

# 什么是2PC

大道至简 悟在天成

Andy | 悟纤

## 前言

前面学习了分布式的基础理论，以理论为基础，针对不同的分布式场景业界常见的解决方案有2PC、TCC、可靠消息最终一致性、最大努力通知。

## 一、什么是2PC?

2PC即两阶段提交协议，是将整个事务流程分为两个阶段，准备阶段 (Prepare phase)、提交阶段 (Commit phase)，2是指两个阶段，P是准备阶段，C是提交阶段。

## 一、什么是2PC?

### 1.1 生活中例子来说明两阶段提交协议的工作过程

A组织B、C和D三个人去爬长城：如果所有人都同意去爬长城，那么活动将举行；如果有一人不同意去爬长城，那么活动将取消。用2PC算法解决该问题的过程如下：首先A将成为该活动的协调者，B、C和D将成为该活动的参与者。

## 一、什么是2PC?

### 1.1 生活中例子来说明两阶段提交协议的工作过程

#### 阶段1:

(1) A发邮件给B、C和D，提出下周三去爬山，问是否同意。那么此时A需要等待B、C和D的邮件。

(2) B、C和D分别查看自己的日程安排表。B、C发现自己在当日没有活动安排，则发邮件告诉A它们同意下周三去爬长城。由于某种原因，D白天没有查看邮件。那么此时A、B和C均需要等待。到晚上的时候，D发现了A的邮件，然后查看日程安排，发现周三当天已经有别的安排，那么D回复A说活动取消吧。

## 一、什么是2PC?

### 1.1 生活中例子来说明两阶段提交协议的工作过程

#### 阶段2:

(1) A收到了所有活动参与者的邮件，并且A发现D下周三不能去爬山。那么A将发邮件通知B、C和D，下周三爬长城活动取消。

(2) B、C回复A “太可惜了”，D回复A “不好意思”，至此该事务终止。

## 一、什么是2PC?

### 1.2 计算机的两阶段提交协议

(1) **准备阶段** (Prepare phase) : 事务管理器给每个参与者发送prepare消息, 每个数据库参与者在本地执行事务, 并写本地的**Undo/Redo**, 此时事务没有提交。

(Undo日志是记录修改前的数据, 用户数据库回滚, Redo日志是记录修改后的数据, 用于提交事务后写入数据。)

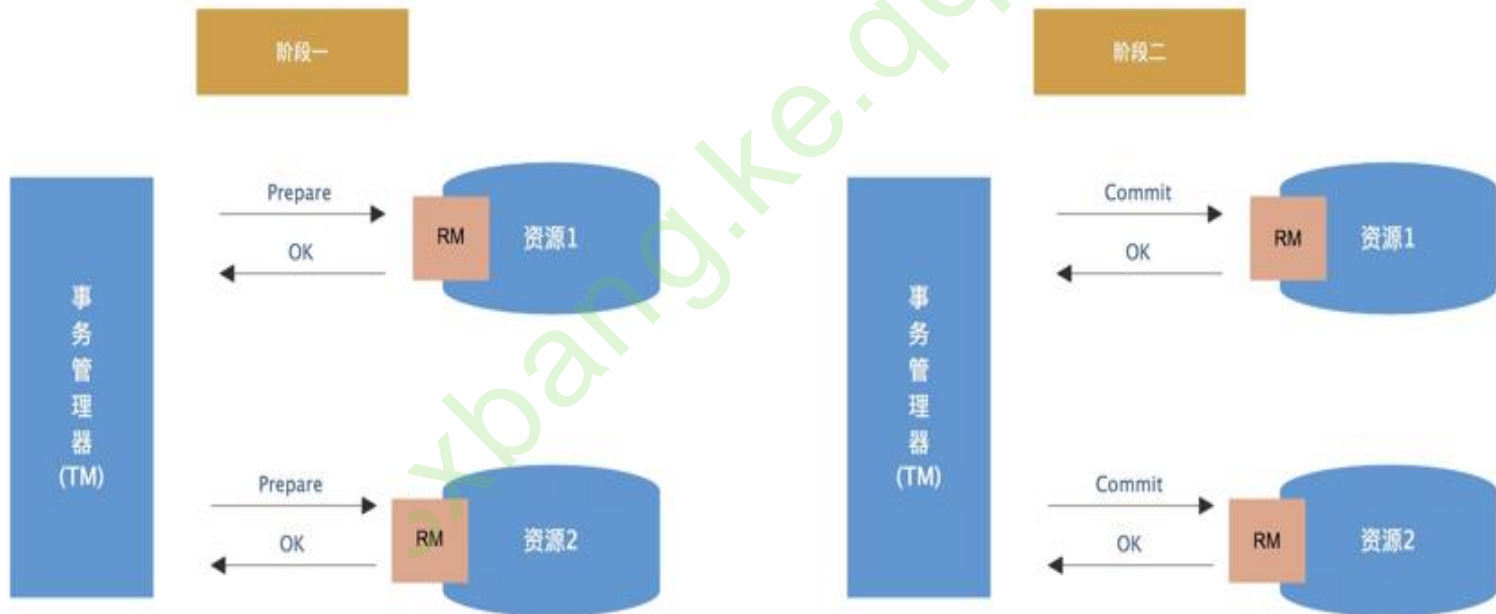
(2) **提交阶段** (Commit phase) : 如果事务管理器接收了参与者执行失败或者超时消息时, 直接给每个参与者发送回滚消息, 否则发送提交消息; 参与者根据事务管理器的指令执行提交或者回滚操作, 并释放事务处理过程中使用的锁资源。

**注意:** 必须在最后阶段释放锁资源。

## 一、什么是2PC?

### 1.2 计算机的两阶段提交协议

成功情况:

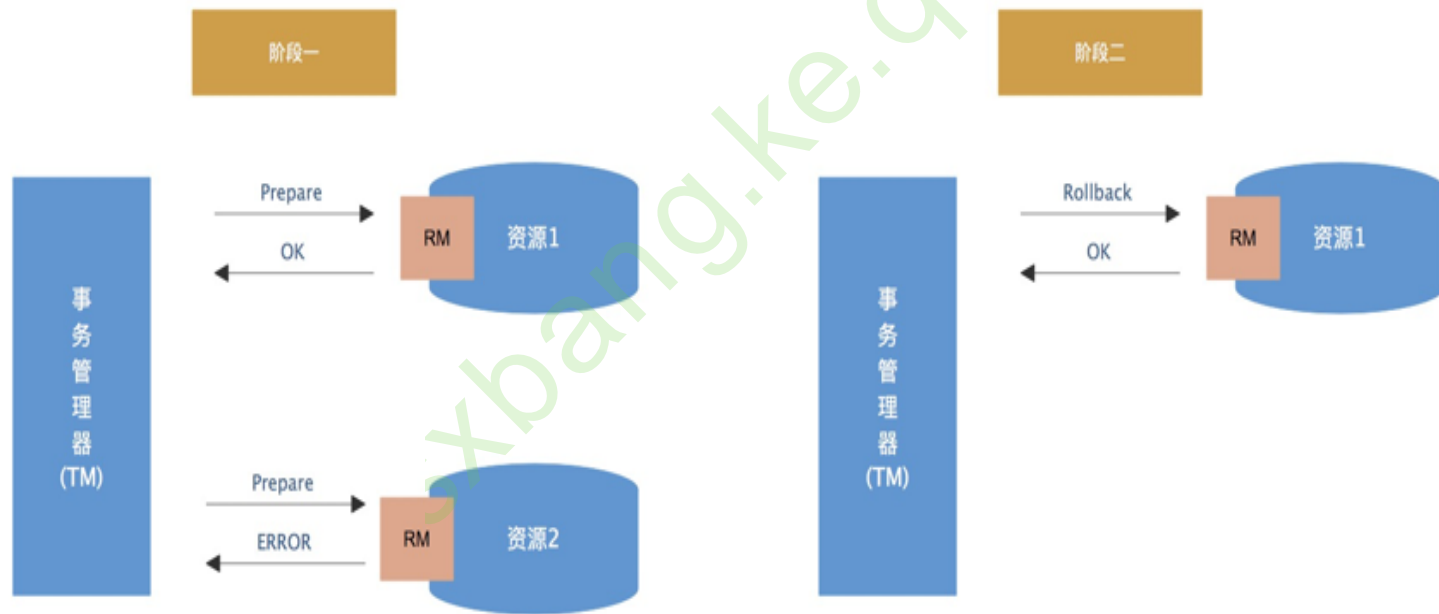




## 一、什么是2PC?

### 1.2 计算机的两阶段提交协议

失败情况:



## 一、什么是2PC?

### 1.2 计算机的两阶段提交协议

值得注意的是，二阶段提交协议的第一阶段准备阶段不仅仅是回答YES or NO，还是要执行事务操作的，只是执行完事务操作，并没有进行commit或者rollback。也就是说，一旦事务执行之后，在没有执行commit或者rollback之前，资源是被锁定的。这会造成阻塞。

# 什么是3PC

大道至简 悟在天成

Andy | 悟纤

## 一、什么是3PC

**三阶段提交协议** (3PC) 主要是为了解决两阶段提交协议的**阻塞问题**, 2pc存在的问题是当协作者崩溃时, 参与者不能做出最后的选择。因此参与者可能在协作者恢复之前保持阻塞。三阶段提交 (Three-phase commit), 是二阶段提交 (2PC) 的改进版本。

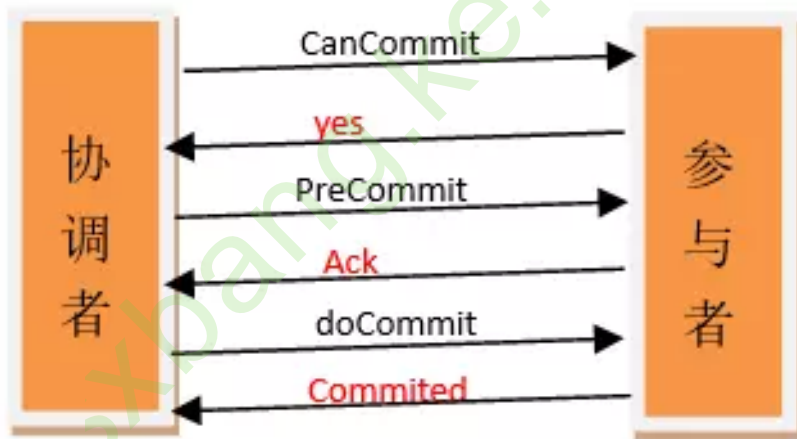
## 一、什么是3PC

与两阶段提交不同的是，三阶段提交有两个改动点：

- (1) **引入超时机制**：同时在协调者和参与者中都引入超时机制。
- (2) **新增一个阶段**：在第一阶段和第二阶段中插入一个准备阶段，保证了在最后提交阶段之前各参与节点的状态是一致的。

## 一、什么是3PC

也就是说，除了引入超时机制之外，3PC把2PC的准备阶段再次一分为二，这样三阶段提交就有CanCommit、PreCommit、DoCommit三个阶段。



# 一、什么是3PC

## 1.1 CanCommit

之前2PC的一阶段是本地事务执行结束后，最后不Commit,等其它服务都执行结束并返回Yes，由协调者发生commit才真正执行commit。而这里的CanCommit指的是尝试获取数据库锁 如果可以，就返回Yes。

(1) 协调者 向 参与者 发送CanCommit请求。询问是否可以执行事务提交操作。然后开始等待 参与者的响应。

(2) 参与者 接到CanCommit请求之后，正常情况下，如果其自身认为可以顺利执行事务，则返回Yes响应，并进入预备状态。否则反馈No。

**特别注意：此时并未执行事务操作，资源未被锁定。**

## 一、什么是3PC

### 1.2 PreCommit

在阶段一中，如果所有的参与者都返回Yes的话，那么就会进入PreCommit阶段进行事务预提交。这里的PreCommit阶段跟2PC的第一阶段是差不多的，只不过这里协调者和参与者都引入了超时机制（2PC中只有协调者可以超时，参与者没有超时机制）。



# 一、什么是3PC

## 1.2 PreCommit

**情况1:** 假如协调者从所有的参与者获得的反馈都是Yes响应，那么就会执行事务的预执行。

- (1) . **发送预提交请求:** 协调者向参与者发送PreCommit请求，并进入Prepared阶段。
- (2) . **事务预提交:** 参与者接收到PreCommit请求后，会执行事务操作，并将undo和redo信息记录到事务日志中。
- (3) . **响应反馈:** 如果参与者成功的执行了事务操作，则返回ACK响应，同时开始等待最终指令。

## 一、什么是3PC

### 1.2 PreCommit

**情况2：**假如有任何一个参与者向协调者发送了No响应，或者等待超时之后，协调者都没有接到参与者的响应，那么就执行事务的中断。

- (1) . **发送中断请求：** 协调者向所有参与者发送abort请求。
- (2) . **中断事务：** 参与者收到来自协调者的abort请求之后（或超时之后，仍未收到协调者的请求），执行事务的中断。

## 一、什么是3PC

### 1.3 DoCommit

执行提交事务或者回滚操作。

5xibang.ke.qq.cc

## 一、什么是3PC

### 1.3 DoCommit

#### 情况1：执行提交

- (1) . **发送提交请求**：协调接收到参与者发送的ACK响应，那么他将从预提交状态进入到提交状态。并向所有参与者发送doCommit请求。
- (2) . **事务提交**：参与者接收到doCommit请求之后，执行正式的事务提交。并在完成事务提交之后释放所有事务资源。
- (3) . **响应反馈**：事务提交完之后，向协调者发送Ack响应。
- (4) . **完成事务**：协调者接收到所有参与者的ack响应之后，完成事务。

## 一、什么是3PC

### 1.3 DoCommit

#### 情况2：中断事务

协调者没有接收到参与者发送的ACK响应（可能是接受者发送的不是ACK响应，也可能响应超时），那么就会执行中断事务。

- (1) . **发送中断请求**： 协调者向所有参与者发送abort请求
- (2) . **事务回滚**： 参与者接收到abort请求之后，利用其在阶段二记录的undo信息来执行事务的回滚操作，并在完成回滚之后释放所有的事务资源。
- (3) . **反馈结果**： 参与者完成事务回滚之后，向协调者发送ACK消息
- (4) . **中断事务**： 协调者接收到参与者反馈的ACK消息之后，执行事务的中断。

## 二、2PC与3PC的区别

### 1.3 DoCommit

相对于2PC，3PC主要解决的单点故障问题，并减少阻塞，因为一旦参与者无法及时收到来自协调者的信息之后，他会默认执行commit。而不会一直持有事务资源并处于阻塞状态。但是这种机制也会导致数据一致性问题，因为，由于网络原因，协调者发送的abort响应没有及时被参与者接收到，那么参与者在等待超时之后执行了commit操作。这样就和其他接到abort命令并执行回滚的参与者之间存在数据不一致的情况。

# XA方案

大道至简 悟在天成

Andy | 悟纤

## 一、XA方案

2PC的传统方案是在数据库层面实现的，如Oracle、MySQL都支持2PC协议，为了统一标准减少行业内不必要的对接成本，需要指定标准化的处理模型及接口标准，国际开发标准组织Open Group定义了分布式事务处理模型DTP (Distributed Transaction Processing Reference Model) 。



## 一、XA方案

**DTP（分布式事务处理模型）** 模型定义如下角色：

- (1) **AP** (Application Program) : 即应用程序，可以理解为使用DTP分布式事务的程序。
- (2) **RM** (Resource Manager) : 即资源管理器，可以理解为事务的参与者，一般情况下是指一个数据库实例，通过资源管理器对该数据库进行控制，资源管理器控制着分支事务。
- (3) **TM** (Transaction Manager) : 事务管理器，负责协调和管理事务，事务管理器控制着全局事务，管理事务生命周期，并协调各个RM。

**全局事务**：分布式事务处理环境中，需要操作多个数据库共同完成一个工作，这个工作即是一个全局事务。

## 一、XA方案

DTP模型定义TM与RM之间通讯的接口规范叫XA，简单理解就是为数据库提供的2PC协议。基于数据库的XA协议来实现2PC又称为XA方案。

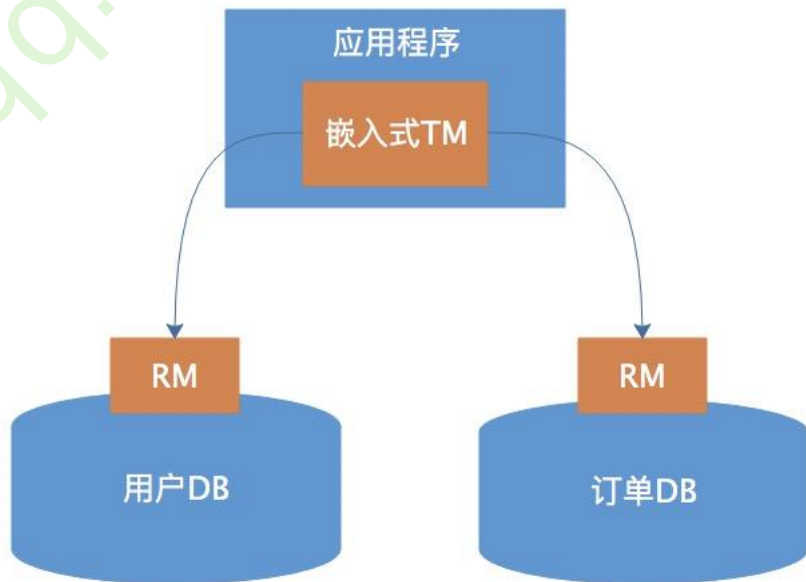
以上各个角色之间的交互方式如下：

- (1) TM向AP提供应用程序编程接口，AP通过TM提交及回滚事务。
- (2) TM交易中间件通过XA接口来通知RM数据库事务的开始、结束以及提交、回滚等。

## 一、XA方案

举个例子：用户消费扣减金额，创建订单：

- (1) 应用程序 (AP) 持有用户库和订库两个数据库。
- (2) 应用程序 (AP) 通过TM通知用户库RM扣减金额，同时通知订单库RM为该用户创建订单，RM此时并未提交事务，此时用户和订单资源锁定。
- (3) TM收到执行回复，只要有一方失败则向其它RM发起回顾事务，回滚完毕，资源锁释放。
- (4) TM收到执行回复，全部成功，此时向所有RM发起提交事务，提交完毕，资源锁释放。



# Seata方案

大道至简 悟在天成

Andy | 悟纤

# 前言

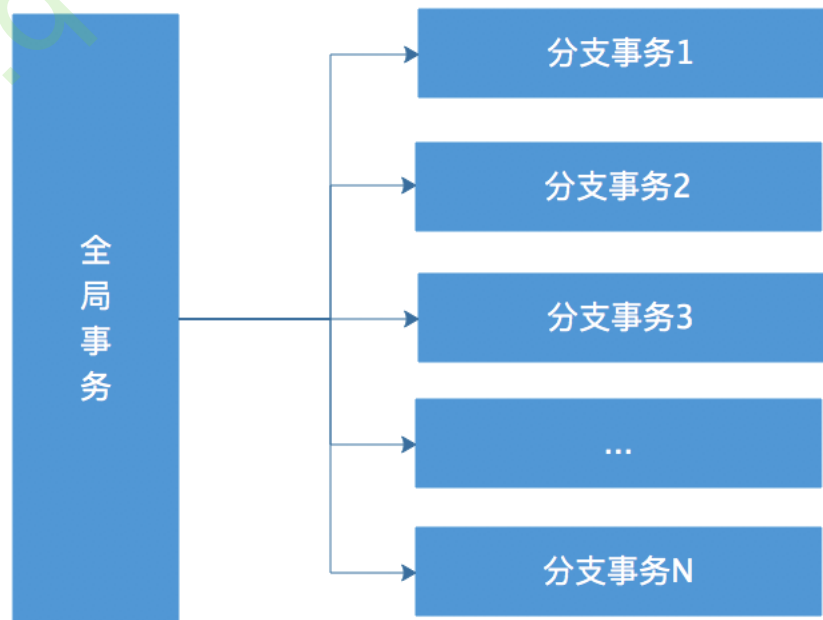
Seata是由阿里中间件发起的开源项目Fescar，后改命Seata，它是一个开源的分布式事务框架。

传统 2PC 资源锁占用时间过长、性能差的问题在Seata中得到了解决，它通过对本地关系数据库的分支事务的协调来驱动完成全局事务，在工作在应用层的中间件。

主要优点：性能较好、不长时间占用连接资源，对业务代码零侵入。目前提供AT模式（即2PC）和TCC模式的分布式事务解决方案。

## 一、Seata的设计思想

Seata把一个分布式事务理解成一个包含了若干分支事务的全局事务，全局事务的职责是协调其下管辖的分支事务达成一致，要么一起成功提交，要么一起失败回滚。此外通常一个分支事务就是一个关系型数据库的本地事务。



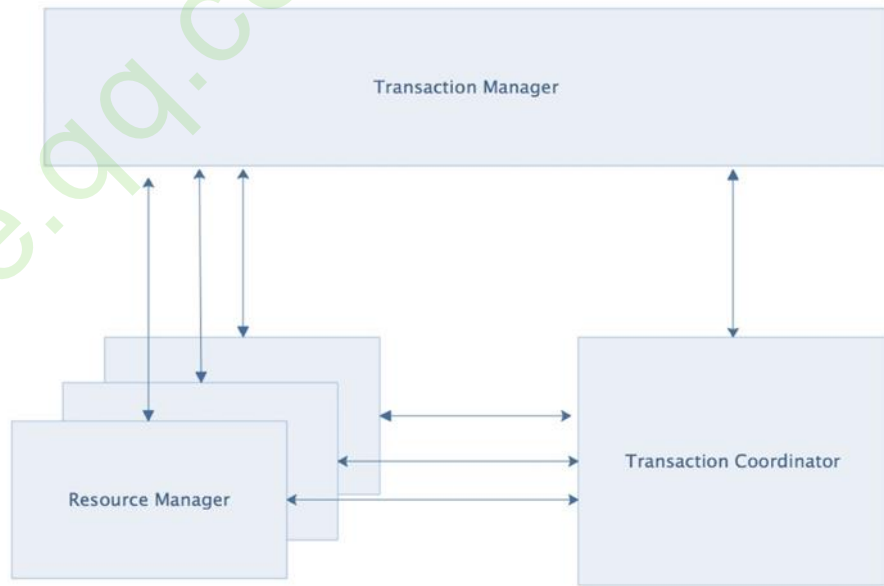
## 一、Seata的设计思想

Seata定义了3个组件来协调分布式事务的处理：

(1) **Transaction Coordinator (TC)**：事务协调器，它是独立的中间件，需要独立部署运行，它维护全局事务的运行状态，接收TM指令发起全局事务的提交与回滚，负责与RM通信协调各分支事务的提交或回滚。

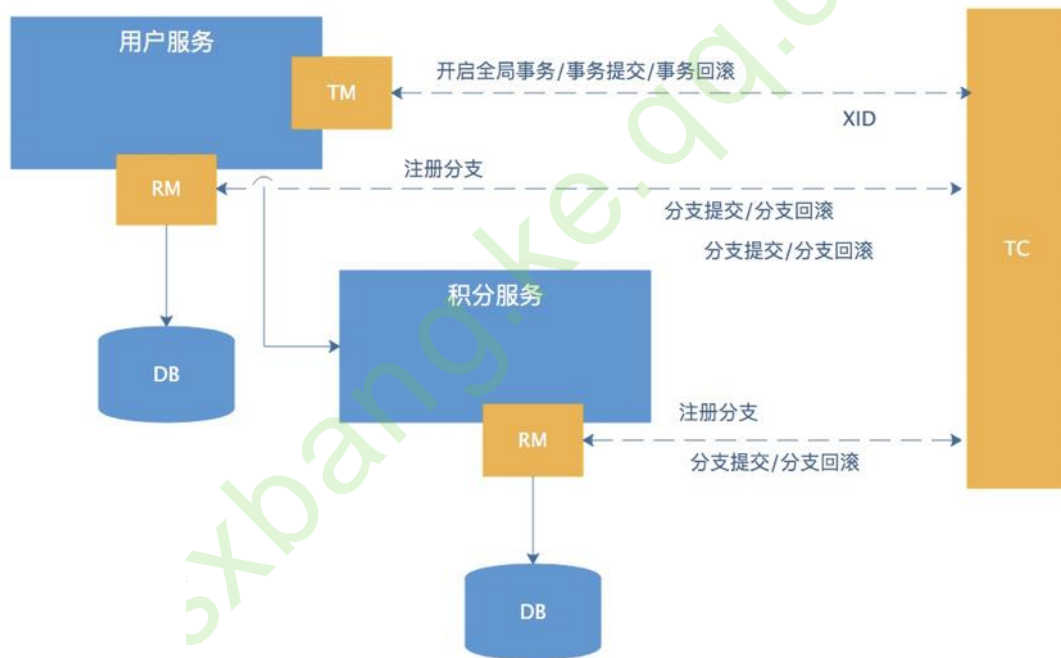
(2) **Transaction Manager (TM)**：事务管理器，TM需要嵌入应用程序中工作，它负责开启一个全局事务，并最终向TC发起全局提交或回滚的指令。

(3) **Resource Manager (RM)**：控制分支事务，负责分支注册、状态汇报，并接收事务协调器TC的指令，驱动分支（本地）事务的提交和回滚。



## 二、例子：新用户注册送积分

我们来看下新用户注册送积分seata的分布式事务处理过程：





### 三、Seata实现2PC与传统2PC的差别

(1) **架构层次方面**：传统2PC方案的RM实际上在数据库层，RM本质上就是数据库本身，通过XA协议实现，而Seata的RM是以jar包的形式作为中间件层部署在应用程序这一侧的。

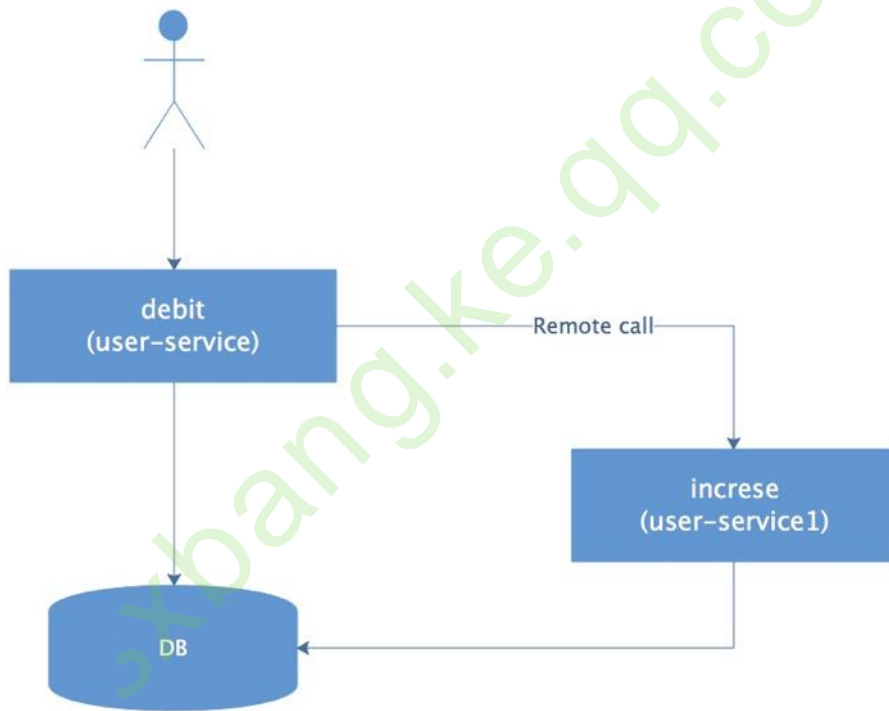
(2) **两阶段提交方面**：传统2PC无论第二阶段的决议是commit还是rollback, 事务资源的锁都要保持到第二阶段完成才释放，而Seata的做法是在第一阶段将本地事务提交，这样就可以省去第二阶段持锁的时间，整体提高效率。

# 搭建框架

大道至简 悟在天成

Andy | 悟纤

## 张三给李四转账



## 张三给李四转账

使用STS搭建：

tx-transfer（父模块）

- tx-transfer-discovery（注册中心）
- tx-transfer-user-service（用户模块）
- tx-transfer-user-service1（用户模块）

## 1.1 sc-tx-transfer (父模块)

spring-cloud-dependencies (Greenwich. SR3)

spring-boot-dependencies (2.1.5. RELEASE)

```
<spring-boot.version>2.1.5.RELEASE</spring-boot.version>  
<spring-cloud.version>Greenwich.SR3</spring-cloud.version>
```

## 1.1 sc-tx-transfer (父模块)

```
<dependencyManagement>
  <dependencies>
    <!-- spring cloud -->
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    <!-- spring boot -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>${spring-boot.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

# 注册中心编码

大道至简 悟在天成

Andy | 悟纤

### 1.2 tx-transfer-discovery (注册中心)

注册中心主要思路是：

- (1) 新建一个Maven module;
- (2) 引入eureka-server的依赖;
- (3) 添加配置即可。
- (4) 编写启动类
- (5) 验证注册中心



### 1.2.1 新建一个Maven module

新建一个module，名称为tx-transfer-discovery。

### 1.2.2 引入eureka-server的依赖

在pom.xml文件引入依赖：

```
<dependency>  
    <groupId>org.springframework.cloud</groupId>  
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>  
  
</dependency>
```

## 注册中心

### 1.2.3 添加配置

在application.properties添加配置:

```
#####  
##为了与后续要进行注册的服务区分, 这里将服务注册中心的端口通过server.port属性设置为1111。  
##启动工程后, 访问: http://localhost:1111/  
#####  
#端口号设置.  
server.port=1111  
#是否将自身注册 ;禁用eureka的客户端注册行为,在默认设置下, 该服务注册中心也会将自己作为客户端来尝试注册它自己  
eureka.client.register-with-eureka=false  
#获取注册表  
eureka.client.fetch-registry=false  
#设置zone地址  
eureka.client.serviceUrl.defaultZone=http://127.0.0.1:${server.port}/eureka/
```

### 1.2.4 编写启动类

将App.java重命名为DiscoveryApp.java, 在里面编写启动代码:

```
@SpringBootApplication
@EnableEurekaServer
public class DiscoveryApp {
    public static void main(String[] args) {
        SpringApplication.run(DiscoveryApp.class, args);
    }
}
```

### 1.2.5 验证注册中心

运行DiscoveryApp, 访问地址: <http://localhost:1111/>

# 账户模块

大道至简 悟在天成

Andy | 悟纤

### 1.3.1 复制一份discovery的代码

为了不重复进行编码，复制一份discovery的代码出来（直接在父类项目下，复制，粘贴就可以复制出来项目），然后修改pom.xml的名称为tx-transfer-user-service。

# 账户模块

## 1.3.2 添加依赖

```
<!-- web支持: 1、web mvc; 2、restful; 3、jackjson支持; 4、aop ..... -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<!-- mysql数据库 -->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
</dependency>

<!-- spring data jpa -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```



## 账户模块

### 1.3.3 添加配置 在application.properties文件中添加配置：

```
#通过spring.application.name属性，  
#我们可以指定微服务的名称后续在调用的时候只需要使用该名称就可以进行服务的访问。  
spring.application.name=user-service  
#为了在本机上测试区分服务提供方和服务注册中心，使用server.port属性设置不同的端口。  
server.port=2222
```

```
#####  
###eureka  
#####  
eureka.client.serviceUrl.defaultZone=http://127.0.0.1:1111/eureka/
```

```
#####  
###datasource  
#####  
spring.datasource.url = jdbc:mysql://localhost:3306/db_account?createDatabaseIfNotExist=true  
spring.datasource.username = root  
spring.datasource.password = root  
spring.datasource.driverClassName = com.mysql.cj.jdbc.Driver
```

```
#####  
### Java Persistence Api  
#####  
# Specify the DBMS  
spring.jpa.database = MYSQL  
# Show or not log for each sql query  
spring.jpa.show-sql = true  
# Hibernate ddl auto (create, create-drop, update)  
spring.jpa.hibernate.ddl-auto = update  
#N...
```

### 1.3.4 编写启动类

将启动类名称修改为UserServiceApp，然后添加如下代码：

```
@SpringBootApplication
@EnableDiscoveryClient
public class DiscoveryApp {
    public static void main(String[] args) {
        SpringApplication.run(DiscoveryApp.class, args);
    }
}
```

### 1.3.5 编写实体类

```
@Entity
@DynamicUpdate
@DynamicInsert
public class Account {
    @Id
    private Long uid;

    private BigDecimal money;
}
```

### 1.3.6 编写持久类

```
public interface AccountRepository extends JpaRepository<Account, Long>{  
    Account findByUid(long uid);  
}
```

## 账户模块

### 1.3.7 编写服务类 定义服务接口实现：

```
@Service
public class AccountServiceImpl implements AccountService {

    private static final Long ERROR_USER_ID = 1002L;

    @Autowired
    private AccountRepository accountRepository;

    @Override
    @Transactional
    public void debit(Long uid, BigDecimal num) {
        Account account = accountRepository.findByUid(uid);
        if(account == null) {
            throw new RuntimeException("Account not exist");
        } else if(account.getMoney().compareTo(num)<0) {
            throw new RuntimeException("Money not enough");
        }

        account.setMoney(account.getMoney().subtract(num));
        accountRepository.save(account);

        if (ERROR_USER_ID.equals(uid)) {
            throw new RuntimeException("account branch exception");
        }
    }
}
```

## 账户模块

### 1.3.8 编写controller

```
@RestController
public class AccountController {
    @Autowired
    private AccountService accountService;

    //http://127.0.0.1:2222/debit?uid=1001&money=1
    @RequestMapping("/debit")
    public Boolean debit(Long uid, BigDecimal money) {
        accountService.debit(uid, money);
        return true;
    }
}
```

## 账户模块

### 1.3.9 准备数据

```
create schema if Not Exists db_account;
use db_account;

-----
-- Table structure for account
-----

DROP TABLE IF EXISTS `account`;
CREATE TABLE `account` (
  `uid` bigint(20) NOT NULL,
  `money` decimal(19,2) DEFAULT NULL,
  PRIMARY KEY (`uid`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

INSERT INTO account(uid, money) VALUES (1001, 10000);

INSERT INTO account(uid, money) VALUES (1002, 10000);
```

## 账户模块

### 1.3.10 验证本地事务

启动注册中心、启动用户服务，进行测试：

正常提交事务：

<http://127.0.0.1:2222/debit?uid=1001&money=1>

异常提交事务：

账号有误：<http://127.0.0.1:2222/debit?uid=1002&money=1>

余额不足：<http://127.0.0.1:2222/debit?uid=1001&money=99999>



# 账户模块1

大道至简 悟在天成

Andy | 悟纤

### 1.4.1 复制一份user-service的代码

复制一份user-service的代码出来，然后修改pom.xml的名称为tx-transfer-user-service1，启动类修改为：UserService1App

## 账户模块1

### 1.4.2 修改配置

修改application.properties文件：

```
spring.application.name=user-service1
```

```
server.port=2223
```

另外就是datasource.url 指向数据库：db\_account，也可以指向db\_account1数据库。

# 账户模块1

## 1.4.3 修改service方法

```
@Override
@Transactional
public void increase(Long uid, BigDecimal money) {
    Account account = accountRepository.findByUid(uid);
    if(account == null) {
        throw new RuntimeException("Account not exist");
    }

    account.setMoney(account.getMoney().add(money));
    accountRepository.save(account);

    if (ERROR_USER_ID.equals(uid)) {
        throw new RuntimeException("account branch exception");
    }
}
```

### 1.4.4 验证本地事务

正常访问:

<http://127.0.0.1:2223/increase?uid=1002&money=1>

异常访问:

<http://127.0.0.1:2223/increase?uid=1001&money=1>

由于本地事务存在，所以事务可以正常回滚。

# 实现转账业务

大道至简 悟在天成

Andy | 悟纤

## 实现转账业务

### 1.5.1 启用FeignClient

在user-service实现转账业务。

在UserServiceApp添加注解进行启用：

```
@EnableFeignClients
```

## 实现转账业务

### 1.5.2 编写client

编写请求user-service的client:

```
@FeignClient(name="user-service1")
public interface UserService1Client {

    @GetMapping("/increase")
    public Boolean increse(@RequestParam("uid") Long uid, @RequestParam("money") BigDecimal money);

}
```



# 实现转账业务

## 1.5.3 编写转账的service

```
public boolean transfer(BigDecimal money) {  
  
    //张三钱减少  
    accountService.debit(1001L, money);  
    //李四增加  
    userService1Client.increase(1002L, money);  
  
    //执行其他的代码，正常的情况或者不正常的情况.  
    if(money.compareTo(new BigDecimal(10)) == 0) {  
        //==10的话，抛出异常.为了模拟事务.  
        int i=1/0;  
        System.out.println(i);  
    }  
    return true;  
}
```

# 实现转账业务

## 1.5.3 编写转账的service

```
public boolean transfer(BigDecimal money) {  
  
    //张三钱减少  
    accountService.debit(1001L, money);  
    //李四增加  
    userService1Client.increase(1002L, money);  
  
    //执行其他的代码，正常的情况或者不正常的情况.  
    if(money.compareTo(new BigDecimal(10)) == 0) {  
        //==10的话，抛出异常.为了模拟事务.  
        int i=1/0;  
        System.out.println(i);  
    }  
    return true;  
}
```

## 实现转账业务

### 1.5.4 编写controller

```
@RequestMapping("/transfer")
public Boolean transfer(BigDecimal money) {
    businessService.transfer(money);
    return true;
}
```

## 实现转账业务

### 1.5.5 验证事务

正常访问:

<http://127.0.0.1:2222/transfer?money=1>

异常访问:

<http://127.0.0.1:2222/transfer?money=10>

本地事务能够确保，金额不扣减，但是却不能保证远程调用的事务能够正常。

# 使用Seata实战2PC分布式事务

大道至简 悟在天成

Andy | 悟纤

## 集成说明

(1) 添加seata的依赖：spring-cloud-alibaba-seata

(2) 使用seata数据源代理：io.seata.rm.datasource.DataSourceProxy，需要添加一个配置文件进行配置即可：核心就是拦截数据源对连接进行的commit和rollback进行操作来实现的。

(3) 在业务发起方添加全局事务注解：@GlobalTransactional：业务方如果需要本地事务的话，是不需要的@Transactional注解的。

(4) 需要添加file.conf和registry.conf：registry.conf如果配置为file的方式才需要使用file.conf的方式，否则不需要的。另外file.conf有一个很重要的配置，就是事务组名称的配置：

```
vgroup_mapping.user-service-fescar-service-group="default"
```

没有配置正常，很有可能就连接不上seata-server了。

(5) 事务的控制依赖于一个服务端，需要在github上进行下载，下载地址：<https://github.com/seata/seata/releases>

## 一、Seata接入

### 1.1 下载seata-server

在官网下载seata-server, 下载地址:

<https://github.com/seata/seata/releases>

CSDN下载:

<https://download.csdn.net/download/linxingliang/11979837>

## 一、Seata接入

### 1.2 启动seata-server

使用一下指令启动seata-server:

```
sh seata-server.sh -p 8091 -m file
```



## 一、Seata接入

### 1.3 父项目添加依赖

在父项目添加依赖：

```
<dependency>  
    <groupId>com.alibaba.cloud</groupId>  
    <artifactId>spring-cloud-alibaba-seata</artifactId>  
    <version>2.1.0.RELEASE</version>  
</dependency>
```

## 一、Seata接入

### 1.4 在每个子项目中添加依赖

在每个子项目中添加依赖：

```
<dependency>  
    <groupId>com.alibaba.cloud</groupId>  
    <artifactId>spring-cloud-alibaba-seata</artifactId>  
</dependency>
```

## 一、Seata接入

### 1.5 添加数据源代理配置类

在有需要参与事务的项目中添加一个数据源代理配置类（如果是发起事务的服务不想要使用到数据源，那么就无需进行配置）：

```
@Configuration
public class DataSourceConfig {

    @Bean
    @ConfigurationProperties(prefix = "spring.datasource")
    public DruidDataSource druidDataSource() {
        return new DruidDataSource();
    }

    /**
     * 需要将 DataSourceProxy 设置为主数据源，否则事务无法回滚
     *
     * @param druidDataSource The DruidDataSource
     * @return The default datasource
     */
    @Primary
    @Bean("dataSource")
    public DataSource dataSource(DruidDataSource druidDataSource) {
        return new DataSourceProxy(druidDataSource);
    }
}
```

## 一、Seata接入

### 1.6 添加seata相关配置文件

添加两个配置文件registry.conf和file.conf（这个记得修改事务组的名称）：

registry.conf

```
registry {  
  # file 、nacos 、eureka、redis、zk  
  type = "file"  
  
  nacos {  
    serverAddr = "localhost"  
    namespace = "public"  
    cluster = "default"  
  }  
  eureka {  
    serviceUrl = "http://localhost:1001/eureka"  
    application = "default"  
    weight = "1"  
  }  
  redis {  
    serverAddr = "localhost:6381"  
    db = "0"  
  }  
  zk {  
    cluster = "default"  
    serverAddr = "127.0.0.1:2181"  
    session.timeout = 6000  
    connect.timeout = 2000  
  }  
}
```

## 一、Seata接入

### 1.6 添加seata相关配置文件

添加两个配置文件registry.conf和file.conf（这个记得修改事务组的名称）：

file.conf

```
transport {  
    # tcp udt unix-domain-socket  
    type = "TCP"  
    #NIO NATIVE  
    server = "NIO"  
    #enable heartbeat  
    heartbeat = true  
    #thread factory for netty  
    thread-factory {  
        boss-thread-prefix = "NettyBoss"  
        worker-thread-prefix = "NettyServerNIOWorker"  
        server-executor-thread-prefix = "NettyServerBizHandler"  
        share-boss-worker = false  
        client-selector-thread-prefix = "NettyClientSelector"  
        client-selector-thread-size = 1  
        client-worker-thread-prefix = "NettyClientWorkerThread"  
        # netty boss thread size,will not be used for UDT  
        boss-thread-size = 1  
        #auto default pin or 8  
        worker-thread-size = 8  
    }  
    shutdown {  
        # when destroy server, wait seconds
```

## 一、Seata接入

### 1.7 在全局事务发起的地方标注全局事务

在transfer()方法使用@GlobalTransactional进行标注。

当添加了@GlobalTransactional之后，如果这里没有本地事务的可以不用添加注解@Transactional

## 一、Seata接入

### 1.8 创建日志undo\_log表

Seata-server之所以能够进行事务的回滚是依赖于undo\_log表的(每个数据库都需要添加):

```
DROP TABLE IF EXISTS `undo_log`;
CREATE TABLE `undo_log` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `branch_id` bigint(20) NOT NULL,
  `xid` varchar(100) NOT NULL,
  `context` varchar(128) NOT NULL,
  `rollback_info` longblob NOT NULL,
  `log_status` int(11) NOT NULL,
  `log_created` datetime NOT NULL,
  `log_modified` datetime NOT NULL,
  `ext` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `ux_undo_log` (`xid`,`branch_id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

## 一、Seata接入

### 1.9 验证分布式事务

访问 <http://127.0.0.1:2222/transfer?money=10>

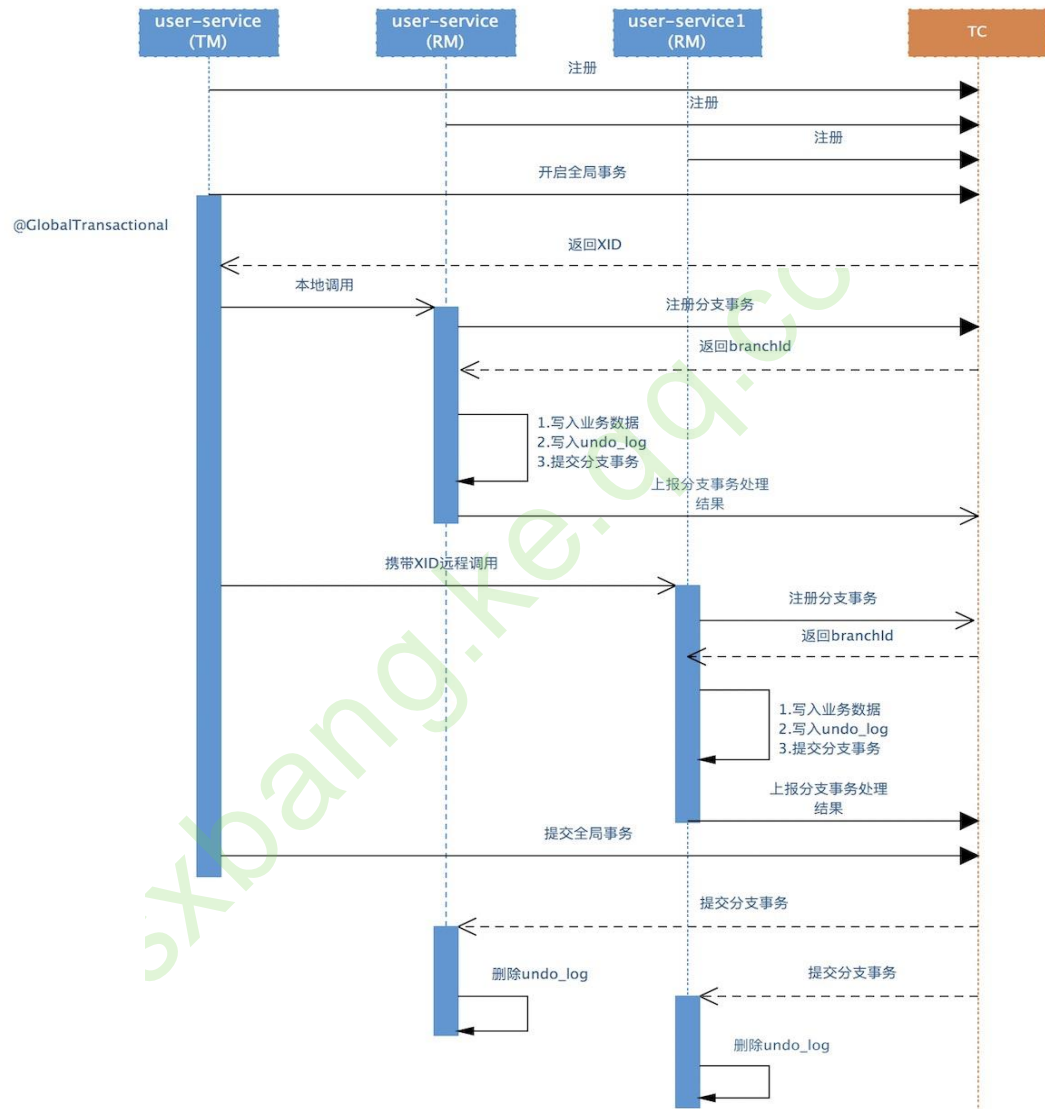
就可以验证分布式事务，事务能够正常回滚。

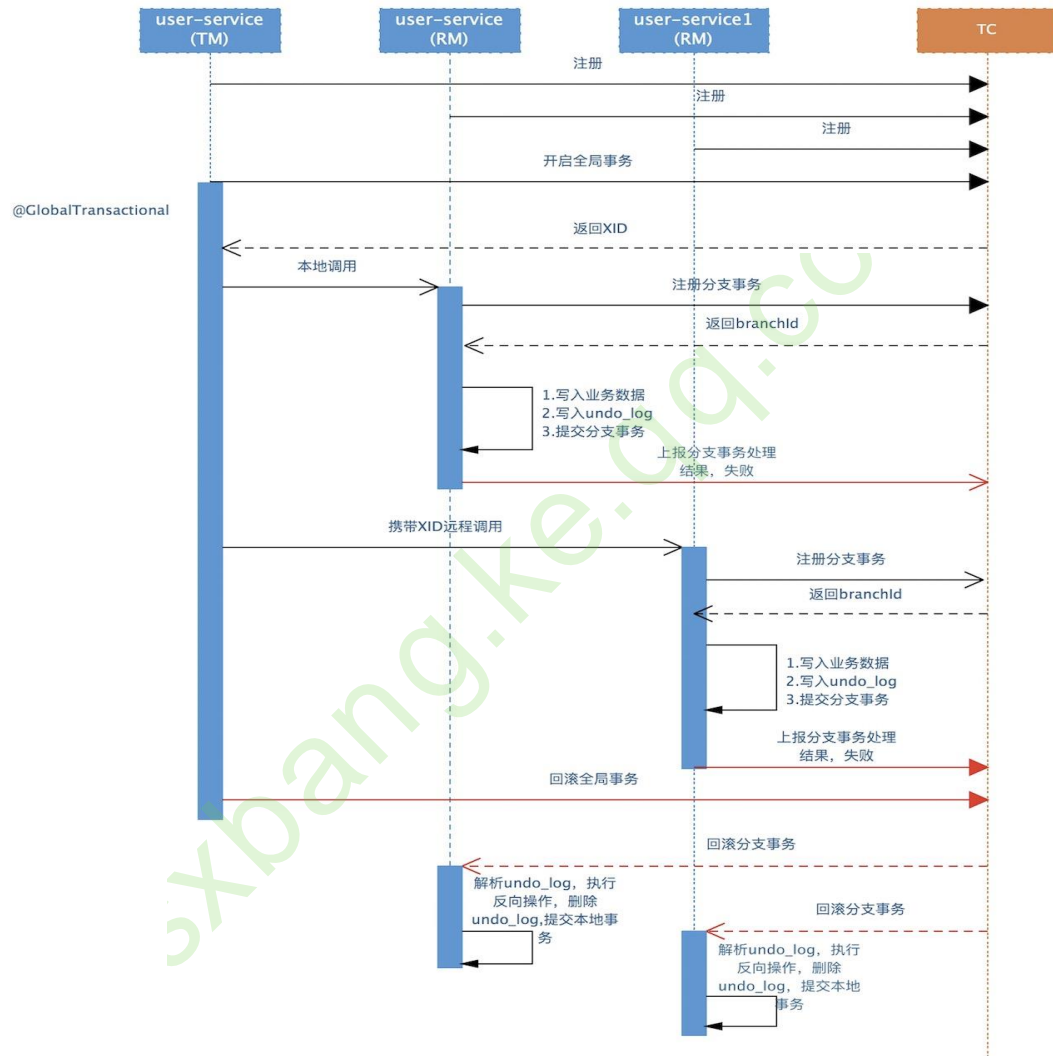


# Seata执行流程分析

大道至简 悟在天成

Andy | 悟纤





# TX-LCN背景

大道至简 悟在天成

Andy | 悟纤

## TX-LCN来源

LCN框架在2017年6月份发布第一个版本，从开始的1.0，已经发展到了5.0版本。

LCN名称是由早期版本的LCN框架命名，在设计框架之初的1.0 ~ 2.0的版本时框架设计的步骤是如下，各取其首字母得来的LCN命名：

锁定事务单元 (lock)

确认事务模块状态 (Confirm)

通知事务 (notify)

## TX-LCN来源

5.0以后由于框架兼容了LCN、TCC、TXC三种事务模式，为了避免区分LCN模式，特此将LCN分布式事务改名为TX-LCN分布式事务框架。

**说明:** TXC (Taobao Transaction Constructor) 模式命名来源于阿里云的GTS，实现原理是在执行SQL之前，先查询SQL的影响数据保存起来然后再执行业务，当需要回滚的时候就采用这些记录数据回滚事务。

## 框架定位

LCN并不生产事务，LCN只是本地事务的协调工。

TX-LCN定位于一款事务协调性框架，框架其本身并不操作事务，而是基于对事务的协调从而达到事务一致性的效果。

## 解决方案

在一个分布式系统下存在多个模块协调来完成一次业务。那么就存在一次业务事务下可能**横跨多种数据源节点**的可能。TX-LCN将可以解决这样的问题。



## 解决方案

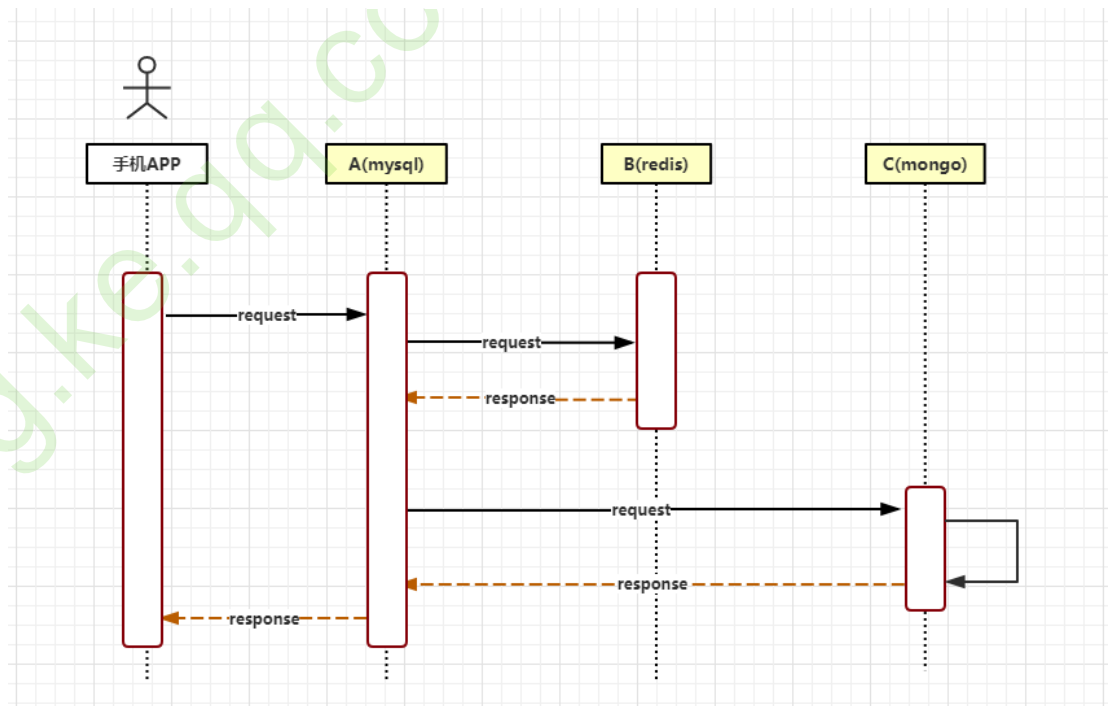
例如存在服务模块A、B、C。

A模块是mysql作为数据源的服务，

B模块是基于redis作为数据源的服务，

C模块是基于mongo作为数据源的服务。

若需要解决他们的事务一致性就需要针对不同的节点采用不同的方案，并且统一协调完成分布式事务的处理。



若采用TX-LCN分布式事务框架，则可以将A模块采用LCN模式、B/C采用TCC模式就能完美解决。

# TX-LCN原理

大道至简 悟在天成

Andy | 悟纤

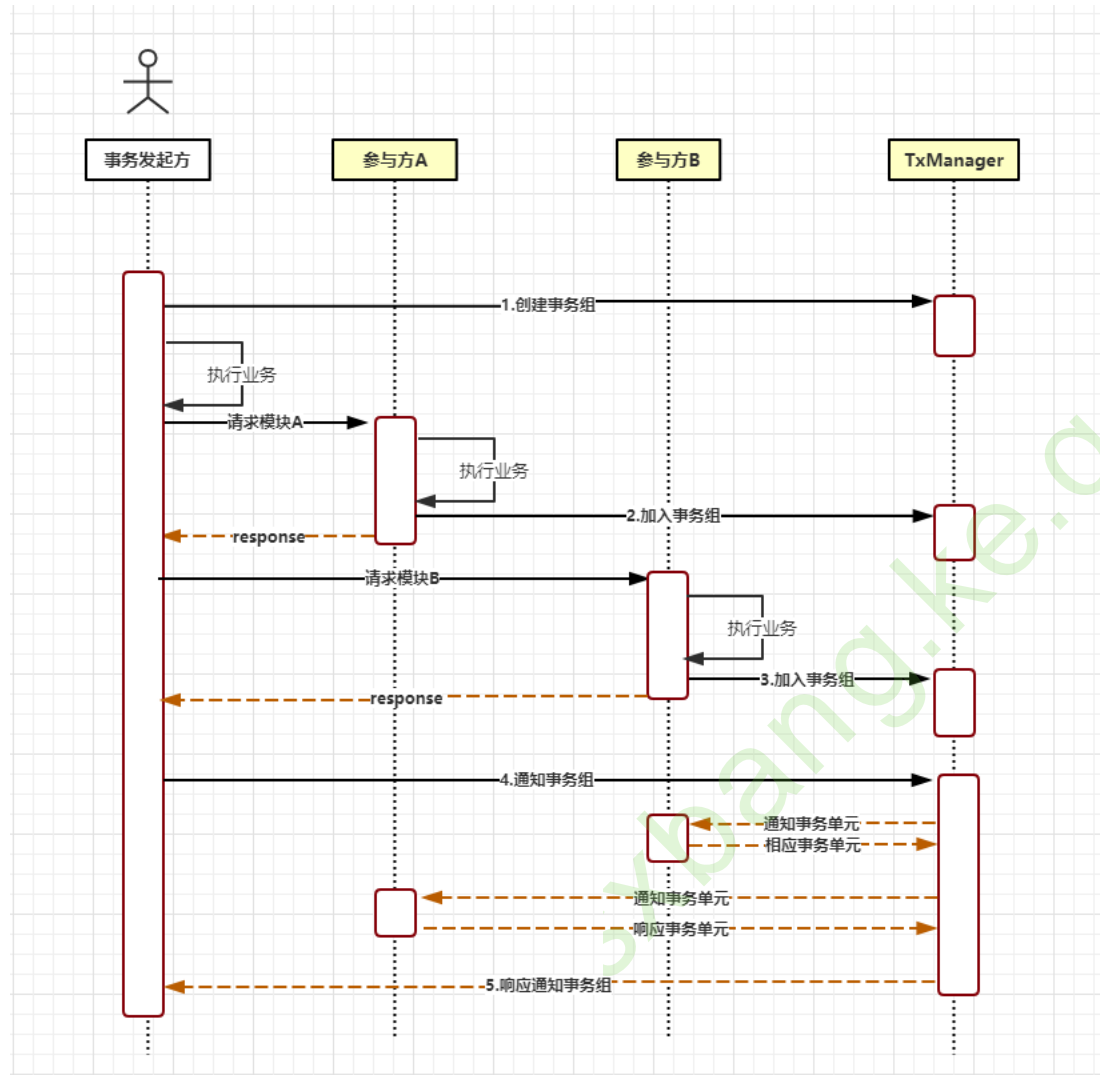
## 一、事务控制原理

TX-LCN由两大模块组成, TxClient、TxManager,

TxClient作为模块的依赖框架, 提供TX-LCN的标准支持,

TxManager作为分布式事务的控制方。

事务发起方或者参与方都由TxClient端来控制。



(1) **创建事务组**: 是指在事务发起方开始执行业务代码之前先调用TxManager创建事务组对象, 然后拿到事务标示GroupId的过程。

(2) **加入事务组**: 添加事务组是指参与方在执行完业务方法以后, 将该模块的事务信息通知给TxManager的操作。

(3) **通知事务组**: 是指在发起方执行完业务代码以后, 将发起方执行结果状态通知给TxManager,TxManager将根据事务最终状态和事务组的信息来通知相应的参与模块提交或回滚事务, 并返回结果给事务发起方。

# 使用TX-LCN实战2PC分布式事务

大道至简 悟在天成

Andy | 悟纤

## 一、TM安装准备

安装TM需要依赖的中间件: JRE1.8+, MySQL5.6+, Redis3.2+

5xibang.ke.qq.com

## 一、TM安装准备

### 1.1 安装JRE

JRE是作为JAVA开发者最基本的一个开发环境，这个基本是没有任何问题，主要要注意这里的版本号是1.8+，低于这个版本TM可能无法启动。

## 一、TM安装准备

### 1.2 MySQL

对于MySQL的安装，对于有这个数据库操作经验的人来说也是小菜一碟了，注意版本号是5.6+即可，安装成功之后记得进行启动。



## 一、TM安装准备

### 1.3 Redis

对于redis的安装的方式有：源码编译安装、brew install的方式，这里我们可以使用brew install redis进行安装。

3.2版本的安装指令：`brew install redis@3.2`

4.0版本的安装指令：`brew install redis@4.0`

reids官网已经是5.0的版本了，可以使用下载源码，make编译一下。

## 二、TM安装

准备工作之后，就可以安装TM了，对于TM的安装其实很简单，就是一个java的jar包，TM依赖于MySQL数据库，所以需要创建数据库才能进行启动TM。

## 二、TM安装

### 2.1 创建数据库

我们说过TM依赖于MySQL数据库，主要体现在异常信息需要有地方进行存储，所以需要先创建一个数据库来存储：

(1) 创建MySQL数据库，名称为：tx-manager

## 二、TM安装

### 2.1 创建数据库

#### (2) 创建数据表

```
CREATE TABLE `t_tx_exception` (  
  `id` bigint(20) NOT NULL AUTO_INCREMENT,  
  `group_id` varchar(64) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL DEFAULT NULL,  
  `unit_id` varchar(32) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL DEFAULT NULL,  
  `mod_id` varchar(128) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci NULL DEFAULT NULL,  
  `transaction_state` tinyint(4) NULL DEFAULT NULL,  
  `registrar` tinyint(4) NULL DEFAULT NULL,  
  `remark` varchar(4096) NULL DEFAULT NULL,  
  `ex_state` tinyint(4) NULL DEFAULT NULL COMMENT '0 未解决 1已解决',  
  `create_time` datetime(0) NULL DEFAULT NULL,  
  PRIMARY KEY (`id`) USING BTREE  
) ENGINE = InnoDB AUTO_INCREMENT = 1 CHARACTER SET = utf8mb4 COLLATE = utf8mb4_general_ci ROW_FORMAT = Dynamic;
```

## 二、TM安装

### 2.2 下载TM

#### JAR包下载:

- (1) CSDN: <https://download.csdn.net/download/linxingliang/11189070>
- (2) 百度云盘: [https://pan.baidu.com/s/1kNe6w1Nyk9rt\\_3HoRQnDGGQ](https://pan.baidu.com/s/1kNe6w1Nyk9rt_3HoRQnDGGQ)

#### 源码下载:

- (1) 从github上进行下载最新版本(源码-编译-打包) (截止到2019.12.1最新版本是5.0.2) :  
<https://github.com/codingapi/tx-lcn/releases>

## 二、TM安装

### 2.2 下载TM

**修改配置信息：** application.properties

下载完成之后，解压zip文件，对于application.properties进行简单的修改，主要是数据库连接信息和redis连接信息（没有必要，使用默认即可）：

```
spring.application.name=tx-manager
server.port=7970

spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/tx-manager?characterEncoding=UTF-8
spring.datasource.username=root
spring.datasource.password=root

mybatis.configuration.map-underscore-to-camel-case=true
mybatis.configuration.use-generated-keys=true
```

## 二、TM安装

### 2.2 下载TM

#### 修改配置信息：application.properties

下载完成之后，解压zip文件，对于application.properties进行简单的修改，主要是数据库连接信息和redis连接信息（没有必要，使用默认即可）：

```
spring.application.name=tx-manager
server.port=7970

spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/tx-manager?characterEncoding=UTF-8
spring.datasource.username=root
spring.datasource.password=root

mybatis.configuration.map-underscore-to-camel-case=true
mybatis.configuration.use-generated-keys=true
```

```
#tx-lcn.logger.enabled=true
# TxManager Host Ip
#tx-lcn.manager.host=127.0.0.1
# TxClient连接请求端口
#tx-lcn.manager.port=8070
# 心跳检测时间(ms)
#tx-lcn.manager.heart-time=15000
# 分布式事务执行总时间
#tx-lcn.manager.dtx-time=30000
#参数延迟删除时间单位ms
#tx-lcn.message.netty.attr-delay-time=10000
#tx-lcn.manager.concurrent-level=128
# 开启日志
#tx-lcn.logger.enabled=true
#logging.level.com.codingapi=debug
#redis 主机
#spring.redis.host=127.0.0.1
#redis 端口
#spring.redis.port=6379
#redis 密码

#spring.redis.password=
```

## 二、TM安装

### 2.3 启动TM

```
java -jar txlcn-tm-5.0.2.RELEASE.jar
```



## 二、TM安装

### 2.4 访问控制台

访问控制台地址: <http://127.0.0.1:7970/>

默认密码: codingapi

# 使用TX-LCN实战2PC分布式事务

大道至简 悟在天成

Andy | 悟纤

### 三、TC业务开发

#### 集成说明：

- (1) 添加依赖txlcn-tc、txlcn-txmsg-netty，在父类指定版本，在单独的工程中进行引入。
- (2) 开启分布式事务注解：@EnableDistributedTransaction
- (3) 所有的业务方法配置：@LcnTransaction

## 三、TC业务开发

### 3.1 复制一份项目

将tx-transfer复制一份出来，取名为：lcn-tx-transfer。

## 三、TC业务开发

### 3.2 在父项目声明TC依赖

在lcn-tx-transfer父类pom.xml文件，添加版本：

```
<txlcn.version>5.0.2.RELEASE</txlcn.version>
```

声明tc依赖：

```
<dependency>
  <groupId>com.codingapi.txlcn</groupId>
  <artifactId>txlcn-tc</artifactId>
  <version>${txlcn.version}</version>
</dependency>

<dependency>
  <groupId>com.codingapi.txlcn</groupId>
  <artifactId>txlcn-txmsg-netty</artifactId>
  <version>${txlcn.version}</version>
</dependency>
```

## 三、TC业务开发

### 3.3 在子项目引入TC依赖

```
<dependency>  
  <groupId>com.codingapi.txlc</groupId>  
  <artifactId>txlc-tc</artifactId>  
</dependency>
```

```
<dependency>  
  <groupId>com.codingapi.txlc</groupId>  
  <artifactId>txlc-txmsg-netty</artifactId>
```

```
</dependency>
```

## 三、TC业务开发

### 3.4 TC开启分布式事务注解

在主类上使用@EnableDistributedTransaction（需要参与的服务都需要添加）：

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
@EnableDistributedTransaction
public class UserServiceApp {
    public static void main(String[] args) {
        SpringApplication.run(UserServiceApp.class, args);
    }
}
```

## 三、TC业务开发

### 3.5 TC微服务分布式事务注解

在参与分布式事务的方法中添加@LcnTransaction（整个事务的分支事务的方法都需要添加）：

BusinessService.transfer :

```
@Transactional  
@LcnTransaction  
public boolean transfer(BigDecimal money) {}
```

AccountServiceImpl.debit :

```
@Transactional  
@LcnTransaction  
public void debit(Long uid, BigDecimal money) {}
```

AccountServiceImpl.increase :

```
@Transactional  
@LcnTransaction  
public void increase(Long uid, BigDecimal money) {}
```

简单来说：就是添加了@Transactional的注解的地方，就需要添加@LcnTransaction。



## 三、TC业务开发

### 3.6 TC配置信息

# 默认之配置为TM的本机默认端口

tx-lcn.client.manager-address=127.0.0.1:8070

## 三、TC业务开发

### 3.7 验证分布式事务

访问如下地址进行验证访问

<http://127.0.0.1:2222/transfer?money=10>

就可以验证分布式事务，事务能够正常回滚。

# TX-LCN不起作用

大道至简 悟在天成

Andy | 悟纤

## TX-LCN不起作用

建议尽量采用最新的版本，还有问题可以按照如下方式进行排查：

- (1) 确认Tx-client (TC) 与Tx-manager (TM) 的版本是完全一致的。
- (2) 确认TC与TM的配置都没有问题，并确认TM是启动状态，可访问后台（默认密码：codingapi）。
- (3) 检测TM下TC在线模块是否正常对应。
- (4) 当以上都没有问题的时候，检查能否进入了DataSourceAspect的拦截，在开发工具的debug下环境下断点确认。

## TX-LCN不起作用

若没有进入拦截器，可能存在两种情况：

一：确认Datasource 是否为spring的bean对象，若非spring对象，请先处理成spring对象。

二：若是spring对象，但是无法进入拦截。可以自行添加切面的方式进入拦截。

该org.apache.tomcat.jdbc.pool.DataSourceProxy.getConnection方法就不能进入DataSource的拦截，可自行添加如下所示：

## TX-LCN不起作用

```
@Component
@Aspect
@Slf4j
public class TomcatDataSourceAspect implements Ordered {

    @Autowired
    private DTXResourceWeaver dtxResourceWeaver;//TX-LCN 资源切面处理对象

    @Around("execution(public java.sql.Connection org.apache.tomcat.jdbc.pool.DataSourceProxy.getConnection(..) )")
    public Object around(ProceedingJoinPoint point) throws Throwable {
        log.info("proxy my aspect..");
        return dtxResourceWeaver.getConnection(() -> (Connection) point.proceed());
    }

    @Override
    public int getOrder() {
        return 0;
    }
}
```

## TX-LCN不起作用

### 如何在springboot 1.5 版本下使用TX-LCN

在5.0.1.RELEASE版本以后开始支持了springboot 1.5版本。使用步骤：

(1) 导入5.0.1.RELEASE版本的pom。

(2) 在properties下设置txlcen-org.springframework.cloud.commons.version版本到

1.3.5.RELEASE

```
<properties>
  <codingapi.txlcen.version>5.0.1.RELEASE</codingapi.txlcen.version>
  <txlcen-org.springframework.cloud.commons.version>1.3.5.RELEASE</txlcen-org.springframework.cloud.commons.version>
</properties>
```

# JTA接口定义

大道至简 悟在天成

Andy | 悟纤



## 前言

前面讲到了很多数据库实现了XA的协议，但是也有些数据库未提供对XA的支持，那么是否有方案不依赖与XA协议的数据库实现，这就是本节要提到的XA协议的JAVA实现。

那么XA协议的Java实现也有自己的规范，那么就是JTA（Java Transaction API）。

## 一、JTA

JTA(Java Transaction API)，是J2EE的编程接口规范，它是XA协议的JAVA实现。

简单理解：

DTP模型定义TM与RM之间通讯的接口规范叫XA；

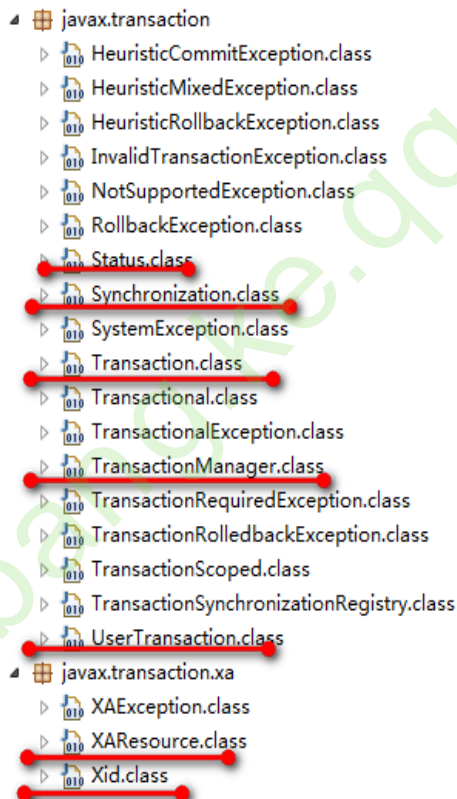
XA协议的数据库实现，是XA方案；

XA协议的JAVA实现，是JTA；

# 一、JTA

## JTA的包结构:

- javax.transaction
- javax.transaction.xa



## 一、JTA

### 1.1 AP、TM、RM三大对象

- AP: 我们的应用程序
- TM: 即 `javax.transaction.TransactionManager` 事务管理器
- RM: 即 `javax.transaction.xa.XAResource` 我称之为与资源管理器的一个通信代表, 我们通过 `XAResource` 接口方法和资源管理器进行通信。

## 一、JTA

### 1.1 AP、TM、RM三大对象

TM: 即 `javax.transaction.TransactionManager`, 总体接口定义如下:

- `begin()`: 创建一个新的事务并关联到当前线程
- `Transaction getTransaction()`: 获取与当前线程关联的事务
- `commit()`: 提交与当前线程关联的事务
- `rollback()`: 回滚与当前线程关联的事务

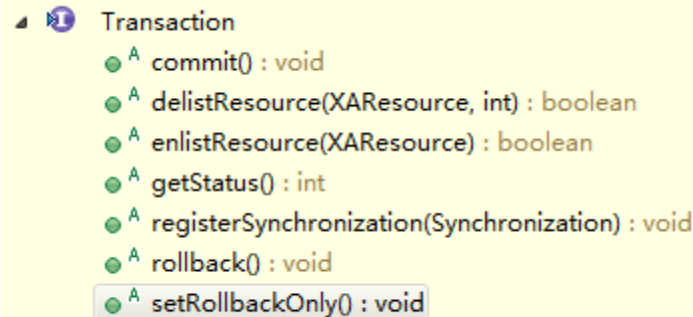
```
TransactionManager
  A begin() : void
  A commit() : void
  A getStatus() : int
  A getTransaction() : Transaction
  A resume(Transaction) : void
  A rollback() : void
  A setRollbackOnly() : void
  A setTransactionTimeout(int) : void
  A suspend() : Transaction
```

# 一、JTA

## 1.1 AP、TM、RM三大对象

javax.transaction.Transaction，接口如下所示：

- enlistResource(XAResource xaRes)：把给定的XAResource（资源管理器的一个通信代表）加入当前事务中来，一个分布式事务会涉及与多个资源管理器交互，该操作就是把某个资源管理器的通信代表纳入当前事务中来。
- delistResource(XAResource xaRes, int flag)：把给定的XAResource（资源管理器的一个通信代表）从当前事务中去除。



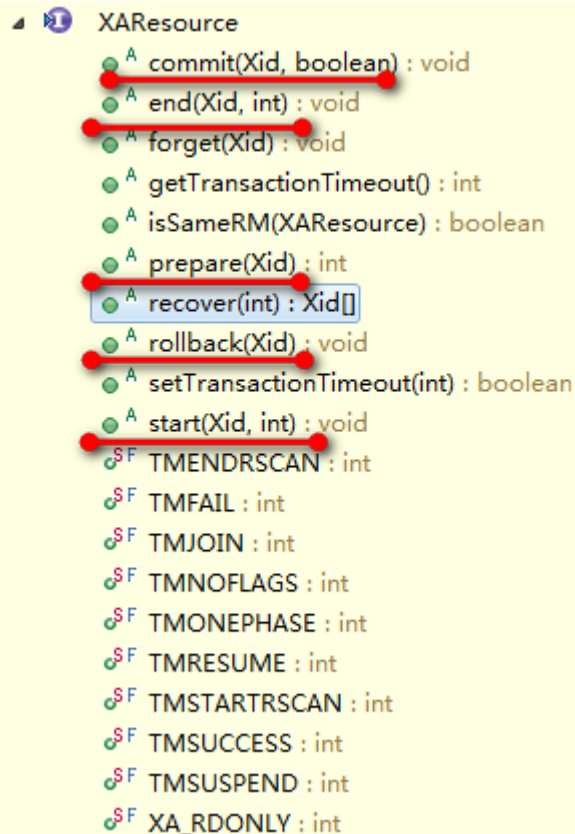
```
Transaction
  commit() : void
  delistResource(XAResource, int) : boolean
  enlistResource(XAResource) : boolean
  getStatus() : int
  registerSynchronization(Synchronization) : void
  rollback() : void
  setRollbackOnly() : void
```

# 一、JTA

## 1.1 AP、TM、RM三大对象

### RM: 资源管理器 `javax.transaction.xa.XAResource`

- `start(Xid xid, int flags)` 和 `end(Xid xid, int flags)`  
用于告知资源管理器的事务的边界，即在这两个方法之间的sql等操作才会纳入xid对应的分布式事务中
- `prepare(Xid xid)`：则就是上述图片中的第一个阶段，针对xid对应的分布式事务，向资源管理器发送预提交命令
- `rollback(Xid xid)`、`commit(Xid xid, boolean onePhase)`  
如果所有资源管理器都回复OK，则向所有资源管理器发送commit提交命令，否则发送rollback回滚命令



```
XAResource
  A commit(Xid, boolean) : void
  A end(Xid, int) : void
  A forget(Xid) : void
  A getTransactionTimeout() : int
  A isSameRM(XAResource) : boolean
  A prepare(Xid) : int
  A recover(int) : Xid[]
  A rollback(Xid) : void
  A setTransactionTimeout(int) : boolean
  A start(Xid, int) : void
  SF TMENDRSCAN : int
  SF TMFAIL : int
  SF TMJOIN : int
  SF TMNOFLAGS : int
  SF TMONEPHASE : int
  SF TMRESUME : int
  SF TMSTARTRSCAN : int
  SF TMSUCCESS : int
  SF TMSUSPEND : int
  SF XA_RDONLY : int
```

# 一、JTA

## 1.1 AP、TM、RM三大对象

```
xaRes1.start(xid, XAResource.TMNOFLAGS);
stmt1.executeUpdate("insert into test_table1 values (100)");
xaRes1.end(xid, XAResource.TMSUCCESS);

stmt1.executeUpdate("insert into test_table1 values (99)");

xaRes2.start(xid, XAResource.TMNOFLAGS);
stmt2.executeUpdate("insert into test_table2 values (100)");
xaRes2.end(xid, XAResource.TMSUCCESS);

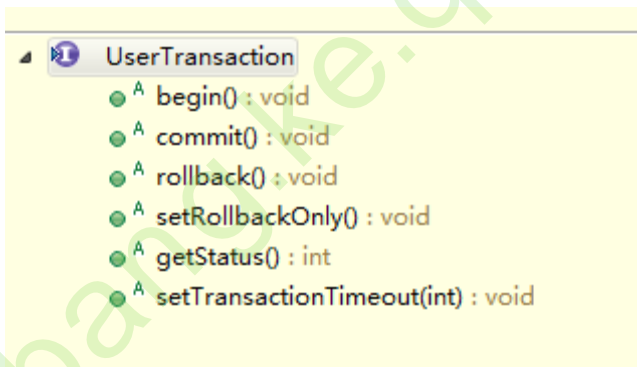
ret1 = xaRes1.prepare(xid);
ret2 = xaRes2.prepare(xid);
if (ret1 == XAResource.XA_OK && ret2 == XAResource.XA_OK) {
    xaRes1.commit(xid, false);
    xaRes2.commit(xid, false);
}else{
    xaRes1.rollback(xid);
    xaRes2.rollback(xid);
}
```



# 一、JTA

## 1.2 UserTransaction接口

**UserTransaction**接口是给开发人员使用的事务接口，屏蔽了底层的实现细节，通过该接口就可以操作一个分布式事务。该接口的实现通常是委托给TM即事务管理器来完成。接口内容如下：



## 二、实现JTA的框架

实现了JTA（模拟XA协议）的框架有：JOTM、Atomikos，推荐最多的是Atomikos。