
# 一条sql语句慢在哪之抓包分析

Great DB 万里数据库

01 抓包前我们需要知道的武功秘籍

02 如何使用tcpdump抓取所需之包

03 如何分析数据包抓取想要的结果

04 如何分析优化业务中接口慢问题

05 使用第三方工具傻瓜式分析

# 思考？

1、计算机编码为什么只识别0和1，数据也是以0和1组成编码存储的

2、计算机之间是如何交流的，我们通过应用程序发送的信息如何接收的

3、抓包抓的是什么？需要看那些内容?

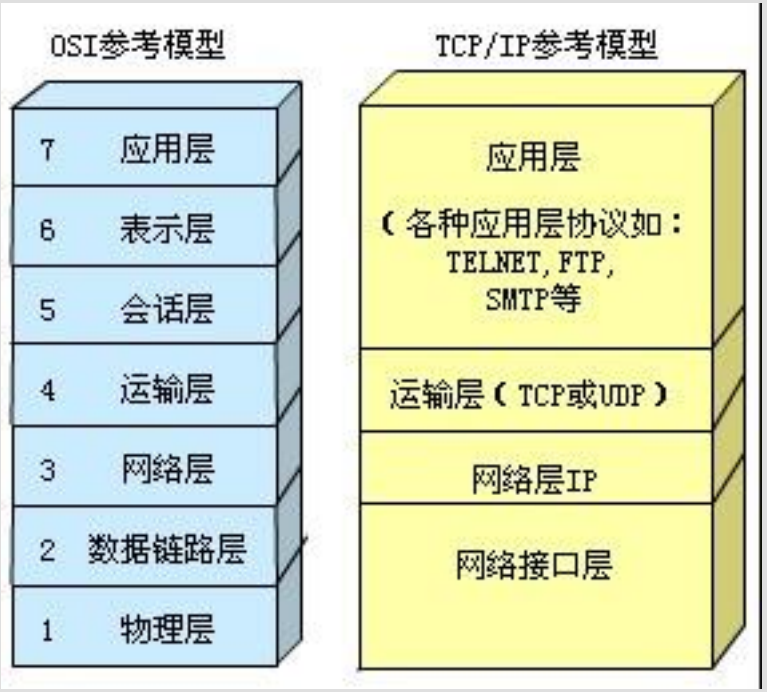
4、sql响应慢但不是数据库的原因该如何甩锅

5、sql慢到底如何排查及优化

Great DB
万里数据库

# 01 抓包前我们需要知道的武功秘籍

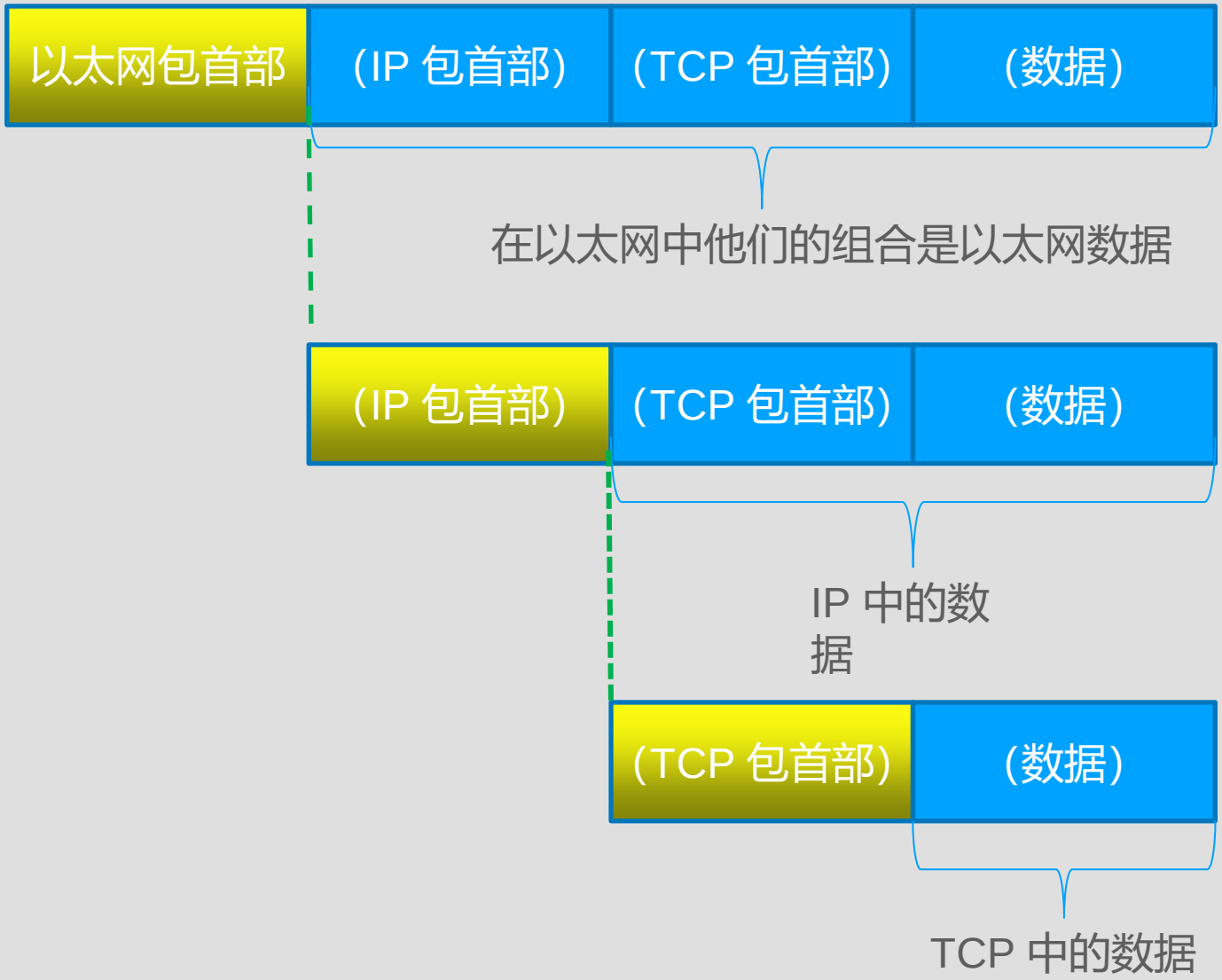
# 为什么会有TCP/IP协议




在世界各地，各种各样的电脑运行着各自不同的操作系统为大家服务，例如：windows系统、linux、mac等。
这些电脑在表达同一种信息时所使用的方法也是千差万别，为了将不同的电脑或操作系统进行连接（类似于将人聚集到一起，形成一个村落），方便交流。于是产生了TCP/IP，TCP/IP不是一个协议，而是一个协议栈的统称，包含IP协议、IMCP协议、TCP协议，以及我们更熟悉的http、ftp、pop3协议等等。
电脑终端有了这些协议后，就像学会了外语，可以和其他的电脑终端进行自由交流及数据传输

# 数据包的传输流程-网络七层协议

# 每个分层不同的数据包类型




1、网络中传输的数据包由两部分组成：一部分是协议所要用到的首部，另一部分是上一层传过来的数据。

2、每个分层中，都会对所发送的数据附加一个首部，在这个首部包含了该层必要的信息，例如：发送的目标地址以及协议相关信息。通常，为协议提供的信息称为包首部，所要发送的内容为数据。

3、在下一层的角度看，从上一层收到的包全部都被认为是本层的数据。

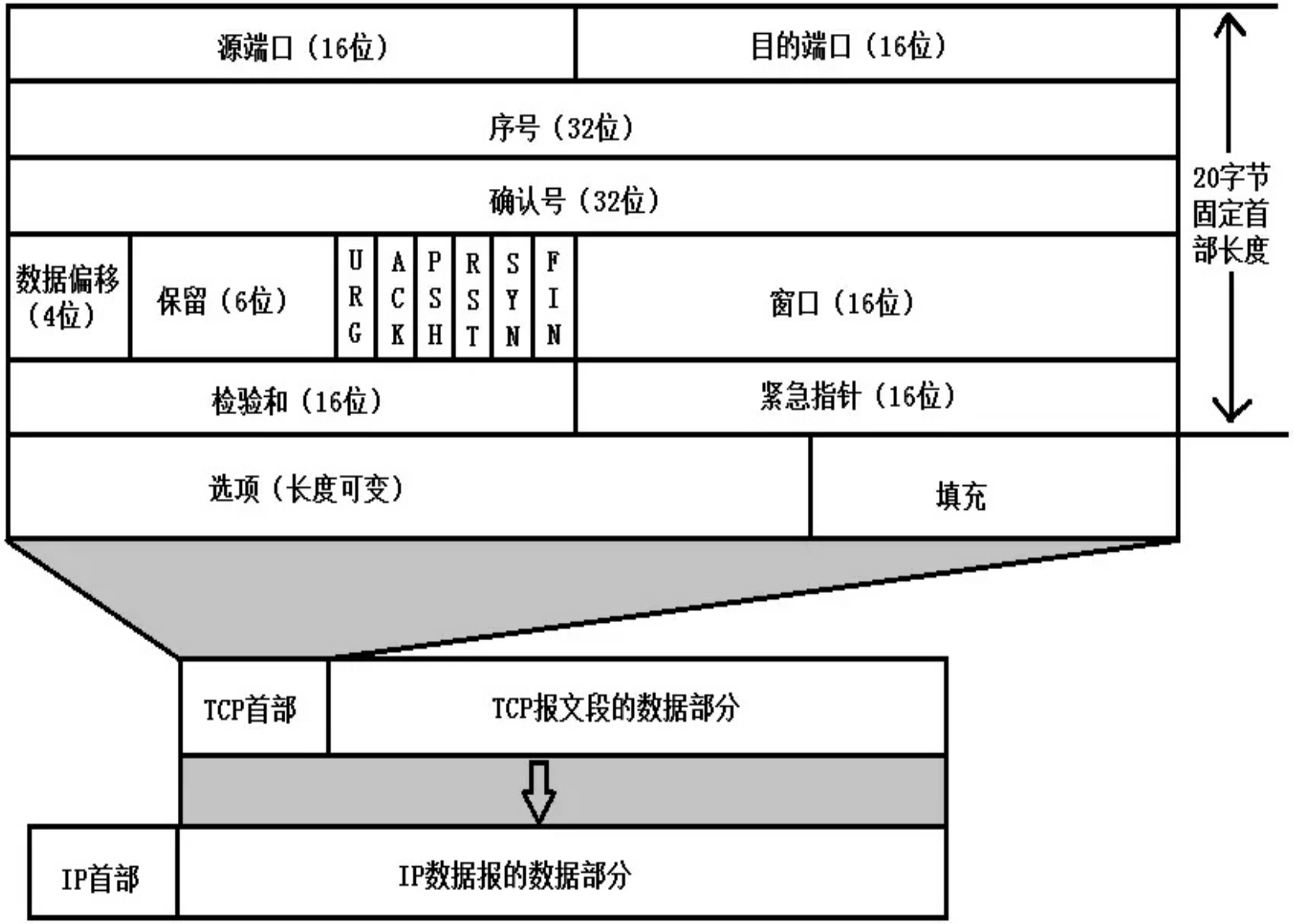
4、首部的结构由协议的具体规范详细定义。在数据包的首部，明确标明了协议应该如何读取数据。也就是说看到首部就能了解该协议必要的信息以及所要处理的数据

# 不同分层数据包封装


```
2021-09-09 11:22:40.874228  2.151320000 172.16.50.161  172.16.50.54   TCP  82 52152 → 16310 [PSH, ACK] Seq=285 Ack=511 Win=30336 Len=16 TSval
2021-09-09 11:22:40.889643  0.015415000 172.16.50.54   172.16.50.161  TCP  277 16310 → 52152 [PSH, ACK] Seq=511 Ack=301 Win=30080 Len=211 TSva
```

```
> Frame 734: 82 bytes on wire (656 bits), 82 bytes captured (656 bits)
v Ethernet II, Src: PcsCompu_19:3a:00 (08:00:27:19:3a:00), Dst: PcsCompu_d0:bd:f1 (08:00:27:d0:bd:f1)
  > Destination: PcsCompu_d0:bd:f1 (08:00:27:d0:bd:f1)
  > Source: PcsCompu_19:3a:00 (08:00:27:19:3a:00)
    Type: IPv4 (0x0800)
> Internet Protocol Version 4, Src: 172.16.50.161, Dst: 172.16.50.54
v Transmission Control Protocol, Src Port: 52152, Dst Port: 16310, Seq: 285, Ack: 511, Len: 16
    Source Port: 52152
    Destination Port: 16310
    [Stream index: 0]
```

# TCP协议包结构




源目端口：使用4个字节来标识发送方端口及接收方端口，即谁发送，谁接收。

序号（seq）：tcp时面向字节流的，一个tcp连接中传输的字节流中每一个字节都是按顺序编号。整个要传送的字节流的起始序号必须在连接建立时设置。首部中的序号字段值则是指本报文段所发送的数据的第一个字节的序号。长度为4字节，序号是32bit的无符号数，序号到达2^32 -1后从0开始

确认号（ack）：发送确认方所期待收到的下一个序号。确认序号为上次接受的最后一个字节加1，ack为1时，确认序号才有效

数据偏移：也叫首部长度，占4个bit，他指出TCP报文段的数据起始处距离TCP报文段的起始处有多远

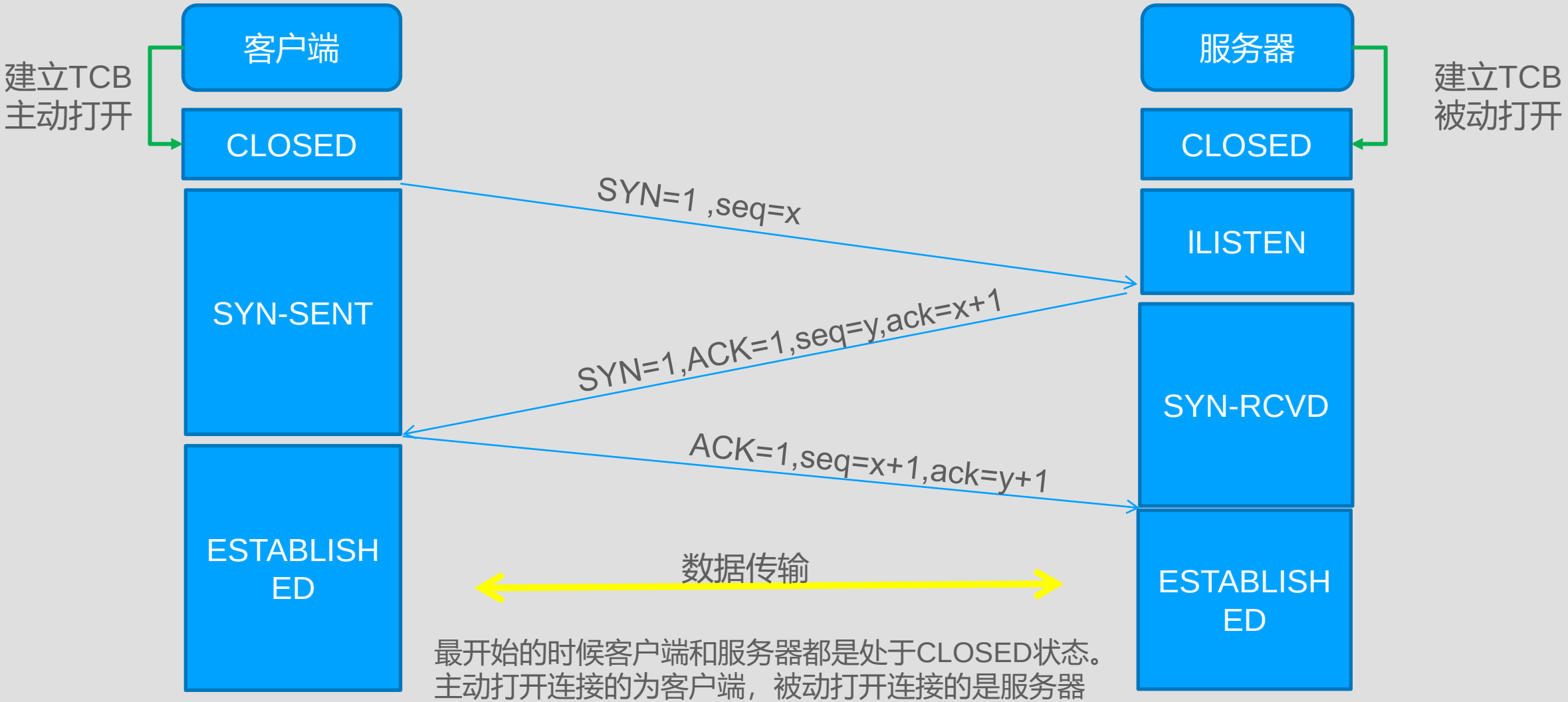
窗口：告诉对方一次发送的数据量（以字节为单位）窗口字段明确指出现在允许对方发送的数据量。窗口值经常在动态变化

校验和：校验和覆盖了整个tcp报文段：TCP首部+TCP数据

# TCP协议包结构-状态码

**紧急URG（urgent）紧急指针字段：**
当URG为1时，表明紧急指针字段有效，告诉系统此报文中有紧急数据，应尽快发送（相当于高优先级的数据），而不是按原来的排队顺序来传送。
**确认ack（acknowledgment）**
仅当ack=1时确认号字段才有效，当ack=0时，确认号无效。TCP规定，在连接建立后所有的传送报文字段都必须设置为1
**推送PSH（push）**
当两个应用程序进行交互式通信时，如果发送大的文件或者图片等，一次可能传送不完，则发送方会将文件切割成多个数据段进行发送，接收方收到后，会将同一个数据包的数据段进行缓存，等都接收完成后，才会将接收的数据段推送给应用层（当缓存区满了也会推送）。发送方TCP把PSH置为1，并立即创建一个报文段发送出去。接收方TCP收到PSH=1的报文段，就尽快交付给接收应用程序
**复位RST（reset）**
当RST=1时，表明TCP连接出现了严重错误（例如：主机崩溃或其他原因），必须释放连接，然后在重新建立传输连接。RST置为1，还用来拒绝一个非法的报文段或打开一个连接
**同步SYN（synchronization）**
在连接建立时用来同步序号。当SYN=1而ACK=0时，表明这是一个连接请求报文段，若对方同意建立连接，则应在响应的报文段中使SYN=1和ACK=1
**终止FIN（finis，意思使“完”“终”）**
用来释放一个连接。当FIN=1时，表明此报文段的发送方发送的数据完毕，并要求释放连接

# TCP 三次握手




最开始的时候客户端和服务器都是处于CLOSED状态。
主动打开连接的为客户端，被动打开连接的是服务器

# TCP 三次握手

为什么TCP客户端最后还要发送一次确认呢?

答：主要是防止已经失效的连接请求报文突然又传送到了服务器，从而产生错误。

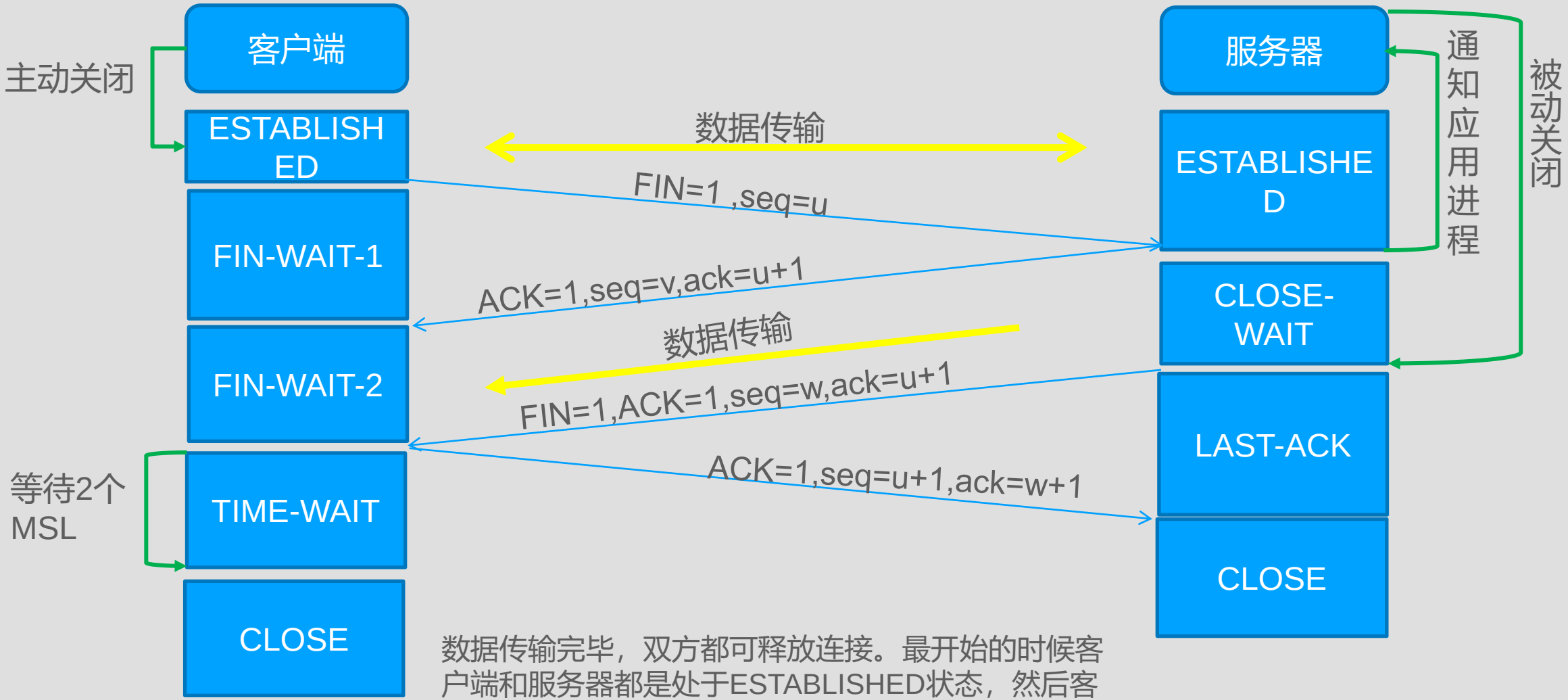
如果使用两次握手建立连接，假设如下场景：客户端发送了第一个请求连接并且没有丢失，只是因为网络节点中滞留的时间太长了，由于TCP的客户端迟迟没有收到确认报文，以为服务器没有收到，此时重新向服务器发送这条报文，此后客户端和服务器经过两次握手完成连接，传输数据，然后关闭连接，此时滞留的那一次连接，网络畅通后到达了服务器，这个报文本该是失效的，但是两次握手的机制将会让客户端和服务器再次建立连接，这将导致不必要的错误和资源浪费。如果采用的是三次握手，就算是那一次生效的报文传送过来了，服务端接收到了那条失效报文并且回复了确认报文，但是客户端不会再次发出确认。由于服务器收不到确认信息，就知道客户端并没有请求连接

# TCP 三次握手


| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2021-08-31 17:06:56.996776 | 0.000000000 | 172.16.50.161 | 172.16.50.54 | TCP | 74 | 57506 → 16310 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 |
| 2 | 2021-08-31 17:06:56.996859 | 0.000083000 | 172.16.50.54 | 172.16.50.161 | TCP | 74 | 16310 → 57506 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 S |
| 3 | 2021-08-31 17:06:56.997162 | 0.000303000 | 172.16.50.161 | 172.16.50.54 | TCP | 66 | 57506 → 16310 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=571486218 |
| 4 | 2021-08-31 17:06:56.999364 | 0.002202000 | 172.16.50.54 | 172.16.50.161 | TCP | 148 | 16310 → 57506 [PSH, ACK] Seq=1 Ack=1 Win=29056 Len=82 TSval=286 |
| 5 | 2021-08-31 17:06:56.999729 | 0.000365000 | 172.16.50.161 | 172.16.50.54 | TCP | 66 | 57506 → 16310 [ACK] Seq=1 Ack=83 Win=29312 Len=0 TSval=57148622 |
| 6 | 2021-08-31 17:06:56.999741 | 0.000012000 | 172.16.50.161 | 172.16.50.54 | TCP | 262 | 57506 → 16310 [PSH, ACK] Seq=1 Ack=83 Win=29312 Len=196 TSval=5 |
| 7 | 2021-08-31 17:06:56.999749 | 0.000008000 | 172.16.50.54 | 172.16.50.161 | TCP | 66 | 16310 → 57506 [ACK] Seq=83 Ack=197 Win=30080 Len=0 TSval=286550 |
| 8 | 2021-08-31 17:06:57.000426 | 0.000677000 | 172.16.50.54 | 172.16.50.161 | TCP | 77 | 16310 → 57506 [PSH, ACK] Seq=83 Ack=197 Win=30080 Len=11 TSval= |
| 9 | 2021-08-31 17:06:57.000854 | 0.000428000 | 172.16.50.161 | 172.16.50.54 | TCP | 103 | 57506 → 16310 [PSH, ACK] Seq=197 Ack=94 Win=29312 Len=37 TSval= |
| | 2021-08-31 17:06:57.007023 | 0.006169000 | 172.16.50.54 | 172.16.50.161 | TCP | 165 | 16310 → 57506 [PSH, ACK] Seq=94 Ack=234 Win=30080 Len=99 TSval= |
| | 2021-08-31 17:06:57.065834 | 0.058811000 | 172.16.50.161 | 172.16.50.54 | TCP | 66 | 57506 → 16310 [ACK] Seq=234 Ack=193 Win=29312 Len=0 TSval=57148 |
| | 2021-08-31 17:06:59.036034 | 0.000000000 | 172.16.110.33 | 172.16.50.54 | TCP | 74 | 48378 → 16310 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 |
| | 2021-08-31 17:06:59.036105 | 0.000071000 | 172.16.50.54 | 172.16.110.33 | TCP | 74 | 16310 → 48378 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 S |
| | 2021-08-31 17:06:59.036751 | 0.000646000 | 172.16.110.33 | 172.16.50.54 | TCP | 66 | 48378 → 16310 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=173893372 |

```
Transmission Control Protocol, Src Port: 57506, Dst Port: 16310, Seq: 1, Ack: 1, Len: 0
    Source Port: 57506
    Destination Port: 16310
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 2045793833
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 1856564275
    1000 .... = Header Length: 32 bytes (8)
```

# TCP 四次挥手




数据传输完毕，双方都可释放连接。最开始的时候客户端和服务器都是处于ESTABLISHED状态，然后客户端主动关闭，服务器被动关闭

# TCP 四次挥手

为什么客户端最后还要等待2MSL?

答：MSL(Maximum Segment Lifetime)，TCP中报文最大生存时间。
第一：保证客户端发送的最后一个ACK报文能够到达服务器，因为这个ACK报文可能丢失，站在服务器的角度来看，我已经发送了FIN+ACK报文请求断开了，客户端还没给我回应，应该是我发送的请求断开报文他没有收到，于是服务器又会重新发送一次，而客户端就能在这个2MSL时间段内收到这个重传的报文，接着给出回应报文，并且会重启2MSL计时器。
第二：防止类似与“三次握手”中提到了的“已经失效的连接请求报文段”出现在本连接中。客户端发送完最后一个确认报文后，在这个2MSL时间中，就可以使本连接持续时间内所产生的所有报文段都从网络中消失。这样新的连接中不会出现旧连接的请求报文。

为什么建立连接是三次握手，关闭连接确是四次挥手呢?

答：建立连接的时候，服务器在LISTEN状态下，收到建立连接请求的SYN报文后，把ACK和SYN放在一个报文里发送给客户端。而关闭连接时，服务器收到对方的FIN报文时，仅仅表示对方不在发送数据了，但是还能接收数据，而自己也未必全部数据都发送给对方了，所以己方可以立即关闭，也可以发送一些数据给对方后，再发送FIN报文给对方来表示同意现在关闭连接，因此，己方ACK和FIN一般都会分开发送，从而导致多了一次

# TCP 四次挥手


tcp.stream eq 52

分组字节流 | 宽窄 | 区分大小写 | 字符串 | FIN | 查找

| No. | Time | tcp.time_delta | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|---|
| | 2021-09-13 15:42:06.191228 | 0.000000000 | 110.242.68.4 | 172.16.11.208 | TLSv1.2 | 85 | Encrypted Alert |
| | 2021-09-13 15:42:06.191228 | 0.000000000 | 110.242.68.4 | 172.16.11.208 | TCP | 60 | 443 → 52829 [FIN, ACK] Seq=982 Ack=13600 Win=2380 Len=0 |
| | 2021-09-13 15:42:06.192660 | 0.001432000 | 172.16.11.208 | 110.242.68.4 | TCP | 54 | 52829 → 443 [ACK] Seq=13600 Ack=983 Win=511 Len=0 |
| | 2021-09-13 15:42:06.192815 | 0.000155000 | 172.16.11.208 | 110.242.68.4 | TCP | 54 | 52829 → 443 [FIN, ACK] Seq=13600 Ack=983 Win=511 Len=0 |
| | 2021-09-13 15:42:06.205714 | 0.012899000 | 110.242.68.4 | 172.16.11.208 | TCP | 60 | 443 → 52829 [ACK] Seq=983 Ack=13601 Win=2380 Len=0 |

```
Acknowledgment number (raw): 1096764807
0101 .... = Header Length: 20 bytes (5)
Flags: 0x011 (FIN, ACK)
  000. .... .... = Reserved: Not set
  ...0 .... .... = Nonce: Not set
  .... 0... .... = Congestion Window Reduced (CWR): Not set
  .... .0.. .... = ECN-Echo: Not set
  .... ..0. .... = Urgent: Not set
  .... ...1 .... = Acknowledgment: Set
  .... .... 0... = Push: Not set
  .... .... .0.. = Reset: Not set
  .... .... ..0. = Syn: Not set
  .... .... ...1 = Fin: Set
```

# TCP 数据传输之传输过程

分析之前，再次强调一下seq和ack的含义：
sequence number：表示发送方packet的数据部分的第一位应该在整个data stream中所在的位置。（如果没有数据的传输，如ACK，虽然有一个seq，但这次传输在整个data stream中不占位置，下一个实际有数据的传输，依然会从上一次发送ACK的数据包的seq开始）
acknowledge number：表示接收方的下一次sequence number（SYN/FIN的传输虽然没有data，但是会让下一次传输的packet seq+1，但ack的传输不会让下一次的传输packet+1）

| No. | Source IP | SYN/ACK/Length | Destination IP | Comment |
|---|---|---|---|---|
| 1 | 172.16.50.161 | ==> SYN | 172.16.50.54 | Seq = 0,Ack = 0 |
| 2 | 172.16.50.161 | <== SYN,ACK | 172.16.50.54 | Seq = 0,Ack = 1 |
| 3 | 172.16.50.161 | ==> ACK | 172.16.50.54 | Seq = 1,Ack = 1 |
| ..... | | | | |
| 4 | 172.16.50.161 | ==>PSH,ACK,Len: 54 | 172.16.50.54 | Seq = 281,Ack = 497 |
| 5 | 172.16.50.161 | <== ACK,Len: 2896 | 172.16.50.54 | Seq = 497,Ack = 335 |
| 6 | 172.16.50.161 | <== ACK,Len: 2896 | 172.16.50.54 | Seq = 3393,Ack = 335 |
| 7 | 172.16.50.161 | <== ACK,Len: 2896 | 172.16.50.54 | Seq = 6289,Ack = 335 |
| 8 | 172.16.50.161 | ==> ACK,Len: 0 | 172.16.50.54 | Seq = 335,Ack = 6289 |
| 9 | 172.16.50.161 | ==> ACK,Len: 0 | 172.16.50.54 | Seq = 335,Ack = 3393 |
| ..... | | | | |

此处用到了TCP的滑动窗口传输

# TCP 数据传输之超时



1、如何处理丢包现象？
TCP针对乱序和拥塞进行了响应处理，TCP引入了快速重传机制。如果发送方认为发生了丢包现象，就会重发数据包。如何确定是否发生丢包，最简单的办法就是接收方每收到一个包，就向发送方返回一个ACK，表示自己已经收到了这段数据，反之，经过一段时间内没有收到ACK，很大可能是数据包丢失了，紧接着就会重发该数据包，直到收到ACK为止

2、超时时间如何确定？
1）将超时时间设置为固定值，例如200ms，超过固定超时时间就重传，但设置固定值很不靠谱，例如访问国外或者距离远的服务器，或者网络出现抖动，则会导致大量数据包被重发，引发雪崩效应
2）根据网络延迟，动态调整超时时间
RTT（Round Trip Time）：往返时延，也就是数据包从发出到收到对应的ACK的时间。RTT针对连接，每一个连接都有一个独立的RTT
RTO（Retransmission Time Out）:重传超时，也就是超时时间
计算公式：
SRTT <- (1 - α)·SRTT + α·RTT　　//求SRTT的加权平均
rttvar <- (1 - h)·rttvar + h·(|RTT - SRTT |)　//计算SRTT与真实值的差距，同样用加权平均
RTO = SRTT + 4·rttvar　//估算出新的RTO，rttvar的系数4 是调参调出来的
这个算法的整体思想就是结合平均值和平均偏差来进行估算。具体请参考[RFC6298]

# TCP 数据传输之重传机制

TCP在传输数据时遇到一旦超过RTO而没有收到ACK，就会重发该数据包，没有收到ACK的数据包会放到重传缓存区里，等收到ACK后，就从缓存区里删除。**所以，在这种机制下，每个数据包都有相应的计数器。如果发生重传事件，对于TCP来说，这是相当重要的时间（RTO往往大于两倍的RTT，超时往往意味这拥塞），TCP不仅会重传对应数据段，还会降低当前的数据发送速率，因为TCP会认为网络发生了拥塞**

重传机制

◆ 超时重传

发送数据时设定一个定时器，若在指定时间内没有收到应答报文，就会重发数据

◆ 快速重传

发送方一次发送多个数据包中若某个数据包丢失了且接收方一直回复这个丢失的数据包ACK报文，发送方收到三次以上就知道该报文还没有被接收方收到，可以重传这个数据包。但发送方不知后续数据包是否丢失 [RFC5681]
https://datatracker.ietf.org/doc/html/rfc5681#page-11

◆ SACK：

选择性重传，在tcp头部“选项”字段里加一个SACK，将收到的序号范围给发送方，这样发送方就知道哪些数据接收方收到了，哪些数据接收方没有收到，以便重传接收方没有收到的数据（设置net.ipv4.tcp_sack开启SACK）[RFC2018]
https://datatracker.ietf.org/doc/html/rfc2018

◆ D-SACK：

对SACK的扩展，若有发送方有重复发送数据包，通过SACK告诉发送方哪些数据被重复接收了。若SACK若告知这个包已被发送过，那么说明是接收方发送的ACK丢了 [RFC2883]

## 02 如何使用tcpdump抓取所需之包

# 如何抓取TCP报文-tcpdump

◆ tcpdump介绍

抓包工具tcpdump（dump traffic on network）时一个用于截取网络分组，并输出分组内容的工具。tcpdump 支持针对网络层、协议、主机、网络或端口的过滤，并提供and、or、not等逻辑语句来帮助你去掉无用的信息

命令格式：

tcpdump [ -DenNqvX ] [ -c count ] [ -F file ] [ -i interface ] [ -r file ] [ -s snaplen ] [ -w file ] [ expression ]

◆ 抓包选项：

-c：指定要抓取的包数量

-i interface：指定tcpdump需要监听的接口。默认抓取第一个网络接口

-P：指定要抓取的包使流入还是流出的包。可以给定的值为“in”、“out”和“inout”，默认为“inout”

-s len：设置tcpdump的数据包抓取长度为len，如果不设置默认将会是65535字节，如果抓取大的数据包大于设置的len长度，则会出现包截断，抓取len越大，包处理时间就会越长，并且会减少tcpdump可缓存的数据包的数量

-n：对地址以数字方式显式，否则显式为主机名，也就是说-n选项不做主机名解析

◆ 输出选项：

-X：输出包的头部数据，会以16进制和ASCII两种方式同时输出。

-XX：输出包的头部数据，会以16进制和ASCII两种方式同时输出，更详细。

-t：　在每行输出中不打印时间戳

-tt：　不对每行输出的时间进行格式处理(nt: 这种格式一眼可能看不出其含义, 如时间戳打印成1261798315)

-ttt：　tcpdump 输出时, 每两行打印之间会延迟一个段时间(以毫秒为单位)

-tttt：　在每行打印的时间戳之前添加日期的打印

-v：当分析和打印的时候，产生详细的输出。

-vv：产生比-v更详细的输出。

-vvv：产生比-vv更详细的输出。

# tcpdump命令实战


**1、抓取指定网卡或所有网卡的所有流量报文**
**//抓取特定网卡enp0s3的流量报文**
**tcpdump -i enp0s3 -n -ttt -vvv -X -w outfile2.pcap**
**//抓取所有网卡的流量报文**
**tcpdump -i any -n -ttt -vvv -X -w outfile2.pcap**

**2、抓取特定源或特定目的流量报文**
**//抓取特定源或目的流量报文**
**tcpdump -i enp0s3 [src | dst ] host 172.16.50.161 -n -ttt -vvv -X -w outfile2.pcap**
**//抓取指定主机的流量报文，不分源目**
**tcpdump -i enp0s3 host 172.16.50.161 -n -ttt -vvv -X -w outfile2.pcap**

**3、抓取特定端口的流量报文**
**//抓取指定端口**
**tcpdump -i enp0s3 port 16310 -n -ttt -vvv -X -w outfile2.pcap.pcap**
**//抓取源或目的指定端口流量报文**
**tcpdump -i enp0s3 [src | dst ] port 16310 -n -ttt -vvv -X -w outfile2.pcap**
**//抓取指定源目且指定端口的流量报文**
**tcpdump -i enp0s3 src host 172.16.50.161 and dst host 172.16.50.54 and port 16310 -n -ttt -vvv -X -w outfile2.pcap**

**4、万能抓包命令**
**tcpdump -i enp0s3 port 16310 -n -ttt -vvv -X -w outfile2.pcap**

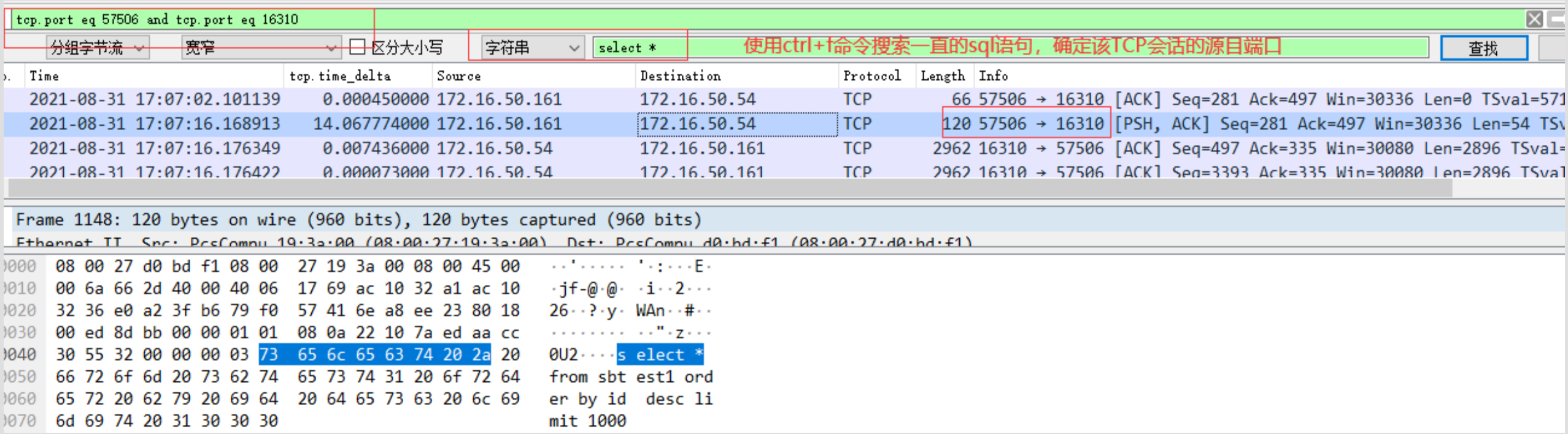
Great DB
万里数据库

## 03 如何分析数据包抓取想要的结果

# wireshark分析之跟踪tcp流并分组

Great DB 万里数据库

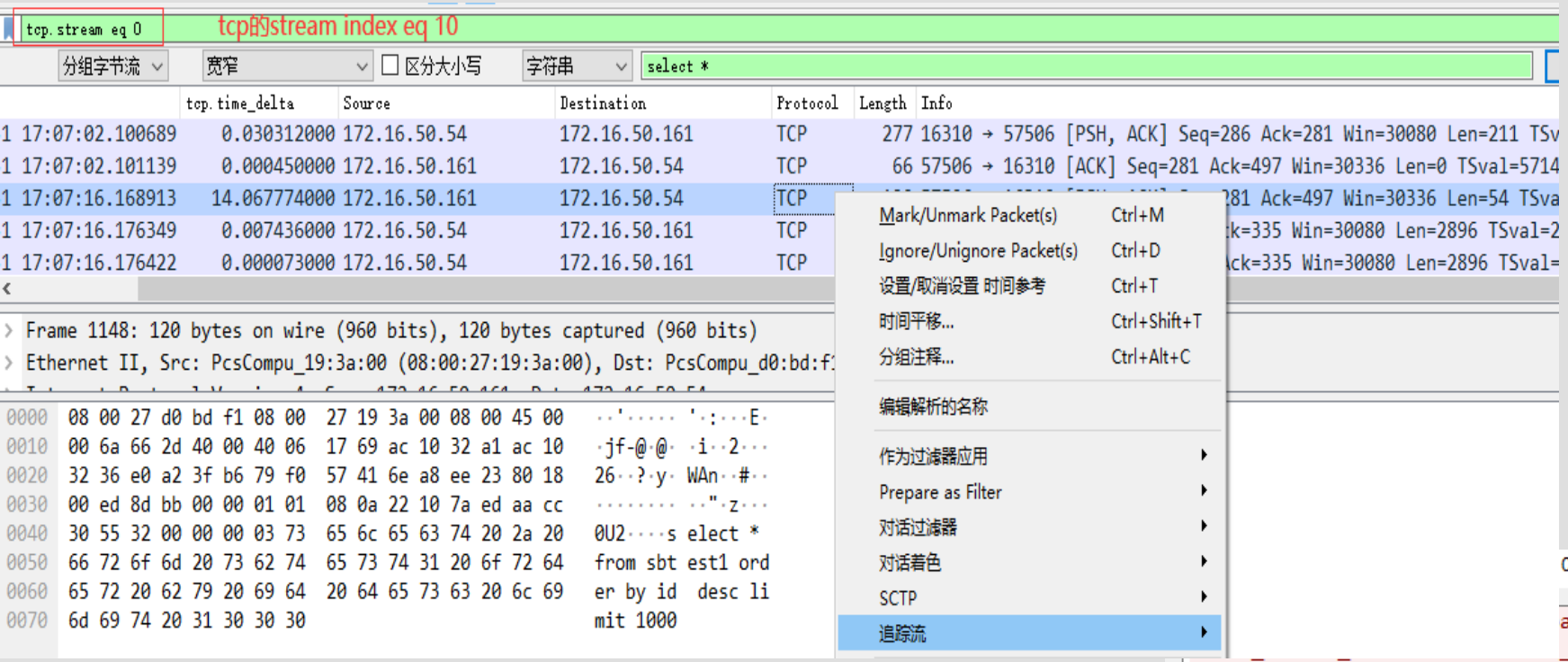
## 方式一：使用命令过滤的方式



# wireshark分析之跟踪tcp流并分组

Great DB 万里数据库

## 方式二：使用图形化操作

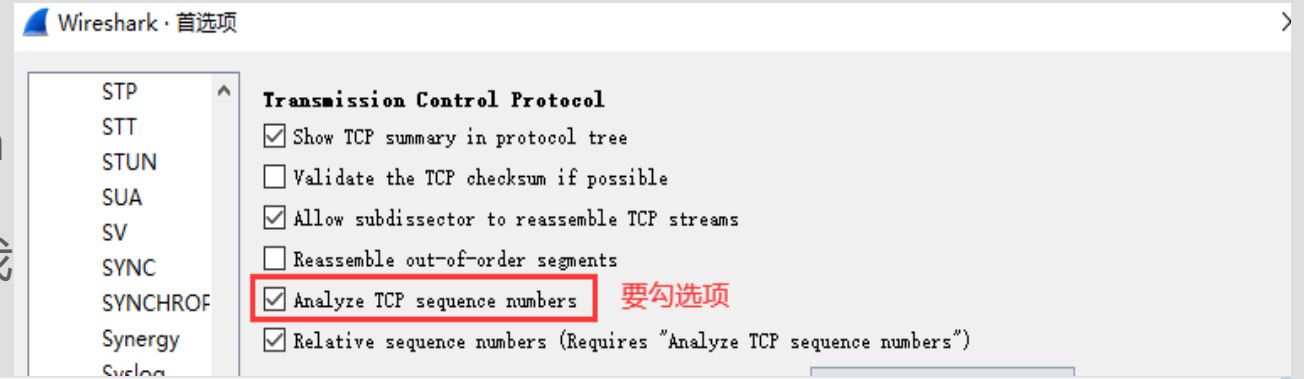


# wireshark分析之跟踪tcp流并分组

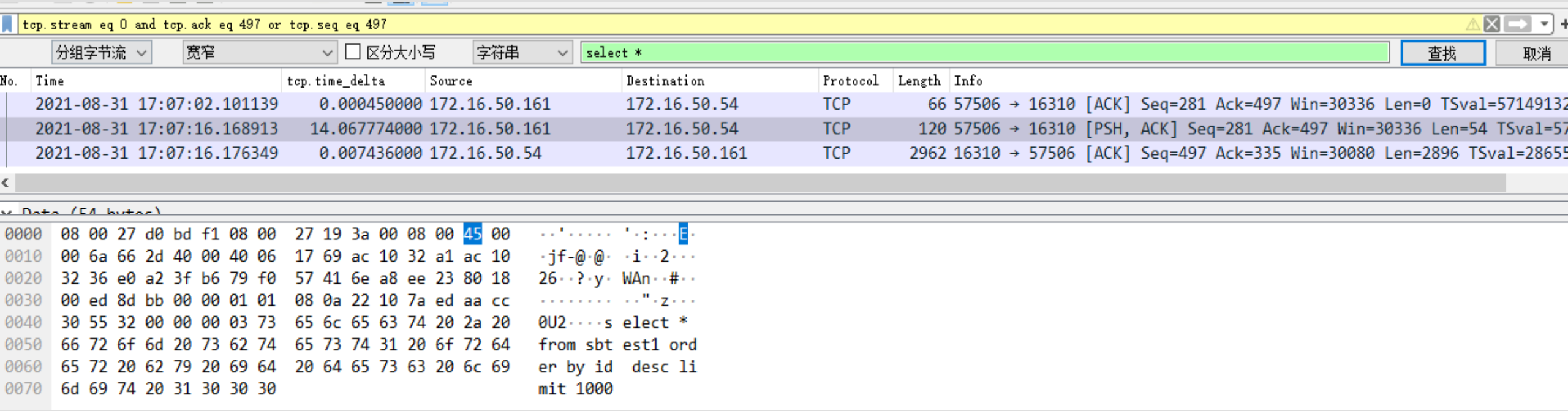
Great DB 万里数据库

## 方式三：通过tcp stream进行分组

tcp中从一个连接的握手到keep alive到fin，这个tcp stream index时不变的，比如说三次握手和四次挥手tcp stream都为10，则这个会话的tcp stream index 序号一直会不变。我们可以通过这个值进行tcp流的跟踪




# wireshark分析之确定server接收到client sql语句的时间

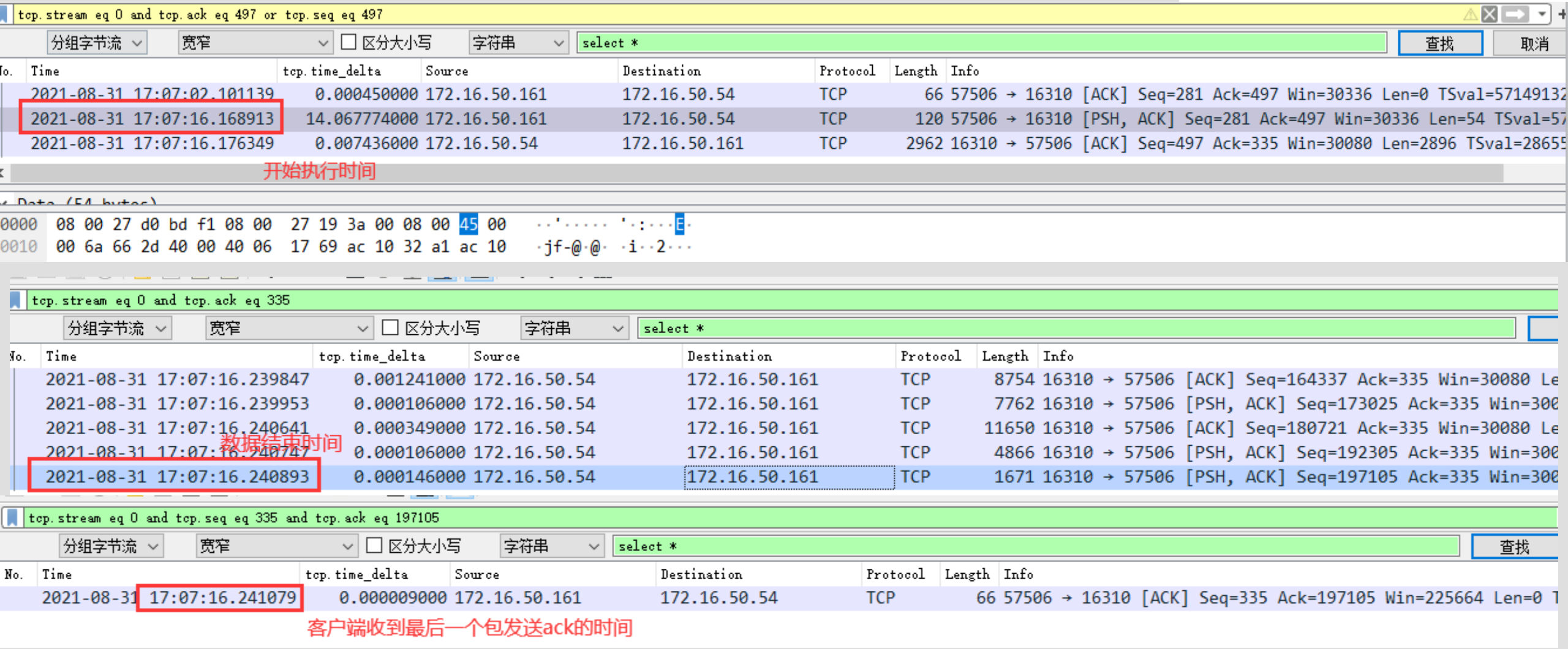



由上图可知：客户端执行一条sql语句：select * from sbtest1 order by id desc limit 1000，到server端接收到这条sql的时间为8ms（0.176349-0.168913）。

如何确认呢?

在TCP流的传输中我们需要通过seq和ack好来保证tcp流的无序包重组以及包的重传，通过ack的收包确认机制，来保证不存在丢包现象。所以由上图可知172.16.50.54在给172.16.50.161发包的时候seq的序号是收包的ack序号，ack是seq+len。则证明已经收到包了

# wireshark分析之确定client接收到server 数据的时间




由上图可知：客户端执行sql语句，到完全接收数据的耗时是73ms（0.241-0.168）。

Great DB
万里数据库

# 04 如何分析优化业务中接口慢问题

# 使用tcpdump+pt工具分析接口响应时间


**有时候我们会遇到一些接口响应时间慢，而我们又不知道具体的sql语句，需要对整个抓包文件进行分析，筛选出哪些sql比较慢，平均响应时间是多少，确定是中间件处理慢还是后端数据库响应慢等等。此时我们可以借助pt工具中的pt-query-digest对抓包进行分析。**

tcpdump抓包命令：
shell> tcpdump -s 65535 -x -nn -q -tttt -i any -c 100000 port 16310 > tcpdump_data.txt
命令解读：
-s 65535 或者-s 0表是抓取一个完整的数据包。包长度=首部固定大小+数据部分
-x 打印每个数据包包头跟数据包内容，如果要分析数据，此参数必须
-nn 不解析ip到主机名、端口号的服务名，而是以ip、port的形式显示
-tttt 在每行打印前打印日期，有必要，作为时间统计
-i any 抓取所有网口的包
-q 静默输出
-c 100000 抓取1w个数据包
port 16310  服务端口

使用pt-query-digest工具分析
由于pt-query-digest工具无法指定端口，所以在分析非3306端口时，需要使用sed进行替换
shell> sed -i 's/3306/16310/g' pt-query-digest
shell> pt-query-digest --limit 95%:100 --type tcpdump tcpdump_data.txt > slow_report.log
-type [binlog | genlog | slowlog | tcpdump]

# 使用tcpdump+pt工具分析接口响应时间

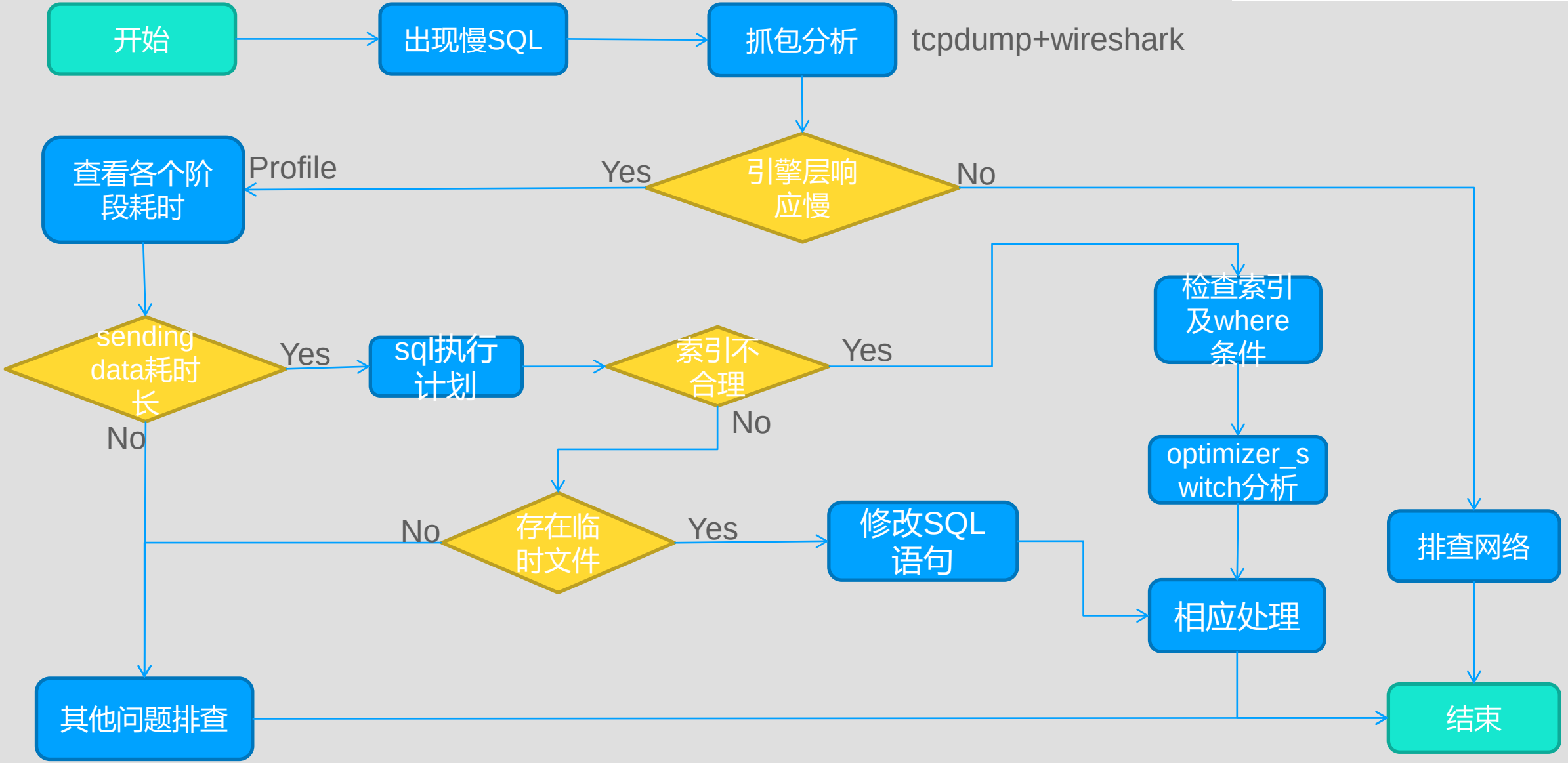
pt-query-digest分析结果输出

```
# 19.8s user time, 550ms system time, 33.67M rss, 237.39M vsz
# Current date: Fri Sep 10 18:41:36 2021
# Hostname: lh-3
# Files: tcpdump_data.txt
# Overall: 1.82k total, 22 unique, 38.64 QPS, 0.92x concurrency _________
# Time range: 2021-09-10 17:09:29.962160 to 17:10:17.011328
# Attribute          total     min     max     avg     95%  stddev  median
# ============     ======= ======= ======= ======= ======= ======= =======
# Exec time            43s       0       1s    24ms   122ms    64ms     3ms
# Rows affecte           0       0       0       0       0       0       0
# Query size        24.96k       5     980   14.06   26.08   80.93    4.96
# Warning coun           0       0       0       0       0       0       0

# Profile
# Rank Query ID                           Response time Calls R/Call V/M
# ==== ================================== ============= ===== ====== ====
#    1 0x8D589AFA4DFAEEED85FFF5AA78E5FF6A  33.4764 77.5%  1629 0.0206  0.11 BEGIN
#    2 0x653543C7C4D722B15D845DCA40B22094   1.4979  3.5%    13 0.1152  0.15 SELECT
#    3 0x995F41A1456C1CF6746D96521AE5B82C   1.2300  2.8%    12 0.1025  0.02 SET
#    4 0x8C29B77D3CE62B89EFBAC5BB3F2954BF   1.1600  2.7%     1 1.1600  0.00
#    5 0x20AB056BD78B0B8ECDCB69ACA4AC515F   1.1138  2.6%     3 0.3713  0.71
```

可以根据在不同服务间进行抓包，通过对于参数来判断那一块响应时间慢。讲课时，需要进行案例分享

# 如何使用抓包分析一条慢sql呢？



Great DB
万里数据库

## 05 使用第三方工具傻瓜式分析

# 第三方抓包分析集成工具


1、360开源的基于C写的mysql-sniffer
项目地址：https://github.com/Qihoo360/mysql-sniffer
结果输出：

```
[root@localhost ~]# mysql-sniffer -i enp3s0 -p 3306
2017-03-03 16:27:56    zhoujy    192.168.200.64    NULL    0ms    1    select @@version_comment limit 1
2017-03-03 16:27:56    zhoujy    192.168.200.64    NULL    0ms    1    select USER()
2017-03-03 16:28:00    zhoujy    192.168.200.64    NULL    0ms    1    SELECT DATABASE()
2017-03-03 16:28:00    zhoujy    192.168.200.64    xx      0ms    0    use xx
2017-03-03 16:28:03    zhoujy    192.168.200.64    xx      0ms    8    select * from xx
2017-03-03 16:28:13    zhoujy    192.168.200.64    xx      0ms    3    select user,host from mysql.user
2017-03-03 16:28:24    zhoujy    192.168.200.64    xx      0ms    2    show grants for root@localhost
2017-03-03 16:28:33    zhoujy    192.168.200.64    xx      18ms   1    select now()
2017-03-03 16:28:41    zhoujy    192.168.200.64    xx      32ms   1    select 1+1
```

2、准备开源使用go写的my-sniffer
项目地址：
github  https://github.com/ywlianghang/mysniffer
gitee    https://gitee.com/ywlianghang/mysniffer

3、mysql-sniffer的缺陷
1）使用前需要编译，不能跨平台
2）输出信息较少，关于每阶段的耗时输出不多，无法详细判断
3）输出的sql执行时间也不是很准确


# THANKS
# Q&A