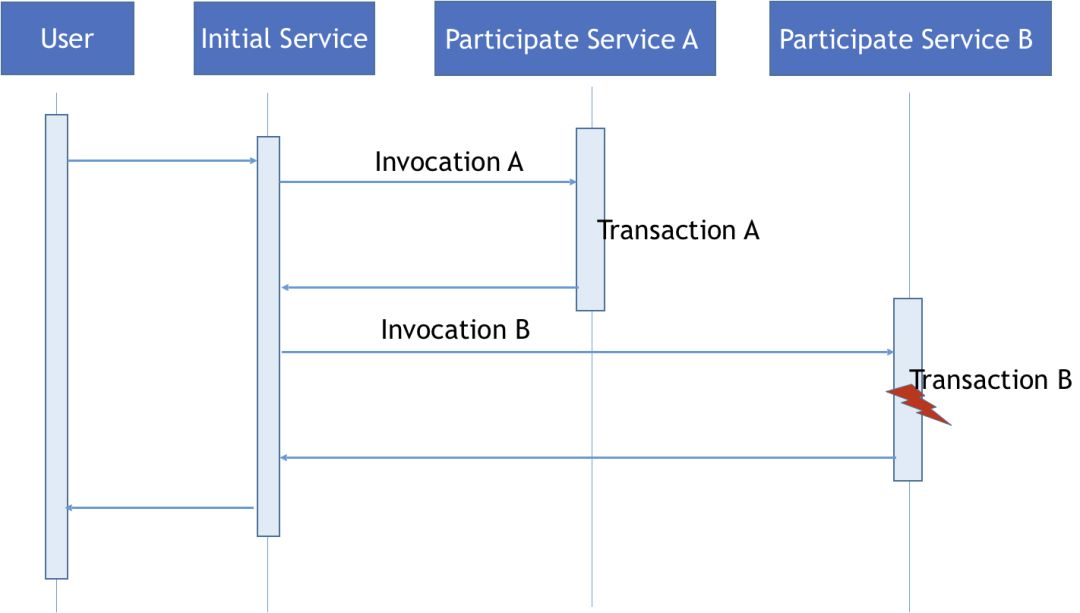
**基于服务的分布式事务（下）**

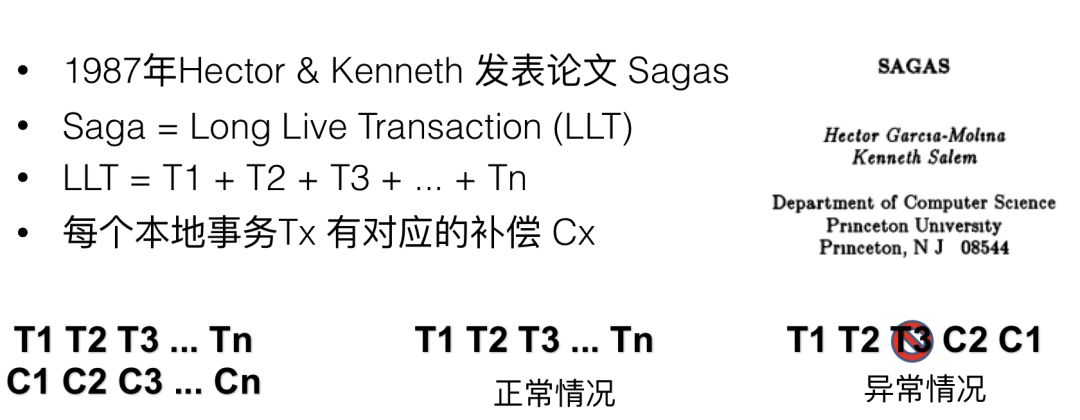
**分布式事务协调场景介绍**

在[基于服务的分布式事务（上）](http://mp.weixin.qq.com/s?__biz=MzUxNTEwNTg5Mg==&mid=2247485544&idx=1&sn=fee2e7292215e8985347f044719c7914&chksm=f9bafd63cecd7475b7198f4cdd5033144ca9a75c9ff44d8f14db68e0daca9cee4560ed4810a1&scene=21#wechat_redirect)中，我们举了一个业务场景的例子，就是一个初始服务创建了一个分布式事务，在这个分布式事务中包含了两个参与服务的本地事务，这两个本地事务由初始服务通过调用两个参与事务的服务方式组合在一起。根据分布式事务一致性的要求，这两个本地事务要么同时成功，要么同时失败。由于这两个参与事务的服务并不知道对方的存在，当一个参与服务调用（Invocation A）成功而另外一个参与服务调用（Invocation B）失败，我们就需要分布式事务协调器的进行相关的补偿，保证分布式事务的一致性。

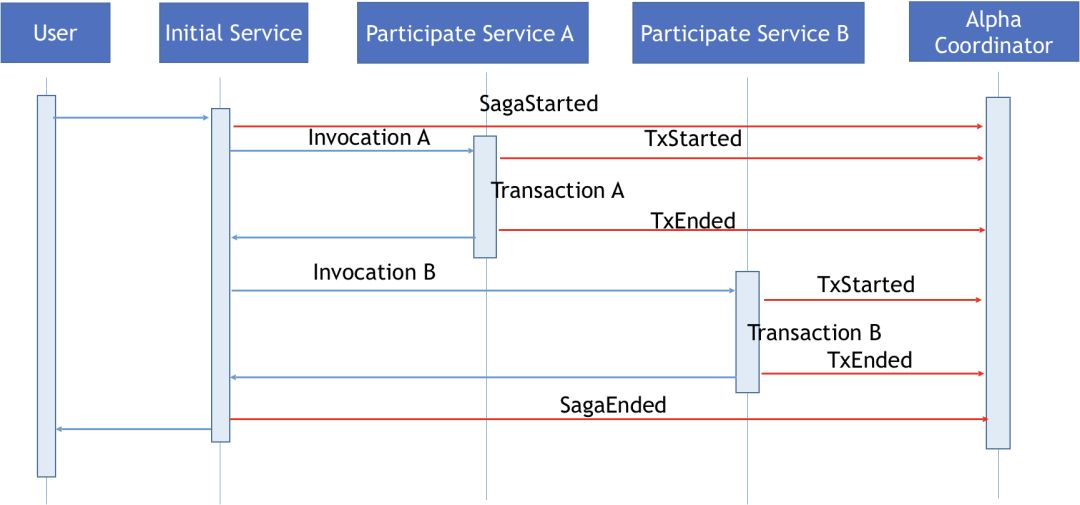


**分布式Saga**

ServiceComb Pack[1]默认采用的是名为Saga的分布式事务协调方案。Sagas[2]这个概念来源于三十多年前的一篇数据库论文，一个Saga事务是一个由多个短时事务组成的长时事务。在分布式事务场景下，我们把一个Saga分布式事务看做是一个由多个本地事务组成的事务，每个本地事务都有一个与之对应的补偿事务。在Saga事务的执行过程中，如果出现某一步执行出现异常的，Saga事务会被终止，同时会调用之前执行成功的事务对应的补偿事务完成相关的恢复操作，这样保证Saga相关的本地事务要么同时成功，要么通过执行补偿恢复成为Saga执行之前的状态。

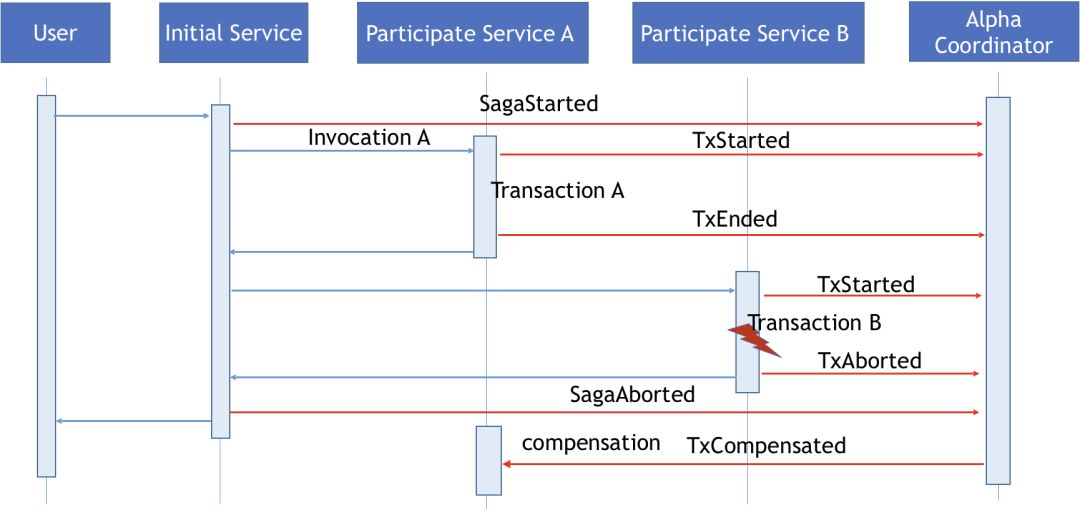


ServiceComb Pack 在实现分布式Saga协调协议的过程中需要追踪分布式事务的执行情况。首先介绍一下正常流程下分布式事务执行流程是如何记录下来的，下图的红线部分是Omega端扩展与Alpha端交互序列图。



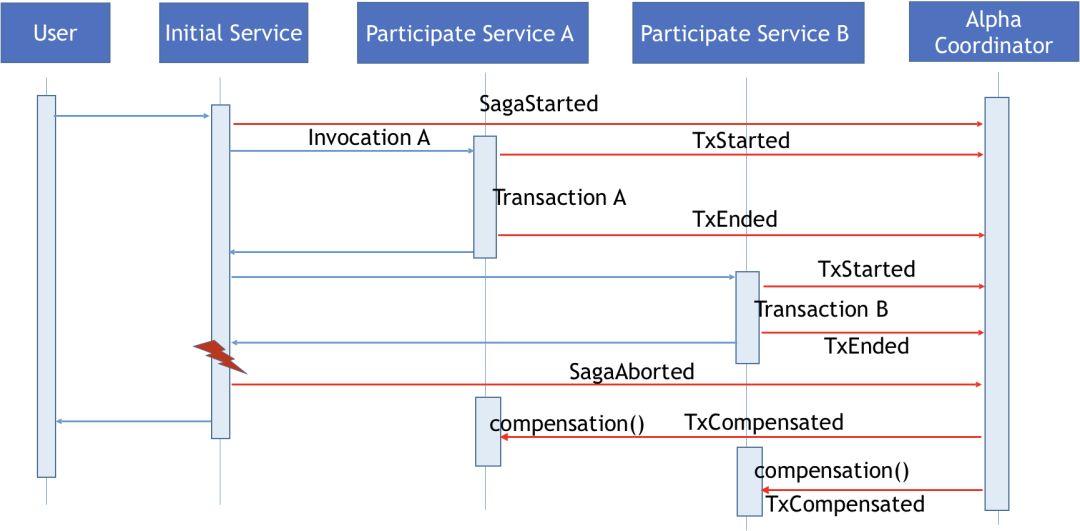
在分布式事务初始阶段由初始服务的Omega将**SagaStarted**事件发送到Alpha进行分布式事务备案。当有新的服务参与到这个分布式事务中，参与服务A的Omega会在本地事务执行前发送**TxStarted**到Alpha端；并在本地事务执行成功之后将**TxEnded**事件发送到Alpha。如果分布式事务正常结束，初始服务Omega会直接发送**SagaEnded**事件到Alpha，结束整个分布式事务。

参与分布式事务的服务在执行本地事务出现异常，如下图所示Transaction B执行出现错误。



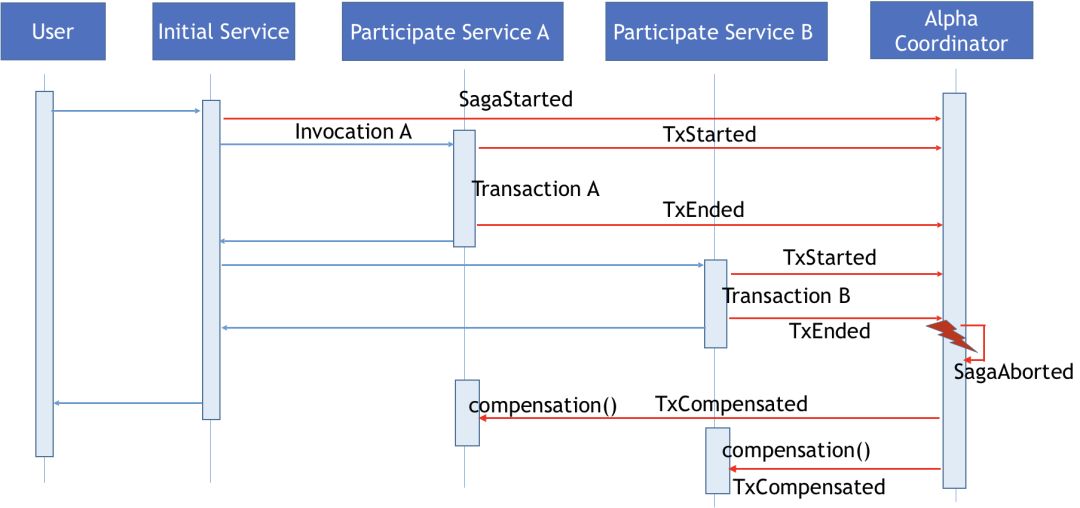
此时参与服务B会向Alpha发送一个**TxAborted**的事件，Alpha收到这个事件后会将整个Saga事务挂起，终止Saga事务的继续执行。如果这个时候还有其他的Omega向Alpha发送挂载在这个Saga事务下的**TxStarted**事件的话，Alpha会直接发送拒绝应答消息通知Omega这个Saga事务已经出现异常，不应该执行新的本地事务。由于初始服务在调用参与服务B的过程中，也知道了服务调用失败的消息，所以，初始服务也会发生**SagaAborted**事件给Alpha来关闭整个Saga事件。虽然在**TxAborted**存在的情况下，**SagaAborted**事件看上去有点多余，但是为了应对诸如初始服务无法调用参与服务B的情况下（此时Alpha没有收到**TxAborted**事件），设置**SagaAborted**事件还是非常有必要的。

现在Alpha可以通过查询**TxEnded**事件获取到需要进行补偿恢复的服务信息，Alpha会向相关的服务实例Omega发送**TxCompensated**事件，由Omega调用服务实例补偿方法进行相关的恢复操作。在具体的过程中，为了恢复本地事务执行上下文，ServiceComb Pack会将**TxStarted**传递过来的方法参数列表信息放入**TxCompensated**消息中传递给Omega，除此之外ServiceComb Pack 还会将OmegaContext[3]的全局事务和本地事务设置成之前的状态，这样应用代码可以在此基础上扩展获取自定义的上下文环境。

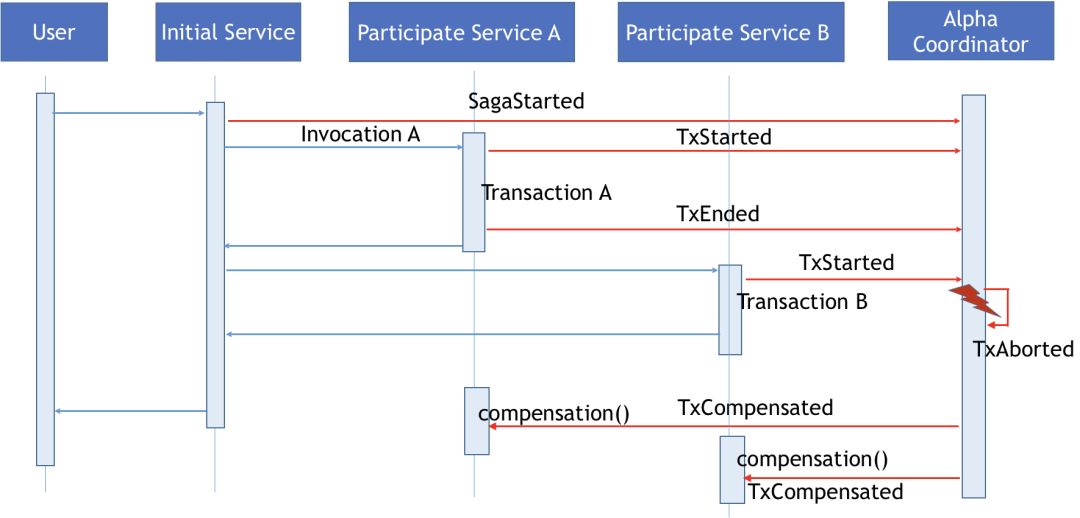


除了要考虑异常情况，我们还需要考虑事务执行超时的处理问题（为了简化场景，这里我们不考虑由于网络连接中断导致的事务异常或者结束消息丢包的情况）。目前我们可以在Saga以及本地事务之间设置超时时间，Alpha上的事件扫描器会定时查找Started事件在设定的超时时间内是否有对应的Aborted或者Ended事件，如果没有，则会生成对应的Aborted事件触发相关的补偿操作。

当整个Saga事务执行超时，Alpha事件扫描器会向Alpha发送**SagaAborted**事件终止整个Saga事务，并且调用恢复函数进行相关的恢复操作。



下图展示的就是在本地事务执行超时的情况下，Alpha事件扫描器会识别出Transaction B 执行超时，同时会发生**TxAborted**事件终止整个Saga 事务，调用相关的恢复方法进行恢复。由于Alpha无法确认对应的事务的执行情况，Alpha会采用向Omega发送**TxCompensated**的方式恢复事务，即使这时参与的服务事务已经执行成功了，Alpha还是会执行相关的恢复操作。



**编写Saga应用代码**

在文章开头我们给大家介绍了一个典型的分布式事务业务场景，其中涉及到一个初始服务，以及两个参与服务。为了方便大家理解，我们将以ServiceComb PackSpring Demo[4]为例介绍如何使用Saga实现分布式事务。

这里预订服务(Booking) 相当于之前提到的分布式事务初始服务，对外提供一个租车(Car)和酒店(Hotel)聚合服务，在BookingController[5]中使用Spring提供的RestTemplate[6]向租车和酒店服务转发请求。租车和酒店服务分别作为Saga事务参与方参与整个事务。预订服务、租车和酒店服务都是基于Spring-Boot[7]编写独立进程应用，应用代码通过@EnableOmega[8]加载Omega相关的配置[9]，同时需要在Spring的配置文件中配置与Alpha服务相关的信息。

import org.apache.servicecomb.pack.omega.spring.EnableOmega**;**import org.springframework.boot.SpringApplication**;**import org.springframework.boot.autoconfigure.SpringBootApplication**;**  
@SpringBootApplication  
@EnableOmega  
**public** **class** **Application** **{**

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

   SpringApplication**.**run**(**Application**.**class**,** args**);**

**}  
}**

在应用代码中需要描述出Saga事务的边界，我们可以在BookingController的order方法上标准@SagaStart；

import org.apache.servicecomb.pack.omega.context.annotations.SagaStart**;**@SagaStart  
@PostMapping**(**"/booking/{name}/{rooms}/{cars}"**)  
public** String **order(**@PathVariable String name**,** @PathVariable Integer rooms**,** @PathVariable Integer cars**)** **{**

*// Calling the car service*

 template**.**postForEntity**(**

     carServiceUrl **+** "/order/{name}/{cars}"**,**

**null,** String**.**class**,** name**,** cars**);**

*// Calling the hotel service*

 template**.**postForEntity**(**

     hotelServiceUrl **+** "/order/{name}/{rooms}"**,**

**null,** String**.**class**,** name**,** rooms**);**

**return** name **+** " booking " **+** rooms **+** " rooms and " **+** cars **+** " cars OK"**;  
}**

本地事务是通过@Compensable来标识，并且在Compensable的compensationMethod属性中描述补偿方法。注意补偿方法和本地事务方法的参数必须一致，否则Omega在进行恢复操作的时候会报找不到方法的错误。

import org.apache.servicecomb.pack.omega.transaction.annotations.Compensable**;**@Compensable**(**compensationMethod **=** "cancel"**)  
void** **order(**CarBooking booking**)** **{**

 booking**.**confirm**();**

 bookings**.**put**(**booking**.**getId**(),** booking**);  
}  
  
void** **cancel(**CarBooking booking**)** **{**

 Integer id **=** booking**.**getId**();**

**if** **(**bookings**.**containsKey**(**id**))** **{**

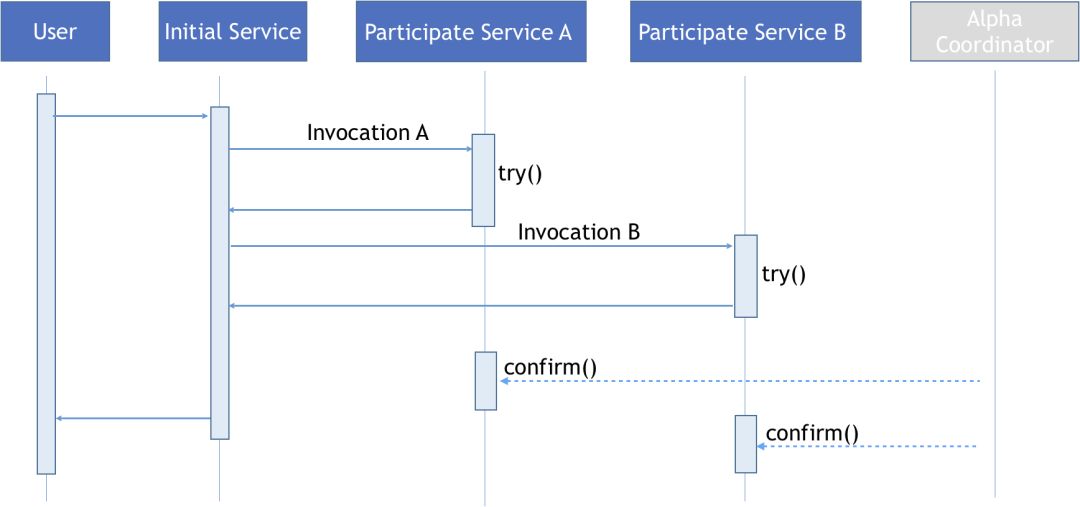
   bookings**.**get**(**id**).**cancel**();**

**}  
}**

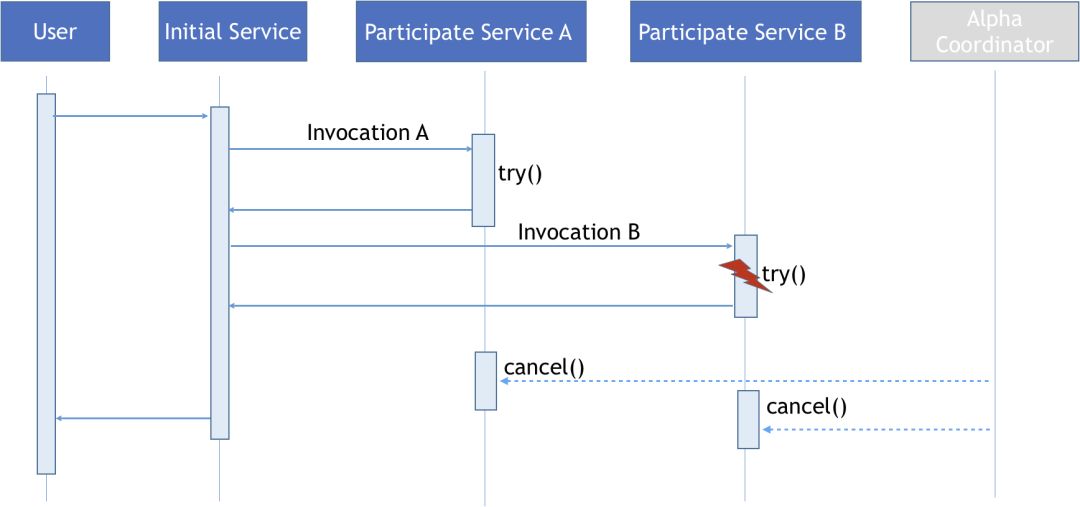
**TCC实现**

ServiceComb Pack 还提供了一个名为TCC（Try-Cancel/Confirm实现）分布式事务协调实现。TCC借助两阶段提交协议提供了一种比较完美的恢复方式。在TCC方式下，cancel补偿显然是在第二阶段需要执行业务逻辑来取消第一阶段产生的后果。try是在第一阶段执行相关的业务操作，完成相关业务资源的占用，例如预先分配票务资源，或者检查并刷新用户账户信用额度。在取消阶段释放相关的业务资源，例如释放预先分配的票务资源或者恢复之前占用的用户信用额度。那我们为什么还要加入确认操作呢？这需要从业务资源的使用生命周期来入手。在try过程中，我们只是占用的业务资源，相关的执行操作还处于待定状态，只有在确认操作执行完毕之后，业务资源才能真正被确认。

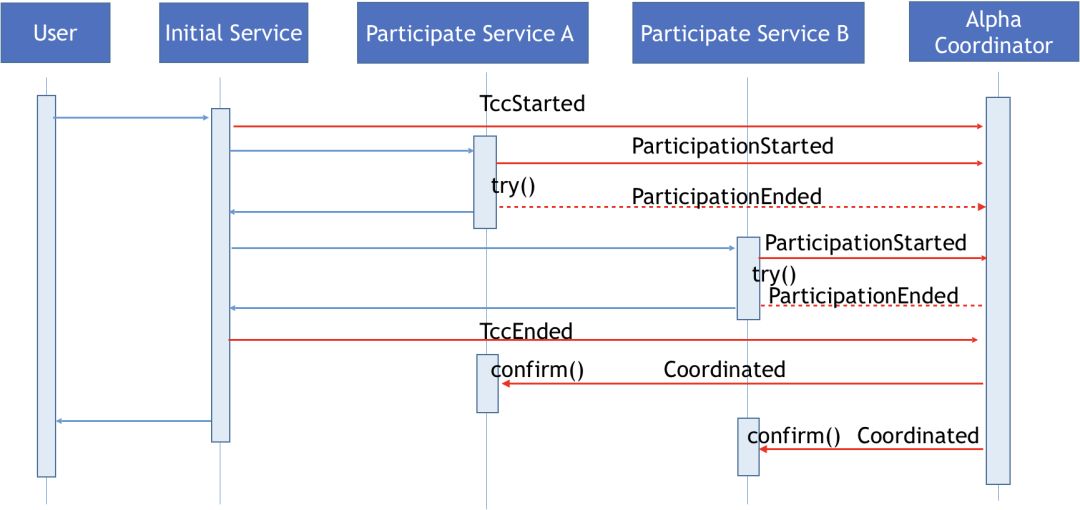
下图展示了正常的TCC调用流程，可以看到参与服务A，B分别在尝试方法中完成相关业务资源的预先分配，然后在提交阶段完成业务资源的确认操作。在实现层面和前面提到的Saga实现一样，我们需要协调器在分布式事务执行完成时向各个参与服务发送执行确认消息，由服务实例执行确认操作。



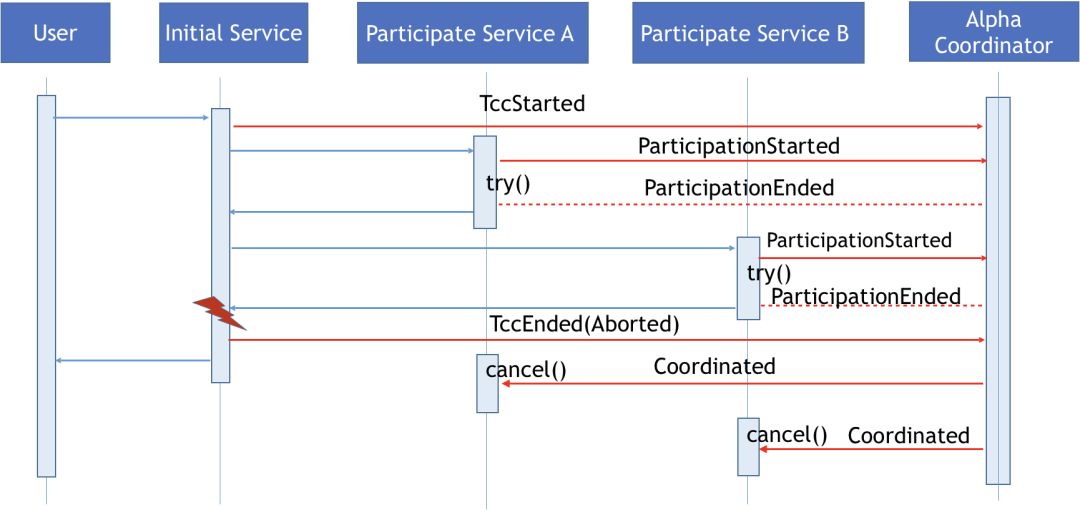
如果参与服务自身的本地事务执行出现错误了，事务协调器会终止整个分布式事务，同时事务协调器会向注册成功的参与服务发送取消消息，由服务实例执行取消操作。假如这时还有其他的服务想参与到这个分布式事务的话，事务协调器会以这个分布式事务已经失败为由，向参与的服务发送失败的应答消息。



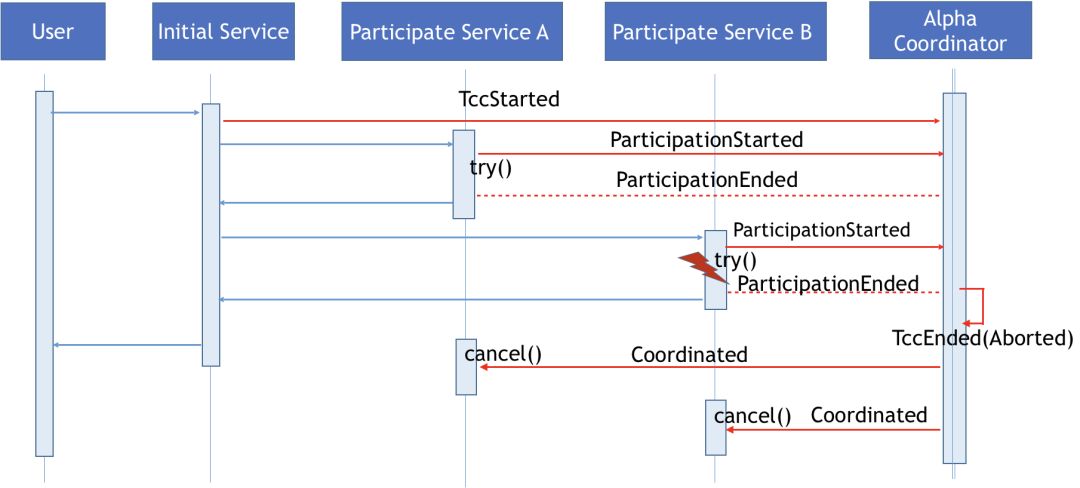
在ServiceComb Pack中，为了实现上面描述的TCC业务诉求，初始服务需要在分布式事务开始时向Alpha协调器发送**TccStarted**事件，Alpha协调器在接收到**TccStarted**事件之后，会创建相关事务追踪资源跟踪这个TCC事务的整个生命周期。初始服务会在参与服务调用try方法之前发送**ParticipationStarted**事件来声明与TCC相关本地事务。Alpha协调器会根据TCC事务当前的状态决定是否允许参与服务参加到TCC事务中。如果参与的TCC事务没有终止，Alpha协调器会回复确认消息，参与服务会继续执行相关的try方法调用；如果TCC事务已经终止了，Alpha协调器会回复终止消息，参与服务所在的Omega将抛出异常，直接终止try方法调用。如果参与服务调用try方法成功，则会向Alpha发送**ParticipationEnded**事件，因为这个事件发送之后Omega端不需要做任何操作，为了提高系统效率，Omega可采用异步方式通知Alpha协调器。当初始服务执行完标注好的TCC调用之后，初始服务所在的Omega会向Alpha协调器发送**TccEnded**事件，Alpha协调器在接收到这个事件之后会查询与本次TCC调用相关的**ParticipationStarted**事件识别相关的参与服务实例，然后通过向这些服务实例所对应的Omega发送**Coordinated**事件，由Omega调用相关的确认方法，完成本地事务提交工作。



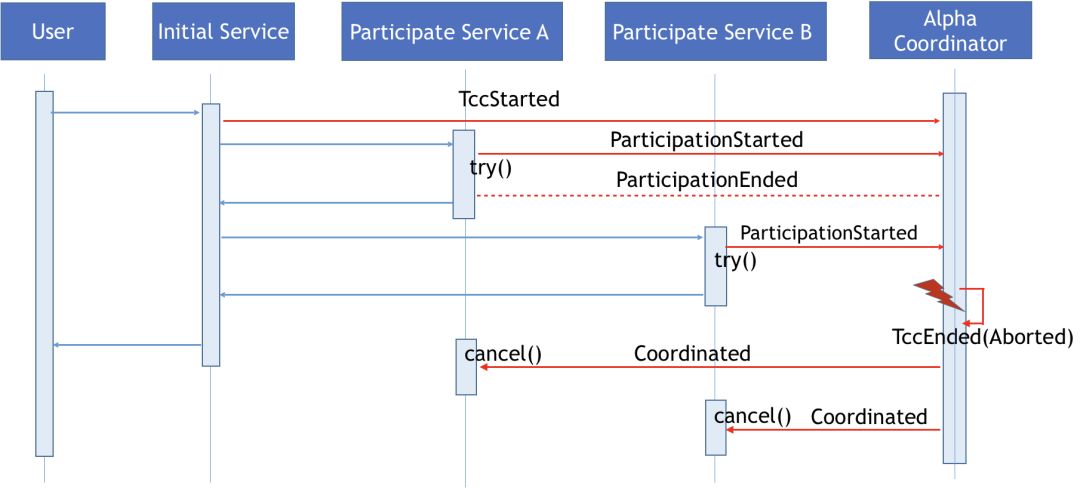
当TCC调用执行过程中出现异常，调用服务所在的Omega会向Alpha协调器发送**TccEnded**事件来终止当前的TCC事务。Alpha协调器则会根据其记录的分布式事务的参与情况，向相关服务的Omega发送**Coordinated**事件，由Omega调用相关的取消方法。



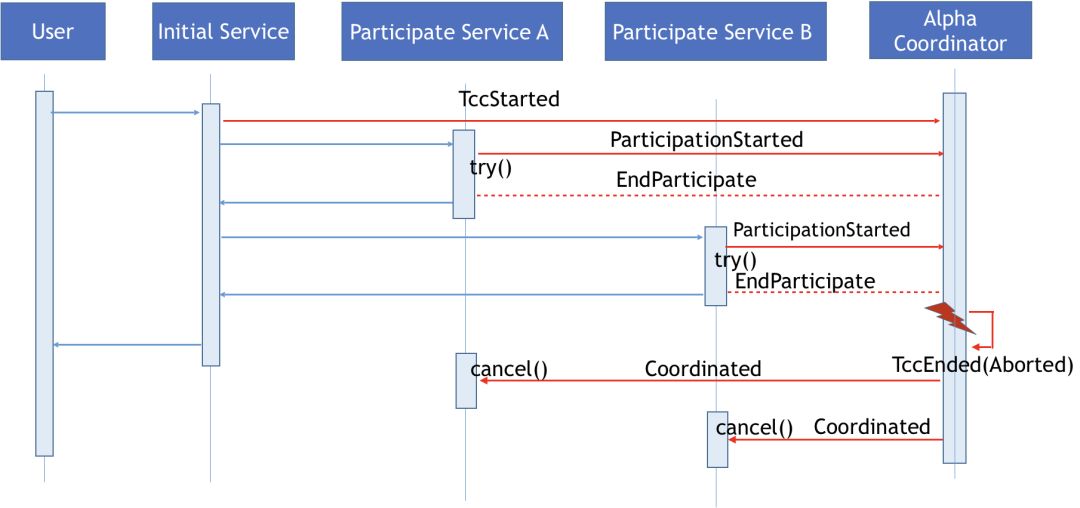
如果是参与服务在执行try方法时出错了，那Alpha协调器会收到一条标注try方法执行状态的**ParticipationEnded**事件，Alpha协调器会给自己发送一个包含Aborted信息的**TccEnded**事件来关闭正在执行的TCC事务，同时触发Omega相关恢复操作的调用。



下面介绍事务执行超时的处理的设计，对于参与服务的try方法来说，Alpha协调器可以通过是否接收到**ParticipationEnded**事件来进行判断。如果在超时时间内没有收到**ParticipationEnded**事件，Alpha事件扫描器会向自己发送**TccEnded**，触发和之前提到过的一样的事务错误处理流程，进行分布式事务关闭以及调用Omega进行相关恢复的操作。



同理当TCC分布式事务执行超时， Alpha事件扫描器会发送TccEnded触发相同的恢复操作。



**编写TCC应用代码**

下面我们会结合ServiceComb中的TCC示例[10]，向大家详解如何编写与TCC应用代码。这个示例以我们常见的电商场景为例，用户通过Ordering应用进行下单，Odering会调用Inventory以及Payment两个服务进行相关的业务操作。和之前Saga的示例一样，我们可以通过@EnableOmega的方式在这几个应用中注入Omega。

import org.apache.servicecomb.pack.omega.spring.EnableOmega**;**import org.springframework.boot.SpringApplication**;**import org.springframework.boot.autoconfigure.SpringBootApplication**;**@SpringBootApplication  
@EnableOmega  
**public** **class** **TccOrderingApplication** **{**

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

   SpringApplication**.**run**(**TccOrderingApplication**.**class**,** args**);**

**}  
}**

接下来我们需要在OrderingController中通过加入@TccStart来定义这个TCC分布式事务的范围，这个分布式事务的范围就是order方法，order方法会调用Inventory的order服务接口，以及Payment的pay服务接口。

import org.apache.servicecomb.pack.omega.context.annotations.TccStart**;**@TccStart  
@PostMapping**(**"/order/{userName}/{productName}/{productUnit}/{unitPrice}"**)**@ResponseBody  
**public** String **order(**

   @PathVariable String userName**,**

   @PathVariable String productName**,** @PathVariable Integer productUnit**,** @PathVariable Integer unitPrice**)** **{**

*// Calling the inventory service*

 restTemplate**.**postForEntity**(**

     inventoryServiceUrl **+** "/order/{userName}/{productName}/{productUnit}"**,**

**null,** String**.**class**,** userName**,** productName**,** productUnit**);**

**int** amount **=** productUnit **\*** unitPrice**;**

*// Calling the payment service*

 restTemplate**.**postForEntity**(**paymentServiceUrl **+** "/pay/{userName}/{amount}"**,**

**null,** String**.**class**,** userName**,** amount**);**

**return** userName **+** " ordering " **+** productName **+** " with " **+** productUnit **+** " OK"**;  
}**

Inventory会在try阶段先进行库存的扣减，在分布式业务执行成功之后进行设置库存订单状态；如果业务执行失败，Inventory服务会执行恢复操作。通过定义@Participate，ServiceComb Omega可以标注相关try方法，同时通过confirmMethod以及cancelMethod定义相关确认和取消方法名。需要注意的是，这里提到的confirm、cancel方法的参数必须和try方法的相同。

import org.apache.servicecomb.pack.omega.transaction.annotations.Participate**;**import org.springframework.transaction.annotation.Transactional**;**@Participate**(**confirmMethod **=** "confirm"**,** cancelMethod **=** "cancel"**)**@Transactional  
**public** **void** **pay(**Payment payment**)** **{**

 Account account **=** getAccount**(**payment**);**

**if** **(**account**.**getCredit**()** **>=** payment**.**getAmount**())** **{**

   account**.**setCredit**(**account**.**getCredit**()** **-** payment**.**getAmount**());**

   accountDao**.**saveAndFlush**(**account**);**

   payments**.**put**(**payment**.**getId**(),** payment**);**

**}** **else** **{**

**throw** **new** **IllegalArgumentException(**"Insufficient funds!"**);**

**}  
}**@Transactional  
Account **getAccount(**Payment payment**)** **{**

 Account account **=** accountDao**.**findByUserName**(**payment**.**getUserName**());**

**if** **(**Objects**.**isNull**(**account**))** **{**

**throw** **new** **IllegalArgumentException(**"Cannot find the account!"**);**

**}**

**return** account**;  
}**@Transactional  
**public** **void** **confirm(**Payment payment**)** **{**

 Account account **=** getAccount**(**payment**);**

 payment**.**setConfirmed**(true);**

 payment**.**setCancelled**(false);**

 account**.**setBalance**(**account**.**getBalance**()** **-** payment**.**getAmount**());**

 payment**.**setBalance**(**account**.**getBalance**());**

 accountDao**.**saveAndFlush**(**account**);  
}**@Transactional  
**public** **void** **cancel(**Payment payment**)** **{**

 Account account **=** getAccount**(**payment**);**

 account**.**setCredit**(**account**.**getCredit**()** **+** payment**.**getAmount**());**

 accountDao**.**saveAndFlush**(**account**);**

 payment**.**setBalance**(**account**.**getBalance**());**

 payment**.**setConfirmed**(false);**

 payment**.**setCancelled**(true);  
}**

Payment会在try阶段先验证用户账户的信用值，在确认阶段执行扣减用户账户余额的操作，在恢复阶段恢复信用值。

import org.apache.servicecomb.pack.omega.transaction.annotations.Participate**;**import org.springframework.transaction.annotation.Transactional**;**@Participate**(**confirmMethod **=** "confirm"**,** cancelMethod **=** "cancel"**)**@Transactional  
**public** **void** **reserve(**ProductOrder order**)** **{**

 Product product **=** getProduct**(**order**.**getProductName**());**

**if** **(**product**.**getInStock**()** **>=** order**.**getUnits**())** **{**

   product**.**setInStock**(**product**.**getInStock**()** **-** order**.**getUnits**());**

   productDao**.**saveAndFlush**(**product**);**

   orders**.**put**(**order**.**getId**(),** order**);**

**}** **else** **{**

**throw** **new** **IllegalArgumentException(**"The Product is out of stock!"**);**

**}  
}  
  
public** **void** **confirm(**ProductOrder order**)** **{**

 order**.**setConfirmed**(true);  
}**@Transactional  
**public** **void** **cancel(**ProductOrder order**)** **{**

 Product product **=** productDao**.**findProduceByName**(**order**.**getProductName**());**

 product**.**setInStock**(**product**.**getInStock**()** **+** order**.**getUnits**());**

 productDao**.**saveAndFlush**(**product**);**

 order**.**setCancelled**(true);  
}**@Transactional  
**private** Product **getProduct(**String productName**)** **{**

 Product product **=** productDao**.**findProduceByName**(**productName**);**

**if** **(**Objects**.**isNull**(**product**))** **{**

**throw** **new** **IllegalArgumentException(**"Product not exists at all!"**);**

**}**

**return** product**;  
}**

**小结**

在本文中我们先从分布式事务场景入手，采用交互图的方式，向大家介绍分布式Saga以及TCC分布式事务协调协议交互，最后结合ServiceComb Pack所提供的示例向大家介绍如何编写Saga以及TCC的应用代码。

**参考链接详情**

[1]ServiceComb Pack

https://github.com/apache/servicecomb-pack

[2]Sagas

https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf

[3]OmegaContext

https://willemjiang.github.io/servicecomb/pack/2019/01/10/[OmegaContext](https://github.com/apache/servicecomb-packblob/master/omega/omega-context/src/main/java/org/apache/servicecomb/saga/omega/context/OmegaContext.java)

[4]ServiceComb Pack Spring Demo

https://github.com/apache/servicecomb-pack/tree/master/demo/saga-spring-demo

[5]BookingController

https://github.com/apache/servicecomb-pack/blob/master/demo/saga-spring-demo/booking/src/main/java/org/apache/servicecomb/pack/demo/booking/BookingController.java

[6]RestTemplate

https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html

[7]Spring-Boot

https://spring.io/projects/spring-boot

[8]@EnableOmega

https://github.com/apache/servicecomb-pack/blob/master/omega/omega-spring-starter/src/main/java/org/apache/servicecomb/pack/omega/spring/EnableOmega.java

[9]配置

https://github.com/apache/servicecomb-pack/blob/master/demo/saga-spring-demo/booking/src/main/resources/application.yaml

[10]TCC示例

https://github.com/apache/servicecomb-pack/tree/master/demo/tcc-spring-demo