Distributed Snapshots-Determining Global States of a Distributed System

作者：Chandy & *Lamport* 1985

原文：Determining-Global-States-of-a-Distributed-System.pdf

作者：phylips@bmy 2018-10-27

译文：http://duanple.com/?p=1066

[序：分布式 Snapshot 算法，又称为 Chandy-Lamport 算法。分布式 snapshot 的难点在于在没有全局时钟及全局观察者的情况下，各个节点各自异步记录的状态很可能是不一致的。本文描述了一种用于记录异步分布式系统的一致性全局状态的算法。分布式 snapshot 对于解决分布式系统中的一些基本问题具有重要意义，比如它可以用来进行死锁检测、系统终止检测或者是用来验证某些全局性的 stable property 是否成立。同时 Chandy 和 Lamport 提出的算法也非常优雅和简单，这也是它之所以这么成功的原因。

Chandy-Lamport 算法流程实际上很简单，而本文真正的难点在于如何证明通过这个算法记录的全局状态是有效的，因为通过该算法记录的这个状态可能在系统实际的执行过程中根本就没有出现。本文第一次清晰定义了在分布式系统中何谓一致性的全局状态，一致性的全局状态是我们观察异步分布式系统的重要基石。后续提出的很多重要概念比如 vector clock、执行过程的同构性、全局性的断言检测、concurrent common knowledge 等都是基于一致性的全局状态。

另外关于该算法的一个最新应用是 Flink 中的 checkpoint 机制。

]

1. Stable Property

令 y 是关于分布式系统 D 的全局状态 S 的一个断言函数，也就是说 y(S)的值为 true 或 false。如果对于 D 来说，所有从 S 可达的状态 S’，都满足 y(S)为 true 可以推出 y(S’)为 true，那么我们就说断言 y 就是系统 D 的一个 stable property。换句话说，如果 y 是一个 stable property，那么如果 y(S)在系统 D 的某一个时间点上为 true 此后将一直保持为 true。像“计算终止了”、“系统死锁了”、“token ring 上的 token 都消失了”就属于 stable property。

如果可以确定分布式系统的全局状态，就可以通过这个全局状态进行 stable property 的检测，比如可以检查系统是否发生了死锁、运算过程是否终止等。

2.分布式系统模型

我们把分布式系统看做是由进程集合和进程间通信的 channel 集合组成，可以用一个有向图来进行表示，图中顶点代表进程，边代表 channel。如下图所示：

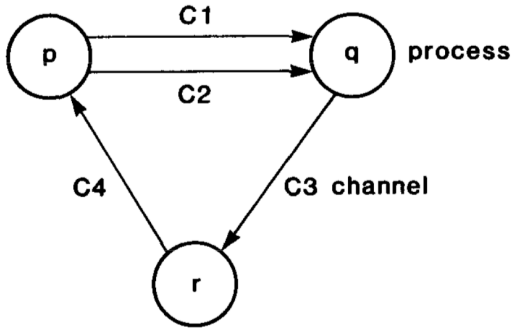

Fig. 1. A distributed system with processes *p*, *q*, and *r* and channels c1, c2, c3, and c4.

2.1 Channel

我们假设 channel 具有无限大的 buffer、传输中没有错误产生、消息按序发送、消息可以在 channel 中停留任意长的时间。对于一个 channel 来说，通过该 channel 接受到的消息序列，是通过该 channel 发送的消息序列的子序列。将通过该通道发送的消息减去已经接受到的消息后，剩余未被接受的消息序列就是该 Channel 对应的状态。

2.2 Process

进程可以通过一组状态集合、一个初始状态(来自于状态集合)和一组事件集合进行定义。

2.3 事件

事件(设为 e)则是进程(设为 p)上的一个原子动作，该动作可能会改变进程自己的

状态，以及与进程相连的最多一个 channel(设为 c)的状态：当沿着 channel c 有消息从 p 发送或者被 p 接收时 c 的状态就可能会被改变。一个事件定义包含如下要素：

(1)事件发生所在的进程 p

(2)事件发生前，进程 p 处于的状态 s

(3)事件发生后，进程 p 处于的状态 s'

(4)被事件改变了状态的 channel c(如果有的话)

(5)消息 M，(如果有的话)。如果 c 是从 p 发出，该消息就是沿着 c 向外发送，如果 c 是指向 p，该消息就是沿着 c 被 p 接收的。

如上，可以通过一个五元组(p, s, s', M, c)定义事件 e，另外如果 e 不会改变任何 channel 的状态的话，M 和 c 就都是 null。

2.4 分布式系统的全局状态

分布式系统全局状态可以看做是由进程和 channel 的状态组成的集合。在初始的全局状态中，所有进程都处于各自的初始状态，所有 channel 的状态都是空的消息序列。事件的发生可能会改变全局状态。令事件 e=< p, s, s', M, c>，e 能够发生在全局状态 S 当且仅当：

(1) 进程 p 在全局状态 S 中的状态为 s

(2) 如果 c 是指向 p 的 channel，那么 c 在 S 中的状态是一个以 M 为首的消息序列

定义一个 next 函数，next(S, e)代表全局状态 S 在发生事件 e 后处于的状态。首先只有当 e 能够发生在全局状态 S 的情况下 next(S, e)的值才有意义，在这种情况下，next(S, e)的值除下面几点外与 S 完全一样：

(1) p 在 next(S, e)中的状态为 s'

(2) 如果 c 是一个指向 p 的 channel，c 在 next(S, e)中的状态就是 c 在 S 中的状态去掉队首的 M

(3) 如果 c 是从 p 发出的 channel，c 在 next(S, e)中的状态就是 c 在 S 中的状态再在队尾加上 M

令 seq=(ei, 0 =< i <= n)，代表分布式系统中的事件序列。我们说一个 seq 是系统的 computation，当且仅当：

事件 ei 可以发生在全局状态 Si，0 =< i <= n，S0 代表初始的全局状态；并且

Si+l = next(Si, ei) 0 =< i <= n.

以一个具有两个进程，两个 channel 的“单 token 会话系统”为例。如下：

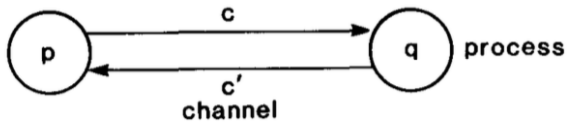

Fig. 2. The simple distributed system of Examples 2.1 and 2.2.

每个进程有两个状态，s0 代表进程不持有 token，s1 代表持有 token，整个系统只有一个 token，该 token 在两个进程间流转。对应的全局状态及其转换过程如下：

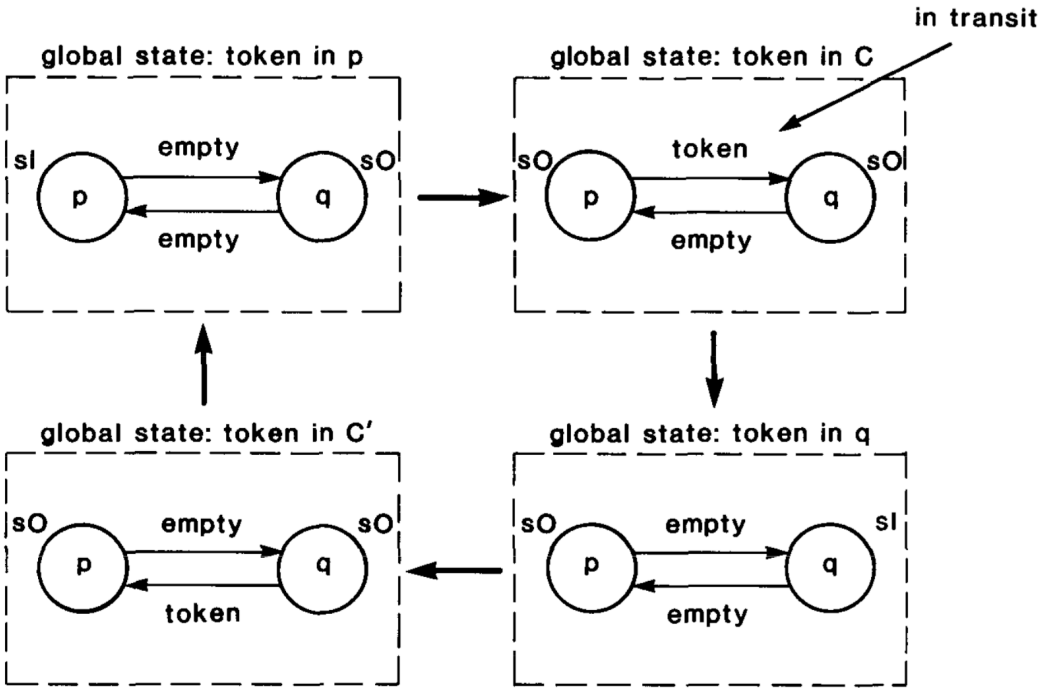

Fig. 4. Global states and transitions of the single-token conservation system.

如上图中，代表了 4 个全局状态，我们根据 token 的位置把它们简称为：(1)in-p (2)in-c (3)in-q (4)in-c’。(1)(2)(3)(4)就是一个 computation。

3.算法

3.1 基本要求

全局状态记录算法与底层计算重叠，它与底层计算同时进行但是不会改变底层计算。该算法可能需要发送消息和需要节点参与计算，但是它们不能干扰底层计算。

以上面的“单 token 会话系统”为例，令 c 代表从 p 到 q 的 channel。通过这个例子我们可以来看下状态记录需要满足的一些基本要求。假设记录进程 p 的状态时系统处于全局状态“in-p”，之后全局状态变成了“in-c”，而进程 q 和 channe c 和 c’的状态记录就是在“in-c”状态的时候。这些记录的状态，组合成一个全局状态，但是在这个状态里有两个 token，一个在进程 p 了另一个在 channel c 里。但是对于一个“单 token 会话系统”，是不可能从初始只有一个 token 的全局状态变成具有两个 token 的全局状态的。这种不一致的产生，是由于 p 的状态记录是在 p 沿着 channel c 发送消息之前，而 c 的状态记录是在 p 沿着 c 发送这个消息之后。令 n 表示记录 p 的状态之前 p 沿着 c 发送的消息数，令 n’表示记录 c 的状态之前 p 沿着 c 发送的消息数。我们的例子表明，如果 n<n’，记录的全局状态可能会出现不一致。

现在考虑另外一种场景，channel c 的状态记录是发生在全局状态“in-p”，而进程 p、q 和 channel c’的状态记录是发生在全局状态“in-c”。这样记录下的全局状态则显示系统中没有 token。这个例子表明如果 c 的状态记录是在 p 沿着 c 发送消息之前，p 的状态记录是在 p 沿着 c 发送消息之后，即 n>n’的情况下，记录的全局状态也可能会出现不一致。{!被发送的消息可能在 channel 中，也可能到达了目标进程}

通过上述两个例子，可以看出要得到一个一致的全局状态，需要：n=n’

令 m 表示记录进程 q 状态之前沿着 c 收到的消息数，m’表示记录 channel c 的状态之前沿着 c 收到的消息数，类似地如果要确保是一致的全局状态需要：m=m’。{!消息被收到，是指消息已经不在 channel，已经到达了目标进程 q}

另外对于每一个状态来说，沿着 channel 收到的消息都不会超过沿着 channel 发送的消息数，即：n’ >= m’。{! n’和 m’都有同一个观察点，都是从 channel c 的角度看。分别代表了在一个记录下的全局状态中，记录 channel c 的状态之前，沿着 channel c 发送的消息数和收到的消息数。而沿着 channel 发送的消息可能已经被收到，也可能还在 channel 中，因此 n’是 m’再加上还在 channel 中的消息数}

再根据 n’=n 和 m=m’，可以得出 n>=m。{!这里 n>=m，再加上后面的 marker 算法，容易让人产生一个误解：对于同一个 channel 连接的两个进程来说，发送者要一定要比接受者先记录状态，很明显这样是有问题的，因为如果是这样的话，如果两个进程间有 p->q 的 channel，也有 q->p 的 channel 的话就会产生矛盾。实际上对于 n>=m 这个条件来说，p 和 q 谁先记录状态都可以，因为消息可能还在 channel 中。实际上接受者先记录状态，反而更容易满足 n>=m，因为发送者后记录状态，它就可以发送更多的消息，这样发送的消息就更容易大于接收到的，发送者记录的状态与接受者记录的状态再加上停留在 channel 中的状态就组成了一个一致的状态}

要求 n>=m，实际是因为 p 和 q 是各自分别自己记录状态的，无法确保是二者是在同一时刻进行记录的，而通过 n>=m 确保了 p 和 q 记录状态时，只是差了 channel 中的消息，只要把 q 记录的状态再加上 channel 中的消息，就等价于它们两个同一时刻记录的状态。因此 **channel c 需要记录的状态，实际上就是进程 p 记录状态之前沿 channe c 发出的但是进程 q 记录状态之前还没收到的消息序列**。也就是说如果 n’=m’，这个序列就是空的，如果 n’>m’，这个序列就是 p 沿着 c 发出的第 m’+1...直到第 n’个消息。为了找到这中间在 channel c 中滞留的消息序列。我们可以让进程 p 发送一个特殊的消息(称之为 marker)，在进程 p 发送第 n 个消息之后发送它，并且保证在发送这个 marker 之前不会再发送其他消息，而 maker 本身不影响底层计算。这样 channel c 的状态，就是进程 q 记录了自己的状态之后直到收到 marker 之前收到的那些消息序列组成。为了保证 n>=m 成立，进程 q 在收到 maker 的时候，如果它还没有记录自己的状态，就必须要记录自己的状态，因为此时进程 q 收到的消息数已经等于进程 p 记录状态之前发送的消息数了，再不记录将 m 将大于 n 了{!如果在后面才去记录 q 的状态，就会出现 q 的状态比 p 超前，而 p 的状态变化是无法用 channel c 上的消息来进行状态补偿的}

3.2 具体算法

3.2.1 针对进程 p 的 maker 发送规则

对于所有以 p 为源节点的 channel c，在 p 记录了自己的状态之后，发送其他消息之前，沿着 c 发送一个 marker 消息。

3.2.2 针对进程 q 的 maker 接收规则

在收到一个沿着 channel c 的 maker 之后，如果 q 还没有记录自己的状态，那么：

q 记录它自己的状态；

q 把 c 的状态记录为空序列。

否则：

q 记录 c 的状态，此时 c 的状态由 q 记录自己的状态之后到收到 maker 之间的消息序列组成。

3.3.3 算法的终止性

maker 收发规则保证了如果每个 channel 上都收到了 maker，那么所有进程将会记录下它自己以及它的所有 incoming-channel 的状态。为了保证全局状态记录算法会在有限时间内终止，每个进程必须要保证：

(1) maker 不会一直停留在进程的 input channel 中

(2) 进程自身状态的记录必须在有限时间内完成

如果进程 p 记录了自己的状态，同时有一个 channel c 是从 p 到 q 的，那么 q 将会在有限时间内记录自己的状态。因为 p 将会沿着 channel c 发送 maker，而 q 会在有限时间内收到它(根据上面的条件 1：maker 不会一直停留在 q 的 input channel 中)。通过归纳法可以得出，对于进程 p 可以通过某个 path 到达的那些进程来说，它们都可以在有限时间内记录下自己的状态。特别地，如果进程和 channel 的有向图是强连通的，那么只要有一个进程记录它自己的状态，那么最终所有进程都会记录下自己的状态。

{!上面的算法天然地可以处理图中有环的情况，具体可以通过下面 4 中的例子对这种有环图怎么进行 snapshot 记录有一个更直观的感受。在有环图中，节点的状态也是只会记录一次，同一个节点可能会收到不同的 channel 发来的 marker，节点本身的状态只会记录一次，通过 channel 的状态来对两边节点状态的差异进行补偿即可。比如 p->q->p 的情况，p 先记录状态，然后发送 maker 到 q，q 记录状态，之后 q 再发送 maker 到 p，此时 p 已经记录过状态了，p 不需要重新记录一个状态，只要把 p 上次记录状态之后到收到 q 发送的 maker 之前，从 q 收到的那些消息作为 q->p 这个 channel 的状态记录即可。

}

4. 全局状态的有效性证明

首先通过一个例子，来加深下对于上述算法和通过上述算法得到的全局状态的理解。由进程和 channel 组成的有向图如下：

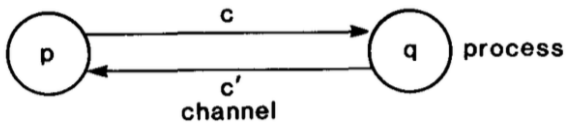

Fig. 2. The simple distributed system of Examples 2.1 and 2.2.

同时具有以下图中描述的状态转换规则和过程：

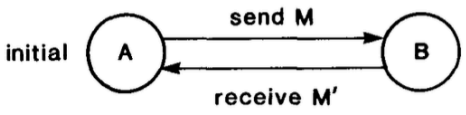

Fig. 5. State-transition diagram for process p in Example 2.2.

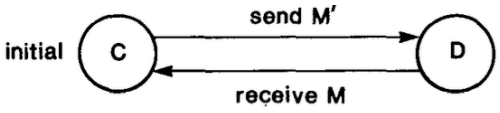

Fig. 6. State-transition diagram for process q in Example 2.2.

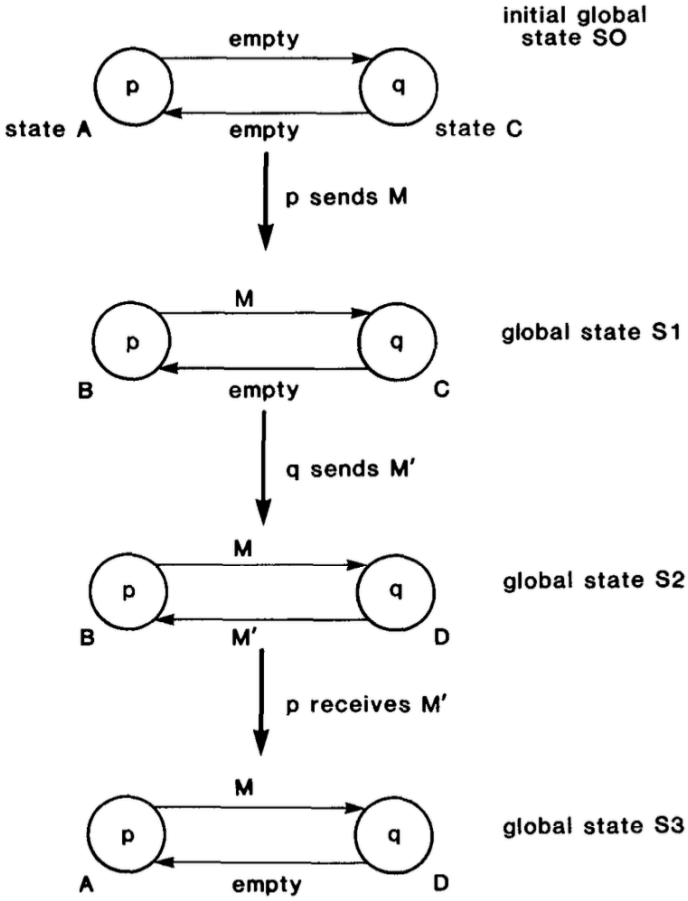

Fig. 7. A computation for Example 2.2.

假设进程 p 的状态记录时系统处于全局状态 S0，此时进程 p 处于状态 A。在 p 记录完自己的状态后，沿着 channel c 发送了一个 marker。现在假设在 maker 还在 c 中的时候，系统进入了全局状态 S1，然后是 S2，然后是 S3，在 S3 的时候，maker 被 q 所接收。在收到该 marker 后，进程 q 记录下了自己的状态，此时它的状态是 D，同时记录下 channel c 的状态为空消息序列。在 q 记录完自己的状态后，沿着 channel c'发送了一个 maker 消息。在收到该 maker 消息后，进程 p 记录下了 channel c'的状态为包含一个消息 M'{!因为自 p 记录自己的状态开始到沿着 c'收到 maker 为止，它沿着 c'只收到了一个消息 M'}。令上述过程记录下的全局状态为 S*，S*的具体内容如下图所示：

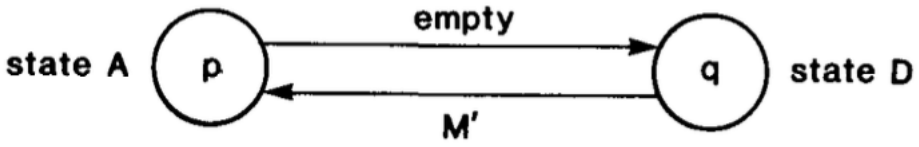

Fig. 8. A recorded global state for Example 2.2.

该记录算法开始时系统处于状态 S0，结束时系统处于状态 S3。但是我们观察记录下的全局状态 S*，却会发现该状态与整个执行过程中的 S0、S1、S2 和 S3 均不相同。如果该算法记录的是一个系统运行中从来没有发生过的全局状态，这还有什么意义呢？通过如下的证明可以回答这个问题。

令 seq=(ei, i>=0)表示一个分布式 computation，同时令 Si 代表 seq 中的事件 ei 发生之前那一刻的系统全局状态。假设上面的 snapshot 记录算法开始运行时系统处于状态 S ι，结束系统处于于全局状态 S Φ，0=< ι <= Φ。换句话说，该算法的初始化是在事件 e ι -1(if ι >0)之后 ei 之前，结束是在事件 e Φ-1(if Φ>0)之后，e Φ之前。正如我们在上面的例子看到的，记录下的系统全局状态 S*可能与所有的全局状态 Sk, ι =<k<= Φ。但是我们可以证明如下命题：

(1)全局状态 S*从 S ι 可达，并且

(2)全局状态 S Φ从 S*可达

具体来说，我们可以证明存在一个满足如下条件的 computation seq'：

(1) seq'是 seq 的一个排列{!事件集合相同只是事件发生的先后顺序不同}，S ι ,S*,S Φ是在 seq'里面真实发生的全局状态，

(2) S ι =S*或者 S ι 发生在 S*之前，并且

(3) S Φ=S*或者 S Φ发生在 S*之后

定理 1.存在一个 computation seq'，满足如下条件：

(1) 对于所有的 i，在 i< ι 或 i>= Φ时，ei'=ei，并且

(2) 子序列(ei', ι =< i < Φ)是子序列的一个排列(ei, ι =< i < Φ)，并且

(3) 对于所有的 i，在 i< ι 或 i>= Φ时，Si'=S，并且

(4) 存在某一个 k 值， ι =<k <= Φ，满足 S*=Sk'

证明：对于 seq 中的事件 ei 我们称之为“prerecording”事件，当且仅当在 seq

中它是发生在进程 p 并且进程 p 的状态记录是在 ei 之前。当且仅当它不是一个“prerecording”事件时—即在 seq 中它是发生在进程 p 但是进程 p 的状态记录在该事件之后，我们称之为“postrecording”事件。根据这个定义，对于所有 i< ι 的事件 ei，都属于“prerecording”事件，而所有的 i>= Φ事件 ei 都属于“postrecording”事件。对于 ι <j< Φ的那些事件 ej，可能存在 ej-1 是“postrecording”事件而 ej 是“prerecording”事件的情况{!也就是说在序列中，“postrecording”事件排在了“prerecording”事件前面}。对于这种情况来说，ej 和 ej-1 肯定是发生在不同的进程{!如果是同一个进程的话，postrecording 事件肯定发生在 prerecording 之后，不可能出现 postrecording 在前的情况}。

整个证明分两大步，第一步先说明对于事件发生序列 seq 来说，可以通过调整事件的排序，让所有的“postrecording”事件在“prerecording”事件前面；第二步，则是证明 S*这个全局状态就是在“postrecording”事件和“prerecording”事件中间分界点上的系统全局状态。

现在假设在 seq 中，有一个“postrecording”事件 ej-1 在“prerecording”事件 ej 之前。可以证明将 ej-1 和 ej 互换之后的序列仍然是一个 computation。如前所述，事件 ej-1 和 ej 发生在不同进程上，设 ej-1 发生在进程 p，ej 发生在进程 q。那么肯定不存在这样一个消息：该消息是在 ej-1 发出在 ej 被接收。原因有两点(1)如果消息是在事件 ej-1 发生时沿着 channel c 发出，那么肯定在该消息之前有一个 marker 沿着 c 被发出{!因为 ej-1“postrecording”事件，在它之前进程已经记录下了自己的状态，根据我们的算法，在进程记录下状态之后发送任何其他消息之前要先发一个 marker} (2)如果在 ej 发生时，该消息沿着 channel c 被收到了，那么 maker 肯定在事件 ej 发生前就被收到了(因为消息通道是 FIFO 的)，在这种情况下(根据 maker 接收规则)，ej 也应该是“postrecording”事件。

进程 q 的状态不会被事件 ej-1 改变，因为 ej-1 发生在另一个进程 p。如果在事件 ej 中，进程 q 沿着 channel c 收到了一个消息 M，那么该消息肯定是在事件 ej-1 发生之前就被发送了的，因为前面我们已经证明了在“postrecording”事件 ej-1 中被发送的消息不可能在“prerecording”事件 ej 中被接收。因此事件 ej 可以发生在全局状态 Sj-1，即事件 ej-1 发生之前那一刻。{!回忆下事件定义的几个要素，包含改变进程状态或者收发消息，我们可以看到因为 ej-1 没有改变进程 q 的状态，因此 ej 发生在 ej-1 之前，对应 ej 来说，它看到的进程 q 的状态是

不变的，对于接收的消息来说，因为 ej-1 不会发出在 ej 接收的消息，因此把 ej 放前面从消息接收这一层面也是没有问题的。另外因为我们本身观察到了 ej 发生在 ej-1 之后，因此是不可能有一个消息是在 ej 发出然后被 ej-1 接收的，因此从消息发送这一层面看也没有问题}

进程 p 的状态不会被事件 ej 的发生改变，因此 ej-1 可以发生在 ej 之后。因此事件序列 e1,e2…ej-2,ej,ej-1 是一个 computation。同时通过上面的结论，容易得出事件序列 e1,e2…ej-2,ej-1,ej 之后系统的全局状态与 e1,e2…ej-2,ej,ej-1 之后达到的系统全局状态是相同的。
{!上面这个过程实际上与 Lamport Logical Clock 及 Vector Clock 具有很大的联系。实际上，上面的事件 ej 和 ej-1 可以互换就是因为它们之间没有 happen before 关系，而上面的证明实际上也是在说这件事情。而 seq 可以看做是我们为事件分配 logical clock 后根据这个 clock 记录下的一个事件发生顺序，虽然记录下的顺序是这样的，但是实际上有些事件是并行的，没有 happen before 关系，对于这些事件来说它们的发生顺序实际上本身是没有定义的。而通过 maker 机制则确保了上述 ej 和 ej-1 是没有 happen before 关系的，而这一点与 vector clock 可以表明事件没有 happen before 关系的功能类似}

令 seq*代表 seq 中 ej 与 ej-1 互换后的事件序列，seq*也是一个 computation。令 Si*代表 seq*中第 i 个事件 e 发生之前那一刻的全局状态。根据上面的结论可以得出对于所有的 i，除了 i=j 的情况外，均满足 Si*=Si。

通过不断的对 seq 中相邻的“postrecording”事件在“prerecording”事件之前的情况进行 swap。我们可以把所有的“prerecording”事件调整到“postrecording”事件之前。{!我们可以通过归纳法进行证明，首先对于 1 个事件的情况满足 prerecording 事件在前的条件，如果是 2 个事件，如果不满足直接 swap 即可，现在我们假设前 i 个事件已经满足 prerecording 在前，来看第 i+1 个事件，如果它是个 postrecording 事件，那么也满足，如果是个 prerecording 实际，那么就把它与第 i 个事件交换，如果第 i-1 个事件还是 postrecording 事件，继续交换，直到遇到 prerecording 事件或者到头，最终前 i+1 个事件也会变成 prerecording 在前，重复执行这个过程，最终所有的事件就会变成 prerecording 在前}

下面我们来完成证明的第二步：S*这个全局状态就是在“postrecording”事件和

“prerecording”事件中间分界点上的系统全局状态。为了证明这一点，我们需要展示如下两点成立：

(1) S*中每个进程 p 的状态与所有“prerecording”事件发生之后的进程 p 状态一致

(2) S*中每个 channel c 的状态就是所有“prerecording”事件沿着 c 发出的消息序列-所有“prerecording”事件沿着 c 收到的消息序列

第(1)点很明显因为两种情况下，进程 p 上发生的所有事件及发生顺序是一样的。对于第(2)点，假设 channel c 是从进程 p 到进程 q，那么在 S*中记录下的 channel c 的状态就是由进程 q 记录下自己的状态之后，到收到一个沿着 c 的 maker 之前收到的消息序列组成。而进程 p 在发送 marker 之前沿着 c 发送的消息序列就是所有“prerecording”事件沿着 c 发出的消息，因此(2)得证**{!根据前面的算法描述，channel c 记录的状态，实际上就是进程 p 记录状态之前沿 channe c 发出的但是进程 q 记录状态之前还没收到的消息序列，也就是说是进程 p 记录状态之前沿 channe c 发出的消息序列-进程 q 记录状态之前已经收到的消息序列。而进程 p 记录状态之前沿 channe c 发出的消息序列就是所有“prerecording”事件沿着 c 发出的消息，进程 q 记录状态之前已经收到的消息序列所有“prerecording”事件沿着 c 收到的消息序列}**。

为了更直观的理解这个过程，我们重新回到本节开头即上面图 7 的例子中。在这个例子中有如下三个事件：

e0：p 发送消息 M，状态变成 B (这是一个 postrecording 事件)

e1：q 发送消息 M’，状态变成 D (这是一个 prerecording 事件)

e2：p 收到了消息 M’，然后状态变成了 A (这是一个 postrecording 事件)

可以看到 e0 和 e1 就属于相邻的 postrecording 在前 prerecording 在后的两个事件，我们将它们进行交换得到如下新的事件序列：

e0’：q 发送消息 M’，状态变成 D (这是一个 prerecording 事件)

e1’：p 发送消息 M，状态变成 B (这是一个 postrecording 事件)

e2’：p 收到了消息 M’，然后状态变成了 A (这是一个 postrecording 事件)

现在的 seq’就是所有的 prerecording 事件在前 postrecording 事件在后的情况了。而我们记录下的那个全局状态，就是在 e0’之后的系统状态。

{!从这个例子里面，我们可以看到 e0 和 e1 两个事件，它们就是没有 happen before 关系的：首先它们是不同进程上的事件，其次这两个事件发生的间隔之间没有一

个消息从 p 到 q。它们交换后也是一个 computation。而反观 e0 和 e2，e1 和 e2 一个属于同一个进程上的事件，一个属于存在消息收发关系的事件，是不能像 e0 和 e1 那样进行替换的。

更直观的理解下就是 e0 和 e1 这两个事件本身是并行的，无 happen before 关系的，只不过观察到的好像是 e0 先发生了，但是我们可以想象一下就算是 e1 先发生，系统的后续状态也是一样的，也是一个合法的执行过程。而且因为两个事件本身就是并行的，到底是谁先发生的，实际上对于分布式系统实际上也很难测量。从 logical clock 的角度来看，对于没有 happen before 关系的事件来说，如果我们变换下进程 id 大小，它们在最终的全局事件里面的顺序本身就是可能不一样的。更极端的想象一下，如果系统中所有的节点都是独立的，相互之间没有消息通道，所有的事件都是节点内部的事件，不同节点间的事件都没有 happen before 关系，那么只要把每个节点自己的状态记录下来，就是一个一致的全局状态}

5.Stability 检测

现在我们看下怎么解决第 1 节里描述的 stability 检测问题。我们之所以研究 stability 检测问题，是因为它为很多实际问题(比如死锁检测)提供了通用的解决范式。关于 stability 检测算法的定义如下：

Input：一个 stable property y

Output：具有如下属性的一个 bool 值 *definite*：

(y(S ι)->definite) 并且 (definite->y(S Φ))

其中，S ι 和 S Φ分别代表算法开始和结束时的系统全局状态。“->”则代表逻辑蕴含关系{!A->B，意味着如果 A 为 true，则 B 也一定为 true。但是如果 A 为 false，B 可以为 true 也可以为 false}。

根据“->”，如果 definite 为 true，可以确定 y(S Φ)也为 true，意味着 stable property 在算法结束时成立。但是 definite 为 false，意味着 y(S ι)肯定不为 true，也就是说在算法开始时，stable property 肯定不成立。可以看出，definite 为 true 可以告诉我们算法结束时的系统状态，而 definite 为 false 可以告诉我们的是算法开始时的系统状态信息。需要特别强调的是，definite 为 false 并不意味着在算法结束时 stable property 也不成立。

stability 检测问题的解决方案如下：

```
begin
    record a global state S*;
    definite := y(S*)
end.
```

该算法的正确性，源自如下事实：

(1) S*从 S ι 可达

(2) S Φ从 S*可达，(这两条根据定理 1 可以得出)

(3) 如果 S'从 S 可达，y(S)->y(S') (根据 stable property 的定义可以得出)

{!综合上面这三条就可以得出 y(S ι)->y(S*)及 y(S*)-> y(S Φ)，又 definite=y(S*)，替换后即可得到 y(S ι)-> definite 及 definite-> y(S Φ)}