

RESTful Web APIs

可因时而变的服务

RESTful Web APIs

中文版



[美] Leonard Richardson & Mike Amundsen 著

Sam Ruby 序

赵震一 李哲 译

O'REILLY®



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

RESTful Web APIs 中文版

近年来，REST的流行导致了各种“RESTful” API的巨大增长，但是这些API却错失了很多架构的好处。通过这本实用指南，你将可以学习到如何设计可用的，并能随着时间不断进化的REST API。通过专注于跨多种领域的解决方案，本书向你展示了该如何使用那些为世界上最成功的分布式计算系统——万维网而设计的工具，从而来创建强大且安全的应用。你将探索REST背后的概念，学习多种可用于创建基于超媒体API的策略，并在本书一步步的指导下整合你所学到的所有内容，从而去设计RESTful的web API。

- 审查了包括集合模式和纯超媒体在内的API设计策略。
- 理解如何将超媒体与表述整合进一个一致的API。
- 探索XMDP和ALPS[®] profile格式是如何帮助你应对web API的“语义挑战”的。
- 学习近二十种标准化的超媒体数据格式。
- 应用在API实现中使用HTTP的最佳实践。
- 使用JSON-LD标准及其他Linked Data方法来创建web API。
- 理解在嵌入式系统使用REST的CoAP协议。

“这是一本了不起的书！
RESTful Web APIs覆盖了当今API领域最重要的趋势和实践。”

——John Musser
Programmable Web创始人

Leonard Richardson, *Ruby Cookbook* (O'Reilly) 一书的作者，曾创建了包括Beautiful Soup在内的多个开源代码库。

Mike Amundsen是包括《使用HTML5和Node构建超媒体API》(O'Reilly)在内的十几本为人所称道的技术图书的作者。

Sam Ruby是W3C HTML工作组的联合主席，同时也是IBM新兴技术组的一名高级技术人员。

图书分类：Web开发

责任编辑：张春雨
策划编辑：张春雨

 **Broadview[®]**
www.broadview.com.cn

O'REILLY[®]
oreilly.com.cn

ISBN 978-7-121-23115-5



9 787121 231155 >

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 SAR and Taiwan)

定价：79.00元

RESTful Web APIs中文版

RESTful Web APIs

[美] Leonard Richardson & Mike Amundsen 著

赵震一 李哲 译

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本书是针对RESTful API的实用指南，通过展示各种用来创建高可用应用的强大工具，讲解REST的深层原理，以及介绍基于超媒体API的策略，使读者得以在将上述内容融会贯通后，设计出让客户高度满意的RESTful的web API。本书极具权威性与前瞻性，既代表了API领域的最前沿趋势，也覆盖了API领域的最重要实践。

本书适合所有从事Web开发和架构工作的读者阅读参考。

©2013 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Publishing House of Electronics Industry, 2014. Authorized translation of the English edition, 2013 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.

本书简体中文版专有版权由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。专有版权受法律保护。

版权贸易合同登记号 图字：01-2014-2675

图书在版编目（CIP）数据

RESTful Web APIs 中文版 / (美) 理查德森 (Richardson,L.) , (美) 阿蒙森 (Amundsen,M.) 著；赵震一，李哲译. —北京：电子工业出版社，2014.6

书名原文：RESTful Web APIs

ISBN 978-7-121-23115-5

I . ① R… II . ①理… ②阿… ③赵… ④李… III . ① 互联网络－网络服务器－程序设计
IV . ① TP368.5

中国版本图书馆 CIP 数据核字 (2014) 第 087092 号

策划编辑：张春雨

责任编辑：张春雨

印 刷：北京丰源印刷厂

装 订：三河市鹏成印业有限公司

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：26 字数：540.8千字

版 次：2014年6月第1版

印 次：2014年6月第1次印刷

定 价：79.00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zlt@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 的会议和峰会聚集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 是一位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

推荐序

“hypermedia as the engine of application state”

上面这段话看起来有些神秘，甚至不能算是一个完整的句子。它究竟是什么意思？它是一段咒语吗？它有什么神奇的魔力？

在我看来，如果把 Web 系统比作是电影《黑客帝国》(The Matrix) 里面那座精密宏伟、无与伦比的 Matrix 系统，上面这段话就是那位华人钥匙匠制作的、能够到达 Matirix 系统后台部分的钥匙。REST 之父 Roy Fielding 无疑是设计建造这座 Matrix 系统的主架构师之一，此外还有 Tim-Berners Lee 这样大神级的人物。这座 Matrix 系统的架构师不是一个人，而是一个英雄的团队。

Roy Fielding 博士一贯反对 design by buzzword (按照时髦的词汇来做设计)，在其 2000 年的博士论文中，他借用《建筑师讽刺剧》(The Architects Sketch) 里面某人所构思的一座到处悬挂着屠宰刀的公寓楼设计，来辛辣地讽刺那些只会 design by buzzword 的人。然而极为讽刺的是，Fielding 在其博士论文中创造出来的“REST”(包括后来出现的“RESTful API”)这个缩写词现在已经成为了 Web 开发社区中最引人注目的一个 buzzword。很多人在尚未真正理解 REST 的一些核心概念的情况下，就到处公然宣称他们所设计的 API 是“RESTful API”。这样的情况太普遍了，以至于 Fielding 本人实在无法忍受，他在 2008 年 10 月写了一篇博客“REST APIs must be hypertext-driven”(RESTful API 必须是超文本驱动的)。这篇博客引起了 Web 开发社区广泛的讨论和反思，后来 Web 开发社区将 Fielding 在其博士论文中关于超媒体作用的论述，特别是本文开头的这句话，简化成了缩写词“HATEOAS”。“超文本驱动”和“HATEOAS”是完全相同的概念，只是表述方式不同，完全可以互换。

没错，HATEOAS 正是 REST 的灵魂，抛弃了 HATEOAS，REST 就失去了灵魂。虽然不支持 HATEOAS 的所谓“RESTful API”在很多场合仍然是很有用的，但是这样的 API 在松耦合和可伸缩性方面会受到一些损失。

Web 开发社区对于设计 RESTful API 最佳实践的探索，有点像 20 世纪初人类征服南极点或者征服更高山峰的竞赛，相关的图书反映出了竞赛的进展。从 2007 年第一本 REST 开发方面的图书《RESTful Web Services》，到今天为止这方面的图书已经不下 30 本。《RESTful Web APIs》这本新书，在对设计 RESTful API 的探索方面可谓是征服了一个新的高度。这本书并不是一本面向初学者的书，因此并没有重复其他 REST 开发入门图书中的一些基础知识的介绍，而是直接站在了前人的肩头（也包括三位作者本人以前写的两本 REST 开发图书《RESTful Web Services》和《Building Hypermedia APIs with HTML5 and Node》）。设计 RESTful API 的一些高级概念，例如对于超媒体的使用（HATEOAS）、各种为超媒体添加语义的技术、媒体类型的选择和设计、设计支持 HATEOAS 的 RESTful API 的流程等，在这本书里面讲的最为清楚。简单来说，这本书最核心的内容，就是如何设计支持 HATEOAS 的 RESTful API。

我在几年前翻译 Roy Fielding 博士论文的过程中，感觉论文虽然非常深刻、精彩，但是也比较抽象。REST 的这些核心概念，在具体的 RESTful API 设计和开发过程中如何实现，仍然是一个巨大的挑战。《RESTful Web APIs》这本书令人满意地解决了这些挑战，它把 REST 的抽象概念，忠实地具像化了，真的是非常棒的工作！

本书的中文版译者赵震一是我的一位朋友，我曾经和震一对于 REST 的一些核心概念做过深入探讨，震一对待技术问题的严谨态度让我印象非常深刻。震一和他的朋友李哲翻译的这本书，阅读起来非常流畅，体验很好，可以想见他们做过大量的推敲和润色。翻译技术图书，是一件辛苦的差事，需要大量的付出。我们在为原著作者的深厚功力而叹服的同时，也应该为好的译者而喝彩！

深入理解学习 REST，就像是一种禅修的过程，可能需要持续几年时间。在这个过程中，有些时候会有顿悟，但是更多的还是渐悟。《RESTful Web APIs》这本书将会带领我们翻越一座新的山峰，过了那座山，后面还会有什么？

李锟

2014 年 5 月 22 日

对《RESTful Web APIs》一书的赞誉

“本书是学习 API 设计必备技艺的最佳起点。”

——Matt McLarty

API Academy 的联合创始人

“在阅读这本书的全部时间里，我一直在心里咒骂。我之所以这样做是因为在我逐个阅读了每一条解释之后，我不由地开始担心——它们写得实在太棒了，以至于在编写我自己的书时很难再找到一个更好的解释。你将再也无法找到另一部能将这个主题挖掘得如此彻底，解释得如此清晰的书了。请拿好这些工具，去创造一些奇妙的东西吧，并将它分享给世界上的其他人，好吗？”

——Steve Klabnik

《Designing Hypermedia APIs》一书的作者

“超媒体是最不容易被理解的 REST 要义，而本书对超媒体格式的解说精彩彻底。”

——Stefan Tilkov

REST 的布道者、作家及顾问

“超媒体 API 的最佳实用手册。人手必备。”

——Ruben Verborgh

语义超媒体研究员

献给 Sienna、Dalton 和 Maggie。——Leonard

献给“主管”Milo，不管是在本次还是很多其他的项目中，
你自始至终是我不变和耐心的朋友！——Mike

序

渐进呈现是用户界面设计中的一个概念，它提倡只在用户需要的时候呈现用户所需的信息。从很多方面来讲，你正在阅读的这本书就是一个实践了该原则的实例。而事实上，仅仅是回到七年前，这本书中所提到的内容很可能还无法“工作”。

正如你所见，相比于编写《RESTful Web Services》（本书的前身）之时，编程世界如今已时过境迁。在那个时候，“REST”这个词还是很少有人使用的。即便在有人使用时也往往被误用，人们对该词也存在着广泛的误解。

在 20 世纪 90 年代中后期，REST 所基于的标准 HTTP 和 HTML 已经被开发出来，并成为了 IETF 和 W3C 的标准，大致已接近它们现今的形态。尽管如此，还是存在着上述这样的情况。Roy Fielding 在他 2000 年发布的论文中引入了 REST 一词，而本书便是基于这篇论文所编写的。

Leonard Richardson 和我曾打算纠正这一（对 REST 来说）不公正的现状。为此，我们主要专注于那些支撑 HTTP 的基础概念，并且就如何将这些概念应用到实际应用中提供了一些具有实践意义的指导。

我认为我们的努力起到了松动这一现状根基的作用，并自此引发了对 REST 支持的雪崩效应。REST 迅速有了它自己的生命，并逐渐成为了一个流行的词汇。事实上现在的情况差不多是：只要有新的 web 接口推出，几乎默认都会称它为 REST。在短短的几年里，我们确实走了很长一段路。

不可否认，REST 作为一个词已经被用过了头，而且通常都没有被正确地应用。但是综合所有的考虑，我还是非常欣慰的，资源和 URI 的概念已经成功地渗入到应用接口设计之中。Web 毕竟是一个颇具弹性的地方，这些新的接口尽管不完美，但是比起那些它们所替代的老的接口却有着飞跃式的进步。

但是我们还可以做得更好。

如果将此事比喻成构建一座大厦，那么现在用于构建大厦的材料已经具备，是时候回过来重新审视一下整个领域，并基于这些概念来构建我们的大厦。下一步应该做的就是去探究那些通用的媒体类型以及特定的超媒体格式。上一本书几乎完全专注于构建正确的 HTTP 应用，而现在是时候让我们去深入探究下超媒体类型（那些像 HTML 一样没有与单独的应用或者甚至是单个厂商紧密绑定的媒体类型）背后的概念了。

HTML 就是这样一个超媒体格式的典型例子，它在 web 架构中占据着特殊的位置。事实上，我个人的发现之旅已经深入到了 HTML 的 W3C 标准的发展，也就是现在的 HTML5。虽然 HTML 确实在这本新书中占据了一个非常显著的位置，但是超媒体主题还有更多的内容需要讨论。所以尽管我们保持着联系，Leonard 选择了一个能够胜任我原来角色的人——Mike Amundsen 来作为本书的联合作者。

非常高兴能看到本书的编写，而在本书的阅读过程中我学习到了大量我从未在任何其他来源接触过的媒体类型。更重要的是，本书展示了这些类型所具有的共同点，以及如何对它们进行区分，因为它们中的每一个都具有自己的特性。

希望本书所做的一切能起到跟它的前身（《RESTful Web Services》）相同的效果。谁知道在另一个七年内所有的事情会不会有机会再重来一次呢，并且对表述性状态移交（Representational State Transfer）^{注1} 中那些仍然被忽视的其他方面做出强调。

——Sam Ruby

注1 此处 Representational State Transfer 参考了李锐修订后的译法，译作“表述性状态移交”，此前曾被翻译为“表述性状态转移”——译者注。

前言

“大多数软件系统在创建时都有一个隐含的假设：整个系统处在一个实体的控制之下；或者至少参与到系统中的所有实体都向着一个共同目标行动，而不是有着各自不同的目标。当系统在互联网上开放地运行时，无法安全地满足这样的假设。”

——Roy Fielding

Architectural Styles and the Design of Network-based Software Architectures

“Discordia 信徒应该一直使用官方的 Discordian 文档编号系统。”

—— Malaclypse the Younger 和 Lord Omar Khayyam Ravenhurst

Principia Discordia^{注1}

我要向你展示一种可以更好地进行分布式计算的方式，它使用了有史以来最成功的分布式系统，即万维网的根本思想。如果你已经决定（或者你的经理已经决定）需要为你的公司发布一个 web API 的话，我希望你能够读一下这本书。不管在你计划中的是一个公共的 API，还是一个纯粹的内部 API，抑或是一个只有受信伙伴可以访问的 API——它们都可以从 REST 的哲学中受益。

如果你想学习如何编写 API 客户端的话，那么这本书对你来说并不是必要的。这是因为大多数现有的 API 设计都基于一些有着数年之久的假设，而这些假设正是我想要摧毁的。

大部分今天的 API 都有着一个很大的问题：一旦部署，它们将无法改变。有一些大名鼎鼎的 API 会在一次部署后多年保持静态不变，即使围绕它们的行业发生着改变，这是因

注1 Discordia 是罗马神话中专司纷争与混乱的女神，Principia Discordia 则是一部与该女神相关的宗教书籍。——译者注

为要改变它们非常困难。

但是 RESTful 架构是为掌控变化而设计的。万维网由数百万的网站组成，运行在数千种不同的服务器实现之上，并且经历着周期性的重新设计。这些网站被数十亿的用户访问着，而这些用户使用着几十种硬件平台之上的数百种不同的客户端实现。你的部署在一开始可能看上去不会如此混乱，但是当你的应用越发接近 web 的规模时，你将会看到越发相似的混乱景象。

要改变一个非常简单的系统通常都是很容易的。在规模很小时，一个 RESTful 系统比一个一键式的解决方案（push-button solution）需要花费更多预支的设计成本。但是当你的 API 逐渐成熟并开始发生变化时，你将会真正需要一些像 REST 这样的方式来应对变化。

- 一个商业上成功的 API 将保持连续多年的可用。一些 API 拥有数百甚至是数以千计的用户。就算问题域只是偶然地发生变化，对客户端带来的累积效应将是非常大的。
- 有一些 API 一直都在发生变化，新的数据元素和业务规则不断地被添加进来。
- 在某些 API 中，每个客户端都可以通过改变工作流来使其适合自己的需求。即使 API 自身从不变化，每个客户端对 API 的经历（鉴于经历不同的工作流）将会不同。
- 编写 API 客户端的人通常不会和编写服务器的人隶属于同一个团队。所有向公共开放的 API 都属于这一类。

如果你不知道外部的客户端是哪种类型的话，你需要在做出变化时格外小心——否则你就需要一个能够在发生变化时保证不会破坏所有客户端的设计。如果你为你的 API 复制了现有的设计，你将很可能只是在重复以往犯过的错误。不幸的是，大部分的改进发生在幕后，它们大都还处于实验阶段并需要经过漫长的标准流程。我将会在本书中讨论到数十种特定的技术，包括很多还仍然处于开发之中。但是我的主要目标是要教会你 REST 的基本原则。通过对这些内容的学习，你将可以对任何实验成果以及那些通过流程审核的标准善加利用。

这里有两个我想在本书中尝试解决的具体问题：重复的工作以及对超媒体的逃避。让我们来看看它们。

重复的工作

现今已发布的 API 都是根据托管它们的公司的名字进行命名的。我们谈论着“Twitter API”、“Facebook API”和“Google+ API”。这三套 API 做着相似的事情。它们都拥一些用户账户的概念，（在其他方面）它们都允许用户向自己的账户发布文本信息。但是每个 API 都具有完全不同的设计，学习一个 API 并不能帮助你学习下一个。当然，

Twitter、Facebook 和 Google 都是互相竞争的大型公司，它们并不想让你很容易地就学会它们竞争对手的 API。但是小公司和非营利性组织也在做着相同的事。它们重新设计着自己的 API，就好比从来没有人在这方面有过相似的想法一样，但是这干扰了它们想让人们实际使用它们的 API 的目标。

让我来向你展示一个例子吧。网站 ProgrammableWeb (<http://www.programmableweb.com/>) 拥有着一个超过 8000 个 API 的目录。当我正在编写此书之时，它已经收录了 57 种微博 API——这些 API 的主要用途是向用户的账户发布文本信息^{注2}。很不错，有 57 家公司在这个领域发布了 API，但是我们真的需要 57 种不同的设计吗？我们在这里讨论并不是那些复杂难懂的业务，例如保险政策或法规守则，我们讨论的只是向用户账号发布少量的文本信息。你想成为那个设计第 58 种微博 API 的人吗？

最显而易见的解决方案便是创建一个微博 API 的标准。但是我们已经有了一个可以很好工作的标准：Atom 发布协议（Atom Publishing Protocol）。它发布于 2005 年，然而几乎没有人使用它。有一些关于 API 的原因，使得每个人都想从头开始设计他们自己的 API，即使从业务的角度来看这并没有什么意义。

我不认为凭我一个人的力量就能结束这种无用功，但是我确实认为可以将问题分解成若干有意义的小块，然后提供一些方式来让新的 API 可以复用已经完成的这些工作。

超媒体很难

早在 2007 年，Leonard Richardson 和 Sam Ruby 编写了本书的前身，*RESTful Web Services*(O'Reilly)。那本书同样也尝试于解决两个大的问题。其中一个问题已经被解决，而另一个却没有任何进展^{注3}。

第一个问题是：在 2007 年，在 API 设计的多个阵营中，REST 学派正在与使用基于 SOAP 等重量级技术的对手学派进行对峙，他们忙于应对来自对方的对 REST 学派合理性的质疑。*RESTful Web Services*一书的出现打破了这种对峙的僵局，有效地为 RESTful 设计原则防御了来自 SOAP 学派的进攻。

很好，这场对峙已经结束，而 REST 赢得了胜利。SOAP API 仍在被使用着，不过仅限于那些起初支持 SOAP 学派的大公司。几乎所有面向公众的 API 口头上都说自己遵守了 RESTful 原则^{注4}。

注2 具有微博(microblogging)标签的ProgrammableWeb API 的完整列表提供了有关列表中每个 API 的信息。

注3 *RESTful Web Services*一书现在是 O'Reilly 开放图书项目 (<http://oreilly.com/openbook/>) 中的一部分。你可以从该图书的页面下载到它的 PDF 版本。

注4 如果你想知道，这便是我们为什么更改了本书书名的原因。“web service”这个词与 SOAP 桀骜得过于紧密，当 SOAP 没落之时，将会带着“web service”一词一起日暮西山。这些日子，每个人都开始改为谈论 API 了。

这又将我们带到了第二个问题：REST 并不只是一个技术词汇——它同样还是一个营销术语。在很长一段时间里，REST 成了一个口号，它象征着任何站在 SOAP 学派对立面的势力。任何没有使用 SOAP 的 API 都将自己标榜为 REST，即使它的设计与 REST 毫无关系甚至是违背了 REST 的基本原则。这样做是错误的，是令人困惑的，它给 REST 这个技术词汇带来了一个坏名声。

这种情况自 2007 年便有了较大的改善。每当我审视那些新的 API，我看到了开发者们的工作，可以看得出，这些开发人员是理解那些我将在本书前几章中解释的概念的。今天大部分举着 REST 大旗的开发者都理解资源和表述，理解如何使用 URL 来为资源命名，以及如何正确地使用 HTTP 方法。因此本书的前三章将不需要做过多的事情，只需让新的开发者加速赶上我们即可。

但是在 REST 中，还有一个方面令大部分的开发人员仍然无法理解，即：超媒体。我们都理解 Web 环境中的超媒体。它只是作为代表链接的一个华丽的词汇。网页经过互相的链接，随即产生了万维网，万维网便是由超媒体驱动的。但是，貌似只要在 web API 中涉及到超媒体，我们便有了心理障碍。这是一个大问题，因为超媒体是一项能让 web API 优雅处理变化的特性。

从 *RESTful Web APIs* 一书的第 4 章开始，我的首要目标便是教会你超媒体的工作原理。如果你从未听到过这个词，我将会结合其他重要的 REST 概念向你讲授该词。如果你听到过超媒体，但是这个概念吓到了你，我将会尽我所能来为你建立勇气。如果你无法将超媒体装进你的大脑，我将会以各种我所能想到的方式来向你展示超媒体，直到你记住并理解它。

RESTful Web Services 一书也涉及到了超媒体，但是这并不是该书的重心所在。就算跳过该书的超媒体部分也可以照样设计出一个功能性的 API。相比之下，*RESTful Web APIs* 则是一本真正有关超媒体的书。

我之所以这样做是因为超媒体是 REST 最重要的一个方面，也是最不被理解的一个方面。在我们完全理解超媒体之前，REST 将会被继续视为一个营销术语，而不是对处理分布式计算复杂性的一次认真的尝试。

这本书讲了什么？

前 4 个章节介绍了 REST 背后的概念，并将其应用于 web API。

第 1 章，网上冲浪

这一章通过一个你已经熟悉的 RESTful 系统：网站，来对基本的术语进行了说明。

第 2 章，一个简单的 API

这一章将我们在 Web 上的经验转换到了一个可编程 API 中，该 API 与第 1 章中所讨论的网站具有相同功能。

第 3 章，资源和表述

资源是 HTTP 中的基本概念，而表述则是 REST 中的基本概念。本章对它们之间是如何进行关联的进行了说明。

第 4 章，超媒体

超媒体是一种缺失的材料，可以将它和表述一起整合进一个一致的 API。本章展示了超媒体可以做些什么，并大都使用了你已经熟悉的超媒体数据格式：HTML。

接下来的 4 个章节描述了用于设计超媒体 API 的不同策略：

第 5 章，领域特定设计

显而易见的一个策略是设计一个全新的标准来处理你的具体问题。我使用了 Maze+XML 标准来举例说明。

第 6 章，集合模式（Collection Pattern）

很特别的一个模式：集合模式，在 API 设计中出现了一次又一次。在这一章中，我展示了两个不同的标准：Collection+JSON 和 AtomPub，它们都实现了这种模式。

第 7 章，纯 - 超媒体设计

当集合模式无法满足你的需求时，你可以使用一种通用的超媒体格式来传达任意的表述。这一章使用了 3 种通用超媒体格式（HTML、HAL 和 Siren）作为实例来展示上述方式是如何工作的。这一章同样还介绍了 HTML 微格式和微数据，并以此引出了下一章的内容。

第 8 章，Profile

Profile 可以用于填平某种数据格式（可以被多种不同的 API 使用）与某个特定 API 实现之间的鸿沟。我所推荐的 profile 格式是 ALPS，但是我同样也提到了 XMDP 和 JSON-LD。

在这一章中，我给出的建议开始超越了编写本书时的艺术状态（outstrip the state of the art）。因此我不得不为本书开发了 ALPS 格式，因为当时还没有其他可以完成这项工作的选择。如果你已经对基于超媒体的设计非常熟悉，那么你可以跳过前面的部分直接来到第 8 章，但是我不认为你应该跳过第 8 章。

第 9 章到第 13 章涉及到了例如选择正确的超媒体格式以及如何充分利用 HTTP 协议这些实用主题。

第 9 章，API 设计流程

这一章将本书到目前为止讨论过的所有内容整合在了一起，并给出了一个用于设计 RESTful API 的按部就班的指引。

第 10 章，超媒体动物园

为了展示超媒体的能力，本章讨论了近 20 种标准化的超媒体数据格式，它们中的大部分并没有在本书的其他章节中有所涉及。

第 11 章，API 中的 HTTP

本章给出了在 API 实现中使用 HTTP 的一些最佳实践。我同样也讨论了一些 HTTP 的扩展，包括即将到来的 HTTP 2.0 协议。

第 12 章，资源描述和 Linked Data

Linked Data 是语义网社区用以实现 REST 的方法。JSON-LD 毫无疑问是最重要的 Linked Data 标准。我们曾在第 8 章谈到过它，而我在本章中对它进行了重温。本章同样讨论了 RDF 数据模型和一些我在第 10 章没能谈到的基于 RDF 的超媒体格式。

第 13 章，CoAP：嵌入式系统的 REST

本章通过对 CoAP 的讨论结束了本书的核心部分，CoAP 是一个完全没有使用 HTTP 的 RESTful 协议。

附录 A，状态法典

作为对第 11 章的扩展，本附录对 HTTP 规范中定义的 41 个标准状态码以及一些作为扩展定义的有用的状态码进行了深入的考察。

附录 B，HTTP 报头法典

与附录 A 相似，本附录也是对第 11 章的扩展。它为 HTTP 规范中定义的 46 个请求和响应报头，以及一些扩展提供了详细的概述。

附录 C，为 API 设计者准备的 Fielding 论文导读

本附录包括了一个围绕 REST 的基础文档（即 Fielding 论文）展开的深度讨论，用以说明该文档在 API 设计中的意义。

词汇表

该词汇表包含了你在 *RESTful Web APIs* 一书中会经常遇到的一些术语的定义。如果

你想要熟悉基本概念，或者是需要一个快速的、用于浏览特定概念定义的提示工具，那么这将是一个很好的去处。

这本书没讲什么

RESTful Web Services 是最早有关 REST 的一部书籍，并且涉及了很多的背景知识。幸运的是，现在已经有了超过一打的关于 REST 的各个方面的书，而这样便让 *RESTful Web APIs* 可以腾出精力来专注于核心的概念。

为了保持本书的专注度，我去除了部分你可能会希望我覆盖到的主题。我想要告诉你这本书没讲什么，这样你便可以选择不购买本书，从而不会为此感到失望：

- 本书没有涉及到客户端编程。编写客户端来消费基于超媒体的 API 是一种新的挑战。眼下，对一个通用的 API 客户端来说，我们可以拥有的就是一个能够发送 HTTP 请求的代码库。这在 2007 年如此，时至今日仍然如此。因为问题存在于服务器端。当你要为一个现有的 API 编写客户端时，你将只能仰仗 API 设计者。我无法给你任何通用的建议，因为现在并不存在跨 API 的一致性。这就是为什么我在本书中鼓动大家保持对服务器端一致性的积极性的原因。当 API 之间变得更加相似，我们也就能够编写更为精细的客户端工具。

第 5 章包含了一些示例的客户端实现，并尝试对不同类型的客户端进行归类，但是如果你想要一本全部关于 API 客户端的书，那么这本书并不适合你。我不认为目前市场上存在着你想要的书。

- 世界上部署最广泛的 API 客户端应当属 JavaScript 的 XMLHttpRequest 代码库。在每一个浏览器中都拥有一份该代码的副本，而当今大部分的网站也都构建于那些专门设计供 XMLHttpRequest 消费的 API 之上。对于本书来讲，这个领域过于庞大而无法完全涉及。市场上有着全篇专门讲述 JavaScript 代码库的书籍。
- 我花费了相当多的时间来讨论 HTTP 的机制（第 11 章、附录 A 和附录 B），但是我没有覆盖到任何具有深度的特定 HTTP 主题，有一些主题，尤其是关于缓存和代理这样的 HTTP 中间组件的相关主题，我只是简单地在本书中有所涉及。
- *RESTful Web Services* 重点专注于将你的业务需求拆分成一组互相关联的资源。根据我 2007 年以来的经验，我深信将 API 设计作为一种资源设计来思考可以有效地避免考虑超媒体。但本书却采用了一种截然不同的方式，它专注于表述和状态转换，而非资源。

也就是说，资源设计的方式无疑也是有效的。如果想听听在该方向发展的建议，我推荐由 Subbu Allamaraju 编著的 *RESTful Web Services Cookbook*(O'Reilly)。

管理备注

本书有两名作者（Leonard 和 Mike），但是在编写本书的过程中，我们被合并成了一个第一人称，即“我”。

本书中的内容没有与任何特定的编程语言绑定。所有的代码都采用了在网络协议（通常是 HTTP）中发送的消息格式（通常是 JSON 或 XML 文档）。我将假定你已经熟悉了常规的编程概念，比如反模式和广度优先搜索，以及你对万维网是如何工作的有着一个基本的理解。

我将不会呈现真实的代码，但是我在第 1 章、第 2 章以及第 5 章中所讨论的服务器和客户端背后的真实代码是存在的。你可以通过 *RESTful Web APIs* 一书的 GitHub 仓库 (<https://github.com/organizations/RESTful-Web-APIs>)，或者是通过官方的网站 (<http://www.restfulwebapis.org/>) 来获取这些代码，从而可以自行将它运行起来。这些客户端和服务器是采用 JavaScript 编写的，使用的是 Node 代码库。

我之所以选择 Node 是因为它让我可以使用相同的编程语言来编写客户端及服务端的代码。你无须为了理解某个客户 - 服务器事务的两端而需要费神地在两种编程语言之间来回切换。Node 是开源的，并且可以运行在 Windows、Mac 和 Linux 系统上。它很容易在这些操作系统上进行安装，而你应该无须碰触太多的麻烦就可以获取这些例子，并将它们运行起来。

我将这些代码托管在 GitHub 上，因为随着时间的变化，我可以非常容易地更新这些实现。这同样也使得读者们可以为这些示例的客户端和服务器贡献其他编程语言的移植版本。

理解标准

万维网并不是一个供科学的研究的客观事物。它是一种社会建构（social construct）——一组按特定方式做事的协议。幸运的是，它与其他的社会建构不同（比如礼节），Web 底层的协议通常是已经约定好的。人类的 web 底层最核心的协议就是 RFC 2616（HTTP 标准）、HTML4 的 W3C 规范以及 ECMA-262（JavaScript 的基础标准，也被称为 ECMAScript）。每个标准都做了不同的工作，在本书的课程中，我还讨论数十个专门设计供 API 使用的其他标准。

这些标准的伟大之处在于它们给了你坚实的基础。你可以使用它们来构建一种全新的网站或 API，即某些没有人曾经尝试过的东西。并且无须向你的所有用户进行对整个系统的说明，你将只需要对那些新的部分进行说明即可。

而坏消息便是这些协议通常晦涩难读：采用拗口而精确的英文编写而成的、由 ASCII 文本组成的冗长的段落，像“should”这样的日常词汇也具有了技术含义，并被标准化为大写的“SHOULD”^{注5}。市场上大量的技术书籍之所以热卖，便是因为很多人希望通过阅读书籍而避免直接阅读这样的标准文档。

好吧，我无法做出任何的担保。如果这些标准中的任何一个貌似会成为某个你可以在工作中采纳的选择，那么你必须愿意深入其规范并真正地去理解它（或者买一本能详细覆盖它的书籍）。对于像 Siren、CoAP 和 Hydra 这样的标准，我没有空间来为它们提供除基本概览之外的更多详情。更不用说一旦给出了太多的细节，将会让所有在工作中不需要这些特定标准的读者对此产生厌倦。

当你穿梭于标准的森林之中时，请记住一点，并不是所有的标准都具有相同的力量。有些标准制定得非常完善，每个人都在使用，如果你违背了这些标准，那么将会为你带来大量的麻烦。而其他的标准只是某些人的一家之言，而他们想法也未必比你自己的想法更加高明。

我认为将标准划归到 4 个分类中将会很有帮助：fiat 标准、个人标准、公司标准以及开放标准。我将在本书中一直使用这些词，所以让我来逐个对它们进行深入的解释。

Fiat 标准

Fiat 标准并不是真正的标准；它们是一些行为。没有人认同它们。它们只是对某些人做事方式的一种描述。这些行为可以用文档记录下来，但是它缺少了作为标准的一个前提——其他人应该以相同的方式做事。

几乎当今的每个 API 都是一种 fiat 标准，即某种与特定公司相关联的一次性设计。这就是我们谈论着“Twitter API”、“Facebook API”以及“Google+ API”的原因。你可能需要通过理解这些设计来做好你的工作，并且可能要为这些设计编写你自己的客户端，除非你服务的便是我们讨论中的这家公司。请别指望在你自己的 API 中使用这样的设计。如果你复用了一个 fiat 标准，我们不会说你的 API 符合某个标准，我们只会认为它是一种复制。

我在本书中尝试解决的一个主要的问题便是设计工作中数以百计的人年（person-year）都被束缚在制定 fiat 标准这种无法复用的工作中。必须要结束这样的事情。如今设计一个新的 API 意味着重新发明一长串轮子。一旦你的 API 完成了，你的客户端开发人员必须在客户端一侧对应地重新发明一串轮子。

注 5 “SHOULD”的意义见 RFC 2119。

即便在理想的情况下，你的 API 仍将是一个 fiat 标准，因为你的业务需求将会与其他人的需求有所不同。但是从观念上来说，一个 fiat 标准将只是一束掩盖了若干其他标准的强光。

当我在描述一个 fiat 标准时，我将会链接到它的人类可读的文档。

个人标准

个人标准是一种标准（你将被邀请阅读文档，并自行实现该标准），但是它们只是一个人的意见。我在第 5 章中描述的 Maze+XML 标准便是一个很好的例子。别指望 Maze+XML 是用来实现迷宫游戏 API 的标准方式，但是如果它能为你工作，你便可以很好地使用它。某些其他人已经帮你完成了设计工作。

个人标准相对于别的标准，通常会采用较为不够正式的语言。很多开放标准开始是作为个人标准起步的——在经历了大量的实验后，作为编外项目被扶正。我在第 7 章中谈到 Siren 就是一个很好的例子。

当我在描述一个个人标准时，我将会链接到它的规范。

公司标准

公司标准是由企业集团为了尝试解决某个困扰着它们的问题而共同创建的标准，或者是由一个单一的公司试图代表其客户解决某个反复出现的问题而制定的。公司标准相比于个人标准往往定义得更加完善，所使用的语言也更加规范，但是它们并没有比个人标准具有更大的力量。它们只是某个公司（或某个企业集团）的意见。

公司标准包括了活动流（Activity Streams）和 schema.org 的元数据模式，这两项都在第 10 章有所涉及。很多的行业标准都是以公司标准的形式发起的。OData（同样在第 10 章中讨论过）最初以微软项目的形式启动，但是在 2012 年被提交到了 OASIS，并最终成为了一项 OASIS 标准。

当我在描述一个公司标准的时候，我将会链接到它的规范。

开放标准

一项开放标准将会历经一个由委员会发起的设计过程，或者至少拥有一个开放的评论期，在此期间，大量的人员会阅读规范，对它提出意见，并提供改进的建议。在这个过程的最后，该规范将得到某些公认的标准组织的祝福。

该过程给开放标准带来了一定的道德力量。如果一个开放标准的内容或多或少正是你想

要的，你真应该使用它来替代制定自有的 fiat 标准。开放标准在设计过程和评论期间可能会暴露出大量的问题，这样一来你在实际的使用中将不大会遇到太多的问题。

一般来说，伴随着开放标准还会有很多的协议，这些协议会对你做出承诺，即当你在实现了这些开放标准之后不会受到那些参与了该标准过程的公司的专利侵权诉讼。相比之下，实现别人的 fiat 标准有可能会激起他们对你的专利侵权诉讼。

本书中提到的一些开放标准大都出自那些大名鼎鼎的标准组织：ANSI、ECMA、ISO、OASIS，尤其是 W3C。我无法描述在成为这些标准组织的一员后将可以怎么样，因为我从未这样做过。但是大部分重要的标准组织^{注6} 中的任何一个都可以向 IETF 做出贡献，该组织管理着所有重要的 RFC。

RFCs(Requests for Comments) 和互联网草案

大部分的 RFC 都是通过一个叫作标准追踪（Standards Track）的流程进行创建的。贯穿本书，我将会引用那些处于标准追踪不同环节的文档。我想要简明地讨论下追踪是如何运作的，这样一来你便会知道在采纳我的建议时需要多么认真。

一项 RFC 的生命开始于一项互联网草案。这是一个看上去很像标准文档的文档，但是你不应该建立基于它的实现。你应该查找该规范的问题并给予反馈。

一项互联网草案具有一个固定 6 个月的生命时间。在它发布的 6 个月之后，该草案必须作为一项 RFC 被通过，或者是由一项更新后的草案进行替代。如果以上的事情都没有发生，那么该草案就会被认为已经过期，并且不应该被应用于任何事物。另一方面，如果草案通过了审核，那么它就会立即过期并由一项 RFC 来取代它。

因为具有固定的过期时间，同时也因为一项互联网草案从技术上说并不是任何类型的标准，所以在一本书中提及它们是一件非常棘手的事情。同时，API 设计是一个迅速变化的领域，多一项互联网草案可以选择总比没有好。我将在本书中提到多个互联网草案，并且假设它们在成为 RFC 后不会有太大的变化。这个假设保持着良好的状态；有多个在我编写本书时提到的互联网草案，目前已经成为了 RFC。如果某个特定的互联网草案最后并不成功，那么我在此只能先行道歉了。

RFC 和互联网草案都指定了代码名称。当我描述它们中的一员时，我不会链接到它的规范。我将只会提及它的代码并让你自己去查找它。举个例子，我会将 HTTP/1.1 规范称为 RFC 2616。我也会使用名字来指代某个互联网草案。举个例子，我将会使用“draft-snell-link-method”来指代向 HTTP 添加 LINK 和 UNLINK 方法的提案。

^{注 6} 不管怎样，我们要达成这本书的目的。如果你需要螺丝和螺栓的标准型号，你将需要 ANSI 或 ISO。

当你看到它们中某个的代码名称时，你都可以通过网络搜索来找到该 RFC 或互联网草案的最新版本。如果一项互联网草案在本书出版之后成为了 RFC，那么该互联网草案的最终版本会链接到对应的 RFC。

当我在描述一项 W3C 或 OASIS 标准时，我将会链接到对应的规范，因为那些规范没有给出代码名称。

本书所使用的约定

以下是本书所使用的排版约定：

斜体

表示新的术语（中文则用楷体）、URL、邮件地址、文件名称和文件扩展名。

等宽字体

用于表示程序片段，也可以用在正文中表示例如变量名或函数名等程序元素、数据库、数据类型、环境变量、语句和关键词（中文则用黑体）。

加粗的等宽字体

展示了应该由用户逐字输入的命令或其他文本。

倾斜的等宽字体

展示了应该由用户提供值或由上下文决定值来进行替换的文本。



这个图标代表小窍门、建议和说明。



这个图标代表警告信息。

使用代码示例

本书就是要帮读者完成工作的。通常，如果本书包含了代码示例，你可以在你的程序和文档中使用本书中的代码。除非你复制了大段的代码，否则你无须联系我们来取得许可。举个例子，在编写程序时使用了本书中的数块代码是不需要经过许可的。出售或分发来自 O'Reilly 图书的示例 CD-ROM 是必须经过许可的。引用本书及本书中的示例代码来回答问题是不需要经过许可的。将大量的示例代码整合到你的产品文档中必须经过许可。

我们希望（但不是必须）你在使用我们的代码时标明出处。出处通常都包含书名、作者、

出版社和 ISBN。例如：“*RESTful Web APIs* by Leonard Richardson and Mike Amundsen (O'Reilly). Copyright 2013 Leonard Richardson and amundsen.com, Inc., and Sam Ruby. 978-1-449-35806-8”。

如果还有其他需要使用代码的情形需要与我们沟通，可以随时与我们联系：permissions@oreilly.com。

Safari® Books Online

Safari Books Online (www.safaribooksonline.com) 是一家应需而变的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

Safari Books Online 是技术专家、软件开发人员、Web 设计师、商务人士和创意人士开展调研、解决问题、学习和认证培训的第一手资料。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，请访问我们的网站。

联系我们

关于本书的建议和疑问，可以与下面的出版社联系：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室 (100035)
奥莱利技术咨询（北京）有限公司

我们将关于本书的勘误表，例子以及其他信息列在本书的网页上，网页地址是：

<http://www.oreilly.com/catalog/9781449358068>

如果要评论本书或者咨询关于本书的技术问题，请发邮件到：

bookquestions@oreilly.com

想了解关于 O'Reilly 图书、课程、会议和新闻的更多信息，请访问以下网站：

<http://www.oreilly.com.cn>

<http://www.oreilly.com>

致谢

我们欠 Glenn Block 一声谢谢，他花费了无数的时间来聆听我们的想法，并编写了真实的代码来测试这些想法。我们要向 Benjamin Young 和 RESTFest 的所有人表示感谢，是他们同意成为我们实验的一部分，并给了我们很好的反馈和建议，即使我们有时听不大进去。我们要感谢 Mike 在 Layer 7 Technologies 的同事，包括 Dimitri Sirota 和 Matt McLarty，他们在这个项目的工作上给予了他支持和鼓励。我们要感谢 Sam Ruby 和 Mike Loukides，他们对于本书的前身 *RESTful Web Services* 一书来说至关重要。我们要感谢 Sumana Hariharan，她是 Leonard 的贤内助。我们还要感谢社区，它为我们创造了良好的协作和交流 REST 与 API 的环境；尤其是 Yahoo 的 REST-Discuss、Google Groups 的 API-Craft 以及 LibreList 的 Hypermedia group。

最后，感谢那些阅读了早期手稿并提供了必要的批评和支持的人，他们是：Carsten Bormann、Todd Brackley、Tom Christie、Timothy Haas、Jamie Hodge、Alex James、David Jones、Markus Lanthaler、Even Maler、Mark Nottingham、Cheryl Phair、Sergey Shishkin、Brian Sletten、Mark Stafford、Stefan Tilkov、Denny Vrandečić、Ruben Verborgh 和 Andrew Wahbe。

目录

序	xix
前言	xxi
第 1 章 网上冲浪.....	1
场景 1：广告牌.....	2
资源和表述	2
可寻址性	3
场景 2：主页	3
短会话（Short Session）.....	5
自描述消息（self-descriptive message）.....	5
场景 3：链接	6
标准方法	8
场景 4：表单和重定向	9
应用状态（Application State）.....	11
资源状态（resource state）	12
连通性（connectedness）	13
与众不同的 Web.....	14
Web API 落后于 Web	15
语义挑战	16

第 2 章 一个简单的 API.....	17
HTTP GET：安全的投注	18
如何读取 HTTP 响应	19
JSON	20
Collection+JSON	21
向 API 写入数据	23
HTTP POST：资源是如何生成的	24
由约束带来解放	26
应用语义所产生的语义鸿沟	27
第 3 章 资源和表述	29
万物皆可为资源	30
表述描述资源状态	30
往来穿梭的表述	31
资源有多重表述	32
HTTP 协议语义（Protocol Semantics）.....	33
GET	35
DELETE	36
幂等性（Idempotence）.....	36
POST-to-Append	37
PUT	38
PATCH	39
LINK 和 UNLINK	40
HEAD	40
OPTIONS	41
Overloaded POST	41
应该使用哪些方法？	42
第 4 章 超媒体	45
将 HTML 作为超媒体格式	46
URI 模板	49
URI vs URL	50
Link 报头	51

超媒体的作用	52
引导请求	52
对响应做出承诺	54
工作流控制	55
当心冒牌的超媒体！	56
语义挑战：我们该怎么做？	57
第 5 章 领域特定设计	59
Maze+XML：领域特定设计	60
Maze+XML 是如何工作的.....	61
链接关系	62
访问链接来改变应用状态	64
迷宫集合	65
Maze+XML 是 API 吗？	67
客户端 1：游戏.....	68
Maze+XML 服务器	72
客户端 2：地图生成器	74
客户端 3：吹牛者	76
客户端做自己想要做的事.....	77
对标准进行扩展	77
地图生成器的缺陷	80
修复（以及修复后的瑕疵）	81
迷宫的暗喻	83
解决语义鸿沟	83
领域特定设计在哪里？	83
最终的奖赏	84
报头中的超媒体	84
抄袭应用语义	84
如果找不到相关的领域特定设计，不要自己制造.....	86
API 客户端的种类	86
人类驱动的客户端	86
自动化客户端.....	87

第 6 章 集合模式 (Collection Pattern)	91
什么是集合?	93
链向子项的集合	93
Collection+JSON.....	94
子项的表示	95
写入模板 (Write Template).....	98
搜索模板	99
一个 (通用的) 集合是如何工作的.....	100
GET	101
POST-to-Append.....	101
PUT 和 PATCH.....	101
DELETE.....	102
分页	102
搜索表单	103
Atom 发布协议 (AtomPub)	103
AtomPub 插件标准.....	105
为什么不是每个人都选择使用 AtomPub ?	106
语义挑战 : 我们应该怎么做?	107
第 7 章 纯 - 超媒体设计	111
为什么是 HTML?.....	111
HTML 的能力	112
超媒体控件	112
应用语义插件	113
微格式.....	115
hMaze 微格式	116
微数据.....	118
改变资源状态	119
为表单添加应用语义	121
与超媒体相对的是普通媒体	125
HTML 的局限性	126
拯救者 HTML5?.....	127
超文本应用语言	128

Siren.....	131
语义挑战：我们现在要怎么做？	133
第 8 章 Profile	135
客户端如何找寻文档?	136
什么是 Profile ?	137
链接到 Profile	137
Profile 链接关系.....	137
Profile 媒体类型参数	138
特殊用途的超媒体控件.....	139
Profile 对协议语义的描述.....	139
Profile 对应用语义的描述.....	140
链接关系	141
不安全的链接关系	142
语义描述符	142
XMDP：首个机器可读的 Profile 格式	143
ALPS.....	146
ALPS 的优势.....	150
ALPS 并不是万金油.....	152
JSON-LD	153
内嵌的文档	156
总结	158
第 9 章 API 设计流程	161
两个步骤的设计流程	161
七步骤设计流程.....	162
第 1 步：罗列语义描述符	163
第 2 步：画状态图	164
第 3 步：调整命名	168
第 4 步：选择一种媒体类型	172
第 5 步：编写 Profile	173
第 6 步：实现.....	174
第 7 步：发布.....	174

实例：You Type It, We Post It.....	177
罗列语义描述符	177
画状态图	178
调整名称	179
选择一种媒体类型	180
编写 Profile	181
设计建议	182
资源是实现的内部细节	182
不要掉入集合陷阱	183
不要从表述格式着手	184
URL 设计并不重要	184
标准名称优于自定义名称	186
设计媒体类型	187
当你的 API 改变时	189
为现有 API 添加超媒体	194
改进基于 XML 的 API	195
值不值得？	196
Alice 的第二次探险	196
场景 1：没有意义的表述	196
场景 2：Profile	198
Alice 明白了	200
 第 10 章 超媒体动物园	203
领域特定格式	204
Maze+XML	204
OpenSearch	205
问题细节文档	205
SVG	206
VoiceXML	208
集合模式的格式	210
Collection+JSON	211
Atom 发布协议	211
OData	212

纯超媒体格式	219
HTML.....	219
HAL	220
Link 报头	222
Location 和 Content-Location 报头	222
URL 列表	223
JSON 主文档 (Home Documents).....	223
Link-Template 报头	224
WADL	225
XLink	226
XForms.....	227
GeoJSON : 一个令人困惑的类型	228
GeoJSON 没有通用的超媒体控件.....	230
GeoJSON 没有媒体类型	232
从 GeoJSON 学习到的经验	233
语义动物园	234
链接关系的 IANA 注册表	234
微格式 WiKi.....	235
来自微格式 Wiki 的链接关系.....	236
第 11 章 API 中的 HTTP	241
新 HTTP/1.1 规范	242
响应码.....	242
报头	243
表述选择	243
内容协商 (Content Negotiation)	243
超媒体菜单	244
标准 URL (Canonical URL)	245
HTTP 性能	246
缓存 (Caching)	246
条件 GET 请求 (Conditional GET)	247
Look-Before-You-Leap 请求.....	249
压缩.....	250

部分 GET 请求 (Partial GET)	250
Pipelining	251
避免更新丢失问题	252
认证	254
WWW-Authenticate 报头和 Authorization 报头	255
Basic 认证	255
OAuth 1.0	256
OAuth 1.0 的缺点	259
OAuth 2.0	260
何时不采用 OAuth	261
HTTP 扩展	261
PATCH 方法	262
LINK 和 UNLINK 方法	262
WebDAV	263
HTTP 2.0	264
第 12 章 资源描述和 Linked Data	267
RDF	268
RDF 将 URL 作为 URI 对待	270
什么时候使用描述策略	271
资源类型	273
RDF Schema	274
Linked Data 运动	277
JSON-LD	278
将 JSON-LD 作为一种表述格式	279
Hydra	280
XRD 家族	285
XRD 和 JRD	285
Web 主机元数据文档	286
WebFinger	287
本体动物园 (Ontology Zoo)	289
schema.org RDF	289
FOAF	290

vocab.org	290
总结：描述策略生机盎然！	290
第 13 章 CoAP: 嵌入式系统的 REST	293
CoAP 请求	294
CoAP 响应	294
消息种类	295
延迟响应 (Delayed Response)	296
多播消息 (Multicast Message)	296
CoRE Link Format	297
结论：非 HTTP 协议的 REST	298
附录 A 状态法典	301
附录 B HTTP 报头法典	325
附录 C 为 API 设计者准备的 Fielding 论文导读	349
词汇表	365

网上冲浪

万维网变得越来越普遍的一个重要原因就是它的易用性，普通人不需要经过多少培训就可以使用它来完成一些很有价值的工作。但是在这表象之后，Web 也是一个用于分布式计算的功能强大的平台。

那些让普通人易于使用互联网的原则，也同样适用于某些自动化的软件 agent “用户”。比如，一些软件被设计用来在不同的银行账户间自动地进行货币交易（或者用来完成现实世界的其他任务），为了完成相应的任务，它们所采用的基本技术和人类所使用的技术是相同的。

就本书而言，有三项技术支撑着当今的互联网，它们分别是：URL 命名约定、HTTP 协议和 HTML 文档格式。URL 和 HTTP 是比较简单的，但是如果想要将它们应用到分布式编程中，和大多数的 web 开发人员相比，你就必须更加了解它们的细节。本书的前几章就专门用来帮助大家学习这些内容。

HTML 的内容就有点复杂了。在 web API 的世界里，有许许多多的数据格式在试图取代 HTML 的地位。对这些数据格式的研究将从第 5 章开始，并占据本书的多个章节的篇幅。暂时，我将重点放在 URL 和 HTTP 上面，仅仅使用 HTML 作为一个例子。

我计划从一个和万维网相关的简单故事开始，以此来解释隐藏在万维网设计背后的原则以及驱动它成功的缘由。虽然你对 Web 很熟悉，但是对于那些使得 Web 运转起来的技术和概念你可能并没听说过，所以这个故事是很简单的，以便于你能够理解它。如果你对类似于“将超媒体作为应用状态的引擎 (hypermedia as the engine of application state)”的技术不是很清楚，我希望你能通过这个简单而又具体的例子来得到一些收获。

那我们现在开始吧。

2 场景1：广告牌

一天，Alice 正在城里散步，她看到了一个广告牌（见图 1-1）。

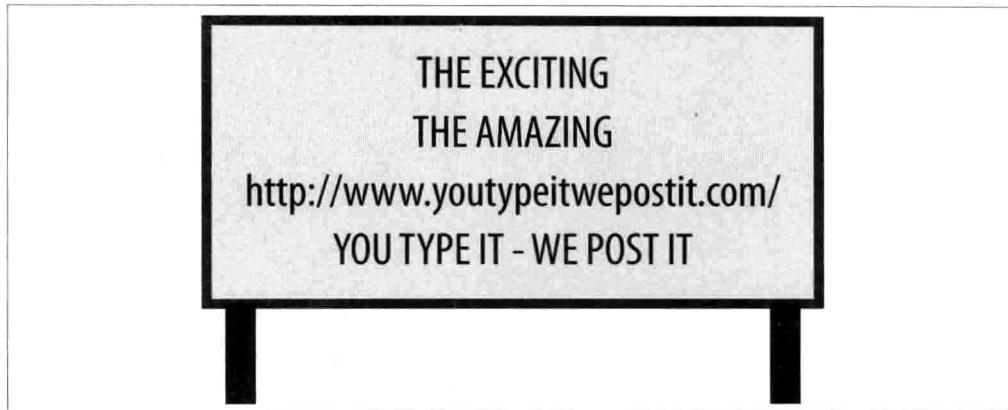


图1-1 广告牌

（顺便说一下，这个虚构的广告牌上面宣传的是我为这本书设计的一个真实的网站，你可以自己试着访问一下。）

这个广告牌是在 20 世纪 90 年代中期建立的，那时候 Alice 已经可以记事了，她至今都还记得这个广告牌刚开始展示这个 URL 时候公众的反应。起初，人们都拿这些看起来很怪异的字符串开玩笑，那时的人们并不清楚 “http://” 和 “youtypeitwepostit.com” 的真正含义。但是 20 年后的今天，几乎每个人都知道如何使用这个 URL：将 URL 输入到 web 浏览器的地址栏，然后按下回车键。

Alice 也是这样做的：她拿出了自己的手机，将 “<http://www.youtypeitwepostit.com/>” 输入进了浏览器地址栏。我们的故事的第一个场景就到此结束了，我们保留一个悬念：URL 的另一端是什么呢？

资源和表述

非常抱歉中断了这个故事，但是我需要介绍一些基本术语。Alice 的 web 浏览器会试图向特定的 web 服务器的 URL:<http://www.youtypeitwepostit.com/> 发送一条 HTTP 请求。一个 web 服务器可以管理许多不同的 URL，并且每个 URL 被授权访问服务器上的不同数据。

我们所说的 URL 是一些事物的 URL，比如一个产品、一个用户或主页等。这些用 URL

命名的事物的专业术语是资源 (resource)。

这个 URL:<http://www.youtypeitwepostit.com/> 标识的资源应该就是那个广告牌上宣传的网站的主页。但是也许只有将这个故事继续下去，等到 Alice 的浏览器真正发送了 HTTP 请求以后，我们才能确定这些猜想。

web 浏览器为一个资源发送了 HTTP 请求以后，服务器会发送一个文档作为响应（通常是一个 HTML 文档，但是有时候是二进制图片或者其他东西）。不论服务器发送了什么文档，我们都将这个文档称为资源的表述 (representation of the resource)。3

每个 URL 标识一个资源。客户端向某个 URL 发送了一条 HTTP 请求以后，它就会收到对应资源的表述。客户端从来都不会直接看到资源，看到的都是资源的表述。

我会在第 3 章讲解更多关于资源和表述的内容。现在我只是想要使用资源和表述这两个术语来开始讨论可寻址性原则。

可寻址性

每个 URL 代表一个也仅代表一个资源。如果一个网站上面有两个从概念上讲并不相同的事物，那么我们就应该将它们作为两个资源并为它们分配不同的 URL。当网站违背了这个规则的时候，我们便会因为在使用时无所适从而心情沮丧。有一些餐厅的网站在这一点上做得就很差劲。有时候，整个网站堆砌在一个 Flash 界面里，并没有提供那些指向我们就其本身可以讨论的资源的独立 URL，比如菜单以及用来显示餐厅地址的地图等。

可寻址性原则就是说每个资源应该有一个属于自己的 URL。如果你的程序里面有些东西非常重要，它就应该有一个唯一的名字，一个 URL，这样你和你的用户就可以非常清楚地、毫无歧义地引用它了。

场景2：主页

回到我们的故事中，当 Alice 将广告牌上的 URL 输入到她的浏览器地址栏的时候，她就通过因特网向 web 服务器上面的 <http://www.youtypeitwepostit.com/> 发送了一个 HTTP 请求：

```
GET / HTTP/1.1
Host: www.youtypeitwepostit.com
```

web 服务器处理这个请求 (Alice 和她的 web 浏览器都不需要知道是如何处理的) 然后发送响应结果：

HTTP/1.1 200 OK
Content-type: text/html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Home</title>
  </head>
  <body>
    <div>
      <h1>You type it, we post it!</h1>
      <p>Exciting! Amazing!</p>

      <p class="links">
        <a href="/messages">Get started</a>
        <a href="/about">About this site</a>
      </p>
    </div>
  </body>
</html>
```

4 >

在响应信息的头部的 200 是状态码，也被叫作响应码。这是服务器告诉客户端发生了什么事情的快捷方式。实际上还有很多其他的 HTTP 状态码，我会在附录 A 中对它们都进行一一介绍，但是最常见的状态码是我们前面见到的状态码 200。200 (OK) 表示这个请求被接受并正确无误地处理了。

Alice 的 web 浏览器将响应数据作为 HTML 文档进行了解析，并以图形化的形式展示了出来（见图 1-2）。



图1-2 You Type It... 主页

现在 Alice 可以浏览这个网页了，她终于明白那个广告牌在说什么了。这个广告牌是在给一个类似于 Twitter 的微博网站做宣传，尽管这个网站实际上并不像广告牌说的那样令人兴奋，但是作为一个例子也已经足够了。

Alice 和 web 服务器的第一次即时互动揭示了 Web 的一系列更重要的特性。

短会话 (Short Session)

故事发展到了这里，Alice 的 web 浏览器正在显示那个网站的主页。从她的角度来看，她已经“登录到”了这个页面，这也就是她在虚拟的网络空间的当前位置。但是就服务器考虑而言，Alice 哪也不存在。服务器也已经忘记了她的存在。

HTTP 会话只维持在一次请求过程中，客户端发送请求，服务器进行响应。这就意味着，Alice 可以将她的手机关一晚上，然后，当她的浏览器从内部缓存 (cache) 中恢复出这个网页以后，她依旧可以单击页面上的任意一个链接，网页应该还是可以继续工作的（和 SSH 会话相比，如果你将你的电脑关机，SSH 会话就终止了）。

Alice 甚至可以将她手机里面的网页一直打开着，在她在六个月后再次单击网页上面的链接时，web 服务器还是会迅速地做出响应，仿佛她只是等待了几秒钟。Web 服务器不需要为了 Alice 而通宵工作。当 Alice 不发送 HTTP 请求的时候，服务器并不清楚 Alice 的存在。

这项原则有时候就被称为无状态性 (statelessness)。我想这是一个令人困惑的术语，因为系统里面的客户端和服务器端都要保存状态：它们只是保存不同类型的状态。“无状态性”术语是指服务器不关心客户端的状态（在后面的章节中，我将讲解不同类型的状态）。

自描述消息 (self-descriptive message)

通过查看前面的 HTML，我们会清楚看到这个网站不仅仅有一个主页。主页的标签包含两个链接：其中一个是相对路径 URL “/about”（也就是 <http://www.youtypeitwepostit.com/about>），另一个是“/messages”（也就是 <http://www.youtypeitwepostit.com/messages>）。起初，Alice 只知道一个 URL——那个 URL 指向主页——但是现在她知道三个 URL 了。服务器正在慢慢将它的结构揭示给 Alice。

我们现在可以根据服务器揭示给 Alice 的信息给网站画一张地图了（见图 1-3）。

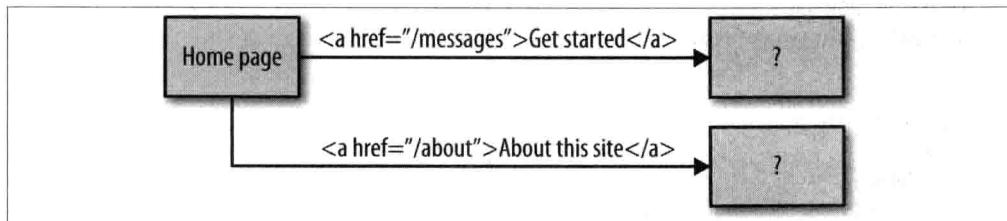


图1-3 网站地图

/messages 和 /about 这两个链接的背后是什么呢？唯一确定的办法就是单击这些链接并找到真相。但是在这之前，Alice 可以查看一下 HTML 标记或浏览器呈现的可视化页面，进而根据这些信息来做一个推测：写有“About this site”文字的链接很可能是指向一个介绍这个网站的网页，而另一个写着“Get started”的链接很可能是能够让她真真正正地发布一条消息的页面。

- 6 当你请求一个网页的时候，你收到的 HTML 文档不仅仅可以提供给你所要求的即时信息，还会帮助你来决定下一步的操作。

场景3：链接

在浏览了主页以后，Alice 决定继续对这个网站做出进一步的尝试。她很自然地单击了那个写着“Get started”的链接。任何时候，你在浏览器上单击一个链接都表示你要求浏览器发送一个 HTTP 请求。

Alice 所单击的那个链接的代码如下：

```
<a href="/messages">Get started</a>
```

她的浏览器和上次一样向同样的服务器发送了一个 HTTP 请求：

```
GET /messages HTTP/1.1  
Host: www.youtypeitwepostit.com
```

请求中的 GET 是一个 HTTP 方法（HTTP method），也就是大家所知道的的 HTTP 动词（HTTP verb）。HTTP 方法是客户端告诉服务器端它想要如何操作一个资源的方法。“GET”方法是最常见的 HTTP 方法。它的意思是“将这个资源的表述提供给我”。对于浏览器而言，GET 是默认值。当你点击一个链接，或者向地址栏输入一个 URL 的时候，你的浏览器就发送了一个 GET 的请求。

服务器处理这个特定的 GET 请求，发送了 /messages 的表述给浏览器：

```
HTTP/1.1 200 OK  
Content-type: text/html  
...  
<!DOCTYPE html>  
<html>  
  <head>  
    <title>Messages</title>  
  </head>  
  <body>  
    <div>
```

```
<h1>Messages</h1>

<p>
    Enter your message below:
</p>

<form action="http://youtypeitwepostit.com/messages" method="post">
    <input type="text" name="message" value="" required="true"
        maxlength="6"/>
    <input type="submit" value="Post" />
</form>

<div>
    <p>
        Here are some other messages, too:
    </p>
    <ul>
        <li><a href="/messages/32740753167308867">Later</a></li>
        <li><a href="/messages/7534227794967592">Hello</a></li>
    </ul>
</div>

<p class="links">
    <a href="http://youtypeitwepostit.com/">Home</a>
</p>

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

像之前一样，Alice 的浏览器将 HTML 文本进行了图形渲染（见图 1-4）。



图1-4 You Type It… “Get started” 页面

Alice 浏览这个页面以后，她会发现这个页面是一个消息列表，列表里面罗列了其他人已经发布到这个网站的消息，页面的上方还有一个引人注目的文本框和一个 Post 按钮。

现在我们获取到了这个服务器运行的更多信息，图 1-5 展示了到现在为止，Alice 的浏览器所了解到的新的网站地图。

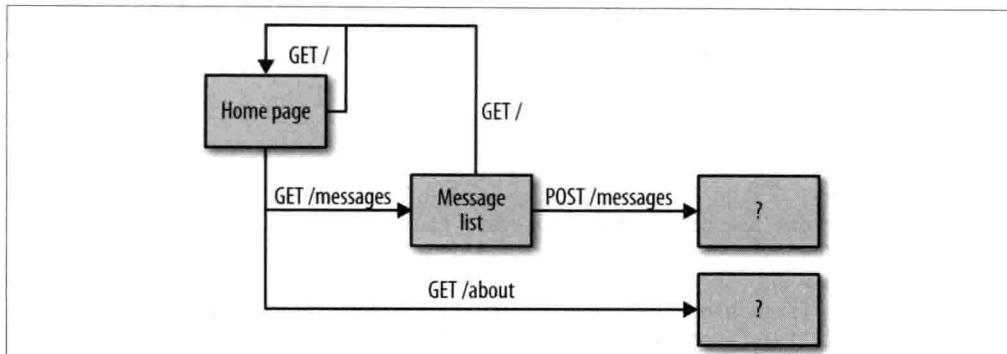


图1-5 关于You Type It...网站的浏览器的视图

标准方法

Alice 的浏览器前两次发送的 HTTP 请求都是使用 GET 作为它们的 HTTP 方法，但是在这个最新的表述里面却有一部分特殊的 HTML，它们的作用是当 Alice 单击 Post 按钮时触发一个 HTTP POST 请求：

```
<form action="http://youtypeitwepostit.com/messages" method="post">
  <input type="text" name="message" value="" required="true"
         maxlength="6"/>
  <input type="submit" />
</form>
```

HTTP 标准 (RFC 2616) 定义了客户端可以应用到一个资源上的 8 种方法。在本书中，我将重点放在其中的 5 个方法上面，它们分别是 GET、HEAD、POST、PUT 和 DELETE。在第 3 章中，我会对这些方法以及一个扩展方法 PATCH 进行更加细致的讲解。现在，最重要的事情就是记住这里有一些标准方法。

提出新的 HTTP 方法也是不可能的 (PATCH 方法就是后来提出的)，但是要付出很大的代价。在这个方面，它并不像一门编程语言：在编程语言的世界里，你可以将你的方法命名为任何东西。在我为这个例子而搭建这个简单的微博网站的时候，我并没有定义类似于 GETHOME PAGE (获取主页) 和 HELLO PLEASE SHOW ME THE MESSAGE LIST THANKS BYE (你好，请向我显示消息列表，谢谢，再见) 这样的新 HTTP 方法。我使

用 GET 方法来同时实现了“显示主页”和“显示消息列表”的目的，因为在这两个例子中，GET 方法（“将资源的表述提供给我”）最符合 HTTP 接口以及我想要的。我并不是通过定义新的方法来区分主页和消息列表页，而是把这两个文档看作不同的资源：每个资源都有它自己的 URL，每个资源都可以通过 GET 进行访问。

场景4：表单和重定向

回到我们的故事，Alice 对这个微博网站的表单很有兴趣，她在文本框中输入了“Test”，然后单击了 Post 按钮。

现在，Alice 的浏览器又发送了一个 HTTP 请求：

```
POST /messages HTTP/1.1
Host: www.youtypeitwepostit.com
Content-type: application/x-www-form-urlencoded

message=Test&submit=Post
```

服务器返回的信息如下：

```
HTTP/1.1 303 See Other
Content-type: text/html

Location: http://www.youtypeitwepostit.com/messages/5266722824890167
```

Alice 的浏览器之前发送的两个 GET 请求，服务器返回了 HTTP 状态码 200 (“OK”)，并提供了一个可用于浏览器显示的 HTML 文档。但是这一次，服务器并没有返回 HTML 文档，取而代之的是在 Location 报头里面提供了另一个 URL 链接以及在响应信息头部提供状态码 303 (“See Other”)，而不是 200 (“OK”)。

状态码 303 是告诉 Alice 的浏览器要自动向 Location 报头提供的那个 URL 发起第四个 HTTP 请求。这个过程不需要获得 Alice 的许可，浏览器仅仅执行了下面的操作：

```
GET /messages/5266722824890167 HTTP/1.1
```

这一次，浏览器返回了 200 (“OK”) 状态码以及一个 HTML 文档：

```
HTTP/1.1 200 OK
Content-type: text/html

<!DOCTYPE html>
<html>
  <head>
    <title>Message</title>
  </head>
```

```
10 > <body>
    <div>
        <h2>Message</h2>
        <dl>
            <dt>ID</dt><dd>2181852539069950</dd>
            <dt>DATE</dt><dd>2014-03-28T21:51:08Z</dd>
            <dt>MSG</dt><dd>Test</dd>
        </dl>
        <p class="links">
            <a href="http://www.youtypeitwepostit.com/">Home</a>
        </p>
    </div>
</body>
</html>
```

Alice的浏览器以可视化的方式显示了这个文档(见图 1-6),然后就继续等待 Alice 的输入。

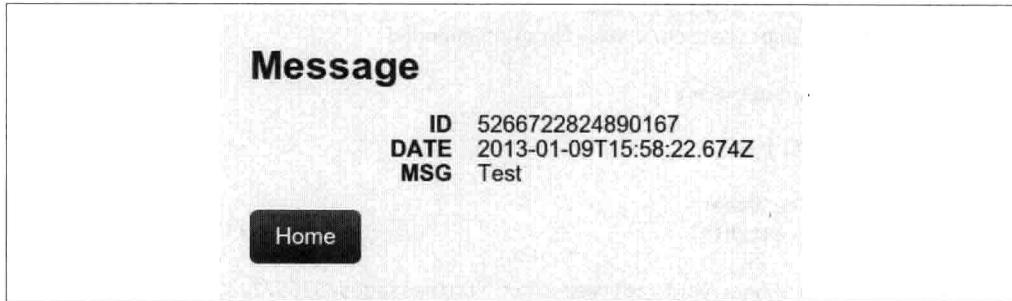


图1-6 You Type It… 所提交的消息



我可以肯定你之前一定遇到过 HTTP 重定向, HTTP 协议就包含了很多类似于 HTTP 重定向的细微的功能特性,其中有些可能你都没有接触过。我们有很多方法来让服务器告诉客户端如何采用不同的方式来处理一个响应,我们也有许多方法来让客户端将条件或者额外的功能特性添加到一个请求中。API 设计的很大一部分工作内容就是恰当地使用这些功能特性。第 11 章介绍了 HTTP 中对于 web API 非常重要的一些功能特性,并且附录 A 和附录 B 提供了关于这一主题的补充信息。

通过浏览图形化页面,Alice 看到她提交的消息(“Test”)现在已经是 YouTypeItWePostIt.com 网站上面一个正式的帖子了。我们的故事到此就结束了。Alice 已经完成了她的试用这个微博网站的目标,但是在这四次简单的交互中,还是有许多需要学习的内容。

应用状态 (Application State)

图 1-7 是一个状态图，它从浏览器的角度展示了 Alice 的完整的探索之旅。

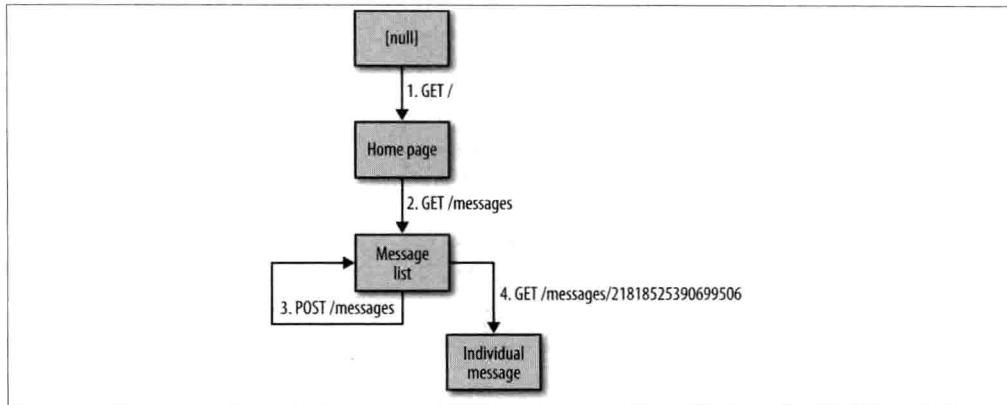


图1-7 Alice的探险：客户端的角度

当 Alice 刚启动她手机上的浏览器的时候，浏览器没有加载任何页面，界面上还是一片空白。然后，Alice 输入了一个 URL 并且通过 GET 请求将浏览器带到了网站的主页。Alice 单击了一个链接，第二个 GET 请求将浏览器带到了消息列表页面。她提交了一个表单，这触发了第三个请求（一个 POST 请求）。对应的响应是一个 HTTP 重定向，这个重定向是由浏览器自动做出的。最后，Alice 的浏览器停留在了一个显示有 Alice 发布的消息的页面。

这张图里面的每个状态都对应到 Alice 的浏览器窗口打开的一个特定的页面（或者没有页面）。在 REST 的世界里，我们将这些信息（比如你停留在哪个页面？）称为应用状态 (application state)。

当你上网的时候，你从一个应用状态转换到另一个应用状态，这个过程都对应于你单击的一个链接或者提交的一个表单。不是所有的状态之间的转换都是可用的。Alice 不能在主页直接提交一个 POST 请求，因为主页没有提供让浏览器生成 POST 请求的表单。

资源状态 (resource state)

图 1-8 是从 web 服务器的角度展示 Alice 的探索之旅的状态图。

12

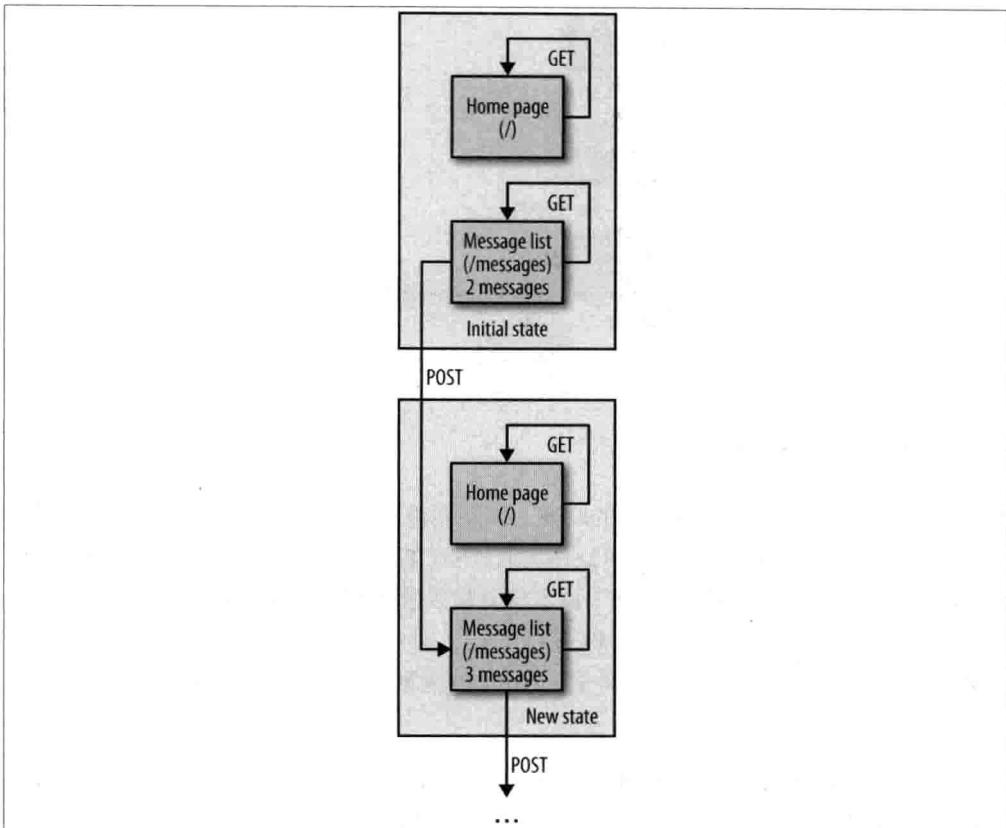


图1-8 Alice的探险：服务器的角度

服务器管理了两个资源：主页（通过“/”提供服务）和消息列表（通过“/messages”提供服务）。（服务器同时也将每条单独的消息作为资源进行管理，我为了简化状态从图中省略掉了这些资源。）简单来说，这些资源的状态就叫作资源状态。

故事开始的时候，消息列表中有两条消息：“Hello” 和 “Later”。浏览器向主页发送 GET 请求不会改变资源状态，因为主页是一个永远不会变化的静态文档。浏览器发送 GET 请求到消息列表页也同样不会改变资源的状态。

但是，当 Alice 向消息列表发送了一个 POST 请求的时候，这个请求将服务器切换到了一个新的状态。现在消息列表包含了三条消息：“Hello”、“Later” 以及 “Test”。现在已经没有办法回到原来的状态了，但是这个新的状态和原来的状态还是很相似的。和以前

一样，向主页和消息列表发送 GET 请求不会引起任何变化，但是向消息列表发送一个新的 POST 请求会使得增加第四条消息到列表里面。

由于 HTTP 会话非常短，所以服务器不知道客户端的应用状态的任何信息。客户端也不能直接控制资源状态——所有的资料都保存在服务器端。然而，网站却正常运行着。网站是通过 REST- 表达性状态移交 (representational state transfer) 工作的。

应用状态保存在客户端，但是服务器端可以通过发送表述 (representation) —— HTML 文档来操纵它。在这种情况下，提交的这个表单描述了可能的状态转换 (state transition)。资源状态是保存在服务器端的，但是客户端可以通过向服务器发送表述 (representation) —— 提交一个 HTML 表单来操作它，在这种情况下，这个提交的表单描述了客户端所期望的新的状态。

连通性 (connectedness)

在这个故事中，Alice 向 YouTypeItWePostIt.com 发送了 4 个 HTTP 请求，并且获得了 3 个 HTML 文档作为返回结果。尽管 Alice 没有一一单击这些文档的所有链接，但是我们可以使用那些链接来从客户端的角度构建一幅粗略的网站地图。

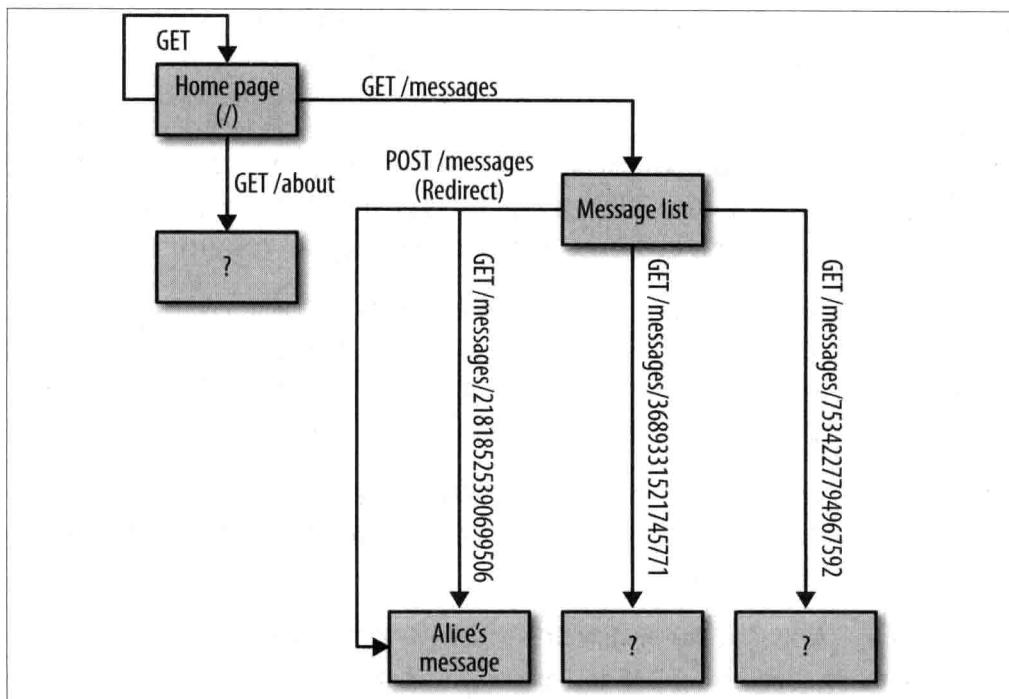


图1-9 客户端所看到的事物

这是一个由 HTML 页面组成的网。这个网间的连线是 HTML<a> 标签和 <form> 标签，它们分别描述了 Alice 可能会发起的各个 GET 或者 POST 的 HTTP 请求。我将此称为连通性原则：每个网页会告诉你如何获取相邻的网页。

- 14 网络作为一个整体按照连通性原则运转，这个原则更为人所熟知的叫法是“将超媒体作为应用状态引擎（hypermedia as the engine of application state）”，有时候简写为 HATEOAS。我更倾向于“连通性”或者“超媒体约束”，因为“将超媒体作为应用状态引擎”听起来比较吓人。但是现在，你应该没有理由害怕了。你现在已经知道什么是应用状态——简单说就是客户端当前处在哪个网页上面。超媒体是对类似于 HTML 链接、表单等的事物抽象出来的通用术语，服务器端可以通过这种技术来向客户端说明下一步的操作。

我们所说的超媒体是应用状态的引擎，其实也就是说我们都是通过填写表单以及访问各种链接来浏览 Web 的。

与众不同的Web

Alice 的故事看起来并不怎么吸引人，因为万维网在过去的 20 年间已经成为最主流的互联网应用。但是在 20 世纪 90 年代，这是一个非常令人激动的故事。如果你将万维网和其他早期的竞争者相比较的话，你会发现它们的不同之处。

Gopher 协议（在 RFC 1436 中定义）看起来很像 HTTP，但是它缺少可寻址性。它没有一个简洁的方式来标识 Gopherspace 里面的一个特定文档。最起码在万维网为 Gopherspace 考虑和发布 URL 标准（首次定义在 RFC 1738）之前并不存在这种寻址的能力。这个后来发布的 URL 标准提供了和 *http://* 类似的 *gopher://* 的 URL 方案。

FTP，这个在 web 出现之前用于文件传输的非常流行的协议（RFC959 中定义）也缺乏寻址性。在 RFC1738 定义 *ftp://* 这样的 URL 方案之前，根本没有一个机器可读的方案可以指定某个 FTP 服务器上面的一个文件。你不得不长篇大论地来解释这个文件到底在哪里。为了定位服务器上的一个文件就要花费很大的精力，这是极大的浪费。

FTP 也是一个长会话的协议。一个普通的用户可以登录到 FTP 服务器上并无限期地占用服务器的一个 TCP 连接。与之对照的是，就算是一个“持久性”的 HTTP 连接最多也不会占用一个 TCP 连接超过 30 秒。

20 世纪 90 年代见证了太多的用于检索不同类型的文档和数据库的互联网协议，像 Archie、Veronica、Jughead、WAIS 和 Prospero。但是最终证明，我们不需要所有的这些协议。我们仅仅需要的是能够向不同类型的网站发送 GET 请求。所有的那些协议都渐

渐消亡了或者被网站取而代之。它们那些复杂的特定协议的规则都合并为统一的 HTTP GET。

一旦 Web 接管了这些，就更难以证明创建新的应用协议的合理性了。为什么要在你可以搭建一个每一个人都可以使用的网站的时候，开发一种只有计算机专业人员才能理解的工具呢？所有成功的后 Web 协议（post-Web protocols）都是在做一些 Web 做不了的事情：P2P 协议比如 BitTorrent，实时协议比如 SSH。对于大部分的目的，HTTP 已经足够使用了。

Web 的这种空前的灵活性都来源于 REST 原则。在 20 世纪 90 年代，我们已经发现 Web 比其他的竞争者工作得更好。在 2000 年，Roy T. Fielding 的博士论文^{注1} 解释了其中的奥秘，并将其精简为“REST”这个术语。15

Web API 落后于 Web

Fielding 的文章也花了很大篇幅解释 21 世纪初的许多 Web API 的问题。我前面介绍的那个简单网站比当今大部分网站部署的 Web API 或者自称的 REST API 要精致得多。如果你曾经设计过 Web API，或者为这些 API 写过客户端程序，你肯定遇到过一些下面的问题。

- Web API 经常有大量的阅读文档来告诉你如何为不同的资源构造 URL。其实这就像花很大篇幅来告诉你如何在 FTP 服务器上找到一个指定的文件一样。如果网站是这样做的，没有人会愿意使用这个网站。
和每次告诉你要往浏览器输入什么 URL 不同，Web 通常都是在 <a> 和 <form> 这样的超媒体控件中嵌入 URL，你可以通过单击链接或者提交表单来激活它。
在 REST 的世界中，将有关构造 URL 的信息放到单独的阅读文档中违背了连通性和自描述信息的原则。
- 许多网站都有帮助文档，但是你上次阅读这些文档是什么时候的事情了？除非有很严重的问题（比如你想要提交一些信息，但是所有的尝试都以失败告终），更简单的方法是随便点击网站的一些链接，并通过浏览这些服务器发送给你的这些互相连接的、能够自我描述的 HTML 文档来确定网站是如何运作的。
而如今的 API 呈现资源的方式更像是一个巨型的选项菜单，而不是一张相互连通的网。这使得很难了解资源之间的相互影响。
- 要集成一个新的 API 不可避免地需要编写新的定制化软件，或者安装别人编写的一次性的代码库。但是当你要访问一个新的网站的时候，你并不需要为此而编写任何的定制化的软件。你看到广告牌上的一个 URL，你直接将这个 URL 输入到你的浏览器就可以了，对于世界上所有的网站而言，你都可以使用同一个浏览器

注 1 Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. 博士论文，加利福尼亚大学欧文分校，2000.

来访问它们。

我们不可能做到让一个 API 客户端能理解世界上所有的 API，但是当今很多的客户端包含着许多实际上应该进行重构成为通用库的代码。这只有在 API 提供了自描述的表述的前提下才有可能实现。

- 当 Web API 发生了变化以后，定制化的 API 客户端就不能正常使用了，并且需要维护者为此进行一些代码修复。但是当网站经过重新设计改版以后，用户可能会抱怨一段时间，然后慢慢适应新的版式。他们的浏览器在此期间不会停止工作。在 REST 的世界中，网站的改版是封装在服务器提供的自描述的 HTML 文档中的，所以，一个能理解旧版的文档的客户端也是能够理解新的版式的。

这些就是本书中试图去解决的一些问题。好消息是，在以前，这样的情况更加严重和恶劣。在以前，用不安全的方式使用安全的 HTTP 方法设计 REST API 或者将应用状态和资源状态混合使用的情况比比皆是。现在这些情况已经比较少见到了。现在的设计已经好多了，但是还可以更好。

语义挑战

现在该讲一下负面的消息了。正如文章前面的故事——Alice 浏览网站的故事，Alice 访问网站的过程非常顺利，这要归功于一个运行速度很慢但是又非常昂贵的硬件：Alice 本人。每次浏览器显示了一个网页，Alice，这个人类就会去浏览这个渲染过的页面，然后决定下一步的操作。Web 就是在人类不断地决定单击哪个链接以及填写哪个表单的过程中运行着。

Web API 的目标是在没有人类参与的前提下完成相应的工作。但是我们该如何编写程序让计算机来决定单击哪个链接呢？计算机可以解析 HTML 标记 `Get started`，但是它并不能理解“Get started”这个词组。如果提供的自描述信息不能被软件理解，我们又何苦设计这种提供自描述信息的 API 呢？

Web API 设计的最大的挑战就是：消除“理解文档的结构”和“理解文档的含义”之间的语义鸿沟 (semantic gap)。简单来讲，我将其称为是：语义挑战 (semantic challenge)。现在，这方面的进展非常小，我们也不可能完全解决它。好消息是到现在为止，正是由于这方面的研究进展很少和有限，所以做出一些成绩还是比较容易的。我们现在要做的是开始一起工作，而不是重复对方的工作。

因为我谈到了 Web 的技术以及如何将这些技术应用到 API 设计，所以我会在后面的几章里对语义挑战方面的内容进行介绍。在第 8 章，我们会使用一些必要的工具来直接处理语义挑战的问题。

一个简单的API

在第1章中，我展示了一个非常简单的微博网站，它的网址是 <http://www.youtypeitwepostit.com/>。与此同时，我也为该网站设计了一个可编程的 API，你可以亲临 <http://www.youtypeitwepostit.com/api/> 来访问它。

这个理想中的 API 具备了一些似曾相识的特性，而正是这些特性使得互联网简单易用。作为一名开发者，仅仅凭借在广告牌上看到的 URL，你便可以理解如何使用它。

让我们将这个想象的场景继续延伸下去，同时来看看 API 是如何工作的。首先，假设你使用你的可编程客户端程序向广告牌上的 URL 发起了一次 GET 请求——这等效于将 URL 手动敲入 web 浏览器地址栏。至此你的客户端程序将开始接管通信，并对响应中的可用选项进行检查。它随后将可能访问响应中的链接（不一定是 HTML 链接）、填充表单（不一定是 HTML 表单）并最终完成你分配给它的任务。

本书并不会带领大家完全达成上述的理想目标。有些问题是我无法在一本书内解决的，比如：围绕因为标准缺失而产生的相关问题、工具支持方面现有水平的局限性问题以及计算机无法像人类般聪明的残酷事实。但是我们会一起向着这个目标进行深入地探索——其深度将远远超乎你的想象。

我已经说过，我们已经有了一个实实在在的微博 API，它位于 <http://www.youtypeitwepostit.com/api/>。如果你是个具有探索精神的家伙，请尝试着编写一些代码来使用该 API 做些什么。看看在只知道这个 URL 的情况下，你能够对该 API 理解到什么程度。其实你在浏览各种网站时曾经完成过相同的事：你开始的时候所知道的全部信息仅仅是主页的 URL，而你却可以由此逐渐完全理解这个网站。而如今只通过一个 API，你又可以走多远呢？

如果你不愿意冒险又或者在为 web API 编写客户端程序方面没有太多经验的话（如果你在很久的未来才来阅读本书，我很可能已不再托管这个网站），我们将一起来完成这项工作。而第一步就是要获取该 API 首页的表述。

18 HTTP GET：安全的投注

如果你得到了一个以 `http://` 或 `https://` 开头的 URL，但是你并不知道该 URL 的另一头是什么，你首先可以做的就是向它发起一个 HTTP GET 请求。在 REST 的世界里，你除了知道指向资源的 URL 以外其他一无所知。你需要进一步去发现你的可选项，这意味着你需要从该资源处获取一个表述。这便是 HTTP GET 的作用所在。

你可以通过使用某种编程语言编写代码来发起 GET 请求，但是当我们只是对某个陌生 API 进行初步探测时，采用像 Wget 这样的命令行工具通常会更容易些。在下面的示例中，我使用了 `-S` 和 `-O` 两个参数选项，前者将打印出来自服务器的完整的 HTTP 响应内容，而后者将以打印出响应文档内容的方式替换原来将响应内容保存为文件的方式。^{注1}

```
$ wget -S -O - http://www.youtypeitwepostit.com/api/
```

上面的命令将向服务器发起一个如下的请求：

```
GET /api/ HTTP/1.1  
Host: www.youtypeitwepostit.com
```

HTTP 规范告诉我们，GET 请求是为了获取一个表述而作的一次请求。该种类型的请求主观上并没有去改变服务器上资源状态的意图。这就是说如果你得到了一个 URL 而没有更多信息可供参考时，你总是可以向它发起一个 GET 请求从而得到一个资源表述作为回报。你的 GET 请求将不会造成如删除所有数据这样的破坏性效果，因此我们说 GET 是一个安全的方法。

当然我们也允许服务器在处理 GET 请求时改变一些附属性的事物，例如递增计数器或将请求日志记录到文件中，但是这并非是 GET 请求原本的用途。没有人会仅仅为了递增计数器而去使用 HTTP 请求。

而在现实生活中，我们并不能保证 HTTP GET 请求是安全的。一些较早的设计会强制你使用 HTTP GET 请求来删除数据，但是这种糟糕的特性在新近的设计中相当少见。大部分的 API 设计者现在都能理解：客户端之所以频繁向 URL 发起 GET 请求只是为了看看

注1 `-O` 参数在 wget 帮助文档中的描述是将响应内容写入指定的文件，这与作者的描述看似不符。但是当没有指定输出文件名时，wget 会将内容输出到默认输出流，即命令行控制台。若不指定 `-O`，wget 会根据请求的资源名称生成默认的文件，并将响应内容写入该文件，所以这与作者的描述的效果是一致的。——译者注

该 URL 背后的内容是什么。在设计时给 GET 请求赋予重大的副作用是不合理的。

如何读取HTTP响应

为了响应我发起的 GET 请求，服务器发送了如下所示的数据块：

```
HTTP/1.1 200 OK
ETag: "f60e0978bc9c458989815b18ddad6d75"
Last-Modified: Thu, 10 Jan 2013 01:45:22 GMT
Content-Type: application/vnd.collection+json

{ "collection":
  {
    "version" : "1.0",
    "href" : "http://www.youtypeitwepostit.com/api/",
    "items" : [
      { "href" : "http://www.youtypeitwepostit.com/api/
messages/21818525390699506",
        "data": [
          { "name": "text", "value": "Test." },
          { "name": "date_posted", "value": "2013-04-22T05:33:58.930Z" }
        ],
        "links": []
      },
      { "href" : "http://www.youtypeitwepostit.com/api/messages/3689331521745771",
        "data": [
          { "name": "text", "value": "Hello." },
          { "name": "date_posted", "value": "2013-04-20T12:55:59.685Z" }
        ],
        "links": []
      },
      { "href" : "http://www.youtypeitwepostit.com/api/messages/7534227794967592",
        "data": [
          { "name": "text", "value": "Pizza?" },
          { "name": "date_posted", "value": "2013-04-18T03:22:27.485Z" }
        ],
        "links": []
      }
    ],
    "template": {
      "data": [
        {"prompt": "Text of message", "name": "text", "value":""}
      ]
    }
  }
}
```

19

通过以上内容我们能获悉到哪些信息呢？首先，每一个 HTTP 响应可以被分成 3 个部分：状态码，有时我们也称它为响应码

这部分是由三位数字组成的，它简要说明了请求目前的进展。响应码是 API 客户端从响应中最先看到的信息，它奠定了响应剩余部分的基调。以上的示例中，我们看到的状态码是 200 (OK)，这是客户端所期盼的——这意味着一切进展顺利。

在附录 A 中，我对所有的 HTTP 响应码进行了说明，同时还进行了一些有价值的扩展。

实体消息体 (entity-body)，有时我们也称它为消息体 (body)

这部分是一个采用某种数据格式书写成的文档，并且我们预期该文档是可以被客户端所理解的。如果你将 GET 请求理解成为获取表述而发起的一次请求，那么你可以将实体消息体理解为你最终得到的表述（严格来说，整个 HTTP 响应都是“表述”，但是重要的信息通常都记录在实体消息体中）。

在本例中，实体消息体是响应中最后较大那部分文档，其中充斥着很多花括号。

响应报头

响应报头的发送顺序排在状态码和实体消息体之间，通常是一系列用于描述实体消息体和 HTTP 响应的键值对。在附录 B 中，我将对所有的标准 HTTP 报头进行说明，并做了一些有助于理解的延伸。

最重要的 HTTP 报头是 Content-Type，它向 HTTP 客户端说明了如何去理解实体消息体。因为它非常重要，所以它的值都具有特定的名称。我们将 Content-Type 报头的值称为实体消息体的媒体类型 (media type)（它同样也可以被称为 MIME 类型或 *content type*，有时“media type”是可以带有连接符的：*media-type*）。

就平时人们通过浏览器就能看到的 Web 而言，最常见的媒体类型是 `text/html`（针对 HTML）以及图片类型，例如 `image/jpeg`。而在此例中的媒体类型，你或许闻所未闻：`application/vnd.collection+json`。

JSON

如果你是一个 web 开发人员，你或许已经认出该实体消息体其实是一个 JSON 文档。如果你没有，以下是一段为你准备的关于 JSON 的快速介绍。

JSON，参见 RFC 4627 中的表述，它是一种使用普通文本来表示简单数据结构的标准。它使用双引号来描述字符串：

```
"this is a string"
```

它使用方括号来描述列表：

```
[1, 2, 3]
```

它使用花括号来描述对象（键值对的集合）：

```
{ "key": "value" }
```

JSON 数据看上去非常像 JavaScript 或 Python 的代码。JSON 标准将约束建立在普通文本之上，它认为一个像 `It was the best of times` 这样的光秃秃的字符串是不能被接受的，即便任何人都能看到并理解它。要想成为合法的 JSON 数据，字符串必须用双引号括起来：“`It was the best of times.`”

Collection+JSON

21

这么说来，该实体消息体文档就是 JSON 格式了，对吗？请别着急！你可以将这个文档交由 JSON 解析器来顺利地进行解析，但是这并不是 web 服务器希望你所做的。以下是服务器对你说的：

```
Content-Type: application/vnd.collection+json
```

这便与 JSON RFC 所描述的不符了，JSON 文档应该以 `application/json` 的类型提供，像这样：

```
Content-Type: application/json
```

那么 `application/vnd.collection+json` 又是何方神圣？很明显，这种格式也是基于 JSON 的，因为它看起来很像 JSON 并且它的媒体类型名中也含有“`json`”的字样。那它到底是什么呢？

如果你尝试在 web 中搜索 “`application/vnd.collection+json`”，你将会发现它是一种注册为 Collection+JSON^{注2} 的媒体类型。当你向 `http://www.youtypeitwepostit.com/api/` 发起一个 GET 请求时，你不会得到任何的 JSON 文档——你得到的其实是一个 Collection+JSON 文档。

在第 6 章中，我将会详细地讨论 Collection+JSON，而眼下先做个简短的介绍。Collection+JSON 是一个用于在 Web 上发布资源的可搜索列表的标准。JSON 将约束建立在普通文本之上，而 Collection+JSON 则将约束建立在 JSON 之上。服务器不仅能提供像 `application/vnd.collection +json` 这样的任意 JSON 文档，它也可以只提供 JSON 对象：

注 2 Collection+JSON 是一种在该页中定义的个人标准 (<http://amundsen.com/media-types/collection>)。

```
{}
```

但是又不仅仅是一个对象，这个对象还必须拥有一个称为 `collection` 的属性，该属性可以映射到另一个对象：

```
{"collection": {}}
```

而该“`collection`”对象应该具有一个称为 `items` 的属性，该属性映射到了一个列表：

```
{"collection": {"items": []}}
```

在“`items`”列表属性中的项目也必须是对象：

```
{"collection": {"items": [{"}, {"}, {""}]}}
```

如此反复，约束相承。最终你将会得到一个如你所见的高度格式化的文档。它的开始部分如下：

```
{ "collection":  
  {  
    "version" : "1.0",  
    "href" : "http://www.youtypeitwepostit.com/api/",  
    "items" : [  
  
      { "href" : "http://www.youtypeitwepostit.com/api/messages/21818525390699506",  
        "data": [  
          { "name": "text", "value": "Test." },  
          { "name": "date_posted", "value": "2013-04-22T05:33:58.930Z" }  
        ],  
        "links": []  
      },  
      ...  
    ]  
  }
```

通过总览这个文档，所有这些约束的用途变得逐渐清晰起来。`Collection+JSON` 是提供列表的一种方式，我指的列表并非是标准 JSON 也可以提供的那种列表数据结构，而是一种描述 HTTP 资源的列表。

`collection` 对象拥有一个 `href` 属性，而它的值是一个 JSON 字符串。但是它不仅仅是一个字符串——它是我向其发起 GET 请求的 URL 地址：

```
{ "collection":  
  {  
    "href" : "http://www.youtypeitwepostit.com/api/"  
  }  
}
```

`Collection+JSON` 标准将该字符串定义为“用于获取一个文档表述的地址”（换句话说，

它是 collection 资源的 URL)。collection 的 items 列表中的每个对象都有其自己的 href 属性，并且每个值都是一个包含了一个 URL 的字符串，比如 <http://www.youtypeitweposit.com/api/messages/21818525390699506> (换句话说，列表中的每一项都代表了一个拥有 URL 的 HTTP 资源)。

一个没有遵守以上规则的文档将不能算是一个 Collection+JSON 文档：它只能算是某种 JSON。通过遵守 Collection+JSON 的约束，你将获得使用资源和 URL 这些概念的能力。在 JSON 中只能使用像字符串和列表这样的简单元素，而以上这些概念在 JSON 中是没有定义的。

向 API 写入数据

我该如何使用 API 来向微博发布一条消息呢？以下是 Collection+JSON 规范里所描述的：

如果想要在 collection 中创建一个新的 item 项，客户端首先要使用模板对象来组装一个有效的 item 表述，然后使用 HTTP POST 来向服务器发送该表述以获得处理。

以上所述并不能算是一个按部就班的描述，但是它指明了答案的方向。Collection+JSON 的工作方式与 HTML 相似，在 HTML 中，服务器向你提供了各种表单（就比如 Collection+JSON 中的模板），你可以填充这些表单来创建文档，然后使用 POST 请求来向服务器发送该文档。

再次说明，第 6 章将会具体讨论 Collection+JSON，而随后我只会快速地介绍下相关的内容。我们再来查看下我早前向大家展示的那个大对象。它的 template 属性便是在 Collection+JSON 规范中提及的“**模板对象**”。

```
{  
  ...  
  "template": {  
    "data": [  
      {"prompt": "Text of message", "name": "text", "value": ""}  
    ]  
  }  
}
```

我们来填写这个模板，我将 value 对应值中的空白字符串替换成我想要发布的内容：

```
{ "template":  
  {  
    "data": [  
      {"prompt": "Text of the message", "name": "text", "value": "Squid!"}  
    ]  
  }  
}
```

< 23

随后我将这个填充完毕的模板作为 HTTP POST 请求的一部分进行发送：

```
POST /api/ HTTP/1.1
Host: www.youtypeitwepostit.com
Content-Type: application/vnd.collection+json

{ "template":
  {
    "data": [
      {"prompt": "Text of the message", "name": "text", "value": "Squid!"}
    ]
  }
}
```

(请注意，我请求中的 Content-Type 是 application/vnd.collection+json。该模板在填充后仍是一个有效的 Collection+JSON 文档。)

服务器做出了应答：

```
HTTP/1.1 201 Created
Location: http://www.youtypeitwepostit.com/api/47210977342911065
```

此处的 201 响应码 (Created) 相比 200 (OK) 稍显特殊；它表示一切进展顺利并且在本次响应中已针对我当次请求创建了一个新的资源，而 Location 报头给出了新生资源的 URL。

24 在第1章中，Alice 曾使用 web 界面在微博网站上发过帖子。而我现在已经成功使用该网站的 web API 完成了相同的事情。

HTTP POST：资源是如何生成的

为了向 collection 添加一个新的 item，你向 collection 的 URL 发送了一个 POST 请求。这里所涉及到的不仅仅是 Collection+JSON 是如何工作的内容，这里还涉及到了 HTTP 中有关 POST 的一个基本事实。RFC 2616，即 HTTP 规范中是这样描述 POST 的：

我们将 POST 设计为一个具备如下功能的统一方法：

- 对现有资源的注解。
- 向布告栏、新闻组、邮件列表或类似的文章组发布消息。
- 向数据处理流程提供例如表单提交结果的数据块。
- 通过追加操作来扩充数据库。

其中提到的第二个重点功能，“向布告栏、新闻组、邮件列表或类似的文章组发表消息”便准确地覆盖到了微博。

我所发出的 POST 请求看上去非常像一个 HTTP 响应，因为它具有 Content-Type 报头和实体消息体。虽然我在早前所展示的 GET 请求中没有提供任何 HTTP 报头，但事实上任何 HTTP 请求都可以具有报头，甚至有好几个报头（比如 Accept）对于 GET 请求来说是非常重要的。后续我将会在这些特别重要的 HTTP 报头出现的时候进行重点讨论，但是请务必去附录 B 查阅标准 HTTP 报头的完整列表。

让我们继续往下走。再次回顾，以下是我通过 POST 请求获取的响应：

201 Created

Location: http://www.youtypeitwepostit.com/api/47210977342911065

当你收到一个 201 响应码时，后面的 Location 报头将会告诉你应该去何处找寻你刚才所创建的内容。RFC2616 详细说明了 201 响应码和 Location 报头的含义，但是为了清晰起见，Collection+JSON 规范同样也提及了这些内容。

如果我发送了如下的 GET 请求：

```
GET /api/47210977342911065 HTTP/1.1
Host: www.youtypeitwepostit.com
```

我将会看到熟悉的面孔：

```
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
{
  "collection": [
    {
      "version" : "1.0",
      "href" : "http://www.youtypeitwepostit.com/api/47210977342911065",
      "items" : [
        {
          "href" : "http://www.youtypeitwepostit.com/api/messages/47210977342911065",
          "data": [
            { "name": "date_posted", "value": "2014-04-20T20:15:32.858Z" },
            { "name": "text", "value": "Squid!" }
          ],
          "links": []
        }
      ]
    }
}
```

25

我们通过一个完整的 application/vnd.collection +json 文档来表示这个单独的微博帖子。它是一个 collection，但是它的 items 列表只含有一个列表项。而这个经过填充的模板同样也是一个有效的 application/vnd.collection +json 文档，即使它根本没有使用 collection 属性。

这是 Collection+JSON 的一个便利的特性。几乎文档中的所有内容都是可选的。这意味着你无须为了处理不同类型的文档而编写不同的解析器。Collection+JSON 使用相同的 JSON 格式来表示列表项、单独项、已填充的模板及搜索结果。

由约束带来解放

在 RESTful 设计中，一个违反直觉的经验就是：约束成为了一种解放手段。HTTP GET 方法的安全约束就是一个很好的例子。因为有了这个安全约束，当你还不知道可以对某个 URL 做些什么时，你总是可以先向它发送 GET 请求从而看看它的表述。就算最后发现这样做没有带来什么帮助，也不会产生任何糟糕的副作用，因为你所使用的是一个 GET 请求。这便是一个对解放的承诺，而唯一让这成为可能的原因是我们在服务器端制定了更加苛刻的约束。

如果服务器向你发送了一个普通文本文档，该文档中记录着 9，你将无法知晓它表示的是一个数字 9 还是一个字符串 “9”。但是如果你得到的是一个 JSON 文档，那么你可以确信文档中所记录的 9 是一个数字。JSON 标准对文档的含义进行了约束，这样便为客户端和服务器进行有意义的交流提供了可能性。

在过去的若干年里，数以百计的公司都经历过以下的思考：

1. 我们需要一个 API。
2. 我们将使用 JSON 作为文档的格式。
3. 我们将使用 JSON 来发布我们的事物列表。

26

以上的三点想法都不错，但是不足以说明我们的 API 应该是什么样的。最终的结果就是数以百计的 API 表面上很相似（因为它们都是使用 JSON 来发布事物列表的！）但却无法完全兼容。对一个 API 的学习经验无法帮助客户端去理解下一个 API。

这意味着我们需要更多的约束，而 Collection+JSON 标准为我们提供了更多的约束。如果我选择使用自定义的 API 设计而非 Collection+JSON 的话，那么在我列表中的一个独立的项将可能看上去是以下这样的：

```
{  
  "self_link": "http://www.youtypeitwepostit.com/api/messages/47210977342911065",  
  "date": "2014-04-20T20:15:32.858Z",  
  "text": "Squid!"  
}
```

反之，如果我遵守了 Collection+JSON 的约束，那么一个独立的项看上去是这样的：

```
{ "href" : "http://www.youtypeitwepostit.com/api/messages/1xe5",
```

```
"data": [
  { "name": "date_posted", "value": "2014-04-20T20:15:32.858Z" },
  { "name": "text", "value": "Squid!" }
],
"links": []
}
```

自定义设计看上去确实更加简洁，但这并不非常重要——JSON 的压缩率原本就已经很好了。在换成 Collection+JSON 这种紧凑性稍逊的表述后，我却得到了大量更有用的特性：

- 我无须再向所有的用户说明 `href` 的值是一个 URL，而且我也无须对什么是 URL 进行说明。Collection+JSON 标准已经说明了一个 item 项的 `href` 包含了指向该 item 项的 URL。
- 我无须为了向用户说明 `text` 表示的是消息的文本内容而特地编写一个供人们阅读的独立文档。这些信息会直接展现在实际需要的地方——即出现在你在投递消息时所要填写的模板中。

```
"template": {
  "data": [
    {"prompt": "Text of the message", "name": "text", "value": null}
  ]
}
```

- 任何理解 `application/vnd.collection+json` 的代码库都可以自然而然地懂得如何使用我的 API。如果我选择了一种自定义的设计，那我将只能基于仅有的 JSON 解析器和 HTTP 代码库来编写全新的客户端代码，或者让我的用户们自行编写这些代码。

◀ 27

通过采纳 Collection+JSON 的约束，我从原本需要编写的一大堆的文档和代码中解放出来，同时我也将我的用户从学习一个接一个的自定义 API 中解放了出来。

应用语义所产生的语义鸿沟

当然，Collection+JSON 约束并不能约束所有的事情。Collection+JSON 并没有指定 collection 中的 items 应该是具有 `date_posted` 字段和 `text` 字段的微博帖子。这部分的工作是我完成的，因为我想为本书设计一个简单的微博实例。如果我选择以“烹饪书”来作为实例的话，我仍然可以使用 Collection+JSON，但是 items 的字段将可能换成是 `ingredients`（配料）和 `preparation_time`（准备时间）了。

我将这些设计中的额外信息称为应用语义（application semantic），因为它们在应用之间有着很大的差别。应用语义是引起我在第 1 章中所提及的语义鸿沟的原因。

如果要设计一个真正的微博 API，我会选择远比 `text` 和 `date_posted` 更加复杂的应用语义。就该 API 自身来说，这没什么问题。但是时下有很多公司都在设计着各种微博 API，因此也冒出了很多的设计，可这些设计的应用语义是相互不兼容的，由此产生了很多不同的语义鸿沟。所有这些公司都在以不同的方式做着相同的事情，它们的用户需要编写不同的软件客户端来完成相同的任务。

了解到 Collection+JSON 无法处理语义兼容问题这一现实并不意味着我们不再需要使用 Collection+JSON，兼容性只是一个度的问题。从我们在 20 世纪 90 年代停止发明自定义的 Internet 协议并对 HTTP 进行标准化开始，我们已经向兼容性迈进了一大步。如果我们全都赞同以 JSON 文档提供服务，虽然从技术上来说这未必是个好主意，但是这样确实能缩窄我们的语义鸿沟，而标准化地采用 Collection+JSON 将会更进一步地改善这个问题。

如果微博 API 的发布者们可以走到一起并协议使用一组通用的应用语义，那么微博的语义鸿沟将荡然无存（这将可能涉及到 `profile`，我将会在第 8 章中讨论这块内容）。一旦我们共享更多的约束，我们将可以设计出更加兼容的接口，语义鸿沟也会变得更小，而我们的用户将受益更多。

28 也许你并不希望自家的 API 与对手的 API 进行互相协作，从而人为地扩大了语义鸿沟。但是在 API 差异化方面我们有比这种办法更好的方式。我在本书中的目标便是让你能专注于 API 中语义鸿沟藏身的部分，为填平鸿沟提供一些新的方式，因为从未有人尝试过这样做。

资源和表述

到现在，我已经展示了两个 REST 实例：一个是网站（见第 1 章），一个是 web API（见第 2 章）。我都是以案例的形式来展开论述的，因为对于 REST 而言，并不存在一套 RFC 标准，在这一点上，它与 HTTP 或 JSON 有所不同。

REST 不是一种协议，也不是一种文件格式，更不是一种开发框架。它是一系列的设计约束的集合：无状态性、将超媒体作为应用状态的引擎等，我们将这些约束统称为 *Fielding 约束*，因为它们最早是由 Roy T. Fielding 博士在 2000 年的时候发表的关于软件架构的论文中提出的，这篇论文将这些内容整理到一起命名为“REST”。

“REST”这个术语的受欢迎程度现在已经远远超出了其在 Fielding 的论文中的重要性。在 Fielding 的论文中，REST 主要是作为一个案例来将那些万维网中习以为常的事物与一个更具有广泛意义的设计过程连接到一起。之所以 REST 会流行，是由于这个术语描述了人类历史上最为成功的技术之一：万维网的架构。

在本章中，我会从万维网的方面来讲解 Fielding 约束。我所引用的“圣经”并不是 Fielding 的论文（你可以在附录 C 中了解更多内容，Fielding 关于以 API 为中心的讨论），而是 W3C 颁布的 Web 指南：《万维网的架构（第一卷）》(*The Architecture of the World Wide Web, Volume One*)（实际上并不存在第二卷）。Fielding 的论文解释了 Web 设计背后的决策，而《架构》一书进一步说明了发源于这些决策的三项技术：URL、HTTP 和 HTML。

我确信你知道这三项技术，但是更深层次地理解这些技术是理解 Fielding 约束的关键、是理解这些约束如何推动 Web 成功的关键，更是理解如何将这些约束应用于自己的 API 的钥匙。

潜藏在这三项 web 技术背后的两大核心理念是：资源（resource）和表述（representation）。我前面提到过它们，现在我们该仔细看看它们的庐山真面目了。

30 万物皆可为资源

就其本质而言，任何足够重要并被引用的事物都可以是资源。如果你的用户“想要建立指向它的超文本链接，提出或者反对关于它的断言，获取或者缓存它的表述，供另外的表述引用它的全部或者部分，给它增加注释信息，或者对它执行某些操作”（源自《万维网的架构》），你都应该将它定义为资源。

资源一般是可以保存到计算机里面的事物。比如电子文档，数据库的一条记录，或者一个算法的运行结果。《架构》一书将它们统称为“信息资源（information resource）”，因为它们的本质形式都是一串比特数据流。但是资源实际上可以是任何事物：比如一个石榴、一个人、黑色、勇气、母女关系、质数的集合。唯一的条件是每个资源必须拥有 URL。

你还记得那个东西吗？就是前些日子你拿着的那个，后来给……你知道我在说什么吗？你当然听不懂。因为我说的不够具体。我可能说的是任何一件东西。Web 也是一样的。客户端和服务器端只有在对事物的命名上达成一致以后才能针对这个事物相互通信。在 Web 上，我们使用 URL 来为每个资源提供一个全球唯一的地址。将一个事物赋以 URL，它就会成为一个资源。

从客户端的角度看，它并不关心资源是什么，因为它从来看不到资源，它看到的永远只是 URL 和表述。

表述描述资源状态

一个石榴可以是一个 HTTP 资源，但是你不可能通过互联网将石榴进行传输。数据库的一条记录可以是一个 HTTP 资源；事实上，它是一个信息资源，因为你可以通过互联网将它一个字符一个字符地发送出去。但是客户端如何处理这堆来自来源不明的数据库且没有上下文信息的二进制数据呢？

当客户端对一个资源发起一个 GET 请求的时候，服务器会以一种有效的方式提供一个采集了资源信息的文档作为回应。这就是表述——一种以机器可读的方式对资源当前状态的说明。石榴的大小和成熟度、数据库字段中的数据都属于这样的说明。

对于数据库中的一条记录，服务器可以用 XML 文档、JSON 对象、逗号分隔的数值或者用来生成它的 SQL INSERT 语句来描述它。它们都是合法的表述；这依赖于客户端的请求。

有的应用程序可能使用自定义的 XML 词汇表来表示一个作为商品出售的石榴。有的程序可能会用通过摄影机拍摄的石榴照片来表示一个石榴。这都依赖于具体的程序的设计。表述可以是任何机器可读的包含资源相关信息的文档。

往来穿梭的表述

在第 2 章中，我展示了一个使用 POST 请求来发布微博的客户端。随后，这个客户端发送了一个 HTTP GET 请求来获取这个微博的表述：

```
GET /api/5266722824890167 HTTP/1.1
Host: www.youtypeitwepostit.com
```

服务器返回一个 application/vnd.collection+json 格式的表述，如下：

```
HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
...
{
  "collection" :
  {
    "version" : "1.0",
    "href" : "http://localhost:1337/api/",

    "items" :
    [
      {
        "href": "http://localhost:1337/api/5266722824890167",
        "data": [
          {
            "name": "text",
            "value": "tasting"
          },
          {
            "name": "date_posted",
            "value": "2013-01-09T15:58:22.674Z"
          }
        ]
      },
      {
        "template" : {
          "data" : [
            {
              "prompt" : "Text of message",
              "name" : "text",
              "value" : ""
            }
          ]
        }
      }
    ]
  }
}
```

```
}
```

32 >

但是这条微博还有另一种形式的表述：也就是客户端起初通过 POST 请求发送给服务器端的表述。它也是一个 application/vnd.collection+json 文档，内容如下：

```
{ "template":  
  {  
    "data": [  
      {"prompt": "Text of the message", "name": "text", "value": "Squid."}  
    ]  
  }  
}
```

这两个表述看起来明显不同。其中一个是在 template 对象中包含那些微博的基本信息，另一个是在一个 items 列表中包含那些信息。但是事实很清楚，它们是同一资源的两种不同的表述，它们都是那条写有“Squid”的微博的表述。

当客户端为了创建一个新的资源而发起一个 POST 请求的时候，它会发送一个表述：客户端所期望的新资源的样式。服务器端的工作就是创建这个资源或者拒绝创建这个资源。客户端的表述只是一个建议。服务器可以增加、修改或者忽略表述的任何一部分（此处，服务器对表述数据增加了一个 date_posted 字段）。

Web 也以同样的方式工作着。回到第 1 章，故事中的虚拟主人公 Alice 通过发送一个 POST 请求在微博网站上发布了一条微博，这个 POST 请求包含了一个 application/x-www-form-urlencoded 格式的表述：

```
message=Test&submit=Post
```

这条表述看起来一点也不像 Alice 之前收到的复杂的 HTML 文档，但是它们都是那条写有“Test”微博的表述。

我们通常都认为表述是服务器发送给客户端的东西，这是由于在我们上网的时候，发送的大部分的请求都是 GET 请求，我们一直都在请求获取表述。但是实际上，在 POST、PUT 或者 PATCH 请求中，客户端也会向服务器端发送表述，服务器随后的工作就是改变资源状态，这种情况下的表述反映的是将来的表述。

服务器发送的表述用于描述资源当前的状态。客户端发送的表述用于描述客户端希望资源拥有的状态。这就是表达性状态移交。

资源有多重表述

一个资源可以有很多种表述。政府的官方文档也经常有好多个语言版本。有的资源既有整

体概括性的表述，也有面面俱全的细致化的表述。有一些 API 可以使用 JSON 和 XML 数据格式来表示同一数据。当这种情况发生的时候，客户端应该如何指定它想要的表述呢？

这里有两种策略，我将会在第 11 章对它们进行更详细的描述。第一种就是内容协商 (content negotiation)，客户端通过一个 HTTP 报头的值来区分这些表述。第二种就是为资源分配多个 URL——一个 URL 对应一种表述。

就像一个人在不同的场景下会有不同的称呼一样^{注1}，一个资源也可以由多个 URL 进行标识。当这种情况发生的时候，公开这些资源的服务器应该将其中的一个 URL 指定为“标准” URL。我将在第 11 章中介绍其中的更多细节内容。

HTTP协议语义 (Protocol Semantics)

尽管任何事物都可以成为一个资源，但是客户端并不能随心所欲地对资源进行任意的操作。所能进行的操作是有规定的。在一个 RESTful 系统里，客户端和服务器端只能通过相互发送遵循预定义协议的消息来进行交互。

在 web API 的世界里，这个协议就是 HTTP（第 13 章中的 RESTful API 架构并未采用 HTTP。）API 客户端可以通过发送一些不同类型的 HTTP 消息与 API 进行交互。

HTTP 标准定义了 8 种不同类型的消息，下面 4 个是最常用的：

GET

获取资源的某个表述。

DELETE

销毁一个资源。

POST

基于给定的表述信息，在当前资源的下一级创建新的资源。

PUT

用给定的表述信息替换资源的当前状态。

下面两个方法是客户端在分析研究 API 的时候经常使用到的：

HEAD

^{注1} “你好，Mike!”，“@mamund”（网名），“晚上好，Amundsen 先生。”

获取服务器发送过来的报头信息（不是资源的表述），这些报头信息是在服务器发送资源的表述的时候被一起发送过来的。

OPTIONS

获取这个资源所能响应的 HTTP 方法列表。

另外两个定义在 HTTP 标准中的方法：CONNECT 和 TRACE 只用于 HTTP 代理，所以在本书中我并不对它们进行介绍。

34 我建议 API 设计者还要考虑第 9 个 HTTP 方法，这个方法并没有被写进 HTTP 标准中，而是作为补充内容在 RFC 5789 中被定义的：

PATCH

根据提供的表述信息修改资源的部分状态。如果有某些资源状态在提供的表述中没有被提到，这些状态就保持不变。PATCH 类似于 PUT，但是允许对资源状态进行一些细粒度的改动。

我同时还希望你了解两个正处于标准化进程中的 HTTP 扩展方法。它们是在互联网草案“*snell-link-method*”中定义的，我会在第 11 章进行介绍以进一步说明它们的意义：

LINK

将其他资源连接到当前资源。

UNLINK

销毁当前资源和其他某些资源的连接关系。

总体来说，就是前面介绍的这些方法确定了 HTTP 的协议语义。仅仅通过查看 HTTP 请求中所采用的方法，你就大概了解客户端要做什么了：它是打算获取一个资源表述？还是删除资源？又或者是将两个资源连接到一起？

但是你还做不到完全明白它们要做什么，因为资源可以是任何事物。向一个“博客日志”发送的 GET 请求和向一个“股票代码”发送的 GET 请求是类似的。这两个请求拥有相同的协议语义，但是却有完全不同的应用语义（application semantic）。HTTP 是 HTTP，但是日志 API 不会是一个股票报价 API。

我们无法仅仅通过恰当地使用 HTTP 来满足这样的语义要求，因为 HTTP 协议并没有定义任何应用语义。但是你的应用语义必须和 HTTP 的协议语义保持一致。“获取一篇日志”和“获取一个股票报价”都应该归为“获取一个资源的表述”，所以这两个请求都应该

使用 HTTP GET。

下面的章节会针对一些使用范围比较广泛的 HTTP 方法，更加详细地介绍它们在协议语义方面的内容。

GET

你肯定已经很熟悉这个方法了。客户端会通过发送 GET 请求来获取某个 URL 所标识的资源的表述。在前面的内容中，客户端请求一条微博的表述，服务器以 application/vnd.collection+json 格式返回了这条微博：

35

```
GET /api/45ty HTTP/1.1
Host: www.youtypeitwepostit.com

HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
...
{
  "collection" :
  {
    "version" : "1.0",
    "href" : "http://localhost:1337/api/",

    "items" :
    [
      {
        "href": "http://localhost:1337/api/2csl73jr6j5",
        "data": [
          {
            "name": "text",
            "value": "Bird"
          },
          {
            "name": "date_posted",
            "value": "2013-01-24T18:40:42.190Z"
          }
        ]
      ],
      "template" : {
        "data" : [
          {"prompt" : "Text of message", "name" : "text", "value" : ""}
        ]
      }
    }
  }
}
```

我前面提到过，GET 是被定义为安全的 HTTP 方法。它仅仅是对信息的一次请求。向服务器发送一条 GET 请求对资源的影响应该和没有发送 GET 请求一样——也就是，没有任何影响。像一些日志记录、速率限制等副作用是可以接受的，但是客户端绝对不要希望发起的 GET 请求能改变资源的状态。

GET 请求中最常见的响应码是 200 (OK)。此外像 300 (Moved Permanently) 这样的重定向码也是很常见的。

DELETE

当客户端想要一个资源消失的时候，它可以发送一个 DELETE 请求。客户端这时候希望服务器将资源进行销毁，并且以后再也不会提到它。当然，服务器没有义务来删除一些它不希望删除的资源。

在下面这个 HTTP 片段中，客户端要求删除一条微博：

36 ➔ `DELETE /api/45ty HTTP/1.1
Host: www.youtypeitwepostit.com`

服务器返回的状态码是 204 (No Content)，这个状态码表示微博已经被成功删除了，现在已经没有任何关于它的信息可讨论了：

`HTTP/1.1 204 No Content`

如果一个 DELETE 请求发送成功了，收到的状态码可能是 204 (No Content, 也就是，“删除成功，我没有其他关于这个资源的信息描述了），也可能是 200 (OK, 也就是“删除成功，这里是关于它的一条消息”）或者 202 (Accepted, 也就是“我稍后将删除这个资源”）。

如果客户端试图 GET 一个已经被 DELETE 的资源，服务器会返回错误响应码，通常是 404 (Not Found) 或者 410 (Gone)：

`GET /api/45ty HTTP/1.1
Host: www.youtypeitwepostit.com`

`HTTP/1.1 404 Not Found`

幂等性 (Idempotence)

DELETE 很明显不是一个安全的方法。发送一个 DELETE 请求的效果不同于未发送 DELETE 请求。但是 DELETE 方法有另外一个很有用的属性：它是幂等的 (idempotent)。

一旦你删除了一个资源，这个资源就消失了。资源状态也就永久性地改变了。你可以再次发送一条 DELETE 请求，这时，你可能会收到一个 404 错误，但是资源状态和你第一

次发送 DELETE 请求之后的状态是一致的。资源还是不存在的。这就是幂等性。发送两次请求对资源状态的影响和发送一次请求的影响是一样的。

幂等性是一个很有用的特性，因为互联网不是一个可靠的网络。假设你发送了一个 DELETE 请求，然后你的连接超时了。由于你并没有收到响应信息，所以你无法知道前面的 DELETE 请求是否顺利完成。你只要再次发送一条 DELETE 请求并不断重试直到收到响应信息就可以了。执行两次 DELETE 请求并不比只执行一次造成更多的影响。

幂等的概念来自于数学。零乘运算是一个幂等运算。 5×0 是 0 , $5 \times 0 \times 0$ 还是 0 。一旦你将一个数值乘以零，你就可以无限次地乘以零，你得到的都是同样一个结果：零。HTTP DELETE 方法就相当于用零乘以一个资源。

乘以 1 是一个安全的运算，HTTP GET 也是一个安全的方法。你可以将一个数值不停地乘以 1，数值不会产生任何变化。每个安全的运算都是幂等的。

POST-to-Append

37

POST 是另外一个你以前应该用到过的 HTTP 方法。我会分开介绍 POST 方法的两项工作。第一种就是 *POST-to-append*，向某个资源发送一条 POST 请求用以在该资源的下一级中创建一个新的资源。当客户端发送一个 POST-to-append 请求的时候，它会在请求的实体消息体中添加所希望创建的资源的表述信息并发送给服务器。

我在第 2 章中曾经使用 POST-to-append 来通过微博 API 发布一条微博。因为我在展示 DELETE 方法的时候已经将那条微博删除了，下面我们就创建一条新的微博：

```
POST /api/ HTTP/1.1
Content-Type: application/vnd.collection+json

{
  "template" : {
    "data" : [
      {"name" : "text", "value" : "testing"}
    ]
  }
}
```

对 POST-to-append 请求而言，最常见的响应码是 201 (*Created*)。它用于告知客户端一个新的资源已经创建成功。Location 报头用于告诉客户端这个新资源的 URL 地址。另一种常见的响应码是 202 (*Accepted*)，这表示服务器打算按照提供的表述信息来创建一个资源，但是现在还没有真正创建完成。

POST 方法既不安全也不幂等，如果我发送了 5 次 POST 请求，我会收到 5 条新的微博，这 5 条微博的 text 的值都一模一样，唯一不同的是 `date_created`。

这就是 POST-to-append。但是很可能你除了曾经用 POST “创建一个新的资源”之外，你还曾用它做过各种各样其他的事情。这是 POST 的另一项工作。这项工作被称为 overloaded POST，我会在本章后面的内容中进行讨论。

PUT

PUT 请求是用于修改资源状态的请求。客户端一般会通过 GET 请求获取表述，然后对其进行修改，最后再将修改后的资源表述作为 PUT 请求的负载数据（payload）发送回去。在本节中，我将要修改一条微博的文本信息（我想要将 `text` 字段的值修改成字符串 `tasting` 以取代之前的内容）：

```
PUT /api/q1w2e HTTP/1.1
Content-Type: application/vnd.collection+json

38 >
{
  "template" : {
    "data" : [
      {"name" : "text", "value" : "tasting"}
    ]
  }
}
```

服务器可以自由地拒绝一个 PUT 请求，理由可以是多种多样的，比如实体消息类的意义不够明确、实体消息类试图修改服务器认为是只读的资源状态。如果服务器决定接受一个 PUT 请求，服务器就会修改资源状态来和客户端在表述中说明的状态保持一致，修改完成之后，通常会发送 200 (OK) 或者 204 (No Content) 状态码。

PUT 和 DELETE 一样是幂等的。如果你发送 10 次同样的 PUT 请求，请求的结果和你只发送 1 次请求的结果是一样的。

如果客户端知道新资源的 URL 的话，它同样能够使用 PUT 来新建一个资源。在下面的假定的例子中，我想要发布一条新的微博，并且恰好我还知道这条新微博的 URL：

```
PUT /api/a1s2d3
Content-Type: application/vnd.collection+json

{
  "template" : {
    "data" : [
      {"name" : "text", "value" : "Created."}
    ]
  }
}
```

```
}
```

客户端应该如何构造这个充满魔力的 URL 呢？我们将在第 4 章中对此做进一步介绍。就目前而言，我们只要注意到：即便我们用 PUT 来创建一个新的资源，PUT 也是一个幂等的操作。如果我发送 5 次 PUT 请求，这并不会相应生成 5 条具有同样内容的微博（而 5 次 POST 请求会生成 5 条相同的微博）。

PATCH

表述的信息量可能非常大。“修改表述，然后通过 PUT 方法提交”是一个简单的规则。但是如果你只是想修改资源状态中的很小的一部分，这就可能造成极大的浪费。此外，PUT 规则还可能导致与其他修改同样文档的用户发生意外的修改冲突。如果你可以仅仅向服务器发送你想要修改的那一部分数据文档，那将是非常令人高兴的。

PATCH 方法就提供了这样的功能来允许你这么做。和将完整的表述信息通过 PUT 方法发送出去不同，你可以建立一个特别的“diff”表述，并将它作为 PATCH 请求的负载数据发送给服务器。RFC 5261 介绍了一个针对 XML 文档的 patch 格式，RFC 6902 则介绍了一种针对 JSON 文档的类似格式。

```
PATCH /my/data HTTP/1.1
Host: example.org
Content-Length: 326
Content-Type: application/json-patch+json
If-Match: "abc123"

[
    { "op": "test", "path": "/a/b/c", "value": "foo" },
    { "op": "remove", "path": "/a/b/c" },
    { "op": "add", "path": "/a/b/c", "value": [ "foo", "bar" ] },
    { "op": "replace", "path": "/a/b/c", "value": 42 },
    { "op": "move", "from": "/a/b/c", "path": "/a/b/d" },
    { "op": "copy", "from": "/a/b/d", "path": "/a/b/e" }
]
```

对一个执行成功的 PATCH 请求而言，最好的响应码其实是和 PUT 以及 DELETE 请求所希望的那些响应码一样的：如果服务器想要向客户端发送数据（比如已经更新的资源表述），那么 200 (OK) 是最好的选择；而如果服务器仅仅想要表示执行已经成功，那么 204 (No Content) 就已经足够了。

PATCH 方法既不是安全的，也不能保证幂等。一个 PATCH 请求有可能结果是幂等的，这种情况下，如果你不小心对同一个文档应用了两次 patch，你可能会在第二次收到一个错误信息。但这并没有定义在相关标准中。考虑到 PATCH 的协议语义，它跟 POST 一样，

是一个不安全的操作。

需要记住的是 PATCH 方法并没有定义在 HTTP 规范中。它是最近几年（RFC5789 是在 2010 年发布的）针对 Web API 而特别设计的一个扩展方法。这也就意味着，在工具支持方面，为 PATCH 方法及其所使用的 diff 文档提供的工具不如 PUT 方法的工具支持得那样好。

LINK和UNLINK

LINK 和 UNLINK 方法用于管理资源之间的超媒体链接。为了理解这些方法，你必须首先理解超媒体和链接关系（link relation），所以我会在第 11 章进行详细讨论。在这里，我先暂时仅仅展示一些简单的例子。

下面这个 UNLINK 请求是要用于删除一个故事（story）（用 `http://www.example.com/story` 进行标识）和它的作者（author）（用 `http://www.example.com/~omjennyg` 进行标识）之间的链接的：

```
UNLINK /story HTTP/1.1
Host: www.example.com
Link: <http://www.example.com/~omjennyg>;rel="author"
```

下面这是一个 LINK 请求，用于声明另一个资源（用 `http://www.example.com/~drmilk` 标识）是这个故事资源的作者：

```
LINK /story HTTP/1.1
Host: www.example.com
Link: <http://www.example.com/~drmilk>;rel="author"
```

LINK 和 UNLINK 都是幂等的，但不是安全的。这些方法是在互联网草案（“snell-link-method”）中定义的，在草案被批准成为 RFC 之前，对它们的工具支持会比对 PATCH 方法的支持更差。

HEAD

HEAD 方法是像 GET 方法一样的安全方法，对 HEAD 方法最好的理解就是轻量级版本的 GET。服务器应该和处理 GET 方法一样处理 HEAD 方法，但是不需要发送实体消息体——只需要发送 HTTP 状态码和报头：

```
HEAD /api/ HTTP/1.1
Accept: application/vnd.collection+json

HTTP/1.1 200 OK
Content-Type: application/vnd.collection+json
```

```
ETag: "dd9b7c436ab247a7b69f355f2d57994c"  
Last-Modified: Thu, 24 Jan 2013 18:40:42 GMT  
Date: Thu, 24 Jan 2013 19:14:23 GMT  
Connection: keep-alive  
Transfer-Encoding: chunked
```

用 HEAD 来代替 GET 并不会节约任何时间（服务器还是需要生成所有的 HTTP 报头），但是它确实能够节省带宽的使用。

OPTIONS

OPTIONS 是 HTTP 的原生探索机制。一个 OPTIONS 请求的返回结果包含一个 HTTP Allow 报头，这个报头展示了这个资源所支持的所有 HTTP 方法。下面是一个 OPTIONS 请求，这个请求是发送给那条我之前通过 PUT 请求创建的微博的：

```
OPTIONS /api/a1s2d3 HTTP/1.1  
Host: www.youtypeitwepostit.com  
  
200 OK  
Allow: GET PUT DELETE HEAD OPTIONS
```

现在我就知道我下一步所能发送的 HTTP 请求的方法了。我可以 GET 这个资源的表述，用 PUT 来修改它，或者用 DELETE 来删除它。这个资源也支持 HEAD 和（当然的）OPTIONS，但是不能理解 PATCH 扩展以及 LINK 和 UNLINK。

OPTIONS 是一个很好的方法，但是很少有人使用它。设计良好的 API 会为 GET 请求发送超媒体文档（见第 4 章）作为响应，并用这个文档来宣传一个资源的功能。这些文档中的链接和表单阐明了客户端下一步所能发起的 HTTP 请求。而设计低劣的 API 使用人类可读的文档来说明客户端能发起哪种 HTTP 请求。

41

Overloaded POST

现在是时候揭示 HTTP 密室里的骷髅了。HTTP POST 方法有一个很肮脏的秘密。一个如果你做过 web 开发的工作就有可能遇到过的秘密。POST 不仅仅被用于创建新的资源。在我们用浏览器上网的时候，HTTP POST 也被用于传输任何形式的变化。它将 PUT、DELETE、PATCH、LINK 和 UNLINK 所有的方法混合成一个方法。

这是一个在 web 上可能会看到的一个 HTML 表单。这个表单的目的是编辑以前发布的微博：

```
<form method="POST" action="/blog/entries/123">  
<textarea>  
Original content of the blog post.
```

```
</textare>
<input type="submit" class="edit-post" value="Edit this blog post.">
</form>
```

在协议语义的范围内，“编辑这条微博”这个操作听起来像是一个 PUT 请求。但是 HTML 表单不能触发一个 PUT 请求。HTML 数据格式并不允许这么做，所以我们使用 POST 来进行代替。

这完全是合法的。HTTP 规范中说 POST 可以用于：

向数据处理流程提供例如表单提交结果的数据块。

这个“数据处理流程”就可以无限扩展了。它可以将任何数据作为 POST 请求的一部分发送出去，不论是出于什么目的，这都是合法的。这个定义太模糊了以至于 POST 请求实际上没有任何协议语义了。此时的 POST 并不真正表示“创建一个新的资源”，而是表示“任意而为 (whatever)”。

我将 POST 的这种“任意而为 (whatever)”用法称为重载的 (overloaded) POST。因为一个重载的 POST 请求没有协议语义，你只能在应用语义的层面上来理解它。

我会在后面的章节中讲到很多应用语义的内容，现在我仅仅要指出的是这个 HTML 表单中的应用语义。这个表单的应用语义是这个提交按钮的 CSS 类 (edit-post) 以及这个按钮上的可读标签 (“Edit this blog post”)。

那两个字符串还是不足以说明一切。直到现在，应用语义还是知道得太少，以致于我建议一点也不要使用重载的 POST。但是如果你按照我在第 8 章中的建议执行的话，你可以使用一个配置文件来可靠地向客户端传达应用语义。但是它还是不能够像协议语义那么可靠(每个发送 GET 请求的 HTTP 客户端都知道 GET 的含义)，但是我们将来会做到的。

由于重载的 POST 请求可以用来完成任何事情，所以这种 POST 方法既不是安全的也不是幂等的。某个特定的重载的 POST 可能事实上是安全的，但是从 HTTP 考虑，POST 方法是不安全的。

应该使用哪些方法？

一个 REST 系统是由各个独立的组件组合而成的：服务器、客户端、缓存、代理、缓存代理等。这些组件是由不同的人建立的，并且在开始会话之前互相是不知道对方的存在，它们只能通过 HTTP (或者类似的协议) 相互传递文档来进行通信。在进行通信之前，所有的组件都要在协议语义上达成一致，这是必需的，否则这些组件就不能理解对方的含义了。

HTTP 的协议语义很大程度上是通过 HTTP 方法定义的。但是这些方法之间还有很多的冗余。PUT 可以代替 PATCH, GET 方法可以完成 HEAD 的工作, POST 可以代替任何方法。我们真的需要所有的这些方法吗?

现实中，并不存在官方的协议语义集合。我们可以看到和听到很多关于“哪些 HTTP 方法是最好的”的好笑的争论，但是这归根到底都是某些社区成员提起的。当你选择了某些你想要使用的 HTTP 方法的时候，你也就选择了某个社区：一个理解这些方法的客户端及其他组件的社区。

对大部分的 web API 而言，我建议使用的方法有 GET、POST、PUT、DELETE 和 PATCH。但是我也记得我曾经在很多情况下建议过不同的方法：

- 在 2008 年之前，PATCH 方法还不存在。那时候，我对 web API 所建议的方法集合是 GET、POST、PUT 和 DELETE。
- 1997 年，HTTP 1.1 (RFC 2068) 规范的第一个版本定义了 LINK 和 UNLINK 方法。但是到了 1999 年，这两个方法在最终规范 (RFC 2616) 中被删除了，因为在当时并没有人使用这两个方法。

LINK 和 UNLINK 作为 HTTP 正式的协议语义的一部分的状态只维持了短短两年的时间。然后它们很快就被删除了。由于这些方法对很多的 API 而言是很有帮助的，所以互联网草案“snell-link-method”又试图将它们重新添加回来。
- WebDAV 标准（由 RFC 4981 规定，我将在第 11 章进行简要介绍）定义了 7 个新的 HTTP 方法用于让 API 处理文件系统中的文件资源。这些方法包括有 COPY、MOVE 和 LOCK。
- 当人们在通过 web 浏览器上网的时候，我们完全忽略了 HTTP 规范中定义的大部分方法，而仅仅使用 GET 和 POST 就可以顺利操作。这是由于 HTML 文档的协议语义仅仅支持 GET 和 POST。
- CoAP 协议（见第 13 章）也定义了 GET、POST、PUT 和 DELETE 方法。这些方法是根据 HTTP 方法命名的，但是它们的含义略有不同，因为 CoAP 毕竟是 HTTP。

43

如果你想要一个完全由 HTML 文档描述的 API，那么你的协议语义将被局限于 GET 和 POST。如果你想要和类似于 Web Folders(微软公司研发)这样的文件系统 GUI 程序通信，你就要使用 HTTP 和 WebDAV 扩展了。如果你想要同各式各样的 HTTP 缓存和代理进行会话，你就应该远离 PATCH 方法，远离 RFC2616 中没有定义的其他方法。

在 Web API 的世界里，有的社区比较大，有的社区比较小。如果你离开大路而去创造你自己的协议语义，你就会把自己孤立于一个只有你一个人的社区。

超媒体

到目前为止，整个故事是这样的：URL 标识出资源，客户端向这些 URL 发起一系列 HTTP 请求，然后服务器在响应中向客户端发送表述。随着时间的推移，客户端通过这些表述建立起了一个资源状态的全景图。最后，客户端发起一个意义重大的 PUT、POST 或者 PATCH 请求，将一个表述发送回服务器从而更改资源的状态。

更进一步看，你将会发现还有很多问题尚未解答：客户端如何知道应该发起哪种请求？互联网上存在着那么多的 URL，客户端怎么知道哪些 URL 背后存在着表述，而哪些将会返回给你一个 404 错误？客户端应该在它的 POST 请求中发送实体消息体吗？如果应该，那实体消息体应该是什么样的？HTTP 定义了一系列的协议语义，但是在当前 URL 所对应的服务器上又支持这些语义中的哪些呢？

解开这些谜题所缺失的那块拼图便是超媒体。超媒体将资源互相连接起来，并以机器可读的方式来描述它们的能力。通过合理地使用超媒体，便可以解决或至少是改善现今 web API 存在的可用性和稳定性问题。

与 REST 类似，超媒体并不是由某个标准文档所描述的一种单一的技术。超媒体是一种策略，并且可以由多种技术以不同的方式来实现。我将会在接下去的 3 章中讨论到数个超媒体标准，而更多标准将留待第 10 章讨论。不过在实际工作中，一切还得取决于哪些技术可以满足你的业务需求。

超媒体策略通常都拥有同样的目标。超媒体是服务器用以和客户端进行对话的一种方式，客户端从而可以知道未来将可以向服务器发起什么样的请求。它就是一个由服务器提供的菜单，客户端可以从中自由地进行选择。服务器通常都知道可能发生哪些事，但是客户端将决定实际发生什么。

其实这并不是什么新鲜的话题，我们的万维网就是以这种方式工作的，并且我们都想当然地认为它就应该是这样工作的，其他的任何方式都是一种无用的历史的倒退。但是在 API 的世界里，超媒体仍然是一个令人难以理解且富有争议的话题。这也说明了为什么如今的 API 在应对变化时还是显得如此糟糕。

在这一章里，我将揭开超媒体神秘的面纱，从而可以让你为 Web 创建出更具灵活性的 API。

将HTML作为超媒体格式

你应该已经对 HTML^{注1} 非常熟悉，所以我们从一个 HTML 实例开始讲起，下面是一个 HTML<a> 标签：

```
<a href="http://www.youtypeitwepostit.com/messages/">a
  See the latest messages
</a>
```

该标签是一个简单的超媒体控件。它是一段对 HTTP 请求的描述，用于描述浏览器在将来可以发起的一种 HTTP 请求。<a> 标签对浏览器来说就像是一种暗号，暗示它可以发起一个如下的 HTTP GET 请求：

```
GET /messages HTTP/1.1
Host: www.youtypeitwepostit.com
```

HTML 标准中指出，当用户激活一个链接时，该用户将会“访问”到该链接另一端的资源^{注2}。而实际情况是，这表示获取该资源的一个表述并将它在浏览器窗口中展现出来，从而替换掉了原来的表述（即包含该链接的那个页面）。当然，这并不是自动发生的。在你单击该链接前什么都不会发生。每个 <a> 标签都是 web 服务器的一个承诺，它确保每个确定的 URL 都命名了一个你可以访问到的资源。如果你向自己随意构造的一个 URL 发送 GET 请求，例如：*http://www.youtypeitwepostit.com/give-me-the-messages?please=true*，你将可能仅仅得到一个 404 错误。

相比于 <a> 标签，另一个 HTML 的超媒体控件 标签如下：

```

```

 标签同样描述了浏览器在不久的将来可以发起的一种 HTTP 请求，但是它并不会导致你从一个文档转移到另一个。相反，被链接资源的表述应该会以图片的形式嵌入到

^{注1} 有两个 HTML 规范你应该需要了解：HTML 4 规范和 HTML 5 规范。它们都是由 W3C 制定的开发标准。HTML 4 已经稳定存在了超过十年的时间；HTML 5 的制定工作则还在进行中。

^{注2} 见 HTML 4 规范的 12.1.1 章节。

当前的文档中。当你的浏览器发现一个 `` 标签时，它并不会提示你单击任何事物，而是自动向该图片发起一个请求。然后它会将该图片的表述整合到你当前浏览的文档中，这个过程同样不需要你的许可。

让我们来看一个更加复杂的超媒体控件——HTML 表单：

```
<form action="http://www.youtypeitwepostit.com/messages" method="post">
  <input type="text" name="message" value="" required="true" />
  <input type="submit" value="Post" />
</form>
```

这个表单描述了一个目标 URL 为 `http://www.youtypeitwepostit.com/messages/` 的请求。这和我在 `<a>` 标签中所使用的 URL 是同一个，但是 `<a>` 标签描述的是一个 GET 请求，而该表单描述的是一个 POST 请求。

这个表单不仅仅给你提供了一个 URL，它还会帮你发起一个 POST 请求。这里还有两个控件——一个 `text` 文本框和一个 `submit` 提交按钮，它们都在浏览器中被渲染成了 GUI 元素。

当你单击提交按钮时，你在文本框中输入的值和按钮本身的 `value` 属性值都会被浏览器根据 HTML 规范中指定的规则集转化为一个表述。这些规则集指定了该表述的媒体类型将会是 `application/x-www-form-urlencoded`，它看上去是这样的：

```
message=Hello%21&submit=Post
```

将上面的内容整合在一起，该 `<form>` 标签告诉你的浏览器，它可以发起一个如下的 POST 请求：

```
POST /messages HTTP/1.1
Host: www.youtypeitwepostit.com
Content-Type: application/x-www-form-urlencoded

message=Hello%21&submit=Post
```

它和 `<a>` 标签一样，服务器可以通过它来对你进行引导，只是形式上没有这么强硬。如果你不想填写这个表单，你完全可以忽略它。但是如果你愿意填写这个表单，你可以在 `message` 字段中填写任何你想要的内容（虽然服务器有可能会拒绝掉某些值）。服务器通过 `<form>` 标签告诉你，在所有你可能发起的 POST 请求中，有一个请求类型有可能会产生一些有用的东西。这就是向 `/messages` 发起的 POST 请求，它包含了一个经由表单编码 (form-encoded) 方式编码过的实体消息体，这个消息体含有 `message` 的值。

我们再讨论一个 `<form>` 标签：

```
<form method="GET" action="http://www.youtypeitwepostit.com/messages/">
<input type="text" id="query" name="query"/>
<input type="submit" name="Search"/>
</form>
```

这个表单也拥有一个你可以向其填写内容的 text 输入框，但是该表单同时也告诉你它将发起的是一个 GET 请求，而 GET 请求将不会包含实体消息体。取而代之的是，你输入文本框的数据将会整合到请求的 URL 中——同样地，这也是根据 HTML 规范中指定的规则来拼接的。

如果你填写完了这个表单，你的浏览器将会发起如下的 HTTP 请求：

```
GET /messages/?query=rest HTTP/1.1
Host: www.youtypeitwepostit.com
```

做一下总结，常见的 HTML 控件允许服务器可以向客户端描述如下 4 种类型的 HTTP 请求。

- <a> 标签描述了一个针对特定 URL 的 GET 请求，该请求只会在用户触发控件时产生。
- 标签描述了一个针对特定 URL 的 GET 请求，它将会在后台自动发起。
- 拥有 method="POST" 属性的 <form> 标签描述了一个针对特定 URL 的 POST 请求，该请求将会拥有一个由客户端构造的实体消息体。该请求只会在用户触发控件时产生。
- 拥有 method="GET" 属性的 <form> 标签描述了一个针对由客户端构造的自定义 URL 的 GET 请求。该请求只会在用户触发控件时产生。

HTML 同样也定义了很多其他稀奇古怪的超媒体控件，还有其他一些同样很生僻的可以用来定义控件的数据格式。所有这些都被归入到由 Fielding 论文给出的正式的超媒体定义中：

超媒体是由应用控制信息来定义的，而这些控制信息或内嵌在信息表达（presentation of information）之中，或作为上层凌驾于信息表达之上。

万维网到处充斥着 HTML 文档，而这些文档描述的又全都是人们喜欢阅读的事物——报价、统计数字、个人信息、散文和诗歌。但是所有这些事物都被归入到信息表述的范畴。在信息表述的概念里，Web 与一本印刷出版的图书没有太多区别。

是应用控制信息将一个 HTML 文档与图书区别开来。我现在就在讨论超媒体控件，也就是那些人们一直在与之交互但是又很少更进一步地进行审查的东西。 标签告诉浏览器嵌入某些图片，<a> 标签将最终用户传送到 Web 的另一处，而 <script> 标签则为

浏览器提供 JavaScript 以供执行。

一个包含有一首诗歌的 HTML 文档或许会拥有一个链接来指向“该作者的其他诗歌”，或者是一个让读者可以“为该诗歌评分”的表单。这便是无法在印刷出版的诗歌图书中所展现的应用控制信息。应用控制信息的存在或多或少会减弱一首诗歌在情感上所产生的影响，但是只包含一首诗歌文本内容的 HTML 文档将不算是 Web 的全面参与者，它只是在模拟一本印刷出版的图书。

URI模板

49

你使用 HTML 的 `<form>` 标签而创建的自定义 URL 只能局限在表单中。`http://www.youtypeitwepostit.com/messages/?search=rest` 这个 URL 看上去并不美观。但从技术的层面来说，这并不要紧。URL 并不需要看上去很美观，甚至不需要被人类的眼睛所理解。但是我们人类偏好美观的 URL，比如 `http://www.youtypeitwepostit.com/search/rest`。

HTML 中的超媒体控件无法告诉浏览器如何来构造一个像 `http://www.youtypeitwepostit.com/search/rest` 这样的 URL。但是另一种超媒体技术——URI 模板可以做到这一点。URI 模板由 RFC 6570 定义，它们看上去是这样的：

`http://www.youtypeitwepostit.com/search/{search}`

这并不是一个合法的 URL，因为它含有花括号。通过这些花括号，我们可以识别出这个字符串是一个 URI 模板。RFC 6570 中说明了你可以如何将这个字符串转化为无数的 URL。它规定你可以将 `{search}` 替换为任何你想要的内容，只要该字符串在 URL 内是合法内容。

- `http://www.youtypeitwepostit.com/search/rest`
- `http://www.youtypeitwepostit.com/search/RESTful%20Web%20APIs`

该 HTML 表单：

```
<form method="GET" action="http://www.youtypeitwepostit.com/messages/">
  <input type="text" id="query" name="query"/>
  <input type="submit" name="Search"/>
</form>
```

几乎等价于以下的 URI 模板：

`http://www.youtypeitwepostit.com/messages/?query={query}`

这是一种非常常见的情况，因此 URI 模板标准为带有查询字符串的 URL 定义了一种简写方式。以下的 URI 模板与前一个模板完全是等价的，同样也等价于前面的 HTML 表单：

<http://www.youtypeitwepostit.com/messages/{?query}>

URI 模板标准有着非常丰富的实例，以下是一些范例模板以及通过这些模板可以获得的一些 URL：

如果参数值按如下设置：

```
var   := "title"  
hello := "Hello World!"  
path  := "/foo/bar"
```

那么这些 URI 模板：

50 > <http://www.example.org/greeting?g={+hello}>
<http://www.example.org{+path}/status>
<http://www.example.org/document#{+var}>

将扩展为以下的 URL：

```
http://www.example.org/greeting?g=Hello%20World!  
http://www.example.org/foo/bar/status  
http://www.example.org/document#title
```

虽然，相比于 HTML GET 表单，URI 模板显得更加简短和灵活，但是这两种技术差别并不是很大。URI 模板和 HTML 表单都允许 web 服务器通过简短的字符串来描述无穷多的 URL。而 HTTP 客户端可以从无穷多的 URL 家族中选取一个，通过插入一些值来发起一个特定 URL 的 GET 请求。

URI 模板本身并没有意义，一个 URI 模板需要内嵌到一种超媒体格式中。这个概念是指每个需要该功能的标准应该只需要使用 URI 模板，而非像在 RFC 6570 发布前那个时代，都需要再制定一种自定义格式。

URI vs URL

我一直在尽可能延后这个话题的讨论，不过现在已经是时候来说明 URL（我几乎在本书的每个地方都在使用这个词）和 URI（更加一般化的一个词且常被用于技术名称中，比如 URI 模板）之间的区别了。大部分的 web API 都只专门处理 URL，本书中的 web API 也是如此。对本书大部分内容而言，它们之间的区别并不要紧，但是一旦重要起来（将会在第 12 章中出现这样的场景），也可以说是非常重要的。

URL 是一个用以标识资源的简短字符串，URI 也是一个用以标识资源的简短字符串，而每个 URL 都是一个 URI。它们都在同一个标准：RFC 3986 中进行描述。

那它们之间的区别到底是什么呢？就本书关注的主题而言：一个 URI 并不保证具有表述。

一个 URI 什么都不是，它只是一个标识。而一个 URL 是一个可以被引用 (dereferenced) 的标识。这就是说，一台计算机可以以某种方式得到一个 URL 从而获取其背后的资源。

如果你看到 `http:URI`，你就已经知道计算机如何从该目标获取表述：通过发起一个 HTTP GET 请求。如果你看到得是 `ftp:URI`，你也知道计算机如何从该目标获取表述：通过启动一个 FTP 客户端并执行某些 FTP 命令。这些 URI 其实都是 URL。它们都拥有与之关联的协议：即从这些资源获取表述的规则（指导计算机遵循的非常具体的规则）。

这是一个非 URL 的 URI：`urn:isbn:9781449358063`。它也指向一个资源：该图书的一个印刷版本。它不是该图书特定的复制品，而是该完整版本的一个抽象概念（请记住，一个资源可以是任何事物）。这个 URI 并不是 URL，因为……它的协议是什么？计算机将如何从它获取表述？你做不到。

如果没有了 URL，你将无法获取表述。如果没有了表述，也就没有了表述性状态移交。51如果一个资源没有使用 URL 来标识，该资源将无法满足多项 Fielding 约束。它同样也无法满足自描述信息的约束，因为它无法发送任何消息。一个表述可以链接到 URI 而非 URL (``)，但是这并不满足超媒体约束，因为客户端将无法访问该链接。

这里有一个标识了图书印刷版的 URL：`http://shop.oreilly.com/product/0636920028468.do`。你可以向该 URL 发送一个 GET 请求并获取到该版本的一个表述。获取到的并不是该书的实体复制品，而是一个表达了该资源某些状态的 HTML 文档：标题、书的页码数等。该 HTML 文档同样也包含有超媒体，比如该书作者的链接——并不是其人本身，而是关于他们的一些信息。由 URL 标识的资源将满足所有 Fielding 约束。

出于很多原因，我们需要使用 URI 而非 URL，我将会在第 12 章讨论资源描述策略时涉及到这些场景，但是这种情况相当罕见。通常，当你的 web API 在引用资源时，它都应该使用 http 或 https 模式的 URL，而这样的 URL 才能正常运作：它将可以在 GET 请求的响应中发送一个有用的表述。

Link报头

有这样一项技术，它可以将超媒体设置到你想象不到的地方：将超媒体放入一个 HTTP 请求或响应的报头中。RFC 5988 定义了 HTTP 的这项扩展，一个称为 Link 的报头。该报头让你可以为通常不支持超媒体的实体消息体（比如 JSON 对象和二进制图片）添加简单的超媒体控件。

下面是一个故事的普通文本表述，该故事被拆分成多个连续的部分（该 HTTP 响应的实

体消息体包含了该故事的第一部分，而 Link 报头指向了第二部分)：

```
HTTP/1.1 200 OK
Content-Type: text/plain
Link: <http://www.example.com/story/part2>;rel="next"
It was a dark and stormy night. Suddenly, a...
(剩余部分将在 part 2 中继续)
```

该 Link 报头的功能近似于 HTML <a> 标签。我建议在任何可能的情况下使用真正的超媒体格式，但是在无法选择它们的场景中，Link 报头将会非常有用。

HTTP 中的 LINK 和 UNLINK 扩展方法使用了 Link 报头。你现在应该可以理解这个来自第 3 章的例子的含义了：

52

```
LINK /story HTTP/1.1
Host: www.example.com
Link: <http://www.example.com/~drmilk>;rel="author"
```

超媒体的作用

我将在本书中提到很多的超媒体数据格式，但是只是一味地逐一向你介绍这些技术的话，恐怕对你并没有太大的帮助。我们需要退回来首先看看超媒体到底是用来做什么的。

超媒体控件承担了 3 项工作：

- 它们告诉客户端如何构建 HTTP 请求：使用什么 HTTP 方法，使用什么 URL，发送怎样的 HTTP 报头和 / 或实体消息体。
- 它们对 HTTP 响应做出承诺，给出建议的状态码、HTTP 报头以及服务器有可能在请求的响应中发送的数据。
- 它们给出客户端应该如何将响应集成到其工作流的建议。由于 HTML GET 表单和 URI 模板所完成的工作是相同的，所以让人们觉得它们很相似。它们都可以告诉客户端如何构造一个在 HTTP GET 请求中使用的 URL。

引导请求

一个 HTTP 请求具有 4 个部分：方法、目标 URL、HTTP 报头和实体消息体。超媒体控件可以引导客户端来指定所有这 4 个部分。HTML <a> 标签可同时指定请求中所使用的目标 URL 和 HTTP 方法：

```
<a href="http://www.example.com/">An outbound link</a>
```

目标 URL 被显式地定义在 href 属性中，而 HTTP 方法也被显式地定义了：HTML 规范

指出当一个最终用户单击链接时，`<a>` 标签将转变成一个 GET 请求。下面的 HTML 表单定义了一个将来可能发起的 HTTP 请求的方法、目标 URL 以及实体消息体。

```
<form action="/stores" method="get">
  <input type="text" name="storeName" value="" />
  <input type="text" name="nearbyCity" value="" />
  <input type="submit" value="Lookup" />
</form>
```

HTTP 方法和目标 URL 都被显式地定义了。实体消息体也按照一系列针对客户端的问题进行了定义。客户端需要指定它想赋给变量 `storeName` 和 `nearbyCity` 的值。如此客户端便可以构建一个经过表单编码 (form-encoded) 的服务器可以接受的实体消息体 (谁说一定要经过表单编码的？这是在 HTML 规范中 `<form>` 标签处理规则中明确定义的)。

下面的 URI 模板仅仅指定了一个 HTTP 请求的目标 URL，其他部分并没有指定：

```
http://www.youtypeitwepostit.com/messages/{?search}
```

该目标 URL 取决于那个有待填充的变量值，就像 HTML 表单产生的实体消息体取决于客户端赋予的变量值一样。客户端将使用某种算法来将 URI 模板和它的 `search` 变量的期待值转化为一个真正的 URL，比如：`http://www.youtypeitwepostit.com/messages/?search=rest`。

一个 URI 模板仅仅定义了 HTTP 请求的目标 URI。它并不会告诉你应该发起 GET 请求，还是 POST 请求，或者是任何特定类型的请求。这就是我说 URI 模板本身并没有意义以及需要将它们绑定到另一种超媒体技术上的原因。

下面的 HTML 表单告诉客户端将为 HTTP 的 Content-Type 报头设置一个特定值：

```
<form action="POST" enctype="text/plain">
  ...
</form>
```

通常，HTML POST 表单产生的实体消息体都是经过表单编码的，它们在被发送到网络时，请求的 Content-Type 报头都会被设置为 `application/x-www-form-urlencoded`。但是只要指定 `<form>` 标签的 `enctype` 属性，便可以覆盖这种默认的行为。一个拥有属性 `enctype="text/plain"` 的表单将会告诉浏览器使用普通文本格式来编码它的实体消息体，并在将其发送到网络时将 Content-Type 报头设置为 `text/plain`。

这并不是一个完美的例子，因为 `enctype` 属性只会改变 Content-Type 报头的值，它只能起到一种改变实体消息体的辅助作用。但是这是我在使用像 HTML 这样流行的超媒体格式时能想到的最好的例子了。

超媒体控件通常都允许 HTTP 客户端自由发送任何它们希望的报头，但是这种放任的态度仅仅是一种约定俗成。一个超媒体控件可以非常具体地描述一个 HTTP 请求。它可以指导客户端使用指定的 HTTP 方法，按照指定的规则构建实体消息体，提供指定的 HTTP 报头值，进而向指定的 URL 发送一个 HTTP 请求。

对响应做出承诺

下面是另一个 HTML 标签：

```

```

- 54 像 `<a>` 标签一样，一个 `` 标签将会承诺客户端可以向特定的 URL 发起一个 GET 请求。但是 `` 标签还做出了另一个承诺：服务器将会在 GET 的响应中发送某种图片的表述。

下面是另一个例子——一个来自 Atom 发布协议（Atom Publishing Protocol，我将会在第 6 章对该协议进行详细讨论）的简单 XML 超媒体控件：

```
<link rel="edit" href="http://example.org/posts/1"/>
```

这看上去足够简单；事实上，这个 `<link>` 标签可以合法地展示在一个 HTML 文档中，但是需要根据 AtomPub 标准来解释该标签，这个 `rel="edit"` 给你提供了关于 `http://example.org/posts/1` 处的资源的足够多的信息。

首先，`rel="edit"` 告诉我们在 `http://example.org/posts/1` 处的资源支持 PUT、DELETE 以及 GET 方法。你可以通过 GET 获取该资源的一个表述，然后修改该表述，通过 PUT 将它提交回服务端来改变资源的状态。这是使用 HTTP 的一个完美的典范，很多事情无须显式地声明。但鉴于大多数的 HTTP 资源都不会对 PUT 和 DELETE 做出响应，这就需要花费时间来进行沟通。

更重要的是，`rel="edit"` 意味着客户端无须再猜测向 `http://example.org/posts/1` 发送 GET 请求后会收到什么类型的表述。你将会收到在 AtomPub 中被称为成员入口（Member Entry）类型的文档（具体的细节现在并不重要——如果你想了解更多关于 AtomPub 的内容请直接跳到第 6 章）。

服务器向客户端做出了承诺：如果你发起一个 GET 请求，你将会接收到一个 AtomPub 成员入口的表述作为回报。客户端无须盲目地发送 GET 请求而只为一探 `Content-Type` 内容的究竟。客户端可以预知表述将会是 `application/atom+xml` 类型，同时客户端也可以知道有关该表述的应用语义。

工作流控制

超媒体的第三件工作便是表述资源之间的关系。下面是一个最适合解释这一点的例子，这是一个 HTML `<a>` 标签：

```
<a href="http://www.example.com/">An outbound link</a>
```

如果你在 web 浏览器中单击这个链接，你的浏览器的当前页面将会跳转到该链接 `href` 属性指定的 web 页面。而老的页面将会被废弃，最终将只会成为你浏览器历史记录中的一条记录。这个 `<a>` 标签是一个转出（outbound）的链接：当一个超媒体控件被激活时，将会以全新的状态来替换客户端当前的应用状态。

相比于 `<a>` 标签，我们来看看 HTML 中的 `` 标签：

```

```

这也是一个链接，但是它不是一个转出的链接；这是一个内嵌的链接。内嵌的链接将不会替换客户端当前的应用状态，内嵌的链接将会增强当前的页面。如果你访问了一个 HTML 中包含 `` 标签的网页，该图片将会在一个独立的 HTTP 请求中自动加载（无须你进行单击操作），并且展示在同一窗口的当前网页中。55

一个 HTML 文档不仅仅可以嵌入图片，下面的 HTML 标签将会下载并运行一些可执行的 JavaScript 脚本：

```
<script type="application/javascript" src="/my_javascript_application.js"/>
```

以下的标签将会下载一个 CSS 样式表并应用于当前的整个文档：

```
<link rel="stylesheet" type="text/css" href="/my_stylesheet.css"/>
```

以下的标签将会在当前页面中嵌入另一个完整的 HTML 文档：

```
<frameset>
  <iframe src="/another-document.html" />
</frameset>
```

所有以上这些都是嵌入性的链接，将一个文档嵌入另一个文档中的过程也叫作内嵌（transclusion）。

当然，客户端可以自由选择是否忽略服务器的引导。有一些浏览器的扩展可以阻止浏览器引用 `<script>` 标签中所引用的文件，还有一些用于增强可读性的浏览器配置选项会覆盖由样式表指定的格式指令。前面讲到的这些标签与 `<form>` 标签一样，都是给予客户端一些暗示，告诉它哪些请求可以给客户端提供所需要的内容，但是客户端通常也可以自由决定不发起任何请求。

当心冒牌的超媒体！

市场上存在着很多由深谙超媒体益处的开发者所设计的 API，但是严格说来它们并不包含超媒体。想象下面是由一个书店 API 提供的一份 JSON 表述：

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

{
  "title": "Example: A Novel",
  "description": "http://www.example.com/"
}
```

这是一本图书的表述，其中 `description` 字段的内容看上去是一个 URL：`http://www.example.com/`。但是这算是一个链接吗？可以认为 `description` 链接到一个提供了描述信息的资源吗？又或者这就是一段文本描述，而一些自作聪明的人输入了一些文本正好是一个有效的 URL 格式？

- 56 正式地说，`http://www.example.com/` 就是一段字符串。而 `application/json` 这种媒体类型并没有定义任何超媒体控件，所以即使表述的某一部分看上去很像一个超媒体链接，但事实上它并不是，那只是一段字符串。

如果你打算尝试使用这样一个 API，你将不会很教条地否定这些链接的存在。相反，你将会阅读某些由 API 提供者编写的人类可读的文档。这些文档将会做出一些约定说明，约定那些 API 提供者如何在不支持超媒体的格式（JSON）中嵌入超媒体链接。这样一来你便可以知道如何来区分链接和字符串，并且你将可以编写客户端程序来发现和访问这些超媒体链接。

但是你的客户端程序将只能为该特定的 API 工作，而你所阅读的文档也只是一份一次性的 fiat 标准。你所使用的下一个 API 将会有另一套截然不同的、用以在 JSON 中嵌入超媒体链接的约定，而你将不得不再次重复你做过的这些工作。

这就是 API 设计者不应该设计提供普通 JSON 的 API 的原因。你应该使用一种真正支持超媒体的媒体类型，你的用户将会为此而对你表示万分感谢。他们将可以使用现有的针对这类媒体类型的代码库，而不是为你的 API 重新造轮子。

JSON 在相当长的时间里一直都是 API 中最流行的表述格式，但是直到几年前，都还并不存在基于 JSON 的超媒体格式。正如你将在接下去几章中所看到的，这些已经发生了变化。请不要为你必须放弃 JSON 而担心，你即将拥有真正的超媒体。

语义挑战：我们该怎么做？

在第 1 章的末尾，我提出了一项挑战：“我们该如何编写程序让计算机来决定单击哪个链接呢？” web 浏览器的工作方式是将表述发送给人类，并由人类来做出所有的决定。如果我们在每个步骤中去除掉与人类的协商，我们又如何才能完成类似的行为呢？

向客户端提供链接便是我们在正确方向上走出的第一步。在无穷多的合法 HTTP 请求集中，超媒体文档将会马上帮你指出该站点中的哪些请求将是有用的，因此客户端无须再为此进行猜测。

但是这还不够，假设一个 HTML 文档只含有两个链接：A 和 B。客户端将可以发起两个可能的请求，那么客户端该如何选择呢？它将基于什么依据来进行决断呢？

好吧，假设这两个链接中有一个由 HTML 标签进行表述，而另一个由 <script> 标签进行表述。就 HTTP 协议而言，这两个链接本身并没有什么差别。它们都拥有相同的协议语义，它们都将分别触发一个 GET 请求，而这个请求发向预先指定好的 URL，但是这两个链接拥有不同的应用语义。在 标签另一端的表述将被作为一张图片进行展示，而 <script> 标签另一端的表述将被作为客户端代码而被执行。

< 57

对于某些客户端而言，这些信息已足够帮它做出决策。一个被设计用于抓取某网页所有图片的客户端将会访问 标签的链接而忽略掉 <script> 标签的链接。

上面的例子展示了超媒体控件可以为语义鸿沟架起桥梁。它们可以告诉客户端为什么要发起某个 HTTP 请求。

但是对于大多数客户端来说， 标签和 <script> 标签之间的区别还不足以帮助它们做出决策。“图片”和“脚本”都是非常一般的应用语义。由 HTML 描述的应用便是万维网，这是一个可用于各行各业的非常灵活的应用。

每当我考虑到应用语义的时候，我通常都会在一个更高的层次进行思考。我会区分 wiki 和在线商店之间概念上的差别。它们都是网站，它们都使用内嵌的图片和脚本，但是它们表示着不同的事物。

一种超媒体格式不需要像 HTML 那样通用，但是它可以将 wiki 或网店应用的语义定义得足够详尽。在下一章中，我将讨论那些被设计用于表述某个特定类型的问题的超媒体格式。如果将它们抽离到问题领域之外，它们将没有任何价值。但是在这些超媒体格式的领域范围内，它们将可以非常好地应对语义挑战。

领域特定设计

在本章中，我将会选择一个问题空间（problem space）并实现一个用于展示该问题空间的 web API。关于问题空间的具体细节其实并不重要。因为它们所需的技术都是相同的。所以我打算选择一个我能想到的最无聊的例子：迷宫游戏。

图 5-1 展示了一个只有一个入口和一个出口的简单的迷宫，我的服务器的工作就是创造出这样的迷宫并展示给客户端。

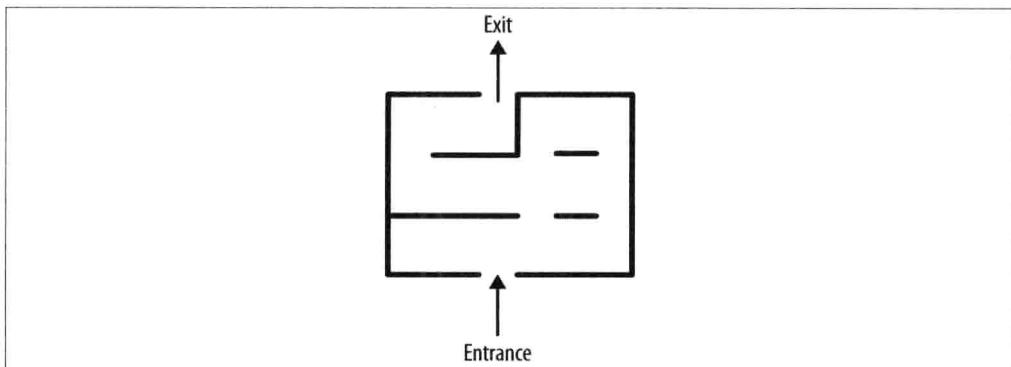


图5-1 迷宫实例（俯视图）

尽管这是一个很无聊的例子，但是对于超媒体应用程序来说，迷宫通常是一个很好的比喻。任何复杂的问题都可以表述为一个客户端必须访问的超媒体迷宫（hypermedia maze）。如果你有过以下的经历，比如曾经从杂乱无章的电话通讯录中寻找相应的联系方式，又或者你在网上商店搜索产品然后从搜索结果中购买了某些产品，你其实就已经访问过超媒体迷宫了。

我曾经见过各种各样的超媒体 API，比如用于修改复杂的保险单的 API、从目录中选择产品然后进行支付的 API 以及描述电话通讯录的 API（见第 10 章 VoiceXML）。所有的这些 API 都和我前面所展示的迷宫游戏一样：

- 问题太复杂以至于不能一次性全部理解，所以需要将问题分割为几个步骤。
- 每个客户端都是从同样的起始步骤开始处理流程的。
- 过程中的每一步，服务器都会给客户端提供若干接下来可能采取的步骤供其从中选择一个作为下一个步骤。
- 每一步，客户端都要自行决定下一步要执行的步骤。
- 客户端知道执行成功的标志是什么以及什么时候结束。

随着本书的深入，我所要处理的问题域也将更加具体。要处理的文档以及接下来可能采取的步骤也将更加复杂，但是这种按部就班的解决问题的算法是相通的，都是同样可以工作的。

Maze+XML：领域特定设计

让我们再看一下图 5-1。那是一个迷宫的图形化表述。这张图片对人类而言还有一种直观上的意义，但是对于计算机而言，计算机只有通过机器视觉算法（machine-vision algorithm）才能理解它。我们如何才能以一种便于计算机理解的方式来展示迷宫的样子呢？

解决方案有很多种，但是为了避免从零开始设计一套解决方案，我计划复用一些已经完成的工作。这里有一个称为 Maze+XML (<http://amundsen.com/media-types/maze/>) 的个人标准，我们可以通过它来以一种机器可读的格式展示迷宫。

Maze+XML 文档的媒体类型是 `application/vnd.amundsen.maze+xml`。如果你发起一个 HTTP 请求，并发现返回结果的 Content-Type 报头使用了这个字符串的话，你就会知道你现在需要一份 Maze+XML 规范说明书来全面理解收到的实体消息体了。一个领域特定的设计就是这样消除语义挑战的：定义一个用于描述问题（比如迷宫的布局）的文档格式，将这个文档格式注册为一个媒体类型，这样客户端就可以在遇到这个问题的时候立刻意识到该问题发生了。

通常来说，我不建议创建一个新的领域特定的媒体类型。将我们的应用语义添加到一种通用的超媒体类型（我会在后面的两章中对该技术进行介绍）通常可以减少很大的工作量。如果你要着手开始某个领域特定的设计工作，你很可能最终设计出的是一个没有充分利用前人的工作的 flat 标准。你可能不会拥有那些折磨了当今很多 API 的灵活性问题，但是你也做了更多的毫无价值的工作。

但是大多数的开发人员在设计 API 的时候，他们的第一直觉还是领域特定设计。有什么比简单解决手上的问题还自然的事情吗？这也就是我首先介绍领域特定设计的原因。展示一个能够消除语义鸿沟的自定义的超媒体格式还是比较容易的。

Maze+XML是如何工作的

我们不再像图 5-1 那样从上往下看迷宫，想象一下我们就身处迷宫内部。这时，你看到的就不是迷宫的全貌了，你能看到的仅仅是身边周围的事物。从迷宫的入口进入，你会看到如图 5-2 所示的一样的场景：你的面前是一堵墙，你的背后是迷宫的入口。你现在有两个选择：向左走还是向右走。你没有办法知道哪个方向通向出口。

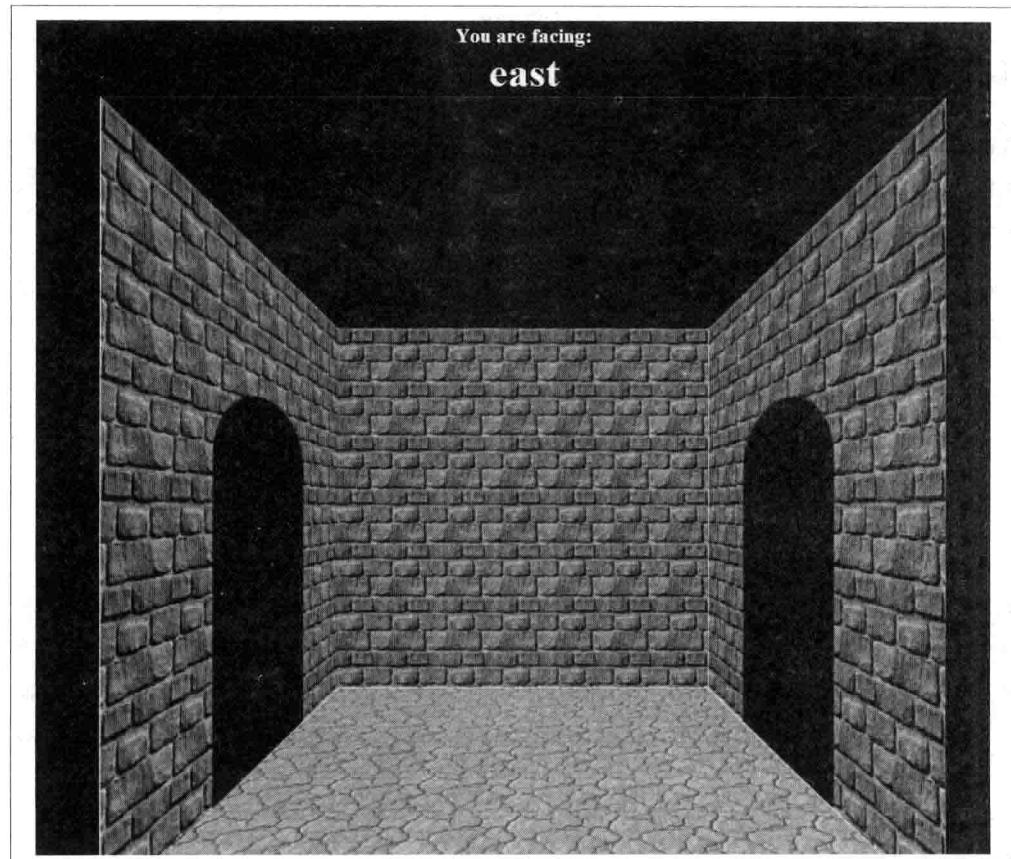


图 5-2 迷宫实例的内部

Maze+XML 格式通过模拟小鼠的视角 (rat's-eye-view) 将迷宫表示为一个由相互连接的单元格构成的网络。图 5-3 展示了如何将图 5-1 的迷宫实例表示成一个由单元格组成

的网络。我将迷宫抽象成一个网格，并为每个小方格创建一个单元格。

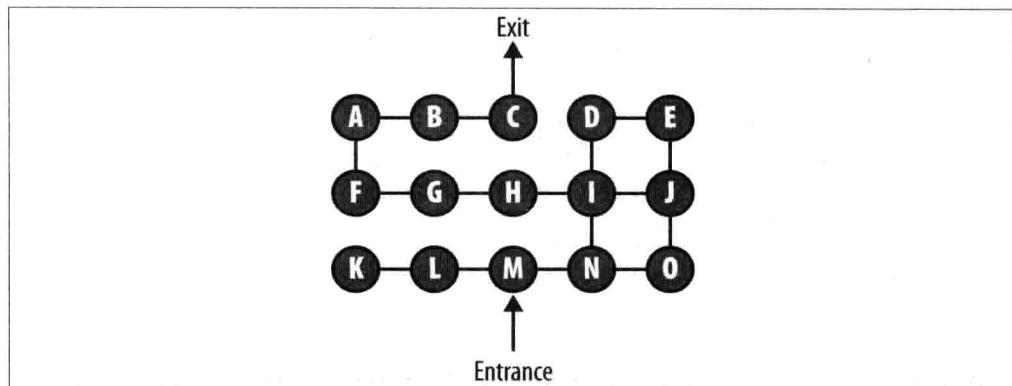


图5-3 迷宫实例——单元格的网络

Maze+XML 的单元格从 4 个方向相互连接：上北、下南、左西、右东。这就表示出口（单元格 C）在人口（单元格 M）的正北方，虽然你不能直接向北走找到出口——你需要先向东走绕个弯。

Maze+XML 迷宫里的每个单元格都是一个拥有 URL 的 HTTP 资源。如果你向这个迷宫中的第一个单元格发送一条 GET 请求，你就会收到一条表述，内容如下：

```
<maze version="1.0">
<cell href="/cells/M" rel="current">
<title>The Entrance Hallway</title>
<link rel="east" href="/cells/N"/>
<link rel="west" href="/cells/L"/>
</cell>
</maze>
```

这条表述包含了一个人类可以理解的单元格的名字：“he Entrance Hallway”，这就和你曾经在那种很老的、文字界面的冒险类游戏里面见到的一样。除此之外，超媒体也在这里出现——这条表述还包括 `<link>` 标签，这些标签用来将这个单元格与它附近的其他单元格连接起来。从单元格 M 开始，你可以选择向西走进入单元格 L，也可以选择向东走进入单元格 N。

链接关系

上面这条表述展示了一个很强大的超媒体工具，我们将它称为链接关系。靠它们自己，`rel="east"` 和 `rel="west"` 不能表示任何含义。计算机也不会理解单词“east”和“west”的意思。但是 Maze+XML 标准为“east”以及“west”定义了明确的含义。

east

指的是当前资源东边的资源。当被用于 Maze+XML 媒体类型的时候，所关联的 URL 指向的是当前迷宫中东边邻近的单元格资源。

west

指的是当前资源西边的资源。当被用于 Maze+XML 媒体类型的时候，所关联的 URL 指向的是当前迷宫中西边邻近的单元格资源。

这些定义看起来好像是一种循环定义。它们仅仅是说这些链接关系 `east`、`west` 和我们平常所说的地理概念的名称是相同的。但是这些定义有助于消除语义鸿沟，这是因为，还是那句话，链接关系靠它们自己并不能表示什么含义。如果没有一个正式的定义，`east` 可以表示的是南方，而 `west` 也可以表示的是地下。

Maze+XML 标准也同样定义了链接关系：`north` 和 `south`。通过这些定义，我们可以预料到这些方向链接将会在某些 Maze+XML 表述中出现，我们也可以编写程序让计算机在遇到这样标记的时候能够理解它们：

```
<link rel="east" href="/cells/N"/>
```

在 Maze+XML 中，访问一个标有链接关系为 `east` 的链接就会使得你的客户端穿过某些抽象的地理空间移向东边。你之后就处在迷宫的另一个单元格中了。从你当前资源的所在位置中移动到东边的资源这一行为类似于你在现实生活中移动到东边，或者至少类似于你在现实生活中将手指在地图上滑动到东边。Maze+XML 就是这样解决语义挑战的：通过定义链接关系来传递应用语义。

链接关系是一个和超媒体控件（比如 Maze+XML 的 `<link>` 标签）相关的很充满魔力的字符串。链接关系说明了应用状态（对安全的请求而言）以及资源状态（对非安全的请求而言）的变化。这种变化是在客户端触发控件的时候发生的。链接关系是在 RFC5988 中正式定义的，但是这种思想其实已经存在很长时间了，几乎所有的超媒体格式都支持它们。

对于一个 RESTful API 的开发人员来说，其中一个最重要的网页就是由互联网编号分配机构（Internet Assigned Numbers Authority——IANA）(<http://www.iana.org/assignments/link-relations>) 管理的链接关系注册表。我会在后面章节中对其进行讲解。它包含了 60 种已经被认为具有通用性的链接关系，并且这些链接关系不是针对某种特定的数据格式的。最简单的例子就是用于访问列表的 `next` 和 `previous` 关系。Maze+XML 规范中的 `east` 和 `west` 并不在这 60 个链接关系中；它们被认为通用性还不够高。

RFC 5988 定义了两种链接关系类型：注册关系类型（registered relation types）和扩展关系类型（extension relation types）。注册链接关系看起来很像你在 IANA 注册表中见到的链接关系：类似于 `east` 和 `previous` 这样的简短的字符串。为了避免冲突，这些短字符串需要在某些地方进行登记——不一定是 IANA，也可以是在某些标准中，比如媒体类型的定义文档中。

扩展关系看起来像 URL。如果你拥有一个域名 `mydoma.in`，你可以将一个链接关系命名为 `http://mydoma.in/whatever`，然后将它定义为你任何想要表示的内容。没有人能够定义一个和你的关系相冲突的链接关系，因为这个域名是由你来管理的。当你的用户在他们的浏览器上访问 `http://mydoma.in/whatever` 的时候，他们应该会看到这个链接关系的人类可读说明^{注1}。

第 9 章包含一个指导来说明什么时候可以使用这种缩写的注册关系。下面是摘要：

- 任何地方你都可以使用扩展关系。
- 任何时候你都可以使用在 IANA 注册的链接关系。
- 如果一个文档的媒体类型定义了一些注册关系，你可以在这个文档中使用它们。
- 如果一个文档包含了一个定义了一些链接关系的 profile（见第 8 章），你可以在这个文档中将这些链接关系作为注册关系进行使用。
- 不要让你的链接关系的名称和 IANA 注册表中的名称发生冲突。

访问链接来改变应用状态

客户端可以通过访问一个恰当的链接（也就是：向标记为 `rel="east"` 的 URL 发送一个 GET 请求）从单元格 M“向东走”。客户端之后就会收到第二个 Maze+XML 格式的表述，内容如下：

```
<maze version="1.0">
<cell href="/cells/N">
  <title>Foyer of Horrors</title>
  <link rel="north" href="/cells/I"/>
  <link rel="west" href="/cells/M"/>
  <link rel="east" href="/cells/O"/>
</cell>
</maze>
```

这是地图里面的单元格 N 的 Maze+XML 格式的表述。它链接回单元格 M（使用链接关

^{注1} 如果你需要将某个 IANA 注册链接关系格式化为扩展关系，你可以使用这个 URI 模板 `http://alps.io/iana/relations#{name}`。对于链接关系 `author` 的可选名称就是 `http://alps.io/iana/relations#author`。这是我们提供的一个服务，它是 ALPS 项目的一部分，并不是 IANA 官方或者授权的内容，我将在第 8 章中进行描述。

系 west)，此外它还链接到单元格 I (north) 和单元格 O(east)。

客户端的应用状态也就因此发生了变化。借用 HTML 标准中的术语，客户端刚才在“访问”单元格 M，现在它正在“访问”单元格 N。现在客户端有了 3 个新的选项，这些选项是通过单元格 N 的表述中的 3 个链接来表示的。

通过访问右边的链接（北、西、西、西、北、东、东，最后继续向东），客户端可以从单元格 N 到达单元格 C。单元格 C 包含了迷宫的出口，这个出口是通过一个链接关系为 exit 的 `<link>` 标签来表明的：

```
<maze version="1.0">
  <cell href="/cells/C">
    <title>The End of the Tunnel</title>
    <link rel="west" href="/cells/B"/>
    <link rel="exit" href="/success.txt"/>
  </cell>
</maze>
```

下面是 Maze+XML 标准对 `exit` 的说明：

`exit`

指的是表示客户端的当前所在活动或流程的出口或者终点的资源。当被用于 Maze+XML 媒体类型时，所关联的 URL 指向当前迷宫的最终的出口资源。

不同于 `east` 和其他的方向关系，Maze+XML 并没有提供指导来说明在 `exit` 链接的另一端应该出现什么事物。它是一个“资源”，也就意味着，它完全可以是任何的事物。在本例中，我选择将其链接到一个本文类型的祝贺消息 (`success.txt`)。

迷宫集合

单元格 C 通往迷宫的外部，因为它的表述中包含一个 `rel="exit"` 的特殊链接。但是单元格 M 作为迷宫的入口，却没有包含任何将它与其他 14 个单元格区分开来的内容。表述中也不存在 `rel="entrance"` 或者其他的内容。单元格 M 的标题是“*The Entrance Hallway*”，但是这个词组对计算机而言没有任何意义。我们如何才能消除这个语义鸿沟呢？客户端应该如何知道迷宫的起点在哪里呢？

Maze+XML 标准是通过一个集合：迷宫列表来解决这个问题的。如果你向迷宫 API 的根 URL 发送一个 GET 请求，你会收到如下 Maze+XML 格式的表述：

```
<maze version="1.0">
<collection>
  <link rel="maze" title="A Beginner's Maze" href="/beginner">
    <link rel="maze" title="For Experts Only" href="/expert-maze/start">
  </collection>
</maze>
```

66 Maze+XML 文档中的一个集合就是一个 `<collection>` 标签，这个标签包含了一些链接关系为 `maze` 的链接。这个关系（在 Maze+XML 规范中定义，就像 `east` 和 `exit` 一样）会告诉客户端，这个链接的另一端的资源是一个迷宫的起始单元格。这个表述链接到两个迷宫：初学者迷宫（我用图 5-1 表示）以及一个更加复杂的迷宫（我现在暂时不会展示出来）。

向链接关系标记为 `maze` 的 URL（我们暂时使用 `/beginner`）发送一条 GET 请求，你就会收到一条如下内容的表述：

```
<maze version="1.0">
<item>
  <title>A Beginner's Maze</title>
  <link rel="start" href="/cells/C"/>
</item>
</maze>
```

这是从外部查看这个迷宫得到的一个高层次的表述。它得到了一个链接关系为 `start` 并且指向单元格 C 的链接。

Maze+XML 文档就是以此来表示 `/cells/C` 是迷宫入口这一事实。它在处在迷宫的外部视图里。一旦你进入了迷宫，单元格 C 也就没有任何特殊之处了。

指向迷宫集合的 URL 也就是那个众所周知的“广告牌上推广的 URL”。只从这个 URL 开始，所有用 Maze+XML API 能完成的事情，你都可以做了：

1. 首先获取迷宫集合的表述。因为你已经阅读过 Maze+XML 规范并将这些知识编程写进了你的客户端，所以你知道如何解析这个表述。
2. 你的客户端还知道链接关系 `maze` 表示一个迷宫。这样客户端就得到了一个能在第二次 GET 请求中使用的 URL。通过发送这个 GET 请求，你可以得到一个迷宫的表述。
3. 你的客户端知道如何解析一个迷宫的表述（因为你已经将这些知识编码到客户端里面了），然后它就知道了链接关系 `start` 表示这个迷宫的入口。这样你就可以发起第三次 GET 请求来进入迷宫了。
4. 你的客户端知道如何解析一个迷宫单元格的表述。它知道 `east`、`west`、`north`

和 `south` 的含义。所以，它可以将在抽象的迷宫里的移动过程转换为一系列的 HTTP GET 请求。

5. 你的客户端知道 `exit` 的含义，所以它知道什么时候走出了迷宫。

Maze+XML 标准还有很多其他的内容，但是你现在已经了解了它的基本内容。一个集合链接到一个迷宫，一个迷宫链接到一个单元格。你可以从一个单元格访问链接到另一个单元格的链接。最终，你会发现一个通向迷宫外部的标记为 `exit` 的链接的单元格。这些信息已经足够开始编写客户端了。

67

Maze+XML 是 API 吗？

如果你现在已经从这个领域中得到一些经验，你可能想要知道：API 在哪里？迷宫游戏不是一个复杂的应用程序，但是虽然如此，你所希望的可能还不仅仅是有一些 XML 标签名以及链接关系。Maze+XML 规范并没有那些你以为常的东西。规范没有定义任何的 API 调用或者提供任何关于构造 URL 的规则。事实上，它几乎没有提到过 HTTP。我已经在例子表述中展示了一些 URL，但是我故意没有保持 URL 格式设计的内部一致性（比较 `/beginner` 和 `/expert-maze/start`），所以你就不会认为 URL 格式是由 Maze+XML 标准定义的了。

你已经习惯了的东西是很危险的。在供某个组织使用的应用程序中，基于 API 调用的设计会效果很好并且易于开发。API 调用的说法假设打破了网络边界，并且允许客户端像调用本地代码库的 API 一样来调用远程计算机的方法。已经有很多的书和软件工具来帮助你进行那样的设计。

我的经验告诉我“API 调用”的说法不可避免地会将服务器的实施细节暴露给客户端。这就引入了服务器代码与客户端代码的耦合。当涉及这些 API 的人们都是自己的朋友和同事的时候，影响还不大。

但是本书关注的是 web API，也就是说，web 规模的 API（任何公共成员都可以在 web 上使用客户端、编写客户端，或者某些情况下，还可以编写服务器）。当你允许你所在组织之外的某个人发起 API 调用的时候，你也就使得这个人成为了你的服务器实现中的一个无声的搭档。这也就使得在不伤害这个未知客户的情况下对服务器端进行任何改动都变得异常困难。

这就是公共 API 的改动相当罕见的原因。你无法在不给用户造成巨大痛苦的情况下改变基于 API 调用的 API，你仅仅能做到的是改变本地代码库的 API 而不造成痛苦。在 web 级的规模上，API 调用的设计方式是无能为力的。

基于超媒体的设计拥有更多的灵活性。每次客户端发送一个 HTTP 请求，服务器就会发送响应来说明哪些 HTTP 请求作为下一步的选择是最合理的。如果服务器端的选项发生了变化，那么那个响应文档也就会相应发生变化。这不能解决我们所有的 API 问题——语义鸿沟还是一个很大的问题！——但是它至少解决了那些我们已经知道如何解决的问题。

68

客户端1：游戏

Maze+XML API 的一个很明显的用途就是开发供人类玩乐的游戏。这里有一个单页应用程序，这个程序可以获取一个迷宫集合供你从中选择一个迷宫来玩。一旦你进入了一个迷宫，你所看到视图就像一只老鼠看到的视图一样，你可以通过输入方向来游览这个迷宫；一旦你找到了出口，你就得到一个分数——你在迷宫中移动的次数^{注2}。

我们倾向于将“API 客户端”理解为一个自动化的客户端。但是像上面例子一样的由人类驱动的客户端在现代的 API 生态系统里也起了很大的作用。这在移动应用中很常见，这些移动应用就是由人类来驱动进而通过 web API 和服务器进行通信的。最好的办法就是，将人类添加到环节里面，这样前面所讨论的语义鸿沟也就不是问题了。

图 5-4 展示了游戏客户端加载进浏览器以后的界面。

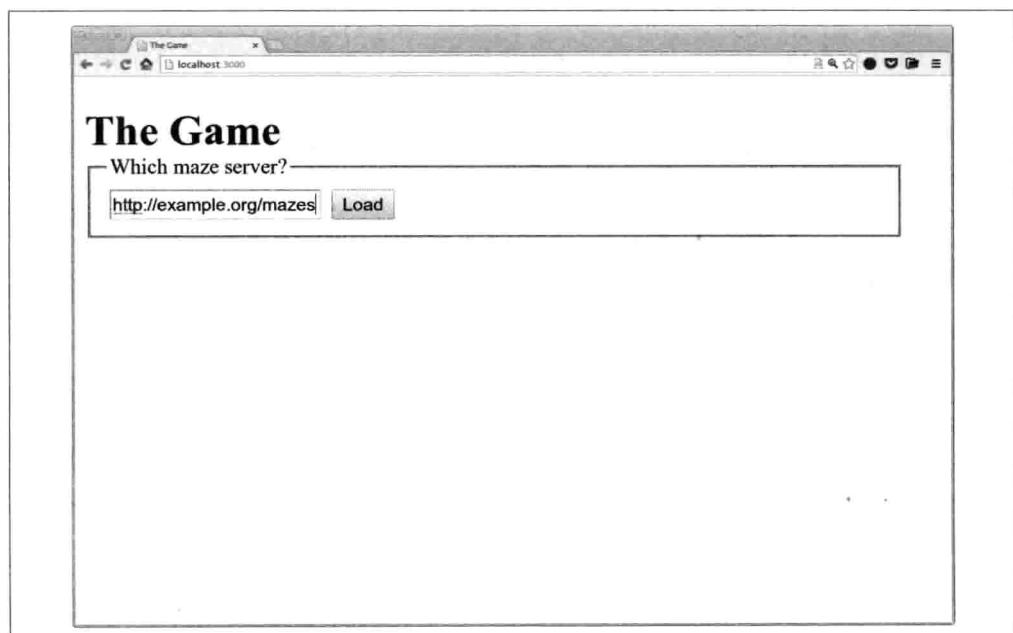


图5-4 游戏客户端的初始状态

注2 你可以在 RESTful Web APIs 的 Github 仓库中找到游戏客户端的 Node 源码(查看 *Maze/the-game/* 目录)。

我输入一个广告牌 URL——也就是那个迷宫集合的 URL——单击 Load 按钮。游戏客户端就会向我输入的那个 URL 发起一次 HTTP GET 请求：

```
GET /mazes/ HTTP/1.1
Host: example.org
Accept: application/vnd.amundsen.maze+xml
```

服务器以 Maze+XML 格式的文档做出响应：

```
<maze version="1.0">
  <collection href="http://example.org/mazes/">
    <link href="http://example.org/mazes/a-beginner-maze" rel="maze"
          title="A Beginner's Maze" />
    <link href="http://example.org/mazes/for-experts-only" rel="maze"
          title="For Experts Only" />
  </collection>
</maze>
```

游戏客户端读取这个文档——迷宫集合的表述，然后将其翻译到 HTML 界面（见图 5-5）。我就会看到两个迷宫选项。它们分别对应 Maze+XML 文档中链接关系为“maze”的两个链接。



图5-5 迷宫选项

我在文本框中输入“1”来选择一个迷宫，然后单击 Go 按钮，这样我就进入了初学者迷宫的内部了。图 5-6 展示了客户端是如何呈现迷宫的第一个宫格的。

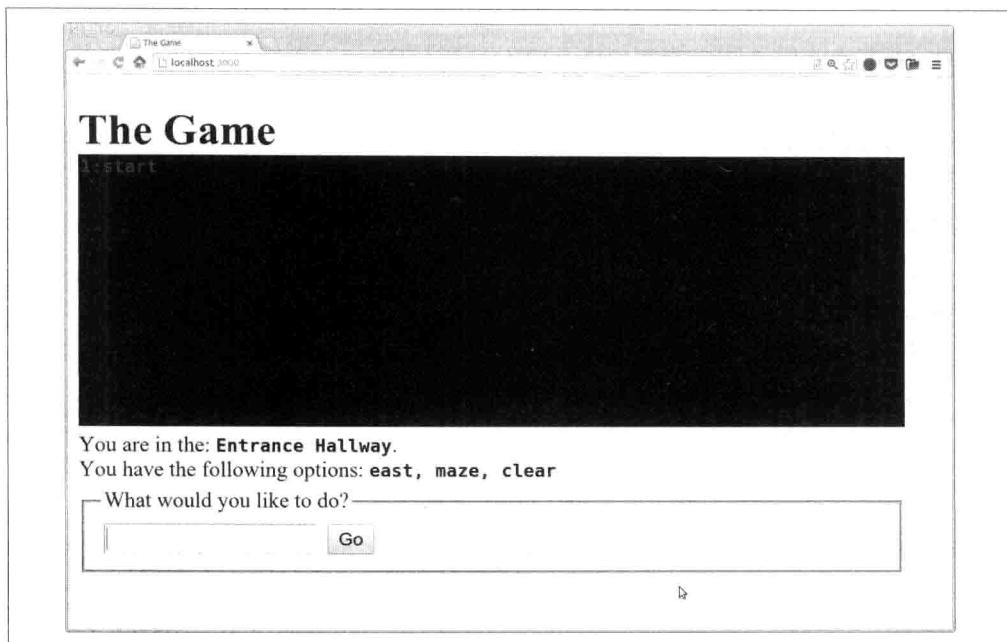


图5-6 初学者迷宫的第一个单元格

这是如何发生的呢？是通过超媒体。当我输入“1”的时候，我就是告诉客户端让其发起一个HTTP GET请求来访问那第一个rel="maze"的链接。这个接口与单击链接的接口有很大的不同，但是它们的效果是相同的。

请求内容如下：

```
GET /mazes/beginner HTTP/1.1
Host: example.org
Accept: application/vnd.amundsen.maze+xml
```

下面是服务器发送的Maze+XML格式的文档作为响应：

```
<maze version="1.0">
  <item href="http://example.org/maze/beginner" title="A Beginner's Maze">
    <link href="http://example.org/mazes/beginner/0" rel="start"/>
  </item>
</maze>
```

由于在这个文档中只有一个链接：指向迷宫起点的链接，人类没有其他选择可选，所以不需要做出决策。游戏客户端甚至不会将这个表述显示出来，取而代之的是，客户端会按照预先编写好的程序的设定而自动访问那个rel="start"的链接。这就表示发起一个新的GET请求：

```
GET /mazes/beginner/0 HTTP/1.1
Host: example.org
Accept: application/vnd.amundsen.maze+xml
```

这个 GET 请求会收到这个迷宫中的一个单元格的表述：

```
<maze version="1.0">
  <cell href="http://example.org/mazes/beginner/0" rel="current"
        title="Entrance Hallway">
    <link href="http://example.org/mazes/beginner/5" rel="east"/>
    <link href="http://example.org/mazes/beginner/1" rel="south"/>
  </cell>
</maze>
```

这些信息在被翻译成 HTML 以后就会被展示给人类用户。这也就是在我最终看到图 5-6 的界面之前所发生的一切。

现在，我就已经处在“初学者迷宫”的内部了。从此以后，我就可以参观这个迷宫了，我会从所提供的列表中选择一个方向（east、north 等）将其输入文本框。每次我单击一下 Go 按钮，我都是在告诉客户端去发起一次 HTTP GET 请求来访问所对应的链接。图 5-7 展示了我在穿过初学者迷宫的过程中位于单元格 G(“The Tool Room”)时的情况。

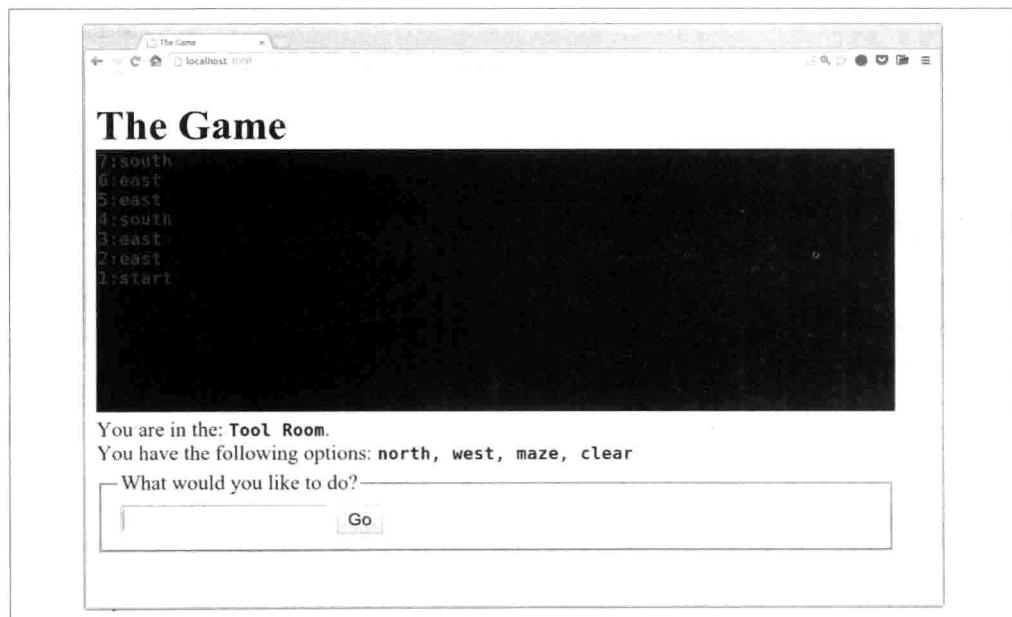


图5-7 初学者迷宫的中间

图 5-8 展示了我从迷宫走出来到达迷宫外部后的情况。

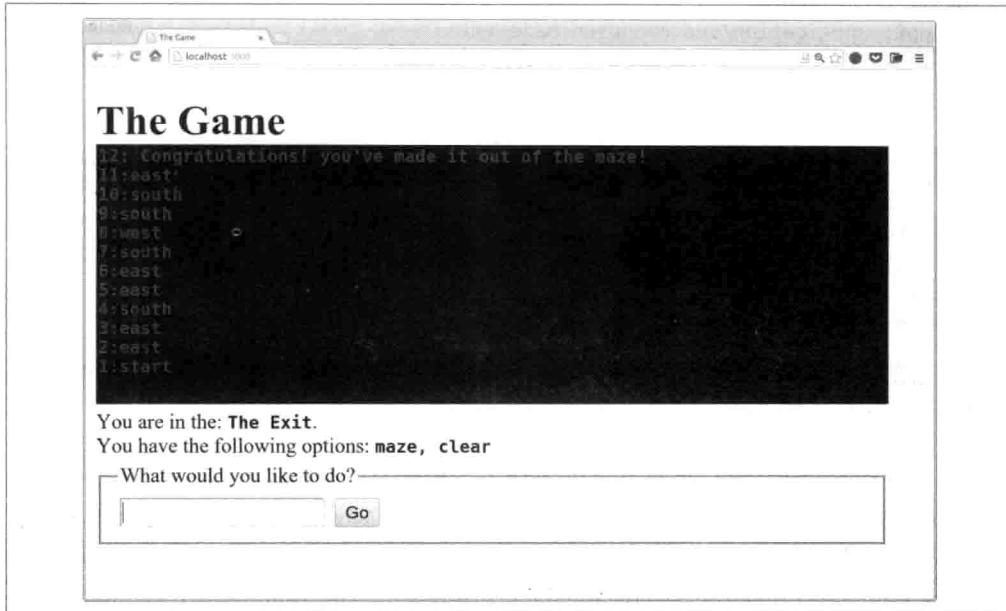


图5-8 初学者迷宫的外部

当我离开迷宫时，客户端展示了一条位于 *exit* 链接另一端的祝贺消息（资源状态的一部分）。

Maze+XML服务器

我将还会编写另外两个 Maze+XML 客户端，但是在此之前，我要先介绍一下服务器端的实现。本章的所有客户端都是运行在一个非常简单的服务器上的，这个服务器是我按照 Maze+XML 标准专门为本书开发实现的^{注3}。

当今大部分的 API 都是 fiat 标准，这些 API 以某家特定的公司为后盾，并且只存在于一台主机上，但是 Maze+XML 是一个个人标准，任何人都可以对其进行开发实现。这就意味着可以有无数的 Maze+XML 服务器，无数的服务器实现。我的服务器实现也并没有什么特别之处。事实上，它的能力相当有限。它只能提供 Maze+XML 迷宫的一个很小的子集：整洁的、功能良好的、能够适应基于 JSON 的文件格式的迷宫。

^{注3} 服务器端代码在 *RESTful Web APIs* 的 Github 仓库（查看 *Maze/server*）目录中。

我的服务器不意味着就是最好的 Maze+XML 实现，但是易于向它添加新的迷宫进去。我的服务器是用简单的 JSON 文档来保存迷宫数据的。下面的 JSON 文档所代表的迷宫就是我之前用来作为例子的那个“初学者”迷宫。这并不是 REST 意义上的迷宫的表述，因为它永远不会通过 HTTP 被发送出去。它只是原始数据，这些原始数据被用于生成能通过 HTTP 发送出去的 Maze+XML 文档：

```
{
  "_id" : "five-by-five",
  "title" : "A Beginner's Maze",
  "cells" : {
    "cell0": {"title": "Entrance Hallway", "doors": [1,1,1,0]},
    "cell1": {"title": "Hall of Knives", "doors": [1,1,1,0]},
    "cell2": {"title": "Library", "doors": [1,1,0,0]},
    "cell3": {"title": "Trophy Room", "doors": [0,1,0,1]},
    "cell4": {"title": "Pantry", "doors": [0,1,1,0]},
    "cell5": {"title": "Kitchen", "doors": [1,0,1,0]},
    "cell6": {"title": "Cloak Room", "doors": [1,0,0,1]},
    "cell7": {"title": "Master Bedroom", "doors": [0,0,1,0]},
    "cell8": {"title": "Fruit Closet", "doors": [1,1,0,0]},
    "cell9": {"title": "Den of Forks", "doors": [0,0,1,1]},
    "cell10": {"title": "Nursery", "doors": [1,0,0,1]},
    "cell11": {"title": "Laundry Room", "doors": [0,1,1,0]},
    "cell12": {"title": "Smoking Room", "doors": [1,0,1,1]},
    "cell13": {"title": "Dining Room", "doors": [1,0,0,1]},
    "cell14": {"title": "Sitting Room", "doors": [0,1,1,0]},
    "cell15": {"title": "Standing Room", "doors": [1,1,1,0]},
    "cell16": {"title": "Hobby Room", "doors": [1,0,1,0]},
    "cell17": {"title": "Observatory", "doors": [1,1,0,0]},
    "cell18": {"title": "Hot House", "doors": [0,1,0,1]},
    "cell19": {"title": "Guest Room", "doors": [0,0,1,0]},
    "cell20": {"title": "Servant's Quarters", "doors": [1,0,0,1]},
    "cell21": {"title": "Garage", "doors": [0,0,0,1]},
    "cell22": {"title": "Tool Room", "doors": [0,0,1,1]},
    "cell23": {"title": "Banquet Hall", "doors": [1,1,0,1]},
    "cell24": {"title": "Spoon Storage", "doors": [0,0,1,1]}
  }
}
```

每个单元格都是由一个名称 ("title") 以及一个由二进制数字组成的列表 ("doors") 来表示的，其中这个数字列表表示的是在北、西、南、东四个方向上是否有门。这些罗列出来的大小相同的单元格组成了一个 5×5 的二维网格。像这样的迷宫被称为完美的迷宫 (perfect mazes)^{注4}，人们很容易找到它们的出口从而走出迷宫。我的服务器所能理解的迷宫也就如此而已。但是 Maze+XML 媒体类型可以展示各种不同大小的二维迷宫拓扑结构——想象一下单行道的迷宫或者随机生成的无限大的迷宫。

^{注4} 你可以在 Astrolog 的迷宫分级页面 (<http://www.astrolog.org/labyrnth/algrithm.htm>) 找到更多这一主题的信息。

客户端2：地图生成器

游戏客户端依靠人类来决定往哪里走。但是现实中已经有很多用于自动穿越迷宫的算法，我们没有理由写不出一个能跟这个手动操作的客户端相匹配的自动化的客户端。

鉴于我已经编写过一个目标是穿越迷宫的客户端（客户端1：游戏）了，为了保持内容的趣味性，这个新的客户端会做一些稍微不同的事情。我将它称为地图生成器，它是一个用于绘制迷宫地图的客户端（地图生成器的代码在 *RESTful Web APIs GitHub* 仓库中的 *Maze/the-mapmaker* 目录下）。这个客户端会试图访问迷宫里的每一个单元格并且构造出一张能够直观地展示出来的地图。这个客户端不会试图离开这个迷宫，它希望能看见全部的东西。当它发现出口以后，它会将出口标记在地图的内部表述里，然后继续移动。它不会访问“exit”链接。

游戏客户端是一款用 Node 编写的运行在 web 浏览器中的 web 应用程序。这个地图生成器也是用 Node 编写的，但是它是一款命令行应用程序，它会将输出（output）打印到控制台中。如果你为这个客户端提供整个 Maze+XML 系统的广告牌 URL 的话，这个程序会绘制出网站上所有的迷宫的地图。如果你给客户端提供的是某一个迷宫的 URL 的话，它将只绘制出这个迷宫的地图。当我将地图生成器客户端运行在初学者迷宫上时，地图生成器程序会输出如下的 ASCII 码样式的结果。

```
$ node the-mapmaker http://localhost:1337/mazes/tiny
Exploring A Beginner's Maze...
```

```
+-----+
|       |       |       |       |
| S 00   05   10 | 15   20 |
|       |       |       |       |
+-----+-----+   +-----+--+
|       |       |       |       |
| 01   06 | 11   16   21 |
|       |       |       |       |
+-----+   +-----+-----+--+
|       |       |       |       |
| 02   07   12 | 17   22 |
|       |       |       |       |
+-   -+-----+-----+   -+-----+
|       |       |       |       |
| 03 | 08   13 | 18   23 |
|       |       |       |       |
+-   -+-- -+-- -+-- -+-- -
|       |       |       |       |
| 04   09 | 14   19   24 E
|       |       |       |       |
+-----+-----+-----+-----+
```

Map Key:

S = Start
E = Exit
0:Entrance Hallway
1:Hall of Knives
2:Library
3:Trophy Room
4:Pantry
5:Kitchen
6:Cloak Room
7:Master Bedroom
8:Fruit Closet
9:Den of Forks
10:Nursery
11:Laundry Room
12:Smoking Room
13:Dining Room
14:Sitting Room
15:Standing Room
16:Hobby Room
17:Observatory
18:Hot House
19:Guest Room
20:Servant's Quarters
21:Garage
22:Tool Room
23:Banquet Hall
24:Spoon Storage

服务器并没有用这种图形化的格式定义迷宫；这些迷宫都是采用 JSON 文档保存的并通过 XML 文档对外提供服务。地图生成器程序通过自动探测来构造出了迷宫的这幅图形化视图。

当地图生成器进入一个迷宫时，它会识别出第一个宫格中的所有门口（链接），然后每次访问其中的一个链接，它会非常高效地从一个宫格瞬间移动到另一个宫格而不用麻烦地原路返回。在每个宫格里面，地图生成器会查看所有的出口并构造出一个它还需要继续访问的宫格的列表。地图生成器会很有效率地对迷宫执行一次广度优先搜索。

一旦所有的单元格（以及单元格之间的所有链接）都被解析过，地图生成器就会利用它所收集来的数据来生成一个 ASCII 码地图，这个地图可以展示出这些单元格以及它们之间的关系。

和大多数的 API 客户端相比，地图生成器客户端拥有更广的应用状态视图。游戏客户端会像人类穿过迷宫那样来行动。你总是正在“访问”某一个特定的单元格，并且你只能移动到和你当前所访问单元格直接相邻的单元格。这些相邻的单元格就是你可能的下一

个状态。当你输入一个方向时，你就从这些可能的状态中选择了一个，而放弃了其他的状态。你在不断地前进。Web 浏览器就是这样工作的。

76 地图生成器不会一直前进。就它而言，它所看到的每个链接都是一个可能的下一个状态。它不会像人一样穿过迷宫，它会像菌类一样扩散，直到它占领了迷宫的所有单元格。

从服务器的角度看，地图生成器好像是发疯似地在迷宫里面来回移动。这很少见，但是就 Maze+XML 规范而言，这是完全合法的。地图生成器仅仅是保持的应用状态比游戏客户端要多一些而已。

客户端3：吹牛者

Maze+XML 标准定义了用 XML 文档展示迷宫以及迷宫集合的方式。它并没有说明这些迷宫是做什么用的。面对一个迷宫，人类的本能倾向是寻找出口走出迷宫。游戏客户端就重现了这样的经验。但是 Maze+XML 并没有要求客户端用人类的方式来穿过迷宫。

我们已经看到了：地图生成器客户端一直在迷宫里来回地瞬间移动，它从来不会访问“exit”链接。它会在里面四处跳跃直到整个迷宫都被绘制出来，然后它就会真正停止发起 HTTP 请求。这看起来和迷宫的用途相反，但是谁会这么说呢？

我的第三个 Maze+XML 客户端，吹牛者，将这个逻辑发挥到了极致。这个客户端甚至从没进入过迷宫。它读取迷宫集合，随机选取一个，然后很绝对地声称已经走出了迷宫^{注5}。过程如下：

```
$ node the-boaster http://example.org/mazes
Starting the maze called: For Experts Only...
*** DONE and it only took 2 moves! ***
```

很明显，你不可能在两步之内从这个专家（Experts）迷宫中走出来。这个吹牛者客户端甚至都没有试过。它首先向 `http://example.org/mazes` 发起一次 HTTP 请求，接着读取迷宫集合，然后选择“`For Experts Only`”，最后声称已经在不切实际的步数内走出了迷宫。

这是欺骗吗？在穿越迷宫问题的范围内，这当然属于欺骗。但是在 REST 或者与 Maze+XML 标准兼容的世界里，这完全是正当的。这个吹牛者客户端的确知道 `vnd.amundsen.application/maze+xml` 文档的含义，它也知道那些带有 `rel="maze"` 的链接指向迷宫，它只是懒得去穿过这个迷宫。

注 5 Boaster 的代码在 *RESTful Web APIs* 的 Github 仓库中（查看 *Maze/the-boaster* 目录）。

客户端做自己想要做的事

这里有 3 个客户端：游戏客户端、地图生成器客户端、吹牛者客户端。它们都是首先从理解 Maze+XML 媒体类型来开始工作的，但是它们的目标不同，所以它们用相同的数据做着不同的事情。

这没有问题。服务器的职责是以一种客户端可以参与其中的方式描述迷宫，而不是给客户端下达目标命令。Maze+XML 规范描述的是一个问题空间，而不是客户端和服务器之间指定的关系。客户端和服务器必须对传递于它们之间的表述达成共识，但是它们不需要在要解决什么问题上有相同的看法。

对标准进行扩展

Maze+XML 是一个无聊的问题领域中的一个特别设计的例子。但是让我们想象一下这样的场景：某个人确实想要提供超媒体迷宫服务，或者是作为某项业务的一部分，抑或是仅仅为了娱乐。这些需求都不会自动使得 Maze+XML 成为正确的选择。甚至于当已经存在有某个面向你的问题领域的标准的时候，它还是有可能无法完全满足你的需求。

任何真正想要采用 Maze+XML 的人都将不会满足于标准中提到的内容。这个标准将你限定在使用东南西北 4 个基本方向的二维迷宫中。这不是很好玩。如果我想要提供三维迷宫的服务，我该怎么办呢？

仅仅为了支持三维迷宫，而从零开始创造一个崭新的标准是很愚蠢的行为。Maze+XML 标准已经几近足够好了。我只需要对其进行稍微的扩展使得它可以支持两个新的方向：up 和 down 就足够了。

幸运的是，Maze+XML 明确地允许这种类型的扩展（见 Maze+XML 规范第 5 节）。只要我不重新定义一些规范中已有的内容，我可以向 Maze+XML 文档中添加任何我想要的内容。为了得到我所想要的三维迷宫，我将仅仅在这里定义两个新的链接关系就足够了：

up

指向位于当前资源的上方的资源。

down

指向位于当前资源的下方的资源。

这是一个简单的扩展，但是它完全改变了迷宫的样式以及迷宫在服务器中的保存方式。我的服务器实现是用一个以单元格为元素填充的二维数组来保存迷宫的，每个单元格周围可能有 4 个单元格。为了支持这两个新的关系，我需要修改服务器代码来反映如下事实：一个迷宫是一个三维数组，每个单元格周围可能有 6 个单元格。

但是客户端不会看到很大的变化。客户端看到的仅仅是在表述中增加了两个新的链接关系：

```
<maze version="1.0">
  <cell href="/cells/middle-of-ladder">
    <title>The Middle of the Ladder</title>
    <link rel="up" href="/cells/top-of-ladder"/>
    <link rel="down" href="/cells/bottom-of-ladder"/>
  </cell>
</maze>
```

这些使得 Maze+XML 标准看起来过于简单的事物成功地将所有新增加的服务器端复杂度都对客户端隐藏起来。Maze+XML 标准并没有在迷宫“应该”是什么样子的问题上说明太多。为相互连接的单元格定义两个新的连接方式需要对服务器端实现进行完整的重新设计，但是修改之后的表述仍然与 Maze+XML 标准兼容，客户端还是可以解析它们的。

但是这并不表示客户端能够自动理解这些新的应用语义。考虑一下当我们分别为游戏客户端、地图生成器、吹牛者客户端提供一个三维迷宫的时候，将会发生什么事情呢？

令人惊讶的是，游戏客户端运行正常！这个客户端的代码并没有写死只有 4 个基本方向。它被编程设计为将发现的每个链接都展示给用户，让用户从中选择一个链接。因为我将新的链接关系命名为“up”和“down”，所以当人类穿过三维迷宫的时候，他会看到如图 5-9 所示的界面。

这些选项对人类用户来说是有意义的，如果用户输入“up”，客户端就会访问 `rel="up"` 的链接。向 Maze+XML 增加新的应用语义并不需要对游戏客户端进行任何的改动，因为这个环节中有人类的参与。

吹牛者客户端在三维的迷宫中运行得也很顺利，因为它甚至都不会进入迷宫之中。事实上，不论对迷宫进行怎么样的扩展，吹牛者客户端都应该能工作于任何与 Maze+XML 相兼容的服务器上。

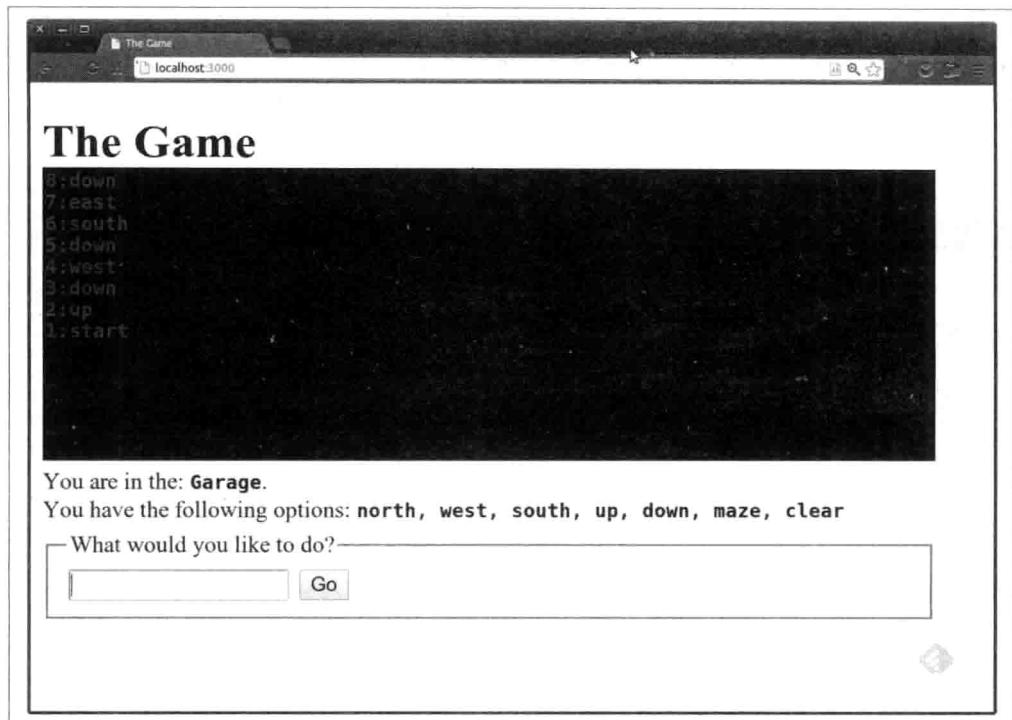


图5-9 游戏客户端自动支持“up”和“down”

但是地图生成器客户端在遇到三维迷宫的时候，就全然不得要领了。考虑如下的表述：

```
<maze version="1.0">
<cell href="/cells/bottom-of-ladder">
  <title>The Bottom of theLadder</title>
  <link rel="up" href="/cells/middle-of-ladder"/>
  <link rel="east" href="/cells/tunnel"/>
  <link rel="north" href="/cells/underwater-garden"/>
</cell>
</maze>
```

如果地图生成器得到如上所述的表述，它会访问“east”和“north”链接，但是绝对不会访问“up”链接。让客户端在这个三维迷宫中自由移动，它将会绘制出迷宫的某一层的地图。它所看到的只是迷宫的一个二维切片。

这是可以理解的。我们不能期望客户端能够理解那些并没有编程实现的链接关系。但是你可能也不会预料到，即便将这些新的连接关系简单地告诉给地图生成器，这也并不会有任何帮助！

即使地图生成器知道如何访问一个“up”连接，但是它依旧不知道如何展示它在该链接的另一端发现的资源。和我们的服务器实现的例子一样，地图生成器客户端也只有一个二维的思维。它生成的是一个二维的 ASCII 码地图。一个三维的迷宫完全不能和地图生成器客户端相兼容。

地图生成器的缺陷

事实上，即便是运行在一个没有使用任何 Maze+XML 扩展的迷宫，地图生成器也同样可能会失败。如果服务器端发送如下表述，地图生成器会如何处理呢？

```
<maze version="1.0">
<cell href="/cells/44">
  <title>Hall of Mirrors</title>
  <link rel="east" href="/cells/45"/>
</cell>
</maze>
```

如果服务器接着发送 east 链接所对应的另一端的表述呢？

```
<maze version="1.0">
<cell href="/cells/45">
  <title>Mirrored Hall</title>
  <link rel="west" href="/cells/129"/>
  <link rel="east" href="/cells/44"/>
</cell>
</maze>
```

它们两个都是合法的 Maze+XML 文档。但是它们所描述的迷宫确是一个非欧几里得地图。从“Hall of Mirrors”的单元格向东走进入“Mirrored Hall”单元格，然后继续向东走，你却回到了“Hall of Mirrors”单元格。你可能在很多电子游戏中见到过这种恶作剧的把戏。这是一个完全合法的没有使用任何 Maze+XML 扩展的迷宫，但是地图生成器会在试图绘制这个迷宫的地图的时候崩溃掉。

看起来好像是地图生成器在设计之初就有一个隐藏的前提假设！那就是：服务器端只提供整齐的、能够用网格表示的迷宫。而我们的服务器实例也只提供这种类型的迷宫，这并非巧合。我在设计地图生成器客户端的时候就考虑了特定的服务器。这也就证明我们的这个客户端并不能工作于符合 Maze+XML 规范的全部迷宫。它只能工作在那些你可以在我们的服务器上能找到的迷宫。

我发现这个规律具有普遍适用性。某个为特定的服务器实现编写的客户端可以针对该服务器上的某些古怪的问题进行特别优化，但是如果你试图让它运行在相同标准的另一种服务器实现上，它很快就会失败。这并不意味着地图生成器是一个完全没用的客户端，

它只不过是只能给某些迷宫绘制地图而已。

想象一下这样的场景，你要启动一个只在某个特定的网站上测试过的 web 浏览器。一旦你将该浏览器访问一个没有被测试过的网站，浏览器很可能就会崩溃。情况就是这样，类似于 Maze+XML 的标准可能有很多服务器端实现，客户端实现需要被设计成能工作于所有的服务器实现，而不是仅仅某个服务器实现。

修复（以及修复后的瑕疵）

我们可以修复这个地图生成器吗？一个“修复方案”就是让客户端去检测每个新发现的单元格是否符合它所要构造的网格结构。和程序崩溃不同的是，当它检测出在同一个网格区域中有两个不同的单元格时，它就会打印一条错误消息然后优雅地退出。

但是那样做仅仅是避免了崩溃而已。我们已经给客户端提供了正好足够的情报用来识别出它不能理解的迷宫。如果我们希望客户端能够真正理解那些并不完美的迷宫，那么这个“网格”的数据结构就不能再用了。我们应该使用的正确的数据结构是有向图（directed graph）。

我们可以编写一个更好的地图生成器，这个地图生成器在迷宫中穿梭的时候会构造出一张有向图，然后使用像 force-directed graph drawing 这样的算法来将有向图呈现出来。对于一个或多或少符合网格结构的迷宫，改进后的地图生成器会将有向图渲染为和之前的老式地图生成器的 ASCII 码图看起来很相似的图形（见图 5-10）。

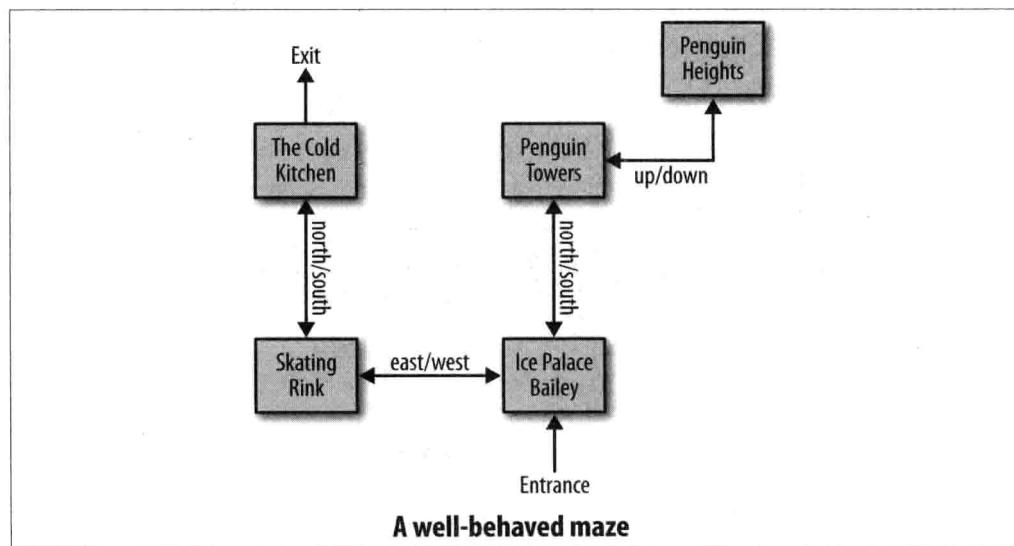


图5-10 被看作有向图的功能良好的迷宫

图 5-10 就是地图生成器所生成的地图，它将书页的上部定义为“north”（北）。对于如何处理“up”和“down”，还不确定，但是可以断定的是，它们很可能是相反的方向，并且将它们以直观的方式展示出来是可行的。

现在想象一下一个拥有无数环路和单行道的恶作剧的迷宫。这样的迷宫很可能会使得那个老式的、改进前的地图生成器崩溃，但是经改进后的地图生成器可以为该迷宫呈现出一张图，如图 5-11 所示。

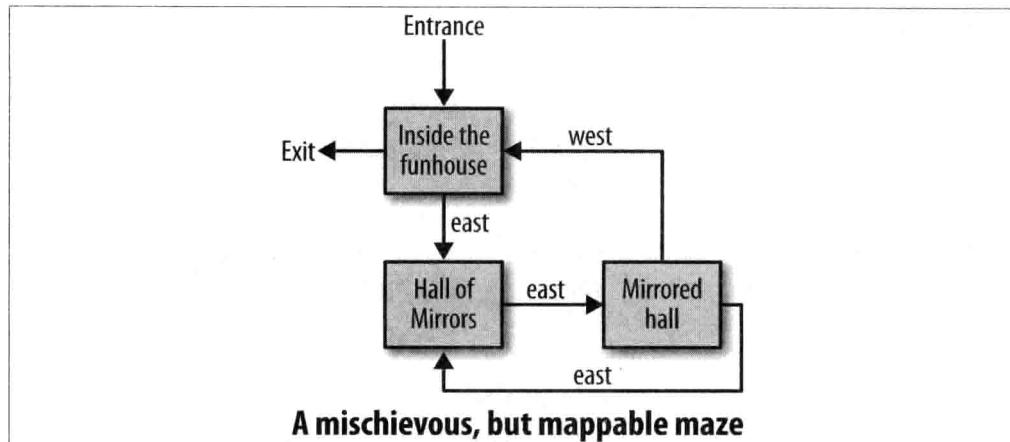


图5-11 恶作剧迷宫

现在的地图生成器已经完美了吗？不幸的是，还没有。这个改进后的地图生成器依旧包含一些隐藏的前提假设。图 5-12 展示了一个无限大的迷宫。人们很容易找到这个迷宫的出口从中出去，但是将这个迷宫的地图绘制出来却是不可能的。

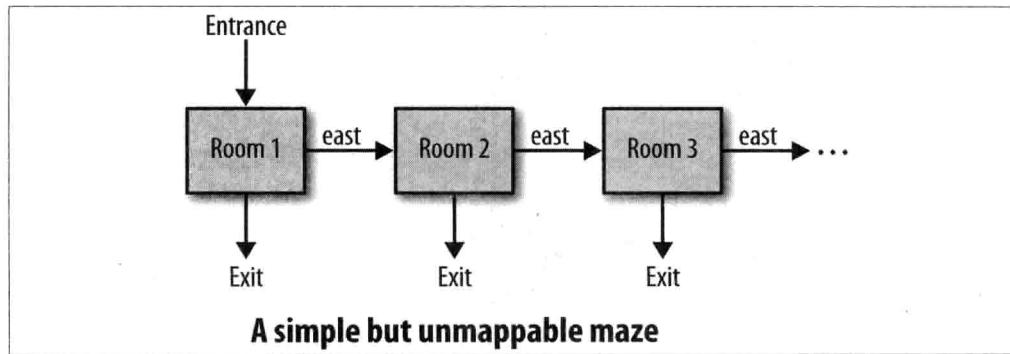


图5-12 一个简单但是无法绘制地图的迷宫

任何地图生成器客户端都不可能为一个无限大的迷宫绘制出地图来：它永远不会有机会

将地图绘制出来。这并不意味着地图生成器是没有用的。这只是说它并不能处理所有符合 Maze+XML 标准的独立迷宫。

迷宫的暗喻

再回过头来看看图 5-10 至图 5-12。将它们与图 1-9 做一下比较，图 1-9 是我之前用来展现网站结构的有向图。

它们的相似性并不是偶然的。正如我在本章开始所讲的那样，对于超媒体应用而言，迷宫通常是一个很好的暗喻。有些“迷宫”很整齐并且功能良好，而有一些迷宫就非常混乱并且无限大。将状态图看作一个需要游览的迷宫，这样你的思路就能进入理解超媒体 API 正确的轨道了。

解决语义鸿沟

对于一个领域特定 API 的设计者而言，消除语义鸿沟需要两个步骤：

1. 将你的应用语义写到一个人类可读的规范（比如 Maze+XML 标准）中。
2. 为你的设计注册一种或者多种 IANA 媒体类型（就像 `vnd.amundsen/application/maze+xml`）。在注册处，将这些媒体类型与你编写的人类可读的文档关联起来。在第 9 章，我将讨论媒体类型的命名和注册流程。

你的客户端开发人员可以颠倒这个过程从反方向来消除语义鸿沟：

1. 在 IANA 注册表中查找陌生的媒体类型。
2. 通过阅读规范来了解如何处理这些陌生的媒体类型的文档。

这里并没有捷径可走。为了使客户端代码能正常工作，你的用户将必须阅读你编写的文档进而做一些其他的工作。我们不可能完全摆脱语义鸿沟，因为计算机还不能像人类一样聪明。

领域特定设计在哪里？

当你想要发布一个 API 的时候，首先要做的是找到一个已有的领域特定设计。重复造轮子是没有意义的。

即便如此，你也不大可能找到一套完整的解决方案。领域特定的数据格式数以百计，但是它们中的很多格式并不包含超媒体控件。在第 10 章中，我会介绍一些例外的格式，比如 VoiceXML 和 SVG。你可能会采纳的领域特定设计是问题细节的文档，一种简单的基

于 JSON 的用于描述错误条件的格式（同样会在第 10 章中讨论到）。

84

但是一个数据格式不包含超媒体控件，并不意味着这个数据格式就是没有用处的。在第 8 章，我将为你展示 JSON-LD 是如何来为任意的 JSON 格式添加基本的超媒体能力的。在第 10 章，我将同样展示 XForms 和 XLink 如何来为 XML 格式完成同样的功能。这些技术可以让你将超媒体控件移植到现有的并不包含超媒体控件的 API 上。

最终的奖赏

我们在超媒体 API 中依旧可以找到那些不支持超媒体的格式的用武之地。考虑一下 JPEG 图片格式。关于该格式的文档记录很完善，并且它也拥有一个注册的媒体类型 (`image/jpeg`)，没有什么能比二进制的图片文件更好地用于展示相片的了。但是你不能使用 JPEG 来作为 Web API 的主要内容，除非你想要设计一个只提供 JPEG 图片的网站。我们也没有办法将一张 JPEG 图片链接到另一张图片。

用于管理相片的 web API 很可能会用 JPEG 格式来发送和接收表述。仅仅因为 JPEG 格式没有超媒体控件就构造你自己的二进制图片格式是很愚蠢的行为。但是 JPEG 格式并不是基于超媒体的相册 API 的核心内容，这项荣誉应该属于像 HTML 这样的格式。虽然 HTML 不能直接展示一张相片，但是它可以将照片嵌入一个文本文档中、将相片及它的标题配对、展示一个相片列表以及展示用于搜索和标记相片的表单。

一个 `image/jpeg` 表述将是客户端浏览相片 API 的超媒体“迷宫”并定位到一个特定的相片后得到的奖品。这个“迷宫”自己是使用支持超媒体控件的文档格式描述的。这两种格式共同工作组成了一个完整的 API。

报头中的超媒体

我在第 4 章中已经展示过如何使用 HTTP 报头 `Link` 来为那些没有超媒体控件的文档添加简单的超媒体链接和表单。通过使用这些报头，你可以令人信服地设计出一个只提供 JPEG 图片的 API，但是我不建议你这么做。

抄袭应用语义

这是一个完全不同的技术。我想要首先介绍一下 vCard 格式，它是由 RFC6350 定义的，并且分配有一个媒体类型 `text/vcard`。这是一个设计用来交换业务名片上的各种个人信息的领域特定的纯文本的格式。听起来很有用，是吗？许多 web API 都要处理与人和业务相关的信息。

如下是一个简单的 vCard 表述：

```
BEGIN:VCARD  
VERSION:4.0  
FN:Jennifer Gallegos  
BDAY:19870825  
END:VCARD
```

85

RFC 6350 所确定的规则定义了 `text/vcard` 文档的语义。你可以按照这些规则对这个文档进行解析进而构造出一个人的大致轮廓——他的名字、出生日期等。

你被束缚在这里了。这些应用语义意义明确，但是却没有任何链接。这个文档对于超媒体来说是个死胡同。

当然，HTTP 的协议语义还是适用的。你发送 GET 请求就会收到这个表述。API 可能允许你修改表述然后 PUT 回去。你可能还可以发送 DELETE 请求来删除所对应的资源。但是你不能从这个表述移动到另外一个相关的资源上去，因为 vCard 不是一个超媒体格式。

由于 vCard 是在电话和软件的通讯录中常用的一种格式，所以将 vCard 表述作为在超媒体迷宫尽头的奖品是有意义的。客户端可以通过超媒体定位一个“person”资源，然后就可以访问一个“export to vCard”链接来获取这个“person”资源的 `text/vcard` 表述。

但是这很可能并不是你想要的。你不想将一个人的基本信息作为“奖品”。它很可能是一个 API 的主要部分。你可能想要抄袭 vCard 的应用语义并将它们应用到一个真正的超媒体文档中。

hCard microformat (<http://microformats.org/wiki/hcard/>) 的设计者正是这么做的。他们并没有重复 vCard 标准的制定者们已经完成的工作，而是让 HTML 这一超媒体文档格式来展示同样的信息成为可能。

我在第 7 章中将对 hCard 进行更多介绍，现在只是一个预览——这是我前面展示过的 vCard 文档的 hCard 版本：

```
<div class="vcard">  
  <span class="fn">Jennifer Gallegos</span>  
  <span class="bdy">1987-08-25</span>  
</div>
```

hCard 微格式使得你可以将一个人的 vCard 样式的表述与超媒体链接以及表单组合起来，从而实现一个完整的 web API。

这也就是在开始设计 API 之前查阅寻找相关领域特定的数据格式非常重要的另一个原因。类似于 vCard 这样的标准都是针对某个问题领域花费了大量的时间和金钱，从而确认了其应用层语义后的成果。你并不需要仅仅因为 vCard 没有超媒体控件就一切都重新来过。

即便你不能直接复用某个领域特定设计，你也可能能够通过将它的应用层语义整合到一个 profile 中来为你节省一些时间。但这将是第 8 章的一个主题。

86 > 如果找不到相关的领域特定设计，不要自己制造

如果你不能为你的问题领域找到一个合适的领域特定 API 的话，不要慌张。人们并不是经常定义可复用的、领域特定的、支持超媒体的格式。这并不意味着你不得不从零开始。你应该能够以某种标准化的格式为基础，对其进行扩展并复用其他人可能已经完成的工作。你需要做的工作并不多，仅仅是将各个部分粘合在一起。

在下面两章中，我将会讨论一些基础内容。需要指出的是，这其中的一些领域特定设计会用来处理一些相当流行和普遍的领域——事物的集合。我也并不是真的认为这是一个领域，它更像是一个设计模式。

API 客户端的种类

除去本章中讲到的 3 个 Maze+XML 客户端，我在本书中也不会再讨论更多关于 API 客户端的内容了。因为现在网上部署的能完全发挥 Fielding 约束的优点的 API 并不多，所以我也无法给予更多的指导。

即便你对于超媒体的知识理解非常深刻，可是在你遇到一个不支持超媒体文档的 API 时，那些知识对于你为这个 API 编写客户端也是毫无帮助的。当你编写客户端时，你是任由服务器设计摆布的，并且实用主义总是打败理想主义。现在，实用主义意味着为每个 API 采用不同的方法。

但是我已经看到过非常多的已经部署的超媒体 API（包括万维网自己），所以我可以讲一讲人们通常编写的客户端的种类。我已经理解：客户端和服务器必须对问题领域有一致的理解，但是它们却并不需要共享同一个目标。老实说，这是我第一次尝试对这些我们所编写的用来完成我们的目标的客户端进行分类。

人类驱动的客户端

人类驱动的客户端可以拥有相对简单的逻辑，这是因为它们并不需要做出任何决定。它们将表述展示给人类，并且将人类的决定回传给服务器。人类驱动的客户端之间的区别取决于它们在多大程度上忠实地将表述展示给它们的人类用户。

标准的 web 浏览器就是一个忠实的渲染程序，几乎网页中的每一个 HTML 标签都会对屏幕上显示的内容产生一些图形影响。HTML 文档中的每个超媒体链接以及表单都会出

现在屏幕上，除非文档中有一些另外的内容声明它们应该被隐藏起来。

现在考虑一下采用文本 - 语音转换技术来向视力受损的用户展示网页的 web 浏览器。这样的浏览器在忠实于原文方面就稍微差了一些。有一些 HTML 标签可以很好地翻译成语音的表现方式 (`` 就是一个很好的例子)，有的就不行（比如 `<div>`）。但是任何 web 浏览器都必须忠实地渲染超媒体控件。视力正常的用户所能触发的每个链接和表单也应该同样能够被视力受损用户所使用。

游戏客户端就是一个相当忠诚的 Maze+XML 文档渲染器。Maze+XML 文档不像 HTML 文档那样包含布局信息，但是游戏客户端能保证将它所发现的资源状态的所有信息（比如一个单元格的标题）、所有的超链接都展示给它的人类用户。

用只有“左转”、“右转”、“前进”命令的坦克控件来取代方向链接的游戏客户端将是一个不够忠实的渲染器，尽管它向服务器发送的请求和我之前展示给你的游戏客户端发送的请求是一样的。但是一个拒绝将“exit”链接展示给用户并导致用户被永远困在迷宫之中的游戏客户端根本不是一个非常忠实的客户端。

一个不那么忠实的渲染器会在服务器发送的内容和用户体验之间增加一些源于它自身的态度和想法的想象。这听起来很糟糕——没有人会喜欢“不忠实”的事物——但是一件事物忠实与否取决于用户想要做什么。一个商店的网站可能会展示给你很多昂贵的而且你不需要的物品。这时，你可能会更喜欢一个不那么忠实的渲染器：一个根据网店的 API 能自动过滤掉昂贵商品来帮助你找到物美价廉的物品的客户端。

当人类用户选择一个迷宫游玩的时候，这个游戏客户端会获取一个迷宫的表述，但是它并不将这个表述显示出来。它会扫描表述找到 `rel="start"` 的链接，然后自动地访问这个链接。这是“不忠实的”。这个游戏客户端认为在迷宫的表述中没有人类用户感兴趣的内容，并且让人类手动单击“start”链接是浪费时间的行为。这很可能是正确的，但是也正是因此，游戏客户端并不是一个完全忠实的客户端。

客户端为了忠实地渲染收到的表述以及不将自己的判断力牵杂其中而付出的努力越多，它在遇到不期望的表述的时候就越不容易崩溃。

自动化客户端

自动化客户端会收到表述，但是不对它们进行渲染。并没有人类成员来看渲染的结果。客户端通过决定触发哪个超媒体控件来消除它们自己所遇到的语义鸿沟。当然，大部分客户端并不能“决定”任何事情。它们仅仅是贯彻执行简单的预先编程实现的规则集合，这些规则有希望能帮助它们来实现一些预定义的目标。

没有哪个客户端像人一样智能。但是它们却实实在在地能够将我们从重复的、并不需要我们太多智力的任务中解放出来。我见过并且开发过一些不同类型的自动化客户端。

爬虫

爬虫模拟了一个好奇但是并不挑剔的人。给它提供一个 URL 来启动，它会获取到一个表述。然后，它会访问所有它能发现的链接用来获取更多的表述。它会以递归的方式重复这样的工作，直到它再也获取不到更多的表述为止。

本章前面讲到的地图生成器客户端就是某种 Maze+XML 文档的爬虫。搜索引擎所使用的蜘蛛是 HTML 文档的爬虫。

为一个没有采用超媒体的 API 编写爬虫是很困难的。但是你可以为基于超媒体的 API 编写爬虫，而且你在编写爬虫的时候甚至都不需要理解这个 API 的链接关系。

通常来说，爬虫只会触发那些安全的状态转换。否则，没人会告诉你资源状态会发生什么事情。一个向它所遇到的每个资源都发送 DELETE 请求而仅仅是要看发生了什么事情的爬虫是一个非常可怕的客户端。

监视器

监视器客户端是和爬虫完全相反的一种客户端。它模拟的是一个需要观察某个特定网页的人。给它提供一个 URL 用于启动，监视器会获取这个 URL 的表述，并通过某种方式进行处理。但是它不会访问任何链接。相反的是，它会等待一段时间然后再次抓取同一个资源的新的表述。和触发某个超媒体控件来改变资源状态不同，监视器客户端是等待另外一些事物来更改资源状态，然后检查并查看发生了什么事情。

RSS 聚合器就是一种监视器。人类用户将聚合器指向一系列他们感兴趣的 RSS 和 Atom 订阅源。监视器客户端就周期性地获取这些订阅源，并通过某种方式通知用户有新的内容发布。

假设其中有一个 Atom 订阅源连接到一个采用 Atom 发布协议的功能全面的 API。聚合客户端并不会注意到这些。它只是想要查看订阅源。用户对 RSS 聚合器所做的任何操作都不会改变发布这些订阅源的网站的资源状态。

脚本

当今大部分的自动化 API 客户端都是脚本。脚本模拟一个有固定的日常工作并从来不会改变的人。当这个人厌烦了这样的日常工作并想要将这些工作自动化的时候，脚本就出现了。

由人类来选择一套 API，并且指定那些对完成这项日常工作所必须的状态转换（对于支持超媒体文档的 API）或者 API 调用（对于不支持超媒体的 API）是必要的。然后，人类就编写一个算法，这个算法会使触发状态转换或者发起 API 调用的过程自动执行。

本章前面讲到的那个吹牛者客户端就是一个非常简单的脚本。它知道一些信息（迷宫的标题），它还知道哪里可以找到这个迷宫。它会在你每次运行它的时候给你提供不同的结果（因为它每次会选择一个不同的迷宫），但是它总是完成同样的任务：假装走出了迷宫。

一个真正进入迷宫并且向东移动三次的客户端会是一个更让人印象深刻的脚本。如果你给它提供一个合适的迷宫，它甚至具有找到出口的能力！但是很明显，这个算法中并没有智能的内容。它只是一个脚本，一个回放一系列预定义的状态转换的脚本。

当脚本背后的假设不再有效的时候，脚本倾向于终止工作。一个只是向东走三步的“迷宫穿越者”的客户端只能穿过极少部分的迷宫。一个从某个网站提取数据的屏幕抓取脚本也会在 HTML 表述被重新设计以后停止工作。

在一些不重要的事情（比如某个资源的 URL 发生变化或者有新的数据要添加到某个表述中）等发生的时候，一个能理解超媒体的脚本不会那么轻易停止工作。这就是说一个超媒体 API 在不阻碍依赖于它的脚本正常运行的情况下有一定的变更空间。但是脚本只是人类思维过程的回放而已。如果它遇到一个人类在起初没有考虑到的情况，脚本将还是不能填补这块空白。

代理机器人

忘记那个吹牛者客户端。忘记向东移动三次然后停止的脚本。想象一个采用像 wall-following 这样的算法能真正依靠自己能力走出迷宫的客户端。这样的客户端将能够走出它以前从来没有见过的迷宫。它会在探路的过程中根据从服务器端收到的表述来改变自己的行为。它会做出决策。^{注6}

一个软件代理机器人会模拟积极地处理某个问题的人类。它不像人类那样智能，它也没有能力来做出主观的判断，但是它会去做人类在相同情况下会做的事情。它会查看某个表述，分析当前形势，决定激活哪个超媒体控件来帮助自己更加接近自己的终极目标。

监视器客户端不会这样做；它从不激活超媒体控件。爬虫也不会这么做；它会激活每个它所发现的安全的超媒体控件。脚本也不会这么做；它总是激活那个程序中写好的那下一个超媒体控件。人类驱动的客户端也不会这么做；它们将任务委派给人类。软件代理机器人是唯一能被称为自主决策的客户端。

^{注6} 你可以在本页 (<http://amundsen.com/examples/misc/maze-client.html>) 中看到一些类似的 Maze+XML 客户端。

软件代理机器人可以像穿越迷宫机器人那样简单，也可以由复杂的推理引擎通过将许多不同来源的信息进行综合来驱动。现在，软件代理机器人客户端倾向于简单的一侧。但是我们也想象的到一些更加复杂的代理机器人，比如：私人导购、科幻小说的自动化新闻采集器以及现实生活中金融软件所采用的高频率交易算法程序。

软件代理机器人是最能发挥超媒体 API 的灵活性的自动化客户端。但是它基于两个很大的潜在的前提假定：它的目标是合理的；编程实现的推理过程最终会通往这个目标。如果违反了这些假设，代理机器人将会停止工作。如果穿越迷宫的客户端遇到了一个使得它的算法不能工作的迷宫，比如，一个只有单行道的迷宫——客户端就会像那个总是向东走三步的脚本一样停止工作。

软件代理机器人是由计算机程序员编程实现的，我们对计算机将重要的决定留给软件处理的过程了解得一清二楚。在关键时刻，API 客户端可以选择让人类来确认一个不安全的状态转换（“你想要我来给你买下这件衬衫吗？”）或者做出主观的判断（“这些景点中，哪一个更漂亮？”）。在这个时候，代理机器人就变成了一个人类驱动的客户端，这很可能减少错误和降低错误发生时所付出的代价。

集合模式 (Collection Pattern)

回顾第2章，我曾向大家展示过一个简单的微博API，它可以提供媒体类型为application/vnd.collection+json的表述。这些表述看上去是这样的：

```
{ "collection":  
  {  
    "version" : "1.0",  
    "href" : "http://www.youtypeitwepostit.com/api/",  
  
    "items" : [  
      { "href" :  
        "http://www.youtypeitwepostit.com/api/messages/21818525390699506",  
        "data" : [  
          { "name" : "text", "value" : "Test." },  
          { "name" : "date_posted", "value" : "2013-04-22T05:33:58.930Z" }  
        ],  
        "links" : []  
      },  
  
      { "href" :  
        "http://www.youtypeitwepostit.com/api/messages/3689331521745771",  
        "data" : [  
          { "name" : "text", "value" : "Hello." },  
          { "name" : "date_posted", "value" : "2013-04-20T12:55:59.685Z" }  
        ],  
        "links" : []  
      },  
  
      { "href" :  
        "http://www.youtypeitwepostit.com/api/messages/7534227794967592",  
        "data" : [
```

```
92 ➤
    {
        "name" : "text", "value" : "Pizza?" },
        { "name" : "date_posted", "value" : "2013-04-18T03:22:27.485Z" }
    ],
    "links" : []
}
],
{
    "template" : {
        "data" : [
            {"prompt" : "Text of message", "name" : "text", "value" : ""}
        ]
    }
}
}
```

在本章中，我将会讨论更多关于 Collection+JSON^{注1} 的内容，它是用以定义上述文档结构的标准。

Collection+JSON 是众多被设计用于表示非特定问题域（相对地，Maze+XML 是针对特定问题域的）的通用标准之中的一员，它符合集合模式，这种模式在各个领域中频繁显现。我们通过该标准构建了一个很好的例子，因为它是那些初出茅庐的 API 设计新手们很容易理解的一个基于 JSON 的 API 的正式版本。Collection+JSON 让你可以遵循内心最自然的设计偏好，而不会与 Fielding 约束发生冲突。

刚才所展示的这个文档代表了一组微博帖子。而购物车中的一组商品或者是来自气象传感器的一组读数看上去是非常相似的，它们都具有非常相近的协议语义。我对 Collection+JSON 标准所添加的唯一元素就是那一点点应用语义。我确定一条微博应该具有一个 `date_posted` 字段和一个 `text` 字段。而购物车中的一件商品和来自气象传感器的一条读数都将会具有不同的字段，这都反映出了它们拥有不同的应用语义。

如果恰巧在你的问题域中还没有领域特定标准（这或许不太可能），你或许可以采用一个基于集合的标准来代替。相比于你从零开始，你可以将你的精力专注于将你的应用语义适应集合模式。这不仅仅节省了你的时间，你同时将会得到一些既有的客户端程序和服务端工具。

虽然本章专注于讨论 Collection+JSON，我同样也会谈及 Atom 发布协议（Atom Publishing Protocol），或者叫作 AtomPub。AtomPub 是基于集合的 API 的原始标准，它的定义见 RFC 5023。它是一种相对比较老的标准，但是除了在谷歌公共 API 中被采用过之外，它并没有流行起来——其中一部分原因是它是基于 XML 格式的，而目前在该领域处于霸主地位的是 JSON 形式的表述。

注1 Collection+JSON 由一份个人标准定义，见 <http://amundsen.com/media-types/collection/>。

在第 10 章中，我将会谈到 OData，它是集合模式中的第三项主要标准。OData 是一项尚在制定中的开放标准，它最初是基于 AtomPub 的。它具备了 JSON 表述的优势，并且由微软提供支持。微软已经在它的 Visual Studio 开发平台中集成了对 OData 的支持。

Hydra 标准（见第 12 章）同样也支持集合模式，虽然这并不是它的主要用途。如果我们在基于集合的 API 方面能拥有一个单一且一致的标准，那将是一件非常不错的事情。但是眼下却有 4 种标准正在互相竞争，即使如此，也总是胜过我们目前所拥有的数千个一次性设计。

什么是集合？

在深入围绕集合模式设计的标准细节之前，让我们先探讨下该模式本身。这其实相当简单，但是我还是想显式地将每件事都罗列出来，所以请不要过于惊讶。

集合是一种特别的资源类型。回顾第 3 章中关于资源的定义，资源是足够重要并提供其自身 URL 的任何事物。一个资源可以是一条数据、一个物理对象或者是一个抽象的概念——甚至是任何东西。所有的重点便是它拥有一个 URL 及其表述——即当客户端向该 URL 发送 GET 请求时所收到的文档。

一个集合资源相比于上述的资源会更加特殊些。它的存在主要是为了将其他资源组合到一起，它的表述主要专注于那些指向其他资源的链接，尽管它也可能包含来自其他资源表述的一些片段（或甚至可能是全部表述！）。



集合是一个以链接方式罗列其他资源的资源。

链向子项的集合

包含在集合中的一个独立的资源通常可以被称为集合的子项 (item)、条目 (entry) 或者是成员 (member)。联想到你朋友手机中的联系人列表，你被展示在这样一个列表里：里面有你的名字和你的手机号码。你是该“联系人列表”集合的一个子项。

但是你又不只是集合中的一个子项：你是一个人类。你朋友手机中保存的并不是你，而只是一个指向你的链接（通过你的手机号码）和一些关于你的信息（你的名字）。你拥有一个独立的存在，你朋友手机里的数据只是你的部分表述。

相似的是，一个为集合所描述的资源并不会在突然间就成为了一个被称为“子项”的特

殊事物。该资源仍然拥有自己的 URL 和一个在集合之外的独立存在。当我们在讨论某个“子项”或者“条目”，甚至是“成员”时，实际上是在讨论的是一个恰好被链接到集合表述中的独立资源。

94 Collection+JSON

现在，让我们瞧瞧 Collection+JSON 是如何实现集合模式的，从而获得一些更具体的感受。Collection+JSON 标准定义了一个基于 JSON 的表述格式，它同样也为 HTTP 资源定义了协议语义，这些资源会以这样的格式来响应 GET 请求。

下面是一个 Collection+JSON 文档：

```
{ "collection":  
  {  
    "version" : "1.0",  
    "href" : "http://www.youtypeitwepostit.com/api/",  
  
    "items" : [  
      { "href" : "/api/messages/21818525390699506",  
        "data" : [  
          { "name" : "text", "value" : "Test." },  
          { "name" : "date_posted", "value" : "2013-04-22T05:33:58.930Z" }  
        ],  
        "links" : []  
      },  
  
      { "href" : "/api/messages/3689331521745771",  
        "data" : [  
          { "name" : "text", "value" : "Hello." },  
          { "name" : "date_posted", "value" : "2013-04-20T12:55:59.685Z" }  
        ],  
        "links" : []  
      }  
    ],  
    "links" : [  
      {"href" : "/logo.png", "rel" : "icon", "render" : "image"}  
    ],  
  
    "queries" : [  
      { "href" : "/api/search",  
        "rel" : "search",  
        "prompt" : "Search the microblog archives",  
        "data" : [ {"name" : "query", "value" : ""} ]  
      }  
    ],
```

```
    "template" : {
        "data" : [
            {"prompt" : "Text of message", "name" : "text", "value" : ""}
        ]
    }
}
```

一个对象基本上都具有 5 个特定属性，这是为应用特定的数据所预定义的数据槽 (predefined slots)：◀95

href

一个指向集合本身的永久链接。

items

包含指向集合成员的链接，以及它们的部分表述。

links

指向其他与集合相关资源的链接。

queries

用于搜索集合的超媒体控件。

template

用于向集合添加子项的超媒体控件。其中还有一个可选的用于错误消息的错误区域，但是我在此处的讨论中将不会涉及到。

子项的表示

让我们聚焦子项，这是 Collection+JSON 表述中最重要的一个字段：

```
  "items" : [
    { "href" : "/api/messages/21818525390699506",
      "data" : [
        { "name" : "text", "value" : "Test." },
        { "name" : "date_posted", "value" : "2013-04-22T05:33:58.930Z" }
      ],
      "links" : []
    },
    { "href" : "/api/messages/3689331521745771",
      "data" : [
```

```
{ "name" : "text", "value" : "Hello." },
{ "name" : "date_posted", "value" : "2013-04-20T12:55:59.685Z" }
],
"links" : []
}
]
```

我之所以说它是最重要的字段，是因为它让我们清晰地了解到集合中有哪些子项。在 Collection+JSON 中，每个成员都被表示成一个 JSON 对象。像集合本身一样，每个成员都具有一定数量的预定义数据槽，它们可以被应用特定的数据填充：

href 属性

一个指向子项的永久链接，这个子项被作为独立的资源看待。

96 links

指向子项相关的其他资源的超媒体链接。

data

任何其他信息，这是子项表述的一个重要部分。

子项的永久链接

成员的 href 属性是一个指向它所在集合环境之外的资源的链接。如果你向 href 属性中的 URL 发起 GET 请求，服务器将会向你发送该单个子项的 Collection+JSON 表述。它看上去将会像下面这样：

```
{ "collection":
{
  "version" : "1.0",
  "href" : "http://www.youtypeitwepostit.com/api/" ,

  "items" : [
    { "href" : "/api/messages/21818525390699506",
      "data": [
        { "name" : "text", "value" : "Test." },
        { "name" : "date_posted", "value" : "2013-04-22T05:33:58.930Z" }
      ],
      "links" : []
    }
  ]
}
```

你或许可以通过向其永久链接发送 HTTP 的 PUT 请求来修改这个子项，或者是通过

HTTP 的 DELETE 请求来删除它，这些都是子项的协议语义。它们作为 Collection+JSON 中关于“item”定义的一部分，讲得非常清楚。

子项的数据

任何 Collection+JSON 应用的核心都是你尝试传达的应用级语义：即那些与每个独立子项关联的数据。大部分的数据都进入了子项的 `data` 数据槽，该数据槽需要包含一个 JSON 对象的列表，每个对象都具有 `name` 和 `value` 属性，并都被描述为单个键值对。下面是一个来自我们的微博 API 的例子：

```
"data" : [
  {
    "name" : "text",
    "value" : "Test.",
    "prompt" : "The text of the microblog post."
  },
  {
    "name" : "date_posted",
    "value" : "2013-04-22T05:33:58.930Z",
    "prompt" : "The date the microblog post was added."
  }
]
```

97

其中的 `name` 属性是该键值对的键，而 `value` 当然就是值了，还有一个（可选的）`prompt` 值是一段人类可读的描述信息。Collection+JSON 标准并没有说明你应该用什么样的键、值以及提示，这取决于你的需求以及你为 API 定义的应用级语义。

子项的链接

Collection+JSON 中最简单的超媒体控件便是 `href` 属性。我在上文中曾有所提及，它是一个特定的链接，客户端可以在想要引用一个特定的子项的时候随时使用链接所提供的 URL。

```
"href" : "/api/messages/21818525390699506"
```

子项的表述可以同样包含一个称为 `links` 的列表。它包含了任意数量的指向相关资源的其他的超媒体链接。下面是一个你可能会在一个“图书”资源的表述中看到的链接：

```
{
  "name" : "author",
  "rel" : "author",
  "prompt" : "Author of this book",
  "href" : "/authors/441",
  "render" : "link"
}
```

它近似等价于下面的 HTML 片段：

```
<a href="/authors/441" id="author" rel="author">Author of this book</a>
```

`rel` 属性是一个为链接关系准备的数据槽，就好比 Maze+XML 中的 `rel` 属性。它是一个可以让你放置某些应用语义的地方。而 `prompt` 属性是一个用于放置人类可读的描述信息的地方，这就好比是 HTML 的 `<a>` 标签中的链接文本。

下面是另一个你可能在一段图书表述中看到的链接：

```
{
  "name" : "cover",
  "rel" : "icon",
  "prompt" : "Book cover",
  "href" : "/covers/1093149.jpg",
  "render" : "image"
}
```

它近似等价于下面的 HTML 片段：

```

```

`author` 链接和 `icon` 链接之间的区别在于 `render` 属性。如果将 `render` 属性设置为 `link`，则等于告诉 Collection+JSON 客户端应该将该链接呈现为一个转出的链接（见第 4 章），就好比 HTML 的 `<a>` 标签。用户可以单击该链接，从而将客户端的视图转向另一个表述。当 `render` 属性设置为 “`image`” 时，等于告诉客户端应该将该链接呈现为一张内嵌的图片，就好比 HTML 的 `` 标签。该链接的表述会被自动抓取，并且该表述将被直接整合到当前表述的视图中。

98

写入模板 (Write Template)

假设你想要向集合添加一个子项，那你应该发起什么 HTTP 请求呢？为了回答这个问题，你需要看看集合的写入模板。

下面是为我们的微博 API 准备的写入模板：

```
"template": {
  "data": [
    {"prompt" : "Text of message", "name" : "text", "value" : ""}
  ]
}
```

根据 Collection+JSON 标准对模板进行解释，解释完没有问题之后你便可以填写该模板的空白处，然后提交一个如下的文档：

```
{ "template" :  
{  
  "data" : [  
    {"prompt" : "Text of the message", "name" : "text", "value" : "Squid!"}  
  ]  
}  
}
```

该请求会去向何方？Collection+JSON 标准中提到你需要通过向某个集合（也就是向它的 href 属性中的 URL）发送一个 POST 请求才能向该集合添加一个子项：

```
"href" : "http://www.youtypeitwepostit.com/api/",
```

因此，该 POST 请求看上去将会是如下的样子：

```
POST /api/ HTTP/1.1  
Host: www.youtypeitwepostit.com  
Content-Type: application/vnd.collection+json  
  
{ "template" :  
{  
  "data" : [  
    {"prompt" : "Text of the message", "name" : "text", "value" : "Squid!"}  
  ]  
}  
}
```

这意味着写入模板概念上等同于如下的 HTML 表单：

```
<form action="http://www.youtypeitwepostit.com/api/" method="post">  
  <label for="text">Text of the message</label>  
  <input id="text"/>  
  <input type="submit"/>  
</form>
```

99

它们并不完全相同，因为填写完的 HTML 表单发送的是一个 application/x-www-form-urlencoded 的表述，而填写完的写入模板发送的是一个 application/vnd.collection+json 表述。但是从概念上来说，这两种超媒体控件是非常相似的。

搜索模板

如果一个集合拥有数百万的子项，那么服务器为每个客户端的 GET 请求都发送所有子项的表述将是一种非常愚蠢的行为。服务器可以通过提供搜索模板来避免这种情况——客户端可以通过填写超媒体表单来对一个 Collection+JSON 集合进行过滤。

一个集合的搜索模板被保存在 `queries` 数据槽中，下面的 `queries` 数据槽包含了一个简单的搜索模板：

```
{  
  "queries" :  
  [  
    {  
      "href" : "http://example.org/search",  
      "rel" : "search",  
      "prompt" : "Search a date range",  
      "data" :  
      [  
        {"name" : "start_date", "prompt": "Start date", "value" : ""},  
        {"name" : "end_date", "prompt": "End date", "value" : ""}  
      ]  
    }  
  ]  
}
```

Collection+JSON 搜索模板等价于下面的 HTML 表单：

```
<form action="http://example.org/search" method="get">  
  <p>Search a date range</p>  
  <label for="start_date">Start date</label>  
  <input label="Start date" id="start_date" name="end_date" value="" />  
  
  <label for="end_date">End date</label>  
  <input label="End date" id="end_date" name="end_date" value="" />  
</form>
```

同样也等价于下面的 URI 模板：

```
http://example.org/search{?start_date,end_date}
```

我之所以说它们都是等价的，是因为这 3 种方式根据相同的输入都会产生相同的 HTTP GET 请求。该请求看上去将会是像下面这样的：

```
100> GET /search?start_date=2010-01-01&end_date=2010-12-31 HTTP/1.1  
Host: example.org
```

一个（通用的）集合是如何工作的

关于 Collection+JSON 的内容我已经展示得够多了。它的设计中并没有掺杂任何实际的应用语义，所以它可以在很多不同的应用中被使用。由于它如此通用，从而很好地勾画出了集合模式的公共特性。

在我们开始转向 AtomPub 的讨论之前，我想先对 HTTP 下的一个普通的“集合”资源的行为进行描述，进而上升一个层次来对我所认知的模式展开介绍。Collection+JSON、AtomPub、OData 以及 Hydra 对集合都采用了不同的方式，但是它们都或多或少具有相似的协议语义。

GET

和大多数资源一样，集合在响应 GET 请求时会提供一个表述。虽然这 3 种主要的集合标准都没有详细指明集合中的子项应该是什么样子的，但是它们都非常具体地说明了集合的表述应该是怎样的。

表述的媒体类型告诉了你可以对资源作何处理。如果你得到的是一个 `application/vnd.collection+json` 表述，你将可以应用 Collection+JSON 标准的规则。而如果表述是 `application/atom+xml` 格式的，你就会知道应该应用 AtomPub 的规则。

如果表述是 `application/json` 格式的，你就没这么幸运了，因为 JSON 标准并没有说明任何关于集合资源的内容。你正在使用一个自生自灭的独自维持的 API、一个 fiat 标准的 API。你将需要对你所使用的这个 API 进行更加仔细的审视。

POST-to-Append

集合的典型特征就是它在 HTTP POST 请求下的行为。除非集合是只读的（比如搜索结果的集合），否则客户端就可以通过 POST 请求向该集合添加子项。

当你向集合 POST 一个表述时，服务器便会基于你的表述创建一个新的资源，该资源将会成为这个集合最新的成员。回顾第 2 章，当向微博 API 发送 POST 时，一条新的帖子将会“插入”到该微博中。

PUT和PATCH

< 101

主要的集合标准都没有对集合在应答 PUT 和 PATCH 请求时的响应进行定义。一些应用通过实现这些方法来一次性修改集合中的多个元素，或是从集合中删除个别元素。

Collection+JSON、AtomPub 和 OData 都为子项定义了应答 PUT 方法的响应：它们认为 PUT 方法是客户端用以改变子项状态的方式。但是这些标准只是重复了 HTTP 标准中规定的内容，它们并没有为子项资源添加新的约束。PUT 是客户端用以改变任意 HTTP 资源状态的方式。

DELETE

3个主要标准都没有定义集合该如何响应 DELETE 请求。一些应用通过删除集合来实现 DELETE 方法，而另一些则会删除掉集合以及在集合中罗列的所有子项所对应的资源。

几个主要的集合标准都对子项如何响应 DELETE 进行了定义，但是，它们也只是重述了 HTTP 标准中所陈述的内容。DELETE 方法用于删除事物。

分页

一个集合有可能包含数百万的子项，但是服务器并没有义务在单个文档中提供数百万的链接。而最普遍的替代方案便是分页。服务器可以选择提供集合中的前 10 个子项，并且给客户端提供一个指向剩余项的链接：

```
<link rel="next" href="/collection/4iz6"/>
```

“next”的链接关系已经在 IANA 进行了注册，它的意义是“该连续内容的下一部分”，通过访问该链接你将可以获取到集合的第 2 页内容。你将可能无限次地访问 `real="next"` 链接，直到你达到该集合的末尾。

目前有很多用于在分页列表中导航的通用链接关系，它们包括“`next`”、“`previous`”、“`first`”、“`last`”以及“`prev`”（“`previous`”的同义词）。这些链接关系原本是为 HTML 定义的，但是现在它们都在 IANA 上注册过，所以你可以在任何媒体类型中使用它们。

某些基于集合的标准明确地定义了分页技术，而其他的则简单地假定你知道“`next`”和“`previous`”。Collection+JSON 便属于后者。它没有明确提供对分页的支持，但是你可以通过将它的通用媒体链接与 IANA 的通用链接关系结合来得到这项功能：

```
102 >
"links" : [
    {
        "name" : "next_page",
        "prompt" : "Next",
        "rel" : "next",
        "href" : "/collection/page/3",
        "render" : "link"
    },
    {
        "name" : "previous_page",
        "prompt" : "Back",
        "rel" : "previous",
        "href" : "/collection/page/1",
        "render" : "link"
    }
]
```

搜索表单

集合模式最后一项共有的特性是超媒体搜索表单。这同样也对那些非常大的集合带来了很大的帮助。搜索表单可以让客户端在无须下载集合全部内容的情况下找到自己感兴趣的部分。

Collection+JSON 和 OData 明确定义了它们的超媒体搜索表单格式。我曾在本章前面的内容中向你展示过 Collection+JSON 的搜索模板，而 AtomPub 并没有提供对搜索的原生支持，它假定你在需要这项特性时，会自行附加另一个标准，例如 OpenSearch。

Atom发布协议（AtomPub）

当时之所以开发 Atom 文件格式是为了将它作为 RSS 的一种可选方案，用于聚合新闻文章和博客。该文件格式由 RFC 4287 定义，并在 2005 年定稿。Atom 发布协议是一个用于编辑和发布新闻文章的标准化工作流，它使用 Atom 文件格式来作为表述格式。该协议由 RFC 5023 定义，于 2007 年定稿。在 REST API 的世界里，这些都属于非常早期的时间点。事实上，AtomPub 是第一个用于描述集合模式的标准。

下面是一个微博的 Atom 表述，此前我曾使用 Collection +JSON 格式展示过同一条微博。AtomPub 具有和 Collection+JSON 一样的概念，但是使用了不同的术语。相对于包含了“item”的“collection”，下面所采用的是包含了“entry”的“feed”。

```
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>You Type It, We Post It</title>
  <link href="http://www.youtypeitwepostit.com/api" rel="self" />
  <id>http://www.youtypeitwepostit.com/api</id>
  <updated>2013-04-22T05:33:58.930Z</updated>

  <entry>
    <title>Test.</title>
    <link
      href="http://www.youtypeitwepostit.com/api/messages/21818525390699506" />
    <link rel="edit"
      href="http://www.youtypeitwepostit.com/api/messages/21818525390699506" />
    <id>http://www.youtypeitwepostit.com/api/messages/21818525390699506</id>
    <updated>2013-04-22T05:33:58.930Z</updated>
    <author><name/></author>
  </entry>

  <entry>
    <title>Hello.</title>
    <link
      href="http://www.youtypeitwepostit.com/api/messages/3689331521745771" />
    <link rel="edit"
```

```
    href="http://www.youtypeitwepostit.com/api/messages/3689331521745771" />
<id>http://www.youtypeitwepostit.com/api/messages/3689331521745771</id>
<updated>2013-04-20T12:55:59.685Z</updated>
<author><name/></author>
</entry>

<entry>
<title>Pizza?</title>
<link
  href="http://www.youtypeitwepostit.com/api/messages/7534227794967592" />
<link rel="edit"
  href="http://www.youtypeitwepostit.com/api/messages/7534227794967592" />
<id>http://www.youtypeitwepostit.com/api/messages/7534227794967592</id>
<updated>2013-04-18T03:22:27.485Z</updated>
<author><name/></author>
</entry>

</feed>
```

该文档以媒体类型 `application/atom+xml` 的格式提供，并容许 AtomPub 客户端对该文档做出某些特定的假设。你知道你可以向集合的 `href` 发起 POST 请求从而向集合添加一个新的 Atom 条目。如果你想编辑该条目，你可以向条目的 `rel="edit"` 链接指定的 URL 发送 PUT 请求。又或者你想删除该条目则可以向该 URL 发送一个 DELETE 请求。

这些都没有什么让人感觉新奇的，这与 Collection+JSON 的方式非常相似，而且大部分都是重述了 HTTP 标准中的一些思想。

Collection+JSON 和 AtomPub 在概念上有个很大的差别。在 Collection+JSON 中并没有给“item”定义特别的应用语义，一个“item”看上去可以是任何事物。但是因为 Atom 是被设计用来聚合新的文章的，所以每个 AtomPub 条目看上去就像一篇新的文章。AtomPub feed 中的每个条目都必须拥有一个唯一的 ID（我使用了该微博的 URL 作为 ID）、一个标题（我使用了该微博的内容文本），以及它发布或最近一次更新的日期和时间。Atom 文件格式为新闻报道定义了一些应用语义：比如“副标题（subtitle）”和“作者（author）”这样的字段。Collection+JSON 则没有定义任何这些内容，它甚至没有要求集合的每个成员都必须具备一个永久链接（虽然你真的应该让它具备一个）。

104 尽管专注于新闻和博客文章，AtomPub 仍然是一个集合模式的完整的通用实现。谷歌是 AtomPub 最大的企业采用者，它使用 Atom 文档来展示视频、日历事件、电子表格的单元格以及地图上的地点等。

AtomPub 之所以被谷歌选择的秘密在于其可扩展性。它允许你定义任何你所关心的应用语义来扩展 Atom 的词汇表。谷歌为它所有基于 Atom 的 API 定义了一个公共的 Atom 扩展，称为 GData，并且又为视频、日历、电子表格等定义了额外的扩展。

一些关于 AtomPub 在遵守集合模式方面的有趣事实：

- 因为新闻文章通常被归类到一或多个分类下，所以 Atom 文件格式定义了一个简单的分类系统，并且 AtomPub 为分类列表单独定义了一种媒体类型 (`application/atomcat+xml`)。
- AtomPub 同样为服务文档（Service Document^{注2}，即集合的集合）定义了一种媒体类型。
- Atom 是一种严格的基于 XML 的文件格式。AtomPub 这套技术并不能提供 JSON 表述，这让 Ajax 客户端很难使用 AtomPub API。谷歌认识到了这个问题，并在提供 AtomPub 表述的同时，为文档添加了 JSON 表述。但是谷歌只是将它作为一种 *flat* 标准，而不是一项鼓励大家复用的标准。
- 尽管 Atom 是一种 XML 的文件格式，客户端也可以向一个 AtomPub 的 API POST 二进制文件。一个在服务器端的上传文件将被表示为两个不同的资源：一个是媒体资源，其表述是一组二进制数据；另一个是 Entry 资源，它的表述是 Atom 格式中的元数据。该特性允许你在使用 AtomPub 保存包含了描述信息和相关链接的 Atom 文档的同时，也将你的一组照片或音频文件存储起来。

AtomPub 插件标准

因为它们非常容易扩展，所以 Atom 和 AtomPub 经常被用来作为很多小型插件标准的基础，从而增强集合模式：

- Atom 线索扩展（Atom Threading Extensions，由 RFC 4685 定义）可以更加容易地描述常见于邮件线索和留言板的会话结构。这个扩展并不大——仅仅包含一些额外的标签和一个称为“replies”的新的链接关系。
- Atom `deleted-entry` 元素（由 RFC 6721 定义）让服务器为那些从集合中被删除的子项建立一个“墓碑”，而不是简单地移除它们。这等于告诉客户端需要去清除被删除的条目，而不是一直被缓存着。
- RFC 5005（“Feed 分页和归档（Feed Paging and Archiving）”）定义了“归档 feed”的概念，这是一种对跨多个资源的大型 feed 更加有效的分页方式。它还定义了“`next-archive`”、“`prev-archive`”和“`current`”3 个链接关系来替代“`next`”、“`prev`”和“`first`”。
- OpenSearch (<http://www.opensearch.org/Specification/OpenSearch/1.1>) 是为基于 XML 的搜索协议而制定的联合标准。一个 OpenSearch 文档等价于一个 HTML 表单，也等价于 Collection+JSON 文档中的“`queries`”片段。客户端填写完表单可以执行一次搜索（通过 HTTP 的 GET 请求）并获取一份搜索结果的 Atom feed。

注 2 关于服务文档的内容见 <http://bitworking.org/projects/atom/rfc5023.html#appdocs>。——译者注

OpenSearch 定义了一个新的链接关系 “`search`”，通过该链接关系可以将 Atom feed 链接到一个 OpenSearch 文档。OpenSearch 并不是特定于 Atom 的，你的 web 浏览器的搜索框也可以使用 OpenSearch。OpenSearch 让你可以搜索不同的网站，无须真正去访问那些网站和使用它们的 HTML 搜索引擎。我之所以将 OpenSearch 纳入本章，是因为 AtomPub 并没有定义搜索协议，但这又是你应该会使用到的。在第 10 章中，我将会对 OpenSearch 有更详细的讨论。

- PubSubHubbub (<http://code.google.com/p/pubsubhubbub/>) 是一个企业标准，它描述了一个发布订阅的协议，该协议可以让客户端进行注册，并在 Atom feed 更新时接收到相应的通知。它定义了一个新的链接关系：“`hub`”。

所有由这些插件标准定义的链接关系都在 IANA 进行了注册。这意味着“`replies`”、“`next-archive`”、“`prev-archive`”、“`current`”、“`search`”和“`hub`”都是通用的关系，可以在任何地方使用而无须特别说明。其中“`search`”链接关系是由 OpenSearch 定义的，但是 `rel="search"` 并不是说“这是一个指向 OpenSearch 文档的链接”，它是指“这是一个指向某个类型的搜索表单的链接”。

即使你现在没有使用 AtomPub，你仍可以从那些多年致力于 Atom 扩展的人的工作成果中受益。他们为很多常用操作创建了标准的词汇表，你只需要选择是否要复用它。

为什么不是每个人都选择使用 AtomPub?

在 RFC 定稿到现在已经六年有余，尽管还有各种各样的插件标准，但是肯定地讲，AtomPub 并没有流行起来。除了谷歌之外，该标准没有得到任何来自其他公司的推动，甚至连谷歌看起来也正在逐步淘汰该标准。AtomPub 到底哪里出了问题？

这个问题要追溯到 2003 年时的一个技术决策：AtomPub 的表述被设计为 XML 文档。这在 2003 年看来是个非常明智的决定，但是在接下去的 10 年里，浏览器内的 API 客户端变得越来越流行，而 JSON 这种表述格式获得了势不可挡的人气。对于浏览器中的 JavaScript 代码来说，处理 JSON 比处理 XML 容易太多了。时至今日，绝大多数的 API 都只提供 JSON 格式的表述，或者是在 XML 和 JSON 表述两者间提供选择。AtomPub 几乎已经销声匿迹了^{注3}。

那么为什么我们还要在本书中给 AtomPub 专门开辟了这么一大段的章节呢？部分原因是该标准本身并没有什么问题，就它的原理来说它工作得很不错。它的历史意义在于它是“集合”API 模式的首个通用实现。它的插件标准所定义的 IANA 注册的连接关系可以在其他的表述格式中被干净地复用。

注 3 Joe Gregorio 是 Atom 和 AtomPub 的主要贡献者，他在博文中 (<http://bitworking.org/news/425/atompub-is-a-failure>) 也提到了相同的情况。

但是 AtomPub 的故事同样也说明了一个道理：“标准没有什么问题”还不够好。人们都很怕麻烦，他们不会刻意去学习一个标准，除非它契合了他们的需求。采用一个基于 JSON 的 fiat 标准来重新实现“集合”模式会更加容易些，这也是数以千计的开发者曾经做过并还在继续这样做着事情。

我写本书的主要目标就是为了终结这种重复的劳动。我不知道答案会不会是 Collection+JSON，又或者是我在后续章节中将会提到的任何其他的超媒体格式。很可能答案并不唯一。

我知道“集合”模式已经证明了它的优势。问题在于我们是否还会允许我们自己一遍遍地重复发明轮子。

语义挑战：我们应该怎么做？

请牢记，语义的挑战是：“我们该如何编写程序让计算机来决定单击哪个链接呢？”为了回答这个问题，我们必须在 HTTP 的协议语义（由 URL 来标识并响应 GET 和 PUT 等方法的一般性“资源”）和你特定、独特的 web API 的应用语义（微博服务、支付处理或任何你正在做的事情）之间建立起桥梁。

像 Maze+XML 这样的领域特定设计通过专门设计的超媒体类型，以及为你的问题空间特别设计的链接关系来为语义鸿沟架起桥梁的。但是这需要做很多的工作，而且几乎没有人在这项工作上走得远。

集合模式识别出了两种不同类型的资源：子项类型的资源（通常会响应 GET、PUT 和 DELETE 请求）以及集合类型的资源（通常会响应 GET 和 POST-to-append 请求）。一个集合类型的资源包含了一定数量的子项类型资源，它的表述链接到这些子项，并包含了它们的部分表述。

集合和子项之间的差异形成了 HTTP 协议语义之上的应用语义的边界层。◀107 Collection+JSON、AtomPub 和 OData 在集合和子项之间都定义了相同的差异点。因为这些区别的存在，IANA 的很多链接关系顿时便有了意义：用于对集合进行导航的关系，比如“first”、“next”以及“next_archive”；“search”关系用于对集合进行搜索；“item”关系用于指向集合中的一个子项；“edit”关系用于编辑一个子项；而“collection”关系将一个子项和包含它的集合连接起来。

但是就一个“item”而言并没有什么特殊的，它几乎就是一个像“资源”一样笼统和不明确的术语。在一个微博 API 中，一个“item”只有少量的文本和一个时间戳。在一个支付处理器中，一个“item”将包含贷方、借方、支付的方式以及金额的总数。在集合

模式的应用语义和你自有的 API 的应用语义之间还是存在着巨大的鸿沟的。

再看一眼下面微博的 Collection+JSON 表述：

```
{ "collection":  
  {  
    "version" : "1.0",  
    "href" : "http://www.youtypeitwepostit.com/api/",  
  
    "items" : [  
      {  
        "href" :  
          "http://www.youtypeitwepostit.com/api/messages/21818525390699506",  
        "data" : [  
          {  
            "name" : "text",  
            "value": "Test.",  
            "prompt" : "The text of the microblog post."  
          },  
          {  
            "name" : "date_posted",  
            "value": "2013-04-22T05:33:58.930Z",  
            "prompt" : "The date the microblog post was added."  
          }  
        ]  
      }  
    ]  
  }  
}
```

HTTP 告诉我们如何来编辑该 item：以某种方式改变表述并将它 PUT 回去。Collection+JSON 则告诉我们表述应该是什么样子的。它应该看上去像一个被填写完整的 Collection+JSON “模板”：

```
PUT /api/messages/21818525390699506 HTTP/1.1  
Host: www.youtypeitwepostit.com  
Content-Type: application/vnd.collection+json  
"template" : {  
  "data" : [  
    {"prompt" : "Text of message", "name" : "text", "value" : "The new value"}  
  ]  
}
```

但是 Collection+JSON 并没有说明“text”和“date_posted”的意义。为了理解这些事物，人们必须阅读在“prompt”元素中那些人类可读的说明。这便是 Collection+JSON 用以为语义鸿沟架起桥梁的方式。Maze+XML 处理语义鸿沟的方式是提前在媒体类型的规范中对应用语义进行定义。而 Collection+JSON 将应用语义写入散落于表述各处的

“prompt”元素。

如果每个人都在它们的 API 中使用 Collection+JSON，我们将可以共享“集合”的共同定义。但是将可能会出现 57^{注4} 种关于“item”的不同定义，57 套不同“prompt”值的数据元素。一些 API 会将文本字段称为“text”，而其他的则可能将它称为“content”或“post”或是“blogPost”，但其实它们都在使用不同的单词描述相同的事物。我们仍将拥有 57 种不同的微博 API。

所以还没有达到我们的目的，我们仍然需要更多别的东西。

注 4 此处的 57 源自前言中提到的 ProgrammableWeb (<http://www.programmableweb.com/>)，它收录了 57 种微博 API。——译者注

纯-超媒体设计

集合模式的功能很强大，但是它不具有普适性。从技术上说，第 5 章中的迷宫游戏客户端可以采用 Collection+JSON 格式的表述来实现，但是这看起来很糟糕。游戏客户端的意义在于客户端每次只能看到一个单元格。在这其中并不存在需要“集合”到某个集合中的东西。迷宫游戏的应用语义和集合模式所提供的语义并不一致。

没有人要求你必须使用集合模式，但是对于 API 而言，它是最流行的设计模式。如果你想要实现一些其他的模式，或者如果你的 API 设计方案并不符合任意一种特别的模式，你可以使用纯超媒体（pure hypermedia）来描述 API 的语义。你并不需要创造一种类似于 Maze+XML 的拥有自己的媒体类型的崭新标准。你可以用一种通用的超媒体语言（generic hypermedia language）来表示你的资源的状态。

在本章中，我将对一些采用通用的超媒体语言作为它们的表述格式的 API 展开讨论。我会讲到很多新型的表述格式，但是我讲解的重点将是你已经很熟悉的很老的格式：HTML。

为什么是HTML？

我们要以万维网（由无数的供人类阅读的文档组成的网络）为背景来认识 HTML。由于 HTML 的广为流行，只要 API 中有任何一部分是提供供人类消费的文档的，HTML 都将是显而易见的选择。即便你的 API 的其他部分提供的都是基于 XML 或者 JSON 的表述，对于需要呈现给用户的那部分，你还是可以使用 HTML 作为表述的。HTML 如此流行以至于每个现代操作系统都会在发布时捆绑一款用于调试基于 HTML 的 web API 的工具：web 浏览器。

即便是对于那些定位于仅供机器调用的 API 而言，HTML 也具有显著的优势。和 XML 以及 JSON 相比，HTML 对文档施加了更多的结构信息，但是它又不像 Maze+XML 那样结构信息太多以至于只能用于解决某个特定问题。HTML 处于中间层，类似于 Collection+JSON。

不同于一无所有的 XML 和 JSON，HTML 打包了一整套标准化的超媒体控件。但是 HTML 的控件非常通用，并且没有和某个特定的问题空间绑定。Collection+JSON 为搜索查询定义了一个特殊的超媒体控件；而 HTML 则定义了一个可以用于任何目的的超媒体控件（`<form>` 标签）。

最终，出现了一种流行的说法。那就是，HTML 是迄今为止世界上最流行的超媒体格式。如今已经出现了许许多多的 HTML 解析和生成工具，并且大部分的开发人员都知道如何读取一个 HTML 文档。HTML 如此流行，使得它成为了人们为了消除语义鸿沟而正在做出的两项巨大努力的基础标准，这两项工作分别是：微格式（microformats）和微数据（microdata），我将会在本章后面部分进行介绍。

HTML的能力

110 HTML 被设计用来展示文本文档的嵌套结构。任何一个 HTML 标签都可以包含由文本内容和其他标签组成的组合体：

```
<p>
  This 'p' tag contains text
  <a href="http://www.example.com/"> and a link </a>.
</p>
```

这个文档并不符合任何数据结构（英文句子也几乎不符合任何的数据结构）。但是，如同你在 JSON 文档中发现的数据结构那样，HTML 文档是可以包含相同的数据结构的。排序的列表可以使用 `` 标签，键值对的集合可以用 `<dl>` 标签（被称为“dl”是由于 HTML 将这种数据结构称为“definition list”）。

HTML 同样支持未排序的列表（`` 标签）、二维数组（`<table>` 标签）以及不考虑标准的数据结构而将标签进行随意分组的方式（使用 `<div>` 和 `` 标签）。

超媒体控件

更重要的是，HTML 拥有内建的超媒体控件。我在第 4 章中曾提到过这些控件，下面再次对一些最重要的超媒体控件做简要的讲解：

- `<link>` 标签和 `<a>` 标签是简单的转出链接，它们就像 Maze+XML 中的 `<link>`

标签一样。它们会命令客户端来向某个指定的 URL 发起一次 GET 请求用以获取一个表述。这个表述之后就会变为当前视图。

- 标签和 <script> 标签是内嵌链接。它们命令客户端自动地向另一个资源发起 GET 请求，并将该资源的表述内嵌到当前视图中。 标签表示将收到的表述作为一幅图片嵌入；<script> 标签表示将收到的表述作为代码执行。HTML 同时还定义了一些其他类型的内嵌链接，但是上面这两个是最主要的。
- 当 <form> 标签将字符串“GET”作为它的 method 属性（也就是 <form method="GET">）的时候，它充当了一个模板化的转出链接。它会像 URI 模板以及 Collection+JSON 中的查询数据槽那样工作。服务器为客户端提供一个基础 URL 以及若干用于输入的字段（HTML<input> 标签）。客户端会为这些字段插入值，并将这些值与基地址合并来组成一个独一无二的 URL，然后向这个 URL 发起一次 GET 请求。
- 当 <form> 标签使用“POST”作为它的 method 属性时，它所描述的就是一个能够做任何事情的 HTTP POST 请求。那些 <input> 标签还是存在的，但是它们不再用于生成请求 URL，而是用于创建一个媒体类型为 application/x-www-form-urlencoded 的实体消息体。请求 URL 是在 <form> 标签的 action 属性中写死的。

应用语义插件

HTML 为一个非常通用的应用（人类可读的文档）定义了应用语义。HTML 标准为相片、标题、章节、列表以及其他在新闻和书籍中常见的结构化元素都定义了标签。

HTML 没有为迷宫或者迷宫中的单元格定义标签。那不是它的应用范围。但是 HTML 不同于 Maze+XML 和 Collection+JSON 的地方就在于我们可以很容易地在 HTML 的应用范围之外的地方使用它。HTML4 定义了三个通用的属性，我们可以使用它们来添加一些 HTML 标准没有定义的应用语义（HTML5 定义的更多，我将在后面章节中介绍）。

rel 属性

HTML 的 <a> 和 <link> 标签都有一个称为 rel 的属性，这个属性用来定义所链接的资源和与当前资源的关系。我们之前已经见到过 rel：

```
<link rel="stylesheet" type="text/css" href="/my_stylesheet.css"/>
```

这段 HTML 代码是说 /my_stylesheet.css 资源被获取后应该自动地应用到当前页面上以修改其页面样式。在这里，HTML 的 <link> 标签是作为一个内嵌的链接而提供服务的。当 <link> 标签的 rel 使用不同的值（比如，rel="self"）时，它就是作为一个转出

链接^{注1}而提供服务了。

- 112 ➤ 尽管有一些关于链接关系的标准列表（就像我在第 5 章中提到的 IANA 注册表），但是这些字符串“stylesheet”和“self”也并没有什么特殊之处。某些人为了增强 HTML 而制定了这些字符串。如果你想发布一个 HTML 格式的迷宫 API，你可以继续在 HTML 文档中采用 Maze+XML 中定义的链接关系（“north”、“south”等）。这会为 HTML 格式提供一些它原本并不拥有的应用层语义：迷宫以及迷宫中的单元格的语义。你也可以创建扩展链接关系（看起来像 URL 的链接关系）来描述你的应用中不同资源之间的特定关系。

制定你自己的链接关系的一个缺点就是：你的用户将不清楚那些链接关系的含义，你将需要在某个 profile 中将这些应用语义记录下来（见第 8 章）。

id属性

几乎任何 HTML 标签^{注2}都可以为 id 属性赋值。这个属性唯一地标识了文档中的一个元素：

```
<div id="content">
```

如果你碰巧正在找一个 id="content" 的标签，很好，上面这个就是了。一个 HTML 文档中不能包含两个拥有相同 ID 的元素。

我不建议使用 id 属性来和你的应用层语义产生关联。在一个文档中 IDs 必须唯一的要求太苛刻了。这会使得在两个 HTML 文档定义了相同的 id 的情况下，我们不能够将这两个文档合并成一个更大的文档。

class属性

几乎每一个 HTML 标签^{注3}都可以为 class 属性赋值。这是 HTML 中最为灵活的一个语义属性。在万维网中，class 通常被用于应用 CSS 格式化，但是它同样可以用于传递某些标签的应用语义；按照字面意思，它属于什么“class”。

下面是一个简单的例子，它是一个 `<div>` 标签，但是它包含了两个 `` 标签：

注 1 HTML4 同样允许链接拥有 rev 属性，它是 rel 的反义词。rev 的值所代表的是这个资源与所链接的资源的关系。在指向下一页的链接中，rel 应该是 next，而 rev 应该是 previous。这也就证明了 rev 属性不是必要的。而且它已经从 HTML5 中移除了。这也就是我只在脚注提到它的原因。请不要将 rel 和它的反义词混淆。

注 2 在 HTML4 中，不能拥有 id 属性的标签有：base、head、html、meta、script、style 以及 title。而在 HTML5 中任何标签都可以拥有 id 属性。我将这个标签列表重复打印出来，这样你可以明白这很可能不是一个问题。

注 3 在 HTML5 中，所有标签都可以拥有 class 属性。而在 HTML4 中，我前面的脚注中提到的 7 个标签既不能定义 id 属性，也不能定义 class 属性。Param 标签可以定义 id 属性，但是它不能定义 class 属性。重申一遍，这很可能不是一个问题。

```
<div class="vcard">
  <span class="fn">Jennifer Gallegos</span>
  <span class="bdy">1987-08-25</span>
</div>
```

单独而言，`<div>`标签没有任何含义——它仅仅是将其他的标签集合到一起。一个``标签自身也没有任何含义。但如果说，我告诉你`vcard` class 会将一个人的个人信息整合到一起（此时，不要担心我如何告诉你这些信息；我会在后面讲到），我告诉你用`fn` class 标记的标签包含的是一个人的姓名；而用`bdy` class 标记的标签包含的是一个人的 ISO 8601 格式的出生日期，那会怎么样呢？

这个`<div>`标签立刻就成为对某个人的说明介绍了，它有具体的含义了。现在你也就知道“Jennifer Gallegos”是某个人的姓名，而不是某本书的标题了。你也就知道“1987-08-25”是某个特定格式的日期，而不是一个恰好看起来像日期的随机字符串了。当你明白`class`的某些值的含义的时候，你也就理解了那些 HTML 规范并没有定义的应用语义。

在同一文档中的多个标签可以拥有相同的`class`，而且一个的标签可以拥有多个`class`值，这些值之间用空格进行分隔：

```
<ul>
  <li><a class="link external" href="http://www.example.com/">Link 1</a></li>
  <li><a class="link external" href="http://www.example.org/">Link 2</a></li>
  <li><a class="link internal" href="/page2">Link 3</a></li>
</ul>
```

如果你曾经尝试过使用`id`属性来作为应用语义的一部分的话，那我现在建议你使用`class`属性来进行代替。和`id`属性不同，在一个表述中的多个标签可以拥有相同的`class`属性。

微格式

我选择了一些相当模糊的 CSS`class`名称来使得`<div>`标签和两个``标签成为某个人的说明描述：“`vcard`”，`fn` 和 `bdy`。如果让我自己来制定`class`名称的话，我会用类似于`birthday`这样描述更准确的名称。但是我没有这样做。我采用了一个已有标准`hCard` (<http://microformat.org/wiki/hcard>) 中的名称。如果你在某个 HTML 标签中见到了`class="vcard"`字样，你就会知道这个标签中的所有内容都应该按照`hCard`标准来进行解释。

和`Maze+XML`相同的是，`hCard`标准并不和哪个 RFC 或者互联网草案相关。与`Maze+XML`不相同的是，`hCard`并不是一个个人标准。`hCard`是一种微格式：一种轻量级的工业标准，它是通过 wiki 上的非正式的协作方式定义的，而不像 RFC 那样要经过

正式的 IETF 流程。

查阅 hCard 标准之后，你就会发现 `fn` class 是用来标记一个人的全名（full name）的，而 `bday` class 是用来标记一个人的 ISO 8601 格式的出生日期（date of birth）的。现在，当一个文档使用这些 CSS class 的时候，你就知道这个文档的含义了。HTML 标准并没有对于姓名或者生日做任何的说明，但是 hCard 标准却对这些内容进行了讨论。

114 微格式使得你可以向 HTML 添加额外的应用语义。HTML 的 `class` 属性，加上 hCard 微格式，就可以让你建立一个能够描述一个人的 HTML 文档。

hCard 仅仅定义了 `class` 属性的值。我所使用的 `` 标签以及 `<div>` 标签，对于 hCard 而言，毫无意义：我可以使用任何其他的标签。因为几乎所有的 HTML 标签都支持 `class` 属性，我可以编写一段非结构化的文本，它同时也是一个 hCard 文档：

```
<p class="vcard">我的名字是 <i class="fn">Jennifer Gallegos</i> 我  
出生于 <date class="bday">1987-08-25</date>.</p>
```

人类可以将这个表述当作一句话来进行阅读。而 hCard 处理器将忽略所有的“无关的”文字，而将重点放在使用 hCard 的 CSS class 的标签上。

尽管 hCard 微格式没有通过正式的标准化过程，但是它是基于一个已经完成了这些过程的标准而创建的。这个标准就是：vCard，一个在 RFC 6350 中定义的用来展示商业名片的纯文本格式。

我曾在第 5 章中提及过 vCard，并将 vCard 视为一个缺少超媒体控件的领域特定标准的例子。hCard 不过是 HTML 版的 vCard 而已。这也就是 hCard 文档中的最上层的 `class` 的值是 `vcard` 而不是 `hcard` 的原因。

对于哪些类型的信息常常会被添加到商业名片，专家们做了非常多的成本很高的研究调查和长期的讨论，vCard RFC 就是这些研究和讨论的结晶。正如我在第 5 章所说的，我们没有理由仅仅因为 vCard 没有超媒体控件而重复这些研究和讨论。我们可以照抄 vCard 的语义，并将它们应用到一个通用的超媒体语言：HTML 上。

hMaze微格式

在本节中，我将像 hCard 对 vCard 的改造一样对 Maze+XML 进行改造。我会将这个为特定领域（迷宫）设计的、非 HTML 标准改造成一个 HTML 微格式。这样，我就可以使用 HTML 来表示那些基本 HTML 标准所不理解的领域的语义了。

和“hCard”一样，我将我的新的微格式称为“hMaze”（“h”代表“HTML”）。我的微

格式定义了一些特别的 CSS class：

hmaze

表明 hmaze 文档的父标签，类似于 hCard 的 vcard class。

collection

可能在 hmaze 中出现。用于描述迷宫集合。

maze

可能在 hmaze 中出现。用于描述一个单独的迷宫。

error

<115

可能在 hmaze 中出现。用于描述一条错误消息。

cell

可能在 hmaze 中出现。用于描述迷宫中的某个单元格。

title

可能在 cell 中出现。用于包含单元格的名称。

微格式同样可以定义链接关系，我照抄了 Maze+XML 所定义的全部链接关系。这些关系：north、south、east、west、exit 以及 current 在 class="cell" 的标签中都有着特定的含义（更确切地说，是 Maze+XML 标准定义的特定的含义）。当 maze 链接关系出现在 class="collection" 的标签中时，它也有特殊含义（就像在 Maze+XML 中那样，它链接到一个特定的迷宫）。

这就是 hmaze 微格式。最起码是 hmaze 微格式的第一个版本。如果我打算将其发布到微格式 wiki 上，我就需要编写更多的 CSS class，比如一些和错误关联的 class，但是这个作为例子来说已经足够好了。这个微格式可以展示任何 Maze+XML 所能展示的迷宫，但是它使用的却是 HTML。

我的微格式只定义了 class 和 rel 的值。就像 hCard 一样，标签的选择权留给了服务器。服务器可以提供或多或少类似于 Maze+XML 的古板的 HTML 文档：

```
<div class="hmaze">
<div class="cell">
  <div class="title">
    Hall of Pretzels
```

```
</div>
<div>
<a href="/cells/143" rel="west"/>
<a href="/cells/145" rel="east"/>
</div>
</div>
</div>
```

或者服务器也可以用一种人类可读的方式来展示相同的数据，这样人类就可以使用他们的 web 浏览器作为 API 客户端了：

```
<div class="hmaze">
<div class="cell">
<p><b class="title">Hall of Pretzels</b></p>

<ul>
<li><a href="/cells/143" rel="west">Go west</a></li>
<li><a href="/cells/145" rel="east">Go east</a></li>
</ul>
</div>
</div>
```

116 这两个文档都是合法的 hMaze 文档，就 hMaze 而言，它们拥有等价的应用语义。重要的是你要按照 hMaze 规范所要求的那样来使用 `class` 和 `rel` 属性（就 HTML 本身而言，这两个文档拥有不同的应用层语义，因为 HTML 的“应用”是人类可读的文档）。

微数据

微数据是针对 HTML5 而对微格式的概念进行的改进。你也看到了，微格式是一种侵入式的。HTML 的 `class` 属性本来是设计用来传递与可视化显示相关的信息的（通过 CSS），不是用来传递任何应用语义内容的。

HTML 微数据⁴ 介绍了 5 种新的属性：`itemprop`、`itemscope`、`itemtype`、`itemid` 和 `itemref`，它们是专门用来展示应用语义的。这些属性可以出现在任何 HTML 标签中。

我将重点关注它们之中的前 3 个属性。`itemprop` 属性的用法和微格式中的 `class` 属性的用法相同。`Itemscope` 属性是一个 Boolean 属性，用来表明标签中是否包含微数据。`Itemtype` 属性是一个超媒体控件，它用来告诉客户端从哪里找到这个微数据的含义。

通过稍加改进，微格式的大部分信息都可以展示为微数据。下面的 HTML 文档就展示了一种微数据类型，它是对 hMaze 的轻微的变形体：

```
<div itemscope itemtype="http://www.example.com/microdata/Maze">
```

注 4 一个开放标准，在 W3C 规范中定义，目前是以草案形式出现的。