**Transaction-lcn**

**CAP定律**

    这个定理的内容是指的是在一个分布式系统中、Consistency（一致性）、 Availability（可用性）、Partition tolerance（分区容错性），三者不可得兼。

**一致性（C）**

    在分布式系统中的所有数据备份，在同一时刻是否是同样的值。（等同于所有节点访问同一份最新的数据副本）

**可用性（A）**

    在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求。（对数据更新具备高可用性）

**分区容错性（P）无法避免-必须**

    以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择。

**BASE理论**

    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中一致性和可用性权衡的结果，其来源于对大规模互联网系统分布式实践的总结， 是基于CAP定理逐步演化而来的。BASE理论的核心思想是：即使无法做到强一致性，但每个应用都可以根据自身业务特点，采用适当的方式来使系统达到最终一致性。

**基本可用**

    基本可用是指分布式系统在出现不可预知故障的时候，允许损失部分可用性----注意，这绝不等价于系统不可用。比如：

（1）响应时间上的损失。正常情况下，一个在线搜索引擎需要在0.5秒之内返回给用户相应的查询结果，但由于出现故障，查询结果的响应时间增加了1~2秒

（2）系统功能上的损失：正常情况下，在一个电子商务网站上进行购物的时候，消费者几乎能够顺利完成每一笔订单，但是在一些节日大促购物高峰的时候，由于消费者的购物行为激增，为了保护购物系统的稳定性，部分消费者可能会被引导到一个降级页面

**软状态**

    软状态指允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据同步的过程存在延时

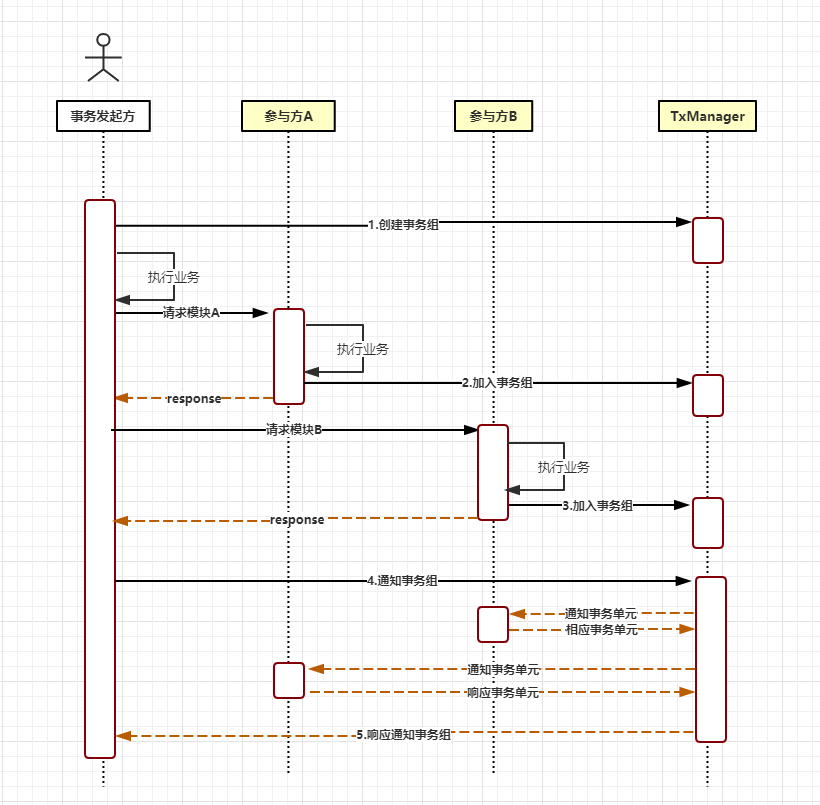
**最终一致性**

    最终一致性强调的是所有的数据副本，在经过一段时间的同步之后，最终都能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。

## 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。事务发起方或者参与方都由TxClient端来控制。

原理图:



#### 核心步骤

* 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。
* 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。
* 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。

**LCN事务模式**

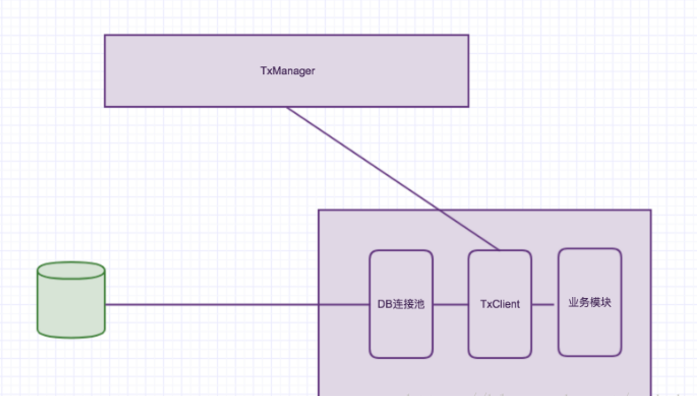
**一、原理介绍**

    LCN模式是通过代理Connection的方式实现对本地事务的操作，然后再由TxManager统一协调控制事务。当本地事务提交回滚或者关闭连接时将会执行假操作，该代理的连接将由LCN连接池管理。

**二、模式特点**

* 该模式对代码的嵌入性为低。
* 该模式仅限于本地存在连接对象且可通过连接对象控制事务的模块。
* 该模式下的事务提交与回滚是由本地事务方控制，对于数据一致性上有较高的保障。
* 该模式缺陷在于代理的连接需要随事务发起方一共释放连接，增加了连接占用的时间。

**三、事务协调机制**



如图：假设服务已经执行到关闭事务组的过程，那么接下来作为一个模块执行通知给TxManager，然后告诉他本次事务已经完成。那么如图中Txmanager下一个动作就是通过事务组的id，然后获取到本次事务组的事务信息；然后查看一下对应有哪几个模块参与，然后如果是有A/B/C三个模块；那么对应的对三个模块做通知、提交、回滚。

那么提交的时候是提交给谁呢？

是提交给了我们的TxClient模块。然后TxCliient模块下有一个连接池，就是框架自定义的一个连接池（如图DB连接池）；这个连接池其实就是在没有通知事务之前一直占有着这次事务的连接资源，就是没有释放。但是他在切面里面执行了close方法。在执行close的时候。如果需要（TxManager）分布式事务框架的连接。他被叫做“假关闭”,也就是没有关闭，只是在执行了一次关闭方法。实际的资源是没有释放的。这个资源是掌握在LCN的连接池里的。

然后当TxManager通知提交或事务回滚的时候呢？

TxManager会通知我们的TxClient端。然后TxClient会去执行相应的提交或回滚。提交或回滚之后再去关闭连接，然后在返回给DB连接池。这就只事务的协调机制。说白了就是代理DataSource的机制；相当于是拦截了一下连接池，控制了连接池的事务提交。

**四、补偿机制**

为什么需要事务补偿？

事务补偿是指在执行某个业务方法时，本应该执行成功的操作却因为服务器挂机或者网络抖动等问题导致事务没有正常提交，此种场景就需要通过补偿来完成事务，从而达到事务的一致性。

补偿机制的触发条件？

当执行关闭事务组步骤时，若发起方接受到失败的状态后将会把该次事务识别为待补偿事务，然后发起方将该次事务数据异步通知给TxManager。TxManager接受到补偿事务以后先通知补偿回调地址，然后再根据是否开启自动补偿事务状态来补偿或保存该次切面事务数据。

补偿事务机制？

LCN的补偿事务原理是模拟上次失败事务的请求，然后传递给TxClient模块然后再次执行该次请求事务。

简单的说：lcn事务补偿是在在服务挂机和网络抖动情况下；服务挂机是指在完成三个核心步骤的时候尤其也只有最后一步关闭事务组时，让我去执行关闭事务组的时候，比如本次事务是要提交的。txManager接收到提交的请求再去通知的时候发现通知不到了（通知不到也就两种原因服务挂了和网络出问题）在这种情况下TxManager会做一个标示;然后返回给发起方。告诉他本次事务有存在没有通知到的情况。

那么如果是接收到这个信息之后呢，发起方就会做一个标示，标示本次事务是需要补偿事务的。这就是事务补偿机制。

LCN是怎么去实现事务补偿呢？

首先他或根据发起方拿到的TxManager的标示之后，判断是否需要做事务补偿，如果需要，他会首先在本地去记录一下日志 ，然后再把本次切面的信息，也就是发起方本次事务切面的信息

以及事务组的id信息提交给TxManager;然后TxManager接收到这次数据之后，他会查询到本次事务组的整个信息，获取到本次事务组信息之后呢，他会把这些信息一块保存到redis下，也就是做为补偿数据一块存下来。

在存下来的时候，他会先执行一次叫做回调接口的请求；这个回调接口其实是指的是回调给第三方服务的一个地址，也就是我们自己的服务地址。这里是作为一次通知用的，我们来看下4.0的界面

当补偿发生之后，TxManager记录完数据以后会通知这个回调接口地址如图：

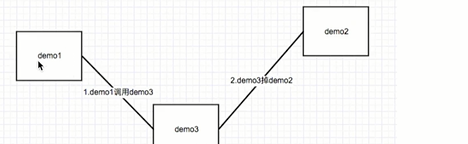


告诉你有补偿信息存在。这个地方我们就可以做一些通知，例如邮件、[短信提醒](https://www.baidu.com/s?wd=%E7%9F%AD%E4%BF%A1%E6%8F%90%E9%86%92&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)功能，通知给相应的人员。让他们知道现在我们的服务里存在了补偿，要及时处理。



当然你也可以开启自动补偿功能（如上图），当开启自动补偿之后的话，当补偿上传上来他也会通知的上面补偿回调接口，通知完之后，他会把本次事务去补偿一下。他是如何执行的呢？上面说到的他会把切面的信息上传上来，他会把切面的信息

上传给发起方，传递给发起方以后让发起方重复执行本次事务。但是有一个差异性的地方是在于，他在去做模块提交的时候呢，会根据历史的提交数据（补偿数据）做一个逆向的操作，也就是说，必须如下图来说：



比如我们这三个模块demo1、demo2、demo3.那么现在如果通知demo2通知成功，是在指没有补偿之前（也就是正常执行的事务状态下）。在通知demo3失败了。然后这一次作为补偿通知给了发起方demo1，发起方调用TxManager，告诉他这一次需要补偿

那么TxManager下发信息给demo1，demo1执行补偿的时候呢，首先启动方事务是要回滚的（异常的状态下不会执行补偿）；那么demo2、demo3是否回滚取决于上次的事务请求。上面说demo2提交成功了，也就是说他要回滚，demo3失败，他是要补偿的，是要提交的。那么怎么去实现这一点呢？还是跟以前一样，就是说其实这次是与之前的执行事务流程是相同的；唯一不同的是在于执行关闭，就是在这次补偿的事务过程中的关闭事务的时候。

TxManager会判断历史数据，然后在下发数据的时候，会根据历史数据（补偿数据）做差异性的通知，是判断谁要需要提交，谁需要回滚。这就是补偿机制的原理。

## TCC事务模式

### 一、原理介绍

    TCC事务机制相对于传统事务机制（X/Open XA Two-Phase-Commit），其特征在于它不依赖资源管理器(RM)对XA的支持，而是通过对（由业务系统提供的）业务逻辑的调度来实现分布式事务。主要由三步操作，Try: 尝试执行业务、 Confirm:确认执行业务、 Cancel: 取消执行业务。

### 二、模式特点

* 该模式对代码的嵌入性高，要求每个业务需要写三种步骤的操作。
* 该模式对有无本地事务控制都可以支持使用面广。
* 数据一致性控制几乎完全由开发者控制，对业务开发难度要求高。

## TXC事务模式

### 一、原理介绍

    TXC模式命名来源于淘宝，实现原理是在执行SQL之前，先查询SQL的影响数据，然后保存执行的SQL快走信息和创建锁。当需要回滚的时候就采用这些记录数据回滚数据库，目前锁实现依赖redis分布式锁控制。

### 二、模式特点

* 该模式同样对代码的嵌入性低。
* 该模式仅限于对支持SQL方式的模块支持。
* 该模式由于每次执行SQL之前需要先查询影响数据，因此相比LCN模式消耗资源与时间要多。
* 该模式不会占用数据库的连接资源。

## 使用

### Center-eureka

服务注册中心，tm，tc都注册到注册中心。

### Lcn-tm

#### 引入pom依赖

<dependency>

<groupId>com.codingapi.txlcn</groupId>

<artifactId>txlcn-tm</artifactId>

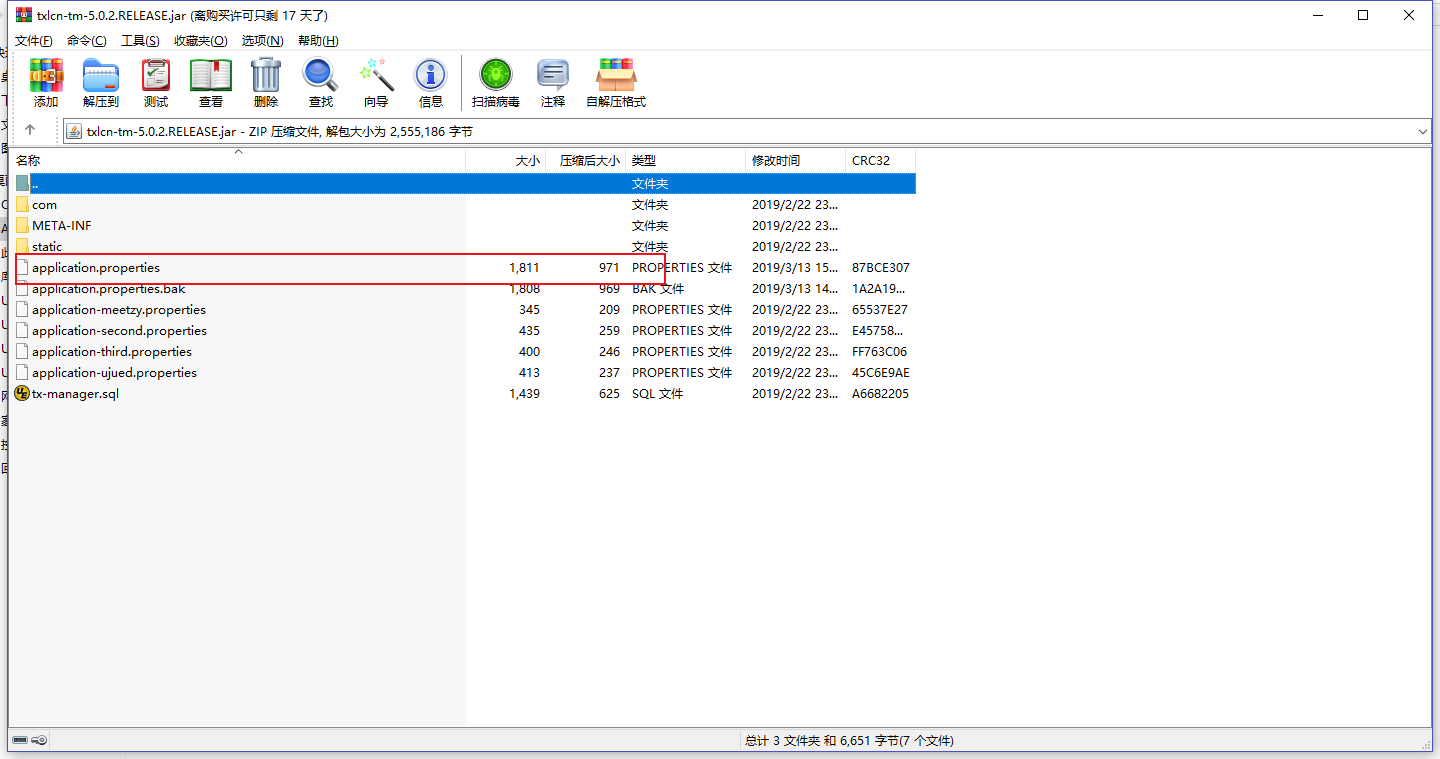
<version>5.0.2.RELEASE</version>

</dependency>

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

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

#### TM修改jar包配置



#### application.properties配置说明

**spring.application.name**=TransactionManager

**server.port**=7970

*# JDBC 数据库配置*

**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**=123456

*# 数据库方言*

**spring.jpa.database-platform**=org.hibernate.dialect.MySQL5InnoDBDialect

*# 第一次运行可以设置为: create, 为TM创建持久化数据库表*

**spring.jpa.hibernate.ddl-auto**=validate

*# TM监听IP. 默认为 127.0.0.1*

**tx-lcn.manager.host**=127.0.0.1

*# TM监听Socket端口. 默认为 ${server.port} - 100*

**tx-lcn.manager.port**=8070

*# 心跳检测时间(ms). 默认为 300000*

**tx-lcn.manager.heart-time**=300000

*# 分布式事务执行总时间(ms). 默认为36000*

**tx-lcn.manager.dtx-time**=8000

*# 参数延迟删除时间单位ms 默认为dtx-time值*

**tx-lcn.message.netty.attr-delay-time**=${tx-lcn.manager.dtx-time}

*# 事务处理并发等级. 默认为机器逻辑核心数5倍*

**tx-lcn.manager.concurrent-level**=160

*# TM后台登陆密码，默认值为codingapi*

**tx-lcn.manager.admin-key**=codingapi

*# 分布式事务锁超时时间 默认为-1，当-1时会用tx-lcn.manager.dtx-time的时间*

**tx-lcn.manager.dtx-lock-time**=${tx-lcn.manager.dtx-time}

*# 雪花算法的sequence位长度，默认为12位.*

**tx-lcn.manager.seq-len**=12

*# 异常回调开关。开启时请制定ex-url*

**tx-lcn.manager.ex-url-enabled**=false

*# 事务异常通知（任何http协议地址。未指定协议时，为TM提供内置功能接口）。默认是邮件通知*

**tx-lcn.manager.ex-url**=/provider/email-to/\*\*\*@\*\*.com

*# 开启日志,默认为false*

**tx-lcn.logger.enabled**=true

**tx-lcn.logger.enabled**=false

**tx-lcn.logger.driver-class-name**=${spring.datasource.driver-class-name}

**tx-lcn.logger.jdbc-url**=${spring.datasource.url}

**tx-lcn.logger.username**=${spring.datasource.username}

**tx-lcn.logger.password**=${spring.datasource.password}

*# redis 的设置信息. 线上请用Redis Cluster*

**spring.redis.host**=127.0.0.1

**spring.redis.port**=6379

**spring.redis.password**=

注意（NOTE）

(1) TxManager所有配置均有默认配置，请按需覆盖默认配置。

(2) 特别注意 TxManager进程会监听两个端口号，一个为TxManager端口，另一个是事务消息端口。TxClient默认连接事务消息端口是8070， 所以，为保证TX-LCN基于默认配置运行良好，请设置TxManager端口号为8069 或者指定事务消息端口为8070

(3) 分布式事务执行总时间 a 与 TxClient通讯最大等待时间 b、TxManager通讯最大等待时间 c、微服务间通讯时间 d、微服务调用链长度 e 几个时间存在着依赖关系。 a >= 2c + (b + c + d) \* (e - 1), 特别地，b、c、d 一致时，a >= (3e-1)b。你也可以在此理论上适当在减小a的值，发生异常时能更快得到自动补偿，即 a >= (3e-1)b - Δ（[原因](http://www.txlcn.org/zh-cn/docs/fqa.html)）。 最后，调用链小于等于3时，将基于默认配置运行良好

(4) 若用tx-lcn.manager.ex-url=/provider/email-to/xxx@xx.xxx 这个配置，配置管理员邮箱信息(如QQ邮箱)：

**spring.mail.host**=smtp.qq.com

**spring.mail.port**=587

**spring.mail.username**=xxxxx@\*\*.com

**spring.mail.password**=\*\*\*\*\*\*\*\*\*

#### TM的准备环境

1.安装TM需要依赖的中间件： JRE1.8+, Mysql5.6+, Redis3.2+

如果需要手动编译源码， 还需要Git, Maven, JDK1.8+

2.创建MySQL数据库, 名称为: tx-manager

3.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

#### 4.在主类上标注 @EnableTransactionManagerServer

**@SpringBootApplication**

**@EnableTransactionManagerServer**

**public** **class** **TransactionManagerApplication** {

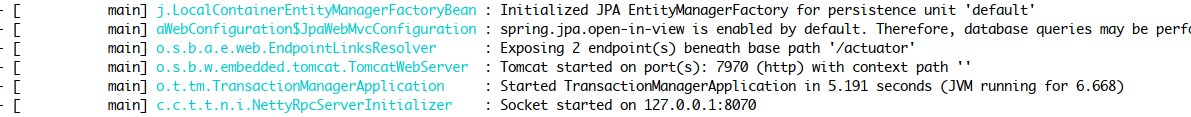
**public** **static** **void** **main**(String[] args) {

SpringApplication.run(TransactionManagerApplication.class, args);

}

}

#### 5.运行



### Lcn-tc

**共三个模块**  
Txlcn-demo-spring-service-a (发起方 | LCN模式)

Txlcn-demo-spring-service-b (参与方 | TXC模式)

Txlcn-demo-spring-service-c (参与方 | TCC模式)

三个模块依赖txlcn-common

#### 引入pom依赖

<dependency>

<groupId>com.codingapi.txlcn</groupId>

<artifactId>txlcn-tc</artifactId>

<version>5.0.2.RELEASE</version>

</dependency>

<dependency>

<groupId>com.codingapi.txlcn</groupId>

<artifactId>txlcn-txmsg-netty</artifactId>

<version>5.0.2.RELEASE</version>

</dependency>

#### application.properties配置说明

*# 是否启动LCN负载均衡策略(优化选项，开启与否，功能不受影响)*

**tx-lcn.ribbon.loadbalancer.dtx.enabled**=true

*# tx-manager 的配置地址，可以指定TM集群中的任何一个或多个地址*

*# tx-manager 下集群策略，每个TC都会从始至终<断线重连>与TM集群保持集群大小个连接。*

*# TM方，每有TM进入集群，会找到所有TC并通知其与新TM建立连接。*

*# TC方，启动时按配置与集群建立连接，成功后，会再与集群协商，查询集群大小并保持与所有TM的连接*

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

*# 该参数是分布式事务框架存储的业务切面信息。采用的是h2数据库。绝对路径。该参数默认的值为{user.dir}/.txlcn/{application.name}-{application.port}*

**tx-lcn.aspect.log.file-path**=logs/.txlcn/demo-8080

*# 调用链长度等级，默认值为3（优化选项。系统中每个请求大致调用链平均长度，估算值。）*

**tx-lcn.client.chain-level**=3

*# 该参数为tc与tm通讯时的最大超时时间，单位ms。该参数不需要配置会在连接初始化时由tm返回。*

**tx-lcn.client.tm-rpc-timeout**=2000

*# 该参数为分布式事务的最大时间，单位ms。该参数不允许TC方配置，会在连接初始化时由tm返回。*

**tx-lcn.client.dtx-time**=8000

*# 该参数为雪花算法的机器编号，所有TC不能相同。该参数不允许配置，会在连接初始化时由tm返回。*

**tx-lcn.client.machine-id**=1

*# 该参数为事务方法注解切面的orderNumber，默认值为0.*

**tx-lcn.client.dtx-aspect-order**=0

*# 该参数为事务连接资源方法切面的orderNumber，默认值为0.*

**tx-lcn.client.resource-order**=0

*# 是否开启日志记录。当开启以后需要配置对应logger的数据库连接配置信息。*

**tx-lcn.logger.enabled**=false

**tx-lcn.logger.driver-class-name**=${spring.datasource.driver-class-name}

**tx-lcn.logger.jdbc-url**=${spring.datasource.url}

**tx-lcn.logger.username**=${spring.datasource.username}

**tx-lcn.logger.password**=${spring.datasource.password}

#### 特别配置

##### 1、微服务集群且用到 LCN事务模式时，为保证性能请开启TX-LCN重写的负载策略。

* Dubbo 开启

**@Reference**(version = "${demo.service.version}",

application = "${dubbo.application.e}",

retries = -1,

registry = "${dubbo.registry.address}",

loadbalance = "txlcn\_random") *// here*

**private** EDemoService eDemoService;

* SpringCloud 开启 (application.properties)

**tx-lcn.springcloud.loadbalance.enabled**=true

##### 2、关闭业务RPC重试

* Dubbo 开启

**@Reference**(version = "${demo.service.version}",

application = "${dubbo.application.e}",

retries = -1,

registry = "${dubbo.registry.address}",

loadbalance = "txlcn\_random") *// here*

**private** EDemoService eDemoService;

* SpringCloud 开启 (application.properties)

*# 关闭Ribbon的重试机制*

**ribbon.MaxAutoRetriesNextServer**=0

NOTE  
1、TxClient所有配置均有默认配置，请按需覆盖默认配置。  
2、为什么要关闭服务调用的重试。远程业务调用失败有两种可能： （1），远程业务执行失败 （2）、远程业务执行成功，网络失败。对于第2种，事务场景下重试会发生，某个业务执行两次的问题。 如果业务上控制某个事务接口的幂等，则不用关闭重试。

##### 3、通过AOP配置本地事务与分布式事务

@Configuration

@EnableTransactionManagement

public class TransactionConfiguration {

/\*\*

\* 本地事务配置

\* @param transactionManager

\* @return

\*/

@Bean

@ConditionalOnMissingBean

public TransactionInterceptor transactionInterceptor(PlatformTransactionManager transactionManager) {

Properties properties = new Properties();

properties.setProperty("\*", "PROPAGATION\_REQUIRED,-Throwable");

TransactionInterceptor transactionInterceptor = new TransactionInterceptor();

transactionInterceptor.setTransactionManager(transactionManager);

transactionInterceptor.setTransactionAttributes(properties);

return transactionInterceptor;

}

/\*\*

\* 分布式事务配置 设置为LCN模式

\* @param dtxLogicWeaver

\* @return

\*/

@ConditionalOnBean(DTXLogicWeaver.class)

@Bean

public TxLcnInterceptor txLcnInterceptor(DTXLogicWeaver dtxLogicWeaver) {

TxLcnInterceptor txLcnInterceptor = new TxLcnInterceptor(dtxLogicWeaver);

Properties properties = new Properties();

properties.setProperty(Transactions.DTX\_TYPE,Transactions.LCN);

properties.setProperty(Transactions.DTX\_PROPAGATION, "REQUIRED");

txLcnInterceptor.setTransactionAttributes(properties);

return txLcnInterceptor;

}

@Bean

public BeanNameAutoProxyCreator beanNameAutoProxyCreator() {

BeanNameAutoProxyCreator beanNameAutoProxyCreator = new BeanNameAutoProxyCreator();

//需要调整优先级，分布式事务在前，本地事务在后。

beanNameAutoProxyCreator.setInterceptorNames("txLcnInterceptor","transactionInterceptor");

beanNameAutoProxyCreator.setBeanNames("\*Impl");

return beanNameAutoProxyCreator;

}

}

##### 4、TXC模式定义表的实际主键

TXC 是基于逆向sql的方式实现对业务的回滚控制，在逆向sql操作数据是会检索对应记录的主键作为条件处理回滚业务。但是在有些情况下可能表中并没有主键字段(primary key)，仅存在业务上的名义主键，此时可通过重写PrimaryKeysProvider方式定义表对应的主键关系。

如下所示:

**@Component**

**public** **class** **MysqlPrimaryKeysProvider** **implements** **PrimaryKeysProvider** {

**@Override**

**public** Map<String, List<String>> provide() {

*//t\_demo 表的回滚主键为 kid字段*

**return** Maps.newHashMap("t\_demo", Collections.singletonList("kid"));

}

}

##### 5、TC模块标识策略

TC模块在负载时，TM为了区分具体模块，会要求TC注册时提供唯一标识。默认策略是，应用名称加端口方式标识。也可以自定义，自定义需要保证各个模块标识不能重复。

**@Component**

**public** **class** **MyModIdProvider** **implements** **ModIdProvider** {

**@Override**

**public** String **modId**() {

**return** ip + port;

}

}

#### 启动类

**@SpringBootApplication**

**@EnableDiscoveryClient**

**@EnableDistributedTransaction**

**public** **class** **SpringServiceAApplication** {

**public** **static** **void** **main**(String[] args) {

SpringApplication.run(SpringServiceAApplication.class, args);

}

}

#### 启动SpringCloud微服务

事务参与方 ServiceB  
spring-b事务参与方 ServiceC  
spring-c事务发起方 ServiceA  
spring-a

## 测试

### 正常提交事务

访问 发起方提供的Rest接口 /txlcn?value=the-value。发现事务全部提交  
result

### 回滚事务 访问 发起方提供的Rest接口 /txlcn?value=the-value&ex=throw。发现发起方由本地事务回滚，而参与方ServiceB、ServiceC，由于TX-LCN的协调，数据也回滚了。 error_result

详细见demo：micro.zip

参考：http://www.txlcn.org/zh-cn/docs/demo/env.html