# 一. 概述

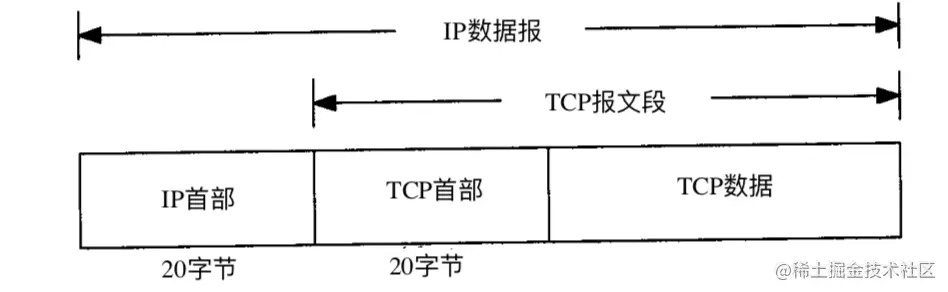
**1. tcp连接的特点**

* 提供面向连接的，可靠的字节流服务
* 为上层应用层提供服务，不关心具体传输的内容是什么，也不知道是二进制流，还是ascii字符。

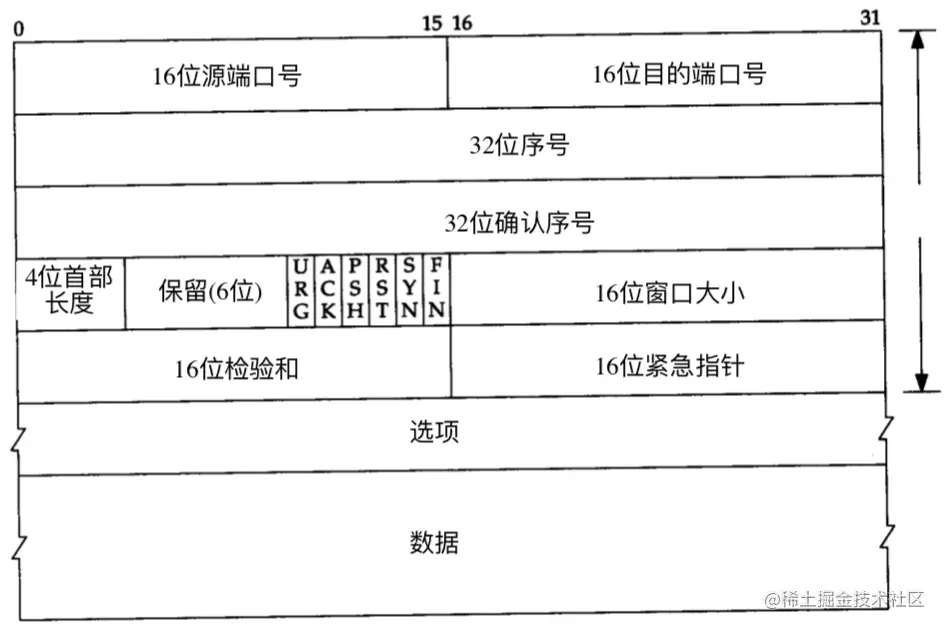
**2. tcp的可靠性如何保证**

* 分块传送：数据被分割成**最合适的**数据块（UDP的数据报长度不变）
* 等待确认：通过定时器等待接收端发送确认请求，收不到确认则重发
* 确认回复：收到确认后发送确认回复(不是立即发送，通常推迟几分之一秒)
* 数据校验：保持首部和数据的校验和，检测数据传输过程有无变化
* 乱序排序：接收端能重排序数据，以正确的顺序交给应用端
* 重复丢弃：接收端能丢弃重复的数据包
* 流量缓冲：两端有固定大小的缓冲区（滑动窗口），防止速度不匹配丢数据

**3. tcp的首部格式**

**3.1 宏观位置**

* 从应用层->传输层->网络层->链路层，每经过一次都会在报文中增加相应的首部。[参考之前的文章http协议](https://juejin.cn/post/6844903683424010247)
* TCP数据被封装在IP数据报中

**3.2 首部格式**

* tcp首部数据通常包含20个字节（不包括任选字段）
* 第1-2两个字节：源端口号
* 第3-4两个字节：目的端口号

源端口号+ip首部中的源ip地址+目的端口号+ip首部中的目的ip地址，唯一的确定了一个tcp连接。对应编码级别的socket。

* 第5-8四个字节：32位序号。tcp提供全双工服务，两端都有各自的序号。**编号：解决网络包乱序的问题**

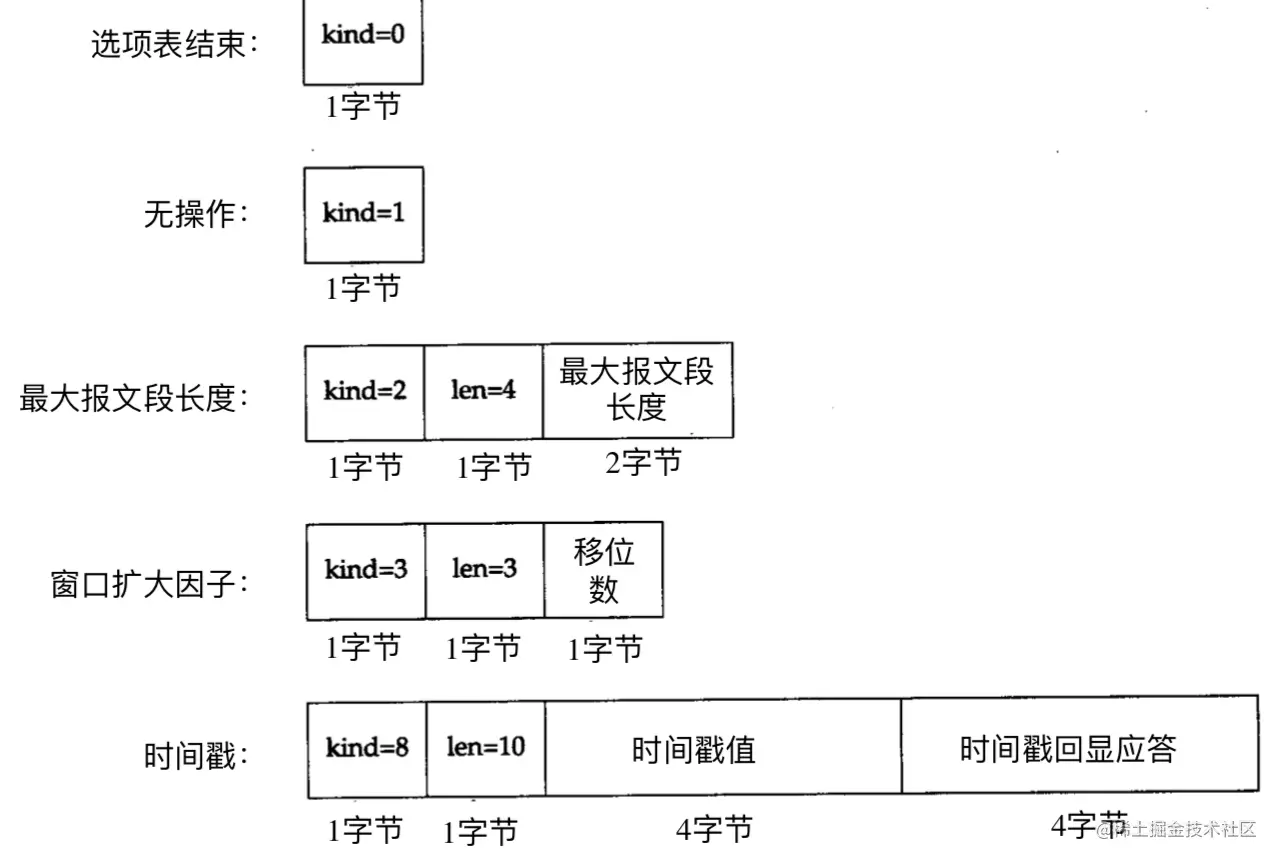
序号如何生成：不能是固定写死的，否则断网重连时序号重复使用会乱套。tcp基于时钟生成一个序号，每4微秒加一，到2^32-1时又从0开始

* 第9-12四个字节：32位确认序列号。上次成功收到数据字节序号加1，ack为1才有效。**确认号：解决丢包的问题**
* 第13位字节：首部长度。因为任选字段长度可变
* 后面6bite：保留
* 随后6bite：标识位。**控制各种状态**
* 第15-16两个字节：窗口大小。接收端期望接收的字节数。**解决流量控制的问题**
* 第17-18两个字节：校验和。由发送端计算和存储，由接收端校验。**解决数据正确性问题**
* 第19-20两个字节：紧急指针

**3.3 标识位说明**

* URG：为1时，表示紧急指针有效
* ACK：确认标识，连接建立成功后，总为1。为1时确认号有效
* PSH：接收方应尽快把这个报文交给应用层
* RST：复位标识，重建连接
* SYN：建立新连接时，该位为0
* FIN：关闭连接标识

**3.4 tcp选项格式**



* 每个选项开始是1字节kind字段，说明选项的类型
* kind为0和1的选项，只占一个字节
* 其他kind后有一字节len，表示该选项总长度（包括kind和len）
* kind为11，12，13表示tcp事务

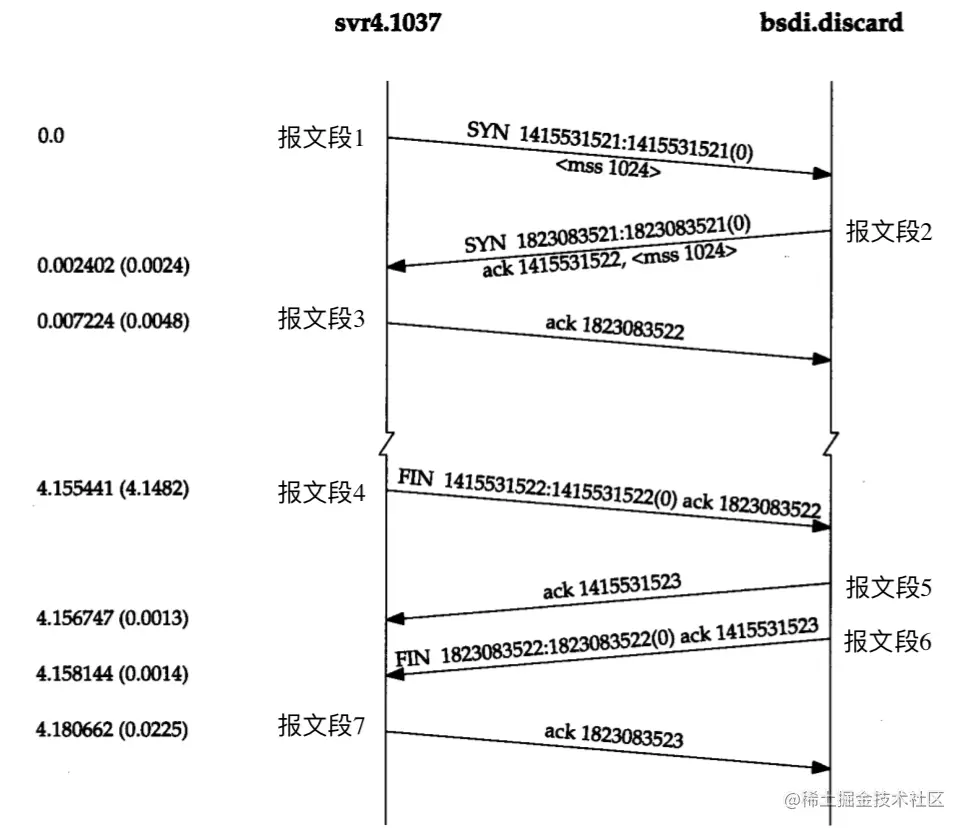
**3.5 MSS 最长报文大小**

* 最常见的可选字段
* MSS只能出现在SYN时传过来（第一次握手和第二次握手时）
* 指明本端能接收的最大长度的报文段
* 建立连接时，双方都要发送MSS
* 如果不发送，默认为536字节（这个值的来源在于 IPv4 有一个最小重组缓冲区大小，其值为 576 字节，是 IPv4 的任何实现都必须保证支持的最小 IP 数据报大小，IPv6 对应的值为 1500 字节。从运输层到 IP 层，PDU 增加了 IP 首部，20字节，因此 TCP 包为 556 字节，再去掉首部 20 字节，即为最小的 MSS，即 536 字节。）

**二. 连接的建立与释放**

**1. 连接建立的“三次握手”**

**1.1 三次握手流程**



* 客户端发送SYN，表明要向服务器建立连接。同时带上序列号ISN
* 服务器返回ACK（序号为客户端序列号+1）作为确认。同时发送SYN作为应答（SYN的序列号为服务端唯一的序号）
* 客户端发送ACK确认收到回复（序列号为服务端序列号+1）

**1.2 为什么是三次握手**

* tcp连接是全双工的，数据在两个方向上能同时传递。
* 所以要**确保双方，同时能发数据和收数据**
* 第一次握手：证明了发送方能发数据
* 第二次握手：ack确保了接收方能收数据，syn确保了接收方能发数据
* 第三次握手：确保了发送方能收数据
* 实际上是四个维度的信息交换，不过中间两步合并为一次握手了。
* 四次握手浪费，两次握手不能保证“双方同时具备收发功能”

**2. 连接关闭的“四次挥手”**

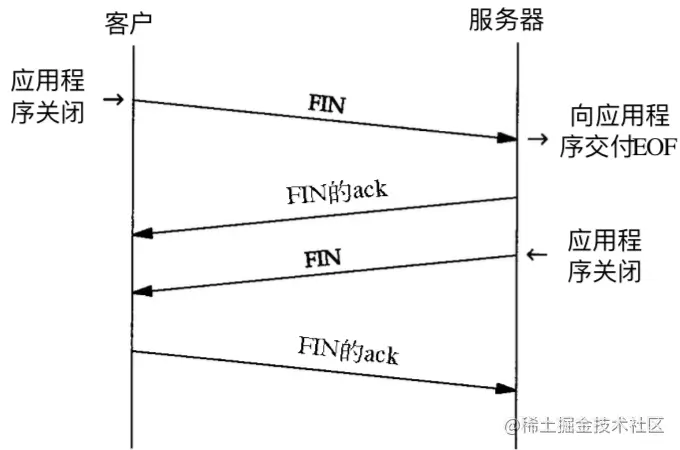
**2.1 为什么是四次挥手**

* 因为tcp连接是全双工的，数据在两个方向上能同时传递。
* 同时tcp支持半关闭（发送一方结束发送还能接收数据的功能）。
* 因此每个方向都要单独关闭，且收到关系通知需要发送确认回复

**2.2 为什么要支持半关闭**

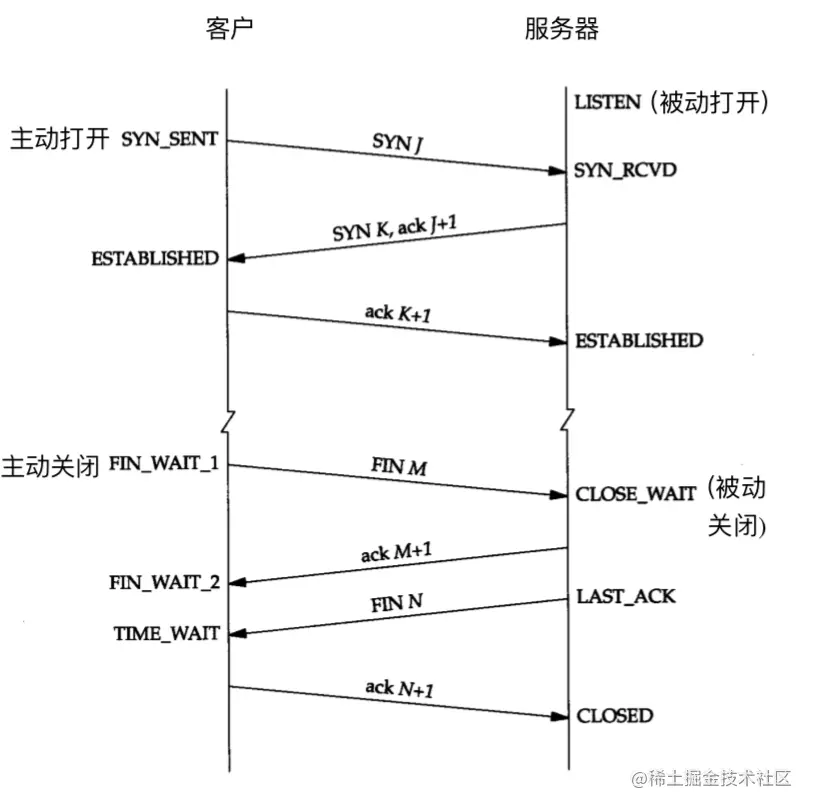
* 客户端需要通知服务端，它的数据已经传输完毕
* 同时仍要接收来自服务端的数据
* 使用半关闭的单连接效率要比使用两个tcp连接更好

**2.3 四次握手流程**



* 主动关闭的一方发送FIN，表示要单方面关闭数据的传输
* 服务端收到FIN后，发送一个ACK作为确认（序列号为收到的序列号+1）
* 等服务器数据传输完毕，也发送一个FIN标识，表示关闭这个方向的数据传输
* 客户端回复ACK以确认回复

**3. 连接和关闭对应的状态**



**3.1 状态说明**

* 服务端等待客户端连接时，处于Listen监听状态
* 客户端主动打开请求，发送SYN时处于SYN\_SENT发送状态
* 客户端收到syn和ack，并回复ack时，处与Established状态等待发送报文
* 服务端收到ack确认后，也处于Established状态等待发送报文
* 客户端发送fin后，处于fin\_wait\_1状态
* 服务端收到fin并发送ack时，处于close\_wait状态
* 客户端收到ack确认后，处于fin\_wait\_2状态
* 服务端发送fin后，处于last\_ack状态
* 客户端收到fin后发送ack，处于time\_wait状态
* 服务端收到ack后，处于closed状态

**3.2 time\_wait状态**

* 也称为2MSL等待状态，MSL=Maximum Segment LifetIme，报文段最大生存时间，根据不同的tcp实现自行设定。常用值为30s，1min，2min。linux一般为30s。
* 主动关闭的一方发送最后一个ack所处的状态
* 这个状态必须维持2MSL等待时间

**3.2.1 为什么需要这么做？**

* 设想一个场景，最后这个ack丢失了，接收方没有收到
* 这时候接收方会重新发送fin给发送方
* 这个等待时间就是为了防止这种情况发生，让发送方重新发送ack
* 总结：预留足够的时间给接收端收ack。同时保证，这个连接不会和后续的连接乱套（有些路由器会缓存数据包）

**3.2.2 这么做的后果？**

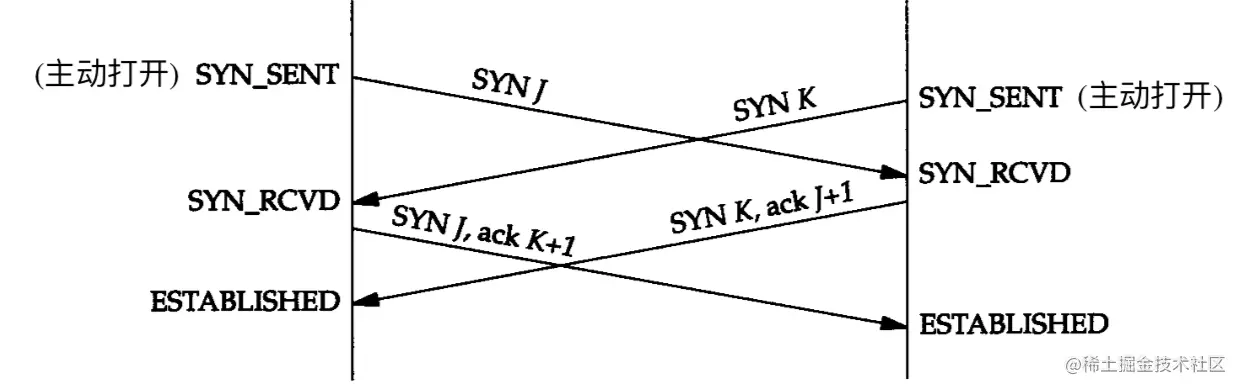
* 在这2MSL等待时间内，该连接（socket，ip+port）将不能被使用
* 很多时候linux上报too many open files，说端口不够用了，就需要检查一些代码里面是不是创建大量的socket连接，而这些socket连接并不是关闭后就立马释放的
* 客户端连接服务器的时候，一般不指定客户端的端口。因为客户端关闭然后立马启动，按照理论来说是会提示端口被占用。同样的道理，主动关闭服务器，2MSL时间内立马启动是会报端口被占用的错误
* 多并发的短连接情况下，会出现大量的Time\_wait状态。这两个参数可以解决问题，但是它违背了tcp协议，是有风险的。参数为：tcp\_tw\_reuse和tcp\_tw\_recycle
* 如果是服务端开发，可设置keep-alive，让客户端主动关闭连接解决这个问题

**4. 复位报文段**

一个报文段从源地址发往目的地址，只要出现错误，都会发出复位的报文段，首部字段的RST是用于“复位”的。这些错误包括以下情况

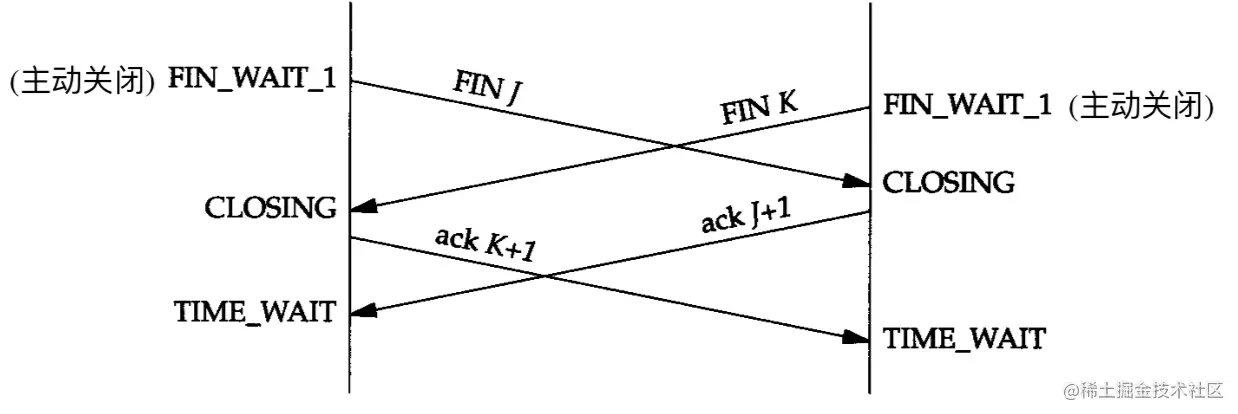
* 端口没有在监听
* 异常中止：通过发送RST而不是fin来中止连接

**5. 同时打开**



* 两个应用程序同时执行主动打开，称为“同时打开“
* 这种情况极少发生
* 两端同时发送SYN，同时进入SYN\_SENT状态
* 打开一条连接而不是两条
* 要进行四次报文交换过程，“四次握手”

**6. 同时关闭**



* 双方同时执行主动关闭
* 进行四次报文交换
* 状态和正常关闭不一样

**7. 服务器对于并发请求的处理**

* 正等待连接的一端有一个固定长度的队列（长度叫做“积压值”，大多数情况长度为5）
* 该队列中的连接为：已经完成了三次握手，但还没有被应用层接收（应用层需要等待最后一个ack收到后才知道这个连接）
* 应用层接收请求的连接，将从该队列中移除
* 当新的请求到来时，先判断队列情况来决定是否接收这个连接
* 积压值的含义：tcp监听的端点已经被tcp接收，但是等待应用层接收的最大值。与系统允许的最大连接数，服务器接收的最大并发数无关

**三. 数据的传输**

**1. tcp传输的数据分类**

* 成块数据传输：量大，报文段常常满
* 交互数据传输：量小，报文段为微小分组，大量微小分组，在广域网传输会增加拥堵的出现
* tcp处理的数据包括两类，有不同的特点，需要不同的传输技术

**2. 交互数据的传输技术**

**2.1 经受时延的确认**

* 概念：tcp收到数据时，并不立马发送ack确认，而是稍后发送
* 目的：将ack与需要沿该方向发送的数据一起发送，以减少开销
* 特点：接收方不必确认每一个收到的分组，ACk是累计的，它表示接收方已经正确收到了一直到确认序号-1的所有字节
* 延时时间：绝大多数为200ms。不能超过500ms

**2.2 Nagle算法**

* 解决什么问题：微小分组导致在广域网出现的拥堵问题
* 核心：减少了通过广域网传输的小分组数目
* 原理：要求一个tcp连接上最多只能有一个未被确认的未完成的分组，该分组的确认到达之前，不能发送其他分组。tcp收集这些分组，确认到来之前以一个分组的形式发出去
* 优点：自适应。确认到达的快，数据发送越快。确认慢，发送更少的组。
* 使用注意：局域网很少使用该算法。且有些特殊场景需要禁用该算法

**3. 成块数据的传输**

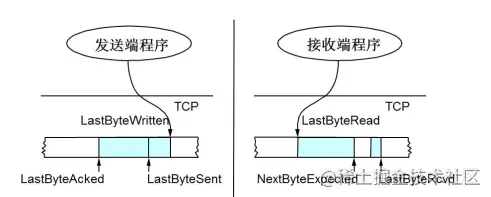
* 主要使用滑动窗口协议

**四. 滑动窗口协议**

**1. 概述**

* 解决了什么问题：发送方和接收方速率不匹配时，保证可靠传输和包乱序的问题
* 机制：接收方根据目前缓冲区大小，通知发送方目前能接收的最大值。发送方根据接收方的处理能力来发送数据。通过这种协调机制，防止接收端处理不过来。
* 窗口大小：接收方发给发送端的这个值称为窗口大小

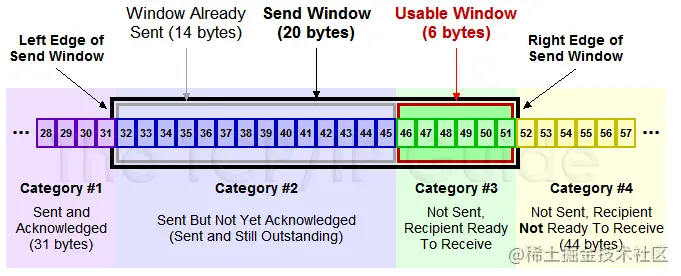
**2. tcp缓冲区的数据结构**



* 接收端：
  + LastByteRead: 缓冲区读取到的位置
  + NextByteExpected：收到的连续包的最后一个位置
  + LastByteRcvd：收到的包的最后一个位置
  + 中间空白区：数据没有到达
* 发送端：
  + LastByteAcked: 被接收端ack的位置，表示成功发送确认
  + LastByteSent：发出去了，还没有收到成功确认的Ack
  + LastByteWritten：上层应用正在写的地方

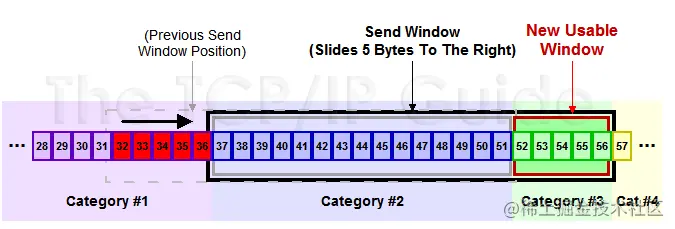
**3. 滑动窗口示意图**

**3.1 初始时示意图**



* 黑框表示滑动窗口
* #1表示收到ack确认的数据
* #2表示还没收到ack的数据
* #3表示在窗口中还没有发出的（接收方还有空间）
* #4窗口以外的数据（接收方没空间）

**3.2 滑动过程示意图**



* 收到36的ack，并发出46-51的字节

**4. 拥塞窗口**

* 解决什么问题：发送方发送速度过快，导致中转路由器拥堵的问题
* 机制：发送方增加一个拥塞窗口（cwnd），每次受到ack，窗口值加1。发送时，取拥塞窗口和接收方发来的窗口大小取最小值发送
* 起到发送方流量控制的作用

**5. 滑动窗口会引发的问题**

**5.1 零窗口**

* 如何发生： 接收端处理速度慢，发送端发送速度快。窗口大小慢慢被调为0
* 如何解决：ZWP技术。发送zwp包给接收方，让接收方ack他的窗口大小。

**5.2 糊涂窗口综合征**

* 如何发生：接收方太忙，取不完数据，导致发送方越来越小。最后只让发送方传几字节的数据。
* 缺点：数据比tcp和ip头小太多，网络利用率太低。
* 如何解决：避免对小的窗口大小做响应。
  + 发送端：前面说到的Nagle算法。
  + 接收端：窗口大小小于某个值，直接ack（0），阻止发送数据。窗口变大后再发。

**五. 超时与重传**

**1. 概述**

* tcp提供可靠的运输层，使用的方法是确认机制。
* 但是数据和确认都有可能丢失
* tcp通过在发送时设置定时器解决这种问题
* 定时器时间到了还没收到确认，就重传该数据

**2. tcp管理的定时器类型**

* 重传定时器：等待收到确认
* 坚持定时器：使窗口大小信息保持不断流动
* 保活定时器：检测空闲连接崩溃或重启
* 2MSL定时器：检测time\_wait状态

**3. 超时重传机制**

**3.1 背景**

* 接收端给发送端的Ack确认只会确认最后一个连续的包
* 比如发送1,2,3,4,5共五份数据，接收端收到1,2，于是回ack3，然后收到4（还没收到3），此时tcp不会跳过3直接确认4，否则发送端以为3也收到了。这时你能想到的方法是什么呢？tcp又是怎么处理的呢？

**3.1 被动等待的超时重传策略**

* 直观的方法是：接收方不做任何处理，等待发送方超时，然后重传。
  + 缺点：发送端不知道该重发3，还是重发3,4,5
* 如果发送方如果只发送3：节省宽度，但是慢
* 如果发送方如果发送3,4,5：快，但是浪费宽带
* 总之，都在被动等待超时，超时可能很长。所以tcp不采用此方法

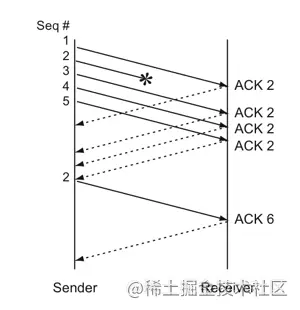
**3.2 主动的快速重传机制**

**3.2.1 概述**

* 名称为：Fast Retransmit
* 不以实际驱动，而以数据驱动重传

**3.2.2 实现原理**

* 如果包没有送达，就一直ack最后那个可能被丢的包
* 发送方连续收到3相同的ack，就重传。不用等待超时



* 图中发生1,2,3,4,5数据
* 数据1到达，发生ack2
* 数据2因为某些原因没有送到
* 后续收到3的时候，接收端并不是ack4，也不是等待。而是主动ack2
* 收到4,5同理，一直主动ack2
* 客户端收到三次ack2，就重传2
* 2收到后，结合之前收到的3,4,5，直接ack6

**3.2.3 快速重传的利弊**

* 解决了被动等待timeout的问题
* 无法解决重传之前的一个，还是所有的问题。
* 上面的例子中是重传2，还是重传2,3,4,5。因为并不清楚ack2是谁传回来的

**3.3 SACK方法**

**3.3.1 概述**

* 为了解决快速重传的缺点，一种更好的SACK重传策略被提出
* 基于快速重传，同时在tcp头里加了一个SACK的东西
* 解决了什么问题：客户端应该发送哪些超时包的问题

**3.3.2 实现原理**

* SACK记录一个数值范围，表示哪些数据收到了
* linux2.4后默认打开该功能，之前版本需要配置tcp-sack参数
* SACK只是一种辅助的方式，发送方不能完全依赖SACK。主要还是依赖ACK和timout

**3.3.3 Duplicate SACK(D-SACK)**

* 使用SACK标识的范围，还可以知道告知发送方，有哪些数据被重复接收了
* 可以让发送方知道：是发出去的包丢了，还是回来的ack包丢了

**4. 超时时间的确定**

**4.1 背景**

* 路由器和网络流量均会变化
* 所以超时时间肯定不能设置为一个固定值
* 超时长：重发慢，效率低，性能差
* 超时短：并没有丢就重发，导致网络拥塞，导致更多超时和更多重发
* tcp会追踪这些变化，并相应的动态改变超时时间（RTO）

**4.2 如何动态改变**

* 每次重传的时间间隔为上次的一倍，直到最大间隔为64s，称为“指数退避”
* 首次重传到最后放弃重传的时间间隔一般为9min
* 依赖以往的往返时间计算（RTT）动态的计算

**4.3 往返时间（RTT）的计算方法**

* 并不是简单的ack时间和发送时间的差值。因为有重传，网络阻塞等各种变化的因素。
* 而是通过采样多次数值，然后做估算
* tcp使用的方法有：
  + 被平滑的RTT估计器
  + 被平滑的均值偏差估计器

**4.4. 重传时间的具体计算**

* 计算往返时间（RTT），保存测量结果
* 通过测量结果维护一个被平滑的RTT估计器和被平滑的均值偏差估计器
* 根据这两个估计器计算下一次重传时间

**5. 超时重传引发的问题-拥塞**

**5.1 为什么重传会引发拥塞**

* 当网络延迟突然增加时，tcp会重传数据
* 但是过多的重传会导致网络负担加重，从而导致更大的延时和丢包，进入恶性循环
* 也就是tcp的拥塞问题

**5.2 解决拥塞-拥塞控制的算法**

* 慢启动：降低分组进入网络的传输速率
* 拥塞避免：处理丢失分组的算法
* 快速重传
* 快速恢复

**六. 其他定时器**

**1. 坚持定时器**

**1.1 坚持定时器存在的意义**

* 当窗口大小为0时，接收方会发送一个没有数据，只有窗口大小的ack
* 但是，如果这个ack丢失了会出现什么问题？双方可能因为等待而中止连接
* 坚持定时器周期性的向接收方查询窗口是否被增大。这些发出的报文段称为窗口探查

**1.2 坚持定时器启动时机**

* 发送方被通告接收方窗口大小为0时

**1.3 与超时重传的相同和不同**

* 相同：同样的重传时间间隔
* 不同：窗口探查从不放弃发送，直到窗口被打开或者进程被关闭。而超时重传到一定时间就放弃发送

**2. 保活定时器**

**2.1 保活定时器存在的意义**

* 当tcp上没有数据传输时，服务器如何检测到客户端是否还存活