAuth 涉及以下步骤：

1. 客户端事先请求服务器获取消费方 key 和消费方 secret，前者是客户端的标识符，

后者是客户端与服务器之间共享的 secret。

2. 发起请求访问受保护资源时，客户端在请求中带上一个 Authorization 头，其中包含消费方 key、签名方法和签名、时间戳、nonce 和可选的 OAuth 协议版本。

服务器在授权访问资源前要验证签名。因为在身份验证流程中只涉及两方，所以这种方法被称为两方 OAuth。

当服务器需要验证客户端提供访问控制、日志、统计、速率限制、度量等功能时，可以使用两方 OAuth。

问题讨论

两方 OAuth 非常适合用于有多个客户端访问服务器上的受保护资源的情况。服务器给每个客户端发布一个消费方 key 和 secret，要求客户端提交 Authorization 头，其中包含每次 OAuth 计算出的签名。两方 OAuth 还可以方便地支持那些原本就支持三方 OAuth 的服务器。

举例来说，考虑 5.5 节中介绍的雇佣流程范例。在这个例子中，客户端通过与服务器的交互来实现雇员的雇佣流程。客户端有自己的身份验证机制，在适当的地方验证最终用户。比方说，客户端可能是一个基于 Web 的应用程序，使用 Cookie 来验证用户。服务器要求客户端在每次请求时使用两方 OAuth 来进行验证。通过一些事先途径，服务器给客户端颁发了一个 oauth_consumer_key 和 secret，假设服务器分配的消费方 key 是 a1191fd420e0164c2f9aeac32ed35d23，共享 secret 是 fd9b9d0f769c3bcc548496e4b5077da79c02d7be。

假设客户端提交候选者信息创建一个新资源。要发起一个经过验证的请求，客户端需要带上一个 Authorization 头，12.3 节中讲述了计算签名的大概方法。

计算结果作为 oauth_signature 参数的值，客户端随后生成如下 Authorization 头，发起请求：

230

```
# 输入候选者信息的请求
POST /hires HTTP/1.1 ❶
Host: www.example.org
Authorization: OAuth realm="http://www.example.com/hires",
                     oauth_consumer_key=a1191fd420e0164c2f9aeac32ed35d23,
                     oauth_nonce=85a55859fde262ba,
                     oauth_signature=d8e19bb988110380a72f6dba33b2ba5903272fe1,
                     oauth_signature_method=HMAC-SHA1,
                     oauth_timestamp=1258308689,
                     oauth_version=1.0 ❷
```

```
Content-Type: application/json

{
  "name": "Joe Prospect",
  ...
}

# 响应
HTTP/1.1 201 Created
Location: http://www.example.org/hires/099
Content-Location: http://www.example.org/hires/099
Content-Type: application/json

{
  "name": "Joe Prospect",
  "id": "urn:example:hr:hiring:099",
  ...
  "link" :
{
    "rel" : "http://www.example.org/rels/hiring/post-ref-result",
    "href" : "http://www.example.org/hires/099/refs"
  }
}
```

❶ 访问受保护资源的请求

❷ 用消费方 key 和消费方 secret 计算的 Authorization 头

请注意，OAuth 还允许客户端通过查询参数来提供身份信息。

```
# 输入候选者信息的请求
POST /hires?oauth_consumer_key=a1191fd420e0164c2f9aeac32ed35d23&
            oauth_nonce=85a55859fde262ba&
            oauth_signature=d8e19bb988110380a72f6dba33b2ba5903272fe1&
            oauth_signature_method=HMAC-SHA1&
            oauth_timestamp=1258308689&oauth_version=1.0 HTTP/1.1
Host: www.example.org
Content-Type: application/json
{
  "name": "Joe Prospect",
  ...
}
```

可以使用 Authorization 头来避免 URI 过长。

231 如果客户端没有包含合法的 Authorization 头，服务器就应该返回 WWW-Authenticate 头和 401（Unauthorized）响应头。

```
# 输入候选者信息的请求
POST /hires HTTP/1.1
Host: www.example.org
Content-Type: application/json

{
  "name": "Joe Prospect",
  ...
}

# 响应
401 Unauthorized
WWW-Authenticate: OAuth realm="http://www.example.com/hires"
Content-Type: text/html;charset=UTF-8

<html>
  ...
  <body>
    <p>Unauthorized.</p>
  </body>
</html>
```

这说明服务器使用 OAuth 进行身份验证。

12.5 如何处理 URI 中的敏感信息

正如 1.3 节说的那样，服务器可以将应用程序状态编码进 URI 里。在某些情况下，这种状态可能是很敏感的。URI 在网络中传输时，使用 TLS 可以帮助保证此类状态的完整性，但服务器无法控制客户端是如何管理 URI 的。在此类情况下，服务器需要确保 URI 没有被篡改，并且 URI 中的信息仍然是保密的。

问题描述

您想知道如何保护包含在 URI 中的敏感数据的完整性或机密性。

解决方案

要检测篡改，可以使用类似 HMAC-SHA1 和 RSA-SHA1 这样的算法来计算 URI 中数据的数字签名，将签名作为查询参数包含在资源 URI 中。

如果 URI 中的数据是机密数据，可以用 AES，Blowfish，DES，Triple DES，Serpent 和 Twofish 等算法来进行加密，在将结果放入 URI 之前，保证对其做 Base64 编码。

问题讨论

以 1.3 节中介绍的保险报价为例，服务器将用于发布报价的数据编码在表述的一个链接里。客户端可以使用该链接购买基于该报价的保险。

```
# 请求
GET /quotegen?fname=...&lname=...&... HTTP/1.1
Host: www.example.org

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

<quote xmlns:atom="http://www.w3.org/2005/Atom">
  <driver>
    ...
  </driver>
  <vehicle>
    ...
  </vehicle>
  <offer>
    ...
    <valid-until>2009-08-02</valid-until>
    <atom:link href="http://www.example.org/quotes/buy?fname=...&lname=...&..."
          rel="http://www.example.org/quotes/buy"/>
  </offer>
</quote>
```

链接中，编码在 URI 里的状态很有可能被篡改。为了避免这种情况，服务器可以包含一个签名，该签名基于所有用于生成该报价的关键参数。

```
http://www.example.org/quotes/buy?fname=...&lname=...&...
     &sign=f5b244520c2a452a0ee8c8b6ab5b6828317d2f7f
```

本例中的签名在计算时使用了 HMAC-SHA1 算法和一个只有服务器知道的 secret。客户端发起请求时，服务器会用 URI 中的数据重新计算签名，将计算结果与 URI 中包含的签名相比较，只要有不同就说明数据被篡改了。

服务器也可以加密状态，使用加过密的状态。

```
http://www.example.org/quotes/buy?gZwEW9oJIlZhYa1CuJ9IshGyvYJp2Gfo99M5115
  hWRKk497mkAOrnBZhkSb18UBzYftLpnryxUT2Y0C8GFDpNT64hypV4kMu
```

当不能使用 TLS 时，服务器可以要求客户端将请求正文包含在签名里。这时，服务器需要为每个客户端分配一个标识符和共享 secret，用文档说明客户端生成签名的算法。举例来说，访问资源的 OAuth 请求中有一个签名，包含了请求正文中的全部

参数。qop=auth-int 的摘要身份验证也将请求正文作为摘要的一部分。两者都是为了保证请求的完整性。

12.6 如何维护表述的机密性与完整性

233

问题描述

您想知道如何维护表述的机密性和完整性。

解决方案

使用 TLS，同时配置服务器，只允许那些使用 HTTPS 的请求访问资源。

问题讨论

HTTP 是一个分层协议[注3]，它依赖于 TCP/IP 这样的传输协议来提供消息传输的可靠性。通过将 HTTP 置于 TLS（RFC 5246）协议（它是 SSL 的后继者）之上，无须在客户端和服务器代码中处理加密和摘要签名，就可以维护请求及响应消息的机密性和完整性。

TLS 还可用于相互身份验证，此时服务器和客户端可以相互确认身份。举例来说，可以用基本身份验证来验证用户身份，但依靠 TLS 来验证客户端和服务器。

当您使用 TLS 来确保机密性和完整性时，可以避免为了此类安全目的直接在请求和应答消息中构建协议。除此之外，TLS 是与消息无关的，能被用于任何媒体类型或请求中。

配置 TLS 的细节是特定于 Web 服务器和客户端软件的，甚至在不同的编程语言之间也可能有不同。可以参考服务器或软件的手册来了解如何在客户端和服务器之间配置 TLS。

请注意，基于 SOAP 的 Web 服务要依靠 WS-Security（*http://www.oasis-open.org/committees/ tc_home.php?wg_abbrev=wss*）。WS-Security 中规定了将安全令牌作为 SOAP 头来发送的方式。例如，要避免篡改，基于 SOAP 的 Web 服务要在 SOAP 消息头中加入一个签名，就像下面的例子一样：

```
# 一个包含签名的 SOAP 消息
```

234

注 3：HTTP 是应用层协议，这里作者说"HTTP is a layered protocol"应该是指它可以工作于其他协议之上。

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header>
   <wsse:Security
    soapenv:actor="http://www.example.org"
    soapenv:mustUnderstand="1"
    xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext">
    <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
      <SignedInfo>
        <CanonicalizationMethod
         Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
        <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
        <Reference URI="#abcd">
          <Transforms>
            <Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          </Transforms>
          <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
          <DigestValue>... digest value</DigestValue>
        </Reference>
      </SignedInfo>
      <SignatureValue>...</SignatureValue>
      <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
        ... 密钥信息 ...
      </KeyInfo>
     </Signature>
    </wsse:Security>
   </soapenv:Header>
   <soapenv:Body>... 应用程序数据 ...</soapenv:Body>
 </soapenv:Envelope>
```

该消息在 Body 元素中包含了应用程序数据，签名包含在 Header 中。相比之下，RESTful Web 服务使用 HTTPS（即 TLS 之上的 HTTP），让 TLS 来处理数字签名和加密，一切都独立于使用的方法和媒体类型之外。

HTTP 的分层架构将应用层消息从传输层安全中解耦出来了。

第 13 章

可扩展性与版本控制

235

在任何分布式的客户端/服务器环境中管理变更都是很困难的。在这样的环境中，客户端依赖于服务器能履行自己的承诺。RESTful Web 服务也不例外，对它们而言，这包括 URI、资源、结构体和表述的内容、它们的格式，以及每个资源的 HTTP 方法。

直到您开始考虑向后兼容性（backward compatibility）之后，对服务器的变更好像都成了有益的了。变化是向后兼容的，在修改服务器时就无须升级客户端。客户端会忽略您升级了服务器这一事实，继续按原有方式进行调用，就好像什么变化都没发生过一样，当然这不包括服务器升级期间的不可用时间。

还有一种兼容性被称为向前兼容性（forward compatibility），当有多个客户端和服务器在不同时间点升级时，这一点是很重要的。在这种情况下，较新的客户端可能会与老的服务器进行交互。向前兼容性的目的是保证较新的客户端能继续使用老的服务器，虽然功能有所减少，但不会出现问题。您的应用程序中仅需要考虑向后兼容性，还是需要同时考虑向后和向前兼容性取决于您的操作环境。本章描述的内容能帮助您处理好这两种兼容性。

让您能够保持兼容性的特性是可扩展性，这是为未来的变化预留余地的设计过程。HTTP 是可扩展的，可以通过增加带有特定警告的新方法或标头来扩展 HTTP（详见 1.12 节和 1.13 节），但这并不意味着构建于 HTTP 之上的应用程序是自动可扩展的。本章将向您展示一些步骤，通过这些步骤就能保持 Web 服务的可扩展性了。

处理变更需要规则、周密的计划和防御性编程实践。大多数变更的目的是好的，修复某些问题或增强某些功能，但如果没有好的计划，它们就会造成危害。

虽然维持兼容性是很值得做的，但却无法做到始终让变更都是可兼容的。举例来说，您可能增加了安全措施，需要客户端升级。在这种情况下，客户端别无选择。除此之外，您还可能为服务器添加了新特性，需要客户端进行修改后才能使用。这里的挑战是在允许新客户端使用服务器的新特性时，仍能让现有客户端像往常一样工作

236

一段时间。

就算您计划升级所有的客户端以便可以使用新服务器，要同一时间升级所有服务器和客户端也是不现实的事。您的 Web 服务可能要求不间断工作，同时升级可能会要求有一定停机时间。因此需要制定计划，逐步升级服务器和客户端，以保证整个系统的可用性。

请注意，无论客户端还是服务器都需要一些恰当的步骤，以便能在发生变更时平稳运转。服务器的目标是保证客户端不会出问题，客户端的目标是在新的未知数据或链接出现在表述中时不会失败。本章中讨论了如下内容:

13.1 节，“如何维持 URI 的兼容性”
　通过本节了解如何保持 URI 变更的兼容性。

13.2 节，“如何维持 XML 和 JSON 表述的兼容性”
　通过本节了解如何在维护兼容性的同时扩展 XML 和 JSON 表述。

13.3 节，“如何扩展 Atom”
　通过本节了解扩展 Atom 的方法。

13.4 节，“如何维护链接的兼容性”
　通过本节了解如何保持链接的兼容性。

13.5 节，“如何实现支持可扩展性的客户端”
　通过本节了解客户端实现相关的内容，让客户端在服务器发生可兼容变更时不出问题。

13.6 节，“何时需要版本化”
　通过本节的内容来决定何时需要版本化服务器。

13.7 节，“如何版本化 RESTful Web 服务”
　通过本节了解如何版本化 Web 服务。

13.1 如何维持 URI 的兼容性

问题描述

您想知道如何让 URI 的变更能兼容现有客户端。

解决方案

237

就如 4.4 节所描述的那样，保持 URI 永久有效。将包含相同查询参数但顺序不同的请求 URI 视为同一个 URI。客户端必须能在不考虑请求参数顺序的情况下得到同样的行为。

当向 URI 中添加了新参数时，要继续服务于现有参数，并将新参数视为可选。在改变查询参数的数据格式时，继续支持现有格式。如果这是不可行的，那就通过新的查询参数或 URI 来引入格式变更。默认情况下，除非并发操作或安全原因需要 URI 中的请求参数，否则它们都应该是可选的。

问题讨论

考虑如下 URI:

```
http://www.example.org/catalog?q=rest&pub=oreilly&y=2010
http://www.example.org/catalog?pub=oreilly&q=rest&y=2010
```

这两个 URI 在符号组合上有所不同，在协议级别上表示两个不同的资源。但是在服务器代码上将这两个 URI 视为等价的可以为客户端带来 URI 解析和构造上的灵活性。

下面的 URI 中对一个查询参数使用了不同的格式，它们是兼容的:

```
# 原始 URI 格式
http://www.example.org/catalog?q=fiction&pubdate=1230796800

# 新 URI 格式
http://www.example.org/catalog?q=fiction&pubdate=2009-01-01Z
```

第一个 URI 的 `pubdate` 参数使用 UNIX 时间，第二个 URI 的 `pubdate` 参数则支持 RFC-3339 兼容格式。服务器需要继续支持前者以保持对已有客户端的兼容性。如果无法支持两种格式，比方说因为服务器的库程序无法同时支持多种格式，那么可以为新格式增加一个新的参数。

```
http://www.example.org/catalog?q=fiction&pubdate=1230796800
http://www.example.org/catalog?q=fiction&published=2009-01-01Z
```

第二个 URI 用一个不同的名字来表示查询参数，这暗示第一个参数是可选的。设计 URI 和服务器代码时，如果能够保持查询参数可选，那么在引入新参数时能带来一定的灵活性。

13.2 如何维持 XML 和 JSON 表述的兼容性

本节讨论了如何对 XML 和 JSON 表述做出可兼容的变更。详见 13.3 节了解扩展 Atom 表述的相关内容。

问题描述

238

您想了解如何让 XML 和 JSON 表述的变更与现有客户端保持兼容。

解决方案

设计一个 XML 格式，保持子元素是无序的。在修改 XML 和 JSON 时，保持分层结构，这样客户端可以继续按照相同结构提取数据。

将请求中的新数据元素做成可选的，以此来保持和现有客户端的兼容性。那些不发送新数据域的客户端必须要能继续工作。

不要从响应正文的表述中删除或重命名任何数据域。

问题讨论

HTTP 标头是表述的一部分，只要客户端和服务器在每次 HTTP 请求时都正确地使用标头，那么它们应该不会影响兼容性。大多数的兼容性问题都发生在正文的表述中。

要允许服务器进行变更，就需要使用可扩展的格式，例如 XML 和 JSON。但是，使用这些格式并不能自动保证表述是可兼容的。要让客户端不出问题，需要保持客户端从表述正文中读取数据的方式。下面的例子描述了一个可兼容变更：

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

<user>
  <first-name>John</first-name>
  <last-name>Doe</last-name>
  <street>1 Some Street</street>
  <city>Some City</city>
<user>

# 变更后的表述正文
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

<user>
  <first-name>John</first-name>
```

```
  <last-name>Doe</last-name>
  <street>1 Some Street</street>
  <city>Some City</city>
  <state>WA</state>
<user>
```

239

但下面的表述就引入了一个不兼容的变更：

```
# 变更后的表述正文
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

<user>
  <first-name>John</first-name>
  <last-name>Doe</last-name>
  <address>
    <street>1 Some Street</street>
    <city>Some City</city>
  </address>
<user>
```

在这个表述中，客户端为了读取城市名称而需要解析的元素与之前不同。从 user 元素的 city 子元素中提取城市名称的客户端在第二个例子中无法找到城市名称，这就破坏了兼容性。

对于任意允许数据分层排列的表述格式，不要改变分层。

一些针对 XML 表述的 XML Schema 语言应用会限制可扩展性。例如，考虑下面的 XML Schema：

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="address">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="first-name" type="xs:string"/>
        <xs:element name="last-name" type="xs:string"/>
        <xs:element name="street" type="xs:string"/>
        <xs:element name="city" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

这段 Schema 防止到处添加新的子元素，只能添加在 city 元素后面。此外，它要求

客户端按照子元素的顺序来构造 XML。针对这个例子，下面的 Schema 是更好的选择，它使用 xs:all 让子元素保持无序状态。

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="address">
    <xs:complexType>
      <xs:all>
        <xs:element name="first-name" type="xs:string"/>
        <xs:element name="last-name" type="xs:string"/>
        <xs:element name="street" type="xs:string"/>
        <xs:element name="city" type="xs:string"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

240

对于无序子元素，还可以使用 xs:choice 或者替换组(substitution group)。在 RelaxNG 中，可以使用交错（interleave）模式来描述无序元素。

可以使用 Schema 来辅助文档编制，但运行时不要在 XML 文档上实施约束。

在 JSON 表述中，JSON 对象的属性顺序无关紧要，根据定义，属性是无序的。因此，下面两个表述是等价的：

```
{
  "first-name" : "John",
  "last-name" : "Doe",
  "street" : "1 Some Street",
   "city" : "Some City"
}

{
  "city" : "Some City"
  "last-name" : "Doe",
  "first-name" : "John",
  "street" : "1 Some Street",
}
```

请注意，当服务器向表述中添加新内容时，并不能保证客户端在发起 PUT 或 POST 请求时会把它们提交回服务器。

```
# 请求
GET /user/001 HTTP/1.1
Host: www.example.org
```

```
# 包含新的 email 属性的响应
HTTP/1.1 200 OK
Content-Type: application/json

{
  "first-name" : "John",
  "last-name" : "Doe",
  "street" : "1 Some Street",
  "city" : "Some City",
  "email" : "john.doe@example.org"
}

# 更新的请求
PUT /user/001 HTTP/1.1 241
Content-Type: application/json

{
   "first-name" : "John A.",
   "last-name" : "Doe",
   "street" : "1012 North 1st Street",
   "city" : "Some City"
}

# 响应
HTTP/1.1 204 No Content
```

241

不知道新内容的客户端不会在本地存储它们。在本例中，客户端不会在 PUT 请求中包含 email 属性。如果这会导致服务器以为用户没有电子邮件地址，可以引入一个包含 email 的新资源版本。13.7 节中可以找到相应的例子。

13.3 如何扩展 Atom

Atom 格式的设计让它可以支持未来的扩展，其中所有的元素都允许外来 XML 元素和属性。例如，在下面的代码片段中，对 atom:author 元素进行了扩展，包含了作者的电话号码：

```
<atom:author xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:name>John Author</atom:name>
  <atom:uri>http://www.example.org/authors/john-author</atom:uri>
  <atom:email>john.author@mail.example.org</atom:mail>
  <ex:phone xmlns:ex="http://www.example.org/ns">425-123-4567</ex:phone>
</atom:author>
```

这是一个合法的 atom:author 元素，懂得这个扩展的客户端能解释出作者的电话号码，不懂的客户端会忽略它。可以用下面的方式来扩展 Atom：

- 添加新的链接关系类型。例如，“Feed 分页和归档”扩展（RFC 5005），其中引入了 first，last，previous 和 next 链接关系类型。
- 在 atom:entry，atom:feed 和 atom:link 之类的 Atom 元素中添加新元素。例子包括“Atom 条目扩展”（RFC 4685），其中引入了新元素 in-reply-to 和 total；“Atom 内联扩展”（In-Lining Extensions for Atom），它扩展了 atom:link 元素来包含被链接资源的 atom:entry 或 atom:feed 文档。
- 使用内嵌在 atom:content 元素中的 XML 和其他文本内容。

本节回顾了扩展 Atom 的多种方法并给出首选方法。

242

问题描述

您想知道扩展 Atom 的可行方法。

解决方案

像 5.4 节中那样定义新的链接关系，向 atom:feed 和 atom:entry 中添加新的子元素或属性，只要这些扩展不会妨碍那些不知道该扩展的客户端或软件的正常功能就可以了。

当在 atom:content 元素中加入外来内容时，用 atom:summary 元素提供给人阅读的文本或 XHTML。

问题讨论

引入扩展的一个关键的考量点是它们对互操作性的影响。最好避免那些有可能降低互操作性的扩展。下面是 OpenSearch（*http://www.opensearch.org*）用的一个扩展：

```
<!-- 来自 http://www.opensearch.org/Specifications/OpenSearch/1.1 的例子-->
<atom:feed xmlns:atom="http://www.w3.org/2005/Atom"
      xmlns:opensearch="http://a9.com/-/spec/opensearch/1.1/">
  <opensearch:Query role="request" searchTerms="General Motors annual report"/> ❶
  <opensearch:Query role="related" searchTerms="GM"
                    title="General Motors stock symbol"/>
  <opensearch:Query role="related" searchTerms="automotive industry revenue"/>
  <opensearch:Query role="subset" searchTerms="General Motors annual report 2005"
  <opensearch:Query role="superset" searchTerms="General Motors"/>
  ...
</atom:feed>
```

❶ 通过向 atom:feed 添加可选的子元素来进行扩展

不知道 OpenSearch 扩展的客户端会忽略该扩展。下面的例子是另一个不会降低互操

作性的扩展，基于“Atom 内联扩展”的 Internet 草案。

```
<atom:feed xmlns:atom="http://www.w3.org/2005/Atom">
  ...
  <atom:link rel="http://www.example.org/rels/comments" ❶
     href="http://www.example.org/comments"> ❷
     <ae:inline xmlns:ae="http://purl.org/atom/ext/">
      <atom:feed>
         <!-- 一个完整的 feed -->
       </atom:feed>
     </ae:inline>
  </atom:link>
   ...
</atom:feed>
```

❶ 通过引入新链接关系类型来进行扩展

❷ 通过向 atom:link 添加可选子元素来进行扩展

这个例子中有两个扩展。其中使用扩展链接关系类型引入了一个没有定义在 Atom 中的新链接类型；还扩展了 atom:link 元素，将评论 Feed 包含在一个链接之中，该链接用于提供指向评论 Feed 的 URI。

243

上述两个例子扩展了 Atom 但没有影响互操作性。下面这个例子，大多数 Atom 的工具都不能处理，是 6.2 节中范例的一个变体。

```
<!-- 避免这么做 -->
<atom:entry xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:title>Johnny Web Sample Production Schedule</atom:title>
  <atom:id>urn:sked:1111</atom:id>
  <atom:updated>2011-11-11T11:11:11Z</atom:updated>
  <atom:author><name>J. W. Smith</name></atom:author>
  <atom:link rel="self" href="http://www.example.org/ps/1111"/>
  <atom:content type="text">
    Johnny Web Sample Production Schedule
  </atom:content>
  <ex:story-development xmlns:ex="http://www.example.org/ns/ps"> ❶
    <ex:days>5</ex:days>
    <ex:planned-start>2012-01-01</ex:planned-start>
  </ex:story-development>
  <ex:pencil-roughs xmlns:ex="http://www.example.org/ns/ps">
    <ex:days>2</ex:days>
    <ex:planned-start>2012-01-10</ex:planned-start>
  </ex:pencil-roughs>
  <ex:layouts-and-ink xmlns:ex="http://www.example.org/ns/ps">
    <ex:days>3</ex:days>
    <ex:planned-start>2012-01-15</ex:planned-start>
```

```
  </ex:layouts-and-ink>
</atom:entry>
```

❶ 要求向 atom:entry 或 atom:feed 元素中添加元素，这会降低互操作性

该表述中有几个用于产生数据的扩展元素。不知道该扩展的 Atom 工具看到的内容如下：

```
<atom:entry xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:title>Johnny Web Sample Production Schedule</atom:title>
  <atom:id>urn:sked:1111</atom:id>
  <atom:updated>2011-11-11T11:11:11Z</atom:updated>
  <atom:author><name>J. W. Smith</name></atom:author>
  <atom:ink rel="self" href="http://www.example.org/ps/1111"/>
  <atom:content type="text">
    Johnny Web Sample Production Schedule
  </atom:content>
</atom:entry>
```

可以提供 XHTML 内容或摘要，让这个表述对这些客户端更有意义一些。

```
<!-- 避免这么做 -->
<atom:entry xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:title>Johnny Web Sample Production Schedule</atom:title>
  <atom:id>urn:sked:1111</atom:id>
  <atom:updated>2011-11-11T11:11:11Z</atom:updated>
  <atom:author><name>J. W. Smith</name></atom:author>
  <atom:link rel="self" href="http://www.example.org/ps/1111"/>
  <atom:content type="xhtml">
    <html>
      <head>
        <title>Johnny Web Sample Production Schedule</title>
      </head>
      <body>
        <!-- HTML 格式的生产日程 -->
        ...
      </body>
    </html>
  </atom:content>
  <ex:story-development xmlns:ex="http://www.example.org/ns/ps">
    <ex:days>5</ex:days>
    <ex:planned-start>2012-01-01</ex:planned-start>
  </ex:story-development>
  <ex:pencil-roughs xmlns:ex="http://www.example.org/ns/ps">
    <ex:days>2</ex:days>
    <ex:planned-start>2012-01-10</ex:planned-start>
  </ex:pencil-roughs>
  <ex:layouts-and-ink xmlns:ex="http://www.example.org/ns/ps">
```

244

```
    <ex:days>3</ex:days>
    <ex:planned-start>2012-01-15</ex:planned-start>
  </ex:layouts-and-ink>
</atom:entry>
```

通常情况下，可以安全地被客户端忽略的扩展会提升互操作性，应该被优先考虑。

13.4 如何维持链接的兼容性

问题描述

您想知道如何让对链接的变更兼容现有的客户端。

解决方案

避免删除链接，不要改变链接中 rel 和 href 属性的值。在引入新资源时，使用链接为客户端提供新资源的 URI。

问题讨论

链接允许客户端将 URI 视为模糊资源标识符。客户端可以使用链接关系类型来判断应该使用哪个 URI。但是，客户端也可以将 URI 存储在数据库里，当服务器改变 href 的值时，客户端可能没有替换新的值。举例来说，下面的变更可能会破坏客户端：

245

```
<!-- 旧的链接 -->
<atom:link rel="edit" href="http://www.example.org/catalog?prodid=32543Y2009"/>

<-- 新的链接 -->
<atom:link rel="edit" href="http://www.example.org/catalog/2009/32543Y"/>
```

客户端可能在本地存储了链接的值，并继续使用老的 URI。当您需要改变 URI 时，可以使用服务器端 URI 重写规则继续保留老的 URI。4.4 节中说明了保持 URI 永久可用的好处。

改变链接关系类型的值同样会破坏客户端的功能，可以引入一个新的 rel 来代替改名。

```
<!-- 旧表述中的链接 -->
<atom:link rel="edit" href="http://www.example.org/catalog?prodid=32543Y2009"/>
```

```
<-- 新表述中的链接 -->
<atom:link rel="edit" href="http://www.example.org/catalog?prodid=32543Y2009"/>
<atom:link rel="http://www.example.org/rels/update"
  href="http://www.example.org/catalog?prodid=32543Y2009"/>
```

链接在保持 Web 服务可扩展的过程中也起着重要的作用。例如，如果产品编目现在要支持诸如注释、评论和照片之类的用户生成内容，服务器可以通过添加新链接来引入新功能。

```
# 请求
GET /catalog/2009/32543Y HTTP/1.1
Host: www.example.org

# 包含新链接的响应
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8

<product xmlns:atom="http://www.w3.org/2005/Atom">
  ...
  <atom:link rel="http://www.example.org/rels/ugc-comments"
    href="http://www.example.org/catalog/2009/32543/comments"/>
  <atom:link rel="http://www.example.org/rels/ugc-reviews"
    href="http://www.example.org/catalog/2009/32543/reviews"/>
  <atom:link rel="http://www.example.org/rels/ugc-photos"
    href="http://www.example.org/catalog/2009/32543/photos"/>
</product>
```

该表述中的新链接为用户生成的内容提供了一组 URI，以此来扩展表述。支持新链接关系的客户端便可以使用新功能了。

246

13.5 如何实现支持可扩展性的客户端

问题描述

您想知道如何实现客户端，当服务器发生可兼容的变更时，它不会发生故障。

解决方案

在解析表述的正文时，查找已知数据。以 XML 为例，根据名称而非位置来查找已知元素和属性。当发现无法识别的数据时，所实现的客户端不能发生故障。如果客户端可以在本地保存完整的表述，把所有东西都保存下来。

不要假设从服务器收到的表述是某个固定的媒体类型、字符编码、内容语言或内容编码。就像 3.2 节中描述的那样，从对应的 Content-*头中获取这些值，并做相应的

处理。

问题讨论

在编写客户端应用程序时要记住一条关键的规则——不要在与客户端功能无关的数据上犹豫不决。举个例子，在一个 XML 表述中，服务器可能会增强用户档案的表述，在其中包含用户的博客地址和电子邮件。

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

<person>
  <first-name>John</first-name>
  <last-name>Doe</last-name>
  <street>1 Some Street</street>
  <city>Some City</city>
<person>

# 修改后的响应
HTTP/1.1 200 OK
Content-Type: application/xml;charset=UTF-8
<person xmlns:atom="http://www.w3.org/2005/Atom">
  <first-name>John</first-name>
  <last-name>Doe</last-name>
  <atom:link href="http://blog.example.org/johndoe" rel="related"/>
  <email>...</email>
  <street>1 Some Street</street>
  <city>Some City</city>
<person>
```

在发现 person 元素的第二个子元素后的链接时，所实现的客户端不能发生任何故障。 247

13.6 何时需要版本化

问题描述

您想知道何时需要引入一个 Web 服务的新版本。

解决方案

当服务器无法维持兼容性时，可以考虑版本化。如果某些客户端所要求的行为或功能与其他客户端不同，也可以考虑版本化。

问题讨论

有些时候，版本化与维持兼容性相比是一个简单的任务。要维持兼容性，您不得不经常评估一个变更是否会破坏客户端，然后慎重地进行变更。有时对服务器代码做的小变更会破坏客户端。在现实中，版本化可能会引入新的问题。

- 客户端为一个 Web 服务存储的数据可能无法自动与来自同一 Web 服务的另一版本的数据一同工作。客户端不得不在迁移到新版本前处理好本地存储的数据。您可以通过可兼容的变更避免此事。
- 版本变更可能涉及新的业务规则和应用流，这要求客户端修改代码。
- 同时维护资源的多个版本并不轻松，您可能需要针对各个版本划分服务器、代码或数据存储。
- 当使用链接向客户端提供 URI 时，客户端可能会将它们存储在本地。在分配了新 URI 后，客户端将不得不升级那些 URI，为该资源保存的其他数据也要一并升级。

但是，也有例外。举例来说，可以合并组织中所有的客户数据存储，对一些向客户端提供客户数据的服务器而言，它们可以使用完全不同的内容域。类似的，一个照片分享网站现在想支持针对微博的视频，视频的资源定义不能被塞到照片的资源定义里去。

另一个例外发生在服务器要为每个客户端维护不同的特性集的时候。这在多租户或“软件即服务”（Software as a Service）的平台上是很常见的。例如，一台为雇主提供健康保险管理的服务器，它需要为每个雇主独立维护不同版本的服务器软件，每个版本可能包含特殊的定制内容。

248 详见 13.7 节以了解如何引入新版本。

13.7 如何版本化 RESTful Web 服务

当无法做出可兼容变更时，版本化一部分或全部资源，以便将变化与现有客户端隔离开。

问题描述

您想知道如何引入一个 Web 服务的新版本。

解决方案

当资源的行为或表述中包含的信息发生变化时，可以用新的 URI 来添加新资源。使用容易检测的模式，根据版本区分 URI，例如在子域名中的 v1 或 v2、路径段（Path Segment）或查询参数。

不要把这些版本当做是同一资源的带新资源类型的新表述形式。

问题讨论

版本化 RESTful Web 服务涉及使用新 URI 来版本化资源，这是因为 HTTP 规定了资源 URI 及其表述以外的所有内容。尽管可以添加自定义的 HTTP 方法和标头，就像 1.12 节与 1.13 节中所讨论的那样，但这种做法会削弱与其他客户端和服务器的互操作性。此时，可以选择版本化资源。

以下是一些使用了版本标识符的 URI:

```
http://www.example.org/v1/customer/1234
http://www.example.org/v2/customer/1234
http://www.example.org/customer/1234?version=v3
http://v4.example.org/customer/1234
```

在这些 URI 中，哪个最好是取决于您的软件和服务器的部署情况的。同一台服务器管理多个版本时，使用路径段或查询参数可能会比较方便。

考虑 13.2 节中的电子邮件范例，既然 email 是新添加的，并且客户端可以修改它，那么服务器可以用新的 URI 来引入一个新版本的 Person 资源。

```
# 请求
GET /v2/person/001 HTTP/1.1
Host: www.example.org

# 包含新的 email 属性的响应
HTTP/1.1 200 OK
Content-Type: application/json

{
  "first-name" : "John",
  "last-name" : "Doe",
  "street" : "1 Some Street",
  "city" : "Some City",
  "email" : "john.doe@example.org"
}

# 更新请求
PUT /v2/person/001 HTTP/1.1
```

249

```
Content-Type: application/json

{
  "first-name" : "John A.",
  "last-name" : "Doe",
  "street" : "1012 North 1st Street",
  "city" : "Some City",
  "email" : "john.doe@example.com
}

# 响应
HTTP/1.1 204 No Content
```

使用新版本的客户端会看到 email，并且可以修改它，使用老版本的就做不到这些。

当服务器通过这种方式引入现有资源的新版本时，客户端需要升级数据存储以支持新版资源中的新内容。举例来说，在数据存储中，客户端可能保存了如下数据：

```
# ID、名字、姓、URI, ...
user001    "John"    "Doe"    "http://www.example.org/user/001"    ...
user002    "Jane"    "Doe"    "http://www.example.org/user/002"    ...
user003    "Bob"     "Coder"  "http://www.example.org/user/003"    ...
...
```

客户端在数据库中保存 URI 的做法和书签有点类似。既然客户端可以将一组服务器用于开发环境，另一组用于生产环境，那么您也许可以只保存 URI 的路径部分，将域名作为可配置的部分。

当服务器引入这些资源的新版本时，客户端需要更新其数据库以便能指向新的 URI。

```
# 用户 ID、名字、姓、电子邮件、URI……
user001  "John"  "Doe"    -  "http://www.example.org/v2/user/001"
user002  "Jane"  "Doe"    -  "http://www.example.org/v2/user/002"
user003  "Bob"   "Coder"  -  "http://www.example.org/v2/user/003"
...
```

250 带有容易识别的版本标识符的 URI 可以帮助客户端进行 URI 迁移。举例来说，作为升级新版本过程中的一部分，客户端可以通过编写代码将数据库中所有的 `http://www.example.org/user/`替换为 `http://www.example.org/v2/user/`。URI 升级后，客户端就能获取新内容，更新存储的数据了。

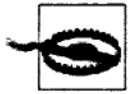
```
#用户 ID、名字、姓、电子邮件、URI……
user001  "John"  "Doe"    "john.doe@emample.org"    "http://www.example.org/v2/
user/001"
user002  "Jane"  "Doe"    "jane.doe@example.org"    "http://www.example.org/v2/
```

```
user/002"
user003  "Bob"   "Coder"  "bob.Coder@example.org"  "http://www.example.org/v2/
user/003"
...
```

请注意，有些服务器应用程序更倾向于用版本标识符来扩展媒体类型，而不是在 URI 中使用版本标识符，正如下面的例子：

```
application/xml;version=1
application/vnd.user+xml;version=1
application/vns.user+xml;version=2
```

这种做法的理念是将资源的各个版本视为不同的表述形式，这样一来，客户端可以通过发送带有某个版本的媒体类型的 Accept 头来获取指定的版本。如果服务器支持该版本，就会返回它的表述。在客户端升级支持新版本后，它只需修改 Accept 头中用到的媒体类型就能切换到新版本了。



避免为每个版本都引入新的媒体类型，这会导致媒体类型过多，可能会降低与其他服务器/客户端的互操作性，也会降低与现有 HTTP 层工具的互操作性。

PDG

第 14 章

251

服务发现

构建 RESTful Web 服务时，需要处理两类可发现性（discoverability），即设计时可发现性和运行时可发现性。设计时可发现性帮助他人设计并构建客户端，它描述了客户端开发团队和管理员需要知道的所有关于构建和运行客户端所需的知识。另一方面，运行时可发现性帮助保持客户端与服务器之间的松耦合，使得即插即用式的自动化成为可能。运行时可发现性涉及到 HTTP 的统一接口、媒体类型、链接和链接关系。本章讨论的是设计时可发现性。

简单来说，设计时可发现性就是在文档中描述您的 Web 服务，无论这个文档是用工具生成的，还是由 Web 服务的设计者或开发者手工编写的。客户端开发者可以参考该文档来理解资源的“语义”、媒体类型、链接关系等内容，并据此实现客户端。

本章主要讨论如下内容:

14.1 节，“如何编写 RESTful Web 服务的文档”
本节描述了如何编写文档，以便客户端开发者能了解您的 Web 服务。

14.2 节，“如何使用 OPTIONS”
通过本节了解何时及如何使用 `OPTIONS` 方法。

14.1 如何编写 RESTful Web 服务的文档

提升设计时与开发时的可发现性的有效方法是编写清晰的文档，提供实现客户端所需的信息。

252

问题描述

您想知道如何为您的 Web 服务编写文档。

解决方案

以容易阅读的方式在文档中详细描述以下内容:

- 所有资源以及每个资源支持的方法
- 资源在请求和响应中的媒体类型和表述形式
- 每个用到的链接关系、它的业务含义、要用的 HTTP 方法以及该链接标识的资源
- 所有没有通过链接提供的固定 URI
- 所有固定 URI 使用的查询参数
- URI 模板和标记置换规则（token substitution rule）
- 访问资源所需的身份验证和安全凭证

对于 XML 表述，如果您的客户端和服务器支持 XML Schema，可以将一种 Schema 语言作为“契约”，用来描述请求和响应表述中使用的 XML 文档结构。对于其他格式，在文档中用契约来描述表述。

供机器阅读的描述是无法代替供人类阅读的文档的。用容易阅读的格式（例如 HTML）为您的 Web 服务编写文档，这是在设计时发现服务最有用的方式。在编写服务文档时，要包含实现客户端所需的所有信息。

缺乏标准描述语言经常被说成是 REST 的一个局限。事实上，机器可读的描述语言无法为客户端开发者传递编码时所需的语义。例如，可以看看这些文档——Yahoo! Web 服务（*http://developer.yahoo.com*）、Flickr（*http://www.flickr.com/ services api/*）、Twitter （*http://apiwiki.twitter.com/*）和 Google Data Protocol （*http://code.google.com/apis/gdata/*）。所有这些服务都提供了大量便于阅读的文档，还附有范例。

下面是一个例子，展示了在编写 RESTful Web 服务的文档时要包含什么内容。参考一下 11.2 节中的相册范例，它支持查找、创建、编辑、复制和合并相册。表 14-1、表 14-2 和表 14-3 说明了如何为该 Web 服务编写文档。

表 14-1：资源

253

资源	方法	描述
照片	GET，PUT	该资源中包含照片的元数据与一个指向相关二进制照片媒体资源（例如一个 JPEG 文件）的链接。 媒体类型：*application/xml*
照片媒体	GET，PUT	该资源表示上传的照片。向相册资源提交带 multipart/form-data 媒体类型的 POST 请求，可以创建照片和照片媒体资源。 媒体类型：*image/jpeg*，*image/gif*，*image/png*
相册	GET，DELETE 和 POST	该资源中包含零或多张照片。使用 POST 可以往相册中添加照片。 媒体类型：*application/xml*，*multipart/form-data*
相册集合	GET，DELETE 和 POST	该资源中包含零或多个相册。使用 POST 可以向集合中添加相册。 媒体类型：*application/xml*
复制相册控制器	POST	该资源允许客户复制相册。 媒体类型：*application/xml*
合并相册控制器	GET 和 POST	该资源允许客户端合并相册。 媒体类型：*application/xml*

表 14-2：URI

资源	URI	描述
相册集合	http://www.example.org/albums	对于该 URI，使用 GET 可以获取最新的 10 个相册，使用 POST 可以创建一个新相册。 使用带 next 和 previous 关系类型的链接可以浏览全部相册
相册搜索	http://www.example.org/albums?q={keyword}&ym={year-month}	可以使用该 URI 模板来搜索相册。 {keyword}接受一个关键字，而{year-month}既可以接受年份也可以接受年和月。 下面是经过标记置换后的一些 URI 范例： http://www.example.org/albums?q=paris&ym= http://www.example.org/albums?q=hiking&ym=2009-08 http://www.example.org/albums?q=&ym=2000
相册合并	http://www.example.org/albums/merge?src={srcid1}&src={srcid2}	使用该模板得到一个链接，可以将两个相册合并成一个新相册。在执行合并操作后，服务器会删除原来的相册，该操作是不可撤销的。 在这个模板中，{srcid1}和{srcid2}是要合并的相册的标识符

表 14-3：链接关系类型

名称	描述
http://www.example.org/rels/duplicate	向带有这个关系类型的链接提交 POST 请求可以复制现有相册
http://www.example.org/rels/merge	使用带有这个关系类型的链接来访问控制器资源可以合并相册

254

简单起见，上述文档去掉了 XML 文档的细节和用到的表单参数。

14.2 如何使用 OPTIONS

通过实现 HTTP OPTIONS 方法，可以帮助工具了解 Web 服务中的资源。

问题描述

您想知道如何使用 OPTIONS 方法为客户端提供资源或者服务器信息。

解决方案

在服务器端，实现 OPTIONS，通过 Allow 响应头返回支持的方法。

当资源支持 PATCH 方法（11.9 节）时，添加一个 Accept-Patch 头为 PATCH 请求列出支持的媒体类型。

还可以添加一个 Link 头，其中包含一段易于阅读的文档来描述资源。

问题讨论

还是考虑相册的例子。针对范例中的每个资源，都可以提供一个链接，指向一篇易于阅读的文档，还有一个所支持方法的列表，帮助工具更多地了解您的资源。下面是一个例子：

```
# 请求
OPTIONS /photos HTTP/1.1
Host: http://www.example.org

# 响应
HTTP/1.1 204 No Content
Allow: POST, GET
Link: <http://www.example.org/docs/photos>; type=text/html; rel=help
```

在这个例子中，除了告诉客户端该资源支持 HTTP 的 POST 和 GET 方法以外，服务器

255

还提供了一个指向文档的链接，客户端开发者可以浏览该文档。可以为您喜欢的浏览器开发一个插件，当在浏览器中键入资源 URI 时自动显示关于这个资源的文档。

尽管可以在运行时用这个方法发现任何给定资源所支持的方法，但这么做的代价很大。在 HTTP 中，OPTIONS 方法是无法缓存的。举个例子，下面这一串请求为客户端应用程序引入了额外的延时：

```
# 请求
OPTIONS /photos HTTP/1.1
Host: http://www.example.org

# 响应
HTTP/1.1 204 No Content
Allow: POST, GET
Link: <http://www.example.org/docs/photos>; type=text/html; rel=help

# 提交一个 POST 请求来创建照片
POST /photos HTTP/1.1
Host: http://www.example.org
Content-Type: application/xml;charset=UTF-8

<photo>
  ...

</photo>

# 响应
HTTP/1.1 201 Created
Location: http://www.example.org/photo/4312
Content-Type: application/xml;charset=UTF-8

<photo>
  ...
</photo>
```

要使用开发时了解到的服务器和链接来发现 URI 并发起请求，而不是在运行时使用 OPTIONS。

附录 A

辅助读物 257

图书

Ross J. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems, 2nd Edition*. Indianapolis, IN: Wiley, 2008.[注1]

Rich Bowen, Ken Coar, *Apache Cookbook*. Sebastopol, CA: O'Reilly, 2007.[注2]

Bill Burke, *RESTful Java with JAX-RS*. Sebastopol, CA: O'Reilly, 2009.

Jon Flanders, *.NET*. Sebastopol, CA: O'Reilly, 2008.[注3]

David Gourley, Brian Totty, *HTTP: The Definitive Guide*. Sebastopol, CA: O'Reilly, 2002.

Emily P. Lewis, *Microformats Made Simple*. Berkeley, CA: New Riders, 2009.

Leonard Richardson、Sam Ruby, *RESTful Web Services*. Sebastopol, CA: O'Reilly, 2007.[注4]

Ivan Ristic, *Apache Security*. Sebastopol, CA: O'Reilly, 2005.

Sam Ruby、Dave Thomas、David Hansson, *Agile Web Development with Rails*. Raleigh,

注 1：第一版中文版《信息安全工程》，孙彦妍译，机械工业出版社，2003 年出版。

注 2：《Apache Cookbook 中文版》（第 2 版），蔡文凯译，电子工业出版社，2009 年出版。

注 3：这里估计作者指的是《RESTful .NET》。

注 4：《RESTful Web Services 中文版》，徐涵、李红军、胡伟译，电子工业出版社，2008 年出版。

NC: Pragmatic Bookshelf, 2009. [注5]

Chris Shiflett, *HTTP Developer's Handbook*. Indianapolis, IN: Sams, 2003.

Eric van der Vlist, *RELAX NG*. Sebastopol, CA: O'Reilly, 2003.

———, *XML Schema*. Sebastopol, CA: O'Reilly, 2002.

Duane Wessels, *Squid: The Definitive Guide*. Sebastopol, CA: O'Reilly, 2004. [注6]

———, *Web Caching*. Sebastopol, CA: O'Reilly, 2001.

258

参考文献

基础内容

Architectural Styles and the Design of Network-Based Software Architectures, Roy Fielding 的博士论文, *http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm* [注7]

HTTP Authentication: Basic and Digest Access Authentication, *http://tools.ietf.org/html/rfc2617*

HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV), *http://tools.ietf.org/html/rfc4918*

HTTP Multipart Batched Request Format, *http://tools.ietf.org/html/draft-snell-http-batch*

Hypertext Transfer Protocol: HTTP 1.0, *http://tools.ietf.org/html/rfc1945*

Hypertext Transfer Protocol: HTTP 1.1, *http://tools.ietf.org/html/rfc2616*

PATCH Method for HTTP, *http://tools.ietf.org/html/draft-dusseault-http-patch*

Uniform Resource Identifier (URI): Generic Syntax, *http://www.ietf.org/rfc/ rfc3986.txt*

注5: 此处是指原书第3版，中文版《Web开发敏捷之道——应用Rails进行敏捷Web开发》(第3版)，林芷薰译，电子工业出版社，2010年出版。

注6: 《Squid中文权威指南》，彭勇华译，免费发布于 http://home.arcor.de/pangj/squid/。

注7: 《架构风格与基于网络的软件架构设计》，李锟、廖志刚、刘丹、杨光译，免费发布于 http://www.redsaga.com/opendoc/REST_cn.pdf。

URI Template，*http://tools.ietf.org/html/draft-gregorio-uritemplate-03*

URN Syntax，*http://tools.ietf.org/html/rfc2141*

Web Linking Internet-Draft ， *http://tools.ietf.org/html/draft-nottingham-http-link-header*

Atom 与 AtomPub

The Atom Publishing Protocol，*http://tools.ietf.org/html/rfc5023*

The Atom Syndication Format，*http://tools.ietf.org/html/rfc4287*

Atom Threading Extensions，*http://tools.ietf.org/html/rfc4685*

Feed Paging and Archiving，*http://tools.ietf.org/html/rfc5005*

In-Lining Extensions for Atom，*http://tools.ietf.org/html/draft-mehta-atom-inline*

缓存

Caching Tutorial，*http://www.mnot.net/cache_docs/*

HTTP Cache Channels ， *http://ietfreport.isoc.org/idref/draft-nottingham-http-cache-channels*

The stale-if-error HTTP Cache-Control Extension ， *http://tools.ietf.org/html/draft-nottingham-http-stale-if-error*

The stale-while-revalidate HTTP Cache-Control Extension ， *http://tools.ietf.org/html/draft-nottingham-http-stale-while-revalidate-01*

259

格式与媒体类型

The application/json Media Type for JavaScript Object Notation (JSON) ， *http://tools.ietf.org/html/rfc4627*

BCP 47: Matching of Language Tags，*http://tools.ietf.org/html/bcp47*

Date and Time on the Internet: Timestamps，*http://tools.ietf.org/html/rfc3339*

English country names and code elements，*http://www.iso.org/iso/english_country_names_and_code_elements*

Extensible Markup Language (XML) 1.0（Fifth Edition），*http://www.w3.org/*

PDG

*TR/REC-xml/*HTML 4.01 Specification，*http://www.w3.org/TR/html401/*

Internet Assigned Numbers Authority (IANA) MIME Media Types，*http://www.iana.org/assignments/media-types/*

ISO 4217 currency names and code elements，*http://www.iso.org/iso/support/currency_codes_list-1.htm*

Olson Time Zone Database: Sources for Time Zone and Daylight Saving Time Data，*http://www.twinsun.com/tz/tz-link.htm*

RDFa in XHTML: Syntax and Processing，*http://www.w3.org/TR/rdfa-syntax/*

Tags for Identifying Languages，*http://tools.ietf.org/html/rfc5646*

XML Base（Second Edition），*http://www.w3.org/TR/xmlbase/*

XML Media Types，*http://tools.ietf.org/html/rfc3023*

XML Schema Part 2: Datatypes Second Edition，*http://www.w3.org/TR/xmlschema-2*

安全

Beginner's Guide to OAuth，*http://hueniverse.com/oauth/*

HTTP Authentication: Basic and Digest Access Authentication，*http://tools.ietf.org/html/rfc2617*

OAuth Core 1.0，*http://oauth.net/core/1.0a*

附录 B

REST 概述

261

REST 是表述性状态转移（Representational State Transfer）的简称。要明白它的含义，可以设想一个简单的基于 Web 的社交应用程序。

1. 用户在浏览器中键入地址，访问应用程序的主页。

2. 浏览器向服务器提交了一个 HTTP 请求。

3. 服务器返回一个 HTML 文档，其中包含了一些链接和表单。

4. 用户在表单中键入了她的状态，然后提交表单。

5. 浏览器向服务器提交另一个 HTTP 请求。

6. 服务器处理该请求，返回另一个页面。

在用户停止浏览前，这个过程不断地周而复始。除了一些特例，大多数网站和基于 Web 的应用程序都遵循相同的模式。让我们看看这个应用程序和 REST 有什么关系。

统一资源标识符

在上述的交互中，用户最开始键入浏览器的就是一个统一资源标识符（Uniform Resource Identifier，URI）。它的另一个常用的名字是统一资源定位器（Uniform Resource Locator，URL）。URI 更通用一些，可以用来表示一个地址（URL）或是一个名字。

URI 是资源的标识符，大多数情况下，URI 对客户端来说是模糊的（opaque）。

资源

资源是任意可以用 URI 来标识的东西。在上述流程的第一步中，用户键入的 URI

是一个资源的地址，它对应了一个 Web 页面。在传统的静态网站里，每个 Web 页面都是一个资源。

262 在第 4 步中，服务器更新用户状态的部分是另一个资源。用来提交的 HTML 表单有这个资源的地址（URI），这个地址被编码为 form 元素的 action 属性值。

表述

服务器返回给客户端的 HTML 文档是资源的一个表述。表述是资源信息（状态、数据或标记）的一种封装，使用诸如 XML，JSON 或 HTML 这样的格式进行编码。

资源可以有一个或多个表述。客户端和服务器使用媒体类型来标识给接收方（客户端或服务器）的表述类型。大多数网站和应用程序通常会使用 HTML 格式，将 text/html 作为媒体类型。类似的，用户提交表单时，浏览器用 URI 编码格式发送了一个表述，它用的媒体类型是 application/x-www-form-urlencoded。

统一接口

客户端使用 Hypertext Transfer Protocol（HTTP）来向资源提交请求并获得响应。在第 1 步中，客户端提交了一个 GET 请求来获取 HTML 文档。在第 4 步中，客户端提交了一个 POST 请求来更新用户状态。

这两个方法是 HTTP 统一接口中的一部分。使用统一接口让请求和响应，可以自描述并变成可见的。除了这两个方法，该接口还包含其他方法，例如 OPTIONS，HEAD，PUT，DELETE，TRACE 和 CONNECT。其中，除了 CONNECT 方法是 HTTP 1.1 保留下来用于穿隧（tunneling）基于 TCP 的协议（比如 TLS）的，HTTP 为每个方法都定义了语义。

HTTP 是介于客户端和资源之间的协议。在这个协议中，除非您定义了新方法来扩展 HTTP，否则方法的列表和它们的语义是固定的，那些语义是独立于资源的，这就是 HTTP 被称为统一接口的原因。这不同于远程调用（RPC）或基于 SOAP 的 Web 服务，其中每个请求的语义都是应用程序特有的。

超媒体和应用程序状态

最后，客户端从服务器收到的每个表述都反映了应用程序中用户交互的状态。举例来说，当用户提交了表单后接收到另一个页面，从她的观点来说用户改变了应用程

序的状态。当用户只是在浏览网站时，她每次点击链接，加载另一个页面的时候都是在改变应用程序状态。

在这个例子里，要改变应用程序的状态，用户需要依赖 HTML 中的表单和链接。HTML 是一种超媒体格式，允许用链接和表单控件穿梭于应用程序之中，改变它的状态。 263

这种使用表述的超媒体（例如 HTML）来表示和管理应用程序状态的方式被称为以超媒体作为应用程序状态引擎（hypermedia as the engine of application state），或者简称超文本约束（hypertext constraint）。

附录 C

265

HTTP 方法

HTTP 的统一接口包括 OPTIONS，GET，HEAD，POST，PUT，DELETE 和 TRACE 方法。本附录提供了一个入门，说明这些 HTTP 方法的使用方式，下文按照 RFC 2616 进行排序。

OPTIONS

使用该方法来获取资源支持的 HTTP 方法列表，或者 ping 服务器。

请求：只有标头没有正文。

响应：默认只有标头没有正文。服务器可以在正文里提供一段资源的描述。

范例：

```
# 1.获取资源所支持方法的请求
OPTIONS /movie/gone_with_the_wind HTTP/1.1
Host: www.example.org

# 带有资源所支持方法的响应
HTTP/1.1 204 No Content
Allow: HEAD, GET, OPTIONS, PUT, DELETE

# 2. ping 服务器或者获取所支持的 HTTP 版本的请求
OPTIONS * HTTP/1.1
Host: www.example.org

# 响应
HTTP/1.1 204 No Content
```

266

GET

使用该方法来获取资源的表述。

请求：按照 HTTP 1.1 的规定，只有标头，没有正文。

响应：所请求 URI 的资源的表述，通常带有一个正文。诸如 Content-Type，Content-Length，Content-Language，Last-Modified 和 ETag 之类的响应头要与响应中的表述一致。

范例：

```
# 获取一个资源表述的请求
GET /tx/1234 HTTP/1.1
Host: www.example.org

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

<status>
...
</status>
```

HEAD

使用该方法可以获取与 GET 响应相同的标头，但是响应中没有任何正文。换言之，这个方法会返回与 GET 相同的响应，只是服务器返回的正文是空的。客户端可以用它来检查资源是否存在，或者了解资源的元数据（详见 3.1 节）。

请求：按照 HTTP 1.1 规范，只有标头，没有正文。

响应：只有标头，没有正文。服务器必须不包括正文。

范例：

```
# 获取一个资源表述的请求
HEAD /movie/gone_with_the_wind HTTP/1.1
Host: www.example.org

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

POST

使用该方法让资源在服务器上执行一系列动作，例如创建新资源、更新已有资源、对一个或多个资源进行变更。

267 请求：一个资源的表述。

响应：一个资源的表述，或是一个重定向指令。如果正文中有一个表述，它对应的URI和请求URI不一致，包含一个带有该资源URI的Content-Location头。

范例：

```
# 1. 执行某个资源的特定动作
POST /admin/purge HTTP/1.1
Host: www.example.org

HTTP/1.1 204 No Content

# 2. 创建资源的请求
POST /user/smith HTTP/1.1
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/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>

# 3. 修改资源的请求
POST /user/smith/address_merge HTTP/1.1
Host: www.example.org
Content-Type: text/csv;charset=UTF-8

John Doe, 1 Main Street, Seattle, WA
```

```
Jane Doe, 100 North Street, Los Angeles, CA
...

# 响应
HTTP/1.1 303 See Other
Location: http://www.example.org/user/smith/address_book
Content-Type: text/html;charset=UTF-8

<html>
  <head> ... </head>
  <body>
    <p>See <a href="http://www.example.org/user/smith/address_book">address
    book</a> for the merged address book.</p>
  </body>
</html>
```

268

PUT

使用该方法来完整更新或替换一个现有资源，也可以用客户端指定的 URI 来创建一个新资源。

请求：一个资源的表述。请求的正文可以与客户端后续收到的 GET 请求一样，也可以不一样。在某些情况下，服务器可以要求客户端只包含资源的可变部分。

响应：响应可以是更新的状态。可以在响应中包含被更新资源的完整表述，但客户端不能假设响应中包含完整表述，除非响应有一个 Content-Location 头。如果服务器没有包含这个标头，客户端必须提交一个无条件 GET 请求来获取更新后的表述，带有 Last-Modified 和/或 ETag 头。

范例：

```
# 1. 更新资源的请求
PUT /movie/gone_with_the_wind HTTP/1.1
Host: www.example.org

# 响应
HTTP/1.1 204 No Content

# 2. 创建新资源的请求
PUT /movie/gone_with_the_wind HTTP/1.1
Host: www.example.org

# 响应
HTTP/1.1 201 Created
Location: http://www.example.org/movie/gone_with_the_wind
```

```
Content-Length:0
```

DELETE

使用该方法来删除资源。

请求：只有标头，没有正文。如果必须提交数据才能删除资源，像 2.6 节中说的那样，结合控制器资源来使用 POST 请求。

响应：成功或失败。正文中可以包含操作的状态。

范例：

```
# 删除资源的请求
DELETE /movie/gone_with_the_wind HTTP/1.1
Host: www.example.org

# 响应
HTTP/1.1 204 No Content
```

269

对客户端而言，资源在成功响应后就不复存在了。

TRACE

使用该方法回显服务器接收到的标头。支持该方法的服务器可能会存在跨站跟踪（cross-site tracing，XST）安全隐患。

请求：标头与正文。

响应：正文中包含整个请求消息。

范例：

```
# 请求
TRACE /movie/gone_with_the_wind HTTP/1.1
Host: www.example.org
Accept: text/html

# 响应
HTTP/1.1 200 OK
Content-Type: message/http

TRACE /movie/gone_with_the_wind HTTP/1.1
Host: www.example.org
Accept: text/html
```

附录 D

Atom Syndication Format

271

本附录提供了一个概述，介绍如何针对资源使用 Atom 的 Entry 和 Feed 文档。图 D-1 和图 D-2 是 atom:entry 和 atom:feed 的一个高层视图。如果需要这些元素的完整描述，详见 RFC 4287。

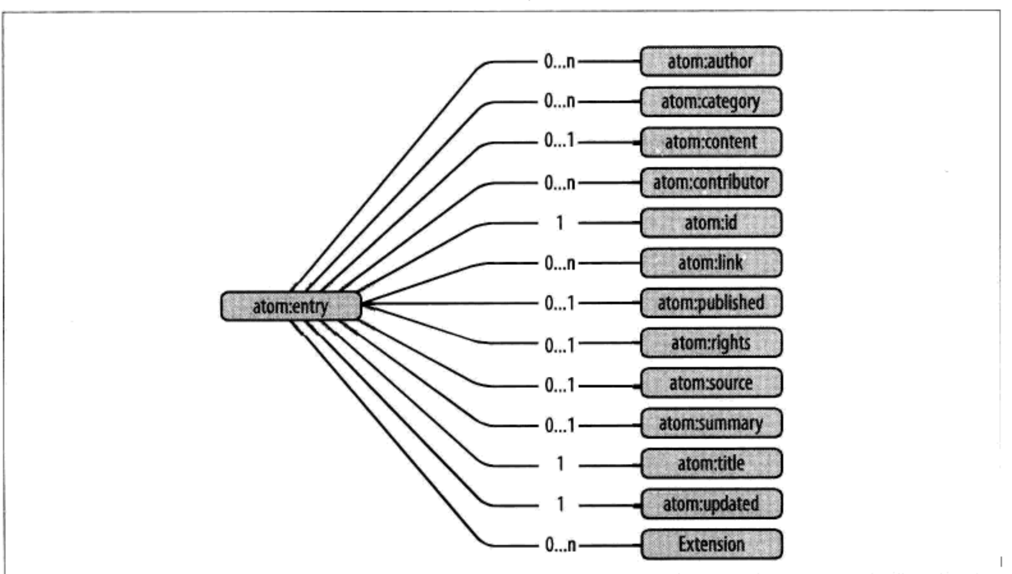


图 D-1： atom:entry

Feed 和 Entry 的关键元素

下面的列表是 atom:entry 和 atom:feed 中的关键元素。请注意，atom:entry 和 atom:feed 都是可扩展的，可以引入新的属性和元素。

272

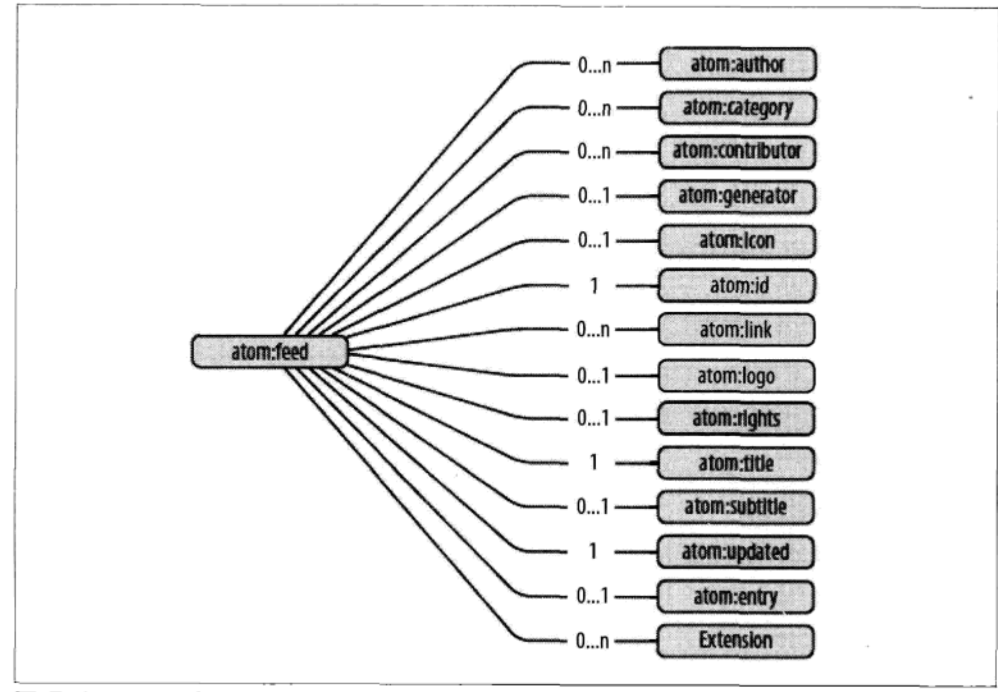


图 D-2: atom:feed

atom:author

位置: *atom:feed* 和 *atom:entry* 中

atom:author 元素表示创建了该 Entry 或 Feed 的"人"或"实体"。它有一些子元素，包括 atom:name，atom:uri 和 atom:email。单个 Atom Entry 或 Feed 可以包含多个 atom:author 元素。每个 atom:author 元素应该有至少一个 atom:name 元素。atom:uri 和 atom:email 元素是可选的。

273 下面是一个 atom:author 元素的范例:

```
<atom:author xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:name>J. P. Williams</atom:name>
  <atom:uri>http://www.example.org/jpwill</atom:uri>
  <atom:email>jpwill@example.org</atom:email>
</atom:author>
```

如果 Atom Feed 有一个 atom:author 元素，那其中的各个 Atom Entry 就不需要 atom:author 元素了。同样的，如果每个 Atom Entry 都有 atom:author，那么 Atom Feed 就不需要 atom:author 元素了。但是，始终在每个 Entry 和 Feed 中包含一个 atom:author 元素能简化处理它们的代码。

atom:content 与 atom:summary

位置：*atom:entry* 中

每个 Atom Entry 都应该有一个 atom:content 或 atom:summary 元素。

atom:summary 元素的作用是为 Entry 提供一个简短的摘要或描述。与 atom:title 元素类似，它支持一个 type 属性。

可以按照以下几种方式来使用 atom:content 元素：

- 以明文、HTML 或 XHTML 的格式包含 Entry 的内容
- 包含任意其他内容，需要带有一个媒体类型
- 使用 src 和 type 属性链接到任意资源

以下是有效 atom:content 元素的范例：

```
<!--内嵌明文-->
<atom:content xmlns:atom="http://www.w3.org/2005/Atom" type="text">
  This is some plain text
</atom:content>

<!-- 内嵌 HTML -->
<atom:content xmlns:atom="http://www.w3.org/2005/Atom" type="html">
  &lt;p&gt;This is some HTML.&lt;/p&gt;
</atom:content>

<!-- 内嵌 XHTML -->
<atom:content xmlns:atom="http://www.w3.org/2005/Atom" type="xhtml">
  <div>
    <p>This is some XHTML.</p>
  </div>
</atom:content>

<!-- 内嵌其他内容-->
<atom:content xmlns:atom="http://www.w3.org/2005/Atom" type="text/csv">
name,email
"John Doe","john@example.org"
"Jane Doe",jane@example.org"
</atom:content>

<!--外部内容 -->
<atom:content src="http://www.example.org/reports/2009.pdf" type="application/pdf"/>
```

274

atom:id

位置：*atom:entry* 中

atom:id 元素包含了 Entry 的全局唯一标识符，以 URN 形式的值作为其元素值（例如，urn:guid:550e8400-e29b-41d4-a716-446655440000）。就算 Entry 或 Feed 被更新或移动了，atom:id 的值也不能发生改变。客户端应该可以通过比较标识符来判断两个 Entry 或 Feed 是否相同。详见 3.10 节了解此类表述中的标识符的其他用途。

atom:link

位置：*atom:feed* 和 *atom:entry* 中

每个 Atom Entry 或 Feed 都可以包含多个 atom:link 元素。详见 5.1 节了解该元素的结构。RFC 4287 中详细描述了何时需要包含 atom:link 元素的规则，下面给出一个概要：

- 每个 Feed 和 Entry 必须包含一个 rel 是 self 的 atom:link 元素。
- 可以包含额外的 rel 值为 alternate 的 atom:link 元素，只要 type 和 hreflang 属性的组合是唯一的。详见 7.8 节。
- 还可以包含额外的 atom:link 元素链接到相关资源。

atom:title

位置：*atom:feed*，*atom:entry* 和 *atom:source* 中

这个元素包含了 Entry 或 Feed 标题的字符串表述(例如，<atom:title type="text">My Title</atom:title>）。title 元素支持 type 属性，属性值可以是 text，html 或 xhtml，默认是 type="text"。当 type 是 html 或 xhtml 时，必须为元素的值做转码，比方说<atom:title type="html">My Title </atom:title>。

atom:updated

位置：*atom:feed* 和 *atom:entry* 中

该元素包含了一个日期-时间值，这是 Entry 或 Feed 最后更新的时间。欲了解该格式，详见 3.9 节。

其他可以考虑的 Atom 元素

atom:category

位置：*atom:feed* 和 *atom:entry* 中

该元素的作用是分类 Feed 和 Entry。每个 Atom Entry 可以包含一个或多个 atom:category 元素。

```
<atom:category term="animal"
  scheme="http://example.org/categories/animal"
  label="Animal">Animal</atom:category>
```

atom:contributor

每个 Atom Entry 可以包含一个或多个 atom:contributor 元素。

```
<atom:contributor>
  <atom:name>E. Pound</atom:name>
  <atom:email>epound@example.org</atom:email>
  <atom:uri>http://epound.example.org</atom:uri>
</atom:contributor>
```

atom:generator

位置：*atom:feed* 和 *atom:source* 中

可以使用该元素来标明生成这个 Feed 或 Entry 源的软件。每个 Atom Entry 都可以包含一个 atom:generator 元素。

```
<atom:generator uri="http://www.example.org/generator/"
        version="1.0">Atom Generator 1.0</atom:generator>
```

atom:icon

位置：*atom:feed* 中

每个 Feed 都可以包含一个 atom:icon 元素。

```
<atom:icon uri="http://example.org/image/icon.png" />
```

atom:logo

位置：*atom:feed* 中

每个 Feed 都可以包含一个 atom:logo 元素。

```
<atom:logo
```

```
        uri="http://example.org/image/logo.png" />
```

276

atom:published

位置：*atom:entry* 中

每个 Atom Entry 都可以包含一个 atom:published 元素，它的值通常是这个 Entry 首次发布的日期-时间值。

```
<atom:published>2010-06-24T12:15:30Z</atom:published>
```

atom:rights

位置：*atom:entry* 中

每个 Atom Entry 可以包含一个 atom:rights 元素，用它来描述权利，例如版权。

```
<atom:rights type="text">*2010 All rights
        reserved.</atom:rights>
```

atom:subtitle

位置：*atom:feed* 和 *atom:source* 中

每个 atom:entry 或 entry:source 都可以包含一个 atom:subtitle 元素。

```
<atom:subtitle type="text">How I learned to
        love the Atom format</atom:subtitle>
```

附录 E

链接关系类型

277

该附录列举了在 *http://www.iana.org/assignments/link-relations/link-relations.xhtml* 中注册过的所有链接关系类型，以及何时使用这些链接关系类型。在 Web Linking Internet-Draft 定稿后，这个列表会做合并。

alternate

在提供同一资源的另一个版本的 URI 时使用这个类型。当资源有一个带不同 URI 的可选表述时，就可以使用有此类型的链接。

```
<!-- 链接到 PDF 格式的可选表述 -->
<atom:link rel="alternate"
      href="http://www.example.org/report.pdf" type="application/pdf"/>

<!-- 链接到法语的可选表述 -->
      href="http://www.example.org/report.fr.pdf" hreflang="fr"
      type="application/pdf"/>
```

appendix

当所链接的资源是作为一组资源的附录时，使用这个类型。在以内容为中心的应用程序中，带这个关系类型的链接非常有用。

```
<atom:link rel="appendix"
     href="http://www.example.org/books/restful-webservices-cookbook/appendix"/>
```

bookmark

WordPress 这样的博客平台使用这个链接关系类型在资源的摘要中创建永久链接。

```
<div id="p1">
  <h2>My First Post</h2>
```

278

```
    <p>Hello world. This is my first post.</p>

    <p><a href="/2009/10/1.html" rel="bookmark">Read more</a>.</p>
  </div>
```

chapter，section 和 subsection

这些关系类型可以用在指向资源集合的章、节和小节的链接上。

```
<book xmlns:atom="http://www.w3.org/2005/Atom">
  <title>RESTful Web Services Cookbook</title>
  <atom:link rel="chapter" title="Introduction"
        href="http://www.example.org/contents/ch01.xml"/>
  <atom:link rel="chapter" title="Using the Protocol"
        href="http://www.example.org/contents/ch02.xml"/>
  ...
</book>
```

contents

这个关系类型可以用于指向资源集合的目录。

```
<book xmlns:atom="http://www.w3.org/2005/Atom">
  <title>RESTful Web Services Cookbook</title>
  <atom:link rel="contents" title="Table of Contents"
        href="http://www.example.org/contents/toc.xml"/>
  ...
</book>
```

copyright

这个关系类型用于指向资源版权信息的链接。

```
<book xmlns:atom="http://www.w3.org/2005/Atom">
  <title>RESTful Web Services Cookbook</title>
  <atom:link rel="copyright" title="Copyright"
        href="http://www.example.org/contents/copyright.xml"/>
  ...
</book>
```

current

这个关系类型指向一个资源，其中包含集合中最近的一项或几项内容。下面是一个

例子，提供了一个指向最新文章的链接。

```
<atom:feed xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:title>Archives - 2008</atom:title>
  <atom:link rel="current" href="http://www.example.org/latest"/>
  <atom:link rel="self" href="http://www.example.org/2008/archive"/>
  ...
</atom:feed>
```

279

describedby

这个链接关系类型指向描述链接上下文的资源。举例来说，可以用它链接到一个文档，其中描述了某个用户的家乡。

```
<user xmlns:atom="http://www.w3.org/2005/Atom">
  <name>John Doe</name>
  <location>
    <latitude>47.45</latitude>
     <lingitude>122.30</logitude>
     <atom:link rel="describedby" href="http://www.example.org/places/Seattle"/>
  </location>
</user>
```

edit

使用这个关系链接到一个可以让客户端获取、更新或删除资源的 URI 上。

```
<user xmlns="http://www.w3.org/2005/Atom">
  <atom:link rel="edit"
        href="http://www.example.org/users/john.doe/profile"/>
  <name>John Doe</name>
   ...
</user>
```

当在一个非集合的资源中包含带有这个关系的链接时，通常意味着客户端可以使用该 URI 来获取（通过 GET）、更新（通过 PUT）或删除（通过 DELETE）资源。尽管通常都是让链接的 URI 与获取资源表述的请求 URI 保持一致，但有些情况下，服务器可能会选择为编辑提供一个单独的 URI。

存在带有这个关系的链接并不意味着可以编辑资源。在编辑前，客户端必须检查 HTTP OPTIONS 响应以及服务器的文档。

edit-media

这个关系类型是用于 Atom Entry 文档上的，这些文档有一个相关的媒体资源。例如，一篇附有视频资源的文章。客户端可以使用带有这个关系类型的链接来编辑相关媒体。

280

```
<atom:entry xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:title>Balloon Boy</atom:title>
  <atom:content type="text/html"
     src="http://news.example.org/balloon_boy.html"/>
   <atom:link rel="edit-media"
     href="http://media.example.org/balloon_boy.mov"
     type="video/quicktime"/>
  ...
</atom:entry>
```

enclosure

使用这个关系类型来指向比较大的相关资源。下面这个例子是一个视频播客预览链接，它指向了完整的播客内容。

```
Content-Type: video/quicktime
Link: <http://www.example.org/podcasts/what-is-rest.mov>;
       type="video/quicktime;length="124143";rel="enclosure"
...
```

first, last, next, next-archive, prev, previous, prev-archive, start

使用这些关系类型来提供用于浏览资源集合的链接。下面是一个范例:

```
<result xmlns="http://www.w3.org/2005/Atom">
  <atom:link rel="first" href="http://www.example.org/item/12321"/>
  <atom:link rel="last" href="http://www.example.org/item/6721"/>
  <atom:link rel="next" href="http://www.example.org/item/54674"/>
  ...
</result>
```

glossary

使用这个关系类型来链接到提供术语表的资源。

```
<book xmlns="http://www.w3.org/2005/Atom">
  <title>RESTful Web Services Cookbook</title>
  <atom:link rel="glossary" title="Glossary"
        href="http://www.example.org/contents/glossary.xml"/>
  ...
</book>
```

help

使用该类型来链接到提供与当前资源相关的信息或帮助的资源。

```
<div class="approval">
  <a href="/help.html" rel="help"/>
  <form ...>
  </form>
</div>
```

281

index

使用这个链接类型来链接到提供索引的资源。

```
<book xmlns="http://www.w3.org/2005/Atom">
  <title>RESTful Web Services Cookbook</title>
  <atom:link rel="glossary" title="Index"
        href="http://www.example.org/contents/index.xml"/>
  ...
</book>
```

license

可以使用带这个关系类型的链接来链接到资源（例如文章、绘画等）的许可上。

```
Content-Type: video/quicktime
Link: <http://creativecommons.org/licenses/by-nd/3.0/us/>;rel="license"
...
```

payment

使用这个关系类型链接到提供购买或引导支付的资源上。下面是一个买书的范例:

```
<book xmlns="http://www.w3.org/2005/Atom">
  <title>RESTful Web Services Cookbook</title>
  <atom:link rel="payment" title="Buy this book"
```

```
        href="http://my.safaribooksonline.com/9780596809140"/>
  <atom:link rel="chapter" title="Using the Protocol"
        href="http://www.example.org/contents/ch02.xml"/>
  ...
</book>
```

related

使用这个关系类型来链接到相关资源上。

```
<book xmlns="http://www.w3.org/2005/Atom">
  <title>RESTful Web Services Cookbook</title>
  <atom:link rel="related" title="RESTful Web Services"
        href="http://my.safaribooksonline.com/9780596529260"/>
  ...
</book>
```

282 这个关系是通用的，简单说来，这个链接的 URI 上的资源与包含该链接的资源有关系。如果倾向于更特定的关系，使用扩展链接关系，比如针对照片资源的 http://www.example.org/rels/photos/owner，用来指向它的拥有者资源；针对用户资源的 http://www.example.org/rels/friend，用来指向好友资源。

replies

使用这个链接关系类型来链接到反馈链接上下文的资源上。在以内容为中心的系统或者管理用户生成内容的服务器中，这个关系类型的链接之间可能是互相关联的。

```
<article xmlns="http://www.w3.org/2005/Atom">
  <title>State of the State</title>
  <atom:link rel="replies" title="Comments"
        href="http://www.example.org/ch01/comments"/>
  ...
</article>
```

self

使用这个类型来链接到资源的最佳 URI 上。下面是一个例子:

```
<user xmlns:atom="http://www.w3.org/2005/Atom">
  <name>John Doe</name>
  <atom:link rel="self" href="http://www.example.org/users/0012"/>
</user>
```

service

带有这个关系类型的链接指向 Atom Feed 的服务文档。

```
<atom:feed xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:title>Sci-Fi Books</title>
  <atom:link href="http://www.example.org/books" rel="self"
     hreflang="en" type="application/atom+xml"/>
  <atom:link href="http://www.example.org/service" rel="service"
     hreflang="en" type="application/atomsvc+xml"/>
  <atom:updated>2013-12-13T18:30:02Z</atom:updated>
  ...
</atom:feed>
```

stylesheet

这是文档中常用的链接关系，用于链接到样式表。

```
<html>
  <head>
    <title>Hello World</title>
    <link rel="stylesheet" type="text/css" href="/style.css" />
  <head>
  <body>
   ...
  </body>
</html>
```

283

up

客户端可以使用带有这个关系类型的链接导航到资源的上一级。

下面是一个范例:

```
<place xmlns:atom="http://www.w3.org/2005/Atom">
  <name>Seattle </name>
  <atom:link rel="self" href="http://www.example.org/us/wa/seattle"/>
  <atom:link rel="up" href="http://www.example.org/us/wa"/>
</user>
```

via

使用这个类型来标识作为信息源的资源。在这里的范例中，Atom Entry 的来源指向

了另一个资源：

```
<atom:entry xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:title>...</atom:title>
  <atom:link rel="alternate" type="text/html"
     href="http://www.subbu.org/archives/2009/10/15/announce.html"/>
  <atom:id>urn:example:10001</atom:id>
  <atom:link rel="via" type="text/html"
     href="http://www.example.org/blog/213"
     title="Jeffrey Veen"/>
</atom:entry>
```

PDG

索引✦

Symbols

& (ampersand), URIs, 76
, (comma), URIs, 76
- (hyphen), URIs, 76
. (dot) character, URI extensions, 134
. (full stop), URIs, 78
/ (forward-slash separator), URIs, 76
_ (underscore), URIs, 76

A

absolute URIs, 90
abstraction versus visibility, 6
Accept-* headers, 124
Accept-Charset headers, 129
Accept-Language headers, 128
ad hoc queries, SQL and XPath, 139
address book example, 114
address correction process example, 207
Age headers, 150
agent-driven content negotiation, 133
alternate link relation type, 93, 277
Amazon S3, Authorization headers, 220
ampersand (&), URIs, 76
annotating representations using entity headers, 46–49
appendix link relation type, 277
application flow
content versus server-driven negotiation, 135
  links, 95–99
application protocol (see HTTP)
application state
  defined, 262
  HTTP, 7
asynchronous tasks
  DELETE, 23
  POST, 19–22
Atom Syndication Format, 107–116, 271–276
  content model, 111–116
  extensibility, 241–244
  link relation types, 94
  modeling resources, 108–111
atom:author element, 272
atom:category element, 275
atom:content element, 273
atom:contributor element, 275
atom:generator element, 275
atom:icon element, 275

✦ 索引部分给出的页码为英文原版图书中的页码，请参照正文中用“▭▷”标出的原书页码。

atom:id element, 274
atom:link element, 274
atom:logo element, 275
atom:published element, 276
atom:subtitle element, 276
atom:summary element, 273
atom:title element, 274
atom:updated element, 274
atomicity, tunneling, 211
AtomPub (Atom Publishing Protocol), 116–122
  about, 107
  category documents, 116
  feed and entry resources, 118
  media resources, 119–122
  media type, 109
  service documents, 116–117
authentication
  clients, 218–223
  OAuth, 223–231
Authorization headers, 220, 226
auto insurance application example, 7

B

backward compatibility, 235
bank transfer application example, 176
Base64 encoding, security, 219
batch HTTP requests, 211
binary data, representations, 66
bookmark link relation type, 277
bookstore example, 41
bulk operations, resources, 203–208

C

Cache-Control headers, 148–151
caching, 147–158
  composite resources, 154
  content negotiation, 136
  expiration caching headers, 148–154
  keeping caches up-to-date and nonempty, 156
  POST method, 143, 144
  proxy HTTP caches, 31
  queries, 144
caching proxy servers, 163
capital letters, URIs, 79
category documents, AtomPub, 116
chapter link relation type, 278
character encoding negotiation, 129
characters, encoding mismatch, 50
charset parameter, 51
clients
  agent-driven negotiation, 134
  application state, 7
  authentication, 218–223
  client preferences and content negotiation, 124
  conditional GET and HEAD requests, 165
  conditional PUT and DELETE requests from clients, 174
  convenience versus visibility, 6
  decoupling from application state, 95
  expiration caching headers, 153
  extensions and interoperability, 244
  HTTP error codes, 71
  links in, 103
  media types and HTTP clients, 132
  safety and idempotency, 12
  supporting extensibility, 246

unconditional GET requests, 172
URIs as *opaque identifiers*, 81
collections
representations of, 59–62
resources, 32
comma (,), URIs, 76
compatibility
forward and backward, 235
links, 244
URIs, 236
composite resources
caching, 154
combining into, 34–37
compression, content negotiation, 130
concurrency control
DELETE requests, 174
PUT requests, 167, 174
types of, 159
concurrency, tunneling, 210
conditional requests, 159–181
conditional DELETE requests in servers, 171
conditional GET and HEAD requests from clients, 165
conditional POST requests, 176
conditional PUT and DELETE requests from clients, 174
conditional PUT requests in servers, 167–170
GET requests in servers, 162–165
Last-Modified and ETag Headers, 161
one-time URLs for POST requests, 179
unconditional GET requests from clients, 172
conneg (see content negotiation)
content encoding, 130
content model, Atom, 111–116
content negotiation, 123–136
agent-driven content negotiation, 133
character encoding negotiation, 129
client preferences, 124
compression, 130
expensiveness of implementing, 135
language negotiation, 127
media type negotiation, 126
negotiation failures, 132
server-driven negotiation, 135
Vary headers, 131
Content-Encoding headers, 49
Content-Language headers, 48
Content-Length headers, 48
Content-MD5 headers, 48
contents link relation type, 278
contextual links, 97
controllers, operating on resources, 39–43
"Cool URIs don't change", 83
COPY WebDAV method, 189
copying resources, 184
copyright link relation type, 278
country formats, 63
CRUD-style (Create, Read, Update, Delete)applications, 30
currency formats, 64
current link relation type, 278

D

data formats, portable data formats in representations, 62–64
data, binary data in representations, 66

databases
mapping tables, 32
storing application state, 8
timestamps, 161
URIs, 249
date and time formats, 64
DELETE method, 11, 23, 268
DELETE requests
conditional DELETE requests from clients,174
conditional DELETE requests in servers,171
denial-of-service attacks, 204, 211
describedby link relation type, 279
design-time discoverability, 251–255
digest authentication, clients, 221
discovery, 251–255
documenting RESTful web services, 251
OPTIONS, 254
documenting RESTful web services, 251
documents, 45
(see also representations)
category documents and AtomPub, 116
entry and feed documents in Atom, 108
service documents in AtomPub, 116-117
XML documents and schemas, 240
domain nouns, identifying resources, 30
domains, URIs, 76
dot (.) character, 134

E

edit link relation type, 93, 279
edit-media link relation type, 279
efficiency, 142
(see also performance)
composite resources and caching, 155
network latency, 6
queries and caching, 142
resource granularity, 32
elements
Atom feed and entry methods, 272
link elements, 87, 88, 90
email example example, 248
employee hiring process example, 104
enclosure link relation type, 280
entity headers
annotating representations, 46–49
interpreting, 49
entity identifiers, representations, 65
entry documents, Atom, 108
entry elements (Atom), 272
entry resources, AtomPub, 118
ephemeral URIs, links, 99
errors
handling and tunneling, 211
representation errors in clients, 73
returning in representations, 69–73
ETag headers
generating, 161
sending on requests, 169
expiration caching headers, 148–154
expiration caching, optimizing, 150
Expires headers, 148
extensibility, 235–247
Atom, 241–244
Authorization headers, 220
client support, 246
compatibility of links, 244
compatibility of XML and JSON representations, 237–241

defined, 235
HTTP status codes, 74
interoperability, 244
link relation types in Atom Syndication Format, 94
URI compatibility, 236
URI extensions, 134
WebDAV methods, 189, 190

F

failures, negotiation failures, 132
feed documents, Atom, 108
feed elements (Atom), 272
feed resources, AtomPub, 118
Fielding, Roy, REST, ix
filesystems, storing application state, 8
first link relation type, 93, 280
formats
JSON, 58
for media types, 55
portable data formats in representations, 62–64
representation formats, 52–56
forward compatibility, 235
forward proxy caches, 154
forward-slash separator (/), URIs, 76
freshness lifetime, defined, 148
full stop (.), URIs, 78
functions, supporting processing functions, 37
(see also methods)

G

GET method, 13, 142, 266
GET requests
combining resources into composites, 35
conditional, 160
conditional GET requests from clients, 165
conditional GET requests in servers, 162–165
unconditional GET requests from clients, 172
glossary link relation type, 280
granularity, resources, 31

H

hcard microformat, 69
HEAD method, 266
HEAD requests
conditional HEAD requests from clients, 165
limitations of, 175
headers
Accept-* headers, 124
Accept-Charset headers, 129
Accept-Language headers, 128
Age headers, 150
Authorization headers, 220, 226
Cache-Control header, 148–151
custom HTTP headers, 25–27
entity headers annotating representations, 46–49
ETag headers, 161, 169
expiration caching headers, 148–154
Expires headers, 148
interpreting entity headers, 49
keeping interactions visible, 2
Last-Modified headers, 161, 169
link headers, 91
representations, 47

Slug headers, 119
Vary headers and content negotiation, 131
help link relation type, 280
hiring process example, 229
homogeneous collections, 61
HTML representations, serving, 67
HTTP, 1-27, 23
(see also methods)
application state, 7
batch HTTP requests, 211
caching headers, 151
custom HTTP headers, 25-27
custom methods, 23
DELETE, 23
GET, 13
media types and HTTP clients, 132
optimistic concurrency control, 160
POST, 14, 16, 19-22
PUT, 18
safety and idempotency, 9-13
security, 233
visibility of interactions, 2-6
hypermedia, defined, 262
hyphen (-), URIs, 76

I

IANA (Internet Assigned Numbers Authority), 52
idempotency
and safety in clients, 12
and safety on servers, 9-12
identifiers, entity identifiers
in representations, 65
(see also URIs)
identifying resources from domain nouns, 30
image-processing web service example, 20
index link relation type, 281
interface (see HTTP)
Internet Assigned Numbers Authority (IANA), 52
interoperability
Atom, 108
extensions, 244
interpreting entity headers, 49

J

JSON representations
compatibility of XML and, 237-241
designing, 58
links, 90

L

language formats, 64
language negotiation
content versus server driven negotiation, 136
implementing, 127
last link relation type, 93, 280
Last-Modified headers
about, 49
generating, 161
sending on requests, 169
latency
network cost, 36
network latency, 6
query requests with large inputs, 143
unconditional GET requests, 174

web caching, 147
length, URIs, 142
license link relation type, 281
link elements
about, 87
JSON, 90
XML, 88
link relation types, 277–283
links, 87–105
application flow, 95–99
in clients, 103
compatibility, 244
contextual links, 97
ephemeral URIs, 99
headers, 91
links in JSON representations, 90
relation types, 93
self links, 56
storing application state, 9
URI templates, 101
XML representations, 88
localization, language negotiation, 127
LOCK WebDAV method, 190

M

mapping
database tables or object models, 32
operations to methods, 42
max-age Cache-Control directive, 149
media resources, AtomPub, 119–122
media type negotiation, 126
media types
Atom and AtomPub, 109
HTTP clients, 132
list of, 53
multipart media types, 66
representation formats, 52–56
versioning, 250
memcached, 147
merging resources, 186
metadata, representation metadata, 46
methods, 265–269
(see also conditional requests; GET requests; HEAD requests; POST method; POST requests; requests)
custom methods in HTTP, 23
DELETE method, 11, 23, 268
GET method, 13, 142, 266
HEAD method, 266
mapping operations to, 42
OPTIONS method, 254, 265
PATCH method, 201
PUT method, 18, 193, 268
safety and idempotency, 10
TRACE method, 269
WebDAV methods, 189
microformats, 68
mimicking transaction protocols, 215
MKCOL WebDAV method, 189
modeling
Atom content model, 111–116
mapping object models, 32
modeling resources using Atom, 108–111
MOVE WebDAV method, 190
moving resources, 188
multipart media types, 66
must-revalidate Cache-Control directive,149

N

negative caching, 152

negotiation (see content negotiation)
networks
collections of resources in social networks, 33
efficiency versus visibility, 6
next link relation type, 93
next-archive link relation type, 280
no-cache Cache-Control directive, 149, 151
no-store Cache-Control directive, 149, 151
nonce directive, 222
nonces, defined, 180
number formats, 63

O

OAuth, 223~231
oauth_consumer_key OAuth parameter, 225
oauth_nonce OAuth parameter, 226
oauth_signature_method OAuth parameter, 226
oauth_timestamp OAuth parameter, 226
oauth_version OAuth parameter, 226
object models, mapping, 32
opaque resource identifiers, 79–83, 244
OpenSearch, 242
operations, mapping to methods, 42
optimistic concurrency control, 159
OPTIONS method, 254, 265
origin servers, defined, 147
Oslo Time Zone Database, 64

P

pagination: latency and POST, 143
parameters
OAuth, 225
q parameters (Accept-* headers), 125
query parameters, 134
URIs, 237
PATCH method, 201
payment link relation type, 281
performance, 147
(see also efficiency)
ad hoc queries, 139
conditional GET requests, 159
encoding application state in links, 9
memcached and HTTP caches, 147
treating URIs as opaque, 83
unconditional GET requests, 174
pessimistic concurrency control, 159
photo album service example, 185, 225, 252, 254
photo management service example, 30
photo-sharing service example, 81, 156
portable data formats, representations, 62–64
POST method
about, 266
asynchronous tasks, 19–22
controllers to operate on resources, 39
creating resources, 16
queries, 142, 144
tunneling multiple requests, 208–211
using, 14
POST requests
conditional POST requests, 176
one-time URLs for POST requests, 179
predefined queries, 139
preferences, client preferences and content negotiation, 124
prev link relation type, 280
prev-archive link relation type, 280

previous link relation type, 93, 280
private Cache-Control directive, 149
processing functions, supporting, 37
production schedule example, 112
programmatic caching, 154
PROPFIND WebDAV method, 189
PROPPATCH WebDAV method, 189
protocol (see HTTP)
proxy HTTP caches, 31
proxy-revalidate Cache-Control directive, 149
public Cache-Control directive, 149
PUT method, 18, 193, 268
PUT requests
  conditional PUT requests from clients, 174
  conditional PUT requests in servers, 167–170

Q

q parameters (Accept-* headers), 125
qop directive, 222
queries, 137–145
  requests with large inputs, 142
  responses, 140
  storing, 144
  URIs, 138
query parameters, 134

R

range requests, queries, 140
RDFa, 68
realm directive, 222
redirects, disabling, 84
related link relation type, 93, 281
relation types, links, 93
relative URIs, 90
replies link relation type, 282
Representational State Transfer(see REST)
representations, 45–74
  (see also documents)
  annotating with entity headers, 46–49
  binary data, 66
  character encoding mismatch, 50
  collections, 59–62
  compatibility of XML and JSON representations, 237–241
  defined, 262
  entity identifiers, 65
  errors in clients, 73
  formats and media type, 52–56
  HTML representations, 67
  interpreting entity headers, 49
  JSON representations, 58
  links in JSON representations, 90
  links in XML representations, 88
  portable data formats, 62–64
  returning errors, 69–73
  security, 233
  XML representations, 56, 252
request tokens, 226
requests, 159
  (see also conditional requests; GET requests; HEAD requests; methods; POST method; POST requests)
  batch HTTP requests, 211
  range requests, 140
  tunneling multiple requests using POST, 208–211
resources, 29–43
  caching for composite resources, 154

collections, 32
composites, 34–37
computing/processing functions, 37
controllers, 39–43
copying, 184
creating using POST, 16
creating using PUT, 18
defined, 261
feed and entry resources, and AtomPub, 118
granularity, 31
identifying from domain nouns, 30
media resources and AtomPub, 119–122
merging, 186
modeling using Atom, 108–111
moving, 188
partial updates, 198–200
processing in bulk, 203–206
snapshots, 193–196
undoing updates, 196
responses, queries, 140
REST (Representational State Transfer), 261
about, ix
machine-readable description language, 252
returning representation errors in HTTP, 69–73

S

s-maxage Cache-Control directive, 149
S3 (Simple Storage Service), Authorization headers, 220
safety
and idempotency in clients, 12
and idempotency on servers, 9–12
methods, 10
scalability
maintaining application state, 8
mimicking transaction protocols, 215
POST method, 144
transactions, 214
tunneling, 211
schemas, XML representations, 240, 252
section link relation type, 278
security, 217–234
authentication, 218–223
denial-of-service attacks, 204, 211
OAuth, 223–231
representations, 233
URIs, 231
security tokens, 99
self link relation type, 93, 282
self links, XML representations, 56
semi-opaque URIs, 102
servers
conditional DELETE requests, 171
conditional GET requests, 162–165
conditional PUT requests, 167–170
HTTP error codes, 72
maintaining application state, 7
origin servers, 147
overloading, 204
safety and idempotency, 9–12, 10
server-driven negotiation, 135
supporting operations across servers, 191
wiki-like server example, 169
service documents, Atom, 116–117
service link relation type, 282
Simple Storage Service (S3),

Authorization headers, 220
Slug headers, 18, 119
snapshots, resources, 193–196
social application example, 261
social networks, collections of resources, 33
spaces, URIs, 78
SQL, ad hoc queries, 139
Squid, 147, 156
start link relation type, 280
states, application state, 7, 95, 262
storing
application state, 8
queries, 144
stylesheet link relation type, 282
subdomains
language-specific representations, 134
URIs, 76
submitting
conditional GET and HEAD requests from clients, 165
conditional PUT and DELETE requests from clients, 174
subsection link relation type, 278

T

tables, mapping, 32
tasks
asynchronous deletion, 23
asynchronous tasks, 19–22
templates, URI templates, 101
three-legged OAuth, 223
time and date formats, 64
time zone identifiers, 64
timestamps, 161
TLS (Transport Layer Security), 233
tokens
candidate IDs, 96
conditional POST requests, 176
as hidden form fields, 181
OAuth, 224
one-time URIs, 179
optimistic concurrency control, 159
request tokens, 226
security tokens, 99
URI templates, 101
TRACE method, 269
Traffic Server, 147
transactions, supporting, 213
tunneling
multiple requests using POST, 208–211
and protocol-level visibility, 43
repeated state changes, 79
two-legged OAuth, 228
types
link relation types, 93, 277–283
media type negotiation, 126
media types, 53
multipart media types, 66
of representations, 47

U

underscore (_), URIs, 76
undoing resource updates, 196
uniform interface (see HTTP)
defined, 262
visibility, 205
UNLOCK WebDAV method, 190
up link relation type, 283
updates
PATCH method for partial updates, 201

refining resources for partial updates, 198–200
undoing resource updates, 196
URIs (Uniform Resource Identifiers), 75–85
absolute URIs, 90
agent-driven negotiation, 134
as opaque identifiers, 244
compatibility, 236
"Cool URIs don't change", 83
creating new resources with POST, 19
in databases, 249
defined, 261
designing, 75–79
ephemeral URIs, 99
file or media type extensions, 134
length, 142
link relation types, 93, 95
maintaining application state, 7
one-time URLs for POST requests, 179
as opaque identifiers, 79–83
opaque resource identifiers, 244
parameters, 237
as parameters for merging, 187
queries, 138
relative URIs, 90
security, 231
syntax, 39
tampering, 180
templates, 101
URNs (Uniform Resource Names), 65
use cases, generalizing from, 183
UTF-8 encoding, 129

V

variants (see documents; representations)
Vary headers, content negotiation, 131
vcard format, 69
versioning
about, 247–250
ETag values, 162
value of, 236
via link relation type, 283
visibility
of interactions in HTTP, 2–6
tunneling, 43, 211
uniform interface, 205

W

web caching (see caching)
WebDAV methods, 189
wiki example, 169, 193, 196
WordPress blogging platform, HTTP headers, 26

X

X-HTTP-Method-Override, 27
XML
Atom, 108, 113
compatibility of XML and JSON representations, 237–241, 237–241
XML representations
designing, 56
links, 88
schemas, 240, 252
XML-RPC tunneling example, 16
xml:base attribute, 90
Xpath, ad hoc queries, 139

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。

[General Information]
书名=RESTful Web Services Cookbook 中文版
作者=（美）沙布·阿拉马拉尤著
页码=299
ISBN=299
SS号=12879413
dxNumber=000008172502
出版时间=2011.09
出版社=该引擎未能查询到
定价：59.00
试读地址=http://book.szdnet.org.cn/bookDetail.jsp?dxNumber=000008172502&d=FA168BA5A64053772718FB1D503555FF&fenlei=181704080104&sw=RESTful+Web+Services+Cookbook
全文地址=http://Wqa.5read.com/image/ss2jpg.dll?did=b1&pid=E27A8AE0ACD44FB4E7DF5E96D7AD79E512F8263D25DCE55BA8BE4C04CF519381DA9B5F31301DE92307CB1A63E6B0BB5976406853C66352F0705A7EDFE3AE80E4887EF9137C6B62148698C9EA966ABD96BBBCA3814B805CB8D44680D971766B413A59449EEB1C78F4242AE70B8CCD7911FEEA&jid=/