TCP/IP模型——传输层

本篇文章，篇幅较长，全文大概18000多字，花费了近一周的时间才写完。写这篇文章的原因很简单：在面试中，总是会问到计算机网络方面的问题，而TCP是必不可少的。每一次的回答都不尽满意，甚至有些点自己还不清楚，所以自己决定将传输层协议这一块儿总结一下，相当于是一个再学习过程。如果你碰巧看到了这篇文章，又刚好对传输层协议感兴趣，那么不妨花费一点时间好好看一下，相信会对你有一定的帮助。

若是你在浏览的过程中，发现文章有问题，请留言，我会尽快更正。谢谢。

注：本篇文章是以谢希仁教授的《计算机网络》第七版为基础学习总结的。参考文献包括但不限于《计算机网络》、百度百科及CSDN等博文。

1. 传输层协议概述

1.1 进程间的通信

1.1.1 传输层的作用

从通信和信息处理的角度看，**传输层向它上面的应用层提供通信服务**，它属于面向通信部分的最高层，同时也是用户功能中的最低层。当网络的边缘部分中的两台主机使用网络的核心部分的功能进行端到端的通信时，只有主机的协议栈才有运输层，而网络核心部分中的路由器在转发分组时都只用到下三层的功能。

传输层的任务：负责向两台主机中进程之间的通信提供通用的数据传输服务。

1.1.2 为什么需要传输层?

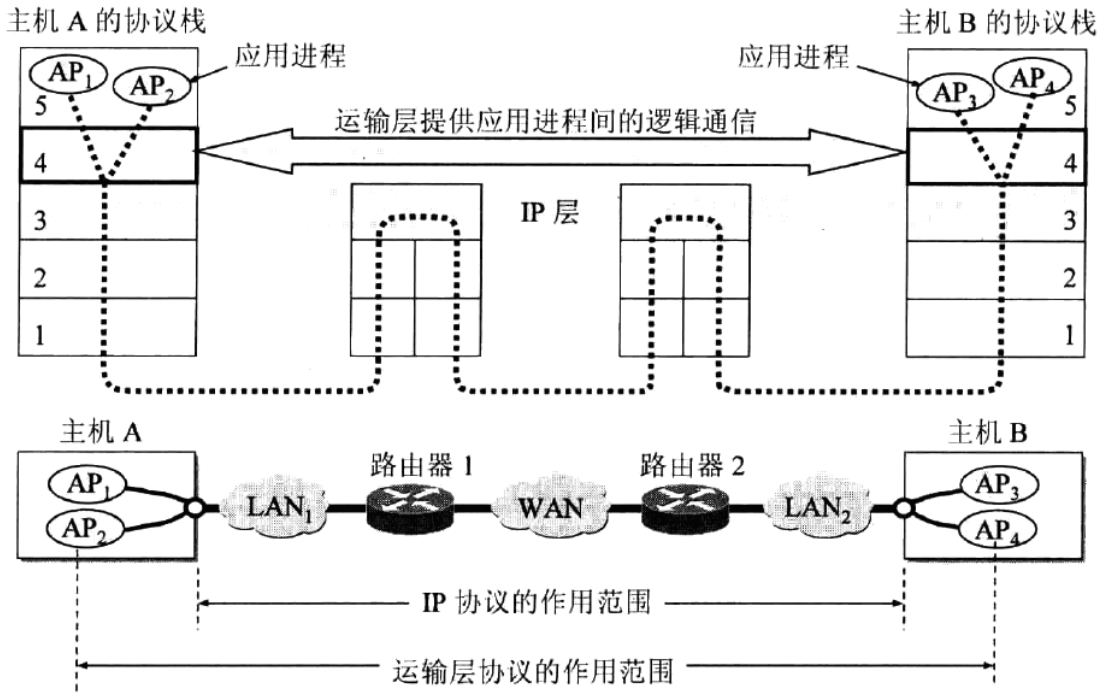
从IP层来说，通信的两端是两台主机。IP协议能够把源主机A发送出的分组，按照首部中的目的地址，送交到目的主机B。但是，**真正通信的实体不是主机，而是主机中的进程。**所以，两台主机进行通信就是两台主机中的**应用进程互相通信。**IP协议只能将分组发送到目的主机，但是这个分组还停留在主机的网络层而没有交付给主机中的应用进程。从传输层的角度看，通信的真正端点是主机中的进程，端到端的通信是应用进程之间的通信。

1.1.3 复用和分用

在一台主机中经常有多个应用进程同时分别和另一台主机中的多个应用进程进行通信。如图1，主机A的应用进程AP1和主机的AP3通信，而主机A中的AP2也和主机B中的AP4通信。这表明，传输层具有复用和分用的功能。

> 复用：发送方不同的应用进程都可以使用同一个传输层协议传送数据。
>
> 分用：接收方的运输层在去掉报文的首部后能够把这些数据正确交付目的应用进程。

图1：传输层为相互通信的应用进程提供了逻辑通信



1.2 传输层协议

1.2.1 协议划分

传输层主要使用以下两种协议：

- **用户数据报协议 UDP**（User Datagram Protocol）——提供无连接的、尽最大努力的数据传输服务（不保证数据传输的可靠性），其传输的单位是用户数据报。
- **传输控制协议 TCP**（Transmission Control Protocol）——提供面向连接的、可靠的数据传输服务，其传输的单位是报文段。

UDP与TCP的区别：

- UDP在传输数据之前不需要先建立连接。目的主机收到UDP报文后，不需要给出确认，UDP提供不可靠服务。有单播，广播，多播的功能。头部开销小，传输数据报文高效。
- TCP提供面向连接的服务。传输数据之前必须先建立连接，数据传输结束后要释放连接。TCP不提供广播或多波服务。因为TCP提供可靠的、面向连接的运输服务，因此需要连接管理、确认、流量控制等。提供全双工通信，允许通信的双方应用程序在任何时候同时发送数据。

图2：使用 UDP和TCP协议的各种应用和应用层协议

应　　用	应用层协议	运输层协议
名字转换	DNS（域名系统）	UDP
文件传送	TFTP（简单文件传送协议）	UDP
路由选择协议	RIP（路由信息协议）	UDP
IP 地址配置	DHCP（动态主机配置协议）	UDP
网络管理	SNMP（简单网络管理协议）	UDP
远程文件服务器	NFS（网络文件系统）	UDP
IP 电话	专用协议	UDP
流式多媒体通信	专用协议	UDP
多播	IGMP（网际组管理协议）	UDP
电子邮件	SMTP（简单邮件传送协议）	TCP
远程终端接入	TELNET（远程终端协议）	TCP
万维网	HTTP（超文本传送协议）	TCP
文件传送	FTP（文件传送协议）	TCP

1.2.2 协议端口号

协议端口号标识了一个主机上进行通信的不同的应用程序。

为什么需要端口号?

> 一台拥有IP地址的主机可以提供许多服务，比如Web服务、FTP服务、SMTP服务等，这些服务完全可以通过1个IP地址来实现。因为IP 地址与网络服务的关系是一对多的关系，不能只靠IP地址来区分不同的服务。实际上是通过“IP地址+端口号”来区分不同的服务的。

在TCP/IP协议，用源IP、源端口号、目的IP、目的端口号、协议号这样一个五元组来标识一个通信。

端口号划分：

> - **服务端使用的端口号：**

- **系统端口号：数值 0~1023。**IANA把这些端口号指派给了TCP/IP最重要的一些应用程序，让所有用户知道并使用。

应用程序	FTP	SSH	TELNET	SMTP	DNS	TFTP	HTTP	SNMP	HTTPS
端口号	21	22	23	25	53	69	80	161	443

- **登记端口号：数值为 1024~49151。**这类端口号是没有被系统端口号的应用程序使用。使用这类端口号需要在IANA按照规定的手续等级，以防止重复。
- **客户端使用的端口号：数值为 1024~49151。**这类端口号在客户端进程运行时动态选择，也叫短暂端口号。这类端口号留给客户端进行选择暂时使用。当服务器进程收到客户进程的报文时，就知道客户进程使用的端口号，因而可以把数据发送给客户进程。通信结束后，刚才已经使用过的客户端口号就不存在，这个端口号可以供给其他客户进程使用。

两个问题

1. **一个进程是否可以bind多个端口号?**

   > 可以，因为一个进程可以打开多个文件描述符，而每个文件描述符对应一个端口号，所以一个进程可以绑定多个端口号。

2. **一个端口号是否可以被多个进程bind?**

   > 不可以，如果进程先绑定一个端口号，然后在fork一个子进程，这样的话就可以实现多个进程绑定一个端口号，但是两个不同的进程绑定同一个端口是不可以的。

---

2. 用户数据报协议 UDP

2.1 UDP 概述

UDP提供无连接，尽最大努力交付的数据传输服务。

2.1.1 主要特点

- **UDP是无连接的，**即发送数据不需要建立连接（发送数据结束也无需释放连接），因此减少了开销和发送数据之前的时延。
- **UDP使用尽最大努力交付，**既不保证可靠交付，因此主机不需要维持复杂的连接状态表。
- **UDP是面向数据报文的。**发送方的UDP对应用程序叫下来的报文，再添加首部后就向下交付IP层。UDP对应用层交下来的报文，既不拆分，也不合并，而是保留这些报文的边界。
- **UDP没有拥塞控制，**因此网络出现拥塞不会使源主机的发送速率降低。
- **UDP支持一对一、多对一、一对多和多对多的交互通信。**
- **UDP首部开销小，**只有8个字节，比TCP的20个字节的首部要短。
- **UDP没有真正意义上的发送缓冲区，**调用sendto会直接交给内核，由内核将数据传给网络层协议进行后续的传输动作；**UDP具有接收缓冲区，**但这个接收缓冲区不能保证收到的UDP报的顺序和发送UDP报的顺序一致；如果缓冲区满了，再到达的UDP数据就会被丢弃。

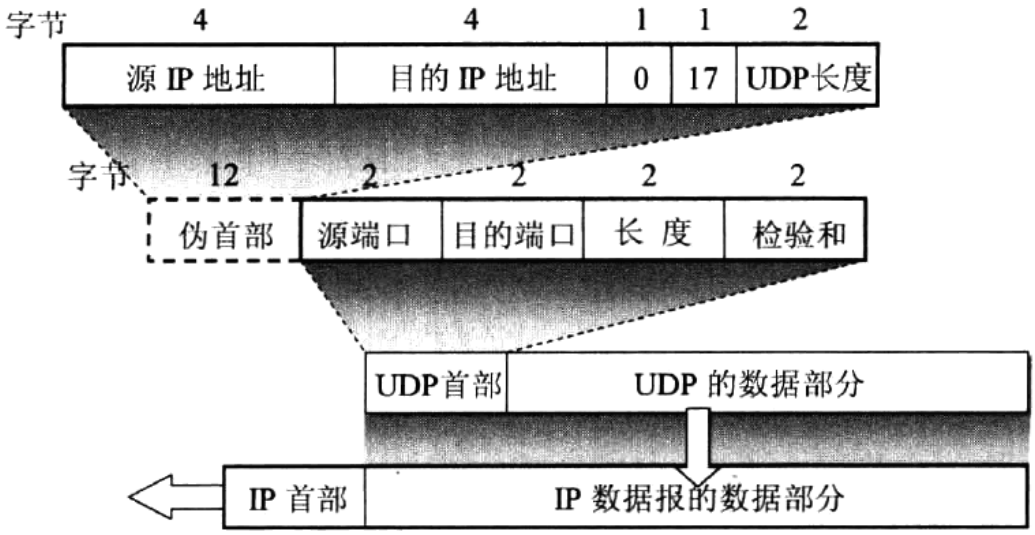
2.2 UDP首部格式

用户数据报UDP有两个字段：数据字段和首部字段。如图3，各字段含义如下：

- **源端口**　　源端口号。在需要对方回信时选用。不需要时可以全0。

- **目的端口**　目的端口号。终点交付报文时必须使用。
- **长度**　　UDP用户数据报的长度，其最小值为8（仅有首部）
- **检验和**　　检测UDP用户数据报在传输中是否有错。有错就丢弃。

图3：UDP用户数据报格式



当传输层从IP层收到UDP数据报时，就根据首部中的目的端口，把UDP数据报通过相应的端口，上交最后的终点——应用进程。如果接收方UDP发现收到的报文中的目的端口号不正确，就丢弃该报文，并由网际控制报文协议ICMP发送“端口不可达”差错报文给发送方。

---

3. 传输控制协议 TCP

3.1 TCP 概述

TCP提供一种面向连接的 、可靠的字节流服务。

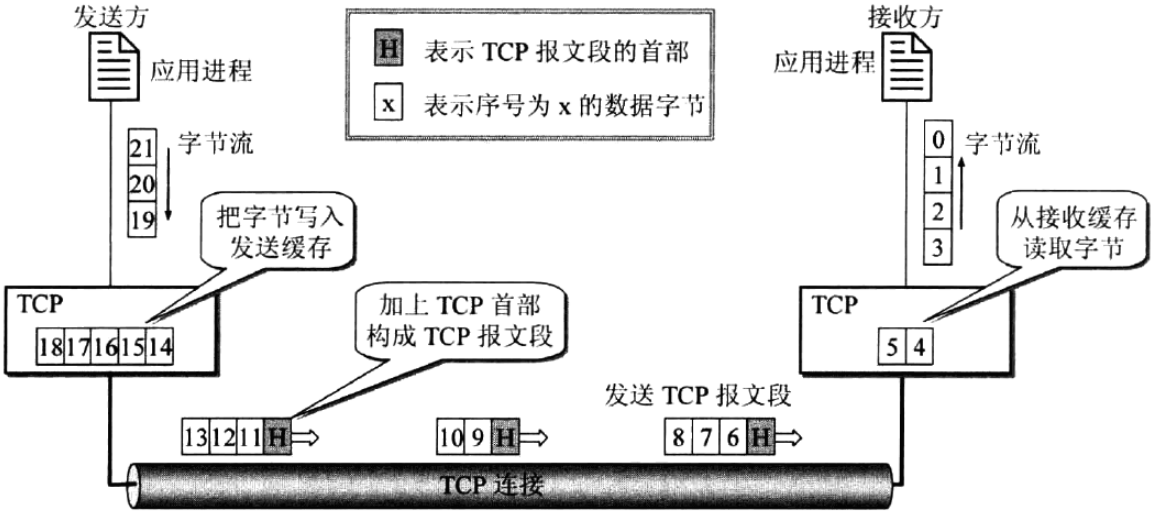
3.1.1 主要特点

- **TCP是面向连接的传输层协议。**应用程序在使用TCP协议之前，必须先建立TCP连接。在传送数据完毕后，必须释放已经建立的TCP连接。
- **每一条TCP连接只能有两个端点，每一条TCP连接只能是点对点的。**
- **TCP提供可靠交付的服务。**通过TCP连接传送的数据，无差错、不丢失、不重复，并且能按序到达。
- **TCP提供全双工通信。**TCP允许通信双方的应用进程在任何时候都能发送数据。**TCP连接的两端均设有发送缓存和接收缓存，**用来临时存放双向通信的数据。
    - **在发送时，**应用程序把数据传送给TCP的缓存后，就可以自己做自己的事，而TCP在合适的时候把数据发送出去。
    - **在接收时，**TCP把收到的数据放入缓存，上层应用程序在合适的时候读取缓存中的数据。
- **面向字节流。**TCP中的“流”指的是流入到进程或从进程流出的字节序列。虽然应用程序和TCP交互是一次一个数据块，但TCP把应用程序交下来的数据仅仅看成是一连串的无结构的字节流。
    - **TCP不保证接收方的应用数据块和发送方所发出的数据块具有对应的大小关系（**例如，发送方的应用程序给发送方的 TCP 10个数据块，接TCP的接收方可能只用了4

个数据块就把收到的字节流交付上层的应用程序）。

- **TCP不关心应用进程一次把多长的报文发送到TCP缓存中，而是根据对方给出的窗口值和当前网络的拥塞程度来决定一个报文段应包含多少个字节。**如果应用程序发送的数据块太长，TCP 会将其划分短一些在传送。如果应用程序一次只发一个字节，TCP收端也可以累积到足够多的字节发送给收端的应用程序。

图4：TCP面向字节流

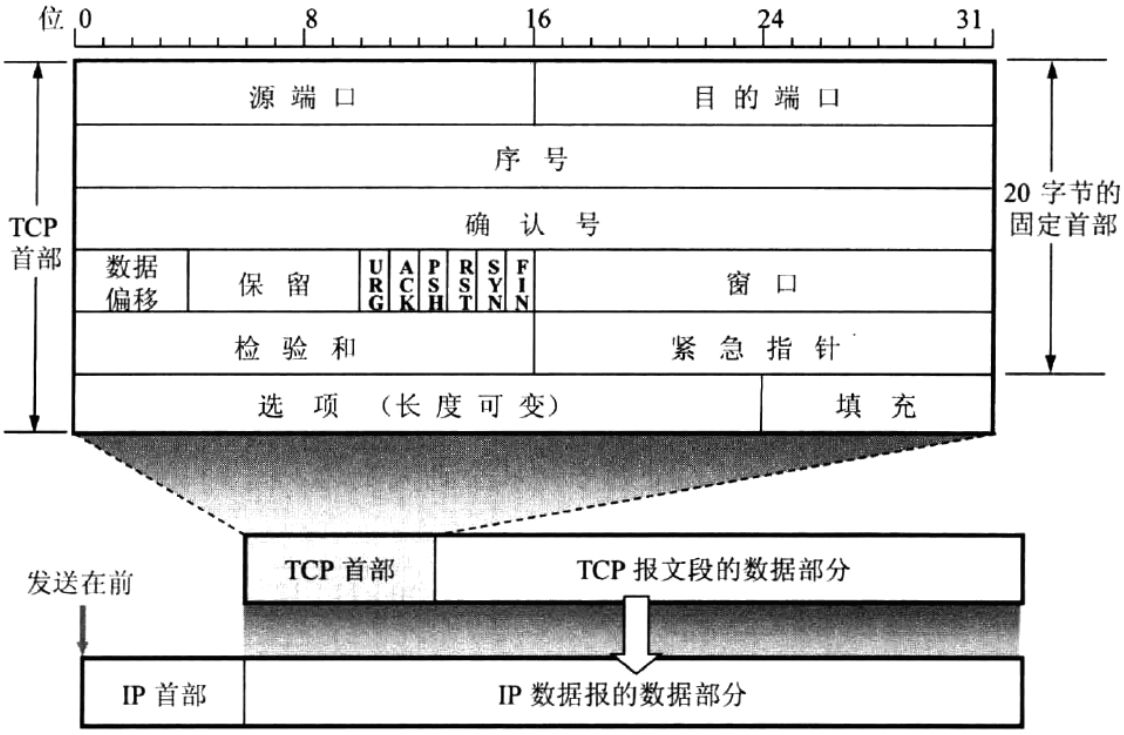


3.2 TCP 首部格式

TCP虽然是面向字节流的，但TCP传送的数据单元却是报文段。一个TCP报文段分为首部和数据两部分，而TCP的全部功能在首部的各个字段体现。

TCP报文段首部前20个字节是固定的，后面有4n个字节是根据需要而增加的选项。因此TCP首部的最小长度是20个字节。

图5：TCP 报文段的首部格式



各字段意义如下：

1. **源端口和目的端口：**各占2 个字节， 分别写入源端口号和目的端口号。

2. **序号：**占4 字节。若序号达到最大，则下一个序号右回到0。在一个TCP连接中传送的字节流中的每一个字节都按顺序编号。首部中的序号字段值指的是本报文段所发送的数据的第一个字节的序号。

   > 例如， 一报文段的序号字段值是301, 而携带的数据共有100字节。这就表明： 本报文段的数据的第一个字节的序号是301, 最后一个字节的序号是400。显然， 下一个报文段（如果还有的话） 的数据序号应当从401 开始， 即下一个报文段的序号字段值应为401。这个字段的名称也叫做“ 报文段序号”。

3. **确认号：**占4个字节，是期望收到对方下一个报文段的第一个数据字节的序号。若确认号 = N, 则表明：到序号N-1为止的所有数据都已经正确收到。

   > 例如， B 正确收到了A 发送过来的一个报文段，其序号字段值是501，而数据长度是200 字节（序号501 - 700)，这表明B 正确收到了A 发送的到序号700 为止的数据。因此，B 期望收到A 的下一个数据序号701，于是B 在发送给A 的确认报文段中把确认号置为701。请注意，现在的确认号不是501，也不是700，而是701。

4. **数据偏移：**占4位，占4 位，它指出TCP 报文段的数据起始处距离TCP报文段的起始处有多远。这个字段实际上是指出TCP 报文段的首部长度。TCP 首部的最大长度为60个字节。
5. **保留：**占6 位，保留为今后使用，但目前应置为0。
6. **标志位**，各占1位。
   - **紧急URG：**当URG=1时，表明紧急指针字段有效。它告诉系统此报文段中有紧急数据， 应尽快传送（相当于高优先级的数据），而不要按原来的排队顺序来传送。当URG 置1时，发送应用进程就告诉发送方的TCP 有紧急数据要传送。于是发送方TCP 就把紧急数据插入到本报文段数据的最前面， 而在紧急数据后面的数据仍是普通数据。这时要与首部中紧急指针(Urgent Pointer)字段配合使用。
   - **确认ACK：**仅当ACK= 1时确认号字段才有效。当ACK=0时，确认号无效。TCP 规定，在连接建立后所有传送的报文段都必须把ACK 置1。
   - **推送PSH：**当两个应用进程进行交互式的通信时，有时在一端的应用进程希望在键入一个命令后立即就能够收到对方的响应。在这种情况下，TCP就可以使用推送(push)操作。这时，发送方TCP 把PSH置1, 并立即创建一个报文段发送出去。接收方TCP收到PSH= 1的报文段，就尽快地（即“推送“ 向前）交付接收应用进程，而不再等到整个缓存都填满了后再向上交付。
   - **复位RST：**当RST= 1时，表明TCP连接中出现严重差错（如由于主机崩溃或其他原因），必须释放连接，然后再重新建立运输连接。RST置1还用来拒绝一个非法的报文段或拒绝打开一个连接。RST 也可称为重建位或重置位。
   - **同步SYN：**在连接建立时用来同步序号。当SYN=1而ACK=0时，表明这是一个连接请求报文段。对方若同意建立连接， 则应在响应的报文段中使SYN=1 和ACK=1。因此，SYN置为1就表示这是一个连接请求或连接接受报文。
   - **终止FIN：**用来释放一个连接。当FIN= 1时，表明此报文段的发送方的数据已发送完毕， 并要求释放运输连接。
7. **窗口：**占2字节。窗口指的是发送本报文段的一方的接收窗口（而不是自己的发送窗口）。窗口值告诉对方：从本报文段首部中的确认号算起， 接收方目前允许对方发送的数据量（以字节为单位）。之所以要有这个限制，是因为接收方的数据缓存空间是有限的。**总之，窗口值作为接收方让发送方设置其发送窗口的依据。**
8. **检验和：**占2 字节。检验和字段检验的范围包括首部和数据这两部分。
9. **紧急指针：**占2字节。紧急指针仅在URG= 1时才有意义，它指出本报文段中的紧急数据的字节数（紧急数据结束后就是普通数据）。因此，紧急指针指出了紧急数据的末尾在报文段中的位置。当所有紧急数据都处理完时，TCP就告诉应用程序恢复到正常操作。值得注意的是，即使窗口为零时也可发送紧急数据。
10. **选项：**长度可变，最长可达40字节。当没有使用“ 选项“ 时，TCP的首部长度是20 字节。最后的填充字段仅仅是为了使整个TCP首部长度是4字节的整倍。

- **最大报文段长度 MSS**：MSS是每一个TCP报文段中的数据字段的最大长度。
- 窗口扩大选项
- 时间戳选项
- 选择确认选项

3.3 TCP 可靠传输的实现

为了讲述可靠传输原理的方便，我们假定数据传输只在一个方向进行，即A发送数据，B给出确认。

3.3.1 确认应答机制

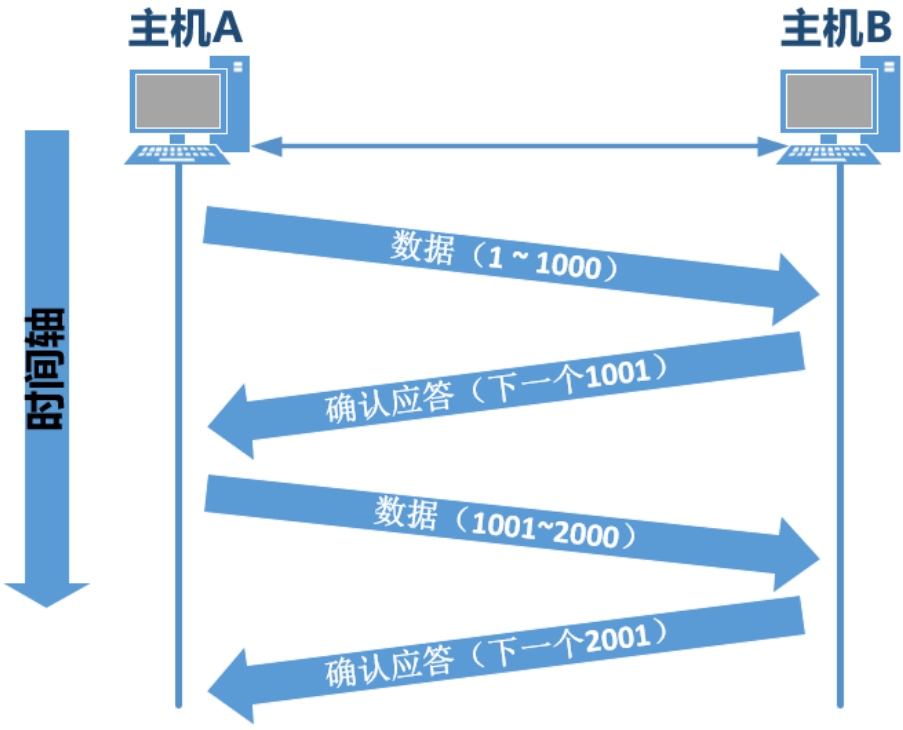
a. 确认应答 ACK

TCP为了保证报文传输的可靠，就给每个包一个序号，同时序号也保证了传送到接收端实体的包的按序接收。然后接收端实体对已成功收到的字节发回一个相应的确认（ACK）。

- 每一个ACK都带有对应的确认。
- 确认号告诉发送端已经收到了哪些数据。
- 确认号代表接收端下次需要接收的数据的序号。
- 确认号代表发送端要发送的数据的序号。

> 如图6，主机B给主机A发送了确认号为1001，说明：序号为1000的数据（包括1000）主机B已经收到了，下一次主机A需要发送序号为1001的数据。

图6：确认应答机制



b. 选择确认 SACK（了解）

若收到的报文无差错，只是未按序到达，通过选择确认 SACK机制可以只重传缺少的数据，而不重传已经正确到达接收方的数据。

如果需要使用选择确认 SACK，那么建立TCP连接时，就要在TCP首部中加上SACK选项，通信双方需要约定好。确认号用法不变，SACK选项用来报告收到的不连续的字节块的边界。

因为，SACK文档未明确指出发送方如何响应SACK，所以目前仍然是重传所有未被确认的数据块。

3.3.2 滑动窗口

a. 为什么采用滑动窗口?

> 为了提高传输效率，发送方可以不使用低效率的停止等待协议，而是采用流水线传输。流水线传输就是发送方可以连续发送多个分组，不必每发完一个分组就停下来等对方确认。这种方式可以获得很高的信道利用率。
>
> **TCP协议采用滑动窗口来提高信道利用率，从而提高传输效率。**

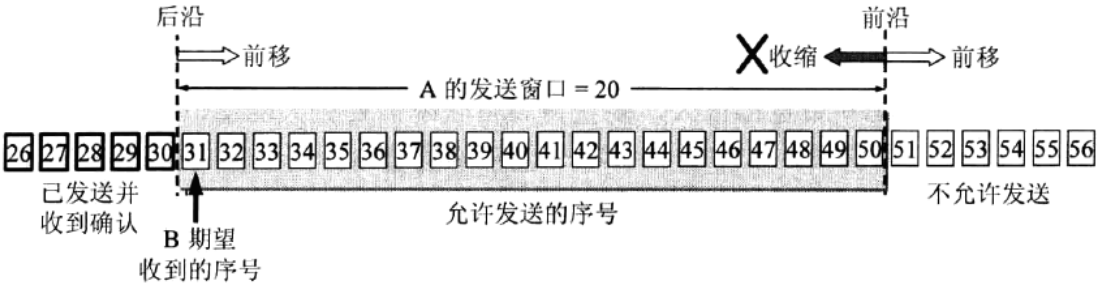
TCP 的滑动窗口是以字节为单位的。

---

b. 滑动窗口的实现

如图7，现假定A收到B发来的确认报文段，其中窗口是20字节，而确认号是31（表明B期望收到的下一个序号是31，而序号30为止的数据已经收到了）。

图7：A的发送窗口



发送窗口的特点：

- **发送窗口表示：**在没有收到B的确认的情况下，A可以连续把窗口内的数据都发送出去。已经发送，但未收到确认的数据都必须暂时保留，以便超时重传时使用。
- A的**发送窗口不能超过**B的**接收窗口数值。**
- 发送方的发送窗口大小还要受到网络拥塞程度的制约。
- 发送窗口后沿的后面部分表示已发送且已收到了确认，数据无需保留；发送窗口前沿的前面部分表示不允许发送的，因为接收方没有为这部分数据保留临时存放的缓存空间。
- 发送**窗口的位置**由窗口前沿和后沿的位置共同确定。

发送窗口的变化：

- 发送窗口后沿的变化：
    - 不动 —— 没有收到新的确认。
    - 前移 —— 收到了新的确认。
- 发送窗口前沿的变化：
    - 向前移动
    - 不动 —— 一是没有收到新的确认，对方通知的窗口大小也不变；二是收到了新的确认但对方通知的窗口缩小了，使得发送窗口前沿刚好不动。
    - 向后收缩 —— 对方通知的窗口缩小了。TCP不赞成这样做。

如图8，现在假定A发送了序号为31 ~ 41的数据。这时，发送窗口位置并未改变，但发送窗口内靠后面有11个字节（灰色方框）表示已发送但未收到确认。而发送窗口内靠前面的9（42~50）个字节是允许发送但尚未发送的。

图8：A发送11个字节的数据

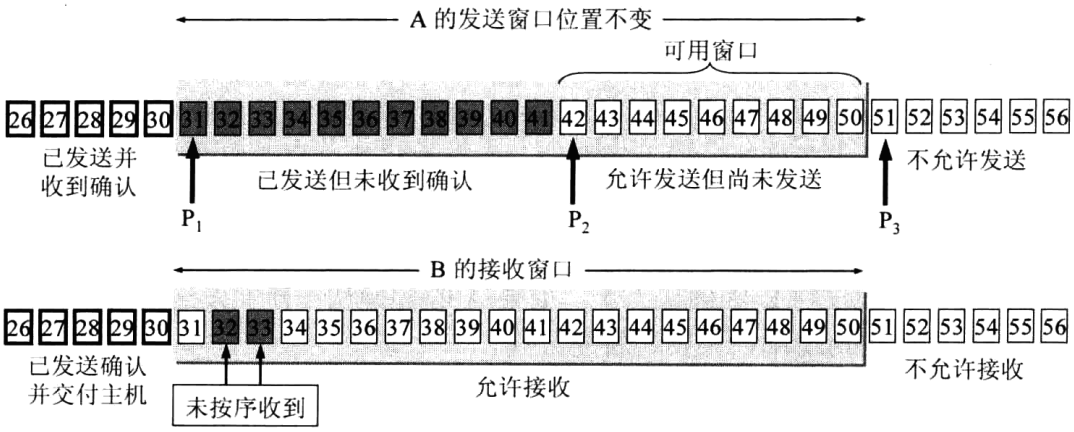


图9：A收到新的确认号，发送窗口向前滑动

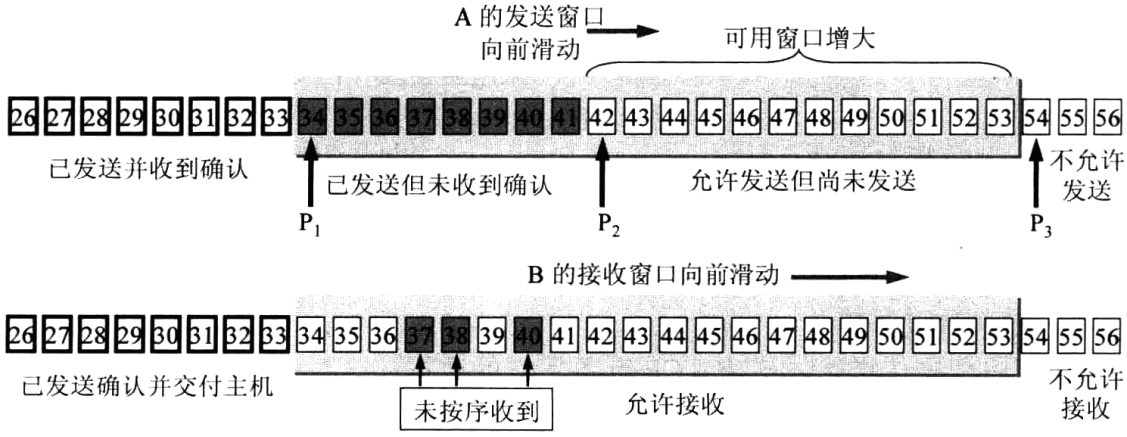
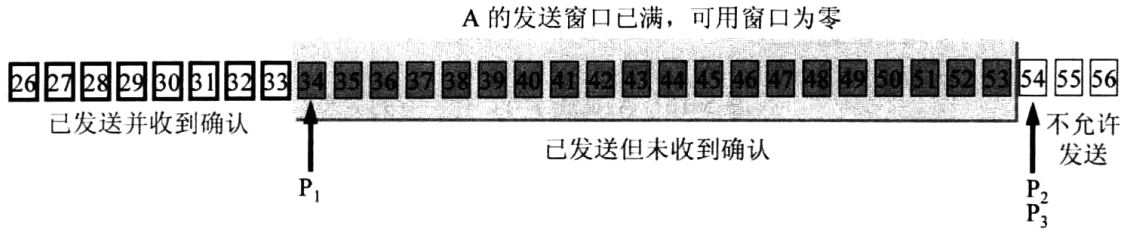


图10：发送窗口内的序号都已经发送，但未被确认



描述一个发送窗口的状态需要三个指针：P_1，P_2和P3。指针指向字节的序号。

- 小于P_1的是已发送并已收到确认的部分， 而大于P_3 的是不允许发送的部分。
- P_3 - P_1 = A的发送。
- P_2 - P_1 = 已发送但尚未收到确认的字节数。
- P_3 - P_2 = 允许发送但当前尚未发送的字节数（又称为可用窗口或有效窗口）。

接收窗口的特点：

- 已经接收并发送确认报文的数据交付主机后，不必保留。

> 如图8，在接收窗口外面，到30号的数据已经发送过确认并交付主机了。因此B可以不保留这些数据。下一个待接收的数据为31，此时B会给A发送确认报文段，报文段中的确认号为31。

- 若接收窗口内的数据没有按序到达，即某个数据未收到（可能丢失，也可能滞留在网络某处），B只能对按序收到的数据中的最高序号给出确认。

> 如图8，B的接收窗口收到了序号为32和33的数据，但是没有收到序号为31的数据，这意味着这两个数据未按序到达。所以B发送给A的确认报文段中的确认号为31（31即期望收到的序号，31之前均为已收到的的数据）。

- 数据接收成功后，滑动窗口将继续向前滑动。

> 如图9，假定B收到了序号为31的数据，并把序号为31~33的数据交付给主机，然后B删除这些数据。接着把接收窗口向前移动3个序号，同时给A发送确认，其中窗口值仍然为20，但确认号为34。表明B已经收到了到序号33未知的数据。

- 若数据没有按序到达，为按序到达的数据将暂存在接收窗口中。

> 如图9，B收到了序号为37,38和40的数据，但是未收到34,35,36和39的数据。所以序号为37,38和40的数据未按序到达，只能暂存在接收窗口，并发送确认号34，等待发送窗口发送未收到数据。

- 若A的发送窗口已满，必须停止发送，此时原因可能是这部分数据丢掉了，也可能是B收到了，但返回的确认号滞留在网络中，A会在一定时间后重传这部分数据（超时重传）。此时B会收到重复数据，TCP协议将根据序号去重。

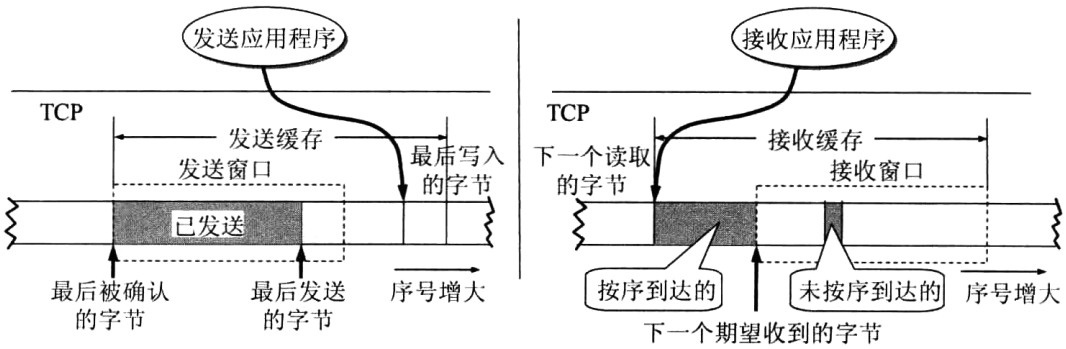
> 如图10，A在发送完序号42~53的数据后，指针P_2向前移动和P_3重合。发送窗口内的序号都已经用完，但是还是没有收到确认。由于A的发送窗口已经满了，可用窗口已经减小到零，因此必须停止发送。为了保证可靠传输，A只能认为B没有收到这部分数据，因此在一段时间后（超时计时器控制）就重传这部分数据，重新设置超时计时器，直到收到B的确认为止。A收到确认号落在发送窗口内，那么A就可以使发送窗口继续向前滑动，并发送新的数据

---

c. 窗口和缓存的关系

TCP连接的两端具有接收缓存和发送缓存，发送方的应用程序把字节流写入TCP的发送缓存，接收方的应用程序从TCP的接收缓存中读取字节流。那么缓存和窗口之间有什么关系呢?

图11：TCP的缓存和窗口的关系（左为发送方，右为接收方）



这里注意两点：

- 第一，缓存空间和序号空间都是有限的，并且都是循环使用的。
- 第二，实际上缓存或窗口中的字节数是非常大的，图11仅仅是示意图，未标出具体数字。

我们先看，图11所示的发送方情况。

发送缓存用来暂时存放：

- 发送应用程序传送给发送方TCP准备发送的数据；
- TCP已经发送但尚未收到确认的数据。

发送窗口通常只是发送缓存的一部分。

- 已被确认的数据应当从发送缓存中删除，因此发送缓存和发送窗口的后沿是重合的。
- 发送应用程序最后写入发送缓存的字节数 - 最后被确认的字节数 = 保留在发送缓存中被写入的字节数。
- 发送应用程序必须控制写入缓存的速率，不能太快，否则发送缓存就会没有存放数据的空间。

接下来看，图11所示的接收方情况。

接收缓存用来暂时存放：

- 按序到达的、但尚未被接收应用程序读取的数据。
- 未按序到达的数据。

接收窗口通常只是接收缓存的一部分。

- 如果收到的分组被检测出由差错，则要丢弃。
- 接收应用程序来不及读取收到的数据，接收缓存最终会被填满，使接收窗口减小到0。
- 接收应用程序若及时从接收缓存中读取收到的数据，接收窗口就可以增大。但最大不能超过接收缓存的大小。

此处需要强调：

- 第一，A的**发送窗口是根据**B的**接收窗口设置的**，但在同一时刻，A的发送窗口并不总和B的接收窗口一样大。因为通过网络传送窗口值需要经历一定的时间滞后。发送方A还可能根据网络当时的拥塞情况适当减小自己的发送窗口数值。
- TCP通常对不按序到达的数据是先临时存放在接收窗口中，等到字节流中所缺少的字节收到后，再**按序交付给上层的应用程序。**
- TCP要求接收方必须有累积确认的功能，这样可以减少开销。接收方可以在合适的时候发送确认，也可以在自己有数据发送时把**确认信息捎带**上。需注意两点：
    - 接收方不应过分推迟发送确认，否则会导致发送方进行不必要的重传，浪费网络资源。TCP标准规定，确认推迟的时间不应超过0.5秒。若接收一连串具有最大长度的报文段，需每隔一个报文段就发送一个确认。
    - 捎带确认实际上并不经常发生，因为大多数应用程序很少同时在两个方向上发送数据。
- TCP通信是全双工通信。通信双方都在发送和接收报文段，双方均有各自的发送窗口和接收窗口。

3.3.3 超时重传

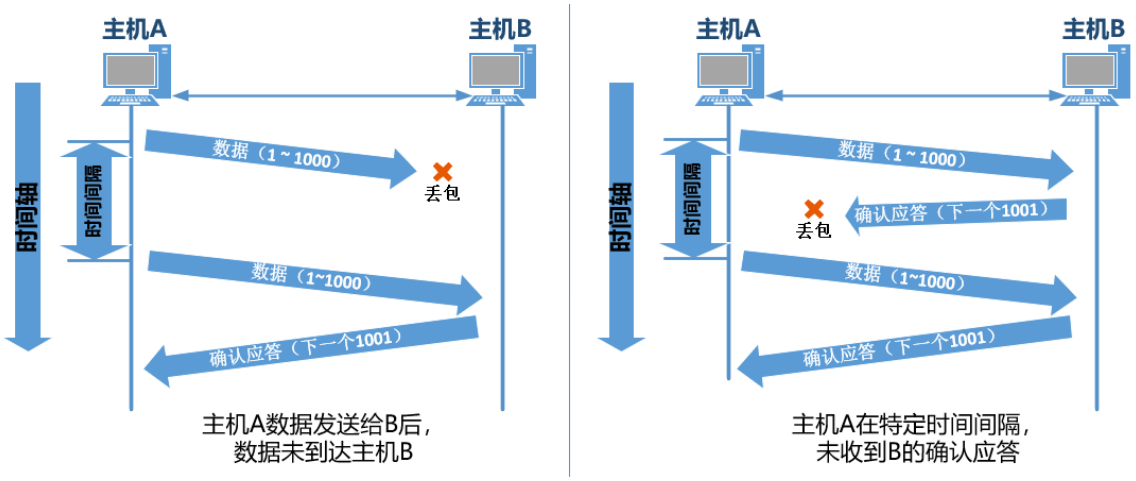
a. 超时重传的含义

超时重传：TCP的发送方在规定时间未收到接收方的确认就要重传已发送的报文段。

超时重传的原因：

- 主机A发送数据给主机B后，可能因为网络堵塞等原因，数据无法到达主机B；
- 如果主机A在一个特定时间间隔未收到B发来的确认应答（可能由于ACK丢失了），就会进行重发。

图12：超时重传原因



超时重传后，主机B会收到很多重复数据，TCP如何处理?

> 主机B收到重复数据后，TCP协议需要识别出哪些包是重复包，并将重复的包丢掉。利用序列号，可以很容易去重。

---

b. 如何选择超时重传时间?

TCP 为了保证无论在任何环境下都能比较高性能的通信，因此会动态计算这个最大超时时间。

> - **Linux中（Windows相同），超时以500ms为一个单位进行控制，每次判定超时重传时间都是500ms的整数倍。**
> - 如果重发一次之后，仍然得不到应答，等待2*500ms后在进行重传。
> - 如果仍然得不到应答，等待4*500ms进行重传，依次类推，以指数形式递增。
> - 累积到一定的重传次数，TCP认为网络或者对端主机出现异常，强制关闭连接。

以下进行详细说明，如若不感兴趣，可以跳过。

由于TCP下层是互联网环境，发送的报文段可能只经过一个高速率的局域网，也可能经过多个低速率的网络，并且每个IP数据报选择的路由可能不同。如果把**超时重传时间设置的太短，**就会**引起很多报文的不必要重传**，使得网络负荷增大。但如若**超时重传时间设置的过长**，则又使**网络的空闲时间增大，**降低了传输效率。

那么，应该如何选择?

TCP采用一种自适应算法，它记录一个报文段发出的时间，以及收到相应的确认的时间。这两个时间之差就是**报文段的往返时间RTT。**

$$RTT = \text{报文段发出时间} - \text{收到相应确认的时间}$$

TCP保留RTT的一个加权平均往返时间RTT_S(称为平滑的往返时间，S:Smoothed，表示平滑)。每当第一次测量到RTT样本，RTT_S值就取为所测量到的RTT样本值。以后没测量到一个新的RTT样本，按下式重新计算一次RTT_S。

$$\text{新的}\ RTT_s = (1-\alpha) \times (\text{旧的}\ RTT_s) + \alpha \times (\text{新的}\ RTT_{\text{样本}})$$

在上式中，$0 \leq \alpha < 1$。

- 若 α 接近0，表示新的RTT_S值和旧的RTT_S值相比变化不大，而对新的RTT样本影响不大（RTT值更新较慢）。
- 若 α 接近1，则表示新的RTT_S值受新的RTT样本影响较大（RTT值更新较快）。
- 标准的 RFC 6298推荐的 α 值为$1/8$，即0.125。

显然，超时计时器设置的超时重传时间 RTO 应略大于RTT_S。RFC6298建议使用下列计算RTO：

$$RTO = RTT_S + 4 \times RTT_D$$

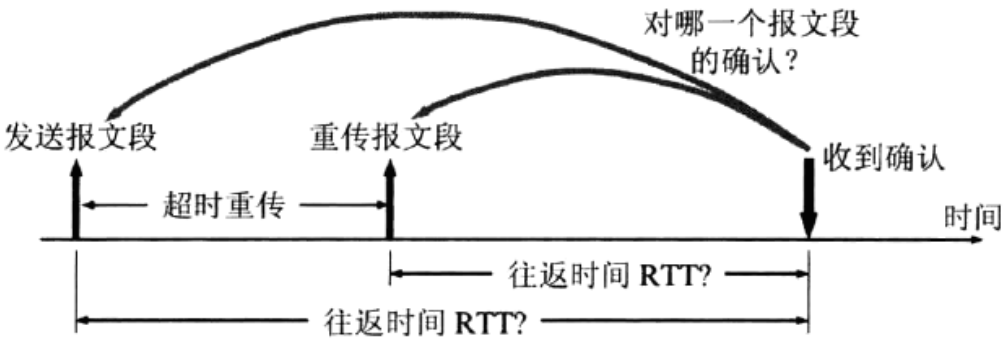
RTT_D是RTT的偏差的加权平均值，它与RTT_S和新的RTT样本只差有关。第一次测量时，RTT_D值取为测量到的RTT样本值的一半。后续计算公式如下：

$$\text{新的}\ RTT_D = (1-\beta) \times (\text{旧的}\ RTT_D) + \beta \times \left|RTT_S - \text{新的}\ RTT_{\text{样本}}\right|$$

这里β是一个小于1的系数，它的推荐值是$1/4$，即 0.25。

这里有一个新的问题：

图13：收到的确认是对哪一个报文段的确认?



如图13，假如A发送出报文段后，在重传时间内未收到确认。于是重传报文段。一段时间后收到了确认报文段。那么问题来了：**如何判断此确认报文段是对先发送的报文段的确认，还是对后来重传的报文段的确认?**正确的判断对确定加权平均值RTT_S的值关系很大。

> Karn提出一个算法：**在计算加权平均RTT_S时，只要报文段重传了，就不采用其往返时间样本。这样得出的加权平均RTT_S和RTO就较准确。**由于这样可能会导致超时重传时间无法更新，所以对此算法修正。方法是：**报文段每重传一次，就把超时重传时间RTO增大一些。**典型的做法是**新的重传时间为旧的重传时间的2倍。**当不再发生报文段的重传时才根据RTO公式计算超时重传时间。

3.4 流量控制

3.4.1 利用滑动窗口实现流量控制

接收端处理数据的速度是有限的。如果发送端发送数据太快, 导致接收端的缓冲区被打满， 这个时候如果发送端继续发送，就会造成丢包， 继而引起丢包重传等等一系列连锁反应。通过流量控制可以解决这一问题。

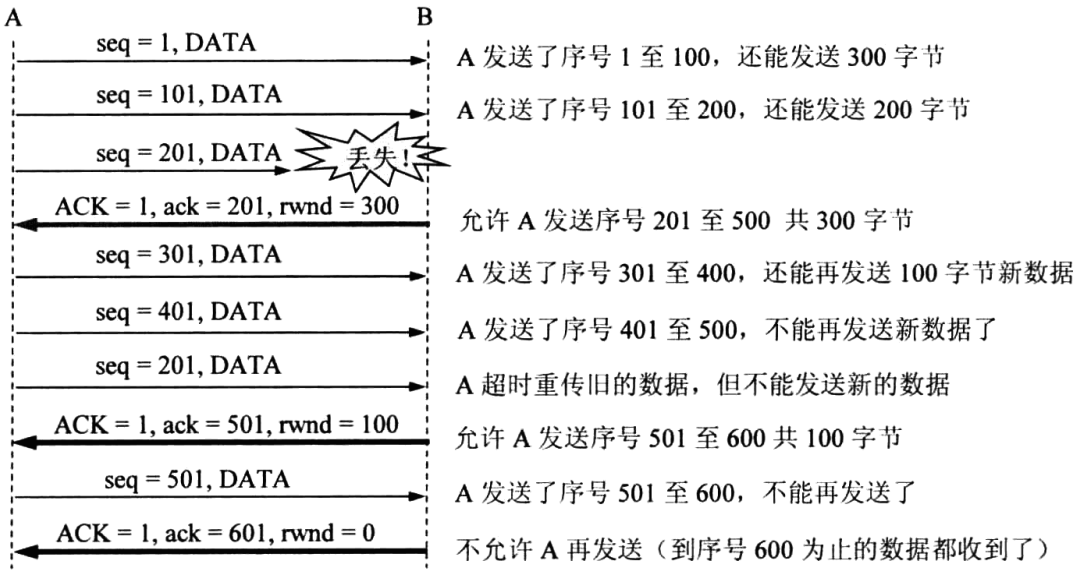
流量控制的目的：让发送方的发送速率不要太快，要让接受方来得及接收。

实现方法：利用滑动窗口机制。

> - 接收端将自己可以接收的缓冲区大小放入 TCP 首部中的“窗口大小”字段，通过ACK端通知发送端；
> - 窗口大小字段越大，说明网络的吞吐量越高；
> - 接收端一旦发现自己的缓冲区快满了，就会将窗口大小设置成一个更小的值通知给发送端；发送端接受到这个窗口之后，就会减慢自己的发送速度；
> - 如果接收端缓冲区满了，就会将窗口置为0；这时发送方不再发送数据，但是需要定期发送一个零窗口探测报文段（仅携带1字节数据），使接收端把窗口大小告诉发送端。

现在根据下图14，详细说明这一机制。

图14：利用滑动窗口进行流量控制（seq：序号 rwnd:接收端窗口值）



如图14，A向B发送数据。在TCP建立连接时，B会告诉A自己的接收窗口大小是多少？图中未表明这一过程。通过该图，我们可以看出B的rwnd(receiver window)接收窗口大小为400。

> - 如图14，接收方的主机B进行了三次流量控制。第一次为300，第二次为100，第三次为0。rwnd为0意味着接收方不能继续发送数据，此状态将维持到主机B重新发送一个新的窗口值为止。
> - 需注意：确认位ACK = 1时，确认号ack才有效。
> - 若B向A发送了零窗口的报文段之后，B的接收缓存又有了一些空间，于是B向A发送窗口值，但是这个报文段丢失了，此时A一直等待B发送的非零窗口的通知，而B也一直等A发送的数据。如果没有其他措施，此时会造成死锁局面。
> - TCP为每一个连接设有一个持续计时器。只要TCP连接的一方收到对方的零窗口通知，就启动持续计时器。持续计时器设置的时间到期，就发送一个零窗口探测报文段。对方就在确认这个探测报文段给出现在的窗口值。如果窗口值仍然为0，那么收到这个报文段的一方就重新设置持续计时器。如果窗口值不为0，死锁的僵局就可以打破了。

3.4.2 TCP 的传输效率

这里说两个影响TCP传输效率的问题。

问题一：如何控制TCP发送报文段的时机?

应用程序把数据传送到TCP的发送缓存之后，剩下的发送任务就由TCP控制。

可以用不同的机制来控制TCP报文段的发送时机。

> 1. 第一种机制，TCP维持一个变量，它等于最大报文段长度MSS。只要缓存中存放的数据达到MSS字节时，就组成一个TCP报文发送出去。
> 2. 第二种机制，由发送方的应用进程指明要求发送报文段，即TCP支持的推送（PUSH）操作。
> 3. 第三种机制，发送方的一个计时器期限到了，这时就把当前已有的缓存数据装入报文段（长度不能超过MSS）发送出去。

虽然有以上三种机制，但是如何控制TCP发送报文段的时机？仍然是一个较为复杂的问题。

在TCP的实现中广泛使用Nagle算法。算法如下：

> 若发送应用进程把要发送的数据逐个字节地送到TCP 的发送缓存， 则发送方就把第一个数据字节先发送出去， 把后面到达的数据字节都缓存起来。当发送方收到对第一个数据字符的确认后， 再把发送缓存中的所有数据组装成一个报文段发送出去， 同时继续对随后到达的数据进行缓存。只有在收到对前一个报文段的确认后才继续发送下一个报文段。当数据到达较快而网络速率较慢时， 用这样的方法可明显地减少所用的网络带宽。

Nagle算法还规定：

> 当到达的数据已达到发送窗口大小的一半或已达到报文段的最大长度时， 就立即发送一个报文段。这样做， 就可以有效地提高网络的吞吐量。

问题二：糊涂窗口综合征

另一个问题：做糊涂窗口综合征(silly window syndrome)，也会导致TCP的性能变差。

> 设想一种情况： TCP 接收方的缓存已满， 而交互式的应用进程一次只从接收缓存中读取1个字节（这样就使接收缓存空间仅腾出1个字节）， 然后向发送方发送确认，把窗口设置为1个字节（但发送的数据报是40字节长）。接着， 发送方又发来1个字节的数据（请注意， 发送方发送的IP数据报是41字节长）。接收方发回确认， 仍然将窗口设置为1个字节。这样进行下去， 使网络的效率很低。

解决方法：

> 让接收方等待一段时间， 使得或者接收缓存已有足够空间容纳一个最长的报文段， 或者等到接收缓存已有一半空闲的空间。只要出现这两种情况之一， 接收方就发出确认报文， 并向发送方通知当前的窗口大小。此外， 发送方也不要发送太小的报文段而是把数据积累成足够大的报文段， 或达到接收方缓存的空间的一半大小。

3.5 拥塞控制

3.5.1 拥塞控制的原理

a. 基本概念

拥塞：在计算机网络中的链路容量（即带宽）、交换节点中的缓存和处理机等，都是网络的资源。在某段时间。**若对网络中某一资源的需求超过了该资源所能提供的可用部分，网络的性能就会变差。**这种情况就叫做拥塞。出现网络拥塞的条件如下：

$$\sum 对资源的需求 > 可用资源$$

网络拥塞的原因:

网络拥塞往往是由许多因素引起的。

- 某个结点缓存的容量太小时，到达该点的分组因无存储空间暂存而不得不被丢弃。
- 处理机处理的速度太慢。
- 拥塞常趋于恶化。如路由器没有足够缓存空间，就会丢弃一部分分组。源点会重发这部分分组，甚至重传多次，这样会导致更多的分组流入网络和被路由器丢弃。此时，拥塞引起的重传加剧了网络的拥塞。

拥塞控制与流量控制的区别：

- **拥塞控制的目的**：拥塞控制就是**防止过多的数据注入到网络**中，这样可以使网络中的路由器或链路不至于过载。
- **拥塞控制所做的前提是：网络能够承受现有的网络负荷。**
- **拥塞控制是一个全局性的过程，**涉及到所有的主机、所有的路由器，以及与降低网络传输性能有关的所有因素。
- TCP连接的两端只要迟迟收不到对方的确认信息，就会猜想当前网络可能出现拥塞，但无法获知拥塞位置和拥塞的原因。
- 流量控制是指点对点通信量的控制，是一个端到端的问题。
- 流量控制所要做的就是要抑制发送端发送数据的速率，以便使接收端来得及接收。

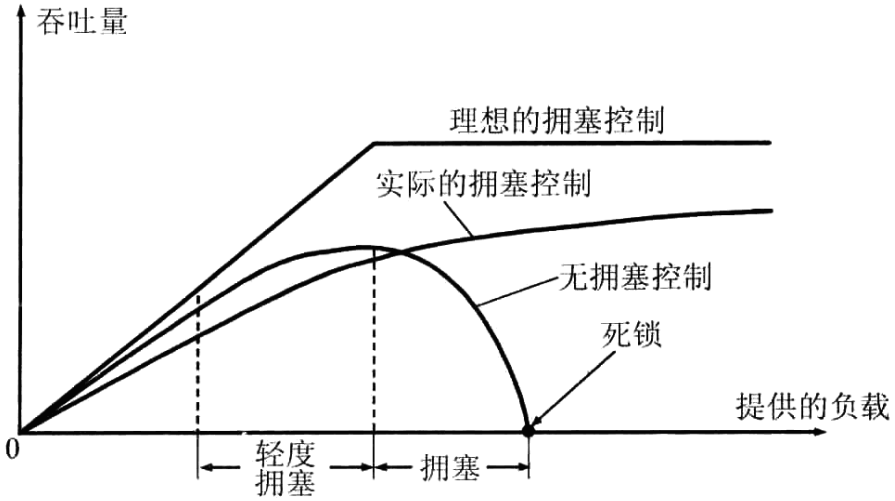
拥塞控制需要付出代价：

- 首先需要获取网络内部流量分布的信息。

- 在实施控制时，还需要在结点之间交换信息和各种命令，以便选择控制的策略和实施控制。
- 有时需要将一些资源（如缓存、带宽等）分配给个别用户或一类别的用户单独使用，这将导致网络不能更好的实现共享。

b. 拥塞控制的作用

图15：拥塞控制所起的作用



- 横坐标 ——提供的负载，代表单位时间内输入给网络的分组数量。提供的负载也称输入负载或者网络负载。
- 纵坐标 —— 吞吐量，代表单位时间内从网络输出的分组数目

从图中我们可以看出：

- 理想拥塞控制的网络，吞吐量饱和之前，网络吞吐量等于提供的负载。当提供的负载超过一定限度，因为资源有限，吞吐量达到饱和。此时表明提供的负载中有一部分丢失。但在理想的拥塞控制作用下，网络吞吐量依然维持在其所能达到的最大值。
- 实际网络中，随着提供的负载增大，网络吞吐量的增长速率逐渐减小。也就意味着，网络吞吐量未达到饱和之前，一部分输入分组被丢弃。
- 无拥塞控制的网络，当网络的吞吐量明显小于理想的吞吐量时，网络就进入了**轻度拥塞**的状态；当提供的负载达到某一数值时，网络的吞吐量随着提供的负载的增大而下降，这时网络就进入了**拥塞状态。**当提供的负载继续增大到某一数值时，网络的吞吐量就下降为0，网络已经无法工作，这就是所谓的**死锁。**

3.5.2 拥塞控制方法

TCP进行拥塞控制的算法有四种：**慢开始、拥塞避免、快重传、快恢复。**通过这四种算法的合作，达到对网络拥塞控制的目的。

发送方如何知道网络发生了拥塞?

> 判断网络拥塞的依据就是出现了超时。一般情况下，少量丢包判断为重传，大量丢包就认为网络堵塞。

拥塞控制也称为**基于窗口**的拥塞控制发送。为此，发送方维持一个叫做拥塞窗口cwnd的状态变量。拥塞窗口的大小取决于网络的拥塞程度，并且动态的变化。**发送方让自己的发送窗口等于拥塞窗口。**

发送方控制拥塞窗口的原则：只要网络没有出现拥塞，拥塞窗口就可以再增大一些，以便把更多的分组发送出去，这样就可以提高网络的利用率。只要网络出现拥塞且有可能出现拥塞，就必须把拥塞窗口减小一些，以减少网络中的分组数，以便缓解出现的拥塞。

慢开始算法的思路：当主机开始发送数据时，由于并不清楚网络的负荷情况呢，所以如果立即把大量数据字节注入网络，那么可能会引起网络发生拥塞。较好的方法是先探测一下，即由小到大逐渐增大发送窗口，也就是说，由小到大逐渐增大拥塞窗口数值。

初始拥塞窗口cwnd设置为不超过2至4个最大报文段SMSS的数值。具体规定如下：

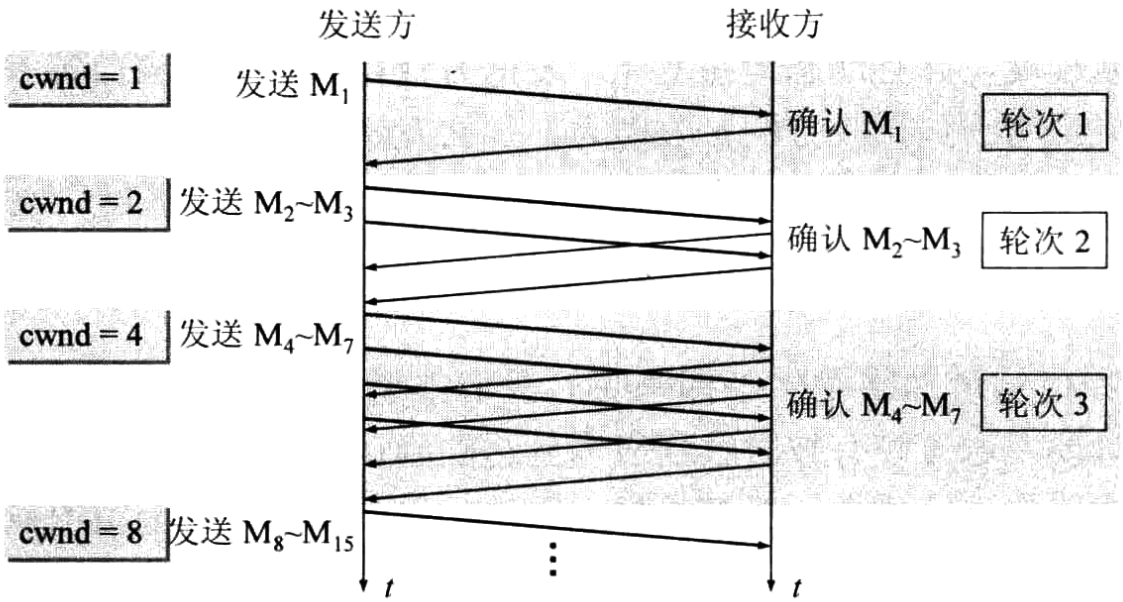
> - 若SMSS > 2190字节，则设置初始拥塞窗口cwnd = 2 * SMSS字节，且不能超多2个报文段。
> - 若1095字节 < SMSS <= 2190字节，则设置初始拥塞窗口cwnd = 3 * SMSS字节，且不得超过3个报文段。
> - 若SMSS <= 1095字节，则设置初始拥塞窗口cwnd = 4 * SMSS字节，且不得超过4个报文段。

慢开始规定，在每收到一个对新的报文段的确认后，可以把拥塞窗口增加最多一个SMSS的数值。就是

$$拥塞窗口cwnd每次的增加量 = min(N, SMSS)$$

其中N是原先未被确认的，但现在被刚收到的确认报文段所确认的字节数。

图16：慢开始示意图



> 如图16，一开始发送方设置cwnd=1，发送M1，接收方收到后确认M1。发送方收到确认后，把cwnd增大到2，于是发送方继续发送M2和M3。接受方收到后确认M2和M3。发送方每收到一个接收方的确认，就将发送方的拥塞窗口加1，以你发送方在收到两个确认后，cwnd从2变为了4。**使用慢开始算法后，每经过一个传输轮次，拥塞窗口cwnd就加倍。**

传输轮次：一个传输轮次所经历的时间就是往返时间RTT，更加强调，把拥塞窗口cwnd所允许发送的报文段度连续发送出去，并收到已发送的最后一个字节的确认。

> 如图16，cwnd = 4 时，往返时间 RTT 就是发送方连续发出4个报文段并收到这4个报文段的确认，总共经历的时间。

慢开始的"慢"并不是指cwnd的增长速率慢，而是指初始慢，但增长速率特别快。

为了防止拥塞窗口cwnd增长过大引起网络拥塞，还需要设置一个慢开始门限 ssthresh状态变量。**慢开始门限ssthresh的用法如下：**

- 当 cwnd < ssthresh时，使用慢开始算法。
- 当 cwnd > ssthresh时，停止使用慢算法而改用使用拥塞避免算法。
- 当 cwnd = ssthresh时，既可以使用慢开始算法，也可以使用拥塞避免算法。

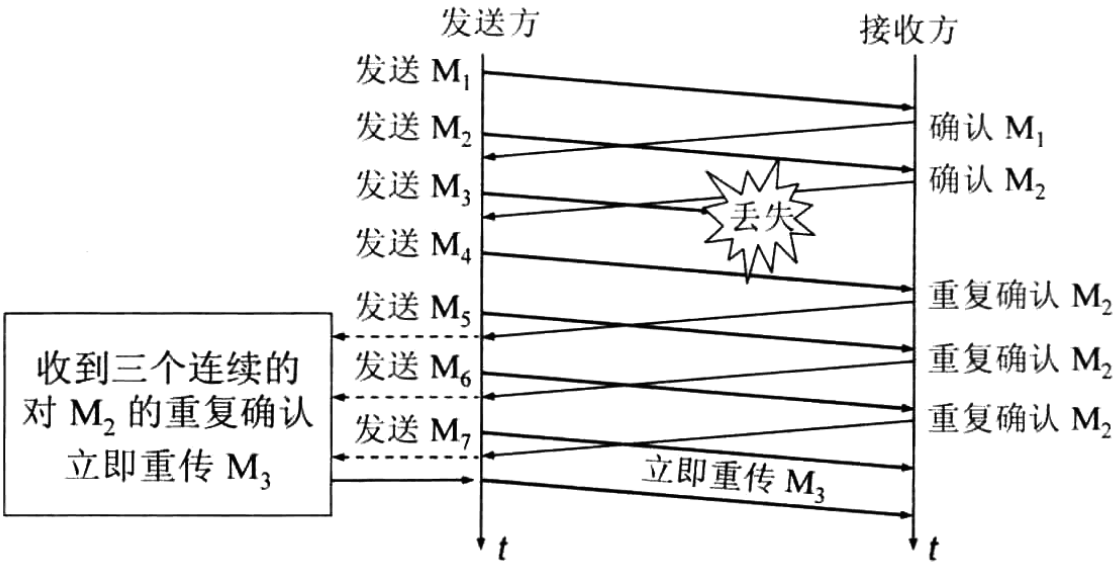
拥塞避免算法：让拥塞窗口 cwnd 缓慢的增大，即每经过一个往返时间就把发送方的拥塞窗口 cwnd 加1，而不是想慢开始算法那样加倍增长。

注意：拥塞避免是指把拥塞窗口控制为线性规律增长，使网络比较不容易出现拥塞。

有时，个别报文会在网络中丢失，但实际上网络并未发生拥塞。如果发送方迟迟收不到确认，就会产生超时，就会误认为网络发生了拥塞这就导致发送方错误的启动慢开始，把拥塞窗口cwnd设置为1，因此降低了传输效率。采用快重传算法可以让发送方尽早知道发生了个别报文段的丢失。

快重传算法：首先要求接收方不要等待自己发送数据时才进行捎带确认，而是立即发送确认，即使收到了失序的报文段也要立即发出对已收到的报文段的重复确认。快重传算法规定，发送方只要一连收到3个重复确认，就知道接收方没有收到报文段，因而应当立即进行重传。

图17：快重传示意图

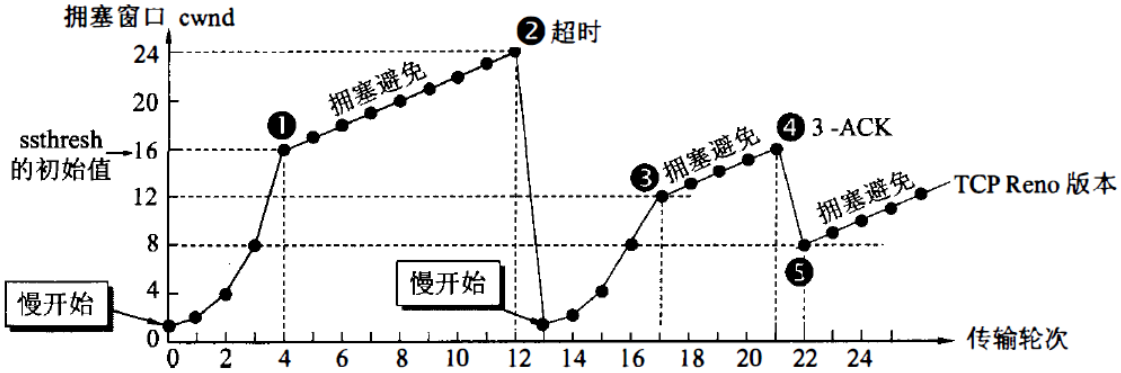


因为快重传算法，发送方知道现在只是丢失个别报文段。于是不启动慢开始，而是执行慢恢复算法。

慢恢复算法：当发送方收到三个重复的确认，就调整门限值ssthresh = cwnd / 2，然后设置拥塞窗口cwnd = ssthresh，并开始执行拥塞避免算法。

接下来通过具体例子来说明拥塞控制的过程。现假定TCP的发送窗口等于拥塞窗口。

图18：TCP拥塞窗口cwnd在拥塞控制时的变化情况

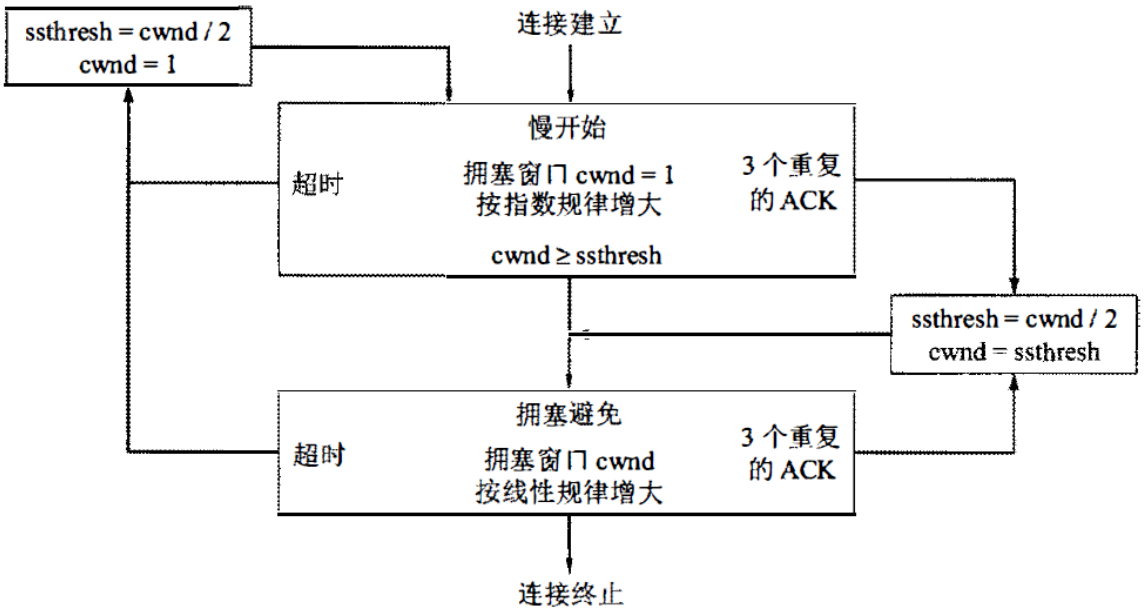


> 如图18，TCP开始连接时进行初始化，把拥塞窗口设置为1，慢开始门限的初始值设置为16个报文段 ，即ssthresh = 16。
>
> - 在执行慢开始算法时，发送方没收到一个对新报文段的确认ACK，就把拥塞窗口值加1，然后开始下一轮传输。因此拥塞窗口cwnd随着传输轮次按指数规律增长。

- 当拥塞窗口cwnd增长到慢开始门限值ssthresh时（图中点1，此时拥塞窗口 cwnd = 16），就该为执行拥塞避免算法，拥塞窗口按现行规律增长。
- 当拥塞窗口cwnd = 24时，网络出现了超时，发送方判断为网络拥塞。于是调整门限值ssthresh = cwnd / 2 = 12，同时设置拥塞窗口cwnd = 1，进入慢开始阶段。
- 接着，接着当拥塞窗口cwnd = ssthresh时，改为执行拥塞避免算法，拥塞窗口按线性规律增大。
- 当拥塞窗口cwnd= 16时，发送方连收3个对重复报文段的确认（3-ACK）。这是因为出现了个别报文段丢失，TCP采用快重传算法，提醒发送方重传丢失报文段，防止接收方错误判断为网络拥塞。
- 在点4，发送方知道丢失了个别的报文段。于是不启动慢开始，而是执行快恢复算法。这是发送方调整门限值ssthresh = cwnd/2=8，同时设置拥塞窗口cwnd = ssthresh = 8，并开始执行拥塞避免算法。

图18没有说明慢开始阶段如果出现了网络拥塞或者出现3-ACK，发送方应采取什么措施。下图给出发送方应采取什么措施。

图19：TCP的拥塞控制流程图



注意：发送方的发送窗口大小一定不能超过接收方的接收窗口大小。若将拥塞控制和流量控制一起考虑，发送方的窗口的上限值应当取为接收方窗口rwnd和拥塞窗口cwnd中较小的一个。

$$发送方窗口的上限值 = Min(rwnd, cwnd)$$

- rwnd < cwnd 时，是接收方的接受能力限制发送方窗口的最大值。
- cwnd < rwnd 时，则是网络的拥塞程度限制发送方窗口的最大值。
- 也就是说，rwnd 和 cwnd 中数值较小一个，控制了发送方发送数据的速率。

3.6 延迟应答和捎带应答

3.6.1 延迟应答

数据传输的时候，发送端给接收端发送数据，接收端给发送端发去确认应答信息，这样比较耗时，效率低下，延迟应答就是接收端收到数据之后，稍微等一会再应答，这样可以提高数据的传输效率，因为发送端发好几次数据，接收端只需要一次来确认应答，这样可以降低网络拥塞的概率。

如果接收数据的主机立刻返回ACK应答，这时候返回的窗口可能比较小。

- 假设接收端缓冲区为1M，一次收到了500K的数据; 如果立刻应答, 返回的窗口就是500K;

> - 但实际上可能处理端处理的速度很快，10ms之内就把500K数据从缓冲区消费掉了；
> - 在这种情况下，接收端处理还远没有达到自己的极限，即使窗口再放大一些，也能处理过来；
> - 如果接收端稍微等一会再应答，比如等待200ms再应答，那么这个时候返回的窗口大小就是1M；

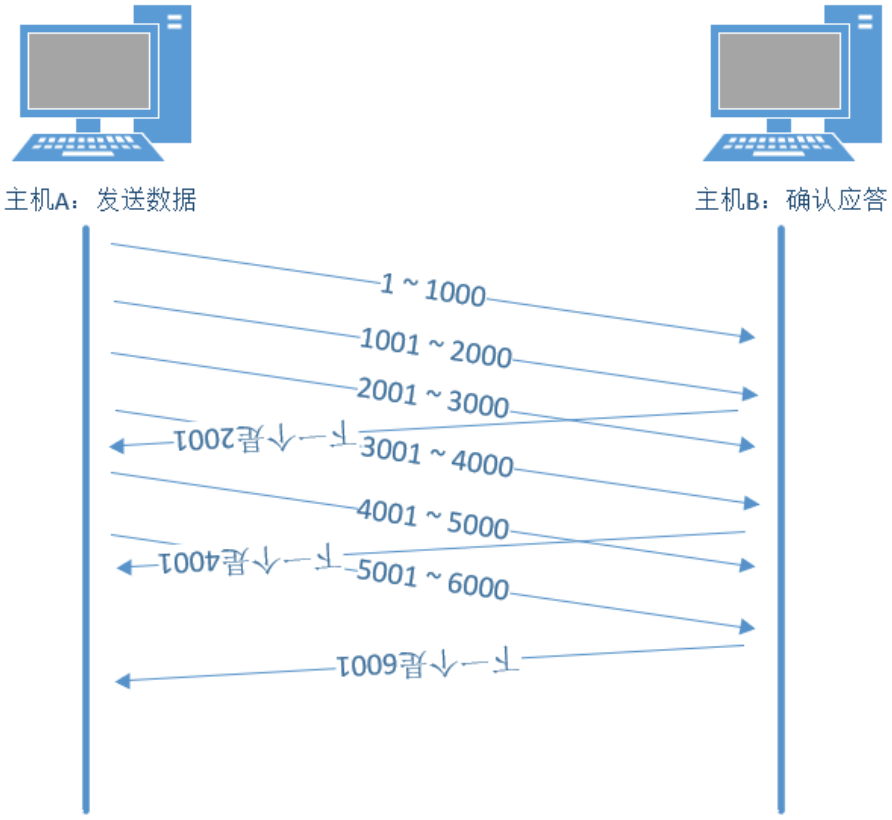
一定要记得，窗口越大，网络吞吐量就越大，传输效率就越高。我们的目标是在保证网络不拥塞的情况下尽量提高传输效率；

那么所有的包都可以延迟应答么？肯定也不是；

> - 数量限制: 每隔N个包就应答一次；
> - 时间限制: 超过最大延迟时间就应答一次。

具体的数量和超时时间，依操作系统不同也有差异；一般N取2，超时时间取200ms；

图20：延迟应答示意图

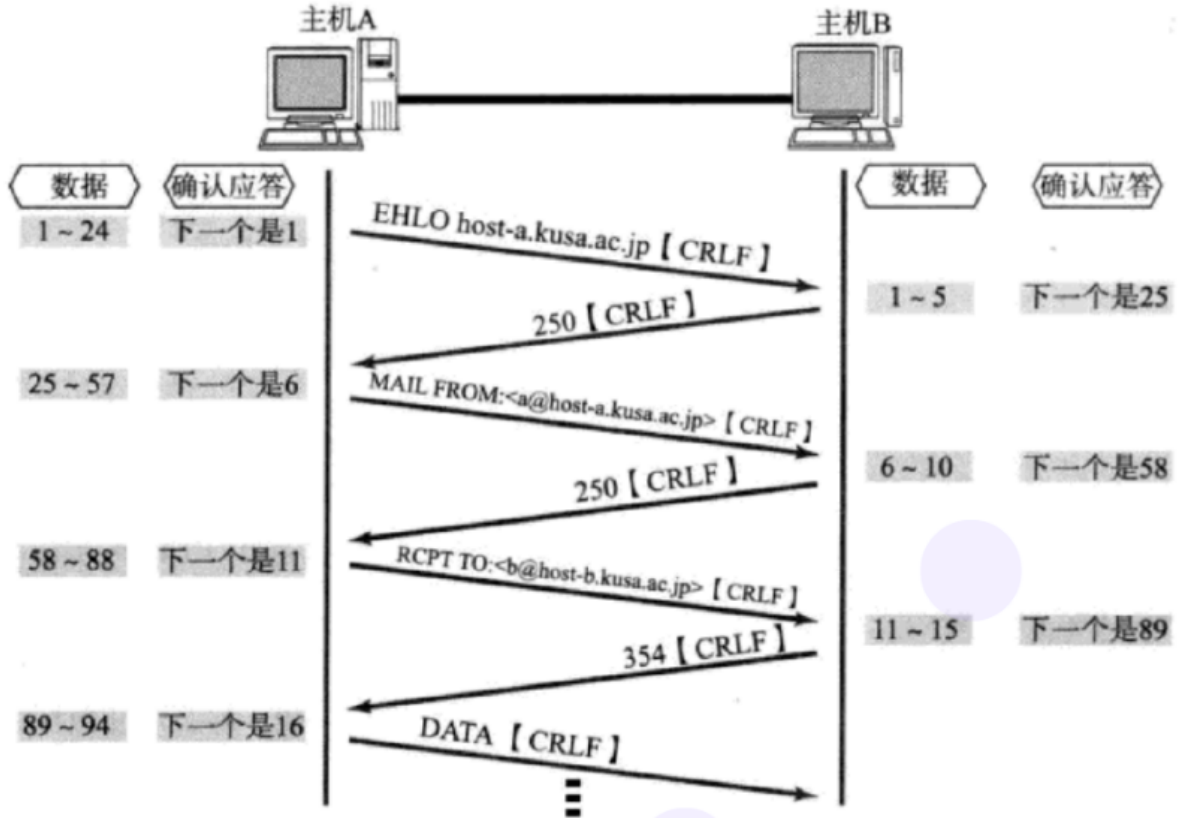


3.6.2 捎带应答

虽然有延迟应答，但是客户端和服务器在应用层还是一发一收，此时就会导致数据传输效率低下，**捎带应答就是接收端在给发送端发送数据的时候，捎带着向发送端发去确认应答，应答的内容是接收端已经收到发送端发送的数据。**

> 意味着客户端给服务器说了 "How are you"，服务器也会给客户端回一个 "Fine, thank you"；那么这个时候ACK就可以搭顺风车，和服务器回应的 "Fine, thank you" 一起回给客户端。

图21：捎带应答



3.7 TCP 连接管理机制

TCP是面向连接的协议。TCP连接的建立与释放是每一次面向连接的通信必不可少的过程。传输连接有三个阶段，即：**连接建立、数据传输和连接释放。**

在TCP连接建立过程中需要解决以下三个问题：

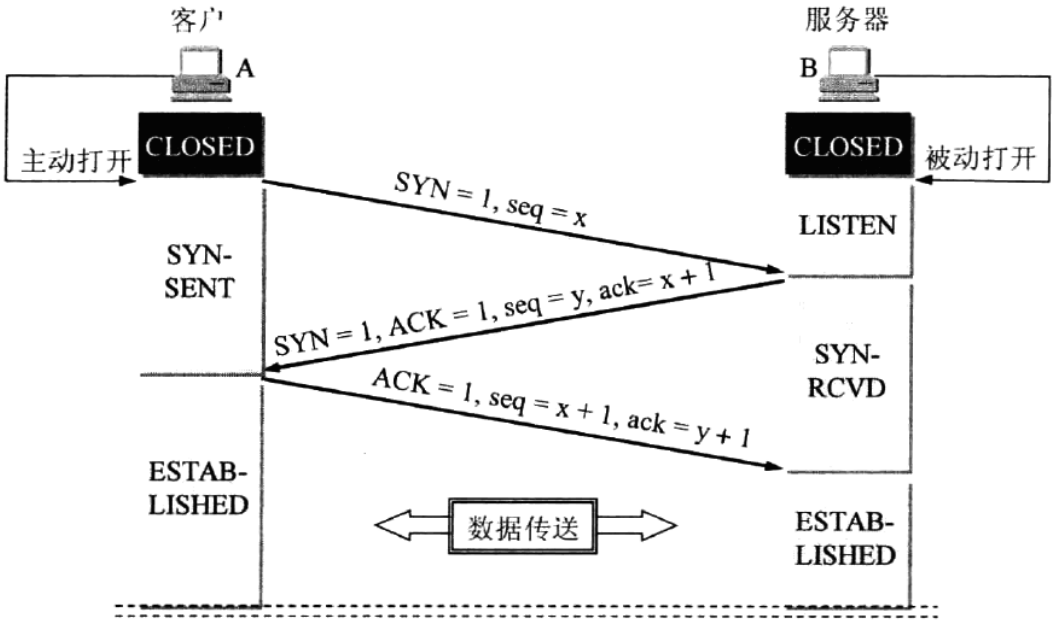
1. 要使每一方能够确知对方的存在。
2. 要允许双方协商一些参数（如最大窗口值、是否使用窗口扩大选项和时间戳选项等）。
3. 能够对传输实体资源（如缓存大小、连接表中的项目等）进行分配。

TCP连接机制采用的是客户-服务器方式。主动发起连接建立的应用程序叫做**客户端**，而被动等待连接的应用进程叫做**服务器**。

3.7.1 建立连接（三次握手）

如下图22，假定主机A运行的是客户端程序，而主机B运行的是TCP服务器程序。最初，客户端和服务器的TCP进程都处于 CLOSED（关闭）状态。

图22：TCP建立连接过程 - 三次握手



三次握手过程：

- 一开始，B的TCP服务器进程先创建传输控制块TCB，准备接受客户端进程的连接请求。然后服务器进程就处于LISTEN（监听）状态，等待客户端的连接。A的TCP客户进程也是首先创建传输控制块TCB。

  > 传输控制块TCB存储了每一个链接中的重要信息。如TCP连接表，指向发送和接收缓存的指针，指向重传队列的指针，当前发送和接收序号等。

- **A在打算建立TCP连接时，向B发送连接请求报文段。**该报文段首部中的同步位`SYN = 1`，同时选择一个初始序号`seq = x`。这时，TCP连接的客户端进入`SYN_SENT`（同步已发送）状态。TCP规定，SYN报文段（`SYN=1`的报文段）不能携带数据，但是要消耗一个序号。
- **B收到A的连接请求报文段后，则向B发送确认。**在确认报文段中应把SYN位和ACK位都置1，即`SYN=1`，`ACK = 1`，确认号为`ack = x + 1`，同时为自己选择一个初始序号`seq = y`。这时TCP服务器进入`SYN_RCVD`（同步收到）状态。注意，这个报文段也不能携带数据，但同样要消耗一个序号。

  > 此处，B发送给A的报文段，也可以拆分为两个报文段。先发送一个确认报文段(ACK=1，ack=x+1)，然后在发送一个同步报文段(SYN=1,seq=y)。此时变为四次握手，但效果一样。

- **TCP客户进程收到B的确认后，还要向B发出确认。**在确认报文段中，`ACK = 1`，确认号`ack = y + 1`，而自己的序号`seq = x + 1`。这时TCP连接已经建立，A进入`ESTABLISHED`（已建立连接）状态。当B收到A的确认后，也进入`ESTABLISHED`状态。TCP标准规定，此处ACK报文段可以携带数据。但如果不携带数据则不消耗序号，在这种情况下，下一个数据报文段的序号还是`seq = x + 1`。

为什么A最后还要发送一次确认呢?

主要是为了防止已失效的连接请求报文段突然又传输到了B，因而产生错误。

> 所谓"已失效的连接请求报文段"是这样产生的。
>
> - 假如只进行两次握手，A发出的第一个连接请求报文段，因某些原因在某些网络结点长时间滞留了。
> - 但由于A在规定时间内未收到该确认报文段，所以误以为报文丢失，于是第二次重新发送请求连接报文段。
> - 重传之后，连接建立成功，数据传输完毕，就释放了连接。

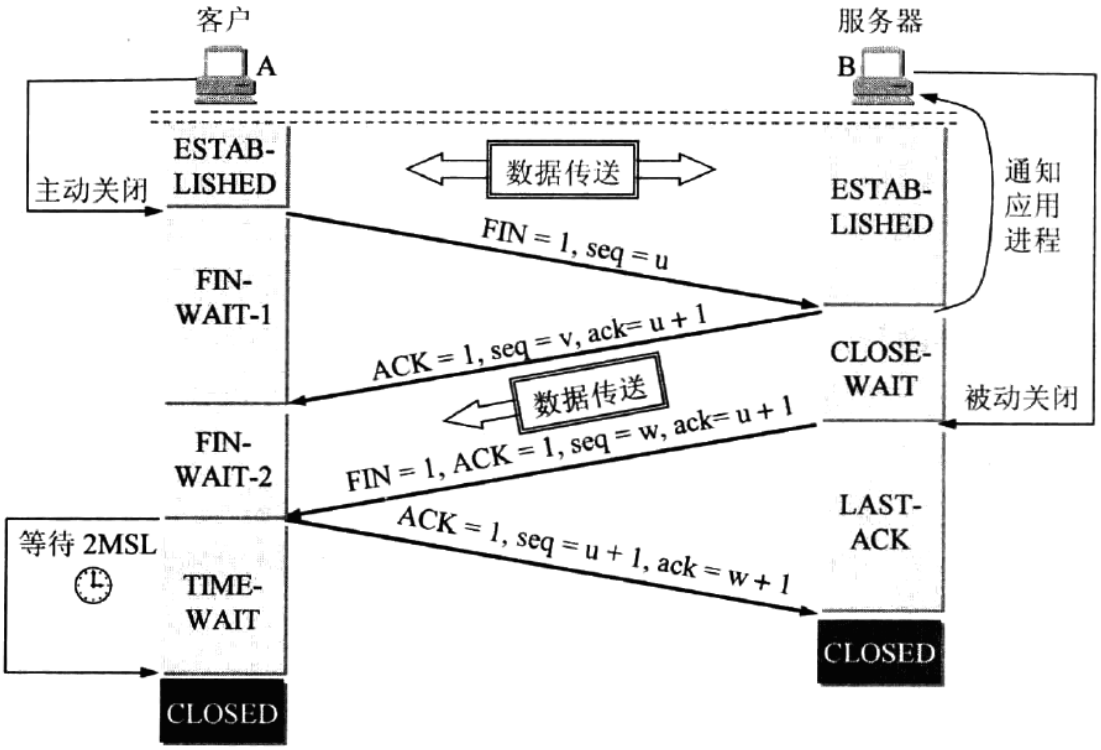
> - 释放连接以后的某个时间，第一次发送的连接请求报文段到达了B，B误以为A又发送了一次新的连接请求，于是向A发送确认报文段，同意建立连接。
> - 这时候，新的连接又建立了，但是A并没有请求建立连接，因此不会理睬B的确认，也不会向B发送数据。但B却以为新的连接建立成功，并一直等待A发送数据。这样白白浪费了资源。

采用三次握手，可以防止这类现象发生。例如在上述情况中，A不会向的确认发出确认。由于B收不到确认，就知道A 并没有请求建立连接。

3.7.2 释放连接（四次挥手）

数据传输完毕后，通信的双方都可以释放连接。现在A和B都处于`ESTABLISHED`状态。

图23：TCP释放连接过程 - 四次挥手



四次挥手过程：

- **客户端A发送连接释放报文段，并停止发送数据，主动关闭TCP连接。**A的释放连接报文段首部的终止控制为`FIN`置1（`FIN= 1`），其序号`seq = u`，它等于前面已经发送过的数据的最后一个字节加1。此时，A进入`FIN_WAIT_1`（终止等待1）状态，等待B的确认。TCP规定，FIN报文段即使不携带数据也要消耗一个序号。
- **B收到连接释放报文段后，立即发出确认报文段。**报文段中，确认位`ACK = 1`，确认号为`ack = u + 1`，这个报文段自己的序号为`v`，等于B前面已经传送过的数据的最后一个字节加1。然后B进入`CLOSE_WAIT`（关闭等待）状态。TCP服务器进程这时通知高层应用进程，因为A到B方向上的连接释放了，这时候TCP处于半关闭状态，即A没有数据要发送了，但B若要发送A仍要接受。也就是说B到A的这个方向的连接并未关闭，这个状态可能会持续一段时间。
- **A收到B的确认后，就进入`FIN_WAIT_2`（终止等待2）状态，等待B发送的连接释放报文段。**
- **当B没有像A发送的数据，B就需要释放连接，发出连接释放报文段。**这时B发出连接释放报文段必须使`FIN = 1`。现假定B的序号为`seq=w`（在半连接状态B可能还发送了一些数据）。B必须重复上次已经发送过的确认号`ack=u+1`，`ACK=1`。
- **这时B进入`LAST_ACK`（最后确认）状态，等待A的确认。**

- **A在收到B的连接释放报文段后，必须对此发出确认，发出确认报文段。**在确认报文段中，确认位：`ACK = 1`，确认号`ack=w+1`，而自己的序号为`seq=u+1`（根据TCP标准，前面发送过的FIN报文段要消耗一个序号）。然后进入`TIME_WAIT`（时间等待）状态。此时，TCP连接还没有释放掉。必须经过时间等待计时器设置的时间2MSL后，A才进入到`CLOSED`状态。

  > TCP协议规定,主动关闭连接的一方要处于TIME_ WAIT状态，等待2个MSL(maximum segment lifetime)的时间后才能回到`CLOSED`状态。
  >
  > **MSL称为最长报文段寿命，**RFC 793建议设置为2分钟（规定为两分钟,但是各操作系统的实现不同, 在Centos7上默认配置的值是60s）。因此，从A进入到`TIME_WAIT`状态必须等待4分钟后才能进入`CLOSED`状态，才能开始建立下一个新的连接。当A撤销相应的传输控制块TCB后，就结束了这次的TCP连接。
  >
  > 可以通过`cat /proc/sys/net/ipv4/tcp_fin_timeout`查看MSL的值;

- **B收到A发出的确认后，就进入`CLOSED`状态。**

为什么在TIME_WAIT状态必须等待2MSL的时间呢?

1. **第一，为了保证客户端发送的最后一个ACK报文段能够到达B。**

   > 这个ACK报文段有可能丢失，因而使处在`LAST_ACK`状态的B收不到对已发送的确认报文段。B会超时重传这个`FIN+ACK`报文段，而A就能在2MSL时间内收到这个重传的`FIN+ACK`报文段。接着A 重传一次确认报文段， 重新启动2MSL计时器。最后，A和B都正常进入到`CLOSED`状态。如果A在`TIME_WAIT`态不等待一段时间， 而是在发送完ACK报文段后立即释放连接， 那么就无法收到B重传的FIN+ ACK报文段， 因而也不会再发送一次确认报文段。这样， B就无法按照正常步骤进入`CLOSED`状态。

2. **第二，防止" 已失效的连接请求报文段" 出现在本连接中。**

   > A在发送完最后一个ACK 报文段后，再经过时间2MSL, 就可以使本连接持续的时间内所产生的所有报文段都从网络中消失。这样就可以使下一个新的连接中不会出现这种旧的连接请求报文段。B只要收到了A发出的确认，就进入CLOSED 状态。同样，B在撤销相应的传输控制块TCB后，就结束了这次的TCP连接。我们注意到，B结束TCP连接的时间要比A早一些。

解决`TIME_WAIT`状态引起的bind失败的方法?

在server的TCP连接没有完全断开之前不允许重新监听，某些情况下可能是不合理的。服务器需要处理非常大量的客户端的连接(每个连接的生存时间可能很短, 但是每秒都有很大数量的客户端来请求)。这个时候如果由服务器端主动关闭连接(比如某些客户端不活跃，就需要被服务器端主动清理掉)，就会产生大量TIME_WAIT连接。由于我们的请求量很大，就可能导致TIME_WAIT的连接数很多, 每个连接都会占用一个通信五元组(源ip,源端口, 目的ip, 目的端口, 协议)。其中服务器的ip和端口和协议是固定的。如果新来的客户端连接的ip和端口号和TIME_WAIT占用的链接重复了，就会出现问题，bind失败。

解决方法：使用`setsockopt()`函数设置socket描述符的 选项`SO_REUSEADDR`为1, 表示允许创建**端口号相同**但**IP地址**不同的多个socket描述符。

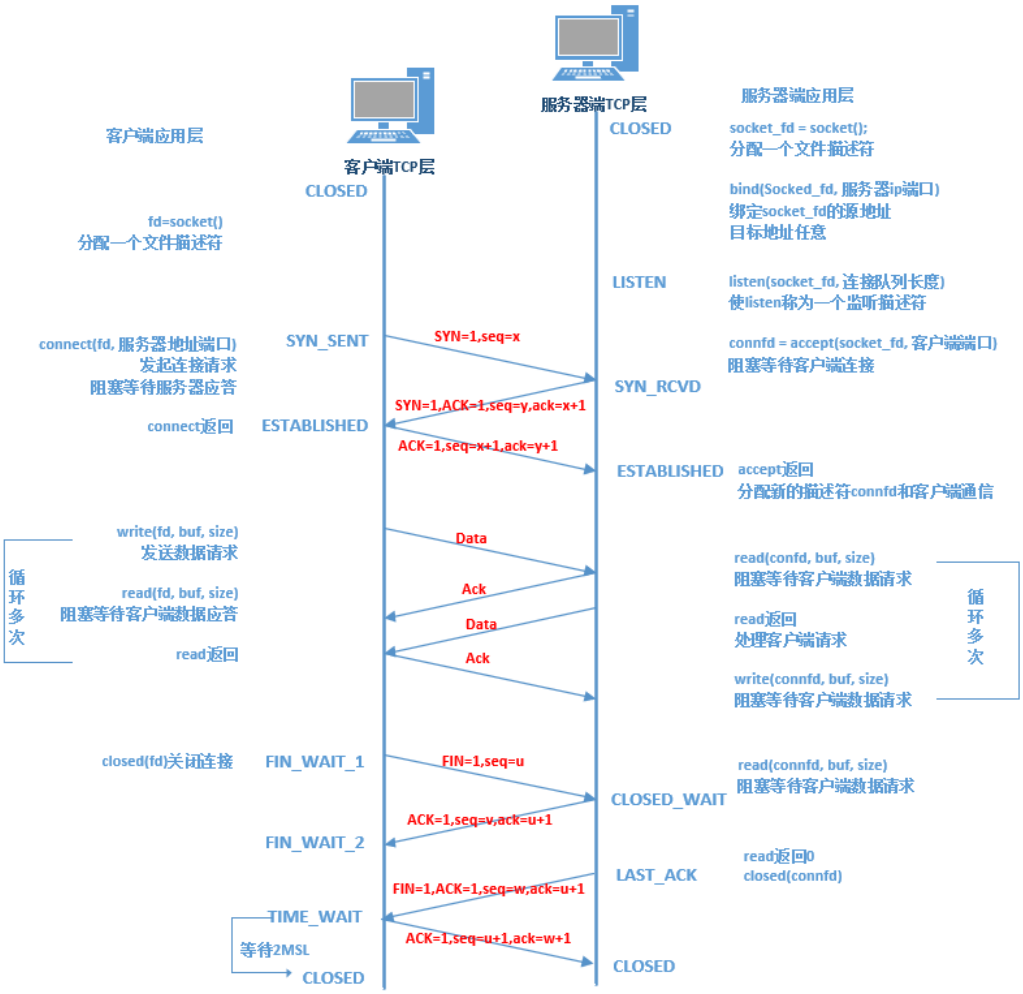
setsockopt()函数，用于任意类型、任意状态套接口的设置选项值。尽管在不同协议层上存在选项，但本函数仅定义了最高的"套接口"层次上的选项。 想要更多了解可以去查阅文档。

```
int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);
```

3.7.2 状态转化

接下来我们看一下完整的TCP连接和释放时，客户端与服务器端各自的状态转化。

图24：三次握手和四次挥手



服务端状态转化：

- [CLOSED -> LISTEN] 服务器端调用listen后进入LISTEN状态，等待客户端连接；
- [LISTEN -> SYN_RCVD] 一旦监听到连接请求(同步报文段)，就将该连接放入内核等待队列中，并向客户端发送SYN确认报文。
- [SYN_RCVD -> ESTABLISHED] 服务端一旦收到客户端的确认报文，就进入ESTABLISHED状态，可以进行读写数据了。
- [ESTABLISHED -> CLOSE_WAIT] 当客户端主动关闭连接(调用close)，服务器会收到结束报文段，服务器返回确认报文段并进入CLOSE_WAIT；
- [CLOSE_WAIT -> LAST_ACK] 进入CLOSE_WAIT后说明服务器准备关闭连接(需要处理完之前的数据)；当服务器真正调用close关闭连接时，会向客户端发送FIN，此时服务器进入LAST_ACK状态，等待最后一个ACK到来(这个ACK是客户端确认收到了FIN)。
- [LAST_ACK -> CLOSED] 服务器收到了对FIN的ACK，彻底关闭连接。

客户端状态转化：

- [CLOSED -> SYN_SENT] 客户端调用connect，发送同步报文段；
- [SYN_SENT -> ESTABLISHED] connect调用成功，则进入ESTABLISHED状态，开始读写数据；
- [ESTABLISHED -> FIN_WAIT_1] 客户端主动调用close时，向服务器发送结束报文段，同时进入
  FIN_WAIT_1；
- [FIN_WAIT_1 -> FIN_WAIT_2] 客户端收到服务器对结束报文段的确认，则进入FIN_WAIT_2，开始等待服务器的结束报文段；
- [FIN_WAIT_2 -> TIME_WAIT] 客户端收到服务器发来的结束报文段，进入TIME_WAIT，并发出LAST_ACK；

> - [TIME_WAIT -> CLOSED] 客户端要等待一个2MSL(Max Segment Life，报文最大生存时间)的时间，才会进入CLOSED状态。

下图是TCP状态转化的汇总。

图25：TCP的有限状态机

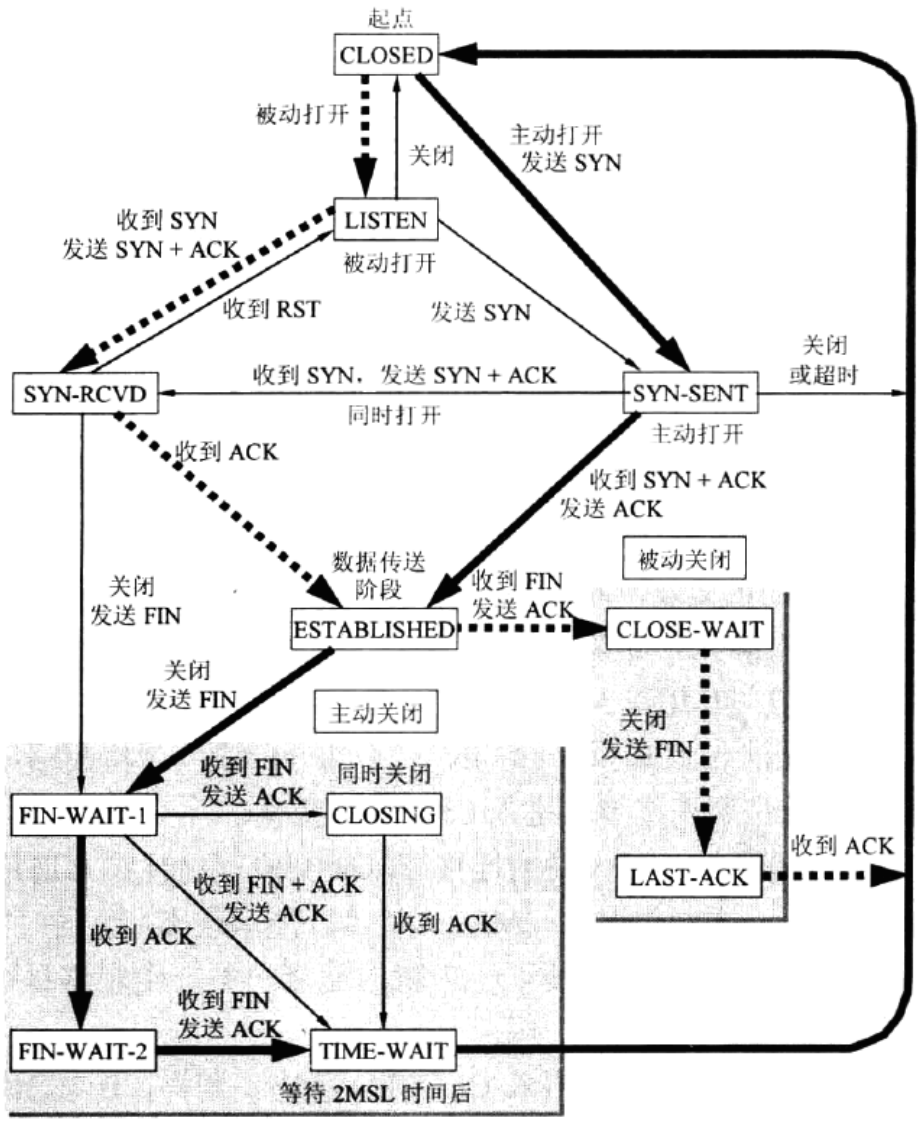


图25中：

- 粗实线箭头所指的是客户进程的状态变迁
- 粗虚线箭头所指的是服务器进程的状态变迁。