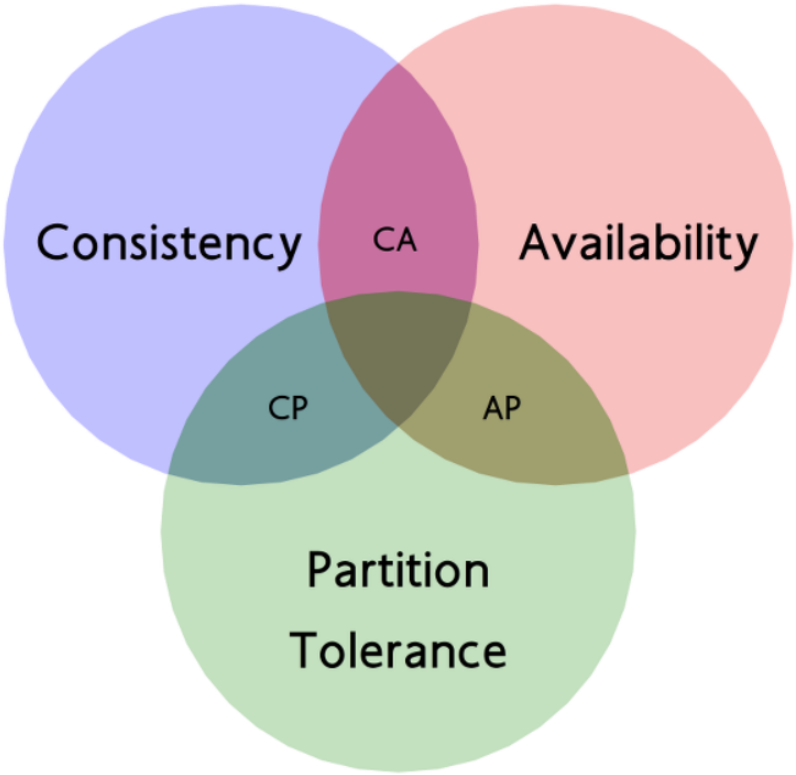
主讲老师：Fox

> 有道笔记：https://note.youdao.com/s/QWLCcyfF

1. 分布式基础理论

1.1 CAP理论

CAP 理论可以表述为，一个分布式系统最多只能同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition Tolerance）这三项中的两项。



一致性是指“所有节点同时看到相同的数据”，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，等同于所有节点拥有数据的最新版本。

可用性是指“任何时候，读写都是成功的”，即服务一直可用，而且是正常响应时间。我们平时会看到一些 IT 公司的对外宣传，比如系统稳定性已经做到 3 个 9、4 个 9，即 99.9%、99.99%，这里的 N 个 9 就是对可用性的一个描述，叫做 SLA，即服务水平协议。比如我们说月度 99.95% 的 SLA，则意味着每个月服务出现故障的时间只能占总时间的 0.05%，如果这个月是 30 天，那么就是 21.6 分钟。

分区容错性具体是指“当部分节点出现消息丢失或者分区故障的时候，分布式系统仍然能够继续运行”，即系统容忍网络出现分区，并且在遇到某节点或网络分区之间网络不可达的情况下，仍然能够对外提供满足一致性和可用性的服务。

分布式系统所关注的，就是在 Partition Tolerance 的前提下，如何实现更好的 A 和更稳定的 C。业务上对一致性的要求会直接反映在系统设计中，典型的就是 CP 和 AP架构。

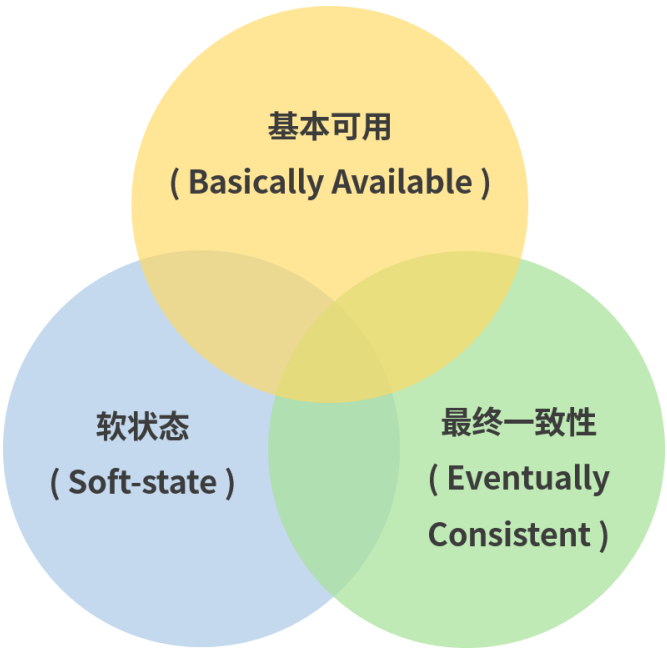
- **CP 架构**：对于 CP 来说，放弃可用性，追求一致性和分区容错性。
- **AP 架构**：对于 AP 来说，放弃强一致性，追求分区容错性和可用性，这是很多分布式系统设计时的选择，后面

的 Base 也是根据 AP 来扩展的。

对于多数大型互联网应用的场景，结点众多、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9（99.99..%），并要达到良好的响应性能来提高用户体验，因此一般都会做出如下选择：保证P和A，舍弃C强一致，保证最终一致性。

1.2 BASE理论

Base 是三个短语的简写，即基本可用（Basically Available）、软状态（Soft State）和最终一致性（Eventually Consistent）。



Base 理论的核心思想是最终一致性，即使无法做到强一致性（Strong Consistency），但每个应用都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性（Eventual Consistency）。

基本可用

基本可用比较好理解，就是不追求 CAP 中的「任何时候，读写都是成功的」，而是系统能够基本运行，一直提供服务。基本可用强调了分布式系统在出现不可预知故障的时候，允许损失部分可用性，相比正常的系统，可能是响应时间延长，或者是服务被降级。

举个例子，在双十一秒杀活动中，如果抢购人数太多超过了系统的 QPS 峰值，可能会排队或者提示限流，这就是通过合理的手段保护系统的稳定性，保证主要的服务正常，保证基本可用。

软状态

软状态可以对应 ACID 事务中的原子性，在 ACID 的事务中，实现的是强一致性，要么全做要么不做，所有用户看到的数据一致。其中的原子性（Atomicity）要求多个节点的数据副本都是一致的，强调数据的一致性。

原子性可以理解为一种“硬状态”，软状态则是允许系统中的数据存在中间状态，并认为该状态不影响系统的整体可用性，即允许系统在多个不同节点的数据副本存在数据延时。

最终一致性

数据不可能一直是软状态，必须在一个时间期限之后达到各个节点的一致性，在期限过后，应当保证所有副本保持数据一致性，也就是达到数据的最终一致性。

在系统设计中，最终一致性实现的时间取决于网络延时、系统负载、不同的存储选型、不同数据复制方案设计等因素。

CAP 及 Base 的关系

Base 理论是在 CAP 上发展的，CAP 理论描述了分布式系统中数据一致性、可用性、分区容错性之间的制约关系，当你选择了其中的两个时，就不得不对剩下的一个做一定程度的牺牲。

Base 理论则是对 CAP 理论的实际应用，也就是在分区和副本存在的前提下，通过一定的系统设计方案，放弃强一致性，实现基本可用，这是大部分分布式系统的选择，比如 NoSQL 系统、微服务架构。

2. 分布式事务模型

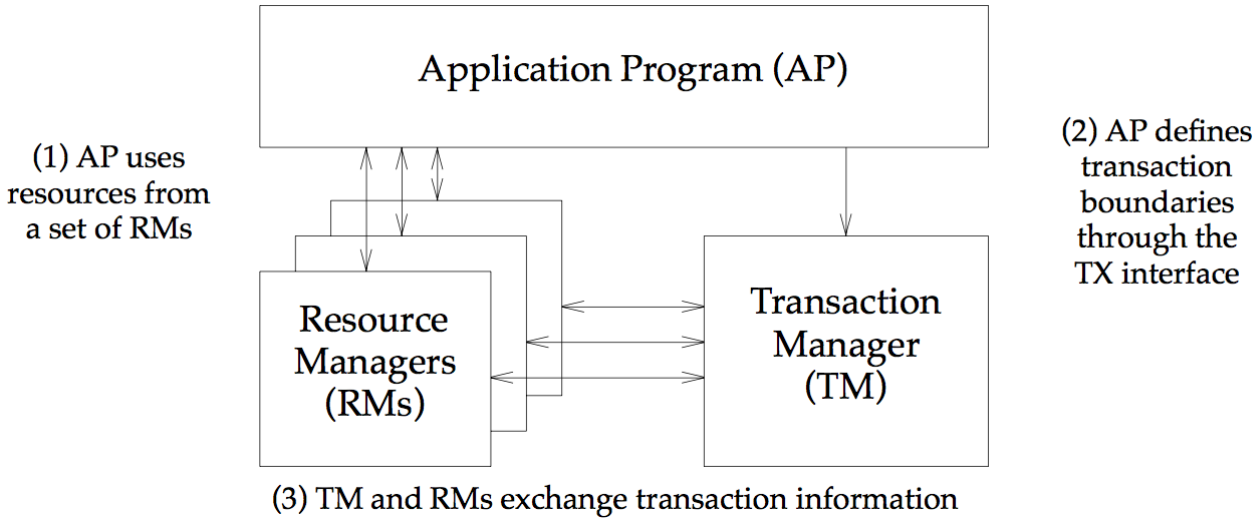
2.1 强一致性模型

强一致性事务主要用于对数据一致性要求比较高， 在任意时刻都能查询出最新写入的数据的场景，比如跨行转账。

DTP模型

DTP模型是X/Open组织定义的一套分布式事务标准。这套标准主要定义了实现分布式事务的规范和API接口，具体的实现则交给相应的厂商来实现。

> DTP 参考模型： **<<Distributed Transaction Processing: Reference Model>>**
> DTP XA规范： **<< Distributed Transaction Processing: The XA Specification>>**



在DTP模型中定义了3个核心组件:

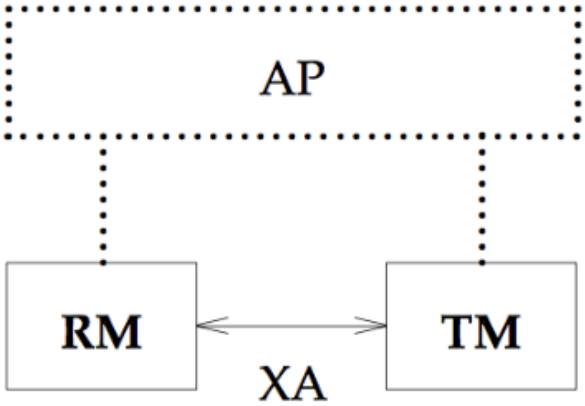
- **AP(Application Program)**应用程序，AP组件定义了分布式事务（也即全局事务）的边界（即事务的开始和结束）以及组成事务的具体操作（Actions）；
- **RM(Resource Managers)**资源管理器，RM指的是诸如MySQL、Oracle这样的数据库或者相应的数据库驱动或可访问的文件系统或者打印机服务器，用以提供访问数据库资源的接口；
- **TM(Transaction Manager)**事务管理器，TM是分布式事务的协调者，其负责为分布式事务分配事务ID，监控事务的执行过程，负责事务的完成和容错工作。TM管理的分布式事务可以跨多个RM，TM还管理2PC协议，协调分布式事务的提交/回滚决策。

在DTP模型中使用了2种通信规范：

- **TX规范**，其定义了用于在AP组件和TM组件之间的通讯API规范，如tx_begin( )、tx_end()、tx_info()等接口用以开启、结束和查询分布式事务；
- **XA(eXtended Architecture)规范**，其定义了TM和RM之间通信的API规范。XA协议中规定了DTP模型中RM需要提供prepare、commit、rollback接口给TM调用，以实现两阶段提交。

XA规范

XA规范是X/Open 组织针对二阶段提交协议的实现做的规范。目前几乎所有的主流数据库都对XA规范提供了支持。XA规范的最主要的作用是，就是定义了RM-TM的交互接口。



两阶段提交协议（Two Phase Commit）不是在XA规范中提出，但是XA规范对其进行了优化。XA规范对两阶段提交协议有2点优化：

- **只读断言**

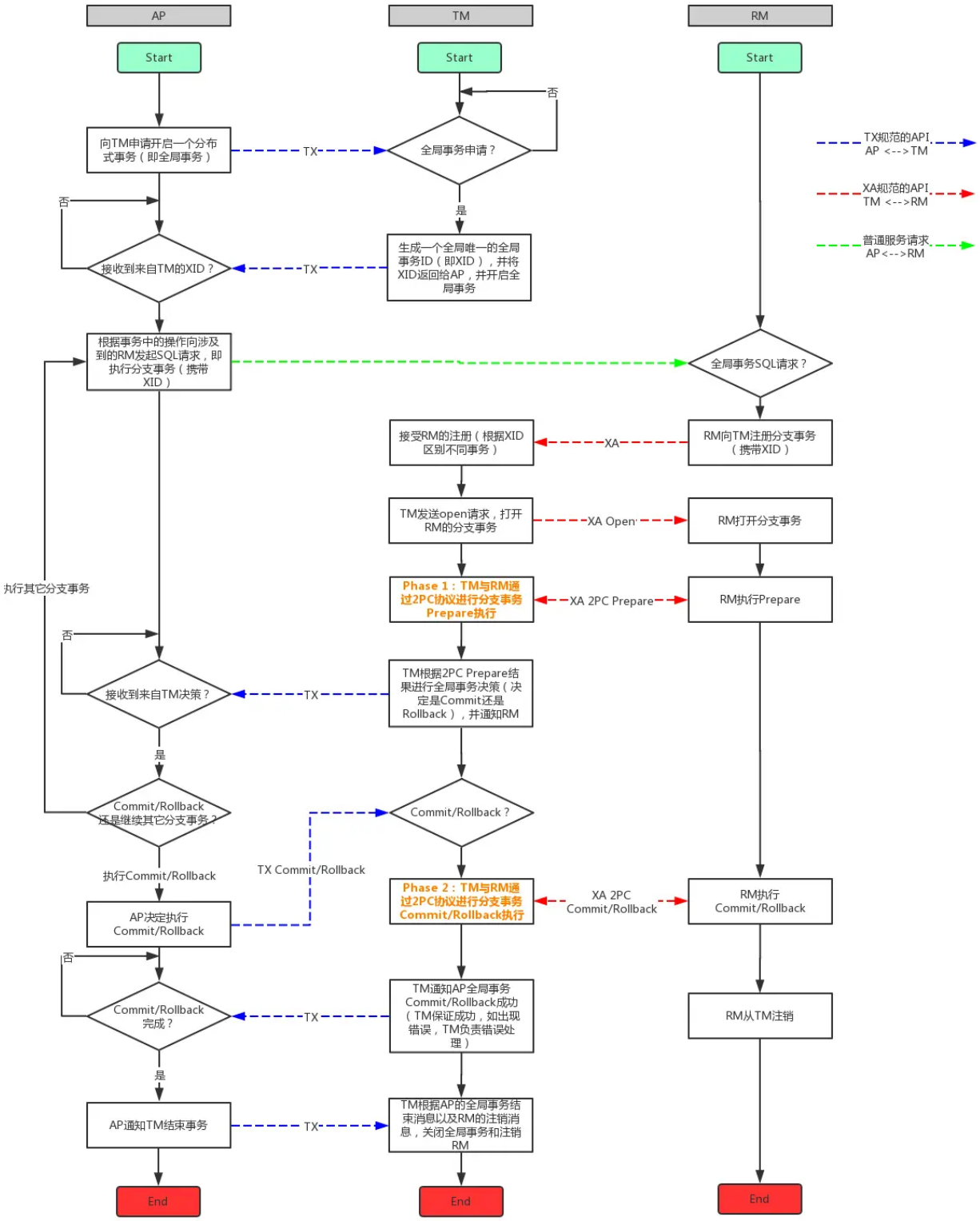
  在Phase 1中，RM可以断言“我这边不涉及数据增删改”来答复TM的prepare请求，从而让这个RM脱离当前的全局事务，从而免去了Phase 2。

- **一阶段提交**

  如果需要增删改的数据都在同一个RM上，TM可以使用一阶段提交——跳过两阶段提交中的Phase 1，直接执行Phase 2。

执行流程

DTP中的分布式事务（即全局事务）的执行流程如下：



（1）首先，在系统中的所有RM通过XA规范提供的API向TM注册，即RM涉及的分支事务处理将纳入到TM统一管理；

（2）然后，AP开启全局事务，AP通过TX规范提供的API向TM申请全局事务的开启，此时，全局事务正式开启。TM会返回其为本次全局事务分配的全局事务ID，即XID，给提出全局事务的AP；

（3）接着，AP根据获取到的XID开始其操作序列，具体的操作就是向不同的RM发送SQL操作请求；

（4）然后，RM根据接受的来自AP的SQL操作请求进行本地数据库操作（即执行本地事务）。RM在执行本地事务时会通过TM提供的XA规范API提交分支事务请求（请求中携带XID）。TM在接收到RM的分支事务请求后，分配分支事务ID并将其反馈给相应的RM；

（5）TM与RM进行基于2PC协议的事务预提交，及进行Prepare操作；

（6）当TM根据RM Prepare操作的执行情况决策出现有分支事务已经全部完成Prepare操作，如果是，那么TM向AP发送Commit决策结果；否则，那么TM向AP发送Rollback决策结果。当进行2PC协议时，如果RM出现故障，TM负责协调故障处理，保证其决策可进行；

（7）当AP接收到TM的决策时，如果后续还有分支事务需要执行，那么AP继续步骤③；否则，AP决策分布式事务的结果是Commit还是Rollback。

（8）如果AP决定Commit/Rollback分布式事务，那么它将通过TX规范API通知TM，最终的执行决定由TM通过2PC协议通知RM完成；

（9）最后，RM向TM提出注销申请，TM注销已注册的RM；AP向TM提交分布式事务结束请求，TM结束分布式事务。

2PC

两阶段提交（Two Phase Commit），就是将提交(commit)过程划分为2个阶段(Phase)：

阶段1：

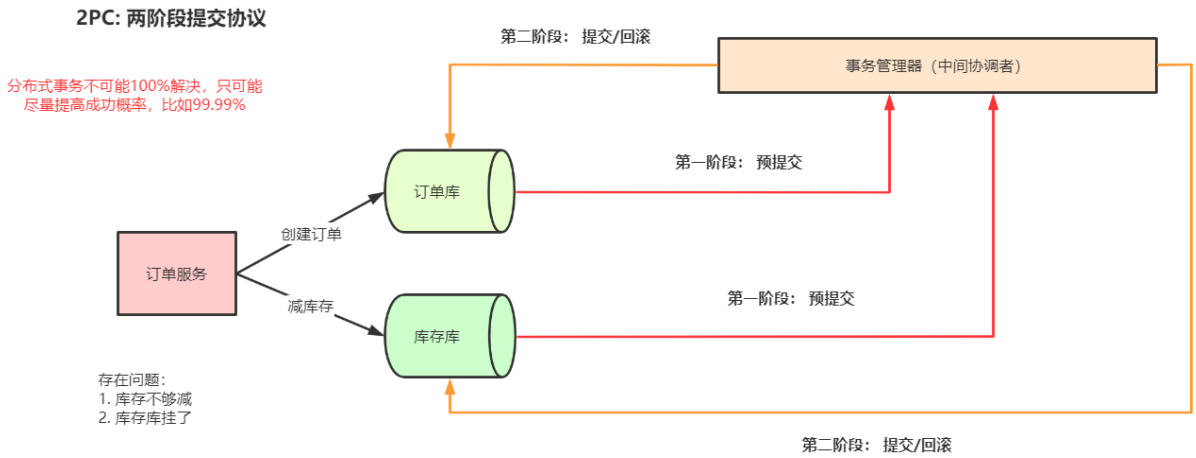
TM（事务管理器）通知各个RM（资源管理器）准备提交它们的事务分支。如果RM判断自己进行的工作可以被提交，那就对工作内容进行持久化，再给TM肯定答复；要是发生了其他情况，那给TM的都是否定答复。

以mysql数据库为例，在第一阶段，事务管理器向所有涉及到的数据库服务器发出prepare"准备提交"请求，数据库收到请求后执行数据修改和日志记录等处理，处理完成后只是把事务的状态改成"可以提交",然后把结果返回给事务管理器。

阶段2

TM根据阶段1各个RM prepare的结果，决定是提交还是回滚事务。如果所有的RM都prepare成功，那么TM通知所有的RM进行提交；如果有RM prepare失败的话，则TM通知所有RM回滚自己的事务分支。

以mysql数据库为例，如果第一阶段中所有数据库都prepare成功，那么事务管理器向数据库服务器发出"确认提交"请求，数据库服务器把事务的"可以提交"状态改为"提交完成"状态，然后返回应答。如果在第一阶段内有任何一个数据库的操作发生了错误，或者事务管理器收不到某个数据库的回应，则认为事务失败，回撤所有数据库的事务。数据库服务器收不到第二阶段的确认提交请求，也会把"可以提交"的事务回撤。



两阶段提交方案下全局事务的ACID特性，是依赖于RM的。一个全局事务内部包含了多个独立的事务分支，这一组事务分支要么都成功，要么都失败。各个事务分支的ACID特性共同构成了全局事务的ACID特性。也就是将单个事务分支支持的ACID特性提升一个层次到分布式事务的范畴。

2PC存在的问题

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

- **同步阻塞问题**

2PC 中的参与者是阻塞的。在第一阶段收到请求后就会预先锁定资源，一直到 commit 后才会释放。

- **单点故障**

由于协调者的重要性，一旦协调者TM发生故障，参与者RM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。

- **数据不一致**

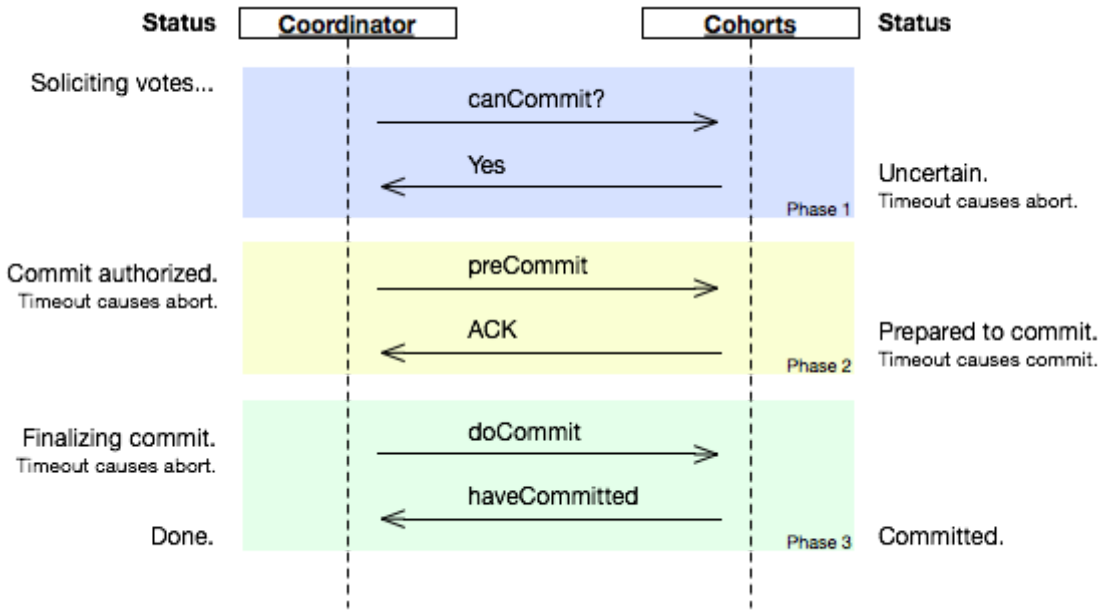
若协调者第二阶段发送提交请求时崩溃，可能部分参与者收到commit请求提交了事务，而另一部分参与者未收到commit请求而放弃事务，从而造成数据不一致的问题。

3PC

三阶段提交（3PC)，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：

- 引入超时机制。同时在协调者和参与者中都引入超时机制。
- 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记性事务的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。

  - **Case 1：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。

  - **Case 2：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。

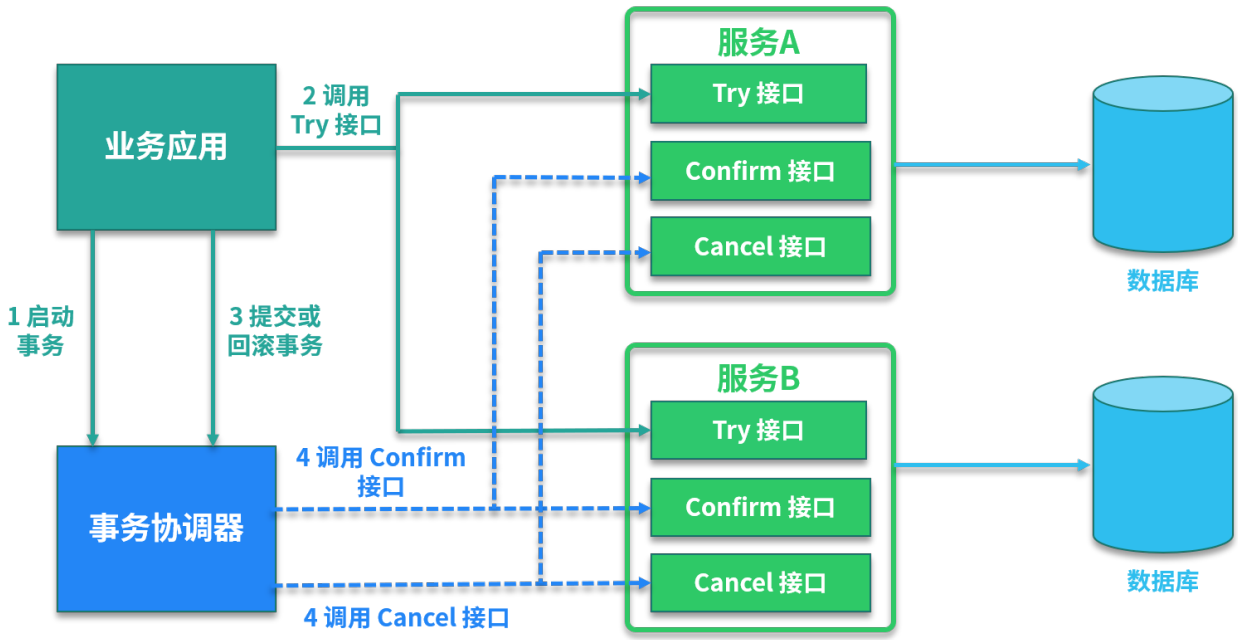
相对于2PC，3PC主要解决的单点故障问题，并减少阻塞，因为一旦参与者无法及时收到来自协调者的信息之后，他会默认执行commit。而不会一直持有事务资源并处于阻塞状态。但是这种机制也会导致数据一致性问题，因为，由于网络原因，协调者发送的abort响应没有及时被参与者接收到，那么参与者在等待超时之后执行了commit操作。这样就和其他接到abort命令并执行回滚的参与者之间存在数据不一致的情况。

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。

2.2 最终一致性模型

TCC

TCC是Try-Confirm-Cancel的简称:



Try 阶段：调用 Try 接口，尝试执行业务，完成所有业务检查，预留业务资源。

Confirm 或 Cancel 阶段：两者是互斥的，只能进入其中一个，并且都满足幂等性，允许失败重试。

- Confirm 操作：对业务系统做确认提交，确认执行业务操作，不做其他业务检查，只使用 Try 阶段预留的业务资源。
- Cancel 操作：在业务执行错误，需要回滚的状态下执行业务取消，释放预留资源。

Try 阶段失败可以 Cancel，如果 Confirm 和 Cancel 阶段失败了怎么办？

TCC 中会添加事务日志，如果 Confirm 或者 Cancel 阶段出错，则会进行重试，所以这两个阶段需要支持幂等；如果重试失败，则需要人工介入进行恢复和处理等。

TCC VS XA

XA是资源层面的分布式事务，强一致性，在两阶段提交的整个过程中，一直会持有资源的锁。TCC是业务层面的分布式事务，最终一致性，不会一直持有资源的锁。

1) 在阶段1：

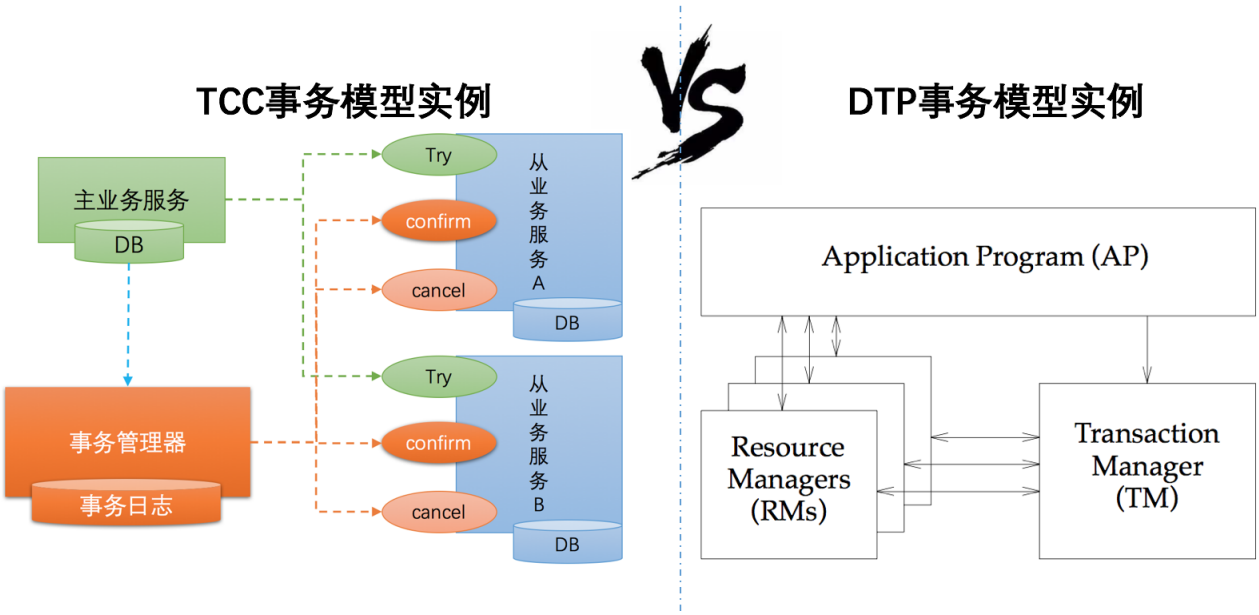
在XA中，各个RM准备提交各自的事务分支，事实上就是准备提交资源的更新操作(insert、delete、update等)；而在TCC中，是主业务活动请求(try)各个从业务服务预留资源。

2) 在阶段2：

XA根据第一阶段每个RM是否都prepare成功，判断是要提交还是回滚。如果都prepare成功，那么就commit每个事务分支，反之则rollback每个事务分支。

TCC中，如果在第一阶段所有业务资源都预留成功，那么confirm各个从业务服务，否则取消(cancel)所有从业务服务的资源预留请求。

TCC VS DTP



- TCC模型中的主业务服务 相当于 DTP模型中的AP，TCC模型中的从业务服务 相当于 DTP模型中的RM
- TCC模型中，从业务服务提供的try、confirm、cancel接口， 相当于 DTP模型中RM提供的prepare、commit、rollback接口
- DTP模型和TCC模型中都有一个事务管理器。不同的是：
    - 在DTP模型中，阶段1的(prepare)和阶段2的(commit、rollback)，都是由TM进行调用的。
    - 在TCC模型中，阶段1的try接口是主业务服务调用(绿色箭头)，阶段2的(confirm、cancel接口)是事务管理器TM调用(红色箭头)。

可靠消息最终一致性

异步化在分布式系统设计中随处可见，基于消息队列的最终一致性就是一种异步事务机制，在业务中广泛应用。

本地消息表

本地消息表的方案最初是由 ebay 的工程师提出，核心思想是将分布式事务拆分成本地事务进行处理，通过消息日志的方式来异步执行。

本地消息表是一种业务耦合的设计，消息生产方需要额外建一个事务消息表，并记录消息发送状态，消息消费方需要处理这个消息，并完成自己的业务逻辑，另外会有一个异步机制来定期扫描未完成的消息，确保最终一致性。

下面我们用下单减库存业务来简单模拟本地消息表的实现过程：

（1）系统收到下单请求，将订单业务数据存入到订单库中，并且同时存储该订单对应的消息数据，比如购买商品的 ID 和数量，消息数据与订单库为同一库，更新订单和存储消息为一个本地事务，要么都成功，要么都失败。

（2）库存服务通过消息中间件收到库存更新消息，调用库存服务进行业务操作，同时返回业务处理结果。

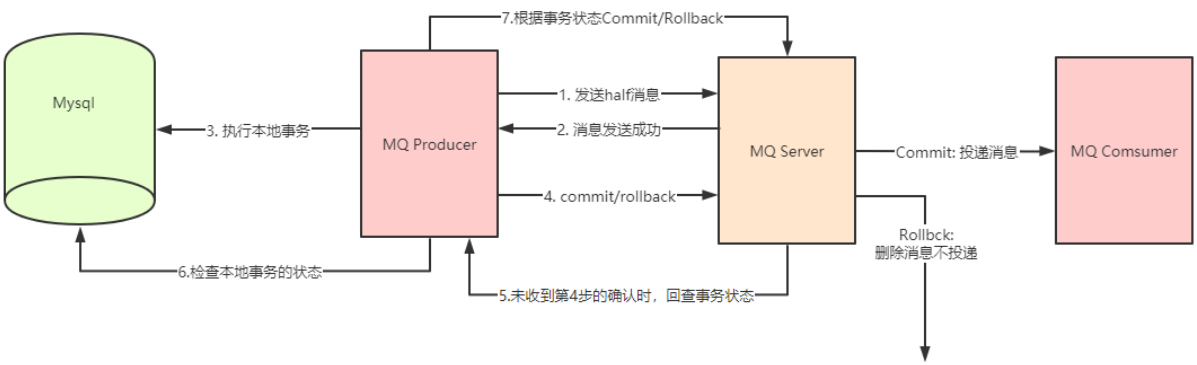
（3）消息生产方，也就是订单服务收到处理结果后，将本地消息表的数据删除或者设置为已完成。

（4）设置异步任务，定时去扫描本地消息表，发现有未完成的任务则重试，保证最终一致性。

RocketMQ 事务消息

RocketMQ 事务消息是一种支持分布式事务的消息。它通过引入 `prepare`、`commit` 和 `rollback` 三个阶段，来确保事务消息的一致性。

- prepare 阶段：消息发送方发送半消息，此时消息的状态为“待提交”。
- commit 阶段：消息发送方向 RocketMQ 发送 commit 消息请求，RocketMQ 判断此时半消息是否被确认，如果半消息已被确认，则将消息标记为“可消费”并提交事务。如果半消息未被确认，则将消息标记为“不可消费”并终止事务。
- rollback 阶段：消息发送方向 RocketMQ 发送 rollback 消息请求，RocketMQ 将半消息标记为“不可消费”并回滚事务。

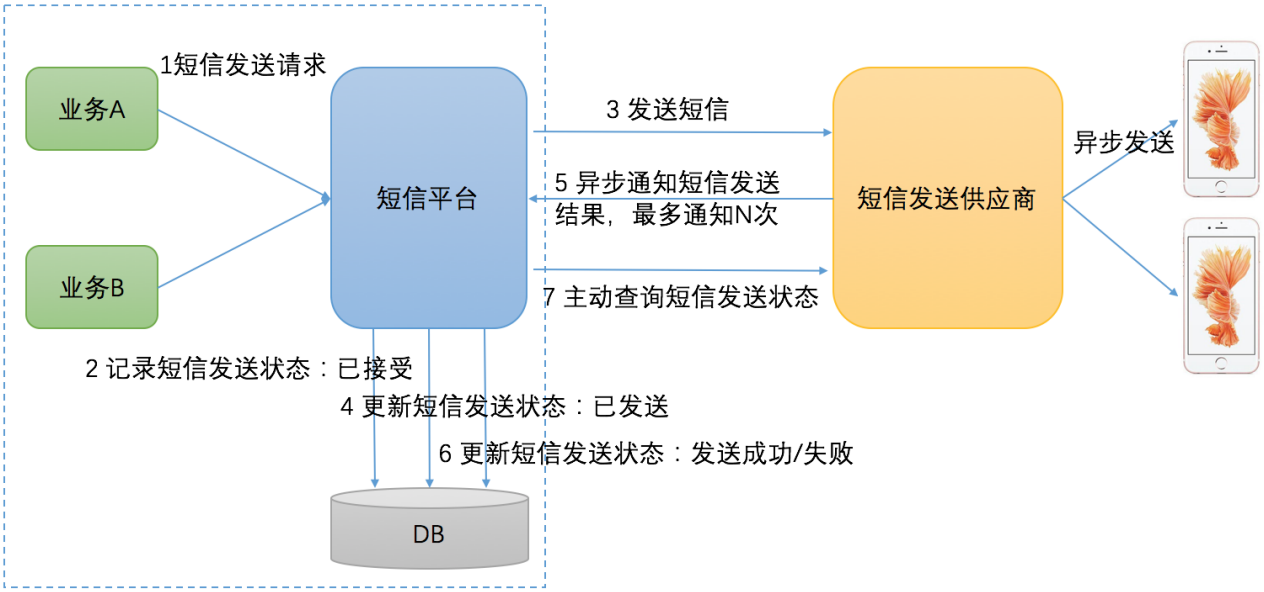


最大努力通知

最大努力通知型(Best-effort delivery)是最简单的一种柔性事务，适用于一些最终一致性时间敏感度低的业务，且被动方处理结果 不影响主动方的处理结果。典型的使用场景：如银行通知、商户通知等。最大努力通知型的实现方案，一般符合以下特点：

1、不可靠消息：业务活动主动方，在完成业务处理之后，向业务活动的被动方发送消息，直到通知N次后不再通知，允许消息丢失(不可靠消息)。

2、定期校对：业务活动的被动方，根据定时策略，向业务活动主动方查询(主动方提供查询接口)，恢复丢失的业务消息。



3. 数据一致性算法

3.1 Paxos算法

兰伯特提出的 Paxos 算法包含 2 个部分：

- 一个是 Basic Paxos 算法，描述的是多节点之间如何就某个值（提案 Value）达成共识；
- 另一个是 Multi-Paxos 思想，描述的是执行多个 Basic Paxos 实例，就一系列值达成共识。

> https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf

Quorum 机制

在学习 Paxos 算法之前，我们先来看分布式系统中的 Quorum 选举算法。在各种一致性算法中都可以看到Quorum 机制的身影，主要数学思想来源于抽屉原理，用一句话解释那就是，在 N 个副本中，一次更新成功的如果有 W 个，那么我在读取数据时是要从大于 N - W 个副本中读取，这样就能至少读到一个更新的数据了。

Quorum 机制的使用需要配合一个获取最新成功提交的版本号的 metadata 服务，这样可以确定最新已经成功提交的版本号，然后从已经读到的数据中就可以确认最新写入的数据。

Quorum 是分布式系统中常用的一种机制，用来保证数据冗余和最终一致性的投票算法，在 Paxos、Raft 和 ZooKeeper 的 Zab 等算法中，都可以看到 Quorum 机制的应用。

Paxos 的节点角色

在 Paxos 协议中，有三类节点角色，分别是 Proposer、Acceptor 和 Learner。

Proposer 提案者

提出提案 (Proposal)。Proposal信息包括提案编号 (Proposal ID) 和提议的值 (Value)。

Acceptor 批准者（接受者）

参与决策，回应Proposers的提案。在集群中，Acceptor 有 N 个，Acceptor 之间完全对等独立，Proposer 提出的 value 必须获得超过半数（N/2+1）的 Acceptor 批准后才能通过。

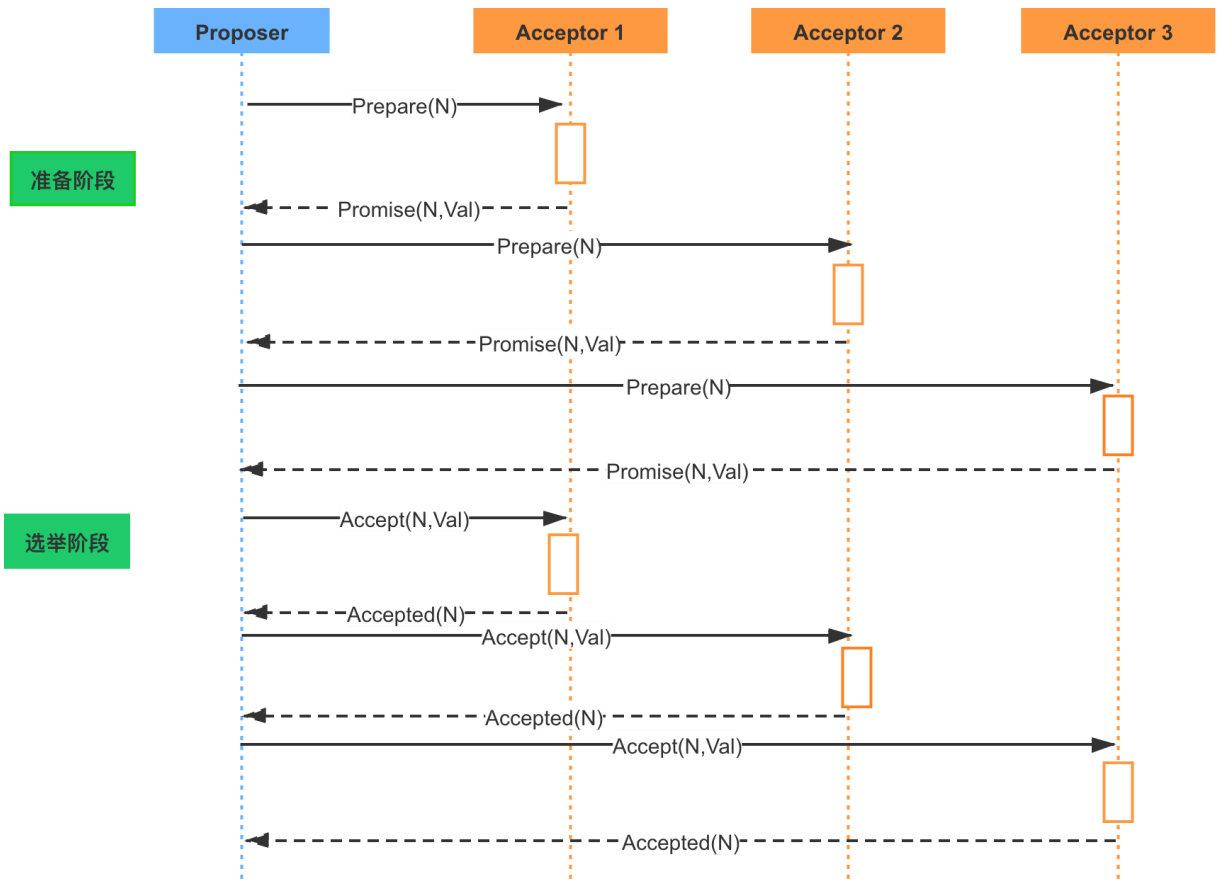
Learner 学习者

不参与决策，从Proposers/Acceptors学习最新达成一致的提案（Value）

Proposer 和 Acceptor 是算法核心角色，Paxos 描述的就是在一个由多个 Proposer 和多个 Acceptor 构成的系统中，如何让多个 Acceptor 针对 Proposer 提出的多种提案达成一致的过程，而 Learner 只是“学习”最终被批准的提案。

Paxos 选举过程

选举过程可以分为两个部分，准备阶段和选举阶段，可以查看下面的时序图：



Phase 1 准备阶段

Proposer 生成全局唯一且递增的 ProposalID，向 Paxos 集群的所有机器发送 Prepare 请求，这里不携带 value，只携带 N 即 ProposalID。

Acceptor 收到 Prepare 请求后，判断收到的 ProposalID 是否比之前已响应的所有提案的 N 大，如果是，则：

- 在本地持久化 N，可记为 Max_N；
- 回复请求，并带上已经 Accept 的提案中 N 最大的 value，如果此时还没有已经 Accept 的提案，则返回 value 为空；
- 做出承诺，不会 Accept 任何小于 Max_N 的提案。

如果否，则不回复或者回复 Error。

Phase 2 选举阶段

为了方便描述，我们把 Phase 2 选举阶段继续拆分为 P2a、P2b 和 P2c。

P2a：Proposer 发送 Accept

经过一段时间后，Proposer 收集到一些 Prepare 回复，有下列几种情况：

- 若回复数量 > 一半的 Acceptor 数量，且所有回复的 value 都为空时，则 Porposer 发出 accept 请求，并带上自己指定的 value。
- 若回复数量 > 一半的 Acceptor 数量，且有的回复 value 不为空时，则 Porposer 发出 accept 请求，并带上回复中 ProposalID 最大的 value，作为自己的提案内容。
- 若回复数量 <= 一半的 Acceptor 数量时，则尝试更新生成更大的 ProposalID，再转到准备阶段执行。

P2b：Acceptor 应答 Accept

Accpetor 收到 Accpet 请求 后，判断：

- 若收到的 N >= Max_N（一般情况下是等于），则回复提交成功，并持久化 N 和 value；
- 若收到的 N < Max_N，则不回复或者回复提交失败。

P2c: Proposer 统计投票

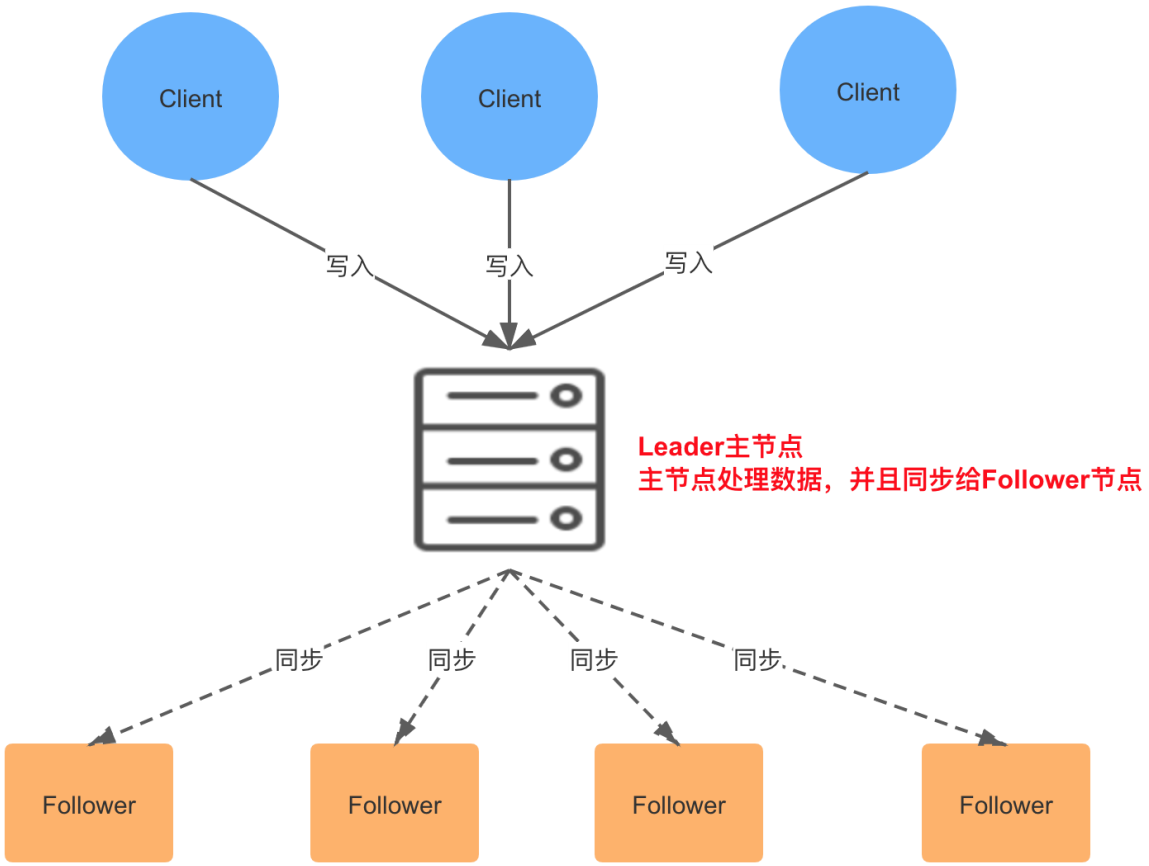
经过一段时间后，Proposer 会收集到一些 Accept 回复提交成功的情况，比如：

- 当回复数量 > 一半的 Acceptor 数量时，则表示提交 value 成功，此时可以发一个广播给所有的 Proposer、Learner，通知它们已 commit 的 value；
- 当回复数量 <= 一半的 Acceptor 数量时，则尝试更新生成更大的 ProposalID，转到准备阶段执行。

当收到一条提交失败的回复时，则尝试更新生成更大的 ProposalID，也会转到准备阶段执行。

3.2 Zab 一致性协议

ZooKeeper 是通过 Zab 协议来保证分布式事务的最终一致性。Zab（ZooKeeper Atomic Broadcast，ZooKeeper 原子广播协议）支持崩溃恢复，基于该协议，ZooKeeper 实现了一种主备模式的系统架构来保持集群中各个副本之间数据一致性。



在 ZooKeeper 集群中，所有客户端的请求都是写入到 Leader 进程中的，然后，由 Leader 同步到其他节点，称为 Follower。在集群数据同步的过程中，如果出现 Follower 节点崩溃或者 Leader 进程崩溃时，都会通过 Zab 协议来保证数据一致性。

Zab 协议的具体实现可以分为以下两部分：

消息广播阶段

Leader 节点接受事务提交，并且将新的 Proposal 请求广播给 Follower 节点，收集各个节点的反馈，决定是否进行 Commit，在这个过程中，也会使用前面提到的 Quorum 选举机制。

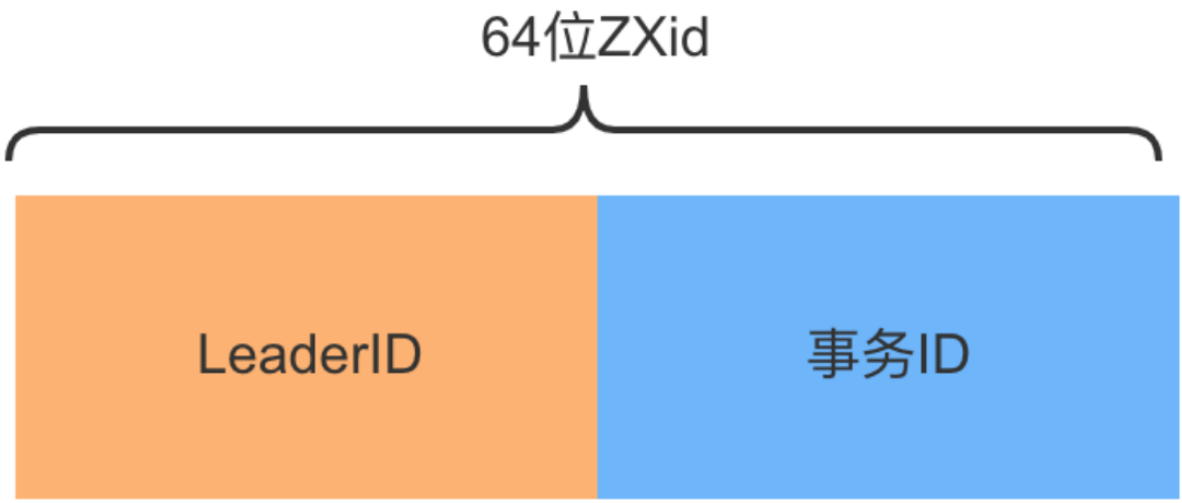
崩溃恢复阶段

如果在同步过程中出现 Leader 节点宕机，会进入崩溃恢复阶段，重新进行 Leader 选举，崩溃恢复阶段还包含数据同步操作，同步集群中最新的数据，保持集群的数据一致性。

整个 ZooKeeper 集群的一致性保证就是在上面两个状态之前切换，当 Leader 服务正常时，就是正常的消息广播模式；当 Leader 不可用时，则进入崩溃恢复模式，崩溃恢复阶段会进行数据同步，完成以后，重新进入消息广播阶段。

Zxid

Zxid 是 Zab 协议的一个事务编号，Zxid 是一个 64 位的数字，其中低 32 位是一个简单的单调递增计数器，针对客户端每一个事务请求，计数器加 1；而高 32 位则代表 Leader 周期年代的编号。



Zab 流程分析

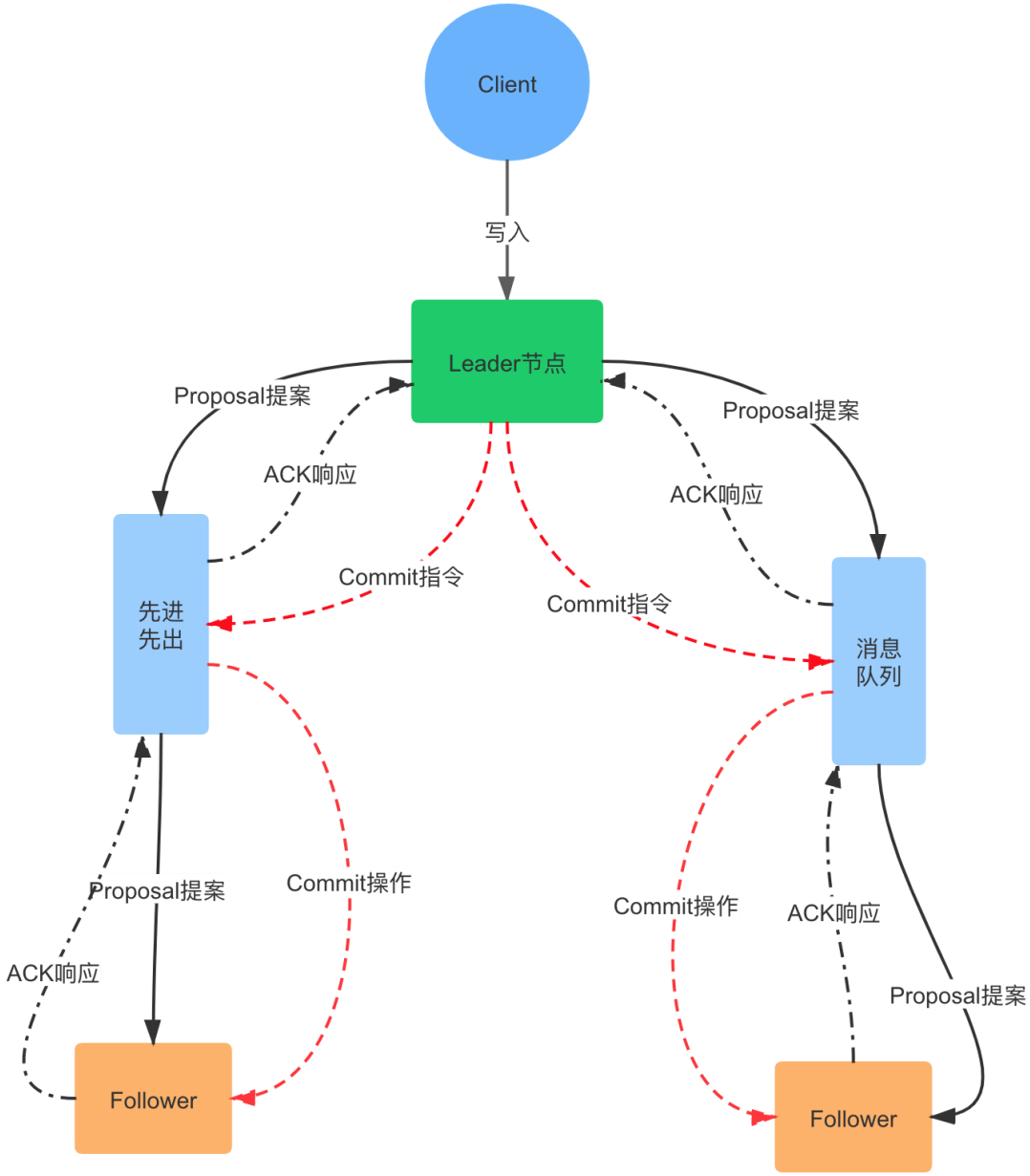
Zab 的具体流程可以拆分为消息广播、崩溃恢复和数据同步三个过程，下面我们分别进行分析。

消息广播

在 ZooKeeper 中所有的事务请求都由 Leader 节点来处理，其他服务器为 Follower，Leader 将客户端的事务请求转换为事务 Proposal，并且将 Proposal 分发给集群中其他所有的 Follower。

完成广播之后，Leader 等待 Follwer 反馈，当有过半数的 Follower 反馈信息后，Leader 将再次向集群内 Follower 广播 Commit 信息，Commit 信息就是确认将之前的 Proposal 提交。

Leader 节点的写入也是一个两步操作，第一步是广播事务操作，第二步是广播提交操作，其中过半数指的是反馈的节点数 >=N/2+1，N 是全部的 Follower 节点数量。



- 客户端的写请求进来之后，Leader 会将写请求包装成 Proposal 事务，并添加一个递增事务 ID，也就是 Zxid，Zxid 是单调递增的，以保证每个消息的先后顺序；
- 广播这个 Proposal 事务，Leader 节点和 Follower 节点是解耦的，通信都会经过一个先进先出的消息队列，Leader 会为每一个 Follower 服务器分配一个单独的 FIFO 队列，然后把 Proposal 放到队列中；
- Follower 节点收到对应的 Proposal 之后会把它持久到磁盘上，当完全写入之后，发一个 ACK 给 Leader；
- 当 Leader 收到超过半数 Follower 机器的 ack 之后，会提交本地机器上的事务，同时开始广播 commit，Follower 收到 commit 之后，完成各自的事务提交。

崩溃恢复

消息广播通过 Quorum 机制，解决了 Follower 节点宕机的情况，但是如果在广播过程中 Leader 节点崩溃呢?

这就需要 Zab 协议支持的崩溃恢复，崩溃恢复可以保证在 Leader 进程崩溃的时候可以重新选出 Leader，并且保证数据的完整性。

崩溃恢复和集群启动时的选举过程是一致的，也就是说，下面的几种情况都会进入崩溃恢复阶段：

- 初始化集群，刚刚启动的时候
- Leader 崩溃，因为故障宕机
- Leader 失去了半数的机器支持，与集群中超过一半的节点断连

崩溃恢复模式将会开启新的一轮选举，选举产生的 Leader 会与过半的 Follower 进行同步，使数据一致，当与过半的机器同步完成后，就退出恢复模式， 然后进入消息广播模式。

Zab 中的节点有三种状态，伴随着的 Zab 不同阶段的转换，节点状态也在变化：

- **following**: 当前节点是跟随者，服从Leader节点的命令
- **leading:** 当前节点是Leader,负责协调事务
- **looking:** 节点处于选举状态

我们通过一个模拟的例子，来了解崩溃恢复阶段，也就是选举的流程。

假设正在运行的集群有五台 Follower 服务器，编号分别是 Server1、Server2、Server3、Server4、Server5，当前 Leader 是 Server2，若某一时刻 Leader 挂了，此时便开始 Leader 选举。

选举过程如下：

1.各个节点变更状态，变更为 Looking

ZooKeeper 中除了 Leader 和 Follower，还有 Observer 节点，Observer 不参与选举，Leader 挂后，余下的 Follower 节点都会将自己的状态变更为 Looking，然后开始进入 Leader 选举过程。

2.各个 Server 节点都会发出一个投票，参与选举

在第一次投票中，所有的 Server 都会投自己，然后各自将投票发送给集群中所有机器，在运行期间，每个服务器上的 Zxid 大概率不同。

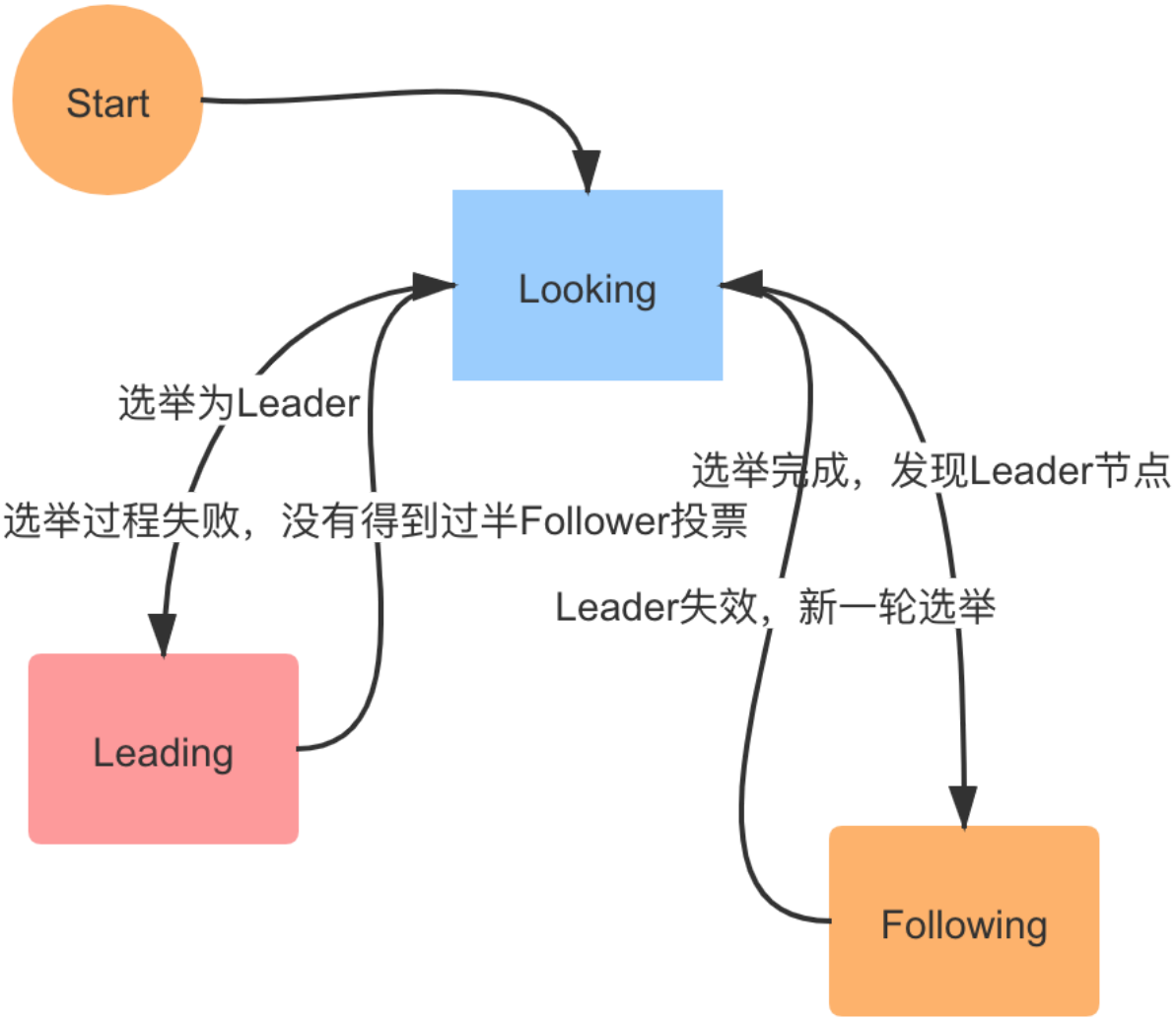
3.集群接收来自各个服务器的投票，开始处理投票和选举

处理投票的过程就是对比 Zxid 的过程，假定 Server3 的 Zxid 最大，Server1 判断 Server3 可以成为 Leader，那么 Server1 就投票给 Server3，判断的依据如下：

- 首先选举 epoch 最大的
- 如果 epoch 相等，则选 zxid 最大的
- 若 epoch 和 zxid 都相等，则选择 server id 最大的，就是配置 zoo.cfg 中的 myid

在选举过程中，如果有节点获得超过半数的投票数，则会成为 Leader 节点，反之则重新投票选举。

4.选举成功，改变服务器的状态，参考下面这张图的状态变更



数据同步

崩溃恢复完成选举以后，接下来的工作就是数据同步，在选举过程中，通过投票已经确认 Leader 服务器是最大Zxid 的节点，同步阶段就是利用 Leader 前一阶段获得的最新Proposal历史，同步集群中所有的副本。

3.3 Raft算法

Raft 算法属于 Multi-Paxos 算法，它是在兰伯特 Multi-Paxos 思想的基础上，做了一些简化和限制，比如增加了日志必须是连续的，只支持领导者、跟随者和候选人三种状态，在理解和算法实现上都相对容易许多。**Raft 算法是现在分布式系统开发首选的共识算法。**

从本质上说，Raft 算法是通过一切以领导者为准的方式，实现一系列值的共识和各节点日志的一致。

领导者就是 Raft 算法中的霸道总裁，Raft 算法是强领导者模型，集群中只能有一个“霸道总裁”。

> https://raft.github.io/
>
> http://thesecretlivesofdata.com/raft/

Raft 算法支持 3 种状态：

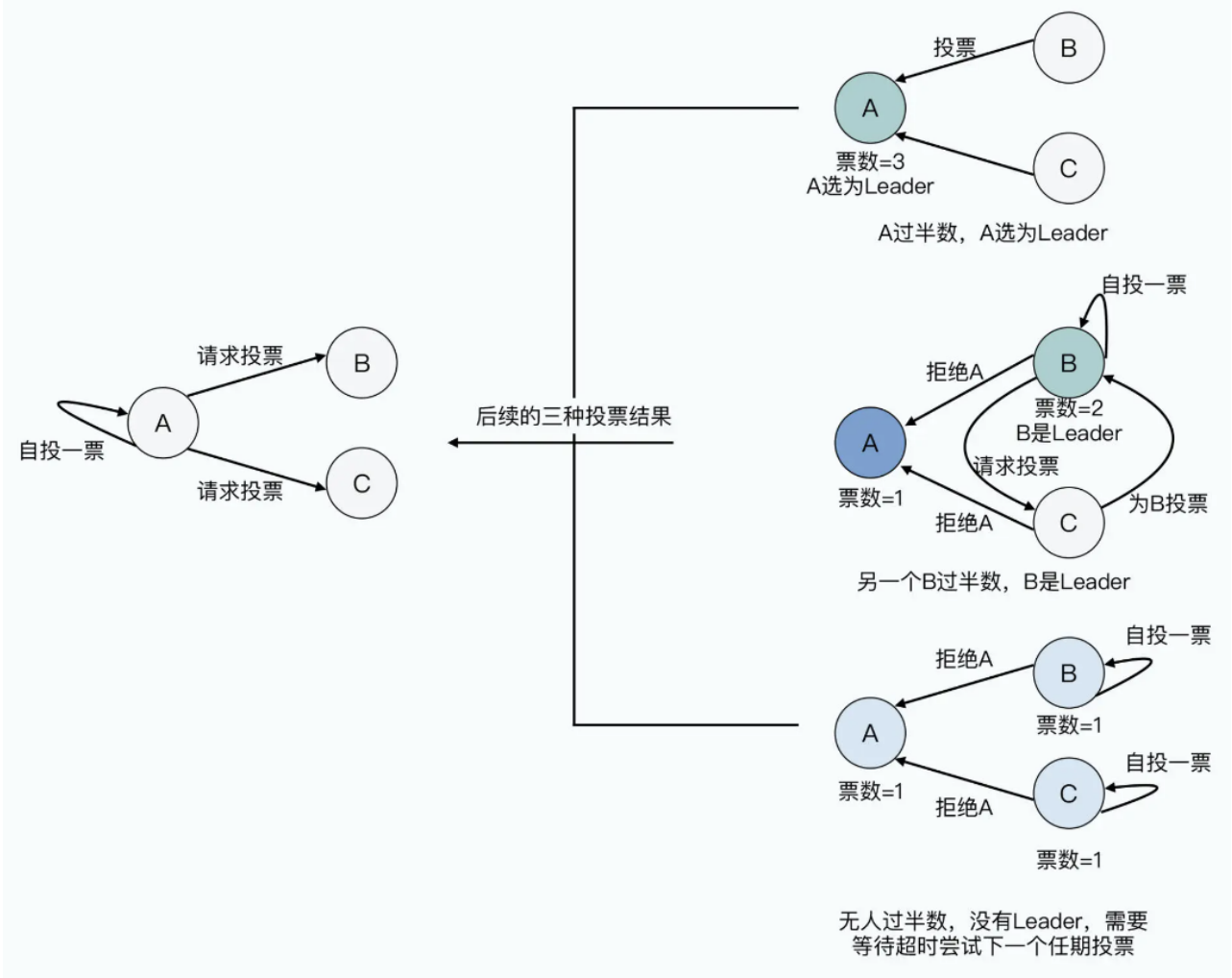
- **领导者（Leader）**：蛮不讲理的霸道总裁，一切以我为准，平常的主要工作内容就是 3 部分，处理写请求、管理日志复制和不断地发送心跳信息，通知其他节点“我是领导者，我还活着，你们现在不要发起新的选举，找个新领导者来替代我。”
- **跟随者（Follower）**：就相当于普通群众，默默地接收和处理来自领导者的消息，当等待领导者心跳信息超时的时候，就主动站出来，推荐自己当候选人。
- **候选人（Candidate）**：候选人将向其他节点发送请求投票（RequestVote）RPC 消息，通知其他节点来投票，如果赢得了大多数选票，就晋升当领导者。

和 Paxos 不同的是，一致性问题，被 Raft 明确拆解成了三个比较独立的、更好理解的子问题：Leader 选举、日志复制、安全性。在 Leader 选举上，Raft 采用的是典型的心跳检测 Leader 存活，以及随机超时时间投票，确保不会死循环，来确保我们可以快速选出 Leader。在日志复制层面，Raft 采用的是一个典型的两阶段提交。为了让算法简单，它直接在日志复制过程中，强制 Follower 和 Leader 同步，来解决删除未提交的脏日志的办法。而为了做到这一点，Raft 又需要确保无论 Leader 如何通过选举切换，都需要包含最新已提交的日志的办法。而要确保这一点，它也只是在 Leader 选举的时候，带上了日志索引和任期信息就简单地解决了。

Leader选举

Raft 的 Leader 选举是这样的：

- 在 Raft 里，Leader 会定期向 Follower 发送心跳。Follower 也会设置一个超时时间，如果超过超时时间没有接收到心跳，那么它就会认为 Leader 可能挂掉了，就会发起一轮新的 Leader 选举。
- Follower 发起选举，会做两个动作。第一个，是先给自己投一票；第二个，是向所有其他的 Follower 发起一个 RequestVote 请求，也就是要求那些 Follower 为自己投票。这个时候，Follower 的角色，就转变成了 Candidate。
- 在每一个 RequestVote 的请求里，除了有发起投票的服务器信息之外，还有一个任期（Term）字段。这个字段，本质上是一个 Leader 的“版本信息”或者说是“逻辑时钟”。
- 每个 Follower，在本地都会保留当前 Leader 是哪一个任期的。当它要发起投票的时候，会把任期自增 1，和请求一起发出去。
- 其他 Follower 在接收到 RequestVote 请求的时候，会去对比请求里的任期和本地的任期。如果请求的任期更大，那么它会投票给这个 Candidate。不然，这个请求会被拒绝掉。
- 在一个任期里，一台服务器最多给一个 Candidate 投票，所以投票过程是先到先得，两个服务器都发起了 RequestVote，我们的 Follower 也只能投给一台。

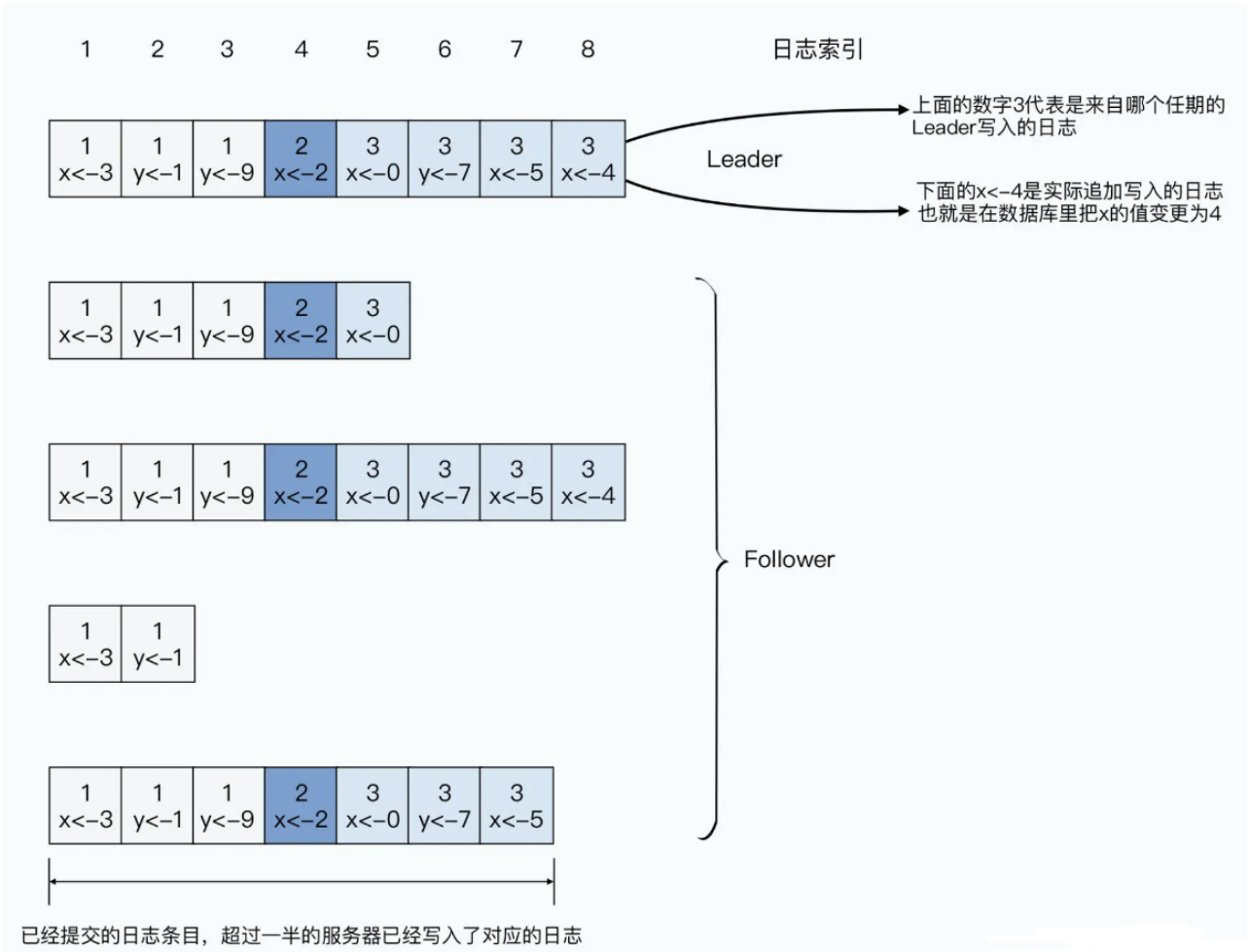


日志复制

Leader 选出来了，那我们自然就可以向 Leader 发送写入数据的请求。整个数据写入，就是一个两阶段提交的过程，只不过这个两阶段提交只需要“半数”通过，就可以发起第二阶段的提交操作，而不需要等待所有服务器都确认可以提交。

我们追加写入的每一条日志，都包含三部分信息：

- 第一个，是日志的索引，这是一个随着数据写入不断自增的字段。
- 第二个，是日志的具体内容，也就是具体的状态机更新的操作是什么。
- 第三个，就是这一次数据写入，来自 Leader 的哪一个任期。



我们在追加写入日志的时候，不只是单单追加最新的一条日志，还需要做一个校验，确保对应的 Follower 的数据和 Leader 是一致的：

- 首先，在发起追加写入日志的复制请求的时候，Leader 的 AppendEntries 的 RPC 里，不仅会有最新的一条日志，还会有上一条日志里的日志索引和任期信息。
- Follower 会先对比日志索引和任期信息，在自己的日志里寻找相同索引和任期的日志所在的位置。
  - 如果找到了，Follower 会把这个位置之后的日志都删除掉。然后把新日志追加上去。
  - 如果找不到，Follower 会拒绝新追加的日志。然后，Leader 就知道，这个 Follower 没有同步到最新的日志，那么 Leader 会在自己的日志里，找到前一条日志，再重新发送给 Follower。
- Leader 会不断重复这个过程，确保找到和 Follower 的同步点，找到了之后，就会把这个位置之后的日志都删除掉，然后把同步点之后的日志一条条复制过去。

安全性

在 Leader 挂掉，切换新 Leader 之后，我们会遇到一个挑战，新的 Leader 可能没有同步到最新的日志写入。而这可能会导致，新的 Leader 会尝试覆盖之前 Leader 已经写入的数据。这个问题就是我们需要解决的第三个问题，也就是“安全性”问题。

Raft 里，每一个服务器写入的日志，会有两种状态，一种是未提交的，一种是已提交的。我们想要确保 Leader 的日志是最新的（已提交的日志），只需要在 Leader 选举的时候，让只有最新日志的 Leader 才能被选上就好了。

Raft 是这么做的：

- 在 RequestVote 的请求里，除了预期的下一个任期之外，还要带上 Candidate 已提交的日志的最新的索引和任期信息。
- 每一个 Follower，也会比较本地已提交的日志的最新的索引和任期信息。
- 如果 Follower 本地有更新的数据，那么它会拒绝投票。

3.4 Gossip协议

Gossip协议也可以称之为流行病算法、流言算法、八卦算法、瘟疫算法，是一种最终一致性的分布式共识协议。原本用于分布式数据库中节点同步数据使用，后被广泛用于数据库复制、信息扩散、集群成员身份确认、故障探测等。Gossip 协议最有名的应用包括 Redis-Cluster，Bitcoin，Cassandra，Consul等。

Gossip 协议主要用于解决大规模去中心化 P2P 网络中的数据一致性问题，其显著特点在于简单易于理解，并且不要求节点间均可以相互通信，只要一个节点能与部分节点通信，那它就可以把数据变更消息广播至整个联通网络中。

六度分隔理论

Gossip Protocol是基于**六度分隔理论（Six Degrees of Separation）哲学**的体现，简单的来说，一个人通过6个中间人可以认识世界任何人，数学公式如下：

$$c = f^w$$

c即复杂度，f为每个人的朋友圈数量，w为人际宽度。

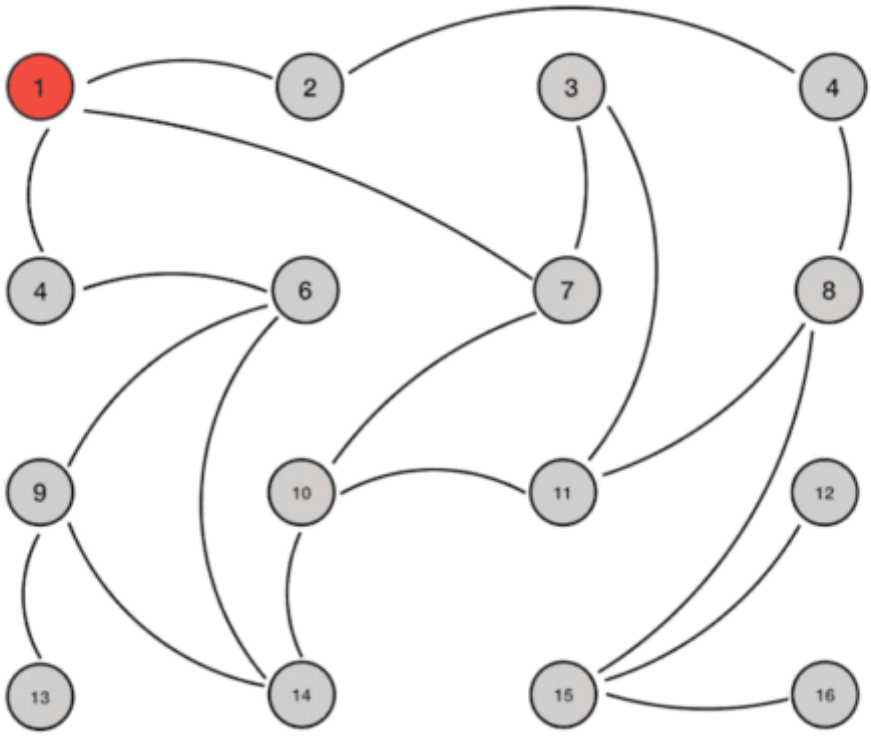
意思为每个人朋友圈数量为150人，当人际宽度为6时，大约可以容纳11.4万亿人的体量

基于六度分隔理论得知，任何信息的传播其实非常迅速，而且网络交互次数不会很多。比如Facebook在2016年2月4号做了一个实验：研究了当时已注册的15.9亿使用者资料，发现这个神奇数字的“网络直径”是4.57，翻成白话文意味着每个人与其他人间隔为4.57人。

工作原理

Gossip 协议执行过程：

- 种子节点周期性的散播消息 【假定把周期限定为 1 秒】。
- 被感染节点随机选择 N 个邻接节点散播消息【假定 fan-out(扇出)设置为 6，每次最多往 6 个节点散播】。
- 节点只接收消息不反馈结果。
- 每次散播消息都选择尚未发送过的节点进行散播。
- 收到消息的节点不再往发送节点散播：A -> B，那么 B 进行散播的时候，不再发给 A。



实现机制

在《Epidemic Algorithms for Replicated Database Maintenance》论文中主要论述了直接邮寄（direct mail）、反熵传播（anti-entropy）、谣言传播（rumor mongering）三种机制来实现数据更新。
Gossip协议主要是通过反熵传播（anti-entropy）、谣言传播（rumor mongering）实现的。

直接邮寄

每个节点更新都会立即从其变更节点邮寄通知到所有其他节点。

机制	时间复杂度	网络流量	优点	缺点
直接邮寄（direct mail）	O(n)，n为节点数	1**m**n，m为更新消息数，n为节点数	更新效率高	不完全可靠，存在 息传递丢失风险

对于**直接邮寄（direct mail）** 这种方式，主要是当节点有数据更新便开始遍历节点池，遍历发送其他所有节点消息来通知自身节点数据的更新情况，实现算法较为简单。由于是一次性遍历通知，在遇到网络通信故障、节点宕机之后恢复等现实情况时没有办法容错和补偿，这是较为致命性的地方，因此极端情况下它是无法保证分布式环境下各节点数据一致性的。

反熵传播

每个节点都会定期随机选择节点池中的一些节点，通过交换数据内容来解决两者之间的任何差异。如下图T1、T2、T3表示不同时间点各节点通知周边节点数据交互的示例。

反熵传播（anti-entropy）中的节点只有两种状态，病原（Suspective）和感染（Infective），因此称作**SI模型**，一般叫做**简易流行病（simple epidemics）**。反熵传播（anti-entropy）的消息数量非常庞大，且无限制；通常只用于新加入节点的数据初始化。

反熵传播（anti-entropy）的消息数量非常庞大，且无限制；通常只用于新加入节点的数据初始化。

机制	时间复杂度	网络流量	优点	缺点
反熵（anti-entropy）	O(log2n)，n为节点数	O((m*n)t)，n为节点数，m为更新消息数，t为周期数	1.可靠 2.定时重复3.可容错	1.消息冗余2.消息延 3.网络流量耗费较多

对于反熵（anti-entropy）这种方式，和直接邮寄（direct mail）相比的最大特点就是解决了消息丢失无法补偿容错导致的数据无法保持一致的致命问题。它通过单点的定时随机通知周边节点进行数据交互的方式保持各节点之间数据的一致性。这里需要注意的是，一致性的保持是在节点数据变更后一段时间内通过节点间的数据交互逐渐完成的最终一致，并且由于每个节点都定期广播数据到周边随机的一部分节点，因此在数据交互上是存在冗余和延迟的。

谣言传播

这里先以**谣言传播**方式来讲解分布式节点数据交互如下：

- 所有的节点在最开始没有产生数据变更时都假设是**未知状态**，它是不知道任何谣言信息的
- 当节点收到其他节点更新数据通知时，相当于听到了一条**谣言**，并将其视为**热门**开始传播给周边节点
- 当某个节点**谣言**盛行时，它会定期随机选择其他节点，并确保另一个节点知道
- 当某个节点发现周边节点都知道这个**谣言**时，该节点将停止将该谣言视为**热点**，并保留更新，而不会进一步传播

谣言传播周期可能比反熵周期更频繁，因为它们在每个站点所需要的资源更少，但是有可能更新不会到达所有站点。

谣言传播（rumor mongering）中的节点状态有Suspective(病原)、Infective(感染)、Removed(愈除)，因此称作**SIR模型**，一般叫做**复杂流行病（complex epidemics）**。

- 消息生产节点即为Suspective(病原)状态
- 消息接收节点即为Infective(感染)状态，会进行消息传播
- 节点接收消息后即为Removed(愈除)状态，不再进行传播

消息只包含最新更新数据，消息在某个时间点之后会被标记为Removed(愈除)状态，并且不再被传播，通常用于节点间数据增量同步。

机制	时间复杂度	网络流量	优点	缺点
谣言传播（rumor mongering）	O(log2n)，n为总节点数	O((m*n)t)，n为节点数(递减)，m为更新消息数，t为周期数	1.可靠 2.定时重复3.可容错	1.消息冗余2.消息延 3.网络流量耗费较多

小结

Gossip是一个**去中心化**的分布式协议，数据通过节点像病毒一样逐个传播，整体传播速度非常快，很像现在全球蔓延的新冠病毒（2019-nCoV）。Gossip的信息传播和扩散通常需要由**种子节点**发起。整个传播过程可能需要一定的时间，由于不能保证某个时刻所有节点都收到消息，但是理论上最终所有节点都会收到消息，因此它是**最终一致性协**

议。Gossip是一个**多主协议**，所有写操作可以由不同节点发起，并且同步给其他副本。Gossip内组成的网络节点都是**对等节点**，是非结构化网络。

优点

- 可扩展性 允许节点的任意增加和减少，新增节点的状态最终会与其他节点一致
- 分布式容错 任意节点的宕机和重启都不会影响 Gossip 消息的传播，具有天然的分布式系统容错特性
- 去中心化 无需中心节点，所有节点都是对等的，任意节点无需知道整个网络状况，只要网络连通，任意节点可把消息散播到全网
- 最终一致性 消息会以一传十、十传百的指数级速度在网络中传播，因此系统状态的不一致可以在很快的时间内收敛到一致，消息传播速度达到了**logN**
- 通俗易懂 算法简单，容易理解，实现成本低

缺点

- 消息延迟 节点随机向少数几个节点发送消息，消息最终是通过多个轮次的散播而到达全网，不可避免的造成消息延迟
- 消息冗余 节点定期随机选择周围节点发送消息，而收到消息的节点也会重复该步骤；不可避免的引起同一节点消息多次接收，增加消息处理压力，一般通过**文件校验和、缓存节点列表**等方式来进行优化减少数据对比带来的性能损耗

基于以上优缺点分析，Gossip协议满足**CAP**分布式理论中基于**AP**场景的数据最终一致性处理，常见应用有：P2P网络通信、Apache Cassandra、Redis Cluster、Consul等，还有Apache Gossip框架的开源实现供Gossip协议的学习。