


时间，才能根据这些信息按照需要执行重发操作。

上面说的只是其中一个例子。套接字中记录了用于控制通信操作的各种控制信息，协议栈则需要根据这些信息判断下一步的行动，这就是套接字的作用。

 协议栈是根据套接字中记录的控制信息来工作的。

讲了这么多抽象的概念，可能大家还不太容易理解，所以下面来看看真正的套接字。在 Windows 中可以用 `netstat` 命令显示套接字内容（图 2.2）^①。图中每一行相当于一个套接字，当创建套接字时，就会在这里增加一行新的控制信息，赋予“即将开始通信”的状态，并进行通信的准备工作，如分配用于临时存放收发数据的缓冲区空间。

既然有图，我们就来讲讲图上这些到底都是什么意思。比如第 8 行，它表示 PID^② 为 4 的程序正在使用 IP 地址为 10.10.1.16 的网卡与 IP 地址为 10.10.1.18 的对象进行通信。此外我们还可以看出，本机使用 1031 端口，对方使用 139 端口，而 139 端口是 Windows 文件服务器使用的端口，因此我们能够看出这个套接字是连接到一台文件服务器的。我们再来看第 1 行，这一行表示 PID 为 984 的程序正在 135 端口等待另一方的连接，其中本地 IP 地址和远程 IP 地址都是 0.0.0.0，这表示通信还没开始，IP 地址不确定^③。

① 图中只显示了部分内容，除了图上的内容之外，套接字中还记录了其他很多种控制信息。

② PID: Process ID（进程标识符）的缩写，是操作系统为了标识程序而分配的编号，使用任务管理器可以查询所对应的程序名称。

③ 对于处于等待连接状态的套接字，也可以绑定 IP 地址，如果绑定了 IP 地址，那么除绑定的 IP 地址之外，对其他地址进行连接操作都会出错。当服务器上安装有多块网卡时，可以用这种方式来限制只能连接到特定的网卡。

netstat是用于显示套接字内容的命令，-ano选项表示下面的意思。

- a 不仅显示正在通信的套接字，还显示包括尚未开始通信等状态的所有套接字
- n 显示IP地址和端口号
- o 显示使用该套接字的程序PID

命令提示符

```
C:\>netstat -ano
```

Active Connections

Proto	Local Address	Foreign Address	State	PID
TCP	0.0.0.0:135	0.0.0.0:0	LISTENING	984
TCP	0.0.0.0:445	0.0.0.0:0	LISTENING	4
TCP	0.0.0.0:1025	0.0.0.0:0	LISTENING	1128
TCP	0.0.0.0:1040	0.0.0.0:0	LISTENING	4
TCP	0.0.0.0:5000	0.0.0.0:0	LISTENING	1336
TCP	10.10.1.16:139	0.0.0.0:0	LISTENING	4
TCP	10.10.1.16:1031	0.0.0.0:0	LISTENING	4
TCP	10.10.1.16:1031	10.10.1.80:139	ESTABLISHED	4
TCP	10.10.1.16:1362	0.0.0.0:0	LISTENING	4
TCP	10.10.1.16:1362	10.10.1.166:139	ESTABLISHED	4
TCP	192.0.2.229:139	0.0.0.0:0	LISTENING	4
UDP	0.0.0.0:135	0.0.0.0:0		984
UDP	0.0.0.0:135	0.0.0.0:0		984

使用该套接字的程序PID（进程标识符）。可以使用任务管理器来查询PID对应的程序名称，不过任务管理器默认是不显示PID的，需要在“查看”→“选择列”中设置显示PID

表示通信状态

LISTENING等待对方连接的状态

ESTABLISHED完成连接并正在进行数据通信的状态表

通信对象（远端）的IP地址和端口号。0.0.0.0表示还没有开始通信，没有绑定IP地址和端口号。此外，UDP协议中的套接字不绑定对方的地址和端口，因此这里显示*:*

运行netstat命令的计算机本身（本地端）的IP地址和端口号。本例中的计算机上安装有多块网卡，因此会显示出多个IP地址。0.0.0.0表示不绑定IP地址

协议类型。使用TCP/IP协议通信的情况下，会显示TCP或UDP

图 2.2 显示套接字内容

2.1.3 调用 socket 时的操作

看过套接字的具体样子之后，我们的探索之旅将继续前进，看一看当浏览器调用 socket^①、connect 等 Socket 库中的程序组件时，协议栈内部是如何工作的。

首先，我们再来看一下浏览器通过 Socket 库向协议栈发出委托的一系列操作（图 2.3）。这张图和介绍浏览器时用的那张图的内容大体相同，只作了少许修改。正如我们之前讲过的那样，浏览器委托协议栈使用 TCP 协议来收发数据^②，因此下面的讲解都是关于 TCP 的。

首先是创建套接字的阶段^③。如图 2.3 ①所示，应用程序调用 socket 申请创建套接字，协议栈根据应用程序的申请执行创建套接字的操作。

在这个过程中，协议栈首先会分配用于存放一个套接字所需的内存空间。用于记录套接字控制信息的内存空间并不是一开始就存在的，因此我们先要开辟出这样一块空间来^④，这相当于为控制信息准备一个容器。但光一个容器并没有什么用，还需要往里面存入控制信息。套接字刚刚创建时，数据收发操作还没有开始，因此需要在套接字的内存空间中写入表示这一初始状态的控制信息。到这里，创建套接字的操作就完成了。

① socket：大写字母开头的 Socket 表示 Socket 库，而小写字母开头的 socket 表示 Socket 库中名为 socket 的程序组件。

② 关于为什么要使用 TCP，以及 TCP 和 UDP 的区别，我们将在后面讲解。

③ 图 2.3 最开始的调用 gethostbyname（解析器）向 DNS 服务器发送查询消息的部分在第 1 章关于 DNS 的内容中已经讲过，此处省略。

④ 计算机内部会同时运行多个程序，如果每个程序都擅自使用内存空间的话，就有可能发生多个程序重复使用同一个内存区域导致数据损坏的问题。为了避免出现这样的问题，操作系统中有一个“内存管理”模块，它相当于内存的管理员，负责根据程序的申请分配相应的内存空间，并确保这些内存空间不会被其他程序使用。因此，分配内存的操作就是向内存管理模块提出申请，请它划分一块内存空间出来。

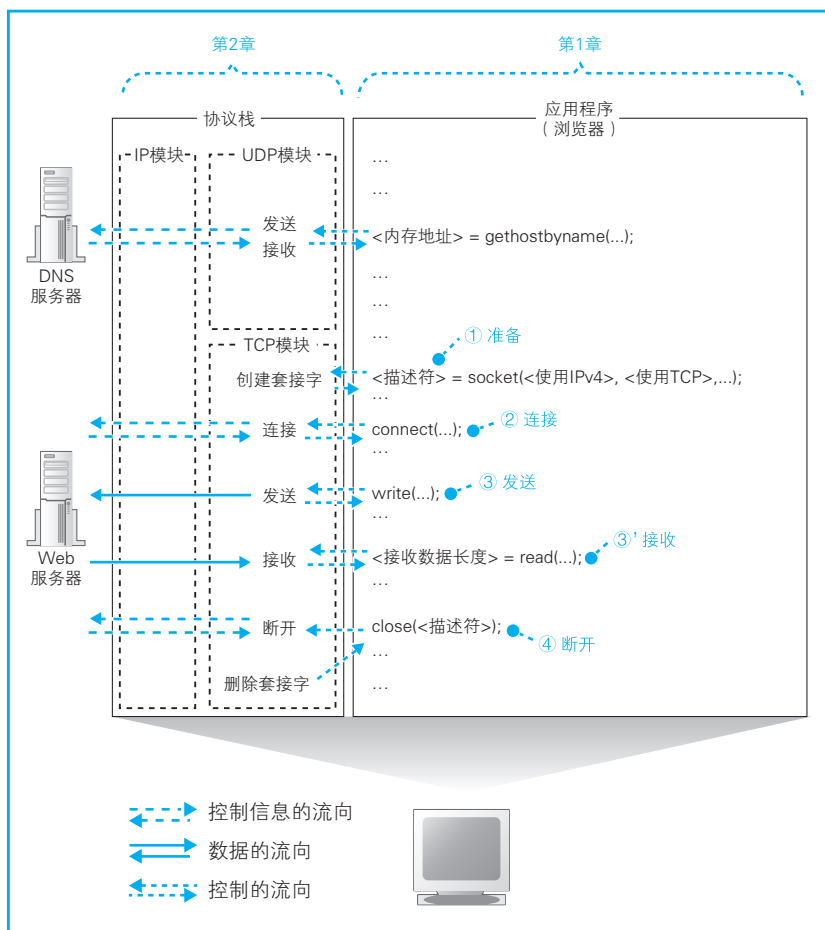


图 2.3 消息收发操作

创建套接字时，首先分配一个套接字所需的内存空间，然后向其中写入初始状态。

接下来，需要将表示这个套接字的描述符告知应用程序。描述符相当

于用来区分协议栈中的多个套接字的号码牌^①。

收到描述符之后，应用程序在向协议栈进行收发数据委托时就需要提供这个描述符。由于套接字中记录了通信双方的信息以及通信处于怎样的状态，所以只要通过描述符确定了相应的套接字，协议栈就能够获取所有的相关信息，这样一来，应用程序就不需要每次都告诉协议栈应该和谁进行通信了。

2.2 连接服务器

2.2.1 连接是什么意思

创建套接字之后，应用程序（浏览器）就会调用 `connect`，随后协议栈会将本地的套接字与服务器的套接字进行连接。话说，以太网的网线都是一直连接的状态，我们并不需要来回插拔网线，那么这里的“连接”到底是什么意思呢？连接实际上是通信双方交换控制信息，在套接字中记录这些必要信息并准备数据收发的一连串操作，在讲解具体的过程之前，我们先来说一说“连接”到底代表什么意思。

网线是一直连接着的，随时都有信号从中流过，如果通信过程只是将数据转换为电信号，那么这一操作随时都可以进行。不过，在这个时间点，也就是套接字刚刚创建完成时，当应用程序委托发送数据的时候，协议栈会如何操作呢？

套接字刚刚创建完成的时候，里面并没有存放任何数据，也不知道通信的对象是谁。在这个状态下，即便应用程序要求发送数据，协议栈也不知道数据应该发送给谁。浏览器可以根据网址来查询服务器的 IP 地址，而且根据规则也知道应该使用 80 号端口，但只有浏览器知道这些必要的信息是不够的，因为在调用 `socket` 创建套接字时，这些信息并没有传递给协议栈。因此，我们需要把服务器的 IP 地址和端口号等信息告知协议栈，这是

^① 1.4.2 节有相关介绍。

连接操作的目的之一。

那么，服务器这边又是怎样的情况呢？服务器上也会创建套接字^①，但服务器上的协议栈和客户端一样，只创建套接字是不知道应该和谁进行通信的。而且，和客户端不同的是，在服务器上，连应用程序也不知道通信对象是谁，这样下去永远也没法开始通信。于是，我们需要让客户端向服务器告知必要的信息，比如“我想和你开始通信，我的 IP 地址是 xxx.xxx.xxx.xxx，端口号是 yyyy。”可见，客户端向服务器传达开始通信的请求，也是连接操作的目的之一。

之前我们讲过，连接实际上是通信双方交换控制信息，在套接字中记录这些必要信息并准备数据收发的一连串操作，像上面提到的客户端将 IP 地址和端口号告知服务器这样的过程就属于交换控制信息的一个具体的例子。所谓控制信息，就是用来控制数据收发操作所需的一些信息，IP 地址和端口号就是典型的例子。除此之外还有其他一些控制信息，我们后面会逐一进行介绍。连接操作中所交换的控制信息是根据通信规则来确定的，只要根据规则执行连接操作，双方就可以得到必要的信息从而完成数据收发的准备。此外，当执行数据收发操作时，我们还需要一块用来临时存放要收发的数据的内存空间，这块内存空间称为缓冲区，它也是在连接操作的过程中分配的。上面这些就是“连接”^②这个词代表的具体含义。

① 服务器程序一般会在系统启动时就创建套接字并等待客户端连接，关于服务器的工作原理我们会在第 6 章进行介绍。

② 使用“连接”这个词是有原因的。通信技术的历史已经有 100 多年，从通信技术诞生之初到几年之前的很长一段时间内，电话技术一直都是主流。而电话的操作过程分为三个阶段：(1) 拨号与对方连接；(2) 通话；(3) 挂断。人们将电话的思路套用在现在的计算机网络中了，所以也就自然而然地将通信开始之前的准备操作称为“连接”了。如果没有这段历史的话，说不定现在我们就叫“连接”而是叫“准备”了。因此，如果觉得“连接”这个词听起来有些怪，那么用“准备”这个词来替换也问题不大。

2.2.2 负责保存控制信息的头部

关于控制信息，这里再补充一些。之前我们说的控制信息其实可以大体上分为两类。

第一类是客户端和服务端相互联络时交换的控制信息。这些信息不仅连接时需要，包括数据收发和断开连接操作在内，整个通信过程中都需要，这些内容在 TCP 协议的规格中进行了定义。具体来说，表 2.1 中的这些字段就是 TCP 规格中定义的控制信息^①。这些字段是固定的，在连接、收发、断开等各个阶段中，每次客户端和服务端之间进行通信时，都需要提供这些控制信息。具体来说，如图 2.4 (a) 所示，这些信息会被添加在客户端与服务端之间传递的网络包的开头。在连接阶段，由于数据收发还没有开始，所以如图 2.4 (b) 所示，网络包中没有实际的数据，只有控制信息。这些控制信息位于网络包的开头，因此被称为头部。此外，以太网和 IP 协议也有自己的控制信息，这些信息也叫头部，为了避免各种不同的头部发生混淆，我们一般会记作 TCP 头部、以太网头部^②、IP 头部。

客户端和服务端在通信中会将必要的信息记录在头部并相互确认，例如下面这样。

发送方：“开始数据发送。”

接收方：“请继续。”

发送方：“现在发送的是 $\times \times$ 号数据。”

接收方：“ $\times \times$ 号数据已收到。”

……（以下省略）

正是有了这样的交互过程，双方才能够进行通信。头部的信息非常重要，理解了头部各字段的含义，就等于理解了整个通信的过程。在后面介绍协议栈的工作过程时，我们将根据需要讲解头部各字段的含义，现在大

① 这张表中只列出了必需字段，TCP 协议规格中还定义了另外一些可选字段。

② 以太网头部又称“MAC 头部”。

表 2.1 TCP 头部格式

	字段名称	长度 (比特)	含 义
TCP 头部 (20 字节 ~)	发送方端口号	16	发送网络包的程序的端口号
	接收方端口号	16	网络包的接收方程序的端口号
	序号 (发送数据的顺序编号)	32	发送方告知接收方该网络包发送的数据相当于所有发送数据的第几个字节
	ACK 号 (接收数据的顺序编号)	32	接收方告知发送方接收方已经收到了所有数据的第几个字节。其中, ACK 是 acknowledge 的缩写
	数据偏移量	4	表示数据部分的起始位置, 也可以认为表示头部的长度
	保留	6	该字段为保留, 现在未使用
	控制位	6	该字段中的每个比特分别表示以下通信控制含义。 URG: 表示紧急指针字段有效 ACK: 表示接收数据序号字段有效, 一般表示数据已被接收方收到 PSH: 表示通过 flush 操作发送的数据 RST: 强制断开连接, 用于异常中断的情况 SYN: 发送方和接收方相互确认序号, 表示连接操作 FIN: 表示断开连接
	窗口	16	接收方告知发送方窗口大小(即无需等待确认可一起发送的数据量)
	校验和	16	用来检查是否出现错误
	紧急指针	16	表示应紧急处理的数据位置
	可选字段	可变 长度	除了上面的固定头部字段之外, 还可以添加可选字段, 但除了连接操作之外, 很少使用可选字段

家只要先记住头部是用来记录和交换控制信息的就可以了。

控制信息还有另外一类, 那就是保存在套接字中, 用来控制协议栈操作的信息^①。应用程序传递来的信息以及从通信对象接收到的信息都会保存

^① 前面已经讲过, 这些信息保存在协议栈中的套接字内存空间中。

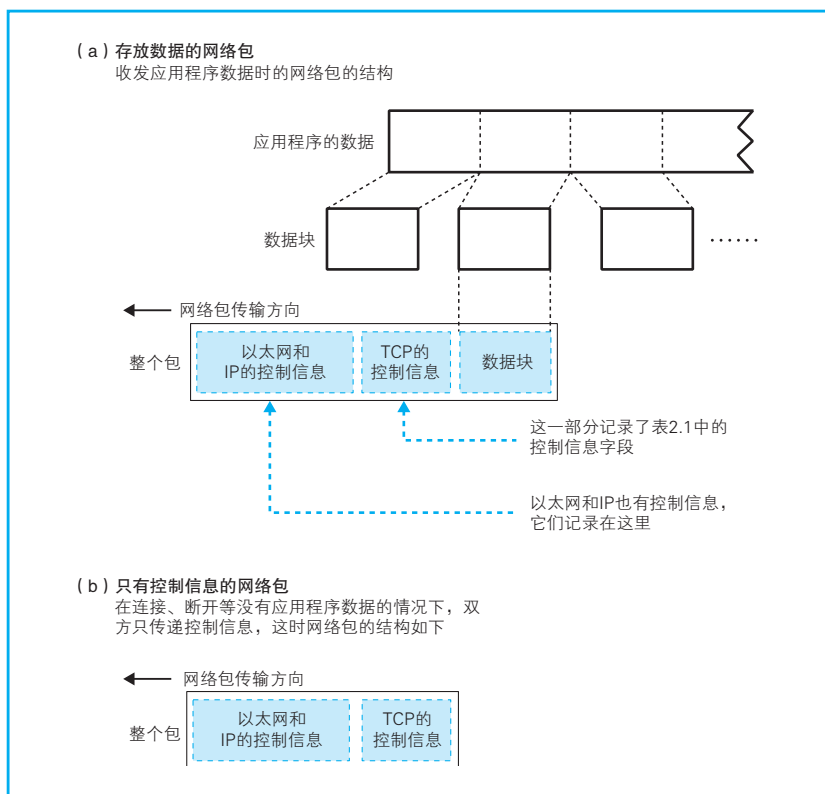


图 2.4 客户端与服务器之间交换的控制信息

在这里，还有收发数据操作的执行状态等信息也会保存在这里，协议栈会根据这些信息来执行每一步的操作。我们可以说，套接字的控制信息和协议栈的程序本身其实是一体的，因此，“协议栈具体需要哪些信息”会根据协议栈本身的实现方式不同而不同^①，但这并没有什么问题。因为协议栈中的控制信息通信对方是看不见的，只要在通信时按照规则将必要的信息写入头部，客户端和服务端之间的通信就能够得以成立。例如，Windows 和 Linux 操作系统的内部结构不同，协议栈的实现方式不同，必要的控制信

① 无论协议栈的实现如何不同，IP 地址和端口号这些重要的信息都是共通的。

息也就不同。但即便如此，两种系统之间依然能够互相通信，同样地，计算机和手机之间也能够互相通信。正如前面所说，协议栈的实现不同，因此我们无法具体说明协议栈里到底保存了哪些控制信息，但可以用命令来显示一些重要的套接字控制信息（图 2.2），这些信息无论何种操作系统的协议栈都是共通的，通过理解这些重要信息，就能够理解协议栈的工作方式了。



通信操作中使用的控制信息分为两类。

(1) 头部中记录的信息

(2) 套接字（协议栈中的内存空间）中记录的信息




2.2.3 连接操作的实际过程

我们已经了解了连接操作的含义，下面来看一下具体的操作过程。这个过程是从应用程序调用 Socket 库的 `connect` 开始的（图 2.3 ②）。


`connect(<描述符>, <服务器 IP 地址和端口号>, ...)`

上面的调用提供了服务器的 IP 地址和端口号，这些信息会传递给协议栈中的 TCP 模块。然后，TCP 模块会与该 IP 地址对应的对象，也就是与服务器的 TCP 模块交换控制信息，这一交互过程包括下面几个步骤。首先，客户端先创建一个包含表示开始数据收发操作的控制信息的头部。如表 2.1 所示，头部包含很多字段，这里要关注的重点是发送方和接收方的端口号。到这里，客户端（发送方）的套接字就准确找到了服务器（接收方）的套接字，也就是搞清楚了我应该连接哪个套接字。然后，我们将头部中的控制位的 SYN 比特设置为 1，大家可以认为它表示连接^①。此外还需要设置适当的序号和窗口大小，这一点我们会稍后详细讲解。

① SYN 比特的含义我们将在后面介绍序号时讲解。



连接操作的第一步是在 TCP 模块处创建表示连接控制信息的头部。



通过 TCP 头部中的发送方和接收方端口号可以找到要连接的套接字。

当 TCP 头部创建好之后,接下来 TCP 模块会将信息传递给 IP 模块并委托它进行发送^①。IP 模块执行网络包发送操作后,网络包就会通过网络到达服务器,然后服务器上的 IP 模块会将接收到的数据传递给 TCP 模块,服务器的 TCP 模块根据 TCP 头部中的信息找到端口号对应的套接字,也就是说,从处于等待连接状态的套接字中找到与 TCP 头部中记录的端口号相同的套接字就可以了。当找到对应的套接字之后,套接字中会写入相应的信息,并将状态改为正在连接^②。上述操作完成后,服务器的 TCP 模块会返回响应,这个过程和客户端一样,需要在 TCP 头部中设置发送方和接收方端口号以及 SYN 比特^③。此外,在返回响应时还需要将 ACK 控制位设为 1^④,这表示已经接收到相应的网络包。网络中经常会发生错误,网络包也会发生丢失,因此双方在通信时必须相互确认网络包是否已经送达^⑤,而设置 ACK 比特就是用来进行这一确认的。接下来,服务器 TCP 模块会将 TCP 头部传递给 IP 模块,并委托 IP 模块向客户端返回响应。

然后,网络包就会返回到客户端,通过 IP 模块到达 TCP 模块,并通过 TCP 头部的信息确认连接服务器的操作是否成功。如果 SYN 为 1 则表

① IP 模块接到委托并发送网络包的实际操作过程我们将稍后讲解。

② 与此相关的操作我们将在第 6 章探索服务器内部时讲解。

③ 如果由于某些原因不接受连接,那么将不设置 SYN,而是将 RST 比特设置为 1。

④ 客户端向服务器发送第一个网络包时,由于服务器还没有接收过网络包,所以需要 ACK 比特设为 0。

⑤ 相互确认的具体过程我们将稍后讲解。

示连接成功，这时会向套接字中写入服务器的 IP 地址、端口号等信息，同时还会将状态改为连接完毕。到这里，客户端的操作就已经完成，但其实还剩下最后一个步骤。刚才服务器返回响应时将 ACK 比特设置为 1，相应地，客户端也需要将 ACK 比特设置为 1 并发回服务器，告诉服务器刚才的响应包已经收到。当这个服务器收到这个返回包之后，连接操作才算全部完成。

现在，套接字就已经进入随时可以收发数据的状态了，大家可以认为这时有一根管子把两个套接字连接了起来。当然，实际上并不存在这么一根管子，不过这样想比较容易理解，网络业界也习惯这样来描述。这根管子，我们称之为连接^①。只要数据传输过程在持续，也就是在调用 close 断开之前，连接是一直存在的。

建立连接之后，协议栈的连接操作就结束了，也就是说 connect 已经执行完毕，控制流程被交回到应用程序。

2.3 收发数据

2.3.1 将 HTTP 请求消息交给协议栈

当控制流程从 connect 回到应用程序之后，接下来就进入数据收发阶段了。数据收发操作是从应用程序调用 write 将要发送的数据交给协议栈开始的(图 2.3 ③)，协议栈收到数据后执行发送操作，这一操作包含如下要点。

首先，协议栈并不关心应用程序传来的数据是什么内容。应用程序在调用 write 时会指定发送数据的长度，在协议栈看来，要发送的数据就是**一定长度的二进制字节序列**而已。

其次，协议栈并不是一收到数据就马上发送出去，而是会将数据存放在内部的发送缓冲区中，并等待应用程序的下一段数据。这样做是有道理

^① 这里的“连接”是一个名词，对应英文的 Connection。也有人把连接称为“会话”(session)，它们的意思大体上相同。

的。应用程序交给协议栈发送的数据长度是由应用程序本身来决定的，不同的应用程序在实现上有所不同，有些程序会一次性传递所有的数据，有些程序则会逐字节或者逐行传递数据。总之，一次将多少数据交给协议栈是由应用程序自行决定的，协议栈并不能控制这一行为。在这样的情况下，如果一收到数据就马上发送出去，就可能会发送大量的小包，导致网络效率下降，因此需要在数据积累到一定量时再发送出去。至于要积累多少数据才能发送，不同种类和版本的操作系统会有所不同，不能一概而论，但都是根据下面几个要素来判断的。

第一个判断要素是每个网络包能容纳的数据长度，协议栈会根据一个叫作 MTU^① 的参数来进行判断。MTU 表示一个网络包的最大长度，在以太网中一般是 1500 字节（图 2.5）^②。MTU 是包含头部的总长度，因此需要从 MTU 减去头部的长度，然后得到的长度就是一个网络包中所能容纳的最大数据长度，这一长度叫作 MSS^③。当从应用程序收到的数据长度超过或者接近 MSS 时再发送出去，就可以避免发送大量小包的问题了。

MTU：一个网络包的最大长度，以太网中一般为 1500 字节。

MSS：除去头部之后，一个网络包所能容纳的 TCP 数据的最大长度。

另一个判断要素是时间。当应用程序发送数据的频率不高的时候，如果每次都等到长度接近 MSS 时再发送，可能会因为等待时间太长而造成发

① MTU: Maximum Transmission Unit, 最大传输单元。——编者注

② 在使用 PPPoE 的 ADSL 等网络中，需要额外增加一些头部数据，因此 MTU 会小于 1500 字节。关于 PPPoE，我们将在 4.3.2 节进行讲解。

③ MSS: Maximum Segment Size, 最大分段大小。TCP 和 IP 的头部加起来一般是 40 字节，因此 MTU 减去这个长度就是 MSS。例如，在以太网中，MTU 为 1500，因此 MSS 就是 1460。TCP/IP 可以使用一些可选参数（protocol option），如加密等，这时头部的长度会增加，那么 MSS 就会随着头部长度增加而相应缩短。

送延迟，这种情况下，即便缓冲区中的数据长度没有达到 MSS，也应该果断发送出去。为此，协议栈的内部有一个计时器，当经过一定时间之后，就会把网络包发送出去^①。

判断要素就是这两个，但它们其实是互相矛盾的。如果长度优先，那么网络的效率会提高，但可能会因为等待填满缓冲区而产生延迟；相反地，如果时间优先，那么延迟时间会变少，但又会降低网络的效率。因此，在进行发送操作时需要综合考虑这两个要素以达到平衡。不过，TCP 协议规格中并没有告诉我们怎样才能平衡，因此实际如何判断是由协议栈的开发者来决定的，也正是由于这个原因，不同种类和版本的操作系统在相关操作上也存在差异。

正如前面所说，如果仅靠协议栈来判断发送的时机可能会带来一些问题，因此协议栈也给应用程序保留了控制发送时机的余地。应用程序在发送数据时可以指定一些选项，比如如果指定“不等待填满缓冲区直接发送”，则协议栈就会按照要求直接发送数据。像浏览器这种会话型的应用程序在向服务器发送数据时，等待填满缓冲区导致延迟会产生很大影响，因此一般会使用直接发送的选项。

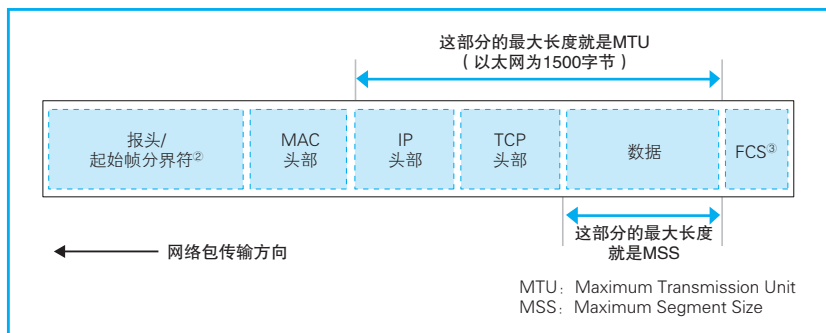


图 2.5 MTU 与 MSS

① 这个时间并没有多长，是以毫秒为单位来计算的。

② 起始帧分界符: Start Frame Delimiter, SFD。——编者注

③ FCS: Frame Check Sequence, 帧校验序列。——编者注

2.3.2 对较大的数据进行拆分

HTTP 请求消息一般不会很长，一个网络包就能装得下，但如果其中要提交表单数据，长度就可能超过一个网络包所能容纳的数据量，比如在博客或者论坛上发表一篇长文就属于这种情况。

这种情况下，发送缓冲区中的数据就会超过 MSS 的长度，这时我们当然不需要继续等待后面的数据了。发送缓冲区中的数据会被以 MSS 长度为单位进行拆分，拆分出来的每块数据会被放进单独的网络包中。

根据发送缓冲区中的数据拆分的情况，当判断需要发送这些数据时，就在每一块数据前面加上 TCP 头部，并根据套接字中记录的控制信息标记发送方和接收方的端口号，然后交给 IP 模块来执行发送数据的操作（图 2.6）^①。

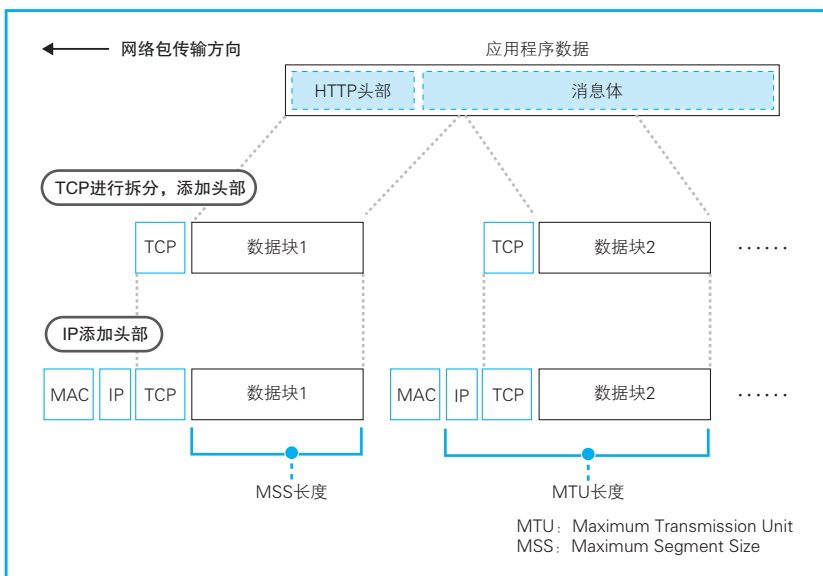


图 2.6 应用程序数据的拆分发送

应用程序的数据一般都比较大，因此 TCP 会按照网络包的大小对数据进行拆分。

^① IP 模块会在网络包前面添加 IP 头部和以太网的 MAC 头部后发送网络包，这些操作我们将稍后讲解。