F-S-06-01：数据库连接池耗尽

典型现象

- 应用日志中出现 Could not get JDBC Connection 、 DataSource: wait_timeout 或 `Connection pool exhausted`。
- 调用数据库的接口响应变慢或直接失败。
- 增加连接池最大连接数后问题暂时缓解，但后续再次出现。

可能原因

1. **连接未关闭**（占比约 50%）：代码中使用了数据库连接但未在 finally 块中关闭，或连接池配置的 removeAbandoned 未开启。
2. **并发请求激增**（占比约 20%）：业务高峰期瞬间请求数超过连接池的最大连接数，等待超时。
3. **慢 SQL 阻塞**（占比约 15%）：大量慢 SQL 长期占用连接，导致其他请求无法获取连接。
4. **事务过长**（占比约 10%）：事务中包含了非数据库操作（如调用外部接口、文件处理），连接长时间被持有。
5. **连接泄漏检测失灵**（占比约 5%）：连接池的 validationQuery 或 testOnBorrow 配置不当，无法检测失效连接。

排查思路

1. **查看连接池监控**：如果使用 HikariCP，开启 registerMbeans ，通过 JMX 查看 activeConnections 、 idleConnections 、 pendingThreads 。
2. **分析数据库连接数**：登录数据库执行 SHOW PROCESSLIST ，查看当前连接数及每个连接的 Time 列，找出长时间不释放的连接。
3. **检查代码**：搜索是否有 getConnection() 后未调用 close() 的地方。使用 try-with-resources 或确保finally 中关闭。
4. **调整连接池参数**：
   - 增大 maximumPoolSize （但不宜超过数据库 max_connections ）。
   - 设置 connectionTimeout 和 idleTimeout 合理值。
   - 开启 leakDetectionThreshold （如 60 秒）检测连接泄漏。
5. **优化慢 SQL**：配合 S-06-02 排查慢查询，减少连接占用时间。

保养提示

- 定期审查数据库连接使用情况，通过监控告警当活跃连接数超过最大连接数的 80% 时预警。
- 强制使用连接池的"连接泄漏检测"功能（如 HikariCP 的 leakDetectionThreshold ）。
- 对高并发接口采用分批处理或异步方式，避免长时间持有连接。

F-S-06-02：慢 SQL 导致系统卡顿

典型现象

- 系统响应变慢，接口耗时增加。
- 数据库 CPU 使用率飙升。
- 慢查询日志中出现大量超过阈值的 SQL 语句。

可能原因

1. **缺少索引**（占比约 60%）：查询条件字段未建索引，导致全表扫描。
2. **统计信息过旧**（占比约 20%）：执行计划未更新，选择了低效的索引或 Join 顺序。
3. **SQL 写法低效**（占比约 15%）：使用了 SELECT * 、 LIKE '%xxx' 、 OR 等无法利用索引的写法，或未分页返回大数据量。
4. **锁等待**（占比约 5%）：SQL 本身不慢，但因行锁或表锁而等待时间长。

排查思路

1. **开启慢查询日志**：设置 long_query_time = 1 （秒），记录所有超过 1 秒的 SQL。
2. **分析执行计划**：使用 EXPLAIN 分析慢 SQL，关注 type （ALL 全表扫描）、 possible_keys 、 key 、 rows 、 Extra （Using filesort、 Using temporary）。
3. **添加或优化索引**：根据 WHERE 、 JOIN 、 ORDER BY 字段创建复合索引，注意索引顺序（等值在前，范围在后）。
4. **更新统计信息**：执行 ANALYZE TABLE 或 OPTIMIZE TABLE ，让优化器重新选择执行计划。
5. **改写 SQL**：
   - 避免 SELECT * ，只取必要字段。
   - 将 LIKE '%keyword' 改为全文索引或反向匹配。
   - 拆分复杂查询为多个简单查询，在应用层关联。
6. **检查锁等待**：使用 SHOW ENGINE INNODB STATUS 或 performance_schema 查看是否存在锁竞争。

保养提示

- 每周定期分析慢查询日志，每周至少优化一条最慢的 SQL。
- 使用数据库监控工具（如 PT-Query-Digest）自动汇总慢 SQL 模式。
- 对重要查询进行压测，提前发现性能瓶颈。

F-S-06-03：数据库锁争用

典型现象

- 应用出现死锁错误（ Deadlock found when trying to get lock ）。
- 接口响应延迟，大量事务处于锁等待状态。
- 数据库 CPU 正常但连接堆积。

可能原因

1. **行锁未及时提交**（占比约 50%）：事务代码未提交或回滚，导致行锁长时间不释放。
2. **死锁**（占比约 25%）：多个事务互相持有对方需要的锁，形成循环等待。
3. **间隙锁（Gap Lock）**（占比约 15%）：REPEATABLE-READ 隔离级别下范围查询产生间隙锁，阻塞插入。
4. **表锁**（占比约 10%）：使用了 LOCK TABLES 或 DDL 操作导致表锁。

排查思路

1. **查看当前锁等待**：执行 SHOW ENGINE INNODB STATUS ，查看 LATEST DETECTED DEADLOCK 部分，分析死锁事务的 SQL。
2. **查询锁持有情况**：使用 performance_schema 或 information_schema 中的 INNODB_TRX 、 INNODB_LOCKS 、 INNODB_LOCK_WAITS 表。
3. **分析事务代码**：定位长时间未提交的事务，检查是否缺少 commit 或 rollback ，或者存在异常未回滚。
4. **降低隔离级别**：如果业务允许，将隔离级别改为 READ COMMITTED ，减少间隙锁。
5. **优化索引**：确保 UPDATE 、 DELETE 使用索引，避免锁表或锁大量行。
6. **拆分大事务**：将批量操作拆分为多个小事务，减少锁持有时间。

保养提示

- 所有数据库事务必须设置超时时间（如 Spring @Transactional(timeout=30) ）。
- 部署死锁监控告警， 一旦出现死锁立即通知 DBA 分析。
- 在代码中添加重试机制：死锁发生后等待数毫秒后重试事务。

F-S-06-04：数据丢失或损坏

典型现象

- 某些记录不翼而飞，或字段值变成乱码 / NULL。
- 业务报错（如主键冲突、外键错误）提示数据不一致。
- 数据库表出现“使用中”状态，需要修复。

可能原因

1. **未开启binlog**（占比约 40%）：数据库没有开启二进制日志，误操作后无法基于时间点恢复。
2. **硬盘故障**（占比约 25%）：磁盘损坏、坏道导致数据文件损坏。
3. **误删除/误更新**（占比约 20%）：运维或开发人员执行了 DELETE 或 UPDATE 未加 WHERE 条件。
4. **应用程序 BUG**（占比约 10%）：代码逻辑错误，覆盖了不该覆盖的数据。
5. **主从同步错误**（占比约 5%）：从库执行了写入操作导致数据不一致，或同步中断后恢复时产生冲突。

排查思路

1. **确认丢失范围**：通过审计日志或应用日志找到最后正确状态的时间点。
2. **检查 binlog 是否开启**：执行 SHOW VARIABLES LIKE 'log_bin' 。若未开启，后续必须开启。
3. **使用备份恢复**：如果有定期全备 + binlog，可基于时间点恢复数据到指定时间。
4. **分析慢日志和错误日志**：查看是否有 Disk full 、 I/O error 等硬件相关错误。
5. **检查误操作记录**：查看数据库通用日志或审计插件，定位执行危险语句的账号、 IP、时间。
6. **修复损坏表**：使用 CHECK TABLE 和 REPAIR TABLE （InnoDB 建议用 mysqldump 导出再导入）。

保养提示

- 必须开启 binlog，并设置 expire_logs_days 至少 7 天。
- 制定定期备份策略（如每天全备 + 每小时增量备份），并定期演练恢复流程。
- 对重要数据表启用“回收站”功能（如通过逻辑删除 is_deleted 字段）。
- 数据库账号权限最小化，禁止使用 root 操作业务数据。

F-S-06-05：主从同步延迟

典型现象

- 在主库写入数据后，从库查询不到最新数据。
- 监控显示 Seconds_Behind_Master 持续大于 0 且不断增大。
- 读写分离环境下，用户操作后刷新页面数据不显示。

可能原因

1. **从库性能差**（占比约 40%）：从库硬件配置低于主库，或从库上执行了分析查询占用资源。
2. **主库写入压力大**（占比约 30%）：主库大量 DML 操作，binlog 生成速度超过从库回放速度。
3. **大事务**（占比约 15%）：主库执行了长时间的事务（如批量删除百万行），从库回放该事务时同样耗时。
4. **网络延迟**（占比约 10%）：主从间网络带宽不足或延迟高，影响 binlog 传输。
5. **从库复制线程异常**（占比约 5%）：SQL 线程卡死或锁等待。

排查思路

1. **查看延迟值**：在从库执行 SHOW SLAVE STATUS\G ，关注 Seconds_Behind_Master 、Relay_Log_Space 、 Slave_IO_Running 、 Slave_SQL_Running 。
2. **对比主从配置**：检查从库的 innodb_buffer_pool_size 、 innodb_flush_log_at_trx_commit 等参数是否与主库一致且合理。
3. **分析从库负载**：查看从库的 CPU、I/O 使用率，是否有其他查询占用资源。可暂停从库上的统计任务，观察同步延迟是否下降。
4. **拆分大事务**：将主库的大事务拆分为多个小事务（如每 1000 条提交一次），减少单事务回放时间。
5. **升级网络带宽**：监控主从之间的网络流量，确保带宽未饱和。
6. **启用并行复制**：MySQL 5.7+ 可设置 slave_parallel_workers ，加快回放速度。

保养提示

- 配置 slave_parallel_workers （如 4-8 个），开启并行复制。
- 监控 Seconds_Behind_Master 设置告警阈值（如 30 秒）。
- 对读写分离架构的关键查询（如库存、余额）可强制路由到主库。

F-S-06-06：定时清理任务未执行

典型现象

- 数据库表持续膨胀，磁盘使用率报警。
- 日志表明应每天凌晨清理过期数据，但数据仍存在。
- 手动执行清理脚本可正常删除。

可能原因

1. **调度器故障**（占比约 40%）：操作系统 crontab、K8s CronJob 或 Spring Scheduled 配置错误，或调度进程未运行。
2. **清理条件错误**（占比约 30%）：WHERE 条件中的时间字段与实际不符（如 create_time vs update_time），或时区不一致导致今天凌晨删除的是昨天凌晨之前的数据，误以为未清理。
3. **权限不足**（占比约 15%）：执行清理任务的数据账户缺少 DELETE 权限。
4. **任务冲突**（占比约 10%）：多个清理任务同时运行，后一个因表锁等待而失败。
5. **磁盘空间满了**（占比约 5%）：清理任务需要额外的临时空间或 binlog 空间，磁盘满导致回滚。

排查思路

1. **检查调度配置**：
   - 如果是 crontab：查看 /var/log/cron 确认任务是否按时触发。
   - 如果是 K8s CronJob：kubectl get jobs 查看执行历史。
   - 如果是 Spring Scheduled：检查应用日志中是否有 Executing scheduled task 的记录。
2. **手动执行清理 SQL**：在数据库中执行清理语句，观察删除行数及错误信息。
3. **核对清理条件**：确认 WHERE 子句是否正确（例如 DATE(create_time) < CURDATE() - INTERVAL 90 DAY），用 SELECT COUNT(*) 测试待删除数据量。
4. **检查账号权限**：SHOW GRANTS FOR 'task_user'@'%' 确认有 DELETE 权限。
5. **查看表锁**：清理任务执行时是否有其他长事务锁表，导致清理等待超时。
6. **检查是否因磁盘满回滚**：看错误日志是否有 The table is full 或 No space left on device 。

保养提示

- 为清理任务设置详细日志，记录每次删除的行数和耗时，并发送报告。
- 使用分区表代替删除（按月分区），直接 DROP PARTITION 效率更高且不产生大量日志。
- 监控磁盘使用率，低于 20% 时提前预警，避免任务因磁盘满而失败。

F-S-06-07：数据库连接超时或连接已关闭

典型现象

- 应用日志中出现 Communications link failure 、 Connection timed out 或 Connection is closed 。
- 服务运行一段时间后出现间歇性数据库访问失败。
- 重启服务后恢复正常。

可能原因

1. **数据库 wait_timeout 超时**（占比约 50%）：数据库默认 wait_timeout （如 28800 秒）内，客户端空闲连接被服务端主动断开，但连接池未检测到。
2. **网络设备超时**（占比约 25%）：防火墙或负载均衡对空闲 TCP 连接设置了较短的超时（如 300 秒），导致连接被中间设备关闭。
3. **连接池验证查询未配置**（占比约 15%）：连接池未设置 testWhileIdle 或 validationQuery ，无法检测到失效连接。
4. **数据库重启或网络闪断**（占比约 10%）：数据库实例重启或交换机故障，导致已有连接失效。

排查思路

1. **检查数据库超时设置**：执行 SHOW VARIABLES LIKE 'wait_timeout' 和 interactive_timeout 。默认 8 小时，若业务空闲可能触发。
2. **检查防火墙策略**：确认是否有会话超时配置（如阿里云的 SLB 空闲超时 60 秒）。若有，调整防火墙或使用 TCP KeepAlive。
3. **配置连接池**：设置 validationQuery="SELECT 1" ， testWhileIdle=true ， timeBetweenEvictionRunsMillis 小于 60 秒。
4. **开启 TCP KeepAlive**：在数据库连接 URL 中添加 ?socketTimeout=60000 ，或在操作系统层面启用keepalive。
5. **查看数据库错误日志**：确认是否发生过重启或网络故障。

保养提示

- 将数据库 wait_timeout 与连接池的 maxLifetime 统一配置（如 maxLifetime 比 wait_timeout 小 30秒）。
- 使用 HikariCP 推荐配置： keepaliveTime=120000 （2 分钟）发送心跳保持连接活跃。
- 定期检查防火墙和负载均衡的超时设置，确保大于连接池的最长空闲时间。