挥手，即告别的意思，当应用程序完成通信后，还需要关闭它们之间的 TCP 连接。是的，我们不会无限期地保持连接状态。如果我们从不释放它们，那么所有的端口很快都会被占用。

提到告别，想起吴奇隆的《祝你一路顺风》，如果你在旅途上，累了可以听一听：

> 当你背上行囊 卸下那份荣耀
> 我只能让眼泪留在心底
> 面带着微微笑 用力的挥挥手
> 祝你一路顺风

本篇文章，我们不必为告别而伤心，而要为能成功断开连接而开心，我们来看下TCP是如何断开连接的，中间会发生哪些问题。

一、天下没有不散的宴席

双方都可以主动断开连接，断开连接后主机中的「资源」将被释放，我们先来用打电话的过程白话说明下结束过程：

- fossi：hey，老弟，今天我说的就这么多，没有啥要说的了，我想挂电话了（FIN）
- stephen：收到（ACK），不过我刚才说的话还没说完，你下面就听我把最后两句话说完哈，巴拉巴拉。。。
- stephen：老哥，我说完了，我也想挂电话了呢(FIN)
- fossi：收到（ACK），老弟你挂吧！我等待2MSL再挂电话，我怕老哥你收不到我的ACK，保险起见，我等一会

其中涉及到一个标志位：**FIN**：

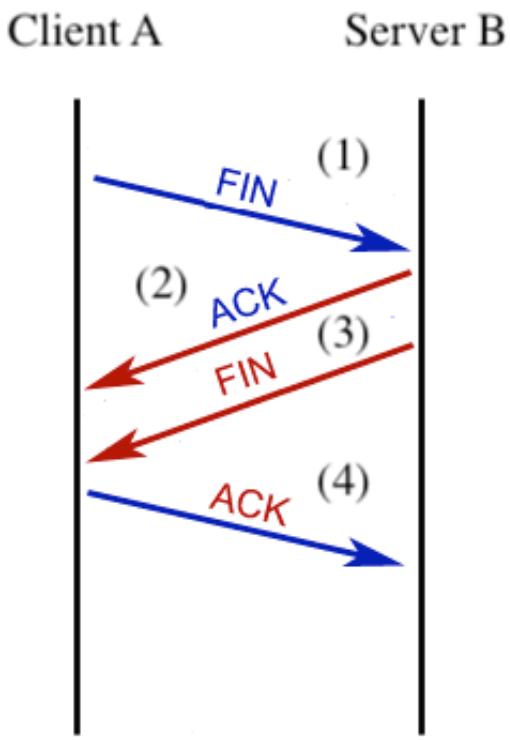
发送方的TCP报文段首部中该字段为1时，则表示该发送者今后不会再有数据发送，希望断开连接。

由于TCP中通信是全双工的，因此需要两个方向上都发送FIN来各自表明自己数据已经发完，希望断开连接。

连接的终止将以如下步骤进行：

- 1、客户端 A（Client A）发送一个设置了 FIN 标志位的数据包给服务器 B（Server B），请求关闭从 A 到 B 的连接。
- 2、服务器 B 收到这个设置了 FIN 标志位的数据包，发回一个 ACK 数据包作为确认。**从 A 到 B 这个方向的连接被终止**。
- 3、服务器 B 请求关闭从 B 到 A 的连接，发送一个 FIN 数据包给客户端 A。
- 4、客户端 A 发回 ACK 数据包确认关闭请求。**从 B 到 A 这个方向的连接被终止**。

可以用下图来总结上面所述的终止连接的过程：

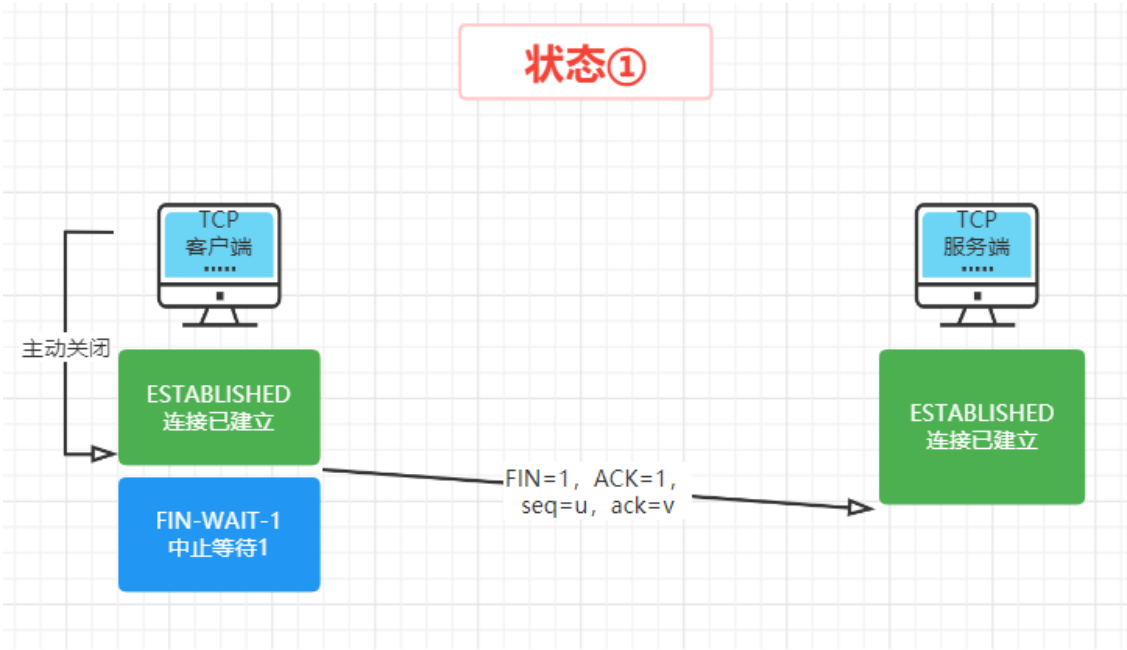


二、四次挥手详细过程

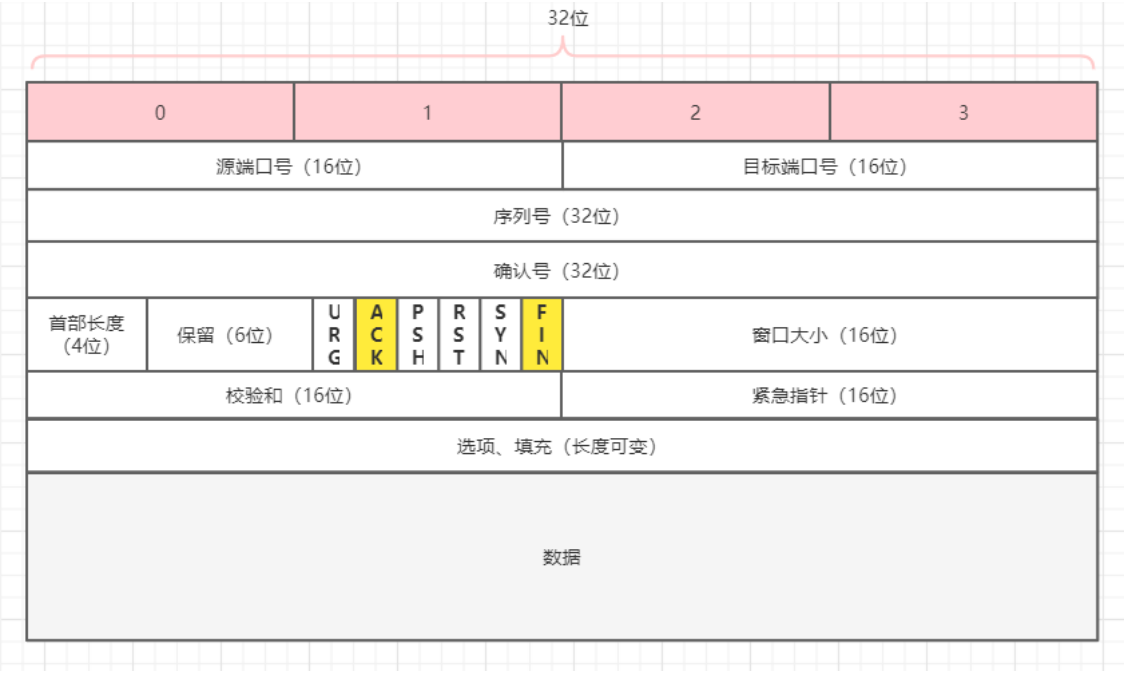
数据传输结束后，TCP通信双方都可以释放连接，现在客户端和服务端都处于连接已建立状态，假设我们现在是客户端主动发起关闭连接。

主动发起关闭的一方称为「主动关闭方」，另外一段称为「被动关闭方」。

状态①：客户端发送TCP连接释放报文段后，**客户端进入中止等待1状态**。



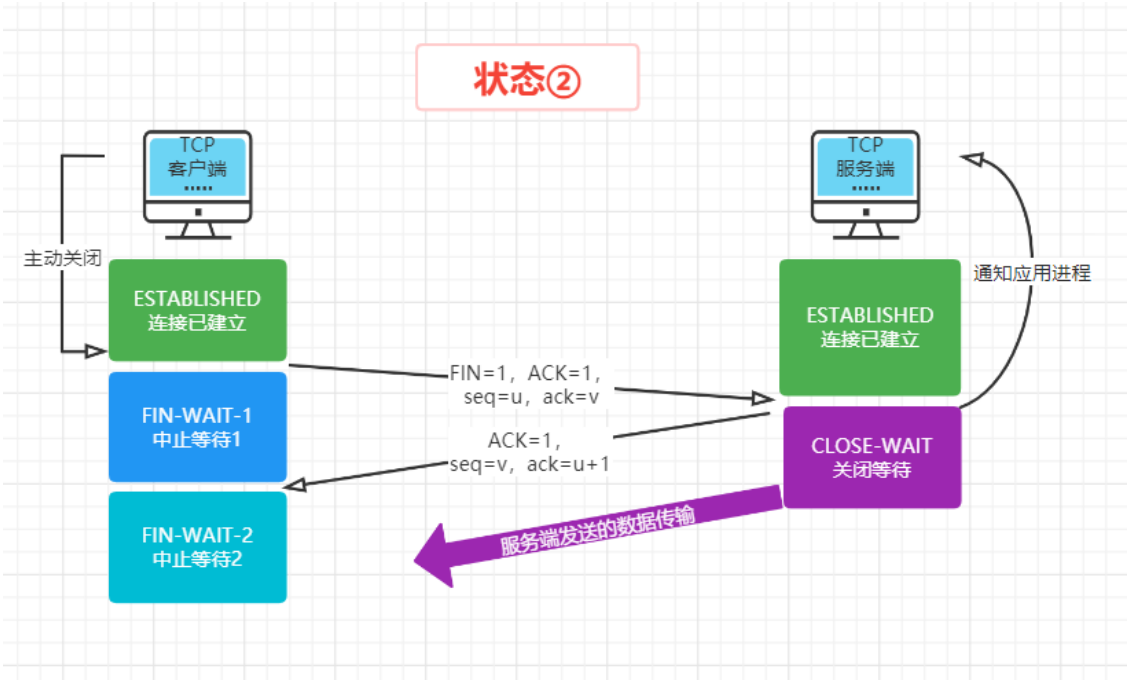
其中FIN和ACK都被设置为1，表明这是一个TCP连接释放报文段，同时也对之前收到的报文段进行确认；序列号seq的值设置为u，等于客户端之前已传送过的数据的最后一个字节的序号加1；ack的值设置为v，等于客户端之前已收到的数据的最后一个字节的序号加1。



FIN 段是可以携带数据的，比如客户端可以在它最后要发送的数据块可以“捎带”FIN 段。当然也可以不携带数据。**不管 FIN 段是否携带数据，都需要消耗一个序列号。**

客户端发送 FIN 包以后不能再发送数据给服务端，但是还可以接受服务端发送的数据。这个状态就是所谓的「半关闭（half-close）」

状态②：服务端收到 FIN 包以后回复确认 ACK 报文给客户端，**服务端进入CLOSE_WAIT关闭等待状态，客户端收到ACK以后进入FIN-WAIT-2状态。**



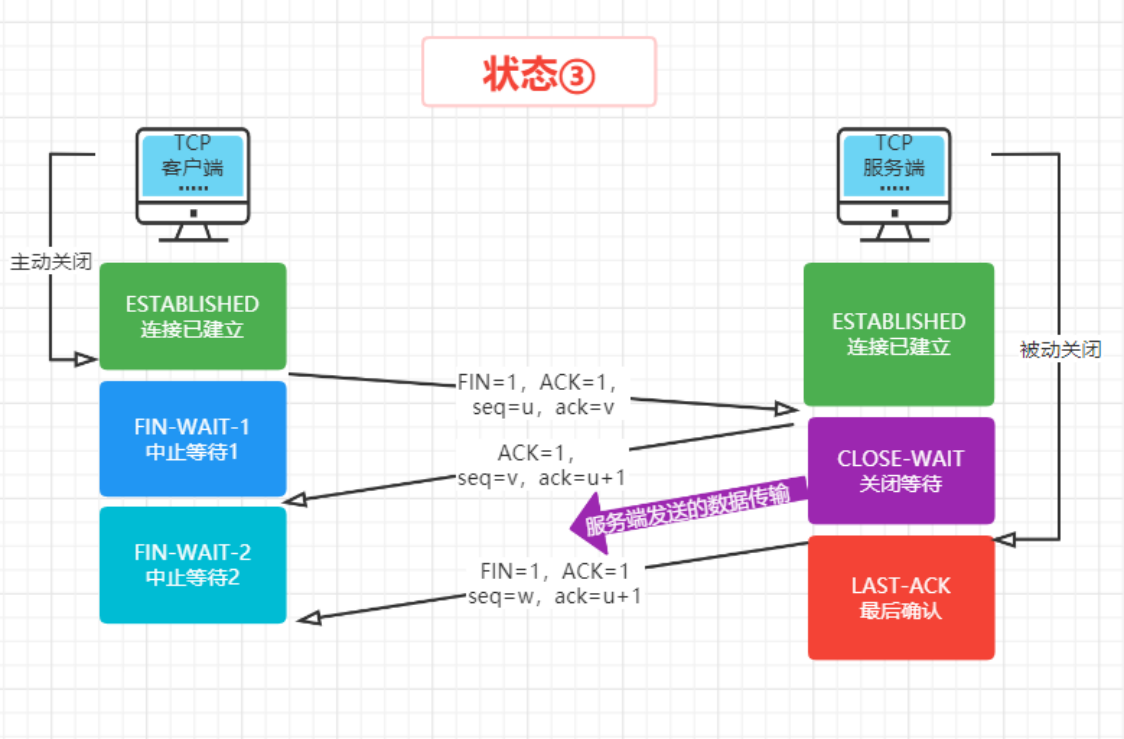
并且服务端告知高层的应用进程客户端要断开与自己的TCP连接了，**此时，从TCP客户端到服务端这个方向的连接就彻底释放了**。

不过服务端如果还有数据要发送，客户端仍然要接收，也就是说，服务端到客户端这个方向并没有关闭，这个状态可能要持续一段时间。

处于FIN-WAIT-2状态的客户端等待来自服务端的FIN报文段，以进行下一步操作。

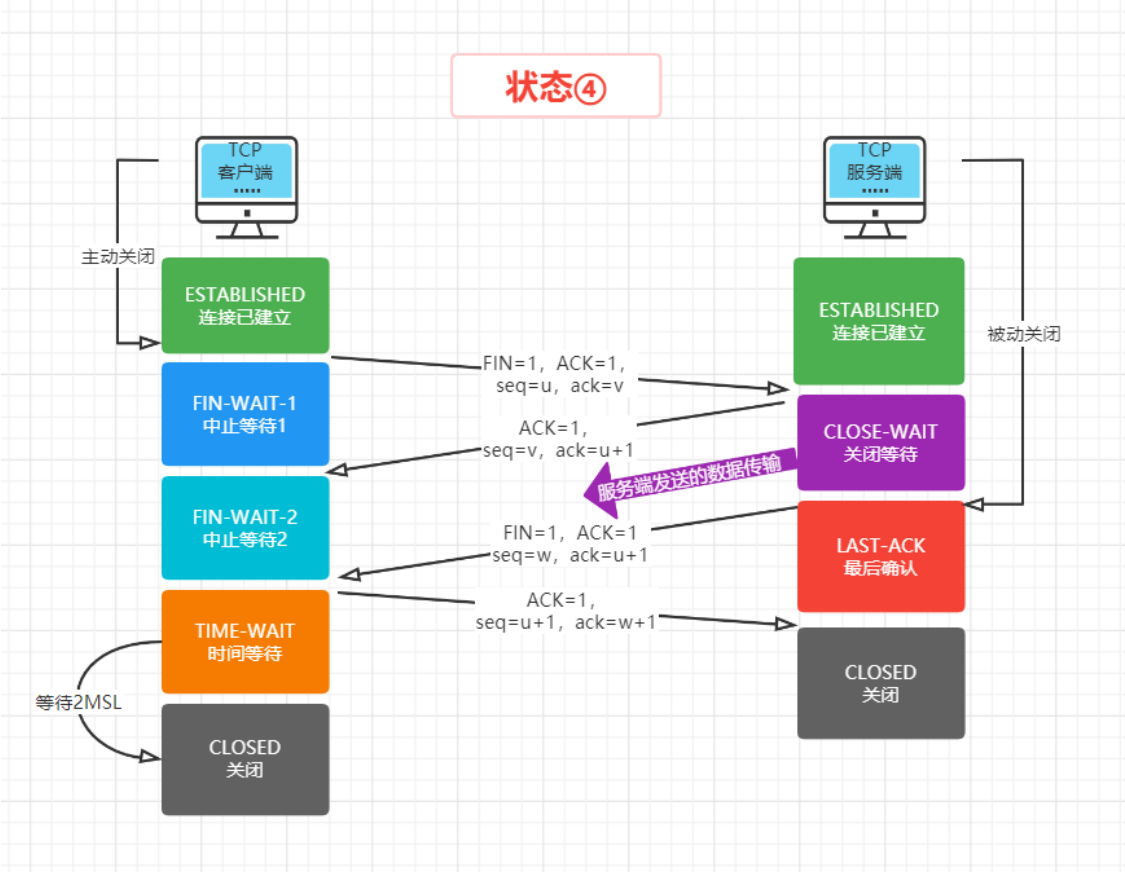
其中ACK的值被设置为1，表明这是一个确认报文段；seq的值为v，等于服务端之前已传送过的数据的最后一个字节的序号加1；ack的值设置为u+1，等于服务端之前已收到的数据的最后一个字节的序号加1（注意，这里seq和ack都是一个假定值，为了方便理解用的，假设的是FIN并没有携带任何数据了，如果携带了那就可能不是u+1了，所以要根据实际情况来）。

状态③：服务端也没有数据要发送了，发送 FIN 报文给客户端，然后**服务端进入LAST-ACK状态**，等待客户端的 ACK。同前面一样如果 FIN 段没有携带数据，也需要消耗一个序列号。



该报文段首部中的FIN和ACK被设置为1，表明这是一个TCP连接释放报文段，同时也对之前收到的报文段进行确认；seq假定为w，因为服务端在发送本FIN报文段前可能又发送了一些数据；ack被设置为u+1，这是对之前收到的TCP连接释放报文段的重复确认。

状态④：客户端收到服务端的 FIN 报文以后，回复 ACK 报文用来确认第三步里的 FIN 报文，客户端进入TIME_WAIT状态，等待2个MSL以后进入CLOSED状态。服务端收到 ACK 以后进入CLOSED状态。



ACK被设置为1，表明这是一个普通的TCP确认报文段；序列号seq被设置为u+1，这是因为客户端之前发送的TCP连接释放报文虽然不携带数据，但是也要消耗一个序列号；确认号ack被设置为w+1，这是客户端对服务端发出的FIN报文段的一个确认。

服务端收到ACK后立即进入关闭状态，但是客户端还要经过2MSL后才能进入关闭状态。

MSL全称是Maximim Segement Lifetime，即最长报文寿命，RFC793中建议为2分钟，也就是说客户端进入时间等待状态后再等待4分钟后进入关闭状态。

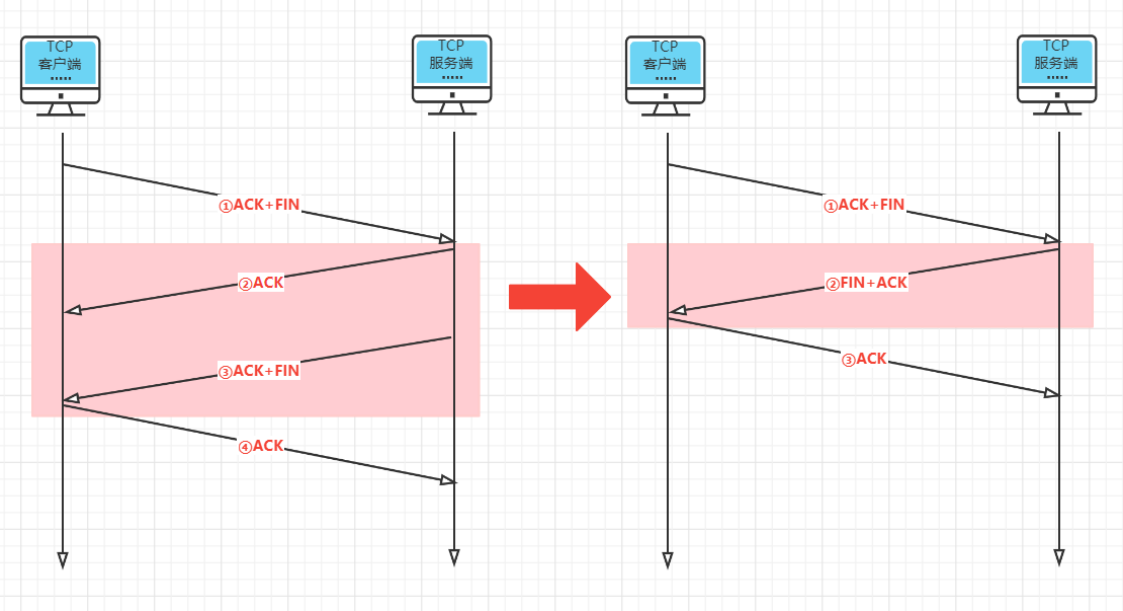
对于现在的网络，MSL取为2分钟可能太长了，因此TCP允许不同的实现可根据具体情况使用更小的MSL值。

三、为什么挥手要四次？变成3次可以吗？

回答是可以，但是一般不会这么做。

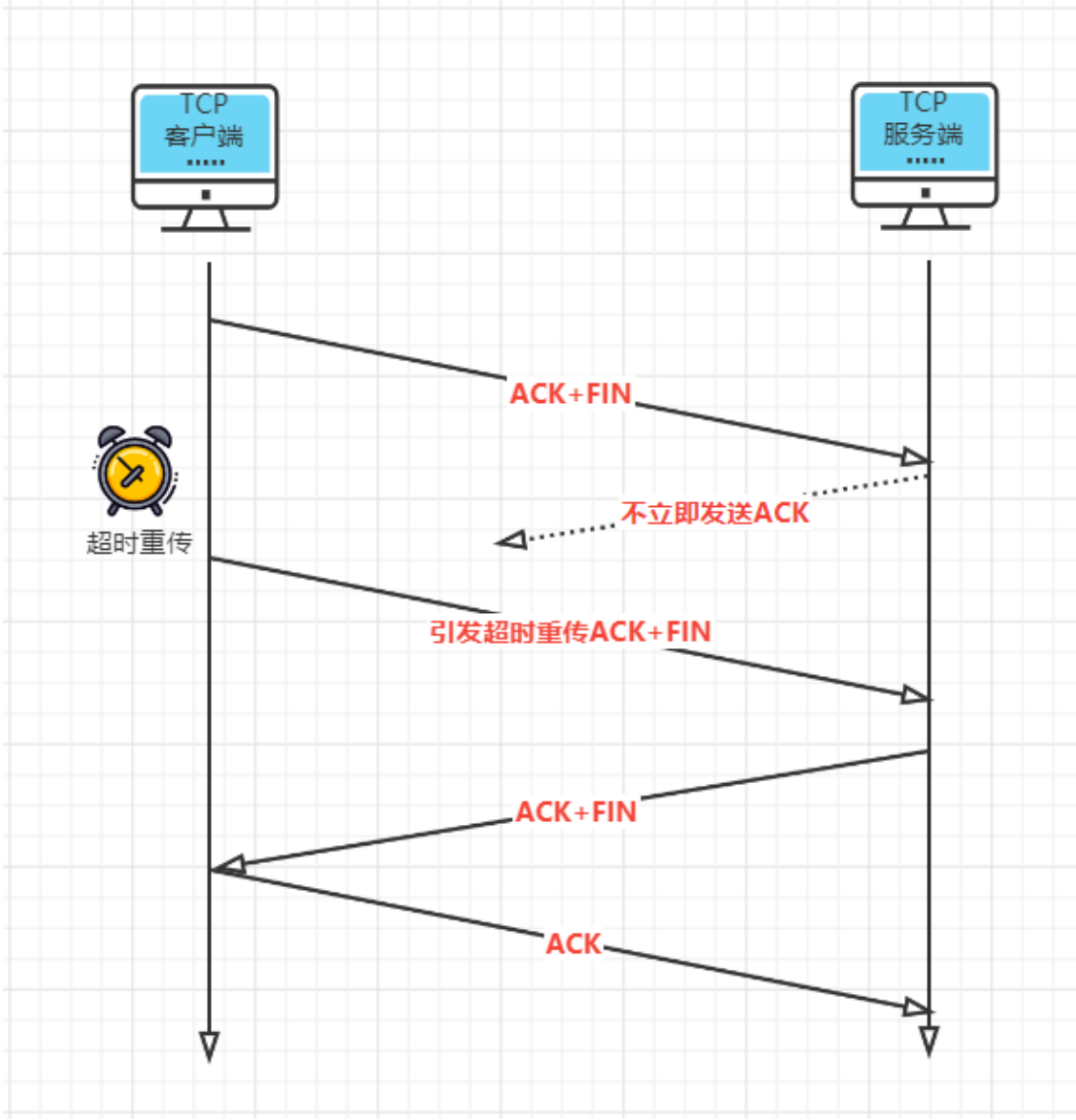
客户端发送 FIN 包以后，会进入半关闭（half-close）状态，表示自己不会再给对方发送数据了。

如果服务端确定没有什么数据需要发给客户端，那么当然是可以把FIN和ACK合并成一个包，四次挥手的过程就成了三次，如下图所示：



不过往往是客户端数据发完了，服务端还没来得及发送完全部的数据，因此TCP给主动发起释放连接的一方设置了这么一个半关闭的状态，假设还是客户端主动发起释放连接的，那么这个状态下，客户端不会再给服务端发送数据了，但是仍然需要可以接收来自服务端发来的数据。

在这种情况下，如果服务端不及时回复ACK，等所有数据发完才一起回复ACK+FIN的话，很可能会造成客户端不必要的重发 FIN 包，如下图所示。

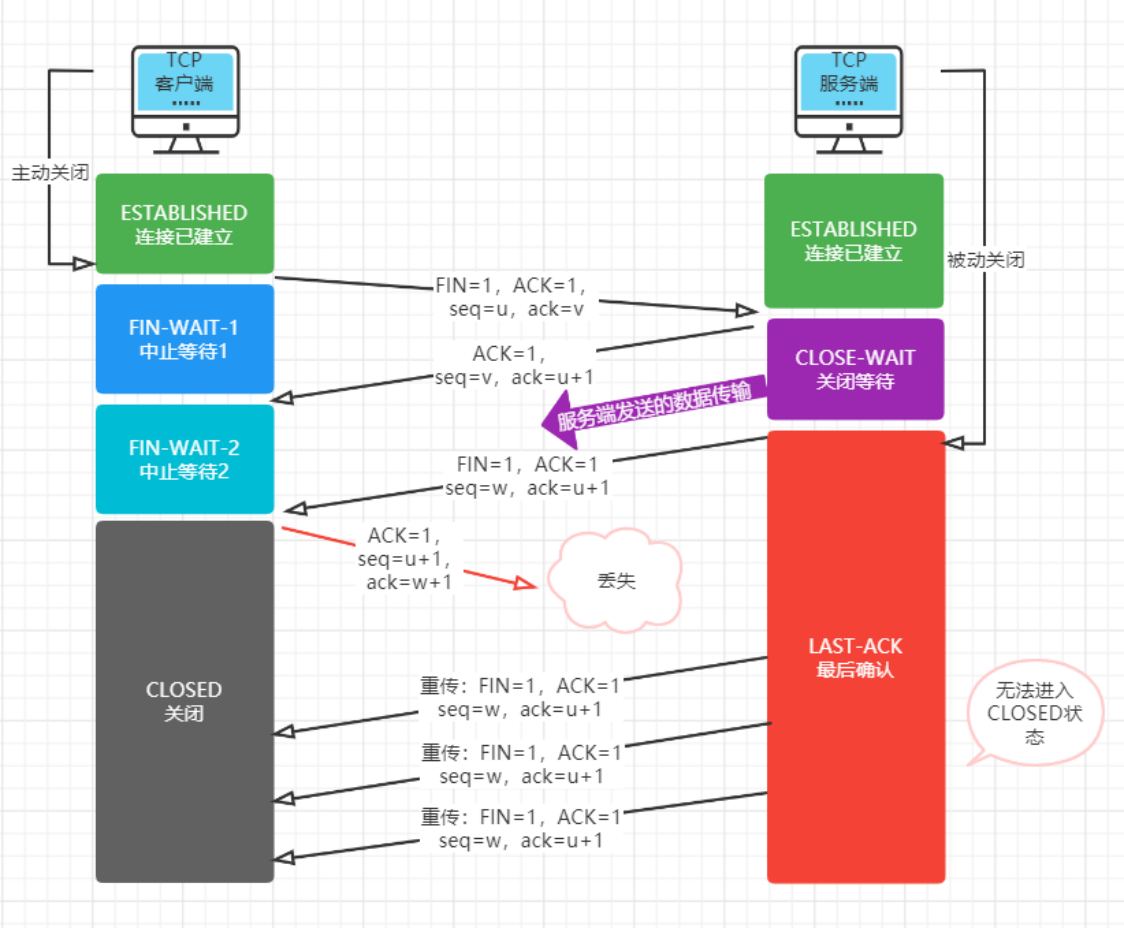


所以一般情况下，还是四次挥手比较合理。

四、为什么要有时间等待状态

想下，这个时间等待状态真的有必要吗？客户端给服务端ACK后为什么不直接进入关闭状态呢？

客户端收到服务端发送FIN报文段后，直接进入关闭状态，并回复一个普通的ACK确认报文段，然而，**该ACK报文段丢失了，这必然会造成服务端对之前所发送的FIN报文段的超时重传**，客户端此时已经处于关闭状态，不会理睬该重传的FIN报文，必然还会造成服务端的反复重传FIN报文段，**从而导致服务端迟迟无法进入关闭状态**。



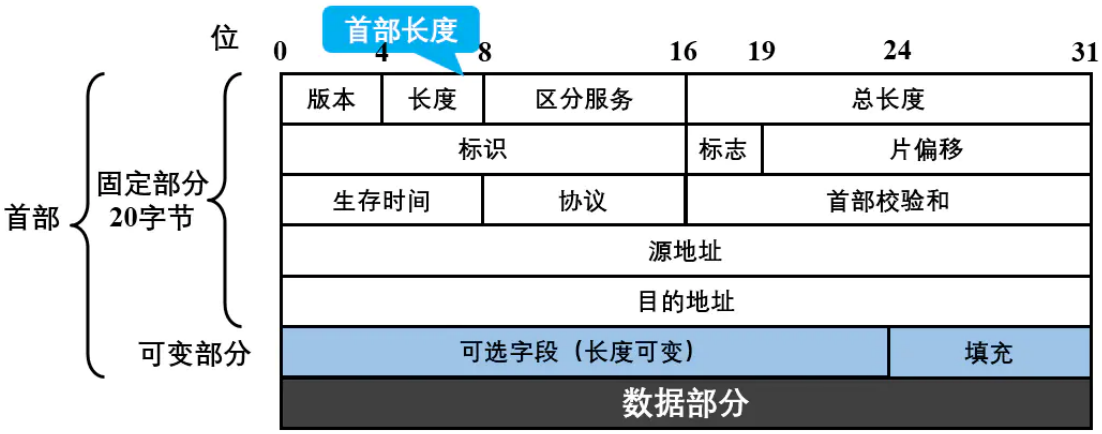
> 在A关闭连接后，B重传FIN的次数有上限，所以超过了上限B就会reset连接，所以我们要尽可能地正常关闭，而不是这种异常关闭，毕竟异常关闭是兜底操作，比较浪费时间，且这个过程中，服务端一直处于 LAST_ACK 状态。

此外，在这种情况下，**主动关闭方认为连接已经释放，端口可以重用了，如果使用相同的端口三次握手发送 SYN 包，会被处于 LAST-ACK状态状态的被动关闭方返回一个 RST，三次握手失败。**

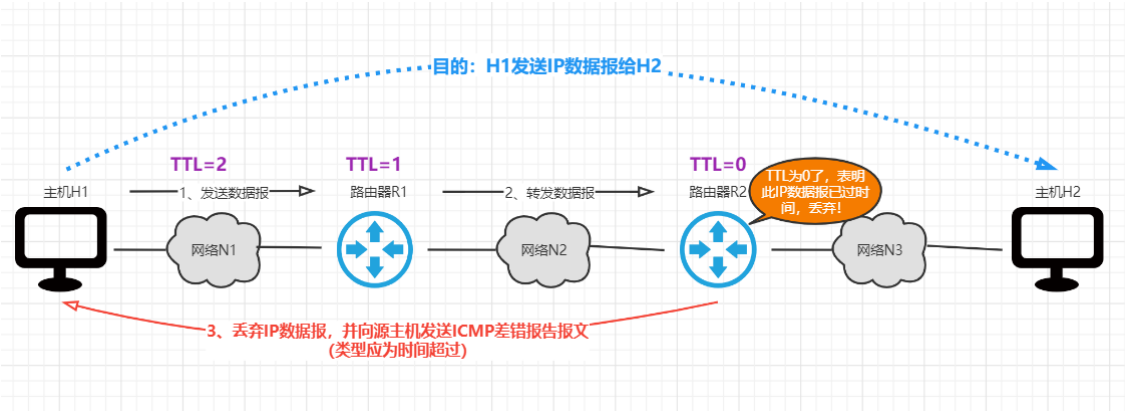
五、为什么要等待2MSL?

MSL（报文最大生存时间）是 TCP 报文在网络中的最大生存时间。这个值与 IP 报文头的 TTL 字段有密切的关系。

我们来回顾下IP首部结构。



其中，生存时间：占8位，表示数据报在网络中的寿命。由发送数据报的源点设置这个字段，其目的是为了防止那些无法交付的数据报无限制的在互联网中兜圈子（例如从路由器R1转发到R2，再转发到R3，然后又转发到R1），因而白白浪费网络资源。数据报每经过一个路由器，这个值就会减1，当减至0时，就丢弃该数据报。



从上面可以看到 TTL 说的是「跳数」限制而不是「时间」限制，尽管如此我们依然假设最大跳数的报文在网络中存活的时间不可能超过 MSL 秒。Linux 的套接字实现假设 MSL 为 30 秒，因此在 Linux 机器上 TIME_WAIT 状态将持续 60秒。

既然报文最大存活时间不会超过MSL的值，那么就可以解决第一个问题：**数据报文可能在发送途中延迟但最终会到达，因此要等老的“迷路”的重复报文段在网络中过期失效，这样可以避免用相同源端口和目标端口创建新连接时收到旧连接姗姗来迟的数据包，造成数据错乱。**

TIME_WAIT 等待时间是 2 个 MSL，已经足够让一个方向上的包最多存活 MSL 秒就被丢弃，保证了在创建新的 TCP 连接以后，老连接姗姗来迟的包已经在网络中被丢弃、消失，不会干扰新的连接。

解决的第二个问题是：我们上面说明了，当最后客户端的ACK丢失后，会引起服务端的FIN报文段的超时重传，**那么这里等待一个MSL可以确保对端没有收到 ACK 重传的 FIN 报文可以到达。**

综上所述，要2MSL的原因为：

- 1 个 MSL 确保四次挥手中主动关闭方最后的 ACK 报文最终能达到对端。
- 1 个 MSL 确保对端没有收到 ACK 重传的 FIN 报文可以到达。
- 并且这个时间足够让两个方向上的包最多存活MSL秒就被丢弃，保证了在创建新的TCP连接以后，老连接姗姗来迟的包已经在网络中被丢弃消失，不会干扰新的连接。

因此2MS = 去向 ACK 消息最大存活时间（MSL）+ 来向 FIN 消息的最大存活时间（MSL）