分布式事务 第一天

学习目标

目标1 理解本地事务和分布式事务的概念和区别(理解)

目标2 分布式事务的理论基础(了解)

目标3 各分布式事务方案的优缺点及适用场景(知道)

目标4 如何根据自己的业务场景选择合适的分布式事务方案(掌握)

目标5 Atomikos+jta实现分布式事务(掌握)

目标6 fescar实现分布式事务(掌握)

目标7 TX-LCN解决分布式事务(掌握)

目标8 RocketMQ事务消息(掌握)

第1章 本地事务和分布式事务的概念和区别

1.1 事务概念回顾目标:

目标

1.什么是事务?

2.事务的四个特性

1.1.1什么是事务:

1.1.2事务的四个特性ACID：(四字总结)

原子性(Atomicity):同生共死(要么全部成功,要么全部失败)

一致性(Consistency):== 相互抵消==(卖方+200元和买方-200元,)

隔离性(Isolation)：并行不扰:理论上,并行的两个事物,相互之间是不干扰的.

持久性(Durability)：落子无悔(不可逆)

1.3 事务的隔离级别及可能发生的情况以及解决方案

1.3.1目标:

1.知道两组事务并发时隔离性理论情况和现实状况的区别:

隔离性:a-独占库,独占表;b独占库,独占表. a和b同时操作数据库中某一个表.

2.事务的隔离级别

读未提交\读已提交\可重复读\串行化

脏读：事务a,读到了事务b未提交的数据.

不可重复读：可重复读(理想的状态),

虚读（幻读）**：一个事务在查询,另一个事务在做插入或者删除,此时就会出现幻读.

例子:用户a,要给价格>100且<200的数据,更新状态,若在第一时间查到10条数据;

用户b,在同一时间删除了一条数据,

此时用户a,再对数据更新时,就只能更新9条数据,此时便出现了幻读.

脏读和幻读的区别：

脏读:一个事务更新,另一个事务在读取

幻读:一个事务在做插入或者删除,另一个事务在做读取时.

四种隔离级别起的作用,如何解决隔离性问题:

2	读已提交	read committed	否	是	是	Oracle 和 SQL Server
3	可重复读	repeatable read	否	否	是	MySQL
4	串行化	serializable	否	否	否	

隔离级别与性能的关系

隔离级别越高,性能越低.

小结:

有哪三种并发访问问题

脏读

不可重复读

幻读

哪四种隔离级别?能解决什么问题?

读未提交

读已提交

可重复读

串行化

1.2 本地事务(回顾)

目标:

1.什么是本地事务?(单\单\原子组)

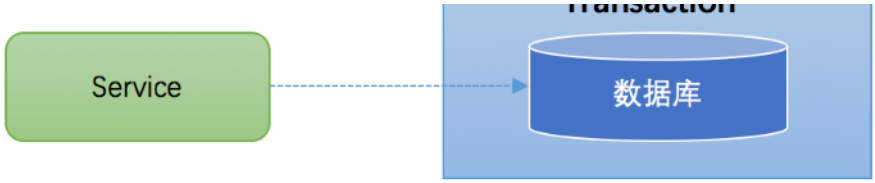
本质:单个数据库,单个会话,原子性

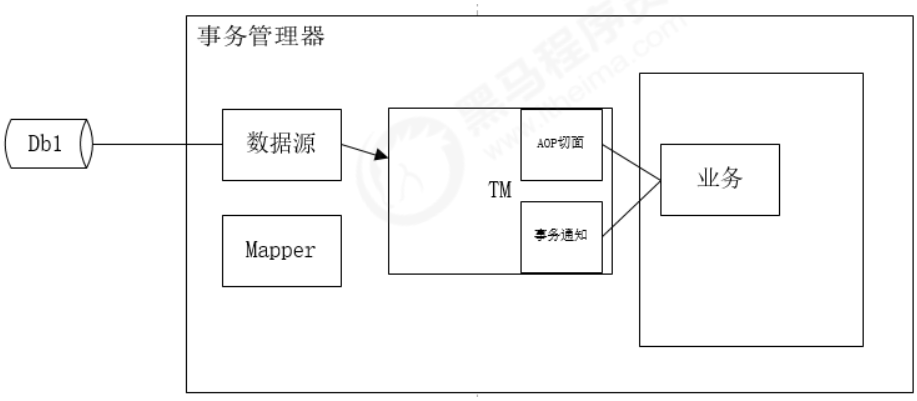
2.如何实现本地事务?(jdbc\spring)

手动事务编程和自动事务编程

3.spring如何实现本地事务

aop实现(面向切面编程)





spring配置事务方式及步骤:

方式:

aop'实现

xml\注解形式(@Transactionl)

步骤:

1.配置数据源

2.配置事务管理器,跟数据源匹配

3.切面和切入点(决定给哪些业务\什么方法配置事务)

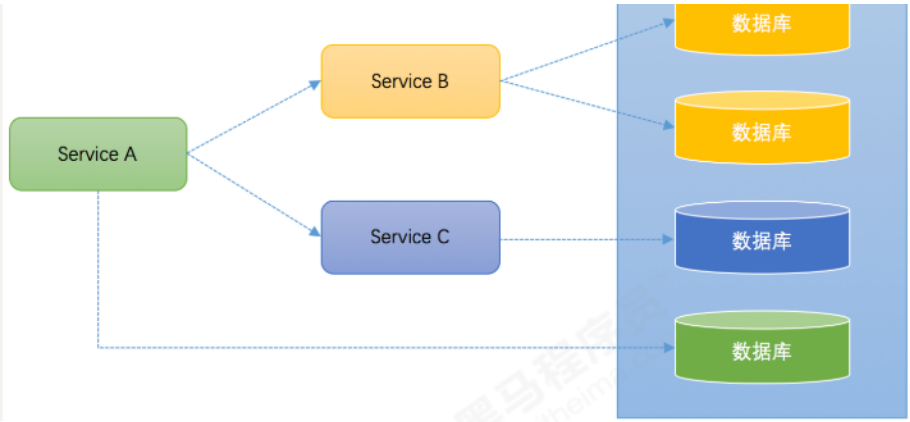
4.配置事务通知

5.将切入点和事务通知联系起来

1.3 什么是分布式事务

看图说话:

事务的参与者\服务器\数据源\事务管理器,分布在不同的节点上.原子性操作,要么全部成功,要么全部失败.



目标:

1.理解分布式事务概念

2.理解分布式事务本质

3.解决办法

概念:

事务的参与者\服务器\数据源\事务管理器,分布在不同的节点上.原子性操作,要么全部成功,要么全部失败.

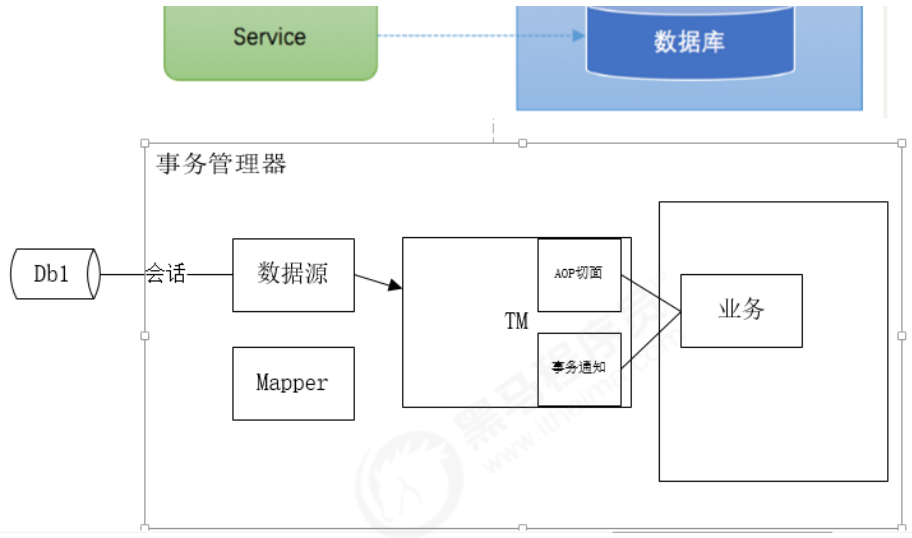
本质:

- 对于不同数据库,要求达到最终的一致性。

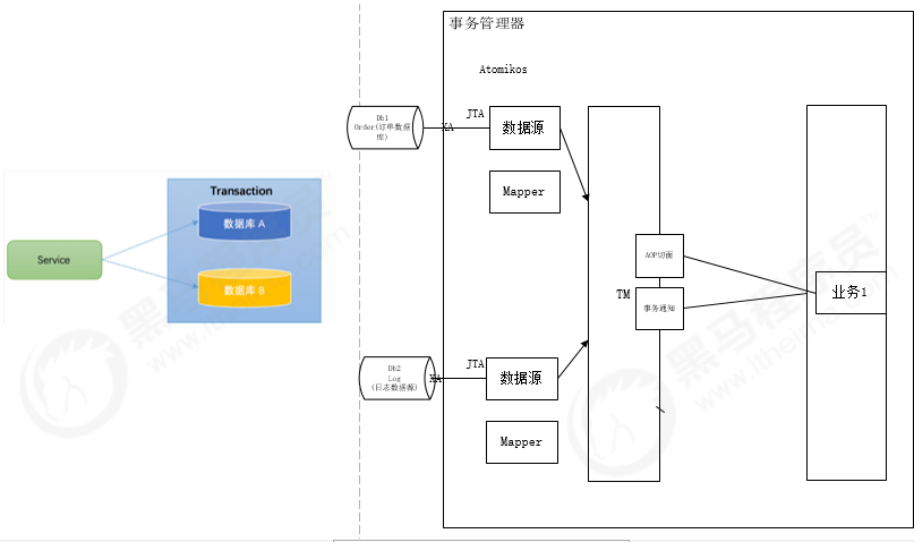
事务类别	数据库	会话	类比	事务管理器
本地事务	单数据库	单会话	单机游戏	本地事务管理器
分布式事务	多数据源	多个会话	斗地主()	全局事务管理器

配图分析解决办法:

本地事务



分布式事务架构图(解决方法:使用同一的全局事务管理器管理两遍的本地事务)



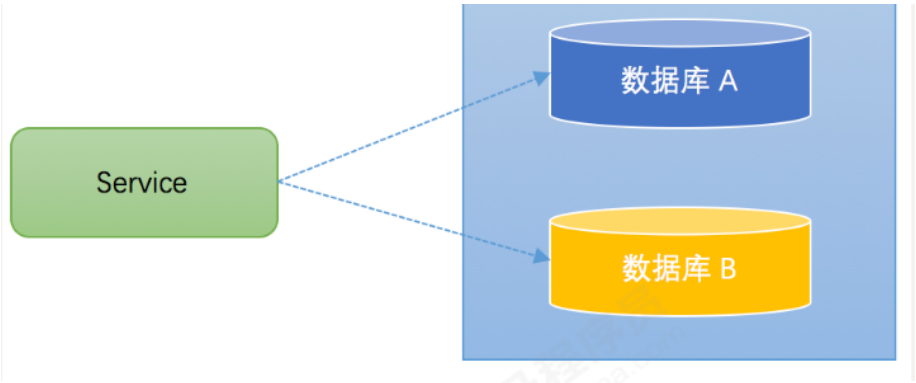
1.4 分布式事务应用架构

目标:

- 知道三种分布式服务架构
- 知道每种分布式服务架构的解决方案
- 知道柔性事务和刚性事务的区别

1.4.1 单一服务分布式事务(重点讨论) atomikos

单服务-多库。



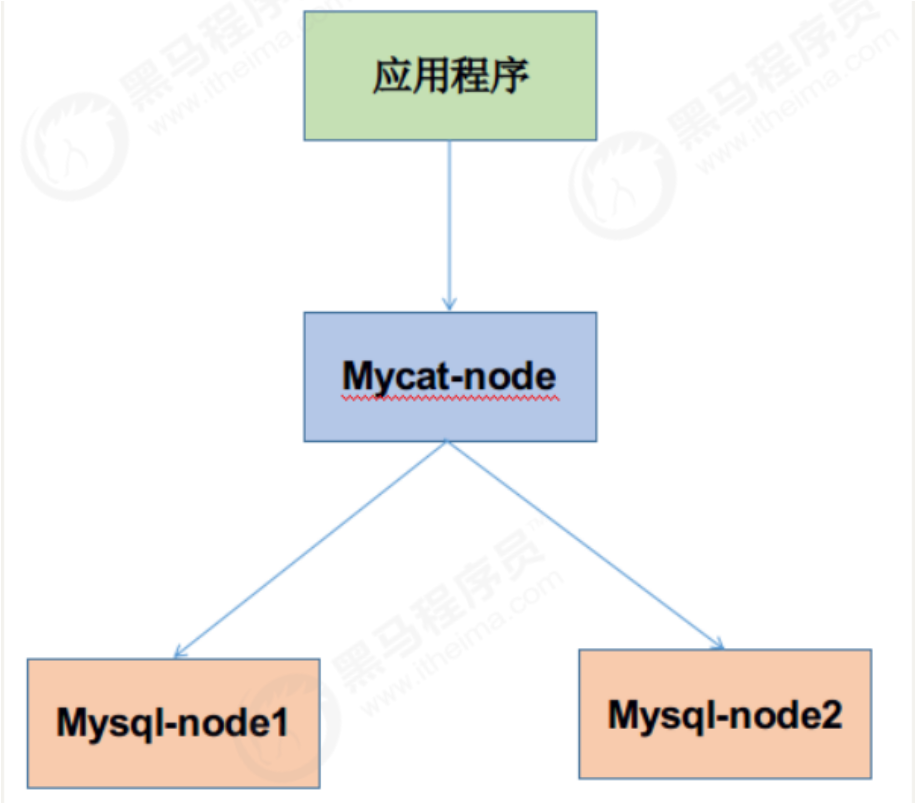
解决方案:(全局事务管理器TM)

1.4.2 分库分表(不做重点讨论)

数据量大时,分库分表

解决方案:中间件+数据库同步技术

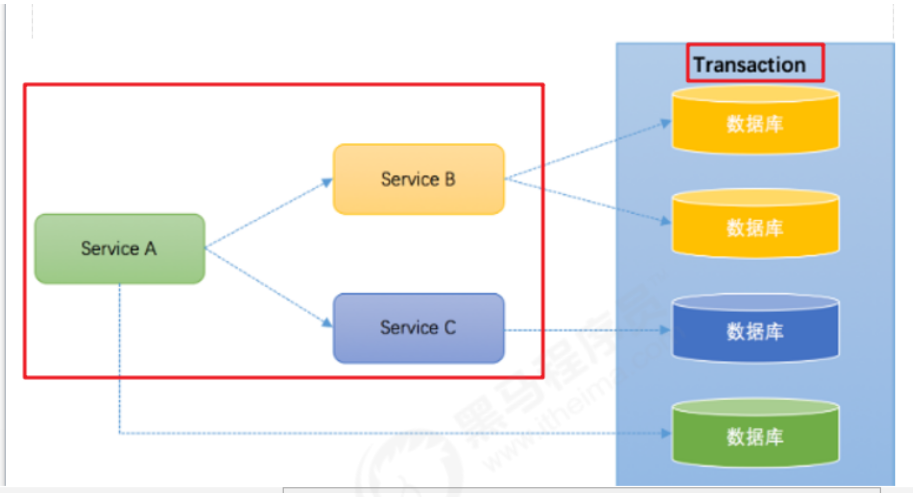
分库分表中间件:mycat \shardingdb



1.4.3 多服务多数据源分布式事务(重点掌握)

多数据库--多服务:

解决方案:全局TM(数据库) 和 服务协调者(服务),共同协调\数据库资源\服务



1.4.4 解决方案的分类

刚性事务:ACID(本地事务)

柔性事务:(CAP\BASE)(分布式事务管理)

比较:

事务类型	时间要求	一致性要求	应用类型	场景
刚性事务	立刻\马上	强一致性	局域网\企业应用	订单\日志
柔性事务	有时间弹性	最终一致性	互联网应用	付款\订单\收货

1.5 CAP理论

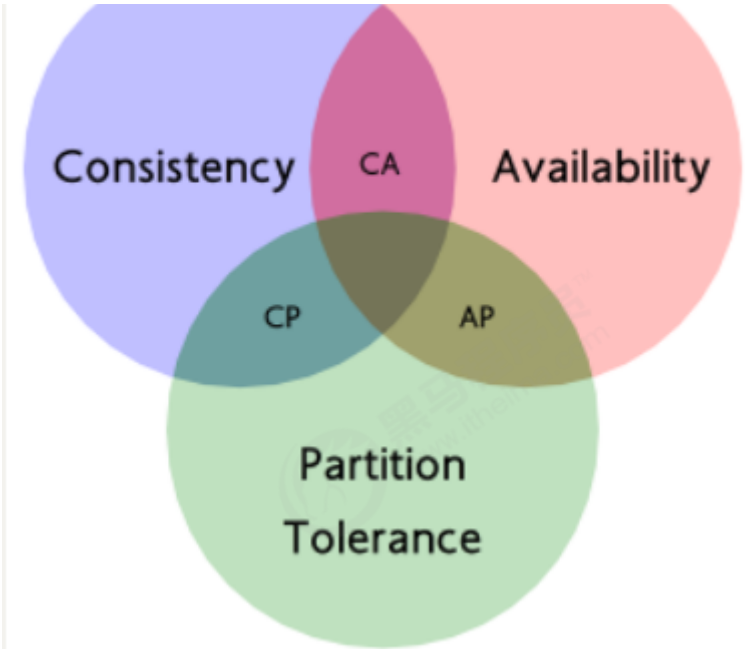
目标:

- 1.CAP是什么
- 2.CAP理论的结论是什么

定义:数据一致性(consistency)、服务可用性(availability)、分区容错性(partition-tolerance)

结论:只能满足其中2个\P是必须满足的 , cap中,最多只有两个条件能成立.

p必须要保证. cp \ ap



C (一致性)：

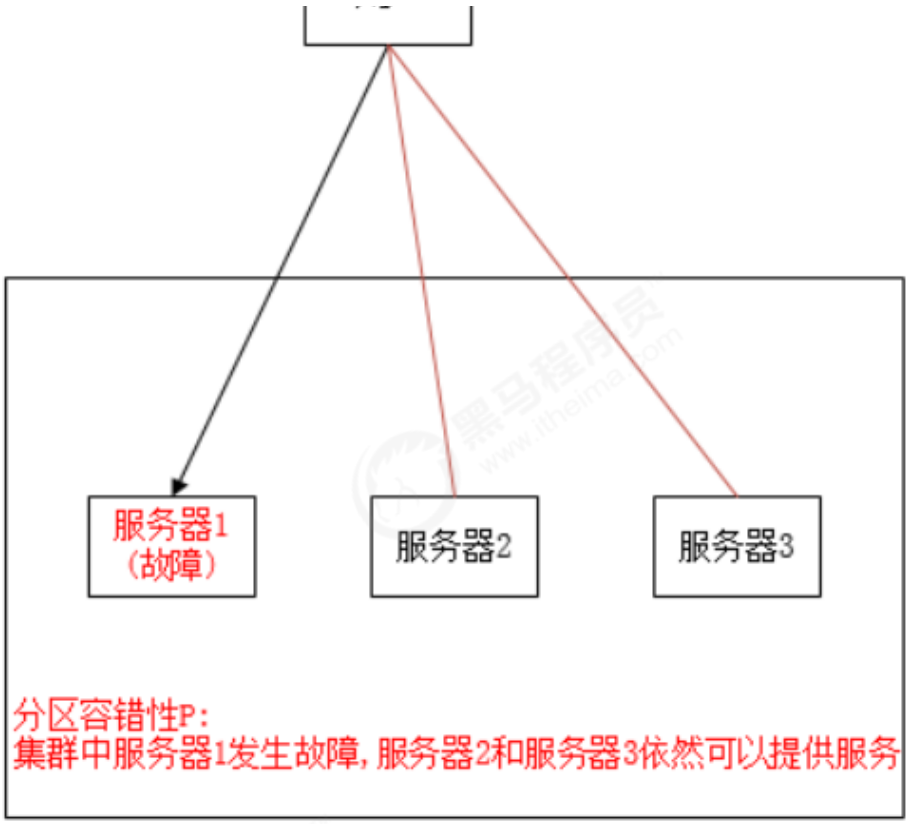
A (可用性)：非故障的节点在合理的时间内返回合理的响应.

合理的时间:三秒钟(合理)\两小时(不合理)

合理的相应:正在处理,请稍后(合理)\ 出现空指针对象(不合理)

P (分区容错性)：集群可用性。

网络无法 **100%** 可靠，分区其实是一个必然现象。**p**必须保证



CA无法保证:

如果我们选择了 **CA** 而放弃了 **P**，那么当发生分区现象时，为了保证一致性，这个时候必须拒绝请求，但是 **A** 又不允许，所以分布式系统理论上不可能选择 **CA** 架构，只能选择 **CP** 或者 **AP** 架构。

CP应用， Zookeeper 。

AP 来说，放弃强一致性(这里说的一致性是强一致性)， BASE理论的基础

ap例子:下单,购买了一件商品,不要求马上最库存进行扣减,但只要最终库存扣减即可.

总结:

CAP:一致性\可用性\分区容错性(集群可用性)

cp: zookeeper

ap: 最终一致性(BASE)

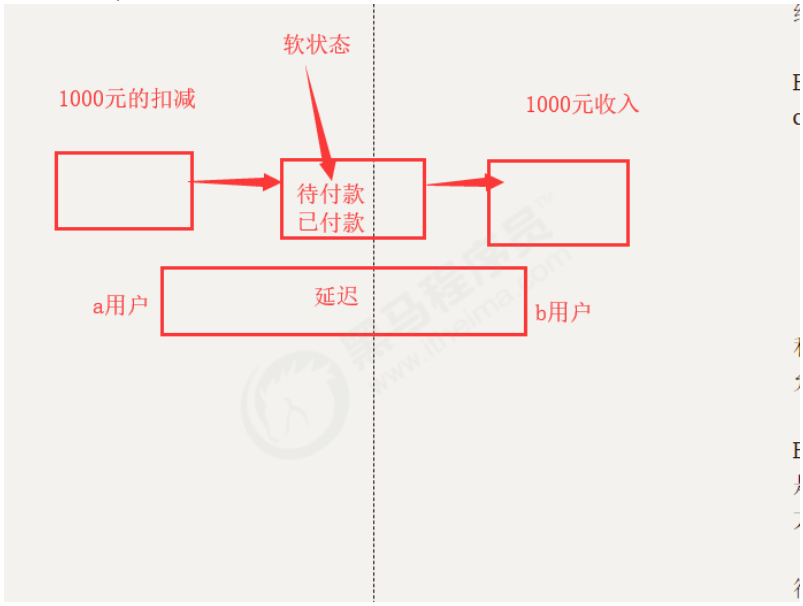
1.6 BASE理论

1.6.1目标:

- 1.什么是BASE理论
- 2.什么是柔性事务

核心思想:分布式事务,只需达到最终一致性（Eventual Consitency）即可。

结束)

- Eventually consistent（最终一致性）
- 

1.7 柔性事务解决方案

概念:符和BASE理论的分布式解决方案,就叫做柔性事务

1.典型的柔性事务方案如下:(重点)

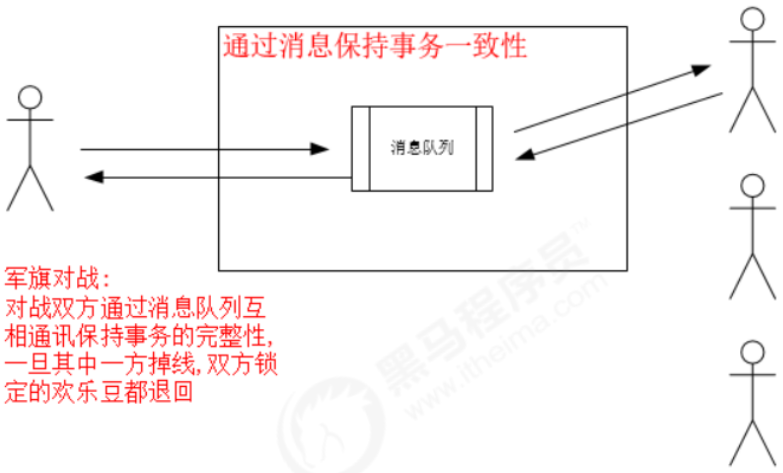
```
1) TCC（两阶段型、补偿型）  案例:(定婚-领证)
2) 可靠消息最终一致性（异步确保型）消息队列来保证事务的一致性

- 非事务型消息中间件(activimq\rabbitmq\kafka)
- 事务性消息rocketmq(阿里的消息队列)
- 案例:(在线下棋)

3)最大努力通知（非可靠消息、定期校对）

- 案例:滴滴打车有未付款的订单,每个一段时间滴滴都会通知我们付款.
      滴滴最大努力通知用户;用户定期校验.如果未付款,则执行付款,付款后
滴滴不再重复通知.数据状态保持一致.
```

2.举例类比

2.1 TCC (订婚-结婚) 二阶段2pc(二阶段提交协议) 补偿(回滚)

订婚,(第一阶段确定)---结婚领证(成功,提交事务\失败,双方回滚事务)

2.2 通过消息保证事务的最终一致性(在线进行下棋)



2.3 最大努力通知（非可靠消息、定期校对）

> 滴滴打车有未付款的订单，每隔一段时间滴滴都会通知我们付款.
> 滴滴最大努力通知用户；用户定期校验.如果未付款，则执行付款，付款后滴滴不再重复通知.数据状态保持一致.

小结:

1.base理论 cap理论的延伸

一致性\可用性\p分区容错

cp\ap

2. 柔性事务(刚性事务相对应)
    a. acid 刚性事务
    b. base/cap柔性事务
3. 柔性事务的几种解决方案
    a. tcc (订婚-领证)
    b. 消息保障型(在线下棋)
    c. 最大努力通知(迪迪通知)

提问环节:

第2章 分布式事务解决方案模型\规范\接口\方案的关系

目标:

1.DTP -----XA-----二阶段提交协议(2pc tcc)

3.二阶段2pc与TCC关系

2. DTP -----XA-----二阶段提交协议(2pc tcc)

转载自http://www.tianshouzhi.com/api/tutorials/distributed_transaction

2.1分布式事务处理模型(DTP)标准的提供者.

谁提出来的?x/open,open group.是一个独立的组织，主要负责制定各种行业技术标准

2.1.2 DTP模型与XA规范

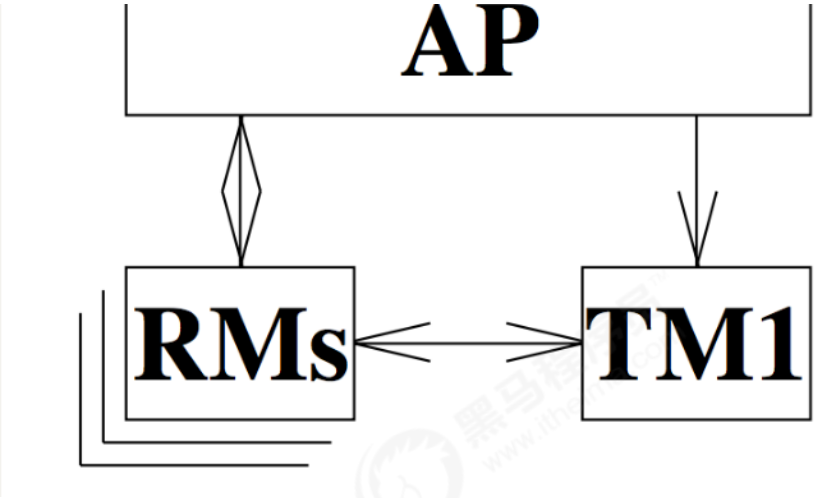
2.1.3 DTP模型

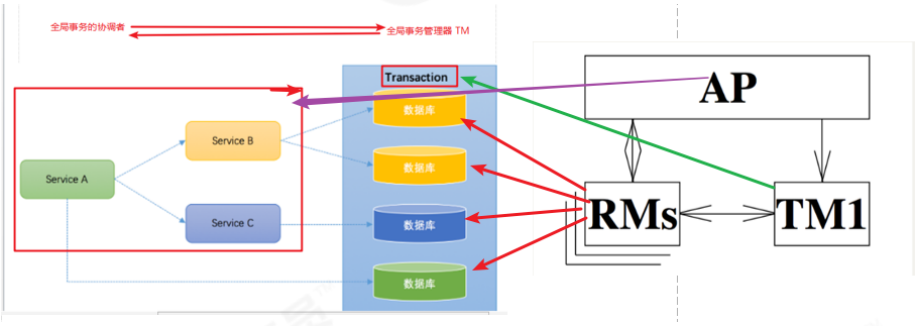
1.*模型元素*(5个)

- 应用程序(Application Program，简称AP)：
- 资源管理器(Resource Manager，简称RM)：如数据库、文件系统等，并提供访问资源的方式。
- 事务管理器(Transaction Manager，简称TM)：负责分配事务唯一标识，监控事务的执行进度，并负责事务的提交、回滚等。
- 通信资源管理器(Communication Resource Manager，简称CRM)：控制一个TM域(TM domain)内或者跨TM域的分布式应用之间的通信。
- 通信协议(Communication Protocol，简称CP)：

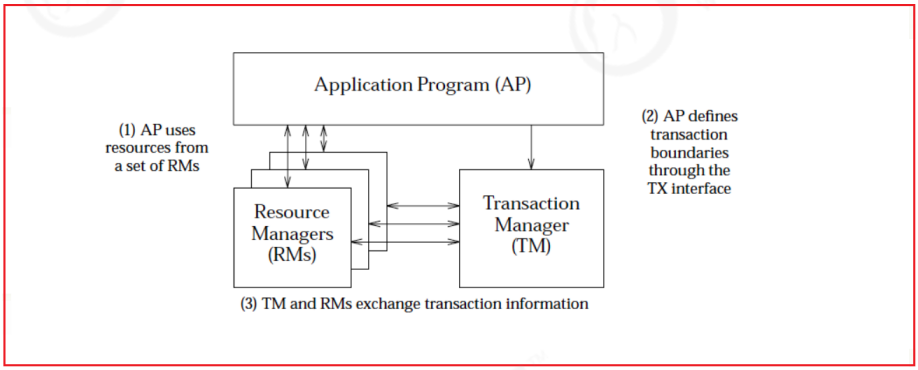
2.模型实例(Instance of the Model)

3部分：



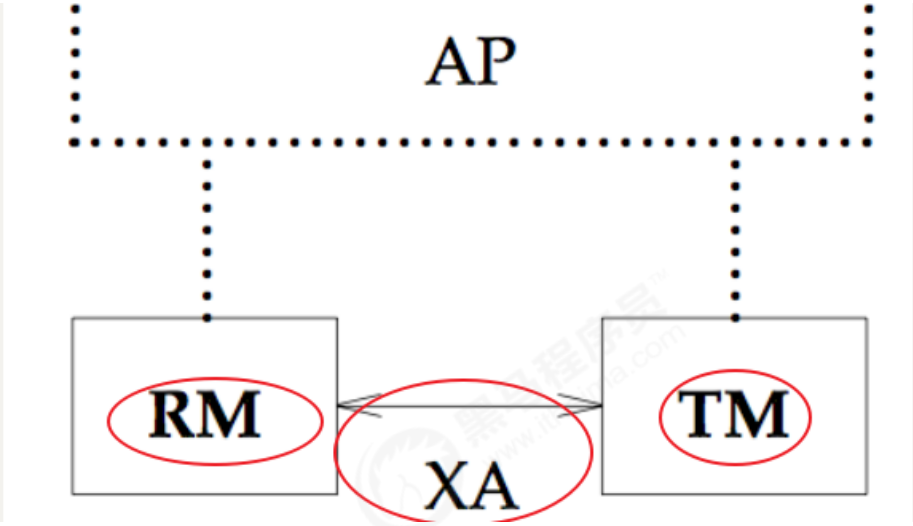


2.1.4 XA规范



定义:**XA规范描述如下:XA规范的最主要的作用是，就是定义了RM-TM的交互接口**，

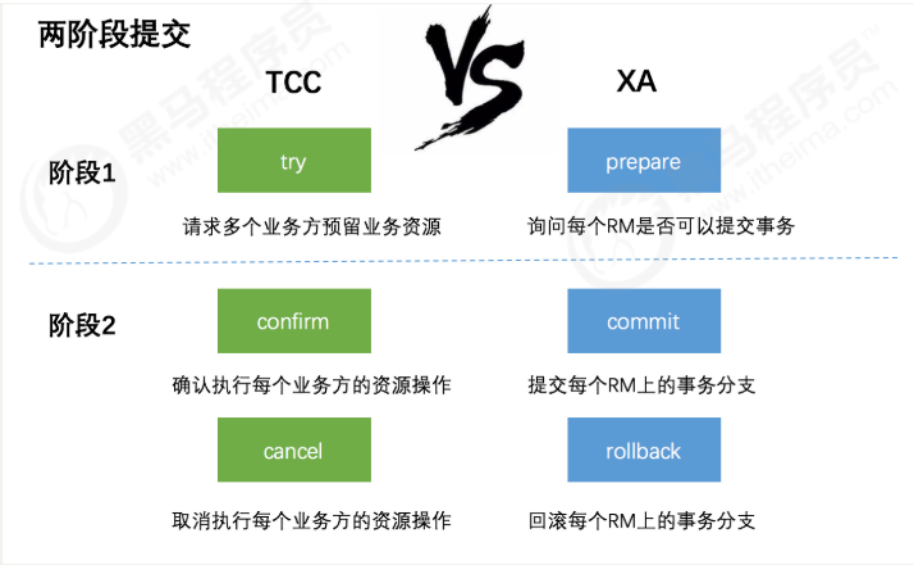
下图更加清晰了演示了XA规范在DTP模型中发挥作用的位置



2.1.5 XA规范与二阶段协议的关系:

协议与接口(XA)

二阶段提交协议并非在XA规范中提出来的。但XA规范定义了两阶段提交协议中需要使用到的接口.



总结:

DTP :分布式事务处理的模型

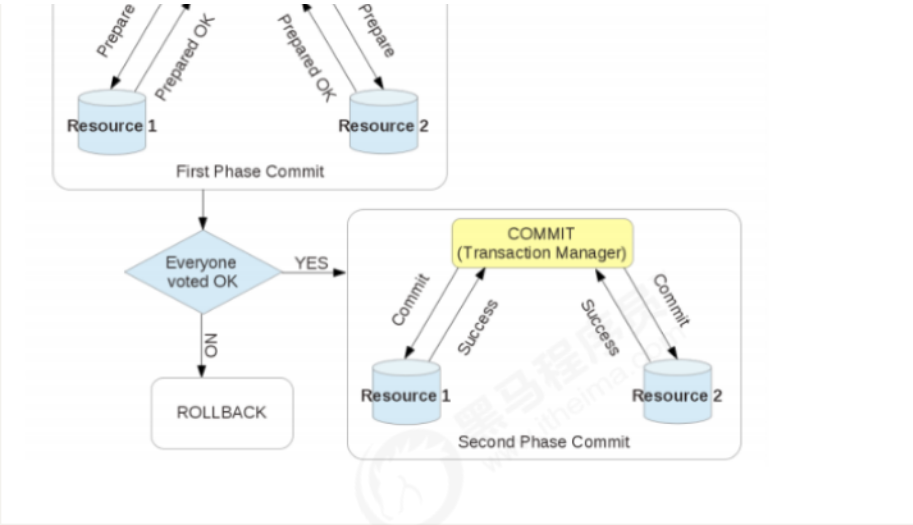
XA: 数据库和tm之间的接口

xa和2pc是相互参考的

2.1.6 二阶段协议和tcc之间的关系:

二阶段协议:

订婚和领证的关系(订了娃娃亲)



第一阶段。准备阶段

第二阶段:执行阶段 commit/rollback

优点： 尽量保证了数据的强一致，适合对数据强一致要求很高的关键领域。

缺点： 实现复杂，牺牲了可用性，对性能影响较大，不适合高并发高性能场景，如果分布式系统跨接口调用。

XA的性能问题 XA的性能很低。。 只有在这些都无法实现，且性能不是瓶颈时才应该使用XA。

总结:

1.XA :接口

二阶段:一组协议,

2.二阶段:

优点: 实现起来相对来说比较简单

缺点: 性能不高.

应用场景:对性能要求不太高,并且要求数据强一致性时,可选择二阶段协议的方案

2.1.7 JTA与xa及atomikos的关系

概念:（JTA：Java Transaction Api）xa的java实现版

某种程度上，可以认为**JTA规范是XA规范的Java版**，在JTA 中，事务管理器抽象为javax.transaction.TransactionManager接口，并通过底层事务服务（即JTS）实现。

JTA仅仅定义了接口

- 1.J2EE容器所提供的JTA实现(JBoss)
- 2.独立的JTA实现:如JOTM，Atomikos.用于Tomcat,Jetty以及普通的java应用。
- Atomikos JTA的实现,用于tomcat等容器

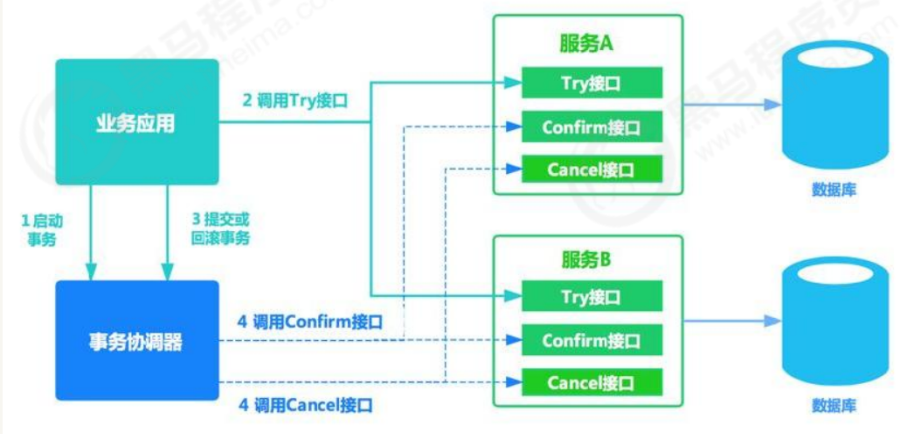
总结:JTA 是xa的java实现；Atomikos是JTA的一种实现

2.2 补偿事务（TCC）(自由恋爱.)

tcc是二阶段协议的一种,(优化:锁定了一部分资源,剩余的资源,其他的事务可以使用)

TCC 其实就是采用的补偿机制，其核心思想是：针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作。它分为三个阶段：

- Try 阶段主要是对业务系统做检测及资源预留
- Confirm 阶段主要是对业务系统做确认提交，Try阶段执行成功并开始执行 Confirm阶段时，默认 Confirm阶段是不会出错的。即：只要Try成功，Confirm一定成功。
- Cancel 阶段主要是在业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放。



例如： A要向 B 转账，思路大概是：

```
我们有一个本地方法，里面依次调用
1、首先在=Try 阶段，要先调用远程接口把 B和 A的钱给冻结起来。=
2、在 Confirm 阶段，执行远程调用的转账的操作，转账成功进行解冻。
3、如果第2步执行成功，那么转账成功，如果第二步执行失败，则调用远程冻结接口对应的解冻方法 (Cancel)。
```

优点： 跟2PC比起来，实现以及流程相对简单了一些，但数据的一致性比2PC也要差一些

缺点： 缺点还是比较明显的，在2,3步中都有可能失败。TCC属于应用层的一种补偿方式，所以需要程序员在实现的时候多写很多补偿的代码，在一些场景中，一些业务流程可能用TCC不太好定义及处理。在代码无法完成事务时,可以通过手工干预

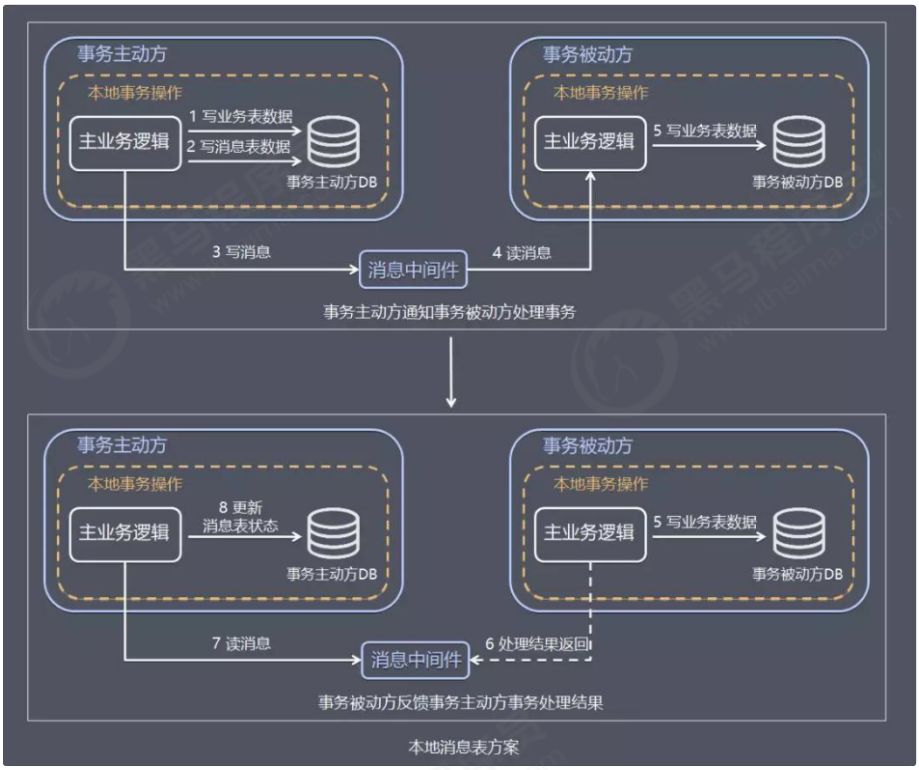
1.TCC (TRY-CONMIT 或者 TRY-CANCEL)

2. 优点:可以再try阶段,预留一部分资源,使剩余的资源得到释放(与二阶段做比较,二阶段在第一阶段锁定资源,所以二阶段的效率低)
3. 缺点:2 和3 步都可能失败,需要些更多的补偿代码

2.3 通过消息队列保证事务的一致性

本地消息表（异步确保）

本地消息表这种实现方式应该是业界使用最多的，其核心思想是将分布式事务拆分成本地事务进行处理，这种思路是来源于ebay。我们可以从下面的流程图中看出其中的一些细节：



具体保存一致性的容错处理如下：

- **当步骤 1 处理出错**，事务回滚，相当于什么都没发生。
- **当步骤 2、步骤 3 处理出错**，由于未处理的事务消息还是保存在事务发送方，事务发送方可以定时轮询为超时消息数据，再次发送到消息中间件进行处理。事务被动方消费事务消息重试处理。
- **如果是业务上的失败**，事务被动方可以发消息给事务主动方进行回滚。
- **如果多个事务被动方已经消费消息**，事务主动方需要回滚事务时需要通知事务被动方回滚。

基本思路就是：

发送到消息的消费方。如果消息发送失败，会进行重试发送。

消息消费方，需要处理这个消息，并完成自己的业务逻辑。此时如果本地事务处理成功，表明已经处理成功了，如果处理失败，那么就会重试执行。如果是业务上面的失败，可以给生产方发送一个业务补偿消息，通知生产方进行回滚等操作。

生产方和消费方定时扫描本地消息表，把还没处理完成的消息或者失败的消息再发送一遍。如果有靠谱的自动对账补账逻辑，这种方案还是非常实用的。

这种方案遵循BASE理论，采用的是最终一致性，笔者认为是这几种方案里面比较适合实际业务场景的，即不会出现像2PC那样复杂的实现(当调用链很长的时候，2PC的可用性是非常低的)，也不会像TCC那样可能出现确认或者回滚不了的情况。

优点： 一种非常经典的实现，避免了分布式事务，实现了最终一致性。

缺点： 消息表会耦合到业务系统中，如果没有封装好的解决方案，会有很多杂活需要处理。

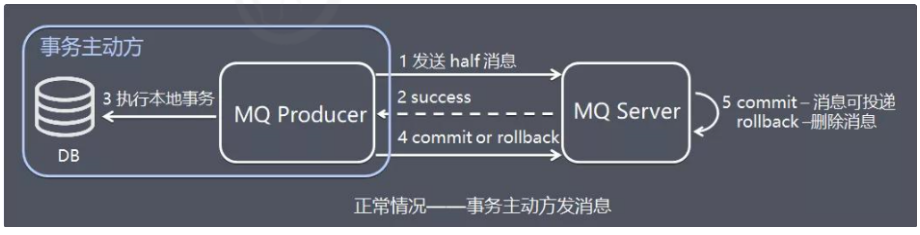
2.4 MQ 事务消息(rocketMQ)

有一些第三方的MQ是支持事务消息的，比如RocketMQ，他们支持事务消息的方式也是类似于采用的二阶段提交，但是市面上一些主流的MQ都是不支持事务消息的，比如 RabbitMQ 和 Kafka 都不支持。

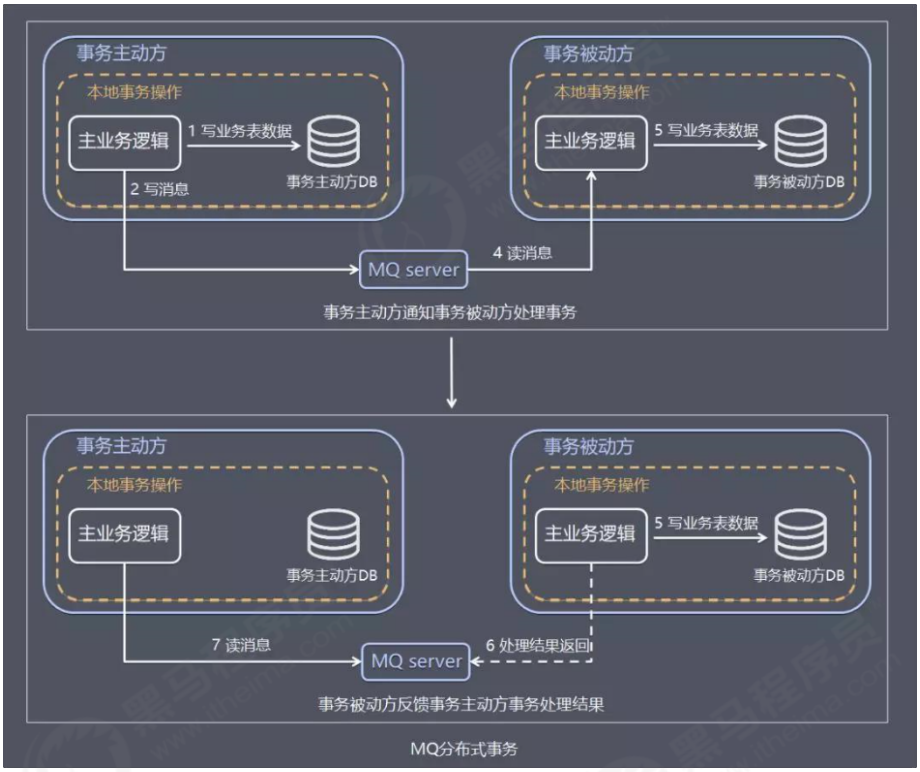
以阿里的 RocketMQ 中间件为例，其思路大致为：

第一阶段Prepared消息，会拿到消息的地址。 第二阶段执行本地事务，第三阶段通过第一阶段拿到的地址去访问消息，并修改状态。

也就是说在业务方法内要想消息队列提交两次请求，一次发送消息和一次确认消息。如果确认消息发送失败了RocketMQ会定期扫描消息集群中的事务消息，这时候发现了Prepared消息，它会向消息发送者确认，所以生产方需要实现一个check接口，RocketMQ会根据发送端设置的策略来决定是回滚还是继续发送确认消息。这样就保证了消息发送与本地事务同时成功或同时失败。



异常情况：事务主动方消息恢复



容错处理

如果事务被动方消费消息异常，需要不断重试，业务处理逻辑需要保证幂等。

如果是事务被动方业务上的处理失败，可以通过 MQ 通知事务主动方进行补偿或者事务回滚。

优点： 实现了最终一致性，不需要依赖本地数据库事务。

缺点： 目前主流MQ中只有RocketMQ支持事务消息。

共同点:

特点:都需要自己写业务补偿代码(cancle代码)

(优点)用消息队列的方式实现分布式事务,效率较高

缺点:实现难度较大,和业务耦合比较紧密

第3章 分布式事务实战--Atomikos

xa--jta---具体实现的一种Atomikos ()

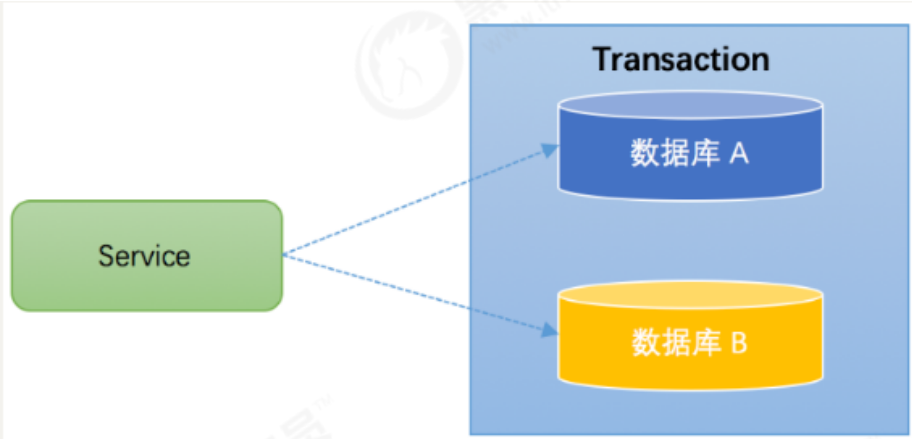
1.知道atomikos适合什么样的分布式场景

2.了解案例的业务场景和数据库表结构

3.了解测试用例

3.1 案例说明:

适合:单服务,多数据源



我们这里准备2个数据库，分别是订单数据库和日志数据库，订单数据库用于接收用户订单，日志数据库用于记录用户的订单创建操作。

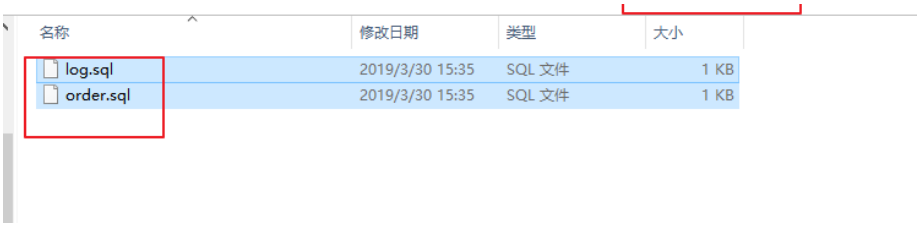
order数据库

```
CREATE TABLE `order_info` (
  `id` int(11) NOT NULL,
  `money` double NOT NULL,
  `userid` varchar(20) DEFAULT NULL,
  `address` varchar(200) DEFAULT NULL,
  `createTime` datetime DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

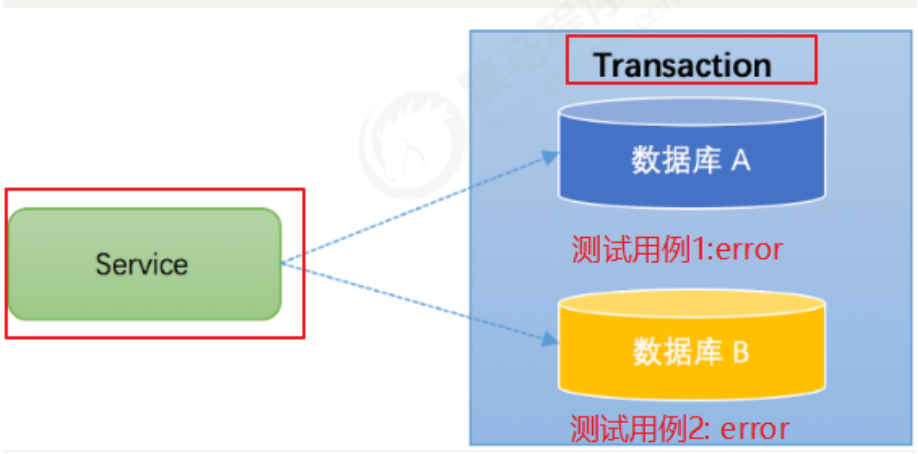
log数据库

```
CREATE TABLE `log_info` (
  `id` int(11) NOT NULL,
  `createTime` datetime DEFAULT NULL,
  `content` longtext,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

数据库脚本文件位置:



测试用例说明:



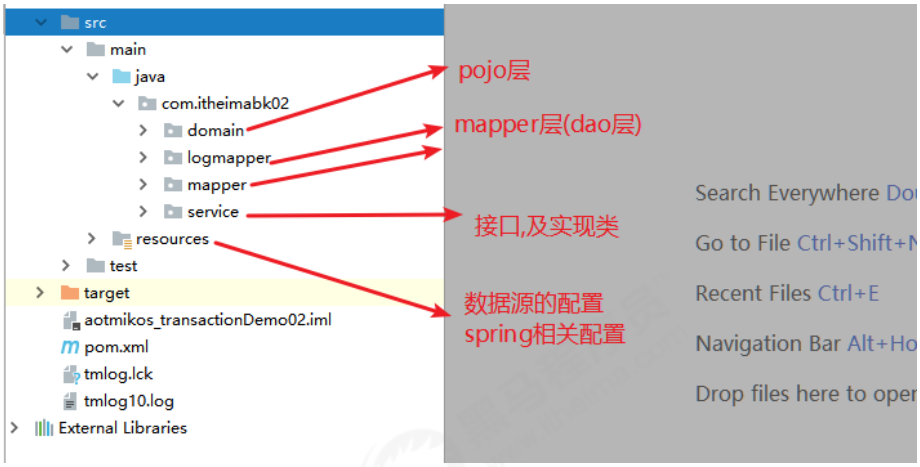
3.2 核心步骤:

```
Atomikos步骤分析:
1.公共数据源代理的配置
    1)配置两个数据源
    2)xa数据源统一管理
    3)分别配置两个sessionFactoryBean
    4)管理各自的mapper
2.配置全局事务管理器
3.将全局事务管理器集成到spring之中
4.切面管理配置和事务通知管理
5.将切入点和事务通知关联起来
```

3.3 代码实现:

pojo--->dao层--->interface接口层--->实现类层---->controller层

1.导入半成品项目与说明



2.业务实现:

- 业务分析与测试用例:
- 业务代码:

下订单,在另一个数据库中写日志.

```
package com.itheimabk02.service.impl;
import com.itheimabk02.domain.LogInfo;
import com.itheimabk02.domain.OrderInfo;
import com.itheimabk02.logmapper.LogInfoMapper;
import com.itheimabk02.mapper.OrderInfoMapper;
import com.itheimabk02.service.OrderInfoService;
import
org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.Date;

/**
 * @version 1.0
 * @Author 周洪丙
 **/
@Service
public class OrderInfoServiceImpl implements
OrderInfoService {
    @Autowired
    private OrderInfoMapper orderInfoMapper;

    @Autowired
    private  LogInfoMapper logInfoMapper;

    public int add(OrderInfo orderInfo) {


        //测试用例1:此处发生异常(不用测)
```

```
        System.out.println("orderInfo表受影响行数"+i1);

        //测试用例2:此处发生异常,orderInfo已经保存到 order库中
order_info表中,但是前面的保存要做要做回滚
//        System.out.println("发生异常:测试用例
1.............................");
//        int i=10/0;
        //保存logInfo到log库中log_info表
        LogInfo logInfo = new LogInfo();
        logInfo.setId((int) (Math.random()*1000));
        logInfo.setCreateTime(new Date());
        logInfo.setContent(orderInfo.toString());
        int i2 = logInfoMapper.add(logInfo);

        System.out.println("loginfo表受影响行数"+i2);
        //测试用例3:此处发生异常,orderInfo已经保存到 order库中
order_info表中,保存logInfo到log库中log_info表,但是前面的保存的
orderinfo和loginfo要做要做回滚
        System.out.println("发生异常:测试用例
2.............................");
        int i=10/0;
        return 0;
    }
}
```

3.测试代码:

```
package com.itcastbk02.test;

import com.itheimabk02.domain.OrderInfo;
import com.itheimabk02.service.OrderInfoService;
import
org.springframework.context.support.ClassPathXmlApplicatio
nContext;

import java.util.Date;

/**
 * @version 1.0
 * @Author 周洪丙
 **/
public class AtomikosTest02 {

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

```
        OrderInfoService orderInfoService =
cat.getBean(OrderInfoService.class);

        OrderInfo orderInfo = new OrderInfo();
        orderInfo.setId((int) (Math.random()*1000));
        orderInfo.setAddress("顺义黑马训练营");
        orderInfo.setCreateTime(new Date());
        orderInfo.setUserid("zhangxiaoliu");
        orderInfo.setMoney(666d);
        orderInfoService.add(orderInfo);
    }
}
```

3.4 atomikos的核心配置步骤实现(核心重点内容)

```
Atomikos步骤分析:
1.公共数据源代理的配置
   1)配置两个数据源
   2)xa数据源统一管理
   3)分别配置两个sessionFactoryBean
   4)管理各自的mapper
2.配置全局事务管理器(atomikos提供)
3.将全局事务管理器集成到spring之中
4.切面管理配置和事务通知管理
5.将切入点和事务通知关联起来
```

引入相关jar包,maven(pom配置jar包依赖)

改造一些配置文件

1.导入atomikos的jar(pom)

```
<!--JTA atomikos-->
        <dependency>
            <groupId>javax.transaction</groupId>
            <artifactId>jta</artifactId>
            <version>${jta.version}</version>
        </dependency>
        <dependency>
            <groupId>com.atomikos</groupId>
            <artifactId>atomikos-util</artifactId>
            <version>${atomikos.version}</version>
        </dependency>
        <dependency>
            <groupId>com.atomikos</groupId>
            <artifactId>transactions</artifactId>
            <version>${atomikos.version}</version>
        </dependency>
```

```
            <artifactId>transactions-jta</artifactId>
            <version>${atomikos.version}</version>
        </dependency>
        <dependency>
            <groupId>com.atomikos</groupId>
            <artifactId>transactions-jdbc</artifactId>
            <version>${atomikos.version}</version>
        </dependency>
        <dependency>
            <groupId>com.atomikos</groupId>
            <artifactId>transactions-api</artifactId>
            <version>${atomikos.version}</version>
        </dependency>
        <dependency>
            <groupId>cglib</groupId>
            <artifactId>cglib-nodep</artifactId>
            <version>${cglib.nodep.version}</version>
        </dependency>
```

2 配置多个数据源:(有区别)

1)需要配置两个数据源 jdbc.properties

```
#\u8BA2\u5355\u6570\u636E\u5E93
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://127.0.0.1:3306/order?
useUnicode=true&characterEncoding=utf8&autoReconnect=true
jdbc.username=root
#jdbc.pwd=123456
jdbc.pwd=


#\u65E5\u5FD7\u6570\u636E\u5E93
jdbc.log.driver=com.mysql.jdbc.Driver
jdbc.log.url=jdbc:mysql://127.0.0.1:3306/log?
useUnicode=true&characterEncoding=utf8&autoReconnect=true
jdbc.log.username=root
#jdbc.log.pwd=123456
jdbc.log.pwd=
```

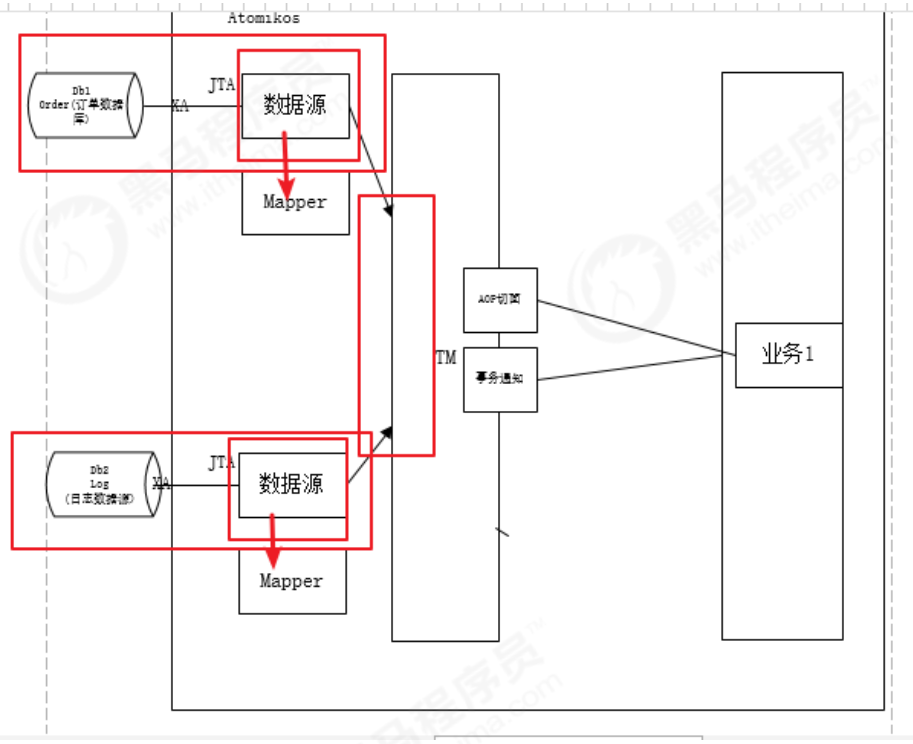
2) spring.xml中配置两个数据源

```
<!--数据源基础配置-->
```

```
method="init" destroy-method="close" abstract="true">
        <property name="xaDataSourceClassName"
value="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource"/>
        <property name="poolSize" value="10"/>
        <property name="minPoolSize" value="10"/>
        <property name="maxPoolSize" value="30"/>
        <property name="borrowConnectionTimeout"
value="60"/>
        <property name="reapTimeout" value="20"/>
        <property name="maxIdleTime" value="60"/>
        <property name="maintenanceInterval" value="60"/>
        <property name="testQuery">
            <value>SELECT 1</value>
        </property>
    </bean>

 <!-- 数据库基本信息配置 -->
<bean id="dataSourceOne" parent="abstractXADataSource">
        <property name="uniqueResourceName">
            <value>dataSourceOne</value>
        </property>
        <!--数据库驱动-->
        <property name="xaDataSourceClassName"
value="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource"/>
        <property name="xaProperties">
            <props>
                <prop key="URL">${jdbc.url}</prop>
                <prop key="user">${jdbc.username}</prop>
                <prop key="password">${jdbc.pwd}</prop>
            </props>
        </property>
    </bean>

    <!--日志数据源-->
    <bean id="dataSourceLog"
parent="abstractXADataSource">
        <property name="uniqueResourceName">
            <value>dataSourceLog</value>
        </property>
        <property name="xaDataSourceClassName"
value="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource"/>
        <property name="xaProperties">
            <props>
                <prop key="URL">${jdbc.log.url}</prop>
                <prop key="user">${jdbc.log.username}
</prop>
                <prop key="password">${jdbc.log.pwd}
</prop>
            </props>
        </property>
```

```
    <!--SqlSessionFactoryBean的配置-->
    <bean id="sqlSessionFactoryBeanOne"
class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="typeAliasesPackage"
value="com.itheimabk01.domain" />
        <property name="mapperLocations">
            <array>

 <value>classpath:com/itheimabk01/mapper/*Mapper.xml</valu
e>
            </array>
        </property>
        <property name="dataSource" ref="dataSourceOne"/>
    </bean>

    <bean id="sqlSessionFactoryBeanLog"
class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="typeAliasesPackage"
value="com.itheimabk01.domain" />
        <property name="mapperLocations">
            <array>

 <value>classpath:com/itheimabk01/logmapper/*Mapper.xml</v
alue>
            </array>
        </property>
        <property name="dataSource" ref="dataSourceLog"/>
    </bean>


    <!--包扫描-->
    <bean id="mapperScannerConfigurerOne"
class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <property name="basePackage"
value="com.itheimabk01.mapper" />
        <property name="sqlSessionFactoryBeanName"
value="sqlSessionFactoryBeanOne" />
    </bean>
    <bean id="mapperScannerConfigurerLog"
class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <property name="basePackage"
value="com.itheimabk01.logmapper" />
        <property name="sqlSessionFactoryBeanName"
value="sqlSessionFactoryBeanLog" />
    </bean>
```

```
    <bean id="sqlSessionFactoryBeanOne"
class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="typeAliasesPackage"
value="com.itheimabk01.domain" />
        <property name="mapperLocations">
            <array>

 <value>classpath:com/itheimabk01/mapper/*Mapper.xml</value>
            </array>
        </property>
        <property name="dataSource" ref="dataSourceOne"/>
    </bean>

    <bean id="sqlSessionFactoryBeanLog"
class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="typeAliasesPackage"
value="com.itheimabk01.domain" />
        <property name="mapperLocations">
            <array>

 <value>classpath:com/itheimabk01/logmapper/*Mapper.xml</val
ue>
            </array>
        </property>
        <property name="dataSource" ref="dataSourceLog"/>
    </bean>
```



3.配置公共事务管理器

```
</bean>
<!--配置本地事务管理器-->
<bean id="atomikosUserTransaction" class="com.atomikos.icatch.jta.UserTransactionImp">
    <property name="transactionTimeout" value="300000"/>
</bean>

<!--JTA事务管理器-->
<bean id="springTransactionManager" class="org.springframework.transaction.jta.JtaTransactionManager">
    <property name="transactionManager">
        <ref bean="atomikosTransactionManager"/>
    </property>
    <property name="userTransaction">
        <ref bean="atomikosUserTransaction"/>
    </property>
    <property name="allowCustomIsolationLevels" value="true"/>
</bean>
```

```
<!-- 配置atomikos事务管理器 -->
    <bean id="atomikosTransactionManager"
class="com.atomikos.icatch.jta.UserTransactionManager"
init-method="init" destroy-method="close">
        <property name="forceShutdown" value="false"/>
    </bean>
    <!--配置本地事务管理器-->
    <bean id="atomikosUserTransaction"
class="com.atomikos.icatch.jta.UserTransactionImp">
        <property name="transactionTimeout"
value="300000"/>
    </bean>

    <!--JTA事务管理器-->
    <bean id="springTransactionManager"
class="org.springframework.transaction.jta.JtaTransactionM
anager">
        <property name="transactionManager">
            <ref bean="atomikosTransactionManager"/>
        </property>
        <property name="userTransaction">
            <ref bean="atomikosUserTransaction"/>
        </property>
        <property name="allowCustomIsolationLevels"
value="true"/>
    </bean>
```

```
class="com.atomikos.icatch.jta.UserTransactionManager" init-
method="init" destroy-method="close">
        <property name="forceShutdown" value="false"/>
    </bean>
    <!--配置本地事务管理器-->
    <bean id="atomikosUserTransaction"
class="com.atomikos.icatch.jta.UserTransactionImp">
        <property name="transactionTimeout" value="300000"/>
    </bean>

    <!--JTA事务管理器-->
    <bean id="springTransactionManager"
class="org.springframework.transaction.jta.JtaTransactionMan
ager">
        <property name="transactionManager">
            <ref bean="atomikosTransactionManager"/>
        </property>
        <property name="userTransaction">
            <ref bean="atomikosUserTransaction"/>
        </property>
        <property name="allowCustomIsolationLevels"
value="true"/>
    </bean>
```

4.配置事务通知

```
    <!--@Aspect-->
    <aop:aspectj-autoproxy/>

    <!--使用CGLIB动态代理-->
    <tx:annotation-driven transaction-
manager="springTransactionManager" proxy-target-
class="true" />

    <!--配置事务的通知-->
    <!-- the transactional advice (what 'happens'; see the
<aop:advisor/> bean
        below) 事务传播特性配置 -->
    <tx:advice id="txAdvice" transaction-
manager="springTransactionManager">
        <!-- the transactional semantics... -->
        <tx:attributes>
            <tx:method name="add*" propagation="REQUIRED"
isolation="DEFAULT"
                       rollback-for="java.lang.Exception"
/>
            <tx:method name="save*" propagation="REQUIRED"
isolation="DEFAULT"
```

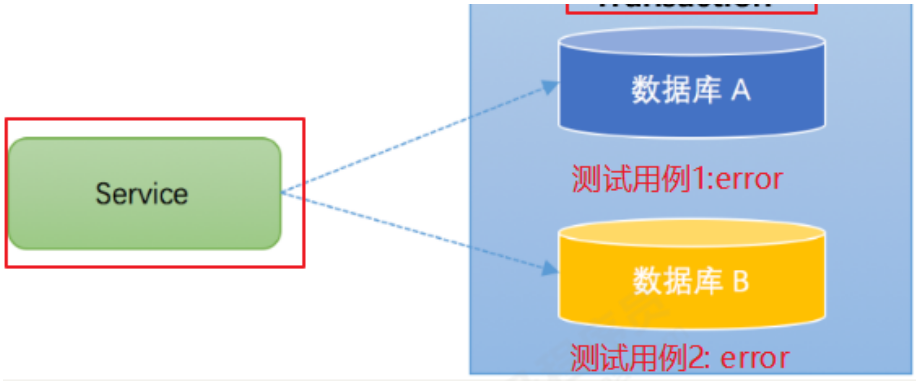
```
            <tx:method name="insert*"
propagation="REQUIRED" isolation="DEFAULT"
                       rollback-for="java.lang.Exception"
/>
            <tx:method name="update*"
propagation="REQUIRED" isolation="DEFAULT"
                       rollback-for="java.lang.Exception"
/>
            <tx:method name="modify*"
propagation="REQUIRED" isolation="DEFAULT"
                       rollback-for="java.lang.Exception"
/>
            <tx:method name="delete*"
propagation="REQUIRED" isolation="DEFAULT"
                       rollback-for="java.lang.Exception"
/>
            <!-- 查询方法 -->
            <tx:method name="query*" read-only="true" />
            <tx:method name="select*" read-only="true" />
            <tx:method name="find*" read-only="true" />
        </tx:attributes>
    </tx:advice>
```

5.配置切面和切入点

6.织入业务(关联切入点和事务通知给服务service的impl层)

```
<!-- 声明式事务AOP配置 -->
    <aop:config>
        <!--配合aop的切入点-->
        <aop:pointcut expression="execution(*
com.itheimabk01.service.impl.*.*(..))" id="tranpointcut"
/>

        <!--生命式事务通知,配置事务通知和切入点关系-->
        <aop:advisor advice-ref="txAdvice" pointcut-
ref="tranpointcut" />
    </aop:config>
```

7.测试结果




```
信息: Using JTA UserTransaction: com.atomikos.icatch.jta.UserTransactionImp@7d898981
八月 31, 2019 9:51:17 下午 org.springframework.transaction.jta.JtaTransactionManager checkUse
信息: Using JTA TransactionManager: com.atomikos.icatch.jta.UserTransactionManager@48d61b48
八月 31, 2019 9:51:17 下午 com.atomikos.logging.JULLogger logWarning
警告: Attempt to create a transaction with a timeout that exceeds maximum - truncating to: 3
插入orderinfo受影响的数据为1条数据
插入loginfo受影响的数据为1条数据

Process finished with exit code 0
```

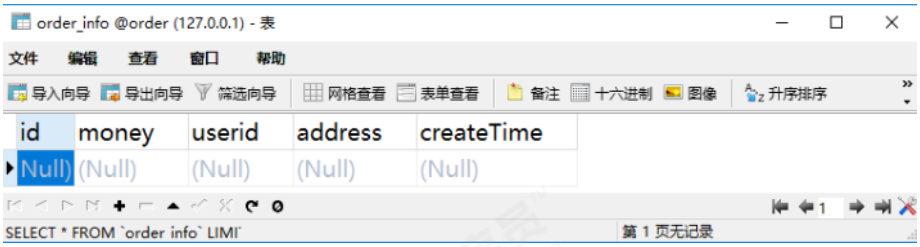
```
警告: Attempt to create a transaction with a timeout that exceeds maximum - truncating to: 300000
插入orderinfo受影响的数据为1条数据
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at com.itheimabk02.service.impl.OrderInfoServiceImpl.add(OrderInfoServiceImpl.java:43)
    at com.itheimabk02.service.impl.OrderInfoServiceImpl$$FastClassBySpringCGLIB$$4bd955b3.invoke(<generated>)
    at org.springframework.cglib.proxy.MethodProxy.invoke(MethodProxy.java:204)
    at org.springframework.aop.framework.CglibAopProxy$CglibMethodInvocation.invokeJoinpoint(CglibAopProxy.java:747)
    at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:163)
```

插入成功后抛异常,插入orderinfo做回滚

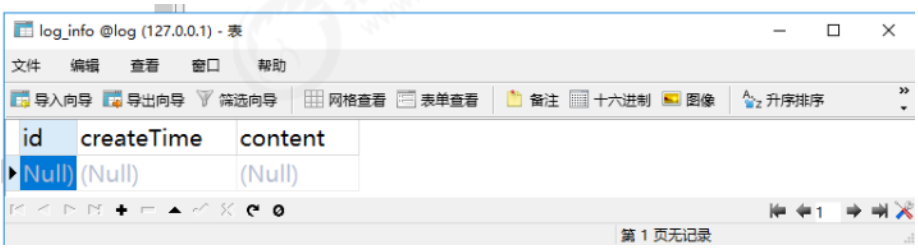
```
信息: Using JTA TransactionManager: com.atomikos.icatch.jta.UserTransactionManager@48d61b48
八月 31, 2019 9:56:53 下午 com.atomikos.logging.JULLogger logWarning
警告: Attempt to create a transaction with a timeout that exceeds maximum - truncating to: 300000
插入orderinfo受影响的数据为1条数据
插入loginfo受影响的数据为1条数据
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at com.itheimabk02.service.impl.OrderInfoServiceImpl.add(OrderInfoServiceImpl.java:53)
    at com.itheimabk02.service.impl.OrderInfoServiceImpl$$FastClassBySpringCGLIB$$4bd955b3.invoke(<gener
    at org.springframework.cglib.proxy.MethodProxy.invoke(MethodProxy.java:204)
```

数据库数据

order_info表的数据为空



log_info表的数据为空



从上面结果我们可以看得到，分布式事务成功了！