## 分布式柔性事务

## 分布式柔性事务

倘若限界上下文之间采用跨进程通信，且遵循零共享架构，各个限界上下文访问自己专有的数据库，这时的架构就演变为微服务风格。微服务架构不能绕开的一个问题，就是如何处理分布式事务。如果微服务访问的资源支持 X/A 规范，可以采用诸如二阶段提交协议等分布式事务来保证数据的强一致性。当一个系统的并发访问量越来越大，分区的节点越来越多时，用这样一种分布式事务去维护数据的强一致性，成本是非常昂贵的。

作为典型的分布式系统，微服务架构受到CAP平衡理论的制约。所谓的 CAP 就是 **C** onsistency（一致性）、 **A** vailablity（可用性）和 **P** artition-tolerance（分区容错性）的缩写：

* 一致性：要求所有节点每次读操作都能保证获取到最新数据；
* 可用性：要求无论任何故障产生后都能保证服务仍然可用；
* 分区容错性：要求被分区的节点可以正常对外提供服务。

CAP 平衡理论是 Eric Brewer 教授在 2000 年提出的猜想，即“一致性、可用性和分区容错性三者无法在分布式系统中被同时满足，并且最多只能满足其中两个！”这一猜想在 2002 年得到 Lynch 等人的证明。由于分布式系统必然需要保证分区容忍性，在这一前提下，就只能在可用性与一致性二者之间进行取舍。如果要追求数据的强一致性，就只有牺牲系统的可用性，保障强一致性的手段就是遵循XA协议的方案包括二阶段提交协议，以及基于它进行改进的三阶段提交协议。

许多业务场景对数据一致性要求并非不能妥协。这个时候BASE理论就体现了另一种平衡思想的价值。BASE 是 **B** asically **A** vailable（基本可用）、 **S** oft-state（软状态）和 **E** ventually Consistent（最终一致性）的缩写，它是最终一致性的理论支撑。BASE 理论放松了对数据一致性的要求，允许在一段时间内牺牲数据的一致性来换取分布式系统的基本可用，只要最终数据能够达到一致状态。

如果将满足数据强一致性即 ACID 要求的分布式事务称之为 **刚性事务** ，则满足数据最终一致性的分布式事务则称之为 **柔性事务** 。业界常用的柔性事务模式包括：

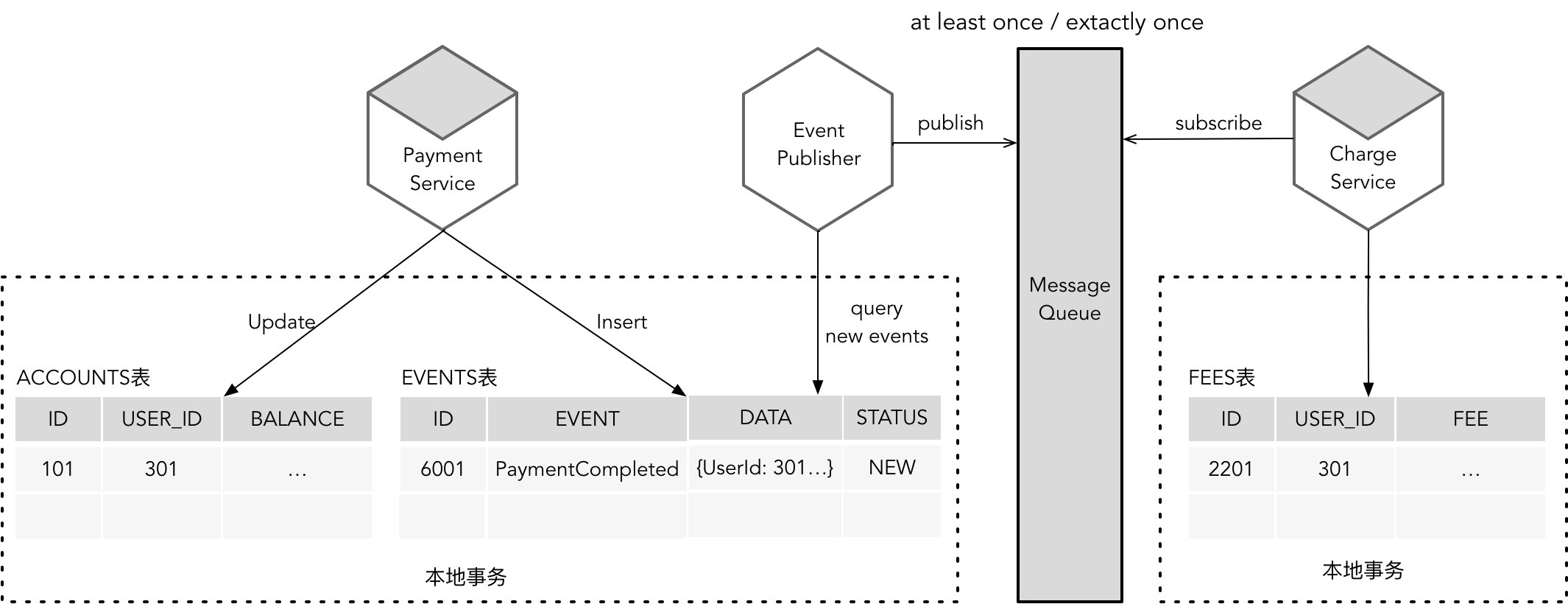
* 可靠事件模式
* TCC 模式
* SAGA 模式

接下来，我将探讨这些模式对领域驱动设计带来的影响。为了便于说明，我为这些模式选择了一个共同的业务场景：手机用户在营业厅通过信用卡为话费充值。该业务场景牵涉到交易服务、支付服务与充值服务之间的跨进程通信，这三个服务是完全独立的微服务，且无法采用 X/A 分布式事务来满足数据一致性的需求。

### 可靠事件模式

可靠事件模式的一种实现结合了本地事务与可靠消息传递的特性。在本地事务中，当前限界上下文的业务表与事件消息表处于同一个数据库，如此就可以保证业务数据的更改与事件消息的插入能够保证强一致性。当事件消息成功插入到事件消息表后，再利用事件发布者（Event Publisher）轮询该事件消息表，向消息队列（消息代理）发布事件。这时候，就需要利用消息队列传递消息的“至少一次（at least once）”特性，保证该事件消息无论如何都要传递到消息队列中，并被消息的订阅者成功订阅。只要保证事件处理器对该事件的处理是幂等的，就能保证执行操作的可靠性，最终达成数据的一致。

以话费充值业务场景为例，由交易服务发起支付操作，调用支付服务。支付服务在更新了 ACCOUNTS 表的账户余额同时，还要将 PaymentCompleted 事件追加到属于同一个数据库的 EVENTS 表中。EventPublisher 会定时轮询 EVENTS 表，获得新追加的事件后将其发布给消息队列。充值服务订阅 PaymentCompleted 事件。一旦收到该事件之后，就会执行充值服务的领域逻辑，更新 FEES 表。这个执行过程如下图所示：



33308538.png

充值服务在成功完成充值后，在更新话费的同时，还要将 PhoneBillCharged 事件追加到充值数据库的 EVENTS 表中，然后由 EventPublisher 轮询 EVENTS 表并发布事件。交易服务会订阅 PhoneBillCharged 事件，然后添加一条新的交易记录在 TRANSACTIONS 数据表。这个流程同样采用可靠事件模式。

在实现可靠事件模式时，领域事件是领域模型中不可缺少的一部分。领域事件的持久化与聚合的持久化发生在一个本地事务范围内，为了保证它们的事务一致性，可以将该领域事件放在聚合边界内部，同时为聚合以及对应的资源库增加操作领域事件的功能。ThoughtWorks 的滕云在文章《[后端开发实践系列——事件驱动架构(EDA)编码实践](https://insights.thoughtworks.cn/back-end-%20development-practice-series-3/)》中给出了支持领域事件的聚合抽象类与资源库抽象类的范例。例如，能够感知领域事件的聚合抽象类：

public abstract class DomainEventAwareAggregate {  
 @JsonIgnore  
 private final List<DomainEvent> events = newArrayList();  
  
 protected void raiseEvent(DomainEvent event) {  
 this.events.add(event);  
 }  
  
 void clearEvents() {  
 this.events.clear();  
 }  
  
 List<DomainEvent> getEvents() {  
 return Collections.unmodifiableList(events);  
 }  
}

能够在持久化聚合的同时支持对领域事件持久化的资源库抽象类：

public abstract class DomainEventAwareRepository<AR extends DomainEventAwareAggregate> {  
 @Autowired  
 private DomainEventDao eventDao;  
  
 public void save(AR aggregate) {  
 eventDao.insert(aggregate.getEvents());  
 aggregate.clearEvents();  
 doSave(aggregate);  
 }  
  
 protected abstract void doSave(AR aggregate);  
}

事件消息的发布可以由应用服务协调面向消息队列的南向网关抽象进行发布。消息发布成功之后，还要更新或删除事件表的记录，避免事件消息的重复发布。事件消息的订阅者则作为订阅上下文的远程服务，通过调用消息队列的 SDK 实现对消息队列的侦听。在处理事件时，需要保证处理逻辑的幂等性，这样就可以规避因为事件重复发送对业务逻辑的影响。

整体看来，倘若采用可靠事件模式的柔性事务机制，对领域驱动设计的影响，就是 **增加对事件消息的处理，包括事件的创建、存储、发布、订阅以及事件的删除** 。显然，可靠事件模式改变了限界上下文之间的协作方式，不再采用客户方/供应方的上下游关系，而是选择了发布/订阅事件模式。只是在发布/订阅事件模式的基础上，增加了本地事件表来保证了事件消息的可靠性。

### TCC 模式

要保证数据的最终一致性，就必须考虑在失败情况下如何让不一致的数据状态恢复到一致状态。一种有效办法就是采用补偿机制。补偿与回滚不同，回滚在强一致的事务下，资源修改的操作并没有真正提交到事务资源上，变更的数据仅仅限于内存，一旦发生异常，执行回滚操作就是撤销内存中还未提交的操作。补偿则是对已经发生的事实做事后补救，由于数据已经发生了真正的变更，因而无法对数据进行撤销，而是执行相应的逆操作。例如插入了订单，逆操作就是删除订单；反过来也当如是。因此，采用补偿机制实现数据的最终一致性，就需要为每个修改状态的操作定义对应的逆操作。

采用补偿机制的柔性事务模式被称之为“补偿模式”或“业务补偿”模式。该模式与可靠事件模式最大的不同在于，可靠事件模式的上游服务不依赖于下游服务的运行结果，上游服务一旦执行成功，可靠事件模式就通过可靠的消息传递机制要求下游服务无论如何也要执行成功；而补偿模式的上游服务则会强依赖于下游服务，它需要根据下游服务执行的结果判断是否需要进行补偿。

在话费充值的业务场景中，支付服务为上游服务，充值服务为下游服务。在支付服务执行成功之后，如果充值服务操作失败，就会导致数据不一致；这时，补偿机制就会要求上游服务执行事先定义好的逆操作，将银行账户中已经扣掉的金额重新退回。

如果进行补偿的逆操作迟迟没有执行，就会导致软状态的时间过长，服务长期处于不一致状态。为了解决这一问题，就引入了一种特殊的补偿模式：TCC 模式。TCC 模式根据业务角色的不同，将参与整个完整业务用例的分布式应用分为主业务服务和从业务服务。主业务服务负责发起流程，从业务服务执行具体的业务，业务行为被拆分为三个操作：Try、Confirm、Cancel。这三个操作分属于二阶段提交的准备和提交阶段，提交阶段又分为 Confirm 阶段和 Cancel 阶段：

* Try 阶段：对各个业务服务的资源做检测以及对资源进行锁定或者预留。
* Confirm 阶段：各个业务服务中执行的实际操作。
* Cancel 阶段：如果任何一个业务服务的方法执行出错，就需要进行补偿，即释放Try阶段预留的资源。

与普通的补偿模式不同，Cancel 阶段的操作更像是回滚，但实际并非如此。Try 阶段对资源的锁定与预留，并不像本地事务那样以同步的方式对资源加锁，而是采用真正执行资源更改的操作，但它在业务接口的设计上需要对资源进行锁定。正如阿里的觉生在文章《[分布式事务 Seata TCC 模式深度解析](https://www.infoq.cn/article/G33hCC-QoSJPlkt4e64E)》中所写：

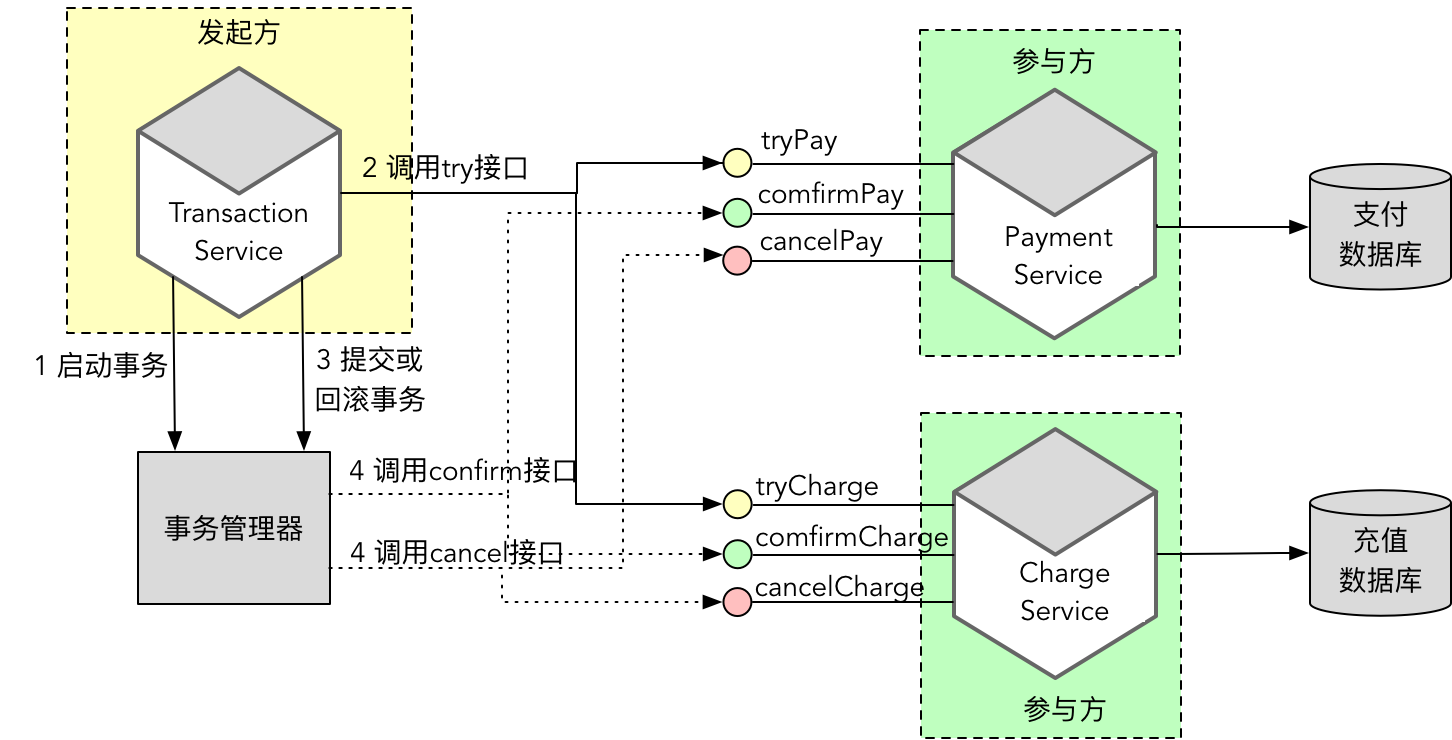
TCC 模型的隔离性思想就是通过业务的改造，在第一阶段结束之后，从底层数据库资源层面的加锁过渡为上层业务层面的加锁，从而释放底层数据库锁资源，放宽分布式事务锁协议，将锁的粒度降到最低，以最大限度提高业务并发性能。

什么意思呢？那就是 Try 阶段对资源的锁定既不是本地事务的加锁机制，也非直接粗暴的资源更改，而是改造参与 TCC 事务的资源模型，使其能够支持业务上的加锁。以支付服务为例，如果不采用 TCC 模式，则支付服务的 Account 模型只需要定义 balance 字段即可。当扣款发生时，首先检查账户余额，如果余额充足，就直接扣除账户余额。若采用 TCC 模式，Try 阶段通常不会直接扣除账户余额，而是采用冻结金额的方式，增加一个 frozenBalance 字段。当扣款发生时，首先检查账户余额，如果余额充足，就会在减少余额的同时，增加冻结金额。注意，资源的锁定或预留仅限于需要扣减的资源而言，例如充值服务是为账户增加话费余额，就无需调整话费资源的模型。

Try 阶段锁定了资源后，进入 Confirm 阶段执行业务的实际操作，由于 Try 阶段已经做了业务检查，Confirm 阶段的操作只需要直接使用 Try 阶段锁定的业务资源。如果Try阶段采用冻结金额来锁定资源，在 Confirm 阶段只需扣掉之前冻结的金额即可。如果 Try 阶段的操作执行失败，就进入到 Cancel 阶段，这个阶段的操作需要释放 Try 阶段锁定的资源，即扣掉之前冻结的金额，并增加（注意：是增加而非撤销，这正是回滚与补偿的区别）账户的可用余额。

显然，为了应对 TCC 模式的这三个阶段，必须为支付服务的支付操作定义对应的操作，即对应的 tryPay()、confirmPay() 与 cancelPay() 方法。同理，既然 TCC 模式用于协调多个分布式服务，则参与该模式的所有服务都需要遵循该模式划分的三个阶段。故而在话费充值业务场景中，充值服务也需要定义这三个操作：tryCharge()、confirmCharge() 与 cancelCharge()。不同之处在于话费充值的 tryCharge() 操作无需锁定或预留资源。

TCC 模式将参与主体分为发起方（又称为主业务服务）与参与方（又称为从业务服务）。发起方负责协调事务管理器和多个参与方，由其在启动事务之后，调用所有参与方的 Try 方法。当所有 Try 方法均执行成功时，由事务管理器调用每个参与方的 Confirm 方法。倘若任何一个参与方的 Try 方法执行失败，事务管理器就会调用每个参与方的 Cancel 方法完成补偿。以话费充值场景为例，交易服务为发起方，支付服务与充值服务为参与方，它们采用 TCC 模式的执行流程如下图所示：



85720380.png

显然， **TCC 模式改变了参与方代表事务资源的领域模型，并对服务接口的定义做出了要求** 。倘若一个领域模型要作为 TCC 模式的事务资源，就需要定义相关属性支持对资源自身的锁定或预留。同时，每个作为参与方的业务服务接口都需要定义 Try、Confirm 与 Cancel 方法，在实现这些方法时，还需要保证这些方法具有幂等性。

为了尽量避免 TCC 模式对领域模型产生影响，关键之处在于遵循整洁架构思想，让领域模型不要依赖本属于基础设施的 TCC 实现机制或框架。因此，在领域驱动分层架构中，应由基础设施层中扮演北向网关的远程服务作为 TCC 模式发起方与参与方的服务。例如，如果使用 tcc- transaction 框架实现 Dubbo 服务的 TCC 模式，那么在基础设施层的提供者实现类中，Try 方法需要通过 @Compensable 标注来指定 Confirm 方法和 Cancel 方法，如支付服务：

package com.dddsample.paymentcontext.gateway.providers;  
  
public class DebitCardPaymentProvider implements PaymentProvider {  
 @Compensable(confirmMethod = "confirmPay", cancelMethod = "cancelPay", transactionContextEditor = DubboTransactionContextEditor.class)  
 public void tryPay(PaymentRequest request) {}   
  
 public void confirmPay(PaymentRequest request) {}  
  
 public void cancelPay(PaymentRequest request) {}   
}

tcc-transaction 框架对 Dubbo 的 Provider 接口有一个要求，就是对发起执行请求的 TRY 方法接口需要标记 @Compensable，这意味着位于应用层的应用服务需要依赖 tcc-transaction 框架：

package com.dddsample.paymentcontext.application.providers;  
  
public interface PaymentProvider {  
 @Compensable  
 public void tryPay(PaymentRequest request) {}   
  
 public void confirmPay(PaymentRequest request) {}  
  
 public void cancelPay(PaymentRequest request) {}   
}

当我们将 TCC 模式的实现尽量推到应用服务和远程服务之后，就能将它对领域模型的影响降到最低。TCC 模式需要实现的三个方法，在领域模型中仍然属于领域逻辑的一部分，即使不采用 TCC 模式，也需要扣除余额、增加余额。相较而言，为了更好地实现资源锁定，引入的 frozenBalance 属性对领域逻辑的影响反而更大。但如果将冻结余额视为领域逻辑的一部分，在支付操作进行扣款时，同时修改冻结余额的值，似乎也在清理之中，我们完全可以在领域建模阶段考虑余额冻结这一领域逻辑。

### Saga 模式

Saga 模式又叫做长时间运行事务（Long-running-transaction）, 由普林斯顿大学的 H.Garcia-Molina 等人提出，可以在没有两阶段提交的的情况下解决分布式系统中复杂的业务事务问题。本质上讲，它认为一个长事务应该被拆分为多个短事务进行事务之间的协调。但在微服务分布式通信这个前提下，通常一个完整的业务用例需要多个微服务协作才能完成。因此讨论的焦点并不在于这个事务的执行时间长短，而在于该业务场景是否牵涉到多个分布式服务。因此，在 Saga 模式的语境下，可以认为“一个 Saga 表示需要更新多个服务中数据的一个系统操作”。

Saga 模式亦是一种补偿操作，但本质上它与 TCC 模式不同。Saga 模式没有 TRY 阶段，不需要预留或冻结资源。 **参与 Saga 模式的微服务就服务自身来讲，都是一个本地服务，在执行该服务的方法时，会直接操作该服务内的事务资源。** 这样就可能导致一个本地服务操作成功，另一个本地服务操作失败，形成数据的不一致。这时，就需要以合理的方式执行补偿操作，即之前执行操作的逆操作。

根据微服务之间协作方式的不同，Saga模式也有两种不同的实现，Chris Richardson 的《微服务架构设计模式》将其分别定义为：

* 协同式（Choreography）：把 Saga 的决策和执行顺序逻辑分布在 Saga 的每个参与方中，它们通过交换事件的方式来进行沟通
* 编排式（Orchestrator）：把 Saga 的决策和执行顺序逻辑集中在一个 Saga 编排器类中。Saga 编排器发出命令式消息给各个 Saga 参与方，指示这些参与方服务完成具体操作（本地事务）

#### 协同式

协同式的 Saga 的关键核心是 **事务消息** 与事件的 **发布者—订阅者模式** 。

所谓“事务消息”就是满足消息发送与本地事务执行的原子性问题。以本地数据库更新与消息发送为例，如果首先执行数据库更新，待执行成功后再发送消息，只要消息发送成功，就能保证事务的一致。但是，一旦消息发送不成功（包括经过多次重试），又该如何确保更新操作的回滚？如果首先发送消息，然后再执行数据库更新，又会面临当数据库更新失败时该如何撤销业已发送消息的难题。事务消息解决的就是这类问题。

Chris Richardson 提出了事务消息的多个方案，如使用数据库表作为消息队列、事务日志拖尾（Transaction Log Tailing）模式等。我们也可以选择支持事务消息的消息队列中间件，如 RocketMQ。RocketMQ 将消息的发送分为了 2 个阶段：准备阶段和确认阶段，从而将本地数据库更新与消息发送分解成了三个步骤：

* 向 RocketMQ 发送准备（Prepared）消息
* 执行数据库更新
* 如果更新成功，确认之前发送的准备消息，否则取消

一旦满足事务消息的基本条件，协同式 Saga 模式中的各个参与服务之间的通信就通过事件来完成，并能确保每个参与服务在本地是满足事务一致性的。与可靠事件模式不同的是，对于产生副作用的业务操作，需要定义对应的补偿操作。在 Saga 模式中，事件的订阅顺序刚好对应正向流程与逆向流程，执行的操作也与之对应。

仍以话费充值为例。在正向流程中，支付服务执行的操作为 pay()，完成支付后发布 PaymentCompleted 事件；充值服务订阅该事件，并在接收到该事件后，执行 charge() 操作，成功充值后发布 PhoneBillCharged 事件。而在逆向流程中，充值失败将发布 PhoneBillChargeFailed 事件，该事件由支付服务订阅，一旦接收到该事件，就需要执行 rejectPay() 补偿操作。

为了减少 Saga 模式对领域模型的影响，参与 Saga 协同的服务应由应用服务来承担，故而正向流程的操作与逆向流程的补偿操作皆定义在应用服务中。由于领域模型需要支持各种业务场景，即使不考虑分布式事务，也当提供完整的领域行为。以支付服务为例，即使不考虑充值失败时执行的补偿操作，也需要在领域模型中提供支付与退款这两种领域行为，这两个行为实则就是互逆的。它们的定义并不受分布式事务的影响。

#### 编排式

编排式的 Saga 需要开发人员定义一个编排器类，用于编排一个Saga中多个参与服务执行的流程，故而可认为它是一个 Saga 工作流引擎，通过它来协调这些参与的微服务。如果整个业务流程正常结束，业务就成功完成，一旦这个过程的任何环节出现失败，Sagas 工作流引擎就会以相反的顺序调用补偿操作，重新进行业务回滚。

如果说协同式的 Saga 是一种自治的协作行为，那么编排式的 Saga 就是一种集权的控制行为。但这种控制行为并不像一个过程脚本那样由编排器类按照顺序依次调用各个参与服务，而是将编排器类作为服务协作的调停者（Mediator），且仍然采用消息传递来实现服务之间的跨进程通信。它传递的消息是一个请求/响应消息。请求消息的发起者皆为 Saga 编排器，并将消息发送给消息队列（消息代理）。消息队列为响应消息创建一个专门的通道，作为 Saga 的回复通道。当对应的参与服务接收到该请求消息后，会将执行结果作为响应消息（本质上还是事件）返回给回复通道。Saga 编排器在接收到响应消息后，根据结果的成败决定调用正向操作还是逆向的补偿操作。

编排式方式减轻了各个参与服务的压力，因为参与服务不再需要订阅上游服务返回的事件，减少了服务之间对事件协议的依赖。例如，编排式的支付服务就无需了解充值服务返回的 PhoneBillChargeFailed 事件。支付服务的补偿操作是由 Saga 编排器接收到作为响应消息的 PhoneBillChargeFailed 事件，然后再由编排器向支付服务发起退款的命令请求。在引入编排器后，每个参与服务的职责就变得更为单一，更加一致：

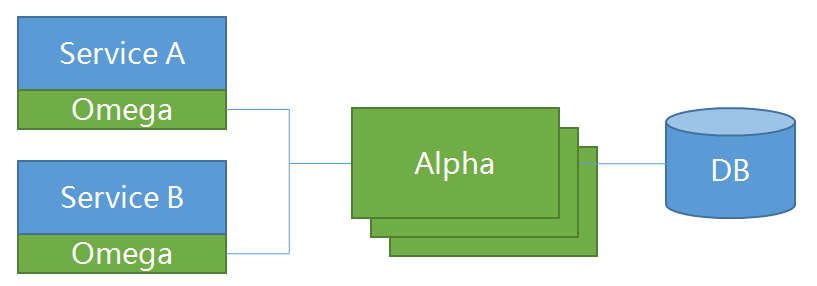
* 订阅并处理命令消息，该命令消息属于当前服务公开接口需要的输入参数
* 执行命令后返回响应消息

编排式与协同式的差异仅在于服务之间的协作方式，每个参与服务的接口定义却没有任何区别。只要隔离了应用层与领域模型，仍然能够保证领域模型的稳定性。

#### 使用 Saga 框架

若能使用已有的 Saga 框架，就无需开发人员再去处理繁琐的服务协作与补偿方法调用的技术实现细节。《微服务架构设计模式》的作者 Chris Richardson 提供了 Eventuate Tram Saga 框架，能够支持编排式的 Saga 模式。但是，它对 Saga 的封装并不够彻底，要使用该框架，仍然需要了解太多 Saga 的细节。ServiceComb Pack 属于微服务平台 [Apache ServiceComb](http://servicecomb.apache.org/cn/) 的一部分，它为分布式柔性事务提供了整体解决方案，能够同时支持 Saga 与 TCC 模式。

ServiceComb Pack 通过 gRPC 与 Kyro 序列化来实现微服务之间的分布式通信，它包含了两个组件：Alpha 与 Omega。Alpha 作为 Saga 协调者，负责对事务进行管理和协调。Omega 是内嵌到每个参与服务的引擎，可以通过它拦截调用请求并向 Alpha 上报事务事件。下图为官方文档给出的设计图：



ServiceComb Pack 架构

采用协调者与拦截引擎的设计机制可以有效地将 Saga 机制封装起来，开发人员在实现微服务之间的调用时，几乎感受不到 Saga 模式的事务处理。除了必要的配置与依赖之外，只需要在各个参与方应用服务的正向操作上指定补偿方法即可。例如支付服务的应用服务定义：

import org.apache.servicecomb.pack.omega.transaction.annotations.Compensable;  
import org.springframework.stereotype.Service;  
  
@Service  
public class PaymentAppService {  
 @Compensable(compensationMethod = "rejectPay")  
 public void pay(PaymentRequest paymentRequest) {}  
  
 void rejectPay(String paymentId) {}  
}

PaymentAppService 应用服务在接收到支付请求后，可以交由领域服务来完成支付功能；补偿操作 rejectPay() 方法同样可以交由领域服务完成取消支付的功能。由于在设计上我们遵循了整洁架构思想，领域模型对象并不知道应用服务，也没有依赖外部的 ServiceComb Pack 框架，保持了领域模型的纯粹性与稳定性。

下一章

还没有评论



评论

## 更多资源下载交流请加微信：Morstrong,加入永久会员,网盘更新更快捷！

# 本资源由微信公众号：光明顶一号，提供支持