> 考虑到传输层是计网的重中之重也是企业笔试面试的重要内容，做了一个知识小结。

传输层

1. 传输层解决什么问题

- 传输层位于网络层之上，为运行在不同主机上的**进程**之间提供了逻辑通信。（参见课本两个家庭的举例
- 关键词：端到端，TCP，UDP

2. 多路复用与多路分解

- 多路分解： 将传输层报文段中的数据交付到正确的套接字的工作叫做多路分解。
- 多路复用：在源主机从不同套接字收集数据块，并为每个数据块封装首部信息生成报文段，然后将报文段传递到网络层，这些工作称为多路复用。
- 借助运输层报文段中的**源端口号字段**、**目的端口号字段**，其中目的端口号字段是为了定向目的进程具有的端口号，源端口号字段是为了用作“返回地址”的一部分。

3. UDP与TCP

3.1 UDP与TCP的特点

- UDP（User Datagram Protocol）是无连接的，因此两个进程通信前没有握手过程；是没有拥塞控制的；是面向报文的，因此应用程序传下来的报文既不合并也不拆分，只是添加在UDP首部；是支持一对一、一对多、多对一和多对多的。
- TCP（Transmission Control Protocol）包括面向连接服务和可靠数据传输服务，前者意味着应用层数据报文开始流动之前，TCP让客户端与服务器互相交换数据层控制信息，即进行三次握手，后者意味着通信进程能依靠TCP无差错、按适当顺序交付所有发送的数据；TCP提供流量控制、拥塞控制；每一条TCP连接只能是一对一的。

3.2 首部格式

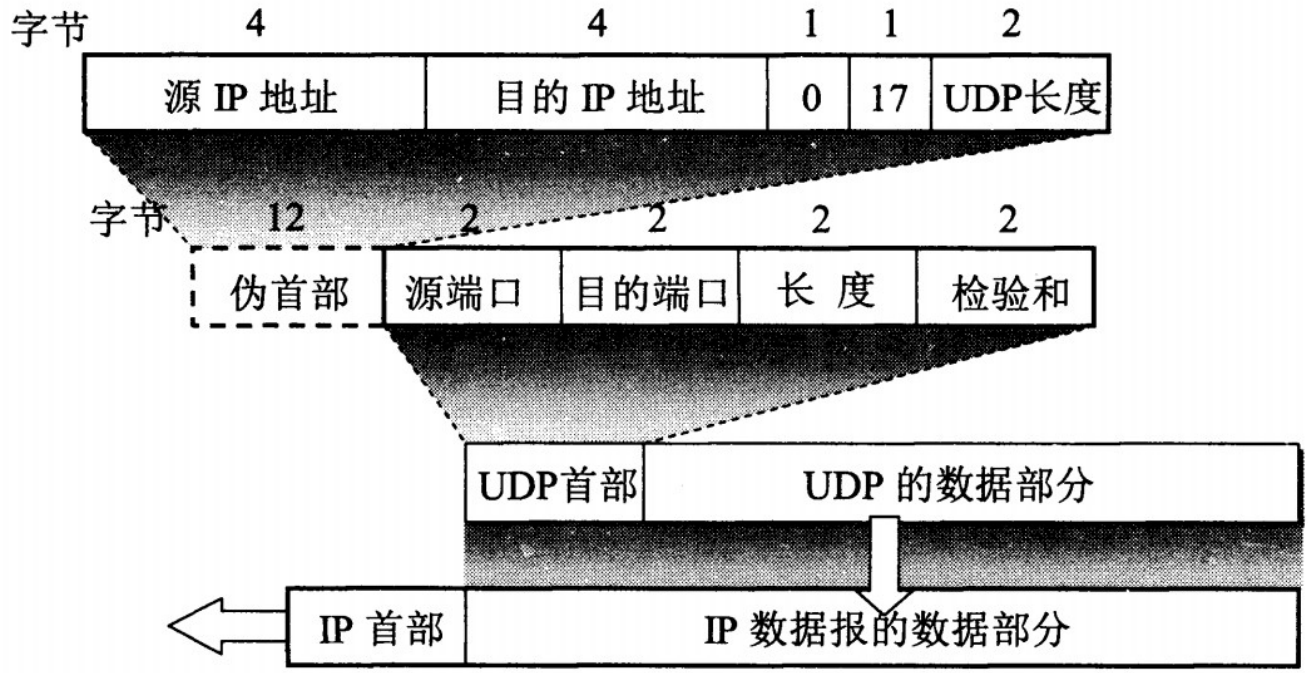

图 5-5　UDP 用户数据报的首部和伪首部

1. 如图所示，UDP的报文段首部开销仅有四个字段、每个字段由两个字节组成，总共8个字节

2. 端口号用于执行多路分解

3. 长度字段指示了UDP报文段首部加数据的总字节数

4. 检验和用于确定UDP报文段从源到目的地移动时其中的比特数是否发生了改变，有错则丢弃。

   4.1. UDP报文段中伪首部、首部、数据部分，被分为几个16比特字，相加，若加法溢出则回卷，最后得到的结果取反码运算，得到检验和(也是16比特字、即两个字节)。

   4.2. 在接收方，所有的16比特字（包括检验和）进行相加，如果没有差错，结果应该是1111111111111111.

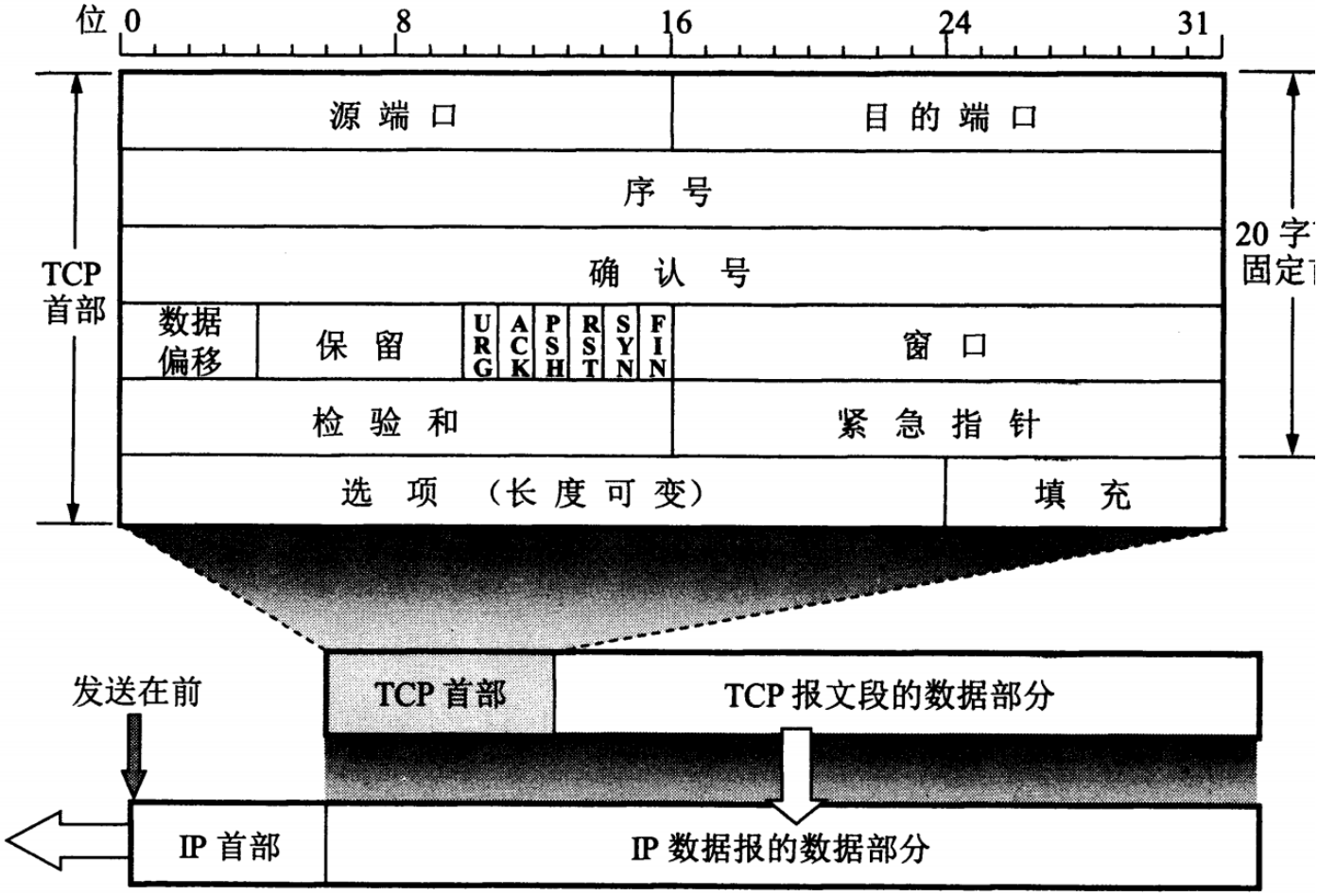

图 5-14　TCP 报文段的首部格式

1. 报文段首部长为**20个字节**

2. **序号**用于对字节流进行编号，若序号为 301，表示第一个字节的编号为 301，如果携带的数据长度为 100 字节，那么下一个报文段的序号应为 401。

3. **确认号**表示期望收到的下一个报文段的序号，例如 B 正确收到 A 发送来的一个报文段，序号为 501，携带的数据长度为 200 字节，因此 B 期望下一个报文段的序号为 701，B 发送给 A 的确认报文段中确认号就为 701。

4. 序号与确认号是TCP报文段中最重要的两个字段，是可靠传输服务的关键部分。

5. 数据偏移指的是数据部分距离报文开头部分的偏移量，也就是首部长度，考虑到**选项字段**通常为空，首部长度典型值为20字节。

6. 6 比特的**标志字段**，ACK用于指示确认字段中的值是有效的；RST、SYN、FIN用于连接建立与拆除。

7. 当 ACK=1 时确认号字段有效，否则无效。TCP 规定，在连接建立后所有传送的报文段都必须把 ACK 置 1。

8. SYN 在连接建立时用来同步序号。当 SYN=1，ACK=0 时表示这是一个连接请求报文段。若对方同意建立连接，则响应报文中 SYN=1，ACK=1。

9. 当 FIN=1 时，表示此报文段的发送方的数据已发送完毕，并要求释放连接。

10. **窗口**字段用于流量控制，指示接收方愿意接受的字节数量。

3.3 TCP与UDP对比

- UDP 对于关于发送什么数据以及何时发送的应用程控制更加精细。
- UDP 无需建立连接，没有这一段时延，UDP 无连接状态，不用维护相关的参数，可以支持更多的活跃用户。
- UDP 首部开支较小。
- 文件传输、电子邮件、Web 文档等等不能容忍丢失，需要TCP协议。

4. TCP

4.1 TCP 三次握手

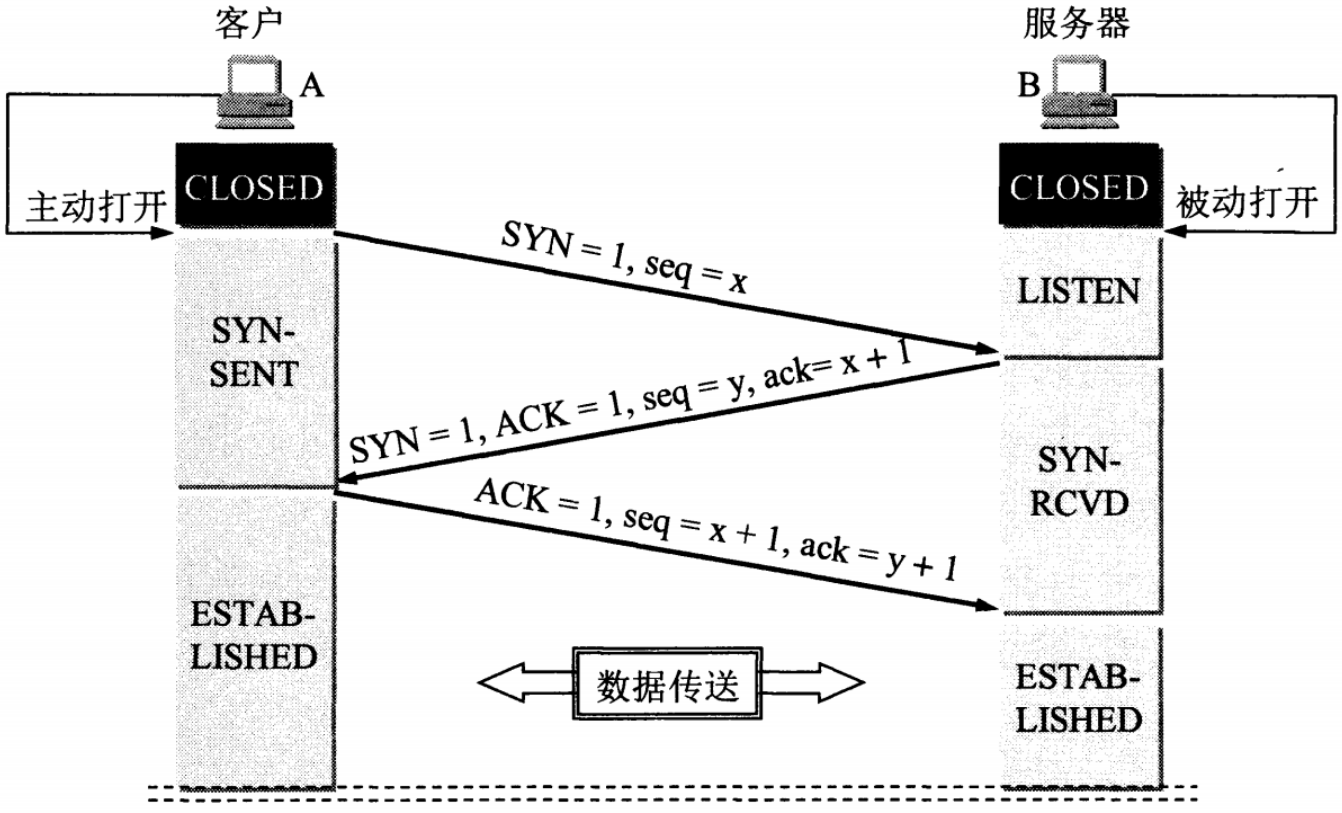

图 5-28 用三报文握手建立 TCP 连接

- 第一步： 客户端的TCP向服务器端发送**SYN 报文段**，该报文段不含应用层数据，但是其首部的SYN 标

志位为 1。 同时客户端会随机选择一个初始序号 *client_isn* ，并将其放在报文段首部的**序号**字段中。

- 第二步： 服务器接收到后，为该 TCP 连接分配 TCP 缓存和变量，并向该用户发送允许连接的报文段即 **SYNACK 报文段** 。该报文段依然不包含应用层数据，但是包含了3个重要信息： SYN 与 ACK 比特为 1 ； **确认号**字段被置为 *client_isn+1* ； 服务器选择自己的初始序号 *sever_isn* 并将其放置到 TCP 报文首段的**序号**字段。

- 第三步： 客户端收到 **SYNACK报文段**后需要为该连接分配缓存与变量，并向服务器发送另一个报文段进行确认， *sever_isn+1* 被放在**确认**字段， 而 *client_isn+1* 被放在**序号**字段。该阶段可以在报文段负载中携带客户端到服务器端的数据。

- 完成三次握手后，二者就可以互相发送包括数据的报文段了，之后的每个报文段 **SYN** 均为0。

- 三次握手的原因： 假设只有两次握手，当A想要建立连接时发送一个SYN，然后等待ACK，结果这个SYN因为网络问题没有及时到达B，所以A在一段时间内没收到ACK后，在发送一个SYN，B也成功收到，然后A也收到ACK，这时A发送的第一个SYN终于到了B，对于B来说这是一个新连接请求，然后B又为这个连接申请资源，返回ACK，然而这个SYN是个无效的请求，A收到这个SYN的ACK后也并不会理会它，而B却不知道，B会一直为这个连接维持着资源，造成资源的浪费。

4.2 TCP 四次挥手

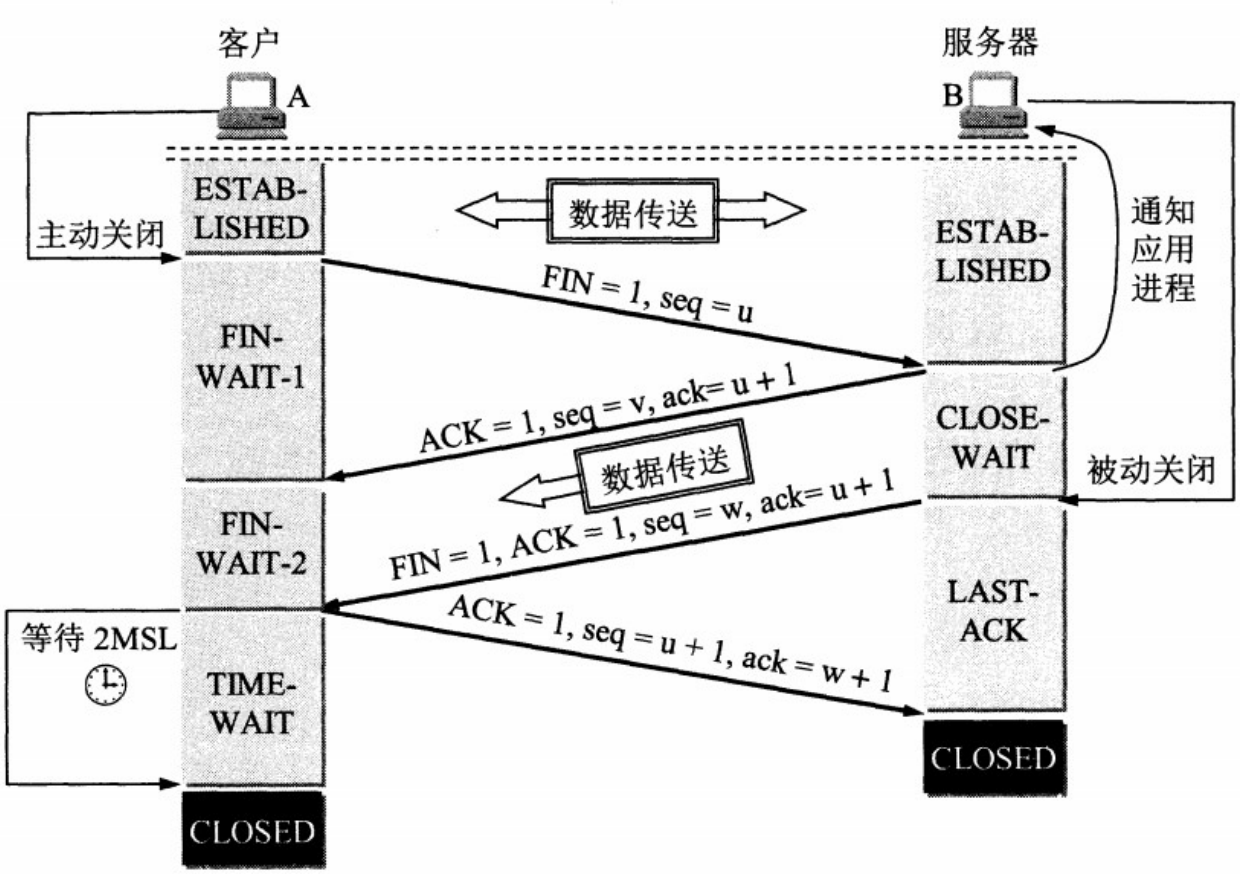

图 5-29　TCP 连接释放的过程

- 第一步： 客户端向服务器端发送一个特殊的TCP报文段，其中 FIN 比特被置为1.
- 第二步： 当服务器收到该报文段的时候，返回 ACK 报文段，其中ACK位被置为 1 ，同时确认位为收到的序号位+1. 此时服务器端的TCP状态为 **CLOSE_WAIT** 状态，不接收客户端的数据。
- 第三步： 当服务器端不再需要连接时，向客户端发送释放报文，其中 FIN 为1。
- 第四步： 当客户端收到时，向服务器端发送确认报文，ACK 为 1。随后进入**TIME_WAIT**状态，等待 **2 MSL**（Maximum Segment Lifetime）后进入**CLOSE**状态。 等待两个生命周期是为了避免服务器端没有接收到ACK报文而重发FIN报文，假设没有这两个生命周期，若服务器端再次发送FIN报文，客户端会返回 **RST报文**表示错误，甚至正处于下一轮中且端口号与上一轮相同，导致新旧数据包混淆，产生更棘手的问题。
- **CLOSE_WAIT** 状态的原因是为了使服务器端已经处理过的数据发送给客户端。

4.3 滑动窗口

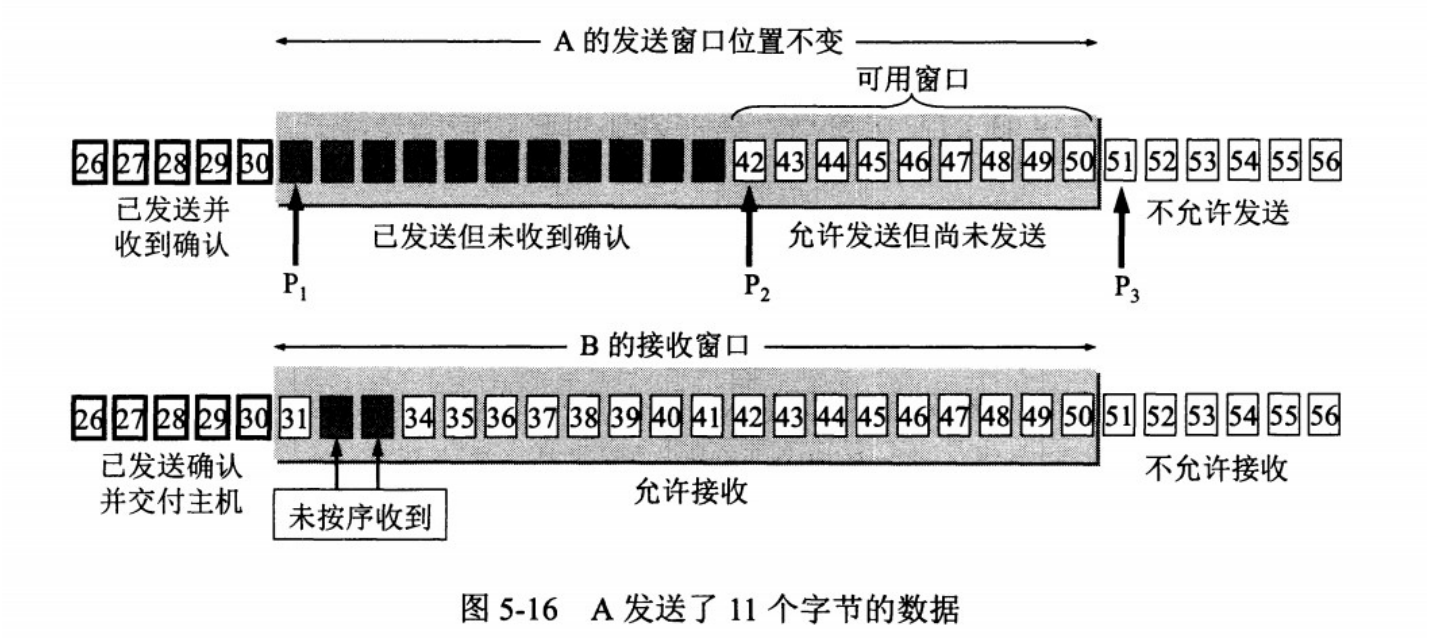

图 5-16　A 发送了 11 个字节的数据

1. “窗口”是一段可以被发送方发送的连续字节序列，其连续的范围被称为窗口
2. “滑动”是指这段连续的范围是随着发送的过程而变化的。
3. 发送窗口是发送缓存中的一部分，是可以被TCP协议发送的那部分，其实应用层需要发送的所有数据都被放进了发送者的发送缓冲区

发送方：

- *P1 --> base* 基序号（即最早未确认分组的序号）； *P2 -->nextseqnum* 最小未使用序号（即下一个待发分组的序号）
- [0, base-1] 已发送已被确认；[base, nextseqnum-1] 已发送但未被确认； [nextseqnum, base + N - 1] 表示那些要被立即发送的分组

接收方：

- 由于TCP是全双工的，故两端均同时作为发送方与接收方，均维护着一个独立的发送缓冲区和接收缓冲区，且二者是对等的
- 接收方的窗口大小根据发送方给出的窗口值得出
- 接收窗口会对窗口内最后一个按序到达的字节进行确认，从而得到该字节之前的所有字节已经确认接收。

4.4 流量控制

- 接收方传递信息给发送方，使其不要发送信息太快，是一种端到端的控制。
- TCP通过让发送方维护一个被称为接收窗口的变量来提供流量控制，接收窗口被用于给发送方提供接收方还有多少缓存空间可用的指示。（注意依然考虑全双工）
- 变量 *LastByteRead* 指的是接收方的应用进程从缓存中读出的数据流的最后一个字节的编号
- 变量 *LastByteRcvd* 指的是从网络中到达的并且已放入接收方接收缓存中的数据流的最后一个字节的编号
- TCP 不允许已分配的内存溢出，因此必须成立：

$$\text{LastByteRcvd} - \text{LastByteRead} <= \text{RcvBuffer}$$

- 接收窗口为 rwnd，根据**缓存可用空间的数量**来设置：

$$\text{rwnd} = \text{RcvBuffer} - [\ \text{LastByteRevd} - \text{LastByteRead}\ ]$$

- 主机B通过把当前的rwnd值放入它发给主机A的报文段接收窗口字段中，通知主机A它在该连接的缓存中还有多少可用空间。开始时设定 rwnd = RevBuffer

- 主机A需要轮流跟踪变量 LastByteSent 与 LastByteAcked ，这两个变量之差即为主机A发送到连接中但未被确认的数据量，通过将该值控制在rwnd以内，就可以保证主机A不会使主机B的接受缓存溢出。

- 假设主机B的接收缓存已满，使得 rwnd = 0， 此时主机A应该继续发送只有一个字节数据的报文段，这些报文段将被接收方确认。这是为了避免出现死锁，避免主机B已经有了新空间而主机A不知道。

4.5 拥塞控制

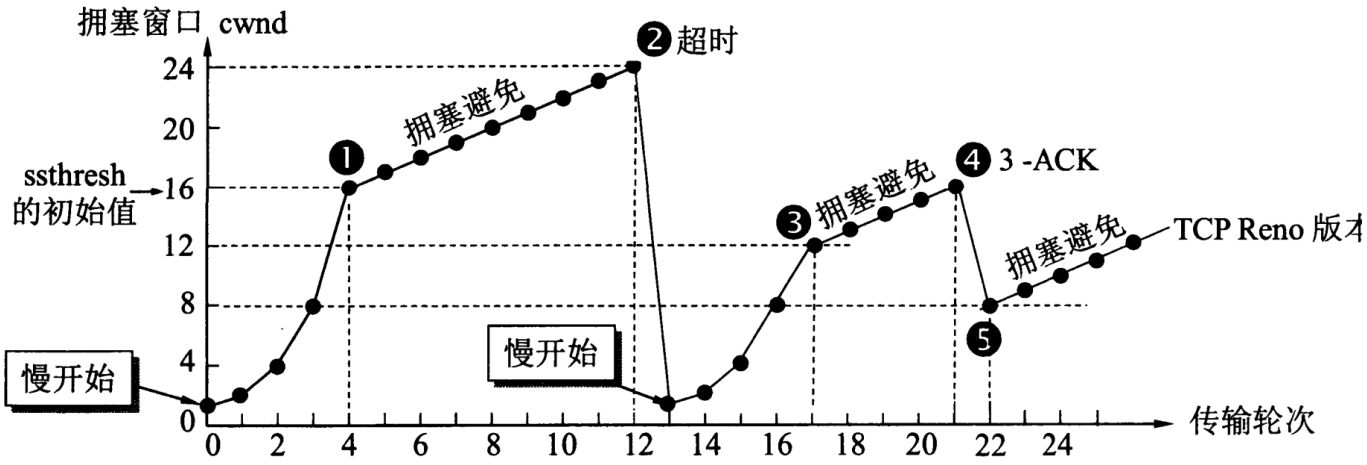

图 5-25　TCP 拥塞窗口 cwnd 在拥塞控制时的变化情况

- 如果网络出现拥塞，分组将会丢失，此时发送方会继续重传，从而导致网络拥塞程度更高。因此当出现拥塞时，应当控制发送方的速率。这一点和流量控制很像，但是出发点不同。流量控制是为了让接收方能来得及接收，而拥塞控制是为了降低整个网络的拥塞程度。
- 拥塞的表现形式：丢包（路由器缓存溢出）、高延迟（拥塞）
- 四个方法进行拥塞控制：慢开始、拥塞避免、快重传、快恢复
- 运行在发送方的TCP拥塞控制机制跟踪一个额外的变量cnwd（拥塞窗口），它对一个TCP发送方能够向网络中发送流量的速率进行了限制。发送方中未被确认的数据量不会超过cwnd和rwnd的较小值。（我们一般假设TCP接收缓存足够大）

$$LastByteSent - LastByteAcked <= min\{ cwnd, rwnd \}$$

4.5.1 慢开始与拥塞避免

- 发送的最初执行慢开始，cwnd 的值为一个 MSS 的较小值（通常为1），这使得初始发送速率大约为 MSS/RTT。 而此时对于 TCP 发送方来说，可用带宽可能比 MSS/RTT 大得多，并且其希望迅速找到可用带宽的数量，因此每次确认后都将 cwnd 翻倍。
- 为了约束这样一个过程，设置一个 *ssthresh*（慢开始阈值），当cwnd超过该阈值的时候，进入拥塞避免模式，cwnd 开始每次只增加 1。
- 当检测到拥塞时，令 **ssthresh = cwnd/2**，重新进入慢开始状态。

4.5.2 快重传与快恢复

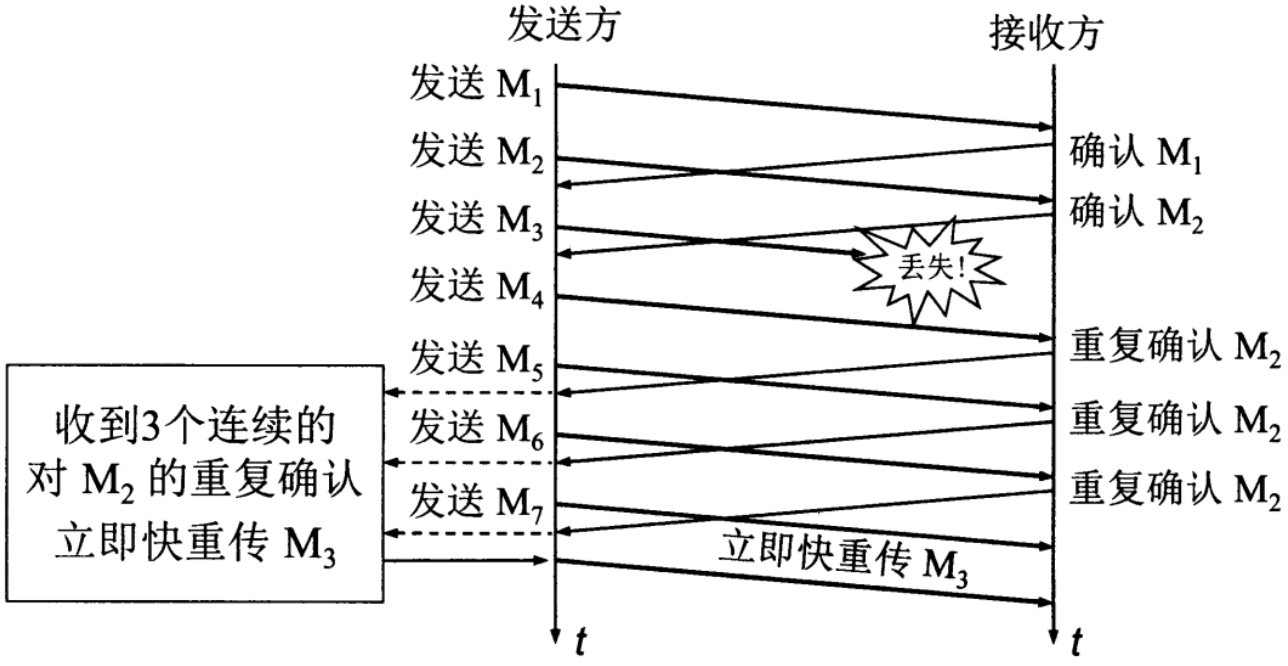

图 5-26　快重传的示意图

- 接收方每次接到一个有序报文都需要给出快速回复，如果接受到的无序报文即检测到丢包，则快速重传

3次重复确认。

- 发送方若检测到3次重复确认即意味着明白出现丢包，则快速重传该重复确认的下一个报文，同时将慢启动阈值 ssthresh 减半，并且将 cwnd 设置为减半后的ssthresh，同时**开始执行拥塞避免**。