超越分布式事务的方法

**摘要 ABSTRACT**  
   在分布式事务方面已经有几十年的研究，例如2PC、Paxos这样的协议以及其它有代表性的实现方式，这些协议为应用程序员提供了一个全局序列化 (global serializability)编程外观。我个人职业生涯中的重要时期也极力提倡实现和使用这种提供全局序列化能力的平台。  
   近十年来的经历让我觉得这样的平台像马其诺防线。一般应用开发者不会使用分布式事务实现高伸缩性应用，当他们试图使用分布式事务时，项目创始人出于性能成 本和不稳定性否决这样的方案，这时自然选择闯入到项目中(译注：选用适合项目的方案)。另外使用不同技术构建的应用程序并不具备相同的事物担保机制，但都 能够满足业务需求。  
   这篇文章探索和命名一些在不能使用分布式事务的情况下，用于实现高伸缩、任务关键的应用程序的实践方法。文中讨论了对细粒度应用程序数据块的管理，随着应用的增长它们可能需要重新分区，另外也讨论了在这些可重新分区的数据块之间发送消息的设计模式。  
   发起这一讨论出于两方面考虑，希望能提高对新的设计模式的认识。首先我相信这些认识能够为那些正在处理高伸缩应用的人简化困难。其次通过了解这些模式，希望业界能够构建这样的平台简化这些大型应用的构建工作。  
  
**1. 介绍 INTRODUCTION**  
   先看一下这篇文章的目的，我在讨论中提出的一些假设，以及从这些假设得出的推论。虽然我对高可用性也非常感兴趣，但这篇文章将忽略这个方面而只专注于伸缩性，特别针对于不能使用大型分布式事务的场景。  
  
**目的 Goals**  
   这篇文章有三个主要目的：  
**•  讨论可伸缩应用 Discuss Scalable Applications**  
   构建大型系统的设计者都知道很多设计可伸缩系统的技巧，问题是对这些事务交互和可伸缩系统的问题、概念、总结等，没有命名没有清晰的理解，不正确的运用有 时反而反咬我们一口。这篇文章的目的之一就是发起一个讨论，加强对这些概念的了解，希望促成一系列的共识和一致的实现方案。  
   这篇文章尝试对多年来用于实现可伸缩系统的经验进行总结和形式化。  
**•  考虑无限伸缩应用 Think about Almost-Infinite Scaling of Applications**  
   为了勾画这个伸缩方面的讨论，文章提出了一个非正式的(informal)实现无限伸缩的思考方案(thought experiment)。我假设客户、可采购实体(purchasable entities)、订单、发货(shipments)、保健病人、纳税人、银行账户，以及应用程序中使用到的其它所有业务概念，他们的数量都随着时间高 速增长，但单个数据块并不会变得太大，我们只是获得了越来越多的数据块而已。计算机的哪种资源会首先过载真的不重要，需求的增长只是促使我们一开始使用少 量机器运行转而使用大量机器来运行。(译注：这就是后面会讨论的无限伸缩实现思想)  
   无限伸缩是一种随意、含糊、笼统的说法，在不清楚什么时候一台机器足够，如果不确定一台机器是否足够时该如何处理的情况下，使得我们的需求变得非常明确 (译注：意指线性伸缩是比较专业的说法，它听起来没有无限伸缩直观)。准确说我们希望对负载进行线性伸缩(包括数据和计算 scale almost linearly)。  
**•  描述可伸缩应用的几种通用模式 Describe a Few Common Patterns for Scalable Apps**  
   无限伸缩对业务逻辑有什么影响？我假设实现伸缩时我们在程序中需要使用一个叫做"实体"的新概念。实体在某个时刻位于单一机器上，应用程序同一时刻只能操作一个实体。无限伸缩的结论是这个编程概念必须暴露给应用逻辑的开发者。  
   提出和讨论这个目前还没有正式命名的概念，是希望我们达成一致的程序实现方式，以及对构建可伸缩系统涉及的问题达成一致的理解。  
   此外，实体的使用涉及连接实体的消息模式，应用程序开发者试图为业务问题构建可伸缩解决方案时必须处理消息分发的一致性问题，我们创建状态机来处理这些方面。  
  
**假设 Assumptions**  
   我们从三个假设开始，它们仅仅是没有被证明的假设，我们基于经验认为是它们是正确的。  
**•  应用程序的分层与伸缩无关性 Layers of the Application and Scale-Agnosticism**  
   我们假设每个可伸缩应用至少有两层，它们对伸缩机制的了解程度不一样，可能还有其它区别但与本次讨论无关。  
   应用程序的底层(lower layer)知道有很多机器组合起来使系统可以伸缩，除了其它工作外，它们管理上层(upper layer)代码到具体机器和位置的映射关系。底层是伸缩相关(scale-aware)的即它们了解这个映射关系。我们假设底层为上层提供了一个伸缩无 关的(scale-agnostic)编程抽象，使用它编写应用程序上层代码时无需考虑伸缩问题。遵守伸缩无关的��程抽象我们就能编写应用程序代码，而 不用担心将应用程序部署在前所未有的负载增长环境下带来更改。  
        
   随着时间推移这些应用程序底层可能发展为新的平台或中间件，简化伸缩无关应用程序的创建(类似过去CICS和TP-Monitor发展为简化批模式终端应用程序的创建)。  
   本次讨论的重点是这种新生的伸缩无关API的可能性。  
**•  事务序列化范围 Scopes of Transactional Serializability**  
   为分布式系统提供事务序列化进行了很多理论工作，例如2PC(两阶段提交)在某节点不可用时容易阻塞，而其它协议例如Paxos算法，在节点失败时不会阻塞。  
   我们说这些算法提供了全局事务序列化(global transactional serializability)，它们的目是为分布在一系列机器上的数据提供严格的原子更新操作，允许在单个序列化范围中跨越一系列机器进行更新。  
   我们试想一下不使用分布式事务会怎样。真实系统开发者以及目前我们看到的已部署系统很少跨越机器使用事务序列化，即使有也是在小范围被紧密联结起来用作群集的机器上。简而言之除了紧密联结起来可以看作一个群集的简单场景下，我们不会跨越机器使用事务。  
   相反我们使用多个分离的事务序列化范围(multiple disjoint scopes of transactional serializability)。把每台机器看作一个独立的事务序列化范围，每个数据项位于一台机器或一个群集中，原子事务可能涉及这个事务序列范围 (单一的机器或群集)中的任何数据，但无法跨越这些分离的事务序列化范围执行原子事务，这也正是它们分离的原因。  
**•  大部分应用使用"至少一次"的消息方式 Most Applications Use "At-Least-Once" Messaging**  
   TCP-IP对于像Unix形式的短周期(short-lived)进程非常好(译注: 连接建立过程有询问、应答)，但我们看一下一个需要处理消息，修改磁盘上一些持久化数据(SQL数据库或其它持久化仓库中)的应用开发者面临的问题。消息 接收后不会立即应答，需要等待数据库处理完毕。如果失败，这个过程需要重新开始，再次处理这个消息。  
   译注: 这里的场景指消息发送者判定处理失败，需要重新处理(retry)这个消息，当然消息接收者可以将处理失败的消息回馈给发送者，但例如超时、网络通信丢包偶尔中断等，将使得发送者无法鉴定。  
   问题产生的原因是消息分发与持久化数据的更新不是直接结合在一起，中间有应用程序的行为。虽然有可能将消息处理与持久化数据更新结合起来，但通常做不到。 缺乏这种结合在消息重复分发多次时会导致错误窗口的出现，因为其它资源可能造成消息偶然丢失(指"最多一次, at-most-once"的消息方式)，这种情况也很难处理。  
   消息探测(messaging plumbing)的这种行为导致的结果是应用程序必须能处理消息重试以及消息到达的无序性，这篇文章讨论了一些业务逻辑开发者在大型无限伸缩应用中必须处理这种负担时可以运用的应用模式。  
  
**有待证明的观点 Opinions to Be Justified**  
   撰写立场论文(position paper)的好处是可以表达疯狂的想法，下面这些在后面的章节会进一步详细讨论。  
**•  可伸缩应用使用唯一标志的"实体" Scalable Apps Use Uniquely Identified "Entities"**  
   这篇文章将会讨论到每个应用程序的上层代码必须操作单一一个我们称作"实体"(entity)的数据集合。对实体大小没有限制但它必须位于单一序列化范围中(例如一台机器或一个群集上)。  
   每个实体有一个唯一标识(identifier)或键值(key)，键值可以是任何形式，但必须能唯一的标识一个实体以及实体中包含的数据。  
        
   对实体的表现形式没有限制，它可以是SQL记录，XML文档，文件，文件系统中包含的数据，例如二进制数据块(blobs)，或其它任何对应用程序方便、适合的表现方式。一种可能的表现形式是SQL记录集(可能位于很多表中)，它们的主键都以实体键值开始。  
   实体代表了分离的数据系列(disjoint sets of data)，每个数据项(datum)只位于一个实体中，实体中的数据决不与其它实体的数据交叉(overlap)。  
   应用程序包括很多实体，例如一个"订单处理"程序会有很多订单，每个订单通过一个唯一订单ID标识，为了成为可伸缩的"订单处理"程序，单个订单数据必须与其它订单数据分离。  
**•  原子事务不能跨越实体 Atomic Transactions Cannot Span Entities**  
   下面我们会讨论为什么原子事务无法跨越实体。程序员必须明确每个事物只作用于单一实体中的数据，这个限制适用于同一个应用程序的不同实体以及不同应用程序中的实体。  
   从程序员的角度看，唯一标识的实体表明了序列化的范围，这个概念对伸缩设计的应用程序行为有巨大的影响。我们将探讨的一个推论是无限伸缩设计无法保证辅助索引(alternate indices)的事务一致性。     
**•  消息发送到实体 Messages Are Addressed to Entities**  
   绝大部分消息系统并不考虑数据的分区键值(partitioning key)，只是将消息发送到一个队列，然后由一个无状态进程进行处理。  
   通常的做法是在消息中包含一些数据，就是上面提到的实体键值，通知无状态的应用程序从哪里获取需要的数据，实体数据取自数据库或其它持久化仓库。  
   业界中出现了很多有意思的趋势。首先应用程序中实体系列(sets of entities)的大小已经增长到无法在一个数据仓库中存放，每个实体存放在一个数据仓库，但整个实体系列却不一定，无状态应用程序基于某些分区描述 (partitioning scheme)获取实体。其次分区信息被分离到应用程序的底层，特意与负责业务逻辑的应用程序上层分离。  
   这极大的促成了使用实体键值标识消息目的地，Unix风格的无状态进程以及应用程序的底层都是业务逻辑伸缩无关API��实现，上层伸缩无关的业务逻辑只是根据实体键值发送消息，实体键值则标识了持久化状态即实体。  
**•  实体管理每个协作伙伴状态(活动) Entities Manage Per-Partner State (“Activities”)**  
   伸缩无关的消息就是实体对实体的消息，发送方实体(反映了持久化状态，通过实体键值标识)将消息发送给另一个实体，接收方实体既包含上层(伸缩无关的)业务逻辑，也包含使用实体键值存取，反映了其状态的持久化数据。  
   前面假设消息至少分发一次，这意味着接收方实体必须能够忽略掉多余无效的消息。实际上消息分为两种类型，会影响接收方实体状态的和不会影响的。不会给实体状态带来变化的消息很容易处理，它们一般是幂等的(idempotent)。  
   为了确保幂等性(idempotence，例如确保重试的消息不会带来副作用)，通常设计实体接收方让它们记住已经处理过的消息。这样做之后重复的消息一般产生一个新的响应(即回发的消息)，它与早先被处理过的消息结果一样。  
   根据收到的消息创建的状态基于每个协作伙伴进行封装，这里的关键思想是按照协作伙伴组织状态，协作伙伴也是一个实体。  
   我们使���"活动"这个术语表示状态，它在两方关系(two-party relationship)之间管理每一方的消息。每个"活动"位于一个实体中，实体对每一个会向他们发送消息的协作伙伴都有一个"活动"。  
   除了管理消息混乱外，活动还管理松散关联(loosely-coupled)的协议。在这个不能使用原子事务的地方通常使用尝试性操作协商结果，这在实体之间进行由活动进行管理。  
   这篇文章并不假定活动能够解决各种已知的难题，以达成在工作流的讨论中描述的那样详细的协议。但我们指出无限伸缩会令人惊讶的带来细粒度工作流风格的解决 方案，其中参与者是实体，每个实体使用涉及其它实体的特定知识管理自己的工作流，这个在实体内部维护的协作双方的知识正是我们称作活动的东西。  
   关于活动的示例有时候显得很费解。订单应用会向发货应用发送消息，消息中包含发货ID和订单ID，消息类型将触发发货应用中的状态发生改变，将相关订单记 录为等待发货的状态。通常实现者不会设计消息重试，直到出现bug，少数情况下应用程序设计者考虑和打算设计活动。  
   文章接下来的部分将深入的分析这些假设，提出观点，以及对这些观点的阐述。  
  
**2. 实体 ENTITIES**  
   本节深层次地探讨实体的本质。首先我们需要保证原子事务位于单一实体中，接下来讨论使用唯一键值存取实体，以及在重分区时怎样让应用程序的底层(伸缩相关 的部分)重新定位实体，然后讨论在单一原子事务中可以存取哪些东西，最后探讨无限伸缩中一些关于辅助索引的隐含结论。  
  
**分离的序列化范围 Disjoint Scopes of Serializability**  
   实体定义为拥有一个唯一键值的数据集合，他们位于单一的序列化范围中。因为实体位于单一的序列化范围因此我们确定在一个实体中总是可以使用原子事务。  
   正是这方面原因，我们使用"实体"而不是"对象"这个名字。对象之间可以共享事务范围，但在实体之间永远不会，因为重分区可能会将它们放到不同机器上。  
  
**唯一标识的实体 Uniquely Keyed Entities**  
   应用程序的上层代码通常围绕具有唯一键值的数据集合进行设计，我们可以看到客户ID、社会安全号码、产品库存单位(SKUs，译注：不是指度量单位，而是 库存系统中的产品唯一标识符)，以及应用程序中其它各种随处可见的唯一标志符，它们作为键值用来查找应用程序处理的数据。这是通常的情况，实际中我们发现 分离的序列化范围(例如"实体")总是使用唯一的键值进行标识。  
  
**重分区和实体 Repartitioning and Entities**  
   我们的一个假设是不断增长的上层是伸缩无关的，当伸缩需求改变时由底层决定怎样布署。这意味着布署过程中实体的位置可能发生改变，应用程序上层不能对实体位置做任何假设，否则就不是伸缩无关了。  
        
  
**原子事务和实体 Atomic Transactions and Entities**  
   在可伸缩系统中不能跨越不同实体使用更新事务。每个实体拥有一个唯一键值，可以方便的将它们放入一个序列化范围中，怎样才能确认多个独立的实体一定位于相 同的序列化范围中(因此可以自动完成更新事务)？只有这些实体都统一具有相同键值时才可以，这时它们已经是一个实体了！  
   如果对实体键值使用哈希进行分区，没有什么可以表明具有不同键值的实体会落在相同的哈希桶中(译注：即意味着位于同一个序列化范围)。如果对实体键值使用 键值范围进行分区，绝大部分时候相同键值将位于同一台机器上，但很不幸有时相邻键值会位于不同机器。对键值范围的分区使用相邻键值进行原子性测试的简单测 试用例，在测试布署环境下可能通过，但以后重新布署时在不同的序列化范围中移动实体，会遇到潜伏的bug，因为(跨越序列化范围的，译注)更新不再具备原 子性。永远也不要指望不同的实体键值会位于同一个地方。  
   简而言之，应用程序底层将确保每个实体键值(和实体)位于单一的机器(或群集)上，而不同实体则可能分布在任何地方。  
   伸缩无关的设计必须具有实体概念，作为原子性的边界。把对实体存在(译注：所处位置)的认知作为一个设计抽象，使用实体键值，以及明确的声明在跨越实体时缺乏原子性，这些是为应用程序提供伸缩无关的上层的关键所在。  
   现在行业中的高伸缩性应用无疑都是这样做的，我们只是对实体概念没有一个正式的名字而已。对上层应用程序而言，它必须明确实体是序列化的范围，更进一步的假设在布署发生改变时会被打破。  
  
**考虑辅助索引 Considering Alternate Indices**  
   我们经常使用多个键值或索引查找数据，例如有时使用社会安全号引用客户，有时使用信用卡号，有时使用街道地址。极端情况下这些索引无法位于同一台机器或一 个大型群集中，此时一个客户相关的数据(译注：例如索引数据)无法保证位于单一的序列化范围中！实体本身位于单一的序列化范围中，麻烦的是那些构成辅助索 引的数据副本必须考虑位于不同的序列化范围！  
   考虑辅助索引位于相同序列化范围中这样一种假设。在需要使用无限伸缩而实体系列(set of entities)分布在大量机器上时，主索引和辅助索引数据必须位于相同的序列化范围中，怎样确保这一点？唯一的方法就是使用主索引查找辅助索引(译 注：原文应当指扫描主索引来匹配辅助索引的方案，这种方案并没有建立辅助索引结构，通过动态查找实现)，这使得它们可以位于相同的序列化范围中！当没有主 索引键值但必须在整个序列化范围中进行查找时，每个辅助索引搜索必须检查无限数量的序列化范围以匹配辅助索引键值！这始终是不可行的。  
        
   唯一的替代方案是使用两步搜索，首先查找辅助索引得到实体键值，然后使用实体键值存取实体(译注：这种方案需要建立辅助索引结构，使用辅助索引可以查找到 主索引的键值)。这跟关系型数据库中的辅助索引使用两个步骤存取纪录非常相似，但无限伸缩的前提是无法保证两个索引(主索引和辅助索引)位于相同的序列化 范围中。  
   伸缩无关的应用程序将无法自动更新实体和它的辅助索引！上层伸缩无关的应用必须设计为能够处理这样的情况：辅助索引可能与使用主索引(例如实体键值)存取的实体失去同步的情况。   以前可以自动维护的辅助索引，现在必须由应用程序手动维护，使用异步消息进行更新的工作流也都是这样，需要无限伸缩的应用程序自己处理。以前使用辅助索引 读取数据，现在就必须清楚它们可能与实体的主要呈现窗口失去了同步，因此基于辅助索引实现的功能现在变得麻烦了。这就是大型系统残酷世界中的真实生活。  
  
**3. 跨越实体的消息通讯 MESSAGING ACROSS ENTITIES**  
   这一节中我们讨论使用消息连接不同实体的方法，包括事务和消息，看一下消息分发的语义，以及讨论一下对实体位置重新分区给消息分发带来的影响。  
  
**跨越实体的消息通讯 Messages to Communicate across Entities**  
   如果不能在同一个事务中跨越两个实体更新数据，就需要一种机制在不同事物中来完成，我们使用消息连接这些实体。  
  
**异步发送事务 Asynchronous with Respect to Sending Transactions**  
   消息是跨越实体的，待发送消息位于一个实体中，消息终端是另一个实体。根据实体定义，我们必须清楚它们无法自动完成(译注：跨越实体的事务)。  
   应用开发者通过发送消息的方式使用事务可能是异常复杂的，将消息发送出去，然后事务可能被中断，你可能对这不以为然但它确实可能发生。由于这些原因，必须迎难而上使事务消息入队。  
        
   如果发送事务(sending transactions)提交之后无法立即接收到目的地的回馈，我们看到相对于发送事务(sending transactions)消息是异步的。实体在事务中会转化到新的状态，而消息是触发器，它来自于一个事物(transaction)到达另一个实体并 引发新的事务。  
  
**确定消息终端 Naming the Destination of Messages**  
   在开发应用程序伸缩无关的部分时，一个实体需要向另一个实体发送消息，伸缩无关的代码并不知道目标实体的位置，即实体键值。这由应用程序伸缩相关的��分来处理，它将实体键值和实体位置关联起来。  
  
**重分区和消息分发 Repartitioning and Message Delivery**  
   应用程序伸缩无关的部分发送消息时，底层伸缩相关部分捕获到目标地址，将消息至少分发一次。  
   系统伸缩时会移动实体，这通常叫做重新分区。实体数据的位置即消息的目的地可能发生变化，有时消息仍将发送到旧地址，只是发现该死的实体已经被移到其它地方，这时消息需要路由。  
   移动实体有时会中断发送方和目的地之间先进先出队列的通路，这时消息被重发(retry)，后来的消息会比先前的更早到达，世界变得更混乱。  
   由于这些原因，我们看到伸缩无关的应用程序对应用程序可见的消息支持幂等处理，这也意味着重新订阅(reorder)消息分发。  
  
**4. 实体、SOA和对象 ENTITIES, SOA, AND OBJECTS**  
   这一节将本文的观点和面向对象、面向服务观点进行对照。  
  
**实体和对象实例 Entities and Object Instances**  
   有人可能会问："实��和对象实例有什么区别？"，答案不象是非黑白这样清晰。对象有很多形式，有些是实体有些不是，对象成为实体有两个重要前提。  
   首先对象封装的数据必须严格与其它数据分离，其次分离的数据永远不能和其它数据一起自动更新。  
   一些对象系统对数据库数据采用多重封装(ambiguous encapsulation)，从某些方面来说这不见得脆弱也不值得提倡，但这些对象不是本文定义的实体。有时会使用物化视图(materialized views)和辅助索引，当系统需要伸缩而你的对象又不是实体时就不会再使用它们了。  
   很多对象系统允许事务范围跨越对象，这种开发便利性避免了很多这篇文章中提到过的难题，不幸的是它不适用于无限伸缩，除非将这些事务耦合的对象布署在一起。给它们分配一个通用的键值可以确保它们布署在一起，以实现两个事务耦合的对象成为同一实体的一部分！  
   对象非常好但它们属于不同的概念。  
  
**消息与方法的比较 Messages versus Methods**  
   方法调用通常与调用线程是同步的，因此也与调用对象的事务同步。然而被调用对象与调用对象并不一定能自动结合(译注：指跨越序列化范围的事务无法自动结 合)，普通的方法调用并不记录处理的消息，对被调用消息也不遵守至少一次这一信条。一些系统将消息发送封装成方法调用，我认为这是消息而不是方法了。  
   我们并不明确区分列集(Marshaling)和绑定(Binding)，虽然他们通常用于区分消息和方法调用，我们只是简单的指出在事务边界上需要使用异步通讯，这在方法调用中是不常见的。  
  
**实体和面向服务架构 Entities and Service Oriented Architectures**  
   本文讨论的内容都支持SOA，绝大部分SOA实现(implementations)在服务之间采用独立的事务范围。  
   这里对SOA的主要增强(enhancement)是每个服务本身可能需要处理无限伸缩，文章的内容指示了怎样实现，这些内容适用于SOA服务间的设计，也适用于那些设计为可独立伸缩的单个服务。  
  
**5. 活动：处理混乱的消息 ACTIVITIES: COPING WITH MESSY MESSAGES**  
   这一节讨论攻克消息重试(retry)和重新订阅(reorder)这些难题，我们引入了活动这一概念作为必要的本地信息管理协作伙伴实体的关系(relationship)。  
  
**重试和幂等性 Retries and Idempotence**  
   因为之前发送过的任何消息可能被分发多次，在应用中我们需要一种机制处理重复的消息。尽管可以构建一个支持消除重复消息的底层，在无限伸缩环境中这个底层 支持需要了解实体，发送给实体的消息在重分区移动实体时必须跟随转移。实际中对这种情况的底层管理很少使用，因此消息可能被多次分发。  
   通常应用程序伸缩无关(上层)部分必须实现一些机制，确保接收的消息是幂等的。这对问题的本质不是必须的，当然也可以采用在应用程序伸缩相关部分构建消除 重复的机制来解决。不过目前还没有这方面的应用，因此我们讨论可怜的伸缩无关应用开发者必须采用的方式(译注：确保幂等性)。  
  
**定义本质行为的幂等性 Defining Idempotence of Substantive Behavior**  
   如果后续对消息处理的重复执行不会给实体带来本质变化，这个消息处理就是幂等的。这不是一个严谨的定义，关于什么才是本质变化留待应用程序确定。  
   如果消息不会改变调用实体而只是读取信息，这个消息处理是幂等的。即使写入了一条描述本次读取的日志记录我们也认为是幂等的，因为日志记录不会对实体行为造成本质影响。本质的定义是应用程序相关的。  
  
**自然幂等性 Natural Idempotence**  
   消息不会造成本质副作用是实现幂等性的关键，有些消息任何时候都不会造成本质影响，他们就是自然幂等的。  
   只从实体读取数据的消息是自然幂等的。如果消息处理确实改变了实体但并不带来本质影响，那也是自然幂等的。  
   接下来是更麻烦的，有些消息带来了本质变化因此他们不是自然幂等的，而应用程序必须引入一些机制确保这些消息也是幂等的，这意味着采用某种方式记录已处理过的消息，以使后续重复的调用不会造成本质变化。  
   这就是接下来我们要讨论的非自然幂等的消息处理。  
  
**将消息记录为状态 Remembering Messages as State**  
   为了确保非自然幂等消息的幂等处理，实体必须记住哪些消息已经处理过了，这就是状态，状态随着消息处理不断记录下来。  
   除了记录消息已经处理过之外，如果消息需要回复则必须返回相同的回复内容，因为我们无法确定原发送者是否已经收到了这个回复。  
  
**活动：管理每个协作伙伴的状态 Activities: Managing State for Each Partner**  
   为了跟踪关系和收到的消息，伸缩无关应用中的每个实体必须采���某种方式记录协作伙伴的状态信息，并且必须针对每个协作伙伴分别记录，我们将这个状态命 名为活动。如果一个实体与其它多个实体交互，它就会有多个活动。活动跟踪实体与每个协作伙伴的关系(relationship)。  
        
   每个实体可能包含一系列活动，某些数据可能需要跨越多个活动。  
   考虑处理一个包含很多采购项目的订单，为每个采购项的发货预留库存都将是独立的活动，库房中对应订单和每个采购项都会维护一个独立的实体，无法假设事务可以跨越这些实体。  
   订单中的每个库存项目将被独立维护，因此消息协议也必须独立管理，订单实体中的每个库存项目数据都是一个活动。尽管不是这样称呼，这种模式大量的存在于高伸缩应用中。   在无限伸缩应用中你必须非常清楚这些关系，因为无法简单的看一下就描述出是怎样关联的。任何东西必须有效地使用一个双方关系网结合起来，结合元素 (knitting)是实体键值。因为协作伙伴距离遥远，因此当它拜访时你必须将了解到的信息当作全新的知识有效的管理起来。这个让你能够了解远方协作伙 伴的本地信息称作一个活动。  
        
  
**通过活动确保最多接受一次消息 Ensuring At-Most-Once Acceptance via Activities**  
   非自然幂等消息的处理必须确保最多处理一次(例如消息的本质影响最多只会产生一次)。为了实现这个目的必须有一些唯一性机制，确保消息不会重复处理多次。  
   实体必须将等待处理的消息转换 持久化记录到状态中，以使重复的消息处理不会造成本质影响。  
   通常实体基于每个协作伙伴使用活动实现这种状态管理，这一点很重要，因为有时实体会有很多不同的协作伙伴并且使用特定形式的消息跟每个协作伙伴进行交互。  
   针对各个协作伙伴有效的使用状态集合，程序员能够专注于协作伙伴的交互。  
   结论是只需关注各个协作伙伴的信息时很容易构建可伸缩应用，例如在实现了幂等消息处理的平台上。  
  
**6. 活动： ACTIVITIES: COPING WITHOUT ATOMICITY**  
   这一节讲述在没有分布式事务的情况下可伸缩系统怎样使用一些武断的决策方式。  
   管理分布式协议是一项艰巨的任务，这是本节的重点。另外由于是在无限伸缩这样一个环境中，必须采用以每个协作伙伴关系为中心这样一种细粒度设计来解决不确定性，这些数据在实体内部使用活动这一概念进行管理。  
  
**目的地的不确定性 Uncertainty at a Distance**  
   没有分布式事务意味着跨越不同实体的决策必须考虑不确定性，目前跨越分布式系统的决策仍无法规避不确定性这一问题。使用分布式事务时，这些不确定性发生在数据锁上，由事务管理器管理。  
   不能使用分布式事务的系统必须在业务逻辑中管理不确定性，使用业务语义(business semantics)而不是记录锁(record lock)控制不确定性的影响，这就是工作流了。没什么玄乎的，只是由于不能使用分布式事务而必须采用工作流而已。  
   这些因素使得我们使用实体和消息，使我们明白如果伸缩无关的应用需要跨越多个实体达成协议，就必须自己使用工作流管理不确定性。  
   参考普通商务活动之间的交涉方式，商务合同会包含承诺时间、终止条款、预留资源，以及其他更多内容。同样不确定性语义(译注：指代码中解决不确定性问题的机制)也穿插在业务功能行为中，这比简单使用分布式事务更难实现，但现实世界就是这样。  
   再次声明，这里只是简单讨论工作流。  
**活动与不确定性管理 Activities and the Management of Uncertainty**  
   实体在与其它实体交互时可能出现不确定性，这种不确定性必须基于每个协作伙伴进行管理，即在具体的协作伙伴活动状态中实现。  
   大部分时候不确定性缘于实体间的关系，但有必要按照协作伙伴进行跟踪，在每个协作伙伴进入新的状态时，活动将跟踪记录下来。  
   如果订单系统从库房中预留库存，库房会分派库存但并不知道它是否会被使用(译注：订单是否真的会使用这些库存？假如订单被取消了，订单因变更数量减少了？)，这就是遇到了不确定性。以后库房会知道这个答案，这样不确定性问题就解决了。  
   库房存货管理器必须使用一些条目为每个订单维护关联数据，在将条目和订单关联时，按照条目进行组织，每个条目包含与该条目相关的未完成订单信息，条目中的每个活动(每个订单对应一个活动)管理相关订单的不确定性。  
**处理尝试性业务操作 Performing Tentative Business Operations**  
   为了在实体间达成协议，实体必须能够让其它实体来处理不确定性，这通过发送一个确认消息请求实现，同时也需要能应对取消的情况，这就叫做尝试性操作。每个尝试性操作最终会被确认或取消。  
   实体允许尝试性操作时，它允许其它实体决定操作结果，这样实体遇到不确定性时为纠纷的处理带来了改善，取消或确认消息的到达意味着不确定性的减少。以前不断增加或减少的不确定性问题解决了，新的问题又会来到你身边，这在生活中很正常(译注：下面接着讨论新的问题)。  
   同样这只是工作流，但它是基于实体精细设计的工作流。  
  
**不确定性和无限伸缩 Uncertainty and Almost-Infinite Scaling**  
   这种无限伸缩方案有意思的方面是围绕两方协议(two-party agreement)管理不确定性。经常会存在多个两方协议，我们还是使用实体键值作为连接器，使用活动跟踪远方协作伙伴的当前状态，这样这些两方协议就被连结成一个细粒度的两方协议网。  
   考虑有第三方委托公司参与的购房业务。买方、卖方、担保公司以及这次交易的所有其它参与方都会与委托公司签订信任协议。  
   当你签订购房协议时你并不知道最终处理结果，你清楚在委托公司结案前你都无法确定，唯一的决策方是委托公司。  
   这就是一个两方关系集的枢纽中心，用于在大量的参与方之间达成一致而不需要使用分布式事务。  无限伸缩中考虑两方关系是很有意思的事情，基于两方关系构建尝试性/取消/确认操作框架(就像传统的工作流)，我们看到分布式协议的达成原理。就像委托公司一样，很多实体可以通过某个组织参与到一个协议中。  
   因为是两方关系，活动的简单意义就是"我保存的那个合作伙伴的资料"，这也是管理大型系统的基础。就算数据是保存在实体中，你并不知道它具体位于哪儿而必须假定是在很远的地方，这样就能够采用伸缩无关的方式开发。  
   真实世界中无限伸缩应用程序喜欢享受两阶段提交或其他算法实现的全局序列化范围带来的便利性，不幸的是它将导致可用性上不可接受的压力(译注：性能负 载)，因此为伸缩无关应用的开发者提供尝试性方法管理不确定性，象预留库存、信用额度的分配以及其它应用的相关��念必须这样处理。  
  
**7. 结论 CONCLUSIONS**  
   计算机产业在发展，应用程序的一个发展趋势是需要使用伸缩解决大小不再适合一台机器或一系列紧密结合的机器的情况。我们经常看到首先会出现应用于某个应用程序的特定解决方案，然后得到一些通用的模式，基于这些通用模式构建工具集使得应用逻辑的构建更简便。  
   20世纪70年代很多大型伸缩应用程序在提供业务解决方案时，挣扎在在线终端多路复用处理的困难之中，后来涌现出一些终端控制模式，一些高端应用发展为 TP-Monitor，后来TP-Monitor的重写中也一直沿用了这些模式。这些平台使业务逻辑开发者专著在他们擅长的领域：业务逻辑的开发。  
   今天我们看到新的设计压力被强加给那些只是想解决业务问题的程序员，现实将他们带入无限伸缩的世界，迫使他们做大量与手头真正业务无关的设计问题。  
   很不幸程序员在解决电子商务、供应链管理、财务、保健应用等业务目标时，需要不断的思考不使用分布式事务的伸缩问题，因为分布式事务的脆弱和低性能他们必须这样做。  
   我们又处在了这样一个时刻，已经出现了构建可伸缩应用的模式但还没有一致的应用。这篇文章讨论了这些新生的模式能够更一致的运用于手头无限伸缩应用的开 发，并且在未来几年中，我们可能会看到为这些应用提供自动化管理的中间件和平台的开发，采用标准的开发方式为应用程序结束伸缩难题，这与20世纪70年代 出现的TP-Monitor非常相似。  
   这篇文章中我们讨论、命名了一些新出现在高伸缩应用中的模式  
•  实体是命名的(keyed-索引的)数据集合，可以在实体内部但无法跨越实体自动更新。  
•  活动包含了实体的状态集合，为单个协作伙伴实体管理消息关系。  
   在实体的活动中使用讨论了多年的工作流进行决策，当你看一下无限伸缩(的解决方案，译注)你会惊讶的发现它天生就是一种细粒度的工作流。  
   我们已经讨论目前很多应用程序隐式的使用实体和活动这样的设计，只是没有标准化没有一致的运用而已。通过讨论和一致的运用这些模式，可以构建更好的高伸缩应用，作为一个产业，我们可以更进一步构建解决方案，使业务逻辑开发者专注在业务问题而不是伸缩问题上。

Tag标签: [Scalability](http://www.cntxk.com/images/spd/Scalability/),[Design](http://www.cntxk.com/images/spd/Design/),[Papers](http://www.cntxk.com/images/spd/Papers/)