电信CRM客户中心分布式事务设计方案

文档信息

适用范围： 电信CRM客户中心微服务系统（日4亿次服务调用）
核心目标： 在超大规模分布式场景下保证跨微服务的数据一致性
事务吞吐： 支持百万级/天的分布式事务

一、分布式事务面临的核心难点

1.1 为什么需要分布式事务

在传统单体应用中，可以使用数据库本地事务（ACID）保证数据一致性。但在微服务架构下，一个业务操作可能跨越多个服务和多个数据库，无法使用本地事务。

典型场景举例：

场景一：客户开户流程

1. 客户管理服务创建客户信息
2. 用户管理服务创建登录账号
3. 账户管理服务创建资金账户并初始化余额
4. 权限控制服务分配默认权限
5. 日志监控服务记录开户日志

如果第3步账户创建失败，前面的客户信息和登录账号需要回滚，否则会出现数据不一致。

场景二：账户充值流程

1. 账户管理服务更新账户余额
2. 交易记录服务创建充值交易记录
3. 积分服务增加用户积分
4. 消息通知服务发送充值成功短信

如果第2步交易记录创建失败，余额已经增加，但没有交易记录，会造成账务不清。

1.2 分布式事务的核心难点

难点一：跨服务的数据一致性保证

- 不同服务使用不同的数据库实例，无法用数据库事务保证一致性
- 服务间通过网络通信，可能出现网络延迟、超时、丢包等问题
- 需要在应用层实现分布式事务协调

难点二：性能与一致性的平衡

- 强一致性需要所有服务同步确认，性能较差
- 最终一致性性能好，但业务逻辑复杂，需要处理中间状态
- 需要根据业务特点选择合适的一致性级别

难点三：异常处理和补偿

- 服务调用可能失败、超时、部分成功
- 需要设计补偿逻辑回滚已完成的操作
- 补偿操作本身也可能失败，需要重试机制

难点四：高并发下的性能瓶颈

- 分布式事务需要协调多个服务，延迟较高
- 锁和资源占用时间长，影响并发性能
- 日均300万业务需要高效的事务处理能力

二、分布式事务解决方案选型

2.1 三种主流方案对比

方案	一致性	性能	适用场景	优点	缺点
TCC	强一致性	中	账户充值、支付	实时性好、可控性强	代码侵入性大、实现复杂
Saga	最终一致性	高	客户注销、订单流程	性能好、支持长事务	隔离性差、需设计补偿
本地消息表	最终一致性	高	日志记录、数据同步	实现简单、可靠性高	有延迟、不适合实时场景

2.2 方案选择原则

使用TCC的场景：

- 对一致性要求高的核心业务（如账户充值、支付、转账）
- 需要立即知道操作结果的业务
- 事务流程较短（3-5个步骤）的业务

使用Saga的场景：

- 事务流程较长（5个以上步骤）的业务
- 可以接受短暂数据不一致的业务
- 对性能要求较高的业务
- 如：客户注销、合约签订、订单处理

使用本地消息表的场景：

- 非关键业务（如日志记录、消息通知）
- 跨中心数据同步
- 可以接受较长延迟的业务

三、TCC模式详细设计

3.1 TCC原理

TCC代表**Try-Confirm-Cancel**三个阶段：

Try阶段（资源预留）：

- 检查业务条件是否满足
- 预留业务资源（如冻结账户余额）
- 不实际提交，但要记录操作日志

Confirm阶段（确认提交）：

- 所有Try成功后，执行Confirm
- 使用Try阶段预留的资源完成业务
- Confirm必须成功，失败需要重试

Cancel阶段（取消回滚）：

- 任何Try失败，执行Cancel
- 释放Try阶段预留的资源
- Cancel必须成功，失败需要重试

3.2 账户充值流程设计

业务流程： 用户账户充值100元，需要更新账户余额、创建交易记录、增加积分。

Try阶段实现：

1. **账户服务Try：** 检查账户状态正常，创建冻结记录，标记"待充值100元"
2. **交易服务Try：** 创建交易记录，状态为"处理中"
3. **积分服务Try：** 创建积分变更记录，状态为"待确认"

Confirm阶段实现：

1. **账户服务Confirm：** 更新账户余额+100，删除冻结记录
2. **交易服务Confirm：** 更新交易记录状态为"成功"
3. **积分服务Confirm：** 更新用户积分+10，更新状态为"已确认"

Cancel阶段实现：

1. **账户服务Cancel：** 删除冻结记录
2. **交易服务Cancel：** 更新交易记录状态为"已取消"
3. **积分服务Cancel：** 删除积分变更记录

3.3 TCC实现关键点

幂等性保证：

- 每个TCC方法（Try/Confirm/Cancel）必须支持幂等
- 使用事务ID+步骤ID作为唯一标识
- 在执行前检查是否已经执行过，避免重复执行

并发控制：

- 使用乐观锁（版本号）防止并发修改冲突
- Try阶段锁定资源，其他事务无法操作
- 使用数据库行级锁保证原子性

超时处理：

- 设置Try阶段超时时间（如30秒）
- 超时后自动触发Cancel回滚
- 定时任务扫描超时事务，执行恢复

日志记录：

- 记录每个阶段的执行状态和时间
- 失败时记录详细的错误信息
- 用于问题排查和事务恢复

3.4 TCC性能优化

并行执行Confirm/Cancel：

- Confirm和Cancel阶段的多个服务调用可以并行执行
- 使用CompletableFuture实现异步并行调用
- 性能提升2-3倍

资源池化：

- 事务协调器使用连接池
- 减少频繁创建连接的开销
- 提高并发处理能力

分片存储：

- TCC事务日志分32个分片存储
- 根据事务ID哈希分片
- 避免单表数据量过大

四、Saga模式详细设计

4.1 Saga原理

Saga模式将长事务拆分为多个本地短事务，每个本地事务都有对应的补偿操作。

正向流程： T1 → T2 → T3 → T4 → T5 **补偿流程：** C1 ← C2 ← C3 ← C4 ← C5（逆序执行）

特点：

- 支持长事务（10+个步骤）
- 性能好，每个步骤独立提交
- 最终一致性，存在中间状态
- 需要精心设计补偿逻辑

4.2 客户注销流程设计

业务流程： 客户申请注销，需要终止合约、清算账户、回收权限、注销用户、注销客户信息。

正向流程定义：

步骤1：终止合约

- 查询客户的所有有效合约
- 将合约状态更新为"已终止"
- 记录终止时间和操作人
- 补偿操作：恢复合约状态为"有效"

步骤2：清算账户

- 查询账户余额
- 如果有余额，生成退款单
- 将余额设为0
- 补偿操作：恢复原有余额

步骤3：回收权限

- 查询客户的所有权限
- 删除或禁用所有权限记录
- 补偿操作：恢复原有权限

步骤4：注销用户

- 将用户状态更新为"已注销"
- 清除用户会话和缓存
- 补偿操作：恢复用户状态为"正常"

步骤5：注销客户

- 将客户状态更新为"已注销"
- 更新注销时间和原因
- 补偿操作：恢复客户状态为"正常"

4.3 Saga实现关键点

补偿操作设计原则：

- 补偿操作必须能成功执行（设计合理的补偿逻辑）
- 补偿操作必须幂等（可能重复执行）
- 补偿操作尽量避免对外部系统的依赖
- 补偿失败需要人工介入处理

状态机管理：

- Saga维护整体状态：待执行、执行中、已完成、补偿中、已补偿、失败
- 每个步骤维护独立状态：待执行、执行中、已完成、补偿中、已补偿
- 使用状态机确保状态转换的正确性

超时和恢复：

- 设置每个步骤的超时时间（如5分钟）
- 超时后自动触发补偿流程
- 定时任务（每5分钟）扫描超时的Saga事务
- 自动恢复或人工介入

日志和监控：

- 记录Saga整体日志和每个步骤日志
- 包含开始时间、结束时间、状态、输入输出、错误信息
- 提供可视化界面查询Saga执行情况
- 失败时发送告警通知

4.4 Saga性能优化

状态缓存：

- Saga状态存储在Redis，减少数据库查询
- 执行完成后异步持久化到数据库
- 提高查询和更新性能

异步执行：

- 非关键步骤可以异步执行
- 使用消息队列实现异步调用
- 减少整体事务耗时

分片存储：

- Saga日志表分64个分片
- 根据Saga ID哈希分片
- 单表数据量控制在100万以内

五、本地消息表模式详细设计

5.1 本地消息表原理

核心思想： 利用数据库本地事务保证业务操作和消息发送的原子性。

实现步骤：

1. 业务操作和消息保存在同一个数据库事务中
2. 提交事务后，业务数据和消息都已持久化
3. 定时任务扫描消息表，将消息发送到消息队列
4. 消费者消费消息，执行后续业务操作
5. 发送成功后更新消息状态为"已发送"

优点：

- 实现简单，不需要额外的事务协调器

- 可靠性高，消息一定会发送（重试机制）
- 性能好，不阻塞主业务流程

缺点：

- 有延迟（秒级），不适合实时性要求高的场景
- 需要定时任务扫描消息表
- 消息表可能积累大量数据，需要定期清理

5.2 跨中心数据同步场景

业务需求： 客户在A中心更新了个人信息，需要同步到B中心和C中心。

实现流程：

步骤1：本地更新和消息保存（同一事务）

- A中心更新客户信息
- 在本地消息表中插入两条消息：
  - 消息1：同步到B中心，状态"待发送"
  - 消息2：同步到C中心，状态"待发送"
- 提交数据库事务

步骤2：定时任务扫描发送

- 定时任务每5秒扫描一次消息表
- 查询状态为"待发送"且创建时间>当前时间的消息
- 批量发送到消息队列（1000条/批）
- 发送成功后更新消息状态为"已发送"

步骤3：消费者处理

- B中心和C中心的消费者监听消息队列
- 收到消息后更新本地客户信息
- 处理成功后向A中心发送确认消息

步骤4：清理历史消息

- 定时任务每天凌晨2点清理历史消息
- 删除30天前且状态为"已发送"的消息

5.3 本地消息表实现关键点

幂等性保证：

- 每条消息有唯一的消息ID
- 消费者记录已处理的消息ID
- 重复消息直接忽略

重试机制：

- 发送失败后自动重试，最多3次

- 使用指数退避策略（1秒、2秒、4秒）
- 3次失败后标记为"失败"，人工处理

批量处理：

- 定时任务每次批量查询1000条消息
- 使用Pipeline批量发送到消息队列
- 减少网络开销，提高吞吐量

分片存储：

- 本地消息表分64个分片
- 根据消息ID哈希分片
- 支持百万级消息/天的吞吐量

六、分布式事务监控和恢复

6.1 事务监控

监控指标：

- **事务总量：** 每天处理的事务数量
- **事务成功率：** 成功/总量的比例
- **事务平均耗时：** 从开始到结束的平均时间
- **补偿执行次数：** 需要补偿的事务数量
- **超时事务数量：** 超过预设时间未完成的事务
- **失败事务数量：** 补偿也失败的事务

监控实现：

- 使用SkyWalking追踪事务调用链路
- 使用Prometheus采集事务指标
- 使用Grafana可视化展示
- 关键指标异常时发送告警

6.2 超时事务恢复

定时恢复任务：

- 每5分钟执行一次
- 查询状态为"执行中"且开始时间超过30分钟的事务
- 判断每个步骤的执行状态
- 自动触发补偿或重试

恢复策略：

- **TCC事务：** 超时后自动执行Cancel回滚
- **Saga事务：** 超时后自动执行补偿流程
- **本地消息表：** 超时消息自动重试发送

6.3 失败事务处理

告警通知：

- 补偿失败的事务立即发送告警
- 包含事务ID、业务类型、失败原因、影响范围
- 通过企业微信、短信、邮件多渠道通知

人工介入：

- 提供管理后台查询失败事务
- 显示详细的执行日志和错误信息
- 支持手动重试或手动补偿
- 记录处理结果和处理人

6.4 日志清理

清理策略：

- 每天凌晨2点执行清理任务
- 删除30天前且状态为"已完成"的事务日志
- 保留失败事务日志3个月（用于问题分析）
- 超过3个月的日志归档到历史库

七、事务性能优化

7.1 减少网络调用

问题： 分布式事务涉及多次网络调用，延迟累加

优化方案：

- 合并多次调用为一次批量调用
- 使用gRPC代替HTTP，减少序列化开销
- 启用HTTP/2多路复用
- 服务部署在同一机房，减少网络延迟

7.2 并行执行

问题： 多个步骤串行执行，耗时长

优化方案：

- TCC的Confirm/Cancel阶段并行执行
- Saga的无依赖步骤并行执行
- 使用线程池或异步框架实现并行

7.3 异步化

问题： 同步等待所有步骤完成，阻塞主流程

优化方案：

- 非关键步骤改为异步执行
- 使用消息队列解耦
- 主流程立即返回，后台异步处理

7.4 缓存优化

问题： 频繁查询事务状态，数据库压力大

优化方案：

- 事务状态缓存到Redis
- 执行完成后异步持久化到数据库
- 减少数据库查询，提高性能

八、容量规划

8.1 事务处理能力

日均300万业务办理，需要的分布式事务能力：

事务类型	占比	日处理量	平均耗时	峰值TPS
TCC（账户充值/支付）	25%	75万笔	100-500ms	50
Saga（客户注销/合约签订）	25%	75万笔	5-10秒	25
本地消息表（日志/数据同步）	50%	150万笔	<10ms	70
合计	100%	**300万笔**	**-**	**145**

峰值计算：

- 午后高峰占比35%，持续2小时
- 峰值事务量：300万 × 35% / 7200秒 ≈ 146 TPS
- 考虑2倍冗余，需要支持300 TPS

8.2 事务协调器集群

- **TCC协调器：** 4个节点（8核16G），集群总吞吐200 TPS
- **Saga协调器：** 3个节点（8核16G），集群总吞吐120 TPS

8.3 事务日志存储

日志类型	分片数	单分片日写入	保留时间
TCC日志表	32	~2.3万条	30天
Saga日志表	64	~1.2万条	30天
本地消息表	64	~2.3万条	7天

九、最佳实践建议

9.1 选型建议

1. **优先考虑业务特点：** 根据一致性要求、实时性要求、事务长度选择方案
2. **不要过度设计：** 简单业务可以用本地消息表，不一定要用TCC或Saga
3. **混合使用：** 一个系统可以同时使用多种方案，针对不同场景选择最合适的

9.2 设计建议

1. **设计好补偿逻辑：** 补偿操作必须能成功，失败要有兜底方案
2. **保证幂等性：** 所有操作都要支持幂等，避免重复执行导致数据错误
3. **详细日志记录：** 记录每个步骤的输入输出，方便问题排查
4. **监控和告警：** 及时发现问题，快速响应

9.3 实施建议

1. **分阶段实施：** 先在非核心业务试点，验证后再推广到核心业务
2. **充分测试：** 测试正常流程、异常流程、超时场景、并发场景
3. **灰度发布：** 小流量验证，逐步放量，确保稳定
4. **持续优化：** 根据线上反馈不断优化性能和可靠性

十、总结

本方案针对电信CRM客户中心日均300万业务量和4亿次服务调用的场景，设计了完整的分布式事务解决方案。

核心方案：

- **TCC模式：** 用于账户充值、支付等强一致性业务，支持100万笔/天
- **Saga模式：** 用于客户注销、合约签订等长流程业务，支持75万笔/天
- **本地消息表：** 用于日志记录、数据同步等最终一致性业务，支持1000万笔/天

关键设计：

- 幂等性保证：防止重复执行
- 超时恢复机制：自动恢复超时事务
- 并行执行优化：提升性能2-3倍
- 分片存储：支持海量事务日志

性能指标：

- 总吞吐：300 TPS（峰值）
- TCC平均耗时：100-500ms
- Saga平均耗时：5-10秒
- 事务成功率：>99.9%

通过合理的方案选型、精心的设计实现、完善的监控恢复机制，可以在保证数据一致性的同时，支撑超大规模的业务量。