**分布式一致性**

**一致性的重要性**

分布式领域CAP理论告诉我们，任何一个分布式系统都无法同时满足Consistency(一致性)、Availability(可用性)、Partition tolerance(分区容错性) 这三个基本需求。最多只能满足其中两项。 但是，一个分布式系统无论在CAP三者之间如何权衡，都无法彻底放弃一致性（Consistency），如果真的放弃一致性，那么就说明这个系统中的数据根本不可信，数据也就没有意义，那么这个系统也就没有任何价值可言。所以，无论如何，分布式系统的一致性问题都需要重点关注。

**CAP的定义**

**Consistency 一致性**

一致性指“**all nodes see the same data at the same time**”，即更新操作成功并返回客户端完成后，所有节点在同一时间的数据完全一致。

对于一致性，可以分为从客户端和服务端两个不同的视角。从客户端来看，一致性主要指的是多并发访问时更新过的数据如何获取的问题。从服务端来看，则是更新如何复制分布到整个系统，以保证数据最终一致。一致性是因为有并发读写才有的问题，因此在理解一致性的问题时，一定要注意结合考虑并发读写的场景。

从客户端角度，多进程并发访问时，更新过的数据在不同进程如何获取的不同策略，决定了不同的一致性。对于关系型数据库，要求更新过的数据能被后续的访问都能看到，这是强一致性。如果能容忍后续的部分或者全部访问不到，则是弱一致性(弱一致性可以理解为，系统数据可以不一致，一般少有这种情况)。如果经过一段时间后要求能访问到更新后的数据，则是最终一致性。

**Availability 可用性**

可用性指“**Reads and writes always succeed**”，即服务一直可用，而且是正常响应时间。

对于一个可用性的分布式系统，每一个非故障的节点必须对每一个请求作出响应。也就是，该系统使用的任何算法必须最终终止。当同时要求分区容忍性时，这是一个很强的定义：即使是严重的网络错误，每个请求必须终止。

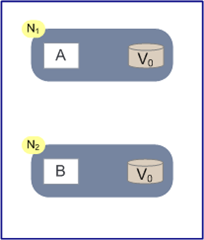
好的可用性主要是指系统能够很好的为用户服务，不出现用户操作失败或者访问超时等用户体验不好的情况。可用性通常情况下可用性和分布式数据冗余，负载均衡等有着很大的关联。

**Partition Tolerance分区容错性**

分区容错性指“**the system continues to operate despite arbitrary message loss or failure of part of the system**”，即分布式系统在遇到某节点或网络分区故障的时候，仍然能够对外提供满足一致性和可用性的服务。

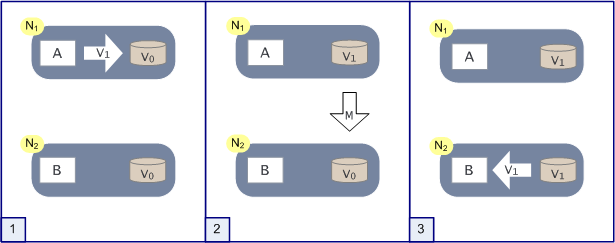
分区容错性和扩展性紧密相关。在分布式应用中，可能因为一些分布式的原因导致系统无法正常运转。好的分区容错性要求能够使应用虽然是一个分布式系统，而看上去却好像是在一个可以运转正常的整体。比如现在的分布式系统中有某一个或者几个机器宕掉了，其他剩下的机器还能够正常运转满足系统需求，或者是机器之间有网络异常，将分布式系统分隔未独立的几个部分，各个部分还能维持分布式系统的运作，这样就具有好的分区容错性。

**CAP的证明**



如上图，是我们证明CAP的基本场景，网络中有两个节点N1和N2，可以简单的理解N1和N2分别是两台计算机，他们之间网络可以连通，N1中有一个应用程序A，和一个数据库V，N2也有一个应用程序B2和一个数据库V。现在，A和B是分布式系统的两个部分，V是分布式系统的数据存储的两个子数据库。

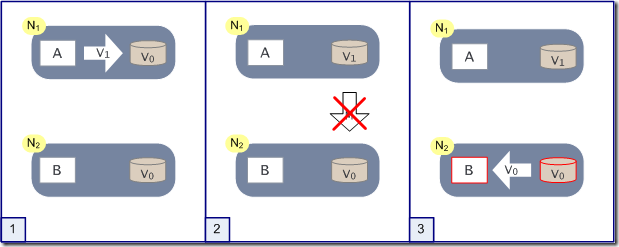
在满足一致性的时候，N1和N2中的数据是一样的，V0=V0。在满足可用性的时候，用户不管是请求N1或者N2，都会得到立即响应。在满足分区容错性的情况下，N1和N2有任何一方宕机，或者网络不通的时候，都不会影响N1和N2彼此之间的正常运作。



如上图，是分布式系统正常运转的流程，用户向N1机器请求数据更新，程序A更新数据库Vo为V1，分布式系统将数据进行同步操作M，将V1同步的N2中V0，使得N2中的数据V0也更新为V1，N2中的数据再响应N2的请求。

这里，可以定义N1和N2的数据库V之间的数据是否一样为一致性；外部对N1和N2的请求响应为可用行；N1和N2之间的网络环境为分区容错性。这是正常运作的场景，也是理想的场景，然而现实是残酷的，当错误发生的时候，一致性和可用性还有分区容错性，是否能同时满足，还是说要进行取舍呢？

作为一个分布式系统，它和单机系统的最大区别，就在于网络，现在假设一种极端情况，N1和N2之间的网络断开了，我们要支持这种网络异常，相当于要满足分区容错性，能不能同时满足一致性和响应性呢？还是说要对他们进行取舍。



假设在N1和N2之间网络断开的时候，有用户向N1发送数据更新请求，那N1中的数据V0将被更新为V1，由于网络是断开的，所以分布式系统同步操作M，所以N2中的数据依旧是V0；这个时候，有用户向N2发送数据读取请求，由于数据还没有进行同步，应用程序没办法立即给用户返回最新的数据V1，怎么办呢？有二种选择，第一，牺牲数据一致性，响应旧的数据V0给用户；第二，牺牲可用性，阻塞等待，直到网络连接恢复，数据更新操作M完成之后，再给用户响应最新的数据V1。

这个过程，证明了要满足分区容错性的分布式系统，只能在一致性和可用性两者中，选择其中一个。

**CAP权衡**

通过CAP理论，我们知道无法同时满足一致性、可用性和分区容错性这三个特性，那要舍弃哪个呢？

CA without P：如果不要求P（不允许分区），则C（强一致性）和A（可用性）是可以保证的。但其实分区不是你想不想的问题，而是始终会存在，因此CA的系统更多的是允许分区后各子系统依然保持CA。

**CP without A：如果不要求A（可用），相当于每个请求都需要在Server之间强一致，而P（分区）会导致同步时间无限延长，如此CP也是可以保证的。很多传统的数据库分布式事务都属于这种模式**。

AP wihtout C：要高可用并允许分区，则需放弃一致性。一旦分区发生，节点之间可能会失去联系，为了高可用，每个节点只能用本地数据提供服务，而这样会导致全局数据的不一致性。现在众多的NoSQL都属于此类。

对于多数大型互联网应用的场景，主机众多、部署分散，而且现在的集群规模越来越大，所以节点故障、网络故障是常态，而且要保证服务可用性达到N个，即保证P和A，舍弃C（退而求其次保证最终一致性）。虽然某些地方会影响客户体验，但没达到造成用户流程的严重程度。

对于涉及到钱财这样不能有一丝让步的场景，C必须保证。网络发生故障宁可停止服务，这是保证CA，舍弃P。貌似这几年国内银行业发生了不下10起事故，但影响面不大，报到也不多，广大群众知道的少。还有一种是保证CP，舍弃A。例如网络故障事只读不写。

孰优孰略，没有定论，只能根据场景定夺，适合的才是最好的。

这里先简单提一下，由于一个分布式系统不可能放弃一致性，那么为什么有的架构师还说在某些场景中可以牺牲一致性呢？通常这里说的放弃一致性指的是放弃数据的强一致性。

**数据一致性**

数据一致性其实是数据库系统中的概念。我们可以简单的把一致性理解为正确性或者完整性，那么数据一致性通常指关联数据之间的逻辑关系是否正确和完整。我们知道，在数据库系统中通常用事务（访问并可能更新数据库中各种数据项的一个程序执行单元）来保证数据的一致性和完整性。**而在分布式系统中，数据一致性往往指的是由于数据的复制，不同数据节点中的数据内容是否完整并且相同。**

**为什么会有数据一致性问题**

虽然分布式系统有着诸多优点，但是由于采用多机器进行分布式部署的方式提供服务，必然存在着数据的复制。分布式系统的数据复制需求主要来源于以下两个原因：

可用性。将数据复制到分布式部署的多台机器中，可以消除单点故障，防止系统由于某台（些）机器宕机导致的不可用。 性能。通过负载均衡技术，能够让分布在不同地方的数据副本全都对外提供服务。有效提高系统性能。

在分布式系统引入复制机制后，不同的数据节点之间由于网络延时等原因很容易产生数据不一致的情况。复制机制的目的是为了保证数据的一致性。但是数据复制面临的主要难题也是如何保证多个副本之间的数据一致性。

一个系统如果想保证数据一致性很有可能影响其性能。因为并发的写请求需要在前一个写请求结束（分布式系统数据同步结束）之后才能进行。因此，如何能既保证数据一致性，又保证系统的性能，是每一个分布式系统都需要重点考虑和权衡的。一致性模型可以在做这些权衡的时候给我们很多借鉴和思考。

**一致性模型**

**强一致性**

当更新操作完成之后，任何多个后续进程或者线程的访问都会返回最新的更新过的值。这种是对用户最友好的，就是用户上一次写什么，下一次就保证能读到什么。但是这种实现对性能影响较大。

**弱一致性**

系统并不保证续进程或者线程的访问都会返回最新的更新过的值。系统在数据写入成功之后，不承诺立即可以读到最新写入的值，也不会具体的承诺多久之后可以读到。但会尽可能保证在某个时间级别（比如秒级别）之后，可以让数据达到一致性状态。（补充：也可能永远达不到一致性，表示系统对一致性没有强制要求，允许出现数据不一致的现象，此类情况较少）。

**最终一致性**

弱一致性的特定形式。系统保证在没有后续更新的前提下，系统最终返回上一次更新操作的值。在没有故障发生的前提下，不一致窗口的时间主要受通信延迟，系统负载和复制副本的个数影响。DNS是一个典型的最终一致性系统。

**最终一致性模型的变种**

会话一致性：读己所写一致性的特定形式。进程在访问存储系统同一个会话内，系统保证该进程读己之所写。

单调读一致性：如果一个进程已经读取到一个特定值，那么该进程不会读取到该值以前的任何值。

上述最终一致性的不同方式可以进行组合，例如单调读一致性和读己之所写一致性就可以组合实现。并且从实践的角度来看，这两者的组合，读取自己更新的 数据，和一旦读取到最新的版本不会再读取旧版本，对于此架构上的程序开发来说，会少很多额外的烦恼。

为了解决分布式的一致性问题，在长期的研究探索过程中，涌现出了一大批经典的一致性协议和算法，其中比较著名的有二阶段提交协议，三阶段提交协议和Paxos算法。

**分布式一致性协议**

本文将简单介绍如何有效的解决分布式的一致性问题，其中包括什么是**分布式事务**，**二阶段提交**和**三阶段提交**。

**分布式一致性回顾**

在分布式系统中，为了保证数据的高可用，通常，我们会将数据保留多个副本(replica)，这些副本会放置在不同的物理的机器上。为了对用户提供正确的增\删\改\差等语义，我们需要保证这些放置在不同物理机器上的副本是一致的。

为了解决这种分布式一致性问题，前人在性能和数据一致性的反反复复权衡过程中总结了许多典型的协议和算法。其中比较著名的有**二阶提交协议**（Two Phase Commitment Protocol）、**三阶提交协议**（Two Phase Commitment Protocol）和**Paxos算法**。

**分布式事务**

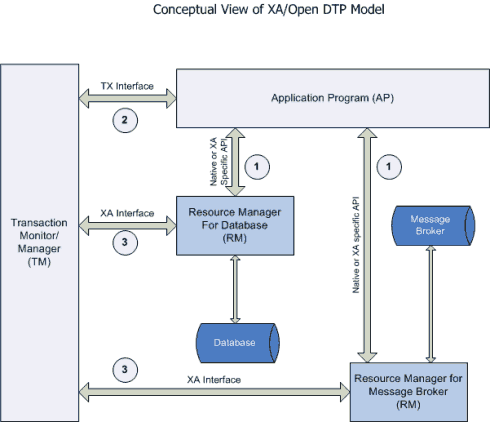
分布式事务是指会涉及到操作多个数据库的事务。其实就是将对同一库事务的概念扩大到了对多个库的事务。目的是为了保证分布式系统中的数据一致性。分布式事务处理的关键是必须有一种方法可以知道事务在任何地方所做的所有动作，提交或回滚事务的决定必须产生统一的结果（全部提交或全部回滚）

在分布式系统中，各个节点之间在物理上相互独立，通过网络进行沟通和协调。由于存在事务机制，可以保证每个独立节点上的数据操作可以满足ACID。但是，相互独立的节点之间无法准确的知道其他节点中的事务执行情况。所以从理论上讲，两台机器理论上无法达到一致的状态。如果想让分布式部署的多台机器中的数据保持一致性，那么就要保证在所有节点的数据写操作，要不全部都执行，要么全部的都不执行。但是，一台机器在执行本地事务的时候无法知道其他机器中的本地事务的执行结果。所以他也就不知道本次事务到底应该commit还是 roolback。所以，常规的解决办法就是引入一个“**协调者**”的组件来统一调度所有分布式节点的执行。

**XA规范**

X/Open 组织（即现在的 Open Group ）定义了分布式事务处理模型。 X/Open DTP (Distributed Transaction Process)模型（ 1994 ）包括应用程序（ AP ）、事务管理器（ TM ）、资源管理器（ RM ）、通信资源管理器（ CRM ）四部分。一般，常见的事务管理器（ TM ）是交易中间件，常见的资源管理器（ RM ）是数据库，常见的通信资源管理器（ CRM ）是消息中间件。 通常把一个数据库内部的事务处理，如对多个表的操作，作为本地事务看待。数据库的事务处理对象是本地事务，而分布式事务处理的对象是全局事务。 所谓全局事务，是指分布式事务处理环境中，多个数据库可能需要共同完成一个工作，这个工作即是一个全局事务，例如，一个事务中可能更新几个不同的数据库。对数据库的操作发生在系统的各处但必须全部被提交或回滚。此时一个数据库对自己内部所做操作的提交不仅依赖本身操作是否成功，还要依赖与全局事务相关的其它数据库的操作是否成功，如果任一数据库的任一操作失败，则参与此事务的所有数据库所做的所有操作都必须回滚。 一般情况下，某一数据库无法知道其它数据库在做什么，因此，在一个 DTP 环境中，**交易中间件**是必需的，由它通知和协调相关数据库的提交或回滚。而一个数据库只将其自己所做的操作（可恢复）影射到全局事务中。

XA协议 就是 X/Open DTP 定义的交易中间件与数据库之间的接口规范（即接口函数），交易中间件用它来通知数据库事务的开始、结束以及提交、回滚等。 XA 接口函数由数据库厂商提供。



作为java平台上事务规范JTA（Java Transaction API）也定义了对XA事务的支持，实际上，JTA是基于XA架构上建模的，在JTA 中，事务管理器抽象为javax.transaction.TransactionManager接口，并通过底层事务服务（即JTS）实现。像很多其他的java规范一样，JTA仅仅定义了接口，具体的实现则是由供应商(如J2EE厂商)负责提供

**二阶提交协议**和**三阶提交协议**就是根据这一思想衍生出来的。可以说二阶段提交其实就是实现**XA分布式事务**的关键(确切地说：两阶段提交主要保证了分布式事务的原子性：即所有结点要么全做要么全不做)

**2PC**

二阶段提交(Two-phaseCommit)是指，在计算机网络以及数据库领域内，为了使基于分布式系统架构下的所有节点在进行事务提交时保持一致性而设计的一种算法(Algorithm)。通常，二阶段提交也被称为是一种协议(Protocol))。在分布式系统中，每个节点虽然可以知晓自己的操作时成功或者失败，却无法知道其他节点的操作的成功或失败。当一个事务跨越多个节点时，为了保持事务的ACID特性，需要引入一个作为协调者的组件来统一掌控所有节点(称作参与者)的操作结果并最终指示这些节点是否要把操作结果进行真正的提交(比如将更新后的数据写入磁盘等等)。因此，**二阶段提交的算法思路可以概括为：参与者将操作成败通知协调者，再由协调者根据所有参与者的反馈情报决定各参与者是否要提交操作还是中止操作。**

所谓的两个阶段是指：第一阶段：**准备阶段(投票阶段)**和第二阶段：**提交阶段（执行阶段）**。

**准备阶段**

事务协调者(事务管理器)给每个参与者(资源管理器)发送Prepare消息，每个参与者要么直接返回失败(如权限验证失败)，要么在本地执行事务，写本地的redo和undo日志，但不提交，到达一种“万事俱备，只欠东风”的状态。

可以进一步将准备阶段分为以下三个步骤：

1. 协调者节点向所有参与者节点询问是否可以执行提交操作(vote)，并开始等待各参与者节点的响应。
2. 参与者节点执行询问发起为止的所有事务操作，并将Undo信息和Redo信息写入日志。（注意：若成功这里其实每个参与者已经执行了事务操作）
3. 各参与者节点响应协调者节点发起的询问。如果参与者节点的事务操作实际执行成功，则它返回一个”同意”消息；如果参与者节点的事务操作实际执行失败，则它返回一个”中止”消息。

**提交阶段**

如果协调者收到了参与者的失败消息或者超时，直接给每个参与者发送回滚(Rollback)消息；否则，发送提交(Commit)消息；参与者根据协调者的指令执行提交或者回滚操作，释放所有事务处理过程中使用的锁资源。(注意:必须在最后阶段释放锁资源)

接下来分两种情况分别讨论提交阶段的过程。

当协调者节点从所有参与者节点获得的相应消息都为”同意”时:

1. 协调者节点向所有参与者节点发出”正式提交(commit)”的请求。
2. 参与者节点正式完成操作，并释放在整个事务期间内占用的资源。
3. 参与者节点向协调者节点发送”完成”消息。
4. 协调者节点受到所有参与者节点反馈的”完成”消息后，完成事务。

如果任一参与者节点在第一阶段返回的响应消息为”中止”，或者 协调者节点在第一阶段的询问超时之前无法获取所有参与者节点的响应消息时：

1. 协调者节点向所有参与者节点发出”回滚操作(rollback)”的请求。
2. 参与者节点利用之前写入的Undo信息执行回滚，并释放在整个事务期间内占用的资源。
3. 参与者节点向协调者节点发送”回滚完成”消息。
4. 协调者节点受到所有参与者节点反馈的”回滚完成”消息后，取消事务。

不管最后结果如何，第二阶段都会结束当前事务。

二阶段提交看起来确实能够提供原子性的操作，但是不幸的事，二阶段提交还是有几个**缺点**的：

1. **同步阻塞问题**。执行过程中，所有参与节点都是事务阻塞型的。当参与者占有公共资源时，其他第三方节点访问公共资源不得不处于阻塞状态。
2. **单点故障**。由于协调者的重要性，一旦协调者发生故障。参与者会一直阻塞下去。尤其在第二阶段，协调者发生故障，那么所有的参与者还都处于锁定事务资源的状态中，而无法继续完成事务操作。（如果是协调者挂掉，可以重新选举一个协调者，但是无法解决因为协调者宕机导致的参与者处于阻塞状态的问题）
3. **数据不一致**。在二阶段提交的阶段二中，当协调者向参与者发送commit请求之后，发生了局部网络异常或者在发送commit请求过程中协调者发生了故障，这回导致只有一部分参与者接受到了commit请求。而在这部分参与者接到commit请求之后就会执行commit操作。但是其他部分未接到commit请求的机器则无法执行事务提交。于是整个分布式系统便出现了数据不一致性的现象。
4. 二阶段无法解决的问题：协调者在发出commit消息之后宕机，而唯一接收到这条消息的参与者同时也宕机了。那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道事务是否被已经提交。

由于二阶段提交存在着诸如同步阻塞、单点问题、脑裂等缺陷，所以，研究者们在二阶段提交的基础上做了改进，提出了三阶段提交。

**3PC**

三阶段提交（Three-phase commit），也叫三阶段提交协议（Three-phase commit protocol），是二阶段提交（2PC）的改进版本。

与两阶段提交不同的是，三阶段提交有两个改动点。

1. 引入超时机制。同时在协调者和参与者中都引入超时机制。
2. 在第一阶段和第二阶段中插入一个准备阶段。保证了在最后提交阶段之前各参与节点的状态是一致的。

也就是说，除了引入超时机制之外，3PC把2PC的准备阶段再次一分为二，这样三阶段提交就有CanCommit、PreCommit、DoCommit三个阶段。

**CanCommit阶段**

3PC的CanCommit阶段其实和2PC的准备阶段很像。协调者向参与者发送commit请求，参与者如果可以提交就返回Yes响应，否则返回No响应。

1. **事务询问** 协调者向参与者发送CanCommit请求。询问是否可以执行事务提交操作。然后开始等待参与者的响应。
2. **响应反馈** 参与者接到CanCommit请求之后，正常情况下，如果其自身认为可以顺利执行事务，则返回Yes响应，并进入预备状态。否则反馈No

**PreCommit阶段**

协调者根据参与者的反应情况来决定是否可以执行事务的PreCommit操作。根据响应情况，有以下两种可能。

**假如协调者从所有的参与者获得的反馈都是Yes响应，那么就会执行事务的预执行。**

1. **发送预提交请求** 协调者向参与者发送PreCommit请求，并进入Prepared阶段。
2. **事务预提交** 参与者接收到PreCommit请求后，会执行事务操作，并将undo和redo信息记录到事务日志中。
3. **响应反馈** 如果参与者成功的执行了事务操作，则返回ACK响应，同时开始等待最终指令。

假如有任何一个参与者向协调者发送了No响应，或者等待超时之后，协调者都没有接到参与者的响应，那么就执行事务的中断**。**

1. **发送中断请求** 协调者向所有参与者发送abort请求。
2. **中断事务** 参与者收到来自协调者的abort请求之后（或超时之后，仍未收到协调者的请求），执行事务的中断。

**doCommit阶段**

该阶段进行真正的事务提交，也可以分为以下两种情况。

**执行提交**

1. **发送提交请求** 协调接收到参与者发送的ACK响应，那么他将从预提交状态进入到提交状态。并向所有参与者发送doCommit请求。
2. **事务提交** 参与者接收到doCommit请求之后，执行正式的事务提交。并在完成事务提交之后释放所有事务资源。
3. **响应反馈** 事务提交完之后，向协调者发送Ack响应。
4. **完成事务** 协调者接收到所有参与者的ack响应之后，完成事务。

**中断事务** 协调者没有接收到参与者发送的ACK响应（可能是接受者发送的不是ACK响应，也可能响应超时），那么就会执行中断事务。

1. **发送中断请求** 协调者向所有参与者发送abort请求
2. **事务回滚** 参与者接收到abort请求之后，利用其在阶段二记录的undo信息来执行事务的回滚操作，并在完成回滚之后释放所有的事务资源。
3. **反馈结果** 参与者完成事务回滚之后，向协调者发送ACK消息
4. **中断事务** 协调者接收到参与者反馈的ACK消息之后，执行事务的中断。

在doCommit阶段，如果参与者无法及时接收到来自协调者的doCommit或者rebort请求时，会在等待超时之后，会继续进行事务的提交。（其实这个应该是基于概率来决定的，当进入第三阶段时，说明参与者在第二阶段已经收到了PreCommit请求，那么协调者产生PreCommit请求的前提条件是他在第二阶段开始之前，收到所有参与者的CanCommit响应都是Yes。（一旦参与者收到了PreCommit，意味他知道大家其实都同意修改了）所以，一句话概括就是，当进入第三阶段时，由于网络超时等原因，虽然参与者没有收到commit或者abort响应，但是他有理由相信：成功提交的几率很大。 ）

**2PC与3PC的区别**

相对于2PC，3PC主要解决的单点故障问题，并减少阻塞，因为一旦参与者无法及时收到来自协调者的信息之后，他会默认执行commit。而不会一直持有事务资源并处于阻塞状态。但是这种机制也会导致数据一致性问题，因为，由于网络原因，协调者发送的abort响应没有及时被参与者接收到，那么参与者在等待超时之后执行了commit操作。这样就和其他接到abort命令并执行回滚的参与者之间存在数据不一致的情况。另外，如果在doCommit阶段，部分参与者机器宕机，从而导致部分数据commit失败，那么也会造成数据不一致。

**其他**

**一阶段提交(Best Efforts 1PC模式)**

不像两阶段提交那样复杂，一阶段提交非常直白，就是从应用程序向数据库发出提交请求到数据库完成提交或回滚之后将结果返回给应用程序的过程。一阶段提交不需要“协调者”角色，各结点之间不存在协调操作，因此其事务执行时间比两阶段提交要短，但是提交的“危险期”是每一个事务的实际提交时间，相比于两阶段提交，一阶段提交出现在“不一致”的概率就变大了。但是我们必须注意到：只有当基础设施出现问题的时候(如网络中断，当机等)，一阶段提交才可能会出现“不一致”的情况，相比它的性能优势，很多团队都会选择这一方案。关于在spring环境下如何实现一阶段提交,有一篇非常优秀的文章值得参考：<http://www.javaworld.com/javaworld/jw-01-2009/jw-01-spring-transactions.html?page=5>

**事务补偿机制**

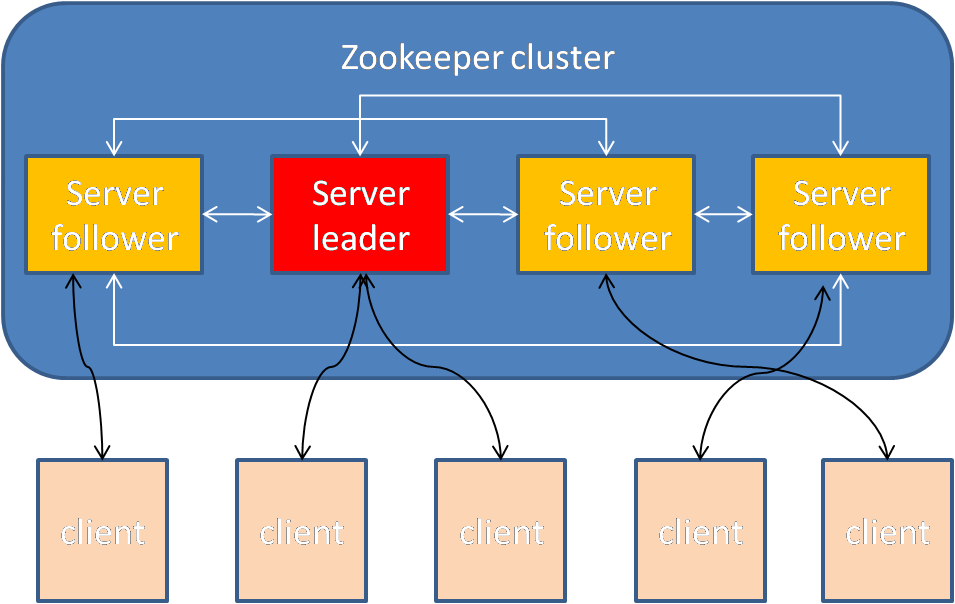
像best efforts 1PC这种模式，前提是应用程序能获取所有的数据源，然后使用同一个事务管理器(这里指是的spring的事务管理器)管理事务。这种模式最典型的应用场景非数据库sharding莫属。但是对于那些基于消息通讯等构建的高度自治(autonomy)的分布式系统接口，best efforts 1PC模式是无能为力的，此类场景下，还有最后一种方法可以帮助我们实现“最终一致性”，那就是事务补偿机制。关于事务补偿机制是一个大话题，本文只简单提及。

了解了2PC和3PC之后，我们可以发现，无论是二阶段提交还是三阶段提交都无法彻底解决分布式的一致性问题。Google Chubby的作者Mike Burrows说过， there is only one consensus protocol, and that’s Paxos” – all other approaches are just broken versions of Paxos. 意即**世上只有一种一致性算法，那就是Paxos**，所有其他一致性算法都是Paxos算法的不完整版。

**Zookeeper实现的Paxos选举算法**

**概述**

Zookeeper集群中，只有一个节点是leader节点，其他节点都是follower节点（实际上还有observer节点，不参与选举的投票，在这里我们先忽略，下同）。所有的更新操作，必须经过leader节点，leader节点和follower节点之间保持着数据同步和心跳。



客户端使用zookeeper时，可能会连到follower身份的server上，也可能会连到leader身份的server上。

三类角色的分工如下：

Leader：处理写请求，单点

Follower：处理客户端请求，参与投票

Observer：不参与leader选举的投票，只处理客户端请求

在一个zookeeper集群里，有多少个server是固定的，每个节点有一个唯一的id，标识它自己，另外，每个server还有用于选举的IP和port，这些都在配置文件中。一个具体的例子如下：

server.1=192.168.0.11:2888:3888

server.2=192.168.0.12:2888:3888

server.3=192.168.0.13:2888:3888

这里有3个server，其id分别为1、2、3。2888为节点和leader交换信息的端口，3888为群首选举的端口。这个节点的id，在投票时，用户标识参加竞选的节点的身份。

问题：这个leader节点是怎么确定的？

答案：zookeeper系统自己选举出来的，所有的server节点（observer除外），都参与这个选举。这样做的好处是：当现在的leader挂掉了之后，系统可以重新选举一个节点做leader。

Zookeeper的选举算法能保证：只要超过半数的节点还活着，就一定能选举出唯一个一个节点作为leader。

**节点的状态**

Zookeeper中的节点有以下三种状态（忽略observer节点）：

LOOKING：初始化状态，处于选举过程中，leader还没有选出

LEADING：leader已经选出，本节点是leader

FOLLOWING：leader已经选出，本节点是follower

**选举发生的时机**

当任何一个节点进入looking状态时，选举开始，进入looking状态有如下原因：

1. 节点刚启动，使自己进入选举状态
2. 发现leader节点挂掉了

Zookeeper中的leader怎么知道follower还活着？follower怎么知道leader还活着？

leader会定时向follower发ping消息；follower会定时向leader发ping消息。当发现无法ping通leader时，就会将自己的状态改为LOOKING，并发起新的一轮选举。处于选举模式时，zookeeper的服务不可用。

**一个节点成为leader的条件**

一个节点要成为leader，必须得到至少**n/2+1**（即半数以上节点）的投票，实际上，在实现时，还可以考虑其他规则，比如节点权重。

为什么要保证至少n/2+1的节点同意？因为这样能保证本节点得到多数派的支持。因为每一个节点，只能支持一个节点成为leader，因此，只要一个节点获得至少n/2+1的选票，就一定会比其他任何节点得到的选票多。这个规则意味着，如果超过半数以上的节点挂掉，zookeeper是选举不出leader节点的，因此，zookeeper集群最多允许n/2的节点故障。

**要解决的问题**

选举算法的目标是确保一定要选出一个唯一的leader节点。这有两层含义：

1. 一定要选出一个节点作为leader
2. 这个leader一定要唯一

**在一次选举中，节点应该把票投给谁？**

* 1. 规则1: 谁的zxid大选谁（zxid大表示该机器处理最多的zk数据更新事务，选该机器可以避免选举之后的Leader数据复制，只需要把所有的zk 事物分给其他Flower即可）。
  2. 规则2: 如果存在2台机器具有相同的zxId，则选择id最大的作为Leader，id是写在zk启动conf文件里面的。

zookeeper的leader选举源代码在：src\java\main\org\apache\zookeeper\server\quorum目录下。zookeeper实现了3种选举算法，如FastLeaderElection.java等。