消费对账中心使用说明

**目录**

[一、 意义及目标 2](#_Toc488333634)

[二、原理及整体方案设计 2](#_Toc488333635)

[三、消费事务消息框架设计 3](#_Toc488333636)

[3.1、事务消息框架接口和用法介绍 5](#_Toc488333637)

[3.1.1、事务消息接口方法说明 5](#_Toc488333638)

[3.1.2、事件消息接口方法说明 6](#_Toc488333639)

[3.2、事务消息应用实例 6](#_Toc488333640)

[3.3、其他说明 9](#_Toc488333641)

[四、相关依赖（暂提供SNAPSHOT供测试试用） 9](#_Toc488333642)

## 意义及目标

随着公司后台从单体API web服务到分布式服务化的架构调整，业务从原先耦合的整体逐步拆解为一个个独立的业务模块，同时为了保持业务模块独立、稳定和对外封闭性（仅支持外部通过接口访问），原则上要求业务模块使用自己独立的存储和缓存，这必然导致原先一个业务功能需要跨多个业务模块调用而引入分布式事务问题。结合对于目前公司的业务，分布式事务主要会体现在消费类场景中（充值、送礼、购买座驾、购买VIP等），而且对用户体验影响也很大。如购买座驾会有扣币、增加秀币流水、增加座驾、增加购买流水等关键步骤，涉及API、用户中心、商场等多个模块或组件，这些关键步骤构成一个完整的事务，任何一个失败都会导致整体业务数据的不一致。

分布式事务问题解决方案世面上有很多，针对数据一致性的要求级别，方案复杂度也各异，如两阶段提交、TCC、消息异步消费模型等等，这些方案或者会引入过高的复杂度，或者与我们目前架构实践相去甚远、改造遥遥无期。因此我们设计了这样一个相对妥协的分布式事务方案（消费对账中心），即在不对目前系统伤筋动骨的基础上，及时定位分布式事务不一致数据、事后补齐的方案。

## 二、原理及整体方案设计

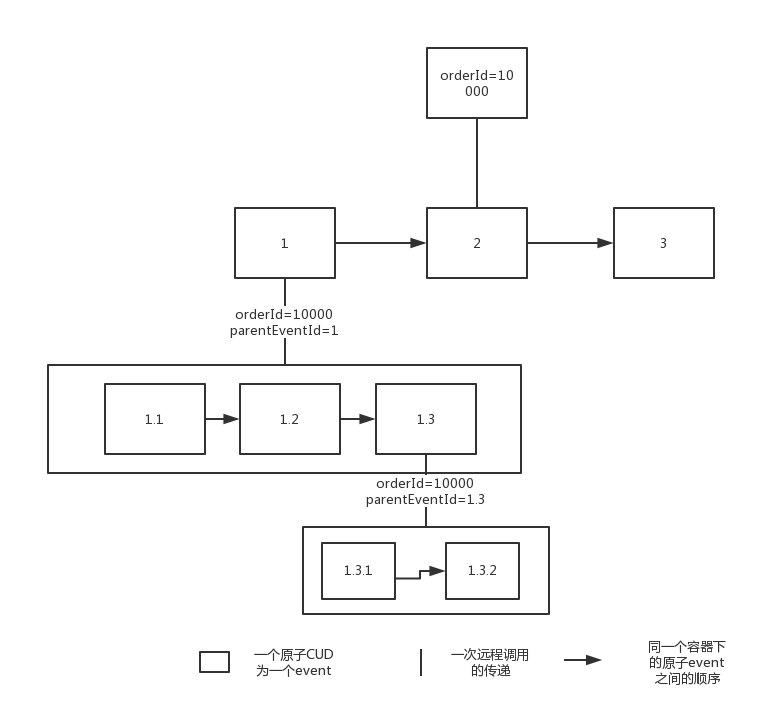
方案的核心思想就是通过整合分布式事务中的各个参与业务模块的事务记录来达到快速定位异常事务的目的，并根据事务具体场景结合事务处理中失败的具体步骤来人工（或自动）或重试失败操作、或补偿回滚来达到数据的最终一致性。比如一个功能接口A需要同步调用模块B、C来完成事务。

方案整体架构包括三个部分：1）事务消息埋点框架工具包，提供事务及关键事件埋点记录接口，保证事务消息唯一并确保送达；2）事务消息消费组件，实时幂等的消费事务消息，并将事务、事件记录入库，提供相应的查询统计接口；3）对账处理后台，提供后台可以查询异常事务，设置处理对账结果。详细架构如下图所示：



## 三、消费事务消息框架设计

一个事务消息有一个唯一的事务ID，事务消息中可以包含若干顺序步骤事件消息，事务或事件消息可以设置自定义属性、参数列表等信息，事务或事件结束时设置处理结果状态，本地方法处理过程结束时发送事务消息至MQ。事务开始时新建一个Transaction消息自动生成一个全局唯一的TransactionId(事务号), 事务处理过程每个关键步骤开始时新建一个Event消息，EventID（事件序号）按顺序递增，发起跨业务模块的RPC调用时自动传递当前的TransactionId和该处调用对应的EventID作为该远程调用的上下文发送至下级业务模块服务中。消息模型见下图所示：



例如用户购买座驾的消费行为，调用api的购买汽车接口，api调用商城模块进行买车操作，商城模块中调用了用户中心进行扣币和增加座驾的操作，而商城模块自身会操作数据添加流水等操作。那么其简单流程如下：

1）API: 新建transaction生成全局唯一的TansactionId=10000，调用商场模块购买座驾接口方法，新增步骤事件carService.buyCar，此时事件ID为1

2）商城模块：购买座驾接口方法，根据从RPC调用上下文中获取到本次transaction的TansactionId=10000,EventId=1.1，新建子transaction。调用用户中心扣秀币接口方法，新增步骤事件userService.decreaseMoney，此时调用步骤事件ID为1.1。

3）用户中心：加减秀币接口方法，根据从RPC调用上下文中获取到本次transaction的TansactionId=10000,EventId=1.1.1，新建子transaction。操作数据库扣秀币，新增步骤事件sql.incUserAssets，此时调用步骤事件ID为1.1.1。用户中心扣秀币事务结束，设置处理状态，发送MQ事务消息transaction(transactionId=10000,pEvent=1.1,eventList=[1.1.1,1.1.2])。

4）商城模块：步骤2）成功，调用用户中心增加用户资源接口方法，新增步骤事件userService.addResource此时调用埋点事件ID为1.2。

5）用户中心：增加用户资源接口方法，根据从RPC调用上下文中获取到本次transaction的TansactionId=10000,EventId=1.2.1，新建子transaction。操作数据库增加用户座驾，新增步骤事件sql.addUserResource，此时调用步骤事件ID为1.2.1。用户中心增加用户资源事务结束，设置处理状态，发送MQ事务消息transaction(transactionId=10000,pEvent=1.2,eventList=[1.2.1])。

6）商城模块：步骤4）成功，操作数据库新增购买座驾流水，新增步骤事件sql.buycar此时调用埋点事件ID为1.3。商城模块事务结束，设置处理状态，发送MQ事务消息

transaction(transactionId=10000,pEvent=1,eventList=[1.1,1.2,1.3])。

7) API：API购买座驾事务结束，设置处理状态，发送MQ事务消息

transaction(transactionId=10000,pEvent=null,eventList=[1])。

可以看出，一个涉及三次远程调用的分布式事务，在事务消息模型中，一次成功的购买会产生4条transaction消息。他们拥有相同的TransactionId和过程中详细的操作步骤及操作状态，从而形成了一棵完整的过程调用树，整个事务过程中任何一个步骤出现问题都很容易定位，再加上每一步失败原因和失败调用时参数的埋点，很容易就可以得出补帐的策略和动作，从而保证数据的最终一致性。

### 3.1、事务消息框架接口和用法介绍

事务消息框架对原先的MQ消息框架进行了扩展，因此基本的配置和使用方法与ActionHistory一样，这里对事务消息常用的埋点方法进行说明。

#### 3.1.1、事务消息接口方法说明

消息接口类KKHistory新增新建事务消息方法

/\*\*

\* 新建事务消息

\* **@param** type 事务类型名称

\* **@return** DefaultTransactionHistory事务消息实例

\*/

public TransactionHistory newTransactionHistory(String type)

/\*\*

\* 返回事务ID

\*/

public String getTransactionId()

/\*\*

\* 增加消息属性

\*/

Public void addAttribute(String key, Object val)

/\*\*

\* 批量增加消息属性

\*/

public void addAttributeMap(Map<String, Object> attrMap)

/\*\*

\* 设置事务结果状态

\*/

public void setStatus(String status)

/\*\*

\* 增加事件入口参数

\*/

Public void addParameter(Object paramObj)

/\*\*

\* 增加事件入口参数数组

\*/

public void addParameterArray(Object[] paramArr)

#### 3.1.2、事件消息接口方法说明

消息接口类KKHistory新增新建事务事件消息方法

/\*\*

\* 新建事务事件消息

\* **@param** type 事件类型名称

\* **@return** DefaultEventHistory事件消息实例

\*/

public EventHistory newEventHistory(String type)

/\*\*

\* 返回事务事件ID

\*/

public String getEventId()

/\*\*

\* 增加消息属性

\*/

Public void addAttribute(String key, Object val)

/\*\*

\* 批量增加消息属性

\*/

public void addAttributeMap(Map<String, Object> attrMap)

/\*\*

\* 设置事件结果状态

\*/

public void setStatus(String status)

/\*\*

\* 增加事件入口参数

\*/

Public void addParameter(Object paramObj)

/\*\*

\* 增加事件入口参数数组

\*/

public void addParameterArray(Object[] paramArr)

### 3.2、事务消息应用实例

事务消息埋点方式类似cat，具体步骤如下：

        1）新建事务消息，指定事务名称（自动产生唯一的流水号）

        2）增加事务相关的业务信息

        3）新建事件消息（自动添加到当前事务消息中），增加事件相关业务信息

        4）设置事件状态（成功与否或失败原因）

        5）重复步骤3、4

        6）设置事务状态（成功与否或失败原因）

        7）事务流水完成，投递流水至MQ

Demo及代码示例如下：

LocalService为事务发起方，对应api。

RemoteProvider为业务处理方，对应模块。

***public******class*** *LocalService {*

***public******boolean*** *buyCar() {*

*TransactionHistory trans = kkhistory.newTransactionHistory("api-buyCar");*

*trans.addAttribute("userid", 123456);*

*// trans.addAttribute("", ""); 在事务中添加信息*

***boolean*** *ret =* ***false****;*

***try*** *{*

*EventHistory callBuyEvent = kkhistory.newEventHistory("call-module-buyCar");*

*ret = provider.buyCar(); //RPC 远程调用*

*callBuyEvent.setStatus(ret ? History.SUCCESS : History.FAIL);*

***if*** *(ret) {*

*EventHistory insertHistoryEvent = kkhistory.newEventHistory("insertHistory");*

*ret = insertHistory(); //本地数据操作*

*insertHistoryEvent.setStatus(ret ? History.SUCCESS : History.FAIL);*

*}*

***if*** *(ret) {*

*//...*

*//more event*

*}*

*trans.setStatus(ret ? History.SUCCESS : History.FAIL);*

*}* ***catch*** *(Exception e) {*

*//log error*

*ret =* ***false****;*

*trans.setStatus(e.getMessage());*

*}* ***finally*** *{*

*System.out.println("payload: " + trans.getPayload());*

*trans.complete(); // 发送整条流水记录到MQ*

*}*

***return*** *ret;*

*}*

*}*

其中ret = provider.buyCar();调用RemoteProvider的buyCar方法。一个EventHistory即为一个event,orderId和event会在创建时自动计算和分配。

RemoteProvider的埋点如下。

***public******class*** *RemoteProviderImpl* ***implements*** *RemoteProvider {*

***private******static*** *Random rd =* ***new*** *Random();*

*@Autowired*

***private*** *KKHistory kkHistory;*

***public******boolean*** *buyCar() {*

***if*** *(!checkMoney()) {*

***return******false****;*

*}*

*TransactionHistory trans = kkHistory.newTransactionHistory("rpc-buyCar");*

*Object[] objects =* ***new*** *Object[2];*

*objects[0] = 7502500;*

*objects[1] = 1000l;*

***boolean*** *ret =* ***false****;*

***try*** *{*

*EventHistory event = kkHistory.newEventHistory("decreaseMoney");*

*event.addParameterArray(objects);*

*ret = decreaseMoney(7502500, 1000l);*

*event.setStatus(ret ? History.SUCCESS : History.FAIL);*

***if*** *(ret) {*

*EventHistory addEvent = kkHistory.newEventHistory("addShowMoney");*

*try{*

*ret = addShowMoneyHistory();*

*addEvent.setStatus(ret ? History.SUCCESS : History.FAIL);*

*} catch(Exception e)*

*{*

*addEvent.setStatus(e.getMessage());*

*}*

*}*

***if*** *(ret) {*

*EventHistory addCarEvent = kkHistory.newEventHistory("addCar");*

*ret = addCar();*

*addCarEvent.setStatus(ret ? History.SUCCESS : History.FAIL);*

*}*

*trans.setStatus(ret ? History.SUCCESS : History.FAIL);*

*}* ***catch*** *(Exception e) {*

*ret =* ***false****;*

*e.printStackTrace();*

*}*

***finally*** *{*

*System.out.println("payload: " + trans.getPayload());*

*trans.complete();*

*}*

***return*** *ret;*

*}*

### 3.3、其他说明

订单号说明：

由一个long型+char型拼接的string串组成。即占64+16位。

Long为snowFlake改造。

第一位预留，前41位为时间戳与预设时间戳(2017-7-7)差值，中间10位为ip末位，最后12作为自增序列。

Char为pid。

通过这样的组合可以保证orderId唯一并且有一定意义。理论单实例QPS为2的12次\*100。

## 四、相关依赖（暂提供SNAPSHOT供测试试用）

使用流水模型需要以下包的版本:

Kkrpc相关1.1.1-SNAPSHOT

<parent>

<groupId>com.melot.module</groupId>

<artifactId>kkrpc-parent</artifactId>

<version>1.1.1-SNAPSHOT</version>

</parent>

kk-transaction-hist

<dependency>

<groupId>com.melot.common</groupId>

<artifactId>kk-transaction-hist</artifactId>

<version>1.1.0-SNAPSHOT</version>

</dependency>

melot-utils

<dependency>

<groupId>com.melot.common</groupId>

<artifactId>melot-utils</artifactId>

<version>1.0.3-SNAPSHOT</version>

</dependency>