分布式算法

一、拜占庭将军问题

1. 二忠一叛问题

正常节点之间出现恶意节点，会进行一些干扰信息的转发，此时无法保证正常节点收到一致性的信息。

2. 解决办法

（1）解决办法一：口信消息型拜占庭问题之解

简而言之：增加正常节点的数量，可以保证正常节点接收到一致性的信息。

论文出处：兰伯特《The Byzantine Generals Problem》

详细内容：

> 如果叛将人数为m，将军人数不能少于3m+1，那么拜占庭问题就可以解决了。
>
> **这个算法有个前提，也就是叛将人数 m，或者说能容忍的叛将数 m，是已知的。在这个算法中，叛将数 m 决定递归循环的次数（也就是说，叛将数 m 决定将军们要进行多少轮作战信息协商），即 m+1 轮（所以，你看，只有楚是叛变的，那么就进行了两轮）。你也可以从另外一个角度理解：n 位将军，最多能容忍 (n - 1) / 3 位叛将。**

（2）解决办法二：签名消息型拜占庭问题之解

简而言之：不增加节点的数量，各节点的消息传递通过签名的方式进行。

关于签名：

- 完整性
- 可溯源
- 不可抵赖性

核心内容：

> n 位将军，能容忍 (n - 2) 位叛将（只有一位忠将没有意义，因为此时不需要达成共识了）
>
> 消息的多轮传递链条，确保了所有的忠将能最终收到相同的指令集合，然后指令集合排序，采用一定规则（例如取中间的指令）
>
> **m个叛将，需要进行m+1次，这样的话才能够保证忠将最终接收到相同的指令集合。**

原理：

> a,b,**c**
>
> a和b说，我收到**c**是什么样
>
> b和a说，我收到**c**是什么样
>
> 间接的，a和b都收到了c发给a的信息和c发给b的信息。因此最终的指令集合是一样的。只不过顺序可能不同，因此需要排个序

3. 两类算法

（1）BFT

拜占庭容错算法，最困难的，也是最复杂的，因为除了存在故障行为，还存在恶意行为。

其他：PBFT算法，PoW算法。

（2）CFT

故障容错算法。存在故障行为，但不存在恶意节点的情况下的共识问题。（可能会丢失消息，或者有消息重复，但不存在错误消息，或者伪造消息的情况）

常见算法：Paxos算法，Raft算法，ZAB协议。

4. 场景下算法选择

如果能确定该环境中各节点是可信赖的，不存在篡改消息或者伪造消息等恶意行为（例如 DevOps 环境中的分布式路由寻址系统），推荐使用非拜占庭容错算法；反之，推荐使用拜占庭容错算法，例如在区块链中使用 PoW 算法。

二、CAP理论

1. CAP三指标

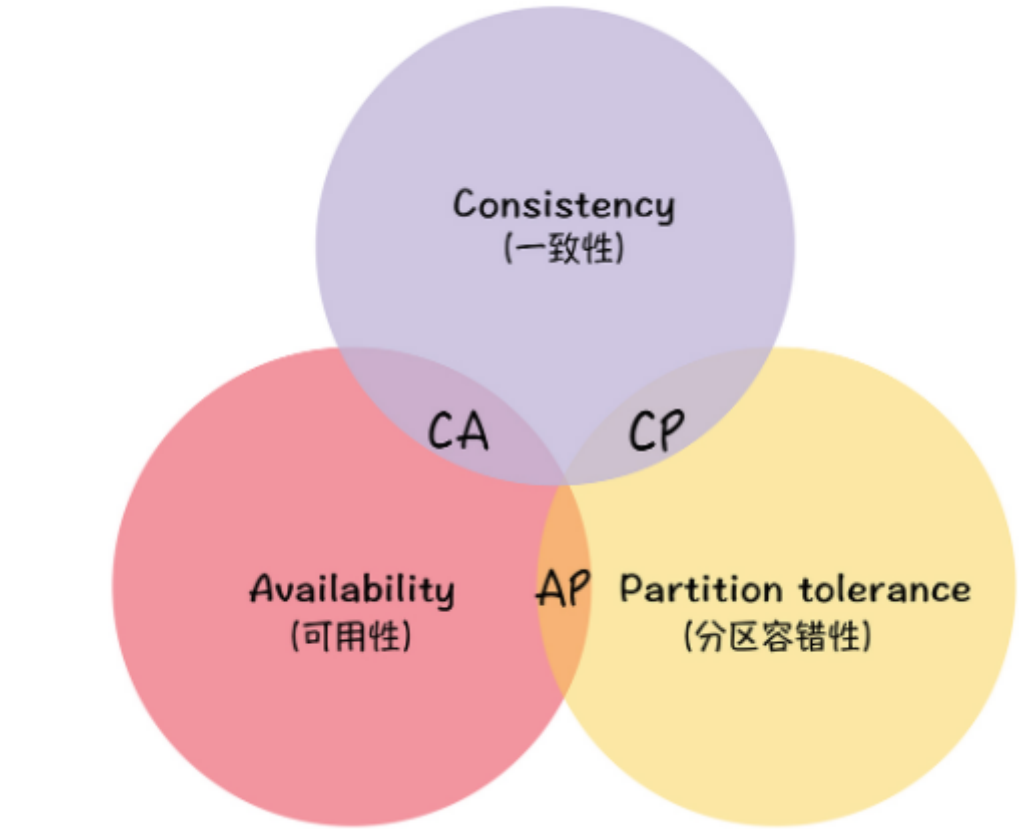
- 一致性（Consistency）
  - 一致性说的是客户端的每次读操作，不管访问哪个节点，要么读到的都是同一份最新写入的数据，要么读取失败。强调的是**数据正确。**
  - 当发生分区故障的时候，有时不能仅仅因为节点间出现了通讯问题，无法响应最新写入的数据，之后在客户端查询数据时，就一直返回给客户端**出错信息。**
- 可用性（Availability）
  - 可用性说的是任何来自客户端的请求，不管访问哪个非故障节点，都能得到响应数据，但不保证是同一份最新数据。
  - 这个指标强调的是**服务可用，但不保证数据正确。**
- 分区容错性（Partition Tolerance）
  - 分区容错性说的是，当节点间出现任意数量的消息丢失或高延迟的时候，系统仍然在继续工作。
  - 强调的是**集群对分区故障的容错能力。**

在分布式系统中分区容错性是必须要考虑的。

> 什么是网络分区：
>
> 网络分区是指因为网络故障导致网络不连通，不同节点分布在不同的子网络中，各个子网络内网络正常。其实，你可以这么理解，节点之间的网络通讯出现了消息丢失、高延迟的问题。

2. CAP不可能三角

CAP 不可能三角说的是对于一个分布式系统而言，一致性（Consistency）、可用性（Availability）、分区容错性（Partition Tolerance）3 个指标不可兼得，只能在 3 个指标中选择 2 个。



> CAP 不能三角最初是埃里克·布鲁尔（Eric Brewer）基于自己的工程实践，提出的一个猜想，后被赛斯·吉尔伯特（Seth Gilbert）和南希·林奇（Nancy Lynch）证明.
>
> 基于证明严谨性的考虑，赛斯·吉尔伯特（Seth Gilbert）和南希·林奇（Nancy Lynch）对指标的含义做了预设和限制，比如，将一致性限制为原子一致性。

3. CAP的使用

分区容错性必须考虑，分区容错性(P)是前提，必须保证。

（1）CP模型

当选择了一致性（C）的时候，一定会读到最新的数据，不会读到旧数据，但如果因为消息丢失、延迟过高发生了网络分区，那么这个时候，当集群节点接收到来自客户端的读请求时，为了不破坏一致性，可能会因为无法响应最新数据，而返回出错信息。

（2）AP模型

当选择了可用性（A）的时候，系统将始终处理客户端的查询，返回特定信息，如果发生了网络分区，一些节点将无法返回最新的特定信息，它们将返回自己当前的相对新的信息。

> 其实，在不存在网络分区的情况下，也就是分布式系统正常运行时（这也是系统在绝大部分时候所处的状态），就是说在不需要 P 时，C 和 A 能够同时保证。只有当发生分区故障的时候，也就是说需要 P 时，才会在 C 和 A 之间做出选择。而且如果读操作会读到旧数据，影响到了系统运行或业务运行（也就是说会有负面的影响），推荐选择 C，否则选 A。

4. 总结

- CA模型：舍弃P，即是舍弃了分布式系统；比如单机版关系型数据库 MySQL，如果 MySQL 要考虑主备或集群部署时，它必须考虑 P。

- CP模型：舍弃了可用性，一定会读到最新数据，不会读到旧数据。一旦因为消息丢失、延迟过高发生了网络分区，就影响用户的体验和业务的可用性。典型的应用是 Etcd，Consul 和 Hbase。
- AP模型：舍弃了一致性，实现了服务的高可用。用户访问系统的时候，都能得到响应数据，不会出现响应错误，但会读到旧数据。典型应用就比如 Cassandra 和 DynamoDB。

> CP 模型的 KV 存储和 AP 模型的 KV 存储，分别适合怎样的业务场景呢?
>
> 1. 适合用于提供基础服务，保存少量数据，作用类似zookeeper。
> 2. 适合查询量大的场景，不要求数据的强一致性，目前广泛应用于分布式缓存系统。

三、ACID

1. ACID

- 原子性（Atomicity）
- 隔离性（Isolation）
- 一致性（Consistency）
- 持久性（Durability）

ACID 理论是对事务特性的抽象和总结，方便我们实现事务。你可以理解成：如果实现了操作的 ACID 特性，那么就实现了事务。

> ACID与分布式一致性的区别?
>
> ？？？

2. 分布式事务协议

（1）二阶段提交协议

协调者和参与者

1. 协调者发起二阶段提交，进入提交请求阶段（投票阶段）
2. 提交请求阶段，会有资源预留和锁定
3. 协调者收到回复后，进入提交执行阶段（完成阶段）

> 需要注意的是，在第一个阶段，每个参与者投票表决事务是放弃还是提交。一旦参与者投票要求提交事务，那么就不允许放弃事务。也就是说，在一个参与者投票要求提交事务之前，它必须保证能够执行提交协议中它自己那一部分，即使参与者出现故障或者中途被替换掉。 **这个特性，是我们需要在代码实现时保障的。**
>
> 如果说第一阶段确认了，然后崩溃了，没有执行怎么办?
>
> 需要将提交相关信息保存到持久存储上，新进程启动后，恢复到之前的状态。

二阶段提交协议最早是用来实现数据库的分布式事务的，不过现在最常用的协议是 XA 协议。这个协议是 X/Open 国际联盟基于二阶段提交协议提出的，也叫作 X/Open Distributed Transaction Processing（DTP）模型，比如 MySQL 就是通过 MySQL XA 实现了分布式事务。

XA 协议、TCC、Paxos、Raft

存在的问题：

- 在提交请求阶段，需要预留资源，在资源预留期间，其他人不能操作（比如，XA 在第一阶段会将相关资源锁定）；

（2）TCC（Try-Confirm-Cancel）

TCC 是 Try（预留）、Confirm（确认）、Cancel（撤销） 3 个操作的简称，它包含了预留、确认或撤销这 2 个阶段。

1. 预留阶段，协调者通知操作和预留资源
2. 确认阶段，都预留没问题，进入确认阶段。
3. 如果预留阶段执行错误，进入撤销阶段。

> TCC 本质上是补偿事务，它的核心思想是针对每个操作都要注册一个与其对应的确认操作和补偿操作（也就是撤销操作）。 它是一个业务层面的协议，你也可以将 TCC 理解为编程模型，TCC 的 3 个操作是需要在业务代码中编码实现的，为了实现一致性，确认操作和补偿操作必须是等幂的，因为这 2 个操作可能会失败重试。
>
> 关于幂等性：
>
> 是指同一操作对同一系统的任意多次执行，所产生的影响均与一次执行的影响相同，不会因为多次执行而产生副作用。常见的实现方法有 Token、索引等。它的本质是通过唯一标识，标记同一操作的方式，来消除多次执行的副作用。

> TCC 不依赖于数据库的事务，而是在业务中实现了分布式事务，这样能减轻数据库的压力，但对业务代码的入侵性也更强，实现的复杂度也更高。所以，我推荐在需要分布式事务能力时，优先考虑现成的事务型数据库（比如 MySQL XA），当现有的事务型数据库不能满足业务的需求时，再考虑基于 TCC 实现分布式事务。

3. 总结

ACID 特性理解为 CAP 中一致性的边界，最强的一致性。建议在开发实现分布式系统，如果不是必须，尽量不要实现事务，可以考虑采用最终一致性。

2pc: 两阶段提交
优点：尽量保证了数据的强一致性
缺点：1、单点故障问题，协调者挂了，尤其在第二阶段的时候参与者资源都锁着时候影响更大。
　　2、同步阻塞，所有节点在执行时是同步阻塞的
　　3、数据不一致问题，例如网络分区，部分节点接受不到提交的请求。或是当协调者在第二阶段发送提交命令后挂了，此时有个节点接受到命令执行后也挂了。其他节点都没接受过命令。如果通过选择得出新的协调者，上线后该提交还是回滚都有可能和之前挂了的节点不一致。
3pc：三阶段提交
　优点：1、参与者超时机制，不会再傻等。
　　2、增加询问阶段不会直接锁资源
　　3、解决协调者和个别参与者一起挂了之后，选择上线不知该提交还是回滚的问题
　缺点：1、多了一步、耗时更多
　　2、网络分区情况下还是有数据不一致问题
TCC：基于两阶段的业务层面的分布式事务。
　优点：1、基于业务，不需要占用阻塞数据库宝贵资源
　　2、基于业务也更加灵活
　缺点：1、业务侵入性大
　　2、实现比较麻烦
　　3、需要保证cc的幂等

四、BASE理论

BASE 理论是 CAP 理论中的 AP 的延伸，是对互联网大规模分布式系统的实践总结，强调可用性。
NoSQL中应用广泛

核心

- 基本可用（Basically Available）

- 最终一致性（Eventually consistent）

软状态

软状态描述的是实现服务可用性的时候系统数据的一种过渡状态，也就是说不同节点间，数据副本存在短暂的不一致。

1. 实现基本可用的4板斧

- 流量削峰
- 延迟响应
- 体验降级
- 过载保护

其他：异步处理，故障隔离，弹性扩容

2. 最终一致性

最终一致性是说，系统中所有的数据副本在经过一段时间的同步后，最终能够达到一个一致的状态。也就是说，在数据一致性上，存在一个短暂的延迟。

> 几乎所有的互联网系统采用的都是最终一致性，只有在实在无法使用最终一致性，才使用强一致性或事务，比如，**对于决定系统运行的敏感元数据，需要考虑采用强一致性，对于与钱有关的支付系统或金融系统的数据，需要考虑采用事务。**

如何实现最终一致性？以什么为准？

两种方式

- 以最新写入的数据为准，**比如 AP 模型的 KV 存储采用的就是这种方式**；
- 以第一次写入的数据为准，**如果你不希望存储的数据被更改，可以以它为准。**

实现最终一致性的方式

- 读时修复：在读取数据时，检测数据的不一致，进行修复。比如 Cassandra 的 Read Repair 实现，具体来说，在向 Cassandra 系统查询数据的时候，如果检测到不同节点的副本数据不一致，系统就自动修复数据。
- 写时修复：在写入数据，检测数据的不一致时，进行修复。比如 Cassandra 的 Hinted Handoff 实现。具体来说，Cassandra 集群的节点之间远程写数据的时候，如果写失败就将数据缓存下来，然后定时重传，修复数据的不一致性。**写失败就会发现数据的不一致性，进行重传处理，即不需要进行一致性对比**
- 异步修复：这个是最常用的方式，通过定时对账检测副本数据的一致性，并修复

> - 因为写时修复不需要做数据一致性对比，性能消耗比较低，对系统运行影响也不大，所以我推荐你在实现最终一致性时优先实现这种方式。
> - 读时修复和异步修复因为需要做数据的一致性对比，性能消耗比较多，在开发实际系统时，你要尽量优化一致性对比的算法，降低性能消耗，避免对系统运行造成影响。

补充

在实现最终一致性的时候，推荐同时实现自定义写一致性级别（比如 All、Quorum、One、Any），让用户可以自主选择相应的一致性级别，比如可以通过设置一致性级别为 All，来实现强一致性

五、Paxos

分布式公式算法，基于Paxos改进的共识算法：

- Fast Paxos算法
- Cheap Paxos算法

- Raft算法

兰伯特提出

- 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；
- 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。

1. 三种角色

（1）提议者Proposer

提议一个值，用于投票表决。

（2）接受者（Acceptor）

对每个提议的值进行投票，并存储接受的值。一般来说，集群中的所有节点都在扮演接受者的角色，参与共识协商，并接受和存储数据。

（3）学习者（Learner）

被告知投票的结果，接受达成共识的值，存储保存，不参与投票的过程。一般来说，学习者是数据备份节点，比如“Master-Slave”模型中的 Slave，被动地接受数据，容灾备份。

2. 工作原理

二阶段提交

（1）准备（Prepare）阶段

1. 提议者，分别向所有接受者发送包含提案编号的准备请求。**在准备请求中是不需要指定提议的值的，只需要携带提案编号就可以了。**
2. 接受者会：
    - 之前没有通过任何提案，返回一个 “尚无提案”的响应。
    - 当提案编号大于它们之前响应的准备请求的提案编号时，返回一个 “尚无提案”的响应，并承诺以后不再响应提案编号小于等于此提案号的准备请求。
    - 提案号小于已经承诺的可接受的最小的提案编号的时候，丢弃准备请求，不做响应。

（2）接受（Accept）阶段

1. 提议者，发送接受请求。关于接受请求的值：
    - 当提议者收到大多数接受者的准备响应后，根据响应中提案编号最大的提案的值，设置接受请求中的值。
    - 如果都是空，则由提议者自己决定值。
2. 接受者会：
    - 当提案编号小于已经承诺的最小的提案编号的时候，拒绝。
    - 否则，通过。

如果集群中有学习者，当接受者通过了一个提案时，就通知给所有的学习者。当学习者发现大多数的接受者都通过了某个提案，那么它也通过该提案，接受该提案的值。

> Basic Paxos 的容错能力，源自“大多数”的约定，你可以这么理解：当少于一半的节点出现故障的时候，共识协商仍然在正常工作。
>
> 本质上而言，提案编号的大小代表着优先级，你可以这么理解，根据提案编号的大小，接受者保证三个承诺，具体来说：

> - 如果准备请求的提案编号，小于等于接受者已经响应的准备请求的提案编号，那么接受者将承诺不响应这个准备请求；
> - 如果接受请求中的提案的提案编号，小于接受者已经响应的准备请求的提案编号，那么接受者将承诺不通过这个提案；
> - 如果接受者之前有通过提案，那么接受者将承诺，会在准备请求的响应中，包含已经通过的最大编号的提案信息。

注意一点是，最终的目的是达到共识，也就是数据一致

两个阶段，先提议，然后看一下整体的情况，然后再accept（必须收到至少一半的回复才可以，不然的话就重新提议），然后进行数据的一致性同步，选择的值呢，是所有收到的响应中编号最大的提案的value，如果响应中不包含任何提案（即都是null），那么这个值就由proposer自己决定。

另外，关于多proposer的问题：

当两个proposer一次提出一系列编号递增的议案，那么会陷入死循环，无法完成第二阶段，也就是无法选定一个提案——可以通过选取主porposer来解决，只有主proposer可以进行提案。

> 面试精简答案：
> Paxos算法解决的是一个分布式系统如何就某个值（决议）达成一致。一个典型的场景是，在一个分布式数据库系统中，如果各个节点的初始状态一致，每个节点执行相同的操作序列，那么他们最后能够得到一个一致的状态。为了保证每个节点执行相同的命令序列，需要在每一条指令上执行一个“一致性算法”以保证每个节点看到的指令一致。zookeeper使用的zab算法是该算法的一个实现。在Paxos算法中，有三种角色：Proposer (提议者),Acceptor（接受者），Learners（记录员）
>
> - Proposer提议者：只要Proposer发的提案Propose被半数以上的Acceptor接受，Proposer就认为该提案例的value被选定了。
> - Acceptor接受者：只要Acceptor接受了某个提案，Acceptor就认为该提案例的value被选定了
> - Learner记录员：Acceptor告诉Learner哪个value就是提议者的提案被选定，Learner就认为哪个value被选定。
>
> Paxos算法分为两个阶段，具体如下：
> 阶段一 （准leader 确定 ）：
> (a) Proposer 选择一个提案编号 N，然后向半数以上的Acceptor 发送编号为 N 的 Prepare 请求。
> (b) 如果一个 Acceptor 收到一个编号为 N 的 Prepare 请求，且 N 大于该 Acceptor 已经响应过的所有 Prepare 请求的编号，那么它就会将它已经接受过的编号最大的提案（如果有的话）作为响应反馈给 Proposer，同时该Acceptor 承诺不再接受任何编号小于 N 的提案。
> 阶段二 （leader 确认）：
> (a) 如果 Proposer 收到半数以上 Acceptor 对其发出的编号为 N 的 Prepare 请求的响应，那么它就会发送一个针对[N,V]提案的 Accept 请求给半数以上的 Acceptor。注意：V 就是收到的响应中编号最大的提案的 value ，如果响应中不包含任何提案，那么V 就由 Proposer 自己决定。
> (b) 如果 Acceptor 收到一个针对编号为 N 的提案的 Accept 请求，只要该 Acceptor 没有对编号大于 N 的 Prepare 请求做出过响应，它就接受该提案。

3. Multi-paxos

兰伯特提到的 Multi-Paxos 是一种思想，不是算法。而 Multi-Paxos 算法是一个统称，它是指基于 Multi-Paxos 思想，通过多个 Basic Paxos 实例实现一系列值的共识的算法（比如 Chubby 的 Multi-Paxos 实现、Raft 算法等）。

（1）Basic Paxos痛点

- 如果多个提议者同时提交提案，可能出现因为提案编号冲突，在准备阶段没有提议者接收到大多数准备响应，协商失败，需要重新协商。
- 2 轮 RPC 通讯（准备阶段和接受阶段）往返消息多、耗性能、延迟大。

（2）解决方案

引入领导者和优化Basic Paxos执行来解决。

> 领导者节点作为唯一提议者，这样就不存在多个提议者同时提交提案的情况，也就不存在提案冲突的情况了

在论文中，兰伯特没有说如何选举领导者，需要我们在实现 Multi-Paxos 算法的时候自己实现。 比如在 Chubby 中，主节点（也就是领导者节点）是通过执行 Basic Paxos 算法，进行投票选举产生的。

> 可以采用“当领导者处于稳定状态时，省掉准备阶段，直接进入接受阶段”这个优化机制，优化 Basic Paxos 执行。也就是说，领导者节点上，序列中的命令是最新的，不再需要通过准备请求来发现之前被大多数节点通过的提案，领导者可以独立指定提案中的值。这时，领导者在提交命令时，可以省掉准备阶段，直接进入到接受阶段。
>
> 这个优化机制，省掉 Basic Paxos 的准备阶段，提升了数据的提交效率，但是所有写请求都在主节点处理，限制了集群处理写请求的并发能力，约等于单机。
>
> 本质上而言，“当领导者处于稳定状态时，省掉准备阶段，直接进入接受阶段”这个优化机制，是通过减少非必须的协商步骤来提升性能的。这种方法非常常用，也很有效。比如，Google 设计的 QUIC 协议，是通过减少 TCP、TLS 的协商步骤，优化 HTTPS 性能。

关于什么是稳定阶段，然后采用优化机制的问题：

> 如何理解领导者处于稳定状态?
> 领导者节点上，序列中的命令是最新的，不再需要通过准备请求来发现之前被大多数节点通过的提案，领导者可以独立指定提案中的值。
> 我来具体说说，**准备阶段的意义，是发现接受者节点上，已经通过的提案的值。如果在所有接受者节点上，都没有已经通过的提案了，这时，领导者就可以自己指定提案的值了，那么，准备阶段就没有意义了，也就是可以省掉了。**

（3）Chubby的Multi-Paxos实现

- 通过引入主节点，实现了兰伯特提到的领导者（Leader）节点的特性。也就是说，主节点作为唯一提议者，这样就不存在多个提议者同时提交提案的情况，也就不存在提案冲突的情况了。
- 在 Chubby 中，主节点是通过执行 Basic Paxos 算法，进行投票选举产生的，并且在运行过程中，主节点会通过不断续租的方式来延长租期（Lease）。比如在实际场景中，几天内都是同一个节点作为主节点。如果主节点故障了，那么其他的节点又会投票选举出新的主节点，也就是说主节点是一直存在的，而且是唯一的。
- 在 Chubby 中实现了兰伯特提到的，“当领导者处于稳定状态时，省掉准备阶段，直接进入接受阶段”这个优化机制。
- 在 Chubby 中，实现了成员变更（Group membership），以此保证节点变更的时候集群的平稳运行。
- 在 Chubby 中，为了实现了强一致性，读操作也只能在主节点上执行。 也就是说，只要数据写入成功，之后所有的客户端读到的数据都是一致的。

> 因为在 Chubby 的 Multi-Paxos 实现中，也约定了“大多数原则”，也就是说，只要大多数节点正常运行时，集群就能正常工作，所以 Chubby 能容错（n - 1）/2 个节点的故障。
>
> 在实际使用时，我不推荐你设计和实现新的 Multi-Paxos 算法，而是建议优先考虑 Raft 算法，因为 Raft 的正确性是经过证明的。当 Raft 算法不能满足需求时，你再考虑实现和优化 Multi-Paxos 算法。

关于Chubby这样子相当于单机，为什么还这样子做呢?

> 有些场景需要的是强一致性、故障容错等，场景决定系统的形态。

六、Raft

Raft 算法是现在分布式系统开发首选的共识算法。绝大多数选用 Paxos 算法的系统（比如 Cubby、Spanner）都是在 Raft 算法发布前开发的，当时没得选；而全新的系统大多选择了 Raft 算法（比如 Etcd、Consul、CockroachDB）

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。

1. 成员身份

（1）领导者（Leader）

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
- 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。

（2）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。

（3）跟随者（Follower）

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。

Raft 算法是强领导者模型，集群中只能有一个“霸道总裁”

2. 选举过程

- 初始状态下，集群中所有的节点都是跟随者的状态。
- Raft 算法实现了随机超时时间的特性。也就是说，每个节点等待领导者节点心跳信息的超时时间间隔是随机的。
- 最先因为没有等到领导者的心跳信息，发生超时。这个时候，节点 A 就增加自己的任期编号，并推举自己为候选人，先给自己投上一张选票，然后向其他节点发送请求投票 RPC 消息，请它们选举自己为领导者。
- 如果其他节点接收到候选人 A 的请求投票 RPC 消息，在编号为 1 的这届任期内，也还没有进行过投票，那么它将把选票投给节点 A，并增加自己的任期编号。
- 如果候选人在选举超时时间内赢得了大多数的选票，那么它就会成为本届任期内新的领导者。
- 节点 A 当选领导者后，他将周期性地发送心跳消息，通知其他服务器我是领导者，阻止跟随者发起新的选举，篡权。

（1）节点间如何通讯？

在 Raft 算法中，服务器节点间的沟通联络采用的是远程过程调用（RPC），在领导者选举中，需要用到这样两类的 RPC：

- 请求投票（RequestVote）RPC，是由候选人在选举期间发起，通知各节点进行投票；
- 日志复制（AppendEntries）RPC，是由领导者发起，用来复制日志和提供心跳消息。

日志复制 RPC 只能由领导者发起，这是实现强领导者模型的关键之一。

（2）什么是任期？

与现实议会选举中的领导者的任期不同，Raft 算法中的任期不只是时间段，而且**任期编号的大小**，会影响领导者选举和请求的处理。

- **在 Raft 算法中约定，如果一个候选人或者领导者，发现自己的任期编号比其他节点小，那么它会立即恢复成跟随者状态。**比如分区错误恢复后，任期编号为 3 的领导者节点 B，收到来自新领导者的，包含任期编号为 4 的心跳消息，那么节点 B 将立即恢复成跟随者状态。
- **约定如果一个节点接收到一个包含较小的任期编号值的请求，那么它会直接拒绝这个请求。**比如节点 C 的任期编号为 4，收到包含任期编号为 3 的请求投票 RPC 消息，那么它将拒绝这个消息。

（3）选举有哪些规则？

1. 领导者周期性地向所有跟随者发送心跳消息（即不包含日志项的日志复制 RPC 消息），通知大家我是领导者，阻止跟随者发起新的选举。
2. 如果在指定时间内，跟随者没有接收到来自领导者的消息，那么它就认为当前没有领导者，推举自己为候选人，发起领导者选举。
3. 在一次选举中，赢得大多数选票的候选人，将晋升为领导者。
4. 在一个任期内，领导者一直都会是领导者，直到它自身出现问题（比如宕机），或者因为网络延迟，其他节点发起一轮新的选举。
5. 在一次选举中，每一个服务器节点最多会对一个任期编号投出一张选票，并且按照“先来先服务”的原则进行投票。
6. 日志完整性高的跟随者（也就是最后一条日志项对应的任期编号值更大，索引号更大），拒绝投票给日志完整性低的候选人。

选举是跟随者发起的，推举自己为候选人；大多数选票是指集群成员半数以上的选票；大多数选票规则的目标，是为了保证在一个给定的任期内最多只有一个领导者。

（4）如何理解随机超时时间？

Raft 算法巧妙地使用随机选举超时时间的方法，把超时时间都分散开来，在大多数情况下只有一个服务器节点先发起选举，而不是同时发起选举，这样就能**减少因选票瓜分导致选举失败**的情况。

在 Raft 算法中，随机超时时间是有 2 种含义的，这里是很多同学容易理解出错的地方，需要你注意一下：

- 跟随者等待领导者心跳信息超时的时间间隔，是随机的；
- 如果候选人在一个随机时间间隔内，没有赢得过半票数，那么选举无效了，然后候选人发起新一轮的选举，也就是说，等待选举超时的时间间隔，是随机的。

> Raft 算法和兰伯特的 Multi-Paxos 不同之处，主要有2点：
>
> - 首先，在 Raft 中，不是所有节点都能当选领导者，只有日志较完整的节点（也就是日志完整度不比半数节点低的节点），才能当选领导者；
> - 其次，在 Raft 中，日志必须是连续的。
>
> Raft 算法通过任期、领导者心跳消息、随机选举超时时间、先来先服务的投票原则、大多数选票原则等，保证了一个任期只有一位领导，也极大地减少了选举失败的情况。
>
> 本质上，Raft 算法以领导者为中心，选举出的领导者，以“一切以我为准”的方式，达成值的共识，和实现各节点日志的一致。
>
> 关于“一切以我为准”的强领导者模型“的限制和局限性：
>
> 1. 读写请求和数据转发压力落在领导者节点，导致领导者压力。
> 2. 大规模跟随者的集群，领导者需要承担大量元数据维护和心跳通知的成本。
> 3. 领导者单点问题，故障后直到新领导者选举出来期间集群不可用。
> 4. 随着候选人规模增长，收集半数以上投票的成本更大。

3. 日志复制

在 Raft 算法中，副本数据是以日志的形式存在的，领导者接收到来自客户端写请求后，处理写请求的过程就是一个复制和应用（Apply）日志项到状态机的过程。

（1）日志

副本数据是以日志的形式存在的，日志是由日志项组成。日志项是一种数据格式，它主要包含用户指定的数据，也就是指令（Command），还包含一些附加信息，比如索引值（Log index）、任期编号（Term）。

- 指令：一条由客户端请求指定的、状态机需要执行的指令。你可以将指令理解成客户端指定的数据。
- 索引值：日志项对应的整数索引值。它其实就是用来标识日志项的，是一个连续的、单调递增的整数号码。
- 任期编号：创建这条日志项的领导者的任期编号。

（2）复制过程

一个优化后的二阶段提交（将二阶段优化成了一阶段），减少了一半的往返消息，也就是降低了一半的消息延迟。

- 接收到客户端请求后，领导者基于客户端请求中的指令，创建一个新日志项，并附加到本地日志中。
- 领导者通过日志复制 RPC，将新的日志项复制到其他的服务器。
- 当领导者将日志项，成功复制到大多数的服务器上的时候，领导者会将这条日志项应用到它的状态机中。
- 领导者将执行的结果返回给客户端。
- 当跟随者接收到心跳信息，或者新的日志复制 RPC 消息后，如果跟随者发现领导者已经提交了某条日志项，而它还没应用，那么跟随者就将这条日志项应用到本地的状态机中。

（3）实现日志的一致

在实际环境中，复制日志的时候，你可能会遇到进程崩溃、服务器宕机等问题，这些问题会导致日志不一致。

过程：

- 领导者通过日志复制 RPC 的一致性检查，找到跟随者节点上，与自己相同日志项的最大索引值。也就是说，这个索引值之前的日志，领导者和跟随者是一致的，之后的日志是不一致的了。
- 领导者强制跟随者更新覆盖的不一致日志项，实现日志的一致。

> 兰伯特的 Multi-Paxos 不要求日志是连续的，但在 Raft 中日志必须是连续的。而且在 Raft 中，日志不仅是数据的载体，日志的完整性还影响领导者选举的结果。也就是说，日志完整性最高的节点才能当选领导者。

4. 成员变更

最初实现成员变更的是联合共识（Joint Consensus），但这个方法实现起来难，后来 Raft 的作者就提出了一种改进后的方法，**单节点变更（single-server changes）**。

配置：它就是在说集群是哪些节点组成的，是集群各节点地址信息的集合。比如节点 A、B、C 组成的集群，那么集群的配置就是[A, B, C]集合。

成员变更最大的风险：可能会同时出现 2 个领导者。

目前的集群配置为[A, B, C]，我们先向集群中加入节点 D，这意味着新配置为[A, B, C, D]。成员变更，是通过这么两步实现的：

- 第一步，领导者（节点 A）向新节点（节点 D）同步数据；
- 第二步，领导者（节点 A）将新配置[A, B, C, D]作为一个日志项，复制到新配置中所有节点（节点 A、B、C、D）上，然后将新配置的日志项应用（Apply）到本地状态机，完成单节点变更。

这样一来，我们就通过一次变更一个节点的方式，完成了成员变更，保证了集群中始终只有一个领导者，而且集群也在稳定运行，持续提供服务。

> 在正常情况下，不管旧的集群配置是怎么组成的，旧配置的“大多数”和新配置的“大多数”都会有一个节点是重叠的。 也就是说，不会同时存在旧配置和新配置 2 个“大多数”。

注意：在分区错误、节点故障等情况下，如果我们并发执行单节点变更，那么就可能出现一次单节点变更尚未完成，新的单节点变更又在执行，导致集群出现 2 个领导者的情况。

> 如果你遇到这种情况，可以在领导者启动时，创建一个 NO_OP 日志项（也就是空日志项），只有当领导者将 NO_OP 日志项应用后，再执行成员变更请求。
>
> 具体的实现，可参考 Hashicorp Raft 的源码，也就是 runLeader() 函数中：
>
> ```
> noop := &logFuture{
>     log: Log{
>     Type: LogNoop,
> },
> }r.dispatchLogs([]*logFuture{noop})
> ```

- 成员变更的问题，主要在于进行成员变更时，可能存在新旧配置的 2 个“大多数”，导致集群中同时出现两个领导者，破坏了 Raft 的领导者的唯一性原则，影响了集群的稳定运行。
- 单节点变更是利用“一次变更一个节点，不会同时存在旧配置和新配置 2 个‘大多数’”的特性，实现成员变更。
- 因为联合共识实现起来复杂，不好实现，所以绝大多数 Raft 算法的实现，采用的都是单节点变更的方法（比如 Etcd、Hashicorp Raft）。其中，Hashicorp Raft 单节点变更的实现，是由 Raft 算法的作者迭戈·安加罗（Diego Ongaro）设计的，很有参考价值。

5. 补充

- **Raft 不是一致性算法而是共识算法，是一个 Multi-Paxos 算法，实现的是如何就一系列值达成共识**
- **Raft 能容忍少数节点的故障。**
- **虽然 Raft 算法能实现强一致性，也就是线性一致性（Linearizability），但需要客户端协议的配合。**
- **在实际场景中，我们一般需要根据场景特点，在一致性强度和实现复杂度之间进行权衡。**

Consul实现的三种一致性模型：

- default：客户端访问领导者节点执行读操作，领导者确认自己处于稳定状态时（在 leader leasing 时间内），返回本地数据给客户端，否则返回错误给客户端。在这种情况下，客户端是可能读到旧数据的，比如此时发生了网络分区错误，新领导者已经更新过数据，但因为网络故障，旧领导者未更新数据也未退位，仍处于稳定状态。
- consistent：客户端访问领导者节点执行读操作，领导者在和大多数节点确认自己仍是领导者之后返回本地数据给客户端，否则返回错误给客户端。在这种情况下，客户端读到的都是最新数据。
- stale：从任意节点读数据，不局限于领导者节点，客户端可能会读到旧数据。

> 有什么办法能突破 Raft 集群的写性能瓶颈呢?
>
> - 可以参考Kafka的分区和ES的主分片副本分片这种机制，虽然写入只能通过leader写，但每个leader可以负责不同的片区，来提高写入的性能
> - leader可以合并请求
> - leader提交日志和日志复制RPC两个步骤可以并行和批量处理

> - leader可以并发和异步对所有follower 并发日志复制，同时可以利用pipeline的方式提高效率
> - 每个节点启动多个raft实例，对请求进行hash或者range后，让每个raft实例负责部分请求

七、一致性哈希

问题：分集群，突破单集群的性能限制。

方法：使用哈希算法，加个 Proxy 层，由 Proxy 层处理来自客户端的读写请求，接收到读写请求后，通过对 Key 做哈希找到对应的集群。

缺点：当需要变更集群数时（比如从 2 个集群扩展为 3 个集群），这时大部分的数据都需要迁移，重新映射，数据的迁移成本是非常高的。

解决：一致性哈希

1. 工作原理

一致哈希算法也用了取模运算，但与哈希算法不同的是，哈希算法是对节点的数量进行取模运算，而一致哈希算法是对 2^32 进行取模运算。一致哈希算法，将整个哈希值空间组织成一个虚拟的圆环，也就是哈希环。

当需要对指定 key 的值进行读写的时候，你可以通过下面 2 步进行寻址：

- 首先，将 key 作为参数执行 c-hash() 计算哈希值，并确定此 key 在环上的位置；
- 然后，从这个位置沿着哈希环顺时针“行走”，遇到的第一节点就是 key 对应的节点。

总的来说，使用了一致哈希算法后，扩容或缩容的时候，都只需要重定位环空间中的一小部分数据。也就是说，一致哈希算法具有较好的容错性和可扩展性。

2. 冷热不均问题

哈希寻址中常出现这样的问题：客户端访问请求集中在少数的节点上，出现了有些机器高负载，有些机器低负载的情况，那么在一致哈希中，有什么办法能让数据访问分布的比较均匀呢？答案就是**虚拟节点。**

在一致哈希中，如果节点太少，容易因为节点分布不均匀造成数据访问的冷热不均，也就是说大多数访问请求都会集中少量几个节点上。

> 对每一个服务器节点计算多个哈希值，在每个计算结果位置上，都放置一个虚拟节点，并将虚拟节点映射到实际节点。**使每个节点在哈希环上的分布相对均匀。**

一致哈希本质上是一种路由寻址算法，适合简单的路由寻址场景。比如在 KV 存储系统内部，它的特点是简单，不需要维护路由信息。

> 问题：Raft 集群具有容错能力，能容忍少数的节点故障，那么在多个 Raft 集群组成的 KV 系统中，如何设计一致哈希，实现当某个集群的领导者节点出现故障，并选举出新的领导者后，整个系统还能稳定运行呢？
>
> **答案：可以做两层“寻址”，先通过一致性哈希寻址到集群，然后再找到“领导者”。**

八、Gossip协议

问题：根据 Base 理论，你需要实现最终一致性，怎么样才能实现最终一致性呢？

方法：可以通过 Gossip 协议实现这个目标。

Gossip 协议，顾名思义，就像流言蜚语一样，利用一种随机、带有传染性的方式，将信息传播到整个网络中，并在一定时间内，使得系统内的所有节点数据一致。

1. Gossip的三板斧

- 直接邮寄（Direct Mail）
- 反熵（Anti-entropy）
- 谣言传播（Rumor mongering）

（1）直接邮寄（Direct Mail）

直接发送更新数据，当数据发送失败时，将数据缓存下来，然后重传。

直接邮寄虽然实现起来比较容易，数据同步也很及时，但可能会因为缓存队列满了而丢数据。也就是说，只采用直接邮寄是无法实现最终一致性的。

（2）反熵（Anti-entropy）

那如何实现最终一致性呢？答案就是反熵。

本质上，反熵是一种通过异步修复实现最终一致性的方法。反熵指的是集群中的节点，每隔段时间就随机选择某个其他节点，然后通过互相交换自己的所有数据来消除两者之间的差异，实现数据的最终一致性：

常见的最终一致性系统（比如 Cassandra），都实现了反熵功能。

其实，在实现反熵的时候，主要有推、拉和推拉三种方式。

- 推方式：将自己的所有副本数据，推给对方，修复对方副本中的熵
- 拉方式：拉取对方的所有副本数据，修复自己副本中的熵

> 反熵中的熵是指混乱程度，反熵就是指消除不同节点中数据的差异，提升节点间数据的相似度，降低熵值。

注意

> 虽然反熵很实用，但是执行反熵时，相关的节点都是已知的，而且节点数量不能太多，如果是一个动态变化或节点数比较多的分布式环境（比如在 DevOps 环境中检测节点故障，并动态维护集群节点状态），这时反熵就不适用了。

（3）谣言传播（Rumor mongering）

问题：当环境是一个动态变化或节点数比较多的分布式环境的时候，反熵不再适用，这个时候怎么实现最终一致性？

方法：答案就是谣言传播。

谣言传播，广泛地散播谣言，它指的是当一个节点有了新数据后，这个节点变成活跃状态，并周期性地联系其他节点向其发送新数据，直到所有的节点都存储了该新数据。

2. 实际使用

在分布式存储系统中，实现数据副本最终一致性，最常用的方法就是反熵了。

例子：自研 InfluxDB 的反熵实现为例

在自研 InfluxDB 中，一份数据副本是由多个分片组成的，也就是实现了数据分片，三节点三副本的集群。

反熵的目标是确保每个 DATA 节点拥有元信息指定的分片，而且不同节点上，同一分片组中的分片都没有差异。比如说，节点 A 要拥有分片 Shard1 和 Shard2，而且，节点 A 的 Shard1 和 Shard2，与节点 B、C 中的 Shard1 和 Shard2，是一样的。

数据缺失，分为这样 2 种情况：

- 缺失分片：也就是说，在某个节点上整个分片都丢失了。
- 节点之间的分片不一致：也就是说，节点上分片都存在，但里面的数据不一样，有数据丢失的情况发生。

第一种情况修复起来不复杂，我们只需要将分片数据，通过 RPC 通讯，从其他节点上拷贝过来就可以了。

第二种情况修复起来要复杂一些。我们需要设计一个闭环的流程，按照一个顺序修复，执行完流程后，也就是实现了一致性了。

> 它是按照一定顺序来修复节点的数据差异，**先随机选择一个节点，然后循环修复**，每个节点生成自己节点有、下一个节点没有的差异数据，发送给下一个节点，进行修复。
>
> 为什么这么设计：因为我们希望能在一个确定的时间范围内实现数据副本的最终一致性，而不是基于随机性的概率，在一个不确定的时间范围内实现数据副本的最终一致性。
>
> **因为反熵需要做一致性对比，很消耗系统性能，所以建议你将是否启用反熵功能、执行一致性检测的时间间隔等，做成可配置的，能在不同场景中按需使用。**

3. 总结

在实际场景中，实现数据副本的最终一致性时：

- 一般而言，直接邮寄的方式是一定要实现的，因为不需要做一致性对比，只是通过发送更新数据或缓存重传，来修复数据的不一致，性能损耗低。
- 在存储组件中，节点都是已知的，一般采用反熵修复数据副本的一致性。
- 当集群节点是变化的，或者集群节点数比较多时，这时要采用谣言传播的方式，同步更新数据，实现最终一致。

> 问题：既然使用反熵实现最终一致性时，需要通过一致性检测发现数据副本的差异，如果每次做一致性检测时都做数据对比的话，肯定是比较消耗性能的，那有什么办法降低一致性检测时的性能消耗呢？
>
> 答案：
>
> 1. 假设从某一时刻开始，所有节点数据都是一致的，每个节点都记录从这个时刻开始的数据变化，
> 2. 当反墒放生时，先看相关的两个节点上数据变化是否为空，如果为空就证明两个节点此时数据一致，不需要数据同步。
> 3. 如果相关的两个节点上数据有变化，则只比较变化的数据，然后只同步变化的差集，同时也要更新数据变化状态记录，
> 4. 在某个时间段内数据的变化只增不减，某个时刻所有节点做全量比对，然后重置数据变化记录。
> 5. 为了快速比较，可以同时计算数据变化记录的hash值用于比较，hash值一样就说明数据变化是相同的，两节点数据一致，否则就需要一致性查询并同步。

九、 Quorum NWR算法

通过 Quorum NWR，你可以**自定义一致性级别**，通过临时调整写入或者查询的方式，当 W + R > N 时，就可以实现强一致性了。

在 AP 型分布式系统中（比如 Dynamo、Cassandra、InfluxDB 企业版的 DATA 节点集群），Quorum NWR 是通常都会实现的一个功能

1. Quorum NWR算法的三要素

- N 表示副本数，又叫做复制因子（Replication Factor）。也就是说，N 表示集群中同一份数据有多少个副本。副本数可以不等于节点数，不同的数据可以有不同的副本数。
- W，又称写一致性级别（Write Consistency Level），表示成功完成 W 个副本更新，才完成写操作。
- R，又称读一致性级别（Read Consistency Level），表示读取一个数据对象时需要读 R 个副本。你可以这么理解，读取指定数据时，要读 R 副本，然后返回 R 个副本中最新的那份数据。

效果

- 当 W + R > N 的时候，对于客户端来讲，整个系统能保证强一致性，一定能返回更新后的那份数据。
- 当 W + R <= N 的时候，对于客户端来讲，整个系统只能保证最终一致性，可能会返回旧数据。

2. 如何实现Quorum NWR算法

InfluxDB 企业版，支持"any、one、quorum、all"4 种写一致性级别，具体的含义是这样的。

- any：任何一个节点写入成功后，或者接收节点已将数据写入 Hinted-handoff 缓存（也就是写其他节点失败后，本地节点上缓存写失败数据的队列）后，就会返回成功给客户端。
- one：任何一个节点写入成功后，立即返回成功给客户端，不包括成功写入到 Hinted-handoff 缓存。
- quorum：当大多数节点写入成功后，就会返回成功给客户端。此选项仅在副本数大于 2 时才有意义，否则等效于 all。
- all：仅在所有节点都写入成功后，返回成功。

3. 总结

- 一般而言，不推荐副本数超过当前的节点数，因为当副本数据超过节点数时，就会出现同一个节点存在多个副本的情况。当这个节点故障时，上面的多个副本就都受到影响了。
- 当 W + R > N 时，可以实现强一致性。另外，如何设置 N、W、R 值，取决于我们想优化哪方面的性能。比如，N 决定了副本的冗余备份能力；如果设置 W = N，读性能比较好；如果设置 R = N，写性能比较好；如果设置 W = (N + 1) / 2、R = (N + 1) / 2，容错能力比较好，能容忍少数节点（也就是 (N - 1) / 2）的故障。

> 问题：实现 Quorum NWR 时，需要实现自定义副本的能力，那么，一般设置几个副本就可以了，为什么呢?
>
> 答案：
>
> - 在需要提供高性能和高可用性的分布式存储系统中，副本的数量即 n 通常超过 2 个。
> - 只关注容错的系统通常使用 n=3 （W=2和R=2配置）。
> - 需要提供非常高读取负载的系统通常会复制超出容错要求的数据

十、PBFT

PBFT 算法非常实用，是一种能在实际场景中落地的拜占庭容错算法，它在区块链中应用广泛（比如 Hyperledger Sawtooth、Zilliqa）。

1. 口信消息型拜占庭问题之解的局限

- 如果将军数为 n、叛将数为 f，那么算法需要递归协商 f+1 轮，消息复杂度为 O(n ^ (f + 1))，消息数量指数级暴增。
- 尽管对于签名消息，不管叛将数（比如 f）是多少，经过 f + 1 轮的协商，忠将们都能达成一致的作战指令，但是这个算法同样存在"理论化"和"消息数指数级暴增"的痛点。

2. PBFT 是如何达成共识的?

核心过程有三个阶段，分别

- pre-prepare （预准备）阶段

  对于 pre-prepare 阶段，主节点广播 pre-prepare 消息给其它节点即可，因此通信次数为 n-1；

- prepare （准备）阶段

  对于 prepare 阶段，每个节点如果同意请求后，都需要向其它节点再 广播 parepare 消息，所以总的通信次数为 n*（n-1），即 n^2-n；

- commit （提交）阶段

  当某个将军收到 2f 个一致的包含作战指令的准备消息后，会进入提交（Commit）阶段（这里的 2f 包括自己，其中 f 为叛徒数）

  对于 commit 阶段，每个节点如果达到 prepared 状态后，都需要向其它节点广播 commit 消息，所以总的通信次数也为 n*（n-1），即 n^2-n 。

最后，当某个将军收到 2f + 1 个验证通过的提交消息后（包括自己，其中 f 为叛徒数），也就是说，大部分的将军们已经达成共识，这时可以执行作战指令了，那么该将军将执行苏秦的作战指令，执行完毕后发送执行成功的消息给苏秦。

最后，当苏秦收到 f+1 个相同的响应（Reply）消息时，说明各位将军们已经就作战指令达成了共识，并执行了作战指令（其中 f 为叛徒数）。**即保证了忠诚节点之间的一致性**

> 问题：客户端要收到 f+1 个结果，我理解这个是为了防止 f 个叛徒直接给客户端返回 ok。不太理解的是为什么准备阶段要收到 2f 个一致的包含作战指令的准备消息，提交阶段需要 2f+1 个验证通过呢？这两个也设置成 f+1，不可以吗？
>
> 答案：问题1：2f个准备消息和预准备消息是相同的，所以，加上主节点，就是2f + 1了，也就是说准备阶段的2f个，等同于提交阶段的2f + 1。问题2：如果设置为f + 1，那么就会被恶意节点干扰，比如，在准备和提交阶段，f个恶意节点，都很配合，但在最后，它们却不返回响应消息给客户端，这时客户端可能始终无法收到f + 1个一致的响应消息，也就是达成共识失败，然后客户端不断重试，肯定不行了。

所以总通信次数为 （n-1）+（n^2-n）+（n^2-n），即 2n^2-n-1，因此pbft算法复杂度为 O（n^2）。

- **PBFT 算法是通过签名（或消息认证码 MAC）约束恶意节点的行为，也就是说，每个节点都可以通过验证消息签名确认消息的发送来源，一个节点无法伪造另外一个节点的消息。最终，基于大多数原则（2f + 1）实现共识的。**
- **采用三阶段协议，基于大多数原则达成共识的。**
- **最终的共识是否达成，客户端是会做判断的，如果客户端在指定时间内未收到请求对应的 f + 1 相同响应，就认为集群出故障了，共识未达成，客户端会重新发送请求。**
- **PBFT 算法通过视图变更（View Change）的方式，来处理主节点作恶，当发现主节点在作恶时，会以“轮流上岗”方式，推举新的主节点。**
- **尽管 PBFT 算法相比口信消息型拜占庭之解已经有了很大的优化，将消息复杂度从 O(n ^ (f + 1)) 降低为 O(n ^ 2)，能在实际场景中落地，并解决实际的共识问题。但 PBFT 还是需要比较多的消息。**

在中小型分布式系统中使用 PBFT 算法。

3. 总结

- 相比 Raft 算法完全不适应有人作恶的场景，PBFT 算法能容忍 (n - 1)/3 个恶意节点 (也可以是故障节点)。
- 另外，相比 PoW 算法，PBFT 的优点是不消耗算力，所以在日常实践中，PBFT 比较适用于相对“可信”的场景中，比如联盟链。

- PBFT 算法与 Raft 算法类似，也存在一个“领导者”（就是主节点），同样，集群的性能也受限于“领导者”。另外，O(n ^ 2) 的消息复杂度，以及随着消息数的增加，网络时延对系统运行的影响也会越大，这些都限制了运行 PBFT 算法的分布式系统的规模，也决定了 PBFT 算法适用于中小型分布式系统。