TURING 图灵程序设计丛书

HTTP: The Definitive Guide

HTTP

权威指南



[美] *David Gourley, Brian Totty*
Marjorie Sayer, Sailu Reddy, Anshu Aggarwal 著

陈涓 赵振平 译

O'REILLY®

人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

TURING 图灵程序设计丛书

HTTP权威指南

HTTP: The Definitive Guide

[美] David Gourley Brian Totty Marjorie Sayer
Sailu Reddy Anshu Aggarwal 著
陈涓 赵振平 译

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo
O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北 京

图书在版编目（CIP）数据

HTTP权威指南 /（美）古尔利（Gourley,D.）等著；陈涓，赵振平译. -- 北京 : 人民邮电出版社，2012.8
（图灵程序设计丛书）（2012.10重印）
书名原文: HTTP:The Definitive Guide
ISBN 978-7-115-28148-7

Ⅰ. ①H… Ⅱ. ①古… ②陈… ③赵… Ⅲ. ①计算机网络－通信协议 Ⅳ. ①TN915.04

中国版本图书馆CIP数据核字(2012)第095358号

内 容 提 要

本书是HTTP及其相关核心Web技术方面的权威著作，主要介绍了Web应用程序是如何工作的，核心的因特网协议如何与架构构建块交互，如何正确实现因特网客户端和服务器等。

本书适合所有想了解HTTP和Web底层结构的人阅读。

图灵程序设计丛书

HTTP权威指南

◆ 著 [美] David Gourley Brian Totty Marjorie Sayer Sailu Reddy Anshu Aggarwal
译 陈 涓 赵振平
责任编辑 李 盼
执行编辑 丁晓昀

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京 印刷

◆ 开本：800×1000 1/16
印张：45
字数：1067千字 2012年10月第1版
印数：5－8 000册 2012年10月北京第2次印刷

著作权合同登记号 图字：01-2011-1852号

ISBN 978-7-115-28148-7

定价：109.00元

读者服务热线：(010)51095186转604 印装质量热线：(010)67129223
反盗版热线：(010)67171154

版权声明

©2002 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2012. Authorized translation of the English edition, 2012 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.

英文原版由 O'Reilly Media, Inc. 出版 2002。

简体中文版由人民邮电出版社出版，2012。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者 —— O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

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

前言

HTTP（Hypertext Transfer Protocol，超文本传输协议[①]）是在万维网上进行通信时所使用的协议方案。HTTP 有很多应用，但最著名的是用于 Web 浏览器和 Web 服务器之间的双工通信。

HTTP 起初是一个简单的协议，因此你可能会认为关于这个协议没有太多好说的。但现在，你手上拿着的却是一本将近两斤重的书。如果你想知道我们怎么会写出一本 700 多页的关于 HTTP 的书，就去看看目录吧。本书不仅仅是一本 HTTP 首部参考手册，它还是一本名副其实的 Web 架构“圣经”。

本书中，我们会将 HTTP 中一些互相关联且常被误解的规则梳理清楚，并编写了一系列基于各种主题的章节介绍 HTTP 各方面的特性。纵观全书，我们对 HTTP“为什么”这样做进行了详细的解释，而不仅仅停留在它是“怎么做”的。而且，为了节省大家寻找参考文献的时间，我们还介绍了很多 HTTP 应用程序正常工作所必需且重要的非 HTTP 技术。在条理清晰的附录中，可以找到按照字母排序的首部参考（这些首部构成了最常见的 HTTP 文本的基础）。我们希望这种概念性的设计有助于读者更好地使用 HTTP。

本书是为所有希望理解 HTTP 和 Web 底层结构的人编写的。软硬件工程师也可以将本书作为 HTTP 及相关 Web 技术参考书使用。系统架构师和网络管理员可以通过本书更好地了解如何设计、实现并管理复杂的网络架构。性能工程师和分析人员可以从缓存和性能优化的相关章节中获益。市场营销和咨询专家还可以通过概念介绍更好地理解 Web 技术的前景。

① HTTP 译为“超文本传输协议”，其中“transfer”使用了“传输”的含义，但依据 HTTP 制定者之一 Roy Fielding 博士的论文，“transfer”表示的是“（状态的）转移”，而不是“传输”。怎样翻译才更符合 HTTP 的原意，其讨论可参见图灵社区的文章，地址是 iturning.com.cn/article/details/1817。

本书澄清了一些常见的误解，推荐了“各种业内诀窍”，提供了便捷的参考资料，并且用通俗易懂的语言阐述了枯燥且令人费解的标准规范，还详细探讨了 Web 正常工作所必需且互相关联的技术。

本书创作历时良久，是由很多热衷于因特网技术的人共同完成的，希望它能对你有所帮助。

运行实例：Joe的五金商店

本书的很多章节都涉及了一个假想的在线五金与家装商店示例，通过这个“Joe 的五金商店”来说明一些技术概念。我们为这个商店构建了一个真实的 Web 站点（http://www.joes-hardware.com），以便大家能够测试书中的部分实例。只要本书仍在销售，我们就会一直维护好这个 Web 站点。

本书内容

本书包含 21 章，分为 5 个逻辑部分（每部分都是一个技术专题），以及 8 个很有用的附录，这些附录包含了参考资料，以及对相关技术的介绍。

第一部分　HTTP：Web 的基础
第二部分　HTTP 结构
第三部分　识别、认证与安全
第四部分　实体、编码和国际化
第五部分　内容发布与分发
第六部分　附录

第一部分用 4 章的篇幅描述了 Web 的基础构件与 HTTP 的核心技术。

- 第 1 章简要介绍了 HTTP。
- 第 2 章详细阐述了统一资源定位符（Uniform Resource Locator，URL）的格式，以及 URL 在因特网上命名的各种类型的资源，还介绍了统一资源名（Uniform Resource Name，URN）的演变过程。
- 第 3 章详细介绍了 HTTP 报文是如何传送 Web 内容的。
- 第 4 章解释了 HTTP 连接管理过程中一些经常会引起误解且少有文档说明的规则和行为。

第二部分重点介绍了 Web 系统的结构构造块：HTTP 服务器、代理、缓存、网关以及机器人应用程序。（当然，Web 浏览器也是一种构造块，但在本书的第一部分已经对其进行过很详细的介绍了。）第二部分包含以下 6 章。

- 第 5 章简要介绍了 Web 服务器结构。
- 第 6 章深入研究了 HTTP 代理服务器，HTTP 代理服务器是作为 HTTP 服务与控制平台使用的中间服务器。
- 第 7 章深入研究了 Web 缓存的问题。缓存是通过保存常用文档的本地副本来提高性能、减少流量的设备。
- 第 8 章探讨了网关和应用服务器的概念，通过它们，HTTP 就可以与使用不同协议（包括 SSL 加密协议）的软件进行通信了。
- 第 9 章介绍了 Web 上的各种客户端类型，包括无处不在的浏览器、机器人和网络蜘蛛以及搜索引擎。
- 第 10 章讲述了仍在研究之中的 HTTP 协议：HTTP-NG 协议。

第三部分提供了一套用于追踪身份、增强安全性以及控制内容访问的技术和技巧。包含下列 4 章。

- 第 11 章讨论了一些识别用户的技术，以便向用户提供私人化的内容服务。
- 第 12 章重点介绍了一些验证用户身份的基本方式。这一章还对 HTTP 认证机制与数据库的接口问题进行了研究。
- 第 13 章详述了摘要认证，它是对 HTTP 的建议性综合增强措施，可以大幅度提高其安全性。
- 第 14 章说明了因特网的密码体系、数字证书以及 SSL。

第四部分涵盖 HTTP 报文主体和 Web 标准，前者包含实际内容，后者描述并处理主体内容。第四部分包含以下 3 章。

- 第 15 章介绍了 HTTP 内容的结构。
- 第 16 章探讨了一些 Web 标准，通过这些标准，全球范围内的用户都可以交换以不同语言和字符集表示的内容。
- 第 17 章解释了一些用于协商可接受内容的机制。

第五部分介绍了发布和传播 Web 内容的技巧。包括以下 4 章。

- 第 18 章讨论了在现代的网站托管环境中布署服务器的方式以及 HTTP 对虚拟网站托管的支持。
- 第 19 章探讨了一些创建 Web 内容，并将其装载到 Web 服务器中去的技术。
- 第 20 章介绍了能够将输入 Web 流量分散到一组服务器上去的一些工具和技术。
- 第 21 章介绍了一些日志格式和常见问题。

第六部分是一些很有用的参考附录，以及相关技术的教程。

- 附录 A 详述了统一资源描述符（Uniform Resource Identifier，URI）方案所支持的协议。

- 附录 B 列出了 HTTP 的响应代码，方便使用。
- 附录 C 提供了 HTTP 首部字段的参考列表。
- 附录 D 列出了大量的 MIME 类型，解释了 MIME 类型的注册方式。
- 附录 E 介绍了 HTTP 认证中使用的 Base-64 编码。
- 附录 F 详述了如何实现 HTTP 中的各种认证方案。
- 附录 G 定义了 HTTP 首部的语言标签值。
- 附录 H 列出了用以支持国际化 HTTP 的字符编码。

每章都包含很多实例，以及到其他相关的参考资料的链接。

排版约定

本书使用了下列排版约定。

- 楷体

用于 URL、C 函数、命令名、MIME 类型、新术语的定义以及重点内容。

- 等宽字体

用于计算机的输出、代码以及所有文字文本。

- 加粗等宽字体

用于用户的输入。

意见及问题

请将有关此书的意见及问题发给出版商：

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

本书有一个 Web 页面，上面列出了勘误表、一些实例以及所有的附加信息。可以通过以下链接来访问这个页面。

http://www.oreilly.com/catalog/httptdg/

为本书提意见或者询问一些技术性问题，可以向以下地址发送邮件。

bookquestions@oreilly.com

更多与书籍、会议、资源中心以及 O'Reilly 网络有关的问题，都请参见 O'Reilly 的网站。

http://www.oreilly.com

致谢

本书是很多人共同劳动的成果。五位作者要感谢一些人，感谢他们为这本书所作出的巨大贡献。

首先，我们要感谢 O'Reilly 的编辑 Linda Mui。Linda 早在 1996 年就与 David 和 Brian 进行了首次接触，她还提炼了几个概念，并将其融入到今天大家拿到的这本书中。Linda 还帮助我们这帮首次写书、徘徊不定的人协调一致地按计划逐步完成了这本书的写作（尽管我们完成得并不怎么快）。最重要的是，Linda 给了我们一个创作此书的机会。我们要对她表示由衷的感谢。

我们还要感谢以下人士，他们非常聪明博学而且非常友善，为校对、注释并修订本书草稿花费了大量精力。他们是：Tony Bourke、Sean Burke、Mike Chowla、Shernaz Daver、Fred Douglis、Paula Ferguson、Vikas Jha、Yves Lafon、Peter Mattis、Chuck Neerdaels、Luis Tavera、Duane Wessels、Dave Wu 和 Marco Zagha。他们的一些观点和建议大大提升了本书的质量。

本书大部分精美的插图都是由 O'Reilly 的 Rob Romano 制作的。为了能够更加清晰地描述一些微妙的概念，本书使用了大量详尽备至的插图。其中很多插图制作起来都很费劲，而且还经过了大量的修改。如果一幅图相当于一千个字的话，Rob 就相当于为本书增加了数百页的篇幅。

Brian 还要特别感谢所有作者对本项目的付出。为了对 HTTP 作出首次详细而又切实可行的剖析，作者们投入了大量的时间。其间虽然出现了婚礼、孩子出世、刻不容缓的工作项目、创业公司起步以及就读研究生院等诸多问题，但作者们的共同努力使这个项目得以圆满完成。我们相信，每个人的努力付出都是值得的，而且最重要的是，这项工作为大家提供了一项很有价值的服务。Brian 还要感谢 Inktomi 的员工们，感谢他们的热情和支持，以及他们对 HTTP 在实际应用程序中应用状况的深

刻洞察力。同时，还要感谢 Cajun-shop.com 允许我们使用他们的站点来展示书中的一些范例。

David 要感谢他的家人，尤其是母亲和祖父长期以来不懈的支持。他要感谢那些在写书这几年中忍受他古怪作息习惯的家人们。他要对 Slurp、Orctomi 和 Norma 所做的一切表示感谢，还要感谢合作者们的辛勤工作。最后，他要感谢 Brian 说服自己再次冒险。

Marjorie 要感谢她丈夫 Alan Liu 的技术洞察力，以及他对家庭的支持和理解。Marjorie 还要感谢合作者们丰富且深刻的灵感和洞察力。在编写过程中能够与他们共同工作，她感到非常开心。

Sailu 要感谢 David 和 Brian 为他提供机会参与编写本书，感谢 Chuck Neerdaels 将他引入了 HTTP 的世界。

Anshu 要感谢他的妻子 Rashi 和他的父母。尽管本书的编写旷日持久，但家人依旧对他有着足够的耐心，不断地支持并鼓励他。

最后，作者们要集体感谢各位著名和无名的因特网先驱们，他们在过去 40 年中所做的研究、开发和传播工作对我们的科学界、社会及经济团体作出了巨大的贡献。没有他们的工作，就没有本书所要讨论的话题。

目录

第一部分　HTTP：Web 的基础

第 1 章　HTTP 概述……3

1.1　HTTP——因特网的多媒体信使……4
1.2　Web 客户端和服务器……4
1.3　资源……5
1.3.1　媒体类型……6
1.3.2　URI……7
1.3.3　URL……7
1.3.4　URN……8
1.4　事务……9
1.4.1　方法……9
1.4.2　状态码……10
1.4.3　Web 页面中可以包含多个对象……10
1.5　报文……11
1.6　连接……13
1.6.1　TCP/IP……13
1.6.2　连接、IP 地址及端口号……14
1.6.3　一个使用 Telnet 的实例……16
1.7　协议版本……18
1.8　Web 的结构组件……19
1.8.1　代理……19
1.8.2　缓存……20
1.8.3　网关……20
1.8.4　隧道……21

1.8.5 Agent 代理……21
1.9 起始部分的结束语……22
1.10 更多信息……22
1.10.1 HTTP 协议信息……22
1.10.2 历史透视……23
1.10.3 其他万维网信息……23

第 2 章 URL 与资源……25

2.1 浏览因特网资源……26
2.2 URL 的语法……28
2.2.1 方案——使用什么协议……29
2.2.2 主机与端口……30
2.2.3 用户名和密码……30
2.2.4 路径……31
2.2.5 参数……31
2.2.6 查询字符串……32
2.2.7 片段……33
2.3 URL 快捷方式……34
2.3.1 相对 URL……34
2.3.2 自动扩展 URL……37
2.4 各种令人头疼的字符……38
2.4.1 URL 字符集……38
2.4.2 编码机制……38
2.4.3 字符限制……39
2.4.4 另外一点说明……40
2.5 方案的世界……40
2.6 未来展望……42
2.7 更多信息……44

第 3 章 HTTP 报文……45

3.1 报文流……46
3.1.1 报文流入源端服务器……46
3.1.2 报文向下游流动……47
3.2 报文的组成部分……47
3.2.1 报文的语法……48
3.2.2 起始行……50
3.2.3 首部……53
3.2.4 实体的主体部分……55
3.2.5 版本 0.9 的报文……55
3.3 方法……56
3.3.1 安全方法……56

3.3.2 GET……56
3.3.3 HEAD……57
3.3.4 PUT……57
3.3.5 POST……58
3.3.6 TRACE……58
3.3.7 OPTIONS……60
3.3.8 DELETE……60
3.3.9 扩展方法……61
3.4 状态码……62
3.4.1 100 ~ 199——信息性状态码……62
3.4.2 200 ~ 299——成功状态码……63
3.4.3 300 ~ 399——重定向状态码……64
3.4.4 400 ~ 499——客户端错误状态码……68
3.4.5 500 ~ 599——服务器错误状态码……69
3.5 首部……70
3.5.1 通用首部……71
3.5.2 请求首部……72
3.5.3 响应首部……74
3.5.4 实体首部……75
3.6 更多信息……77
第 4 章 连接管理……79
4.1 TCP 连接……80
4.1.1 TCP 的可靠数据管道……80
4.1.2 TCP 流是分段的、由 IP 分组传送……81
4.1.3 保持 TCP 连接持续不断地运行……82
4.1.4 用 TCP 套接字编程……84
4.2 对 TCP 性能的考虑……85
4.2.1 HTTP 事务的时延……86
4.2.2 性能聚焦区域……87
4.2.3 TCP 连接的握手时延……87
4.2.4 延迟确认……88
4.2.5 TCP 慢启动……89
4.2.6 Nagle 算法与 TCP_NODELAY……89
4.2.7 TIME_WAIT 累积与端口耗尽……90
4.3 HTTP 连接的处理……91
4.3.1 常被误解的 Connection 首部……91
4.3.2 串行事务处理时延……92
4.4 并行连接……94
4.4.1 并行连接可能会提高页面的加载速度……94
4.4.2 并行连接不一定更快……95

4.4.3 并行连接可能让人“感觉”更快一些 ····· 95
4.5 持久连接 ····· 96
4.5.1 持久以及并行连接 ····· 96
4.5.2 HTTP/1.0+ keep-alive 连接 ····· 97
4.5.3 Keep-Alive 操作 ····· 98
4.5.4 Keep-Alive 选项 ····· 98
4.5.5 Keep-Alive 连接的限制和规则 ····· 99
4.5.6 Keep-Alive 和哑代理 ····· 100
4.5.7 插入 Proxy-Connection ····· 102
4.5.8 HTTP/1.1 持久连接 ····· 104
4.5.9 持久连接的限制和规则 ····· 104
4.6 管道化连接 ····· 105
4.7 关闭连接的奥秘 ····· 106
4.7.1“任意”解除连接 ····· 106
4.7.2 Content-Length 及截尾操作 ····· 107
4.7.3 连接关闭容限、重试以及幂等性 ····· 107
4.7.4 正常关闭连接 ····· 108
4.8 更多信息 ····· 110
4.8.1 HTTP 连接 ····· 110
4.8.2 HTTP 性能问题 ····· 110
4.8.3 TCP/IP ····· 111

第二部分 HTTP 结构

第 5 章 Web 服务器 ····· 115

5.1 各种形状和尺寸的 Web 服务器 ····· 116
5.1.1 Web 服务器的实现 ····· 116
5.1.2 通用软件 Web 服务器 ····· 117
5.1.3 Web 服务器设备 ····· 117
5.1.4 嵌入式 Web 服务器 ····· 118
5.2 最小的 Perl Web 服务器 ····· 118
5.3 实际的 Web 服务器会做些什么 ····· 120
5.4 第一步——接受客户端连接 ····· 121
5.4.1 处理新连接 ····· 121
5.4.2 客户端主机名识别 ····· 122
5.4.3 通过 ident 确定客户端用户 ····· 122
5.5 第二步——接收请求报文 ····· 123
5.5.1 报文的内部表示法 ····· 124
5.5.2 连接的输入 / 输出处理结构 ····· 125
5.6 第三步——处理请求 ····· 126

5.7 第四步——对资源的映射及访问……126
5.7.1 docroot……127
5.7.2 目录列表……129
5.7.3 动态内容资源的映射……130
5.7.4 服务器端包含项……131
5.7.5 访问控制……131
5.8 第五步——构建响应……131
5.8.1 响应实体……131
5.8.2 MIME 类型……132
5.8.3 重定向……133
5.9 第六步——发送响应……134
5.10 第七步——记录日志……134
5.11 更多信息……134

第 6 章 代理……135

6.1 Web 的中间实体……136
6.1.1 私有和共享代理……136
6.1.2 代理与网关的对比……137
6.2 为什么使用代理……138
6.3 代理会去往何处……143
6.3.1 代理服务器的部署……144
6.3.2 代理的层次结构……144
6.3.3 代理是如何获取流量的……147
6.4 客户端的代理设置……148
6.4.1 客户端的代理配置：手工配置……149
6.4.2 客户端代理配置：PAC 文件……149
6.4.3 客户端代理配置：WPAD……150
6.5 与代理请求有关的一些棘手问题……151
6.5.1 代理 URI 与服务器 URI 的不同……151
6.5.2 与虚拟主机一样的问题……152
6.5.3 拦截代理会收到部分 URI……153
6.5.4 代理既可以处理代理请求，也可以处理服务器请求……154
6.5.5 转发过程中对 URI 的修改……154
6.5.6 URI 的客户端自动扩展和主机名解析……155
6.5.7 没有代理时 URI 的解析……155
6.5.8 有显式代理时 URI 的解析……156
6.5.9 有拦截代理时 URI 的解析……157
6.6 追踪报文……158
6.6.1 Via 首部……158
6.6.2 TRACE 方法……162
6.7 代理认证……164

6.8 代理的互操作性……165
6.8.1 处理代理不支持的首部和方法……166
6.8.2 OPTIONS：发现对可选特性的支持……166
6.8.3 Allow 首部……167
6.9 更多信息……167

第 7 章 缓存……169

7.1 冗余的数据传输……170
7.2 带宽瓶颈……170
7.3 瞬间拥塞……171
7.4 距离时延……172
7.5 命中和未命中的……173
7.5.1 再验证……173
7.5.2 命中率……175
7.5.3 字节命中率……176
7.5.4 区分命中和未命中的情况……176
7.6 缓存的拓扑结构……177
7.6.1 私有缓存……177
7.6.2 公有代理缓存……177
7.6.3 代理缓存的层次结构……179
7.6.4 网状缓存、内容路由以及对等缓存……180
7.7 缓存的处理步骤……181
7.7.1 第一步——接收……181
7.7.2 第二步——解析……182
7.7.3 第三步——查找……182
7.7.4 第四步——新鲜度检测……182
7.7.5 第五步——创建响应……182
7.7.6 第六步——发送……183
7.7.7 第七步——日志……183
7.7.8 缓存处理流程图……183
7.8 保持副本的新鲜……183
7.8.1 文档过期……184
7.8.2 过期日期和使用期……185
7.8.3 服务器再验证……185
7.8.4 用条件方法进行再验证……186
7.8.5 If-Modified-Since:Date 再验证……187
7.8.6 If-None-Match：实体标签再验证……189
7.8.7 强弱验证器……190
7.8.8 什么时候应该使用实体标签和最近修改日期……190
7.9 控制缓存的能力……191
7.9.1 no-Store 与 no-Cache 响应首部……191

7.9.2 max-age 响应首部 ····192
7.9.3 Expires 响应首部 ····192
7.9.4 must-revalidate 响应首部 ····192
7.9.5 试探性过期 ····193
7.9.6 客户端的新鲜度限制 ····194
7.9.7 注意事项 ····194
7.10 设置缓存控制 ····195
7.10.1 控制 Apache 的 HTTP 首部 ····195
7.10.2 通过 HTTP-EQUIV 控制 HTML 缓存 ····196
7.11 详细算法 ····197
7.11.1 使用期和新鲜生存期 ····198
7.11.2 使用期的计算 ····198
7.11.3 完整的使用期计算算法 ····201
7.11.4 新鲜生存期计算 ····202
7.11.5 完整的服务器——新鲜度算法 ····202
7.12 缓存和广告 ····204
7.12.1 发布广告者的两难处境 ····204
7.12.2 发布者的响应 ····204
7.12.3 日志迁移 ····205
7.12.4 命中计数和使用限制 ····205
7.13 更多信息 ····205

第 8 章 集成点：网关、隧道及中继 ····207

8.1 网关 ····208
8.2 协议网关 ····210
8.2.1 HTTP/*：服务器端 Web 网关 ····211
8.2.2 HTTP/HTTPS：服务器端安全网关 ····212
8.2.3 HTTPS/HTTP 客户端安全加速器网关 ····212
8.3 资源网关 ····213
8.3.1 CGI ····215
8.3.2 服务器扩展 API ····215
8.4 应用程序接口和 Web 服务 ····216
8.5 隧道 ····217
8.5.1 用 CONNECT 建立 HTTP 隧道 ····217
8.5.2 数据隧道、定时及连接管理 ····219
8.5.3 SSL 隧道 ····219
8.5.4 SSL 隧道与 HTTP/HTTPS 网关的对比 ····220
8.5.5 隧道认证 ····221
8.5.6 隧道的安全性考虑 ····221
8.6 中继 ····222
8.7 更多信息 ····224

第 9 章　Web 机器人……225
9.1　爬虫及爬行方式……226
9.1.1　从哪儿开始：根集……226
9.1.2　链接的提取以及相对链接的标准化……227
9.1.3　避免环路的出现……228
9.1.4　循环与复制……228
9.1.5　面包屑留下的痕迹……229
9.1.6　别名与机器人环路……230
9.1.7　规范化 URL……230
9.1.8　文件系统连接环路……231
9.1.9　动态虚拟 Web 空间……232
9.1.10　避免循环和重复……233
9.2　机器人的 HTTP……236
9.2.1　识别请求首部……236
9.2.2　虚拟主机……236
9.2.3　条件请求……237
9.2.4　对响应的处理……238
9.2.5　User-Agent 导向……239
9.3　行为不当的机器人……239
9.4　拒绝机器人访问……240
9.4.1　拒绝机器人访问标准……241
9.4.2　Web 站点和 robots.txt 文件……242
9.4.3　robots.txt 文件的格式……243
9.4.4　其他有关 robots.txt 的知识……246
9.4.5　缓存和 robots.txt 的过期……246
9.4.6　拒绝机器人访问的 Perl 代码……246
9.4.7　HTML 的 robot-control 元标签……249
9.5　机器人的规范……251
9.6　搜索引擎……254
9.6.1　大格局……255
9.6.2　现代搜索引擎结构……255
9.6.3　全文索引……255
9.6.4　发布查询请求……257
9.6.5　对结果进行排序，并提供查询结果……258
9.6.6　欺诈……258
9.7　更多信息……258
第 10 章　HTTP-NG……261
10.1　HTTP 发展中存在的问题……262
10.2　HTTP-NG 的活动……263

10.3 模块化及功能增强 ····263
10.4 分布式对象 ····264
10.5 第一层——报文传输 ····264
10.6 第二层——远程调用 ····265
10.7 第三层——Web 应用 ····265
10.8 WebMUX ····265
10.9 二进制连接协议 ····266
10.10 当前的状态 ····267
10.11 更多信息 ····267

第三部分 识别、认证与安全

第 11 章 客户端识别与 cookie 机制 ····271

11.1 个性化接触 ····272
11.2 HTTP 首部 ····273
11.3 客户端 IP 地址 ····274
11.4 用户登录 ····275
11.5 胖 URL ····277
11.6 cookie ····278
11.6.1 cookie 的类型 ····278
11.6.2 cookie 是如何工作的 ····279
11.6.3 cookie 罐：客户端的状态 ····280
11.6.4 不同站点使用不同的 cookie ····282
11.6.5 cookie 成分 ····283
11.6.6 cookies 版本 0（Netscape）····284
11.6.7 cookies 版本 1（RFC 2965）····285
11.6.8 cookie 与会话跟踪 ····288
11.6.9 cookie 与缓存 ····290
11.6.10 cookie、安全性和隐私 ····291
11.7 更多信息 ····292

第 12 章 基本认证机制 ····293

12.1 认证 ····294
12.1.1 HTTP 的质询 / 响应认证框架 ····294
12.1.2 认证协议与首部 ····295
12.1.3 安全域 ····296
12.2 基本认证 ····297
12.2.1 基本认证实例 ····298
12.2.2 Base-64 用户名 / 密码编码 ····298
12.2.3 代理认证 ····299

12.3 基本认证的安全缺陷……300
12.4 更多信息……301

第 13 章 摘要认证……303

13.1 摘要认证的改进……304
13.1.1 用摘要保护密码……304
13.1.2 单向摘要……306
13.1.3 用随机数防止重放攻击……307
13.1.4 摘要认证的握手机制……307
13.2 摘要的计算……308
13.2.1 摘要算法的输入数据……308
13.2.2 算法 H(*d*) 和 KD(*s*,*d*)……310
13.2.3 与安全性相关的数据（A1）……310
13.2.4 与报文有关的数据（A2）……310
13.2.5 摘要算法总述……311
13.2.6 摘要认证会话……312
13.2.7 预授权……312
13.2.8 随机数的选择……315
13.2.9 对称认证……315
13.3 增强保护质量……316
13.3.1 报文完整性保护……316
13.3.2 摘要认证首部……317
13.4 应该考虑的实际问题……317
13.4.1 多重质询……318
13.4.2 差错处理……318
13.4.3 保护空间……318
13.4.4 重写 URI……319
13.4.5 缓存……319
13.5 安全性考虑……320
13.5.1 首部篡改……320
13.5.2 重放攻击……320
13.5.3 多重认证机制……320
13.5.4 词典攻击……321
13.5.5 恶意代理攻击和中间人攻击……321
13.5.6 选择明文攻击……321
13.5.7 存储密码……322
13.6 更多信息……322

第 14 章 安全 HTTP……323

14.1 保护 HTTP 的安全……324
14.2 数字加密……326

14.2.1 密码编制的机制与技巧……326
14.2.2 密码……327
14.2.3 密码机……328
14.2.4 使用了密钥的密码……328
14.2.5 数字密码……328
14.3 对称密钥加密技术……330
14.3.1 密钥长度与枚举攻击……330
14.3.2 建立共享密钥……332
14.4 公开密钥加密技术……332
14.4.1 RSA……333
14.4.2 混合加密系统和会话密钥……334
14.5 数字签名……334
14.6 数字证书……336
14.6.1 证书的主要内容……336
14.6.2 X.509 v3 证书……337
14.6.3 用证书对服务器进行认证……338
14.7 HTTPS——细节介绍……339
14.7.1 HTTPS 概述……339
14.7.2 HTTPS 方案……340
14.7.3 建立安全传输……341
14.7.4 SSL 握手……341
14.7.5 服务器证书……343
14.7.6 站点证书的有效性……344
14.7.7 虚拟主机与证书……345
14.8 HTTPS 客户端实例……345
14.8.1 OpenSSL……346
14.8.2 简单的 HTTPS 客户端……347
14.8.3 执行 OpenSSL 客户端……350
14.9 通过代理以隧道形式传输安全流量……351
14.10 更多信息……353
14.10.1 HTTP 安全性……353
14.10.2 SSL 与 TLS……353
14.10.3 公开密钥基础设施……354
14.10.4 数字密码……354

第四部分 实体、编码和国际化

第 15 章 实体和编码……357

15.1 报文是箱子，实体是货物……359
15.2 Content-Length：实体的大小……361

15.2.1 检测截尾……361
15.2.2 错误的 Content-Length……362
15.2.3 Content-Length 与持久连接……362
15.2.4 内容编码……362
15.2.5 确定实体主体长度的规则……362
15.3 实体摘要……364
15.4 媒体类型和字符集……364
15.4.1 文本的字符编码……365
15.4.2 多部分媒体类型……365
15.4.3 多部分表格提交……366
15.4.4 多部分范围响应……367
15.5 内容编码……368
15.5.1 内容编码过程……368
15.5.2 内容编码类型……369
15.5.3 Accept-Encoding 首部……369
15.6 传输编码和分块编码……371
15.6.1 可靠传输……371
15.6.2 Transfer-Encoding 首部……372
15.6.3 分块编码……373
15.6.4 内容编码与传输编码的结合……375
15.6.5 传输编码的规则……375
15.7 随时间变化的实例……375
15.8 验证码和新鲜度……376
15.8.1 新鲜度……377
15.8.2 有条件的请求与验证码……378
15.9 范围请求……380
15.10 差异编码……382
15.11 更多信息……385
第 16 章 国际化……387
16.1 HTTP 对国际性内容的支持……388
16.2 字符集与 HTTP……389
16.2.1 字符集是把字符转换为二进制码的编码……389
16.2.2 字符集和编码如何工作……390
16.2.3 字符集不对，字符就不对……391
16.2.4 标准化的 MIME charset 值……391
16.2.5 Content-Type 首部和 Charset 首部以及 META 标志……393
16.2.6 Accept-Charset 首部……393
16.3 多语言字符编码入门……394
16.3.1 字符集术语……394
16.3.2 字符集的命名很糟糕……395

16.3.3 字符……396
16.3.4 字形、连笔以及表示形式……396
16.3.5 编码后的字符集……397
16.3.6 字符编码方案……399
16.4 语言标记与 HTTP……402
16.4.1 Content-Language 首部……402
16.4.2 Accept-Language 首部……403
16.4.3 语言标记的类型……404
16.4.4 子标记……404
16.4.5 大小写……405
16.4.6 IANA 语言标记注册……405
16.4.7 第一个子标记——名字空间……405
16.4.8 第二个子标记——名字空间……406
16.4.9 其余子标记——名字空间……407
16.4.10 配置和语言有关的首选项……407
16.4.11 语言标记参考表……407
16.5 国际化的 URI……408
16.5.1 全球性的可转抄能力与有意义的字符的较量……408
16.5.2 URI 字符集合……408
16.5.3 转义和反转义……409
16.5.4 转义国际化字符……409
16.5.5 URI 中的模态切换……410
16.6 其他需要考虑的地方……410
16.6.1 首部和不合规范的数据……410
16.6.2 日期……411
16.6.3 域名……411
16.7 更多信息……411
16.7.1 附录……411
16.7.2 互联网的国际化……411
16.7.3 国际标准……412
第 17 章 内容协商与转码……413
17.1 内容协商技术……414
17.2 客户端驱动的协商……415
17.3 服务器驱动的协商……415
17.3.1 内容协商首部集……416
17.3.2 内容协商首部中的质量值……417
17.3.3 随其他首部集而变化……417
17.3.4 Apache 中的内容协商……417
17.3.5 服务器端扩展……418
17.4 透明协商……419

17.4.1 进行缓存与备用候选……419
17.4.2 Vary 首部……420
17.5 转码……422
17.5.1 格式转换……422
17.5.2 信息综合……423
17.5.3 内容注入……423
17.5.4 转码与静态预生成的对比……423
17.6 下一步计划……424
17.7 更多信息……424

第五部分 内容发布与分发

第 18 章 Web 主机托管……429

18.1 主机托管服务……430
18.2 虚拟主机托管……431
18.2.1 虚拟服务器请求缺乏主机信息……432
18.2.2 设法让虚拟主机托管正常工作……433
18.2.3 HTTP/1.1 的 Host 首部……437
18.3 使网站更可靠……438
18.3.1 镜像的服务器集群……438
18.3.2 内容分发网络……440
18.3.3 CDN 中的反向代理缓存……440
18.3.4 CDN 中的代理缓存……440
18.4 让网站更快……441
18.5 更多信息……441

第 19 章 发布系统……443

19.1 FrontPage 为支持发布而做的服务器扩展……444
19.1.1 FrontPage 服务器扩展……444
19.1.2 FrontPage 术语表……445
19.1.3 FrontPage 的 RPC 协议……445
19.1.4 FrontPage 的安全模型……448
19.2 WebDAV 与协作写作……449
19.2.1 WebDAV 的方法……449
19.2.2 WebDAV 与 XML……450
19.2.3 WebDAV 首部集……451
19.2.4 WebDAV 的锁定与防止覆写……452
19.2.5 LOCK 方法……453
19.2.6 UNLOCK 方法……456
19.2.7 属性和元数据……456

19.2.8 PROPFIND 方法……457
19.2.9 PROPPATCH 方法……459
19.2.10 集合与名字空间管理……460
19.2.11 MKCOL 方法……460
19.2.12 DELETE 方法……461
19.2.13 COPY 与 MOVE 方法……462
19.2.14 增强的 HTTP/1.1 方法……465
19.2.15 WebDAV 中的版本管理……466
19.2.16 WebDAV 的未来发展……466
19.3 更多信息……467

第 20 章 重定向与负载均衡……469

20.1 为什么要重定向……470
20.2 重定向到何地……471
20.3 重定向协议概览……471
20.4 通用的重定向方法……474
20.4.1 HTTP 重定向……474
20.4.2 DNS 重定向……475
20.4.3 任播寻址……480
20.4.4 IP MAC 转发……481
20.4.5 IP 地址转发……482
20.4.6 网元控制协议……484
20.5 代理的重定向方法……485
20.5.1 显式浏览器配置……485
20.5.2 代理自动配置……485
20.5.3 Web 代理自动发现协议……487
20.6 缓存重定向方法……492
20.7 因特网缓存协议……496
20.8 缓存阵列路由协议……497
20.9 超文本缓存协议……500
20.9.1 HTCP 认证……502
20.9.2 设置缓存策略……503
20.10 更多信息……504

第 21 章 日志记录与使用情况跟踪……505

21.1 记录内容……506
21.2 日志格式……507
21.2.1 常见日志格式……507
21.2.2 组合日志格式……508
21.2.3 网景扩展日志格式……509
21.2.4 网景扩展 2 日志格式……510

21.2.5 Squid 代理日志格式……512
21.3 命中率测量……515
21.3.1 概述……515
21.3.2 Meter 首部……516
21.4 关于隐私的考虑……517
21.5 更多信息……518

第六部分 附 录

附录 A URI 方案……521
附录 B HTTP 状态码……529
附录 C HTTP 首部参考……533
附录 D MIME 类型……557
附录 E Base-64 编码……603
附录 F 摘要认证……607
附录 G 语言标记……615
附录 H MIME 字符集注册表……641
索引……661

第一部分

HTTP：Web的基础

本部分主要概述 HTTP 协议。接下来的 4 章介绍了 Web 的基础构件以及 HTTP 的核心技术。

- 第 1 章简要概述 HTTP。
- 第 2 章详细介绍了 URL 的格式，以及 URL 在因特网上命名的各种类型的资源，并对其向 URN 的发展作了概要介绍。
- 第 3 章详细说明了用来传输 Web 内容的 HTTP 报文。
- 第 4 章讨论了一些通过 HTTP 管理 TCP 连接时常被误解且很少有文档说明的规则和行为。

第1章

HTTP概述

Web 浏览器、服务器和相关的 Web 应用程序都是通过 HTTP 相互通信的。HTTP 是现代全球因特网中使用的公共语言。

本章是对 HTTP 的简要介绍。在本章中可以看到 Web 应用程序是如何使用 HTTP 进行通信的，这样就可以对 HTTP 如何完成其工作有个大概印象。我们将特别介绍以下方面的内容：

- Web 客户端与服务器是如何通信的；
- （表示 Web 内容的）资源来自何方；
- Web 事务是怎样工作的；
- HTTP 通信所使用的报文格式；
- 底层 TCP 网络传输；
- 不同的 HTTP 协议变体；
- 因特网上安装的大量 HTTP 架构组件中的一部分。

我们有很多话题要讨论，就此开始 HTTP 之旅吧。

1.1 HTTP——因特网的多媒体信使

每天，都有数以亿万计的 JPEG 图片、HTML 页面、文本文件、MPEG 电影、WAV 音频文件、Java 小程序和其他资源在因特网上游弋。HTTP 可以从遍布全世界的 Web 服务器上将这些信息块迅速、便捷、可靠地搬移到人们桌面上的 Web 浏览器上去。

HTTP 使用的是可靠的数据传输协议，因此即使数据来自地球的另一端，它也能够确保数据在传输的过程中不会被损坏或产生混乱。这样，用户在访问信息时就不用担心其完整性了，因此对用户来说，这是件好事。而对因特网应用程序开发人员来说也同样如此，因为这样就无需担心 HTTP 通信会在传输过程中被破坏、复制或产生畸变了。开发人员可以专注于应用程序特有细节的编写，而不用考虑因特网中存在的一些缺陷和问题。

1
~
3

下面，就让我们来近距离地观察一下 HTTP 是如何传输 Web 流量的。

1.2 Web客户端和服务器

Web 内容都是存储在 Web 服务器上的。Web 服务器所使用的是 HTTP 协议，因此经常会被称为 HTTP 服务器。这些 HTTP 服务器存储了因特网中的数据，如果

HTTP 客户端发出请求的话，它们会提供数据。客户端向服务器发送 HTTP 请求，服务器会在 HTTP 响应中回送所请求的数据，如图 1-1 所示。HTTP 客户端和 HTTP 服务器共同构成了万维网的基本组件。

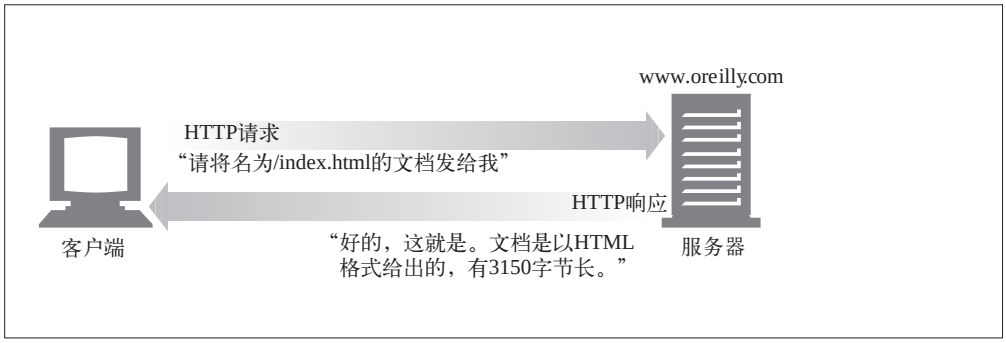


图 1-1　Web 客户端和服务器

可能你每天都在使用 HTTP 客户端。最常见的客户端就是 Web 浏览器，比如微软的 Internet Explorer 或网景的 Navigator。Web 浏览器向服务器请求 HTTP 对象，并将这些对象显示在你的屏幕上。

浏览一个页面时（比如 http://www.oreilly.com/index.html），浏览器会向服务器 www.oreilly.com 发送一条 HTTP 请求（参见图 1-1）。服务器会去寻找所期望的对象（在这个例子中就是 /index.html），如果成功，就将对象、对象类型、对象长度以及其他一些信息放在 HTTP 响应中发送给客户端。

1.3　资源

Web 服务器是 Web 资源（Web resource）的宿主。Web 资源是 Web 内容的源头。最简单的 Web 资源就是 Web 服务器文件系统中的静态文件。这些文件可以包含任意内容：文本文件、HTML 文件、微软的 Word 文件、Adobe 的 Acrobat 文件、JPEG 图片文件、AVI 电影文件，或所有其他你能够想到的格式。

但资源不一定非得是静态文件。资源还可以是根据需要生成内容的软件程序。这些动态内容资源可以根据你的身份、所请求的信息或每天的不同时段来产生内容。它们可以为你显示照相机中活生生的照片，也可以帮你进行股票交易，搜索房产数据库，或者从在线商店中购买礼物（参见图 1-2）。

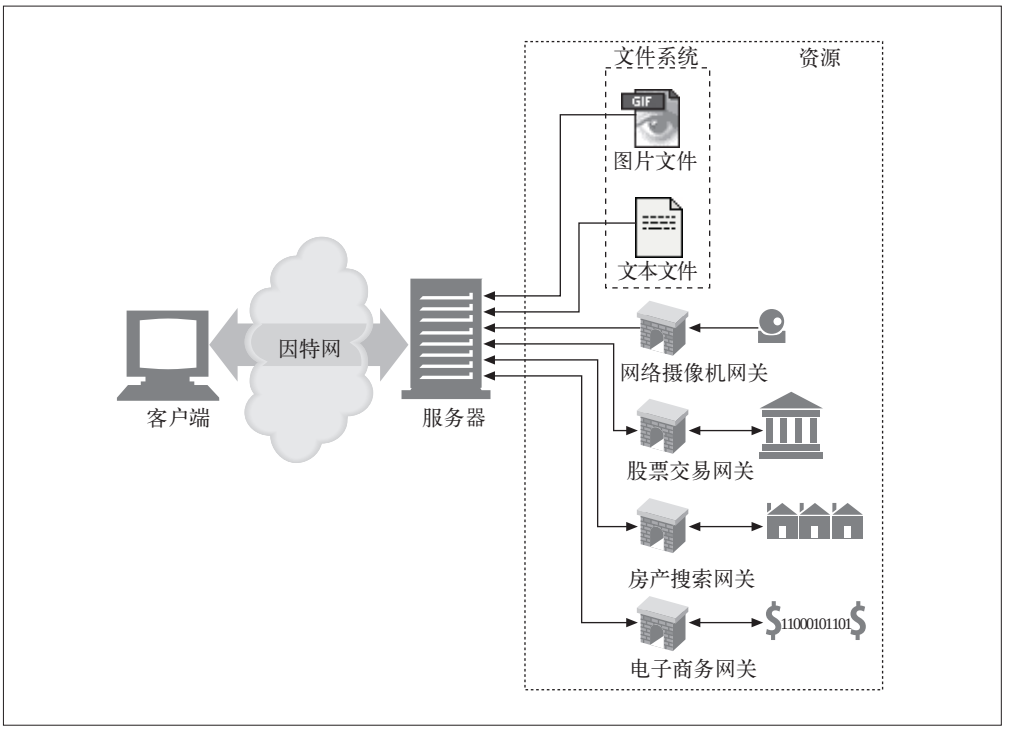


图 1-2　所有能够提供 Web 内容的东西都是 Web 资源

总之，所有类型的内容来源都是资源。包含公司销售预测电子表格的文件是一种资源。扫描本地公共图书馆书架的 Web 网关是一种资源。因特网搜索引擎也是一种资源。

1.3.1　媒体类型

因特网上有数千种不同的数据类型，HTTP 仔细地给每种要通过 Web 传输的对象都打上了名为 MIME 类型（MIME type）的数据格式标签。最初设计 MIME（Multipurpose Internet Mail Extension，多用途因特网邮件扩展）是为了解决在不同的电子邮件系统之间搬移报文时存在的问题。MIME 在电子邮件系统中工作得非常好，因此 HTTP 也采纳了它，用它来描述并标记多媒体内容。

Web 服务器会为所有 HTTP 对象数据附加一个 MIME 类型（参见图 1-3）。当 Web 浏览器从服务器中取回一个对象时，会去查看相关的 MIME 类型，看看它是否知道应该如何处理这个对象。大多数浏览器都可以处理数百种常见的对象类型：显示图片文件、解析并格式化 HTML 文件、通过计算机声卡播放音频文件，或者运行外部插件软件来处理特殊格式的数据。

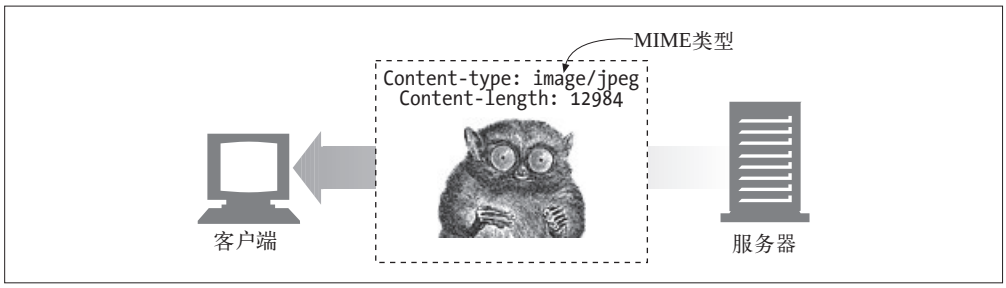


图 1-3　与数据内容一同回送的 MIME 类型

MIME 类型是一种文本标记，表示一种主要的对象类型和一个特定的子类型，中间由一条斜杠来分隔。

- HTML 格式的文本文档由 `text/html` 类型来标记。
- 普通的 ASCII 文本文档由 `text/plain` 类型来标记。
- JPEG 格式的图片为 `image/jpeg` 类型。
- GIF 格式的图片为 `image/gif` 类型。
- Apple 的 QuickTime 电影为 `video/quicktime` 类型。
- 微软的 PowerPoint 演示文件为 `application/vnd.ms-powerpoint` 类型。

常见的 MIME 类型有数百个，实验性或用途有限的 MIME 类型则更多。附录 D 提供了一个非常完整的 MIME 类型列表。

1.3.2　URI

每个 Web 服务器资源都有一个名字，这样客户端就可以说明它们感兴趣的资源是什么了。服务器资源名被称为*统一资源标识符*（Uniform Resource Identifier，URI）。URI 就像因特网上的邮政地址一样，在世界范围内唯一标识并定位信息资源。

这是 Joe 的五金商店的 Web 服务器上一个图片资源的 URI：

http://www.joes-hardware.com/specials/saw-blade.gif

图 1-4 显示了 URI 是怎样指示 HTTP 协议去访问 Joe 商店服务器上的图片资源的。给定了 URI，HTTP 就可以解析出对象。URI 有两种形式，分别称为 URL 和 URN。现在我们分别来看看这些资源标识符类型。

1.3.3　URL

统一资源定位符（URL）是资源标识符最常见的形式。URL 描述了一台特定服务器上某资源的特定位置。它们可以明确说明如何从一个精确、固定的位置获取资源。

图 1-4 显示了 URL 如何精确地说明某资源的位置以及如何去访问它。表 1-1 显示了
6 几个 URL 实例。

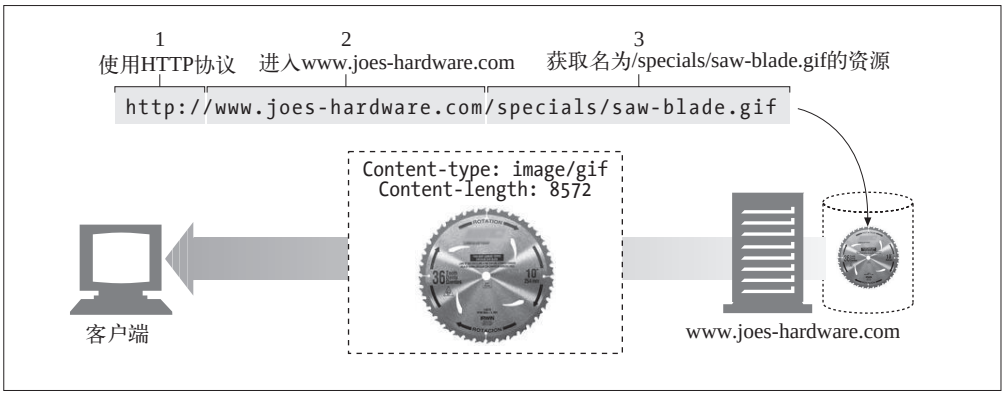


图 1-4　URL 说明了协议、服务器和本地资源

表1-1　URL实例

URL	描　　述
http://www.oreilly.com/index.html	O'Reilly & Associates 公司的主 URL
http://www.yahoo.com/images/logo.gif	Yahoo! 的 Web 站点标志 URL
http://www.joes-hardware.com/inventory-check.cgi?item=12731	一个查看库存条目 #12731 是否有现货的程序的 URL
ftp://joe:tools4u@ftp.joes-hardware.com/locking-pliers.gif	以密码保护的 FTP 作为访问协议的 locking-pliers.gif 图片文件的 URL

大部分 URL 都遵循一种标准格式，这种格式包含三个部分。

- URL 的第一部分被称为*方案*（scheme），说明了访问资源所使用的协议类型。这部分通常就是 HTTP 协议（http://）。
- 第二部分给出了服务器的因特网地址（比如，www.joes-hardware.com）。
- 其余部分指定了 Web 服务器上的某个资源（比如，/specials/saw-blade.gif）。

现在，几乎所有的 URI 都是 URL。

1.3.4　URN

URI 的第二种形式就是*统一资源名*（URN）。URN 是作为特定内容的唯一名称使用的，与目前的资源所在地无关。使用这些与位置无关的 URN，就可以将资源四处搬移。通过 URN，还可以用同一个名字通过多种网络访问协议来访问资源。

比如，不论因特网标准文档 RFC 2141 位于何处（甚至可以将其复制到多个地方），都可以用下列 URN 来命名它：

urn:ietf:rfc:2141

URN 仍然处于试验阶段，还未大范围使用。为了更有效地工作，URN 需要一个支撑架构来解析资源的位置。而此类架构的缺乏也延缓了其被采用的进度。但 URN 确实为未来发展作出了一些令人兴奋的承诺。我们将在第 2 章较为详细地讨论 URN，而本书的其余部分讨论的基本上都是 URL。

除非特殊说明，否则本书的其余部分都会使用约定的术语，并且会不加区别地使用 URI 和 URL。

1.4 事务

我们来更仔细地看看客户端是怎样通过 HTTP 与 Web 服务器及其资源进行事务处理的。一个 HTTP 事务由一条（从客户端发往服务器的）请求命令和一个（从服务器发回客户端的）响应结果组成。这种通信是通过名为 HTTP 报文（HTTP message）的格式化数据块进行的，如图 1-5 所示。

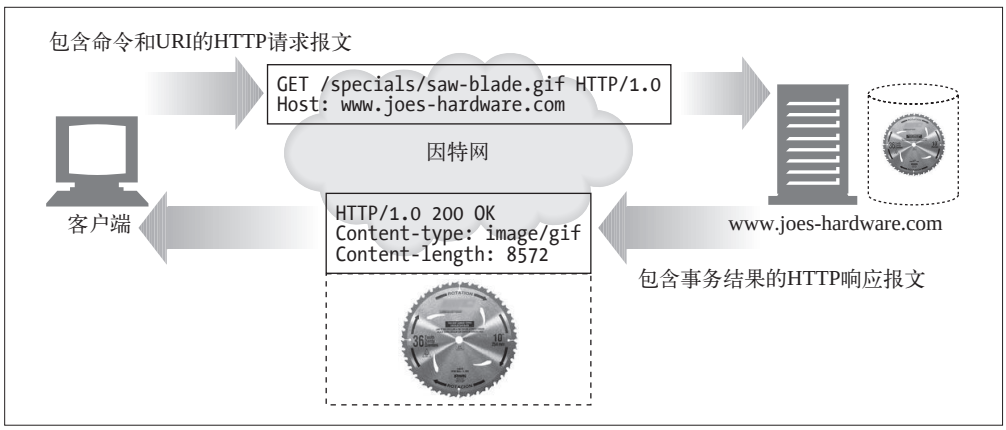


图 1-5 包含请求及响应报文的 HTTP 事务

1.4.1 方法

HTTP 支持几种不同的请求命令，这些命令被称为 HTTP 方法（HTTP method）。每条 HTTP 请求报文都包含一个方法。这个方法会告诉服务器要执行什么动作（获取一个 Web 页面、运行一个网关程序、删除一个文件等）。表 1-2 列出了五种常见的 HTTP 方法。

表1-2　一些常见的HTTP方法

HTTP方法	描　　述
GET	从服务器向客户端发送命名资源
PUT	将来自客户端的数据存储到一个命名的服务器资源中去
DELETE	从服务器中删除命名资源
POST	将客户端数据发送到一个服务器网关应用程序
HEAD	仅发送命名资源响应中的 HTTP 首部

我们会在第 3 章详细讨论 HTTP 方法。

1.4.2　状态码

每条 HTTP 响应报文返回时都会携带一个状态码。状态码是一个三位数字的代码，告知客户端请求是否成功，或者是否需要采取其他动作。表 1-3 显示了几种常见的状态码。

表1-3　一些常见的HTTP状态码

HTTP状态码	描　　述
200	OK。文档正确返回
302	Redirect（重定向）。到其他地方去获取资源
404	Not Found（没找到）。无法找到这个资源

伴随着每个数字状态码，HTTP 还会发送一条解释性的“原因短语”文本（参见图 1-5 中的响应报文）。包含文本短语主要是为了进行描述，所有的处理过程使用的都是数字码。

HTTP 软件处理下列状态码和原因短语的方式是一样的。

```
200 OK
200 Document attached
200 Success
200 All's cool, dude
```

第 3 章详细解释了 HTTP 状态码。

1.4.3　Web页面中可以包含多个对象

应用程序完成一项任务时通常会发布多个 HTTP 事务。比如，Web 浏览器会发布一系列 HTTP 事务来获取并显示一个包含了丰富图片的 Web 页面。浏览器会执行一个事务来获取描述页面布局的 HTML“框架”，然后发布另外的 HTTP 事务来获取每个嵌入式图片、图像面板、Java 小程序等。这些嵌入式资源甚至可能位于不同的服

务器上，如图 1-6 所示。因此，一个“Web 页面”通常并不是单个资源，而是一组资源的集合。

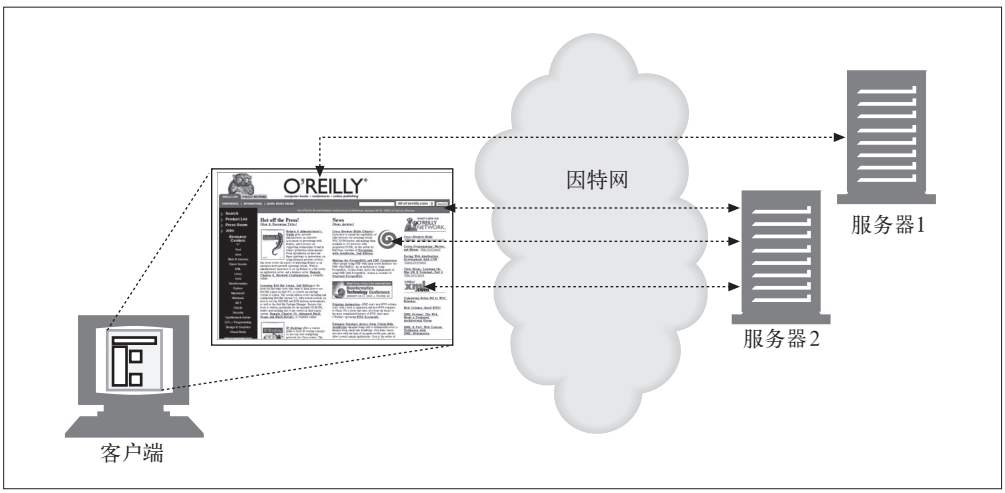


图 1-6　复合 Web 页面要为每个嵌入式资源使用一个单独的 HTTP 事务

1.5　报文

现在我们来快速浏览一下 HTTP 请求和响应报文的结构。第 3 章会深入研究 HTTP 报文。

HTTP 报文是由一行一行的简单字符串组成的。HTTP 报文都是纯文本，不是二进制代码，所以人们可以很方便地对其进行读写[1]。图 1-7 显示了一个简单事务所使用的 HTTP 报文。

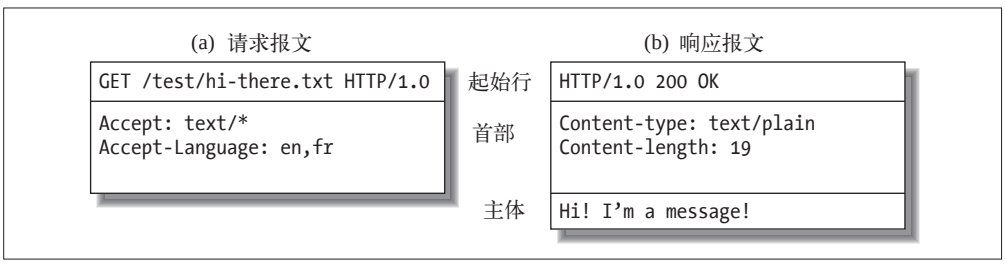


图 1-7　由一行行的简单文本结构组成的 HTTP 报文

注 1：有些程序员会抱怨 HTTP 的语法解析太困难了，这项工作需要很多技巧，而且很容易出错，尤其是在设计高速软件的时候更是如此。二进制格式或更严格的文本格式可能更容易处理，但大多数 HTTP 程序员都很欣赏 HTTP 的可扩展性以及可调试性。

从 Web 客户端发往 Web 服务器的 HTTP 报文称为*请求报文*（request message）。从服务器发往客户端的报文称为*响应报文*（response message），此外没有其他类型的 HTTP 报文。HTTP 请求和响应报文的格式很类似。

HTTP 报文包括以下三个部分。

- *起始行*
  报文的第一行就是起始行，在请求报文中用来说明要做些什么，在响应报文中说明出现了什么情况。

- *首部字段*
  起始行后面有零个或多个首部字段。每个首部字段都包含一个名字和一个值，为了便于解析，两者之间用冒号（:）来分隔。首部以一个空行结束。添加一个首部字段和添加新行一样简单。

- *主体*
  空行之后就是可选的报文主体了，其中包含了所有类型的数据。请求主体中包括了要发送给 Web 服务器的数据；响应主体中装载了要返回给客户端的数据。起始行和首部都是文本形式且都是结构化的，而主体则不同，主体中可以包含任意的二进制数据（比如图片、视频、音轨、软件程序）。当然，主体中也可以包含文本。

简单的报文实例

图 1-8 显示了可能会作为某个简单事务的一部分发送的 HTTP 报文。浏览器请求资源 http://www.joes-hardware.com/tools.html。

在图 1-8 中，浏览器发送了一条 HTTP 请求报文。这条请求的起始行中有一个 GET 命令，且本地资源为 /tools.html。这条请求说明它使用的是 1.0 版的 HTTP 协议。请求报文没有主体，因为从服务器上 GET 一个简单的文档不需要请求数据。

服务器会回送一条 HTTP 响应报文。这条响应中包含了 HTTP 的版本号（HTTP/1.0）、一个成功状态码（200）、一个描述性的原因短语（OK），以及一块响应首部字段，在所有这些内容之后跟着包含了所请求文档的响应主体。`Content-Length` 首部说明了响应主体的长度，`Content-Type` 首部说明了文档的 MIME 类型。

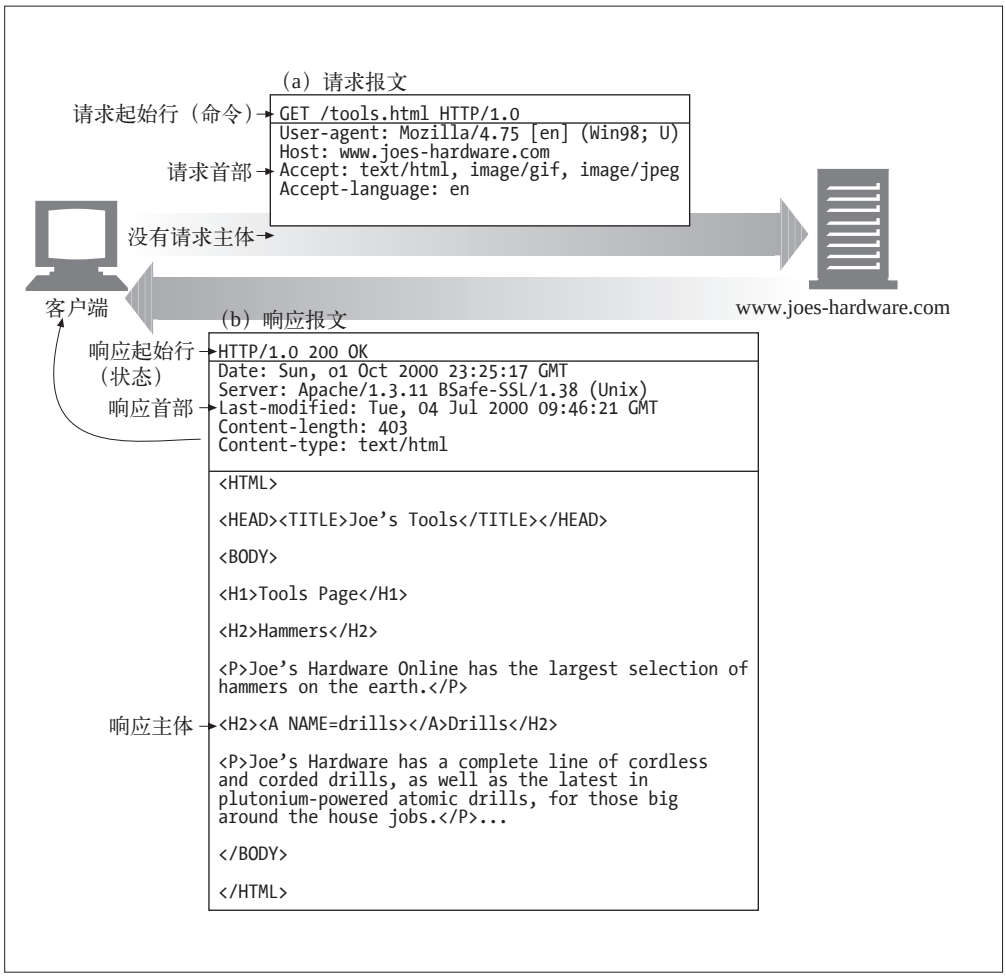


图 1-8　http://www.joes-hardware.com/tools.html 的 GET 事务实例

1.6　连接

概要介绍了 HTTP 报文的构成之后，我们来讨论一下报文是如何通过传输控制协议（Transmission Control Protocol，TCP）连接从一个地方搬移到另一个地方去的。

1.6.1　TCP/IP

HTTP 是个应用层协议。HTTP 无需操心网络通信的具体细节；它把联网的细节都交给了通用、可靠的因特网传输协议 TCP/IP。

11

TCP 提供了：

- 无差错的数据传输；
- 按序传输（数据总是会按照发送的顺序到达）；
- 未分段的数据流（可以在任意时刻以任意尺寸将数据发送出去）。

因特网自身就是基于 TCP/IP 的，TCP/IP 是全世界的计算机和网络设备常用的层次化分组交换网络协议集。TCP/IP 隐藏了各种网络和硬件的特点及弱点，使各种类型的计算机和网络都能够进行可靠地通信。

只要建立了 TCP 连接，客户端和服务器之间的报文交换就不会丢失、不会被破坏，也不会在接收时出现错序了。

用网络术语来说，HTTP 协议位于 TCP 的上层。HTTP 使用 TCP 来传输其报文数据。与之类似，TCP 则位于 IP 的上层（参见图 1-9）。

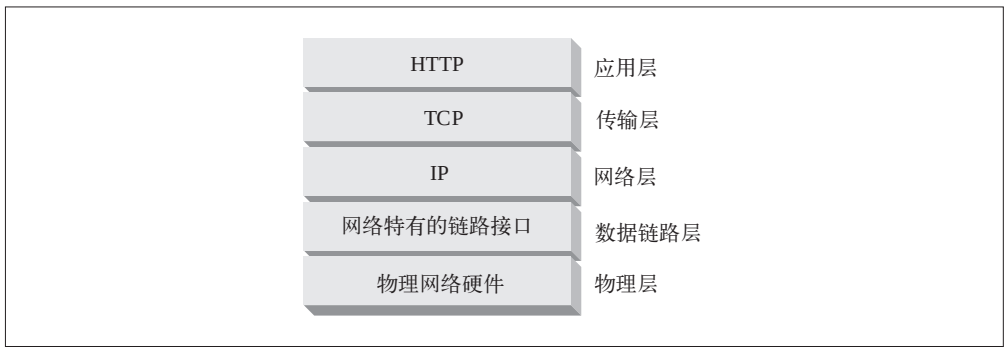


图 1-9　HTTP 网络协议栈

1.6.2　连接、IP地址及端口号

在 HTTP 客户端向服务器发送报文之前，需要用网际协议（Internet Protocol，IP）地址和端口号在客户端和服务器之间建立一条 TCP/IP 连接。

建立一条 TCP 连接的过程与给公司办公室的某个人打电话的过程类似。首先，要拨打公司的电话号码。这样就能进入正确的机构了。其次，拨打要联系的那个人的分机号。

在 TCP 中，你需要知道服务器的 IP 地址，以及与服务器上运行的特定软件相关的 TCP 端口号。

这就行了，但最初怎么获得 HTTP 服务器的 IP 地址和端口号呢？当然是通过 URL 了！我们前面曾提到过，URL 就是资源的地址，所以自然能够为我们提供存储资源的机器的 IP 地址。我们来看几个 URL：

http://207.200.83.29:80/index.html
http://www.netscape.com:80/index.html
http://www.netscape.com/index.html

第一个 URL 使用了机器的 IP 地址，207.200.83.29 以及端口号 80。

第二个 URL 没有使用数字形式的 IP 地址，它使用的是文本形式的域名，或者称为*主机名*（www.netscape.com）。主机名就是 IP 地址比较人性化的别称。可以通过一种称为*域名服务*（Domain Name Service，DNS）的机制方便地将主机名转换为 IP 地址，这样所有问题就都解决了。第 2 章会介绍更多有关 DNS 和 URL 的内容。

最后一个 URL 没有端口号。HTTP 的 URL 中没有端口号时，可以假设默认端口号是 80。

有了 IP 地址和端口号，客户端就可以很方便地通过 TCP/IP 进行通信了。图 1-10 显示了浏览器是怎样通过 HTTP 显示位于远端服务器中的某个简单 HTML 资源的。 13

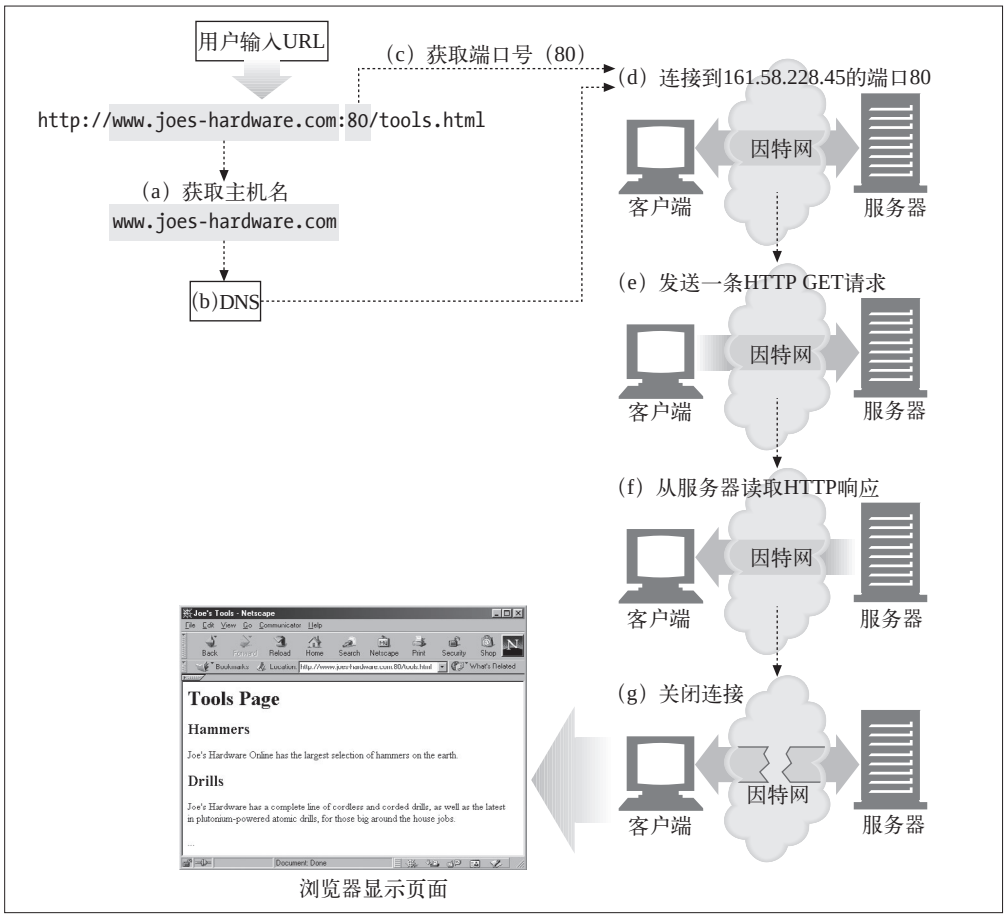


图 1-10　基本的浏览器连接处理 14

步骤如下：

(a) 浏览器从 URL 中解析出服务器的主机名；
(b) 浏览器将服务器的主机名转换成服务器的 IP 地址；
(c) 浏览器将端口号（如果有的话）从 URL 中解析出来；
(d) 浏览器建立一条与 Web 服务器的 TCP 连接；
(e) 浏览器向服务器发送一条 HTTP 请求报文；
(f) 服务器向浏览器回送一条 HTTP 响应报文；
(g) 关闭连接，浏览器显示文档。

1.6.3 一个使用Telnet的实例

由于 HTTP 使用了 TCP/IP 传输协议，而且它是基于文本的，没有使用那些难以理解的二进制格式，因此很容易直接与 Web 服务器进行对话。

Telnet 程序可以将键盘连接到某个目标 TCP 端口，并将此 TCP 端口的输出回送到显示屏上。Telnet 常用于远程终端会话，但它几乎可以连接所有的 TCP 服务器，包括 HTTP 服务器。

可以通过 Telnet 程序直接与 Web 服务器进行对话。通过 Telnet 可以打开一条到某台机器上某个端口的 TCP 连接，然后直接向那个端口输入一些字符。Web 服务器会将 Telnet 程序作为一个 Web 客户端来处理，所有回送给 TCP 连接的数据都会显示在屏幕上。

我们用 Telnet 与一个实际的 Web 服务器进行交互。我们要用 Telnet 获取 URL http://www.joes-hardware.com:80/tools.html 所指向的文档（你可以自己尝试一下这个实例）。

我们来看看会发生什么情况。

- 首先，查找 www.joes-hardware.com 的 IP 地址，打开一条到那台机器端口 80 的 TCP 连接。Telnet 会为我们完成那些“跑腿儿”的工作。
- 一旦打开了 TCP 连接，就要输入 HTTP 请求了。
- 请求结束（由一个空行表示）之后，服务器会在一条 HTTP 响应中将内容回送并关闭连接。

例 1-1 显示了对 http://www.joes-hardware.com:80/tools.html 的 HTTP 请求实例。我们输入的内容用粗体字表示。

例 1-1　一个使用 Telnet 的 HTTP 事务

```
% telnet www.joes-hardware.com 80
Trying 161.58.228.45...
Connected to joes-hardware.com.
Escape character is '^]'.
GET /tools.html HTTP/1.1
Host: www.joes-hardware.com

HTTP/1.1 200 OK
Date: Sun, 01 Oct 2000 23:25:17 GMT
Server: Apache/1.3.11 BSafe-SSL/1.38 (Unix) FrontPage/4.0.4.3
Last-Modified: Tue, 04 Jul 2000 09:46:21 GMT
ETag: "373979-193-3961b26d"
Accept-Ranges: bytes
Content-Length: 403
Connection: close
Content-Type: text/html

<HTML>
<HEAD><TITLE>Joe's Tools</TITLE></HEAD>
<BODY>
<H1>Tools Page</H1>
<H2>Hammers</H2>
<P>Joe's Hardware Online has the largest selection of hammers on the
earth.</P>
<H2><A NAME=drills></A>Drills</H2>
<P>Joe's Hardware has a complete line of cordless and corded drills,
as well as the latest
in plutonium-powered atomic drills, for those big around the house
jobs.</P> ...
</BODY>
</HTML>
Connection closed by foreign host.
```


Telnet 会查找主机名并打开一条连接，连接到在 www.joes-hardware.com 的端口 80 上监听的 Web 服务器。这条命令之后的三行内容是 Telnet 的输出，告诉我们它已经建立了连接。

然后我们输入最基本的请求命令 `GET/tools.html HTTP/1.1`，发送一个提供了源端主机名的 `Host` 首部，后面跟上一个空行，请求从服务器 www.joes-hardware.com 上获取资源 tools.html。随后，服务器会以一个响应行、几个响应首部、一个空行和最后面的 HTML 文档主体来应答。

要明确的是，Telnet 可以很好地模拟 HTTP 客户端，但不能作为服务器使用。而且对 Telnet 做脚本自动化是很繁琐乏味的。如果想要更灵活的工具，可以去看看 `nc`（netcat）。通过 `nc` 可以很方便地操纵基于 UDP 和 TCP 的流量（包括 HTTP），还可以为其编写脚本。更多细节参见 http://www.bgw.org/tutorials/utilities/nc.php[2]。

注 2：该链接已失效，读者可以访问 http://en.wikipedia.org/wiki/Netcat。（编者注）

1.7 协议版本

现在使用的 HTTP 协议有几个版本。HTTP 应用程序要尽量强健地处理各种不同的 HTTP 协议变体。目前仍在使用的版本如下。

- HTTP/0.9
  HTTP 的 1991 原型版本称为 HTTP/0.9。这个协议有很多严重的设计缺陷，只应该用于与老客户端的交互。HTTP/0.9 只支持 GET 方法，不支持多媒体内容的 MIME 类型、各种 HTTP 首部，或者版本号。HTTP/0.9 定义的初衷是为了获取简单的 HTML 对象，它很快就被 HTTP/1.0 取代了。

- HTTP/1.0
  1.0 是第一个得到广泛使用的 HTTP 版本。HTTP/1.0 添加了版本号、各种 HTTP 首部、一些额外的方法，以及对多媒体对象的处理。HTTP/1.0 使得包含生动图片的 Web 页面和交互式表格成为可能，而这些页面和表格促使万维网为人们广泛地接受。这个规范从未得到良好地说明。在这个 HTTP 协议的商业演进和学术研究都在快速进行的时代，它集合了一系列的最佳实践。

- HTTP/1.0+
  在 20 世纪 90 年代中叶，很多流行的 Web 客户端和服务器都在飞快地向 HTTP 中添加各种特性，以满足快速扩张且在商业上十分成功的万维网的需要。其中很多特性，包括持久的 keep-alive 连接、虚拟主机支持，以及代理连接支持都被加入到 HTTP 之中，并成为非官方的事实标准。这种非正式的 HTTP 扩展版本通常称为 HTTP/1.0+。

- HTTP/1.1
  HTTP/1.1 重点关注的是校正 HTTP 设计中的结构性缺陷，明确语义，引入重要的性能优化措施，并删除一些不好的特性。HTTP/1.1 还包含了对 20 世纪 90 年代末正在发展中的更复杂的 Web 应用程序和部署方式的支持。HTTP/1.1 是当前使用的 HTTP 版本。

- HTTP-NG（又名 HTTP/2.0）
  HTTP-NG 是 HTTP/1.1 后继结构的原型建议，它重点关注的是性能的大幅优化，以及更强大的服务逻辑远程执行框架。HTTP-NG 的研究工作终止于 1998 年，编写本书时，还没有任何要用此建议取代 HTTP/1.1 的推广计划。更多信息请参见第 10 章。

1.8 Web的结构组件

在本章的概述中，我们重点介绍了两个 Web 应用程序（Web 浏览器和 Web 服务器）是如何相互发送报文来实现基本事务处理的。在因特网上，要与很多 Web 应用程序进行交互。在本节中，我们将列出其他一些比较重要的应用程序，如下所示。

- *代理*
  位于客户端和服务器之间的 HTTP 中间实体。
- *缓存*
  HTTP 的仓库，使常用页面的副本可以保存在离客户端更近的地方。
- *网关*
  连接其他应用程序的特殊 Web 服务器。
- *隧道*
  对 HTTP 通信报文进行盲转发的特殊代理。
- Agent *代理*
  发起自动 HTTP 请求的半智能 Web 客户端。

1.8.1 代理

首先我们来看看 HTTP 代理服务器，这是 Web 安全、应用集成以及性能优化的重要组成模块。

如图 1-11 所示，代理位于客户端和服务器之间，接收所有客户端的 HTTP 请求，并将这些请求转发给服务器（可能会对请求进行修改之后转发）。对用户来说，这些应用程序就是一个代理，代表用户访问服务器。

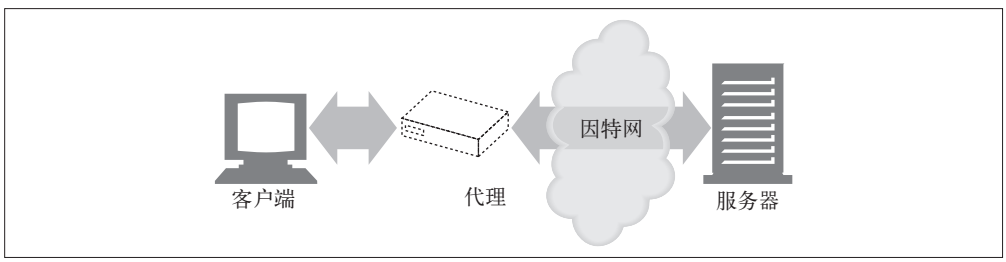


图 1-11 在客户端和服务器之间转发流量的代理

出于安全考虑，通常会将代理作为转发所有 Web 流量的可信任中间节点使用。代理还可以对请求和响应进行过滤。比如，在企业中对下载的应用程序进行病毒检测，或者对小学生屏蔽一些成人才能看的内容。我们将在第 6 章详细介绍代理。

1.8.2 缓存

Web 缓存（Web cache）或代理缓存（proxy cache）是一种特殊的 HTTP 代理服务器，可以将经过代理传送的常用文档复制保存起来。下一个请求同一文档的客户端就可以享受缓存的私有副本所提供的服务了（参见图 1-12）。

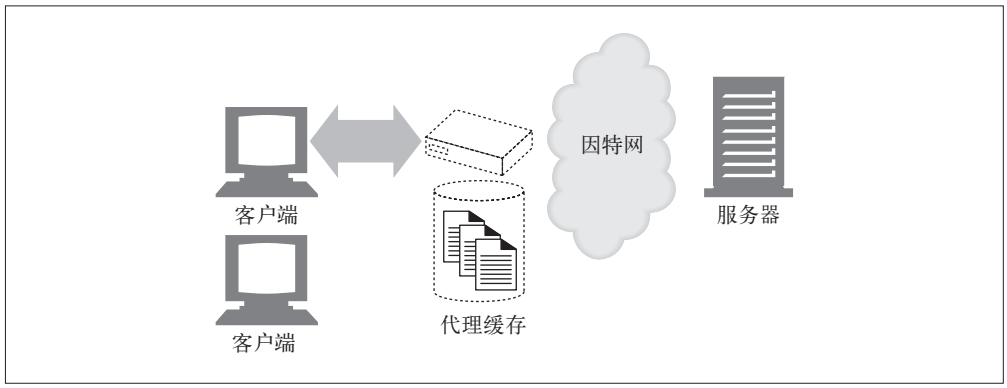


图 1-12 保存常用文档本地副本以提高性能的代理缓存

客户端从附近的缓存下载文档会比从远程 Web 服务器下载快得多。HTTP 定义了很多功能，使得缓存更加高效，并规范了文档的新鲜度和缓存内容的隐私性。第 7 章介绍了缓存技术。

1.8.3 网关

网关（gateway）是一种特殊的服务器，作为其他服务器的中间实体使用。通常用于将 HTTP 流量转换成其他的协议。网关接受请求时就好像自己是资源的源端服务器一样。客户端可能并不知道自己正在与一个网关进行通信。

例如，一个 HTTP/FTP 网关会通过 HTTP 请求接收对 FTP URI 的请求，但通过 FTP 协议来获取文档（参见图 1-13）。得到的文档会被封装成一条 HTTP 报文，发送给客户端。第 8 章将探讨网关。

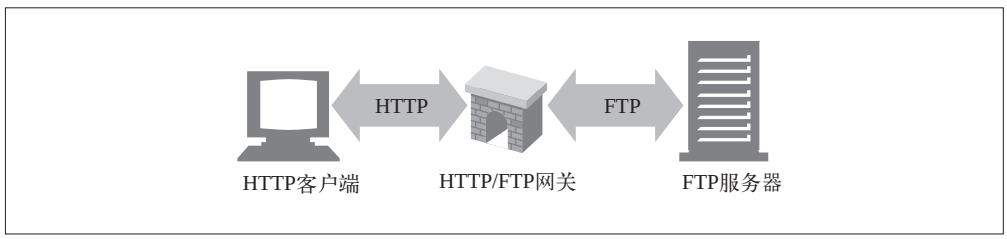


图 1-13 HTTP/FTP 网关

1.8.4 隧道

隧道（tunnel）是建立起来之后，就会在两条连接之间对原始数据进行盲转发的HTTP 应用程序。HTTP 隧道通常用来在一条或多条 HTTP 连接上转发非 HTTP 数据，转发时不会窥探数据。

HTTP 隧道的一种常见用途是通过 HTTP 连接承载加密的安全套接字层（SSL，Secure Sockets Layer）流量，这样 SSL 流量就可以穿过只允许 Web 流量通过的防火墙了。如图 1-14 所示，HTTP/SSL 隧道收到一条 HTTP 请求，要求建立一条到目的地址和端口的输出连接，然后在 HTTP 信道上通过隧道传输加密的 SSL 流量，这样就可以将其盲转发到目的服务器上去了。

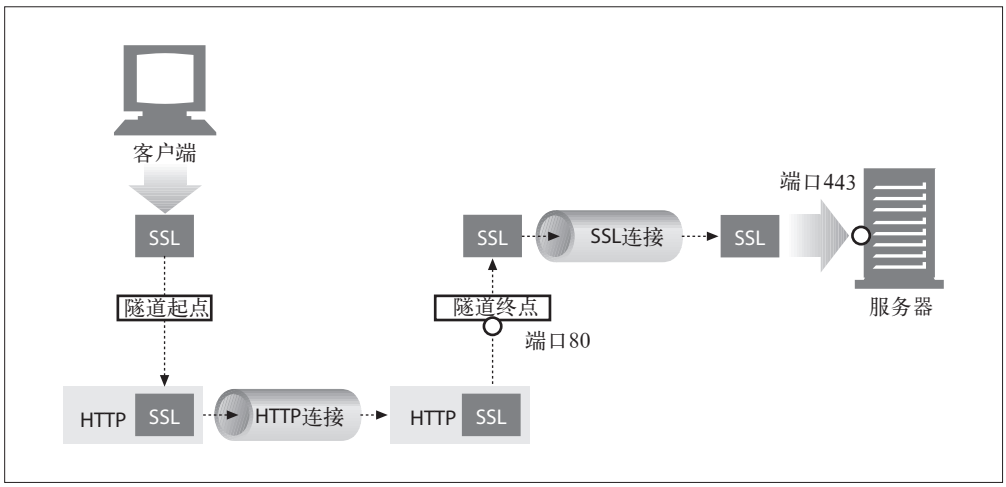


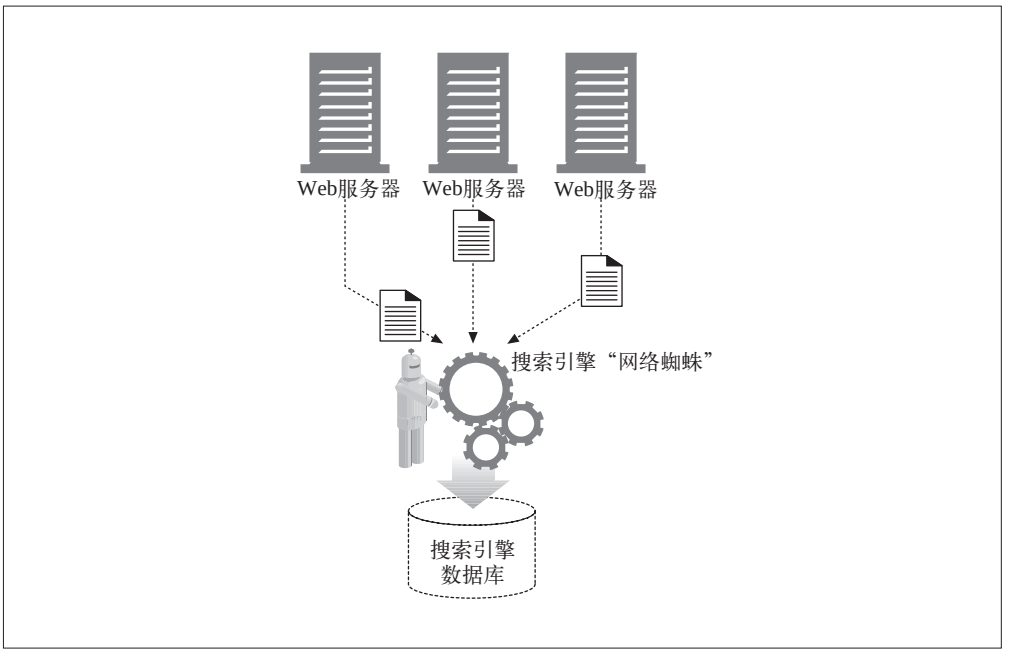
图 1-14　隧道可以在非 HTTP 网络上转发数据（显示的是 HTTP/SSL 隧道）

1.8.5 Agent代理

用户 Agent 代理（或者简称为 Agent 代理）是代表用户发起 HTTP 请求的客户端程序。所有发布 Web 请求的应用程序都是 HTTP Agent 代理。到目前为止，我们只提到过一种 HTTP Agent 代理：Web 浏览器，但用户 Agent 代理还有很多其他类型。

19

比如，有些自己会在 Web 上闲逛的自动用户 Agent 代理，可以在无人监视的情况下发布 HTTP 事务并获取内容。这些自动代理的名字通常都很生动，比如“网络蜘蛛”（spiders）或者“Web 机器人”（Web robots）（参见图 1-15）。网络蜘蛛会在 Web 上闲逛，搜集信息以构建有效的 Web 内容档案，比如一个搜索引擎的数据库或者为比较购物机器人生成的产品目录。更多信息请参见第 9 章。



20 图 1-15 自动搜索引擎“网络蜘蛛”就是 Agent 代理，可以从世界范围内获取 Web 页面

1.9 起始部分的结束语

这就是我们对 HTTP 的简要介绍。本章中，我们重点介绍了作为多媒体传输协议使用的 HTTP。概要说明了 HTTP 是怎样使用 URI 来命名远程服务器上的多媒体资源的，粗略介绍了如何利用 HTTP 请求和响应报文操纵远程服务器上的多媒体资源，最后考察了几种使用 HTTP 的 Web 应用程序。

本书的其余章节会更加详细地介绍 HTTP 协议、应用程序及资源的技术机制。

1.10 更多信息

本书稍后的章节将更深入地研究 HTTP，下面这些资源中也包含了与本章所涵盖的特定主题有关的背景知识。

1.10.1 HTTP协议信息

- *HTTP Pocket Reference*（《HTTP 口袋书》）
  Clinton Wong 著，O’Reilly & Associates 出版公司。这本书详细介绍了 HTTP，可以作为构成 HTTP 事务的首部和状态码的快速参考手册。

- http://www.w3.org/Protocols/
  这个 W3C 的 Web 页面中包含了很多与 HTTP 协议有关的重要链接。

- http://www.ietf.org/rfc/rfc2616.txt
  RFC2616“超文本传输协议——HTTP/1.1”是当前 HTTP 协议版本 HTTP/1.1 的官方规范。这个规范是一本编写流畅、组织良好而且非常详细的 HTTP 参考手册，但并不适于那些希望了解 HTTP 底层概念和动因，或者原理与实际应用之间区别的读者阅读。希望本书能够对这些底层概念进行补充，以便读者更好地使用这个规范。

- http://www.ietf.org/rfc/rfc1945.txt
  RFC1945“超文本传输协议——HTTP/1.0”是一个描述了 HTTP 现代基础的知识性 RFC。它对编写此规范时，已得到官方认可且具有“最佳实践”的 Web 应用程序行为进行了详细描述，还讨论了一些虽被 HTTP/1.1 所摒弃，但在一些老旧的应用程序中仍在广泛使用的行为。

- http://www.w3.org/Protocols/HTTP/AsImplemented.html
  这个 Web 页面介绍了 1991 年的 HTTP/0.9 协议，这个协议只实现了 GET 请求，而且不包含内容类型。

1.10.2 历史透视

- http://www.w3.org/Protocols/WhyHTTP.html
  这个简要的 Web 页面从 1991 年开始，从 HTTP 作者的角度，介绍了 HTTP 的一些起源以及初级目标。

- http://www.w3.org/History.html
  “A Little History of the World Wide Web”（万维网的简要历史）对万维网和 HTTP 的一些早期目标和构建基础进行了简短但有趣的剖析。

- http://www.w3.org/DesignIssues/Architecture.html
  “Web Architecture from 50,000 feet”（高空俯瞰 Web 结构）绘制了一幅广阔、远大的万维网蓝图，并详述了影响 HTTP 和相关 Web 技术的设计原则。

1.10.3 其他万维网信息

- http://www.w3.org
  W3C 是 Web 的科技驱动团队。W3C 致力于促进 Web 演化的互操作性技术（规范、准则、软件及工具）研究。W3C 站点是一个包含了 Web 技术简介和详细文档的宝库。

- http://www.ietf.org/rfc/rfc2396.txt
  RFC 2396“Uniform Resource Identifiers（URI）：Generic Syntax”，（“统一资源标识符（URI）：通用语法”）是 URI 和 URL 的详细参考。

- http://www.ieft.ofg/rfc/rfc2141.txt
  RFC2141“URN Syntax”（“URN 的语法”）是一个写于 1997 年的描述 URN 语法的规范。

- http://www.ietf.org/rfc/rfc2046.txt
  RFC2046“MIME Part 2：Media Types”（“MIME 第 II 部分：媒体类型”）是为进行多媒体内容管理而定义的多用途因特网邮件扩展标准的五部因特网规范中的第二部。

- http://www.wrec.org/Drafts/draft-ietf-wrec-taxonomy-06.txt
  这个因特网草案“Internet Web Replication and Caching Taxonomy”（“因特网 Web 复制和缓存分类法”）解释了 Web 结构组件中的标准术语。

22

第2章

URL与资源

我们可以把因特网当作一个巨大的正在扩张的城市，里面充满了各种可看的东西，可做的事情。你和其他居民，以及到这个正在蓬勃发展的社区旅游的游客都要为这个城市大量的景点和服务使用标准命名规范。博物馆、饭店和家庭住址要使用街道地址，消防局、老板的秘书，以及抱怨你太少打电话给她的母亲要使用电话号码。

所有的东西都有一个标准化的名字，以帮助人们寻找城市中的各种资源。书籍有ISBN 号，公交车有线路号，银行账户有账户编码，个人有社会保险号码。明天，你要到机场的 31 号出口去接你的生意伙伴。每天早上你都要乘坐红线火车，并在Kendall 广场站出站。

所有人都对这些名字的标准达成了一致，所以才能方便地共享这座城市的宝藏。你告诉出租车司机把你载到 McAllister 大街 246 号，他就知道你是什么意思了（即使他走的是一条很远的路）。

URL 就是因特网资源的标准化名称。URL 指向一条电子信息片段，告诉你它们位于何处，以及如何与之进行交互。

本章，我们将介绍以下内容：

- URL 语法，以及各种 URL 组件的含义及其所做的工作；
- 很多 Web 客户端都支持的 URL 快捷方式，包括相对 URL 和自动扩展 URL；
- URL 编码和字符规则；
- 支持各种因特网信息系统的常见 URL 方案；
- URL 的未来，包括 URN——这种框架可以在对象从一处搬移到另一处时，保持稳定的访问名称。

23

2.1 浏览因特网资源

URL 是浏览器寻找信息时所需的资源位置。通过 URL，人类和应用程序才能找到、使用并共享因特网上大量的数据资源。URL 是人们对 HTTP 和其他协议的常用访问点：一个人将浏览器指向一个 URL，浏览器就会在幕后发送适当的协议报文来获取人们所期望的资源。

URI 是一类更通用的资源标识符，URL 实际上是它的一个子集。URI 是一个通用的概念，由两个主要的子集 URL 和 URN 构成，URL 是通过描述资源的位置来标识资源的，而 URN（本章稍后会介绍）则是通过名字来识别资源的，与它们当前所处位置无关。

HTTP 规范将更通用的概念 URI 作为其资源标识符，但实际上，HTTP 应用程序处理的只是 URI 的 URL 子集。本书有时会不加区分地使用 URI 和 URL，但我们讲的基本上都是 URL。

比如说，你想要获取 URL http://www.joes-hardware.com/seasonal/index-fall.html。那么 URL 分以下三部分。

- URL 的第一部分（http）是 URL 方案（scheme）。方案可以告知 Web 客户端怎样访问资源。在这个例子中，URL 说明要使用 HTTP 协议。
- URL 的第二部分（www.joes-hardware.com）指的是服务器的位置。这部分告知 Web 客户端资源位于何处。
- URL 的第三部分（/seasonal/index-fall.html）是资源路径。路径说明了请求的是服务器上哪个特定的本地资源。

对此的说明请参见图 2-1。

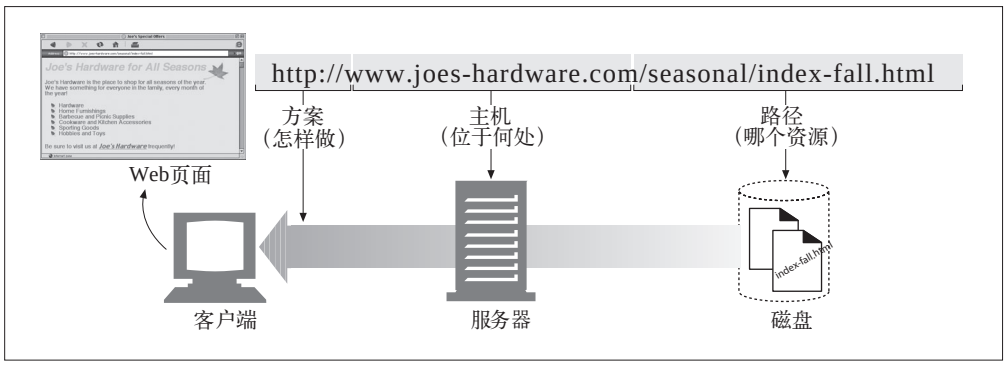


图 2-1　URL 是怎样与浏览器、客户端、服务器以及服务器文件系统中的位置进行关联的

URL 可以通过 HTTP 之外的其他协议来访问资源。它们可以指向因特网上的任意资源，或者个人的 E-mail 账户：

```
mailto:president@whitehouse.gov
```


或者通过其他协议（比如 FTP 协议）访问的各种文件：

```
ftp://ftp.lots-o-books.com/pub/complete-price-list.xls
```

或者从流视频服务器上下载电影：

```
rtsp://www.joes-hardware.com:554/interview/cto_video
```

URL 提供了一种统一的资源命名方式。大多数 URL 都有同样的：“方案 :// 服务器位置 / 路径”结构。因此，对网络上的每个资源以及获取这些资源的每种方式来说，命名资源的方法都只有一种，这样不管是谁都可以用名字来找到这个资源了。但是，事情并不是一开始就是这样的。

URL出现之前的黑暗岁月

在 Web 和 URL 出现之前，人们要靠分类杂乱的应用程序来访问分布在网络中的数据。大多数人都不会幸运地拥有所有合适的应用程序，或者不能够理解，也没有足够的耐心来使用这些程序。

在 URL 出现之前，要想和朋友共享 complete-catalog.xls 文件，就得说这样一些话："用 FTP 连接到 ftp.joes-hardware.com 上。用匿名登录，然后输入你的用户名作为密码。变换到 pub 目录。转换为二进制模式。现在，可以将名为 complete-catalog.xls 的文件下载到本地文件系统，并在那里浏览这个文件了。"

现在，像网景的 Navigator 和微软的 Internet Explorer 这样的浏览器都将很多这样的功能捆绑成一个便捷包。通过 URL，这些应用程序就可以通过一个接口，以统一的方式去访问许多资源了。只要说"将浏览器指向 ftp://ftp.lots-o-books.com/pub/complete-catalog.xls"就可以取代上面那些复杂的指令了。

URL 为应用程序提供了一种访问资源的手段。实际上，很多用户可能都不知道他们的浏览器在获取所请求资源时所使用的协议和访问方法。

有了 Web 浏览器，就不再需要用新闻阅读器来阅读因特网新闻，或者用 FTP 客户端来访问 FTP 服务器上的文件了，而且也无需用电子邮件程序来收发 E-mail 报文了。URL 告知浏览器如何对资源进行访问和处理，这有助于简化复杂的网络世界。[1] 应用程序可以使用 URL 来简化信息的访问过程。

URL 为用户及他们的浏览器提供了找到信息所需的所有条件。URL 定义了用户所需
25 的特定资源，它位于何处以及如何获取它。

2.2 URL的语法

URL 提供了一种定位因特网上任意资源的手段，但这些资源是可以通过各种不同的方案（比如 HTTP、FTP、SMTP）来访问的，因此 URL 语法会随方案的不同而有所不同。

这是不是意味着每种不同的 URL 方案都会有完全不同的语法呢？实际上，不是的。大部分 URL 都遵循通用的 URL 语法，而且不同 URL 方案的风格和语法都有不少重叠。

注 1：浏览器通常会用其他应用程序来处理特殊的资源。比如，Internet Explorer 就启动了一个 E-mail 应用程序来处理那些表示 E-mail 资源的 URL。

大多数 URL 方案的 URL 语法都建立在这个由 9 部分构成的通用格式上：

```
<scheme>://<user>:<password>@<host>:<port>/<path>;<params>?<query>#<frag>
```

几乎没有哪个 URL 中包含了所有这些组件。URL 最重要的 3 个部分是方案（scheme）、主机（host）和路径（path）。表 2-1 对各种组件进行了总结。

表2-1　通用URL组件

组　件	描　述	默 认 值
方案	访问服务器以获取资源时要使用哪种协议	无
用户	某些方案访问资源时需要的用户名	匿名
密码	用户名后面可能要包含的密码，中间由冒号（:）分隔	<E-mail 地址 >
主机	资源宿主服务器的主机名或点分 IP 地址	无
端口	资源宿主服务器正在监听的端口号。很多方案都有默认端口号（HTTP 的默认端口号为 80）	每个方案特有
路径	服务器上资源的本地名，由一个斜杠（/）将其与前面的 URL 组件分隔开来。路径组件的语法是与服务器和方案有关的（本章稍后会讲到 URL 路径可以分为若干个段，每段都可以有其特有的组件。）	无
参数	某些方案会用这个组件来指定输入参数。参数为名 / 值对。URL 中可以包含多个参数字段，它们相互之间以及与路径的其余部分之间用分号（;）分隔	无
查询	某些方案会用这个组件传递参数以激活应用程序（比如数据库、公告板、搜索引擎以及其他因特网网关）。查询组件的内容没有通用格式。用字符“?”将其与 URL 的其余部分分隔开来	无
片段	一小片或一部分资源的名字。引用对象时，不会将 frag 字段传送给服务器；这个字段是在客户端内部使用的。通过字符“#”将其与 URL 的其余部分分隔开来	无

比如，我们来看看 URL：http://www.joes-hardware.com:80/index.html，其方案是 http，主机为 www.joes-hardware.com，端口是 80，路径为 /index.html。

26

2.2.1　方案——使用什么协议

方案实际上是规定如何访问指定资源的主要标识符，它会告诉负责解析 URL 的应用程序应该使用什么协议。在我们这个简单的 HTTP URL 中所使用的方案就是 http。

方案组件必须以一个字母符号开始，由第一个“:”符号将其与 URL 的其余部分分隔开来。方案名是大小写无关的，因此 URL“http://www.joes-hardware.com”和“HTTP://www.joes-hardware.com”是等价的。

2.2.2 主机与端口

要想在因特网上找到资源，应用程序要知道是哪台机器装载了资源，以及在那台机器的什么地方可以找到能对目标资源进行访问的服务器。URL 的*主机*和*端口*组件提供了这两组信息。

主机组件标识了因特网上能够访问资源的宿主机器。可以用上述主机名（www.joes-hardware.com），或者 IP 地址来表示主机名。比如，下面两个 URL 就指向同一个资源——第一个 URL 是通过主机名，第二个是通过 IP 地址指向服务器的：

```
http://www.joes-hardware.com:80/index.html
http://161.58.228.45:80/index.html
```

端口组件标识了服务器正在监听的网络端口。对下层使用了 TCP 协议的 HTTP 来说，默认端口号为 80。

2.2.3 用户名和密码

更有趣的组件是*用户*和*密码*组件。很多服务器都要求输入用户名和密码才会允许用户访问数据。FTP 服务器就是这样一个常见的实例。这里有几个例子：

```
ftp://ftp.prep.ai.mit.edu/pub/gnu
ftp://anonymous@ftp.prep.ai.mit.edu/pub/gnu
ftp://anonymous:my_passwd@ftp.prep.ai.mit.edu/pub/gnu
http://joe:joespasswd@www.joes-hardware.com/sales_info.txt
```

第一个例子没有用户或密码组件，只有标准的方案、主机和路径。如果某应用程序使用的 URL 方案要求输入用户名和密码，比如 FTP，但用户没有提供，它通常会插入一个默认的用户名和密码。比如，如果向浏览器提供一个 FTP URL，但没有指定用户名和密码，它就会插入 anonymous（匿名用户）作为你的用户名，并发送一个默认的密码（Internet Explorer 会发送 IEUser，Netscape Navigator 则会发送 mozilla）。

27

第二个例子显示了一个指定为 anonymous 的用户名。这个用户名与主机组件组合在一起，看起来就像 E-mail 地址一样。字符“@”将用户和密码组件与 URL 的其余部分分隔开来。

在第三个例子中，指定了用户名（anonymous）和密码（my_passwd），两者之间由字符“:”分隔。

2.2.4 路径

URL 的路径组件说明了资源位于服务器的什么地方。路径通常很像一个分级的文件系统路径。比如：

http://www.joes-hardware.com:80/seasonal/index-fall.html

这个 URL 中的路径为 /seasonal/index-fall.html，很像 UNIX 文件系统中的文件系统路径。路径是服务器定位资源时所需的信息。[2] 可以用字符“/”将 HTTP URL 的路径组件划分成一些*路径段*（path segment）（还是与 UNIX 文件系统中的文件路径类似）。每个路径段都有自己的*参数*（param）组件。

2.2.5 参数

对很多方案来说，只有简单的主机名和到达对象的路径是不够的。除了服务器正在监听的端口，以及是否能够通过用户名和密码访问资源外，很多协议都还需要更多的信息才能工作。

负责解析 URL 的应用程序需要这些协议参数来访问资源。否则，另一端的服务器可能就不会为请求提供服务，或者更糟糕的是，提供错误的服务。比如，像 FTP 这样的协议，有两种传输模式，二进制和文本形式。你肯定不希望以文本形式来传送二进制图片，这样的话，二进制图片可能会变得一团糟。

为了向应用程序提供它们所需的输入参数，以便正确地与服务器进行交互，URL 中有一个*参数*组件。这个组件就是 URL 中的名值对列表，由字符“;”将其与 URL 的其余部分（以及各名值对）分隔开来。它们为应用程序提供了访问资源所需的所有附加信息。比如：

ftp://prep.ai.mit.edu/pub/gnu;type=d

在这个例子中，有一个参数 type=d，参数名为 type，值为 d。

如前所述，HTTP URL 的路径组件可以分成若干路径段。每段都可以有自己的参数。比如：

http://www.joes-hardware.com/hammers;sale=false/index.html;graphics=true

这个例子就有两个路径段，hammers 和 index.html。hammers 路径段有参数 sale，其值为 false。index.html 段有参数 graphics，其值为 true。

注 2：这是一种简化的说法。在 18.2 节我们会看到，路径并不总能为资源定位提供足够的信息。有时服务器还需要其他的信息。

2.2.6 查询字符串

很多资源，比如数据库服务，都是可以通过提问题或进行查询来缩小所请求资源类型范围的。

假设 Joe 的五金商店在数据库中维护着一个未售货物的清单，并可以对清单进行查询，以判断产品是否有货，那就可以用下列 URL 来查询 Web 数据库网关，看看编号为 12731 的条目是否有货：

http://www.joes-hardware.com/inventory-check.cgi?item=12731

这个 URL 的大部分都与我们见过的其他 URL 类似。只有问号（?）右边的内容是新出现的。这部分被称为查询（query）组件。URL 的查询组件和标识网关资源的 URL 路径组件一起被发送给网关资源。基本上可以将网关当作访问其他应用程序的访问点（第 8 章会对网关进行详细的讨论）。

图 2-2 中有一个作为 Joe 的五金商店清单查询应用程序的网关的服务器，在这个例子中向此服务器发送了一个查询组件。查询的目的是检查清单中是否有尺寸为 large、颜色为 blue 的条目 12731。

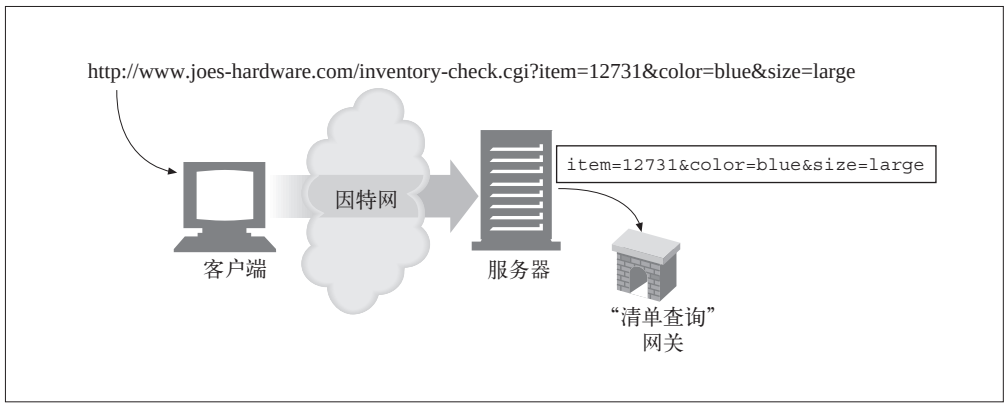


图 2-2 发送给网关应用程序的 URL 查询组件

在本章稍后会看到，除了有些不合规则的字符需要特别处理之外，对查询组件的格式没什么要求。按照常规，很多网关都希望查询字符串以一系列“名 / 值”对的形式出现，名值对之间用字符“&”分隔：

29

http://www.joes-hardware.com/inventory-check.cgi?item=12731&color=blue

在这个例子中，查询组件有两个名 / 值对：item=12731 和 color=blue。

2.2.7 片段

有些资源类型，比如 HTML，除了资源级之外，还可以做进一步的划分。比如，对一个带有章节的大型文本文档来说，资源的 URL 会指向整个文本文档，但理想的情况是，能够指定资源中的那些章节。

为了引用部分资源或资源的一个片段，URL 支持使用片段（frag）组件来表示一个资源内部的片段。比如，URL 可以指向 HTML 文档中一个特定的图片或小节。

片段挂在 URL 的右手边，最前面有一个字符“#”。比如：

http://www.joes-hardware.com/tools.html#drills

在这个例子中，片段 drills 引用了 Joe 的五金商店 Web 服务器上页面 /tools.html 中的一个部分。这部分的名字叫做 drills。

HTTP 服务器通常只处理整个对象，[3] 而不是对象的片段，客户端不能将片段传送给服务器（参见图 2-3）。浏览器从服务器获得了整个资源之后，会根据片段来显示你感兴趣的那部分资源。

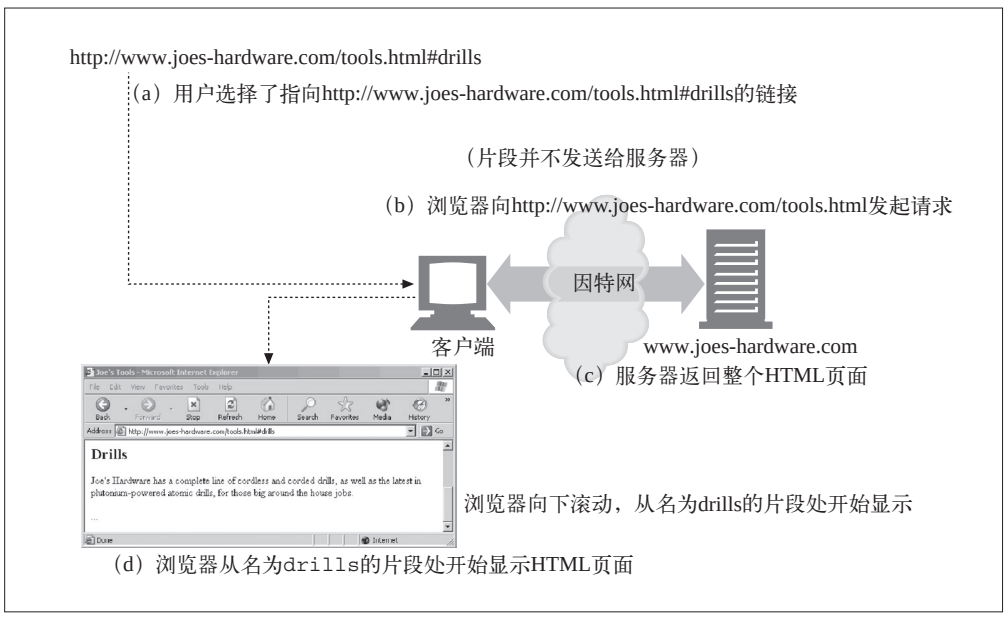


图 2-3　服务器处理的是整个对象，因此 URL 片段仅由客户端使用

注 3：在 15.9 节会看到 HTTP Agent 代理可能会请求某个字节范围内的对象，但在 URL 片段的上下文中，服务器会发送整个对象，由 Agent 代理将片段标识符应用于资源。

2.3 URL快捷方式

Web 客户端可以理解并使用几种 URL 快捷方式。相对 URL 是在某资源内部指定一个资源的便捷缩略方式。很多浏览器还支持 URL 的“自动扩展”，也就是用户输入 URL 的一个关键（可记忆的）部分，然后由浏览器将其余部分填充起来。2.3.2 节对此进行了解释。

2.3.1 相对URL

URL 有两种方式：*绝对的*和*相对的*。到目前为止，我们只见过绝对 URL。绝对
30 URL 中包含有访问资源所需的全部信息。

另一方面，相对 URL 是不完整的。要从相对 URL 中获取访问资源所需的全部信息，就必须相对于另一个，被称为其*基础*（base）的 URL 进行解析。

相对 URL 是 URL 的一种便捷缩略记法。如果你手工写过 HTML 的话，可能就会发现相对 URL 是多么便捷了。例 2-1 是一个嵌入了相对 URL 的 HTML 文档实例。

例 2-1　带有相对 URL 的 HTML 代码片段

```
<HTML>
<HEAD><TITLE>Joe's Tools</TITLE></HEAD>
<BODY>
<H1> Tools Page </H1>
<H2> Hammers <H2>
<P> Joe's Hardware Online has the largest selection of <A HREF="./
hammers.html">hammers
</A> on earth.
</BODY>
</HTML>
```

例 2-1 是资源：

http://www.joes-hardware.com/tools.html

的 HTML 文档。

在这个 HTML 文档中有一个包含了 URL ./hammers.html 的超链接。这个 URL 看起来是不完整的，但实际上是个合法的相对 URL。可以相对于它所在文档的 URL 对其进行解释；在这个例子中，就是相对于 Joe 的五金商店 Web 服务器的资源 /tools.
31 html。

使用缩略形式的相对 URL 语法，HTML 的编写者就可以省略 URL 中的方案、主机和其他一些组件了。这些组件可以从它们所属资源的*基础* URL 中推导出来。其他资源的 URL 也可以用这种缩略形式来表示。

在例 2-1 中，基础 URL 为：

http://www.joes-hardware.com/tools.html

用这个 URL 作为基础，可以推导出缺失的信息。我们知道资源名为 ./hammers.html，但并不知道方案或主机名是什么。通过这个基础 URL，可以推导出方案为 http，主机为 www.joes-hardware.com。图 2-4 对此进行了说明。



图 2-4　使用基础 URL

相对 URL 只是 URL 的片段或一小部分。处理 URL 的应用程序（比如浏览器）要能够在相对和绝对 URL 之间进行转换。

还需要注意的是，相对 URL 为保持一组资源（比如一些 HTML 页面）的可移植性提供了一种便捷方式。如果使用的是相对 URL，就可以在搬移一组文档的同时，仍然保持链接的有效性，因为相对 URL 都是相对于新基础进行解释的。这样就可以实现在其他服务器上提供镜像内容之类的功能了。

1. 基础URL

转换处理的第一步就是找到基础 URL。基础 URL 是作为相对 URL 的参考点使用的。可以来自以下几个不同的地方。

- *在资源中显式提供*
  有些资源会显式地指定基础 URL。比如，HTML 文档中可能会包含一个定义了基础 URL 的 HTML 标记 `<BASE>`，通过它来转换那个 HTML 文档中的所有相对 URL。

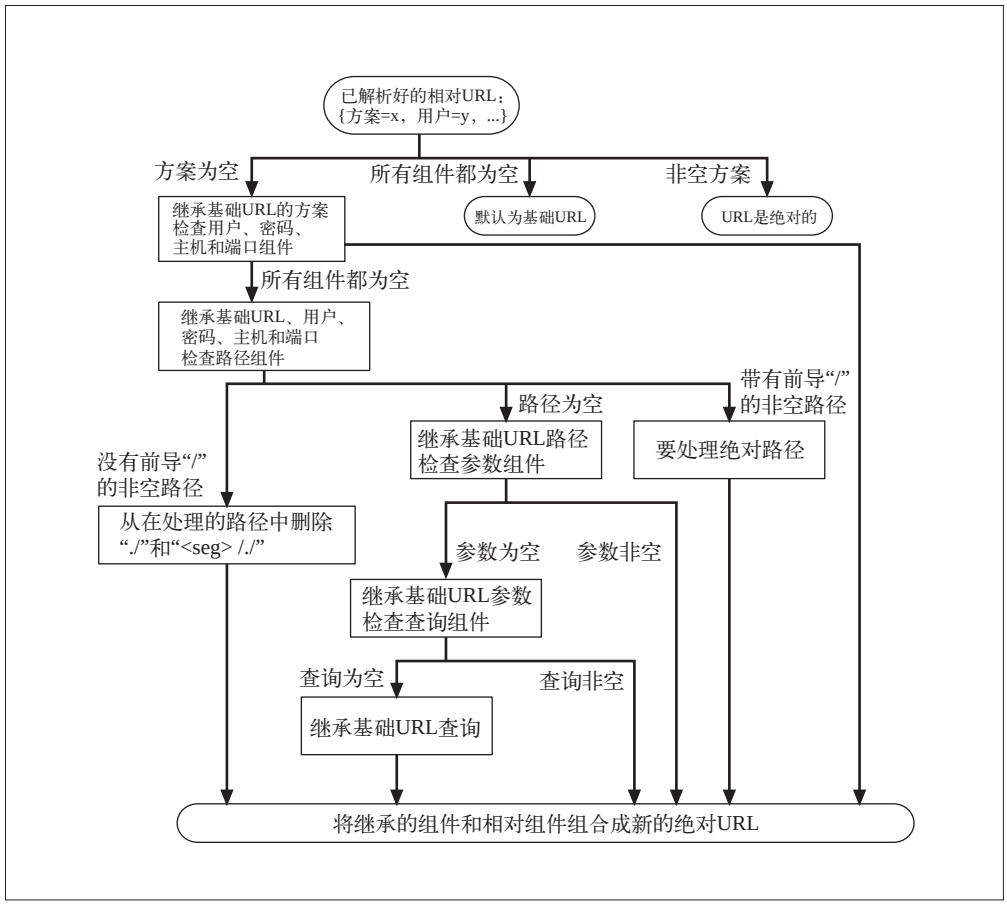
- *封装资源的基础 URL*
  如果在一个没有显式指定基础 URL 的资源中发现了一个相对 URL，如例 2-1 所示，可以将它所属资源的 URL 作为基础（如例中所示）。

- *没有基础 URL*
  在某些情况下，没有基础 URL。这通常意味着你有一个绝对 URL；但有时可能只是一个不完整或损坏了的 URL。

2. 解析相对引用

前面我们介绍了 URL 的基本组件和语法。要将相对 URL 转换为一个绝对 URL，下一步要做的就是将相对 URL 和基础 URL 划分成组件段。

实际上，这样只是在解析 URL，但这种做法会将其划分成一个个组件，因此通常会称作分解（decomposing）URL。只要将基础和相对 URL 划分成了组件，就可以应用图 2-5 中的算法来完成转换了。



33 图 2-5　将相对 URL 转换成绝对 URL

这个算法将一个相对 URL 转换成了其绝对模式，之后就可以用它来引用资源了。这个算法最初是在 RFC 1808 中制定的，后来被合并到了 RFC 2396 中。

可以对例 2-1 中的 ./hammers.html 实例使用图 2-5 中描述的算法。

(1) 路径为 ./hammers.html，基础 URL 为 http://www.joes-hardware.com/tools.html。
(2) 方案为空，沿着图表的左半边向下处理，继承基础 URL 方案（HTTP）。
(3) 至少一个组件非空，一直处理到底端，继承主机和端口组件。
(4) 将来自相对 URL（路径：./hammers.html）的组件与我们继承来的组件（方案：http，主机：www.joes-hardware.com，端口：80）合并起来，得到新的绝对 URL：http ://www.joes-hardware.com/hammers.html。

2.3.2 自动扩展URL

有些浏览器会在用户提交 URL 之后，或者在用户输入的时候尝试着自动扩展 URL。这就为用户提供了一条捷径：用户不需要输入完整的 URL，因为浏览器会自动扩展。

这些“自动扩展”特性有以下两种方式。

- *主机名扩展*
  在主机名扩展中，只要有些小提示，浏览器通常就可以在没有帮助的情况下，将你输入的主机名扩展为完整的主机名。
  比如，如果在地址栏中输入 yahoo，浏览器就会自动在主机名中插入 www. 和 .com，构建出 www.yahoo.com。如果找不到与 yahoo 匹配的站点，有些浏览器会在放弃之前尝试几种扩展形式。浏览器通过这些简单的技巧来节省你的时间，减少找不到的可能。
  但是，这些主机名扩展技巧可能会为其他一些 HTTP 应用程序带来问题，比如代理。第 6 章将详细讨论这些问题。

- *历史扩展*
  浏览器用来节省用户输入 URL 时间的另一种技巧是，将以前用户访问过的 URL 历史存储起来。当你输入 URL 时，它们就可以将你输入的 URL 与历史记录中 URL 的前缀进行匹配，并提供一些完整的选项供你选择。因此，如果你输入了一个以前访问过的 URL 的开始部分，比如 http://www.joes-，浏览器就可能会建议使用 http://www.joes-hardware.com。然后你就可以选择这个地址，不用输入完整的 URL 了。

注意，与代理共同使用时，URL 自动扩展的行为可能会有所不同。6.5.6 节将对此进行进一步讨论。

2.4 各种令人头疼的字符

URL 是*可移植的*（portable）。它要统一地命名因特网上所有的资源，这也就意味着要通过各种不同的协议来传送这些资源。这些协议在传输数据时都会使用不同的机制，所以，设计 URL，使其可以通过任意因特网协议安全地传输是很重要的。

安全传输意味着 URL 的传输不能丢失信息。有些协议，比如传输电子邮件的简单邮件传输协议（Simple Mail Transfer Protocol，SMTP），所使用的传输方法就会剥去一些特定的字符。[4] 为了避开这些问题，URL 只能使用一些相对较小的、通用的安全字母表中的字符。

除了希望 URL 可以被所有因特网协议进行传送之外，设计者们还希望 URL 也可供人类阅读。因此，即使不可见、不可打印的字符能够穿过邮件程序，从而成为可移植的，也不能在 URL 中使用。[5]

URL 还得是*完整的*，这就使问题变得更加复杂了。URL 的设计者们认识到有时人们可能会希望 URL 中包含除通用的安全字母表之外的二进制数据或字符。因此，需要有一种转义机制，能够将不安全的字符编码为安全字符，再进行传输。

本节总结了 URL 的通用字母表和编码规则。

2.4.1 URL字符集

默认的计算机系统字符集通常都倾向于以英语为中心。从历史上来看，很多计算机应用程序使用的都是 US-ASCII 字符集。US-ASCII 使用 7 位二进制码来表示英文打字机提供的大多数按键和少数用于文本格式和硬件通知的不可打印控制字符。

由于 US-ASCII 的历史悠久，所以其可移植性很好。但是，虽然美国用户使用起来很便捷，它却并不支持在各种欧洲语言或全世界数十亿人使用的数百种非罗马语言
35 中很常见的变体字符。

而且，有些 URL 中还会包含任意的二进制数据。认识到对完整性的需求之后，URL 的设计者就将*转义序列*集成了进去。通过转义序列，就可以用 US-ASCII 字符集的有限子集对任意字符值或数据进行编码了，这样就实现了可移植性和完整性。

2.4.2 编码机制

为了避开安全字符集表示法带来的限制，人们设计了一种编码机制，用来在 URL 中表示各种不安全的字符。这种编码机制就是通过一种“转义”表示法来表示不安全

注 4：这是由报文的 7 位二进制码编码造成的。如果源端是以 8 位或更多位编码的，就会有部分信息被剥离。
注 5：不可打印字符中包括空格符（注意，RFC 2396 建议应用程序忽略空格符）。

字符的，这种转义表示法包含一个百分号（%），后面跟着两个表示字符 ASCII 码的十六进制数。

表 2-2 中列出了几个例子。

表2-2　一些编码字符示例

字　符	ASCII码	示例URL
~	126(0x7E)	http://www.joes-hardware.com/%7Ejoe
空格	32(0x20)	http://www.joes-hardware.com/more%20tools.html
%	37(0x25)	http://www.joes-hardware.com/100%25satisfaction.html

2.4.3　字符限制

在 URL 中，有几个字符被保留起来，有着特殊的含义。有些字符不在定义的 US-ASCII 可打印字符集中。还有些字符会与某些因特网网关和协议产生混淆，因此不赞成使用。

表 2-3 列出了一些字符，在将其用于保留用途之外的场合时，要在 URL 中对其进行编码。

表2-3　保留及受限的字符

字　符	保留/受限
%	保留作为编码字符的转义标志
/	保留作为路径组件中分隔路径段的定界符
.	保留在路径组件中使用
..	保留在路径组件中使用
#	保留作为分段定界符使用
?	保留作为查询字符串定界符使用
;	保留作为参数定界符使用
:	保留作为方案、用户 / 口令，以及主机 / 端口组件的定界符使用
$, +	保留
@ & =	在某些方案的上下文中有特殊的含义，保留
{ } | \ ^ ~ [ ] '	由于各种传输 Agent 代理，比如各种网关的不安全处理，使用受限
< > "	不安全；这些字符在 URL 范围之外通常是有意义的，比如在文档中对 URL 自身进行定界（比如 http://www.joes-hardware.com），所以应该对其进行编码
0x00-0x1F, 0x7F	受限，这些十六进制范围内的字符都在 US-ASCII 字符集的不可打印区间内
>0x7F	受限，十六进制值在此范围内的字符都不在 US-ASCII 字符集的 7 比特范围内

2.4.4 另外一点说明

你可能会感到奇怪，为什么使用一些不安全字符的时候并没有发生什么不好的事情。比如，你可以访问 http://www.joes-hardware.com/~joe 上的 Joe 主页，而无需对“~”字符进行编码。对某些传输协议来说，这并不是什么问题，但对应用程序开发人员来说，对非安全字符进行编码仍然是明智的。

应用程序要按照一定规范工作。客户端应用程序在向其他应用程序发送任意 URL 之前，最好把所有不安全或受限字符都进行转换[6]。只要对所有不安全字符都进行了编码，这个 URL 就是可在各应用程序之间共享的*规范形式*；也就无需操心其他应用程序会被字符的任何特殊含义所迷惑了。

最适合判断是否需要对字符进行编码的程序就是从用户处获取 URL 的源端应用程序。URL 的每个组件都会有自己的安全 / 不安全字符，哪些字符是安全 / 不安全的与方案有关，因此只有从用户那里接收 URL 的应用程序才能够判断需要对哪些字符进行编码。

当然，另一种极端的做法就是应用程序对所有字符都进行编码。尽管并不建议这么做，但也没有什么强硬且严格的规则规定不能对那些安全字符进行编码；但在实际应用中，有些应用程序可能会假定不对安全字符进行编码，这么做的话可能会产生一些奇怪的破坏性行为。

有时，有些人会恶意地对额外的字符进行编码，以绕过那些对 URL 进行模式匹配的应用程序——比如，Web 过滤程序。对安全的 URL 组件进行编码会使模式匹配程序无法识别出它们所要搜寻的模式。总之，解释 URL 的应用程序必须在处理 URL 之前对其进行解码。

有些 URL 组件要便于识别，并且必须由字母开头，比如 URL 的方案。更多关于不同 URL 组件中保留字符和不安全字符的使用指南请回顾 2.2 节[7]。

2.5 方案的世界

本节将介绍更多 Web 常用方案格式。附录 A 给出了一个相当完整的方案列表，及各种方案文档的参考文献。

注 6：这里我们特指的是客户端应用程序，而不是其他的 HTTP 中间点，比如代理。6.5.5 节探讨了代理和其他中间 HTTP 应用程序试图代表客户端修改（比如编码）URL 时可能产生的一些问题。

注 7：表 2-3 列出了各种 URL 组件的保留字符。总之，只应该对这些在传输过程中不安全的字符进行编码。

表 2-4 总结了最常见的一些方案。回顾一下 2.2 节有助于理解表格中的语法部分。

表2-4 常见的方案格式

方　案	描　述
http	超文本传输协议方案，除了没有用户名和密码之外，与通用的 URL 格式相符。如果省略了端口，就默认为 80。 基本格式： http://<host>:<port>/<path>?<query>#<frag> 示例： http://www.joes-hardware.com/index.html http://www.joes-hardware.com:80/index.html
https	方案 https 与方案 http 是一对。唯一的区别在于方案 https 使用了网景的 SSL，SSL 为 HTTP 连接提供了端到端的加密机制。其语法与 HTTP 的语法相同，默认端口为 443。 基本格式： https://<host>:<port>/<path>?<query>#<frag> 示例： https://www.joes-hardware.com/secure.html
mailto	Mailto URL 指向的是 E-mail 地址。由于 E-mail 的行为与其他方案都有所不同（它并不指向任何可以直接访问的对象），所以 mailto URL 的格式与标准 URL 的格式也有所不同。因特网 E-mail 地址的语法记录在 RFC 822 中。 基本格式： mailto:<RFC-822-addr-spec> 示例： mailto:joe@joes-hardware.com
ftp	文件传输协议 URL 可以用来从 FTP 服务器上下载或向其上载文件，并获取 FTP 服务器上的目录结构内容的列表。 在 Web 和 URL 出现之前 FTP 就已经存在了。Web 应用程序将 FTP 作为一种数据访问方案使用。URL 语法遵循下列通用格式。 基本格式： ftp://<user>:<password>@<host>:<port>/<path>;<params> 示例： ftp://anonymous:joe%40joes-hardware.com@prep.ai.mit.edu:21/pub/gnu/
rtsp，rtspu	RTSP URL 是可以通过实时流传输协议（Real Time Streaming Protocol）解析的音 / 视频媒体资源的标识符。 方案 rtspu 中的 u 表示它是使用 UDP 协议来获取资源的。 基本格式： rtsp://<user>:<password>@<host>:<port>/<path> rtspu://<user>:<password>@<host>:<port>/<path> 示例： rtsp://www.joes-hardware.com:554/interview/cto_video

（续）

方　案	描　述
file	方案 file 表示一台指定主机（通过本地磁盘、网络文件系统或其他一些文件共享系统）上可直接访问的文件。各字段都遵循通用格式。如果省略了主机名，就默认为正在使用 URL 的本地主机。 基本格式： file://<host>/<path> 示例： file://OFFICE-FS/policies/casual-fridays.doc
news	根据 RFC 1036 的定义，方案 news 用来访问一些特定的文章或新闻组。它有一个很独特的性质：news URL 自身包含的信息不足以对资源进行定位。 news URL 中缺乏到何处获取资源的信息——没有提供主机名或机器名称。从用户那里获取此类信息是解释程序的工作。比如，在网景浏览器的“选项”（Options）菜单中，就可以指定自己的 NNTP（news）服务器。这样，浏览器有了 news URL 的时候就知道应该使用哪个服务器了。 新闻资源可以从多台服务器中获得。它们被称为位置无关的，因为对它们的访问不依赖于任何一个源服务器。 news URL 中保留了字符“@”，用来区分指向新闻组的 news URL 和指向特定新闻文章的 news URL。 基本格式： news:<newsgroup> news:<news-article-id> 示例： news:rec.arts.startrek
telnet	方案 telnet 用于访问交互式业务。它表示的并不是对象自身，而是可通过 telnet 协议访问的交互式应用程序（资源）。 基本格式： telnet://<user>:<password>@<host>:<port>/ 示例： telnet://slurp:webhound@joes-hardware.com:23/

39

2.6　未来展望

URL 是一种强有力的工具。它可以用来命名所有现存对象，而且可以很方便地包含一些新格式。URL 还提供了一种可以在各种因特网协议间共享的统一命名机制。

但 URL 并不完美。它们表示的是实际的地址，而不是准确的名字。这就意味着 URL 会告诉你资源此时处于什么位置。它会为你提供特定端口上特定服务器的名字，告诉你在何处可以找到这个资源。这种方案的缺点在于如果资源被移走了，URL 也就不再有效了。那时，它就无法对对象进行定位了。

如果有了对象的准确名称，则不论其位于何处都可以找到这个对象，那该多完美啊。就像人一样，只要给定了资源的名称和其他一些情况，无论资源移到何处，你都能够追踪到它。

为了应对这个问题，因特网工程任务组（Internet Engineering Task Force，IETF）已经对一种名为统一资源名（uniform resource name，URN）的新标准做了一段时间的研究了。无论对象搬移到什么地方（在一个 Web 服务器内或是在不同的 Web 服务器间），URN 都能为对象提供一个稳定的名称。

永久统一资源定位符（persistent uniform resource locators，PURL）是用 URL 来实现 URN 功能的一个例子。其基本思想是在搜索资源的过程中引入另一个中间层，通过一个中间资源定位符（resource locator）服务器对资源的实际 URL 进行登记和跟踪。客户端可以向定位符请求一个永久 URL，定位符可以以一个资源作为响应，将客户端重定向到资源当前实际的 URL 上去（参见图 2-6）。更多有关 PURL 的信息，请访问 http://purl.oclc.org。

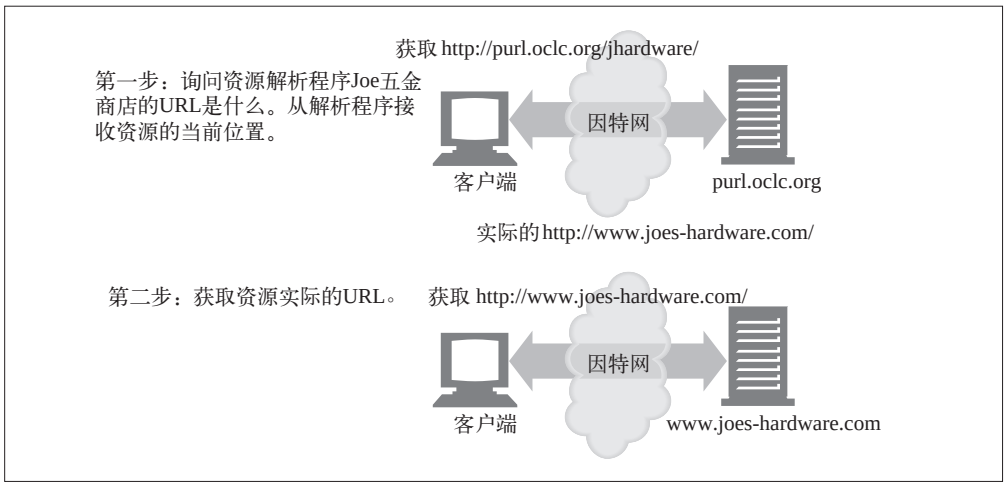


图 2-6　PURL 通过资源定位符服务器来命名资源的当前位置

如果不是现在，那是什么时候

URN 背后的思想已经提出一段时间了。实际上，如果去看看某些相关规范的发布日期，你可能会问，为什么它们现在都还没有投入使用。

40

从 URL 转换成 URN 是一项巨大的工程。标准化工作的进程很缓慢，而且通常都有很充分的理由。支持 URN 需要进行很多改动——标准主体的一致性，对各种 HTTP

应用程序的修改等。做这种改动需要进行大量的工作，而且很不幸（或者可能很幸运）的是 URL 还有很大的能量，还要等待更合适的时机才能进行这种转换。

在 Web 爆炸性增长的过程中，因特网用户——包括从计算机科学家到普通因特网用户的每一个人——都已经学会使用 URL 了。在备受笨拙语法（对新手来说）和顽固问题困扰的同时，人们已经学会了如何使用 URL，以及如何对付它们的一些缺陷。URL 有一些限制，但这并不是 Web 开发社区所面临的最紧迫的问题。

目前看来，在可预见的未来，因特网资源仍然会以 URL 来命名。它们无处不在，而且是 Web 的成功过程中一个非常重要的部分。其他命名方案想要取代 URL 还要一段时间。但是，URL 确实有其局限型，可能会出现新的标准（可能就是 URN），对这种标准进行部署会解决其中的某些问题。

2.7 更多信息

更多有关 URL 的信息，请参考以下资源。

- http://www.w3.org/Addressing/
  这是 W3C 有关 URI 和 URL 命名及寻址的 Web 页面。

- http://www.ietf.org/rfc/rfc1738
  RFC 1738，T. Berners-Lee、L. Masinter 和 M. McCahill 编写的“Uniform Resource
41 Locators（URL）”（“统一资源定位符”）。

- http://www.ietf.org/rfc/rfc2396.txt
  RFC 2396，T. Berners-Lee、R. Fielding 和 L. Masinter 编写的“Uniform Resource Identifiers（URI）：Generic Syntax”（“URI：通用语法”）。

- http://www.ietf.org/rfc/rfc2141.txt
  RFC 2141，R. Moats 编写的“URN Syntax”（“URN 语法”）。

- http://purl.oclc.org
  永久统一资源定位符的 Web 站点。

- http://www.ietf.org/rfc/rfc1808.txt
  RFC 1808，R. Fielding 编写的“Relative Uniform Resource Locators”（“相对统
42 一资源定位符”）。

第3章

HTTP报文

如果说 HTTP 是因特网的信使，那么 HTTP 报文就是它用来搬东西的包裹了。第 1 章说明了 HTTP 程序是怎样互相发送报文来完成工作的。本章则会介绍所有与 HTTP 报文有关的事情——如何创建报文，以及如何理解它们。通过阅读本章，就可以了解编写自己的 HTTP 应用程序所需掌握的大部分内容。具体来说，你会理解下列概念：

- 报文是如何流动的；
- HTTP 报文的三个组成部分（起始行、首部和实体的主体部分）；
- 请求和响应报文之间的区别；
- 请求报文支持的各种功能（方法）；
- 和响应报文一起返回的各种状态码；
- 各种各样的 HTTP 首部都是用来做什么的。

3.1 报文流

HTTP 报文是在 HTTP 应用程序之间发送的数据块。这些数据块以一些文本形式的元信息（meta-information）开头，这些信息描述了报文的内容及含义，后面跟着可选的数据部分。这些报文在客户端、服务器和代理之间流动。术语“流入”、“流出”、“上游”及“下游”都是用来描述报文方向的。

3.1.1 报文流入源端服务器

HTTP 使用术语流入（inbound）和流出（outbound）来描述事务处理（transaction）的方向。报文流入源端服务器，工作完成之后，会流回用户的 Agent 代理中（参见图 3-1）。

43

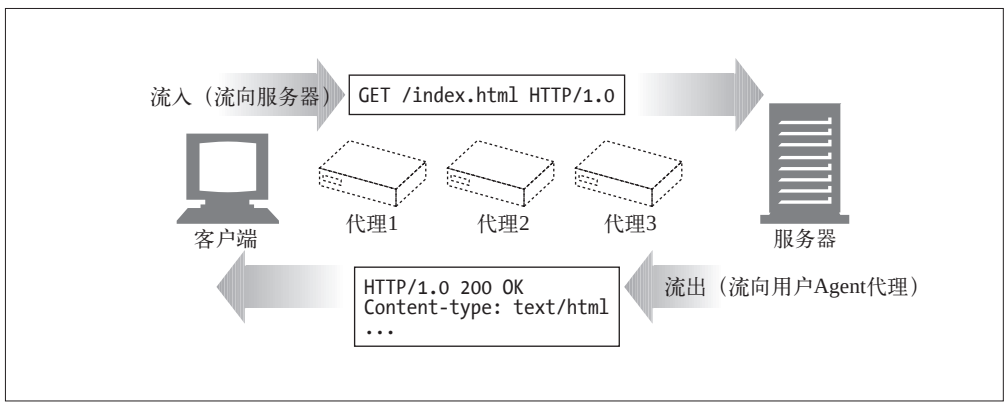


图 3-1 报文流入源端服务器并流回到客户端

3.1.2 报文向下游流动

HTTP 报文会像河水一样流动。不管是请求报文还是响应报文，所有报文都会向下游（downstream）流动（参见图 3-2）。所有报文的发送者都在接收者的上游（upstream）。在图 3-2 中，对请求报文来说，代理 1 位于代理 3 的上游，但对响应报文来说，它就位于代理 3 的下游[1]。

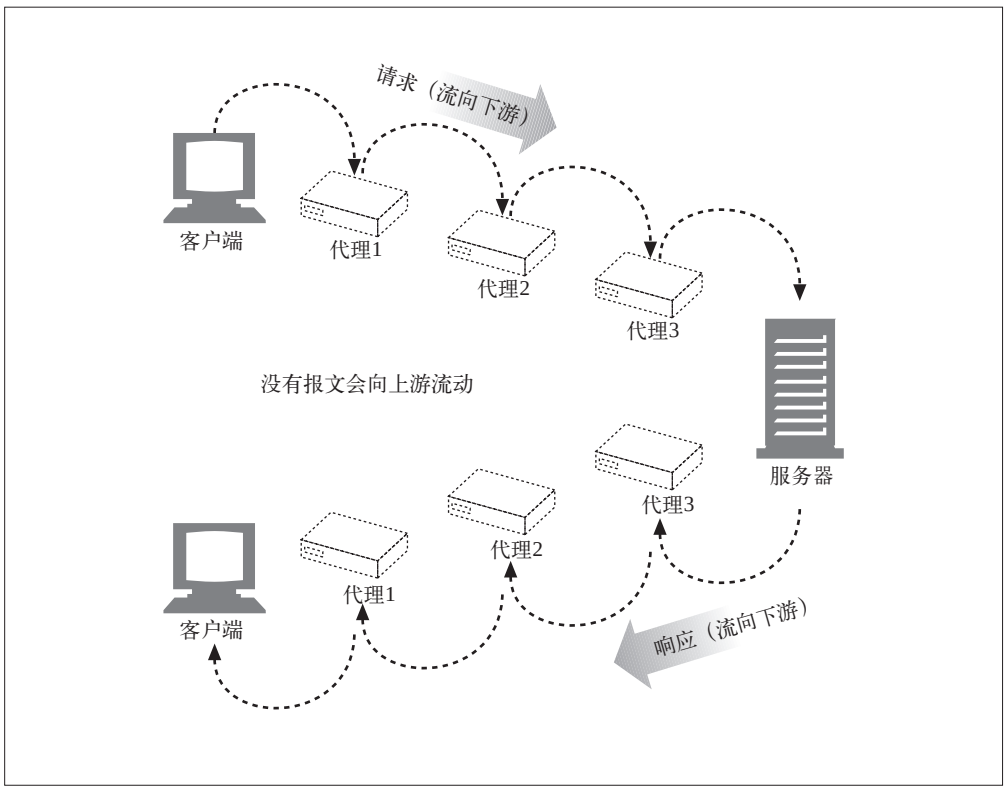


图 3-2　所有报文都向下游流动

3.2 报文的组成部分

HTTP 报文是简单的格式化数据块。看一下图 3-3 给出的例子。每条报文都包含一条来自客户端的请求，或者一条来自服务器的响应。它们由三个部分组成：对报文进行描述的起始行（start line）、包含属性的首部（header）块，以及可选的、包含数据的主体（body）部分。

注 1：术语“上游”和“下游”都只与发送者和接收者有关。我们无法区分报文是发送给源端服务器的还是发送给客户端的，因为两者都是下游节点。

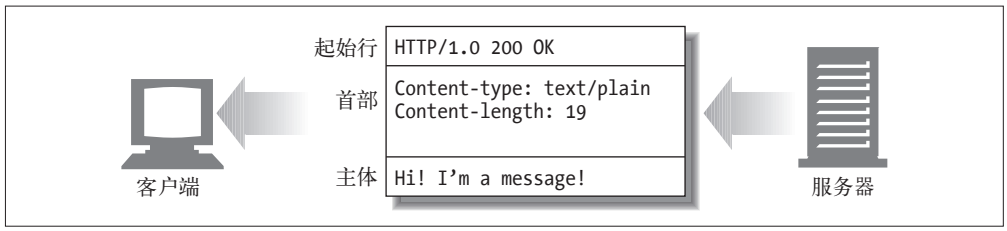


图 3-3　HTTP 报文的三个部分

起始行和首部就是由行分隔的 ASCII 文本。每行都以一个由两个字符组成的行终止序列作为结束，其中包括一个回车符（ASCII 码 13）和一个换行符（ASCII 码 10）。这个行终止序列可以写做 CRLF。需要指出的是，尽管 HTTP 规范中说明应该用 CRLF 来表示行终止，但稳健的应用程序也应该接受单个换行符作为行的终止。有些老的，或不完整的 HTTP 应用程序并不总是既发送回车符，又发送换行符。

实体的主体或报文的主体（或者就称为主体）是一个可选的数据块。与起始行和首部不同的是，主体中可以包含文本或二进制数据，也可以为空。

在图 3-3 的例子中，首部给出了一些与主体有关的信息。`Content-Type` 行说明了主体是什么——在这个例子中，就是纯文本文档。`Content-Length` 行说明了主体有多大，在这里就只有 19 个字节。

3.2.1　报文的语法

所有的 HTTP 报文都可以分为两类：*请求报文*（request message）和*响应报文*（response message）。请求报文会向 Web 服务器请求一个动作。响应报文会将请求的结果返回给客户端。请求和响应报文的基本报文结构相同。图 3-4 显示了获取一张 GIF 图片所需的请求和响应报文。

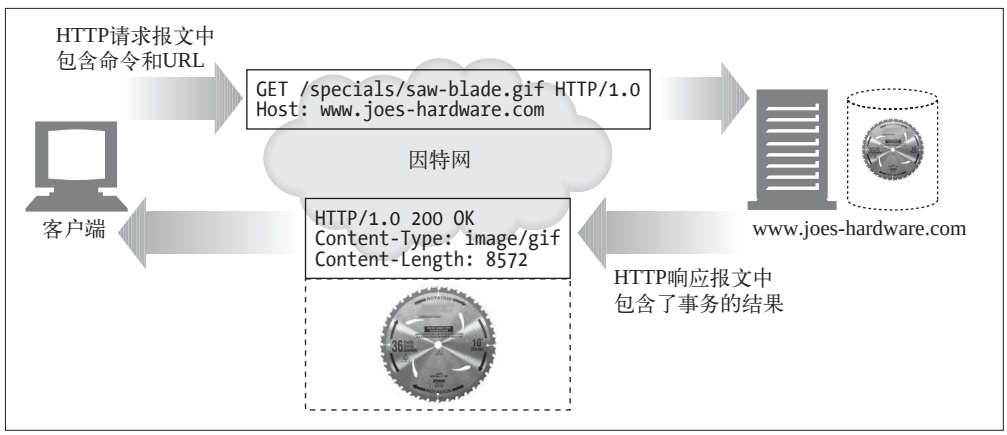


图 3-4　包含请求和响应报文的 HTTP 事务

这是请求报文的格式：

45

```
<method> <request-URL> <version>
<headers>

<entity-body>
```

这是响应报文的格式（注意，只有起始行的语法有所不同）：

```
<version> <status> <reason-phrase>
<headers>

<entity-body>
```

下面是对各部分的简要描述。

- *方法*（method）
  客户端希望服务器对资源执行的动作。是一个单独的词，比如 GET、HEAD 或 POST。本章稍后将详细介绍方法。

- *请求*URL（request-URL）
  命名了所请求资源，或者 URL *路径*组件的完整 URL。如果直接与服务器进行对话，只要 URL 的路径组件是资源的绝对路径，通常就不会有什么问题——服务器可以假定自己是 URL 的主机 / 端口。第 2 章详细地介绍了 URL 的语法。

- *版本*（version）
  报文所使用的 HTTP 版本，其格式看起来是这样的：

  ```
  HTTP/<major>.<minor>
  ```

  其中*主要版本号*（major）和*次要版本号*（minor）都是整数。本章稍后会详细说明 HTTP 的版本问题。

- *状态码*（status-code）
  这三位数字描述了请求过程中所发生的情况。每个状态码的第一位数字都用于描述状态的一般类别（“成功”、“出错”等）。本章稍后提供了 HTTP 规范定义的状态码及其含义的完整列表。

46

- *原因短语*（reason-phrase）
  数字状态码的可读版本，包含行终止序列之前的所有文本。本章稍后提供了 HTTP 规范定义的所有状态码的原因短语示例。原因短语只对人类有意义，因此，比如说，尽管响应行 HTTP/1.0 200 NOT OK 和 HTTP/1.0 200 OK 中原因短语的含义不同，但同样都会被当作成功指示处理。

- 首部（header）
  可以有零个或多个首部，每个首部都包含一个名字，后面跟着一个冒号（:），然后是一个可选的空格，接着是一个值，最后是一个 CRLF。首部是由一个空行（CRLF）结束的，表示了首部列表的结束和实体主体部分的开始。有些 HTTP 版本，比如 HTTP/1.1，要求有效的请求或响应报文中必须包含特定的首部。本章稍后会探讨各种 HTTP 首部。

- 实体的主体部分（entity-body）
  实体的主体部分包含一个由任意数据组成的数据块。并不是所有的报文都包含实体的主体部分，有时，报文只是以一个 CRLF 结束。第 15 章详述了实体。

图 3-5 展示了一些假想的请求和响应报文。

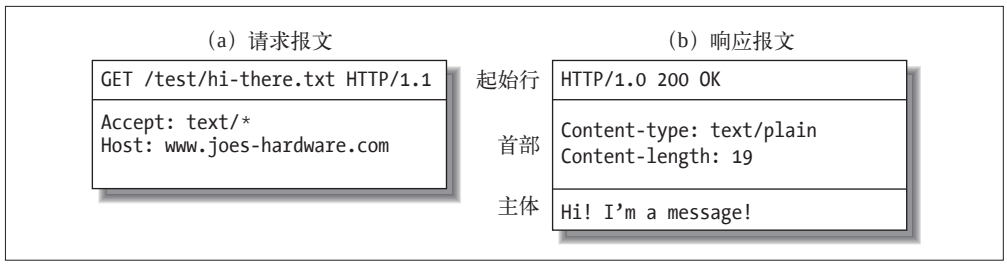


图 3-5　请求和响应报文示例

注意，一组 HTTP 首部总是应该以一个空行（仅有 CRLF）结束，甚至即使没有首部和实体的主体部分也应如此。但由于历史原因，很多客户端和服务器都在没有实体的主体部分时，（错误地）省略了最后的 CRLF。为了与这些流行但不符合规则的实现进行互通，客户端和服务器都应该接受那些没有最后那个 CRLF 的报文。

3.2.2　起始行

所有的 HTTP 报文都以一个起始行作为开始。请求报文的起始行说明了要做些什么。
47 响应报文的起始行说明发生了什么。

1. 请求行

请求报文请求服务器对资源进行一些操作。请求报文的起始行，或称为请求行，包含了一个方法和一个请求 URL，这个方法描述了服务器应该执行的操作，请求 URL 描述了要对哪个资源执行这个方法。请求行中还包含 HTTP 的版本，用来告知服务器，客户端使用的是哪种 HTTP。

所有这些字段都由空格符分隔。在图 3-5a 中，请求方法为 GET，请求 URL 为

/test/hi-there.txt，版本为HTTP/1.1。在HTTP/1.0之前，并不要求请求行中包含HTTP版本号。

2. 响应行

响应报文承载了状态信息和操作产生的所有结果数据，将其返回给客户端。响应报文的起始行，或称为*响应行*，包含了响应报文使用的HTTP版本、数字状态码，以及描述操作状态的文本形式的原因短语。

所有这些字段都由空格符进行分隔。在图3-5b中，HTTP版本为HTTP/1.0，状态码为200（表示成功），原因短语为OK，表示文档已经被成功返回了。在HTTP/1.0之前，并不要求在响应中包含响应行。

3. 方法

请求的起始行以方法作为开始，方法用来告知服务器要做些什么。比如，在行“GET /specials/saw-blade.gif HTTP/1.0”中，方法就是GET。

HTTP规范中定义了一组常用的请求方法。比如，GET方法负责从服务器获取一个文档，POST方法会向服务器发送需要处理的数据，OPTIONS方法用于确定Web服务器的一般功能，或者Web服务器处理特定资源的能力。

表3-1描述了7种这样的方法。注意，有些方法的请求报文中有主体，有些则是无主体的请求。

表3-1　常用的HTTP方法

方　法	描　述	是否包含主体
GET	从服务器获取一份文档	否
HEAD	只从服务器获取文档的首部	否
POST	向服务器发送需要处理的数据	是
PUT	将请求的主体部分存储在服务器上	是
TRACE	对可能经过代理服务器传送到服务器上去的报文进行追踪	否
OPTIONS	决定可以在服务器上执行哪些方法	否
DELETE	从服务器上删除一份文档	否

并不是所有服务器都实现了表3-1列出的所有7种方法。而且，由于HTTP设计得易于扩展，所以除了这些方法之外，其他服务器可能还会实现一些自己的请求方法。这些附加的方法是对HTTP规范的扩展，因此被称为*扩展方法*。

4. 状态码

方法是用来告诉服务器做什么事情的，状态码则用来告诉客户端，发生了什么事情。

状态码位于响应的起始行中。比如，在行 HTTP/1.0 200 OK 中，状态码就是 200。

客户端向一个 HTTP 服务器发送请求报文时，会发生很多事情。幸运的话，请求会成功完成。但你不会总是那么幸运的。服务器可能会告诉你无法找到所请求的资源，你没有访问资源的权限，或者资源被移到了其他地方。

状态码是在每条响应报文的起始行中返回的。会返回一个数字状态和一个可读的状态。数字码便于程序进行差错处理，而原因短语则更便于人们理解。

可以通过三位数字代码对不同状态码进行分类。200 到 299 之间的状态码表示成功。300 到 399 之间的代码表示资源已经被移走了。400 到 499 之间的代码表示客户端的请求出错了。500 到 599 之间的代码表示服务器出错了。

表 3-2 列出了状态码的分类。

表3-2　状态码分类

整体范围	已定义范围	分　类
100 ~ 199	100 ~ 101	信息提示
200 ~ 299	200 ~ 206	成功
300 ~ 399	300 ~ 305	重定向
400 ~ 499	400 ~ 415	客户端错误
500 ~ 599	500 ~ 505	服务器错误

当前的 HTTP 版本只为每类状态定义了几个代码。随着协议的发展，HTTP 规范中会正式地定义更多的状态码。如果收到了不认识的状态码，可能是有人将其作为当前协议的扩展定义的。可以根据其所处范围，将它作为那个类别中一个普通的成员来处理。

比如，如果收到了状态码 515（在表 3-2 所列 *5XX* 代码的已定义范围之外），就应该
49 认为这条响应指出了服务器的错误，这是 *5XX* 报文的通用类别。

表 3-3 列出了部分最常见的状态码。本章稍后会详细解释当前在用的所有 HTTP 状态码。

表3-3　常见状态码

状　态　码	原因短语	含　义
200	OK	成功。请求的所有数据都在响应主体中
401	Unauthorized（未授权）	需要输入用户名和密码
404	Not Found（未找到）	服务器无法找到所请求 URL 对应的资源

5. 原因短语

原因短语是响应起始行中的最后一个组件。它为状态码提供了文本形式的解释。比如，在行 HTTP/1.0 200 OK 中，OK 就是原因短语。

原因短语和状态码是成对出现的。原因短语是状态码的可读版本，应用程序开发者将其传送给用户，用以说明在请求期间发生了什么情况。

HTTP 规范并没有提供任何硬性规定，要求原因短语以何种形式出现。本章稍后列出了状态码和一些建议使用的原因短语。

6. 版本号

版本号会以 HTTP/*x.y* 的形式出现在请求和响应报文的起始行中。为 HTTP 应用程序提供了一种将自己所遵循的协议版本告知对方的方式。

使用版本号的目的是为使用 HTTP 的应用程序提供一种线索，以便互相了解对方的能力和报文格式。在与使用 HTTP 1.1 的应用程序进行通信的 HTTP 1.2 应用程序应该知道，它不能使用任何新的 1.2 特性，因为使用老版本协议的应用程序很可能无法实现这些特性。

版本号说明了应用程序支持的最高 HTTP 版本。但 HTTP/1.0 应用程序在解释包含 HTTP/1.1 的响应时，会认为这个响应是个 1.1 响应，而实际上这只是响应应用程序所使用的协议等级，在这些情况下，版本号会在应用程序之间造成误解[2]。

注意，版本号不会被当作小数来处理。版本中的每个数字（比如 HTTP/1.0 中的 1 和 0）都会被当作一个单独的数字来处理。因此，在比较 HTTP 版本时，每个数字都必须单独进行比较，以便确定哪个版本更高。比如，HTTP/2.22 就比 HTTP/2.3 的版本要高，因为 22 比 3 大。

50

3.2.3 首部

前一小节的重点是请求和响应报文的第一行（方法、状态码、原因短语和版本号）。跟在起始行后面的就是零个、一个或多个 HTTP 首部字段（参见图 3-5）。

HTTP 首部字段向请求和响应报文中添加了一些附加信息。本质上来说，它们只是一些名 / 值对的列表。比如，下面的首部行会向 `Content-Length` 首部字段赋值 19：

```
Content-length: 19
```

注 2：http://httpd.apache.org/docs-2.0/misc/known_client_problems.html 上有更多在 Apache 与客户端之间出现此问题的案例。

1. 首部分类

HTTP 规范定义了几种首部字段。应用程序也可以随意发明自己所用的首部。HTTP 首部可以分为以下几类。

- 通用首部
  既可以出现在请求报文中，也可以出现在响应报文中。
- 请求首部
  提供更多有关请求的信息。
- 响应首部
  提供更多有关响应的信息。
- 实体首部
  描述主体的长度和内容，或者资源自身。
- 扩展首部
  规范中没有定义的新首部。

每个 HTTP 首部都有一种简单的语法：名字后面跟着冒号（：），然后跟上可选的空格，再跟上字段值，最后是一个 CRLF。表 3-4 列出了一些常见的首部实例。

表3-4　常见的首部实例

首部实例	描　　述
`Date:Tue,3Oct 1997 02:16:03 GMT`	服务器产生响应的日期
`Content-length:15040`	实体的主体部分包含了 15 040 字节的数据
`Content-type:image/gif`	实体的主体部分是一个 GIF 图片
`Accept: image/gif, image/jpeg, text/html`	客户端可以接收 GIF 图片和 JPEG 图片以及 HTML

2. 首部延续行

将长的首部行分为多行可以提高可读性，多出来的每行前面至少要有一个空格或制表符（tab）。

51

例如：

```
HTTP/1.0 200 OK
Content-Type: image/gif
Content-Length: 8572
```

```
Server: Test Server
    Version 1.0
```

在这个例子中，响应报文里包含了一个 Server 首部，其值被划分成了多个延续行。该首部的完整值为 Test Server Version 1.0。

本章稍后将简要介绍所有的 HTTP 首部。附录 C 提供了所有首部更为详细的参考。

3.2.4 实体的主体部分

HTTP 报文的第三部分是可选的实体主体部分。实体的主体是 HTTP 报文的负荷。就是 HTTP 要传输的内容。

HTTP 报文可以承载很多类型的数字数据：图片、视频、HTML 文档、软件应用程序、信用卡事务、电子邮件等。

3.2.5 版本0.9的报文

HTTP 版本 0.9 是 HTTP 协议的早期版本。是当今 HTTP 所拥有的请求及响应报文的鼻祖，但其协议要简单得多（参见图 3-6）。

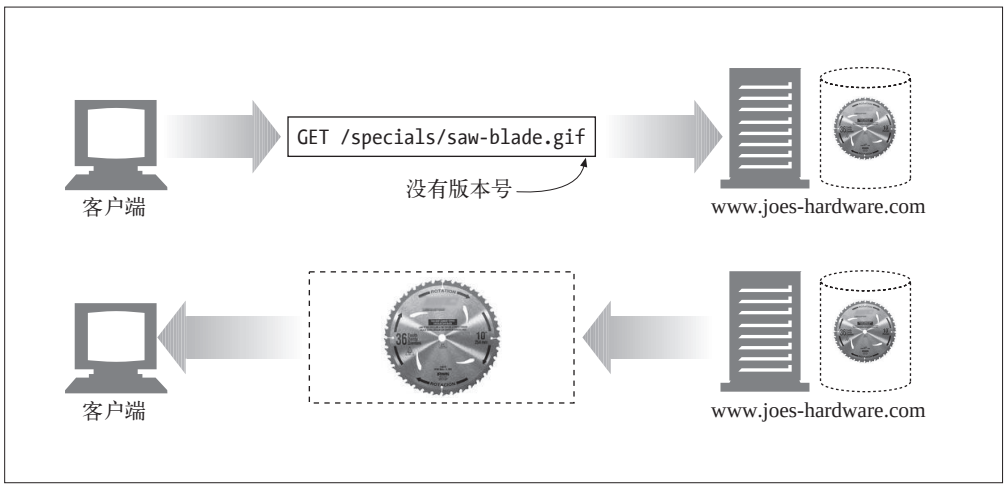


图 3-6 HTTP/0.9 事务

HTTP/0.9 报文也由请求和响应组成，但请求中只包含*方法*和*请求* URL，响应中只包含*实体*。它没有版本信息（它是第一个，而且是当时唯一的版本），没有状态码或原因短语，也没有首部。

但这种简单协议无法提供更多的灵活性，也无法实现本书中描述的大部分 HTTP 特性和应用。这里对其进行简要的描述，是因为仍然有一些客户端、服务器和其他应用程序在使用这个协议，应用程序的编写者应该清楚它的局限性。

3.3 方法

现在，我们对前面在表 3-1 中列出的一些基本 HTTP 方法进行更为深入的讨论。注意，并不是每个服务器都实现了所有的方法。如果一台服务器要与 HTTP 1.1 兼容，那么只要为其资源实现 GET 方法和 HEAD 方法就可以了。

即使服务器实现了所有这些方法，这些方法的使用很可能也是受限的。例如，支持 DELETE 方法或 PUT 方法（本节稍后介绍）的服务器可能并不希望任何人都能够删除或存储资源。这些限制通常都是在服务器的配置中进行设置的，因此会随着站点和服务器的不同而有所不同。

3.3.1 安全方法

HTTP 定义了一组被称为安全方法的方法。GET 方法和 HEAD 方法都被认为是安全的，这就意味着使用 GET 或 HEAD 方法的 HTTP 请求都不会产生什么动作。

不产生动作，在这里意味着 HTTP 请求不会在服务器上产生什么结果。例如，你在 Joe 的五金商店购物时，点击了“提交购买”按钮。点击按钮时会提交一个带有信用卡信息的 POST 请求（稍后讨论），那么在服务器上，就会为你执行一个动作。在这种情况下，为购买行为支付信用卡就是所执行的动作。

安全方法并不一定是什么动作都不执行的（实际上，这是由 Web 开发者决定的）。使用安全方法的目的就是当使用可能引发某一动作的不安全方法时，允许 HTTP 应用程序开发者通知用户。在 Joe 的五金商店的例子中，你的 Web 浏览器可能会弹出一条警告消息，说明你正在用不安全的方法发起请求，这样可能会在服务器上引发一些事件（比如用你的信用卡支付费用）。

3.3.2 GET

GET 是最常用的方法。通常用于请求服务器发送某个资源。HTTP/1.1 要求服务器实现此方法。图 3-7 显示了一个例子，在这个例子中，客户端用 GET 方法发起了一次 HTTP 请求。

53

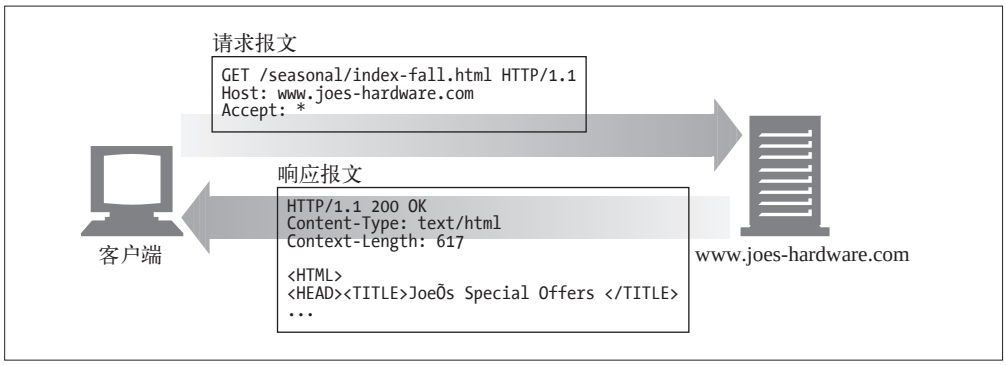


图 3-7　GET 示例

3.3.3　HEAD

HEAD 方法与 GET 方法的行为很类似，但服务器在响应中只返回首部。不会返回实体的主体部分。这就允许客户端在未获取实际资源的情况下，对资源的首部进行检查。使用 HEAD，可以：

- 在不获取资源的情况下了解资源的情况（比如，判断其类型）；
- 通过查看响应中的状态码，看看某个对象是否存在；
- 通过查看首部，测试资源是否被修改了。

服务器开发者必须确保返回的首部与 GET 请求所返回的首部完全相同。遵循 HTTP/1.1 规范，就必须实现 HEAD 方法。图 3-8 显示了实际的 HEAD 方法。

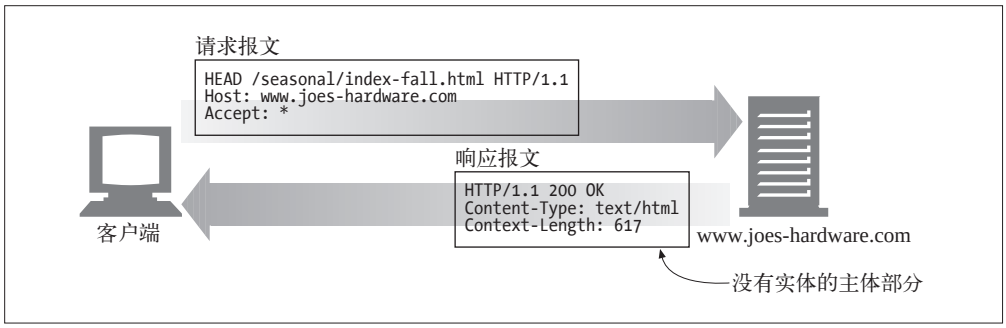


图 3-8　HEAD 示例

3.3.4　PUT

与 GET 从服务器读取文档相反，PUT 方法会向服务器写入文档。有些发布系统允许用户创建 Web 页面，并用 PUT 直接将其安装到 Web 服务器上去（参见图 3-9）。

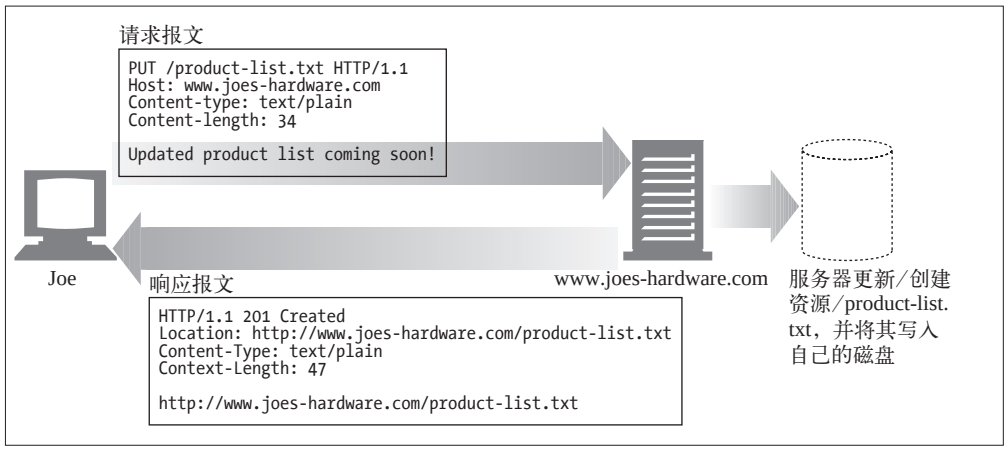


图 3-9　PUT 示例

PUT 方法的语义就是让服务器用请求的主体部分来创建一个由所请求的 URL 命名的新文档，或者，如果那个 URL 已经存在的话，就用这个主体来替代它。

因为 PUT 允许用户对内容进行修改，所以很多 Web 服务器都要求在执行 PUT 之前，用密码登录。在第 12 章中可以读到更多有关密码认证的内容。

3.3.5　POST

POST 方法起初是用来向服务器输入数据的[3]。实际上，通常会用它来支持 HTML 的表单。表单中填好的数据通常会被送给服务器，然后由服务器将其发送到它要去的地方（比如，送到一个服务器网关程序中，然后由这个程序对其进行处理）。图 3-10 显示了一个用 POST 方法发起 HTTP 请求——向服务器发送表单数据——的客户端。

3.3.6　TRACE

客户端发起一个请求时，这个请求可能要穿过防火墙、代理、网关或其他一些应用程序。每个中间节点都可能会修改原始的 HTTP 请求。TRACE 方法允许客户端在最终将请求发送给服务器时，看看它变成了什么样子。

TRACE 请求会在目的服务器端发起一个“环回”诊断。行程最后一站的服务器会弹回一条 TRACE 响应，并在响应主体中携带它收到的原始请求报文。这样客户端就可以查看在所有中间 HTTP 应用程序组成的请求 / 响应链上，原始报文是否，以及如何被毁坏或修改过（参见图 3-11）。

55

注 3：POST 用于向服务器发送数据。PUT 用于向服务器上的资源（例如文件）中存储数据。

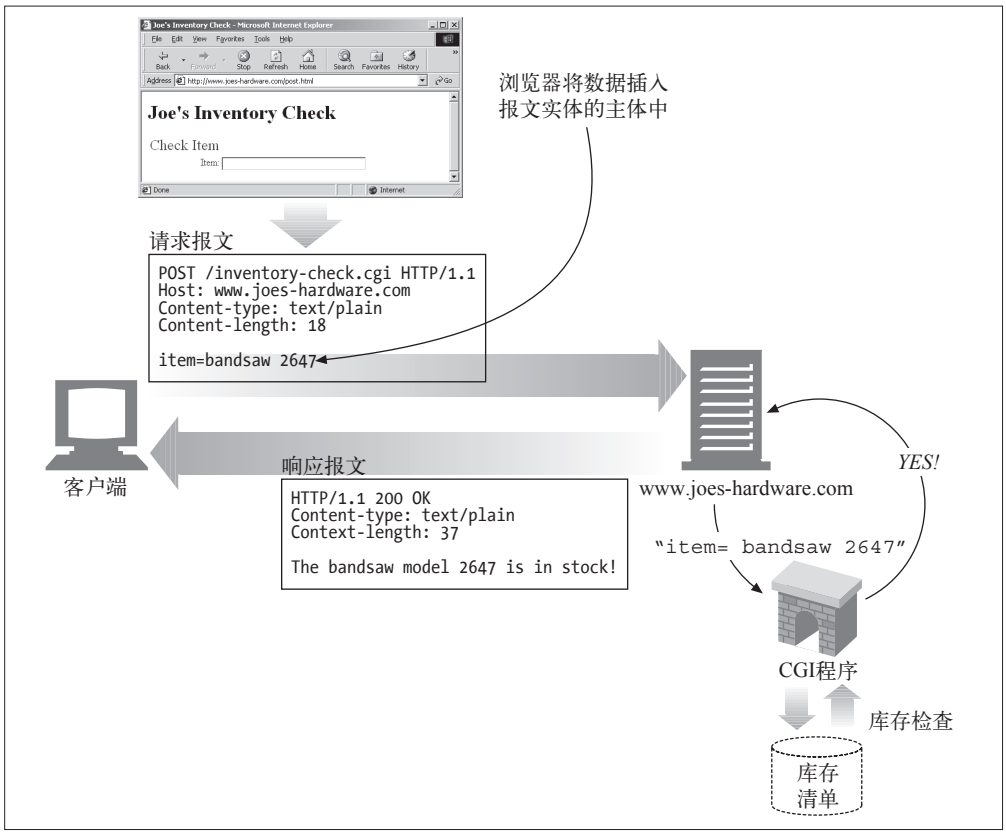


图 3-10　POST 示例

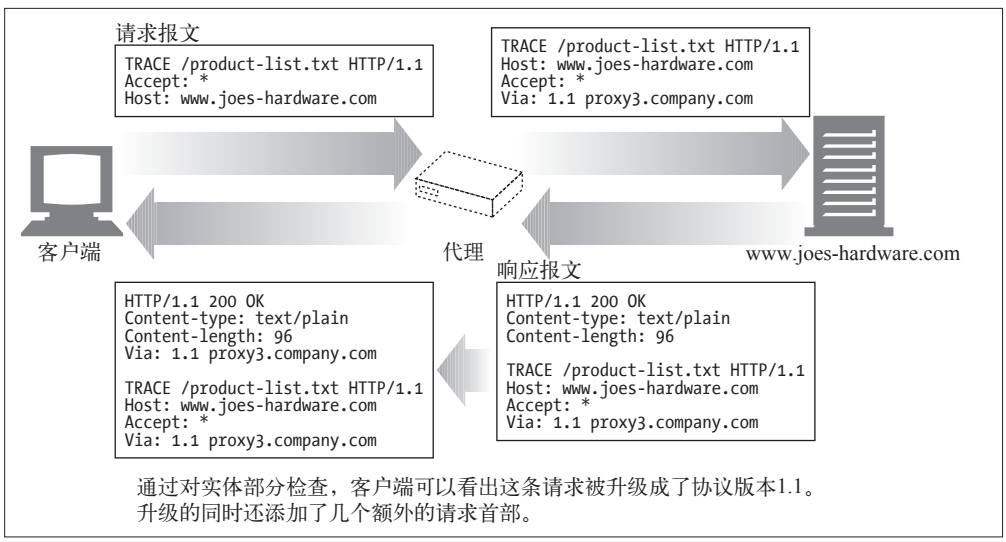


图 3-11　TRACE 示例

TRACE 方法主要用于诊断；也就是说，用于验证请求是否如愿穿过了请求 / 响应链。它也是一种很好的工具，可以用来查看代理和其他应用程序对用户请求所产生效果。

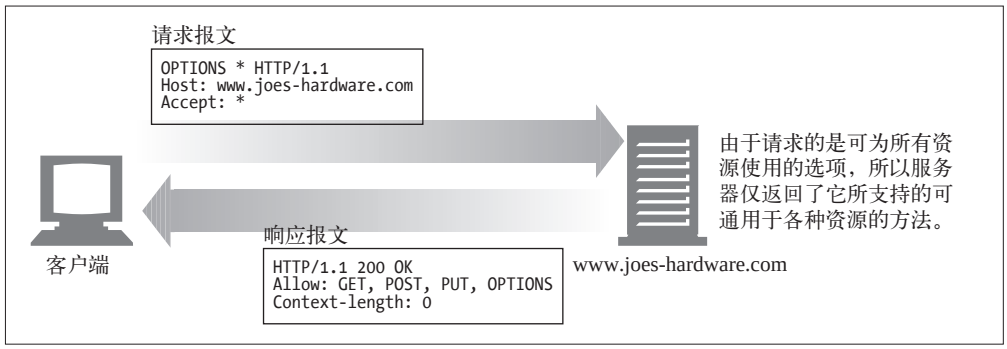
尽管 TRACE 可以很方便地用于诊断，但它确实也有缺点，它假定中间应用程序对各种不同类型请求（不同的方法——GET、HEAD、POST 等）的处理是相同的。很多 HTTP 应用程序会根据方法的不同做出不同的事情——比如，代理可能会将 POST 请求直接发送给服务器，而将 GET 请求发送给另一个 HTTP 应用程序（比如 Web 缓存）。TRACE 并不提供区分这些方法的机制。通常，中间应用程序会自行决
56 定对 TRACE 请求的处理方式。

TRACE 请求中不能带有实体的主体部分。TRACE 响应的实体主体部分包含了响应服务器收到的请求的精确副本。

3.3.7 OPTIONS

OPTIONS 方法请求 Web 服务器告知其支持的各种功能。可以询问服务器通常支持哪些方法，或者对某些特殊资源支持哪些方法。（有些服务器可能只支持对一些特殊类型的对象使用特定的操作）。

这为客户端应用程序提供了一种手段，使其不用实际访问那些资源就能判定访问各种资源的最优方式。图 3-12 显示了一个使用 OPTIONS 方法的请求。



57 图 3-12 OPTIONS 示例

3.3.8 DELETE

顾名思义，DELETE 方法所做的事情就是请服务器删除请求 URL 所指定的资源。但是，客户端应用程序无法保证删除操作一定会被执行。因为 HTTP 规范允许服务器在不通知客户端的情况下撤销请求。图 3-13 显示了一个 DELETE 方法实例。

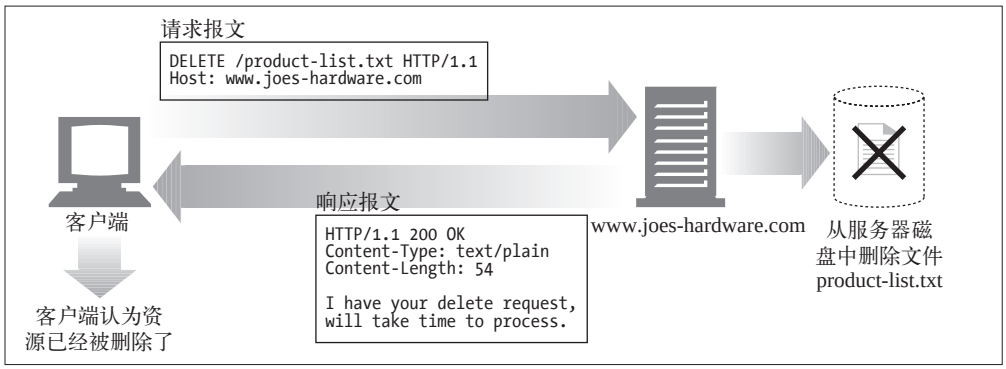


图 3-13　DELETE 示例

3.3.9　扩展方法

HTTP 被设计成字段可扩展的，这样新的特性就不会使老的软件失效了。扩展方法指的就是没有在 HTTP/1.1 规范中定义的方法。服务器会为它所管理的资源实现一些 HTTP 服务，这些方法为开发者提供了一种扩展这些 HTTP 服务能力的手段。表 3-5 列出了一些常见的扩展方法实例。这些方法就是 WebDAV HTTP 扩展（参见第 19 章）包含的所有方法，这些方法有助于通过 HTTP 将 Web 内容发布到 Web 服务器上去。

表3-5　Web发布扩展方法示例

方　法	描　述
LOCK	允许用户“锁定”资源——比如，可以在编辑某个资源的时候将其锁定，以防别人同时对其进行修改
MKCOL	允许用户创建资源
COPY	便于在服务器上复制资源
MOVE	在服务器上移动资源

并不是所有的扩展方法都是在正式规范中定义的，认识到这一点很重要。如果你定义了一个扩展方法，很可能大部分 HTTP 应用程序都无法理解。同样，你的 HTTP 应用程序也可能会遇到一些其他应用程序在用的，而它并不理解的扩展方法。

58

在这些情况下，最好对扩展方法宽容一些。如果能够在不破坏端到端行为的情况下将带有未知方法的报文传递给下游服务器，代理应尝试传递这些报文。如果可能破坏端到端行为则应以 501 Not Implemented（无法实现）状态码进行响应。最好按惯例“对所发送的内容要求严一点，对所接收的内容宽容一些”来处理扩展方法（以及一般的 HTTP 扩展）。

3.4 状态码

如前面的表 3-2 所示，HTTP 状态码被分成了五大类。本节对这五类 HTTP 状态码中的每一类都进行了总结。

状态码为客户端提供了一种理解事务处理结果的便捷方式。尽管并没有实际的规范对原因短语的确切文本进行说明，本节还是列出了一些原因短语示例。我们所列的是 HTTP/1.1 规范推荐使用的原因短语。

3.4.1 100 ~ 199——信息性状态码

HTTP/1.1 向协议中引入了信息性状态码。这些状态码相对较新，关于其复杂性和感知价值存在一些争论，而受到限制。表 3-6 列出了已定义的信息性状态码。

表3-6 信息性状态码及原因短语

状 态 码	原因短语	含 义
100	Continue	说明收到了请求的初始部分，请客户端继续。发送了这个状态码之后，服务器在收到请求之后必须进行响应。更多信息请参见附录 C 中的 Expect 首部介绍
101	Switching Protocols	说明服务器正在根据客户端的指定，将协议切换成 Update 首部所列的协议

100 Continue 状态码尤其让人糊涂。它的目的是对这样的情况进行优化：HTTP 客户端应用程序有一个实体的主体部分要发送给服务器，但希望在发送之前查看一下服务器是否会接受这个实体。这可能会给 HTTP 程序员带来一些困扰，因此在这里进行了比较详细（它如何与客户端、服务器和代理进行通信）的讨论。

1. 客户端与100 Continue

如果客户端在向服务器发送一个实体，并且愿意在发送实体之前等待 100 Continue 响应，那么，客户端就要发送一个携带了值为 100 Continue 的 Expect 请求首部（参见附录 C）。如果客户端没有发送实体，就不应该发送 100 Continue Expect 首部，因为这样会使服务器误以为客户端要发送一个实体。

59

从很多方面来看，100 Continue 都是一种优化。客户端应用程序只有在避免向服务器发送一个服务器无法处理或使用的大实体时，才应该使用 100 Continue。

由于起初对 100 Continue 状态存在一些困惑（而且以前有些实现在这里出过问题），因此发送了值为 100 Continue 的 Expect 首部的客户端不应该永远在那儿等待服务器发送 100 Continue 响应。超时一定时间之后，客户端应该直接将实体发送出去。

实际上，客户端程序的实现者也应该做好应对非预期 100 Continue 响应的准备（这很烦人，但确实如此）。有些出错的 HTTP 应用程序会不合时宜地发送这个状态码。

2. 服务器与100 Continue

如果服务器收到了一条带有值为 100 Continue 的 `Expect` 首部的请求，它会用 100 Continue 响应或一条错误码来进行响应（参见表 3-9）。服务器永远也不应该向没有发送 100 Continue 期望的客户端发送 100 Continue 状态码。但如前所述，有些出错的服务器可能会这么做。

如果出于某种原因，服务器在有机会发送 100 Continue 响应之前就收到了部分（或全部）的实体，就说明客户端已经决定继续发送数据了，这样，服务器就不需要发送这个状态码了。但服务器读完请求之后，还是应该为请求发送一个最终状态码（它可以跳过 100 Continue 状态）。

最后，如果服务器收到了带有 100 Continue 期望的请求，而且它决定在读取实体的主体部分之前（比如，因为出错而）结束请求，就不应该仅仅是发送一条响应并关闭连接，因为这样会妨碍客户端接收响应（参见 4.7.4 节）。

3. 代理与100 Continue

如果代理从客户端收到了一条带有 100 Continue 期望的请求，它需要做几件事情。如果代理知道下一跳服务器（在第 6 章中讨论）是 HTTP/1.1 兼容的，或者并不知道下一跳服务器与哪个版本兼容，它都应该将 `Expect` 首部放在请求中向下转发。如果它知道下一跳服务器只能与 HTTP/1.1 之前的版本兼容，就应该以 417 Expectation Failed 错误进行响应。[4]

如果代理决定代表与 HTTP/1.0 或之前版本兼容的客户端，在其请求中放入 `Expect` 首部和 100 Continue 值，那么，（如果它从服务器收到了 100 Continue 响应）它就不应该将 100 Continue 响应转发给客户端，因为客户端可能不知道该拿它怎么办。

代理维护一些有关下一跳服务器及其所支持的 HTTP 版本的状态信息（至少要维护那些最近收到过请求的服务器的相关状态）是有好处的，这样它们就可以更好地处理收到的那些带有 100 Continue 期望的请求了。

60

3.4.2 200 ~ 299——成功状态码

客户端发起请求时，这些请求通常都是成功的。服务器有一组用来表示成功的状态码，分别对应于不同类型的请求。表 3-7 列出了已定义的成功状态码。

注 4：还有一种合理的方法，是向客户端先返回 100 Continue，在向服务器转发请求时，删掉 `Expect` 首部。（译者注）

表3-7　成功状态码和原因短语

状态码	原因短语	含　义
200	OK	请求没问题，实体的主体部分包含了所请求的资源
201	Created	用于创建服务器对象的请求（比如，PUT）。响应的实体主体部分中应该包含各种引用了已创建的资源的URL，Location首部包含的则是最具体的引用。更多有关Location首部的信息参见表3-21。 服务器必须在发送这个状态码之前创建好对象
202	Accepted	请求已被接受，但服务器还未对其执行任何动作。不能保证服务器会完成这个请求；这只是意味着接受请求时，它看起来是有效的。 服务器应该在实体的主体部分包含对请求状态的描述，或许还应该有对请求完成时间的估计（或者包含一个指针，指向可以获取此信息的位置）
203	Non-Authoritative Information	实体首部（更多有关实体首部的信息参见3.5.4节）包含的信息不是来自于源端服务器，而是来自资源的一份副本。如果中间节点上有一份资源副本，但无法或者没有对它所发送的与资源有关的元信息（首部）进行验证，就会出现这种情况。 这种响应码并不是非用不可的；如果实体首部来自源端服务器，响应为200状态的应用程序就可以将其作为一种可选项使用
204	No Content	响应报文中包含若干首部和一个状态行，但没有实体的主体部分。主要用于在浏览器不转为显示新文档的情况下，对其进行更新（比如刷新一个表单页面）
205	Reset Content	另一个主要用于浏览器的代码。负责告知浏览器清除当前页面中的所有HTML表单元素
206	Partial Content	成功执行了一个部分或Range（范围）请求。稍后我们会看到，客户端可以通过一些特殊的首部来获取部分或某个范围内的文档——这个状态码就说明范围请求成功了。更多有关Range首部的内容参见15.9节。 206响应中必须包含Content-Range、Date以及ETag或Content-Location首部

3.4.3　300 ~ 399——重定向状态码

重定向状态码要么告知客户端使用替代位置来访问他们所感兴趣的资源，要么就提供一个替代的响应而不是资源的内容。如果资源已被移动，可发送一个重定向状态码和一个可选的Location首部来告知客户端资源已被移走，以及现在可以在哪里找到它（参见图3-14）。这样，浏览器就可以在不打扰使用者的情况下，透明地转入新的位置了。

61

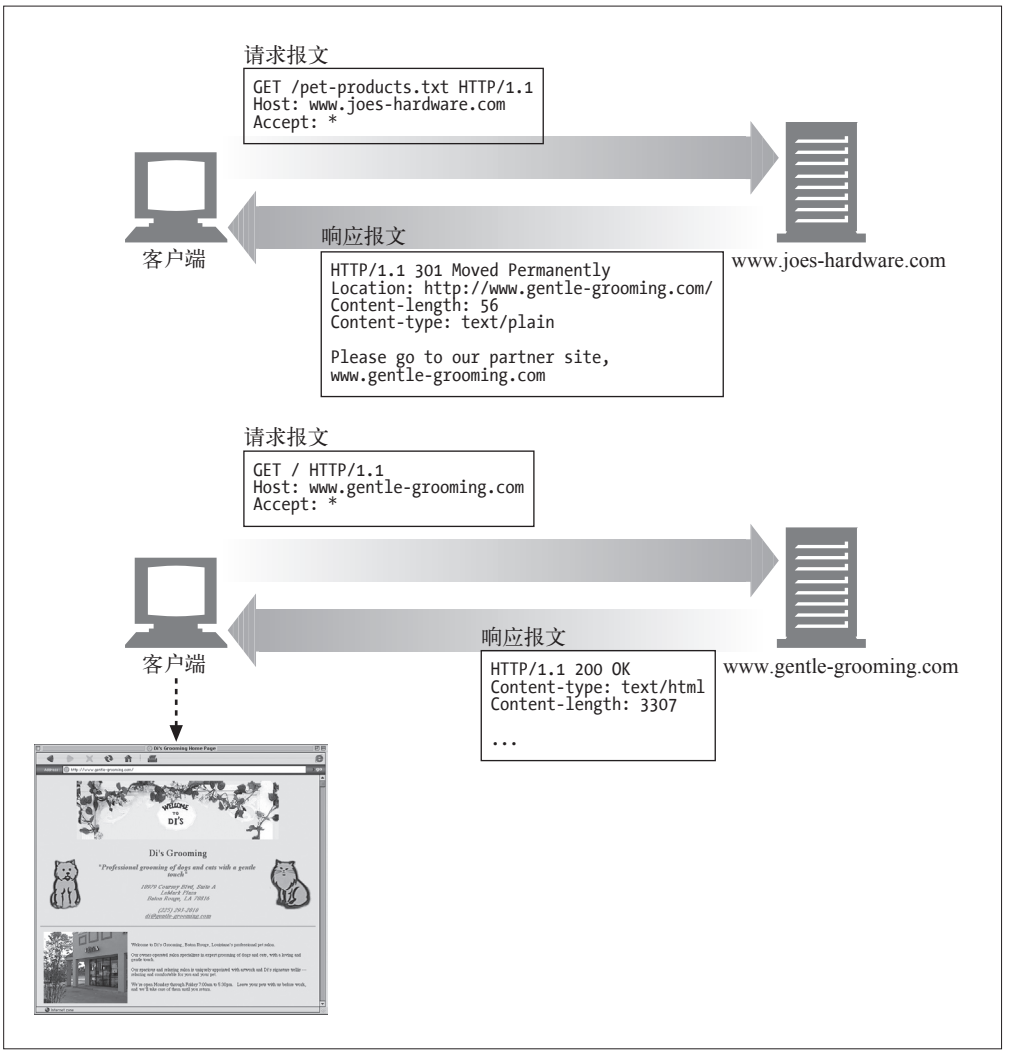


图 3-14　将请求重定向到新的位置

可以通过某些重定向状态码对资源的应用程序本地副本与源端服务器上的资源进行验证。比如，HTTP 应用程序可以查看其资源的本地副本是否仍然是最新的，或者在源端服务器上资源是否被修改过。图 3-15 显示了一个这样的例子。客户端发送了一个特殊的 `If-Modified-Since` 首部，说明只读取 1997 年 10 月之后修改过的文档。这个日期之后，此文档并未被修改过，因此，服务器回送了一个 304 状态码，而不是文档的内容。

62

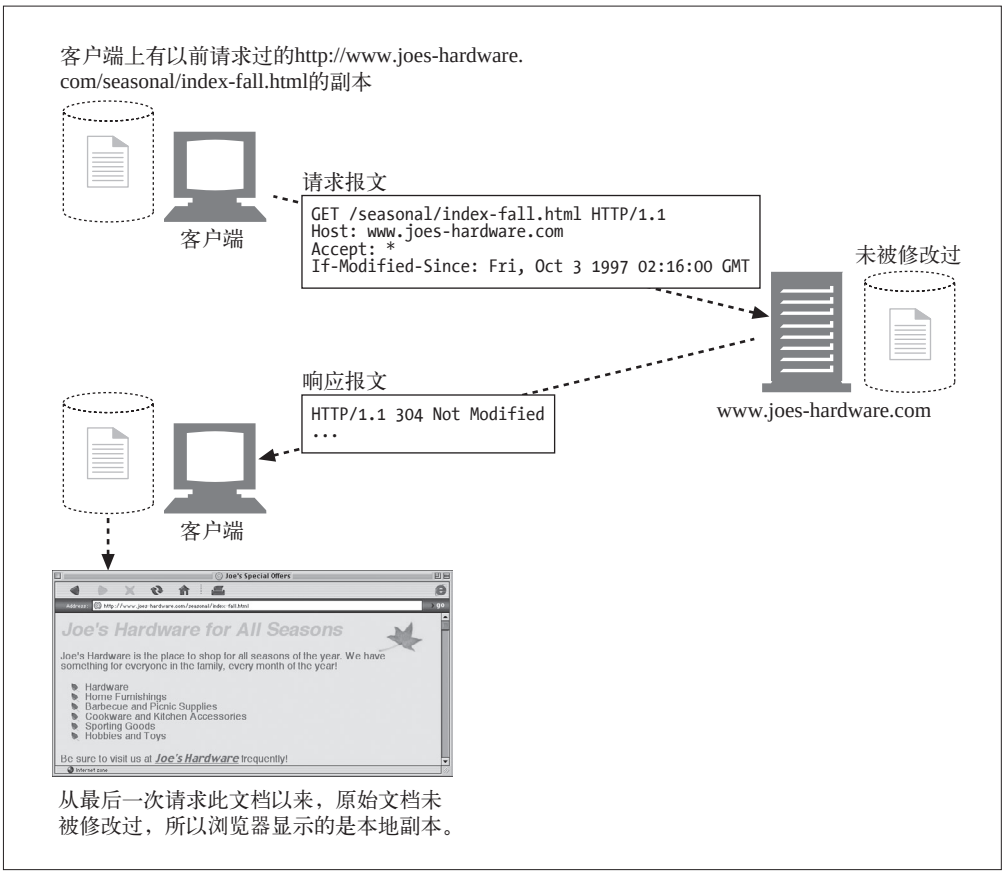


图 3-15　重定向为使用本地副本的请求

总之，在对那些包含了重定向状态码的非 HEAD 请求进行响应时，最好要包含一个实体，并在实体中包含描述信息和指向（多个）重定向 URL 的链接——参见图 3-14 的第一个响应报文。表 3-8 列出了已定义的重定向状态码。

表3-8　重定向状态码与原因短语

状态码	原因短语	含　义
300	Multiple Choices	客户端请求一个实际指向多个资源的 URL 时会返回这个状态码，比如服务器上有某个 HTML 文档的英语和法语版本。返回这个代码时会带有一个选项列表；这样用户就可以选择他希望使用的那一项了。有多个版本可用时，客户端需要沟通解决，更多与此有关的信息请参见第 17 章。服务器可以在 Location 首部包含首选 URL
301	Moved Permanently	在请求的 URL 已被移除时使用。响应的 Location 首部中应该包含资源现在所处的 URL

（续）

状态码	原因短语	含　义
302	Found	与 301 状态码类似；但是，客户端应该使用 Location 首部给出的 URL 来临时定位资源。将来的请求仍应使用老的 URL
303	See Other	告知客户端应该用另一个 URL 来获取资源。新的 URL 位于响应报文的 Location 首部。其主要目的是允许 POST 请求的响应将客户端定向到某个资源上去
304	Not Modified	客户端可以通过所包含的请求首部，使其请求变成有条件的。更多有关条件首部的内容请参见第 3 章。如果客户端发起了一个条件 GET 请求，而最近资源未被修改的话，就可以用这个状态码来说明资源未被修改。带有这个状态码的响应不应该包含实体的主体部分
305	Use Proxy	用来说明必须通过一个代理来访问资源；代理的位置由 Location 首部给出。很重要的一点是，客户端是相对某个特定资源来解析这条响应的，不能假定所有请求，甚至所有对持有所请求资源的服务器的请求都通过这个代理进行。如果客户端错误地让代理介入了某条请求，可能会引发破坏性的行为，而且会造成安全漏洞
306	（未使用）	当前未使用
307	Temporary Redirect	与 301 状态码类似；但客户端应该使用 Location 首部给出的 URL 来临时定位资源。将来的请求应该使用老的 URL

从表 3-8 中，你可能已经注意到 302、303 和 307 状态码之间存在一些交叉。这些状态码的用法有着细微的差别，大部分差别都源于 HTTP/1.0 和 HTTP/1.1 应用程序对这些状态码处理方式的不同。

当 HTTP/1.0 客户端发起一个 POST 请求，并在响应中收到 302 重定向状态码时，它会接受 Location 首部的重定向 URL，并向那个 URL 发起一个 GET 请求（而不会像原始请求中那样发起 POST 请求）。

HTTP/1.0 服务器希望 HTTP/1.0 客户端这么做——如果 HTTP/1.0 服务器收到来自 HTTP/1.0 客户端的 POST 请求之后发送了 302 状态码，服务器就期望客户端能够接受重定向 URL，并向重定向的 URL 发送一个 GET 请求。

问题出在 HTTP/1.1。HTTP/1.1 规范使用 303 状态码来实现同样的行为（服务器发送 303 状态码来重定向客户端的 POST 请求，在它后面跟上一个 GET 请求）。

为了避开这个问题，HTTP/1.1 规范指出，对于 HTTP/1.1 客户端，用 307 状态码取代 302 状态码来进行临时重定向。这样服务器就可以将 302 状态码保留起来，为 HTTP/1.0 客户端使用了。

这样一来，服务器要选择适当的重定向状态码放入重定向响应中发送，就需要查看客户端的 HTTP 版本了。

64

3.4.4 400 ~ 499——客户端错误状态码

有时客户端会发送一些服务器无法处理的东西，比如格式错误的请求报文，或者最常见的是，请求一个不存在的 URL。

浏览网页时，我们都看到过臭名昭著的 404 Not Found 错误码——这只是服务器在告诉我们，它对我们请求的资源一无所知。

很多客户端错误都是由浏览器来处理的，甚至不会打扰到你。只有少量错误，比如 404，还是会穿过浏览器来到用户面前。表 3-9 显示了各种客户端的错误状态码。

表3-9 客户端错误状态码及原因短语

状态码	原因短语	含 义
400	Bad Request	用于告知客户端它发送了一个错误的请求
401	Unauthorized	与适当的首部一同返回，在这些首部中请求客户端在获取对资源的访问权之前，对自己进行认证。更多有关认证的内容请参见 12.1 节
402	Payment Required	现在这个状态码还未使用，但已经被保留，以作未来之用
403	Forbidden	用于说明请求被服务器拒绝了。如果服务器想说明为什么拒绝请求，可以包含实体的主体部分来对原因进行描述。但这个状态码通常是在服务器不想说明拒绝原因的时候使用的
404	Not Found	用于说明服务器无法找到所请求的 URL。通常会包含一个实体，以便客户端应用程序显示给用户看
405	Method Not Allowed	发起的请求中带有所请求的 URL 不支持的方法时，使用此状态码。应该在响应中包含 `Allow` 首部，以告知客户端对所请求的资源可以使用哪些方法。更多有关 `Allow` 首部的信息请参见 3.5.4 节
406	Not Acceptable	客户端可以指定参数来说明它们愿意接收什么类型的实体。服务器没有与客户端可接受的 URL 相匹配的资源时，使用此代码。通常，服务器会包含一些首部，以便客户端弄清楚为什么请求无法满足。更多信息请参见第 17 章
407	Proxy Authentication Required	与 401 状态码类似，但用于要求对资源进行认证的代理服务器
408	Request Timeout	如果客户端完成请求所花的时间太长，服务器可以回送此状态码，并关闭连接。超时时长随服务器的不同有所不同，但通常对所有的合法请求来说，都是够长的
409	Conflict	用于说明请求可能在资源上引发的一些冲突。服务器担心请求会引发冲突时，可以发送此状态码。响应中应该包含描述冲突的主体

（续）

状态码	原因短语	含　义
410	Gone	与 404 类似，只是服务器曾经拥有过此资源。主要用于 Web 站点的维护，这样服务器的管理者就可以在资源被移除的情况下通知客户端了
411	Length Required	服务器要求在请求报文中包含 Content-Length 首部时使用。更多有关 Content-Length 首部的信息请参见 3.5.4 节
412	Precondition Failed	客户端发起了条件请求，且其中一个条件失败了的时候使用。客户端包含了 Expect 首部时发起的就是条件请求。更多有关 Expect 首部的内容请参见附录 C 中 Expect 部分
413	Request Entity Too Large	客户端发送的实体主体部分比服务器能够或者希望处理的要大时，使用此状态码
414	Request URI Too Long	客户端所发请求中的请求 URL 比服务器能够或者希望处理的要长时，使用此状态码
415	Unsupported Media Type	服务器无法理解或无法支持客户端所发实体的内容类型时，使用此状态码
416	Requested Range Not Satisfiable	请求报文所请求的是指定资源的某个范围，而此范围无效或无法满足时，使用此状态码
417	Expectation Failed	请求的 Expect 请求首部包含了一个期望，但服务器无法满足此期望时，使用此状态码。更多有关 Expect 首部的内容请参见附录 C 中 Expect 部分 如果代理或其他中间应用程序有确切证据说明源端服务器会为某请求产生一个失败的期望，就可以发送这个响应状态码

3.4.5　500 ~ 599——服务器错误状态码

有时客户端发送了一条有效请求，服务器自身却出错了。这可能是客户端碰上了服务器的缺陷，或者服务器上的子元素，比如某个网关资源，出了错。

代理尝试着代表客户端与服务器进行交流时，经常会出现问题。代理会发布 5XX 服务器错误状态码来描述所遇到的问题（参见第 6 章）。表 3-10 列出了已定义的服务器错误状态码。

表3-10　服务器错误状态码及原因短语

状态码	原因短语	含　义
500	Internal Server Error	服务器遇到一个妨碍它为请求提供服务的错误时，使用此状态码
501	Not Implemented	客户端发起的请求超出服务器的能力范围（比如，使用了服务器不支持的请求方法）时，使用此状态码

（续）

状态码	原因短语	含　义
502	Bad Gateway	作为代理或网关使用的服务器从请求响应链的下一条链路上收到了一条伪响应（比如，它无法连接到其父网关）时，使用此状态码
503	Service Unavailable	用来说明服务器现在无法为请求提供服务，但将来可以。如果服务器知道什么时候资源会变为可用的，可以在响应中包含一个 Retry-After 首部。更多有关 Retry-After 首部的信息请参见 3.5.3 节
504	Gateway Timeout	与状态码 408 类似，只是这里的响应来自一个网关或代理，它们在等待另一服务器对其请求进行响应时超时了
505	HTTP Version Not Supported	服务器收到的请求使用了它无法或不愿支持的协议版本时，使用此状态码。有些服务器应用程序会选择不支持协议的早期版本

3.5 首部

首部和方法配合工作，共同决定了客户端和服务器能做什么事情。本节快速介绍了使用标准 HTTP 首部及一些没有在 HTTP/1.1 规范（RFC 2616）中明确定义的首部的目的。附录 C 对所有这些首部进行了更详细的总结。

在请求和响应报文中都可以用首部来提供信息，有些首部是某种报文专用的，有些首部则更通用一些。可以将首部分为五个主要的类型。

- 通用首部
  这些是客户端和服务器都可以使用的通用首部。可以在客户端、服务器和其他应用程序之间提供一些非常有用的通用功能。比如，Date 首部就是一个通用首部，每一端都可以用它来说明构建报文的时间和日期：

```
Date: Tue, 3 Oct 1974 02:16:00 GMT
```

- 请求首部
  从名字中就可以看出，请求首部是请求报文特有的。它们为服务器提供了一些额外信息，比如客户端希望接收什么类型的数据。例如，下面的 Accept 首部就用来告知服务器客户端会接受与其请求相符的任意媒体类型：

```
Accept: */*
```

- 响应首部
  响应报文有自己的首部集，以便为客户端提供信息（比如，客户端在与哪种类型的服务器进行交互）。例如，下列 Server 首部就用来告知客户端它在与一个版

本 1.0 的 Tiki-Hut 服务器进行交互：

```
Server: Tiki-Hut/1.0
```

- 实体首部
  实体首部指的是用于应对实体主体部分的首部。比如，可以用实体首部来说明实体主体部分的数据类型。例如，可以通过下列 Content-Type 首部告知应用程序，数据是以 iso-latin-1 字符集表示的 HTML 文档：

```
Content-Type: text/html; charset=iso-latin-1
```


- 扩展首部
  扩展首部是非标准的首部，由应用程序开发者创建，但还未添加到已批准的 HTTP 规范中去。即使不知道这些扩展首部的含义，HTTP 程序也要接受它们并对其进行转发。

3.5.1 通用首部

有些首部提供了与报文相关的最基本的信息，它们被称为通用首部。它们像和事佬儿一样，不论报文是何类型，都为其提供一些有用信息。

例如，不管是构建请求报文还是响应报文，创建报文的日期和时间都是同一个意思，因此提供这类信息的首部对这两种类型的报文来说也是通用的。表 3-11 列出了通用的信息性首部。

表3-11 通用的信息性首部

首　部	描　述
Connection	允许客户端和服务器指定与请求 / 响应连接有关的选项
Date[5]	提供日期和时间标志，说明报文是什么时间创建的
MIME-Version	给出了发送端使用的 MIME 版本
Trailer	如果报文采用了分块传输编码（chunked transfer encoding）方式，就可以用这个首部列出位于报文拖挂（trailer）部分的首部集合[6]
Transfer-Encoding	告知接收端为了保证报文的可靠传输，对报文采用了什么编码方式
Update	给出了发送端可能想要“升级”使用的新版本或协议
Via	显示了报文经过的中间节点（代理、网关）

注 5：Date 中列出了 Date 首部可接受的日期格式。
注 6：15.6.3 节详细探讨了分块传输编码。

通用缓存首部

HTTP/1.0 引入了第一个允许 HTTP 应用程序缓存对象本地副本的首部，这样就不需要总是直接从源端服务器获取了。最新的 HTTP 版本有非常丰富的缓存参数集。第 7 章深入讨论了缓存。表 3-12 列出了基本的缓存首部。

表3-12　通用缓存首部

首　部	描　述
Cache-Control	用于随报文传送缓存指示
Pragma[7]	另一种随报文传送指示的方式，但并不专用于缓存

3.5.2　请求首部

请求首部是只在请求报文中有意义的首部。用于说明是谁或什么在发送请求、请求源自何处，或者客户端的喜好及能力。服务器可以根据请求首部给出的客户端信息，试着为客户端提供更好的响应。表 3-13 列出了请求的信息性首部。

表3-13　请求的信息性首部

首　部	描　述
Client-IP[8]	提供了运行客户端的机器的 IP 地址
From	提供了客户端用户的 E-mail 地址[9]
Host	给出了接收请求的服务器的主机名和端口号
Referer	提供了包含当前请求 URI 的文档的 URL
UA-Color	提供了与客户端显示器的显示颜色有关的信息
UA-CPU[10]	给出了客户端 CPU 的类型或制造商
UA-Disp	提供了与客户端显示器（屏幕）能力有关的信息
UA-OS	给出了运行在客户端机器上的操作系统名称及版本
UA-Pixels	提供了客户端显示器的像素信息
User-Agent	将发起请求的应用程序名称告知服务器

1. Accept首部

Accept 首部为客户端提供了一种将其喜好和能力告知服务器的方式，包括它们想要什么，可以使用什么，以及最重要的，它们不想要什么。这样，服务器就可以根

注 7：从技术角度来看，Pragma 是一种请求首部。从未被指定用于响应首部。由于经常被错误地用于响应首部，很多客户端和代理都会将 Pragma 解释为响应首部，但其确切语义并未得到很好地定义。任何情况下 Cache-Control 的使用都优于 Pragma。

注 8：RFC 2616 没有定义 Client-IP 和 UA-* 首部，但很多 HTTP 客户端应用程序都实现了这两个首部。

注 9：使用 RFC 822 E-mail 地址格式。

注 10：尽管有些客户端实现了 UA-* 首部，但我们认为 UA-* 首部是有副作用的。不应该将内容，尤其是 HTML，局限于特定的客户端配置。

据这些额外信息，对要发送的内容做出更明智的决定。Accept 首部会使连接的两端都受益。客户端会得到它们想要的内容，服务器则不会浪费其时间和带宽来发送客户端无法使用的东西。表 3-14 列出了各种 Accept 首部。

表3-14　Accept首部

首　部	描　述
Accept	告诉服务器能够发送哪些媒体类型
Accept-Charset	告诉服务器能够发送哪些字符集
Accept-Encoding	告诉服务器能够发送哪些编码方式
Accept-Language	告诉服务器能够发送哪些语言
TE[11]	告诉服务器可以使用哪些扩展传输编码

2. 条件请求首部

有时客户端希望为请求加上某些限制。比如，如果客户端已经有了一份文档副本，就希望只在服务器上的文档与客户端拥有的副本有所区别时，才请求服务器传输文档。通过条件请求首部，客户端就可以为请求加上这种限制，要求服务器在对请求进行响应之前，确保某个条件为真。表 3-15 列出了各种条件请求首部。

表3-15　条件请求首部

首　部	描　述
Expect	允许客户端列出某请求所要求的服务器行为
If-Match	如果实体标记与文档当前的实体标记相匹配，就获取这份文档[12]
If-Modified-Since	除非在某个指定的日期之后资源被修改过，否则就限制这个请求
If-None-Match	如果提供的实体标记与当前文档的实体标记不相符，就获取文档
If-Range	允许对文档的某个范围进行条件请求
If-Unmodified-Since	除非在某个指定日期之后资源没有被修改过，否则就限制这个请求
Range	如果服务器支持范围请求，就请求资源的指定范围[13]

3. 安全请求首部

HTTP 本身就支持一种简单的机制，可以对请求进行质询 / 响应认证。这种机制要求客户端在获取特定的资源之前，先对自身进行认证，这样就可以使事务稍微安全一些。我们会在第 14 章讨论这种质询 / 响应机制，同时还会对在 HTTP 之上实现的其他安全机制进行讨论。表 3-16 列出了一些安全请求首部。

注 11：更多有关 TE 首部的内容请参见 15.6.2 节。

注 12：更多有关实体标记的内容请参见第 7 章。标记本质上就是某版本资源的标识符。

注 13：更多有关 Range 首部的内容请参见 15.9 节。

表3-16　安全请求首部

首　部	描　述
Authorization	包含了客户端提供给服务器，以便对其自身进行认证的数据
Cookie	客户端用它向服务器传送一个令牌——它并不是真正的安全首部，但确实隐含了安全功能[14]
Cookie2	用来说明请求端支持的 cookie 版本，参见 11.6.7 节

4. 代理请求首部

随着因特网上代理的普遍应用，人们定义了几个首部来协助其更好地工作。第 6 章对这些首部进行了详细的讨论。表 3-17 列出了一些代理请求首部。

表3-17　代理请求首部

首　部	描　述
Max-Forward	在通往源端服务器的路径上，将请求转发给其他代理或网关的最大次数——与 TRACE 方法一同使用[15]
Proxy-Authorization	与 Authorization 首部相同，但这个首部是在与代理进行认证时使用的
Proxy-Connection	与 Connection 首部相同，但这个首部是在与代理建立连接时使用的

3.5.3　响应首部

响应报文有自己的响应首部集。响应首部为客户端提供了一些额外信息，比如谁在发送响应、响应者的功能，甚至与响应相关的一些特殊指令。这些首部有助于客户端处理响应，并在将来发起更好的请求。表 3-18 列出了一些响应的信息性首部。

表3-18　响应的信息性首部

首　部	描　述
Age	（从最初创建开始）响应持续时间[16]
Public[17]	服务器为其资源支持的请求方法列表
Retry-After	如果资源不可用的话，在此日期或时间重试
Server	服务器应用程序软件的名称和版本
Title[18]	对 HTML 文档来说，就是 HTML 文档的源端给出的标题
Warning	比原因短语中更详细一些的警告报文

注 14：RFC 2616 并没有定义 Cookie 首部，在第 11 章详细讨论了该首部。

注 15：参见 6.6.2 节。

注 16：暗示响应是通过中间节点，很可能是从代理的缓存传送过来的。

注 17：Public 首部是在 RFC 2068 中定义的，但在最新的 HTTP 定义（RFC 2616）中并没有出现。

注 18：RFC 2616 并没有定义 Title 首部。请参见原始的 HTTP/1.0 草案定义（http://www.w3.org/Protocols/HTTP/HTTP2.html）。

1. 协商首部

如果资源有多种表示方法——比如，如果服务器上有某文档的法语和德语译稿，HTTP/1.1 可以为服务器和客户端提供对资源进行协商的能力。第 17 章详细讨论了协商。这里列出了几个首部，服务器可以用它们来传递与可协商资源有关的信息。表 3-19 列出了协商首部。

表3-19　协商首部

首　部	描　述
Accept-Ranges	对此资源来说，服务器可接受的范围类型
Vary	服务器查看的其他首部的列表，可能会使响应发生变化；也就是说，这是一个首部列表，服务器会根据这些首部的内容挑选出最适合的资源版本发送给客户端

2. 安全响应首部

我们已经看到过安全请求首部了，本质上这里说的就是 HTTP 的质询 / 响应认证机制的响应侧。我们会在第 14 章对安全问题进行详细的讨论。现在这里介绍的是一些基本的质询首部。表 3-20 列出了安全响应首部。

表3-20　安全响应首部

首　部	描　述
Proxy-Authenticate	来自代理的对客户端的质询列表
Set-Cookie	不是真正的安全首部，但隐含有安全功能；可以在客户端设置一个令牌，以便服务器对客户端进行标识 [19]
Set-Cookie2	与 Set-Cookie 类似，RFC 2965 Cookie 定义；参见 11.6.7 节
WWW-Authenticate	来自服务器的对客户端的质询列表

3.5.4　实体首部

有很多首部可以用来描述 HTTP 报文的负荷。由于请求和响应报文中都可能包含实体部分，所以在这两种类型的报文中都可能出现这些首部。

实体首部提供了有关实体及其内容的大量信息，从有关对象类型的信息，到能够对资源使用的各种有效的请求方法。总之，实体首部可以告知报文的接收者它在对什么进行处理。表 3-21 列出了实体的信息性首部。

注 19：Set-Cookie 和 Set-Cookie2 都是扩展首部，参见第 11 章。

表3-21　实体的信息性首部

首　部	描　述
Allow	列出了可以对此实体执行的请求方法
Location	告知客户端实体实际上位于何处；用于将接收端定向到资源的（可能是新的）位置（URL）上去

1. 内容首部

内容首部提供了与实体内容有关的特定信息，说明了其类型、尺寸以及处理它所需的其他有用信息。比如，Web 浏览器可以通过查看返回的内容类型，得知如何显示对象。表 3-22 列出了各种内容首部。

表3-22　内容首部

首　部	描　述
Content-Base[20]	解析主体中的相对 URL 时使用的基础 URL
Content-Encoding	对主体执行的任意编码方式
Content-Language	理解主体时最适宜使用的自然语言
Content-Length	主体的长度或尺寸
Content-Location	资源实际所处的位置
Content-MD5	主体的 MD5 校验和
Content-Range	在整个资源中此实体表示的字节范围
Content-Type	这个主体的对象类型

2. 实体缓存首部

通用的缓存首部说明了如何或什么时候进行缓存。实体的缓存首部提供了与被缓存实体有关的信息——比如，验证已缓存的资源副本是否仍然有效所需的信息，以及更好地估计已缓存资源何时失效所需的线索。

第 7 章深入讨论了 HTTP 请求和响应的缓存。在那里我们会再次看到这些首部。表 3-23 列出了一些实体缓存首部。

表3-23　实体缓存首部

首　部	描　述
ETag	与此实体相关的实体标记[21]
Expires	实体不再有效，要从原始的源端再次获取此实体的日期和时间
Last-Modified	这个实体最后一次被修改的日期和时间

注 20：RFC 2616 中没有定义 Content-Base 首部。

注 21：实体标记本质上来说就是某个特定资源版本的标识符。

3.6 更多信息

更多信息请参见下列资源。

- http://www.w3.org/Protocols/rfc2616/rfc2616.txt
  RFC 2616，“Hypertext Transfer Protocol”，由 R. Fielding、J. Gettys、J. Mogul、H. Frystyk、L. Mastinter、P. Leach 和 T. Berners-Lee 编写。
- *HTTP Pocket Reference*（《HTTP 口袋书》）
  Clintin Wong 编写，O’Reilly & Associates 公司出版。
- http://www.w3.org/Protocols/
  HTTP 的 W3C 架构页面。

73

第4章

连接管理

HTTP 规范对 HTTP 报文解释得很清楚，但对 HTTP 连接介绍的并不多，HTTP 连接是 HTTP 报文传输的关键通道。编写 HTTP 应用程序的程序员需要理解 HTTP 连接的来龙去脉以及如何使用这些连接。

HTTP 连接管理有点像魔术，应当从经验与实践，而不仅仅是出版的文献中学习。通过本章，可以了解到：

- HTTP 是如何使用 TCP 连接的；
- TCP 连接的时延、瓶颈以及存在的障碍；
- HTTP 的优化，包括并行连接、keep-alive（持久连接）和管道化连接；
- 管理连接时应该以及不应该做的事情。

4.1 TCP连接

世界上几乎所有的 HTTP 通信都是由 TCP/IP 承载的，TCP/IP 是全球计算机及网络设备都在使用的一种常用的分组交换网络分层协议集。客户端应用程序可以打开一条 TCP/IP 连接，连接到可能运行在世界任何地方的服务器应用程序。一旦连接建立起来了，在客户端和服务器的计算机之间交换的报文就永远不会丢失、受损或失序。[1]

比如，你想获取 Joe 的五金商店最新的电动工具价目表：

http://www.joes-hardware.com:80/power-tools.html

浏览器收到这个 URL 时，会执行图 4-1 所示的步骤。第 (1) ~ (3) 步会将服务器的
74 IP 地址和端口号从 URL 中分离出来。在第 (4) 步中建立到 Web 服务器的 TCP 连接，并在第 (5) 步通过这条连接发送一条请求报文。在第 (6) 步读取响应，并在第 (7) 步关闭连接。

4.1.1 TCP的可靠数据管道

HTTP 连接实际上就是 TCP 连接及其使用规则。TCP 连接是因特网上的可靠连接。要想正确、快速地发送数据，就需要了解 TCP 的一些基本知识。[2]

注 1：尽管报文不会丢失或受损，但如果计算机或网络崩溃了，客户端和服务器之间的通信仍然会被断开。在这种情况下，会通知客户端和服务器通信中断了。

注 2：如果要编写复杂的 HTTP 应用程序，尤其是，希望程序能够快速运行的话，所需学习的、与 TCP 内部原理及性能有关的知识就要比本章所讨论的内容多得多。我们推荐 W. Richard Stevens 编写的 *TCP/IP Illustrated*（《TCP/IP 详解》）系列图书（Addison Wesley 公司出版）。

TCP 为 HTTP 提供了一条可靠的比特传输管道。从 TCP 连接一端填入的字节会从另一端以原有的顺序、正确地传送出来（参见图 4-2）。

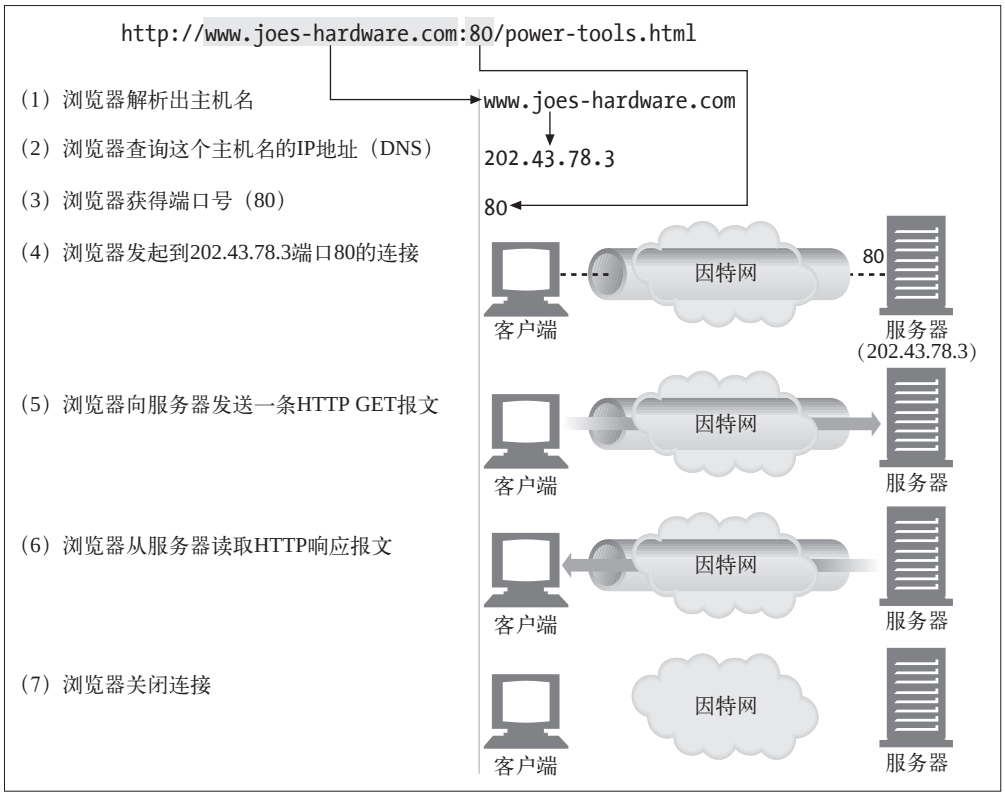


图 4-1　Web 浏览器通过 TCP 连接与 Web 服务器进行交互



图 4-2　TCP 会按序、无差错地承载 HTTP 数据

4.1.2　TCP流是分段的、由IP分组传送

TCP 的数据是通过名为 IP 分组（或 IP 数据报）的小数据块来发送的。这样的话，如图 4-3a 所示，HTTP 就是“HTTP over TCP over IP”这个“协议栈”中的最顶层了。其安全版本 HTTPS 就是在 HTTP 和 TCP 之间插入了一个（称为 TLS 或 SSL 的）密码加密层（图 4-3b）。

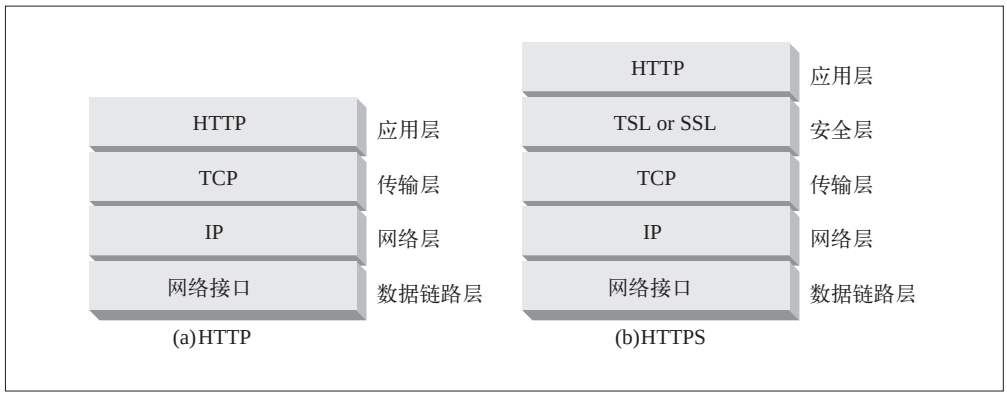


图 4-3　HTTP 和 HTTPS 网络协议栈

HTTP 要传送一条报文时，会以流的形式将报文数据的内容通过一条打开的 TCP 连接按序传输。TCP 收到数据流之后，会将数据流砍成被称作段的小数据块，并将段封装在 IP 分组中，通过因特网进行传输（参见图 4-4）。所有这些工作都是由 TCP/IP 软件来处理的，HTTP 程序员什么都看不到。

每个 TCP 段都是由 IP 分组承载，从一个 IP 地址发送到另一个 IP 地址的。每个 IP 分组中都包括：

- 一个 IP 分组首部（通常为 20 字节）；
- 一个 TCP 段首部（通常为 20 字节）；
- 一个 TCP 数据块（0 个或多个字节）。

IP 首部包含了源和目的 IP 地址、长度和其他一些标记。TCP 段的首部包含了 TCP
76 端口号、TCP 控制标记，以及用于数据排序和完整性检查的一些数字值。

4.1.3　保持TCP连接持续不断地运行

在任意时刻计算机都可以有几条 TCP 连接处于打开状态。TCP 是通过*端口号*来保持所有这些连接持续不断地运行。

端口号和雇员使用的电话分机号很类似。就像公司的总机号码能将你接到前台，而分机号可以将你接到正确的雇员位置一样，IP 地址可以将你连接到正确的计算机，而端口号则可以将你连接到正确的应用程序上去。TCP 连接是通过 4 个值来识别的：

< 源 IP 地址、源端口号、目的 IP 地址、目的端口号 >

77 这 4 个值一起唯一地定义了一条连接。两条不同的 TCP 连接不能拥有 4 个完全相同的地址组件值（但不同连接的部分组件可以拥有相同的值）。

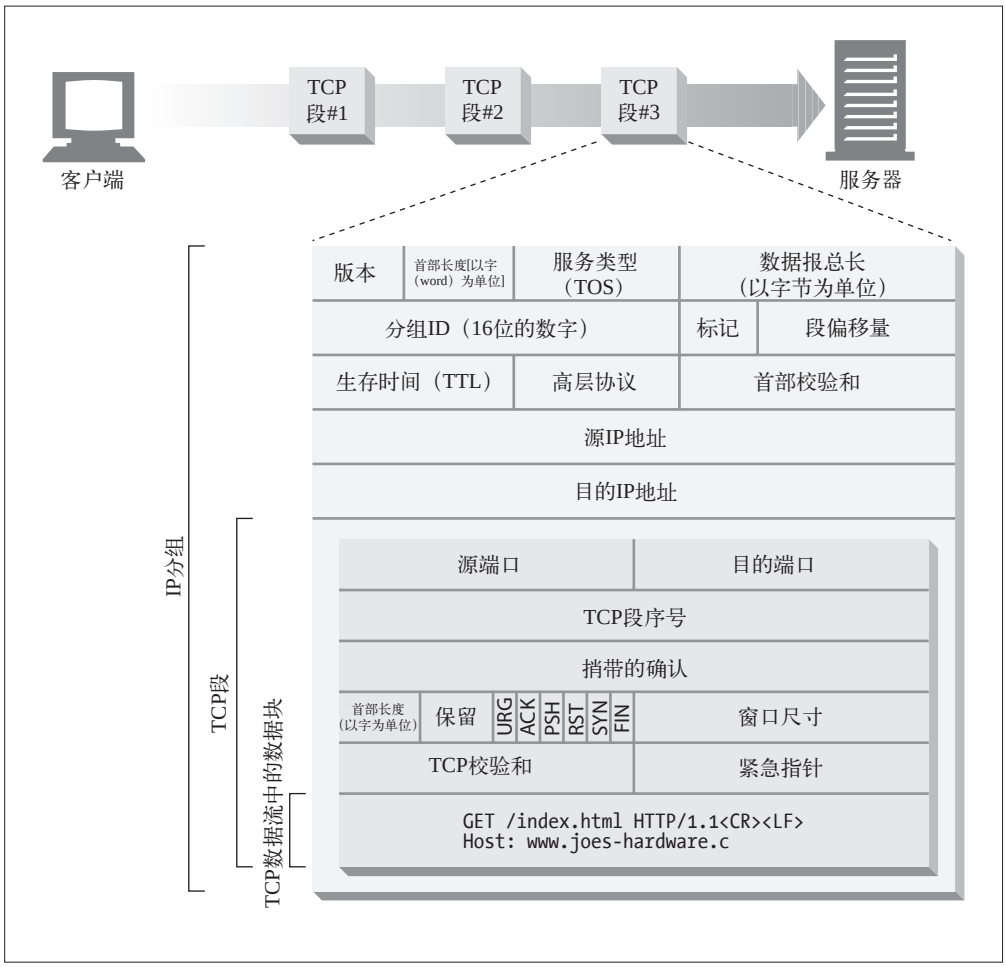


图 4-4　承载 TCP 段的 IP 分组，它承载了 TCP 数据流中的小块数据

在图 4-5 中，有 4 条连接：A、B、C 和 D。表 4-1 列出了每个端口的相关信息。

表4-1　TCP连接值

连　接	源IP地址	源　端　口	目的IP地址	目的端口
A	209.1.32.34	2034	204.62.128.58	4133
B	209.1.32.35	3227	204.62.128.58	4140
C	209.1.32.35	3105	207.25.71.25	80
D	209.1.33.89	5100	207.25.71.25	80

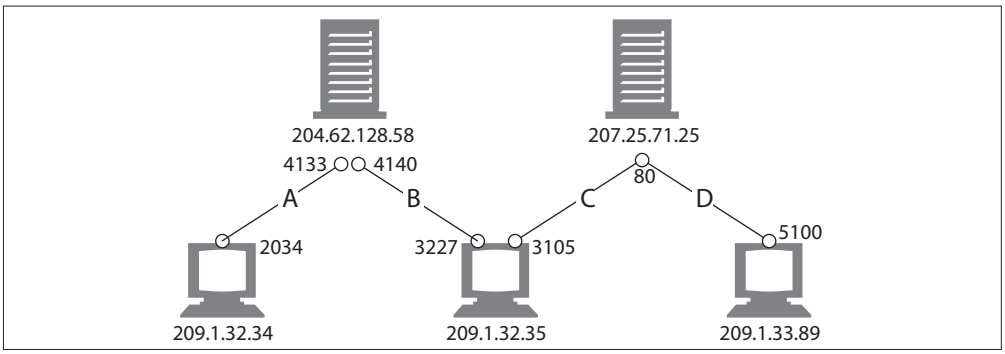


图 4-5　4 个不同的 TCP 连接

注意，有些连接共享了相同的目的端口号（C 和 D 都使用目的端口号 80）。有些连接使用了相同的源 IP 地址（B 和 C）。有些使用了相同的目的 IP 地址（A 和 B，C 和 D）。但没有两个不同连接所有的 4 个值都一样。

4.1.4　用TCP套接字编程

操作系统提供了一些操纵其 TCP 连接的工具。为了更具体地说明问题，我们来看一个 TCP 编程接口。表 4-2 显示了套接字 API 提供的一些主要接口。这个套接字 API 向 HTTP 程序员隐藏了 TCP 和 IP 的所有细节。套接字 API 最初是为 Unix 操作系统开发的，但现在几乎所有的操作系统和语言中都有其变体存在。

表4-2　对TCP连接进行编程所需的常见套接字接口函数

套接字API调用	描　　述
`s = socket(<parameters>)`	创建一个新的、未命名、未关联的套接字
`bind(s,<local IP:port>)`	向套接字赋一个本地端口号和接口
`connect(s, <remote IP:port>)`	创建一条连接本地套接字与远程主机及端口的连接
`listen(s,...)`	标识一个本地套接字，使其可以合法接受连接
`s2 = accept(s)`	等待某人建立一条到本地端口的连接
n `= read(s, buffer, n)`	尝试从套接字向缓冲区读取 *n* 个字节
n `= write(s, buffer, n)`	尝试从缓冲区中向套接字写入 *n* 个字节
`close(s)`	完全关闭 TCP 连接
`shutdown(s,<side>)`	只关闭 TCP 连接的输入或输出端
`getsockopt(s,...)`	读取某个内部套接字配置选项的值
`setsockopt(s,...)`	修改某个内部套接字配置选项的值

套接字 API 允许用户创建 TCP 的端点数据结构，将这些端点与远程服务器的 TCP 端点进行连接，并对数据流进行读写。TCP API 隐藏了所有底层网络协议的握手细节，以及 TCP 数据流与 IP 分组之间的分段和重装细节。

图 4-1 显示了 Web 浏览器是如何用 HTTP 从 Joe 的五金商店下载 power-tools.html 页面的。图 4-6 中的伪代码说明了可以怎样通过套接字 API 来凸显客户端和服务器在实现 HTTP 事务时所应执行的步骤。

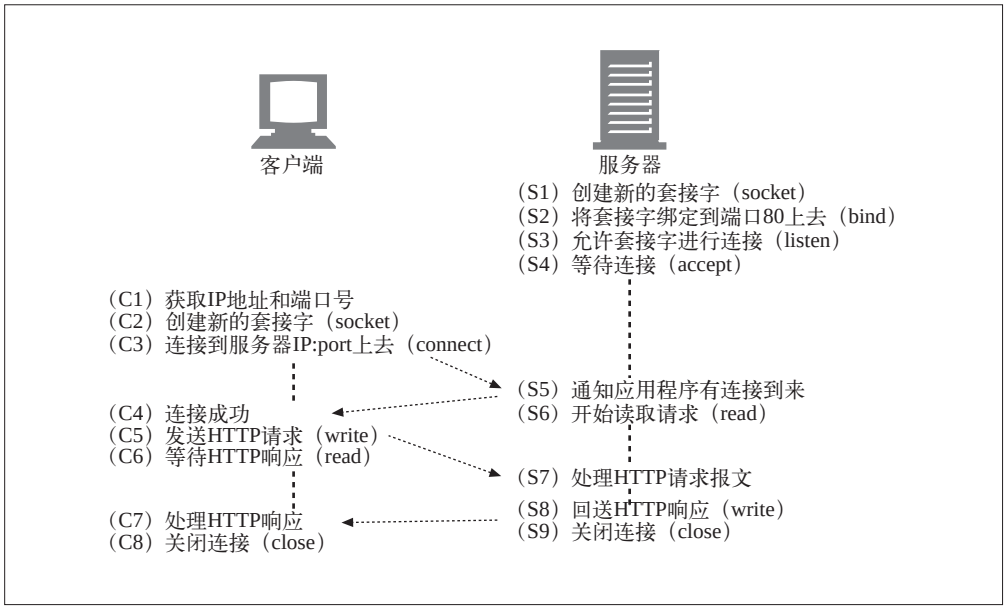


图 4-6　TCP 客户端和服务器是如何通过 TCP 套接字接口进行通信的

我们从 Web 服务器等待连接（参见图 4-6，S4）开始。客户端根据 URL 判定出 IP 地址和端口号，并建立一条到服务器的 TCP 连接（参见图 4-6，C3）。建立连接可能要花费一些时间，时间长短取决于服务器距离的远近、服务器的负载情况，以及因特网的拥挤程度。

一旦建立了连接，客户端就会发送 HTTP 请求（参见图 4-6，C5），服务器则会读取请求（参见图 4-6，S6）。一旦服务器获取了整条请求报文，就会对请求进行处理，执行所请求的动作（参见图 4-6，S7），并将数据写回客户端。客户端读取数据（参见图 4-6，C6），并对响应数据进行处理（参见图 4-6，C7）。

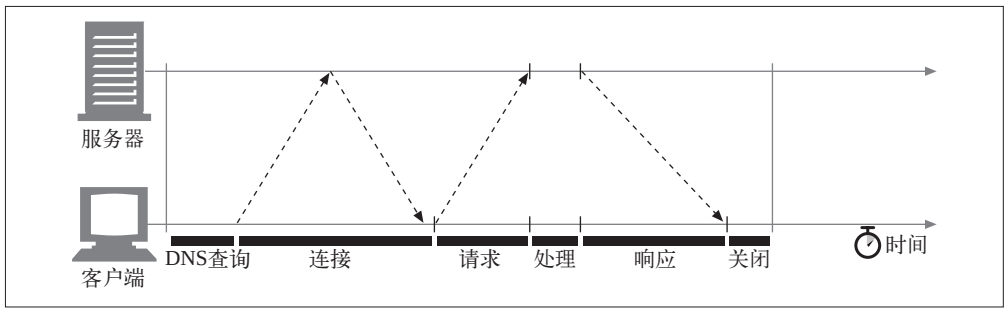
4.2　对TCP性能的考虑

HTTP 紧挨着 TCP，位于其上层，所以 HTTP 事务的性能在很大程度上取决于底层 TCP 通道的性能。本节重点介绍了一些很重要的、对这些 TCP 连接的性能考虑。理解了 TCP 的某些基本性能特点之后，就可以更好地理解 HTTP 的连接优化特性，这样就能设计实现一些更高性能的 HTTP 应用程序了。

本节要求大家对 TCP 协议的内部细节有一定的了解。如果对 TCP 性能考虑的细节不感兴趣（或者很熟悉这些细节），可以直接跳到 4.3 节。TCP 是个很复杂的话题，所以这里我们只能提供对 TCP 性能的简要概述。本章末尾的 4.8 节列出了一些很好的 TCP 参考书，以供参考。

4.2.1 HTTP事务的时延

我们来回顾一下，在 HTTP 请求的过程中会出现哪些网络时延，并以此开始我们的 TCP 性能之旅。图 4-7 描绘了 HTTP 事务主要的连接、传输以及处理时延。



80 图 4-7 串行 HTTP 事务的时间线

注意，与建立 TCP 连接，以及传输请求和响应报文的时间相比，事务处理时间可能是很短的。除非客户端或服务器超载，或正在处理复杂的动态资源，否则 HTTP 时延就是由 TCP 网络时延构成的。

HTTP 事务的时延有以下几种主要原因。

(1) 客户端首先需要根据 URI 确定 Web 服务器的 IP 地址和端口号。如果最近没有对 URI 中的主机名进行访问，通过 DNS 解析系统将 URI 中的主机名转换成一个 IP 地址可能要花费数十秒的时间 [3]。

(2) 接下来，客户端会向服务器发送一条 TCP 连接请求，并等待服务器回送一个请求接受应答。每条新的 TCP 连接都会有连接建立时延。这个值通常最多只有一两秒钟，但如果有数百个 HTTP 事务的话，这个值会快速地叠加上去。

(3) 一旦连接建立起来了，客户端就会通过新建立的 TCP 管道来发送 HTTP 请求。数据到达时，Web 服务器会从 TCP 连接中读取请求报文，并对请求进行处理。

注 3：幸运的是，大多数 HTTP 客户端都有一个小的 DNS 缓存，用来保存近期所访问站点的 IP 地址。如果已经在本地“缓存”（记录）了 IP 地址，查询就可以立即完成。因为大多数 Web 浏览器浏览的都是少数常用站点，所以通常都可以很快地将主机名解析出来。

因特网传输请求报文，以及服务器处理请求报文都需要时间。

(4) 然后，Web 服务器会回送 HTTP 响应，这也需要花费时间。

这些 TCP 网络时延的大小取决于硬件速度、网络和服务器的负载，请求和响应报文的尺寸，以及客户端和服务器之间的距离。TCP 协议的技术复杂性也会对时延产生巨大的影响。

4.2.2 性能聚焦区域

本节其余部分列出了一些会对 HTTP 程序员产生影响的、最常见的 TCP 相关时延，其中包括：

- TCP 连接建立握手；
- TCP 慢启动拥塞控制；
- 数据聚集的 Nagle 算法；
- 用于捎带确认的 TCP 延迟确认算法；
- TIME_WAIT 时延和端口耗尽。

如果要编写高性能的 HTTP 软件，就应该理解上面的每一个因素。如果不需要进行这个级别的性能优化，可以跳过这部分内容。

81

4.2.3 TCP连接的握手时延

建立一条新的 TCP 连接时，甚至是在发送任意数据之前，TCP 软件之间会交换一系列的 IP 分组，对连接的有关参数进行沟通（参见图 4-8）。如果连接只用来传送少量数据，这些交换过程就会严重降低 HTTP 的性能。

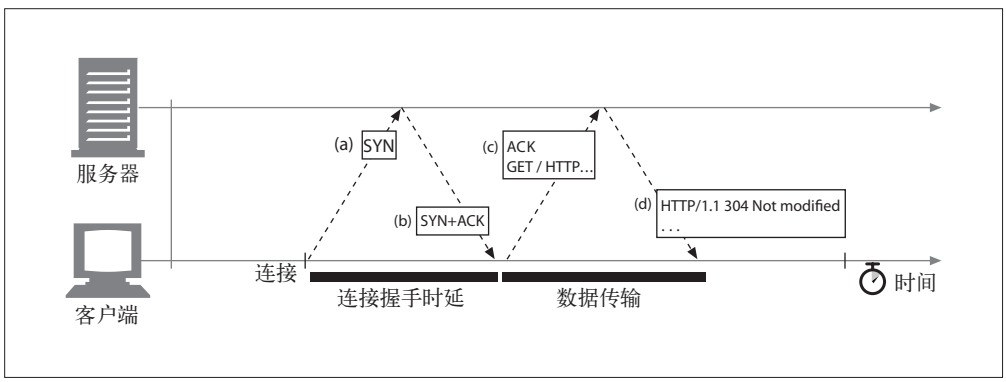


图 4-8 在发送数据之前，TCP 要传送两个分组来建立连接

TCP 连接握手需要经过以下几个步骤。

(1) 请求新的 TCP 连接时，客户端要向服务器发送一个小的 TCP 分组（通常是 40 ～ 60 个字节）。这个分组中设置了一个特殊的 SYN 标记，说明这是一个连接请求。（参见图 4-8a）。

(2) 如果服务器接受了连接，就会对一些连接参数进行计算，并向客户端回送一个 TCP 分组，这个分组中的 SYN 和 ACK 标记都被置位，说明连接请求已被接受（参见图 4-8b）。

(3) 最后，客户端向服务器回送一条确认信息，通知它连接已成功建立（参见图 4-8c）。现代的 TCP 栈都允许客户端在这个确认分组中发送数据。

HTTP 程序员永远不会看到这些分组——这些分组都由 TCP/IP 软件管理，对其是不可见的。HTTP 程序员看到的只是创建 TCP 连接时存在的时延。

通常 HTTP 事务都不会交换太多数据，此时，SYN/SYN+ACK 握手（参见图 4-8a 和图 4-8b）会产生一个可测量的时延。TCP 连接的 ACK 分组（参见图 4-8c）通常都足够大，可以承载整个 HTTP 请求报文[4]，而且很多 HTTP 服务器响应报文都可以放入一个 IP 分组中去（比如，响应是包含了装饰性图片的小型 HTML 文件，或
82 者是对浏览器高速缓存请求产生的 304 Not Modified 响应）。

最后的结果是，小的 HTTP 事务可能会在 TCP 建立上花费 50%，或更多的时间。后面的小节会讨论 HTTP 是如何通过重用现存连接，来减小这种 TCP 建立时延所造成的影响的。

4.2.4 延迟确认

由于因特网自身无法确保可靠的分组传输（因特网路由器超负荷的话，可以随意丢弃分组），所以 TCP 实现了自己的确认机制来确保数据的成功传输。

每个 TCP 段都有一个序列号和数据完整性校验和。每个段的接收者收到完好的段时，都会向发送者回送小的确认分组。如果发送者没有在指定的窗口时间内收到确认信息，发送者就认为分组已被破坏或损毁，并重发数据。

由于确认报文很小，所以 TCP 允许在发往相同方向的输出数据分组中对其进行“捎带”。TCP 将返回的确认信息与输出的数据分组结合在一起，可以更有效地利用网络。为了增加确认报文找到同向传输数据分组的可能性，很多 TCP 栈都实现了一种

注 4：因特网流量中的 IP 分组通常是几百字节，本地流量中的 IP 分组为 1500 字节左右。

“延迟确认”算法。延迟确认算法会在一个特定的窗口时间（通常是 100 ~ 200 毫秒）内将输出确认存放在缓冲区中，以寻找能够捎带它的输出数据分组。如果在那个时间段内没有输出数据分组，就将确认信息放在单独的分组中传送。

但是，HTTP 具有双峰特征的请求 - 应答行为降低了捎带信息的可能。当希望有相反方向回传分组的时候，偏偏没有那么多。通常，延迟确认算法会引入相当大的时延。根据所使用操作系统的不同，可以调整或禁止延迟确认算法。

在对 TCP 栈的任何参数进行修改之前，一定要对自己在做什么有清醒的认识。TCP 中引入这些算法的目的是防止设计欠佳的应用程序对因特网造成破坏。对 TCP 配置进行的任意修改，都要绝对确保应用程序不会引发这些算法所要避免的问题。

4.2.5 TCP慢启动

TCP 数据传输的性能还取决于 TCP 连接的*使用期*（age）。TCP 连接会随着时间进行自我“调谐”，起初会限制连接的最大速度，如果数据成功传输，会随着时间的推移提高传输的速度。这种调谐被称为 TCP *慢启动*（slow start），用于防止因特网的突然过载和拥塞。

TCP 慢启动限制了一个 TCP 端点在任意时刻可以传输的分组数。简单来说，每成功接收一个分组，发送端就有了发送另外两个分组的权限。如果某个 HTTP 事务有大量数据要发送，是不能一次将所有分组都发送出去的。必须发送一个分组，等待确认；然后可以发送两个分组，每个分组都必须被确认，这样就可以发送四个分组了，以此类推。这种方式被称为“打开拥塞窗口”。

由于存在这种拥塞控制特性，所以新连接的传输速度会比已经交换过一定量数据的、“已调谐”连接慢一些。由于已调谐连接要更快一些，所以 HTTP 中有一些可以重用现存连接的工具。本章稍后会介绍这些 HTTP“持久连接”。

4.2.6 Nagle算法与TCP_NODELAY

TCP 有一个数据流接口，应用程序可以通过它将任意尺寸的数据放入 TCP 栈中——即使一次只放一个字节也可以！但是，每个 TCP 段中都至少装载了 40 个字节的标记和首部，所以如果 TCP 发送了大量包含少量数据的分组，网络的性能就会严重下降。[5]

注 5：发送大量单字节分组的行为称为“发送端傻窗口综合症”。这种行为效率很低、违反社会道德，而且可能会影响其他的因特网流量。

Nagle 算法（根据其发明者 John Nagle 命名）试图在发送一个分组之前，将大量 TCP 数据绑定在一起，以提高网络效率。RFC 896“IP/TCP 互连网络中的拥塞控制”对此算法进行了描述。

Nagle 算法鼓励发送全尺寸（LAN 上最大尺寸的分组大约是 1500 字节，在因特网上是几百字节）的段。只有当所有其他分组都被确认之后，Nagle 算法才允许发送非全尺寸的分组。如果其他分组仍然在传输过程中，就将那部分数据缓存起来。只有当挂起分组被确认，或者缓存中积累了足够发送一个全尺寸分组的数据时，才会将缓存的数据发送出去。[6]

Nagle 算法会引发几种 HTTP 性能问题。首先，小的 HTTP 报文可能无法填满一个分组，可能会因为等待那些永远不会到来的额外数据而产生时延。其次，Nagle 算法与延迟确认之间的交互存在问题——Nagle 算法会阻止数据的发送，直到有确认分组抵达为止，但确认分组自身会被延迟确认算法延迟 100 ～ 200 毫秒。[7]

HTTP 应用程序常常会在自己的栈中设置参数 TCP_NODELAY，禁用 Nagle 算法，提高性能。如果要这么做的话，一定要确保会向 TCP 写入大块的数据，这样就不会产生一堆小分组了。

4.2.7 TIME_WAIT累积与端口耗尽

TIME_WAIT 端口耗尽是很严重的性能问题，会影响到性能基准，但在现实中相对较少出现。大多数遇到性能基准问题的人最终都会碰到这个问题，而且性能都会变得出乎意料地差，所以这个问题值得特别关注。

当某个 TCP 端点关闭 TCP 连接时，会在内存中维护一个小的控制块，用来记录最近所关闭连接的 IP 地址和端口号。这类信息只会维持一小段时间，通常是所估计的最大分段使用期的两倍（称为 2MSL，通常为 2 分钟[8]）左右，以确保在这段时间内不会创建具有相同地址和端口号的新连接。实际上，这个算法可以防止在两分钟内创建、关闭并重新创建两个具有相同 IP 地址和端口号的连接。

现在高速路由器的使用，使得重复分组几乎不可能在连接关闭的几分钟之后，出现在服务器上。有些操作系统会将 2MSL 设置为一个较小的值，但修改此值时要特别

注 6：这个算法有几种变体，包括对超时和确认逻辑的修改，但基本算法会使数据的缓存比一个 TCP 段小一些。

注 7：使用管道化连接（本章稍后介绍）时这些问题可能会更加严重，因为客户端可能会有多条报文要发送给同一个服务器，而且不希望有时延存在。

注 8：将 2MSL 的值取为 2 分钟是有历史原因的。很早以前，路由器的速度还很慢，人们估计，在将一个分组的复制副本丢弃之前，它可以在因特网队列中保留最多一分钟的时间。现在，最大分段生存期要小得多了。

小心。分组确实会被复制，如果来自之前连接的复制分组插入了具有相同连接值的新 TCP 流，会破坏 TCP 数据。

2MSL 的连接关闭延迟通常不是什么问题，但在性能基准环境下就可能会成为一个问题。进行性能基准测试时，通常只有一台或几台用来产生流量的计算机连接到某系统中去，这样就限制了连接到服务器的客户端 IP 地址数。而且，服务器通常会在 HTTP 的默认 TCP 端口 80 上进行监听。用 TIME_WAIT 防止端口号重用时，这些情况也限制了可用的连接值组合。

在只有一个客户端和一台 Web 服务器的异常情况下，构建一条 TCP 连接的 4 个值：

```
<source-IP-address, source-port, destination-IP-address, destination-port>
```

其中的 3 个都是固定的——只有源端口号可以随意改变：

```
<client-IP, source-port, server-IP, 80>
```

客户端每次连接到服务器上去时，都会获得一个新的源端口，以实现连接的唯一性。但由于可用源端口的数量有限（比如，60 000 个），而且在 2MSL 秒（比如，120 秒）内连接是无法重用的，连接率就被限制在了 60 000/120=500 次 / 秒。如果再不断进行优化，并且服务器的连接率不高于 500 次 / 秒，就可确保不会遇到 TIME_WAIT 端口耗尽问题。要修正这个问题，可以增加客户端负载生成机器的数量，或者确保客户端和服务器在循环使用几个虚拟 IP 地址以增加更多的连接组合。

即使没有遇到端口耗尽问题，也要特别小心有大量连接处于打开状态的情况，或为处于等待状态的连接分配了大量控制块的情况。在有大量打开连接或控制块的情况下，有些操作系统的速度会严重减缓。

4.3 HTTP连接的处理

本章的前两节对 TCP 连接及其性能含义进行了精要的介绍。要想学习更多与 TCP 联网有关的知识，请参见本章末尾的资源列表。

现在我们切回到 HTTP 上来。本章其余部分将解释操作和优化连接的 HTTP 技术。我们从 HTTP 的 `Connection` 首部开始介绍，这是 HTTP 连接管理中一个很容易被误解，但又很重要的部分。然后会介绍一些 HTTP 连接优化技术。

4.3.1 常被误解的Connection首部

HTTP 允许在客户端和最终的源端服务器之间存在一串 HTTP 中间实体（代理、高速缓存等）。可以从客户端开始，逐跳地将 HTTP 报文经过这些中间设备，转发到源

端服务器上去（或者进行反向传输）。

在某些情况下，两个相邻的 HTTP 应用程序会为它们共享的连接应用一组选项。HTTP 的 Connection 首部字段中有一个由逗号分隔的*连接标签*列表，这些标签为此连接指定了一些不会传播到其他连接中去的选项。比如，可以用 Connection: close 来说明发送完下一条报文之后必须关闭的连接。

Connection 首部可以承载 3 种不同类型的标签，因此有时会很令人费解：

- HTTP 首部字段名，列出了只与此连接有关的首部；
- 任意标签值，用于描述此连接的非标准选项；
- 值 close，说明操作完成之后需关闭这条持久连接。

如果连接标签中包含了一个 HTTP 首部字段的名称，那么这个首部字段就包含了与一些连接有关的信息，不能将其转发出去。在将报文转发出去之前，必须删除
86 Connection 首部列出的所有首部字段。由于 Connection 首部可以防止无意中对本地首部的转发，因此将逐跳首部名放入 Connection 首部被称为“对首部的保护”。图 4-9 显示了一个这样的例子。

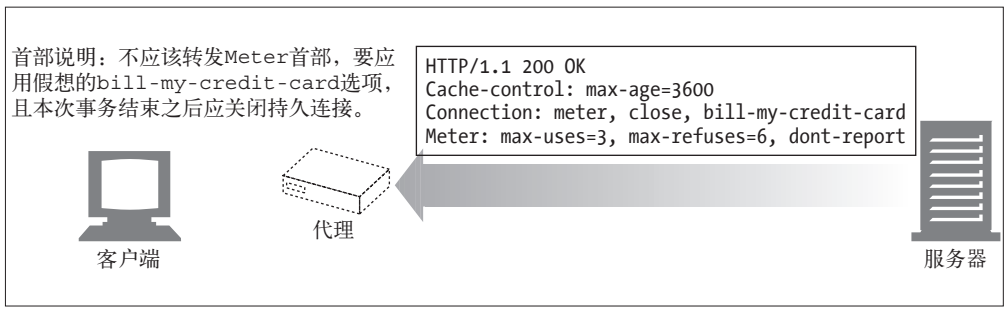


图 4-9 Connection 首部允许发送端指定与连接有关的选项

HTTP 应用程序收到一条带有 Connection 首部的报文时，接收端会解析发送端请求的所有选项，并将其应用。然后会在将此报文转发给下一跳地址之前，删除 Connection 首部以及 Connection 中列出的所有首部。而且，可能还会有少量没有作为 Connection 首部值列出，但一定不能被代理转发的逐跳首部。其中包括 Prxoy-Authenticate、Proxy-Connection、Transfer-Encoding 和 Upgrade。更多有关 Connection 首部的内容请参见附录 C。

4.3.2 串行事务处理时延

如果只对连接进行简单的管理，TCP 的性能时延可能会叠加起来。比如，假设有一

个包含了 3 个嵌入图片的 Web 页面。浏览器需要发起 4 个 HTTP 事务来显示此页面：1 个用于顶层的 HTML 页面，3 个用于嵌入的图片。如果每个事务都需要（串行地建立）一条新的连接，那么连接时延和慢启动时延就会叠加起来（参见图 4-10）。[9]

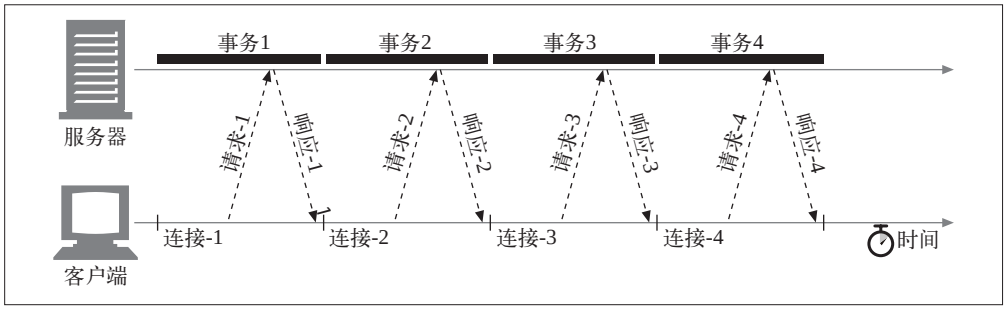


图 4-10　4 个事务（串行）

除了串行加载引入的实际时延之外，加载一幅图片时，页面上其他地方都没有动静也会让人觉得速度很慢。用户更希望能够同时加载多幅图片。[10]

串行加载的另一个缺点是，有些浏览器在对象加载完毕之前无法获知对象的尺寸，而且它们可能需要尺寸信息来决定将对象放在屏幕的什么位置上，所以在加载了足够多的对象之前，无法在屏幕上显示任何内容。在这种情况下，可能浏览器串行装载对象的进度很正常，但用户面对的却是一个空白的屏幕，对装载的进度一无所知。[11]

还有几种现存和新兴的方法可以提高 HTTP 的连接性能。后面几节讨论了四种此类技术。

- *并行连接*
  通过多条 TCP 连接发起并发的 HTTP 请求。

- *持久连接*
  重用 TCP 连接，以消除连接及关闭时延。

- *管道化连接*
  通过共享的 TCP 连接发起并发的 HTTP 请求。

注 9：根据举此例的目的，假设所有对象的长度基本上都一样，并且是从同一台服务器发出的，而且 DNS 条目被缓存了，排除了 DNS 的查找时间。

注 10：即使同时加载多幅图片比一次加载一幅图片要慢，人们也会有同样的感觉！用户通常会认为多幅图片同时加载要快一些。

注 11：HTML 的设计者可以在图片等嵌入式对象的 HTML 标签中显式地添加宽高属性，以消除这种“布局时延”。显式地提供了嵌入图片的宽度和高度，浏览器就可以在从服务器收到对象之前确定图形的布局了。

- 复用的连接
  交替传送请求和响应报文（实验阶段）。

4.4 并行连接

如前所述，浏览器可以先完整地请求原始的 HTML 页面，然后请求第一个嵌入对象，然后请求第二个嵌入对象等，以这种简单的方式对每个嵌入式对象进行串行处理。但这样实在是太慢了！

如图 4-11 所示，HTTP 允许客户端打开多条连接，并行地执行多个 HTTP 事务。在这个例子中，并行加载了四幅嵌入式图片，每个事务都有自己的 TCP 连接。[12]

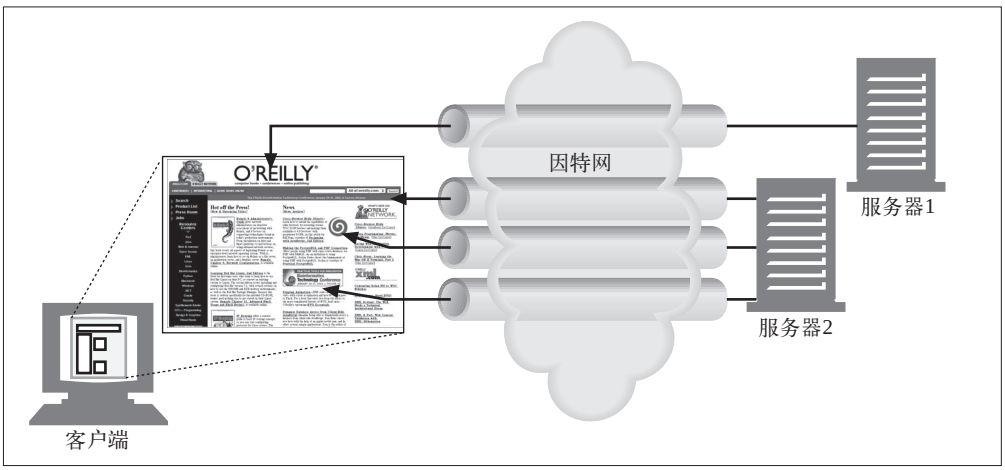


图 4-11 页面上的每个组件都包含一个独立的 HTTP 事务

4.4.1 并行连接可能会提高页面的加载速度

包含嵌入对象的组合页面如果能（通过并行连接）克服单条连接的空载时间和带宽限制，加载速度也会有所提高。时延可以重叠起来，而且如果单条连接没有充分利用客户端的因特网带宽，可以将未用带宽分配来装载其他对象。

88

图 4-12 显示了并行连接的时间线，比图 4-10 要快得多。首先装载的是封闭的 HTML 页面，然后并行处理其余的 3 个事务，每个事务都有自己的连接。[13] 图片的装载是并行的，连接的时延也是重叠的。

注 12：嵌入的组件不一定都在同一台 Web 服务器上，可以同多台服务器建立并行的连接。

注 13：由于软件开销的存在，每个连接请求之间总是会有一些小的时延，但连接请求和传输时间基本上都是重叠起来的。

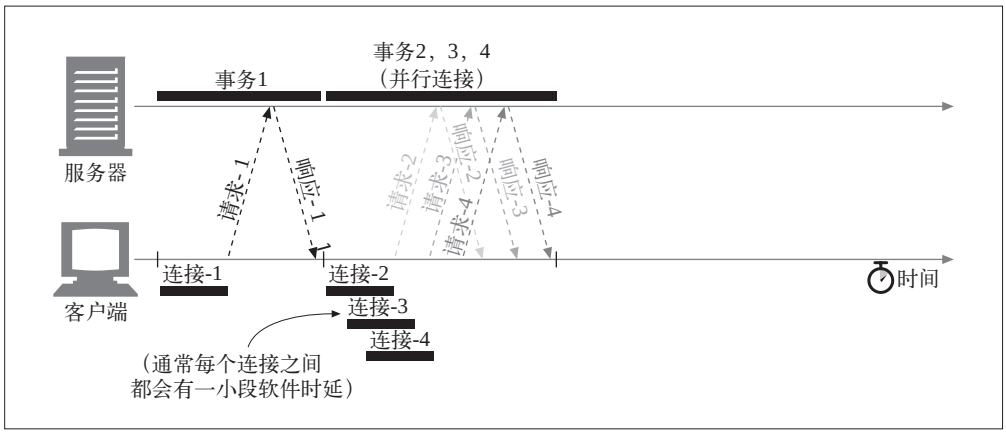


图 4-12　4 个事务（并行）

4.4.2　并行连接不一定更快

即使并行连接的速度可能会更快，但并不一定总是更快。客户端的网络带宽不足（比如，浏览器是通过一个 28.8kbps 的 Modem 连接到因特网上去的）时，大部分的时间可能都是用来传送数据的。在这种情况下，一个连接到速度较快服务器上的 HTTP 事务就会很容易地耗尽所有可用的 Modem 带宽。如果并行加载多个对象，每个对象都会去竞争这有限的带宽，每个对象都会以较慢的速度按比例加载，这样带来的性能提升就很小，甚至没什么提升。[14]

而且，打开大量连接会消耗很多内存资源，从而引发自身的性能问题。复杂的 Web 页面可能会有数十或数百个内嵌对象。客户端可能可以打开数百个连接，但 Web 服务器通常要同时处理很多其他用户的请求，所以很少有 Web 服务器希望出现这样的情况。一百个用户同时发出申请，每个用户打开 100 个连接，服务器就要负责处理 10 000 个连接。这会造成服务器性能的严重下降。对高负荷的代理来说也同样如此。

实际上，浏览器确实使用了并行连接，但它们会将并行连接的总数限制为一个较小的值（通常是 4 个）。服务器可以随意关闭来自特定客户端的超量连接。

4.4.3　并行连接可能让人“感觉”更快一些

好了，这样看来并行连接并不总是能使页面加载得更快一些。但如前所述，即使实际上它们并没有加快页面的传输速度，并行连接通常也会让用户觉得页面加载得更

注 14：实际上，多条连接会产生一些额外的开销，使用并行连接装载整个页面所需的时间很可能比串行下载的时间更长。

快了，因为多个组件对象同时出现在屏幕上时，用户能够看到加载的进展。[15] 如果整个屏幕上有很多动作在进行，即使实际上秒表显示整个页面的下载时间更长，人们也会认为 Web 页面加载得更快一些。

4.5 持久连接

Web 客户端经常会打开到同一个站点的连接。比如，一个 Web 页面上的大部分内嵌图片通常都来自同一个 Web 站点，而且相当一部分指向其他对象的超链通常都指向同一个站点。因此，初始化了对某服务器 HTTP 请求的应用程序很可能会在不久的将来对那台服务器发起更多的请求（比如，获取在线图片）。这种性质被称为*站点局部性*（site locality）。

因此，HTTP/1.1（以及 HTTP/1.0 的各种增强版本）允许 HTTP 设备在事务处理结束之后将 TCP 连接保持在打开状态，以便为未来的 HTTP 请求重用现存的连接。在事务处理结束之后仍然保持在打开状态的 TCP 连接被称为*持久连接*。非持久连接会在每个事务结束之后关闭。持久连接会在不同事务之间保持打开状态，直到客户端或服务器决定将其关闭为止。

重用已对目标服务器打开的空闲持久连接，就可以避开缓慢的连接建立阶段。而且，已经打开的连接还可以避免慢启动的拥塞适应阶段，以便更快速地进行数据的传输。

4.5.1 持久以及并行连接

我们看到，并行连接可以提高复合页面的传输速度。但并行连接也有一些缺点。

- 每个事务都会打开 / 关闭一条新的连接，会耗费时间和带宽。
- 由于 TCP 慢启动特性的存在，每条新连接的性能都会有所降低。
- 可打开的并行连接数量实际上是有限的。

持久连接有一些比并行连接更好的地方。持久连接降低了时延和连接建立的开销，将连接保持在已调谐状态，而且减少了打开连接的潜在数量。但是，管理持久连接时要特别小心，不然就会累积出大量的空闲连接，耗费本地以及远程客户端和服务器上的资源。

持久连接与并行连接配合使用可能是最高效的方式。现在，很多 Web 应用程序都会打开少量的并行连接，其中的每一个都是持久连接。持久连接有两种类型：比较老

注 15：渐进式图片会先显示低分辨率的近似图形，然后再逐渐增加图片的分辨率，而随着渐进式图片应用的逐步增加，这种效果就更加明显了。

的 HTTP/1.0+“keep-alive”连接，以及现代的 HTTP/1.1“persistent”连接。在接下来的几节中我们将对这两种类型进行介绍。

4.5.2 HTTP/1.0+ keep-alive连接

大约从 1996 年开始，很多 HTTP/1.0 浏览器和服务器都进行了扩展，以支持一种被称为 keep-alive 连接的早期实验型持久连接。这些早期的持久连接受到了一些互操作性设计方面问题的困扰，这些问题在后期的 HTTP/1.1 版本中都得到了修正，但很多客户端和服务器仍然在使用这些早期的 keep-alive 连接。

图 4-13 显示了 keep-alive 连接的一些性能优点，图中将在串行连接上实现 4 个 HTTP 事务的时间线与在一条持久连接上实现同样事务所需的时间线进行了比较。由于去除了进行连接和关闭连接的开销，所以时间线有所缩减。[16]

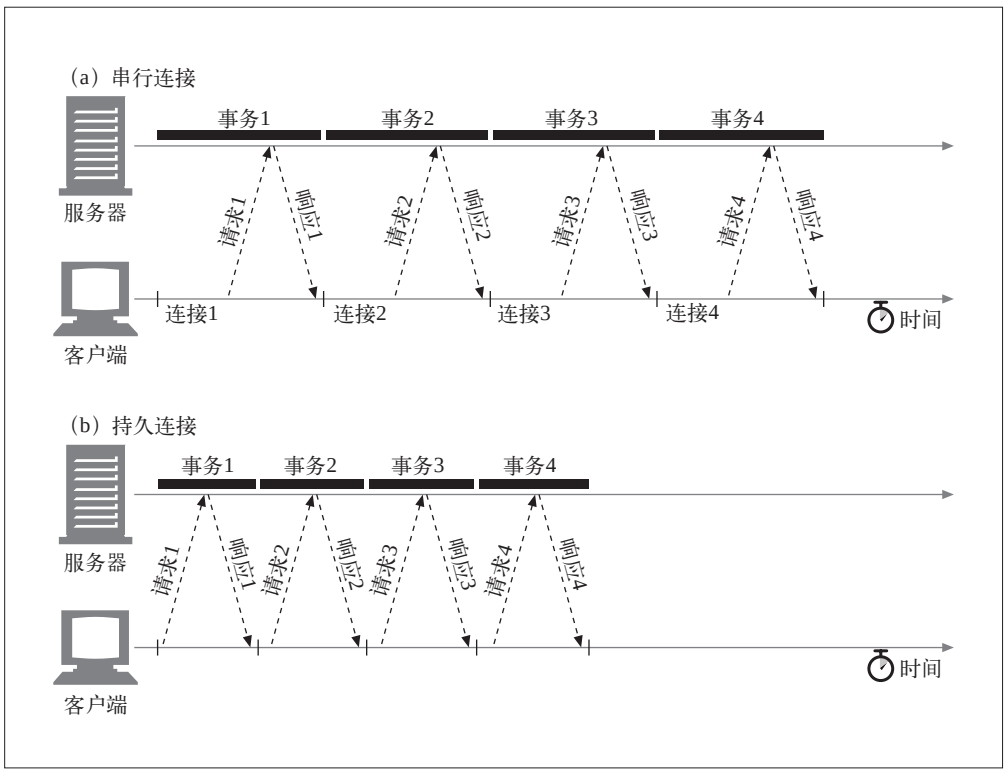


图 4-13　4 个事务（串行与持久连接）

注 16：由于去除了慢启动阶段，请求和响应时间可能也有缩减。这种性能收益在图中没有显示出来。

4.5.3 Keep-Alive操作

keep-alive 已经不再使用了，而且在当前的 HTTP/1.1 规范中也没有对它的说明了。但浏览器和服务器对 keep-alive 握手的使用仍然相当广泛，因此，HTTP 的实现者应该做好与之进行交互操作的准备。现在我们来快速浏览一下 keep-alive 的操作。对 keep-alive 握手更详细的解释请参见较早的 HTTP/1.1 规范版本（比如 RFC 2068）。

实现 HTTP/1.0 keep-alive 连接的客户端可以通过包含 `Connection: Keep-Alive` 首部请求将一条连接保持在打开状态。

如果服务器愿意为下一条请求将连接保持在打开状态，就在响应中包含相同的首部（参见图 4-14）。如果响应中没有 `Connection: Keep-Alive` 首部，客户端就认为服务器不支持 keep-alive，会在发回响应报文之后关闭连接。

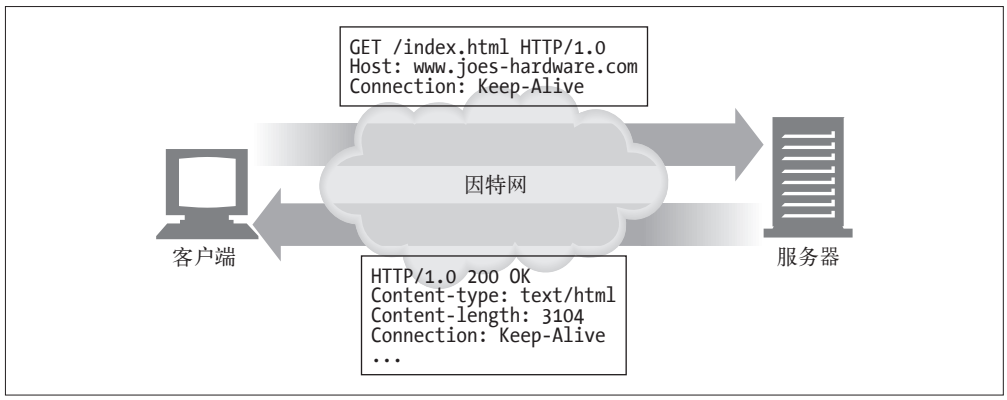


图 4-14　HTTP/1.0 keep-alive 事务首部的握手过程

4.5.4 Keep-Alive选项

注意，`keep-Alive` 首部只是请求将连接保持在活跃状态。发出 keep-alive 请求之
92 后，客户端和服务器并不一定会同意进行 keep-alive 会话。它们可以在任意时刻关闭空闲的 keep-alive 连接，并可随意限制 keep-alive 连接所处理事务的数量。

可以用 `Keep-Alive` 通用首部中指定的、由逗号分隔的选项来调节 keep-alive 的行为。

- 参数 `timeout` 是在 `Keep-Alive` 响应首部发送的。它估计了服务器希望将连接保持在活跃状态的时间。这并不是一个承诺值。
- 参数 `max` 是在 `Keep-Alive` 响应首部发送的。它估计了服务器还希望为多少个事务保持此连接的活跃状态。这并不是一个承诺值。

- Keep-Alive 首部还可支持任意未经处理的属性，这些属性主要用于诊断和调试。语法为 name [=value]。

Keep-Alive 首部完全是可选的，但只有在提供 Connection: Keep-Alive 时才能使用它。这里有个 Keep-Alive 响应首部的例子，这个例子说明服务器最多还会为另外 5 个事务保持连接的打开状态，或者将打开状态保持到连接空闲了 2 分钟之后。

```
Connection: Keep-Alive
Keep-Alive: max=5, timeout=120
```

4.5.5 Keep-Alive连接的限制和规则

使用 keep-alive 连接时有一些限制和一些需要澄清的地方。

- 在 HTTP/1.0 中，keep-alive 并不是默认使用的。客户端必须发送一个 Connection: Keep-Alive 请求首部来激活 keep-alive 连接。
- Connection: Keep-Alive 首部必须随所有希望保持持久连接的报文一起发送。如果客户端没有发送 Connection: Keep-Alive 首部，服务器就会在那条请求之后关闭连接。
- 通过检测响应中是否包含 Connection: Keep-Alive 响应首部，客户端可以判断服务器是否会在发出响应之后关闭连接。
- 只有在无需检测到连接的关闭即可确定报文实体主体部分长度的情况下，才能将连接保持在打开状态——也就是说实体的主体部分必须有正确的 Content-Length，有多部件媒体类型，或者用分块传输编码的方式进行了编码。在一条 keep-alive 信道中回送错误的 Content-Length 是很糟糕的事，这样的话，事务处理的另一端就无法精确地检测出一条报文的结束和另一条报文的开始了。
- 代理和网关必须执行 Connection 首部的规则。代理或网关必须在将报文转发出去或将其高速缓存之前，删除在 Connection 首部中命名的所有首部字段以及 Connection 首部自身。
- 严格来说，不应该与无法确定是否支持 Connection 首部的代理服务器建立 keep-alive 连接，以防止出现下面要介绍的哑代理问题。在实际应用中不是总能做到这一点的。
- 从技术上来讲，应该忽略所有来自 HTTP/1.0 设备的 Connection 首部字段（包括 Connection: Keep-Alive），因为它们可能是由比较老的代理服务器误转发的。但实际上，尽管可能会有在老代理上挂起的危险，有些客户端和服务器还是会违反这条规则。
- 除非重复发送请求会产生其他一些副作用，否则如果在客户端收到完整的响应之前连接就关闭了，客户端就一定要做好重试请求的准备。

4.5.6 Keep-Alive和哑代理

我们来仔细看看keep-alive和哑代理中一些比较微妙的问题。Web客户端的Connection: Keep-Alive首部应该只会对这条离开客户端的TCP链路产生影响。这就是将其称作“连接”首部的原因。如果客户端正在与一台Web服务器对话，客户端可以发送一个Connection: Keep-Alive首部来告知服务器它希望保持连接的活跃状态。如果服务器支持keep-alive，就回送一个Connection: Keep-Alive首部，否则就不回送。

1. Connection首部和盲中继

问题出在代理上——尤其是那些不理解Connection首部，而且不知道在沿着转发链路将其发送出去之前，应该将该首部删除的代理。很多老的或简单的代理都是盲中继（blind relay），它们只是将字节从一个连接转发到另一个连接中去，不对Connection首部进行特殊的处理。

假设有一个Web客户端正通过一个作为盲中继使用的哑代理与Web服务器进行对话。图4-15显示的就是这种情形。

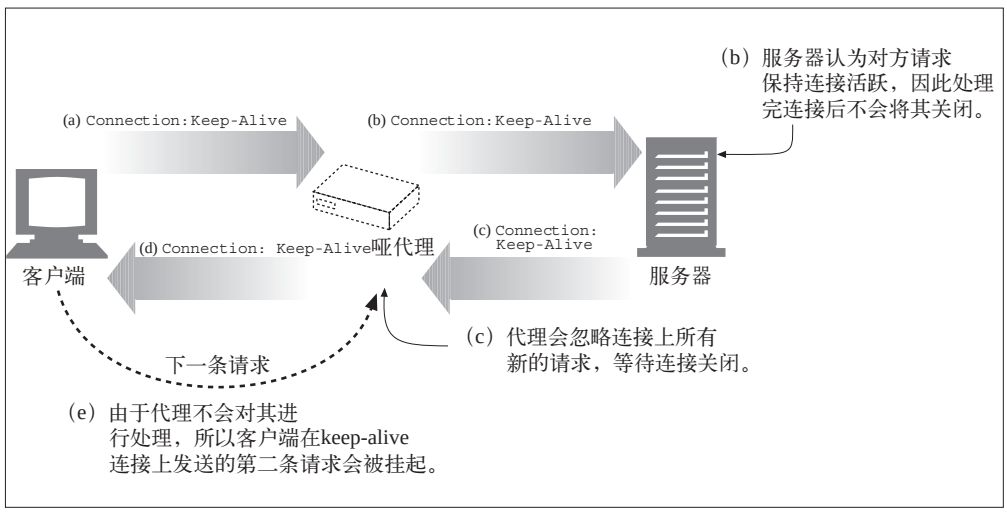


图4-15 keep-alive无法与不支持Connection首部的代理进行互操作

这幅图中发生的情况如下所示。

(1) 在图4-15a中，Web客户端向代理发送了一条报文，其中包含了Connection: Keep-Alive首部，如果可能的话请求建立一条keep-alive连接。客户端等待响应，以确定对方是否认可它对keep-alive信道的请求。

94

(2) 哑代理收到了这条 HTTP 请求，但它并不理解 Connection 首部（只是将其作为一个扩展首部对待）。代理不知道 keep-alive 是什么意思，因此只是沿着转发链路将报文一字不漏地发送给服务器（图 4-15b）。但 Connection 首部是个逐跳首部，只适用于单条传输链路，不应该沿着传输链路向下传输。接下来，就要发生一些很糟糕的事情了。

(3) 在图 4-15b 中，经过中继的 HTTP 请求抵达了 Web 服务器。当 Web 服务器收到经过代理转发的 Connection: Keep-Alive 首部时，会误以为代理（对服务器来说，这个代理看起来就和所有其他客户端一样）希望进行 keep-alive 对话！对 Web 服务器来说这没什么问题——它同意进行 keep-alive 对话，并在图 4-15c 中回送了一个 Connection: Keep-Alive 响应首部。所以，此时 Web 服务器认为它在与代理进行 keep-alive 对话，会遵循 keep-alive 的规则。但代理却对 keep-alive 一无所知。不妙。

(4) 在图 4-15d 中，哑代理将 Web 服务器的响应报文回送给客户端，并将来自 Web 服务器的 Connection: Keep-Alive 首部一起传送过去。客户端看到这个首部，就会认为代理同意进行 keep-alive 对话。所以，此时客户端和服务器都认为它们在进行 keep-alive 对话，但与它们进行对话的代理却对 keep-alive 一无所知。

(5) 由于代理对 keep-alive 一无所知，所以会将收到的所有数据都回送给客户端，然后等待源端服务器关闭连接。但源端服务器会认为代理已经显式地请求它将连接保持在打开状态了，所以不会去关闭连接。这样，代理就会挂在那里等待连接的关闭。

(6) 客户端在图 4-15d 中收到了回送的响应报文时，会立即转向下一条请求，在 keep-alive 连接上向代理发送另一条请求（参见图 4-15e）。而代理并不认为同一条连接上会有其他请求到来，请求被忽略，浏览器就在这里转圈，不会有任何进展了。 95

(7) 这种错误的通信方式会使浏览器一直处于挂起状态，直到客户端或服务器将连接超时，并将其关闭为止。[17]

2. 代理和逐跳首部

为避免此类代理通信问题的发生，现代的代理都绝不能转发 Connection 首部和所有名字出现在 Connection 值中的首部。因此，如果一个代理收到了一个 Connection: Keep-Alive 首部，是不应该转发 Connection 首部，或所有名为 Keep-Alive 的首部的。

注 17：在很多类似的情形下，盲中继和转发的握手信息都会引发问题。

另外，还有几个不能作为 Connection 首部值列出，也不能被代理转发或作为缓存响应使用的首部。其中包括 Proxy-Authenticate、Proxy-Connection、Transfer-Encoding 和 Upgrade。更多信息，请参考 4.3.1 节。

4.5.7 插入Proxy-Connection

Netscape 的浏览器及代理实现者们提出了一个对盲中继问题的变通做法，这种做法并不要求所有的 Web 应用程序支持高版本的 HTTP。这种变通做法引入了一个名为 Proxy-Connection 的新首部，解决了在客户端后面紧跟着一个盲中继所带来的问题——但并没有解决所有其他情况下存在的问题。在显式配置了代理的情况下，现代浏览器都实现了 Proxy-Connection，很多代理都能够理解它。

问题是哑代理盲目地转发 Connection: Keep-Alive 之类的逐跳首部惹出了麻烦。逐跳首部只与一条特定的连接有关，不能被转发。当下游服务器误将转发来的首部作为来自代理自身的请求解释，用它来控制自己的连接时，就会引发问题。

在网景的变通做法是，浏览器会向代理发送非标准的 Proxy-Connection 扩展首部，而不是官方支持的著名的 Connection 首部。如果代理是盲中继，它会将无意义的 Proxy-Connection 首部转发给 Web 服务器，服务器会忽略此首部，不会带来任何问题。但如果代理是个聪明的代理（能够理解持久连接的握手动作），就用一个 Connection 首部取代无意义的 Proxy-Connection 首部，然后将其发送给服务器，以收到预期的效果。

图 4-16a ～图 4-16d 显示了盲中继是如何向 Web 服务器转发 Proxy-Connection
96 首部，而不带来任何问题的，Web 服务器忽略了这个首部，这样在客户端和代理，或者代理和服务器之间就不会建立起 keep-alive 连接了。图 4-16e ～图 4-16h 中那个聪明的代理知道 Proxy-Connection 首部是对 keep-alive 对话的请求，它会发送自己的 Connection: Keep-Alive 首部来建立 keep-alive 连接。

在客户端和服务器之间只有一个代理时可以用这种方案来解决问题。但如图 4-17 所示，如果在哑代理的任意一侧还有一个聪明的代理，这个问题就会再次露头了。

而且，网络中出现“不可见”代理的情况现在变得很常见了，这些代理可以是防火墙、拦截缓存，或者是反向代理服务器的加速器。这些设备对浏览器是不可见的，所以浏览器不会向它们发送 Proxy-Connection 首部。透明的 Web 应用程序正确地实现持久连接是非常重要的。

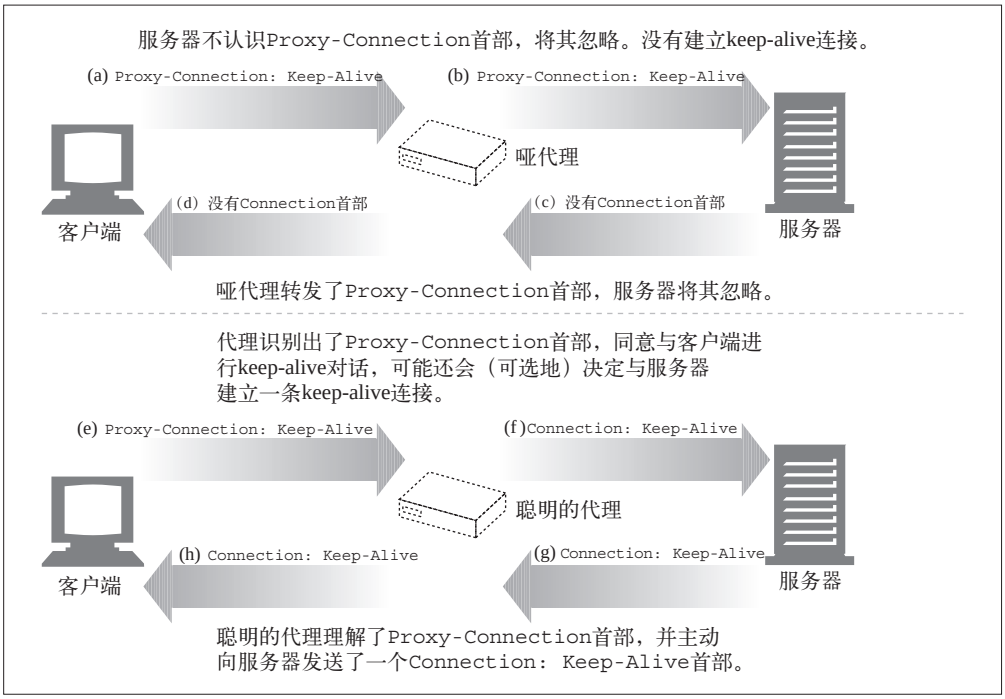


图 4-16　**Proxy-Connection** 首部修正了单个盲中继带来的问题

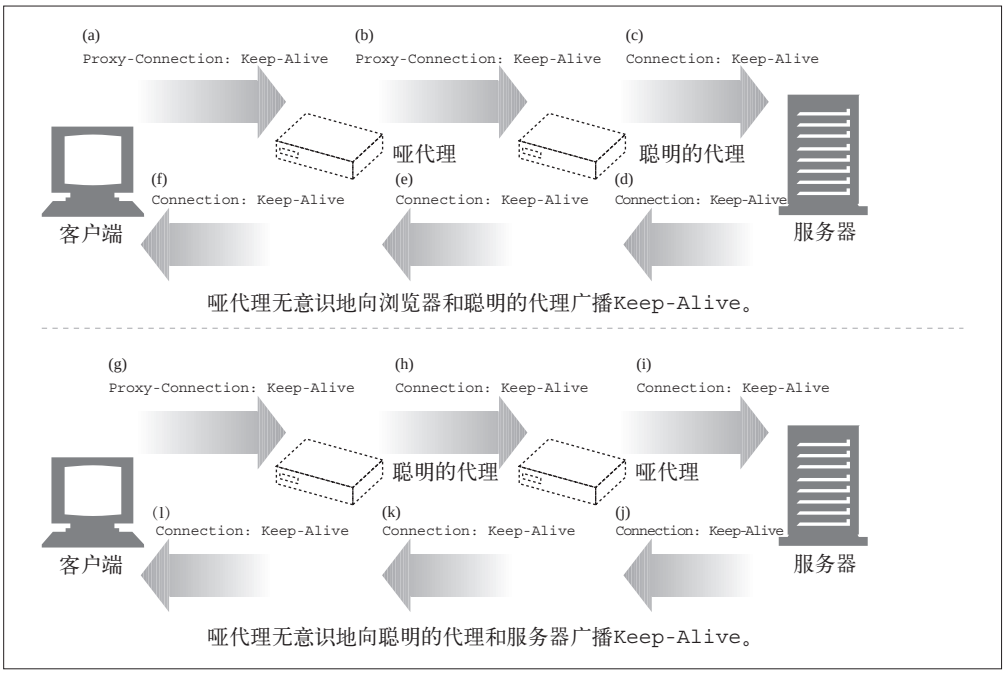


图 4-17　对有多层次代理的情况，**Proxy-Connection** 仍然无法解决问题

4.5.8 HTTP/1.1持久连接

HTTP/1.1 逐渐停止了对 keep-alive 连接的支持，用一种名为*持久连接*（persistent connection）的改进型设计取代了它。持久连接的目的与 keep-alive 连接的目的相
97 同，但工作机制更优一些。

与 HTTP/1.0+ 的 keep-alive 连接不同，HTTP/1.1 持久连接在默认情况下是激活的。除非特别指明，否则 HTTP/1.1 假定所有连接都是持久的。要在事务处理结束之后将连接关闭，HTTP/1.1 应用程序必须向报文中显式地添加一个 `Connection: close` 首部。这是与以前的 HTTP 协议版本很重要的区别，在以前的版本中，keep-alive 连接要么是可选的，要么根本就不支持。

HTTP/1.1 客户端假定在收到响应后，除非响应中包含了 `Connection: close` 首部，不然 HTTP/1.1 连接就仍维持在打开状态。但是，客户端和服务器仍然可以随时关闭空闲的连接。不发送 `Connection: close` 并不意味着服务器承诺永远将连接保持在打开状态。

4.5.9 持久连接的限制和规则

在持久连接的使用中有以下限制和需要澄清的问题。

- 发送了 `Connection: close` 请求首部之后，客户端就无法在那条连接上发送更多的请求了。
- 如果客户端不想在连接上发送其他请求了，就应该在最后一条请求中发送一个 `Connection: close` 请求首部。
- 只有当连接上所有的报文都有正确的、自定义报文长度时——也就是说，实体主体部分的长度都和相应的 `Content-Length` 一致，或者是用分块传输编码方式
98 编码的——连接才能持久保持。
- HTTP/1.1 的代理必须能够分别管理与客户端和服务器的持久连接——每个持久连接都只适用于一跳传输。
- （由于较老的代理会转发 `Connection` 首部，所以）HTTP/1.1 的代理服务器不应该与 HTTP/1.0 客户端建立持久连接，除非它们了解客户端的处理能力。实际上，这一点是很难做到的，很多厂商都违背了这一原则。
- 尽管服务器不应该试图在传输报文的过程中关闭连接，而且在关闭连接之前至少应该响应一条请求，但不管 `Connection` 首部取了什么值，HTTP/1.1 设备都可以在任意时刻关闭连接。

- HTTP/1.1 应用程序必须能够从异步的关闭中恢复出来。只要不存在可能会累积起来的副作用，客户端都应该重试这条请求。
- 除非重复发起请求会产生副作用，否则如果在客户端收到整条响应之前连接关闭了，客户端就必须要重新发起请求。
- 一个用户客户端对任何服务器或代理最多只能维护两条持久连接，以防服务器过载。代理可能需要更多到服务器的连接来支持并发用户的通信，所以，如果有 *N* 个用户试图访问服务器的话，代理最多要维持 2*N* 条到任意服务器或父代理的连接。

4.6 管道化连接

HTTP/1.1 允许在持久连接上可选地使用*请求管道*。这是相对于 keep-alive 连接的又一性能优化。在响应到达之前，可以将多条请求放入队列。当第一条请求通过网络流向地球另一端的服务器时，第二条和第三条请求也可以开始发送了。在高时延网络条件下，这样做可以降低网络的环回时间，提高性能。

图 4-18a-c 显示了持久连接是怎样消除 TCP 连接时延，以及管道化请求（参见图 4-18c）是如何消除传输时延的。

对管道化连接有几条限制。

- 如果 HTTP 客户端无法确认连接是持久的，就不应该使用管道。
- 必须按照与请求相同的顺序回送 HTTP 响应。HTTP 报文中没有序列号标签，因此如果收到的响应失序了，就没办法将其与请求匹配起来了。
- HTTP 客户端必须做好连接会在任意时刻关闭的准备，还要准备好重发所有未完成的管道化请求。如果客户端打开了一条持久连接，并立即发出了 10 条请求，服务器可能在只处理了，比方说，5 条请求之后关闭连接。剩下的 5 条请求会失败，客户端必须能够应对这些过早关闭连接的情况，重新发出这些请求。 99
- HTTP 客户端不应该用管道化的方式发送会产生副作用的请求（比如 POST）。总之，出错的时候，管道化方式会阻碍客户端了解服务器执行的是一系列管道化请求中的哪一些。由于无法安全地重试 POST 这样的非幂等请求，所以出错时，就存在某些方法永远不会被执行的风险。 100

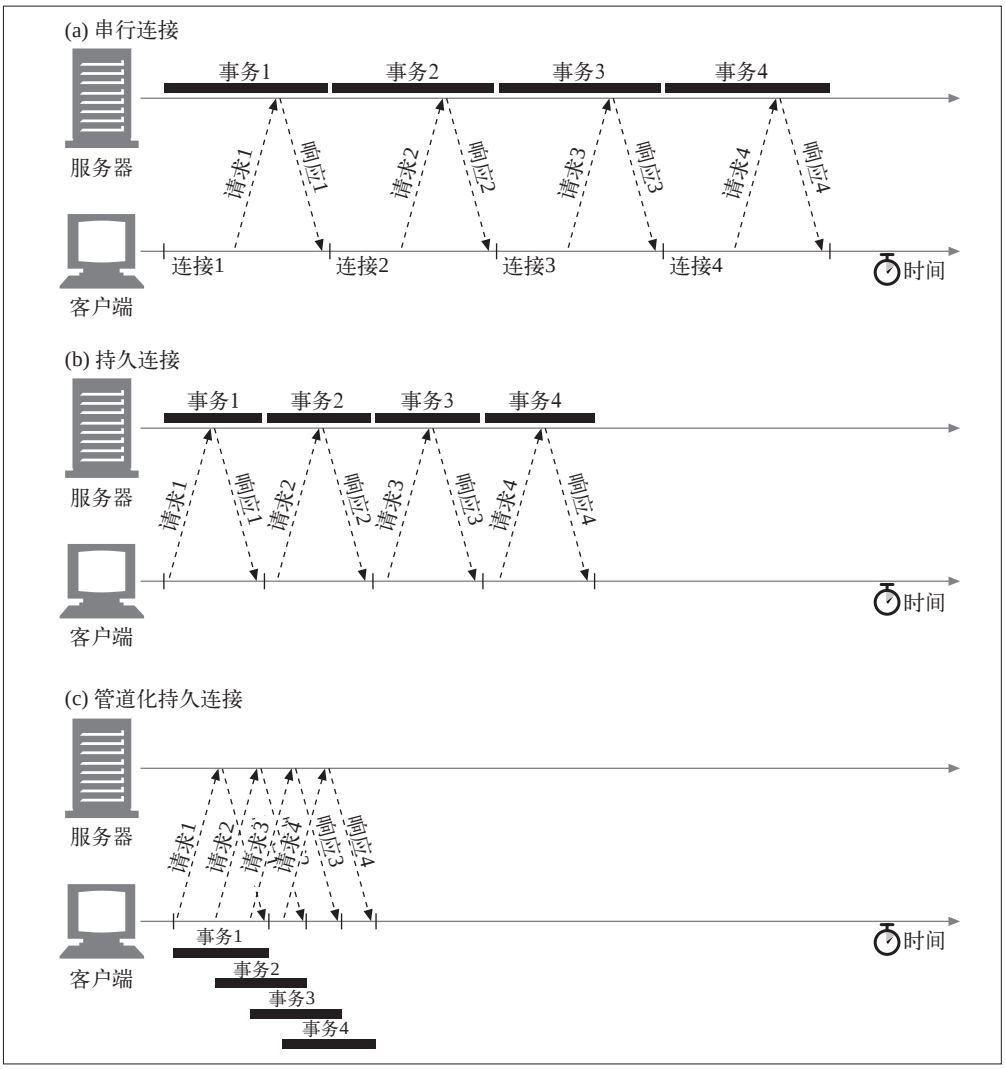


图 4-18　4 个事务（管道化连接）

4.7　关闭连接的奥秘

连接管理——尤其是知道在什么时候以及如何去关闭连接——是 HTTP 的实用魔法之一。这个问题比很多开发者起初意识到的复杂一些，而且没有多少资料涉及这个问题。

4.7.1　“任意”解除连接

所有 HTTP 客户端、服务器或代理都可以在任意时刻关闭一条 TCP 传输连接。通常会在一条报文结束时关闭连接，[18] 但出错的时候，也可能在首部行的中间，或其他

奇怪的地方关闭连接。

对管道化持久连接来说，这种情形是很常见的。HTTP 应用程序可以在经过任意一段时间之后，关闭持久连接。比如，在持久连接空闲一段时间之后，服务器可能会决定将其关闭。

但是，服务器永远都无法确定在它关闭“空闲”连接的那一刻，在线路那一头的客户端有没有数据要发送。如果出现这种情况，客户端就会在写入半截请求报文时发现出现了连接错误。

4.7.2 Content-Length及截尾操作

每条 HTTP 响应都应该有精确的 Content-Length 首部，用以描述响应主体的尺寸。一些老的 HTTP 服务器会省略 Content-Length 首部，或者包含错误的长度指示，这样就要依赖服务器发出的连接关闭来说明数据的真实末尾。

客户端或代理收到一条随连接关闭而结束的 HTTP 响应，且实际传输的实体长度与 Content-Length 并不匹配（或没有 Content-Length）时，接收端就应该质疑长度的正确性。

如果接收端是个缓存代理，接收端就不应该缓存这条响应（以降低今后将潜在的错误报文混合起来的可能）。代理应该将有问题的报文原封不动地转发出去，而不应该试图去“校正”Content-Length，以维护语义的透明性。

4.7.3 连接关闭容限、重试以及幂等性

即使在非错误情况下，连接也可以在任意时刻关闭。HTTP 应用程序要做好正确处理非预期关闭的准备。如果在客户端执行事务的过程中，传输连接关闭了，那么，除非事务处理会带来一些副作用，否则客户端就应该重新打开连接，并重试一次。对管道化连接来说，这种情况更加严重一些。客户端可以将大量请求放入队列中排队，但源端服务器可以关闭连接，这样就会留下大量未处理的请求，需要重新调度。

101

副作用是很重要的问题。如果在发送出一些请求数据之后，收到返回结果之前，连接关闭了，客户端就无法百分之百地确定服务器端实际激活了多少事务。有些事务，比如 GET 一个静态的 HTML 页面，可以反复执行多次，也不会有什么变化。而其他一些事务，比如向一个在线书店 POST 一张订单，就不能重复执行，不然会有下多张订单的危险。

注 18：除非服务器怀疑出现了客户端或网络故障，否则就不应该在请求的中间关闭连接。

如果一个事务，不管是执行一次还是很多次，得到的结果都相同，这个事务就是幂等的。实现者们可以认为 GET、HEAD、PUT、DELETE、TRACE 和 OPTIONS 方法都共享这一特性。[19] 客户端不应该以管道化方式传送非幂等请求（比如 POST）。否则，传输连接的过早终止就会造成一些不确定的后果。要发送一条非幂等请求，就需要等待来自前一条请求的响应状态。

尽管用户 Agent 代理可能会让操作员来选择是否对请求进行重试，但一定不能自动重试非幂等方法或序列。比如，大多数浏览器都会在重载一个缓存的 POST 响应时提供一个对话框，询问用户是否希望再次发起事务处理。

4.7.4 正常关闭连接

如图 4-19 所示，TCP 连接是双向的。TCP 连接的每一端都有一个输入队列和一个输出队列，用于数据的读或写。放入一端输出队列中的数据最终会出现在另一端的输入队列中。

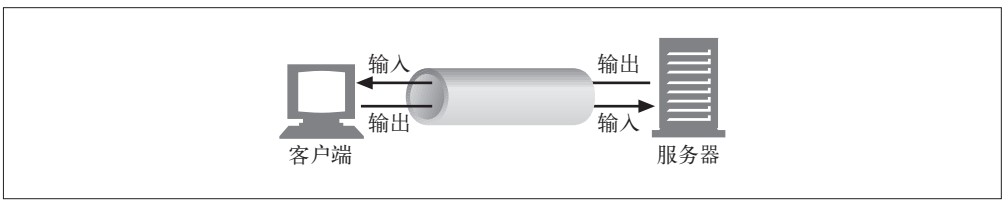


图 4-19 TCP 连接是双向的

1. 完全关闭与半关闭

应用程序可以关闭 TCP 输入和输出信道中的任意一个，或者将两者都关闭了。套
102 接字调用 `close()` 会将 TCP 连接的输入和输出信道都关闭了。这被称作“完全关闭”，如图 4-20a 所示。还可以用套接字调用 `shutdown()` 单独关闭输入或输出信道。这被称为“半关闭”，如图 4-20b 所示。

2. TCP关闭及重置错误

简单的 HTTP 应用程序可以只使用完全关闭。但当应用程序开始与很多其他类型的 HTTP 客户端、服务器和代理进行对话且开始使用管道化持久连接时，使用半关闭来防止对等实体收到非预期的写入错误就变得很重要了。

注 19：基于 GET 构建动态表单的管理者们要确保这些表单是幂等的。

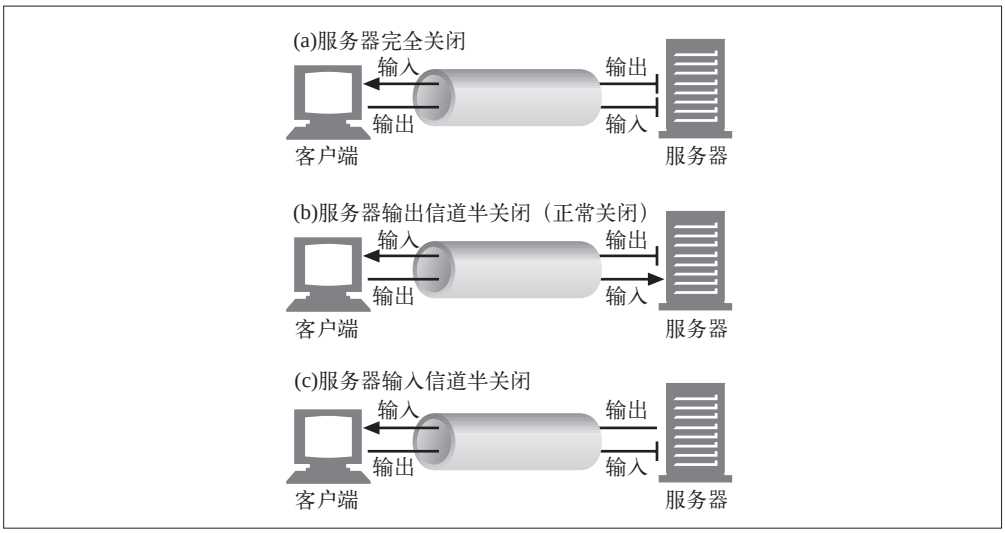


图 4-20　完全关闭和半关闭

总之，关闭连接的输出信道总是很安全的。连接另一端的对等实体会在从其缓冲区中读出所有数据之后收到一条通知，说明流结束了，这样它就知道你将连接关闭了。

关闭连接的输入信道比较危险，除非你知道另一端不打算再发送其他数据了。如果另一端向你已关闭的输入信道发送数据，操作系统就会向另一端的机器回送一条 TCP“连接被对端重置”的报文，如图 4-21 所示。大部分操作系统都会将这种情况作为很严重的错误来处理，删除对端还未读取的所有缓存数据。对管道化连接来说，这是非常糟糕的事情。

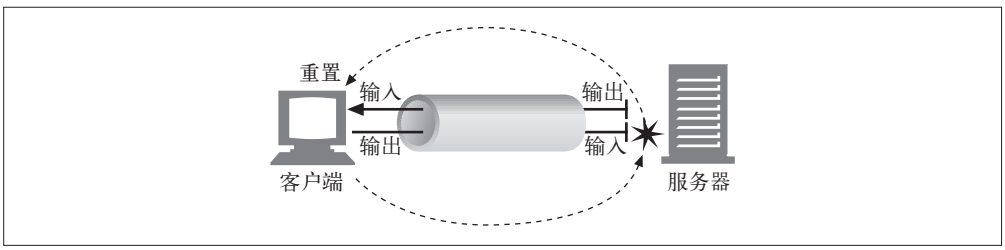


图 4-21　将数据传送到已关闭连接时会产生“连接被对端重置”错误

比如你已经在一条持久连接上发送了 10 条管道式请求了，响应也已经收到了，正在操作系统的缓冲区中存着呢（但应用程序还未将其读走）。现在，假设你发送了第 11 条请求，但服务器认为你使用这条连接的时间已经够长了，决定将其关闭。那么你的第 11 条请求就会被发送到一条已关闭的连接上去，并会向你回送一条重置信息。这个重置信息会清空你的输入缓冲区。

103

当你最终要去读取数据的时候，会得到一个连接被对端重置的错误，已缓存的未读响应数据都丢失了，尽管其中的大部分都已经成功抵达你的机器了。

3. 正常关闭

HTTP 规范建议，当客户端或服务器突然要关闭一条连接时，应该“正常地关闭传输连接”，但它并没有说明应该如何去做。

总之，实现正常关闭的应用程序首先应该关闭它们的输出信道，然后等待连接另一端的对等实体关闭它的输出信道。当两端都告诉对方它们不会再发送任何数据（比如关闭输出信道）之后，连接就会被完全关闭，而不会有重置的危险。

但不幸的是，无法确保对等实体会实现半关闭，或对其进行检查。因此，想要正常关闭连接的应用程序应该先半关闭其输出信道，然后周期性地检查其输入信道的状态（查找数据，或流的末尾）。如果在一定的时间区间内对端没有关闭输入信道，应用程序可以强制关闭连接，以节省资源。

4.8 更多信息

到这里我们对 HTTP 通道事务处理的介绍就结束了。更多有关 TCP 性能和 HTTP 连接管理功能的内容请参见下列参考资源。

4.8.1 HTTP连接

- http://www.ietf.org/rfc/rfc2616.txt
  RFC 2616，“超文本传输协议——HTTP/1.1”是 HTTP/1.1 的官方规范；解释了并行、持久和管道式 HTTP 连接的使用，以及用于实现这些连接的 HTTP 首部字段。此文档并未涵盖对底层 TCP 连接的正确使用。

104

- http://www.ietf.org/rfc/rfc2068.txt
  RFC 2068 是 HTTP/1.1 协议的 1997 年的版本。其中包含了 RFC 2616 中没有的、对 HTTP/1.0+ keep-alive 连接的解释。

- http://www.ics.uci.edu/pub/ietf/http/draft-ietf-http-connection-00.txt
  这个过期的因特网草案“HTTP Connection Management”，（“HTTP 连接管理”）探讨了 HTTP 连接管理面临的问题。

4.8.2 HTTP性能问题

- http://www.w3.org/Protocols/HTTP/Performance/

这个名为“HTTP Performance Overview”（“HTTP 性能概览”）的 W3C Web 页面包含了几篇与 HTTP 性能和连接管理有关的文章和一些工具。

- http://www.w3.org/Protocols/HTTP/1.0/HTTPPerformance.html
  这个由 Simon Spero 制作的简短备忘录“Analysis of HTTP Performance Problems”（“HTTP 性能问题分析”）是最早（1994 年）对 HTTP 连接性能进行评估的文献之一。对早期由于缺乏连接建立、慢启动和连接共享所造成的影响进行了一些性能测试，这个备忘录给出了一些测试结果。

- ftp://gatekeeper.dec.com/pub/DEC/WRL/research-reports/WRL-TR-95.4.pdf
  “The Case for Persistent-Connection HTTP.”（“持久连接 HTTP 的实例。”）

- http://www.isi.edu/lsam/publications/phttp_tcp_interactions/paper.html
  “Performance Interactions Between P-HTTP and TCP”Implementations（“P-HTTP 和 TCP 实现之间的性能交互”）。

- http://www.sun.com/sun-on-net/performance/tcp.slowstart.html
  “TCP Slow Start Tuning for Solaris”（“Solaris 的 TCP 慢启动调试”）是 Sun 微系统公司的一个 Web 页面，介绍了 TCP 慢启动带来的一些实际的影响。即使工作在不同的操作系统上，读一读这份资料也是有所帮助的。

4.8.3 TCP/IP

下面三本 W. Richard Stevens 的书都非常棒，详细介绍了 TCP/IP 的工程问题。对使用 TCP 的人来说尤其有用：

- *TCP Illustrated, Volume 1: The Protocols*[20]（《TCP 详解，卷 1：协议》）
  W. Richard Stevens，Addison Wesley 公司出版。

- *UNIX Network Programming, Volume 1: Networking APIs*[21]（《UNIX 网络编程，卷 1：套接字联网 API（第 3 版）》）
  W. Richard Stevens，Prentice-Hall 公司出版。

- *UNIX Network Programming, Volume 2: The Implementation*[22]（《UNIX 网络编程，卷 2：进程间通信（第 2 版）》）
  W. Richard Stevens，Prentice-Hall 公司出版。

注 20：本书影印版已由人民邮电出版社出版。（编者注）
注 21~22：两本书中文版已由人民邮电出版社出版。（编者注）

下面的文章和规范介绍了 TCP/IP 及影响其性能的特性。其中有些规范已经有 20 多年的历史了，鉴于 TCP/IP 在全球范围内的成功，很可能已经可以将其归为历史宝藏了。

- http://www.acm.org/sigcomm/ccr/archive/2001/jan01/ccr-200101-mogul.pdf
  在“Rethinking the TCP Nagle Algorithm”（“对 TCP Nagle 算法的反思”）一文中，Jeff Mogul 和 Greg Minshall 提出了 Nagle 算法的一种现代视角，概括了哪些应用程序应该，哪些不应该使用这个算法，并提出了几条改进意见。

- http://www.ietf.org/rfc/rfc2001.txt
  RFC 2001，“TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms”（“TCP 慢启动、避免拥塞、快速重传以及快速恢复算法”）定义了 TCP 慢启动算法。

- http://www.ietf.org/rfc/rfc1122.txt
  RFC 1122，“Requirements for Internet Hosts—Communication Layers”（“对因特网主机的要求——通信层”）讨论了 TCP 确认和延迟确认。

- http://www.ietf.org/rfc/rfc896.txt
  RFC 896，“Congestion Control in IP/TCP Internetworks”（“IP/TCP 网络间的拥塞控制”）是 John Nagle 于 1984 年发布的。描述了 TCP 拥塞控制的必要性。介绍了现在称为“Nagle 算法”的算法。

- http://www.ietf.org/rfc/rfc0813.txt
  RFC 813，“Window and Acknowledgement Strategy in TCP”（“TCP 中的窗口和确认机制”）是一个早些年（1982 年）制定的规范，它描述了 TCP 窗口和确认的实现机制，解释了延迟确认技术的早期技术。

- http://www.ietf.org/rfc/rfc0793.txt
  RFC 793，“Transmission Control Protocol”（“传输控制协议”），是 Jon Postel 于 1981 年给出的 TCP 协议经典定义。

106

第二部分

HTTP结构

第二部分的 6 章主要介绍了 HTTP 服务器、代理、缓存、网关和机器人应用程序，这些都是 Web 系统架构的构造模块。

- 第 5 章概述了 Web 服务器结构。
- 第 6 章详细介绍了 HTTP 代理服务器，它们是连接 HTTP 客户端的中间服务器，是 HTTP 服务和控制的平台。
- 第 7 章深入研究了 Web 的缓存机制。缓存是通过对常用文档进行本地复制来提高性能、减少流量的设备。
- 第 8 章介绍了一些应用程序，通过这些程序，HTTP 就可以与使用不同协议（比如 SSL 加密协议）的软件进行互操作了。
- 第 9 章介绍了 Web 客户端，结束了 HTTP 架构之旅。
- 第 10 章涵盖了 HTTP 未来发展的一些主题，特别介绍了 HTTP-NG 技术。

第5章

Web服务器

Web 服务器每天会分发出数十亿的 Web 页面。这些页面可以告诉你天气情况，装载在线商店的购物车，还能帮你找到许久未联系的高中同学。Web 服务器是万维网的骨干。本章将介绍以下话题。

- 对多种使用不同类型软硬件的 Web 服务器进行调查。
- 介绍如何用 Perl 编写简单的诊断性 Web 服务器。
- 一步一步地解释 Web 服务器是如何处理 HTTP 事务的。

为了对问题进行具体的说明，例子中使用了 Apache Web 服务器及其配置选项。

5.1 各种形状和尺寸的Web服务器

Web 服务器会对 HTTP 请求进行处理并提供响应。术语“Web 服务器”可以用来表示 Web 服务器的软件，也可以用来表示提供 Web 页面的特定设备或计算机。

Web 服务器有着不同的风格、形状和尺寸。有普通的 10 行 Perl 脚本的 Web 服务器、50MB 的安全商用引擎以及极小的卡上服务器。但不管功能有何差异，所有的 Web 服务器都能够接收请求资源的 HTTP 请求，将内容回送给客户端（参见图 1-5）。

5.1.1 Web服务器的实现

Web 服务器实现了 HTTP 和相关的 TCP 连接处理。负责管理 Web 服务器提供的资源，以及对 Web 服务器的配置、控制及扩展方面的管理。

Web 服务器逻辑实现了 HTTP 协议、管理着 Web 资源，并负责提供 Web 服务器的
109 管理功能。Web 服务器逻辑和操作系统共同负责管理 TCP 连接。底层操作系统负责管理底层计算机系统的硬件细节，并提供了 TCP/IP 网络支持、负责装载 Web 资源的文件系统以及控制当前计算活动的进程管理功能。

Web 服务器有各种不同的形式。

- 可以在标准的计算机系统上安装并运行通用的软件 Web 服务器。
- 如果不想那么麻烦地去安装软件，可以买一台 Web 服务器设备，通常会是一台安装在时髦机架上的计算机，里面的软件会预装并配置好。
- 随着微处理器奇迹般地出现，有些公司甚至可以在少量计算机芯片上实现嵌入式 Web 服务器，使其成为完美的（便携式）消费类设备管理控制台。

我们分别来看看这些实现方式。

5.1.2 通用软件Web服务器

通用软件 Web 服务器都运行在标准的、有网络功能的计算机系统上。可以选择开源软件（比如 Apache 或 W3C 的 Jigsaw）或者商业软件（比如微软和 iPlanet 的 Web 服务器）。基本上所有的计算机和操作系统中都有可用的 Web 服务器软件。

尽管不同类型的 Web 服务器程序有数万个（包括定制的和特殊用途的 Web 服务器），但大多数 Web 服务器软件都来自少数几个组织。

2002 年 2 月，Netcraft 调查（http://www.netcraft.com/survey/）显示有三家厂商主宰了公共因特网 Web 服务器市场（参见图 5-1）。

- 免费的 Apache 软件占据了所有因特网 Web 服务器中大约 60% 的市场。
- 微软的 Web 服务器占据了另外 30%。
- Sun 的 iPlanet 占据了另外 3%。

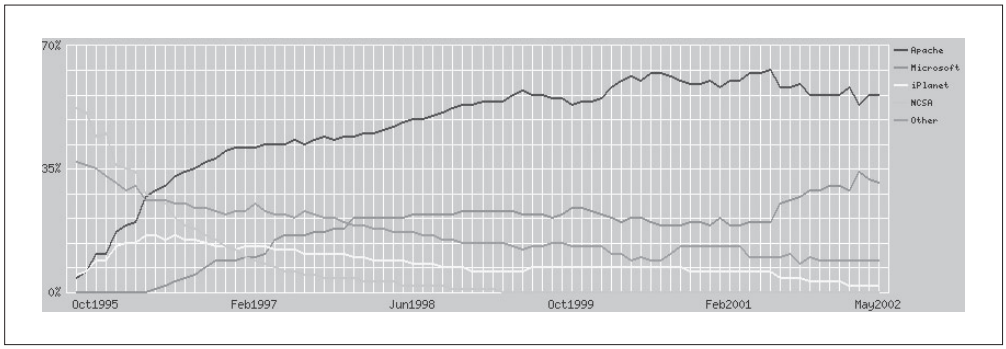


图 5-1 Netcraft 的自动化调查估计的 Web 服务器市场份额

但这些数据也不能尽信，通常大家都认为 Netcraft 调查会夸大 Apache 软件的优势。首先，在调查计算服务器的时候没有考虑其流行程度。各大 ISP 的代理服务器访问研究表明，Apache 服务器提供的页面数量远小于 60%，但仍然超过了微软和 Sun 的 iPlanet。然而，据说微软和 iPlanet 服务器在公司企业中要比 Apache 更受欢迎。

5.1.3 Web服务器设备

Web 服务器设备（Web server appliance）是预先打包好的软硬件解决方案。厂商会在他们选择的计算机平台上预先安装好软件服务器，并将软件配置好。下面是一些 Web 服务器设备的例子：

- Sun/Cobalt RaQ Web 设备（http://www.cobalt.com）；
- 东芝的 Magnia SG10（http://www.toshiba.com）；

- IBM 的 Whistle Web 服务器设备（http://www.whistle.com）。

应用解决方案不再需要安装及配置软件，通常可以极大地简化管理工作。但是，Web 服务器通常不太灵活，特性不太丰富，而且服务器硬件也不太容易重用或升级。

5.1.4 嵌入式Web服务器

嵌入式服务器（embeded server）是要嵌入到消费类产品（比如打印机或家用设备）中去的小型 Web 服务器。嵌入式 Web 服务器允许用户通过便捷的 Web 浏览器接口来管理其消费者设备。

有些嵌入式 Web 服务器甚至可以在小于一平方英寸的空间内实现，但通常只能提供最小特性功能集。下面是两种非常小的嵌入式 Web 服务器实例：

- IPic 火柴头大小的 Web 服务器（http://www-ccs.cs.umass.edu/~shri/iPic.html）；
- NetMedia SitePlayer SP1 以太网 Web 服务器（http://www.siteplayer.com）。

5.2 最小的Perl Web服务器

要构建一个特性完备的 HTTP 服务器，是需要做一些工作的。Apache Web 服务器的内核有超过 50 000 行的代码，那些可选处理模块的代码量更是远远超过这个数字。

这个软件所要做的就是支持 HTTP/1.1 的各种特性：丰富的资源支持、虚拟主机、访问控制、日志记录、配置、监视和性能特性。在这里，可以用少于 30 行的 Perl 代码来创建一个最小的可用 HTTP 服务器。我们来看看这是怎么实现的。

111

例 5-1 显示了一个名为 type-o-serve 的小型 Perl 程序。这个程序是个很有用的诊断工具，可以用来测试与客户端和代理的交互情况。与所有 Web 服务器一样，type-o-serve 会等待 HTTP 连接。只要 type-o-serve 收到了请求报文，就会将报文打印在屏幕上，然后等待用户输入（或粘贴）一条响应报文，并将其回送给客户端。通过这种方式，type-o-serve 假扮成一台 Web 服务器，记录下确切的 HTTP 请求报文，并允许用户回送任意的 HTTP 响应报文。

这个简单的 type-o-serve 实用程序并没有实现大部分的 HTTP 功能，但它是一种很有用的工具，产生服务器响应报文的方式与 Telnet 产生客户端请求报文的方式相同（参见例 5-1）。可以从 http://www.http-guide.com/tools/type-o-serve.pl 上下载 type-o-serve 程序。

例 5-1 type-o-serve——用于 HTTP 调试的最小型 Perl Web 服务器

```
#!/usr/bin/perl

use Socket;
use Carp;
use FileHandle;

# (1) use port 8080 by default, unless overridden on command line
$port = (@ARGV ? $ARGV[0] : 8080);

# (2) create local TCP socket and set it to listen for connections
$proto = getprotobyname('tcp');
socket(S, PF_INET, SOCK_STREAM, $proto) || die;
setsockopt(S, SOL_SOCKET, SO_REUSEADDR, pack("l", 1)) || die;
bind(S, sockaddr_in($port, INADDR_ANY)) || die;
listen(S, SOMAXCONN) || die;

# (3) print a startup message
printf("    <<<Type-O-Serve Accepting on Port %d>>>\n\n",$port);

while (1)
{
    # (4) wait for a connection C
    $cport_caddr = accept(C, S);
    ($cport,$caddr) = sockaddr_in($cport_caddr);
    C->autoflush(1);

    # (5) print who the connection is from
    $cname = gethostbyaddr($caddr,AF_INET);
    printf("    <<<Request From '%s'>>>\n",$cname);

    # (6) read request msg until blank line, and print on screen
    while ($line = <C>)
    {
        print $line;
        if ($line =~ /^\r/) { last; }
    }

    # (7) prompt for response message, and input response lines,
    #     sending response lines to client, until solitary "."
    printf("    <<<Type Response Followed by '.'>>>\n");

    while ($line = <STDIN>)
    {
        $line =~ s/\r//;
        $line =~ s/\n//;
        if ($line =~ /^\./) { last; }
        print C $line . "\r\n";
    }
    close(C);
}
```

图 5-2 显示了 Joe 的五金商店的管理员是如何用 type-o-serve 来测试 HTTP 通信的。

- 首先，管理员启动了 type-o-serve 诊断服务器，在一个特定的端口上监听。由于 Joe 的五金商店已经有一个产品化的 Web 服务器在监听 80 端口了，所以管理员用下面这条命令在端口 8080（可以选择任意未用端口）上启动了 type-o-serve 服务：

```
% type-o-serve.pl 8080
```

- 只要 type-o-serve 开始运行了，就可以将浏览器指向这个 Web 服务器。在图 5-2 中，浏览器指向了 http://www.joes-hardware.com:8080/foo/bar/blah.txt。
- type-o-serve 程序收到来自浏览器的 HTTP 请求报文，并将 HTTP 请求报文的内容打印在屏幕上。然后 type-o-serve 诊断工具会等待用户输入一条简单的响应报文，后面跟着只有一个句号的空行。
- type-o-serve 将 HTTP 响应报文回送给浏览器，浏览器会显示响应报文的主体。

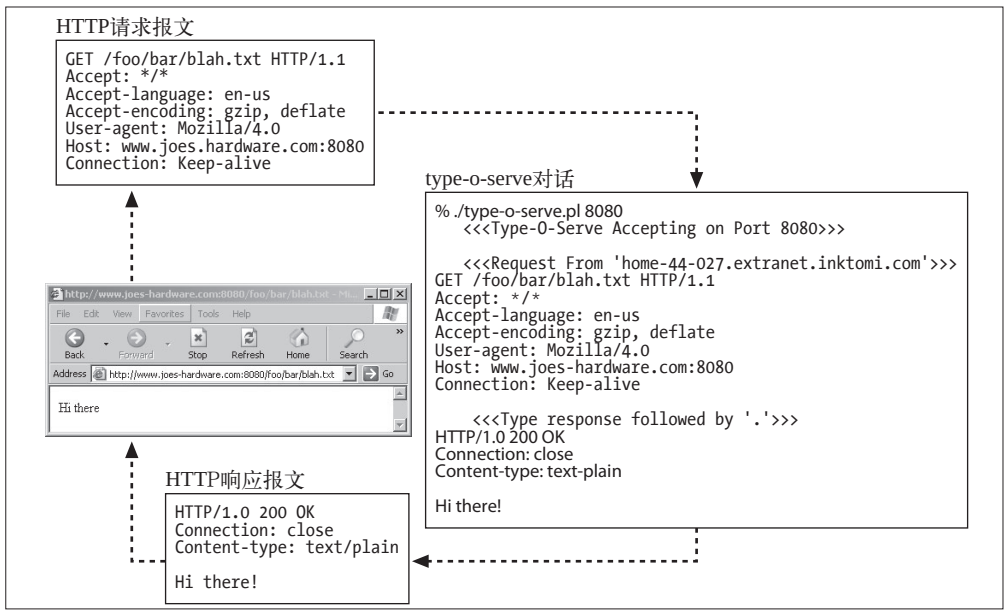


图 5-2　type-o-serve 实用程序让用户输入服务器响应，将其回送给客户端

5.3　实际的Web服务器会做些什么

例 5-1 显示的 Perl 服务器是一个 Web 服务器的小例子。最先进的商用 Web 服务器要比它复杂得多，但它们确实执行了几项同样的任务，如图 5-3 所示。

(1) 建立连接——接受一个客户端连接，或者如果不希望与这个客户端建立连接，就将其关闭。

(2) 接收请求——从网络中读取一条 HTTP 请求报文。
(3) 处理请求——对请求报文进行解释，并采取行动。
(4) 访问资源——访问报文中指定的资源。
(5) 构建响应——创建带有正确首部的 HTTP 响应报文。
(6) 发送响应——将响应回送给客户端。
(7) 记录事务处理过程——将与已完成事务有关的内容记录在一个日志文件中。 113

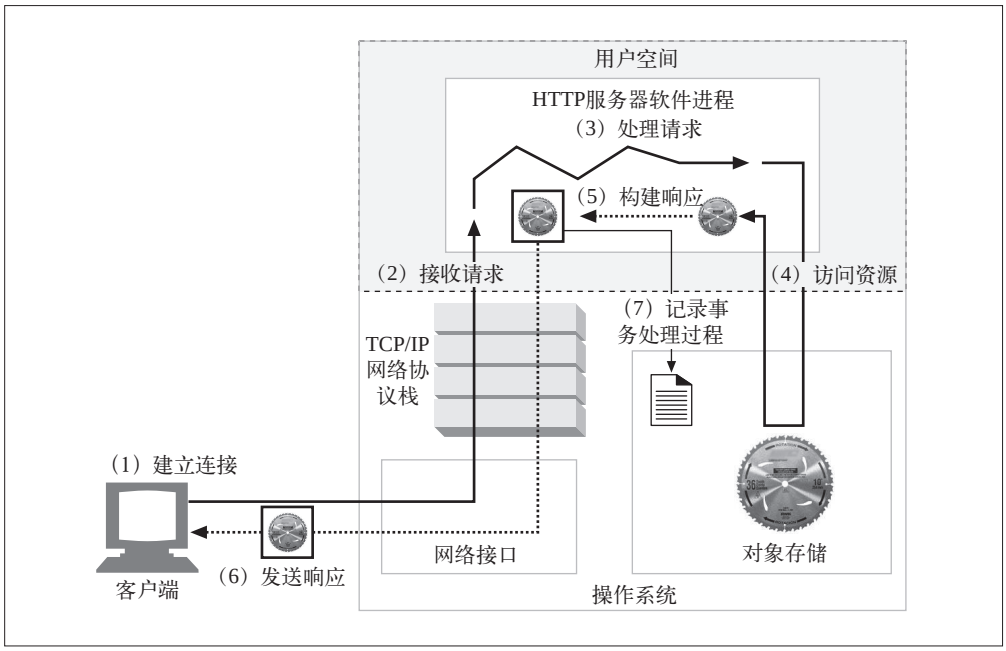


图 5-3 基本 Web 服务器请求的步骤 114

接下来的 7 个小节重点说明了 Web 服务器是怎样实现这些基本任务的。

5.4 第一步——接受客户端连接

如果客户端已经打开了一条到服务器的持久连接，可以使用那条连接来发送它的请求。否则，客户端需要打开一条新的到服务器的连接（回顾第 4 章，复习一下 HTTP 的连接管理技术）。

5.4.1 处理新连接

客户端请求一条到 Web 服务器的 TCP 连接时，Web 服务器会建立连接，判断连接的另一端是哪个客户端，从 TCP 连接中将 IP 地址解析出来。[1] 一旦新连接建立起来

并被接受，服务器就会将新连接添加到其现存 Web 服务器连接列表中，做好监视连接上数据传输的准备。

Web 服务器可以随意拒绝或立即关闭任意一条连接。有些 Web 服务器会因为客户端 IP 地址或主机名是未认证的，或者因为它是已知的恶意客户端而关闭连接。Web 服务器也可以使用其他识别技术。

5.4.2 客户端主机名识别

可以用“反向 DNS”对大部分 Web 服务器进行配置，以便将客户端 IP 地址转换成客户端主机名。Web 服务器可以将客户端主机名用于详细的访问控制和日志记录。但要注意的是，主机名查找可能会花费很长时间，这样会降低 Web 事务处理的速度。很多大容量 Web 服务器要么会禁止主机名解析，要么只允许对特定内容进行解析。

可以用配置指令 HostnameLookups 启用 Apache 的主机查找功能。比如，例 5-2 中的 Apache 配置指令就只打开了 HTML 和 CGI 资源的主机名解析功能。

例 5-2 配置 Apache，为 HTML 和 CGI 资源查找主机名

```
HostnameLookups off
<Files ~ "\.(html|htm|cgi)$">
    HostnameLookups on
</Files>
```

5.4.3 通过ident确定客户端用户

有些 Web 服务器还支持 IETF 的 ident 协议。服务器可以通过 ident 协议找到发起
115 HTTP 连接的用户名。这些信息对 Web 服务器的日志记录特别有用——流行的通用日志格式（Common Log Format）的第二个字段中就包含了每条 HTTP 请求的 ident 用户名。[2]

如果客户端支持 ident 协议，就在 TCP 端口 113 上监听 ident 请求。图 5-4 说明了 ident 协议是如何工作的。在图 5-4a 中，客户端打开了一条 HTTP 连接。然后，服务器打开自己到客户端 ident 服务器端口（113）的连接，发送一条简单的请求，询问与（由客户端和服务器端口号指定的）新连接相对应的用户名，并从客户端解析出包含用户名的响应。

注 1：不同的操作系统在对 TCP 连接进行操作时会使用不同的接口和数据结构。在 Unix 环境下，TCP 连接是由一个*套接字*表示的，可以用 getpeername 调用从套接字中获取客户端的 IP 地址。

注 2：这个通用日志格式的 ident 字段被称为“rfc931”，这是根据定义 ident 协议的过时 RFC 版本（更新过的 ident 规范记录在 RFC1413 中）命名的。

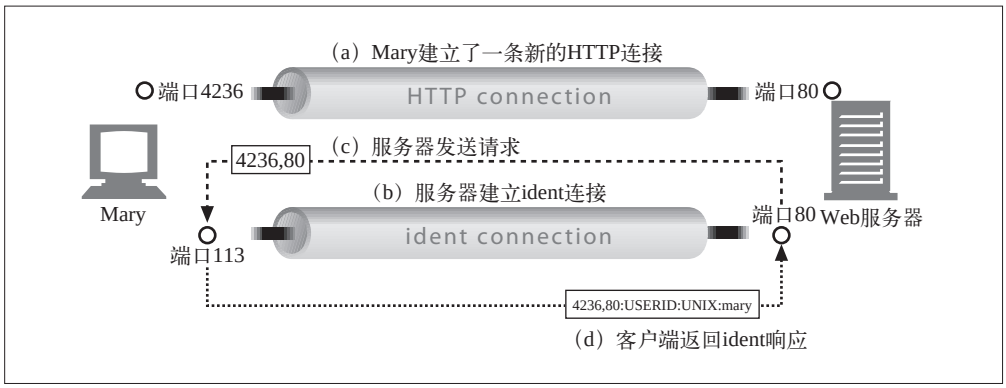


图 5-4　使用 ident 协议来确定 HTTP 的客户端用户名

ident 在组织内部可以很好地工作，但出于多种原因，在公共因特网上并不能很好地工作，原因包括：

- 很多客户端 PC 没有运行 ident 识别协议守护进程软件；
- ident 协议会使 HTTP 事务处理产生严重的时延；
- 很多防火墙不允许 ident 流量进入；
- ident 协议不安全，容易被伪造；
- ident 协议也不支持虚拟 IP 地址；
- 暴露客户端的用户名还涉及隐私问题。

可以通过 Apache 的 IdentityCheck on 指令告知 Apache Web 服务器使用 ident 查找功能。如果没有 ident 信息可用，Apache 会用连字符（-）来填充 ident 日志字段。由于没有 ident 信息可用，在使用通用日志格式的日志文件中，第二个字段通常都是连字符。

5.5　第二步——接收请求报文

连接上有数据到达时，Web 服务器会从网络连接中读取数据，并将请求报文中的内容解析出来（参见图 5-5）。

116

解析请求报文时，Web 服务器会：

- 解析请求行，查找请求方法、指定的资源标识符（URI）以及版本号，[3] 各项之间由一个空格分隔，并以一个回车换行（CRLF）序列作为行的结束；[4]

注 3：HTTP 的初始版本 HTTP/0.9 并不支持版本号。有些 Web 服务器也支持没有版本号的情况，会将报文作为 HTTP/0.9 请求进行解析。

注 4：很多客户端会错误地将 LF 作为行结束的终止符发送，所以很多 Web 服务器都支持将 LF 或 CRLF 作为行结束序列使用。

- 读取以 CRLF 结尾的报文首部；
- 检测到以 CRLF 结尾的、标识首部结束的空行（如果有的话）；
- 如果有的话（长度由 Content-Length 首部指定），读取请求主体。

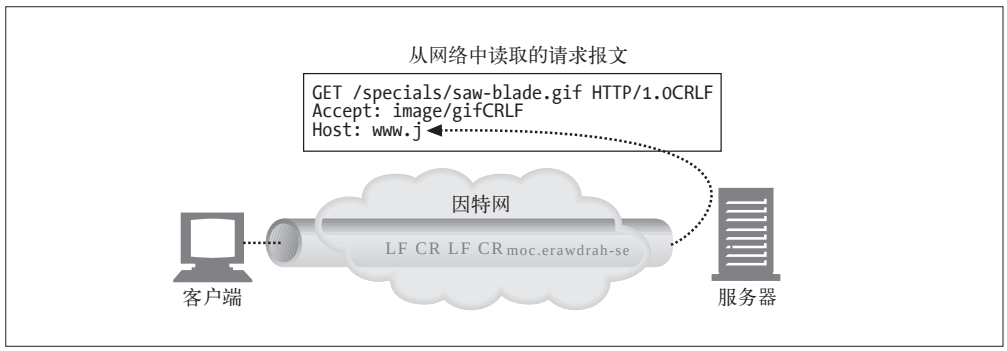


图 5-5　从连接中读取请求报文

解析请求报文时，Web 服务器会不定期地从网络上接收输入数据。网络连接可能随时都会出现延迟。Web 服务器需要从网络中读取数据，将部分报文数据临时存储在内存中，直到收到足以进行解析的数据并理解其意义为止。

5.5.1　报文的内部表示法

有些 Web 服务器还会用便于进行报文操作的内部数据结构来存储请求报文。比如，数据结构中可能包含有指向请求报文中各个片段的指针及其长度，这样就可以将这些首部存放在一个快速查询表中，以便快速访问特定首部的具体值了（参见图 5-6）。

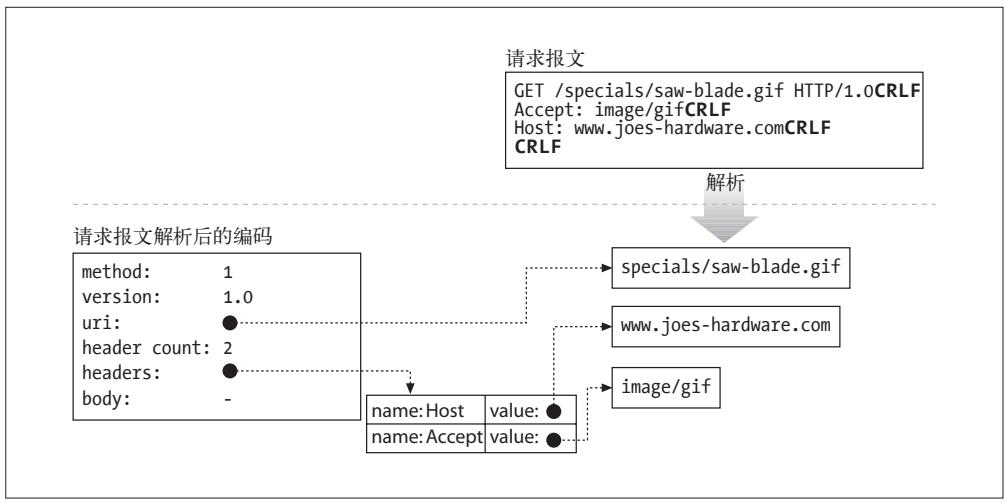


图 5-6　将请求报文解析为便捷的内部表示形式

5.5.2 连接的输入/输出处理结构

高性能的 Web 服务器能够同时支持数千条连接。这些连接使得服务器可以与世界各地的客户端进行通信，每个客户端都向服务器打开了一条或多条连接。某些连接可能在快速地向 Web 服务器发送请求，而其他一些连接则可能在慢慢发送，或者不经常发送请求，还有一些可能是空闲的，安静地等待着将来可能出现的动作。 117

因为请求可能会在任意时刻到达，所以 Web 服务器会不停地观察有无新的 Web 请求。不同的 Web 服务器结构会以不同的方式为请求服务，如图 5-7 所示。

- *单线程 Web 服务器*（参见图 5-7a）
  单线程的 Web 服务器一次只处理一个请求，直到其完成为止。一个事务处理结束之后，才去处理下一条连接。这种结构易于实现，但在处理过程中，所有其他连接都会被忽略。这样会造成严重的性能问题，只适用于低负荷的服务器，以及 type-o-serve 这样的诊断工具。

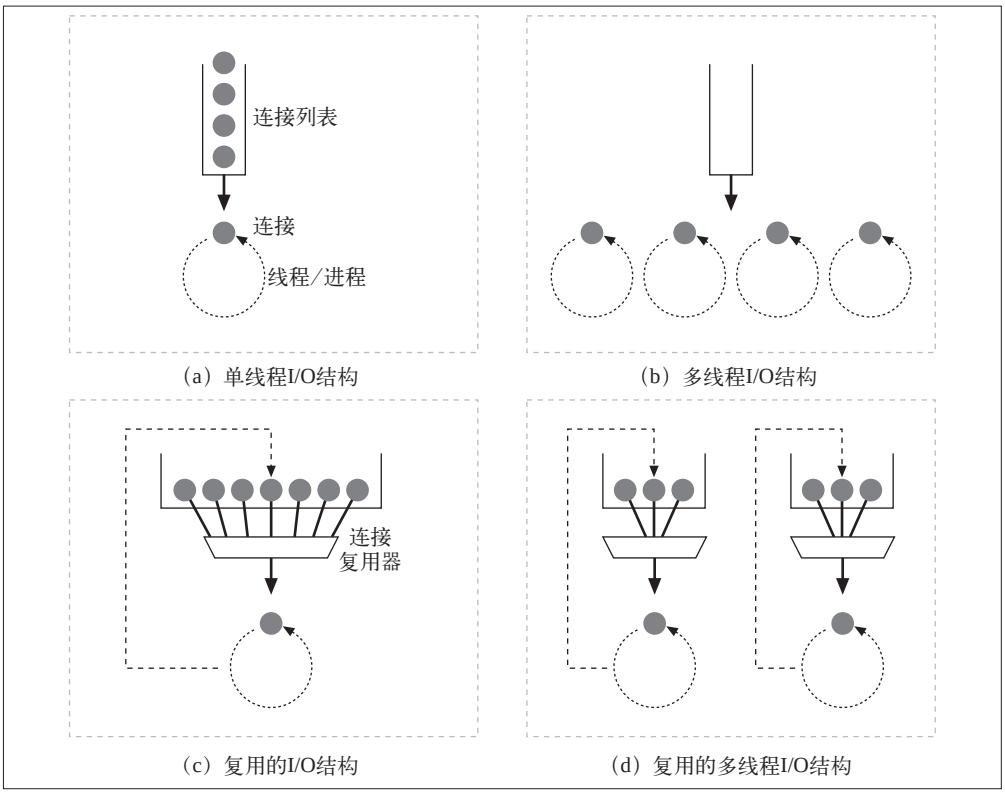
- *多进程及多线程 Web 服务器*（参见图 5-7b）
  多进程和多线程 Web 服务器用多个进程，或更高效的线程同时对请求进行处理。[5] 可以根据需要创建，或者预先创建一些线程 / 进程。[6] 有些服务器会为每条连接分配一个线程 / 进程，但当服务器同时要处理成百、上千，甚至数以万计的连接时，需要的进程或线程数量可能会消耗太多的内存或系统资源。因此，很多多线程 Web 服务器都会对线程 / 进程的最大数量进行限制。 118

- *复用 I/O 的服务器*（参见图 5-7c）
  为了支持大量的连接，很多 Web 服务器都采用了复用结构。在复用结构中，要同时监视所有连接上的活动。当连接的状态发生变化时（比如，有数据可用，或出现错误时），就对那条连接进行少量的处理；处理结束之后，将连接返回到开放连接列表中，等待下一次状态变化。只有在有事情可做时才会对连接进行处理；在空闲连接上等待的时候并不会绑定线程和进程。

- *复用的多线程 Web 服务器*（参见图 5-7d）
  有些系统会将多线程和复用功能结合在一起，以利用计算机平台上的多个 CPU。多个线程（通常是一个物理处理器）中的每一个都在观察打开的连接（或打开的连接中的一个子集），并对每条连接执行少量的任务。

注 5：进程是一个独立的程序控制流，有自己的变量集。线程是一种更快、更高效的进程版本。单个程序可以通过线程和进程同时处理多件事情。为了便于解释，我们将线程和进程当作是可以互换的概念。但由于性能的不同，很多高性能服务器既是多进程的，又是多线程的。

注 6：会预先创建一些线程的系统被称为“工作池”系统，因为池中会有一组线程在等待工作。



119 图 5-7 Web 服务器输入 / 输出结构

5.6 第三步——处理请求

一旦 Web 服务器收到了请求，就可以根据方法、资源、首部和可选的主体部分来对请求进行处理了。

有些方法（比如 POST）要求请求报文中必须带有实体主体部分的数据。其他一些方法（比如 OPTIONS）允许有请求的主体部分，也允许没有。少数方法（比如 GET）禁止在请求报文中包含实体的主体数据。

这里我们并不对请求的具体处理方式进行讨论，因为本书其余大多数章节都在讨论这个问题。

5.7 第四步——对资源的映射及访问

Web 服务器是资源服务器。它们负责发送预先创建好的内容，比如 HTML 页面或 JPEG 图片，以及运行在服务器上的资源生成程序所产生的动态内容。

在 Web 服务器将内容传送给客户端之前，要将请求报文中的 URI 映射为 Web 服务器上适当的内容或内容生成器，以识别出内容的源头。

5.7.1 docroot

Web 服务器支持各种不同类型的资源映射，但最简单的资源映射形式就是用请求 URI 作为名字来访问 Web 服务器文件系统中的文件。通常，Web 服务器的文件系统中会有一个特殊的文件夹专门用于存放 Web 内容。这个文件夹被称为*文档的根目录*（document root，或 docroot）。Web 服务器从请求报文中获取 URI，并将其附加在文档根目录的后面。

在图 5-8 中，有一条对 /specials/saw-blade.gif 的请求到达。这个例子中 Web 服务器的文档根目录为 /usr/local/httpd/files。Web 服务器会返回文件 /usr/local/httpd/files/specials/saw-blade.gif。



图 5-8　将请求 URI 映射为本地 Web 服务器上的资源

120

在配置文件 httpd.conf 中添加一个 `DocumentRoot` 行就可以为 Apache Web 服务器设置文档的根目录了：

```
DocumentRoot /usr/local/httpd/files
```

服务器要注意，不能让相对 URL 退到 docroot 之外，将文件系统的其余部分暴露出来。比如，大多数成熟的 Web 服务器都不允许这样的 URI 看到 Joe 的五金商店文档根目录上一级的文件：

```
http://www.joes-hardware.com/../
```

1. 虚拟托管的docroot

虚拟托管的 Web 服务器会在同一台 Web 服务器上提供多个 Web 站点，每个站点在服务器上都有自己独有的文档根目录。虚拟托管 Web 服务器会根据 URI 或 `Host` 首

部的 IP 地址或主机名来识别要使用的正确文档根目录。通过这种方式，即使请求 URI 完全相同，托管在同一 Web 服务器上的两个 Web 站点也可以拥有完全不同的内容了。

图 5-9 中的服务器托管了两个站点：www.joes-hardware.com 和 www.marys-antiques.com。服务器可以通过 HTTP 的 `Host` 首部，或根据不同的 IP 地址来区分不同的 Web 站点。

- 当请求 A 到达时，服务器会获取文件 /docs/joe/index.html。
- 当请求 B 到达时，服务器会获取文件 /docs/mary/index.html。

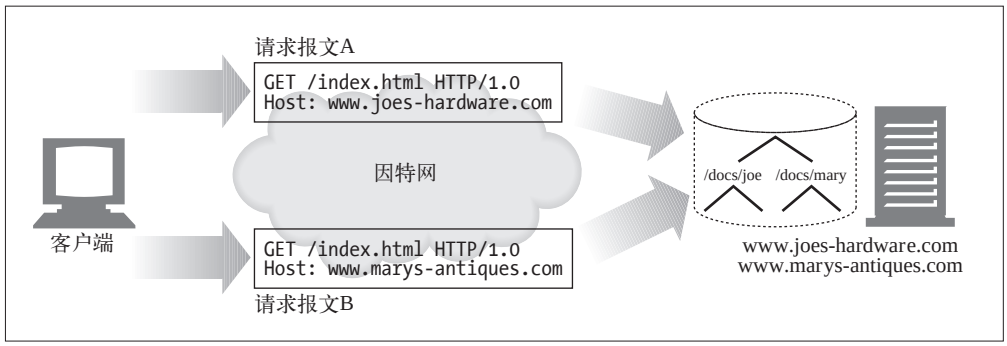


图 5-9　虚拟托管的请求会使用不同的文档根目录

对大多数 Web 服务器来说，配置虚拟托管的文档根目录是很简单的。对常见的 Apache Web 服务器来说，需要为每个虚拟 Web 站点配置一个 VirtualHost 块，而且每个虚拟服务器都要包含 `DocumentRoot`（例 5-3）。

例 5-3　Apache Web 服务器虚拟主机的 docroot 配置

```
<VirtualHost www.joes-hardware.com>
  ServerName www.joes-hardware.com
  DocumentRoot /docs/joe
  TransferLog /logs/joe.access_log
  ErrorLog /logs/joe.error_log
</VirtualHost>

<VirtualHost www.marys-antiques.com>
  ServerName www.marys-antiques.com
  DocumentRoot /docs/mary
  TransferLog /logs/mary.access_log
  ErrorLog /logs/mary.error_log
</VirtualHost>
    ...
```

更多与虚拟托管有关的信息可以参考 18.2 节。

2. 用户的主目录docroot

Docroot 的另一种常见应用是在 Web 服务器上为人们提供私有的 Web 站点。通常会把那些以斜杠和波浪号（/~）开始，后面跟着用户名的 URI 映射为此用户的私有文档根目录。私有 docroot 通常都是用户主目录下那个名为 public_html 的目录，但也可将其配置为其他值（参见图 5-10）。

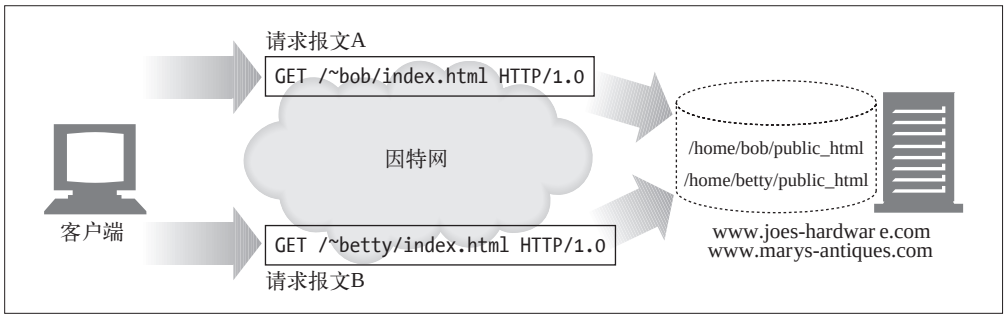


图 5-10　不同用户有不同的 docroot

5.7.2　目录列表

Web 服务器可以接收对目录 URL 的请求，其路径可以解析为一个目录，而不是文件。我们可以对大多数 Web 服务器进行配置，使其在客户端请求目录 URL 时采取不同的动作。

- 返回一个错误。
- 不返回目录，返回一个特殊的默认“索引文件”。
- 扫描目录，返回一个包含目录内容的 HTML 页面。

大多数 Web 服务器都会去查找目录中一个名为 index.html 或 index.htm 的文件来代表此目录。如果用户请求的是一个目录的 URL，而且这个目录中有一个名为 index.html（或 index.htm）的文件，服务器就会返回那个文件的内容。

122

在 Apache Web 服务器上，可以用配置指令 `DirectoryIndex` 来配置要作为默认目录文件使用的文件名集合。指令 `DirectoryIndex` 会按照优先顺序列出所有可以作为目录索引文件使用的文件名。下列配置行会使 Apache 在收到一个目录 URL 请求时，在目录中搜索命令中列出来的任意一个文件：

```
DirectoryIndex index.html index.htm home.html home.htm index.cgi
```

如果用户请求目录 URI 时，没有提供默认的索引文件，而且没有禁止使用目录索引，很多 Web 服务器都会自动返回一个 HTML 文件，此文件中会列出那个目录里

的文件名，以及每个文件的大小和修改日期，还包括到每个文件的 URI 链接。使用这个文件列表可能会很方便，但有些好事者也可以通过它在 Web 服务器上找到一些通常找不到的东西。

可以通过以下 Apache 指令禁止自动生成目录索引文件：

```
Options -Indexes
```

5.7.3 动态内容资源的映射

Web 服务器还可以将 URI 映射为动态资源——也就是说，映射到按需动态生成内容的程序上去（参见图 5-11）。实际上，有一大类名为*应用程序服务器*的 Web 服务器会将 Web 服务器连接到复杂的后端应用程序上去。Web 服务器要能够分辨出资源什么时候是动态的，动态内容生成程序位于何处，以及如何运行那个程序。大多数 Web 服务器都提供了一些基本的机制以识别和映射动态资源。

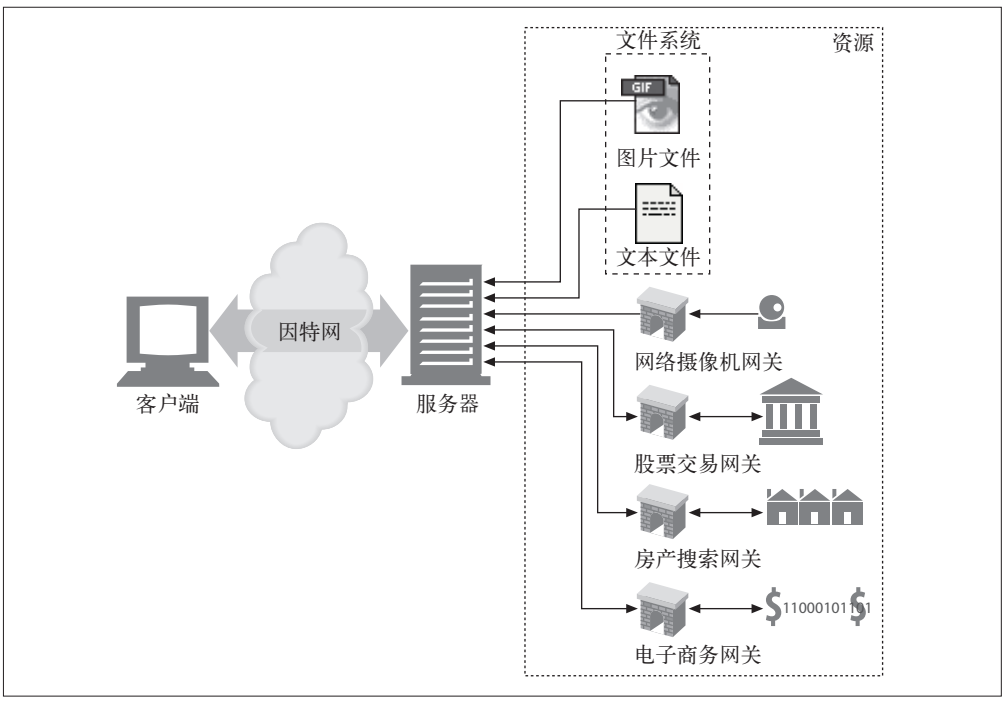


图 5-11　Web 服务器可以提供静态资源和动态资源

Apache 允许用户将 URI 路径名组件映射为可执行文件目录。服务器收到一条带有可执行路径组件的对 URI 的请求时，会试着去执行相应服务器目录中的程序。例如，下列 Apache 配置指令就表明所有路径以 /cgi-bin/ 开头的 URI 都应该执行在目录 /usr/local/etc/httpd/cgi-programs/ 中找到的相应文件：

```
ScriptAlias /cgi-bin/ /usr/local/etc/httpd/cgi-programs/
```

Apache 还允许用户用一个特殊的文件扩展名来标识可执行文件。通过这种方式就可以将可执行脚本放在任意目录中了。下面的 Apache 配置指令说明要执行所有以 .cgi 结尾的 Web 资源：

```
AddHandler cgi-script .cgi
```

CGI 是早期出现的一种简单、流行的服务端应用程序执行接口。现代的应用程序服务器都有更强大更有效的服务端动态内容支持机制，包括微软的动态服务器页面（Active Server Page）和 Java servlet。 123

5.7.4 服务器端包含项

很多 Web 服务器还提供了对服务器端包含项（SSI）的支持。如果某个资源被标识为存在服务器端包含项，服务器就会在将其发送给客户端之前对资源内容进行处理。

要对内容进行扫描，以查找（通常包含在特定 HTML 注释中的）特定的模板，这些模板可以是变量名，也可以是嵌入式脚本。可以用变量的值或可执行脚本的输出来取代特定的模板。这是创建动态内容的一种简便方式。

5.7.5 访问控制

Web 服务器还可以为特定资源进行访问控制。有请求到达，要访问受控资源时，Web 服务器可以根据客户端的 IP 地址进行访问控制，也可以要求输入密码来访问资源。

更多与 HTTP 认证有关的信息请参见第 12 章。 124

5.8 第五步——构建响应

一旦 Web 服务器识别出了资源，就执行请求方法中描述的动作，并返回响应报文。响应报文中包含有响应状态码、响应首部，如果生成了响应主体的话，还包括响应主体。3.4 节详细介绍了 HTTP 响应代码。

5.8.1 响应实体

如果事务处理产生了响应主体，就将内容放在响应报文中回送过去。如果有响应主体的话，响应报文中通常包括：

- 描述了响应主体 MIME 类型的 `Content-Type` 首部；

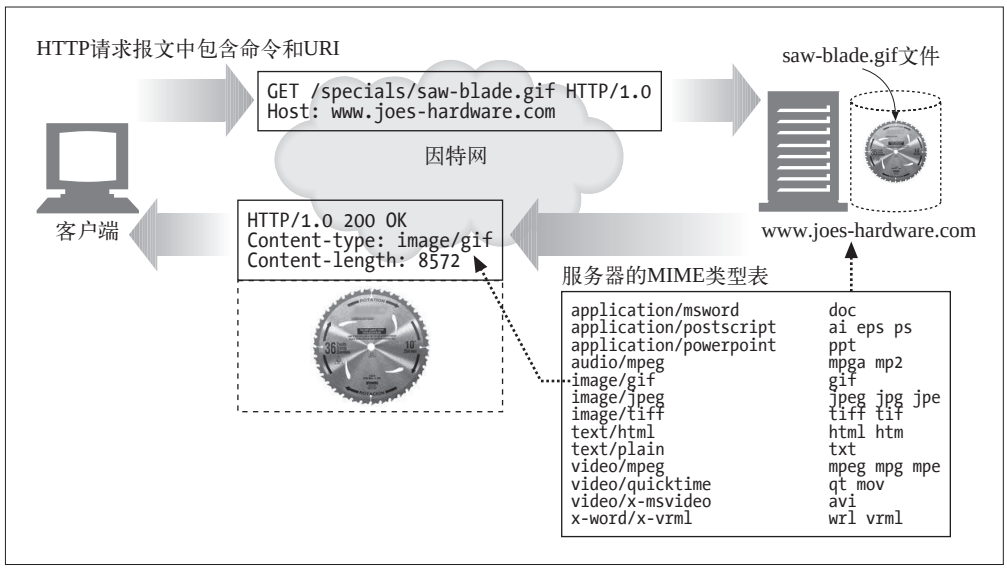
- 描述了响应主体长度的 Content-Length 首部；
- 实际报文的主体内容。

5.8.2 MIME类型

Web 服务器要负责确定响应主体的 MIME 类型。有很多配置服务器的方法可以将 MIME 类型与资源关联起来。

- MIME 类型（mime.types）

  Web 服务器可以用文件的扩展名来说明 MIME 类型。Web 服务器会为每个资源扫描一个包含了所有扩展名的 MIME 类型的文件，以确定其 MIME 类型。这种基于扩展名的类型相关是最常见的，参见图 5-12。



125 图 5-12 Web 服务器用 MIME 类型文件来设置资源输出的 Content-type 首部

- 魔法分类（Magic typing）

  Apache Web 服务器可以扫描每个资源的内容，并将其与一个已知模式表（被称为魔法文件）进行匹配，以决定每个文件的 MIME 类型。这样做可能比较慢，但很方便，尤其是文件没有标准扩展名的时候。

- 显式分类（Explicit typing）

  可以对 Web 服务器进行配置，使其不考虑文件的扩展名或内容，强制特定文件或目录内容拥有某个 MIME 类型。

- 类型协商

  有些 Web 服务器经过配置，可以以多种文档格式来存储资源。在这种情况下，可以配置 Web 服务器，使其可以通过与用户的协商来决定使用哪种格式（及相关的 MIME 类型）“最好”。这个问题将在第 17 章讨论。

  还可以通过配置 Web 服务器，将特定的文件与 MIME 类型相关联。

5.8.3 重定向

Web 服务器有时会返回重定向响应而不是成功的报文。 Web 服务器可以将浏览器重定向到其他地方来执行请求。重定向响应由返回码 3XX 说明。`Location` 响应首部包含了内容的新地址或优选地址的 URI。重定向可用于下列情况。

- 永久搬离的资源

  资源可能已经被移动到了新的位置，或者被重新命名，有了一个新的 URL。Web 服务器可以告诉客户端资源已经被重命名了，这样客户端就可以在从新地址获取资源之前，更新书签之类的信息了。状态码 301 Moved Permanently 就用于此类重定向。

- 临时搬离的资源

  如果资源被临时移走或重命名了，服务器可能希望将客户端重定向到新的位置上去。但由于重命名是临时的，所以服务器希望客户端将来还可以回头去使用老的 URL，不要对书签进行更新。状态码 303 See Other 以及状态码 307 Temporary Redirect 就用于此类重定向。

- URL 增强

  服务器通常用重定向来重写 URL，往往用于嵌入上下文。当请求到达时，服务器会生成一个新的包含了嵌入式状态信息的 URL，并将用户重定向到这个新的 URL 上去。[7] 客户端会跟随这个重定向信息，重新发起请求，但这次的请求会包含完整的、经过状态增强的 URL。这是在事务间维护状态的一种有效方式。状态码 303 See Other 和 307 Temporary Redirect 用于此类重定向。

126

- 负载均衡

  如果一个超载的服务器收到一条请求，服务器可以将客户端重定向到一个负载不太重的服务器上去。状态码 303 See Other 和 307 Temporary Redirect 可用于此类重定向。

- 服务器关联

  Web 服务器上可能会有某些用户的本地信息；服务器可以将客户端重定向到包含了那个客户端信息的服务器上去。状态码 303 See Other 和 307 Temporary Redirect 可用于此类重定向。

注 7：有时会将这些经过扩展和状态增强的 URL 称为“胖 URL”。

- 规范目录名称
  客户端请求的 URI 是一个不带尾部斜线的目录名时，大多数 Web 服务器都会将客户端重定向到一个加了斜线的 URI 上，这样相对链接就可以正常工作了。

5.9 第六步——发送响应

Web 服务器通过连接发送数据时也会面临与接收数据一样的问题。服务器可能有很多条到各个客户端的连接，有些是空闲的，有些在向服务器发送数据，还有一些在向客户端回送响应数据。

服务器要记录连接的状态，还要特别注意对持久连接的处理。对非持久连接而言，服务器应该在发送了整条报文之后，关闭自己这一端的连接。

对持久连接来说，连接可能仍保持打开状态，在这种情况下，服务器要特别小心，要正确地计算 `Content-Length` 首部，不然客户端就无法知道响应什么时候结束了（参见第 4 章）。

5.10 第七步——记录日志

最后，当事务结束时，Web 服务器会在日志文件中添加一个条目，来描述已执行的事务。大多数 Web 服务器都提供了几种日志配置格式。更多细节请参见第 21 章。

5.11 更多信息

更多有关 Apache、Jigsaw 和 ident 的信息，参见以下参考资源

- *Apache:The Definitive Guide*[8]（《Apache 权威指南》）
  Ben Laurie 和 Peter Laurie 编写，O'Reilly & Associates 公司出版。
- *Professional Apache*
  Peter Wainwright 编写，Wrox 公司出版。
- http://www.w3c.org/Jigsaw/
  Jigsaw——W3C 的服务器 W3C 联盟 Web 站点。
- http://www.ietf.org/rfc/rfc1413.txt
  RFC 1413，M. St. Johns 编写的“Identification Protocol”（“标识协议”）。

127

128

注 8：本书影印版由中国电力出版社出版。（编者注）

第6章

代　理

Web 代理（proxy）服务器是网络的中间实体。代理位于客户端和服务器之间，扮演“中间人”的角色，在各端点之间来回传送 HTTP 报文。本章介绍了所有与 HTTP 代理服务器有关的内容，为代理特性提供的特殊支持，以及使用代理服务器时会遇到的一些棘手的问题。

本章主要内容如下：

- 对 HTTP 代理进行解释，将其与 Web 网关进行对比，并说明如何部署代理；
- 给出一些代理所能提供的帮助；
- 说明在现实网络中是怎样部署代理以及如何将网络流量导向代理服务器；
- 说明如何配置浏览器来使用代理；
- 展示 HTTP 的代理请求，说明它们与服务器请求的区别，以及代理是如何微妙地改变浏览器行为的；
- 解释如何通过 `Via` 首部和 TRACE 方法来记录报文传输路径上的代理服务器链；
- 描述基于代理的 HTTP 访问控制方法；
- 解释代理如何与客户端和服务器进行交互，每个客户端和服务器支持的特性和使用的版本都可能有所不同。

6.1 Web的中间实体

Web 上的代理服务器是代表客户端完成事务处理的中间人。如果没有 Web 代理，HTTP 客户端就要直接与 HTTP 服务器进行对话。有了 Web 代理，客户端就可以与代理进行对话，然后由代理代表客户端与服务器进行交流。客户端仍然会完成对事务的处理，但它是通过代理服务器提供的优质服务来实现的。

129

HTTP 的代理服务器既是 Web 服务器又是 Web 客户端。HTTP 客户端会向代理发送请求报文，代理服务器必须像 Web 服务器一样，正确地处理请求和连接，然后返回响应。同时，代理自身要向服务器发送请求，这样，其行为就必须像正确的 HTTP 客户端一样，要发送请求并接收响应（参见图 6-1）。如果要创建自己的 HTTP 代理，就要认真地遵循为 HTTP 客户端和 HTTP 服务器制定的规则。

6.1.1 私有和共享代理

代理服务器可以是某个客户端专用的，也可以是很多客户端共享的。单个客户端专用的代理被称为*私有代理*。众多客户端共享的代理被称为*公共代理*。

- *公共代理*
  大多数代理都是公共的共享代理。集中式代理的成本效率更高，更容易管理。某些

代理应用，比如高速缓存代理服务器，会利用用户间共同的请求，这样的话，汇入同一个代理服务器的用户越多，它就越有用。

- *私有代理*
  专用的私有代理并不常见，但它们确实存在，尤其是直接运行在客户端计算机上的时候。有些浏览器辅助产品，以及一些 ISP 服务，会在用户的 PC 上直接运行一些小型的代理，以便扩展浏览器特性，提高性能，或为免费 ISP 服务提供主机广告。

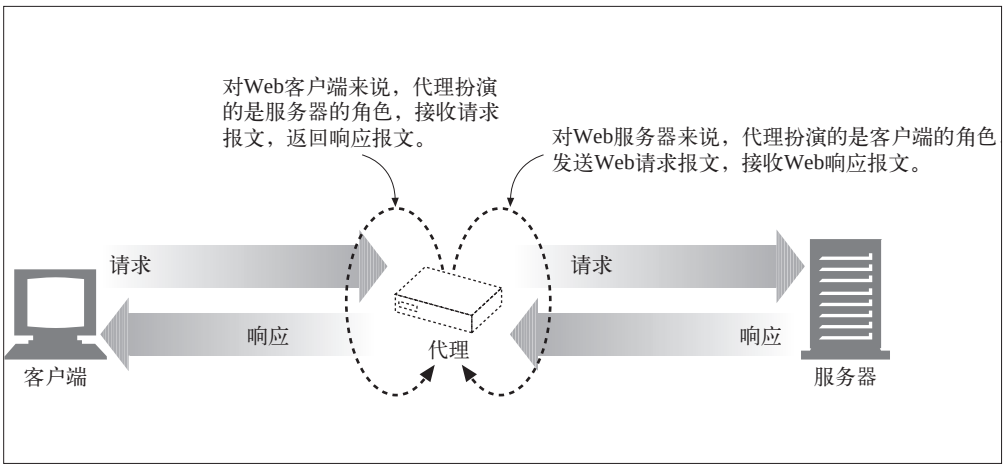


图 6-1　代理既是服务器，又是客户端

6.1.2　代理与网关的对比

严格来说，代理连接的是两个或多个使用相同协议的应用程序，而网关连接的则是两个或多个使用不同协议的端点。网关扮演的是“协议转换器”的角色，即使客户端和服务器使用的是不同的协议，客户端也可以通过它完成与服务器之间的事务处理。

130

图 6-2 显示了代理和网关之间的区别。

- 图 6-2a 中的中间设备是一个 HTTP 代理，因为代理与客户端和服务器之间使用的都是 HTTP 协议。
- 图 6-2b 中的中间设备是一个 HTTP/POP 网关，因为它把 HTTP 的前台与 POP E-mail 的后端连接了起来。网关将 Web 事务转换成适当的 POP 事务，这样用户就可以通过 HTTP 读取 E-mail 了。基于 Web 的 E-mail 程序，比如 Yahoo! 邮件和 MSN Hotmail 都是 HTTP E-mail 网关。

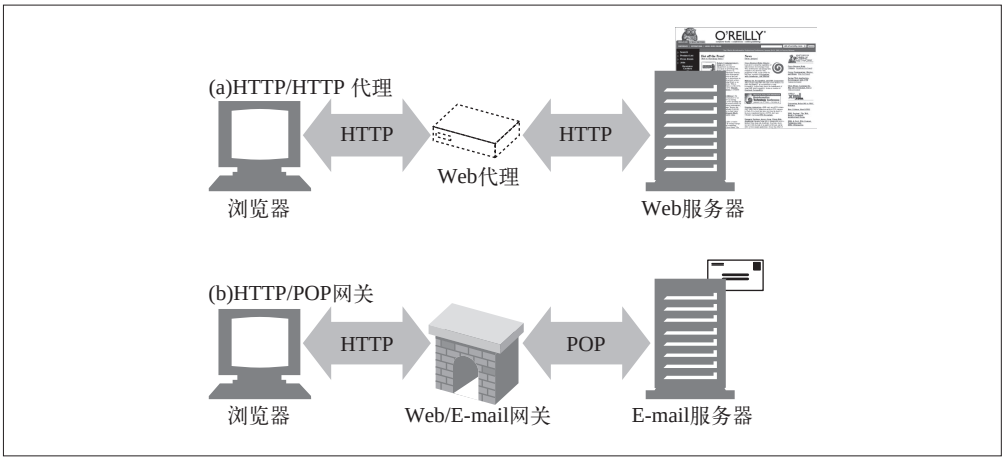


图 6-2　代理使用同一种协议，网关则将不同的协议连接起来

实际上，代理和网关之间的区别很模糊。由于浏览器和服务器实现的是不同版本的 HTTP，代理也经常要做一些协议转换工作。而商业化的代理服务器也会实现网关的功能来支持 SSL 安全协议、SOCKS 防火墙、FTP 访问，以及基于 Web 的应用程序。我们将在第 8 章详细介绍网关。

6.2　为什么使用代理

代理服务器可以实现各种时髦且有用的功能。它们可以改善安全性，提高性能，节省费用。代理服务器可以看到并接触到所有流过的 HTTP 流量，所以代理可以监视流量并对其进行修改，以实现很多有用的增值 Web 服务。这里给出了几种代理使用方法的示例。

- *儿童过滤器*

  小学在为教育站点提供无阻碍访问的同时，可以利用过滤器代理来阻止学生访问成人内容。如图 6-3 所示，代理应该允许学生无限制地访问教育性内容，但对不适合儿童的站点要强行禁止访问。[1]

131

- *文档访问控制*

  可以用代理服务器在大量 Web 服务器和 Web 资源之间实现统一的访问控制策略，创建审核跟踪机制。这在大型企业环境或其他分布式机构中是很有用的。

  在集中式代理服务器上可以对所有访问控制功能进行配置，而无需在众多由不同组织管理、不同厂商制造、使用不同模式的 Web 服务器上进行经常性

注 1：有些公司和非营利性组织提供了过滤软件，维护了一些“黑名单”，以识别并限制对危害性内容的访问。

的访问控制升级。[2]

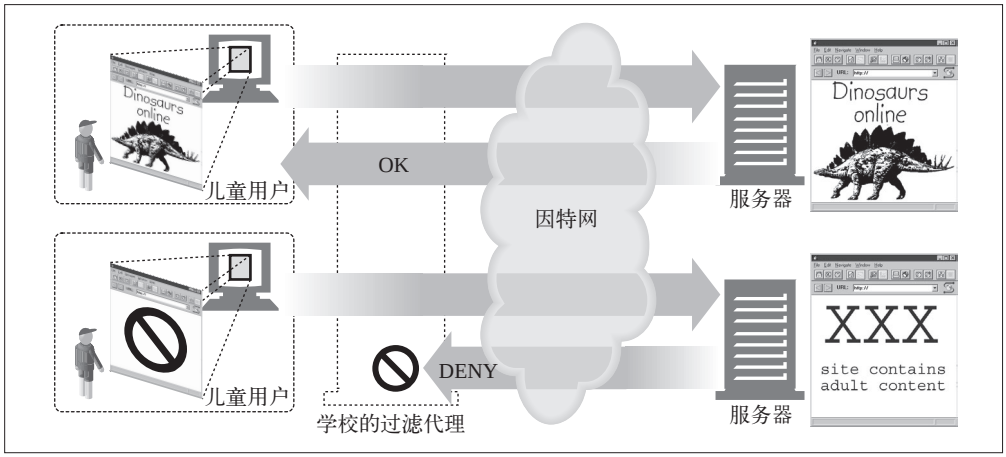


图 6-3　代理应用程序实例：儿童安全的因特网过滤器

在图 6-4 中，集中式访问控制代理：

- 允许客户端 1 无限制地访问服务器 A 的新闻页面；
- 客户端 2 可以无限制地访问因特网；
- 在允许客户端 3 访问服务器 B 之前，要求其输入口令。

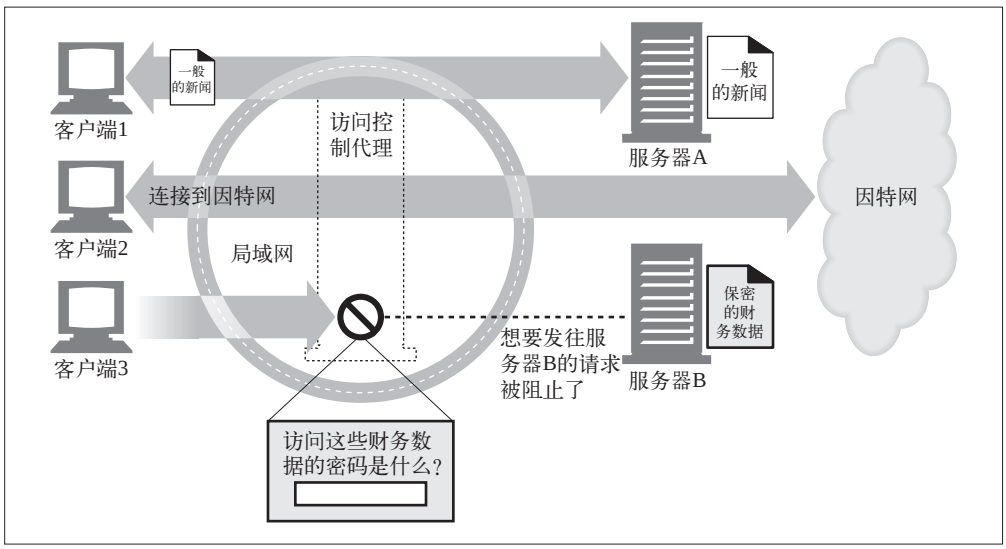


图 6-4　代理应用程序实例：集中式文档访问控制

注 2：为防止一些经验丰富的用户蓄意绕过控制代理，可以静态地配置 Web 服务器，使其仅接受来自代理服务器的请求。

- *安全防火墙*

  网络安全工程师通常会使用代理服务器来提高安全性。代理服务器会在网络中的单一安全节点上限制哪些应用层协议的数据可以流入或流出一个组织。还可以提供用来消除病毒的 Web 和 E-mail 代理使用的那种挂钩程序，以便对流量进行详
132 细的检查（参见图 6-5）。

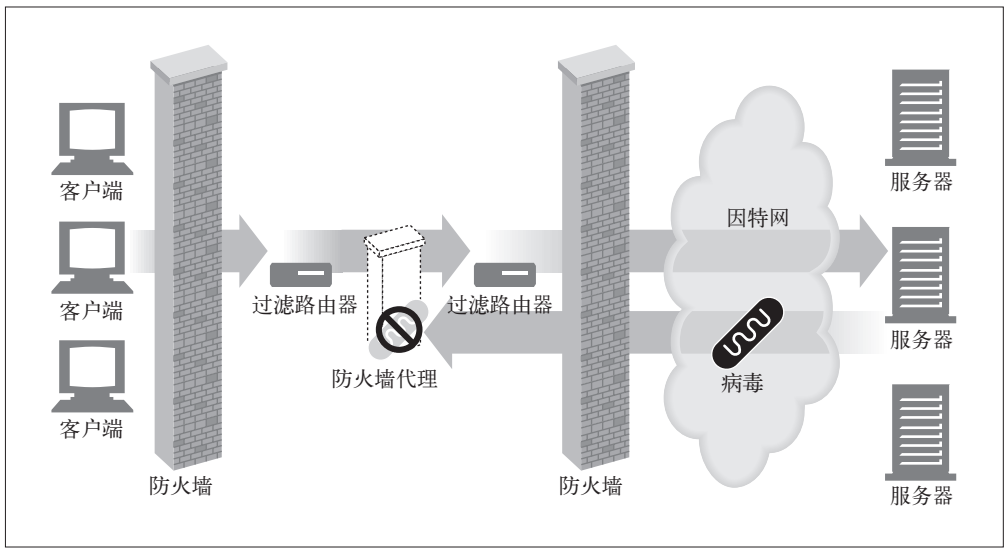


图 6-5　代理应用程序实例：安全防火墙

- *Web 缓存*

  代理缓存维护了常用文档的本地副本，并将它们按需提供，以减少缓慢且昂贵的因特网通信。

  在图 6-6 中，客户端 1 和客户端 2 会去访问附近 Web 缓存上的对象 A，而客户
133 端 3 和客户端 4 访问的则是原始服务器上的文档。

- *反向代理*

  代理可以假扮 Web 服务器。这些被称为*替代物*（surrogate）或*反向代理*（reverse proxy）的代理接收发给 Web 服务器的真实请求，但与 Web 服务器不同的是，它们可以发起与其他服务器的通信，以便按需定位所请求的内容。

  可以用这些反向代理来提高访问慢速 Web 服务器上公共内容时的性能。在这种配置中，通常将这些反向代理称为*服务器加速器*（server accelerator）（参见图 6-7）。还可以将替代物与内容路由功能配合使用，以创建按需复制内容的分布式网络。

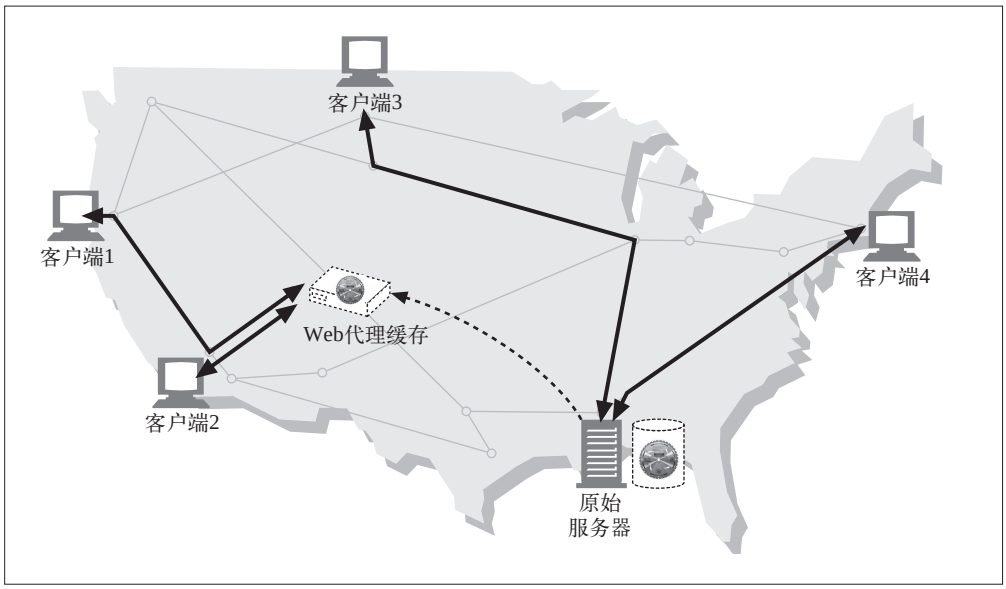


图 6-6　代理应用程序实例：Web 缓存

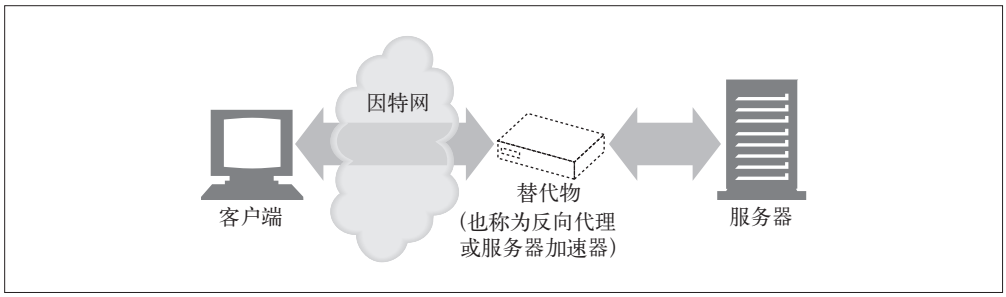


图 6-7　代理应用程序实例：（服务器加速器实现方式中的）替代物

- *内容路由器*

  代理服务器可以作为“内容路由器”使用，根据因特网流量状况以及内容类型将请求导向特定的 Web 服务器。

  内容路由器也可以用来实现各种服务级的请求。比如，如果用户或内容提供者付费要求提供更高的性能，内容路由器可以将请求转发到附近的复制缓存（参见图 6-8），或者如果用户申请了过滤服务，还可以通过过滤代理来转发 HTTP 请求。可以用自适应内容路由代理来构建很多有趣的服务。

- *转码器*

  代理服务器在将内容发送给客户端之前，可以修改内容的主体格式。在这些数据

表示法之间进行的透明转换被称为*转码*（transcoding）。[3]

转码代理可以在传输 GIF 图片时，将其转换成 JPEG 图片，以减小尺寸。也可以对图片进行压缩，或降低颜色的色彩饱和度以便在电视上观看。同样，可以对文本文件进行压缩，并为能够使用因特网的呼机和智能手机生成小型的文本摘要 Web 页面。代理甚至可以在传输文档的过程中将其转换成外语。

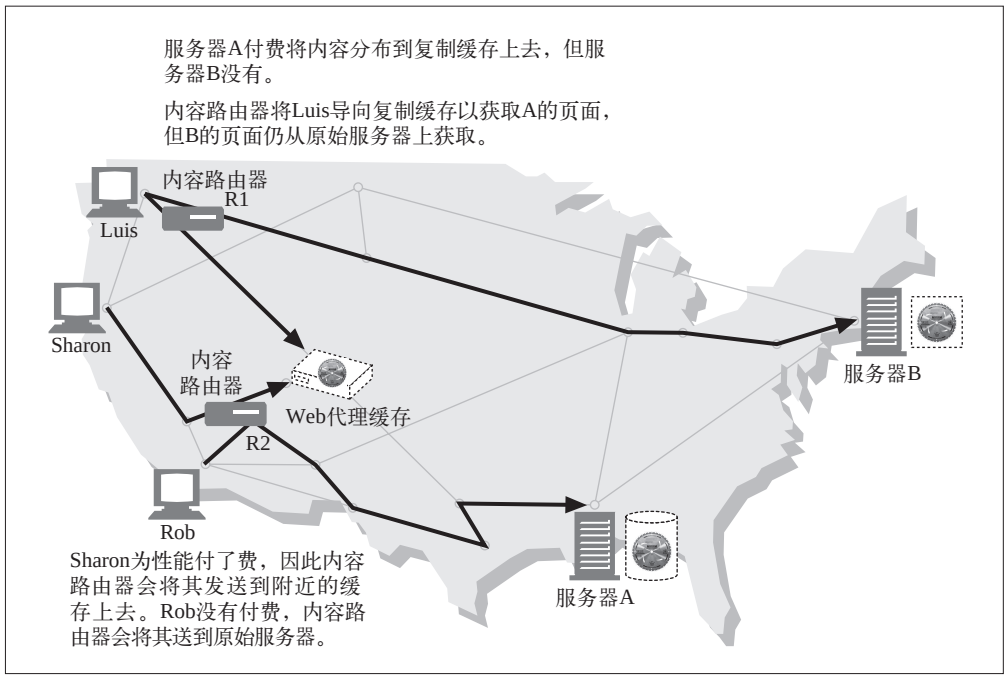


图 6-8　代理应用程序实例：内容路由器

图 6-9 显示了一个转码代理，这个代理可以将英语文本转换成西班牙语文本，将 HTML 页面重新格式化为较简单的文本，以便显示在手机的小屏幕上。

135

- *匿名者*

  匿名者代理会主动从 HTTP 报文中删除身份特性（比如客户端 IP 地址、`From` 首部、`Referer` 首部、cookie、URI 的会话 ID），从而提供高度的私密性和匿名性。[4]

注 3：有些人会对“转码”和“翻译”进行区分，将转码定义为对数据编码进行的相对简单的转换（比如无损压缩），将翻译定义为更重要的、对数据的重新格式化或语义转换。我们用术语转码来表示所有在中间实体上对内容进行的修改。

注 4：但是，由于删除了识别信息，用户浏览体验的质量可能会有所下降，有些Web站点可能会无法正常工作。

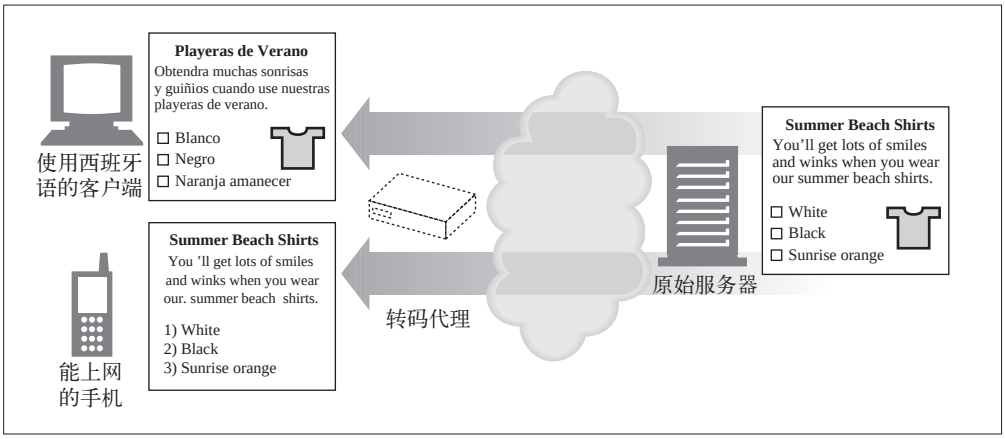


图 6-9　代理应用程序实例：内容转码器

在图 6-10 中，匿名代理会对用户报文进行下列修改以增加私密性。

- 从 `User-Agent` 首部删除用户的计算机与 OS 类型。
- 删除 `From` 首部以保护用户的 E-mail 地址。
- 删除 `Referer` 首部来掩盖用户访问过的其他站点。
- 删除 `Cookie` 首部以剔除概要信息和身份的数据。

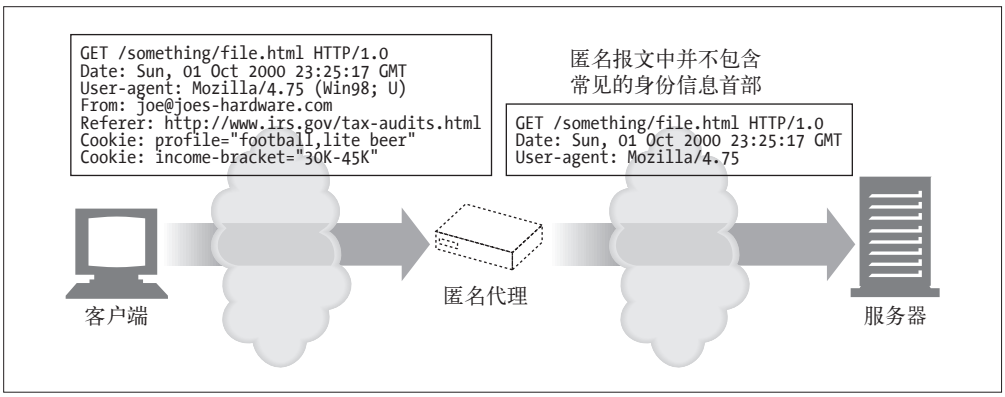


图 6-10　代理应用程序实例：匿名者

136

6.3　代理会去往何处

前一小节解释了代理会做些什么。现在我们来看看在一个网络结构中部署代理的时候，它会位于何处。本节会涵盖以下内容：

- 怎样将代理部署到网络中去；

- 怎样将代理以层级方式连接在一起；
- 怎样先将网络流量直接导入代理服务器中。

6.3.1 代理服务器的部署

可以根据其目标用途，将代理放在任意位置。图 6-11 给出了部署代理服务器的几种方式。

- *出口代理*
  可以将代理固定在本地网络的出口点，以便控制本地网络与大型因特网之间的流量。可以在公司网络中使用出口代理（参见图 6-11a），提供针对公司外部恶意黑客的防火墙保护，或降低带宽费用，提高因特网流量的性能。小学可能会使用过滤出口代理来防止早熟的学生浏览不恰当的内容。

- *访问（入口）代理*
  代理常被放在 ISP 访问点上，用以处理来自客户的聚合请求。ISP 使用缓存代理来存储常用文档的副本，以提高用户（尤其是高速连接用户）的下载速度，降低因特网带宽耗费（参见图 6-11b）。

- *反向代理*
  代理通常会被部署在网络边缘，在 Web 服务器之前，作为替代物（也常被称为反向代理，参见图 6-11c）使用，在那里它们可以处理所有传送给 Web 服务器的请求，并只在必要时向 Web 服务器请求资源。替代物可以提高 Web 服务器的安全特性，或者将快速的 Web 服务器缓存放在较慢的服务器之前，以提高性能。反向代理通常会直接冒用 Web 服务器的名字和 IP 地址，这样所有的请求就会被发送给代理而不是服务器了。

- *网络交换代理*
  可以将具有足够处理能力的代理放在网络之间的因特网对等交换点上，通过缓存来减轻因特网节点的拥塞，并对流量进行监视，参见图 6-11d。[5]

137

6.3.2 代理的层次结构

可以通过*代理层次结构*（proxy hierarchy）将代理级联起来。如图 6-12 所示，在代理的层次结构中，会将报文从一个代理传给另一个代理，直到最终抵达原始服务器为止（然后通过代理传回给客户端）。

注 5：核心代理通常被部署在因特网带宽很昂贵的地方（尤其是在欧洲）。有些国家（比如英国）还会出于对国家安全的考虑，对有争议的代理部署进行评估，以监测因特网流量。

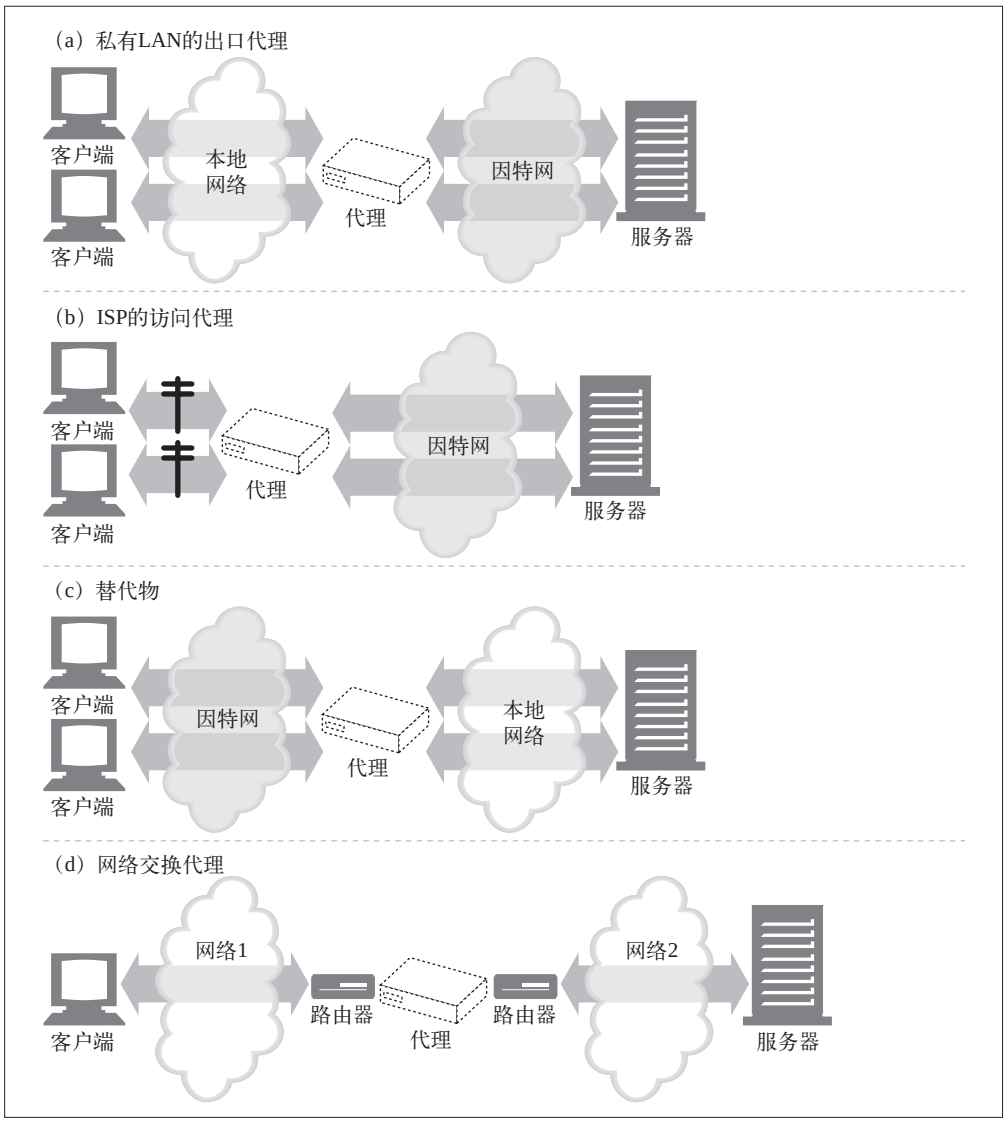


图 6-11　可以根据其目标用途，以多种方式来部署代理

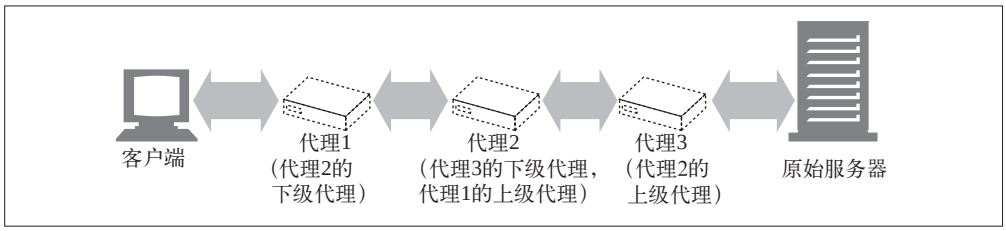


图 6-12　三级的代理层次结构

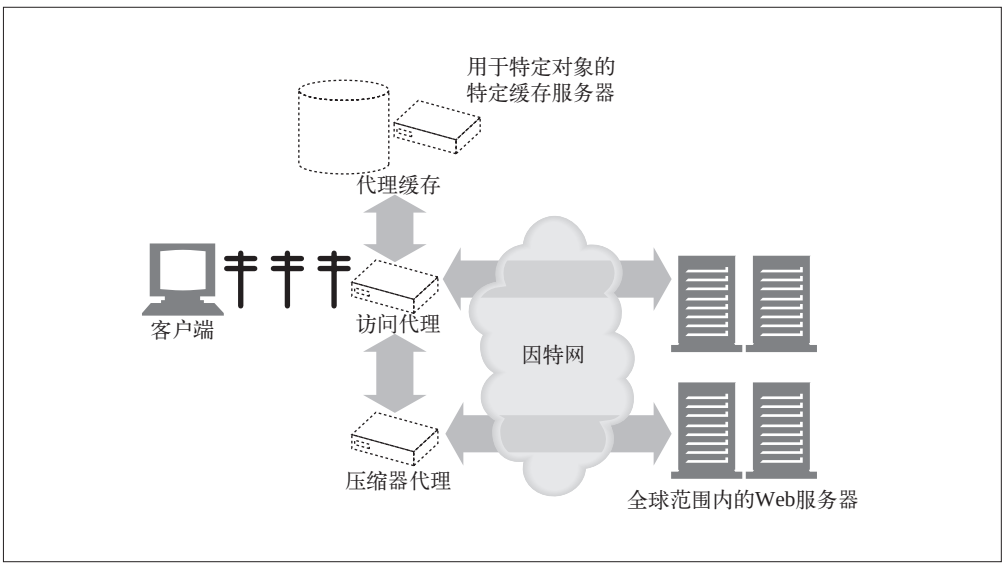
Proxy 层次结构中的代理服务器被赋予了父（parent）和子（child）的关系。下一个
138 入口（inbound）代理（靠近服务器）被称为父代理，下一个出口（outbound）代理（靠近客户端）被称为子代理。在图 6-12 中，代理 1 是代理 2 的子代理。同样，代理 2 是代理 3 的子代理，代理 3 是代理 2 的父代理。

代理层次结构的内容路由

图 6-12 中的代理层次结构是静态的——代理 1 总是会将报文转发给代理 2，代理 2 总是会将报文转发给代理 3。但是，层次不一定非得是静态的。代理服务器可以根据众多因素，将报文转发给一个不断变化的代理服务器和原始服务器集。

比如，在图 6-13 中，访问代理会根据不同的情况将报文转发给父代理或原始服务器。

- 如果所请求的对象属于一个付费使用内容分发服务的 Web 服务器，代理就会将请求发送给附近的一个缓存服务器，这个服务器会返回已缓存对象，或者如果它那儿没有的话，它会去取回内容。
- 如果请求的是特定类型的图片，访问代理会将请求转发给一个特定的压缩代理，这个代理会去获取图片，然后对其进行压缩，这样通过到客户端的慢速 Modem 下载时，速度会更快一些。



139 图 6-13　代理层次结构可以是动态的，随请求而变的

这里还有几个动态选择父代理的例子。

- 负载均衡
  子代理可能会根据当前父代理上的工作负载级别来决定如何选择一个父代理，以均衡负载。

- 地理位置附近的路由
  子代理可能会选择负责原始服务器所在物理区域的父代理。

- 协议 / 类型路由
  子代理可能会根据 URI 将报文转发到不同的父代理和原始服务器上去。某些特定类型的 URI 可能要通过一些特殊的代理服务器转发请求，以便进行特殊的协议处理。

- 基于订购的路由
  如果发布者为高性能服务额外付费了，它们的 URI 就会被转发到大型缓存或压缩引擎上去，以提高性能。

  在不同的产品中，动态父路由逻辑的实现方式各有不同，包括使用配置文件、脚本语言和动态可执行插件等。

6.3.3 代理是如何获取流量的

客户端通常会直接与 Web 服务器进行通信，所以我们要解释清楚 HTTP 流量怎样才能首先流向代理。有四种常见方式可以使客户端流量流向代理。

- 修改客户端
  很多 Web 客户端，包括网景和微软的浏览器，都支持手工和自动的代理配置。如果将客户端配置为使用代理服务器，客户端就会将 HTTP 请求有意地直接发送给代理，而不是原始服务器（参见图 6-14a）。

- 修改网络
  网络基础设施可以通过若干种技术手段，在客户端不知道，或没有参与的情况下，拦截网络流量并将其导入代理。这种拦截通常都依赖于监视 HTTP 流量的交换设备及路由设备，在客户端毫不知情的情况下，对其进行拦截，并将流量导入一个代理（参见图 6-14b）。这种代理被称为*拦截*（intercepting）代理。[6]

注 6：拦截代理通常被称为“透明代理”，因为你会在不知情的情况下连接到这些代理上去。但 HTTP 规范中已用“透明”来表示那些不会对语义进行修改的功能了，所以标准制定机构建议在流量捕获中使用术语“拦截”。这里采纳了这一术语。

- 修改 DNS 的命名空间

140 放在 Web 服务器之前的代理服务器——替代物，会直接假扮 Web 服务器的名字和 IP 地址，这样，所有的请求就会发送给这些替代物，而不是服务器了（参见图 6-14c）。要实现这一点，可以手工编辑 DNS 名称列表，或者用特殊的动态 DNS 服务器根据需要来确定适当的代理或服务器。有时在安装过程中，真实服务器的 IP 地址和名称被修改了，替代物得到的会是之前的地址和名称。

- 修改 Web 服务器

  也可以将某些 Web 服务器配置为向客户端发送一条 HTTP 重定向命令（响应码 305），将客户端请求重定向到一个代理上去。收到重定向命令后，客户端会与代理进行通信（参见图 6-14d）。

下一节解释了如何配置客户端才能使其将流量发送给代理。第 20 章会说明如何配置网络、DNS 以及服务器，才能将流量重定向到代理服务器。

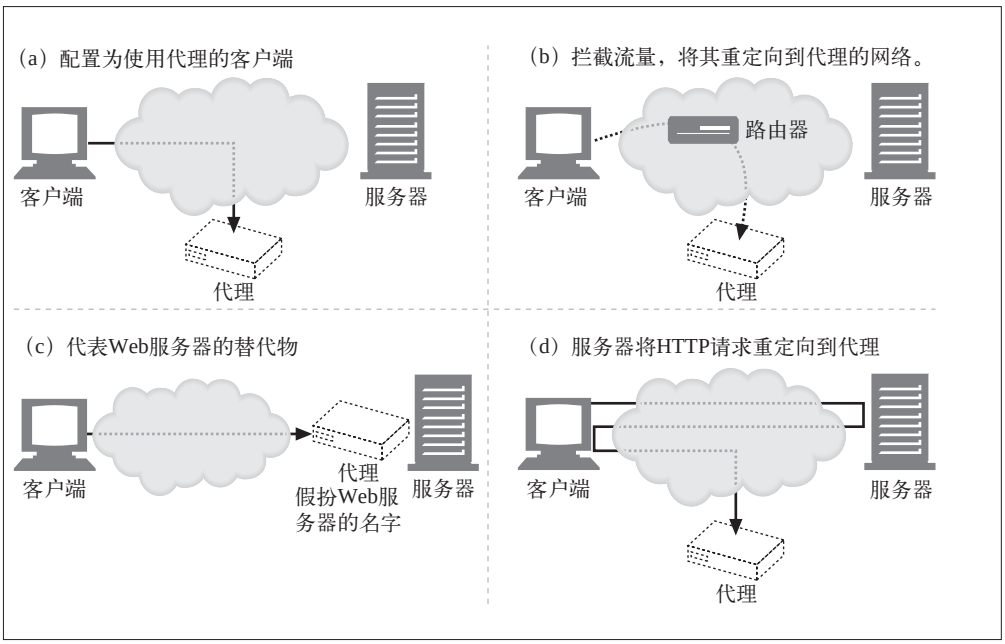


图 6-14　有很多技巧可以将 Web 请求导向代理

6.4　客户端的代理设置

所有现代的 Web 浏览器都允许用户对代理的使用进行配置。实际上，很多浏览器都提供了多种配置代理的方式，其中包括以下几种。

- *手工配置*
  显式地设置要使用的代理。

- *预先配置浏览器*
  浏览器厂商或发行商会在将浏览器发送给其客户之前预先对浏览器（或所有其他Web客户端）的代理设置进行手工配置。

- *代理的自动配置*（Proxy Auto-Configuration，PAC）
  提供一个URI，指向一个用JavaScript语言编写的*代理自动配置*文件；客户端会取回这个JavaScript文件，并运行它以决定是否应该使用一个代理，如果是的话，应该使用哪个代理服务器。

- WPAD的*代理发现*
  有些浏览器支持Web代理自动发现协议（Web Proxy Autodiscovery Protocol，WPAD），这个协议会自动检测出浏览器可以从哪个“配置服务器”下载到一个自动配置文件。[7]

6.4.1 客户端的代理配置：手工配置

很多Web客户端都允许用户手工配置代理。网景的Navigator和微软的Internet Explorer都为代理配置提供了便捷的支持。

在网景的Navigator 6中，可以通过菜单选项Edit（编辑）→ Preferences（首选项）→ Advanced（高级）→ Proxies（代理），然后选中单选按钮“Manual proxy”（手工配置代理）来指定代理。

在微软的Internet Explorer 5中，可以在Tools（工具）→ Internet Options（Internet选项）菜单中，选择一个连接，点击“Settings”（设置），选中“Use a proxy server”（使用代理服务器）框，并点击“Advanced”（高级），来手工指定代理。

其他浏览器都有不同的方式来进行手工配置的修改，但其思想是一样的：为代理指定主机和端口。有些ISP会向客户发送预先配置好的浏览器，或定制好的操作系统，使其将Web流量重定向到代理服务器上。

6.4.2 客户端代理配置：PAC文件

手工代理配置很简单但有些死板。只能为所有内容指定唯一的一个代理服务器，而且不支持故障转移。手工代理配置还会给大型组织带来管理问题。如果配置过的浏

注7：当前只有Internet Explorer支持这一特性。

览器基数很大，那么需要进行修改的时候，重新配置每个浏览器是非常困难，甚至是不可能的。

PAC 文件是一些小型的 JavaScript 程序，可以在运行过程中计算代理设置，因此，是一种更动态的代理配置解决方案。访问每个文档时，JavaScript 函数都会选择恰当的代理服务器。

要使用 PAC 文件，就要用 JavaScript PAC 文件的 URI 来配置浏览器［配置方式与
142 手工配置类似，但要在“automatic configuration”（自动配置）框中提供一个 URI］。浏览器会从这个 URI 上获取 PAC 文件，并用 JavaScript 逻辑为每次访问计算恰当的代理服务器。PAC 文件的后缀通常是 .pac，MIME 类型通常是 application/x-ns-proxy-autoconfig。

每个 PAC 文件都必须定义一个名为 `FindProxyForURL(url,host)` 的函数，用来计算访问 URI 时使用的适当的代理服务器。函数的返回值可以是表 6-1 列出的任意值。

表6-1　代理自动配置脚本的返回值

FindProxyForURL的返回值	描　述
DIRECT	不经过任何代理，直接进行连接
PROXY host:port	应该使用指定的代理
SOCKS host:port	应该使用指定的 SOCKS 服务器

例 6-1 中的 PAC 文件为 HTTP 事务处理指定了一个代理，为 FTP 事务处理指定了另一个代理，并为所有其他类型的事务处理使用直连方式。

例 6-1　代理自动配置文件示例

```
function FindProxyForURL(url, host) {
    if (url.substring(0,5) == "http:") {
        return "PROXY http-proxy.mydomain.com:8080";
    } else if (url.substring(0,4) =="ftp:") {
        return "PROXY ftp-proxy.mydomain.com:8080";
    } else {
        return "DIRECT";
    }
}
```

更多有关 PAC 文件的细节，请参见第 20 章。

6.4.3　客户端代理配置：WPAD

另一种浏览器配置机制是 WPAD 协议。WPAD 协议的算法会使用发现机制的逐

级上升策略自动地为浏览器查找合适的 PAC 文件。实现 WPAD 协议的客户端需要：

- 用 WPAD 找到 PAC 的 URI；
- 从指定的 URI 获取 PAC 文件；
- 执行 PAC 文件来判定代理服务器；
- 为请求使用代理服务器。

WPAD 会使用一系列的资源发现技术来判定适当的 PAC 文件。并不是所有组织都能够使用所有的发现技术，所以 WPAD 使用了很多种发现技术。WPAD 会一个接一个地对每种技术进行尝试，直到成功为止。

当前的 WPAD 协议规范按顺序定义了下列技术：

- 动态主机配置协议（Dynamic Host Configuration Protocol，DHCP）；
- 服务定位协议（Service Location Protocol，SLP）；
- DNS 知名主机名；
- DNS SRV 记录；
- TXT 记录中的 DNS 服务 URI。

更多信息，请参阅第 20 章。

6.5 与代理请求有关的一些棘手问题

本节对与代理服务器请求有关的一些比较棘手且易被误解的问题进行了解释，其中包括：

- 代理请求中的 URI 和服务器请求中的 URI 有何不同；
- 拦截和反向代理是如何将服务器主机信息隐藏起来的；
- 修改 URI 的规则；
- 代理是怎样影响浏览器的智能 URI 自动完成机制，或主机名扩展特性的。

6.5.1 代理URI与服务器URI的不同

除了一点之外，Web 服务器报文和 Web 代理报文的语法是一样的。客户端向服务器而不是代理发送请求时，HTTP 请求报文中的 URI 会有所不同。

客户端向 Web 服务器发送请求时，请求行中只包含部分 URI（没有方案、主机或端口），如下例所示：

```
GET /index.html HTTP/1.0
User-Agent: SuperBrowser v1.3
```

但当客户端向代理发送请求时，请求行中则包含完整的 URI。例如：

```
GET http://www.marys-antiques.com/index.html HTTP/1.0
User-Agent: SuperBrowser v1.3
```

为什么会有两种不同的请求格式，一种用于代理，另一种用于服务器呢？在原始的 HTTP 设计中，客户端会直接与单个服务器进行对话。不存在虚拟主机，也没有为代理制定什么规则。单个的服务器都知道自己的主机名和端口，所以，为了避免发送冗余信息，客户端只需发送部分 URI 即可，无需发送方案和主机（以及端口）。

代理出现之后，使用部分 URI 就有问题了。代理需要知道目标服务器的名称，这样
144 它们才能建立自己与服务器的连接。基于代理的网关要知道 URI 的方案才能连接到 FTP 资源和其他方案上去。HTTP/1.0 要求代理请求发送完整的 URI，解决了这个问题，但它为服务器请求保留部分 URI 的形式（已经有相当多的服务器都改为支持完整 URI 了）。[8]

因此，我们要将部分 URI 发送给服务器，将完整 URI 发送给代理。在显式地配置客户端代理设置的情况下，客户端就知道要发布哪种类型的请求了。

(1) 没有设置客户端使用代理时，它会发送部分 URI（参见图 6-15a）。
(2) 设置客户端使用代理时，它会发送完整 URI（参见图 6-15b）。

6.5.2 与虚拟主机一样的问题

代理“缺少方案 / 主机 / 端口”的问题与虚拟主机 Web 服务器面临的问题相同。虚拟主机 Web 服务器会在很多 Web 站点间共享同一个物理 Web 服务器。包含部分 URI（比如 /index.html）的请求到达时，虚拟主机 Web 服务器需要知道目的 Web 站点的主机名（更多信息请参见 5.7.1 节和 18.2 节）。

尽管它们出现的问题相似，但解决方法却有所不同：

- 显式的代理要求在请求报文中使用完整 URI 来解决这个问题；
- 虚拟主机 Web 服务器要求使用 `Host` 首部来承载主机和端口信息。

注 8：现在，HTTP/1.1 要求服务器为代理请求和服务器请求都提供完整的 URI 处理，但实际上，很多已部署的服务器仍然只接受部分 URI。

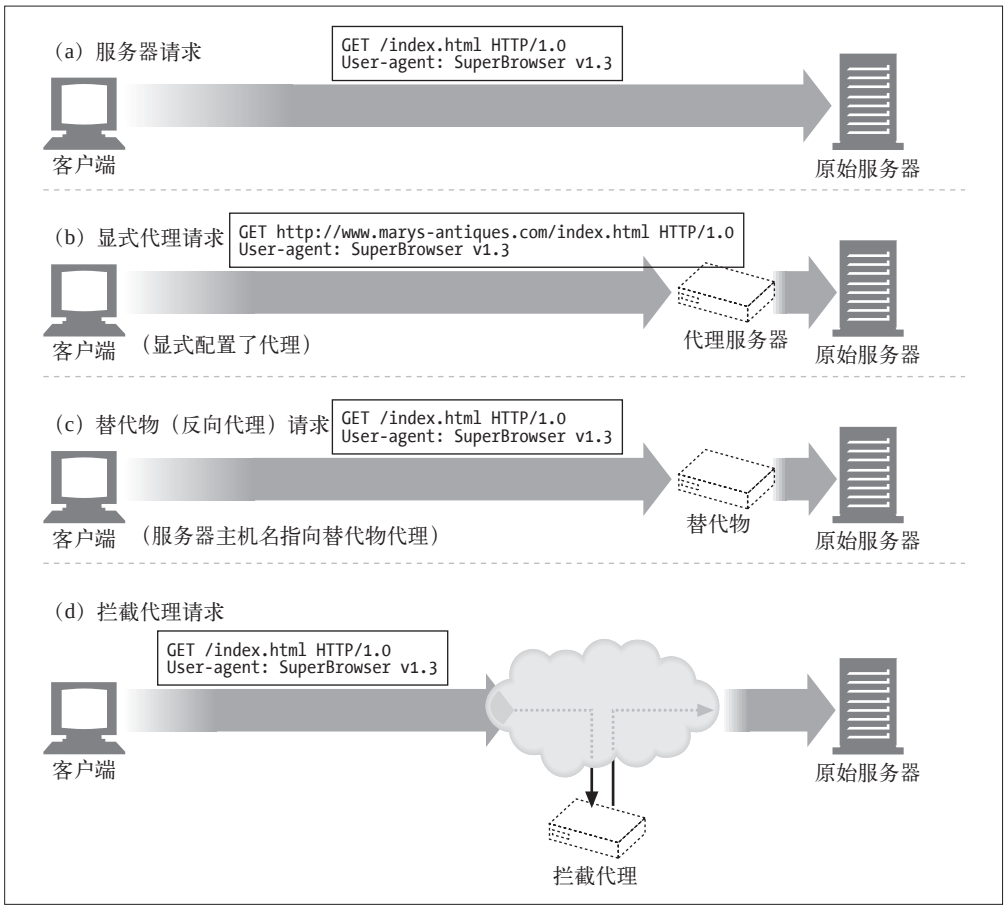


图 6-15　拦截代理会获取服务器请求

6.5.3　拦截代理会收到部分URI

只要客户端正确地实现了 HTTP，它们就会在请求中包含完整的 URI，发送给经过显式配置的代理。这样解决了部分问题，但还有一个问题：客户端*并不总是知道*它是在和代理进行对话，因为有些代理对客户端可能是不可见的。即使没有将客户端配置为使用代理，客户端的流量也可能会经过替代物或拦截代理。在这两种情况下，客户端都会认为它在与 Web 服务器进行对话，不会发送完整的 URI。

- 如前所述，*反向代理*是一个用来取代原始服务器的代理服务器，它通常会通过假扮服务器的主机名或 IP 地址来做到这一点。它会收到 Web 服务器请求，可能会向真正的服务器提供缓存的响应或者代理请求。客户端无法区分反向代理和 Web 服务器，因此它会发送部分 URI（参见图 6-15c）。

- *拦截代理*是网络流量中的代理服务器，它会拦截从客户端发往服务器的请求，并提供一个缓存响应，或对其进行转发。由于拦截代理拦截了从客户端到服务器的流量，所以它会收到发送给 Web 服务器的部分 URI（参见图 6-15d）。[9]

6.5.4 代理既可以处理代理请求，也可以处理服务器请求

由于将流量重定向到代理服务器的方式有所不同，通用的代理服务器既应该支持请求报文中的完整 URI，也应该支持部分 URI。如果是显式的代理请求，代理就应该
146 使用完整 URI，如果是 Web 服务器请求，就应该使用部分 URI 和虚拟 `Host` 首部。

使用完整和部分 URI 的规则如下所示。

- 如果提供的是完整 URI，代理就应该使用这个完整 URI。
- 如果提供的是部分 URI，而且有 `Host` 首部，就应该用 `Host` 首部来确定原始服务器的名字和端口号。
- 如果提供的是部分 URI，而且没有 `Host` 首部，就要用其他方法来确定原始服务器：
  - 如果代理是代表原始服务器的替代物，可以用真实服务器的地址和端口号来配置代理；
  - 如果流量被拦截了，而且拦截者也可以提供原始的 IP 地址和端口，代理就可以使用拦截技术提供的 IP 地址和端口号（参见第 20 章）；
  - 如果所有方法都失败了，代理没有足够的信息来确定原始服务器，就必须返回一条错误报文（通常是建议用户升级到支持 `Host` 首部的现代浏览器）。[10]

6.5.5 转发过程中对URI的修改

代理服务器要在转发报文时修改请求 URI 的话，需要特别小心。对 URI 的微小修改，甚至是看起来无害的修改，都可能给下游服务器带来一些互操作性问题。

尤其是，现在已知有些代理会在将 URI 转发给下一跳节点之前将 URI“规范”为标准格式。有些看起来无害的转换行为，比如用显式的“:80”来取代默认的 HTTP 端口，或者用适当的换码转义符来取代非法的保留字符以校正 URI，就可能造成互操作性问题。

总之，代理服务器要尽量宽容一些。它们的目标不是成为强制实现严格协议一致性

注 9：在某些情况下，拦截代理可能也会拦截客户端到代理的流量，在这种情况下，拦截代理可能会收到完整 URI，并需要对其进行处理。由于显式代理的通信端口通常与 HTTP 使用的端口有所不同（通常是 8080 而不是 80），而且拦截代理通常只对端口 80 进行拦截，所以这种情况并不会经常发生。

注 10：不应该经常这么做。因为用户会收到之前从未收到过的神秘错误页面。

的“协议警察”，因为这样可能会严重破坏之前能正常工作的服务。

特别是，HTTP 规范禁止一般的拦截代理在转发 URI 时重写其绝对路径部分。唯一的例外是可以用“/”来取代空路径。

6.5.6 URI的客户端自动扩展和主机名解析

根据是否有代理，浏览器对请求 URI 的解析会有所不同。没有代理时，浏览器会获取你输入的 URI，尝试着寻找相应的 IP 地址。如果找到了主机名，浏览器会尝试相应的 IP 地址直到获得成功的连接为止。

但是，如果没有找到主机，很多浏览器都会尝试着提供某种主机名自动“扩展”机制，以防用户输入的是主机“简短”的缩写形式（回顾一下 2.3.2 节）。[11]

- 很多浏览器会尝试着加入前缀 www. 和后缀 .com，以防用户只输入了常见 Web 站点名的中间部分（比如，人们可以输入 yahoo 而不是 www.yahoo.com）。
- 有些浏览器甚至会将未解析出来的 URI 传递给第三方站点，这个站点会尝试着校正拼写错误，并给出一些用户可能希望访问的 URI 建议。
- 而且，大多数系统中的 DNS 配置允许用户只输入主机名的前缀，然后 DNS 会自动搜索域名。如果用户位于域名 oreilly.com 的范围之内，并输入了主机名 host7，DNS 会自动尝试将其与 host7.oreilly.com 进行匹配。这并不是完整有效的主机名。

6.5.7 没有代理时URI的解析

图 6-16 显示了一个在没有代理的情况下，浏览器进行主机名自动扩展的例子。在第 (2a) ~ (3c) 步中，浏览器会去查找各种主机名，直到找到一个有效主机名为止。

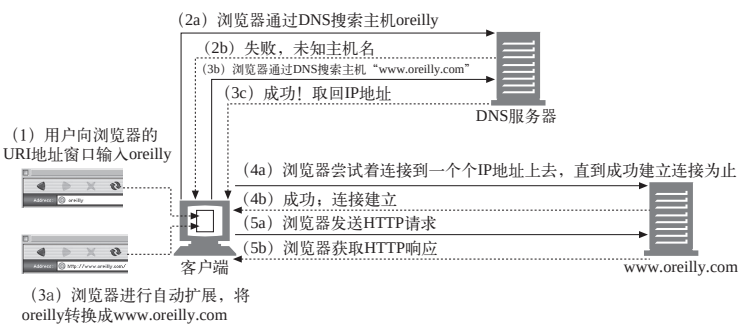


图 6-16 没有提供显式的代理时，浏览器会对部分主机名进行自动扩展

注 11：当你输入 yahoo 时，大多数浏览器都会自动将其扩展为 www.yahoo.com。类似地，浏览器允许用户省略前缀 http://，如果省略了浏览器会自行插入。

图中发生的情况如下所示。

- 在第 (1) 步中，用户向浏览器的 URI 窗口中输入 oreilly。浏览器用 oreilly 作为主机名，并假定默认方案为 http://，默认端口为 80，默认路径为“/”。
148
- 在第 (2a) 步中，浏览器会去查找主机 oreilly。查找失败了。
- 在第 (3a) 步中，浏览器对主机名进行自动扩展，请求 DNS 解析 www.oreilly.com。这次成功了。
- 然后，浏览器成功地连接 www.oreilly.com。

6.5.8 有显式代理时URI的解析

使用显式代理时，用户的 URI 会被直接发送给代理，所以浏览器就不再执行所有这些便捷的扩展功能了。

如图 6-17 所示，有显式代理时，浏览器没有对不完整的主机名进行自动扩展。因此，当用户在浏览器的地址窗口中输入 oreilly 时，发送给代理的就是 http://oreilly/（浏览器添加了默认方案和路径，但主机名和输入的一样）。

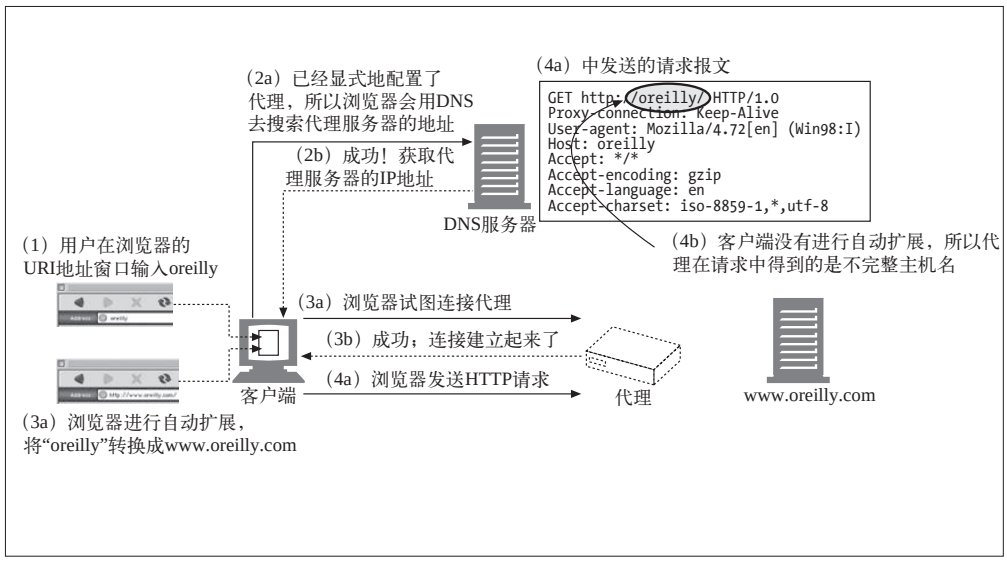


图 6-17　有显式代理时，浏览器不会对不完整主机名进行自动扩展

因此，有些代理会尽力尝试着去模仿浏览器的便捷服务，包括 www...com 自动扩展，以及添加本地域名后缀。[12]

注 12：但对得到广泛共享的代理来说，知道单个用户的正确域名后缀基本上是不可能的。

6.5.9 有拦截代理时URI的解析

使用不可见的拦截代理时，对主机名的解析会略有不同，因为对客户端来说，是没有代理的！这种情况下的行为与使用服务器的情形很类似，浏览器会自动扩展主机名，直到 DNS 成功为止。但如图 6-18 所示，建立到服务器的连接时，有一个很重要的区别。

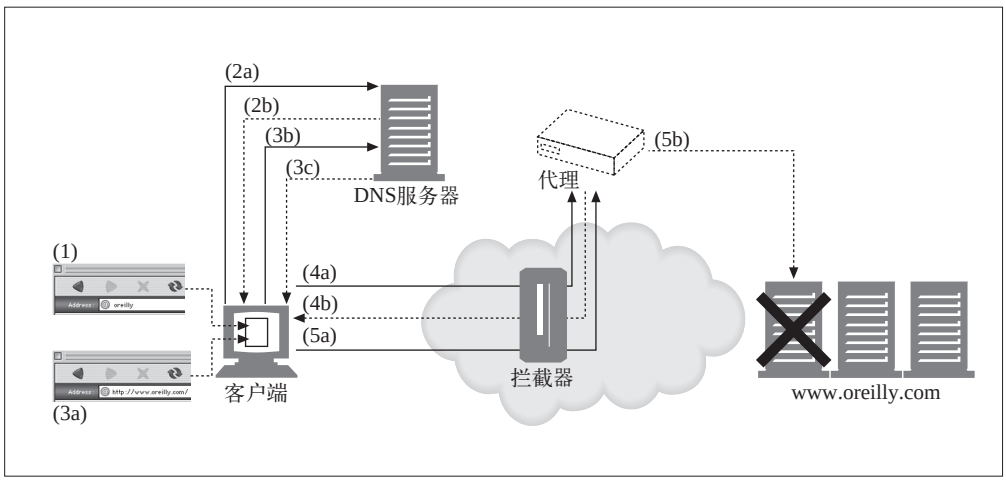


图 6-18　使用拦截代理时，浏览器无法检测出已停用服务器的 IP 地址

图 6-18 显示了下列事务处理过程。

- 在第 (1) 步中，用户在浏览器的 URI 地址窗口中输入 oreilly。
- 在第 (2a) 步中，浏览器通过 DNS 查找主机 oreilly，但 DNS 服务器失败了，并回送响应说明主机未知，如第 (2b) 步所示。
- 在第 (3a) 步中，浏览器进行了自动扩展，将 oreilly 转换成 www.oreilly.com。在第 (3b) 步中，浏览器通过 DNS 来查找主机 www.oreilly.com。这一次，如第 (3c) 步所示，DNS 服务器成功了，将 IP 地址返回给了浏览器。
- 在第 (4a) 步中，客户端已经成功解析了主机名，并有了一张 IP 地址列表。有些 IP 地址可能已经停用了，所以，通常客户端会尝试着连接每个 IP 地址，直到成功为止。但对拦截代理来说，第一次连接请求就会被代理服务器拦截成功，不会连接到原始服务器上去。客户端认为它在与 Web 服务器进行成功的对话，但那个 Web 服务器可能甚至都不处于活跃状态。
- 当代理最终准备好与真正的原始服务器进行交互时［第 (5b) 步］，代理可能会发现那个 IP 地址实际指向的是一个已停用的服务器。为了提供与浏览器相同级别的容错机制，代理可以通过解析 `Host` 首部的主机名，也可以通过对 IP 地址的反

向 DNS 查找来尝试其他 IP 地址。将浏览器配置为使用显式代理时，它们会依赖代理的容错机制，所以对拦截和显式的代理实现来说，在 DNS 解析到已停用服务器时，提供容错机制是很重要的。

6.6 追踪报文

现在，在将 Web 请求从客户端传送到服务器的路径上，经过两个或多个代理是很常见的（参见图 6-19）。比如，出于安全和节省费用的考虑，很多公司都会用缓存代理服务器来访问因特网，而且很多大型 ISP 都会使用代理缓存来提高性能并实现各种特性。现在，有相当比例的 Web 请求都是通过代理转发的。同时，出于性能原因，把内容复制到遍布全球的替代物缓存库中的情形也越来越常见了。

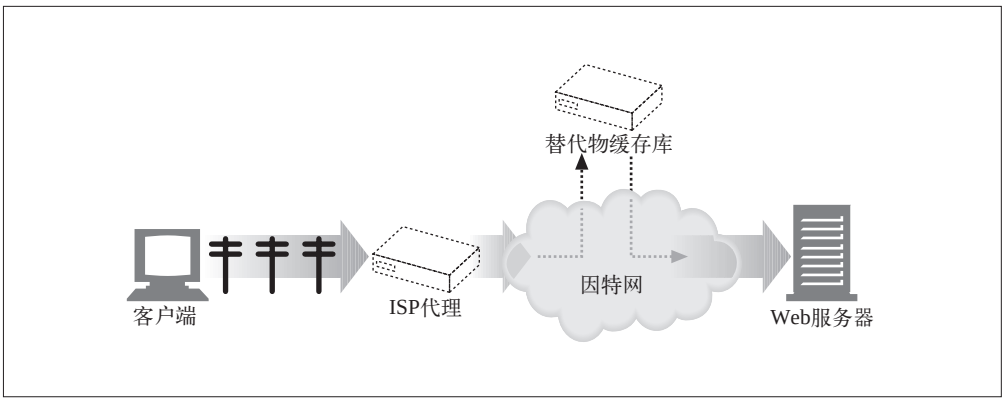


图 6-19　访问代理和 CDN 代理构建了一个两级代理层次结构

代理是由不同厂商开发的。它们有不同的特性和缺陷，由各种不同的组织负责管理。

随着代理的逐渐流行，我们要能够追踪经过代理的报文流，以检测出各种问题，其重要性就跟追踪经过不同交换机和路由器传输的 IP 分组流一样。

6.6.1 Via首部

Via 首部字段列出了与报文途经的每个中间节点（代理或网关）有关的信息。报文每经过一个节点，都必须将这个中间节点添加到 Via 列表的末尾。

下面的 Via 字符串告诉我们报文流经了两个代理。这个字符串说明第一个代理名为 proxy-62.irenes-isp.net，它实现了 HTTP/1.1 协议，第二个代理被称为 cache.joes-hardware.com，实现了 HTTP/1.0：

```
Via: 1.1 proxy-62.irenes-isp.net, 1.0 cache.joes-hardware.com
```

Via 首部字段用于记录报文的转发，诊断报文循环，标识请求 / 响应链上所有发送者的协议能力（参见图 6-20）。

代理也可以用 Via 首部来检测网络中的路由循环。代理应该在发送一条请求之前，在 Via 首部插入一个与其自身有关的独特字符串，并在输入的请求中查找这个字符串，以检测网络中是否存在路由循环。

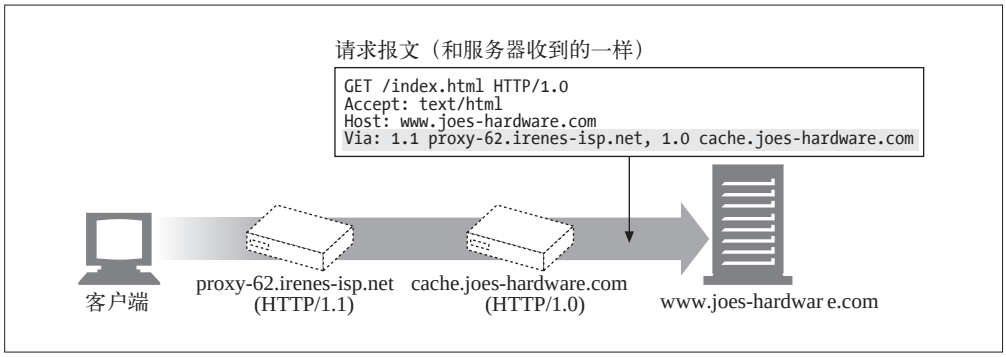


图 6-20　Via 首部实例

1. Via的语法

Via 首部字段包含一个由逗号分隔的路标（waypoint）。每个路标都表示一个独立的代理服务器或网关，且包含与那个中间节点的协议和地址有关的信息。下面是一个带有两个路标的 Via 首部实例：

```
Via = 1.1 cache.joes-hardware.com, 1.1 proxy.irenes-isp.net
```

Via 首部的正规语法如下所示：

```
Via               = "Via" ":" 1#( waypoint )
waypoint          = ( received-protocol received-by [ comment ] )
received-protocol = [ protocol-name "/" ] protocol-version
received-by       = ( host [ ":" port ] ) | pseudonym
```

注意，每个 Via 路标中最多包含 4 个组件：一个可选的协议名（默认为 HTTP）、一个必选的协议版本、一个必选的节点名和一个可选的描述性注释。

- 协议名

  中间节点收到的协议。如果协议为 HTTP 的话，协议名就是可选的。否则，要在版本之前加上协议名，中间用“/”分隔。网关将 HTTP 请求连接到其他协议（HTTPS、FTP 等）时，可能会使用非 HTTP 协议。

- 协议版本
  所收到的报文版本。版本的格式与协议有关。HTTP 使用的是标准版本号（1.0、1.1 等）。版本包含在 Via 字段中，这样，之后的应用程序就会知道前面所有中间节点的协议能力了。

- 节点名
  中间节点的主机和可选端口号（如果没有包含端口号，可以假定使用的是协议的默认端口号）。在某些情况下，出于隐私方面的考虑，某个组织可能不愿意给出
152 真实的主机名，在这种情况下可以用一个假名来代替。

- 节点注释
  进一步描述这个中间节点的可选注释。通常会在这里包含厂商和版本信息，有些代理服务器还会在注释字段中包含一些与此设备上所发生事件有关的诊断信息。[13]

2. Via的请求和响应路径

请求和响应报文都会经过代理进行传输，因此，请求和响应报文中都要有 Via 首部。

请求和响应通常都是通过同一条 TCP 连接传送的，所以响应报文会沿着与请求报文相同的路径回传。如果一条请求报文经过了代理 A、B 和 C，相应的响应报文就会通过代理 C、B、A 进行传输。因此，响应的 Via 首部基本上总是与请求的 Via 首部相反（参见图 6-21）。

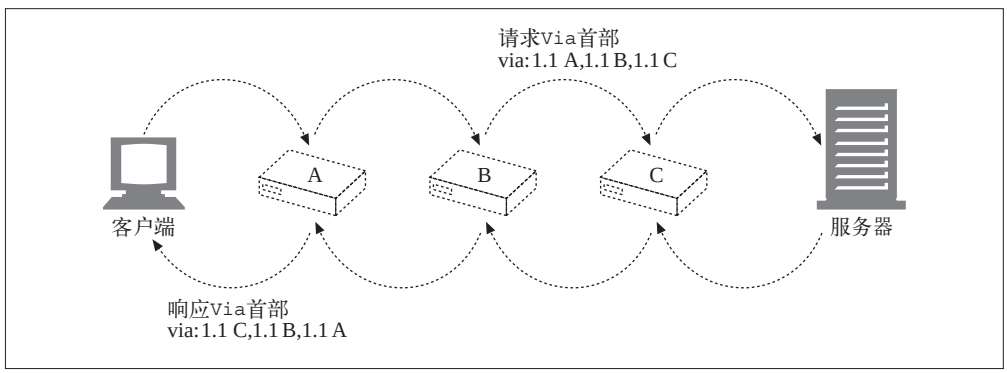


图 6-21　响应 Via 通常与请求 Via 相反

注 13：比如，缓存代理服务器中可能会包含一些成功 / 失败信息。

3. Via与网关

有些代理会为使用非 HTTP 协议的服务器提供网关的功能。Via 首部记录了这些协议转换，这样，HTTP 应用程序就会了解代理链上各点的协议处理能力以及所做的协议转换了。图 6-22 显示了一个通过 HTTP/FTP 网关请求某个 FTP URI 的 HTTP 客户端。

客户端向网关 proxy.irenes-isp.net 发送了一条对 ftp://http-guide.com/pub/welcome.tst 的 HTTP 请求。作为协议网关使用的代理会用 FTP 协议从 FTP 服务器获取预期的对象。然后代理会用下面这个 Via 首部字段，在一条 HTTP 响应中将对象回送到客户端上去：

```
Via: FTP/1.0 proxy.irenes-isp.net (Traffic-Server/5.0.1-17882 [cMs f ])
```


注意，接收到的协议是 FTP。可选注释中包含有代理服务器的品牌和版本号，以及一些厂商的诊断信息。在第 8 章可以读到所有与网关有关的内容。

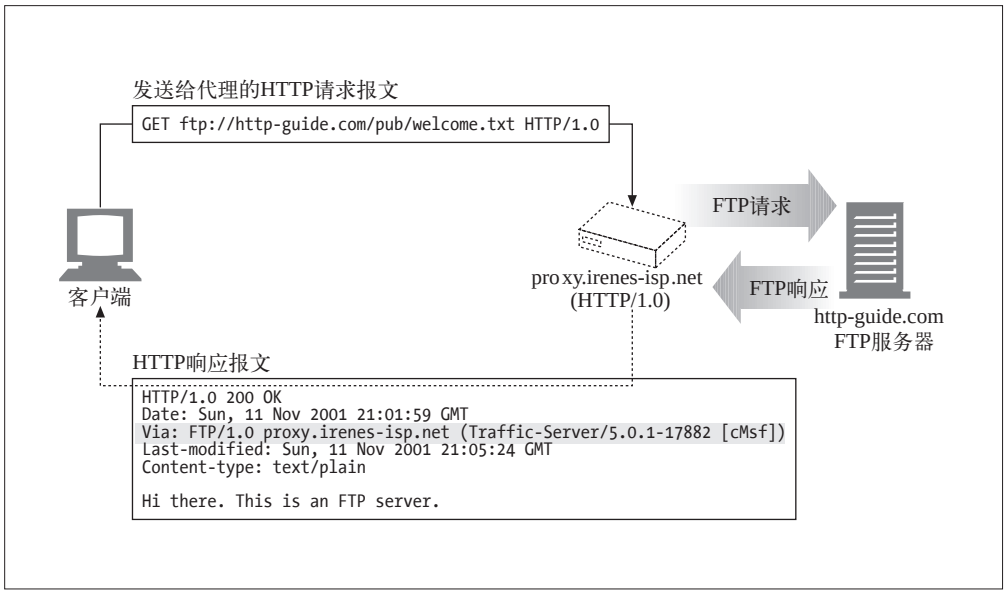


图 6-22　HTTP/FTP 网关生成了 Via 首部，用于记录所收到的协议（FTP）

4. Server和Via首部

Server 响应首部字段对原始服务器使用的软件进行了描述。这里有几个例子：

```
Server: Apache/1.3.14 (Unix) PHP/4.0.4
Server: Netscape-Enterprise/4.1
Server: Microsoft-IIS/5.0
```

如果响应报文是通过代理转发的，一定要确保代理没有修改 Server 首部。Server 首部是用于原始服务器的。代理应该添加的是 Via 条目。

5．Via的隐私和安全问题

有时候，我们并不希望在 Via 字符串中使用确切的主机名。总地来说，除非显式地允许了这种行为，否则，当代理服务器作为网络防火墙的一部分使用时，是不应该转发防火墙后面那些主机的名字和端口号的，因为防火墙后面的网络结构信息可能

会被恶意群体利用。[14]

如果不允许进行 Via 节点名转发，作为安全防线的一部分使用的代理就应该用适当的假名来取代那台主机的名字。一般来说，即使隐藏了真实名称，代理也应该尝试着为每台代理服务器保留一个 Via 路标条目。

对那些有着非常强烈的隐私要求，需要隐藏内部网络设计和拓扑结构的组织来说，代理应该将一个（接收协议值相同的）有序 Via 路标条目序列合并成一个联合条目。比如，可以将：

```
Via: 1.0 foo, 1.1 devirus.company.com, 1.1 access-logger.company.com
```

压缩成：

```
Via: 1.0 foo, 1.1 concealed-stuff
```

除非这些条目都在同一个组织的控制之下，而且已经用假名取代了主机名，否则就不能将其合并起来。同样，接收协议值不同的条目也不能合并起来。

6.6.2 TRACE方法

代理服务器可以在转发报文时对其进行修改。可以添加、修改或删除首部，也可以将主体部分转换成不同的格式。代理变得越来越复杂，开发代理产品的厂商也越来越多，互操作性问题也开始逐渐显现。为了便于对代理网络进行诊断，我们需要有一种便捷的方式来观察在通过 HTTP 代理网络逐跳转发报文的过程中，报文是怎样变化的。

通过 HTTP/1.1 的 TRACE 方法，用户可以跟踪经代理链传输的请求报文，观察报文经过了哪些代理，以及每个代理是如何对请求报文进行修改的。TRACE 对代理流的调试非常有用。[15]

注 14：恶意用户可以通过计算机名字和版本号来了解安全防线之后的网络结构。这类信息可能有助于进行安全攻击。而且，计算机名还可能泄露一个组织内部私有项目的线索。

注 15：但是，它还没有得到广泛的实现。

当 TRACE 请求到达目的服务器时，[16] 整条请求报文都会被封装在一条 HTTP 响应的主体中回送给发送端（参见图 6-23）。当 TRACE 响应到达时，客户端可以检查服务器收到的确切报文，以及它所经过的代理列表（在 Via 首部）。TRACE 响应的 Content-Type 为 message/http，状态为 200 OK。

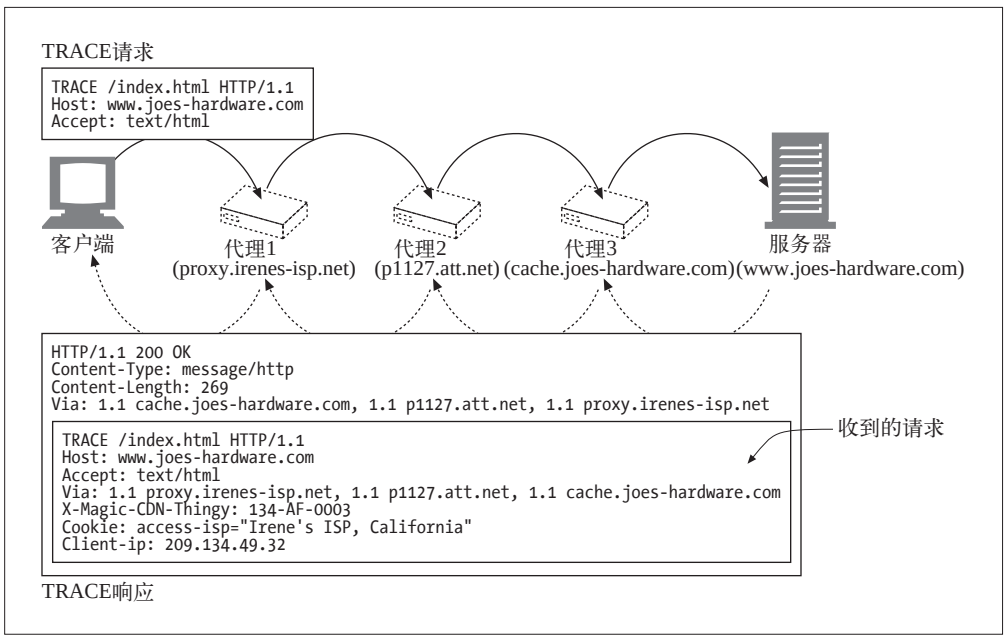


图 6-23　TRACE 响应回送了收到的请求报文

Max-Forwards

通常，不管中间插入了多少代理，TRACE 报文都会沿着整条路径传到目的服务器上。可以使用 Max-Forwards（最大转发次数）首部来限制 TRACE 和 OPTIONS 请求所经过的代理跳数，在测试代理链是否是在无限循环中转发报文，或者查看链中特定代理服务器的效果时，它是很有用的。Max-Forwards 也可以限制 OPTIONS 报文的转发（参见 6.8 节）。

155

Max-Forwards 请求首部字段包含了一个整数，用来说明这条请求报文还可以被转发的次数（参见图 6-24）。如果 Max-Forwards 的值为零（Max-Forwards:0），那么即使接收者不是原始服务器，它也必须将 TRACE 报文回送给客户端，而不应该继续转发。

注 16：最后的接收者可以是原始服务器，也可以是第一个收到了 Max-Forwards 值为零的请求的代理或网关。

如果收到的 Max-Forwards 值大于零，转发的报文中就必须包含一个更新了的 Max-Forwards 字段，其值会被减一。所有的代理和网关都应该支持 Max-Forwards。可以用 Max-Forwards 来查看在代理链的任意一跳上接收到的请求。

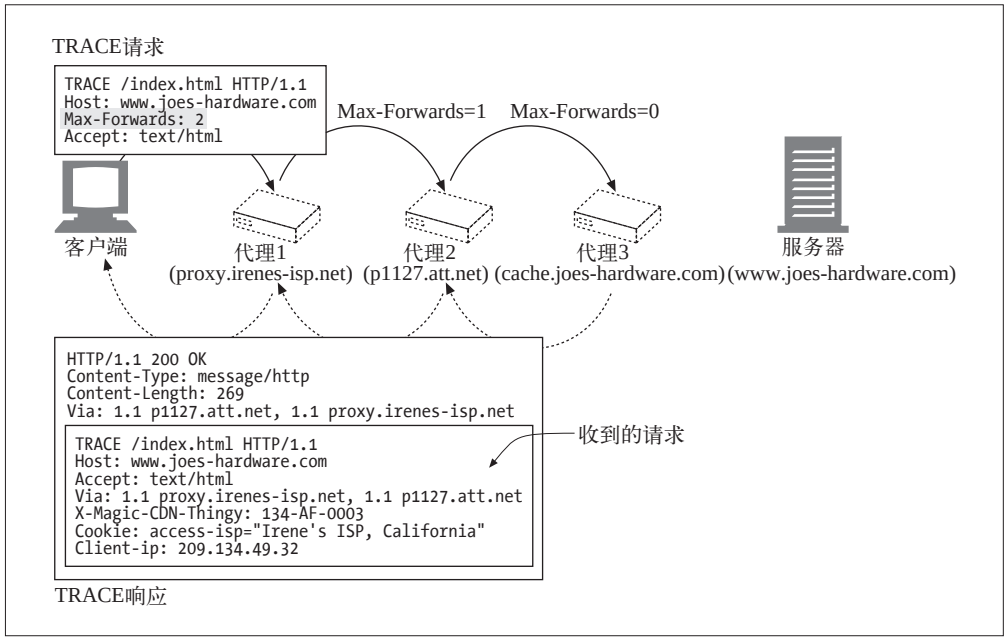


图 6-24　可以用 **Max-Forwards** 首部字段来限制转发跳数

6.7　代理认证

代理可以作为访问控制设备使用。HTTP 定义了一种名为代理认证（proxy authentication）的机制，这种机制可以阻止对内容的请求，直到用户向代理提供了有效的访问权限证书为止。

- 对受限内容的请求到达一台代理服务器时，代理服务器可以返回一个要求使用访问证书的 407 Proxy Authorization Required 状态码，以及一个用于描述怎样提供这些证书的 Proxy-Authenticate 首部字段（参见图 6-25b）。
- 客户端收到 407 响应时，会尝试着从本地数据库中，或者通过提示用户来搜集所需要的证书。
- 只要获得了证书，客户端就会重新发送请求，在 Proxy-Authorization 首部字段中提供所要求的证书。
- 如果证书有效，代理就会将原始请求沿着传输链路向下传送（参见图 6-25c）；否则，就发送另一条 407 应答。

156

若传输链路中有多个代理，且每个代理都要进行认证时，代理认证通常无法很好地工作。人们建议，应该对 HTTP 进行升级，将认证证书与代理链中特定的路标联系起来，但这些升级措施并没有得到广泛实现。

有关 HTTP 认证机制的详细解释请参见第 12 章。

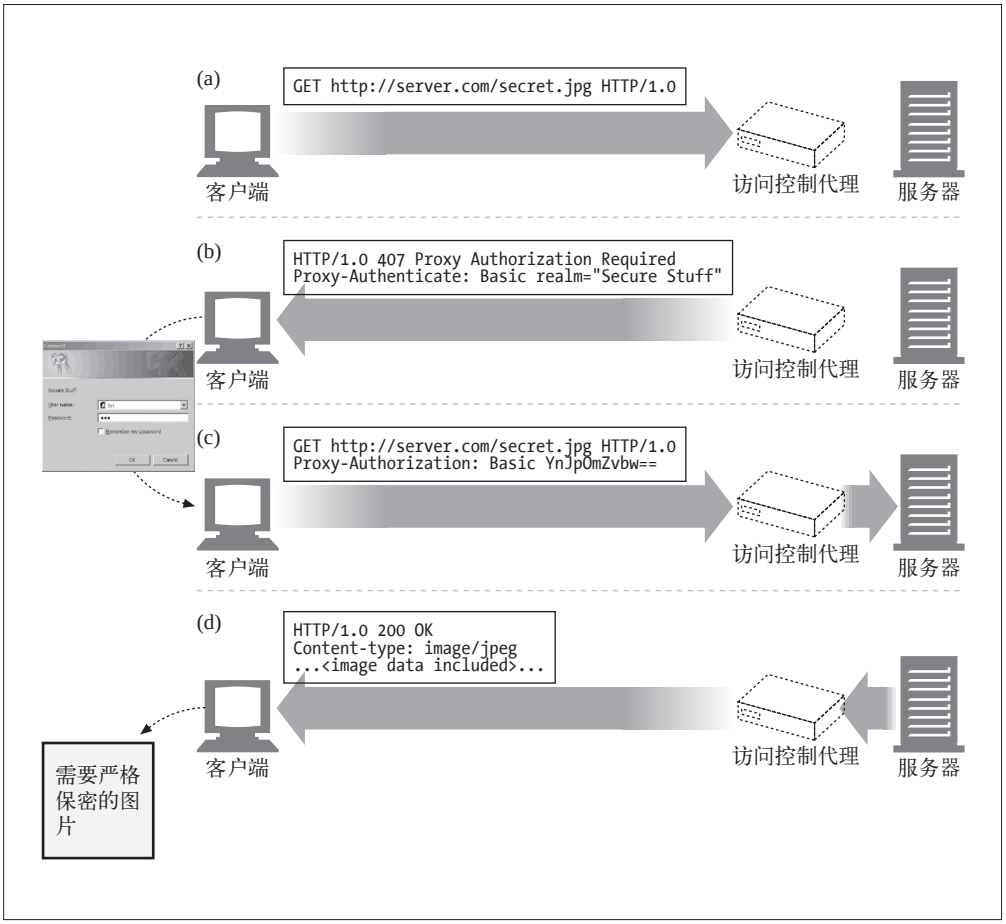


图 6-25　代理可以实现认证机制以控制对内容的访问

6.8　代理的互操作性

客户端、服务器和代理是由不同厂商构建的，实现的是不同版本的 HTTP 规范。它们支持的特性各不相同，也存在着不同的问题。代理服务器位于客户端和服务器设备之间，这些设备实现的协议可能有所不同，可能存在着很棘手的问题。

157

6.8.1 处理代理不支持的首部和方法

代理服务器可能无法理解所有经其传输的首部字段。有些首部可能比代理自身还要新；其他首部可能是特定应用程序独有的定制首部。代理必须对不认识的首部字段进行转发，而且必须维持同名首部字段的相对顺序。[17] 类似地，如果代理不熟悉某个方法，那么只要可能，就应该尝试着将报文转发到下一跳节点上去。

在当今的大部分网络中，如果代理不能转发它不支持的方法，可能就无法生存下去了，因为通过微软的 Outlook 进行 Hotmail 访问就大量地使用了 HTTP 扩展方法。

158

6.8.2 OPTIONS：发现对可选特性的支持

通过 HTTP OPTIONS 方法，客户端（或代理）可以发现 Web 服务器或者其上某个特定资源所支持的功能（比如，它们所支持的方法）（参见图 6-26）。通过使用 OPTIONS，客户端可以在与服务器进行交互之前，确定服务器的能力，这样它就可以更方便地与具备不同特性的代理和服务器进行互操作了。

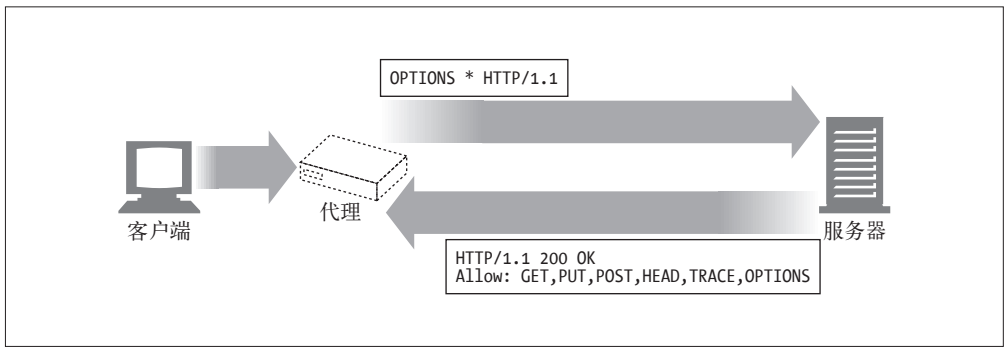


图 6-26 用 OPTIONS 来判定服务器支持的方法

如果 OPTIONS 请求的 URI 是个星号（*），请求的就是整个服务器所支持的功能。比如：

```
OPTIONS * HTTP/1.1
```

如果 URI 是个实际资源地址，OPTIONS 请求就是在查询那个特定资源的可用特性：

```
OPTIONS http://www.joes-hardware.com/index.html HTTP/1.1
```

注 17：报文中可能会出现多个报文首部字段具有相同字段名的情况，如果存在这种情况的话，就要将其等价地合并为一个由逗号分隔的列表。因此，要对合并后的字段值进行解释，具有相同字段名的首部字段的接收顺序就变得非常重要了，因此，代理在转发报文时，就不能修改这些同名字段值的相对顺序。

如果成功，OPTIONS 方法就会返回一个包含了各种首部字段的 200 OK 响应，这些字段描述了服务器所支持的，或资源可用的各种可选特性。HTTP/1.1 在响应中唯一指定的首部字段是 Allow 首部，这个首部用于描述服务器所支持的各种方法（或者服务器上的特定资源）。[18]OPTIONS 允许在可选的响应主体中包含更多的信息，但并没有对这种用法进行定义。

6.8.3 Allow首部

Allow 实体首部字段列出了请求 URI 标识的资源所支持的方法列表，如果请求 URI 为 * 的话，列出的就是整个服务器所支持的方法列表。例如：

```
Allow: GET, HEAD, PUT
```

可以将 Allow 首部作为请求首部，建议在新的资源上支持某些方法。并不要求服务器支持这些方法，但应该在相应的响应中包含一个 Allow 首部，列出它实际支持的方法。

159

因为客户端可能已经通过其他途径与原始服务器进行了交流，所以即使代理无法理解指定的所有方法，也不能对 Allow 首部字段进行修改。

6.9 更多信息

更多信息，请参见以下资源。

- http://www.w3.org/Protocols/rfc2616/rfc2616.txt
  RFC 2616，R. Fielding、J. Gettys、J. Mogul、H. Frystyk、L. Mastinter、P. Leach 和 T. Berners-Lee 编写的“Hypertext Transfer Protocol”。
- http://www.ietf.org/rfc/rfc3040.txt
  RFC 3040，“Internet Web Replication and Caching Taxonomy”（“因特网 Web 复制和缓存分类”）。
- *Web Proxy Servers*
  Ari Luotonen 著，Prentice Hall 出版的计算机图书。
- http://www.ietf.org/rfc/rfc3143.txt
  RFC 3143，“Known HTTP Proxy/Caching Problems”（“已知的 HTTP 代理 / 缓存问题”）。

注 18：并不是所有资源都支持每种方法的。比如，CGI 脚本查询可能就不支持文件 PUT，而静态的 HTML 文件则不接受 POST 方法。

- *Web Caching*（《Web 缓存》[19]）
160 Duane Wessels 著，O'Reilly & Associates 公司出版。

注 19：影印版由清华大学出版社出版。（编者注）

第7章

缓　存

Web 缓存是可以自动保存常见文档副本的 HTTP 设备。当 Web 请求抵达缓存时，如果本地有“已缓存的”副本，就可以从本地存储设备而不是原始服务器中提取这个文档。使用缓存有下列优点。

- 缓存减少了冗余的数据传输，节省了你的网络费用。
- 缓存缓解了网络瓶颈的问题。不需要更多的带宽就能够更快地加载页面。
- 缓存降低了对原始服务器的要求。服务器可以更快地响应，避免过载的出现。
- 缓存降低了距离时延，因为从较远的地方加载页面会更慢一些。

本章解释了缓存是怎样提高性能降低费用的、如何去衡量其有效性以及将缓存置于何处可以发挥它的最大作用。我们还会解释 HTTP 如何保持已缓存副本的新鲜度，缓存如何与其他缓存和服务器通信等问题。

7.1 冗余的数据传输

有很多客户端访问一个流行的原始服务器页面时，服务器会多次传输同一份文档，每次传送给一个客户端。一些相同的字节会在网络中一遍遍地传输。这些冗余的数据传输会耗尽昂贵的网络带宽，降低传输速度，加重 Web 服务器的负载。有了缓存，就可以保留第一条服务器响应的副本，后继请求就可以由缓存的副本来应对了，这样可以减少那些流入 / 流出原始服务器的、被浪费掉了的重复流量。

7.2 带宽瓶颈

缓存还可以缓解网络的瓶颈问题。很多网络为本地网络客户端提供的带宽比为远程服务器提供的带宽要宽（参见图 7-1）。客户端会以路径上最慢的网速访问服务器。如果客户端从一个快速局域网的缓存中得到了一份副本，那么缓存就可以提高性
161 能——尤其是要传输比较大的文件时。

在图 7-1 中，Joe 的五金商店旧金山分店的用户通过 1.4Mbit/s 的 T1 因特网连接，从亚特兰大总店下载一个 5MB 的库存文件要花 30 秒的时间。如果在旧金山分店里缓存了这个文档，本地用户通过以太网连接只要花费不到 1 秒的时间就可以获得同一份文档了。

表 7-1 说明了在几种不同的网速下，传输几种不同大小的文档时，带宽会对传输速度产生什么样的影响。带宽会给较大的文档带来显而易见的时延，不同类型网络的速度差异会非常明显。[1] 一个 54kbit/s 的 Modem 传输一个 5MB 的文件需要 749 秒（超过 12 分钟），而在快速以太网 LAN 中，只要不到一秒的时间。

注 1：这张表只列出了网络带宽对传输时间的影响。它假定网络效率为 100%，而且不存在网络或应用程序的处理时延。通过这种方式给出的时延是下限值。实际的时延要大一些，而小型对象的时延则主要是由非带宽开销造成的。

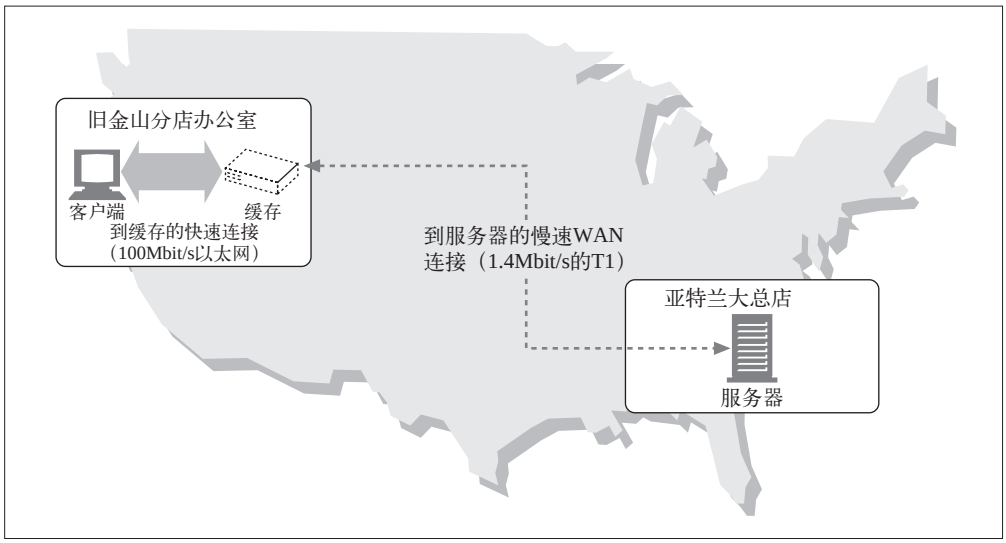


图 7-1　缓存可以改善由有限广域带宽造成的网络瓶颈

表7-1　带宽造成的传输时延，理想化情况（以秒为时间单位）

	大型HTML（15KB）	JPEG（40KB）	大型JPEG（150KB）	大型文件（5MB）
拨号 modem（56kbit/s）	2.19	5.85	21.94	748.98
DSL（256Kbit/s）	0.48	1.28	4.80	163.84
T1（1.4Mbit/s）	0.09	0.23	0.85	29.13
慢速以太网（10Mbit/s）	0.01	0.03	0.12	4.19
DS3（45Mbit/s）	0.00	0.01	0.03	0.93
快速以太网（100Mbit/s）	0.00	0.00	0.01	0.42

7.3　瞬间拥塞

缓存在破坏瞬间拥塞（Flash Crowds）时显得非常重要。突发事件（比如爆炸性新闻、批量 E-mail 公告，或者某个名人事件）使很多人几乎同时去访问一个 Web 文档时，就会出现瞬间拥塞（参见图 7-2）。由此造成的过多流量峰值可能会使网络和 Web 服务器产生灾难性的崩溃。

1998 年 9 月 11 日，详细描述 Kenneth Starr 对美国总统克林顿调查情况的“Starr 报告”发布到因特网上去的时候，美国众议院的 Web 服务器每小时收到了超过三百万次的请求，是其平均服务器负荷的 50 倍。据报道，新闻站点 CNN.com 的服务器每秒钟平均会收到超过 50 000 次的请求。

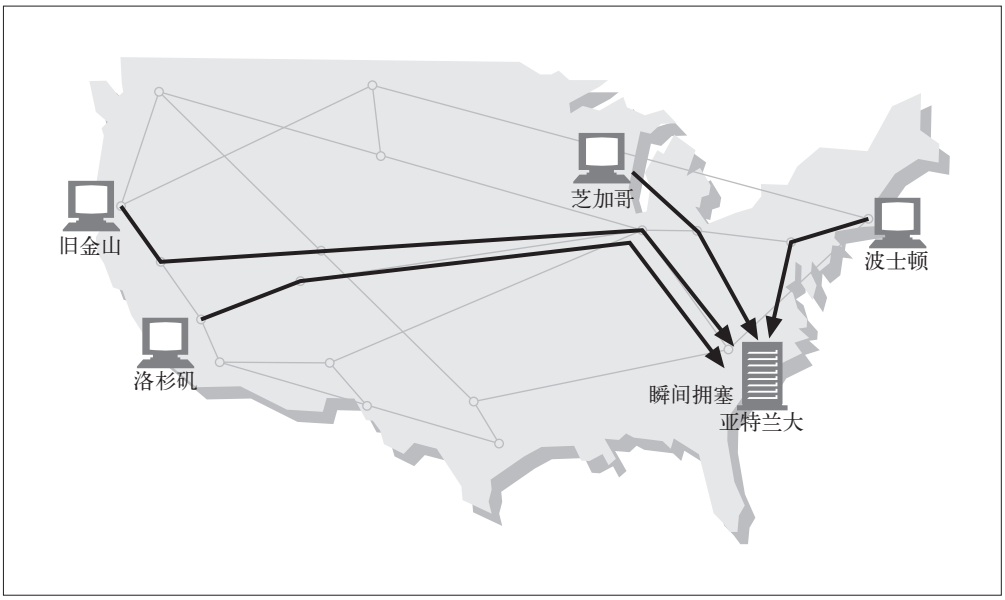


图 7-2　瞬间拥塞会使 Web 服务器过载

7.4　距离时延

即使带宽不是问题，距离也可能成为问题。每台网络路由器都会增加因特网流量的时延。即使客户端和服务器之间没有太多的路由器，光速自身也会造成显著的时延。

波士顿到旧金山的直线距离大约有 2700 英里。在最好的情况下，以光速传输（186 000 英里 / 秒）的信号可以在大约 15 毫秒内从波士顿传送到旧金山，并在 30
163 毫秒内完成一个往返。[2]

假设某个 Web 页面中包含了 20 个小图片，都在旧金山的一台服务器上。如果波士顿的一个客户端打开了 4 条到服务器的并行连接，而且保持着连接的活跃状态，光速自身就要耗费大约 1/4 秒（240 毫秒）的下载时间（参见图 7-3）。如果服务器位于（距离旧金山 6700 英里的）东京，时延就会变成 600 毫秒。中等复杂的 Web 页面会带来几秒钟的光速时延。

将缓存放在附近的机房里可以将文件传输距离从数千英里缩短为数十米。

注 2：在实际应用中，信号的传输速度会比光速低一些，因此，距离时延会更加严重。

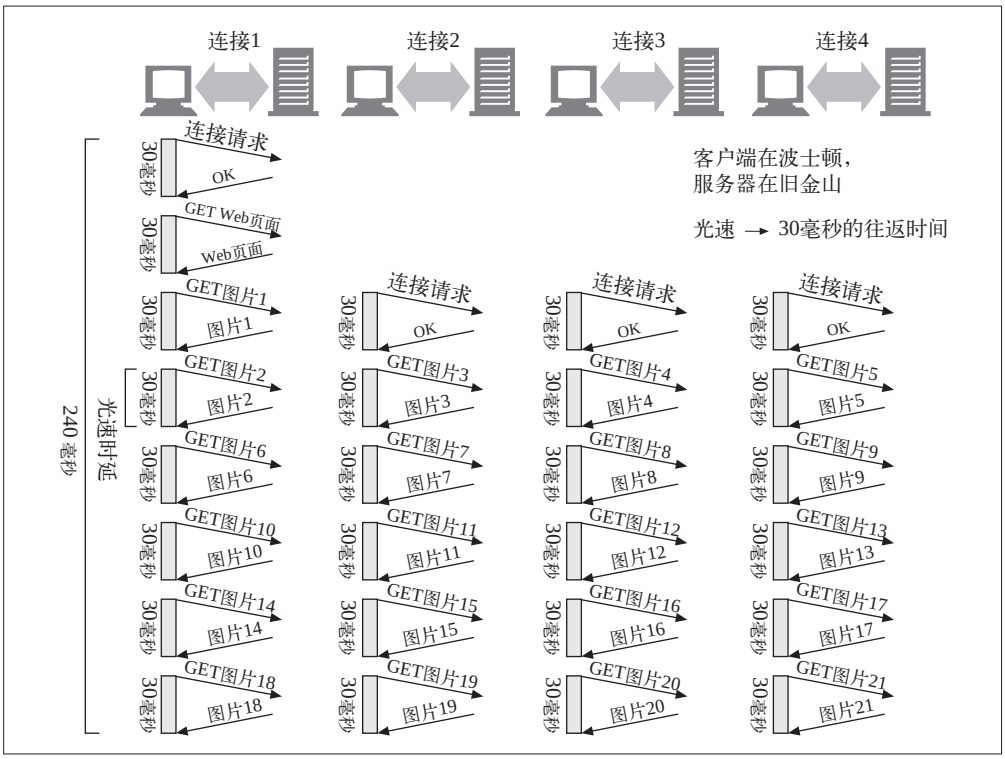


图 7-3　即便使用的是并行的持久连接，光速也会造成显著的时延

7.5　命中和未命中的

这样看来缓存是有所帮助的。但缓存无法保存世界上每份文档的副本。[3]

164

可以用已有的副本为某些到达缓存的请求提供服务。这被称为*缓存命中*（cache hit），参见图 7-4a。其他一些到达缓存的请求可能会由于没有副本可用，而被转发给原始服务器。这被称为*缓存未命中*（cache miss），参见图 7-4b。

7.5.1　再验证

原始服务器的内容可能会发生变化，缓存要不时对其进行检测，看看它们保存的副本是否仍是服务器上最新的副本。这些“新鲜度检测”被称为 HTTP *再验证*（revalidation）（参见图 7-4c）。为了有效地进行再验证，HTTP 定义了一些特殊的请求，不用从服务器上获取整个对象，就可以快速检测出内容是否是最新的。

注 3：几乎没人能够买得起一个大得足以装下 Web 上所有文档的缓存。即便可以买得起巨大的“整个 Web 的缓存”，有些文档也经常会发生变化，很多缓存中的内容都不是最新的。这样的话，在很多缓存中都无法对其进行及时的更新。

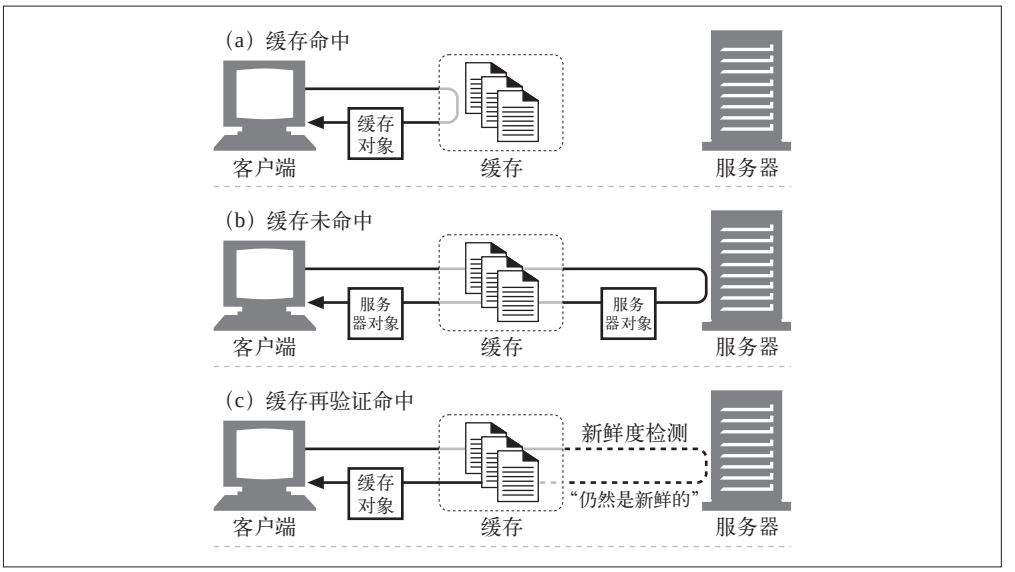


图 7-4　缓存命中、未命中以及再验证

缓存可以在任意时刻，以任意的频率对副本进行再验证。但由于缓存中通常会包含数百万的文档，而且网络带宽是很珍贵的，所以大部分缓存只有在客户端发起请求，并且副本旧得足以需要检测的时候，才会对副本进行再验证。本章稍后会解释 HTTP 的新鲜度检测规则。

缓存对缓存的副本进行再验证时，会向原始服务器发送一个小的再验证请求。如果内容没有变化，服务器会以一个小的 304 Not Modified 进行响应。只要缓存知道副本仍然有效，就会再次将副本标识为暂时新鲜的，并将副本提供给客户端（参见图 7-5a）这被称作*再验证命中*（revalidate hit）或*缓慢命中*（slow hit）。这种方式确实要与原始服务器进行核对，所以会比单纯的缓存命中要慢，但它没有从服务器中获
165 取对象数据，所以要比缓存未命中快一些。

HTTP 为我们提供了几个用来对已缓存对象进行再验证的工具，但最常用的是 `If-Modified-Since` 首部。将这个首部添加到 GET 请求中去，就可以告诉服务器，只有在缓存了对象的副本之后，又对其进行了修改的情况下，才发送此对象。

这里列出了在 3 种情况下（服务器内容未被修改，服务器内容已被修改，或者服务器上的对象被删除了）服务器收到 GET `If-Modified-Since` 请求时会发生的情况：

- *再验证命中*
  如果服务器对象未被修改，服务器会向客户端发送一个小的 HTTP 304 Not Modified 响应。图 7-6 对此进行了描述。

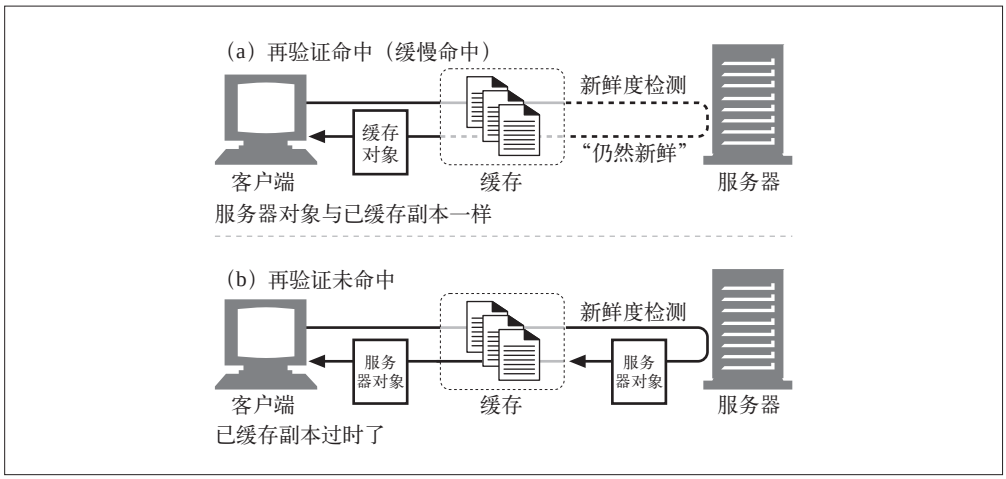


图 7-5　成功的再验证比缓存未命中要快，失败的再验证几乎和未命中的速度一样

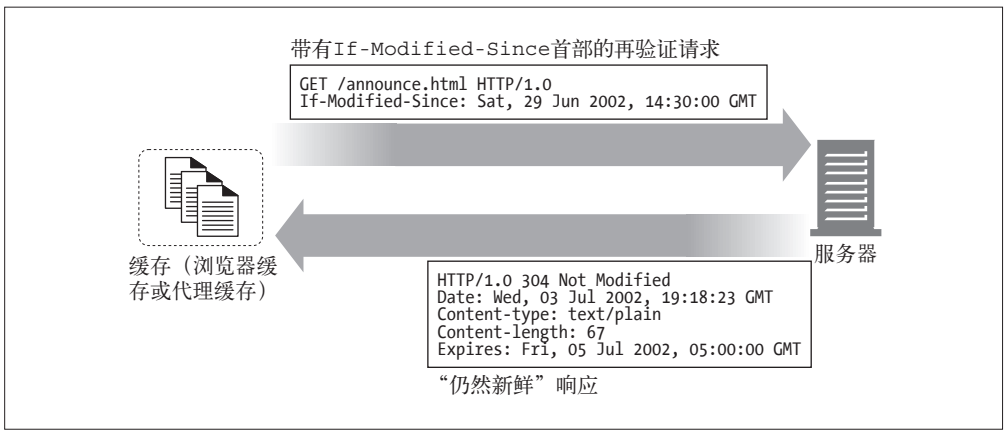


图 7-6　HTTP 使用 `If-Modified-Since` 首部进行再验证

- *再验证未命中*
  如果服务器对象与已缓存副本不同，服务器向客户端发送一条普通的、带有完整内容的 HTTP 200 OK 响应。

166

- *对象被删除*
  如果服务器对象已经被删除了，服务器就回送一个 404 Not Found 响应，缓存也会将其副本删除。

7.5.2　命中率

由缓存提供服务的请求所占的比例被称为*缓存命中率*（cache hit rate，或称为缓存

命中比例），[4] 有时也被称为*文档命中率*（document hit rate）。命中率在 0 到 1 之间，但通常是用百分数来描述的，0% 表示每次请求都未命中（要通过网络来获取文档），100% 表示每次请求都命中了（在缓存中有一份副本）。[5]

缓存的管理者希望缓存命中率接近 100%。而实际得到的命中率则与缓存的大小、缓存用户兴趣点的相似性、缓存数据的变化或个性化频率，以及如何配置缓存有关。命中率很难预测，但对现在中等规模的 Web 缓存来说，40% 的命中率是很合理的。缓存的好处是，即使是中等规模的缓存，其所包含的常见文档也足以显著地提高性能、减少流量了。缓存会努力确保将有用的内容保存在缓存中。

7.5.3 字节命中率

由于文档并不全是同一尺寸的，所以文档命中率并不能说明一切。有些大型对象被访问的次数可能较少，但由于尺寸的原因，对整个数据流量的贡献却更大。因此，有些人更愿意使用*字节命中率*（byte hit rate）作为度量值（尤其那些按流量字节付费的人！）。

字节命中率表示的是缓存提供的字节在传输的所有字节中所占的比例。通过这种度量方式，可以得知节省流量的程度。100% 的字节命中率说明每个字节都来自缓存，没有流量流到因特网上去。

文档命中率和字节命中率对缓存性能的评估都是很有用的。文档命中率说明阻止了多少通往外部网络的 Web 事务。事务有一个通常都很大的固定时间成分（比如，建立一条到服务器的 TCP 连接），提高文档命中率对降低整体延迟（时延）很有好处。
167 字节命中率说明阻止了多少字节传向因特网。提高字节命中率对节省带宽很有利。

7.5.4 区分命中和未命中的情况

不幸的是，HTTP 没有为用户提供一种手段来区分响应是缓存命中的，还是访问原始服务器得到的。在这两种情况下，响应码都是 200 OK，说明响应有主体部分。有些商业代理缓存会在 `Via` 首部附加一些额外信息，以描述缓存中发生的情况。

客户端有一种方法可以判断响应是否来自缓存，就是使用 `Date` 首部。将响应中 `Date` 首部的值与当前时间进行比较，如果响应中的日期值比较早，客户端通常就可

注 4：术语“命中比例”可能比“命中率”要好，因为“命中率”会让人错误地想到时间因素。但是“命中率”这个词很常用，所以这里我们也使用它。

注 5：有时，人们会在命中率中包括再验证命中，但有时候命中率和再验证命中率是分别测量的。在检测命中率的时候，要确定自己知道什么才是“命中”。

以认为这是一条缓存的响应。客户端也可以通过 Age 首部来检测缓存的响应，通过这个首部可以分辨出这条响应的使用期（参见附录 C 中的 Age 首部）。

7.6 缓存的拓扑结构

缓存可以是单个用户专用的，也可以是数千名用户共享的。专用缓存被称为*私有缓存*（private cache）。私有缓存是个人的缓存，包含了单个用户最常用的页面（参见图 7-7a）。共享的缓存被称为*公有缓存*（public cache）。公有缓存中包含了某个用户团体的常用页面（参见图 7-7b）。

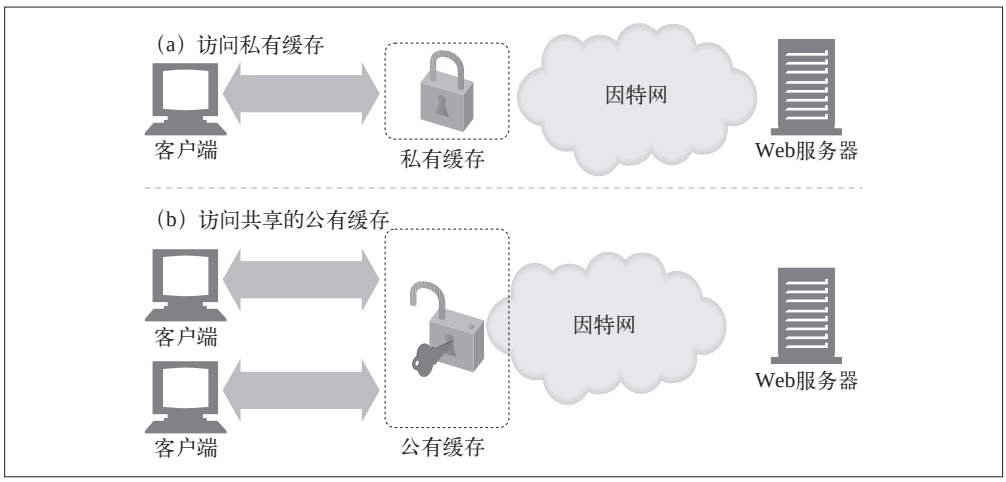


图 7-7 公有和私有缓存

7.6.1 私有缓存

私有缓存不需要很大的动力或存储空间，这样就可以将其做得很小，很便宜。Web 浏览器中有内建的私有缓存——大多数浏览器都会将常用文档缓存在你个人电脑的磁盘和内存中，并且允许用户去配置缓存的大小和各种设置。还可以去看看浏览器的缓存中有些什么内容。比如，对微软的 Internet Explorer 来说，可以从 Tools（工具）→ Internet Options...（因特网选项）对话框中获取缓存内容。MSIE 将缓存的文档称为“临时文件”，并将其与相关的 URL 和文档过期时间一起在文件列表中列出。通过特殊的 URL `about:cache` 可以查看网景的 Navigator 的缓存内容，这个 URL 会给出一个显示了缓存内容的“磁盘缓存统计”页面。

168

7.6.2 公有代理缓存

公有缓存是特殊的共享代理服务器，被称为*缓存代理服务器*（caching proxy server），

或者更常见地被称为*代理缓存*（proxy cache）（第 6 章讨论过代理）。代理缓存会从本地缓存中提供文档，或者代表用户与服务器进行联系。公有缓存会接受来自多个用户的访问，所以通过它可以更好地减少冗余流量。[6]

在图 7-8a 中，每个客户端都会重复地访问一个（还不在私有缓存中的）新的“热门”文档。每个私有缓存都要获取同一份文档，这样它就会多次穿过网络。而如图 7-8b 所示，使用共享的公有缓存时，对于这个流行的对象，缓存只要取一次就行了，它会用共享的副本为所有的请求服务，以降低网络流量。

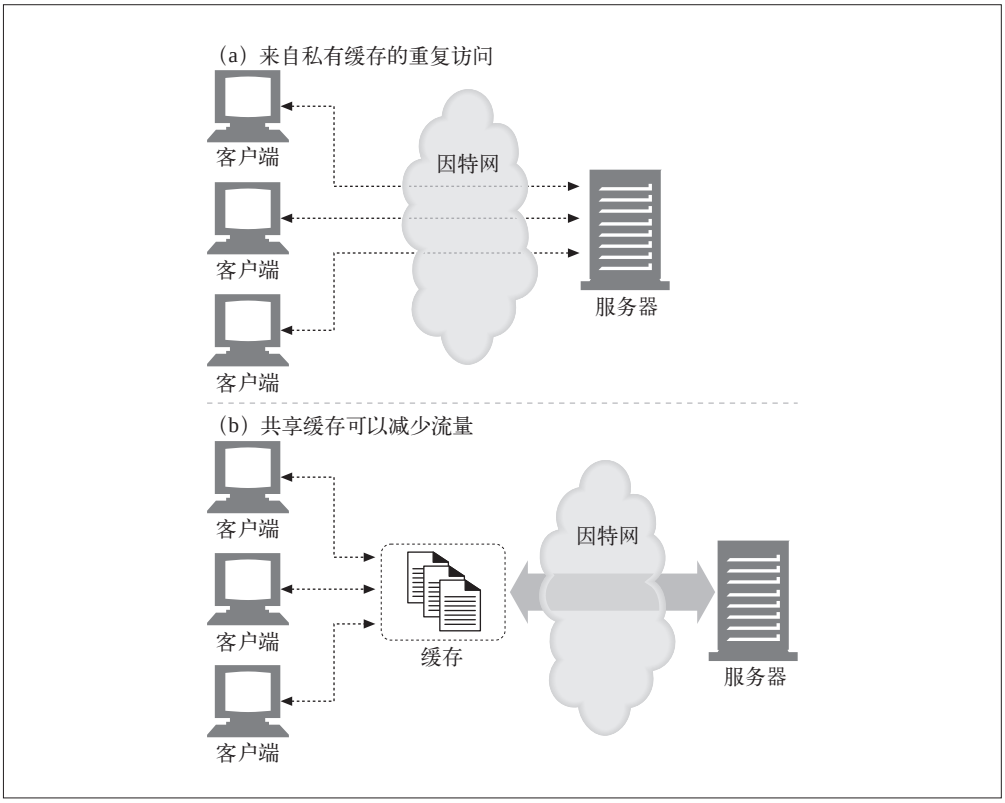


图 7-8　共享的公有缓存可以降低网络流量

代理缓存遵循第 6 章描述的代理规则。可以通过指定手工代理，或者通过代理自动配置文件，将你的浏览器配置为使用代理缓存（参见 6.4.1 节）。还可以通过使用拦截代理在不配置浏览器的情况下，强制 HTTP 请求经过缓存传输（参见第 20 章）。

注 6：公有缓存要缓存用户群体中各种不同的兴趣点，所以要足够大才能承载常用的文档集，而不会被单个用户所感兴趣的文档占满。

7.6.3 代理缓存的层次结构

在实际中，实现*层次化*（hierarchy）的缓存是很有意义的，在这种结构中，在较小缓存中未命中的请求会被导向较大的*父缓存*（parent cache），由它来为剩下的那些“提炼过的”流量提供服务。图 7-9 显示了一个两级的缓存层次结构。[7] 其基本思想是在靠近客户端的地方使用小型廉价缓存，而更高层次中，则逐步采用更大、功能更强的缓存来装载多用户共享的文档。[8]

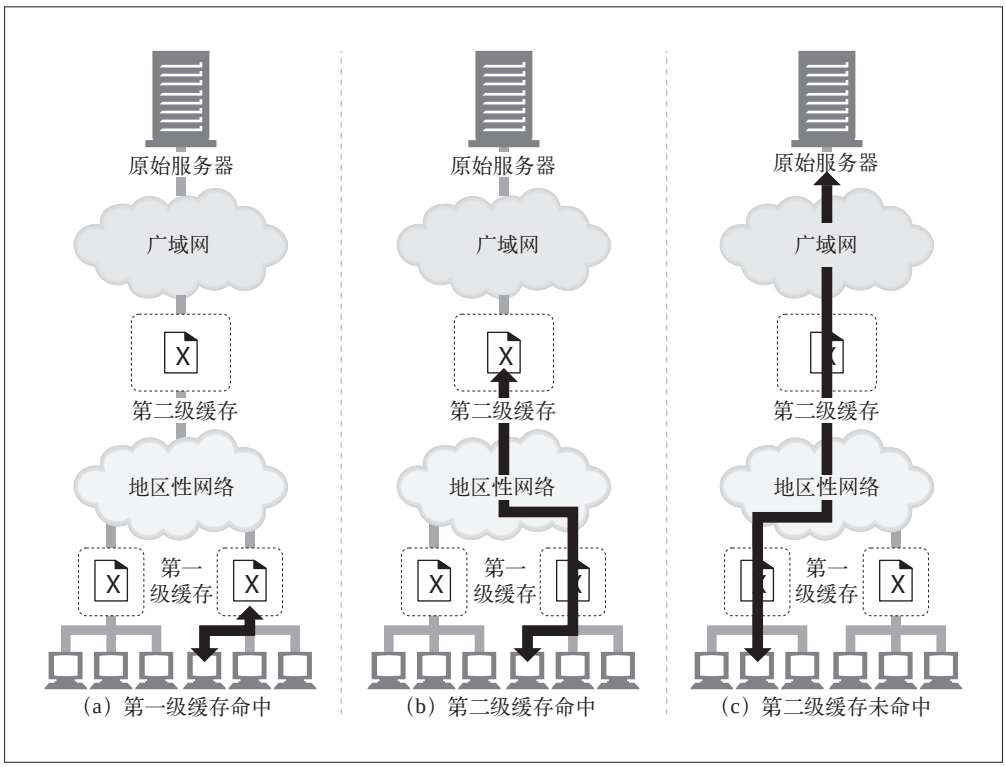


图 7-9 在两级的缓存层次结构中访问文档

我们希望大部分用户都能在附近的第一级缓存中命中（参见图 7-9a）。如果没有命中，较大的父缓存可能能够处理它们的请求（参见图 7-9b）。在缓存层次结构很深 169
的情况下，请求可能要穿过很长一溜缓存，但每个拦截代理都会添加一些性能损耗，当代理链路变得很长的时候，这种性能损耗会变得非常明显。[9]

注 7：如果客户端浏览器自带缓存，那么从技术上来讲，图 7-9 显示的就是一个三级的缓存层次结构。

注 8：父缓存可能要更大一些，以便装载在多用户间流行的文档，它们还要接收来自很多子缓存的聚合流量，这些子缓存的兴趣点可能很分散，所以还需要更高的性能。

注 9：在实际中，网络结构会尝试着将其深度限制在连续的两到三个代理以内。但是，新一代的高性能代理服务器会使代理链的长度变得不那么重要。

7.6.4 网状缓存、内容路由以及对等缓存

有些网络结构会构建复杂的*网状缓存*（cache mesh），而不是简单的缓存层次结构。网状缓存中的代理缓存之间会以更加复杂的方式进行对话，做出动态的缓存通信决策，决定与哪个父缓存进行对话，或者决定彻底绕开缓存，直接连接原始服务器。这种代理缓存会决定选择何种路由对内容进行访问、管理和传送，因此可将其称为*内容路由器*（content router）。

网状缓存中为内容路由设计的缓存（除了其他任务之外）要完成下列所有功能。

- 根据 URL 在父缓存或原始服务器之间进行动态选择。
- 根据 URL 动态地选择一个特定的父缓存。
- 前往父缓存之前，在本地缓存中搜索已缓存的副本。
- 允许其他缓存对其缓存的部分内容进行访问，但不允许因特网流量通过它们的缓存。

170

缓存之间这些更为复杂的关系允许不同的组织互为*对等*（peer）实体，将它们的缓存连接起来以实现共赢。提供可选的对等支持的缓存被称为*兄弟缓存*（sibling cache）（参见图 7-10）。HTTP 并不支持兄弟缓存，所以人们通过一些协议对 HTTP 进行了扩展，比如因特网缓存协议（Internet Cache Protocol，ICP）和超文本缓存协议（HyperText Caching Protocol，HTCP）。我们将在第 20 章讨论这些协议。

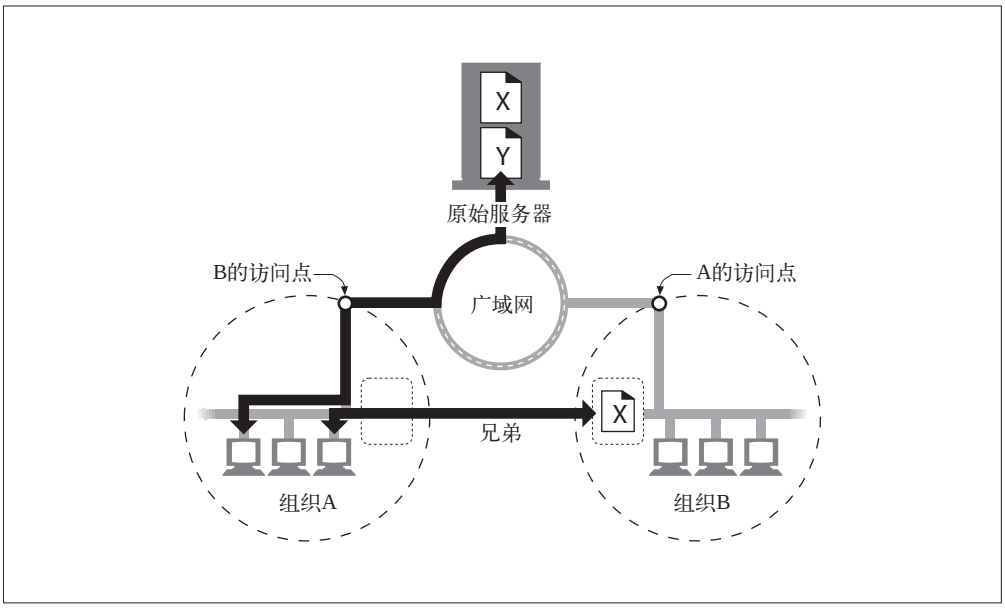


图 7-10　兄弟缓存

7.7 缓存的处理步骤

现代的商业化代理缓存相当地复杂。这些缓存构建得非常高效，可以支持 HTTP 和其他一些技术的各种高级特性。但除了一些微妙的细节之外，Web 缓存的基本工作原理大多很简单。对一条 HTTP GET 报文的基本缓存处理过程包括 7 个步骤（参见图 7-11）。

(1) 接收——缓存从网络中读取抵达的请求报文。

(2) 解析——缓存对报文进行解析，提取出 URL 和各种首部。

(3) 查询——缓存查看是否有本地副本可用，如果没有，就获取一份副本（并将其保存在本地）。

(4) 新鲜度检测——缓存查看已缓存副本是否足够新鲜，如果不是，就询问服务器是否有任何更新。

(5) 创建响应——缓存会用新的首部和已缓存的主体来构建一条响应报文。

(6) 发送——缓存通过网络将响应发回给客户端。

(7) 日志——缓存可选地创建一个日志文件条目来描述这个事务。

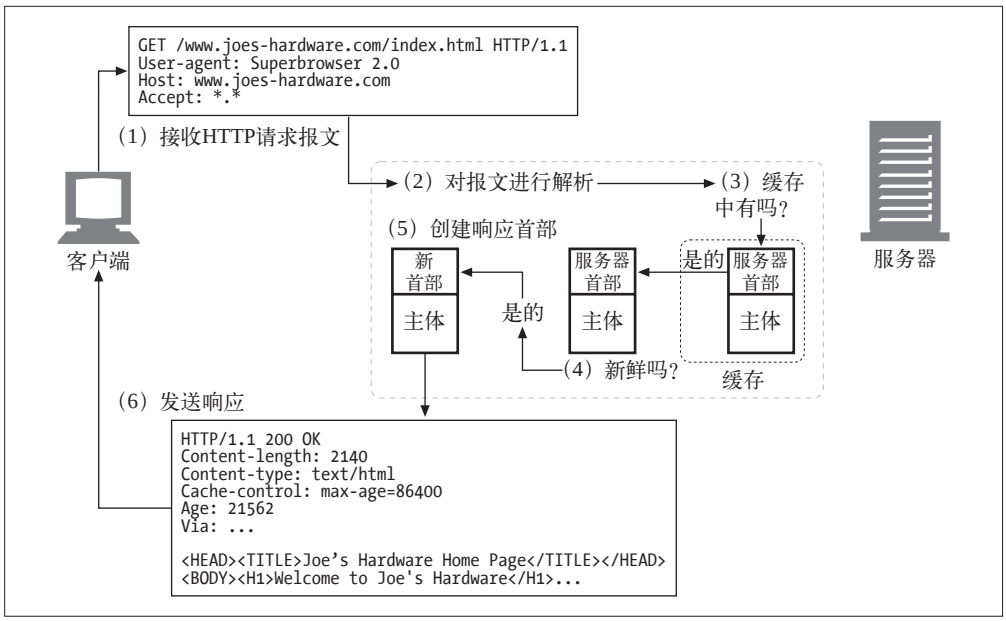


图 7-11 处理一个新鲜的缓存命中

7.7.1 第一步——接收

在第一步中，缓存检测到一条网络连接上的活动，读取输入数据。高性能的缓存会

同时从多条输入连接上读取数据，在整条报文抵达之前开始对事务进行处理。

7.7.2 第二步——解析

接下来，缓存将请求报文解析为片断，将首部的各个部分放入易于操作的数据结构
172 中。这样，缓存软件就更容易处理首部字段并修改它们了。[10]

7.7.3 第三步——查找

在第三步中，缓存获取了 URL，查找本地副本。本地副本可能存储在内存、本地磁盘，甚至附近的另一台计算机中。专业级的缓存会使用快速算法来确定本地缓存中是否有某个对象。如果本地没有这个文档，它可以根据情形和配置，到原始服务器或父代理中去取，或者返回一条错误信息。

已缓存对象中包含了服务器响应主体和原始服务器响应首部，这样就会在缓存命中时返回正确的服务器首部。已缓存对象中还包含了一些*元数据*（metadata），用来记录对象在缓存中停留了多长时间，以及它被用过多少次等。[11]

7.7.4 第四步——新鲜度检测

HTTP 通过缓存将服务器文档的副本保留一段时间。在这段时间里，都认为文档是
173 “新鲜的”，缓存可以在不联系服务器的情况下，直接提供该文档。但一旦已缓存副本停留的时间太长，超过了文档的*新鲜度限值*（freshness limit），就认为对象“过时”了，在提供该文档之前，缓存要再次与服务器进行确认，以查看文档是否发生了变化。客户端发送给缓存的所有请求首部自身都可以强制缓存进行再验证，或者完全避免验证，这使得事情变得更加复杂了。

HTTP 有一组非常复杂的新鲜度检测规则，缓存产品支持的大量配置选项，以及与非 HTTP 新鲜度标准进行互通的需要则使问题变得更加严重了。本章其余的大部分篇幅都用于解释新鲜度的计算问题。

7.7.5 第五步——创建响应

我们希望缓存的响应看起来就像来自原始服务器的一样，缓存将已缓存的服务器响

注 10：解析程序还要负责首部各部分的标准化，将大小写或可替换数据格式之类不太重要的区别都看作等效的。而且，某些请求报文中包含有完整的绝对 URL，而其他一些请求中包含的则是相对 URL 和 `Host` 首部，所以解析程序通常都要将这些细节隐藏起来（参见 2.3.1 节）。

注 11：复杂的缓存还会保留引发服务器响应的原始客户端响应首部的一份副本，以用于 HTTP/1.1 内容协商（参见第 17 章）。

应首部作为响应首部的起点。然后缓存对这些基础首部进行了修改和扩充。

缓存负责对这些首部进行改造，以便与客户端的要求相匹配。比如，服务器返回的可能是一条 HTTP/1.0 响应（甚至是 HTTP/0.9 响应），而客户端期待的是一条 HTTP/1.1 响应，在这种情况下，缓存必须对首部进行相应的转换。缓存还会向其中插入新鲜度信息（`Cache-Control`、`Age` 以及 `Expires` 首部），而且通常会包含一个 `Via` 首部来说明请求是由一个代理缓存提供的。

注意，缓存不应该调整 `Date` 首部。`Date` 首部表示的是原始服务器最初产生这个对象的日期。

7.7.6 第六步——发送

一旦响应首部准备好了，缓存就将响应回送给客户端。和所有代理服务器一样，代理缓存要管理与客户端之间的连接。高性能的缓存会尽力高效地发送数据，通常可以避免在本地缓存和网络 I/O 缓冲区之间进行文档内容的复制。

7.7.7 第七步——日志

大多数缓存都会保存日志文件以及与缓存的使用有关的一些统计数据。每个缓存事务结束之后，缓存都会更新缓存命中和未命中数目的统计数据（以及其他相关的度量值），并将条目插入一个用来显示请求类型、URL 和所发生事件的日志文件。

最常见的缓存日志格式为 Squid 日志格式和网景的可扩展通用日志格式，但很多缓存产品都允许用户创建自定义的日志文件。第 21 章探讨了日志文件格式。 174

7.7.8 缓存处理流程图

图 7-12 以简化形式显示了缓存是如何处理请求，以 GET 一个 URL 的。[12]

7.8 保持副本的新鲜

可能不是所有的已缓存副本都与服务器上的文档一致。毕竟，这些文档会随着时间发生变化。报告可能每个月都会变化。在线报纸每天都会发生变化。财经数据可能每过几秒钟就会发生变化。如果缓存提供的总是老的数据，就会变得毫无用处。已缓存数据要与服务器数据保持一致。

HTTP 有一些简单的机制可以在不要求服务器记住有哪些缓存拥有其文档副本的情

注 12：可以通过条件请求在一步里完成图 7-12 列出的资源再验证及获取（参见 7.8.4 节）。

况下，保持已缓存数据与服务器数据之间充分一致。HTTP 将这些简单的机制称为文档过期（document expiration）和服务器再验证（server revalidation）。

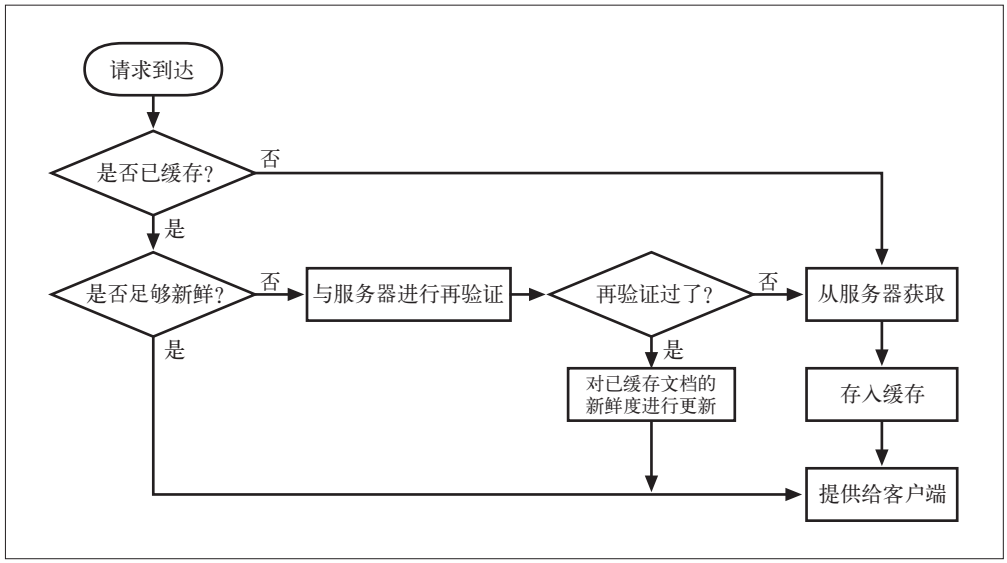


图 7-12　缓存 GET 请求的流程图

7.8.1　文档过期

通过特殊的 HTTP `Cache-Control` 首部和 `Expires` 首部，HTTP 让原始服务器向每个文档附加了一个“过期日期”（参见图 7-13）。就像一夸脱牛奶上的过期日期一样，这些首部说明了在多长时间内可以将这些内容视为新鲜的。

175

```
HTTP/1.0 200 OK
Date: Sat, 29 Jun 2002, 14:30:00 GMT
Content-type: text/plain
Content-length: 67
Expires: Fri, 05 Jul 2002, 05:00:00 GMT

Independence Day sale at Joe's Hardware
Come shop with us today!
```

(a) Expires首部

```
HTTP/1.0 200 OK
Date: Sat, 29 Jun 2002, 14:30:00 GMT
Content-type: text/plain
Content-length: 67
Cache-Control: max-age=484200

Independence Day sale at Joe's Hardware
Come shop with us today!
```

(b) Cache-Control: max-age首部

图 7-13　**`Expires` 首部和 `Cache-Control` 首部**

在缓存文档过期之前，缓存可以以任意频率使用这些副本，而无需与服务器联系——当然，除非客户端请求中包含有阻止提供已缓存或未验证资源的首部。但一旦已缓存文档过期，缓存就必须与服务器进行核对，询问文档是否被修改过，如果被修改过，就要获取一份新鲜（带有新的过期日期）的副本。

7.8.2 过期日期和使用期

服务器用 HTTP/1.0+ 的 Expires 首部或 HTTP/1.1 的 Cache-Control：max-age 响应首部来指定过期日期，同时还会带有响应主体。Expires 首部和 Cache-Control：max-age 首部所做的事情本质上是一样的，但由于 Cache-Control 首部使用的是相对时间而不是绝对日期，所以我们更倾向于使用比较新的 Cache-Control 首部。绝对日期依赖于计算机时钟的正确设置。表 7-2 列出了各种过期响应首部。

表7-2 过期响应首部

首　部	描　述
Cache-Control:max-age	max-age 值定义了文档的最大使用期——从第一次生成文档到文档不再新鲜、无法使用为止，最大的合法生存时间（以秒为单位） Cache-Control: max-age=484200
Expires	指定一个绝对的过期日期。如果过期日期已经过了，就说明文档不再新鲜了 Expires: Fri, 05 Jul 2002, 05:00:00 GMT

假设今天是美国东部标准时间（EST，Eastern Standard Time）2002 年 6 月 29 日上午 9：30，Joe 的五金商店正在准备进行 7 月 4 日（美国国庆日）特卖（只剩 5 天了）。Joe 想在他的 Web 服务器上放置一个特殊的 Web 页面，并将其设置为 2002 年 7 月 5 日晚上的 EST 午夜时间过期。如果 Joe 的服务器使用的是老式的 Expires 首部，服务器响应报文（参见图 7-13a）中可能就会包含这个首部：[13]

```
Expires: Fri, 05 Jul 2002, 05:00:00 GMT
```

176

如果 Joe 的服务器使用了较新的 Cache-Control：max-age 首部，服务器响应报文（参见图 7-13b）中可能就会包含这个首部：

```
Cache-Control: max-age=484200
```

如果这还不够明确的话，可以这样来看，当前时间，EST 时间 2002 年 6 月 29 日早上 9：30，到售卖结束时间 2002 年 7 月 5 日午夜之间有 484 200 秒。到售卖结束之前还有 134.5 小时（大约 5 天）。每小时有 3600 秒，这样到售卖结束之前还有 484 200 秒。

7.8.3 服务器再验证

仅仅是已缓存文档过期了并不意味着它和原始服务器上目前处于活跃状态的文档有

注 13：所有 HTTP 日期和时间都会在格林尼治标准时间（GMT）过期。GMT 是穿过英国格林尼治的本初子午线（经度为零）上的时间。GMT 比美国东部标准时间早五个小时，因此 EST 的午夜就是 05：00GMT。

实际的区别；这只是意味着到了要进行核对的时间了。这种情况被称为“服务器再验证”，说明缓存需要询问原始服务器文档是否发生了变化。

- 如果再验证显示内容发生了变化，缓存会获取一份新的文档副本，并将其存储在旧文档的位置上，然后将文档发送给客户端。
- 如果再验证显示内容没有发生变化，缓存只需要获取新的首部，包括一个新的过期日期，并对缓存中的首部进行更新就行了。

这是个很棒的系统。缓存并不一定要为每条请求验证文档的有效性——只有在文档过期时它才需要与服务器进行再验证。这样不会提供陈旧的内容，还可以节省服务器的流量，并拥有更好的用户响应时间。

HTTP 协议要求行为正确的缓存返回下列内容之一：

- “足够新鲜”的已缓存副本；
- 与服务器进行过再验证，确认其仍然新鲜的已缓存副本；
- 如果需要与之进行再验证的原始服务器出故障了，就返回一条错误报文[14]；
- 附有警告信息说明内容可能不正确的已缓存副本。

7.8.4 用条件方法进行再验证

HTTP 的条件方法可以高效地实现再验证。HTTP 允许缓存向原始服务器发送一个“条件 GET”，请求服务器只有在文档与缓存中现有的副本不同时，才回送对象主体。通过这种方式，将新鲜度检测和对象获取结合成了单个条件 GET。向 GET 请求报文中添加一些特殊的条件首部，就可以发起条件 GET。只有条件为真时，Web 服务器才会返回对象。

177

HTTP 定义了 5 个条件请求首部。对缓存再验证来说最有用的 2 个首部是 `If-Modified-Since` 和 `If-None-Match`。[15] 所有的条件首部都以前缀“If-”开头。表 7-3 列出了在缓存再验证中使用的条件请求首部。

注 14：如果原始服务器不可访问，但缓存需要进行再验证，那么缓存就必须返回一条错误或一条用来描述通信故障的警告报文。否则，来自已移除服务器上的页面未来可能会在网络的缓存中存留任意长的时间。

注 15：其他条件首部包括 `If-Unmodified-Since`（在进行部分文件的传输时，获取文件的其余部分之前要确保文件未发生变化，此时这个首部是非常有用的）、`If-Range`（支持对不完整文档的缓存）和 `If-Match`（用于与 Web 服务器打交道时的并发控制）。

表7-3 缓存再验证中使用的两个条件首部

首 部	描 述
If-Modified-Since:<date>	如果从指定日期之后文档被修改过了，就执行请求的方法。可以与 Last-Modified 服务器响应首部配合使用，只有在内容被修改后与已缓存版本有所不同的时候才去获取内容
If-None-Match:<tags>	服务器可以为文档提供特殊的标签（参见 ETag），而不是将其与最近修改日期相匹配，这些标签就像序列号一样。如果已缓存标签与服务器文档中的标签有所不同，If-None-Match 首部就会执行所请求的方法

7.8.5 If-Modified-Since:Date再验证

最常见的缓存再验证首部是 If-Modified-Since。If-Modified-Since 再验证请求通常被称为 IMS 请求。只有自某个日期之后资源发生了变化的时候，IMS 请求才会指示服务器执行请求：

- 如果自指定日期后，文档被修改了，If-Modified-Since 条件就为真，通常 GET 就会成功执行。携带新首部的新文档会被返回给缓存，新首部除了其他信息之外，还包含了一个新的过期日期。
- 如果自指定日期后，文档没被修改过，条件就为假，会向客户端返回一个小的 304 Not Modified 响应报文，为了提高有效性，不会返回文档的主体。[16] 这些首部是放在响应中返回的，但只会返回那些需要在源端更新的首部。比如，Content-Type 首部通常不会被修改，所以通常不需要发送。一般会发送一个新的过期日期。

If-Modified-Since 首部可以与 Last-Modified 服务器响应首部配合工作。原始服务器会将最后的修改日期附加到所提供的文档上去。当缓存要对已缓存文档进行再验证时，就会包含一个 If-Modified-Since 首部，其中携带有最后修改已缓存副本的日期：

178

```
If-Modified-Since: <cached last-modified date>
```

如果在此期间内容被修改了，最后的修改日期就会有所不同，原始服务器就会回送新的文档。否则，服务器会注意到缓存的最后修改日期与服务器文档当前的最后修改日期相符，会返回一个 304 Not Modified 响应。

注 16：如果有一个不认识 If-Modified-Since 首部的老服务器收到了条件请求，它会将其作为一个普通的 GET 解释。在这种情况下，系统仍然能够工作，但由于要对未修改的文档数据进行不必要的传输，所以效率会比较低。

例如，如图 7-14 所示，如果你的缓存在 7 月 3 日对 Joe 的五金商店的 7 月 4 日特卖声明进行再验证，就会收到一条 Not Modified 响应（参见图 7-14a）。但如果你的缓存在 7 月 5 日午夜售卖结束后对文档进行再验证，缓存就会收到一个新文档，因为服务器内容已经发生了变化（参见图 7-14b）。

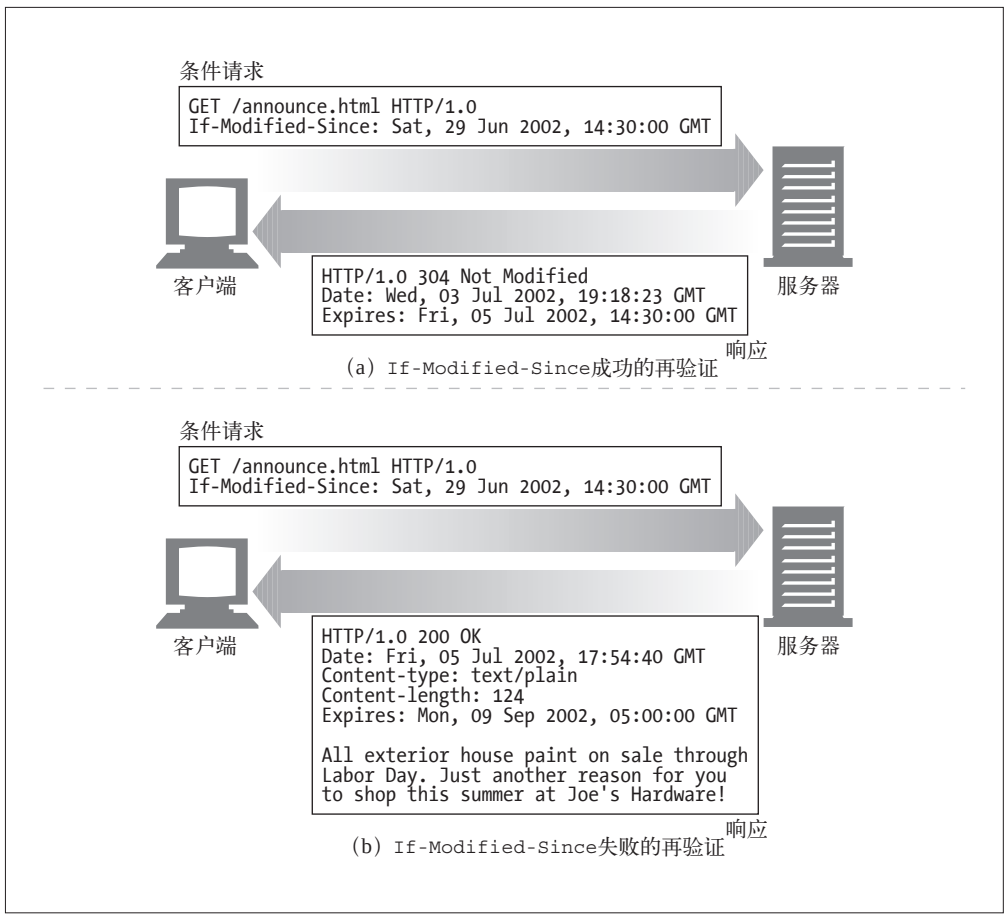


图 7-14　如果未发生变化，`If-Modified-Since` 再验证会返回 304 响应，如果发生了变化，就返回带有新主体的 200 响应

注意，有些 Web 服务器并没有将 `If-Modified-Since` 作为真正的日期来进行比对。相反，它们在 IMS 日期和最后修改日期之间进行了字符串匹配。这样得到的语义就是“如果最后的修改不是在这个确定的日期进行的”，而不是“如果在这个日期之后没有被修改过”。将最后修改日期作为某种序列号使用时，这种替代语义能够很好地识别出缓存是否过期，但这会妨碍客户端将 `If-Modified-Since` 首部用于真正基于时间的一些目的。

179

7.8.6 If-None-Match：实体标签再验证

有些情况下仅使用最后修改日期进行再验证是不够的。

- 有些文档可能会被周期性地重写（比如，从一个后台进程中写入），但实际包含的数据常常是一样的。尽管内容没有变化，但修改日期会发生变化。
- 有些文档可能被修改了，但所做修改并不重要，不需要让世界范围内的缓存都重装数据（比如对拼写或注释的修改）。
- 有些服务器无法准确地判定其页面的最后修改日期。
- 有些服务器提供的文档会在亚秒间隙发生变化（比如，实时监视器），对这些服务器来说，以一秒为粒度的修改日期可能就不够用了。

为了解决这些问题，HTTP 允许用户对被称为实体标签（ETag）的“版本标识符”进行比较。实体标签是附加到文档上的任意标签（引用字符串）。它们可能包含了文档的序列号或版本名，或者是文档内容的校验和及其他指纹信息。

当发布者对文档进行修改时，可以修改文档的实体标签来说明这个新的版本。这样，如果实体标签被修改了，缓存就可以用 If-None-Match 条件首部来 GET 文档的新副本了。

在图 7-15 中，缓存中有一个实体标签为 v2.6 的文档。它会与原始服务器进行再验证，如果标签 v2.6 不再匹配，就会请求一个新对象。在图 7-15 中，标签仍然与之匹配，因此会返回一条 304 Not Modified 响应。

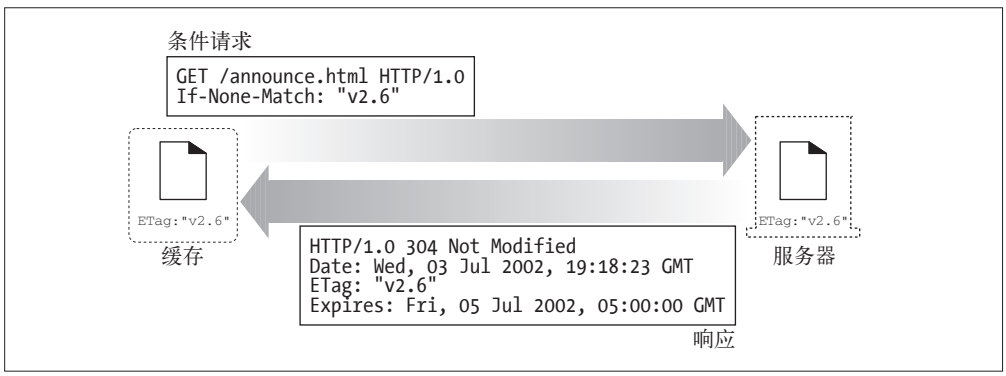


图 7-15　因为实体标签仍然匹配，If-None-Match 再验证成功

180

如果服务器上的实体标签已经发生了变化（可能变成了 v3.0），服务器会在一个 200 OK 响应中返回新的内容以及相应的新 Etag。

可以在 If-None-Match 首部包含几个实体标签，告诉服务器，缓存中已经存在带

有这些实体标签的对象副本：

```
If-None-Match: "v2.6"
If-None-Match: "v2.4","v2.5","v2.6"
If-None-Match: "foobar","A34FAC0095","Profiles in Courage"
```

7.8.7 强弱验证器

缓存可以用实体标签来判断，与服务器相比，已缓存版本是不是最新的（与使用最近修改日期的方式很像）。从这个角度来看，实体标签和最近修改日期都是*缓存验证器*（cache validator）。

有时，服务器希望在对文档进行一些非实质性或不重要的修改时，不要使所有的已缓存副本都失效。HTTP/1.1 支持“弱验证器”，如果只对内容进行了少量修改，就允许服务器声明那是“足够好”的等价体。

只要内容发生了变化，强验证器就会变化。弱验证器允许对一些内容进行修改，但内容的主要含义发生变化时，通常它还是会变化的。有些操作不能用弱验证器来实现（比如有条件地获取部分内容），所以，服务器会用前缀“W/”来标识弱验证器。

```
ETag: W/"v2.6"
If-None-Match: W/"v2.6"
```

不管相关的实体值以何种方式发生了变化，强实体标签都要发生变化。而相关实体在语义上发生了比较重要的变化时，弱实体标签也应该发生变化。

注意，原始服务器一定不能为两个不同的实体重用一个特定的强实体标签值，或者为两个语义不同的实体重用一个特定的弱实体标签值。缓存条目都可能会留存任意长的时间，与过期时间无关，有人可能希望当缓存验证条目时，绝对不会再次使用在过去某一时刻获得的验证器，这种愿望可能不太现实。

7.8.8 什么时候应该使用实体标签和最近修改日期

如果服务器回送了一个实体标签，HTTP/1.1 客户端就必须使用实体标签验证器。如果服务器只回送了一个`Last-Modified`值，客户端就可以使用`If-Modified-Since`验证。如果实体标签和最后修改日期都提供了，客户端就应该使用这两种再验证方
181 案，这样 HTTP/1.0 和 HTTP/1.1 缓存就都可以正确响应了。

除非 HTTP/1.1 原始服务器无法生成实体标签验证器，否则就应该发送一个出去，如果使用弱实体标签有优势的话，发送的可能就是个弱实体标签，而不是强实体标签。而且，最好同时发送一个最近修改值。

如果 HTTP/1.1 缓存或服务器收到的请求既带有 If-Modified-Since，又带有实体标签条件首部，那么只有这两个条件都满足时，才能返回 304 Not Modified 响应。

7.9 控制缓存的能力

服务器可以通过 HTTP 定义的几种方式来指定在文档过期之前可以将其缓存多长时间。按照优先级递减的顺序，服务器可以：

- 附加一个 Cache-Control: no-store 首部到响应中去；
- 附加一个 Cache-Control: no-cache 首部到响应中去；
- 附加一个 Cache-Control: must-revalidate 首部到响应中去；
- 附加一个 Cache-Control: max-age 首部到响应中去；
- 附加一个 Expires 日期首部到响应中去；
- 不附加过期信息，让缓存确定自己的过期日期。

本节描述了缓存控制首部。下一节，也就是 7.10 节介绍了如何为不同的内容分配不同的缓存信息。

7.9.1 no-Store与no-Cache响应首部

HTTP/1.1 提供了几种限制对象缓存，或限制提供已缓存对象的方式，以维持对象的新鲜度。no-store 首部和 no-cache 首部可以防止缓存提供未经证实的已缓存对象：

```
Pragma: no-cache
Cache-Control: no-store
Cache-Control: no-cache
```

标识为 no-store 的响应会禁止缓存对响应进行复制。缓存通常会像非缓存代理服务器一样，向客户端转发一条 no-store 响应，然后删除对象。

标识为 no-cache 的响应实际上是可以存储在本地缓存区中的。只是在与原始服务器进行新鲜度再验证之前，缓存不能将其提供给客户端使用。这个首部使用 do-not-serve-from-cache-without-revalidation 这个名字会更恰当一些。

HTTP/1.1 中提供 Pragma: no-cache 首部[17]是为了兼容于 HTTP/1.0+。除了与只理解 Pragma: no-cache 的 HTTP/1.0 应用程序进行交互时，HTTP 1.1 应用程序都应该使用 Cache-Control: no-cache。

182

注 17：从技术上来讲，Pragma:no-cache 首部只能用于 HTTP 请求，但在实际中它作为扩展首部已被广泛地用于 HTTP 请求和响应之中。

7.9.2 max-age响应首部

Cache-Control: max-age 首部表示的是从服务器将文档传来之时起，可以认为此文档处于新鲜状态的秒数。还有一个 s-maxage 首部（注意 maxage 的中间没有连字符），其行为与 max-age 类似，但仅适用于共享（公有）缓存：

```
Cache-Control: max-age=3600
Cache-Control: s-maxage=3600
```

服务器可以请求缓存不要缓存文档，或者将最大使用期设置为零，从而在每次访问的时候都进行刷新：

```
Cache-Control: max-age=0
Cache-Control: s-maxage=0
```

7.9.3 Expires响应首部

不推荐使用 Expires 首部，它指定的是实际的过期日期而不是秒数。HTTP 设计者后来认为，由于很多服务器的时钟都不同步，或者不正确，所以最好还是用剩余秒数，而不是绝对时间来表示过期时间。可以通过计算过期值和日期值之间的秒数差来计算类似的新鲜生存期：

```
Expires: Fri, 05 Jul 2002, 05:00:00 GMT
```

有些服务器还会回送一个 Expires:0 响应首部，试图将文档置于永远过期的状态，但这种语法是非法的，可能给某些软件带来问题。应该试着支持这种结构的输入，但不应该产生这种结构的输出。

7.9.4 must-revalidate响应首部

可以配置缓存，使其提供一些陈旧（过期）的对象，以提高性能。如果原始服务器希望缓存严格遵守过期信息，可以在原始响应中附加一个 Cache-Control: must-revalidate 首部。

```
Cache-Control: must-revalidate
```

Cache-Control: must-revalidate 响应首部告诉缓存，在事先没有跟原始服务器进行再验证的情况下，不能提供这个对象的陈旧副本。缓存仍然可以随意提供新鲜的副本。如果在缓存进行 must-revalidate 新鲜度检查时，原始服务器不可
183 用，缓存就必须返回一条 504 Gateway Timeout 错误。

7.9.5 试探性过期

如果响应中没有 Cache-Control：max-age 首部，也没有 Expires 首部，缓存可以计算出一个试探性最大使用期。可以使用任意算法，但如果得到的最大使用期大于 24 小时，就应该向响应首部添加一个 Heuristic Expiration Warning（试探性过期警告，警告 13）首部。据我们所知，很少有浏览器会为用户提供这种警告信息。

LM-Factor 算法是一种很常用的试探性过期算法，如果文档中包含了最后修改日期，就可以使用这种算法。LM-Factor 算法将最后修改日期作为依据，来估计文档有多么易变。算法的逻辑如下所示。

- 如果已缓存文档最后一次修改发生在很久以前，它可能会是一份稳定的文档，不太会突然发生变化，因此将其继续保存在缓存中会比较安全。
- 如果已缓存文挡最近被修改过，就说明它很可能会频繁地发生变化，因此在与服务器进行再验证之前，只应该将其缓存很短一段时间。

实际的 LM-Factor 算法会计算缓存与服务器对话的时间跟服务器声明文档最后被修改的时间之间的差值，取这个间隔时间的一部分，将其作为缓存中的新鲜度持续时间。下面是 LM-factor 算法的 Perl 伪代码：

```
$time_since_modify = max(0, $server_Date - $server_Last_Modified);
$server_freshness_limit = int($time_since_modify * $lm_factor);
```

图 7-16 以图形方式给出了 LM-factor 的新鲜周期。图中用交叉线画出的阴影表示的是将 LM-factor 设置为 0.2 计算出的新鲜周期。

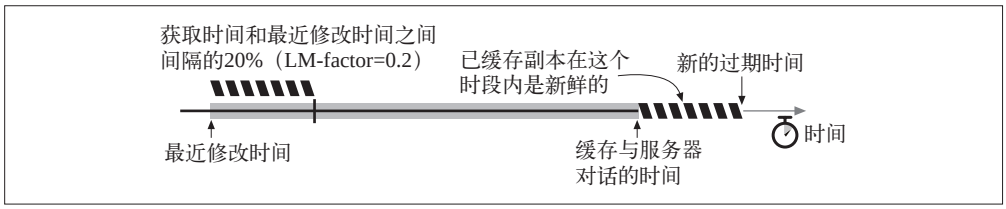


图 7-16　用 LM-factor 算法计算新鲜周期

通常人们会为试探性新鲜周期设置上限，这样它们就不会变得太大了。尽管比较保守的站点会将这个值设置为一天，但通常站点会将其设置为一周。

如果最后修改日期也没有的话，缓存就没什么信息可利用了。缓存通常会为没有任何新鲜周期线索的文档分配一个默认的新鲜周期（通常是一个小时或一天）。有时，比较保守的缓存会将这种试探性新鲜生存期设置为 0，强制缓存在每次将其提供给客户端之前，都去验证一下这些数据仍然是新鲜的。

与试探性新鲜计算有关的最后一点是——它们可能比你想象的要常见得多。很多原始服务器仍然不会产生 Expires 和 max-age 首部。选择缓存过期的默认时间时要特别小心！

7.9.6 客户端的新鲜度限制

Web 浏览器都有 Refresh（刷新）或 Reload（重载）按钮，可以强制对浏览器或代理缓存中可能过期的内容进行刷新。Refresh 按钮会发布一个附加了 Cache-Control 请求首部的 GET 请求，这个请求会强制进行再验证，或者无条件地从服务器获取文档。Refresh 的确切行为取决于特定的浏览器、文档以及拦截缓存的配置。

客户端可以用 Cache-Control 请求首部来强化或放松对过期时间的限制。有些应用程序对文档的新鲜度要求很高（比如人工刷新按钮），对这些应用程序来说，客户端可以用 Cache-Control 首部使过期时间更严格。另一方面，作为提高性能、可靠性或开支的一种折衷方式，客户端可能会放松新鲜度要求。表 7-4 对 Cache-Control 请求指令进行了总结。

表7-4 Cache-Control请求指令

指　令	目　的
Cache-Control: max-stale Cache-Control: max-stale = <s>	缓存可以随意提供过期的文件。如果指定了参数 <s>，在这段时间内，文档就不能过期。这条指令放松了缓存的规则
Cache-Control: min-fresh=<s>	至少在未来 <s> 秒内文档要保持新鲜。这就使缓存规则更加严格了
Cache-Control: max-age = <s>	缓存无法返回缓存时间长于 <s> 秒的文档。这条指令会使缓存规则更加严格，除非同时还发送了 max-stale 指令，在这种情况下，使用期可能会超过其过期时间
Cache-Control: no-cache Pragma: no-cache	除非资源进行了再验证，否则这个客户端不会接受已缓存的资源
Cache-Control: no-store	缓存应该尽快从存储器中删除文档的所有痕迹，因为其中可能会包含敏感信息
Cache-Control: only-if-cached	只有当缓存中有副本存在时，客户端才会获取一份副本

7.9.7 注意事项

文档过期系统并不是一个完美的系统。如果发布者不小心分配了一个很久之后的过期日期，在文档过期之前，她要对文档做的任何修改都不一定能显示在所有缓存中。[18]

185

注 18：文档过期采用了“生存时间”技术，这种技术用于很多因特网协议，比如 DNS 中。与 HTTP 一样，如果发布了一个很久之后才到时的过期日期，然后发现需要进行修改，DNS 就会遇到麻烦。但是，与 DNS 不同的是，HTTP 为客户端提供了一些覆盖和强制重载机制。

因此，很多发布者都不会使用很长的过期日期。而且，很多发布者甚至都不使用过期日期，这样缓存就很难确定文档会在多长时间内保持新鲜了。

7.10 设置缓存控制

不同的 Web 服务器为 HTTP `Cache-Control` 和 `Expiration` 首部的设置提供了一些不同的机制。本节简要介绍了流行的 Apache Web 服务器是怎样支持缓存控制的。具体细节请参见你的 Web 服务器文档。

7.10.1 控制Apache的HTTP首部

Apache Web 服务器提供了几种设置 HTTP 缓存控制首部的机制。其中很多机制在默认情况下都没有启动——你要启动它们（有些情况下先要获取 Apache 的扩展模块）。下面是对某些 Apache 特性的简要描述。

- mod_headers

  通过 mod_headers 模块可以对单独的首部进行设置。装载了这个模块，就可以用设置单个 HTTP 首部的指令来扩充 Apache 的配置文件了。还可以将这些设置与 Apache 的常用表达式以及过滤器结合在一起使用，将这些首部与个别内容关联起来。这里有一个配置实例，这个例子将某目录下所有的 HTML 文件都标识为非缓存的：

```
<Files *.html>
  Header set Cache-control no-cache
</Files>
```

- mod_expires

  mod_expires 模块提供的程序逻辑可以自动生成带有正确过期日期的 `Expires` 首部。通过这个模块，就可以将文档的过期日期设置为对其最后一次被访问之后或者其最近修改日期之后的某一时间段。通过这个模块可以为不同的文件类型设置不同的过期日期，还可以使用便捷的详尽描述信息来描述其缓存能力，比如“access plus 1 month（自访问之后起 1 个月）”。这里有几个例子：

```
ExpiresDefault A3600
ExpiresDefault M86400
ExpiresDefault "access plus 1 week"
ExpiresByType text/html "modification plus 2 days 6 hours 12 minutes"
```

- mod_cern_meta

  通过 mod_cern_meta 模块可以将一个包含 HTTP 首部的文件与特定的对象联系起来。启动这个模块时，就创建了一组“元文件”，每个需要控制的文档一个，而且还会为每个元文件添加所期望的首部。

7.10.2 通过HTTP-EQUIV控制HTML缓存

HTTP 服务器响应首部用于回送文档的到期信息以及缓存控制信息。Web 服务器与配置文件进行交互，为所提供的文档分配正确的 Cache-Control 首部。

为了让作者在无需与 Web 服务器的配置文件进行交互的情况下，能够更容易地为所提供的 HTML 文档分配 HTTP 首部信息，HTML 2.0 定义了 <META HTTP-EQUIV> 标签。这个可选的标签位于 HTML 文档的顶部，定义了应该与文档有所关联的 HTTP 首部。这里有一个 <META HTTP-EQUIV> 标签设置的例子，它将 HTML 文档标记为非缓冲的：

```
<HTML>
  <HEAD>
    <TITLE>My Document</TITLE>
    <META HTTP-EQUIV="Cache-control" CONTENT="no-cache">
  </HEAD>
      ...
```

最初，HTTP-EQUIV 标签是给 Web 服务器使用的。如 HTML RFC 1866 所述，Web 服务器应该为 HTML 解析 <META HTTP-EQUIV> 标签，并将规定的首部插入 HTTP 响应中：

> HTTP 服务器可以用此信息来处理文档。特别是，它可以在为请求此文档的报文所发送的响应中包含一个首部字段：首部名称是从 HTTP-EQUIV 属性值中获取的，首部值是从 CONTENT 属性值中获取的。

不幸的是，支持这个可选特性会增加服务器的额外负载，这些值也只是静态的，而且它只支持 HTML，不支持很多其他的文件类型，所以很少有 Web 服务器和代理支持此特性。

但是，有些浏览器确实会解析并在 HTML 内容中使用 HTTP-EQUIV 标签，像对待真的 HTTP 首部那样来处理嵌入式首部（参见图 7-17）。这样的效果并不好，因为支持 HTTP-EQUIV 标签的 HTML 浏览器使用的 Cache-control 规则可能会与拦截代理缓存所用的规则有所不同。这样会使缓存的过期处理行为发生混乱。

总之，<META HTTP-EQUIV> 标签并不是控制文档缓存特性的好方法。通过配置正确的服务器发出 HTTP 首部，是传送文档缓存控制请求的唯一可靠的方法。

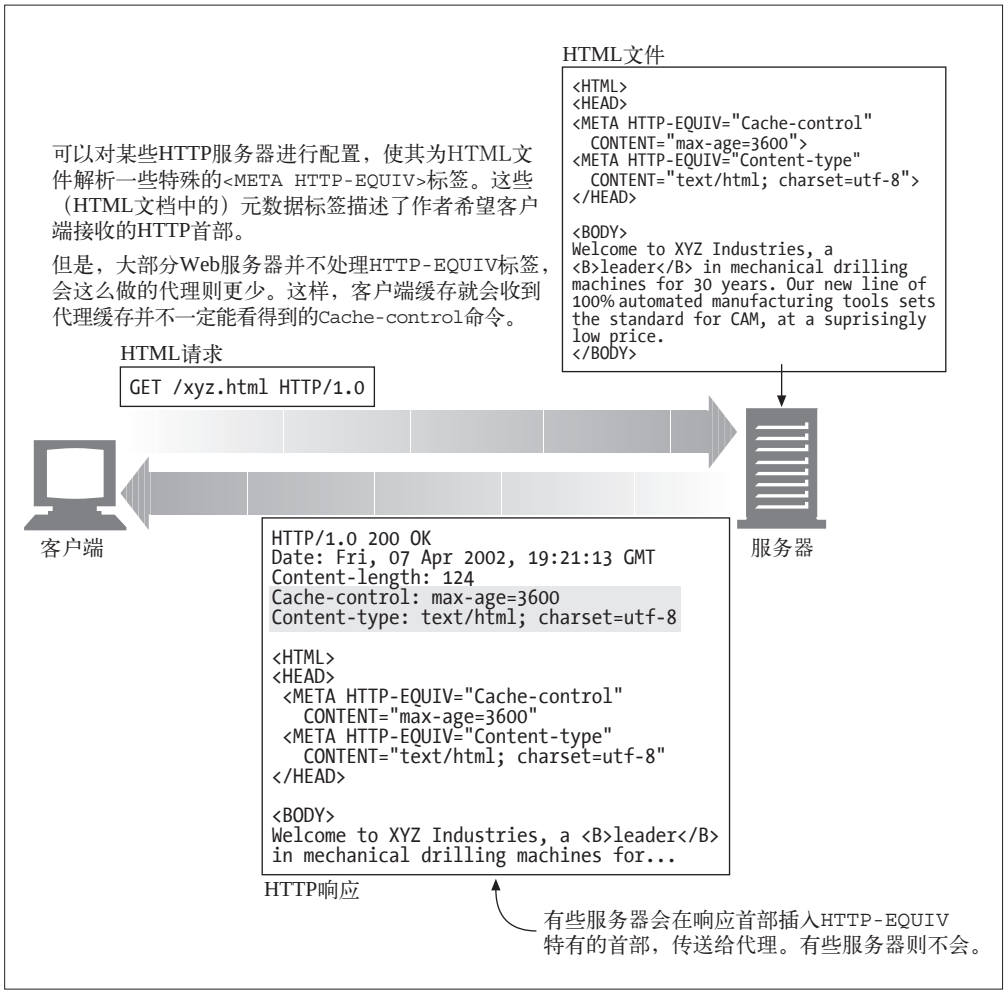


图 7-17　大多数软件都会忽略 **HTTP-EQUIV** 标签，所以这些标签可能会带来一些问题

7.11　详细算法

HTTP 规范提供了一个详细，但有点儿含糊不清而且经常会让人混淆的算法，来计算文档的使用期以及缓存的新鲜度。本节会对 HTTP 的新鲜度计算算法进行详细的讨论（参见图 7-12 中那个“足够新鲜？”菱形框），并对此算法的动机进行解释。 187

本节最适用于那些研究缓存内部机制的人。为了便于说明 HTTP 规范中的内容，我们使用了 Perl 伪代码。如果对计算缓存过期时间的公式中那些繁复的细节不感兴趣的话，可以跳过这一节。 188

7.11.1 使用期和新鲜生存期

为了分辨已缓存文档是否足够新鲜，缓存只需要计算两个值：已缓存副本的*使用期*（age），和已缓存副本的*新鲜生存期*（freshness lifetime）。如果已缓存副本的时长小于新鲜生存期，就说明副本足够新鲜，可以使用。用 Perl 表示为：

```
$is_fresh_enough = ($age < $freshness_lifetime);
```

文档的使用期就是自从服务器将其发送出来（或者最后一次被服务器再验证）之后“老去”的总时间。[19] 缓存可能不知道文档响应是来自上游缓存，还是来自服务器的，所以它不能假设文档是最新的。它必须根据显式的 Age 首部（优先），或者通过对服务器生成的 Date 首部的处理，来确定文档的使用期。

文档的新鲜生存期表明，已缓存副本在经过多长时间之后，就会因新鲜度不足而无法再向客户端提供了。新鲜生存期考虑了文档的过期日期，以及客户端可能请求的任何新鲜度覆盖范围。

有些客户端可能愿意接受稍微有些过期的文档（使用 Cache-Control: max-stale 首部）。有些客户端可能无法接受会在近期过期的文档（使用 Cache-Control: min-fresh 首部）。缓存将服务器过期信息与客户端的新鲜度要求结合在一起，以确定最大的新鲜生存期。

7.11.2 使用期的计算

响应的使用期就是服务器发布响应（或服务器对其进行了再验证）之后经过的总时间。使用期包含了响应在因特网路由器和网关中游荡的时间，在中间节点缓存中存储的时间，以及响应在你的缓存中停留的时间。例 7-1 给出了使用期计算的伪代码。

例 7-1　HTTP/1.1 使用期计算算法计算了已缓存文档的总体使用期

```
$apparent_age = max(0, $time_got_response - $Date_header_value);
$corrected_apparent_age = max($apparent_age, $Age_header_value);
$response_delay_estimate = ($time_got_response - $time_issued_request);
$age_when_document_arrived_at_our_cache =
    $corrected_apparent_age + $response_delay_estimate;
$how_long_copy_has_been_in_our_cache = $current_time - $time_got_response;

$age = $age_when_document_arrived_at_our_cache +
    $how_long_copy_has_been_in_our_cache;
```

注 19：记住，服务器上总是有所有文档的最新版本的。

HTTP 使用期计算的细节有点儿棘手，但其基本概念很简单。响应到达缓存时，缓存可以通过查看 Date 首部或 Age 首部来判断响应已使用的时间。缓存还能记录下文档在本地缓存中的停留时间。把这些值加在一起，就是响应的总使用期。HTTP 用一些魔法对时钟偏差和网络时延进行了补偿，但基本计算非常简单：


```
$age = $age_when_document_arrived_at_our_cache +
       $how_long_copy_has_been_in_our_cache;
```

缓存可以很方便地判断出已缓存副本已经在本地缓存了多长时间（这就是简单的簿记问题），但很难确定响应抵达缓存时的使用期，因为不是所有服务器的时钟都是同步的，而且我们也不知道响应到过哪里。完善的使用期计算算法会试着对此进行补偿。

1. 表面使用期是基于Date首部的

如果所有的计算机都共享同样的、完全精确的时钟，已缓存文档的使用期就可以是文档的“表面使用期”——当前时间减去服务器发送文档的时间。服务器发送时间就是 Date 首部的值。最简单的起始时间计算可以直接使用表面时间：

```
$apparent_age = $time_got_response - $Date_header_value;
$age_when_document_arrived_at_our_cache = $apparent_age;
```

但并不是所有的时钟都实现了良好的同步。客户端和服务器时钟之间可能有数分钟的差别，如果时钟没有设置好的话，甚至会有数小时或数天的区别。[20]

Web 应用程序，尤其是缓存代理，要做好与时间值有很大差异的服务器进行交互的准备。这种问题被称为*时钟偏差*（clock skew）——两台计算机时钟设置的不同。由于时钟偏差的存在，表面使用期有时会不太准确，而且有时会是负的。

如果使用期是负的，就将其设置为零。我们还可以对表面使用期进行完整性检查，以确定它没有大得令人不可思议，不过，实际上，表面使用期可能并没错。我们可能在与一个将文档缓存了很久的父缓存对话（缓存可能还存储了原始的 Date 首部）：

```
$apparent_age = max(0, $time_got_response - $Date_header_value);
$age_when_document_arrived_at_our_cache = $apparent_age;
```

要明确 Date 首部描述的是原始服务器的日期。代理和缓存一定不能修改这个日期！

注 20：HTTP 规范建议客户端、服务器和代理使用 NTP 这样的时间同步协议来强制使用统一的时间基准。

2. 逐跳使用期的计算

这样就可以去除时钟偏差造成的负数使用期了，但对时钟偏差给精确性带来的整体偏差，我们能做的工作很少。文档经过代理和缓存时，HTTP/1.1 会让每台设备都将相对使用期累加到 Age 首部中去，以此来解决缺乏通用同步时钟的问题。这种方式并不需要进行跨服务器的、端到端的时钟对比。

文档经过代理时，Age 首部值会随之增加。使用 HTTP/1.1 的应用程序应该在 Age
190 首部值中加上文档在每个应用程序和网络传输过程中停留的时间。每个中间应用程序都可以很容易地用本地时钟计算出文档的停留时间。

但响应链中所有的非 HTTP/1.1 设备都无法识别 Age 首部，它们会将首部未经修改地转发出去，或者将其删除掉。因此，在 HTTP/1.1 得到普遍应用之前，Age 首部都将是低估了的相对使用期。

除了基于 Date 计算出来的 Age 之外，还使用了相对 Age 值，而且不论是跨服务器的 Date 值，还是计算出来的 Age 值都可能被低估，所以会选择使用估计出的两个 Age 值中最保守的那个（最保守的值就是最老的 Age 值）。使用这种方式，HTTP 就能容忍 Age 首部存在的错误，尽管这样可能会搞错究竟哪边更新鲜：

```
$apparent_age = max(0, $time_got_response - $Date_header_value);
$corrected_apparent_age = max($apparent_age, $Age_header_value);
$age_when_document_arrived_at_our_cache = $corrected_apparent_age;
```

3. 对网络时延的补偿

事务处理可能会很慢。这是使用缓存的主要动因。但对速度非常慢的网络，或者那些过载的服务器来说，如果文档在网络或服务器中阻塞了很长时间，相对使用期的计算可能会极大地低估文档的使用期。

Date 首部说明了文档是在什么时候离开原始服务器的，[21] 但并没有说明文档在到缓存的传输过程中花费了多长时间。如果文档的传输经过了一长串的代理和父缓存，网络时延可能会相当大。[22]

没有什么简便的方法可以用来测量从服务器到缓存的单向网络时延，但往返时延则比较容易测量。缓存知道它请求文档的时间，以及文档抵达的时间。HTTP/1.1 会在这些网络时延上加上整个往返时延，以便对其进行保守地校正。这个从缓存到服务

注 21：注意，如果文档来自一个父缓存，而不是原始服务器，Date 首部反映的仍是原始服务器，而不是父缓存上的日期。

注 22：实际上，这个时延不会高于几十分之一秒（不然用户就会放弃），但即使是对生存期很短的对象来说，HTTP 的设计者也希望使用尽可能精确的过期时间。

器再到缓存的时延高估了从服务器到缓存的时延，但它是保守的。如果出错了，它只会使文档看起来比实际使用期要老，并引发不必要的再验证。计算是这样进行的：

```
$apparent_age = max(0, $time_got_response - $Date_header_value);
$corrected_apparent_age = max($apparent_age, $Age_header_value);
$response_delay_estimate = ($time_got_response - $time_issued_request);
$age_when_document_arrived_at_our_cache =
    $corrected_apparent_age + $response_delay_estimate;
```

7.11.3 完整的使用期计算算法

上一节说明了当 HTTP 所承载的文档抵达缓存时，如何计算其使用期。只要将这条响应存储到缓存中去，它就会进一步老化。当对缓存中文档的请求到达时，我们需要知道文档在缓存中停留了多长的时间，这样才能计算文档现在的使用期：

```
$age = $age_when_document_arrived_at_our_cache +
    $how_long_copy_has_been_in_our_cache;
```

嗒嗒！这样就有了例 7-1 中给出的完整的 HTTP/1.1 使用期计算算法。这就是简单的簿记问题了——我们知道了文档是什么时候到达缓存的（`$time_got_reponse`），也知道当前请求是什么时候到达的（刚才），这样停留时间就是两者之差了。所有这些都以图形方式显示在图 7-18 中了。

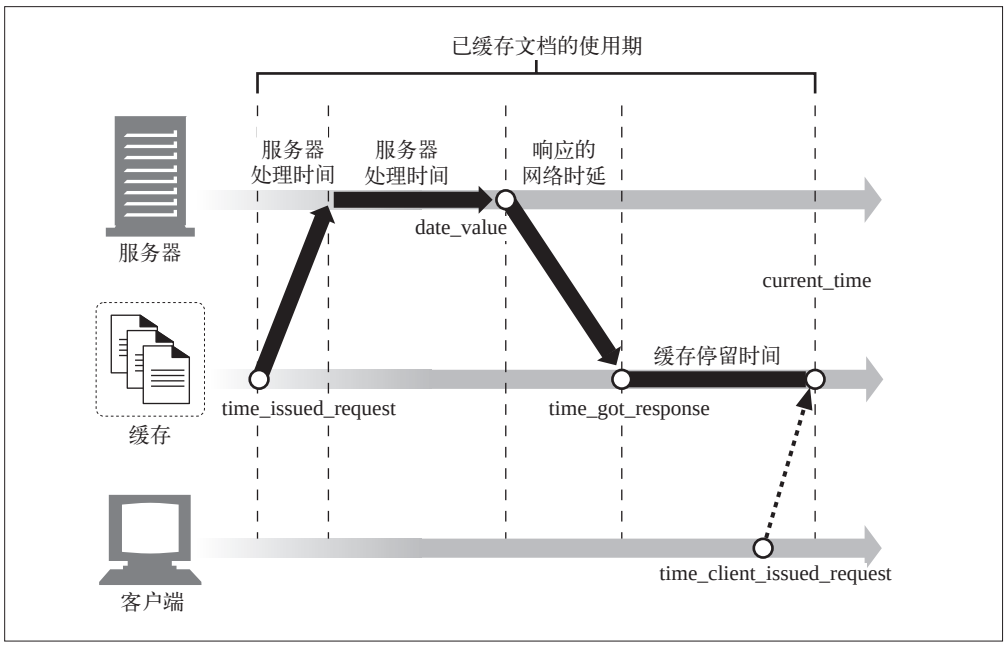


图 7-18　已缓存文档的使用期包括在网络和缓存中停留的时间

7.11.4 新鲜生存期计算

回想一下，我们是在想办法弄清楚已缓存文档是否足够新鲜，是否可以提供给客户端。要回答这个问题，就必须确定已缓存文档的使用期，并根据服务器和客户端限制来计算新鲜生存期。我们刚刚解释了如何计算使用期；现在我们来看看新鲜生存期的计算。

文档的新鲜生存期说明了在文档不再新鲜，无法提供给某个特定的客户端之前能够停
192 留多久。新鲜生存期取决于服务器和客户端的限制。服务器上可能有一些与文档的出版变化率有关的信息。那些非常稳定的已归档报告可能会在数年内保持新鲜。期刊可能只在下一期的出版物出来之前的剩余时间内有效——下一周，或是明早 6 点。

客户端可能有些其他指标。如果稍微有些过期的内容速度更快的话，它们可能也愿意接受，或者它们可能希望接收最新的内容。缓存是为用户服务的。必须要满足他们的要求。

7.11.5 完整的服务器——新鲜度算法

例 7-2 给出了一个用于计算服务器新鲜度限制的 Perl 算法。它会返回文档仍由服务器提供时所能到达的最大使用期。

例 7-2 服务器新鲜度限制的计算

```
sub server_freshness_limit
{
    local($heuristic,$server_freshness_limit,$time_since_last_modify);

    $heuristic = 0;

    if ($Max_Age_value_set)
    {
        $server_freshness_limit = $Max_Age_value;
    }
    elseif ($Expires_value_set)
    {
        $server_freshness_limit = $Expires_value - $Date_value;
    }
    elseif ($Last_Modified_value_set)
    {
        $time_since_last_modify = max(0, $Date_value -
             $Last_Modified_value);
        $server_freshness_limit = int($time_since_last_modify *
             $lm_factor);
        $heuristic = 1;
    }
    else
    {
```

```
        $server_freshness_limit = $default_cache_min_lifetime;
        $heuristic = 1;
    }

    if ($heuristic)
    {
        if ($server_freshness_limit > $default_cache_max_lifetime)
        { $server_freshness_limit = $default_cache_max_lifetime; }
        if ($server_freshness_limit < $default_cache_min_lifetime)
        { $server_freshness_limit = $default_cache_min_lifetime; }
    }

    return($server_freshness_limit);
}
```


现在，我们来看看客户端怎样修正服务器为文档指定的使用期限制。例 7-3 显示了一个 Perl 算法，此算法获取了服务器的新鲜度限制并根据客户端的限制对其进行修改。它会返回一个最大使用期，这是在无需再次验证，仍由缓存提供文档的前提下，文档的最大生存时间。

例 7-3　客户端新鲜度限制的计算

```
sub client_modified_freshness_limit
{
    $age_limit = server_freshness_limit( ); ## From Example 7-2

    if ($Max_Stale_value_set)
    {
        if ($Max_Stale_value == $INT_MAX)
        { $age_limit = $INT_MAX; }
        else
        { $age_limit = server_freshness_limit( ) + $Max_Stale_value; }
    }

    if ($Min_Fresh_value_set)
    {
    $age_limit = min($age_limit, server_freshness_limit( ) -
           $Min_Fresh_value_set);
    }

    if ($Max_Age_value_set)
    {
        $age_limit = min($age_limit, $Max_Age_value);
    }
}
```

整个进程中包含两个变量：文档的使用期及其新鲜度限制。如果使用期小于新鲜度限制，就说明文档“足够新鲜”。例 7-3 中的算法只是考虑了服务器的新鲜度限制，并根据附加的客户端限制对其进行了调整。希望通过本节的介绍能使在 HTTP 规范中描述的比较微妙的过期算法更清晰一些。

7.12 缓存和广告

读到这里，你一定已经意识到缓存可以提高性能并减少流量。知道缓存可以帮助用户，并为用户提供更好的使用体验，而且缓存也可以帮助网络运营商减少流量。

7.12.1 发布广告者的两难处境

你可能认为内容提供商会喜欢缓存。毕竟，如果到处都是缓存的话，内容提供商就不需要购买大型的多处理器 Web 服务器来满足用户需求了——他们不需要付过高的
194 网络服务费，一遍一遍地向用户发送同样的数据。更好的一点是，缓存可以将那些漂亮的文章和广告以更快，甚至更好看的方式显示在用户的显示器上，鼓励他们去浏览更多的内容，看更多的广告。这就是内容提供商所希望的！吸引更多的眼球和更多的广告！

但这就是困难所在。很多内容提供商的收益都是通过广告实现的——具体来说，每向用户显示一次广告内容，内容提供商就会得到相应的收益。（可能还不到一两便士，但如果一天显示数百万条广告的话，这些钱就会叠加起来！）这就是缓存的问题——它们会向原始服务器隐藏实际的访问次数。如果缓存工作得很好，原始服务器可能根本收不到任何 HTTP 访问，因为这些访问都被因特网缓存吸收了。但如果你的收益是基于访问次数的话，你就高兴不起来了。

7.12.2 发布者的响应

现在，广告商会使用各种类型的“缓存清除”技术来确保缓存不会窃取他们的命中流量。他们会在内容上加上 `no-cache` 首部。他们会通过 CGI 网关提供广告。还会在每次访问时重写广告 URL。

这些缓存清除技术并不仅用于代理缓存。实际上，现在主要将其用于每个 Web 浏览器中都启用了的缓存。但是，如果某些内容提供商维护其命中率的行为太过火了，就会降低缓存为其站点带来的积极作用。

理想情况下，内容提供商会让缓存吸收其流量，而缓存会告诉内容提供商它们拦截了多少次命中。现在，缓存有好几种方式可以做到这一点。

一种解决方案就是配置缓存，每次访问时都与原始服务器进行再验证。这样，每次访问时都会将命中推向原始服务器，但通常不会传送任何主体数据。当然，这样会降低事务处理的速度。[23]

注 23：有些缓存支持这种再验证的变体形式，在这种方式中，它们可以在后台发起条件 GET 或 HEAD 请求。用户不会感觉到时延，但这个请求会触发对原始服务器的离线访问。这是一种改进方式，但这种方式加重了缓存的负荷，极大地增加了流经网络的流量。

7.12.3 日志迁移

理想的解决方案是不需要将命中传递给服务器的。毕竟，缓存就可以记录下所有的命中。缓存只要将命中日志发送给服务器就行了。实际上，为了保持内容提供商们的满意度，有些大型缓存的提供商已经在对缓存日志进行人工处理，并将其传送给受影响的内容提供商了。

但是，命中日志很大，很难移动。而缓存日志并没有被标准化或被组织成独立的日志，以传送给单独的内容提供商。而且，这里面还存在着认证和隐私问题。

已经有一些高效（和不那么高效的）日志分发策略的建议了。但还没有哪个建议成熟到足以为 Web 软件厂商采用。很多建议都非常复杂，需要联合商业伙伴才能实现。[24] 有几家联合厂商已经开始开发广告收入改造工程的支撑框架了。

7.12.4 命中计数和使用限制

RFC 2227，“HTTP 的简单命中计数和使用限制”中定义了一种简单得多的方案。这个协议向 HTTP 中添加了一个称为 `Meter` 的首部，这个首部会周期性地将对特定 URL 的命中次数回送给服务器。通过这种方式，服务器可以从缓存周期性地获取对已缓存文档命中次数的更新。

而且，服务器还能控制在缓存必须向服务器汇报之前，其中的文档还可以使用多少次，或者为缓存文档设置一个时钟超时值。这种控制方式被称为使用限制；通过这种方式，服务器可以对缓存向原始服务器汇报之前，已缓存资源的使用次数进行控制。

我们将在第 21 章详细介绍 RFC 2227。

7.13 更多信息

更多有关缓存的信息，请参见以下参考资源。

- http://www.w3.org/Protocols/rfc2616/rfc2616.txt
  RFC 2616，由 R. Fielding、J. Gettys、J. Mogul、H. Frystyk、L. Mastinter、P. Leach 和 T. Berners-Lee 编写的“Hypertext Transfer Protocol”。
- *Web Caching*（《Web 缓存》）
  Duane Wessels 编写，O’Reilly & Associates 公司出版。

注 24：已经启动了几个商业项目，在尝试开发综合了缓存和日志功能的全球性解决方案。

- http://www.ietf.org/rfc/rfc3040.txt
  RFC 3040，“Internet Web Replication and Caching Taxonomy”（“因特网 Web 复制与缓存分类法”）。

- *Web Proxy Servers*（《Web 代理服务器》）
  Ari Luotonen 编写，Prentice Hall 计算机图书。

- http://www.ietf.org/rfc/rfc3143.txt
  RFC 3143，“Known HTTP Proxy/Caching Problems”（“已知的 HTTP 代理 / 缓存问题”）。

- http://www.squid-cache.org
  Squid Web 代理缓存。

196

第8章

集成点：网关、隧道及中继

事实证明，Web 是一种强大的内容发布工具。随着时间的流逝，人们已经从只在网上发送静态的在线文档，发展到共享更复杂的资源，比如数据库内容或动态生成的 HTML 页面。Web 浏览器这样的 HTTP 应用程序为用户提供了一种统一的方式来访问因特网上的内容。

HTTP 也已成为应用程序开发者的一种基本构造模块，开发者们可以在 HTTP 上捎回其他的协议内容（比如，可以将其他协议的流量包裹在 HTTP 中，用 HTTP 通过隧道或中继方式将这些流量传过公司的防火墙）。Web 上所有的资源都可以使用 HTTP 协议，而且其他应用程序和应用程序协议也可以利用 HTTP 来完成它们的任务。

本章简要介绍了一些开发者用 HTTP 访问不同资源的方法，展示了开发者如何将 HTTP 作为框架启动其他协议和应用程序通信。

本章会讨论：

- 在 HTTP 和其他协议及应用程序之间起到接口作用的网关；
- 允许不同类型的 Web 应用程序互相通信的应用程序接口；
- 允许用户在 HTTP 连接上发送非 HTTP 流量的隧道；
- 作为一种简化的 HTTP 代理，一次将数据转发一跳的中继。

8.1 网关

HTTP 扩展和接口的发展是由用户需求驱动的。要在 Web 上发布更复杂资源的需求
197 出现时，人们很快就明确了一点：单个应用程序无法处理所有这些能想到的资源。

为了解决这个问题，开发者提出了*网关*（gateway）的概念，网关可以作为某种翻译器使用，它抽象出了一种能够到达资源的方法。网关是资源和应用程序之间的粘合剂。应用程序可以（通过 HTTP 或其他已定义的接口）请求网关来处理某条请求，网关可以提供一条响应。网关可以向数据库发送查询语句，或者生成动态的内容，就像一个门一样：进去一条请求，出来一个响应。

图 8-1 显示的是一种资源网关。在这里，Joe 的五金商店服务器就是作为连接数据库内容的网关使用的——注意，客户端只是在通过 HTTP 请求资源，而 Joe 的五金商店的服务器在与网关进行交互以获取资源。

有些网关会自动将 HTTP 流量转换为其他协议，这样 HTTP 客户端无需了解其他协议，就可以与其他应用程序进行交互了（参见图 8-2）。

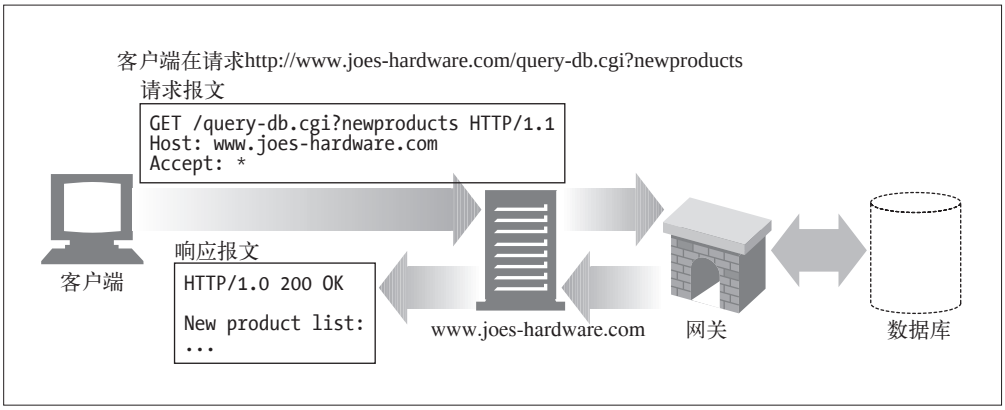


图 8-1　网关的魔力

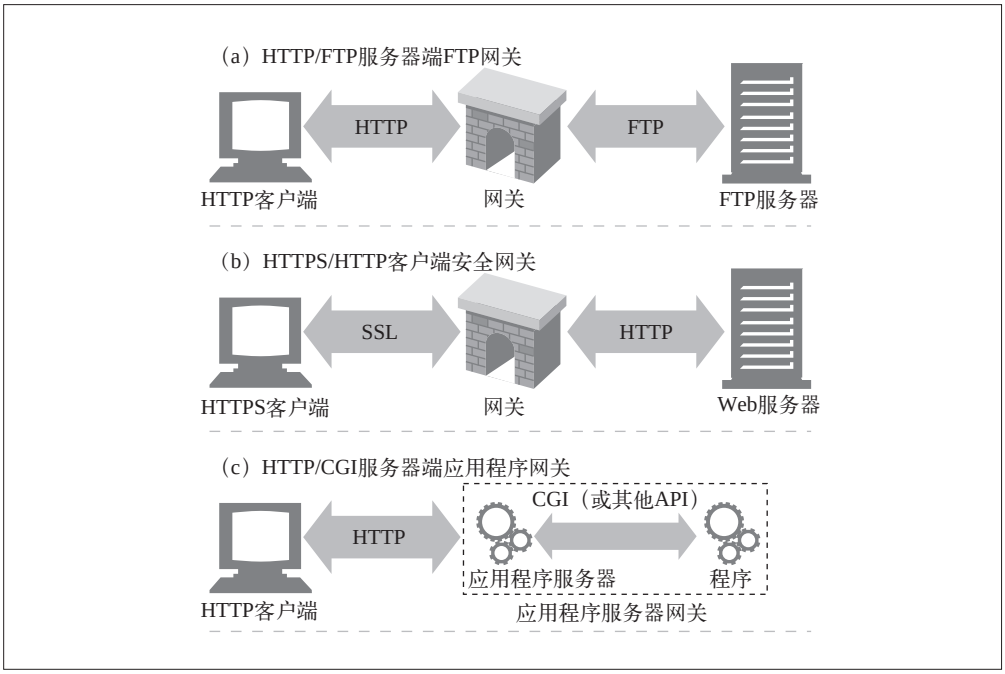


图 8-2　三个 Web 网关实例

图 8-2 显示了三个网关的示例。

- 在图 8-2a 中，网关收到了对 FTP URL 的 HTTP 请求。然后网关打开 FTP 连接，并向 FTP 服务器发布适当的命令。然后将文档和正确的 HTTP 首部通过 HTTP 回送。
- 在图 8-2b 中，网关通过 SSL 收到了一条加密的 Web 请求，网关会对请求进行解

密，[1] 然后向目标服务器转发一条普通的 HTTP 请求。可以将这些安全加速器直
198 接放在（通常处于同一场所的）Web 服务器前面，以便为原始服务器提供高性能的加密机制。

- 在图 8-2c 中，网关通过应用程序服务器网关 API，将 HTTP 客户端连接到服务器端的应用程序上去。在网上的电子商店购物、查看天气预报，或者获取股票报价时，访问的就是应用程序服务器网关。

客户端和服务器端网关

Web 网关在一侧使用 HTTP 协议，在另一侧使用另一种协议。[2]

可以用一个斜杠来分隔客户端和服务器端协议，并以此对网关进行描述：

< 客户端协议 >/< 服务器端协议 >

因此，将 HTTP 客户端连接到 NNTP 新闻服务器的网关就是一个 HTTP/NNTP 网关。我们用术语*服务器端网关*和*客户端网关*来说明对话是在网关的哪一侧进行的。

- *服务器端网关*（server-side gateway）通过 HTTP 与客户端对话，通过其他协议与服务器通信（HTTP/*）。
- *客户端网关*（client-side gateway）通过其他协议与客户端对话，通过 HTTP 与服
199 务器通信（*/HTTP）。

8.2 协议网关

将 HTTP 流量导向网关时所使用的方式与将流量导向代理的方式相同。最常见的方式是，显式地配置浏览器使用网关，对流量进行透明的拦截，或者将网关配置为替代者（反向代理）。

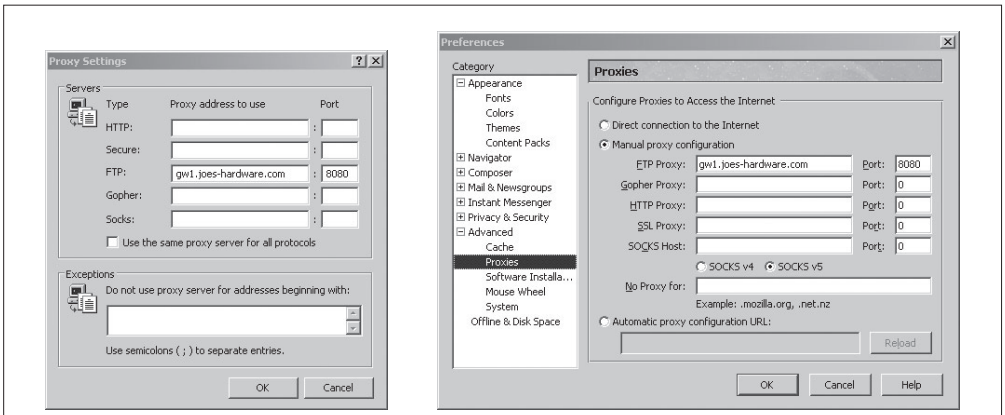
图 8-3 显示了配置浏览器使用服务器端 FTP 网关的对话框。在图中显示的配置中，配置浏览器将 gw1.joes-hardware.com 作为所有 FTP URL 的 HTTP/FTP 网关。浏览器没有将 FTP 命令发送给 FTP 服务器，而是将 HTTP 命令发送给端口 8080 上的 HTTP/FTP 网关 gw1.joes-hardware.com。

图 8-4 给出了这种网关配置的结果。一般的 HTTP 流量不受影响，会继续流入原始服务器。但对 FTP URL 的请求则被放在 HTTP 请求中发送给网关 gw1.joes-

注 1：网关上要安装适当的服务器证书。

注 2：在不同 HTTP 版本之间进行转换的 Web 代理就像网关一样，它们会执行复杂的逻辑，以便在各个端点之间进行沟通。但因为它们在两侧使用的都是 HTTP，所以从技术上来讲，它们还是代理。

hardware.com。网关代表客户端执行 FTP 事务，并通过 HTTP 将结果回送给客户端。



（a）MSIE的手工代理设置　　（b）Navigator的手工代理设置

图 8-3　配置一个 HTTP/FTP 网关

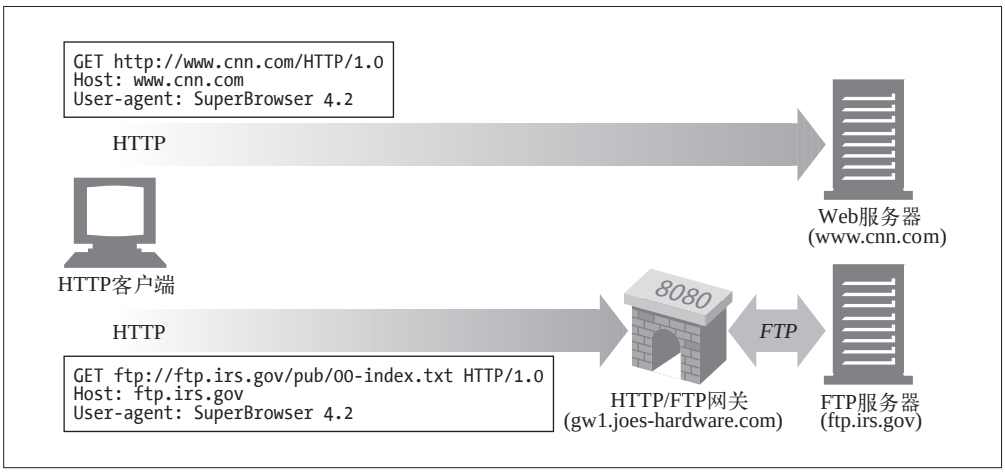


图 8-4　浏览器可以通过配置，让特定的协议使用特定的网关

后面的小节会介绍各种常见网关类型：服务器协议转换器、服务器端安全网关、客户端安全网关以及应用程序服务器。

8.2.1　HTTP/*：服务器端Web网关

请求流入原始服务器时，服务器端 Web 网关会将客户端 HTTP 请求转换为其他协议（参见图 8-5）。

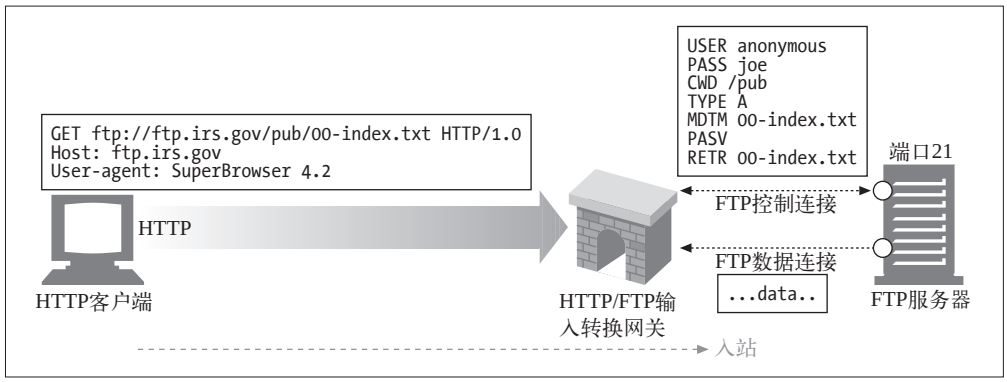


图 8-5　HTTP/FTP 网关将 HTTP 请求转换成 FTP 请求

200 在图 8-5 中，网关收到了一条对 FTP 资源的 HTTP 请求：

ftp://ftp.irs.gov/pub/00-index.txt

网关会打开一条到原始服务器 FTP 端口（端口 21）的 FTP 连接，通过 FTP 协议获取对象。网关会做下列事情：

- 发送 USER 和 PASS 命令登录到服务器上去；
- 发布 CWD 命令，转移到服务器上合适的目录中去；
- 将下载类型设置为 ASCII；
- 用 MDTM 获取文档的最后修改时间；
- 用 PASV 告诉服务器将有被动数据获取请求到达；
- 用 RETR 请求进行对象获取；
- 打开到 FTP 服务器的数据连接，服务器端口由控制信道返回；一旦数据信道打
201 开了，就将对象内容回送给网关。

完成获取之后，会将对象放在一条 HTTP 响应中回送给客户端。

8.2.2　HTTP/HTTPS：服务器端安全网关

一个组织可以通过网关对所有的输入 Web 请求加密，以提供额外的隐私和安全性保护。客户端可以用普通的 HTTP 浏览 Web 内容，但网关会自动加密用户的对话（参见图 8-6）。

8.2.3　HTTPS/HTTP客户端安全加速器网关

最近，将 HTTPS/HTTP 网关作为安全加速器使用的情况是越来越多了。这些 HTTPS/HTTP 网关位于 Web 服务器之前，通常作为不可见的拦截网关或反向代理

使用。它们接收安全的 HTTPS 流量，对安全流量进行解密，并向 Web 服务器发送普通的 HTTP 请求（参见图 8-7）。

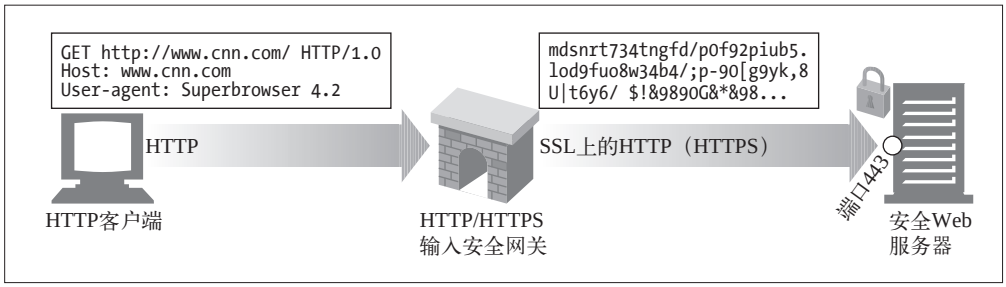


图 8-6　输入 HTTP/HTTPS 安全网关

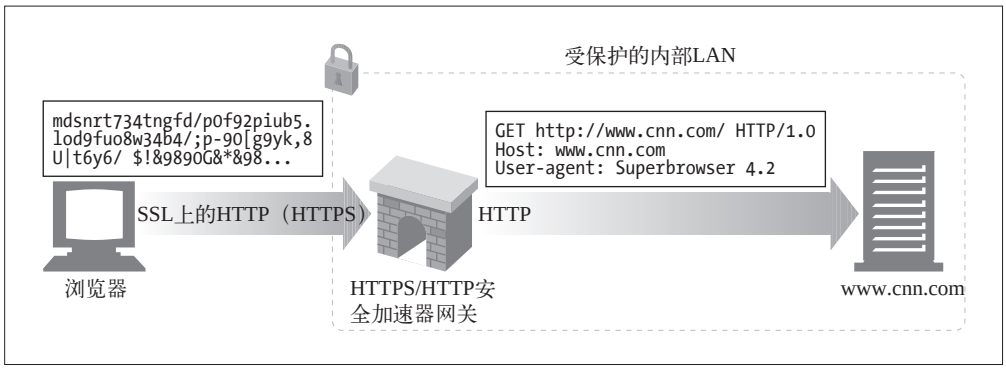


图 8-7　HTTPS/HTTP 安全加速器网关

这些网关中通常都包含专用的解密硬件，以比原始服务器有效得多的方式来解密安全流量，以减轻原始服务器的负荷。这些网关在网关和原始服务器之间发送的是未加密的流量，所以，要谨慎使用，确保网关和原始服务器之间的网络是安全的。 202

8.3　资源网关

到目前为止，我们一直在讨论通过网络连接客户端和服务器的网关。但最常见的网关，应用程序服务器，会将目标服务器与网关结合在一个服务器中实现。应用程序服务器是服务器端网关，与客户端通过 HTTP 进行通信，并与服务器端的应用程序相连（参见图 8-8）。

在图 8-8 中，两个客户端是通过 HTTP 连接到应用程序服务器的。但应用程序服务器并没有回送文件，而是将请求通过一个网关应用编程接口（Application Programming Interface，API）发送给运行在服务器上的应用程序。

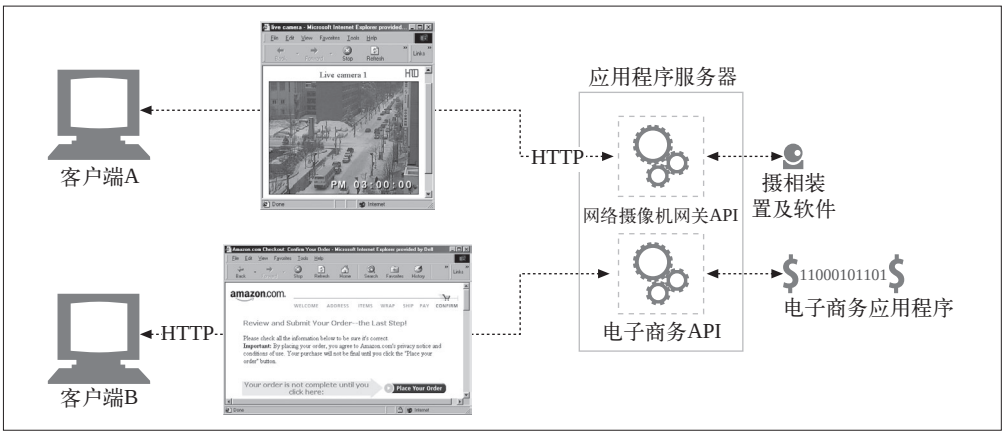


图 8-8　应用程序服务器可以将 HTTP 客户端连接任意后台应用程序

- 收到客户端 A 的请求，根据 URI 将其通过 API 发送给一个数码摄相机应用程序。将得到的图片绑定到一条 HTTP 响应报文中，再回送给客户端，在客户端的浏览器中显示。
- 客户端 B 的 URI 请求的是一个电子商务应用程序。客户端 B 的请求是通过服务器网关 API 发送给电子商务软件的，结果会被回送给浏览器。电子商务软件与客户端进行交互，引导用户通过一系列 HTML 页面来完成购物。

第一个流行的应用程序网关 API 就是通用网关接口（Common Gateway Interface，CGI）。CGI 是一个标准接口集，Web 服务器可以用它来装载程序以响应对特定 URL 的 HTTP 请求，并收集程序的输出数据，将其放在 HTTP 响应中回送。在过去的几年中，商业 Web 服务器提供了一些更复杂的接口，以便将 Web 服务器连接到
203 应用程序上去。

早期的 Web 服务器是相当简单的，在网关接口的实现过程中采用的简单方式一直持续到了今天。

请求需要使用网关的资源时，服务器会请辅助应用程序来处理请求。服务器会将辅助应用程序所需的数据传送给它。通常就是整条请求，或者用户想在数据库上运行的请求（来自 URL 的请求字符串，参见第 2 章）之类的东西。

然后，它会向服务器返回一条响应或响应数据，服务器则会将其转发给客户端。服务器和网关是相互独立的应用程序，因此，它们的责任是分得很清楚的。图 8-9 显示了服务器与网关应用程序之间交互的基本运行机制。

这个简单的协议（输入请求，转交，响应）就是最古老，也最常用的服务器扩展接口 CGI 的本质。

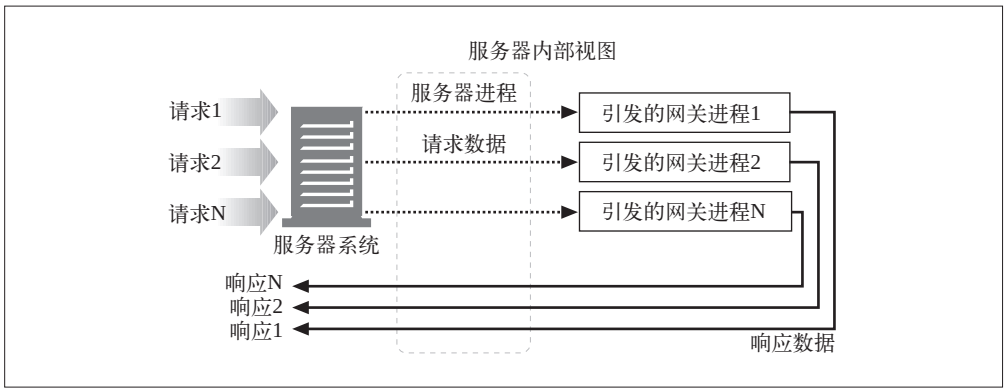


图 8-9　服务器网关应用程序机制

8.3.1　CGI

CGI 是第一个，可能仍然是得到最广泛使用的服务器扩展。在 Web 上广泛用于动态 HTML、信用卡处理以及数据库查询等任务。

CGI 应用程序是独立于服务器的，所以，几乎可以用任意语言来实现，包括 Perl、Tcl、C 和各种 shell 语言。CGI 很简单，几乎所有的 HTTP 服务器都支持它。图 8-9 显示了 CGI 模型的基本运行机制。

CGI 的处理对用户来说是不可见的。从客户端的角度来看，就像发起一个普通请求一样。它完全不清楚服务器和 CGI 应用程序之间的转接过程。URL 中出现字符 cgi 和可能出现的“?”是客户端发现使用了 CGI 应用程序的唯一线索。

204

看来 CGI 是很棒的，对吧？嗯，好吧，既是也不是。它在服务器和众多的资源类型之间提供了一种简单的、函数形式的粘合方式，用来处理各种需要的转换。这个接口还能很好地保护服务器，防止一些糟糕的扩展对它造成的破坏（如果这些扩展直接与服务器相连，造成的错误可能会引发服务器崩溃）。

但是，这种分离会造成性能的耗费。为每条 CGI 请求引发一个新进程的开销是很高的，会限制那些使用 CGI 的服务器的性能，并且会加重服务端机器资源的负担。为了解决这个问题，人们开发了一种新型 CGI——并将其恰当地称为快速 CGI。这个接口模拟了 CGI，但它是作为持久守护进程运行的，消除了为每个请求建立或拆除新进程所带来的性能损耗。

8.3.2　服务器扩展API

CGI 协议为外部翻译器与现有的 HTTP 服务器提供了一种简洁的接口方式，但如

果想要改变服务器自身的行为，或者只是想尽可能地提升能从服务器上获得的性能呢？服务器开发者为这两种需求提供了几种服务器扩展 API，为 Web 开发者提供了强大的接口，以便他们将自己的模块与 HTTP 服务器直接相连。扩展 API 允许程序员将自己的代码嫁接到服务器上，或者用自己的代码将服务器的一个组件完整地替换出来。

大多数流行的服务器都会为开发者提供一个或多个扩展 API。这些扩展通常都会绑定在服务器自身的结构上，所以，大多数都是某种服务器类型特有的。微软、网景、Apache 和其他服务器都提供了一些 API 接口，允许开发者通过这些接口改变服务器的行为，或者为不同的资源提供一些定制的接口。这些定制接口为开发者提供了强大的接口方式。

微软的 FPSE（FrontPage 服务器端扩展）就是服务器扩展的一个实例，它为使用 FrontPage 的作者进行 Web 发布提供支持。FPSE 能够对 FrontPage 客户端发送的 RPC（remote procedure call，远程过程调用）命令进行解释。这些命令会在 HTTP 中（具体来说，就是在 HTTP POST 方法上）捎回。细节请参见 19.1 节。

8.4 应用程序接口和Web服务

我们已经讨论过可以将资源网关作为 Web 服务器与应用程序的通信方式使用。更广泛地说，随着 Web 应用程序提供的服务类型越来越多，有一点变得越来越清晰了：HTTP 可以作为一种连接应用程序的基础软件来使用。在将应用程序连接起来的过程中，一个更为棘手的问题是在两个应用程序之间进行协议接口的协商，以便这些
205 应用程序可以进行数据的交换——这通常都是针对具体应用程序的个案进行的。

应用程序之间要配合工作，所要交互的信息比 HTTP 首部所能表达的信息要复杂得多。第 19 章描述了几个用于交换定制信息的扩展 HTTP 或 HTTP 上层协议实例。19.1 节介绍的是在 HTTP POST 报文之上建立 RPC 层，19.2 节介绍的是向 HTTP 首部添加 XML 的问题。

因特网委员会开发了一组允许 Web 应用程序之间相互通信的标准和协议。尽管 Web 服务（Web service）可以用来表示独立的 Web 应用程序（构造模块），这里我们还是宽松地用这个术语来表示这些标准。Web 服务的引入并不新鲜，但这是应用程序共享信息的一种新机制。Web 服务是构建在标准的 Web 技术（比如 HTTP）之上的。

Web 服务可以用 XML 通过 SOAP 来交换信息。XML（Extensible Markup Language，扩展标记语言）提供了一种创建数据对象的定制信息，并对其进行解释的方法。SOAP（Simple Object Access Protocol，简单对象访问协议）是向 HTTP 报文中添加

XML 信息的标准方式。[3]

8.5 隧道

我们已经讨论了几种不同的方式，通过这些方式可以用 HTTP 对不同类型的资源进行访问（通过网关），或者是用 HTTP 来启动应用程序到应用程序的通信。在本节中，我们要看看 HTTP 的另一种用法——Web 隧道（Web tunnel），这种方式可以通过 HTTP 应用程序访问使用非 HTTP 协议的应用程序。

Web 隧道允许用户通过 HTTP 连接发送非 HTTP 流量，这样就可以在 HTTP 上捎带其他协议数据了。使用 Web 隧道最常见的原因就是要在 HTTP 连接中嵌入非 HTTP 流量，这样，这类流量就可以穿过只允许 Web 流量通过的防火墙了。

8.5.1 用CONNECT建立HTTP隧道

Web 隧道是用 HTTP 的 CONNECT 方法建立起来的。CONNECT 方法并不是 HTTP/1.1 核心规范的一部分，[4] 但却是一种得到广泛应用的扩展。可以在 Ari Luotonen 的过期因特网草案规范“Tunneling TCP based protocols through Web proxy servers”（“通过 Web 代理服务器用隧道方式传输基于 TCP 的协议”），或他的著作 *Web Proxy Servers* 中找到这些技术规范，本章末尾引用了这两份资源。

206

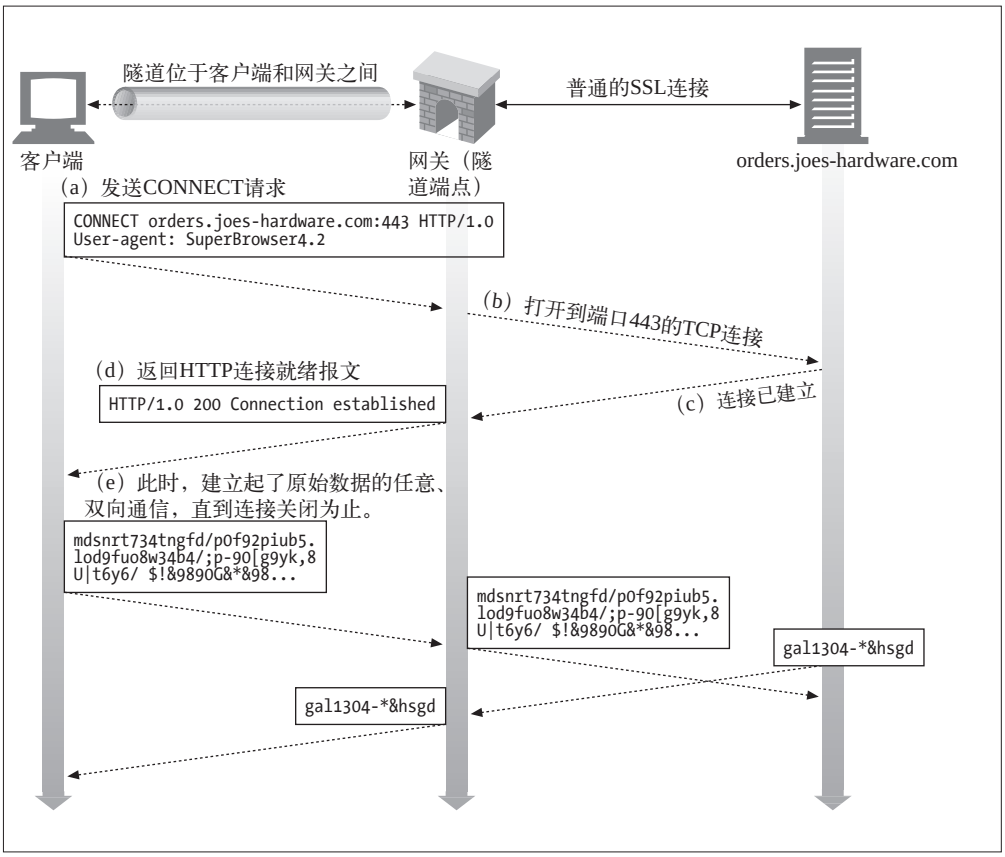
CONNECT 方法请求隧道网关创建一条到达任意目的服务器和端口的 TCP 连接，并对客户端和服务器之间的后继数据进行盲转发。

图 8-10 显示了 CONNECT 方法如何建立起一条到达网关的隧道。

- 在图 8-10a 中，客户端发送了一条 CONNECT 请求给隧道网关。客户端的 CONNECT 方法请求隧道网关打开一条 TCP 连接（在这里，打开的是到主机 orders.joes-hardware.com 的标准 SSL 端口 443 的连接）。
- 在图 8-10b 和图 8-10c 中创建了 TCP 连接。
- 一旦建立了 TCP 连接，网关就会发送一条 HTTP 200 Connection Established 响应来通知客户端（参见图 8-10d）。
- 此时，隧道就建立起来了。客户端通过 HTTP 隧道发送的所有数据都会被直接转发给输出 TCP 连接，服务器发送的所有数据都会通过 HTTP 隧道转发给客户端。

注 3：更多信息，请参见 http://www.w3.org/TR/2001/WD-soap12-part0-20011217/。Doug Tidwell、James Snell 和 Pavel Kulchenko 编写的 *Programming Web Services with SOAP*（SOAP Web 服务开发）一书（O’Reilly）也是非常好的 SOAP 协议信息资源。

注 4：HTTP/1.1 规范保留了 CONNECT 方法，但没有对其功能进行描述。



207 图 8-10 用 CONNECT 建立一条 SSL 隧道

图 8-10 中的例子描述了一条 SSL 隧道，其中的 SSL 流量是在一条 HTTP 连接上发送的，但是通过 CONNECT 方法可以与使用任意协议的任意服务器建立 TCP 连接的。

1. CONNECT请求

除了起始行之外，CONNECT 的语法与其他 HTTP 方法类似。一个后面跟着冒号和端口号的主机名取代了请求 URI。主机和端口都必须指定：

```
CONNECT home.netscape.com:443 HTTP/1.0
User-agent: Mozilla/4.0
```

和其他 HTTP 报文一样，起始行之后，有零个或多个 HTTP 请求首部字段。这些行照例以 CRLF 结尾，首部列表以一个仅有 CRLF 的空行结束。

2. CONNECT响应

发送了请求之后，客户端会等待来自网关的响应。和普通 HTTP 报文一样，响应码 200 表示成功。按照惯例，响应中的原因短语通常被设置为“Connection Established”：

```
HTTP/1.0 200 Connection Established
Proxy-agent: Netscape-Proxy/1.1
```

与普通 HTTP 响应不同，这个响应并不需要包含 `Content-Type` 首部。此时连接只是对原始字节进行转接，不再是报文的承载者，所以不需要使用内容类型了。[5]

8.5.2 数据隧道、定时及连接管理

管道化数据对网关是不透明的，所以网关不能对分组的顺序和分组流作任何假设。一旦隧道建立起来了，数据就可以在任意时间流向任意方向了。[6]

作为一种性能优化方法，允许客户端在发送了 CONNECT 请求之后，接收响应之前，发送隧道数据。这样可以更快地将数据发送给服务器，但这就意味着网关必须能够正确处理跟在请求之后的数据。尤其是，网关不能假设网络 I/O 请求只会返回首部数据，网关必须确保在连接准备就绪时，将与首部一同读进来的数据发送给服务器。在请求之后以管道方式发送数据的客户端，如果发现回送的响应是认证请求，或者其他非 200 但不致命的错误状态，就必须做好重发请求数据的准备。[7]

208

如果在任意时刻，隧道的任意一个端点断开了连接，那个端点发出的所有未传输数据都会被传送给另一个端点，之后，到另一个端点的连接也会被代理终止。如果还有数据要传输给关闭连接的端点，数据会被丢弃。

8.5.3 SSL隧道

最初开发 Web 隧道是为了通过防火墙来传输加密的 SSL 流量。很多组织都会将所有流量通过分组过滤路由器和代理服务器以隧道方式传输，以提升安全性。但有些协议，比如加密 SSL，其信息是加密的，无法通过传统的代理服务器转发。隧道会通过一条 HTTP 连接来传输 SSL 流量，以穿过端口 80 的 HTTP 防火墙（参见图 8-11）。

注 5：为了实现一致性，今后的规范可能会为隧道定义一个媒体类型（比如 application/tunnel）。

注 6：隧道的两端（客户端和网关）必须做好在任意时刻接收来自连接任一端分组的准备，而且必须将数据立即转发出去。由于隧道化协议中可能包含了数据的依赖关系，所以隧道的任一端都不能忽略输入数据。隧道一端对数据的消耗不足可能会将隧道另一端的数据生产者挂起，造成死锁。

注 7：传送的数据不要超过请求 TCP 分组的剩余容量。如果在收到所有管道化传输的 TCP 分组之前，网关关闭了连接，那么，管道化传输的多余数据就会使客户端 TCP 重置。TCP 重置会使客户端丢失收到的网关响应，这样客户端就无法分辨错误是由于网络错误、访问控制，还是认证请求造成的了。

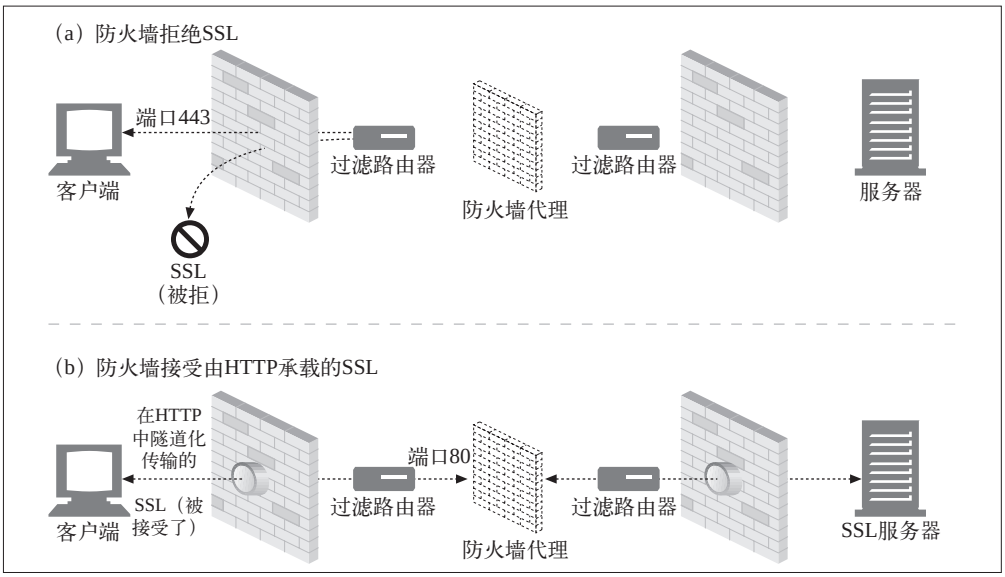


图 8-11　隧道可以经由 HTTP 连接传输非 HTTP 流量

为了让 SSL 流量经现存的代理防火墙进行传输，HTTP 中添加了一项隧道特性，在此特性中，可以将原始的加密数据放在 HTTP 报文中，通过普通的 HTTP 信道传送（参见图 8-12）。

209

在图 8-12a 中，SSL 流量被直接发送给了一个（SSL 端口 443 上的）安全 Web 服务器。在图 8-12b 中，SSL 流量被封装到一条 HTTP 报文中，并通过 HTTP 端口 80 上的连接发送，最后被解封装为普通的 SSL 连接。

通常会用隧道将非 HTTP 流量传过端口过滤防火墙。这一点可以得到很好的利用，比如，通过防火墙传输安全 SSL 流量。但是，这项特性可能会被滥用，使得恶意协议通过 HTTP 隧道流入某个组织内部。

8.5.4　SSL隧道与HTTP/HTTPS网关的对比

可以像其他协议一样，对 HTTPS 协议（SSL 上的 HTTP）进行网关操作：由网关（而不是客户端）初始化与远端 HTTPS 服务器的 SSL 会话，然后代表客户端执行 HTTPS 事务。响应会由代理接收并解密，然后通过（不安全的）HTTP 传送给客户端。这是网关处理 FTP 的方式。但这种方式有几个缺点：

- 客户端到网关之间的连接是普通的非安全 HTTP；
- 尽管代理是已认证主体，但客户端无法对远端服务器执行 SSL 客户端认证（基于 X509 证书的认证）；

- 网关要支持完整的 SSL 实现。 210

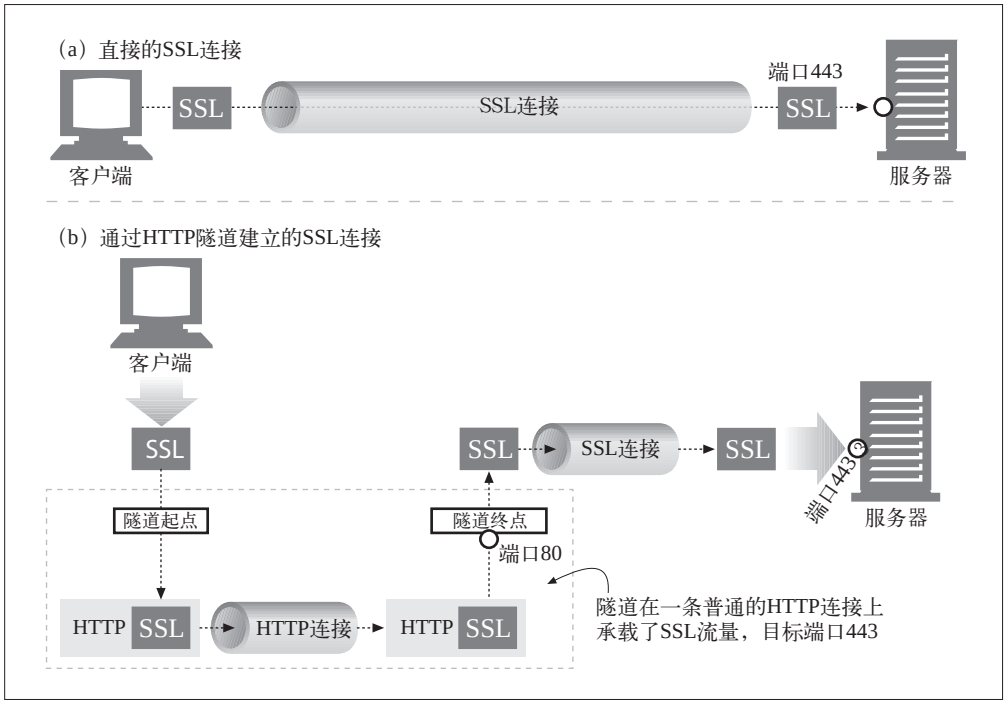


图 8-12　直接的 SSL 连接与隧道化 SSL 连接的对比

注意，对于 SSL 隧道机制来说，无需在代理中实现 SSL。SSL 会话是建立在产生请求的客户端和目的（安全的）Web 服务器之间的，中间的代理服务器只是将加密数据经过隧道传输，并不会在安全事务中扮演其他的角色。

8.5.5　隧道认证

在适当的情况下，也可以将 HTTP 的其他特性与隧道配合使用。尤其是，可以将代理的认证支持与隧道配合使用，对客户端使用隧道的权利进行认证（参见图 8-13）。

8.5.6　隧道的安全性考虑

总的来说，隧道网关无法验证目前使用的协议是否就是它原本打算经过隧道传输的协议。因此，比如说，一些喜欢捣乱的用户可能会通过本打算用于 SSL 的隧道，越过公司防火墙传递因特网游戏流量，而恶意用户可能会用隧道打开 Telnet 会话，或用隧道绕过公司的 E-mail 扫描器来发送 E-mail。 211

为了降低对隧道的滥用，网关应该只为特定的知名端口，比如 HTTPS 的端口 443，打开隧道。

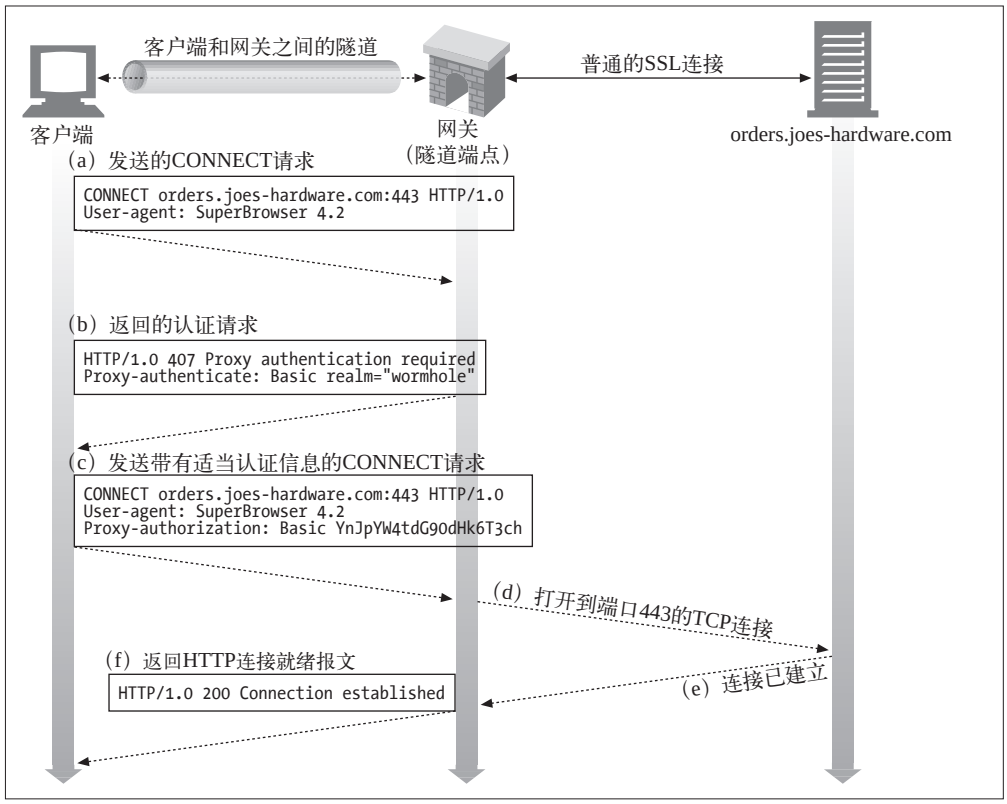


图 8-13　网关允许某客户端使用隧道之前，可以对其进行代理认证

8.6　中继

HTTP 中继（relay）是没有完全遵循 HTTP 规范的简单 HTTP 代理。中继负责处理 HTTP 中建立连接的部分，然后对字节进行盲转发。

HTTP 很复杂，所以实现基本的代理功能并对流量进行盲转发，而且不执行任何首部和方法逻辑，有时是很有用的。盲中继很容易实现，所以有时会提供简单的过滤、诊断或内容转换功能。但这种方式可能潜在严重的互操作问题，所以部署的时候要特别小心。

某些简单盲中继实现中存在的一个更常见（也更声名狼藉的）问题是，由于它们无法正确处理 `Connection` 首部，所以有潜在的挂起 keep-alive 连接的可能。图 8-14 对这种情况进行了说明。

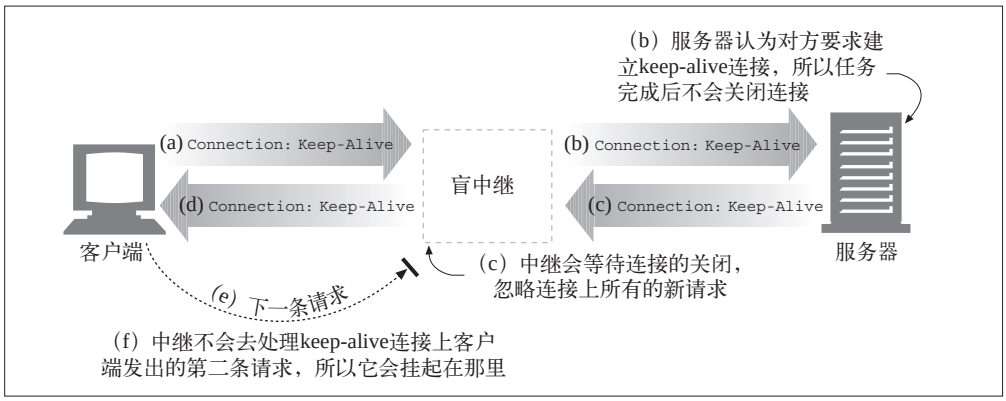


图 8-14　如果简单盲中继是单任务的，且不支持 Connection 首部，就会挂起

这张图中发生的情况如下所述。

- 在图 8-14a 中，Web 客户端向中继发送了一条包含 `Connection: Keep-Alive` 首部的报文，如果可能的话要求建立一条 keep-alive 连接。客户端等待响应，以确定它要求建立 keep-alive 信道的请求是否被认可了。
- 中继收到了这条 HTTP 请求，但它并不理解 `Connection` 首部，因此会将报文一字不漏地沿着链路传递给服务器（参见图 8-14b）。但 `Connection` 首部是个逐跳首部；只适用于单条传输链路，是不应该沿着链路传送下去的。要有不好的事情发生了！ 212
- 在图 8-14b 中，经过中继转发的 HTTP 请求抵达 Web 服务器。当 Web 服务器收到经过代理转发的 `Connection: Keep-Alive` 首部时，会错误地认为中继（对服务器来说，它看起来就和其他客户端一样）要求进行 keep-alive 的对话！这对 Web 服务器来说没什么问题——它同意进行 keep-alive 对话，并在图 8-14c 中回送了一个 `Connection: Keep-Alive` 响应首部。那么，此时，Web 服务器就认为它是在与中继进行 keep-alive 对话，会遵循 keep-alive 对话的规则。但中继对 keep-alive 会话根本就一无所知。
- 在图 8-14d 中，中继将 Web 服务器的响应报文，以及来自 Web 服务器的 `Connection: Keep-Alive` 首部一起发回给客户端。客户端看到这个首部，认为中继同意进行 keep-alive 对话。此时，客户端和服务器都认为它们是在进行 keep-alive 对话，但与它们进行对话的中继却根本不知道什么 keep-alive 对话。
- 中继对持久对话一无所知，所以它会将收到的所有数据都转发给客户端，等待原始服务器关闭连接。但原始服务器认为中继要求服务器将连接保持在活跃状态，所以是不会关闭连接的！这样，中继就会挂起，等待连接的关闭。
- 在图 8-14d 中，当客户端收到回送的响应报文时，它会直接转向第二条请求，在 keep-alive 连接上向中继发送另一条请求（参见图 8-14e）。简单中继通常不会期

待同一条连接上还会有另一条请求到达。浏览器上的圈会不停地转，但没有任何进展。

有一些方法可以使中继稍微智能一些，以消除这些风险，但所有简化的代理都存在着出现互操作性问题的风险。要为某个特定目标构建简单的 HTTP 中继，一定要特别注意其使用方法。对任何大规模部署来说，都要非常认真地考虑使用真正的、完全遵循 HTTP 的代理服务器。

更多与中继和连接管理有关的信息，参见 4.5.6 节。

8.7 更多信息

更多信息，请参见下列参考材料。

- http://www.w3.org/Protocols/rfc2616/rfc2616.txt
  RFC 2616， 由 R. Fielding、J. Gettys、J. Mogul、H. Frystyk、L. Mastinter、P. Leach 和 T. Berners-Lee 编写的“Hypertext Transfer Protocol”。

- *Web Proxy Servers*（《Web 代理服务器》）
213 Ari Luotonen，Prentice Hall 出版的计算机图书。

- http://www.alternic.org/drafts/drafts-l-m/draft-luotonen-Web-proxy-tunneling-01.txt
  Ari Luotonen 编写的“Tunneling TCP based protocols through Web proxy servers”（“用隧道方式通过 Web 代理服务器传输基于 TCP 的协议”）。

- http://cgi-spec.golux.com
  通用网关接口——RFC 项目页面。

- http://www.w3.org/TR/2001/WD-soap12-part0-20011217/
  W3C——SOAP 版本 1.2 工作草案。

- *Programming Web Services with Soap*[8]（《Soap Web 服务开发》）
  James Snell、Doug Tidwell 和 Pavel Kulchenko 编写，O’Reilly & Associates 公司出版。

- http://www.w3.org/TR/2002/WD-wsa-reqs-20020429
  W3C——Web 服务的架构要求。

- *Web Services Essentials*[9]（《Web 服务精髓》）
214 Ethan Cerami，O’Reilly & Associates 公司出版。

注 8~9：这二本书的中文版已由中国电力出版社出版。（编者注）

第9章

Web机器人

本章我们来仔细了解一下被称为 Web 机器人（Web robot）的自活跃（self-animating）用户代理，以继续我们的 HTTP 架构之旅。

Web 机器人是能够在无需人类干预的情况下自动进行一系列 Web 事务处理的软件程序。很多机器人会从一个 Web 站点逛到另一个 Web 站点，获取内容，跟踪超链，并对它们找到的数据进行处理。根据这些机器人自动探查 Web 站点的方式，人们为它们起了一些各具特色的名字，比如“爬虫”、“蜘蛛”、“蠕虫”以及“机器人”等，就好像它们都有自己的头脑一样。

这里有几个 Web 机器人的示例。

- 股票图形机器人每隔几分钟就会向股票市场的服务器发送 HTTP GET，用得到的数据来构建股市价格趋势图。
- Web 统计机器人会收集与万维网规模及发展有关的“统计”信息。它们会在 Web 上游荡，统计页面的数量，记录每个页面的大小、所用语言以及媒体类型。[1]
- 搜索引擎机器人会搜集它们所找到的所有文档，以创建搜索数据库。
- 比较购物机器人会从在线商店的目录中收集 Web 页面，构建商品及其价格的数据库。

9.1 爬虫及爬行方式

Web 爬虫是一种机器人，它们会递归地对各种信息性 Web 站点进行遍历，获取第一个 Web 页面，然后获取那个页面指向的所有 Web 页面，然后是那些页面指向的所有 Web 页面，依此类推。递归地追踪这些 Web 链接的机器人会沿着 HTML 超链
215 创建的网络“爬行”，所以将其称为爬虫（crawler）或蜘蛛（spider）。

因特网搜索引擎使用爬虫在 Web 上游荡，并把它们碰到的文档全部拉回来。然后对这些文档进行处理，形成一个可搜索的数据库，以便用户查找包含了特定单词的文档。网上有数万亿的 Web 页面需要查找和取回，这些搜索引擎蜘蛛必然是些最复杂的机器人。我们来进一步仔细地看看这些爬虫是怎样工作的。

9.1.1 从哪儿开始：根集

在把饥饿的爬虫放出去之前，需要给它一个起始点。爬虫开始访问的 URL 初始集合被称作根集（root set）。挑选根集时，应该从足够多不同的站点中选择 URL，这样，爬遍所有的链接才能最终到达大部分你感兴趣的 Web 页面。

注 1：http://www.netcraft.com 收集了大量统计度量值，用于统计 Web 站点使用的是哪种类型的服务器。

要在图 9-1 所示的 Web 上爬行，使用哪个根集比较好呢？与在真实的 Web 中一样，没有哪个文档最终可以链接到所有其他文档上去。如果从图 9-1 的文档 A 开始，可以到达 B、C 和 D，然后是 E 和 F，然后到 J，然后到 K。但没有从 A 到 G，或从 A 到 N 的链路。

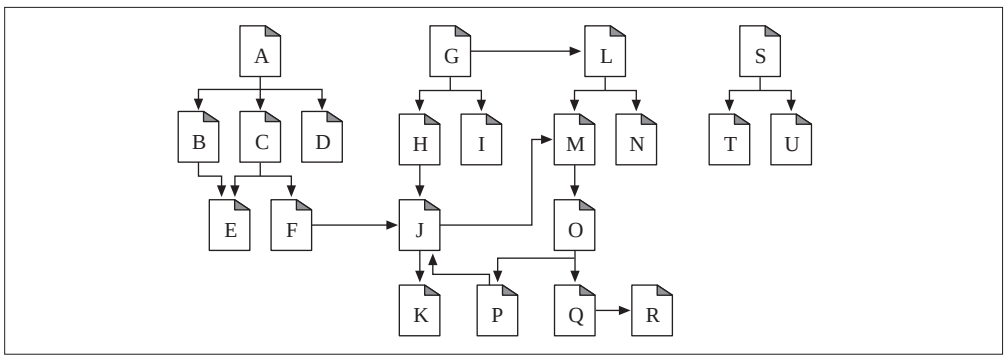


图 9-1　根集要能够到达所有的页面

这个 Web 结构中的某些 Web 页面，比如 S、T 和 U，几乎是被隔离开来的——它们是孤立的，没有任何链接指向它们。可能这些孤独的页面是一些新页面，还没人找到它们。或者可能是一些非常老的或不显眼的页面。

总之，根集中并不需要有很多页面，就可以涵盖一大片 Web 结构。在图 9-1 中，要抵达所有页面，根集中只需要有 A、G 和 S 就行了。

通常，一个好的根集会包括一些大的流行 Web 站点（比如 http://www.yahoo.com）、一个新创建页面的列表和一个不经常被链接的无名页面列表。很多大规模的爬虫产品，比如因特网搜索引擎使用的那些爬虫，都为用户提供了向根集中提交新页面或无名页面的方式。这个根集会随时间推移而增长，是所有新爬虫的种子列表。 216

9.1.2　链接的提取以及相对链接的标准化

爬虫在 Web 上移动时，会不停地对 HTML 页面进行解析。它要对所解析的每个页面上的 URL 链接进行分析，并将这些链接添加到需要爬行的页面列表中去。随着爬虫的前进，当其发现需要探查的新链接时，这个列表常常会迅速地扩张。[2] 爬虫要通过简单的 HTML 解析，将这些链接提取出来，并将相对 URL 转换为绝对形式。2.3.1 节讨论了如何进行这种转换。

注 2：我们会在 9.1.3 节开始讨论爬虫是否需要记住它们到过何处。在爬行过程中，这个已发现 URL 列表会不断扩张，直到已经对 Web 空间进行了彻底的探查为止，这时爬虫就会到达一个不再发现新链接的状态了。

9.1.3 避免环路的出现

机器人在 Web 上爬行时，要特别小心不要陷入循环，或环路（cycle）之中。我们来看看图 9-2 中所示的爬虫。

- 在图 9-2a 中，机器人获取页面 A，看到 A 链接到 B，就获取页面 B。
- 在图 9-2b 中，机器人获取页面 B，看到 B 链接到 C，就获取页面 C。
- 在图 9-2c 中，机器人获取页面 C，会看到 C 链接到 A。如果机器人再次获取页面 A，就会陷入一个环路中，获取 A、B、C、A、B、C、A……

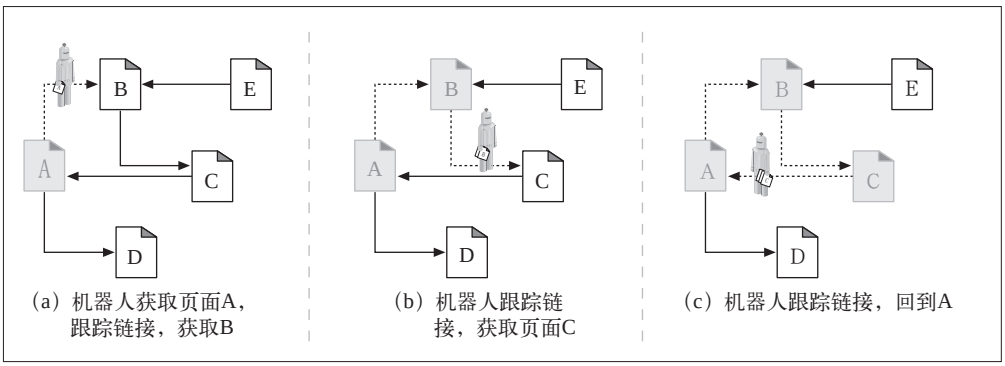


图 9-2 在 Web 的超链中爬行

机器人必须知道它们到过何处，以避免环路的出现。环路会造成机器人陷阱，这些陷阱会暂停或减缓机器人的爬行进程。

9.1.4 循环与复制

至少出于下列三个原因，环路对爬虫来说是有害的。

- 它们会使爬虫陷入可能会将其困住的循环之中。循环会使未经良好设计的爬虫不停地兜圈子，把所有时间都耗费在不停地获取相同的页面上。爬虫会消耗掉很多网络带宽，可能完全无法获取任何其他页面了。
- 爬虫不断地获取相同的页面时，另一端的 Web 服务器也在遭受着打击。如果爬虫与服务器连接良好，它就会击垮 Web 站点，阻止所有真实用户访问这个站点。这种拒绝服务是可以作为法律诉讼理由的。
- 即使循环自身不是什么问题，爬虫也是在获取大量重复的页面 [通常被称为“dups”（重复），以便与“loops”（循环）押韵]。爬虫应用程序会被重复的内容所充斥，这样应用程序就会变得毫无用处。返回数百份完全相同页面的因特网搜索引擎就是一个这样的例子。

9.1.5 面包屑留下的痕迹

但是，记录曾经到过哪些地方并不总是一件容易的事。编写本书时，因特网上有数十亿个不同的 Web 页面，其中还不包括那些由动态网关产生的内容。

如果要爬行世界范围内的一大块 Web 内容，就要做好访问数十亿 URL 的准备。记录下哪些 URL 已经访问过了是件很具挑战的事情。由于 URL 的数量巨大，所以，要使用复杂的数据结构以便快速判定哪些 URL 是曾经访问过的。数据结构在访问速度和内存使用方面都应该是非常高效的。

数亿 URL 需要具备快速搜索结构，所以速度是很重要的。穷举搜索 URL 列表是根本不可能的。机器人至少要用到搜索树或散列表，以快速判定某个 URL 是否被访问过。

数亿 URL 还会占用大量的空间。如果平均每个 URL 有 40 个字符长，而且一个 Web 机器人要爬行 5 亿个 URL（只是 Web 的一小部分），那么搜索数据结构只是存储这些 URL 就需要 20GB 或更多的存储空间（40 字节 /URL × 5 亿个 URL = 20GB）！

这里列出了大规模 Web 爬虫对其访问过的地址进行管理时使用的一些有用的技术。

- *树和散列表*
  复杂的机器人可能会用搜索树或散列表来记录已访问的 URL。这些是加速 URL 查找的软件数据结构。

- *有损的存在位图*
  为了减小空间，一些大型爬虫会使用有损数据结构，比如存在位数组（presence bit array）。用一个散列函数将每个 URL 都转换成一个定长的数字，这个数字在 218
  数组中有个相关的“存在位”。爬行过一个 URL 时，就将相应的“存在位”置位。如果存在位已经置位了，爬虫就认为已经爬行过那个 URL 了。[3]

- *检查点*
  一定要将已访问 URL 列表保存到硬盘上，以防机器人程序崩溃。

- *分类*
  随着 Web 的扩展，在一台计算机上通过单个机器人来完成爬行就变得不太现实了。那台计算机可能没有足够的内存、磁盘空间、计算能力，或网络带宽来完成爬行任务。

注 3：由于 URL 的潜在数量是无限的，而存在位数组中的比特数是有限的，所以有出现冲突的可能——两个 URL 可能会映射到同一个存在位上去。出现这种情况时，爬虫会错误地认为它已经爬行过某个实际未爬行过的页面了。在实际应用中，使用大量的存在位就可以使这种情况极少发生。产生冲突的后果就是爬虫会忽略某个页面。

有些大型 Web 机器人会使用机器人“集群”，每个独立的计算机是一个机器人，以汇接方式工作。为每个机器人分配一个特定的 URL“片”，由其负责爬行。这些机器人配合工作，爬行整个 Web。机器人个体之间可能需要相互通信，来回传送 URL，以覆盖出故障的对等实体的爬行范围，或协调其工作。

Witten 等人编写的 *Managing Gigabytes: Compressing and Indexing Documents and Images*（《海量数据管理——文档和图像的压缩与索引》）[4]，*Morgan Kaufmann* 出版社出版，是实现大规模数据结构的很好的参考书。这本书讲的全是管理大量数据所需的各种诀窍和技巧。

9.1.6 别名与机器人环路

由于 URL“别名”的存在，即使使用了正确的数据结构，有时也很难分辨出以前是否访问过某个页面。如果两个 URL 看起来不一样，但实际指向的是同一资源，就称这两个 URL 互为“别名”。

表 9-1 列出了不同 URL 指向同一资源的几种简单方式。

表9-1 同一文档的不同URL别名

	第一个URL	第二个URL	什么时候是别名
a	http://www.foo.com/bar.html	http://www.foo.com:80/bar.html	默认端口为 80
b	http://www.foo.com/~fred	http://www.foo.com/%7Ffred	%7F 与 ~ 相同
c	http://www.foo.com/x.html#early	http://www.foo.com/x.html#middle	标签并没有修改页面内容
d	http://www.foo.com/readme.htm	http://www.foo.com/README.HTM	服务器是大小写无关的
e	http://www.foo.com/	http://www.foo.com/index.html	默认页面为 index.html
f	http://www.foo.com/index.html	http://209.231.87.45/index.html	www.foo.com 使用这个 IP 地址

219

9.1.7 规范化URL

大多数 Web 机器人都试图通过将 URL“规范化”为标准格式来消除上面那些显而易见的别名。机器人首先可先通过下列步骤将每个 URL 都转化为规范化的格式。

(1) 如果没有指定端口的话，就向主机名中添加“:80”。
(2) 将所有转义符 %xx 都转换成等价字符。
(3) 删除 # 标签。

注 4：本书中文版已由科学出版社出版。（编者注）

经过这些步骤就可以消除表 9-1 中 a~c 所列的别名问题了。但如果不知道特定 Web 服务器的相关信息，机器人就没什么好办法来避免表 9-1 中 d~f 的问题了。

- 机器人需要知道 Web 服务器是否是大小写无关的才能避免表 9-1d 中的别名问题。
- 机器人需要知道 Web 服务器上这个目录下的索引页面配置才能知道表 9-1e 中的情况是否是别名。
- 即使机器人知道表 9-1f 中的主机名和 IP 地址都指向同一台计算机，它也还要知道 Web 服务器是否配置为进行（参见第 5 章）虚拟主机操作，才能知道这个 URL 是不是别名。

URL 规范化可以消除一些基本的语法别名，但机器人还会遇到其他的、将 URL 转换为标准形式也无法消除的 URL 别名。

9.1.8 文件系统连接环路

文件系统中的符号连接会造成特定的潜在环路，因为它们会在目录层次深度有限的情况下，造成深度无限的假象。符号连接环路通常是由服务器管理员的无心错误造成的，但“邪恶的网管”也可能会恶意地为机器人制造这样的陷阱。

图 9-3 显示了两个文件系统。在图 9-3a 中，subdir 是个普通的目录。在图 9-3b 中，subdir 是个指回到“/”的符号连接。在这两个图中，都假设文件 /index.html 中包含了一个指向文件 subdir/index.html 的超链。

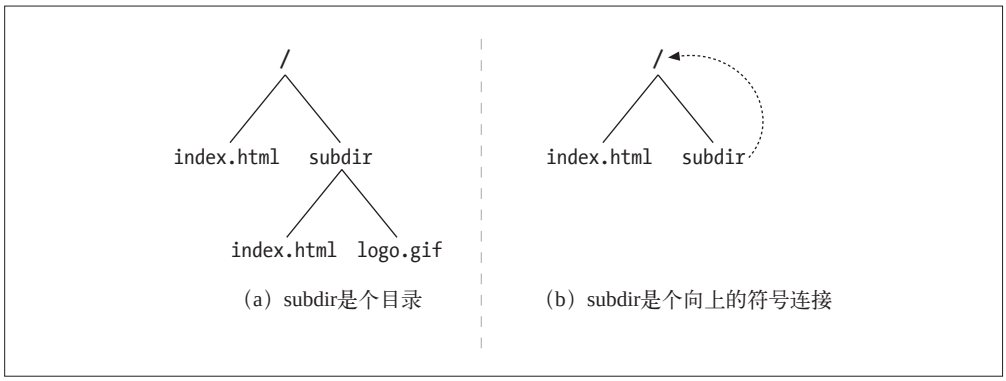


图 9-3 符号连接环路

使用图 9-3a 所示的文件系统时，Web 爬虫可能会采取下列动作：

(1) GET http://www.foo.com/index.html
获取 /index.html，找到指向 subdir/index.html 的链接。

(2) GET http://www.foo.com/subdir/index.html
获取 subdir/index.html，找到指向 subdir/logo.gif 的链接。

(3) GET http://www.foo.com/subdir/logo.gif
220 获取 subdir/logo.gif，再没有链接了，结束。

但在图 9-3b 的文件系统中，可能会发生下列情况：

(1) GET http://www.foo.com/index.html
获取 /index.html，找到指向 subdir/index.html 的链接。

(2) GET http://www.foo.com/subdir/index.html
获取 subdir/index.html，但得到的还是同一个 index.html。

(3) GET http://www.foo.com/subdir/subdir/index.html
获取 subdir/subdir/index.html。

(4) GET http://www.foo.com/subdir/subdir/subdir/index.html
获取 subdir/subdir/subdir/index.html。

图 9-3b 的问题是 subdir/ 是个指向“/”的环路，但由于 URL 看起来有所不同，所以机器人无法单从 URL 本身判断出文档是相同的。毫无戒备的机器人就有了陷入循环的危险。如果没有某种循环检测方式，这个环路就会继续下去，通常会持续到 URL 的长度超过机器人或服务器的限制为止。

9.1.9 动态虚拟Web空间

恶意网管可能会有意创建一些复杂的爬虫循环来陷害那些无辜的、毫无戒备的机器人。尤其是，发布一个看起来像普通文件，实际上却是网关应用程序的 URL 是很容易的。这个应用程序可以在传输中构造出包含了到同一服务器上虚构 URL 链接的 HTML。请求这些虚构的 URL 时，这个邪恶的服务器就会捏造出一个带有新的虚构 URL 的新 HTML 页面来。

即使这个恶意 Web 服务器实际上并不包含任何文件，它也可以通过无限虚拟的 Web 空间将可怜的机器人带入爱丽丝漫游仙境之旅。更糟的是，每次的 URL 和 HTML 看起来都有很大的不同，机器人很难检测到环路。图 9-4 显示了一个恶意 Web 服务
221 器生成假内容的例子。

更常见的情况是，那些没有恶意的网管们可能会在无意中通过符号连接或动态内容构造出爬虫陷阱。比如，我们来看一个基于 CGI 的日历程序，它会生成一个月历和

一个指向下个月的链接。真正的用户是不会不停地请求下个月的链接的，但不了解其内容的动态特性的机器人可能会不断向这些资源发出无穷的请求。[5]

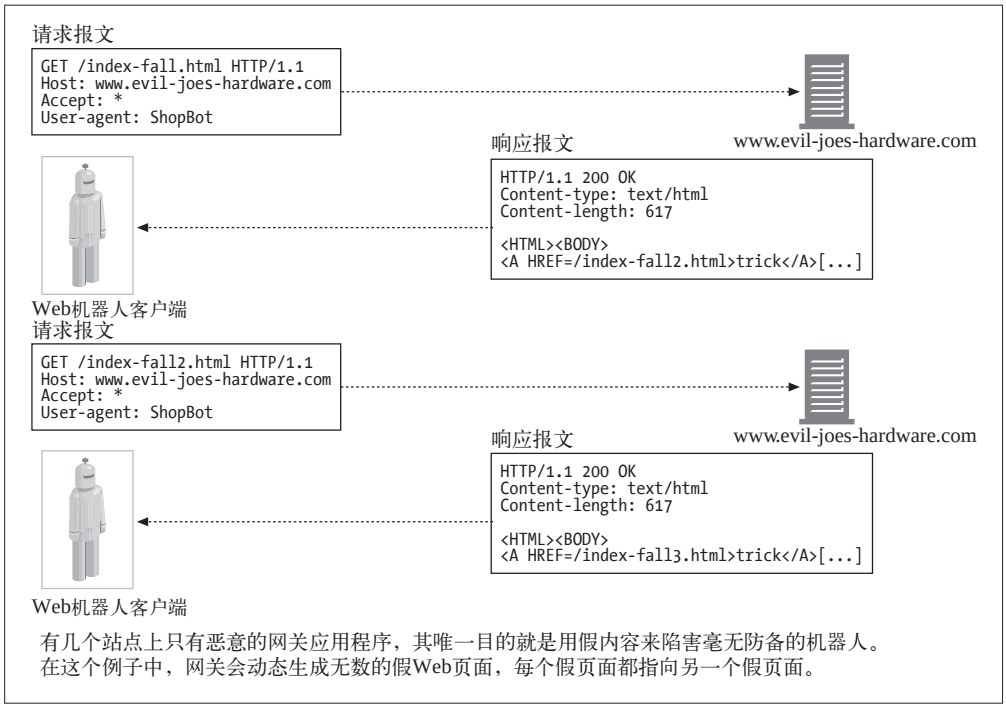


图 9-4　恶意的动态 Web 空间示例

9.1.10　避免循环和重复

没有什么简单明了的方式可以避免所有的环路。实际上，经过良好设计的机器人中要包含一组试探方式，以避免环路的出现。

总的说来，爬虫的自动化程度越高（人为的监管越少），越可能陷入麻烦之中。机器人的实现者需要做一些取舍——这些试探方式有助于避免问题的出现，但你可能会终止扫描那些看起来可疑的有效内容，因此这种方式也是“有损失”的。 222

在机器人会遇到的各种危险的 Web 中，有些技术的使用可以使机器人有更好的表现。

注 5：这是 http://www.searchtools.com/robots/robot-checklist.html 上提到的日历站点 http://cgi.umbc.edu/cgi-bin/WebEvent/Webevent.cgi 上的真实例子。这样的动态内容带来的后果就是，很多机器人都拒绝爬行 URL 中包含子字符串“cgi”的页面。

- 规范化 URL
  将 URL 转换为标准形式以避免语法上的别名

- 广度优先的爬行
  每次爬虫都有大量潜在的 URL 要去爬行。以广度优先的方式来调度 URL 去访问 Web 站点，就可以将环路的影响最小化。即使碰到了机器人陷阱，也可以在回到环路中获取的下一个页面之前，从其他 Web 站点中获取成百上千的页面。如果采用深度优先方式，一头扎到单个站点中去，就可能会跳入环路，永远无法访问其他站点。[6]

- 节流[7]
  限制一段时间内机器人可以从一个 Web 站点获取的页面数量。如果机器人跳进了一个环路，试图不断地访问某个站点的别名，也可以通过节流来限制重复的页面总数和对服务器的访问总数。

- 限制 URL 的大小
  机器人可能会拒绝爬行超出特定长度（通常是 1KB）的 URL。如果环路使 URL 的长度增加，长度限制就会最终终止这个环路。有些 Web 服务器在使用长 URL 时会失败，因此，被 URL 增长环路困住的机器人会使某些 Web 服务器崩溃。这会让网管错误地将机器人当成发起拒绝服务攻击的攻击者。

  要小心，这种技术肯定会让你错过一些内容。现在很多站点都会用 URL 来管理用户的状态（比如，在一个页面引用的 URL 中存储用户 ID）。用 URL 长度来限制爬虫可能会带来些麻烦；但如果每当请求的 URL 达到某个特定长度时，都记录一次错误的话，就可以为用户提供一种检查某特定站点上所发生情况的方法。

- URL/ 站点黑名单
  维护一个与机器人环路和陷阱相对应的已知站点及 URL 列表，然后像躲避瘟疫一样避开它们。发现新问题时，就将其加入黑名单。

  这就要求有人工进行干预。但现在很多大型爬虫产品都有某种形式的黑名单，用于避开某些存在固有问题或者有恶意的站点。还可以用黑名单来避开那些对爬行大惊小怪的站点。[8]

223

注 6：总之，广度优先搜索是个好方法，这样可以更均匀地分配请求，而不是都压到任意一台服务器上去。这样可以帮助机器人将用于一台服务器的资源保持在最低水平。

注 7：在 9.5 节也讨论了请求率的节流问题。

注 8：9.4 节讨论了站点怎样才能避免被爬行，但有些用户拒绝使用这种简单的控制机制，在其站点被爬行时又会变得非常愤怒。

- 模式检测

  文件系统的符号连接和类似的错误配置所造成的环路会遵循某种模式；比如，URL 会随着组件的复制逐渐增加。有些机器人会将具有重复组件的 URL 当作潜在的环路，拒绝爬行带有多于两或三个重复组件的 URL。

  重复并不都是立即出现的（比如，“/subdir/subdir/subdir...”）。有些环路周期可能为 2 或其他间隔，比如“/subdir/images/subdir/images/subdir/images/...”。有些机器人会查找具有几种不同周期的重复模式。

- 内容指纹

  一些更复杂的 Web 爬虫会使用指纹这种更直接的方式来检测重复。使用内容指纹的机器人会获取页面内容中的字节，并计算出一个校验和（checksum）。这个校验和是页面内容的压缩表示形式。如果机器人获取了一个页面，而此页面的校验和它曾经见过，它就不会再去爬行这个页面的链接了——如果机器人以前见过页面的内容，它就已经爬行过页面上的链接了。

  必须对校验和函数进行选择，以求两个不同页面拥有相同校验和的几率非常低。MD5 这样的报文摘要函数就常被用于指纹计算。

  有些 Web 服务器会在传输过程中对页面进行动态的修改，所以有时机器人会在校验和的计算中忽略 Web 页面内容中的某些部分，比如那些嵌入的链接。而且，无论定制了什么页面内容的动态服务器端包含（比如添加日期、访问计数等）都可能会阻碍重复检测。

- 人工监视

  Web 就是一片荒野。勇敢的机器人最终总会陷入一个采用任何技术都无能为力的困境。设计所有产品级机器人时都要有诊断和日志功能，这样人类才能很方便地监视机器人的进展，如果发生了什么不寻常的事情就可以很快收到警告。在某些情况下，愤怒的网民会给你发送一些无礼的邮件来提示你出了问题。

爬行 Web 这样规模庞大的数据集时，好的蜘蛛探测法总是会不断改进其工作的。随着新的资源类型不断加入 Web，它会随着时间的推移构建出一些新的规则，并采纳这些规则。好的规则总是在不断发展之中的。

当受到错误爬虫影响的资源（服务器、网络带宽等）处于可管理状态，或者处于执行爬行工作的人的控制之下（比如在内部站点上）时，很多较小的、更具个性的爬虫就会绕开这些问题。这些爬虫更多的是依赖人类的监视来防止这些问题的发生。

224

9.2 机器人的HTTP

机器人和所有其他 HTTP 客户端程序并没有什么区别。它们也要遵守 HTTP 规范中的规则。发出 HTTP 请求并将自己广播成 HTTP/1.1 客户端的机器人也要使用正确的 HTTP 请求首部。

很多机器人都试图只实现请求它们所查找内容所需的最小 HTTP 集。这会引发一些问题；但短期内这种行为不会发生什么改变。结果就是，很多机器人发出的都是 HTTP/1.0 请求，因为这个协议的要求很少。

9.2.1 识别请求首部

尽管机器人倾向于只支持最小的 HTTP 集，但大部分机器人确实实现并发送了一些识别首部——最值得一提的就是 `User-Agent` 首部。建议机器人实现者们发送一些基本的首部信息，以通知各站点机器人的能力、机器人的标识符，以及它是从何处起源的。

在追踪错误爬虫的所有者，以及向服务器提供机器人所能处理的内容类型时，这些信息都是很有用的。鼓励机器人实现者们使用的基本识别首部包括如下内容。

- `User-Agent`
  将发起请求的机器人名字告知服务器。
- `From`
  提供机器人的用户 / 管理者的 E-mail 地址。[9]
- `Accept`
  告知服务器可以发送哪些媒体类型。[10] 这有助于确保机器人只接收它感兴趣的内容（文本、图片等）。
- `Referer`
  提供包含了当前请求 URL 的文档的 URL。[11]

9.2.2 虚拟主机

225 机器人实现者要支持 `Host` 首部。随着虚拟主机（参见第 5 章）的流行，请求中不

注 9：一种 RFC 822 E-mail 地址格式。

注 10：3.5.2 节列出了所有 `Accept` 相关的首部；机器人可能会发现，如果它们对特定版本感兴趣的话，发送 `Accept-Charset` 之类的首部是很有帮助的。

注 11：有些站点管理者会尝试着记录机器人是如何找到指向其站点内容的链接的，对这些人来说，这个首部非常有用。

包含 Host 首部的话，可能会使机器人将错误的内容与一个特定的 URL 关联起来。因此，HTTP/1.1 要求使用 Host 首部。

在默认情况下，大多数服务器都被配置为提供一个特定的站点。因此，不包含 Host 首部的爬虫向提供两个站点的服务器发起请求时，就像图 9-5 中的站点一样（www.joes-hardware.com 和 www.foo.com），假设默认情况下服务器被配置为提供 www.joes-hardware.com 站点（且不需要 Host 首部），那么，若请求 www.foo.com 上的某个页面，爬虫实际获取的就是 Joe 的五金商店的站点上的内容。更糟糕的是，爬虫会认为来自 Joe 的五金站点上的那些内容是来自 www.foo.com 的。如果带有相对立的政治色彩或其他观点的两个站点是由同一台服务器提供的，你肯定能想象到会有更不幸的局面出现。



图 9-5　发送请求时没有携带 Host 首部，虚拟 docroot 会引发问题的例子

9.2.3　条件请求

鉴于这些机器人的努力程度，尽量减少机器人所要获取内容的数量通常是很有意义的。对因特网搜索引擎机器人来说，需要下载的潜在页面有数十亿，所以，只在内容发生变化时才重新获取内容是很有意义的。

有些机器人实现了条件 HTTP 请求，[12] 它们会对时间戳或实体标签进行比较，看看它们最近获取的版本是否已经升级了。这与 HTTP 缓存查看已获取资源的本地副本是否有效的方法非常相似。更多与缓存对资源本地副本的验证有关的信息请参见第 7 章。

226

注 12：3.5.2 节给出了一个机器人可以实现的条件首部的完整列表。

9.2.4 对响应的处理

很多机器人的兴趣主要在于用简单的 GET 方法来获取所请求的内容，所以，一般不会在处理响应的方式上花费太多时间。但是，使用了某些 HTTP 特性（比如条件请求）的机器人，以及那些想要更好地探索服务器，并与服务器进行交互的机器人则要能够对各种不同类型的 HTTP 响应进行处理。

1. 状态码

总之，机器人至少应该能够处理一些常见的，以及预期的状态码。所有机器人都应该理解 200 OK 和 404 Not Found 这样的状态码。它们还应该能够根据响应的一般类别对它并不十分理解的状态码进行处理。第 3 章的表 3-2 给出了不同状态码的分类及其含义。

有些服务器并不总能返回适当的错误代码，认识到这一点是很重要的。有些服务器甚至会将 HTTP 状态码 200 OK 与描述错误状态的报文主体文本一同返回！很难对此做些什么——只是实现者应该要了解这些情况。

2. 实体

除了 HTTP 首部所嵌的信息之外，机器人也会在实体中查找信息。HTML 元标签，[13] 比如元标签 `http-equiv`，就是内容编写者用于嵌入资源附加信息的一种方式。

服务器可能会为它所处理的内容提供一些首部，标签 `http-equiv` 为内容编写者提供了一种覆盖这些首部的方式：

```
<meta http-equiv="Refresh" content="1;URL=index.html">
```

这个标签会指示接收者处理这个文档时，要当作其 HTTP 响应首部中有一个值为 `1,URL=index.html` 的 `Refresh` HTTP 首部。[14]

有些服务器实际上会在发送 HTML 页面之前先对其内容进行解析，并将 `http-equiv` 指令作为首部包含进去；有些服务器则不会。机器人实现者可能会去扫描 HTML 文
227 档的 HEAD 组件，以查找 `http-equiv` 信息。[15]

注 13：9.4.7 节列出了一些附加的元指令，站点管理员和内容编写者可以通过这些元指令来控制机器人的行为，以及这些机器人对已获取文档所执行的操作。

注 14：有时会将 `Refresh` HTTP 首部作为将用户（或者在这种情况下，就是将机器人）从一个页面重定向到另一个页面的手段。

注 15：根据 HTML 的规范，元标签一定要出现在 HTML 文档的 HEAD 部分。但并不是所有的 HTML 文档都会遵循规范，因此，它们有时也会出现在 HTML 文档的其他区域中。

9.2.5 User-Agent导向

Web 管理者应该记住，会有很多的机器人来访问它们的站点，因此要做好接收机器人请求的准备。很多站点会为不同的用户代理进行内容优化，并尝试着对浏览器类型进行检测，以确保能够支持各种站点特性。这样的话，当实际的 HTTP 客户端根本不是浏览器，而是机器人的时候，站点为机器人提供的就会是出错页面而不是页面内容了。在某些搜索引擎上执行文本搜索，搜索短语“your browser does not support frames”（你的浏览器不支持框架），会生成一个包含那条短语的出错页面列表。

站点管理者应该设计一个处理机器人请求的策略。比如，它们可以为所有其他特性不太丰富的浏览器和机器人开发一些页面，而不是将其内容限定在特定浏览器所支持的范围。至少，管理者应该知道机器人是会访问其站点的，不应该在机器人访问时感到猝不及防。[16]

9.3 行为不当的机器人

不守规矩的机器人会造成很多严重问题。这里列出了一些机器人可能会犯的错误，及其恶劣行为所带来的后果。

- 失控机器人
  机器人发起 HTTP 请求的速度要比在 Web 上冲浪的人类快得多，它们通常都运行在具有快速网络链路的高速计算机上。如果机器人存在编程逻辑错误，或者陷入了环路之中，就可能会向 Web 服务器发出大量的负载——很可能会使服务器过载，并拒绝为任何其他人提供服务。所有的机器人编写者都必须特别小心地设计一些保护措施，以避免失控机器人带来的危害。

- 失效的 URL
  有些机器人会去访问 URL 列表。这些列表可能很老了。如果一个 Web 站点对其内容进行了大量的修改，机器人可能会对大量不存在的 URL 发起请求。这会激怒某些 Web 站点的管理员，他们不喜欢他们的错误日志中充满了对不存在文档的访问请求，也不希望提供出错页面的开销降低其 Web 服务器的处理能力。

- 很长的错误 URL
  由于环路和编程错误的存在，机器人可能会向 Web 站点请求一些很大的、无意义的 URL。如果 URL 足够长的话，就会降低 Web 服务器的性能，使 Web 服务

注 16：如果某站点上有一些不应该让机器人访问的内容，站点管理员该如何控制机器人在其站点上的行为呢？ 9.4 节给出了相关的信息。

器的访问日志杂乱不堪，甚至会使一些比较脆弱的 Web 服务器崩溃。

- *爱打听的机器人*
  有些机器人可能会得到一些指向私有数据的 URL，这样，通过因特网搜索引擎和其他应用程序就可以很方便地访问这些数据了。如果数据的所有者没有主动宣传这些 Web 页面，那么在最好的情况下，他只是会认为机器人的发布行为惹人讨厌，而在最坏的情况下，则会认为这种行为是对隐私的侵犯。[17]

  通常，发生这种情况是由于机器人所跟踪的、指向“私有”内容的超链已经存在了（也就是说，这些内容并不像其所有者认为的那么隐密，或者其所有者忘记删除先前存在的超链了）。偶尔也会因为机器人非常热衷于寻找某站点上的文档而出现这种情况，很可能就是在没有显式超链的情况下去获取某个目录的内容造成的。

  从 Web 上获取大量数据的机器人的实现者们应该清楚，他们的机器人很可能会在某些地方获得敏感的数据——站点的实现者不希望通过因特网能够访问到这些数据。这些敏感数据可能包含密码文件，甚至是信用卡信息。很显然，一旦被指出，就应该有某种机制可以将这些数据丢弃（并从所有搜索索引或归档文件中将其删除），这是非常重要的。现在已知一些恶意使用搜索引擎和归档的用户会利用大型 Web 爬虫来查找内容——有些搜索引擎，比如 Google，[18] 实际上会对它们爬行过的页面进行归档，这样，即使内容被删除了，在一段时间内还是可以找到并访问它。

- *动态网关访问*
  机器人并不总是知道它们访问的是什么内容。机器人可能会获取一个内容来自网关应用程序的 URL。在这种情况下，获取的数据可能会有特殊的目的，计算的开销可能很高。很多 Web 站点管理员并不喜欢那些去请求网关文档的幼稚机器人。

9.4 拒绝机器人访问

机器人社团能够理解机器人访问 Web 站点时可能引发的问题。1994 年，人们提出了一项简单的自愿约束技术，可以将机器人阻挡在不适合它的地方之外，并为网站

注 17：通常，如果某资源可以通过公共因特网获取的话，它很可能会在某处被引用。由于因特网上链路网的存在，很少有资源是真正私有的。

注 18：参见 http://www.google.com 上的搜索结果。已缓存链接就是 Google 爬虫解析并索引过的页面的副本，大多数搜索结果中都会有已缓存链接。

管理员提供了一种能够更好地控制机器人行为的机制。这个标准被称为“拒绝机器人访问标准”，但通常只是根据存储访问控制信息的文件而将其称为 robots.txt。

robots.txt 的思想很简单。所有 Web 服务器都可以在服务器的文档根目录中提供一个可选的、名为 robots.txt 的文件。这个文件包含的信息说明了机器人可以访问服务器的哪些部分。如果机器人遵循这个自愿约束标准，它会在访问那个站点的所有其他资源之前，从 Web 站点请求 robots.txt 文件。例如，图 9-6 中的机器人想要从 Joe 的五金商店下载 http://www.joes-hardware.com/specials/acetylene-torches.html。但在机器人去请求这个页面之前，要先去查看 robots.txt 文件，看看它是否有获取这个页面的权限。在这个例子中，robots.txt 文件并没有拦截机器人，因此机器人获取了这个页面。

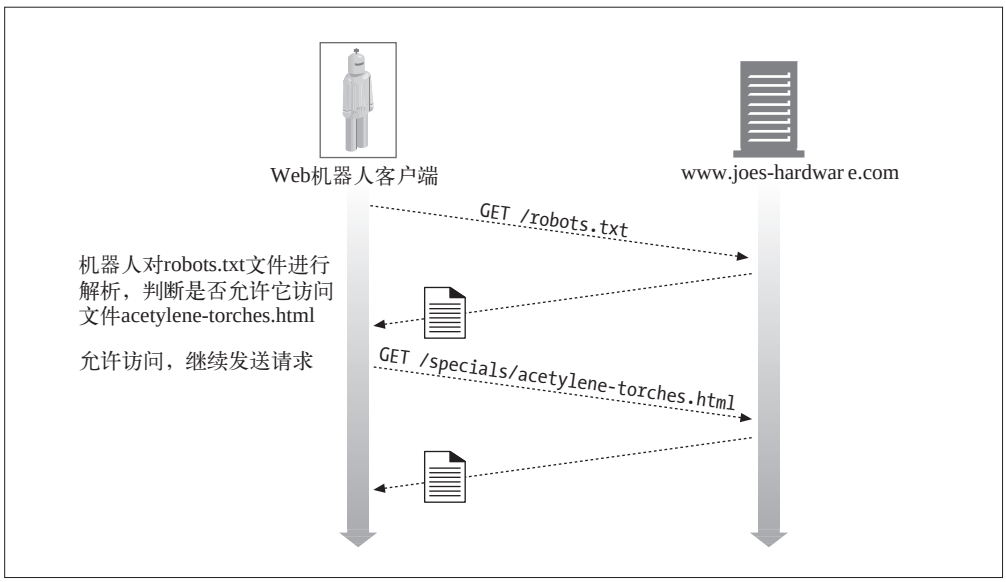


图 9-6　在爬行目标文件之前，先获取 robots.txt，验证是否可以进行访问

9.4.1　拒绝机器人访问标准

拒绝机器人访问标准是一个临时标准。编写本书的时候还没有官方标准机构承认这个标准，不同的厂商实现了这个标准的不同子集。但是，具备一些对机器人访问 Web 站点的管理能力，即使并不完美，也总比一点儿都没有要好，而且大部分主要的生产厂商和搜索引擎爬虫都支持这个拒绝访问标准。

尽管没有很好地定义版本的名称，但拒绝机器人访问标准是有三个版本的。我们采用了表 9-2 列出的版本编号。

表9-2　拒绝机器人访问标准的版本

版　本	标题及描述	日　期
0.0	拒绝机器人标准——Martijn Koster 提出的带有 Disallow（不允许）指令的原始 robots.txt 机制	1994 年 6 月
1.0	控制 Web 机器人的方法——Martijn Koster 提供了额外支持 Allow（允许）的 IETF 草案	1996 年 11 月
2.0	拒绝机器人访问的扩展标准——Sean Conner 提出的扩展标准，包括了正则表达式和定时信息；没有得到广泛的支持	1996 年 11 月

现在大多数机器人采用的都是标准 v0.0 或 v1.0。版本 v2.0 要复杂得多，没有得到广泛的应用。可能永远也不会得到广泛应用。这里我们重点介绍 v1.0 标准，因为它的应用很广泛，而且与 v0.0 完全兼容。

9.4.2　Web站点和robots.txt文件

如果一个 Web 站点有 robots.txt 文件，那么在访问这个 Web 站点上的任意 URL 之前，机器人都必须获取它并对其进行处理。[19] 由主机名和端口号定义的整个 Web 站点上仅有一个 robots.txt 资源。如果这个站点是虚拟主机，每个虚拟的 docroot 都可以有一个不同的 robots.txt 文件，像所有其他文件一样。

通常不能在 Web 站点上单独的子目录中安装“本地”robots.txt 文件。网管要负责创建一个聚合型 robots.txt 文件，用以描述 Web 站点上所有内容的拒绝访问规则。

1. 获取robots.txt

机器人会用 HTTP 的 GET 方法来获取 robots.txt 资源，就像获取 Web 服务器上所有其他资源一样。如果有 robots.txt 文件的话，服务器会将其放在一个 text/plain 主体中返回。如果服务器以 404 Not Found HTTP 状态码进行响应，机器人就可以认为这个服务器上没有机器人访问限制，它可以请求任意的文件。

机器人应该在 From 首部和 User-Agent 首部中传输标识信息，以帮助站点管理者对机器人的访问进行跟踪，并在站点管理者要查询，或投诉的机器人事件中提供一些联系信息。下面是一个来自商业 Web 机器人的 HTTP 爬虫请求实例：

```
GET /robots.txt HTTP/1.0
Host: www.joes-hardware.com
User-Agent: Slurp/2.0
Date: Wed Oct 3 20:22:48 EST 2001
```

注 19：尽管我们说的是 robots.txt 文件，但 robots.txt 资源并不一定要严格地位于文件系统中。比如，可以由一个网关应用程序动态地生成这个 robots.txt 资源。

2. 响应码

很多 Web 站点都没有 robots.txt 资源，但机器人并不知道这一点。它必须尝试着从每个站点上获取 robots.txt 资源。机器人会根据对 robots.txt 检索的结果采取不同的行动。

- 如果服务器以一个成功状态（HTTP 状态码 2XX）为响应，机器人就必须对内容进行解析，并使用排斥规则从那个站点上获取内容。
- 如果服务器响应说明资源并不存在（HTTP 状态码 404），机器人就可以认为服务器没有激活任何排斥规则，对此站点的访问不受 robots.txt 的限制。
- 如果服务器响应说明有访问限制（HTTP 状态码 401 或 403），机器人就应该认为对此站点的访问是完全受限的。
- 如果请求尝试的结果是临时故障（HTTP 状态码 503），机器人就应该推迟对此站点的访问，直到可以获取该资源为止。
- 如果服务器响应说明是重定向（HTTP 状态码 3XX），机器人就应该跟着重定向，直到找到资源为止。

9.4.3 robots.txt文件的格式

robots.txt 文件采用了非常简单的，面向行的语法。robots.txt 文件中有三种类型的行：空行、注释行和规则行。规则行看起来就像 HTTP 首部（`<Field>:<value>`）一样，用于模式匹配。比如：

```
# this robots.txt file allows Slurp & Webcrawler to crawl
# the public parts of our site, but no other robots...

User-Agent: slurp
User-Agent: webcrawler
Disallow: /private

User-Agent: *
Disallow:
```

robots.txt 文件中的行可以从逻辑上划分成“记录”。每条记录都为一组特定的机器人描述了一组排斥规则。通过这种方式，可以为不同的机器人使用不同的排斥规则。

每条记录中都包含了一组规则行，由一个空行或文件结束符终止。记录以一个或多个 `User-Agent` 行开始，说明哪些机器人会受此记录的影响，后面跟着一些 `Disallow` 和 `Allow` 行，用来说明这些机器人可以访问哪些 URL。[20]

注 20：出于实际应用的原因，机器人软件应该很强壮，可以灵活地使用行结束符。应该支持 CR、LF 和 CRLF。

前面的例子显示了一个 robots.txt 文件，这个文件允许机器人 Slurp 和 Webcrawler 访问除了 private 子目录下那些文件之外所有的文件。这个文件还会阻止所有其他机器人访问那个站点上的任何内容。

我们来看看 User-Agent、Disallow 和 Allow 行。

1. User-Agent行

每个机器人记录都以一个或多个下列形式的 User-Agent 行开始：

```
User-Agent: <robot-name>
```

或

232

```
User-Agent: *
```

在机器人 HTTP GET 请求的 User-Agent 首部中发送（由机器人实现者选择的）机器人名。

机器人处理 robots.txt 文件时，它所遵循的记录必须符合下列规则之一：

- 第一个 <robot-name> 是机器人名的大小写无关的子字符串；
- 第一个 <robot-name> 为“*”。

如果机器人无法找到与其名字相匹配的 User-Agent 行，而且也无法找到通配的 User-Agent:* 行，就是没有记录与之匹配，访问不受限。

由于机器人名是与大小写无关的子字符串进行匹配，所以要小心不要匹配错了。比如，User-Agent:bot 就与名为 Bot、Robot、Bottom-Feeder、Spambot 和 Dont-Bother-Me 的所有机器人相匹配。

2. Disallow和Allow行

Disallow 和 Allow 行紧跟在机器人排斥记录的 User-Agent 行之后。用来说明显式禁止或显式允许特定机器人使用哪些 URL 路径。

机器人必须将期望访问的 URL 按序与排斥记录中所有的 Disallow 和 Allow 规则进行匹配。使用找到的第一个匹配项。如果没有找到匹配项，就说明允许使用这个 URL。[21]

注 21：总是应该允许访问 robots.txt 的 URL，它一定不能出现在 Allow/Disallow 规则中。

要使 Allow/Disallow 行与一个 URL 相匹配，规则路径就必须是 URL 路径大小写相关的前缀。例如，Disallow: /tmp 就和下面所有的 URL 相匹配：

```
http://www.joes-hardware.com/tmp
http://www.joes-hardware.com/tmp/
http://www.joes-hardware.com/tmp/pliers.html
http://www.joes-hardware.com/tmpspc/stuff.txt
```

3. Disallow/Allow前缀匹配

下面是 Disallow/Allow 前缀匹配的一些细节。

- Disallow 和 Allow 规则要求大小写相关的前缀匹配。（与 User-Agent 行不同）这里的星号没什么特殊的含义，但空字符串可以起到通配符的效果。
- 在进行比较之前，要将规则路径或 URL 路径中所有“被转义”的字符（%XX）都反转为字节（除了正斜杠 %2F 之外，它必须严格匹配）。
- 如果规则路径为空字符串，就与所有内容都匹配。

表 9-3 列出了几个在规则路径和 URL 路径间进行匹配的例子。

233

表9-3　robots.txt路径匹配示例

规则路径	URL路径	匹配吗?	注　　释
/tmp	/tmp	✓	规则路径 == URL 路径
/tmp	/tmpfile.html	✓	规则路径是 URL 路径的前缀
/tmp	/tmp/a.html	✓	规则路径是 URL 路径的前缀
/tmp/	/tmp	×	/tmp/ 不是 /tmp 的前缀
	README.TXT	✓	空的规则路径匹配于所有的路径
/~fred/hi.html	/%7Efred/hi.html	✓	将 %7E 与 ~ 同等对待
/%7Efred/hi.html	/~fred/hi.html	✓	将 %7E 与 ~ 同等对待
/%7efred/hi.html	/%7Efred/hi.html	✓	转义符是大小写无关的
/~fred/hi.html	~fred%2Fhi.html	×	%2F 是一个斜杠，但斜杠是种特殊情况，必须完全匹配

前缀匹配通常都能很好地工作，但有几种情况下它的表达力却不够强。如果你希望无论使用什么路径前缀，都不允许爬行一些特别的子目录，那 robots.txt 是无能为力的。比如，你可能希望禁止在用于 RCS 版本控制的子目录中爬行。除了将到达各 RCS 子目录的每条路径都分别枚举出来之外，1.0 版的 robots.txt 方案无法提供此功能。

9.4.4 其他有关robots.txt的知识

解析 robots.txt 文件时还需遵循其他一些规则。

- 随着规范的发展，robots.txt 文件中可能会包含除了 `User-Agent`、`Disallow` 和 `Allow` 之外的其他字段。机器人应该将所有它不理解的字段都忽略掉。
- 为了实现后向兼容，不能在中间断行。
- 注释可以出现在文件的任何地方；注释包括可选的空格，以及后面的注释符（#）、注释符后面的注释，直到行结束符为止。
- 0.0 版的拒绝机器人访问标准并不支持 `Allow` 行。有些机器人只实现了 0.0 版的规范，因此会忽略 `Allow` 行。在这种情况下，机器人的行为会比较保守，有些允许访问的 URL 它也不去获取。

9.4.5 缓存和robots.txt的过期

如果一个机器人在每次访问文件之前都要重新获取 robots.txt 文件，Web 服务器上的负载就会加倍，机器人的效率也会降低。机器人使用的替代方法是，它会周期性地获取 robots.txt 文件，并将得到的文件缓存起来。机器人会使用这个 robots.txt 文
234 件的缓存副本，直到其过期为止。原始服务器和机器人都会使用标准的 HTTP 缓存控制机制来控制 robots.txt 文件的缓存。机器人应该留意 HTTP 响应中的 `Cache-Control` 和 `Expires` 首部。[22]

现在很多产品级爬虫都不是 HTTP/1.1 的客户端；网管应该意识到这些爬虫不一定能够理解那些为 robots.txt 资源提供的缓存指令。

如果没有提供 `Cache-Control` 指令，规范草案允许将其缓存 7 天。但实际上，这个时间通常太长了。不了解 robots.txt 文件的 Web 服务器管理员通常会在响应机器人的访问时创建一个新的文件，但如果将缺乏信息的 robots.txt 文件缓存一周，新创建的 robots.txt 文件就没什么效果了，站点管理员会责怪机器人管理员没有遵守拒绝机器人访问标准。[23]

9.4.6 拒绝机器人访问的Perl代码

有几个公共的 Perl 库可以用来与 robots.txt 文件进行交互。CPAN 公共 Perl 文档中的 `WWW::RobotsRules` 模块就是一个这样的例子。

注 22：更多有关缓存指令处理方面的内容请参见 7.8 节。

注 23：有几种大型的 Web 爬虫，如果它们在 Web 上勤奋爬行的话，每天都会重新获取 robots.txt。

将已解析的 robots.txt 文件保存在 WWW::RobotRules 对象中，这个对象提供了一些方法，可以用于查看是否禁止对某指定 URL 进行访问。同一个 WWW::RobotRules 可以用于解析多个 robots.txt 文件。

下面是 WWW::RobotRules API 的一些主要方法。

- 创建 RobotRules 对象

```
$rules = WWW::RobotRules->new($robot_name);
```

- 装载 robots.txt 文件

```
$rules->parse($url, $content, $fresh_until);
```

- 查看站点 URL 是否可获取

```
$can_fetch = $rules->allowed($url);
```

下面这个短小的 Perl 程序说明了 WWW::RobotRules 的用法：

```
require WWW::RobotRules;

# Create the RobotRules object, naming the robot "SuperRobot"
my $robotsrules = new WWW::RobotRules 'SuperRobot/1.0';
use LWP::Simple qw(get);

# Get and parse the robots.txt file for Joe's Hardware, accumulating
# the rules
$url = "http://www.joes-hardware.com/robots.txt";
my $robots_txt = get $url;
$robotsrules->parse($url, $robots_txt);

# Get and parse the robots.txt file for Mary's Antiques, accumulating
# the rules
$url = "http://www.mary's antiques.com/robots.txt";
my $robots_txt = get $url;
$robotsrules->parse($url, $robots_txt);

# Now RobotRules contains the set of robot exclusion rules for several
# different sites. It keeps them all separate. Now we can use RobotRules
# to test if a robot is allowed to access various URLs.
if ($robotsrules->allowed($some_target_url))
{
    $c = get $url;
    ...
}
```


下面是 www.marys-antiques.com 的假想 robots.txt 文件：

```
##############################################################################
# This is the robots.txt file for Mary's Antiques web site
##############################################################################
```

```
# Keep Suzy's robot out of all the dynamic URLs because it doesn't
# understand them, and out of all the private data, except for the
# small section Mary has reserved on the site for Suzy.

User-Agent: Suzy-Spider
Disallow: /dynamic
Allow: /private/suzy-stuff
Disallow: /private

# The Furniture-Finder robot was specially designed to understand
# Mary's antique store's furniture inventory program, so let it
# crawl that resource, but keep it out of all the other dynamic
# resources and out of all the private data.

User-Agent: Furniture-Finder
Allow: /dynamic/check-inventory
Disallow: /dynamic
Disallow: /private

# Keep everyone else out of the dynamic gateways and private data.

User-Agent: *
Disallow: /dynamic
Disallow: /private
```

这个 robots.txt 文件中包含了一条机器人 SuzySpider 的记录，一条机器人 FurnitureFinder 的记录，以及一条用于所有其他机器人的默认记录。每条记录都对不同的机器人使用了一组不同的访问策略。

- SuzySpider 的排斥记录不允许机器人爬行以 /dynamic 开头的商店库存网关 URL，以及在为 Suzy 保留的区域之外的其他私有用户数据。
236
- FurnitureFinder 机器人的记录允许机器人爬行家具库存网关 URL。这个机器人可能能够理解 Mary 的网关格式和规则。
- 其他机器人都不能访问所有的动态和私有 Web 页面，但它们可以爬行其余的 URL。

表 9-4 列出了几个机器人实例，这几个机器人具有不同的 Mary 古董网站访问权限。

表9-4　Mary古董网站的机器人访问权限

URL	SuzySpider	FurnitureFinder	NosyBot
http://www.marys-antiques.com/	✓	✓	✓
http://www.marys-antiques.com/index.html	✓	✓	✓

（续）

URL	SuzySpider	FurnitureFinder	NosyBot
http://www.marys-antiques.com/private/payroll.xls	×	×	×
http://www.marys-antiques.com/private/suzy-stuff/taxes.txt	✓	×	×
http://www.marys-antiques.com/dynamic/buy-stuff?id=3546	×	×	×
http://www.marys-antiques.com/dynamic/check-inventory?kitchen	×	✓	×

9.4.7 HTML的robot-control元标签

robots.txt 文件允许站点管理员将机器人排除在 Web 站点的部分或全部内容之外。robots.txt 文件的一个缺点就是它是 Web 站点管理员，而不是各部分内容的作者所有的。

HTML 页面的作者有一种更直接的方式可以限制机器人访问那些独立的页面。他们可以直接在 HMTL 文档中添加 robot-control 标签。遵循 robot-control HTML 标签规则的机器人仍然可以获取文档，但如果其中有机器人排斥标签，它们就会忽略这些文档。比如，因特网搜索引擎机器人就不会在其搜索索引中包含这个目录了。和 robots.txt 标准一样，鼓励但并不强制使用这个标签。

机器人排斥标签是以如下形式，通过 HTML 的 META 标签来实现的：

```
<META NAME="ROBOTS" CONTENT=directive-list>
```

1. 机器人的META指令

机器人 META 指令有几种不同的类型，而且随着时间的推移，以及搜索引擎及机器人对其行为和特性集的扩展，很可能还会添加一些新的指令。最常用的两个机器人 META 指令如下所示。

- NOINDEX
  告诉机器人不要对页面的内容进行处理，忽略文档（也就是说，不要在任何索引或数据库中包含此内容）。

```
<META NAME="ROBOTS" CONTENT="NOINDEX">
```

- NOFOLLOW
  告诉机器人不要爬行这个页面的任何外连链接。

```
<META NAME="ROBOTS" CONTENT="NOFOLLOW">
```

除了 NOINDEX 和 NOFOLLOW 之外，还有相对应的 INDEX 指令、FOLLOW 指令、NOARCHIVE 指令以及 ALL 和 NONE 指令。下面对这些机器人 META 标签指令进行了总结。

- INDEX
  告诉机器人它可以对页面的内容进行索引。

- FOLLOW
  告诉机器人它可以爬行页面上的任何外连链接。

- NOARCHIVE
  告诉机器人不应该缓存这个页面的本地副本。[24]

- ALL
  等价于 INDEX、FOLLOW。

- NONE
  等价于 NOINDEX、NOFOLLOW。

与所有 HTML 的 META 标签类似，机器人 META 标签必须出现在 HTML 页面的 HEAD 区域中：

```
<html>
<head>
    <meta name="robots" content="noindex,nofollow">
    <title>...</title>
</head>
<body>
    ...
</body>
</html>
```

注意，标签的名称 robots 和内容都是大小写无关的。

很显然，不能发出一些会产生冲突或重复的指令，比如：

```
<meta name="robots" content="INDEX,NOINDEX,NOFOLLOW,FOLLOW,FOLLOW">
```

注 24：那些运行 Google 搜索引擎的人引入这个 META 标签，是为了向网管提供一种不允许 Google 提供其内容缓存页面的手段。此标签还可以与 META NAME="googlebot" 一起使用。

这种指令的行为很可能是未定义的，肯定会随机器人实现的不同而有所不同。

2. 搜索引擎的META标签

我们刚刚讨论了机器人的 META 标签，可以用来控制 Web 机器人的爬行和索引行为。所有的机器人 META 标签中都包含了 name="robots" 属性。

还有很多其他类型的 META 标签可用，包括表 9-5 所示的各种标签。对内容索引型搜索引擎机器人来说，DESCRIPTION 和 KEYWORDS META 标签都非常有用。

表9-5 其他META标签指令

name=	content=	描 述
DESCRIPTION	< 文本 >	允许作者为 Web 页面定义一个短小的文本摘要。很多搜索引擎都会查看 META DESCRIPTION 标签，允许页面作者指定一些短小的摘要来描述其 Web 页面 <meta name="description" content="Welcome to Mary's Antiques Web site">
KEYWORDS	< 逗号列表 >	关联一个由逗号分隔的 Web 页面描述词列表，为关键字搜索提供帮助 <meta name="keywords" content="antiques,mary,furniture,restoration">
REVISIT-AFTER[25]	< 天数 >	告诉机器人或搜索引擎应该在指定天数之后重访页面，估计那时候页面可能会发生变化 <meta name="revisit-after" content="10 days">

9.5 机器人的规范

1993 年，Web 机器人社会的先驱 Martijn Koster 为 Web 机器人的编写者们编写了一个指南列表。有些建议已经过时了，但有很多建议仍然非常有用。在 http://www.robotstxt.org/wc/guidelines.html 上可以找到 Martijn 的原始论文“Guidelines for Robot Writers”。

表 9-6 是为机器人设计者和操作人员提供的现代更新，这些更新的建议主要还是建立在原始列表的思想和内容之上的。大部分指南都是针对万维网机器人提出的；但它们同样适用于较小规模的爬虫。

注 25：这个指令很可能没有得到广泛的支持。

表9-6　Web机器人操作员指南

操作指南	描　　述
(1) 识别	
识别你的机器人	用 HTTP 的 `User-Agent` 字段将机器人的名字告诉 Web 服务器。这样可以帮助管理员理解机器人所做的事情。有些机器人还会在 `User-Agent` 首部包含一个描述机器人目的和策略的 URL
识别你的机器	确保机器人是从一台带有 DNS 条目的机器上运行的，这样 Web 站点才能够将机器人的 IP 地址反向 DNS 为主机名。这有助于管理者识别出对机器人负责的组织
识别联络人	用 HTTP 的 `From` 字段提供一个联络的 E-mail 地址
(2) 操作	
保持警惕	机器人可能会惹一些麻烦或引发一些抱怨。其中一些是由那些行为有偏差的机器人造成的。一定要小心，注意保持机器人的正常行为。如果机器人要全天候运行，就要格外小心。需要有操作人员不间断地对机器人进行监视，直到它有了丰富的经验为止
做好准备	开始一次重要的机器人之旅时，一定要通知你所在的组织。你的组织可能要观测网络带宽的耗费，作好应对各种公共查询的准备
监视并记录日志	机器人应该装备有丰富的诊断和日志记录工具，这样才能记录进展、识别所有的机器人陷阱，进行完整性检查看看工作是否正常。监视并记录机器人行为的重要性怎么强调也不过分。问题和抱怨总是会有的，对爬虫行为的详细记录，有助于机器人操作者回溯所发生的事情。不管是为了调试出错的 Web 爬虫，还是为了在不合理的投诉面前为其行为进行辩护，监视和记录工作都是非常重要的
学习并适应	在每次爬行中你都会学到新的东西。要让机器人逐步适应，这样，它在每次爬行之后都会有所进步，并能避开一些常见的陷阱
(3) 约束自己的行为	
对 URL 进行过滤	如果一个 URL 指向的好像是你不理解或不感兴趣的数据，你可能会希望跳过它。比如，以 .Z、.gz、.tar 或者 .zip 结尾的 URL 很可能是压缩文件或归档文件。以 .exe 结尾的 URL 可能就是程序。以 .gif、.tif、.jpg 结尾的 URL 很可能是图片。要确保你得到的就是你想要的
过滤动态 URL	通常，机器人不会想去爬行来自动态网关的内容。机器人不知道应该如何正确地格式化查询请求，并将其发送给网关，而它得到的结果也很可能是错误的或临时的。如果一个 URL 中包含了 cgi，或者有一个“?”，机器人可能就不会去爬行这个 URL 了
对 `Accept` 首部进行过滤	机器人应该用 HTTP 的 `Accept` 首部来告诉服务器它能够理解哪种内容
遵循 robots.txt	机器人应该接受站点上 robots.txt 的控制

（续）

操作指南	描　述
制约自己	机器人应该记录访问每个站点的次数以及访问的时间，并通过这些信息来确保它没有太频繁地访问某个站点。机器人访问站点的频率高于几分钟一次时，管理员就要起疑心了。机器人每隔几秒钟就访问一次站点时，有些管理员就会生气了。机器人尽可能频繁地去访问一个站点，将所有其他流量都拒之门外时，管理员就会暴怒起来。 总之，应该限制机器人，使其每分钟最多只发送几条请求，并确保每条请求之间有几秒钟的间隔。还应该限制对站点的访问总次数，以防止环路的出现
(4) 容忍存在环路、重复和其他问题	
处理所有返回代码	必须做好处理所有 HTTP 状态码的准备，包括各种重定向和错误码。还应该对这些代码进行记录和监视。如果某站点出现大量不成功的结果，就应该对其进行调查。可能是很多 URL 过期了，或者服务器拒绝向机器人提供这些文档
规范 URL	试着将所有 URL 都转化为标准形式来消除常见的别名
积极地避免环路的出现	努力地检测并避免环路的出现。将操纵爬虫的过程当作一个反馈回路。应该将问题的结果和解决方法回馈到下一次爬行中，使爬虫在每次迭代之后都能表现得更好
监视陷阱	有些环路是故意造成的恶意环路。这些环路可能很难检测。有的站点会带有一些怪异的 URL，要监视对这类站点进行的大量访问。这种情况可能就是陷阱
维护一个黑名单	找到陷阱、环路、故障站点和不希望机器人访问的站点时，要将其加入一个黑名单，不要再次访问这些站点
(5) 可扩展性	
了解所需空间	事先通过数学计算明确你要解决的问题规模有多大。你可能会对应用程序完成一项机器人任务所需的内存规模感到非常吃惊，这是由 Web 庞大的规模造成的
了解所需带宽	了解你有多少网络带宽可用，以及在要求的时间内完成机器人任务所需的带宽大小。监视网络带宽的实际使用情况。你很可能会发现输出带宽（请求）要比输入带宽（响应）小得多。通过对网络使用情况的监视，可能还会找到一些方法来更好地优化你的机器人，通过更好地使用其 TCP 连接更好地利用网络带宽 [26]
了解所需的时间	了解机器人完成其任务所需花费的时间，检查这个进度是否与自己的估计相符。如果机器人的耗费与自己的估计相去甚远，可能就会有问题，需要进行调查

注 26：更多有关 TCP 性能优化的内容请参见第 4 章。

（续）

操作指南	描　述
分而治之	对大规模的爬行来说，很可能需要使用更多的硬件来完成这项工作，可以使用带有多个网卡的大型多处理器服务器，也可以使用多台较小的计算机共同配合工作
(6) 可靠性	
彻底测试	在将机器人放出去之前，要对其进行彻底的内部测试。作好非现场测试准备时，要先进行几次小型的处女航。收集大量结果并对性能和内存使用情况进行分析，估计一下它们会怎样累积成较大问题
检查点	所有严谨的机器人都要保存其进展的快照，出现故障时可以从那里重新开始。故障总是存在的：你会发现一些软件的 bug，硬件也会出故障。大规模机器人不能在每次出现这种情况时都从头开始。一开始就要设计检查点 / 重启机制
故障恢复	预测故障的发生，对机器人进行设计，使其能够在发生故障时继续工作
(7) 公共关系	
做好准备	机器人可能会让很多人感到困惑。要作好快速响应其询问的准备。制定一个 Web 页面政策声明，对机器人进行描述，其中包括创建 robots.txt 文件的详细指南
充分理解	有些与你联系，讨论机器人问题的人是了解情况并赞成的，有些人则很幼稚。少数人会异常愤怒。有些人看起来好像都要发疯了。去争辩机器人的努力有多么重要通常是没什么效果的。向他们解释拒绝机器人访问标准，如果他们仍然很不高兴，就立即将投诉者的 URL 从爬行列表中删除，并将其加入黑名单
积极响应	大多数不满意的网管都只是不太了解机器人。如果你能够进行迅速且专业的响应，90% 的投诉都会很快消失。另一方面，如果你等好几天才响应，而机器人在继续访问这个站点，你面对的就将是一个非常愤怒的对手

241

9.6　搜索引擎

得到最广泛使用的 Web 机器人都是因特网搜索引擎。因特网搜索引擎可以帮助用户找到世界范围内涉及任意主题的文档。

现在 Web 上很多最流行的站点都是搜索引擎。很多 Web 用户将其作为起始点，它们会为用户提供宝贵的服务，帮助用户找到他们感兴趣的信息。

Web 爬虫为因特网搜索引擎提供信息，它们获取 Web 上的文档，并允许搜索引擎创建与本书后面的索引类似的索引，用以说明哪些文档中有哪些词存在。搜索引擎是 Web 机器人的主要来源——让我们来快速了解一下它们是如何工作的。

9.6.1 大格局

Web 发展的初期，搜索引擎就是一些相当简单的数据库，可以帮助用户在 Web 上定位文档。现在，Web 上有数十亿可供访问的页面，搜索引擎已经成为因特网用户查找信息不可缺少的工具。它们在不断地发展，以应对 Web 庞大的规模，因此，现在已经变得相当复杂了。

面对数十亿的 Web 页面，和数百万要查找信息的用户，搜索引擎要用复杂的爬虫来获取这数十亿 Web 页面，还要使用复杂的查询引擎来处理数百万用户产生的查询负荷。

我们来考虑一下产品级 Web 爬虫的任务，它要获取搜索索引所需的页面，它要发出数十亿条 HTTP 请求。如果每条请求都要花半秒钟的时间（对有些服务器来说可能慢了，对另一些服务器来说可能快了[27]），（对十亿份文件来说）就要花费：

0.5 秒 ×（100 0000 000）/（60 秒 / 天）×（60 分 / 小时）×（24 小时 / 天）

如果请求是连续发出的，结果差不多是 5700 天！很显然，大型爬虫得更聪明一些，要对请求进行并行处理，并使用大量机器来完成这项任务。但由于其规模庞大，爬行整个 Web 仍然是件十分艰巨的任务。

9.6.2 现代搜索引擎结构

现在的搜索引擎都构建了一些名为“全文索引”的复杂本地数据库，装载了全世界的 Web 页面，以及这些页面所包含的内容。这些索引就像 Web 上所有文档的卡片目录一样。

242

搜索引擎爬虫会搜集 Web 页面，把它们带回家，并将其添加到全文索引中去。同时，搜索引擎用户会通过 HotBot（http://www.hotbot.com）或 Google（http://www.google.com）这样的 Web 搜索网关对全文索引进行查询。Web 页面总是在不断地发生变化，而且爬行一大块 Web 要花费很长的时间，所以全文索引充其量也就是 Web 的一个快照。

现代搜索引擎的高层结构如图 9-7 所示。

9.6.3 全文索引

全文索引就是一个数据库，给它一个单词，它可以立即提供包含那个单词的所有文档。创建了索引之后，就不需要对文档自身进行扫描了。

注 27：这取决于服务器的资源、客户端的机器人，以及两者之间的网络状况。

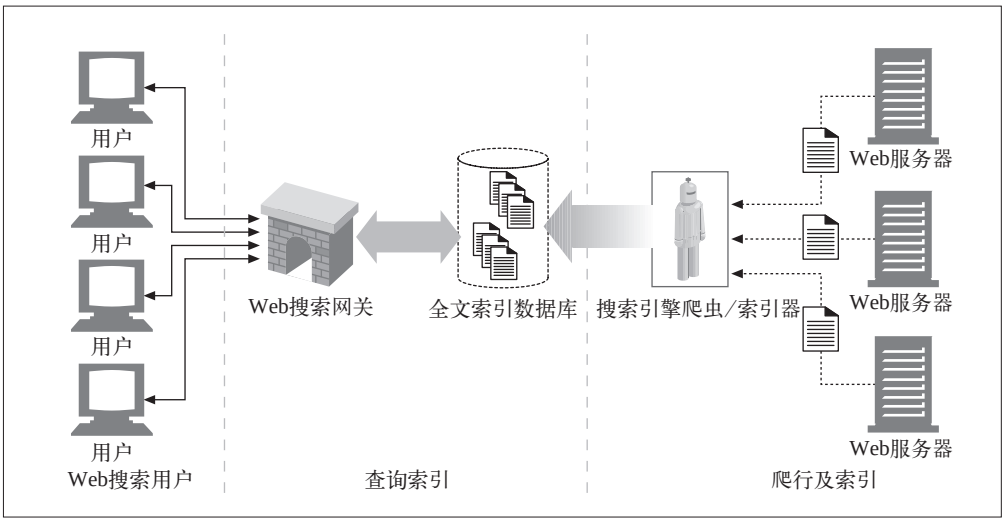


图 9-7　产品级搜索引擎中包含了一些协作的爬虫和查询网关

图 9-8 显示了三份文档和相应的全文索引。全文索引列出了包含每个单词的文档。

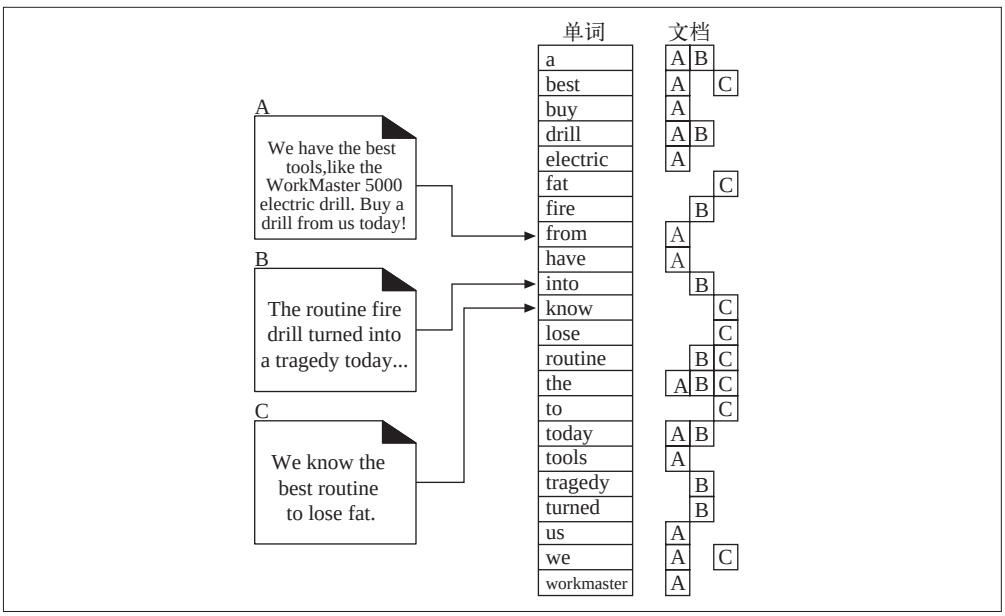


图 9-8　三份文档和一个全文索引

比如：

- 单词“a”位于文档 A 和 B 中；

- 单词“best”位于文档 A 和 C 中；
- 单词“drill”位于文档 A 和 B 中；
- 单词“routine”位于文档 B 和 C 中；
- 单词“the”位于所有的三份文档 A、B 和 C 中。

9.6.4 发布查询请求

用户向 Web 搜索引擎网关发布一条请求时，会填写一个 HTML 表单，他的浏览器会用一个 HTTP GET 或 POST 请求将这个表单发送给网关。网关程序对搜索请求进行解析，并将 Web UI 查询转换成搜索全文索引所需的表达式。[28]

图 9-9 显示了一条对 www.joes-hardware.com 站点的简单用户查询。用户在搜索框表单中输入 drills，然后浏览器就会将这个动作转换成一条在 URL 中包含请求参数的 GET 请求。[29]Joe 的五金商店的 Web 服务器收到这条请求，并将其转发给其搜索网关应用程序，这个程序会将文档的结果列表返回给 Web 服务器，然后 Web 服务器又会将这些结果转换成 HTML 页面提供给用户。

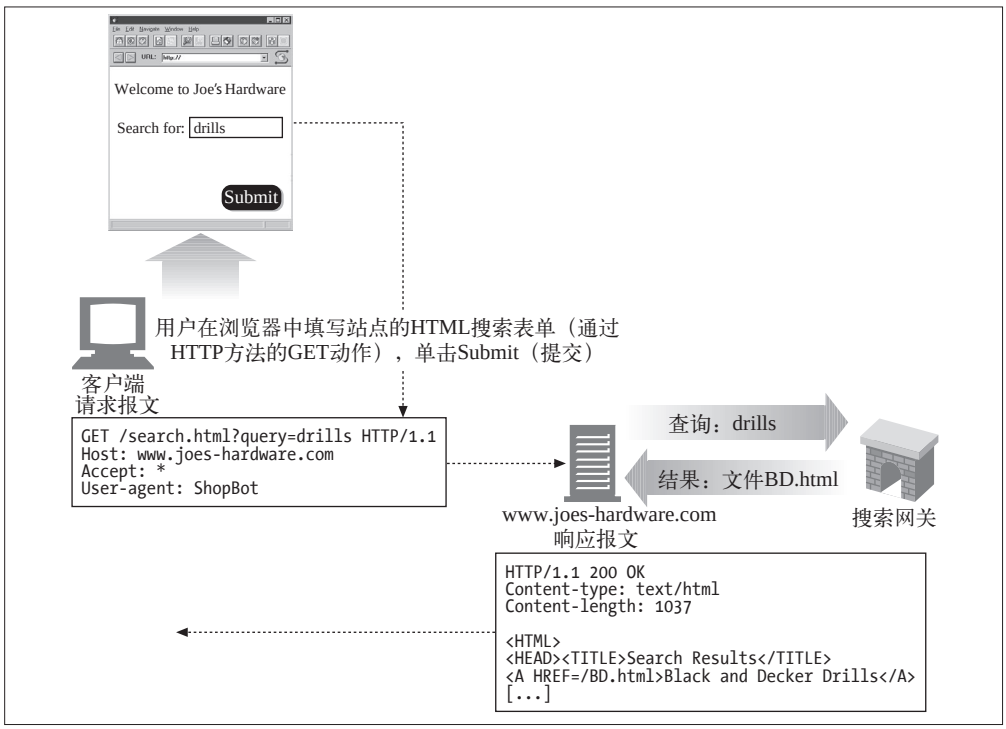


图 9-9 搜索查询请求的实例

注 28：传送这条请求的方法与所使用的搜索策略有关。

注 29：2.2.6 节讨论了 URL 中查询参数的常见用法。

9.6.5 对结果进行排序，并提供查询结果

一旦搜索引擎通过其索引得到了查询结果，网关应用程序会获取结果，并将其拼成
244 结果页面提供给终端用户。

很多 Web 页面都可能包含任意指定的单词，所以搜索引擎采用了一些很聪明的算法，尝试着对结果进行排名。比如，在图 9-8 中，单词 best 出现在很多文档中；为了将相关度最高的结果提供给用户，搜索引擎要知道应该按照什么顺序来提供结果列表中的文档。这被称为*相关性排名*（relevancy ranking）——这是对一系列搜索结果的评分和排序处理。

为了更好地辅助这一进程，在爬行 Web 的过程中都会进行数据统计。比如，对指向指定页面的链接进行计数有助于判断其流行程度，还可以用此信息来衡量提供结果的顺序。算法、爬行中获取的辅助信息以及搜索引擎所使用的其他技巧都是保守最森严的秘密。

9.6.6 欺诈

在搜索请求得到的前几个结果中没有看到自己想要查找的内容时，用户通常会感到很沮丧，因此，查找站点时搜索结果的顺序是很重要的。在搜索网管们认为能够最
245 好地描述其站点功能的单词时，会有众多因素激励着这些网管，努力使其站点排在靠近结果顶端的位置上，尤其是那些依赖于用户找到它们，并使用其服务的商业站点。

这种对较好排列位置的期待引发了很多对搜索系统的博弈，也在搜索引擎的实现者和那些想方设法要将其站点列在突出位置的人之间引发了持久的拉锯战。很多网管都列出了无数关键字（有些是毫不相关的），使用一些假冒页面，或者采用*欺诈*（spoof）行为——甚至会用网关应用程序来生成一些在某些特定单词上可以更好地欺骗搜索引擎相关性算法的假冒页面。

这么做的结果就是，搜索引擎和机器人实现者们要不断地修改相关性算法，以便更有效地抓住这些欺诈者。

9.7 更多信息

更多有关 Web 客户端的信息，请参见下列资料。

- http://www.robotstxt.org/wc/robots.html
  Web 机器人页面——机器人开发者所需的资源，包括因特网机器人的登记注册。

- http://www.searchengineworld.com
  搜索引擎世界——搜索引擎和机器人的相关资源。

- http://www.searchtools.com
  Web 站点和内部网络的搜索工具——搜索工具和机器人的相关资源。

- http://search.cpan.org/doc/ILYAZ/perl_ste/WWW/RobotRules.pm
  RobotRules 的 Perl 语言资源。

- http://www.conman.org/people/spc/robots2.html
  拒绝机器人访问的扩展标准。

- *Managing Gigabytes: Compressing and Indexing Documents and Images*（《海量数据管理——文档和图像的压缩与索引》）
  Witten, I.、Moffat, A. 和 Bell, T. 编写，Morgan Kaufmann 公司出版。 246

第10章

HTTP-NG

在这本书即将完稿的时候，HTTP 正在庆祝它的第十个生日。这十年是这个因特网协议成就辉煌的十年。现在，世界上绝大多数的数字流量都是由 HTTP 传输的。

但随着 HTTP 迈入青少年时期，它也面临着一些挑战。从某些方面来说，HTTP 应用的步伐已经超越了其设计。现在，人们将 HTTP 作为各种不同应用程序的基础，并将其运行在很多不同的联网技术之上。

本章对 HTTP 未来的一些发展趋势和所要面临的挑战，以及被称为 HTTP-NG 的下一代 HTTP 结构方案进行了介绍。尽管 HTTP-NG 工作组已经解散了，而且看起来也不太可能得到快速的应用，但它还是给出了 HTTP 未来一些潜在的发展方向。

10.1 HTTP发展中存在的问题

HTTP 最初被设想为一种简单的技术，用于访问分布式信息服务器上链接的多媒体内容。但在过去的十年中，HTTP 及其衍生产品起到了更为广泛的作用。

HTTP/1.1 现在提供了可以追踪文档版本的标记和指纹，提供了一些方法来支持文档的上传以及与可编程网关之间的交互，还提供对多语言内容、安全及认证功能、降低流量的缓存功能、减小时延的管道功能、降低启动时间提高带宽使用效率的持久连接，以及用来进行部分更新的访问范围功能的支持。HTTP 的扩展及衍生产品具有更为广泛的功能，可以提供对文档发布、应用程序服务、任意的消息服务、视频流以及无线多媒体访问的支持。HTTP 正在成为分布式多媒体应用程序的“操作系统”。

247

尽管 HTTP/1.1 的设计经过了充分的考量，但随着 HTTP 被越来越多地用作复杂远程操作的统一载体，HTTP/1.1 已经开始显现出了一些局限性。HTTP 的发展中至少存在 4 个方面的问题。

- 复杂性
  HTTP 相当复杂，而且其特性之间是相互依存的。由于存在一些复杂的、相互交织的要求，以及连接管理、报文处理和功能逻辑之间的混合作用，要想正确地实现 HTTP 软件肯定是非常痛苦、很容易出错的。

- 可扩展性
  HTTP 很难实现递增式扩展。很多流传下来的 HTTP 应用程序中都没有自主的功能性扩展技术，使协议的扩展无法兼容。

- 性能
  HTTP 中有些部分效率不高。其中很多低效特性会随着高时延、低吞吐量的无线

访问技术的广泛使用而变得更加严重。

- 传输依赖性
  HTTP 是围绕 TCP/IP 网络协议栈设计的。尽管没有限制说不能使用替代协议栈，但在这方面所做的工作非常少。HTTP 要为替代协议栈提供更多的支持，才能作为一个更广阔的报文发送平台应用于嵌入式和无线应用程序之中。

10.2 HTTP-NG的活动

1997 年夏天，万维网联盟启动了一个特殊项目，调查并提出一个新的 HTTP 版本，以修正与复杂性、可扩展性、性能及传输依赖性有关的一些问题。这个新的 HTTP 被称为 HTTP：下一代系统（HTTP-NG）。

在 1998 年 12 月举行的一次 IETF 会议上，提出了一组 HTTP-NG 建议。这些建议勾勒出了一种可能的 HTTP 主要发展方向。这项技术还未被广泛采用（可能永远也不会被广泛采用了），但 HTTP-NG 确实在扩展 HTTP 系统方面做出了最认真的努力。下面我们来仔细看看 HTTP-NG。

10.3 模块化及功能增强

可以用三个英语单词来描述 HTTP-NG 的主题：“模块化及功能增强”（modularize and enhance）。如图 10-1 所示，HTTP-NG 工作组建议将协议模块化为三层，而不是将连接管理、报文处理、服务器处理逻辑和协议方法全都混在一起。 248

- 第一层，报文传输层（message transport layer），这一层不考虑报文的功能，而是致力于端点间报文的不透明传输。报文传输层支持各种子协议栈（比如无线环境下的协议栈），主要负责处理高效报文传输及处理方面的问题。HTTP-NG 项目组为本层提出了一个名为 WebMUX 的协议。
- 第二层，远程调用层（remote invocation layer），定义了请求 / 响应的功能，客户端可以通过这些功能调用对服务器资源的操作。本层独立于报文的传输以及操作的精确语义。它只是提供了一种标准的方法来调用服务器上所有的操作。本层试图提供一种像 CORBA、DCOM 和 Jave RMI 那样的面向对象的可扩展框架，而不是 HTTP/1.1 中那种静态的、服务器端定义的方法。HTTP-NG 项目组建议本层使用二进制连接协议（Binary Wire Protocol）。
- 第三层，Web 应用层（Web application layer），提供了大部分的内容管理逻辑。所有的 HTTP/1.1 方法（GET、POST、PUT 等），以及 HTTP/1.1 首部参数都是在这里定义的。本层还支持其他构建在远程调用基础上的服务，比如 WebDAV。



图 10-1　HTTP-NG 将功能都分散到各层之中实现

只要将 HTTP 组件模块化了，就可以对其进行改进，以提供更好的性能和更丰富的特性。

10.4　分布式对象

HTTP-NG 的很多基本原理和功能目标都是从 CORBA 和 DCOM 这样的结构化、面向对象的分布式对象系统中借鉴来的。分布式对象系统对可扩展性和功能特性都很有帮助。

从 1996 年开始，一个研究团体就在争论是否要将 HTTP 与更复杂的分布式对象系统聚合在一起。在 Web 中使用分布式对象模型有很多好处，更多与此相关的信息请查阅 Xerox PARC 早期名为“Migrating the Web Toward Distributed Objects”（“Web 向分布式对象的迁移”）的文章（ftp://ftp.parc.xerox.com/pub/ilu/misc/Webilu.html）。

249

将 Web 和分布式对象统一起来的雄心使得 HTTP-NG 的应用受到了某些社团的抵制。过去的一些分布式对象系统受到了重量级实现方案和形式上复杂性的影响。HTTP-NG 项目组也尝试去解决需求中提到的一些问题。

10.5　第一层——报文传输

我们从最底层开始，近距离地看看 HTTP-NG 这三层的功能。报文传输层关心的是报文的有效传输，不考虑报文的含义和目的。报文传输层为报文传输提供了一个 API，无论底层实际采用的是什么网络协议栈都可以使用。

本层关注的是提高报文传输的性能，其中包括：

- 对报文进行管道化和批量化传输，以降低往返时延；
- 重用连接，以降低时延，提高传输带宽；
- 在同一条连接上并行地复用多个报文流，在防止报文流饿死的同时优化共享连接；
- 对报文进行有效的分段，使报文边界的确定更加容易。

HTTP-NG 工作组将大部分精力都放在了为第一层的报文传输开发 WebMUX 协议上。WebMUX 是个高性能的报文协议，可以对报文进行分段，并在一条复用的 TCP 连接上交错地传输报文。本章会对 WebMUX 进行较为详细的介绍。

10.6 第二层——远程调用

HTTP-NG 结构的中间层提供了对远程方法调用的支持。本层提供了通用的请求 / 响应框架，客户端可通过此框架调用对服务器资源的操作。本层并不关心特定操作的实现及语义（缓存、安全性以及方法逻辑等）；它只关心允许客户端远程调用服务器操作的接口。

现在已经有很多远程方法调用标准了（举几个例子来说，比如 CORBA、DOM 和 Java RMI），本层并不打算支持这些系统中所有好的特性。但它有一个明确的目标，就是要对 HTTP/1.1 所提供的 HTTP RMI 支持进行扩展。特别是，要以可扩展的面向对象方式提供更通用的远程过程调用支持。

HTTP-NG 小组建议本层采用二进制连接协议。这个协议支持一种高性能的可扩展技术，通过这种技术可以调用服务器上经过良好描述的操作，并将结果返回。本章稍后将对二进制连接协议进行较为详细的讨论。

250

10.7 第三层——Web应用

Web 应用层是执行语义和应用程序特定逻辑的地方。HTTP-NG 工作组避开了扩展 HTTP 应用特性的诱惑，专注于正规的基础建设工作。

Web 应用层描述了一个用于提供应用程序特定服务的系统。这些服务并不单一，不同的应用程序可能使用不同的 API。比如，HTTP/1.1 的 Web 应用构成的应用程序与 WebDAV 可能会共享一些公用的部分，但又会有所不同。HTTP-NG 结构允许多个应用共存于本层，共享底层特性，它还提供了一种添加新应用程序的机制。

Web 应用层的基本思想是提供与 HTTP/1.1 等价的功能和一些扩展接口，同时将其映射到一个可扩展的分布式对象框架中去。更多与 Web 应用层接口有关的内容可以参见 http://www.w3.org/Protocols/HTTP-NG/1998/08/draft-larner-nginterfaces-00.txt。

10.8 WebMUX

HTTP-NG 工作组花费了很多精力，为报文传输开发了 WebMUX 标准。WebMUX 是一个复杂的高性能报文系统，通过这个系统，可以在一个复用的 TCP 连接上并行

地传输报文。可以对以不同速度产生和消耗的独立报文流进行高效的分组，并将其复用到一条或少数几条 TCP 连接上去（参见图 10-2）。

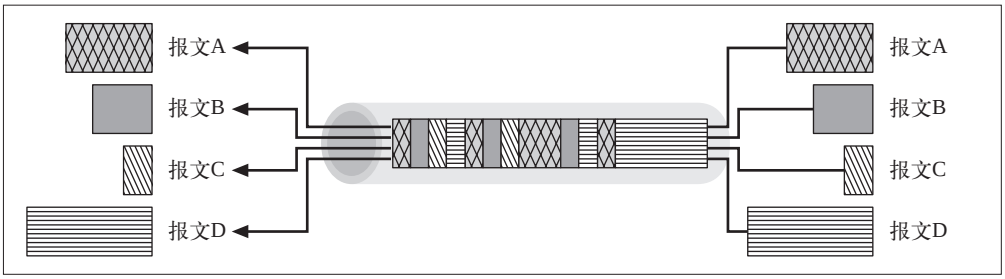


图 10-2　WebMUX 可以在一条连接上复用多条报文

WebMUX 协议的关键目标包括如下几条。

- 设计简单。
- 高性能。
- 复用——可以在一条连接上动态、高效地交错传递多个（使用任意高层协议的）数据流，不用因为等待那些速度很慢的生产者程序而延迟数据的传输。
- 基于信用的流量控制——数据是以不同的速率产生和消耗的，发送者和接收者的内存和可用的 CPU 资源都有所不同。WebMUX 使用的是“基于信用的”流量控制方案，接收者可以预先声明期望的数据接收速度，防止出现资源缺乏产生的死锁。
- 保持对齐——保持复用流中数据的对齐，这样才能有效地发送并处理二进制数据。
- 丰富的功能——接口足够丰富，能支持套接字 API。

251

更多有关 WebMUX 协议的内容请参阅 http://www.w3.org/Protocols/MUX/WD-mux-980722.html。

10.9　二进制连接协议

HTTP-NG 小组建议使用二进制连接协议来提高下一代 HTTP 协议支持远程操作的能力。

HTTP-NG 定义了一些“对象类型”，并为每种对象类型分配了一组方法。为每种对象类型分配一个 URI，以便将对它的描述和它的方法宣传出去。通过这种方式，HTTP-NG 提供了一种比 HTTP/1.1 的扩展性更强，且面向对象的执行模型，HTTP/1.1 中所有的方法都是在服务器中静态定义的。

二进制连接协议通过一条有状态的连接承载了从客户端发往服务器的操作调用请求，以及从服务器发往客户端的操作结果应答。有状态的连接可以提供更高的效率。

请求报文中包含操作、目标对象和可选的数据值。应答报文带回了操作的最终状态、所对应请求的序列号（允许以任意顺序传递并行的请求和响应），以及可选的返回值。除了请求和应答报文之外，这个协议还定义了几种内部控制报文，用来提高连接的效率和强壮性。

更多有关二进制连接协议的内容请参阅 http://www.w3.org/Protocols/HTTP-NG/1998/08/draft-janssen-httpng-wire-00.txt。

10.10 当前的状态

1998 年底，HTTP-NG 小组认定现在将 HTTP-NG 建议引入 IETF 还为时尚早。工业界和各社团都还没有完全调整到 HTTP/1.1 上来，如果没有明确的转换计划就进行重大的 HTTP-NG 重构计划，将其重构为分布式对象模式可能会造成严重的破坏。

252

对此，人们提出了两点建议。

- 不要指望能在一步之内实现整个 HTTP-NG 重构计划，建议重点关注 WebMUX 传输技术。但是，在编写此书的时候，人们还没有足够的兴趣去建立一个 WebMUX 工作组。
- 要去探讨能否将正式的协议类型修改得足够灵活（可能是通过 XML 实现的），以满足 Web 上应用的需要。对于可扩展的分布式对象系统来说，这一点尤其重要。这项工作仍在进行之中。

编写此书时，还没有什么主导力量在驱动 HTTP-NG 的应用。但随着 HTTP 应用的不断增多，随着它越来越多地作为各种应用程序的平台使用，以及无线因特网技术和面向消费者的因特网技术的不断应用，HTTP-NG 中提出的一些技术可能会在 HTTP 的青少年时期逐渐显现出其重要性。

10.11 更多信息

更多有关 HTTP-NG 的信息，请参见下面列出的详细规范和活动报告。

- http://www.w3.org/Protocols/HTTP-NG/
  HTTP-NG 工作组（建议），W3C 联盟 Web 站点。

- http://www.w3.org/Protocols/MUX/WD-mux-980722.html
  J. Gettys 和 H. Nielsen 编写的“The WebMUX Protocol”(“WebMUX 协议”)。

- http://www.w3.org/Protocols/HTTP-NG/1998/08/draft-janssen-httpng-wire-00.txt
  B. Janssen 编写的“Binary Wire Protocol for HTTP-NG”(“HTTP-NG 的二进制连接协议”)。

- http://www.w3.org/Protocols/HTTP-NG/1998/08/draft-larner-nginterfaces-00.txt
  D. Larner 编写的“HTTP-NG Web Interfaces”(“HTTP-NG Web 接口”)。

- ftp://ftp.parc.xerox.com/pub/ilu/misc/Webilu.html
  D. Larner 编写的“Migrating the Web Toward Distributed Objects”(“Web 向分布式对象的迁移”)。

253

第三部分

识别、认证与安全

第三部分的 4 章提供了一系列的技术和技巧，可用来跟踪身份、进行安全性检查，控制对内容的访问。

- 第 11 章介绍了识别用户的技巧，这样就可以实现用户个性化的内容了。
- 第 12 章重点概述了用户身份验证的基本原理。这一章还探讨了 HTTP 认证与数据库之间的接口机制。
- 第 13 章解释了摘要验证，这是一种复杂的、针对 HTTP 的增强方式，可以极大地提高安全性。
- 第 14 章详细介绍了因特网密码学、数字证书和安全套接字层。

254
~
256

第11章

客户端识别与cookie机制

Web 服务器可能会同时与数千个不同的客户端进行对话。这些服务器通常要记录下它们在与谁交谈，而不会认为所有的请求都来自匿名的客户端。本章讨论了一些服务器可以用来识别其交谈对象的技巧。

11.1 个性化接触

HTTP 最初是一个匿名、无状态的请求 / 响应协议。服务器处理来自客户端的请求，然后向客户端回送一条响应。Web 服务器几乎没有什么信息可以用来判定是哪个用户发送的请求，也无法记录来访用户的请求序列。

现代的 Web 站点希望能够提供个性化的接触。它们希望对连接另一端的用户有更多的了解，并且能在用户浏览页面时对其进行跟踪。Amazon.com 这样流行的在线商店网站可以通过以下几种方式实现站点的个性化。

- 个性化的问候
  专门为用户生成的欢迎词和页面内容，使购物体验更加个性化。

- 有的放矢的推荐
  通过了解客户的兴趣，商店可以推荐一些它们认为客户会感兴趣的商品。商店还可以在临近客户生日或其他一些重要日子的时候提供生日特定的商品。

- 管理信息的存档
  在线购物的用户不喜欢一次又一次地填写繁琐的地址和信用卡信息。有些站点会将这些管理细节存储在一个数据库中。只要他们识别出用户，就可以使用存档的管理信息，使得购物体验更加便捷。

257

- 记录会话
  HTTP 事务是无状态的。每条请求 / 响应都是独立进行的。很多 Web 站点希望能在用户与站点交互的过程中（比如，使用在线购物车的时候）构建增量状态。要实现这一功能，Web 站点就需要有一种方式来区分来自不同用户的 HTTP 事务。

本章对 HTTP 识别用户的几种技巧进行了总结。HTTP 并不是天生就具有丰富的识别特性的。早期的 Web 站点设计者们（他们都是些注重实际的人）都有自己的用户识别技术。每种技术都有其优势和劣势。本章我们将讨论下列用户识别机制。

- 承载用户身份信息的 HTTP 首部。
- 客户端 IP 地址跟踪，通过用户的 IP 地址对其进行识别。
- 用户登录，用认证方式来识别用户。
- 胖 URL，一种在 URL 中嵌入识别信息的技术。

- cookie，一种功能强大且高效的持久身份识别技术。

11.2 HTTP首部

表 11-1 给出了七种最常见的用来承载用户相关信息的 HTTP 请求首部。这里我们先讨论前三个；后面四个首部用于更高级的识别技术，我们稍后讨论。

表11-1 承载用户相关信息的HTTP首部

首部名称	首部类型	描　述
From	请求	用户的 E-mail 地址
User-Agent	请求	用户的浏览器软件
Referer	请求	用户是从这个页面上依照链接跳转过来的
Authorization	请求	用户名和密码（稍后讨论）
Client-IP	扩展（请求）	客户端的 IP 地址（稍后讨论）
X-Forwarded-For	扩展（请求）	客户端的 IP 地址（稍后讨论）
Cookie	扩展（请求）	服务器产生的 ID 标签（稍后讨论）

From 首部包含了用户的 E-mail 地址。每个用户都有不同的 E-mail 地址，所以在理想情况下，可以将这个地址作为可行的源端来识别用户。但由于担心那些不讲道德的服务器会搜集这些 E-mail 地址，用于垃圾邮件的散发，所以很少有浏览器会发送 From 首部。实际上，From 首部是由自动化的机器人或蜘蛛发送的，这样在出现问题时，网管还有个地方可以发送愤怒的投诉邮件。

258

User-Agent 首部可以将用户所用浏览器的相关信息告知服务器，包括程序的名称和版本，通常还包含操作系统的相关信息。要实现定制内容与特定的浏览器及其属性间的良好互操作时，这个首部是非常有用的，但它并没有为识别特定的用户提供太多有意义的帮助。下面是两种 User-Agent 首部，一种是网景的 Navigator 发送的，另一种是微软的 Internet Explorer 发送的：

- Navigator 6.2

```
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:0.9.4)
Gecko/20011128
  Netscape6/6.2.1
```

- Internet Explorer 6.01

```
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
```

Referer 首部提供了用户来源页面的 URL。Referer 首部自身并不能完全标识用户，但它确实说明了用户之前访问过哪个页面。通过它可以更好地理解用户的浏览

行为，以及用户的兴趣所在。比如，如果你是从一个篮球网站抵达某个 Web 服务器的，这个服务器可能会推断你是个篮球迷。

From、User-Agent 和 Referer 首部都不足以实现可靠的识别。后面的小节对识别特定用户的策略进行了更为详细的讨论。

11.3 客户端IP地址

早期的 Web 先锋曾尝试着将客户端 IP 地址作为一种标识形式使用。如果每个用户都有不同的 IP 地址，IP 地址（如果会发生变化的话）也很少会发生变化，而且 Web 服务器可以判断出每条请求的客户端 IP 地址的话，这种方案是可行的。通常在 HTTP 首部并不提供客户端的 IP 地址，[1] 但 Web 服务器可以找到承载 HTTP 请求的 TCP 连接另一端的 IP 地址。

比如，在 Unix 系统中，函数调用 getpeername 就可以返回发送端机器的客户端 IP 地址：

```
status = getpeername(tcp_connection_socket,...);
```

但是，使用客户端 IP 地址来识别用户存在着很多缺点，限制了将其作为用户识别技术的效能。

- 客户端 IP 地址描述的是所用的机器，而不是用户。如果多个用户共享同一台计算机，就无法对其进行区分了。
- 很多因特网服务提供商都会在用户登录时为其动态分配 IP 地址。用户每次登录时，都会得到一个不同的地址，因此 Web 服务器不能假设 IP 地址可以在各登录会话之间标识用户。
- 为了提高安全性，并对稀缺的地址资源进行管理，很多用户都是通过网络地址转换（Network Address Translation，NAT）防火墙来浏览网络内容的。这些 NAT 设备隐藏了防火墙后面那些实际客户端的 IP 地址，将实际的客户端 IP 地址转换成了一个共享的防火墙 IP 地址（和不同的端口号）。
- HTTP 代理和网关通常会打开一些新的、到原始服务器的 TCP 连接。Web 服务器看到的将是代理服务器的 IP 地址，而不是客户端的。有些代理为了绕过这个问题会添加特殊的 Client-IP 或 X-Forwarded-For 扩展首部来保存原始的 IP 地址（参见图 11-1）。但并不是所有的代理都支持这种行为。

259

有些 Web 站点仍然使用客户端 IP 地址在会话之间跟踪用户的行为，但这种站点并不多。无法用 IP 地址确定目标的地方太多了。

注 1：稍后我们会看到，有些代理确实会添加一个 Client-IP 首部，但这并不是 HTTP 标准的一部分。

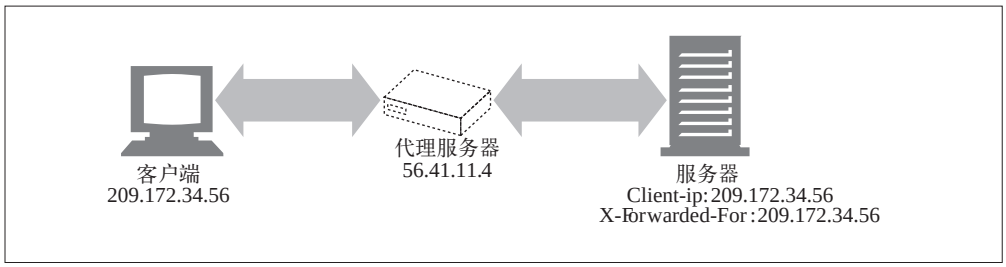


图 11-1　代理可以添加扩展首部，来传递原始客户端的 IP 地址

少数站点甚至将客户端 IP 地址作为一种安全特性使用，它们只向来自特定 IP 地址的用户提供文档。在内部网络中可能可以这么做，但在因特网上就不行了，主要是因为因特网上 IP 地址太容易被欺骗（伪造）了。路径上如果有拦截代理也会破坏此方案。第 14 章讨论了一些强大得多的特权文档访问控制策略。

11.4　用户登录

Web 服务器无需被动地根据用户的 IP 地址来猜测他的身份，它可以要求用户通过用户名和密码进行认证（登录）来显式地询问用户是谁。

为了使 Web 站点的登录更加简便，HTTP 中包含了一种内建机制，可以用 `WWW-Authenticate` 首部和 `Authorization` 首部向 Web 站点传送用户的相关信息。一旦登录，浏览器就可以不断地在每条发往这个站点的请求中发送这个登录信息了，这样，就总是有登录信息可用了。我们将在第 12 章对这种 HTTP 认证机制进行更加详细的讨论，现在我们先来简单地看一看。

260

如果服务器希望在为用户提供对站点的访问之前，先行登录，可以向浏览器回送一条 HTTP 响应代码 401 Login Required。然后，浏览器会显示一个登录对话框，并用 `Authorization` 首部在下一条对服务器的请求中提供这些信息。[2] 图 11-2 对此进行了说明。

此图中发生的情况如下所述。

- 在图 11-2a 中，浏览器对站点 www.joes-hardware.com 发起了一条请求。
- 站点并不知道这个用户的身份，因此在图 11-2b 中，服务器会返回 401 Login Required HTTP 响应码，并添加 `WWW-Authentication` 首部，要求用户登录。这样浏览器就会弹出一个登录对话框。

261

注 2：为了不让用户每发送一条请求都要登录一次，大多数浏览器都会记住某站点的登录信息，并将登录信息放在发送给该站点的每条请求中。

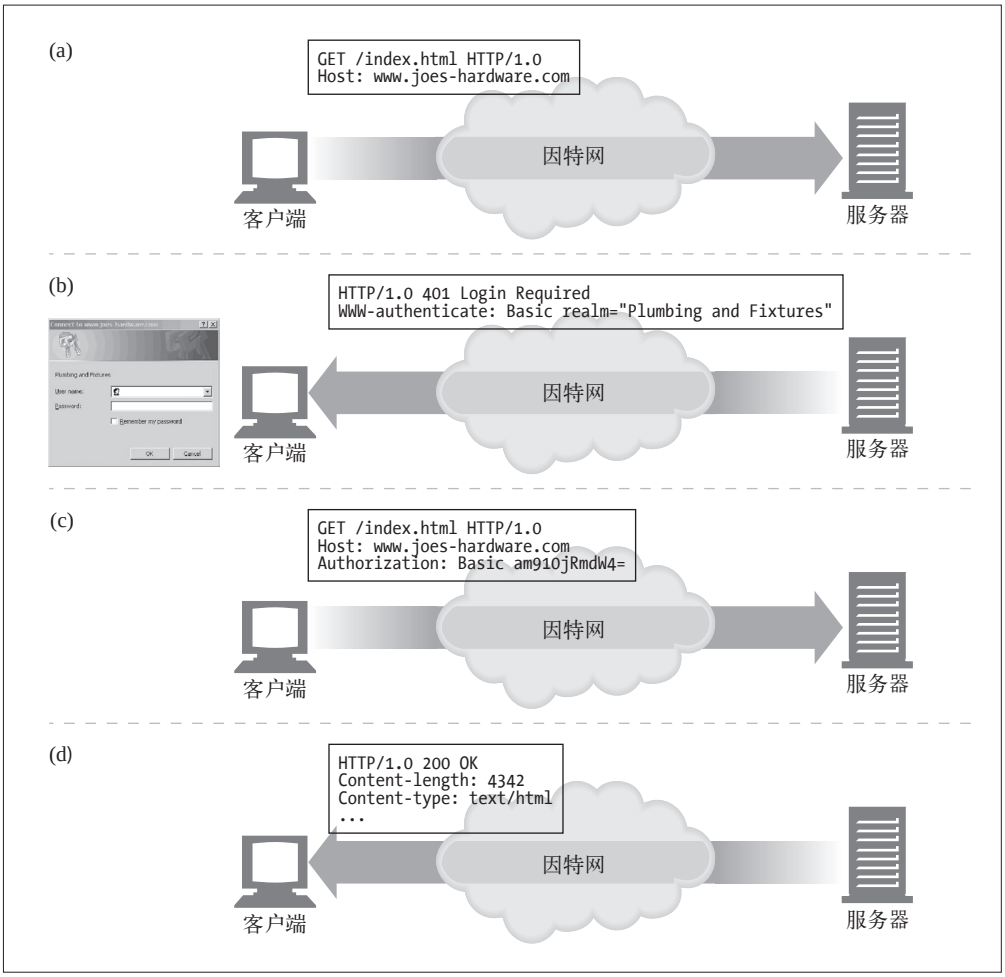


图 11-2　用 HTTP 认证首部注册用户名

- 只要用户输入了用户名和密码（对其身份进行完整性检查），浏览器就会重复原来的请求。这次它会添加一个 `Authorization` 首部，说明用户名和密码。对用户名和密码进行加密，防止那些有意无意的网络观察者看到。[3]
- 现在，服务器已经知道用户的身份了。
- 今后的请求要使用用户名和密码时，浏览器会自动将存储下来的值发送出去，甚至在站点没有要求发送的时候也经常会向其发送。浏览器在每次请求中都向服务器发送 `Authorization` 首部作为一种身份的标识，这样，只要登录一次，就可以在整个会话期间维持用户的身份了。

注 3：在第 14 章我们会看到，任何有这种想法的人，不用费多大事就可以轻易地将 HTTP 基本的认证用户名和密码破解出来。稍后将讨论一些更安全的技术。

但是，登录多个 Web 站点是很繁琐的。Fred 从一个站点浏览到另一个站点的时候，需要在每个站点上登录。更糟的是，可怜的 Fred 很可能要为不同的站点记住不同的用户名和密码。他访问很多站点的时候，他最喜欢的用户名 fred 可能已经被其他人用过了，而且有些站点为用户名和密码的长度和组成设置了不同的规则。Fred 很快就会放弃上网，回去看奥普拉（Oprah）的脱口秀了。下一节我们来讨论这个问题的解决方案。

11.5 胖URL

有些 Web 站点会为每个用户生成特定版本的 URL 来追踪用户的身份。通常，会对真正的 URL 进行扩展，在 URL 路径开始或结束的地方添加一些状态信息。用户浏览站点时，Web 服务器会动态生成一些超链，继续维护 URL 中的状态信息。

改动后包含了用户状态信息的 URL 被称为胖 URL（fat URL）。下面是 Amazon.com 电子商务网站使用的一些胖 URL 实例。每个 URL 后面都附加了一个用户特有的标识码（在这个例子中就是 002-1145265-8016838），这个标识码有助于在用户浏览商店内容时对其进行跟踪。


```
...
<a href="/exec/obidos/tg/browse/-/229220/ref=gr_gifts/002-1145265-
   8016838">All Gifts</a><br>
<a href="/exec/obidos/wishlist/ref=gr_pl1_/002-1145265-8016838">Wish
   List</a><br>
...
<a href="http://s1.amazon.com/exec/varzea/tg/armed-forces/-//ref=gr_
af_/002-1145265-8016838">Salute Our Troops</a><br>
<a href="/exec/obidos/tg/browse/-/749188/ref=gr_p4_/002-1145265-8016838"
   >Free Shipping</a><br>
<a href="/exec/obidos/tg/browse/-/468532/ref=gr_returns/002-1145265-8016838"
   >Easy Returns</a>
...
```

可以通过胖 URL 将 Web 服务器上若干个独立的 HTTP 事务捆绑成一个“会话”或“访问”。用户首次访问这个 Web 站点时，会生成一个唯一的 ID，用服务器可以识别的方式将这个 ID 添加到 URL 中去，然后服务器就会将客户端重新导向这个胖 URL。不论什么时候，只要服务器收到了对胖 URL 的请求，就可以去查找与那个用户 ID 相关的所有增量状态（购物车、简介等），然后重写所有的输出超链，使其成为胖 URL，以维护用户的 ID。

可以在用户浏览站点时，用胖 URL 对其进行识别。但这种技术存在几个很严重的问题。

- *丑陋的* URL
  浏览器中显示的胖 URL 会给新用户带来困扰。

- *无法共享 URL*
  胖 URL 中包含了与特定用户和会话有关的状态信息。如果将这个 URL 发送给其他人，可能就在无意中将你积累的个人信息都共享出去了。

- *破坏缓存*
  为每个 URL 生成用户特有的版本就意味着不再有可供公共访问的 URL 需要缓存了。

- *额外的服务器负荷*
  服务器需要重写 HTML 页面使 URL 变胖。

- *逃逸口*
  用户跳转到其他站点或者请求一个特定的 URL 时，就很容易在无意中“逃离”胖 URL 会话。只有当用户严格地追随预先修改过的链接时，胖 URL 才能工作。如果用户逃离此链接，就会丢失他的进展（可能是一个已经装满了东西的购物车）信息，得重新开始。

- *在会话间是非持久的*
  除非用户收藏了特定的胖 URL，否则用户退出登录时，所有的信息都会丢失。

11.6 cookie

cookie 是当前识别用户，实现持久会话的最好方式。前面各种技术中存在的很多问题对它们都没什么影响，但是通常会将它们与那些技术共用，以实现额外的价值。cookie 最初是由网景公司开发的，但现在所有主要的浏览器都支持它。

cookie 非常重要，而且它们定义了一些新的 HTTP 首部，所以我们要比前面那些技术更详细地介绍它们。cookie 的存在也影响了缓存，大多数缓存和浏览器都不允许
263 对任何 cookie 的内容进行缓存。后面的小节对此进行了更为详细的介绍。

11.6.1 cookie的类型

可以笼统地将 cookie 分为两类：会话 cookie 和持久 cookie。会话 cookie 是一种临时 cookie，它记录了用户访问站点时的设置和偏好。用户退出浏览器时，会话 cookie 就被删除了。持久 cookie 的生存时间更长一些；它们存储在硬盘上，浏览器退出，计算机重启时它们仍然存在。通常会用持久 cookie 维护某个用户会周期性访问的站点的配置文件或登录名。

会话 cookie 和持久 cookie 之间唯一的区别就是它们的过期时间。稍后我们会看到，如果设置了 `Discard` 参数，或者没有设置 `Expires` 或 `Max-Age` 参数来说明扩展的

过期时间，这个 cookie 就是一个会话 cookie。

11.6.2 cookie是如何工作的

cookie 就像服务器给用户贴的“嗨，我叫”的贴纸一样。用户访问一个 Web 站点时，这个 Web 站点就可以读取那个服务器贴在用户身上的所有贴纸。

用户首次访问 Web 站点时，Web 服务器对用户一无所知（参见图 11-3a）。Web 服务器希望这个用户会再次回来，所以想给这个用户“拍上”一个独有的 cookie，这样以后它就可以识别出这个用户了。cookie 中包含了一个由*名字 = 值*（name=value）这样的信息构成的任意列表，并通过 `Set-Cookie` 或 `Set-Cookie2` HTTP 响应（扩展）首部将其贴到用户身上去。

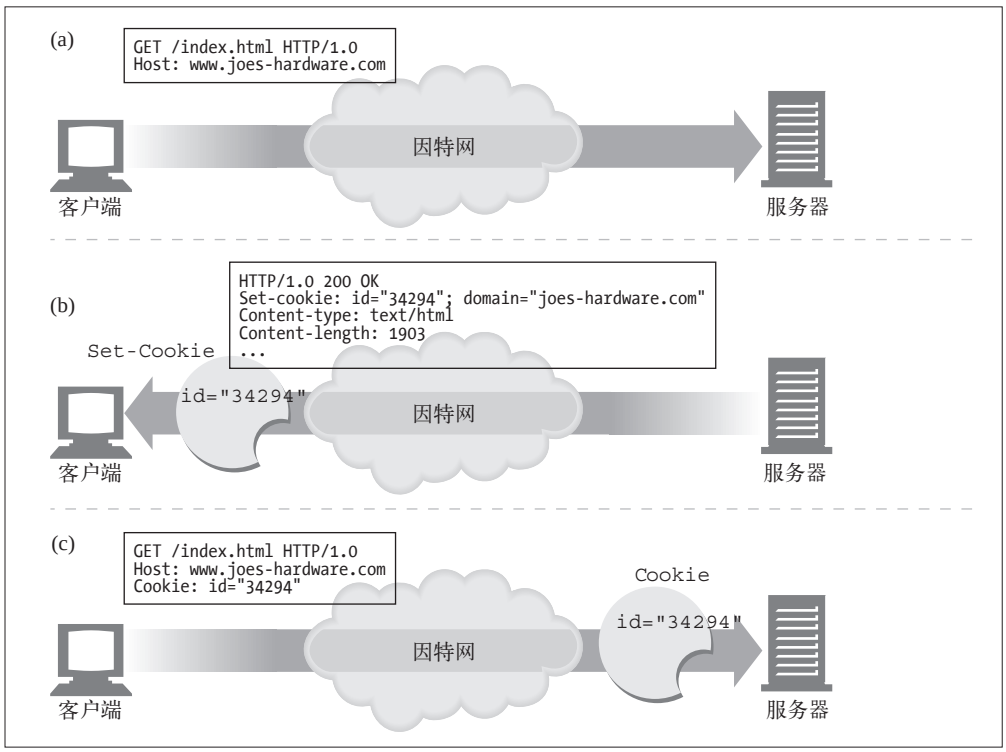


图 11-3 给用户贴一个 cookie

cookie 中可以包含任意信息，但它们通常都只包含一个服务器为了进行跟踪而产生的独特的识别码。比如，在图 11-3b 中，服务器会将一个表示 `id="34294"` 的 cookie 贴到用户上去。服务器可以用这个数字来查找服务器为其访问者积累的数据库信息（购物历史、地址信息等）。

但是，cookie 并不仅限于 ID 号。很多 Web 服务器都会将信息直接保存在 cookie 中。比如：

```
Cookie: name="Brian Totty"; phone="555-1212"
```

浏览器会记住从服务器返回的 `Set-Cookie` 或 `Set-Cookie2` 首部中的 cookie 内容，并将 cookie 集存储在浏览器的 cookie 数据库中（把它当作一个贴有不同国家贴纸的旅行箱）。将来用户返回同一站点时（参见图 11-3c），浏览器会挑中那个服务器贴到用户上的那些 cookie，并在一个 cookie 请求首部中将其传回去。

11.6.3 cookie罐：客户端的状态

cookie 的基本思想就是让浏览器积累一组服务器特有的信息，每次访问服务器时都
264 将这些信息提供给它。因为浏览器要负责存储 cookie 信息，所以此系统被称为*客户端侧状态*（client-side state）。这个 cookie 规范的正式名称为 HTTP 状态管理机制（HTTP state management mechanism）。

1. 网景的Navigator的cookie

不同的浏览器会以不同的方式来存储 cookie。网景的 Navigator 会将 cookie 存储在
265 一个名为 cookies.txt 的文本文件中。例如：

```
# Netscape HTTP Cookie File
# http://www.netscape.com/newsref/std/cookie_spec.html
# This is a generated file! Do not edit.
#
# domain                  allh  path      secure expires     name         value

www.fedex.com             FALSE /         FALSE  1136109676  cc           /us/
.bankofamericaonline.com  TRUE  /         FALSE  1009789256  state        CA
.cnn.com                  TRUE  /         FALSE  1035069235  SelEdition   www
secure.eepulse.net        FALSE /ePulse   FALSE  1007162968  cid          %FE%FF%002
www.reformamt.org         TRUE  /forum    FALSE  1033761379  LastVisit    1003520952
www.reformamt.org         TRUE  /forum    FALSE  1033761379  UserName     Guest
  ...
```

文本文件中的每一行都代表一个 cookie。有 7 个用 tab 键分隔的字段。

- domain（域）
  cookie 的域。
- allh
  是域中所有的主机都获取 cookie，还是只有指定了名字的主机获取。
- path（路径）
  域中与 cookie 相关的路径前缀。

- secure（安全）
  是否只有在使用 SSL 连接时才发送这个 cookie。
- expiration（过期）
  从格林尼治标准时间 1970 年 1 月 1 日 00:00:00 开始的 cookie 过期秒数。
- name（名字）
  cookie 变量的名字。
- value（值）
  cookie 变量的值。

2. 微软Internet Explorer的cookie

微软的 Internet Explorer 将 cookie 存储在高速缓存目录下独立的文本文件中。可以通过浏览这个目录来查看 cookie，如图 11-4 所示。Internet Explorer 中 cookie 文件格式是特有的，但很多字段都很容易理解。cookie 一个接一个地存储在文件中，每个 cookie 都由多行构成。

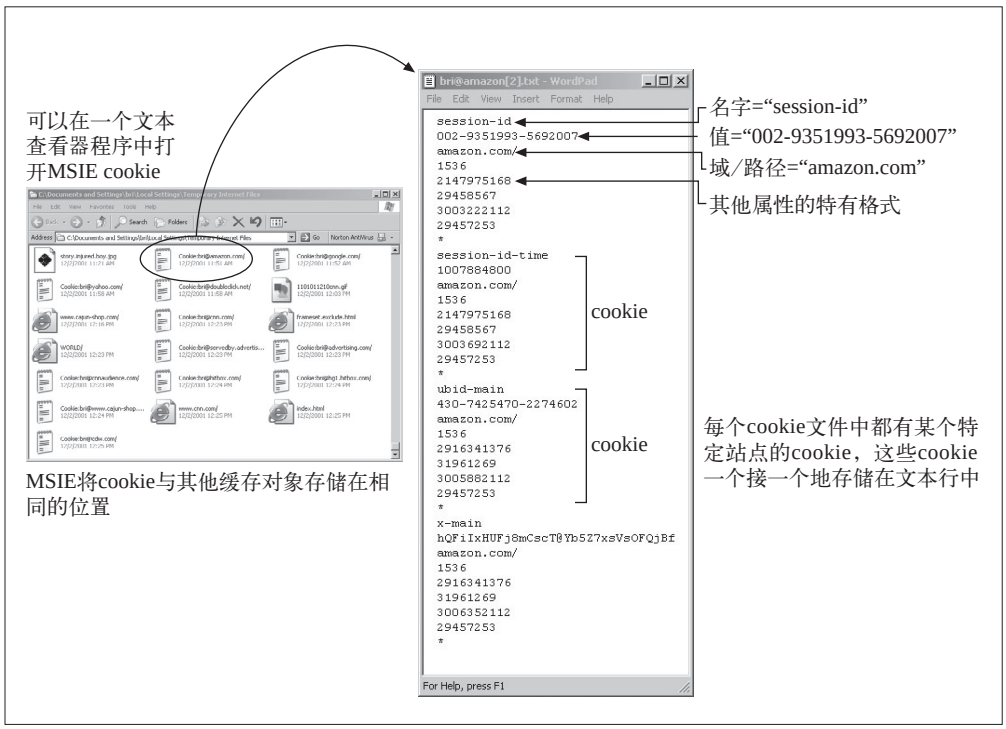


图 11-4　Internet Explorer 的 cookie 存储在缓存目录下独立的文本文件中

文件中每个 cookie 的第一行中都包含了 cookie 的变量名。下一行是变量的值。第三行是域和路径。剩下的行就是一些特有的数据，可能包含日期和一些标记。

11.6.4 不同站点使用不同的cookie

浏览器内部的 cookie 罐中可以有成百上千个 cookie，但浏览器不会将每个 cookie 都发送给所有的站点。实际上，它们通常只向每个站点发送 2 ~ 3 个 cookie。原因如下。

- 对所有这些 cookie 字节进行传输会严重降低性能。浏览器实际传输的 cookie 字节数要比实际的内容字节数多！
- cookie 中包含的是服务器特有的名值对，所以对大部分站点来说，大多数 cookie 都只是无法识别的无用数据。
- 将所有的 cookie 发送给所有站点会引发潜在的隐私问题，那些你并不信任的站点也会获得你只想发给其他站点的信息。

266

总之，浏览器只向服务器发送服务器产生的那些 cookie。joes-hardware.com 产生的 cookie 会被发送给 joes-hardware.com，不会发送给 bobs-books.com 或 marys-movies.com。

很多 Web 站点都会与第三方厂商达成协议，由其来管理广告。这些广告被做得像 Web 站点的一个组成部分，而且它们确实发送了持久 cookie。用户访问另一个由同一广告公司提供服务的站点时，（由于域是匹配的）浏览器就会再次回送早先设置的持久 cookie。营销公司可以将此技术与 `Referer` 首部结合，暗地里构建一个用户档案和浏览习惯的详尽数据集。现代的浏览器都允许用户对隐私特性进行设置，以限制第三方 cookie 的使用。

1. cookie的域属性

产生 cookie 的服务器可以向 `Set-Cookie` 响应首部添加一个 Domain 属性来控制哪些站点可以看到那个 cookie。比如，下面的 HTTP 响应首部就是在告诉浏览器将 `cookie user="mary17"` 发送给域 `".airtravelbargains.com"` 中的所有站点：

267

```
Set-cookie: user="mary17"; domain="airtravelbargains.com"
```

如果用户访问的是 www.airtravelbargains.com、specials.airtravelbargains.com 或任意以 .airtravelbargains.com 结尾的站点，下列 `Cookie` 首部都会被发布出去：

```
Cookie: user="mary17"
```

2. cookie路径属性

cookie 规范甚至允许用户将 cookie 与部分 Web 站点关联起来。可以通过 Path 属性来实现这一功能，在这个属性列出的 URL 路径前缀下所有 cookie 都是有效的。

例如，某个 Web 服务器可能是由两个组织共享的，每个组织都有独立的 cookie。站点 www.airtravelbargains.com 可能会将部分 Web 站点用于汽车租赁——比如，http://www.airtravelbargains.com/autos/——用一个独立的 cookie 来记录用户喜欢的汽车尺寸。可能会生成一个如下所示的特殊汽车租赁 cookie：

```
Set-cookie: pref=compact; domain="airtravelbargains.com"; path=/autos/
```

如果用户访问 http://www.airtravelbargains.com/specials.html，就只会获得这个 cookie：

```
Cookie: user="mary17"
```

但如果访问 http://www.airtravelbargains.com/autos/cheapo/index.html，就会获得这两个 cookie：

```
Cookie: user="mary17"
Cookie: pref=compact
```

因此，cookie 就是由服务器贴到客户端上，由客户端维护的状态片段，只会回送给那些合适的站点。下面我们来更仔细地看看 cookie 的技术和标准。

11.6.5 cookie成分

现在使用的 cookie 规范有两个不同的版本：cookies 版本 0（有时被称为 Netscape cookies）和 cookies 版本 1（RFC 2965）。cookies 版本 1 是对 cookies 版本 0 的扩展，应用不如后者广泛。

cookie 规范版本 0 和版本 1 都不是作为 HTTP/1.1 规范的一部分提供的。表 11-2 对两个主要的附属文档进行了总结，这两个文档对 cookie 的使用进行了很好的描述。

表11-2 cookie规范

标　题	描　述	位　置
持久客户端状态：HTTP cookies	最初的 Netscape cookie 标准	http://home.netscape.com/newsref/std/cookie_spec.html
RFC 2965：HTTP 状态管理机制	2000 年 10 月的 cookie 标准，废弃了 RFC 2109	http://www.ietf.org/rfc/rfc2965.txt

11.6.6 cookies版本0（Netscape）

最初的cookie规范是由网景公司定义的。这些“版本0”的cookie定义了Set-Cookie响应首部、cookie请求首部以及用于控制cookie的字段。版本0的cookie看起来如下所示：

```
Set-Cookie: name=value [; expires=date] [; path=path] [; domain=domain]
[; secure]

Cookie: name1=value1 [; name2=value2] ...
```

1. 版本0的Set-Cookie首部

Set-Cookie首部有一个强制性的cookie名和cookie值。后面跟着可选的cookie属性，中间由分号分隔。表11-3描述了Set-Cookie字段。

表11-3 版本0（网景）的Set-Cookie属性

Set-Cookie属性	描述及实例
NAME=VALUE	强制的。NAME和VALUE都是字符序列，除非包含在双引号内，否则不包括分号、逗号、等号和空格。Web服务器可以创建任意的NAME=VALUE关联，在后继对站点的访问中会将其送回给Web服务器： `Set-Cookie: customer=Mary`

Expires	可选的。这个属性会指定一个日期字符串，用来定义cookie的实际生存期。一旦到了过期日期，就不再存储或发布这个cookie了。日期的格式为：
	`Weekday, DD-Mon-YY HH:MM:SS GMT`
	唯一合法的时区为GMT，各日期元素之间的分隔符一定要是长划线。如果没有指定Expires，cookie就会在用户会话结束时过期：
	`Set-Cookie: foo=bar; expires=Wednesday, 09-Nov-99 23:12:40 GMT`

Domain	可选的。浏览器只向指定域中的服务器主机名发送cookie。这样服务器就将cookie限制在了特定的域中。acme.com域就与anvil.acme.com和shipping.crate.acme.com相匹配，但与www.cnn.com就不匹配了。 只有指定域中的主机才能为一个域设置cookie，这些域中至少要有两个或三个句号，以防止出现.com、.edu和va.us等形式的域。这里列出了一组固定的特定高层域，落在这个范围中的域只需要两个句号。所有其他域都至少需要三个句号。特定的高层域包括：.com、.edu、.net、.org、.gov、.mil、.int、.biz、.info、.name、.museum、.coop、.aero和.pro。 如果没有指定域，就默认为产生Set-Cookie响应的服务器的主机名： `Set-Cookie: SHIPPING=FEDEX; domain="joes-hardware.com"`

（续）

Set-Cookie属性	描述及实例
Path	可选的。通过这个属性可以为服务器上特定的文档分配 cookie。如果 Path 属性是一个 URL 路径前缀，就可以附加一个 cookie。路径 /foo 与 /foobar 和 /foo/bar.html 相匹配。路径“/”与域名中所有内容都匹配。 如果没有指定路径，就将其设置为产生 Set-Cookie 响应的 URL 的路径： Set-Cookie: lastorder=00183; path=/orders
Secure	可选的。如果包含了这一属性，就只有在 HTTP 使用 SSL 安全连接时才会发送 cookie： Set-Cookie: private_id=519; secure

269

2. 版本0的Cookie首部

客户端发送请求时，会将所有与域、路径和安全过滤器相匹配的未过期 cookie 都发送给这个站点。所有 cookie 都被组合到一个 Cookie 首部中：

```
Cookie: session-id=002-1145265-8016838; session-id-time=1007884800
```

11.6.7 cookies版本1（RFC 2965）

RFC 2965（以前的 RFC 2109）定义了一个 cookie 的扩展版本。这个版本 1 标准引入了 Set-Cookie2 首部和 Cookie2 首部，但它也能与版本 0 系统进行互操作。

RFC 2965 cookie 标准比原始的网景公司的标准略微复杂一些，还未得到完全的支持。RFC 2965 cookie 的主要改动包括下列内容。

- 为每个 cookie 关联上解释性文本，对其目的进行解释。
- 允许在浏览器退出时，不考虑过期时间，将 cookie 强制销毁。
- 用相对秒数，而不是绝对日期来表示 cookie 的 Max-Age。
- 通过 URL 端口号，而不仅仅是域和路径来控制 cookie 的能力。
- 通过 Cookie 首部回送域、端口和路径过滤器（如果有的话）。
- 为实现互操作性使用的版本号。
- 在 Cookie 首部从名字中区分出附加关键字的 $ 前缀。

cookie 版本 1 的语法如下所示：

270

```
set-cookie      =       "Set-Cookie2:" cookies
cookies         =       1#cookie
cookie          =       NAME "=" VALUE *(";" set-cookie-av)
NAME            =       attr
VALUE           =       value
set-cookie-av   =       "Comment" "=" value
```

```
                    |       "CommentURL" "=" <"> http_URL <">
                    |       "Discard"
                    |       "Domain" "=" value
                    |       "Max-Age" "=" value
                    |       "Path" "=" value
                    |       "Port" [ "=" <"> portlist <"> ]
                    |       "Secure"
                    |       "Version" "=" 1*DIGIT
    portlist        =       1#portnum
    portnum         =       1*DIGIT

    cookie          =       "Cookie:" cookie-version 1*((";" | ",") cookie-
                                value)
    cookie-value    =      NAME "=" VALUE [";" path] [";" domain] [";" port]
    cookie-version  =       "$Version" "=" value
    NAME            =       attr
    VALUE           =       value
    path            =       "$Path" "=" value
    domain          =       "$Domain" "=" value
    port            =       "$Port" [ "=" <"> value <"> ]
    cookie2         =       "Cookie2:" cookie-version
```

1. 版本1的Set-Cookie2首部

版本 1 的 cookie 标准比网景公司标准的可用属性要多。表 11-4 对这些属性做了快速汇总。更详细的解释请参见 RFC 2965。

表11-4 版本1（RFC 2965）的Set-Cookie2属性

Set-Cookie2属性	描述及实例
NAME=VALUE	强制的。Web 服务器可以创建任意的 NAME=VALUE 关联，可以在后继对站点的访问中将其发回给 Web 服务器。“$”是保留字符，所以名字一定不能以它开头
Version	强制的。这个属性的值是一个整数，对应于 cookie 规范的版本。RFC 2965 为版本 1： `Set-Cookie2: Part="Rocket_Launcher_0001"; Version="1"`
Comment	可选。这个属性说明了服务器准备如何使用这个 cookie。用户可以通过检查此策略来确定是否允许使用带有这个 cookie 的会话。这个值必须采用 UTF-8 编码
CommentURL	可选。这个属性提供了一个 URL 指针，指向详细描述了 cookie 目的及策略的文档。用户可以通过查看此策略来判定是否允许使用带有这个 cookie 的会话
Discard	可选。如果提供了这个属性，就会在客户端程序终止时，指示客户端放弃这个 cookie
Domain	可选。浏览器只向指定域中的服务器主机名发送 cookie。这样服务器就可以将 cookie 限制在特定域中了。acme.com 域与主机名 anvil.acme.com 和 shipping.crate.acme.com 相匹配，但不匹配于 www.cnn.com。域名匹配的规则基本上与 Netscape cookie 一样，但有几条附加的规则。细节请参见 RFC 2965

（续）

Set-Cookie2属性	描述及实例
Max-Age	可选。这个属性的值是一个整数，用于设置以秒为单位的 cookie 生存期。客户端应该根据 HTTP/1.1 的使用期计算规则来计算 cookie 的使用期。cookie 的使用期比 Max-Age 大时，客户端就应该将这个 cookie 丢弃。值为零说明应该立即将那个 cookie 丢弃
Path	可选。通过这个属性可以为服务器上的特定文档指定 cookie。如果 Path 属性是一个 URL 路径的前缀，就可以附加一个 cookie。路径 /foo 匹配于 /foobar 和 /foo/bar.html。路径“/”匹配于域中所有内容。如果没有指定路径，就将其设置为生成 Set-Cookie 响应的 URL 的路径

Port	可选。这个属性可以单独作为关键字使用，也可以包含一个由逗号分隔的、可以应用 cookie 的端口列表。如果有端口列表，就只能向端口与列表中的端口相匹配的服务器提供 cookie。如果单独提供关键字 Port 而没有值，就只能向当前响应服务器的端口号提供 cookie：
	`Set-Cookie2: foo="bar"; Version="1";`
	`Port="80,81,8080"`
	`Set-Cookie2: foo="bar"; Version="1"; Port`

Secure	可选。如果包含这个属性，就只有在 HTTP 使用 SSL 安全连接时才能发送 cookie

271

2. 版本1的Cookie首部

版本 1 的 cookie 会带回与传送的每个 cookie 相关的附加信息，用来描述每个 cookie 途径的过滤器。每个匹配的 cookie 都必须包含来自相应 Set-Cookie2 首部的所有 Domain、Port 或 Path 属性。

比如，假设客户端以前曾收到下列五个来自 Web 站点 www.joes-hardware.com 的 Set-Cookie2 响应：

```
Set-Cookie2: ID="29046"; Domain=".joes-hardware.com"
Set-Cookie2: color=blue
Set-Cookie2: support-pref="L2"; Domain="customer-care.joes-hardware.
com"
Set-Cookie2: Coupon="hammer027"; Version="1"; Path="/tools"
Set-Cookie2: Coupon="handvac103"; Version="1"; Path="/tools/cordless"
```

如果客户端对路径 /tools/cordless/specials.html 又发起了一次请求，就会同时发送这样一个很长的 Cookie 首部：

```
Cookie: $Version="1";
        ID="29046"; $Domain=".joes-hardware.com";
        color="blue";
        Coupon="hammer027"; $Path="/tools";
        Coupon="handvac103"; $Path="/tools/cordless"
```

注意，所有匹配 cookie 都是和它们的 Set-Cookie2 过滤器一同传输的，而且保留关键字都是以美元符号（$）开头的。

3. 版本1的Cookie2首部和版本协商

Cookie2 请求首部负责在能够理解不同 cookie 规范版本的客户端和服务器之间进行互操作性的协商。Cookie2 首部告知服务器，用户 Agent 代理理解新形式的 cookie，并提供了所支持的 cookie 标准版本（将其称为 Cookie-Version 更合适一些）：

```
Cookie2: $Version="1"
```

如果服务器理解新形式的 cookie，就能够识别出 Cookie2 首部，并在响应首部发送 Set-Cookie2（而不是 Set-Cookie）。如果客户端从同一个响应中既获得了 Set-Cookie 首部，又获得了 Set-Cookie2 首部，就会忽略老的 Set-Cookie 首部。

如果客户端既支持版本 0 又支持版本 1 的 cookie，但从服务器获得的是版本 0 的 Set-Cookie 首部，就应该带着版本 0 的 Cookie 首部发送 cookie。但客户端还应该发送 Cookie2: $Version="1"来告知服务器它是可以升级的。

11.6.8 cookie与会话跟踪

可以用 cookie 在用户与某个 Web 站点进行多项事务处理时对用户进行跟踪。电子商务 Web 站点用会话 cookie 在用户浏览时记录下用户的购物车信息。我们以流行
272 的购物网站 Amazon.com 为例。在浏览器中输入 http://www.amazon.com 时，就启动了一个事务链，在这些事务中 Web 服务器会通过一系列的重定向、URL 重写以及 cookie 设置来附加标识信息。

图 11-5 显示了从一次 Amazon.com 访问中捕获的事务序列。

- 图 11-5a——浏览器首次请求 Amazon.com 根页面。
- 图 11-5b——服务器将客户端重定向到一个电子商务软件的 URL 上。
- 图 11-5c——客户端对重定向的 URL 发起一个请求。
- 图 11-5d——服务器在响应上贴上两个会话 cookie，并将用户重定向到另一个 URL，这样客户端就会用这些附加的 cookie 再次发出请求。这个新的 URL 是个胖 URL，也就是说有些状态嵌入到 URL 中去了。如果客户端禁止了 cookie，只要用户一直跟随着 Amazon.com 产生的胖 URL 链接，不离开网站，仍然可以实现一些基本的标识功能。
- 图 11-5e——客户端请求新的 URL，但现在会传送两个附加的 cookie。

- 图 11-5f——服务器重定向到 home.html 页面，并附加另外两个 cookie。
- 图 11-5g——客户端获取 home.html 页面并将所有四个 cookie 都发送出去。
- 图 11-5h——服务器回送内容。

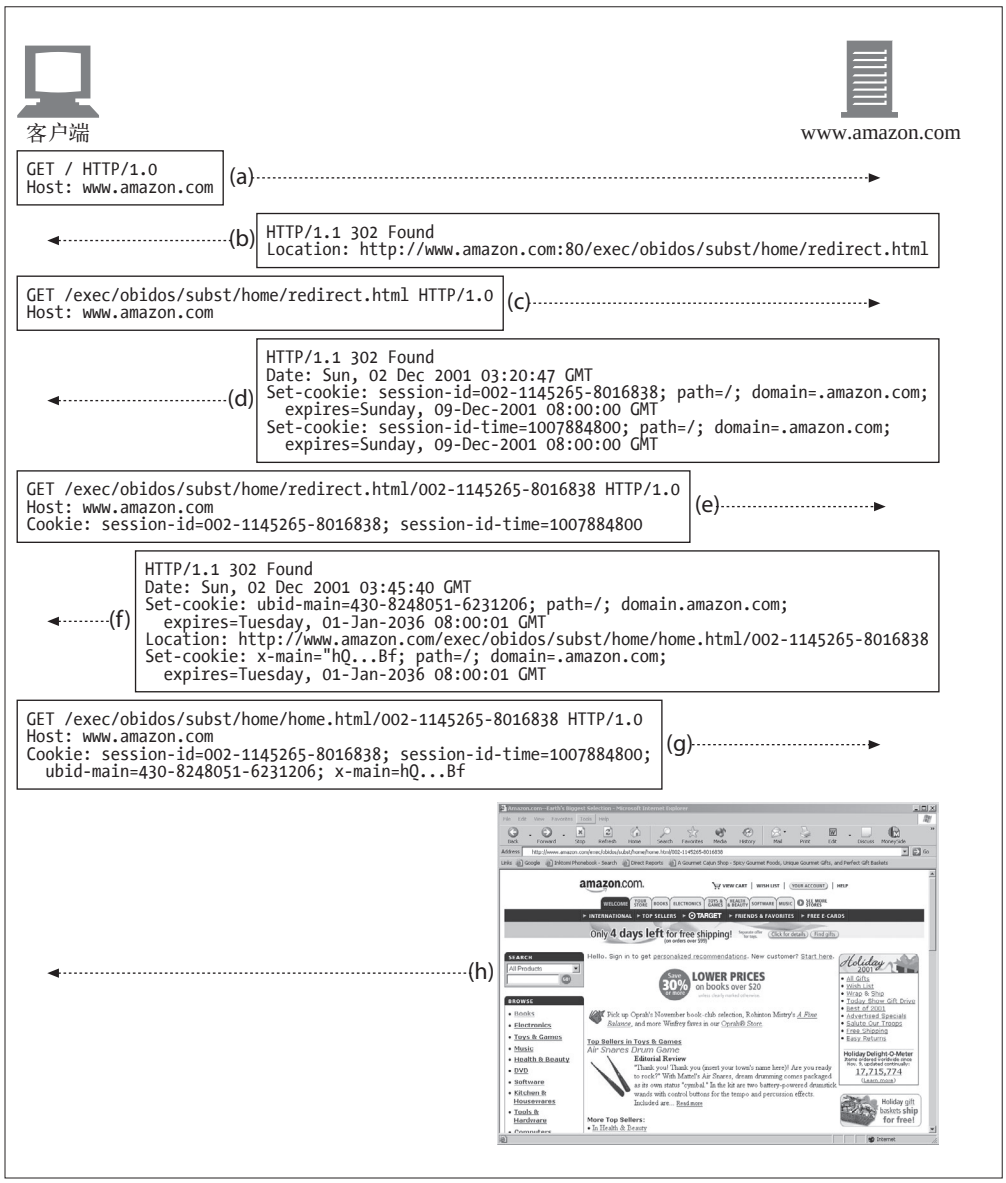


图 11-5　Amazon.com 网站用会话 cookie 来跟踪用户

11.6.9 cookie与缓存

缓存那些与 cookie 事务有关的文档时要特别小心。你不会希望给用户分配一个过去某些用户用过的 cookie，或者更糟糕的是，向一个用户展示其他人私有文档的内容。

cookie 和缓存的规则并没有很好地建立起来。下面是处理缓存时的一些指导性规则。

- 如果无法缓存文档，要将其标示出来
  文档的所有者最清楚文档是否是不可缓存的。如果文档不可缓存，就显式地注明——具体来说，如果除了 `Set-Cookie` 首部之外文档是可缓存的，就使用 `Cache-Control:no-cache="Set-Cookie"`。另一种更通用的做法是为可缓存文档使用 `Cache-Control:public`，这样有助于节省 Web 中的带宽。

- 缓存 `Set-Cookie` 首部时要小心
  如果响应中有 `Set-Cookie` 首部，就可以对主体进行缓存（除非被告知不要这么做），但要特别注意对 `Set-Cookie` 首部的缓存。如果向多个用户发送了相同的 `Set-Cookie` 首部，可能会破坏用户的定位。
  有些缓存在将响应缓存起来之前会删除 `Set-Cookie` 首部，但这样也会引发一些问题，因为在没有缓存的时候，通常都会有 cookie 贴在客户端上，但由缓存提供服务的客户端就不会有 cookie 了。强制缓存与原始服务器重新验证每条请求，并将返回的所有 `Set-Cookie` 首部都合并到客户端的响应中去，就可以改善这种状况。原始服务器可以通过向缓存的副本中添加这个首部来要求进行这种再验证：

  ```
  Cache-Control: must-revalidate, max-age=0
  ```

  即便内容实际上是可以缓存的，比较保守的缓存可能也会拒绝缓存所有包含 `Set-Cookie` 首部的响应。有些缓存允许使用缓存 `Set-Cookie` 图片，但不缓存文本的模式。

- 小心处理带有 `Cookie` 首部的请求
  带有 `Cookie` 首部的请求到达时，就在提示我们，得到的结果可能是私有的。一定要将私有内容标识为不可缓存的，但有些服务器可能会犯错，没有将此内容标记为不可缓存的。
  有些响应文档对应于携带 `Cookie` 首部的请求，保守的缓存可能会选择不去缓存这些响应文档。同样，有些缓存允许使用缓存 cookie 图片，而不缓存文本的模式。得到更广泛接受的策略是缓存带有 `Cookie` 首部的图片，将过期时间设置为零，强制每次都进行再验证。

273
~
274

11.6.10 cookie、安全性和隐私

cookie 是可以禁止的，而且可以通过日志分析或其他方式来实现大部分跟踪记录，所以 cookie 自身并不是很大的安全隐患。实际上，可以通过提供一个标准的审查方法在远程数据库中保存个人信息，并将匿名 cookie 作为键值，来降低客户端到服务器的敏感数据传送频率。

但是，潜在的滥用情况总是存在的，所以，在处理隐私和用户跟踪信息时，最好还是要小心一些。第三方 Web 站点使用持久 cookie 来跟踪用户就是一种最大的滥用。将这种做法与 IP 地址和 `Referer` 首部信息结合在一起，这些营销公司就可以构建起相当精确的用户档案和浏览模式信息。

尽管有这么多负面的宣传，人们通常还是认为，如果能够小心地确认在向谁提供私人信息，并仔细查阅站点的隐私政策，那么，cookie 会话处理和事务处理所带来的便利性要比大部分风险更重要。

1998 年，计算机事故咨询能力组织（Computer Incident Advisory Capability）（美国能源部的一部分）编写了一份过分使用 cookie 的风险评估报告。下面是那份报告的摘要。

> CIAC I-034：因特网cookie
>
> （http://www.ciac.org/ciac/bulletins/i-034.shtml）
>
> - 问题
>
> cookie是Web服务器用来识别Web用户的小块数据。关于cookie功能的流行说法和谣言之间的比例已经达到了令人不解的地步，使用户恐惧，使管理者忧心。
>
> - 脆弱性评估
>
> 由于使用Web浏览器cookie使得系统被破坏或窃听，从而带来的系统脆弱性本质上并不存在。cookie只能告知Web服务器你以前是否到过某个网站，并在下次访问时将来自Web服务器的一些短小信息（比如用户编码）回送给它。大部分cookie只会持续到用户退出浏览器为止，然后就会被破坏掉。第二种名为持久cookie的cookie有一个过期日期，会在你的硬盘上存储到那个日期为止。无论用户何时返回一个站点，都可以通过持久cookie来识别其身份，以便跟踪用户的浏览习惯。你来自何处，以及访问过哪些Web页面等信息已经存储在Web服务器的日志文件中了，也可以用这些信息来跟踪用户的浏览习惯，只是使用cookie更简单一些罢了。

11.7 更多信息

这里还有几份有用的资源，介绍了更多与 cookie 有关的信息。

- *Cookies*
  Simon St.Laurent 著，McGraw-Hill 公司出版。

- http://www.ietf.org/rfc/rfc2965.txt
  RFC 2965，“HTTP State Management Mechanism”（“HTTP 状态管理机制”）（废弃了 RFC 2109）。

- http://www.ietf.org/rfc/rfc2964.txt
  RFC 2964，“Use of HTTP State Management”（“HTTP 状态管理的用途”）。

- http://home.netscape.com/newsref/std/cookie_spec.html
  这个经典的网景公司文档“Persistent Client State: HTTP Cookies”（“持久的客户端状态：HTTP Cookies”）描述了现在仍在广泛使用的 HTTP cookie 的最初形式。

276

第12章

基本认证机制

有数百万的人在用 Web 进行私人事务处理，访问私有的数据。通过 Web 可以很方便地访问这些信息，但仅仅是方便访问还是不够的。我们要保证只有特定的人能看到我们的敏感信息并且能够执行我们的特权事务。并不是所有的信息都能够公开发布的。

未授权用户无法查看我们的在线旅游档案，也不能在未经许可的情况下向 Web 站点发布文档，这会让我们感觉舒服一些。我们还要确保，组织中未经授权或不怀好意的成员无法获取那些最敏感的公司计划文档。我们与孩子、配偶以及暗恋对象的私人 Web 通信都是在带有些许隐私保护的情况下进行的，这样我们才能放心。

服务器需要通过某种方式来了解用户身份。一旦服务器知道了用户身份，就可以判定用户可以访问的事务和资源了。认证就意味着要证明你是谁。通常是通过提供用户名和密码来进行认证的。HTTP 为认证提供了一种原生工具。尽管我们可以在 HTTP 的认证形式和 cookie 基础之上“运行自己的”认证工具，但在很多情况下，HTTP 的原生认证功能就可以很好地满足要求。

本章阐述了 HTTP 的认证机制，深入介绍了最常见的 HTTP 认证形式，*基本认证*（basic authentication）。下一章将介绍一种称为*摘要认证*（digest authentication）的功能更强的认证技术。

12.1 认证

认证就是要给出一些身份证明。当出示像护照或驾照那样有照片的身份证件时，就给出了一些证据，说明你就是你所声称的那个人。在自动取款机上输入 PIN 码，或
277 在计算机系统的对话框中输入了密码时，也是在证明你就是你所声称的那个人。

现在，这些策略都不是绝对有效的。密码可以被猜出来或被人偶然听到，身份证件可能被偷去或被伪造，但每种证据都有助于构建合理的信任，说明你就是你所声称的那个人。

12.1.1 HTTP的质询/响应认证框架

HTTP 提供了一个原生的*质询 / 响应*（challenge/response）框架，简化了对用户的认证过程。HTTP 的认证模型如图 12-1 中所示。

Web 应用程序收到一条 HTTP 请求报文时，服务器没有按照请求执行动作，而是以一个“认证质询”进行响应，要求用户提供一些保密信息来说明他是谁，从而对其进行质询。

用户再次发起请求时，要附上保密证书（用户名和密码）。如果证书不匹配，服务器可以再次质询客户端，或产生一条错误信息。如果证书匹配，就可以正常完成请求了。

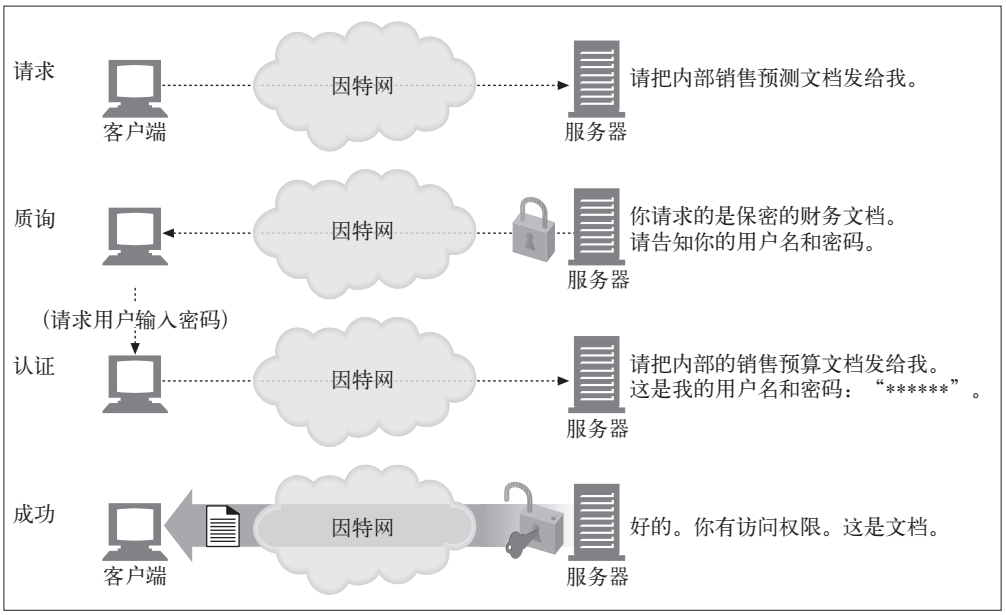


图 12-1　简化的质询 / 响应认证框架

12.1.2　认证协议与首部

HTTP 通过一组可定制的控制首部，为不同的认证协议提供了一个可扩展框架。表 12-1 列出的首部格式和内容会随认证协议的不同而发生变化。认证协议也是在 HTTP 认证首部中指定的。

278

HTTP 定义了两个官方的认证协议：基本认证和摘要认证。今后人们可以随意设计一些使用 HTTP 质询 / 响应框架的新协议。本章的其余部分将解释基本认证机制。摘要认证的细节请参见第 13 章。

表12-1　认证的4个步骤

步骤	首　部	描　述	方法/状态
请求		第一条请求没有认证信息	GET
质询	WWW-Authenticate	服务器用 401 状态拒绝了请求，说明需要用户提供用户名和密码。 服务器上可能会分为不同的区域，每个区域都有自己的密码，所以服务器会在 WWW-Authenticate 首部对保护区域进行描述。 同样，认证算法也是在 WWW-Authenticate 首部中指定的	401 Unauthorized

（续）

步骤	首　部	描　述	方法/状态
授权	Authorization	客户端重新发出请求，但这一次会附加一个 Authorization 首部，用来说明认证算法、用户名和密码	GET
成功	Authentication-Info	如果授权证书是正确的，服务器就会将文档返回。有些授权算法会在可选的 Authentication-Info 首部返回一些与授权会话相关的附加信息	200 OK

为了具体地说明这个问题，我们来看看图 12-2。

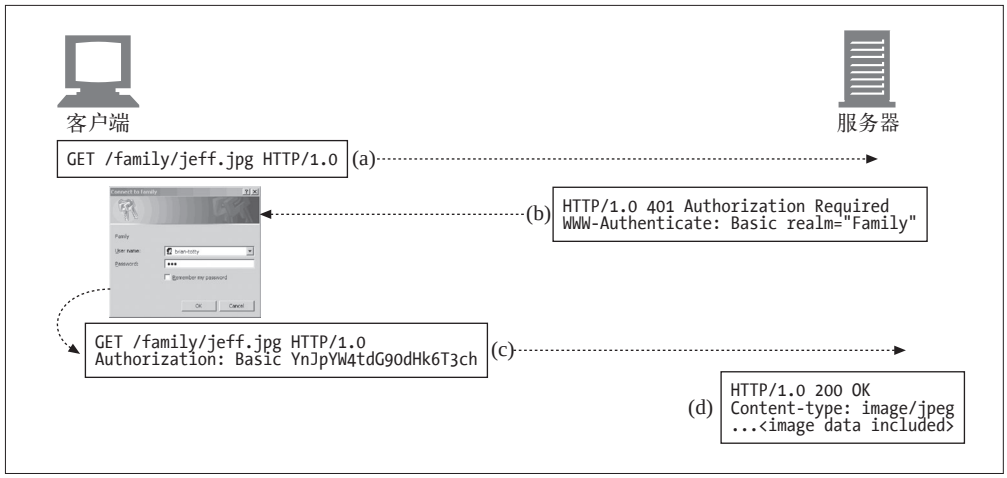


图 12-2　基本认证实例

服务器对用户进行质询时，会返回一条 401 Unauthorized 响应，并在 WWW-Authenticate 首部说明如何以及在哪里进行认证（参见图 12-2b）。

279

当客户端授权服务器继续处理时，会重新发送请求，但会在 Authorization 首部附上加密的密码和其他一些认证参数（参见图 12-2c）。

授权请求成功完成时，服务器会返回一个正常的状态码（比如，200 OK）；对高级认证算法来说，可能还会在 Authentication-Info 首部附加一些额外的信息（参见图 12-2d）。

12.1.3　安全域

在对基本认证的细节进行讨论之前，需要解释一下 HTTP 是怎样允许服务器为

不同的资源使用不同的访问权限的。你可能已经注意到了，图 12-2b 的 WWW-Authenticate 质询中包含了一个 realm 指令。Web 服务器会将受保护的文档组织成一个安全域（security realm）。每个安全域都可以有不同的授权用户集。

比如，假设 Web 服务器建立了两个安全域：一个用于公司的财务信息，另一个用于个人家庭文档（参见图 12-3）。不同的用户对各个安全域的访问权限是不同的。公司的 CEO 应该能够访问销售额预测资料，但不应该允许他访问员工和其家人度假的照片！

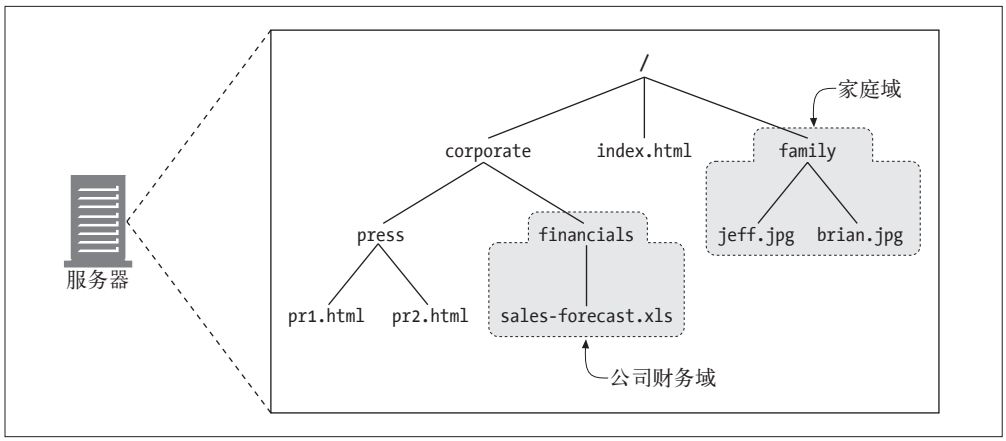


图 12-3　Web 服务器上的安全域

下面是一个假想的基本认证质询，它指定了一个域：

```
HTTP/1.0 401 Unauthorized
WWW-Authenticate: Basic realm="Corporate Financials"
```

域应该有一个描述性的字符名，比如 Corporate Financials（公司财务资料），以帮助用户了解应该使用哪个用户名和密码。在安全域的名称中列出服务器主机名也是很有帮助的——比如，executive-committee@bigcompany.com。

280

12.2　基本认证

基本认证是最流行的 HTTP 认证协议。几乎每个主要的客户端和服务器都实现了基本认证机制。基本认证最初是在 HTTP/1.0 规范中提出的，但此后被移到了 RFC 2617 中，它详细介绍了 HTTP 的认证机制。

在基本认证中，Web 服务器可以拒绝一个事务，质询客户端，请用户提供有效的用户名和密码。服务器会返回 401 状态码，而不是 200 状态码来初始化认证质询，并

用 WWW-Authenticate 响应首部指定要访问的安全域。浏览器收到质询时，会打开一个对话框，请求用户输入这个域的用户名和密码。然后将用户名和密码稍加扰码，再用 Authorization 请求首部回送给服务器。

12.2.1 基本认证实例

图 12-2 是一个详细的基本认证实例。

- 在图 12-2a 中，用户请求了私人家庭相片 /family/jeff.jpg。
- 在图 12-2b 中，服务器回送一条 401 Authorization Required，对私人家庭相片进行密码质询，同时回送的还有 WWW-Authenticate 首部。这个首部请求对 Family 域进行基本认证。
- 在图 12-2c 中，浏览器收到了 401 质询，弹出对话框，询问 Family 域的用户名和密码。用户输入用户名和密码时，浏览器会用一个冒号将其连接起来，编码成“经过扰码的”Base-64 表示形式（下节介绍），然后将其放在 Authorization 首部中回送。
- 在图 12-2d 中，服务器对用户名和密码进行解码，验证它们的正确性，然后用一条 HTTP 200 OK 报文返回所请求的报文。

表 12-2 总结了 HTTP 基本认证的 WWW-Authenticate 和 Authorization 首部。

表12-2 基本认证首部

质询/响应	首部语法及描述
质询（服务器发往客户端）	网站的不同部分可能有不同的密码。域就是一个引用字符串，用来命名所请求的文档集，这样用户就知道该使用哪个密码了： `WWW-Authenticate: Basic realm=quoted-realm`
响应（客户端发往服务器）	用冒号（:）将用户名和密码连接起来，然后转换成 Base-64 编码，这样在用户名和密码中包含国际字符会稍微容易一些，也能尽量避免通过观察网络流量并只进行一些粗略的检查就可以获取用户名和密码情况的发生： `Authorization: Basic base64-username-and-password`

281

注意，基本认证协议并没有使用表 12-1 所示的 Authentication-Info 首部。

12.2.2 Base-64 用户名/密码编码

HTTP 基本认证将（由冒号分隔的）用户名和密码打包在一起，并用 Base-64 编码方式对其进行编码。如果不知道 Base-64 编码是什么意思，也不用担心。你并不需要对它有太多的了解，如果对此感兴趣，可以在附录 E 中读到所有与之有关的内容。简单来说，Base-64 编码会将一个 8 位字节序列划分为一些 6 位的块。用每个 6

位的块在一个特殊的由 64 个字符组成的字母表中选择一个字符，这个字母表中包含了大部分字母和数字。

图 12-4 显示了使用 Base-64 编码的基本认证实例。在这个例子中，用户名为 brian-totty，密码为 Ow!。浏览器用冒号将用户名和密码连接起来，生成一个打包字符串 brian-totty:Ow!。然后对这个字符串进行 Base-64 编码，变成一串乱码：YnJpYW4tdG90dHk6T3ch。

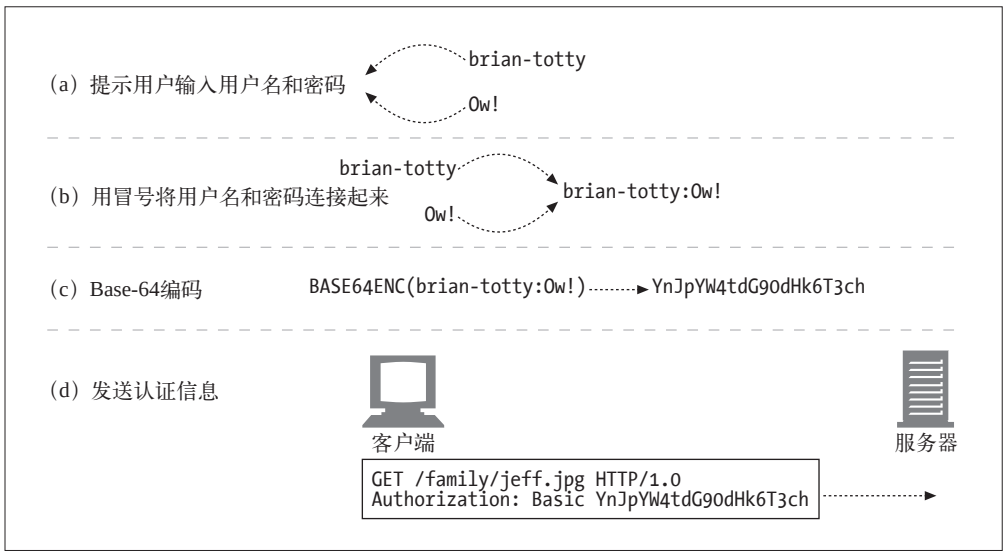


图 12-4　从用户名和密码中生成一个基本认证首部

Base-64 编码可以接受二进制字符串、文本、国际字符表示的数据（在某些系统中会引发一些问题），将其暂时转换成一个易移植的字母表以便传输。然后，在远端就可以解码出原始字符串，而无需担心传输错误了。

有些用户名和密码中会包含国际字符或其他在 HTTP 首部中非法的字符（比如引号、冒号和回车换行符），对这些用户名和密码来说，Base-64 编码是非常有用的。而且，Base-64 编码扰乱了用户名和密码，这样也可以防止管理员在管理服务器和网络时，不小心看到用户名和密码。

282

12.2.3　代理认证

中间的代理服务器也可以实现认证功能。有些组织会在用户访问服务器、LAN 或无线网络之前，用代理服务器对其进行认证。可以在代理服务器上对访问策略进行集中管理，因此，通过代理服务器提供对某组织内部资源的统一访问控制是一种很便捷的

方式。这个过程的第一步就是通过代理认证（proxy authentication）来识别身份。

代理认证的步骤与 Web 服务器身份验证的步骤相同。但首部和状态码都有所不同。表 12-3 列出了 Web 服务器和代理在认证中使用的状态码和首部的差异。

表12-3　Web服务器与代理认证

Web服务器	代理服务器
Unauthorized status code: 401	Unauthorized status code: 407
WWW-Authenticate	Proxy-Authenticate
Authorization	Proxy-Authorization
Authentication-Info	Proxy-Authentication-Info

12.3　基本认证的安全缺陷

基本认证简单便捷，但并不安全。只能用它来防止非恶意用户无意间进行的访问，或将其与 SSL 这样的加密技术配合使用。

基本认证存在下列安全缺陷。

(1) 基本认证会通过网络发送用户名和密码，这些用户名和密码都是以一种很容易解码的形式表示的。实际上，密码是以明文形式传输的，任何人都可以读取并将其捕获。虽然 Base-64 编码通过隐藏用户名和密码，致使友好的用户不太可能在进行网络观测时无意中看到密码，但 Base-64 编码的用户名和密码可以很轻易地通过反向编码过程进行解码，甚至可以用纸笔在几秒钟内手工对其进行解码！所以经过 Base-64 编码的密码实际上就是“明文”传送的。如果有动机的第三方用户有可能会去拦截基本认证发送的用户名和密码，就要通过 SSL 加密信道发送所有的 HTTP 事务，或者使用更安全的认证协议，比如摘要认证。

283

(2) 即使密码是以更难解码的方式加密的，第三方用户仍然可以捕获被修改过的用户名和密码，并将修改过的用户名和密码一次一次地重放给原始服务器，以获得对服务器的访问权。没有什么措施可用来防止这些重放攻击。

(3) 即使将基本认证用于一些不太重要的应用程序，比如公司内部网络的访问控制或个性化内容的访问，一些不良习惯也会让它变得很危险。很多用户由于受不了大量密码保护的服务，会在这些服务间使用相同的用户名和密码。比如说，某个狡猾的恶徒会从免费的因特网邮件网站捕获明文形式的用户名和密码，然后会发现用同样的用户名和密码还可以访问重要的在线银行网站！

(4) 基本认证没有提供任何针对代理和作为中间人的中间节点的防护措施，它们没有修改认证首部，但却修改了报文的其余部分，这样就严重地改变了事务的本质。

(5) 假冒服务器很容易骗过基本认证。如果在用户实际连接到一台恶意服务器或网关的时候，能够让用户相信他连接的是一个受基本认证保护的合法主机，攻击者就可以请求用户输入密码，将其存储起来以备未来使用，然后捏造一条错误信息传送给用户。

这一切说明，在友好的环境，或者说是希望有隐私保护但隐私保护并不十分必要的环境中，可以通过基本认证来提供便捷的文档个性化服务或访问控制保护。通过这种方式，可以用基本认证来防止一些好奇的用户无意中或不小心对文档进行访问。[1]

比如，在一个公司内部，产品管理可能要对未来的产品计划进行密码保护，以防止信息的过早发布。对一般用户而言，基本认证就足以让他们感到不便而不会再去访问这些数据了。[2] 同样，你可能会用密码来保护那些并非高度机密的，或者没什么信息价值的私人照片或私有站点，这些信息确实和其他人也没什么关系。

将基本认证与加密数据传输（比如 SSL）配合使用，向恶意用户隐藏用户名和密码，会使基本认证变得更加安全。这是一种常用的技巧。

我们会在第 14 章讨论安全加密技术。下一章将介绍更复杂的 HTTP 认证协议——摘要认证，摘要认证具有比基本认证更强的安全特性。 284

12.4 更多信息

更多与基本认证和 LDAP 有关的信息，请参见以下资源。

- http://www.ietf.org/rfc/rfc2617.txt
  RFC 2617，“HTTP Authentication：Basic and Digest Access Authentication.”（“HTTP 认证：基本和摘要访问认证”）

- http://www.ietf.org/rfc/rfc2616.txt 285
  RFC 2616，“Hypertext Transfer Protocol-HTTP/1.1.”（“超文本传输协议——HTTP/1.1”。）

注 1：小心，基本认证中使用的用户名和密码要有别于你在更安全的系统中所使用的密码，否则恶意用户就可以用它们来攻破你的安全账户了！

注 2：尽管不是非常安全，但公司内部的员工通常也没有太大的动力去恶意捕获这些密码。这也说明，公司确实会有间谍，也确实会有不满，想要报复的员工，所以，明智的做法是对一旦被恶意获取就会造成很大损害的数据应用更安全的策略。

第13章

摘要认证

基本认证便捷灵活，但极不安全。用户名和密码都是以明文形式传送的，[1] 也没有采取任何措施防止对报文的篡改。安全使用基本认证的唯一方式就是将其与 SSL 配合使用。

摘要认证与基本认证兼容，但却更为安全。本章主要介绍摘要认证的原理和实际应用。尽管摘要认证还没有得到广泛应用，但对实现安全事务来说，这些概念是非常重要的。

13.1 摘要认证的改进

摘要认证是另一种 HTTP 认证协议，它试图修复基本认证协议的严重缺陷。具体来说，摘要认证进行了如下改进。

- 永远不会以明文方式在网络上发送密码。
- 可以防止恶意用户捕获并重放认证的握手过程。
- 可以有选择地防止对报文内容的篡改。
- 防范其他几种常见的攻击方式。

摘要认证并不是最安全的协议。[2] 摘要认证并不能满足安全 HTTP 事务的很多需求。对这些需求来说，使用传输层安全（Transport Layer Security，TLS）和安全 HTTP
286 （Secure HTTP，HTTPS）协议更为合适一些。

但摘要认证比它要取代的基本认证强大很多。与很多建议其他因特网服务使用的常用策略相比，（比如曾建议 LDAP、POP 和 IMAP 使用的 CRAM-MD5），摘要认证也要强大很多。

迄今为止，摘要认证还没有被广泛应用。但由于基本认证存在固有的安全风险，HTTP 设计者曾在 RFC 2617 中建议：“在可行的情况下应该将目前在用的所有使用基本认证的服务，尽快地转换为摘要认证方式。”[3] 这个标准的前景还不太明朗。

13.1.1 用摘要保护密码

摘要认证遵循的箴言是“绝不通过网络发送密码”。客户端不会发送密码，而是会发送一个“指纹”或密码的“摘要”，这是密码的不可逆扰码。客户端和服务器都知道这个密码，因此服务器可以验证所提供的摘要是否与密码相匹配。只拿到摘要的话，除了将所有的密码都拿来试试之外，没有其他方法可以找出摘要是来自哪个密

注 1：用户名和密码用 Base-64 编码进行了扰码，但很容易被解码。只能防止无意中的查看，没有任何防止恶意用户攻击的手段。

注 2：比如，与基于公有密钥的机制相比，摘要认证所提供的认证机制就不够强。同样，摘要认证除了能保护密码外，并没有提供保护其他内容的方式——请求和应答中的其余部分仍然可能被窃听。

注 3：随着 SSL 加密 HTTP 的流行和广泛采用，有关摘要认证的现实意义曾有过很激烈的争论。时间将会告诉我们摘要认证能否达到所需的规模。

码的！[4]

下面来看看摘要认证的工作原理（这是一个简化版本）。

- 在图 13-1a 中，客户端请求了某个受保护文档。
- 在图 13-1b 中，在客户端能够证明其知道密码从而确认其身份之前，服务器拒绝提供文档。服务器向客户端发起质询，询问用户名和摘要形式的密码。
- 在图 13-1c 中，客户端传递了密码的摘要，证明它是知道密码的。服务器知道所有用户的密码，[5] 因此可以将客户提供的摘要与服务器自己计算得到的摘要进行比较，以验证用户是否知道密码。另一方在不知道密码的情况下，很难伪造出正确的摘要。
- 在图 13-1d 中，服务器将客户端提供的摘要与服务器内部计算出的摘要进行对比。如果匹配，就说明客户端知道密码（或者很幸运地猜中了！）。可以设置摘要函数，使其产生很多数字，让人不可能幸运地猜中摘要。服务器进行了匹配验证之后，会将文档提供给客户端——整个过程都没有在网络上发送密码。

287

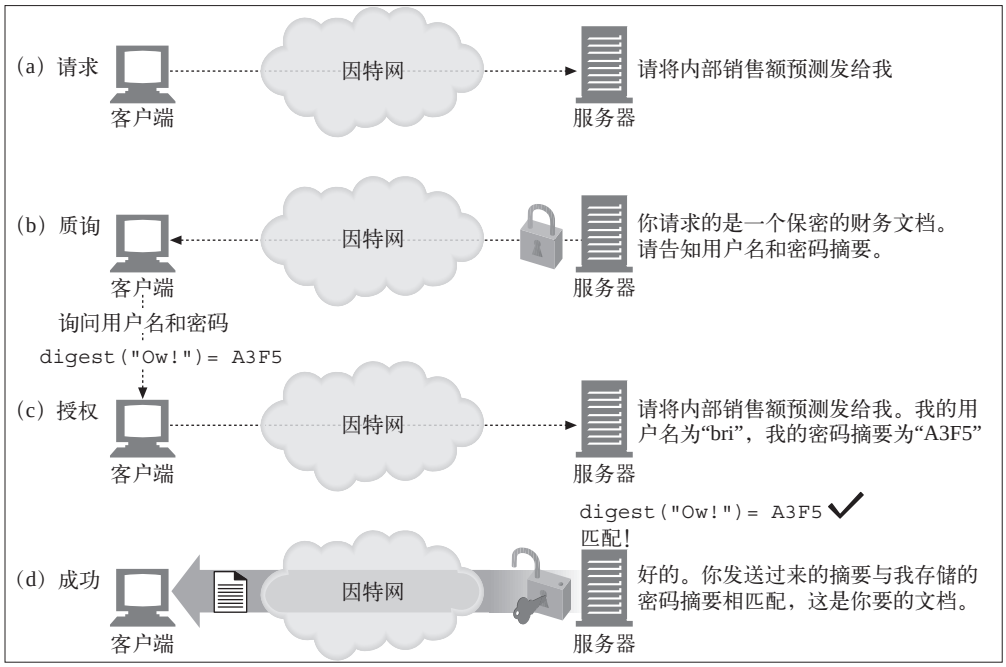


图 13-1　用摘要来实现隐藏密码的认证

注 4：有一些技术，比如词典攻击，会首先尝试一些常见的密码。这些密码分析技术可以极大地简化密码破译进程。

注 5：实际上，服务器只需要知道密码的摘要即可。

我们将在表 13-8 中更详细地讨论摘要认证中那些特殊的首部。

13.1.2 单向摘要

摘要是“对信息主体的浓缩”。[6] 摘要是一种单向函数，主要用于将无限的输入值转换为有限的浓缩输出值。[7] 常见的摘要函数 MD5，[8] 会将任意长度的字节序列转换为一个 128 位的摘要。

128 位 = 2^{128}，或者大约

1 000 000 000 000 000 000 000 000 000 000 000 000 000 种不同的输出值。

对这些摘要来说，最重要的是如果不知道密码的话，要想正确地猜出发送给服务器的摘要将是非常困难的。同样，如果有摘要，想要判断出它是由无数输入值中的哪一个产生的，也是非常困难的。

MD5 输出的 128 位的摘要通常会被写成 32 个十六进制的字符，每个字符表示 4 位。表 13-1 给出了几个示例输入的 MD5 摘要。注意 MD5 是怎样根据任意的输入产生定长的摘要输出的。

表13-1 MD5摘要实例

输　入	MD5摘要
"Hi"	C1A5298F939E87E8F962A5EDFC206918
"bri:Ow!"	BEAAA0E34EBDB072F8627C038AB211F8
"3.1415926535897"	475B977E19ECEE70835BC6DF46F4F6DE
"http://www.http-guide.com/index.htm"	C617C0C7D1D05F66F595E22A4B0EAAA5
"WE hold these Truths to be self-evident, that all Men are created equal, that they are endowed by their Creator with certain unalienable Rights, that among these are Life, Liberty and the Pursuit of Happiness—That to secure these Rights, Governments are instituted among Men, deriving their just Powers from the Consent of the Governed, that whenever any Form of Government becomes destructive of these Ends, it is the Right of the People to alter or to abolish it, and to institute new Government, laying its Foundation on such Principles, and organizing its Powers in such Form, as to them shall seem most likely to effect their Safety and Happiness."	66C4EF58DA7CB956BD04233FBB64E0A4

注 6：韦氏词典，1998 年。

注 7：理论上来讲，我们将数量无限的输入值转换成了数量有限的输出值，所以两个不同的输入值就可能映射为同一个摘要。这种情况被称为*冲突*（collision）。实际上，由于可用输出值的数量足够大，所以在现实生活中，出现冲突的可能是微乎其微的，对我们要实现的密码匹配来说并不重要。

注 8：MD5 表示“报文摘要的第五版”，是摘要算法系列中的一种。*安全散列算法*（Secure Hash Algorithm，SHA）是另一种常见的摘要函数。

有时也将摘要函数称为加密的校验和、单向散列函数或指纹函数。

13.1.3 用随机数防止重放攻击

使用单向摘要就无需以明文形式发送密码了。可以只发送密码的摘要，而且可以确信，没有哪个恶意用户能轻易地从摘要中解码出原始密码。

但是，仅仅隐藏密码并不能避免危险，因为即便不知道密码，别有用心的人也可以截获摘要，并一遍遍地重放给服务器。摘要和密码一样好用。

为防止此类重放攻击的发生，服务器可以向客户端发送一个称为*随机数*（nonce）[9]的特殊令牌，这个数会经常发生变化（可能是每毫秒，或者是每次认证都变化）。客户端在计算摘要之前要先将这个随机数令牌附加到密码上去。

289

在密码中加入随机数就会使摘要随着随机数的每一次变化而变化。记录下的密码摘要只对特定的随机值有效，而没有密码的话，攻击者就无法计算出正确的摘要，这样就可以防止重放攻击的发生。

摘要认证要求使用随机数，因为这个小小的重放弱点会使未随机化的摘要认证变得和基本认证一样脆弱。随机数是在 `WWW-Authenticate` 质询中从服务器传送给客户端的。

13.1.4 摘要认证的握手机制

HTTP 摘要认证协议是一种升级版的认证方式，所用首部与基本认证类似。它在传统首部中添加了一些新的选项，还添加了一个新的可选首部 `Authorization-Info`。

图 13-2 描述了简化的摘要认证三步握手机制。

图 13-2 中发生的情况如下所述。

- 在第（1）步中，服务器会计算出一个随机数。在第（2）步中，服务器将这个随机数放在 `WWW-Authenticate` 质询报文中，与服务器所支持的算法列表一同发往客户端。

290

- 在第（3）步中，客户端选择一个算法，计算出密码和其他数据的摘要。在第（4）步中，将摘要放在一条 `Authorization` 报文中发回服务器。如果客户端要对服务器进行认证，可以发送客户端随机数。
- 在第（5）步中，服务器接收摘要、选中的算法以及支撑数据，计算出与客户端相同的摘要。然后服务器将本地生成的摘要与网络传送过来的摘要进行比较，验

注 9：随机数这个词表示“本次”或“临时的”。在计算机安全的概念中，随机数捕获了一个特定的时间点，将其加入到安全计算之中。

证其是否匹配。如果客户端反过来用客户端随机数对服务器进行质询，就会创建客户端摘要。服务器可以预先将下一个随机数计算出来，提前将其传递给客户端，这样下一次客户端就可以预先发送正确的摘要了。

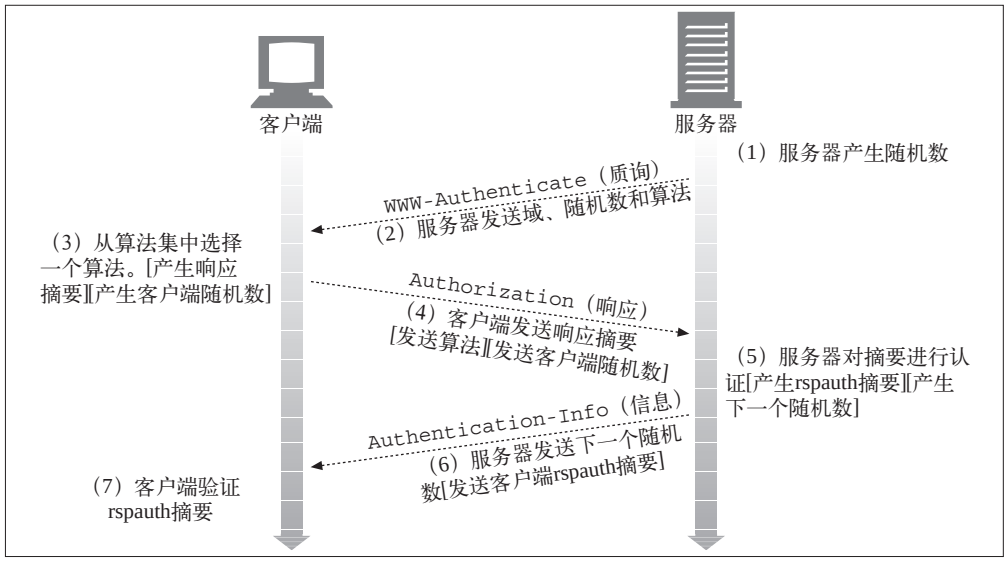


图 13-2　摘要认证的握手机制

这些信息中很多是可选的，而且有默认值。为了说明问题，图 13-3 对比了基本认证中发送的报文（参见图 13-3a 至图 13-3d）与简单的摘要认证实例发送的报文（参见图 13-3e 至图 13-3h）。

现在我们来更详细地探讨摘要认证的内部工作原理。

13.2　摘要的计算

摘要认证的核心就是对公共信息、保密信息和有时限的随机值这个组合的单向摘要。现在我们来看看这些摘要是如何计算出来的。摘要计算通常都是简单易懂的。[10] 附录 F 提供了示例源代码。

13.2.1　摘要算法的输入数据

摘要是根据以下三个组件计算出来的。

- 由单向散列函数 H(*d*) 和摘要 KD(*s*,*d*) 组成的一对函数，其中 *s* 表示密码，*d* 表示数据。

注 10：但对初学者来说，可选的 RFC 2617 兼容模式以及规范中背景资料的缺乏，使其变得有些复杂。我们会努力提供一些帮助。

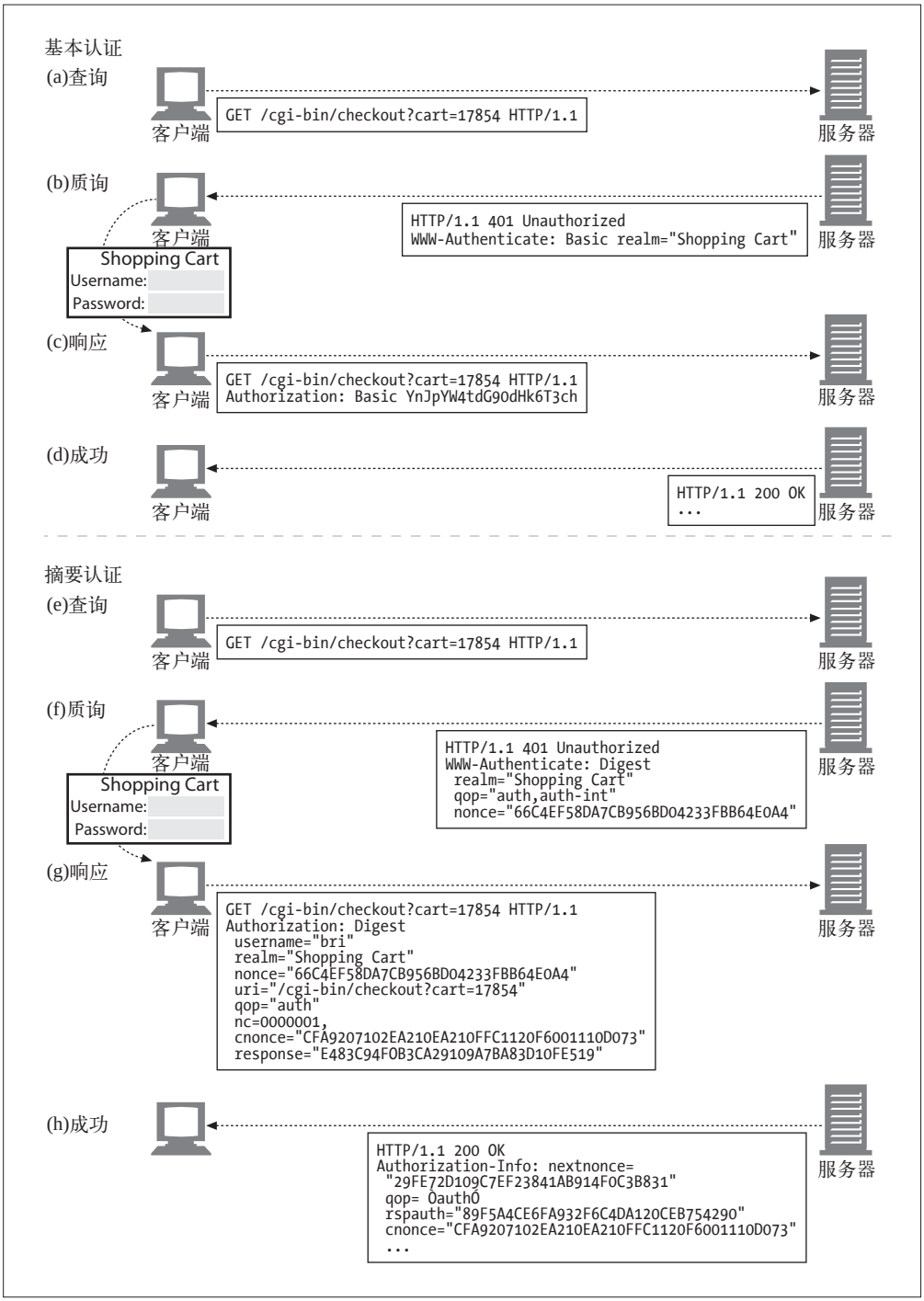


图 13-3　基本认证与摘要认证的语法对比

- 一个包含了安全信息的数据块，包括密码，称为 A1。
- 一个包含了请求报文中非保密属性的数据块，称为 A2。

H 和 KD 处理两块数据 A1 和 A2，产生摘要。

13.2.2 算法H(*d*)和KD(*s*,*d*)

291 ~ 292

摘要认证支持对各种摘要算法的选择。RFC 2617 建议的两种算法为 MD5 和 MD5-sess（“sess”表示会话），如果没有指定其他算法，默认算法为 MD5。

不管使用的是 MD5 还是 MD5-sess，都会用函数 H 来计算数据的 MD5，用摘要函数 KD 来计算以冒号连接的密码和非保密数据的 MD5。例如：

```
H(<data>) = MD5(<data>)
KD(<secret>,<data>) = H(concatenate(<secret>:<data>))
```

13.2.3 与安全性相关的数据（A1）

被称为 A1 的数据块是密码和受保护信息的产物，它包含有用户名、密码、保护域和随机数等内容。A1 只涉及安全信息，与底层报文自身无关。A1 会与 H、KD 和 A2 一同用于摘要计算。

RFC 2617 根据选择的算法定义了两种计算 A1 的方式。

- MD5

为每条请求运行单向散列函数。A1 是由冒号连接起来的用户名、域以及密码三元组。

- MD5-sess

只在第一次 `WWW-Authenticate` 握手时运行一次散列函数。对用户名、域和密码进行一次 CPU 密集型散列，并将其放在当前随机数和客户端随机数（cnonce）的前面。

表 13-2 显示了 A1 的定义。

表13-2 算法对A1的定义

算法	A1
MD5	`A1 = <user>:<realm>:<password>`
MD5-sess	`A1 = MD5(<user>:<realm>:<password>):<nonce>:<cnonce>`

13.2.4 与报文有关的数据（A2）

数据块 A2 表示的是与报文自身有关的信息，比如 URL、请求方法和报文实体的主

体部分。A2 有助于防止方法、资源或报文被篡改。A2 会与 H、KD 和 A1 一起用于摘要的计算。

RFC 2617 根据所选择的保护质量（qop），为 A2 定义了两种策略。

- 第一种策略只包含 HTTP 请求方法和 URL。当 qop="auth" 时使用这种策略，这是默认的情况。
- 第二种策略添加了报文实体的主体部分，以提供一定程度的报文完整性检测。qop="auth-int" 时使用。

表 13-3 显示了 A2 的定义。

表13-3　算法对A2的定义（请求摘要）

qop	A2
未定义	<request-method>:<uri-directive-value>
auth	<request-method>:<uri-directive-value>
auth-int	<request-method>:<uri-directive-value>:H(<request-entity-body>)

request-method 是 HTTP 的请求方法。uri-directive-value 是请求行中的请求 URI。可能是个 "*"、absoluteURL 或者 abs_path，但它必须与请求 URI 一致。尤其需要注意的是，如果请求 URI 是 absoluteURL，它必须是个绝对 URL。

13.2.5　摘要算法总述

RFC 2617 定义了两种给定了 H、KD、A1 和 A2 之后，计算摘要的方式。

- 第一种方式要与老规范 RFC 2069 兼容，在没有 qop 选项的时候使用。它是用保密信息和随机报文数据的散列值来计算摘要的。
- 第二种方式是现在推荐使用的方式——这种方式包含了对随机数计算和对称认证的支持。只要 qop 为 auth 或 auth-int，就要使用这种方式。它向摘要中添加了随机计数、qop 和 cnonce 数据。

表 13-4 给出了得到的摘要函数定义。注意得到的摘要使用了 H、KD、A1 和 A2。

表13-4　新/老摘要算法

qop	摘要算法	备　注
未定义	KD(H(A1), <nonce>:H(A2))	不推荐
auth 或 auth-int	KD(H(A1), <nonce>:<nc>:<cnonce>:<qop>:H(A2))	推荐

这些派生封装层很容易把人弄晕。这也是有些读者觉得 RFC 2617 难懂的原因之一。为了简化，表 13-5 扩展了 H 和 KD 的定义，用 A1 和 A2 来表示摘要。

表13-5　展开的摘要算法备忘单

qop	算　法	展开的算法
未定义	<undefined> MD5 MD5-sess	MD5(MD5(A1):<nonce>:MD5(A2))
auth	<undefined> MD5 MD5-sess	MD5(MD5(A1):<nonce>:<nc>:<cnonce>:<qop>:MD5(A2))
auth-int	<undefined> MD5 MD5-sess	MD5(MD5(A1):<nonce>:<nc>:<cnonce>:<qop>:MD5(A2))

13.2.6　摘要认证会话

客户端响应对保护空间的 WWW-Authenticate 质询时，会启动一个此保护空间的认证会话（与受访问服务器的标准根结合在一起的域就定义了一个“保护空间”）。

在客户端收到另一条来自保护空间的任意一台服务器的 WWW-Authenticate 质询之前，认证会话会一直持续。客户端应该记住用户名、密码、随机数、随机数计数以及一些与认证会话有关的隐晦值，以便将来在此保护空间中构建请求的 Authorization 首部时使用。

随机数过期时，即便老的 Authorization 首部所包含的随机数不再新鲜了，服务器也可以选择接受其中的信息。服务器也可以返回一个带有新随机数的 401 响应，让客户端重试这条请求；指定这个响应为 stale=true，表示服务器在告知客户端用新的随机数来重试，而不再重新提示输入新的用户名和密码了。

13.2.7　预授权

在普通的认证方式中，事务结束之前，每条请求都要有一次请求 / 质询的循环，参见图 13-4a。

如果客户端事先知道下一个随机数是什么，就可以取消这个请求 / 质询循环，这样客户端就可以在服务器发出请求之前，生成正确的 Authorization 首部了。如果客户端能在服务器要求它计算 Authorization 首部之前将其计算出来，就可以预先将 Authorization 首部发送给服务器，而不用进行请求 / 质询了。图 13-4b 显示了这种方式对性能的影响。

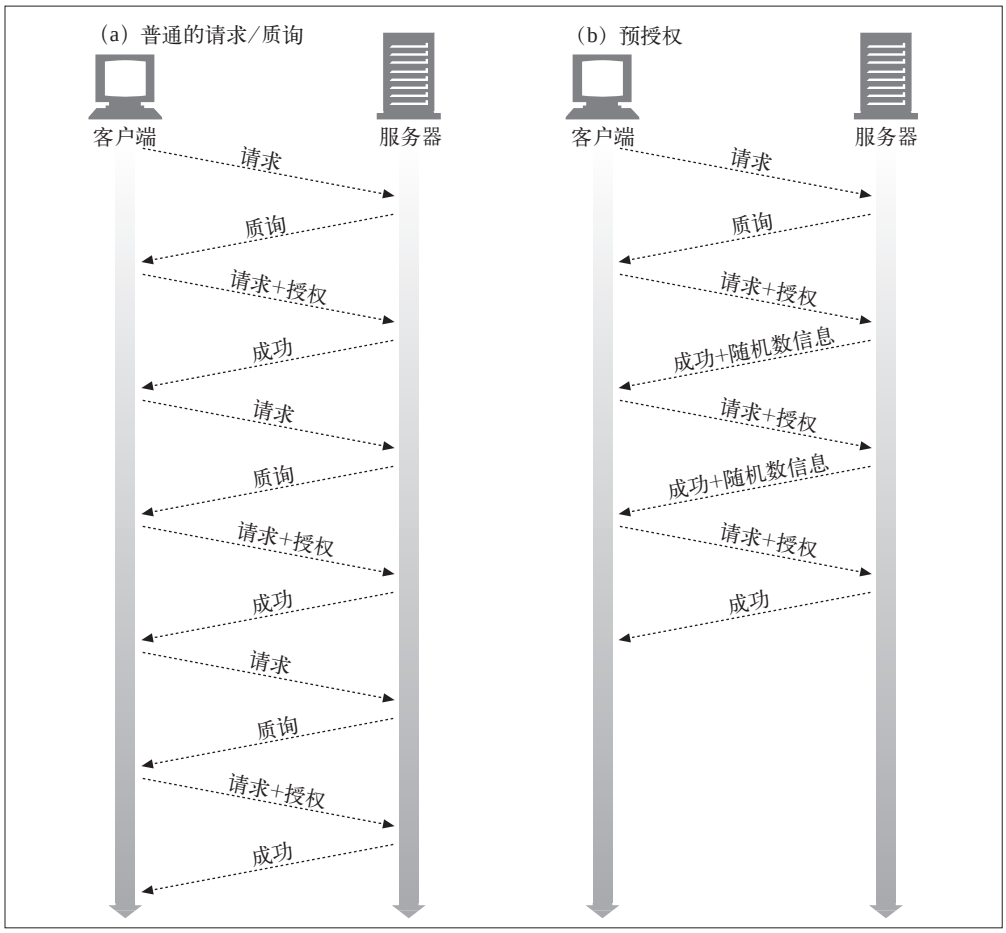


图 13-4　预授权减少了报文的数量

预授权对基本认证来说并不重要（而且很常见）。浏览器通常会维护一些客户端数据库以存储用户名和密码。一旦用户与某站点进行了认证，浏览器通常会为后继对那个 URL 的请求发送正确的 Authorization 首部（参见第 12 章）。 295

由于摘要认证使用了随机数技术来破坏重放攻击，所以对摘要认证来说，预授权要稍微复杂一些。服务器会产生任意的随机数，所以在客户端收到质询之前，不一定总能判定应该发送什么样的 Authorization 首部。

摘要认证在保留了很多安全特性的同时，还提供了几种预授权方式。这里列出了三种可选的方式，通过这些方式，客户端无需等待新的 WWW-Authenticate 质询，就可以获得正确的随机数：

- 服务器预先在 Authentication-Info 成功首部中发送下一个随机数；

- 服务器允许在一小段时间内使用同一个随机数；
296 - 客户端和服务器使用同步的、可预测的随机数生成算法。

1. 预先生成下一个随机数

可以在 Authentication-Info 成功首部中将下一个随机数预先提供给客户端。这个首部是与前一次成功认证的 200 OK 响应一同发送的。

```
Authentication-Info: nextnonce="<nonce-value>"
```

有了下一个随机数，客户端就可以预先发布 Authorization 首部了。

尽管这种预授权机制避免了请求 / 质询循环（加快了事务处理的速度），但实际上它也破坏了对同一台服务器的多条请求进行管道化的功能，因为在发布下一条请求之前，一定要收到下一个随机值才行。而管道化是避免延迟的一项基本技术，所以这样可能会造成很大的性能损失。

2. 受限的随机数重用机制

另一种方法不是预先生成随机数序列，而是在有限的次数内重用随机数。比如，服务器可能允许将某个随机数重用 5 次，或者重用 10 秒。

在这种情况下，客户端可以随意发布带有 Authorization 首部的请求，而且由于随机数是事先知道的，所以还可以对请求进行管道化。随机数过期时，服务器要向客户端发送 401 Unauthorized 质询，并设置 WWW-Authenticate:stale=true 指令：

```
WWW-Authenticate: Digest
    realm="<realm-value>"
    nonce="<nonce-value>"
    stale=true
```

重用随机数使得攻击者更容易成功地实行重放攻击。虽然这确实降低了安全性，但重用的随机数的生存期是可控的（从严格禁止重用到较长时间的重用），所以应该可以在安全和性能间找到平衡。

此外，还可以通过其他一些特性使重放攻击变得更加困难，其中就包括增量计数器和 IP 地址测试。但这些技术只能使攻击的实施更加麻烦，并不能消除由此带来的安全隐患。

3. 同步生成随机数

还可以采用时间同步的随机数生成算法，客户端和服务器可根据共享的密钥，生成第三方无法轻易预测的、相同的随机数序列（比如安全 ID 卡）。

297 这些算法都超出了摘要认证规范的范畴。

13.2.8 随机数的选择

随机数的内容不透明，而且与实现有关。但性能、安全性和便捷性的优劣都取决于明智的选择。

RFC 2617 建议采用这个假想的随机数公式：

```
BASE64(time-stamp H(time-stamp ":" ETag ":" private-key))
```

其中 time-stamp 是服务器产生的时间或其他不会重复的值，ETag 是与所请求实体有关的 HTTP ETag 首部的值，private-key 是只有服务器知道的数据。

有了这种形式的随机数，服务器就可以在收到客户端的认证首部之后重新计算散列部分，如果结果与那个首部的随机数不符，或者时间戳的值不够新，就拒绝请求。服务器可以通过这种方式来限制随机数的有效持续时间。

包含 Etag 可以防止对已更新资源版本的重放请求。（注意，在随机数中包含客户端的 IP 地址，服务器好像就可以限制原来获得此随机数的客户端重用这个随机数了，但这会破坏代理集群的工作。使用代理集群时，来自单个用户的多条请求通常会经过不同的代理进行传输，而且 IP 地址欺骗实现起来也不是很难。）

实现可以选择不接受以前使用过的随机数或摘要，以防止重放攻击。实现也可以选择为 POST 或 PUT 请求使用一次性的随机数或摘要，为 GET 请求使用时间戳。

会影响到随机数选取的一些实际安全问题参见 13.5 节。

13.2.9 对称认证

RFC 2617 扩展了摘要认证机制，允许客户端对服务器进行认证。这是通过提供客户端随机值来实现的，服务器会根据它对共享保密信息的正确了解生成正确的响应摘要。然后，服务器在 Authorization-Info 首部中将此摘要返回给客户端。

这种对称认证方式被标准化为 RFC 2617。为了与原有 RFC 2069 标准后向兼容，它是可选的，但由于它提供了一些重要的安全提升机制，强烈推荐现今所有的客户端和服务器都要实现全部 RFC 2617 特性。特别是，只要提供了 qop 指令，就要求执行对称认证，而没有 qop 指令时则不要求执行对称认证。

响应摘要的计算方法与请求摘要类似，但由于响应中没有方法，而且报文实体数据有所不同，所以只有报文主体信息 A2 不同。表 13-6 和表 13-7 对比了请求和响应摘要中 A2 的计算方法。

298

表13-6　算法中A2的定义（请求摘要）

qop	A2
未定义	<request-method>:<uri-directive-value>
auth	<request-method>:<uri-directive-value>
auth-int	<request-method>:<uri-directive-value>:H(<request-entity-body>)

表13-7　算法中A2的定义（响应摘要）

qop	A2
未定义	:<uri-directive-value>
auth	:<uri-directive-value>
auth-int	:<uri-directive-value>:H(<response-entity-body>)

cnonce 值和 nc 值必须是本报文所响应的客户端请求中的相应值。如果指定了 qop="auth" 或 qop="auth-int"，就必须提供响应 auth、cnonce 和 nonce 计数指令。

13.3　增强保护质量

可以在三种摘要首部中提供 qop 字段：WWW-Authenticate、Authorization 和 Authentication-Info。

通过 qop 字段，客户端和服务器可以对不同类型及质量的保护进行协商。比如，即便会严重降低传输速度，有些事务可能也要检查报文主体的完整性。

服务器首先在 WWW-Authenticate 首部输出由逗号分隔的 qop 选项列表。然后客户端从中选择一个它支持且满足其需求的选项，并将其放在 Authorization 的 qop 字段中回送给服务器。

qop 字段是可选的，但只是在后向兼容原有 RFC 2069 规范的情况下才是可选的。现代所有的摘要实现都应该支持 qop 选项。

RFC 2617 定义了两种保护质量的初始值：表示认证的 auth，带有报文完整性保护的认证 auth-int。将来可能还会出现其他 qop 选项。

13.3.1　报文完整性保护

如果使用了完整性保护（qop="auth-int"），H（实体的主体部分）就是对实体主体部分，而不是报文主体部分的散列。对于发送者，要在应用任意传输编码方式之前计算；而对于接收者，则应在去除所有传输编码之后计算。注意，对于任何含有多部份的内容类型来说，多部分的边界和每部分中嵌入的首部都要包含在内。

299

13.3.2 摘要认证首部

基本认证和摘要认证协议都包含了 WWW-Authenticate 首部承载的授权质询、Authorization 首部承载的授权响应。摘要认证还添加了可选的 Authorization-Info 首部，这个首部是在成功认证之后发送的，用于实现三步握手机制，并传送下一个随机数。表 13-8 给出了基本认证和摘要认证的首部。

表13-8 HTTP认证首部

阶段	基 本	摘 要
质询	WWW-Authenticate: Basic realm="<realm-value>"	WWW-Authenticate: Digest realm="<realm-value>" nonce="<nonce-value>" [domain="<list-of-URIs>"] [opaque="<opaque-token-value>"] [stale=<true-or-false>] [algorithm=<digest-algorithm>] [qop="<list-of-qop-values>"] [<extension-directive>]
响应	Authorization: Basic <base64(user:pass)>	Authorization: Digest username="<username>" realm="<realm-value>" nonce="<nonce-value>" uri=<request-uri> response="<32-hex-digit-digest>" [algorithm=<digest-algorithm>] [opaque="<opaque-token-value>"] [cnonce="<nonce-value>"] [qop=<qop-value>] [nc=<8-hex-digit-nonce-count>] [<extension-directive>]
Info	n/a	Authentication-Info: nextnonce="<nonce-value>" [qop="<list-of-qop-values>"] [rspauth="<hex-digest>"] [cnonce="<nonce-value>"] [nc=<8-hex-digit-nonce-count>]

摘要认证首部要复杂得多。详细介绍参见附录 F。

13.4 应该考虑的实际问题

使用摘要认证时需要考虑几件事情。本节讨论了其中一些问题。 300

13.4.1 多重质询

服务器可以对某个资源发起多重质询。比如，如果服务器不了解客户端的能力，就可以既提供基本认证质询，又提供摘要认证质询。客户端面对多重质询时，必须以它所支持的最强的质询机制来应答。

质询自身可能会包含由逗号分隔的认证参数列表。如果 WWW-Authenticate 或 Proxy-Authenticate 首部包含了多个质询，或者提供了多个 WWW-Authenticate 首部，用户 Agent 代理在解析 WWW-Authenticate 或 Proxy-Authenticate 首部字段值时就要特别小心。注意，很多浏览器只支持基本认证，要求这是提交给它的第一种认证机制。

在提供了认证选项范围的情况下，安全问题上就会存在明显的“最薄弱环节”。只有当基本认证是最低可接受认证方式时，服务器才应该包含它，而且管理员还应该警告用户，即使运行了不同层次安全措施，系统间使用相同密码也存在一定危险性。

13.4.2 差错处理

在摘要认证中，如果某个指令或其值使用不当，或者缺少某个必要指令，就应该使用响应 400 Bad Request。

如果请求的摘要不匹配，就应该记录一次登录失败。某客户端连续多次失败可能说明有攻击者正在猜测密码。

认证服务器一定要确保 URI 指令指定的资源与请求行中指定的资源相同。如果不同，服务器就应该返回 400 Bad Request 错误。（这可能是一种攻击的迹象，因此服务器设计者可能会考虑将此类错误记录下来。）这个字段包含的内容与请求 URL 中的内容是重复的，用来应对中间代理可能对客户端请求进行的修改。这个经过修改（但估计语义是等价的）的请求计算后得到的摘要可能会与客户端计算出的摘要有所不同。

13.4.3 保护空间

域值，与被访问服务器的标准根 URL 结合在一起，定义了保护空间。

通过域可以将服务器上的受保护资源划分为一组保护空间，每个空间都有自己的认
301 证机制和 / 或授权数据库。域值是一个字符串，通常由原始服务器分配，可能会有认证方案特有的附加语义。注意，可能会有多个授权方案相同，而域不同的质询。

保护空间确定了可以自动应用证书的区域。如果前面的某条请求已被授权，在一段

时间内，该保护空间中所有其他请求都可以重用同一个证书，时间的长短由认证方案、参数和/或用户喜好来决定。除非认证方案进行了其他定义，否则单个保护空间是不能扩展到其服务器范围之外的。

对保护空间的具体计算取决于认证机制。

- 在基本认证中，客户端会假定请求URI中或其下的所有路径都与当前的质询处于同一个保护空间内。客户端可以预先提交对此空间中资源的认证，无需等待来自服务器的另一条质询。
- 在摘要认证中，质询的WWW-Authenticate:domain字段对保护空间作了更精确的定义。domain字段是一个用引号括起来的、中间由空格分隔的URI列表。通常认为，domain列表中的所有URI和逻辑上处于这些前缀之下的所有URI，都位于同一个保护空间中。如果没有domain字段，或者此字段为空，质询服务器上的所有URI就都在保护空间内。

13.4.4 重写URI

代理可以通过改变URI语法，而不改变所描述的实际资源的方式来重写URI。比如：

- 可以对主机名进行标准化，或用IP地址来取代；
- 可以用“%”转义形式来取代嵌入的字符；
- 如果某类型的一些附加属性不会影响从特定原始服务器上获取资源，就可以将其附加或插入到URI中。

代理可修改URI，而且摘要认证会检查URI值的完整性，所以如果进行了任意一种修改，摘要认证就会被破坏。更多信息参见13.2.4节。

13.4.5 缓存

共享的缓存收到包含Authorization首部的请求和转接那条请求产生的响应时，除非响应中提供了下列两种Cache-Control指令之一，否则一定不能将那条响应作为对任何其他请求的应答使用。

- 如果原始响应中包含有Cache-Control指令must-revalidate，缓存可以在应答后继请求时使用那条响应的实体部分。但它首先要用新请求的请求首部，与原始服务器再次进行验证，这样原始服务器就可以对新请求进行认证了。
- 如果原始响应中包含有Cache-Control指令public，在对任意后继请求的应答中都可以返回响应的实体部分。

302

13.5 安全性考虑

RFC 2617 总结了 HTTP 认证方案固有的一些安全风险，这是很令人钦佩的。本节描述了其中的部分风险。

13.5.1 首部篡改

为了提供一个简单明了的防首部篡改系统，要么就得进行端到端的加密，要么就得对首部进行数字签名——最好是两者的结合！摘要认证的重点在于提供一种防篡改认证机制，但并不一定要将这种保护扩展到数据上去。具有一定保护级别的首部只有 `WWW-Authenticate` 和 `Authorization`。

13.5.2 重放攻击

在当前的上下文中，重放攻击指的就是有人将从某个事务中窃取的认证证书用于另一个事务。尽管对 GET 请求来说这也是个问题，但为 POST 和 PUT 请求提供一种简单的方式来避免重放攻击才是非常必要的。在传输表单数据的同时，成功重放原先用过的证书会引发严重的安全问题。

因此，为了使服务器能够接受“重放的”证书，还必须重复发送随机数。缓解这个问题的方法之一就是让服务器产生的随机数包含（如前所述）根据客户端 IP 地址、时间戳、资源 Etag 和私有服务器密钥算出的摘要。这样，IP 地址和一个短小超时值的组合就会给攻击者造成很大的障碍。

但这种解决方案有一个很重要的缺陷。我们之前讨论过，用客户端 IP 地址来创建随机数会破坏经过代理集群的传输。在这类传输中。来自单个用户的多条请求可能会穿过不同的代理。而且，IP 欺骗也并不难实现。

一种可以完全避免重放攻击的方法就是为每个事务都使用一个唯一的随机数。在这种实现方式中，服务器会为每个事务发布唯一的随机数和一个超时值。发布的随机数只对指定的事务有效，而且只在超时值的持续区间内有效。这种方式会增加服务器的负担，但这种负担可忽略不计。

13.5.3 多重认证机制

服务器支持多重认证机制（比如基本认证和摘要认证）时，通常会在 `WWW-Authenticate`
303 首部提供选项。由于没有要求客户端选择功能最强的认证机制，所以得到的认证效果就和功能最弱的认证方案差不多。

要避免出现这个问题，最直接的方法就是让客户端总是去选择可用认证方案中功能

最强的那个。如果无法实现（因为大部分人使用的都是商业化客户端），唯一的选择就是使用一个只维护最强认证方案的代理服务器。但只有在已知所有客户端都支持所选认证方案的区域中才能采用这种方式——比如，在公司网络中。

13.5.4 词典攻击

词典攻击是典型的密码猜测型攻击方式。恶意用户对某个事务进行窃听，并对随机数 / 响应对使用标准的密码猜测程序。如果用户使用的是相对比较简单的密码，而且服务器使用的也是简单的随机数，它很可能会找到匹配项。如果没有密码过期策略，只要有足够的时间和破解密码所需的一次性费用，就很容易搜集到足够多的密码，造成实质性的破坏。

除了使用复杂的相对难以破译的密码和合适的密码过期策略之外，确实没有什么好的方法可以解决这个问题。

13.5.5 恶意代理攻击和中间人攻击

现在很多因特网流量都会在这个或那个地方流经某个代理。随着重定向技术和拦截代理的出现，用户甚至都意识不到他的请求穿过了某个代理。如果这些代理中有一个是恶意的或者容易被入侵的，就会使客户端置于中间人攻击之下。

这种攻击可以采用窃听的形式，也可以删除提供的所有选项，用最薄弱的认证策略（比如基本认证）来取代现有的认证机制，对其进行修改。

入侵受信代理的方式之一就是使用其扩展接口。有时代理会提供复杂的编程接口，可以为这类代理编写一个扩展（比如，plug-in）来拦截流量并对其进行修改。不过，数据中心和代理自身提供的安全性使得通过恶意 plug-in 进行中间人攻击的可能性变得很渺茫。

没有什么好办法可以解决这个问题。可行的解决方案包括由客户端提供与认证功能有关的可见线索，对客户端进行配置使其总是使用可用认证策略中功能最强的那一种，等等。但即使使用的是最强大的认证策略，客户端仍然很容易被窃听。防止这些攻击唯一简便的方式就是使用 SSL。

304

13.5.6 选择明文攻击

使用摘要认证的客户端会用服务器提供的随机数来生成响应。但如果中间有一个被入侵的或恶意的代理在拦截流量（或者有个恶意的原始服务器），就可以很容易地为客户端的响应计算提供随机数。使用已知密钥来计算响应可以简化响应的密码分析过程。这种方式被称为*选择明文攻击*（chosen plaintext attack）。选择明文攻击有以下几种变体形式。

- 预先计算的词典攻击

  这是词典攻击和选择明文攻击的组合。首先，发起攻击的服务器会用预先确定的随机数和常见密码的变化形式产生一组响应，创建一个词典。一旦有了规模可观的词典，攻击服务器或代理就可以完成对流量的封锁，向客户端发送预先确定的随机数。攻击者从客户端得到一个响应时，会搜索生成的词典，寻找匹配项。如果有匹配项，攻击者就捕获了这个用户的密码。

- 批量暴力型攻击

  批量暴力型攻击的不同之处在于计算密码的方式。它没有试图去匹配预先计算出来的摘要，而是用一组机器枚举了指定空间内所有可能的密码。随着机器运行速度变得越来越快，暴力型攻击的可行性也变得越来越强了。

总之，这些攻击所造成的威胁是很容易应对的。防止这些攻击的一种方法就是配置客户端使用可选的 `cnonce` 指令，这样响应就是基于客户端的判断产生的，而不是用服务器提供的随机数（这个随机数可能会被攻击者入侵）产生的。通过这种方法，再结合一些强制使用合理强密码的策略，以及一个好的密码过期策略，就可以完全消除选择明文攻击的威胁。

13.5.7 存储密码

摘要认证机制将对比用户的响应与服务器内部存储的内容——通常就是用户名和 H(A1) 元组对，其中 H(A1) 是从用户名、域和密码的摘要中导出的。

与 Unix 机器中传统的密码文件不同，如果摘要认证密码文件被入侵了，攻击者马上就能够使用域中所有文件，不需要再进行解码了。

消除这个问题的方法包括：

- 就像密码文件中包含的是明文密码一样来保护它；
- 确保域名在所有域中是唯一的。这样，如果密码文件被入侵，所造成的破坏也只局限于一个特定的域中。包含主机和 `domain` 的全路径域名就可以满足这个要求。

305

尽管摘要认证提供的解决方案比基本认证要强壮且安全得多，但它并没有为内容的安全提供任何保证——真正安全的事务只有通过 SSL 才能实现，我们将在下一章介绍。

13.6 更多信息

更多有关认证的信息，参见以下资源。

- http://www.ietf.org/rfc/rfc2617.txt

RFC 2617，“HTTP Authentication：Basic and Digest Access Authentication.”（“HTTP 认证：基本和摘要访问认证”）。

306

第14章

安全HTTP

前面三章讨论了一些有助于识别和认证用户的 HTTP 特性。在友好环境中，这些技术都能够很好地工作，但在充满各种利益驱动和恶意对手的环境中，它们并不足以保护那些重要的事务处理。

本章提供了一种更复杂，更安全的技术，通过数字密码来保护 HTTP 事务免受窃听和篡改的侵害。

14.1 保护HTTP的安全

人们会用 Web 事务来处理一些很重要的事情。如果没有强有力的安全保证，人们就无法安心地进行网络购物或使用银行业务。如果无法严格限制访问权限，公司就不能将重要的文档放在 Web 服务器上。Web 需要一种安全的 HTTP 形式。

前面的章节讨论了一些提供认证（基本认证和摘要认证）和报文完整性检查（摘要 `qop="auth-int"`）的轻量级方法。对很多网络事务来说，这些方法都是很好用的，但对大规模的购物、银行事务，或者对访问机密数据来说，并不足够强大。这些更为重要的事务需要将 HTTP 和数字加密技术结合起来使用，才能确保安全。

HTTP 的安全版本要高效、可移植且易于管理，不但能够适应不断变化的情况而且还应该能满足社会和政府的各项要求。我们需要一种能够提供下列功能的 HTTP 安全技术。

- 服务器认证（客户端知道它们是在与真正的而不是伪造的服务器通话）。
- 客户端认证（服务器知道它们是在与真正的而不是伪造的客户端通话）。
- 完整性（客户端和服务器的数据不会被修改）。
- 加密（客户端和服务器的对话是私密的，无需担心被窃听）。
- 效率（一个运行的足够快的算法，以便低端的客户端和服务器使用）。
- 普适性（基本上所有的客户端和服务器都支持这些协议）。
- 管理的可扩展性（在任何地方的任何人都可以立即进行安全通信）。
- 适应性（能够支持当前最知名的安全方法）。
- 在社会上的可行性（满足社会的政治文化需要）。

307

HTTPS

HTTPS 是最流行的 HTTP 安全形式。它是由网景公司首创的，所有主要的浏览器和服务器都支持此协议。

HTTPS 方案的 URL 以 https://，而不是 http:// 开头，据此就可以分辨某个 Web 页面是通过 HTTPS 而不是 HTTP 访问的（有些浏览器还会显示一些标志性的安全提示，如图 14-1 所示）。

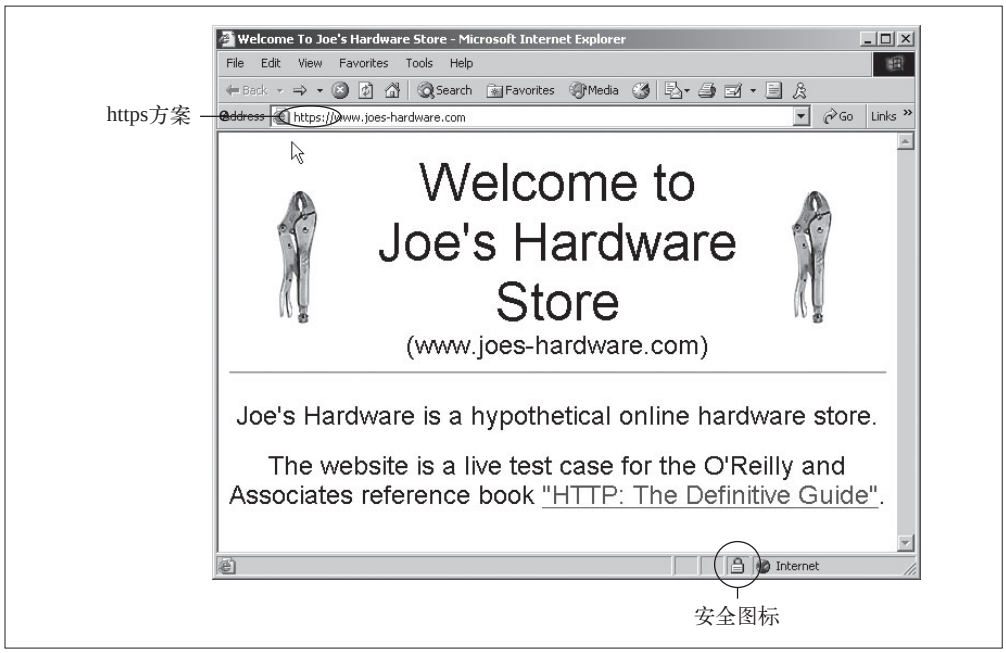


图 14-1　浏览安全 Web 站点

使用 HTTPS 时，所有的 HTTP 请求和响应数据在发送到网络之前，都要进行加密。HTTPS 在 HTTP 下面提供了一个传输级的密码安全层（参见图 14-2）——可以使用 SSL，也可以使用其后继者——传输层安全（Transport Layer Security，TLS）。由于 SSL 和 TLS 非常类似，所以在本书中我们不太严格地用术语 SSL 来表示 SSL 和 TLS。

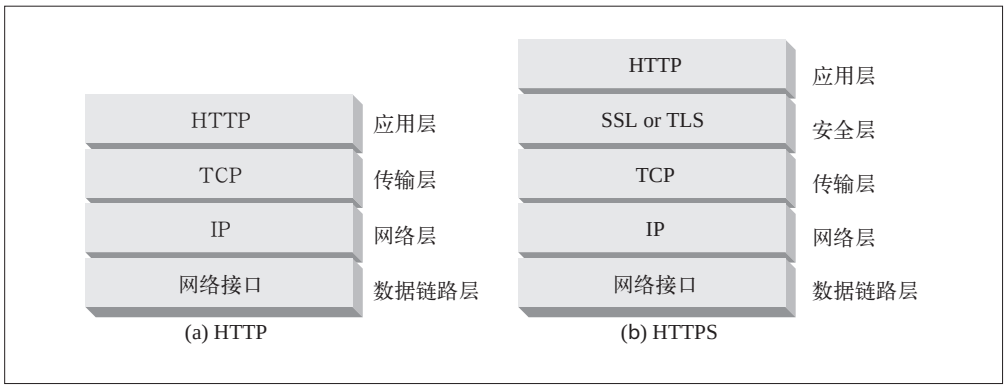


图 14-2　HTTPS 是位于安全层之上的 HTTP，这个安全层位于 TCP 之上

大部分困难的编码及解码工作都是在 SSL 库中完成的，所以 Web 客户端和服务器在使用安全 HTTP 时无需过多地修改其协议处理逻辑。在大多数情况下，只需要用 308

SSL 的输入 / 输出调用取代 TCP 的调用，再增加其他几个调用来配置和管理安全信息就行了。

14.2 数字加密

在详细探讨 HTTPS 之前，我们先介绍一些 SSL 和 HTTPS 用到的加密编码技术的背景知识。在接下来的几节里，我们会对数字加密的本质进行一个快速的入门性介绍。如果你已经掌握了数字加密的技术和术语，可以直接阅读 14.7 节。

在这个数字加密技术的入门介绍中，我们会讨论以下内容。

- *密码*

对文本进行编码，使偷窥者无法识别的算法。

- *密钥*

改变密码行为的数字化参数。

- *对称密钥加密系统*

编 / 解码使用相同密钥的算法。

- *不对称密钥加密系统*

编 / 解码使用不同密钥的算法。

- *公开密钥加密系统*

一种能够使数百万计算机便捷地发送机密报文的系统。

- *数字签名*

用来验证报文未被伪造或篡改的校验和。

- *数字证书*

309 由一个可信的组织验证和签发的识别信息。

14.2.1 密码编制的机制与技巧

密码学是对报文进行编 / 解码的机制与技巧。人们用加密的方式来发送秘密信息已经有数千年了。但密码学所能做的还不仅仅是加密报文以防止好事者的读取，我们还可以用它来防止对报文的篡改，甚至还可以用密码学来证明某条报文或某个事务确实出自你手，就像支票上的手写签名或信封上的压纹封蜡一样。

14.2.2 密码

密码学基于一种名为*密码*（cipher）的秘密代码。密码是一套编码方案——一种特殊的报文编码方式和一种稍后使用的相应解码方式的结合体。加密之前的原始报文通常被称为*明文*（plaintext 或 cleartext）。使用了密码之后的编码报文通常被称作*密文*（ciphertext）。图 14-3 显示了一个简单的例子。

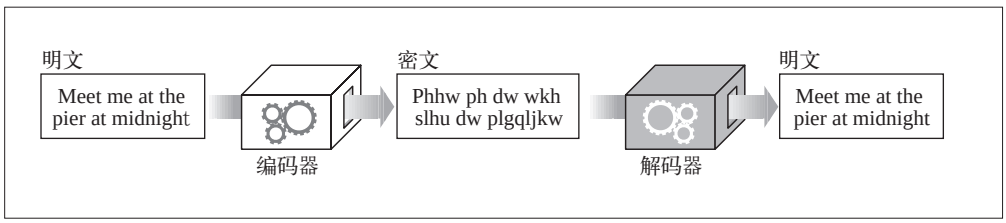


图 14-3　明文和密文

用密码来生成保密信息已经有数千年了。传说尤利乌斯 • 凯撒（Julius Caesar）曾使用过一种三字符循环移位密码，报文中的每个字符都由字母表中三个位置之后的字符来取代。在现代的字母表中，“A”就应该由“D”来取代，“B”就应该由“E”来取代，以此类推。

比如，在图 14-4 中，用 rot3（循环移位 3 字符）密码就可以将报文“meet me at the pier at midnight”编码为密文“phhw ph dw wkh slhu dw plgqljkw”。[1] 通过解码，在字母表中循环移位 3 个字符，就可以将密文解密回原来的明文报文。



图 14-4　循环移位 3 字符密码实例

310

注 1：为了简化这个例子，我们没有对标点和空格进行循环移位，但你可以自己试一试。

14.2.3 密码机

最初，人们需要自己进行编码和解码，所以起初密码是相当简单的算法。因为密码很简单，所以人们通过纸笔和密码书就可以进行编解码了，但聪明人也可以相当容易地“破解”这些密码。

随着技术的进步，人们开始制造一些机器，这些机器可以用复杂得多的密码来快速、精确地对报文进行编解码。这些密码机不仅能做一些简单的旋转，它们还可以替换字符、改变字符顺序，将报文切片切块，使代码的破解更加困难。[2]

14.2.4 使用了密钥的密码

编码算法和编码机都可能会落入敌人的手中，所以大部分机器上都有一些号盘，可以将其设置为大量不同的值以改变密码的工作方式。即使机器被盗，没有正确的号盘设置（密钥值），解码器也无法工作。[3]

这些密码参数被称为*密钥*（key）。要在密码机中输入正确的密钥，解密过程才能正确进行。密码密钥会让一个密码机看起来好像是多个虚拟密码机一样，每个密码机都有不同的密钥值，因此其行为都会有所不同。

图 14-5 显示了使用密钥的密码实例。加密算法就是普通的“循环移位 *N* 字符”密码。*N* 的值由密钥控制。将同一条输入报文“meet me at the pier at midnight”通过同一台编码机进行传输，会随密钥值的不同产生不同的输出。现在，基本上所有的加密算法都会使用密钥。

14.2.5 数字密码

随着数字计算的出现，出现了以下两个主要的进展。

311

- 从机械设备的速度和功能限制中解放出来，使复杂的编 / 解码算法成为可能。
- 支持超大密钥成为可能，这样就可以从一个加密算法中产生出数万亿的虚拟加密算法，由不同的密钥值来区分不同的算法。密钥越长，编码组合就越多，通过随机猜测密钥来破解代码就越困难。

注 2：最著名的机械编码机可能就是第二次世界大战期间德国的 Enigma 编码机了。尽管 Enigma 密码非常复杂，但阿兰 · 图灵（Alan Turing）和他的同事们在 20 世纪 40 年代初期就可以用最早的数字计算机破解 Enigma 代码了。

注 3：在现实中，机器逻辑可能会指向一些可利用的模式，所以拥有机器逻辑有时会有助于密码的破解。现代的加密算法通常都设计为，即使大家都知道这些算法，恶意的攻击者也很难发现任何有助于破解代码的模式。实际上，很多功能最强大的密码都会将其源代码放在公共域中，供大家浏览和学习！

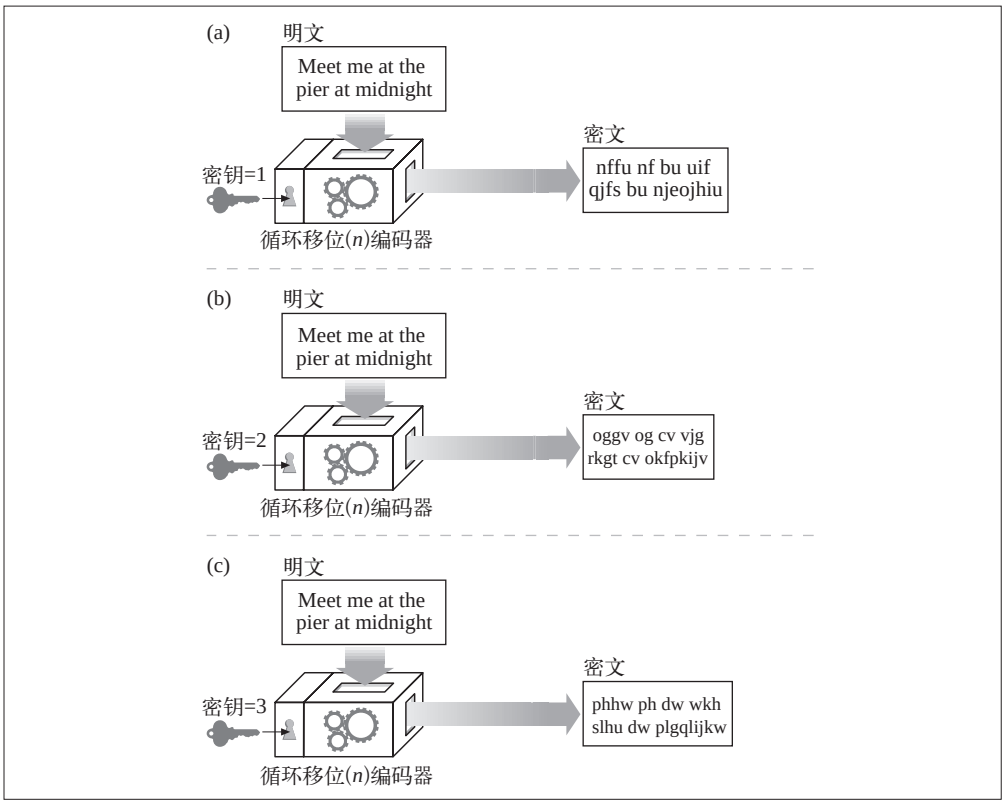


图 14-5　使用不同密钥的旋转 *N* 字符密码

与金属钥匙或机械设备中的号盘设置相比，数字密钥只是一些数字。这些数字密钥值是编 / 解码算法的输入。编码算法就是一些函数，这些函数会读取一块数据，并根据算法和密钥值对其进行编 / 解码。

给定一段明文报文 P、一个编码函数 E 和一个数字编码密钥 e，就可以生成一段经过编码的密文 C（参见图 14-6）。通过解码函数 D 和解码密钥 d，可以将密文 C 解码为原始的明文 P。当然，编 / 解码函数都是互为反函数的，对 P 的编码进行解码就会回到原始报文 P 上去。

312

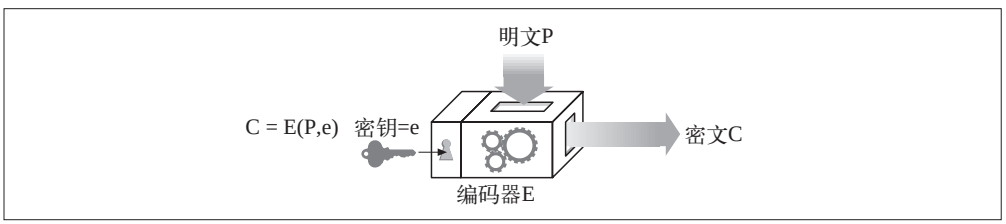


图 14-6　用编码密钥 e 对明文进行编码，用解码密钥 d 进行解码

14.3 对称密钥加密技术

我们来更详细地看看密钥和密码是怎样配合工作的。很多数字加密算法都被称为对称密钥（symmetric-key）加密技术，这是因为它们在编码时使用的密钥值和解码时一样（e=d）。我们就将其统称为密钥 k。

在对称密钥加密技术中，发送端和接收端要共享相同的密钥 k 才能进行通信。发送端用共享的密钥来加密报文，并将得到的密文发送给接收端。接收端收到密文，并对其应用解密函数和相同的共享密钥，恢复出原始的明文（参见图 14-7）。

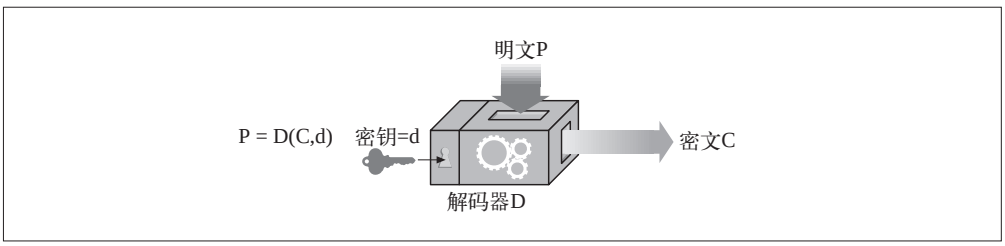


图 14-7 对称密钥加密算法为编 / 解码使用相同的密钥

流行的对称密钥加密算法包括：DES、Triple-DES、RC2 和 RC4。

14.3.1 密钥长度与枚举攻击

保持密钥的机密状态是很重要的。在很多情况下，编 / 解码算法都是众所周知的，因此密钥就是唯一保密的东西了。

好的加密算法会迫使攻击者试遍每一个可能的密钥，才能破解代码。用暴力去尝试所有的密钥值称为*枚举攻击*（enumeration attack）。如果只有几种可能的密钥值，居心不良的人通过暴力遍历所有值，就能最终破解代码了。但如果有大量可能的密钥值，他可能就要花费数天、数年，甚至无限长的时间来遍历所有的密钥，去查找能够破解密码的那一个。

313

可用密钥值的数量取决于密钥中的位数，以及可能的密钥中有多少是有效的。就对称密钥加密技术来说，通常所有的密钥值都是有效的。[4] 8 位的密钥只有 256 个可能的密钥值，40 位的密钥可以有 2^{40} 个可能的密钥值（大约是一万亿个密钥），128 位的密钥可以产生大约 340 000 000 000 000 000 000 000 000 000 000 000 000 个可能的密钥值。

注 4：有些加密技术中只有部分密钥值是有效的。比如，在最知名的非对称密钥加密系统 RSA 中，有效密钥必须以某种方式与质数相关。可能的密钥值中只有少量密钥具备此特性。

在传统的对称密钥加密技术中，对小型的、不太重要的事务来说，40 位的密钥就足够安全了。但现在的高速工作站就可以将其破解，这些工作站每秒可以进行数十亿次计算。

相比之下，对于对称密钥加密技术，128 位的密钥被认为是非常强大的。实际上，长密钥对密码安全有着非常重要的影响，美国政府甚至对使用长密钥的加密软件实施了出口控制，以防止潜在的敌对组织创建出美国国家安全局（National Security Agency，NSA）自己都无法破解的秘密代码。

Bruce Schneier 编写的 *Applied Cryptography*（John Wiley & Sons 出版社）是一本很棒的书，书中有一张表，表中对使用 1995 年的技术和耗费，通过猜测所有的密钥来破解一个 DES 密码所需的时间进行了描述。[5] 表 14-1 摘录了这张表。

表14-1　较长的密钥要花费更多的精力去破解（来自*Applied Cryptography*一书，1995年的数据）

攻击耗费	40位密钥	56位密钥	64位密钥	80位密钥	128位密钥
100 000 美元	2 秒	35 小时	1 年	70 000 年	10^{19} 年
1 000 000 美元	200 毫秒	3.5 小时	37 天	7 000 年	10^{18} 年
10 000 000 美元	20 毫秒	21 分钟	4 天	700 年	10^{17} 年
100 000 000 美元	2 毫秒	2 分钟	9 小时	70 年	10^{16} 年
1 000 000 000 美元	200 微秒	13 秒	1 小时	7 年	10^{15} 年

根据 1995 年微处理器的速度，愿意花费 100 000 美元的攻击者可以在大约 2 秒内破解一个 40 位的 DES 代码。2002 年的计算机就已经比 1995 年的快 20 倍了。除非用户经常修改密钥，否则对于别有用心的攻击者来说，40 位的密钥是不安全的。

DES 的 56 位标准密钥长度就更安全一些。从 1995 年的经济水平来说，花费 100 万美元进行的攻击还是要几个小时才能破解密码。但可使用超级计算机的用户则只需 314
数秒钟即可通过暴力方法破解密码。与之相对的是，通常大家都认为长度与 Triple-DES 密钥相当的 128 位 DES 密钥实际上是任何人以任何代价都无法通过暴力攻击破解的。[6]

注 5：1995 年之后，计算速度得到了飞速的提高，费用也降低了。你越晚读到这本书，计算的速度就会越快！但即使所需的时间会成 5 倍、10 倍或更多倍的减少，这张表仍然是有参考价值的。

注 6：但是，长的密钥并不意味着可以高枕无忧了！加密算法或实现中可能会有不为人注意的缺陷，为攻击者提供了可攻击的弱点。攻击者也可能会有一些与密钥产生方式有关的信息，这样他就会知道使用某些密钥的可能性比另一些要大，从而有助于进行有目的的暴力攻击。或者用户可能将保密的密钥落在了什么地方，被攻击者偷走了。

14.3.2 建立共享密钥

对称密钥加密技术的缺点之一就是发送者和接收者在互相对话之前，一定要有一个共享的保密密钥。

如果想要与 Joe 的五金商店进行保密的对话，可能是在看了公共电视台的家装节目之后，想要订购一些木工工具，那么在安全地订购任何东西之前，要先在你和 www.joes-hardware.com 之间建立一个私有的保密密钥。你需要一种产生保密密钥并将其记住的方式。你和 Joe 的五金商店，以及因特网上所有其他人，都要产生并记住数千个密钥。

比如 Alice（A）、Bob（B）和 Chris（C）都想与 Joe 的五金商店（J）对话。A、B 和 C 都要建立自己与 J 之间的保密密钥。A 可能需要密钥 K^{AJ}，B 可能需要密钥 K^{BJ}，C 可能需要密钥 K^{CJ}。每对通信实体都需要自己的私有密钥。如果有 N 个节点，每个节点都要和其他所有 N-1 个节点进行安全对话，总共大概会有 N^2 个保密密钥：这将是一个管理噩梦。

14.4 公开密钥加密技术

公开密钥加密技术没有为每对主机使用单独的加密 / 解密密钥，而是使用了两个非对称密钥：一个用来对主机报文编码，另一个用来对主机报文解码。编码密钥是众所周知的（这也是公开密钥加密这个名字的由来），但只有主机才知道私有的解密密钥（参见图 14-8）。这样，每个人都能找到某个特定主机的公开密钥，密钥的建立变得更加简单。但解码密钥是保密的，因此只有接收端才能对发送给它的报文进行解码。

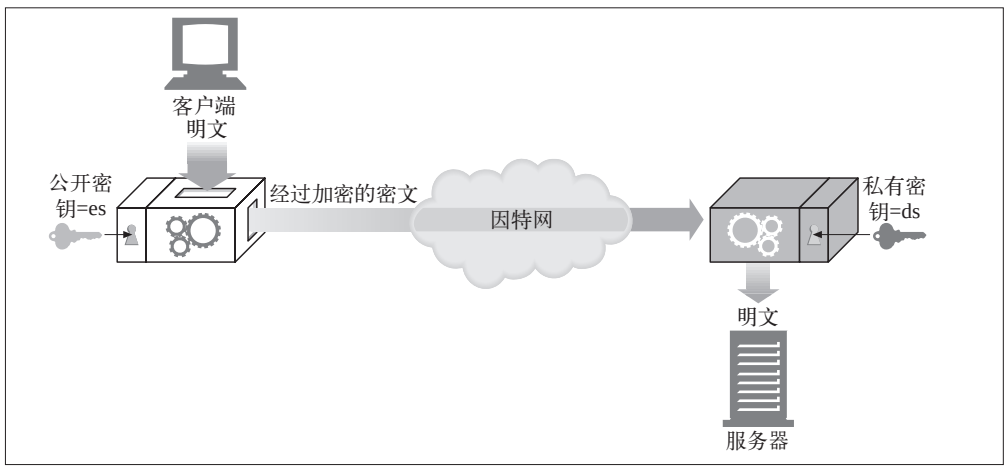


图 14-8 公开密钥加密技术是非对称的，为编码和解码使用了不同的密钥

节点 X 可以将其加密密钥 ex 公之于众。[7] 现在，任何想向节点 X 发送报文的人都可以使用相同的公开密钥了。因为每台主机都分配了一个所有人均可使用的编码密钥，所以公开密钥加密技术避免了对称密钥加密技术中成对密钥数目的 N^2 扩展问题（参见图 14-9）。 315

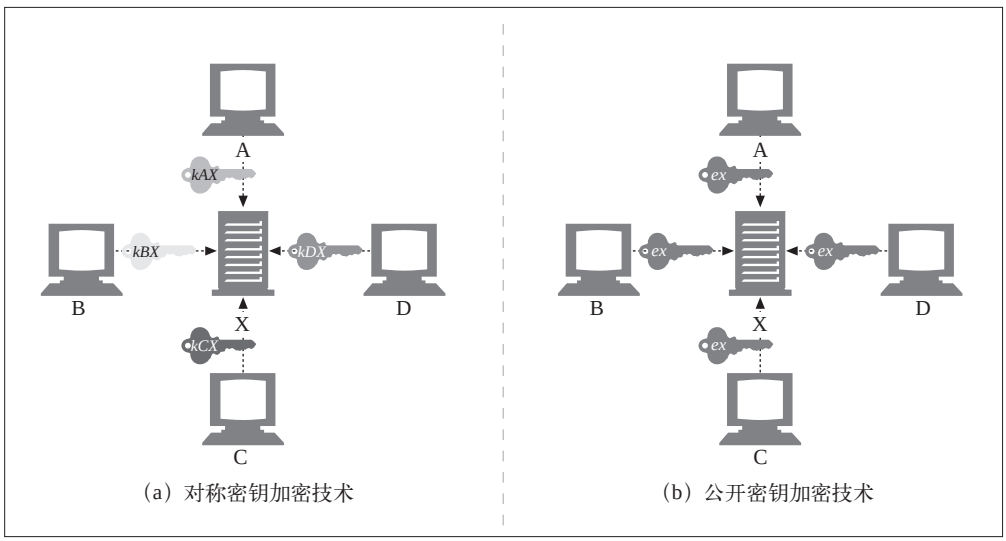


图 14-9　公开密钥加密技术为每台主机分配了一个公开编码密钥

尽管每个人都可以用同一个密钥对发给 X 的报文进行编码，但除了 X，其他人都无法对报文进行解码，因为只有 X 才有解码的私有密钥 d^x。将密钥分隔开来可以让所有人都能够对报文进行编码，但只有其所有者才能对报文进行解码。这样，各节点向服务器安全地发送报文就更加容易了，因为它们只要查找到服务器的公开密钥就行了。

通过公开密钥加密技术，全球所有的计算机用户就都可以使用安全协议了。制定标准化的公开密钥技术包是非常重要的，因此，大规模的公开密钥架构（Public-Key Infrastructure，PKI）标准创建工作已经开展十多年了。 316

14.4.1　RSA

所有公开密钥非对称加密系统所面临的共同挑战是，要确保即便有人拥有了下面所有的线索，也无法计算出保密的私有密钥：

注 7：我们稍后会看到，大部分公开密钥查找工作实际上都是通过数字证书来实现的，但如何找到公开密钥现在并不重要——只要知道可以在某个地方公开获取就行了。

- 公开密钥（是公有的，所有人都可以获得）；
- 一小片拦截下来的密文（可通过对网络的嗅探获取）；
- 一条报文及与之相关的密文（对任意一段文本运行加密器就可以得到）。

RSA 算法就是一个满足了所有这些条件的流行的公开密钥加密系统，它是在 MIT 发明的，后来由 RSA 数据安全公司将其商业化。即使有了公共密钥、任意一段明文、用公共密钥对明文编码之后得到的相关密文、RSA 算法自身，甚至 RSA 实现的源代码，破解代码找到相应的私有密钥的难度仍相当于对一个极大的数进行质因数分解的困难程度，这种计算被认为是所有计算机科学中最难的问题之一。因此，如果你发现了一种能够快速地将一个极大的数字分解为质因数的方法，就不仅能够入侵瑞士银行的账户系统，而且还可以获得图灵奖了。

RSA 加密技术的细节中包括很多繁琐的数学问题，我们的介绍不会那么深入。你不需要拥有数论方面的博士学位，有大量的库可以用来执行 RSA 算法。

14.4.2 混合加密系统和会话密钥

任何人只要知道了其公开密钥，就可以向一台公共服务器发送安全报文，所以非对称的公开密钥加密系统是很好用的。两个节点无须为了进行安全的通信而先交换私有密钥。

但公开密钥加密算法的计算可能会很慢。实际上它混合使用了对称和非对称策略。比如，比较常见的做法是在两节点间通过便捷的公开密钥加密技术建立起安全通信，然后再用那条安全的通道产生并发送临时的随机对称密钥，通过更快的对称加密技术对其余的数据进行加密。

14.5 数字签名

到目前为止，我们已经讨论了各种使用对称和非对称密钥加 / 解密保密报文的密钥
317 加密技术。

除了加 / 解密报文之外，还可以用加密系统对报文进行*签名*（sign），以说明是谁编写的报文，同时证明报文未被篡改过。这种技术被称为*数字签名*（digital signing），对下一节将要讨论的因特网安全证书系统来说非常重要。

签名是加了密的校验和

数字签名是附加在报文上的特殊加密校验码。使用数字签名有以下两个好处。

- 签名可以证明是作者编写了这条报文。只有作者才会有最机密的私有密钥，[8] 因此，只有作者才能计算出这些校验和。校验和就像来自作者的个人“签名”一样。
- 签名可以防止报文被篡改。如果有恶意攻击者在报文传输过程中对其进行了修改，校验和就不再匹配了。由于校验和只有作者保密的私有密钥才能产生，所以攻击者无法为篡改了的报文伪造出正确的校验码。

数字签名通常是用非对称公开密钥技术产生的。因为只有所有者才知道其私有密钥，所以可以将作者的私有密钥当作一种“指纹”使用。

图 14-10 显示了一个例子，说明了节点 A 是如何向节点 B 发送一条报文，并对其进行签名的。

- 节点 A 将变长报文提取为定长的摘要。
- 节点 A 对摘要应用了一个“签名”函数，这个函数会将用户的私有密钥作为参数。因为只有用户才知道私有密钥，所以正确的签名函数会说明签名者就是其所有者。在图 14-10 中，由于解码函数 D 中包含了用户的私有密钥，所以我们将其作为签名函数使用。[9]
- 一旦计算出签名，节点 A 就将其附加在报文的末尾，并将报文和签名都发送给 B。

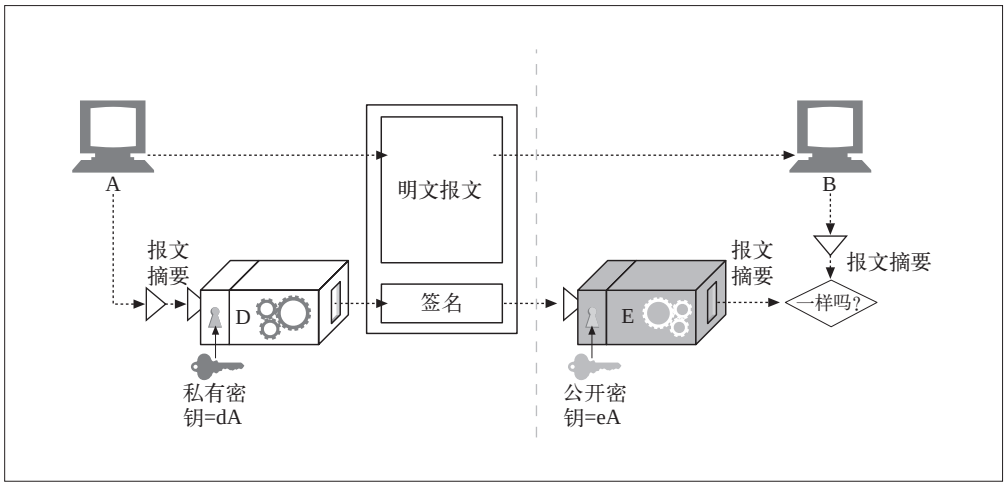


图 14-10　解密的数字签名

注 8：此时假定私有密钥没有被人偷走。大多数私有密钥都会在一段时间后过期。还有一些“取消列表”记录了被偷走或入侵的密钥。

注 9：RSA 加密系统将解码函数 D 作为签名函数使用，是因为 D 已经将私有密钥作为输入使用了。注意，解码函数只是一个函数，因此，可以将其用于任意的输入。同样，在 RSA 加密系统中，以任意顺序应用 D 和 E 函数时，两者都会相互抵消。因此 E(D(stuff)) = stuff，就像 D(E(stuff)) = stuff 一样。