

# 第 5 章 运输层

运输层的基本功能

TCP、UDP协议

# 运输层

- 运输层
  - 运输层概述
  - 运输层的作用
  - 复用与分用
  - TCP与UDP

运输层协议概述

用户数据报协议 UDP

可靠传输的工作原理

传输控制协议 TCP 概述

TCP 报文段的首部格式

TCP 可靠传输的实现

TCP 的流量控制

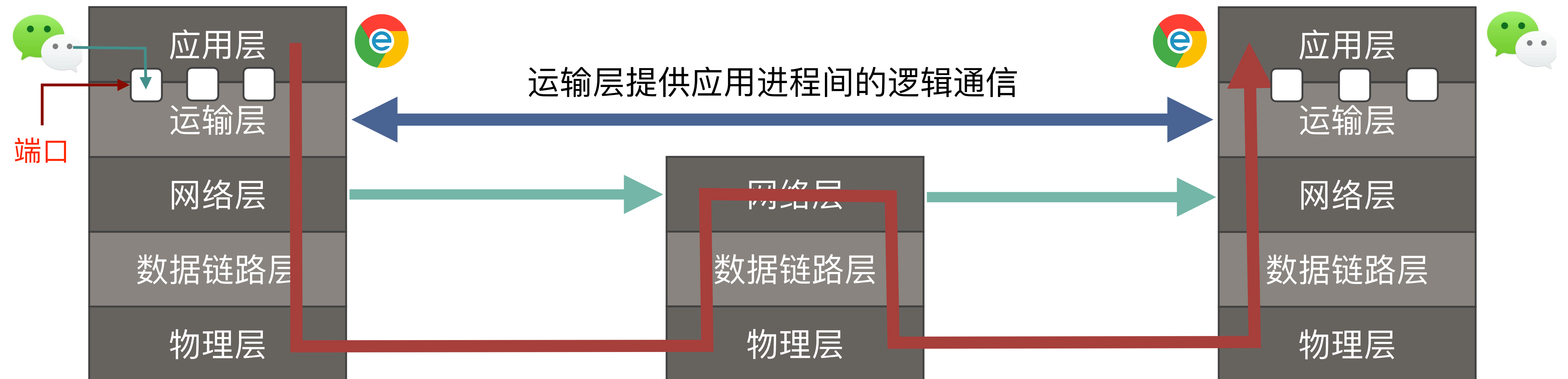
TCP 的拥塞控制

TCP 的运输连接管理

# 为什么需要运输层

- 运输层
  - 运输层概述
  - 运输层的作用
  - 复用与分用
  - TCP与UDP
- 网络层最终解决的问题：
  - 分组从一台主机经过网络到达另一台主机，即主机到主机间的通信。
- 网络层没有解决的问题：
  - 主机中谁发送的数据、谁接收数据？
  - IP分组无序到达目的主机，接收进程如何处理？
  - 可靠传输问题

# 运输层的作用





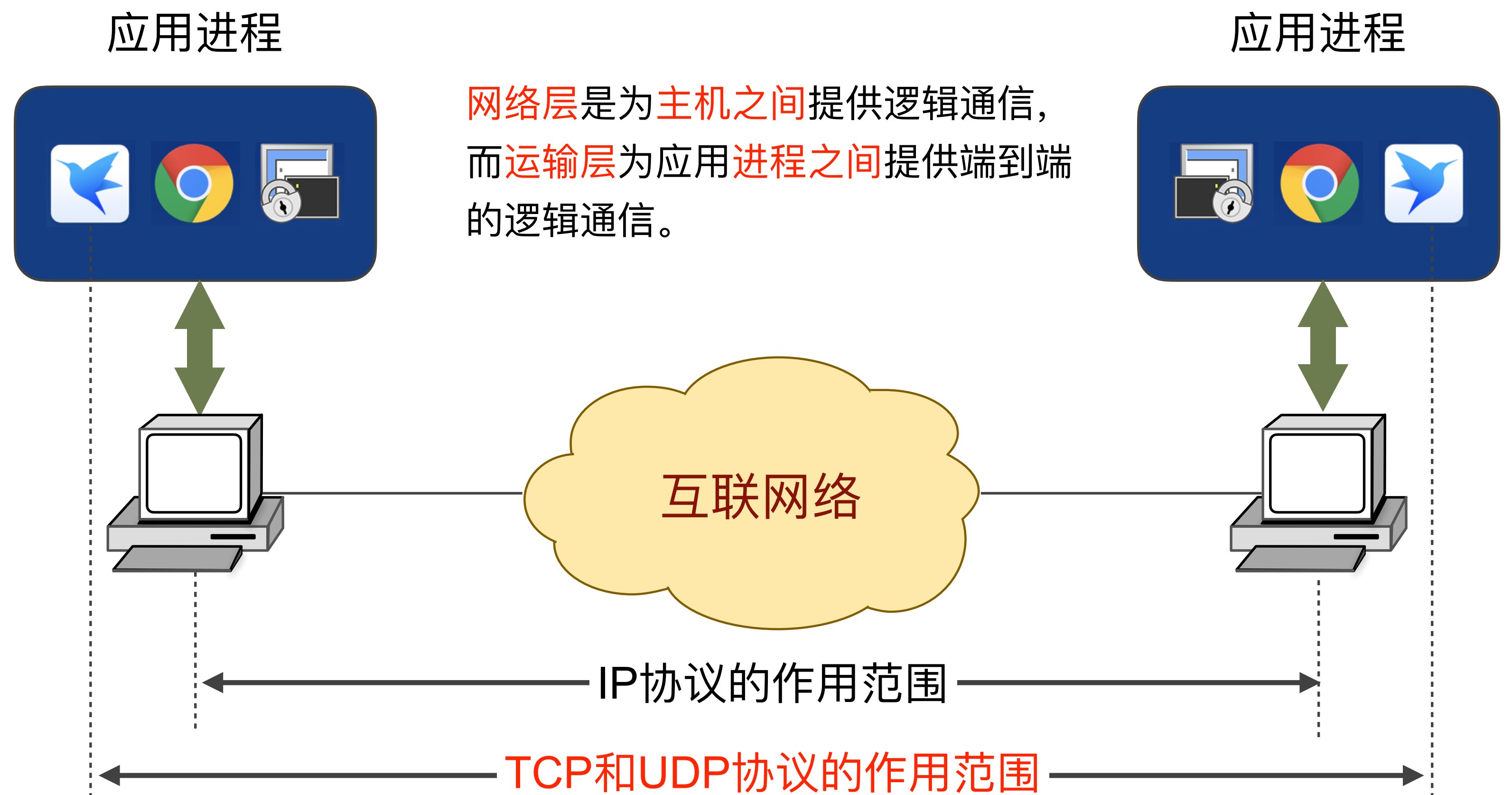
# 运输层的作用

- 运输层
  - 运输层概述
  - 运输层的作用
  - 复用与分用
  - TCP与UDP

- “**逻辑通信**”的意思是“**好像是这样通信**，但事实上并非真的这样通信”：
  - 从**IP层**来说，**通信的两端是两台主机**。但“两台主机之间的通信”这种说法还不够清楚；
  - 严格地讲，两台主机进行通信就是两台主机中的**应用进程互相通信**；
  - 从运输层的角度看，**通信的真正端点**并不是主机而是主机中的**进程**。也就是说，端到端的通信是应用进程之间的通信。

# 网络层和运输层有明显的区别

- 运输层
  - 运输层概述
  - 运输层的作用
  - 复用与分用
  - TCP与UDP

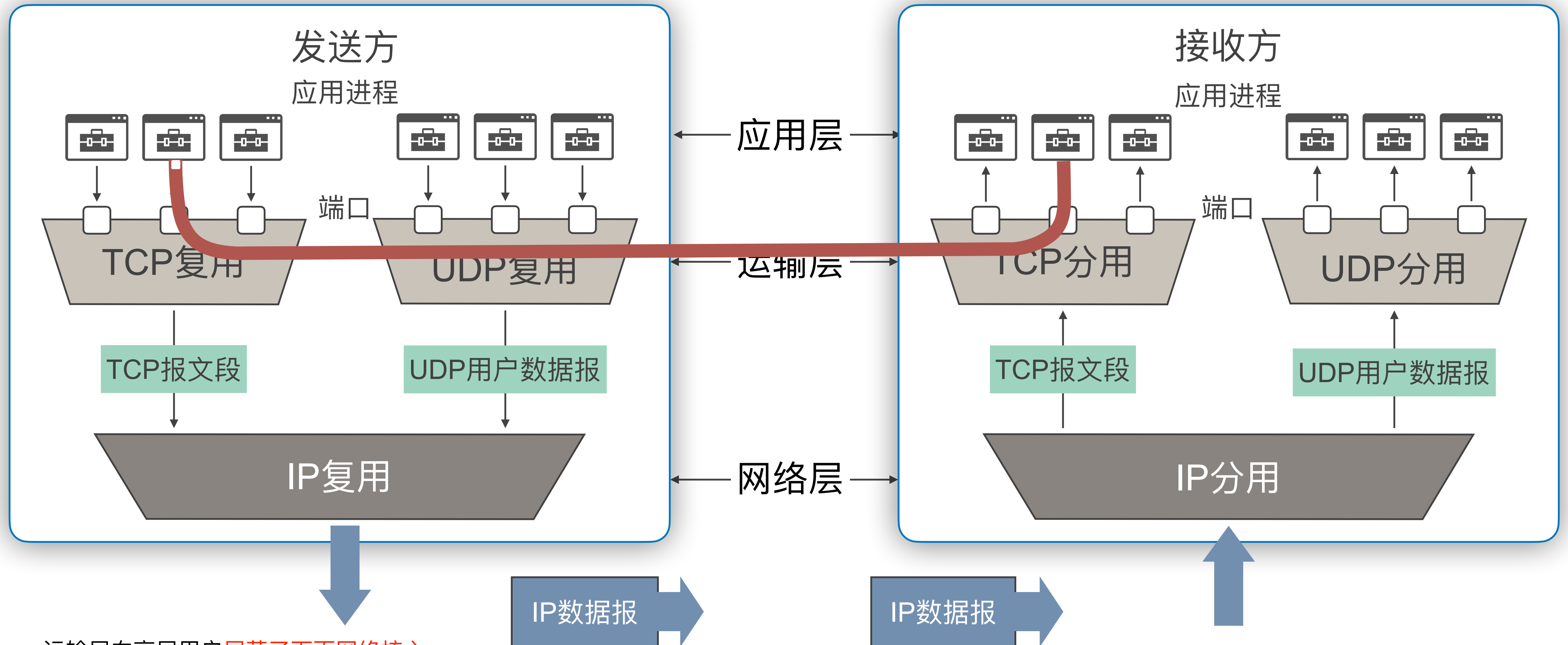


# 运输层复用和分用

- 运输层
  - 运输层概述
  - 运输层的作用
  - 复用与分用
  - TCP与UDP

- 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通信。
- 表明运输层有一个很重要的功能：
  - 复用 (multiplexing);
  - 分用 (demultiplexing)。
- 根据应用程序的不同需求，运输层需要有两种不同的运输协议：
  - 即面向连接的 TCP ；
  - 无连接的 UDP 。

# 运输层复用和分用



- 运输层向高层用户屏蔽了下面网络核心的细节，使应用进程感觉在两个运输层实体之间有一条端到端的逻辑通信信道。

基于端口的复用和分用

# TCP与UDP（逻辑信道的差异性）

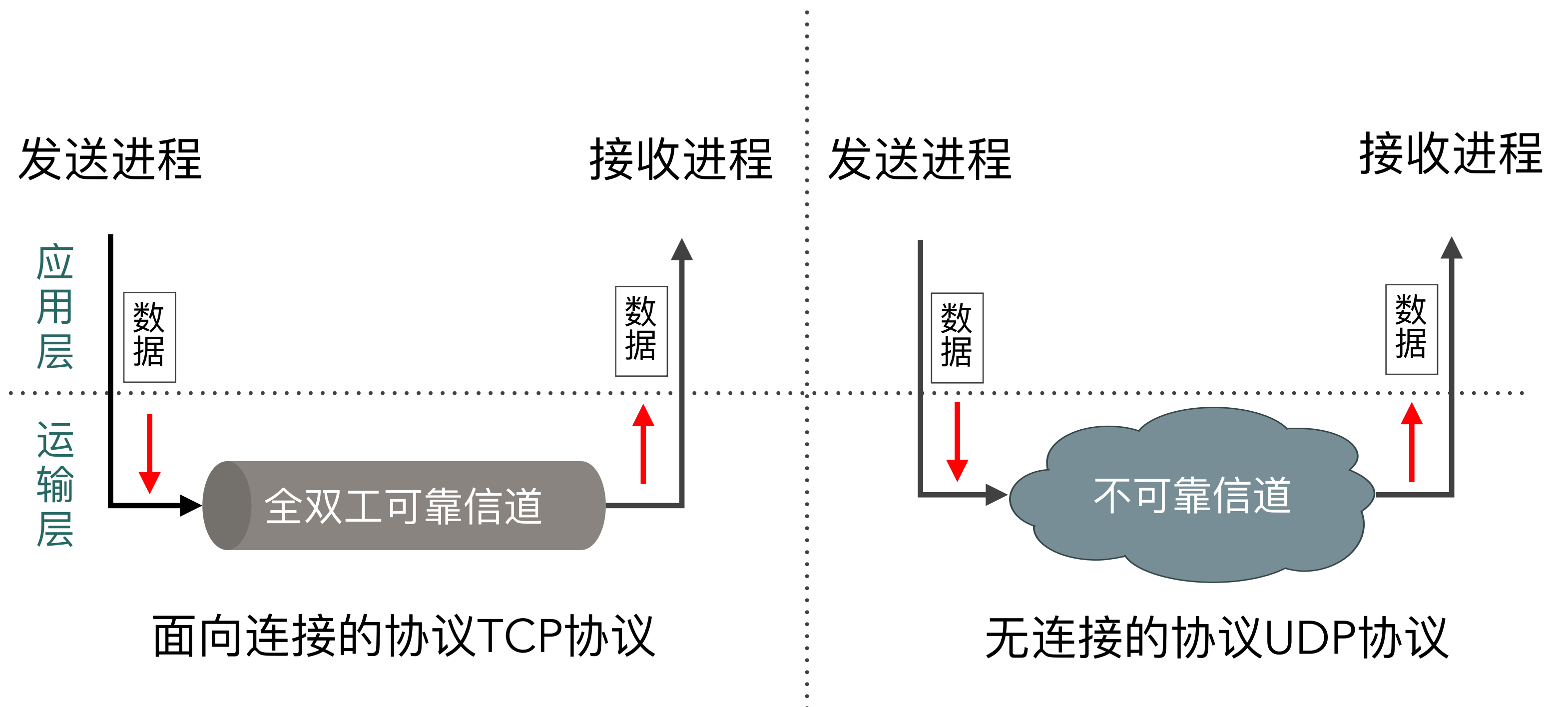
- 运输层
  - 运输层概述
  - 运输层的作用
  - 复用与分用
  - TCP与UDP

- 运输层的逻辑通信信道的差别：
  - 当运输层采用面向连接的 TCP 协议时，尽管下面的网络是不可靠的，但这种逻辑通信信道就相当于一**条全双工的可靠信道**；
  - 当运输层采用无连接的 UDP 协议时，这种逻辑通信信道是一条**不可靠信道**；
  - TCP靠谱，UDP不靠谱。



# 可靠信道与不可靠信道

- 运输层
  - 运输层概述
  - 运输层的作用
  - 复用与分用
  - TCP与UDP



# 小结

- 运输层
  - 运输层概述
  - 运输层的作用
  - 复用与分用
  - TCP与UDP

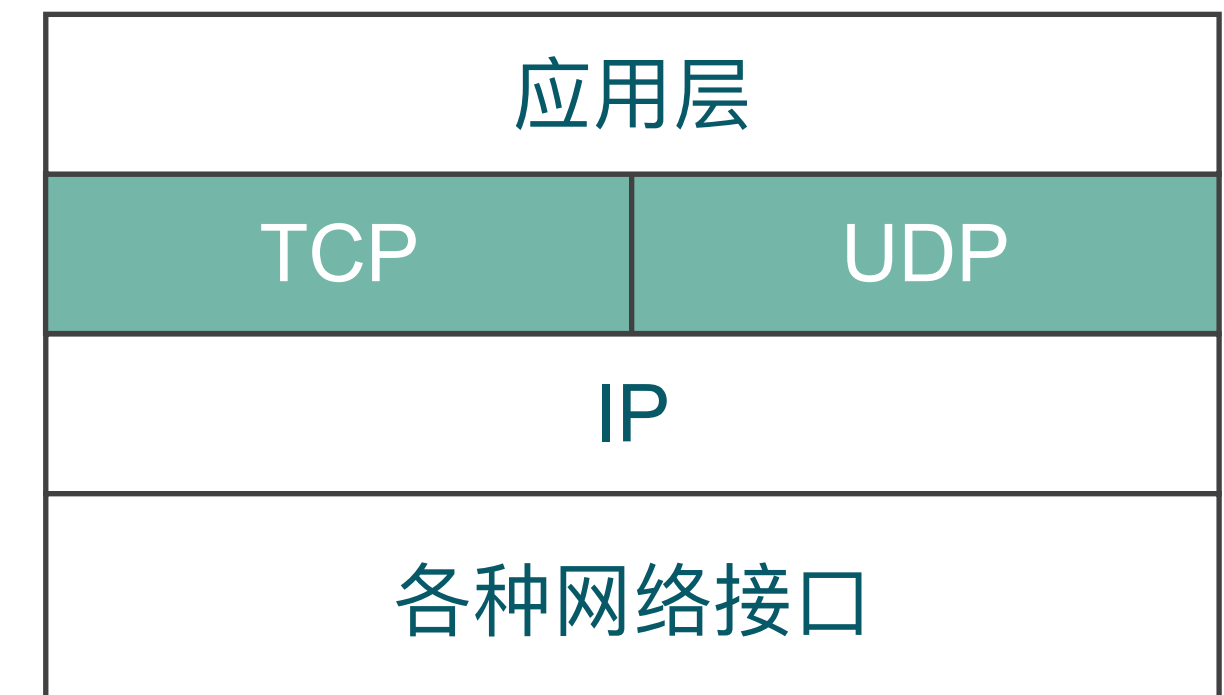
- 运输层协议概述。
- 为什么需要运输层。
- 异构计算机进程间的通信。
- TCP、UDP复用与解复用。
- 可靠信道与不可靠信道。

# 运输层上协议

- 运输层
  - 运输层协议
  - UDP和TCP特点
  - 典型应用
  - IP与TCP的区别
  - 运输层端口

- 用户数据报协议 UDP (User Datagram Protocol)。
- 传输控制协议 TCP (Transmission Control Protocol)。
- 两个对等运输实体传送的数据单元称为运输协议数据单元 TPDU (Transport Protocol Data Unit):
  - TCP 传送的数据单位是 TCP 报文段(segment);
  - UDP 传送的数据单位是 UDP 报文或用户数据报。

TCP/IP体系中的  
运输层协议





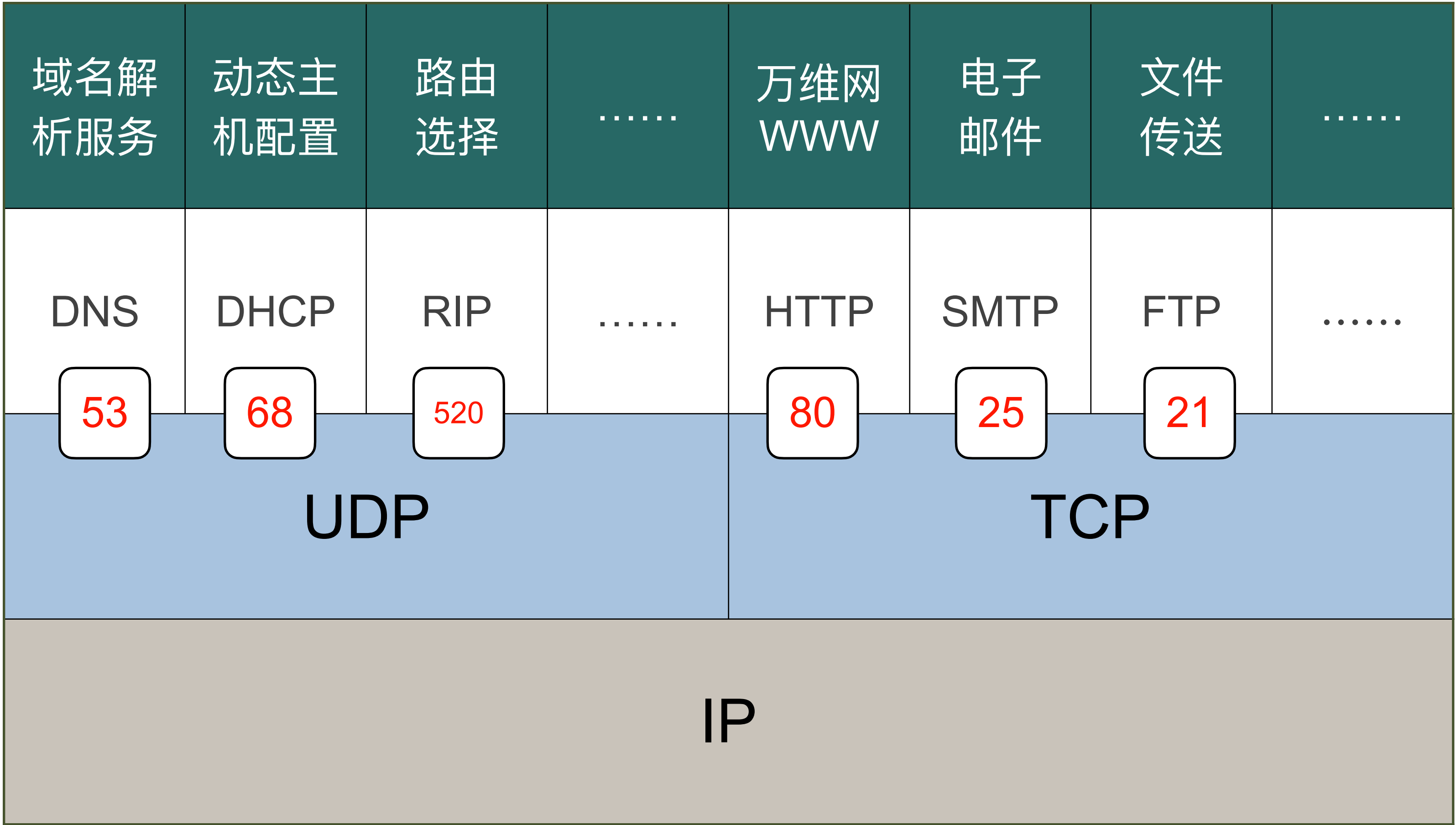
# UDP和TCP协议特点

- 运输层
  - 运输层协议
  - **UDP和TCP特点**
  - 典型应用
  - IP与TCP的区别
  - 运输层端口

- **UDP协议：**
  - 不需要先建立连接，提供无连接服务；
  - 协议数据单元为 UDP 报文或用户数据报；
  - 收到 UDP 报文后，无需任何确认；
  - UDP 不提供可靠交付；
  - 简单、支持单播、多播、广播；
  - 适用于多媒体应用。
- **TCP协议：**
  - 提供面向连接的服务；
  - 传送的数据单位协议是 TCP 报文段 (segment)；
  - TCP 不提供广播或多播服务；
  - 收到TCP报文段后，需要确认；
  - 协议复杂、开销大，占用较多的处理机资源；
  - 应用较多：万维网、电子邮件、文件传送等。

# UDP和TCP协议典型应用

- 运输层
  - 运输层协议
  - UDP和TCP特点
  - 典型应用
  - IP与TCP的区别
  - 运输层端口



## 注意两点

- 运输层
  - 运输层协议
  - UDP和TCP特点
  - 典型应用
  - IP与TCP的区别
  - 运输层端口

- 运输层的UDP用户数据报与网际层的IP数据报有**很大区别**：
  - **IP数据报**要经过互联网中许多路由器的**存储转发**；
  - **UDP用户数据报**是在运输层的端到端抽象的**逻辑信道中传送的**。
- TCP报文段是在运输层抽象的**端到端逻辑信道中传送的**：
  - 这种信道是可靠的**全双工信道**；
  - 这种信道不知道究竟经过了哪些路由器，而这些路由器也根本不知道上面的运输层是否建立了TCP连接。

# 为什么需要“运输层的端口”

- 运输层
  - 运输层协议
  - UDP和TCP特点
  - 典型应用
  - IP与TCP的区别
  - 运输层端口

- 为了使运行不同操作系统的计算机的应用进程能够互相通信，就必须用统一的方法对TCP/IP体系的应用进程进行标志：
  - 运行在计算机中的进程是用进程标识符来标志的；
  - 但应用层的各种应用进程却不应当让计算机操作系统指派它的进程标识符。这是因为在互联网上使用的计算机的操作系统种类很多，而不同的操作系统又使用不同格式的进程标识符。

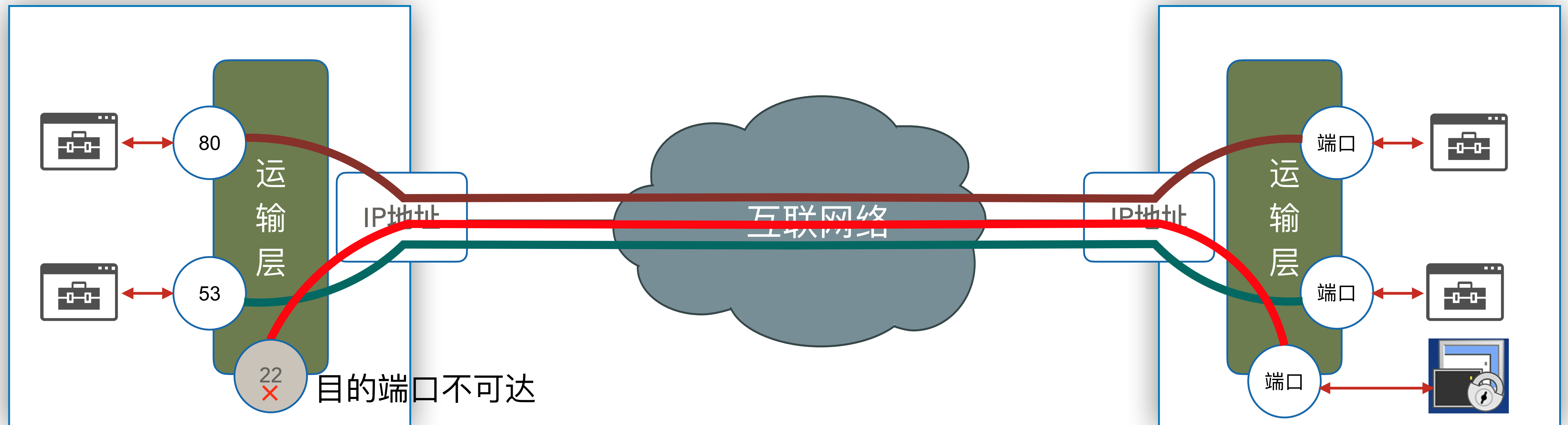
# 需要解决的问题

- 运输层
  - 运输层协议
  - UDP和TCP特点
  - 典型应用
  - IP与TCP的区别
  - 运输层端口

- 由于进程的创建和撤销都是动态的，发送方几乎无法识别其他机器上的进程。
- 有时我们会改换接收报文的进程，但并不需要通知所有发送方。
- 我们往往需要利用目的主机提供的功能来识别终点，而不需要知道实现这个功能的进程。
- 解决这个问题的方法就是在运输层使用协议端口号 (protocol port number)。
- 通信的终点是应用进程，但可以把端口想象是通信的终点，只要把要传送的报文交到目的主机的目的端口，剩下的工作（即最后交付目的进程）由 TCP 来完成。

# 端口

- 端口用一个 16 位端口号进行标志：
  - 端口号只具有本地意义，即端口号只是为了标志本计算机应用层中的各进程；
  - 两个计算机中的进程要互相通信，不仅必须知道对方的 IP 地址（为了找到对方的计算机），而且还要知道对方的端口号（为了找到对方计算机中的应用进程）。



# 两大类端口

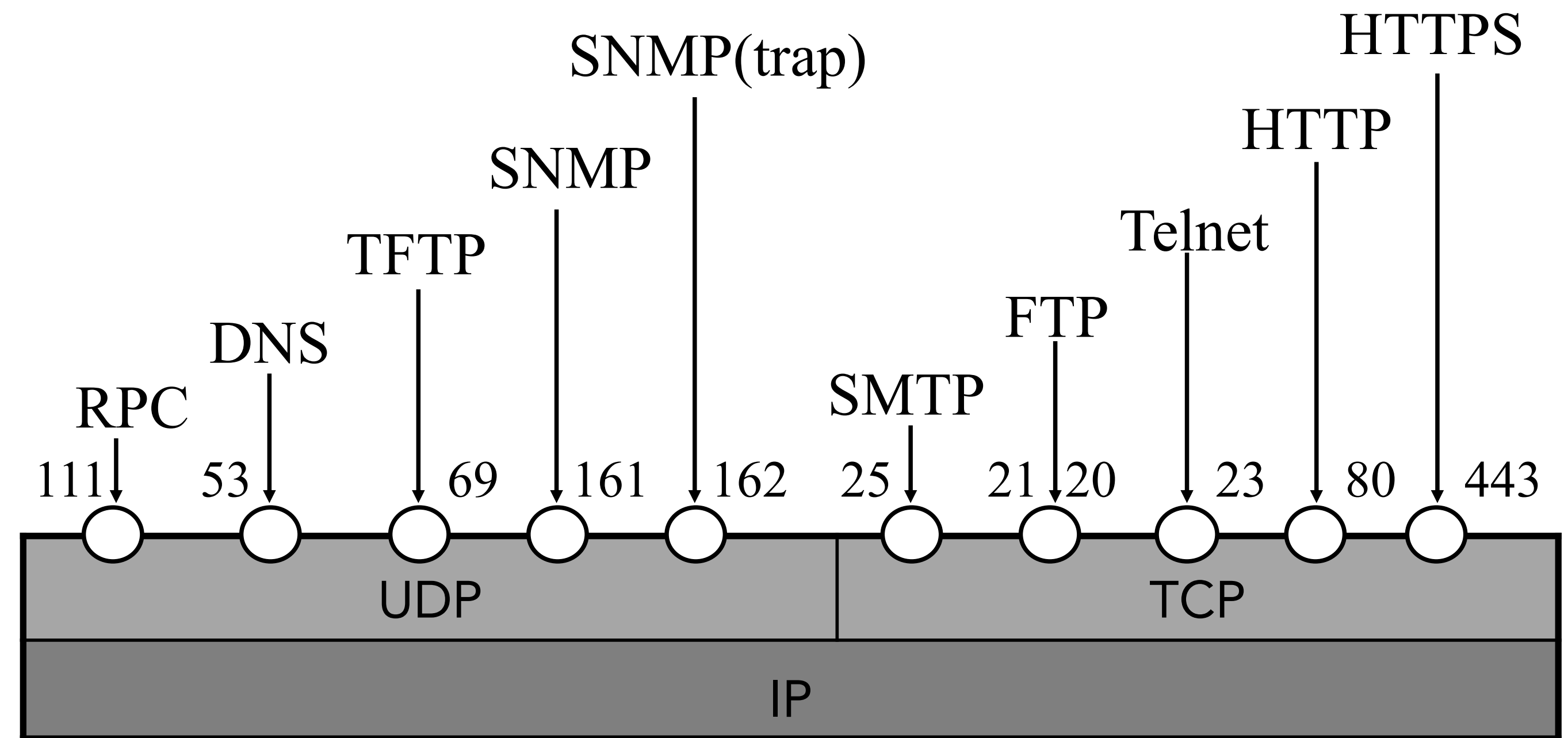
- 运输层
  - 运输层协议
  - UDP和TCP特点
  - 典型应用
  - IP与TCP的区别
  - 运输层端口

- 服务器端使用的端口号：
  - 熟知端口，数值一般为 0~1023（全世界都知道的）；
  - 登记端口号，数值为 1024~49151，给没有熟知端口号的应用程序使用的。使用这个范围的端口号必须在 IANA 登记，以防止重复。
- 客户端使用的端口号：
  - 又称为短暂端口号，数值为 49152~65535，留给客户进程暂时使用；
  - 当服务器进程收到客户进程的报文时，就知道了客户进程所使用的动态端口号。通信结束后，这个端口号可供其他客户进程以后使用。



# 常用熟知端口号

- 运输层
  - 运输层协议
  - UDP和TCP特点
  - 典型应用
  - IP与TCP的区别
  - 运输层端口



保存有常用的熟知端口号的文件:

- linux中的/etc/services
- Windows中的C:\Windows\System32\drivers\etc\services



# 小结

- 运输层
  - 运输层协议
  - UDP和TCP特点
  - 典型应用
  - IP与TCP的区别
  - 运输层端口

- 运输层两个协议TCP、UDP概述：
  - TCP传输数据单位；
  - UDP传输数据单位；
  - UDP用户数据报与IP数据报的区别；
  - 逻辑信道的概念。
  - UDP与TCP的区别。
  - TCP与UDP典型应用。
  - 端口的作用与分类：
    - 常用的熟知端口。

# 用户数据报协议 UDP

- 运输层
- UDP协议
- UDP主要特点
- 使用场景
- 报文格式
- 检验和计算
- UDP实例

- UDP 只在 IP 的数据报服务之上增加了很少一点的功能：
  - 复用和分用的功能；
  - 差错检测的功能。
- 虽然 UDP 用户数据报只能提供不可靠的交付，但 UDP 在某些方面有其特殊的优点。

# UDP 的主要特点

- 运输层
- UDP协议
- **UDP主要特点**
- 使用场景
- 报文格式
- 检验和计算
- UDP实例

- **不需要建立连接：**
  - 减少开销和发送数据之前的时延。
- **尽最大努力交付：**
  - 即不保证可靠交付。
- **面向报文的：**
  - 一次交付一个完整的报文。
- **没有拥塞控制：**
  - 网络出现拥塞不会降低源主机的发送速率。
- **支持多种交互通信：**
  - 支持一对一、一对多、多对一和多对多的交互通信。
- **首部开销小：**
  - 8 个字节，比 TCP 的 20 个字节的首部要短。

# 什么是“面向报文的 UDP”

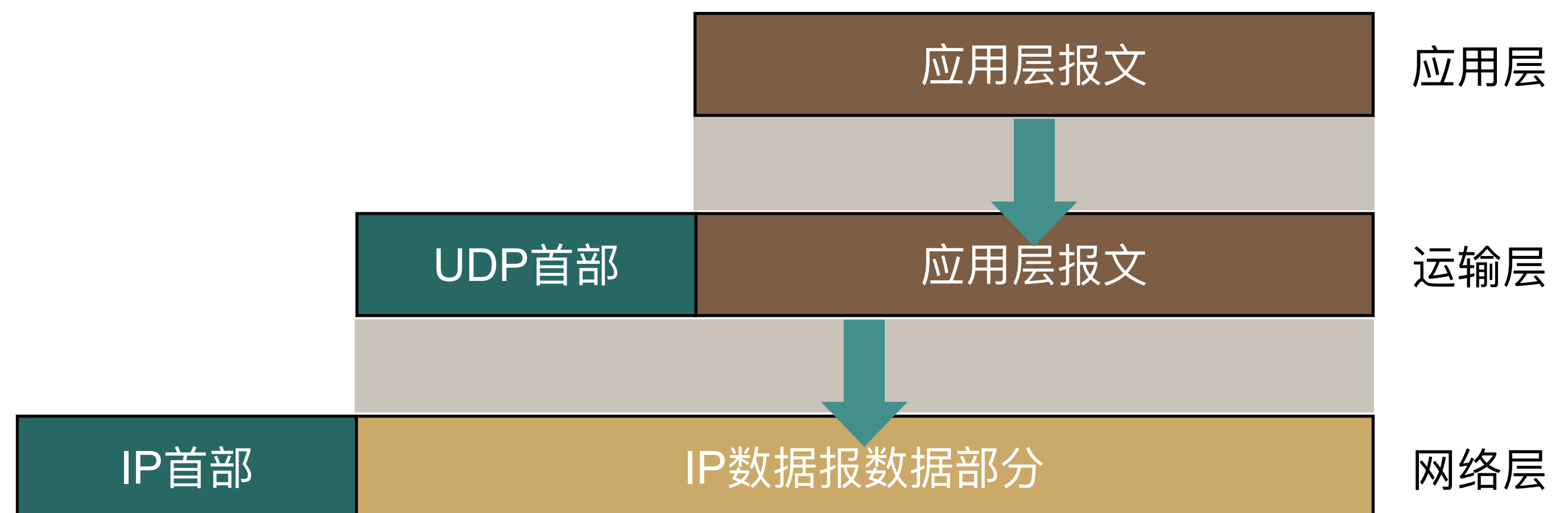
- 运输层
  - UDP协议
  - UDP主要特点
  - 使用场景
  - 报文格式
  - 检验和计算
  - UDP实例

- 发送方 UDP 对应用程序交下来的报文，添加首部后就向下交付 IP 层：
  - 应用层交给 UDP 多长的报文，UDP 就照样发送，即一次发送一个报文；
  - UDP 对应用层交下来的报文，既不合并，也不拆分，保留这些报文的边界。
- 接收方UDP去除UDP报文首部，原封不动地交付上层的应用进程，一次交付一个完整的报文。

# UDP 是面向报文的

- 运输层
  - UDP协议
  - **UDP主要特点**
  - 使用场景
  - 报文格式
  - 检验和计算
  - UDP实例

- 应用层产生数据交给UDP，UDP打包并直接送到网络层。
- 应用程序必须选择**合适大小的报文**：
  - **若报文太长**，UDP 把它交给 IP 层后，IP 层在传送时可能要进  
行分片，这会降低 IP 层的效率；
  - **若报文太短**，UDP 把它交给 IP 层后，会使 IP 数据报的首部的  
相对长度太大，这也降低了 IP 层的效率。



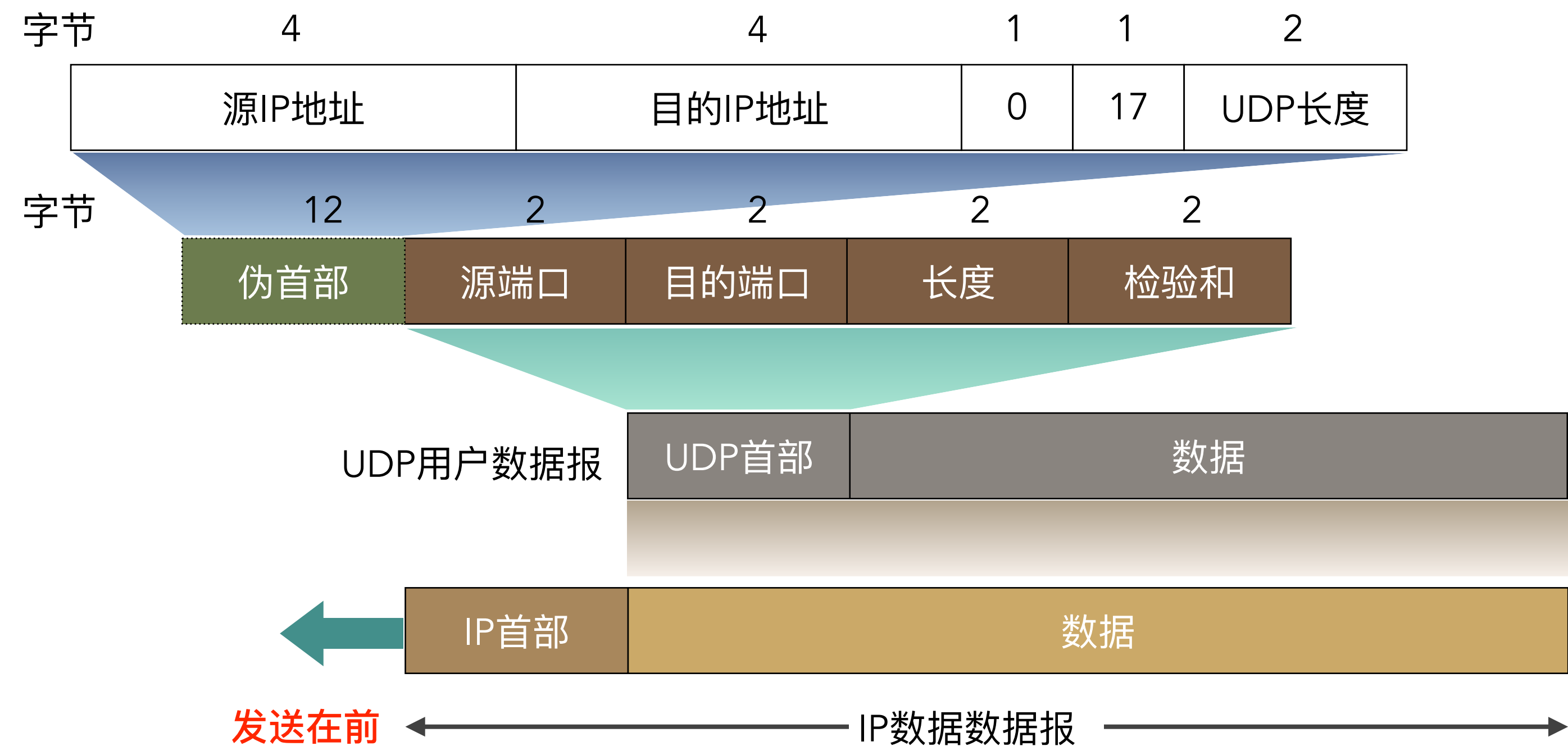
# UDP协议的使用场景

- 运输层
  - UDP协议
  - UDP主要特点
  - 使用场景
  - 报文格式
  - 检验和计算
  - UDP实例

- 可以重复请求信息的情况下：
  - 例如，DNS，DHCP。
  - 一次性传小量数据的应用（面向报文的）
  - 实时应用：
    - IP电话、视频会议等。
  - 多媒体应用。

# UDP 的首部格式（语法、语义）

- 运输层
  - UDP协议
  - UDP主要特点
  - 使用场景
  - 报文格式
  - 检验和计算
  - UDP实例



# 伪首部

- 运输层
  - UDP协议
  - UDP主要特点
  - 使用场景
  - 报文格式
  - 检验和计算
  - UDP实例

- 伪首部不是UDP真实首部，仅仅是为了计算检验和。
- 可以理解为UDP的两次检查：
  - 一次是对IP地址进行检验，确认该IP分组是发送给本机的；
  - 一次是对端口号和数据进行检验，确认交给哪个进程并且数据是无误的。

- 发送方：
  - 增加伪首部，UDP首部检验和填充0；
  - 数据部分填充0至4字节整数倍；
  - 计算机检验和，伪首部+首部+数据；
  - 首部填上检验和；
  - 删除伪首部，发送UDP数据报。

- 接收方：
  - 增加伪首部；
  - 计算机检验和，伪首部+首部+数据；
  - 检验和全1无差错，否则丢弃或上交应用进程（附上错误警告）。



# 计算UDP校验和的例子

伪首部	153.19.8.104			
	171.3.14.11			
	0	17	15	
UDP首部	1087		13	
	15		0	
数据	data	data	data	data
	data	data	data	0

二进制反码运算求和  
将得出的结果求反码

10011001	00010011	153.19: 39187	
00001000	01101000	8.104: 2152	
10101011	00000011	171.3: 43779	
00001110	00001011	14.11: 3595	
00000000	00010001	0和17: 17	
00000000	00001111	15: 15	
00000100	00111111	1087: 1087	
00000000	00001101	13: 13	
00000000	00001111	15: 15	
00000000	00000000	0 (检验和) : 0	
01010100	01000101	data: 21573	
01010011	01010100	data: 21332	
01001001	01001110	data: 18766	
01000111	00000000	data和0 (填充) : 18176	
10	10010110	11101011	1001011011101101
			取反
检验和			0110100100010010

# 抓取UDP用户数据报（DNS查询）

- 运输层
  - UDP协议
  - UDP主要特点
  - 使用场景
  - 报文格式
  - 检验和计算
  - **UDP实例**



# 小结

- 运输层
  - UDP协议
  - UDP主要特点
  - 使用场景
  - 报文格式
  - 检验和计算
  - UDP实例

- UDP的功能：
  - 复用和分用；
  - 差错检测功能。
- UDP的主要特点。
- UDP首部格式。
- UDP检验和计算：
  - 伪首部。

# 传输控制协议 TCP 概述

- 运输层
- TCP协议
- 面向连接的概念
- 面向字节流的概念
- TCP连接的概念

- TCP 是面向连接的：
  - TCP 连接只能有两个端点，TCP 连接是点对点的（一对一）；
  - TCP 提供可靠交付的服务；
  - TCP 提供全双工通信。
- 面向字节流：
  - TCP 中的“流”(stream)指的是流入或流出进程的字节序列；
  - 面向字节流的含义是：虽然应用程序和 TCP 的交互是一次一个数据块，但 TCP 把应用程序交下来的数据看成仅仅是一连串无结构的字节流。

# 几个面向连接的概念

- 运输层
- TCP协议
- 面向连接的概念
- 面向字节流的概念
- TCP连接的概念

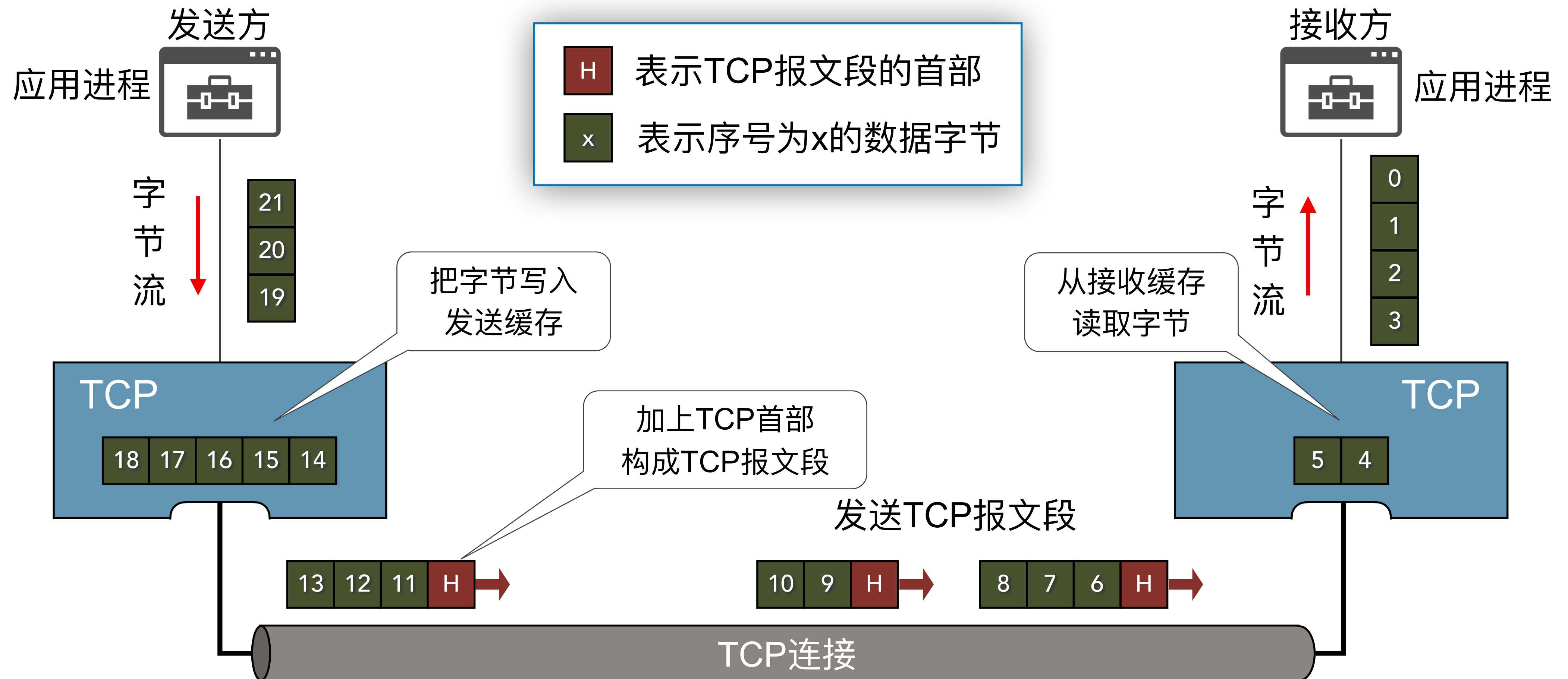
- 面向连接的电路交换（物理层保证可靠）：
  - 通信双方之间必须有一条物理连接的通路（直接相连），且被通信双方独享，数据按序发送并按序接收。
- 面向连接的虚电路（网络层保证可靠）：
  - 通信双方采用复用技术，逐段占用物理通路，每段物理通路可被多对通信使用，分组按序发送并按序接收。
- 面向连接TCP（运输层协议保证可靠）：
  - 采用协议的方法（确认、序号、重传），确保通信双方有一条全双工的、可靠的逻辑信道（事实上，提供服务的IP数据报是不可靠的），字节按序发送并按序接收（但网络层IP数据报并不一定按序到达）。

# 几个面向连接的概念

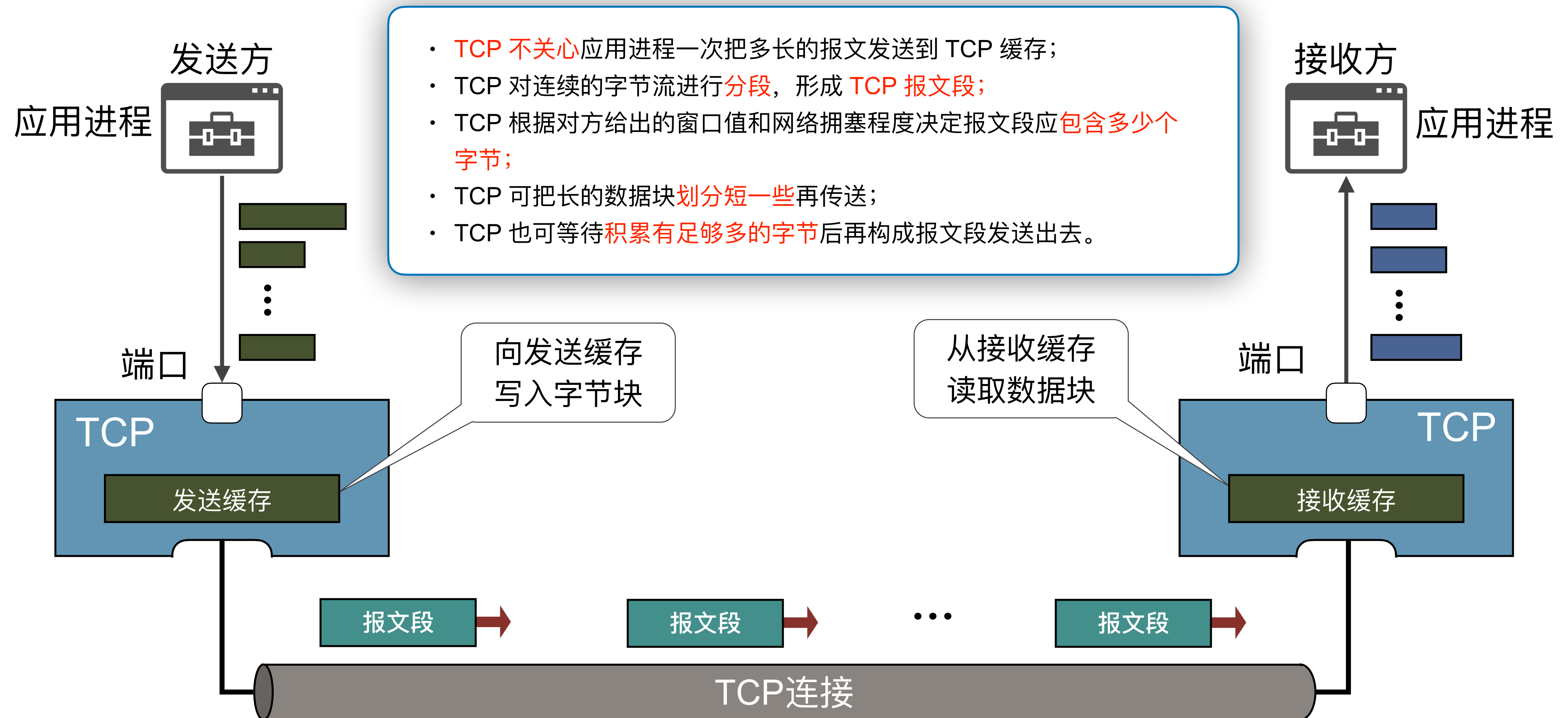
- 运输层
  - TCP协议
  - 面向连接的概念
  - 面向字节流的概念
  - TCP连接的概念

- TCP 不保证接收方应用程序所收到的数据块和发送方应用程序所发出的数据块具有对应大小的关系。
- 接收方应用程序收到的字节流必须和发送方应用程序发出的字节流完全一样。

# TCP 面向流的概念



# TCP 面向流的概念





# TCP 的连接

- 运输层
- TCP协议
- 面向连接的概念
- 面向字节流的概念
- TCP连接的概念

- TCP 把连接作为最基本的抽象：
  - 每一条 TCP 连接有两个端点；
  - TCP 连接的端点不是主机，不是主机的IP 地址，不是应用进程，也不是运输层的协议端口；
  - TCP 连接的端点叫做套接字 (socket) 或插口；
  - 端口号拼接到 (concatenated with) IP 地址即构成了套接字。

套接字 socket = (IP地址 : 端口号)  
TCP 连接 ::= {socket1, socket2}  
= {(IP1: port1), (IP2: port2)}

- 同一个 IP 地址可以有多个不同的 TCP 连接；
- 同一个端口号也可以出现在多个不同的 TCP 连接中。

物流的连接 ::= {(寄件人通信地址 : 寄件人), (收件人通信地址: 收件人)}

# 小结

- 运输层
  - TCP协议
  - 面向连接的概念
  - 面向字节流的概念
  - TCP连接的概念

- TCP的主要特点：
  - TCP是面向连接的；
  - TCP提供可靠交付；
  - TCP提供全双式通信；
  - TCP是面向字节流的。
- TCP的连接：
  - 套接字的概念。

# 理想的传输条件特点

- 运输层
- 可靠传输原理
- 停止等待协议
  - 无差错情况
  - 有差错情况
- 自动重传ARQ
- 利用率
- 流水线传输
- 连续ARQ协议

- 理想的传输条件有以下两个特点：
  - 传输信道不产生差错；
  - 不管发送方以多快的速度发送数据，接收方总是来得及处理收到的数据。
- 在这样的理想传输条件下，不需要采取任何措施就能够实现可靠传输。
- 实际的网络都不具备以上两个理想条件。
- 必须使用一些可靠传输协议，在不可靠的传输信道实现可靠传输。

# 停止等待协议

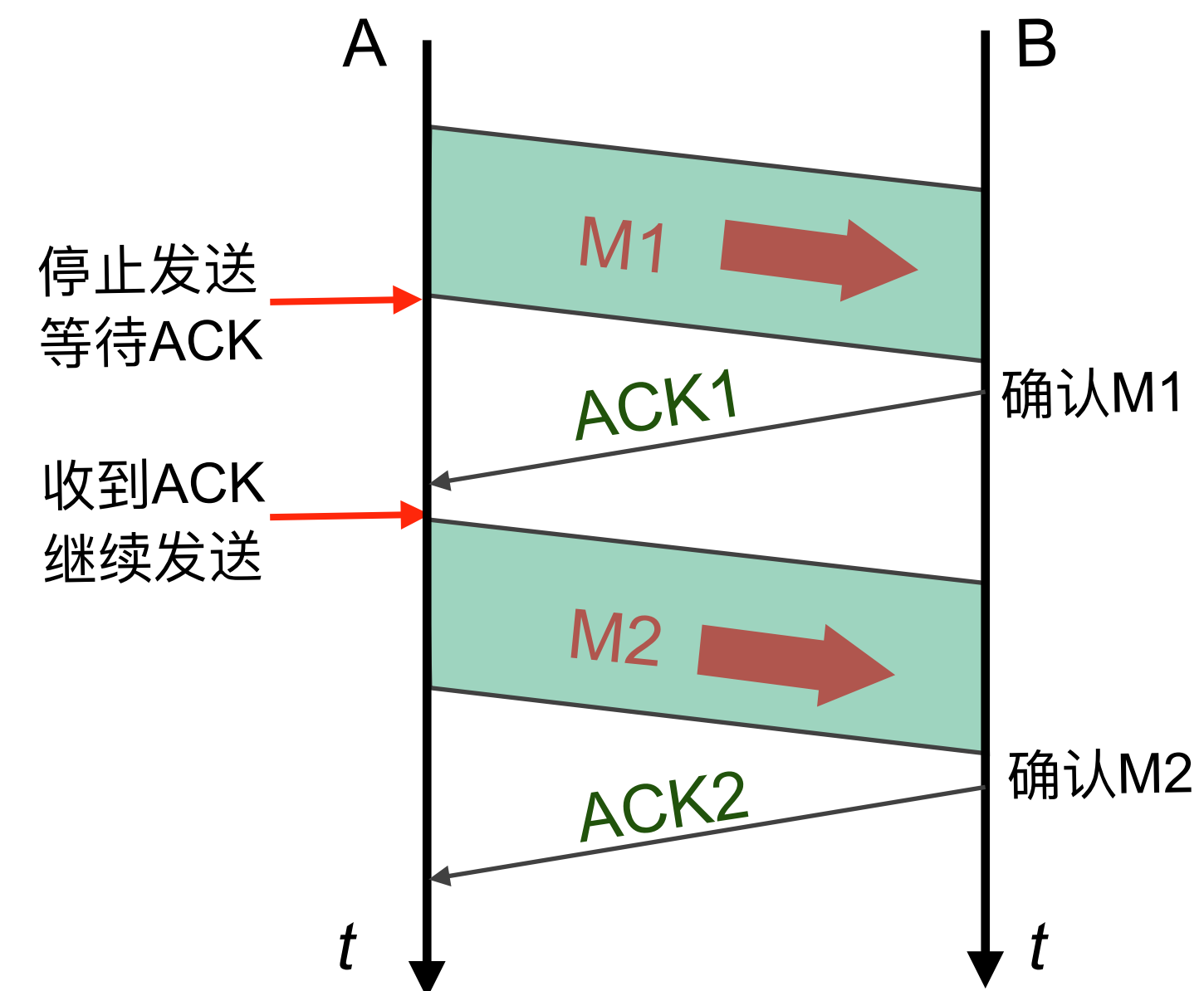
- 运输层
- 可靠传输原理
- 停止等待协议
  - 无差错情况
  - 有差错情况
  - 自动重传ARQ
  - 利用率
  - 流水线传输
  - 连续ARQ协议

- “停止等待”就是每发送完一个分组就停止发送，等待对方的确认。在收到确认后再发送下一个分组（TCP报文段）。
- 全双工通信的双方既是发送方也是接收方。
- 为了讨论问题的方便：
  - 仅考虑 A 发送数据， B 接收数据并发送确认；
  - A 称为发送方，而 B 称为接收方。

# 无差错

- 运输层
- 可靠传输原理
- 停止等待协议
  - 无差错情况
  - 有差错情况
- 自动重传ARQ
- 利用率
- 流水线传输
- 连续ARQ协议

- A 发送完分组 M1，就暂停发送；
- 等待 B 的确认 (ACK)；
- B 收到了 M1，向 A 发送 ACK；
- A 在收到了对 M1 的确认后，继续发送下一个分组 M2。

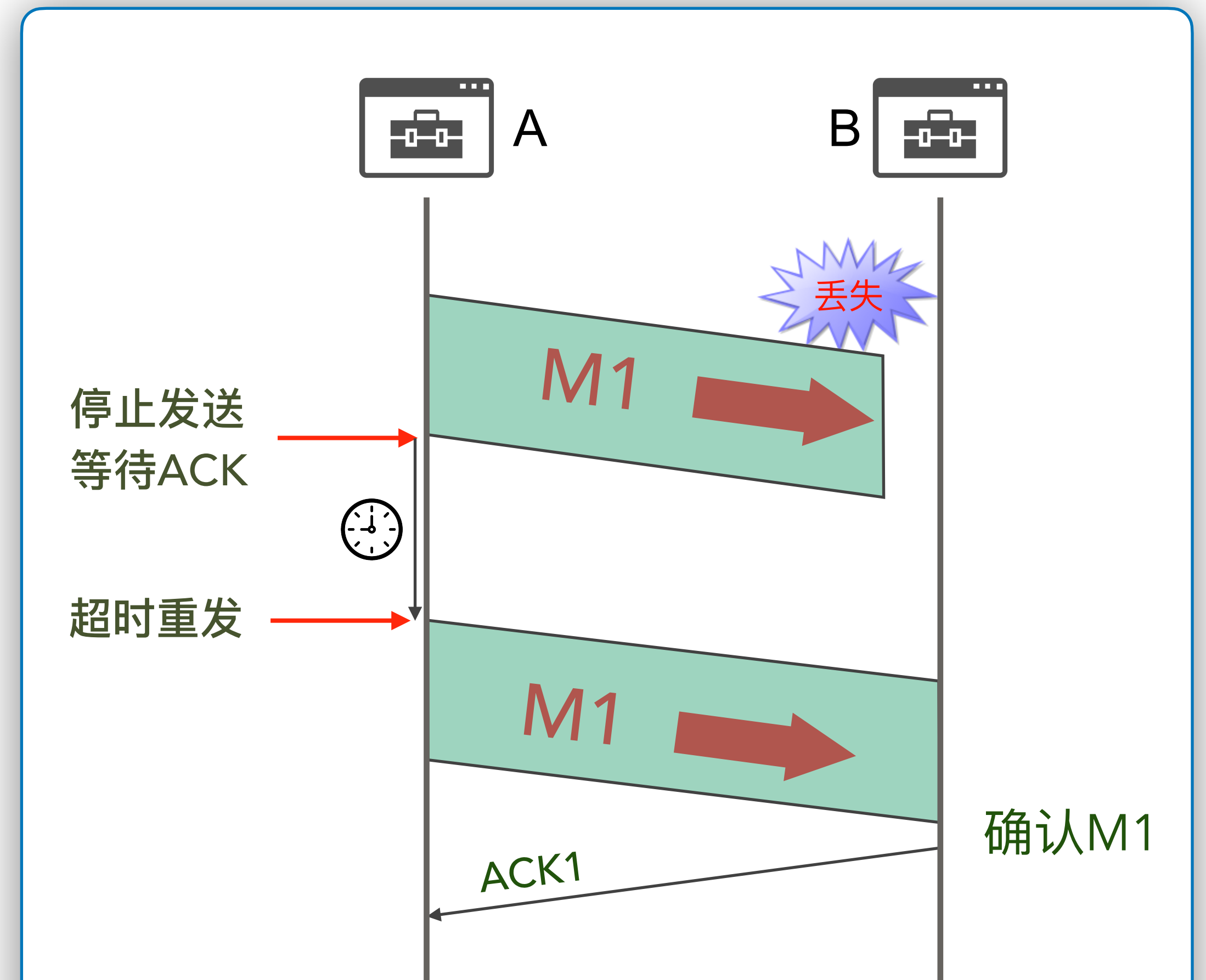
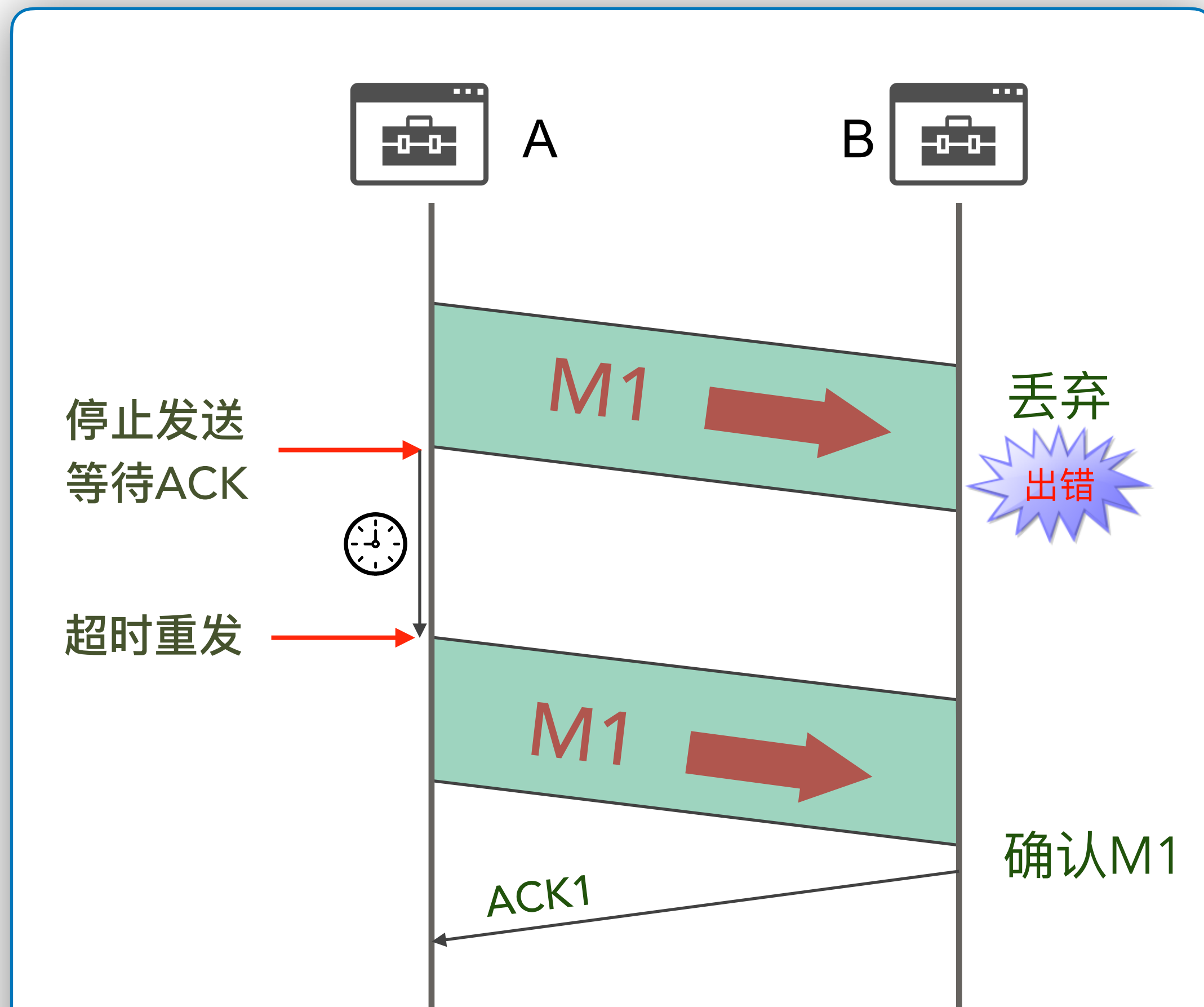


# 有差错情况

- 运输层
  - 可靠传输原理
  - 停止等待协议
    - 无差错情况
    - 有差错情况
  - 自动重传ARQ
  - 利用率
  - 流水线传输
  - 连续ARQ协议

- 在接收方 B 会出现两种情况：
  - B 接收 M1 时检测出了差错，就丢弃 M1，其他什么也不做；
  - M1 在传输过程中丢失了，这时 B 当然什么都不知道，也什么都不做。
- 如何保证 B 正确收到了 M1 呢？
  - 解决方法：超时重传；
  - A 为每一个已发送的分组都设置超时计时器；
  - A 在超时计时器到期之前收到了相应的确认，撤销该超时计时器，继续发送下一个分组 M2 。

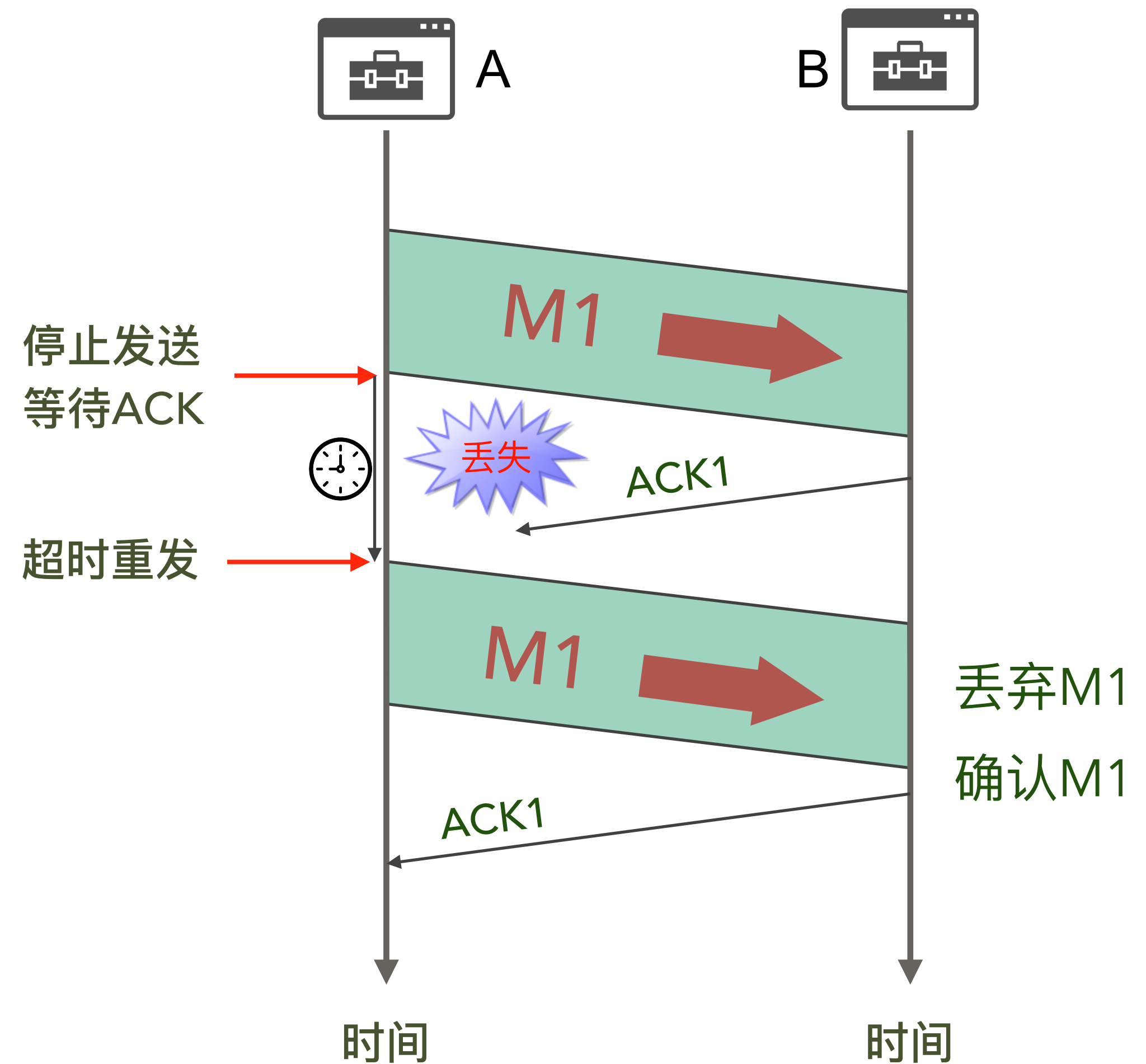
## 出现差错（发送的报文出错）



超时重传时间  $\geq RTT$ ；发送方需要缓存数据；必须对数据编号。

## 出现差错（确认丢失）

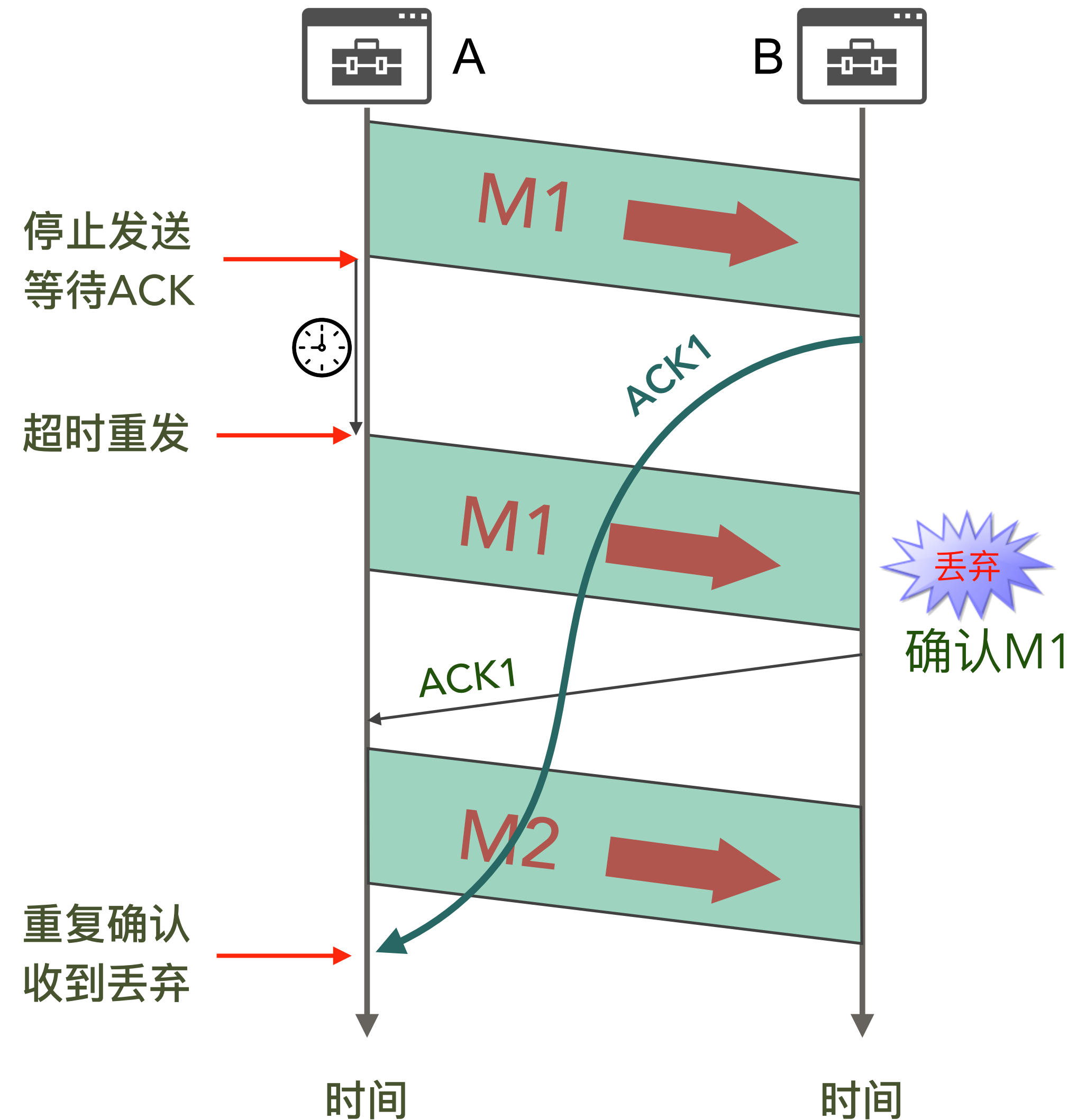
- A 在超时计时器到期后重传 M1。
- 如果 B 又收到了重传的分组 M1：
- 丢弃这个重复的分组 M1；
- 向 A 发送确认。





## 出现差错（确认迟到）

- B 对分组 M1 的**确认迟到了**。
- B 丢弃重复的 M1，并重传确认分组。
- A 会收到重复的确认：收下后就丢弃。

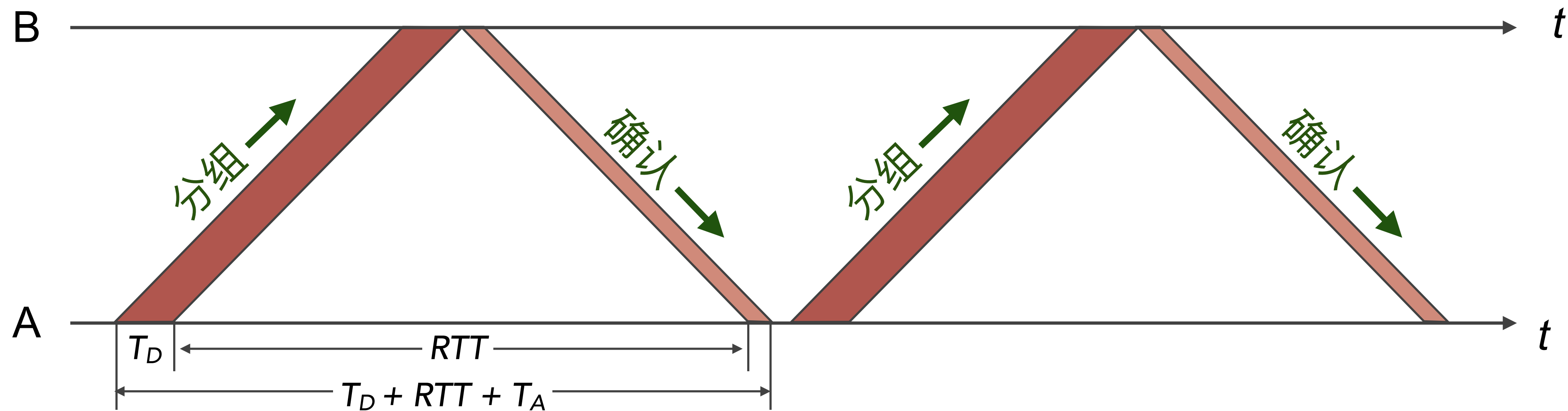


# 自动重传请求 ARQ

- 运输层
  - 可靠传输原理
  - 停止等待协议
    - 无差错情况
    - 有差错情况
    - 自动重传ARQ
  - 利用率
  - 流水线传输
  - 连续ARQ协议

- 通常A 最终总是可以收到对所有发出的分组的确认；
- 如果 A 不断重传分组但总是收不到确认，就说明通信线路太差，不能进行通信；
- 使用上述的确认和重传机制，我们就可以在不可靠的传输网络上实现可靠的通信；
- 传输协议常称为自动重传请求 ARQ (Automatic Repeat reQuest)。意思是重传的请求是自动进行的，接收方不需要请求发送方重传某个出错的分组。

# 信道利用率



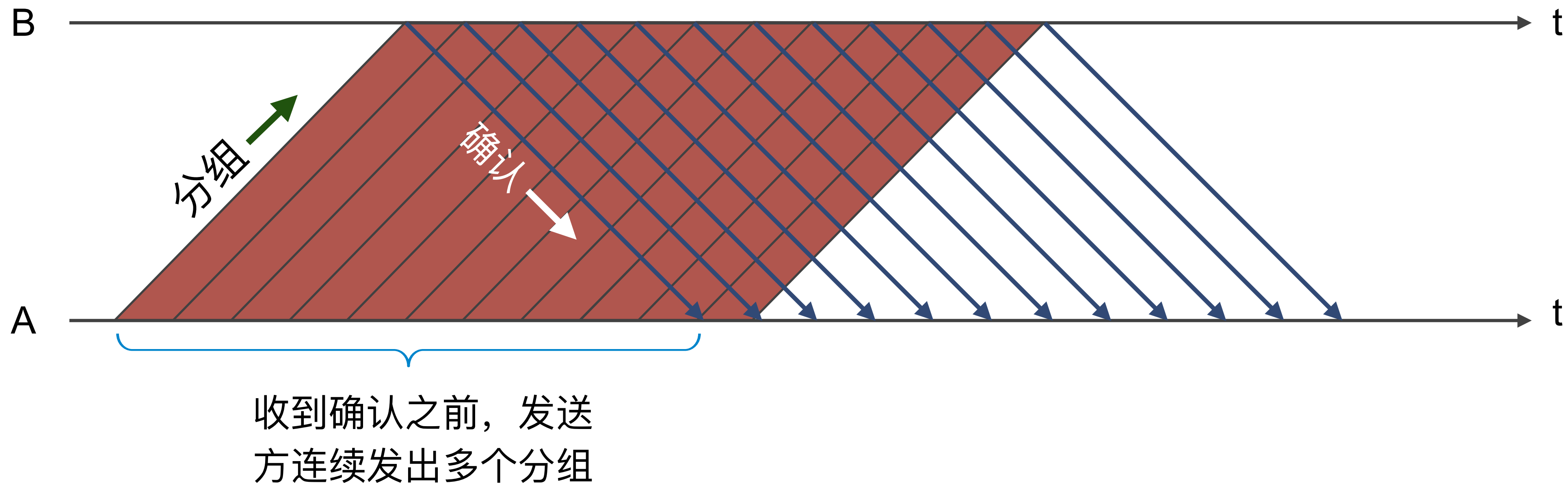
信道利用率: 
$$U = \frac{T_D}{T_D + RTT + T_A}$$

停止等待协议的优点是简单，缺点是信道利用率太低

# 流水线传输

- 流水线传输就是发送方可连续发送多个分组，不必每发完一个分组就停顿下来等待对方的确认。这样可使信道上一直有数据不间断地传送。

- 由于信道上一直有数据不间断地传送，信道的利用率高。

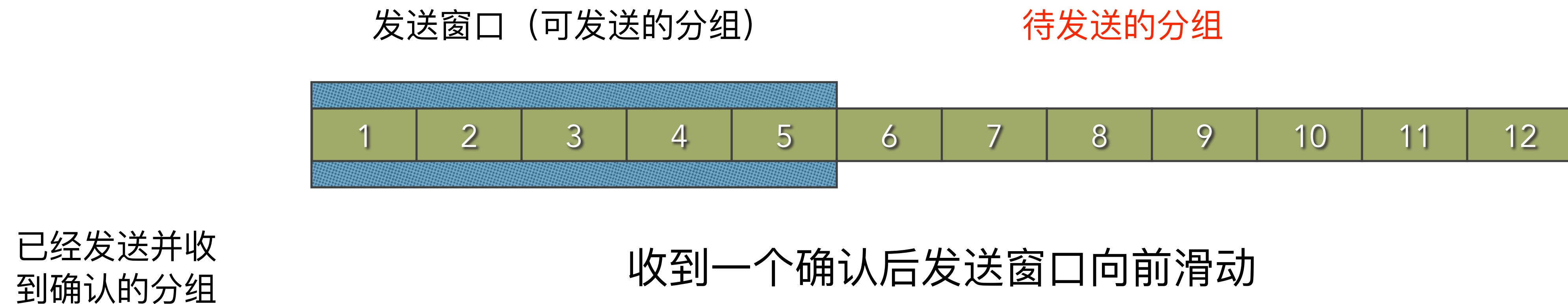


# 连续 ARQ协议

- 运输层
- 可靠传输原理
- 停止等待协议
  - 无差错情况
  - 有差错情况
- 自动重传ARQ
- 利用率
- 流水线传输
- 连续ARQ协议

- 基本思想：
  - 发送窗口内的分组都可连续发送出去，无需等待对方的确认。  
即发送方一次可以发出多个分组；
  - 使用滑动窗口协议控制发送方和接收方所能发送和接收的分组  
的数量和编号；
  - 每收到一个确认，发送方就把发送窗口向前滑动；
  - 接收方一般采用累积确认的方式；
  - 采用回退N（Go-Back-N）方法进行重传。

# 连接 ARQ协议原理



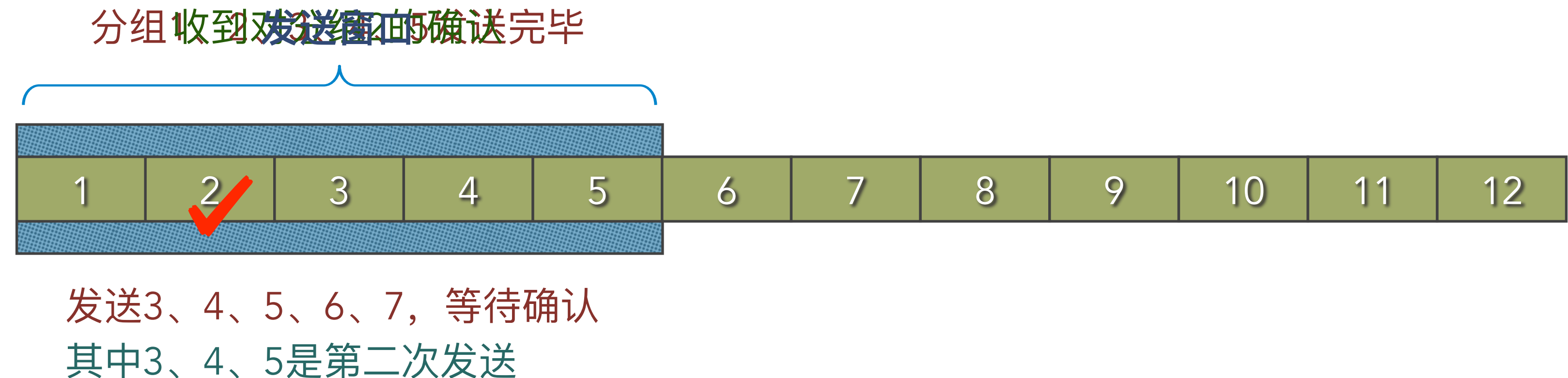
- 接收方采用**累积确认**：
  - 对按序到达的最后一个分组发送确认；
  - **优点**：容易实现，即使确认丢失也不必重传；
  - **缺点**：不能向发送方反映出接收方已经正确收到的所有分组的信息。



# Go-back-N (回退 N)

- 发送方发送了前 5 个分组，第 3 个分组丢失，接收方对收到 1、2 分组发出确认，发送方无法知道 3、4、5 的下落，只好把这三个分组再重传一次。

- Go-back-N (回退 N)：表示需要再退回来重传已发送过的 N 个分组。当通信线路质量不好时，连续 ARQ 协议会带来负面的影响。



# 小结：连续 ARQ 协议与停止等待协议对比

- 运输层
  - 可靠传输原理
  - 停止等待协议
    - 无差错情况
    - 有差错情况
  - 自动重传ARQ
  - 利用率
  - 流水线传输
  - 连续ARQ协议

对比内容	连续ARP协议	停止等待协议
发送的分组数量	一次发送多个分组	一次发送一个分组
传输控制	滑动窗口协议	停止、等待
确认	单独确认、累积确认	单独确认
重传计时器	每个发送的分组	每个发送的分组
序号	每个发送的分组	每个发送的分组
重传	回退N，多个分组	一个分组



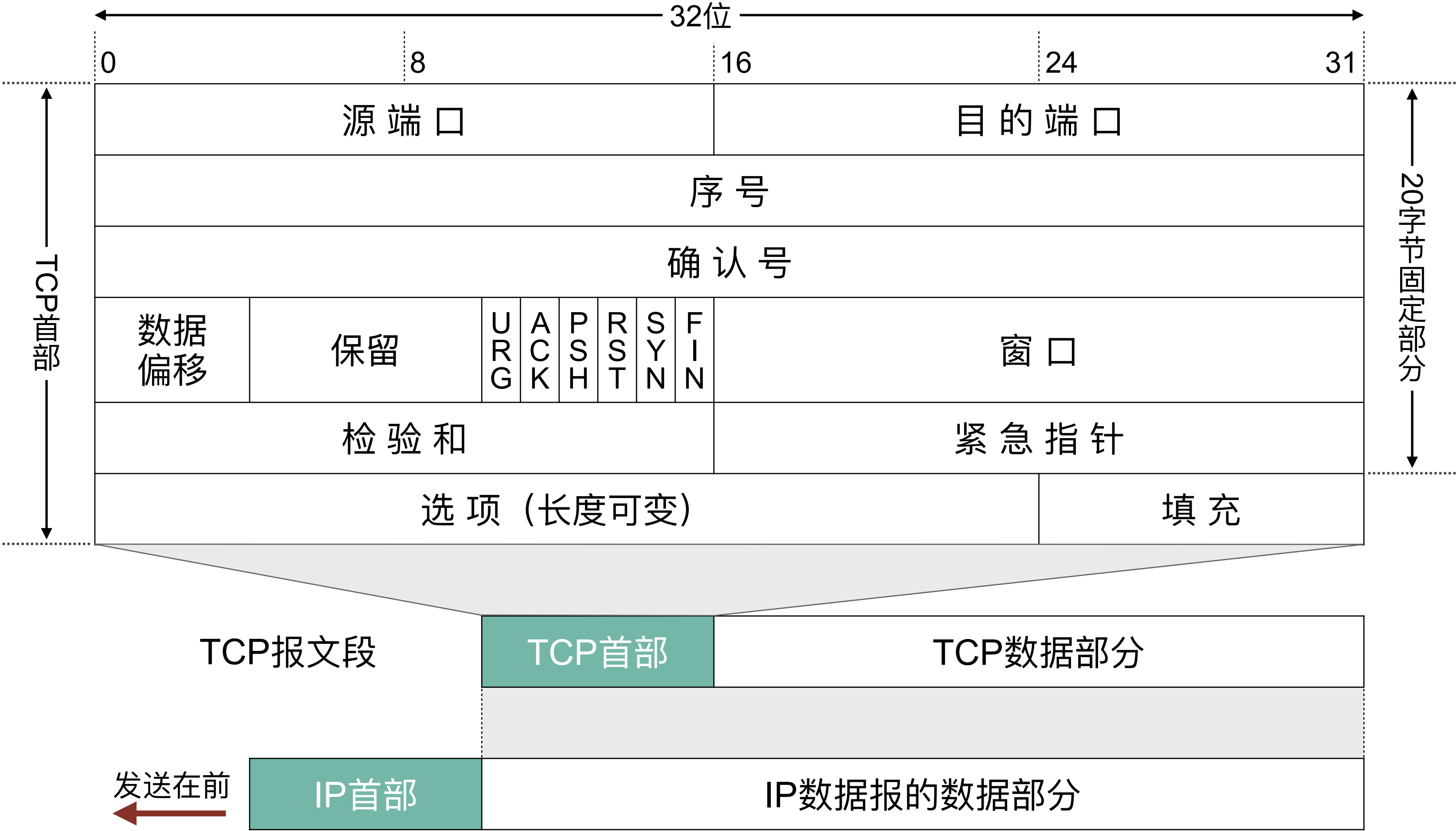
# TCP 报文段的首部格式

- 运输层
- TCP报文格式
- 源端口目的端口
- 序号、确认号
- 数据偏移
- 六个控制位
- 窗口、检验和
- 紧急指针
- 选项、填充

- TCP 报文段首部的**前20个字节是固定的**；
- 后面有  $4n$  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 字节**；
- TCP 虽然是面向字节流的，但 TCP 传送的**数据单元是报文段**；
- TCP 报文段分为首部和数据两部分，TCP 的全部功能体现在首部中各字段的作用。

# TCP报文段首部格式（语法、语义）

- 运输层
- TCP报文格式
- 源端口目的端口
- 序号、确认号
- 数据偏移
- 六个控制位
- 窗口、检验和
- 紧急指针
- 选项、填充



# TCP报文段首部格式（源端口、目的端口）

- 运输层
  - TCP报文格式
  - 源端口目的端口
  - 序号、确认号
  - 数据偏移
  - 六个控制位
  - 窗口、检验和
  - 紧急指针
  - 选项、填充



源端口和目的端口：各占 2 字，是运输层与应用层的服务接口。

# TCP报文段首部格式（序号、确认号）

- 运输层
  - TCP报文格式
  - 源端口目的端口
  - 序号、确认号
  - 数据偏移
  - 六个控制位
  - 窗口、检验和
  - 紧急指针
  - 选项、填充

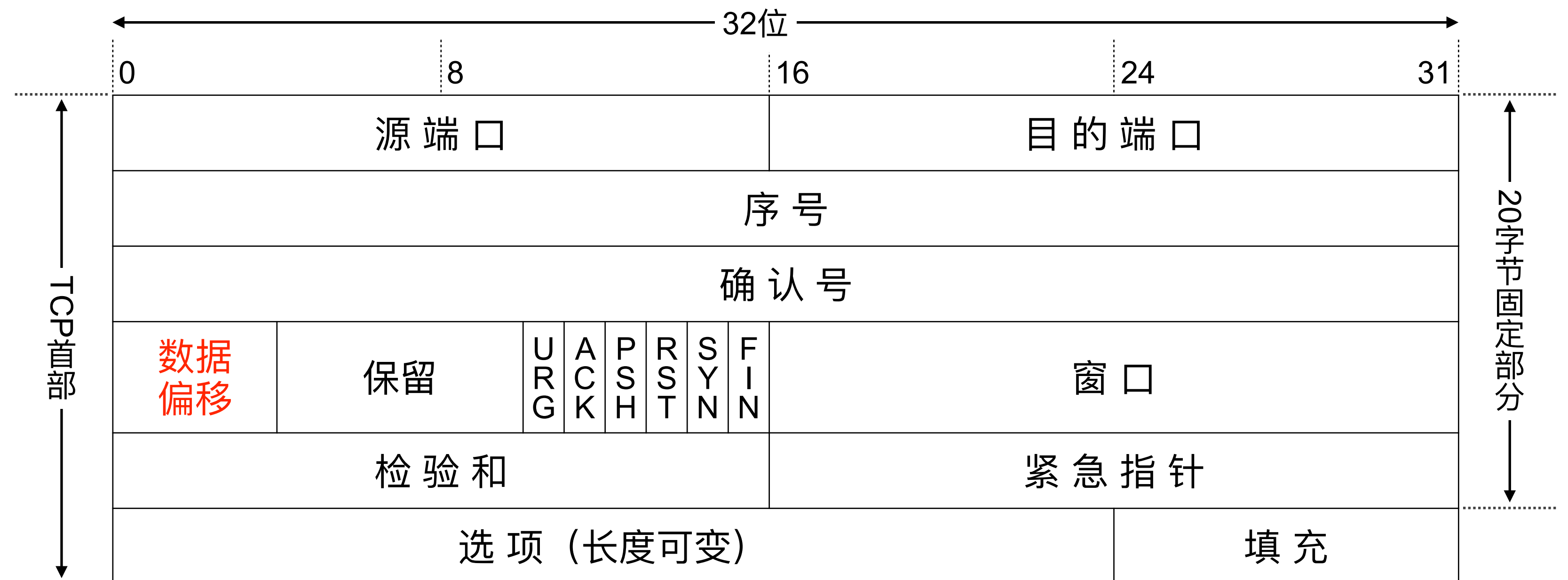


**序号：** 4 字节，指是本报文段所携带数据的第一个字节的序号。

**确认号：** 4 字节，是期望收到对方的下一个报文段中数据的第一个字节的序号。

# TCP报文段首部格式（数据偏移）

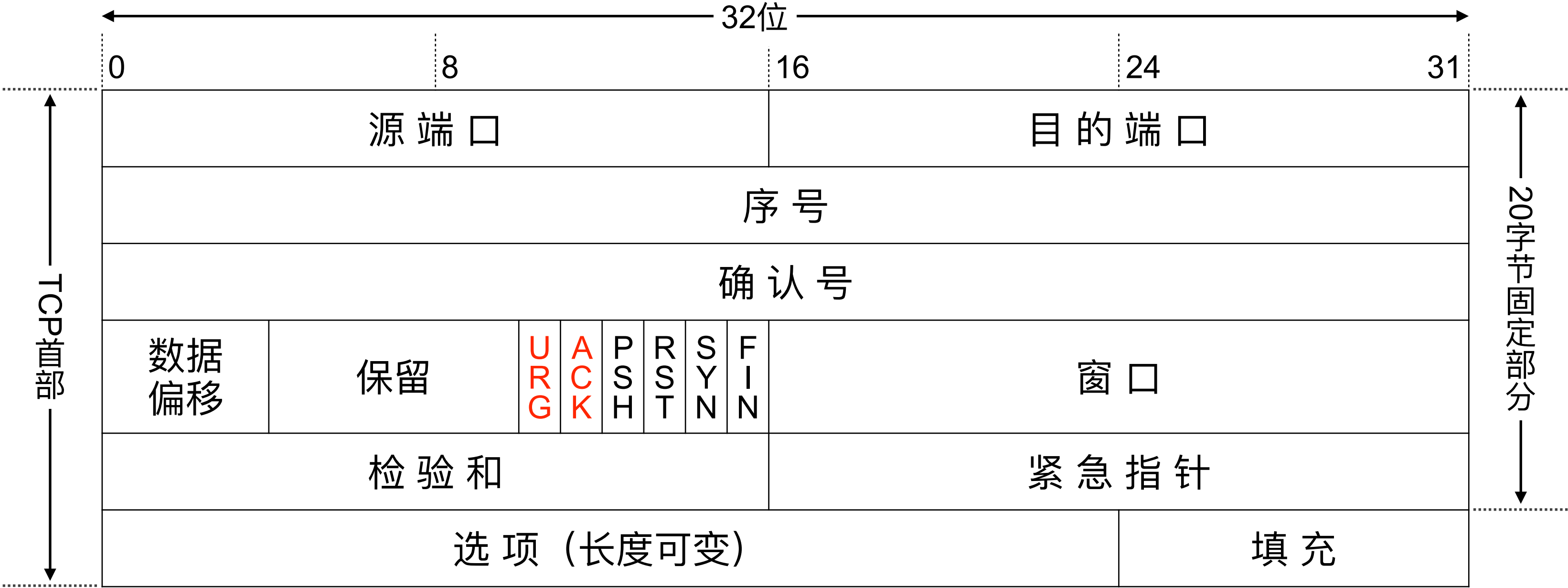
- 运输层
- TCP报文格式
- 源端口目的端口
- 序号、确认号
- 数据偏移
- 六个控制位
- 窗口、检验和
- 紧急指针
- 选项、填充



**数据偏移**（即首部长度的）：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。“数据偏移”的单位是 **32 位** 字（以 4 字节为计算单位），**最大60字节**。

# TCP报文段首部格式（紧急、确认）

- 运输层
- TCP报文格式
- 源端口目的端口
- 序号、确认号
- 数据偏移
- 六个控制位
- 窗口、检验和
- 紧急指针
- 选项、填充

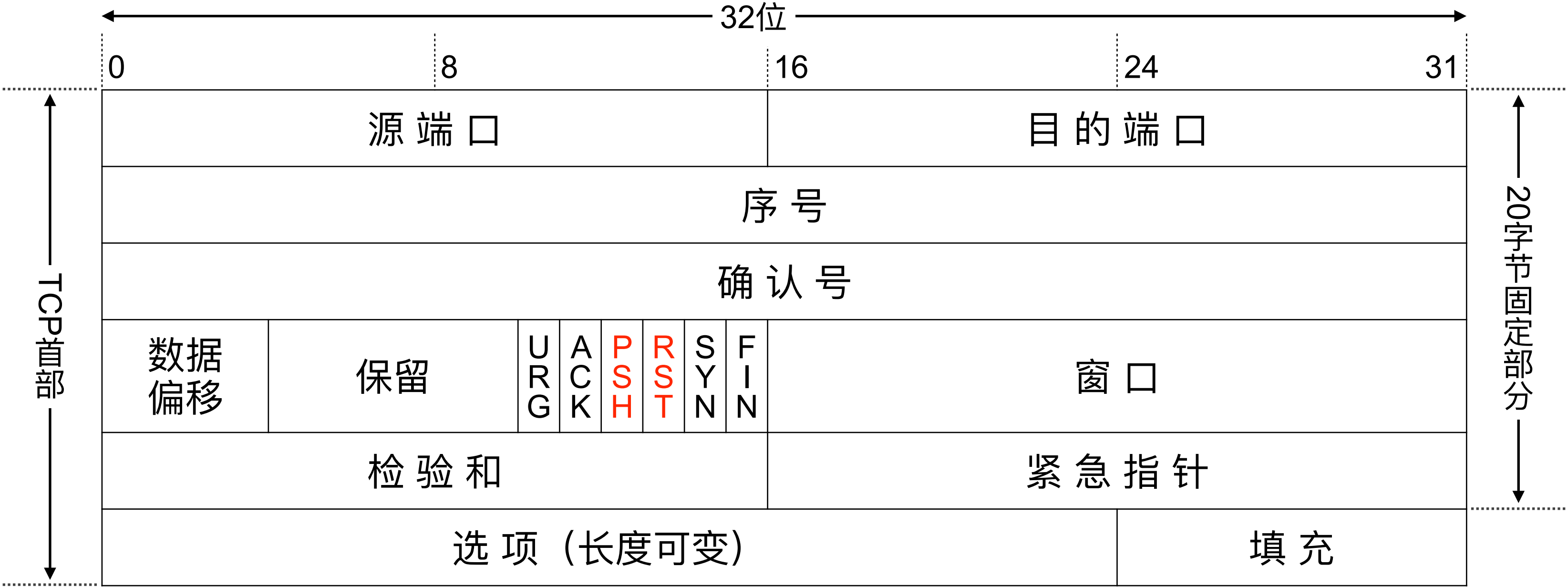


**紧急 URG:** 当 URG = 1 时，紧急指针字段有效，此报文段中有紧急数据，应尽快传送。

**确认 ACK:** 只有当 ACK = 1 时确认号字段才有效。当 ACK = 0 时，确认号无效。

# TCP报文段首部格式（推送、复位）

- 运输层
  - TCP报文格式
  - 源端口目的端口
  - 序号、确认号
  - 数据偏移
  - 六个控制位
  - 窗口、检验和
  - 紧急指针
  - 选项、填充



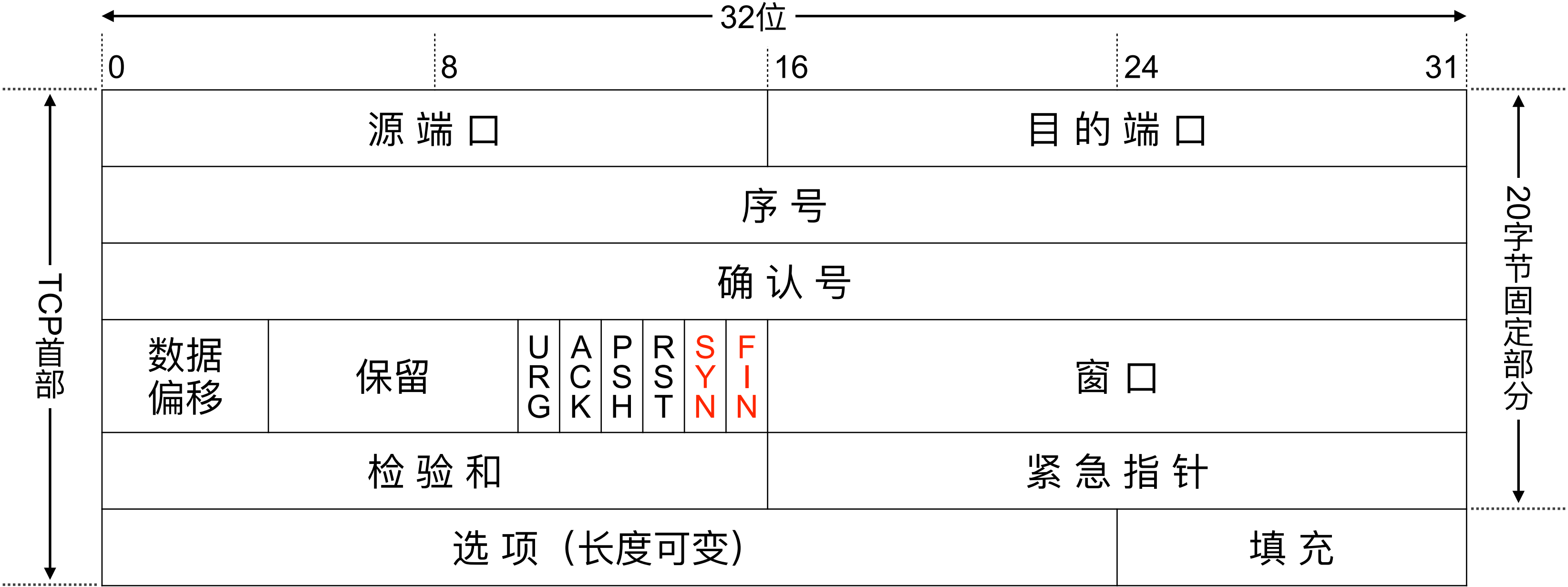
**推送 PSH:** 收到 PSH = 1 的报文段，发送方立即发送，接收方尽快上交接收应用进程。

**复位 RST:** 当 RST = 1 时，表明 TCP 连接中出现严重差错，必须释放连接，然后再重新建立运输连接。



# TCP报文段首部格式（同步、终止）

- 运输层
- TCP报文格式
- 源端口目的端口
- 序号、确认号
- 数据偏移
- 六个控制位
- 窗口、检验和
- 紧急指针
- 选项、填充



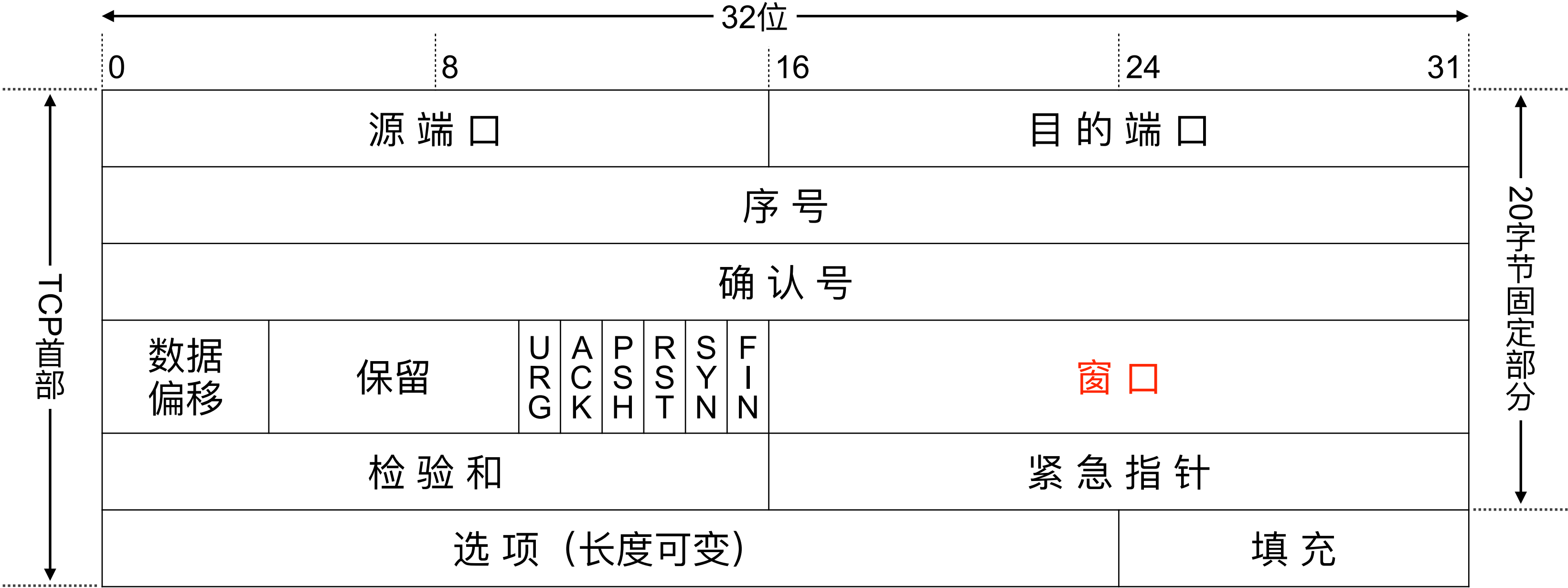
**同步 SYN:** 同步SYN = 1表示这是一个连接请求或连接接受报文。

**终止 FIN:** 用来释放一个连接。FIN = 1 表明此报文段的发送端的数据已发送完毕，并要求释放运输连接。



# TCP报文段首部格式（窗口）

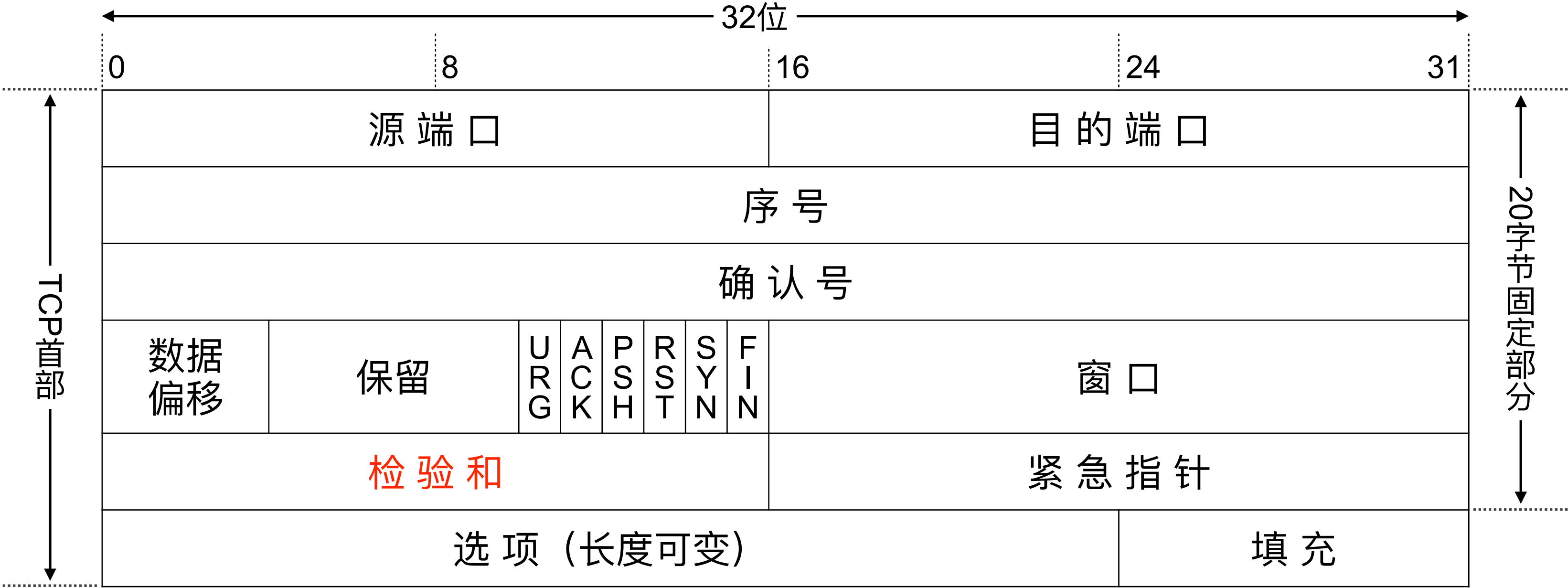
- 运输层
  - TCP报文格式
  - 源端口目的端口
  - 序号、确认号
  - 数据偏移
  - 六个控制位
  - 窗口、检验和
  - 紧急指针
  - 选项、填充



窗口字段：占 2 字节，用来让对方设置发送窗口的依据，单位为字节。

# TCP报文段首部格式（检验和）

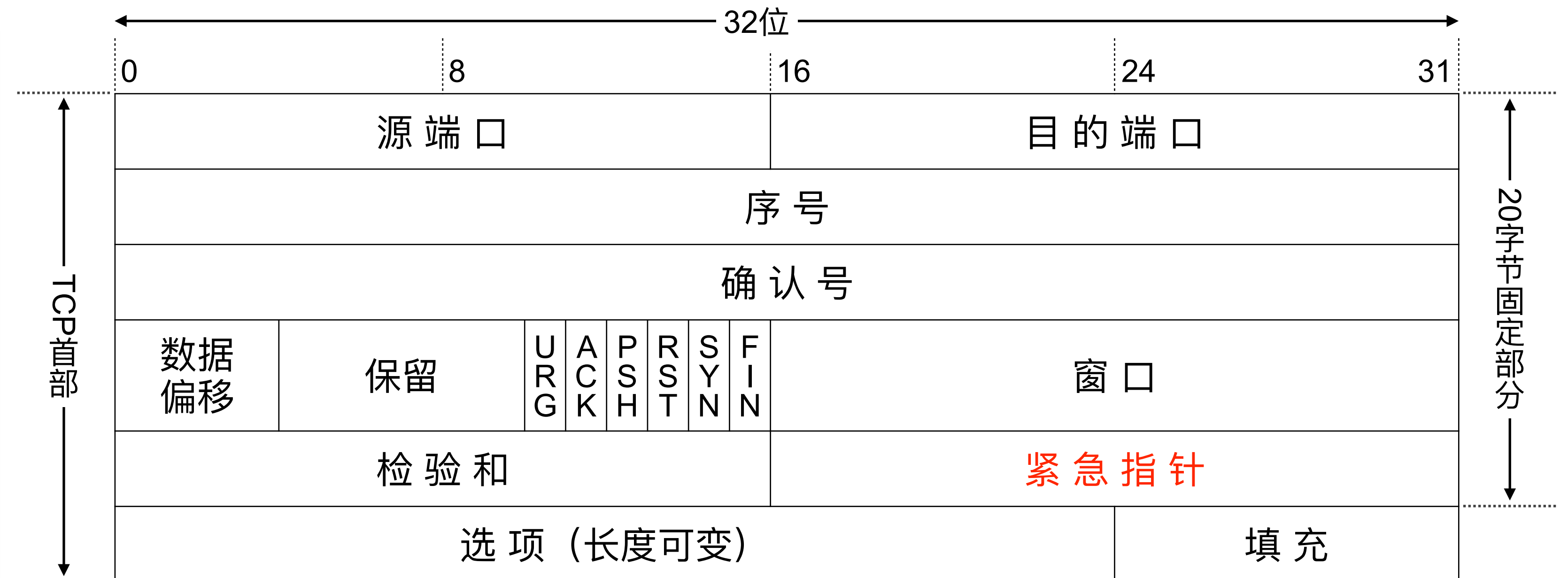
- 运输层
  - TCP报文格式
  - 源端口目的端口
  - 序号、确认号
  - 数据偏移
  - 六个控制位
  - 窗口、检验和
  - 紧急指针
  - 选项、填充



**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。在计算检验和时，要在 TCP 报文段的前面加上 12 字节的伪首部。

# TCP报文段首部格式（紧急指针）

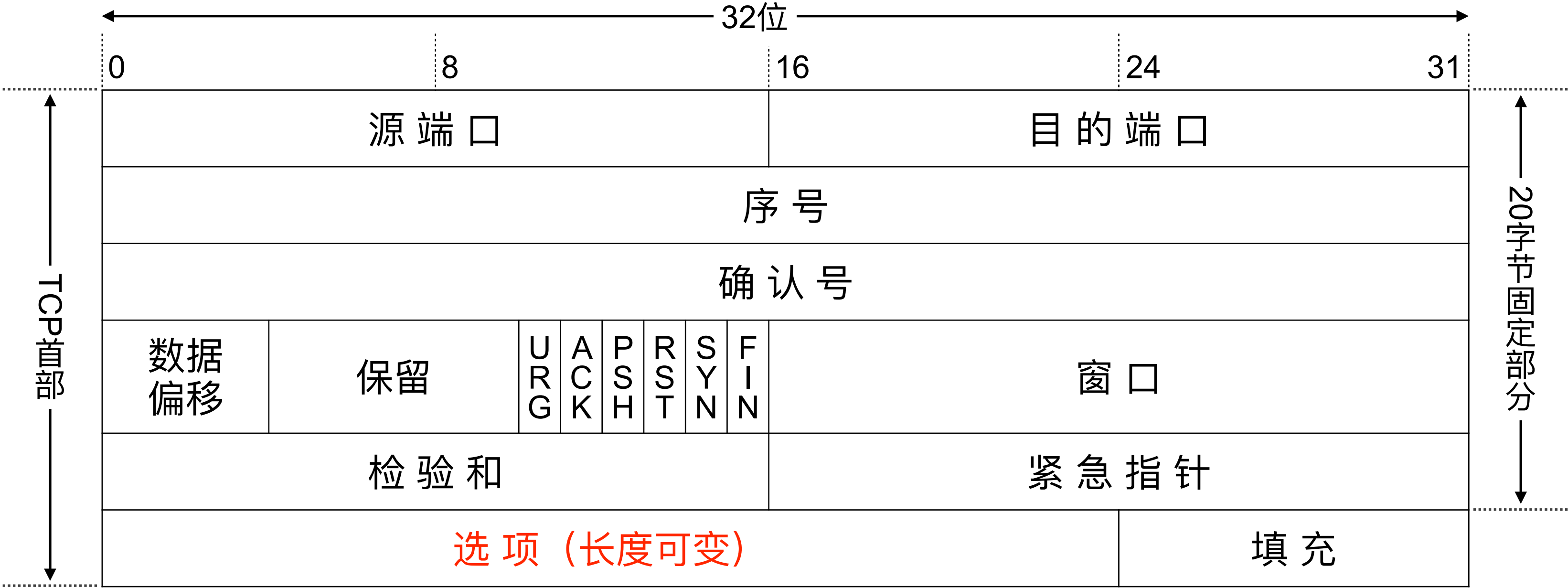
- 运输层
- TCP报文格式
- 源端口目的端口
- 序号、确认号
- 数据偏移
- 六个控制位
- 窗口、检验和
- 紧急指针
- 选项、填充



**紧急指针字段：**占 16 位，指出在本报文段中紧急数据共有多少个字节，紧急数据放在本报文段数据的最前面。

# TCP报文段首部格式（选项）

- 运输层
  - TCP报文格式
  - 源端口目的端口
  - 序号、确认号
  - 数据偏移
  - 六个控制位
  - 窗口、检验和
  - 紧急指针
  - 选项、填充



**选项字段：**长度可变。TCP 最初只有一种选项，即最大报文段长度 MSS。MSS 告诉对方 TCP：“我的缓存所能接收的报文段的数据字段的最大长度是 MSS 个字节。”

**MSS (Maximum Segment Size)：**是 TCP 报文段中的数据字段的最大长度。数据字段加上 TCP 首部才等于整个的 TCP 报文段。

# 为什么要规定MSS

- 运输层
  - TCP报文格式
  - 源端口目的端口
  - 序号、确认号
  - 数据偏移
  - 六个控制位
  - 窗口、检验和
  - 紧急指针
  - 选项、填充

- 提升传输效率：
  - MSS太小，效率太低；
  - MSS太大，IP层需要分片，只要有一片出错，TCP需要重传；
  - MSS尽可能大些，只要在IP层不分片即可；
  - 最佳的MSS很难选择。

# 其他选项

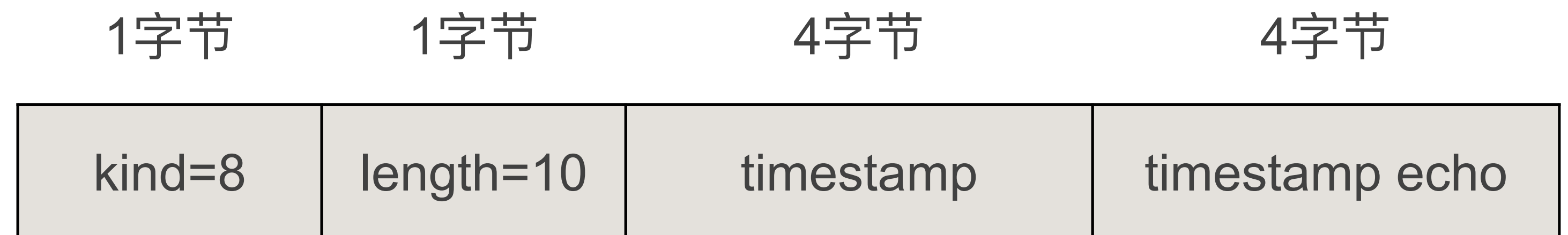
- 运输层
  - TCP报文格式
  - 源端口目的端口
  - 序号、确认号
  - 数据偏移
  - 六个控制位
  - 窗口、检验和
  - 紧急指针
  - 选项、填充

- **窗口扩大选项**：占3字节，其中有一个字节表示移位值S。新的窗口值等于TCP首部中的窗口位数增大到  $(16 + S)$ ，相当于把窗口值向左移动S位后获得实际的窗口大小。
- **时间戳选项**：占 10 字节，其中最主要的字段时间戳值字段（4 字节）和时间戳回送回答字段（4 字节）。
- **选择确认选项**：告诉发送方收到的连续的字节块。

## 其他选项：时间戳选项

- 运输层
  - TCP报文格式
  - 源端口目的端口
  - 序号、确认号
  - 数据偏移
  - 六个控制位
  - 窗口、检验和
  - 紧急指针
  - 选项、填充

- **计算往返时延**：发送把发送时间放入timestamp，确认报文把该时间戳复制到timestamp echo字段，并在timestamp放入确认时间。
- **防止序号回绕**： $2^{32}$ 的序号空间对于高带宽很容易消耗完。





## 其他选项：时间戳选项

- 运输层
  - TCP报文格式
  - 源端口目的端口
  - 序号、确认号
  - 数据偏移
  - 六个控制位
  - 窗口、检验和
  - 紧急指针
  - 选项、填充

TCP Option - Timestamps: TSval 830150137, TSecr 0  
Kind: Time Stamp Option (8)  
Length: 10  
Timestamp value: 830150137  
Timestamp echo reply: 0

TCP Option - Timestamps: TSval 905704537, TSecr 830150137  
Kind: Time Stamp Option (8)  
Length: 10  
Timestamp value: 905704537  
Timestamp echo reply: 830150137

```
Mac-mini:~ $ ssh -p xxx -l userxx 202.xxx.xxx.xxx
```



## 其他选项：时间戳选项

- 运输层
  - TCP报文格式
  - 源端口目的端口
  - 序号、确认号
  - 数据偏移
  - 六个控制位
  - 窗口、检验和
  - 紧急指针
  - 选项、填充

TCP Option - Timestamps: TSval 830150207, TSecr 905704537

Kind: Time Stamp Option (8)

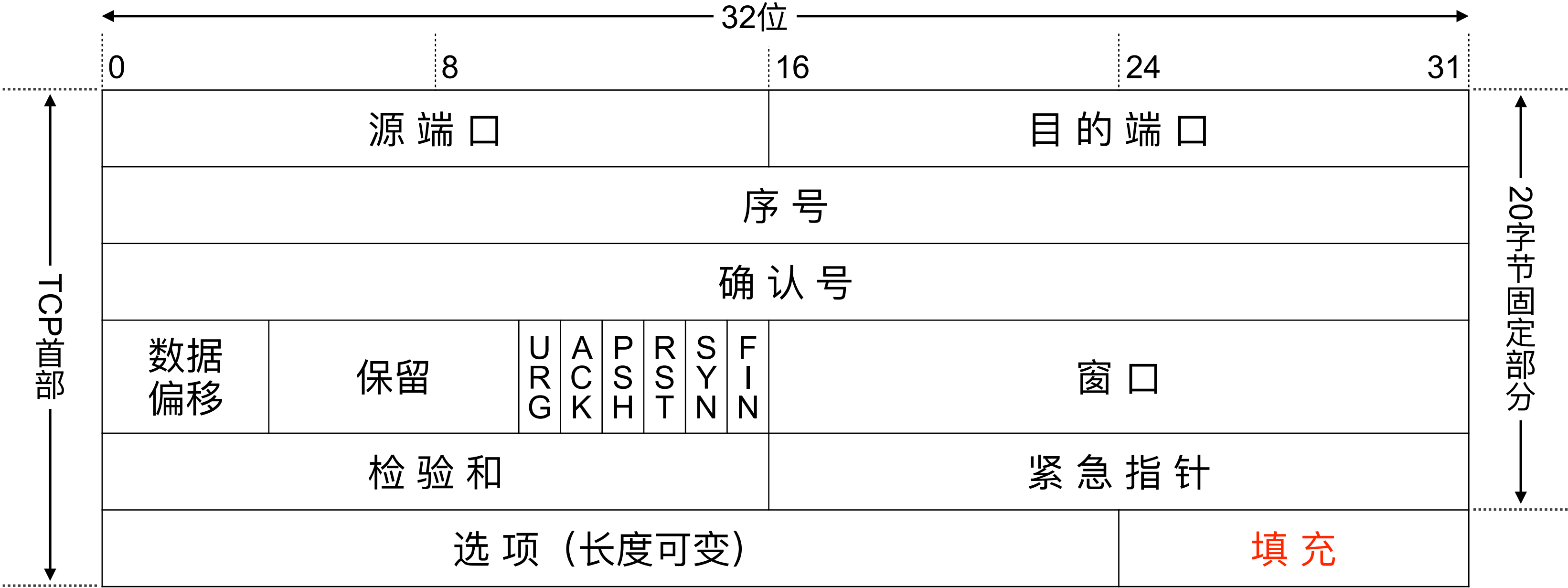
Length: 10

Timestamp value: 830150207

Timestamp echo reply: 905704537

# TCP报文段首部格式（选项）

- 运输层
  - TCP报文格式
  - 源端口目的端口
  - 序号、确认号
  - 数据偏移
  - 六个控制位
  - 窗口、检验和
  - 紧急指针
  - 选项、填充



填充字段：这是为了使整个首部长度是 4 字节的整数倍。

# 小结

- 运输层
- TCP报文格式
- 源端口目的端口
- 序号、确认号
- 数据偏移
- 六个控制位
- 窗口、检验和
- 紧急指针
- 选项、填充

- 固定首部长度20字节：
  - 源端口、目的端口；
  - 序号、确认号；
  - 数据偏移（首部长度）；
  - SYN、ACK、FIN、URG、PSH、RST；
  - 窗口、检验和、紧急指针。
- 选项最大40字节：
  - 时间戳选项、窗口扩大、选择确认。

# TCP 可靠传输的实现

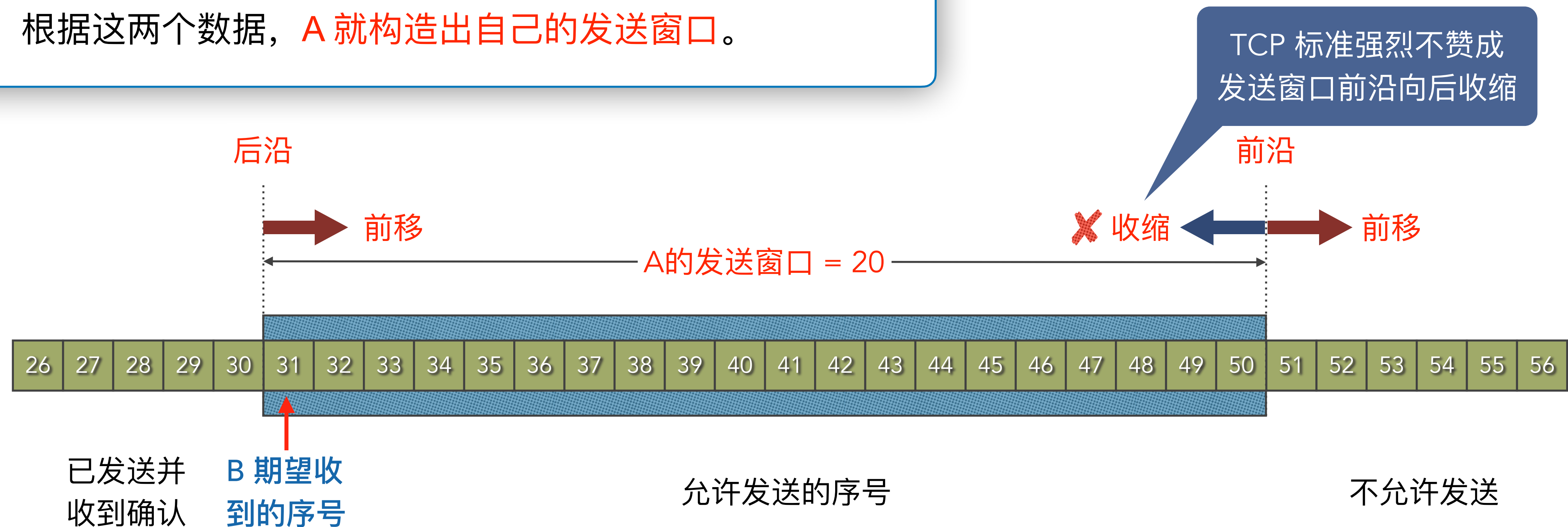
- 运输层
- 可靠传输的实现
- 加权平均往返时间
- 发送确认的时机
- 超时重传时间
- 选择确认

- TCP连接的两个端点都有两个窗口：一个发送窗口和一个接收窗口。
- TCP 的可靠传输机制用字节的序号进行控制。TCP 所有的确认都是基于序号而不是基于报文段的。
- TCP 两端的四个窗口经常处于动态变化之中。
- TCP连接的往返时间 RTT 不是固定不变的，需要使用特定的算法估算较为合理的重传时间。
- 发送窗口：准备发送的数据和已发送但未收到确认的数据。
- 接收窗口：按序到达但未被应用程序接收的数据、不按序到达的数据。

窗口、序号、确认、重传

# 以字节为单位的滑动窗口

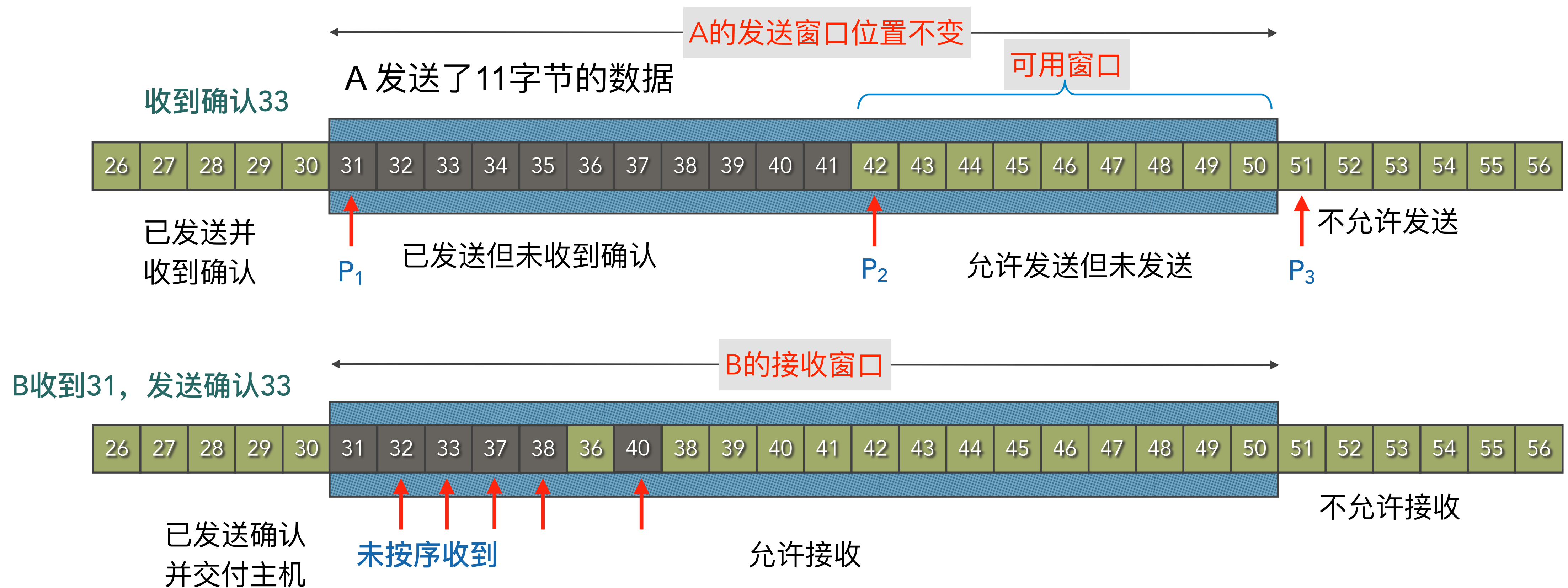
- A 收到了 B 的确认报文段：窗口值 20 字节，确认号为 31。
- 根据这两个数据，A 就构造出自己的发送窗口。



A 可以把落入发送窗口中的序号字节一次连续性全部发送出去：边发送边接收确认。



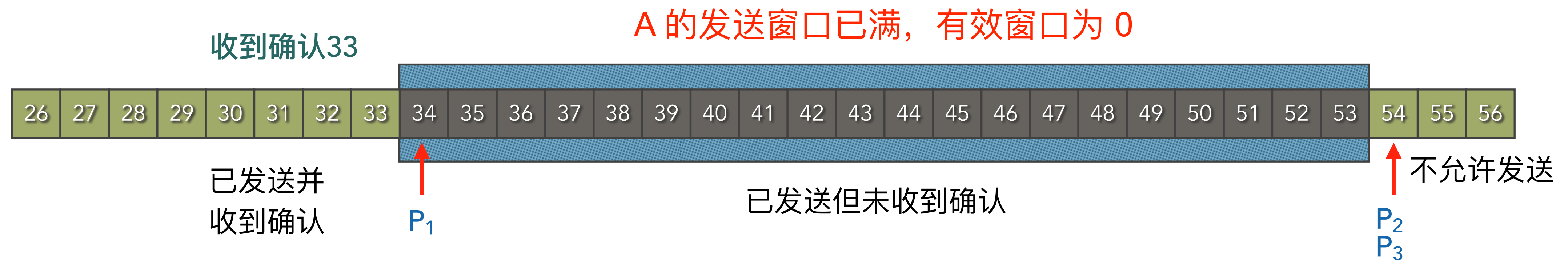
# 以字节为单位的滑动窗口



- $P_3 - P_1$  = A 的发送窗口（又称为通知窗口）；
- $P_2 - P_1$  = 已发送但尚未收到确认的字节数；
- $P_3 - P_2$  = 允许发送但尚未发送的字节数（又称为可用窗口）。

# 以字节为单位的滑动窗口

- A 的发送窗口内的序号都已用完，**由于未收到确认，必须停止发送。**
- A 的发送窗口并不总是和 B 的接收窗口一样大。
- **未按序到达的数据应如何处理：临时存放在接收窗口中。**
- 接收方必须有**累积确认**的功能，这样可以**减小传输开销**。



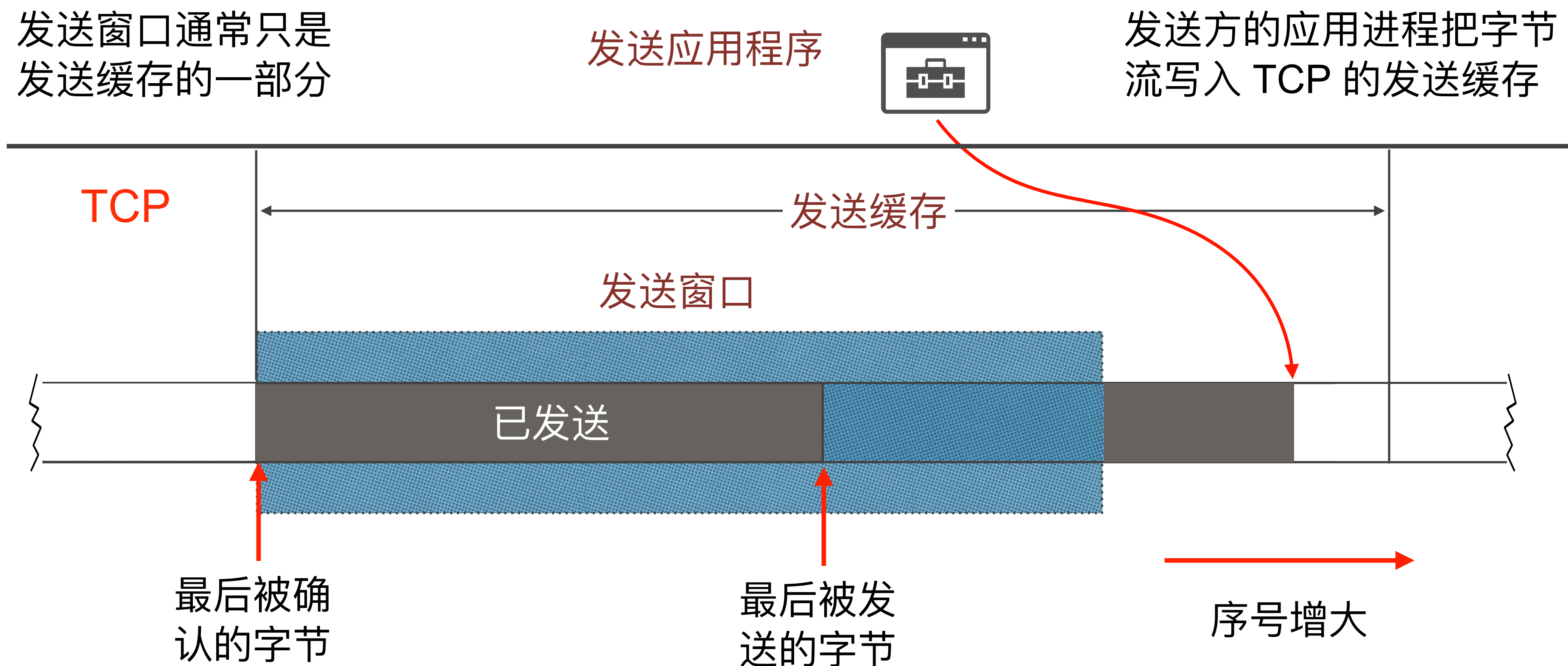
# 接收方发送确认的时机

- 运输层
- 可靠传输的实现
- 加权平均往返时间
- 发送确认的时机
- 超时重传时间
- 选择确认

- 接收方可以在合适的时候发送确认，也可以在自己有数据要发送时把确认信息顺便捎带上。
- 注意两点：
  - 第一，接收方不应过分推迟发送确认，否则会导致发送方不必要的重传；
  - 第二，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。



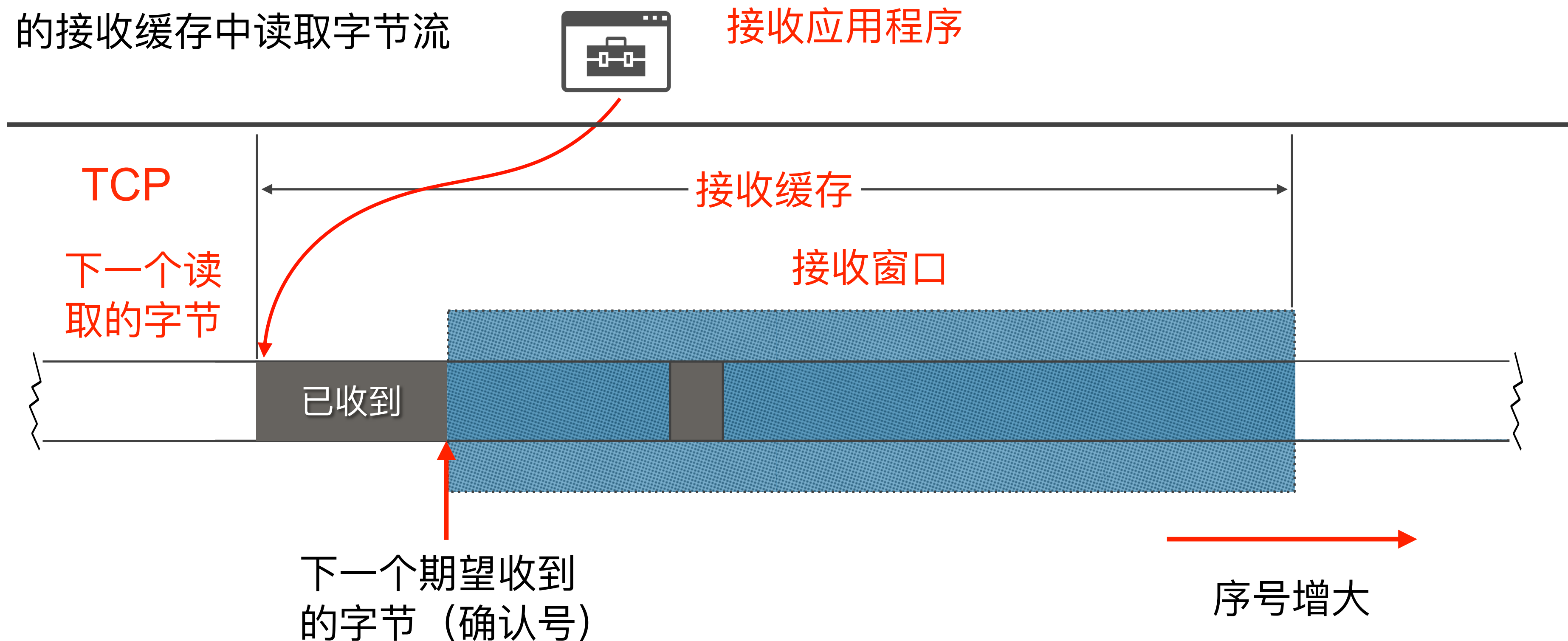
# 发送缓存的作用



发送应用程序传送给发送方 TCP 准备发送的数据；TCP 已发送出但尚未收到确认的数据以及等待进入发送窗口的数据

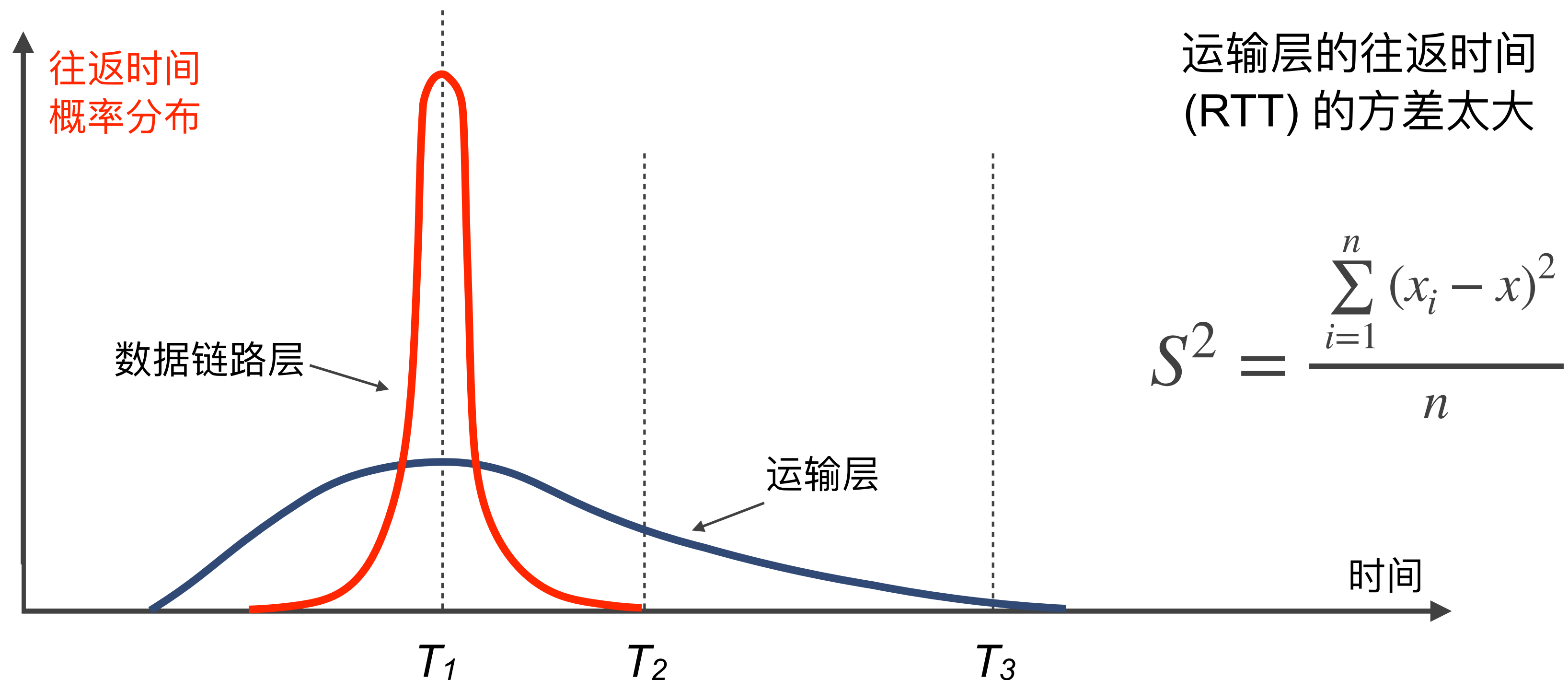
# 接收缓存的作用

接收方的应用进程从 TCP  
的接收缓存中读取字节流



按序到达的、但尚未被接收应用程序读取的数据；不按序到达的数据以及未进入到接收窗口的数据

# 超时重传时间的选择



- TCP 每发送一个报文段，就对这个报文段设置一次计时器。
- 只要计时器设置的重传时间到但还没有收到确认，就要重传这一报文段。
- 重传时间的选择是 TCP 最复杂的问题之一。

# TCP 超时重传时间设置

- 运输层
  - 可靠传输的实现
  - 加权平均往返时间
  - 发送确认的时机
  - 超时重传时间
  - 选择确认

- 超时重传时间设置得太短，引起过多的不必要的重传，网络负荷增大。
- 超时重传时间设置得过长，网络的空闲时间增大，网络传输效率降低。
- TCP 采用了一种自适应算法：
  - 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。

# 加权平均往返时间

- 运输层
  - 可靠传输的实现
  - 加权平均往返时间
  - 发送确认的时机
  - 超时重传时间
  - 选择确认

- TCP 保留了  $RTT$  的一个加权平均往返时间  $RTT_s$ （这又称为平滑的往返时间）。
- 第一次测量到  $RTT$  样本时， $RTT_s$  值就取为所测量到的  $RTT$  样本值。
- 以后每测量到一个新的  $RTT$  样本，就按下式重新计算一次  $RTT_s$ ：

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

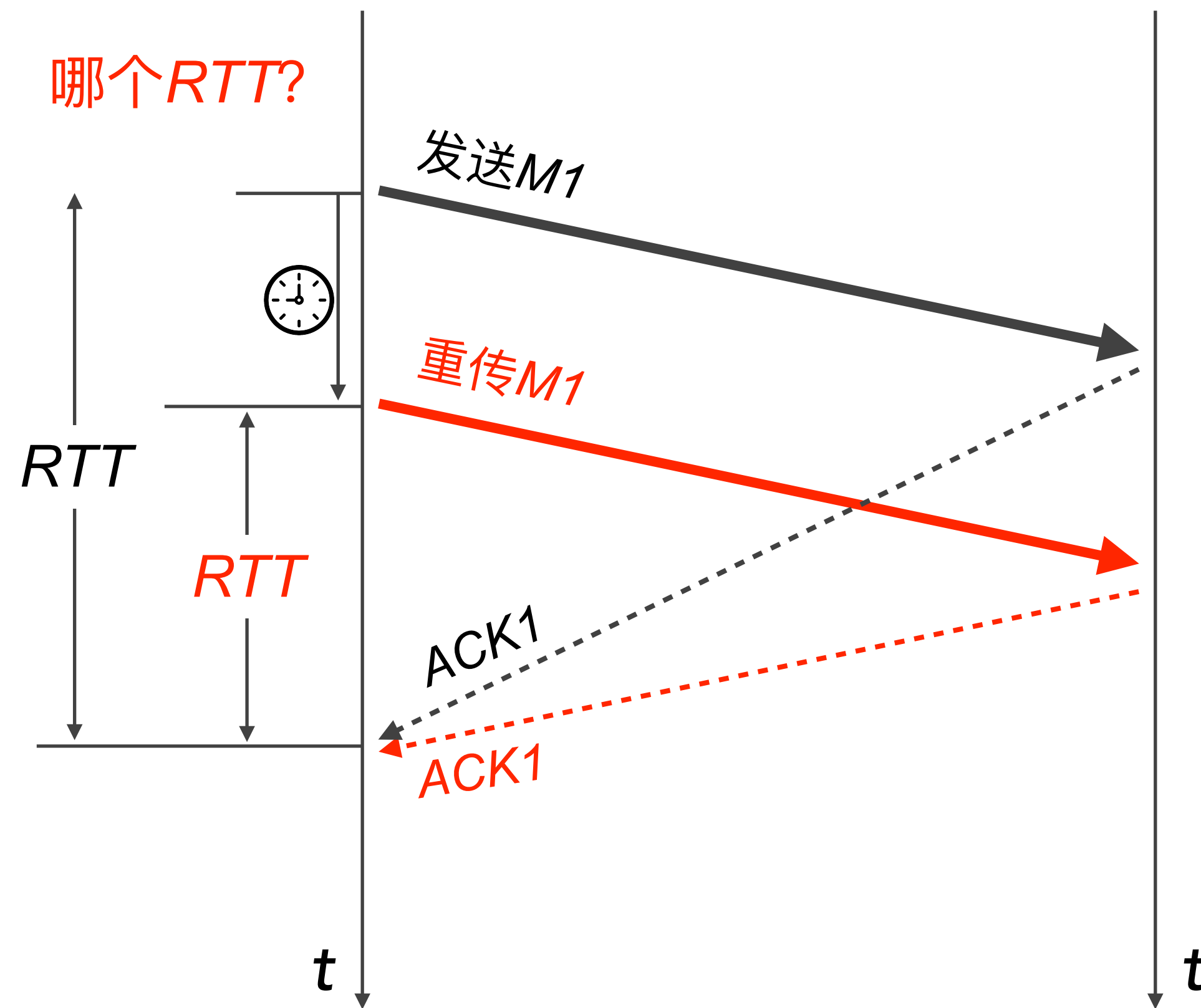
- 式中， $0 \leq \alpha \leq 1$ 。若  $\alpha$  接近于零，表示  $RTT$  值更新较慢。
- 若选择  $\alpha$  接近于 1，则表示  $RTT$  值更新较快。
- RFC 2988 推荐的  $\alpha$  值为 1/8，即 0.125。



# 超时重传时间 RTO

- 运输层
  - 可靠传输的实现
  - 加权平均往返时间
  - 发送确认的时机
  - 超时重传时间
  - 选择确认
- $RTO$  应略大于加权平均往返时间  $RTT_S$ 。
- RFC 2988 建议使用下式计算  $RTO$ :
$$RTO = RTT_S + 4 \times RTT_D$$
  - $RTT_D$  是  $RTT$  的偏差的加权平均值；
  - 第一次测量时， $RTT_D$  值取为测量到的  $RTT$  样本值的一半，在以后的测量中，则使用下式计算  $RTT_D$ :
$$\text{新的 } RTT_D = (1 - \beta) \times (\text{旧的 } RTT_D) + \beta \times |RTT_S - \text{新的 } RTT \text{ 样本}|$$
  - $\beta$  是个小于 1 的系数，其推荐值是 1/4，即 0.25。

# 往返时间 (RTT) 的测量相当复杂



- **Karn 算法:**
- 只要报文段重传了, **不采用**其往返时间样本。
- **新的问题:**
  - 当时延突然增大了很多时, 一段时间内, TCP可能重传很多报文段;
  - 根据 Karn 算法, 不考虑重传的报文段的往返时间样本的话, **超时重传时间就无法更新。**

- TCP 报文段 1 没有收到确认。重传 (即报文段 1) 后, 收到了确认报文段 ACK1。
- 如何判定此确认报文段是**对原来的报文段 1 的确认**, 还是**对重传的报文段 1 的确认**?

# 修正的 Karn 算法

- 运输层
  - 可靠传输的实现
  - 加权平均往返时间
  - 发送确认的时机
  - 超时重传时间
  - 选择确认

报文段每重传一次，就把 RTO 增大一些：

$$\text{新的 } RTO = \gamma \times (\text{旧的 } RTO)$$

- 系数  $\gamma$  的典型值是 2。
- 当不再发生报文段的重传时，才根据报文段的往返时延更新平均往返时延  $RTT$  和超时重传时间  $RTO$  的数值。
- 实践证明，这种策略较为合理。

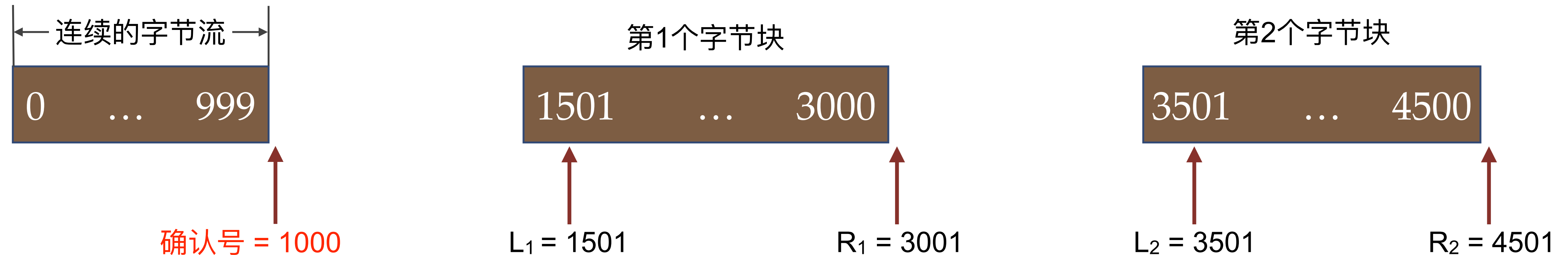


# 选择确认

- 运输层
  - 可靠传输的实现
  - 加权平均往返时间
  - 发送确认的时机
  - 超时重传时间
  - 选择确认

- 问题：
  - 若收到的报文段无差错，只是未按序号，中间还缺少一些序号的数据，那么能否设法只传送缺少的数据而不重传已经正确到达接收方的数据？
- 答案：
  - 选择确认 SACK (Selective ACK) 就是一种可行的处理方法。

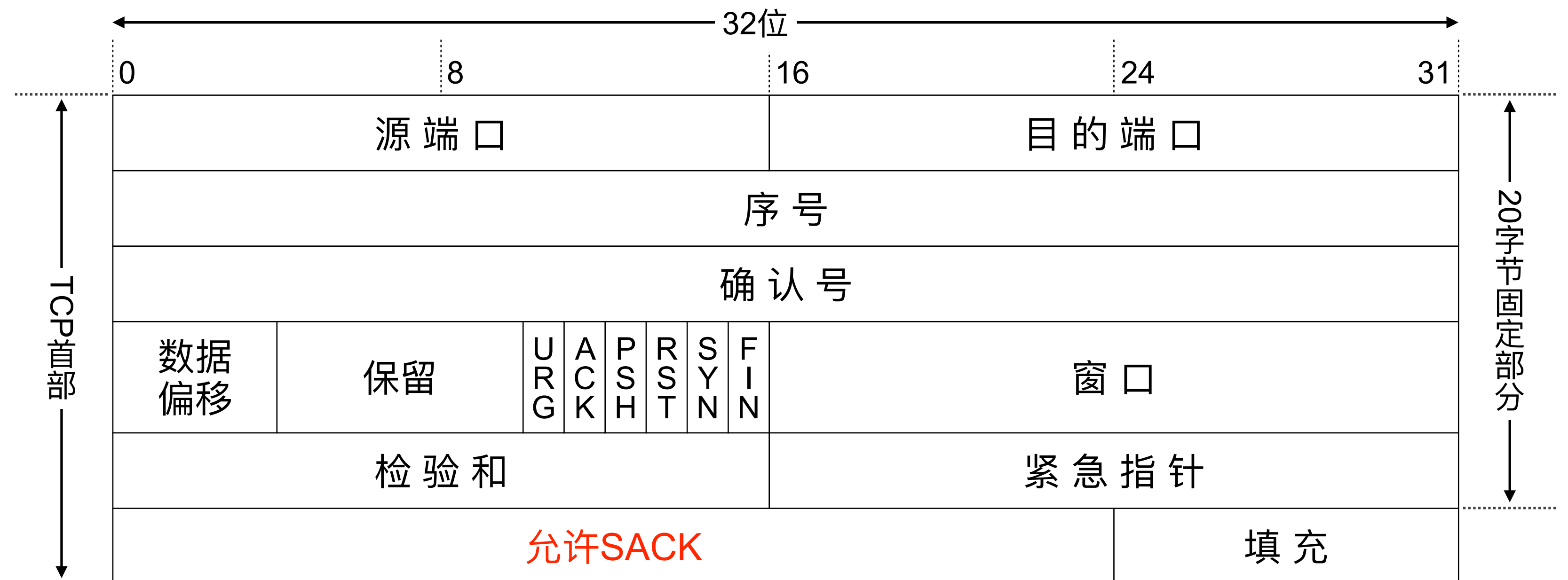
# 接收到的字节流序号不连续的问题



- 接收方收到了和前面的字节流不连续的两个字节块。
- 如果这些字节的序号都在接收窗口之内，那么接收方就先收下这些数据，但要把这些信息准确地告诉发送方，使发送方不要再重复发送这些已收到的数据。

# 选择确认

- 运输层
  - 可靠传输的实现
  - 加权平均往返时间
  - 发送确认的时机
  - 超时重传时间
  - 选择确认



- 双方建立 TCP 连接时，在 TCP 首部的选项中加上“允许 SACK”的选项。
- 首部选项的长度最大有 40 字节，指明一个字节块用掉 8 字节，因此在选项中最多只能指明 4 个字节块的边界信息（另一个字节用于指明是什么选项）。

# 小结

- 运输层
  - 可靠传输的实现
  - 加权平均往返时间
  - 发送确认的时机
  - 超时重传时间
  - 选择确认
- 可靠传输的实现：
  - 滑动窗口机制；
  - 发送窗口、接收窗口；
  - 确认、序号、重传。
  - 发送缓存、接收缓存：
  - 超时重传时间选择：
    - $RTO = RTT_S + 4 \times RTT_D$
  - Karn 算法。
    - 新的  $RTO = \gamma \times$  ( 旧的  $RTO$  )
  - 选择确认。

# TCP 的流量控制

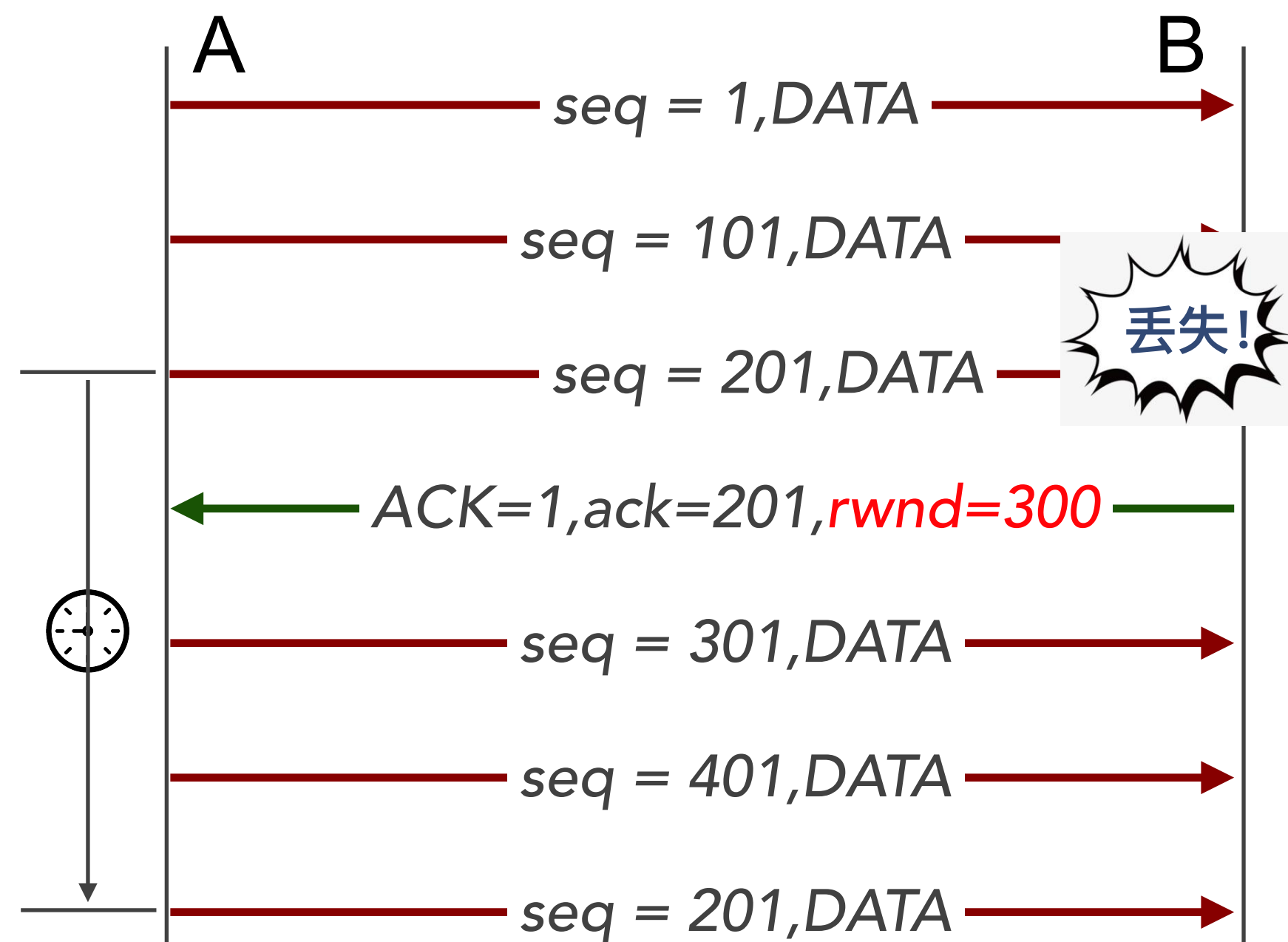
- 运输层
  - TCP流量控制
  - 死锁问题
  - TCP传输效率
  - 糊涂窗口综合症
    - 发送窗口
    - 接收窗口

- 利用滑动窗口实现流量控制。
- TCP 的传输效率。



# 流量控制（A发B收，B的接收窗口为400字节）

ACK: 确认标志  
ack: 确认号  
seq: 序号  
每个TCP报文携带100字节数据



A发送序号1~100，还可发送300字节

A发送序号101~200，还可发送200字节

允许A发送序号201~500，共300字节

A发送序号301~400，还可发送100字节

A发送序号401~500，停止发送

A超时重传，不能发送新数据

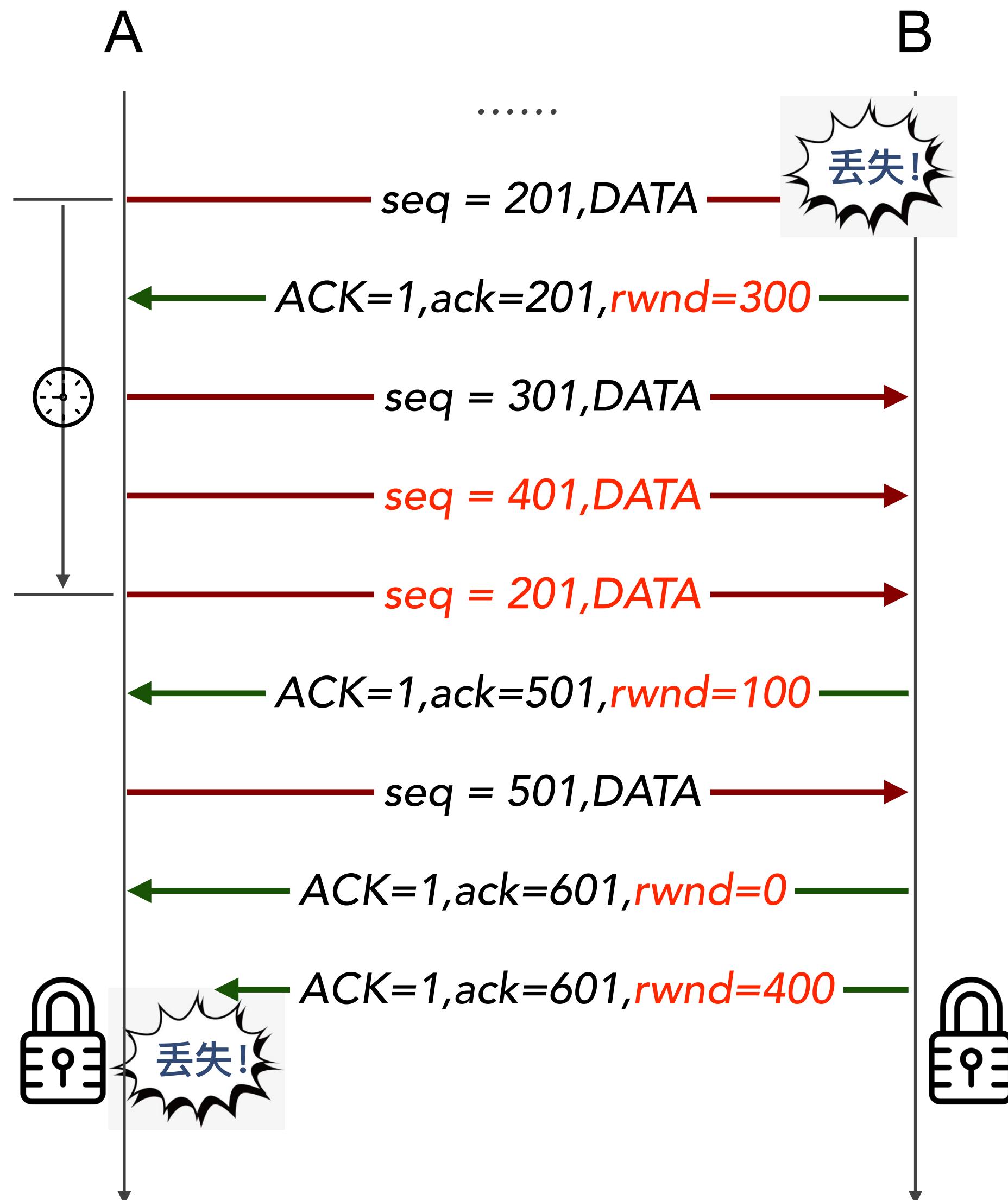
允许A发送序号501~600，共100字节

A发送序号501~600，停止发送

不允许A发送数据

让发送方的发送速度不要太快，既要让接收方来得及接收，也不要使网络发生拥塞。窗口机制在TCP连接上实现流量控制。

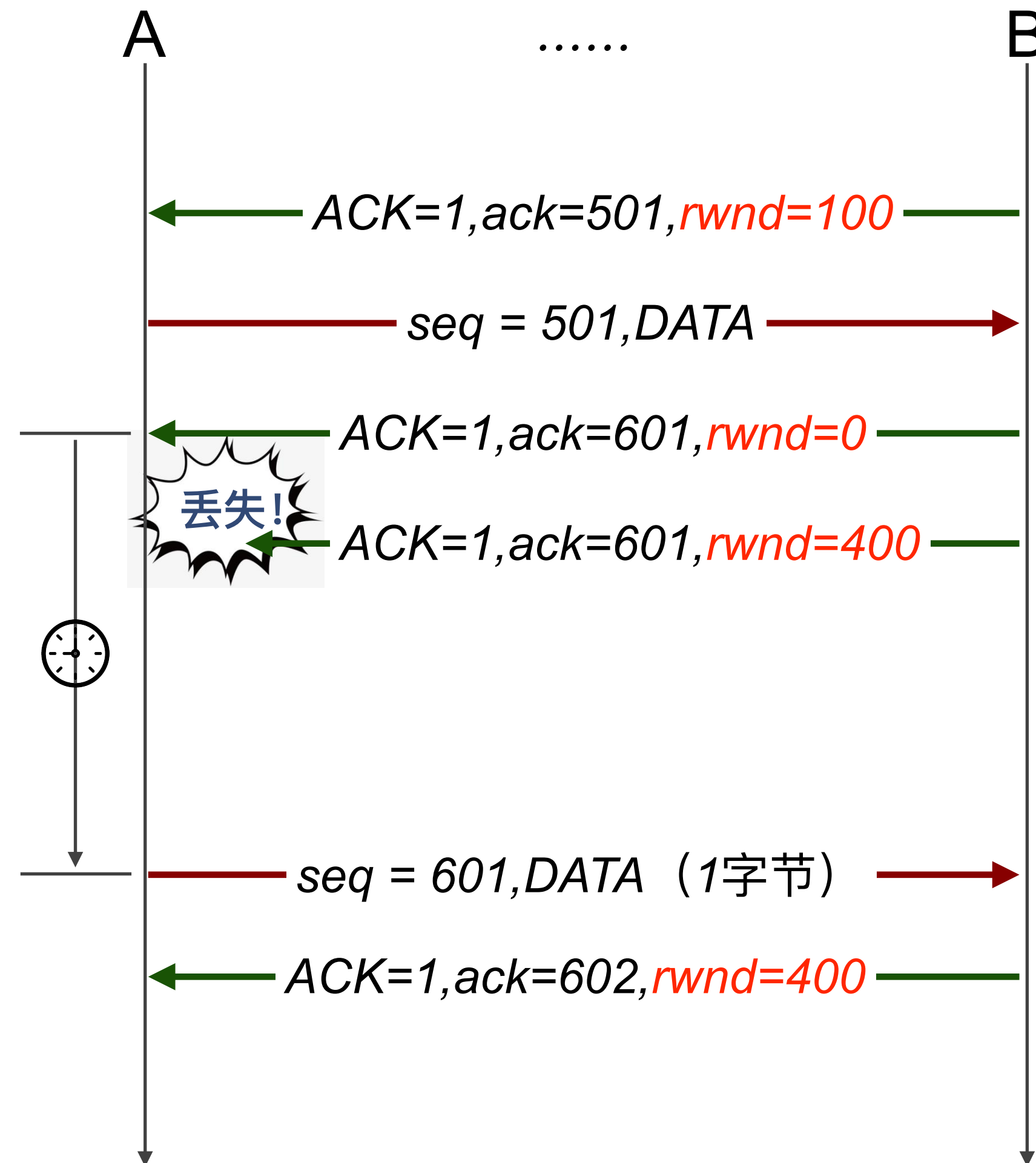
# 死锁问题



- B 向 A 发送了零窗口的报文段后，B 的接收缓存又有了一些存储空间。
- B 向 A 发送 `rwnd = 400` 的报文段。
- 这个报文段在传送过程中丢失了：
  - A 一直等待收到 B 发送的非零窗口的通知；
  - B 一直等待 A 发送的数据；
  - 如果没有其他措施，互相等待的死锁局面将一直延续下去。



# 死锁问题



- TCP 为每一个连接设有一个**持续计时器**。
- 只要收到对方的零窗口通知，就**启动该持续计时器**：
  - 持续计时器到期，发送一个**零窗口探测报文段**，对方在确认这个探测报文段时给出**现在的窗口值**；
  - 若**窗口仍然是零**，接收确认报文方**重新设置持续计时器**；
  - 若窗口不是零，死锁的僵局便被打破了。



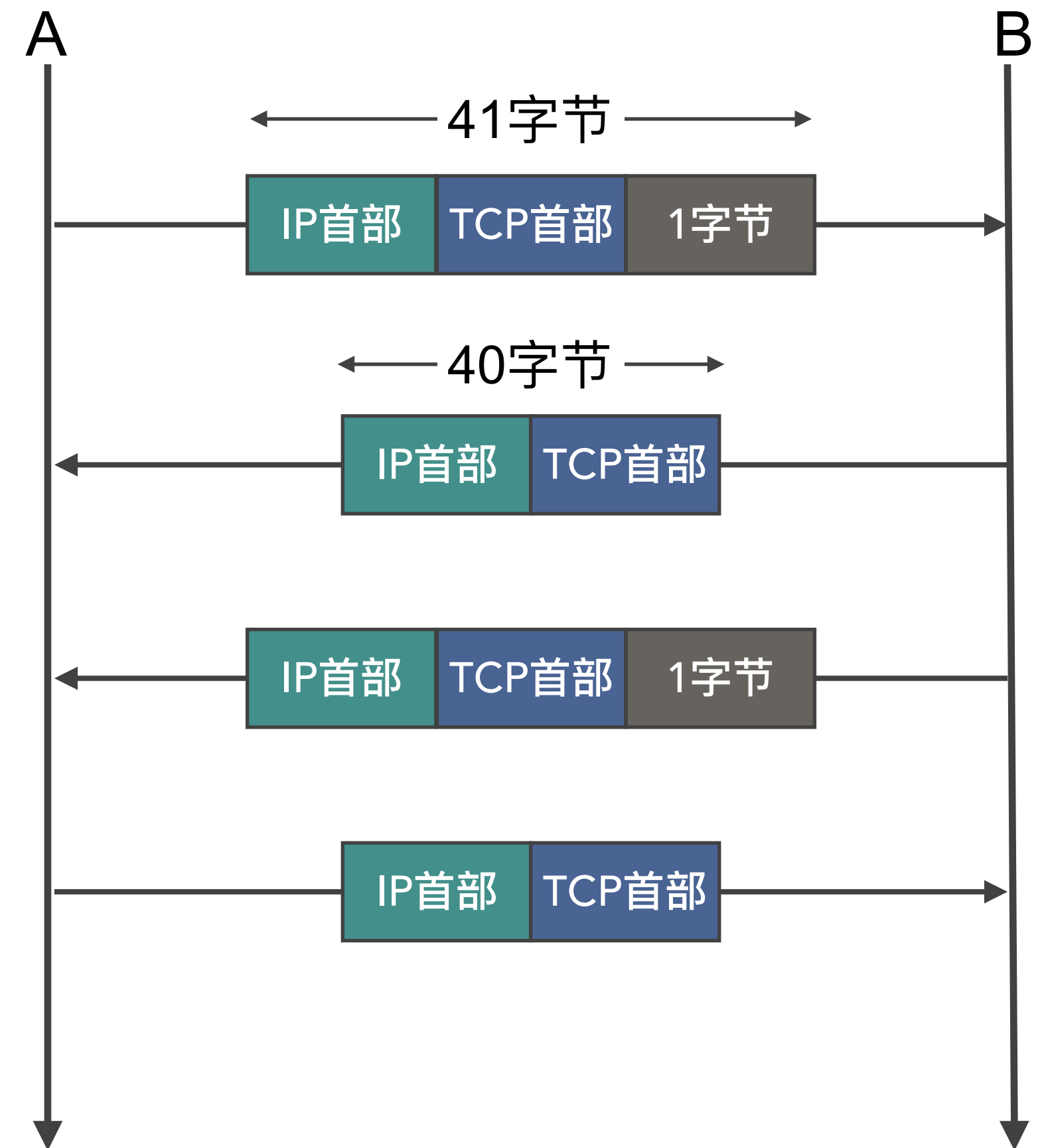
# TCP协议传输效率问题

- 运输层
  - TCP流量控制
  - 死锁问题
  - TCP传输效率
  - 糊涂窗口综合症
    - 发送窗口
    - 接收窗口

- 可以用不同的机制来控制 TCP 报文段的发送时机：
  - 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，就组装成一个 TCP 报文段发送出去；
  - 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；
  - 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（但长度不能超过 MSS）发送出去。

# 发送方糊涂窗口综合症

- 考虑Telnet连接上**字符传输问题**：
  - 发送方发送一个字符；
  - 接收方确认该字符（未携带数据）；
  - 接收方回送该字符；
  - 发送方确认回显字符（未携带数据）。
- 传输 **2 字节数据**，共需4个TCP报文，传输**总长度为162字节**，效率太低。



# Nagle算法

- 运输层
  - TCP流量控制
  - 死锁问题
  - TCP传输效率
  - 糊涂窗口综合症
    - 发送窗口
    - 接收窗口

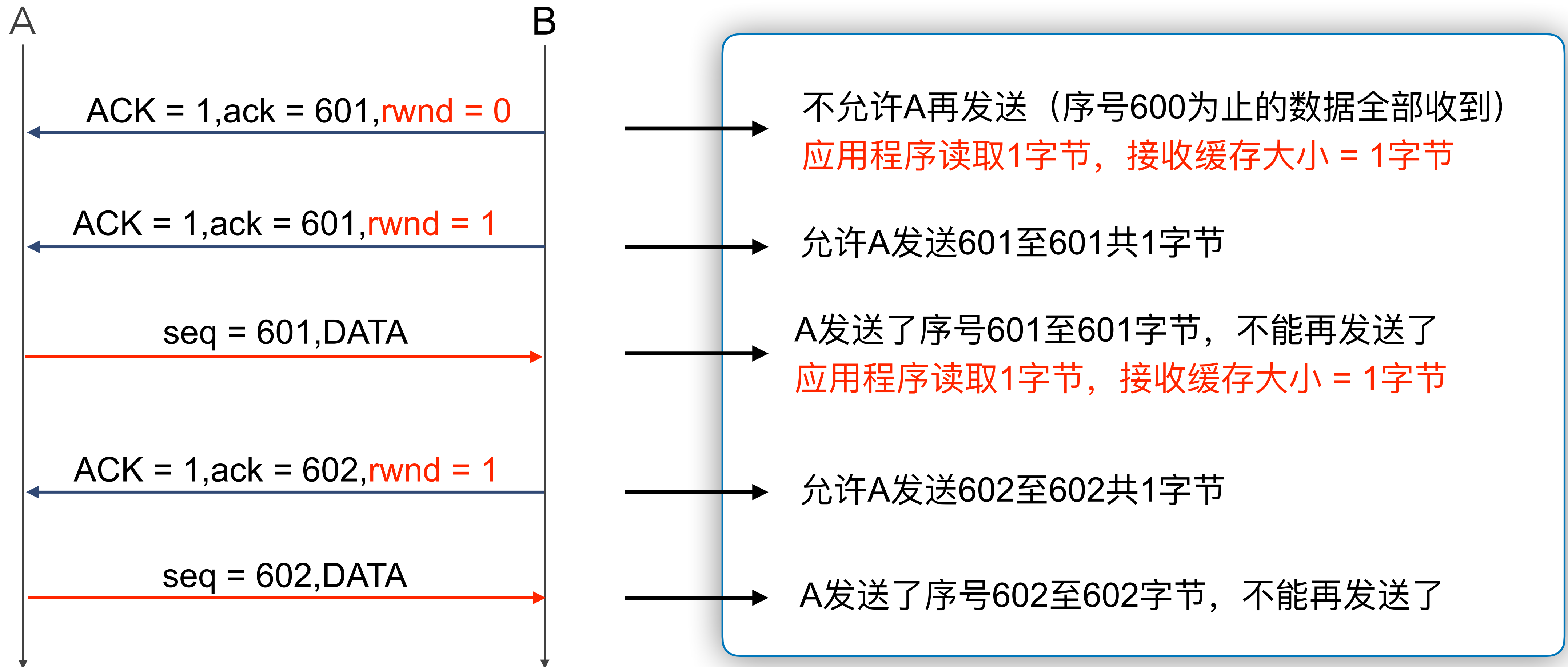
- 发送方先发送第一个数据字节，后面到达的数据字节缓存起来。
- 发送方收到对第一个数据字符的确认后，把发送缓存中的所有数据组装成一个报文段发送出去，继续对随后到达的数据进行缓存。
- 只有在收到对前一个报文段的确认后才继续发送下一个报文段。
- 当到达的数据已达到发送窗口大小的一半或已达到报文段的最大长度时，就立即发送一个报文段。

# 接收方糊涂窗口综合症

- 运输层
  - TCP流量控制
  - 死锁问题
  - TCP传输效率
  - 糊涂窗口综合症
    - 发送窗口
    - 接收窗口

- 接收方的 TCP 缓冲区已满，接收方会向发送方发送窗口大小为 0 的报文。
- 接收方的应用进程以交互方式每次只读取一个字节，接收方发送窗口大小为一个字节的确认报文，发送方发送一个字节的数  
据，接收窗口又满了，如此循环往复。
- 解决方法：
  - 接收方等待一段时间，使得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。

# 接收方糊涂窗口综合症



**原因：**接收方应用进程消耗数据太慢，例如：每次只读取一个字节。

# 小结

- 运输层
  - TCP流量控制
  - 死锁问题
  - TCP传输效率
  - 糊涂窗口综合症
    - 发送窗口
    - 接收窗口

- 流量控制：
  - 利用可变的滑动窗口控制发送方的发送数据的速率。
- 死锁问题，解决办法：
  - 持续计时器、探测报文。
- 传输效率（发送一个TCP报文段的时机）：
  - 数据达到MSS；
  - 推送（PUSH）；
  - 计时器到时。
- 糊涂窗口综合症：
  - 发送窗口糊涂综合症（Nagle算法）；
  - 接收窗口糊涂综合症（等待一段时间）。

# 拥塞控制的一般原理（拥塞的概念）

- 运输层
  - TCP拥塞控制
  - 拥塞控制的作用
  - 拥塞控制的原理
  - 网络拥塞的指标
  - 拥塞控制的方法
  - 拥塞的判断依据

- 在某段时间，若对网络中某资源的需求超过了该资源所能提供的可用部分，网络的性能就要变坏。这种现象称为拥塞 (congestion)。
- 若网络中有许多资源同时产生拥塞，网络的性能就要明显变坏，整个网络的吞吐量将随输入负荷的增大而下降。

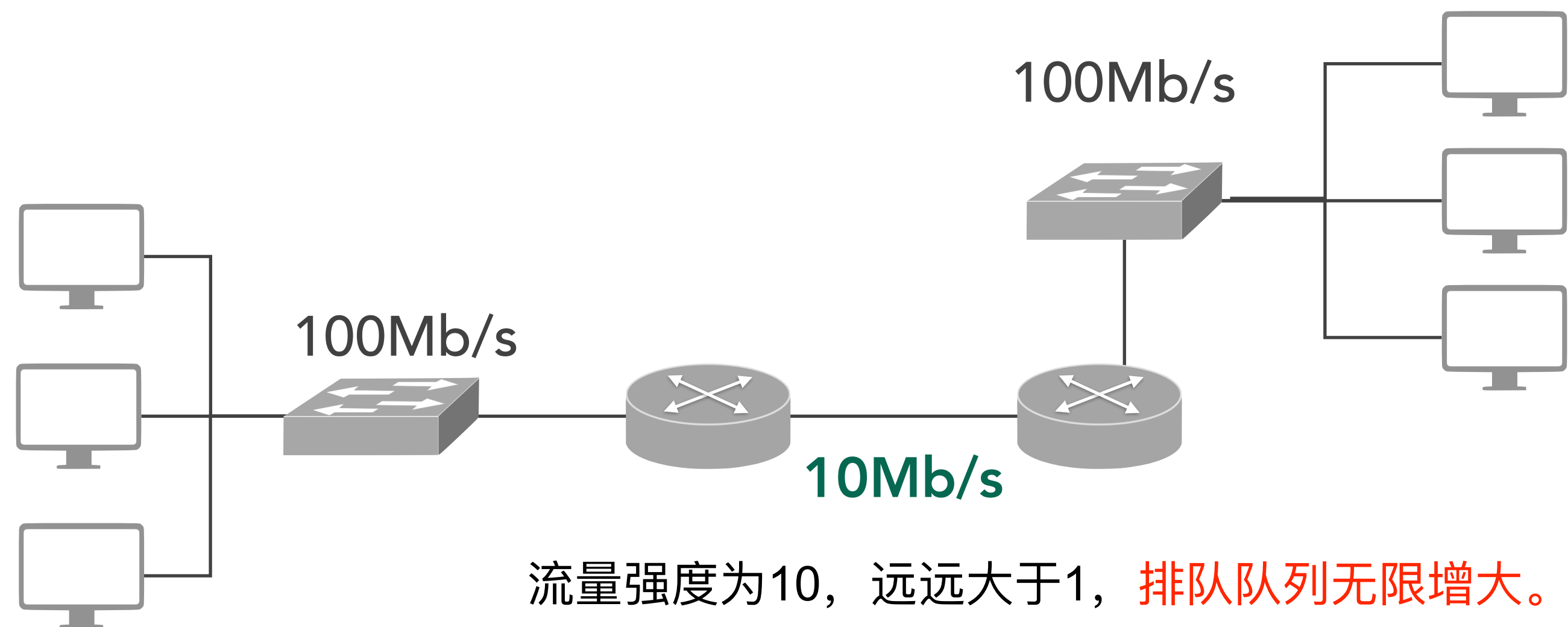


# 拥塞控制的一般原理（拥塞的原因）

- 运输层
  - **TCP拥塞控制**
  - 拥塞控制的作用
  - 拥塞控制的原理
  - 网络拥塞的指标
  - 拥塞控制的方法
  - 拥塞的判断依据

- 链路容量不足、资源分配不均衡；
- 路由器缓存空间、流量分布不均衡；
- 处理机速度太慢。

出现拥塞的原因： $\Sigma$  对资源需求 > 可用资源





# 增加资源能解决拥塞吗？

- 运输层
  - **TCP拥塞控制**
  - 拥塞控制的作用
  - 拥塞控制的原理
  - 网络拥塞的指标
  - 拥塞控制的方法
  - 拥塞的判断依据

- **不能：**
  - 网络拥塞是一个非常复杂的问题；
  - 简单地采用上述做法，不但不能解决拥塞问题，还可能使网络的性能更坏。
  - 网络拥塞往往是由许多因素引起的，例如：
    - **增大缓存**，但未提高输出链路的容量和处理机的速度，排队等待时间将会大大增加，引起大量超时重传，解决不了网络拥塞；
    - **提高处理机处理的速率**会将瓶颈转移到其他地方。

# 拥塞常常趋于恶化

- 运输层
  - TCP拥塞控制
  - 拥塞控制的作用
  - 拥塞控制的原理
  - 网络拥塞的指标
  - 拥塞控制的方法
  - 拥塞的判断依据
- 路由器没有足够的缓存空间，它就会丢弃一些新到的分组。
- 分组被丢弃时，发送这一分组的源点就会重传这一分组，甚至可能还要重传多次。这样会引起更多的分组流入网络和被网络中的路由器丢弃。
- 可见拥塞引起的重传并不会缓解网络的拥塞，反而会加剧网络的拥塞。
- 拥塞控制：防止过多的数据注入到网络中，使网络中的路由器或链路不致过载。
- 拥塞控制的前提：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。

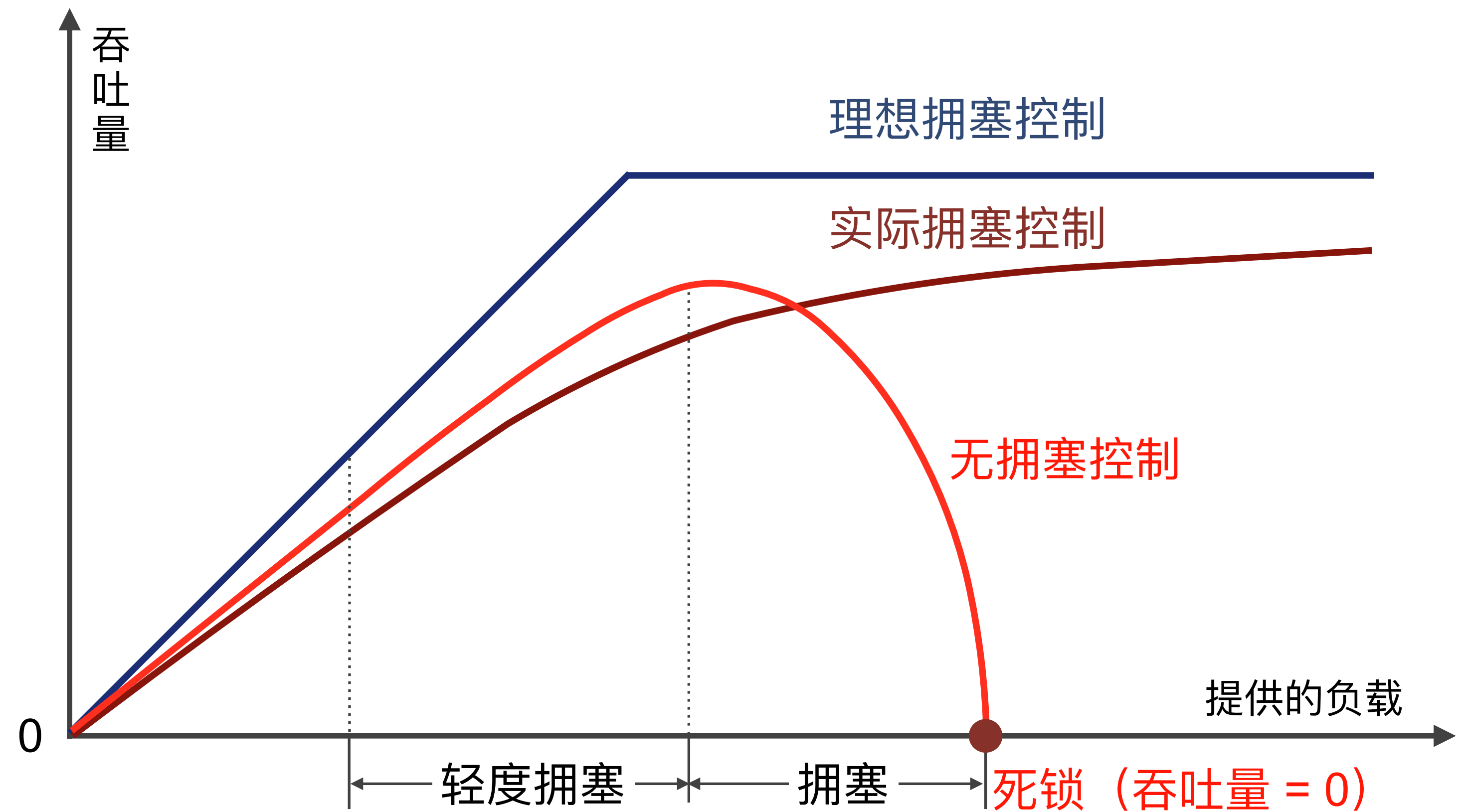
# 拥塞控制与流量控制的区别

- 运输层
  - TCP拥塞控制
  - 拥塞控制的作用
  - 拥塞控制的原理
  - 网络拥塞的指标
  - 拥塞控制的方法
  - 拥塞的判断依据

- **流量控制**：是端到端的问题（接收端控制发送端），**点对点通信量的控制**。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**：是一个**全局性的过程**，涉及到与降低网络传输性能有关的所有因素。防止过多数据注入到网络，使网络中的路由器或链路不致过载。
- 某些拥塞控制算法是**向发送端发送控制报文**，并告诉发送端，网络已出现麻烦，必须**放慢发送速率**，与流量控制是**很相似**。

# 拥塞控制所起的作用

- 运输层
  - TCP拥塞控制
  - 拥塞控制的作用
  - 拥塞控制的原理
  - 网络拥塞的指标
  - 拥塞控制的方法
  - 拥塞的判断依据



# 拥塞控制的一般原理

- 运输层
  - TCP拥塞控制
  - 拥塞控制的作用
  - 拥塞控制的原理
  - 网络拥塞的指标
  - 拥塞控制的方法
  - 拥塞的判断依据
- 由于拥塞控制是个动态问题，因此拥塞控制很难设计。
- 分组的丢失是网络发生拥塞的征兆而不是原因：
  - 数据链路层：帧出错被丢弃；
  - 网络层：出错IP分组被丢弃；
  - 拥塞控制本身也可能成为网络性能恶化甚至发生死锁的原因。
- 开环控制方法：在设计网络时考虑发生拥塞的因素，力求网络不产生拥塞。
- 闭环控制方法，基于反馈环路的概念：
  - 监测网络系统以便检测到拥塞在何时、何处发生；
  - 将拥塞发生的信息传送到可采取行动的地方；
  - 调整网络系统的运行以解决出现的问题。

# 监测网络的拥塞的指标

- 运输层
  - TCP拥塞控制
  - 拥塞控制的作用
  - 拥塞控制的原理
  - 网络拥塞的指标
  - 拥塞控制的方法
  - 拥塞的判断依据

- 主要指标有：
  - 由于缺少缓存空间而被丢弃的分组的百分数；
  - 平均队列长度；
  - 超时重传的分组数；
  - 平均分组时延；
  - 分组时延的标准差，等等。
- 上述这些指标的上升都标志着拥塞的增长。

# 拥塞通知的传递与时机

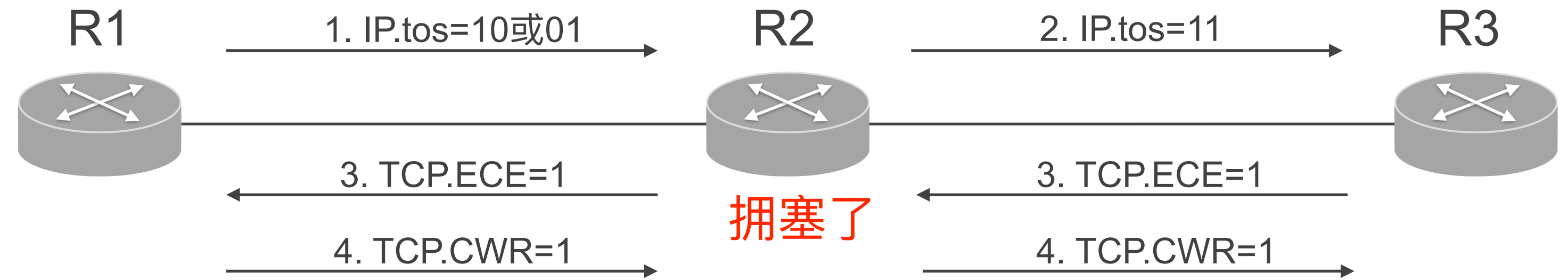
- 运输层
  - TCP拥塞控制
  - 拥塞控制的作用
  - 拥塞控制的原理
  - 网络拥塞的指标
  - 拥塞控制的方法
  - 拥塞的判断依据

- 拥塞控制的时机选择：
  - 过于频繁，会使系统产生不稳定的振荡；
  - 过于迟缓，采取行动又不具有任何价值。
- 采取的策略：
  - 发送“通知拥塞发生”的分组；
  - 在分组中保留表示拥塞状态的字段；
  - 周期性地发出探测分组等。



# 拥塞通知的传递与时机

- 运输层
  - TCP拥塞控制
  - 拥塞控制的作用
  - 拥塞控制的原理
  - 网络拥塞的指标
  - 拥塞控制的方法
  - 拥塞的判断依据



- 处理流程：
  - 设置IP首部为10或者01，IP经过拥塞处时设置IP首部的ECN域为11；
  - 接收端收到ECN位为11的TCP报文，设置TCP首部中的ECE标志；
  - 发送方收到设置ECE标志的ACK时，减小发送窗口，并在发送下一个TCP中设置CWR标志。接收方停止设置ECE标志。

- IP中的TOS字段的两个最低有效位：
  - 00 – 不支持 ECN 的传输；
  - 10 – 支持 ECN 的传输，ECT(0)；
  - 01 – 支持 ECN 的传输，ECT(1)；
  - 11 – 发生拥塞，CE(Congestion Encountered)。

- TCP报文首部中，除了6个常用的标志位，还有2个与拥塞有关的标志位：
  - E : ECE，显式拥塞提醒回应；
  - W : CWR，拥塞窗口减少。

# TCP 的拥塞控制方法

- 运输层
- TCP拥塞控制
- 拥塞控制的作用
- 拥塞控制的原理
- 网络拥塞的指标
- 拥塞控制的方法
- 拥塞的判断依据

- TCP 采用基于窗口的方法进行拥塞控制，该方法属于闭环控制方法。
- TCP发送方维持一个拥塞窗口 **CWND** (Congestion Window):
  - 拥塞窗口的大小取决于网络的拥塞程度，并且动态地在变化；
  - 发送端利用拥塞窗口根据网络的拥塞情况调整发送的数据量；
  - 网络没有出现拥塞，拥塞窗口增大一些，以便发送更多的分组，提高网络的利用率；
  - 网络出现拥塞或有可能出现拥塞，拥塞窗口减小一些，减少注入到网络中的分组数。

真正的发送窗口值 =  $\text{Min}(\text{公告窗口值}, \text{拥塞窗口值})$

# TCP 的拥塞的判断依据

- 运输层
  - TCP拥塞控制
  - 拥塞控制的作用
  - 拥塞控制的原理
  - 网络拥塞的指标
  - 拥塞控制的方法
  - 拥塞的判断依据

- 重传定时器超时：
  - 现在通信线路的传输质量一般都很好，因传输出差错而丢弃分组的概率很小（远小于 1 %）。只要出现了超时，就可以猜想网络可能出现了拥塞。
- 收到三个相同（重复）的 ACK：
  - 个别报文段会在网络中丢失，预示可能会出现拥塞（实际未发生拥塞），因此可以尽快采取控制措施，避免拥塞。

# 小结

- 运输层
  - TCP拥塞控制
  - 拥塞控制的作用
  - 拥塞控制的原理
  - 网络拥塞的指标
  - 拥塞控制的方法
  - 拥塞的判断依据
- 拥塞产生的原因：
  - $\Sigma$  对资源需求 > 可用资源。
  - 增加资源不能解决拥塞。
  - 拥塞控制与流量控制的区别。
  - 拥塞控制的前提，拥塞控制是一个动态问题。
  - 开环控制、闭环控制。
  - 监测网络的拥塞。
  - 拥塞控制的时机。

# TCP拥塞控制算法

- 运输层
  - 拥塞控制算法
  - 前提条件
  - 慢开始算法
  - 轮次与门限值
  - 拥塞避免
  - 快重传算法
  - 快恢复算法

- 慢开始 (slow-start)。
- 拥塞避免 (congestion avoidance)。
- 快重传 (fast retransmit)。
- 快恢复 (fast recovery)。

# 讨论前提条件

- 运输层
  - 拥塞控制算法
  - 前提条件
  - 慢开始算法
  - 轮次与门限值
  - 拥塞避免
  - 快重传算法
  - 快恢复算法

- 单向传输数据，另一个方向只发送确认。
- 接收方接收缓存无限大。
- 拥塞窗口：
  - 发送方根据网络拥塞情况估算的窗口值，反映当前网络容量；
  - 发送窗口 =  $\text{Min}(\text{接收窗口}, \text{拥塞窗口}) = \text{拥塞窗口}$ 。

# 慢开始 (Slow start)

- 运输层
  - 拥塞控制算法
  - 前提条件
  - 慢开始算法
  - 轮次与门限值
  - 拥塞避免
  - 快重传算法
  - 快恢复算法

- 目的：用来确定网络的负载能力或拥塞程度。
- 算法思路：由小到大逐渐增大拥塞窗口数值。
- 初始拥塞窗口 cwnd 设置：
  - 1 至 2 个最大报文段（旧的规定）；
  - 2 至 4 个最大报文段（RFC 5681 规定）。
- 慢开始门限 ssthresh（状态变量）：
  - 防止拥塞窗口cwnd 增长过大引起网络拥塞。
- 发送方最大报文段SMSS：
  - Sender Maximum Segment Size。

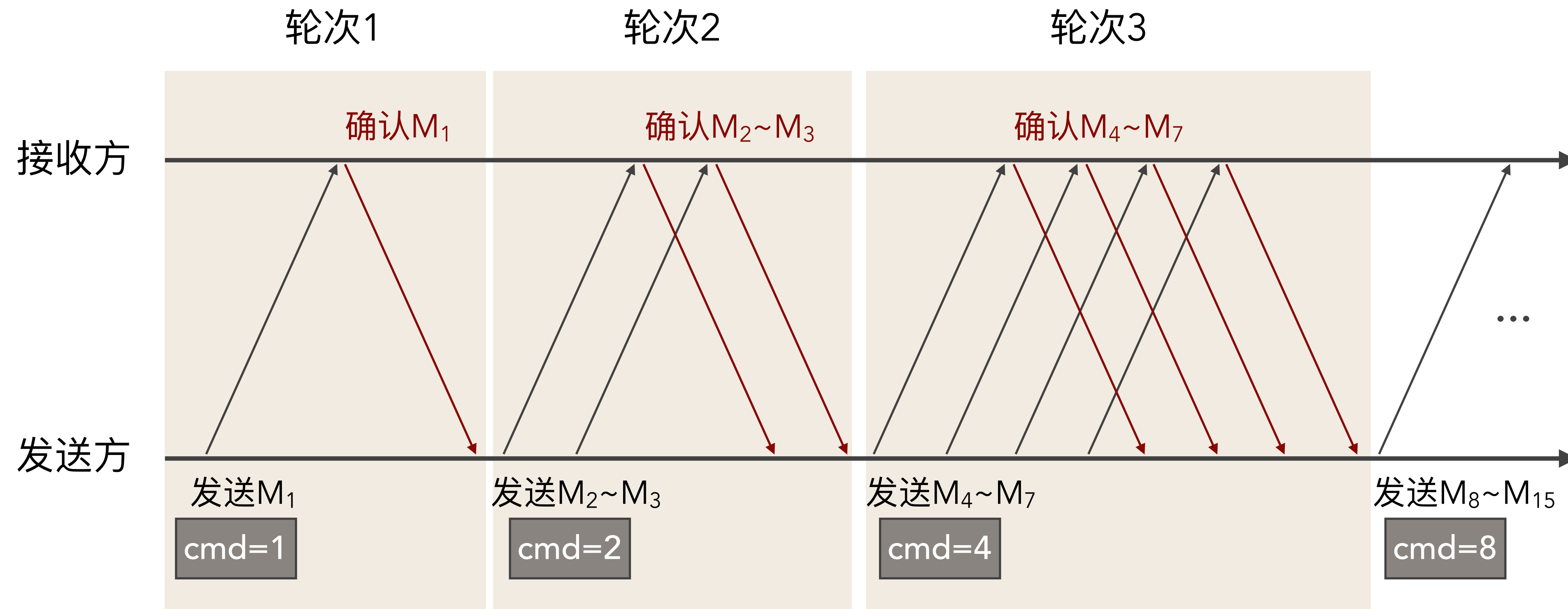


# 慢开始 (Slow start)

- 运输层
  - 拥塞控制算法
  - 前提条件
  - 慢开始算法
  - 轮次与门限值
  - 拥塞避免
  - 快重传算法
  - 快恢复算法

- 拥塞窗口控制方法：
  - 在每收到一个对新的报文段的确认后，把拥塞窗口增加最多一个 SMSS 的数值：  
**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**；
  - 当  $N < SMSS$  时，拥塞窗口每次的增加量小于 SMSS；
  - 采用逐步增大发送方的拥塞窗口的方法，注入数据至网络的速率更加合理；
  - 为讨论方便，窗口大小单位采用**报文段个数**计算。

# 慢开始 (Slow start)



- 发送方每收到一个对新报文段的确认  
(重传的不算在内) 就使 **cwnd 加 1**。
- 每经过一个传输轮次, 拥塞窗口就**加倍**。
- 窗口大小按指数增加, **不慢!**

# 传输轮次和慢开始门限值

- 运输层
  - 拥塞控制算法
  - 前提条件
  - 慢开始算法
  - 轮次与门限值
  - 拥塞避免
  - 快重传算法
  - 快恢复算法

- 慢开始门限 `ssthresh` :
  - 当  $cwnd < ssthresh$  时, 使用慢开始算法;
  - 当  $cwnd > ssthresh$  时, 停止使用慢开始算法而改用拥塞避免算法;
  - 当  $cwnd = ssthresh$  时, 既可使用慢开始算法, 也可使用拥塞避免算法。
- “传输轮次”更加强调:
  - 把拥塞窗口 `cwnd` 所允许发送的报文段都连续发送出去, 并收到了对已发送的最后一个字节的确认。

# 当网络出现拥塞时

- 运输层
  - 拥塞控制算法
  - 前提条件
  - 慢开始算法
  - 轮次与门限值
  - 拥塞避免
  - 快重传算法
  - 快恢复算法

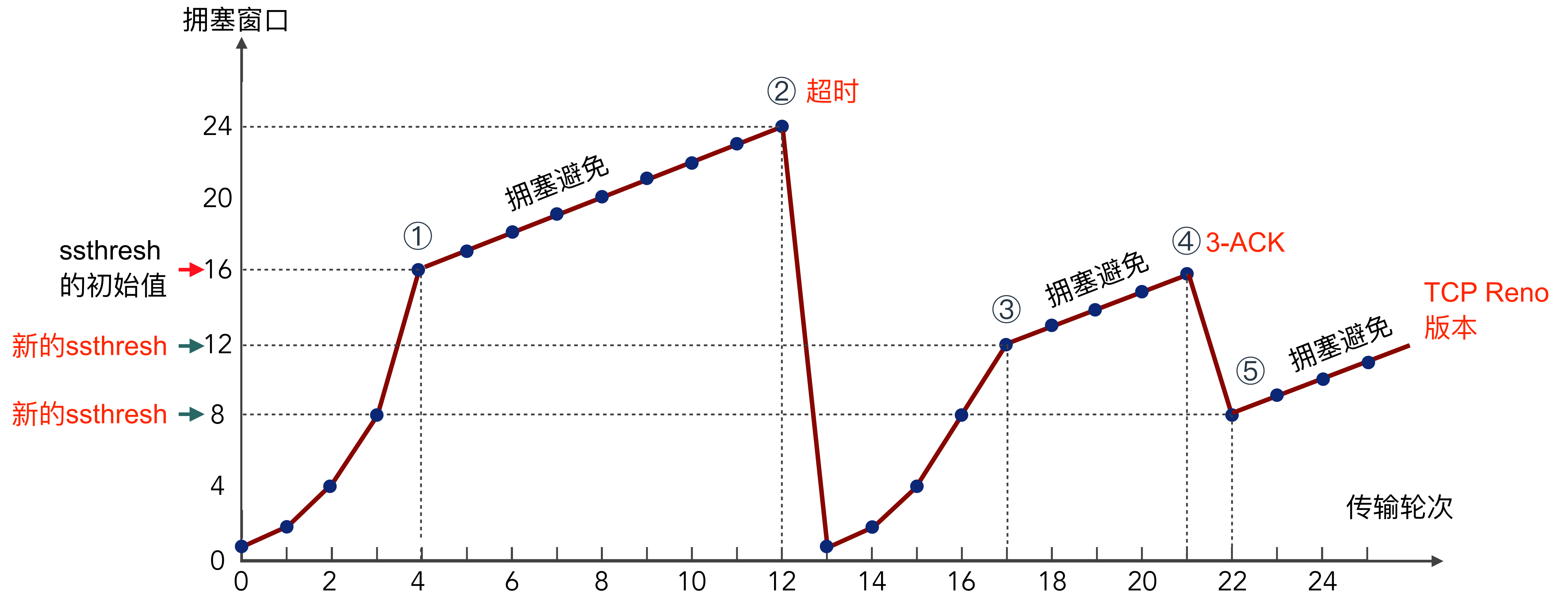
- 无论在慢开始阶段还是在拥塞避免阶段，只要发送方判断网络出现拥塞（重传定时器超时）：
  - $ssthresh = \max(cwnd/2, 2)$ ;
  - $cwnd = 1$ ;
  - 执行慢开始算法。
- 目的：
  - 就是要迅速减少主机发送到网络中的分组数，使得发生拥塞的路由器有足够时间把队列中积压的分组处理完毕。

# 拥塞避免

- 运输层
  - 拥塞控制算法
  - 前提条件
  - 慢开始算法
  - 轮次与门限值
  - 拥塞避免
  - 快重传算法
  - 快恢复算法

- 思路：
  - 让拥塞窗口  $cwnd$  缓慢地增大，避免出现拥塞；
  - 每经过一个传输轮次，拥塞窗口  $cwnd = cwnd + 1$ ；
  - 使拥塞窗口  $cwnd$  按线性规律缓慢增长；
  - 在拥塞避免阶段，具有“加法增大” (Additive Increase) 的特点。

# 慢开始和拥塞避免算法的实现举例



慢开始，拥塞避免；快重传，快恢复；加法增大，乘法减小。

# 快重传算法

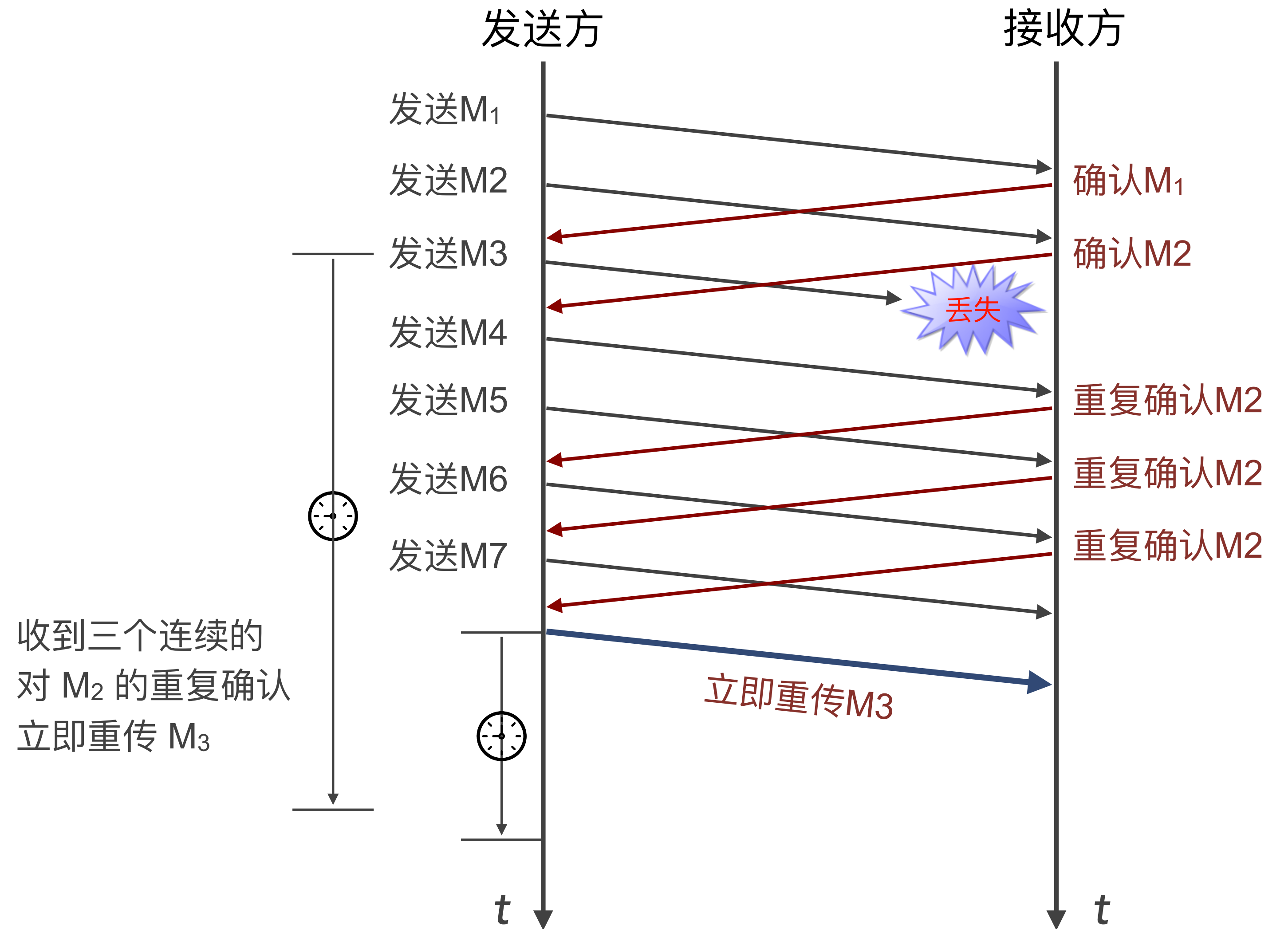
- 运输层
  - 拥塞控制算法
  - 前提条件
  - 慢开始算法
  - 轮次与门限值
  - 拥塞避免
  - 快重传算法
  - 快恢复算法

- 快重传FR (Fast Retransmission) 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，而是要立即发送确认，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。
- 发送方只要一连收到三个重复确认，应当立即进行重传（即“快重传”），这样就不会出现超时，发送方不会误认为出现了网络拥塞。
- 使用快重传可以使整个网络的吞吐量提高约20%。



# 快重传算法

- 运输层
  - 拥塞控制算法
  - 前提条件
  - 慢开始算法
  - 轮次与门限值
  - 拥塞避免
  - 快重传算法
  - 快恢复算法

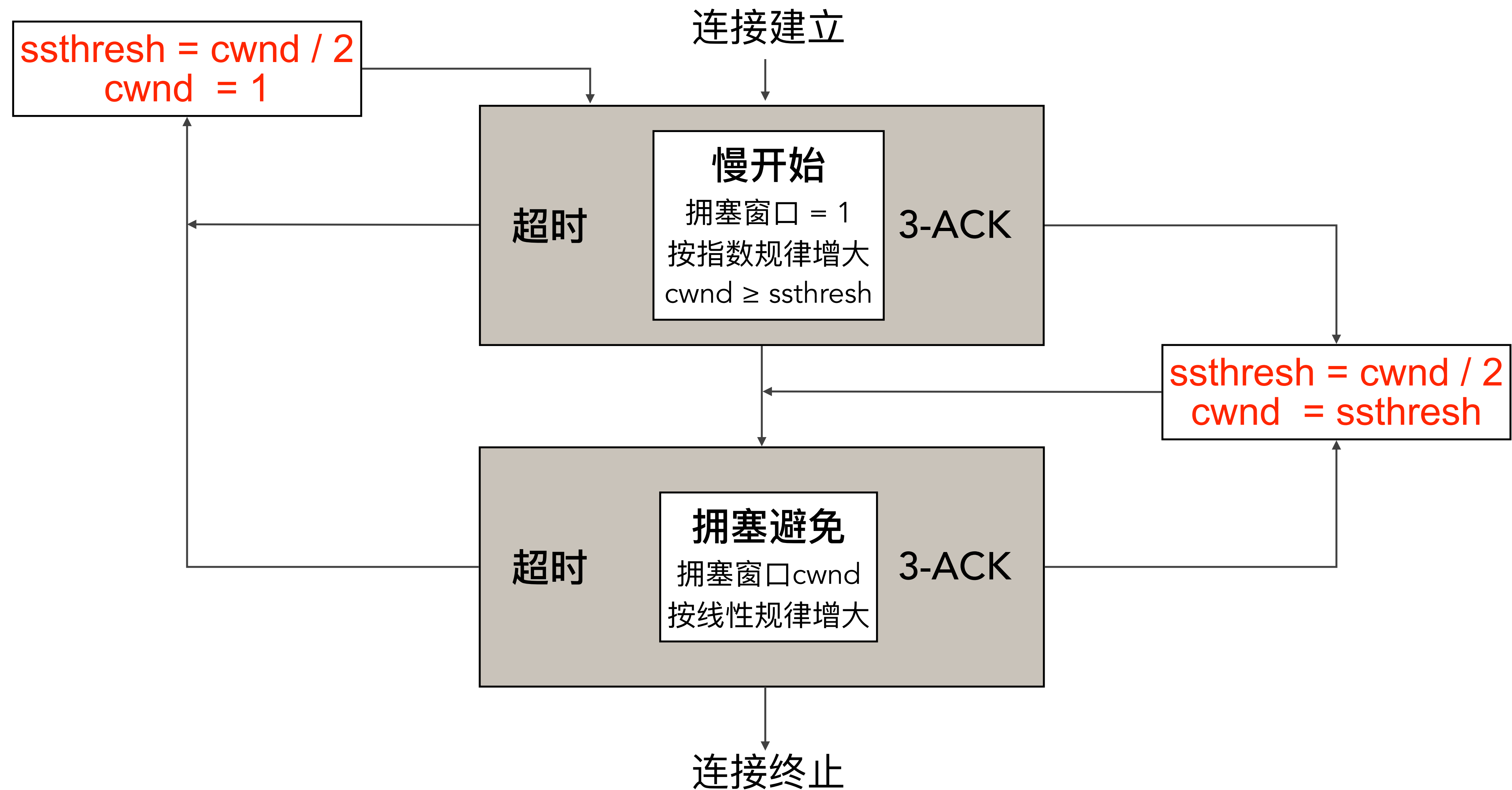


# 快恢复算法

- 运输层
  - 拥塞控制算法
  - 前提条件
  - 慢开始算法
  - 轮次与门限值
  - 拥塞避免
  - 快重传算法
  - 快恢复算法

- 当发送端收到连续三个重复的确认时，发送方认为网络很可能没有发生拥塞，因此不执行慢开始算法，而是执行快恢复算法  
FR (Fast Recovery) 算法：
  - 慢开始门限  $ssthresh = \text{当前拥塞窗口 } cwnd / 2$ ；
  - 新拥塞窗口  $cwnd = \text{慢开始门限 } ssthresh$ ；
  - 开始执行拥塞避免算法，使拥塞窗口缓慢地线性增大。

# TCP拥塞控制流程图



# 小结

- 运输层
  - 拥塞控制算法
  - 慢开始算法
  - 快重传算法
  - 快恢复算法

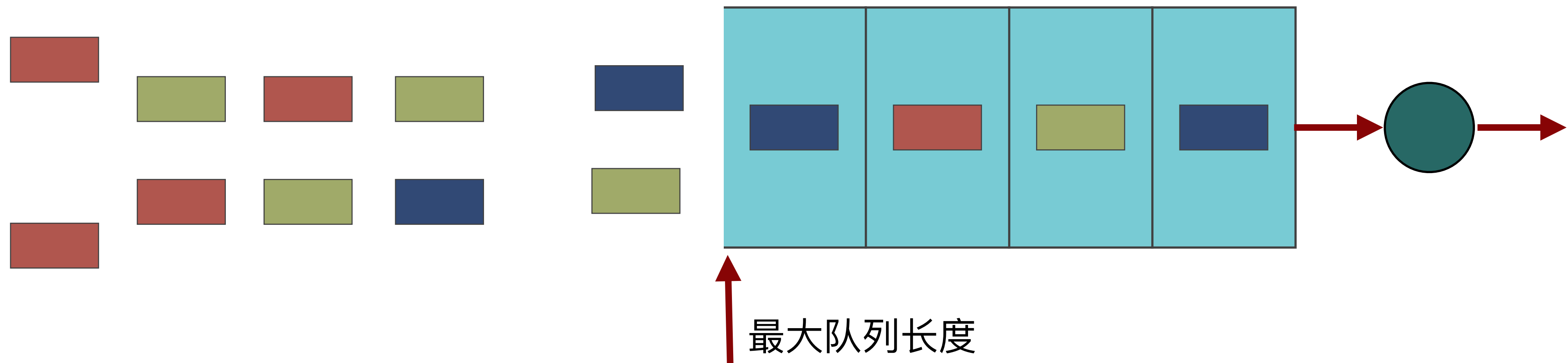
- 拥塞控制方法：
  - 慢开始，每个传输轮次之后，拥塞窗口值乘2；
  - 拥塞避免，每收到一个确认，拥塞窗口加1（加法增大）；
  - 乘法减小，超时重传或收到三个连续重复确认，门限值乘0.5；
  - 快重传，收到三个连续重复确认，立即重传丢失的报文段；
  - 快恢复，收到三个连续重复确认，门限值乘0.5，执行拥塞避免算法。

# 路由器分组管理策略

- 运输层
  - 分组管理策略
  - 主动队列管理

- 先进先出规则。
- 尾部丢弃策略。

## 路由器分组管理策略：先进先出规则与尾部丢弃策略

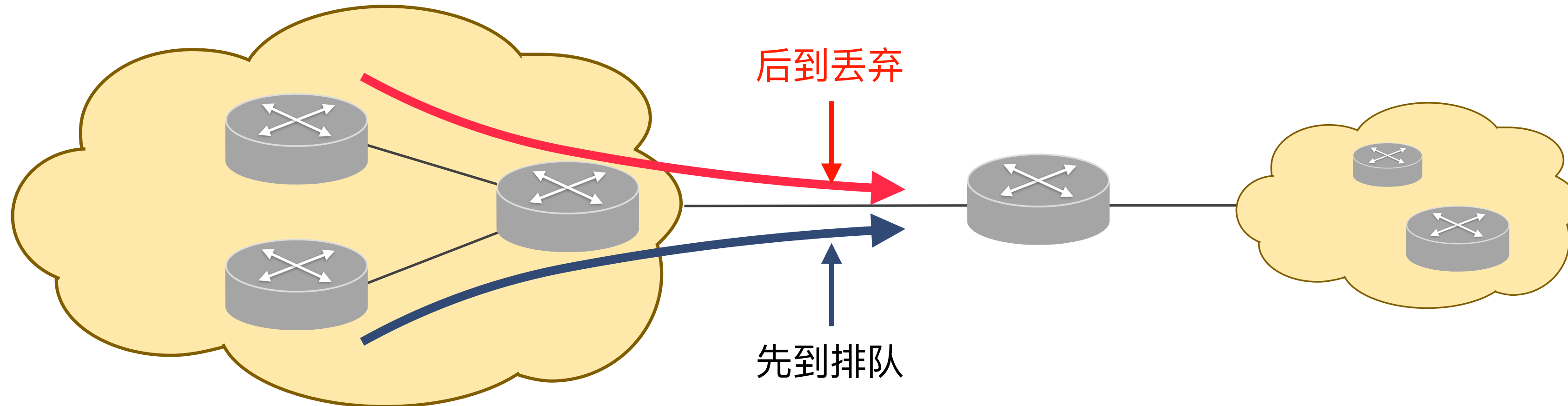


在最简单的情况下，路由器的队列通常采用先进先出 (FIFO) 规则与尾部丢弃策略 (tail-drop policy)。当队列已满时，以后再到达的所有分组将都被丢弃。

# 全局同步

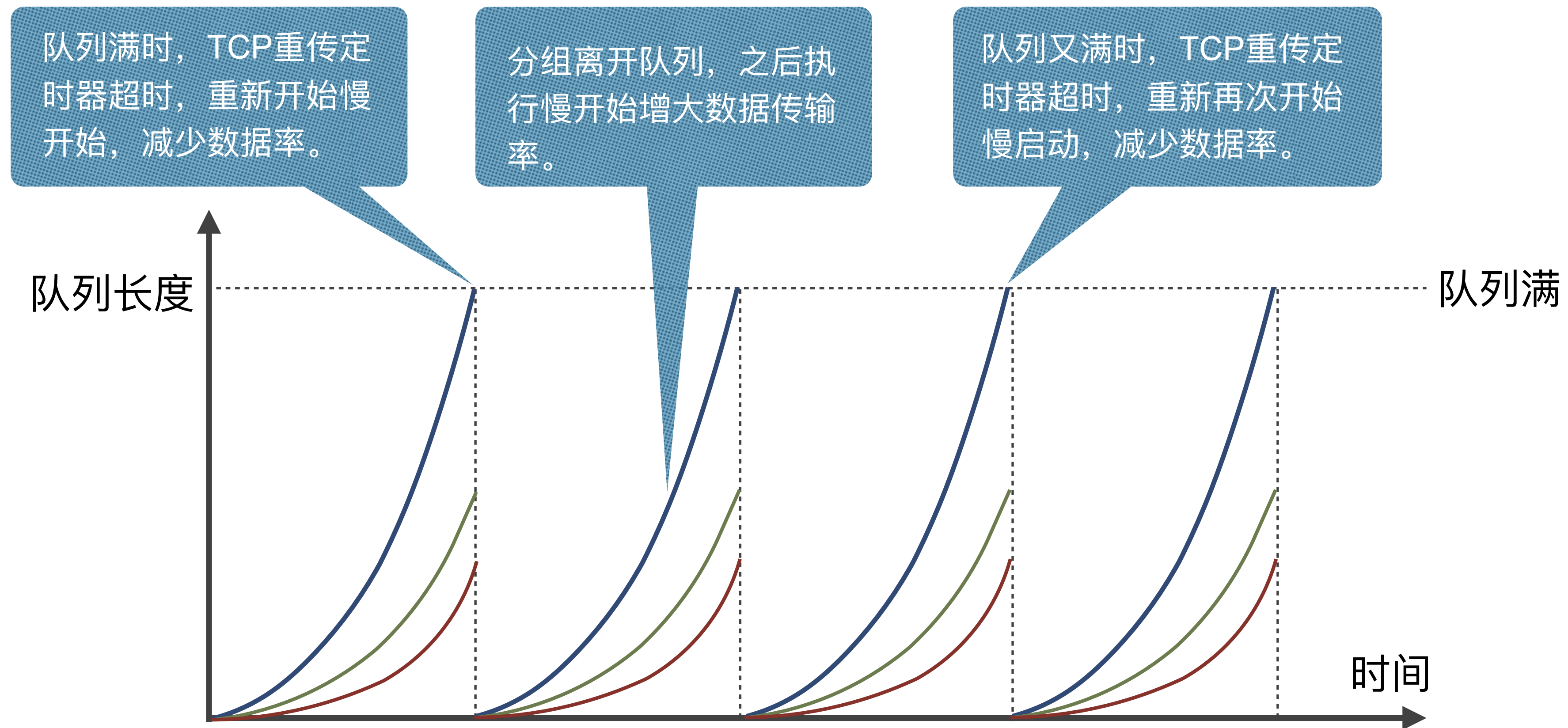
- TCP的拥塞控制与路由器丢弃分组的策略相关。
- 路由器若采用“**先入先出**”、队列满丢弃后续到达的分组，称“**尾部丢弃策略**”。

- 这种丢弃策略会导致“**全局同步**”：
  - 路由器同一方向来源的TCP连接中的分组被丢弃，这一方向上的所有TCP连接同一时间进入慢开始状态，**通信量突然下降**，正常后，**通信量突然增大**。





# 全局同步

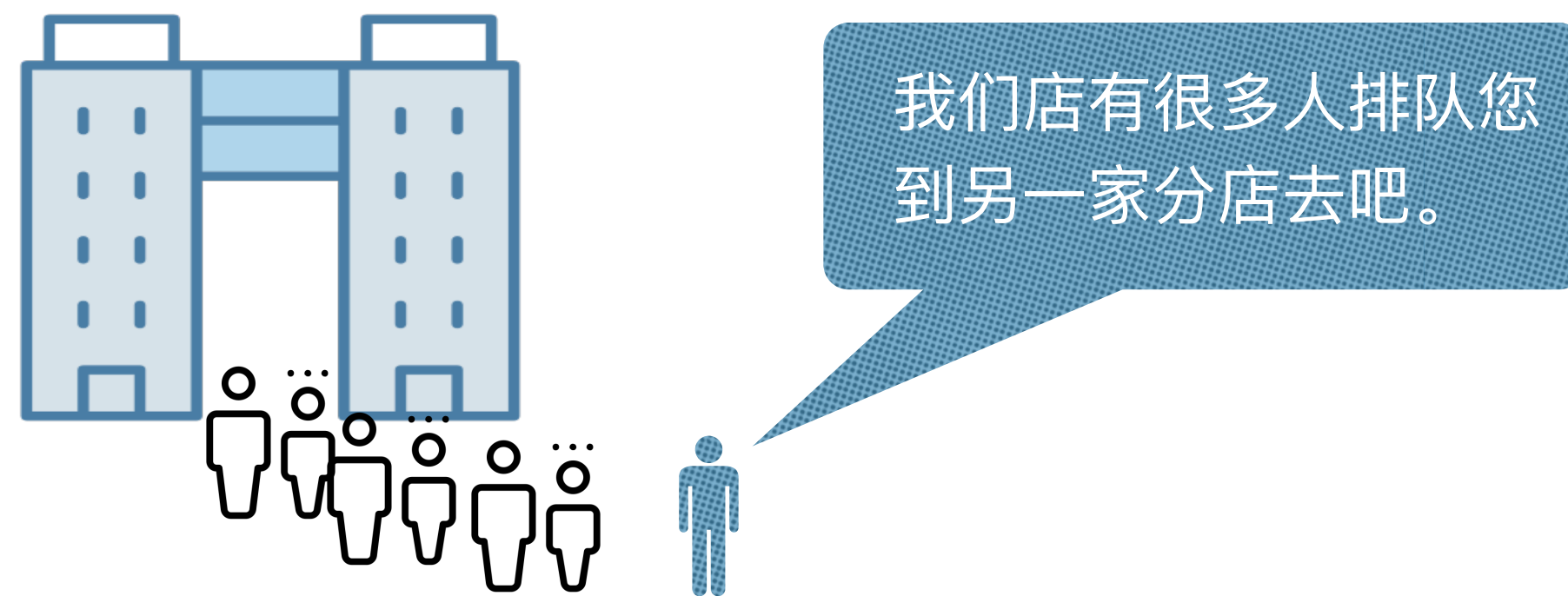


分组丢弃使发送方出现超时重传，使多个 TCP 连接同时进入拥塞控制的慢开始状态。

# 主动队列管理AQM

- 运输层
  - 分组管理策略
    - 主动队列管理

- 不要等到路由器的队列长度已经达到最大值时才不得不丢弃后面到达的分组，在队列长度达到某个值得警惕的数值时（即当网络拥塞有了某些拥塞征兆时），就主动丢弃到达的分组。
- AQM 有不同实现方法，曾流行多年的是随机早期检测 RED (Random Early Detection)。



# 主动队列管理AQM

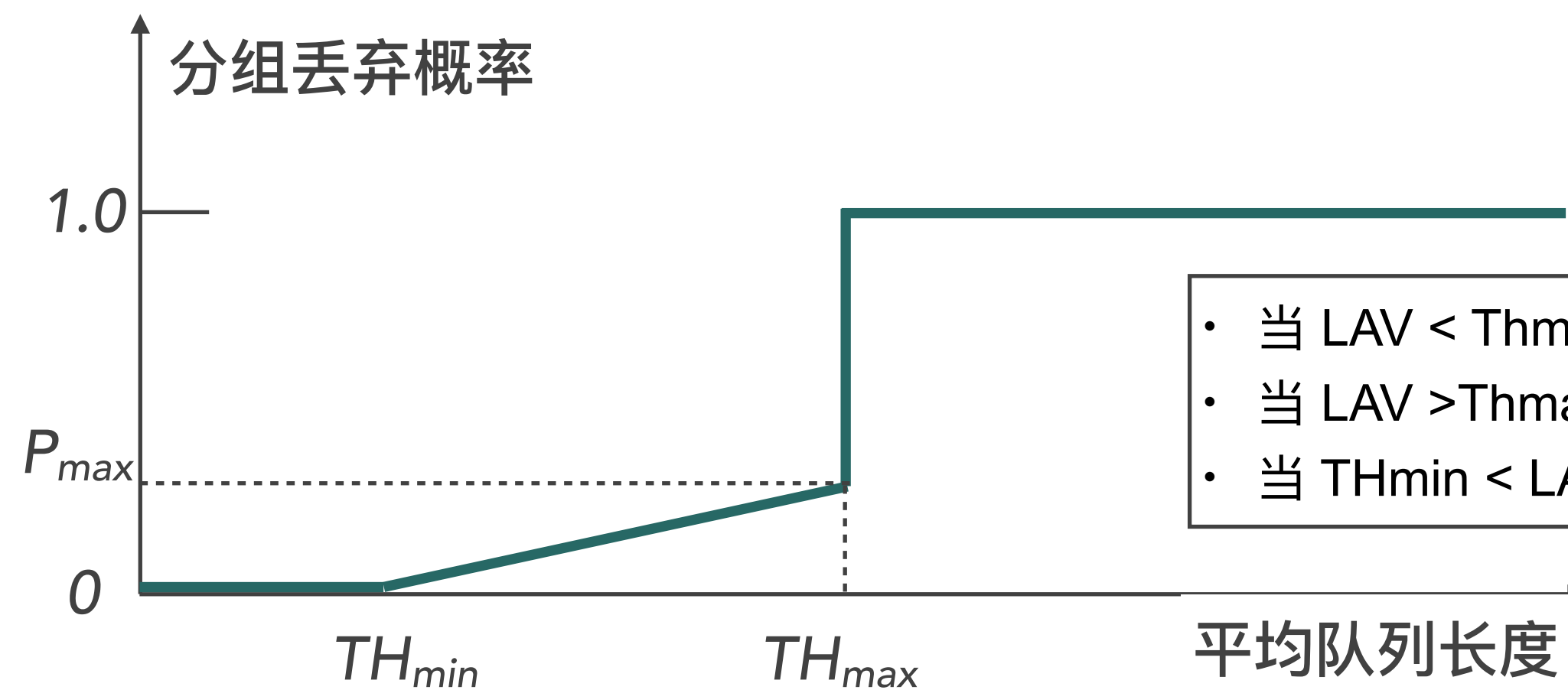
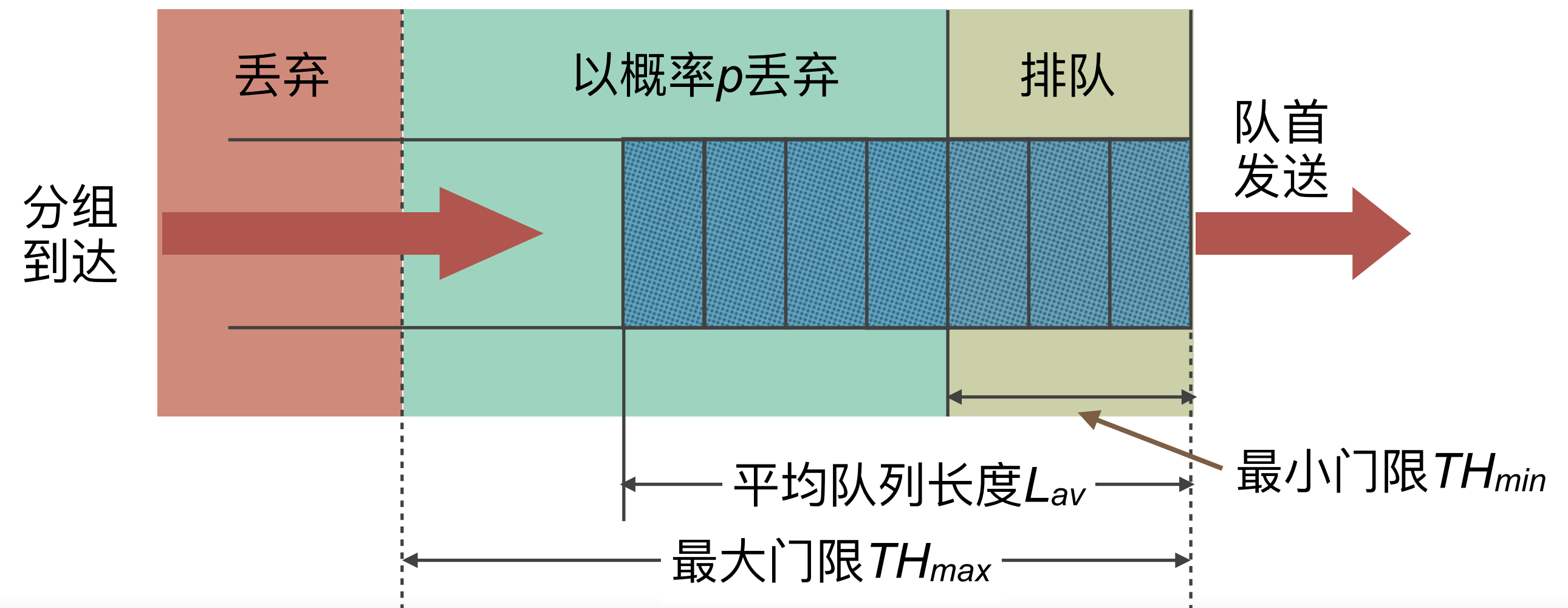
- 运输层
- 分组管理策略
- 主动队列管理

- 路由器维持两个参数：队列长度最小门限  $TH_{min}$  和最大门限  $TH_{max}$ 。
- 分组到达后计算平均队列长度  $LAV$ ：
  - 平均队列长度小于最小门限  $TH_{min}$ ，将新到达的分组放入队列进行排队；
  - 平均队列长度超过最大门限  $TH_{max}$ ，将新到达的分组丢弃；
  - 平均队列长度在最小门限  $TH_{min}$  和最大门限  $TH_{max}$  之间，按照概率  $p$  丢弃新到达的分组。



# 主动队列管理AQM

- 运输层
- 分组管理策略
- 主动队列管理



- 当  $LAV < Thmin$  时, 丢弃概率  $p = 0$ ;
- 当  $LAV > Thmax$  时, 丢弃概率  $p = 1$ ;
- 当  $THmin < LAV < THmax$  时,  $0 < p < 1$ 。

# 小结

- 运输层
  - 分组管理策略
  - 主动队列管理

- 路由器队列管理“先入先出”FIFO规则。
- 尾部丢弃策略。
- 全局同步。
- 主动队列管理AQM。
- 随机早期检测RED。

# TCP 的运输连接管理

- 运输层
  - 连接管理
  - 连接解决的问题
  - 三报文建立连接
  - 连接建立实例
  - 为什么是三报文
  - 四报文释放连接
  - 保活计时器
  - TCP有限状态机

- 运输连接的管理就是使运输连接的建立和释放都能正常地进行，如何打招呼？如何说再见？
  - TCP 的连接建立；
  - TCP 的连接释放；
  - TCP 的有限状态机。

# TCP连接的三个阶段

- 运输层
  - 连接管理
  - 连接解决的问题
  - 三报文建立连接
  - 连接建立实例
  - 为什么是三报文
  - 四报文释放连接
  - 保活计时器
  - TCP有限状态机

- 连接建立：交流之前打招呼。
- 数据传送：信息交流。
- 连接释放：交流完成说再见。



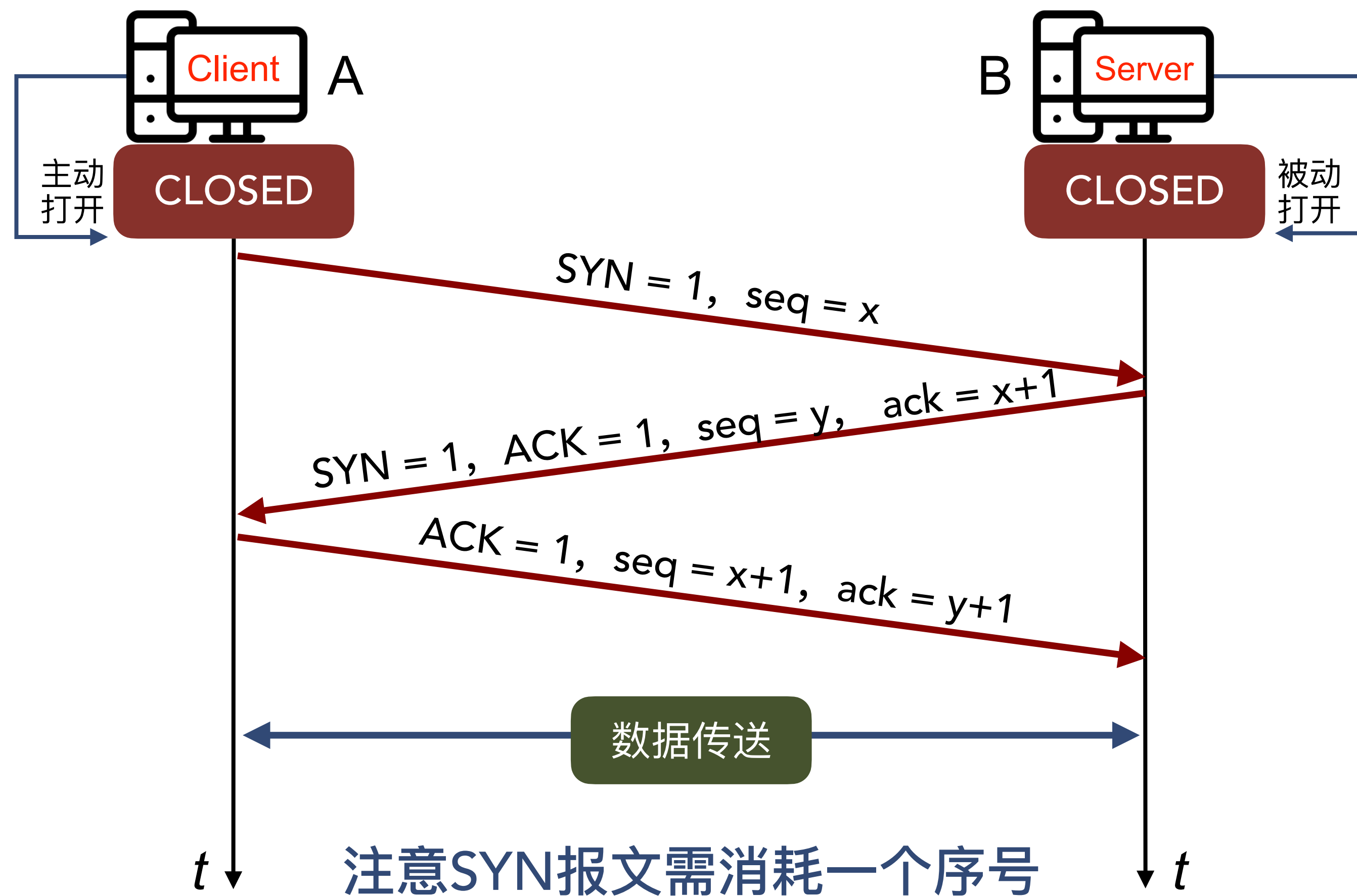
# TCP 连接建立过程中要解决的三个问题

- 运输层
  - 连接管理
  - 连接解决的问题
  - 三报文建立连接
  - 连接建立实例
  - 为什么是三报文
  - 四报文释放连接
  - 保活计时器
  - TCP有限状态机

- 要使每一方能够确知对方的存在。
- 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以及服务质量等）。
- 能够对运输实体资源（如缓存大小、连接表中的项目等）进行分配。
- TCP连接的建立采用客户服务器方式：
  - 主动发起连接建立的应用进程叫做客户(client)；
  - 被动等待连接建立的应用进程叫做服务器(server)。

类似的例子购物前与卖家交流：发什么物流？有什么优惠？是否是赠品？等等。

# TCP 的连接建立：三次握手建立连接



## 第一次握手:

A 的 TCP 向 B 发出连接请求报文段：其首部中的  $SYN = 1$ ,  $seq = x$ , 表明传送数据时的第一个数据字节的序号是  $x$ 。

## 第二次握手:

B 收到连接请求报文段后，如同意，发回确认。确认报文段中  $SYN = 1$ ,  $ACK = 1$ , 其确认号  $ack = x + 1$ , 自己的序号  $seq = y$ 。

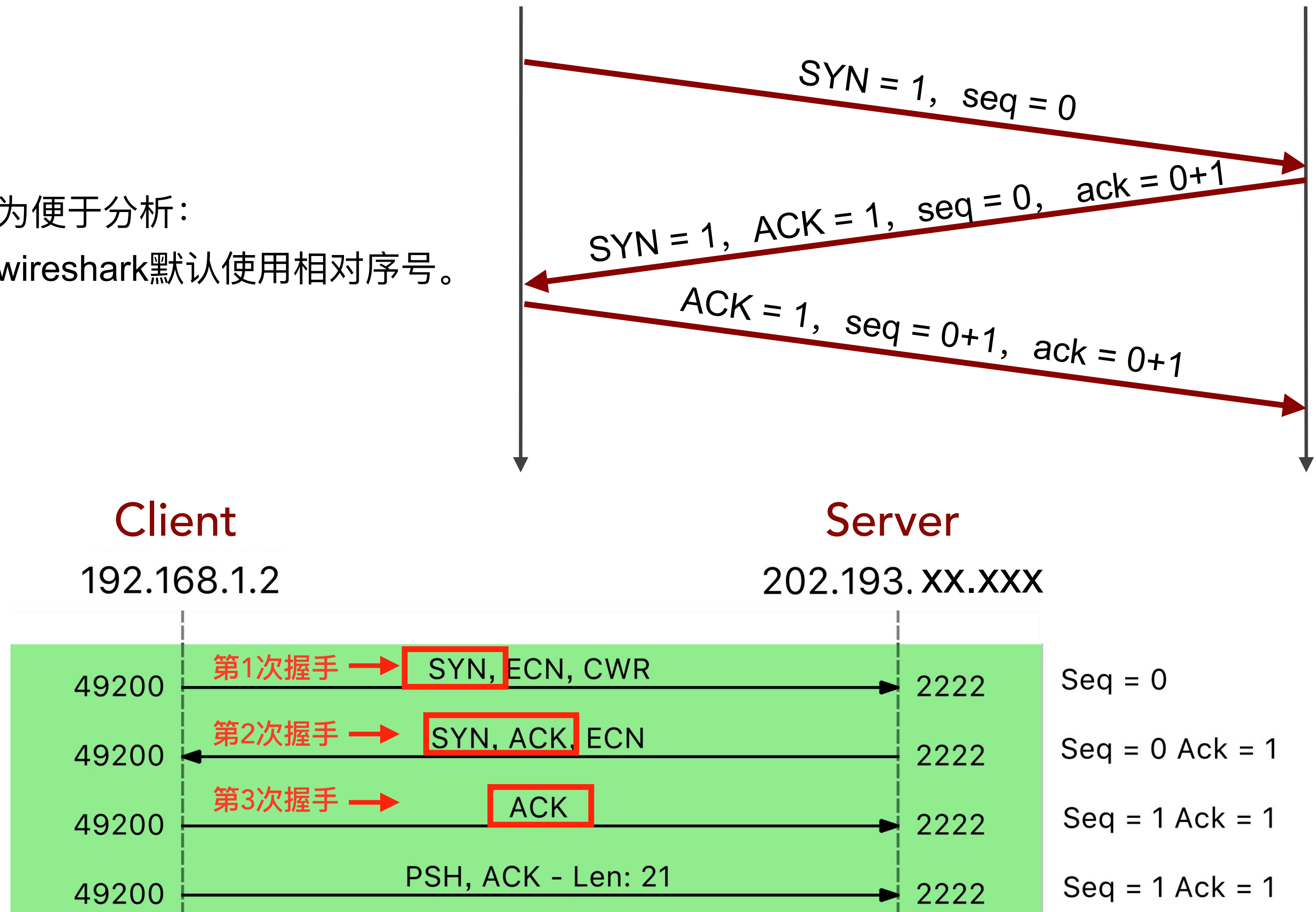
## 第三次握手:

A 收到同意报文后，向 B 给出确认， $ACK = 1$ , 确认号  $ack = y + 1$ 。A 的 TCP 通知上层应用进程，连接已经建立。

# 三次握手建立连接实例

- 运输层
  - 连接管理
  - 连接解决的问题
  - 三报文建立连接
  - 连接建立实例
  - 为什么是三报文
  - 四报文释放连接
  - 保活计时器
  - TCP有限状态机

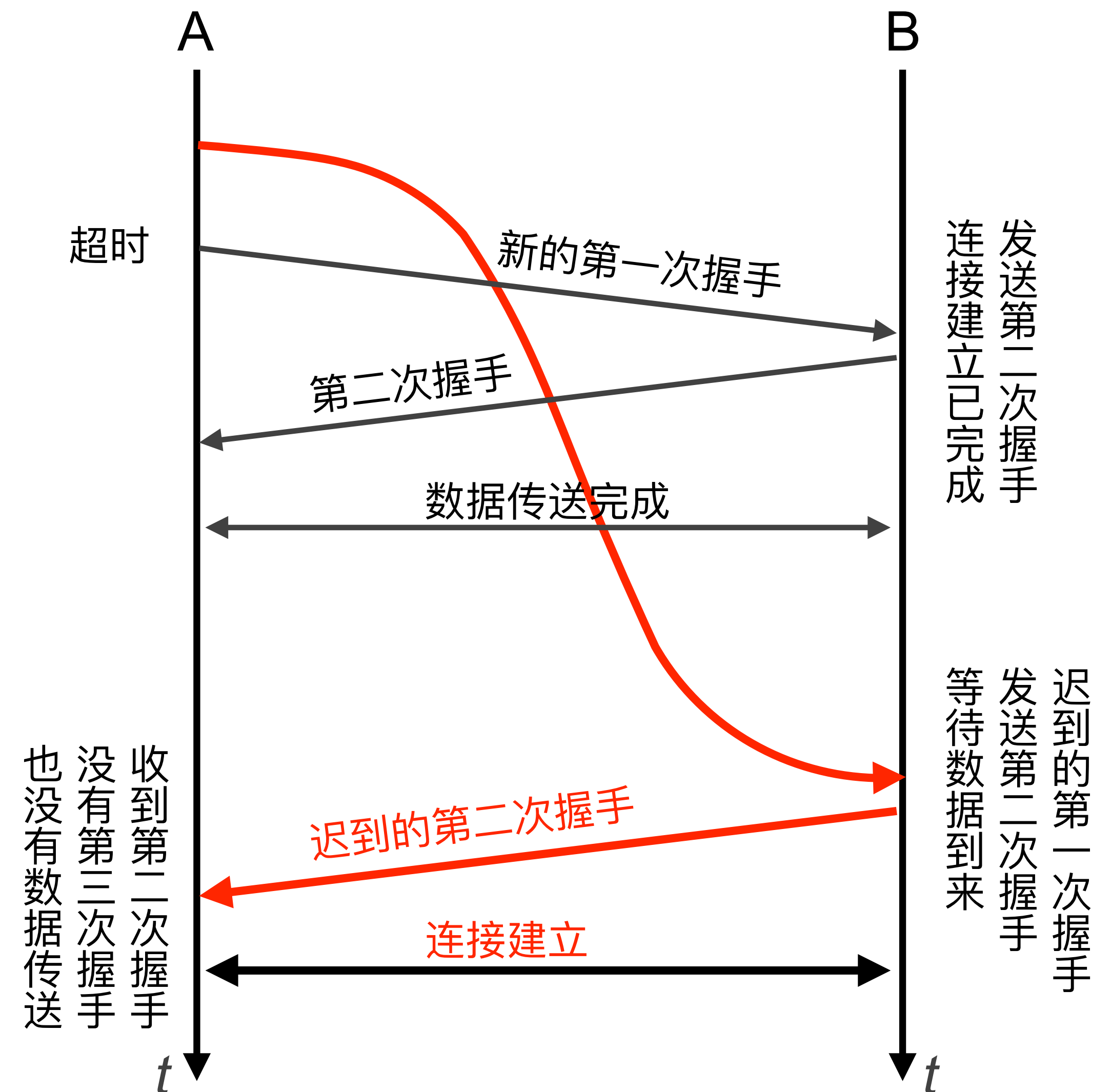
为便于分析：  
wireshark默认使用相对序号。



# 为什么需要三次握手建立连接?

- 二次握手建立连接的问题:
  - 第1次握手迟到, 服务器 B 等数据到来, 客户没有数据要传送。

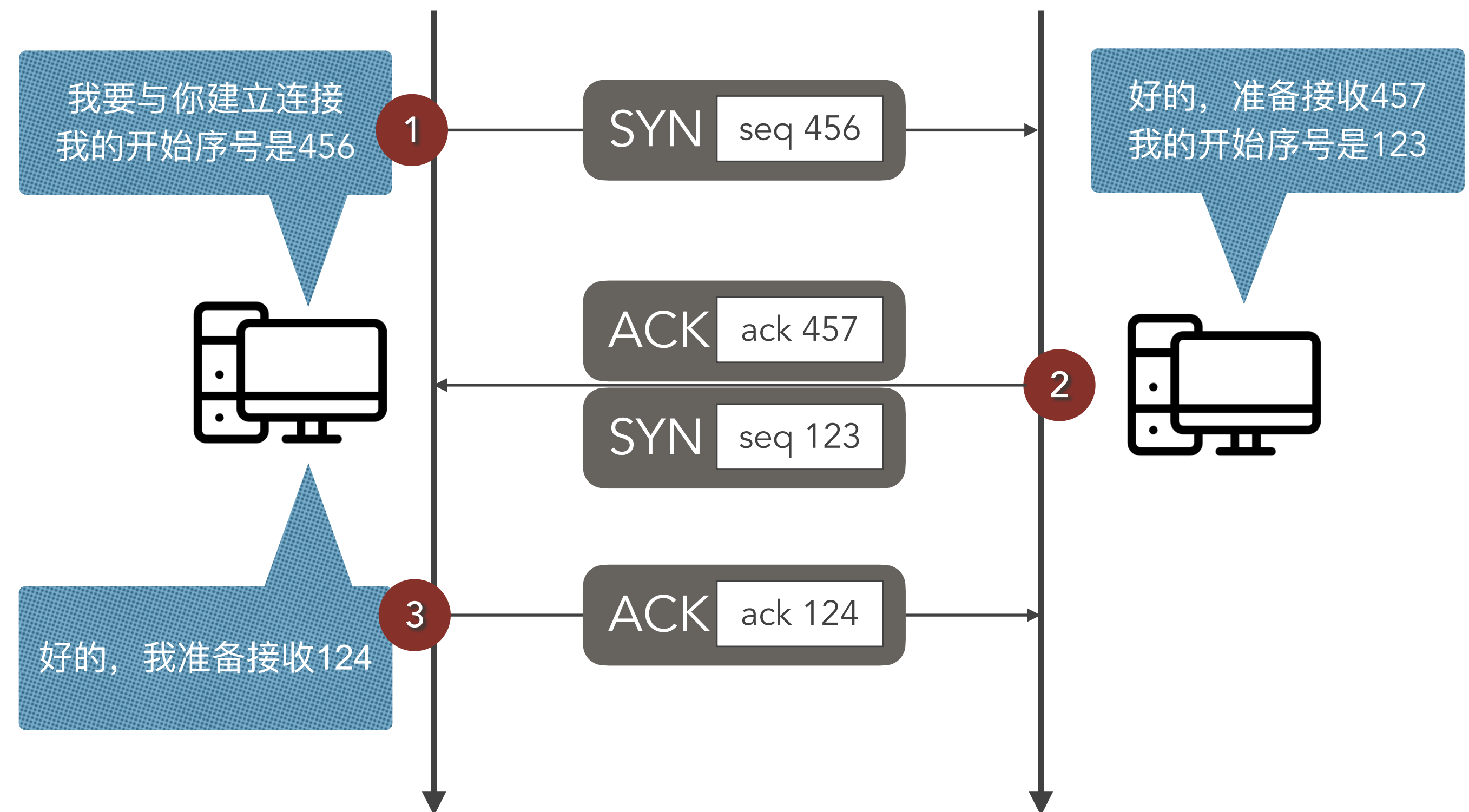
- 正常情况:
  - 同学: 老师我有问题问您。
  - 老师: 好的, 你的问题是?
  - 同学问, 老师答。





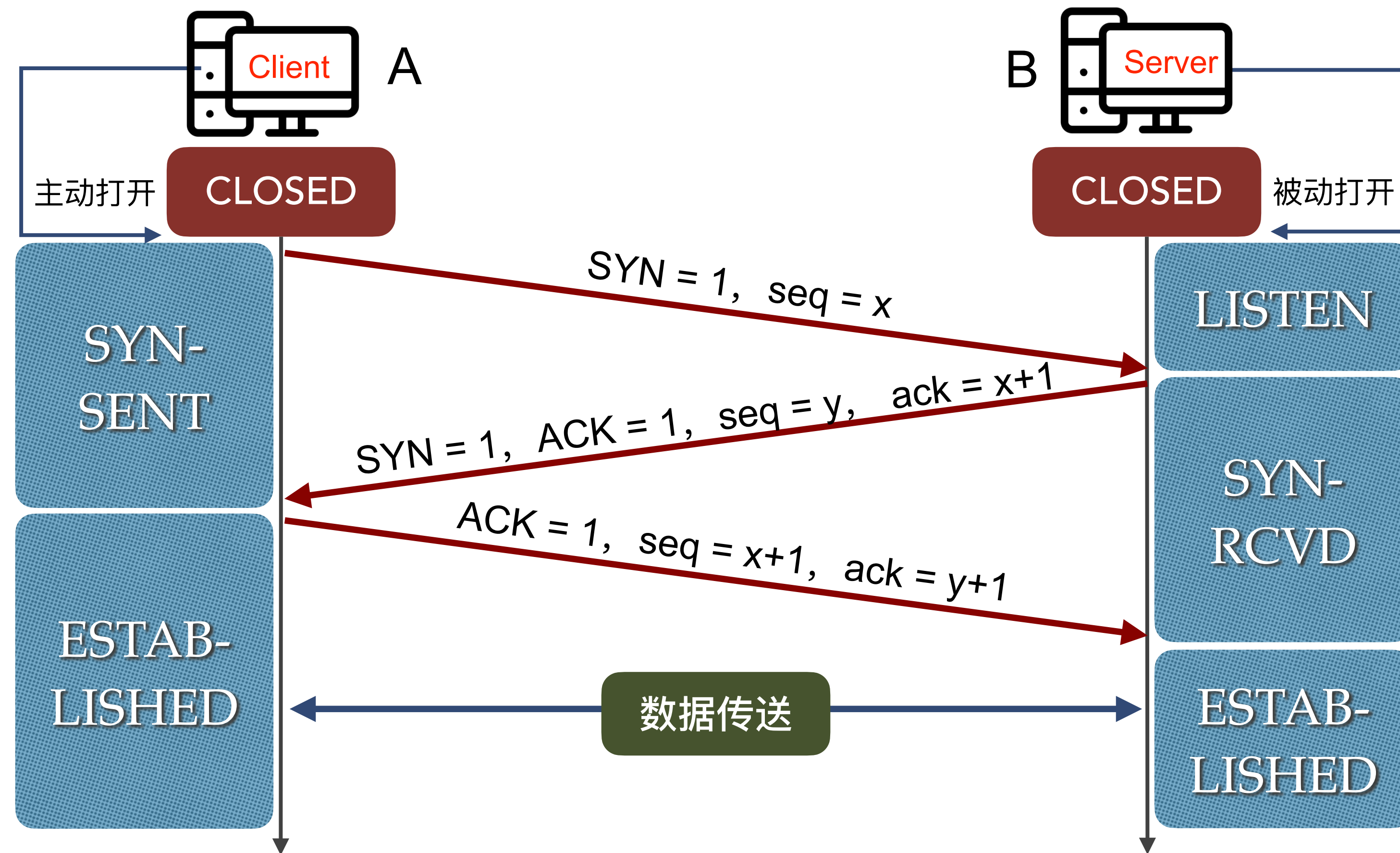
# 为什么需要三次握手建立连接？

- 运输层
  - 连接管理
  - 连接解决的问题
  - 三报文建立连接
  - 连接建立实例
  - 为什么是三报文
  - 四报文释放连接
  - 保活计时器
  - TCP有限状态机



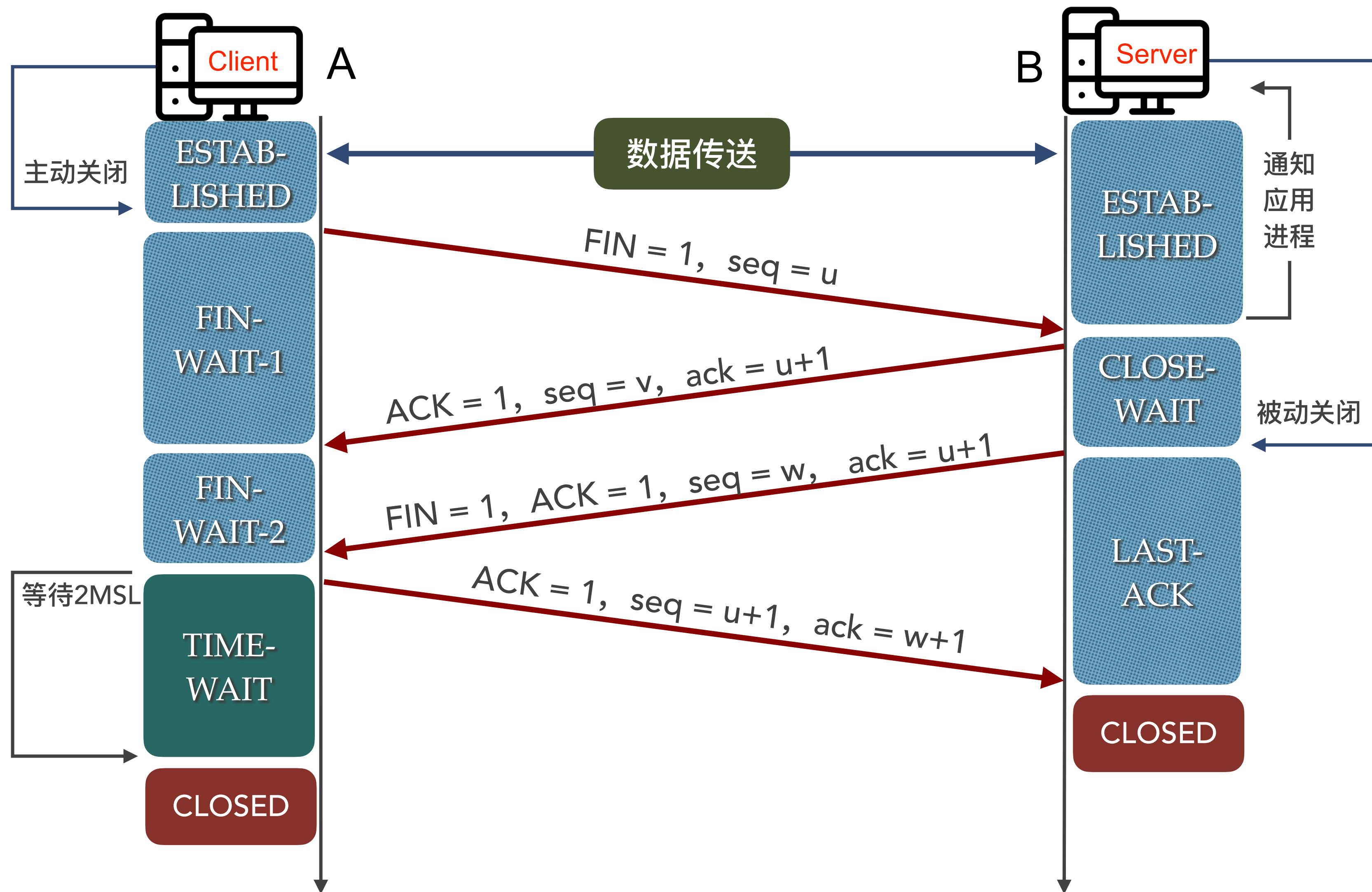
- 可靠性要求之一（确认、序号、重传）：**起始数据字节编号协商。**

# 采用三报文握手建立 TCP 连接的各状态





# TCP 的连接释放：四次报文握手





# A 为什么必须等待 2MSL 的时间后进入CLOSED

- 运输层
  - 连接管理
  - 连接解决的问题
  - 三报文建立连接
  - 连接建立实例
  - 为什么是三报文
  - 四报文释放连接
  - 保活计时器
  - TCP有限状态机

- 如果Client直接进入CLOSED状态, Server有可能没有收到Client最后回复的ACK, Server超时之后继续发送FIN。但Client已经进入CLOSED状态了, 服务器重发的FIN找不到对应的连接, Server就会收到RST而不是ACK:
  - 这种情况不会造成数据丢失, 但导致TCP协议不符合可靠连接的要求;
  - Client不是直接进入CLOSED状态, 而是要保持TIME\_WAIT, 当再次收到FIN的时候, 能够保证对方收到ACK, 最后正确地关闭连接。

保证TCP协议的全双工连接能够可靠关闭

# A 为什么必须等待 2MSL 的时间后进入CLOSED

- 运输层
  - 连接管理
  - 连接解决的问题
  - 三报文建立连接
  - 连接建立实例
  - 为什么是三报文
  - 四报文释放连接
  - 保活计时器
  - TCP有限状态机

- 如果Client直接进入CLOSED状态之后，又向Server发起一个  
新连接，有可能新连接和老连接的端口号是相同的。
- 如果前一次连接的某些数据仍然滞留在网络中，这些数据在建立新连接之后才到达Server，TCP协议就认为数据是属于新连接的，这样就和新连接的数据包发生混淆。
- Client在TIME\_WAIT状态等待2倍MSL，这样可以保证本次连接的所有数据都从网络中消失（Server与Client的连接已经释放，这期间Server收到的TCP报文段，直接丢弃）。

保证这次连接的重复数据从网络中消失

# 保活计时器

- 运输层
  - 连接管理
  - 连接解决的问题
  - 三报文建立连接
  - 连接建立实例
  - 为什么是三报文
  - 四报文释放连接
  - 保活计时器
  - TCP有限状态机

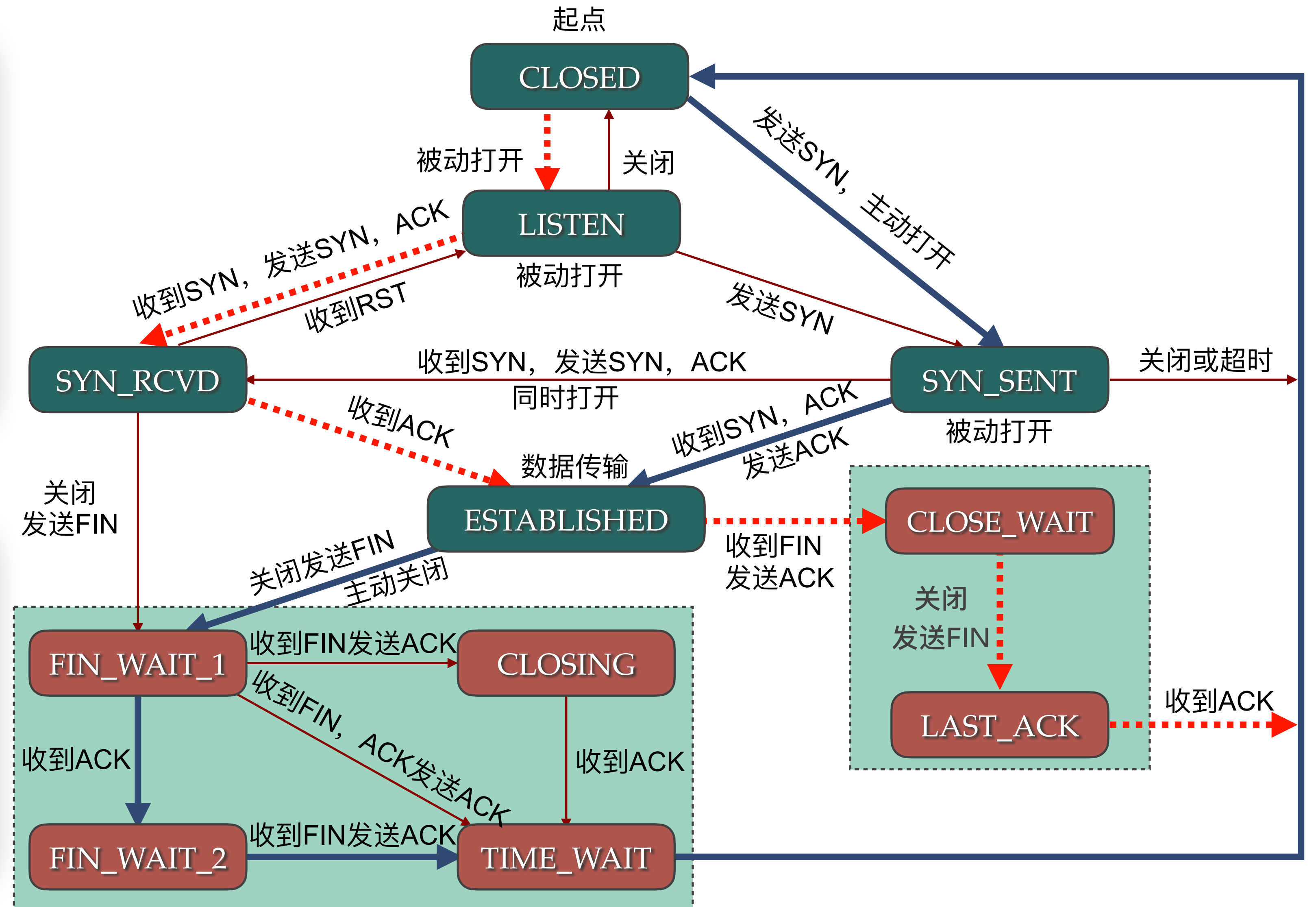
- 用来防止在TCP连接出现长时期的空闲。
- 保活计时器 通常设置为2小时。若服务器过了2小时还没有收到客户的信息，它就发送探测报文段。若发送了10个探测报文段（每一个相隔75秒）还没有响应，就假定客户出了故障，因而就终止该连接。

**TCP四个计时器：**超时重传计时器、坚持计时器（0窗口报文探测计时器）、时间等待计时器（2MSL关闭连接）、保活计时器

# TCP有限状态机

- **粗实线箭头**表示对客户进程的正常变迁。
- **粗虚线箭头**表示对服务器进程的正常变迁。
- **细线箭头**表示异常变迁。

- 箭头旁边的字，表明引起这种**变迁的原因**，或表明发生状态变迁后又出现什么动作。



# 异常状态迁移

- 运输层
  - 连接管理
  - 连接解决的问题
  - 三报文建立连接
  - 连接建立实例
  - 为什么是三报文
  - 四报文释放连接
  - 保活计时器
  - TCP有限状态机

- LISTEN->SYN\_SENT, 服务器主动建立打开连接。
- SYN\_SENT->SYN\_RCVD, 在SYN\_SENT状态, 收到SYN, 需要发送SYN+ACK数据报并进入到SYN\_RCVD态, 准备进入ESTABLISHED状态。
- SYN\_SENT->CLOSED, 发送超时返回到CLOSED状态。
- SYN\_RCVD->LISTEN, 收到RST包, 返回LISTEN状态。
- SYN\_RCVD->FIN\_WAIT\_1, 可以不经过ESTABLISHED状态, 直接跳转到FIN\_WAIT\_1状态等待关闭。



# 小结

- 运输层
  - 连接管理
  - 连接解决的问题
  - 三报文建立连接
  - 连接建立实例
  - 为什么是三报文
  - 四报文释放连接
  - 保活计时器
  - TCP有限状态机

- 为什么需要连接管理：
  - TCP 的连接建立；
  - TCP 的连接释放；
  - TCP有限状态机。