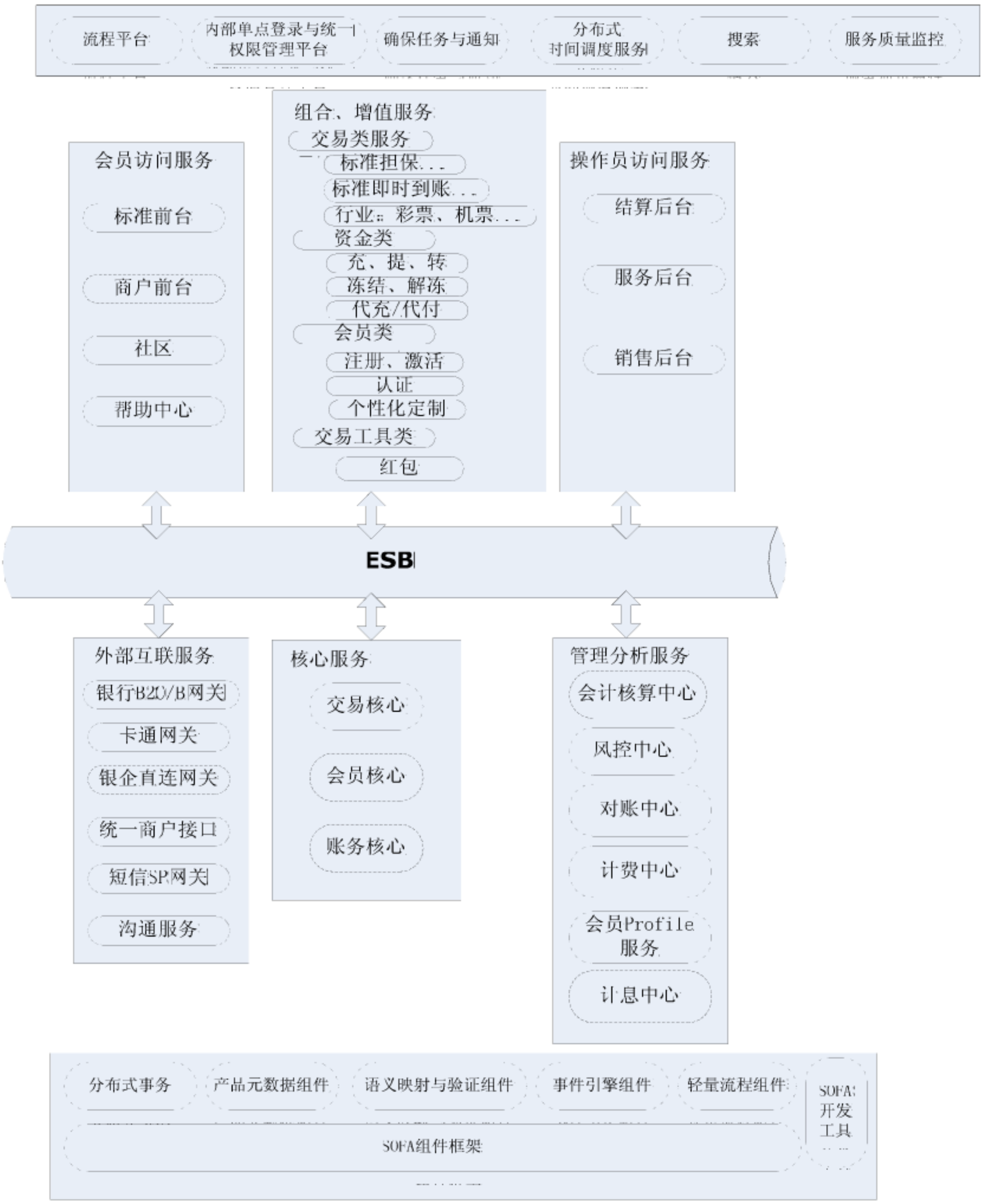
支付宝分布式事务架构设计草案

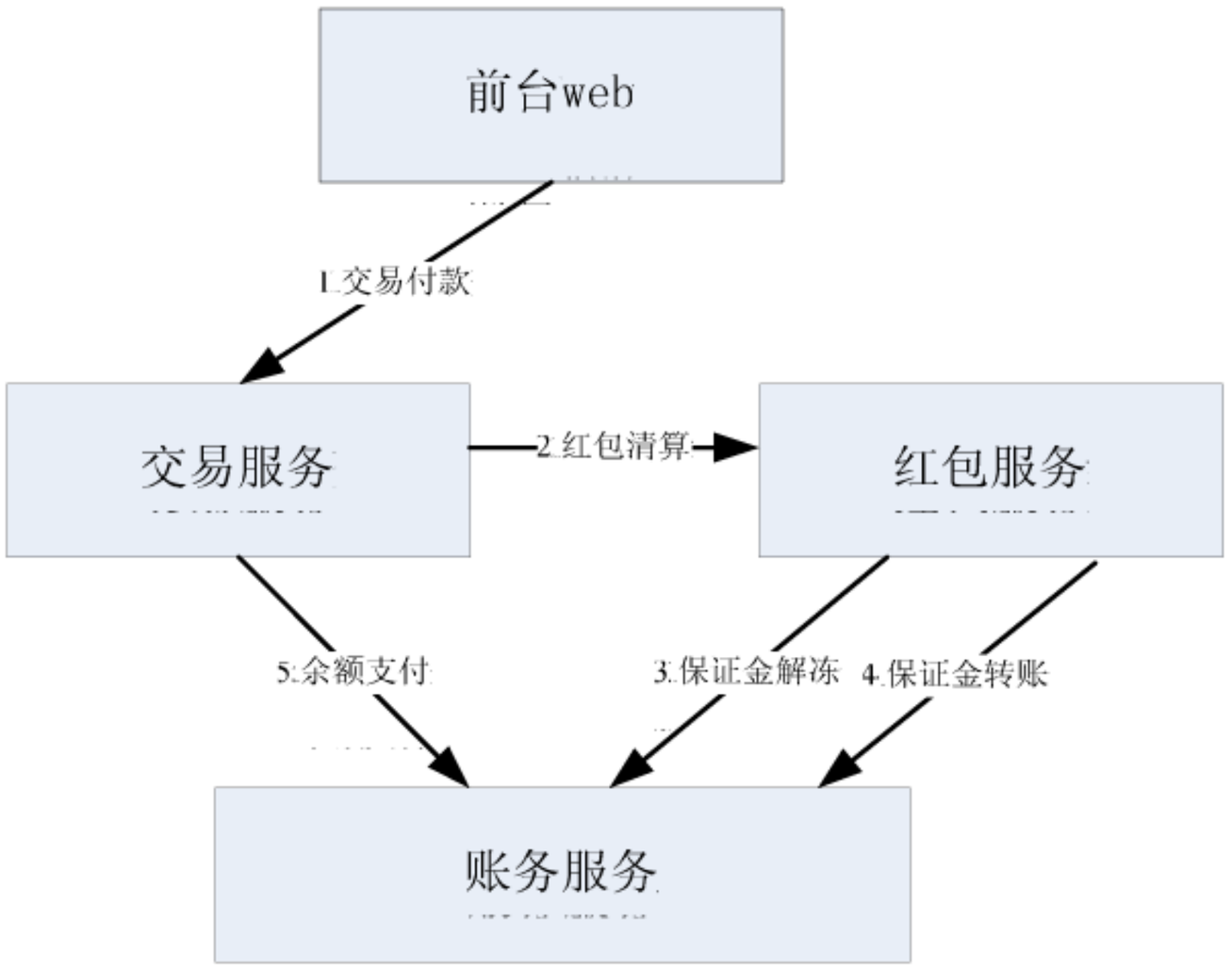
1 背景介绍

为了应对快速变化的市场需求、持续增长的业务量，支付宝系统需要基于 SOA 进行构建与改造，以应对系统规模和复杂性的挑战，更好地进行企业内与企业间的协作。

基于 SOA 图景，整个支付宝系统会拆分成一系列独立开发、自包含、自主运行的业务服务，并将这些服务通过各种机制灵活地组装成最终用户所需要的产品与解决方案。支付宝系统将会有类似下图所示的 SOA 模型：



在 SOA 的系统架构下，一次业务请求将会跨越多个服务。我们举一个使用红包+余额进行交易付款的例子来说明。



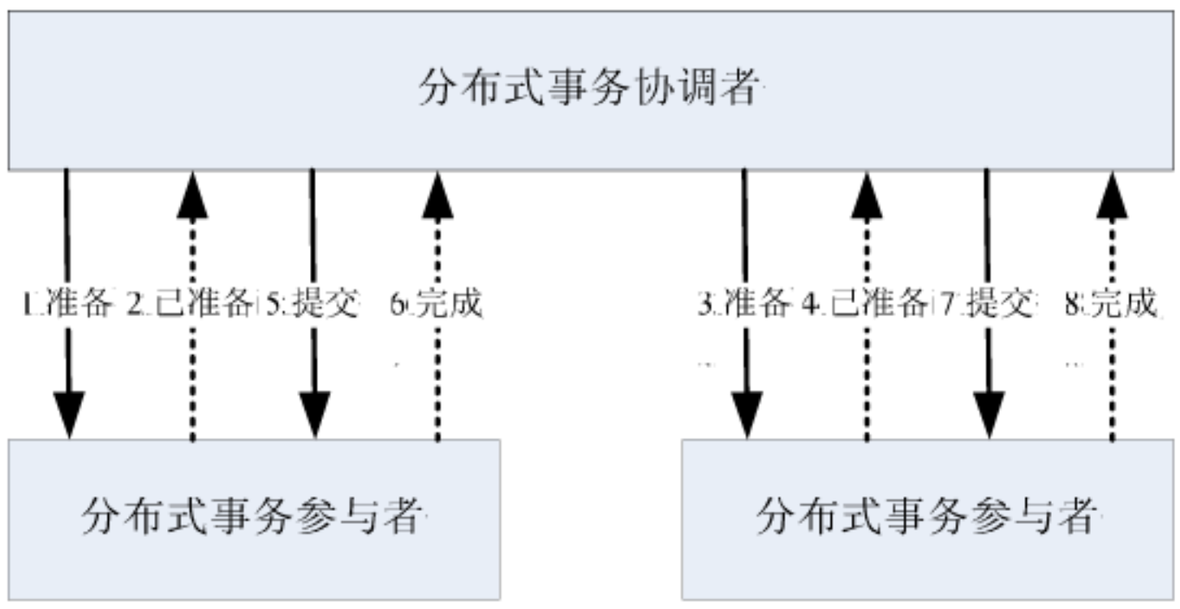
在多个服务协同完成一次业务时，由于业务约束（如红包不符合使用条件、账户余额不足等）、系统故障（如网络或系统超时或中断、数据库约束不满足等），都可能造成服务处理过程在任何一步无法继续，使数据处于不一致的状态，产生严重的业务后果。

传统的基于数据库本地事务的解决方案只能保障单个服务的一次处理具备原子性、隔离性、一致性与持久性，但无法保障多个分布服务间处理的一致性。因此，我们必须建立一套分布式服务处理之间的协调机制，保障分布式服务处理的原子性、隔离性、一致性与持久性。

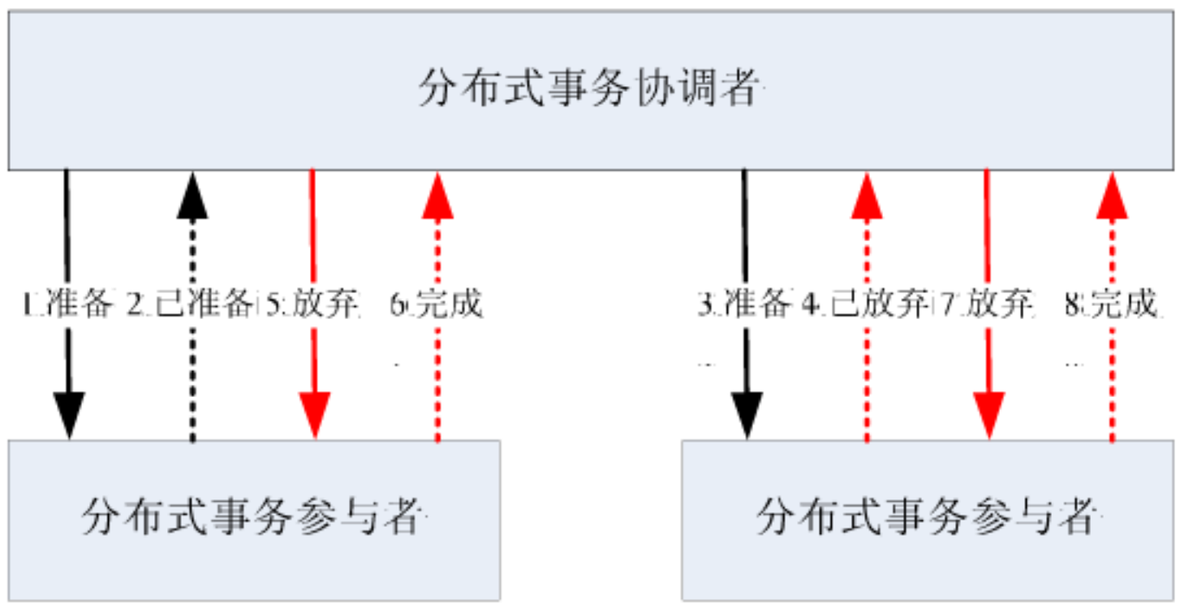
2 基本原理

2.1两阶段提交协议(2PC)

传统的分布式事务处理是基于两阶段提交协议的。两阶段提交协议的原理如下图所示：



成功的两阶段提交(2PC)示例

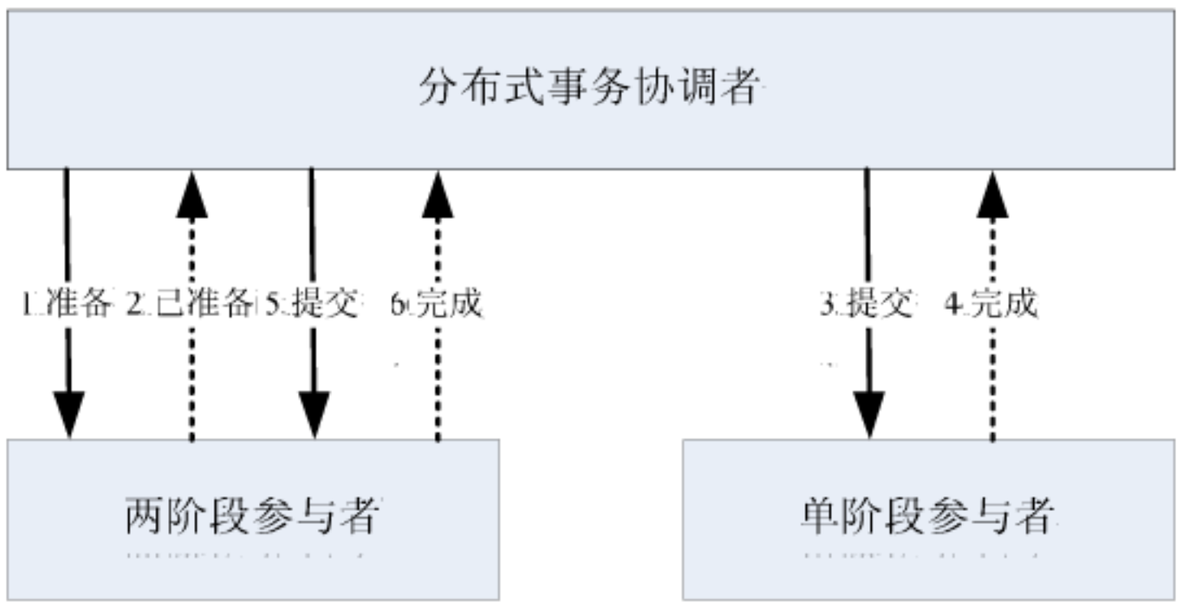


失败的两阶段提交(2PC)示例

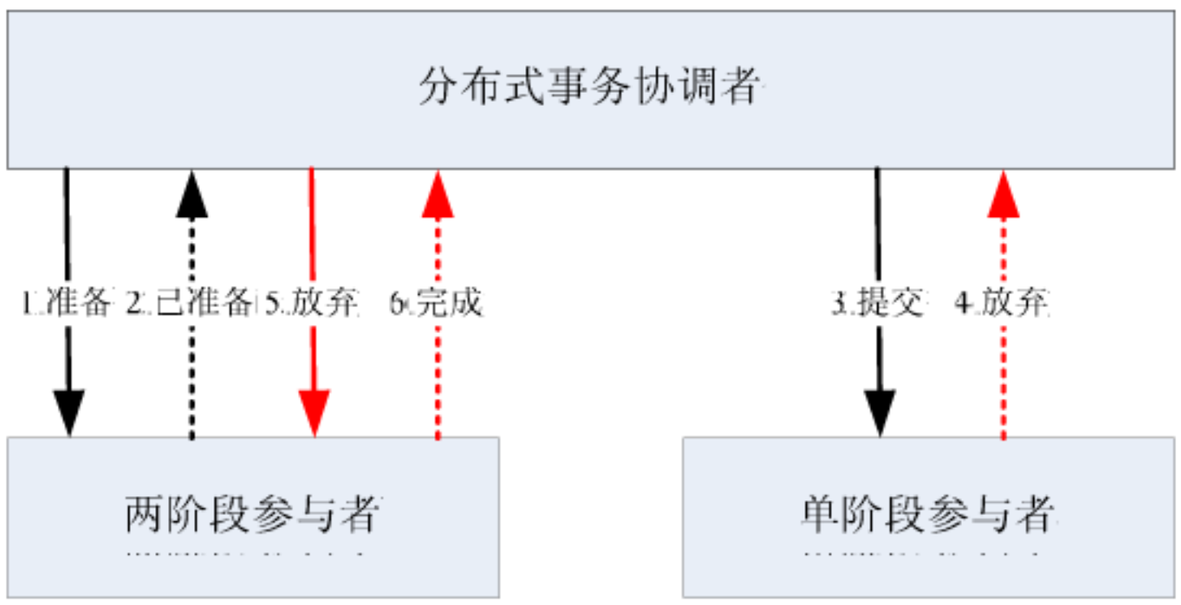
从上图可见，两阶段提交协议的关键在于“准备”操作。分布式事务协调者在第一阶段通过对所有的分布式事务参与者请求“准备”操作，达成关于分布式事务一致性的共识。分布式事务参与者在准备阶段必须完成所有的约束检查、并且确保后续提交或放弃时所需要的数据已持久化。在第二队段，分布式事务协调者根据之前达到的提交或放弃的共识，请求所有的分布式事务参与者完成相应的操作。

2.2最末参与者优化(LPO)

两阶段提交协议要求分布式事务参与者实现一个特别的“准备”操作，无论在资源管理器（如数据库）还是在业务服务中实现该操作都存在效率与复杂性的挑战。因此，两阶段提交协议有一个重要的优化，称为“最末参与者优化”（Last Participant Optimization），允许两阶段提交协议中有一个参与者不实现“准备”操作（称为单阶段参与者）。最末参与者优化的原理如下图所示：



最末参与者优化(LPO)示例
成功场景



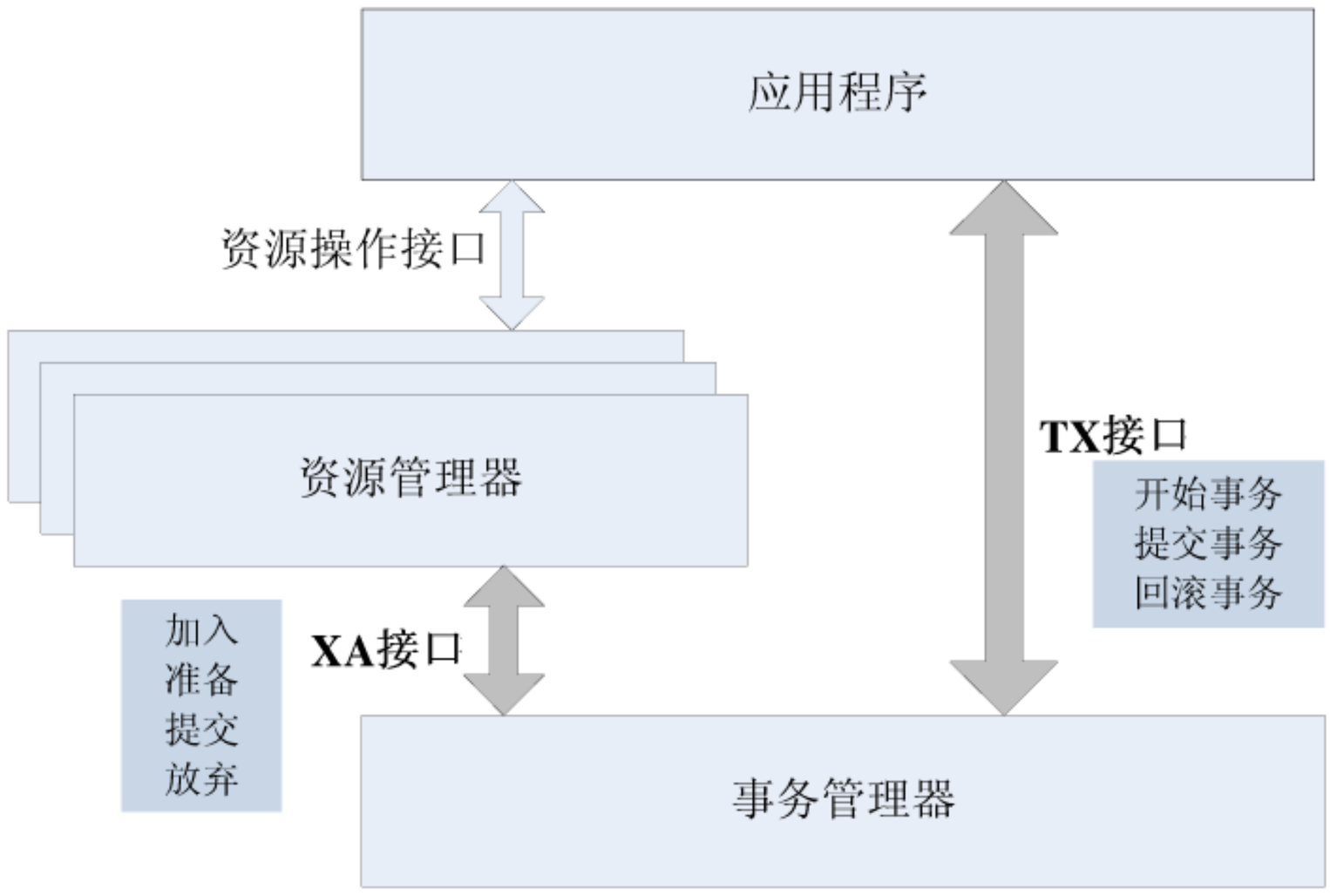
最末参与者优化(LPO)示例
失败场景

从上图可见， LPO 中，单阶段参与者不需要实现准备操作，只需要提供标准的提交操作即可。分布式事务协调者必须等其余两阶段参与者都准备好之后，再请求单阶段参与者提交，单阶段参与者的提交结果将决定整个分布式事务的结果。本质上，LPO 是将最后一个参与者的准备操作与提交/放弃操作合并成一个提交操作。

最末参与者优化方案使得我们能够利用支付宝业务的特点，尽量简化分布式事务的实现。

2.3X/Open 模型

X/Open 组织为基于两阶段协议的分布式事务处理系统提出了标准的系统参考模型（X/Open 事务模型）、以及不同组件间与事务协调相关的接口，使不同厂商的产品能够互操作。X/Open 事务模型如下图所示：



从上图可以看出，X/Open 模型定义了两个标准接口：TX 接口用于应用程序向事务管理器发起事务、提交事务和回滚事务（即确定事务的边界和结果）；XA 接口用于事务管理器将资源管理器（如数据库、消息队列等）加入事务、并控制两阶段提交。

事务管理器一般由专门的中间件提供、或者在应用服务器中作为一个重要的组件提供。资源管理器如数据库、消息队列一般也会提供 XA 支持。通过使用符合 X/Open 标准的分布式事务处理，能够简化分布式事务类应用的开发。

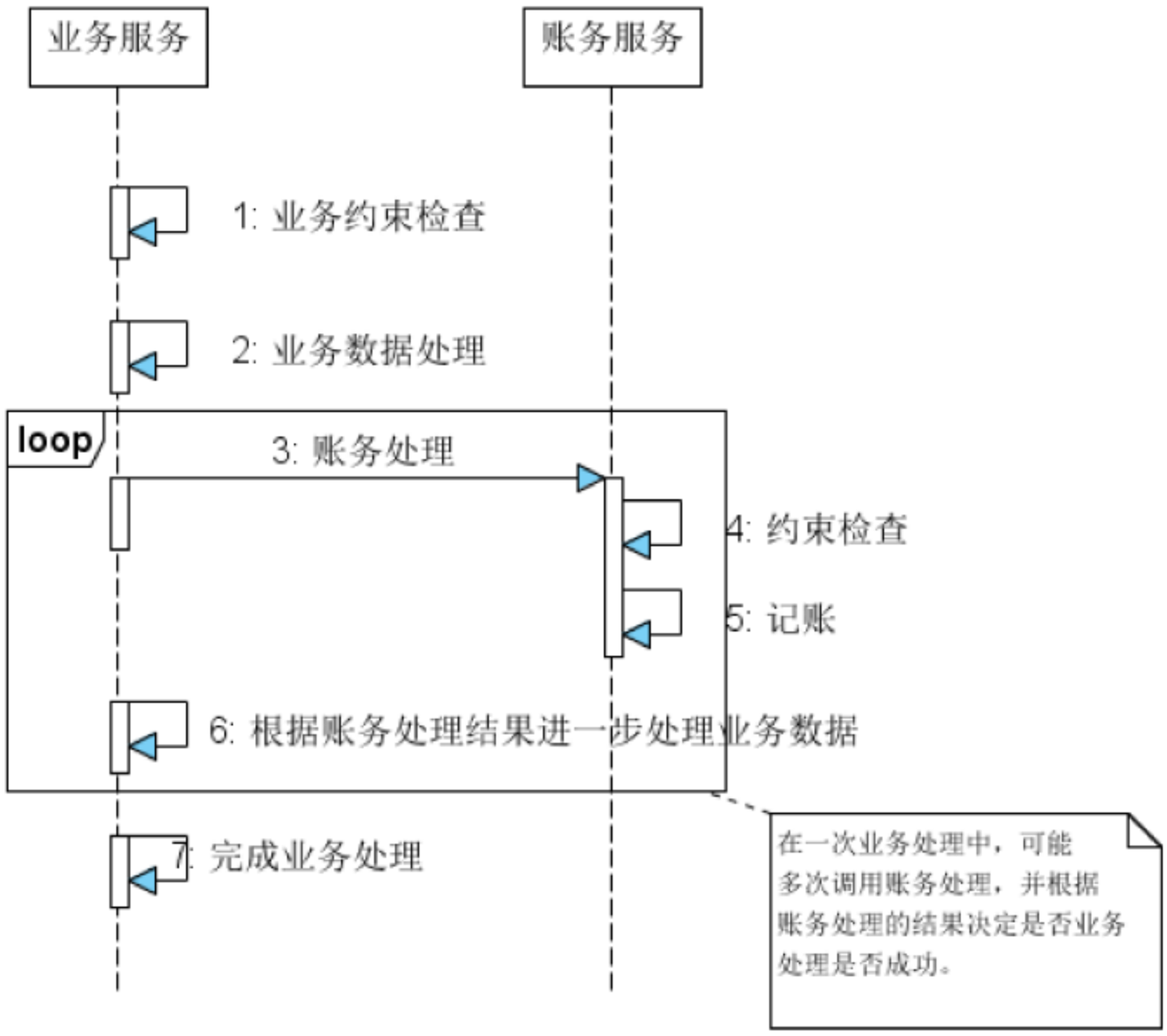
但在现实中，事务管理器与资源管理器对 TX/XA 协议的实现上存在效率、可靠性与伸缩性上的风险；在两阶段提交协议执行过程中的异常恢复起来也很困难；而且在 SOA 体系下当事务需要跨越多个服务（而不是多个资源管理器）时，事务的协调与恢复会变得非常复杂。

在标准分布式事务管理框架不能满足需要的情况下，我们需要根据支付宝业务与系统的特点，设计并实现自己的分布式事务处理机制。下一节介绍支付宝分布式事务处理的基础模型。

3 基础模型

3.1典型业务处理模式

支付宝的主体业务基本都会在一次业务处理中进行一次或多次账务处理。典型的业务处理模式如下图所示：



这种模式可以概括如下：

(1) 支付宝的主体业务服务在执行过程中一般都会涉及到一次或者多次的账务处理。

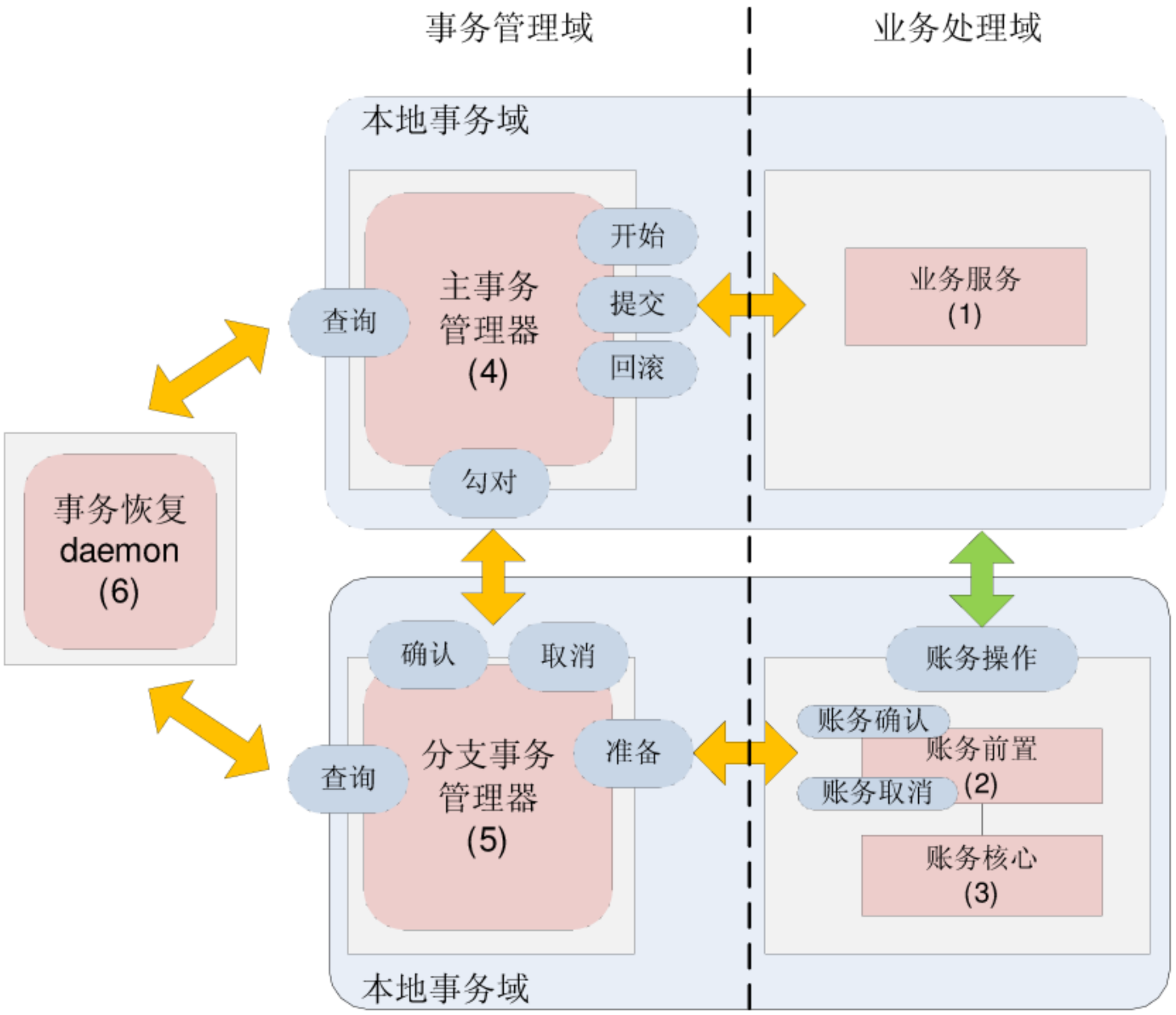
(2) 业务服务与账务服务对业务处理的最终结果有同等的决定权，两者都能够使业务处理失败。

(3) 当一次业务处理中涉及到超过两个参与者时，附加的参与者一般对业务处理的最终结果没有决定权，但它们会根据业务处理的最终结果完成自己的处理。例如，很多业务在完成之后都涉及到收费的处理，但一般收费不成功不会影响业务处理本身的结果。

根据两参与者的特点，以及账务服务的中心地位，我们可以根据“两阶段提交协议”以及“最末参与者优化”原理，设计支付宝分布式事务处理的基础模型。

3.2基础模型

这一基础模型如下图所示：



在上图中，各个主要组件的职责如下：

(1) 业务服务

业务服务负责具体业务处理，如交易服务、红包服务等等。

(2) 账务前置

账务前置负责接收、检查并缓冲从业务服务发起的账务请求。

(3) 账务核心

负责记账并更新分户余额。

(4) 主事务管理器

与业务服务位于同一个本地事务域，负责主事务的启动、提交与回滚。

(5) 分支事务管理器

与账务服务位于同一个本地事务域，负责分支事务的准备，确认与取消。

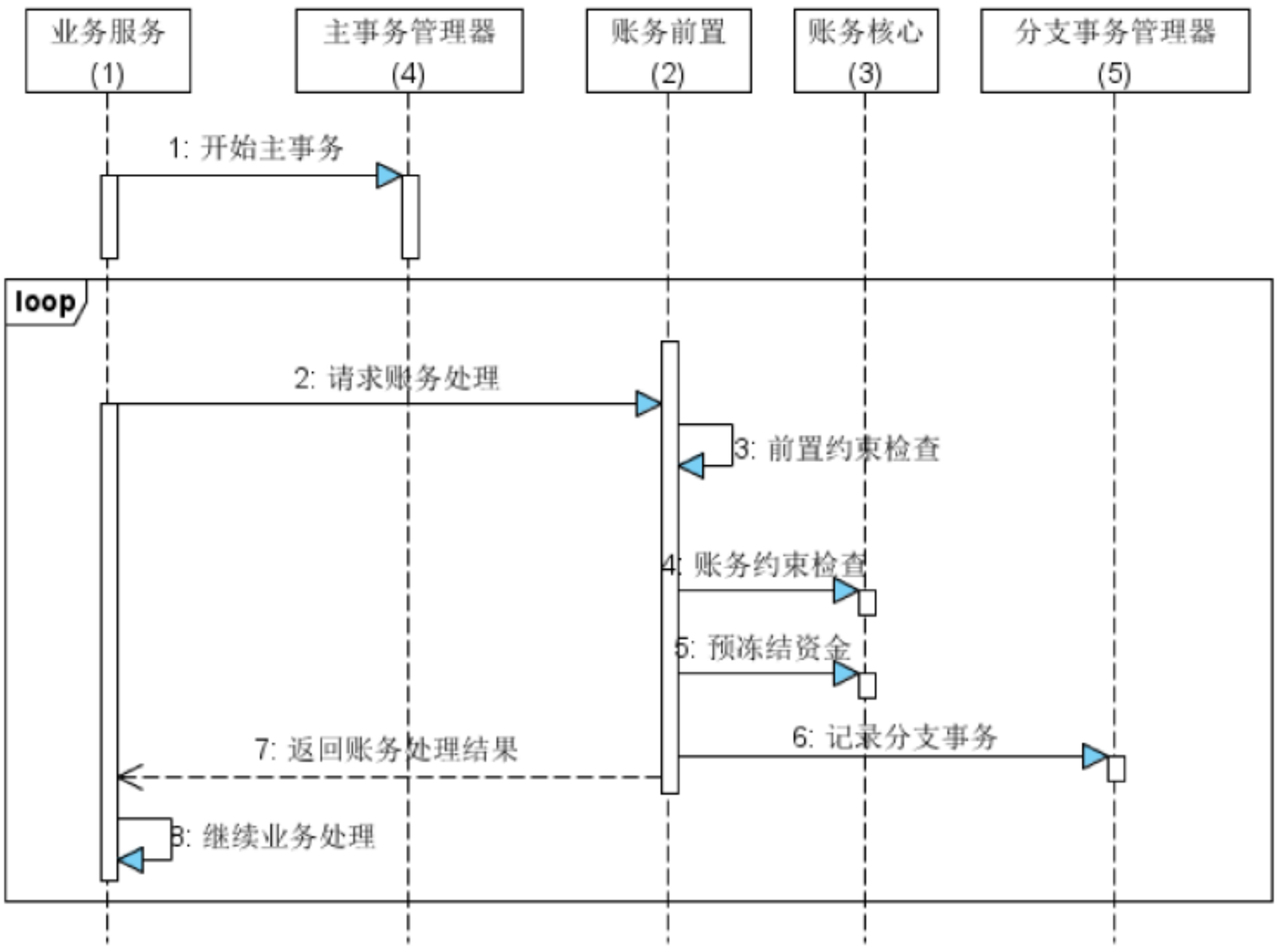
(6) 事务恢复 daemon

定时运行，负责恢复处于已准备状态，但在指定时间阈值内尚未确认或者取消的事务。

下面我们介绍上述组件如何通过协作完成一次包含账务的业务处理。

3.3准备阶段

下图显示在“准备”阶段各个组件间的交互过程。



1. 业务服务(1)首先向主事务管理器请求开始主事务，此时，主事务管理器启动本地事务，按照一定规则生成一个对本次处理唯一的 txId，记录主事务日志，并在事务上下文中记录 txId，这个 txId 在整个分布式事务的生命周期中用于建立主事务与分支事务之间的对应关系，并用于业务重复性检查。
2. 业务服务向账务前置发送账务处理请求。主事务管理器能够拦截本次请求，并将主事务 ID(txId)附加到账务处理请求的上下文中，一起发送给账务前置。
3. 账务前置进行前置约束检查。前置约束检查至少要保证：a. 事务 Id 有效；b. 业务不重复。前置约束检查前，相关账户必须锁定（除特定账户外、如中间账户等）。
4. 账务前置调用账务核心进行账务约束检查。账务约束检查至少要保证：a. 账户状态正确；b. 账户资金足够；c. 其它账务约束满足。账务约束检查时必须考虑到在本事务中尚未到达的资金，因此这是检查中比较特殊的地方，需要恰当处理。
5. 账务前置调用账务核心进行资金冻结。对于完成本次账务处理需要的资金，需要一种特殊的方式冻结起来，但这种冻结没有业务含义，因此，不应该记录资金冻结日志，只是在 freeze_amount 中增加这笔冻结资金，确保账务确认阶段能够使用这笔资金。如果本次账务处理所需要的资金尚未到达，则不需要冻结。
6. 账务前置调用分支事务管理器记录分支事务日志。分支事务日志中记录了本次账务处理的内容以及冻结的金额，在确认阶段，分支事务管理器会根据分支事务日志中记录的内

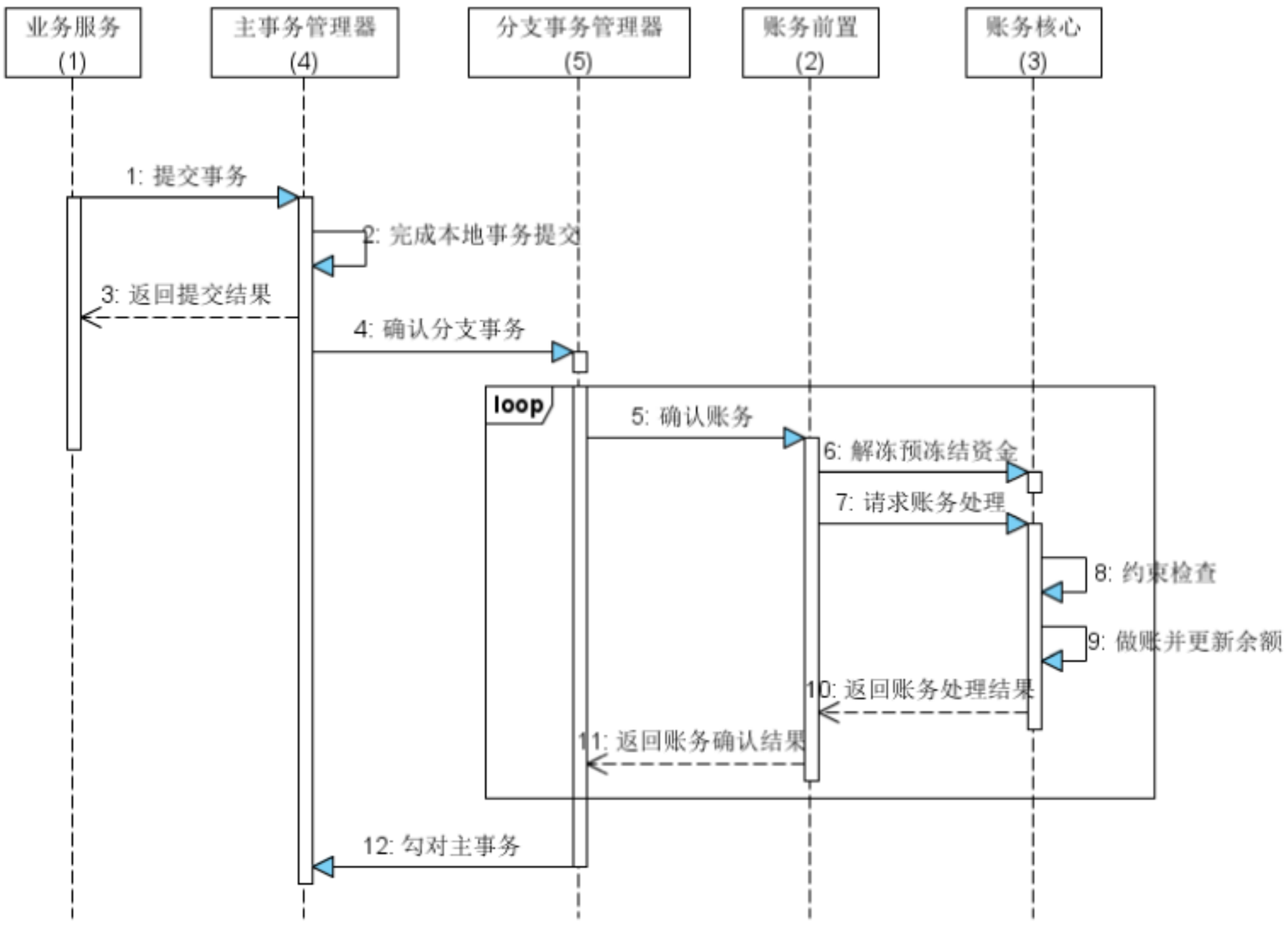
容驱动账务系统完成预冻结金额的解冻与实际的账务处理。

7. 账务前置向业务服务返回账务处理的结果。
8. 业务服务根据账务处理的结果继续进行业务处理。

在一次业务处理过程上，上述交互过程允许进行发生多次。但为了控制远程调用的成本，也可以将账务请求打成包合并发送给账务前置。

3.4确认阶段

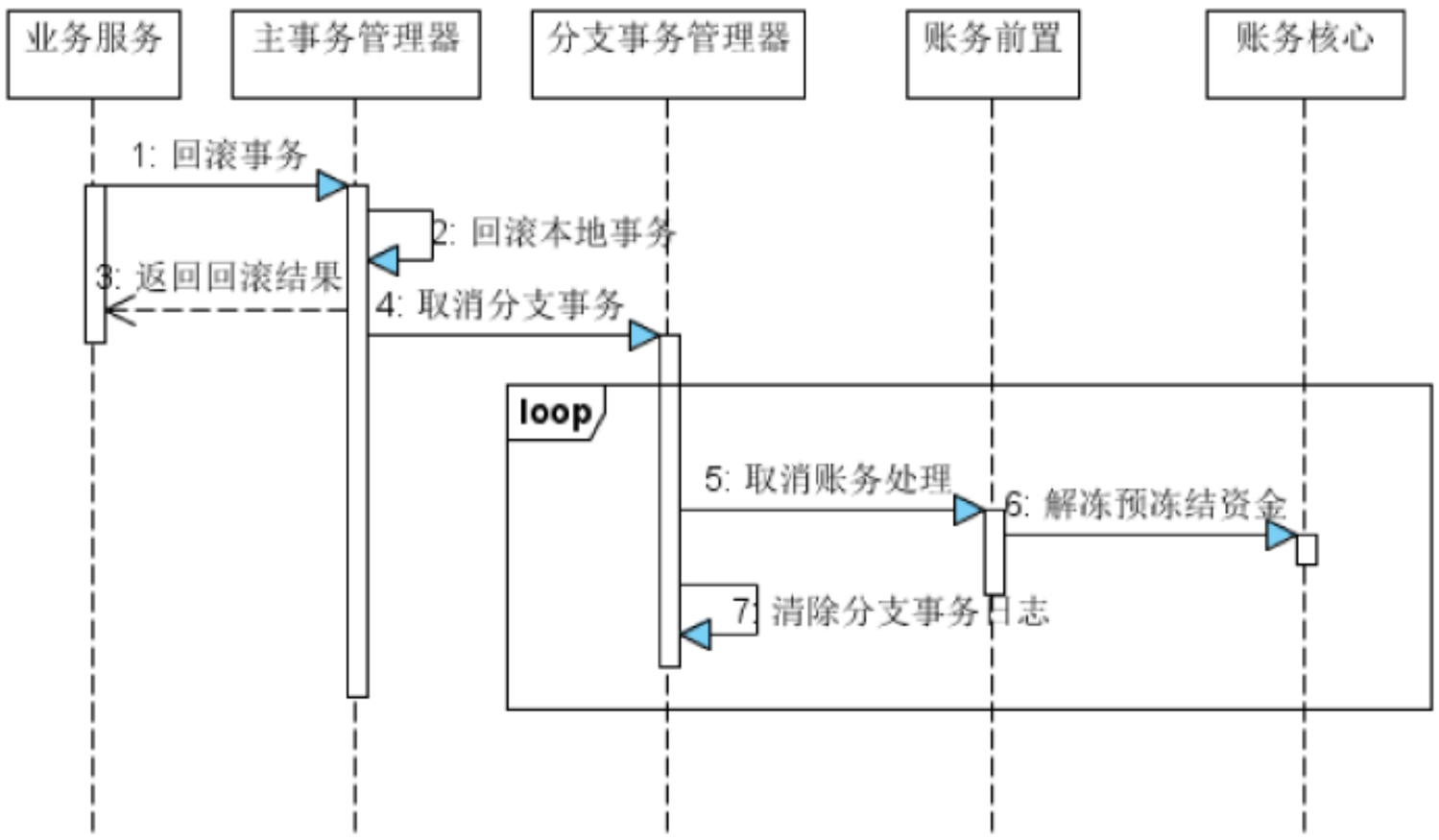
下图显示在“确认”阶段各个组件的交互过程。



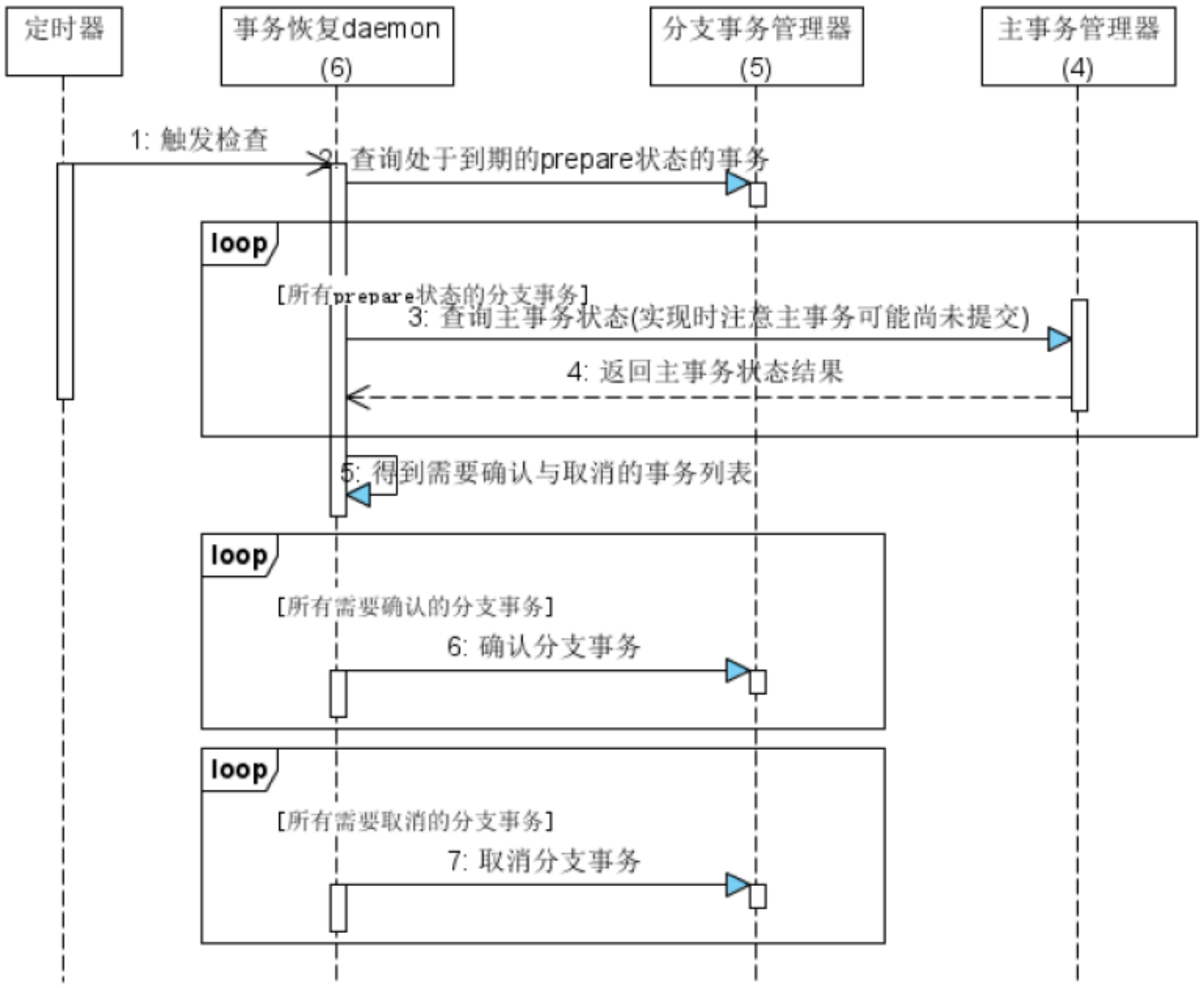
1. 业务服务请求主事务管理器提交事务。
2. 主事务管理器首先完成本地事务的提交。
3. 主事务管理器向业务系统返回事务提交的结果。
4. 主事务管理器向分支事务管理器确认分支事务结果。
5. 分支事务管理器顺序处理对应于本次分布式事务的每一条分支事务日志，对每一条分支事务日志，调用账务前置确认该次处理。
6. 账务前置首先请求账务核心解冻预冻结的资金。
7. 账务前置请求账务核心进行账务处理。
8. 账务核心对本次账务处理进行约束检查。对于特定的检查（比如账户状态是否有效等）是否需要做，视业务而定。
9. 账务核心进行账务处理，包含记录账务日志并更新账户余额等。其它正常账务处理中需要执行的工作也同样需要做。

10. 账务核心向账务前置返回账务处理的结果。
11. 账务前置向分支事务管理器返回账务确认的结果，分支事务管理器提交本地事务。
12. 分支事务管理器请求主事务管理器勾对主事务。勾对的方式可以是删除主事务记录，也可以是为主事务记录打上标志。

3.5回滚阶段



1. 业务服务请求主事务管理器回滚事务。
2. 主事务管理器回滚本地事务。
3. 主事务管理器向业务系统返回回滚结果。
4. 主事务管理器向分支事务管理器请求取消分支事务。
5. 分支事务管理器针对每一条分支事务明细，向账务前置请求取消账务处理。
6. 账务前置向账务核心请求解冻预冻结资金。
7. 分支事务管理器清除分支事务日志。



3.6恢复阶段

1. 定时器定期触发事务恢复 daemon 程序，进行事务恢复。定期的间隔可以是每分钟一次。
2. 事务恢复 daemon 程序向分支事务管理器查询处于已到期的处于 prepare 状态的分支事务。已到期的具体时间一般是允许的最大事务长度，比如 90 秒。
3. 事务恢复 daemon 针对每条到期的处于 prepare 状态的分支事务，向主事务管理器查询主事务状态。
4. 主事务管理器向事务恢复 daemon 返回主事务状态。
5. 事务恢复 daemon 根据查询主事务状态的结果，得到需要确认与取消的分支事务列表。如果主事务状态是已提交，则表明分支事务需要提交。如果主事务状态不存在（即没有主事务日志），则表明主事务已回滚。这里需要特别注意的是，如果主事务执行时间过长，则可能在查询时主事务还处于执行阶段，尚未提交。通过限制主事务长度，可以解决这个问题。需要考虑一下有无更好更安全的方案。
6. 事务恢复 daemon 针对每条需要确认的分支事务，请求分支事务管理器确认事务。
7. 事务恢复 daemon 针对每条需要取消的分支事务，请求分支事务管理器取消事务。

3.7预冻结款与未达款计算

在准备阶段，账务前置需要计算出预冻结金额，并请求账务核心冻结该部分金额，确保在确认阶段相关的账务处理有足够的金额。在一次分布式事务处理中，业务服务可能多次请求账

务处理，资金可能在多个账户间发生转移。由于在准备阶段，账务前置只负责预冻结资金，而不会进行实际的资金转移，因此对于资金转入账户，在准备阶段存在“未达款”（应该转入但实际未转入的资金）。在计算预冻结金额时，必须考虑未达款。

假设在一次业务处理中，有以下三次账务处理：

(1) A - (100)-> B: 从 A 账户转 100 元至 B 账户

(2) B - (50)-> C: 从 B 账户转 50 元至 C 账户

(3) C –(200)-> D: 从 C 账户转 200 元至 D 账户

则预冻结款与未达款的计算如下表所示：

No#	A		B		C		D	
	预冻结	未达	预冻结	未达	预冻结	未达	预冻结	未达
1	100	0	0	100	0	0	0	0
2	100	0	0	50	0	50	0	0
3	100	0	0	50	150	0	0	200

从上表可以看出，账户前置必须在每一次处理时计算并更新本次处理过程中所涉及到的各个账户的预冻结款与未达款。本次处理中涉及到的所有账户的预冻结款之和与未达款之和相同。

账务服务除了提供资金转移操作外，还提供资金冻结与解冻操作。账务前置在准备阶段需要针对资金冻结与解冻操作进行处理。

当请求资金冻结时，首先需要以预冻结的方式将相关资金冻结起来，由于该笔资金被冻结后只能专款专用（即用于指定类型的资金冻结），因此，在账务前置中，需要记录一笔该专项冻结的未达款。

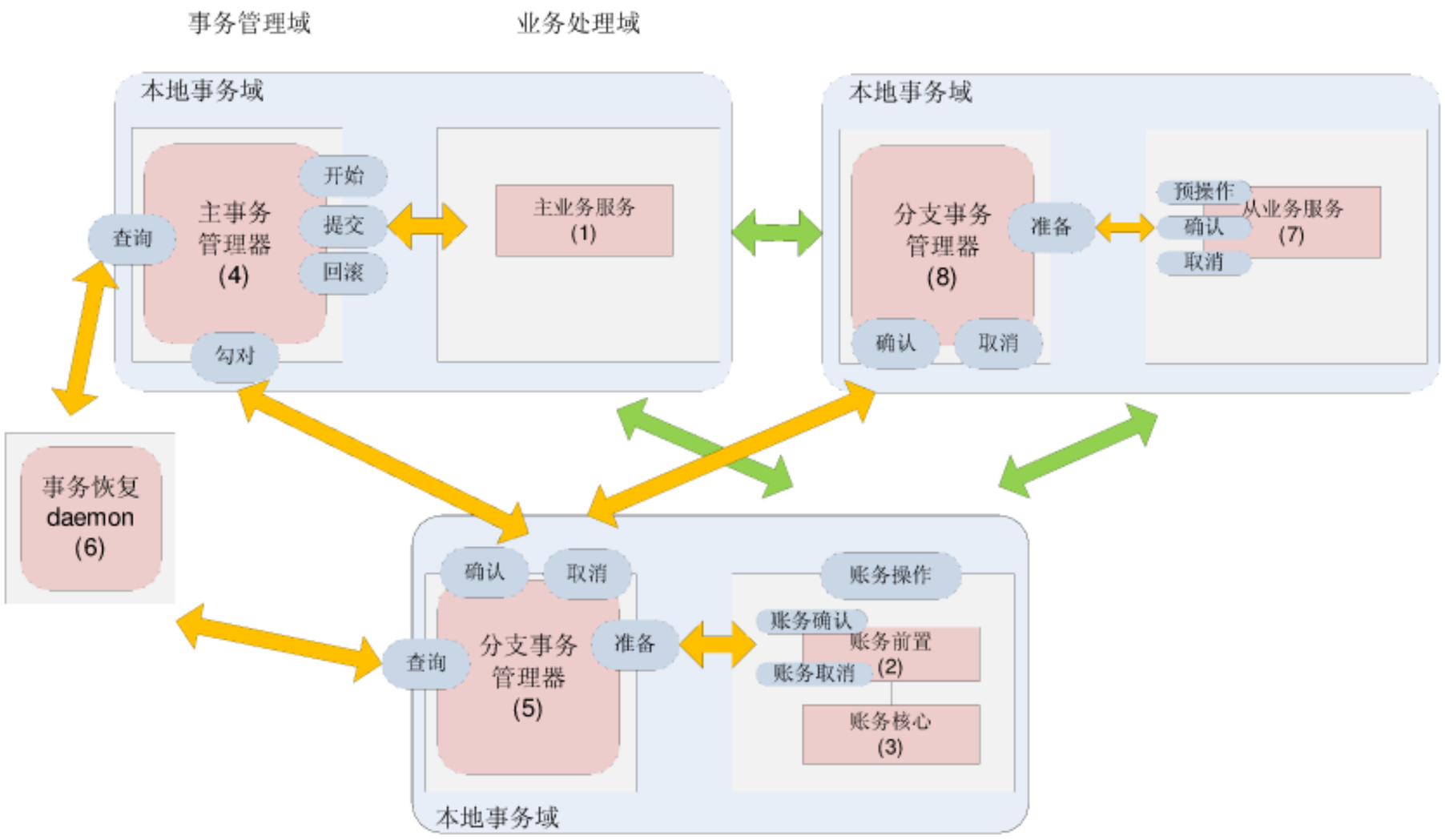
当请求资金解冻时，可解冻的额度取决于目前该专项冻结的资金总额加上该专项冻结未达款的总额，如果满足解冻条件，需要针对该专项冻结记录一笔预解冻。

对于充值类业务，由于只涉及到一个账户，因此只需要记录未达款即可。对于提现类业务，由于只涉及到一个账户，只需要预冻结即可。

对于特殊账户，如中间账户、特定的大账户，可以不进行预冻结、未达款计算。

4 多参与者模型

对于某些业务处理，作为独立服务的参与者可能不止一个，在这种情况下，就需要建立多参与者间的分布式事务处理模型。



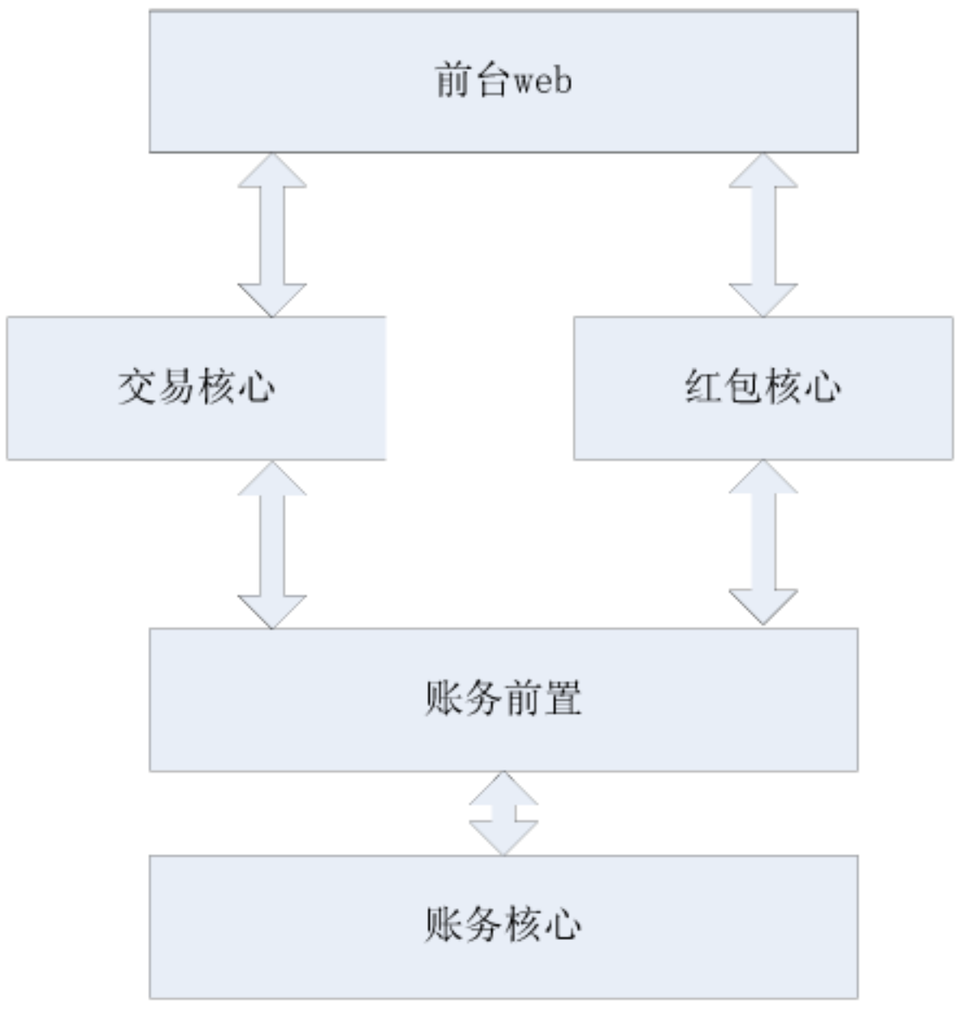
上图显示的是一个三参与者的模型。相比基础模型，上图中引入了“从业务服务”(7)，在从业务服务上也有一个分支事务管理器(8)。从业务服务的执行结果可以影响主业务服务的执行结果，例如，若红包支付不成功，则交易的付款操作不成功。

凡是作为“从业务服务”参与到业务处理中、并影响主业务服务执行结果的服务，都需要进行改造，将对应的业务操作分解为预操作、确认与取消三个操作。预操作完成业务处理之前的约束检查并锁定资源，但不实际进行业务处理；确认操作完成实际业务处理，取消操作完成资源的解锁。

如果从业务服务的执行结果不影响主业务服务的执行结果，则可以不参与到分布式事务处理中。在主业务处理完成之后，可以通过异步确保通知/ESB 来驱动从业务服务完成处理。比如交易成功后的收费，收费服务的执行结果不会影响交易执行结果，就不必参与到分布式事务处理中。

5 实例分析

在本节中，我们以交易余额加红包支付为例，说明在分布式事务控制下的业务处理过程。



红包加余额支付的处理大纲是：

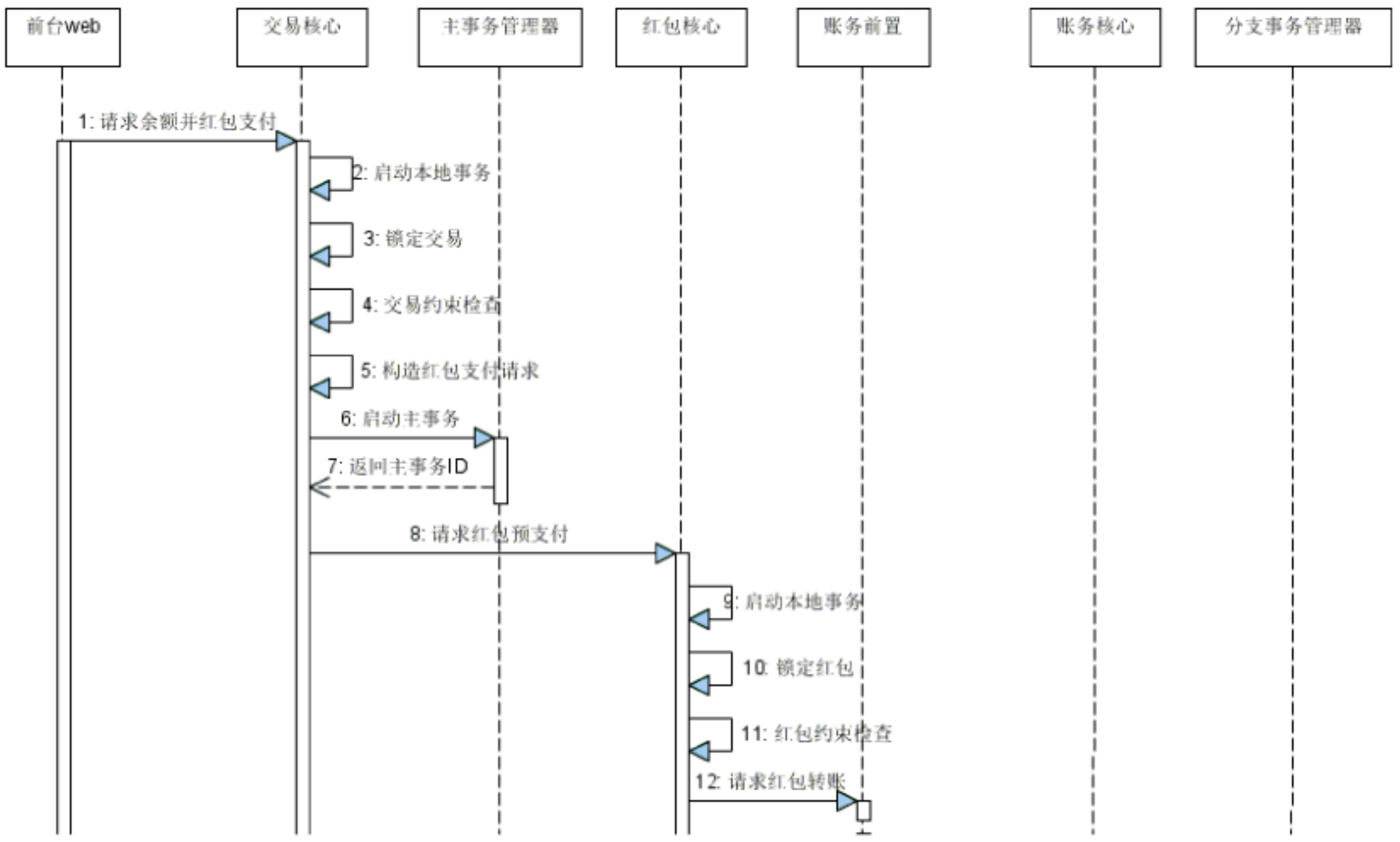
(1) 前台用红包 ID 请求交易核心做支付，交易核心进行交易支付约束检查；

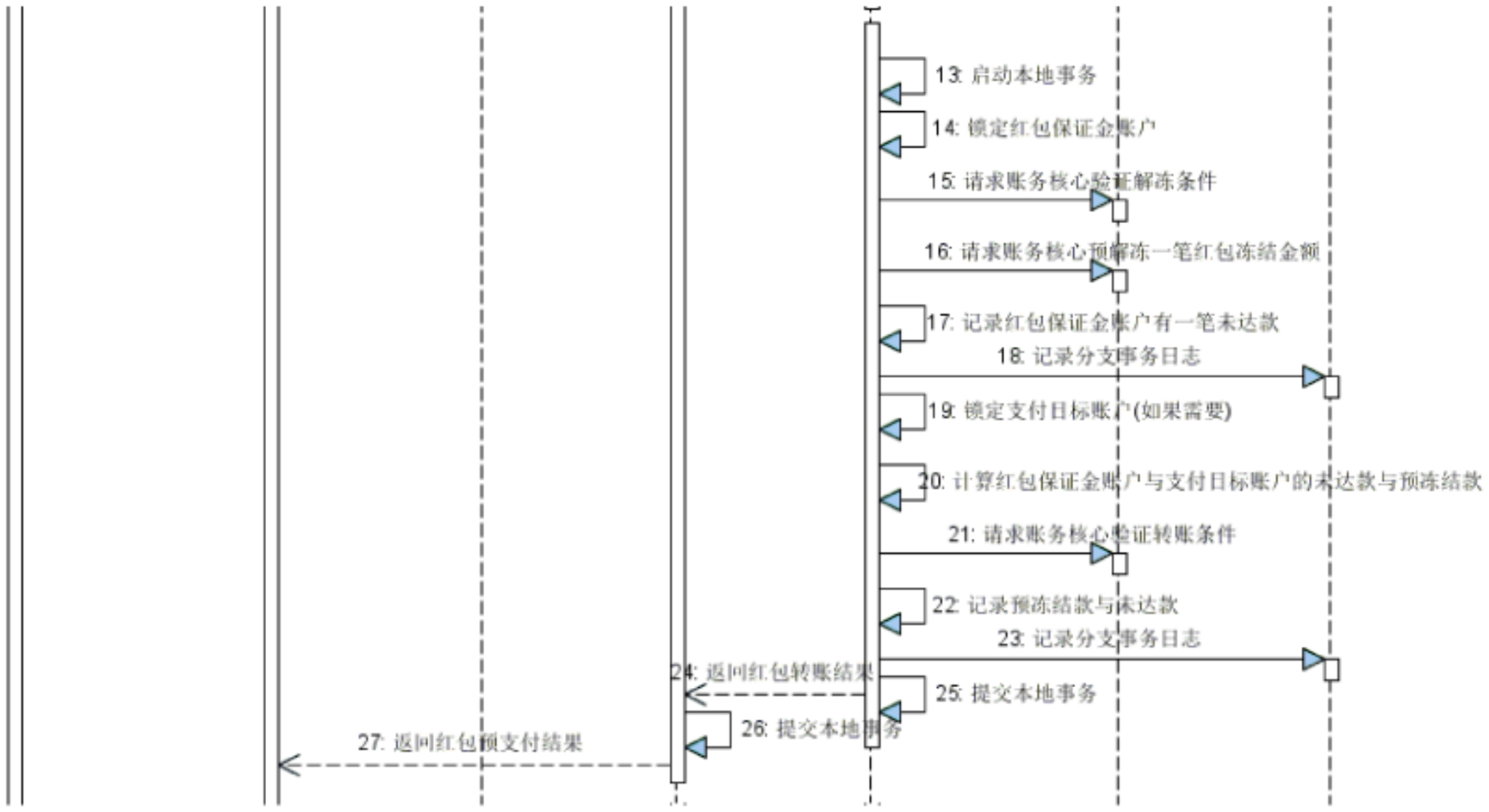
(2) 交易核心请求红包核心进行红包清算，将红包款项从保证金账户转移到支付目标账户。

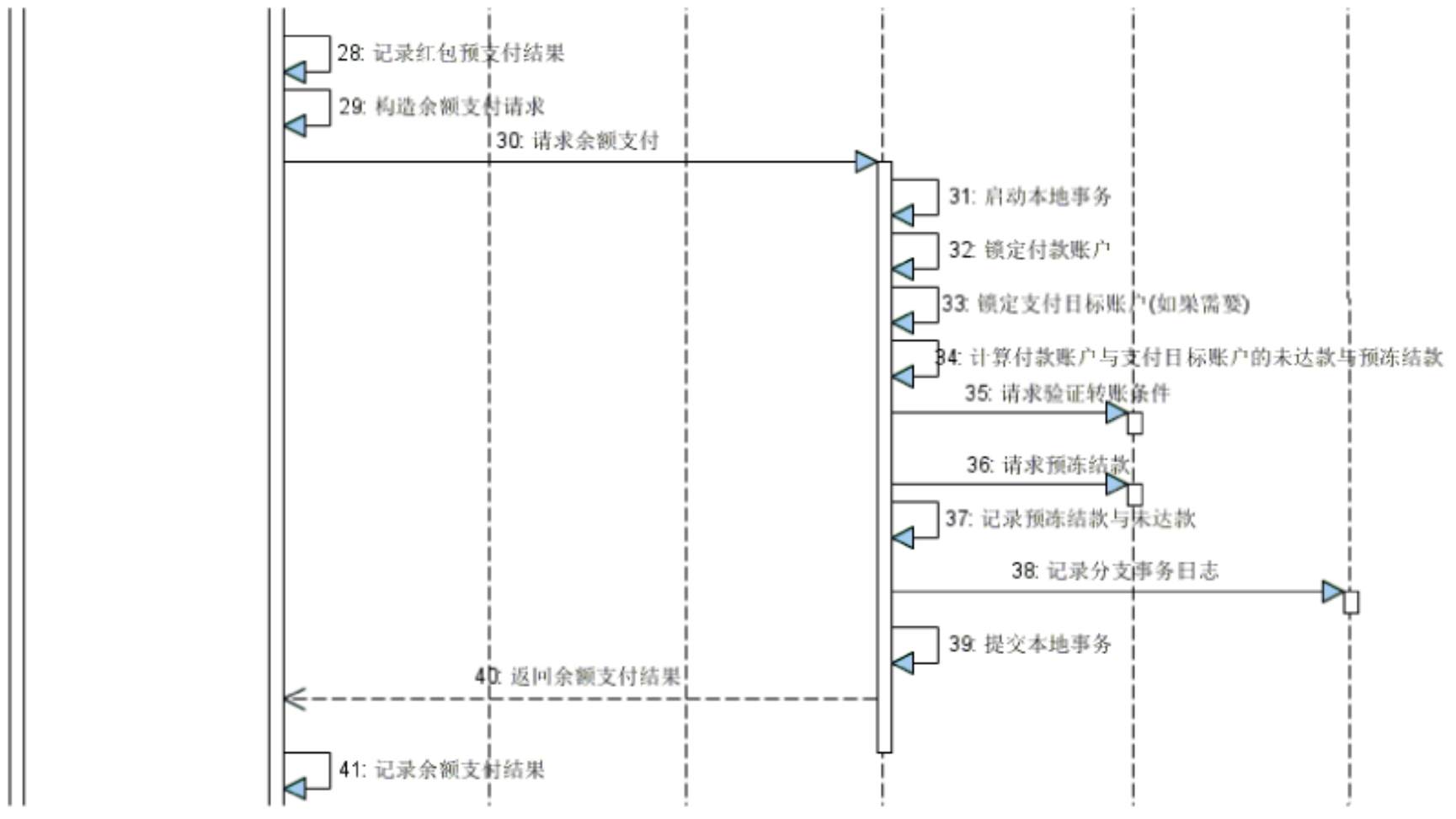
(3) 交易核心请求账务进行余额支付，将余额支付款项从买家账户转移到支付目标账户。

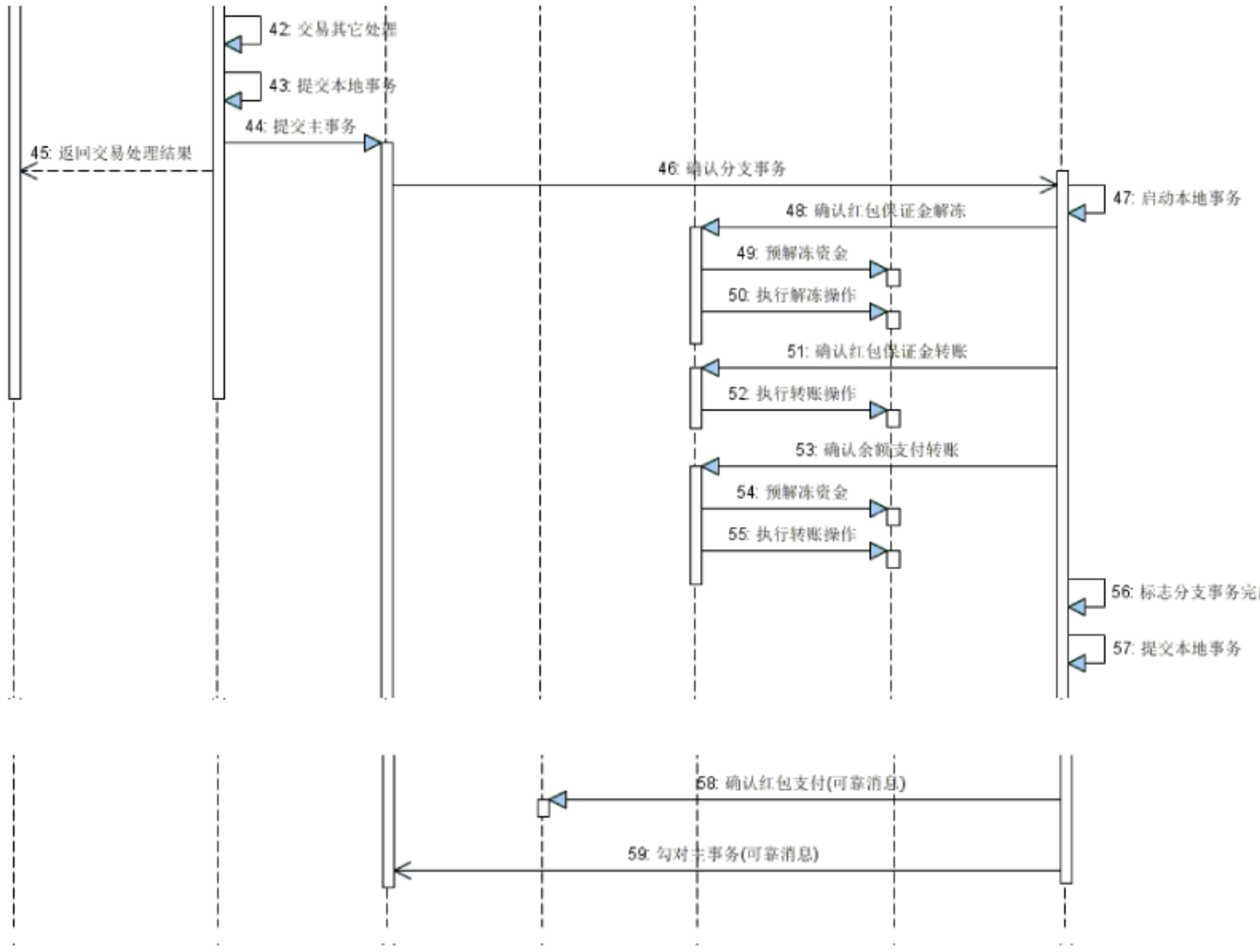
(4) 交易核心完成交易自身的处理，并向前台返回结果。

为了保证交易处理、余额支付、红包清算三者满足一致性约束，基于上述多参与者模型，整个处理的过程如下图所示：




13: 启动本地事务
14: 锁定红包保证金账户
15: 请求账务核心验证解冻条件
16: 请求账务核心预解冻一笔红包冻结金额
17: 记录红包保证金账户有一笔未达款
18: 记录分支事务日志
19: 锁定支付目标账户(如果需要)
20: 计算红包保证金账户与支付目标账户的未达款与预冻结款
21: 请求账务核心验证转账条件
22: 记录预冻结款与未达款
23: 记录分支事务日志
24: 返回红包转账结果
25: 提交本地事务
26: 提交本地事务
27: 返回红包预支付结果


28: 记录红包预支付结果
29: 构造余额支付请求
30: 请求余额支付
31: 启动本地事务
32: 锁定付款账户
33: 锁定支付目标账户(如果需要)
34: 计算付款账户与支付目标账户的未达款与预冻结款
35: 请求验证转账条件
36: 请求预冻结款
37: 记录预冻结款与未达款
38: 记录分支事务日志
39: 提交本地事务
40: 返回余额支付结果
41: 记录余额支付结果


42: 交易其它处理
43: 提交本地事务
44: 提交主事务
45: 返回交易处理结果
46: 确认分支事务
47: 启动本地事务
48: 确认红包保证金解冻
49: 预解冻资金
50: 执行解冻操作
51: 确认红包保证金转账
52: 执行转账操作
53: 确认余额支付转账
54: 预解冻资金
55: 执行转账操作
56: 标志分支事务完成
57: 提交本地事务
58: 确认红包支付(可靠消息)
59: 勾对主事务(可靠消息)