到底在哪些方面造成了难以理解？

1. 没有贯穿始终的例子，完全凭想象在理解这个本来就难懂的抽象概念，没法落地和形成深刻印象。
2. Majority, chosen这些词，首先让大家想到是个选举，而没明白实际上是用来一致地决定状态变化的序列，即使部分成员故障也不影响服务。到文章后面隐含地说明这点时，很多人已经晕了。因此，明确要解决问题的边界，非常重要。
3. 作者为了简化，在讲解单实例及推导过程时(从P1->P2->两阶段协议)，完全没有状态序列的概念。这造成了后来突然抛出个多实例状态机时（a sequence of separate instances），跳跃性太大，无法体会作者的意图。

# 高可用问题的引入

## 鸡蛋不能都放在一个篮子里

假设你开了个蛋糕店，每天生产和销售大量的食品。你有个处理订单的系统，通过这个系统能够记录每一笔订单的详细情况并可以随时查询。用户可以通过手机App或者网页查询。可以想象，如果要让客户信赖，这个订单系统应该做到：

1. 客户下的订单，无论是要几块蛋糕或者冰激凌，一旦成功，系统不能反悔(客户主动取消订单是另外一回事)。
2. 如果用户下订单成功，那么订单不能丢失。
3. 如果不能满足用户需求，系统在客户下订单时就告诉用户订单失败。

假设你有台服务器，运行着订单处理系统，软件也没有bug，运行的很好，一切看起来那么正常。

但是有一天某个店小二跟你报告电脑硬盘坏了，虽然头天晚上做了备份，但是今天的订单全丢了。硬盘故障概率的一般不高，但是天热机器发生个风扇故障或其他小问题的概率要高的多。在这种情况下，虽然之前的数据没丢，订单处理也会有中断。

这里其实提到了两个可能出现的问题：丢订单(数据)，这个属于可靠性问题；如果订单数据没丢，只是一段时间用户不能下订单，这个属于可用性问题。

## 一山容不了二虎

你当然不希望一个风扇故障引起用户抱怨，更不希望丢失了订单数据，因为那些通过网络付了款的人用户可能堵了你的门。

你有个朋友，给你建议上个高可用系统，由两台服务器组成，例如M1和M2，只要其中一个能干活就可以，这样从可靠性和可用性两个方面，都有所大幅提高，毕竟两个机器硬盘同时故障的概率小很多。

看起来挺不错的方案，问题在哪呢？ 我们考虑M1和M2两个人怎么干活。你的客户通过App或者浏览器下订单时，可能发给M1，也可能发给M2，这种情况下，如果现在只剩下一个草莓蛋糕，张三和李四两个同学都通过App下单，但是分别是M1和M2处理的，那么可能两个人都认为自己订蛋糕成功了，但是去取的时候，有一个人发现没有了。显然，两个没有协作分工的服务器，不能正确地干活。

**一张图展示这些推理过程(一个M => M1和M2并列 = 互相不服气 => M1为主，M2为备机 =网络分割=>)**

## 杳无音讯的假死

你决定只有一个服务器能接收请求，比如M1是主服务器，M2从备用。张三李四都得找M1下订单。M1收到订单后，M1和M2做同样的操作，它们都处理完成才算成功。这下可靠性貌似提高了，比如M2宕机了，M1发现没法发订单给小M2，M1就独立处理了。可是如果M2好好的，M1宕机了呢？这个看起来也很简单: 直接让M2接订单并独立干活，不理M1不就完了吗？

问题在于：谁决定让M2独立干活，不理小M1？M2自己？还是店老板您呢？

显然不能是店老板，你没法一直在那看着，也没那工夫。

M2如何判断M1宕机了呢？常见的方法是定期给M1发送心跳，并等待M1的回复。但是在网络不稳定的情况下，**可能出现客户张三能跟M1通讯，李四能跟M2通讯，但是M1和M2之间的通讯中断。在每个订单被处理时，都认为另外一个服务器挂了，自己应该独立处理**。这个就是脑裂问题，关于这个问题，有个用的比较多的解释如下：

TODO: 脑裂问题示意图：

cc# What does "split-brain" mean?

"Split brain" is a condition whereby two or more computers or groups of computers lose contact with one another but still act as if the cluster were intact. This is like having two governments trying to rule the same country. If multiple computers are allowed to write to the same file system without knowledge of what the other nodes are doing, it will quickly lead to data corruption and other serious problems.

Split-brain is prevented by enforcing quorum rules (which say that no group of nodes may operate unless they are in contact with a majority of all nodes) and fencing (which makes sure nodes outside of the quorum are prevented from interfering with the cluster).

简言之，就是每个节点都认为对方挂了，自己应该多干活，结果就乱套了。如果上面的集群中机器个数不是2个，而是多个，问题相同。

看到这里，我们知道不是简单地把几个机器放在一起就变成了一个如你所想象的正常工作的集群。我们需要保证正确性、可靠性、可用性的机制。而目前已有的分布式一致性保障机制基本都是基于Paxos。

# Paxos的Yes和No

能保证所有订单都成功吗？ No，用户能否买到蛋糕，还依赖于你今天能做多少蛋糕。

能100%保证不丢数据吗？ No，不能，比如发生地震，但是只要过半数机器能工作就能找回数据。

能保证系统100%的可用性吗？ No，不能，比如大规模停电就不能。但是只要过半数机器能工作就可以正常服务。

如果有大量客户端并发下订单，Paxos能保证和预知按照某个顺序处理订单吗？ No，消息可能延迟或者丢失，相对顺序不确定。

就上面的例子，Paxos能让系统每秒处理更多的订单吗？ No，Paxos小组内的各个成员按照相同的顺序执行相同序列的操作，并没有提高并发度。但是可以间接帮助提高。

那Paxos到底有什么用？

1) 保证各个服务器M1, M2..Mk上处理订单的顺序是相同的，不论这个顺序具体是什么，不论各个订单到底是成功还是失败。 Yes

2) 只要故障的服务器个数<k/2，系统就能正常提供服务。 Yes

举例来说，假设开始时5个服务器状态均良好，订单B1, B2,..., B9都处理完成，其中B4, B7因为要求的蛋糕卖完了而失败，其他订单成功，且这些订单都是M1接手并处理的。随后，M1宕机，后续的请求发给了M2，那么M2看到的状态，就是M1处理了B9后的状态，必须完全一致!

Paxos的一些假设：

1) 网络传输：消息可以有大的延迟或者丢失，但是不能被篡改

2）成员状态：成员可以短暂或者永久离开，只要有过半成员正常运行，协议即可运转；成员必须记住自己就某件事自己给谁投过票。

# Paxos的推导过程理解

个人认为paper中先讨论的单实例，没有跟状态变化关联，后来突然冒出个多实例，是造成**难以理解的因素**之一。因此，下面的例子中，我们还是讨论谁的订单被第一个处理，谁的第二个被处理...，这样比较容易切换。

前面已经说明了单个服务器所存在的单点故障问题，也说明了多个服务器下，网络分割可能带来的脑裂问题。

Paxos的基本的想法是，多个服务器，组成一个Paxos小组或者委员会。系统不是某一个委员说了算，过半数同意通过的才能形成决议。一旦形成决议，则不能修改，所有的小组成员都必须遵从这个决定。比如，如果决定第一个被处理的订单时张三的，第二个是李四的，第三个被处理订单是王五的。中间的某个或者某些订单可能因为蛋糕卖完了而失败，但是这不是Paxos要解决的问题，那是每天供需关系问题。

为什么是一半以上？

1) 一半以上是必须的，否则可能只剩下一个蛋糕，而两个客户都预订成功了。

2）为什么不要求全部或者90%以上同意？ 要提高可用性，总得允许小组的成员们请个病假、婚假什么的，后面会看到一半以上是正确运转的底线！

**要考虑的因素：**

网络可能丢失数据、可能有很大延迟、可能产生分割。

上述问题可以归结为：各个服务器务必按照相同顺序处理订单，每个订单是否都成功不是一致性协议要解决的问题。在其中一个服务器故障后，如果还有其他服务器可以继续进行处理，那么切换过程不能造成混乱。即使发生切换，对客户来说，必须跟单个服务器在做处理是一样的结果，就是可用性更高了一点而已。具体可以理解为：

各个服务器都执行相同的订单序列，例如:

1号订单，是张三的2个草莓蛋糕

2号订单，是李四的3个慕斯蛋糕

3号订单，是王五的2个巧克力奶昔

...

注意，张三是不是真的下订单时间比李四早，在这里不重要，上面的三个人他们完全可能是同时下的订单，也可以是张三下订单比李四晚，但是张三的手机用的是5G网络。但是我们不在乎这个，只要各个服务器都认为张三的订单是1号订单，李四的是2号订单......就可以了。当然，排在后面的订单有可能买不着哦。

按照上面的理解，我们假设Paxos小组有5个委员(长老)组成，他们分别为：东邪、西毒、南帝、北丐、中通。

为什么是这五大长老? 主要从模型上比较匹配： 1）五大长老说话算数，不会赖账，而且记忆力很好；2)五大长老并不住在一起，通过写信完成讨论订单，信件可以被延迟或者丢失，但是江湖人士都认识五大长老的字迹和签名，不会被伪造；3)他们有自己的活动安排，可能去云游了一段时间再回来，云游期间不收信件，更不会答复。江湖人士都知道这五大长老，在下订单时，可以找这五大长老中的任何一个说明自己的需求，然后由五大长老代表江湖人士下订单，长老们同时有具有投票权。(其实下订单的完全可以不是这五大长老，但是那样画图起来费劲，我们知道提交订单(提议者)与投票者这两个角色可以独立就可以了！)



我们把不同的长老用不同的颜色代替，提议的颜色跟提议人相同，而举手的颜色，跟提议人的颜色一致。这样我们可以通过数红色的手的个数，确定几个人同意了南帝的提议。

Paxos初始规则 P1： 谁最先来我先投谁！

## 早起就是好办事

假设今天早上蛋糕店系统刚上线，南帝就收到了华筝的请求，说要买几个蛋糕。优于华筝起得太早，没人冲突，且各大长老都在，提议很快得到了一致同意，华筝的就成了第一个被处理的订单！



## 稍晚点出门就塞车

假设折天华筝、黄蓉和貂蝉（找个美女名字哈）起床差不多早，分别就近联系北丐、南帝和中通要下订单。三大长老分别提出了自己的主张，并且都得到了部分人的回应，如下图所示：



在上述情况下，我们必须要有规则来处理，最起码的一条：如果冲突了，得允许再投票，否则没法往下玩啦。

Paxos规则演变： 得允许一个长老投多次

## 别把长老们的名声搞坏啦

假设南帝最早提议华筝，并且有三个长老回信同意了，这就形成了决议，虽然七公和黄老邪根本不在场。



问题是: 七公在不知道决议形成的情况下，提议了黄蓉。对于已经在上轮投票的西毒和伯通来说，到底是支持还是不支持七公的提议呢？ 因为他们并不知道决议以及形成，只知道自赞成过南帝的提议而已！

1) 投过的不再投了。由于上次投票有没有形成决议不确定，可能造成一直没有形成决议。

2) 直接再投，可能形成了两次不同的决议，黄蓉和华筝都认为自己的订单是第一个被处理的!



Paxos规则演变(P2)：允许形成多次决议，但是决议内容必须相同！

能否让大家在提议前都问问到底有没有形成决议？

1. 以上图为例，唯一知道决议形成的段王爷，可能在答复华筝后云游去了，长期不回复其他消息，无法进展；
2. 并发始终存在，问完再提议，还是会出现冲突。例如七公和西毒在问询后，三个人都回复说之前没有给谁投票。因此，他俩各自提议，而且还是并发。



Paxos规则演变(P2a)： 一旦形成决议，即v被选中，那么后续只能accept V。

例子很简单，如果已经超过半数投票，黄蓉的订单被选中为第一个处理，那么后续就只能同意黄蓉的订单排第一。

如果长老能明确知道之前已经形成了决议，那很好处理。可惜我们不能这么假设，即使某个长老赞成过某提议，它也不一定能知道形成了决议。在这种情况下，如果又收到一个提议，到底是投？还是不投？

为了让协议变得可操作，进一步演变：

Paxos规则演变(P2b)： 一旦对某个序号形成决议，假设对应的值是v，那么后续的提议只能是V。

例子很简单，如果已经过半数的长老投票，黄蓉(v)的订单被选中为第1个处理(序号），那么后续只能允许提议黄蓉的订单排第一，不允许提议其他的订单排第一。

==> 隐含的要点： 提议人、提议内容(Value)的独立！ 因此，用选举描述这个过程，并不合适！跟选举的不同，V跟提议者可能完全是两个范畴的。

## 占坑，可能是为别人做嫁衣

对于提议来说，就变成需要保证两点：

1) 如果我提议前，形成了决议接受了v，那么我提议的，一定要是v;

2) 如果我提议前，并没有形成决议，那么我可以按照我的意愿去提议。

==》问题是： I从我询问完，到别人收到我的提议这段期间，可能在我不知情的情况下形成了某个决议！ II.或者别人跟我同时问，同时提议,各获得2票。

### PN是个什么鬼？

前面的决议，后面的提议，怎么区分谁先谁后？ 在异步网络环境下。两个并发的提议，可能在不同acceptor上看到的顺序不同，怎么能一致地做决定？ 不管是废掉谁，扶正谁。

由于发送和接收时间不能直接比较，只能使用**逻辑序号**了。

**怎么办？ 基于PN的占坑和承诺机制！**

占坑就是一直查询，占坑和提议，有时候是在用修改的形式，完成了一个读(比较低效的Learn)。比如，西毒通过一次完整的占坑和提议，确定了第1个执行的是黄蓉的订单，那么他下次提议时，就直接提议第2个执行的内容，而不再尝试争第1了。

为什么要有提议号？ ==>解决并发的问题，以及在问询后发现有多个已存value后的处理问题。才能让提议聚焦到一个v上。

提议号与提议内容的松耦合

华筝的订单怎么办？ 简单来说，如果提议者发现序号1已经确定，那么就可以提议其为序号2，递推下去，最终可能是排在第2, 3, 4...，只要形成一致即可。

### 提议者执行一轮协议的产出

对于一个提议者来说，执行了一轮两阶段协议(占坑和决议)的四个可能产出：

1. 被别人的提议给被废掉了，得重来；
2. 帮别人做了嫁衣(完成别人未竟的决议)；
3. 重复别人的决议，Learn了状态修改；
4. 让自己提议的内容，形成了决议。

## Leader的引入：效率，还是效率！

**上面的协议，不考虑后面提到的leader**，**到底能否让系统正确工作？** 先不说效率问题。

==>能，可能北丐在很多轮都是占坑被废，然后帮别人把序号i的决议形成或者重新做了一遍。然后再尝试i+1。这样可能很低效，有的请求可能非常倒霉地被推迟了很多轮。但是从系统的角度，再往前推进不断形成新的决议，且没有不一致问题。

效率体现在两个方面：

用最少的报文完成一个决议；

避免相互纠缠

1. ==> prepare request(proposal number n)

2. <==, response(保证不accept比n小的，保护我已经accept的最大的proposal number accepted.val, accpeted.num)

3.==> accept request(proposal number n, val v)

4.<== accept (n, v)

acceptor需要持久化的部分： max\_responded\_pn (proposal number), max\_accepted\_pn, max\_accepted\_val.

requestor需要持久化的部分：last\_proposed\_pn, last\_proposed\_val，需要二者一致

P1, P2, P2a, P2b, P2c之间的关系，已经在visio图中有，但是不美观。

既然一个value被majority个accept后，就称为被chosen，如果一个acceptor能够accept多个value，那么会形成混乱，多个都被chosen。因此，每个acceptor只能接受一个value。

为什么要有requirement P1?

**P1**. An acceptor must accept the first proposal that it receives //首个必须接受

==>不考虑消息丢失等问题，如果只有一个proposer，只propose了一个value，我们也希望能成功。所以必须acceptor必须接受其收到的第一个proposal。

按照P1，可能存在多个proposal被并发，每个都被少数派accept，从而不能形成结论，即不能chose任何value。为了保证这种情况下能够继续，或者多轮proposal能最终成功，就要求每个acceptor能够允许接受多个proposal。

为了不混淆，需要对各个proposal进行编号，编号方法另外描述。

既然允许每个acceptor接受多个proposal，并且一个proposal一旦被majority个接受，那么其value就被选择。那么就可能有多个value被chosen，即多次形成决议。注意，majority个acceptor接受即认为形成决议，但是proposer可能并未及时收到决议，或者决议本身丢失了，导致新一轮proposal。为了保证多次可能的决议的一致性，这就要求：

**P2**. If a proposal with value *v* is chosen, then every higher-numbered proposal that is chosen has value *v*. **//****只有一个value能被选中**

满足P1的一个自然想法是：

**P2*a***. If a proposal with value *v* is chosen, then every higher-numbered proposal accepted by any acceptor has value *v*. **//只允许接受已被选中的v**(在已有value v被选中的情况下)

由于通信是异步的，一个刚刚醒来的proposer，根本不能知道那些value被accept了，但是其proposal的number可能更大。按照p1，acceptor需要接受它，但是可能违反了P2a。为此，对P2a进行加强。即为了同时满足p1和P2a,P2a变成：

**P2*b*.** If a proposal with value *v* is chosen, then every higher-numbered proposal issued by any proposer has value *v*.  **//只允许提议已被选中的v** (在已有value v被选中的情况下)

如何满足P2b的一段推导，很有意思：

原文下面一段，需要好好理解：

Combining this with the induction assumption, the hypothesis that *m* is chosen implies:

Every acceptor in *C* has accepted a proposal with number in *m . .* (*n −* 1), and every proposal with number in *m . .* (*n −* 1) accepted by any acceptor has value *v*.

我认为这里面的C，可能只是个泛指。对于不同的proposal number，即*m . .* (*n −* 1)，对应的C可能是不同的。但是都有个majority集合。

Since any set *S* consisting of a majority of acceptors contains at least one member of *C* (这句话是很重要的过渡。前面没有提到S，为什么？), we can conclude that a proposal numbered *n* has value *v* by ensuring that the following invariant is maintained:

**P2*c***. For any *v* and *n*, if a proposal with value *v* and number *n* is issued, then there is a set *S* consisting of a majority of acceptors such that either (a) no acceptor in *S* has accepted any proposal numbered less than *n*, or (b) *v* is the value of the highest-numbered proposal among all proposals numbered less than *n* accepted by the acceptors in *S*. **//****总是提议能看到的PN最高的value**

实际上，这个不变式就是为了保证proposal number变化时，*v*的不变性。即要么S中任意一个都没有accept过小于n的proposal，这时候肯定没有形成一致，即没有一个value被accept了；要么v就是S中acceptor接受过的proposal number最高prosoal对应的value。

如何从P2c推出能达到P2b?

如果在PN 为m时，v被选中了，那么表示有个majority C，全部accept了v。

我们要论证按照P2C这个不变式，在PN为m+1.. n时，一定能会propose v。

在确定PN=m+1的proposal 的value x时，我们找到了某个majority，S中的最大value，那么S一定包含C中某个acceptor，因为二者都是majority。而m是最大值，所以一定选择对应m的value，即v。

在PN=m+2..n区间，考虑任意的PN i，由于在PN属于[m, i-1]的proposal 的value都是v，而S必然至少有一个与C重合，因此查到的PN最大的value，仍然是v。

highest-numbered 为什么重要？

首先，既然说highest，那么就隐含着，大家accept的proposal number可能是不同的。

其次，不存在number x, x > highest-number，且被majority个接受了。否则S中一定有至少一个接受了x，矛盾。

我理解，这样好实现，在后面，直接让acceptor承诺，不accept小于n的proposal。

注意，在讨论前面这些条件时，并没有牵涉到prepare 和 accept两个阶段，我认为是按照就一个阶段讨论的。

# Paxos理解的常见误区

1. 一想到投票，首先认为就是选举，认为结果就是某个提议者当选而已，因此想不明白有啥用，怎么用。其实最终的协议中，proposer和value完全可以是独立的。第一个想提议 v的提议者，可能在未能让过半数接受v的情况下宕机了，其他提议者帮助其完成了未竟的工作！只要我们把各个v看出例子中的订单请求，然后系统实际上决定各个订单的处理顺序，整个理解就具体了。
2. 由于论文中对P1, P2, P2a, P2b, P2c之间的联系总结很少，很多人看完这么多变化后迷失。为此，用下面的一张图来表述:

