答疑（二）| 第6~10讲思考题答案

2022-03-16 杨胜辉

《网络排查案例课》 课程介绍 >

讲述：杨胜辉

时长 14:28 大小 13.25M

你好，我是胜辉。

在上一讲里，我们回顾了第 1 到第 5 讲的思考题，分析了每道题目的解题思路，当然更重要的是对这五节课的知识点做了一次复习和补充。你看完了这些解答后的感觉如何呢？跟之前你自己的思考过程和结论相比，又有哪些相同和不同之处呢？相信通过再一次的思考，你对知识的掌握将会更加深刻。

那么这节课，我们就继续讲解这些课后的思考题。先从第 6 讲开始。

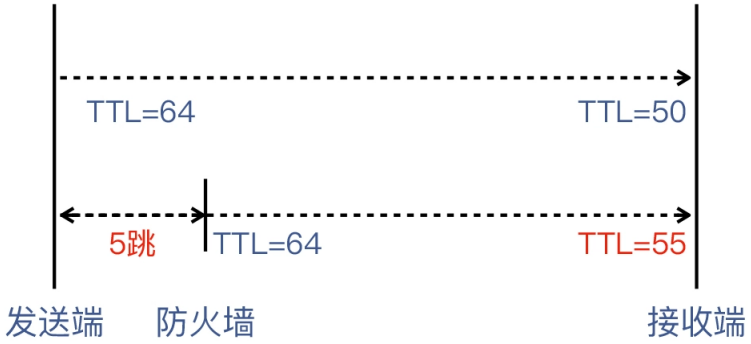
06 讲的答疑

思考题

这节课，我给你介绍了利用防火墙的 TTL 跟原先的正常报文的 TTL 不一致的特点，从而识别出防火墙的方法。那么，假设有一天，防火墙公司把这个特性也完善了，我们再也不能仅凭 TTL 的突变而发现防火墙了，你觉得还有什么办法可以识别出防火墙吗？

答案

这是一个开放式的问题，其实并没有标准答案。不过，通过对这个问题的思考，我们可以对网络的理解更进一步。我们先参考下面这张图，复习一下用 TTL 判断防火墙的原理：



极客时间

> 补充：上图是一个例子，假设防火墙是路径上的第 5 跳，那么防火墙自己发出的报文的 TTL 就比真正的发送端的 TTL 多了 5。

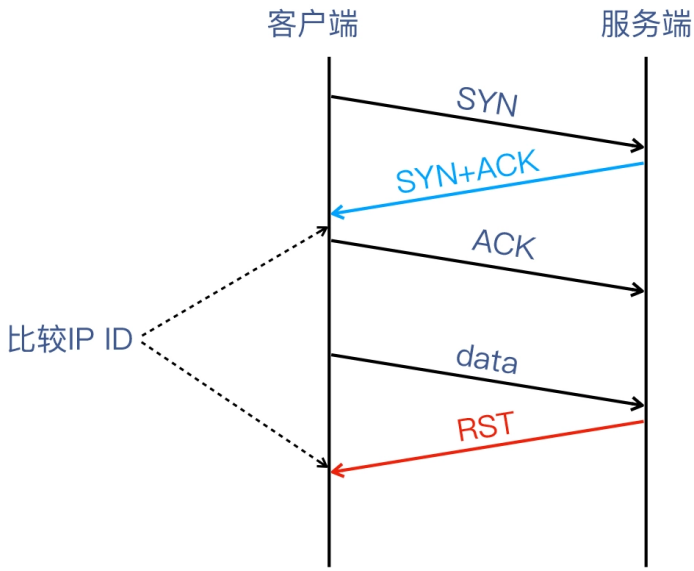
然后想一想，我们之所以能通过 TTL 定位防火墙，本质原因是什么呢？实际上，本质原因就是这个 IP 报文是防火墙自己构造的，而**在构造时 TTL 被设置为了初始值**。

那么，还有哪些元信息也是在这个构造好的报文里呢？其实这些信息也可以帮我们辅助判断防火墙。

另外在 IP 层，除了 TTL，还有一个字段也值得我们注意，它就是 **IP ID**。这个字段占用了 2 个字节，所以它的最大值是 2^16-1，也就是 65535。IP ID 一般是 IP 报文的发起端设置的，每次发包递增 1，然后在 0 到 65535 之间循环使用。

它最主要的用途，是**接收端会根据这个 ID 来组装（assemble）同一组 IP 分片（fragments），**而中间设备（交换机、路由器）不会改动 IP ID。并且在防火墙构造 RST 报文的时候，IP 包的 TTL 和 ID 值也是防火墙自己设置的，也就有可能被我们用来发

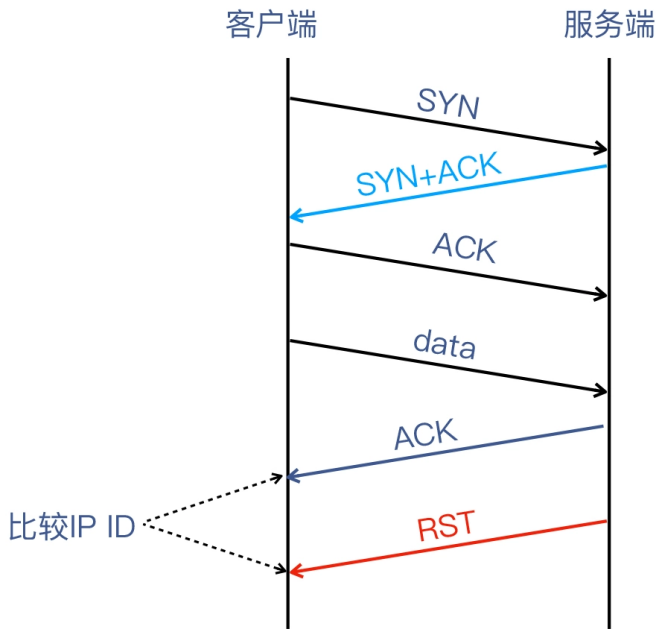
现防火墙。所以，如果 SYN+ACK 和 RST 的 IP ID 不连续，我们就可以判断出防火墙，也就是下图这样：



不过，我在探究这个可能性的时候，又遇到了新的问题。要知道，Linux 系统发送 SYN+ACK 的时候，IP ID 一定是 0，但是我既在 Wireshark 里看到了这个现象，也在内核代码里证实了这一点。而这就给我们利用 IP ID 带来了障碍：**因为即使 RST 真的是对端发出的，两个报文的 IP ID 也会不同。**

只有当我们收到至少三个报文，也就是除了 SYN+ACK 和 RST 之外，我们还需要在中间收到一些报文，才可以对比 RST 跟中间报文的 IP ID 的差别，然后进行判断。

所以，利用 IP ID 的方法，对于 TCP 握手后就发生 RST 的情况并不适用，但对于交互过程中发生 RST 的情况应该是适用的。你有机会也可以在遇到 RST 时，来抓包验证一下。



极客时间

07 讲的答疑

思考题

1. tcp.payload eq abc，这个过滤器可以搜索到精确匹配“abc”字符串的报文。那么，如果是模糊匹配，比如只要包含“abc”的报文我都想搜到，这个过滤器又该如何写呢？
2. 在工作中，你遇到过跟 TCP Keep-alive 相关的问题吗？你是怎么解决的呢？

答案

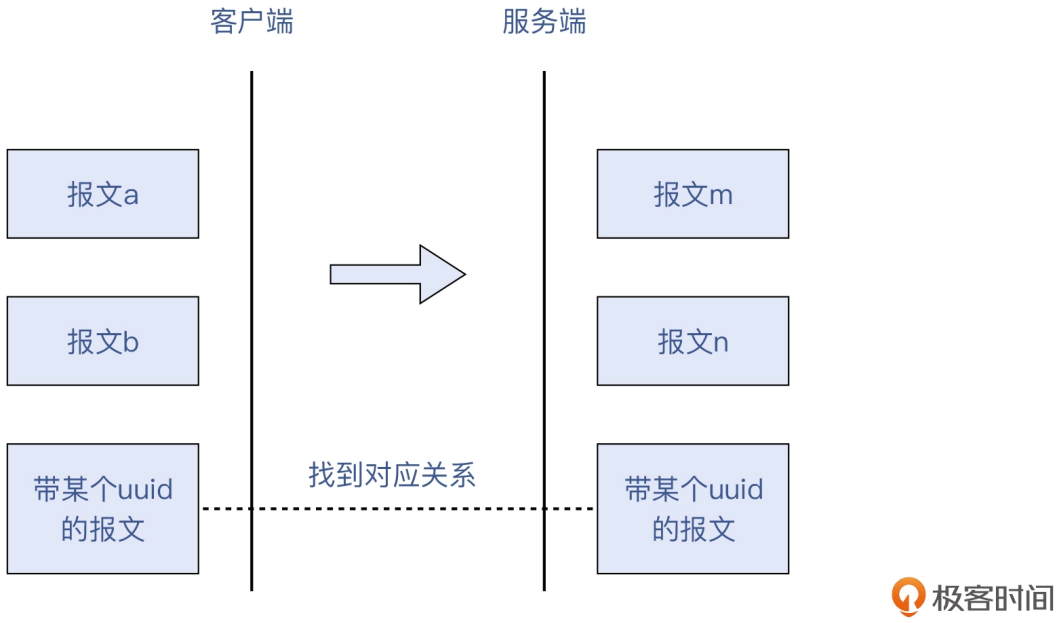
第一个问题，其实在上节答疑课里我也提到过，就是利用各层的 **contains** 动词来写过滤器。比如：

> 应用层可以是 `http contains "abc"`；
>
> 传输层可以是 `tcp contains "abc"`；
>
> 网络层可以是 `ip contains "abc"`；
>
> 数据链路层可以是 `frame contains "abc"`。

这类过滤器的使用场景一般是这样的：我们知道应用层的某一个信息，比如 HTTP 报文里面的某个 uuid，然后再通过这个信息，去找到对应的报文。

而在这里，又可以细分为两种不同的场景。

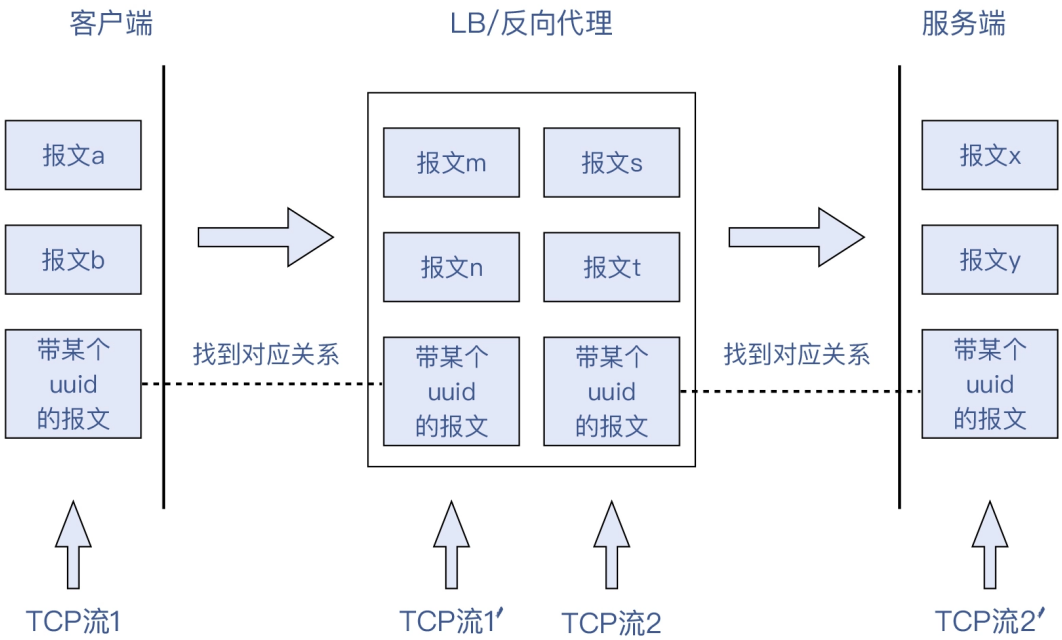
第一种，某一个 TCP 连接在通信两端的报文的对应。比如，客户端发出的 HTTP 报文里含有某个 uuid，那么我们需要在服务端的抓包文件里，把这个 TCP 流过滤出来，就可以用上面说的那些过滤器。



第二种，LB 前后方两个不同 TCP 连接的报文的对应。比如，LB 会转发请求给后端多台服务器中的某一个，而这个选择一般是动态的，这就给我们的排查工作带来了不小的障碍。

那么这里可以参考的方法就是：在“客户端 <->LB”这一侧的连接中，找到 HTTP 报文的某个 uuid，然后在“LB<-> 多个后端 IP”这一侧的抓包中，搜索这个 uuid，就能得到同样是这个请求的相关报文了。

我们可以看看下面这个示意图：



也就是说，TCP 流 1 和 TCP 流 1'，还有 TCP 流 2 和 TCP 流 2'，分别是同一个 TCP 流在连接两侧的不同体现。而**由于流 1 和流 2 确实是完全不同的连接，无法用任何 TCP 报文头部的信息来进行匹配，所以只能用应用层的信息来找到对应关系**。

第二个问题也是一个开放式的问题。我看到不少同学都分享了他们的经验，这也增加了我自己的见识，真是一件多赢的事。比如，**@江山如画**同学对这个问题的回复是这样的：

> 在工作中，我目前还没有碰到过和 TCP Keep-alive 有关的问题，不过我之前看一些建立网络隧道的软件里，有 Keep-alive 这个参数，查询说它和 SNAT 有关，就让我一直很迷惑。
>
> 今天学习了老师的课，我又去深究了下。原来 Virtual Tunnel 会建立虚拟网卡，它和物理网卡之间需要做流量的桥接，进而就需要做 SNAT。然后 SNAT 会维护一个端口映射表，因为链接太多可能会占满本地端口，如果没有设置 Keep-alive，那么在一段时间内都没有传输数据的话，就会把端口转换的记录给删掉，这时候通信双方想再通信就不行了。所以需要每隔一段时间发送心跳包，保证 SNAT 端口映射表中的记录不被删掉，从而保证连接存活。

这实际上就是一个很典型的场景，也就是 NAT 和 TCP Keep-alive 之间的关系。NAT 是维护连接跟踪表的，里面的表项有一定的有效期，过了有效期的表项就会被清空。而 TCP

Keep-alive 正好可以定期去刷新这个计时器，让它“永葆青春”，表项不被删除，这个连接就可以一直愉快地工作下去了。

类似地，**@Geek_535c45** 同学也对 TCP Keep-alive 问题发出了灵魂的拷问：

> Chrome 每隔 45 秒发起 TCP Keep-alive 包的意义是什么呢？

其实答案也跟上面的解释差不多，Chrome 每隔 45 秒的 TCP Keep-alive，主要是起到下面这两个作用：

避免连接被中间环节给撤销掉。一般客户端是在内网里的，出公网的时候 NAT 会转换为公网 IP 跟对端通信。而跟上面的例子类似，为了保持 NAT 表项不被回收，所以 Chrome 要发送保活报文。

避免被服务端认为是无效连接给撤销掉。服务端一般也有 idle timeout 时间，也就是如果一个客户端连接超过一定时间没有报文传送，服务端就会取消这个连接，而且这时很可能不发送 FIN 或者 RST，导致客户端对连接失效这个事情并不知情。因为服务端要面向公网上成千上万的访问者，所以它的逻辑是定时清除无效连接，以保证服务端的资源得到合理的使用。

08 讲的答疑

思考题

1. 在 LB 或者网关上修改 MSS，虽然可以减小 MSS，从而达到让通信成功这个目的，但是这个方案有没有什么劣势或者不足，也同样需要我们认真考量呢？你可以从运维和可用性的角度来思考。
2. 你有没有遇到过 MTU 引发的问题呢？

答案

对于第一个问题，在 LB 或者网关上修改 MSS，这当然起到了作用。从短期来看，也是很不错的临时方案。但是长期来看，一旦客户和自己发生人员流动，又缺乏文档记录，就很可能会引起问题。比如说，这个配置可能会被后来的人当作无用配置而清除，而到时候如果没有人了解这个上下文，显然就要花很多时间去排查，给双方带去的业务影响也不会小。

尽管这可能算是一个非技术问题，但是也会影响到自己团队和兄弟团队的工作体验。所以我们做运维工作，还有一个重要的原则：**优先选择简洁又容易维护的方案**，而不是复杂且不易维护的方案，避免给以后的运维工作埋下隐患。

所以，后来我们选择的长期方案还是在两端修改 MTU，这样的话客户团队就很清楚自己机器上的配置，就有责任也有条件做好维护。

第二个问题，还是 **@江山如画**同学，他分享了一个典型案例：

> 之前我们手动创建了虚拟网卡，和物理网卡之间做了流量的桥接，发现有些报文在这二者之间转发时会被丢掉。然后通过分析，我们发现虚拟网卡的 MTU 设置过大，并且报文 DF 位设置为了 1，后来我们通过 ifconfig 命令把虚拟网卡的 MTU 改小，报文就可以正常转发了。

不用我做更多解释，你应该都很清楚这个场景里的技术细节了。这也是比较常见的情况，你掌握好以后应该也有机会运用到实际工作中，也就是说，**如果遇到丢包的问题，你可以看看 MTU 的值是否超标了**。这是一个比较合算的“投资”，就花几分钟查一下，也许真能解决一个原本要查几个小时的大问题。

09 讲的答疑

思考题

你有没有在工作中遇到过 TCP 传输速度相关的问题呢？通过这节课的学习，你已经掌握了传输速度相关的不少知识，你准备怎么运用这些知识，来解决这个传输问题呢？

答案

我看到留言中有一个同学的回答是这样的：

> 我们公司之前有一个场景是多个客户端连接同一个服务端，如果某个客户端下载文件或上传文件，占用了大量的带宽，就会导致新的客户端连不上服务端。后来我们就在服务端使用 tc 工具做了流控，根据客户端个数来均分带宽，还有限制单个 IP 的最大使用带宽。

其实， tc 的原理是调整 qdisc 发送缓存队列，超出队列的数据就丢弃，这样的话发送端就探测到了“拥塞”，进而会主动降速，从而也就达到了限制速度的效果。而“拥塞”这个知识点，又是我们在第 11 讲里深入探讨过的内容。其实从第 9 到 13 讲的内容都是关于传输的，所以我建议你可以把第 9 到 13 讲的内容结合起来学习，应该就能更加深入和全面地理解 TCP 传输这个重要话题了。

10 讲的答疑

思考题

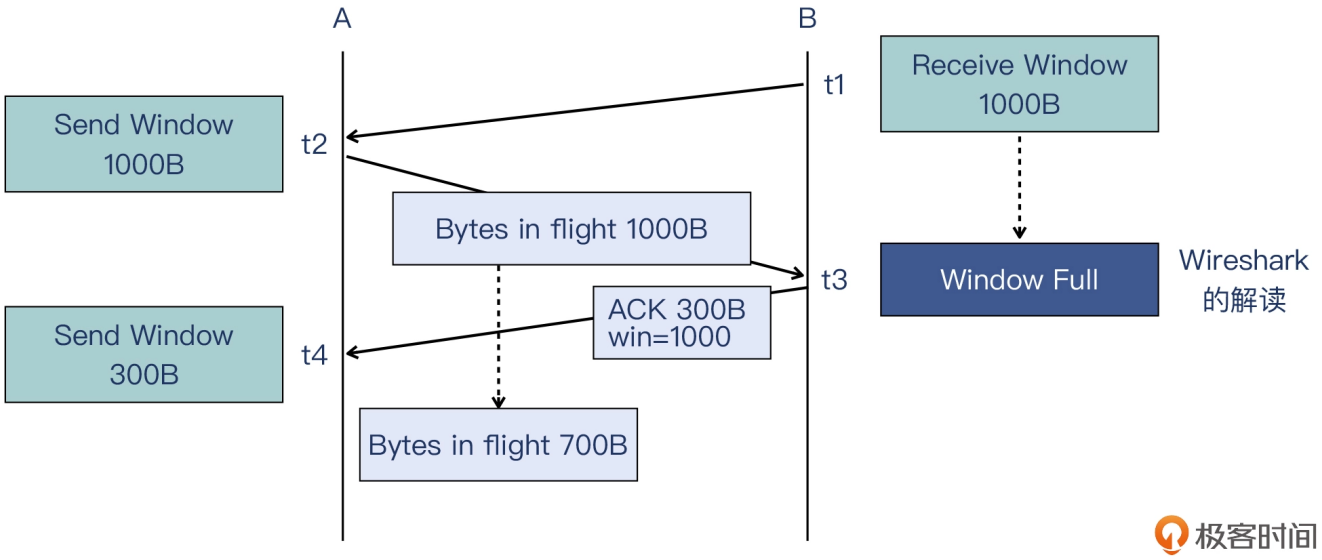
1. TCP 的序列号和确认号，最大可以到多少？
2. 接收端只确认部分数据，导致了“数据滞留”现象，这个现象背后的原因可能是什么呢？

答案

第一个问题呢，其实是一个简单的**协议规范的问题**。这个知识点，你很容易就可以在RFC793里找到。TCP 的序列号和确认号的长度都是 4 个字节，4 个字节也就是 32 位，所以最大值是 2^32-1，也就是 4G。我也建议你记住这些小的知识点，因为它是在工作中高频使用的知识点，你能记住的话就不用去搜索了，可以提升工作的效率。

第二个问题就比较复杂了，而且因为当时的现场没有了，所以也没有标准答案。但是这个思考的过程对我们掌握这一讲的知识是很有帮助的，而且，我们其实还是可以大致推导出当时的情景。

我直接借用第 10 讲里面的一张示意图：



我也来简单描述一下这个过程。

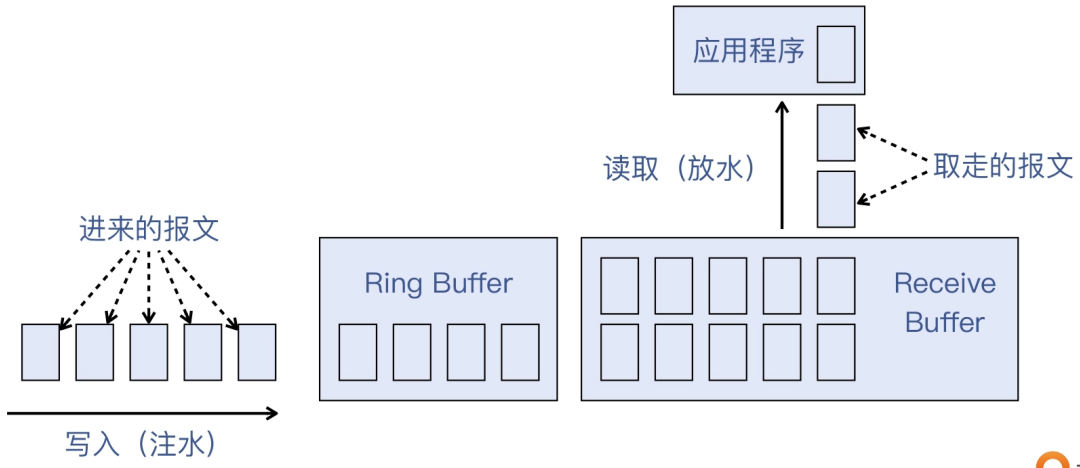
在 t1 这个时间点，B 通知 A：“我的接收窗口是 1000 字节”。

在 t2 这个时间点，A 收到了 B 的通知，然后发送了 1000 字节，此时这些都属于在途字节数。

在 t3 这个时间点，B 收到了 1000 字节，此时接收窗口（缓冲区）全部用完，Wireshark 据此判断出“Window Full”。然后 B 向 A 发送确认包，确认了 300 个字节。

在 t4 这个时间点，A 收到 B 确认 300 个字节的确认包，于是**判断出在途字节数是 700 字节，所以 A 能发的最多只有 300 字节。**

我们再来看一下接收端的报文处理流程。如果我们把缓冲区（包括 Ring Buffer 和 Recieve Buffer）比作水池，那么报文进入缓冲区，就相当于有水龙头往里注水；应用程序到缓冲区取走数据，又相当于有水龙头放出水。这两个水龙头的速度的不同，就造成了缓冲区的动态调整。



极客时间

那么，发生数据“滞留”，其实就相当于水池中一直囤着水，也就是“放水速度不够”，应用程序没有把数据都及时读取走导致的。

显然，比较理想的情况是放水的速度大于等于注水的速度，这是传输速度最快的方式。

但是，我们也要认识到，操作系统设立缓冲区的一大原因，就是考虑到“注水”和“放水”的速度本来就是经常不相等的，所以用缓冲区这个“空间”，来抵消速度不同导致的“时间”上的矛盾。简单来说就是“**用空间换时间**”。而且，不仅是网络接收和发送的缓冲区如此，大部分缓冲区也都是“用空间换时间”原则的体现。

所以通过这个思考，你对 TCP 的缓冲区等概念，是否也有了新的认识了呢？

好了，今天的答疑就到这里。如果你还有什么疑问，同样可以回复到留言区，我们一同进步、成长。

上一篇 答疑（一）| 第1~5讲思考题答案

下一篇 答疑（三）| 第11~15讲思考题答案

精选留言 (1)

写留言

那时刻

2022-03-17

请问老师例子中在途字节数是 700 字节，是因服务器window full直接丢弃？还是有可能服务器把缓冲区数据被应用程序处理后再次接收呢？

作者回复: 在途字节数700字节，是发送端自己判断出来的，是根据"已发送数据 - 被确认数据"得出的。
而Window Full是Wireshark根据报文情况作出的解读，这个信息表示：在被标记的时间点，（服务端的）接收窗口已经被用完。这里不会有丢弃的行为。
在服务端确认了300字节的数据后，发送端就知道，虽然另外700字节还未被确认（因此仍属于在途数据），但是窗口已经空出了300字节了，于是可以继续发送300字节。
在服务端应用程序从缓冲区取走更多数据后，发送端就可以发送更多字节。
概括来说，在TCP里，发送端是不会发出超过对方缓冲区大小的数据的（恶意的除外）。