分布式事务

1.概述

1.1 什么是事务

什么是事务？举个生活中的例子：你去商店买东西就是一个事务的例子，买东西是一个交易，包含“一手交钱，一手交货”两个动作，交钱和交货这两个动作必须全部成功，交易才算成功，其中任何一个动作失败，交易就必须撤销。

明白上述例子，再来看事务的定义：

事务可以看做是一次大的活动，它由不同的小活动组成，这些小活动要么全部成功，要么全部失败。

1.2 本地事务

在软件系统中，通常由关系型数据库来控制事务，这是利用数据库本身的事务特性来实现的，因此叫数据库事务，由于应用主要靠关系数据库来控制事务，而数据库通常和应用在同一个服务器，所以基于关系型数据库的事务又被称为本地事务。

回顾一下数据库事务的四大特性 ACID：

A（Atomic）：原子性，构成事务的所有操作，要么都执行成功，要么全部不执行，不可能出现部分成功部分失败的情况。

C（Consistency）：一致性，在事务执行前后，数据库的一致性约束没有被破坏。比如：张三向李四转100元，转账前和转账后的数据是正确状态这叫一致性，如果出现张三转出100元，李四账户没有增加100元这就出现了数据错误，就没有达到一致性。

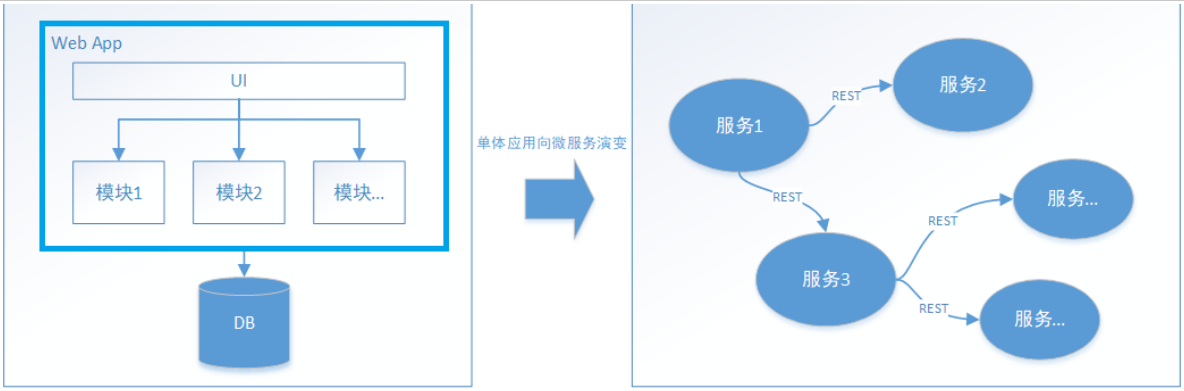
I（Isolation）：隔离性，数据库中的事务一般都是并发的，隔离性是指并发的两个事务的执行互不干扰，一个事务不能看到其他事务运行过程的中间状态。通过配置事务隔离级别可以避脏读、重复读等问题。

D（Durability）：持久性，事务完成之后，该事务对数据的更改会被持久化到数据库，且不会被回滚。

数据库事务在实现时会将一次事务涉及的所有操作全部纳入到一个不可分割的执行单元，该执行单元中的所有操作要么都成功，要么都失败，只要其中任一操作执行失败，都将导致整个事务的回滚。

1.3 分布式事务

随着互联网的快速发展，软件系统由原来的单体应用转变为分布式应用，下图描述了单体应用向分布式微服务应用的演变：



分布式系统会把一个应用系统拆分为可独立部署的多个服务，因此需要服务与服务之间远程协作才能完成事务操作，这种分布式系统环境下的事务机制称之为**分布式事务。**

我们知道本地事务依赖数据库本身提供的事务特性来实现，因此以下逻辑可行：

```
begin transaction;
    -- 1.本地数据库操作：张三减少金额
    -- 2.本地数据库操作：李四增加金额
commit transaction;
```

但是在分布式环境下，可能会变成这样：

```
begin transaction;
    -- 1.A微服务操作本地数据库：张三减少金额
    -- 2.A微服务远程调用B微服务：让李四增加金额
commit transaction;
```

因此在分布式架构的基础上，传统数据库事务就无法使用了，比如上例，张三和李四的账户不在一个数据库中甚至不在一个应用系统里，怎么实现转账事务？也就是说同样一个功能，原来是由一个系统完成的，即使这个功能包含很多个操作，也可以采用数据库事务(本地事务)搞定，而现在这个功能中包含的多个操作可能是由多个系统(微服务)参与完成的，此时数据库事务(本地事务)就无能为力了，这就需要新的分布式事务理论来支撑了。

2.分布式事务基础理论

通过前面的学习，我们了解到了分布式事务的基础概念。与本地事务不同的是，分布式系统之所以叫分布式，是因为提供服务的各个节点分布在不同机器上，相互之间通过网络交互，那么必然存在出现网络故障的风险，这个网络断开的专业场景称之为网络分区，但不能因为这点网络问题就导致整个系统无法提供服务，网络因素成为了分布式事务的考量标准之一。因此，分布式事务需要更进一步的理论支持，接下来，我们先学习一些分布式事务的基础理论，通过理论知识指导我们确定分布式事务控制的目标，从而帮助我们理解每个解决方案。

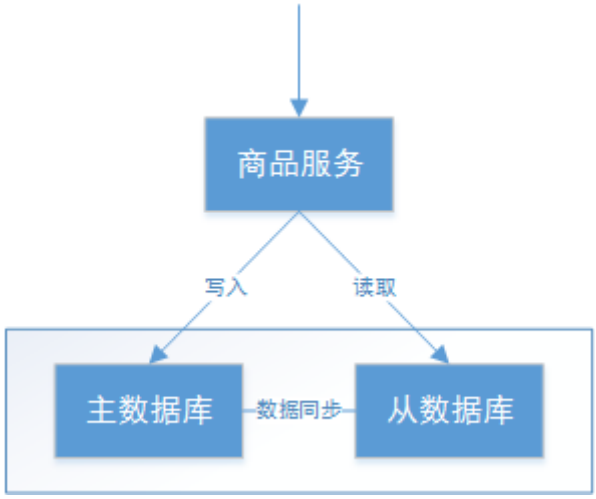
2.1 CAP理论

2.1.1 理解CAP

CAP是 Consistency、Availiability、Partition tolerance三个词语的缩写，分别表示一致性、可用性、分区容忍性。为了方便对CAP理论的理解，我们结合电商平台中的一些业务场景来理解CAP。

业务背景：

- 我们知道每台数据库服务器有他的最大连接数、负载和吞吐量，若有一天无法再满足我们的业务需求，就需要横向去扩展几台 Slave(从数据库) 去分担 Master(主数据库) 的压力。
- 如果服务对数据库的需求是 IO 密集型的，那可能会经常遇到增删改影响到了查询效率。这就需要进行读写分离，由主数据库应付增删改操作，由从数据库应付查询操作，主从数据库的数据要进行同步。



执行流程：

1、商品服务请求主数据库写入商品信息（添加商品、修改商品、删除商品）

2、主数据库向商品服务响应写入成功。

3、商品服务请求从数据库读取商品信息。

C - Consistency：

一致性是指写操作后的读操作可以读取到最新的数据状态，当数据分布在多个节点上，从任意节点读取到的数据都是最新的状态。

上图中，商品信息的读写要满足一致性就是要实现如下目标：

1、商品服务写入主数据库成功，则向从数据库查询新数据也成功。

2、商品服务写入主数据库失败，则向从数据库查询新数据也失败。

A - Availability：

可用性是指任何事务操作都可以得到响应结果，且不会出现响应超时或响应错误。

上图中，商品信息读取满足可用性就是要实现如下目标：

1、从数据库接收到数据查询的请求则立即能够响应数据查询结果。

2、从数据库不允许出现响应超时或响应错误。

为了保证可用性，一般需要通过增加从数据库节点来实现。

P - Partition tolerance：

通常分布式系统的各个节点部署在不同的子网，这就是网络分区，不可避免的会出现由于网络故障而导致节点之间通信失败。分布式系统在遇到某节点或网络分区故障的时候，仍然能够对外提供满足一致性和可用性的服务，这就是分区容忍性。分布式系统中有某一个或者几个机器宕掉了，其他剩下的机器还能够正常运转满足系统需求，或者是机器之间有网络异常，将分布式系统分隔未独立的几个部分，各个部分还能维持分布式系统的运作，这样就具有较好的分区容忍性。

上图中，商品信息读写满足分区容忍性就是要实现如下目标：

1、主数据库向从数据库同步数据失败不影响读写操作。

2、其中一个节点挂掉不影响另一个节点对外提供服务。

2.1.2 CAP组合方式

1、上边的例子是否同时具备CAP呢?

在所有分布式事务场景中不会同时具备CAP三个特性，因为在具备了P的前提下C和A是不能共存的。

在保证分区容忍性的前提下，一致性和可用性无法兼顾，如果要提高系统的可用性就要增加多个节点，如果要保证数据的一致性就要实现每个节点的数据一致，节点越多可用性越好，但是数据一致性会越差。

2、CAP有哪些组合方式呢?

1）AP：

放弃一致性，追求分区容忍性和可用性。这是很多分布式系统设计时的选择。

例如：上边的商品管理，完全可以实现AP，前提是只要用户可以接受所查询的到数据在一定时间内不是最新的即可。通常实现AP都会保证最终一致性，后面讲的BASE理论就是根据AP来扩展的，一些业务场景 比如：订单退款，今日退款成功，明日账户到账，只要用户可以接受在一定时间内到账即可。

2）CP：

放弃可用性，追求一致性和分区容错性，我们的zookeeper其实就是追求的强一致，又比如跨行转账，一次转账请求要等待双方银行系统都完成整个事务才算完成。

3）CA：

放弃分区容忍性，即不进行分区，不考虑由于网络不通或节点挂掉的问题，则可以实现一致性和可用性。那么系统将不是一个标准的分布式系统，我们最常用的关系型数据库就满足了CA。

上边的商品管理，如果要实现CA，则架构如下：

2.1.3 总结

通过上面我们已经学习了CAP理论的相关知识，CAP是一个已经被证实的理论：一个分布式系统最多只能同时满足一致性（Consistency）、可用性（Availability）和分区容忍性（Partition tolerance）这三项中的两项。它可以作为我们进行架构设计、技术选型的考量标准。对于多数大型互联网应用的场景，节点众多、部署分散，而且现在的集群规模越来越大，所以节点故障、网络故障是常态，而且要保证服务可用性达到N个9（99.99..%），并要达到良好的响应性能来提高用户体验，因此一般都会做出如下选择：保证P和A，舍弃C强一致，保证最终一致性。

2.2 BASE理论

1、理解强一致性和最终一致性

CAP理论告诉我们一个分布式系统最多只能同时满足一致性（Consistency）、可用性（Availability）和分区容忍性（Partition tolerance）这三项中的两项，其中AP在实际应用中较多，AP即舍弃一致性，保证可用性和分区容忍性，但是在实际生产中很多场景都要实现一致性，比如前边我们举的例子，主数据库向从数据库同步数据，即使不要一致性，但是最终也要将数据同步成功来保证数据一致，这种一致性和CAP中的一致性不同，CAP中的一致性要求在任何时间查询每个节点数据都必须一致，它强调的是强一致性，但是最终一致性是允许可以在一段时间内每个节点的数据不一致，但是经过一段时间每个节点的数据必须一致，它强调的是最终数据的一致性。

2、Base理论简介

BASE 是 Basically Available(基本可用)、Soft state(软状态)和 Eventually consistent (最终一致性)三个短语的缩写。BASE理论是对CAP中AP的一个扩展，通过牺牲强一致性来获得可用性，当出现故障允许部分不可用但要保证核心功能可用，允许数据在一段时间内是不一致的，但最终达到一致状态。满足BASE理论的事务，我们称之为“**柔性事务**”。

- 基本可用 : 分布式系统在出现故障时，允许损失部分可用功能，保证核心功能可用。如，电商网站交易付款出现问题了，商品依然可以正常浏览。
- 软状态 : 由于不要求强一致性，所以BASE允许系统中存在中间状态（也叫**软状态**），这个状态不影响系统可用性，如订单的"支付中"、“数据同步中”等状态，待数据最终一致后状态改为“成功”状态。
- 最终一致 : 最终一致是指经过一段时间后，所有节点数据都将会达到一致。如订单的"支付中"状态，最终会变为“支付成功”或者"支付失败"，使订单状态与实际交易结果达成一致，但需要一定时间的延迟、等待。

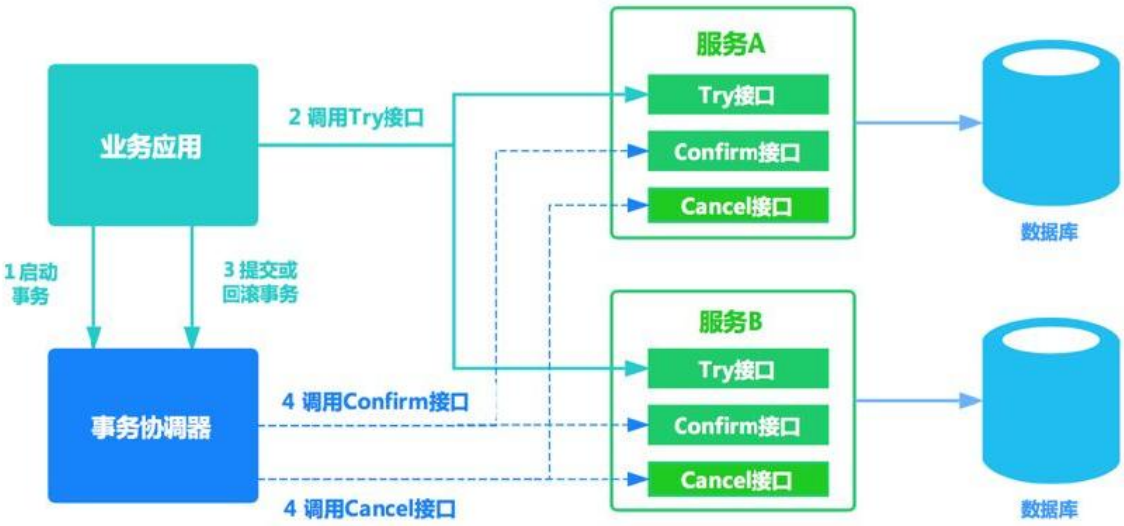
3.解决方案之TCC(补偿事务)

3.1 TCC事务概述

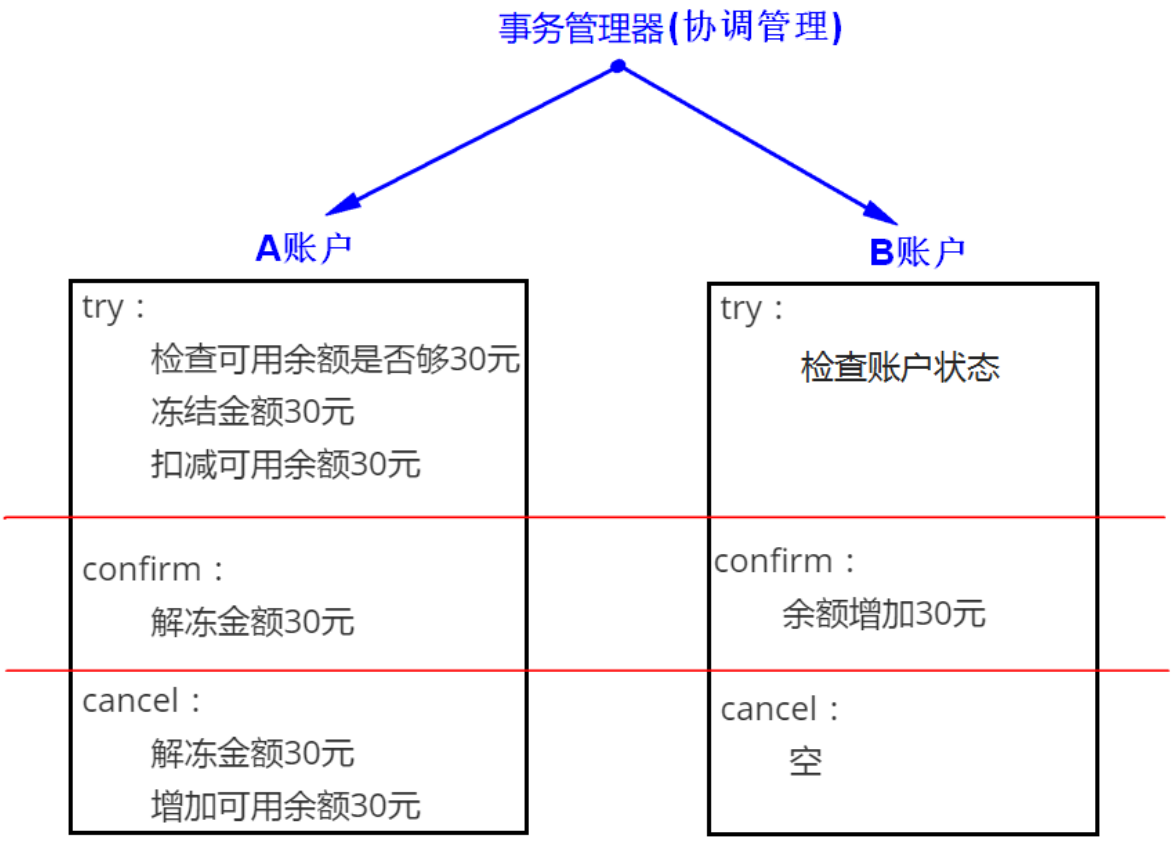
TCC(Try/Confirm/Cancel)编程模式的核心思想是：针对每个分支事务操作，都要向全局事务发起方注册Try、Confirm和Cancel三个操作，具体这些操作由我们自己根据业务进行实现，然后分为两个阶段去执行：

1. **Try** 阶段主要是做业务检查(一致性)及资源预留(隔离)，此阶段仅是一个初步操作，它和后续的 Confirm 一起才能真正构成一个完整的业务逻辑。
2. **Confirm** 阶段主要是做确认提交，Try阶段所有分支事务执行成功后开始执行 Confirm。通常情况下，采用TCC则认为 Confirm阶段是不会出错的。即：只要Try成功，Confirm一定成功。若Confirm阶段真的出错了，需引入重试机制或人工处理。
3. **Cancel** 阶段主要是在业务执行错误，需要回滚的状态下执行分支事务的业务取消，预留资源释放。通常情况下，采用TCC则认为Cancel阶段也是一定成功的。若Cancel阶段真的出错了，需引入

重试机制或人工处理。



案例1： A转账30元给B，A账户和B账户在不同银行(服务)，当前余额都为100元



我们需要把之前实现的转账的代码拆分成三块，套到try-confirm-cancel中，由事务管理器(协调管理)推进AB两个try分别执行，在这个过程中，事务管理器会对AB进行监控，一旦任何一方出现了问题，就推进对方执行cancel；如果双方都没有异常，就推进AB执行confirm。如果在执行confirm或cancel过程中出现问题，就引入重试机制或由人工处理。

TCC解决方案要求每个分支事务实现三个操作Try/Confirm/Cancel。try操作做业务检查及资源预留，Confirm操作做业务确认操作，Cancel操作需要实现一个与try相反的操作。TM(事务管理器)首先发起所有的分支事务的try操作，任何一个分支事务的try操作执行失败，TM将会发起所有分支事务的Cancel操作，若try操作全部成功，TM将会发起所有分支事务的Confirm操作，其中Confirm/Cancel操作若执行

失败，TM会进行重试，因此需要实现幂等。Try/Confirm/Cancel这三个操作的具体实现，由开发者根据业务情况灵活掌握。

TCC不足之处：

- 对应用的侵入性强。业务逻辑的每个分支都需要实现try、confirm、cancel三个操作，应用侵入性较强，改造成本高。
- 实现难度较大。需要按照网络状态、系统故障等不同的失败原因实现不同的回滚策略。为了满足一致性的要求，confirm和cancel接口必须实现幂等。

3.2 TCC框架

目前市面上的TCC框架众多比如下面这几种：

框架名称	Gitbub地址
tcc-transaction	https://github.com/changmingxie/tcc-transaction
Hmily	https://github.com/yu199195/hmily
ByteTCC	https://github.com/liuyangming/ByteTCC
EasyTransaction	https://github.com/QNJR-GROUP/EasyTransaction

Hmily是一个高性能分布式事务tcc开源框架。基于java语言来开发（JDK1.8），支持dubbo，springcloud，motan等rpc框架进行分布式事务。它目前支持以下特性：

- 支持嵌套事务(Nested transaction support)。
- 采用disruptor框架进行事务日志的异步读写，与RPC框架的性能毫无差别。
- 支持SpringBoot-starter 项目启动，使用简单。
- RPC框架支持 : dubbo,motan,springcloud。
- 本地事务存储支持 : redis,mongodb,zookeeper,file,mysql。
- 事务日志序列化支持： java，hessian，kryo，protostuff。
- 采用Aspect AOP 切面思想与Spring无缝集成，天然支持集群。
- RPC事务恢复，超时异常恢复等。

Hmily利用AOP对参与分布式事务的本地方法与远程方法进行拦截处理，通过多方拦截，事务参与者能透明的调用到另一方的Try、Conform、Cancel方法；传递事务上下文；并记录事务日志，酌情进行补偿，重试等。

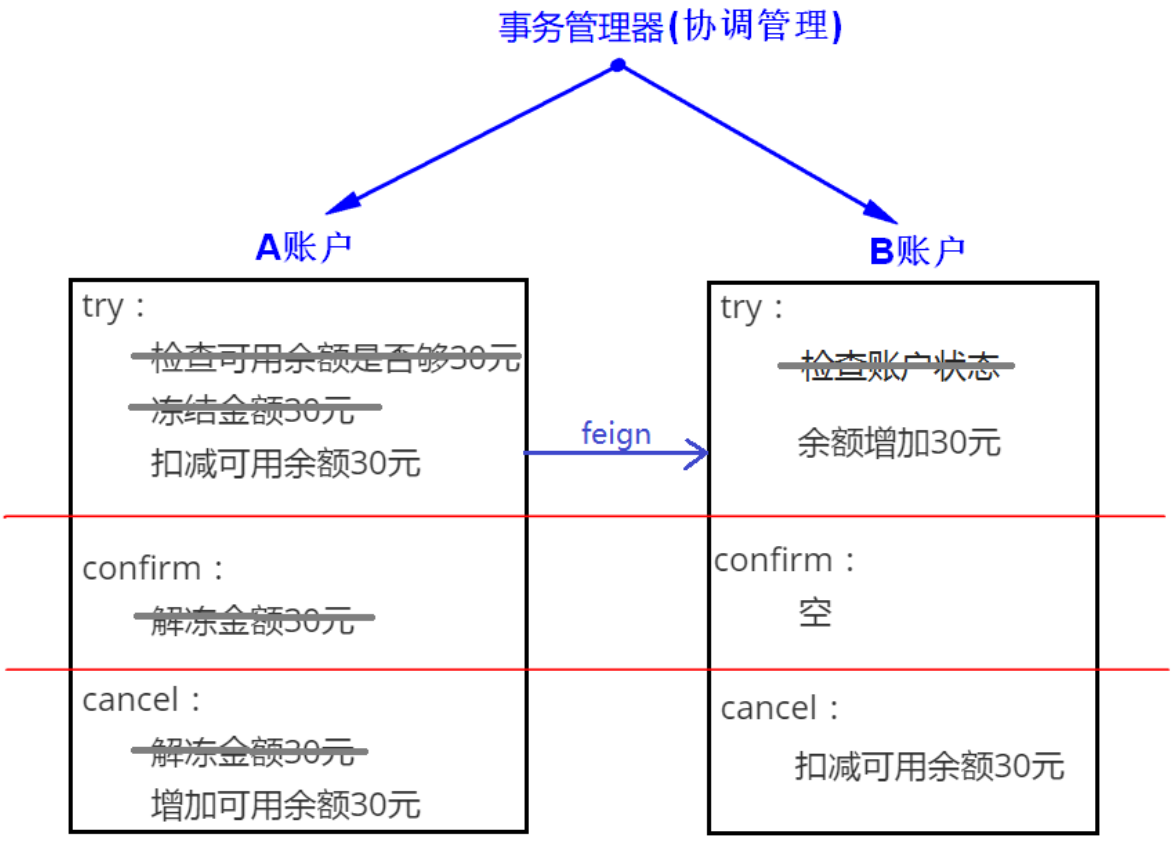
Hmily不需要事务协调服务，但需要提供一个数据库(mysql/mongodb/zookeeper/redis/file)来进行日志存储。Hmily实现的TCC服务与普通的服务一样，只需要暴露一个接口，也就是它的Try业务。Confirm/Cancel业务逻辑，只是因为全局事务提交/回滚的需要才提供的，因此Confirm/Cancel业务只需要被Hmily事务框架发现即可，不需要被调用它的其他业务服务所感知。

官网介绍：https://dromara.org/website/zh-cn/docs/hmily/index.html

3.3 Hmily快速入门

3.3.1 业务说明

本案例通过hmily框架实现TCC分布式事务，模拟两个账户的转账交易过程。两个账户分别在不同的银行(张三在bank1、李四在bank2)，bank1、bank2是两个微服务。对于交易过程中的每个操作，要么都成功，要么都失败。



3.3.2 环境搭建

3.3.2.1 环境要求

数据库：MySQL 5.7.25+

JDK： jdk1.8+

微服务：spring-boot-2.1.3、spring-cloud-Greenwich.RELEASE

hmily：hmily-springcloud.2.0.4-RELEASE

3.3.2.2 数据库

创建bank1库，并导入以下表结构和数据：

```
CREATE DATABASE `bank1` CHARACTER SET 'utf8' COLLATE 'utf8_general_ci';
```

```
USE bank1;
DROP TABLE IF EXISTS `account_info`;
CREATE TABLE `account_info`  (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `account_name` varchar(100) CHARACTER SET utf8 COLLATE utf8_bin NULL DEFAULT
NULL COMMENT '户主姓名',
  `account_no` varchar(100) CHARACTER SET utf8 COLLATE utf8_bin NULL DEFAULT
NULL COMMENT '银行卡号',
  `account_password` varchar(100) CHARACTER SET utf8 COLLATE utf8_bin NULL
DEFAULT NULL COMMENT '帐户密码',
  `account_balance` double NULL DEFAULT NULL COMMENT '帐户余额',
  PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB AUTO_INCREMENT = 5 CHARACTER SET = utf8 COLLATE = utf8_bin
ROW_FORMAT = Dynamic;
INSERT INTO `account_info` VALUES (1, '张三', '1', '', 10000);
```

创建bank2库，并导入以下表结构和数据:

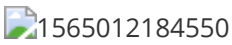
```
CREATE DATABASE `bank2` CHARACTER SET 'utf8' COLLATE 'utf8_general_ci';
```

```
USE bank2;
DROP TABLE IF EXISTS `account_info`;
CREATE TABLE `account_info`  (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `account_name` varchar(100) CHARACTER SET utf8 COLLATE utf8_bin NULL DEFAULT
NULL COMMENT '户主姓名',
  `account_no` varchar(100) CHARACTER SET utf8 COLLATE utf8_bin NULL DEFAULT
NULL COMMENT '银行卡号',
  `account_password` varchar(100) CHARACTER SET utf8 COLLATE utf8_bin NULL
DEFAULT NULL COMMENT '帐户密码',
  `account_balance` double NULL DEFAULT NULL COMMENT '帐户余额',
  PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB AUTO_INCREMENT = 5 CHARACTER SET = utf8 COLLATE = utf8_bin
ROW_FORMAT = Dynamic;
INSERT INTO `account_info` VALUES (2, '李四', '2', NULL, 0);
```

Hmily用来存储日志的数据表由它自动创建

3.3.2.3 Maven工程

（1）搭建maven工程，以tcc-hmily-demo为父工程，进行依赖管理，hmily-demo-bank1负责张三账户操作，hmily-demo-bank2负责李四账户操作，hmily-demo-discover-server是服务注册中心。



（2）pom.xml导入hmily相关依赖，这里不再详述

3.3.4 功能实现

3.3.4.1 hmily-demo-bank1工程

（1）application.yml配置（只显示hmily部分）

```
org:
  dromara:
    hmily :
      serializer : kryo #序列化工具
      retryMax : 30 #最大重试次数
      repositorySupport : db #持久化方式
      started: true #事务发起方
      hmilyDbConfig :
        driverClassName  : com.mysql.jdbc.Driver
        url :  jdbc:mysql://localhost:3306/bank1?useUnicode=true
        username : root
        password : 123
```

(2) Hmily配置类

```
@Configuration
@EnableAspectJAutoProxy(proxyTargetClass=true)
public class DatabaseConfiguration {

    @Autowired
    private Environment env;

    @Bean
    public HmilyTransactionBootstrap hmilyTransactionBootstrap(HmilyInitService
hmilyInitService){
        HmilyTransactionBootstrap hmilyTransactionBootstrap = new
HmilyTransactionBootstrap(hmilyInitService);

 hmilyTransactionBootstrap.setSerializer(env.getProperty("org.dromara.hmily.seri
alizer"));

 hmilyTransactionBootstrap.setRetryMax(Integer.parseInt(env.getProperty("org.dro
mara.hmily.retryMax")));

 hmilyTransactionBootstrap.setRepositorySupport(env.getProperty("org.dromara.hmi
ly.repositorySupport"));

 hmilyTransactionBootstrap.setStarted(Boolean.parseBoolean(env.getProperty("org.
dromara.hmily.started")));
        HmilyDbConfig hmilyDbConfig = new HmilyDbConfig();

 hmilyDbConfig.setDriverClassName(env.getProperty("org.dromara.hmily.hmilyDbConf
ig.driverClassName"));

 hmilyDbConfig.setUrl(env.getProperty("org.dromara.hmily.hmilyDbConfig.url"));

 hmilyDbConfig.setUsername(env.getProperty("org.dromara.hmily.hmilyDbConfig.user
name"));

 hmilyDbConfig.setPassword(env.getProperty("org.dromara.hmily.hmilyDbConfig.pass
word"));
        hmilyTransactionBootstrap.setHmilyDbConfig(hmilyDbConfig);
        return hmilyTransactionBootstrap;
    }
}
```

（3）feign代理

```
@FeignClient(value = "hmily-demo-bank2")
public interface Bank2Client {
    @GetMapping("/bank2/transfer")
    @Hmily
    Boolean transfer(@RequestParam("amount") Double amount);
}
```

（4）转账业务

```
@Service
public class AccountInfoTccImpl implements AccountInfoTcc {

    @Autowired
    private AccountInfoDao accountInfoDao;

    @Autowired
    private Bank2Client bank2Client;

    @Override
    @Hmily(confirmMethod = "commit", cancelMethod = "rollback")
    public void prepare( String accountNo, double amount) {
        System.out.println("...Bank1 Service  prepare..." );
        if(!bank2Client.transfer(amount)){
            throw new RuntimeException("bank2 exception");
        }
    }

    @Override
    public void commit( String accountNo, double amount) {
        System.out.println("...Bank1 Service commit..." );
    }

    @Override
    public void rollback( String accountNo, double amount) {
        accountInfoDao.updateAccountBalance(accountNo ,amount );
        System.out.println("...Bank1 Service rollback..." );
    }
}
```

注意：Try、Confirm、cancel的方法参数必须保持一致。

（5）启动类

```
@SpringBootApplication(exclude = MongoAutoConfiguration.class)
@EnableDiscoveryClient
@EnableFeignClients(basePackages =
{"cn.itcast.wanxintx.hmilydemo.bank1.feignClient"})
@ComponentScan({"cn.itcast.wanxintx.hmilydemo.bank1","org.dromara.hmily"})
public class Bank1HmilyServer {
   public static void main(String[] args) {
      SpringApplication.run(Bank1HmilyServer.class, args);
   }
}
```

3.3.4.2 hmily-demo-bank2工程

（1）application.yml配置（只显示hmily部分）

```yaml
org:
  dromara:
    hmily :
      serializer : kryo #序列化工具
      retryMax : 30 #最大重试次数
      repositorySupport : db #持久化方式
      started: false #事务参与方
      hmilyDbConfig :
        driverClassName  : com.mysql.jdbc.Driver
        url :  jdbc:mysql://localhost:3306/bank2?useUnicode=true
        username : root
        password : 123
```

（2）Hmily配置类，和hmily-demo-bank1一样

（3）转账业务

```java
@Service
public class AccountInfoServiceImpl implements AccountInfoService {
   @Autowired
   private AccountInfoDao accountInfoDao;

   @Override
   @Hmily(confirmMethod = "confirmMethod", cancelMethod = "cancelMethod")
   public Boolean updateAccountBalance(String accountNo, Double amount) {
      System.out.println("...Bank2 Service Begin ...");
      try{
         accountInfoDao.updateAccountBalance(accountNo ,amount);
      }catch(Exception e){
         e.printStackTrace();
         throw new RuntimeException( e.getMessage() );
      }
      return true;
   }
   public Boolean confirmMethod(String accountNo, Double amount) {
      System.out.println("...Bank2 Service commit..." );
      return true;
   }

   public Boolean cancelMethod(String accountNo, Double amount) {
      accountInfoDao.updateAccountBalance(accountNo ,amount * -1);
      System.out.println("...Bank2 Service rollback..." );
      return true;
   }
}
```

注意：Try、Confirm、cancel的方法参数必须保持一致。

（4）启动类

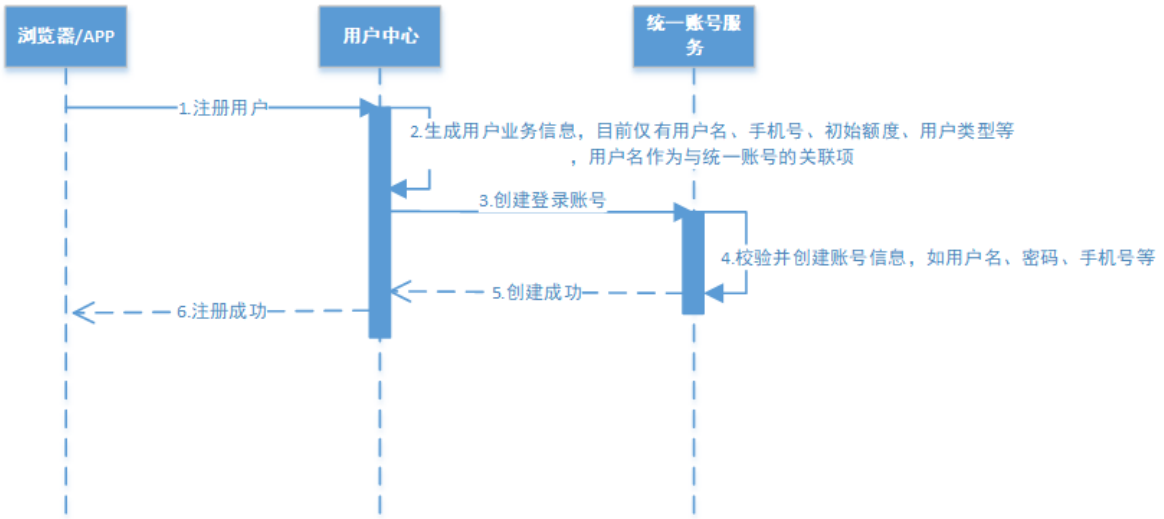
```
@SpringBootApplication(exclude = MongoAutoConfiguration.class)
@EnableDiscoveryClient
@ComponentScan({"cn.itcast.wanxintx.hmilydemo.bank2","org.dromara.hmily"})
public class Bank2HmilyServer {
   public static void main(String[] args) {
      SpringApplication.run(Bank2HmilyServer.class, args);
   }
}
```

3.3.5 功能测试

- Bank1（张三）和Bank2（李四）都执行成功。
- Bank2（李四）执行失败，Bank1(张三)事务回滚。

4.Hmily解决注册功能的事务问题

4.1 业务回顾



用户向用户中心发起注册请求，用户中心保存用户业务信息，然后远程调用统一账号服务保存该用户所对应的账号信息，该业务存在分布式事务问题。

针对注册业务，如果用户与账号信息不一致，则会导致严重问题，因此该业务对一致性要求较为严格，且属于执行时间较短的业务。TCC方案的软状态时间很短，一致性较强，因此在此业务，我们选用TCC型分布式事务解决方案。

4.2 Hmily环境搭建

1. 数据库环境

新建数据库p2p_undo_log，此库用来存储hmily事务日志，空库即可，由hmily自动建表。

```
CREATE DATABASE `p2p_undo_log` CHARACTER SET 'utf8' COLLATE 'utf8_general_ci';
```

2. Maven环境

```
<dependency>
```

```
    <groupId>org.dromara</groupId>
    <artifactId>hmily-springcloud</artifactId>
    <version>2.0.5-RELEASE</version>
    <exclusions>
        <exclusion>
            <artifactId>spring-boot-starter</artifactId>
            <groupId>org.springframework.boot</groupId>
        </exclusion>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
        </exclusion>
        <exclusion>
            <artifactId>logback-core</artifactId>
            <groupId>ch.qos.logback</groupId>
        </exclusion>
        <exclusion>
            <artifactId>slf4j-api</artifactId>
            <groupId>org.slf4j</groupId>
        </exclusion>
        <exclusion>
            <artifactId>logback-classic</artifactId>
            <groupId>ch.qos.logback</groupId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>com.esotericsoftware</groupId>
    <artifactId>kryo</artifactId>
    <version>4.0.2</version>
</dependency>
<dependency>
    <groupId>org.springframework.retry</groupId>
    <artifactId>spring-retry</artifactId>
</dependency>
```

4.3 功能实现

4.3.1 事务发起方：consumer-service

(1) 在Apollo中，为common-template项目新建公共namespace，命名为micro_service.spring-cloud-hmily，并把下面的代码复制进去

```
org.dromara.hmily.serializer = kryo
org.dromara.hmily.retryMax = 30
org.dromara.hmily.repositorySupport = db
org.dromara.hmily.started = true
org.dromara.hmily.hmilyDbConfig.driverClassName = com.mysql.cj.jdbc.Driver
org.dromara.hmily.hmilyDbConfig.url = jdbc:mysql://localhost:3306/p2p_undo_log?
useUnicode=true
org.dromara.hmily.hmilyDbConfig.username = root
org.dromara.hmily.hmilyDbConfig.password = 123
```

在consumer-service项目中，关联刚才新建的micro_service.spring-cloud-hmily

(2) 在application.yml中增加micro_service.spring-cloud-hmily

(3) 在config包中新增Hmily配置类

```
@Configuration
@EnableAspectJAutoProxy(proxyTargetClass=true)
public class HmilyConfig {

    @Autowired
    private Environment env;

    @Bean
    public HmilyTransactionBootstrap hmilyTransactionBootstrap(HmilyInitService
hmilyInitService){
        HmilyTransactionBootstrap hmilyTransactionBootstrap = new
HmilyTransactionBootstrap(hmilyInitService);

 hmilyTransactionBootstrap.setSerializer(env.getProperty("org.dromara.hmily.seri
alizer"));

 hmilyTransactionBootstrap.setRetryMax(Integer.parseInt(env.getProperty("org.dro
mara.hmily.retryMax")));

 hmilyTransactionBootstrap.setRepositorySupport(env.getProperty("org.dromara.hmi
ly.repositorySupport"));

 hmilyTransactionBootstrap.setStarted(Boolean.parseBoolean(env.getProperty("org.
dromara.hmily.started")));
        HmilyDbConfig hmilyDbConfig = new HmilyDbConfig();

 hmilyDbConfig.setDriverClassName(env.getProperty("org.dromara.hmily.hmilyDbConf
ig.driverClassName"));

 hmilyDbConfig.setUrl(env.getProperty("org.dromara.hmily.hmilyDbConfig.url"));

 hmilyDbConfig.setUsername(env.getProperty("org.dromara.hmily.hmilyDbConfig.user
name"));

 hmilyDbConfig.setPassword(env.getProperty("org.dromara.hmily.hmilyDbConfig.pass
word"));
        hmilyTransactionBootstrap.setHmilyDbConfig(hmilyDbConfig);
        return hmilyTransactionBootstrap;
    }
}
```

(4) 启动类上增加org.dromara.hmily的扫描项：

```
@SpringBootApplication(scanBasePackages = {"org.dromara.hmily",
"cn.itcast.wanxinp2p.consumer"})
```

(5) Feign代理(AccountApiAgent接口)中增加@Hmily注解

```
@FeignClient(value="account-service")
public interface AccountApiAgent {
   @PostMapping(value = "/account/1/accounts")
   @Hmily
   RestResponse<AccountDTO> register(@RequestBody AccountRegisterDTO
accountRegisterDTO);
}
```

(6) 修改ConsumerServiceImpl代码，注册Try、Confirm、Cancel

```
@Override
@Hmily(confirmMethod = "confirmRegister", cancelMethod = "cancelRegister")
public void register(ConsumerRegisterDTO consumerRegisterDTO) {
     ... ...
}
public void confirmRegister(ConsumerRegisterDTO consumerRegisterDTO) {
    log.info("execute confirmRegister");
}
public void cancelRegister(ConsumerRegisterDTO consumerRegisterDTO) {
    log.info("execute cancelRegister");
    remove(Wrappers.<Consumer>lambdaQuery().eq(Consumer::getMobile,

consumerRegisterDTO.getMobile()));
}
```

4.3.2 事务参与方：account-service

(1) 在Apollo中，为account-service项目关联micro_service.spring-cloud-hmily名称空间，并修改org.dromara.hmily.started=false。

(2) 在application.yml中增加micro_service.spring-cloud-hmily

(3) 在config包中新增Hmily配置类，同consumer-service一样

(4) 启动类上增加org.dromara.hmily的扫描项

```
@SpringBootApplication(scanBasePackages =
{"org.dromara.hmily","cn.itcast.wanxinp2p.account"})
```

(5) 修改AccountServiceImpl代码，注册Try、Confirm、Cancel

```
@Override
@Hmily(confirmMethod = "confirmRegister", cancelMethod = "cancelRegister")
public AccountDTO register(AccountRegisterDTO registerDTO) {
     ... ...
}

public void confirmRegister(AccountRegisterDTO registerDTO) {
    log.info("execute confirmRegister");
}

public void cancelRegister(AccountRegisterDTO registerDTO) {
    log.info("execute cancelRegister");
     //删除账号
```

```
    remove(Wrappers.<Account>lambdaQuery().eq(Account::getUsername,
                                              registerDTO.getUsername()));
}
```

4.4 功能测试

- 用户中心服务和统一账户服务都执行成功。
- 统一账户服务执行失败，用户中心服务回滚成功。

5.解决方案之可靠消息实现最终一致性

前面在学习CAP和BASE理论时，我们得出结论：一般情况下会保证P和A，舍弃C，保证最终一致性。最终一致是指经过一段时间后，所有节点数据都将会达到一致。如订单的"支付中"状态，最终会变为“支付成功”或者"支付失败"，使订单状态与实际交易结果达成一致，但需要一定时间的延迟、等待。

5.1 概述

此方案的核心是将分布式事务拆分成多个本地事务，然后通过网络由消息队列协调完成所有事务，并实现最终一致性。以转账为例：

1. 消息发送方张三，即Bank1：

   扣减余额30元，然后通过网络发送消息到MQ

2. 消息接收方李四，即Bank2：

   通过网络从MQ中接收消息，然后增加余额30元

该解决方案容易理解，实现成本低，但是面临以下几个问题：

1.消息发送方执行本地事务与发送消息的原子性问题，也就是说如何保证本地事务执行成功，消息一定发送成功

```
begin transaction
    1.数据库操作
    2.发送消息
commit transation
```

这种情况下，貌似没有问题，如果发送消息失败，就会抛出异常，导致数据库事务回滚。但如果是超时异常，数据库回滚，但此时消息已经正常发送了，同样会导致不一致。

2.消息接收方接收消息与本地事务的原子性问题，也就是说如何保证接收消息成功后，本地事务一定执行成功

3.由于消息可能会重复发送，这就要求消息接收方必须实现幂等性

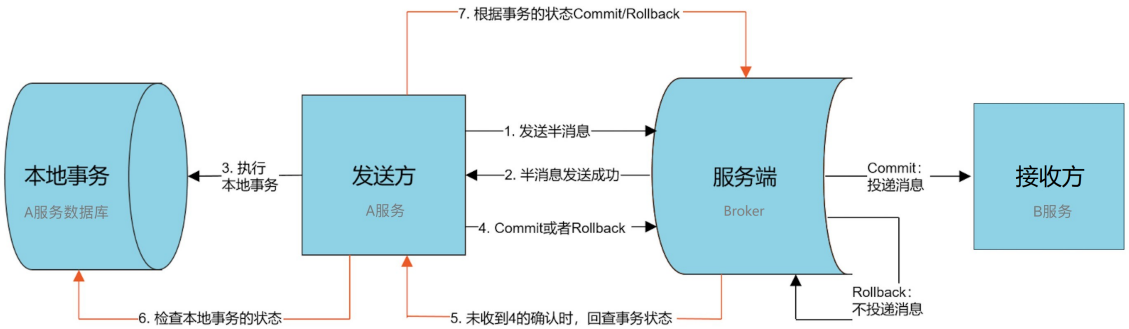
由于在生产环境中，消费方很有可能是个集群，若某一个消费节点超时但是消费成功，会导致集群同组其他节点重复消费该消息。另外意外宕机后恢复，由于消费进度没有及时写入磁盘，会导致消费进度部分丢失，从而导致消息重复消费。

5.2 RocketMQ可靠消息

RocketMQ 是一个来自阿里巴巴的分布式消息中间件，于 2012 年开源，并在 2017 年正式成为 Apache 顶级项目。Apache RocketMQ 4.3之后的版本正式支持事务消息，为分布式事务实现提供了便利性支持。因此，我们通过RocketMQ就可以解决前面的问题。

1.消息发送方执行本地事务与发送消息的原子性问题，也就是说如何保证本地事务执行成功，消息一定发送成功

RocketMQ中的Broker 与 发送方 具备双向通信能力，使得 broker 天生可以作为一个事务协调者存在；并且RocketMQ 本身提供了存储机制，使得事务消息可以持久化保存；这些优秀的设计可以保证即使发生了异常，RocketMQ依然能够保证达成事务的最终一致性。



1. 发送方发送一个事务消息给Broker，RocketMQ会将消息状态标记为“Prepared”，此时这条消息暂时不能被接收方消费。这样的消息称之为Half Message，即半消息。
2. Broker返回发送成功给发送方
3. 发送方执行本地事务，例如操作数据库
4. 若本地事务执行成功，发送commit消息给Broker，RocketMQ会将消息状态标记为“可消费”，此时这条消息就可以被接收方消费；若本地事务执行失败，发送rollback消息给Broker，RocketMQ将删除该消息。
5. 如果发送方在本地事务过程中，出现服务挂掉，网络闪断或者超时，那Broker将无法收到确认结果
6. 此时RocketMQ将会不停的询问发送方来获取本地事务的执行状态(**即事务回查**)
7. 根据事务回查的结果来决定Commit或Rollback，这样就保证了消息发送与本地事务同时成功或同时失败。

以上主干流程已由RocketMQ实现，对于我们来说只需要分别实现本地事务执行的方法以及本地事务回查的方法即可，具体来说就是实现下面这个接口：

```
public interface TransactionListener {
    /**
     - 发送prepare消息成功后回调该方法用于执行本地事务
     - @param msg 回传的消息，利用transactionId即可获取到该消息的唯一Id
     - @param arg 调用send方法时传递的参数，当send时候若有额外的参数可以传递到send方法中，
这里能获取到
     - @return 返回事务状态，COMMIT：提交  ROLLBACK：回滚  UNKNOW：未知，需要回查
      */
    LocalTransactionState executeLocalTransaction(final Message msg, final
Object arg);
```

```
    /**
     - @param msg 通过获取transactionId来判断这条消息的本地事务执行状态
     - @return 返回事务状态，COMMIT：提交  ROLLBACK：回滚  UNKNOW：未知，需要回查
      */
    LocalTransactionState checkLocalTransaction(final MessageExt msg);
}
```

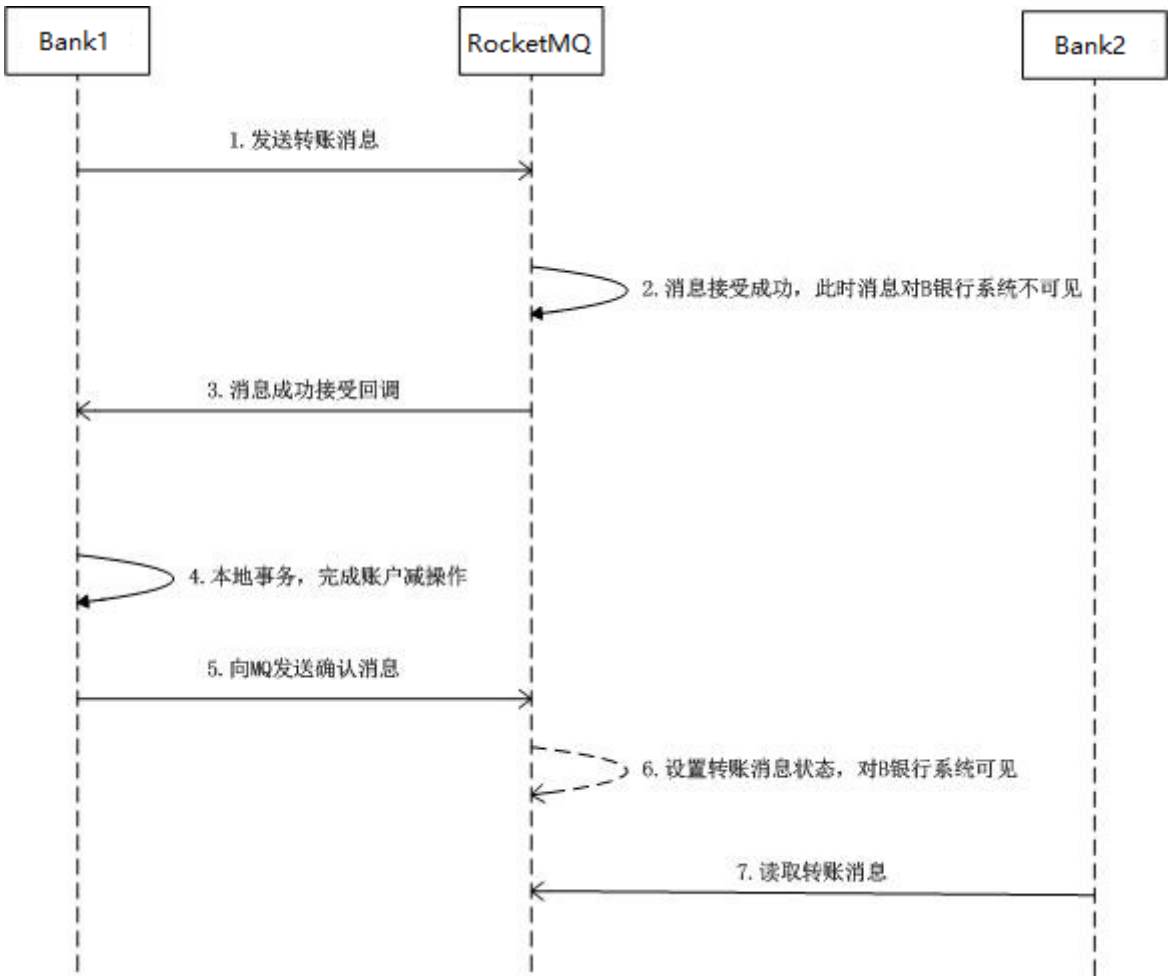
2.消息接收方接收消息与本地事务的原子性问题，也就是说如何保证接收消息成功后，本地事务一定执行成功

如果是出现了异常，RocketMQ会通过重试机制，每隔一段时间消费消息，然后再执行本地事务；如果是超时，RocketMQ就会无限制的消费消息，不断的去执行本地事务，直到成功为止。

5.3 快速入门

5.3.1 业务说明

本实例通过RocketMQ可靠消息实现最终一致性，模拟两个账户的转账交易过程。两个账户分别在不同的银行(张三在bank1、李四在bank2)，bank1、bank2是两个相互独立的微服务。



5.3.2 环境搭建

5.3.2.1 环境要求

- 数据库：MySQL-5.7+
- JDK：64位 jdk1.8+

- 微服务：spring-boot-2.1.3、spring-cloud-Greenwich.RELEASE
- RocketMQ服务端：RocketMQ-4.5.0
- RocketMQ客户端：RocketMQ-spring-boot-starter.2.0.2-RELEASE

5.3.2.2 创建数据库

本案例需要两个数据库，一个是bank1，一个是bank2，无需创建，直接使用Hmily快速入门案例中的数据库即可。另外，为了实现幂等性，需要分别在bank1、bank2数据库中新增de_duplication表，即交易记录表(去重表)。

```
DROP TABLE IF EXISTS `de_duplication`;
CREATE TABLE `de_duplication`  (
  `tx_no` bigint(20) NOT NULL,
  `create_time` datetime(0) NULL DEFAULT NULL,
  PRIMARY KEY (`tx_no`) USING BTREE
) ENGINE = InnoDB CHARACTER SET = utf8 COLLATE = utf8_bin ROW_FORMAT = Dynamic;
```

5.3.2.3 启动RocketMQ

启动nameserver:

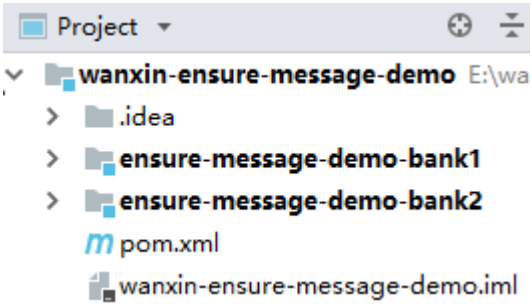
```
set ROCKETMQ_HOME=[RocketMQ服务端解压路径]
start [RocketMQ服务端解压路径]/bin/mqnamesrv.cmd
```

启动broker:

```
set ROCKETMQ_HOME=[RocketMQ服务端解压路径]
start [RocketMQ服务端解压路径]/bin/mqbroker.cmd -n 127.0.0.1:9876
autoCreateTopicEnable=true
```

5.3.2.4 Maven工程

在IDEA中导入maven工程，wanxin-ensure-message-demo为父工程，进行依赖管理。ensure-message-demo-bank1负责操作张三账户，连接数据库bank1。ensure-message-demo-bank2负责操作李四账户，连接数据库bank2。



工程中提供的初始文件，由大家自行查看。

5.3.3 功能实现

5.3.3.1 消息发送方bank1

1. 定义一个类封装转账消息：

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class AccountChangeEvent implements Serializable {
    /**
     * 账号
     */
    private String accountNo;
    /**
     * 变动金额
     */
    private double amount;
    /**
     * 事务号，时间戳
     */
    private long txNo;
}
```

2. 实现数据访问层，一共四个功能

```
@Mapper
@Component
public interface AccountInfoDao {

    /**
     * 修改某账号的余额
     * @param accountNo 账号
     * @param amount 变动金额
     * @return
     */
    @Update("update account_info set account_balance=account_balance+#{amount}
where account_no=#{accountNo}")
    int updateAccountBalance(@Param("accountNo") String accountNo,
@Param("amount") Double amount);

    /**
     * 查询某账号信息
     * @param accountNo 账号
     * @return
     */
    @Select("select * from account_info where where account_no=#{accountNo}")
    AccountInfo findByIdAccountNo(@Param("accountNo") String accountNo);

    /**
     * 查询某事务记录是否已执行
     * @param txNo 事务编号
     * @return
     */
    @Select("select count(1) from de_duplication where tx_no = #{txNo}")
    int isExistTx(long txNo);

    /**
     * 保存某事务执行记录
     * @param txNo 事务编号
     * @return
     */
    @Insert("insert into de_duplication values(#{txNo},now());")
```

```
    int addTx(long txNo);

}
```

3. 实现发送转账消息

```
@Component
@Slf4j
public class BankMessageProducer {
   @Resource
   private RocketMQTemplate rocketMQTemplate;

   public void sendAccountChangeEvent(AccountChangeEvent accountChangeEvent) {
      // 1.构造消息
      JSONObject object = new JSONObject();
      object.put("accountChange", accountChangeEvent);
      Message<String> msg =
MessageBuilder.withPayload(object.toJSONString()).build();
      // 2.发送消息
      rocketMQTemplate.sendMessageInTransaction("producer_ensure_transfer",
            "topic_ensure_transfer",
            msg, null);
   }
}
```

4. 实现业务层代码，分别实现了发送事务消息与本地事务扣减金额，注意doUpdateAccountBalance的本地事务若执行成功，就会在交易记录去重表（de_duplication）保存数据。

```
public interface AccountInfoService {
    /**
     * 更新帐号余额-发送消息
     * @param accountChange
     */
    void updateAccountBalance(AccountChangeEvent accountChange);

    /**
     * 更新帐号余额-本地事务
     * @param accountChange
     */
    void doUpdateAccountBalance(AccountChangeEvent accountChange);
}

@Service
@Slf4j
public class AccountInfoServiceImpl implements AccountInfoService {

   @Autowired
   private BankMessageProducer bankMessageProducer;

   @Autowired
   private AccountInfoDao accountInfoDao;

    /**
     * 更新帐号余额-发送通知
     * @param accountChange
```

```
     */
    @Override
    public void updateAccountBalance(AccountChangeEvent accountChange) {
        bankMessageProducer.sendAccountChangeEvent(accountChange);
    }

     /**
     * 更新帐号余额-本地事务
     * @param accountChange
     */
    @Override
    @Transactional(isolation = Isolation.SERIALIZABLE)
    public void doUpdateAccountBalance(AccountChangeEvent accountChange) {

 accountInfoDao.updateAccountBalance(accountChange.getAccountNo(),accountChange.
getAmount() * -1);
        accountInfoDao.addTx(accountChange.getTxNo());
    }
}
```

5. 实现RocketMQ事务消息监听器，其中有两个功能：

(1)executeLocalTransaction，该方法执行本地事务，会被RocketMQ自动调用

(2)checkLocalTransaction，该方法实现事务回查，利用了交易记录去重表（de_duplication），会被RocketMQ自动调用

```
@Component
@Slf4j
@RocketMQTransactionListener(txProducerGroup = "producer_ensure_transfer")
public class TransferTransactionListenerImpl implements
RocketMQLocalTransactionListener {

    @Autowired
    private AccountInfoService accountInfoService;

    @Autowired
    private AccountInfoDao accountInfoDao;

    /**
     * 执行本地事务
     * @param msg
     * @param arg
     * @return
     */
    @Override
    public RocketMQLocalTransactionState executeLocalTransaction(Message msg,
Object arg) {
        //1.接收并解析消息
        final JSONObject jsonObject = JSON.parseObject(new String((byte[])
                msg.getPayload()));
        AccountChangeEvent accountChangeEvent =

 JSONObject.parseObject(jsonObject.getString("accountChange"),AccountChangeEvent
.
                        class);
```

```
        //2.执行本地事务
        Boolean isCommit = true;
        try {
            accountInfoService.doUpdateAccountBalance(accountChangeEvent);
        }catch (Exception e){
            isCommit = false;
        }

        //3.返回执行结果
        if(isCommit){
            return RocketMQLocalTransactionState.COMMIT;
        }else {
            return RocketMQLocalTransactionState.ROLLBACK;
        }
    }

    /**
     * 事务回查
     * @param msg
     * @return
     */
    @Override
    public RocketMQLocalTransactionState checkLocalTransaction(Message msg) {
        //1.接收并解析消息
        final JSONObject jsonObject = JSON.parseObject(new String((byte[])
                msg.getPayload()));
        AccountChangeEvent accountChangeEvent =

 JSONObject.parseObject(jsonObject.getString("accountChange"),AccountChangeEvent
.
                        class);

        //2.查询de_duplication表
        int isExistTx = accountInfoDao.isExistTx(accountChangeEvent.getTxNo());

        //3.根据查询结果返回值
        if(isExistTx>0){
            return RocketMQLocalTransactionState.COMMIT;
        }else {
            return RocketMQLocalTransactionState.ROLLBACK;
        }
    }
}
```

6. 完善Controller代码

```
@RestController
@Slf4j
public class AccountInfoController {
    @Autowired
    private AccountInfoService accountInfoService;

    @GetMapping(value = "/transfer")
    public String transfer(){
        accountInfoService.updateAccountBalance(new
AccountChangeEvent("1",100,System.currentTimeMillis()));
        return "转账成功";
    }
}
```

5.3.3.2 消息接收方bank2

1. 实现数据访问层，和bank1一样，可以直接拿来用
2. 实现业务层功能, 增加账号余额，注意这里使用了交易记录去重表（de_duplication）实现幂等性控制

```
public interface AccountInfoService {
    /**
     * 更新帐号余额
     * @param accountChange
     */
    void updateAccountBalance(AccountChangeEvent accountChange);
}

@Service
@Slf4j
public class AccountInfoServiceImpl implements AccountInfoService {

   @Autowired
   private AccountInfoDao accountInfoDao;

   @Override
   @Transactional(isolation = Isolation.SERIALIZABLE)
   public void updateAccountBalance(AccountChangeEvent accountChange) {
      int isExistTx = accountInfoDao.isExistTx(accountChange.getTxNo());
      if(isExistTx == 0){

accountInfoDao.updateAccountBalance(accountChange.getAccountNo(),accountChange.g
etAmount());
         accountInfoDao.addTx(accountChange.getTxNo());
      }
   }
}
```

3. 实现RocketMQ事务消息监听器, 收到消息后，解析消息，调用业务层进行处理

```
@Component
@RocketMQMessageListener(topic = "topic_ensure_transfer", consumerGroup =
"consumer_ensure_transfer")
@Slf4j
public class EnsureMessageConsumer implements RocketMQListener<String>{
```

```
    @Autowired
    private AccountInfoService accountInfoService;

    @Override
    public void onMessage(String  projectStr) {
        System.out.println("开始消费消息：" + projectStr);
        final JSONObject jsonObject = JSON.parseObject(projectStr);
        AccountChangeEvent accountChangeEvent =
JSONObject.parseObject(jsonObject.getString("accountChange"),AccountChangeEvent.
class);
        accountChangeEvent.setAccountNo("2");
        accountInfoService.updateAccountBalance(accountChangeEvent);
    }
}
```

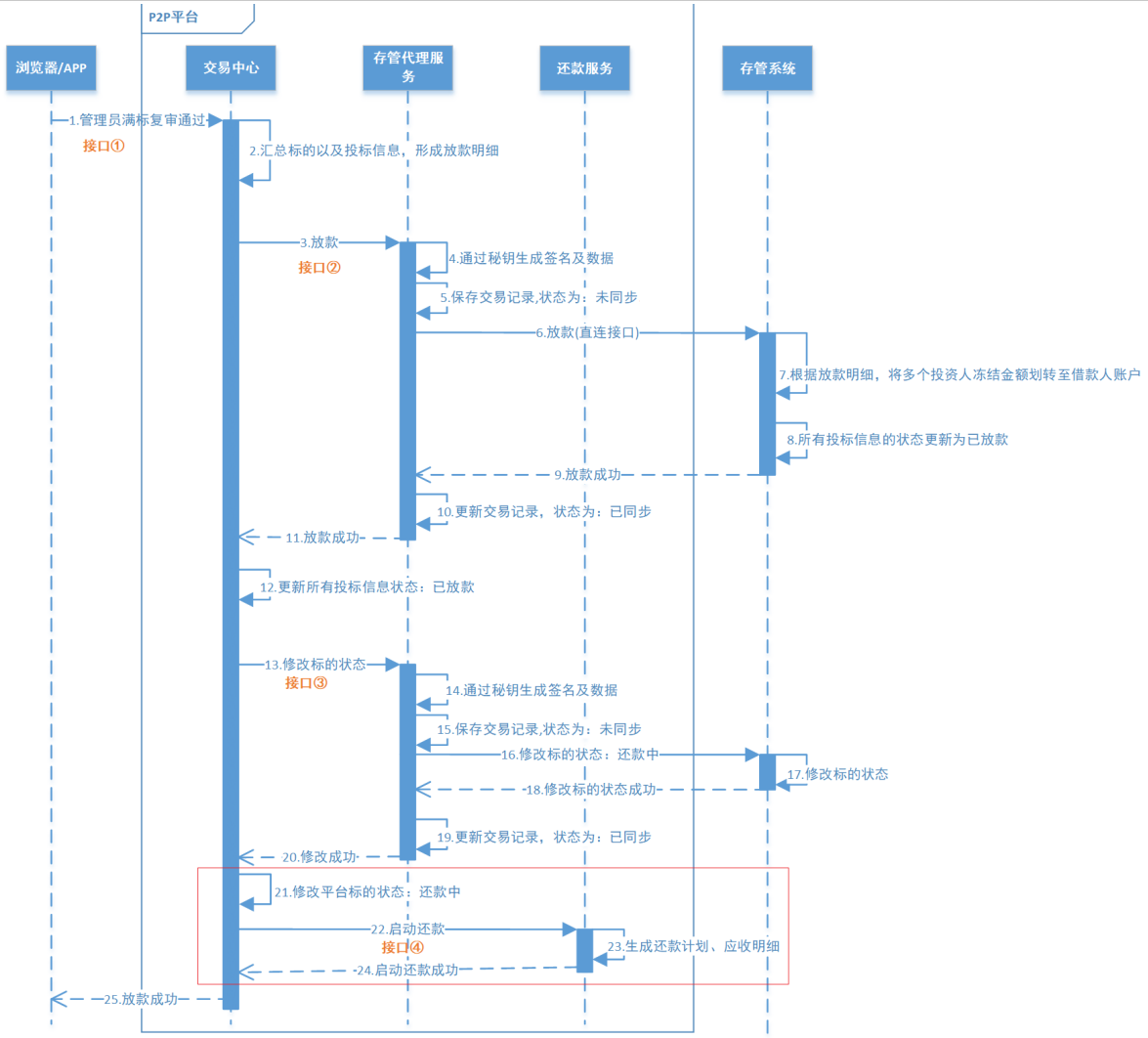
5.3.4 功能测试

- bank1和bank2都成功
- bank1执行本地事务失败，则bank2接收不到转账消息。
- bank1执行完本地事务后，不返回任何信息，则Broker会进行事务回查。
- bank2执行本地事务失败，会进行重试消费。

可靠消息最终一致性事务适合执行周期长且实时性要求不高的场景。引入该机制后，同步的事务操作变为基于消息执行的异步操作，避免了分布式事务中的同步阻塞操作的影响，并实现了两个服务的解耦。

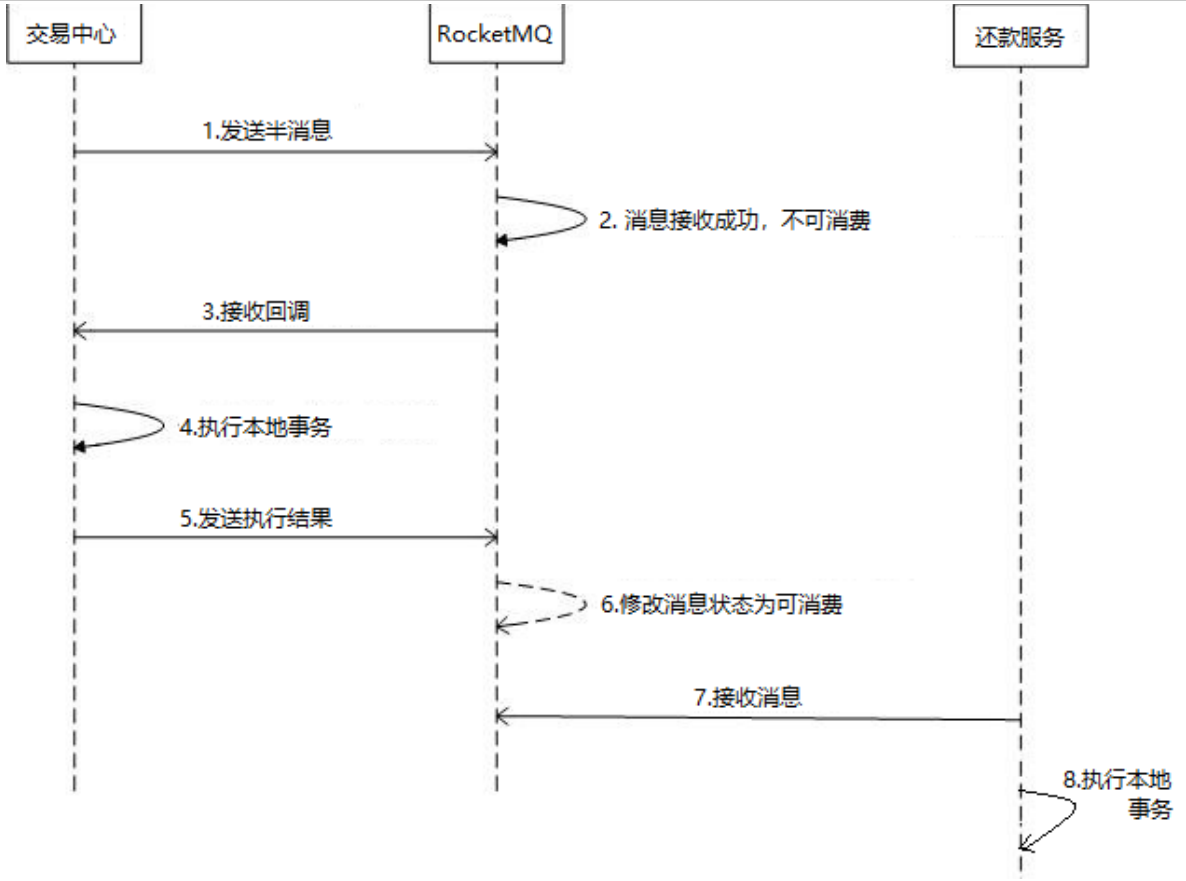
5.4 解决满标放款业务中的事务问题

5.4.1 问题分析



在满标放款业务中(上图红框部分)，交易中心修改标的状态为“还款中”，同时要通知还款服务生成还款计划和应收明细。两者为原子性绑定，即：标的状态修改成功，就必须生成还款计划和应收明细。由于涉及到两个独立的微服务，这里就存在分布式事务问题。

还款微服务生成还款计划和应收明细很有可能是一个较为耗时的业务，不建议阻塞主业务流程，并且此业务对强一致性要求较低，因此我们可以采用RocketMQ可靠消息实现最终一致性这个解决方案。



5.4.2 功能实现

5.4.2.1 消息发送方：交易中心微服务

1. 检查pom.xml的依赖

```
<dependency>
  <groupId>org.apache.rocketmq</groupId>
  <artifactId>rocketmq-spring-boot-starter</artifactId>
  <version>2.0.2</version>
</dependency>
```

2. 检查Apollo中交易中心项目的micro_service.spring-rocketmq名称空间配置

```
rocketmq.producer.group = producer_start_repayment
rocketmq.name-server = 127.0.0.1:9876
```

3. 新建P2pTransactionProducer类，实现交易中心发送消息

```
@Component
@Slf4j
public class P2pTransactionProducer {

    @Resource
    private RocketMQTemplate rocketMQTemplate;

    public void updateProjectStatusAndStartRepayment(Project project,
ProjectWithTendersDTO projectWithTendersDTO) {
        // 1.构造消息
```

```
        JSONObject object = new JSONObject();
        object.put("project", project);
        object.put("projectWithTendersDTO", projectWithTendersDTO);
        Message<String> msg =
MessageBuilder.withPayload(object.toJSONString()).build();

        // 2.发送消息
        rocketMQTemplate
                .sendMessageInTransaction("PID_START_REPAYMENT",
"TP_START_REPAYMENT", msg, null);
    }
}
```

4. 修改完善ProjectServiceImpl类中loansApprovalStatus方法的代码，实现**发送事务消息。**

```
... ...
@Autowired
private P2pTransactionProducer p2pTransactionProducer;
... ...
... ...
p2pTransactionProducer.updateProjectStatusAndStartRepayment(project,
projectWithTendersDTO);
return "审核成功";
... ...
```

5. 在ProjectServiceImpl类中新增updateProjectStatusAndStartRepayment方法，执行本地事务(修改标的状态为还款中)

```
@Transactional(rollbackFor = BusinessException.class)
@Override
public Boolean updateProjectStatusAndStartRepayment(Project project) {
    project.setProjectStatus(ProjectCode.REPAYING.getCode());
    return updateById(project);
}
```

6. 实现RocketMQ事务消息监听器，其中有两个功能：

(1)executeLocalTransaction，该方法执行本地事务，会被RocketMQ自动调用

(2)checkLocalTransaction，该方法实现事务回查，会被RocketMQ自动调用

```
@Component
@RocketMQTransactionListener(txProducerGroup ="PID_START_REPAYMENT")
public class P2pTransactionListenerImpl implements
RocketMQLocalTransactionListener {

    @Autowired
    private ProjectService projectService;

    @Autowired
    private ProjectMapper projectMapper;

    /**
     * 执行本地事务
     * @param msg
     * @param arg
```

```
     * @return
     */
    @Override
    public RocketMQLocalTransactionState executeLocalTransaction(Message msg,
Object arg) {
        //1. 解析消息
        final JSONObject jsonObject = JSON.parseObject(new String((byte[])
                msg.getPayload()));
        Project project =
                JSONObject.parseObject(jsonObject.getString("project"),
Project.class);

        //2. 执行本地事务
        Boolean result =
projectService.updateProjectStatusAndStartRepayment(project);

        //3. 返回执行结果
        if(result){
            return RocketMQLocalTransactionState.COMMIT;
        }else{
            return RocketMQLocalTransactionState.ROLLBACK;
        }
    }

    /*
    * 执行事务回查
     */
    @Override
    public RocketMQLocalTransactionState checkLocalTransaction(Message msg) {
        System.out.println("事务回查");
        //1. 解析消息
        final JSONObject jsonObject = JSON.parseObject(new String((byte[])
                msg.getPayload()));
        Project project =
                JSONObject.parseObject(jsonObject.getString("project"),
Project.class);

        //2. 查询标的状态
        Project pro=projectMapper.selectById(project.getId());

        //3. 返回结果
        if(pro.getProjectStatus().equals(ProjectCode.REPAYING.getCode())){
            return RocketMQLocalTransactionState.COMMIT;
        }else{
            return RocketMQLocalTransactionState.ROLLBACK;
        }
    }
}
```

数据访问层和Controller代码之前已经完成，这里不用再管。

5.4.2.2 消息接收方：还款微服务

1. 实现RocketMQ事务消息监听器, 收到消息后，解析消息，调用业务层进行处理

```
@Component
```

```
@RocketMQMessageListener(topic = "TP_START_REPAYMENT", consumerGroup =
"CID_START_REPAYMENT")
public class StartRepaymentMessageConsumer implements RocketMQListener<String> {

    @Autowired
    private RepaymentService repaymentService;

    @Override
    public void onMessage(String projectStr) {
        System.out.println("消费消息：" + projectStr);
        //1.解析消息
        JSONObject jsonObject = JSON.parseObject(projectStr);
        ProjectWithTendersDTO projectWithTendersDTO = JSONObject
                .parseObject(jsonObject.getString("projectWithTendersDTO"),
ProjectWithTendersDTO.class);
        //2.调用业务层，执行本地事务
        repaymentService.startRepayment(projectWithTendersDTO);
    }
}
```

2. 给业务层的startRepayment方法添加@Transactional(rollbackFor = BusinessException.class)注解，成为本地事务
3. 给repayment_plan表添加唯一索引，利用唯一索引实现幂等性

```
use `p2p_repayment`.`repayment_plan`;
ALTER TABLE `p2p_repayment`.`repayment_plan`
  ADD  UNIQUE INDEX `plan_unique` (`CONSUMER_ID`, `PROJECT_ID`,
`NUMBER_OF_PERIODS`);
```

5.4.3 功能测试

1. Apollo配置(交易中心)：

borrower.annual.rate	0.05	年化利率(平台佣金，利差)
commission.borrower.annual.rate	0.02	借款人给平台的利率
commission.investor.annual.rate	0.03	投资人让出利率

图中这三个数据不能为空，不能为0。

2. 测试场景

- 交易中心执行本地事务失败，则还款服务接收不到消息。
- 交易中心执行完本地事务后，不返回任何信息，则Broker会进行事务回查。
- 还款服务执行本地事务失败，会进行重试消费。
- 交易中心和还款服务都成功。