# YTS - 乐观锁

由 沈冠初创建, 最后修改于八月 12, 2024

## 1. YTS 锁现状分析

现象：

- 幂等问题：多次调用二阶段 confirm/cancel 方法
- 并发问题：try 请求与 confirm/cancel 请求同时到达

原因：

- 超时重试
- 网络异常

YTS 解决方案：通过唯一索引与事务状态的判断同时解决幂等与并发问题

### 1.1. 开启事务

try 正向第一阶段：详细流程查看：4.1. YTS - TCC 模式
首先向 YTS 事务控制表（yts_trans_date）插入本次事务记录（status = confirming）

1. "tx_id" 做为 unique_key 唯一索引
    a. 若插入成功，说明二阶段还未执行，可继续执行本次正向业务逻辑
    b. 若唯一索引冲突，则插入失败
2. 查询事务控制表，查询的同时增加 for update 写锁
3. .....

afterCompletion 异步发起二阶段，此时事务已经提交

### 1.2. 二阶段开始

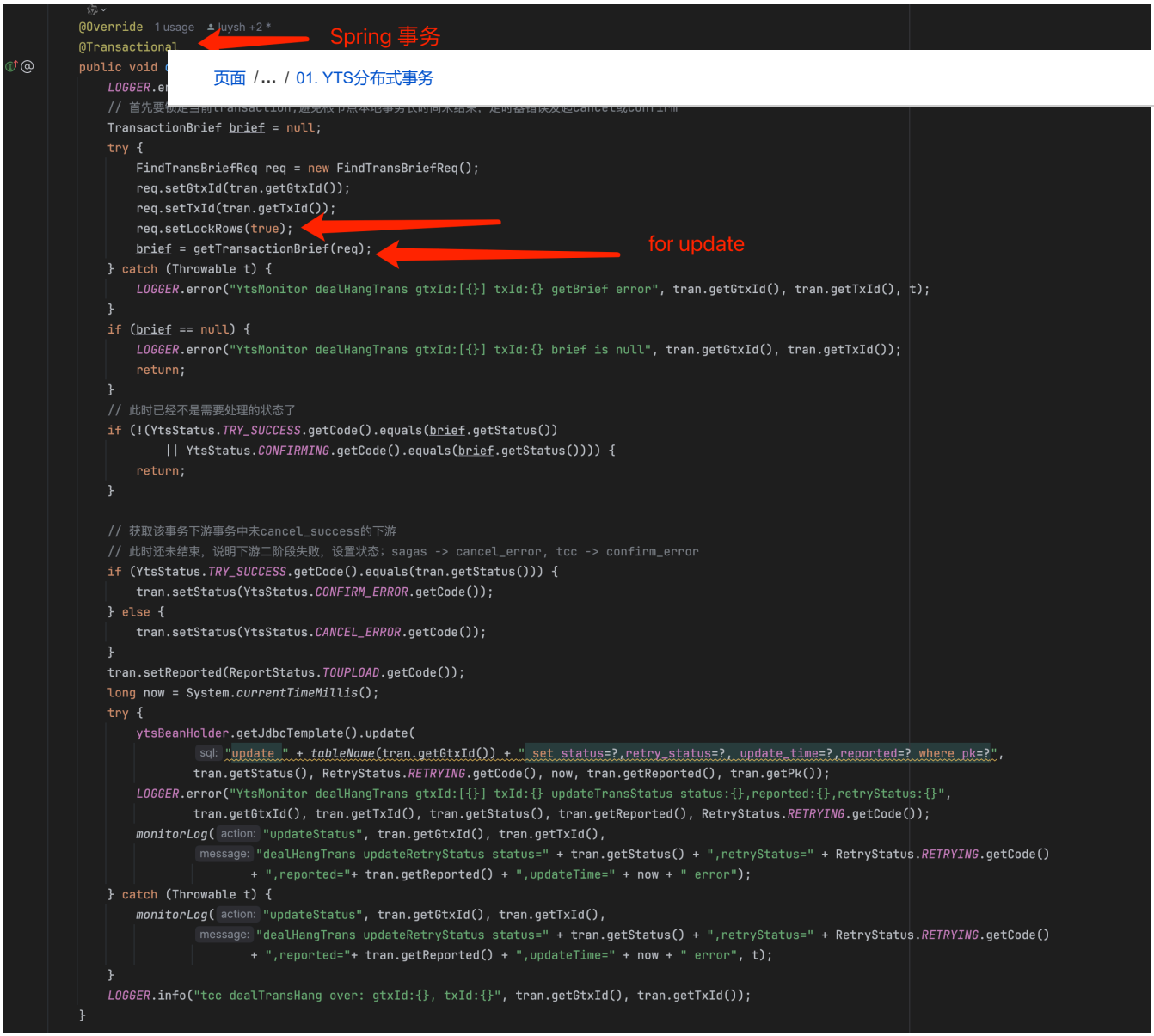
类似一阶段

### 1.3. 定时任务-处理长时间未完成的事务：

com.yonyou.cloud.yts.internal.scheduler.TransHangTask#doTransHangTask：处理长时间未完成的事务，查询当天 yts_trans 表，每分钟执行一次

com/yonyou/cloud/yts/internal/scheduler/TransHangTask.java:117：处理长时间未完成的事务，查询历史 yts_trans 表，每小时执行一次，这里会查询所有的历史表

1. 查询未完成的事务 findUnfinished，查询 yts_trans 表
    a. type = ROOT
    b. status = confirming / try_success
2. 遍历 trans list：超过五分钟仍未完成的事务，并且 mode = tcc / sagas，调用 dealTccConfirming 方法（@Transactional 注解，默认 REQUIRED）
    a. 查询 yts_trans 表，获取事务列表，此时加 for update 锁
    b. 若当前 trans 结果状态为非未完成状态即 !(confirming || try_success) 状态，直接 return
    c. 此时还未结束，说明下游二阶段失败，当前节点状态更新：sagas → cancel_error, tcc → confirm_error
    d. 上报状态设置为待上报：reported → toupload
    e. 事务更新（yts_trans 表）：根据 pk 进行更新




```
@Override 1 usage  luysh +2 *
@Transactional
public void
    LOGGER.er
    // 首先要锁定当前transaction,避免被伪点本地事务长时间未结束，定时器错误发起cancel或confirm
    TransactionBrief brief = null;
    try {
        FindTransBriefReq req = new FindTransBriefReq();
        req.setGtxId(tran.getGtxId());
        req.setTxId(tran.getTxId());
        req.setLockRows(true);
        brief = getTransactionBrief(req);
    } catch (Throwable t) {
        LOGGER.error("YtsMonitor dealHangTrans gtxId:[{}] txId:{} getBrief error", tran.getGtxId(), tran.getTxId(), t);
    }
    if (brief == null) {
        LOGGER.error("YtsMonitor dealHangTrans gtxId:[{}] txId:{} brief is null", tran.getGtxId(), tran.getTxId());
        return;
    }
    // 此时已经不是需要处理的状态了
    if (!(YtsStatus.TRY_SUCCESS.getCode().equals(brief.getStatus())
            || YtsStatus.CONFIRMING.getCode().equals(brief.getStatus()))) {
        return;
    }

    // 获取该事务下游事务中未cancel_success的下游
    // 此时还未结束，说明下游二阶段失败，设置状态；sagas -> cancel_error, tcc -> confirm_error
    if (YtsStatus.TRY_SUCCESS.getCode().equals(tran.getStatus())) {
        tran.setStatus(YtsStatus.CONFIRM_ERROR.getCode());
    } else {
        tran.setStatus(YtsStatus.CANCEL_ERROR.getCode());
    }
    tran.setReported(ReportStatus.TOUPLOAD.getCode());
    long now = System.currentTimeMillis();
    try {
        ytsBeanHolder.getJdbcTemplate().update(
                sql: "update " + tableName(tran.getGtxId()) + " set status=?,retry_status=?, update_time=?,reported=? where pk=?",
                tran.getStatus(), RetryStatus.RETRYING.getCode(), now, tran.getReported(), tran.getPk());
        LOGGER.error("YtsMonitor dealHangTrans gtxId:[{}] txId:{} updateTransStatus status:{},reported:{},retryStatus:{}",
                tran.getGtxId(), tran.getTxId(), tran.getStatus(), tran.getReported(), RetryStatus.RETRYING.getCode());
        monitorLog( action: "updateStatus", tran.getGtxId(), tran.getTxId(),
                message: "dealHangTrans updateRetryStatus status=" + tran.getStatus() + ",retryStatus=" + RetryStatus.RETRYING.getCode()
                        + ",reported="+ tran.getReported() + ",updateTime=" + now + " error");
    } catch (Throwable t) {
        monitorLog( action: "updateStatus", tran.getGtxId(), tran.getTxId(),
                message: "dealHangTrans updateRetryStatus status=" + tran.getStatus() + ",retryStatus=" + RetryStatus.RETRYING.getCode()
                        + ",reported="+ tran.getReported() + ",updateTime=" + now + " error", t);
    }
    LOGGER.info("tcc dealTransHang over: gtxId:{}, txId:{}", tran.getGtxId(), tran.getTxId());
}
```



## 1.4. yts_trans 表锁粒度分析

默认在 MySQL 的 READ_REPEATED 事务隔离级别下进行分析。

trans 表 DDL 建表语句：

主键索引：pk

唯一索引：tx_id

```
CREATE TABLE `yts_trans_20240725` (
  `pk` bigint unsigned NOT NULL AUTO_INCREMENT,
  `id` varchar(64) COLLATE utf8mb4_general_ci NOT NULL,
  `module_name` varchar(100) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `service_name` varchar(64) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `method_name` varchar(64) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `gtx_id` varchar(64) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `tx_id` varchar(64) COLLATE utf8mb4_general_ci NOT NULL,
  `ptx_id` varchar(64) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `cancel_method` varchar(64) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `type` varchar(20) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `status` varchar(40) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `create_time` bigint DEFAULT NULL,
  `interface_name` varchar(255) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `update_time` bigint DEFAULT NULL,
  `confirm_method` varchar(64) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `reported` varchar(16) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `context` text COLLATE utf8mb4_general_ci,
  `env` varchar(100) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `call_tag` varchar(128) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `provider_id` varchar(64) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `invocation` longblob,
  `all_confirmed` int DEFAULT '0',
  `mode` varchar(64) COLLATE utf8mb4_general_ci DEFAULT NULL,
```

```
  `retry_status` varchar(20) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `bill_no` varchar(64) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `trace_id` varc
  `rule_id` varch
  `ytenant_id` varchar(36) COLLATE utf8mb4_general_ci DEFAULT 'N',
  PRIMARY KEY (`pk`),
  UNIQUE KEY `txid` (`tx_id`) USING BTREE,
  KEY `status_all` (`status`,`all_confirmed`) USING BTREE,
  KEY `idx_type_servicename_retrystatus` (`type`,`service_name`,`retry_status`),
  KEY `type_status` (`status`,`type`) USING BTREE,
  KEY `idx_update_time` (`update_time`) USING BTREE,
  KEY `idx_reported` (`reported`,`service_name`,`type`) USING BTREE,
  KEY `i_yts_trans_20240725_ytenant_id` (`ytenant_id`) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=424 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

页面 /... / 01. YTS分布式事务

执行 trans 表加锁查询语句

```
select * from yts_trans_20240725 where gtx_id='20240725092048@9bfa2613-e26d-43a6-b9f1-350306931edd' and tx_id ='3bac2983-eab9-434f-bc52-4d
```

explain 查看执行计划，可以看到使用到的索引为 tx_id, 预计扫描的行数 rows = 1，锁的粒度为行锁

| | id | select_type | table | partitions | type | possible_keys |
|---|---|---|---|---|---|---|
| | 1 | SIMPLE | yts_trans_20240725 | (NULL) | const | txid |

## 1.5. 压测问题分析

压测问题：有大量 YTS 相关的 SQL 语句，长时间持有写锁，占用大量系统资源，导致系统的整体性能下降。

分析：

1. YTS 使用 for update 对性能的影响：
    a. YTS 并不存在针对于同一 gtx_id 与 tx_id 的高并发场景，无论是正向一阶段 try，还是二阶段 confirm/cancel，并发量都很低
    b. 处理未完成的事务-定时任务：在存在大量未完成二阶段的事务时，会有大量的 for update 加锁操作，这种情况较为极端。
2. 长事务问题，YTS 自身设计持有锁时间并不会太长，但是与业务结合之后，事务提交的时机由具体的业务逻辑决定，因此在业务代码提交事务之前 yts 会一直持有写锁，事务提交时，锁才会释放。

总结：长事务问题是业务侧导致的，YTS 在极端场景下（存在大量未完成事务）才会同时持有大量的写锁。

# 2. 乐观锁优化

## 2.1. 优化的必要性

1. 针对压测反应的因为 yts 相关 SQL 导致的持有大量写锁的问题，这个问题的根本原因是业务侧的导致的长事务，长事务问题需要业务侧解决
2. 第一章节分析了三种需要加写锁的场景，这三种场景主要是为了解决
    a. 幂等问题，多次调用二阶段 confirm/cancel 方法：YTS 正常逻辑不存在同时多次调用
        i. try 阶段结束，afterCompletion 钩子方法会异步发起二阶段
        ii. 重试/自动重试，由定时任务进行扫描
    b. 并发问题，try 请求与 confirm/cancel 请求同时到达，这会是因为网络原因导致
3. 针对第二点，我们预期大部分场景下不会出现幂等与并发问题，即数据冲突几率较小，可以采用乐观锁进行优化，减少加锁释放锁的开销，提高性能。

## 2.2. 乐观锁缺点

1. 乐观锁适合读多写少的场景，并且预期是写冲突较少，而 YTS 的场景是写多读少，预期写冲突较少，并不能发挥乐观锁的优势
2. 使用乐观锁，在二阶段请求多次的场景下，因为会重复执行整条链路的业务，乐观锁的性能会比悲观锁要低，并且要注意对业务是否会产生负面影响

## 2.3. 乐观锁设计

### 2.3.1. trans 表新增版本字段

1. 使用现有字段 update_time 作为版本号，类型 bigint
2. 建立 pk 与 update_time 的联合索引
3. update_time = now(6)
    a. 需要使用数据库服务器时间戳，而不是当前系统时间
    b. now(6) 精度为微秒，但仍可能重复，只是概率极小。

```
UPDATE yts_trans_demo SET status = ?, update_time = UNIX_TIMESTAMP(NOW(6)) * 1000000 + MICROSECOND(NOW(6)) WHERE pk = ? AND update_time =
```

- **分布式环境**：在分布式系统中，不同节点可能会并发更新同一条记录，使用时间戳可以确保版本号的唯一性和有序性，避免版本冲突。

- **无需新增字段**：复用现有的 update_time 即可，若新增表字段，会有兼容性问题

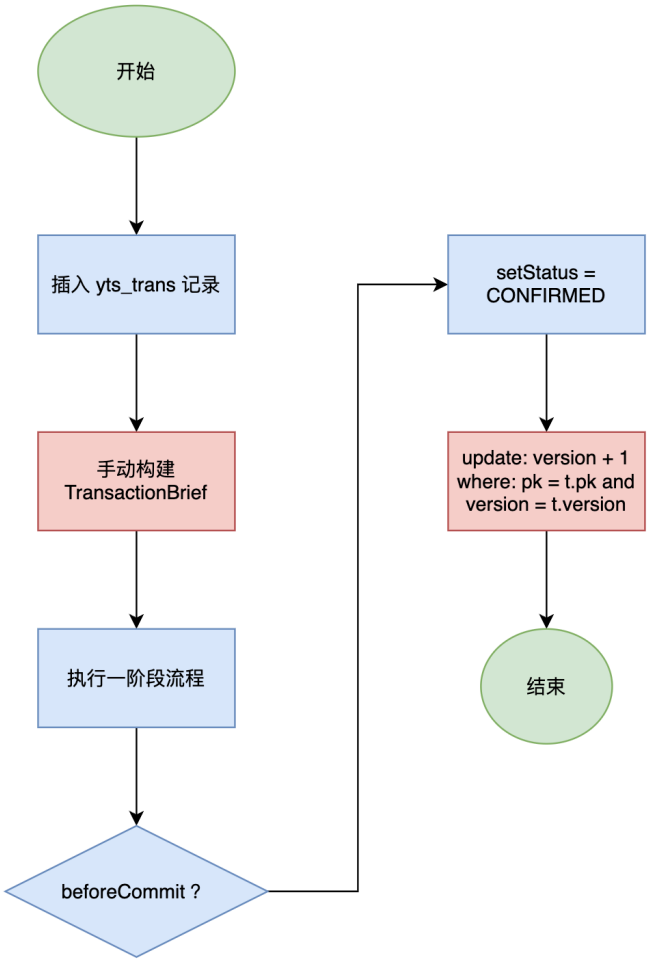
### 2.3.2. startTrans



1. 初始化 trans
2. 持久化 trans （持久化时加入 _version 字段， _version = 1）
3. 加锁 for update （这一步去除，TransactionBrief 改为根据 transaciton 对象信息，手动构建）

### 2.3.3. TransactionHook

1. beforeCommit
    a. setStatus(CONFIRMED)
    b. 持久化
        i. 持久化时 where 条件中根据 trans.version 进行更新
        ii. 创建一个新的持久化方法，根据 pk 与 version，不复用之前的 jdbcTplTransactionService 的 save 方法




（上图为方便表述，使用 version 代表版本号，实际为 update_time）

2. afterCompletion
    1. 无下游，结束事务
        a. updateStatus
    2. 有下游，执行下游
        a. sagas
            i. status = 0 commited
                1. executeTccMethod 开启下游正向二阶段
                    a. 下游二阶段结束
                    b. updateStatus
            ii. status != 0 rollback || unknow
                1. executeSagasCancel 开启回滚
                    a. 下游回滚结束
                    b. updateStatus
        b. tcc

#### 2.3.4. ActionHook

#### 2.3.5. ConfirmHook

#### 2.3.6. irisTccExecut

#### 2.3.7. YmsHttpClientExecutor

#### 2.3.8. MDD



## 2.4. 锁冲突（更新失败）

这里预期是没有其他线程竞争，如果更新时乐观锁冲突，有下面两种情况

1. 二阶段 confirm/cacncel 请求比第一阶段 try 请求先行到达
    a. 直接抛出异常，结束流程。以此来保证并发请求时，数据的一致性。
2. 二阶段 confirm/cancel 请求被多次调用
    a. 直接抛出异常，结束流程。以此来保证多次访问下，幂等性问题。

结合 YTS 的 TCC 与 Sagas 业务特点，为了保证并发与幂等，无重试或可重入机制，直接结束流程。

affectRow = 0 直接抛出异常

```
public void updateStatusWithOptimisticLock(int pk, String status, Long updateTime) {
        String sql = "UPDATE yts_trans_demo SET status = ?, update_time = UNIX_TIMESTAMP(NOW(6)) * 1000000 + MICROSECOND(NOW(6)) WHERE pk
        int rowsAffected = jdbcTemplate.update(sql, status, pk, updateTime);
        if (rowsAffected == 0) {
            throw new RuntimeException("Update failed due to concurrent modification.");
        }
    }
```

## 2.5. 增加开启乐观锁开关

读取环境变量

# 3. 测试

修改 1000 条记录，在没有并发修改的情况下，两者的时间消耗基本一致：

```
Pessimistic lock time cost: 25089ms
Optimistic lock time cost: 24471ms
```

分析：

- **悲观锁**: 在无并发的情况下，虽然悲观锁会加锁，但由于没有其他事务竞争锁资源，数据库引擎能够快速完成锁的获取和释放。加锁的过程不会阻塞其他事务，也不会引发额外的等待时间。

长事务，锁的占用情况

无标签