WebFlux 中将 Parquet 流式转 CSV 并实时压缩为 ZIP 的实现思路与原理说明

版本/环境假设：JDK 17，Spring Boot 3.5.x（Spring Framework 6.1+），WebFlux + Reactor；Parquet Reader 使用 parquet-hadoop 1.16.0 + Hadoop 3.4.x。

本文面向：在 JVM 内存有限（例如 4GB）情况下，支持大文件 Parquet（数十 MB 至数 GB）导出为 CSV 或 ZIP(CSV) 的下载接口。

1. 需求与约束复述

1. 输入：已经落地到本地磁盘的 Parquet File 对象（例如来自 S3/GCS 的下载结果）。
2. 输出：HTTP 下载（parquet / csv / zip 三种格式之一）。
3. 约束：不能把整份 CSV 或整份 ZIP 攒在内存；允许下载慢，但必须能持续输出；客户端断开后要尽快停止上游读取，避免堆积导致 OOM。

2. 什么是背压（Backpressure）

背压是响应式流（Reactive Streams）里的“流量控制”机制：下游（Subscriber）通过 request(n) 明确告诉上游（Publisher）自己还能处理多少个元素，上游必须保证 onNext 的总次数不超过已请求的数量，从而避免无限缓冲导致 OOM。

在 WebFlux 下载场景里，下游通常是网络连接/HTTP 响应写出端：如果客户端带宽慢或暂时不读，服务端会减少 demand，上游就应该自动“慢下来”。

3. 为什么 Flux.create + FluxSink.next 很容易 OOM

Flux.create 适合把回调式 API 桥接成 Flux，但在下游跟不上时会采用缓存（buffer）方式处理背压：如果你在 while 循环里快速 sink.next(...) 产生 Map/行对象，就等于把压力转化为内存占用——下游慢，上游仍在生产，队列越堆越大，最终 OOM。

尤其是每行都 new 一个 Map：对象数量巨大 + GC 压力大 + 可能被 create 的 buffer 队列保留，风险会放大。

4. 总体架构：用 OutputStream 作为“背压闸门”

推荐把“读 Parquet + 生成 CSV + ZIP 压缩 + 写 HTTP 响应”做成一条同步写入管道，并把这个阻塞写入动作放到专用线程池（boundedElastic 或自定义 Executor）执行。

```
HTTP ■■■
    ↓■■■■■■■■ demand■
WebFlux ■■■■■■Publisher<DataBuffer>■
    ↓■DataBufferUtils.outputStreamPublisher ■■■
OutputStream■■■■■ demand ■■■■ write■
    ↓
ZipOutputStream■■■■■■■■■
    ↓
CSV Writer■BufferedWriter/OutputStreamWriter■
    ↓
ParquetReader<Group>■■■■■■■
```

关键点：你不再“生产 Flux”，而是直接把每行转换后写到 OutputStream；当网络端慢时，OutputStream.write 会阻塞，于是整个读取循环自然被限速。这样内存占用主要由：单行对象 + 少量缓冲区组成，而不是整条流的数据堆积。

5. 为什么可以不落地 CSV，也能边写边压缩成 ZIP

ZIP 写入是流式的：ZipOutputStream 在 putNextEntry 后，你可以不断 write(...) 写入该 entry 的内容；写完调用 closeEntry；最后 finish（或 close）写 central directory。整个过程不需要预先知道 CSV 的完整大小，也不要求 CSV 先落地。

因此：Parquet ->（逐行）-> CSV 字节 ->（直接写）-> ZipOutputStream ->（直接写）-> HTTP 响应。解压后得到的 CSV 字节与写入的字节一致（压缩是无损的）。

6. 取消、断流与资源回收：为什么不会一直读导致 OOM

当客户端断开或取消下载时，WebFlux 会触发下游 cancel。写响应的 OutputStream 往往会被关闭或后续写入抛 IOException。你的导出线程应当捕获该异常并尽快退出循环，同时关闭 ParquetReader / ZipOutputStream / 文件句柄。

在这种模型下，上游不会继续生成并缓存大量行对象：因为生产与消费通过 OutputStream 的阻塞语义耦合在一起，天然具备“按需”节奏。

7. 实现要点清单（工程化建议）

- 线程模型：所有 Parquet 读取、CSV 格式化、ZIP 压缩、磁盘 I/O 都是阻塞操作，必须放到 boundedElastic 或自建 Executor，不要跑在 Netty event-loop 线程。
- 缓冲：使用 BufferedWriter/BufferedOutputStream（例如 64KB-256KB）以减少系统调用；但不要过大到影响内存上限与延迟。
- CSV 正确性：统一 UTF-8；对包含逗号、双引号、换行的字段做 RFC 4180 风格转义（双引号包裹、内部双引号重复）。
- Parquet 类型映射：依据 PrimitiveTypeName 做 toString/数字格式化；INT96 通常是时间戳（与写入端编码有关），需要明确转换策略（例如按微秒/毫秒或按原始二进制输出）。
- 导出限制：按行数/字节数/耗时设置硬上限，避免被极端文件拖垮服务；必要时在业务层做异步任务 + 预签名下载。
- 监控：记录导出开始/结束、写出字节、耗时、是否取消、异常类型；并暴露指标（吞吐、取消率、压缩比）。

8. 为什么我们不再输出 Flux<byte[]>

核心原因是“ 对象化 + 上游快生产 ”会让系统更依赖内存：Map/包装类型/字符串拼接都增加分配与 GC；一旦链路上任何位置发生缓冲（例如 create 的 buffer、flatMap 的预取、下游网络回压），这些对象会被保留更久。

直接写 OutputStream 的方案把数据以字节形式尽快向下游推进，避免了大规模对象堆积；同时把背压闸门放在最靠近网络的位置，使得上游读取自然与网络消费速度对齐。

9. 参考资料（可继续深挖）

- Reactive Streams Specification (reactive-streams-jvm README.md) https://github.com/reactive-streams/reactive-streams-jvm
- Spring Framework DataBufferUtils Javadoc https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/io/buffer/DataBufferUtils.html
- Project Reactor Flux.create / FluxSink.OverflowStrategy Javadoc https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html
- Java 17 ZipOutputStream Javadoc https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/zip/ZipOutputStream.html