# 项目优化分析报告

本文基于当前票务系统的实际代码与配置，围绕以下问题形成优化建议： - 熔断和降级机制由哪些中间件完成（结合本项目架构） - OpenFeign 和 Feign 的区别与功能/性能差异 - 用户请求到数据库是否是多线程连接？线程池的上下游模块是什么？线程/连接池规模是多少？多线程、异步与高并发的关系 - Redisson + Lua 原子化操作与本项目各中间件/技术栈之间的直接关联，以及如何配合取舍与优化 - RocketMQ 应该用在哪些模块之间、如何实现异步消息队列、并发能力可提升的量化预期 - 数据一致性落地样例（Outbox、事务消息、幂等Key+Lua、DDL与@Version）

## 一、熔断和降级机制由哪些中间件完成

* 提问
  + 根据本项目架构，熔断和降级机制是由哪些中间件来做？
* 分析
  + 本项目核心依赖：
    - Sentinel：已在网关和业务服务中接入。
      * 证据：
        + ticket-gateway/src/main/resources/application.yml 中 spring.cloud.sentinel 已配置（dashboard、eager 等）。
        + ticket-order/pom.xml 已引入 spring-cloud-starter-alibaba-sentinel。
    - Spring Cloud Gateway：具备 Retry、超时、全局过滤器等能力，可与 Sentinel 网关适配做入口级限流/熔断与统一降级。
    - 说明性澄清：
      * RocketMQ 负责异步解耦/削峰/重试，不直接承担调用链层面的熔断/降级。
      * Nacos 做注册与配置中心，不承担熔断/降级。
      * Redis/Redisson 提供缓存、分布式锁和限流实现细节，不是熔断/降级中间件。
* 落地建议
  + 在 Sentinel Dashboard 为关键接口配置：
    - 网关入口流控（API 分组、阈值、队列等待/直接拒绝策略）。
    - 熔断降级（异常比例/慢调用比例、RT 阈值、熔断时长）。
    - 热点参数限流（如按 userId/ticketId）。
  + 为 OpenFeign 客户端补充 fallback/fallbackFactory，保证下游不可用时的业务兜底。
  + 网关保留合理的 Retry 与超时，并与 Sentinel 的限流/熔断阈值匹配，避免重试风暴。

## 二、OpenFeign 与 Feign 的区别与功能/性能差异

* 提问
  + OpenFeign 和 Feign 有什么区别？在功能和性能上差异如何？
* 分析
  + 名词对齐：
    - Feign（Netflix Feign）：早期的声明式 HTTP 客户端，官方已停更归档。
    - OpenFeign：社区延续版本，仍活跃维护。
    - Spring Cloud OpenFeign：Spring 对 OpenFeign 的整合与自动装配（项目中使用的即此）。
  + 功能差异：
    - 注解与开发体验：Spring Cloud OpenFeign 支持 Spring MVC 注解（@GetMapping/@PostMapping 等），与 Spring 生态一致；基础 OpenFeign/Feign 多用 @RequestLine 等自有注解。
    - 生态整合：Spring Cloud OpenFeign 开箱集成 Spring Boot 自动配置、Spring Cloud LoadBalancer/Nacos、Actuator、Micrometer 指标、日志、压缩、拦截器、超时/重试等。
    - 服务治理：Spring Cloud OpenFeign 更易与 Sentinel/Resilience4j 集成，支持 fallback/fallbackFactory。
  + 性能要点：
    - 运行期每次调用的“框架额外开销”相对网络 IO 毫秒级延迟是微乎其微的（微秒级），总体差异可忽略。
    - 性能主要取决于：底层 HTTP 客户端（建议 OkHttp）、连接池复用、序列化、重试 与 超时策略、负载均衡策略。
* 落地建议
  + 若高并发场景下需要更稳定的连接复用，建议：
    - 引入 feign-okhttp 作为底层 HTTP 客户端，并配置 OkHttp 连接池参数（如 maxIdleConnections、keepAliveDuration）及 connect/read/write 超时。
    - 控制重试策略与退避参数，避免放大流量；结合 Sentinel 做限流/熔断兜底。
  + 在生产环境减少全量详细日志，保留 Micrometer 指标与采样日志用于定位问题。

## 三、从用户请求到数据库的多线程/连接池与并发关系

* 提问
  + 用户请求到数据库是多线程连接的吗？这个线程池的前后分别是什么功能模块？这个线程池有多少条线程？多线程的实现和异步操作以及高并发有什么关系？
* 分析
  + 链路与并发模型：
    - 前置（入口与转发）：Spring Cloud Gateway（Reactor Netty，少量事件循环线程，非阻塞 IO），配置了 httpclient.pool elastic、max-connections=500。
    - 应用 Web 层：Spring MVC（典型为 Tomcat 工作线程池，阻塞式），每个请求占用一个工作线程直至返回。
    - 数据访问层：MyBatis/JDBC 从 HikariCP 连接池借出连接执行 SQL。一个连接一次只服务一个线程，连接池用于多请求并发。
    - 数据库后端：MySQL 线程处理该连接上的 SQL。
    - 旁路组件：
      * Redis：Redisson（Netty 线程 + 连接池）处理缓存/分布式锁。
      * RocketMQ：消息异步处理生产/消费线程。
      * 应用内部异步：Spring Task/Async 线程池处理后台任务。
  + 规模与配置（当前项目实配）——以 ticket-order 为例：
    - HikariCP 连接池：
      * 写库 HighConcurrencyWritePool：maximum-pool-size=100，minimum-idle=20，超时/泄漏检测等已配置。
      * 读库 HighConcurrencyReadPool：maximum-pool-size=200，minimum-idle=50。
    - Web 工作线程：未在 yml 中显式配置，Tomcat 默认通常约 200，可按需设置 server.tomcat.threads.max。
    - 网关连接池：Reactor Netty max-connections=500（连接数，不是线程数）。
    - Redis/Redisson：threads=16，nettyThreads=32，connectionPoolSize=64。
    - 应用异步线程池（订单服务）：execution.pool core=10、max=50、queue=200；scheduling.pool.size=5。
  + 结论：
    - 是的，从用户请求到数据库这一段是“多线程处理 + 连接池化”的模式。线程池承载业务执行并与连接池配合实现并发访问数据库。
    - 并发能力受到多个限流阀共同影响：Web 工作线程上限、Hikari 连接池上限、数据库可并发执行能力、外部依赖（Redis/MQ）等。
    - 仅靠增加线程并不能无限提高吞吐，阻塞调用（JDBC）下应在“线程数 × 连接池大小 × 超时/队列”之间取得平衡，辅以异步解耦与限流熔断。
  + 多线程、异步与高并发的关系：
    - 多线程让阻塞型工作并行执行，但会带来上下文切换、内存占用和锁竞争成本。
    - 异步/非阻塞（如网关层、异步下游）提升单线程的连接处理能力，降低线程需求。
    - 异步解耦（RocketMQ）缩短主链路耗时，把耗时任务移到后台，显著提升系统总体并发与稳定性。
    - Sentinel 限流/熔断在高压下保护线程池与连接池，避免雪崩。
* 落地建议
  + 线程池 × 连接池匹配：
    - 将 Web 工作线程数与 Hikari 连接池大小匹配，避免“线程数远大于连接数”导致普遍阻塞或“连接数远大于线程数”导致资源浪费。
    - 结合数据库 max\_connections、慢查询阈值与锁竞争观察实际承载力，按压测结果微调写库=100、读库=200 是否合适。
  + 严格的超时与队列管理：
    - Hikari connection-timeout、SQL 超时（socketTimeout）、Web 层超时、网关 response-timeout、Feign connect/read 超时需成体系配置，避免“同步等待过长”占满线程与连接。
    - 网关与客户端重试要有上限与退避，避免重试放大流量。
  + 强化异步与削峰：
    - 坚持 RocketMQ 解耦订单创建后续流程、库存同步、通知等，减少主链路阻塞时间。
  + Sentinel 防护：
    - 按接口/资源设置限流、熔断与热点参数，优先保护核心链路，避免一处慢查询拖垮全局。
  + ThreadLocal 使用规范：
    - UserContext 使用 ThreadLocal 存 userId，务必在过滤器/拦截器中 try-finally 调用 clear()，防止在线程复用场景发生内存/数据泄漏。

## 四、Redisson + Lua 原子化操作与各中间件/技术栈的直接关联

* 提问
  + Redisson + Lua 的原子化操作与本项目中哪些中间件/技术栈有直接功能与性能上的关联？如何配合取舍与优化？
* 分析
  + 与 MySQL/HikariCP（数据一致性与并发控制）
    - 作用边界：Redis+Lua 处理“抢占/预扣/校验”等强原子环节；MySQL 负责最终账本的一致性持久化。
    - 协作：预扣在 Redis，确认/回滚落库；缩短数据库事务与锁时间，降低 Hikari 活跃连接占用。
  + 与 RocketMQ（异步解耦与最终一致性）
    - 作用边界：Redis+Lua 做原子状态与幂等标记；MQ 承接后置处理（扣减确认、通知、统计、同步落库）。
    - 协作：Lua 写幂等Key/状态 → 发 MQ → 消费者据标记入库/回滚；DLQ 与补偿兜底。
  + 与 Sentinel（限流/熔断/保护）
    - 作用边界：Sentinel 在入口限流、熔断；Lua 在资源侧做原子扣减。
    - 协作：对热点参数（ticketId/userId）配置限流，减少热Key脚本压力；双层防护。
  + 与 Spring Cloud Gateway（入口治理）
    - 作用边界：网关做前置校验、超时/重试；Lua 做最终是否放行的原子判断。
    - 协作：限制重试与退避，避免对 Redis 热Key的放大冲击；与 Lua RT/错误策略匹配。
  + 与 OpenFeign（同步RPC）
    - 作用边界：Lua 先快速决策（库存不足快速失败/放行），Feign 再同步调用下游。
    - 协作：配置超时与 fallback，减少因下游等待导致的线程占用。
  + 与 Redisson 自身（分布式结构）
    - 作用边界：Lua 适合少量Key的原子多步逻辑；RLock/RRateLimiter 适合粗粒度串行化与令牌桶限流。
    - 取舍：强原子且多步合并优先 Lua；通用互斥/限流优先原生结构。
* 落地建议
  + Lua 使用场景
    - 库存扣减、座位锁定、幂等标记、状态原子迁移（单/少量Key，多步合并）。
    - 用 EVALSHA 缓存脚本，减少编译/传输成本；对脚本做版本管理与回滚策略。
  + 与入口治理联动
    - Sentinel：对热点参数做优先限流（阈值 < Redis 能承载的峰值）。
    - 网关 Retry：配置 retries + 指数退避（如首次10ms、最大50ms、factor=2），避免重试风暴。
  + 与 MQ 的一致性闭环
    - Lua 写入幂等/事务标记 → 发 MQ → 消费者检查标记后落库或回滚 → 必要时回写状态。
    - 异常进入 DLQ 并有定时补偿扫描幂等Key。
  + 与 MySQL 调优配合
    - Redis 预扣 + DB 确认降低锁争用；数据库侧保证关键索引和短事务；连接池上限与 Redis 吞吐匹配。
  + 客户端与线程资源
    - Redisson：根据 CPU 设置 threads/nettyThreads（如 2×CPU）；connectionPoolSize 与并发相称，避免过大。
    - 应用：对执行 Lua 的调用设置合理超时，避免线程长期阻塞。
  + 常见坑规避
    - 序列化：统一 JsonJacksonCodec，Lua 中明确类型，避免反序列化异常。
    - 热点倾斜：必要时对热Key分片（如根据区/排分段库存），并配合 Sentinel 热点限流。
    - 脚本复杂度：避免在 Lua 中做大循环/大集合遍历，把耗时逻辑转移到 MQ 异步。

## 五、RocketMQ 用在哪些模块之间、如何实现异步消息队列、并发提升预期

* 提问
  + RocketMQ 应该用在什么模块之间？如何实现可靠的异步消息？并发能力能提升多少？
* 分析
  + 模块边界与消息流向
    - 订单主链路解耦：ticket-order → ticket-show（库存确认/同步、统计更新）。
    - 通知与触达：ticket-order → notification（SMS/Email/Push）。
    - 支付后处理：payment-service → ticket-order（支付事件）；ticket-order → ticket-show（最终确认/回滚）。
    - 状态/缓存同步：ticket-order → cache-invalidator；ticket-show → analytics（实时统计）。
    - 异常补偿：各业务 → DLQ handler（%DLQ%TOPIC）。
    - 推荐 Topic/Tag：
      * ORDER\_TOPIC: ORDER\_CREATED, ORDER\_PAID, ORDER\_CANCELLED
      * STOCK\_TOPIC: STOCK\_DEDUCT, STOCK\_CONFIRM, STOCK\_SYNC, STOCK\_ROLLBACK
      * NOTIFICATION\_TOPIC: SMS, EMAIL, PUSH
  + 异步实现要点（可靠、可观测、可扩展）
    - 生产端：
      * 事务一致性：优先 Outbox（本地事务写订单+Outbox，定时扫描发送），或 RocketMQ 事务消息（半消息+回查）。
      * 幂等与Key：Message Key=orderNo；Redis 幂等Key order:msg:{orderNo}:{event}。
      * 发送策略：关键事件 syncSend（支付/取消），其余 asyncSend；压缩阈值≥4KB，重试2~3次+退避。
      * 顺序语义：按 orderNo 做一致性哈希到队列，保障单订单有序。
    - 消费端：
      * 并发：consumeThreadMin/Max≈CPU×2~4；queueNum 16~64；pull/consume batch 16~32。
      * 幂等：先查幂等Key或状态机字段，重复即丢弃；小事务+短SQL。
      * 回压：下游抖动时降低并发或转重试队列，避免风暴。
      * DLQ：超重试进DLQ；提供补偿与人工介入通道。
    - 观测与运维：
      * 指标：发送成功率、消费RT、积压深度、DLQ速率。
      * 告警：积压阈值、DLQ增长、失败率。
      * 追踪：使用 message key（orderNo）关联业务日志。
  + 并发与性能提升（量化预期）
    - 主链路RT：
      * 改造前：同步串行做查价/扣减/落库/通知，RT 600~1500ms（示例）。
      * 改造后：仅“校验+最小订单+入队” 100~300ms。
      * 预期：P95/P99 下降 3~10 倍（依赖后置步骤耗时）。
    - 服务吞吐/TPS：
      * 在现有参数（Hikari 写池100、读池200、网关连接池500、Redisson池64）下：
        + 订单创建接口 TPS：保守 2~4 倍；若库存确认/通知完全异步且 DB 写优化到位，可达 5x+（峰值更高）。
        + 全链路承载：依赖队列削峰，可把短时突发从稳态 ~1500 TPS 提升到 10k TPS 级别的峰值承载（以积压换延迟）。
    - 边界条件：
      * 上限受 DB 最终一致性写能力、热点Key策略（Sentinel 热点限流 + Redis 分片）、消费者并发与下游IO能力限制。
      * 消费者侧若存在大事务/慢SQL，积压即转化为延迟。
* 落地建议
  + 流程改造最小集：
    - ticket-order.createOrder：快速落“最小订单”→ asyncSend ORDER\_CREATED。
    - order-process-consumer：查价 + Redis/Lua 预扣 + 更新详情 → send STOCK\_DEDUCT。
    - 支付成功：syncSend ORDER\_PAID（保证一致性）→ 消费端做最终确认/回滚 + 扇出通知。
    - 通知统一由 NOTIFICATION\_TOPIC 承接，独立消费扩容。
  + 参数起步建议：
    - Topic queueNum：ORDER/NOTIFICATION=32，STOCK=16。
    - Producer：sendTimeout=3s、retry=2、压缩阈值=4KB、retryNextServer=true。
    - Consumer：concurrency=16~32、pullBatchSize=32、consumeBatchMaxSize=16、顺序=按orderNo一致性哈希。
    - 幂等：Redis 幂等Key TTL 1~3天；DB 状态机单向推进（CREATED→PAID→CANCELLED）。
  + 治理联动：
    - Sentinel：对创建/支付回调/库存接口做热点限流，入口阈值略低于 Redis/Lua 的稳定峰值能力。
    - 网关 Retry：限制 2~3 次 + 指数退避（10ms→50ms，factor=2），避免重试风暴。
    - 监控：积压阈值、DLQ告警、消费失败率，定期回放抽查。

## 六、数据一致性落地样例（可直接参考）

* 提问
  + 给出 Outbox 表结构与 MyBatis-Plus 实体/DAO/定时任务样例；给出 RocketMQ 事务消息的本地事务、回查逻辑样例；提供“幂等Key约定 + Lua 脚本片段 + 消费端幂等过滤”的代码模板；提供 DDL（订单号唯一索引、库存 version 字段）与 @Version 使用示例。
* 样例一：Outbox 表 + 实体/DAO/定时任务
  + DDL：
  + CREATE TABLE outbox\_event (  
     id BIGINT PRIMARY KEY AUTO\_INCREMENT,  
     aggregate\_type VARCHAR(64) NOT NULL, -- 例如 ORDER  
     aggregate\_id VARCHAR(64) NOT NULL, -- 例如 orderNo  
     event\_type VARCHAR(64) NOT NULL, -- CREATED/PAID/CANCELLED  
     payload JSON NOT NULL,  
     status TINYINT NOT NULL DEFAULT 0, -- 0:NEW 1:SENT 2:FAILED  
     retry\_count INT NOT NULL DEFAULT 0,  
     created\_at DATETIME NOT NULL DEFAULT CURRENT\_TIMESTAMP,  
     updated\_at DATETIME NOT NULL DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,  
     UNIQUE KEY uk\_outbox (aggregate\_type, aggregate\_id, event\_type, created\_at)  
    ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
  + 实体与 Mapper（MyBatis-Plus）：
  + @Data  
    @TableName("outbox\_event")  
    public class OutboxEvent {  
     @TableId(type = IdType.AUTO)  
     private Long id;  
     private String aggregateType;  
     private String aggregateId;  
     private String eventType;  
     private String payload; // JSON 字符串  
     private Integer status; // 0 NEW, 1 SENT, 2 FAILED  
     private Integer retryCount;  
     private LocalDateTime createdAt;  
     private LocalDateTime updatedAt;  
    }  
      
    public interface OutboxEventMapper extends BaseMapper<OutboxEvent> {}
  + 事务写入（与订单最小落库同一事务）：
  + @Transactional  
    public void createMinimalOrderAndOutbox(Order order, Object eventPayload) {  
     orderMapper.insert(order);  
     OutboxEvent evt = new OutboxEvent();  
     evt.setAggregateType("ORDER");  
     evt.setAggregateId(order.getOrderNo());  
     evt.setEventType("CREATED");  
     evt.setPayload(objectMapper.writeValueAsString(eventPayload));  
     evt.setStatus(0);  
     outboxEventMapper.insert(evt);  
    }
  + 定时任务扫描发送：
  + @Component  
    public class OutboxRelayJob {  
     @Autowired OutboxEventMapper mapper;  
     @Autowired RocketMQTemplate rocketMQTemplate;  
      
     @Scheduled(fixedDelay = 3000)  
     public void relay() {  
     List<OutboxEvent> list = mapper.selectList(new QueryWrapper<OutboxEvent>()  
     .lambda().eq(OutboxEvent::getStatus, 0).last("limit 200"));  
     for (OutboxEvent e : list) {  
     try {  
     rocketMQTemplate.syncSend("ORDER\_TOPIC:ORDER\_CREATED", e.getPayload());  
     e.setStatus(1);  
     e.setRetryCount(e.getRetryCount()+1);  
     } catch (Exception ex) {  
     e.setRetryCount(e.getRetryCount()+1);  
     if (e.getRetryCount() >= 5) e.setStatus(2);  
     } finally {  
     mapper.updateById(e);  
     }  
     }  
     }  
    }
* 样例二：RocketMQ 事务消息（本地事务 + 回查）
  + 生产发送：
  + @Autowired RocketMQTemplate mqTemplate;  
      
    public void sendOrderCreatedTxn(String orderNo, Order payload) {  
     mqTemplate.sendMessageInTransaction(  
     "ORDER\_TXN\_GROUP",  
     "ORDER\_TOPIC:ORDER\_CREATED",  
     MessageBuilder.withPayload(payload).setHeader("orderNo", orderNo).build(),  
     orderNo // arg  
     );  
    }
  + 事务监听器：
  + @RocketMQTransactionListener(txProducerGroup = "ORDER\_TXN\_GROUP")  
    public class OrderTxnListener implements RocketMQLocalTransactionListener {  
     @Autowired OrderMapper orderMapper;  
      
     @Override  
     public RocketMQLocalTransactionState executeLocalTransaction(Message msg, Object arg) {  
     String orderNo = (String) arg;  
     try {  
     // 本地事务：落最小订单  
     Order o = new Order(); o.setOrderNo(orderNo); /\*...\*/  
     orderMapper.insert(o);  
     return RocketMQLocalTransactionState.COMMIT;  
     } catch (Exception e) {  
     return RocketMQLocalTransactionState.ROLLBACK;  
     }  
     }  
      
     @Override  
     public RocketMQLocalTransactionState checkLocalTransaction(Message msg) {  
     String orderNo = (String) msg.getHeaders().get("orderNo");  
     Order o = orderMapper.selectByOrderNo(orderNo);  
     if (o != null) return RocketMQLocalTransactionState.COMMIT;  
     return RocketMQLocalTransactionState.ROLLBACK;  
     }  
    }
* 样例三：幂等Key约定 + Lua 脚本 + 消费端幂等过滤
  + 幂等Key约定：idempo:{biz}:{orderNo}:{event}，TTL 1~3 天。
  + Lua（示例：库存预扣 + 幂等标记原子化）：
  + -- KEYS[1] = stock\_key (ticket:{id}:remain)  
    -- KEYS[2] = idempo\_key (idempo:order:{orderNo}:DEDUCT)  
    -- ARGV[1] = qty, ARGV[2] = ttlSeconds  
    if redis.call('EXISTS', KEYS[2]) == 1 then  
     return 2 -- already processed  
    end  
    local remain = tonumber(redis.call('GET', KEYS[1]) or '0')  
    local need = tonumber(ARGV[1])  
    if remain < need then  
     return 0 -- not enough  
    end  
    redis.call('DECRBY', KEYS[1], need)  
    redis.call('SETEX', KEYS[2], tonumber(ARGV[2]), '1')  
    return 1 -- success
  + Java 执行（Redisson 执行脚本略）：返回 1 成功、0 余量不足、2 幂等命中。
  + 消费端幂等过滤：
  + public boolean tryMarkIdempotent(String key, long ttlSeconds) {  
     // SET NX EX  
     return Boolean.TRUE.equals(stringRedisTemplate.opsForValue()  
     .setIfAbsent(key, "1", Duration.ofSeconds(ttlSeconds)));  
    }  
      
    public void onMessage(OrderEvent msg) {  
     String key = String.format("idempo:order:%s:%s", msg.getOrderNo(), msg.getEventType());  
     if (!tryMarkIdempotent(key, 2\*24\*3600)) return; // 已处理  
     // 业务处理... 小事务 + 短SQL  
    }
* 样例四：DDL（唯一索引、版本字段）与 @Version 使用
  + DDL：
  + ALTER TABLE t\_order ADD UNIQUE KEY uk\_order\_no (order\_no);  
      
    ALTER TABLE t\_ticket\_stock  
     ADD COLUMN version INT NOT NULL DEFAULT 0,  
     ADD INDEX idx\_ticket\_id (ticket\_id);
  + 实体与 @Version：
  + @Data  
    public class TicketStock {  
     private Long id;  
     private Long ticketId;  
     private Integer totalStock;  
     private Integer remainStock;  
     @Version  
     private Integer version;  
    }  
      
    // 更新示例：where id=? and version=?，成功则 version+1  
    int updated = ticketStockMapper.updateById(entity);  
    if (updated == 0) {  
     // 版本冲突，重试或走失败分支  
    }
* 结论与建议（数据一致性）
  + 项目已具备最终一致性基础（Redis+Lua 原子、MQ 重试与DLQ、DB账本），建议：
    - 引入 Outbox 或 RocketMQ 事务消息，保证“本地写入与消息发送”一致性。
    - 全链路幂等：消息键 = orderNo，Redis 幂等Key + DB 状态机单向推进，重复消息安全。
    - 消费端小事务 + 索引优化，避免积压转化为长延迟。
    - Sentinel/Gateway 热点限流与退避重试，保护 Redis 热Key 与 MQ/DB 下游。

## 附：可选的进一步优化清单

* OpenFeign 切换 OkHttp：引入 feign-okhttp，配置连接池与超时；减少连接建立抖动，提升高并发稳定性。
* SQL 侧优化：为高频读写的表补齐合适索引、批量写入开启 rewriteBatchedStatements，控制事务范围。
* 压测与观测：联动物理指标（CPU/内存/IO）、线程池利用率、Hikari 活跃连接、MySQL 等待事件，找出瓶颈后再定向调参。
* 网关限流前移：对高风险接口在网关做入口级限流/令牌桶，缓解下游压力。