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

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）
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8月第1版
印数：1－5 000册 2012年8月北京第1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。

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。

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 ituring.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地感谢。

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　使用 Telnet 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 连接的正确运行……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。

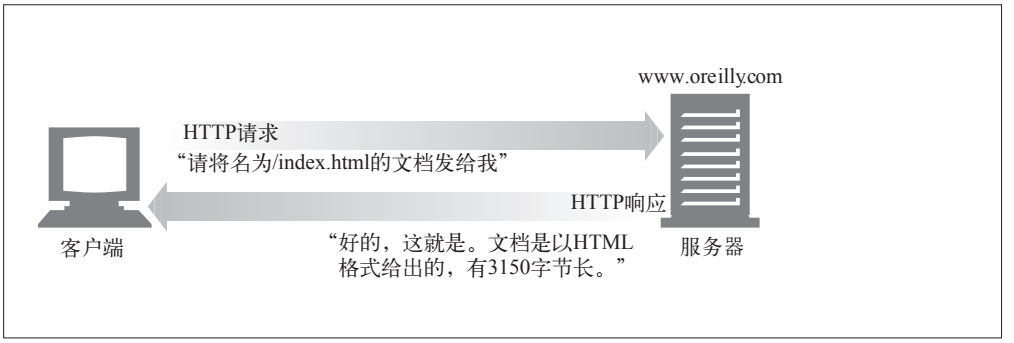


图 1-1　Web 客户端和服务器

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）。

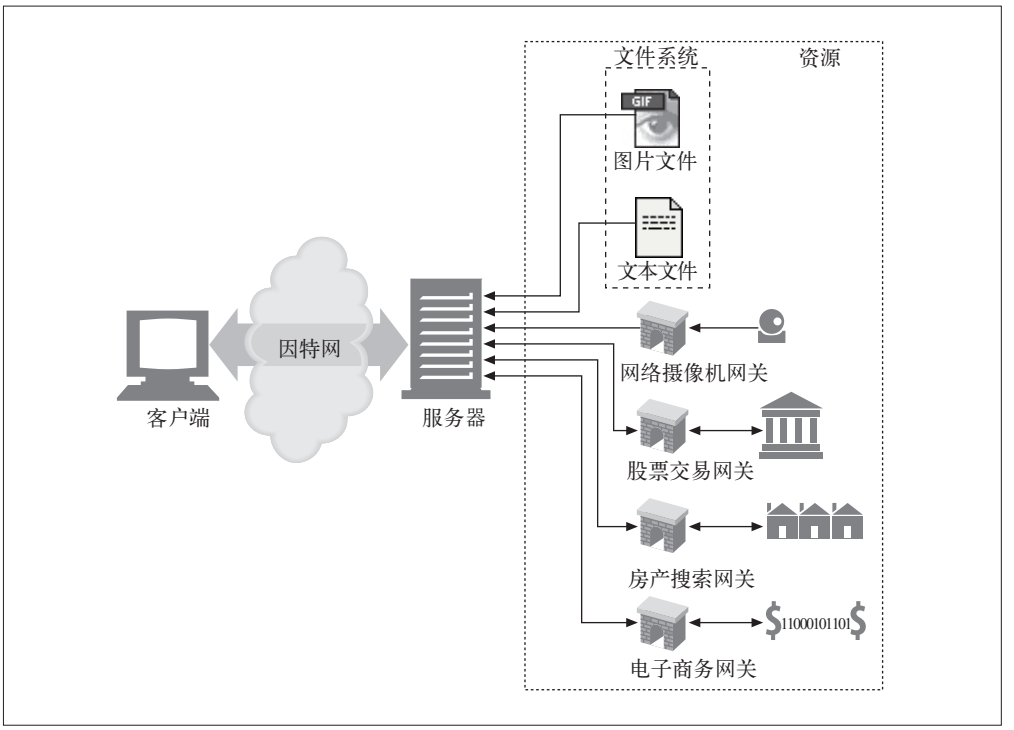


图 1-2　所有能够提供 Web 内容的东西都是 Web 资源

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。

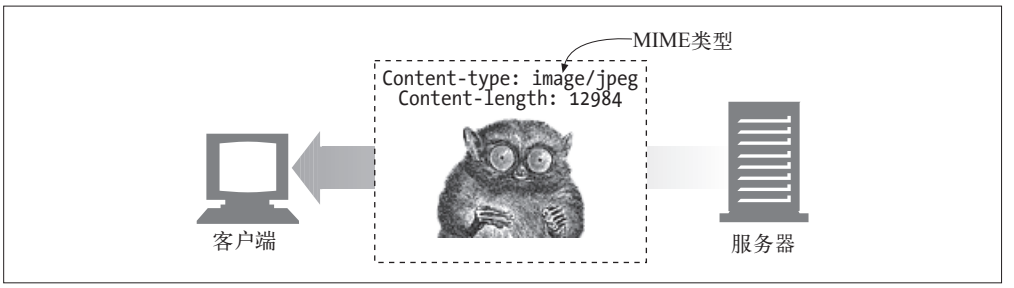


图 1-3　与数据内容一同回送的 MIME 类型

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 版本的图片为 `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。

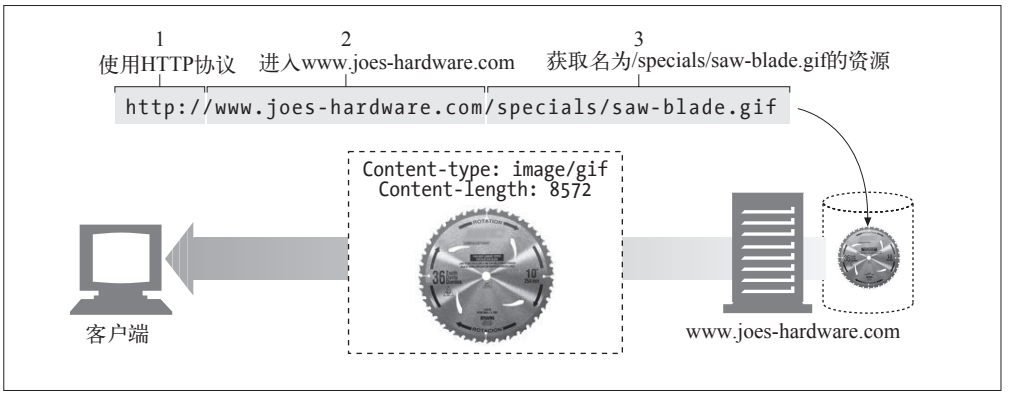


图 1-4　URL 说明了协议、服务器和本地资源

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。

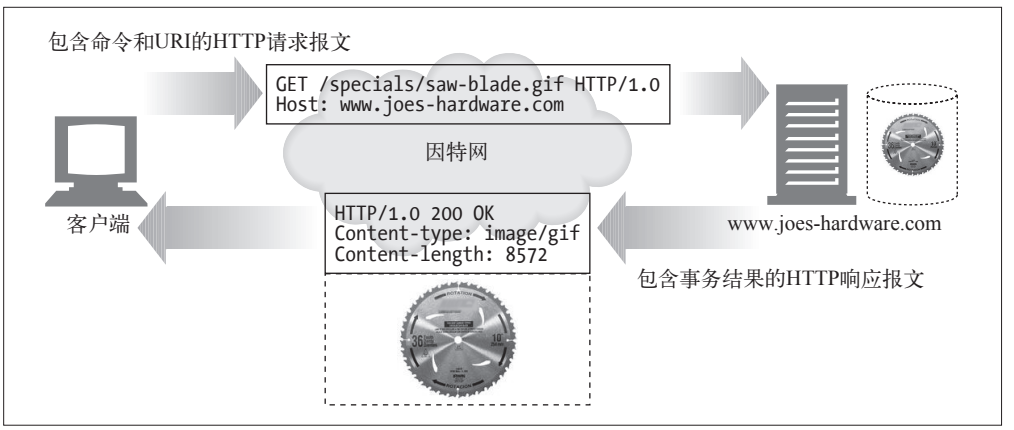


图 1-5 包含请求及响应报文的 HTTP 事务

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。

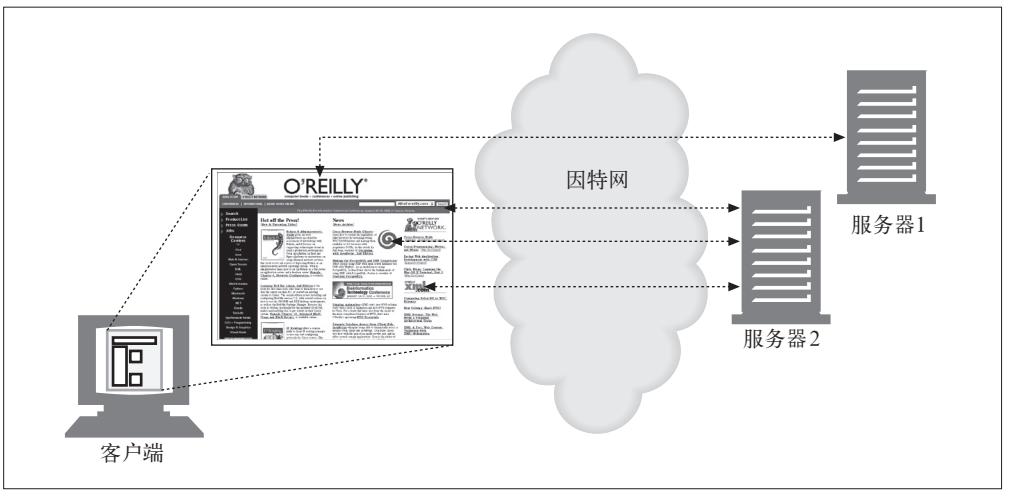


图 1-6　复合 Web 页面要为每个嵌入式资源使用一个单独的 HTTP 事务

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。

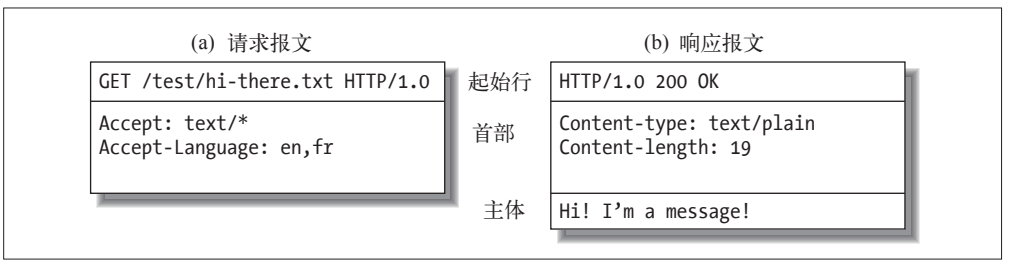


图 1-7　由一行行的简单文本结构组成的 HTTP 报文

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。

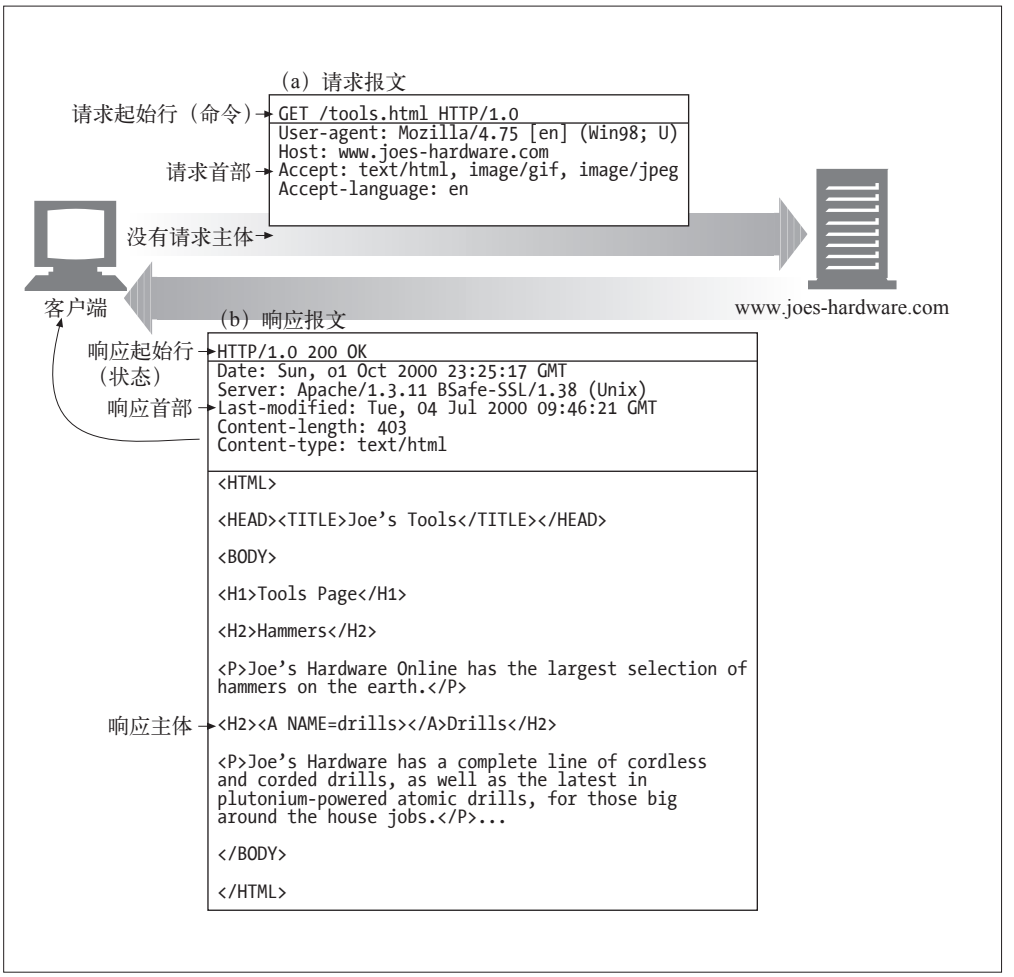


图 1-8　http://www.joes-hardware.com/tools.html 的 GET 事务实例

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）。

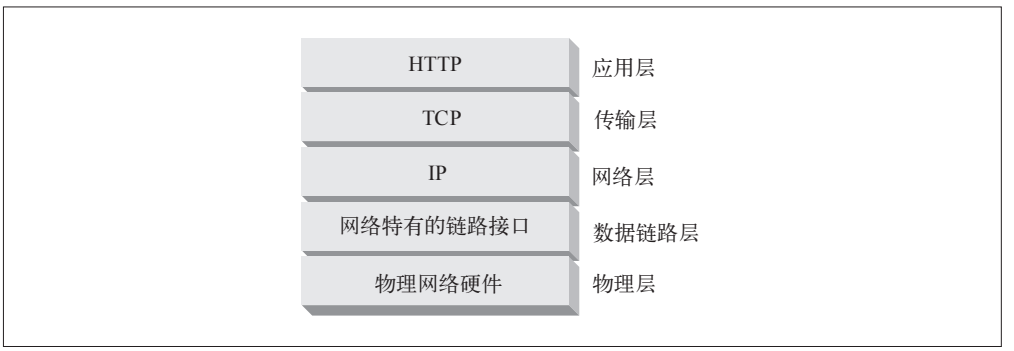


图 1-9　HTTP 网络协议栈

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

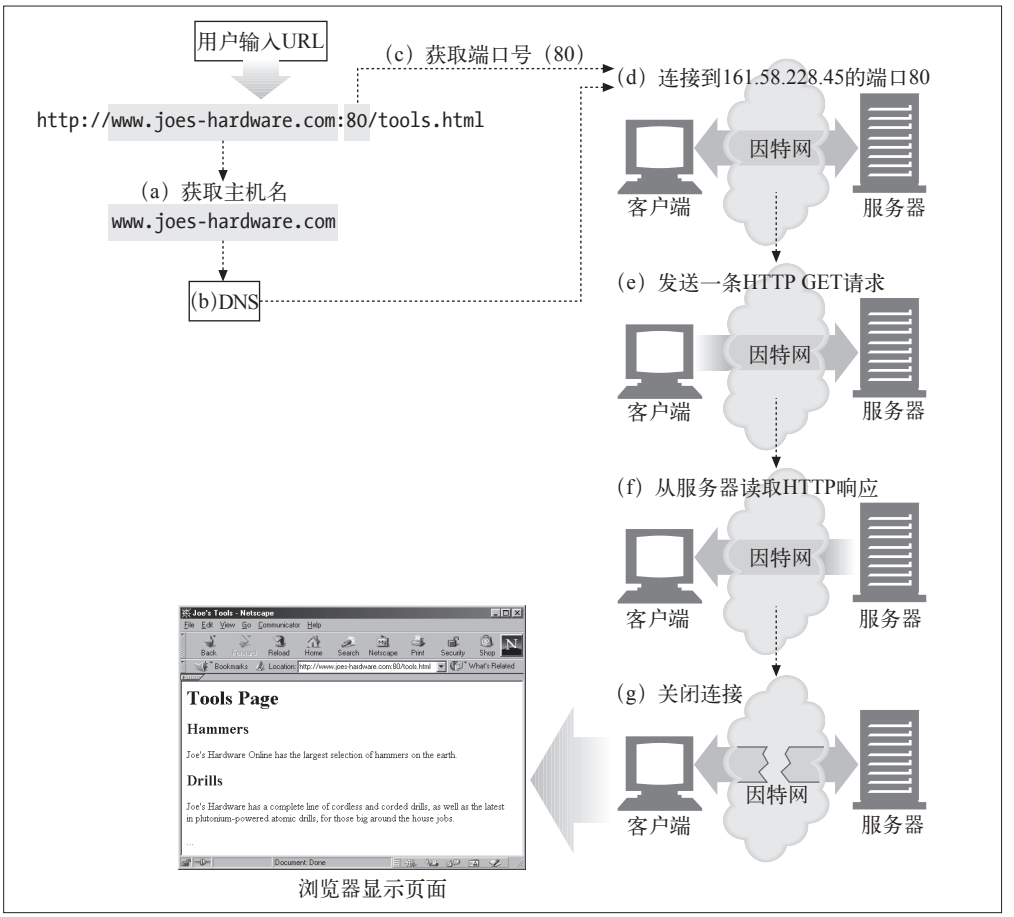


图 1-10　基本的浏览器连接处理 14

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 使用Telnet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。

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。

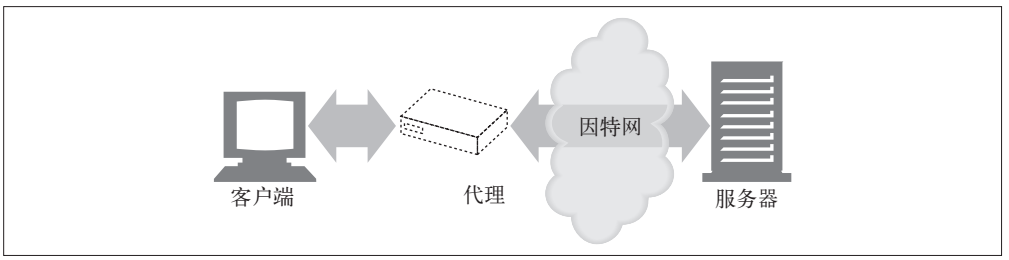


图 1-11 在客户端和服务器之间转发流量的代理

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）。

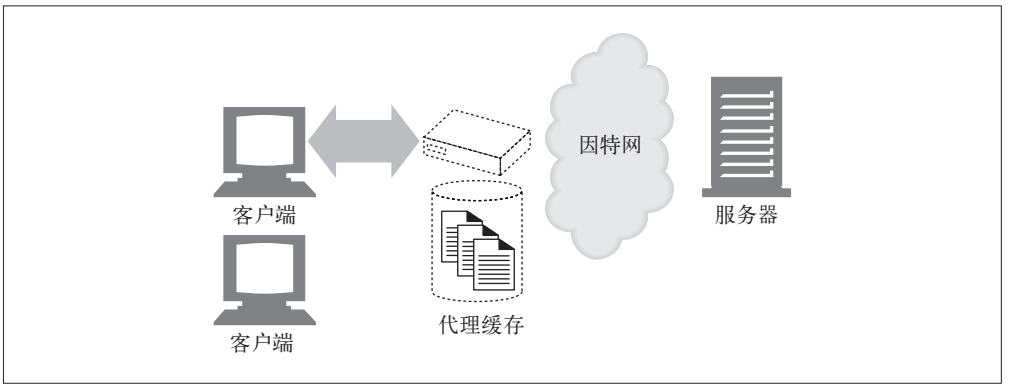


图 1-12 保存常用文档本地副本以提高性能的代理缓存

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。

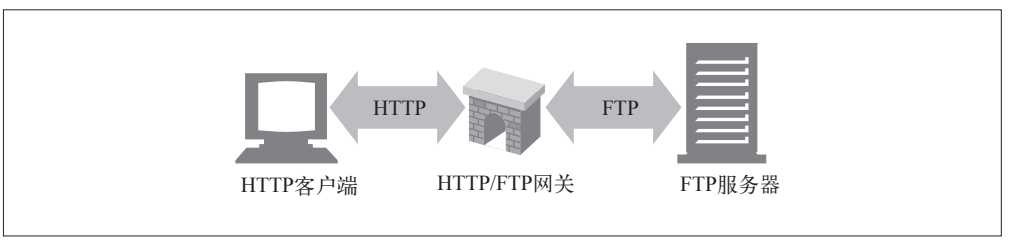


图 1-13 HTTP/FTP 网关

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。

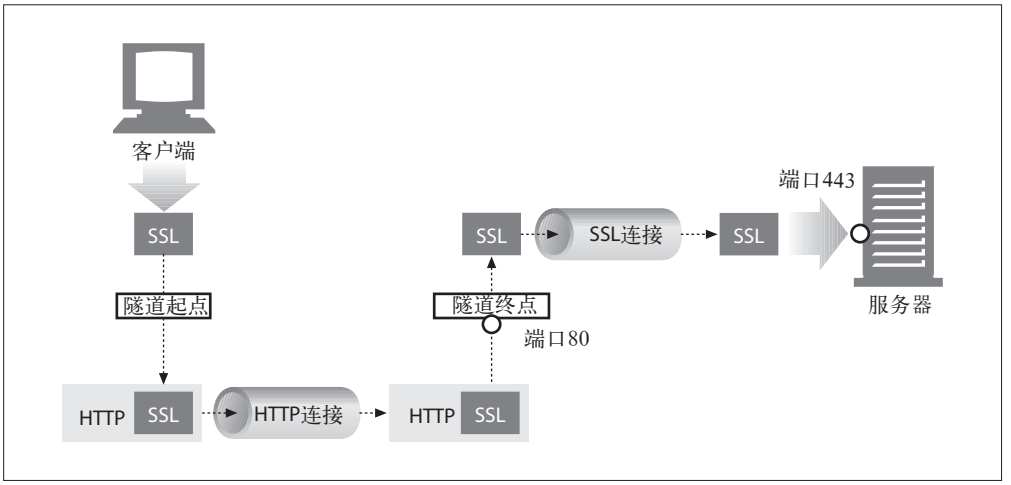


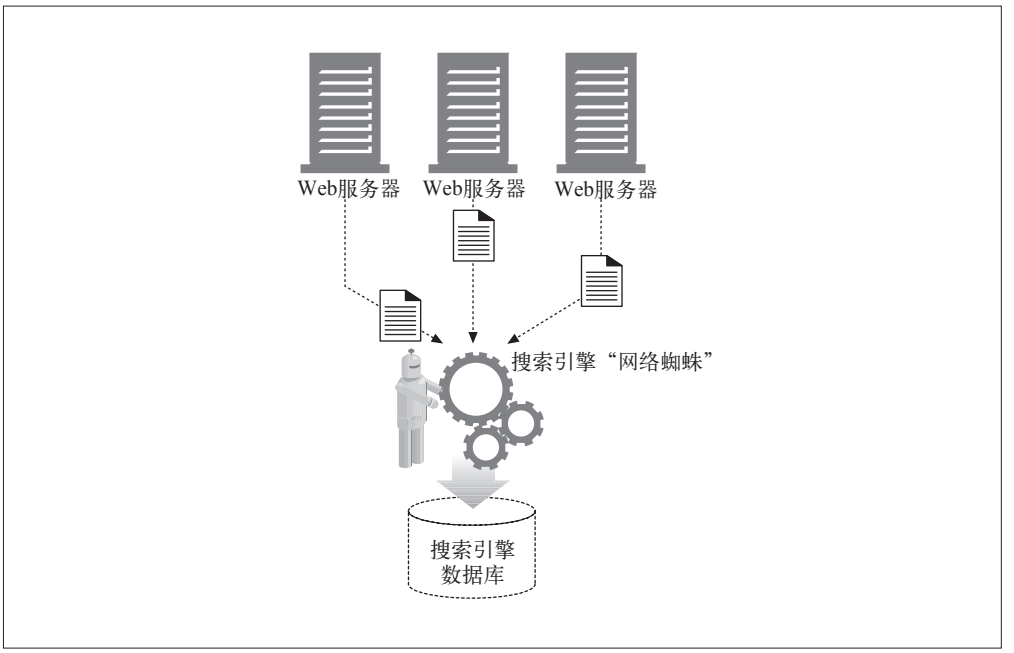
图 1-14　隧道可以在非 HTTP 网络上转发数据（显示的是 HTTP/SSL 隧道）

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。它详细描述了编写此规范时已得到官方认可，且具有“最佳实践”行为的 Web 应用程序。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 连接和一些使用连接的规则。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）。

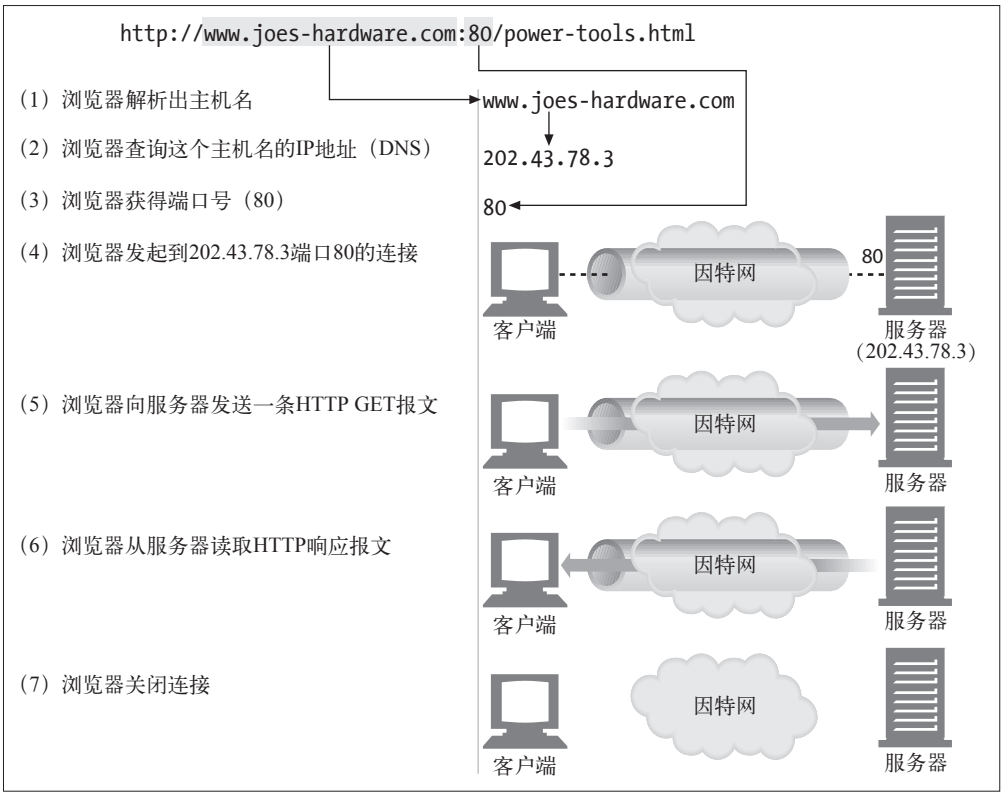


图 4-1　Web 浏览器通过 TCP 连接与 Web 服务器进行交互

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）。

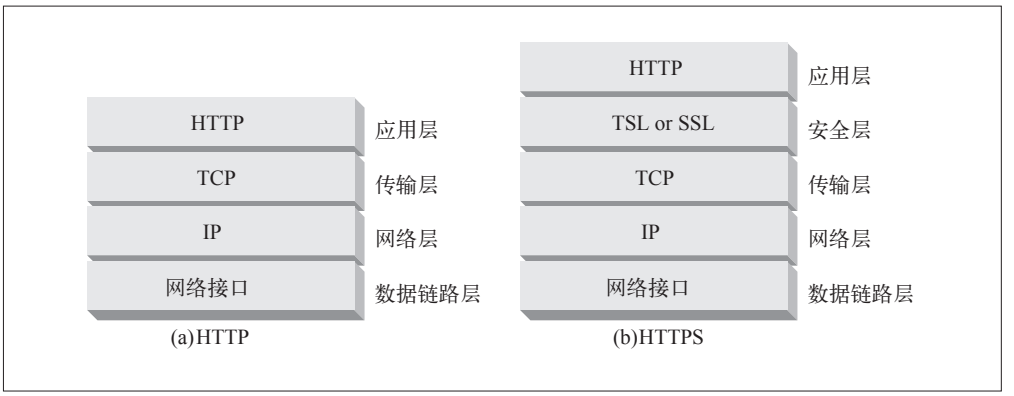


图 4-3　HTTP 和 HTTPS 网络协议栈

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连接的正确运行

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的正确运行的。

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）。

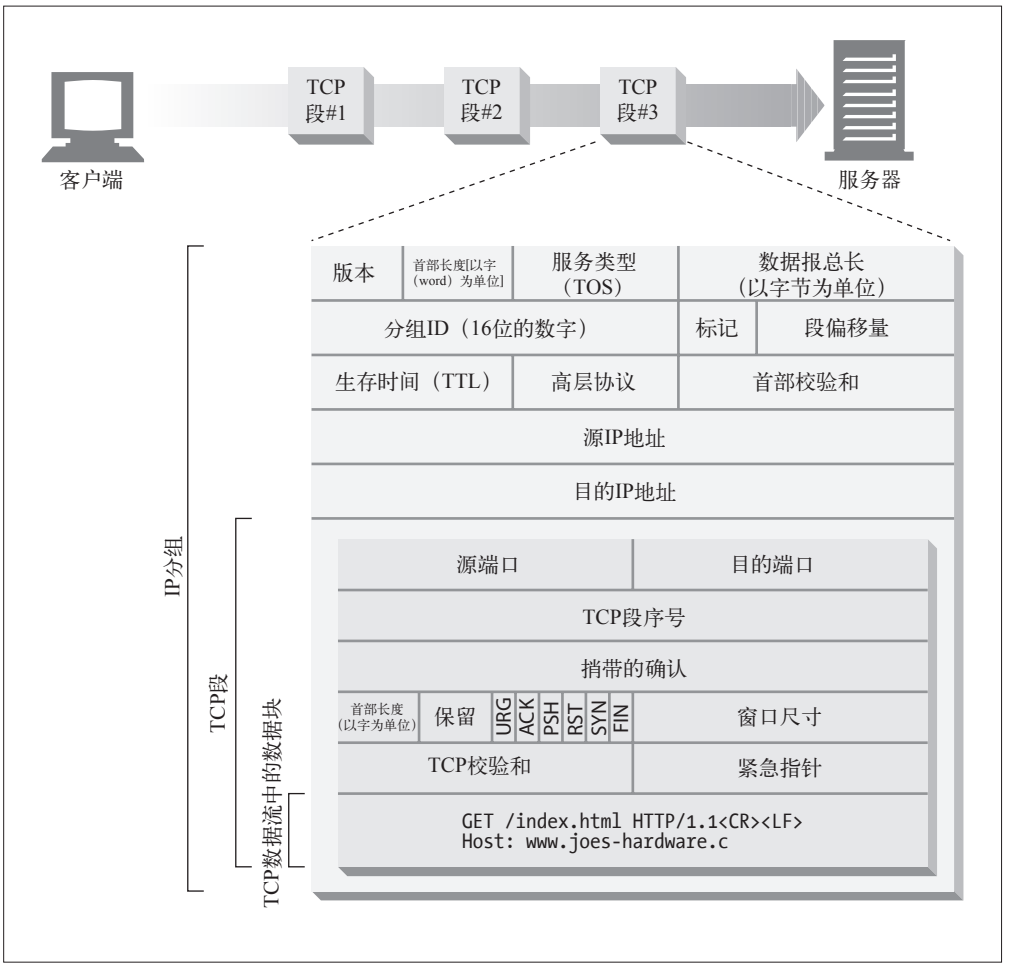


图 4-4　承载 TCP 段的 IP 分组，它承载了 TCP 数据流中的小块数据

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

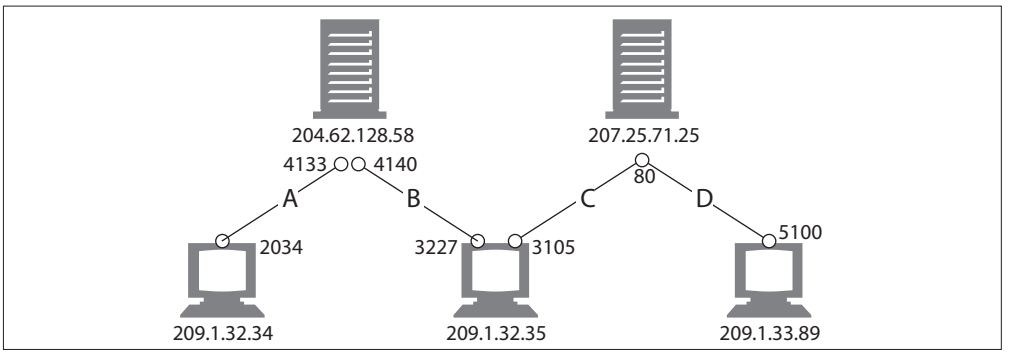


图 4-5　4 个不同的 TCP 连接

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。

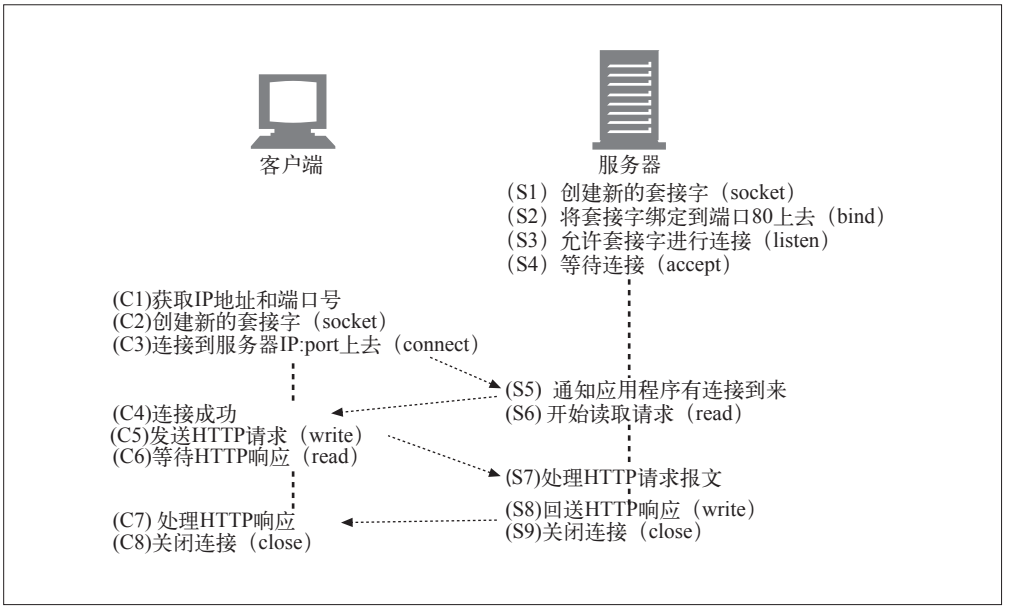


图 4-6　TCP 客户端和服务器是如何通过 TCP 套接字接口进行通信的

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）。

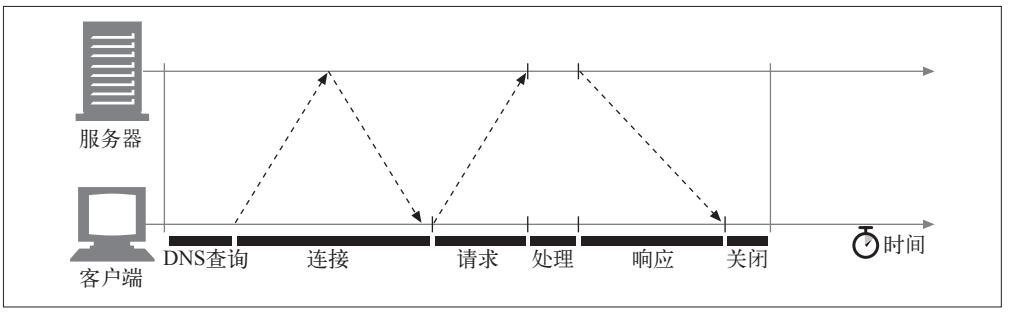
4.2　对TCP性能的考虑

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。

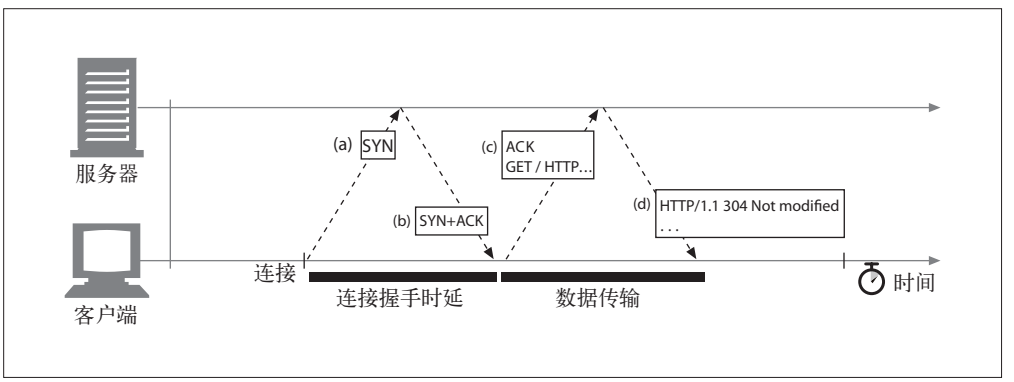


图 4-8 在发送数据之前，TCP 要传送两个分组来建立连接

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。

83

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
84 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，但超过此值时要特别

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。

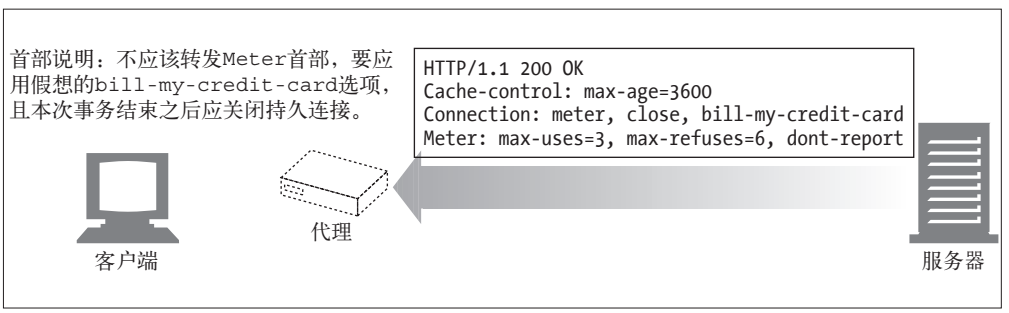


图 4-9 Connection 首部允许发送端指定与连接有关的选项

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]

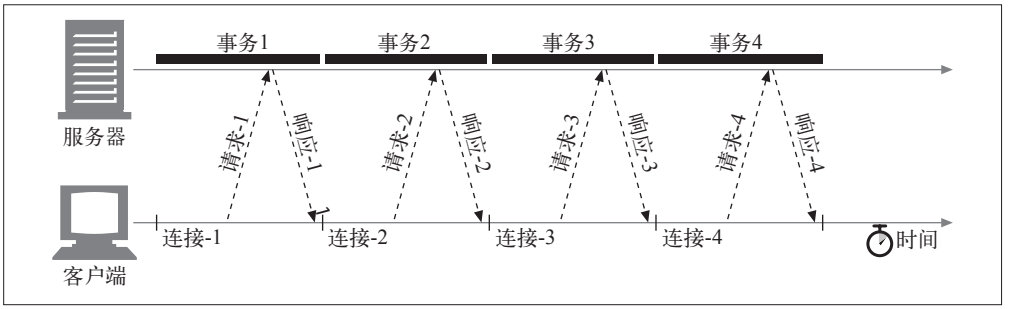


图 4-10　4 个事务（串行）

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]

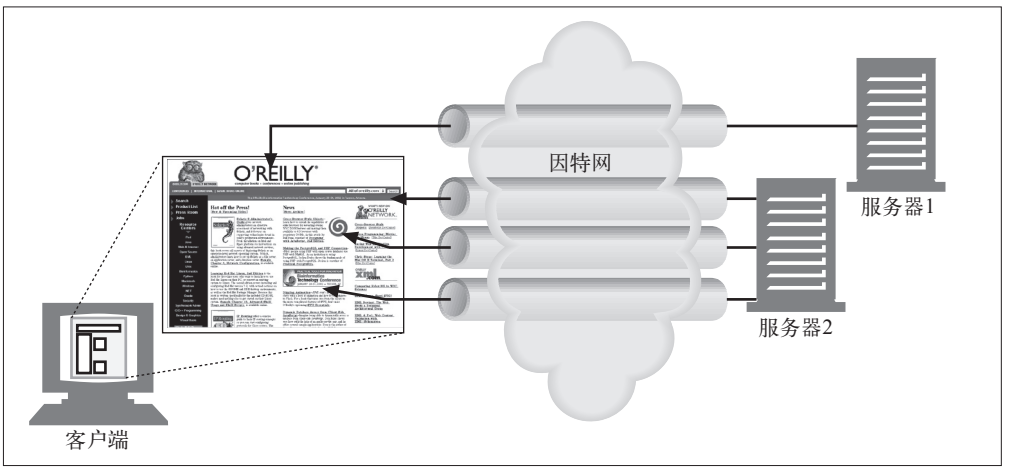


图 4-11 页面上的每个组件都包含一个独立的 HTTP 事务

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
88 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。

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。

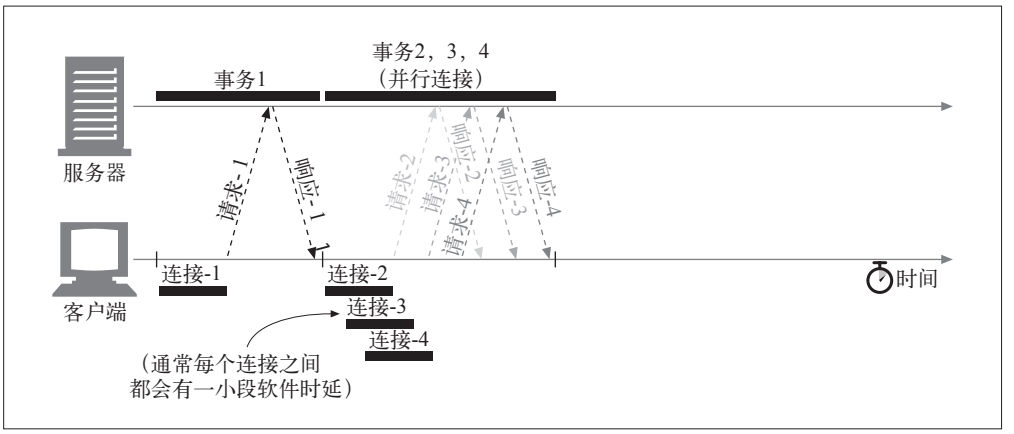


图 4-12　4 个事务（并行）

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。

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）。

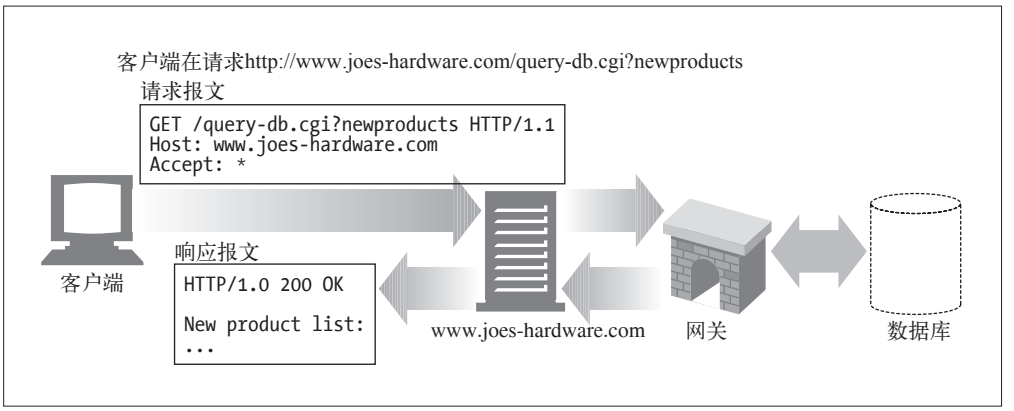


图 8-1　网关的魔力

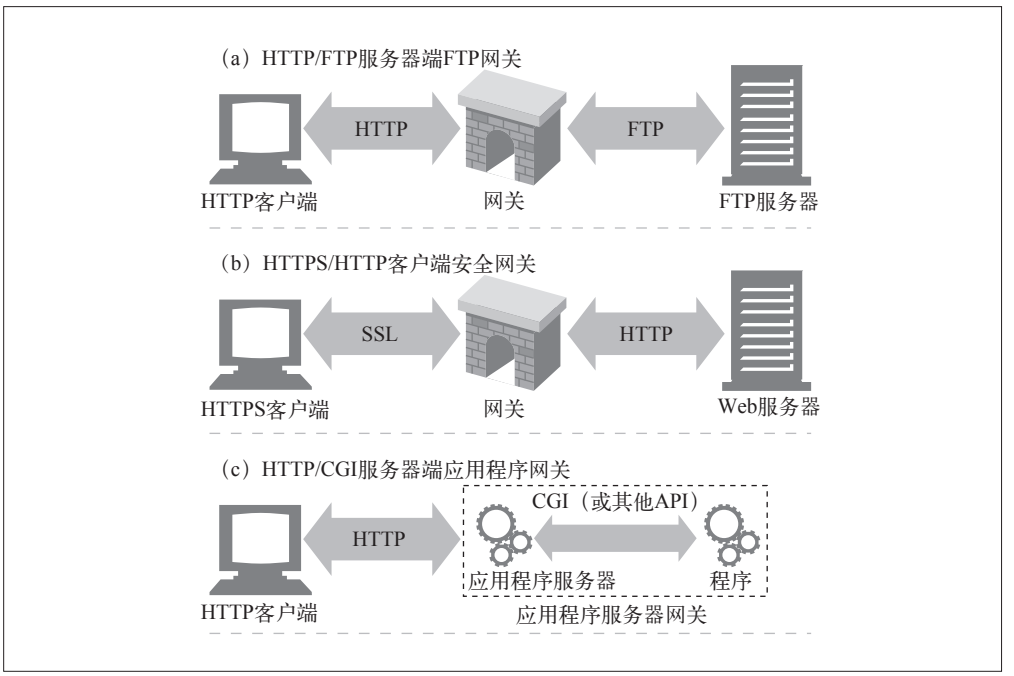


图 8-2　三个 Web 网关实例

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）。

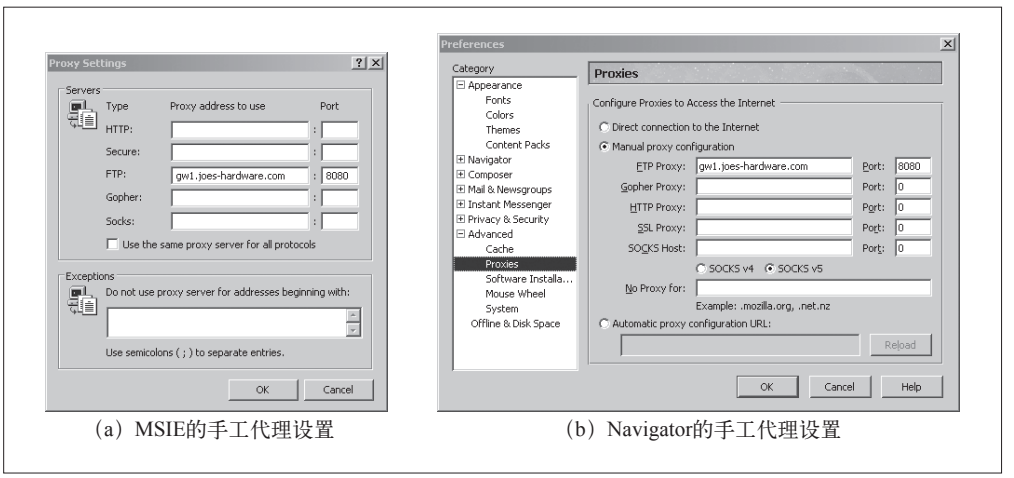
图 8-3 显示了配置浏览器使用服务器端 FTP 网关的对话框。在图中显示的配置中，配置浏览器将 gw1.joes-hardware.com 作为所有 FTP URL 的 HTTP/FTP 网关。浏览器没有将 FTP 命令发送给 FTP 服务器，而是将 HTTP 命令发送给端口 8080 上的 HTTP/FTP 网关 gw1.joes-hardware.com。

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

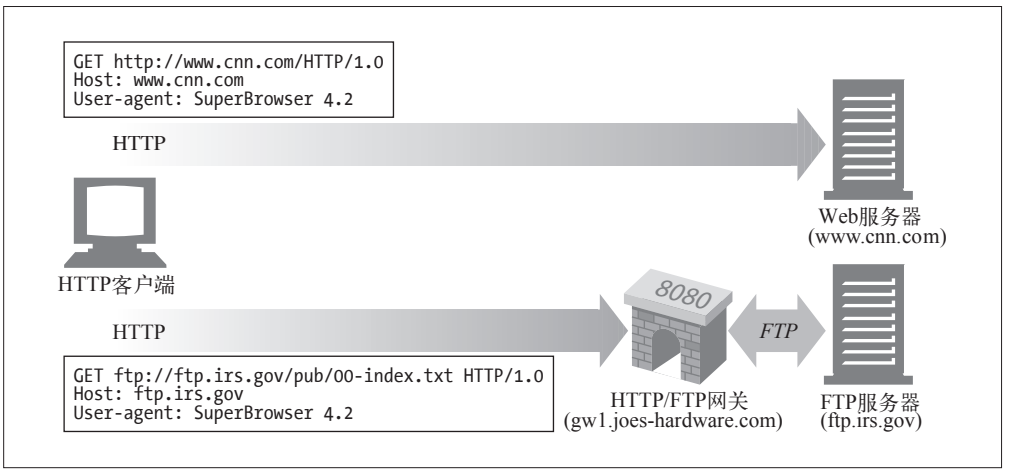


图 8-4　浏览器可以通过配置，让特定的协议使用特定的网关

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）。

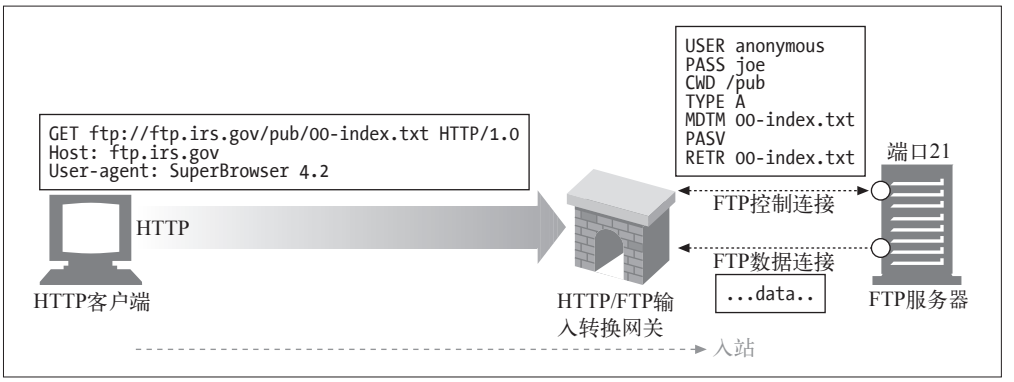


图 8-5　HTTP/FTP 网关将 HTTP 请求转换成 FTP 请求

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）。

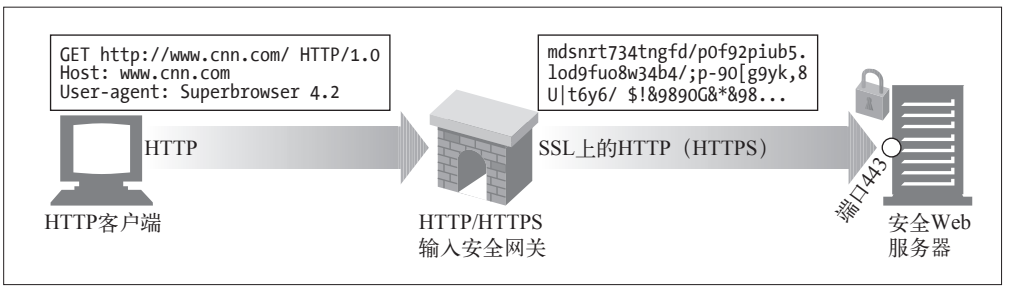


图 8-6　输入 HTTP/HTTPS 安全网关

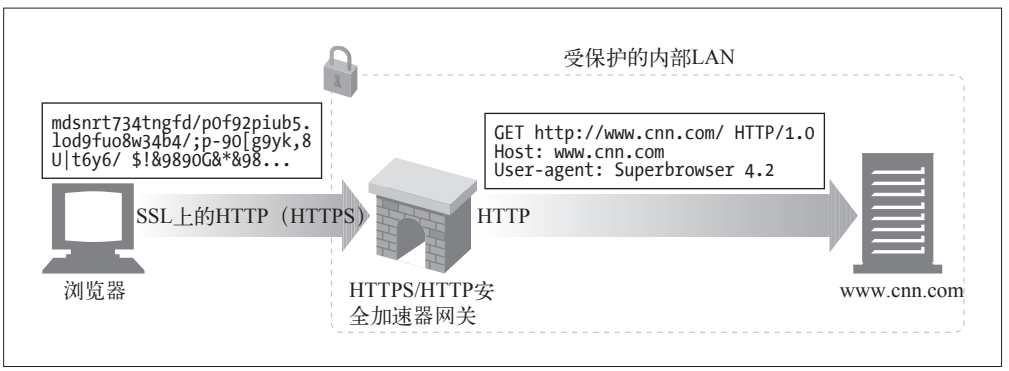


图 8-7　HTTPS/HTTP 安全加速器网关

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。

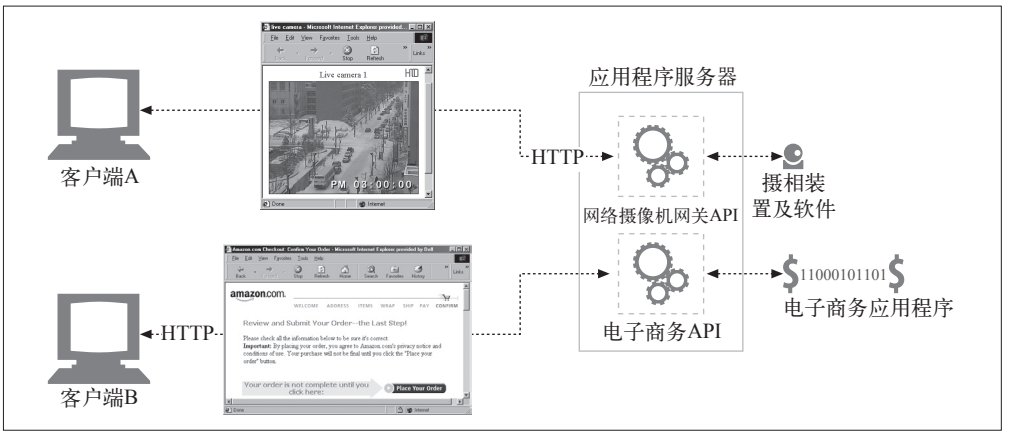


图 8-8　应用程序服务器可以将 HTTP 客户端连接任意后台应用程序

- 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。

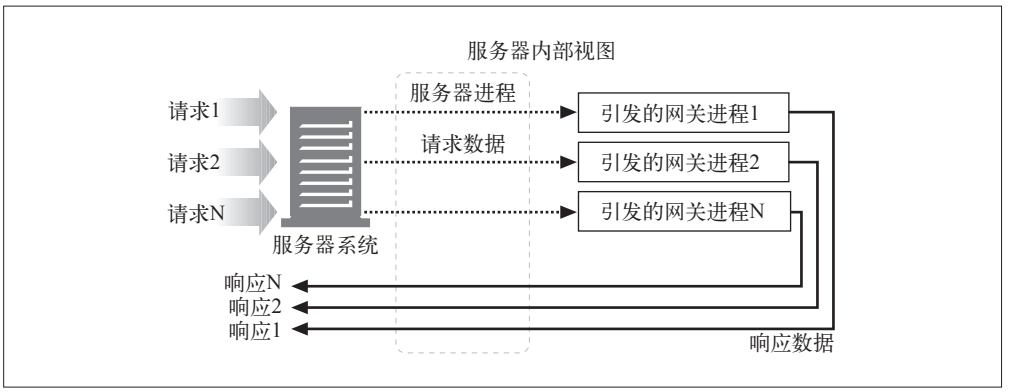


图 8-9　服务器网关应用程序机制

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

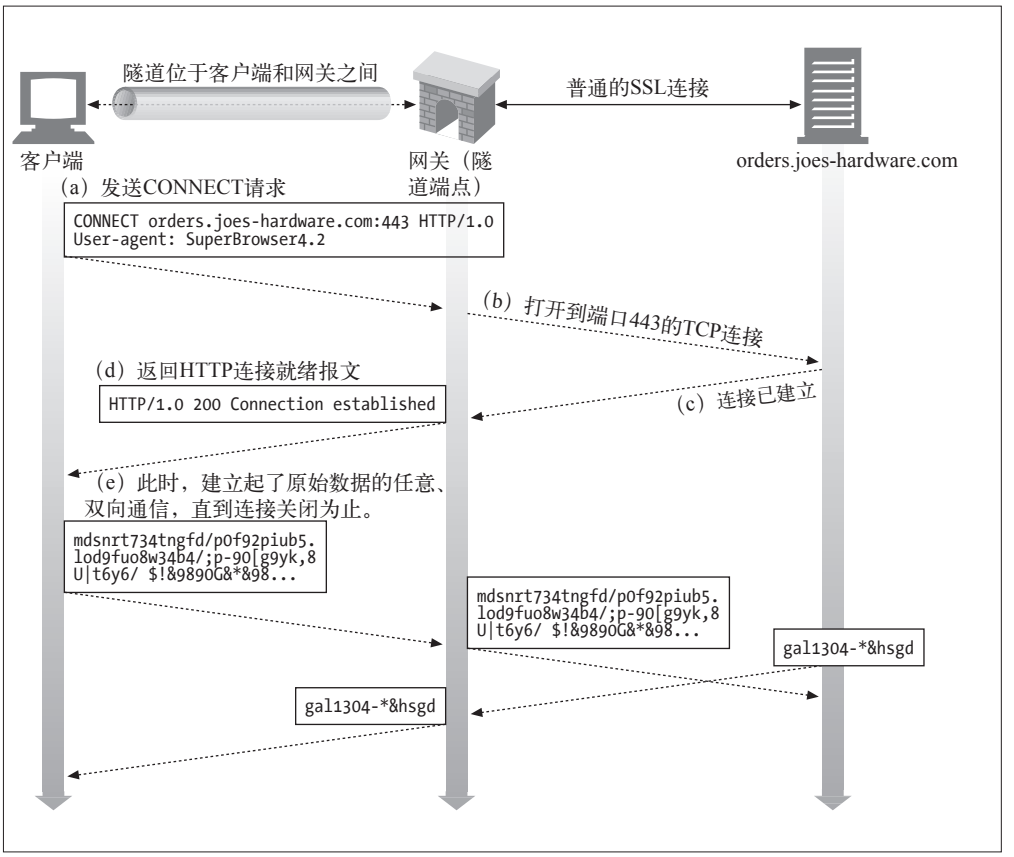
CONNECT 方法请求隧道网关创建一条到达任意目的服务器和端口的 TCP 连接，并对客户端和服务器之间的后继数据进行盲转发。

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，首部列表以一个空的 CRLF 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。

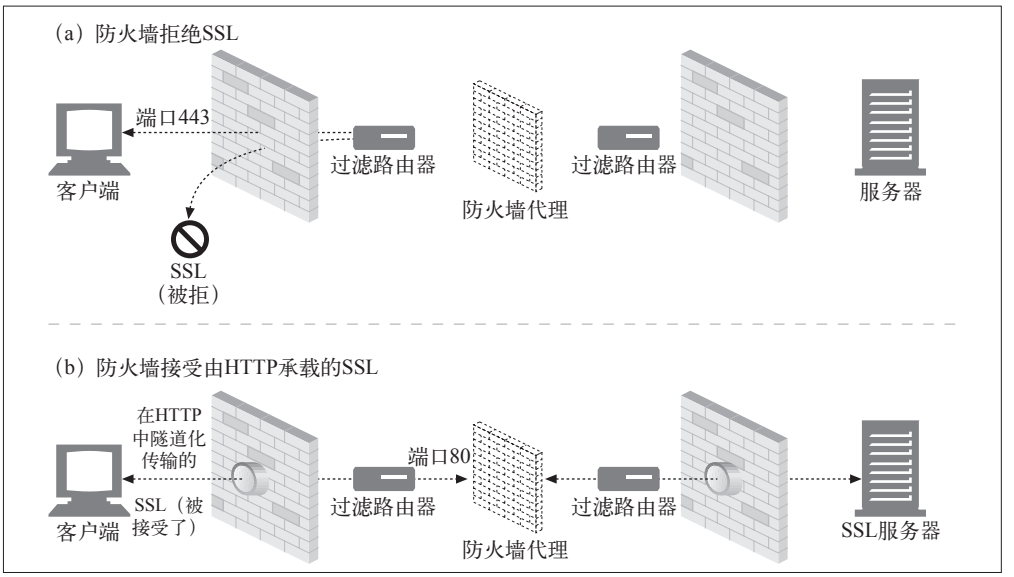


图 8-11　隧道可以经由 HTTP 连接传输非 HTTP 流量

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

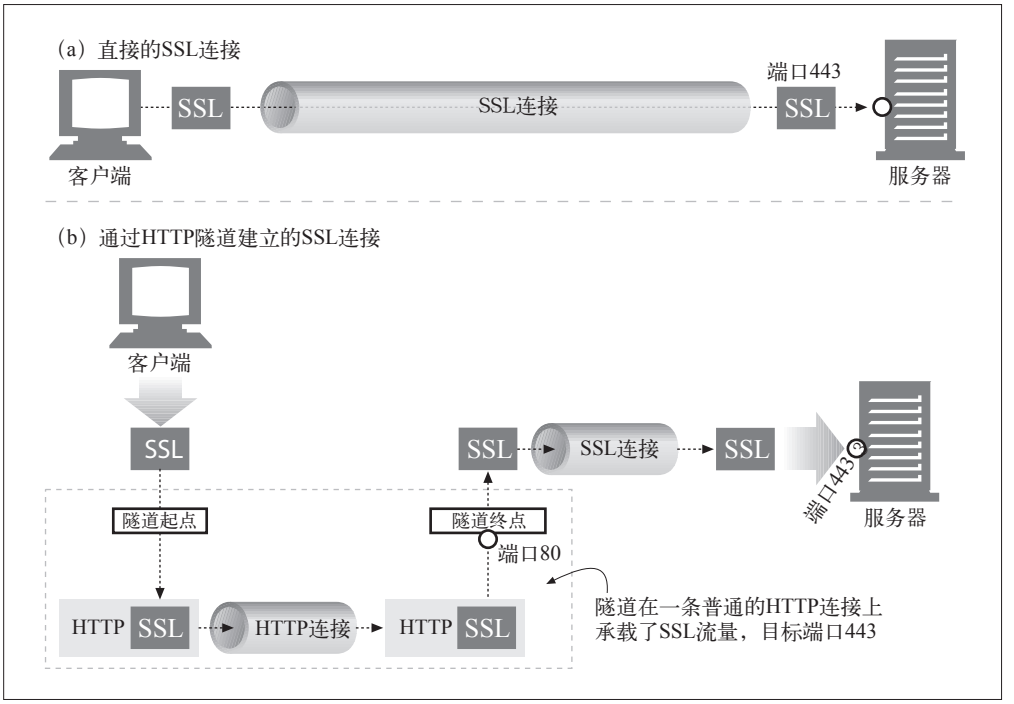


图 8-12　直接的 SSL 连接与隧道化 SSL 连接的对比

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。

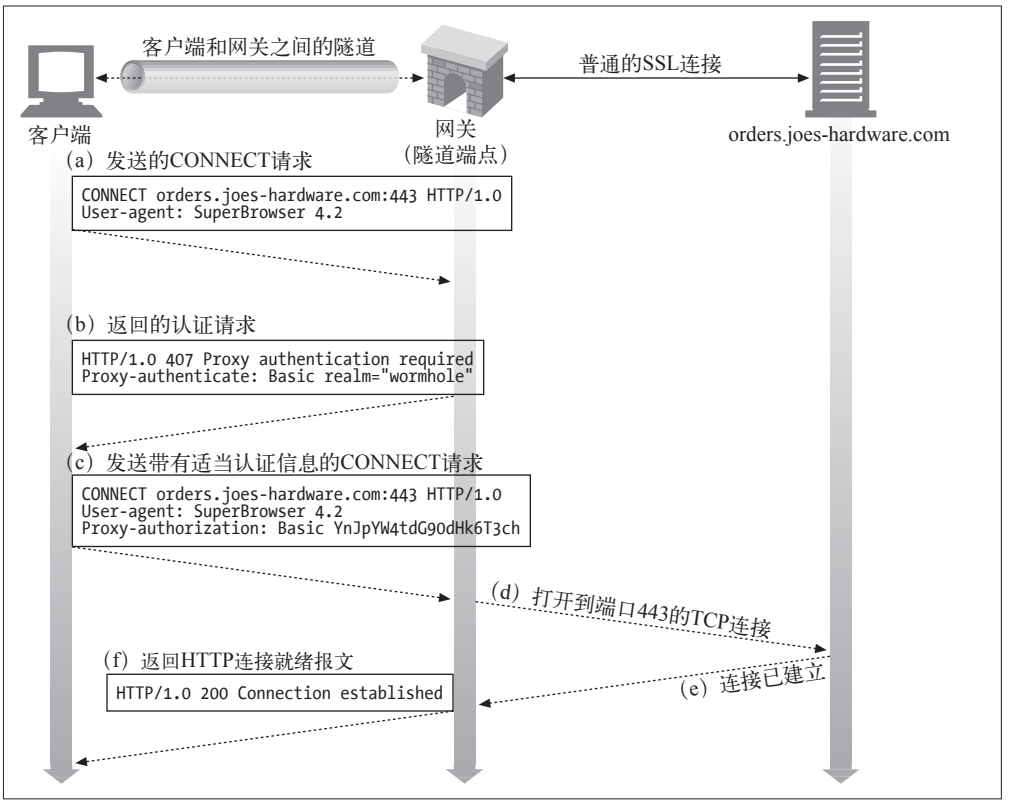


图 8-13　网关允许某客户端使用隧道之前，可以对其进行代理认证

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。

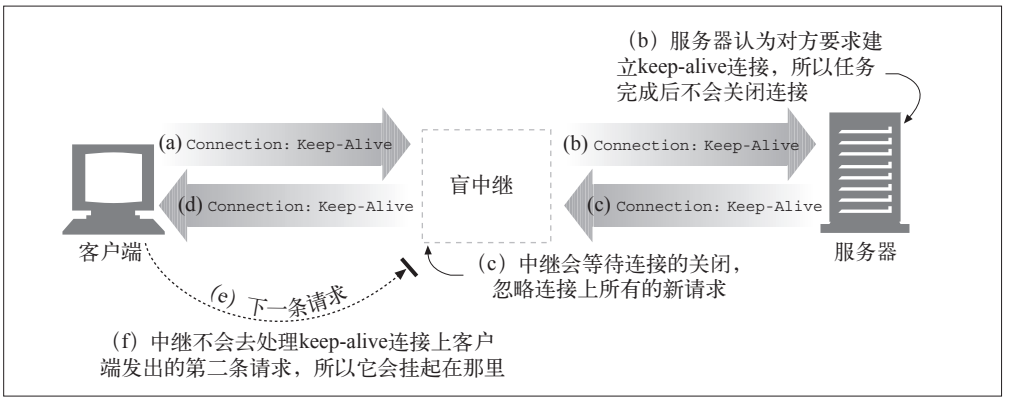


图 8-14　如果简单盲中继是单任务的，且不支持 Connection 首部，就会挂起

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

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）。

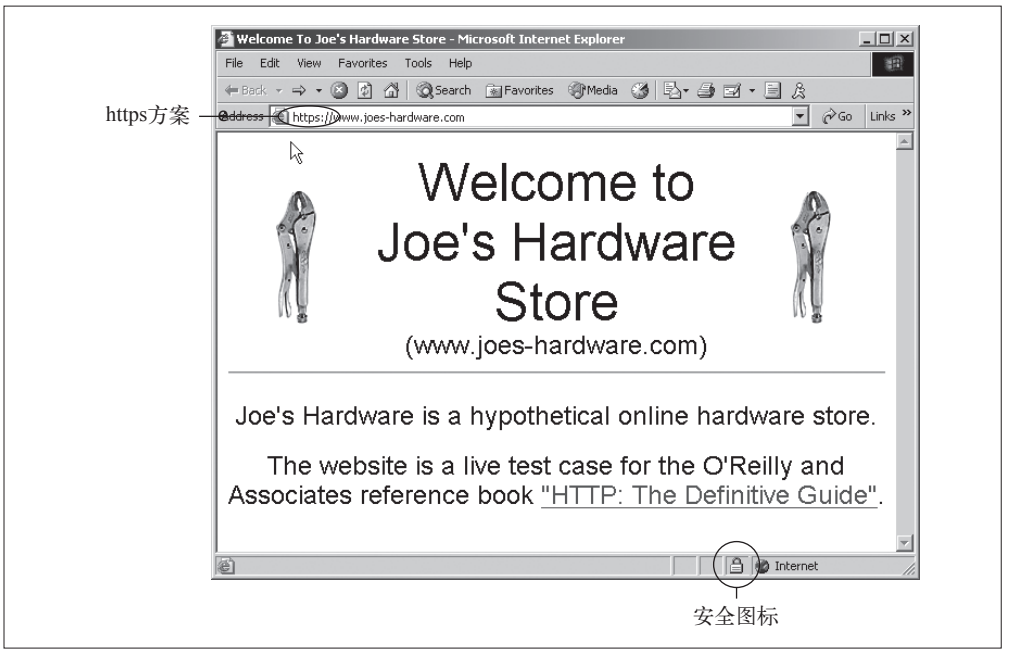


图 14-1　浏览安全 Web 站点

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。

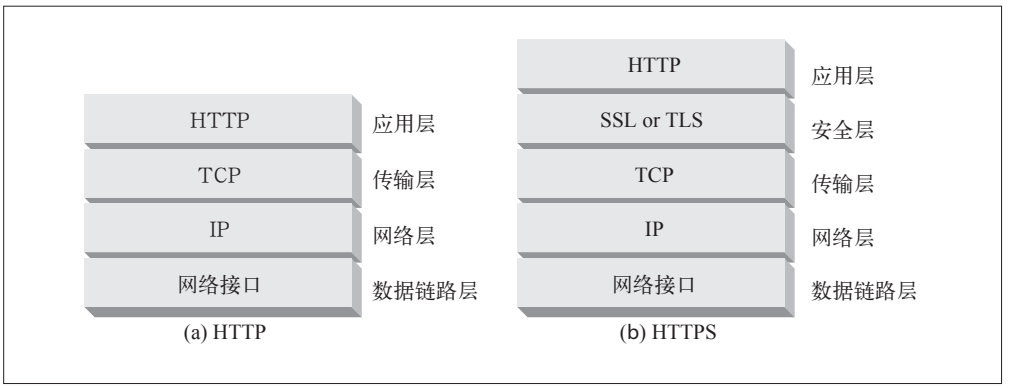


图 14-2　HTTPS 是位于安全层之上的 HTTP，这个安全层位于 TCP 之上

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。

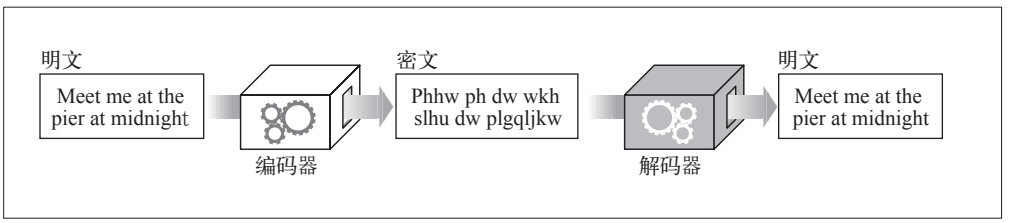


图 14-3　明文和密文

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旋转密码，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（旋转 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，在字母表中旋转 -3 个字符，就可以将密文解密回原来的明文报文。

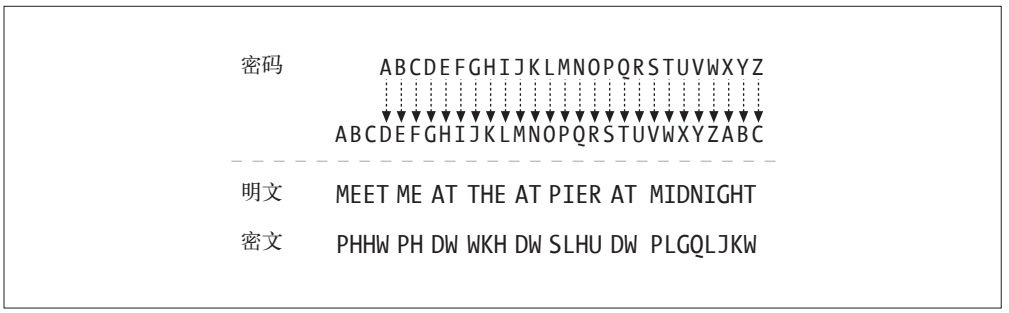


图 14-4　旋转 3 字符密码实例

310

注 1：为了简化这个例子，我们没有对标点和空格进行旋转，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“旋转 –*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！

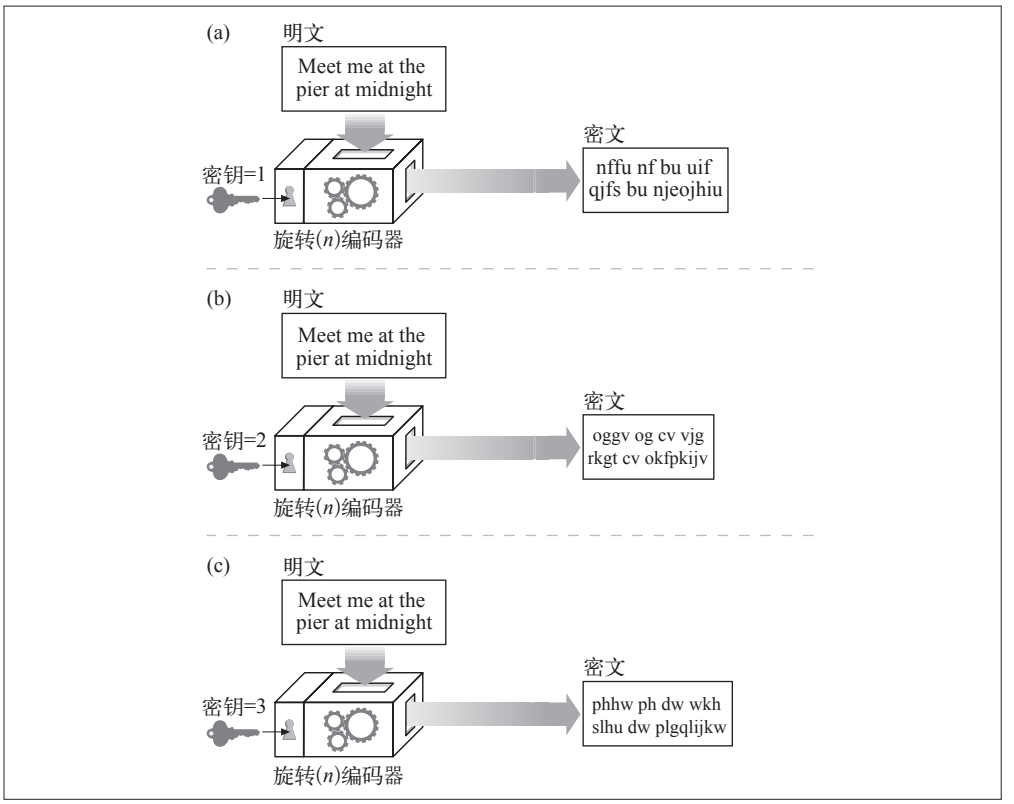


图 14-5　使用不同密钥的旋转 *N* 字符密码

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

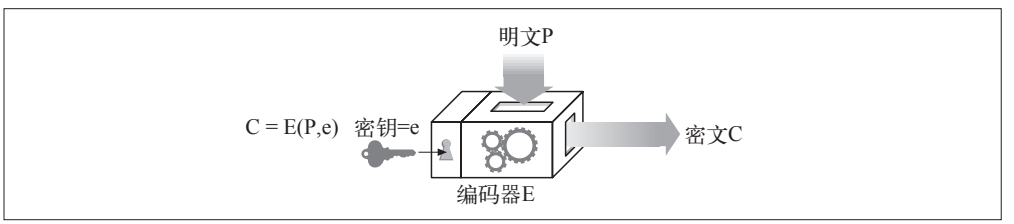


图 14-6　用编码密钥 e 对明文进行编码，用解码密钥 d 进行解码

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）。

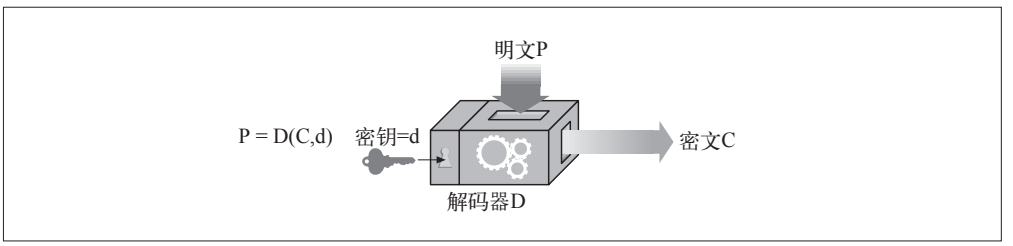


图 14-7 对称密钥加密算法为编 / 解码使用相同的密钥

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]

314

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。

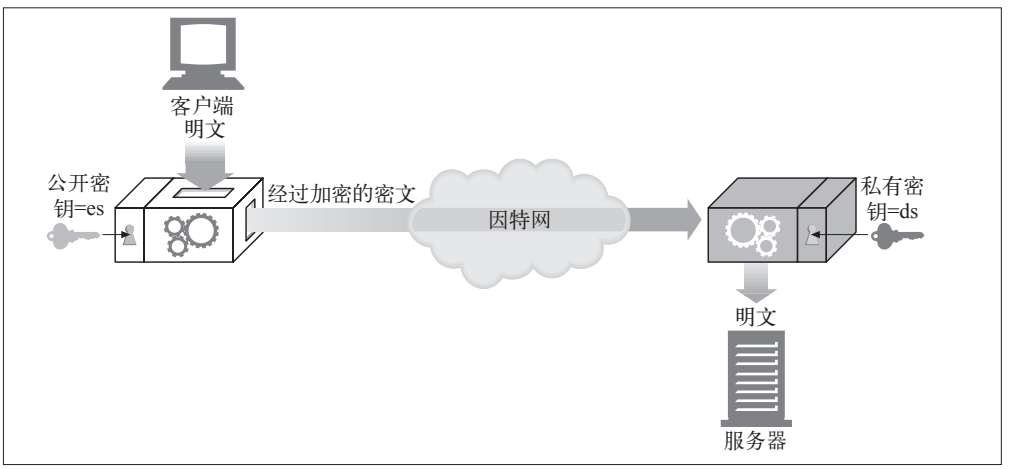


图 14-8 公开密钥加密技术是非对称的，为编码和解码使用了不同的密钥

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

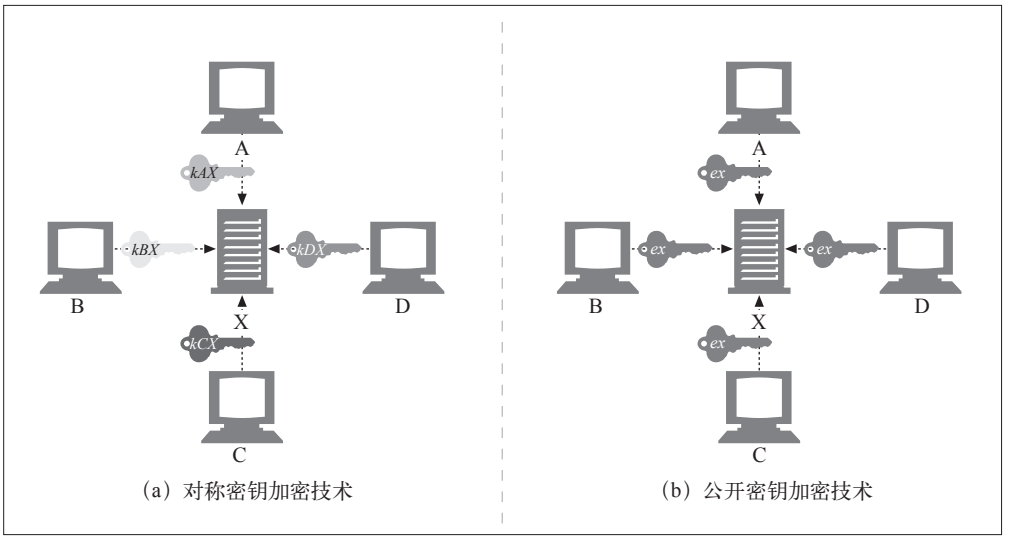


图 14-9　公开密钥加密技术为每台主机分配了一个公开编码密钥

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。

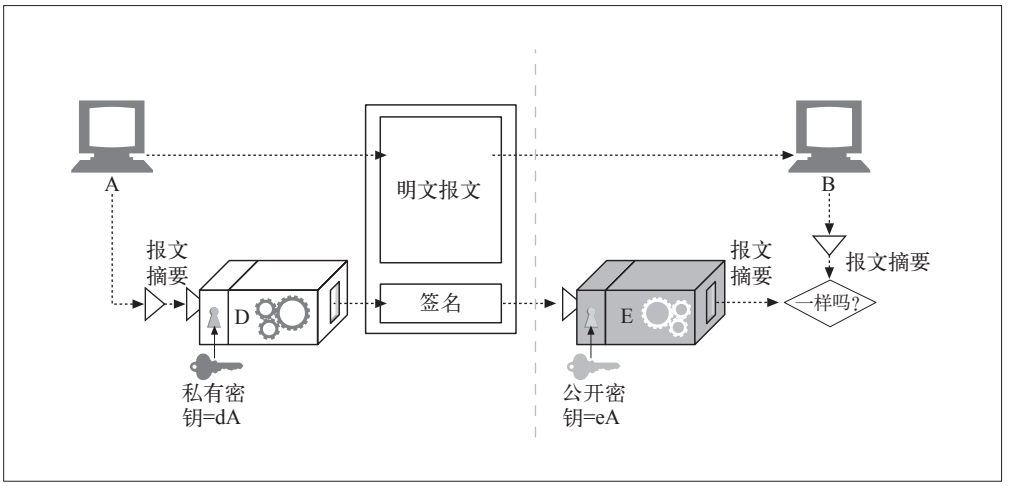


图 14-10　解密的数字签名

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。

- 在接收端，如果节点 B 需要确定报文确实是节点 A 写的，而且没有被篡改过，节点 B 就可以对签名进行检查。节点 B 接收经私有密钥扰码的签名，并应用了使用公开密钥的反函数。如果拆包后的摘要与节点 B 自己的摘要版本不匹配，要么就是报文在传输过程中被篡改了，要么就是发送端没有节点 A 的私有密钥（也
318 就是说它不是节点 A）。

14.6 数字证书

本节将介绍因特网上的“ID 卡”——数字证书。数字证书（通常被称作“certs”，有点像 certs 牌薄荷糖）中包含了由某个受信任组织担保的用户或公司的相关信息。

我们每个人都有很多形式的身份证明。有些 ID，比如护照和驾照，都足以在很多场合证明某人的身份。例如，你可以用美国的驾照在新年前夜搭乘前往纽约的航班，在你到那儿之后，接着用它来证明你的年龄，这样你就能和朋友们一起喝酒了。

受信程度更高的身份证明，比如护照，是由政府在特殊的纸上签发并盖章的。很难伪造，因此可以承载较高的信任度。有些公司的徽章和智能卡中包含有电子信息，以强化使用者的身份证明。有些绝密的政府组织甚至会对你的指纹或视网膜毛细血管模式进行匹配以便确认你的 ID！

有些形式的 ID，比如名片，相对来说更容易伪造，因此人们不太信任这些信息。虽然足以应付职场交流，但申请住房贷款时，可能就不足以证明你的就业情况了。

14.6.1 证书的主要内容

数字证书中还包含一组信息，所有这些信息都是由一个官方的“证书颁发机构”以数字方式签发的。基本的数字证书中通常包含一些纸质 ID 中常见的内容，比如：

- 对象的名称（人、服务器、组织等）；
319 • 过期时间；
- 证书发布者（由谁为证书担保）；
- 来自证书发布者的数字签名。

而且，数字证书通常还包括对象的公开密钥，以及对象和所用签名算法的描述性信息。任何人都可以创建一个数字证书，但并不是所有人都能够获得受人尊敬的签发权，从而为证书信息担保，并用其私有密钥签发证书。典型的证书结构如图 14-11 所示。

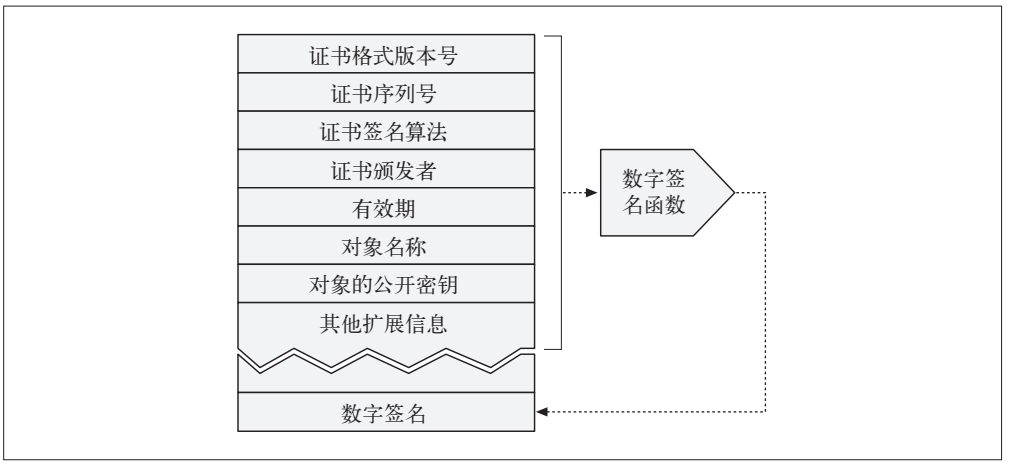


图 14-11　典型的数字签名格式

14.6.2　X.509 v3证书

不幸的是，数字证书没有单一的全球标准。就像不是所有印刷版 ID 卡都在同样的位置包含了同样的信息一样，数字证书也有很多略有不同的形式。 不过好消息就是现在使用的大多数证书都以一种标准格式——X.509 v3，来存储它们的信息。X.509 v3 证书提供了一种标准的方式，将证书信息规范至一些可解析字段中。不同类型的证书有不同的字段值，但大部分都遵循 X.509 v3 结构。表 14-2 介绍了 X.509 证书中的字段信息。

表14-2　X.509证书字段

字　段	描　述
版本	这个证书的 X.509 证书版本号。现在使用的通常都是版本 3
序列号	证书颁发机构（CA）生成的唯一整数。CA 生成的每个证书都要有一个唯一的序列号
签名算法 ID	签名所使用的加密算法。例如，“用 RSA 加密的 MD2 摘要”
证书颁发者	发布并签署这个证书的组织名称，以 X.500 格式表示
有效期	此证书何时有效，由一个起始日期和一个结束日期来表示
对象名称	证书中描述的实体，比如一个人或一个组织。对象名称是以 X.500 格式表示的
对象的公开密钥信息	证书对象的公开密钥，公开密钥使用的算法，以及所有附加参数
发布者唯一的 ID（可选）	可选的证书发布者唯一标识符，这样就可以重用相同的发布者名称
对象唯一的 ID（可选）	可选的证书对象唯一标识符，这样就可以重用相同的对象名称了

320

（续）

字　段	描　述
扩展	可选的扩展字段集（在版本 3 及更高的版本中使用）。每个扩展字段都被标识为关键或非关键的。关键扩展非常重要，证书使用者一定要能够理解。如果证书使用者无法识别出关键扩展字段，就必须拒绝这个证书。目前在使用的常用扩展字段包括： 基本约束 对象与证书颁发机构的关系 证书策略 授予证书的策略 密钥的使用 对公开密钥使用的限制
证书的颁发机构签名	证书颁发机构用指定的签名算法对上述所有字段进行的数字签名

基于 X.509 证书的签名有好几种，（其中）包括 Web 服务器证书、客户端电子邮件证书、软件代码签名证书和证书颁发机构证书。

14.6.3　用证书对服务器进行认证

通过 HTTPS 建立了一个安全 Web 事务之后，现代的浏览器都会自动获取所连接服务器的数字证书。如果服务器没有证书，安全连接就会失败。服务器证书中包含很多字段，其中包括：

- Web 站点的名称和主机名；
- Web 站点的公开密钥；
- 签名颁发机构的名称；
- 来自签名颁发机构的签名。

浏览器收到证书时会对签名颁发机构进行检查。[10] 如果这个机构是个很有权威的公
321 共签名机构，浏览器可能已经知道其公开密钥了（浏览器会预先安装很多签名颁发机构的证书）。这样，就可以像前面的 14.5 节中所讨论的那样验证签名了。图 14-12 说明了如何通过其数字签名来验证证书的完整性。

如果对签名颁发机构一无所知，浏览器就无法确定是否应该信任这个签名颁发机构，它通常会向用户显示一个对话框，看看他是否相信这个签名发布者。签名发布者可能是本地的 IT 部门或软件厂商。

注 10：浏览器和其他因特网应用程序都会尽量隐藏大部分证书管理的细节，使得浏览更加方便。但通过安全连接进行浏览时，所有主要的浏览器都允许你自己去检查所要对话站点的证书，以确保所有内容都是诚实可信的。

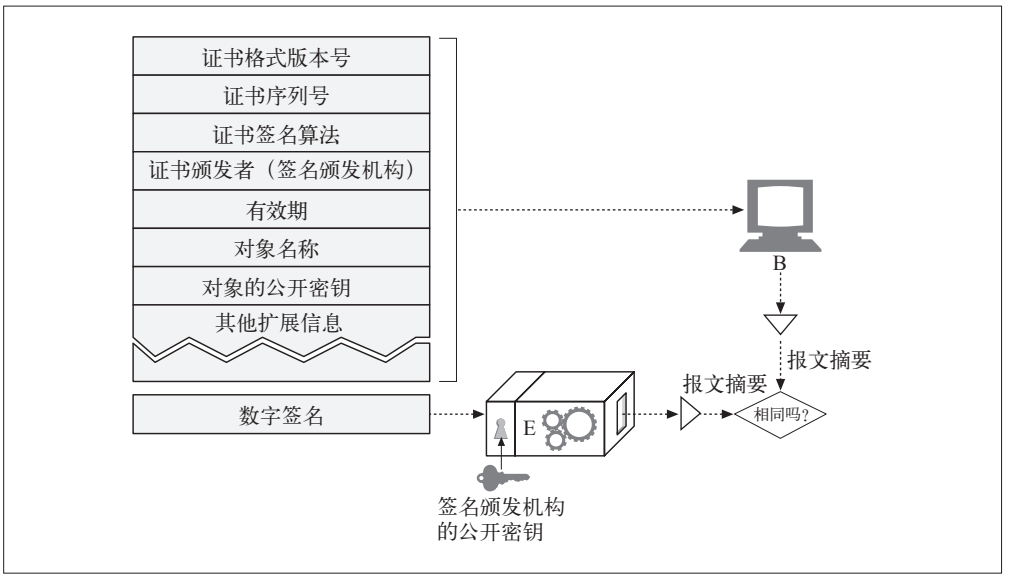


图 14-12　验证签名是真的

14.7　HTTPS——细节介绍

HTTPS 是最常见的 HTTP 安全版本。它得到了很广泛的应用，所有主要的商业浏览器和服务器上都提供 HTTPS。HTTPS 将 HTTP 协议与一组强大的对称、非对称和基于证书的加密技术结合在一起，使得 HTTPS 不仅很安全，而且很灵活，很容易在处于无序状态的、分散的全球互联网上进行管理。

HTTPS 加速了因特网应用程序的成长，已经成为基于 Web 的电子商务快速成长的主要推动力。在广域网中对分布式 Web 应用程序的安全管理方面，HTTPS 也是非常重要的。

14.7.1　HTTPS概述

HTTPS 就是在安全的传输层上发送的 HTTP。HTTPS 没有将未加密的 HTTP 报文发送给 TCP，并通过世界范围内的因特网进行传输（参见图 14-13a），它在将 HTTP 报文发送给 TCP 之前，先将其发送给了一个安全层，对其进行加密（参见图 14-13b）。

322

现在，HTTP 安全层是通过 SSL 及其现代替代协议 TLS 来实现的。我们遵循常见的用法，用术语 SSL 来表示 SSL 或者 TLS。

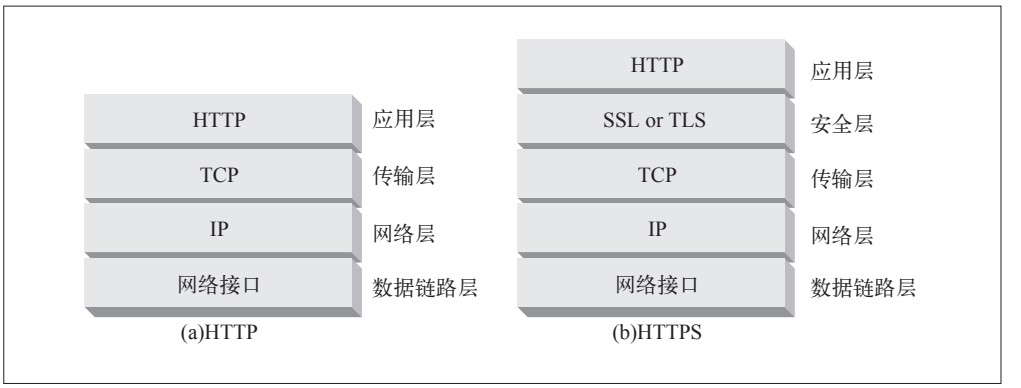


图 14-13　HTTP 传输层安全

14.7.2　HTTPS方案

现在，安全 HTTP 是可选的。因此，对 Web 服务器发起请求时，我们需要有一种方式来告知 Web 服务器去执行 HTTP 的安全协议版本。这是在 URL 的方案中实现的。

通常情况下，非安全 HTTP 的 URL 方案前缀为 http，如下所示：

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

在安全 HTTPS 协议中，URL 的方案前缀为 https，如下所示：

https://cajun-shop.securesites.com/Merchant2/merchant.mv?Store_Code=AGCGS

请求一个客户端（比如 Web 浏览器）对某 Web 资源执行某事务时，它会去检查 URL 的方案。

- 如果 URL 的方案为 http，客户端就会打开一条到服务器端口 80（默认情况下）的连接，并向其发送老的 HTTP 命令（参见图 14-14a）。
- 如果 URL 的方案为 https，客户端就会打开一条到服务器端口 443（默认情况下）的连接，然后与服务器“握手”，以二进制格式与服务器交换一些 SSL 安全参数，附上加密的 HTTP 命令（参见图 14-14b）。

SSL 是个二进制协议，与 HTTP 完全不同，其流量是承载在另一个端口上的（SSL 通常是由端口 443 承载的）。如果 SSL 和 HTTP 流量都从端口 80 到达，大部分 Web 服务器会将二进制 SSL 流量理解为错误的 HTTP 并关闭连接。将安全服务进一步整合到 HTTP 层中去就无需使用多个目的端口了，在实际中这样不会引发严重的问题。

323 我们来详细介绍下 SSL 是如何与安全服务器建立连接的。

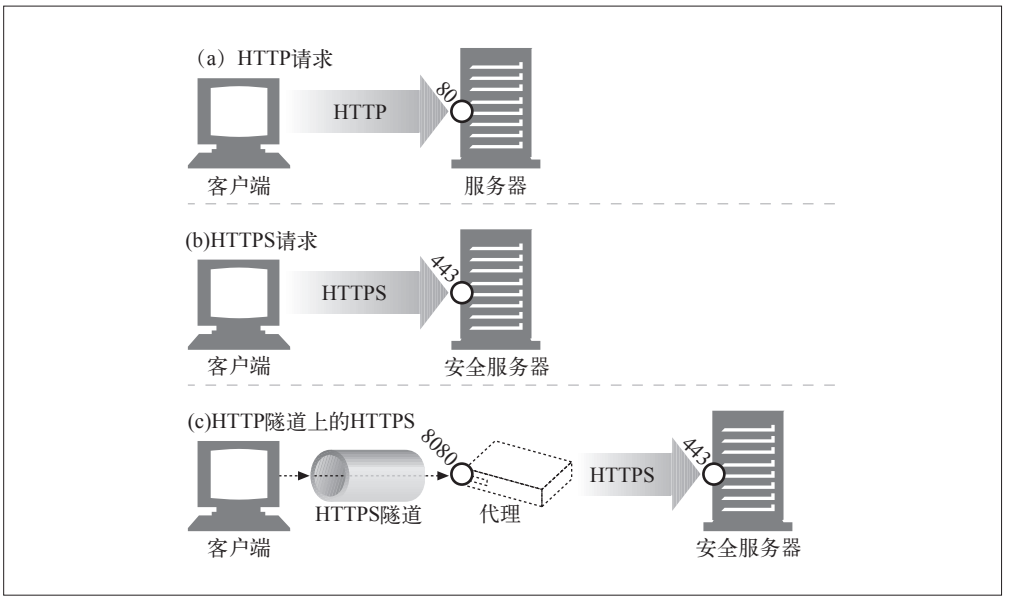


图 14-14　HTTP 和 HTTPS 端口号

14.7.3　建立安全传输

在未加密 HTTP 中，客户端会打开一条到 Web 服务器端口 80 的 TCP 连接，发送一条请求报文，接收一条响应报文，关闭连接。图 14-15a 对此序列进行了说明。

由于 SSL 安全层的存在，HTTPS 中这个过程会略微复杂一些。在 HTTPS 中，客户端首先打开一条到 Web 服务器端口 443（安全 HTTP 的默认端口）的连接。一旦建立了 TCP 连接，客户端和服务器就会初始化 SSL 层，对加密参数进行沟通，并交换密钥。握手完成之后，SSL 初始化就完成了，客户端就可以将请求报文发送给安全层了。在将这些报文发送给 TCP 之前，要先对其进行加密。图 14-15b 对此过程进行了说明。

14.7.4　SSL握手

在发送已加密的 HTTP 报文之前，客户端和服务器要进行一次 SSL 握手，在这个握手过程中，它们要完成以下工作：

- 交换协议版本号；
- 选择一个两端都了解的密码；
- 对两端的身份进行认证；
- 生成临时的会话密钥，以便加密信道。

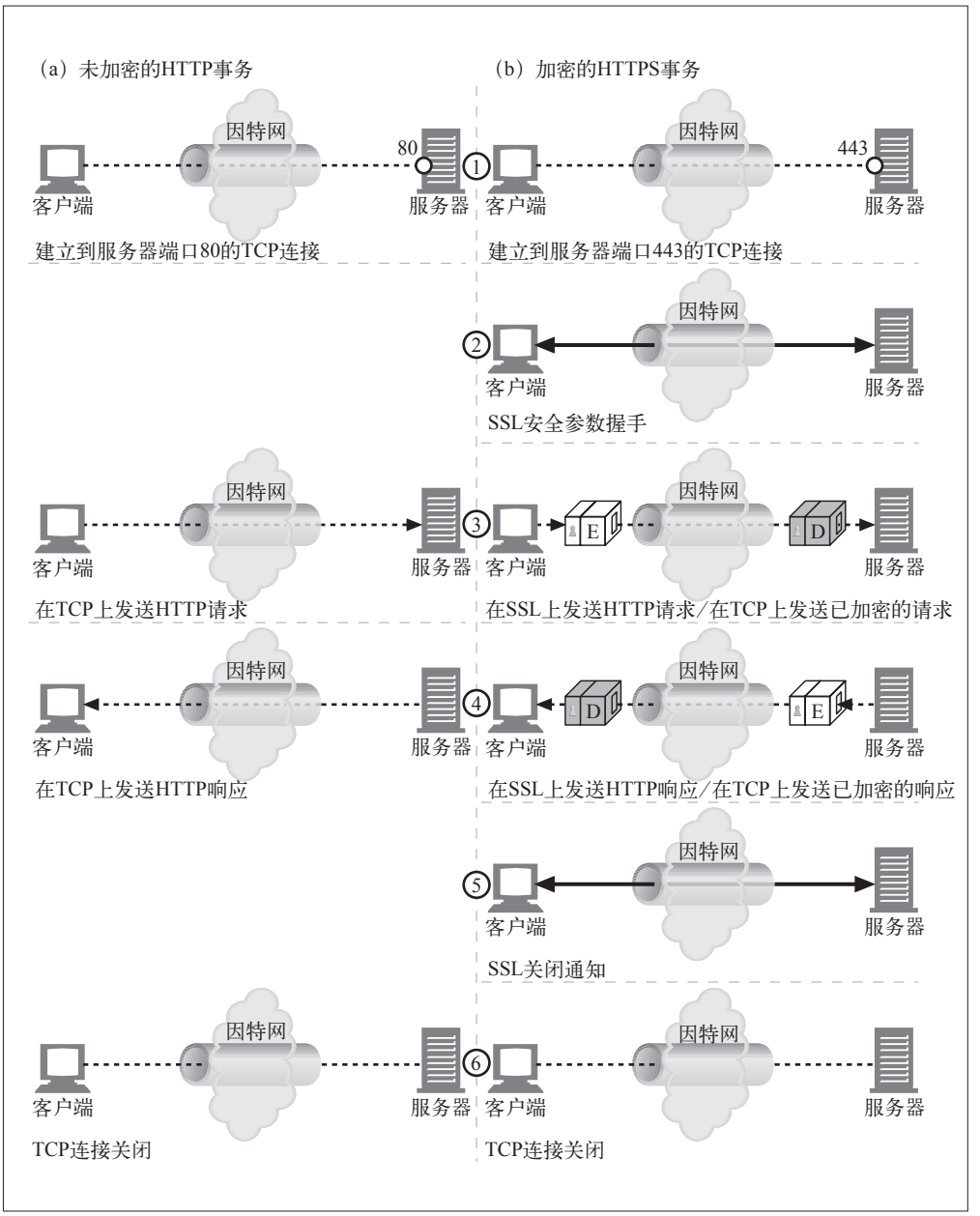


图 14-15　HTTP 和 HTTPS 事务

在通过网络传输任何已加密的 HTTP 数据之前，SSL 已经发送了一组握手数据来建立通信连接了。图 14-16 显示了 SSL 握手的基本思想。

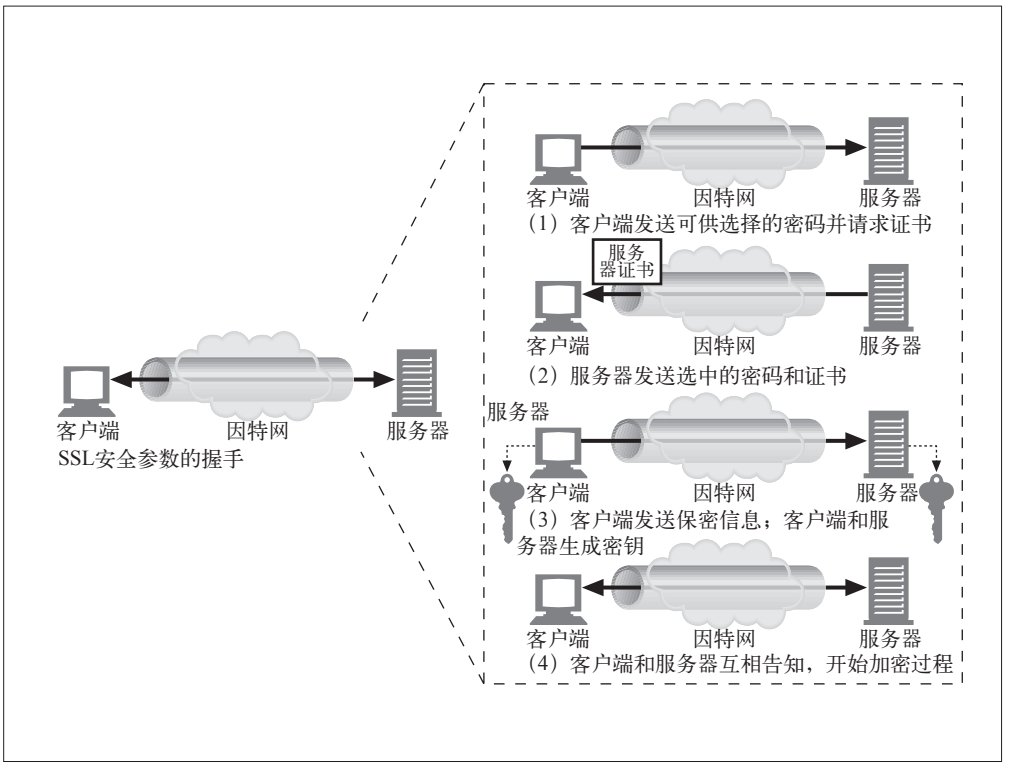


图 14-16 （简化版）SSL 握手

这是 SSL 握手的简化版本。根据 SSL 的使用方式，握手过程可能会复杂一些，但总的思想就是这样。

325

14.7.5 服务器证书

SSL 支持双向认证，将服务器证书承载回客户端，再将客户端的证书回送给服务器。而现在，浏览时并不经常使用客户端证书。大部分用户甚至都没有自己的客户端证书。[11] 服务器可以要求使用客户端证书，但实际中很少出现这种情况。[12]

另一方面，安全 HTTPS 事务总是要求使用服务器证书的。在一个 Web 服务器上执行安全事务，比如提交信用卡信息时，你总是希望是在与你所认为的那个组织对话。由知名权威机构签发的服务器证书可以帮助你在发送信用卡或私人信息之前评估你对服务器的信任度。

注 11：在某些公司的网络设置中会将客户端证书用于 Web 浏览，客户端证书还被用于安全电子邮件。未来，客户端证书可能会更经常地用于 Web 浏览，但现在它们发展的速度非常慢。

注 12：有些组织的内部网络会使用客户端证书来控制雇员对信息的访问。

服务器证书是一个显示了组织的名称、地址、服务器 DNS 域名以及其他信息的 X.509 v3 派生证书（参见图 14-17）。你和你所用的客户端软件可以检查证书，以确保所有的信息都是可信的。

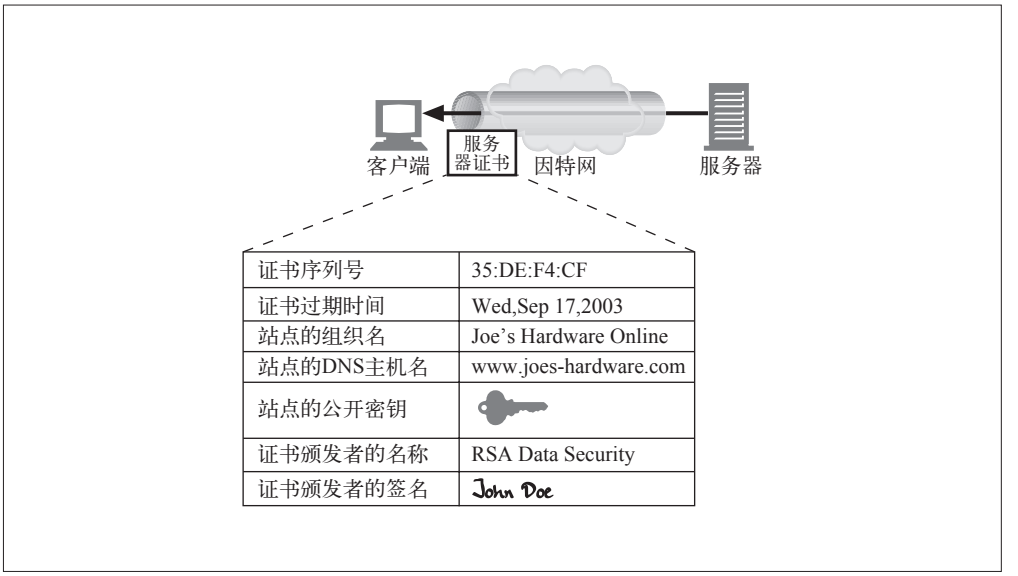


图 14-17　HTTPS 证书是带有站点信息的 X.509 证书

14.7.6　站点证书的有效性

SSL 自身不要求用户检查 Web 服务器证书，但大部分现代浏览器都会对证书进行简单的完整性检查，并为用户提供进行进一步彻查的手段。网景公司提出的一种 Web 服务器证书有效性算法是大部分浏览器有效性验证技术的基础。验证步骤如下所述。

- 日期检测

  首先，浏览器检查证书的起始日期和结束日期，以确保证书仍然有效。如果证书过期了，或者还未被激活，则证书有效性验证失败，浏览器显示一条错误信息。

- 签名颁发者可信度检测

  每个证书都是由某些证书颁发机构（CA）签发的，它们负责为服务器担保。证书有不同的等级，每种证书都要求不同级别的背景验证。比如，如果申请某个电子商务服务器证书，通常需要提供一个营业的合法证明。

  任何人都可以生成证书，但有些 CA 是非常著名的组织，它们通过非常清晰的流程来验证证书申请人的身份及商业行为的合法性。因此，浏览器会附带一个签名颁发机构的受信列表。如果浏览器收到了某未知（可能是恶意的）颁发机构

签发的证书，那它通常会显示一条警告信息。有些证书会携带到受信 CA 的有效签名路径，浏览器可能会选择接受所有此类证书。换句话说，如果某受信 CA 为“Sam 的签名商店”签发了一个证书，而 Sam 的签名商店也签发了一个站点证书，浏览器可能会将其作为从有效 CA 路径导出的证书接受。 327

- *签名检测*
  一旦判定签名授权是可信的，浏览器就要对签名使用签名颁发机构的公开密钥，并将其与校验码进行比较，以查看证书的完整性。

- *站点身份检测*
  为防止服务器复制其他人的证书，或拦截其他人的流量，大部分浏览器都会试着去验证证书中的域名与它们所对话的服务器的域名是否匹配。服务器证书中通常都包含一个域名，但有些 CA 会为一组或一群服务器创建一些包含了服务器名称列表或通配域名的证书。如果主机名与证书中的标识符不匹配，面向用户的客户端要么就去通知用户，要么就以表示证书不正确的差错报文来终止连接。

14.7.7 虚拟主机与证书

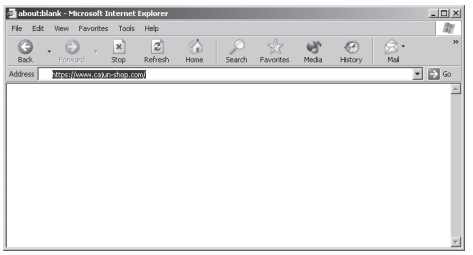
对虚拟主机（一台服务器上有多个主机名）站点上安全流量的处理有时是很棘手的。有些流行的 Web 服务器程序只支持一个证书。如果用户请求的是虚拟主机名，与证书名称并不严格匹配，浏览器就会显示警告框。

比如，我们来看以路易斯安那州为主题的电子商务网站 Cajun-Shop.com。站点的托管服务提供商提供的官方名称为 cajun-shop.securesites.com。用户进入 http://www.cajun-shop.com 时，服务器证书中列出的官方主机名（*.securesites.com）与用户浏览的虚拟主机名（www.cajun-shop.com）不匹配，以至出现图 14-18 中的警告。

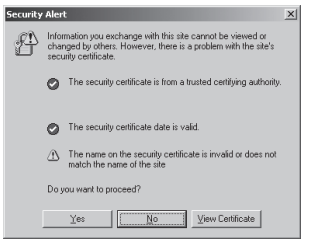
为防止出现这个问题，Cajun-Shop.com 的所有者会在开始处理安全事务时，将所有用户都重定向到 cajun-shop.securesites.com。虚拟主机站点的证书管理会稍微棘手一些。

14.8 HTTPS客户端实例

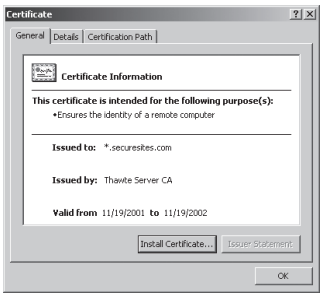
SSL 是个复杂的二进制协议。除非你是密码专家，否则就不应该直接发送原始的 SSL 流量。幸运的是，借助一些商业或开源的库，编写 SSL 客户端和服务器并不十分困难。



（a）由于站点是虚拟主机站点，而证书的主机名为*.securesites.com，所以这个URL（www.cajun-shop.com）中的主机名与证书中的名称不匹配。



(b)对话框警告用户站点证书的日期有效，而且来自有效的证书颁发机构，但证书中所列名称与URL所请求的站点不相符。



（c）为了获取更详细的信息，用户点击了“查看证书”按钮，看到证书是一个通配证书，主机名为*.securesites.com。有此信息之后，用户就可以判定是该接受还是该拒绝这个证书了。

(d)接受证书，通过安全HTTPS协议装载页面。为避免此类用户错误，这个特定的站点将所有 HTTPS流量都导向了主机别名cajun-shop.securesites.com。这个虚拟主机名与ISP在其商业包中提供的证书名字相符。

图 14-18　证书名不匹配引发的证书错误对话框

14.8.1　OpenSSL

OpenSSL 是 SSL 和 TLS 最常见的开源实现。OpenSSL 项目由一些志愿者合作开发，目标是开发一个强壮的、具有完备功能的商业级工具集，以实现 SSL 和 TLS 协议以及一个全功能的通用加密库。可以从 http://www.openssl.org 上获得 OpenSSL 的相关信息，并下载相应软件。

328

你可能还听说过 SSLeay（读作 S-S-L-e-a-y）。OpenSSL 是 SSLeay 库的后继者，接口非常相似。SSLeay 最初是由 Eric A. Young（就是 SSLeay 中的“eay”）开发的。

14.8.2 简单的HTTPS客户端

本节我们将用 OpenSSL 包来编写一个非常初级的 HTTPS 客户端。这个客户端与服务器建立一条 SSL 连接，打印一些来自站点服务器的标识信息，通过安全信道发送 HTTP GET 请求，接收 HTTP 响应，并将响应打印出来。 329

下面显示的 C 程序是普通 HTTPS 客户端的 OpenSSL 实现。为了保持其简洁性，程序中没有包含差错处理和证书处理逻辑。

这个示例程序中删除了差错处理功能，所以只能将其用于示例。在一般的有差错存在的环境中，软件会崩溃或者无法正常运行。 330

```
/**********************************************************************
* https_client.c --- very simple HTTPS client with no error checking
*      usage: https_client servername
**********************************************************************/

#include <stdio.h>
#include <memory.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <openssl/crypto.h>
#include <openssl/x509.h>
#include <openssl/pem.h>
#include <openssl/ssl.h>
#include <openssl/err.h>

void main(int argc, char **argv)
{

    SSL *ssl;
    SSL_CTX *ctx;
    SSL_METHOD *client_method;
    X509 *server_cert;
    int sd,err;
    char *str,*hostname,outbuf[4096],inbuf[4096],host_header[512];
    struct hostent *host_entry;
    struct sockaddr_in server_socket_address;
    struct in_addr ip;

    /*=======================================*/
    /* (1) initialize SSL library */
    /*=======================================*/
```

```
SSLeay_add_ssl_algorithms( );
client_method = SSLv2_client_method( );
SSL_load_error_strings( );
ctx = SSL_CTX_new(client_method);
printf("(1) SSL context initialized\n\n");
/*============================================*/
/* (2) convert server hostname into IP address */
/*============================================*/

hostname = argv[1];
host_entry = gethostbyname(hostname);
bcopy(host_entry->h_addr, &(ip.s_addr), host_entry->h_length);

printf("(2) '%s' has IP address '%s'\n\n", hostname, inet_ntoa(ip));
/*================================================*/
/* (3) open a TCP connection to port 443 on server */
/*================================================*/

sd = socket (AF_INET, SOCK_STREAM, 0);

memset(&server_socket_address, '\0', sizeof(server_socket_address));
server_socket_address.sin_family = AF_INET;
server_socket_address.sin_port = htons(443);
memcpy(&(server_socket_address.sin_addr.s_addr),
       host_entry->h_addr, host_entry->h_length);

err = connect(sd, (struct sockaddr*) &server_socket_address,
            sizeof(server_socket_address));
if (err < 0) { perror("can't connect to server port"); exit(1); }

printf("(3) TCP connection open to host '%s', port %d\n\n",
       hostname, server_socket_address.sin_port);

/*=========================================================*/
/* (4) initiate the SSL handshake over the TCP connection */
/*=========================================================*/

ssl = SSL_new(ctx);     /* create SSL stack endpoint */
SSL_set_fd(ssl, sd);    /* attach SSL stack to socket */
err = SSL_connect(ssl); /* initiate SSL handshake */

printf("(4) SSL endpoint created & handshake completed\n\n");

/*============================================*/
/* (5) print out the negotiated cipher chosen */
/*============================================*/

printf("(5) SSL connected with cipher: %s\n\n", SSL_get_cipher(ssl));

/*=======================================*/
/* (6) print out the server's certificate */
/*=======================================*/

server_cert = SSL_get_peer_certificate(ssl);
printf("(6) server's certificate was received:\n\n");
str = X509_NAME_oneline(X509_get_subject_name(server_cert), 0, 0);
printf("      subject: %s\n", str);
```

```
    str = X509_NAME_oneline(X509_get_issuer_name(server_cert), 0, 0);
    printf("      issuer: %s\n\n", str);

    /* certificate verification would happen here */

    X509_free(server_cert);

    /*---------------------------------------------------------*/
    /* (7) handshake complete --- send HTTP request over SSL */
    /*---------------------------------------------------------*/

    sprintf(host_header,"Host: %s:443\r\n",hostname);
    strcpy(outbuf,"GET / HTTP/1.0\r\n");
    strcat(outbuf,host_header);
    strcat(outbuf,"Connection: close\r\n");
    strcat(outbuf,"\r\n");

    err = SSL_write(ssl, outbuf, strlen(outbuf));
    shutdown (sd, 1); /* send EOF to server */

    printf("(7) sent HTTP request over encrypted channel:\n\n%s\n",outbuf);

    /*-------------------------------------------------*/
    /* (8) read back HTTP response from the SSL stack */
    /*-------------------------------------------------*/

    err = SSL_read(ssl, inbuf, sizeof(inbuf) - 1);
    inbuf[err] = '\0';
    printf ("(8) got back %d bytes of HTTP response:\n\n%s\n",err,inbuf);

    /*-------------------------------------------------*/
    /* (9) all done, so close connection & clean up */
    /*-------------------------------------------------*/

    SSL_shutdown(ssl);
    close (sd);
    SSL_free (ssl);
    SSL_CTX_free (ctx);

    printf("(9) all done, cleaned up and closed connection\n\n");
}
```

这个例子是在 Sun Solaris 上面编译运行的，但它说明了 SSL 在很多 OS 平台上的工作原理。由于 OpenSSL 提供了一些强有力的特性，包括所有加密、密钥以及证书管理在内的整个程序都可以在一个几页左右的 C 程序中实现。

331 ~ 332

下面按部分分析下这个程序。

- 程序的顶端包含了一些用于支持 TCP 联网和 SSL 的支撑文件。
- 第 1 部分通过调用 `SSL_CTX_new` 创建了本地上下文，以记录握手参数及与 SSL 连接有关的其他状态。
- 第 2 部分通过 Unix 的 `gethostbyname` 函数将（由一个命令行变元提供的）输入主机名转换成了 IP 地址。其他平台可能会通过其他方式来提供这项功能。

- 第 3 部分通过创建本地套接字、设置远端地址信息并连接到远端服务器，建立了一条到服务器端口 443 的 TCP 连接。
- 一旦 TCP 连接建立起来，就用 `SSL_new` 和 `SSL_set_fd` 将 SSL 层附加到 TCP 连接之上，并调用 `SSL_connect` 与服务器进行 SSL 握手。第 4 部分完成时，我们就建立了一个已选好密码且交换过证书的可运行的 SSL 信道。
- 第 5 部分打印了选中的批量加密密码值。
- 第 6 部分打印了服务器回送的 X.509 证书中包含的部分信息，其中包括与证书持有者和颁发证书的组织有关的信息。OpenSSL 库没有对服务器证书中的信息作任何特殊的处理。实际的 SSL 应用程序，比如 Web 浏览器会对证书进行一些完整性检查，以确保证书是正确签发的，且是来自正确主机的。我们在 14.7.6 节讨论了浏览器对服务器证书所做的处理。
- 此时，我们的 SSL 连接就已经可以用于安全数据的传输了。在第 7 部分中，用 `SSL_write` 在 SSL 信道上发送了简单的 HTTP 请求 GET / HTTP/1.0，然后关闭了连接的输出端。
- 在第 8 部分中，用 `SSL_read` 从连接上读回响应，并将其打印到屏幕上。SSL 层负责所有的加密和解密工作，因此可以直接读写普通的 HTTP 命令。
- 最后，在第 9 部分进行了一些清理工作。

更多与 OpenSSL 库有关的信息请参见 http://www.openssl.org。

14.8.3 执行OpenSSL客户端

下面显示了指向安全服务器时这个简单 HTTP 客户端的输出。在这个例子中，客户端指向了摩根士丹利的在线证券主页。在线交易公司都在广泛使用 HTTPS。

333

```
% https_client clients1.online.msdw.com
(1) SSL context initialized

(2) 'clients1.online.msdw.com' has IP address'63.151.15.11'

(3) TCP connection open to host 'clients1.online.msdw.com', port 443

(4) SSL endpoint created & handshake completed

(5) SSL connected with cipher: DES-CBC3-MD5

(6) server's certificate was received:

      subject: /C=US/ST=Utah/L=Salt Lake City/O=Morgan Stanley/OU=Online/CN=
        clients1.online.msdw.com
      issuer: /C=US/O=RSA Data Security, Inc./OU=Secure Server Certification
        Authority
(7) sent HTTP request over encrypted channel:
```

```
GET / HTTP/1.0
Host: clients1.online.msdw.com:443
Connection: close

(8) got back 615 bytes of HTTP response:

HTTP/1.1 302 Found
Date: Sat, 09 Mar 2002 09:43:42 GMT
Server: Stronghold/3.0 Apache/1.3.14 RedHat/3013c (Unix) mod_ssl/2.7.1 OpenSSL/0.9.6
Location: https://clients.online.msdw.com/cgi-bin/ICenter/home
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
<TITLE>302 Found</TITLE>
</HEAD><BODY>
<H1>Found</H1>
The document has moved <A HREF="https://clients.online.msdw.com/cgi-bin/ICenter/
home">here</A>.<P>
<HR>
<ADDRESS>Stronghold/3.0Apache/1.3.14 RedHat/3013c Server at clients1.online.msdw.com
Port 443</ADDRESS>
</BODY></HTML>

(9) all done, cleaned up and closed connection
```

只要完成了前面 4 个部分，客户端就有了一条打开的 SSL 连接。这样它就可以查询连接的状态，选择参数，检查服务器证书了。

在这个例子中，客户端和服务器对 DES-CBC3-MD5 批量加密密码进行了沟通。你还能看到服务器站点证书属于美国犹他州盐湖城的摩根士丹利组织。证书由 RSA 数据安全组织授予，主机名为 clients1.online.msdw.com，与请求相符。 334

只要建立起了 SSL 信道，并且客户端对站点的证书没有异议，就可以通过安全信道来发送其 HTTP 请求了。在我们这个例子中，客户端发送了一条简单的“GET / HTTP/1.0”HTTP 请求，并收到了 302 Redirect 响应，请求用户去获取另一个 URL。

14.9 通过代理以隧道形式传输安全流量

客户端通常会用 Web 代理服务器（参见第 6 章）代表它们来访问 Web 服务器。比如，很多公司都会在公司网络和公共因特网的安全边界上放置一个代理（参见图 14-19）。代理是防火墙路由器唯一允许进行 HTTP 流量交换的设备，它可能会进行病毒检测或其他的内容控制工作。

但只要客户端开始用服务器的公开密钥对发往服务器的数据进行加密，代理就再也不能读取 HTTP 首部了！代理不能读取 HTTP 首部，就无法知道应该将请求转向何处了（参见图 14-20）。

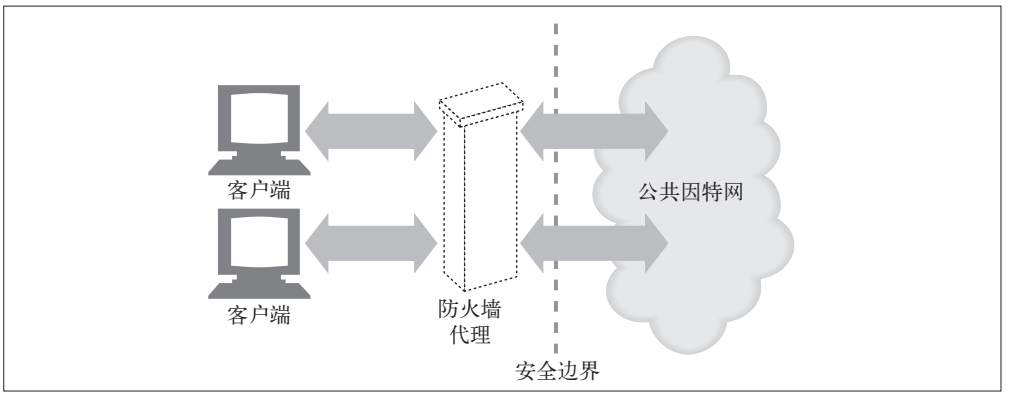


图 14-19　公司防火墙代理

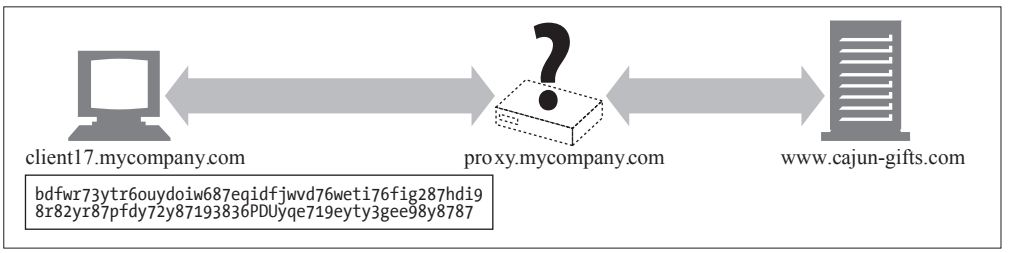


图 14-20　代理无法转发加密请求

为了使 HTTPS 与代理配合工作，要进行几处修改以告知代理连接到何处。一种常
335 用的技术就是 HTTPS SSL 隧道协议。使用 HTTPS 隧道协议，客户端首先要告知代理，它想要连接的安全主机和端口。这是在开始加密之前，以明文形式告知的，所以代理可以理解这条信息。

HTTP 通过新的名为 CONNECT 的扩展方法来发送明文形式的端点信息。CONNECT 方法会告诉代理，打开一条到所期望主机和端口号的连接。这项工作完成之后，直接在客户端和服务器之间以隧道形式传输数据。CONNECT 方法就是一条单行的文本命令，它提供了由冒号分隔的安全原始服务器的主机名和端口号。host:port 后面跟着一个空格和 HTTP 版本字符串，再后面是 CRLF。接下来是零个或多个 HTTP 请求首部行，后面跟着一个空行。空行之后，如果建立连接的握手过程成功完成，就可以开始传输 SSL 数据了。下面是一个例子：

```
CONNECT home.netscape.com:443 HTTP/1.0
User-agent: Mozilla/1.1N

<raw SSL-encrypted data would follow here...>
```

在请求中的空行之后，客户端会等待来自代理的响应。代理会对请求进行评估，确保它是有效的，而且用户有权请求这样一条连接。如果一切正常，代理会建立一条

到目标服务器的连接。如果成功，就向客户端发送一条 200 Connection Established 响应。

```
HTTP/1.0 200 Connection established
Proxy-agent: Netscape-Proxy/1.1
```

更多有关安全隧道和安全代理的信息，请回顾 8.5 节。

14.10 更多信息

安全和密码问题是非常重要，也非常复杂的问题。如果想学习更多有关 HTTP 安全性、数字加密技术、数字证书以及公开密钥基础设施方面的内容，可以从下面这几个地方开始。

14.10.1 HTTP安全性

- *Web security, Privacy & Commerce*[13]（《Web 安全与电子商务》）
  Simson Garfinkel 著，O'Reilly & Associates 公司。这是 Web 安全以及 SSL/TLS 和数字证书方面最好、最可读的入门型书籍之一。

- http://www.ietf.org/rfc/rfc2818.txt
  RFC 2818，“HTTP Over TLS”（“TLS 上的 HTTP”），说明了如何在 SSL 的后继协议——TLS 协议之上实现安全 HTTP。 336

- http://www.ietf.org/rfc/rfc2817.txt
  RFC 2817，“Upgrading to TLS Within HTTP/1.1”（“在 HTTP/1.1 中升级到 TLS”），说明了如何使用 HTTP/1.1 中的升级机制在现存的 TCP 连接上启动 TLS。这样非安全和安全 HTTP 流量就可以共享相同的知名端口了（在这种情况下，使用的是 http: 的 80 端口，而不是 https: 的 443 端口）。还可以使用虚拟主机技术。这样，使用一台 HTTP+TLS 服务器就可以区分出发往同一个 IP 地址上不同主机名的流量了。

14.10.2 SSL与TLS

- http://www.ietf.org/rfc/rfc2246.txt
  RFC 2246，“The TLS Protocol Version 1.0”（“TLS 协议版本 1.0”），对（SSL 的后继协议）TLS 协议的版本 1.0 进行了规范。TLS 提供了因特网上通信的私密性。协议允许客户端 / 服务器应用程序以防止窃听、篡改以及伪造报文的方式进行通信。

- http://developer.netscape.com/docs/manuals/security/sslin/contents.htm
  “Introduction to SSL”（“SSL 简介”）介绍了 SSL 协议。SSL 最初是由网景公司

注 13：本书中文版由中国电力出版社出版。（编者注）

开发的，已广泛应用于万维网上客户端和服务器间的认证及加密通信。

- http://www.netscape.com/eng/ssl3/draft302.txt
  “The SSL Protocol Version 3.0”（“SSL 协议版本 3.0”）是网景公司 1996 年的 SSL 规范。
- http://developer.netscape.com/tech/security/ssl/howitworks.html
  “How SSL Works”（“SSL 是如何工作的”）是网景公司对密钥加密技术的介绍。
- http://www.openssl.org
  OpenSSL 项目是一个合作开发项目，目的是开发一个强壮的、全功能的、商业级开源工具集，以实现安全套接字层（SSL v2/v3）和传输层安全（TLS v1）协议以及强大的通用密码库。这个项目由全世界范围内的志愿者社区管理，那些志愿者通过因特网进行交流、制定计划、开发 OpenSSL 工具集并撰写相关文档。OpenSSL 基于 Eric A. Young 和 Tim J. Hudson 开发的优秀 SSLeay 库。OpenSSL 工具集有一个 Apache 风格的许可证，这基本上就意味着只要遵循一些基本的许可条件，就可免费获得并将其用于商业或非商业目的。

14.10.3 公开密钥基础设施

- http://www.ietf.org/html.charters/pkix-charter.html
  IETF PKIX 工作组组建于 1995 年，目的是开发一些因特网标准，支持基于
337 X.509 的公开密钥基础设施。这是对此小组活动很好的总结。
- http://www.ietf.org/rfc/rfc2459.txt
  RFC 2459，“Internet X.509 Public Key Infrastructure Certificate and CRL Profile”（“因特网 X.509 公开密钥基础设施证书及 CRL 概述”），详细介绍了 X.509 v3 数字证书。

14.10.4 数字密码

- *Applied Cryptography*[14]（《应用密码学》）
  Bruce Schneier 著，John Wiley & Sons 公司出版。这是为实现者编写的经典密码学书籍。
- *The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography*[15]（《密码故事——人类智力的另类较量》）
  Simon Singh 著，Anchor Books 公司出版。这是一本有趣的密码学入门书籍。它
338 不是为技术专家编写的，而是一本生动的密码学历史读物。

注 14：本书中文版由机械工业出版社出版。（编者注）
注 15：本书中文版由海南出版社出版。（编者注）

第六部分

附　录

本书附录集中包含了一些有用的参考表格、背景信息以及关于 HTTP 结构和实现各种主题的指南。

- 附录 A　URI 方案
- 附录 B　HTTP 状态码
- 附录 C　HTTP 首部参考
- 附录 D　MIME 类型
- 附录 E　Base-64 编码
- 附录 F　摘要认证
- 附录 G　语言标记
- 附录 H　MIME 字符集注册表

496
~
498

附录A

URI方案

已定义的 URI 方案有很多，但常用的并不多。一般来说，有相关的 RFC 对其解释说明的 URI 方案更常用一些，但确实也有少数由主导软件公司（特别是 Netscape 和 Microsoft）开发，但未被正式发布的方案得到了广泛应用。

W3C 维护了一个 URI 方案列表，可以通过以下地址访问：

http://www.w3.org/Addressing/schemes.html

IANA 也维护了一个 URL 方案的列表，网址是：

http://www.iana.org/assignments/uri-schemes

表 A-1 介绍了部分已经提出的和正在使用的方案。注意，表中大约有 90 个方案，其中很多都没有得到广泛应用，而且有些已被废弃。

表A-1　在W3C注册的URI方案

方　案	描　述	RFC
about	研究浏览器各方面特性的 Netscape 方案。比如说，使用 about 自身的效果就跟选择 Navigator 的 Help 菜单中的 About Communicator 一样，about:cache 显示的就是磁盘缓存的统计数据，about:plugins 显示的是与已配置的插件有关的信息。其他浏览器，比如微软的 Internet Explorer，也使用了这个方案	
acap	应用程序配置访问协议	2244
afp	用于使用 AFP（Apple Filing Protocol，苹果文件协议）提供的文件共享服务，是作为已过期的 IETF draft-ietf-svrloc-afp-service-01.txt 的一部分定义的	
afs	保留，以备 Andrew 文件系统将来使用	
callto	初始化微软的 NetMeeting 会议的会话，比如：callto:ws3.joes-hardware.com/joe@joes-hardware.com	
chttp	Real 网络公司定义的 CHTTP 缓存协议。RealPlayer 没有缓存 HTTP 传输的所有条目。作为一种替代方式，可以在文件的 URL 中使用 chttp:// 代替 http:// 来说明要缓存的文件。RealPlayer 在 SMIL 文件中读到 CHTTP URL 时，首先会查看磁盘的缓存中是否有该文件。如果没有此文件，就通过 HTTP 请求此文件，并将其存储在自己的缓存中	
cid	在电子邮件中通过 [MIME] 传送 Web 页面及其相关图片时，需要一个 URL 方案允许 HTML 引用报文中所含图片或其他数据 Content-ID URL，即 cid，就用于这个目的	2392 2111
clsid	允许引用微软的 OLE/COM 类。用于向 Web 页面中插入活动对象	

（续）

方　案	描　述	RFC
data	允许将一些小的常量数据条目作为“即时”数据包含在内。这个 URL 会将 text/plain 字符串 A brief note 编码为：data:A%20brief%20note	2397
date	支持日期的方案建议，比如 date:1999-03-04T20:42:08	
dav	为了确保基于此规范的互操作的正确性，IANA 必须保留以 DAV: 和 opaquelocktoken: 开头的 URI 名字空间，供这个规范和它的修订版本以及相关的 WebDAV 规范使用	2518
dns	供 REBOL 软件使用。 参见 http://www.rebol.com/users/valurl.html	
eid	外部 ID 方案提供了一种机制，本地应用程序可以通过这种机制引用通过其他非 URL 方案获取的数据。这个方案试图提供一种通用的转义机制，以便那些无法提出自己方案的专业应用程序访问信息。这个 URI 的使用方式存在争议。参见 http://www.ics.uci.edu/pub/ietf/uri/draft-finseth-url-00.txt	
fax	方案 fax 描述了一条连接，此连接连至可处理电传的终端（传真机）	2806
file	在特定主机上标识出可访问的文件。其中可以包含主机名，但这个方案的特殊性在于它没有为此类文件指定因特网协议或访问方式；这样，它在主机间网络协议中的效用就会受到限制	1738
finger	finger URL 的格式如下：*finger://host*[:*port*][/<*request*>]。 <request> 必须与 RFC 1288 的请求格式一致。参见 http://www.ics.uci.edu/pub/ietf/uri/draft-ietf-uri-url-finger-03.txt	
freenet	获取中迅互联分布式信息系统中信息所用的 URI。参见 http://freenet.sourceforge.net	
ftp	文件传输协议方案	1738
gopher	古老的 gopher 协议	1738
gsm-sms	用于 GSM 移动电话短信业务的 URI。	
h323，h324	多媒体会议的 URI 方案。 参见 http://www.ics.uci.edu/pub/ietf/uri/draft-cordell-sg16-conv-url-00.txt	
hdl	Handle 是个被广泛应用的系统，用于分配、管理数字对象和因特网上的其他资源，并将其解析成名为 handles 的永久标识符。可以将 handle 作为 URN 使用。参见 http://www.handle.net	
hnews	HNEWS 是 NNTP 新闻协议的一个 HTTP 隧道变体。hnews URL 语法设计与当前常用的新闻 URL 方案兼容。参见 http://www.ics.uci.edu/pub/ietf/uri/draft-stockwell-hnews-url-00.txt	
http	HTTP 协议。更多信息请参见本书	2616

（续）

方　案	描　述	RFC
https	SSL 上的 HTTP。 参见 http://sitesearch.netscape.com/eng/ssl3/draft302.txt	
iioploc	CORBA 扩展。可互操作的名字服务定义了一种 URL 格式的对象引用——iioploc，可以将其输入一个程序中，以获取包括名字服务在内的已定义远程服务。比如，以下 iioploc 标识符： iioploc://www.omg.org/NameService 就表示运行在 IP 地址与域名 www.omg.org 相对应的机器上的 CORBA 名字服务。参见 http://www.omg.org	
ilu	ILU（Inter-Language Unification，跨语言统一）系统是一个多语言对象接口系统。ILU 提供的对象接口隐藏了不同语言、不同地址空间和不同操作系统之间的实现差异。可以通过 ILU，用经过良好说明的语言无关接口来构建多语言的面向对象库（类库）。还可以将其用于实现分布式系统。 参见 ftp://parcftp.parc.xerox.com/pub/ilu/ilu.html	
imap	IMAP URL 方案用于分配 IMAP 服务器、邮箱、报文、MIME 主体 [MIME]，在因特网主机上搜索可以通过 IMAP 协议访问的程序	2192
IOR	CORBA 的互操作对象引用。 请参见 http://www.omg.org	
irc	irc URL 方案用于表示 IRC（Internet Relay Chat，因特网中继聊天）服务器或者 IRC 服务器上的独立实体（信道或人）。 参见 http://www.w3.org/Addressing/draft-mirashi-url-irc-01.txt	
isbn	建议用于 ISBN 书籍参考的方案。 参见 http://lists.w3.org/Archives/Public/www-talk/1991NovDec/0008.html	
java	用来识别 Java 类	
javascript	网景的浏览器会处理 javascript URL，如果冒号（：）后面有表达式，则估算它的值，只要表达式字符串的值不是未定义的，就会加载包含了这个值的页面	
jdbc	用于 Java SQL API	
ldap	允许因特网客户端直接访问 LDAP 协议	2255
lid	本地标识符方案。 参见 draft-blackketter-lid-00	
lifn	UTK 开发的批量文件分发分布式存储系统所使用的 lifn（location-independent file name，位置无关文件名）	
livescript	JavaScript 的曾用名	

（续）

方　案	描　述	RFC
lrq	参见 h323	
mailto	URL 方案 mailto 用于访问单个用户或服务的因特网邮件地址	2368
mailserver	1994 ～ 1995 年的老建议，支持将整条报文都编码到一个 URL 中去，这样（比如说）URL 可以自动向邮件服务器发送订阅邮件列表的电子邮件了	
md5	MD5 是一种密码校验和	
mid	mid 方案用电子邮件报文的 message-id（一部分）来引用一个特定的报文	2392 2111
mocha	参见 javascript	
modem	modem 方案描述了一条连接，连接到能够处理输入数据呼叫的终端上去	2806
mms、mmst、mmsu	MMS（Microsoft Media Server，微软媒体服务器）以流方式传送 ASF（Active Streaming Format，活动流格式）文件时使用的方案。强制使用 UDP 传输时，使用 mmsu 方案。强制使用 TCP 传输时，使用 mmst 方案	
news	news URL 方案指的是 USENET 新闻中的新闻组，或独立的文章。news URL 使用下列两种格式之一： *news:<newsgroup-name>* 或 *news:<message-id>*	1738 1036
nfs	指的是 NFS 服务器上的文件和目录	2224
nntp	另一种引用 news 文章的方法，指定 NNTP 服务器上的 news 文章时很有用。nntp URL 看起来如下所示： nntp://*<host>*:*<port>*/*<newsgroup-name>*/*<article-num>* 注意，尽管 nntp URL 为文章资源指定了唯一的位置信息，但现在因特网上大部分 NTTP 服务器都配置为只允许从本地客户端访问，nntp URL 因此就无法指定全球可访问的资源了。因此，URL 的 news 格式更多地是作为识别新闻性文章的一种方式来使用	1738 977
opaquelocktoken	以 URI 形式表示的 WebDAV 锁定令牌，用于标识特定锁的。每个成功的 LOCK 操作都会在响应主体的 lockdiscovery 特性中返回锁定令牌，也可以通过资源的锁发现操作找到它。参见 RFC 2518	
path	path 方案定义了一个统一的层次化命名空间。在这个空间中，path URN 就是由一些组件和可选的不透明字符串组成的序列。 参见 http://www.hypernews.org/~liberte/www/path.html	
phone	在“电话的 URL”中使用。在 RFC 2806 中被 tel: 取代	
pop	POP URL 指定了一个 POP 电子邮件服务器，一个可选的端口号、认证机制、认证 ID 和 / 或授权 ID	2384

（续）

方　案	描　述	RFC
pnm	Real 网络公司的流媒体协议	
pop3	POP3 URL 方案允许 URL 指定一个 POP3 服务器，允许其他协议使用通用的“用于邮件访问的 URL”取代对 POP3 的显式引用。在已过期的 draft-earhart-url-pop3-00.txt 中定义	
printer	用于服务定位标准的抽象 URL。 参见 draft-ietf-srvloc-printer-scheme-02.txt	
prospero	通过 prospero 目录服务访问的名字资源	1738
res	微软的方案，指定了一个要从某模块中获取的资源。包含一个字符串或数字资源类型和一个字符串或数字 ID	
rtsp	实时流协议，是 Real 网络公司现代流媒体控制协议的基础协议	2326
rvp	RVP 集合点协议的 URL，用于在某计算机网络上发布用户到来的通知。参见 draft-calsyn-rvp-01	
rwhois	RWhois 是在 RFC 1714 和 RFC 2167 中定义的因特网目录访问协议。RWhois URL 将 rwhois 的直接访问权赋予了客户端。 参见 http://www.rwhois.net/rwhois/docs/	
rx	一种结构，允许远程图像应用程序在 Web 页面中显示数据。 参见 http://www.w3.org/People/danield/papers/mobgui/	
sdp	sdp（session description protocol，会话描述协议）URL。参见 RFC 2327	
service	service 方案可以为任意网络服务提供访问信息。这些 URL 为基于客户端的网络软件提供了一种可扩展的框架，以获取使用网络服务所需的配置信息	2609
sip	sip* 族方案用于建立使用 sip（session initiation protocol，会话发起协议）的多媒体会议	2543
shttp	S-HTTP 是 HTTP 的超集，用于保护 HTTP 连接，它提供了大量机制用以实现保密性、认证功能和完整性。S-HTTP 没有被广泛采用，主要是被 HTTPS（经过 SSL 加密的 HTTP）取代了。 参见 http://www.homeport.org/~adam/shttp.html	
snews	经 SSL 加密的 news	
STANF	用于可靠网络文件名的老建议。与 URN 有关。 参见 http://Web3.w3.org/Addressing/#STANF	
t120	参见 h323	
tel	通过电话网打电话的 URL	2806
telephone	用于 tel 的早期草案中	

（续）

方　案	描　述	RFC
telnet	指定了可能会被 Telnet 协议访问的交互式业务。Telent URL 的格式如下所示： *telnet://<user>:<password>@<host>:<port>/*	1738
tip	支持 TIP 原子化的因特网事务处理	2371 2372
tn3270	根据 ftp://ftp.isi.edu/in-notes/iana/assignments/url-schemes 保留	
tv	TV URL 命名了一个特定的电视广播信道	2838
uuid	UUID（通用唯一标识符）不包含与位置有关的信息。也称为 GUID（全球唯一标识符）。由一个 128 位的唯一 ID 组成。它和 URN 一样，不会随时间发生变化。要使用无法或不应该依赖于特定的物理根名字空间（比如一个 DNS 名称）的通用标识符时，UUID URI 是很有用的。 参见 draft-kindel-uuid-uri-00.txt	
urn	持久的、与位置无关的 URN	2141
vemmi	允许 VEMMI（万用多媒体接口）客户端软件和 VEMMI 终端连接 VEMMI 兼容的服务。VEMMI 是一种在线多媒体应用服务的国际标准	2122
videotex	允许 videotex 客户端软件或终端连接与 ITU-T 和 ETSI videotex 标准兼容的 videotex 服务。 参见 http://www.ics.uci.edu/pub/ietf/uri/draft-mavrakis-videotex-url-spec-01.txt	
view-source	网景的 Navigator 的源码查看器。这些 view-source URL 可以显示用 JavaScript 生成的 HTML	
wais	广域信息服务——一种早期搜索引擎形式	1738
whois++	WHOIS++ 简单因特网目录协议的 URL。 参见 http://martinh.net/wip/whois-url.txt	1835
whodp	WhoDP（Widely Hosted Object Data Protocol，广泛托管对象数据协议）用于沟通大量动态、可重定位对象的当前位置和状态传递。WhoDP 程序通过“订阅”定位对象，接收与某对象有关的信息，并“发布”这些信息，控制此对象的位置和可见状态	
z39.50r，z39.50s	Z39.50 会话与检索 URL。Z39.50 是一种信息检索协议，不能很好地适用于主要为获取无状态数据而设计的检索模式。它将通用的用户查询设计为面向会话的多步任务，服务器在继续处理任务之前，会向客户端请求额外的参数，因此每一步任务都可能被临时挂起	2056

503

504

附录B

HTTP状态码

附录 B 是 HTTP 状态码及其含义的快速参考。

B.1 状态码分类

HTTP 状态码分为 5 类，如表 B-1 所示。

表B-1 状态码分类

总体范围	已定义范围	类 别
100~199	100~101	信息
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

B.2 状态码

表 B-2 是 HTTP/1.1 规范定义的所有状态码的快速参考，表中概述了每种状态码及其含义。3.4 节曾详细地介绍了这些状态码及其用法。

表B-2 状态码

状态码	原因短语	含 义
100	Continue（继续）	收到了请求的起始部分，客户端应该继续请求
101	Switching Protocols（切换协议）	服务器正根据客户端的指示将协议切换成 Update 首部列出的协议
200	OK	服务器已成功处理请求
201	Created（已创建）	对那些要服务器创建对象的请求来说，资源已创建完毕
202	Accepted（已接受）	请求已接受，但服务器尚未处理
203	Non-Authoritative Information（非权威信息）	服务器已将事务成功处理，只是实体首部包含的信息不是来自原始服务器，而是来自资源的副本
204	No Content（没有内容）	响应报文包含一些首部和一个状态行，但不包含实体的主体内容
205	Reset Content（重置内容）	另一个主要用于浏览器的代码。意思是浏览器应该重置当前页面上所有的 HTML 表单
206	Partial Content（部分内容）	部分请求成功

（续）

状态码	原因短语	含　义
300	Multiple Choices（多项选择）	客户端请求了实际指向多个资源的 URL。这个代码是和一个选项列表一起返回的，然后用户就可以选择他希望使用的选项了
301	Moved Permanently（永久移除）	请求的 URL 已移走。响应中应该包含一个 Location URL，说明资源现在所处的位置
302	Found（已找到）	与状态码 301 类似，但这里的移除是临时的。客户端应该用 `Location` 首部给出的 URL 对资源进行临时定位
303	See Other（参见其他）	告诉客户端应该用另一个 URL 获取资源。这个新的 URL 位于响应报文的 `Location` 首部
304	Not Modified（未修改）	客户端可以通过它们所包含的请求首部发起条件请求。这个代码说明资源未发生过变化
305	Use Proxy（使用代理）	必须通过代理访问资源，代理的位置是在 `Location` 首部中给出的
306	（未用）	这个状态码当前并未使用
307	Temporary Redirect（临时重定向）	和状态码 301 类似。但客户端应该用 `Location` 首部给出的 URL 对资源进行临时定位
400	Bad request（坏请求）	告诉客户端它发送了一条异常请求
401	Unauthorized（未授权）	与适当的首部一起返回，在客户端获得资源访问权之前，请它进行身份认证
402	Payment Required（要求付款）	当前此状态码并未使用，是为未来使用预留的
403	Forbidden（禁止）	服务器拒绝了请求
404	Not Found（未找到）	服务器无法找到所请求的 URL
405	Method Not Allowed（不允许使用的方法）	请求中有一个所请求的 URI 不支持的方法。响应中应该包含一个 `Allow` 首部，以告知客户端所请求的资源支持使用哪些方法
406	Not Acceptable（无法接受）	客户端可以指定一些参数来说明希望接受哪些类型的实体。服务器没有资源与客户端可接受的 URL 相匹配时可使用此代码
407	Proxy Authentication Required（ 要求进行代理认证）	和状态码 401 类似，但用于需要进行资源认证的代理服务器
408	Request Timeout（请求超时）	如果客户端完成其请求时花费的时间太长，服务器可以回送这个状态码并关闭连接
409	Conflict（冲突）	发出的请求在资源上造成了一些冲突

506

（续）

状态码	原因短语	含　义
410	Gone（消失了）	除了服务器曾持有这些资源之外，与状态码 404 类似
411	Length Required（要求长度指示）	服务器要求在请求报文中包含 Content-Length 首部时会使用这个代码。发起的请求中若没有 Content-Length 首部，服务器是不会接受此资源请求的
412	Precondition Failed（先决条件失败）	如果客户端发起了一个条件请求，如果服务器无法满足其中的某个条件，就返回这个响应码
413	Request Entity Too Large（请求实体太大）	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
414	Request URI Too Long（请求 URI 太长）	客户端发送的请求所携带的请求 URL 超过了服务器能够或者希望处理的长度
415	Unsupported Media Type（不支持的媒体类型）	服务器无法理解或不支持客户端所发送的实体的内容类型
416	Requested Range Not Satisfiable（所请求的范围未得到满足）	请求报文请求的是某范围内的指定资源，但那个范围无效，或者未得到满足
417	Expectation Failed（无法满足期望）	请求的 Expect 首部包含了一个预期内容，但服务器无法满足
500	Internal Server Error（内部服务器错误）	服务器遇到了一个错误，使其无法为请求提供服务
501	Not Implemented（未实现）	服务器无法满足客户端请求的某个功能
502	Bad Gateway（网关故障）	作为代理或网关使用的服务器遇到了来自响应链中上游的无效响应
503	Service Unavailable（未提供此服务）	服务器目前无法为请求提供服务，但过一段时间就可以恢复服务
504	Gateway Timeout（网关超时）	与状态码 408 类似，但是响应来自网关或代理，此网关或代理在等待另一台服务器的响应时出现了超时
505	HTTP Version Not Supported（不支持的 HTTP 版本）	服务器收到的请求是以它不支持或不愿支持的协议版本表示的

507

Web Programming

O'REILLY®

HTTP权威指南

HTTP是Web客户端与服务器交互文档和信息时所使用的协议，是每个成功Web事务的幕后推手。众所周知，我们每天访问公司内部网络、搜索绝版书籍、研究统计信息时所使用的浏览器的核心就是HTTP。但HTTP的应用远不仅仅是浏览Web内容。由于HTTP既简单又普及，很多其他网络应用程序也选择了它，尤其是采用SOAP和XML-RPC这样的Web服务。

本书详细解释了HTTP协议，包括它是如何工作的，如何用它来开发基于Web的应用程序。但本书并不只介绍了HTTP，还探讨了HTTP有效工作所依赖的所有其他核心因特网技术。尽管HTTP是本书的中心内容，但本书的本质是理解Web的工作原理，以及如何将这些知识应用到Web编程和管理之中，主要涵盖HTTP的技术运作方式、产生动机、性能和目标以及一些相关技术问题。

本书是HTTP协议及相关Web技术方面的权威著作，主要内容包括：

- HTTP方法、首部以及状态码
- 优化代理和缓存的方法
- 设计Web机器人和爬虫的策略
- Cookies、认证以及安全HTTP
- 国际化及内容协商
- 重定向及负载平衡策略

本书由具有多年实践经验的专家编写，通过简洁、精确的语言和大量翔实的细节图解帮助读者形象地理解Web幕后所发生的事情，详细说明了Web上每条请求的实际运行情况。

要想高效地进行Web开发，所有Web程序员、管理员和应用程序开发者都应该熟悉HTTP。很多书籍只介绍了Web的使用方式，而本书则深入说明了Web的工作原理。

封面设计：Karen Montgomery　张健

图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

O'REILLY®
oreilly.com.cn

分类建议　计算机/网络技术/HTTP

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

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

ISBN 978-7-115-28148-7

ISBN 978-7-115-28148-7
定价：109.00元