转帐历险记

Saga模式最初是由Hector Garcia-Molina和Kenneth Salem在他们的论文[Sagas中](http://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf)创造的。尽管最初在数据库管理系统的上下文中描述，但Saga模式在分布式系统上下文中已经变得流行，作为在处理业务流程中的多个远程参与者时管理故障的方式。这篇论文描述了一个历险故事：

一个周期长的事务，总可以写成一系列可以与其他事务交错的事务

或

系统保证saga中的所有事务都成功完成或者运行补偿事务来修改部分执行

对于业务流程中的每个步骤，如果出现故障，应该有一个补偿步骤，将系统状态回滚到应用该步骤之前的状态。整个过程由称为Saga Execution Component的东西管理。Saga模式的一个很好的介绍是[分布式Sagas：](https://www.youtube.com/watch?v=0UTOLRTwOX0&t=1696s) Caitie McCaffrey的[协调微服务协议](https://www.youtube.com/watch?v=0UTOLRTwOX0&t=1696s)，

## 场景

这里的场景是简单的银行账户转账 - 我们希望将10欧元从Account1转账到账户2。每个帐户的起始余额为10欧元。如果成功，Account1的余额为0欧元，而Account2的余额为20欧元。我们的目标是为帐户转帐流程提供类似交易的结果，即使其中一个步骤失败，系统也会处于一致状态（即整个流程失败，每个帐户产生10欧元，或者过程成功）

从表面上看，这是一个简单的问题。但是，当您将每个帐户视为远程服务时，事情会变得更加有趣，因此会遇到与远程调用相关的常见问题。帐户可能会以多种方式行为不端：

* 拒绝处理请求（例如，如果帐户因某种原因被暂停或关闭）
* 在调整帐户余额之前崩溃
* 调整帐户余额*后*崩溃
* 返回“我很忙”的回复
* 慢一点

对于我们的示例，我们只有一个区域可能需要应用补偿操作 - 如果我们能够从Account1借记但无法贷记Account2，则应该回滚借方（应当应用对Account1进行贷记的补偿操作） ）。否则，Account1将为0欧元，Account2将为10欧元，系统中已丢失10欧元。

对于看似简单的用例，有许多工作流程：

1. Account1成功处理 - > CREDIT ACCOUNT2
2. Account1拒绝借记请求。 - >停止
3. Account1响应“我很忙” - >重试
4. Account1没有响应 - >重试

如果Account1成功响应，那么我们有以下可能性

1. Account2成功处理 - > SUCCESS
2. Account2拒绝信用申请 - > ROLLBACK DEBIT
3. Account2响应“我很忙” - >重试
4. Account2没有响应 - >重试

如果我们必须回滚借记：

1. Account1成功处理 - > STOP
2. Account1拒绝信用请求 - > ESCALATE
3. Account1响应“我很忙” - >重试
4. Account1没有响应 - >重试

最后：

1. TransferProcess saga本身崩溃 - > RESUME

每种可能性都需要处理。在存在“我忙”响应或根本没有响应的情况下，我们应该重试该请求。如果请求被彻底拒绝，则重试没有意义，因此我们应该停止或回滚该传奇。

出现问题的一种情况是，即使在重试之后我们也没有收到对我们请求的回复。在我们的示例银行转帐流程中，我们知道在这种情况下发生了什么？ - 我们可以肯定地说帐户没有处理邮件吗？也许该帐户已贷记/借记但服务未能响应？如果记入帐户的第一步成功，但是记入帐户的第二步没有响应，我们是否回滚第一个帐户的借记？我们知道这是成功的，但我们不知道的是，这笔钱是否转到了另一个帐户。

如果补偿行动失败怎么办？

### 升级

在上一节中，我们描述了我们不确定系统处于什么状态的情况。即使重试和补偿操作，事情仍然可能出错。在一个理想的世界里，这些应该是非常罕见的！但是，它们可能会发生，在这些情况下，最好采用后备策略，将传奇的结果升级为其他内容，很可能是手动/人工过程。

### Atomicitiy

saga没有提供的一件事是原子的。在上面的银行帐户示例中，没有什么可以阻止其他系统与借记和贷记操作之间的帐户进行交互。这需要牢记，因为它可以排除某些类型的操作的saga模式。

## 实施汇款saga

传输过程saga的实现包含以下参与者：

* Account actor：模拟一个远程的服务
* TransferProcess actor：这是一个主要actor，负责协调借贷双方的记帐，从saga的角度来看，这是管理转移过程的Saga执行组件。
* AccountProxy actor：仅作为尝试与Account actor通信

Account Actor:

Account actor模拟远程银行帐户服务

**public** Task **ReceiveAsync**(IContext context)

{

**switch** (context.Message)

{

*// ...*

**case** Credit msg:

**return** AdjustBalance(msg.ReplyTo, msg.Amount);

**case** Debit msg when msg.Amount + \_balance >= 0:

**return** AdjustBalance(msg.ReplyTo, msg.Amount);

*// ...*

}

**return** Actor.Done;

}

当一个Credit或Debit收到要求，我们试图调整账户的余额。尝试可能由于多种原因而失败：

**private** Task AdjustBalance(PID replyTo, decimal amount)

{

**if** (RefusePermanently())

{

\_processedMessages.Add(replyTo, **new** **Refused**());

replyTo.Tell(**new** **Refused**());

}

**if** (Busy())

replyTo.Tell(**new** **ServiceUnavailable**());

**var** behaviour = DetermineProcessingBehavior();

**if** (behaviour == Behavior.FailBeforeProcessing)

**return** Failure(replyTo);

*// simulate potential slow service*

Thread.Sleep(\_random.Next(0, 150));

\_balance += amount; \_processedMessages.Add(replyTo, **new** **OK**());

**if** (behaviour == Behavior.FailAfterProcessing)

**return** Failure(replyTo);

replyTo.Tell(**new** **OK**());

**return** Actor.Done;

}

Account Proxy:

AccountProxy被创建的的唯一目的是TransferProcess actor与Account actor通信和处理对策：

**class** **AccountProxy** : **IActor**

{

**private** **readonly** PID \_account;

**private** **readonly** Func<PID, **object**> \_createMessage;

**public** **AccountProxy**(PID account, Func<PID, **object**> createMessage)

{

\_account = account;

\_createMessage = createMessage;

}

**public** Task **ReceiveAsync**(IContext context)

{

**Switch**(context.Message)

{

**case** Started \_:

\_account.Tell(\_createMessage(context.Self)); context.SetReceiveTimeout(TimeSpan.FromMilliseconds(100));

**break**;

**case** OK msg:

context.CancelReceiveTimeout();

context.Parent.Tell(msg);

**break**;

**case** Refused msg: context.CancelReceiveTimeout();

context.Parent.Tell(msg);

**break**;

*// These represent a failed remote call*

**case** InternalServerError \_:

**case** ReceiveTimeout \_:

**case** ServiceUnavailable \_:

**throw** **new** Exception();

}

**return** Actor.Done;

}

}

在AccountProxy将消息发送到所述Account上Started，并设置在其中的响应应该被接收的100毫秒超时。这允许我们模拟慢速远程进程并将超时作为错误处理。给定任何表示失败的远程调用的消息，AccountProxy只是抛出异常，将责任传递给父级以决定做什么。如果消息是，OK或者Refused我们取消超时并将消息传递给父，即TransferProcess actor。

Transfer Process actor

TransferProcess actor 管理着所有的传递，在saga述中，它是saga扫行组件。这个类跟踪已完成的步骤，并对故障进行补偿操作。

TransferProcess actor的实现利用：

* 行为插件
* 持久化插件
* 错误内核模式
* 监控

使用行为：

我们的传输过程使用Behavior插件建模为状态机。这允许我们根据当前状态交换消息处理代码。这是通过在处理消息时委托给Behavior类来实现的：

**public** **async** Task **ReceiveAsync**(IContext context)

{

**await** \_behavior.ReceiveAsync(context);

}

这里TransferProcess的ReceiveAsync方法只是代表了Behavior的ReceiveAsync方法。

TransferProcess状态有

* Starting
* AwaitingDebitConfirmation
* AwaitingCreditConfirmation
* RollingBackDebit

让我们看看每个状态，看看有什么变化。

Staring

Staring 是这个actor的初始化状态，并且进行尝试借方

**private** **async** Task **Starting**(IContext context)

{

**if** (context.Message **is** Started)

{

context.SpawnNamed(TryDebit(\_from, -\_amount), "DebitAttempt");

\_behavior.Become(AwaitingDebitConfirmation);

}

}

**private** Props **TryDebit**(PID targetActor, **decimal** amount) => Actor

.FromProducer(() => **new** AccountProxy(targetActor, sender => **new** Debit(amount, sender)));

在这里，我们创建一个专门用于处理借记尝试并将actor转换为AwaitingDebitConfirmation状态的actor 。

Awaiting Debit Confirmation

在这个状态好有3种可能的转换

* AwaitingDebitConfirmation - > AwaitingCreditConfirmation
* AwaitingDebitConfirmation - >停止（ConsistentSystem）
* AwaitingDebitConfirmation - >停止（未知）

**private** Task AwaitingDebitConfirmation(IContext context)

{

**switch** (context.Message)

{

**case** OK \_:

\_behavior.Become(AwaitingCreditConfirmation);

context.SpawnNamed(TryCredit(\_to, +\_amount), "CreditAttempt");

**break**;

**case** Refused \_:

\_logger.Log("Transfer failed. System consistent") ;

StopAll(context);

**break**;

**case** Terminated \_:

\_logger.Log("Transfer status unknown. Escalate");

StopAll(context);

**break**;

}

}

**private** Props TryCredit(PID targetActor, decimal amount) => Actor

.FromProducer(() => **new** **AccountProxy**(targetActor, sender => **new** **Credit**(amount, sender)));

我们DebitAttempt演员的3个可能结果是：

##### 1.借记成功

**case** OK **\_:**

\_behavior.Become(AwaitingCreditConfirmation);

context.SpawnNamed(TryCredit(\_to, +\_amount), "CreditAttempt");

第一步的借记是成功的，我们现在可以创建CreditAttempt演员并过渡到AwaitingCreditConfirmation州。

##### 2.借记卡被拒绝

**case** Refused \_:

\_logger.Log("Transfer failed. System consistent")

StopAll(context);

借方被拒绝，不应重审。我们记录并停止。

##### 3.未知

**case** Terminated \_:

\_logger.Log("Transfer status unknown. Escalate")

StopAll(context);

Terminated在AwaitingDebitConfirmation状态中接收消息意味着DebitActor已经停止，并且表示借记过程的未知故障。关于监督和错误内核的部分描述了如何Terminated创建此消息，但现在足以知道借记已被重试但最终失败。我们别无选择，只能停下来。

Awaiting Credit Confirmation

如果借记成功，我们就会过渡到AwaitingCreditConfirmation州。在这种状态下，有3种可能的转换：

* AwaitingCreditConfirmation - >停止（成功）
* AwaitingCreditConfirmation - > RollingBackDebit
* AwaitingCreditConfirmation - >停止（未知）

private async Task AwaitingCreditConfirmation(IContext context)

{

**switch** (context.Message)

{

*//...*

**case** OK msg:

\_logger.Log("Success!") StopAll(context);

break;

**case** Refused msg:

\_behavior.Become(RollingBackDebit);

context.SpawnNamed(TryCredit(\_from, +\_amount), "RollbackDebit");

break;

**case** Terminated msg:

\_logger.Log("Transfer status unknown. Escalate") ;

StopAll(context);

break;

}

}

##### 1.贷方成功

case OK msg:

\_logger.Log("Success!");

StopAll(context);

这代表了畅通的方式 - 我们已经成功完成了转移过程。记录我们已完成并停止的事实。

##### 2.贷方拒绝

case Refused msg:

\_behavior.Become(RollingBackDebit)

context.SpawnNamed(TryCredit(\_from, +\_amount), "RollbackDebit");

贷方尝试被拒绝了。我们知道借方成功但贷方被拒绝，因此我们可以尝试通过贷方账户来应用回滚借方的补偿行动。我们创建一个RollbackDebit actor并过渡到RollingBackDebit状态。

##### 3.未知

case Terminated msg:

\_logger.Log("Transfer status unknown. Escalate")

StopAll(context);

Terminated在AwaitingCreditConfirmation状态中接收消息意味着CreditActor已经停止，并且表示信用过程的未知故障。我们现在处于这样一种状态：我们知道借记成功但却不知道贷方状况。它可能已经成功了。我最初转移到RollingBackDebit这里的状态，但最终决定最好触发任何升级策略来查看这种状态下的转移。

Rolling Back Debit

如果我们的借方成功但我们的贷方被拒绝，我们将过渡到RollingBackDebit有两种可能的过渡的状态：

* RollingBackDebit - >停止（ConsistentSystem）
* RollingBackDebit - >停止（未知）

**private** async Task RollingBackDebit(IContext context)

{

**switch** (context.Message)

{

//...

**case** OK \_:

\_logger.Log("Transfer failed. System consistent")

StopAll(context);

**break**;

**case** Refused \_:

**case** Terminated \_:

\_logger.Log("Transfer status unknown. Escalate")

StopAll(context);

**break**;

}

}

在这种状态下，只有两种可能的转换：

##### 1.回滚成功

case OK \_:

\_logger.Log("Transfer failed. System consistent")

StopAll(context);

我们已成功应用补偿操作，即使实际传输过程失败，系统也处于一致状态。

##### 2.失败

case Refused \_:

case Terminated \_:

\_logger.Log("Transfer status unknown. Escalate")

StopAll(context);

失败可能是拒绝操作或操作员RollbackDebit终止的帐户的结果。如果操作被拒绝，则意味着在进行原始借记和尝试回滚信用之间，帐户已决定拒绝该请求。

如果RollbackDebit演员被终止，它再次意味着我们不知道该操作发生了什么，并且再次别无选择，只能放弃并依赖升级。

我们现在已经通过使用行为插件覆盖了所有可能的状态转换。但是，如果它TransferProcess本身崩溃了怎么办？在第3部分中，我们将介绍如何通过使用Persistence插件来保存TransferProcess状态来处理这种情况。