**分布式微服务--最终一致性事务**

分布式事务一致是一个难点，但也是一个重点,各大数据厂商也都提供了分布式事务解决方案，但是基于XA的二阶段提交方式效率太低，而且对两个库的操作都要以SQL的形式出现在代码中，假如这时候有A数据库和B数据库。A库中有一个帐户a ,B库中有一个账户b，现在要将a的钱转给b,这种情况如果保证事务的一致性呢？当然XA二阶段提交是一种方法，即在代码中获得两个库的连接，然后写两条SQL语句:

update A.account set amount=amount-100 where id=’a’;

update B.account set amount=amount+100 where id=’b’;

通过将两条SQL语句放到JTA的XA事务中去处理，就可以达到事务一致性，如果某一条失败直接回滚,但是这有以下两个缺点:

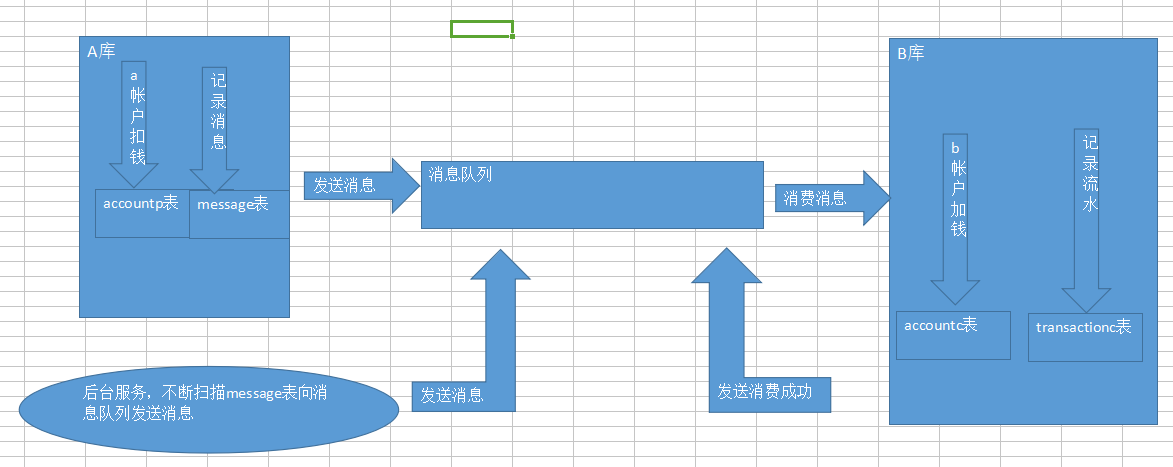
1.效率低，如果大并发量的情况下会导致性能低,因为在JTA事务没有提交前，锁会一直保持，锁持有时间过长就会并发效率低的问题

2.必须直接操作数据库，由于需要使用XA协议，那么对两个库的操作都必须直接作用于数据库，很显然在分布式微服务的架构中这是不理想的，因为一个良好的微服务只对外提供接口，而不希望其它调用者直接操作它的数据库

为了解决上面的问题，我们需要一种可效率更高的可替代方案，那就是分布式事务的最终一致性。

下面我们来演示一下如何在分布式微服务中实现分布式事务的最终一致性。什么叫最终一致性，就是说在用户容忍的时间范围内达到数据的一致性就行了，而不期望用户提交业务后马上看到效果.

我们做以下设计,如图



首先A库中有一个帐户表 accountp,一个事件消息表message，我们从a的帐户扣钱的时候，同时向其本地的数据库message表中添加一个记录，用来存储转账人，被转账人，转账金额，事务Id等信息

B库中有一个帐户表accountc,一个事务流水表transactionc，我们向b帐户加钱的时候，同时向其本地的transactionc表中添加一条记录。

下面是这几张表的表结构：

A 库帐户表

CREATE TABLE `accountp` (

`account\_id` varchar(11) NOT NULL COMMENT '用户ID',

`name` varchar(100) DEFAULT NULL COMMENT '姓名',

`amount` decimal(10,2) DEFAULT NULL COMMENT '帐户余额',

PRIMARY KEY (`account\_id`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

A 库消息表

CREATE TABLE `message` (

`xid` varchar(40) NOT NULL COMMENT '事务ID',

`amount` decimal(10,2) DEFAULT NULL COMMENT '金额',

`\_from` varchar(100) DEFAULT NULL COMMENT '从\_from处转账',

`\_to` varchar(100) DEFAULT NULL COMMENT '转到\_to',

PRIMARY KEY (`xid`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

B库帐户表

CREATE TABLE `accountc` (

`account\_id` varchar(11) NOT NULL COMMENT '帐户ID',

`name` varchar(100) DEFAULT NULL COMMENT '姓名',

`amount` decimal(10,2) DEFAULT NULL COMMENT '帐户余额',

PRIMARY KEY (`account\_id`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

B库流水表

CREATE TABLE `transactionc` (

`xid` varchar(40) NOT NULL COMMENT ' 事务Id',

`amount` decimal(10,2) DEFAULT NULL COMMENT '金额',

`\_from` varchar(100) DEFAULT NULL COMMENT '从\_from转账',

`\_to` varchar(100) DEFAULT NULL COMMENT '转到\_to',

PRIMARY KEY (`xid`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

除了上面两张表我们还需要一个消息队列activemq。

下面我们描述一下转账的具体过程：

（1）我们首先在A库本地执行两条SQL语句分别用来扣除a帐的100元并将事件记录下来插入一个本地消息表中:

update accountp a set a.amount=a.amount-100 where a.account\_id=’a’ ;

insert into message (xid ,amount,\_from,\_to) values('06754e86-5c7d-4071-977a-7da89527f31f',100,'a','b');

上面的操作是一个本地事务，所以是原子性的

1. 向消息队列activemq中的account队列发送一条消息
2. 消息发送成功后删除message中的记录,消息发送失败则不删除message中的记录
3. 如果消息发送失败了，那么我们需要启动一个定时器，定时的从message表中读取没有发送成功的记录，再次发送到消息队列，发送成功后删除，这一步操作相当于一个定时补偿机制

（5）消费者去队列消费，从中取出消息然后根据消息中的值对B库中accountc 中的b用户进行加钱，同时将该操作记录到transactionc表中，这个过程是也一个本地事务:

update accountc c set c.amount=c.amount+100 where c.account\_id='b';

insert into transactionc (xid,amount,\_from,\_to) values('06754e86-5c7d-4071-977a-7da89527f31f',100,'a','b');

(6)消费者向消息队列发送确认收到消息命令，这时消息队列将该条消息从队列中移除。则整个转帐结束。

下面我们进一步分析一下这几个步骤

首先，上面（1）步骤是一个本地事务，也就是说只要将a 的钱扣了，那么肯定会插入一条message信息记录。（2）步骤向消息队列发送消息,如果成功，那么message中的记录就会被删除，如果不成功，那message中的记录就不会被删除,有一种情况是消息发送成功了但是突然断电了，导致message表中的记录没有来得及删除。这种情况我们需要启动一个后台服务定时去扫描message表，将message表中的记录重新发送到消息队列中去,发送成功后删除掉。这样我们就可以保证a 的钱扣了之后一定会发送消息到消息队列中,注意这种设计会导致消息重复发送。所以我们消费者需要具有等幂性，即一次消费和多次消费所产生的结果是一样的。步骤（5）中消费者消费消息并对b进行加钱，同时插入事务流水,这两条SQL语句是也本地事务，如果b帐户加钱了，事务流水一定会被插入成功。如果（5）执行成功，执行步骤（6）的过程中突然断电了，还没有来得及向消息队列发送确认命令，这样b 帐户的钱虽然已经加了，但是消息就不会从队列中移除,等消息队列恢复正常以后，该条消息会重复发送，那么b的钱会被再加一次。这显然是不符合常理的，所以我们的消费逻辑必须设计成等幂性的。我们可以在消费者执行两条SQL语句之前先去库中查询一次去验证该条消费是否已经被消费过，执行SQL

select count(1) as count from transactionc t where t.xid='06754e86-5c7d-4071-977a-7da89527f31f' 来获得结果，如果执行结果大于0说明之前消费过该条消息，那么什么也不做，如果执行结果等于0说明之前没有消费过，那就执行加钱和插入记录操作。

经过上面的设计我们就完美的解决了分布式微服务的事务最终一致性问题

下面我们使用Ice RPC框架来演示上面的案例

slice语言

module com{

module projectice{

module generated{

module test{

module transactionTest{

interface account{

string transfer(string from ,string to ,double amount);

};

interface messageEvent{

string sendMessage(string xid ,string from,string to,double amount);

};

interface receiverAccount{

string receive(string xid,string from,string to,double amount);

};

} ;

};

};

};

};

//消息队列服务类

**public** **class** MessageEventServer **extends** Application {

@Override

**public** **int** run(String[] args) {

ObjectAdapter adapter = *communicator*().createObjectAdapter("messageEventServerAdapter");

adapter.add(**new** MessageEventI(), Ice.Util.*stringToIdentity*("messageEventServer"));

adapter.activate();

*communicator*().waitForShutdown();

**return** 0;

}

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

InitializationData data = **new** InitializationData();

Properties properties = Ice.Util.*createProperties*();

properties.setProperty("messageEventServerAdapter.Endpoints", "tcp -p 9876");

data.properties = properties;

MessageEventServer server = **new** MessageEventServer();

server.main("server", args, data);

}

}

//消息服务骨架类

**public** **class** MessageEventI **extends** \_messageEventDisp {

**public** String sendMessage(String xid, String from, String to, **double** amount, Current \_\_current) {

Connection con = ConnnectionPool.*getConnection*();

String res = "0";

Statement sta = **null**;

**try** {

con.setAutoCommit(**false**);

Map<String, String> map = **new** HashMap<String, String>();

map.put("xid", xid);

map.put("amount", amount + "");

map.put("from", from);

map.put("to", to);

Gson gson = **new** Gson();

Producer p = **new** Producer();

p.init();

p.sendMessage("account", gson.toJson(map));

sta = con.createStatement();

con.setAutoCommit(**false**);

sta.execute("delete from message where xid='" + xid + "'");

con.commit();

} **catch** (Exception e) {

res = "1";

**try** {

con.rollback();

} **catch** (SQLException e1) {

e1.printStackTrace();

}

e.printStackTrace();

} **finally** {

**if** (sta != **null**) {

**try** {

sta.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

**if** (con != **null**) {

**try** {

con.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

**return** res;

}

}

//A库扣钱服务类

**public** **class** AccountServer **extends** Application {

@Override

**public** **int** run(String[] args) {

ObjectAdapter adapter = *communicator*().createObjectAdapter("accountServerAdapter");

adapter.add(**new** AccountI(), Ice.Util.*stringToIdentity*("accountServer"));

adapter.activate();

*communicator*().waitForShutdown();

**return** 0;

}

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

InitializationData data = **new** InitializationData();

Properties properties = Ice.Util.*createProperties*();

properties.setProperty("accountServerAdapter.Endpoints", "tcp -p 9875");

data.properties = properties;

AccountServer server = **new** AccountServer();

server.main("server", args, data);

}

}

//A库扣钱骨架类

**public** **class** AccountI **extends** \_accountDisp {

**public** String transfer(String from, String to, **double** amount, Current \_\_current) {

Connection con = ConnnectionPool.*getConnection*();

PreparedStatement st = **null**;

PreparedStatement st2 = **null**;

String xid = UUID.*randomUUID*().toString();

String result="0";

**try** {

con.setAutoCommit(**false**);

st = con.prepareStatement(" update accountp a set a.amount=a.amount-? where a.account\_id=? ");

st.setBigDecimal(1, **new** BigDecimal(amount));

st.setString(2, from);

st.execute();

st2 = con.prepareStatement("insert into message (xid ,amount,\_from,\_to) values(?,?,?,?)");

st2.setString(1, xid);

st2.setBigDecimal(2, **new** BigDecimal(amount));

st2.setString(3, from);

st2.setString(4, to);

st2.execute();

con.commit();

Communicator ic = CommunicatorUtil.*getCommunicator*();

ObjectPrx base = ic.stringToProxy("messageEventServer:tcp -h localhost -p 9876");

messageEventPrx proxy = messageEventPrxHelper.*checkedCast*(base);

proxy.sendMessage(xid, from, to, amount);

} **catch** (SQLException e) {

result="1";

**try** {

con.rollback();

} **catch** (SQLException e1) {

e1.printStackTrace();

}

e.printStackTrace();

} **finally** {

**if** (st != **null**) {

**try** {

st.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

**if** (st2 != **null**) {

**try** {

st2.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

**if** (con != **null**) {

**try** {

con.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

**return** result;

}

}

//B库帐户加钱服务类

**public** **class** ReceiverAccountServer **extends** Application {

@Override

**public** **int** run(String[] args) {

ObjectAdapter adapter= *communicator*().createObjectAdapter("receiverAccountServerAdapter");

adapter.add(**new** ReceiverAccountI(), Ice.Util.*stringToIdentity*("receiverAccountServer"));

adapter.activate();

*communicator*().waitForShutdown();

**return** 0;

}

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

InitializationData data=**new** InitializationData();

Properties properties=Ice.Util.*createProperties*();

properties.setProperty("receiverAccountServerAdapter.Endpoints", "tcp -p 9874");

data.properties =properties;

ReceiverAccountServer server=**new** ReceiverAccountServer();

server.main("server", args,data);

}

}

//B库帐户加钱骨架类

**public** **class** ReceiverAccountI **extends** \_receiverAccountDisp {

**public** String receive(String xid, String from, String to, **double** amount, Current \_\_current) {

Connection con = ConnnectionPool.*getConnection*();

String result = "0";

Statement sta = **null**;

**try** {

sta = con.createStatement();

ResultSet rs = sta.executeQuery("select count(1) as count from transactionc t where t.xid='" + xid + "'");

rs.next();

Integer count = rs.getInt("count");

**if** (count == 0) {

con.setAutoCommit(**false**);

sta.execute("update accountc c set c.amount=c.amount+" + amount + " where c.account\_id='" + to + "'");

sta.execute("insert into transactionc (xid,amount,\_from,\_to) values('" + xid + "'," + amount + ",'"

+ from + "','" + to + "')");

con.commit();

}

} **catch** (Exception e) {

result = "1";

**try** {

con.rollback();

} **catch** (SQLException e1) {

e1.printStackTrace();

}

e.printStackTrace();

} **finally** {

**if** (sta != **null**) {

**try** {

sta.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

**if** (con != **null**) {

**try** {

con.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

**return** result;

}

}

//生产者

**public** **class** Producer {

//ActiveMq 的默认用户名

**private** **static** **final** String ***USERNAME*** = ActiveMQConnection.***DEFAULT\_USER***;

//ActiveMq 的默认登录密码

**private** **static** **final** String ***PASSWORD*** = ActiveMQConnection.***DEFAULT\_PASSWORD***;

//ActiveMQ 的链接地址

**private** **static** **final** String ***BROKEN\_URL*** ="failover://"+"tcp://" + "192.168.37.135" + ":" + 61616+"";

AtomicInteger count = **new** AtomicInteger(0);

//链接工厂

ConnectionFactory connectionFactory;

//链接对象

Connection connection;

//事务管理

Session session;

ThreadLocal<MessageProducer> threadLocal = **new** ThreadLocal();

**public** **void** init(){

**try** {

//创建一个链接工厂

connectionFactory = **new** ActiveMQConnectionFactory(***USERNAME***,***PASSWORD***,***BROKEN\_URL***);

//从工厂中创建一个链接

connection = connectionFactory.createConnection();

//开启链接

connection.start();

//创建一个事务（这里通过参数可以设置事务的级别）

session = connection.createSession(**true**,Session.***SESSION\_TRANSACTED***);

} **catch** (JMSException e) {

e.printStackTrace();

}

}

**public** **void** sendMessage(String disname,String message) **throws** JMSException{

**try** {

//创建一个消息队列

Queue queue = session.createQueue(disname);

//消息生产者

MessageProducer messageProducer = **null**;

**if**(threadLocal.get()!=**null**){

messageProducer = threadLocal.get();

}**else**{

messageProducer = session.createProducer(queue);

threadLocal.set(messageProducer);

}

//创建一条消息

TextMessage msg = session.createTextMessage(message);

//发送消息

messageProducer.send(msg);

//提交事务

session.commit();

} **catch** (JMSException e) {

**throw** e;

}

}

}

//消费者

**public** **class** Consumer {

**private** **static** **final** String ***USERNAME*** = ActiveMQConnection.***DEFAULT\_USER***;

**private** **static** **final** String ***PASSWORD*** = ActiveMQConnection.***DEFAULT\_PASSWORD***;

**private** **static** **final** String ***BROKEN\_URL*** = "failover://" + "tcp://" + "192.168.37.135" + ":" + 61616+"?jms.prefetchPolicy.queuePrefetch=1";

ConnectionFactory connectionFactory;

Connection connection;

Session session;

ThreadLocal<MessageConsumer> threadLocal = **new** ThreadLocal();

AtomicInteger count = **new** AtomicInteger();

**public** **void** init() {

**try** {

connectionFactory = **new** ActiveMQConnectionFactory(***USERNAME***, ***PASSWORD***, ***BROKEN\_URL***);

connection = connectionFactory.createConnection();

connection.start();

RedeliveryPolicy policy = ((ActiveMQConnection)connection).getRedeliveryPolicy();

policy.setBackOffMultiplier(2);

policy.setRedeliveryDelay(0);

policy.setUseExponentialBackOff(**true**);

policy.setMaximumRedeliveries(5);

session = connection.createSession(**false**, Session.***CLIENT\_ACKNOWLEDGE***);

} **catch** (JMSException e) {

e.printStackTrace();

}

}

**public** **void** getMessage(String disname) {

**try** {

Queue queue = session.createQueue(disname);

MessageConsumer consumer = **null**;

**if** (threadLocal.get() != **null**) {

consumer = threadLocal.get();

} **else** {

consumer = session.createConsumer(queue);

threadLocal.set(consumer);

}

Communicator ic = CommunicatorUtil.*getCommunicator*();

ObjectPrx base = ic.stringToProxy("receiverAccountServer:tcp -h localhost -p 9874");

**while** (**true**) {

**try** {

receiverAccountPrx proxy = receiverAccountPrxHelper.*checkedCast*(base);

TextMessage msg = (TextMessage) consumer.receive();

**if** (msg != **null**) {

Gson gson = **new** Gson();

Map<String, String> map = gson.fromJson(msg.getText(), **new** TypeToken<Map<String, String>>() {

}.getType());

String xid = map.get("xid");

String amount = map.get("amount");

String \_from = map.get("from");

String \_to = map.get("to");

String result = proxy.receive(xid, \_from, \_to, Double.*valueOf*(amount));

**if** ("0".equals(result)) {

System.***out***.println("消费成功:" + msg);

msg.acknowledge();

}**else**{

session.recover();

}

} **else** {

**break**;

}

} **catch** (Exception e) {

e.printStackTrace();

}

}

} **catch** (JMSException e) {

e.printStackTrace();

}

}

}