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Account2。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“忙”的回复
* 太慢

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 - 如果我们已经从Account1扣除但无法存入Account2，则应该回滚借方（应当应用对Account1进行存入的补偿操作）。否则，Account1将为0欧元，Account2将为10欧元，系统中丢失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。 - >STOP
3. Account1响应“我很忙” - >RETRY
4. Account1没有响应 - >RETRY

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” - >RETRY
4. Account2没有响应 - >RETRY

如果我们必须回滚借方：

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

最后：

1. TransferProcess saga本身崩溃 - > RESUME

每种可能性都需要处理。在存在“Busy”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历险。

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？也许该帐户已存入/扣除，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扣除？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Escalation

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。

### 原子性Atomicitiy

Saga没有提供的一件事是原子的。在上面的银行转帐示例中，没有什么可以阻止其他系统与借方和贷方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。

## 实施汇款历险

转帐Saga的实现包含以下Actor：

* 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，这是管理转帐过程的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

class Account : IActor

{

/// <summary>

/// 名称

/// </summary>

private readonly string \_name;

/// <summary>

/// 服务时间

/// </summary>

private readonly double \_serviceUptime;

/// <summary>

/// 拒绝概率

/// </summary>

private readonly double \_refusalProbability;

/// <summary>

/// 繁忙的概率

/// </summary>

private readonly double \_busyProbability;

private readonly Dictionary<PID, object> \_processedMessages = new Dictionary<PID, object>();

/// <summary>

/// 余额

/// </summary>

private decimal \_balance = 10;

private readonly Random \_random;

/// <summary>

///

/// </summary>

/// <param name="name">名称</param>

/// <param name="serviceUptime">服务时间</param>

/// <param name="refusalProbability">拒绝概率</param>

/// <param name="busyProbability">繁忙的概率</param>

/// <param name="random"></param>

public Account(string name, double serviceUptime, double refusalProbability, double busyProbability, Random random)

{

\_name = name;

\_serviceUptime = serviceUptime;

\_refusalProbability = refusalProbability;

\_busyProbability = busyProbability;

\_random = random;

}

public Task ReceiveAsync(IContext context)

{

switch (context.Message)

{

//贷方

case Credit msg when \_processedMessages.ContainsKey(msg.ReplyTo):

msg.ReplyTo.Tell(\_processedMessages[msg.ReplyTo]);

return Actor.Done;

//贷方

case Credit msg:

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

//借方

case Debit msg when \_processedMessages.ContainsKey(msg.ReplyTo):

msg.ReplyTo.Tell(\_processedMessages[msg.ReplyTo]);

return Actor.Done;

//借方

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

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

//借方

case Debit msg:

msg.ReplyTo.Tell(new InsufficientFunds());

break;

//获取余额

case GetBalance \_:

context.Respond(\_balance);

break;

}

return Actor.Done;

}

/// <summary>

/// 调整帐户

/// </summary>

private Task AdjustBalance(PID replyTo, decimal amount)

{

#region

//永久拒绝

if (RefusePermanently())

{

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

replyTo.Tell(new Refused());

}

//繁忙

if (Busy())

{

replyTo.Tell(new ServiceUnavailable());

}

//生成处理此消息时要使用的行为

var behaviour = DetermineProcessingBehavior();

if (behaviour == Behavior.FailBeforeProcessing)

{

replyTo.Tell(new InternalServerError());

return Actor.Done;

}

//模拟潜在的长期运行过程

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

#endregion

\_balance += amount;

\_processedMessages.Add(replyTo, new OK());

//模拟处理后失败，这将迫使引起重试操作，并测试幂等操作

if (behaviour == Behavior.FailAfterProcessing)

{

replyTo.Tell(new InternalServerError());

return Actor.Done;

}

replyTo.Tell(new OK());

return Actor.Done;

}

/// <summary>

/// 繁忙

/// </summary>

/// <returns></returns>

private bool Busy()

{

var comparsion = \_random.NextDouble() \* 100;

return comparsion <= \_busyProbability;

}

/// <summary>

/// 永久拒绝

/// </summary>

/// <returns></returns>

private bool RefusePermanently()

{

var comparsion = \_random.NextDouble() \* 100;

return comparsion <= \_refusalProbability;

}

/// <summary>

/// 确定处理行为

/// </summary>

/// <returns></returns>

private Behavior DetermineProcessingBehavior()

{

var comparision = \_random.NextDouble() \* 100;

if (comparision > \_serviceUptime)

{

return \_random.NextDouble() \* 100 > 50 ? Behavior.FailBeforeProcessing : Behavior.FailAfterProcessing;

}

return Behavior.ProcessSuccessfully;

}

/// <summary>

/// 处理结果

/// </summary>

private enum Behavior

{

/// <summary>

/// 处理前失败

/// </summary>

FailBeforeProcessing,

/// <summary>

/// 处理后失败

/// </summary>

FailAfterProcessing,

/// <summary>

/// 处理成功

/// </summary>

ProcessSuccessfully

}

}

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)

{

#region

//永久拒绝

if (RefusePermanently())

{

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

replyTo.Tell(new Refused());

}

//繁忙

if (Busy())

{

replyTo.Tell(new ServiceUnavailable());

}

//生成处理此消息时要使用的行为

var behaviour = DetermineProcessingBehavior();

if (behaviour == Behavior.FailBeforeProcessing)

{

replyTo.Tell(new InternalServerError());

return Actor.Done;

}

//模拟潜在的长期运行过程

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

#endregion

\_balance += amount;

\_processedMessages.Add(replyTo, new OK());

//模拟处理后失败，这将迫使引起重试操作，并测试幂等操作

if (behaviour == Behavior.FailAfterProcessing)

{

replyTo.Tell(new InternalServerError());

return Actor.Done;

}

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 \_target;

private readonly Func<PID, object> \_createMessage;

public AccountProxy(PID target, Func<PID, object> createMessage)

{

\_target = target;

\_createMessage = createMessage;

}

public Task ReceiveAsync(IContext context)

{

switch (context.Message)

{

case Started \_:

//想象这是某种远程调用而不是本地Actor调用

\_target.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;

case InsufficientFunds \_:

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，并设置接收响应超时时间为100毫秒，这是为了我们模拟远程返回超时的错误处理。如果发生远程调用异常，AccountProxy只是抛出异常，将Supervisor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，它是saga执行组件。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：

* Behavior插件
* Persistence插件
* 错误内核模式
* Supervision

#### 使用Behavior：

我们的转帐过程使用Behavior插件建模状态机。这允许我们根据当前状态来使用对应的代码处理消息。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)

{

var props = Actor.FromProducer(() => new AccountProxy(\_from, sender => new Debit(-\_amount, sender)));

context.SpawnNamed(props, "DebitAttempt");

\_behavior.Become(AwaitingDebitConfirmation);

}

}

在这里，我们创建一个专门用于处理尝试扣除并将actor转换为AwaitingDebitConfirmation状态的actor，这是TransferProcess actor的子actor 。

##### Awaiting Debit Confirmation

在这个状态有3种可能的转换结果

* AwaitingDebitConfirmation - > AwaitingCreditConfirmation
* AwaitingDebitConfirmation - >Stop（ConsistentSystem）
* AwaitingDebitConfirmation - >Stop（Unknown）

public async Task ReceiveAsync(IContext context)

{

switch (context.Message)

{

case OK \_:

\_behavior.Become(AwaitingCreditConfirmation);

var props\_ok = Actor.FromProducer(() => new AccountProxy(\_to, sender => new Credit(+\_amount, sender)));

context.SpawnNamed(props\_ok, "CreditAttempt");

break;

case Refused \_:

StopAll(context);

break;

case Terminated \_:

StopAll(context);

break;

}

}

我们DebitAttempt演员的3个可能结果是：

###### 1.扣款成功

case OK \_:

\_behavior.Become(AwaitingCreditConfirmation);

var props\_ok = Actor.FromProducer(() => new AccountProxy(\_to, sender => new Credit(+\_amount, sender)));

context.SpawnNamed(props\_ok, "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状态。

###### 2.借记卡被拒绝

case Refused \_:

StopAll(context);

借方被拒绝，不应重试。我们记录并停止。

###### 3.未知

case Terminated \_:

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Supervisior和错误内核的部分描述了如何Terminated创建此消息，但现在足以知道扣款已被重试但最终失败。我们别无选择，只能停下来。

##### Awaiting Credit Confirmation

如果扣款成功，我们就会过渡到AwaitingCreditConfirmation状态。在这种状态下，有3种可能的转换结果：

* AwaitingCreditConfirmation - >Stop（Success）
* AwaitingCreditConfirmation - > RollingBackDebit
* AwaitingCreditConfirmation - >Stop（Unknown）

private async Task AwaitingCreditConfirmation(IContext context)

{

switch (context.Message)

{

*//...*

case OK msg:\_

StopAll(context);

reak;

case Refused msg:

\_behavior.Become(RollingBackDebit);

var props\_refused = Actor.FromProducer(() => new AccountProxy(\_from, sender => new Credit(+\_amount, sender)));

context.SpawnNamed(props\_refused, "RollbackDebit");

break;

case Terminated msg:

StopAll(context);

break;

}

}

###### 1.存入成功

case OK msg:\_

StopAll(context);

reak;

这代表了最佳的方式 - 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);

var props\_refused = Actor.FromProducer(() => new AccountProxy(\_from, sender => new Credit(+\_amount, sender)));

context.SpawnNamed(props\_refused, "RollbackDebit")

贷方尝试被拒绝了。我们知道借方扣款成功但贷方存放被拒绝，因此我们可以尝试通过贷方账户来回滚借方的补偿操作。我们创建一个RollbackDebit actor并过渡到RollingBackDebit状态。

###### 3.未知

case Terminated msg:

StopAll(context);

break;

Terminated在AwaitingCreditConfirmation状态中接收消息意味着CreditActor已经停止，并且表示贷方处理过程的未知故障。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帐情况。

##### 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 - >Stop（ConsistentSystem）
* RollingBackDebit - >Stop（Unkown）

private async Task RollingBackDebit(IContext context)

{

switch (context.Message)

{

//...

case OK \_:

StopAll(context);

break;

case Refused \_:

case Terminated \_:

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 \_:

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 \_:

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贷方之间，帐户已决定拒绝该请求。

如果RollbackDebit actor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Behavior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。

## 审核日志

原始的saga讨论用log来跟踪saga的活动。TransferProcess我们在第2部分中创建的actor覆盖了我们的saga的所有状态转换，但没有保存它的状态，所以很容易崩溃。我们可以通过使用Persistence插件来解决这个问题。

Persistence插件允许actor在通过使用事件源重新启动或重新加载时恢复它的状态。每个状态更改都被建模为在恢复阶段和实时运行时 “应用”到actor的事件。Persistence插件将Action<Event> applyEvent方法作为参数 - 每当保存事件时调用此方法，或者在恢复期间从底层存储中加载。重要的是，在此ApplyEvent方法中定义所有状态更改，包括转换为不同的行为。在我们当前的TransferProcess代码中，我们通过调用处理消息时转换为新的行为\_behavior.Become(SomeState)。但是，为了使actor能够从失败中恢复并以最终状态结束，所有状态更改都需要移动到该ApplyEvent方法，我们需要将事件保存为检查点。因此，例如，我们的起始代码更改为

private async Task Starting(IContext context)

{

if (context.Message is Started)

{

var props = Actor.FromProducer(() => new AccountProxy(\_from, sender => new Debit(-\_amount, sender)));

context.SpawnNamed(props, "DebitAttempt");

\_behavior.Become(AwaitingDebitConfirmation);

}

}

至

private async Task Starting(IContext context)

{

if (context.Message is Started)

{

var props = Actor.FromProducer(() => new AccountProxy(\_from, sender => new Debit(-\_amount, sender)));

context.SpawnNamed(props, "DebitAttempt");

await \_persistence.PersistEventAsync(new TransferStarted());

}

}

将相应的行为更改添加到ApplyEvent方法中：

private void ApplyEvent(Event @event)

{

switch (@event.Data)

{

//转帐开始

case TransferStarted msg:

//换成等待贷方确认

\_behavior.Become(AwaitingDebitConfirmation);

break;

// ... other transitions

}

}

将所有状态转换移动到ApplyEvent方法中的副作用是它现在读起来像我们在第1部分中描述的状态机：

private void ApplyEvent(Event @event)

{

switch (@event.Data)

{

//转帐开始

case TransferStarted msg:

//转换成等待贷方确认

\_behavior.Become(AwaitingDebitConfirmation);

break;

//借方扣除

case AccountDebited msg:

//转换成等待借方确认

\_behavior.Become(AwaitingCreditConfirmation);

break;

//贷方拒绝

case CreditRefused msg:

\_behavior.Become(RollingBackDebit);

break;

//贷方存入

case AccountCredited \_:

//借方回滚

case DebitRolledBack \_:

//转帐失败

case TransferFailed \_:

\_processCompleted = true;

break;

}

}

记录所有状态更改事件，是我们从故障中恢复的原因。它TransferProcess本身是受监督的，因此当发生故障时，它会由它的父节点重新启动。这会产生一个Started消息，我们可以在其中重建状态TransferProcess。我们TransferProcess ReceiveAsync在委托给Behavior类之前在方法中执行此操作：

public async Task ReceiveAsync(IContext context)

{

switch (context.Message)

{

case Started msg:

//自定义开始的行为

\_behavior.Become(Starting);

//从持久化库中恢复，如果当有一些事件，当前行为将会改变

await \_persistence.RecoverStateAsync();

break;

// ...

}

await \_behavior.ReceiveAsync(context);

}

RecoverStateAsync将从我们的存储加载所有事件并在其上调用ApplyEvent方法，从而重建我们的行为状态转换，并将actor重新置于正确的状态。注意Starting状态的默认值- 如果我们没有任何事件需要在Starting状态中开始。如果我们确实有事件，那么状态将正常过渡。

##### 忧虑的父母（Parent Actor） - 我们的孩子(Children Actor)去哪儿了？

在我们当前的设计中，当我们从先前的状态接收消息时，创建尝试贷方和借方操作的actor。例如，在AwaitingDebitConfirmation 状态，如果我们收到成功的回复，我们会产生一个新的actor来尝试贷方：

private async Task AwaitingDebitConfirmation(IContext context)

{

switch (context.Message)

{

case OK \_:

//借方扣除成功

await \_persistence.PersistEventAsync(new AccountDebited());

var props\_ok = Actor.FromProducer(() => new AccountProxy(\_to, sender => new Credit(+\_amount, sender)));

context.SpawnNamed(props\_ok, "CreditAttempt");

break;

//...

}

}

但是有一个问题 - 考虑OK收到消息的场景，TransferProcess保存AccountDebited事件，转换到AwaitingCreditConfirmation状态，生成CreditAttempt actor并随后崩溃 - 这将导致所有子节点终止，包括CreditAttempt刚创建的actor 。当TransferProcess重新启动时，我们重新加载所有事件并转换到AwaitingCreditConfirmation状态，我们正在等待CreditAttempt已终止的actor 的响应。由于未重新启动子项，因此不会收到任何消息。

解决此问题的一种方法是CreditAttempt在TransferProcess接收Started事件时重新创建actor ：

private async Task AwaitingCreditConfirmation(IContext context)

{

switch (context.Message)

{

case Started \_:

//如果我们在启动时处于这种状态，那么我们需要重新创建

var props\_started = Actor.FromProducer(() => new AccountProxy(\_to, sender => new Credit(+\_amount, sender)));

context.SpawnNamed(props\_started, "CreditAttempt");

break;

//..... other code omitted

}

}

这将恢复CreditAttempt actor，并将TransferProcess actor恢复到崩溃前的状态。

##### 为什么我的孩子终止了？

当前设计的另一个问题是Terminated消息用于指示AccountProxy actors操作失败。在考虑TransferProcess重启时会发生什么时，这会出现问题。该AccountProxy actor是TransferProcess actor的子actor，重新启动过程中发生的情况是一个actor的所有子域得到停止。因此，当TransferPorcess重新启动actor 时，它会因子actor AccountProxy停止而收到Terminated消息。这现在打破了我们对Terminated消息所代表的语义的意义- AccountProxy它没有与Account通信，而TransferProcess 自身也失败了。TransferProcess我们可以追踪这种情况，而不是重新设计：

public async Task ReceiveAsync(IContext context)

{

switch (context.Message)

{

//.....

case Stopping msg:

\_stopping = true;

break;

case Restarting msg:

\_restarting = true;

break;

case Terminated \_ when \_restarting || \_stopping:

return;

//.....

}

await \_behavior.ReceiveAsync(context);

}

当TransferProcess重新启动或停止时，它接收我们能够处理的消息。这发生在孩子们停止之前。通过在发生这些事件时设置标志，我们可以Terminated在此状态下安全地忽略消息。我们立即从ReceiveAsync阻止await \_behavior.ReceiveAsync(context)方法执行的方法返回。false重启过程完成后，这些标志将重置为其默认值。

## Supervision，错误内核和幂等性

实现saga的一个重要因素是重试操作的概念。您将注意到，在尝试我们的借记和贷记操作时，会创建一个单独的actor来执行此操作。我们可以依靠actor系统中的监督策略来帮助重试失败的操作。如果此调用出现任何问题，则此actor失败，并允许其父级对失败采取行动......由于此actor是TransferProcess actor的子域，因此由TransferProcess actor 监督。这允许我们使用错误内核模式来推出将远程服务调用到单独的actor中的“危险”操作，并为该actor设置监督策略。在这种情况下，我们的监督策略说重试10次（任何失败的SupervisorDirective.Restart）然后放弃并停止actor：

var retryAttempts = 10;

var supervisionStrategy = new OneForOneStrategy((pid, reason) =>

SupervisorDirective.Restart, retryAttempts)

Actor.FromProducer(() => new TransferProcess(...)

.WithChildSupervisorStrategy(supervisionStrategy);

##### 幂等接收者

鉴于我们正在重新对帐户进行操作，我们需要他们成为幂等接收者。在消息传递应用程序中，幂等性定义为

无论是一次还是多次接收都会产生相同效果的消息。这意味着即使接收器接收到相同消息的重复，也可以安全地重新发送消息而不会引起任何问题。[EIP](http://www.enterpriseintegrationpatterns.com/patterns/messaging/IdempotentReceiver.html)

有些消息自然是幂等的，有些则不是。在这种情况下，递增或递减帐户的余额并不是自然是幂等的，因此我们需要跟踪我们已经处理过的消息。每次我们创建AccountProxy它时都具有相同的PID，我们可以使用它来跟踪已处理的消息。当Account actor收到重复的消息时，我们可以发送与原始消息相同的响应：

private readonly Dictionary<PID, object> \_processedMessages = new Dictionary<PID, object>();

public Task ReceiveAsync(IContext context)

{

switch (context.Message)

{

//贷方

case Credit msg when \_processedMessages.ContainsKey(msg.ReplyTo):

msg.ReplyTo.Tell(\_processedMessages[msg.ReplyTo]);

return Actor.Done;

//贷方

case Credit msg:

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

//...

}

return Actor.Done;

}

我们第一次收到Credit，集合中是不包含这个PID的，将会进入到第二人case，这时将变更帐户金额，并且保存ReplyTo（这是AccountProxy的PID,并且在TransferProcess中是唯一的）和响应消息的结果。如果我们再次收到相同的消息，将触发第一个case语句，我们将回复原始响应。

注意\_processedMessages当TransferProcess演员被终止时，有清除集合的余地，这是“ [反应设计模式](https://www.manning.com/books/reactive-design-patterns) ”一书中[商务握手模式](https://github.com/ReactiveDesignPatterns/CodeSamples/blob/master/chapter15/src/main/scala/com/reactivedesignpatterns/chapter15/BusinessHandshake.scala)中采用的一种技术

## 结果

我们现在已经完成了Money Transfer Saga的实施，但是我们需要一种方法来运行它，以及一种产生故障的方法。要运行saga，我们将使用一个简单的控制台应用程序：

class Program

{

static void Main(string[] args)

{

while (true)

{

Console.WriteLine("Starting");

var numberOfTransfers = 1;

var uptime = 99.99;

var retryAttempts = 0;

var refusalProbability = 0.01;

var busyProbability = 0.01;

var props = Actor.FromProducer(() => new Runner(numberOfTransfers, uptime, refusalProbability, busyProbability, retryAttempts, false))

.WithChildSupervisorStrategy(new OneForOneStrategy((pid, reason) => SupervisorDirective.Restart, retryAttempts, null));

Console.WriteLine("Spawning runner");

var runner = Actor.SpawnNamed(props, "runner");

Console.ReadLine();

runner.Stop();

}

}

}

在这里，我们设置一些变量供Account演员使用以模拟各种故障情形：

* uptime确定Account失败的概率
  + refusalProbability确定Account拒绝贷方或借方请求的概率
  + busyProbability确定Account将返回ServiceUnavailable响应的概率

我们还指定了我们将要运行多少saga以及将进行多少次重试。然后我们创建一个Runner actor来运行saga。

##### Runner

该Runner角色负责运行的saga故事，并收集和对结果的报告。它实现了一个散布 - 聚集模式来生成TransferProcessactor，然后在它们全部完成时进行报告：

public Task ReceiveAsync(IContext context)

{

switch (context.Message)

{

//...

case Started \_:

var random = new Random();

var dbfile = @"C:\MyFile\Source\Repos\ProtoActorSample\ProtoActorSample\P100\_Saga\data.sqlite";

var sqliteProvider = new SqliteProvider(new SqliteConnectionStringBuilder() { DataSource = dbfile });

\_inMemoryProvider = sqliteProvider;// new InMemoryProvider();

for (int i = 1; i <= \_numberOfIterations; i++)

{

int j = i;

var fromAccount = CreateAccount($"FromAccount{j}", random);

var toAccount = CreateAccount($"ToAccount{j}", random);

var transferProps = Actor.FromProducer(() => new TransferProcess(fromAccount, toAccount, 10m, \_inMemoryProvider, $"Transfer Process {j}", random, \_uptime))

.WithChildSupervisorStrategy(

new OneForOneStrategy((pid, reason) => SupervisorDirective.Restart, \_retryAttempts,null));

var transfer = context.SpawnNamed(transferProps, $"Transfer Process {j}");

\_transfers.Add(transfer);

if (\_numberOfIterations >= 10)

{

if (j % (\_numberOfIterations / 10) == 0)

Console.WriteLine($"Started {j}/{\_numberOfIterations} processes");

}

else

{

Console.WriteLine($"Started {j}/{\_numberOfIterations} processes");

}

}

break;

}

return Actor.Done;

}

一旦Runner启动，它会循环遍历迭代次数，每次创建两个Accountactor和一个TransferProcessactor，将TransferProcessPID 添加到\_transfers集合中。在Runner监督中的TransferActor，并负责将其重新启动应该会崩溃。在TransferProcessactor 内部，调用context.Parent.Tell();通知Runner结果。在Runner随后等待从接收结果TransferProcess的演员：

public Task ReceiveAsync(IContext context)

{

switch (context.Message)

{

case SuccessResult msg:

\_successResults++;

CheckForCompletion(msg.Pid);

break;

case UnknownResult msg:

\_unknownResults++;

CheckForCompletion(msg.Pid);

break;

case FailedAndInconsistent msg:

\_failedAndInconsistentResults++;

CheckForCompletion(msg.Pid);

break;

case FailedButConsistentResult msg:

\_failedButConsistentResults++;

CheckForCompletion(msg.Pid);

break;

//...

}

对于每种结果类型，计数器递增以跟踪不同的结果类型，然后执行完成检查以确定是否所有传奇都已完成。如果是，则输出结果：

private void CheckForCompletion(PID pid)

{

\_transfers.Remove(pid);

var remaining = \_transfers.Count;

if (\_numberOfIterations >= 10)

{

Console.Write(".");

if (remaining % (\_numberOfIterations / 10) == 0)

{

Console.WriteLine();

Console.WriteLine($"{remaining} processes remaining");

}

}

else

{

Console.WriteLine($"{remaining} processes remaining");

}

if (remaining == 0)

{

Thread.Sleep(250);

Console.WriteLine();

Console.WriteLine(

$"RESULTS for {\_uptime}% uptime, {\_refusalProbability}% chance of refusal, {\_busyProbability}% of being busy and {\_retryAttempts} retry attempts:");

Console.WriteLine(

$"{AsPercentage(\_numberOfIterations, \_successResults)}% ({\_successResults}/{\_numberOfIterations}) successful transfers");

Console.WriteLine(

$"{AsPercentage(\_numberOfIterations, \_failedButConsistentResults)}% ({\_failedButConsistentResults}/{\_numberOfIterations}) failures leaving a consistent system");

Console.WriteLine(

$"{AsPercentage(\_numberOfIterations, \_failedAndInconsistentResults)}% ({\_failedAndInconsistentResults}/{\_numberOfIterations}) failures leaving an inconsistent system");

Console.WriteLine(

$"{AsPercentage(\_numberOfIterations, \_unknownResults)}% ({\_unknownResults}/{\_numberOfIterations}) unknown results");

}

## 一些例子

那么事情怎么样？给定足够的正常运行时间和足够的重试尝试，事情看起来很好：

RESULTS **for** 99.99% uptime, 0.01% chance of refusal, 0.05% of being busy **and** 3 retry attempts:

- 100% (1000/1000) successful transfers

- 0% (0/1000) failures leaving a consistent system

- 0% (0/1000) failures leaving an inconsistent system

- 0% (0/1000) unknown results

即使我们降低正常运行时间并增加繁忙的可能性，事情看起来仍然很好：

RESULTS **for** 99% uptime, 0.01% chance of refusal, 0.1% of being busy **and** 3 retry attempts:

- 100% (1000/1000) successful transfers

- 0% (0/1000) failures leaving a consistent system

- 0% (0/1000) failures leaving an inconsistent system

- 0% (0/1000) unknown results

我们必须大大减少正常运行时间才能开始看到不同的东西：

RESULTS **for** 90% uptime, 0.01% chance of refusal, 0.1% of being busy **and** 3 retry attempts:

- 99.9% (999/1000) successful transfers

- 0.2% (2/1000) failures leaving a consistent system

- 0% (0/1000) failures leaving an inconsistent system

- 0% (0/1000) unknown results

删除重试尝试会显着影响我们的结果：

RESULTS **for** 90% uptime, 0.01% chance of refusal, 0.1% of being busy **and** 1 retry attempts:

- 92% (920/1000) successful transfers

- 0% (0/1000) failures leaving a consistent system

- 0% (0/1000) failures leaving an inconsistent system

- 8% (80/1000) unknown results

显着增加重试次数使我们能够应对一个非常容易出错的系统：

RESULTS **for** 50% uptime, 0.01% chance of refusal, 0.1% of being busy **and** 15 retry attempts:

- 100% (1000/1000) successful transfers

- 0% (0/1000) failures leaving a consistent system

- 0% (0/1000) failures leaving an inconsistent system

- 0% (0/1000) unknown results

增加拒绝的可能性会产生很大的影响，因为重试不会发生：

RESULTS **for** 50% uptime, 20.1% chance of refusal, 0.2% of being busy **and** 15 retry attempts:

68.9% (689/1000) successful transfers

29.2% (292/1000) failures leaving a consistent system

4.6% (46/1000) failures leaving an inconsistent system

0.1% (1/1000) unknown results

最大的影响来自于完全不重试，因为我们有可能超时我们的请求（Account演员已经Thread.Sleep(\_random.Next(0,150)在其中，而AccountProxy期望在100毫秒内回复）：

RESULTS **for** 99.99% uptime, 0.01% chance of refusal, 0.01% of being busy **and** 0 retry attempts:

48.8% (488/1000) successful transfers

0.1% (1/1000) failures leaving a consistent system

0% (0/1000) failures leaving an inconsistent system

51.1% (511/1000) unknown results

总体而言，结果显示了重试我们的操作的重要性，以及需要使用能够让我们重试的幂等接收器的重要性。如果我们只是重试我们的操作，我们可以在非常容易出错的系统中获得非常好的结果。

这是一个人为场景。在现实世界中，您需要更精细的重试策略，允许远程服务从他们可能遇到的高需求或可能是短暂的故障中恢复 - 指数后退策略比立即重试更有用。能够通过使用审计日志从给定点恢复saga的能力也非常重要 - 如果远程服务停机了相当长的时间，您仍然可以在恢复时尝试传奇。