HZ BOOKS
华章科技

no starch press

Web安全领域的圣经级著作，唯一深度探索现代Web浏览器安全技术的专著，由来自Google Chrome浏览器团队的世界顶级黑客、国际一流信息安全专家撰写。

从浏览器设计角度深入剖析现代浏览器的技术原理、安全机制和设计上的安全缺陷，为Web安全工作者应对基于浏览器的各种安全隐患提供指南。

信息安全技术丛书

Web之困

现代Web应用安全指南

The Tangled Web

A Guide to Securing Modern Web Applications

[美] Michal Zalewski 著　　朱筱丹 译　　殷钧钧 审校

机械工业出版社
China Machine Press

信息安全技术丛书

Web之困

现代Web应用安全指南

The Tangled Web: A Guide to Securing Modern Web Applications

（美） Michal Zalewski 著

朱筱丹 译

殷钧钧 审校

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Web 之困：现代 Web 应用安全指南 /（美）扎勒维斯基（Zalewski, M.）著；朱筱丹译 . —北京：机械工业出版社，2013.10
（信息安全技术丛书）
书名原文：The Tangled Web: a Guide to Securing Modern Web Applications

ISBN 978-7-111-43946-2

Ⅰ . W… Ⅱ . ① 扎… ② 朱… Ⅲ . 浏览器 – 安全技术 – 研究 Ⅳ . TP393.092

中国版本图书馆 CIP 数据核字（2013）第 211485 号

本书版权登记号：图字：01-2012-2101

本书在 Web 安全领域有“圣经”的美誉，在世界范围内被安全工作者和 Web 从业人员广为称道，由来自 Google Chrome 浏览器团队的世界顶级黑客、国际一流安全专家撰写，是目前唯一深度探索现代 Web 浏览器安全技术的专著。本书从浏览器设计的角度切入，以探讨浏览器的各主要特性和由此衍生出来的各种安全相关问题为主线，深入剖析了现代 Web 浏览器的技术原理、安全机制和设计上的安全缺陷，为 Web 安全工作者和开发工程师们应对各种基于浏览器的安全隐患提供了应对措施。

本书开篇回顾了 Web 的发展历程和安全风险的演化；第一部分解剖了现代浏览器的工作原理，包括 URL、HTTP 协议、HTML 语言、CSS、文档格式、浏览器插件等内容；第二部分从浏览器的设计角度深入分析了各种现代 Web 浏览器（Firefox、Chrome、IE 等）所引入的重点安全机制，例如同源策略、源的继承、窗口和框架的交互、安全边界、内容识别、应对恶意脚本、外围的网站特权等，并分析了这些机制存在的安全缺陷，同时为 Web 应用开发者提供了如何避免攻击和隐私泄露的应对措施；第三部分对浏览器安全机制的未来趋势进行了展望，包括新的浏览器特性与安全展望、其他值得注意的浏览器、常见的 Web 安全漏洞等。

机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码 100037）
责任编辑：吴 怡
印刷
2013 年 10 月第 1 版第 1 次印刷
186mm × 240mm • 17.5 印张
标准书号：ISBN 978-7-111-43946-2
定 价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzjsj@hzbook.com

译 者 序

这是一本很特别的书。

在翻开本书之前，每位读者可能都曾阅读过一些 Web 安全相关的书籍。但本书的目标和写法，与常规的 Web 安全书籍大相径庭。套句江湖行话来说，这是一本修炼内功的秘籍，它并未传授什么具体的武功，不会手把手地教读者怎么练一门剑术或轻功，但它构建了一个完整的 Web 安全图景，为读者在这个庞杂领域里继续深入钻研打下稳固的技术根基，相信每位打开这本“秘籍”的读者都能从中获益良多。

所以如果读者打算找的是一本常规的“黑客手册”，那它可能不适合你。本书作者浸淫安全领域多年，无疑是圈内顶尖高手，甚至自己也开发了网站漏洞扫描程序 Skipfish。但在这本厚积薄发的技术书里，他却并没有告诉读者，要怎么去黑掉一个网站或有什么趁手工具可用，所以曾有国外读者玩笑似地抱怨本书写得不够“邪恶”。的确，全书的前三分之一，作者都在讨论貌似枯燥的各种 RFC 协议和规范的来龙去脉，解决什么问题，具体机制是什么，都潜藏了哪些漏洞或缺陷，以及这些问题的历史根源。这一部分涵盖了许多我们熟悉的 Web 相关内容，比如 URL、HTTP、HTML、JavaScript、CSS 和插件等最基本的 Web 组件。但作者探索程度之深之广，令人叹服（难怪 Joey 说这是 Web 安全的圣经）！在跟随作者展开这场 Web 安全之旅时，即使最资深的 Web 开发工程师和安全渗透工程师只怕也会觉得乱花渐欲迷人眼，而这些问题所带来的后果更时常令人弹眼落睛，怵目惊心——很多问题和陷阱，我们自己又何尝没吃过亏呢。

本书第二部分为全书核心，主题为浏览器安全机制，是作者更侧重的研究领域。作者在前言里开宗明义，提到本书源自他所维护的一个技术性维基站点“Google’s Browser Security Handbook”项目。作为走在浏览器安全领域技术潮流前端的人物，作者关注的问题不但全面而且深入。这一部分的重头戏是同源策略，围绕经典的同源策略应用，派生出不同场景里的微妙差异、各种继承关系以及跨域的解决方案，另外还涵盖了同源策略无法企及的一些犄角旮旯的方面。此外，作者还谈到了被广泛忽略的浏览器内容自动识别问题，以及一些外围的特权问题和恶意脚本的处理。对浏览器端的安全机制这一主题，可能是第一次有人进行如此全面深入的探讨。也许国内的安全工程师和开发人员往往更侧重服务器端的安全，但在当前的技术浪潮下，服务器端与浏览器端的界限已经越来越模糊，两者越来越紧密地相连，浏览器安全理应获得更多的关注，相信在全新的讨论领域里，读者

会受到许多启发。

作者在最后展望了一批属于未来的 Web 技术，以及它们对今后几年 Web 安全可能产生的影响，有助于读者了解前瞻性的技术动向。当然，由于本书写于两年前，可能其中有部分机制已经被业界采纳成为正式的规范。

在许多章的最后，作者都给出了“安全工程速查表”，内容是和本章主题相关的安全建议。这些整理得当的安全列表对网站和 Web 组件的开发者、架构设计师们想必都会有非常直接的帮助。但同时，这个速查表和本书的其他内容，也完全可以作为安全渗透人员的反向指标，为安全检测带来更多启发性的思路。尽管本书通常被归类为“黑客”书籍，但它也同样能为 Web 开发人员和 Web 防御工程人员带来实际的帮助。攻与防，在这里有机地结合到了一起。

本书另一个特别的地方是：它非常“扎实”！也就是说，干货很多很实在。原书只有不到 300 页篇幅，却涵盖了 Web 各组件、浏览器全部安全机制和未来技术展望，知识点异常密集！可想而知，作者的写法有多简洁明了，这给翻译带来了不小的困难。并且容我冒昧吐槽一句，由于作者本人非常聪明有才，往往不屑于把问题展开来阐述，所以一个非常复杂的知识点也经常被寥寥数语带过，对没有一定基础的读者来说，难免存在阅读和理解障碍。在此译者再啰嗦地提出几点阅读建议：

- ❑ 读慢些，反复读。本书绝对不是那种读一遍就能完全领会、流畅易懂的书籍，适合它的阅读方式应该是放在案头，时常翻阅，常读常新。
- ❑ 不要忽略文中带上标的注释。有些作者只写了几句话的地方，其实对应着一篇长达几十页的论文或很长的一篇网页文章！如果看不懂原文可试着阅读关联的注释内容或自行在搜索引擎里寻找答案。
- ❑ 作者本人的“Google’s Browser Security Handbook”项目是个获取知识的宝库，涉及很多浏览器端安全相关内容。在阅读第二部分时，可配合阅读和实验。
- ❑ 译者水平有限，不能尽显原书之意，读者阅读时如觉理解困难，建议参考英文原文。

在本书的翻译过程中，得到了各位师友无私的帮助，特此鸣谢！感谢姚莉莉的建议，促使我接下本书的翻译，并在此后的翻译过程中给予了无数的帮助和监督；感谢编辑吴怡一直以来对我的耐心、宽容和信任；感谢 Joey（殷钧钧）对本书的推荐和审读，以及在翻译过程中提供的持续帮助；感谢贾洪峰老师和我的同事黄伟，他们对细节的关注，对技术的理解和对书稿的审阅，都使我获益良多；另外也要感谢 Xiaket（夏恺）、钱文芳对部分章节的审读。

本书是译者的第一本译作，水平有限，错误难免。只有诚挚地希望读者诸君在发现错误时请务必告知，我将建立一个勘误表，作为弥补之计。可使用电子邮件与我联系：danzhu@gmail.com，或新浪微博：medanzhu。

前　言

仅在 15 年前，互联网还是简单而无足轻重的：这套古怪的机制不过是让一群学生，还有一伙不太合群、住在地下室里的科学怪人，能访问彼此的个人主页而已，这些主页的内容可能是科学、他们的宠物或诗歌什么的。但到了今天，互联网已成为创建各种复杂交互应用的平台（这些应用包括从邮件客户端、图片编辑器到电脑游戏），它还是一种遍布全球、无数普通用户都能访问的大众媒体，同时它俨然已是重要的商业手段，以致 1999 ~ 2001 年第一次互联网泡沫破灭时，它正是导致经济倒退的主要原因。

即使以我们所处的信息化时代标准来衡量，互联网从默默无闻到无处不在，其发展速度也算异常惊人——但这种惊人的跃升速度也带来许多难以预计的问题。互联网在设计上的缺陷和实现上的漏洞与它的发展状况完全不相称，但我们并没有机会停下脚步回顾之前的错误。这些缺陷很快就导致今时今日许多严重又普遍的数据安全问题：人们发现，当年那个用在简单花哨个人主页上的互联网机制设计标准，已完全不适用于当下每年处理庞大信用卡交易的在线商店。

如果我们回顾过去 10 年，心里难免会略有失落：几乎每个如今值得说道的在线应用，都因为贪图方便而凑合着用从早期互联网搬过来的技术，导致后期付出了沉重代价。以站点 xssed.com 为例，它仅仅收集了无数 Web 安全问题中很特定的一种，但在 3 年的运营时间里，已累计收集超过 5 万次攻击事件，真见鬼！然而，浏览器开发商还是颇为无动于衷，安全社区也未能就这些广泛存在的问题提出什么有见地的建议。与此相对的是安全专家正孜孜不倦地建造一套复杂而炫目的漏洞分类学，并对这种混乱景象的根本原因既习以为常又隐隐担忧绝望。

导致上述问题的部分原因，是由于这些所谓的专家长久以来对 Web 安全的整个混乱状态视而不见，对 Web 安全缺乏真正的了解。他们很利落地给网站漏洞贴上各种标签，比如“责任混淆”（confused deputy）问题[⊖]的各种体现，或者干脆用一些 30 年前商业期刊上惯常的抓眼球字眼。再说了，他们干嘛要费心去关注网络安全呢？一个关于宠物的无聊

⊖ 责任混淆问题（confused deputy problem）是信息安全领域里一个笼统的概念，指某一大类的设计和实现上的缺陷。这个术语描述的是攻击者通过欺骗程序，使其不当地使用“授权”（访问权限），得以某种意想不到的方式操纵资源——通常这种操纵有利于攻击者，这里姑且不论“有利”的具体含义。这个术语在学术界经常被安全专家提及，但从抽象层面来说，现实世界的所有安全问题几乎都可以用这个术语一言以蔽之，所以这个术语其实没多大意义。

个人主页上被加入了一段莫名其妙的注释代码哪能和传统的针对操作系统的漏洞攻击相提并论呢?

回顾过往，我确信我们中的大多数人都有过打落牙齿和血吞的感觉。不仅因为互联网的重要性已远超当初人们的预期，而且我们为了满足自己的心理舒适感，把一些重要的互联网基础特性置于不顾。结果导致即使设计最精良、经过最全面审核的网站应用，也往往比同样功能的非网站应用产生更多的问题。

我们过去搞砸了，现在到悔恨弥补的时候了。出于这个考虑，本书期望能在亟需解决的标准化问题上取得一点进展，除此以外，这也许还是第一本系统而全面地剖析当下 Web 应用安全问题的书籍。为达到这一目标，本书深入描述了我们日常面对的各种安全挑战的独特性，这里的“我们”包括安全专家、网站开发工程师和用户。

本书的章节安排以探讨浏览器的各主要特性和由此衍生出来的各种安全相关问题为主线。因为比起随便采用某种漏洞分类学来罗列问题（这是许多信息安全书籍通常采用的形式），希望这种方式能提供更丰富的信息和更直观的效果。我还希望，这样的安排能使本书更容易阅读。

为使读者便捷地获取答案，我会在每章的最后尽量附上一份“安全工程速查表”。这些速查表为网站应用设计中各种常见问题提供了一些合理的解决方向。此外，在本书的最后一章罗列了最常见的网站漏洞形式及其实现方式。

鸣谢

本书中的许多内容都源自 Google’s Browser Security Handbook 项目，这是我从 2008 年开始维护整理的一个技术性维基站点，该站点以 Creative Commons 授权模式发布。你可以通过浏览以下网址：http://code.google.com/p/browsersec/ 获取相关的源码。我很幸运，因为这个项目不但获得公司的支持，而且能和一群出色的同事一起工作，使得 Browser Security Handbook 的内容能更有用、更准确。在此，我要特别感谢 Filipe Almeida、Drew Hintz、Mariu Schilder 和 Parisa Tabriz 的鼎力相助。

能站在巨人的肩膀上，对此我深感自豪。因为本书从安全社区成员对浏览器安全的广泛研究上获益良多，特别感谢 Adam Barth、Collin Jackson、Chris Evans、Jesse Ruderman、Billy Rios 和 Eduardo Vela Nava，他们极大地提高了我们对这个领域的理解，为这个领域作出了巨大贡献。

无限感激——各位大牛们，继续牛下去吧！

目　录

第 1 章

Web 应用安全

为了给本书后面的技术讨论提供必要的背景知识，我们首先解释清楚安全领域涵盖哪些方面，以及为什么在这个早已被研究得很透彻的领域里，Web 应用的安全仍然值得引起额外的关注。那么，让我们开始吧？

1.1 信息安全速览

表面上看来，信息安全领域属于计算机科学里很成熟、明确且硕果累累的一个分支，自以为无所不知的专家们通过展现他们那分类清晰、数量庞大的安全漏洞集来标榜这一领域的重要性。至于那些漏洞的责任嘛，就全都归到那些“安全文盲”的程序员们头上好了，而理论家们则会从旁指点，说只要遵从今年最热门的某某安全方法学，早就能把这些问题都防患于未然了云云。安全问题更是带动了一个产业的繁荣，但对用户来讲，从普通计算机用户到庞大的国际公司等，其实并没有带来什么有效的安全保障。

从根本上来说，过去几十年，我们甚至没能构建出一个哪怕原始但至少还算可用的框架来理解和评估现代软件的安全性。除了几篇出色的论文和一些小范围内取得的经验，甚至无法拿出什么有说服力的真实的成功案例。现在的侧重点几乎都放在一些响应性质的、次要的安全方法上（如漏洞管理、恶意软件和攻击的检测、沙盒技术以及其他），要不就

是常常对别人代码里的漏洞指指点点。一个令人不安而又秘而不宣的实情是：如果安全系统是由别人开发的，那我们贡献的价值实际上往往乏善可陈；在现代 Web 这件事情上则更是如此。

让我们来看看以下几种最瞩目的信息安全之道，并尝试分析一下为什么到目前为止，它们也没能走出这一困境。

1.1.1　正统之道的尴尬

也许开发一个安全程序最直接的途径就是从算法上证明该程序的运行是正确的。从直觉来说，这个简单的假设听起来还蛮有道理的——那为什么此路不通呢？

首先让我们讨论一下作为形容词时“安全”这两字的含义。准确来说，它到底是什么意思呢？安全（Security）听起来很直观，但在计算机领域，就愣是没法给它下一个确切的定义。没错，我们可以用一些很眩目但基本没啥意义的方式来描述安全，例如，在业界经常被引用的一个关于安全的定义[⊖]如下，但实际上它也是有问题的：

> 如果系统能按既定的方式完成任务，不做额外的事情，那这套系统就是安全的。

这个定义很简洁，也大致描述了一个抽象的目标，但它几乎没提到要怎么做才能达成这个目标。虽然这句话的主题是关于计算机科学的，但其笼统程度，和维克多·雨果的一句诗倒有异曲同工之妙：

> 爱情乃灵魂之一部分，就恍如仙气弥漫于天堂一般。

也许有人会反对说，这个棘手的定义不应该求诸于商业界，那好，我们只管把这个问题抛给学术界吧，但他们也只会给出一个差不多的答案。例如，下面这个常见的学术界对安全的定义，它出自 20 世纪 60 年代出版的贝尔 – 拉帕杜拉安全模型（Bell-La Padula security model，这套规范是企图规范化安全系统需求的诸多努力之一，这是一套针对国家机器[⊜]的规范 [1]；当然也是最知名的一套规范。）

> 当且仅当系统开始于安全的状态，而且一直不会落入非安全状态，它才是安全的。

当然，像这些定义安全的文字从根本上来说都是正确的，用于论文的基调甚至政府规范都毫无问题。但实际上，对真实世界里的软件工程而言，由于以下三个原因，以这些理论为基础建立的模型几乎是没用的。

⊖ 这句话最早出自 Ivan Arce，一位著名的漏洞专家，时间大约在 2000 年前后；自那以后，这句话就被 Crispin Cowan、Michael Howard、Anton Chuvakin 和其他无数安全专家引用过。

⊜ 这套安全模型最初用于美国空军。——译者注

- **对一个足够复杂的计算机系统来说，没有办法定义什么是正确的行为**。对一个操作系统或者 Web 浏览器来说，没有一个权威机构能定义什么是“应该的方式”或者“安全的状态”。最终用户、系统所有者、数据提供者、业务流程所有者和软件硬件开发商之间的利益是南辕北辙，并且说变就变——如果公司的股东们可以为所欲为，能够不加掩饰地优先考虑自己的利益，就更是如此了。雪上加霜的是，社会学和博弈论的研究表明，把各方利益做简单的叠加运算，实际上并不能产生互赢的结果。这种两难的困局就叫“公地悲剧”[⊖]，在日后的互联网发展里它将一直是争论的焦点。
- **美好的想法无法自动转换成规范的约束条件**。即使通过给出系统应包含哪些具体用例，我们能就系统该有的行为达成完美的、高级别的共识，但这些用例几乎不可能与可允许的输入数据、程序的状态和这些状态的转换一一对应起来，而要做正式的系统分析却需要这种对应关系。很简单，譬如，一个很常识性的概念“我不希望别人越权读到我的电子邮件”，却没有办法很好地翻译成数据模型。也许有些剑走偏锋的方法的确能把这种模糊的需求部分地转换成规范化的表达，但对软件开发过程带来严重的限制，并产生比验证算法（Validated Algorithm）更复杂的许多规则集和模型。并且，这些方法自身的正确性也还需要进一步验证……这样就走进了一个死循环。
- **很难令人信服地分析软件的行为**。在复杂的真实世界的场景里，完全没有办法令人信服地通过对计算机程序的静态分析，证明程序的运行是符合详细设计规范的（当然，在高度受限的环境下或针对一个非常狭义的目标还是有可能办到的）。很多案例在实际环境中是无解的（因为计算的复杂程度），甚至有可能由于停机问题（halting problem）而完全无法确定其状态。[⊜]

对安全的这些早期定义既模糊又没用，但令人抓狂的是，尽管几十年过去了，事实上我们取得的进展却非常有限。2001 年由 Naval Research Laboratory 发布的一份学术报告回顾了软件安全领域的早期工作，结果也不过是给软件安全提供了一个更随意的枚举式的定义——而且这个定义还明确否认其自身并不完善和不完整[2]。

> 如果系统在处理信息时恰当地保护了数据，使其不会产生未授权的泄漏、未授权的篡改以及能抵抗未授权的大规模压力（也叫拒绝服务，Denial of Service），那系统就是安全的。我们说“恰当地”是因为如果不加上这个限制，真实环境里实

⊖ 原文是 The tragedy of the commons，一种涉及个人利益与公共利益对资源分配发生冲突时的社会陷阱，意思是“由最大多人数所共享的事物，通常只会得到最少的照顾”。——译者注

⊜ 1936 年时，阿兰 · 图灵（Alan Turing）认为（表达方式上可能略有差异），不可能发明一种算法，由它去确定另一种算法的结果。当然，某些算法在特定用例时会有确定的结果，但并不通用。

> 际上没法达成这一目标；因为安全从根本上来说只是相对的。

这篇论文也回顾和评估了那些早期的安全定义，并指出 BLP 模型为了保持理论上的纯洁性而做出了无谓的牺牲。

> 经验显示，一方面，作为公理的 Bell-La Padula 模型的限制太多：它禁止了在实际应用中一定会出现的用户操作。另一方面，为了克服某些限制条件，它提供的可信对象机制又变得不够严格……导致的结果是，开发人员不得不为每个系统里受信任过程的合理行为，都要设定一个专门的规范。

最后，不论引入多少这种优雅的彼此竞争的安全模型，它们都期望基于也许注定失败的算法，来理解和评估现实世界里的软件安全性。这让开发人员和安全专家没办法对产品代码的质量进行权威的有预见性的判断。那么，我们还有什么选择呢？

1.1.2　进入风险管理

由于缺乏正式的保障，也没有什么实证可用的方法，而商业社会极度依赖的关键性软件又让人心惊肉跳地存在着大量的安全问题，所以业界开始争先恐后地引入另一个抓人眼球的概念：风险管理。

风险管理的理念在保险界大获成功（金融界里的风险管理好像就要逊色那么一点点了），它只是告诉系统所有者应该学会在考虑性价比的情况下，必须接受漏洞必然存在这一现实。一般来说，可以用以下公式计算出为此风险该付出多少代价：

> 风险 = 出现问题的可能性 × 最大损失

例如，根据这一公式，假设某台不太重要的工作站每年受到的攻击对生产效率的影响少于 1000 美金，那么该机构对修复这类损失就应该谨慎投入，不用太费神，无需为此花上譬如 10 多万美金，来引入额外的安全措施应对意外情况，并制订监控计划以避免这些损失。根据风险管理的精神，主要投资应该花在运行关键性任务的主要设备上，进行隔离、加固和监控，因为这些关键性设备处理着客户所有的付账记录呢。

根据目标安排优先级自是理所当然的。但问题是，风险管理主要是和数字打交道，它对帮助我们理解、覆盖和管理真实世界里的问题并无多大裨益。它反而带来一种错误的危险观念：不到位的措施只要被条理化了就能变得合理了，不足的金钱投入再加上风险管理，就能和充足的安全投入达到相似的安全成效。

想碰运气吗？没门呐。

- ❑ **在互联的系统里，损失是没有上限的，范围也不会只固定在单个设备上**。严格意义

上的风险管理是评估某个资源被攻破时会遭受到多少常规损失和最大损失。遗憾的是，这么做其实忽略了很多重大的安全攻击事件——如针对 TJX ㊀和微软㊁的攻击——它们在最开始时都是针对一些相对不重要和被忽略的入口点。但这些开始不起眼的攻击会迅速升级，绕过任何表面的网络隔离，最后几乎完全攻破关键性运营设备。典型的风险管理只关注数字，而和其他节点比起来，初始的事故点往往权重都比较低。同样，因为被利用的可能性较低，能逐步接入敏感资源的内部攻击途径也往往会被忽视了。然而，这两种被忽视的组合会导致代价高昂的损失。

- **健康系统的贡献实际上弥补不了系统被入侵后的非经济损失**。系统被入侵会导致客户信心的受损，业务的中断，甚至可能惹上官司，以及受到监管部门严厉调查等风险，在这些方面很难有什么可靠的保障。至少在理论上，被入侵的后果有可能拖垮一个公司甚至整个行业，浅层地评估它们会带来什么影响，那几乎都只是瞎猜而已。
- **现有的数据对日后的风险也许完全没意义**。与轻微交通事故里的涉事者完全两样，攻击者可不会挺身而出拱手道出事情的原委，更不会事无巨细地记录下事故的损失。除非入侵的程度严重到令人瞠目（因为攻击者的疏怠或攻击的原意就是要搞破坏），通常入侵行动压根就不会被发现。即使行业自身也会报告这类入侵数据，但并没什么方法证实报道的数据是否完整，也搞不清楚会给现在的业务再带来多少风险。
- **基于统计数据的预测对独立事件来说并不靠谱**。尽管普通人在城市里遭到雷击的可能性应该大于被熊袭击，那也不等于我们就该在帽子上戴根避雷针，然后天天泡在蜜糖水里洗澡吧。从单一事件来说，其实无法确定特定的模块会否受到攻击：安全事件几乎是一定存在的，有无数暴露在外的资源，任何服务都有可能受到攻击——所以在企业内部，不太可能说因为哪个特定的服务受到的攻击量特别多，就可以在统计学上预测出有意义的入侵。

1.1.3　分类学的启发

以上讨论的两种思想学派有一个相通之处：都认为可以把安全分解成若干可计算的目

㊀ 指 TJX 事件。2006 年期间在黑客 Albert Gonzalez 的带领下，多位入侵者攻击了某零售商店未加密的无线网络，其后顺势又渗透到这家百货业巨头的公司网络。他们窃取了约 4600 万份客户的信用卡资料和社保号、家庭住址，后来又多加了 45 万份。这起攻击事件中有 11 人被判刑，其中一位为此自杀。

㊁ 指微软的内部讲义“针对微软内部网络的威胁和保护”（Threats Against and Protection of Microsoft’s Internal Network），描述了 2003 年的一次网络攻击，黑客最开始入侵了一位微软工程师家里的工作机，这台机器用 VPN 一直和公司内网保持长连接状态。经过一系列有条不紊逐渐升级的攻击，黑客访问到了内部源代码库并导致数据泄漏。 至少对普通公众而言，攻击者的身份仍无从知晓。

标，由此就能优雅地得到一套统一的安全体系理论或一套能接受的风险管理模型，进而抽象出一些经过优化的底层行为，然后就能借此设计出完美的应用来。

而某些从业者则鼓吹相反的做法，他们不太考虑理论体系，更偏向自然科学的做法。这些从业者就像信息时代的达尔文，他们通过收集足够多的实验数据进行底层抽象，对这些日益复杂的规律进行观察、重组和记录，以期获得某种安全运算的统一模型。

由美国国土安全局资助的 CWE（Common Weakness Enumeration，常见漏洞列表）项目就反映了这种理念。该项目的目标，用它自己的话来说，就是创建一套统一的“漏洞理论”，“以改进对软件缺陷的研究、建模和分类”；并“提供一套通用对话语言，以便于讨论、挖掘和处理软件安全漏洞的成因。”这套体系里的漏洞分类复杂得令人眼花缭乱，它的某些例子可能类似如下这样：

对信息或数据结构的限制不当。
转换不同层次数据时清理不足。
资源标识符控制不当。
对可执行内容的文件名或其他资源名称过滤不充分。

到今时今日，CWE 的词典列表里收录了 800 多条，其中大多数条目在促进交流方面的效果都和上述引用的例子相差无几，并不容易理解。

另一个受自然主义思想影响，但略有差异的学派支持的是 CVSS（Common Vulnerability Scoring System）项目，这个项目由商业机构资助，目标是用一套只有机器能读懂的基本参数，精确量化已知的安全问题。一个真实的漏洞描述可能如下表示：

AV:LN / AC:L / Au:M / C:C / I:N / A:P / E:F / RL:T / RC:UR /
CDP:MH / TD:H / CR:M / IR:L / AR:M

机构和研究者期望通过这 14 项评估因子，根据不同的用例场景，严谨地对潜在漏洞的重要程度打分，达成一个客观、可验证的数字分值（如“42”），这样就完全无需根据主观因素来判断安全漏洞的严重程度。

没错，我是有点儿嘲笑这些项目，但我也完全无意贬低它们的贡献。CWE、CVSS 及其相关项目都胸怀远大目标，如为大型机构的安全事务处理提供更可管理的维度。然而，它们都未能在软件安全这个问题上，诞生出一套真正雄伟的理论体系。而我怀疑在可见的未来也不可能出现这样的理论体系了。

1.1.4　实际的解决之道

现在所有的迹象都表明安全问题与算法无关。可以理解，业界是心不甘情不愿地接受这一结论的，因为这意味着没法到处去吹嘘我们拥有一劳永逸的解决方案（当然，如果在

商业上能获得成功就更称心了）；然后一旦被逼急了，安全圈内的人们又只能回退到诸多最基础的老路子上去。这些方法和诸多新的业务管理模式并不是很兼容，但到目前为止，这些招数是唯一还算管用的法子。这些老方法包括：

- **从失败（最好是别人的失败）中学习**。系统设计应尽量避免出现已知的缺陷。尽管没有自动的解决方案（甚至可能连优雅的方案也欠奉），但最好能不断修正程序的设计指引，确保开发人员知道什么地方有可能出现问题，为他们提供一些工具，尽量以最简单的方式来避免任务出错。
- **开发一些工具来检查和纠正问题**。一般而言，安全上有欠缺不会产生明显的副作用，除非它们被恶意的入侵者利用：但这种反馈成本实在太昂贵了。为解决这个问题，我们可以创建各种安全质量保障（QA）工具，以验证程序的实现是否正确，执行定期的审核以检测是否有一些无意中出现的错误（或者系统工程上的缺陷）。
- **先做最坏的打算**。尽管我们已经尽了最大的努力来避免出现问题，但历史也一再教育我们，还是有可能会出现严重的安全事故。所以要引入恰当的组件隔离、访问控制、数据冗余、监控和响应流程，使服务的相关人员可以在事件还未从小事故演变成大灾难前做出及时的响应。

无论如何，信息安全人员都需要具备足够的耐心、创造力和深厚的技术积累。

当然，即使这些最简单和常识性的规则——差不多也就是安全工程的基本要求吧——也会经常被包装成一堆由缩写拼就的闪亮词语以抓人眼球（如 CIA：代表“Confidentiality、Integrity、Availability”，意即“保密、完整、可用”），还有各种所谓的“方法学”。一般而言，这些方法学不过是把安全业界里那些叫人无比沮丧的失败涂脂抹粉一番，摇身变为另一个成功故事而已，最后再把一堆号称包治百病的产品或认证卖给那些好骗的冤大头客户。尽管这些产品号称无所不能，但它们实际上还是没法取代技术人员对实际环境的把握和对技术的精通——至少现在办不到。

无论如何，在本书的剩余章节里，我会尽量避免建立或重复前面提到过的任何一种宏伟框架，相反，我会采用不那么理论化的做法。在后文中我们会先回顾现代浏览器的方方面面，讨论如何安全地使用相关工具，Web 在哪些方面会被广泛误解，以及出了问题的时候需要怎样控制连带的损失。

我认为这些才是谈论安全工程时的要点。

1.2　Web 的简明历史

Web 曾遭受过很多次重创，层出不穷的安全问题更是令人瞩目。诚然其中一些问题可以归咎为某个版本的客户端或服务器端的具体实现出了问题，但其中很多是由于浏览器基

本机制里各种复杂多变、随心所欲的设计以及它们的相互作用导致的。

我们的 Web 王国建立在一个摇摇欲坠的基础上，原因何在？可能只是由于当初的短视：毕竟，在早期的纯真年代，谁能预料到当今的网络会变得如此危险，大规模安全攻击还往往涉及强烈的金钱诱因？

尽管这些理由对那些真正古老的机制（如 SMTP 和 DNS）确实适用，但它们却并没有带来那么多麻烦：相对来说，Web 还是比较年轻的，Web 诞生和成形的环境与当下并无多大差别。实际上，这一谜团的真正原因可能与多年来 Web 技术那混乱又特殊的演化过程有莫大关系。

所以，请原谅我在此又要简短地打个岔，对 Web 的发展再追本溯源一下。最早期的 Web 尽管相当平淡但仍然值得仔细研究一番。

1.2.1　史前时期的故事：1945 ～ 1994 年

计算机历史学家们都常以美国科学家 Vannevar Bush[3] 在 1945 年虚构出来的一台名为 Memex 的桌面设备作为 Web 理念最早期的原型。Memex 用于在微缩胶卷上创建和标注跨文档链接，并按照这些链接而跳转切换到所引用的其他微缩胶卷上，使用方式大略类似于我们现在的书签和超链接。Bush 大胆推测这种简单的功能将革命性地改变知识管理和数据挖掘的前景（令人莞尔的是，直到 20 世纪 90 年代初期，偶尔还有人觉得这种想法是愚笨且天真可笑的）。但当时并没有任何实际可用的设计，因此那时候它还仅是个充满未来憧憬的愿景而已，直到晶体管计算机登上了历史舞台的中心，一切才可能成真。

20 世纪 60 年代是下一个历史里程碑。这时候诞生了 IBM 的 GML（Generalized Markup Language，通用标记语言），它用可供机器读取识别的指令作为文档的标识符，以标志每段文本的功用，可以明确地指明“这里是文档的头部”，“这里是几个列表项目”诸如此类。在此后经过 20 多年的发展，GML（一开始只是用在一些 IBM 笨重大型机的文本编辑器里）逐渐演变成 SGML（Standard Generalized Markup Language，标准通用标记语言）。SGML 语言更通用灵活，它把 GML 原来基于冒号和句号的笨拙语法，改成了我们熟悉的尖括号格式的语法。

在 GML 进化到 SGML 的过程中，计算机也越来越强大和用户友好。几位研究人员开始试验 Bush 的跨链接概念，把它运用到计算机的文档存储和检索上，看是否可以基于某些关键字对一大堆文件进行交互索引。富有探索精神的各家公司和大学也推出了各种先驱型项目，如 ENQUIRE、NLS 和 Xanadu，但这些项目中的大多数都失败了，未能产生持续的影响。这些项目的共同问题包括可用性太低，过于复杂，可扩展性差等。

历经 10 年之后，两位研究人员，Tim Berners-Lee 和 Dan Connolly 开始寻找新的跨域引用方案——这个方案必须非常简洁明了。他们先草拟了 HTML（HyperText Markup

Language，超文本标记语言）规范，这是一套继承自 SGML 的精简版语言，特别针对带超链接和简单格式的文档进行了设计。在 HTML 方案之后，他们又进而开发了 HTTP 协议（HyperText Transfer Protocol，超文本传输协议），这是一套利用当时已有的 IP 地址、域名和文件路径等概念，专用于访问 HTML 资源的非常基础的协议。他们研究工作的总成果就是这个诞生于 1991 ~ 1993 之间由 Tim Berners-Lee 开发的 World Wide Web 程序（如图 1-1 所示），这个最原始状态的浏览器可以解析 HTML 文件，还可以把用户提交的数据显示出来，并且只需要点击一下鼠标，就可以在不同页面之间切换浏览。

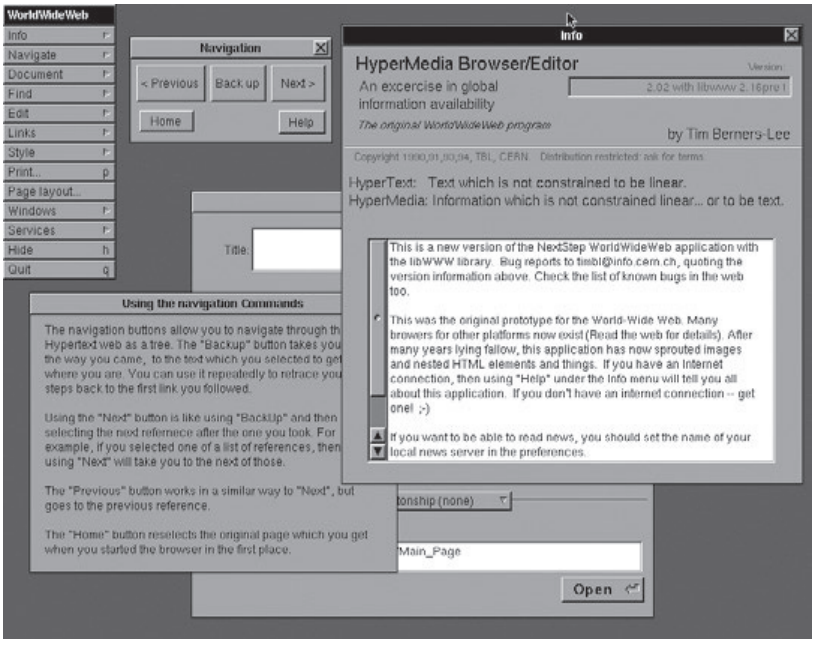


图 1-1　Tim Berners-Lee 的 World Wide Web

与其他心怀高远目标的竞争项目相比，很多人都觉得 HTTP 和 HTML 的设计简直是个巨大的倒退。毕竟，很多早期的构想都鼓吹自己包含数据库整合、安全和数字版权管理，或者整合了内容编辑和出版；即使 Berners-Lee 自己的另一个早期项目 ENQUIRE 看起来格局也更大一些。然而，因为 HTTP 和 HTML 的门槛低，即时可用，扩展性不受限制（正好与那时计算机的运算能力越来越强，价钱更为人接受，互联网也开始普及的时机相吻合），原先毫不起眼的 WWW 项目突然演变成一波热潮。

好吧好吧，这里的“热潮”是以 20 世纪 90 年代的标准来说的。很快，在互联网上就涌现出了 10 多种的 Web 服务器。到了 1993 年 HTTP 已经占美国科学基金会（National Science Foundation）骨干网总流量的 0.1%。在同一年，Mosaic 浏览器登场了，它由美国

伊利诺伊大学（University of Illinois）开发，是第一款广泛使用的、成熟的 Web 浏览器。Mosaic 扩展了原先 World Wide Web 的代码，增加的功能包括：在 HTML 文档里添加内嵌的图像，通过表单提交数据，奠定了今天交互式和多媒体应用的基础。

Mosaic 使网页浏览变得更美观，也令使用者更容易接受 Web 方式。在 20 世纪 90 年代中期，Mosaic 还是另外两个浏览器的基础：Mosaic Netscape（后来改名为 Netscape Navigator）和 Spyglass Mosaic（后来被微软收购并改名为 Internet Explorer）。同时还有好几种非 Mosaic 引擎的同类竞争产品，包括 Opera 和其他几个基于文本的浏览器（例如，Lynx 和 w3m）。很快，互联网上出现了第一个搜索引擎、在线报纸和约会网站。

1.2.2　第一次浏览器大战：1995 ～ 1999 年

到 20 世纪 90 年代中期，很明显 Web 已经站稳了脚跟，用户也愿意为它放弃许多老旧的技术。此时尽管桌面软件巨鳄微软公司先前对互联网的跟进比较慢，现在也逐渐感到了不安，开始投入巨大的资源开发自己的浏览器，最终从 1996 年开始，Windows 操作系统绑定安装了 IE 浏览器[⊖]。微软在这段时间的举动引发了俗称的“浏览器大战”。

浏览器开发商之间的这场军备竞赛主要体现在各竞争产品都在非常快速地开发迭代，以及疯狂加入各种新功能，也就完全无法顾及产品是否符合规范标准，甚至来不及用正儿八经的文档记录下各种新代码新功能。对核心 HTML 特性的擅自调整包括各种蠢事（如闪烁的文字，这是 Netscape 的发明创造，但最终沦为笑柄）乃至一些著名的特性，如可更换字样（Typeface）或可以在所谓的框架（Frame）里嵌入外部文档。在各浏览器厂商的产品里，往往还内置对自家编程语言（如 JavaScript 和 Visual Basic）支持，以及可在用户机器上执行跨平台 Java 或 Flash 小程序的插件，支持有用但颇诡异的各种 HTTP 扩展（如 Cookie）。这一阶段的浏览器尽管囿于某些专利和商标上的原因[⊜]，彼此间会有兼容性问题，但这些不兼容大都还比较表面。

随着 Web 的日益发展壮大和百花齐放，一种隐秘的恶疾悄然在浏览器引擎之间传播开来，尽管表面上还勉强维持着兼容性。这么做最开始的理由听上去还蛮合情合理的：如果浏览器 A 可以正常显示一个有问题的页面，而浏览器 B 却拒绝解析这个页面（无论基于何种原因），用户肯定会认为这是浏览器 B 有问题，而一股脑地选择貌似更强大的浏览器 A。为了确保浏览器可以正确地显示任何网页，工程师的开发变得越来越复杂，也没有什么正式的文档来描述浏览器对于网站管理员胡乱提供的网页，是怎么进行主动猜测解析

⊖ 有趣的是，这个决定非常有争议性。一方面，可以说微软极大地推动了互联网的普及，另一方面，它也损害了其他竞争浏览器的利益，可算是一种反竞争的行为。最后，这个决策导致了一系列关于微软是否滥用自己垄断地位的旷日持久的争执，如“美国诉微软公司案”。

⊜ 例如，微软为了不想因为 JavaScript 商标的授权和 Sun 公司打交道，就直接把自家差不多完全一样但略有不同的脚本语言命名为“JScript”。微软的官方文档上直至今日依然用该名称指代这种软件。

的[⊖]，而在这些处理过程中往往会牺牲掉安全性，偶尔也会累及兼容性。遗憾的是，这样的变动往往又会进一步纵容各种不靠谱的网页设计观念[⊜]，迫使其他浏览器开发商为免掉队，也只能亦步亦趋地跟进。当然，相关规范标准的细节缺失，更新也不及时，更是助长了这种恶疾的蔓延。

到 1994 年，为了解决开发上日益混乱的场面和管理 HTML 的升级扩展，Tim Berners-Lee 和一群资助的公司创建了 W3C 理事会（World Wide Web Consortium）。遗憾的是，在很长一段时间内这个组织也只能眼睁睁地看着 HTML 标准被胡乱扩展和修改。最开始的时候 W3C 只是想制定一个符合当时实际状况的 HTML 2.0 和 HTML 3.2 标准，但最后这些规范都只是半成品，因为等到公开发布之日，它们其实早就过时了。W3C 也尝试过一些经过深思熟虑的创新项目，如层叠样式表（Cascading Style Sheet，CSS），但要浏览器开发商们接纳还需要一点时日。

另外像欧洲计算机制造者协会（European Computer Manufacturers Association，ECMA）、国际标准组织（International Organization for Standardization ，ISO）和互联网工程任务组（Internet Engineering Task Force ，IETF）等一些组织都企图对一些已实现的技术如 HTTP 和 JavaScript 做一些标准化和改进的工作。但可惜，各方的努力很少会相互通气协调，一些讨论和设计决策也往往由大公司和股东们控制，这些人根本不关心技术的长期前景。这样只会产生一些僵硬的标准，互相矛盾的建议，以及协议之间需要交互时各种有害的吓人的案例，实际上这些规范本可以设计得更好些——这个问题在第 9 章讨论到各种内容隔离机制时尤为明显。

1.2.3　平淡期：2000 ～ 2003 年

围绕 Web 的争论仍然持续不断，由于操作系统绑定策略微软浏览器得以一家独大。十年之后，Netscape Navigator 退出市场，Internet Explorer 获得 80% 的市场占有率——这差不多也是 5 年前 Netscape 浏览器的占有率。在这场新功能的攀比大赛中，牺牲的主要就是安全性和可交互性，既然现在战争已结束，尘埃落定后开发者们想来应该可以搁置彼此的异见，坐下来对过往的混乱局面拨乱反正一下吧。

然而，垄断也滋生了自满：在得逞之后，微软就完全缺乏动力去改进自己的浏览器了。在 IE5 之前，微软每年发布一个新版本，然后足足过两年才推出 IE6，其后更是用了漫长的 5 年才从 IE6 升级到 IE7。既然微软不感兴趣，其他的浏览器厂商势单力孤，也很难带来什么翻天覆地的变化；而大部分网站也不愿为了极少数的访问者而修改不符合规范

⊖　详见第 13 章内容。——译者注

⊜　在浏览器的各种功能里，最容易带来误导和产生严重问题的例子，就是对内容和编码字符集的自动检测机制了，我们将在第 13 章讨论这一问题。

的网页。

而另一方面，缓慢的开发进展使 W3C 得以追上浏览器的实际状况，并认真探索未来 Web 的一些新概念。在 2000 年的时候出现的一些新变革包括 HTML4（这是经过整理的 HTML 语言，废弃或禁用了早期 HTML 版本里的一些累赘功能或策略性错误）和 XHTML 1.1（这是一种格式很严格的结构化 XML 文档，不会产生模棱两可的解析，也不会需要浏览器主动猜测这类文件的属性问题）。W3C 理事会更是对 JavaScript 里的 DOM（Document Object Model，文档对象模型）和 CSS 做了重大改良。但遗憾的是，到 20 世纪结束的时候，由于 Web 已经步入成熟，导致早期造下的祸端已没法一笔抹去，但同时它又仍处于青春期，所以这些安全问题又貌似还没到那么急迫和突显。尽管这时语法得到了改善，无用的标签被废弃了，各种验证器也写好了，江湖座次也已排定了，但浏览器差不多还是老样子：臃肿、古怪和难以预测。

不久发生了一件有趣的事情：微软推出了一个颇不起眼的专有 API，名叫 `XMLHttpRequest`，这个名字也颇让人摸不着头脑。这个新玩意儿本来并不重要，最开始这个 API 只是在微软 Web 版的 Outlook 应用里小试了一把牛刀。但最后 `XMLHttpRequest` 却大放异彩，因为它实现了客户端 JavaScript 和服务器之间不受限制的异步 HTTP 通信，而无需额外的时间开销和页面重载。以这种方式，这个 API 更是对其后出现 Web 2.0 热潮推波助澜，Web 2.0 就包括许多响应式的基于浏览器的复杂应用，用户能轻松使用复杂的数据集，方便地实现群体合作和个人内容出版等，它已经一脚踏入属于传统客户端的神圣领域里了，变成“真正的”软件了。可以理解，它引起了极大轰动。

1.2.4　Web 2.0 和第二次浏览器大战：2004 年之后

伴随着互联网和各种浏览器的日益流行，`XMLHttpRequest` 也把 Web 推到了激动人心的新高度，同时，也给我们带来了许多会影响个人和商业界的重大安全隐患。到 2002 年，蠕虫和浏览器漏洞变成了媒体上经常能看到的主题。由于微软的领导地位和对安全相对疏忽的态度，它承受了最多的公关压力。微软对这些问题一概视而不见，但这些压力积累发酵过后，终于造就了一次小规模的反抗。

到 2004 年，浏览器舞台上出现了一位新选手：Mozilla Firefox（原网景公司 Navigator 浏览器的后裔，由开源社区开发），它针对的正是 IE 糟糕的安全性和与标准的不兼容性。在获得 IT 专栏作家和安全专家的普遍肯定后，Firefox 很快获得了 20% 的市场份额。尽管这位后来者很快也被证明和微软浏览器一样，受到各种安全漏洞困扰，但由于 Firefox 的开源特性，以及无需迎合顽固的企业用户，使它的问题修复较为迅速及时。

注意　为什么浏览器开发领域里的竞争如此激烈呢？严格来说，浏览器的市场份

> 额并没有办法直接转化成金钱收入。但专家们认为这关乎权势地位：因为可以通过浏览器来捆绑、推销或边缘化某个在线服务（即使像默认搜索引擎这么简单的服务），也就是说谁控制了浏览器，谁就控制了互联网。

除了 Firefox，微软还有别的忧虑。随着越来越多的应用（从文本剪辑器到游戏）转而以 Web 方式运行，它的旗舰产品微软视窗操作系统正越来越沦为浏览器的工作平台。这显然是个不利的信号。

这些事实连同突然杀入市场的苹果公司浏览器 Safari 和 Opera 浏览器在智能手机领域的步步领先，一定使微软的高层深觉头痛不已。他们已经错失了 20 世纪 90 年代互联网第一波高潮；当然他们不想再犯同样的错误。微软重新加大了对 IE 浏览器的投入，发布了有极大提升和在某些方面来说更安全的版本，从 IE7、8 迅速迭代到了 IE9。

IE 的竞争对手们拿出了各种新功能与之对抗，甚至声称自己对标准的支持更好（虽然也只是表面上的）、浏览更安全、效率更为提高。`XMLHttpRequest` 出乎意料之外的成功引人瞩目，大家迅速把过去的经验教训抛到了脑后，有时候也会单方面引入不成熟或不安全的设计，如 Firefox 浏览器的 `globalStorage` 和 IE 的 `httponly` Cookie，这完全就是在碰运气了。

好像还嫌事情不够混乱，由于对 W3C 理事会在创新性上的不满，一群参与者创建了一个全新的标准组织，叫网页超文本技术工作小组（Web Hypertext Application Technology Working Group，WHATWG）来主导 HTML5 协议的开发，这是对现有标准的第一次整体性和把安全也考虑进去的修订，但据报道，他们经常由于专利纷争而没法和微软达成一致。

在 Web 的整个发展历程中，由于缺乏统一的远景目标和完整的安全规范，其开发模式非常独特，整个发展过程竞争激烈、变幻莫测，与政治牵扯过多，结果错漏百出。这些问题都对浏览器现在的工作方式以及怎样安全地处理用户数据有深远的影响。

但问题在于，在可见的将来这种情况是不会有什么改变。

1.3　风险的演化

很明显，Web 浏览器以及它们相关的文档格式和通信机制，以一种非同寻常的方式演进变化着。今时今日浏览器安全漏洞数一直居高不下的原因，大概都源自这种演进方式，但光是这些演进方式本身还说明不了这些问题的独特性和重要性。作为本章的收尾，让我们回顾 Web 领域里一些最具影响力的风险要素，并探讨为什么在 Web 出现以前，却没有碰到相类似的问题。

1.3.1　用户作为安全风险的一个环节

也许浏览器里最突出（完全和技术无关）的一个特点就是大多数使用者完全不懂技术。当然，打从计算机诞生之日起，电脑小白们就一直是挺娱乐、挺无伤大雅的问题。但自从 Web 日渐深入人们的生活后，由于门槛极低，所以我们碰到了一个新情况：大多数用户对如何安全上网完全没概念。

很长一段时间以来，工程师们在开发普通的软件时，一般来说是完全不会考虑到使用者的计算机水平的。大多数情况下这样做确实没什么大问题；比如某个文本输入框的数值不太对，对整个系统的安全性几乎没任何影响。如果用户的操作有问题，那他也多半玩不转这个软件，这也算是个极好的自我纠错机制了。

但这套规律在 Web 浏览器上却行不通。和其他复杂的软件不同，哪怕使用者连文本编辑器都用不来，但使用浏览器却完全没问题。但同时，又只有对计算机技术和公开密钥体系（Public-Key Infrastructure，PKI）这样的技术术语相当了解的人，才有可能安全地使用浏览器。不用说，在时下林林总总热门 Web 应用的目标人群中，绝大部分都不符合这一要求。

而浏览器的许多做法，却弄得它像是由电脑极客们（geek）设计的，专供极客们使用的软件呢，譬如时不时蹦出一些难解又不连贯的出错提示信息，会涉及一堆复杂的配置，还有无数令人困惑的安全警告和提示。由伯克利和哈佛大学研究人员在 2006 年发布的一份著名的报告中显示，例如像状态栏上是否出现了带锁小图标这样的提示[4]，普通用户几乎肯定不会留意到这些信息，而开发人员却会非常清楚。在另一份研究中，斯坦福和微软的研究人员在检查新的“绿色 URL 地址栏”安全警示标识的功效时也得出了相似的结论。这个机制的本意是为了提供比带锁小图标更明显的安全标识，但结果也往往会误导用户，以为只要见到绿色的特定形状就是可信的，也不管这种绿色标识出现在哪里[5]。

有些专家认为，不能把普通用户的无知怪罪到软件开发商头上，因为这根本不属于软件工程的问题。但另一种观念是，既然这类软件随处可见且广泛使用，却要求用户必须自行判断许多安全相关问题，而这些问题得需要具备一定的技术知识，可在用户最开始使用软件时又完全未做任何技术水平的限定，这是非常不负责任的做法。但仅指责浏览器开发商们也并不公平；毕竟计算机产业作为一个整体，在这个领域确实没什么靠谱的解决方案，要确保在复杂用户界面（UI）里的用户操作万无一失，也几乎没什么研究成果可供参考。毕竟，我们现在的水平还仅停留在勉强确保 ATM 级别的界面里不会犯错呢。

1.3.2　难以隔离的 Web 运行环境

Web 的另一个古怪特点就是，完全无关的应用和应用相关数据之间的隔离度非常弱。

在过去 15 年间，个人电脑时代的传统模式里应用层的数据对象（文档）、用户层的代

码（应用程序）和操作系统内核之间的边界非常清晰，由操作系统内核负责所有跨应用程序的通信、硬件的输入 / 输出（I/O）以及通过可配置的安全策略限制应用程序的越界行为。这些边界已经被研究得很透彻，也确实对创建实际可用的安全策略大有助益。在文本编辑器里打开的文件，几乎不可能去窃取你的电子邮件，除非在具体实现上很不幸确实有重大缺陷，导致所有的隔离层都彻底失效了。

但在浏览器的世界里，却压根不存在这种隔离：文档和代码就交织在同一个 HTML 文件里，完全无关的应用之间最多只能算部分隔离（实际上所有网站使用的是同一个全局 JavaScript 运行环境），除了要遵守寥寥几个灵活的浏览器级别的安全控制框架，不同网站之间各种交互都是隐式默许的。

从某种程度来说，Web 的这种模型与那些没有强壮的内存保护、CPU 抢占或多用户支持的非多任务操作系统（如 CP/M、DOS 等）相类似。但明显不同的是，这些早期系统不太可能出现同时运行多个不受信任且易被攻击的应用的情况，自然也无需特别顾虑安全上的问题。

所以在老系统里不太可能发生文本文件窃取电子邮件这种事情，但令人恼火的是，类似情况在 Web 上却屡见不鲜。实际上，所有的 Web 应用都曾为不请自来的恶意跨域访问，付出过沉重的代价，最后只能用一些笨拙的方法勉强分离代码和要显示的数据。所有的 Web 应用都在这个事情上败下阵来，只是时间早晚而已。许多内容相关的安全问题，如跨站脚本（cross-site scripting）或跨域请求伪造（cross-site request forgery）在 Web 领域都很常见，但在专用客户端的架构里，却极少碰到类似的情况。

1.3.3　缺乏统一的格局

当然，幸运的是，浏览器的安全也不是就完全无药可救了，尽管不同 Web 应用之间的隔离很有限，但某些安全机制还是为那些最严重的攻击提供了基本保障的。说到这里，也带出了为何 Web 这一主题如此有意思的另一特点：因为它完全没有一个通用的整体性安全模型。在这方面，我们也没指望能有一个解决世界和平这么宏大的愿景，这里说的只是一些常规通用又灵活的安全范式集合，即使不能适用于所有场合，但可以解决绝大多数相关的安全逻辑就行。例如，在 UNIX 系统里，`rwx` 方式的用户 / 属组许可模式就是这么一种高度统一的安全模式。但在浏览器领域里有什么呢？

在浏览器领域里，“同源策略”（same-origin）机制就可算是这类的核心安全范式了，但实际上这套本身就问题多多的机制也仅是跨域交互里一个小的子集而已。即使仅讨论同源策略，也还有不少于 7 种的使用场景，这使得不同应用之间的安全边界也会略有差异[⊖]。另

⊖ 这里涉及的 7 个主要类别将在本书第二部分里详细讨论，包括在 JavaScript DOM 访问、XML HttpRequest API、HTTP Cookie、本地存储 API、还有类似 Flash、Silverlight 和 Java 插件等情况下的同源策略。

外还有若干种机制，它们和同源模式没有什么关系，但控制了浏览器的其他关键行为（这些机制的作者往往是怎么容易实现就怎么来）。

所以结果就是，有诸多这类零零碎碎要小聪明的调整，但谁都无法担起浏览器的安全大任。由于缺乏真确性，也无从判断单个应用在什么时候结束，新的应用在什么时候开始。在这样的困境之下，到底怎样才算是出现攻击了呢？究竟是需要加载或取消权限许可，还是需要完成某项安全相关的任务呢？我们能做的，往往不过是“双手合十，听天由命，求个平安”而已。

让人觉得奇怪的是，许多原本出于好意希望改善安全机制的努力，最后效果却往往适得其反。为了获得优雅的效果，许多这类机制往往会引入新的安全边界，但这会导致与已混乱不堪的老的安全机制无法相匹配了。如果新的控制机制粒度更细，新机制则很可能会被老机制所拖累，带来的不过是一种虚幻的安全假象；如果新机制的粒度更粗，则可能导致连现有 Web 机制所依赖的微妙保障也没有了（Adam Barth 和 Collin Jackson 在他们的学术论文里讨论过浏览器不同安全机制之间的有害冲突[6]）。

1.3.4　跨浏览器交互：失败的协同

通常来说，一个包含多种应用程序的生态圈，整体的薄弱程度，可以简单地认为就是每个软件产品问题的叠加。在某些场合里，总体的受影响程度甚至会小于这个叠加值（多样性使得可耐受性也提高了），但一般认为肯定不会超过所有问题的总和。

但 Web 却再次打破常规。安全圈已经发现当多种浏览器企图彼此交互时，有一系列难以归咎到哪段具体代码头上但又非常严重的漏洞。你没法揪出哪个特定产品就是罪魁祸首：它们都不过是在尽责地完成任务而已，唯一问题是，没有为它们定义一个全体浏览器都理应遵守的公共规范。例如，某个浏览器认为，根据它的安全模式，把某个特定的 URL 传给外部程序或者存储 / 读取硬盘上某些类型的数据是安全的。对这类假设，几乎总有某个浏览器完全不认可，并且指望其他浏览器会按自己的规矩办事。而每个厂商也都希望把手伸得尽量长，所以往往会在未告知用户也未得到用户许可的情况下，强行用自家浏览器打开网页。例如，Firefox 通过在其他浏览器里注册 `firefoxurl:` 协议，使得网页强行在它的浏览器里打开；微软则在 Firefox 里安装自己的 .NET 网关插件；而 Chrome 的做法和 Firefox 如出一辙，它会在 IE 浏览器里注册 `cf:` 协议。

> **注意**：在这种混乱的交互中，指责任意一方都属徒劳。最近有一件和 `firefoxurl:` 协议有关的漏洞，微软和信息安全圈里半数的人在指责 Mozilla，而 Mozilla 和另一半的专家则怪罪于微软[7]。谁对谁错其实并不重要：反正结果还是一团乱麻。

另一个紧密相关的问题就是，即使浏览器的安全机制表面上看来很相似，实际上却并不兼容，这种情况在 Web 出现之前很少发生。如果各家浏览器安全模型是不同的，那么某条 Web 应用开发规范对其中一种浏览器可能是合理的，但对另一种却可能完全不适用且会产生误导。实际上，哪怕一些非常基本的任务，如打开一个用户提供的纯文本文件，在某些浏览器里也无法安全地实现。而这些问题程序开发人员往往意识不到，除非他们正好使用了这种受影响的浏览器——即使这样，也往往需要等他们踩上地雷才会意识得到。

最后，本节描述的问题在安全缺陷分类学里会归到一个很吓人的全新类别里：“无法描述又未被记录的各种问题”，但这类问题实际上随处可见。

1.3.5　客户端和服务器端界限的日益模糊

信息安全人员喜欢静态的世界和清晰分配的角色，这样在映射到安全交互上至少还有个熟悉的参照体系，否则事情可能会很复杂。比如，我们假设有两位努力工作的诚实用户爱丽丝和鲍勃想要互相通信，而居心叵测的攻击者马洛里则在暗处想攻陷他们。然后我们有客户端软件（基本上没啥话语权，有时候恶意的 I/O 终端会乱发服务请求），还有谦逊的服务器们会老老实实地响应客户端的这种恶搞。程序员清楚这些角色后就开始处理问题，然后创建出一个相当全面又可供测试的网络计算环境。

Web 的起源完全符合常规的“客户端 – 服务器端”架构，但客户端和服务器端响应的功能边界被迅速模糊了。罪魁就是 JavaScript 技术，它在浏览器里（也就是“客户端”）代理了 HTTP 服务器的应用逻辑的执行，这么做有两个非常有吸引力的原因。首先，这种方式使得用户界面的响应更灵敏，因为每次微小的 UI 状态变化就不需要再和服务器端进行同步了。其次，极大地降低了服务器端的 CPU 和内存要求（也就是降低了服务的运营成本），因为相当于通过遍布全球的每台独立计算机的参与，降低了运算量。

尽管“客户端 – 服务器端”边界的模糊起源于单纯的目标，但连同客户端的常规功能一起，迟早也会需要引入安全机制的。因为既然数据都是由客户端 JavaScript 动态生成的，那仅仅在服务器端严格地清理 HTML 页面又有什么意义呢。

在某些应用里这种趋势已经走向极致，服务器逐渐变成了一个沉默的存储设备，几乎所有的解析、编辑、显示和配置任务都转移到浏览器端执行了。按照这种设计，甚至可以用诸如 HTML5 的持久存储作为离线 Web 扩展而不太需要依赖于服务器端。

尽管在整个应用的设计里，这点改变可能不算什么大事，但决不能指望客户端负责所有的安全问题。例如，即使服务器只是用作沉默的存储设备，客户端也不能不受限制地访问到服务器端其他用户的数据，也不能由客户端来指派访问控制权限。最后，因为把所有的应用安全逻辑都放在服务器端不是很合理，而把它完全移到客户端也完全不可能，结果导致大部分应用取而代之的是搞个随意的中间层，在此中间层里客户端和服务器端的组件

往往就很难区分了，对责任的逻辑隔离也完全欠奉。这有别于传统程序里优雅健康的安全角色划分，导致其设计和应用的行为也完全不可同日而语。

这种情况带来的远不止是设计层面的混乱，还使得复杂度大增。在传统的“客户端 – 服务器”模型里都有用途清晰具体的 API，无需考虑客户端就可以非常容易评估服务器的行为，反之亦然。此外，在每个组件里，都可以轻易地隔离出较小的功能区间，确定在此区间内会有什么操作。但 Web 的全新模型，再加上 Web 上常见应用 API 都是既模糊又临时，所以通过以前那些分析手段，理性地推断一个系统的安全性已经完全不可能了。

尽管人们期望能找出 Web 上标准化的建模和测试协议，但最终出人意料地失败了，这也使得 Web 安全在信息安全领域具有独特而令人胆寒的地位。

全球浏览器份额，截至 2011 年 5 月的数据

开发商	浏览器名称	市场份额	
Microsoft	Internet Explorer 6	10%	52%
	Internet Explorer 7	7%	
	Internet Explorer 8	31%	
	Internet Explorer 9	4%	
Mozilla	Firefox 3	12%	22%
	Firefox 4+	10%	
Google	Chrome	13%	
Apple	Safari	7%	
Opera Software	Opera	3%	

来源：数据来自公开的 Net Applications 报告。⊖

⊖ Net Applications website, *http://marketshare.hitslink.com/browser-market-share.aspx?qprid=0, http://marketshare.hitslink.com/browser-market-share.aspx?qprid=2* (accessed June 13, 2011).

第一部分

对 Web 的解剖分析

本书第一部分重点分析 Web 浏览器的主要工作原理，也就是说，包括协议、文档格式和它内部使用的编程语言。因为在常见浏览器里我们所熟知的，对用户可见的各种安全机制，都和这些内部工作机制深切相关，所以在更深入 Web 的黑森林探险之前，让我们先来关注一下这些内部机制。

第2章

一切从 URL 开始

最易辨识的 Web 标志就是 URL（Uniform Resource Locator，统一资源定位器）了，它由一串很简单的文本字符组成。一条格式正确且符合规范的 URL 必然对应着远程服务器上某个独一无二的资源（在这个解析的过程中，还需要实现另外一些相关的功能）。URL 语法规范是浏览器地址栏的基础，而地址栏中的信息正是每个浏览器用户界面中最重要的安全标识。

除了内容检索时用到的真正的 URL，还有几种语法与之相类似，但用于浏览器端功能的伪 URL，这些功能包括内置的脚本引擎、几种特别的文档渲染模式及诸如此类的功能。所以毫无疑问，这些伪 URL 对应的动作对链接了它们的站点有重大安全影响。

搞清楚浏览器对 URL 的解析机制以及它们带来的副作用，是我们和 Web 应用都需要面对的最基础且最常见的安全问题，同时 URL 机制自身也是错漏百出。URL 的语法是由 Tim Berners-Lee 制定的，具体内容主要参见 RFC 3986 文档[1]；在 Web 里的实际运用请参见 RFC 1738[2]、2616[3] 以及若干相对次要的标准文档。这些文档都非常详细，这导致 URL 的解析模式相当复杂，但它们又未能足够精确地描述出在客户端软件里需要怎样实现 URL 机制才能做到既兼容又准确。此外，各家软件开发商出于各自的考虑，都会稍许偏离这些规范。让我们来仔细看一下貌似简单的 URL 在实际环境中是如何工作的。

2.1　URL 的结构

图 2-1 显示的是一个符合规范的绝对（absolute）URL，它包括了访问某特定资源所需要的全部信息，绝对 URL 和访问时的状态完全无关。与之相对应的是省略了部分信息的相对（relative）URL，如 `../file.php?text=hello+world`，它需要根据当前浏览所在上下文环境里的基准 URL，才能确定完整的 URL 地址。

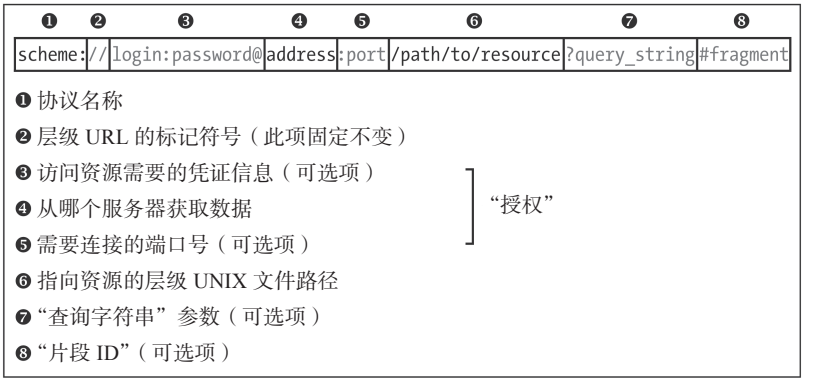


图 2-1　URL 绝对路径的结构

绝对 URL 的各个部分看起来都还算一目了然，但每个部分都有一些值得注意的问题，让我们来回顾一下吧。

2.1.1　协议名称

协议名称由一串不区分大小写的字符串组成，以单个冒号结束，表明获取该资源时需要使用的协议。官方认可的有效 URL 协议统一由 IANA（Internet Assigned Numbers Authority，互联网数字分配机构）维护，该机构更广为人知的功能是管理 IP 地址空间[4]。IANA 当前的有效协议名包括 `http:`、`https:` 和 `ftp:` 等几十项，而实际上，常用浏览器和第三方应用往往还支持若干额外的协议，其中一些还会带来安全问题。（特别值得关注的是几种伪 URL，如 `data:` 和 `javascript:`，在本章和全书的后续内容里都会有详细的讨论。）

在浏览器和 Web 应用要做进一步解析之前，它们还得先区分要处理的是完整的绝对 URL 还是相对 URL。在 URL 地址的最前面是否包含有效的协议名称原本是最关键的区别，RFC 1738 里是这么定义的：绝对 URL 应遵守以下规定，在冒号“：”之前，只能出现字母数字和“+”、“–“和“.”这几个符号。但在实际环境里，每种浏览器都稍许偏离了这个指引，它们全都会忽略前导换行符和空格。IE 还会忽略所有的不可打印字符（ASCII 代码表里 0x01 ~ 0x1F 之间的字符）。在此基础之外，Chrome 更是会忽略 0x00 和 NUL 空字符。大多数浏览器的具体实现还会忽略在协议名中出现的换行符和制表符，而

Opera 浏览器还接受在协议字符串出现高位字符。

由于这些不兼容性，那些需要区分相对和绝对 URL 的 Web 应用就必须谨慎地拒绝任何异常的语法。但我们很快就会发现，即使这样做了也还是不够的。

2.1.2　层级 URL 的标记符号

根据 RFC 1738 规定的语法，在授权信息之前，每个层级结构的绝对 URL 里都应该包括固定的字符串“//”。按这个规范的意思，如果没有这个字符串，就无法确定 URL 后续部分的格式和功能了，只能把它们看成一个含糊的与特定协议相关的值。

> **注意**　一个非层级结构的 URL 例子就是 mailto: 协议，它用于指定电子邮件地址，可能还包括主题信息（mailto:user@example.com?subject=Hello+ world）。这样的 URL 会被传递到默认邮件客户端程序而无需经过其他解析。

理论上，这种统一的层级 URL 语法的概念确实很优雅。因为应用可以无需关注某个协议的具体实现，就能够提取到只和地址有关的信息。例如，看到某种特定的名为 wacky-widget: 的协议，浏览器就能确定 http://example.com/test1/ 和 wacky-widget://example.com/test2/ 指向的是同一个受信任的远程主机。

但相当遗憾的是，这个规范包含着一个有意思的缺陷：上面提到的那个 RFC 文档并没有指明，如果碰到一个非层级结构的 URL 又带有“//”前缀，当做何处理；反之亦然。在更早期的 RFC 1630 协议里用作参考的 URL 解析器实现就无意中包含着这个漏洞，这导致后来与 URL 有关的处理都有问题。在若干年后发表的 RFC 3986 里，作者出于兼容性的考虑，懦弱地接受了这些漏洞并允许解析这类 URL。这样导致的后果就是，各家浏览器对以下这些例子的解析让人很摸不着头脑：

- **http:example.com/**　当没有符合要求的基准 URL 环境时，在 Firefox、Chrome 和 Safari 里这个地址会被认为等同于 http://example.com/，如果有基准 URL，则会认为这是一个指向目录 example.com 的相对路径。
- **javascript://example.com/%0Aalert(1)**　在所有常用浏览器里，会认为这个字符串是一个有效的非层级伪 URL 并以 JavaScript 方式来执行 alert(1) 这段代码，显示一个简单的对话框。
- **mailto://user@example.com**　IE 认为这是一个有效的指向电子邮件地址的非层级 URL；“//”的部分会直接被忽略掉。而其他浏览器则不认可这个写法。

2.1.3　访问资源的身份验证

URL 里身份验证的部分属于可选项。在向服务器端获取数据时，有可能需要在该位

置指定一个用户名或密码。但这个抽象的 URL 语法本身，与具体的用户名密码等身份验证信息的交换并无实质性关系，身份验证信息的传输是和协议相关的。对那些不支持身份验证的协议，如果在 URL 里强行加入这部分信息该做何处理，协议并未做出规定。

如果没有提供身份验证信息，浏览器默认以匿名的方式获取数据。在 HTTP 和其他几种协议里，这意味着没有传送任何身份验证信息；对 FTP 协议，这包含着一个名为 `ftp` 的账号和一个假的密码。

除常规的 URL 分隔符之外，大多数浏览器对身份验证部分的数据几乎接受任何字符，但有两个例外：出于某些尚未明了的原因，Safari 拒绝了许多字符，包括“<”、“>”、“{”和“}”，而 FireFox 还拒绝换行符[㊀]。

2.1.4　服务器地址

对完整的层级 URL 来说，服务器地址部分必须指定一个不区分大小写的域名（例如 `example.com`）、一个 IPv4 地址（例如 127.0.0.1）或在一对方括号里的 IPv6 地址（[0:0:0:0:0:0:0:1]），用以标识请求资源所处的服务器位置。 FireFox 还接受写在一对方括号里的 IPv4 地址和主机名，而其他浏览器则会拒绝这种写法。

尽管 RFC 里只允许符合规范的 IP 地址写法，但大多数应用所依赖的标准 C 类库却比较灵活，可以接受八进制、十进制和十六进制的写法，甚至可以接受把其中几个或全部 8 位元数据（Octet [㊁]）拼在一起再转成单个整数的写法。所以以下的各种写法实际上是一样的：

- **`http://127.0.0.1/`**　这是 IPv4 地址的正规写法。
- **`http://0x7f.1/`**　同一个地址，但先以十六进制数字表示标准写法里的第一个 8 位元（Octet），剩下的 3 个 8 位元数据先分别按十六进制的格式拼在一起，再整体地转化成一个十进制的整数。
- **`http://017700000001/`**　同一个地址，以 0 为前缀，后面是把全部 4 个 8 位元数据的十六进制数值拼在一起，再统一转化成单个八进制整数[㊂]。

这种很随意的解析方式也相似地出现在域名里。理论上来说，域名里能用的字符集是

㊀ 这样做也许是考虑到 FTP 的问题，FTP 协议在传输用户认证信息时无需任何编码；在该协议中，传递一个换行符会被服务器错误地解析为一个新的 FTP 命令的开始。其他浏览器在传输 FTP 客户认证时可能会把不符合规范的字符用百分号方式编码，又或仅仅是把有问题的字符丢弃掉。

㊁ Octet 这个词在本书里译做“8 位元”，在常规情况下它完全等同于 Byte，但实际上使用 Octet 来表示一个八位的信息比 Byte 更精确。因为某些历史原因，Byte 在某些操作系统上，有可能代表 4 ~ 10 位的字符。详见：http://t.cn/zTMjmfS。——译者注

㊂ 如原始 IP 地址为 192.168.100.110，其 16 进制的表达为：0xC0.0xA8.0x64.0x6E，按 8 位元格式拼成 16 进制的 C0A8646E，再转换成 8 进制的 030052062156，4 者完全等效。——译者注

很有限的（按照 RFC 1035[5] 的定义，应该只能出现字母和数字、“.”和“–”），但很多浏览器查询任何信息几乎都依赖于操作系统的解析器，而操作系统的处理通常不太严谨。各种客户端能接受以及传递给系统解析器的主机名（Hostname）字符集也都不尽相同。Safari 最严谨，而 IE 宽容度最高。需要引起注意的是，大部分浏览器都会忽略出现在 URL 内且数值范围在 0x0A ~ 0x0D 和 0xA0 ~ 0xAD 之间的控制字符。

注意　多数主流 URL 解析器都有一个令人诧异的行为，它们都会主动地把主机名里的全角句号“。”（在字面上表示结束，对应的 Unicode 代码是 U+3002）直接替换为点号，但这种情况如果出现在 URL 的其他部分里则不会这样处理。据说这是因为某些中文键盘的映射使得这种环境的用户很可能是想输入那个 7 位 ASCII 值的点号而不是这个全角句号。

2.1.5　服务器端口

服务器端口部分是可选的，通常在服务器连接的网络端口并非标准端口时才会用到。基本上浏览器支持的所有协议以及第三方应用都会以 TCP 或 UDP 作为传输方式，而 TCP 和 UDP 都会依赖于一个 16 位端口号[⊖]来区分运行在一台机器上的不同服务。服务器上每种协议通常都会关联一个默认的服务端口（如 HTTP 为 80、FTP 为 21，等等），但这个默认值可以在 URL 里另行改写。

注意　这个功能无意中造成了一个有趣的副作用，我们可以用浏览器向任意网络服务发送攻击者提供的数据，尽管浏览器并不支持这些服务的协议。例如，你可以把浏览器指向：`http://mail.example.com:25/`，而这里 25 端口实际上运行的是简单邮件传输协议（Simple Mail Transfer Protocol，SMTP）服务，并非 HTTP 协议。这个问题曾导致一系列的安全问题，浏览器也为此引入了一系列不太完美的解决方案，这些在本书第二部分会更详细讨论。

2.1.6　层级的文件路径

URL 的下一个组成部分叫层级文件路径，它非常形象地体现了从服务器获取特定资源的方式，例如 `/documents/2009/my_diary.txt`。规范里也公开表明，这个格式是直接从 UNIX 目录语义借用过来的，所以也支持在路径里出现的“/../”和“/./”，而对非绝对路径形式的 URL，也会根据这种目录格式，加上基准路径再对应到其相对位置上去。

⊖　这意味着端口号数不能大于 2 的 16 次方，即 65536。——译者注

在 20 世纪 90 年代的时候，Web 服务器只是用来存放一堆静态文件和少量可执行脚本的简单的工作网关，所以使用文件系统模型是很自然的选择。但在那之后，很多当前的 Web 应用框架已经和文件系统没什么关联了，它们都是直接和数据库对象或常驻程序的注册位置打交道了。虽然还是有可能把这些数据结构和运行良好的 URL 路径对应起来，但人们往往不会这么做或者不会特别严谨地这么做。所有这些问题都会导致内容的自动索取、索引和安全测试要比预想的来得更复杂。

2.1.7　查询字符串

查询字符串（Query String）也是个可选项，用于把一串非层级格式的任意参数传递给由前面路径所对应的资源。例如，以下例子就是一个把用户提供的信息传递给服务器端脚本，用于搜索的常见形式：

```
http://example.com/search.php?query=Hello+world
```

大多数网站开发者都很熟悉这种搜索字符串的特定格式；它是由浏览器在处理 HTML 文件里的表单时生成的，格式如下：

```
name1=value1&name2=value2...
```

但让人意外的是，在 URL 的 RFC 文档里其实并没有硬性规定要使用这样的格式。实际上，规范里是把查询字符串当作一堆含混笼统的数据对待的，关键是接收者最后要怎么用它，所以有别于路径，它无需遵守特定的解析规则。

关于我们上面提到的这种查询字符串的常见格式的说明，请参见信息量很庞大的 RFC 1630[6]、与邮件协议相关的 RFC 2368[7] 以及 HTML 规范里与表单处理相关的内容 [8]。但这些规定全都不是强制性的，因此，如果某个 Web 应用真要完全不顾常规，把一堆乱七八糟的数据放在 URL 的查询字符串位置上，也并不能算错。

2.1.8　片段 ID

片段 ID（Fragment ID）的角色和查询字符串有点类似，但用法颇为晦涩，它是用于客户端而非服务器端（实际上，这个值根本就不应该传回给服务器端）的一种可选信息。在 RFC 里没有明确规定片段 ID 的格式或功能，但暗示性地提到可以用于在返回的文档里定位“子资源”或对文档渲染的方式提供一些帮助性的指导。

在实际运用中，片段 ID 在浏览器里只有一个用途：指向 HTML 页面里的某个锚点名称，用于页面浏览定位。这套逻辑很简单。如果在 URL 里的锚点名称与 HTML 页面里设定的锚点标签匹配，文档就会滚动到该定位标签的位置上，方便阅读浏览；否则，就啥动作也没有。由于片断 ID 的信息是包含在 URL 里的，所以能用它直接分享一篇长文里的具

体位置或加入书签以便于保存。由于片段 ID 只能用于在当前文档里定位，所以无需再从服务器端加载数据，只是根据用户的点击动作，稍许更新一下 URL 里的片段 ID 信息而已。

这种有趣的属性现在有了新用法，可以用这个值来存放一些临时的数据：如存储客户端脚本需要的各种状态信息。例如，地图浏览的应用可以把地图的当前坐标置于这个标识里，这样就可以把链接加到书签里或发送出去与他人共享，地图应用就能再回到相同的位置。和改变查询字符串的效果不同，片段 ID 不会触发页面重载，也就不需要时间开销，这使得它成为数据存储上的杀手级特性。

2.1.9　把所有的东西整合起来

上述提到的各个 URL 部分都有一些特定的保留字符：如正斜杠、冒号、问号等。为了确保浏览器的正常工作，这些字符除了用于分隔 URL 的不同部分，在 URL 里不能有其他的用途。谨记这点，现在假设我们要设计一个示例算法，来模拟浏览器的解析方式，把绝对 URL 按各个功能项分割开来。一个较为合理的算法大致如下。

步骤 1：提取协议名称。

扫描第一个“:”字符。在该字符左边的 URL 部分就是协议名称。如果获得的协议名称中出现了不应有的字符，这可能就是个相对 URL，获得的并不是协议名称。

步骤 2：去除层级 URL 标记符。

字符串“//”应该跟在协议名称的后面。如果发现有该字符串，则跳过该标记符；如果未找到，就不用管了。

> **注意**　在某些运行环境的解析中，出于可用性的考虑 URL 标记符甚至可以完全不用斜杠、只用 1 个斜杠、用三个甚至三个以上的斜杠。因着同样的思路，可能是为了帮助那些不熟悉的使用者吧[⊖]，IE 浏览器从一开始就接受在 URL 任意位置使用反斜杠（\）替代正斜杠。除 Firefox 以外，所有的浏览器也逐渐遵从这一规律，会接受像这样的 URL `http:\\example.com\`。

步骤 3：: 获取授权信息部分。

依次扫描“/”、“?”或“#”符号，哪个先出现以哪个为准进行截取，从 URL 里提取出来的部分就是授权部分信息。刚才也提过了，大多数浏览器还接受反斜杠“\”作为正斜杠形式分隔符的替换写法，也要考虑这种情况。除了 IE 和 Safari 浏览器之外，分号（;）

⊖ 与各种 Unix 操作系统不同，微软的 Windows 使用反斜杠而非正斜杠作为路径的分隔符（例如：`c:\windows\system32\calc.exe`）。微软可能考虑到用户要在 Web 上使用另一种斜杠会感到困惑，同时也为了解决 `file:`URL 协议以及其他需要和本地文件系统打交道的类似机制中可能存在不一致斜杠的问题。当然其他一些特定的 Windows 文件系统属性（如不区分大小写）就不能照搬到 URL 里了。

也是授权信息部分可接受的分隔符；这么做的原因还未可知。

步骤 3A：定位登录信息，如果有的话。

授权部分信息提取出来后，在截取出来的信息里再寻找（@）符号。如果找到了，那在它前面的部分就是登录信息，这部分登录信息还需要做进一步分解，在第一个冒号前面（如果有的话）是用户名，后面则是密码数据。

步骤 3B：提取目标地址。

授权信息部分剩下的就是目标地址了。第一个冒号分隔开的就是主机名和端口号。用方括号括起来的是 IPv6 地址，这也是特例。

步骤 4：确定路径（如果的确存在）。

如果授权部分的结尾跟着一个正斜杠——或者某些场景里，跟着一个反斜杠或分号，就像之前提到的，依次扫描下一个“?”、“#”或字符串结尾符，哪个先出现以哪个为准。截取出来的就是路径部分，当然最后应根据 UNIX 路径语义进行规范化整理。

步骤 5：提取查询字符串（如果的确存在）。

如果在上一条解析里，后面跟的是一个问号，则继续扫描下一个“#”或到字符串结尾，哪个先出现以哪个为准。这中间的部分就是查询字符串。

步骤 6：提取片段 ID（如果的确存在）。

如果成功解析完上一条信息，它的最后跟着“#”符号，从这个符号到整个字符串的最后，就是片段 ID。换而言之，整个任务完成啦！

这个算法看起来很普通，但它揭示了一些甚至资深程序员也往往忽略掉的微妙细节。它也展示了对普通用户来说，要分辨一个 URL 会被怎么解析其实非常困难。让我们以这个相当简单的例子展开讨论吧：

```
http://example.com&gibberish=1234@167772161/
```

这个 URL 的目标地址——实际上就是那串经过编码的数字，解码翻译过来其实是 10.0.0.1（十进制的 167772161 转成十六进制则为 A000001，所以实际对应的是：0A.00.00.01，即 10.0.0.1。——译者注）——如果不是专家，实在不容易把它辨认出来，很多用户可能会认为他们是在访问 example.com 呢[⊖]。

好吧，也许这个还算简单！让我们再看一个：

```
http://example.com\@coredump.cx/
```

在 Firefox 里，这个 URL 会把用户带往 `coredump.cx`，因为 `example.com\` 会被认为

⊖ 这种基于 @ 字符玩的把戏迅速被用于针对普通用户的各种在线诈骗。针对这个问题，有些厂商的做法既笨拙又古怪（如 IE 禁用基于 URL 地址的授权方式，而 FireFox 则会显示警告信息并中断执行这样的解析），有的则相对合理（如某些浏览器会在地址栏里高亮显示出服务器的名称）。

是个合法的登录信息。而在其他的浏览器里，“\”会被认为是个路径分隔符，所以用户最终会访问 example.com。

IE 里一个更令人抓狂的例子如下：

```
http://example.com;.coredump.cx/
```

微软浏览器允许在主机名称里出现“;”符号，并成功解析到了这个地址，当然这得需要 coredump.cx [⊖]提前做了这样的域名解析设置。大多数其他浏览器会自动把 URL 纠正成 http://example.com/;.coredump.cx，然后把用户带到 example.com（Safari 除外，它会认为这个写法有语法错误）。如果你觉得局面已经一片混乱了，请记住，我们这才刚刚开始讲浏览器是怎么工作的！

2.2　保留字符和百分号编码

上一节的 URL 解析算法里有个大前提：就是某些用于语义分隔的保留字符没有出现在 URL 里（也就是说用户名、请求路径等位置不会出现这些字符）。这些会破坏语法的分隔符包括：

```
: / ? # [ ] @
```

RFC 还规定了若干的底层分隔符，尽管没有给出它们的详细用途，但这些符号是留给上层协议或具体的应用来实现的：

```
! $ & ' ( ) * + , ; =
```

以上所有字符原则上都是被禁止的，但有时候也确实需要在 URL 里用到它们（譬如，要允许用户搜索任意字符串，并通过查询字符串传给服务器端）。既然不能禁用它们，该标准就提供了一种使用方法，即对这些有问题的字符进行编码。简单来说，这个方法就叫百分号编码或 URL 编码，它以一个百分号（%）和该字符的 ASCII 编码所对应的 2 位十六进制数字去替换这些有问题的字符。例如，“/”会被编码成 %2F（一般用大写，但不强制）。为避免混淆，百分号本身被编码成 %25。中间程序在处理已存在的 URL（如在浏览器或 Web 应用里涉及的 URL）时，绝对不要对这些待中转的 URL 中的保留字符再进行编码或解码，以免无意中改变了这些 URL 的原本含义。

但很遗憾，在碰到那些错误使用了这些保留字符，技术上来说并不合法的 URL 时，浏览器会出各种状况，因此对 URL 保留字符的处理也并非完全一成不变的。由于规范里

⊖ coredump.cx 是作者大人的主页域名，有兴趣的读者可自行访问其个人主页 http://lcamtuf.coredump.cx/ 和个人博客站点：http://lcamtuf.blogspot.com/。而作者大人介绍本书的地址在：http://lcamtuf.coredump.cx/tangled/。——译者注

并没有说到这些情况要怎么处理，导致浏览器开发商们完全是自由发挥，各家的实现并不一致。例如，这个 URL：`http://a@b@c/` 到底应该被编码成：`http://a@b%40c/` 还是 `http://a%40b@c/` 呢？ IE 和 Safari 浏览器认为前者更合理；而其他浏览器则选择支持后一种做法。

不在保留字符集里的字符理应不会对 URL 语法有什么特别重大的影响。然而，有一些字符（如非打印 ASCII 控制字符）明显会影响到 URL 的可读性和传输安全。因此 RFC 规定了一个名字很让人摸不着头脑的所谓非保留字符子集（包括字母数字、“-”、“.”、“_”和“~”）并说只有这个子集里的字符和位于正确功能位置上的保留字符才允许出现在 URL 里。

> **注意**　令人奇怪的是，虽然规定里只允许非保留字符才能以未经编码的形式出现在 URL 里，但却未禁止非保留字符不能以编码形式出现。所以客户端软件可以随意地对它们编码或解码，编解码后 URL 的含义其实并没有变化。这个特点带来了另一个让人非常困惑的现象：在 URL 里可以用非正规的方式表示非保留字符。以下几个例子含义实际上完全一样：
>
> - http://example.com/
> - http://%65xample.%63om/
> - http://%65%78%61%6d%70%6c%65%2e%63%6f%6d/ ⊖

另外还有一些可打印字符原本也应该属于非保留字符之列，却被排除在非保留字符集之外了。因此严格来说，根据 RFC 的规定它们应该被无条件编码。然而，浏览器有时候也并非完全按规则办事的，它们对这条规定并没有很当真。特别是，所有的浏览器都允许在 URL 里出现“^”、“{”、“|”和“}”而且无需被编码，就会直接把这些字符发给服务器端。IE 浏览器更是允许“<”、“>”和“`”，IE、FireFox 和 Chrome 全都接受“\”；Chrome 和 IE 还允许双引号，而 Opera 和 IE 都允许直接传送非打印字符 0x7F (DEL)。

最后，和 RFC 里的明确规定不同，绝大部分浏览器对片段 ID 都不会进行编码。这对依赖于这个字串的客户端脚本会产生难以预期的问题，只能指望一些会带来潜在风险的字符最好永远都不要出现。我们会在第 6 章再讨论这个问题。

对非 US-ASCII 文本字符的处理

全球有许多语种都超出了基础 7 位 ASCII 字符集甚至所有 PC 兼容系统默认使用的 8

⊖ 类似的不规范编码 URL 被广泛用于各种社工攻击，相应地，也出现了各种处理的方式。和往常类似，有些处理方式会影响用户的使用（例如，Firefox 不允许在主机名里出现 URL 编码方式），而有的就比较妥当（如对地址栏里非必要的编码字符进行强制解码，按“规范化”的形式显示出来）。

位代码页（也即 CP437）。甚至有些语言压根就不是使用拉丁体系字母表的。

所以在 Web 出现之前，就产生了为兼容这些常被忽视但有广泛使用基础的非英语用户的各种 8 位代码页（8 bit code page），使用的是各种高位字符集：ISO 8859-1、CP850 和用于西欧语言体系的 Windows 1252；用于东欧和中欧体系的 ISO 8859-2、CP852 和 Windows 1250；用于俄语的 KOI8-R 和 Windows 1251。由于某些语言的单词远非 256 个字符空间能覆盖的，就产生了一些更复杂的宽字节编码方式，如用于日文片假名的 Shift JIS。

不兼容的编码字符集令使用不同代码页的计算机之间交换文件变得很困难。到 20 世纪 90 年代初，这个问题日益严重，因此诞生了 Unicode——这是一种统一的通用编码字符集，但它太大了，8 位编码长度是不可能覆盖全部区域用到的各种字符和形形色色的象形文字的。Unicode 的改进版本是 UTF-8，一种涵盖所有字符但相对简单的可变宽度编码方式，理论上来说，UTF-8 对传统 8 位格式编码的应用是安全的。但 UTF-8 比起其他编码方式，需要用更多字节去编码高位字符，很多用户觉得这有点浪费且没必要。因为这些批评，使得 UTF-8 编码方式在出现 10 年之后才逐渐在 Web 上获得认可，而此时相关的 Web 协议早已尘埃落地固定成型了。

很遗憾，这种滞后对 URL 里的用户输入数据有一定的影响。浏览器需要在很早期就兼容这种用法，但开发人员在相关的协议里寻求帮助时，他们找不到任何有意义的建议。甚至在多年以后的 2005 年，RFC3986 也仅是这么说：

> 有了新的技术手段之后，用户在本地或区域性语言相关的上下文里，允许使用更宽范围内的字符；但本规范对具体怎么使用这些技术未作规定。
>
> 在 URI 里，……如出现超过该 URL 相应规范和协议元素限定范围内的字符，就必须用 8 位元数据的百分号编码形式来表示这些超出 US-ASCII 码表之外的字符。但把 URI 里这些字符编码成百分号形式之前，需要先设定字符的编码方式。

唉，虽然心怀美好的愿望，但这些标准就再未对此问题有进一步描述了。所以尽管的确能在 URL 里传递原始的高位字符，但如果不知道它们是按哪种代码页进行解析的，服务器端就不知道传过来的一个 %B1 究竟代表了“±”还是“a”或是某个乱码字符，这完全取决于用户的使用环境了。

令人痛心的是，浏览器开发商们在这个问题上非常消极被动，并没有想出一个持久统一的解决方案。大多数浏览器都会先把 URL 里的路径信息部分，进行内部转码，变成 UTF-8 格式（甚至如果可能，就只按 ISO 8859-1 格式解析），而查询字符串部分则按照当前页面的编码方式进行转换。在某些情况下，对于手工输入的 URL 或发给特定 API 处理的 URL，高位字符就会被自动降级成 7 位 US-ASCII 码所对应的字符，结果产生一堆的问号形式乱码，甚至由于具体实现上的漏洞，出现完全混乱的结果。

不管浏览器上的具体实现效果如何，总之在查询字符串和路径里传递非英语字符的问题都是一项大大的烦扰。但 URL 里服务器地址的编码却没有碰到这种百分号编码的问题：因为目标服务器的名称里不能包含高位字符，至少理论上符合规范的 DNS 域名只允许出现字母数字和横线，然后用句点作为分隔符号——如果有人不遵守这条规则，出错的具体信息取决于每台域名服务器的返回了，并没有成规。

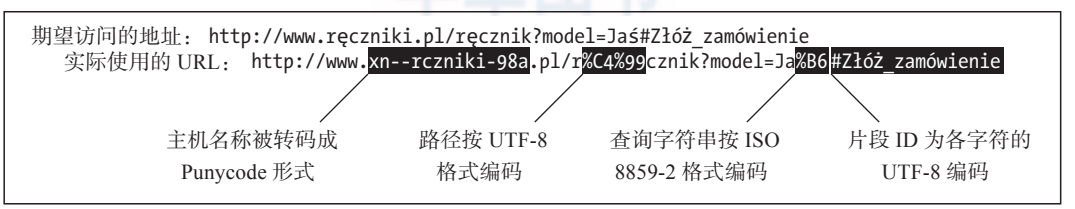
聪明的读者或许要问：为什么会提到这个限制呢？换而言之；真的需要用到非拉丁字母组成的本地化域名吗？要回答这个问题还真有点棘手。其实很简单，从前有些群众认为如果不支持这类编码方式，会降低全球商业用户和个人用户对 Web 的接受度与使用范围——然后，也不管是否的确如此，他们决定支持本地化域名。

这番努力的结果就是诞生了 IDNA（Internationalized Domain Names in Applications，国际化域名）。首先 RFC 3490[9] 描述了一种使用字母数字和横线，很别扭地对任意 Unicode 字符串进行编码的方式，然后 RFC 3492[10] 描述了把这种编码应用在 DNS 域名上的做法，这就是 Punycode 格式。Punycode 的样子大致如下：

```
xn--[US-ASCII 部分 ]-[ 经过编码的 Unicode 数据 ]
```

在兼容这套机制的浏览器的地址栏里，如果输入的主机名中包含了非 US-ASCII 字符，浏览器会先把这种非法字符转换成 Punycode，然后再进行 DNS 解析。同样地，如果在浏览器的地址栏里填入包含 Punycode 的 URL，浏览器应该以解码后的、我们可辨认的形式显示该 URL。

注意　如果把这些各不兼容的编码策略都组合在一起，能获得极富娱乐效果的混搭。下面我们就以这个虚构的波兰毛巾商店为例来看看：



在各种和 URL 编码有关的处理里，IDNA 很快就变成最有问题的一个。因为从根本上来说，显示在浏览器地址栏里的 URL 域名部分是 Web 最重要的安全标记，它能让用户迅速区分出他们信任的并且打过交道的网站和其他陌生的互联网站点。当显示在浏览器里的域名是波兰人民非常熟悉易认的那 38 个的字母⊖时，只有特别粗心的用户才会弄错真域名和假域名。但 IDNA 不加区分地把这 38 个字符扩展成支持 10 多万个象形字符的 Unicode 之后，这些字符再经过编码后就相差无几了，彼此仅有实际功能上的差异而已。

⊖　这里的 38 个字母的说法，是针对作者的母语波兰语而言的。——译者注

事情会有多糟糕？我们以斯拉夫语字母为例，其中一些字母颇有些长得和拉丁语言体系里的一样，但其 Unicode 编码值完全不同，解析后会对应完全不同的 Punycode 域名：

拉丁字母	a	c	e	i	j	o	p	s	x	y
	U+0061	U+0063	U+0065	U+0069	U+006A	U+006F	U+0070	U+0073	U+0078	U+0079
斯拉夫字母	a	c	e	i	j	o	p	s	x	y
	U+0430	U+0441	U+0435	U+0456	U+0458	U+043E	U+0440	U+0455	U+0445	U+0443

在 IDNA 提出这套方案和浏览器刚开始支持这套机制的时候，没人觉得这有什么问题。浏览器开发商们明显假定 DNS 注册机构方面会确保不会有人注册两个很形似的域名，而域名机构却认为该由浏览器开发商负责解决地址栏里非常容易引起误解的视觉效果问题。

到 2002 年，与之有关的各方终于意识到问题的严重性。这一年，Evgeniy Gabrilovich 和 Alex Gontmakher 发表了“The Homograph Attack”[11]，在该论文里探讨了这种漏洞的细节。他们指出，即使在技术上能实现，但任何由注册机构方所做的防范措施，都是有漏洞的。攻击者只需要购买一个正常的顶级域名，然后在自己的域名服务器里设置一条二级域名记录，再经过 IDNA 的转换之后，看上去就和 `example.com/` 完全一模一样（实际上最后的斜杠字符并非功能性的分隔符号，而是一个仅在外观上完全相同的字符而已）。

```
http://example.com .wholesome-domain.com/
```

这个符号看起来就像个真的斜杠似的。

所以域名注册机构干不了什么事情，主动权还是在浏览器开发商手上。但他们究竟能做什么呢？

实际上，开发商们能做的事情也并不多。我们现在意识到，实在很难找到一味既简单又无痛的药方来解决那个非常欠缺远见的 IDNA 标准。浏览器开发商们应对这一风险的做法是，如果用户的本地语言和该域名网页里所用的语种不相符，则浏览器里仍以难懂的 Punycode 方式来显示域名（这导致在浏览他国网站，或使用一台进口的电脑，又或仅仅是电脑的语言配置不对时，会发生浏览出错）；只能在国家相关的顶级域名里，使用 IDNA 方式的域名（这样在 .com 域名和其他较为常用 TLD 里就不能使用国际化的域名了）；或者把那些外观样貌与斜杠、句号、空格符这类字符看起来很相似的“坏”字符放到黑名单里（其实这是徒劳无益的，世上这类字型何其多）。

这些做法极大地阻碍了国际化域名的应用，尽管这个标准现在也只是在苟延残喘，但百足之虫僵而不死，而它为非英语国家的用户提供的帮助其实远比不上它所带来的安全问题。

2.3　常见的 URL 协议及功能

让我们把光怪陆离的 URL 解析先放一边，再次回归本源吧。在本章早些时候，我们就提到过，某些协议可能会带来难以预期的安全问题，因此，任何与用户提交的 URL 打交道的 Web 应用都务必要谨慎小心。为了更清楚地解释这点，很有必要先回顾一下典型的浏览器环境里常见的 URL 协议，基本上可以将其划分为四种类别。

2.3.1　浏览器本身支持、与获取文档相关的协议

这些协议由浏览器内部直接处理，通过特定的传输协议，获得指定文档的内容，并通过常规的内核解析引擎的逻辑处理，把文档最终呈现出来。这也是 URL 最基本和最常用的功能了。

这个最常见类别里的协议却出人意料地很少：http: (RFC 2616) 它是 Web 领域里最主要的传输模式，也是本书下一章的内容；https: (RFC 2818[12]) 加密版的 http；还有 `ftp:` 一个历史相对久远的传输协议（RFC 959[13]）。而所有的浏览器都支持 file: 协议（以前也叫 local:），它用于访问本地文件系统或 NFS 和 SMB 共享（但 file: 协议通常无法从互联网上的网页访问到）。

还有两个不太为人所知的协议也应该简要地说两句：浏览器内置支持的 gopher: 协议，它是一种被淘汰的早期 Web 原型（RFC 1436[14]），但 Firefox 仍然支持这一协议；而 IE 浏览器里则仍然支持 shttp: 这是一个失败的 https 尝试（RFC 2660[15]），但现在它仅仅是作为 HTTP 的一个别名而已。

2.3.2　由第三方应用和插件支持的协议

浏览器碰到这类协议，会根据 URL 的匹配情况，把具体的处理转给外部的某个应用程序，就可以实现类似媒体播放、文档浏览或 IP 电话等功能。到这一阶段，就已经无需浏览器的参与了（大多数情况下如此）。

到今时今日，已经有几十个这类协议处理程序了，要把它们都说清楚只怕还得再写一本厚厚的书。一些最常见的协议包括：`acrobat:`，看字面就知道是用于触发 Adobe Acrobat 的 pdf 阅读器；`callto:` 和 `sip:` 则用于各种即时通信工具和电话软件；`daap:`、`itpc:` 和 `itms:` 用于苹果公司的 iTunes 软件；`mailto:`、`news:` 和 `nntp:` 用于邮件和 Usenet 新闻网客户端软件；`mmst:`、`mmsu:`、`msbd:` 和 `rtsp:` 是流媒体播放器专用的，反正诸如此类还有很多。浏览器自身有时候也在这个列表里。譬如之前提到的 `firefoxurl:` 协议就会在另一个浏览器里启动 Firefox 程序；而 cf: 协议则可以在 IE 里调用 Chrome。

大多数情况下，通常来说 URL 里这些协议对发起它们的 Web 应用自身的安全性并无影响（但这不是绝对的，特别是在一些需要插件支持的内容里）。值得注意的是，这些第

三方协议的处理程序往往漏洞百出，并有可能导致操作系统被入侵。因此，一个可靠的网站应该尽量避免使用那些来历不明的协议。

2.3.3　未封装的伪协议

有一系列内部保留协议是用于方便访问浏览器的脚本解析引擎和某些内部功能的，它们不需要真的去远端获取数据，甚至也不需要创建一个独立的文档上下文环境来展示运行的结果。这种伪协议大多和浏览器紧密相关，一般无法通过互联网访问到，通常也不能干什么大的坏事。然而，这条规律有几个重要的例外。

也许最广为人知的例外就是 `javascript:` 协议了（在更早期的时候，它还有诸如 `livescript:` 或 `mocha:` 这样的别名，后者见于 Netscape 浏览器中）。这个协议可以在当前浏览器访问的网页环境里，调用 JavaScript 编程语言的解析引擎。在 IE 浏览器里，`vbscript:` 协议也有类似的功能，只不过用的是微软专有的 Visual Basic 接口。

另一个重要的例子是 `data:` 协议（RFC 2397[16]）它不需要任何额外的网络请求，就能创建一个短小的内置式文档（inline document），有时候它可以通过继承而获得发起该 URL 的那个页面的某些操作权限。一个 `data:` 协议的例子，其 URL 如下：

```
data:text/plain,Why,%20hello%20there!
```

这些能从外部访问到的伪 URL 对网站的安全有严重影响。一旦被访问到，其所包含的攻击数据（payload）就可以用发起端服务器运行环境的权限，获得执行，最后偷取受影响用户的敏感信息或改变页面的显示内容。我们会在第 6 章时讨论浏览器编程语言的功能，很容易想像得到：它们至关重要（而 URL 运行环境权限的继承规则，会在第 10 章里重点讨论）。

2.3.4　封装过的伪协议

这种特别的伪协议可以放在任意 URL 之前，它指示将取回的内容强制进行特殊的解码或者渲染显示。这一类别里最广为人知的例子就是 Firefox 和 Chrome 支持的 `view-source:` 协议了，它会按照整齐的排版格式显示 HTML 页面的源代码。这个协议的用法如下：

```
view-source:http://www.example.com/
```

其他的类似协议包括 `jar:`，Firefox 可以通过这个协议解压这种类似 ZIP 的文件内容；`wyciwyg:` 和 `view-cache:` 分别为 Firefox 和 Chrome 浏览器访问缓存的方式；`oddballfeed:` 协议用于在 Safari 里获得新闻推送信息[17]；另外还有一堆与 Windows 帮助系统和其他 Windows 组件有关的协议，这类协议的支持文档很差也鲜有资料可查（`hcp:`、`its:`、`mhtml:`、`mk:`、`ms-help:`、`ms-its:` 和 `ms-itss:`）。

许多封装型协议的共同特点是，攻击者通过这种方式，隐藏最后实际是由浏览器处

理的真实 URL 地址，如 view-source:javascript:（或甚至写成这样：view-source:view-source:javascript:），后面再跟一段恶意代码，就能达到目的。可能会有一些安全限制以避免这种漏洞，但通常这类限制都不太能指靠得上。另一个重要的问题是，尤其是在微软的 mhtml: 协议里，它可能会忽略服务器返回的 HTTP 内容设置指令 [18]，而产生广泛的影响。

2.3.5　关于协议检测部分的结语

各种伪协议的数量极多，这也是 Web 应用必须严谨地鉴别用户提供的 URL 到底是什么协议的原因。通常对 URL 的解析模式往往既不靠谱，每种浏览器的处理也不同，再加上浏览器支持的许多协议都有很强的开放性，仅仅把已知的危险协议放进黑名单里并不足够；例如，要检查协议名称是否为 javascript: 其实有各种规避的方法，例如在这个关键字的中间插入一个制表符或一个换行符，或代之以 vbscript:，甚至可以在前面再加上另一层封装形式的协议。

2.4　相对 URL 的解析

在本章之前的篇幅里曾多次提到过相对 URL，它仍然值得我们再额外地说一下。相对 URL 之所以存在，是因为互联网上的每个网页，几乎都不可避免地会引用许多与它同一服务器的其他 URL 地址，甚至多半就与它处于同一个目录。所以，如果这个文件里的每次引用，都要使用完整的 URL 地址，就会非常不方便并且很浪费资源，因此就会用到这种较短的相对 URL 表达方式（如 ../other_file.txt）。相对 URL 里缺失的部分，就由发起引用的那个 URL 自身的信息补齐。

因为相对 URL 的使用场景和绝对 URL 没什么差别，所以浏览器就必须有办法识别出它们的差别。这样对 Web 应用也有好处，因为大多数的 URL 过滤器都只需要对绝对 URL 进行清理，而本地的相对 URL 则可以保持原样。

按照规范里的说法，要区分开两者非常简单：如果 URL 字符串不是以一个有效的协议名开始的，后面没有跟着冒号，又或者没有那个有效的“//”分隔符，那它就是一个需要被引用的相对 URL。如果在解析这个相对 URL 时没有上下文环境，那就拒绝访问。如果有上下文运行环境，那它就必定是个安全的相对链接了，对吧？

很容易想象得到，这不像看上去那么简单轻巧。首先，如上一节里提到过的那样，因为各种浏览器的具体实现千差万别，有效协议名称的字符集也各自不同，然后还有各种替代“//”分隔符的做法。也许最有意思也很常见的一个错误观念，就是认为相对链接只能指向同一个服务器上的资源，但实际上，还是颇有几种不太明显的 URL 变型可以利用的。

让我们看看相对 URL 有哪些类型，以便更进一步阐述各种可能性。

- **有协议名称，但没有授权信息（http:foo.txt）。**这个臭名昭著的漏洞是在 RFC 3986 里埋下的，这要拜更早期规范里的疏忽所赐。这些规范在描述里认为这种 URL 是（无效的）绝对地址，但所提供的解析算法却又含混地把这种地址的解析给搞错了。

  在后来的地址解析算法里，对于这种形式的 URL，它的协议、路径、查询字符串或片段 ID 都以这个 URL 自身为准，但授权信息的部分，以引用它的那个页面地址为准。许多浏览器都按照这个标准执行，但又执行得并非很统一。譬如，在某些情况下，`http:foo.txt` 会被当作一个相对地址，而 `https:example.com` 却会被解析成绝对地址！

- **没有协议名，但有授权信息（//example.com）。**这又是另一种臭名昭著令人困惑的写法，但对怎么处理这种格式的 URL 规范里倒是写得比较完整。尽管原本对于 `/example.com` 这种写法，浏览器会指向当前服务器的本地资源，对于 `//example.com` 这种写法，浏览器依据规范却做了完全不同的处理：保持当前页面的协议不变，然后把这串字符作为一个新的授权信息段。在这种情况下，协议名称由原发起页面确定，而所有接下来的 URL 信息都取自这个相对 URL，构成完整的 URL。
- **没有协议名、没有授权信息，但有路径（../notes.txt）。**这是相对链接里最常见的一种形态了。协议和授权信息都从引用 URL 里复制过来。如果相对 URL 的开头不是斜杠[⊖]，则相对路径会拼接在引用 URL 最右边的“/”后面。例如，如果基础 URL 为 `http://www.example.com/files/`，则路径不变，相对路径直接跟在其后；但如果初始路径是 `http://www.example.com/files/index.html`，则需要把文件名的部分砍掉，再把相对路径接上去。新的路径加在原路径后，拼接起来的路径名再经过标准化的路径规范整理。而查询字符串和片段 ID 都只来自相对 URL。
- **没有协议名、没有授权信息、没有路径，但有查询字符串（?search=bunnies）。**这种情况下，协议、授权信息、路径信息全都原封不动地从原引用 URL 复制过来。查询字符串和片段 ID 则来自相对 URL。
- **只有片段 ID（#bunnies）。**除片段 ID 之外，所有信息全都原封不动地从原引用 URL 复制过来；只替换片段 ID 的部分。如之前章节讨论过的那样，点击这类相对 URL 通常不会导致页面的重新加载。

  因为应用级别的 URL 过滤和浏览器本身的可能会有差异，在处理这种相对地址时，不直接输出任何由用户提供的相对 URL 是靠谱的做法。如果可行，相对地址应该直接改写成绝对地址，而所有的安全检查应针对经过以上调整后符合规范的 URL 地址。

⊖ 补充一句，作者隐含没说的部分是：如果相对 URL 的开头的确是个斜杠，则应该是忽略原引用页面自身的路径信息，直接把相对路径拼在引用 URL 的授权信息部分之后。——译者注

2.5　安全工程速查表

构建基于用户输入的全新 URL 时

- ☑ **如果允许用户构造 URL 的路径、查询或片段 ID 这几部分**：如果其中一部分未能正确地进行转义，这种 URL 很可能会产生出人意料的结果（例如，假造一个用户可见的 HTML 按钮，令它和一个服务器端的处理关联起来）。这里宁可选择保守的做法：如果需要插入攻击者能控制的字段值，除了字母和数字，最好把所有的数据都按百分号编码形式转义一下。
- ☑ **如果允许用户提供协议名称或授权信息部分数据**：这会带来严重的代码注入和钓鱼风险！请采用以下列出的输入验证。

设计 URL 输入过滤器

- ☑ **相对 URL**：禁用相对 URL，或明确地改写成绝对路径以避免麻烦。其他方式都很可能有风险。
- ☑ **协议名称**：只允许出现可接受的几种协议类型，例如 `http://`、`https://` 和 `ftp://`。不要使用黑名单的验证方式，这是非常不安全的。
- ☑ **授权信息部分**：主机名部分应该只包含数字字母以及“-”和“.”，在它的后面只能跟着“/”、“?”、“#”或字符串结束符。允许其他任何数据都有可能带来风险。如果需要检查主机名，请确保合理地使用从右往左的截取匹配模式。

  在很罕见的情况下，可能还需要检查 URL 里的 IDNA、IPv6 括号表示法、端口号或 HTTP 验证授权信息等。如果确实需要，必须对整个 URL 进行彻底的解析，验证各个部分，拒绝不正常的值，重新把整个 URL 序列化成一个没有歧义，符合规范，经过恰当转义的表达形式。

需要从接收到的 URL 里进行参数解码

- ☑ 千万不要因为标准是这么说的，浏览器也理应是这么做的，就想当然地认为特定的字符就一定会被转义。在返回任何从 URL 获取的数值或把它放入数据库进行查询前、拼接成新的 URL 前，都需要仔细严谨地过滤清理危险的字符。

第3章

HTTP 协议

我们要讨论的下一个基本概念是超文本传输协议（Hypertext Transfer Protocol，HTTP）：它是 Web 的核心传输机制，也是服务器与客户端之间交换 URL 引用文档的首选方式。尽管名字里包含了超文本这几个字，但 HTTP 和真正的超文本内容（HTML 语言）其实彼此独立。当然在某些时候，它们会以一种令人意想不到的方式交织在一起。

HTTP 的历史很有意思地展示了其创立者的抱负和 Internet 的发展。Tim Berners-Lee 在 1991 年发表的最初版本的协议草案（HTTP/0.9[1]）只有不足一页半的内容。它甚至没有包含一些在日后明显非常必要的需求，如对传输非 HTML 类型数据的扩展支持。

在历经 5 年的时间和若干版本的更替后，第一个正式版 HTTP/1.0 标准（RFC 1945[2]）诞生了，这时它已经变成一份密密麻麻长达 50 页的文档，期望能弥补过往标准里的诸多缺陷。很快到了 1999 年，HTTP/1.1（RFC 2616[3]）的 7 位署名作者显然指望该协议能涵盖到方方面面，导致的结果就是一份长达 150 页的大作。这还不算，目前在制定中的 HTTPbis[4]，也即 HTTP/1.1 规范的替代品，恐怕会长达 360 页。但这些越来越宏伟的文档里很大篇幅的内容，和我们当下实际在用的 Web 并非特别相关，因为它们对新功能的追逐远比修复旧有缺陷更感兴趣。

现在所有的客户端和服务器端都兼容 HTTP/1.0 协议（但这种兼容又没有完全照搬 HTTP/1.0 协议，而是或多或少有些变化）；除了在某些扩展属性上有例外的情况，大多数

产品都较为合理地支持完整的 HTTP/1.1 协议。尽管没什么实际意义，但部分 Web 服务器和所有常见的浏览器，也向下兼容 HTTP/0.9 协议。

3.1　HTTP 基本语法

如果只是匆匆一瞥，HTTP 相当简单。HTTP 协议建立在 TCP/IP[㊀]传输方式之上，基于文本方式进行交互。每个 HTTP 会话会先与服务器端建立一个 TCP 连接，通常连接 80 端口，然后对要获取的 URL 发出具体的请求信息。而在响应时，服务器端会返回被请求的文件内容。通常情况下，服务器端随后会断开 TCP 连接。

初始版本的 HTTP/0.9 协议完全没有为客户端和服务器提供任何额外的元数据交换空间。它的客户端请求仅有一行，以 GET 开头，后面跟着 URL 路径和查询字符串，然后以一个 CRLF 换行符结束（对应的 ASCII 码为 0x0D 0x0A；按照规范如果结尾处只有一个 LF 字符，服务器也会接受）。一个 HTTP/0.9 请求样例如下：

```
GET /fuzzy_bunnies.txt
```

服务器对这条请求的回应，是立刻返回它需要的 HTML 数据（按照规范，服务器应该按每行 80 个字符换行的方式来返回响应的内容，但基本上没人遵守该条建议）。

HTTP/0.9 的处理有一堆严重缺陷。例如，没法根据浏览器里用户的首选语言或支持的文档类型等进行交互处理。当找不到请求的文件需要被重定向到新的位置，或返回的内容不是 HTML 格式的文件时，在 HTTP/0.9 协议里服务器也没法通知到客户端。最后，这个协议对网站管理员也非常不友好：因为传输的 URL 信息里只包括了路径和请求行，所以不能做到在一台服务器上用一个 IP 地址支持多个不同主机名的网站——有别于域名，IP 地址可是比较珍贵的资源。

为修正这些缺点（也为了日后有更多的改进余地），HTTP/1.0 和 HTTP/1.1 的格式就略有变化了：请求的第一行包含具体的版本信息，后面则跟着零到多条以“名称 : 值”（name:value）格式组成的数据行（也叫头域，Headers），每行为一个头域。常见的请求头包括：`User-Agent`（浏览器版本信息）、`Host`（URL 主机名称）、`Accept`（支持的 MIME 文件类型[㊁]）、

㊀ 传输控制协议（Transmission Control Protocol，TCP）是互联网上最核心的通信协议，其上的各种应用层协议都是基于 TCP 传输方式的。TCP 能提供相对可靠、点应答（peer-acknowledged）、有序和基于会话的连接方式。在大多数情况下，它对互联网上非本地主机发起的猜测型 IP 数据包欺骗（blind packet spoofing）有一定的抵抗能力。

㊁ MIME Type（也叫互联网多媒体类型），简单地用前后两个标识符来确定任意计算机文件的类型和格式的方法。这个概念起源于 RFC 2045 和 RFC 2046，在这 2 个文档中，MIME Type 概念用于描述电子邮件附件的格式。正式的 MIME Type 注册名格式（如 Text/Plain 或 Audio/MPEG）列表目前由 IANA 维护管理，当然自定义的 MIME Type 类型也相当常见。

Accept-Language（支持的语言编码）和 Referer[⊖]（这个字其实拼错了，它的含义是，如果有的话，显示发起当前请求的原始页面）。

请求头[⊜]的部分最后以一个单独的空行结束，这个空行后面可以再跟上客户端希望发送给服务器的任何数据（这些数据的长度必须以 Content-Length 请求头明确标识）。HTTP 协议本身对这段数据的内容格式其实并未做任何规定；在 HTML 页面里，这个位置通常用做表单数据传递。传递的方式有好几种可能的格式，当然这些都不是硬性的规定。

总的来说，一个简单的 HTTP/1.1 请求如下：

```
POST /fuzzy_bunnies/bunny_dispenser.php HTTP/1.1
Host: www.fuzzybunnies.com
User-Agent: Bunny-Browser/1.7
Content-Type: text/plain
Content-Length: 17
Referer: http://www.fuzzybunnies.com/main.html

I REQUEST A BUNNY
```

服务器对这个请求的响应是，第一行包含支持的协议版本和一个数字格式的状态码（用以显示出错状况和其他特殊情况），还有可选的易于理解的状态信息描述。紧随其后的是若干一看自明的响应头，响应头部分以一个空行结束。响应数据的最后再跟着请求资源的具体内容。

```
HTTP/1.1 200 OK
Server: Bunny-Server/0.9.2
Content-Type: text/plain
Connection: close

BUNNY WISH HAS BEEN GRANTED
```

RFC 2616 还规定，除非客户端明确地发过来一个 Accept-Encoding 头，在里面设定了能接受的传输压缩方式，否则服务器端可以在以下三种压缩方式里（gzip、compress 和 deflate）任选其一，用于传输响应数据。

3.1.1　支持 HTTP/0.9 的恶果

HTTP/1.0 和 HTTP/1.1 协议已经做了很多改进，而大家对古老“愚笨”的 HTTP/0.9

⊖ 其实这个单词原本应该是 Referrer，但是 Phillip Hallam-Baker 在起草 RFC 1945 时拼错了，后来就一直将错就错地沿用下来，变成现在的 Referer 了。——译者注

⊜ 许多原文为 Header 的地方，译文里可能会根据实际情况，改为更明确的“请求头”或“响应头”称谓，在没有明确特指的情况下，则译为“头域”。——译者注

协议都很不待见，我们平时也根本意识不到它的存在，但它却并未完全退出历史舞台。部分原因要拜 HTTP/1.0 规范所赐，因为该规范要求以后所有的 HTTP 客户端和服务器都需要支持这个半成品级别的草案。特别是，在第 3.1 节里规定如下：

> HTTP/1.0 客户端必须……能够理解任何有效的 HTTP/0.9 或 HTTP/1.0 格式的响应。

到最近几年，RFC 2616 对这项要求也颇有悔意（第 19.6 节说：“协议规范并不强制要求兼容更早期的版本”），但因为那个更早期的建议，我们现在用的各种浏览器仍然继续支持这个古老的协议。

要理解这个模式为什么有危险，让我们来回顾一下 HTTP/0.9 协议：它返回的内容只有请求的文件本身。没法从这些回应的内容里，表明响应方确实是理解 HTTP 协议的，以及返回的内容是否真的为 HTML 文件。谨记这一点，让我们来分析一下，如果服务器 example.com 的 25 端口有上运行着一个不做任何检查的 SMTP 服务，浏览器向这个端口发送了 HTTP/1.1 请求会有什么效果：

```
GET /<html><body><h1>Hi! HTTP/1.1
Host: example.com:25
...
```

因为 SMTP 服务器不理解这个请求是什么意思，它很可能这么回应：

```
220 example.com ESMTP
500 5.5.1 Invalid command: "GET /<html><body><h1>Hi! HTTP/1.1"
500 5.1.1 Invalid command: "Host: example.com:25"
...
421 4.4.1 Timeout
```

只要是遵守 RFC 协议的浏览器，都会被迫接受返回的信息，认为这些就是有效的 HTTP/0.9 响应内容，并把返回的文档按 HTML 格式解析。所以浏览器会把上面那些由攻击者控制，出现在返回错误信息里的代码，解析为 example.com 网站返回的合法网页内容。与浏览器安全模型的深层交互会在本书第二部分再详加讨论，总之，后果可以很严重。

3.1.2　换行处理带来的各种混乱

先把 HTTP/0.9 和 HTTP/1.0 的巨大差异放到一边不说，这些版本的 HTTP 协议还加了几项核心的语法调整。最著名的可能是，有别于之前的迭代版本，HTTP/1.1 要求客户端不仅要接受 CRLF 和 LF 格式的换行符，还要接受单个的 CR 字符。 尽管最常见的两种 Web 服务器（IIS 和 Apache）都不接受 RFC 规范里这条建议，但除 Firefox 以外所有的客户端

浏览器都遵从这一规定。

这种不一致性使得开发人员在处理 HTTP 头域里任何攻击者控制内容时，不仅要过滤 LF 字符，其实还要过滤 CR 字符。要说清楚这个问题，以下面这个服务器响应为例，其中加粗部分就是出现在响应头里未经足够过滤的用户提供内容：

```
HTTP/1.1 200 OK[CR][LF]
Set-Cookie: last_search_term=[CR][CR]<html><body><h1>Hi![CR][LF]
[CR][LF]
动作完成。
```

对 IE 浏览器来说，这个响应看起来类似：

```
HTTP/1.1 200 OK
Set-Cookie: last_search_term=

<html><body><h1>Hi!

动作完成。
```

实际上，这类在 HTTP 头域里夹带换行符的漏洞，不管是因为数据解析的不一致性还是由于没有正确过滤各种类型换行符而产生，都实在是太常见了，所以这类攻击还有专门的名字：头域注入（Header Injection）或者响应拆分（Response Splitting)。

另一个有潜在安全风险但较不为人知的问题是：由于 HTTP/1.1 协议里的规定，所以实际上 HTTP 协议支持多行请求头，它规定任何一个以空格开头的行，都是接续在前一行之后的内容。例如：

```
X-Random-Comment: 这是个长句子，
  所以我们得换行处理一下，看着更整齐。
```

IIS 和 Apache 都接受由客户端程序发出的多行格式请求头，但实际上，IE、Safari 或 Opera 都不支持发出这种请求头格式。在攻击者控制的环境里，某些具体实现如果必须采用多行标头，或者允许采用多行标头那就可能导致问题。但谢天谢地，这种攻击能发挥作用的地方极少。

3.1.3　经过代理的 HTTP 请求

许多机构和 ISP 供应商都会利用 HTTP 代理，以其用户的名义提供对 HTTP 请求进行拦截、检查和转发等功能。使用代理的原因可能是为了提高性能（先把某些服务器的响应缓存到就近的系统）、或强制应用某些网络访问策略（如禁止访问某些色情站点）或需要以代理方式实现某些独立网络环境的监控和需要授权的接入。

常规情况下 HTTP 代理需要浏览器的支持：浏览器要先修改配置，要求把经过修改的

请求先发往代理服务器，而不是直接和目标地址进行连接。要使用代理方式来获取一个 HTTP 资源，浏览器通常需要发送如下请求：

```
GET http://www.fuzzybunnies.com/ HTTP/1.1
User-Agent: Bunny-Browser/1.7
Host: www.fuzzybunnies.com
...
```

上述例子和普通的 HTTP 请求最大的语法差异，就是请求内容里的第一行，这时候是一个完整的 URL（`http://www.fuzzybunnies.com/`），通过这项信息代理服务器才知道用户要连接的目标服务器在哪里。这项信息实际上有点多余，因为在 `Host` 请求头里也标识了主机名称；这种重复是因为这两套机制其实是相互独立发展起来的。为了避免客户端和服务器串通一气，如果 `Host` 请求头的信息与请求行里的 URL 不匹配时，代理服务器应以请求行里的 URL 为准，或者用特定的“URL-Host”数据对和缓存内容关联起来，而不能只根据其中一项信息做出判断。

许多 HTTP 代理服务器还允许浏览器获取非 HTTP 类型的资源，如 FTP 文件或目录。这种情况下，在把数据返回给用户之前，代理服务器会把 HTTP 响应里返回的内容先封装整理一下，可能会先把它们转化成 HTML 格式[⊖]。也就是说，如果代理服务器不理解所请求的协议，或者代理服务器不便于查看交换的数据时（例如，处理的是加密会话），就必须另作处理了。`Connect` 请求方法就是为这些特殊的连接而准备的，但这种请求方式在 HTTP/1.1 RFC 里并没有更进一步的描述。而与之相关的请求语法却放在另一份 1998 年草拟的独立规定里 [5]。CONNECT 方法的格式如下：

```
CONNECT www.fuzzybunnies.com:1234 HTTP/1.1
User-Agent: Bunny-Browser/1.7
...
```

如果代理服务器同意并且能够和请求的目的端地址连接上，它会为这个请求返回一个特定的 HTTP 响应码，`Connect` 协议的工作也就告一段落了。这时候浏览器开始建立 TCP 流，直接向代理服务器发送和接收原始二进制数据；相应地，代理服务器也会不加选择地直接转发两个端点之间的数据流。

注意　滑稽的是，这个规范里有一个很微妙的疏忽，许多浏览器在使用加密连接时，会错误地处理从代理服务器自身返回的非加密错误响应信息。浏览器错误地认为，这些纯文本格式的响应就是从目的服务器的加密通道返回的。由于这个小疏漏，原本在 Web 上使用加密通信应有的保障，就全被破坏掉了。这个问题 [6]

⊖ 这种情况下，一些由客户端提供的 HTTP 请求头就只供代理服务器内部使用了，而不会再被发往非 HTTP 协议的另一端，这会导致一些未必与安全相关但颇为有趣的协议含糊问题。

存在长达 10 年之久才被发现和纠正[⊖]。

还有一些不使用 HTTP 方式和浏览器打交道的更为底层的代理形式，这些代理为了缓存内容或强制执行某些规则，也会需要检查 HTTP 的信息交换。一个经典的例子就是透明代理，它默默地在 TCP/IP 层拦截数据流量。透明代理所用的方式一般都有缺陷：代理服务器能看到拦截的请求里目标端 IP 和主机头 `Host` 信息，但没有办法立刻确认，要连接的目标端 IP 是否真的和 Host 里设定的服务器名称相匹配。除非额外先做一次查询，确定两者是否真的相关，否则两边串通一气的客户端和服务器就有机可乘了。如果不先做额外的检查，攻击者只需要请求连接自己家中的服务器，但发送的是一个故意误导代理的 `Host` 请求头：`www.google.com`，这样其他的确想访问 `www.google.com` 的用户，获得的可能就是被错误缓存的响应内容了[⊜]。

3.1.4　对重复或有冲突的头域的解析

尽管 RFC 2616 已经写得相对详尽，但对一个符合要求的解析器要怎么解析请求或响应数据里含糊和有冲突的数据，并没有做更详细的描述。这个 RFC 文档的第 19.2 节“宽容的应用程序”里建议在某些“无歧义”的场景里对某些特定的值，可采用较为宽松和对错误较为容忍的解析方式，但这个用词本身，就很“有歧义”啊。

例如，因为规范里没有给出建议，大约一半的浏览器会以 HTTP 头域里第一次出现的值为准，而另一半则以最后一次为准，这使得每种和头域相关的注入漏洞，不管怎么限制，总有另一部分的用户会遭殃。而在服务器端，也是一样地随机：Apache 认可第一个 `Host` 头的值，而 IIS 则完全不接受多个 `Host` 头的情况。

与之类似的情况，相关的 RFC 文档也没有明确规定要禁止可能有冲突的 HTTP/1.0 和 HTTP/1.1 请求头，也没有要求 HTTP/1.0 版本的服务器或客户端程序必须忽略所有 HTTP/1.1 的语法。因为这种设计，很难预料如果同时具有 HTTP/1.0 和 HTTP/1.1 两个版本的不同名称但一样功能的头域时，会产生什么后果，例如 `Expires` 和 `Cache-Control`。

最后，要如何解析某些较为罕见的头域冲突，在规范里的描述反而非常清楚，但为什么要允许这些奇怪的冲突也令人费解。例如，HTTP/1.1 客户端在每个请求里，都要发送 Host 主机头，但也规定了服务器（并非只有代理服务器）端必须能识别在 HTTP 请求第一行里的绝对 URL 路径，而常规的 HTTP 请求第一行里只有路径和查询字符串等相关参数。这让人很好奇要如何解析以下请求：

⊖ 作者对“这个问题”只用寥寥几句略作描述，单是靠这段文字实际上很难理解这个漏洞，强烈推荐读者参阅注释，在链接的页面里获得更有帮助的信息，加深对此问题及其严重性的认识。——译者注

⊜ 这里介绍的就是传说中的 Cache Poison（缓存污染）攻击。——译者注

```
GET http://www.fuzzybunnies.com/ HTTP/1.1
Host: www.bunnyoutlet.com
```

在这种情况下，RFC 2616 的 5.2 节要求客户端不要理会非功能性的 Host 请求头（但这个请求头都必须存在！），大多数客户端程序也遵从这条建议。但问题是，某些底层应用可能不太清楚这条吊诡的要求，而选择根据请求头里的 Host 头来做判断。

注意　当我们抱怨 HTTP RFC 文档的各种疏漏时，也得承认即使有额外的规范，也同样会面临各种问题。比如在 RFC 描述某些场景时，希望对一些很罕见的情况也进行明确的强制性处理，这会导致在模式解析上变得非常荒谬。例如 RFC 1945 的 3.3 节，就对某些 HTTP 头域里的时间日期解析提出了建议。这导致为实现这个建议，Firefor 代码库[7] 里的 prtime.c 文件长达 2000 行，这段 C 代码既难懂又难读，而目标仅仅是为了实现足够的容错，以便解析各种日期、时间和时区格式（用于确定缓存内容是否过期）。

3.1.5　以分号作分隔符的头域值

有一些 HTTP 头域，例如 Cache-Control 或 Content-Disposition，使用分号来分隔在同一行里的几对独立的“名称 = 值”数据组。允许使用这种嵌套式语法的原因，人们可能认为这种方式更有效率或者更直观一些，否则就要用到好几个独立的 HTTP 头域了。

RFC 2616 规定在某些使用场景里，这种数据组对里等号右边的参数值可以使用 Quoted-String 格式。Quoted-String 指的是用双引号作为分隔符，由任意可打印字符组成，两边用双引号括起来的字符串。当然了，缺点是引号本身不能包括在字符串里，但好处是支持分号和空格，使得某些不加引号可能会有问题的字符串仍能保持原样传输。

但很遗憾，对开发者来说，IE 浏览器对 Quoted-String 的语法支持得不是很好，直接导致这种编码策略完全派不上用场。IE 浏览器对以下数据的解析完全出人意料（这行的本意是表明当前返回的是一个要下载的文件，而不是直接显示在线阅读的文档）：

```
Content-Disposition: attachment; filename="evil_file.exe;.txt"
```

在微软的实现里，文件名会在有分号的地方被截断，导致文件最后被存为 evil_file.exe。应用程序在处理这样的文件名时，必须要专门检查整个字符串中的双引号和换行符，否则仅仅去检测后缀名是否“安全”或人为添加一个“安全”的后缀名，都避免不了风险隐患。

注意　还有一种转义处理机制叫 Quoted-Pair，它是在字符前加一个反斜杠进行转义，以允许在引号括起来的字符串里出现的引号（或其他任何字符）能通

过这种转义而正常使用。但其实规范里错误地使用了这个编码机制，所以主流的浏览器里只有 Opera 支持这种做法。要使 Quoted-Pair 转义机制能正常工作，Quoted-String 格式里就不能出现单个的“\”字符，但这又违背了 RFC 2616 的规定。Quoted-Pair 格式还能转义任何字符型的标记（Char-type token），如“\n”这样的换行标记，这也违背了其他 HTTP 解析规则。

另外要值得注意的是，在同一个 HTTP 头域里，以分号分隔的两个字段，如果名称完全一样，那应该如何处理，RFC 里并没有规定这种情况下的解析顺序。以 Content-Disposition 里的 filename= 为例，所有的主流浏览器都以第一次出现的值为准。但其他情况的处理就未必这么统一了。例如，要解析 Refresh 响应头（用于强制在若干时间内重定向到新页面）里的 URL= 值时，IE6 以最后一个值为准，而其他浏览器都取第一个值。而处理 Content-Type 响应头域时，Internet Explorer、Safari 和 Opera 都会采用第一次出现的 charset= 数值，而 Firefox 和 Chrome 则会取最后一次。

注意　有篇文章虽然与安全的主题基本无关，但非常精彩丰富，它提供了各浏览器仅在处理 Content-Disposition 这一个 HTTP 响应头时的各种情况，每种浏览器之间又各有差异，请参见由 Julian Reschke 整理维护的这个页面：http://greenbytes.de/tech/tc2231/。

3.1.6　头域里的字符集和编码策略

和 URL 处理的相关规范类似，HTTP 规范里也基本回避了在头域的值里出现非 US-ASCII 字符时要如何处理的问题。某些含糊的场景里也许可以出现合法的非英文字符（例如，在 Content-Disposition 的文件名 filename 项里），但要真碰到这些情况浏览器要如何应对却没有什么明确规定。

最初，RFC 1945 允许 TEXT 标记中包含 8 比特字符，并对 TEXT 的类型给出了如下定义（TEXT 标记是广泛用于定义其他字段语法的基元）：

OCTET：任意的 8 比特序列字符
CTL：US-ASCII 码表里的所有控制字符，包括八进制的 0 – 31 和 DEL (127)
TEXT：就是除 CTL 控制字符之外的所有 OCTET 字符，但还包括 LWS 字符[⊖]

但之后 RFC 又提出了一个很古怪的建议：如果在 TEXT 的字段里碰到非 US-ASCII 的字符，客户端和服务器可以按 ISO-8859-1 格式（也即标准的西欧语言代码页）进行解析，但这条无需强制执行。然后，RFC 2616 复制黏贴了这段关于 TEXT 类型的模式规

⊖ LWS（Linear Whitespace）直译为“线性空格”，意即“1 到多个换行符后面跟 1 到多个空格或 TAB 字符”。——译者注

范，但加了个备注，说非 ISO-8859-1 的字符串必须按照 RFC 2047[8] 规范里的格式先进行编码，而 RFC 2047 原本是用于电子邮件通信的。这下热闹了；在这个简单的协议里，要编码的字符串得先以“=?”做前缀，然后跟上具体编码字符集的名称，再跟上一个“?q?”或“?b?”，标识其所用的编码方式（两者分别代表 Quoted-Printable ㊀或 Base64 编码方式㊁），然后才是经过编码后的字符串内容。这串信息的最后还要再跟上一个“?=”作为结束符。举例如下：

```
Content-Disposition: attachment; filename="=?utf-8?q?Hi=21.txt?="
```

注意　RFC 理应规定在相关的头域里不能出现任何假冒的“=?...?=”信息，以避免对实际上压根没经过编码的数据进行完全不必要的解码处理。

遗憾的是，对 RFC 2047 规定的编码方式的支持寥寥可数。在 Firefox 和 Chrome 的某些头域里能识别这种编码方式，但其他浏览器就完全不配合了。IE 浏览器能识别 Content-Disposition 域里 URL 风格的百分号编码方式（Chrome 也采用这种方式），并且默认为 UTF-8 格式编码。而另一方面 Firefox 和 Opera 则选择支持 RFC 2231[9] 里提出的一种很特别的百分号编码语法，和 HTTP 常规语法相比，这变化实在有点大：

```
Content-Disposition: attachment; filename*=utf-8'en-us'Hi%21.txt
```

聪明的读者可能会留意到，没有任何一套编码方式能同时为所有浏览器支持。所以一些网站应用开发人员就干脆直接在 HTTP 头域里使用高位字节数据，一般为 UTF-8 编码格式，但这么做在一定程度上也不太安全。例如 Firefox 里，如果在 Cookie 头域里存放 UTF-8 文本的话，长期以来就有乱码的问题，这会导致如果攻击者在 Cookie 里进行注入攻击，可能会出现难以预料的分隔符问题[10]。总而言之，对这个麻烦我们目前还没有简单可靠的解决方案。

说到字符编码问题，对 NUL 空字符（0x00）的处理可能也需要提一下。这个字符在很多编程语言里都作为字符串结束的标志，理论上来说，它就不应该出现在任何 HTTP 头域里（除非出现在之前提到的不太好用的 Quoted-Pair 语法环境里），但我们也要记住，解析器总是尽量容错的。如果允许出现这个字符，它很可能会产生意想不到的副作用。例如，Content-Disposition 请求头里，如果出现 NUL 字符，IE 浏览器、Firefox 和 Chrome

---

㊀ Quoted-Printable 是一种简单的编码策略，碰到不可打印字符和非法字符时，就把这个字符转换成一个等于号（=）后面跟该 8 比特字符的 2 位十六进制编码。在输入的文本里，任何单独的等号则编码为“=3D”。

㊁ Base64 是一种无法直接阅读的编码方式，它把任意 8 位输入编码为 6 位的内容，这些内容由区分大小写的字母数字、“+”和“/”组成。每 3 个字节的输入会映射成 4 个字节的输出。如果输入不足 3 个字节的长度，则在输出字符串的后面还需要再添加 1 ~ 2 个等号。

都会做出字符截断的处理，而 Opera 和 Safari 则不会这么做。

3.1.7　Referer 头域的表现

本章早前提到，HTTP 请求里可以包含 Referer（来源）头域。这个头域里包含的是哪个 URL 地址触发了对当前页面的访问。Referer 头域对某些纠错处理颇有帮助，还可以通过这个头域，强化网页之间的引用关系，有助于 Web 的发展壮大。

但遗憾的是，这个头域也可能向某些心怀恶意的家伙泄漏用户的浏览习惯，还可能暴露引用页面的查询字符串参数，这些经过 URL 编码的参数里面有可能包含敏感信息。因为担心这个问题，同时缺乏解决这个问题的有效建议，这个头域经常在安全控制或策略强制时被错误使用，但实际上它承担不了这种重任。主要是因为我们没办法主动区分以下几种缺乏 Referer 头域的情况：客户端根据用户的隐私策略设置，直接就没有发出这个头域；用户的浏览行为确实没有涉及这个头域；该请求的来源是恶意站点，它刻意屏蔽了这项信息。

正常来说，这个头域在大部分 HTTP 请求里都会出现（在 HTTP 级别的跳转里也会保留下来），但以下情况除外：

- 刻意在地址栏里直接输入一个新的 URL 或从书签里打开网页
- 浏览的动作是由一个伪 URL 文档，如 data: 或 javascript: 触发的
- 当前请求来自 Refresh 响应头（并非基于 Location 的重定向方式）
- 当来源站点是加密站点而请求的是非加密页面时。根据 RFC 2616 的 15.1.2 节规定，这是出于隐私保护的考虑，但这样做其实没太大意义。当用户从加密的域名浏览到另一个无关的加密网站时，Referer 信息还是会泄漏给第三方，所以加密也并不等于安全啊。
- 用户可以通过调整浏览器设置或安装隐私保护插件，选择不发送或干脆伪造一个来源页面。

很明显，以上 5 种情况里有 4 种情况都可能是恶意站点蓄意诱发的。

3.2　HTTP 请求类型

初始的 HTTP/0.9 草案里对请求的文档只提供了唯一的方法（或叫“动作 verb”）：GET。在它之后曾冒出来一批荒谬怪诞的方法提议，像什么 SHOWMETHOD、CHECKOUT，还有个什么 SPACEJUMP[11] 呢。

大多数这类实验性的构想在 HTTP/1.1 里都被抛弃了，HTTP/1.1 里只有 8 种便于管理的方法。实际上只有前 2 种请求类型：GET 和 POST 对我们现在的大多数 Web 应用是比较重要的。

3.2.1　GET

GET 方法对信息的获取至关重要。实际上，所有常规浏览会话在“客户端 – 服务器”交互时都在使用 GET 方法。一般来说 GET 请求不会携带从浏览器端提交的 Payload 数据，尽管这也并非绝对禁止的。

根据 RFC 的规定，GET 请求“除了索取信息之外，不应该承担其他的重要功能”（也就是说，它们应该不能永久性地改变应用的状态）。这个需求在现在的 Web 应用里已经越来越没有意义了，因为现在甚至服务器端也没法决定应用的状态了；因此，这个建议也越来越为开发人员所忽略[⊖]。

注意　在 HTTP/1.1 协议里，客户端使用 GET 方法时，可以通过 Range 请求头域向服务器端请求任意非连续或有重叠的数据片段（虽然比较少见，但其他的请求方法也可能会用到这个请求头）。服务器并非一定会支持这个功能，但如果支持的话，浏览器就可以通过这个方式，实现断点续传方式的下载。

3.2.2　POST

POST 方法原来的目标是把客户端提交的信息（主要是 HTML 表单数据）传递给服务器端。因为 POST 动作更有可能带来持久性的副作用，很多浏览器在重新加载包含 POST 方式的数据时，浏览器都会谨慎地向用户再确认一次，但大多数情况下，GET 和 POST 的交互机制是比较相近的。

POST 请求通常都会带有表单数据，这段数据的长度要明确地设置在 Content-Length 请求头里。在 HTML 格式的情况下，这些提交的数据通常是经过 URL 编码或 MIME 编码的表单数据（这种格式会在第 4 章里再详细介绍），但要再说一遍，在 HTTP 协议里对这些数据的语法格式其实并没有具体的限制。

3.2.3　HEAD

HEAD 是一种很少用到的方法，在本质上 HEAD 近似于 GET 方法，但它只返回响应头域的部分，不包含具体的数据，即响应内容的部分。一般来说无法通过浏览器直接发出 HEAD 请求，但有时候搜索引擎的爬虫或其他自动化测试工具会用到这个方法，例如探测文件是否存在或检查它的最后修改时间。

⊖ 有这么件传闻轶事（但很可能是真事），说的是一位名叫 John Breckman 的网站管理员的悲剧。故事里说，John 的网站无意中被搜索引擎的爬虫全删了。仅仅是因为这个爬虫无意中访问到一个不需要授权的页面，里面是 John 自己做的一个基于 GET 方法的管理页面……爬虫就高高兴兴地把页面里所有的“删除”链接都爬了一遍，然后？然后就没有然后了。

3.2.4　OPTIONS

OPTIONS 是一种元数据请求（metarequest），服务器会根据客户端请求的 URL 地址（如果 URL 设置为“*”，则泛指整个服务器范围内），在一个响应头里返回其所能支持的全部方法列表。除了用于服务器信息检测，OPTIONS 方法在实际中几乎从来不会用到；而即使用于这个目的，由于返回的值过于有限，所以最后的信息也不会精确。

注意　如果想把事儿说个完整，那还得额外地补一句，OPTIONS 请求是当前还处于提议阶段的跨域请求授权协议的基础，所以日后 OPTIONS 请求的应用可能会更广泛。我们以后会再讲到这个协议，并在第 16 章中探讨许多还在试验阶段的浏览器安全功能。

3.2.5　PUT

PUT 请求用来向服务器特定目标 URL 上传文件。但因为浏览器并不支持 PUT 方法，常规的文件上传功能一般是用 POST 方法提交给服务器端脚本来实现的，而不会用这个理论上更优雅的做法。

当然也有某些非浏览器的 HTTP 客户端和服务器出于某种考虑允许使用 PUT。有趣的是，由于某些 Web 服务器上的配置错误，可能导致这些服务器支持 PUT 请求，从而产生明显的安全漏洞。

3.2.6　DELETE

DELETE 这个方法看字面就知道它的意思了，其他方面和 PUT 类似（也同样很不常用）。

3.2.7　TRACE

TRACE 是一种“ping”形式的请求，服务器会返回 HTTP 请求经过所有代理后的每一跳信息，并且把原始 HTTP 请求内容也再回显出来。TRACE 请求不能由浏览器发出，也很少用于合法用途。TRACE 主要用在安全测试里，用于揭示远程网络的内部架构里一些有趣的细节。因为这个缘故，服务器管理员一般都会把这个功能禁掉。

3.2.8　CONNECT

CONNECT 方法是通过 HTTP 代理服务器建立非 HTTP 类型连接时使用的。Connect 指令是不能直接发送给目标服务器的。如果特定服务器上不小心设置了支持 CONNECT 方法，就可能导致安全风险，使攻击者获得了一条访问受保护网络的 TCP 通道。

3.2.9　其他 HTTP 方法

还有一些其他的请求方法，都只能由非浏览器的应用或浏览器扩展发起；最常见的 HTTP 扩展可能就是 WebDAV，RFC 4918[12] 里描述这个 HTTP 方法是用于写作和版本控制协议的。

另外，客户端一般都可以通过 XMLHttpRequest API，使用 JavaScript 向脚本所在页面的服务器发送各种方法的请求，尽管这个功能在某些浏览器里会受到很多限制（我们会在第 9 章再详细讨论这个问题）。

3.3　服务器响应代码

RFC 2616 的第 10 部分列出了将近 50 种状态码，服务器在返回响应时可以选择其中一种。其中 15 种属于常用的响应代码，其他的都是用来标识一些特别离奇或完全不太可能出现的状态，如“402 Payment Required”（需要支付）或“415 Unsupported Media Type”（不支持的媒体类型）。RFC 里列出来的大部分状态码和我们现在碰到的 Web 应用其实对应不起来；它们存在的唯一理由是某些人觉得日后也许能派上用场。

有些代码是值得记住的，因为确实很常用或者具有特殊的含义，这些常用代码描述如下。

200 ～ 299：成功

这个范围内的状态码用于显示成功完成的请求：

200 OK——对成功的 GET 或 POST 请求的正常响应。浏览器会接着给用户显示返回的数据，或以其他上下文相关的方式处理这些数据。

204 No Content（无内容）——该响应码有时候表明请求已执行成功但不需要做任何响应。204 响应码会中断浏览器的页面跳转，浏览器会保持原 URL 不变，页面内容也维持原状。

206 Partial Content（部分内容）——这个响应码和 200 响应码类似，但服务器会根据请求头里 Range 的设置只返回部分的请求内容。 浏览器应该已经接收了请求文档的其中一部分内容（否则它也不会发出一个 `Range` 的请求），通常来说，浏览器会根据响应里 `Content-Range` 信息，在做进一步处理前会先把文档整合到一起。

300 ～ 399: 重定向和其他状态信息

这个范围的响应码代表没有出错，但浏览器需要做额外处理的响应：

301 Moved Permanently（永久移动）、302 Found（找到）、303 See Other（参见

其他）——这些响应码意味着浏览器需要重定向到新的地址，新地址会在 Location 响应头里指定。尽管在 RFC 规范里这几个响应码是有区别的，但所有我们常用的浏览器碰到这些响应码时，都会把 POST 方法替换成 GET 方法，去除请求里的 POST 数据体部分，并自动重新再发送一次请求。

> **注意**　重定向的信息里可能包含信息体数据[⊖]，但正常情况下浏览器并不会显示这部分的内容，除非浏览器无法完成重定向的动作（例如，缺少 Location 值或 Location 头域的格式有问题）。事实上，有些浏览器即使在无法跳转的情况下，也不会显示返回的信息体部分内容。

304 Not Modified（没有变化）——这个非重定向请求告诉客户端，现在这次请求的文档和之前请求过的版本相比并没有变化。这个响应信息会根据 If-Modified-Since 头域的设定，对最后修改时间进行条件判断，然后确定是否需要更新浏览器端的文件缓存副本。响应的信息体部分也不会显示给用户（如果请求里并没有包含任何涉及缓存更新条件的头域，而服务器却返回了 304 响应码，各浏览器端的反应会不尽相同，有的甚至会产生某些荒谬的效果；譬如，Opera 就会弹出一个无法执行成功的下载提示）。

307 Temporary Redirect（临时重定向）——与 302 类似，但和其他重定向不同的是，在 307 状态码时浏览器不会把 POST 重置为 GET 方法。这个响应码在 Web 应用里不太常用，而某些浏览器对它的处理也不是很统一。

400 ～ 499: 客户端错误

这个范围的代码用于显示客户端行为导致的错误：

400 Bad Request（不合规范的请求）——服务器因为某些特定原因，不能或不愿处理该请求。响应的信息体里通常会大略地解释一下出错的原因，其他的就和浏览器处理普通的 200 响应没啥分别了。当然还有更细分的报错信息，如“411 Length Required（需要提供内容长度）”，“405 Method Not Allowed（不允许的方法）”或“414 Request-URI Too Long（请求的 URI 太长）”。但让人摸不着头脑的是，为什么没有 Content-Length 请求头会对应一个专门的 411 返回码，但没有 Host 请求头，却只有一个笼统的 400 返回码。

401 Unauthorized（未授权）——这个代码的意思是，要访问特定的资源，用户需要提供基于 HTTP 协议级别的授权认证信息。接着浏览器一般都会提示用户输入登录信息，只有在授权失败的情况下，才会返回响应消息体。这个机制将在 3.8 节“HTTP 认证”中

⊖ 这里的信息体数据（此处原文为 payload，有时候也叫 message body），指的是 HTTP 响应返回的数据里，除了头域部分以外的内容，也就是常规情况下，HTTP 协议请求获取的文档所对应的具体内容。——译者注

再详细解释。

403 Forbidden（禁止访问）——与不正确的 HTTP 授权无关，所请求的 URL 存在但不能被访问到。原因可能是文件系统权限不足，规则配置不允许处理本次请求，或某种授权不足（例如：无效的 Cookie 或不被接受的源 IP 地址）。通常响应消息体都会显示给用户。

404 Not Found（文件找不到）——请求的 URL 资源不存在。通常响应消息体都会显示给用户。

500 ～ 599: 服务器端错误

这类错误代码用于显示服务器端的错误：

500 Internal Server Error（内部服务器错误）、503 Service Unavailable（服务不可用）等——服务器发生问题，无法正确完成请求。这可能只是个暂时的状态、配置错误，或仅仅是请求了某个找不到的地址。通常响应消息体都会显示给用户。

HTTP 响应码的一致性

乍一看，其实很难区分大部分 2xx、4xx 和 5xx 响应码到底有什么差别，这导致人们对这些数值的选择也就不会特别费心。所以很多网站应用也因为这个问题而招致骂名无数，因为即使应用出错了，访问的页面里还是返回“200 OK”（这也是其中一个原因，使得 Web 应用自动测试超出了其原本应有的难度。）

在很罕见的情况下，必须根据特定的需求，创建新的合理的 HTTP 响应码。在这类新创建的响应码中有些已经被纳为标准响应码了，如 WebDAV RFC[13] 里用到的几个状态码。而微软公司的 Microsoft Exchange 服务器的“449 Retry With”（重试）状态码，则属于未获标准认可的代码。

3.4　持续会话

最开始的时候，HTTP 会话都是单次完结的：每个 TCP 连接发一次请求，断开，再重复。这需要不断重复进行 TCP 三步握手的消耗（而且在传统 Unix 服务器设计模式中，就要为此新开一个进程），可想而知这种做法很快就变成了性能瓶颈，所以 HTTP/1.1 已经规范化了持续会话（keepalive session）的处理。

现有的协议已经能让服务器知道客户端请求在何时结束（遇到空行的时候结束，如果有 `Content-Length` 请求头，则在收完 `Content-Length` 字节长度的数据后结束），但如果希望继续使用当前连接，客户端也需要知道返回的内容在什么时候结束；因为这时候连接的中断和响应的结束已经没有必然联系了。因此，使用 KeepAlive 会话的时候，响应端也

要包含一个 Content-Length 头域，由此判断还有多少数据会传过来。按照这个数值接收完数据之后，客户端就知道可以发送第二个请求和开始等待下一次响应了。

尽管这个机制对性能是大有裨益，但它的设计也加重了 HTTP 请求和响应里出现拆分漏洞的可能。这对判断客户端和服务器之间是否同步，哪个响应对应哪个请求会产生一定的迷惑性。要说清楚这个问题，我们假设服务器认为自己在发送单个 HTTP 响应，结构如下：

```
HTTP/1.1 200 OK[CR][LF]
Set-Cookie: term=[CR]Content-Length: 0[CR][CR]HTTP/1.1 200 OK[CR]Gotcha: Yup[CR][LF]
Content-Length: 17[CR][LF]
[CR][LF]
Action completed.
```

而另一方面，在客户端看来这可能是两个响应，并把第一个响应和距离现在最近的那个请求相关联起来，第二个响应则对应那个甚至还没有发出的请求[⊖]（这个请求甚至可能是发往同一个 IP 但不同主机名的）：

```
HTTP/1.1 200 OK
Set-Cookie: term=
Content-Length: 0

HTTP/1.1 200 OK
Gotcha: Yup
Content-Length: 17

Action completed.
```

如果是 HTTP 代理服务器收到这个响应，就可能被错误地在全局缓存起来并返回给其他用户，那可真是个坏消息啊。更可靠的持续会话设计应该是同时给出头域部分和消息体部分的数据长度，或用一个随机产生不可预期的边界值作为每个响应之间的分隔符。但很遗憾的是，持续会话的设计里这两种方式都欠奉。

在 HTTP/1.1 里持续会话是默认的连接方式，除非请求里明确要求不用这个功能（这种情况下请求头域 Connection: close），很多 HTTP/1.0 服务器在碰到请求头为 Connection: keep-alive 设置时，也支持持续会话这一功能。服务器和浏览器端都可以限定每个连接同时支持的最大请求数，并可限定空闲连接的最大等待时间。

⊖ 按照设计，在一个持续连接会话中，客户端应该在发送完毕那些按顺序排列的请求之前，丢弃那些不请自来（先于后续请求而返回）的服务端响应数据包，以此来限制这种攻击的影响。但事实上，这个原则由于 HTTP 管道化的考虑而没有得到很好地遵守。HTTP 管道化是一种为了提高网络 IO 吞吐效率而设计的技术。采用了这种技术的客户端被设计为一次性发送多个请求，而不会等每个请求的响应都收回来后再发送。

3.5　分段数据传输

在基于 Content-Length 的持续会话里其中一项重要的限制就是，服务器需要提前知道待返回的响应数据的准确长度。这在处理静态文件的时候是件很简单的事情，因为文件大小的信息完全可以从文件系统中获得。但返回动态产生的数据时，这个问题就变得复杂了，因为输出的内容在返回给客户端之前，必须先要把它的内容整体缓存起来。如果要返回的数据非常大，或者数据是一部分一部分传送的（如实时视频流媒体），这就变成个难以解决的问题了。因为在这些情况下，提前缓存文件、计算出返回数据的大小就变得不可能了。

为应对这一挑战，RFC 2616 的 3.6.1 节为服务器提供了 Transfer-Encoding: chunked（分块传输）功能，在这个协议里，只要可能，产生的数据就立刻作为部分内容先发送出去。单独传输的每部分长度，都以 16 进制整数标识并放在一个单独的行里，但整个文件的长度是不确定的，直到出现最后一个 0 字节长度标志着整段传完了。

一个分段传输的响应样例如下：

```
HTTP/1.1 200 OK
Transfer-Encoding: chunked
...
5
Hello
6
world!
0
```

支持分段数据传输不会带来什么明显的副作用，除了有可能因为分段的数量太多，导致浏览器代码的整数溢出问题，或者需要解决 Content-Length 值和 Chunk 长度不匹配的问题（规范里认为应该以 Chunk 的长度为准，但要想优雅地解决这一难题，那基本没可能了。）所有常用浏览器对 Chunk 的处理都比较到位，但新开发的浏览器要实现这个功能就要小心了。

3.6　缓存机制

出于性能和带宽的考虑，HTTP 客户端和一些中转性质的系统，都会对 HTTP 响应进行缓存以便重用。在早期互联网时代，这件事看起来还算简单，但随着 Web 承载的敏感信息、用户相关信息越来越多，更新也越来越频繁，缓存的问题也日益变得步步惊心。

RFC 2616 的 13.4 节里说，如果没有其他服务器端指令，客户端可以默认对若干 GET 请求的 HTTP 响应码（比较知名的如“200 OK”和“301 Moved Permanently”）进行缓存。

这些响应甚至可以不限时长地一直缓存下去，只要请求方法和目标 URL 与之前一致就行，即使其他参数（如 Cookie 的值）不一样，也可以重用缓存。而使用 HTTP 授权方式的请求则不能使用缓存（见 3.8 节“HTTP 认证”），但其他授权认证方式，如 Cookie，则未在规范里明确限定。

当响应被缓存以后，客户端一般都会在重用前判断一下是否需要验证和重载内容，当然大多数时候并不需要主动这么做。服务器往往根据一些条件判断型的请求头，来确认是否需要进行内容刷新，如 `If-Modified-Since` 请求头（它后面的值就是上一次缓存该页面的时间），又比如 `If-None-Match`（后面跟着一个不透明的 `ETag` 值，这个值代表的是上次访问该文件时，服务器端推送过来的文件标识符）。服务器端将根据请求头里发过来的条件，如果 `ETag` 标识符与上次相比没有变化，则响应一个“304 Not Modified”（没有变化）代码，表示无需刷新，又或者根据需要再完整地返回一次该资源的副本。

> **注意**　通过 `Date/If-Modified-Since` 和 `ETag/If-None-Match` 这两组响应 / 请求头的搭配，再结合使用 `Cache-Control: private` 响应头，就能方便而隐秘地获得浏览器在一段时间内的访问规律和使用习惯[14]。同样地，也可以将一个独一无二的字符串标记嵌到可缓存的 JavaScript 文件里，然后在访问该文件时，如果请求头里包含了缓存条件，一律答复“304 Not Modified（没有变化）”，也能达到同样的效果。和专用于追踪用户行为的其他机制如 HTTP Cookie（将在下一节讨论）相比，浏览器会缓存什么信息、什么时候被缓存、会缓存多久等问题，用户几乎完全没有控制权。

由于隐式的缓存机制会带来许多问题，因此服务器几乎总是倾向于使用显式的 HTTP 缓存指令。为了实现由服务器端控制的缓存机制，HTTP/1.0 里提供了 `Expires` 响应头，设定到什么时候，被缓存的副本即告失效；如果这个值正好等于服务器提供的 `Date` 响应头，则表示这个响应不能再被缓存了。但除了这条简单的规则之外，`Expires` 和 `Date` 之间的联系就没有更详细的说明了：所以也不清楚 `Expires` 值是和缓存服务器的系统时间相比呢（如果客户端和服务器的时间不一致，就可能有这问题），还是根据 `Expires` 与 `Date` 两数字相减的具体差值来说的（这种方式相对更可靠些，但如果缺少了 Date 值，这种方式则可能无法正常工作）。Firefox 和 Opera 采用后一种做法，而其他浏览器则选择前一种。对大多数浏览器来说，如果 `Expires` 的值是无效的，则干脆不使用缓存，但这点并不可靠，不能依赖于这个处理。

HTTP/1.0 客户端还有一个 `Pragma: no-cache` 请求头，代理服务器如果收到这个请求头，就会重新再抓取一遍被请求的资源副本，而不是返回已有的缓存内容。某些 HTTP/1.0 代理服务器还会接受非标准的 `Pragma: no-cache` 响应头，然后对使用这个响应

头的文件也不会进行缓存。

HTTP/1.1 的处理则不同，它使用新增的 Cache-Control 做为缓存指令，这种方式的可用性更高。这个头域有 4 种域值：public（可以公开被缓存的文档）、private（代理服务器不得缓存的文档）、no-cache（这个命名看上去比较让人误解，意思是响应的文档可以被缓存但不能重用）[⊖]和 no-store（绝对不需要被缓存）。Public 和 Private 缓存指令可以搭配 max-age 和 must-revalidate 两个确定因子一起使用，前者决定旧副本的存活时间，后者则可以按照条件判断型的请求头来确定是否需要重用内容。

遗憾的是，Web 服务器往往需要同时返回 HTTP/1.0 和 HTTP/1.1 的缓存指令，因为某几种古老的商用代理服务器软件不能正确识别 Cache-Control 指令。为了可靠地禁止 HTTP 缓存，Web 服务器可能需要返回全部以下响应头：

```
Expires: [ 当前日期时间 ]
Date: [ 当前日期时间 ]
Pragma: no-cache
Cache-Control: no-cache, no-store
```

当这些缓存指令之间有冲突时，会出现什么状况很难预测：有的浏览器倾向于认为 HTTP/1.1 协议里的缓存设置和 Cache-Control 里的 no-cache 设置的优先级都较高，即使在 no-cache 后面还错误地跟着个 public 值也还是以 no-cache 为准；而另一些浏览器则刚好相反。

HTTP 缓存的另一个风险和不安全网络环境有关系，例如不加密的 Wi-Fi 网络，黑客可以通过拦截对某些 URL 地址的请求，向受害者返回被篡改过并可以长期缓存的请求内容。然后这种受到污染的浏览器缓存如果在受信任的网络上被重用，被注入的内容可能就出人意料地重新浮现出来了。更邪恶的是，受害者甚至无需直接访问目标网站：攻击者可以精心选择一些敏感域，然后在其他的上下文环境中再引用这些域的内容。这样，被攻击者即使没有访问最早的那个目标网站，攻击者也能达到目的。目前对这个问题还没有很好的解决方法，反正在星巴克咖啡店上网之后，清空浏览器缓存也许是个好主意。

3.7　HTTP Cookie 语义

HTTP Cookie 不属于 RFC 2616 的内容，但它们是 Web 最重要的协议扩展之一。Cookie 机制的使用过程首先是由服务器端返回一个 Set-Cookie 响应头，把一组组短小而不透明的“名称 = 值”格式的数据保存到浏览器端，在客户端再次提交请求时，会把之前

⊖ RFC 在这方面说得有点含糊，貌似其本意是允许被缓存的文件在类似浏览器的“后退”和“前进”按键时使用，而真的涉及页面的加载时则不能用。Firefox 就采用这种处理，而其他浏览器则认为 no-cache 和 no-store 代表了差不多的意思。

保存下来的 Cookie 参数再返回给服务器端，服务器就可以重新再取回这些信息了。Cookie 是到目前为止，保持会话和用户授权请求的最常用方式，它是 Web 环境里 4 种正统的全局授权认证方式（Ambient authority ㊀）㊁之一（另外几种内置的授权方式包括：HTTP 验证授权、IP 检查和客户端证书）。

Cookie 机制最开始是 1994 年由 Lou Montulli 在 Netscape 浏览器里实现的，他简要地在一份 4 页纸的草案里描述了这一机制[15]。而在过去 17 年里，竟然就一直没有一份恰当描述这套机制的标准文档。在 1997 年，RFC 2109[16] 尝试着勾画了当时的 Cookie 现状，但又显得有点含糊不清，它还提议了若干彻底的修改，但这些提议又和现在用的任意一款浏览器的实际表现都不符。而另一项野心勃勃的尝试——Cookie2，出现在 RFC 2965[17] 里，但 10 年过去了，实际上还没有任何一款浏览器支持它，这种情况现在看来也不会有啥变化。一份对 Cookie 进行了较为准确描述的规范性文档就是 RFC 6265[18]，在本书原版付印之前刚刚完成，终于一扫这场关于 Cookie 规范的迷雾。

由于长期以来缺乏真正的标准，所以对 Cookie 的各种具体实现也非常有意思，有时候甚至彼此不兼容。根本上来说，一个新的 Cookie 是通过 `Set-Cookie` 响应头进行设置，后面跟着一串“名字 = 域值”格式的数据对，和若干以分号分开的可选参数，这些参数用于定义 Cookie 的范围和生命周期，关于这些参数的解释如下：

Expires　设定 Cookie 过期的时间，格式与 `Date` 或 `Expires` 响应头里的相类似。如果 Cookie 产生时没有设定过期时间，这意味着 Cookie 将在浏览器的整个会话期间内都有效（尤其是如果便携式计算机上有“挂起”功能时，Cookie 的存活可能会持续数周之久）。定义了过期时间的 Cookie 被存放在硬盘上，所以在不同的会话之间仍然能保持，除非用户的隐私设置里明确地禁止了这种用法。

Max-age　这是另一种 RFC 建议的过期时间设定，但由于 IE 浏览器不支持这种用法，所以实际上不会碰到。

Domain　如果希望 Cookie 的有效范围，超出当前返回 `Set-Cookie` 响应头的那个主机头名范围，则需要设置该参数。关于 Cookie 有效范围的精确规则运用以及相关的安全问题，我们将在第 9 章里再详加讨论。

注意　与 RFC 2109 里的规定不同，其实不能用这个参数来设定 Cookie 只在某

㊀ Ambient authority 是一种基于 HTTP 请求里各种全局和持久型属性，进行访问控制的授权方式，而非那种只针对某个特定提交有效的显式授权方式。用于确定用户身份的 Cookie 在与远程服务器进行连接的每个请求里，都会一视同仁地被发出去，不管这次请求是如何被发起的，因此 Cookie 也属于这种全局授权类别。

㊁ 由于未见约定俗成的称呼，所以参照上面脚注的说明，在本书内会统一译成“全局授权认证方式”，特此说明。——译者注

个主机名范围内有效。例如，如果设置了 domain=example.com，其实也会匹配到 www.example.com 的二级域名范围。只有完全不设置 Domain 项，才是使得 Cookie 只在当前主机名范围内有效的唯一方法，但即使这样，在 IE 浏览器里也仍然会有一些出人意料的问题。

Path　允许 Cookie 在特定的请求路径内有效。但这并不是一个可靠的安全机制，原因会在第 9 章里解释，但因为它的简便易用可以考虑采纳这种设置，它可以避免完全同名的 Cookie 在应用的不同部分产生冲突。

Secure 属性　禁止以非加密传输方式来使用 Cookie。

HttpOnly 属性　禁止通过 JavaScript 的 `document.cookie` API 方式来读取 Cookie。该可选项原本是微软的一个扩展，但现在所有的主流浏览器都支持这一特性。

如果当前访问的域名在浏览器的 Cookie 池中有对应的 Cookie 数据，浏览器就会把所有适用于这个域名的 Cookie，按照“名称 = 值”的数据对格式，中间用分号分隔拼接在一起，无需再带其他的元数据，组合成一行请求头返回给服务器端。如果在特定的请求里，需要发送给服务器的 Cookie 太多了，已经超出服务器允许的头域长度限制，那该次请求可能会失败；除了手工清除 Cookie 池里的内容，也别无其他办法。

让人觉得有意思的是，对服务器来说并没有什么显式的方法来删除不需要的 Cookie。然而，由于每个 Cookie 都是独一无二地通过“名称 – 域名 – 路径”（name-domain-path）这三个基本要素确定的（`Secure` 和 `Httponly` 属性会被忽略 ），所以可以用这个方法来覆盖特定范围内一个老的 Cookie 值。更进一步来说，如果新 Cookie 设置的到期时间（Expires）比当前实际时间要早，那这个 Cookie 就会被剔除掉，这倒是一种有效但有点绕的删除 Cookie 数据的办法。

尽管 RFC 2109 规定要接受由逗号分隔的设置在同一个 `Set-Cookie` 响应头里的多个 Cookie，但这种做法非常危险，所有的浏览器都不再支持这种方式了。Firefox 允许通过 `document.cookie` JavaScript API 一次性地设置多个 Cookie，但是这种方法很绕，需要用换行来做分隔符。没有一个浏览器接受以逗号作为 `Cookie` 的分隔符，而在服务器端接受这种格式的 Cookie 也是不安全的。

另一个与规范有重大差异的地方是，在 HTTP 规范里，Cookie 的值应该使用 `Quoted-String` 的格式（见 3.1.5 节“以分号作为分隔符的头域值”），但实际上只有 Firefox 和 Opera 才接受这种格式。因此完全依赖于 `Quoted-String` 的处理，以及在攻击者控制的 Cookie 里允许出现单个的引号，都是很不安全的做法。

当然 Cookie 本身就不是特别可靠。所以有些客户端程序可以强制使用较为保守的用户配置策略，限定每个域名所允许的 Cookie 数量和大小，这样甚至可以限制 Cookie 的生

命周期，但这类隐私保护措施都是被误导的结果。因为有别的方式能同样可靠地追踪用户的操作，比如上一章节里提到的 ETag/If-None-Match，所以限制 Cookie 的使用可能反而有弊无利。

3.8　HTTP 认证

按 RFC 2617[19] 原本的设想，HTTP 认证方式是专为 Web 程序量身打造的身份验证机制，但现在却几近绝迹。其中的原因可能是 HTTP 认证的界面与浏览器程序自身的 UI 相关，导致界面布局很不灵活；也很难兼容其他更复杂的非基于密码的授权机制；也无法控制授权认证会缓存多长时间才失效，并且很难和其他域名共享授权。

不管怎么说，这套基本的授权协议相当简单。最开始的时候，浏览器会先发送一个未被认证（Unauthenticated）的普通 HTTP 请求，这时服务器会返回一个 “401 Unauthorized”（未授权）响应代码㊀。而服务器返回的响应里，还会包含一个 WWW-Authenticate ㊁的 HTTP 头域，其中包括本次请求的验证方式和标识此范围域的一个名为 Realm 的字符串（这串由任意字符组成的标识符代表着本次授权的范围），此外可能还会跟着其他一些与该方法有关的可选参数。

无论以哪种方式，客户端在获得用户输入的授权信息后，会在之前那次普通 HTTP 提交的基础上，再增加一个 Authorization 请求头㊂，这个请求头里存放着经过编码的授权信息，然后重发一次请求。根据规范，出于对性能的考虑，如果后续访问的是同一服务器的相同目录时，在后续的请求里会一律自动加入 Authorization 请求头，而无需由服务器端再发一次 WWW-Authenticate 提问。另外在访问服务器的不同目录时，如果收到的 Realm String 值和授权方法与之前的都一样，也允许重用之前的 WWW-Authenticate 授权信息而无需重新输入。

在实际应用中，这些建议并未完全被采纳：除了 Safari 和 Chrome，其他大多数浏览器都忽略 Realm String 值，或对路径的匹配采用非常宽松的模式。但另一方面来说，所有的浏览器在确定缓存的授权信息的适用范围时，除比较目的端服务器信息外，还会比较使用的协议和端口，这种做法实际上对安全更有利些。

㊀ 认证（Authentication）和授权（Authorization) 在该 RFC 文档里经常互换使用，但在信息安全领域里，它们有着显著的含义差别。认证通常指证明身份的过程，而授权指的是基于已有身份之上，该用户有权执行哪些动作。

㊁ 一个基本 HTTP 验证的 WWW-Authenticate 响应头示例如下 WWW-Authenticate: Basic realm="This is a bunny domain."——译者注

㊂ 一个基本 HTTP 验证的请求头例子如下：Authorization: Basic a2V2aW46OTUxMS5uZXQ==——译者注

在原始 RFC 文档里，规定了两种授权认证方法，分别为基本认证和摘要认证。基本认证差不多就是把密码用 Base64 编码后以纯文本方式发送。摘要认证的方式则需要计算一个单次有效的加密摘要值，以避免密码被明文查看和防止 `Authorization` 头域被重用。但不幸的是，现在常用的浏览器都同时支持这两种认证方式，而且在使用时不会特别严密地区分两者。因此，攻击者完全可以在初始请求之后，把服务器返回内容里的 `Digest`（摘要）一词简单地偷换成 `Basic`（基本），这样在用户输入认证信息并提交之后，就能获得一个明文的纯文本密码了。让人惊讶的是，RFC 预见到了这种风险，并给出了一些有用但最终未被采纳的建议：

> 客户端应该考虑在界面明显的地方提示当前使用的是哪种验证策略；或者记着与该服务器之间最严格的验证策略是哪种，在采用较弱的策略之前，应该给与必要的警示。或者客户端干脆默认地，或针对某些特定站点，配置为摘要认证方式。

除了这两种 RFC 规定的验证策略，某些浏览器还支持一些不太常见的方法，如微软的 NTLM 和 Negotiate，以用于和 Windows 域身份认证无缝结合的授权 [20]。

尽管 HTTP 认证在互联网上很少碰到，但它对某些网站应用还是会有很大影响。例如，在论坛的帖子里，黑客可以放一个使用外部链接的图片，放图片的服务器对某些 HTTP 请求发来一个“401 Unauthorized（未授权）”响应，此时正在浏览帖子的用户会非常意外地看到这个来路不明的密码提示框。在仔细看过浏览器里的地址没错之后，大部分人恐怕都会上当而输入自己在论坛上的帐号密码，这些信息立刻就会发到攻击者放图片的服务器上。好家伙！

3.9　协议级别的加密和客户端证书

写到这里已经很明显了，HTTP 会话在网络上进行的所有信息交换，都是明文方式的。在上世纪 90 年代，这不算啥大问题：没错，明文信息会把你的浏览习惯暴露给聒噪的 ISP 服务商们，或办公室内网里某个搞恶作剧的家伙，甚至惹来某些过于热心的政府人士，但这些状况和 SMTP、DNS 以及其他常用协议的问题相比，似乎又不算那么严重。但随着 Web 日益成为最受欢迎的商务平台，风险愈加凸显，另外随着公用无线网络的出现，由于 Web 本质上就已经不太安全了，所以更会导致严重的网络安全性倒退，这简直是雪上加霜。

在尝试过几种不太成功的做法后，在 RFC 2818[21] 里终于找到了可行之道：为何不用于几年前出现的多用途传输层加密（Transport Layer Security，简称 TLS，也叫 SSL）机

制来传输常规的 HTTP 请求呢。这种传输方式使用公钥加密算法[⊖]建立一条保密的可信任的通信通道，而在 HTTP 级别服务器和客户端都无需再作任何调整。

为了让网站服务器能证明自己的身份，每个支持 HTTPS 的浏览器都内置了一大堆各种各样的证书发行中心（Certificate Authorities，下文简称 CA 中心）的公钥信息。CA 中心是浏览器开发商信任的授权机构，它为有需要的网站颁发用于验证的服务器公钥，颁发前该机构应尽量确认申请者的身份，并确保颁发的服务器证书确实是由该域名使用的。

每种浏览器里内置的受信任机构各自不同，随心所欲，也没有什么特别详细的文档规范，这点经常招致各种批评。但总体来说，这套系统还算运作良好。到目前为止只有区区几次出问题的报道（其中就有近期非常瞩目的 Comodo[22] 公司 CA 系统被利用的问题），目前还没有大范围的 CA 系统特权被滥用的报道。

至于 CA 中心的具体实现上，就是在创建一个新的 HTTPS 连接时，浏览器端收到服务器的签名公钥，验证签名后（除非 CA 的私钥泄漏，否则签名无法被伪造），检查证书里被签名的 `cn` 项（Common Name，常用名称）或 `subjectAltName` 项的值，由此确认对方确实是浏览器真正要访问的服务器，并确认该公钥不在 CA 机构的公开撤销列表里（例如，证书是假冒或欺诈手段获取到的）。如果所有检查都通过了，浏览器就可以用这个公钥加密信息并传回给服务器端，通过这种方式，确认只有特定的接收者才能对加密信息进行解密。

一般来说，客户端是匿名的。它产生一个临时的密钥，但这个处理并不能证实客户端的身份。当然要想证明客户端身份也是可以的。证书发行机构也都内建支持客户端证书，全球范围内已有若干国家在全国范围内承认电子证书（例如用于政府的电子政务）的使用。客户端证书最常规的用途就是证明真实世界里使用者的身份，所以在访问站点之初，出于隐私的考虑，浏览器往往需要提示一下用户，是否需要发送客户端证书；除此之外，客户端证书也可以作为全局授权认证方式的一种形式。

值得注意的是，尽管 HTTPS 对被动型和主动型攻击者都能防御，但 HTTPS 访问里一些已知的公开信息还是难以避免地暴露在外。譬如，我们还是能获得访问会话里 HTTP 请求和响应的大小、访问流量的进出方向、时间模式的规律，等等，对那些全盘照收数据的被动型攻击者来说，这些信息甚至能泄漏用户正在使用加密通道浏览维基百科上哪些不雅页面。实际上，在一个极端的例子里，微软的研究者甚至通过分析这种数据包的统计信息，重组还原出用户对某个线上应用访问时的键盘输入[23]。

3.9.1　扩展验证型证书

在 HTTPS 的早期阶段，证书授权机构在正式签发证书之前，都需要进行相当繁琐复杂的用

⊖ 公钥加密技术依赖于非对称加密算法，由此创建一对密钥：私钥严格地由使用者自己保管着，用于信息的解密，而公钥则是广泛对外公开使用的，基本上只用来加密给接收者的信息，而不能用于解密。

户身份验证和域名属主权限检查。但遗憾的是，出于方便和降低成本的考虑，某些机构现在可能只需要一张有效的信用卡，和在目标服务器上放一个用于完成验证的文件即可。这种做法导致证书里除了 cn 和 subjectAltName 两个字段以外的大多数信息都变得不再可信了[⊖]。

为解决这一问题，出现了一种新的安全证书，其市场定价非常昂贵，这种证书用一个特殊标记作为自己的标签：Extended Validation SSL（缩写为 EV SSL）。人们希望通过人工验证的过程，确保这种证书不但能证明域名的属主，并且能更可靠地验证申请人的身份。现在所有的浏览器都支持 EV SSL，在使用这种证书时，浏览器地址栏会自动显示为蓝色或者绿色。尽管这种证书本身确实是有价值的，但把更高价的证书似是而非地与“更高级别的安全”显示待遇等同起来的做法，也招致了广泛的批评，显然这更像是个聪明又隐秘的敛财之道。

3.9.2　出错处理的规则

理想情况下，HTTPS 连接碰到有问题的证书，如主机头名与证书里名称不匹配、或无法识别服务器证书的发行机构，就应该出错并直接拒绝连接的建立。而对较轻微的错误，如刚过期的证书或仅是主机名略有不符，则可能只需要返回一个较温和的警告信息。

但很遗憾，大多数浏览器完全不加区分地把对这些问题的理解和做选择的责任全都推搪给用户，这些浏览器往往很卖力（但最终完全无济于事）地用一些外行话向用户解释密码学上的问题，然后让用户去做个两选一的判断：您是希望看到还是不希望看到这个页面呢？（图 3-1 就显示了这么个提示界面）。

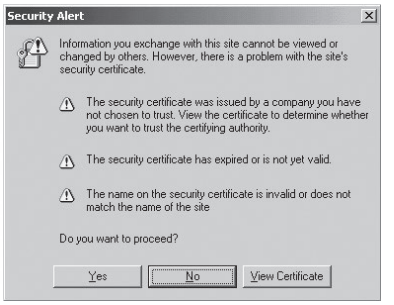


图 3-1　还在广泛使用的 IE 6 浏览器的证书警告提示窗口

多年来，这个 SSL 警告提示里的用语和界面的样式正朝着语言解释越来越傻瓜化（但仍然问题多多），但操作选择却越来越专家化的方向演变。这种趋势可能是一种误导：因为研究显示，即使是最吓人最崩溃的警告提示，也仍然有超过 50% 的用户会选择忽略，一路点击下去选择全部接受[24]。怪罪到用户的头上的确很容易，但说到底，有可能我们问的问题本身就是错的，提供给他们的选择也是错的。简单来说，如果我们确信在某些场景下，用户接受这些警告是有道理的，那么应该直接以“沙箱”（Sandbox）模式打开页面，这种模式下攻击会受到严格限制，并在界面上做出明确的标识，可能是更为合理的解决之道。如果我们确信本来就不该接受这样的警告，那就应彻底杜绝绕过这些警告的可能性（我们在第 16 章里会讨论到试验性的严格传输安全（Strict Transport Security）机制就是为了达到这个目标）。

⊖ 作者的意思是说，尽管证书属性里的 cn 和 subjectAltName 字段是对的，但其他的一些信息，如组织、单位、邮件、国家 / 城市等，CA 就没有认真核对了，因此并不可靠。——译者注

3.10　安全工程速查表

处理 Content-Disposition 头域里用户提供的文件名

☑ **如果不需要用到非拉丁语系文件名**：则只需保留字母数字符号和“.”、“-”和“_”，其他字符一概移除或替换掉。为保护用户免遭有害或具欺骗性文件名的可能性，至少要保证文件名的首字符是字母或数字，并把最后一个句号之外的其他句号一律用其他字符（如下划线）替代。

要时刻记住，允许引号、分号、反斜杠、控制字符（0x00 ～ 0x1F）的出现可能导致安全隐患。

☑ **如果需要用到非拉丁语系的文件名**：应该按照 RFC 2047、RFC 2231 或根据浏览器的情况使用百分号编码的 URL 形式作为文件名。确保过滤控制字符（0x00 ～ 0x1F）和对任何分号、反斜杠和引号进行转义。

处理 HTTP Cookies 里的用户输入信息

☑ **对字母和数字以外的所有字符进行百分号编码**。更好的做法是，干脆用 Base64 编码。那些引号、控制字符（0x00 ～ 0x1F）、高位字符（0x80 ～ 0xFF）、逗号、分号、反斜杠，都有可能导致 Cookie 注入，或使当前 Cookie 的含义和范围发生变化。

发送由用户提供的 Location 头域

☑ **参见第 2 章的速查表**。应对提供的 URL 进行解析和规范化处理，确保 URL 对应的协议在允许的白名单里，以及重定向到指定的主机是安全的。

确保任何控制字符和高位字符都恰当地进行了转义。主机名部分需使用 Punycode 编码，除此以外的 URL 其余部分，则需采用百分号编码方式。

发送用户提供的 Redirect 头域

☑ **遵从与 Location 主机头一样的建议**。注意在这个头域里，分号也是不安全的，而且没办法可靠地进行转义，而分号在某些特定的 URL 里又有特殊的含义。应该干脆杜绝这样的 URL 或对“;”字符进行百分号编码，但这样就违背了 RFC 的语法规范。

构建其他类别的用户输入请求或响应

☑ **进行语法检查，排除头域可能引致的副作用**。常规来说，要尤其注意控制字符、高位字符、逗号、引号、反斜杠和分号；其他字符或字串可能在某些个案上会有影响。要根据需要，对这些字符进行转义或替换处理。

☑ **创建新的 HTTP 客户端、服务器或代理时**：不要全新地开发自己的实现方式，除非真的迫不得已。如果一定要这么做，那必须认真阅读本章内容，并尽可能模仿已有的主流实现。在碰到模棱两可的语法解析时，如果可能，不要理会 RFC 里那些倾向于容忍错误的建议和消极回避的处理。