SOFAStack

分布式事务
技术白皮书

产品版本：AntStack Plus 1.13.1

文档版本：20231214

蚂蚁集团
ANT GROUP

法律声明

蚂蚁集团版权所有©2022，并保留一切权利。

未经蚂蚁集团事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

商标声明

蚂蚁集团 ANT GROUP及其他蚂蚁集团相关的商标均为蚂蚁集团所有。本文档涉及的第三方的注册商标，依法由权利人所有。

免责声明

由于产品版本升级、调整或其他原因，本文档内容有可能变更。蚂蚁集团保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在蚂蚁集团授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过蚂蚁集团授权渠道下载、获取最新版的用户文档。如因文档使用不当造成的直接或间接损失，本公司不承担任何责任。

通用约定

格式	说明	样例
⚠ 危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	⚠ 危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	**单击设置> 网络> 设置网络类型。**
粗体	表示按键、菜单、页面名称等UI元素。	**在结果确认页面，单击确定。**
`Courier字体`	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.什么是分布式事务

分布式事务（DistributedTransaction-eXtended，简称 DTX）是蚂蚁集团自主研发的金融级分布式事务中间件，用来保障分布式架构下跨数据、跨服务的数据一致性问题。

分布式事务伴随着业务的多样性发展演进出了多种接入模式，多种模式分布式适用于各类业务场景，帮用户解决各种业务场景下的业务数据一致性问题。主要接入模式如下：

- TCC 模式

  TCC（Try-Confirm-Cancel）是一种高性能的分布式事务接入方案，该模式提供了更多的灵活性，几乎可满足任何您能想到的事务场景。TCC 模式提供自定义补偿型事务、自定义资源预留型事务、消息事务等场景，用户可以介入两阶段提交的过程，以达到特殊场景下的自定义优化及特殊功能的实现。

- FMT 模式

  为了解决 TCC 模式的易用性问题，分布式事务推出了框架管理事务模式（Framework-managed transactions，简称 FMT）。FMT 是一种无侵入的分布式事务解决方案，该模式解决了分布式事务的易用性问题，最大的特点是易于使用、快速接入以及对业务代码无侵入。

- Saga 模式

  基于 Hector & Kenneth 发表论文 Sagas（1987）理论的的长事务解决方案，在 Saga 模式中，业务流程中每个参与者都提交本地事务，当出现某一个参与者失败，则补偿前面已经成功的参与者。此模式适用于业务流程长、业务流程多、参与者包含其他公司或老系统服务等场景。其优势包括：一阶段提交本地事务无锁，事件驱动架构，参与者可异步执行，高并发高吞吐；补偿服务易于实现或老系统本身就有补偿（冲正）服务；支持服务编排、有可视化的设计器和执行轨迹监控。

- XA 模式

  高性能的标准 XA 事务方案，基于标准 XA 实现，消除与 JEE 开发相关的日益复杂的问题，帮助传统企业的业务无缝上云以及服务化拆分。并可以与蚂蚁集团自研数据库 OceanBase 共同打造实时数据一致性的整体解决方案。

2.产品优势

金融场景的全面覆盖

- **支付与转账**

  金融行业常见的支付、转账、账务等业务场景对于吞吐量有很高的要求。SOFAStack 的分布式事务产品在各类大促中的优异表现证明了性能不会成为瓶颈。

- **财富理财**

  这类场景中往往涉及的金额较大，所以对于产品的稳定性要求非常高，SOFAStack 的分布式事务产品拥有金融级的品质，可为业务的持续性与稳定性保驾护航。

- **保险与监管报送**

  参与方多、业务复杂度高是该类业务的典型特征，SOFAStack 的分布式事务产品历经十多年的演进历程，足以灵活应对各种场景，满足事务一致性要求，保证与各类业务完美结合。

政务领域支付更便捷

- **生活缴费**

  作为支付、转账场景的延伸，生活缴费在政务系统中不可或缺，例如水电费，电话费，上网资费等，都通过手机 APP 或者电脑端进行处理缴费。政务系统需要对缴费信息进行一致性处理，SOFAStack 的分布式事务产品可以保证关联信息同步修改，跨系统信息及时同步。

- **跨地域信息即刻同步**

  当前各地域政府机关往往有自己的数据库，人员流动、企业信息备案，都最初在本地进行登记备案。信息变更频繁的信息化时代，仅通过手工方式进行信息变更后的同步，会带来脏读和脏写的问题，采用 SOFAStack 的分布式事务产品可以保证政务机关的信息高效同步，精准一致。

泛互联网多领域场景

- **订单、会员卡、成长值、积分**

  以积分商城为例，使用会员卡余额购买商品，会涉及到扣减账户余额（数据库）、增加账户积分数量（数据库）、会员卡成长值提升、历史订单增加等服务。目前使用对账的方式来应对此类场景的性能较低，涉及业务扩展或改变时改造成本高。使用 SOFAStack 的分布式事务产品进行简单的改造接入，即可完成数据的同步。

- **担保交易**

  以电商抢购支付场景为例，秒杀抢购并发量高，性能要求高。通常流程尝试扣除用户可用资金，转移预冻结资金，增加中间账户可用资金（担保交易不能立即把钱打给商户，需要有一个中间账户来暂存），七天后需要将资金从中间账户划拨给商户。SOFAStack 的分布式事务产品可以支持大规模的抢购场景，保证客户成功支付，等到低锋期时，再慢慢消化支付数据，异步的执行资金到账流程，并且最终保证资金能顺利转入商户的账户中。

金融级品质的保障

- **金融级容灾保障**

  提供同城以及异地等多种模式以及多种级别的容灾能力，以业界最高规格的标准来保障客户业务的连续性。

- **无与伦比的性能**

  相比传统二阶段模式，减少持有锁时间，大幅提升性能。特有的性能推进模式（Performance Bursting Mode）可以大幅提升吞吐量，曾在 2021 年双 11 中支撑 25.6 万笔/秒的支付操作。

- **使用简洁易于接入**

  蚂蚁集团多年沉淀的实操经验使产品具备了快速灵活的接入能力，易于使用与运维。

- **兼容性保障**

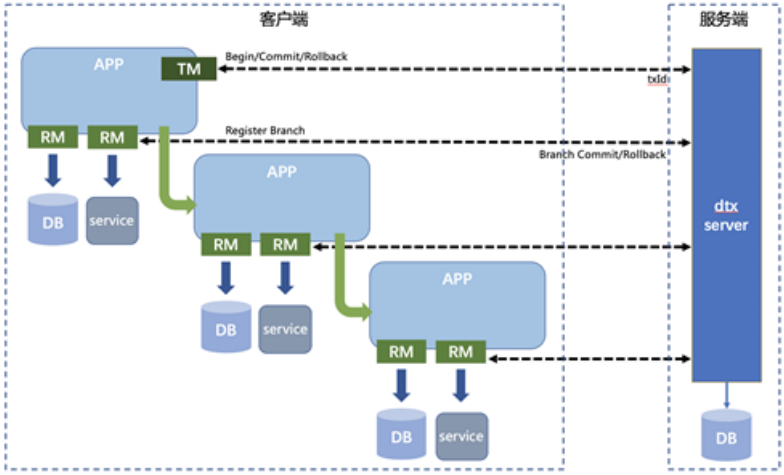
分布式事务是一个抽象的基于 Service 层的概念，与底层事务实现无关，也就是说在分布式事务的范围内，无论是关系型数据库 MySQL、Oracle，还是 KV 存储 MemCache，或是列存数据库 HBase，只要将对它们的操作包装成分布式事务的参与者，就可以接入到分布式事务中。

3.产品架构

整体架构

DTX 整体分为客户端和服务端两部分：

- 客户端作为 SDK ，与业务系统集成在一起。用户使用客户端提供的 API 来开启分布式事务、编排参与者、结束（提交或回滚）分布式事务。
- 服务端是一个独立部署的服务，主要负责事务日志的存储和异常事务的恢复。



产品模块

DTX 的客户端包含TM 和 RM 两个模块；服务端包含 dtxserver 模块，是独立部署服务。

- TM

  TM 是事务管理器，负责开启分布式事务、编排事务参与者、提交或回滚分布式事务。TM 开启分布式事务时，会去服务端创建一条主事务记录；结束分布式事务时，会通知服务端分布式事务提交或者回滚。

- RM

  RM 是资源管理器，负责管理事务资源。管理的资源可以是一个服务，也可以是一个数据库。RM 会在启动时，向服务端注册资源；在运行期负责创建资源的分支事务日志，并驱动资源二阶段方法的执行

- dtxserver

  dtxserver 是分布式事务的服务端，有自己的数据库。负责分布式事务日志的存储，以及异常事务的补偿服务。在分布式事务开启时，TM 会通知服务端创建一条主事务日志，并生成 txId ，一个分布式事务只有一个唯一的 txId。分布式事务执行过程中，RM 会向服务端汇报资源的执行状态，此时服务端会创建一条分支事务记录；在分布式事务结束时，TM 会通知服务端提交或者回滚分布式事务，此时服务端会修改事务的状态为待提交或者待回滚，并通知所有资源去执行二阶段的提交或者回滚操作。

执行流程

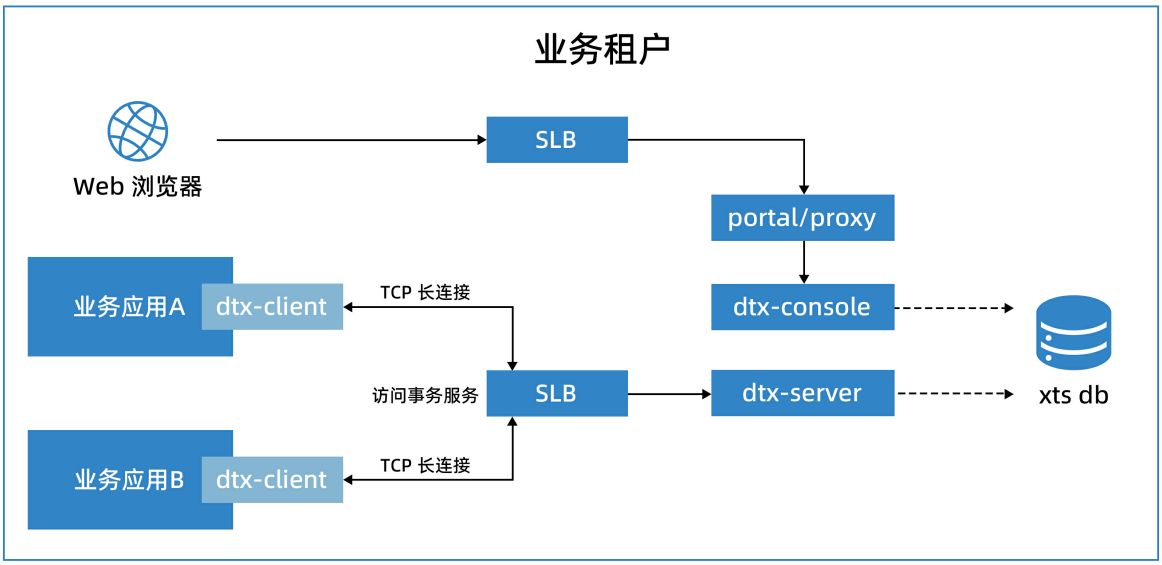
分布式事务的执行流程如下：

1. TM 开启分布式事务，向服务端注册主事务记录，并得到分布式事务的 txId。
2. 开启分布式事务之后，用户按业务场景编排数据库、服务等事务资源，每一个资源的准备操作执行情况，RM 都会汇报给服务端，服务端则会创建相应分支事务日志。
3. TM 结束分布式事务，通知服务端分布式事务提交或者回滚。此时服务端会修改事务的状态为待提交或者待回滚，分布式事务一阶段结束。
4. 分布式事务二阶段开始，服务端根据事务日志信息，决定分布式事务是提交还是回滚。
5. 服务端根据流程 4 的判断，通知所有 RM 提交或者回滚资源，分布式事务二阶段结束。

> **重要**
>
> Saga 模式无二阶段提交。

网络架构

DTX 专有云部署架构如下图：



- 客户端（dtx-client）作为 SDK，集成至业务应用中。
- 服务端（dtx-server）和 DTX 控制台（dtx-console）分别是独立的应用部署在业务租户中，两者共有数据库。
- dtx-client 和 dtx-server 直接通过 TCP 长连接进行通信，用户通过浏览器访问 DTX 控制台。

4.性能压测报告

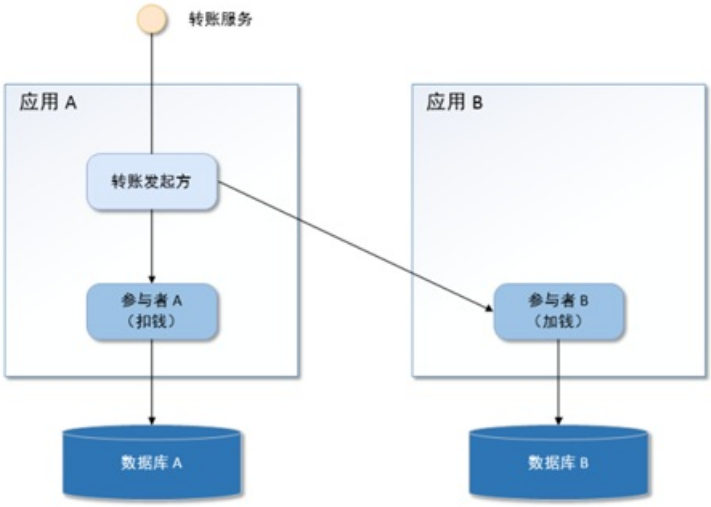
4.1. 压测模型介绍

不同业务场景，不同机器配置，性能不一样。详细性能数据，可以参考以下性能压测报告。

本文模拟了转账的业务场景，分别用分布式事务的 TCC 模式、Saga 模式与 FMT 模式实现了转账功能，并对 TCC 模式、Saga 模式、FMT 模式进行了压测。通过压测报告，您可对 TCC、Saga、FMT 模式的性能有一个大致了解。

压测模型介绍

如下图所示，“应用 A”对外发布“转账服务”，服务完成“账号 A”向“账号 B”转账的功能。



1. “转账服务”内部开启分布式事务；
2. “参与者 A”负责从“账号 A”扣款，“参与者 A”服务由“应用 A”提供（与发起方为同一应用）；
3. “参与者 B”负责向“账号 B”加钱，“参与者 B”服务由“应用B”提供。

> **说明**
>
> A 的账号数据存储在“数据库 A”中，此数据库仅由“应用 A”访问；B 的账号数据存储“数据库 B”中，此数据库由“应用 B”访问。

4.2. 实现转账

TCC 和 FMT 同库压测模型

转账表结构设计

数据库表 A 结构为：

```
## 账户余额表，存储账户余额信息
create table account(
account_no varchar(256) not null comment '账户',
amount DOUBLE comment '账户余额',
freezed_amount DOUBLE comment '账户冻结金额，TCC才会用到的字段',
primary key (account_no)
);
## 账号操作流水表，记录每一笔分布式事务操作的账号、操作金额、操作类型（扣钱/加钱），TCC 模式下才会使用到此表。
create table account_transaction(
tx_id varchar(256) not null,
account_no varchar(256) not null,
amount DOUBLE,
type varchar(256) not null,
primary key (tx_id)
);
```

数据库表 B 结构与数据库表 A 结构相同。

实现 TCC 模式

发起方实现如下：

```
@DtxTransaction(bizType="single-transfer-by-tcc")
public boolean transferByTcc(String from, String to, double amount) {
  try{
    //第一个参与者
    boolean ret = firstTccActionRef.prepare_minus(null, from, amount);
    if(!ret){
      //事务回滚
     RuntimeContext.setRollBack();
      return false;
      }
    //第二个参与者
    ret = secondTccActionRef.prepare_add(null, to, amount);
    if(!ret){
      //事务回滚
      RuntimeContext.setRollBack();
                  return false;
        }
            return ret;
    }catch(Throwable t){
    throw new RuntimeException(t);
    }
  }
```

参与者 A（扣钱）实现：

```
public interface FirstTccAction {
@TwoPhaseBusinessAction(name = "firstTccAction", commitMethod = "commit",
rollbackMethod = "rollback")
public boolean prepare_minus(BusinessActionContext businessActionContext,String account
No,double amount);
public boolean commit(BusinessActionContext businessActionContext);
public boolean rollback(BusinessActionContext businessActionContext);
}
```

参与者 B（加钱）实现：

```
public interface SecondTccAction {
  @TwoPhaseBusinessAction(name = "secondTccAction", commitMethod = "commit",
rollbackMethod = "rollback")
  public boolean prepare_add(BusinessActionContext businessActionContext,String account
No, double amount);
  public boolean commit(BusinessActionContext businessActionContext);
  public boolean rollback(BusinessActionContext businessActionContext);
  }
```

实现 FMT 模式

发起方实现如下：

```
@DtxTransaction(bizType="transfer-by-auto")
public boolean transferByAuto(String from, String to, double amount) {
  boolean ret = false;
  try{
    //第一个参与者
    ret = firstAutoAction.amount_minus(from, amount);
    if(!ret){
      //事务回滚
      RuntimeContext.setRollBack();
      return  false;
      }
    //第二个参与者
    ret = secondAutoAction.amount_add(to, amount);
    if(!ret){
      //事务回滚
      RuntimeContext.setRollBack();
      return  false;
      }
    return ret;
    }catch(Throwable t){
    throw new RuntimeException(t);
    }
  }
```

参与者 A（扣钱）实现：

```
public class FirstAutoActionImpl implements FirstAutoAction {
  ... ...
    @Override
    public boolean amount_minus(final String accountNo, final double amount) {
    try{
      return tccFirstActionTransactionTemplateAuto.execute(new
TransactionCallback<Boolean>() {
        @Override
        public Boolean doInTransaction(TransactionStatus status) {
          try {
            Account account = firstAccountDAOAuto.getAccountForUpdate(accountNo);
            if(account == null){
              throw new RuntimeException("账号不存在");
              }
            //扣钱
            double newAmount = account.getAmount() - amount;
            if (amount < 0) {
              throw new RuntimeException("余额不足");
              }
            account.setAmount(newAmount);
            int n = firstAccountDAOAuto.updateAmount(account);
            if(n == 1){
              return true;
              }else{
              status.setRollbackOnly();
              return false;
              }
            } catch (Exception e) {
            logger.error("amount_minus error", e);
            status.setRollbackOnly();
            return false;
            }
          }
        });
      }catch(Exception e){
      logger.error("amount_minus failed", e);
      return false;
      }
    }
  }
```

参与者 B（加钱）实现：

```
public class SecodeAutoActionImpl implements SecondAutoAction {
  ... ...
    @Override
    public boolean amount_add(final String accountNo, final double amount) {
    try{
      return tccSecondActionTransactionTemplateAuto.execute(new
TransactionCallback<Boolean>() {
        @Override
        public Boolean doInTransaction(TransactionStatus status) {
          try {
            Account account = secondAccountDAOAuto.getAccountForUpdate(accountNo);
            if(account == null){
              throw new RuntimeException("账号不存在");
              }
            //加钱
            double newAmount = account.getAmount() + amount;
            account.setAmount(newAmount);
            secondAccountDAOAuto.updateAmount(account);
            } catch (Exception e) {
            logger.error("amount_add error", e);
            status.setRollbackOnly();
            return false;
            }
          return true;
          }
       });
      } catch (Exception e) {
      logger.error("amount_add failed", e);
                  return false;
        }
    }
  }
```

SAGA 同库压测模型

转账表结构设计

数据库表 A 结构为：

```
## 账户余额表，存储账户余额信息
create table if not exists account(
account_no varchar(64) not null ,
amount DECIMAL ,
primary key (account_no)
);

## 账号操作流水表，记录每一笔分布式事务操作的账号、操作金额、操作类型（扣钱/加钱）、流水状态（成功|已经回滚）
create table if not exists  account_transaction(
business_key varchar(128) not null,
account_no varchar(256) not null,
amount DECIMAL,
type varchar(32) not null,
status varchar(32) not null,
primary key (business_key)
);
```

数据库表 B 结构与数据库表 A 结构相同。

实现 SAGA 模式

发起方实现如下：

```
private boolean transfer(String businessKey, String from, String to, BigDecimal amount,
Map<String, Object> extParams) {
try {
Map<String, Object> params = new HashMap<>(4);
params.put("from", from);
params.put("to", to);
params.put("amount", amount);
     if (extParams != null) {
params.put("extParams", extParams);
}

StateMachineInstance inst = stateMachineEngine.startWithBusinessKey("transferBySaga", n
ull, businessKey, params);
logger.info("transferBySaga:{},{},{}", inst.getMachineId(), inst.getStatus(), inst.getC
ompensationStatus());
if (ExecutionStatus.SU.equals(inst.getStatus())
&& inst.getCompensationStatus() == null) {
//正向状态为成功，补偿状态为空（没为触发回滚），则交易成功
return true;
} else {
//如果执行到了Fail节点会有errorCode和errorMessage
String errorCode = (String)
inst.getContext().get(DomainConstants.VAR_NAME_STATEMACHINE_ERROR_CODE);
String errorMessage = (String)
inst.getContext().get(DomainConstants.VAR_NAME_STATEMACHINE_ERROR_MSG);
logger.warn("ErrorCode:" + errorCode + ", ErrorMsg:" + errorMessage + ", exception: " +
inst.getException());
return false;
}

} catch (Throwable t) {
logger.error("转账交易执行失败", t);
throw new RuntimeException(t);
}
}
```

参与者 A（扣钱）实现：

```
public interface FirstSagaAction {

/**
* 扣钱操作
* @param businessKey
* @param accountNo
* @param amount
* @return
*/
boolean amountMinus(String businessKey, String accountNo, BigDecimal amount,
Map<String, Object> extParams);

/**
* 补偿（冲正）扣钱操作
* @param businessKey
* @param accountNo
* @return
*/
boolean compensateAmountMinus(String businessKey, String accountNo);
}
```

参与者 B（加钱）实现：

```
public interface SecondSagaAction {

/**
* 加钱操作
* @param businessKey
* @param accountNo
* @param amount
* @return
*/
boolean amountAdd(String businessKey, String accountNo, BigDecimal amount, Map<String,
Object> extParams);

/**
* 补偿（冲正）加钱操作
* @param businessKey
* @param accountNo
* @return
*/
boolean compensateAmountAdd(String businessKey, String accountNo);
}
```

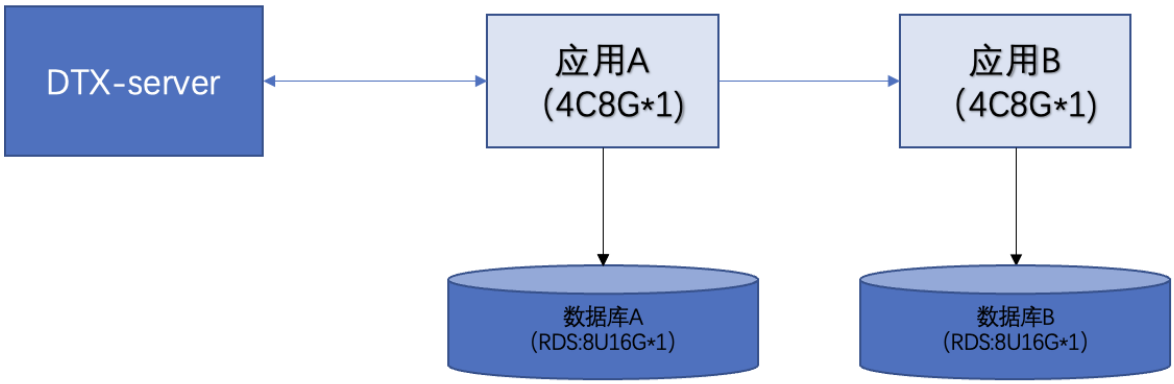
4.3. 压测结果

小规格压测结果

> **说明**
>
> A 的账号数据存储在“数据库 A”中，此数据库仅由“应用 A”访问；B 的账号数据存储“数据库 B”中，此数据库由“应用 B”访问。

规格说明如下：

- **服务端：4C8G*2（同库模式性能不敏感）**
- **应用 A：4C8G*1**
- **数据库 A（事务库）：RDS8C16G*1**
- **应用 B：4C8G*1**
- **数据库 B: RDS8C16G*1**
- **表结构数据一致：余额表和流水表**



压测样本数据策略

用户数据规模（账号数量）：5000->5000，以验证在不同业务数据规模的场景下，分布式事务的 TCC 模式、Saga 模式和 FMT 模式的性能表现。

压测模式	A 组账号数	B 组账号数
TCC、FMT、Saga	5000	5000

压测命令

- **TCC 模式**
  - **压测命令：**

```
./jmeter -n -t /home/admin/jmeter/dtx-poc-tcc.jmx -l
/home/admin/jmeter/results/result.jtl -e -o /home/admin/jmeter/results -JthreadNum=
120 -Jduration=120 -JIP=localhost -JPORT=8341 -
JACCOUNT_PATH=/home/admin/jmeter/accounts_tcc.txt
```

  - **压测配置：**dtx-poc-tcc.jmx
- **FMT 模式**

- 压测命令：

```
./jmeter -n -t /home/admin/jmeter/dtx-poc-fmt.jmx -l
/home/admin/jmeter/results/result.jtl -e -o /home/admin/jmeter/results -JthreadNum=
120 -Jduration=120 -JIP=localhost -JPORT=8341 -
JACCOUNT_PATH=/home/admin/jmeter/accounts_fmt.txt
```

- 压测配置：dtx-poc-fmt.jmx

- **SAGA 模式**
  - 压测命令：

```
./jmeter -n -t /home/admin/jmeter/dtx-poc-saga.jmx -l
/home/admin/jmeter/results/result.jtl -e -o /home/admin/jmeter/results -JthreadNum=
400 -Jduration=120 -JIP=localhost -JPORT=8341 -
JACCOUNT_PATH=/home/admin/jmeter/accounts_saga.txt
```

  - 压测配置：dtx-poc-saga.jmx

压测结果

如下图所示，用户数据规模（A->B）在 5000->5000 的场景下，TCC、FMT、Saga 模式的 TPS 与 Response Time 数据如下：

> **说明**
> SAGA（总）为 SAGA 模式下 TPS 总数，SAGA（成）为 SAGA 模式下成功 TPS 总数。

- A 组转账至 B 组（5000->5000）TPS

用户量（A 组转账至 B 组）	TPS				
	并发数	TCC（总）	FMT（总）	SAGA（总）	SAGA（成）
A 组：5000 B 组：5000	1	50	40	50	50
	5	250	200	250	250
	10	460	380	500	500
	20	800	695	980	980
	50	1450	1220	1500	1500
	100	1500	1400	1600	1600
	200	**1700**	**1500**	**1750**	**1750**

	300	1650	1450	-	-
	400	1600	1450	1750	1750
	500	1580	1400	1400	1400
	700	1500	1300	1300	1300

- A 组转账至 B 组（5000->5000）Response Time

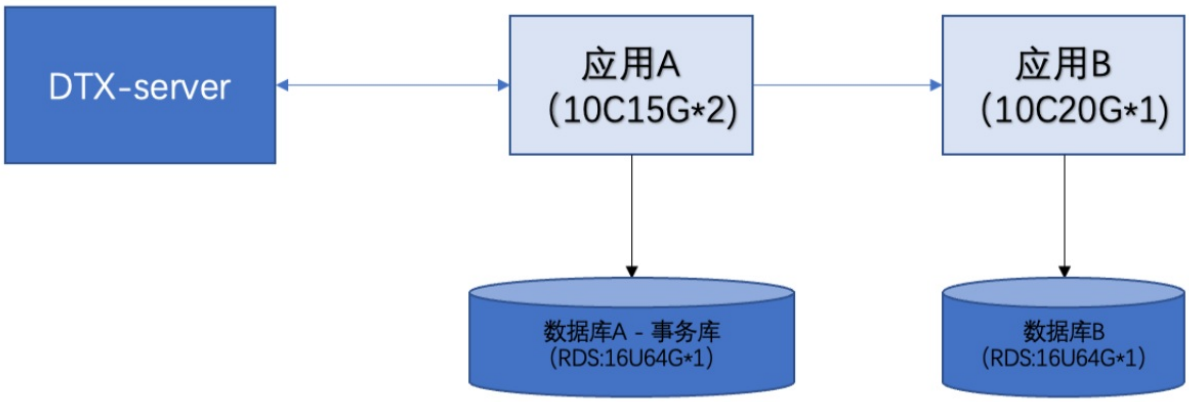
Response Time（成功）			
并发数	**TCC**	**FMT**	**Saga**
1	18	26	18
5	26	27	19
10	34	35	20
20	40	44	21
50	55	58	33
100	85	90	62
200	**135**	**160**	**110**
300	175	250	-
400	210	325	207
500	275	380	336
700	390	520	519

大规格压测结果

> **说明**
>
> A 的账号数据存储在“数据库 A”中，此数据库仅由“应用 A”访问；B 的账号数据存储“数据库 B”中，此数据库由“应用 B”访问。

规格说明如下：

- **服务端：4C8G*2（同库模式性能不敏感）**
- **应用 A：10C15G*2**
- **数据库 A（事务库）：RDS16C64G*1**
- **应用 B：10C20G*1**
- **数据库 B：RDS 16C64G*1**
- **表结构数据一致：余额表和流水表**



压测样本数据策略

用户数据规模（账号数量）：5000->5000，以验证在不同业务数据规模的场景下，分布式事务的 TCC 模式、Saga 模式和 FMT 模式的性能表现。

压测模式	A 组账号数	B 组账号数
TCC、FMT、Saga	5000	5000

压测命令

- **TCC 模式**
  - **压测命令：**

```
./jmeter -n -t /home/admin/jmeter/dtx-poc-tcc.jmx -l
/home/admin/jmeter/results/result.jtl -e -o /home/admin/jmeter/results -JthreadNum=
120 -Jduration=120 -JIP=localhost -JPORT=8341 -
JACCOUNT_PATH=/home/admin/jmeter/accounts_tcc.txt
```

  - **压测配置：**dtx-poc-tcc.jmx
- **FMT 模式**

- 压测命令：

```
./jmeter -n -t /home/admin/jmeter/dtx-poc-fmt.jmx -l
/home/admin/jmeter/results/result.jtl -e -o /home/admin/jmeter/results -JthreadNum=
120 -Jduration=120 -JIP=localhost -JPORT=8341 -
JACCOUNT_PATH=/home/admin/jmeter/accounts_fmt.txt
```

- 压测配置：dtx-poc-fmt.jmx

- **SAGA 模式**
  - 压测命令：

```
./jmeter -n -t /home/admin/jmeter/dtx-poc-saga.jmx -l
/home/admin/jmeter/results/result.jtl -e -o /home/admin/jmeter/results -JthreadNum=
400 -Jduration=120 -JIP=localhost -JPORT=8341 -
JACCOUNT_PATH=/home/admin/jmeter/accounts_saga.txt
```

  - 压测配置：dtx-poc-saga.jmx

压测结果

如下图所示，用户数据规模（A->B）在 5000->5000 的场景下，TCC、FMT、Saga 模式的 TPS 与 Response Time 数据如下：

> **说明**
> SAGA（总）为 SAGA 模式下 TPS 总数。

- A 组转账至 B 组（5000->5000）TPS

用户量（A 组转账至 B 组）	TPS			
	并发数	TCC（总）	FMT（总）	SAGA（总）
A 组：5000 B 组：5000	1	50	40	50
	5	250	200	250
	10	500	400	500
	20	920	750	-
	50	1350	1200	1500
	100	1900	1700	1900
	200	**2400**	**1900**	2500

	300	2900	1950	-
	400	2850	1850	**3500**
	500	2800	1800	3400
	700	2800	1800	3300

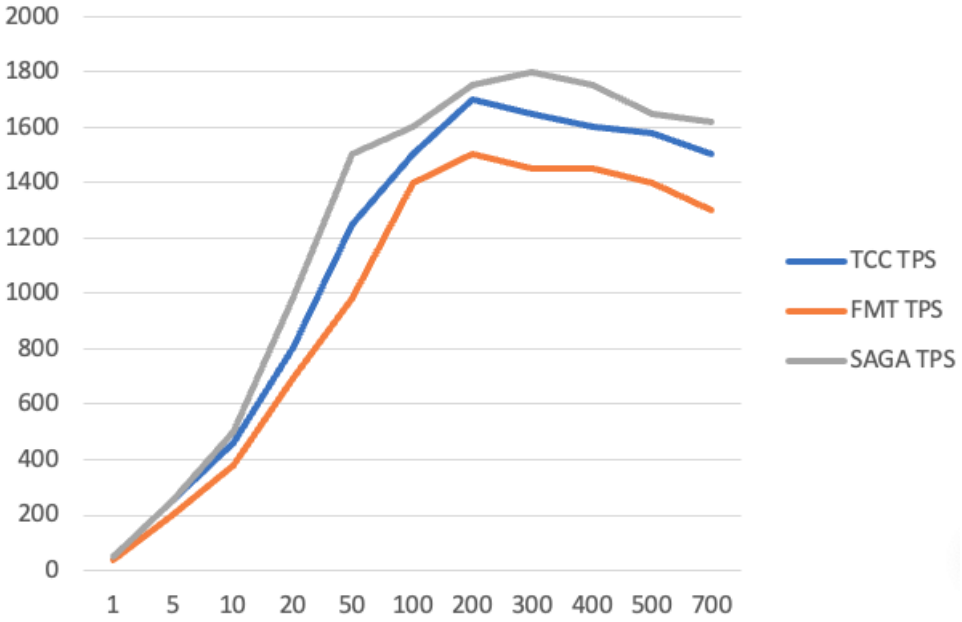
- A 组转账至 B 组（5000->5000）Response Time

Response Time（成功）			
并发数	**TCC**	**FMT**	**Saga**
1	18	26	18
5	26	30	19
10	34	40	-
20	40	44	-
50	46	55	30
100	66	78	-
200	**93**	**105**	79
300	107	120	-
400	128	145	113
500	155	176	130
600	-	-	145
700	196	218	189

总结对比

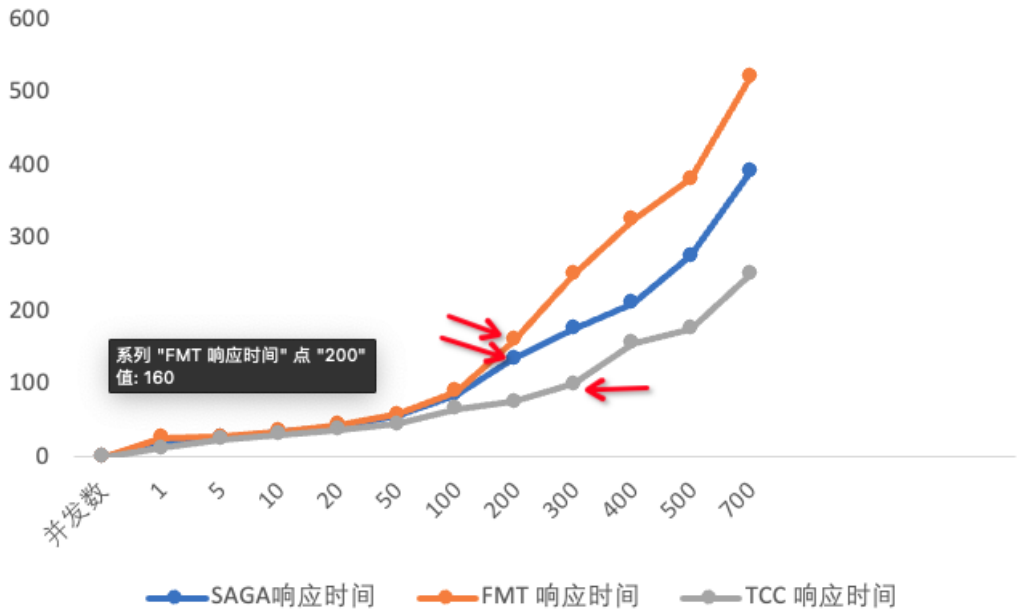
小规格压测 TPS 对比（TCC 1700、FMT 1500、SAGA 1800）

并发	TCC TPS	FMT TPS	SAGA TPS
1	50	40	50
5	250	200	250
10	460	380	500
20	800	695	980
50	1250	980	1500
100	1500	1400	1600
200	**1700**	**1500**	1750
300	1650	1450	**1800**
400	1600	1450	1750
500	1580	1400	1650
700	1500	1300	1620



小规格最高 TPS 下的 RT 数据（TCC 135、FMT 160、SAGA 100）

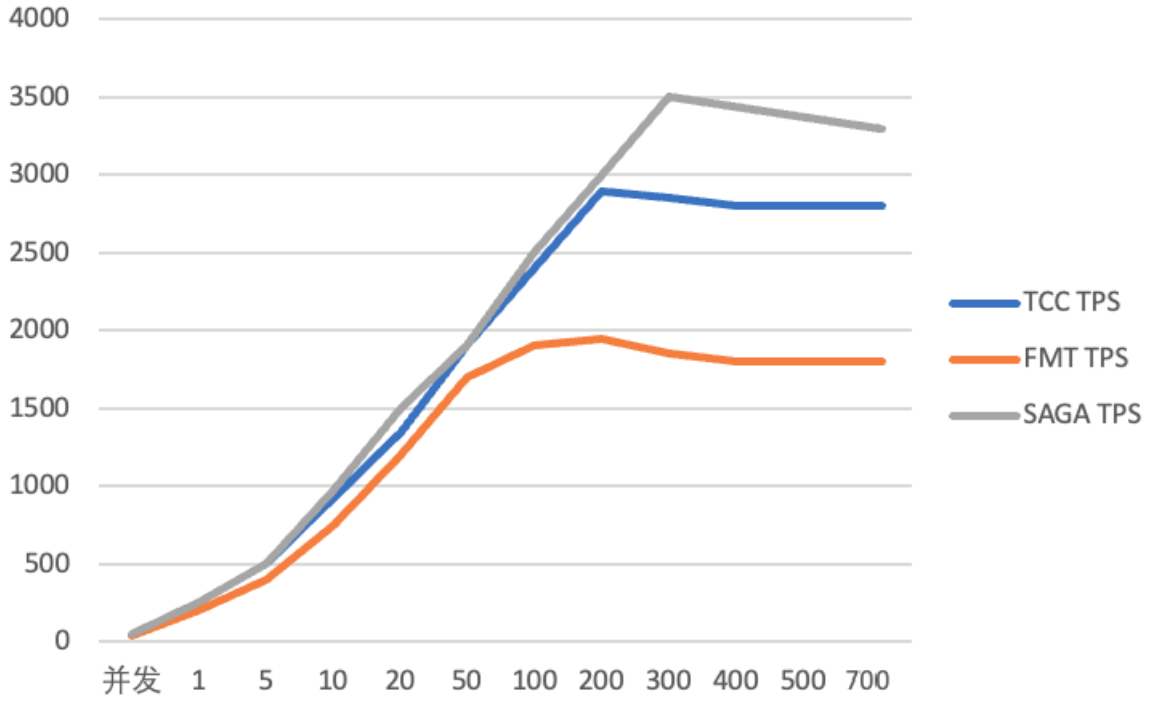
并发数	TCC 响应时间	FMT 响应时间	SAGA响应时间
1	18	26	18
5	26	27	19
10	34	35	20
20	40	44	21
50	55	58	33
100	85	90	62
200	**135**	**160**	110
300	175	250	**165**
400	210	325	227
500	275	380	336
700	390	520	519



大规格压测 TPS 对比（TCC 2900、FMT 1950、SAGA 3500）

并发	TCC TPS	FMT TPS	SAGA TPS
1	50	40	50
5	250	200	250
10	500	400	500
20	920	750	980
50	1350	1200	1500
100	1900	1700	1900
200	2400	1900	2500
300	**2900**	**1950**	3000
400	2850	1850	**3500**
500	2800	1800	3400

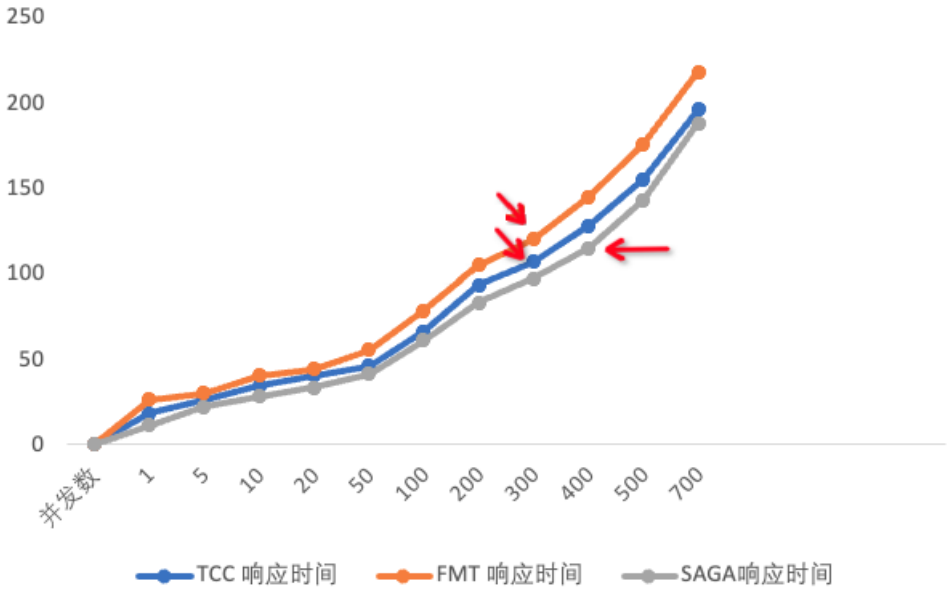
700	2800	1800	3300



大规格最高 TPS 下的 RT 数据（TCC 107、FMT 120、SAGA 113）

并发数	TCC 响应时间	FMT 响应时间	SAGA 响应时间
1	18	26	18
5	26	30	19
10	34	40	20
20	40	44	21
50	46	55	30
100	66	78	58
200	93	105	79
300	**107**	**120**	102

400	128	145	**113**
500	155	176	130
700	196	218	189

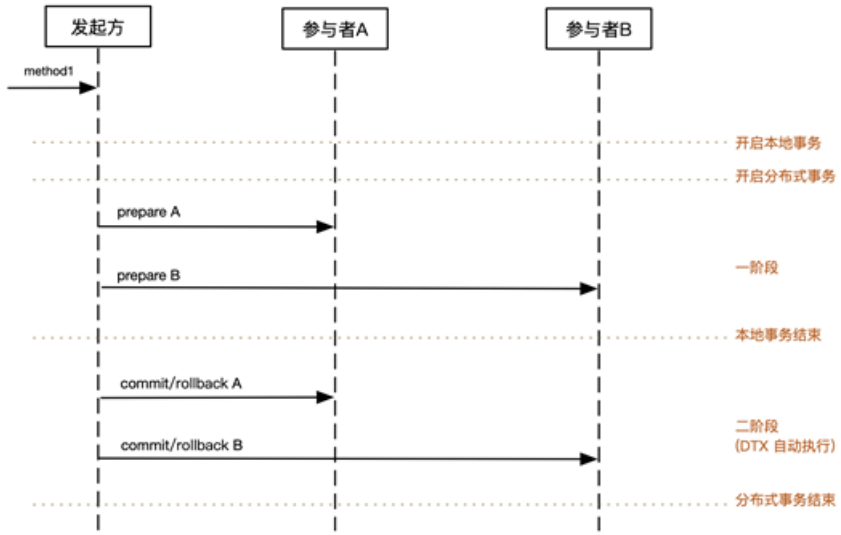


5.功能原理

5.1. 两阶段提交

二阶段提交协议（Two-phase Commit Protocol，简称 2PC）是分布式事务的核心协议。在此协议中，一个事务管理器（Transaction Manager，简称 TM）协调 1 个或多个资源管理器（Resource Manager，简称 RM）的活动，所有资源管理器向事务管理器汇报自身活动状态，由事务管理器根据各资源管理器汇报的状态（完成准备或准备失败）来决定各资源管理器是“提交”事务还是进行“回滚”操作。

二阶段提交的具体流程如下：



1. 应用程序向事务管理器提交请求，发起分布式事务；
2. 在第一阶段，事务管理器联络所有资源管理器，通知它们准备提交事务（prepare）；
3. 各资源管理器返回完成准备（或准备失败）的消息给事务管理器（响应超时算作失败）；
4. 在第二阶段：

所有资源管理器完成准备，事务管理器协调各资源管理器提交事务

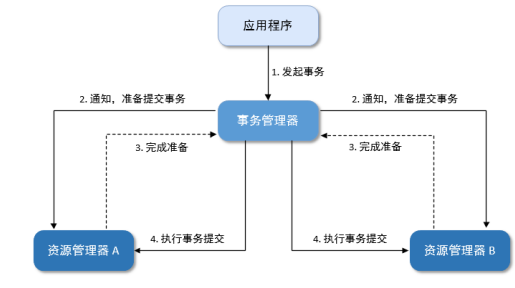


图 1

任一资源管理器准备失败，事务管理器协调各资源管理器回滚事务

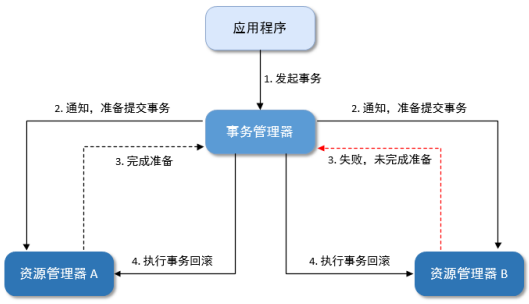


图 2

- 如果所有资源管理器均完成准备（如图 1），则事务管理器会通知所有资源管理器执行事务提交（commit）；
- 如果任一资源管理器准备失败（如图 2 中的资源管理器 B），则事务管理器会通知所有资源管理器进行事务回滚（rollback）。

通过事务管理器的两阶段协调，保证所有资源管理器的状态最终是一致的，要么全部都提交，要么全部都是回滚，从而保障了分布式事务的最终一致性。

5.2. TCC 模式

5.2.1. TCC 原理概述

TCC （Try-Confirm-Cancel）是二阶段提交协议（Two-phase Commit Protocol，简称 2PC）的扩展，Try（初步操作）操作对应 2PC 中一阶段的准备提交事务（Prepare），Confirm（确认操作）对应 2PC 中二阶段事务提交（Commit），Cancel（取消操作）对应 2PC 中二阶段事务回滚（Rollback）。

这三种操作的业务含义如下：

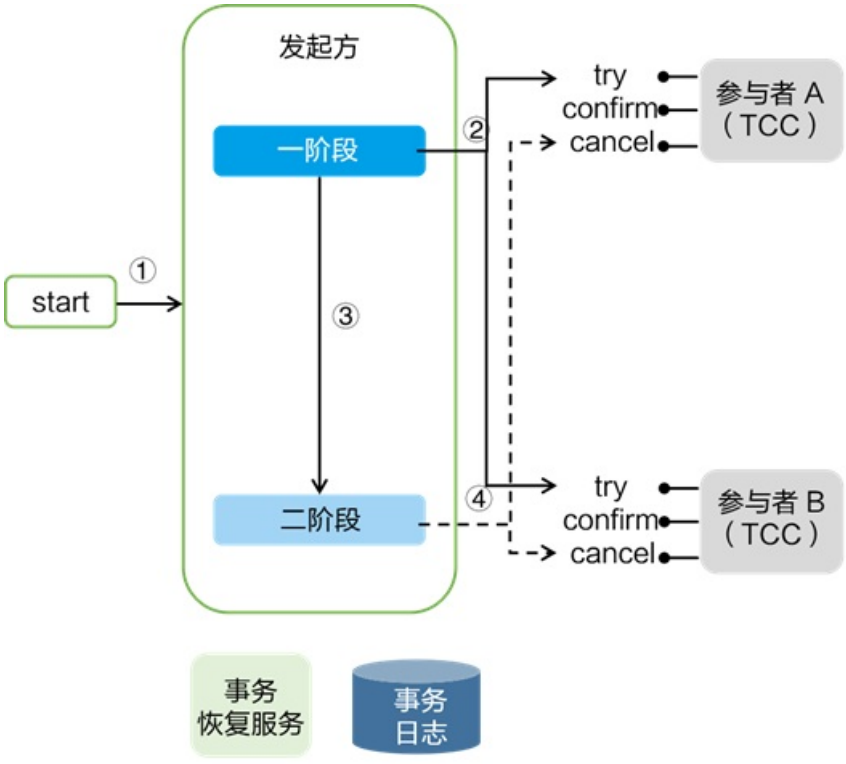
- Try 阶段：对业务系统做检测及资源预留。
- Confirm 阶段：对业务系统做确认提交。默认 Confirm 阶段是不会出错的，只要 Try 成功，Confirm 一定成功。
- Cancel 阶段：当业务执行出现错误，需要回滚的状态下，执行业务取消，释放预留资源。

与 2PC 不同的是，TCC 是一种编程模型，是应用层的 2PC；TCC 的 3 个操作均由编码实现，通过编码实现了 2PC 资源管理器的功能。

TCC 自编码的特性决定 TCC 资源管理器可以跨数据库、跨应用实现资源管理，将对不同的数据库访问、不同的业务操作通过编码方式转换一个原子操作，解决了复杂业务场景下的事务问题。同时 TCC 的每一个操作对于数据库来讲都是一个本地数据库事务，操作结束则本地数据库事务结束，数据库的资源也就被释放；这就规避了数据库层面的 2PC 对资源占用导致的性能低下问题。

DTX 的 TCC 模式需要用户根据自己的业务场景实现 Try、Confirm 和 Chanel 三个操作：

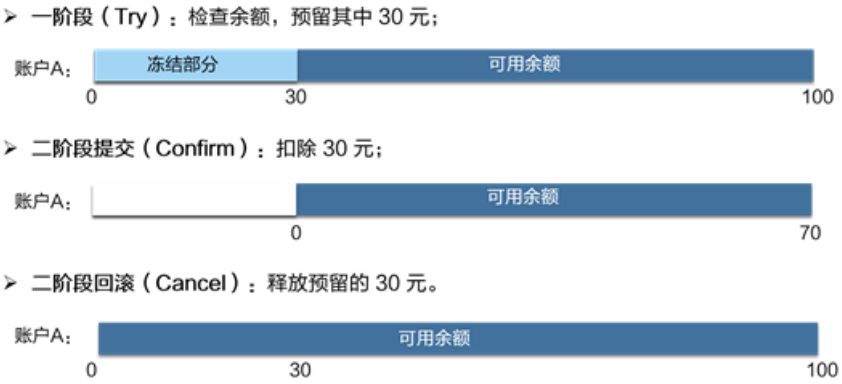
- 事务发起方在一阶段执行 Try 方式；
- 在二阶段提交执行 Confirm 方法；
- 二阶段回滚执行 Cancel 方法。



TCC 业务模型分 2 阶段设计：

用户接入 TCC ，最重要的是考虑如何将自己的业务模型拆成两阶段来实现。

以“扣钱”场景为例，在接入 TCC 前，对 A 账户的扣钱，只需一条更新账户余额的 SQL 便能完成；但是在接入 TCC 之后，用户就需要考虑如何将原来一步就能完成的扣钱操作，拆成两阶段，实现成三个方法，并且保证一阶段 Try 成功后二阶段 Confirm 一定能成功。



如上图所示，

- Try 方法作为一阶段准备方法，需要做资源的检查和预留。在扣钱场景下，Try 要做的事情是就是检查账户余额是否充足，预留转账资金，预留的方式就是冻结 A 账户的转账资金。Try 方法执行之后，账号 A 余额虽然还是 100 元，但是其中 30 元已经被冻结了，不能被其他事务使用。
- 二阶段 Confirm 方法执行真正的扣钱操作。Confirm 会使用 Try 阶段冻结的资金，执行账号扣款。Confirm 方法执行之后，账号 A 在一阶段中冻结的 30 元已经被扣除，账号 A 余额变成 70 元 。
- 如果二阶段是回滚的话，就需要在 Cancel 方法内释放一阶段 Try 冻结的 30 元，使账号 A 的回到初始状态，100元全部可用。

用户接入 TCC 模式，最重要的事情就是考虑如何将业务模型拆成 2 阶段，实现成 TCC 的 3 个方法，并且保证 Try 成功 Confirm 一定能成功。相对于 FMT 模式，TCC 模式对业务代码有一定的侵入性，但是 TCC 模式无 FMT 模式的全局行锁，TCC 性能会比 FMT 模式高很多。

5.2.2. TCC 框架处理流程

DTX 框架会把每组 TCC 接口当做一个 Resource，称为 TCC Resource。这套 TCC 接口一般是 RPC。DTX 框架处理流程如下：

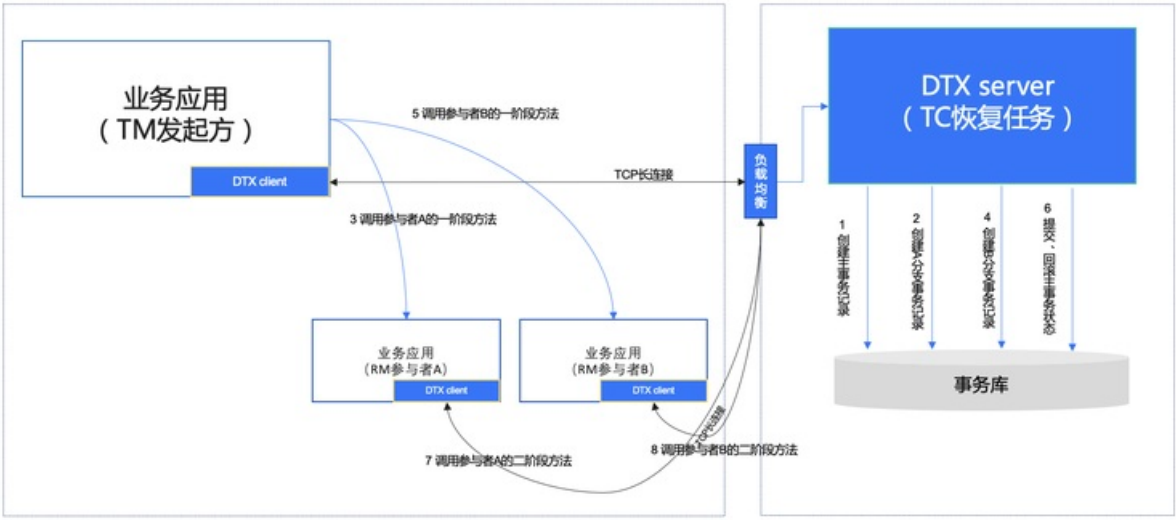
1. 在业务启动时，DTX 框架会自动扫描识别到 TCC 接口的调用方和发布方，即 `sofa:reference` 、 `sofa:service` 、 `dubbo:reference` 、 `dubbo:service` 等。
2. 扫描到 TCC 接口的调用方和发布方之后：
   - 如果是发布方

     会在业务启动时向 TC 注册 TCC Resource，与 DataSource Resource 一样，每个资源也会带有一个资源 ID。
   - 如果是调用方

     DTX 框架会给调用方加上切面，该切面会拦截所有对 TCC 接口的调用。每调用一次 Try 接口，切面会先向 TM 注册一个分支事务，然后才去执行原来的 RPC 调用。当请求链路调用完成后，TM 通过分支事务的资源 ID 回调到正确的参与者去执行对应 TCC 资源的 Confirm 或 Cancel 方法。

DTX 框架本身很简单，主要是扫描 TCC 接口，注册资源，拦截接口调用，注册分支事务，最后回调二阶段接口。最核心的实际上是 TCC 接口的实现逻辑。

具体框架和 DTX 服务端以及应用是如何完成交互的可以参考下图：



5.2.3. TCC 设计原理

从 TCC 模型的框架可以发现，TCC 模型的核心在于 TCC 接口的设计。用户在接入 TCC 时，大部分工作都集中在如何实现 TCC 服务上。

- **需要将操作分成两阶段完成**

  TCC（Try-Confirm-Cancel）分布式事务模型相对于 XA 等传统模型，其特征在于它不依赖 RM 对分布式事务的支持，而是通过对业务逻辑的分解来实现分布式事务。

  TCC 模型认为，对于业务系统中一个特定的业务逻辑在对外提供服务时，必须接受一些不确定性，即对业务逻辑初步操作的调用仅是一个临时性操作，调用它的主业务服务保留了后续的取消权。如果主业务服务认为全局事务应该回滚，它会要求取消之前的临时性操作，这就对应从业务服务的取消操作。而当主业务服务认为全局事务应该提交时，它会放弃之前临时性操作的取消权，这对应从业务服务的确认操作。每一个初步操作，最终都会被确认或取消。因此，针对一个具体的业务服务，TCC 分布式事务模型需要业务系统提供三段业务逻辑：

  i. 初步操作 Try。

     完成所有业务检查，预留必须的业务资源。

  ii. 确认操作 Confirm。

     真正执行的业务逻辑，不做任何业务检查，只使用 Try 阶段预留的业务资源。因此，只要 Try 操作成功，Confirm 必须能成功。另外，Confirm 操作需满足幂等性，保证一笔分布式事务能且只能成功一次。

  iii. 取消操作 Cancel。

     释放 Try 阶段预留的业务资源。同样的，Cancel 操作也需要满足幂等性。

- **要根据自身的业务模型控制并发**

  该操作这个对应 ACID 中的隔离性。详见请参见 账务系统模型优化 。

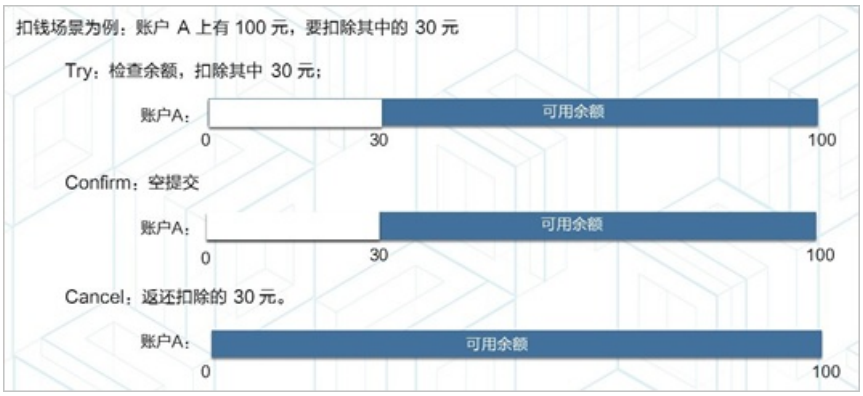
5.2.4. TCC 设计案例

5.2.4.1. 模型设计

以金融核心链路里的账务服务为例进行分析。首先一个最简化的账务模型就是图中所列，每个用户或商户有一个账户及其可用余额。然后，分析下账务服务的所有业务逻辑操作，无论是交易、充值、转账、退款等，都可以认为是对账户的加钱与扣钱。

因此，我们可以把账务系统拆分成两套 TCC 接口，即两个 TCC Resource，一个是加钱 TCC 接口，一个是扣钱 TCC 接口。

场景为 A 转账 30 元给 B。账户 A 的余额中有 100 元，需要扣除其中 30 元。这里的余额就是所谓的业务资源，按照前面提到的原则，在第一阶段需要检查并预留业务资源，因此，我们在扣钱 TCC 资源的 Try 接口里先检查 A 账户余额是否足够，然后预留余额里的业务资源，即扣除 30 元。



在 Confirm 接口，由于业务资源已经在 Try 接口里扣除掉了，那么在第二阶段的 Confirm 接口里，可以什么都不用做。而在 Cancel 接口里，则需要把 Try 接口里扣除掉的 30 元还给账户。这是一个比较简单的扣钱 TCC 资源的实现，后面会继续优化它。

而在加钱的 TCC 资源里。在第一阶段 Try 接口里不能直接给账户加钱，如果这个时候给账户增加了可用余额，那么在一阶段执行完后，账户里的钱就可以被使用了。但是一阶段执行完以后，有可能是要回滚的。因此，真正加钱的动作需要放在 Confirm 接口里。对于加钱这个动作，第一阶段 Try 接口里不需要预留任何资源，可以设计为空操作。那相应的，Cancel 接口没有资源需要释放，也是一个空操作。只有真正需要提交时，再在 Confirm 接口里给账户增加可用余额。

这就是一个最简单的扣钱和加钱的 TCC 资源的设计。在扣钱 TCC 资源里，Try 接口预留资源扣除余额，Confirm 接口空操作，Cancel 接口释放资源，增加余额。在加钱 TCC 资源里，Try 接口无需预留资源，空操作；Confirm 接口直接增加余额；Cancel 接口无需释放资源，空操作。

5.2.4.2. 业务模型控制并发

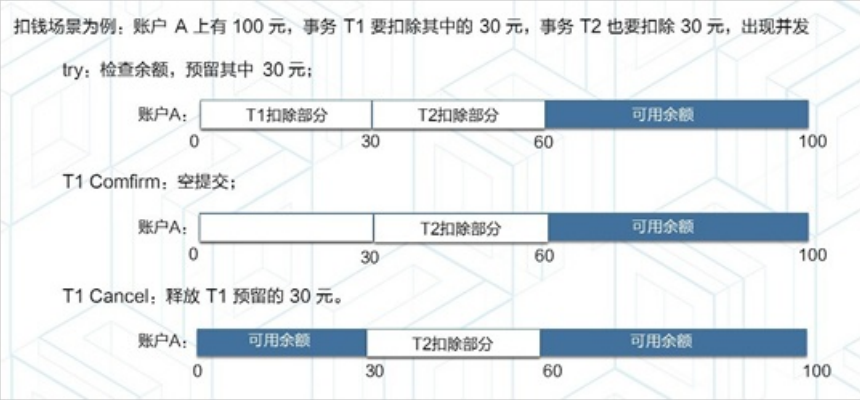
DTX 框架本身仅提供两阶段原子提交协议，保证分布式事务原子性。事务的隔离需要交给业务逻辑来实现。隔离的本质就是控制并发，防止并发事务操作相同资源而引起的结果错乱。

例如金融行业里管理用户资金，当用户发起交易时，一般会先检查用户资金，如果资金充足，则扣除相应交易金额，增加卖家资金，完成交易。如果没有事务隔离，用户同时发起两笔交易，两笔交易的检查都认为资金充足，实际上却只够支付一笔交易，结果两笔交易都支付成功，导致资损。

可以发现，并发控制是业务逻辑执行正确的保证，但是像两阶段锁这样的并发访问控制技术要求一直持有数据库资源锁直到整个事务执行结束，特别是在分布式事务架构下，要求持有锁到分布式事务第二阶段执行结束，也就是说，分布式事务会加长资源锁的持有时间，导致并发性能进一步下降。

因此，TCC 模型的隔离性思想就是通过业务的改造，在第一阶段结束之后，从底层数据库资源层面的加锁过渡为上层业务层面的加锁，从而释放底层数据库锁资源，放宽分布式事务锁协议，将锁的粒度降到最低，以最大限度提高业务并发性能。

还是以上面的例子举例，“账户 A 上有 100 元，事务 T1 要扣除其中的 30 元，事务 T2 也要扣除 30 元，出现并发”。在第一阶段 Try 操作中，需要先利用数据库资源层面的加锁，检查账户可用余额，如果余额充足，则预留业务资源，扣除本次交易金额，一阶段结束后，虽然数据库层面资源锁被释放了，但这笔资金被业务隔离，不允许除本事务之外的其它并发事务动用。

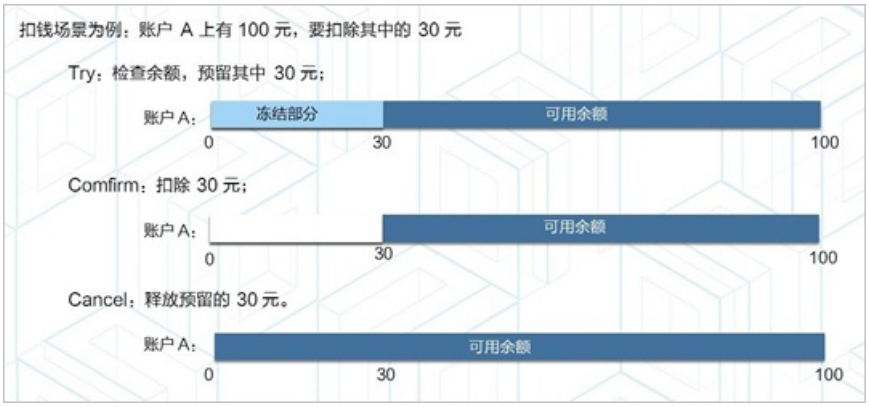


并发的事务 T2 在事务 T1 一阶段接口结束释放了数据库层面的资源锁以后，就可以继续操作，跟事务 T1 一样，加锁，检查余额，扣除交易金额。

事务 T1 和 T2 分别扣除的那一部分资金，相互之间无干扰。这样在分布式事务的二阶段，无论 T1 是提交还是回滚，都不会对 T2 产生影响，这样 T1 和 T2 可以在同一个账户上并发执行。一阶段结束以后，实际上采用业务加锁的方式，隔离账户资金，在第一阶段结束后直接释放底层资源锁，该用户和卖家的其他交易都可以立刻并发执行，而不用等到整个分布式事务结束，可以获得更高的并发交易能力。

5.2.4.3. 账务系统模型优化

上述模型中，为了简单说明 TCC 模型的设计思想，在第一阶段就把钱扣除了。在实际中，为了更好的用户体验，在第一阶段，一般不会直接把账户的余额扣除，而是冻结，这样给用户展示的时候，就可以很清晰的知道，哪些是可用余额，哪些是冻结金额。那业务模型变成什么样了呢？如图所示，需要在业务模型中增加冻结金额字段，用来表示账户有多少金额处以冻结状态。



既然业务模型发生了变化，那扣钱和加钱的 TCC 接口也应该相应的调整。还是以前面的例子来说明。

在扣钱的 TCC 资源里。Try 接口不再是直接扣除账户的可用余额，而是真正的预留资源，冻结部分可用余额，即减少可用余额，增加冻结金额。Confirm 接口也不再是空操作，而是使用 Try 接口预留的业务资源，即将该部分冻结金额扣除；最后在 Cancel 接口里，就是释放预留资源，把 Try 接口的冻结金额扣除，增加账户可用余额。加钱的 TCC资源由于不涉及冻结金额的使用，所以无需更改。

通过这样的优化，可以更直观的感受到 TCC 接口的预留资源、使用资源、释放资源的过程。

那并发控制又变成什么样了呢？跟前面大部分类似，在事务 T1 的第一阶段 Try 操作中，先锁定账户，检查账户可用余额，如果余额充足，则预留业务资源，减少可用余额，增加冻结金额。并发的事务 T2 类似，加锁，检查余额，减少可用余额金额，增加冻结金额。

这里可以发现，事务 T1 和T2 在一阶段执行完成后，都释放了数据库层面的资源锁，但是在各自二阶段的时候，相互之间并无干扰，各自使用本事务内第一阶段 Try 接口内冻结金额即可。这里大家就可以直观感受到，在每个事务的第一阶段，先通过数据库层面的资源锁，预留业务资源，即冻结金额。虽然在一阶段结束以后，数据库层面的资源锁被释放了，但是第二阶段的执行并不会被干扰，这是因为数据库层面资源锁释放以后通过业务隔离的方式为这部分资源加锁，不允许除本事务之外的其它并发事务动用，从而保证该事务的第二阶段能够正确顺利的执行。

5.2.4.4. 案例总结

通过以上案例说明，我们可以了解设计一套完善的 TCC 接口最主要的有两点，一点是将业务逻辑拆分成两个阶段完成，即 Try、Confirm、Cancel 接口。其中 Try 接口检查资源、预留资源、Confirm 使用资源、Cancel 接口释放预留资源。另外一点就是并发控制，采用数据库锁与业务加锁的方式结合。由于业务加锁的特性不影响性能，因此，尽可能降低数据库锁粒度，过渡为业务加锁，从而提高业务并发能力。

5.2.5. TCC 异常处理

5.2.5.1. 异常控制概述

在设计了一套完备的 TCC 接口之后，并不是真的高枕无忧了。在微服务架构下，很有可能出现网络超时、重发、机器宕机等一系列的异常问题。一旦遇到这些问题，就会导致分布式事务执行过程出现异常。根据 SOFAStack 内部多年的使用分析，最常见的主要是空回滚、幂等、悬挂这三种异常。

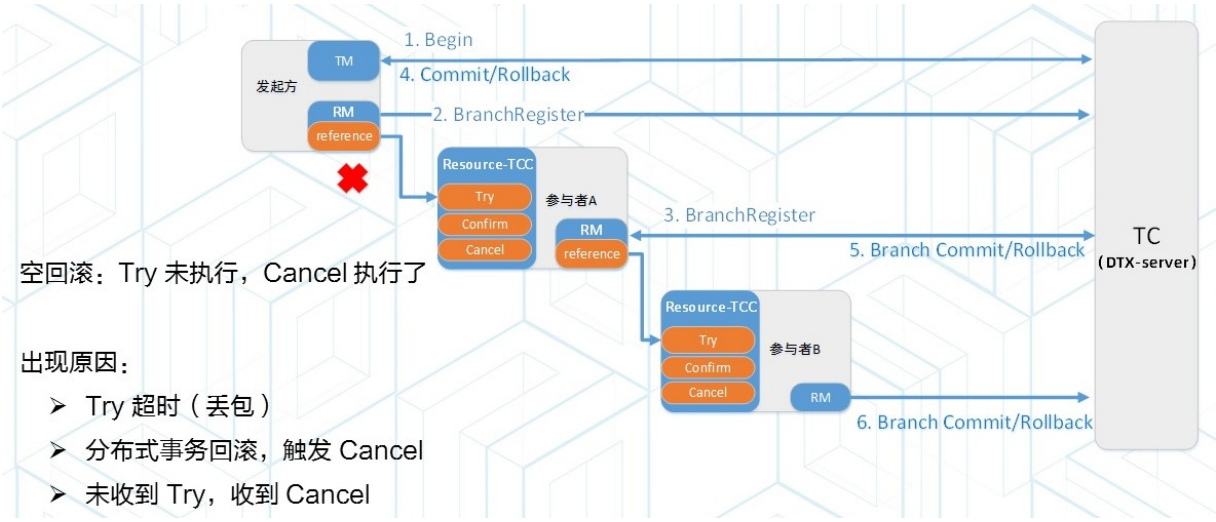
因此，TCC 接口里还需要解决这三类异常。实际上，这三类问题可以在 DTX 框架里完成。虽然业务之后无需关心，但是了解一下其内部实现机制，可以更好的排查问题。

5.2.5.2. 空回滚

空回滚就是对于一个分布式事务，在没有调用 TCC 资源 Try 方法的情况下，调用了二阶段的 Cancel 方法，Cancel 方法需要识别出这是一个空回滚，然后直接返回成功。

造成空回滚的原因

注册分支事务是在调用 RPC 时，DTX 框架的切面会拦截到该次调用请求，先向 TC 注册一个分支事务，然后才去执行 RPC 调用逻辑。如果 RPC 调用逻辑有问题，比如调用方机器宕机、网络异常，都会造成 RPC 调用失败，即未执行 Try 方法。但是分布式事务已经开启了，需要推进到终态，因此 TC 会回调参与者二阶段 Cancel 接口，从而形成空回滚。



空提交

分布式事务理论上不会产生空提交。因为如果是调用方宕机，那分布式事务默认是回滚的；如果是网络异常，RPC 调用失败，发起方应该通知 TC 回滚分布式事务。发起方可以在 RPC 调用失败的情况下依然通知 TC 提交，这时就会发生空提交，造成这种情况的原因一般是编码问题或开发者明确知道需要这样做。

解决方案

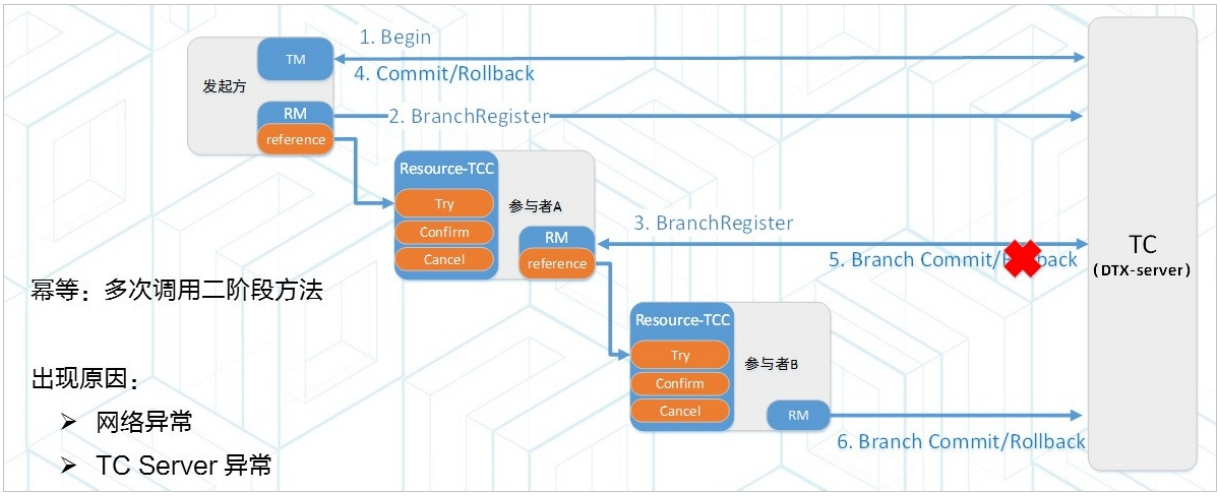
Cancel 要识别出空回滚，直接返回成功。关键就是要识别出这个空回滚。解决方法很简单，就是需要知道一阶段是否执行，如果执行了，那就是正常回滚；如果没执行，那就是空回滚。因此，需要一张额外的事务控制表，其中有分布式事务 ID 和分支事务 ID，第一阶段 Try 方法里会插入一条记录，表示一阶段执行了。Cancel 接口里读取该记录，如果该记录存在，则正常回滚；如果该记录不存在，则是空回滚。

5.2.5.3. 幂等

幂等就是对于同一个分布式事务的同一个分支事务，重复去调用该分支事务的第二阶段接口，因此，要求 TCC 的二阶段 Confirm 和 Cancel 接口保证幂等，不会重复使用或者释放资源。如果幂等控制没有做好，很有可能导致资损等严重问题。

造成重复提交或回滚的原因

如图所示，提交或回滚是一次 TC 到参与者的网络调用。因此，网络故障、参与者宕机等都有可能造成参与者 TCC 资源实际执行了二阶段防范，但是 TC 没有收到返回结果的情况，这时，TC 就会重复调用，直至调用成功，整个分布式事务结束。



解决方案

可以通过记录每个分支事务的执行状态来解决重复提交或者回滚的问题。如果已执行，那就不再执行；否则，正常执行。事务控制表的每条记录关联一个分支事务，我们可以在这张事务控制表上加一个状态字段，用来记录每个分支事务的执行状态。

```
CREATE TABLE `account_transaction` (
  `tx_id` varchar(100) NOT NULL COMMENT '事务TxId',
  `action_id` varchar(100) NOT NULL COMMENT '分支事务Id',
  `gmt_create` datetime NOT NULL COMMENT '创建时间',
  `gmt_modified` datetime NOT NULL COMMENT '修改时间',
  `user_id` varchar(100) NOT NULL COMMENT '账户UID',
  `amount` varchar(100) NOT NULL COMMENT '变动金额',
  `type` varchar(100) NOT NULL COMMENT '变动类型',
  `state` smallint(4) NOT NULL COMMENT '状态：1. 初始化; 2. 已提交; 3. 已回滚,
  PRIMARY KEY (`tx_id`, `action_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='业务流水表'
;
```

如图所示，该状态字段有三个值，分别是初始化、已提交、已回滚。Try 方法插入时，是初始化状态。二阶段 Confirm 和 Cancel 方法执行后修改为已提交或已回滚状态。当重复调用二阶段接口时，先获取该事务控制表对应记录，检查状态，如果已执行，则直接返回成功；否则正常执行。

5.2.5.4. 悬挂

悬挂就是对于一个分布式事务，其二阶段 Cancel 接口比 Try 接口先执行。因为允许空回滚的原因，Cancel 接口认为 Try 接口没执行，空回滚直接返回成功，对于 DTX 框架来说，认为分布式事务的二阶段接口已经执行成功，整个分布式事务就结束了。但是这之后 Try 方法才真正开始执行，预留业务资源，前面提到事务并发控制的业务加锁，对于一个 Try 方法预留的业务资源，只有该分布式事务才能使用，然而 DTX 框架认为该分布式事务已经结束，也就是说，当出现这种情况时，该分布式事务第一阶段预留的业务资源就再也没有人能够处理了，对于这种情况，我们就称为悬挂，即业务资源预留后没法继续处理。

造成悬挂的原因

在 RPC 调用时，先注册分支事务，再执行 RPC 调用，如果此时 RPC 调用的网络发生拥堵，通常 RPC 调用是有超时时间的，RPC 超时以后，发起方就会通知 TC 回滚该分布式事务，可能回滚完成后，RPC 请求才到达参与者真正执行，从而造成悬挂。

解决方案

根据悬挂出现的条件进行分析，悬挂是指二阶段 Cancel 执行完后，一阶段才执行。也就是说，为了避免悬挂，如果二阶段执行完成，那一阶段就不能再继续执行。因此，当一阶段执行时，需要先检查二阶段是否已经执行完成，如果已经执行，则一阶段不再执行；否则可以正常执行。因此，我们可以在二阶段执行时插入一条事务控制记录，状态为已回滚，这样当一阶段执行时，先读取该记录，如果记录存在，就认为二阶段已经执行；否则二阶段没执行。

5.2.5.5. 异常控制实现

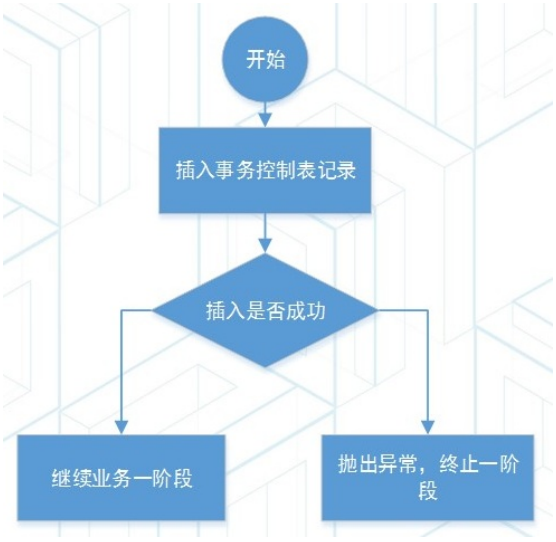
本文主要介绍一个 TCC 接口如何完整的解决空回滚、幂等、悬挂问题。

1. **Try 方法**

   Try 方法主要需要考虑两个问题：

   - Try 方法需要能够告诉二阶段接口，已经预留业务资源成功。
   - Try 方法需要检查第二阶段是否已经执行完成，如果已完成，则不再执行。

   因此，Try 方法的逻辑可以如图所示：



   先插入事务控制表记录，如果插入成功，说明第二阶段还没有执行，可以继续执行第一阶段。如果插入失败，则说明第二阶段已经执行或正在执行，则抛出异常，终止即可。

2. **Confirm 方法**

   因为 Confirm 方法不允许空回滚，也就是说，Confirm 方法一定要在 Try 方法之后执行。因此，Confirm 方法只需要关注重复提交的问题。可以先锁定事务记录。

- 如果事务记录为空，则说明是一个空提交，不允许，终止执行。
- 如果事务记录不为空，则继续检查状态是否为初始化。
  - 如果是，则说明一阶段正确执行，那二阶段正常执行即可。
  - 如果状态是已提交，则认为是重复提交，直接返回成功即可。
  - 如果状态是已回滚，也是一个异常，一个已回滚的事务，不能重新提交，需要能够拦截到这种异常情况，并报警。

3. **Cancel 方法**

   因为 Cancel 方法允许空回滚，并且要在先执行的情况下，让 Try 方法感知到 Cancel 已经执行，所以和 Confirm 方法略有不同。

   首先是锁定事务记录。

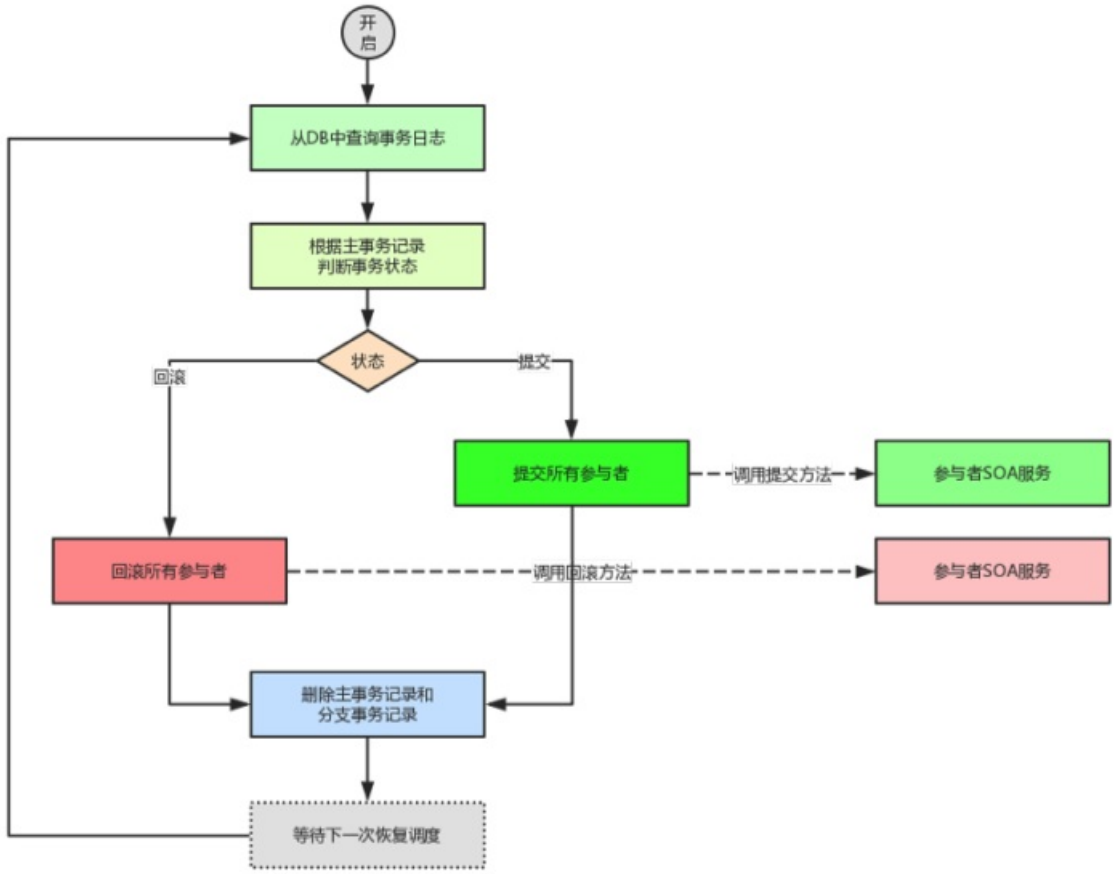
   - 如果事务记录为空，则认为 Try 方法还没执行，即是空回滚。空回滚的情况下，应该先插入一条事务记录，确保后续的 Try 方法不会再执行。
     - 如果插入成功，则说明 Try 方法还没有执行，空回滚继续执行。
     - 如果插入失败，则认为Try 方法正再执行，等待 TC 的重试即可。
   - 如果事务记录不为空，则说明 Try 方法已经执行完毕，再检查状态是否为初始化。
     - 如果是，则还没有执行过其他二阶段方法，正常执行 Cancel 逻辑。
     - 如果状态为已回滚，则说明这是重复调用，允许幂等，直接返回成功即可。
     - 如果状态为已提交，则同样是一个异常，一个已提交的事务，不能再次回滚。

在解决了这三类异常的情况下，我们的 TCC 接口设计就是比较完备的了。后续我们将会把这些解决方案移植到 DTX 框架中，由 DTX 框架来完成异常的处理，TCC 接口开发者就不再需要关心了。

5.2.5.6. 事务恢复

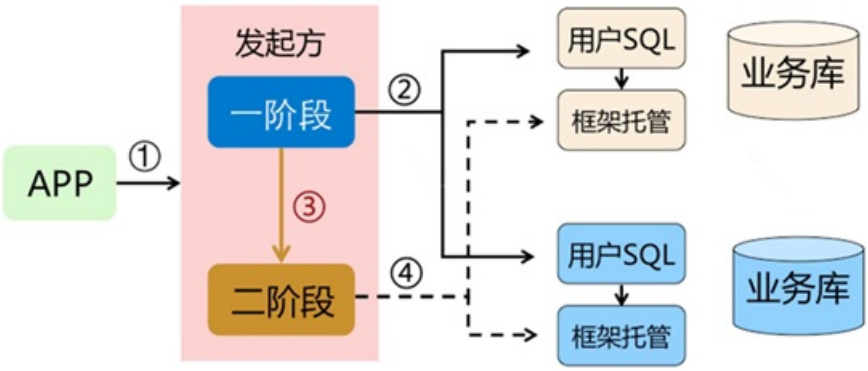
事务发起方在执行过程中的任意环节，均可能发生宕机、重启、网络中断等异常情况，此时事务执行异常中断，事务处于“非最终一致”状态（可能参与者只执行了一阶段未执行二阶段，可能部分参与者执行了二阶段部分参与者未执行二阶段）。

事务恢复服务的功能就是不断扫描事务日志，找到异常中断的事务，根据主事务记录和分支事务记录的信息，去完成剩余参与者的提交或者回滚，使整个分布式事务内所有参展达到最终一致的状态（全部都提交或全部都回滚）。



5.3. FMT 模式

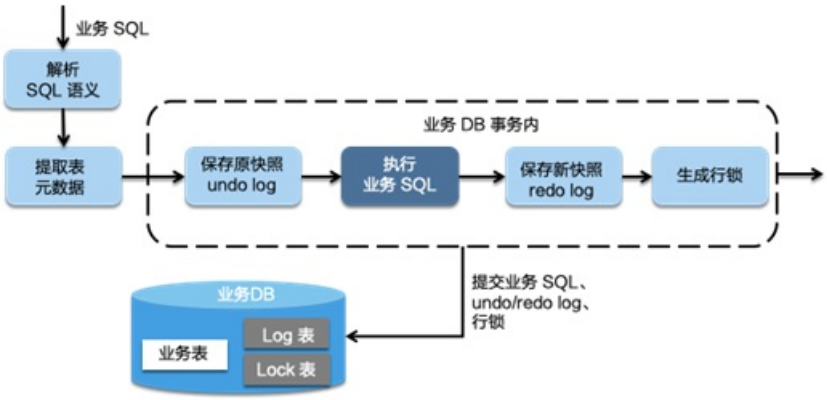
FMT 模式是一种无侵入的分布式事务解决方案。在模式下，用户只需关注自己的“业务 SQL”，用户的 “业务 SQL” 作为一阶段，DTX 框架会自动生成事务的二阶段提交和回滚操作。



FMT 模式如何做到对业务的无侵入：

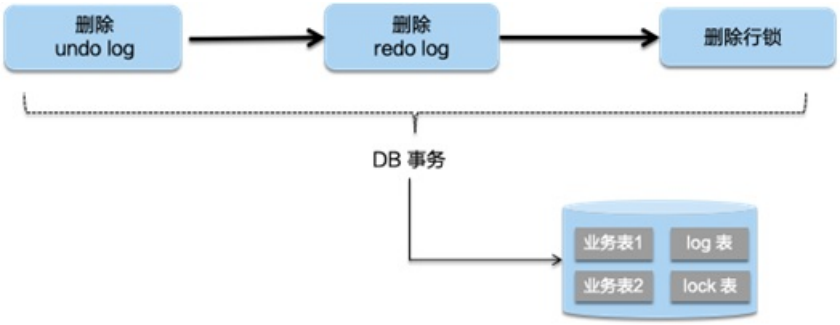
- 一阶段

  在一阶段，DTX 会拦截“业务 SQL”，首先解析 SQL 语义，找到“业务 SQL”要更新的业务数据，在业务数据被更新前，将其保存成“undo log”，然后执行“业务 SQL”更新业务数据，在业务数据更新之后，再将其保存成“redo log”，最后生成行锁。以上操作全部在一个数据库事务内完成，这样保证了一阶段操作的原子性。



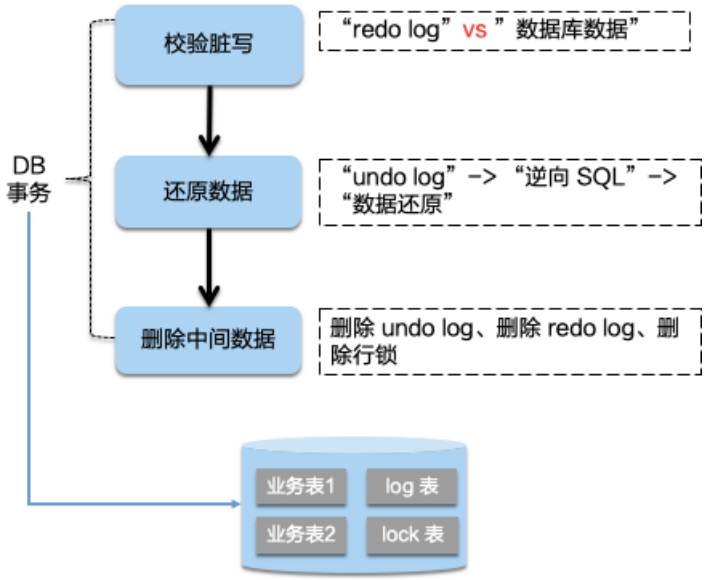
- 二阶段提交

  二阶段如果是提交的话，因为“业务 SQL”在一阶段已经提交至数据库， 所以 DTX 框架只需将一阶段保存的快照数据和行锁删掉，完成数据清理即可。



- 二阶段回滚

  二阶段如果是回滚的话，DTX 就需要回滚一阶段已经执行的“业务 SQL”，还原业务数据。回滚方式是用“undo log”还原业务数据；但在还原前要首先要校验脏写，对比“数据库当前业务数据”和“redo log”，如果两份数据完全一致就说明没有脏写，可以还原业务数据，如果不一致就说明有脏写，出现脏写就需要转人工处理。



FMT 模式的一阶段、二阶段提交和回滚均由 DTX 框架自动生成，用户只需编写“业务 SQL”，便能轻松接入分布式事务，FMT 模式是一种对业务无任何侵入的分布式事务解决方案。

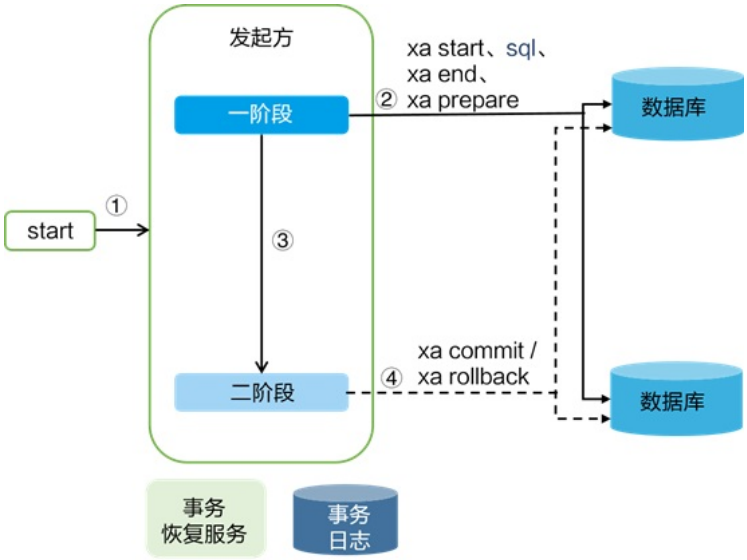
5.4. XA 模式

XA 模式是 DTX 的一种无侵入的分布式事务解决方案，任何实现了 XA 协议的数据库都可以作为资源参与到分布式事务中。目前主流数据库，例如 MySql、Oracle、DB2等均支持 XA 协议。

XA 协议有一系列的指令，分别对应一阶段和二阶段操作。

- `xa start` 和 `xa end` 用于开启和结束 XA 事务。
- `xa prepare` 用于预提交 XA 事务，对应一阶段准备。
- `xa commit` 和 `xa rollback` 用于提交、回滚 XA 事务，对应二阶段提交和回滚。

在 XA 模式下，每一个 XA 事务都是一个事务参与者。分布式事务开启之后，首先在一阶段执行 `xa start` 、业务 SQL 、`xa end` 和 `xaprepare` ，完成 XA 事务的执行和预提交。二阶段如果是提交，就执行 `xa commit` ；如果是回滚，则执行 `xa rollback` 。这样能保证所有 XA 事务都提交或者都回滚。



方案流程

XA 模式下，用户只需关注自己的业务 SQL，DTX 框架会自动生成一阶段、二阶段操作。XA 模式的实现如下：

- 一阶段

  在 XA 模式的一阶段，DTX 会拦截业务 SQL，在业务 SQL 之前开启 XA 事务（xa start），然后执行业务 SQL，结束 XA 事务 `xa end` ，最后预提交 XA 事务（xa prepare），这样便完成业务 SQL 的准备操作。

- 二阶段
  - 二阶段提交： 执行 `xa commit` 指令提交 XA 事务，此时业务 SQL 才算真正的提交至数据库。

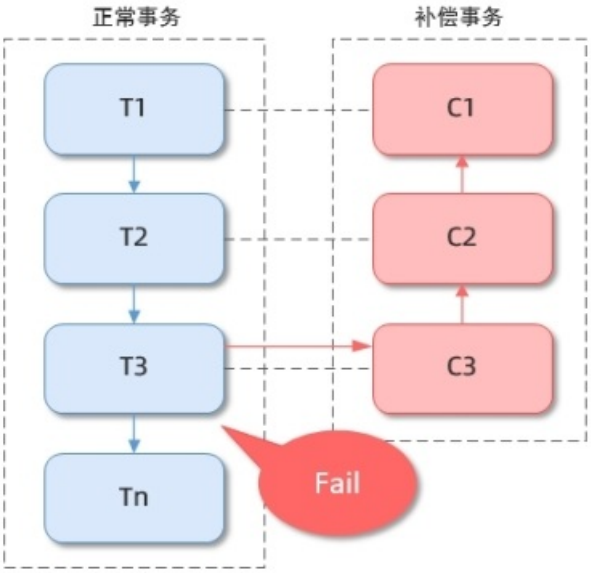
- 二阶段回滚： 执行 `xa rollback` 指令回滚 XA 事务，完成业务 SQL 回滚，释放数据库锁资源。

XA 模式下，用户只需关注业务 SQL ，DTX 会自动生成一阶段、二阶段提交和二阶段回滚操作。XA 模式和 AT 模式一样是一种对业务无侵入性的解决方案，但与 AT 模式不同的是，XA 模式将快照数据和行锁等通过 XA 指令委托给了数据库来完成，这样 XA 模式实现更加轻量化。

5.5. Saga 模式

5.5.1. Saga 原理概述

在 Saga 模式中，业务流程的每个参与者都提交本地事务，当出现某一个参与者失败，则补偿前面已经成功的参与者。如下图所示，左边是正常事务流程。当正常流程出现异常时，触发右边是补偿流程。Tn 是正向服务，Cn 对应是 Tn 的补偿服务。



5.5.2. Saga 设计原理

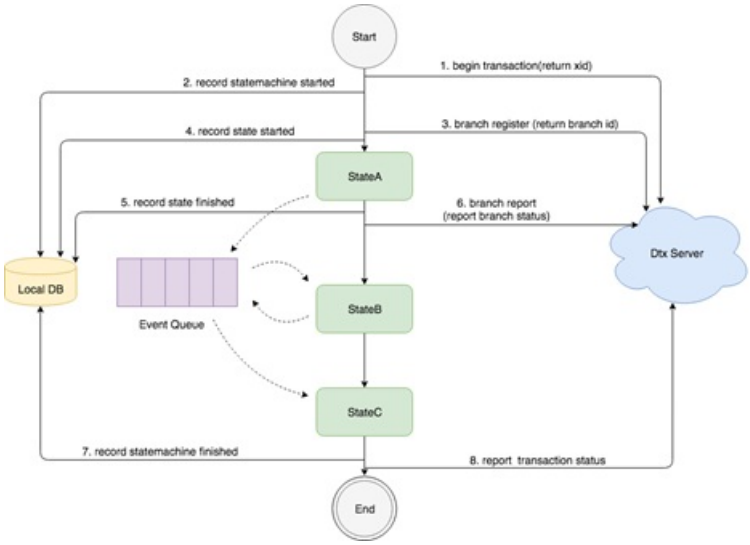
Saga 模式是基于状态机引擎来实现的，机制如下：

1. 通过状态图来定义服务调用的流程并生成 JSON 状态语言定义文件。
2. 状态图中一个节点可以是调用一个服务，节点可以配置它的补偿节点。
3. 状态图 JSON 由状态机引擎驱动执行，当出现异常时状态引擎反向执行已成功节点对应的补偿节点将事务回滚。

> **重要**
>
> 异常发生时是否进行补偿，也可由用户自定义决定。

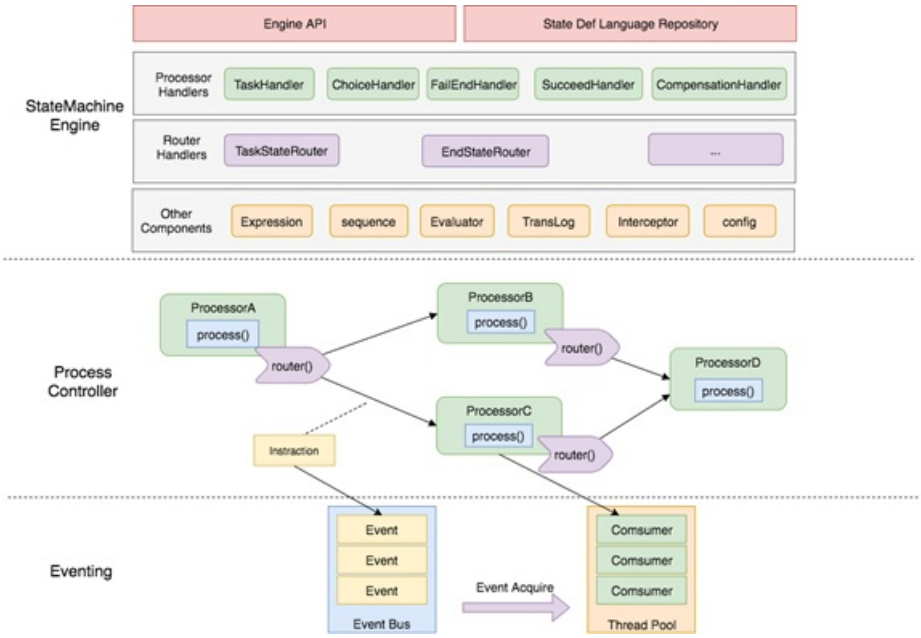
Saga 模式可以实现服务编排需求，支持单项选择、子流程、参数转换、参数映射、异步调用、服务执行状态判断、异常捕获等功能。

状态机引擎原理



- 如上图所示，“状态”的执行是先执行 StateA，再执行 StateB，然后执行 StateC。
- “状态”的执行是基于事件驱动的模型，StateA 执行完成后，会产生路由消息放入 EventQueue，事件消费端从 EventQueue 取出消息，执行 StateB。
- 在整个状态机启动时会调用 Dtx Server 开启分布式事务，并生产 xid，然后记录“状态机实例”启动事件到本地数据库。
- 当执行到一个“状态”时，会调用 Dtx Server 注册分支事务，并生产 branchId，然后记录“状态实例”开始执行事件到本地数据库（该步骤在 Saga 状态机模式非必须，也有开关关闭，以提高性能）。
- 当一个“状态”执行完成后会记录“状态实例”执行结束事件到本地数据库，然后调用 Dtx Server上报分支事务的状态（该步骤在 Saga 状态机模式非必须，也有开关关闭，以提高性能）。
- 当整个状态机执行完成，会记录“状态机实例”执行完成事件到本地数据库，然后调用 Dtx Server 提交或回滚分布式事务。

状态机引擎设计



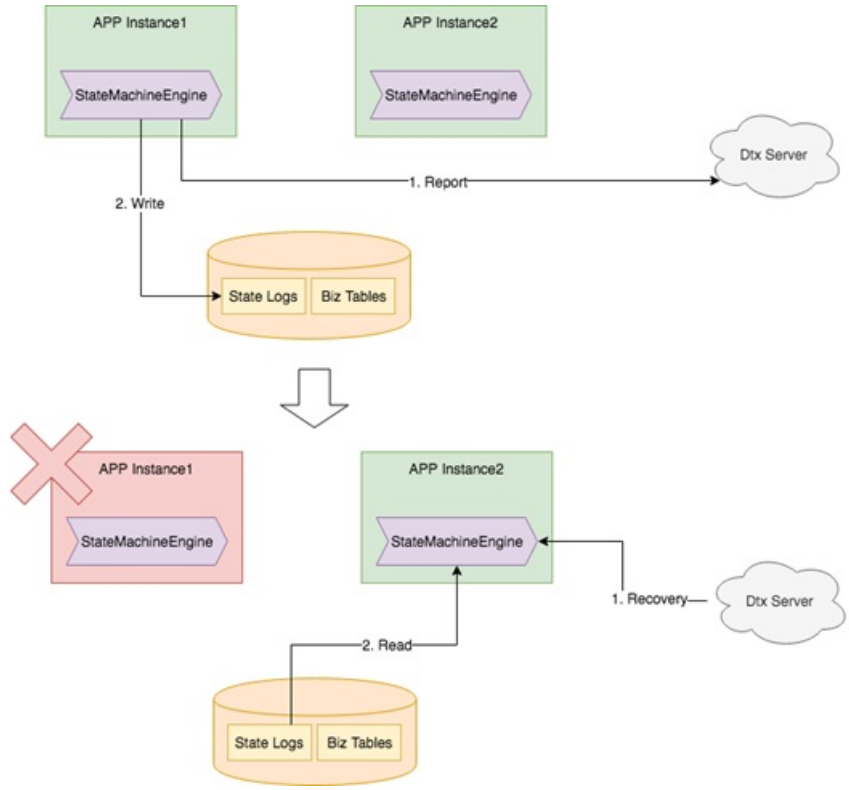
状态机引擎的设计主要分成三层，上层依赖下层，从下往上分别是：

- **Eventing 层**：实现事件驱动架构，可以压入事件，并由消费端消费事件，本层不关心事件是什么消费端执行什么，由上层实现。

- **ProcessController 层：** 由于下层的 Eventing 驱动一个“空”流程引擎的执行，“State”的行为和路由都未实现，由上层实现。
- **StateMachineEngine 层：**
  - 实现状态机引擎每种 State 的行为和路由逻辑。
  - 提供 API、状态机语言仓库。

> **说明**
> 基于 Eventing 层和 ProcessController 层理论，您可以自定义扩展任何流程引擎。

高可用设计



状态机引擎是无状态的，它是内嵌在应用中。如上图上半部分所示，当应用正常运行时：

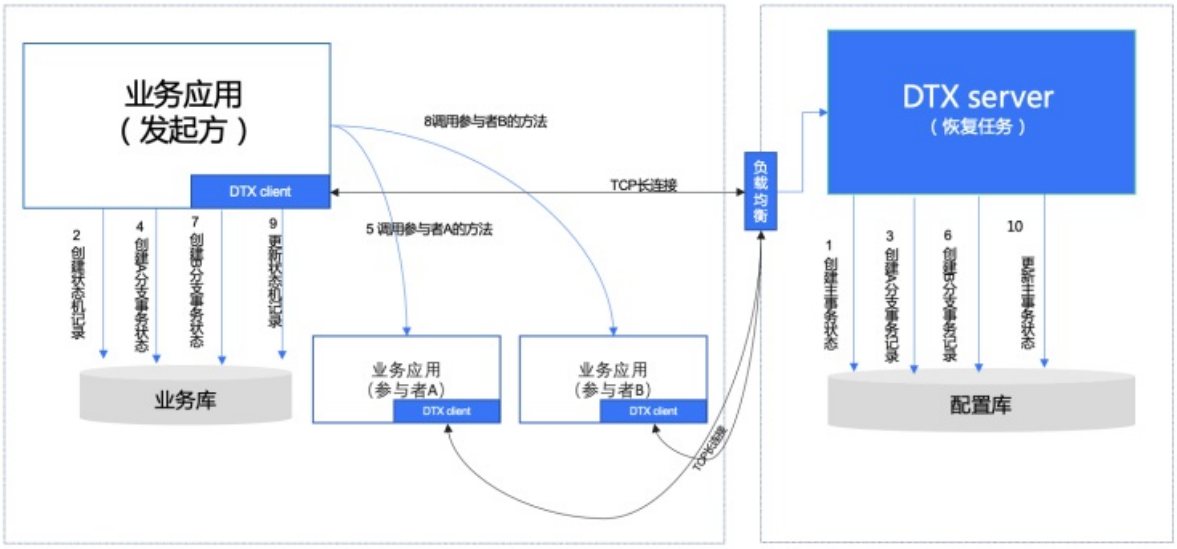
- 状态机引擎会上报状态到 Dtx Server。
- 状态机执行日志存储在业务的数据库中。

如上图下半部分所示，当一台应用实例宕机时：

- Dtx Server 会感知到，并发送事务恢复请求到还存活的应用实例。
- 状态机引擎收到事务恢复请求后，从数据库里装载日志，并恢复状态机上下文继续执行。

5.5.3. Saga 框架处理流程

具体框架和 DTX 服务端以及应用是如何完成交互的可以参考下图：

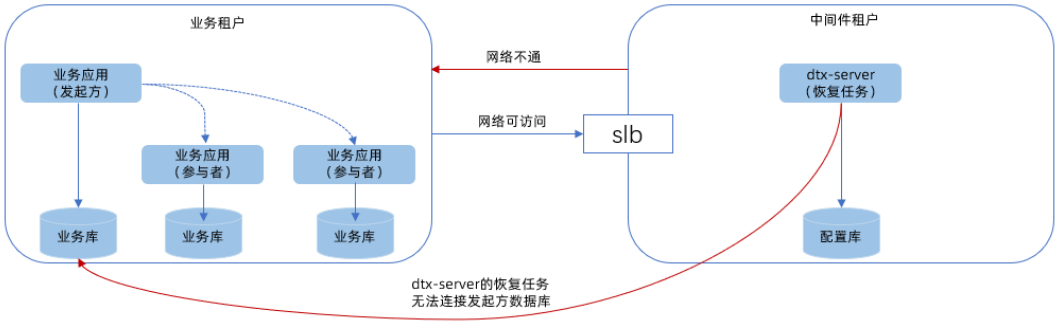


5.6. 同库模式

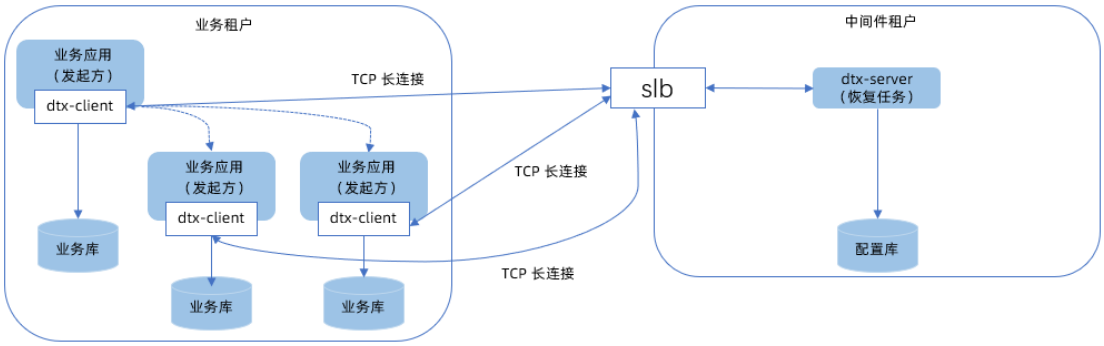
5.6.1. 应用场景

同库模式是指事务数据存储在与业务系统相同的数据库，具有高吞吐、高性能的特点，常用于核心的交易、支付、账务等业务场景。

业务租户和中间件租户属于不同的 VPN 网络。网络环境隔离只允许业务租户访问中间件租户，不允许中间件租户访问业务租户。这种网络络环境就决定了，产品化的同库模式 dtx-server 不能直接连业务数据库（网络不通），业务数据库的访问只能放在业务租户中，由 dtx-client 连业务数据库，dtx-server 负责业务数据的调度和恢复。



业务租户和中间租户的网络连接，只能在业务应用启动的时候，由客户端（业务租户内）发起到服务端 （中间件租户内）TCP 长连接，长连接建立之后，后续客户端和服务端之间的消息通信均可使用此长连接。



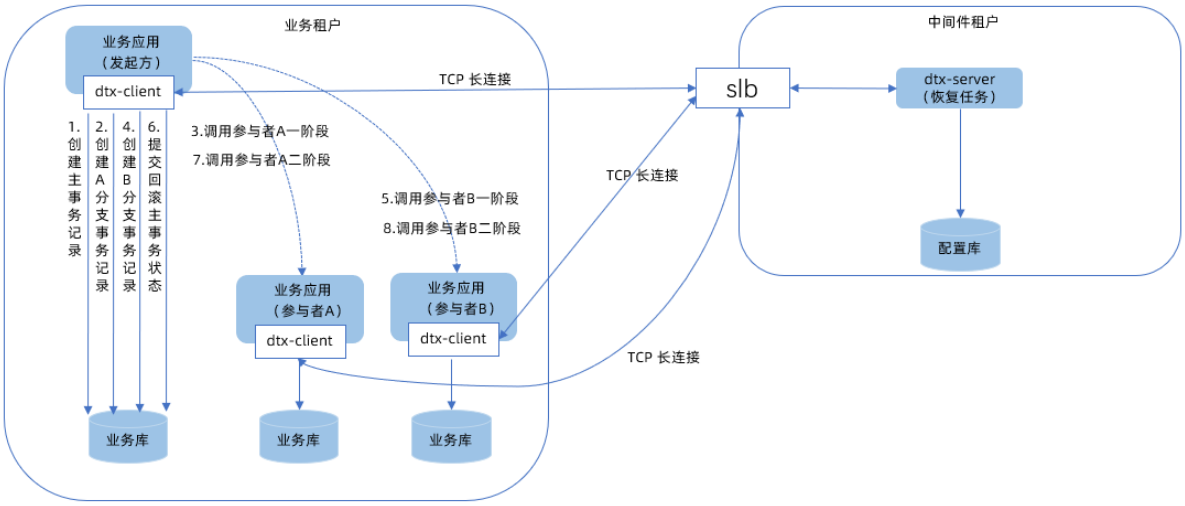
5.6.2. 整体架构设计

在网络环境不通的情况下，产品化的同库模式要使用新的□案。整体设计思路是：发起方数据库由业务应□中的 dtx-client 去访问，dtx-server 负责恢复任务的调度。为了解决嵌套事务场景下（A->B->C 场景），C 应用可能无法确定事务的最终状态的问题，需要在 dtxserver 实现事务状态的回查接口，有 C 应用回查事务状态再执行参与者二阶段。

5.6.3. TCC 同库模式

本节主要介绍 TCC 同库模式的框架处理流程。

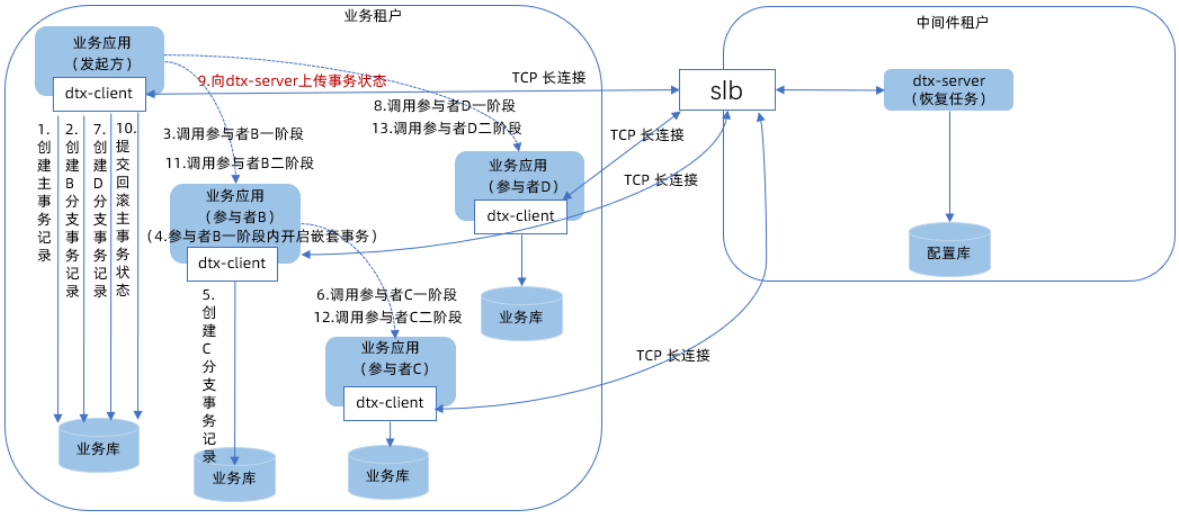
TCC 发起分布式事务



dtx-client 发起分布式事务流程说明如下：

1. 创建主事务记录。
2. 创建 A 分支事务记录。
3. 调用参与者 A 一阶段。
4. 创建 B 分支事务记录。
5. 调用参与者 B 一阶段。
6. 提交回滚主事务状态。
7. 调用参与者 A 二阶段。
8. 调用参与者 B 二阶段。

TCC 发起嵌套事务

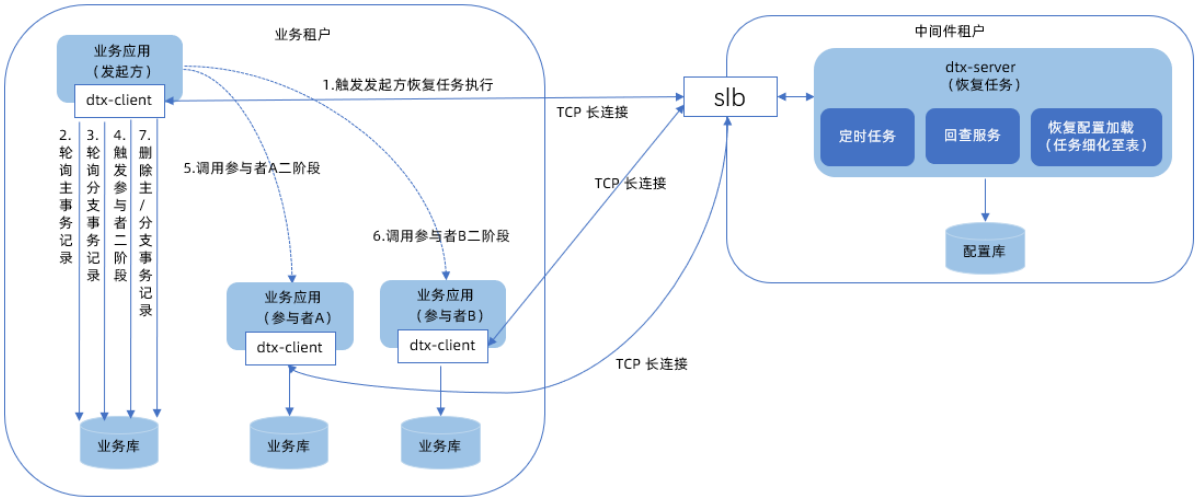


> **重要**
>
> 需要上传事务状态至 dtxserver，供 dtxserver 实现事务状态回查服务。

dtx-client 发起嵌套事务流程说明如下：

1. 创建主事务记录。
2. 创建 B 分支事务记录。
3. 调用参与者 B 一阶段。
4. 参与者 B 一阶段内开启嵌套事务。
5. 创建 C 分支事务记录。
6. 调用参与者 C 一阶段。
7. 创建 D 分支事务记录。
8. 调用参与者 D 一阶段。
9. 向 dtx-server 上传事务状态。
0. 提交回滚主事务状态。
1. 调用参与者 B 二阶段。
2. 调用参与者 C 二阶段。
3. 调用参与者 D 二阶段。

TCC 事务恢复服务



dtx-client 事务恢复服务流程说明如下：

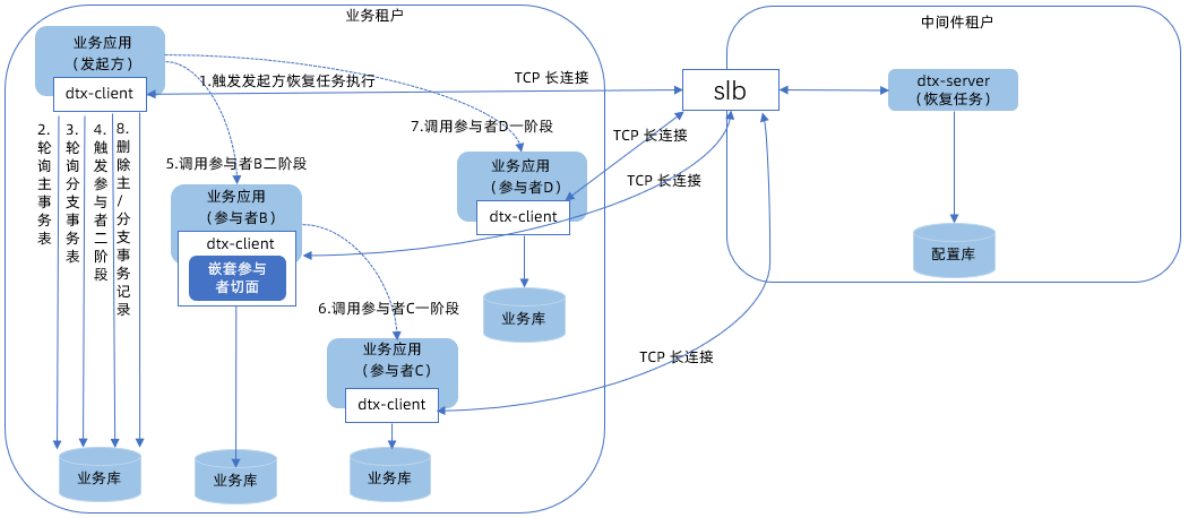
1. dtx-server 触发发起方恢复任务执行。
2. 轮询主事务表。
3. 轮询分支事务表。
4. 触发参与者二阶段。
5. 调用参与者 A 二阶段。
6. 调用参与者 B 二阶段。
7. 删除主或分支事务记录。

TCC 嵌套事务恢复服务

在上面 嵌套事务 场景下，参与者 C 的分支记录在参与者 B 的数据库中，但 dtxserver 只会触发发起方 A 执行恢复任务，A 的数据库无 C 的记录，此时 A 可能无法执行参与者 C 的恢复，需要考虑嵌套事务场景下， 嵌套参与者 C 如何恢复的问题。

TCC 模式嵌套参与者

如果参与者 C 是 TCC 参与者的话，可以在参与者 B 的嵌套事务切面中执行参与者 C 的□阶段方法。



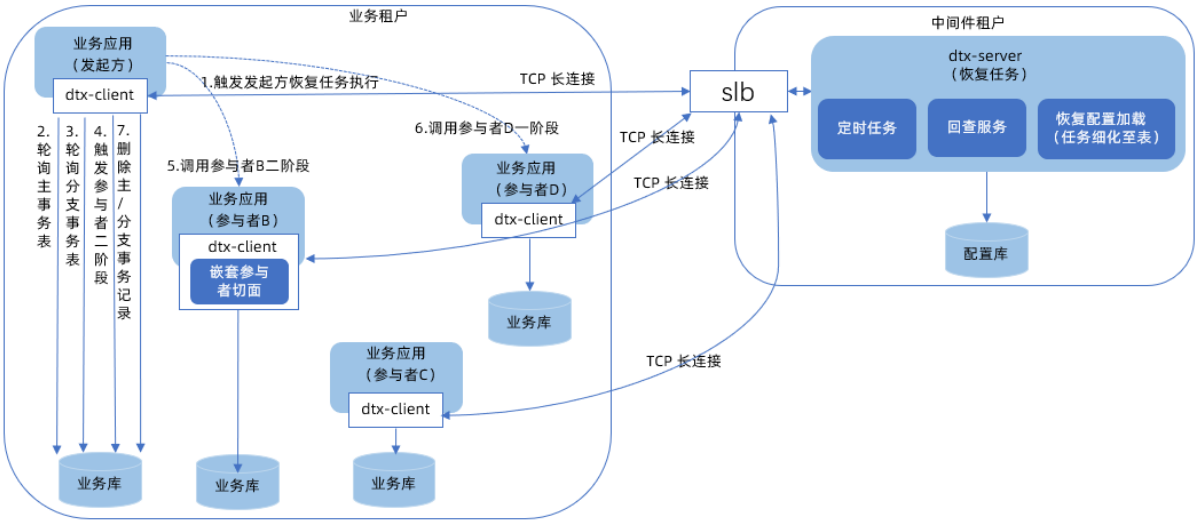
TCC 模式嵌套事务恢复服务流程如下：

1. dtx-server 端的负载均衡 slb，触发发起方恢复任务执行。

2. dtx-client 轮询主事务表。
3. dtx-client 轮询分支事务表。
4. dtx-client 触发参与者二阶段。
5. dtx-client 调用参与者二阶段。
6. dtx-client 调用参与者 C 二阶段。
7. dtx-client 调用参与者 D 一阶段。
8. dtx-client 删除主或分支事务记录。

FMT 嵌套参与者

如果参与者 C 是 FMT 参与者，就不能像 TCC 服务那样可以设置 AOP 切面，在嵌套事务切面里面调用 C 参 与者。为了解决 FMT 嵌套参与者的恢复问题，需要在参与者 C 的客户端内启动轮询任务，定时轮询本地的分支事务记录，再回查 dtxserver 获取事务状态，根据事务状态去提交或者回滚 FMT 参与者。



FMT 模式嵌套事务，发起方 A 的嵌套事务恢复服务流程如下：

1. dtx-server 端的负载均衡 slb，触发发起方恢复任务执行。
2. dtx-client 轮询主事务表。
3. dtx-client 轮询分支事务表。
4. dtx-client 触发参与者二阶段。
5. dtx-client 调用参与者二阶段。
6. dtx-client 调用参与者 D 一阶段。
7. dtx-client 删除主或分支事务记录。

FMT 模式嵌套事务，参与者 C 的嵌套事务恢复服务流程如下：

1. dtx-client 轮询分支事务记录表。
2. dtx-client 向 dtx-server 回查主事务状态。
3. dtx-client 执行 FMT 二阶段。
4. dtx-client 删除分支事务记录。

5.6.4. Saga 同库模式

Saga 模式在客户端有状态机日志表 `state_machine_inst` 和 `state_inst`，这两张表的数据与 `business_activity` 和 `business_action` 对应，所以 Saga 同库模式不在本地写 `business_activity` 和 `business_action` 表，而在 server 触发客户端事务恢复时，查询 `state_machine_inst` 和 `state_inst` 表进行恢复。

实现方法

Client 端状态机执行过程改造点

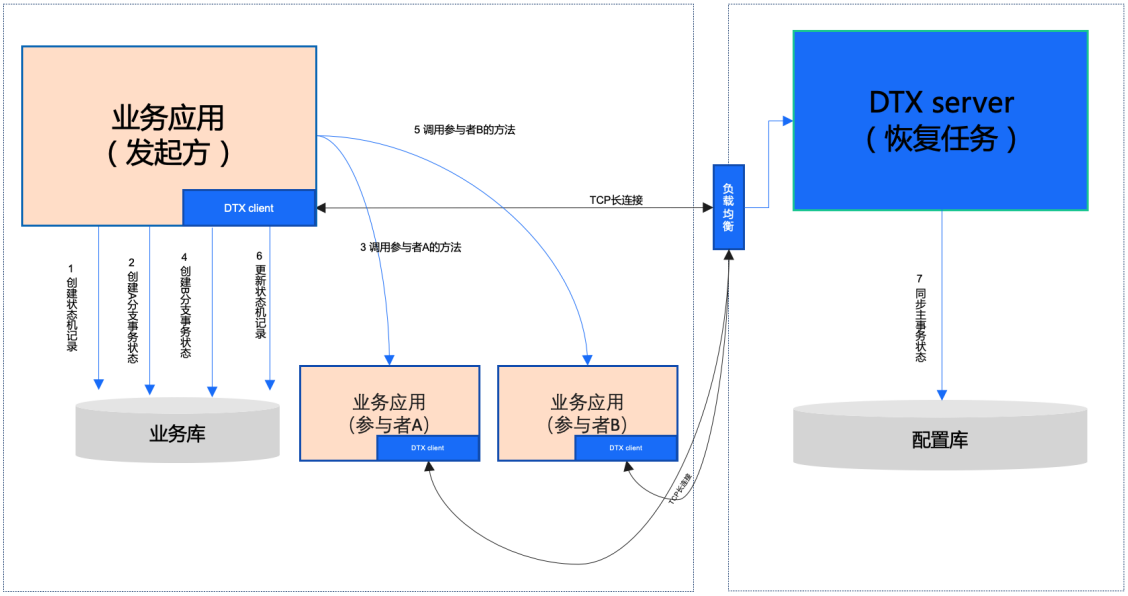
- 启动事务：不调远程服务进行 begin，生成 txId。
- 事务结束：不调远程服务进行上报事务状态，而是调 `DefaultLocalStoreEventHandler.syncLocalRecord` 同步事务记录。

Client 端接收事务恢复逻辑

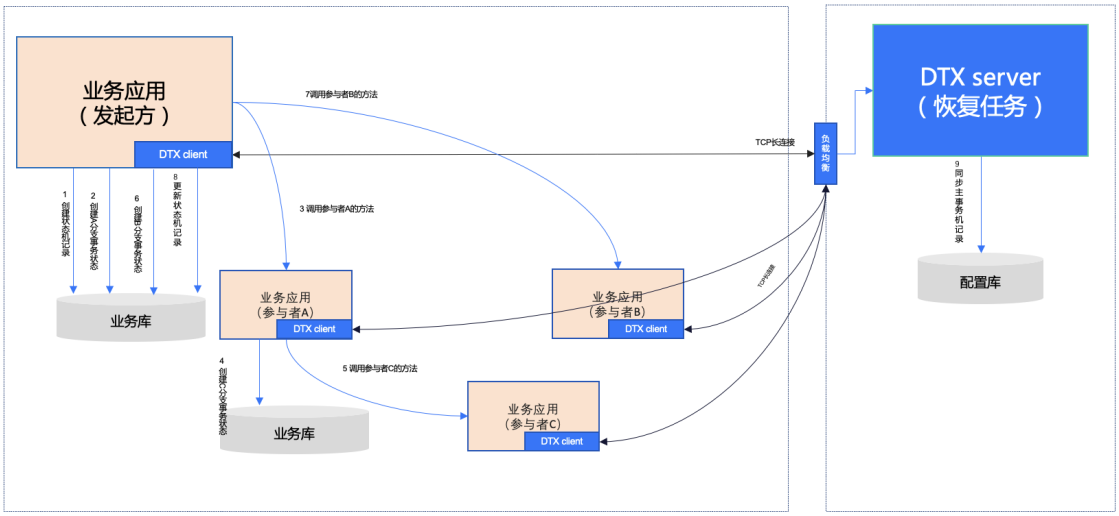
实现 RecoveryManager（比如 SagaRecoveryManager），从表 `state_machine_inst` 和 `state_inst` 查询事务记录，实现 Saga 的事务恢复逻辑。`RecoveryManager.report` 方法实现可以为空，因为在状态机执行完成后自己会上报。在配置任务时在 `DtxRecoveryExcecutor` 中加一个 Saga 恢复任务，RecoveryManager 属性用 `SagaRecoveryManager` 实现。

框架处理流程

Saga 同库发起分布式事务



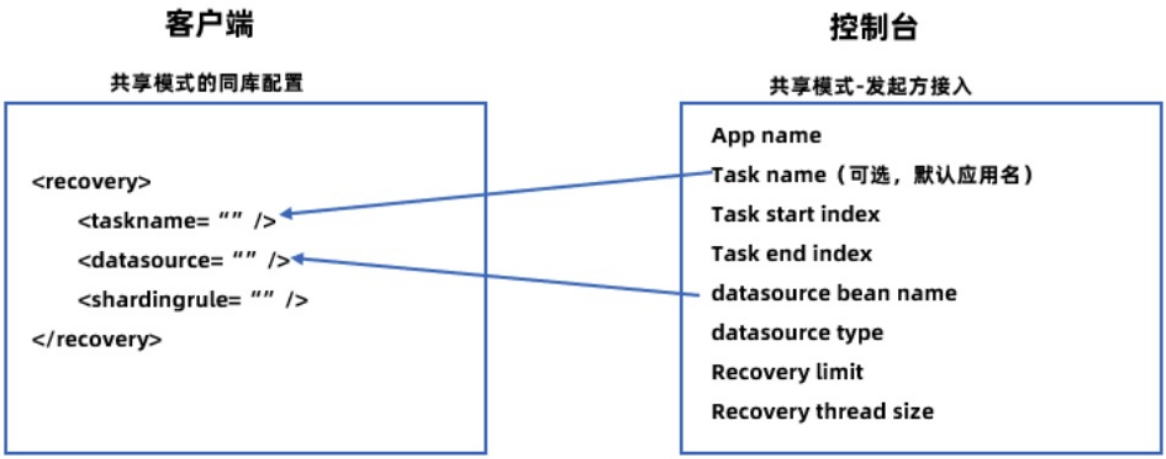
Saga 同库发起嵌套事务



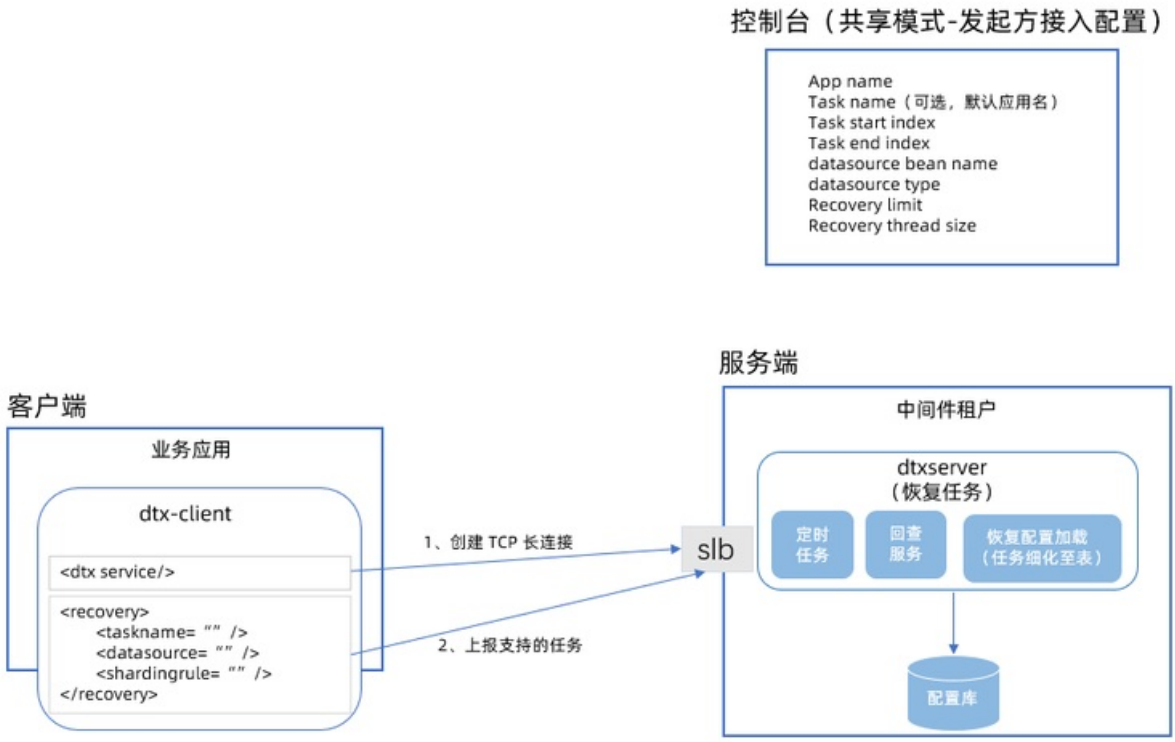
5.6.5. 详细设计

控制台配置

共享模式：由于网络不通，恢复任务只能在客户端执行，所以需要客户端配置恢复任务，主要是配置分库分表规则；dtxserver 负责恢复任务的分布式调度，客户端负责任务的执行。



客户端配置和初始化



客户端配置初始化流程说明如下：

1. dtx-client 创建和 dtx-server 负载均衡 slb 之间的 TCP 长连接。
2. dtx-client 上报支持的任务至 dtx-server 负载均衡 slb。

发起分布式事务流程

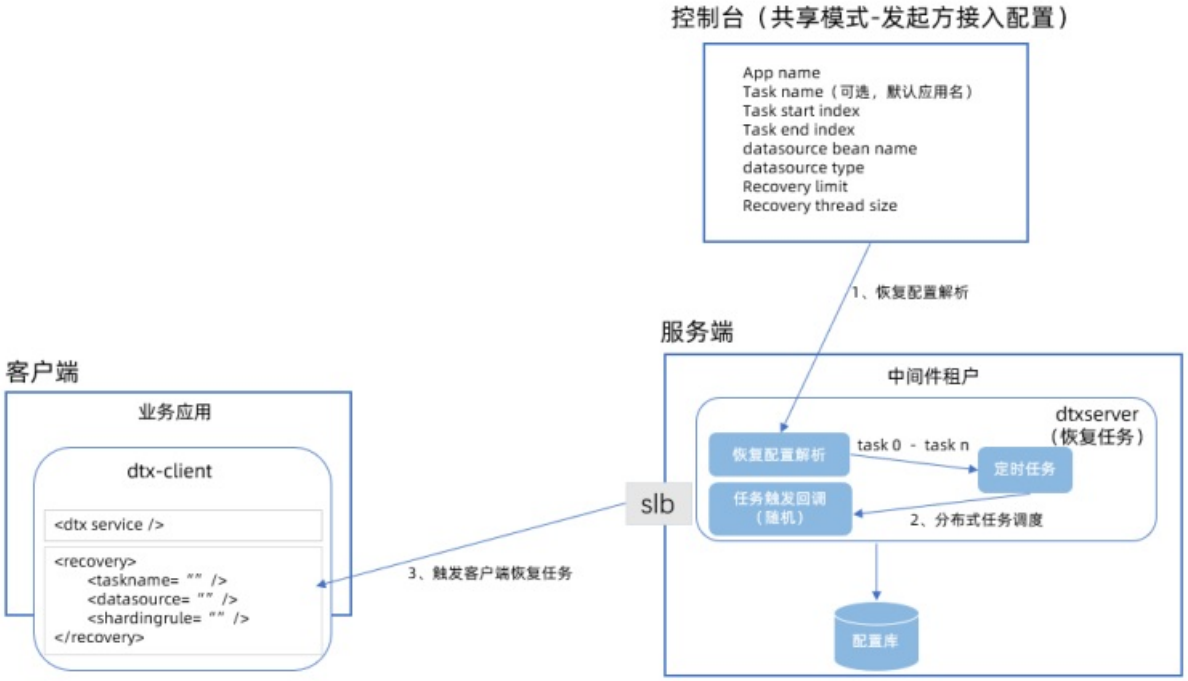
详情请参见 发起分布式事务 。

发起嵌套事务流程

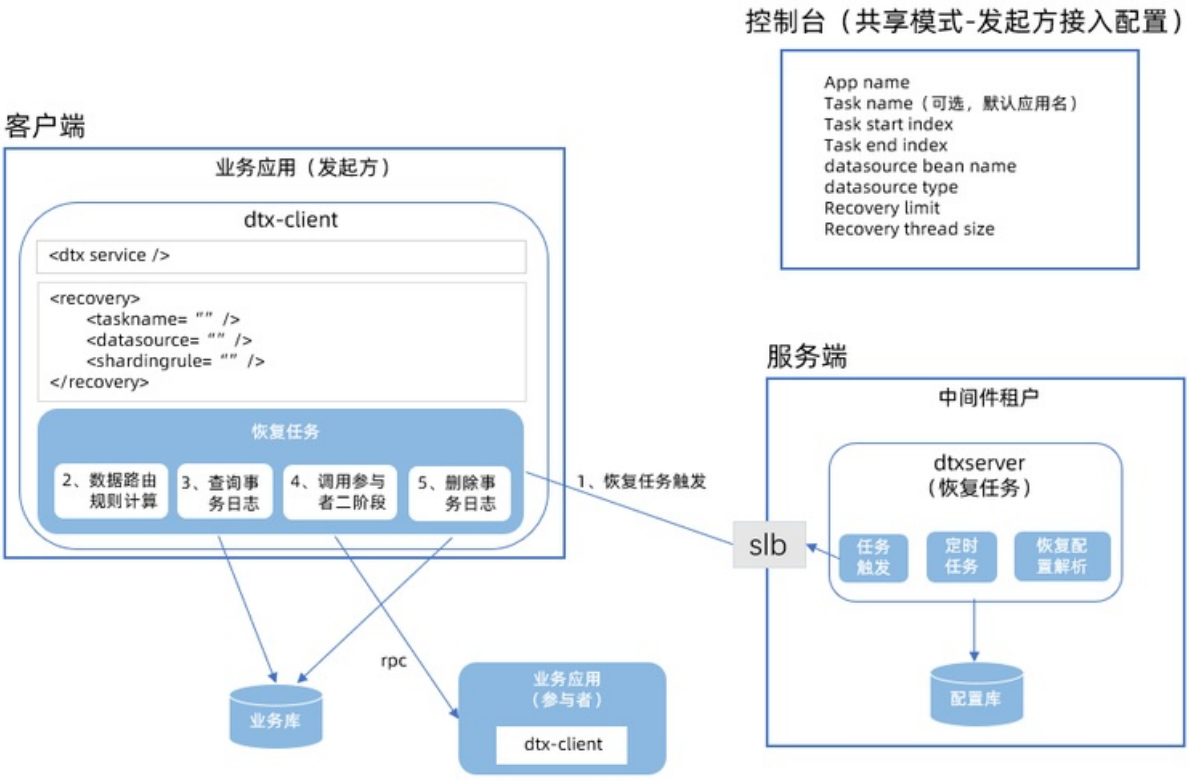
详情请参见 发起嵌套事务 。

恢复任务

服务端恢复任务调度



客户端恢复任务执行



> **说明**
>
> 第 5 步删除是逻辑删除，真正的删除要等事务日志快照同步至 dtxserver 之后执行。

事务日志同步

> **说明**
> dtxconsole 页面需要能查询事务详情。

事务日志同步流程说明如下：

1. dtx-client 后台定时任务，轮询本地业务库中的事务日志。
2. dtx-client 后台定时任务上报事务日志至服务端的负载均衡 slb。
3. 服务端 dtx-server 写入事务日志快照至事务日志。
4. dtx-client 后台定时任务删除事务日志。
5. dtx-server 执行事务统计任务。
6. 控制台查询事务详情。

同库模式客户端配置

- dtx 2.4.0 以上版本，`pom.xml` 配置如下：

```
<dependency>
    <groupId>com.alipay.dtx</groupId>
    <artifactId>dtx-sofaboot</artifactId>
    <version>2.4.0</version>
</dependency>
```

- 发起方应用的 `application.properties` 增加如下配置：

```
com.alipay.dtx.model=LOCAL #同库模式
com.alipay.dtx.local.sync.record=false #同库模式时是否上报事务记录到 server
com.alipay.dtx.report.record.asyn=true #同库模式时是否异步□式上报事务记录到 server
com.alipay.dtx.recovery.expiration.minute=10080 #同库模式时事务恢复超期时间（分）10080 分=7
天
```

- 发起方应用的 Spring 配置同库恢复任务执行器，增加下面的 bean：

```
<bean id="localModelConfigurator" class =
"com.alipay.dtx.client.core.local.LocalModelConfigurator">
     <property name = "dataSource" ref= "transferDataSource" />
     <property name = "dataSourceType" value= "mysql" />
     <property name = "orm" value= "mybatis" />
     <property name = "recoveryExcecutor">
         <bean class= "com.alipay.dtx.recovery.DtxRecoveryExcecutor">
            <property name = "taskName" value= "test_saga_local_task" />
         </bean>
     </property>
 </bean>
```

5.7. 柔性事务

柔性事务的定义

刚性事务（如单数据库）完全遵循 ACID 规范，即数据库事务正确执行的四个基本要素：

- 原子性（Atomicity）
- 一致性（Consistency）

- 隔离性（Isolation）
- 持久性（Durability）

柔性事务（如分布式事务）为了满足可用性、性能与降级服务的需要，降低一致性（Consistency）与隔离性（Isolation）的要求，遵循 BASE 理论：

- 基本业务可用性（Basic Availability）
- 柔性状态（Soft state）
- 最终一致性（Eventual consistency）

同样的，柔性事务也部分遵循 ACID 规范：

- 原子性：严格遵循
- 一致性：事务完成后的一致性严格遵循；事务中的一致性可适当放宽
- 隔离性：并行事务间不可影响；事务中间结果可见性允许安全放宽
- 持久性：严格遵循

柔性事务的分类

柔性事务分为：两阶段型、补偿型、异步确保型、最大努力通知型。

- **两阶段型：**分布式事务二阶段提交，对应技术上的 XA、JTA/JTS，这是分布式环境下事务处理的典型模式。
- **补偿型：**TCC 型事务（Try-Confirm-Cancel）可以归为补偿型。在 Try 成功的情况下，如果事务要回滚，Cancel 将作为一个补偿机制，回滚 Try 操作；TCC 各操作事务本地化，且尽早提交（没有两阶段约束）；当全局事务要求回滚时，通过另一个本地事务实现“补偿”行为。 TCC 是将资源层的二阶段提交协议转换到业务层，成为业务模型中的一部分。
- **异步确保型：**将一些有同步冲突的事务操作变为异步操作，避免对数据库事务的争用，如消息事务机制。
- **最大努力通知型：**通过通知服务器（消息通知）进行，允许失败，有补充机制。

6.API 设计

6.1. 发起方 API

发起方的主要功能是，在事务开启时，通知 DTX 服务端创建主事务日志；在事务结束时，提交或者回滚分布式事务，通知 DTX 服务端修改事务日志状态为待提交或者待回滚。以上发起方的操作，都是在发起方切面中完成，用户只需要在发起方方法上添加注解 `@DtxTransaction`，DTX 框架便能自动扫描到该注解，并添加发起方切面。

发起方示例如下：

```
public class YourClass {
@DtxTransaction(bizType="yourbizType")
public voidyourMethod(yourParams){
try{
//可以是参与者数据源
DAO1;
DAO2;
......
DAO N;
// 调用参与者一阶段try方法，try方法第一个参数值传'null';
TCC.try();
// 普通调用，在该调用方法内可以访问参与者数据源
dataServiceSofaRpc.method();
// 方法正常返回，事务提交
}catch(Throwable t){
// 出现异常，事务回滚
throw t;
}
}
}
```

如上所示，DTX 在设计时，要求用户为需要开启分布式事务的接口增加分布式事务注解 `@DtxTransaction`，表明此方法内部需要开启分布式事务。发起方方法正常返回则分布式事务提交，业务方法抛出异常则分布式事务回滚。

在发起方方法内部，可以执行下列事务操作（无数量与先后顺序的限制）：

- 操作本地 FMT 参与者的 DAO 操作。
- 调用本地 TCC 参与者的第一阶段 Try 方法调用。
- 调用基于 SOFARPC 或 Dubbo 发布的普通服务，该服务内再操作 FMT 参与者，跨服务 FMT 参与者。
- 调用基于 SOFARPC 或 Dubbo 发布的跨服务 TCC 参与者的第一阶段 Try 方法。

6.2. 参与方 API

TCC 参与者

DTX 支持以 TCC 模式接入 SOFARPC 和 Dubbo 远程服务框架。TCC 参与者需要实现 3 个方法，分别是一阶段 Try 方法、二阶段 Confirm 方法以及二阶段 Cancel 方法。在 TCC 参与者的接口中需要先加上 @TwoPhaseBusinessAction 注解，并声明这个 3 个方法，如下所示：

```
public interface TccAction {
@TwoPhaseBusinessAction(name = "yourTccActionName",commitMethod = "confirm", rollbackMe
thod = "cancel")
public boolean try(BusinessActionContext businessActionContext, int a, int b);
public boolean confirm(BusinessActionContext businessActionContext);
public boolean cancel(BusinessActionContext businessActionContext);
}
```

@TwoPhaseBusinessAction 注解属性说明：

- **name**：TCC 参与者的名称，可自定义，但必须全局唯一。
- **commitMethod**：指定二阶段 Confirm 方法的名称，可自定义。
- **rollbackMethod**：指定二阶段 Cancel 方法的名称，可自定义。

方法参数说明：

- **Try**：第一个参数类型必须是 com.alipay.dtx.client.core.api.domain.BusinessActionContext，后续参数的个数和类型可以自定义。
- **Confirm**：有且仅有一个参数，参数类型必须是 com.alipay.dtx.client.core.api.domain.BusinessActionContext，后续为相应的参数名。
- **Cancel**：有且仅有一个参数，参数类型必须是 com.alipay.dtx.client.core.api.domain.BusinessActionContext，后续为相应的参数名。

返回类型说明：

Try、Confirm 和 Cancel 这 3 个方法的返回类型必须为 boolean 类型。

FMT 参与者

FMT 模式参与者只需要替换数据源，即可将对该数据源的访问纳入到分布式事务中。用户可以选择使用自选的数据源，然后使用分布式事务数据源代理（WrappedDtxDataSource）封装该数据源，配置示例如下：

原代码：

```
<!-- 自选的数据源，可选择任何类型
<bean id="xxxDataSource" class="com.xxx.xxx.XXXDataSource">
<!-- 此处为自选数据源的各种参数配置
</bean>
```

修改为：

```
<!-- 自选的数据源，可选择任何类型
<bean id="xxxDataSource_inner" class="com.xxx.xxx.XXXDataSource">
<!-- 此处为自选数据源的各种参数配置
</bean>
<bean id="xxxDataSource" class="com.alipay.sofa.dtx.datasource.WrappedDtxDataSource">
     <property name="targetDataSource" ref="xxxDataSource_inner" />
     <property name="uniqueDbId" value="[数据源全局唯一的
     <property name="dbType" value="[数据源类型]"/>
</bean>
```

业务应用访问数据库时，需要使用代理后的作为 DataSource bean。

分布式事务代理数据源属性介绍：

- **targetDataSource**：指定用户自选数据源 Bean。

- **uniqueDbId**：事务参与方标识，该标识应该是全局唯一的（与TCC 模式注解 name 属性值作用相同）。
- **dbType**：表示数据源类型，支持 MySQL、Oracle 以及 OceanBase。

6.3. Saga 模式 API

Saga 的产品化采用的是 Open Core 模式，即以开源为核心进行产品化，Saga 客户端基于开源的 Seata Saga 为 core，并与 Seata Saga 保持完全兼容的API。详情请参见 SEATA Saga 模式。

商业版采用更为成熟的 dtx server dtx console 进行管理、监控和运维。

Seata 是蚂蚁集团与阿里巴巴集团共同开源的分布式事务产品，详情请参见 Seata。

7.附录：基础术语

事务

事务是指作为单个逻辑工作单元执行的一系列操作，要么完全执行，要么完全不执行。

分布式事务

事务的发起者、资源及资源管理器和事务协调者分别位于不同的分布式系统的不同节点之上。

分支事务

一个分布式事务可能包含多个数据库本地事务，在分布式事务框架下，分支事务可能是一个分库上执行的 SQL 语句，或是一个自定义模式服务的调用。

发起方

分布式事务的发起方负责启动分布式事务，通过调用参与者的服务，将参与者纳入到分布式事务当中，并决定整个分布式事务是提交还是回滚。一个分布式事务有且只能有一个发起方。

参与者

参与者提供分支事务服务。当一个参与者被发起方调用，则被纳入到该发起方启动的分布式事务中，成为该分布式事务的一个分支事务。一个分布式事务可以有多个参与者。

事务管理器

事务管理器是一个独立的服务，用于协调分布式事务，包括创建主事务记录、分支事务记录，并根据分布式事务的状态，调用参与者提交或回滚方法。

主事务记录

又叫 Activity 记录，是整个分布式事务的主体。其最核心的数据结构是事务号（TX_ID）和事务状态（STATE），它是在启动分布式事务时持久化写入数据库的，它的状态决定了这个分布式事务的状态。

分支事务记录

又叫 Action 记录，用于标识分支事务。它记录了该提供该分支事务的参与者的信息，其中包括参与者的唯一标识等。通过分支事务信息，事务管理器就可以对参与者进行提交或回滚操作。