**1. 概述**

    Paxos算法被用来实现一个容错的分布式系统，一直以来以晦涩难懂著称。这可能是因为该算法最开始使用希腊文表述的。事实上，它是所有分布式算法中最简单易懂的。Paxos算法的本质其实就是一个共识算法(我不太同意国内把consensus algorithm翻译成一致性算法，因为一致性的标准英文应该是consistency，这里面的consensus其实就是表述要达成一个共识的意思)——也称为会议算法(synod algorithm)。

    Paxos共识算法是由一些必须要被满足的基本条件(property)推导出来的，在下一节中我们会介绍这些基本条件。本文的最后一节将详细讨论完整的Paxos算法以及构建分布式系统时常用的状态机方法——该方法为人熟知，因为它经常出现在各种分布式系统理论文章中。

**2. 共识算法**

**2.1 问题由来**

    假设进程集合中的每个进程都能够发起包含一个值的提案。一个共识算法就是要确保在众多的提案之中只能有一个提案被接受。如果没有提案被提出，那么自然也就没有提案可以被接受。但一旦接受了一个提案，进程集合中的进程就会获知这一情况。这种达成共识的过程要求满足以下的条件：

* 只能接受发出的提案
* 只能接受一个提案
* 进程在提案选择未达成共识前是不能获知选择了哪个提案

    在这里我们并不详细讨论liveness的需求，但是大致的目标就是要确保某个提案最终会被选择到，并且一旦被接受，进程最终也会获知该提案。

    该共识算法中有三类代理角色：提案发起者(proposer)、接收者(acceptor)和学习者(learner)。在一个具体的实现中，单个进程可能同时兼任多种角色，但是本文并不讨论角色在进程间的分配。

    现在假设角色之间通过消息进行通信。我们使用一个常见的模型：异步的非拜占庭(non-Byzantine)模型。在该模型中：

* 代理角色(agent)处理速度各异且有可能会失败以及重启。由于在提案被选定之后所有的agent都有可能失败并重启，因此失败的agent必须要额外保存一些信息才行
* 发送消息速度也是各异，同时消息也有可能重复或丢失，但消息不能损坏

**2.2 选定提案值(choosing a value)**

    选定一个提案最容易的方式就是只创建一个接收者代理(acceptor agent)。一个提案发起者发送一个提案给接收者，而后者选定它接收到的第一个提案。这个方法虽然简单但却不能令人满意，因为这个接收者一旦宕机整个系统就不能用了——这就构成了一个单点失效。

    因此我们要尝试另一种方案来选定提案。这次我们使用多个接收者代理，某个提案发起者发送提案给一组接收者。一个接收者可能会接收提案中的值，也可能不会。只有当足够多的接收者都接受某个提案时才认为选定了这个提案。那么现在问题来了，足够多是什么含义呢？——若要确保只能有一个提案被选定，我们使用一个足够大的集合来表示半数以上的agent。由于任意两个半数以上的agent集合必定包含至少一个共同的接收者，因此如果限制一个接收者最多只能接受一个值，那么这个方案就必定是可行的。

    在消息不会失败或丢失的情况下，即使一个提案发起者只提交了一个提案我们也要选定那个提案。这就意味着以下的条件必须要满足：

**P1：一个接收者必须要接受它收到的第一个提案**

    但是这个条件引出了另外一个问题：不同的提案发起者几乎同时提出不同的提案，可能造成这样的局面：即每个接收者都接受过某个提案，但却没有一个提案被半数以上的接收者都接受。即使只有两个提案，如果每个都被不同的半数接收者接受，那么只要一个接收者宕机了都会让其他接收者没法获知选定的提案。

    上面提到的P1条件以及"提案只有被半数以上接收者接受才会被选定"的条件意味着一个接收者必须要能够接受多个提案。对于接收者可能会接受的提案，Paxos会给每个提案分配一个提案号(自然数)记录并区分不同的提案，所以一个提案其实是由一个提案号和提案中的值组成。为了避免歧义，Paxos要求每个提案都要有不同提案号。当然了，这依赖于算法的实现，但目前我们不管具体的细节。当某个提案被半数以上的接收者都接受时Paxos就会选定这个提案的值，此时我们可以说这个提案(以及它的值)被选定了。

    Paxos允许选定多个提案，但前提是确保这些被选定的提案必须具有相同的值。基于对提案号的推导，这足以保证：

**P2：如果被选定提案的值是v，那么后续被选定的具有较高编号的提案的值必须也是v**

    因为提案编号是有顺序的，条件P2就能严格地保证只能有一个提案值被选定。

    如果一个提案要被选定，那么这个提案至少要被一个接收者接受。因此，Paxos通过满足以下条件来达到满足P2条件的目的：

**P2a：如果选定提案的值是v，那么接收者如果要接受后续提案则它们的值也必须是v**

    Paxos仍然要维持满足条件P1以确保一定会选定某个提案。由于通信是异步的，一个提案被选定的时候可能存在某个特定的接收者c从来没有收到过任何提案。此时，如果一个新的提案发起者被唤醒并发起了一个具有更大编号但值不同的提案给c，那么根据条件P1，c就必须要接受这个提案，从而就违反了P2a的规定。此时怎么办呢？我们必须要同时满足条件P1和P2a。这就需要我们完善P2a的表述为：

**P2b：如果被选定提案的值是v，那么任何提案发起者发起的后续提案(具有更大的提案号)的值也必须是v**

    提案总要先发出才能被接受，所以满足了P2b自然就满足了P2a，因而也就满足了P2。

    下面要看一下如何才能满足P2b。假设一个提案的编号是m，值是v，我们需要证明任何后续的编号为n的提案(n > m)的值也是v。对n使用归纳法可能会让证明变得容易。即如果证明每个编号介于[m, n - 1]的提案的值都是v，那么我们就能证明编号为n的提案的值也是v。若要选定编号为m的提案，必然要存在一个集合C包含了半数以上的接收者已使得C中的每个接收者接受此提案。将这个提法与归纳法结合起来看就会发现提案m被选定其实表示：

* 集合C中的每个接收者都曾接受过[m, n - 1]中的某个提案，并且被某个接收者接受的编号位于[m, n - 1]间的每个提案，其值都是v。

    因为包含半数以上接收者的任何集合S都至少包含C中的一个成员，所以我们能够推断出编号n的提案的值是v，主要的依据就是：

**P2c：对于任意的v和n，如果值是v编号为n的提案被发出，那么存在一个包含了半数以上接收者的集合S，使得 (a) S中所有接收者都没有收到过编号小于n的提案；或者(b) 在S中接收者接受的编号小于n的所有提案内具有最大编号的提案的值是v。**

    因此如果满足了P2c条件也就满足了P2b。如果要保证满足P2c，想要发出编号n提案的发起人必须要获知已经或者将要被半数以上接收者都接受提案中，哪个提案的编号最接近于n(如果存在的话)。发现并找出那些已经被接受的提案很容易；而预测哪些提案会被接受则困难得多。因此我们并不做任何的预测，相反，提案发起者只是想保证说不要再接受这样的提案。换句话说，提案发起者只是要求接收者不要再接受那些编号小于n的提案。okay，我们现在就可以实现发送提案的算法了，算法如下：

1. 一个提案发起者选择一个新的编号n并发送请求给某个接收者集合中的每个接收者，并希望接收者：

    (a). 给出一个承诺，即不再接受任何编号小于n的提案，并且

    (b). 提供它接受过的编号最接近n的提案(如果存在的话)

Paxos把这样的请求称为编号为n的prepare请求。

2. 如果该提案发起者收到了半数以上接收者发过来的响应(response)，那么它就能发起一个编号为n值是v的提案，其中v是所有响应中编号最大的那个提案的值。但如果响应者并没有反馈任何提案给发起者，那么发起者就可以使用任意值。

    之后提案发起者发送提案请求给一组接收者，请求它们接受该提案。(响应请求的接收者集合不要求就得是响应初始prepare请求的那组接收者。) 我们将这个请求称为accept请求。

    okay，以上就是提案发起者的算法。那么接收者算法是怎么样的呢？它会从发起者处接收到两类请求：prepare请求和accept请求。原则上说，接收者即使丢弃这两类请求也不会破坏安全性。因此，只有它被允许去响应一个请求时我们才会显式地说明。接收者总是能够响应一个prepare请求。另外如果没有显式地被禁止，它也能够响应一个accept请求以接受收到的提案。换言之就是：

**P1a：当且仅当一个接收者没有响应过编号大于n的prepare请求时它才能接受编号为n的提案**

我们可以发现，P1a实际上包括了P1的表述。

    现在我们已经可以有了一个完整的算法来选定提案值以满足那些必须条件——当然一切的前提是要假设每个提案的编号是唯一的。最终的算法只是引入了一处小小的优化。

    假设一个接收者收到了一个prepare请求，编号是n，但是它之前已经响应过编号大于n的prepare请求了，也就是说它不应该再响应任何编号为n的新提案了。接收者此时也不能再响应这个新收到的prepare请求，因为它将不会再接受编号是n的提案。于是我们要让接收者忽略这样的prepare请求，同时这个接收者也要忽略那些包含它已接受提案的prepare请求。——这就是上一段中说的优化。

    对算法进行这样的优化之后，一个接收者只需要记住它接受过提案的最大编号以及它响应过的prepare请求的最大编号即可。不管成功或失败，P2c都是要被满足的。基于这个原因，一个接收者就必须要记住这些信息，即使是在它失败重启之后。值得注意的是，提案发起者总是能够丢弃一个提案并删除所有与该提案相关的信息——只要它从没尝试发起编号相同的另一个提案。

    把提案发起者与接收者的行为结合在一起，我们可以看出Paxos算法的执行分为以下两个阶段：

**阶段1**

　　(a) 提案发起者选择提案编号n并使用该编号发送一个prepare请求给半数以上的接收者

　　(b) 如果有接收者收到了编号n的该prepare请求且它之前响应过的所有prepare请求中的编号都不大于n，那么接收者将响应该请求并保证以后再也不会接受编号小于n的任何提案并且会返回它接受过的具有最大编号的提案(如果存在的话)。

**阶段2**

　　(a) 如果提案发起者从半数以上的接收者处收到了它发出的prepare请求(编号是n)的响应，那么它就会发送一个accept请求给对应的每个接收者请求编号是n值是v的提案被接受。其中v是所有响应包含的提案中最大编号的那个提案的值或者是一个任意值如果反馈的响应中压根就不包含任何提案的话。

　　(b) 当接收者收到一个包含编号n提案的accept请求后，它会进行判断：如果它之前已经响应过编号大于n的prepare请求了，那么它就不会接受该提案；否则它一定要接受该提案。

    只有遵循上述算法中的步骤，一个提案发起者是可以发起多个提案的。在整个过程中间它也可以在任意时候丢弃提案。(当然了还是要保证正确性，即使提案被丢弃很长时间之后请求和响应才各自被接收者和发起者收到。) 如果某个发起者已经开始尝试去发送一个更高编号的提案了，那么此时丢弃那个较小编号的提案似乎是个不错的选择。因此，如果一个接收者由于已经收到了更高编号的prepare请求而将某个prepare请求或accept请求忽略的话，那么该接收者也应该通知对应的提案发起者，让它也丢弃它的提案。这样做既不破坏正确性还又优化了算法的性能。

**2.3 获知被选定提案值(learning a chosen value)**

    若要获知一个提案值被选定出来，一个学习者(learner)必须能够识别出已经被半数以上接收者接受的提案。最简单的做法就是每个接收者，一旦它接受了某个提案就响应所有的学习者，把该提案发给它们。这会让学习者尽快地获知某个提案已被选定，但这种方案需要每个接收者单独响应每个学习者——要发送的总响应数是接收者数目与学习者数目的乘积。

    如果不出现拜占庭式错误(Byzantine failure)的话，一个学习者可以很容易地从另一个学习者处获知某个提案值也被选定。我们可以让接收者将它们接受提案的信息发送给一个统一的学习者(distinguished learner)，然后由这个统一的学习者发送给其他的学习者，告知它们某个提案值已被选出来了。这个方法额外需要一轮通信才能让所有的学习者都能获知被选定的提案值。同时，该方案可能性也不高，因为这个统一的学习者有可能是一个单点失效。这个方案要求的响应数必须等于接收者与学习者数目总和。

    我们对上面的方案进行改进，这次接收者将它们接受提案的信息发送给一组学习者，然后该组中的每个学习者再去通知其他的学习者。使用一组学习者具有更好的可靠性，但代价就是会引入更多的通信。

    由于消息可能会丢失，因而可能出现这样的情景：即当某个提案值被选定出来时没有学习者能够获知。学习者本可以去询问接收者接受了哪些提案，但接收者的失败可能会让这变得不可能，也就是说学习者没法获知是否出现过半数以上的接收者都接受过某个提案的情况。如果是这样的话，只有选定新的提案后学习者才有可能感知到。因此如果学习者需要知道是否有提案被选出，它可以驱动提案发起者使用上面描述的算法新发起一个提案。

**2.4 进度**

    我们可以很容易地构建这样的一个场景：两个发起者进程各自不断地发起编号递增的一组提案，但没有一个提案胜出。发起者p完成了编号n1的提案的阶段1，而另一个发起者q也完成了编号n2>n1提案的阶段1。此时发起者p的阶段2中的编号n1的accept请求就会被忽略掉，因为接收者要确保不再接受任何编号小于n2的提案。于是，发起者p使用编号n3 > n2开启并完成了新的阶段1，导致发起者q的第二个阶段的accept请求也被忽略掉了。因此这种"悲剧"会不断地继续下去，整个共识算法不会有任何进展。

    为了应对这种情况，Paxos选择一个特殊的发起者，规定只能由它来发送提案。如果该发起者能够成功地与半数以上的接收者通信且它使用的提案号比已使用的都要大的话，那么它发出的提案就会被选定。如果使用的提案号比某个请求的编号要小，该发起者会放弃较小编号的提案并不断重试直到最终它选择了一个足够大的提案号。

    如果系统大部分组件(发起者，接收者和网络)工作正常的话，使用这种单个特定发起者的方案是能够避免liveness问题(Liveness问题大概可以理解为因为某些原因，比如死锁或饥饿等使得系统没法向前推进)的。很多论文提出选择一个发起者的可靠算法都必须要使用随机或实时来完成——比如使用超时。但不管怎么样，无论选举成功还是失败，安全性都是要首先保证的。

**2.5 实现**

    Paxos算法涉及了一组进程网络。在它的共识算法中，每个进程都同时扮演了提案发起者、接收者和学习者的角色。算法首先会选取一个领导者进程来充当特定发起者和统一学习者的角色。Paxos共识算法就是我们上面描述的算法——在该算法中，提案的请求与响应都是作为普通的消息被发送与传递。(响应消息将打上对应的提案号标签以关联对应的请求并阻止歧义。) 实现时要使用持久化存储来保存接收者必须要记住的信息以应对接收者的失败。接收者在发送响应之前也会在持久化设备中保存它们。

    下面就该说说如何确保不会发出编号相同的多个提案。不同的发起者进程需要从不相交的编号集合中选择提案号，这样两个不同的发起者进程永远不会发出相同编号的提案。每个发起者将它发送过的最大编号的提案保存在持久化设备中，然后使用一个更大的编号开启算法的阶段1。

**3. 实现一个状态机**

    实现分布式系统的一个简单方法就是实现一组客户端程序向一个集中式的服务器发送命令。这个服务器可以被实现为一个确定性状态机，它按照某种顺序执行客户端命令。这个状态机从当前状态接收一个命令作为输入并产生对应的输出，然后转换到另一个状态。举例来说，分布式银行系统的客户端程序可能是操作的柜员，而状态机由所有用户的账号余额组成。执行取款的操作涉及到执行一个状态机命令让其在满足取款金额的前提下减少一个账号的余额，并返回取款前后的余额。

    单个集中式服务器的实现中单个服务器构成了一个单点失效。于是，我们可以使用一组服务器，每一个都独立地实现了一个状态机。因为状态机是确定性的，只要它们执行相同的额命令序列，所有的服务器都会产生相同的状态序列以及相同的输出。一个发起命令的客户端能够随意使用任何服务器返回的输出结果。

    如果要确保所有服务器都执行相同的状态机命令序列，我们要实现一组独立的Paxos共识算法实例。第i个实例选定的值就是序列中的第i个状态机命令。每个服务器在每个算法实例中都扮演了所有的角色(发起者、接收者、学习者)。现在让我们假设这组服务器是固定的，那么所有的算法实例都是用相同的服务器集合。

    在一般性的操作中，某个服务器被选定为领导者，它在共识算法的所有实例中都扮演了特定发起者(也就是唯一尝试发起提案的进程)的角色。客户端发送命令给这个领导者，领导者稍后决定每条命令在序列中的位置。如果领导者说某条客户端命令的序号是135，那么它就让第135号算法实例选定该条命令作为被选定的提案。通常情况下这是会成功的，当然也有可能失败，比如出现服务器失败或有其他服务器认为它才是leader且它认为第135号命令应该去别的地方的情况。但是不管怎么样，共识算法能够确保最多只有一条命令能够被选定为第135号命令。

    这个方法高效率的关键在于，Paxos算法中只有在第二阶段才会选定提案值。回想一下，阶段1完成之后可能出现的结果是1. 确定了被发起的值或；2. 发起者还可以提出新的值。

    现在先说说Paxos状态机实现是如何工作的，后面再讨论可能出错的情况——主要涉及领导者挂掉了之后新领导者被选出后如何应对。 (系统启动是特殊情况，因为此时没有任何命令被发出)

    作为所有算法实例的一个学习者，新的领导者应该感知到大部分已经被选定的命令。假设它已知晓命令1-134,138和139——也就是说，在实例1-134， 138和139中被选定的命令。(我们稍后会发现这样的一个命令序列间隙是如何产生的)。于是，它会执行实例135-137以及所有大于139实例的阶段1部分(后面会描述这是怎么做的)。假设执行完的结果确定了在实例135和140中被选定的命令，但是该命令在其他实例中并没有什么限制。领导者于是为实例135和140执行算法的阶段2，因此也就选择命令135和140。

    现在，领导者以及那些与领导者知道一样多的其他服务器都可以执行命令1-135了。但是它不能执行138-140，这一点领导者也是知道的，因为命令136和137还没有被选出。领导者当然可以使用客户端请求的下两条命令作为命令136和137，但我们只是想马上填充这个坑，做法就是发起一个特殊的"什么都不做(no-op)"命令作为136和137。该命令并不会修改状态。(执行136和137实例的阶段2) 一旦这两个no-op操作被选定，命令138-140就能够被执行了。

    命令1-140现在就可以被选定了。同时领导者也已完成了大于140的所有实例的阶段1，现在它可以随意地在这些实例中的阶段2发起任意提案值。领导者会将141号命令分配给客户端发起的下一条命令，并在141号实例阶段2中进行提交。之后它会将接收到的下一个客户端命令作为命令142发出，并依次类推。

    有一种情况是，领导者获知它发出的141号命令已经被选定之前就发出了142号命令。有可能在处理141号命令时发送的所有消息都会丢失且在其他服务器获知领导者发起的141号命令之前142号命令就被选定了。如果领导者没有收到141实例中阶段2消息的响应，它会重发这些消息。如果所有都正常的话，它发起的命令就会被选定。但是领导者有可能先失败了，导致命令序列中出现空隙。通常来说，我们会假设一个领导者领先其他的学习者α条命令——也就是说，在确定了1到i号命令之后领导者就能够发起i + 1到i + α号命令的决议了，因此也就产生了α -1 条命令的空隙。

    新选定的leader为无限多的算法实例执行阶段1——在上面的例子中是对实例135-137以及所有大于139的实例。它会发送一条足够小的消息给其他服务器来实现为所有实例使用相同的提案号。在阶段1， 只有当接收者已经从某个发起者处收到了一个阶段2的消息时才会返回更多的消息。(在这个场景中，只有实例135和140属于这种情况) 因此，一个服务器(接收者)能够响应所有实例一条非常短的消息。于是执行无限多的实例并不是问题。

    由于领导者挂掉并选举新领导者的几率很低，执行一个状态机命令的实际代价——即对命令或提案值达成共识的代价——其实就只是执行算法阶段2的代价。我们可以证明Paxos算法阶段2的代价可能是所有需要容错的共识算法中代价最小的。因此Paxos本质上是非常高效的。

    上面关于系统正常操作的讨论都是假设总是存在一个领导者进程，且领导者挂掉与新领导者被选举出来之间的间隔很短。在有些异常情况下，领导者选举也有可能失败。此时如果没有服务器充当领导者，就没法执行任何新的命令。如果多个服务器都认为自己是领导者，那么它们都可能往同一个算法实例中发起提案值的申请，这可能使得所有提案或命令都不会被选出。但是就像前面说过的，安全性是一定要保证的。两个不同的服务器对第i个状态机的命令不能有分歧。选举一个领导者只是为了确保共识算法能够继续推进下去。

    如果服务器集合发生变更，必须有一种机制可以检测到哪些服务器实现了共识算法的哪些实例。最简单的方式就是通过状态机本身。当前的服务器集合作为状态的一部分也可与普通的状态机命令一起进行修改。一个领导者进程可以提前获得α条命令，具体的做法就是在执行完第i个状态机命令后由状态来指定一组服务器来执行(i + α)号实例。这就是复杂重配置算法的一个简单实现。