MQTT 消息阻塞与延迟问题深度复盘

---

1. 问题描述

现象：

- 高延迟：ThingsBoard (TB) 接收到的遥测数据时间戳 (`ts`) 比当前系统时间滞后约 **10**分钟。
- 资源闲置：服务器 CPU、内存、I/O 负载极低，几乎处于空闲状态。
- 监控矛盾：
    - EMQX 端监控显示队列无积压（Message Queue = 0），飞行窗口（In-flight）也未满。
    - Java 应用端 Heap Dump 显示 Paho 客户端内部队列 `messageQueue` 只有 **10** 条数据。
    - 关键异常日志：Paho 客户端频繁打印 `DEBUG` 日志：`wait for spaceAvailable`。

场景数据：

- 输入：8 个监控分站，每秒各上报 1 条消息（共 8 TPS）。
- 负载：每条消息包含该分站下所有传感器的最新值（约 11 个传感器），导致后续处理发生“读写放大”（1 条消息 -> ~11 次 DB 写操作 + 11 次 TB HTTP 请求）。
- 架构：Spring Boot + Spring Integration MQTT (`MqttPahoMessageDrivenChannelAdapter`) -> 单线程 Channel -> 业务处理。

---

2. 根源深度分析

2.1 核心矛盾：处理速度 < 生产速度

虽然表面 TPS 只有 8，但由于“读写放大”效应，单线程处理完一条分站消息（含多次 DB/HTTP IO）耗时约 **150ms ~ 1.3s**。

- 生产速度：1000ms / 8条 = **125ms/**条。
- 消费速度：**>150ms/**条。
- 后果：消费端无法及时消化输入流，必然产生积压。

2.2 Paho 的“隐形”流控机制 (罪魁祸首)

为何 Heap Dump 显示队列只有 10 条，且没有 OOM?

- **MaxInflight** 限制：Paho 客户端有一个核心参数 `maxInflight`（默认 10）。它同时限制了发出未确认的消息数和 接收缓冲区大小 **(**`INBOUND_QUEUE_SIZE`**)**。
- 阻塞设计：
  Paho 的接收线程 (`CommsReceiver`) 逻辑如下：

```
// 当内部队列满了 (达到 10 条)
while (messageQueue.size() >= INBOUND_QUEUE_SIZE) {
    // 阻塞 IO 线程，停止从 Socket 读取!
    spaceAvailable.wait(200);
}
```

- 连锁反应：
  a. 业务线程（Callback）处理慢 -> 无法及时腾出 `messageQueue` 空间。
  b. `messageQueue` 填满 (10条)。
  c. `CommsReceiver` 线程进入 `WAIT` 状态（打印 `wait for spaceAvailable`）。
  d. 操作系统 **TCP** 接收缓冲区 **(**`Recv-Q`**)** 填满。
  e. 通知 EMQX 端 TCP Zero Window。
  f. EMQX 停止发送数据。

> 结论：那消失的“10分钟”数据，实际上是堵在了操作系统的 **TCP** 内核缓冲区里，以及 EMQX 的推迟发送中。Java 内存里确实只有 10 条。

2.3 配置失效陷阱

虽然在 `application.properties` 配置了 `mqtt.maxInFlight=1000`，但实际未生效（Heap Dump 证明仍为 10）。

- 原因：Spring Bean 注入时，可能存在默认 `MqttPahoClientFactory` 覆盖了自定义配置，或者属性绑定存在命名/时机问题。
- 无效性：即便配置生效改为 **1000** 也无济于事。单线程瓶颈下，1000 的缓冲区也只会比 10 多撑几秒钟，随后依然会满并阻塞。

2.4 同步 ACK 机制

Paho 采用 同步回调 机制：

> *"An acknowledgment is not sent back to the server until this method [messageArrived] returns cleanly."*

业务逻辑不执行完，ACK 就不发。EMQX 也就不会发下一批数据。这相当于把高并发的 MQTT 协议强制降级为了 **Stop-and-Wait** 协议。

---

3. 优化方案

3.1 终极方案：引入线程池解耦 (已验证)

在 Spring Integration Flow 中，在 Paho 适配器与业务逻辑之间插入并发通道。

代码修改：

```
return IntegrationFlow.from(adapter)
    // ▼▼▼ 核心改动：引入多线程消费 ▼▼▼
    .channel(c -> c.executor(Executors.newFixedThreadPool(20)))
    // ▲▲▲ Paho 线程只需将任务放入线程池，立刻返回并发送 ACK ▲▲▲

    .<String, StationTelemetryMsg>transform(...)
    .handle(...) // 耗时的 DB/HTTP 操作在线程池中并行执行
    .get();
```

效果：

- **Paho** 线程：耗时从 150ms 降至 0.001ms (提交任务时间)。
- 流控解除：`messageQueue` 永远为空，`wait for spaceAvailable` 消失。
- 吞吐飙升：20 个线程并行处理，系统 TPS 从 7 提升至 100+。
- 延迟清零：积压数据瞬间被消化。

3.2 进阶方案：分区顺序消费 (Partitioned Ordering)

如果要求同一分站/设备的数据必须严格有序，使用 Hash 取模分发。

```
// 自定义分发器
int index = (gwCode.hashCode() & 0x7FFFFFFF) % PARTITION_SIZE;
executors[index].submit(() -> {
    // 业务逻辑
});
```

- 优点：兼顾并发性能与消息顺序，且无需管理动态线程池生命周期。

---

4. Eclipse Paho 客户端的弊端

Paho 是 Java 领域最老牌的 MQTT 客户端，但其设计确实存在时代局限性：

1. 阻塞式 **IO** 模型：接收线程直接参与流控阻塞，可能导致心跳包 (`PINGRESP`) 无法及时处理，引发莫名掉线。

2. 缺乏内置并发：即使是 `MqttAsyncClient`，其 接收回调 **(**`messageArrived`**)** 依然是单线程同步的。它默认把并发的重任完全甩给了用户。
3. **QoS 0** 处理粗暴：为了流控，连允许丢弃的 QoS 0 消息也会阻塞整个网络链路。
4. 配置陷阱：`MaxInflight` 这种关键参数默认值过小 (10)，且在高并发场景下极易成为隐形瓶颈。

6. 关键排查实战手册

6.1 如何验证 Paho 内部积压 (Heap Dump & MAT)

当怀疑内存积压但业务日志看不出异常时，通过 Java 堆转储是唯一实锤手段。

步骤 **1**：生成 **Heap Dump**
在生产服务器上执行（需 root 或对应用户权限）：

```
# <pid> 为 Java 进程 ID
jcmd <pid> GC.heap_dump /tmp/heap.dump
```

步骤 **2**：使用 **MAT** 打开分析
下载文件到本地，使用 **Eclipse Memory Analyzer (MAT)** 打开。

步骤 **3**：查找队列 **(**两种方法**)**

- 方法 **A (OQL** 查询 **-** 推荐**)**：
  点击工具栏 **OQL** 图标，输入并执行：

  ```
  SELECT * FROM
  org.eclipse.paho.client.mqttv3.internal.CommsCallback
  ```

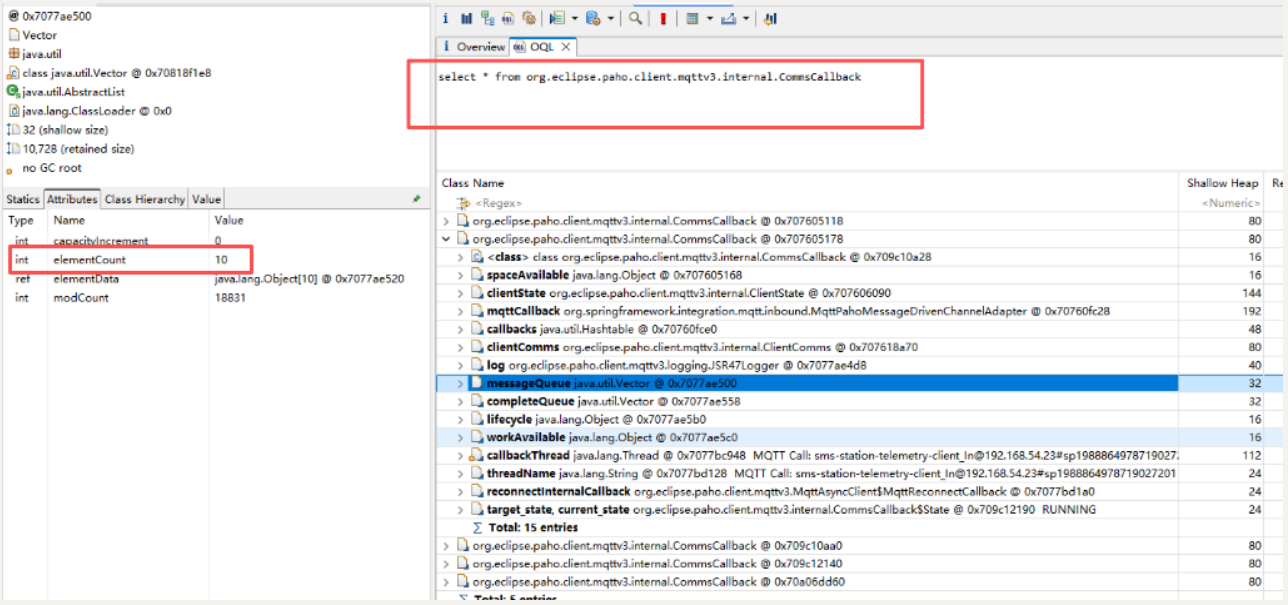
- 方法 **B (Dominator Tree)**：
  点击 Dominator Tree，搜索 `CommsCallback`。

步骤 **4**：查看关键属性
展开对象，找到 `messageQueue` (类型通常为 `Vector`) 字段。

- `elementCount`：当前积压的消息数量（如果为 10，说明通过流控已满；如果为 0，说明完全空闲）。

- `capacity`：验证你的配置是否生效（看它是 10 还是 1000）。

实战截图示例：



上图展示了通过 *OQL* 查询 `CommsCallback` 后，展开 `messageQueue` 看到的 `elementCount` 为 *10*，实锤了内部积压及配置未生效（默认值）。

6.2 如何实锤是“数据本身旧”还是“处理慢” (TCP抓包)

当发现数据延迟时，必须区分是 发过来就晚了 还是 收到了没来得及处理。

步骤 **1**：服务器端抓包

```
# 抓取端口 1883 (MQTT) 或 9999 (你的 HTTP 端口)
# -A: 打印包内容 (ASCII)
# grep: 过滤特定字段 (如 ts 或 POST)
sudo tcpdump -i any port 1883 -A -nvv | grep -E "ts|POST"
```

步骤 **2**：现场对比
观察控制台实时弹出的日志：

1. 看时间戳：抓取数据包中 JSON 体里的 `"ts": 1765881956824`。
2. 换算时间：将时间戳转换为人类可读时间 (例如 18:45)。
3. 对比当前：看一眼现在的系统时间 (例如 18:55)。
4. 结论：

- 如果数据包里的时间戳 已经是 **10** 分钟前 -> 上游问题（设备发了旧数据，或积压在 TCP/EMQX）。
- 如果数据包里的时间戳 就是当前时间 -> 应用问题（应用处理慢，日志里打印的时间晚是应用层积压）。

如果脱离 Spring Integration 框架，推荐使用基于 Netty 的新一代客户端：

1. **HiveMQ MQTT Client**：
    - 全异步非阻塞 IO (Netty)。
    - 支持 Reactive 背压，真正的异步 API。
    - 高并发性能强，不再阻塞 IO 线程。
2. **Vert.x MQTT Client**：
    - 极致性能，适合超高并发网关场景。

5.2 Spring 生态最佳实践

鉴于 **Spring Integration MQTT** 官方目前深度绑定 Paho，**"Paho + 线程池"** 依然是工程上的最优解。

关键原则：

1. 绝不在 `messageArrived` 回调里做耗时操作。
2. 必须 在 Adapter 之后紧跟一个 `ExecutorChannel` (线程池)。
3. **ThreadLocal** 传递：如果在多线程间切换，务必使用“快照+回放”模式手动传递上下文（如数据库 Schema）。
4. 监控：不要只看 broker 监控，要关注应用内部日志。`wait for spaceAvailable` 是 Paho 性能崩溃的标志这一信号。