背景描述

最近在在参与 Apache ShardingSphere 开源项目的一些工作，在 Github 的 issue 列表 认领了一个问题 #3005。

产品描述：首先简单介绍一下 ShardingSphere 中的一个产品 Sharding-Proxy，Sharding-Proxy 产品的功能是做一个透明的中间层代理，后面连接很多的 MySQL 数据库（可能是很多分库后 MySQL 数据库）。用户在使用中可以不用直接连到真实 MySQL 数据库上，而是连接到 Sharding-Proxy 上，通过这个工具再连接到真实的数据库上。

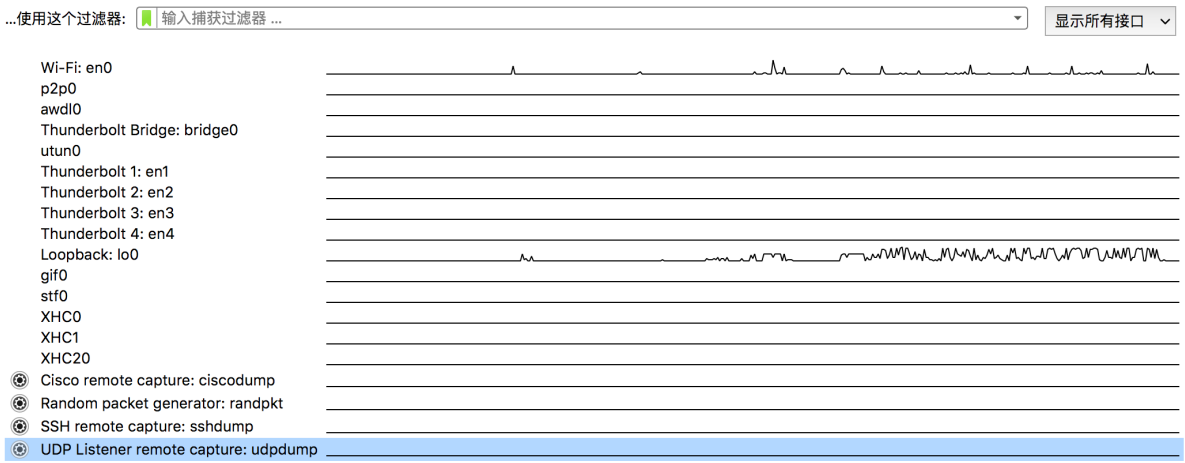
这个过程对于用户应该是无感知的（所以才叫透明的中间层代理），可以像使用 MySQL 一样使用 Sharding-Proxy，例如通过一些工具，像 MySQL JDBC 驱动或者是客户端工具（Navicat、MySQL Workbench）直接使用 Sharding-Proxy，所以 Sharding-Proxy 需要对 MySQL 协议进行解析和封装。对于客户端发送请求，Sharding-Proxy 解析后重新封装发送给 MySQL 服务器，对于 MySQL 服务器响应数据需要解析后重新封装发送给客户端，这个过程要做到精确，才能实现透明的中间层代理的效果。

认领的 issue 中的问题现象是，Navicat 直接连到 MySQL 服务器打开数据表没有问题，而通过 Sharding-Proxy 代理连接后再打开数据表，就会提示 There is no primary key here. Update will only use exact matching of the old values of the columns here. Thus, it may update more than one record. 这个错误，虽然对于使用是没有影响的，但是依然没有做到完全透明。

Wireshark 软件使用

对于 Navicat 这种客户端工具，其实是通过网络连接方式连接管理远端数据库，所以一定要走网络协议。所以对于这个问题，首先想到的办法就是通过网络抓包的方式对比一下两种连接方式返回的数据差别。网络抓包好像是有这么个工具 **Wireshark**，对于这个工具只闻名未曾谋面过，果断安装，百度了一下简单实用。

启动软件后，出现一堆网卡，可能是装虚拟机导致出现了好多虚拟网卡吧，不过大部分网卡没有数据流量。



首先是 **Wi-Fi: en0** 网卡，进去之后列表一堆数据，直接刷屏，根本看不出有效信息，毫无头绪。

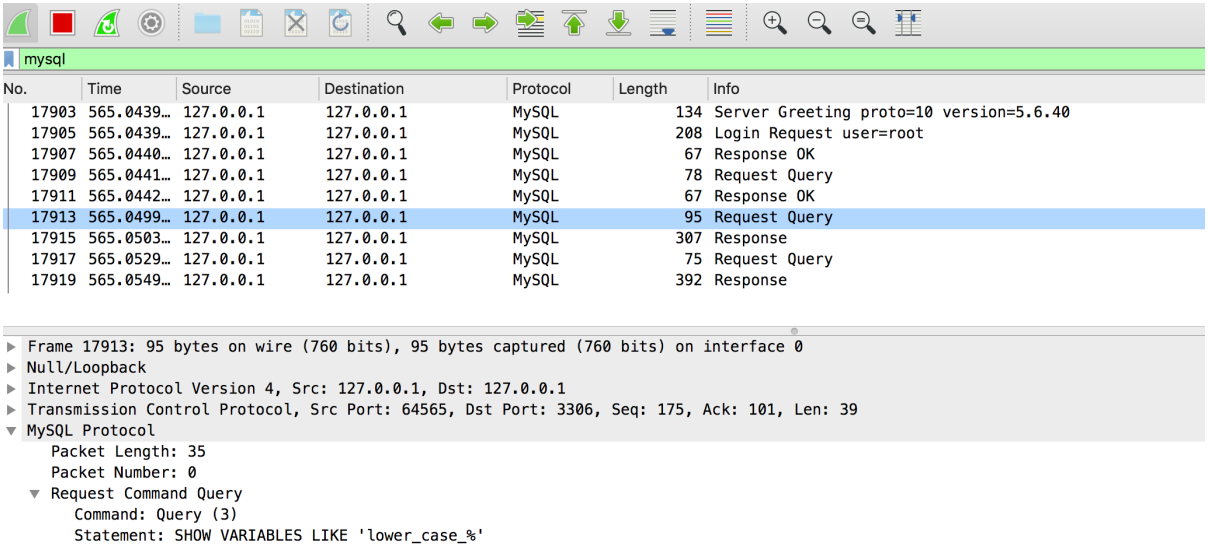
这个时候发现还有一个 **Loopback: lo0** 网卡，应该是 **环回地址（127.0.0.1）**，这个可能有点意思，因为 MySQL、Sharding-Proxy 和 Navicat 都装在本地开发环境，正常情况不会有真正的网络通信，都是本地搞定。

进入 **Loopback: lo0** 网卡，看到的就是 **Wireshark** 抓取的网络请求的数据了，其实第一次用，现学现卖，使用不是很流畅，不过还是百度大概了解到了一些使用技巧，首先 **Wireshark** 可以使用很多过滤器，过滤出我们比较感兴趣的网络请求数据，比如根据协议抓取数据，根据请求端口抓取数据进行区分。

在这个首先想到的是 MySQL 数据库使用的端口是 3306，Sharing-Proxy 使用的端口是 3307，还有就是 MySQL 有自己的应用层协议，协议名字就是：MySQL。

Wireshark 抓取 MySQL 协议数据

首先使用 Navicat 直接连到 MySQL 数据库上，在 Wireshark 软件中只过滤 MySQL 协议的网络数据请求：



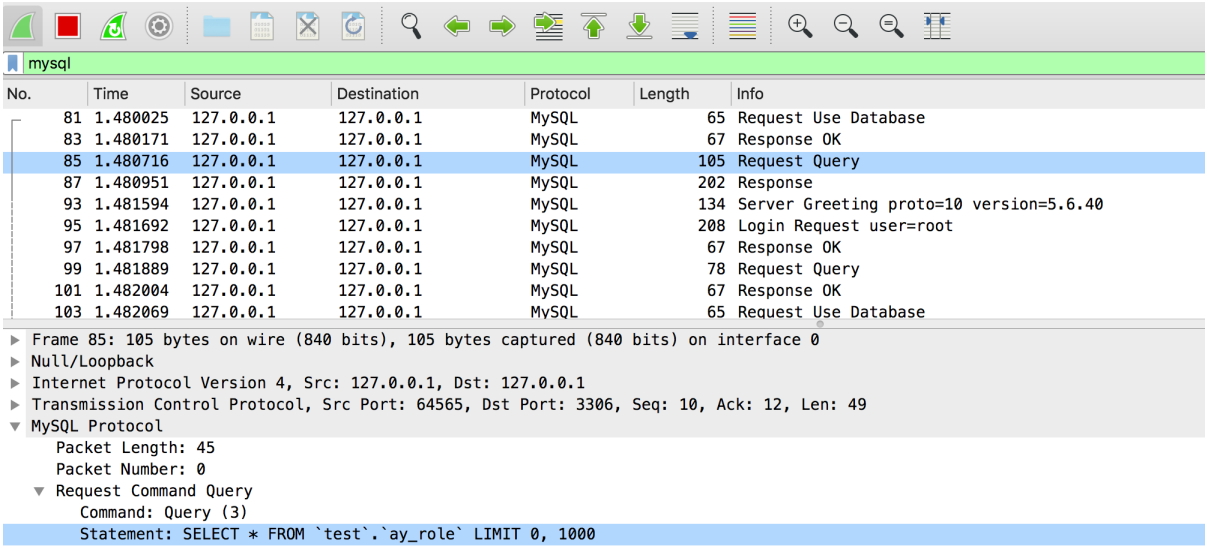
Wireshark 还是很强大，很直观的。每一次的 Request 请求，紧跟着是是配对的 Response 响应数据。其中在 Request 请求中 **MySQL Protocol** 中显示了 Navicat 客户端发送给 MySQL 数据库的命令：**SHOW VARIABLES LIKE 'lower_case_%'**；在 Response 相应的 **MySQL Protocol** 中显示了 MySQL 数据库发送给 Navicat 客户端的相应结果数据，通过上面的操作和观察，至少能大概明白简单的使用方式了。

Wireshark 抓包数据对比

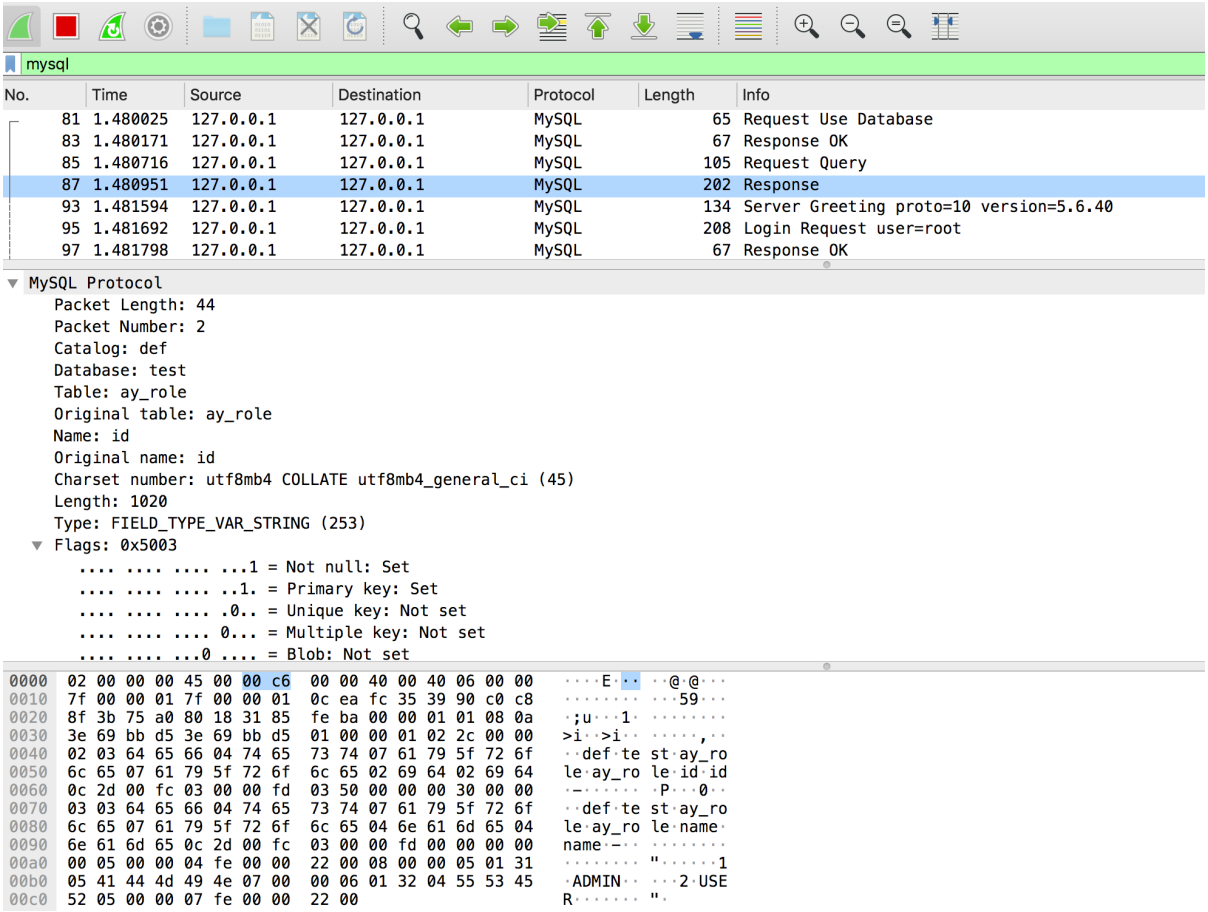
Wireshark 大概使用是明白了，现在需要看的就是 Navicat 直连 MySQL和 Navicat 通过 Sharing-Proxy 代理连接 MySQL 到底数据的请求响应过程到底有什么不同了。抓取同一个请求命令在两种不同情况下相应结果数据的差别，理论上来说这就应该是我们排查问题的方向，当然方向是否正确，还需要实操验证。

Navicat 直连 MySQL 抓包

根据 Navicat 的提示，只有在查询表数据表的时候会提示错误，所以问题大概应该出现在 select 查询语句的请求 & 相应数据上，我们抓取了一张表的 select 语句数据：



对于查询的这张数据表，其实 id 字段就是主键，主键是一定存在的，我们看一下返回结果中有关 id 字段的描述信息：



其中有 **Flags** 属性展开后就是一串标志位，通过实际内容大概就能明白就是标识该字段的一些特殊属性信息：

```
▼ MySQL Protocol
    Packet Length: 44
    Packet Number: 2
    Catalog: def
    Database: test
    Table: ay_role
    Original table: ay_role
    Name: id
    Original name: id
    Charset number: utf8mb4 COLLATE utf8mb4_general_ci (45)
    Length: 1020
    Type: FIELD_TYPE_VAR_STRING (253)
  ▼ Flags: 0x5003
      .... .... .... ...1 = Not null: Set
      .... .... .... ..1. = Primary key: Set
      .... .... .... .0.. = Unique key: Not set
      .... .... .... 0... = Multiple key: Not set
      .... .... ...0 .... = Blob: Not set
      .... .... ..0. .... = Unsigned: Not set
      .... .... .0.. .... = Zero fill: Not set
      .... .... 0... .... = Binary: Not set
      .... ...0 .... .... = Enum: Not set
      .... ..0. .... .... = Auto increment: Not set
      .... .0.. .... .... = Timestamp: Not set
      .... 0... .... .... = Set: Not set
    Decimals: 0
```

Navicat 连接 Sharding-Proxy 抓包

由于 Sharding-Proxy 底层是直接使用 Netty 实现网络连接，使用 TCP 协议，并且监听的端口号是 3307，所以我们在使用 Wireshark 抓包时可以采用端口号过滤的方式过滤网络请求数据：

> tcp.srcport==3307 || tcp.dstport==3307

我们还是抓取了相同一条 select 语句的请求相应数据，由于是直接抓取 TCP 协议包数据，无法像抓取 MySQL 协议数据那样直观了，只能通过二进制数据看出大概样子：

select 请求数据：

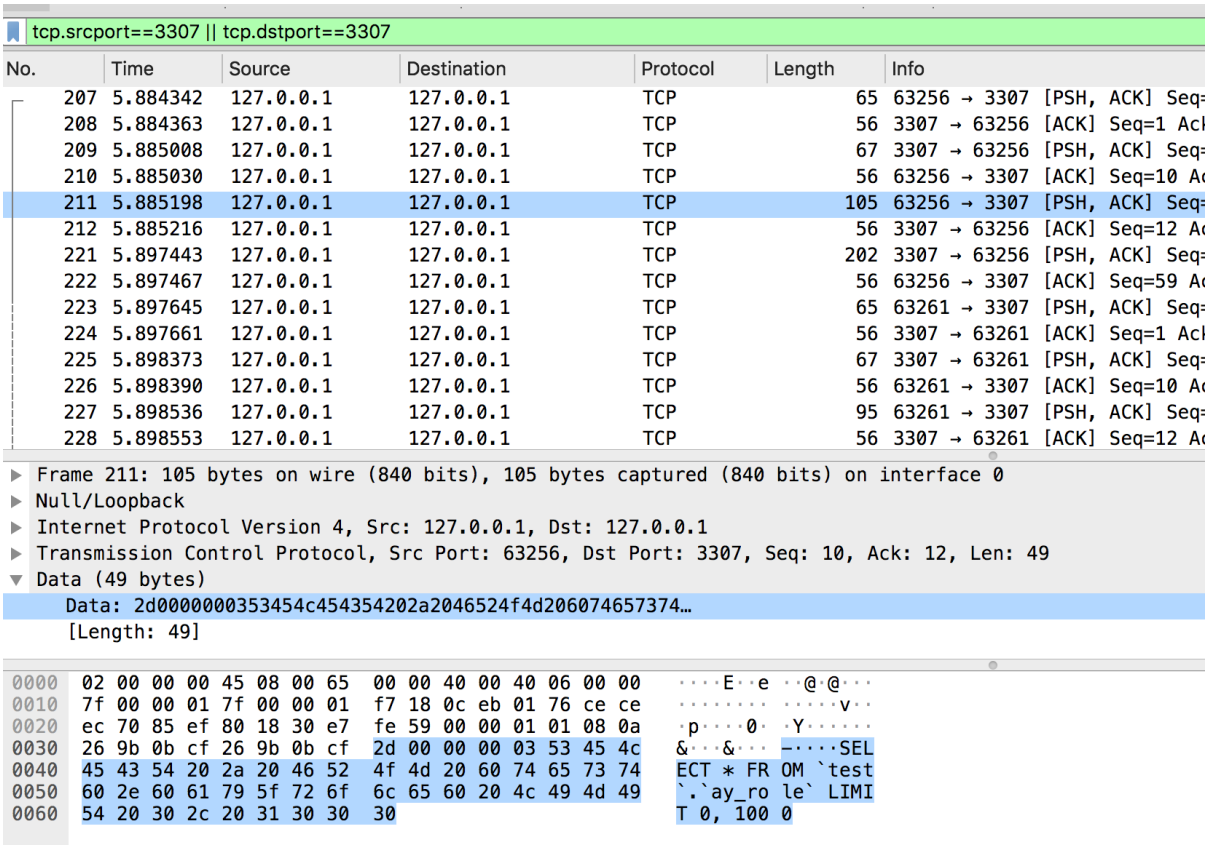
tcp.srcport==3307 || tcp.dstport==3307

No.	Time	Source	Destination	Protocol	Length	Info
207	5.884342	127.0.0.1	127.0.0.1	TCP	65	63256 → 3307 [PSH, ACK] Seq=1 Ack=1 Win=12520 L
208	5.884363	127.0.0.1	127.0.0.1	TCP	56	3307 → 63256 [ACK] Seq=1 Ack=10 Win=12710 Len=0
209	5.885008	127.0.0.1	127.0.0.1	TCP	67	3307 → 63256 [PSH, ACK] Seq=1 Ack=10 Win=12710 L
210	5.885030	127.0.0.1	127.0.0.1	TCP	56	63256 → 3307 [ACK] Seq=10 Ack=12 Win=12519 Len=(
211	5.885198	127.0.0.1	127.0.0.1	TCP	105	63256 → 3307 [PSH, ACK] Seq=10 Ack=12 Win=12519
212	5.885216	127.0.0.1	127.0.0.1	TCP	56	3307 → 63256 [ACK] Seq=12 Ack=59 Win=12708 Len=(
221	5.897443	127.0.0.1	127.0.0.1	TCP	202	3307 → 63256 [PSH, ACK] Seq=12 Ack=59 Win=12708
222	5.897467	127.0.0.1	127.0.0.1	TCP	56	63256 → 3307 [ACK] Seq=59 Ack=158 Win=12515 Len=
223	5.897645	127.0.0.1	127.0.0.1	TCP	65	63261 → 3307 [PSH, ACK] Seq=1 Ack=1 Win=12667 L
224	5.897661	127.0.0.1	127.0.0.1	TCP	56	3307 → 63261 [ACK] Seq=1 Ack=10 Win=12752 Len=0
225	5.898373	127.0.0.1	127.0.0.1	TCP	67	3307 → 63261 [PSH, ACK] Seq=1 Ack=10 Win=12752 L
226	5.898390	127.0.0.1	127.0.0.1	TCP	56	63261 → 3307 [ACK] Seq=10 Ack=12 Win=12667 Len=(
227	5.898536	127.0.0.1	127.0.0.1	TCP	95	63261 → 3307 [PSH, ACK] Seq=10 Ack=12 Win=12667
228	5.898553	127.0.0.1	127.0.0.1	TCP	56	3307 → 63261 [ACK] Seq=12 Ack=49 Win=12751 Len=(

```
▶ Frame 221: 202 bytes on wire (1616 bits), 202 bytes captured (1616 bits) on interface 0
▶ Null/Loopback
▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▶ Transmission Control Protocol, Src Port: 3307, Dst Port: 63256, Seq: 12, Ack: 59, Len: 146
▼ Data (146 bytes)
    Data: 01000001022c0000020364656604746573740761795f726f…
    [Length: 146]
```

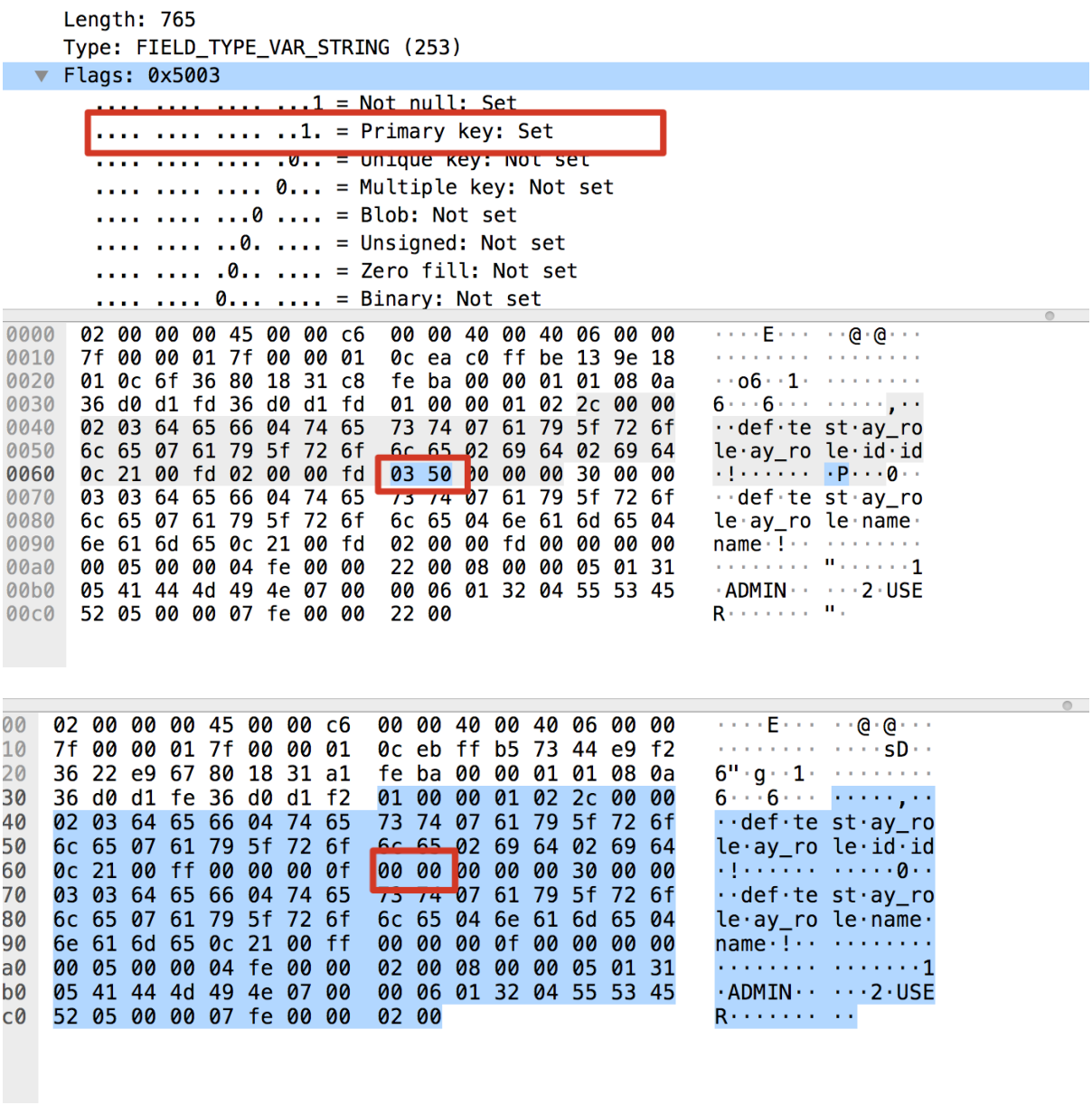
```
0000  02 00 00 00 45 00 00 c6  00 00 40 00 40 06 00 00   ····E··· ··@·@···
0010  7f 00 00 01 7f 00 00 01  0c eb f7 18 ec 70 85 ef   ········ ·····p··
0020  01 76 ce ff 80 18 31 a4  fe ba 00 00 01 01 08 0a   ·v····1· ········
0030  26 9b 0b db 26 9b 0b cf  01 00 00 01 02 2c 00 00   &···&··· ·····,··
0040  02 03 64 65 66 04 74 65  73 74 07 61 79 5f 72 6f   ··def·te st·ay_ro
0050  6c 65 07 61 79 5f 72 6f  6c 65 02 69 64 02 69 64   le·ay_ro le·id·id
0060  0c 21 00 ff 00 00 00 0f  20 00 00 00 00 30 00 00   ·!······ ····0··
0070  03 03 64 65 66 04 74 65  73 74 07 61 79 5f 72 6f   ··def·te st·ay_ro
0080  6c 65 07 61 79 5f 72 6f  6c 65 04 6e 61 6d 65 04   le·ay_ro le·name·
0090  6e 61 6d 65 0c 21 00 ff  00 00 00 0f 20 00 00 00   name·!·· ···· ···
00a0  00 05 00 00 04 fe 00 00  02 00 08 00 00 05 01 31   ········ ·······1
00b0  05 41 44 4d 49 4e 07 00  00 06 01 32 04 55 53 45   ·ADMIN·· ···2·USE
00c0  52 05 00 00 07 fe 00 00  02 00                     R·····   ··
```

Sharding-Proxy 相应结果：



结果分析

仔细分析抓取的数据结果，其实还是能看到一些差别的，不过 Sharding-Proxy 抓取到的结果不是很直观，还是给结果分析带来一定困难的，还是直接上结论吧：



分析的结果可以看出，Navicat 工具直连 MySQL 数据库时执行的 select 查询语句中返回的 `Flags` 信息标志位二进制数据为 `03 50`，标识该字段为 `Not null` 和 `Primary key`；但是 Navicat 连接 Sharding-Proxy 代理后返回的数据中该标志位二进制位 `00 00`，其实到这里，结果就很明显了，Sharding-Proxy 返回的数据中 `Flags` 信息缺失，而该数据正式判断该数据是否为主键、是否允许为null等等信息的一种标识。

写在最后

其实，现在记录整个内容过程感觉很简单，其实过程并不容易，有些分析其实就是 **经验 + 运气**，而且可能运气的成分还要多一些，不过整个过程下来还是给自己增长很多知识的，不过确实要说一句：**Wireshark 是个很不错的软件，以后要多学习、常使用，应用好了事半功倍。**

参考资料

- Wireshark使用教程