大规模数据同步后源端与目标端数据总条数对不上的系统性解决方案

原创 我科绝伦 小周的数据库进阶之路 2025年04月27日 08:03 重庆

热衷于分享各种干货知识，大家有想看或者想学的可以评论区留言，秉承着“开源知识来源于互联网，回归于互联网”的理念，分享一些日常工作中能用到或者比较重要的内容，希望大家能够喜欢，不足之处请大家多提宝贵地意见，我们一起提升，守住自己的饭碗。

小周的数据库进阶之路
致力于各类型数据库日常运维经验分享、新手入门、原理解读、自动化运维和架构优化。
266篇原创内容

公众号

正文开始

一、引言

在数据同步（如系统重构、分库分表、多源整合）场景中，“本地数据一致，生产环境条数对不上”是典型痛点。问题常源于**并发处理失控、数据库性能瓶颈、字段映射错误、缓存脏数据**等多维度缺陷。本文结合实战经验，从**应用层、数据库层（源库/中间库/目标库）、缓存层、字段变更处理**等维度，提供覆盖全链路的系统性参考方案。

二、应用层：并发控制与事务精细化管理

1. 线程池与异步任务失控

核心问题：

- 线程数超过数据库承载能力，导致连接池耗尽（ Too many connections 异常）。
- 主线程提前结束，子线程事务未提交，数据丢失。

解决方案：

- **CPU核数适配的线程池**：

```
// 8核服务器配置：核心线程=4，最大线程=8（避免超过数据库处理能力）
ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
executor.setCorePoolSize(4);
executor.setMaxPoolSize(8);
executor.setQueueCapacity(10000); // 任务队列缓冲，削峰填谷
executor.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPol
```

- **强制子线程独立事务**：

使用 @Transactional(propagation = Propagation.REQUIRES_NEW) 为每个批次创建独立事务，避免跨线程

事务污染：

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void processBatch(List<DataVO> batchData, CountDownLatch latch) {
    batchInsert(batchData); // 独立事务内的批量插入
    latch.countDown(); // 子线程完成后计数器减一
}
```

- **同步屏障机制**：
  通过 CountDownLatch 阻塞主线程，确保所有子线程执行完毕：

```
CountDownLatch latch = new CountDownLatch(totalBatch);
// 提交子线程任务时传入latch
latch.await(); // 主线程等待所有批次完成
```

2. 数据批次处理策略

优化点：

- **分批粒度控制**：单批次数据量控制在2000-20000条（视数据库性能调整），避免内存溢出：

```
List<DataVO> batch = new ArrayList<>(pageSize);
for (DataVO vo : cursor) {
    batch.add(vo);
    if (batch.size() == pageSize) {
        processBatch(batch, latch);
        batch.clear();
    }
}
```

- **批次间隔缓冲**：每批插入后休眠1秒，缓解数据库压力：

```
if (insertCount > 0) {
    log.info("批次插入成功，休眠1秒");
    Thread.sleep(1000); // 给数据库缓冲时间
}
```

三、数据库层：源库、中间库、目标库全链路优化

1. 源库：数据抽取与字段兼容性处理

核心挑战：

- 源端字段类型修改（如 VARCHAR 转 TEXT ）或新增字段，导致数据抽取失败。
- 大表查询阻塞源库性能。

解决方案：

- **字段映射动态校验**：
  同步前通过元数据接口（如JDBC DatabaseMetaData）获取源库与目标库字段信息，建立映射关系，处理类型不匹配：

```
// 示例：处理源库新增字段（目标库无该字段时忽略）
Map<String, String> sourceColumns = getSourceTableColumns("source_table")
Map<String, String> targetColumns = getTargetTableColumns("target_table")
List<String> validColumns = sourceColumns.keySet().stream()
    .filter(targetColumns::contains)
    .collect(Collectors.toList());
```

- **游标分批查询**：
  使用MyBatis游标（Cursor）流式读取数据，避免全量加载到内存：

```
try (SqlSession session = sqlSessionFactory.openSession();
     Cursor<DataVO> cursor = session.getMapper(SourceMapper.class).streaml
    cursor.forEach(vo -> handleData(vo));
}
```

2. 中间交换库：可靠传输与数据清洗

核心问题：

- 网络波动导致数据传输中断，中间库数据不完整。
- 数据清洗逻辑（如脱敏、格式转换）遗漏字段，导致目标库插入失败。

解决方案：

- **重试与幂等性设计**：
  为中间库表添加唯一约束（如 source_id + sync_time），结合Spring Retry实现幂等写入：

```
@Retryable(value = SQLException.class, maxAttempts = 3)
public void writeToMiddleDB(DataVO data) {
    middleMapper.insertOnDuplicateKeyUpdate(data); // 幂等插入（ON DUPLICAT
}
```

- **字段完整性校验**：
  在数据写入中间库前，校验必填字段（如目标库 NOT NULL 字段），缺失时填充默认值或记录错误：

```
if (StringUtils.isBlank(data.getTargetRequiredField())) {
    data.setTargetRequiredField("default_value"); // 填充默认值
```

```
        log.warn("字段缺失，已填充默认值：{}", data.getId());
    }
```

3. 目标库：批量插入与性能调优

关键参数与配置：

- **JDBC批量执行优化**：
  在连接URL中启用 rewriteBatchedStatements=true ，激活MySQL批量写入能力（需驱动5.1.13+）：

```
url: jdbc:mysql://target-host:3306/target-db?rewriteBatchedStatements=tru
```

- **数据库参数永久化配置**（修改 my.cnf ）：

```
[mysqld]
max_allowed_packet=512M          # 支持大批次数据传输
max_connections=60000           # 适应高并发写入
bulk_insert_buffer_size=512M    # 优化批量插入性能
innodb_lock_wait_timeout=300    # 减少长事务锁等待超时
```

- **批量插入语句优化**：
  使用MyBatis-Plus的 insertBatchSomeColumn 或原生 INSERT INTO ... VALUES 批量语法，避免逐条执行：

```
// 批量插入并过滤目标库不存在的字段
creditRecordMapper.insertBatchSomeColumn(
    dataList,
    columnList -> columnList.contains("id", "member_id", "create_time") /
);
```

四、字段变更场景：兼容性与异常处理

1. 源端字段类型修改

处理策略：

- **类型转换映射表**：预定义源库与目标库的类型转换规则（如源库 INT 转目标库 BIGINT ， DATETIME 转 TIMESTAMP ）：

```
Map<Class<?>, Class<?>> typeMapping = new HashMap<>();
typeMapping.put(Integer.class, Long.class);
typeMapping.put(java.sql.Timestamp.class, LocalDateTime.class);
```

- **异常捕获与日志**：在数据转换阶段捕获 TypeMismatchException ，记录失败数据并跳过，避免全量任务中断：

```
try {
    convertField(sourceField, targetType);
} catch (TypeMismatchException e) {
    log.error("字段类型转换失败：source={}, target={}, data={}",
        sourceField, targetType, data.getId(), e);
    errorData.add(data); // 收集错误数据后续处理
}
```

2. 源端新增字段

处理方案：

- **目标库字段扩展**：若目标库需兼容新增字段，提前执行 ALTER TABLE 语句，避免插入时字段不存在：

```
ALTER TABLE target_table ADD COLUMN new_field VARCHAR(50) DEFAULT NULL;
```

- **动态字段忽略**：若目标库暂不支持新增字段，同步时过滤该字段（通过字段白名单机制）：

```
List<String> targetColumnWhitelist = Arrays.asList("id", "name", "create_
dataVO.getColumns().keySet().removeIf(col -> !targetColumnWhitelist.conta
```

五、缓存层：数据一致性保障

1. 缓存脏数据问题

双删策略+延迟失效：

1. **同步前删除缓存**：避免同步过程中旧数据被读取；
2. **数据同步完成后**：通过MQ异步发送缓存失效事件；
3. **延迟二次删除**：针对高并发场景，延迟500ms再次删除缓存，避免并发写导致的脏数据：

```
```

```
// 示例：Redis双删实现
redisTemplate.delete("cache:user:" + userId);
syncDataToDatabase();
CompletableFuture.runAsync(() -> {
    try {
        Thread.sleep(500);
        redisTemplate.delete("cache:user:" + userId);
    } catch (InterruptedException e) { /* 处理中断 */ }
});
```

2. 缓存与数据库最终一致性

异步刷新机制：

- 通过监听数据库变更日志（如Canal监听Binlog），触发缓存异步更新：

```
// Canal监听新增数据，刷新缓存
if (event.getType() == INSERT) {
    CacheKey key = generateCacheKey(event.getData());
    cacheService.refresh(key, loadFromDatabase(key));
}
```

六、数据修复与验证体系

1. 数据对账工具

多维度校验：

- **总量校验**：对比源库、中间库、目标库的 COUNT(*) ，定位数据丢失环节；
- **主键差异**：通过 LEFT JOIN 或 EXCEPT 语句找出源库有而目标库无的记录：

```
-- MySQL查找差异数据
SELECT s.* FROM source_table s
LEFT JOIN target_table t ON s.id = t.id
WHERE t.id IS NULL;
```

- **字段哈希校验**：对关键字段生成MD5值，校验数据内容一致性（防止字段值错误）：

```
String sourceHash = MD5Utils.md5Hex(sourceData.toString());
String targetHash = MD5Utils.md5Hex(targetData.toString());
if (!sourceHash.equals(targetHash)) {
    log.error("数据内容不一致：id={}", data.getId());
}
```

2. 补偿与重试机制

分级处理策略：

- **自动重试**：对网络瞬时失败、数据库短暂阻塞，使用Spring Retry自动重试（最多3次，间隔递增）：

```
@Retryable(value = SQLException.class, backoff = @Backoff(delay = 1000, m
public void retryInsert(DataVO data) { /* 重试插入逻辑 */ }
```

- **人工修复**：对字段类型不匹配、业务逻辑错误等复杂问题，导出错误数据文件，人工核对后通过脚本补录：

```
-- 批量插入补偿数据
INSERT INTO target_table (id, name, create_time) VALUES
(1001, '补录数据', NOW()),
(1002, '补录数据', NOW());
```

3. 缓存强制清理

批量删除策略：

- 通过 SCAN 命令避免阻塞式删除，清理与同步数据相关的所有缓存：

```
# Redis批量删除用户相关缓存（避免KEYS命令阻塞）
redis-cli --scan --pattern "user:123:*" | xargs redis-cli del
```

七、生产环境最佳实践

1. **灰度发布与限流**：
   - 首次同步时，通过Sentinel限流（如并发线程数从2逐步增加至8），观察数据库连接数（ SHOW STATUS LIKE 'Threads_connected' ）和慢查询日志。
2. **全链路监控**：
   - 记录每批次的 start_time 、 end_time 、 data_count 、 error_count ，通过Prometheus+Grafana可视化同步进度。
3. **配置版本管理**：
   - 数据库参数、线程池配置、字段映射规则通过配置中心（如Nacos）管理，支持动态调整，避免硬编码。

八、全量与增量数据未同步的专项解决方案

在数据同步体系中，**全量同步**（首次初始化或重置数据）与**增量同步**（实时/定时更新变化数据）是两类核心场景。若发现数据未同步（如全量漏批、增量丢失），需针对两类场景的特性设计专项修复策略。

1、全量数据未同步：从断点续传到补偿校验

1.1 问题定位：全量同步中断的典型场景

- **中途失败**：同步过程中因数据库连接超时、OOM异常导致任务中断，部分数据未写入目标库。
- **漏批现象**：多线程并发处理时，某批次数据未提交或事务回滚，导致目标库缺失完整批次。
- **结构变更**：同步期间源库字段类型修改/新增字段，导致后续数据解析失败，流程终止。

1.2 解决方案：断点续传+分段重试

（1）断点记录与续传机制

- **创建同步断点表**：记录全量同步的进度（如已处理的最大主键ID、最后批次时间），支持从断点恢复：

```
CREATE TABLE sync_breakpoint (
    table_name VARCHAR(100) PRIMARY KEY,
    last_processed_id BIGINT,  -- 最后处理的记录ID
    last_batch_time TIMESTAMP, -- 最后批次处理时间
    status VARCHAR(20)         -- 状态：RUNNING/PAUSED/FAILED
);
```

- **代码实现**：

```
// 读取断点，确定本次同步起始ID
Long startId = breakpointMapper.getLastProcessedId(tableName);
startId = (startId == null) ? 0 : startId + 1; // 从下一条开始
// 分页查询：WHERE id >= startId LIMIT pageSize
List<DataVO> dataList = sourceMapper.selectByRange(startId, pageSize);
```

（2）失败批次重传策略

- **标记失败批次**：每次批次处理前生成唯一批次号（如 UUID+时间戳 ），失败时记录到日志表，支持精准重传：

```
// 批次处理
String batchNo = generateBatchNo();
try {
    processBatch(dataList, batchNo);
    breakpointMapper.updateLastProcessedId(tableName, maxId); // 成功后更新
} catch (Exception e) {
    syncLogMapper.insertFailedBatch(batchNo, e.getMessage()); // 记录失败批
    throw e; // 触发重试
}
```

（3）全量数据二次校验与补偿

- **总量对比**：同步完成后，对比源库与目标库 COUNT(*) ，若不一致则触发全量扫描：

```
-- 源库与目标库总量差异
SELECT source_count - target_count AS diff FROM (
    SELECT COUNT(*) AS source_count FROM source_table
) s, (
    SELECT COUNT(*) AS target_count FROM target_table
) t;
```

- **差异数据补录**：通过主键范围查询（如 ID IN (漏失ID列表) ）补录数据，避免全量重跑：

```
// 获取漏失ID列表（通过LEFT JOIN）
List<Long> missingIds = sourceMapper.findMissingIds(targetTable);
if (!missingIds.isEmpty()) {
    List<DataVO> missingData = sourceMapper.selectByIds(missingIds);
    targetMapper.batchInsert(missingData); // 批量补录
}
```

2、增量数据未同步：从标记修复到Binlog补抓

2.1 问题定位：增量同步失效的核心原因

- **增量标记未更新**：源库数据变更后，未正确记录变更位点（如时间戳、版本号），导致后续同步遗漏。
- **消息队列丢失**：通过MQ传输增量数据时，消息未被消费或消费失败，且未配置重试机制。
- **Binlog解析中断**：使用Canal监听数据库变更日志时，因网络波动导致位点（Position）丢失，无法继续解析。

2.2 解决方案：标记修复+多源捕获

(1) 基于时间戳/版本号的增量修复

- **修复增量标记**：
  - 若依赖 update_time 时间戳，查询源库中 update_time > 最后同步时间 的数据，重新同步：

```
SELECT * FROM source_table
WHERE update_time > '2025-04-26 10:00:00'  -- 最后成功同步时间
ORDER BY update_time ASC;
```

  - 若依赖 version 版本号（乐观锁字段），查找 version > 最后同步版本 的记录：

```
long lastVersion = incrementalConfig.getLastVersion();
List<DataVO> incrementalData = sourceMapper.selectByVersionGreaterThan(l
```

- **幂等性处理**：目标库使用 ON DUPLICATE KEY UPDATE 避免重复插入（需定义唯一约束，如 source_id ）：

```
INSERT INTO target_table (source_id, data)
VALUES (1001, 'data')
ON DUPLICATE KEY UPDATE data = VALUES(data); -- 冲突时更新
```

（2）Binlog断点续传与补抓

- **恢复Canal位点**：
  1. 从Canal管理后台查询最后成功解析的位点（ binlog_file 和 binlog_pos ）；
  2. 手动设置Canal客户端从指定位点开始解析：

```
canalConnector.connectAndSync();
canalConnector.position(new Position(binlogFile, binlogPos)); // 重置解析
```

- **历史Binlog补抓**：
  若增量数据丢失范围较大（如超过24小时），通过MySQL SHOW BINARY LOGS 获取历史日志文件，使用 mysqlbinlog 工具解析指定时间范围的变更：

```
# 解析2025-04-26 00:00:00到10:00:00的Binlog
mysqlbinlog --start-datetime="2025-04-26 00:00:00" --stop-datetime="2025-
```

（3）MQ消息重试与死信队列处理

- **自动重试**：为MQ消费者配置重试策略（如RocketMQ的 maxReconsumeTimes=3 ），失败消息进入死信队列：

```
// RocketMQ消费者配置
consumer.setMaxReconsumeTimes(3);
consumer.registerMessageListener((MessageListenerConcurrently) (msgs, con
    try {
        processIncrementalData(msgs);
        return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
    } catch (Exception e) {
        context.setDelayLevel(3); // 延迟5秒重试
        return ConsumeConcurrentlyStatus.RECONSUME_LATER;
    }
});
```

- **死信队列人工处理**：定期扫描死信队列，解析失败原因（如字段缺失、格式错误），修复后重新投递到正常队列：

```
// 死信队列消息重投
Message deadLetterMsg = deadLetterQueue.fetchMessage();
if (repairIncrementalData(deadLetterMsg)) { // 修复数据
    normalQueue.sendMessage(deadLetterMsg); // 重新投递
}
```

3、混合场景：全量+增量联动修复

当全量同步失败且已产生增量变更时，需采用“全量打底+增量追补”策略：

1. **重新执行全量同步**：从最新断点开始，覆盖同步历史数据（建议在低峰期执行）；
2. **捕获全量期间的增量**：在全量同步过程中，单独监听源库变更，将增量数据暂存到临时表；
3. **增量数据合并**：全量同步完成后，将临时表中的增量数据按顺序写入目标库，确保最终一致性：

```
-- 临时表存储全量期间的增量数据
CREATE TABLE temp_incremental (
    id BIGINT PRIMARY KEY,
    operation VARCHAR(10), -- INSERT/UPDATE/DELETE
    data JSON
);
-- 合并到目标库
INSERT INTO target_table (id, data)
SELECT id, data FROM temp_incremental
ON DUPLICATE KEY UPDATE data = VALUES(data);
```

4、监控与预防：提前发现未同步数据

1. **增量标记巡检**：
   - 定时任务检查源库与目标库的增量标记（如 last_sync_time ），若超过5分钟未更新则报警：

```
SELECT table_name FROM incremental_config
WHERE last_sync_time < NOW() - INTERVAL 5 MINUTE;
```

2. **变更数据积压监控**：
   - 对Binlog解析延迟、MQ队列堆积量设置阈值（如积压超过1000条报警），通过Prometheus+Alertmanager实时通知：

```
canal_lag_seconds > 300  // Binlog解析延迟超过5分钟
rocketmq_queue_consumer_offset - rocketmq_queue_max_offset > 1000  // MC
```

3. **数据一致性巡检**：
   - 每日凌晨执行全库对账，对比源库与目标库的主键总数、关键业务字段总和（如订单总金额），差异超过0.1%时触发自动修复：

```
long sourceSum = sourceMapper.sumAmount();
long targetSum = targetMapper.sumAmount();
if (Math.abs(sourceSum - targetSum) > sourceSum * 0.1%) {
    triggerAutoRepair(); // 自动触发差异数据修复
}
```

总结：全量与增量同步的修复准则

- **全量同步**：优先采用**断点续传**避免重复劳动，通过**总量校验+主键补录**确保完整性；

- **增量同步**：依赖**增量标记可靠性**（时间戳/版本号/Binlog位点），结合**幂等性设计**防止重复数据；
- **混合场景**：执行**全量打底+增量追补**，确保历史数据与实时变更的最终一致；
- **预防优先**：通过**监控增量标记、积压量、数据一致性巡检**，将问题扼杀在萌芽阶段。

数据同步的核心是“以终为始”——无论全量还是增量，最终目标是让目标库数据与源库“实时、准确、完整”。通过断点续传、幂等插入、Binlog补抓等专项技术，配合自动化监控与修复机制，可将数据未同步的风险降至最低，保障业务系统的稳定运行。

小周的数据库进阶之路

致力于各类型数据库日常运维经验分享、新手入门、原理解读、自动化运维和架构优化。

266篇原创内容

公众号

END

文中的概念来源于互联网，如有侵权，请联系我删除。

欢迎关注公众号：**小周的数据库进阶之路**，一起交流数据库、中间件和云计算等技术。如果觉得读完本文有收获，可以转发给其他朋友，大家一起学习进步！感兴趣的朋友可以加我微信，拉您进群与业界的大佬们一起交流学习。

我科绝伦

喜欢作者

日常运维小技巧 55　数据同步 1　运维 10

日常运维小技巧 · 目录

上一篇
严防脚本泄露！教你几个让 Shell 脚本仅自己可见的加密技巧

下一篇
分享一个可以一键搞定99%磁盘告警的Shell脚本

个人观点，仅供参考