TCP详解

简介

传输控制协议（TCP，Transmission Control Protocol）是为了在不可靠的互联网络上提供可靠的端到端字节流而专门设计的一个传输协议。

互联网络与单个网络有很大的不同，因为互联网络的不同部分可能有截然不同的拓扑结构、带宽、延迟、数据包大小和其他参数。**TCP的设计目标是能够动态地适应互联网络的这些特性，而且具备面对各种故障时的健壮性**。

每台支持TCP的机器都有一个TCP传输实体。**TCP实体可以是一个库过程、一个用户进程，或者内核的一部分**。在所有这些情形下，它管理TCP流，以及与IP层之间的接口。**TCP传输实体接受本地进程的用户数据流，将它们分割成不超过64KB（实际上去掉IP和TCP头，通常不超过1460数据字节）的分段，每个分段以单独的IP数据报形式发送。**当包含TCP数据的数据报到达一台机器时，它们被递交给TCP传输实体，TCP传输实体重构出原始的字节流。为简化起见，我们有时候仅仅用“TCP”来代表TCP传输实体（一段软件）或者TCP协议（一组规则）。根据上下文语义你应该能很消楚地推断出其实际含义。例如，在“用户将数据交给TCP”这句话中，很显然这里指的是TCP传输实体。

IP层并不保证数据报一定被正确地递交到接收方，也不指示数据报的发送速度有多快。正是TCP负责既要足够快地发送数据报，以便使用网络容量，但又不能引起**网络拥塞**：而且，TCP超时后，要重传没有递交的数据报。即使被正确

递交的数据报，也可能存在错序的问题，**这也是TCP的责任，它必须把接收到的数据报重新装配成正确的顺序**。简而言之**，TCP必须提供可靠性的良好性能**，这正是大多数用户所期望的而IP又没有提供的功能。

(源IP，源port，目的IP，目的port) 四元组唯一的标识了一个 TCP 连接，如果不考虑目的 ip 和 端口，在本机只有一个 IP 的情况下，一台主机最多只能有 65536 个 TCP 连接，如果考虑不同 IP 或是 目的地址的话，TCP 连接可以更多，但每一个网络 IO 都是一个特殊文件描述符，因此 TCP 连接数仍然限制于 文件描述符的最大数量与物理极限。

我们主要从 **TCP 报文格式**、**滑动窗口协议**、**超时重传机制**、**拥塞控制机制**、**连接过程**以及**粘包丢包**来分析TCP。

TCP报文格式

偏移	字节	0								1								2								3							
字节	比特	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	来源连接端口																目的连接端口															
4	32	序列号码																															
8	64	确认号码（当ACK设置）																															
12	96	数据偏移				保留 0 0 0			NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	窗口大小															
16	128	校验和																紧急指针（当URG设置）															
20 ...	160 ...	选项（如果数据偏移 > 5，需要在结尾添加0。） ...																															

TCP报文前 20 字节是固定的，选项部分可选的可以补充一些数据，但不管怎么样，TCP 总长度被要求必须是 4 字节的整数倍，这意味着可能会填充0以对齐。

- 源端口与目的端口：各占 2 字节。
- 序列号码：4字节，序列号是**循环使用**的，TCP 中每一个字节对应与一个序列号，这里的**序列号是该数据中第一个字节的序列号**。
- 确认号：4字节，确认号指示 **期望收到对方下一个报文段的第一个数据字节的序号**，例如，当对方发送了 0 - 100 序列字节给你，此时你期望收到第 101 个序列字节，此时便可以设置 ACK = 101。
- 数据偏移：4bit，这个字段指示数据的偏移位置，事实上它等价于 TCP 首部的总长度。数据偏移是基于**字(1字 = 4字节)**为单位的，数据偏移最大4 位，意味着 TCP 首部最大不允许超过 2^4-1字，即 15 字、60字节。
- 保留：3位，必须为 0。
- 标志位：9 位(旧版本只有 6 位，事实上重点也就是标黑的6位)
  - NS。ECN显式拥塞通知（Explicit Congestion Notification）是对TCP的扩展，定义于RFC 3540（2003）。ECN允许拥塞控制的端对端通知而避免丢包。ECN为一项可选功能，如果底层网络设施支持，则可能被启用ECN的两个端点使用。在ECN成功协商的情况下，ECN感知路由器可以在IP头中设置一个标记来代替丢弃数据包，以标明阻塞即将发生。数据包的接收端回应发送端的表示，降低其传输速率，就如同在往常中检测到包丢失那样。
  - CWR—Congestion Window Reduced，定义于RFC 3168（2001）。
  - ECE—ECN-Echo有两种意思，取决于SYN标志的值，定义于RFC 3168（2001）。
  - **URG—为1表示高优先级数据包，紧急指针字段有效。当该字段为 1 时，应用程序可以将紧急数据插入到数据字节流的最前方，以便接收方可以尽快的处理紧急数据。**
  - **ACK—为1表示确认号字段有效。**
  - **PSH—为1表示是带有PUSH标志的数据，指示接收方应该尽快将这个报文段交给应用层而不用等待缓冲区装满。**
  - **RST—为1表示出现严重差错。可能需要重新创建TCP连接。还可以用于拒绝非法的报文段和拒绝连接请求。**
  - **SYN—为1表示这是连接请求或是连接接受请求，用于创建连接和使顺序号同步。**
  - **FIN—为1表示发送方没有数据要传输了，要求释放连接。**
- 窗口大小：2字节，以字节为单位，**窗口大小是接收方告知发送方自己能缓存多大的数据，让发送方动态的调整发送数据的大小。**

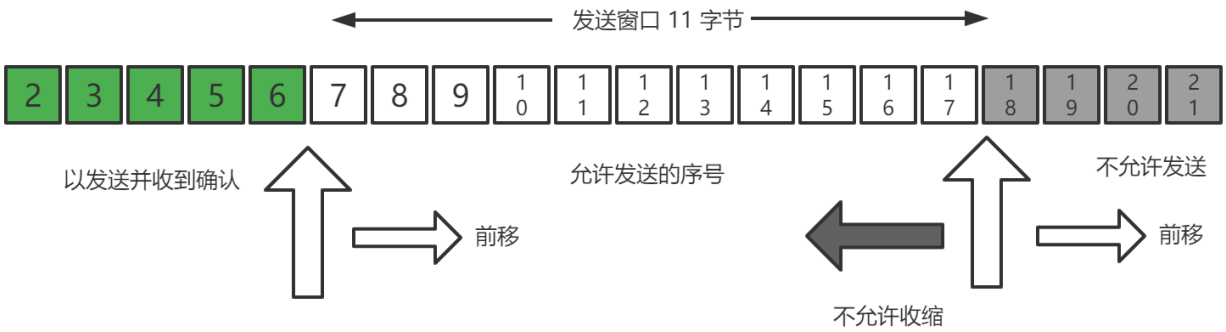
- 校验和：2字节，采用类似与 IP 校验一样的反码加检验，检验不是完全准确的，TCP只校验首部和头7字节数据。
- 紧急指针：2字节，当 URG 为 1 时，紧急指针有效，紧急指针指向紧急数据的最后一个字节，是紧急数据与普通数据的分界线。

流量控制 — 滑动窗口协议

运作原理

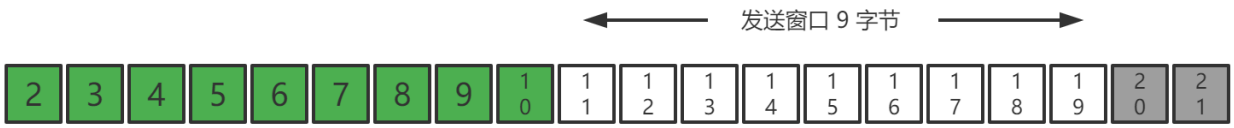
TCP 的发送是基于字节流的，在混乱的网络传输中，每一个字节可能被并不是按需到达的，后发送的完全有可能先到达，为了解决这个问题，TCP为每一个数据字节分配了一个序号，通过序号使得接收方能够有序的整理数据。

窗口通俗的来说就是缓存，发送方和接收方各自维护者自己的窗口，发送方需要将缓存中的数据发送出去，而接收方需要接收数据存储在缓存中，设置中断，等待应用程序读取。**滑动窗口协议就是为了维护发送方发送速率和接收方处理速率的一致性，也就是所谓的流量控制**，不至于说接收方只有 1 字节缓存的大小，而这个时候发送方还疯狂的发送 1460 字节，那么这个时候多余的 1459 字节就会丢失。



来看上面这个简单的窗口，由前沿指针与后沿指针确定了发送方 A 的发送窗口大小，A 只被允许发送 7 - 17 字节大小的字节，当 A 发送完了这 11 字节它必须要等待接收方 B 的 ACK 确认号与窗口大小才能继续调整 A 的发送窗口。

例如，当 A 发送 7 - 17 字节给 B 后，B 只收到了 7 - 10 字节，剩下的可能丢包了，B 期望接收 A 的第 11 字节，于是 B 将 ACK 字段设为 1，将 ACK 序列号设置为 11，同时 B 的缓存最多只能装下 9 字节了，于是 B 将 TCP 首部的窗口大小设置为 9，B 将 TCP 应答发送给 A。



A 收到后 B 的回送消息后，根据 B 的 TCP 应答去动态的调整窗口，此时窗口如上图所示，通过这样一个 ACK 确认机制，使得发送方发送速率始终与接收方接收速率一致，不会发生接收方缓存大小不够但发生方还猛发消息的情况。

前沿指针不允许收缩

如果接收方窗口变小了怎么办？前沿指针可以收缩吗？例如在开始的例子里，如果 B 回送 ACK确认号 = 11，窗口大小 = 1 怎么办？ 要将前沿指针收缩为 12(开区间) 吗？

前沿指针可以收缩，但 TCP 官方**强烈**不赞成这样做，因为这些数据很可能已经发送了，就像我们的例子中 12 - 17 字节数据已经发送给 B 了，这个时候如果还收获前沿指针就有点自相矛盾了，例如，如果 12 -17 字节仅仅只是在网络中多转了几圈，最终还是到达了 B，而恰好这个时候 B 的接收缓存又恢复了，能够收下这 5 字节，于是 B 回送 ACK = 18 给 A，这就产生了一些错误(如果前沿指针回退的话)。

1 字节报文探测

当接收方的窗口为 0 怎么办 —— 这个时候发送方会为这条 TCP 连接启动一个持续计时器，当计时器到期后，发生方就会发送一个 1 字节的 TCP 报文段询问，如果接收方调整过来了，则一切继续，否则，将会重新启动计时器继续探测。

当探测达到一定次数时，发送方会自动断开 TCP 连接。

糊涂窗口综合征

糊涂窗口增综合症可以从发送方和接收方两个层面去解决。

发送端引起的糊涂窗口综合症

如果发送端为产生数据很慢的应用程序服务(典型的有telnet应用)，例如，一次产生一个字节。这个应用程序一次将一个字节的数据写入发送端的TCP的缓存。如果发送端的TCP没有特定的指令，它就产生只包括一个字节数据的报文段。结果有很多41字节的**IP数据报**就在互连网中传来传去，**这样会导致网络由于太多的包而过载，这就是糊涂窗口综合症**。

解决的方法是防止发送端的TCP逐个字节地发送数据。必须强迫发送端的TCP收集数据，等待数据达到一定大小时再发送出去。。

发送端的TCP要等待多长时间呢？如果它等待过长，它就会使整个的过程产生较长的时延。如果它的等待时间不够长，它就可能发送较小的报文段。

Nagle找到了一个很好的解决方法，发明了Nagle算法。

Nagle算法的规则（可参考tcp_output.c文件里tcp_nagle_check函数注释）：

（1）如果包长度达到MSS，则允许发送；

（2）如果该包含有FIN，则允许发送；

（3）设置了TCP_NODELAY选项，则允许发送；

（4）未设置TCP_CORK选项时，若所有发出去的小数据包（包长度小于MSS）均被确认，则允许发送；

（5）上述条件都未满足，但发生了超时（一般为200ms），则立即发送。

接收端引起的糊涂窗口综合症

接收端的TCP也可能产生糊涂窗口综合症。

假如接收端缓存已满，而发送端不断的轮询，但**接收端处理速度非常慢**，可能很久才处理 1字节，接收端的TCP宣布其窗口大小为1字节，正渴望等待发送数据的发送端的TCP会把这个宣布当作一个好消息**，并发送只包括一个字节数据的报文段**。

这样的过程一直继续下去 —— 一个字节的数据被消耗掉，然后发送只包含一个字节数据的报文段。

对于这种糊涂窗口综合症，即应用程序消耗数据比到达的慢，有两种建议的解决方法。

1. Clark解决方法

Clark解决方法是只要有数据到达就发送确认，**但宣布的窗口大小为零，直到或者缓存空间已能放入具有最大长度的报文段(例如常见的，1460字节)，或者缓存空间的一半已经空了**。

2. 延迟确认

这表示**当一个报文段到达时并不立即发送确认**。接收端在确认收到的报文段之前一直等待，**直到入缓存有足够的空间**为止。

超时重传机制

TCP既然要实现可靠的连接，那么肯定不能置哪些丢包的数据而不管，因此必须要有一个超时重传机制。

超时时间如何计算？

我们将超时时间记为 RTO（Retransmission-Time-Out），RTO 应该略大于理想状态的 RTT(报文往返时间)， 我们来思考一下RTO的选定依赖于什么呢？

首先，在网络波动较小的情况下，所有报文的 RTT 相差不大，此时 RTO 可以设置为所有报文的 RTT 的**加权平均值** ERTT，但这只是理想状态，我们仍需要考虑到网络波动情况，例如，如果第一次测试 RTT 样本为 10000ms(设样本 RTT 为 SRTT)，但第二次、第三次、第四次 SRTT 均只有不到 10ms，如果让我们人为的去选择，我们很容易会偏向于小一点的 RTO(例如10ms)，但如果采用计算加权平均的方法，RTO 会被第一次样本 SRTT 所影响，使得 RTO 高达几千，所以我们必须还要考虑到网络波动情况，**即两次 SRTT 之间的偏差值**，我们需要统计一个加权平均的偏差值 DRTT 以作为参考。

现在我们知道，RTO的选定主要依赖于两种数据：

- 样本 SRTT 的加权平均值 ERTT。
- 两次 SRTT 之间的加权平均偏差值 DRTT。

那么有

$$ERTT = \alpha \times ERTT + (1 - \alpha) \times SRTT \; DRTT = \beta \times DRTT + (1 - \beta) \times |SRTT - ERTT|$$

在 DRTT 的计算中，我们用 ERTT 代替上一次的 SRTT，TCP 官方推荐的权重因子为：

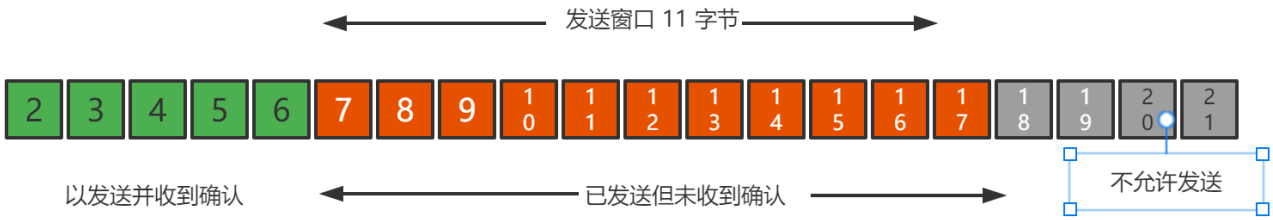
$$\alpha = 0.125 \; \beta = 0.25$$

现在知道了 ERTT 和 DRTT，我们可以计算 RTO 了，官方给出的公式是：

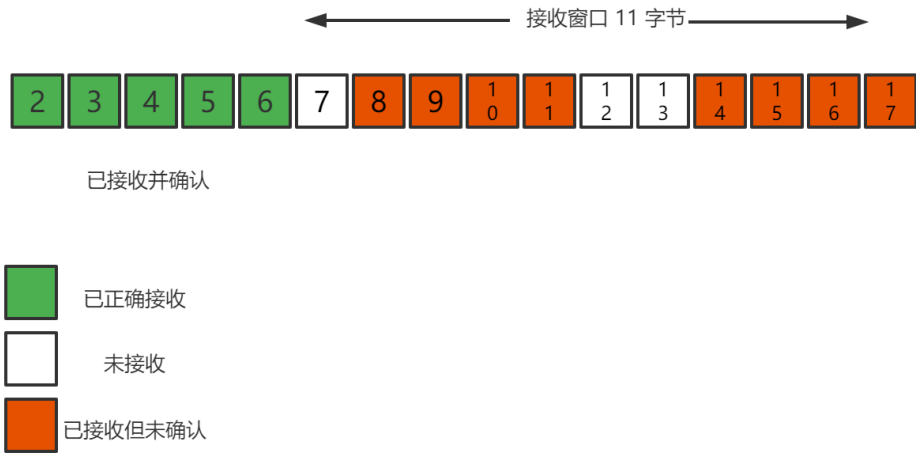
$$RTO = ERTT + 4 \times DRTT$$

注意，**对重传的报文不应该计算 SRTT**，也根本没法计算，你不知道接收到的应答是对重传之前的报文的应答还是重传的报文。

GBN 回退N步(Go Back N) 和 SR 选择重传(SELECR RETURN)



还是这个例子，如果发送方发送了 7 - 17字节，我们假设接收方的窗口如下：



考虑这个例子，此时接收方 B 理应回送 ACK = 7，那接收方收到这个 ACK 之后，重传序号 7 字节不假，那 8，9，10...之后已经发过的还要不要传？

回退 N 步算法中要求 TCP 发送方从 ACK 处重传一切，而接收方接到冗余的报文也会像往常一样回送 ACK，一切的一切就好像回到了之前，这就是回退 N 步名字的由来，这种做法是实现简单，但缺点也比较明显。

选择重传算法中要求 TCP 连接双方协商好要重传的报文，例如这里的例子，接收方 B 应该明确告诉发送方 A 你得给我发 7 和 12、13，这些数据被记为紧急数据，此时，USG标志有效。

糅合的实现，SACK选择确认

GBN 和 SR 各有千秋，那为什么不将它们糅合起来呢？

SACK 就是这样一个思想，**SACK允许接收方有选择的确认最后一个接收的正确有序报文段**，这允许接收方跳过那些已接收但未确认的报文段。

例如还是上面的例子：

- B 设置 ACK = 7
- A 从 7 开始重发，A 会继续发送 7、8、9...等报文
- 当 B 收到 7 后，他会选择确认 ACK = 12，而跳过 8、9、10、11这些报文。
- A 收到 ACK = 12后，之前没发完的报文(小于 12)也不发了，直接从 12 开始继续重传。
- B 收到 12 后，选择确认 ACK = 18，而跳过 13, 14...

这就是选择确认的工作原理，也是当下的主流实现，糅合和 GBN 和 SR 的思想，其中**发送方默认 ACK 之前的报文段均被确认**。

快速重传

有些时候，我们已经明确知道了一些数据报已经丢失了，这个时候没有必要继续等待超时计时器触发，我们可以直接选择重传这些数据报。

TCP 规定，当收到对同一个数据字节的 **3 次冗余 ACK** 时，我们有理由相信这个数据已经丢失了，这是因为能收到 ACK 说明网络不算堵塞，而在网络不堵塞的前提下，按道理数据早就到了，但仍收到 ACK 说明数据并没有到，那么这个数据报大概率已经丢失了，而不是在网络中迷路。

这就好比接收方在说：“大哥，其他的我都收到了啊，就差这一个了啊，赶紧的啊”。

就像接收方说的，此时**没有必要磨蹭**，我们可以直接重传而不必等待超时事件触发。

要几个定时器？

难不成为每一个要发送的 TCP 数据报都维护一个定时器，如果你想你的电脑下一秒就爆炸那就这么做吧。

好吧，事实上是没必要的，我们只需要**维护一个定时器**就好了，发送报文前，来看看超时器是否未启用，如果未启用，则启用，否则什么也不做，仅仅只是发送报文（RFC 6298推荐的做法）。

当计时器超时时，重发在窗口内、收到的、最大的 ACK 数据段(也就是我们要重传的最小序号的报文段)，并重启计时器。

当收到 ACK 时，这意味着**之前发送的报文**得到了确认，那么定时器可能也是**之前启动**的，此时应该停止定时器，因为它很可能是过时的，并且重新启动(如果还有报文未确认的话)。

这样做虽然不是很精确，但它节省了大量的开销，而且大多数情况下工作的良好，因为从整体的角度看过去，计时器一直都在工作，没有一刻闲着，这意味着，即使有误差，误差也绝不会超过一个 RTO。

> 可参阅Information on RFC 6298 » RFC Editor (rfc-editor.org)

TCP 发送方精简伪代码

```
var nextSeqNum = 初始确认的序号; // 下一个要发送的
var sendBase = 初始确认的序号;  // 发送窗口左指针(后沿)
var size = 初始接收方窗口大小; // 窗口大小
while (true) {
    Event e = 堵塞监听事件;
    switch (e) {
        事件：从应用程序中收到数据 data，data 不超过 size，准备将其发送
            生成TCP首部序号为 nextSeqNum 的 TCP 报文段;
            if (定时器没有运行)
                启动定时器;
            向IP传递报文段;
            nextSeqNum = nextSeqNum + data.length;  // 一个字节一个序号
            break;
        事件：定时器超时
            // TCP 应当缓存序号为在窗口内、收到的、最大的 ACK 的数据报大小
            重发在窗口内、收到的、最大的 ACK 数据段(也就是我们要重传的最小序号的报文段);
            启动定时器;
            break;
        事件：收到 ACK，ACK 标志位有效，ACK确认序号为 y，收到窗口大小为 size
            this.size = size;
            // 判断 y 是否在窗口内
            if (y > sendBase) {
                // 这之前的默认确认(SACK)，窗口左指针移动
                sendBase = y;
                if (当前存在未被确认的报文段) {
                    重启定时器
                } else {
                    对 y 收到的冗余 ACK 数加 1;
                    if (对 y 收到的冗余 ACK 数 == 3) {
                        立即重传需要为 y 的数据报 // TCP 应当缓存序号为 y 的数据报大小
                    }
                }
            }
            break;
    }
}
```

拥塞控制

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，网络就会热载，这种情况就叫做网络拥塞。若出现拥塞而不进行控制，仍然我行我素的发送大量资源，整个网络就会变得越来越拥堵，所以我们要做一个文明人，避免拥塞时发送大量消息，**这就是 TCP 的拥塞控制。**

这个时候，多了一个**拥塞窗口**，拥塞窗口的大小根据网络拥塞程度动态改变。现在 TCP 发送速率不仅取决于滑动窗口，还取决去拥塞窗口，即：

$$发送窗口大小 = min(拥塞窗口大小，接收方的滑动窗口大小)$$

TCP 拥塞控制的算法就是根据网络状态去动态的调整拥塞窗口的大小，TCP 希望网络好拥塞窗口能大些，网络差，拥塞窗口能小些，那么现在问题是，如何判断网络好不好呢？这有几种实现的思想：

- 让路由器报告网络拥塞情况。事实上 TCP 标志位里有些就是为此而生的，让路由器报告的优点时相对比较准确，缺点是如果网络本身就是拥堵的，发送报告无疑也增加了负担。
- 每收到一个 ACK 就认为网络没问题，而每一次超时事件都认为网络发送了拥塞。

TCP 广泛采用了第二种实现方式，相对简单，而且误差并不会很大。

TCP拥塞控制算法主要包括 3 个部分：

- 慢启动
- 拥塞避免
- 快速恢复

在次之前，我们先定义一些全局变量，我们定义:

$$cwnd = 拥塞窗口大小\ ssthresh = 慢启动的阈值 = 检测到网络拥塞时cwnd的一半$$

并且我们不会考虑滑动窗口的问题，约定发送窗口 = 拥塞窗口。

慢启动

别被误导了，慢启动可一点也不慢，慢启动存在的意义就是疯狂作死，**它将以最快的速度去试探网络所能承载的极限(网络即将或已经拥塞)，并快速增大 cwnd 到网络能承载的最大阈值**。

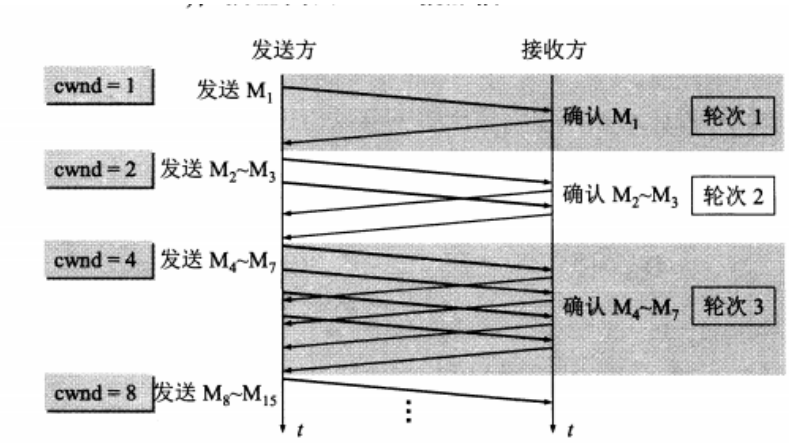
初始时，TCP 通常将 cwnd 置为 1 个 MSS(TCP 最大报文段长度，通常等同于 MTU，1460字节)，此时发送方只能发送 cwnd 个字节。

当 TCP 收到一个报文段 ACK 时，他会认为网络很好，于是开始作死，设置：

$cwnd = SMSS + cwnd$，$SMSS = 发送方发送的最大报文段大小，这通常是一个MSS、1460字节$

> 一般没有特殊说明，将默认 SMSS = MSS = 1460字节
>
> 还要注意如果接收方使用 SACK，那么很可能会一次性确认多个报文段，此时需要计算报文段个数
>
> $$N = (ACK - baseSend) \div SMSS$$

于是下一次 TCP 可以发送 2 个 MSS 字节，此时接收端完全可以一次性确认两个报文段，于是 `cwnd += 2MSS`，下一次 TCP 可以发送 4 个 MSS 字节，而接收端又可以一次性确认 4 个报文段，那么 cwnd 变成 8MSS，下一次可以发送 8 个 MSS 字节...这一点也不慢，这可是指数级别的，如果时间复杂度是这个的话写算法题肯定超时了。



慢启动何时停止？有两种情况:

- 当 cwnd >= ssthresh(初始为无限大)时。根据我们对 ssthresh 的定义，$ssthresh = 慢启动的阈值 = 检测到网络拥塞时 cwnd 的一半$，当 cwnd >= ssthresh 时，cwnd 不能翻倍的增长下去了，TCP已经探测到了阈值，慢启动的任务已经完成了。上次它就是死在 2 * ssthresh，翻倍增长下去就会超过这个数，这次不能犯同样的错误。此时 TCP 会由慢启动转向拥塞避免。
- 当网络确实发生拥塞时，即触发了超时机制，慢启动探测到了阈值，这个阈值应该小于等于当前的 cwnd 的一半，它仍然不确定具体的阈值，此时设置 ssthresh = cwnd / 2，TCP 将 ssthresh 视为阈值，然后**重新进入慢启动状态**去试探阈值是否准确，此时 cwnd 重新设置为 1MSS。
- 当收到三次冗余 ACK 时，正如我们前面讲的，既然能接收到 ACK，网络肯定不会那么糟糕，那到没有必要重新进入慢启动尝试，TCP 将 ssthresh =设为 cwnd / 2，然后直接进入快速恢复阶段。

拥塞避免

进入拥塞避免时，如果将 cwdn 在翻个倍就达到了阈值，但此时的 cwnd 确确实实又是没达到的阈值的，因此不能停止试探，而是要慢慢试探。

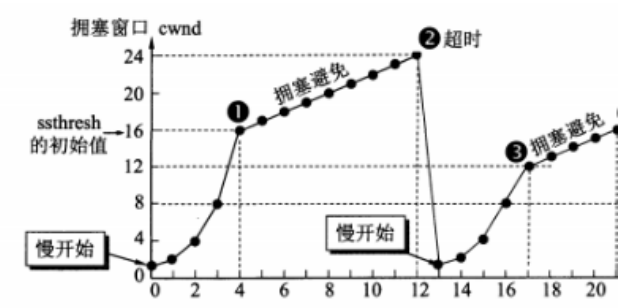
拥塞避免并非完全能够避免拥塞，是说在拥塞避免阶段将拥塞窗口控制为按线性规律增长，让 cwnd 慢慢的增长，使网络比较不容易出现拥塞。

具体的思路是：

让拥塞窗口cwnd缓慢地增大，即报文段每经过一个往返时间 RTT 就把发送方的拥塞控制窗口加上 1 MSS，这等价于每收到一个 ACK(默认是对报文段的 ACK，即一次确认 1MSS) 就增加 cwdn 一个 MSS。

拥塞避免何时停止？与慢启动一样，我直接内容复制过来：

- 当网络确实发生拥塞时，即触发了超时机制，慢启动探测到了阈值，这个阈值应该小于等于当前的 cwnd 的一半，它仍然不确定具体的阈值，此时设置 ssthresh = cwnd / 2，TCP 将 ssthresh 视为阈值，然后**重新进入慢启动状态**去试探阈值是否准确，此时 cwnd 重新设置为 1MSS。
- 当收到三次冗余 ACK 时，正如我们前面讲的，既然能接收到 ACK，网络肯定不会那么糟糕，那到没有必要重新进入慢启动尝试，TCP 将 ssthresh =设为 cwnd / 2，然后直接进入快速恢复阶段。

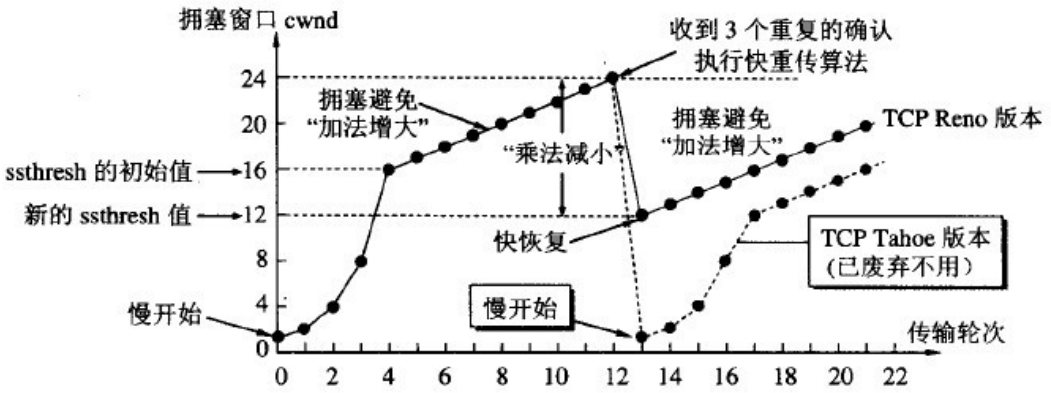


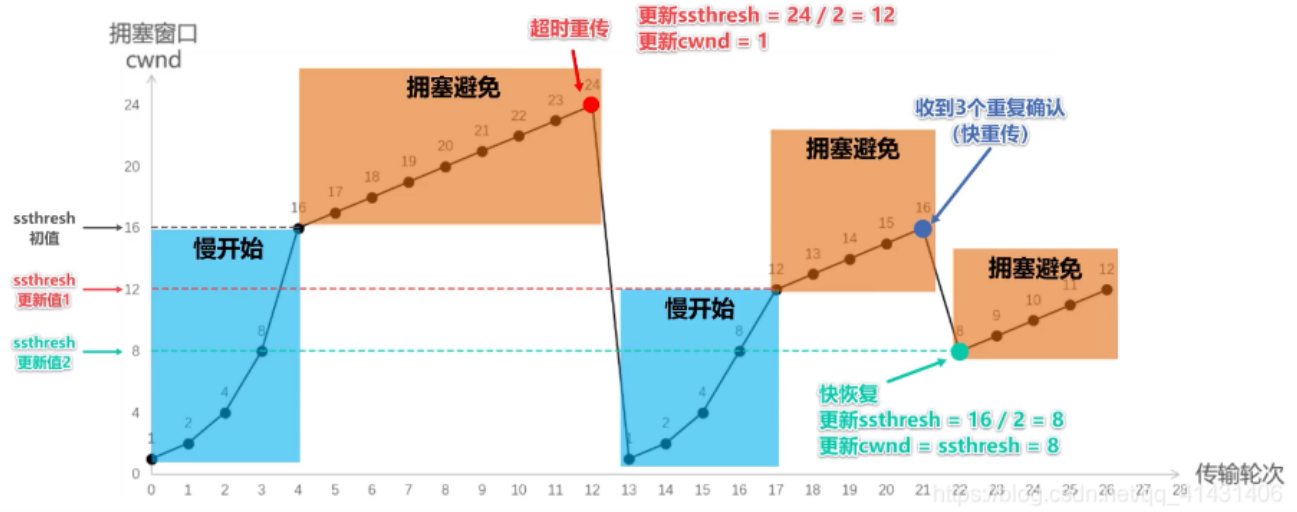
快速恢复

当收到 3 个冗余 ACK 时，并不会将 cwnd 置为 1MSS，我们有理由相信此时网络没那么糟糕，因此快速恢复会将 cwnd 减半，即“乘法减小”：

$$cwnd = cwnd \times 0.5$$

就像我们之前说的，然后 TCP 会快速重传冗余数据报，在此期间，每收到一个 ACK 就增大一个 MSS，直到冗余数据报重传完毕并被确认，此时重新进入拥塞避免状态。





连接管理

3次握手

TCP三次握手如图：

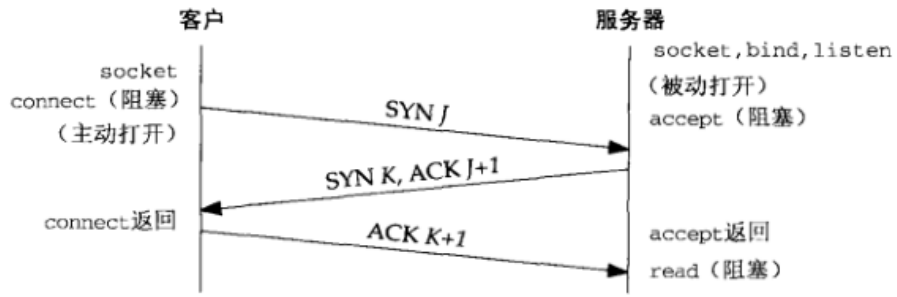


图2-2　TCP的三路握手 og. csdn. net/ jun2016425

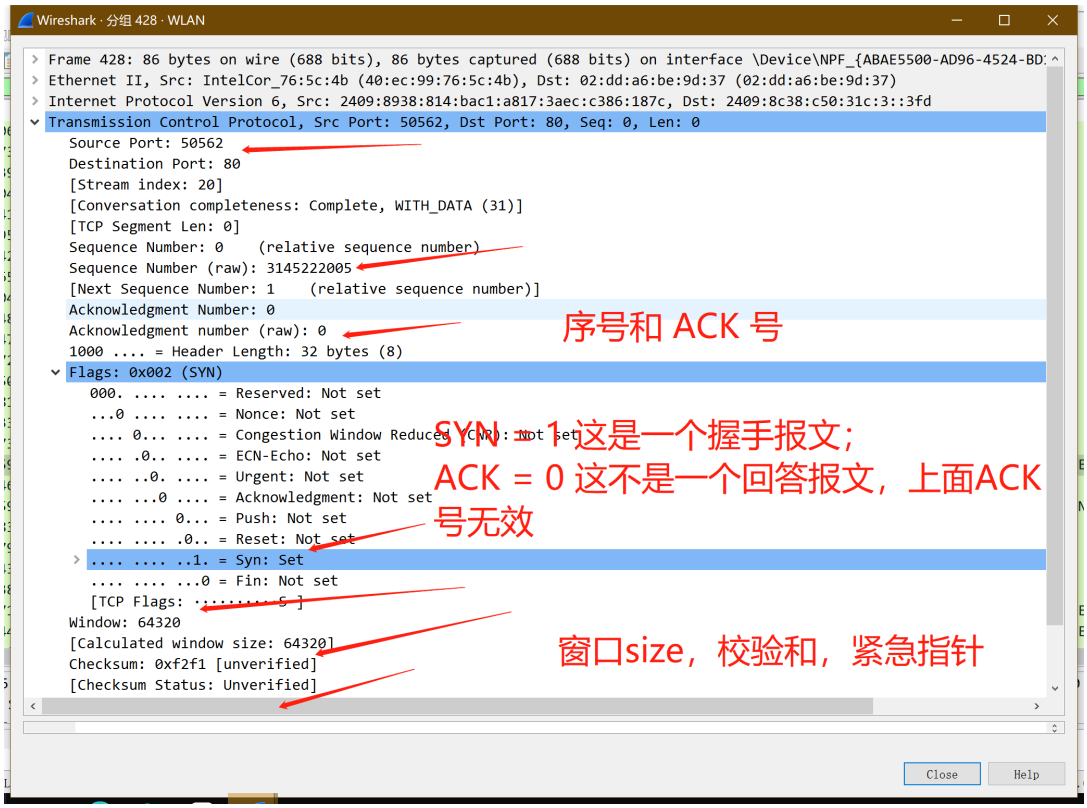
1. 第一次握手 客户端给服务器发送一个SYN数据段(在 TCP 标头中 SYN 位字段为 1 的 TCP/IP 数据包)，该TCP数据段中也包含客户端的初始序列号(Sequence number = J)，**同时会告知服务器缓存窗口大小**。
2. 第二次握手 服务器返回客户端 SYN +ACK 段(在 TCP 标头中SYN和ACK位字段都为 1 的 TCP/IP 数据包)， 该段中包含服务器的初始序列号(Sequence number = K)；同时使 Acknowledgment number = J + 1来表示确认已收到客户端的 SYN段(Sequence number = J)，**同时会告知客户端缓存窗口大小**。

> 同时这个时候服务器会将连接放入半连接队列中，如果半连接队列已满，则不会回送 SYN + SCK，而是丢弃该连接。

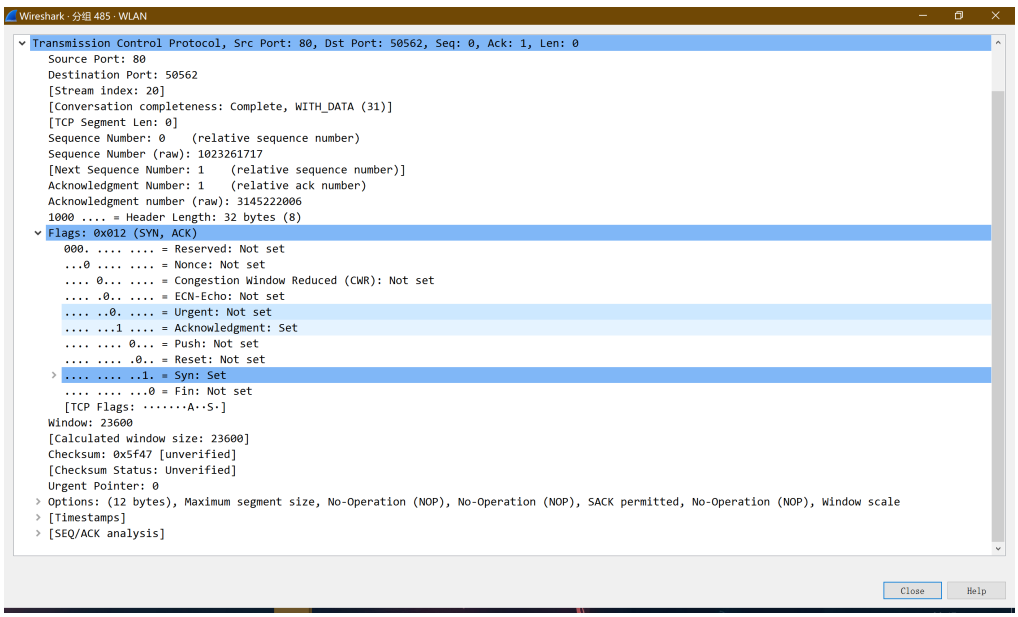
3. 第三次握手 客户端给服务器响应一个ACK段(在 TCP 标头中 ACK 位字段为 1 的 TCP/IP 数据包)，该段中 Acknowledgment number = K + 1 来表示确认已收到服务器的 SYN段(Sequence number = K)。

> TCP连接从半连接队列转向全连接队列，当全连接队列满时，此时不会简单的抛弃，处理比较复杂，服务器可能会丢弃连接，也可能会发送 RST 报文重置连接。

经过三次握手，双方协商好了对应的初始序列号和缓存端口大小(双方都可当发送方或者接收方)。



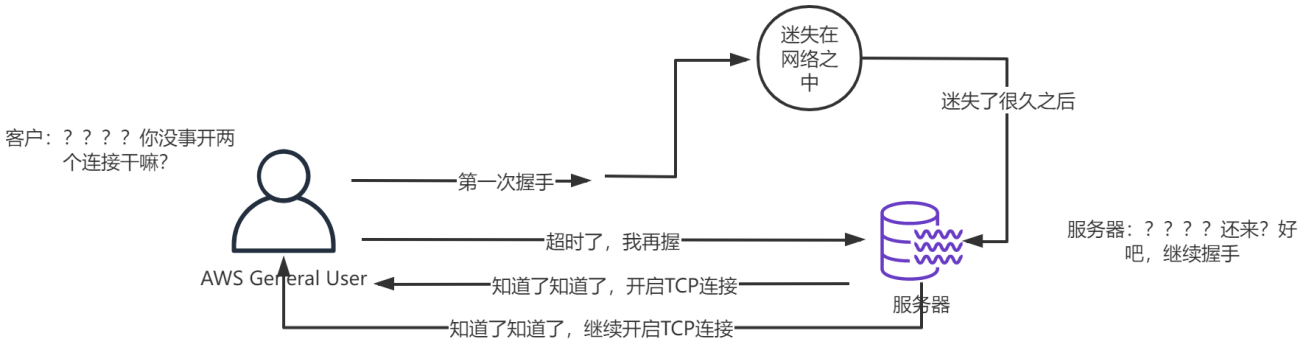
这是我(50562端口)随便搜了个鬼玩意抓的包，上面是我像服务器发送的握手报文，因此 ACK 无效，同时我将我的序列号和窗口大小一并发送给服务器。



这是服务器给我的答复，可以看到 ACK 和 SYN 都设置为 1，同时对方也发送了窗口大小和初始序号给我门，这样我们双方都可作为发送方，而且可以看到 ACK 就是我们发送过去的序号加 1。

客户端回送 ACK 的图就不发了。

为什么不能 2 次握手



看图，其实主要是没法确认客户端的握手报文是已经超时丢弃的，还是真要握手...如上图，如果客户端一直超时就会有大量握手报文留在网络种，而在某一时刻它们突然达到服务器，服务器就不得不开启这么多连接。

让客户端在确认一次就可以解决这个问题，通过客户端的 ACK 可以唯一的标识一条握手报文。

> 啥？为啥不能四次？当然可以啊，谁说不能的，服务器把 ACK 和 SYN 当两次发就是四次了...
>
> 同理，挥手也是可以三次的。

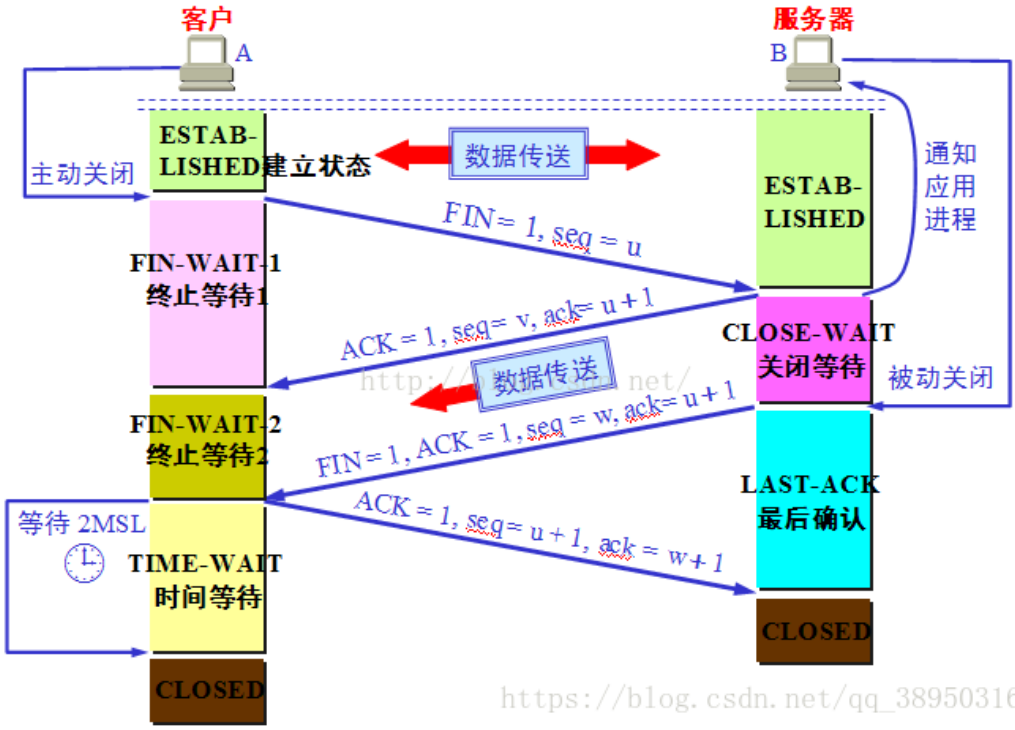
SYN 泛洪

我们在握手阶段说了，如果半连接队列已满，服务器会丢弃该连接，SYN泛洪就是利用了这个漏洞，**攻击者向服务器发送大量 TCP 连接，但是不确认它，让其占满服务器半连接队列，从而让服务器连接资源耗尽，陷入瘫痪的状态**。

> 先说明一个问题，我们为什么要有半连接队列，这是客户端会发送第三次握手的 ACK，我们必须要有足够多的信息来确定这是一个合法的报文，我们要枚举队列中每一个半连接，看看是否匹配，若匹配，则生成全连接，否则，无视这个报文。

解决的方法不少，比较流行的是部署 SYN cookie 防御系统，Syn Cookie技术则完全不使用任何存储资源，这种方法比较巧妙，它使用一种特殊的算法生成 Sequence Number，这种算法考虑到了对方的IP、端口、己方IP、端口的固定信息，以及对方无法知道而己方比较固定的一些信息，如MSS、时间等， Syn Cookie 将这个 Sequence Number 作为序列号发送给客户端，在收到对方的ACK报文后，重新计算一遍，看其是否与对方回应报文中的 ACK Number -1 相同，从而决定是否分配TCB资源。

4次挥手



1. 第一次挥手：客户端发送 FIN 报文段。
2. 第二次挥手：服务器确认该 FIN 报文段。
3. 第三次挥手：服务器发送 FIN 报文段。
4. 第四次挥手：客户端确认，并进入TIME-WAIT阶段等待2MSL（**最大报文生存时间，即报文在网络中理论上能生存的最大时间**）后再断开连接，服务端收到最终确认报文后立即断开连接，双方断开TCP连接。

为什么要Time-Wait

TIME_WAIT状态存在的原因有两点：

1. 可靠的终止TCP连接：保证客户发送的 ACK 能够被服务器正确收到，如果没有收到，服务器会重传 FIN，所以客户必须要等待一定的时间。

   > 如果客户确实异常断开了，服务器端也会有保活机制，在重传一定次数后无反应后，服务器会断开连接。

2. 保证让迟来的相同TCP报文有足够的时间被识别并丢弃。

过多 Time-Wait 占用端口

如果主动发起 close 的是服务器，那么服务器就会存在 Time-wait 在占用端口，当 Time-wait 过多时，服务器压力就会变大，端口紧缺(客户端也是一样)，在 Linux 中可以开启TIMEWAIT重用和快速回收。

- 重用：允许将TIME-WAIT sockets重新用于新的TCP连接。
- 快速回收：不必等待 2MSL，加快速度！

编辑Linux内核配置文件 /etc/sysctl.conf，找到如下参数：

```
net.ipv4.tcp_syncookies = 1  // 表示开启SYN Cookies。 当出现SYN等待队列溢出时，启用cookies来处理，可防范少量SYN攻击，默认为0，表示关闭；
net.ipv4.tcp_tw_reuse = 1 // 表示开启重用。允许将TIME-WAIT sockets重新用于新的TCP连接，默认为0，表示关闭；
net.ipv4.tcp_tw_recycle = 1 // 表示开启TCP连接中TIME-WAIT sockets的快速回收，默认为0，表示关闭。
net.ipv4.tcp_fin_timeout = 30 // 修改系默认的 TIMEOUT 时间 30 s
```

粘包拆包

先提出一个问题，你能通过 TCP 首部确定 TCP 数据大小吗？

很遗憾，你应该不能，好吧，如果算上选项的话，那你应该又能了。

大多数情况下，这个问题的答案是不能，也就是说，TCP是无消息边界的，TCP首部 起到的仅仅只是简单的校验和流量空中，对一条消息是没有边界限定的，你只能不断的去读取 TCP 字节流。

对于像传输文件、传输视频等这样的，是没有粘包的概念的，因为它本身就是一个整体，自然也没有谁粘谁了。

但是考虑下面这种情况，你正在和别人用 TCP 聊天：

你发送了一条消息：`I will select d`。但是 TCP 认为数据太少了，现在发出去会造成糊涂窗口综合征，我要等多一点在发送出去，，然后你又键入了：`ad to carry`，这个时候数据被转换为字节流加入缓存中，TCP认为差不多了，是时候该发送了，于是接收方就受到了：

```
I will select dad to carry
```

啥？你要选你爹去 Carry？你和你朋友二脸懵逼。

这就是粘包，可能引发粘包的原因有：

- 发送方收到数据后并不立即发送，而是等待数据大小达到一定阈值。
- 接收方处理速度不快，使得下一个数据报头部黏在前一个数据报尾部。

什么是拆包呢？

假设发送方缓存只剩下 200 字节，而你现在要发送 1000 字节数据，那么这个时候就会发生拆包，1000 字节的数据将会拆成 200 + 800 发送，200字节将会粘在前一个数据的尾部。

怎么处理粘包拆包呢？

- 在首部字段的选项中加上消息大小。
- 发送方与接收方约定好，每个消息都是固定的长度。
- 以约定好的特殊字符作为结尾。