在 Web 中，一个浏览器进程向一台 Web 服务器进程发起联系，因此该浏览器进程是客户，而该 Web 服务器进程是服务器。在 P2P 文件共享中，当对等方 A 请求对等方 B 发送一个特定的文件时，在这个特定的通信会话中对等方 A 是客户，而对等方 B 是服务器。在不致混淆的情况下，我们有时也使用术语“应用程序的客户端和服务器端”。在本章的结尾，我们将逐步讲解网络应用程序的客户端和服务器端的简单代码。

2. 进程与计算机网络之间的接口

如上所述，多数应用程序是由通信进程对组成，每对中的两个进程互相发送报文。从一个进程向另一个进程发送的报文必须通过下面的网络。进程通过一个称为**套接字**（socket）的软件接口向网络发送报文和从网络接收报文。我们考虑一个类比来帮助我们理解进程和套接字。进程可类比于一座房子，而它的套接字可以类比于它的门。当一个进程想向位于另外一台主机上的另一个进程发送报文时，它把报文推出该门（套接字）。该发送进程假定该门到另外一侧之间有运输的基础设施，该设施将把报文传送到目的进程的门口。一旦该报文抵达目的主机，它通过接收进程的门（套接字）传递，然后接收进程对该报文进行处理。

图 2-3 显示了两个经过因特网通信的进程之间的套接字通信（图 2-3 中假定由该进程使用的下面运输层协议是因特网的 TCP 协议）。如该图所示，套接字是同一台主机内应用层与运输层之间的接口。由于该套接字是建立网络应用程序的可编程接口，因此套接字也称为应用程序和网络之间的**应用程序编程接口**（Application Programming Interface，API）。应用程序开发者可以控制套接字在应用层端的一切，但是对该套接字的运输层端几乎没有控制权。应用程序开发者对于运输层的控制仅限于：①选择运输层协议；②也许能设定几个运输层参数，如最大缓存和最大报文段长度等（将在第 3 章中涉及）。一旦应用程序开发者选择了一个运输层协议（如果可供选择的话），则应用程序就建立在由该协议提供的运输层服务之上。我们将在 2.7 节中对套接字进行更为详细的探讨。

3. 进程寻址

为了向特定目的地发送邮政邮件，目的地需要有一个地址。类似地，在一台主机上运行的进程为了向在另一台主机上运行的进程发送分组，接收进程需要有一个地址。为了标识该接收进程，需要定义两种信息：①主机的地址；②在目的主机中指定接收进程的标识符。

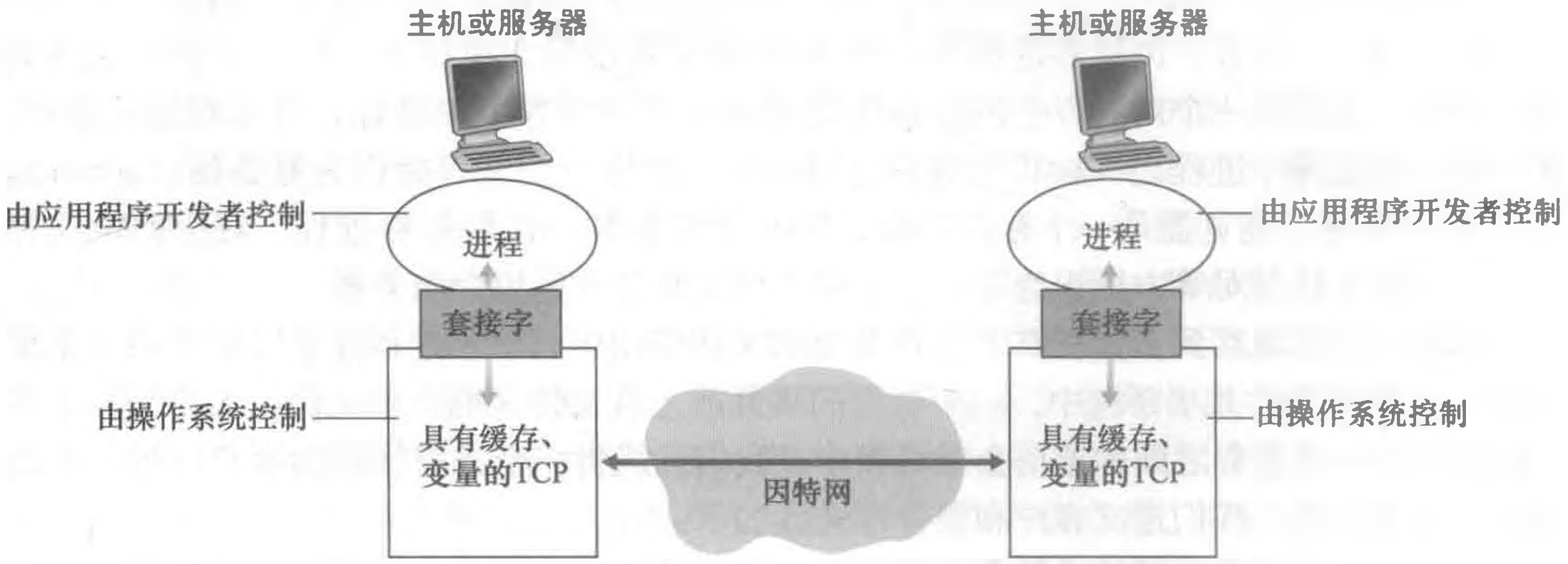

图 2-3　应用进程、套接字和下面的运输层协议

在因特网中，主机由其 **IP 地址**（IP address）标识。我们将在第 4 章中非常详细地讨论 IP 地址。此时，我们只要知道 IP 地址是一个 32 比特的量且它能够唯一地标识该主机就够了。除了知道报文发送目的地的主机地址外，发送进程还必须指定运行在接收主机上的接收进程（更具体地说，接收套接字）。因为一般而言一台主机能够运行许多网络应用，这些信息是需要的。目的地**端口号**（port number）用于这个目的。已经给流行的应用分配了特定的端口号。例如，Web 服务器用端口号 80 来标识。邮件服务器进程（使用 SMTP 协议）用端口号 25 来标识。用于所有因特网标准协议的周知端口号的列表能够在 http://www.iana.org 处找到。我们将在第 3 章中详细学习端口号。

2.1.3 可供应用程序使用的运输服务

前面讲过套接字是应用程序进程和运输层协议之间的接口。在发送端的应用程序将报文推进该套接字。在该套接字的另一侧，运输层协议负责从接收进程的套接字得到该报文。

包括因特网在内的很多网络提供了不止一种运输层协议。当开发一个应用时，必须选择一种可用的运输层协议。如何做出这种选择呢？最可能的方式是，通过研究这些可用的运输层协议所提供的服务，选择一个最能为你的应用需求提供恰当服务的协议。这种情况类似于在两个城市间旅行时选择飞机还是火车作为交通工具。每种运输模式为你提供不同的服务，你必须选择一种或另一种（例如，火车可以直到市区上客和下客，而飞机提供了更短的旅行时间）。

一个运输层协议能够为调用它的应用程序提供什么样的服务呢？我们大体能够从四个方面对应用程序服务要求进行分类：可靠数据传输、吞吐量、定时和安全性。

1. 可靠数据传输

如第 1 章讨论的那样，分组在计算机网络中可能丢失。例如，分组能够使路由器中的缓存溢出，或者当分组中的某些比特损坏后可能被丢弃。像电子邮件、文件传输、远程主机访问、Web 文档传输以及金融应用等这样的应用，数据丢失可能会造成灾难性的后果（在后一种情况下，无论对银行或对顾客都是如此！）。因此，为了支持这些应用，必须做一些工作以确保由应用程序的一端发送的数据正确、完全地交付给该应用程序的另一端。如果一个协议提供了这样的确保数据交付服务，就认为提供了**可靠数据传输**（reliable data transfer）。运输层协议能够潜在地向应用程序提供的一个重要服务是进程到进程的可靠数据传输。当一个运输协议提供这种服务时，发送进程只要将其数据传递进套接字，就可以完全相信该数据将能无差错地到达接收进程。

当一个运输层协议不提供可靠数据传输时，由发送进程发送的某些数据可能到达不了接收进程。这可能能被**容忍丢失的应用**（loss-tolerant application）所接受，最值得注意的是多媒体应用，如交谈式音频/视频，它们能够承受一定量的数据丢失。在这些多媒体应用中，丢失的数据引起播放的音频/视频出现小干扰，而不是致命的损伤。

2. 吞吐量

在第 1 章中我们引入了可用吞吐量的概念，在沿着一条网络路径上的两个进程之间的通信会话场景中，可用吞吐量就是发送进程能够向接收进程交付比特的速率。因为其他会话将共享沿着该网络路径的带宽，并且因为这些会话将会到达和离开，该可用吞吐量将随时间波动。这些观察导致另一种自然的服务，即运输层协议能够以某种特定的速率提供确

保的可用吞吐量。使用这种服务，该应用程序能够请求 r 比特/秒的确保吞吐量，并且该运输协议能够确保可用吞吐量总是为至少 r 比特/秒。这样的确保吞吐量的服务将对许多应用程序有吸引力。例如，如果因特网电话应用程序对语音以 32kbps 的速率进行编码，那么它需要以这个速率向网络发送数据，并以该速率向接收应用程序交付数据。如果运输协议不能提供这种吞吐量，该应用程序或以较低速率进行编码（并且接收足够的吞吐量以维持这种较低的编码速率），或它可能必须放弃发送，这是因为对于这种因特网电话应用而言，接收所需吞吐量的一半是几乎没有或根本没有用处的。具有吞吐量要求的应用程序被称为**带宽敏感的应用**（bandwidth-sensitive application）。许多当前的多媒体应用是带宽敏感的，尽管某些多媒体应用程序可能采用自适应编码技术对数字语音或视频以与当前可用带宽相匹配的速率进行编码。

带宽敏感的应用具有特定的吞吐量要求，而**弹性应用**（elastic application）能够根据当时可用的带宽或多或少地利用可供使用的吞吐量。电子邮件、文件传输以及 Web 传送都属于弹性应用。当然，吞吐量是越多越好。有一句格言说得好，钱越多越好，人越瘦越美，我们永远不会嫌吞吐量太多的！

3. 定时

运输层协议也能提供定时保证。如同具有吞吐量保证那样，定时保证能够以多种形式实现。一个保证的例子如：发送方注入进套接字中的每个比特到达接收方的套接字不迟于 100ms。这种服务将对交互式实时应用程序有吸引力，如因特网电话、虚拟环境、电话会议和多方游戏，所有这些服务为了有效性而要求数据交付有严格的时间限制（参见第 9 章[Gauthier 1999；Ramjee 1994]）。例如，在因特网电话中，较长的时延会导致会话中出现不自然的停顿；在多方游戏和虚拟互动环境中，在做出动作并看到来自环境（如来自位于端到端连接中另一端点的玩家）的响应之间，较长的时延使得它失去真实感。对于非实时的应用，较低的时延总比较高的时延好，但对端到端的时延没有严格的约束。

4. 安全性

最后，运输协议能够为应用程序提供一种或多种安全性服务。例如，在发送主机中，运输协议能够加密由发送进程传输的所有数据，在接收主机中，运输层协议能够在将数据交付给接收进程之前解密这些数据。这种服务将在发送和接收进程之间提供机密性，以防该数据以某种方式在这两个进程之间被观察到。运输协议还能提供除了机密性以外的其他安全性服务，包括数据完整性和端点鉴别，我们将在第 8 章中详细讨论这些主题。

2.1.4　因特网提供的运输服务

至此，我们已经考虑了计算机网络能够提供的通用运输服务。现在我们要更为具体地考察由因特网提供的运输服务类型。因特网（更一般的是 TCP/IP 网络）为应用程序提供两个运输层协议，即 UDP 和 TCP。当你（作为一个软件开发者）为因特网创建一个新的应用时，首先要做出的决定是，选择 UDP 还是选择 TCP。每个协议为调用它们的应用程序提供了不同的服务集合。图 2-4 显示了某些所选的应用程序的服务要求。

1. TCP 服务

TCP 服务模型包括面向连接服务和可靠数据传输服务。当某个应用程序调用 TCP 作为其运输协议时，该应用程序就能获得来自 TCP 的这两种服务。

应用	数据丢失	带宽	时间敏感
文件传输	不能丢失	弹性	不
电子邮件	不能丢失	弹性	不
Web 文档	不能丢失	弹性（几 kbps）	不
因特网电话/视频会议	容忍丢失	音频（几 kbps ~ 1Mbps） 视频（10kbps ~ 5Mbps）	是，100ms
流式存储音频/视频	容忍丢失	同上	是，几秒
交互式游戏	容忍丢失	几 kbps ~ 10kbps	是，100ms
智能手机讯息	不能丢失	弹性	是和不是

图 2-4 选择的网络应用的要求

- 面向连接的服务：在应用层数据报文开始流动之前，TCP 让客户和服务器互相交换运输层控制信息。这个所谓的握手过程提醒客户和服务器，让它们为大量分组的到来做好准备。在握手阶段后，一个 **TCP 连接**（TCP connection）就在两个进程的套接字之间建立了。这条连接是全双工的，即连接双方的进程可以在此连接上同时进行报文收发。当应用程序结束报文发送时，必须拆除该连接。在第 3 章中我们将详细讨论面向连接的服务，并分析它是如何实现的。
- 可靠的数据传送服务：通信进程能够依靠 TCP，无差错、按适当顺序交付所有发送的数据。当应用程序的一端将字节流传进套接字时，它能够依靠 TCP 将相同的字节流交付给接收方的套接字，而没有字节的丢失和冗余。

TCP 协议还具有拥塞控制机制，这种服务不一定能为通信进程带来直接好处，但能为因特网带来整体好处。当发送方和接收方之间的网络出现拥塞时，TCP 的拥塞控制机制会抑制发送进程（客户或服务器）。如我们将在第 3 章中所见，TCP 拥塞控制也试图限制每个 TCP 连接，使它们达到公平共享网络带宽的目的。

关注安全性

TCP 安全

无论 TCP 还是 UDP 都没有提供任何加密机制，这就是说发送进程传进其套接字的数据，与经网络传送到目的进程的数据相同。因此，举例来说如果某发送进程以明文方式（即没有加密）发送了一个口令进入它的套接字，该明文口令将经过发送方与接收方之间的所有链路传送，这就可能在任何中间链路被嗅探和发现。因为隐私和其他安全问题对许多应用而言已经成为至关重要的问题，所以因特网界已经研制了 TCP 的加强版本，称为**安全套接字层**（Secure Sockets Layer，SSL）。用 SSL 加强后的 TCP 不仅能够做传统的 TCP 所能做的一切，而且提供了关键的进程到进程的安全性服务，包括加密、数据完整性和端点鉴别。我们强调 SSL 不是与 TCP 和 UDP 在相同层次上的第三种因特网运输协议，而是一种对 TCP 的加强，这种强化是在应用层上实现的。特别是，如果一个应用程序要使用 SSL 的服务，它需要在该应用程序的客户端和服务器端包括 SSL 代码（利用现有的、高度优化的库和类）。SSL 有它自己的套接字 API，这类似于传统的 TCP 套接字 API。当一个应用使用 SSL 时，发送进程向 SSL 套接字传递明文数据；在发送主

机中的 SSL 则加密该数据并将加密的数据传递给 TCP 套接字。加密的数据经因特网传送到接收进程中的 TCP 套接字。该接收套接字将加密数据传递给 SSL，由其进行解密。最后，SSL 通过它的 SSL 套接字将明文数据传递给接收进程。我们将在第 8 章中更为详细地讨论 SSL。

2. UDP 服务

UDP 是一种不提供不必要服务的轻量级运输协议，它仅提供最小服务。UDP 是无连接的，因此在两个进程通信前没有握手过程。UDP 协议提供一种不可靠数据传送服务，也就是说，当进程将一个报文发送进 UDP 套接字时，UDP 协议并不保证该报文将到达接收进程。不仅如此，到达接收进程的报文也可能是乱序到达的。

UDP 没有包括拥塞控制机制，所以 UDP 的发送端可以用它选定的任何速率向其下层（网络层）注入数据。（然而，值得注意的是实际端到端吞吐量可能小于该速率，这可能是因为中间链路的带宽受限或因为拥塞而造成的。）

3. 因特网运输协议所不提供的服务

我们已经从 4 个方面组织了运输协议服务：可靠数据传输、吞吐量、定时和安全性。TCP 和 UDP 提供了这些服务中的哪些呢？我们已经注意到 TCP 提供了可靠的端到端数据传送。并且我们也知道 TCP 在应用层可以很容易地用 SSL 来加强以提供安全服务。但在我们对 TCP 和 UDP 的简要描述中，明显地漏掉了对吞吐量或定时保证的讨论，即这些服务目前的因特网运输协议并没有提供。这是否意味着诸如因特网电话这样的时间敏感应用不能运行在今天的因特网上呢？答案显然是否定的，因为在因特网上运行时间敏感应用已经有多年了。这些应用经常工作得相当好，因为它们已经被设计成尽最大可能对付这种保证的缺乏。我们将在第 9 章中研究几种设计技巧。无论如何，在时延过大或端到端吞吐量受限时，好的设计也是有限制的。总之，今天的因特网通常能够为时间敏感应用提供满意的服务，但它不能提供任何定时或带宽保证。

图 2-5 指出了一些流行的因特网应用所使用的运输协议。可以看到，电子邮件、远程终端访问、Web、文件传输都使用了 TCP。这些应用选择 TCP 的最主要原因是 TCP 提供了可靠数据传输服务，确保所有数据最终到达目的地。因为因特网电话应用（如 Skype）通常能够容忍某些丢失但要求达到一定的最小速率才能有效工作，所以因特网电话应用的开发者通常愿意将该应用运行在 UDP 上，从而设法避开 TCP 的拥塞控制机制和分组开销。但因为许多防火墙被配置成阻挡（大多数类型的）UDP 流量，所以因特网电话应用通常设计成如果 UDP 通信失败就使用 TCP 作为备份。

应用	应用层协议	支撑的运输协议
电子邮件	SMTP［RFC 5321］	TCP
远程终端访问	Telnet［RFC 854］	TCP
Web	HTTP［RFC 2616］	TCP
文件传输	FTP［RFC 959］	TCP
流式多媒体	HTTP（如 YouTube）	TCP
因特网电话	SIP［RFC 3261］、RTP［RFC 3550］或专用的（如 Skype）	UDP 或 TCP

图 2-5　流行的因特网应用及其应用层协议和支撑的运输协议

2.1.5 应用层协议

我们刚刚学习了通过把报文发送进套接字实现网络进程间的相互通信。但是如何构造这些报文？在这些报文中的各个字段的含义是什么？进程何时发送这些报文？这些问题将我们带进应用层协议的范围。**应用层协议**（application-layer protocol）定义了运行在不同端系统上的应用程序进程如何相互传递报文。特别是应用层协议定义了：

- 交换的报文类型，例如请求报文和响应报文。
- 各种报文类型的语法，如报文中的各个字段及这些字段是如何描述的。
- 字段的语义，即这些字段中的信息的含义。
- 确定一个进程何时以及如何发送报文，对报文进行响应的规则。

有些应用层协议是由 RFC 文档定义的，因此它们位于公共域中。例如，Web 的应用层协议 HTTP（超文本传输协议［RFC 2616］）就作为一个 RFC 可供使用。如果浏览器开发者遵从 HTTP RFC 规则，所开发出的浏览器就能访问任何遵从该文档标准的 Web 服务器并获取相应 Web 页面。还有很多别的应用层协议是专用的，有意不为公共域使用。例如，Skype 使用了专用的应用层协议。

区分网络应用和应用层协议是很重要的。应用层协议只是网络应用的一部分（尽管从我们的角度看，它是应用非常重要的一部分）。我们来看一些例子。Web 是一种客户 - 服务器应用，它允许客户按照需求从 Web 服务器获得文档。该 Web 应用有很多组成部分，包括文档格式的标准（即 HTML）、Web 浏览器（如 Firefox 和 Microsoft Internet Explorer）、Web 服务器（如 Apache、Microsoft 服务器程序），以及一个应用层协议。Web 的应用层协议是 HTTP，它定义了在浏览器和 Web 服务器之间传输的报文格式和序列。因此，HTTP 只是 Web 应用的一个部分（尽管是重要部分）。举另外一个例子，因特网电子邮件应用也有很多组成部分，包括能容纳用户邮箱的邮件服务器、允许用户读取和生成邮件的邮件客户程序（如 Microsoft Outlook）、定义电子邮件报文结构的标准、定义报文如何在服务器之间以及如何在服务器与邮件客户程序之间传递的应用层协议、定义如何对报文首部的内容进行解释的应用层协议。用于电子邮件的主要应用层协议就是 SMTP（简单邮件传输协议［RFC5321］）。因此，电子邮件的首要应用层协议 SMTP 也只是电子邮件应用的一个部分（尽管是重要部分）。

2.1.6 本书涉及的网络应用

每天都有新的公共域或者专用域因特网应用被开发出来。我们不愿像百科全书一样涉及大量的因特网应用，而是选择其中几种重要而流行的应用加以关注。在本章中我们详细讨论 5 种重要的应用：Web、文件传输、电子邮件、目录服务、流式视频和 P2P。我们首先讨论 Web 应用，不仅因为它是极为流行的应用，而且因为它的应用层协议 HTTP比较简单并且易于理解。我们接下来讨论电子邮件，这是因特网上第一个招人喜爱的应用程序。说电子邮件比 Web 更复杂，是因为它使用了多个而不是一个应用层协议。在电子邮件之后，我们学习 DNS，它为因特网提供目录服务。大多数用户不直接与 DNS 打交道，而是通过其他的应用（包括 Web、文件传输和电子邮件）间接使用它。DNS 很好地说明了一种核心的网络功能（网络名字到网络地址的转换）是怎样在因特网的应用层实现的。然后我们讨论 P2P 文件共享应用，通过讨论包括经内容分发网分发存储的视频在内的按需流式视频，结束应用层的学习。在第 9 章中，我们将涉及多媒体

应用，包括 VoIP 视频会议。

2.2　Web 和 HTTP

20 世纪 90 年代以前，因特网的主要使用者还是研究人员、学者和大学生，他们登录远程主机，在本地主机和远程主机之间传输文件，收发新闻，收发电子邮件。尽管这些应用非常有用（并且继续如此），但是因特网基本上不为学术界和研究界之外所知。到了 20 世纪 90 年代初期，一个主要的新型应用即万维网（World Wide Web）登上了舞台[Berners-Lee 1994]。Web 是一个引起公众注意的因特网应用，它极大地改变了人们与工作环境内外交流的方式。它将因特网从只是很多数据网之一的地位提升为仅有的一个数据网。

也许对大多数用户来说，最具有吸引力的就是 Web 的按需操作。当用户需要时，就能得到所想要的内容。这不同于无线电广播和电视，它们迫使用户只能收听、收看内容提供者提供的节目。除了可以按需操作以外，Web 还有很多让人们喜欢和珍爱的特性。任何人使信息在 Web 上可用都非常简单，即只需要极低的费用就能成为出版人。超链接和搜索引擎帮助我们在 Web 站点的海洋里导航。图片和视频刺激着我们的感官。表单、JavaScript、Java 小程序和很多其他的装置，使我们可以与 Web 页面和站点进行交互。并且，Web 及其协议作为平台，为 YouTube、基于 Web 的电子邮件（如 Gmail）和大多数移动因特网应用（包括 Instagram 和谷歌地图）服务。

2.2.1　HTTP 概况

Web 的应用层协议是**超文本传输协议**（HyperText Transfer Protocol，HTTP），它是 Web 的核心，在［RFC 1945］和［RFC 2616］中进行了定义。HTTP 由两个程序实现：一个客户程序和一个服务器程序。客户程序和服务器程序运行在不同的端系统中，通过交换 HTTP 报文进行会话。HTTP 定义了这些报文的结构以及客户和服务器进行报文交换的方式。在详细解释 HTTP 之前，应当回顾某些 Web 术语。

Web 页面（Web page）（也叫文档）是由对象组成的。一个**对象**（object）只是一个文件，诸如一个 HTML 文件、一个 JPEG 图形、一个 Java 小程序或一个视频片段这样的文件，且它们可通过一个 URL 地址寻址。多数 Web 页面含有一个 **HTML 基本文件**（base HTML file）以及几个引用对象。例如，如果一个 Web 页面包含 HTML 文本和 5 个 JPEG 图形，那么这个 Web 页面有 6 个对象：一个 HTML 基本文件加 5 个图形。HTML 基本文件通过对象的 URL 地址引用页面中的其他对象。每个 URL 地址由两部分组成：存放对象的服务器主机名和对象的路径名。例如，URL 地址 http://www.someSchool.edu/someDepartment/picture.gif，其中的 www.someSchool.edu 就是主机名，/someDepartment/picture.gif 就是路径名。因为 **Web 浏览器**（Web browser）（例如 Internet Explorer 和 Firefox）实现了 HTTP 的客户端，所以在 Web 环境中我们经常交替使用“浏览器”和“客户”这两个术语。**Web 服务器**（Web server）实现了 HTTP 的服务器端，它用于存储 Web 对象，每个对象由 URL 寻址。流行的 Web 服务器有 Apache 和 Microsoft Internet Information Server（微软互联网信息服务器）。

HTTP 定义了 Web 客户向 Web 服务器请求 Web 页面的方式，以及服务器向客户传送 Web 页面的方式。我们稍后详细讨论客户和服务器的交互过程，而其基本思想在

图 2-6 中进行了图示。当用户请求一个 Web 页面（如点击一个超链接）时，浏览器向服务器发出对该页面中所包含对象的HTTP请求报文，服务器接收到请求并用包含这些对象的HTTP响应报文进行响应。

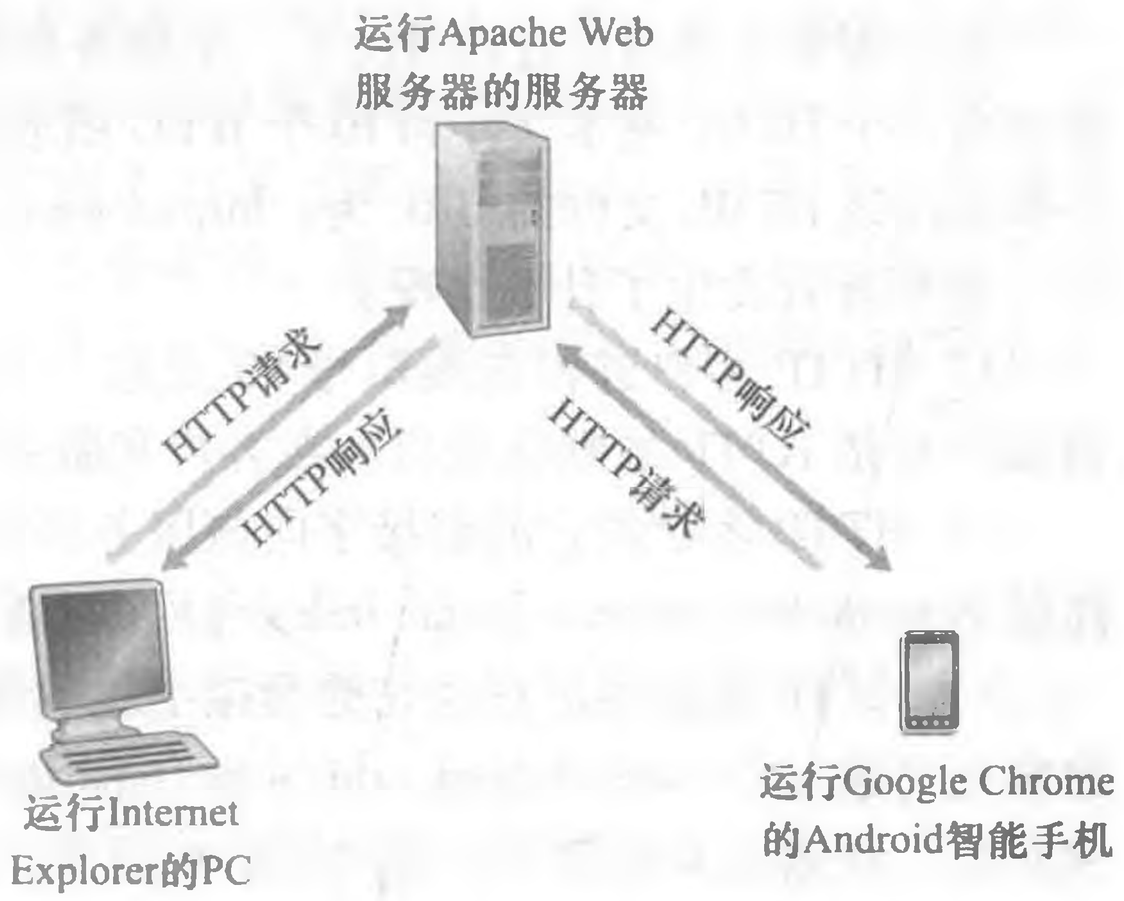

图 2-6 HTTP 的请求 - 响应行为

HTTP 使用 TCP 作为它的支撑运输协议（而不是在 UDP 上运行）。HTTP 客户首先发起一个与服务器的 TCP 连接。一旦连接建立，该浏览器和服务器进程就可以通过套接字接口访问 TCP。如同在 2.1 节中描述的那样，客户端的套接字接口是客户进程与 TCP 连接之间的门，在服务器端的套接字接口则是服务器进程与 TCP 连接之间的门。客户向它的套接字接口发送 HTTP 请求报文并从它的套接字接口接收 HTTP 响应报文。类似地，服务器从它的套接字接口接收 HTTP 请求报文和向它的套接字接口发送 HTTP 响应报文。一旦客户向它的套接字接口发送了一个请求报文，该报文就脱离了客户控制并进入 TCP 的控制。2.1 节讲过，TCP 为 HTTP 提供可靠数据传输服务。这意味着，一个客户进程发出的每个 HTTP 请求报文最终能完整地到达服务器；类似地，服务器进程发出的每个 HTTP 响应报文最终能完整地到达客户。这里我们看到了分层体系结构最大的优点，即 HTTP 协议不用担心数据丢失，也不关注 TCP 从网络的数据丢失和乱序故障中恢复的细节。那是 TCP 以及协议栈较低层协议的工作。

注意到下列现象很重要：服务器向客户发送被请求的文件，而不存储任何关于该客户的状态信息。假如某个特定的客户在短短的几秒内两次请求同一个对象，服务器并不会因为刚刚为该客户提供了该对象就不再做出反应，而是重新发送该对象，就像服务器已经完全忘记不久之前所做过的事一样。因为 HTTP 服务器并不保存关于客户的任何信息，所以我们说 HTTP 是一个**无状态协议**（stateless protocol）。我们同时也注意到 Web 使用了客户 - 服务器应用程序体系结构（如 2.1 节所述）。Web 服务器总是打开的，具有一个固定的 IP 地址，且它服务于可能来自数以百万计的不同浏览器的请求。

2.2.2 非持续连接和持续连接

在许多因特网应用程序中，客户和服务器在一个相当长的时间范围内通信，其中客户发出一系列请求并且服务器对每个请求进行响应。依据应用程序以及该应用程序的使用方式，这一系列请求可以以规则的间隔周期性地或者间断性地一个接一个发出。当这种客户 - 服务器的交互是经 TCP 进行的，应用程序的研制者就需要做一个重要决定，即每个请求/响应对是经一个单独的 TCP 连接发送，还是所有的请求及其响应经相同的 TCP 连接发送呢？采用前一种方法，该应用程序被称为使用**非持续连接**（non-persistent connection）；采用后一种方法，该应用程序被称为使用**持续连接**（persistent connection）。为了深入地理解该设计问题，我们研究在特定的应用程序即 HTTP 的情况下持续连接的优点和缺点，HTTP既能够使用非持续连接，也能够使用持续连接。尽管 HTTP 在其默认方式下使用持续连接，HTTP 客户和服务器也能配置成使用非持续连接。

1. 采用非持续连接的 HTTP

我们看看在非持续连接情况下，从服务器向客户传送一个 Web 页面的步骤。假设该页面含有一个 HTML 基本文件和 10 个 JPEG 图形，并且这 11 个对象位于同一台服务器上。进一步假设该 HTML 文件的 URL 为：http://www. someSchool. edu/someDepartment/home. index。

我们看看发生了什么情况：

1）HTTP 客户进程在端口号 80 发起一个到服务器 www. someSchool. edu 的 TCP 连接，该端口号是 HTTP 的默认端口。在客户和服务器上分别有一个套接字与该连接相关联。

2）HTTP 客户经它的套接字向该服务器发送一个 HTTP 请求报文。请求报文中包含了路径名/someDepartment/home. index（后面我们会详细讨论 HTTP 报文）。

3）HTTP 服务器进程经它的套接字接收该请求报文，从其存储器（RAM 或磁盘）中检索出对象 www. someSchool. edu/someDepartment/home. index，在一个 HTTP 响应报文中封装对象，并通过其套接字向客户发送响应报文。

4）HTTP 服务器进程通知 TCP 断开该 TCP 连接。（但是直到 TCP 确认客户已经完整地收到响应报文为止，它才会实际中断连接。）

5）HTTP 客户接收响应报文，TCP 连接关闭。该报文指出封装的对象是一个 HTML 文件，客户从响应报文中提取出该文件，检查该 HTML 文件，得到对 10 个 JPEG 图形的引用。

6）对每个引用的 JPEG 图形对象重复前 4 个步骤。

当浏览器收到 Web 页面后，向用户显示该页面。两个不同的浏览器也许会以不同的方式解释（即向用户显示）该页面。HTTP 与客户如何解释一个 Web 页面毫无关系。HTTP 规范（[RFC 1945] 和 [RFC 2616]）仅定义了在 HTTP 客户程序与 HTTP 服务器程序之间的通信协议。

上面的步骤举例说明了非持续连接的使用，其中每个 TCP 连接在服务器发送一个对象后关闭，即该连接并不为其他的对象而持续下来。值得注意的是每个 TCP 连接只传输一个请求报文和一个响应报文。因此在本例中，当用户请求该 Web 页面时，要产生 11 个 TCP 连接。

在上面描述的步骤中，我们有意没有明确客户获得这 10 个 JPEG 图形对象是使用 10 个串行的 TCP 连接，还是某些 JPEG 对象使用了一些并行的 TCP 连接。事实上，用户能够配置现代浏览器来控制连接的并行度。在默认方式下，大部分浏览器打开 5 ~ 10 个并行的 TCP 连接，而每条连接处理一个请求响应事务。如果用户愿意，最大并行连接数可以设置为 1，这样 10 条连接就会串行建立。我们在下一章会看到，使用并行连接可以缩短响应时间。

在继续讨论之前，我们来简单估算一下从客户请求 HTML 基本文件起到该客户收到整个文件止所花费的时间。为此，我们给出**往返时间**（Round-Trip Time，RTT）的定义，该时间是指一个短分组从客户到服务器然后再返回客户所花费的时间。RTT 包括分组传播时延、分组在中间路由器和交换机上的排队时延以及分组处理时延（这些时延在 1.4 节已经讨论过）。现在考虑当用户点击超链接时会发生什么现象。如图 2-7 所示，

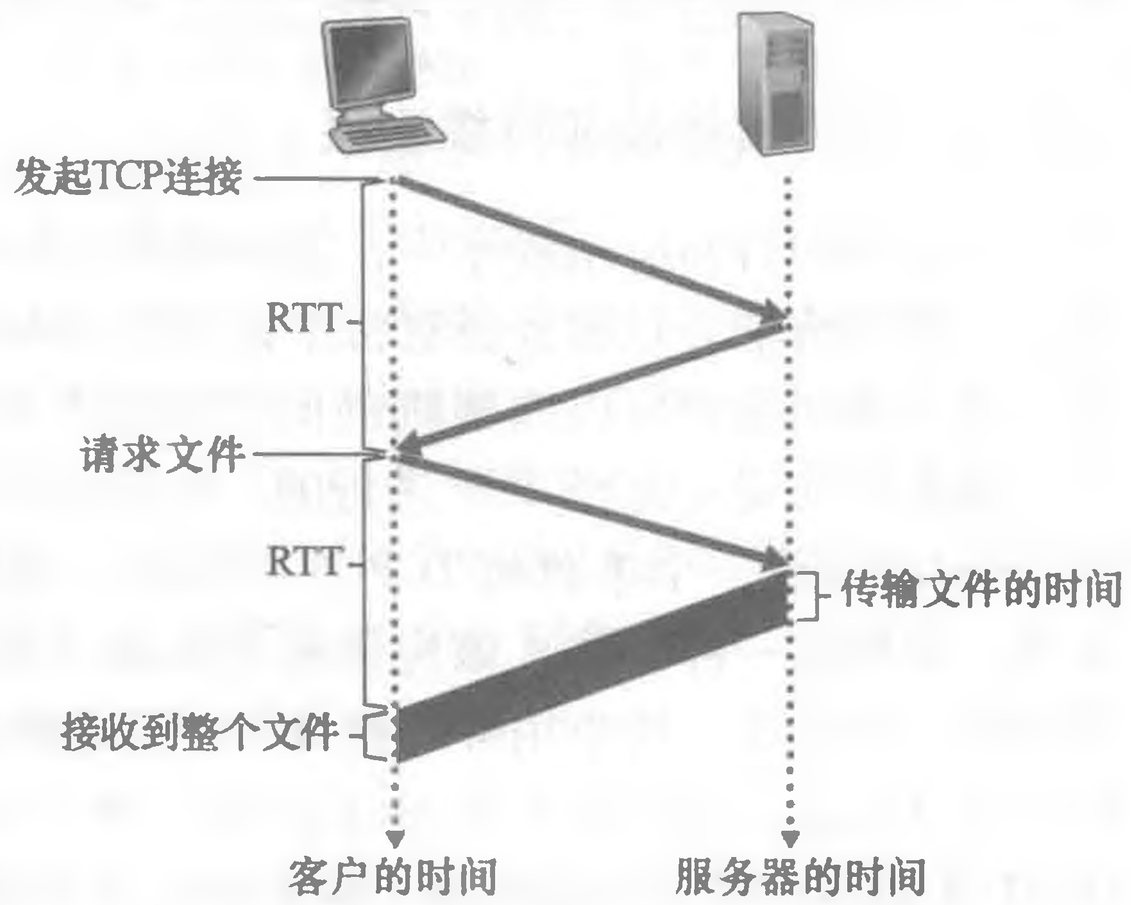

图 2-7　请求并接收一个 HTML 文件所需的时间估算

这引起浏览器在它和 Web 服务器之间发起一个 TCP 连接；这涉及一次“三次握手”过程，即客户向服务器发送一个小 TCP 报文段，服务器用一个小 TCP 报文段做出确认和响应，最后，客户向服务器返回确认。三次握手中前两个部分所耗费的时间占用了一个 RTT。完成了三次握手的前两个部分后，客户结合三次握手的第三部分（确认）向该 TCP 连接发送一个 HTTP 请求报文。一旦该请求报文到达服务器，服务器就在该 TCP 连接上发送 HTML文件。该 HTTP 请求/响应用去了另一个 RTT。因此，粗略地讲，总的响应时间就是两个 RTT 加上服务器传输 HTML 文件的时间。

2. 采用持续连接的 HTTP

非持续连接有一些缺点。第一，必须为每一个请求的对象建立和维护一个全新的连接。对于每个这样的连接，在客户和服务器中都要分配 TCP 的缓冲区和保持 TCP 变量，这给 Web 服务器带来了严重的负担，因为一台 Web 服务器可能同时服务于数以百计不同的客户的请求。第二，就像我们刚描述的那样，每一个对象经受两倍 RTT 的交付时延，即一个 RTT 用于创建 TCP，另一个 RTT 用于请求和接收一个对象。

在采用 HTTP 1.1 持续连接的情况下，服务器在发送响应后保持该 TCP 连接打开。在相同的客户与服务器之间，后续的请求和响应报文能够通过相同的连接进行传送。特别是，一个完整的 Web 页面（上例中的 HTML 基本文件加上 10 个图形）可以用单个持续 TCP 连接进行传送。更有甚者，位于同一台服务器的多个 Web 页面在从该服务器发送给同一个客户时，可以在单个持续 TCP 连接上进行。对对象的这些请求可以一个接一个地发出，而不必等待对未决请求（流水线）的回答。一般来说，如果一条连接经过一定时间间隔（一个可配置的超时间隔）仍未被使用，HTTP 服务器就关闭该连接。HTTP 的默认模式是使用带流水线的持续连接。最近，HTTP/2［RFC 7540］是在 HTTP 1.1 基础上构建的，它允许在相同连接中多个请求和回答交错，并增加了在该连接中优化 HTTP 报文请求和回答的机制。我们把量化比较持续连接和非持续连接性能的任务留作第 2、3 章的课后习题。鼓励读者阅读文献［Heidemann 1997；Nielsen 1997；RFC 7540］。

2.2.3 HTTP 报文格式

HTTP 规范［RFC 1945；RFC 2616；RFC 7540］包含了对 HTTP 报文格式的定义。HTTP 报文有两种：请求报文和响应报文。下面讨论这两种报文。

1. HTTP 请求报文

下面提供了一个典型的 HTTP 请求报文：

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
Connection: close
User-agent: Mozilla/5.0
Accept-language: fr
```

通过仔细观察这个简单的请求报文，我们就能学到很多东西。首先，我们看到该报文是用普通的 ASCII 文本书写的，这样有一定计算机知识的人都能够阅读它。其次，我们看到该报文由 5 行组成，每行由一个回车和换行符结束。最后一行后再附加一个回车换行符。虽然这个特定的报文仅有 5 行，但一个请求报文能够具有更多的行或者至少为一行。HTTP 请求报文的第一行叫作**请求行**（request line），其后继的行叫作**首部行**（header line）。请求行有 3 个字段：方法字段、URL 字段和 HTTP 版本字段。方法字段可以取几种

不同的值，包括 GET、POST、HEAD、PUT 和 DELETE。绝大部分的 HTTP 请求报文使用 GET 方法。当浏览器请求一个对象时，使用 GET 方法，在 URL 字段带有请求对象的标识。在本例中，该浏览器正在请求对象/somedir/page. html。其版本字段是自解释的；在本例中，浏览器实现的是 HTTP/1. 1 版本。

现在我们看看本例的首部行。首部行 Host：www. someschool. edu 指明了对象所在的主机。你也许认为该首部行是不必要的，因为在该主机中已经有一条 TCP 连接存在了。但是，如我们将在 2. 2. 5 节中所见，该首部行提供的信息是 Web 代理高速缓存所要求的。通过包含 Connection：close 首部行，该浏览器告诉服务器不要麻烦地使用持续连接，它要求服务器在发送完被请求的对象后就关闭这条连接。User-agent：首部行用来指明用户代理，即向服务器发送请求的浏览器的类型。这里浏览器类型是 Mozilla/5. 0，即 Firefox 浏览器。这个首部行是有用的，因为服务器可以有效地为不同类型的用户代理实际发送相同对象的不同版本。（每个版本都由相同的 URL 寻址。）最后，Accept-language：首部行表示用户想得到该对象的法语版本（如果服务器中有这样的对象的话）；否则，服务器应当发送它的默认版本。Accept-language：首部行仅是 HTTP 中可用的众多内容协商首部之一。

看过一个例子之后，我们再来看看如图 2-8 所示的一个请求报文的通用格式。我们看到该通用格式与我们前面的例子密切对应。然而，你可能已经注意到了在首部行（和附加的回车和换行）后有一个“实体体”（entity body）。使用 GET 方法时实体体为空，而使用 POST 方法时才使用该实体体。当用户提交表单时，HTTP 客户常常使用 POST 方法，例如当用户向搜索引擎提供搜索关键词时。使用 POST 报文时，用户仍可以向服务器请求一个 Web 页面，但 Web 页面的特定内容依赖于用户在表单字段中输入的内容。如果方法字段的值为 POST 时，则实体体中包含的就是用户在表单字段中的输入值。

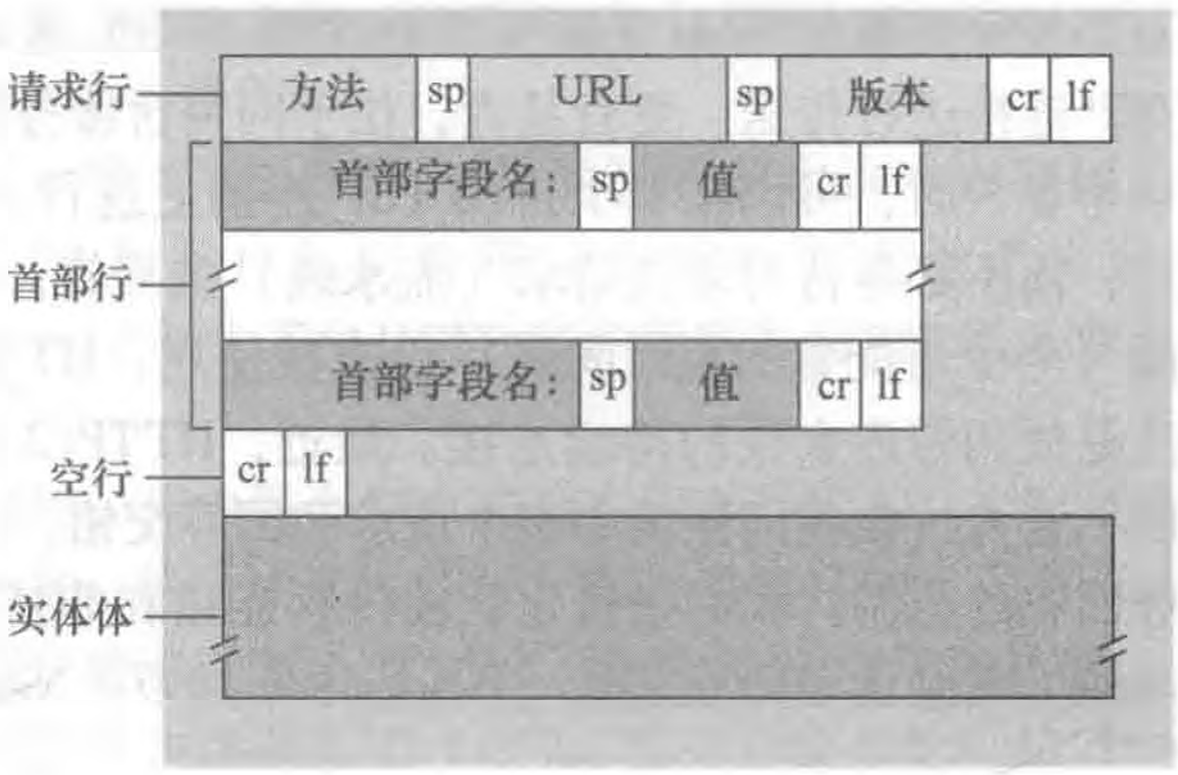

图 2-8　一个 HTTP 请求报文的通用格式

当然，如果不提“用表单生成的请求报文不是必须使用 POST 方法”这一点，那将是失职。相反，HTML 表单经常使用 GET 方法，并在（表单字段中）所请求的 URL 中包括输入的数据。例如，一个表单使用 GET 方法，它有两个字段，分别填写的是“monkeys”和“bananas”，这样，该 URL 结构为 www. somesite. com/animalsearch? monkeys&bananas。在日复一日的网上冲浪中，你也许已经留意到了这种扩展的 URL。

HEAD 方法类似于 GET 方法。当服务器收到一个使用 HEAD 方法的请求时，将会用一个 HTTP 报文进行响应，但是并不返回请求对象。应用程序开发者常用 HEAD 方法进行调试跟踪。PUT 方法常与 Web 发行工具联合使用，它允许用户上传对象到指定的 Web 服务器上指定的路径（目录）。PUT 方法也被那些需要向 Web 服务器上传对象的应用程序使用。DELETE 方法允许用户或者应用程序删除 Web 服务器上的对象。

2. HTTP 响应报文

下面我们提供了一条典型的 HTTP 响应报文。该响应报文可以是对刚刚讨论的例子中

请求报文的响应。

```
HTTP/1.1 200 OK
Connection: close
Date: Tue, 18 Aug 2015 15:44:04 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 18 Aug 2015 15:11:03 GMT
Content-Length: 6821
Content-Type: text/html

(data data data data data ...)
```

我们仔细看一下这个响应报文。它有三个部分：一个初始状态行（status line），6 个首部行（header line），然后是**实体体**（entity body）。实体体部分是报文的主要部分，即它包含了所请求的对象本身（表示为 data data data data data …）。状态行有 3 个字段：协议版本字段、状态码和相应状态信息。在这个例子中，状态行指示服务器正在使用 HTTP/1.1，并且一切正常（即服务器已经找到并正在发送所请求的对象）。

我们现在来看看首部行。服务器用 Connection：close 首部行告诉客户，发送完报文后将关闭该 TCP 连接。Date：首部行指示服务器产生并发送该响应报文的日期和时间。值得一提的是，这个时间不是指对象创建或者最后修改的时间，而是服务器从它的文件系统中检索到该对象，将该对象插入响应报文，并发送该响应报文的时间。Server：首部行指示该报文是由一台 Apache Web 服务器产生的，它类似于 HTTP 请求报文中的 User-agent：首部行。Last-Modified：首部行指示了对象创建或者最后修改的日期和时间。Last-Modified：首部行对既可能在本地客户也可能在网络缓存服务器上的对象缓存来说非常重要。我们将很快详细地讨论缓存服务器（也叫代理服务器）。Content-Length：首部行指示了被发送对象中的字节数。Content-Type：首部行指示了实体体中的对象是 HTML 文本。（该对象类型应该正式地由 Content-Type：首部行而不是用文件扩展名来指示。）

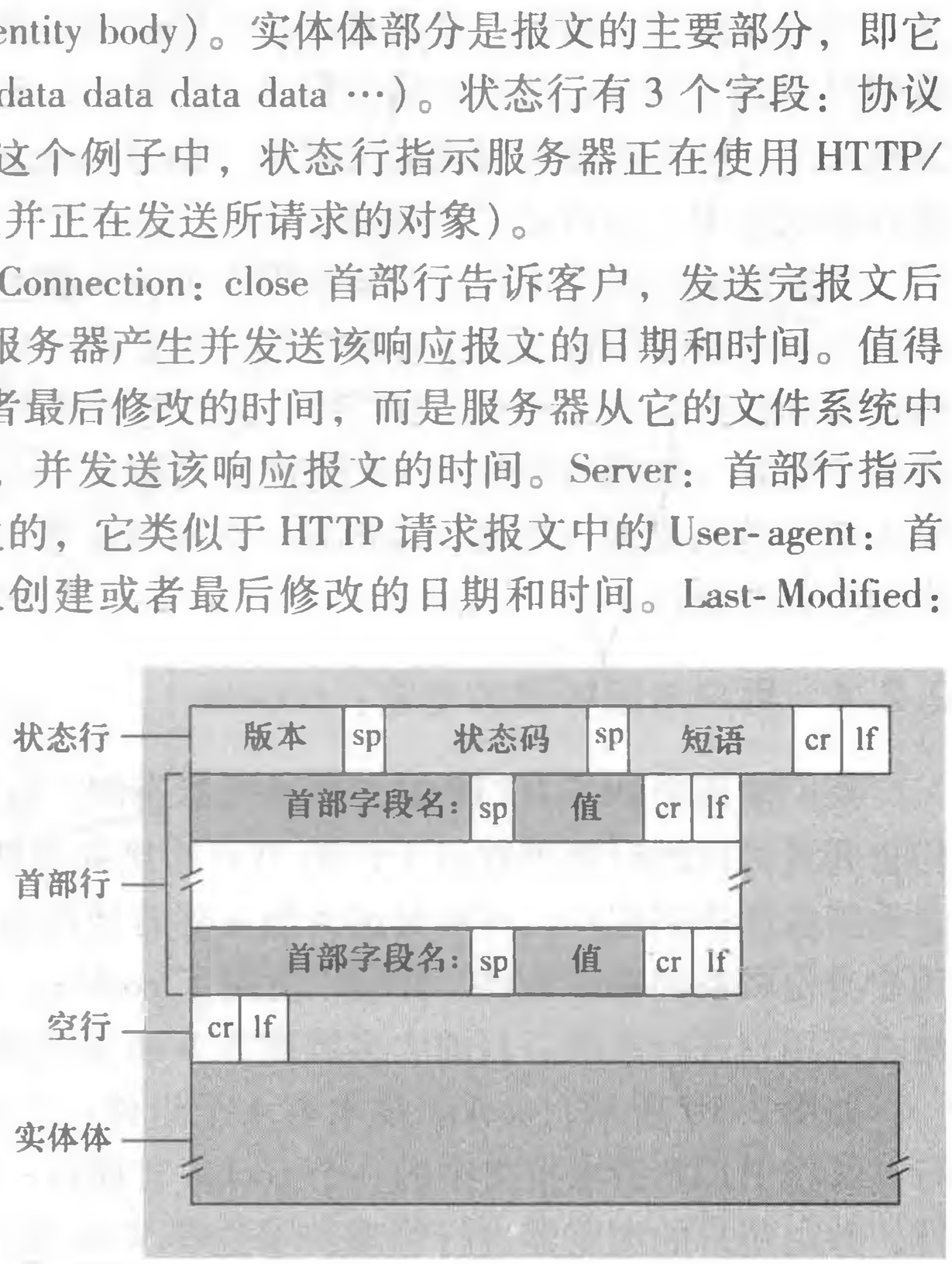

图 2-9 一个 HTTP 响应报文的通用格式

看过一个例子后，我们再来查看响应报文的通用格式，如图 2-9 所示。该通用格式与前面例子中的响应报文相匹配。我们补充说明一下状态码和它们对应的短语。状态码及其相应的短语指示了请求的结果。一些常见的状态码和相关的短语包括：

- 200 OK：请求成功，信息在返回的响应报文中。
- 301 Moved Permanently：请求的对象已经被永久转移了，新的 URL 定义在响应报文的 Location：首部行中。客户软件将自动获取新的 URL。
- 400 Bad Request：一个通用差错代码，指示该请求不能被服务器理解。
- 404 Not Found：被请求的文档不在服务器上。
- 505 HTTP Version Not Supported：服务器不支持请求报文使用的 HTTP 协议版本。

你想看一下真实的 HTTP 响应报文吗？这正是我们高度推荐而且也很容易做到的事。首先用 Telnet 登录到你喜欢的 Web 服务器上，接下来输入一个只有一行的请求报文去请求放在该服务器上的某些对象。例如，假设你看到命令提示，键入：

```
telnet gaia.cs.umass.edu 80
GET /kurose_ross/interactive/index.php HTTP/1.1
Host: gaia.cs.umass.edu
```

（在输入最后一行后连续按两次回车。）这就打开一个到主机 gaia. cs. umass. edu 的 80 端口的 TCP 连接，并发送一个 HTTP 请求报文。你将会看到一个携带包括本书交互式课后作业的基本 HTML 文件的响应报文。如果你只是想看一下 HTTP 协议的报文行，而不是获取对象本身的话，那么可以用 HEAD 代替 GET。

在本节中，我们讨论了 HTTP 请求报文和响应报文中的一些首部行。HTTP 规范中定义了许许多多的首部行，这些首部行可以被浏览器、Web 服务器和 Web 缓存服务器插入。我们只提到了全部首部行中的少数几个，在 2.2.5 节中我们讨论网络 Web 缓存时还会涉及其他几个。一本可读性很强的文献是［Krishnamurty 2001］，它对 HTTP 协议（包括它的首部行和状态码）进行了广泛讨论。

浏览器是如何决定在一个请求报文中包含哪些首部行的呢？Web 服务器又是如何决定在一个响应报文中包含哪些首部行呢？浏览器产生的首部行与很多因素有关，包括浏览器的类型和协议版本（例如，HTTP/1.0 浏览器将不会产生任何 1.1 版本的首部行）、浏览器的用户配置（如喜好的语言）、浏览器当前是否有一个缓存的但是可能超期的对象版本。Web 服务器的表现也类似：在产品、版本和配置上都有差异，所有这些都会影响响应报文中包含的首部行。

2.2.4　用户与服务器的交互：cookie

我们前面提到了 HTTP 服务器是无状态的。这简化了服务器的设计，并且允许工程师们去开发可以同时处理数以千计的 TCP 连接的高性能 Web 服务器。然而一个 Web 站点通常希望能够识别用户，可能是因为服务器希望限制用户的访问，或者因为它希望把内容与用户身份联系起来。为此，HTTP 使用了 cookie。cookie 在［RFC 6265］中定义，它允许站点对用户进行跟踪。目前大多数商务 Web 站点都使用了 cookie。

如图 2-10 所示，cookie 技术有 4 个组件：①在 HTTP 响应报文中的一个 cookie 首部行；②在 HTTP 请求报文中的一个 cookie 首部行；③在用户端系统中保留有一个 cookie 文件，并由用户的浏览器进行管理；④位于 Web 站点的一个后端数据库。使用图 2-10，我们通过一个典型的例子看看 cookie 的工作过程。假设 Susan 总是从家中 PC 使用 Internet Explorer 上网，她首次与 Amazon. com 联系。我们假定过去她已经访问过 eBay 站点。当请求报文到达该 Amazon Web 服务器时，该 Web 站点将产生一个唯一识别码，并以此作为索引在它的后端数据库中产生一个表项。接下来 Amazon Web 服务器用一个包含 Set-cookie：首部的 HTTP 响应报文对 Susan 的浏览器进行响应，其中 Set-cookie：首部含有该识别码。例如，该首部行可能是

```
Set-cookie: 1678
```

当 Susan 的浏览器收到了该 HTTP 响应报文时，它会看到该 Set-cookie：首部。该浏览器在它管理的特定 cookie 文件中添加一行，该行包含服务器的主机名和在 Set-cookie：首部中的识别码。值得注意的是该 cookie 文件已经有了用于 eBay 的表项，因为 Susan 过去访问过该站点。当 Susan 继续浏览 Amazon 网站时，每请求一个 Web 页面，其浏览器就会查询该 cookie 文件并抽取她对这个网站的识别码，并放到 HTTP 请求报文中包括识别码的 cookie 首

部行中。特别是，发往该 Amazon 服务器的每个 HTTP 请求报文都包括以下首部行：

```
Cookie: 1678
```

在这种方式下，Amazon 服务器可以跟踪 Susan 在 Amazon 站点的活动。尽管 Amazon Web 站点不必知道 Susan 的名字，但它确切地知道用户 1678 按照什么顺序、在什么时间、访问了哪些页面！Amazon 使用 cookie 来提供它的购物车服务，即 Amazon 能够维护 Susan 希望购买的物品列表，这样在 Susan 结束会话时可以一起为它们付费。

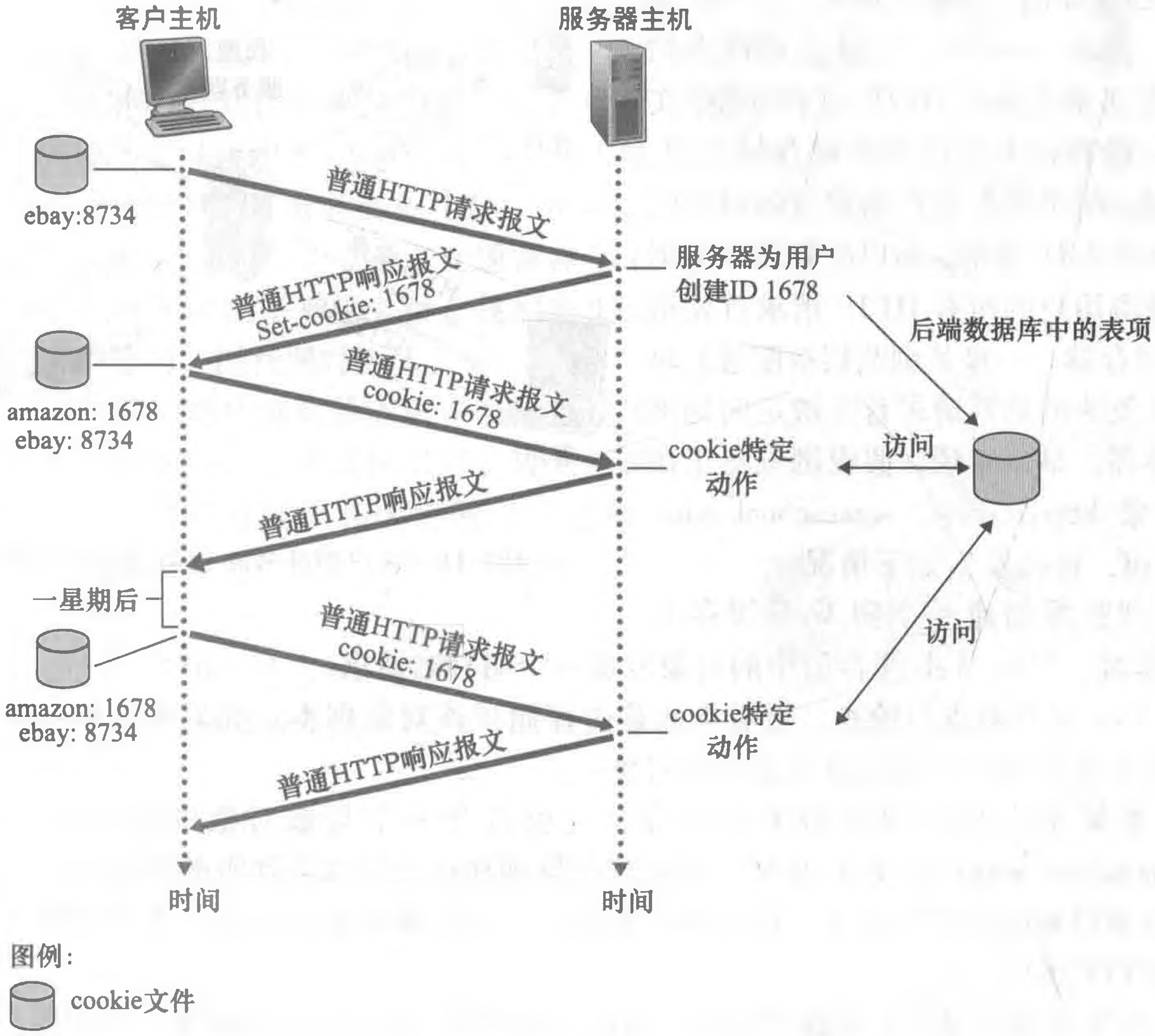

图 2-10 用 cookie 跟踪用户状态

如果 Susan 再次访问 Amazon 站点，比如说一个星期后，她的浏览器会在其请求报文中继续放入首部行 cookie：1678。Amazon 将根据 Susan 过去在 Amazon 访问的网页向她推荐产品。如果 Susan 也在 Amazon 注册过，即提供了她的全名、电子邮件地址、邮政地址和信用卡账号，则 Amazon 能在其数据库中包括这些信息，将 Susan 的名字与识别码相关联（以及她在过去访问过的本站点的所有页面）。这就解释了 Amazon 和其他一些电子商务网站实现“点击购物”（one-click shopping）的道理，即当 Susan 在后继的访问中选择购买某个物品时，她不必重新输入姓名、信用卡账号或者地址等信息了。

从上述讨论中我们看到，cookie 可以用于标识一个用户。用户首次访问一个站点时，可能需要提供一个用户标识（可能是名字）。在后继会话中，浏览器向服务器传递一个 cookie 首部，从而向该服务器标识了用户。因此 cookie 可以在无状态的 HTTP 之上建立一个用户会话层。例如，当用户向一个基于 Web 的电子邮件系统（如 Hotmail）注册时，浏览器向服务器发送 cookie 信息，允许该服务器在用户与应用程序会话的过程中标识该用户。

尽管 cookie 常常能简化用户的因特网购物活动，但是它的使用仍具有争议，因为它们被认为是对用户隐私的一种侵害。如我们刚才所见，结合 cookie 和用户提供的账户信息，Web 站点可以知道许多有关用户的信息，并可能将这些信息卖给第三方。Cookie Central [Cookie Central 2016] 包括了对 cookie 争论的广泛信息。

2.2.5　Web 缓存

Web 缓存器（Web cache）也叫**代理服务器**（proxy server），它是能够代表初始 Web 服务器来满足 HTTP 请求的网络实体。Web 缓存器有自己的磁盘存储空间，并在存储空间中保存最近请求过的对象的副本。如图 2-11 所示，可以配置用户的浏览器，使得用户的所有 HTTP 请求首先指向 Web 缓存器。一旦某浏览器被配置，每个对某对象的浏览器请求首先被定向到该 Web 缓存器。举例来说，假设浏览器正在请求对象 http://www.someschool.edu/campus.gif，将会发生如下情况：

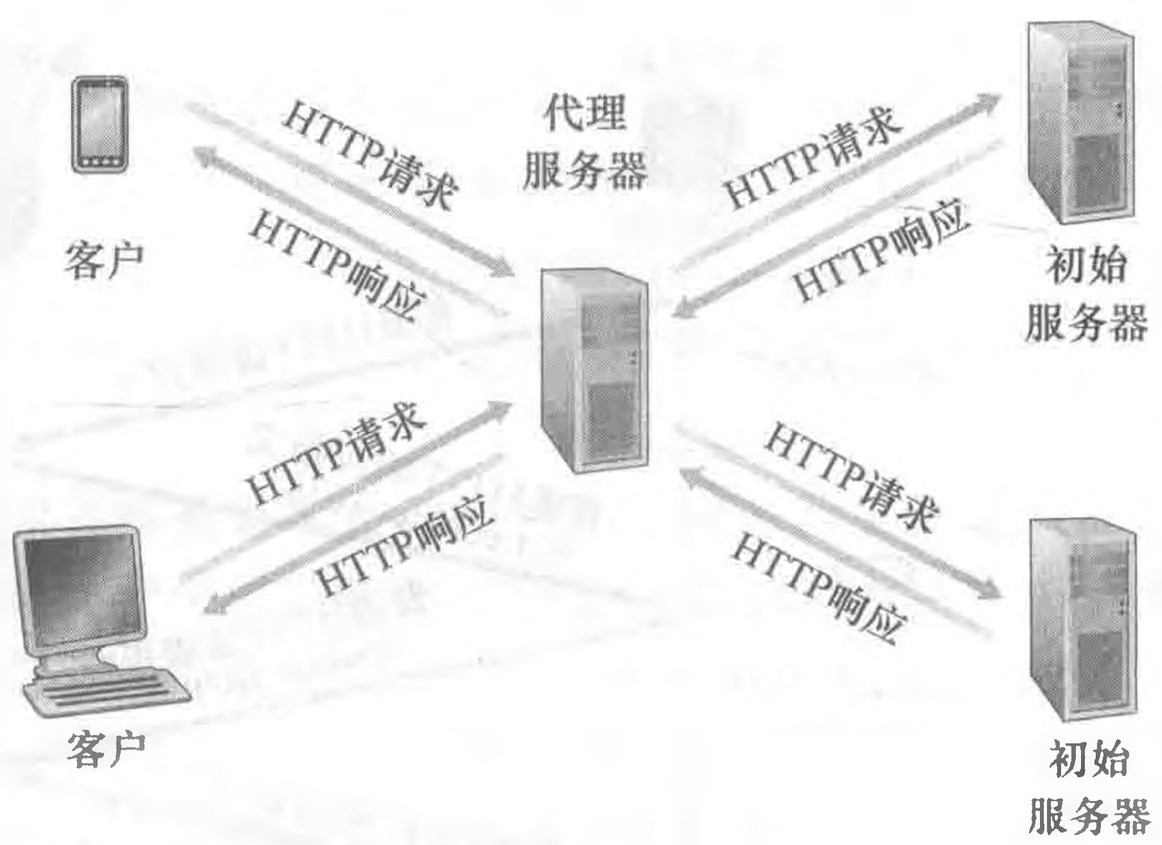

图 2-11　客户通过 Web 缓存器请求对象

1）浏览器创建一个到 Web 缓存器的 TCP 连接，并向 Web 缓存器中的对象发送一个 HTTP 请求。

2）Web 缓存器进行检查，看看本地是否存储了该对象副本。如果有，Web 缓存器就向客户浏览器用 HTTP 响应报文返回该对象。

3）如果 Web 缓存器中没有该对象，它就打开一个与该对象的初始服务器（即 www.someschool.edu）的 TCP 连接。Web 缓存器则在这个缓存器到服务器的 TCP 连接上发送一个对该对象的 HTTP 请求。在收到该请求后，初始服务器向该 Web 缓存器发送具有该对象的 HTTP 响应。

4）当 Web 缓存器接收到该对象时，它在本地存储空间存储一份副本，并向客户的浏览器用 HTTP 响应报文发送该副本（通过现有的客户浏览器和 Web 缓存器之间的 TCP 连接）。

值得注意的是 Web 缓存器既是服务器又是客户。当它接收浏览器的请求并发回响应时，它是一个服务器。当它向初始服务器发出请求并接收响应时，它是一个客户。

Web 缓存器通常由 ISP 购买并安装。例如，一所大学可能在它的校园网上安装一台缓存器，并且将所有校园网上的用户浏览器配置为指向它。或者，一个主要的住宅 ISP（例如 Comcast）可能在它的网络上安装一台或多台 Web 缓存器，并且预先配置其配套的浏览器指向这些缓存器。

在因特网上部署 Web 缓存器有两个原因。首先，Web 缓存器可以大大减少对客户请求的响应时间，特别是当客户与初始服务器之间的瓶颈带宽远低于客户与 Web 缓存器之间的瓶颈带宽时更是如此。如果在客户与 Web 缓存器之间有一个高速连接（情况常常如此），并且如果用户所请求的对象在 Web 缓存器上，则 Web 缓存器可以迅速将该对象交付给用户。其次，如我们马上用例子说明的那样，Web 缓存器能够大大减少一个机构的接入链路到因特网的通信量。通过减少通信量，该机构（如一家公司或者一所大学）就不必急于增加带宽，因此降低了费用。此外，Web 缓存器能从整体上大大减低因特网上的 Web