



分布式事务

传智播客.黑马程序员



1.理解分布式事务

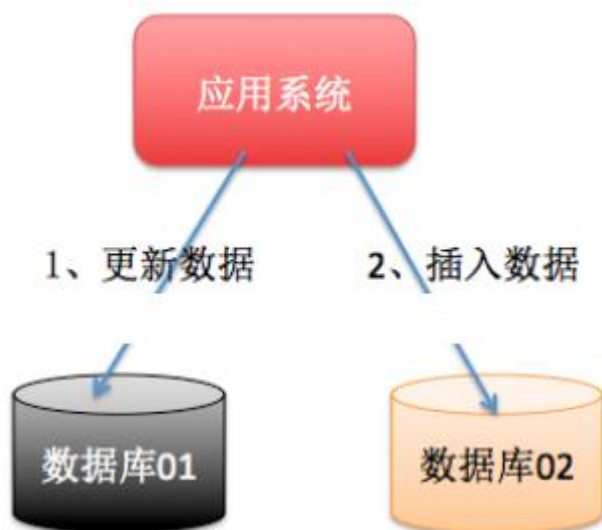
1.1 什么是分布式事务

分布式事务就是指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上。简单的说，就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。本质上来说，分布式事务就是为了保证不同数据库的数据一致性。

1.2 分布式事务产生的原因

1.2.1 数据库分库分表

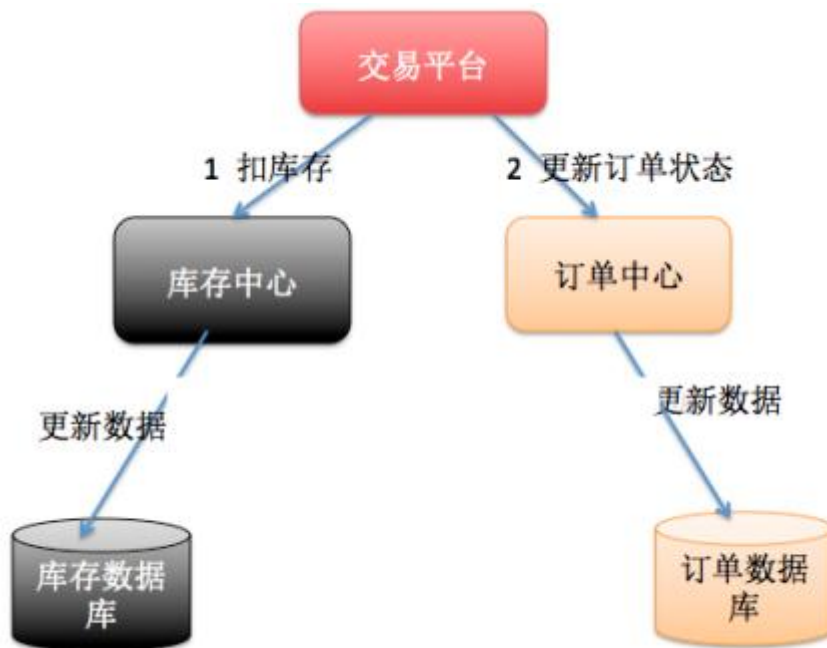
当数据库单表一年产生的数据超过 1000W，那么就要考虑分库分表，具体分库分表的原理在此不做解释，以后有空详细说，简单的说就是原来的一个数据库变成了多个数据库。这时候，如果一个操作既访问 01 库，又访问 02 库，而且要保证数据的一致性，那么就要用到分布式事务。



1.2.2 应用 SOA 化

所谓的 SOA 化，就是业务的服务化。比如原来单机支撑了整个电商网站，现在对整个网站进行拆解，分离出了订单中心、用户中心、库存中心。对于订单中心，有专门的数据库存储订单信息，用户中心也有专门的数据库存储用户信息，库存中心也会有专门的数据库存

储库存信息。这时候如果要同时对订单和库存进行操作，那么就会涉及到订单数据库和库存数据库，为了保证数据一致性，就需要用到分布式事务。



1.3 事务的 ACID 特性

1.3.1 原子性 (A)

所谓的原子性就是说，在整个事务中的所有操作，要么全部完成，要么全部不做，没有中间状态。对于事务在执行中发生错误，所有的操作都会被回滚，整个事务就像从没被执行过一样。

1.3.2 一致性 (C)

事务的执行必须保证系统的一致性，就拿转账为例，A 有 500 元，B 有 300 元，如果在一个事务里 A 成功转给 B 50 元，那么不管并发多少，不管发生什么，只要事务执行成功了，那么最后 A 账户一定是 450 元，B 账户一定是 350 元。

1.3.3 隔离性 (I)

所谓的隔离性就是说，事务与事务之间不会互相影响，一个事务的中间状态不会被其他事务感知。

1.3.4 持久性（D）

所谓的持久性，就是说一单事务完成了，那么事务对数据所做的变更就完全保存在了数据库中，即使发生停电，系统宕机也是如此。

1.4 电商系统分布式事务应用场景

1.4.1 支付

最经典的场景就是支付了，一笔支付，是对买家账户进行扣款，同时对卖家账户进行加钱，这些操作必须在一个事务里执行，要么全部成功，要么全部失败。而对于买家账户属于买家中心，对应的是买家数据库，而卖家账户属于卖家中心，对应的是卖家数据库，对不同数据库的操作必然需要引入分布式事务。

1.4.2 下单

买家在电商平台下单，往往会涉及到两个动作，一个是扣库存，第二个是更新订单状态，库存和订单一般属于不同的数据库，需要使用分布式事务保证数据一致性。

2. 常见分布式事务解决方案

2.1 基于 XA 的两阶段提交

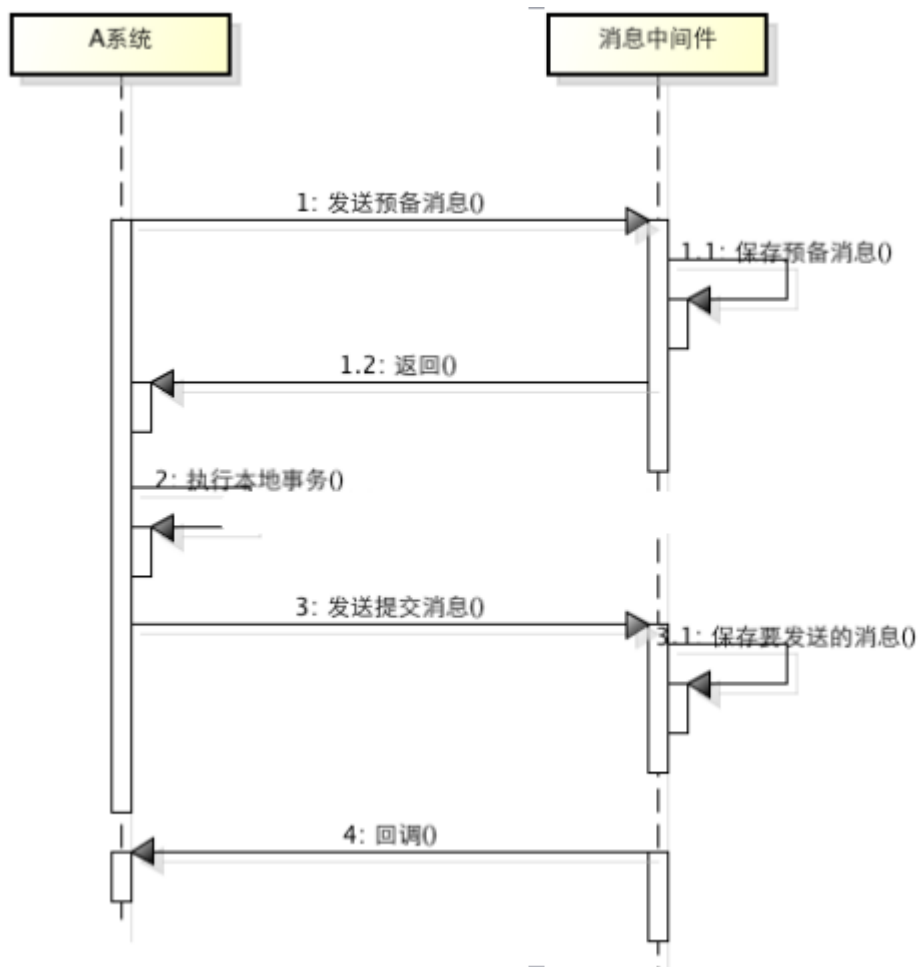
XA 是一个分布式事务协议，由 Tuxedo 提出。XA 中大致分为两部分：事务管理器和本地资源管理器。其中本地资源管理器往往由数据库实现，比如 Oracle、DB2 这些商业数据库都实现了 XA 接口，而事务管理器作为全局的调度者，负责各个本地资源的提交和回滚。XA 实现分布式事务的原理如下：



总的来说，XA 协议比较简单，而且一旦商业数据库实现了 XA 协议，使用分布式事务的成本也比较低。但是，XA 也有致命的缺点，那就是性能不理想，特别是在交易下单链路，往往并发量很高，XA 无法满足高并发场景。XA 目前在商业数据库支持的比较理想，在 mysql 数据库中支持的不太理想，mysql 的 XA 实现，没有记录 prepare 阶段日志，主备切换会导致主库与备库数据不一致。许多 nosql 也没有支持 XA，这让 XA 的应用场景变得非常狭隘。

2.2 消息事务+最终一致性

所谓的消息事务就是基于消息中间件的两阶段提交，本质上是对消息中间件的一种特殊利用，它是将本地事务和发消息放在了一个分布式事务里，保证要么本地操作成功成功并且对外发消息成功，要么两者都失败，开源的 RocketMQ 就支持这一特性，具体原理如下：



- 1、A 系统向消息中间件发送一条预备消息
- 2、消息中间件保存预备消息并返回成功
- 3、A 执行本地事务
- 4、A 发送提交消息给消息中间件

通过以上 4 步完成了一个消息事务。对于以上的 4 个步骤，每个步骤都可能产生错误，下面一一分析：

步骤一出错，则整个事务失败，不会执行 A 的本地操作

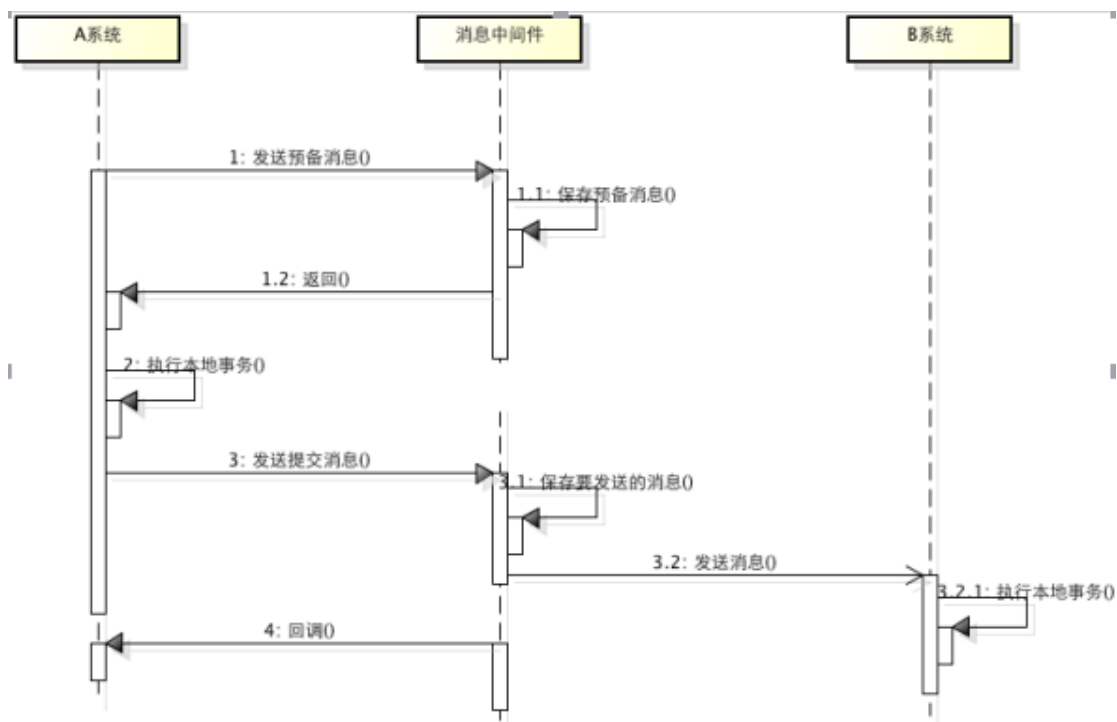
步骤二出错，则整个事务失败，不会执行 A 的本地操作

步骤三出错，这时候需要回滚预备消息，怎么回滚？答案是 A 系统实现一个消息中间件的回调接口，消息中间件会去不断执行回调接口，检查 A 事务执行是否执行成功，如果失败则回滚预备消息

步骤四出错，这时候 A 的本地事务是成功的，那么消息中间件要回滚 A 吗？答案是不需要，其实通过回调接口，消息中间件能够检查到 A 执行成功了，这时候其实不需要 A 发提交消息了，消息中间件可以自己消息进行提交，从而完成整个消息事务



基于消息中间件的两阶段提交往往用在高并发场景下，将一个分布式事务拆成一个消息事务（A 系统的本地操作+发消息）+B 系统的本地操作，其中 B 系统的操作由消息驱动，只要消息事务成功，那么 A 操作一定成功，消息也一定发出来了，这时候 B 会收到消息去执行本地操作，如果本地操作失败，消息会重投，直到 B 操作成功，这样就变相地实现了 A 与 B 的分布式事务。原理如下：



虽然上面的方案能够完成 A 和 B 的操作，但是 A 和 B 并不是严格一致的，而是最终一致的，我们在这里牺牲了一致性，换来了性能的大幅度提升。当然，这种玩法也是有风险的，如果 B 一直执行不成功，那么一致性会被破坏，具体要不要玩，还是得看业务能够承担多少风险。

另外此方案严重依赖消息中间件的高可用性，一旦消息中间件挂了整个系统就挂了。洪峰到来时，所有的系统都依赖消息中间件的话，消息中间件很容易就挂了。

2.3 TCC 编程模式

所谓的 TCC 编程模式，也是两阶段提交的一个变种。TCC 提供了一个编程框架，将整个业务逻辑分为三块：Try、Confirm 和 Cancel 三个操作。以在线下单为例，Try 阶段会去扣库存，Confirm 阶段则是去更新订单状态，如果更新订单失败，则进入 Cancel 阶段，会去恢复库存。总之，TCC 就是通过代码人为实现了两阶段提交，不同的业务场景所写的代码都不一样，复杂度也不一样，我们今天着重讲解 TCC 编程模式的代码实现。



3.理解 TCC 编程模式

3.1 什么是 TCC

TCC 是一种补偿性分布式事务解决方案，最初由支付宝提出。TCC 是三个英文单词的首字母缩写,分别对应 Try、Confirm 和 Cancel 三种操作，这三种操作的业务含义如下：

Try：预留业务资源

Confirm：确认执行业务操作

Cancel：取消执行业务操作

稍稍对照下关系型数据库事务的三种操作：DML、Commit 和 Rollback，会发现和 TCC 有异曲同工之妙。在一个跨应用的业务操作中，Try 操作是先把多个应用中的业务资源预留和锁定住，为后续的确认打下基础，类似的，DML 操作要锁定数据库记录行，持有数据库资源；Confirm 操作是在 Try 操作中涉及的所有应用均成功之后进行确认，使用预留的业务资源，和 Commit 类似；而 Cancel 则是当 Try 操作中涉及的所有应用没有全部成功，需要将已成功的应用进行取消(即 Rollback 回滚)。其中 Confirm 和 Cancel 操作是一对反向业务操作。

简而言之，TCC 是应用层的 2PC(2 Phase Commit, 两阶段提交)，如果你将应用看做资源管理器的话。详细来说，TCC 每项操作需要做的事情如下：

1、Try：尝试执行业务。

完成所有业务检查(一致性)

预留必须业务资源(准隔离性)

2、Confirm：确认执行业务。

真正执行业务

不做任何业务检查

只使用 Try 阶段预留的业务资源

3、Cancel：取消执行业务

释放 Try 阶段预留的业务资源

3.2 业务模拟分析

TCC 的理论有点让人费解。故接下来将以账务拆分为例，对 TCC 事务的流程做一个描述，希望对理解 TCC 有所帮助。账务拆分的业务场景如下，分别位于三个不同分库的帐户 A、B、C，A 和 B 一起向 C 转帐共 80 元：

1、Try：尝试执行业务。

完成所有业务检查(一致性)：检查 A、B、C 的帐户状态是否正常，帐户 A 的余额是否不少于 30 元，帐户 B 的余额是否不少于 50 元。

预留必须业务资源(准隔离性)：帐户 A 的冻结金额增加 30 元，帐户 B 的冻结金额增加 50 元，这样就保证不会出现其他并发进程扣减了这两个帐户的余额而导致在后续的真正转帐操作过程中，帐户 A 和 B 的可用余额不够的情况。

2、Confirm：确认执行业务。

真正执行业务：如果 Try 阶段帐户 A、B、C 状态正常，且帐户 A、B 余额够用，则执行帐户 A 给帐户 C 转账 30 元、帐户 B 给帐户 C 转账 50 元的转帐操作。

不做任何业务检查：这时已经不需要做业务检查，Try 阶段已经完成了业务检查。

只使用 Try 阶段预留的业务资源：只需要使用 Try 阶段帐户 A 和帐户 B 冻结的金额即可。

3、Cancel：取消执行业务

释放 Try 阶段预留的业务资源：如果 Try 阶段部分成功，比如帐户 A 的余额够用，且冻结相应金额成功，帐户 B 的余额不够而冻结失败，则需要对帐户 A 做 Cancel 操作，将帐户 A 被冻结的金额解冻掉。

3.3 什么是幂等性

幂等性就是用户对于同一操作发起的一次请求或者多次请求的结果是一致的，不会因为多次点击而产生了副作用。



4.TCC 开源框架-tcc-transaction

4.1 框架简介

tcc-transaction 是开源的 TCC 补偿性分布式事务框架

Git 地址: <https://github.com/changmingxie/tcc-transaction>

4.2 安装 tcc-transaction 到本地仓库

我们目前使用的版本是 1.2，引入 tcc-transaction 工程。

安装 tcc-transaction-api tcc-transaction-core 和 tcc-transaction-dubbo 到本地仓库

4.3 分布式项目使用 tcc-transaction 框架

4.3.1 配置 tcc-transaction

(1) pom.xml 中引入依赖

```
<dependency>

    <groupId>org.mengyun</groupId>

    <artifactId>tcc-transaction-dubbo-capital-api</artifactId>

    <version>1.2.4.14</version>

</dependency>

<dependency>

    <groupId>org.mengyun</groupId>

    <artifactId>tcc-transaction-dubbo</artifactId>

    <version>1.2.4.14</version>
```



```
</dependency>

<dependency>

    <groupId>org.mengyun</groupId>

    <artifactId>tcc-transaction-spring</artifactId>

    <version>1.2.4.14</version>

</dependency>
```

(2) web.xml 上下文加载 tcc-transaction

```
<context-param>

    <param-name>contextConfigLocation</param-name>

    <param-value>classpath*:config/spring/local/appcontext-*.xml,classpath:tcc-transact
ion.xml,classpath:tcc-transaction-dubbo.xml

    </param-value>

</context-param>
```

4.3.2 发布服务

发布一个 Tcc 服务方法，可被远程调用并参与到 Tcc 事务中，四个约束：

- (1) 在服务提供方的实现方法上加上 @Compensable 注解，并设置注解的属性
- (2) 在服务提供方的接口方法上加上 @Compensable 注解
- (3) 服务方法的入参能被序列化(默认使用 jdk 序列化机制，需要参数实现 Serializable 接口，可以设置 repository 的 serializer 属性自定义序列化实现)
- (4) try 方法、confirm 方法和 cancel 方法入参类型须一样

Compensable 的属性包括 propagation、confirmMethod、cancelMethod、transactionContextEditor。propagation 可不用设置，框架使用缺省值；设置 confirmMethod 指定 CONFIRM 阶段的调用方法；设置 cancelMethod 指定 CANCEL 阶段的调用方法；设置 transactionContextEditor 为 DubboTransactionContextEditor.class。



发布 Tcc 服务示例：

try 接口方法：

```
@Compensable

public String record(CapitalTradeOrderDto tradeOrderDto);
```

try 实现方法：

```
@Compensable(confirmMethod = "confirmRecord", cancelMethod = "cancelRecord",
transactionContextEditor = DubboTransactionContextEditor.class)

public String record(CapitalTradeOrderDto tradeOrderDto) {
```

confirm 方法：

```
public void confirmRecord(CapitalTradeOrderDto tradeOrderDto) {
```

cancel 方法：

```
public void cancelRecord(CapitalTradeOrderDto tradeOrderDto) {
```

4.3.3 调用服务

调用远程 Tcc 服务，将远程 Tcc 服务参与到本地 Tcc 事务中，本地的服务方法也需要声明为 Tcc 服务，有三个约束：

- (1) 在服务方法上加上@Compensable 注解,并设置注解属性
- (2) 服务方法的入参都须能序列化(实现 Serializable 接口)
- (3) try 方法、confirm 方法和 cancel 方法入参类型须一样

调用 Tcc 服务示例：

try 方法：

```
@Compensable(confirmMethod = "confirmMakePayment", cancelMethod =
"cancelMakePayment")

public void makePayment(Order order, BigDecimal redPacketPayAmount, BigDecimal
```



```
capitalPayAmount) {

    System.out.println("order    try    make    payment    called.time    seq:" +
    DateFormatUtils.format(Calendar.getInstance(), "yyyy-MM-dd HH:mm:ss"));

    order.pay(redPacketPayAmount, capitalPayAmount);

    orderRepository.updateOrder(order);

    String result = capitalTradeOrderService.record(buildCapitalTradeOrderDto(order));

    String result2 = redPacketTradeOrderService.record(buildRedPacketTradeOrderDto(order));

}
```

confirm 方法:

```
public void confirmMakePayment(Order order, BigDecimal redPacketPayAmount, BigDecimal
capitalPayAmount) {
```





cancel 方法:

```
public void cancelMakePayment(Order order, BigDecimal redPacketPayAmount, BigDecimal
capitalPayAmount) {
```

4.4 示例工程安装部署

4.4.1 数据库脚本

执行提供的建库建表脚本 （资源文件夹）

 create_db_cap.sql	2016/11/28 15:42	SQL 文件	1 KB
 create_db_ord.sql	2016/11/28 15:42	SQL 文件	2 KB
 create_db_red.sql	2016/11/28 15:42	SQL 文件	1 KB
 create_db_tcc.sql	2016/11/28 15:42	SQL 文件	3 KB

4.4.2 工程说明

tcc-transaction-dubbo-capital 资产服务

tcc-transaction-dubbo-capital-api 资产服务接口



tcc-transaction-dubbo-redpacket 红包服务

tcc-transaction-dubbo-redpacket-api 红包服务接口

tcc-transaction-dubbo-order 订单 WEB 工程（TCC 调用方）

4.4.3 修改 zookeeper 地址与数据库连接地址

修改 sample-dubbo-xxxxx.properties 和 jdbc.properties