2.3.3 使用 ACK 号确认网络包已收到

到这里，网络包已经装好数据并发往服务器了，但数据发送操作还没有结束。TCP 具备确认对方是否成功收到网络包，以及当对方没收到时进行重发的功能，因此在发送网络包之后，接下来还需要进行确认操作。

我们先来看一下确认的原理（图 2.7）。首先，TCP 模块在拆分数据时，会先算好每一块数据相当于从头开始的第几个字节，接下来在发送这一块数据时，将算好的字节数写在 TCP 头部中，“序号”字段就是派在这个用场上的。然后，发送数据的长度也需要告知接收方，不过这个并不是放在 TCP 头部里面的，因为用整个网络包的长度减去头部的长度就可以得到数据的长度，所以接收方可以用这种方法来进行计算。有了上面两个数值，我们就可以知道发送的数据是从第几个字节开始，长度是多少了。

通过这些信息，接收方还能够检查收到的网络包有没有遗漏。例如，假设上次接收到第 1460 字节，那么接下来如果收到序号为 1461 的包，说明中间没有遗漏；但如果收到的包序号为 2921，那就说明中间有包遗漏了。像这样，如果确认没有遗漏，接收方会将到目前为止接收到的数据长度加起来，计算出一共已经收到了多少个字节，然后将这个数值写入 TCP 头部的 ACK 号中发送给发送方[①]。简单来说，发送方说的是“现在发送的是从第 ×× 字节开始的部分，一共有 ×× 字节哦!”而接收方则回复说，“到第 ×× 字节之前的数据我已经都收到了哦!”这个返回 ACK 号的操作被称为确认响应，通过这样的方式，发送方就能够确认对方到底收到了多少数据。

然而，图 2.7 的例子和实际情况还是有些出入的。在实际的通信中，序号并不是从 1 开始的，而是需要用随机数计算出一个初始值，这是因为如果序号都从 1 开始，通信过程就会非常容易预测，有人会利用这一点来

① 返回 ACK 号时，除了要设置 ACK 号的值以外，还需要将控制位中的 ACK 比特设为 1，这代表 ACK 号字段有效，接收方也就可以知道这个网络包是用来告知 ACK 号的。

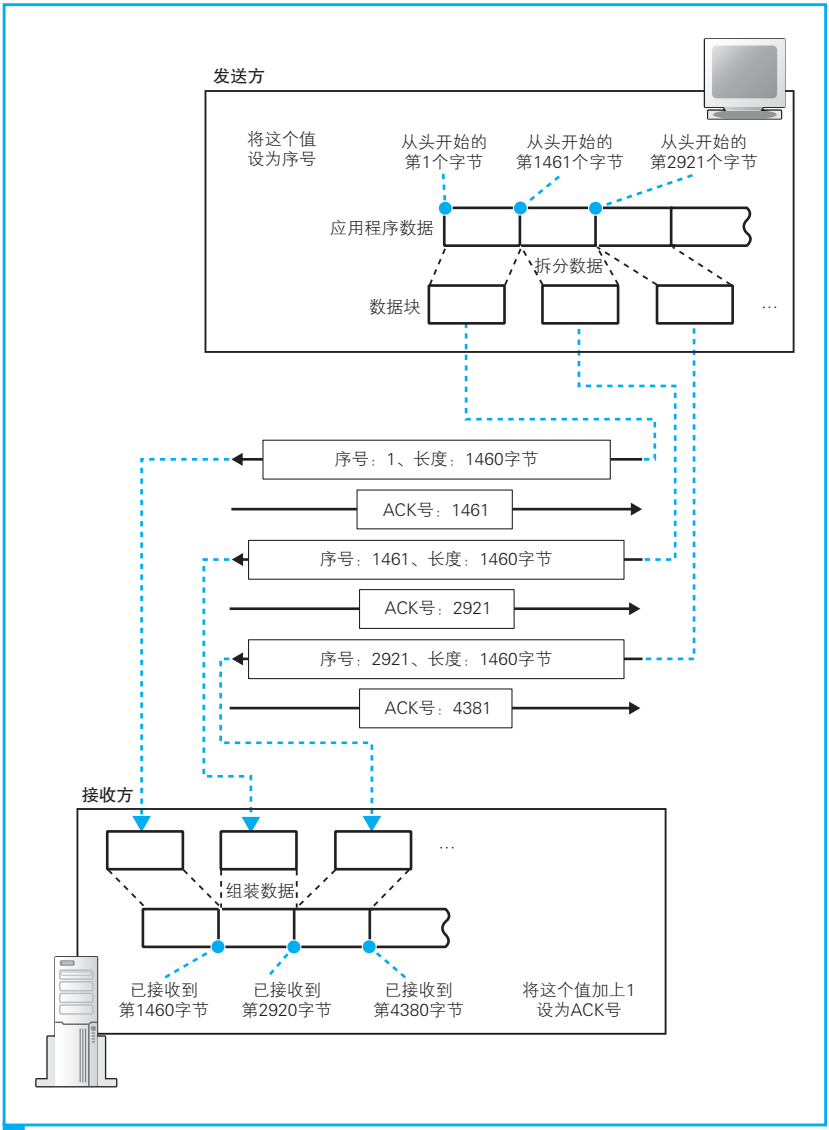

图 2.7　序号和 ACK 号的用法

发动攻击。但是如果初始值是随机的，那么对方就搞不清楚序号到底是从多少开始计算的，因此需要在开始收发数据之前将初始值告知通信对象。大家应该还记得在我们刚才讲过的连接过程中，有一个将 SYN 控制位设为 1 并发送给服务器的操作，就是在这一步将序号的初始值告知对方的。实际上，在将 SYN 设为 1 的同时，还需要同时设置序号字段的值，而这里的值就代表序号的初始值[①]。

前面介绍了通过序号和 ACK 号来进行数据确认的思路，但仅凭这些还不够，因为我们刚刚只考虑了单向的数据传输，但 TCP 数据收发是双向的，在客户端向服务器发送数据的同时，服务器也会向客户端发送数据，因此必须要想办法应对这样的情况。不过，这其实也不难，图 2.7 中展示的客户端向服务器发送数据的情形，我们只要增加一种左右相反的情形就可以了，如图 2.8 所示。首先客户端先计算出一个序号，然后将序号和数据一起发送给服务器，服务器收到之后会计算 ACK 号并返回给客户端；相反地，服务器也需要先计算出另一个序号，然后将序号和数据一起发送给客户端，客户端收到之后计算 ACK 号并返回给服务器。此外，如图所示，客户端和服务器双方都需要各自计算序号，因此双方需要在连接过程中互相告知自己计算的序号初始值。

明白原理之后我们来看一下实际的工作过程（图 2.9）。首先，客户端在连接时需要计算出与从客户端到服务器方向通信相关的序号初始值，并将这个值发送给服务器（图 2.9 ①）。接下来，服务器会通过这个初始值计算出 ACK 号并返回给客户端（图 2.9 ②）。初始值有可能在通信过程中丢失，因此当服务器收到初始值后需要返回 ACK 号作为确认。同时，服务器也需要计算出与从服务器到客户端方向通信相关的序号初始值，并将这

① 我们在前面讲连接操作的时候说过 SYN 为 1 表示进行连接，这是因为将 SYN 设为 1 并告知初始序号这一操作仅在连接过程中出现，因此发送 SYN 为 1 的网络包就表示发起连接的意思。实际上，SYN 是 Synchronize（同步）的缩写，意思是通过告知初始序号使通信双方保持步调一致，以便完成后续的数据收发检查，这才是 SYN 原本的含义。

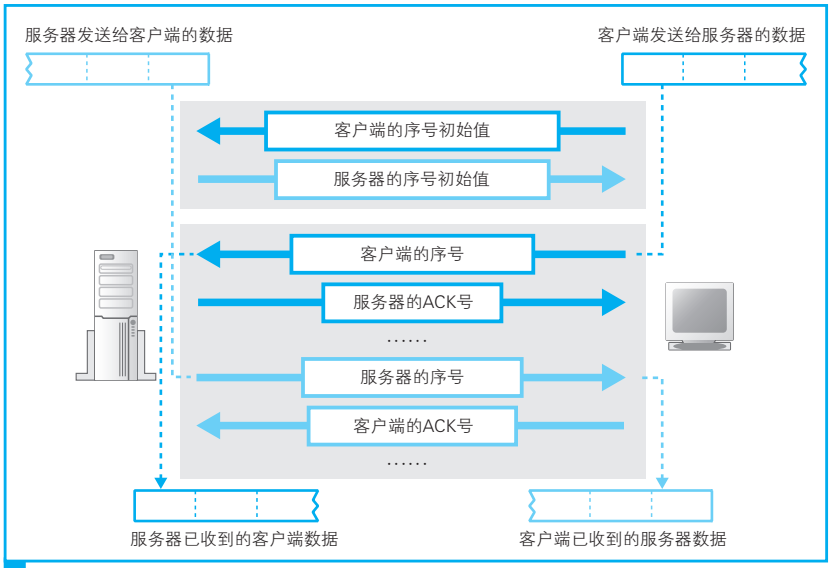

图 2.8　数据双向传输时的情况

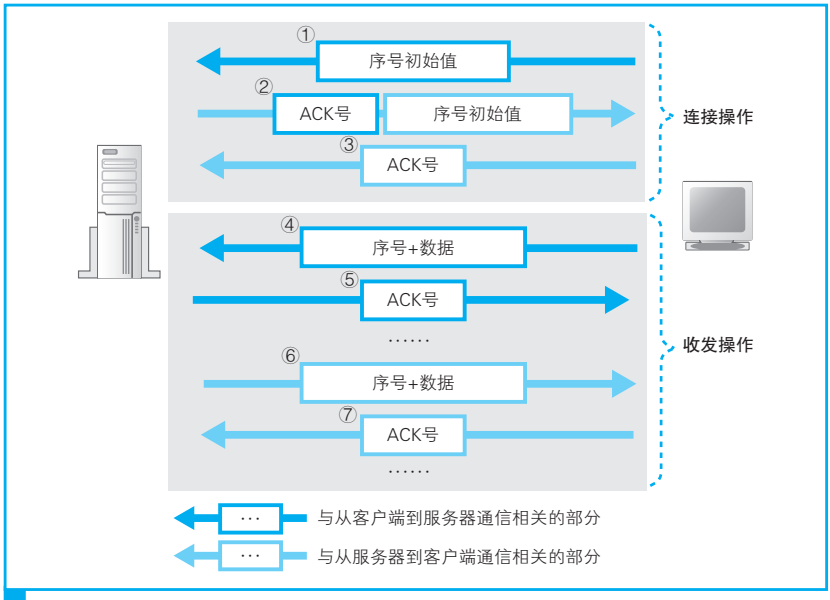

图 2.9　序号和 ACK 号的交互

个值发送给客户端（图 2.9 ②）。接下来像刚才一样，客户端也需要根据服务器发来的初始值计算出 ACK 号并返回给服务器（图 2.9 ③）。到这里，序号和 ACK 号都已经准备完成了，接下来就可以进入数据收发阶段了。数据收发操作本身是可以双向同时进行的，但 Web 中是先由客户端向服务器发送请求，序号也会跟随数据一起发送（图 2.9 ④）。然后，服务器收到数据后再返回 ACK 号（图 2.9 ⑤）。从服务器向客户端发送数据的过程则正好相反（图 2.9 ⑥⑦）。

TCP 采用这样的方式确认对方是否收到了数据，在得到对方确认之前，发送过的包都会保存在发送缓冲区中。如果对方没有返回某些包对应的 ACK 号，那么就重新发送这些包。

这一机制非常强大。通过这一机制，我们可以确认接收方有没有收到某个包，如果没有收到则重新发送，这样一来，无论网络中发生任何错误，我们都可以发现并采取补救措施（重传网络包）。反过来说，有了这一机制，我们就不需要在其他地方对错误进行补救了。

因此，网卡、集线器、路由器都没有错误补偿机制，一旦检测到错误就直接丢弃相应的包。应用程序也是一样，因为采用 TCP 传输，即便发生一些错误对方最终也能够收到正确的数据，所以应用程序只管自顾自地发送这些数据就好了。不过，如果发生网络中断、服务器宕机等问题，那么无论 TCP 怎样重传都不管用。这种情况下，无论如何尝试都是徒劳，因此 TCP 会在尝试几次重传无效之后强制结束通信，并向应用程序报错。

通过“序号”和“ACK 号”可以确认接收方是否收到了网络包。

2.3.4 根据网络包平均往返时间调整 ACK 号等待时间

前面说的只是一些基本原理，实际上网络的错误检测和补偿机制非常复杂。下面来说几个关键的点，首先是返回 ACK 号的等待时间（这个等待时间叫超时时间）。

当网络传输繁忙时就会发生拥塞，ACK 号的返回会变慢，这时我们就必须将等待时间设置得稍微长一点，否则可能会发生已经重传了包之后，前面的 ACK 号才姗姗来迟的情况。这样的重传是多余的，看上去只是多发一个包而已，但它造成的后果却没那么简单[①]。因为 ACK 号的返回变慢大多是由于网络拥塞引起的，因此如果此时再出现很多多余的重传，对于本来就很拥塞的网络来说无疑是雪上加霜。那么等待时间是不是越长越好呢？也不是。如果等待时间过长，那么包的重传就会出现很大的延迟，也会导致网络速度变慢。

看来等待时间需要设为一个合适的值，不能太长也不能太短，但这谈何容易。根据服务器物理距离的远近，ACK 号的返回时间也会产生很大的波动，而且我们还必须考虑到拥塞带来的影响。例如，在公司里的局域网环境下，几毫秒就可以返回 ACK 号，但在互联网环境中，当遇到拥塞时需要几百毫秒才能返回 ACK 号也并不稀奇。

正因为波动如此之大，所以将等待时间设置为一个固定值并不是一个好办法。因此，TCP 采用了动态调整等待时间的方法，这个等待时间是根据 ACK 号返回所需的时间来判断的。具体来说，TCP 会在发送数据的过程中持续测量 ACK 号的返回时间，如果 ACK 号返回变慢，则相应延长等待时间；相对地，如果 ACK 号马上就能返回，则相应缩短等待时间[②]。

2.3.5　使用窗口有效管理 ACK 号

如图 2.10（a）所示，每发送一个包就等待一个 ACK 号的方式是最简单也最容易理解的，但在等待 ACK 号的这段时间中，如果什么都不做那

① 如果某一个包被重复发送多次，接收方可以根据序号判断出这个包是重复的，因此并不会造成网络异常。

② 由于计算机的时间测量精度较低，ACK 返回时间过短时无法被正确测量，因此等待时间有一个最小值，这个值在每个操作系统上不一样，基本上是在 0.5 秒到 1 秒之间。

实在太浪费了。为了减少这样的浪费，TCP 采用图 2.10（b）这样的滑动窗口方式来管理数据发送和 ACK 号的操作。所谓滑动窗口，就是在发送一个包之后，不等待 ACK 号返回，而是直接发送后续的一系列包。这样一来，等待 ACK 号的这段时间就被有效利用起来了。

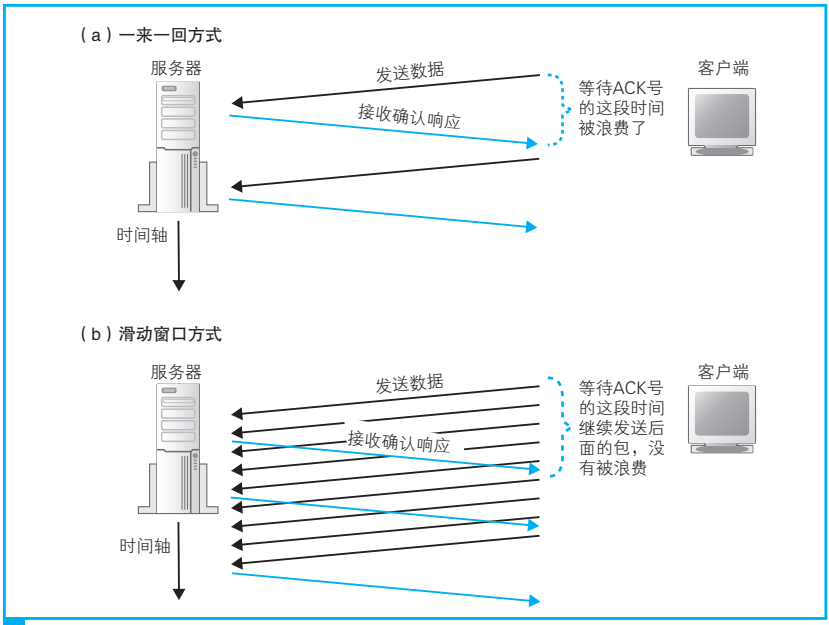

图 2.10　一来一回方式和滑动窗口方式

虽然这样做能够减少等待 ACK 号时的时间浪费，但有一些问题需要注意。在一来一回方式中，接收方完成接收操作后返回 ACK 号，然后发送方收到 ACK 号之后才继续发送下一个包，因此不会出现发送的包太多接收方处理不过来的情况。但如果不等返回 ACK 号就连续发送包，就有可能会出现发送包的频率超过接收方处理能力的情况。

下面来具体解释一下。当接收方的 TCP 收到包后，会先将数据存放到接收缓冲区中。然后，接收方需要计算 ACK 号，将数据块组装起来还原成原本的数据并传递给应用程序，如果这些操作还没完成下一个包就到了

也不用担心，因为下一个包也会被暂存在接收缓冲区中。如果数据到达的速率比处理这些数据并传递给应用程序的速率还要快，那么接收缓冲区中的数据就会越堆越多，最后就会溢出。缓冲区溢出之后，后面的数据就进不来了，因此接收方就收不到后面的包了，这就和中途出错的结果是一样的，也就意味着超出了接收方处理能力。我们可以通过下面的方法来避免这种情况的发生。首先，接收方需要告诉发送方自己最多能接收多少数据，然后发送方根据这个值对数据发送操作进行控制，这就是滑动窗口方式的基本思路。

关于滑动窗口的具体工作方式，还是看图更容易理解（图 2.11）。在这张图中，接收方将数据暂存到接收缓冲区中并执行接收操作。当接收操作完成后，接收缓冲区中的空间会被释放出来，也就可以接收更多的数据了，这时接收方会通过 TCP 头部中的窗口字段将自己能接收的数据量告知发送方。这样一来，发送方就不会发送过多的数据，导致超出接收方的处理能力了。

此外，单从图上看，大家可能会以为接收方在等待接收缓冲区被填满之前似乎什么都没做，实际上并不是这样。这张图是为了讲解方便，故意体现一种接收方来不及处理收到的包，导致缓冲区被填满的情况。实际上，接收方在收到数据之后马上就会开始进行处理，如果接收方的性能高，处理速度比包的到达速率还快，缓冲区马上就会被清空，并通过窗口字段告知发送方。

还有，图 2.11 中只显示了从右往左发送数据的操作，实际上和序号、ACK 号一样，发送操作也是双向进行的。

前面提到的能够接收的最大数据量称为窗口大小[①]，它是 TCP 调优参数中非常有名的一个。

① 一般和接收方的缓冲区大小一致。

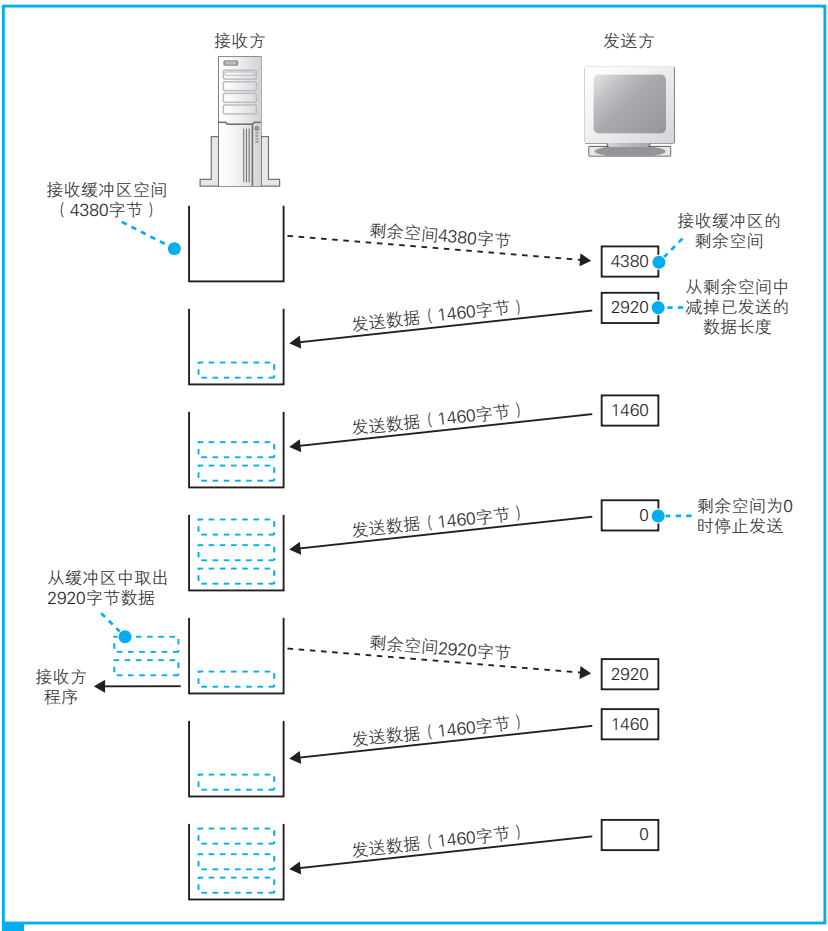

图 2.11 滑动窗口与接收缓冲区

2.3.6 ACK 与窗口的合并

要提高收发数据的效率，还需要考虑另一个问题，那就是返回 ACK 号和更新窗口的时机。如果假定这两个参数是相互独立的，分别用两个单独的包来发送，结果会如何呢？

首先，什么时候需要更新窗口大小呢？当收到的数据刚刚开始填入缓

冲区时，其实没必要每次都向发送方更新窗口大小，因为只要发送方在每次发送数据时减掉已发送的数据长度就可以自行计算出当前窗口的剩余长度。因此，更新窗口大小的时机应该是接收方从缓冲区中取出数据传递给应用程序的时候。这个操作是接收方应用程序发出请求时才会进行的，而发送方不知道什么时候会进行这样的操作，因此当接收方将数据传递给应用程序，导致接收缓冲区剩余容量增加时，就需要告知发送方，这就是更新窗口大小的时机。

那么 ACK 号又是什么情况呢？当接收方收到数据时，如果确认内容没有问题，就应该向发送方返回 ACK 号，因此我们可以认为收到数据之后马上就应该进行这一操作。

如果将前面两个因素结合起来看，首先，发送方的数据到达接收方，在接收操作完成之后就需要向发送方返回 ACK 号，而再经过一段时间[①]，当数据传递给应用程序之后才需要更新窗口大小。但如果根据这样的设计来实现，每收到一个包，就需要向发送方分别发送 ACK 号和窗口更新这两个单独的包[②]。这样一来，接收方发给发送方的包就太多了，导致网络效率下降。

因此，接收方在发送 ACK 号和窗口更新时，并不会马上把包发送出去，而是会等待一段时间，在这个过程中很有可能会出现其他的通知操作，这样就可以把两种通知合并在一个包里面发送了。举个例子，在等待发送 ACK 号的时候正好需要更新窗口，这时就可以把 ACK 号和窗口更新放在一个包里发送，从而减少包的数量。当需要连续发送多个 ACK 号时，也可以减少包的数量，这是因为 ACK 号表示的是已收到的数据量，也就是说，它是告诉发送方目前已接收的数据的最后位置在哪里，因此当需要连续发送 ACK 号时，只要发送最后一个 ACK 号就可以了，中间的可以全部省略。当需要连续发送多个窗口更新时也可以减少包的数量，因为连续发

① 计算机的操作非常快，因此并不需要很长时间，这个时间一般是微秒尺度的。

② 如果应用程序请求接收数据的频率比较低，有可能会在接收多个包之后才发送一个窗口通知包。

生窗口更新说明应用程序连续请求了数据，接收缓冲区的剩余空间连续增加。这种情况和 ACK 号一样，可以省略中间过程，只要发送最终的结果就可以了。

2.3.7 接收 HTTP 响应消息

到这里，我们已经讲解完协议栈接到浏览器的委托后发送 HTTP 请求消息的一系列操作过程了。

不过，浏览器的工作并非到此为止。发送 HTTP 请求消息后，接下来还需要等待 Web 服务器返回响应消息。对于响应消息，浏览器需要进行接收操作，这一操作也需要协议栈的参与。按照探索之旅的思路，本来是应该按照访问 Web 服务器的顺序逐一讲解其中的每一步操作，也就是说接收 HTTP 响应消息应该放在最后再讲，但这样一来大家可能容易忘记前面的部分，所以我们就把这部分内容放在这里讲一讲。

首先，浏览器在委托协议栈发送请求消息之后，会调用 read 程序（之前的图 2.3 ④）来获取响应消息。然后，控制流程会通过 read 转移到协议栈[①]，然后协议栈会执行接下来的操作。和发送数据一样，接收数据也需要将数据暂存到接收缓冲区中，这里的操作过程如下。首先，协议栈尝试从接收缓冲区中取出数据并传递给应用程序，但这个时候请求消息刚刚发送出去，响应消息可能还没返回。响应消息的返回还需要等待一段时间，因此这时接收缓冲区中并没有数据，那么接收数据的操作也就无法继续。这时，协议栈会将应用程序的委托，也就是从接收缓冲区中取出数据并传递给应用程序的工作暂时挂起[②]，等服务器返回的响应消息到达之后再继续执行接收操作。

① 随着控制流程转移，应用程序也会进入暂停状态。

② 大家可以认为这时协议栈会进入暂停状态，但实际上并非如此。协议栈会负责处理来自很多应用程序的工作，因此挂起其中一项工作并不意味着协议栈就完全暂停了，协议栈会继续执行其他的工作。在执行其他工作的时候，挂起的工作并没有在执行，因此看上去和暂停是一样的。

协议栈接收数据的具体操作过程已经在发送数据的部分讲解过了，因此这里我们就简单总结一下[①]。首先，协议栈会检查收到的数据块和 TCP 头部的内容，判断是否有数据丢失，如果没有问题则返回 ACK 号。然后，协议栈将数据块暂存到接收缓冲区中，并将数据块按顺序连接起来还原出原始的数据，最后将数据交给应用程序。具体来说，协议栈会将接收到的数据复制到应用程序指定的内存地址中，然后将控制流程交回应用程序。将数据交给应用程序之后，协议栈还需要找到合适的时机向发送方发送窗口更新[②]。

2.4 从服务器断开并删除套接字

2.4.1 数据发送完毕后断开连接

既然我们已经讲解到了这里，那么索性把数据收发完成后协议栈要执行的操作也讲一讲吧。这样一来，从创建套接字到连接、收发数据、断开连接、删除套接字这一系列关于收发数据的操作就全部讲完了。

毫无疑问，收发数据结束的时间点应该是应用程序判断所有数据都已经发送完毕的时候。这时，数据发送完毕的一方会发起断开过程，但不同的应用程序会选择不同的断开时机。以 Web 为例，浏览器向 Web 服务器发送请求消息，Web 服务器再返回响应消息，这时收发数据的过程就全部结束了，服务器一方会发起断开过程[③]。当然，可能也有一些程序是客户端发送完数据就结束了，不用等服务器响应，这时客户端会先发起断开过程。这一判断是应用程序作出的，协议栈在设计上允许任何一方先发起断开过程。

① 第 6 章我们将对从接收网络包到向应用程序传递数据的整个过程进行整理，大家可以参考该部分内容。

② 如果窗口更新能够和 ACK 号等合并的话，在这里就会发送合并后的包。

③ 这里讲的是 HTTP1.0 的情形，在 HTTP1.1 中，服务器返回响应消息之后，客户端还可以继续发起下一个请求消息，如果接下来没有请求要发送了，客户端一方会发起断开过程。

无论哪种情况，完成数据发送的一方会发起断开过程，这里我们以服务器一方发起断开过程为例来进行讲解。首先，服务器一方的应用程序会调用 Socket 库的 close 程序。然后，服务器的协议栈会生成包含断开信息的 TCP 头部，具体来说就是将控制位中的 FIN 比特设为 1。接下来，协议栈会委托 IP 模块向客户端发送数据（图 2.12 ①）。同时，服务器的套接字中也会记录下断开操作的相关信息。

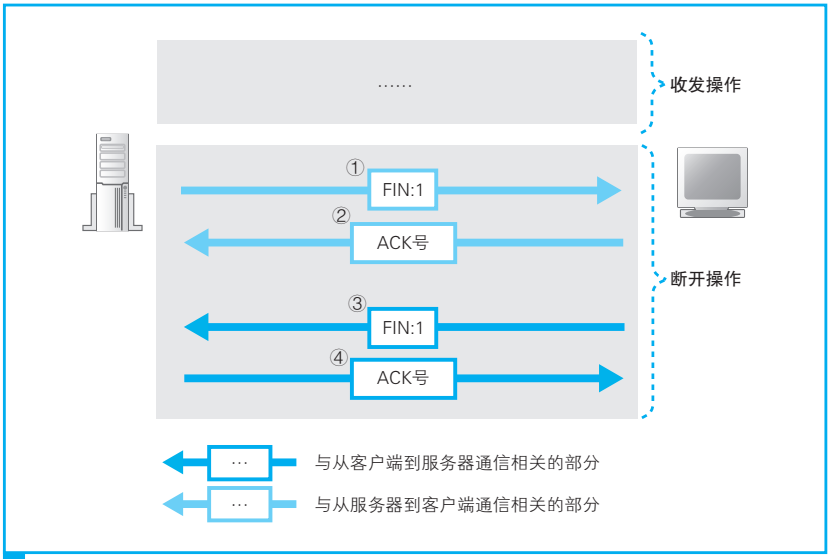

图 2.12　断开连接的交互过程

接下来轮到客户端了。当收到服务器发来的 FIN 为 1 的 TCP 头部时，客户端的协议栈会将自己的套接字标记为进入断开操作状态。然后，为了告知服务器已收到 FIN 为 1 的包，客户端会向服务器返回一个 ACK 号（图 2.12 ②）。这些操作完成后，协议栈就可以等待应用程序来取数据了。

过了一会儿，应用程序就会调用 read 来读取数据[①]。这时，协议栈不会

① 应用程序有可能在收到 FIN 为 1 的包之前就来读取数据，这时读取数据的操作会被挂起，等到 FIN 包到达再继续执行。

向应用程序传递数据[①]，而是会告知应用程序（浏览器）来自服务器的数据已经全部收到了。根据规则，服务器返回请求之后，Web 通信操作就全部结束了，因此只要收到服务器返回的所有数据，客户端的操作也就随之结束了。因此，客户端应用程序会调用 close 来结束数据收发操作，这时客户端的协议栈也会和服务器一样，生成一个 FIN 比特为 1 的 TCP 包，然后委托 IP 模块发送给服务器（图 2.12 ③）。一段时间之后，服务器就会返回 ACK 号（图 2.12 ④）。到这里，客户端和服务器的通信就全部结束了。

2.4.2 删除套接字

和服务器的通信结束之后，用来通信的套接字也就不会再使用了，这时我们就可以删除这个套接字了。不过，套接字并不会立即被删除，而是会等待一段时间之后再被删除。

等待这段时间是为了防止误操作，引发误操作的原因有很多，这里无法全部列举，下面来举一个最容易理解的例子。假设和图 2.12 的过程相反，客户端先发起断开，则断开的操作顺序如下。

（1）客户端发送 FIN
（2）服务器返回 ACK 号
（3）服务器发送 FIN
（4）客户端返回 ACK 号

如果最后客户端返回的 ACK 号丢失了，结果会如何呢？这时，服务器没有接收到 ACK 号，可能会重发一次 FIN。如果这时客户端的套接字已经删除了，会发生什么事呢？套接字被删除，那么套接字中保存的控制信息也就跟着消失了，套接字对应的端口号就会被释放出来。这时，如果别的应用程序要创建套接字，新套接字碰巧又被分配了同一个端口号[②]，而服务器重发的 FIN 正好到达，会怎么样呢？本来这个 FIN 是要发给刚刚删除

① 如果接收缓冲区中还有剩余的已接收数据，则这些数据会被传递给应用程序。
② 客户端的端口号是从空闲的端口号中随意选择的。

的那个套接字的，但新套接字具有相同的端口号，于是这个 FIN 就会错误地跑到新套接字里面，新套接字就开始执行断开操作了。之所以不马上删除套接字，就是为了防止这样的误操作。

至于具体等待多长时间，这和包重传的操作方式有关。网络包丢失之后会进行重传，这个操作通常要持续几分钟。如果重传了几分钟之后依然无效，则停止重传。在这段时间内，网络中可能存在重传的包，也就有可能发生前面讲到的这种误操作，因此需要等待到重传完全结束。协议中对于这个等待时间没有明确的规定，一般来说会等待几分钟之后再删除套接字。

2.4.3 数据收发操作小结

到这里，用 TCP 协议收发应用程序数据的操作就全部结束了。这部分内容的讲解比较长，所以最后我们再整理一下。

数据收发操作的第一步是创建套接字。一般来说，服务器一方的应用程序在启动时就会创建好套接字并进入等待连接的状态。客户端则一般是在用户触发特定动作，需要访问服务器的时候创建套接字。在这个阶段，还没有开始传输网络包。

创建套接字之后，客户端会向服务器发起连接操作。首先，客户端会生成一个 SYN 为 1 的 TCP 包并发送给服务器（图 2.13 ①）。这个 TCP 包的头部还包含了客户端向服务器发送数据时使用的初始序号，以及服务器向客户端发送数据时需要用到的窗口大小[①]。当这个包到达服务器之后，服务器会返回一个 SYN 为 1 的 TCP 包（图 2.13 ②）。和图 2.13 ①一样，这个包的头部中也包含了序号和窗口大小，此外还包含表示确认已收到包①的 ACK 号[②]。当这个包到达客户端时，客户端会向服务器返回一个包含表示确

① 如图 2.11 所示，窗口大小是由接收方告知发送方的，因此，在最初的这个包中，客户端告诉服务器的窗口大小是服务器向客户端发送数据时使用的。窗口大小的更新和序号以及 ACK 号一样，都是双向进行的。图 2.13 显示了窗口的双向交互。

② 设置 ACK 号时需要将 ACK 控制位设为 1。