分布式交易之外的生活：叛者的观点

位置纸

帕特赫兰

亚马逊河通信
第五大道南705号
西雅图，华盛顿州98104
美利坚合众国
PHelland@Amazon.com

本文中所表达的立场是个人观点，并没有以任何方式反映我的雇主亚马逊的立场。com.

摘要

几十年的工作已经投入到分布式交易领域，包括协议，如2PC，Paxos，和各种法定人数的方法。这些协议为应用程序程序员提供了一个全局可序列化的外观。就我个人而言，我投资了我职业生涯的一部分，大力倡导实现和使用提供全球可序列化保证的平台。

我在过去十年里的经验让我把这些平台比作马其诺线[1].一般来说，应用程序开发人员根本不需要假设分布式事务实现大型可伸缩应用程序。当他们尝试使用分布式事务时，项目的创始人，因为性能成本和脆弱性使它们不切实际。自然选择开始了。

相反，应用程序使用不同的技术构建，这些技术不提供相同的事务保证，但仍然满足其业务的需求。

本文探讨并列举了在拒绝分布式事务的世界中实现大规模关键任务应用程序中使用的一些实用方法。我们将讨论细粒度的应用程序数据块的管理，这些数据可以随着应用程序时间的推移而重新分区。我们还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。

开始这个讨论的原因是为了提高人们对新模式的认识，原因有二。首先，我相信这种意识可以减轻人们手工制作非常大的可扩展应用程序的挑战。其次，通过观察这些模式，希望该行业能够努力创建平台，使构建这些非常大的应用程序更容易。

[1] 马其诺线是一座巨大的堡垒，贯穿着法德边境，在第一次世界大战和第二次世界大战之间花费了大量的资金建造。它成功地阻止了德国军队直接越过法德边境。1940年，德国人入侵比利时，很快就绕过了它。

本文根据知识共享许可协议发布的(http://creativecommons.org/licenses/by/2.5/).
您可以复制、分发、展示和执行作品，制作衍生作品并将作品进行商业使用，但您必须将作品归功于作者和CIDR 2007。

3rd两年一次的创新数据系统研究会议（CIDR）1月710日，美国加利福尼亚州阿西洛玛。

1.介绍

让我们来看看本文的一些目标，我为这次讨论所做的一些假设，然后是从这些假设中得出的一些观点。虽然我对高可用性非常感兴趣，但本文将忽略这个问题，只关注可伸缩性。特别是，我们关注假设我们不能有大规模分布式事务的影响。

目标

本文有三个主要目标：

- 讨论 可伸缩 应用程序
  许多构建大型系统的应用程序设计者都隐含地理解了可伸缩系统设计的许多需求。

问题是，事务和可伸缩系统交互的问题、概念和抽象没有名称，也不能清晰地理解。当它们被使用时，它们会不一致，有时会回来咬我们。本文的一个目标是启动一个讨论，可以提高对这些概念的认识，并希望推动一组共同的术语和一个商定的方法。

本文试图命名和形式化一些抽象来实现可伸缩系统。

- 思考 关于 几乎无限的 缩放比例 的 应用程序

为了构建关于尺度的讨论，本文提出了一个关于几乎无限尺度影响的非正式思维实验。我假设客户、可购买实体、订单、发货、医疗保健患者、纳税人、银行账户以及由应用程序操纵的所有其他业务概念的数量随着时间的推移会显著增加。通常情况下，个体事物不会变得更大；我们只是得到越来越多。不管计算机上的什么资源首先饱和，需求的增长将推动我们传播以前运行在小型机器上的东西，以便在更大的机器上运行。[2]

几乎无限缩放是一种松散的、不精确的、故意非定形的方法，以激发需要非常清楚我们在什么时候知道什么东西适合在一台机器上，如果我们不能确保它适合在一台机器上该做什么。此外，我们希望几乎是线性的缩放[3]与负载（包括数据和计算）。

- 描述 a 很少的 常见的 榜样 为了 可伸缩 附录

几乎无限的扩展对业务逻辑的影响是什么？我断言，缩放意味着在编写程序时使用一个名为“实体”的新抽象。一个实体同时生活在一台机器上，而应用程序一次只能操作一个实体。几乎无限缩放的一个结果是，这种编程抽象必须暴露于
业务逻辑的开发人员。

通过命名和讨论这个尚未命名的概念，希望我们能够就一致的编程方法和对构建可伸缩系统所涉及的问题达成一致的理解。此外，实体的使用还会影响到用于连接实体的消息传递模式。这些导致了国家机器的创建来应对

[2] 需要明确的是，这在概念上假设有成千上万或数十万台机器。数量太多了，无法让它们表现得像一台“大”机器。

[3] 在N log N上缩放会真的很好。

当无辜的应用程序开发人员试图为业务问题构建可扩展的解决方案时，消息传递的不一致性强加给了他们。

假设

让我们从三个被断言而又不合理的假设开始。根据经验，我们只是假设这些都是正确的。

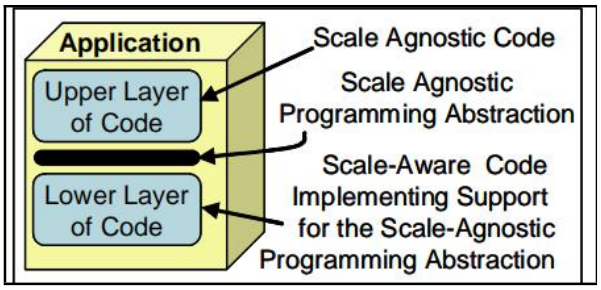
- 层 的 那 应用程序 和 量表不确定性

让我们从假定每个可伸缩应用程序中的两层（至少）有两层开始。这些层对缩放的感知也有所不同。他们可能有其他的差异，但这与这次讨论无关。

较低的 应用程序的层了解添加更多的计算机来增加系统的规模。除了其他工作之外，它还管理着上层代码到物理机器及其位置的映射。较低的层是标度感知的，因为它理解这个映射。我们假设较低的层提供了一个伸缩性的 *规划 抽象到上层*[4].

使用这种与尺度无关的编程抽象，上面的_编写应用程序代码层而不担心缩放问题。通过坚持可伸缩的编程抽象，我们可以编写应用程序代码，不担心在应用程序针对不断增加的负载部署时发生的更改。

随着时间的推移，这些应用程序的下层可能会



发展成为新的平台或中间件，简化了规模无关应用程序的创建（类似于过去CICS和其他TPMononitors发展为简化块模式终端应用程序创建的场景）。

本讨论的重点是这些新生的与规模无关的api所带来的可能性。

- 领域 的 事务性 序列化能力

关于跨分布式系统提供事务序列化的概念，已经做了许多学术工作。这包括2个PC（两阶段提交），它可以很容易地在节点不可用时阻塞，以及其他在面对节点故障时不阻塞的协议，如Paxos算法。

[4] 谷歌的MapReduce是一个与规模无关的编程抽象的例子。

让我们将这些算法描述为一种能够提供全局算法的算法 *交易的 可串行性*[5].他们的目标是允许跨一组机器的数据进行任意的原子更新。这些算法允许更新存在于跨这一组机器的可序列化范围的单一范围内。

我们会考虑当你简单的 don't 做 这正如我们今天看到的真实的系统开发人员和真实的系统，甚至很少尝试跨机器实现事务序列化，或者，如果他们做到了，它是在少数紧密连接的作为集群的机器中。简单地说，我们不是跨机器进行事务，除了在简单的情况下，有一个看起来像一台机器的紧密集群。

*相反，我们假设有多个 脱节 领域 交易的 可串行性*认为每台计算机都是一个单独的事务序列化范围[6].

每个数据项都位于一台计算机或集群中[7].原子事务可以包括驻留在事务序列化的单一范围内的任何数据。e.在单个计算机或集群中)。您不能跨这些事务性可序列化性的不相交范围来执行原子事务。这就是为什么让他们不相交的原因！

- 最 应用程序 使用 "至少一次" 消息传递

  如果您是一个短暂的unix风格的进程，那么TCP-IP是很棒的。但是让我们考虑应用程序开发人员所面临的困境，他的工作是处理消息并修改磁盘上持久表示的一些数据（在SQL数据库或其他持久存储中）。该消息已被消耗掉，但尚未被确认。将更新数据库，然后确认该消息。在失败时，将重新启动，并再次处理消息。

  这种困境源于这样一个事实，即除了通过应用程序操作之外，消息传递并没有直接耦合到持久数据的更新上。虽然可以将消息的消耗与持久数据的更新相结合，但这通常不可用。没有这种耦合会导致消息被多次传递的失败窗口。信息管道

[5] 我故意将严格的串行化性和较弱的锁定模式合并在一起。问题是参与对应用程序可见的事务的数据的范围。

[6] 这并不是要阻止在集群中运行的一小部分计算机像它们是一台机器一样工作。这是为了正式声明，我们假设有许多计算机和我们必须考虑不能被原子地承诺的工作的可能性。

[7] 这排除了高可用性的复制，这不会改变分支序列化的不相交范围的假设。

这样做是因为它唯一的其他办法是偶尔丢失消息（"最多一次"的消息传递），这样处理起来就更加麻烦了[8].

来自消息传递管道的这种行为的一个结果是，应用程序必须允许消息重试和某些消息的无序到达。本文考虑了当业务逻辑程序员必须在几乎无限大的应用程序中处理这一负担时所产生的应用程序模式。

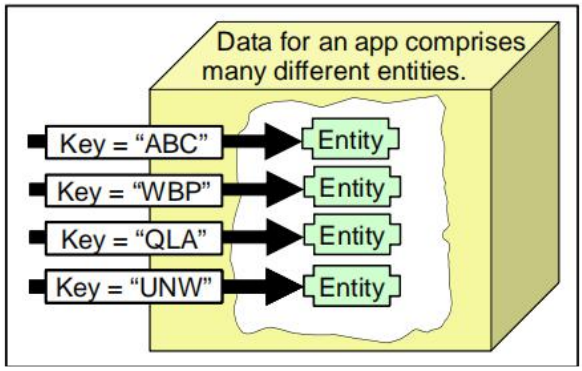
有理由的意见

写求职意见书的好处是你可以表达疯狂的观点。以下是我们将在这篇立场论文的语料库中争论的一些问题[9]：

- 可伸缩 附录 使用 独一无二 已确定 "实体"

  本文将论证，每个应用程序的上层代码必须操作我们称之为实体的单个数据集合。对实体的大小没有任何限制，除了它必须生活在一个可序列化的单一范围内(i。e.一台机器或集群）。

  每个实体都有一个唯一的标识符或密钥。整个键可以具有任何形状、形式或味道，但它以某种方式唯一地标识了一个实体和该实体中包含的数据。



  对实体的表示没有任何约束。它可以被存储为SQL记录、XML文档、文件、包含在文件系统中的数据、blob，或任何其他方便或适合应用程序需要的东西。一种可能的表示方式是作为一个SQL记录的集合（可能跨多个表），它们的主键以实体键开始。

[8] 我非常喜欢"完全一次有序"的消息传递，但要为持久的数据提供它，就需要一个类似于TCP连接的长期编程抽象。这里的断言是，构建可伸缩应用程序的程序员很少可用这些设施。因此，我们正在考虑处理"至少一次"的案例。

[9] 请注意，我们将更详细地讨论这些主题。

实体表示不相交的数据集。每个数据恰好存在在一个实体中。一个实体的数据永远不会与另一个实体的数据重叠。

一个应用程序由许多实体组成。例如，一个“订单处理”应用程序封装了许多订单。每个订单都由一个唯一的订单id来标识。要成为一个可伸缩的“订单处理”应用程序，一个订单的数据必须与其他订单的数据不相交。

- 原子的 交易记录 不能 张成子空间 实体

  下面我们将讨论为什么我们会得出原子事务不能跨越实体的结论。程序员必须始终为每个事务坚持使用包含在单个实体中的数据。此限制适用于同一应用程序中的实体和不同应用程序中的实体。

  从程序员的角度来看*唯一标识 实体 是 那 范围 的 可串行性*这个概念对设计为伸缩的应用程序的行为有强大的影响。我们将探讨的一个含义是，当设计为几乎无限的尺度时，替代指数不能在交易上保持一致。

- 消息 是 已解决 向 实体

  大多数消息传递系统不考虑数据的分区键，而是针对一个队列，然后由一个无状态进程消耗。

  标准实践是在消息中包含一些数据，通知无状态应用程序代码从哪里获取所需的数据。这是上面描述的实体键。实体的数据由应用程序从某个数据库或其他持久存储中获取。

  这个行业已经发生了一些有趣的趋势。首先，单个应用程序中的实体集的大小越来越大于单个数据存储中可以容纳的大小。每个单独的实体都适合一个存储，但它们的集合都适合。越来越多的无状态应用程序是基于某些分区方案进行路由来获取实体。其次，获取和分区方案被分离到应用程序的底层，并有意地与负责业务逻辑的应用程序的上层隔离。

  这有效地推动了作为实体键的消息目的地。无状态的unix风格的流程和应用程序的底层都只是业务逻辑的与规模无关的API实现的一部分。上层与规模无关的业务逻辑简单地将消息传递到实体键，该键标识称为实体的持久状态。

- 实体 管理 每个合作伙伴 状态 （“活动”）

  与规模无关的消息传递是一种有效的实体到实体的消息传递。发送实体（以其持久状态显示并由其实体键标识）发送发送给另一个实体的消息。接收实体包括上层（伸缩）业务逻辑和表示实体键存储和访问其状态的持久数据。

  回想一下消息“至少传递一次”的假设。这意味着收件人实体必须准备好处于其持久状态，才能被必须忽略的冗余消息攻击。实际上，消息分为两类：影响接收方实体状态的和不影响的。不会导致处理实体发生更改的消息很容易发生。它们通常是幂等的。正是那些对接受者做出改变的人给设计带来了挑战。

  以确保幂等(i。e.保证对重试消息的处理是无害的)，收件人实体通常被设计为记住消息已被处理。一旦出现，重复的消息通常会产生一个新的响应（或传出消息），它模仿之前处理过的消息的行为。

  对接收到的消息的知识将创建在每个合作伙伴的基础上封装的状态。这里的关键观察是，国家由伙伴组织，伙伴是一个实体。

  我们将术语活动应用于管理两方关系两侧的每方消息的状态。每个活动的生命周期都恰好是一个实体。一个实体将为其接收消息的每个伙伴实体拥有一个活动。

  除了管理消息混战之外，活动还用于管理松散耦合的协议。在一个不可能进行原子事务的世界中，暂定操作用于协商共享结果。这些操作在实体之间执行，并由活动进行管理。

  本文并不是在断言活动可以解决在工作流讨论中如此彻底地描述的达成协议的众所周知的挑战。然而，我们指出，几乎无限的缩放会导致惊人的细粒度的工作流风格的解决方案。参与者是实体，每个实体使用有关所涉及的其他实体的特定知识来管理其工作流。在一个实体内部保持的两方知识就是我们所谓的活动。

  一些活动的例子有时是很微妙的。订单应用程序将向运输应用程序发送消息，并包括发送id和发送订单id。消息类型可用于刺激运输应用程序中的状态更改以进行记录

指定的订单已准备货。通常，实现者不会在出现错误时才为重试进行设计。很少会有，但偶尔也会有，这个应用程序

设计师们考虑和计划活动的设计。本文的其余部分将更深入地研究这些断言，并为这些观点提出论点和解释。

2. 实体

本节将更深入地研究实体的本质。我们首先考虑对单个实体内的原子事务的保证。接下来，我们将考虑使用唯一的密钥来访问实体，以及这如何授权应用程序的较低级别（可感知规模的）部分在重新分区时重新定位实体。在此之后，我们将考虑在单个原子事务中可以访问的内容，最后，我们将检查在替代索引上的几乎无限缩放的影响。

序列化性的不相交范围

每个实体都被定义为数据的集合，已知的唯一键位于可序列化的单一范围内。因为它存在于可序列化的单一范围内，所以我们确保始终可以在内部执行原子事务 *a 单一的 实体*

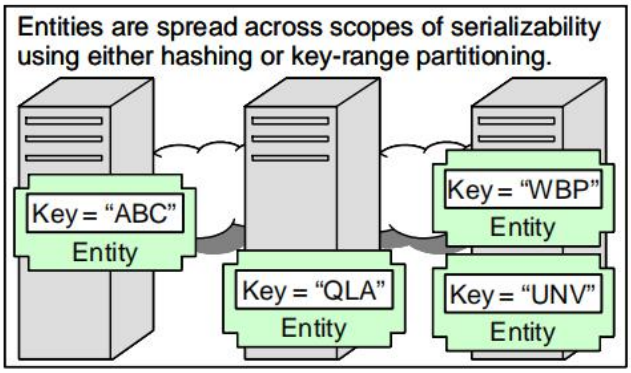
正是这方面保证了给“实体”一个不同于“对象”的名称。对象可能共享也可能不共享事务性作用域。实体从不共享事务范围，因为重新分区可能会将它们放在不同的机器上。

独特的关键实体

应用程序上层的代码自然是围绕着带有唯一键的数据集合而设计的。我们可以一直在应用程序中看到客户id、社会安全号码、产品sku和其他唯一标识符。它们被用作定位实现应用程序的数据的键。这是一个很自然的范例。我们观察到，可串行性的不相交范围的边界(i。e.“实体”)在实践中总是由一个唯一的键来标识。

重新划分和实体

我们的一个假设是，新兴的上层-层是与规模无关的，底层决定部署如何随着规模变化的需求而发展。这意味着一个特定实体的位置很可能会随着部署的发展而变化。应用程序的上层不能对实体的位置做出假设，因为这不会与规模无关。



原子事务和实体

在可扩展的系统中，您 *can’t 假设 处理 用于更新 穿过 这些的 有差别的 实体*每个实体都有一个唯一的键，每个实体都很容易地放在一个可序列化的范围中[10].您如何知道两个独立的实体保证在同一序列化性范围内（因此，原子可更新）？您只知道当有一个唯一的键来统一两者时。现在它真的是一个实体了！

如果我们使用哈希来按实体键进行分区，则不知道当两个具有不同键的实体会落在同一个框上时。如果我们对实体键使用键范围分区，大多数时候相邻的键值驻留在同一台机器上，但偶尔你会不吉利，你的邻居会在另一台机器上。一个简单的测试用例计算键范围分区中的邻居的原子性，很可能在测试部署过程中经历这种原子性。只有在稍后，当重新部署跨不同的可序列化范围移动实体时，潜在的错误才会出现，因为更新不能不再是原子性的。您永远不能指望居住在同一位置的不同实体键值！

更简单地说，应用程序的底层将确保每个实体键（及其实体）驻留在一台机器（或小集群）上。不同的实体可能出现在任何地方。

A 与尺度无关的 规划 抽象 必须 有 概念 的 实体 作为 那 边界 的 原子数 将实体的存在理解为一个程序化的抽象、实体键的使用以及假设跨实体缺乏原子性的明确承诺，对于为应用程序提供一个与规模无关的上层至关重要。

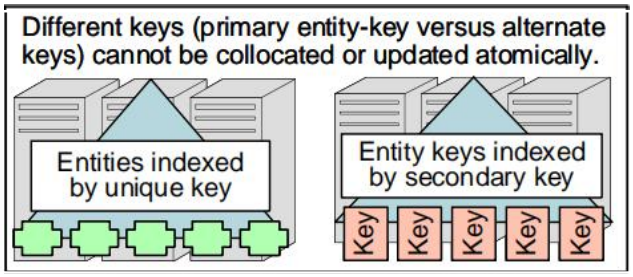
在今天的工业应用程序中，大型应用程序隐式地这样做；我们只是没有实体概念的名称。从上层应用程序的角度来看，它必须假定该实体是可序列化的范围。假设当部署发生更改时，更多的内容将会中断。

考虑替代指数

我们已经习惯了使用多个键或索引来处理数据的能力。例如，有时我们通过社会保险号码，有时通过信用卡号码，有时通过街道地址来引用客户。如果我们假设极端的缩放量，那么这些索引不能驻留在同一台机器或单个大型集群上。*这个 数据 关于 a 单一的 顾客*

[10] 回想一下，几乎无限的比例会导致 *数字 的* 整体无情地增加，但大小 *的 个人* 实体仍然足够小，可以适合于一个可序列化的范围(i。e.一台计算机或小集群)。

*未可 是 大家知道的 向 住 在…之内 a 单一的 范围 可序列化！*实体本身可以位于可序列化的单一范围内。挑战在于，必须假定用于创建替代索引的信息的副本位于不同的可序列化范围内！ 请考虑保证替代索引位于相同的可序列化性范围内。当几乎无限的伸缩开始作用，实体集在大量的机器上涂抹时，主索引和备用索引信息必须位于相同的可序列化性范围内。我们如何确保这一点？确保它们都生活在同一范围内的唯一方法是开始使用主索引查找备用索引！这就把我们带到了相同的可序列化性的范围内。如果我们开始时没有主索引，并且必须搜索所有可序列化的作用域，那么每个替代索引查找都必须检查几乎无限数量的作用域当它寻找与备用键的匹配时！这最终会成为站不住脚的！



唯一合乎逻辑的选择是进行两步查找。首先，我们查找备用键，然后生成实体键。其次，我们使用整个键来访问该实体。这非常像在关系数据库中，因为它使用两个步骤通过备用键访问记录。但是我们几乎无限缩放的前提意味着这两个索引（主要和替代）不能位于同一可序列化性范围内！

*这个 与尺度无关的 申请 程序 can't 原子更新 一 实体 和 它的 交替 索引*上层规模无关应用程序必须理解替代索引可能与使用主索引访问的实体(i。e.实体密钥）。

过去作为替代索引自动管理的内容，现在必须由应用程序手动管理。通过异步消息传递进行的工作流式更新几乎留给了这个无限规模的应用程序。现在必须读取以前作为替代索引保存的数据，同时理解这可能与实现数据的主要表示的实体不同步。以前作为替代索引实现的功能现在更难实现了。这是一个事实的生活在大的残酷的世界的巨大的系统！

3.跨实体的消息传递

在本节中，我们将考虑使用消息连接独立实体的方法。我们检查命名、事务和消息，查看消息传递语义，并考虑重新划分实体的位置对这些消息传递语义的影响。

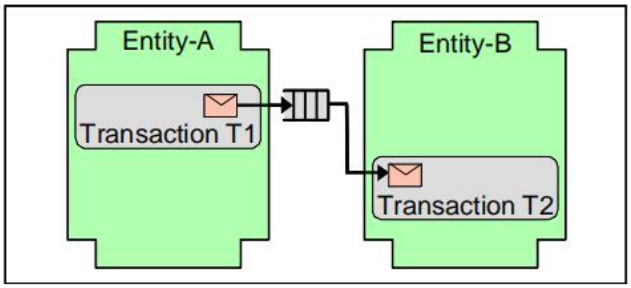
要跨实体进行通信的消息

如果不能在同一事务中的两个实体之间更新数据，则需要一种机制来更新不同事务中的数据。这些实体之间的连接是通过一条消息进行的。

异步的关于发送事务处理

由于消息是跨实体的，因此与决定发送消息相关联的数据在一个实体中，而该消息的目的地在另一个实体中。根据实体的定义，我们必须假定它们不能被原子地更新。

对于应用程序开发人员来说，在处理事务时发送消息，然后发送消息，然后中止事务将是非常复杂的。这将意味着你没有记忆导致了什么事情的发生，但它确实发生了！因此，消息的事务性排队是必要的。



如果在发送事务提交之前无法在目的地看到该消息，那么我们将看到该消息与发送事务相比是异步的。每个实体都通过一个事务前进到一个新的状态。消息是来自一个事务并到达一个新实体的刺激。

命名消息的目的地

当一个实体想要向另一个实体发送消息时，请考虑对应用程序的比例无关部分的编程。与比例无关的代码并不知道目标实体的位置。实体键为。

它属于应用程序的规模感知部分，以便将实体键与实体的位置关联起来。

重新分区和消息传递

当应用程序的比例无关部分发送消息时，较低级别的规模感知部分会搜索目的地并至少传递一次消息。

随着系统的扩展，实体也会移动。这通常被称为重新分区。数据的位置

对于实体，因此，消息的目的地可能在不断变化。有时，信息会追逐到旧位置，却发现讨厌的实体被发送到其他地方。现在，信息必须跟随。

当实体移动时，发送者和目的地之间的先入先出队列的清晰度偶尔会中断。消息重复。稍后的消息比之前的消息到达。生活变得更加混乱。

由于这些原因，我们看到规模不可知的应用程序正在发展，以支持所有应用程序可见消息传递的幂等处理[11]. 这也意味着在消息传递中需要重新排序。

4.实体、SOA和对象

本节将本文的思想与面向对象和面向服务的思想进行了对比。

实体和对象实例

有人可能会问：“一个实体与一个对象实例有何不同？”答案并不是黑白分明的。对象有许多形式，其中一些是实体，而另一些则不是。要考虑一个对象是一个实体，必须作出两个重要的澄清。

首先，由对象封装的数据必须与所有其他数据严格不相交。第二，这些不相交的数据可能永远不会与任何其他数据一起进行原子性更新。

一些对象系统对数据库数据的封装并不明确。在某种程度上，这些不是清晰和努力执行；这些对象不是这里定义的实体。有时，会使用物化视图和替代索引。当系统试图缩放且对象不是实体时，这些情况不会持续下去。

许多对象系统都允许事务范围跨越对象。这种程序性上的便利性消除了本文中描述的大多数挑战。不幸的是，这在几乎无限的缩放下无法工作，除非您的事务耦合对象总是被配置[12].要做到这一点，我们需要为它们分配一个共同的键，以确保共同定位，然后实现这两个交互耦合的对象是同一实体的一部分！

对象很好，但它们是一种不同的抽象。

[11] 通常，具有尺度感知的应用程序最初并不是为消息的幂等性和重新排序而设计的。首先，小规模的部署不会表现出这些微妙的问题，而且工作得很好。只有随着时间的推移和部署的扩展，问题才会显现出来，应用程序才会响应来处理它们。

[12] 或者，您可以忘记配置，而使用两阶段提交。根据我们的假设，我们断言自然选择将开始消除这个问题。

消息与方法

方法调用对于调用线程通常是同步的。它们对于调用对象的事务也是同步的。虽然被调用的对象可能与调用对象原子耦合，也可能不耦合，但典型的方法调用不会原子地记录调用消息的意图，并保证对被调用消息的至少一次调用。有些系统将消息发送包装到方法调用中，我认为它们是消息，而不是方法。

我们没有解决通常将消息传递与方法分开的编组和绑定方面的区别。我们只是指出，事务边界要求异步，这通常在方法调用中找不到。

面向实体和面向服务的体系结构

本文中所讨论的一切都是支持SOA。大多数SOA实现都包含跨服务的独立事务作用域。

这里介绍的对SOA的主要改进是，每个服务都可能面临自身内部几乎无限的扩展，以及对这意味着什么的一些观察。这些观察结果适用于SOA中的服务，也适用于那些被设计为独立规模的个人服务。

5.活动：处理混乱的信息

本节将讨论处理消息重试和重新排序的挑战的方法。我们将活动的概念作为管理与合作伙伴实体的关系所需的本地信息。

重试和阳痿

由于发送的任何消息都可能多次传递，我们需要应用程序中的一个规则来处理重复的消息。虽然可以为消除重复消息构建低级别支持，但在几乎无限伸缩的环境中，低级别支持将需要了解实体。当实体由于重新分区而移动时，知道已传递给实体的消息必须与实体一起传播。在实践中，这种知识的低级管理很少发生；消息可以多次传递。

通常，应用程序的规模不可知（更高级别的）部分必须实现机制，以确保传入的消息是幂等的。这对问题的本质并不是必要的。重复消除当然可以内置到应用程序的规模感知部分中。到目前为止，这还没有可用。因此，我们考虑可伸缩应用程序的糟糕开发人员必须实现什么。

定义实质性行为的无力

*如果处理的后续执行没有执行实质性的处理，则消息的处理是幂等的 更改为实体。*这是一个无定形的定义，它给应用程序留下了什么是实质性的，什么不是实质性的规范。

如果消息没有更改被调用的实体，而只读取信息，则其处理是幂等的。我们认为这是正确的，即使已经写入了描述读取的日志记录。日志记录对实体的行为不是实质性的。什么是实质性的，什么不是实质性的定义是具体应用的。[13]

自然阳痿

要实现幂等性，该信息必须不会产生实质性的副作用。有些信息在任何时候被处理时都不会引发任何实质性的工作。*这些是自然的 幂等*

只从实体中读取一些数据的消息自然是幂等的。如果消息的处理确实改变了实体，但不是以实质性的方式呢？这些也会自然是幂等的。

现在，情况变得更加困难。一些信息所暗示的工作实际上导致了实质性的变化。这些消息天生就不是幂等的。该应用程序必须包括一些机制，以确保这些机制也是幂等的。这意味着以某种方式记住信息已经处理，以便后续尝试不会进行实质性的改变。[14]

我们接下来要考虑的是对自然非幂等的消息的处理。

将消息记住为状态

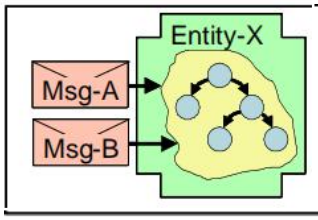
为了确保对非自然幂等的消息进行幂等处理，实体必须记住它们已经被处理过。这种知识就是状态。该状态会随着处理消息而累积。

除了记住消息已经处理，如果需要回复，必须是相同的回复

被送回的毕竟，我们不知道原始发件人是否收到了回复。

活动：管理每个合作伙伴的状态

为了跟踪关系和接收到的消息，规模不可知的应用程序中的每个实体必须以某种方式记住有关其合作伙伴的状态信息。它必须在一个伙伴的基础上捕获这个状态。*让我们将此状态命名为一个活动。*如果每个实体与许多其他实体交互，那么它可能有许多活动。活动会跟踪与每个合作伙伴之间的互动。



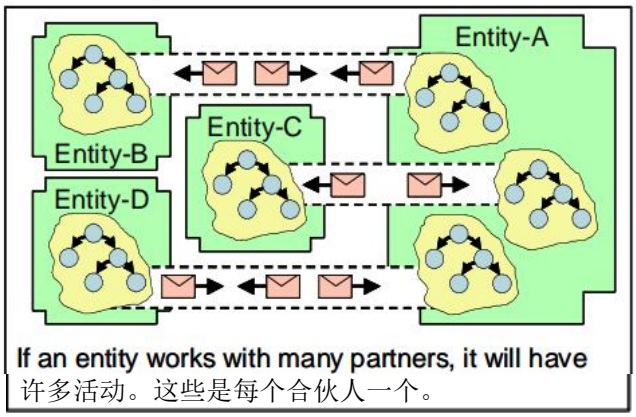
活动很简单。
它们是一个实体
记得其他的
与之合作的实体
包括关于
收到消息。

每个实体都包括一组活动，也许还包括一些跨越这些活动的其他数据。

考虑处理包含许多可购买项目的订单。为每个单独的项目保留库存将是一个单独的活动。订单将有一个实体，而由仓库管理的每个项目都将有一个单独的实体。不能假设在这些实体之间进行交易。

在订单中，每个库存项目都将被单独管理。对消息传递协议必须进行单独的管理。订单实体中包含的每个库存项目数据是一个活动。虽然它没有这样命名，但这种模式经常存在于大型应用程序中。

在一个几乎无限扩展的应用程序中，您需要非常清楚的关系，因为您不能只做一个查询来找出相关的内容。一切都必须通过一个两党关系的网络正式地联系在一起。编织是与实体键。因为离合作伙伴还有很长的路要走，当你有新知识的到来时，你必须正式管理你对合作伙伴的理解。您所知道的关于一个遥远的合作伙伴的本地信息被称为一种活动。



许多活动。这些是每个合伙人一个。

[13] 在数据库社区中，我们经常使用这种技术。例如，在生理日志记录（配置文件类型）系统中，事务的逻辑撤销将使系统保留与事务之前相同的记录。在这样做时，Btree中的页面布局可能会有所不同。这对Btree内容的记录级解释不是实质性的。

[14] 请注意，当消息在细粒度实体之间流动时，很难构建管道以消除重复数据（但肯定是可行的）。大多数持久的重复消除是在比实体粒度更粗的队列上完成的。通常无法使用实体迁移状态以进行重复消除。因此，一个常见的模式是，规模不可知的应用程序包含特定于应用程序的逻辑，以确保消息的冗余处理对实体没有实质性的影响。

通过活动确保最多一次验收

处理非自然幂等的消息需要确保每条消息最多被处理一次(i。e.该信息的实质性影响最多只能发生一次)。要做到这一点，必须记住消息的一些独特特征，以确保它不会被处理不止一次。

实体必须持久地记住从消息正常到处理到消息不会产生实质性影响的状态的转换。

通常，一个实体将利用其活动在一个合作伙伴的基础上实施这种状态管理。这一点很重要，因为有时实体支持许多不同的伙伴，每个伙伴都将通过与该关系相关的消息模式。通过利用每个伙伴的状态集合，程序员可以专注于每个伙伴的关系。

断言，通过关注每个合作伙伴的信息，更容易构建可扩展的应用程序。其中一个例子是在支持幂等消息处理的实现中。

6.活动：没有原子性的应对

本节介绍了可扩展的系统如何在没有分布式事务的情况下做出决策。

本节的重点是，管理分布式协议是一件很困难的工作。此外，在一个几乎无限可伸缩的环境中，不确定性的表示必须以一种面向每个合作伙伴关系的细粒度的方式来完成。此数据在实体中使用活动的概念进行管理。

远处的不确定度

分布式事务的缺乏意味着，当我们试图跨不同实体做出决策时，我们必须接受不确定性。跨分布式系统的决策不可避免地会在一段时间内接受不确定性[15].当可以使用分布式事务时，这种不确定性体现在数据上保存的锁中，并由事务管理器进行管理。

在一个不能依赖于分布式事务的系统中，对不确定性的管理必须在业务逻辑中实现。结果的不确定性被保存在业务语义中，而不是被保存在记录锁中。这个是仅仅工作流程没什么神奇的，只是我们不能使用分布式事务，所以我们需要使用工作流。

引导我们找到实体和信息的假设，引导我们得出结论：规模不可知的应用程序必须使用来管理不确定性本身

[15]在写这篇论文的时候，远距离的滑稽动作还没有被证明，我们受到光速的限制。在远处并没有像同时性这样的东西。。

工作流，如果它需要跨多个实体达成协议。

想想跨企业之间的互动风格。企业之间的合同包括时间承诺、取消条款、保留资源等等。不确定性的语义包含在业务功能的行为中。虽然实现起来比简单地使用分布式事务更复杂，但这是现实世界的工作方式。
同样，这只是工作流的一个参数。

活动与不确定性的管理

实体有时在与其他实体交互时接受不确定性。这种不确定性必须在每个合作伙伴的基础上进行管理，人们可以将其可视化为在合作伙伴的活动状态中被具体化。

很多时候，不确定性是用关系来表示的。有必要通过合作伙伴对其进行跟踪。当每个合作伙伴进入一个新的状态时，该活动就会跟踪它。

如果订购系统从仓库保留库存，仓库在不知道是否使用的情况下分配库存。这是在接受不确定性。稍后，仓库会发现是否需要保留的库存。这解决了不确定性。
仓库库存经理必须为包含其项目的每个订单保存关系数据。当它连接项目和订单时，这些项目将按项目进行组织。在每个项目中都将有关于针对该项目的未完成订单的信息。项目中的每一个活动（每个订单一个）都管理着订单的不确定性。

执行试探性的业务操作

为了在各个实体之间达成协议，一个实体必须要求另一个实体接受一些不确定性。这是通过发送一个要求承诺但保留取消可能性的消息来实现的。*这被称为试探性的尝试*操作，它由在两个实体之间流动的消息表示。在这一步结束时，其中一个实体同意遵守另一个实体的意愿[16].

初步操作、确认和取消

作为一个临时操作的关键，是取消的权利。有时，要求临时操作的实体决定不继续进行。*这是一个取消活动当取消权被释放时，这就是一个确认活动*每一次临时操作最终都会确认或取消。

当一个实体同意执行一个临时操作时，它同意让另一个实体来决定结果。这是在接受不确定性，并增加了该实体的一般困惑体验。随着确认和取消订单的到来，这种情况就会减少

[16]这是一个简单的例子。在某些情况下，可以部分地处理这些操作。在其他情况下，超时和/或违约可能会导致更多的问题。不幸的是，现实世界并不漂亮。

不确定性。随着旧问题得到解决，新问题到达你的身边，不确定性不断增加是正常的。

同样，这只是简单的工作流，但它是细粒度的、以实体作为参与者的工作流。

不确定性和几乎无限的尺度

这种规模的有趣方面是观察到不确定性的管理通常围绕着两方协议。经常会发生多个两方协议。尽管如此，这些都被联系在一起，作为一个细粒度的两方协议网络，使用实体键作为链接和活动来跟踪一个遥远的合作伙伴的已知状态。

考虑购买房屋和与托管公司的关系。买方与第三方托管公司签订了信托协议。卖方、抵押贷款公司和参与交易的所有其他各方也是如此。
当你去买房子去签文件时，你不知道交易的结果。你接受这个事实，直到第三方托管结束，你是不确定的。唯一有权控制决策权的一方是第三方托管公司。
这是一个两方关系的中心和分支的集合，用于让大量各方在不使用分布式事务的情况下达成协议。

当你考虑几乎无限的规模时，考虑两党关系是很有趣的。通过双方暂定/取消/确认（就像传统的工作流程），我们看到了如何实现分布式协议的基础。就像在托管公司中一样，许多实体也可以通过组成来参与协议。

因为这些关系是两党的，所以将一个活动作为“我记得的关于该合作伙伴的事情”的简单概念就成为了管理庞大系统的基础。即使数据存储在实体中，而您不知道数据分布在哪里，必须假设数据很远，也可以以与规模无关的方式进行编程。

现实世界中几乎无限规模的应用程序会喜欢拥有全局范围的可序列化性，正如两阶段提交和其他相关算法所承诺的那样。不幸的是，这些因素的脆弱性对可用性造成了不可接受的压力。相反，对暂定工作的不确定性的管理显然被强加到了这个规模不可知的应用程序的开发者手中。它必须作为预留库存、信用额度分配和其他应用程序特定的概念来处理。

7.结论

和往常一样，计算机行业正在不断变化。一个新兴的趋势是，应用程序可以扩展到不适合单台机器或紧密耦合的机器集的尺寸。正如我们经常看到的，首先出现单个应用程序的特定解决方案，然后出现通用模式观察到。基于这些通用模式，建立了新的设施，从而能够更容易地构建业务逻辑。

在20世纪70年代，许多大型应用程序在提供业务解决方案的同时，努力解决处理多个在线终端的多路复用的困难。新出现的终端控制模式被捕获，一些高端应用程序发展成为tp监视器。最终，这些模式在从头开发的tp监视器中被重复。这些平台允许业务逻辑开发人员专注于他们最擅长的事情：开发业务逻辑。

今天，我们看到新的设计压力强加给那些只想解决业务问题的程序员。他们的现实正把他们带入一个几乎无限规模的世界，并迫使他们陷入与手头的实际业务基本无关的设计问题。

不幸的是，努力解决电子商务、供应链管理、财务和医疗保健应用程序等业务目标的程序员越来越需要考虑在没有分布式事务的情况下进行扩展。它们这样做是因为使用分布式事务的尝试过于脆弱，而且性能很差。

我们正处于一个可以看到构建这些应用程序的模式的关键时刻，但目前还没有人能够一致地应用这些模式。本文认为，这些新生的模式可以更一致地应用于针对几乎无限扩展而设计的应用程序的手工开发中。此外，在几年内，我们可能会看到新的中间件或平台的开发，它们提供了这些应用程序的自动化管理，并消除了在程式化编程范式中开发的应用程序的扩展挑战。这与20世纪70年代出现的tp监测器的情况非常相似。

在本文中，我们介绍并命名了一些出现在大规模应用中的形式主义：

- 实体是命名（键控）数据的集合
  可以在实体内原子更新，但不会在实体间原子更新。
- 活动包括在国家范围内的国家集合
  用于管理与单个合作伙伴实体之间的消息传递关系的实体。

正如多年来讨论的那样，达成决策的工作流在实体内部的活动中发挥作用。当人们看到几乎无限的扩展时，令人惊讶的是工作流的细粒度本质。

有人认为，今天的许多应用程序都在隐式地设计实体和活动。它们只是没有被形式化，也没有被持续使用。在使用不一致的地方，会发现bug并最终进行修补。通过讨论和一致地使用这些模式，可以构建更好的大规模应用程序，作为一个行业，我们可以更接近于构建解决方案，允许业务逻辑程序员专注于业务问题，而不是规模问题。