

[mp.weixin.qq.com](http://mp.weixin.qq.com)

# 这个夜晚，干货爆棚，Hacker真经——分布式系统一致性的发展历史(一) | 点融黑帮



**Daniel** Dianrong | Engineering

点融黑帮，用技术改变传统金融

## 导语：最爱的两件事情： 写代码和重构代码

分布式系统是一个非常有趣的话题, 在我之前的几年工作当中发现很多开发人员在评估和使用分布式系统的时候比较关注

benchmark和高可用性, 但经常不太关心分布式系统的一致性问題, 所以我打算在业余时间花点时间写这样一个系列文章来帮助开发人员更好的理解一致性问题.

## 1前言

在一个理想的世界里, 我们应该只有一个一致性模

型. 但是多路处理器和分布式系统中的一致性问题是 一个非常难以解决的问题. 当系统大多还都是单体应用, 处理器性能提升还是靠缩小晶体管体积的年代, 一致性问题还不是一个非常大的问题, 但是从90年代开始, 互联网公司大量出现, 互联网公司处理的数据量远远超过了

人类历史上任何一个时期的传统企业所需要处理的数据量, 大型的互联网公司甚至开始自己采购零部件自己组装服务器, 服务器市场上排名除了IBM, HP之外还出现了"Others". NoSQL已经被互联网公司广泛接受, Micro Service的出现让数据和计算的分布比历史上

任何时期都更加的水平，在这个背景下，分布式系统给一致性问题带来了巨大的挑战，理想情况下的一致性模型在分布式系统中几乎无法实现，所以我们不得不设计应用于不同场景的各种一致性模型。

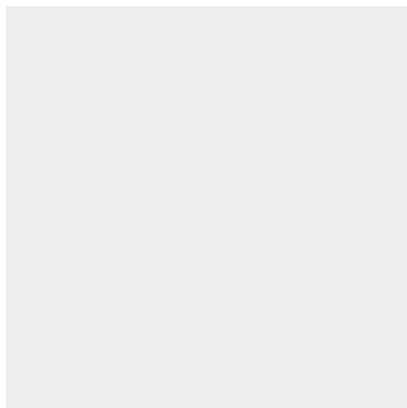
实际上在互联网蓬勃发展之前，有些有远见的科学家们从70年代就开始在多

路处理器级别上研究一致性模型. 过去的10年间, 计算机处理器的速度提升越来越困难, 摩尔在1965年告诉我们12个月晶体管密度提升一倍, 计算性能提升一倍(1975修正为24个月), 但是2002年1月 Moore在Intel开发者论坛上提出Moore's Law预计会在2017年终结, 一旦晶

体管体积进入皮米级别，晶体管将不得不接近原子大小，我们很快将无法突破物理的基础限制. 事实上从2012年开始晶体管密度和主频速度的提高速度明显放慢. 人们不得不开始转向横向伸缩，提高处理器的并行处理能力. 下图显示了在2012年出现了晶体管体积缩小速度



的突然放缓.



图片来源:

*futuretimeline.net*

而在这个过程中, 一致性问题逐渐愈发重要. 本文的目的就是带你回顾从70年代起, 到如今百花齐放的分布式系统中一致性问题的发展和演化. 你可能已经了解了某些一致性模型, 本文会帮你把它们按照发展历程穿连起来, 如果你对一致性模型了解很少, 本文可以帮助你从一

千英尺的高空俯瞰一致性  
问题, 通过非数学的语言,  
结合工程中的例子, 帮你  
入门. 这个系列的文章可  
能需要你放慢阅读速度,  
如果有些地方看不明白或  
者想更深入的探讨, 你可  
以去查阅所引用的论文.  
由于过程仓促, 难免有疏  
漏和错误, 还夹杂着个人  
观点, 如有错误欢迎指出:

daniel.y.woo@gmail.com.

本篇文章作为本系列的第一篇文章将会介绍1978年提出的Lamport Clock对分布式系统的启发，1979年提出的Sequential Consistency和1987年提出的最重要的一致性模型Linearizability.

## 2开天辟地：时间，事

## 件和顺序, 1978

狭义上的分布式系统是指通过网络连接的计算机系统, 每个计算节点承担独立的计算和存储, 节点之间通过网络协同工作, 因此整个系统中的事件可以同时发生. 一致性不是简单的让两个节点最终对一个值的结果一致, 很多时候还需要对这个值的变化

历史在不同节点上的观点也要一致. 比如一个变量x 如果在一个进程上看到的 状态是从1变成2再变成4, 最后变成3, 而另外一个进程看到的状态变化是1, 4, 3, 那么根据应用的场景, 有可能没问题, 也有可能会有严重的问题. 各个节点观察到的顺序不同是因为计算机网络绝大多数都

是异步的, 尽管单个TCP连接内可以保证消息有序, 但是一旦网络出现问题, 你会不得不重建TCP连接, 而多个TCP连接之间的消息没有顺序约定, 异步网络的延时和顺序还是没有保证的, 你不能简单的以接受到消息的时间作为事件的顺序判断的依据. 在多路处理器上也有

同样类似的一致性问题，  
因为多个处理器或者处理  
器核心之间也需要面对不  
可预料的事件顺序问题。  
所以，一致性并不是一个  
简单的问题。

举个现实的例子，比如你  
有两个数据中心，你通过  
DNS解析把用户分别引导  
到离用户最近的数据中  
心，然后数据中心之间做



双向数据同步. 但是如果用户使用你的系统的时候前后两个写入正好发生在切换网络的时候(或者更换了代理服务器, 或者换了手机操作, DNS解析调整, 等等), 那么用户可能会把两个请求写入了两个数据中心. 因为两个数据中心没有办法保证时间精确同步, 网络的延迟也很

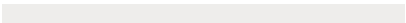
大, 这两个写入操作的时间顺序就非常重要, 想要合并和同步这两个写入操作并不是那么简单的. 如果这两个操作之间是有因果关系的, 比如第二个请求是基于第一个请求的返回结果的, 那么这两个请求之间的逻辑顺序就非常重要.

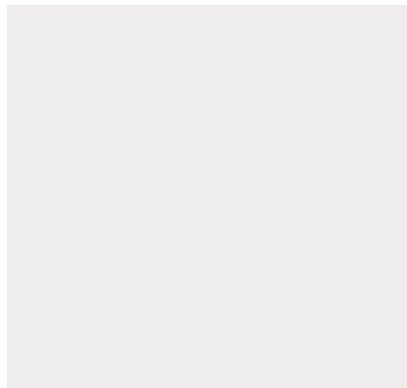
要理解一致性模型, 先要

理解分布式系统中的时间, 事件和顺序, 我们先来介绍事件发生的物理时间 (physical time), 逻辑时间 (logical time).

事件不是一个瞬间的事情, 它有一个起始和结束的时刻, 由于这个特性, 事件之间会有部分重叠. 比如下图中, A和B两个进程或者节点分别对一个队列

enqueue了x和y, 然后再分别dequeue. 你可以看到enqueue操作虽然是A先开始, B后发生, 但是在时间上他们有重叠, 这就是并行, 二者之间的物理时间上顺序可以说的不分先后的, 但是两个dequeue的物理时间上一定是发生在两个enqueue之前的.





物理时间虽然很重要, 但是在分布式系统中物理时间很难同步, 有时候我们更关心程序的逻辑顺序.

从物理角度看当然事件A和B总是有先后顺序或者同时发生, 但是在分布式系统中你非常难判断两个事件发生的先后顺序, 无论是NTP还是原子钟, 都存在误差, 你永远无法同步两台计算机的时间, 不考虑重力和加速度对时间的影响, 在同一个物理时刻 $t$ 发生了一个事件 $x$ , A节

点在他的时间参考系内可能会认为x发生在  $t - 1\text{ns}$ , 而B节点可能认为在它自己的时间参照系里x发生在  $t + 1\text{ns}$  的时刻. 这  $2\text{ns}$  就是A和B不同步造成误差. 假设我们把模型缩小到单个计算机, 想象一下两颗CPU在主板上的距离如果有  $3\text{CM}$ , 假设信号可以通过接近光速传递, 那么两

颗处理器之间这3CM就要传输任何信息至少有1ns的延迟, 而两个处理器分别会有自己的时间参考系, 二者之间必然有差别, 从更宏观的角度来看, 在一个分布式系统中一个节点看到另外一个节点的计算结果, 就像我们通过天文望远镜看其他星系一样, 那已经是几亿年之前



的历史了, 我们永远无从得知现在那个星系是什么样子, 甚至是否还存在. 所以依赖于物理时间我们是很难找到事件的顺序的. 两个事件的先后关系在分布式系统或者多路处理器系统中只能退化为先后顺序的偏序关系(*partial order*), 有时候只要能找出两个事件的因果关系就

行, 并不需要物理时钟, 有时候甚至因果关系 (causal relation) 都不重要, 本系文章后面要介绍的PRAM和Weak Consistency就不需要因果关系.

Leslie Lamport是分布式系统一致性研究的早期科学家, 图灵奖获得者, 他在1978年的论文中提出了

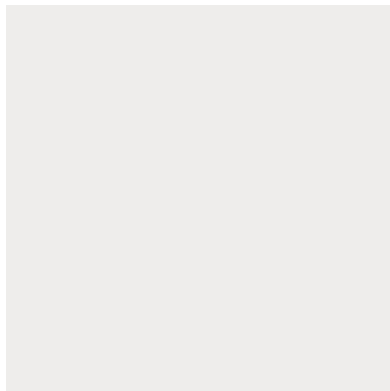
一个分辨分布式系统中的事件因果关系的算法, 后来大家把它叫做Lamport Timestamp或者Lamport Clock. Lamport Clock是一种表达逻辑时间的逻辑时钟(logical clock), 它让我们能够把所有的历史事件找到偏序关系, 而且不仅在各自节点的逻辑时间参考系内顺序一致, 全局

上的顺序也是一致的. 有人会问, 我用一个全局的锁服务来协同处理不就好了么, 干嘛搞一个这么复杂的東西, 其实不然, 全局协同服务有单点故障的问题, 另外全局锁是通过互斥(mutal exclusion)让整个系统串行化, 但是这样子整个系统就失去了并发能力, 而且某个进程的失

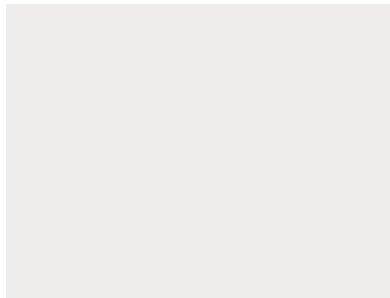
效可能会导致整个系统进入等待状态. 那么有人问了, 如果允许并发不用互斥, 但是让一个全局的 scheduler 作为法官来根据他自己对整个系统的观察来决断事件的历史顺序可以么? 答案是, 也不行. 举个例子, 下图中P0假设是全局的scheduler, 他作为法官一样来解读系统中

事件的顺序. P1如果发了一个事件消息A给P0, 然后再发一个事件消息B给P2, P2接收到B之后再发一个事件消息C给P0, 我们用 $X \rightarrow Y$ 来表示“X早于或者同时于Y”的偏序关系, 那么从P0的角度来看  $A \rightarrow C$ , 从P1看起来是  $A \rightarrow B$ , P2看到的是  $B \rightarrow C$ , 由偏序关系的传导性, 我

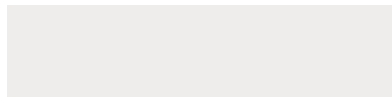
们知道事件的全局顺序应该  
就是A->B->C, 没有冲突  
矛盾, 很不错对吧.



但是消息的传递在分布式系统中要经过网络, 在多路处理器中要经过cache coherence协议, 如果A消息耗时比较久, 在C之后才到达P0呢?







这时候, 从P0的角度来看  
C→A, 从P1的角度来看  
A→B, 从P2的角度来看  
B→C, 看出来没, P1和P2  
的观点还好, 但是P0和P2  
的观点就互相矛盾了. 如  
果P0和P1是正确的, 因为  
C→A并且A→B, 根据传导  
性, 那么应该有C→A→B,

也即是C->B, 而P2看来  
B->C, 他们的观点互相矛盾了, 结点彼此看到的全局顺序的不一致了. 有些应用场景可以接受这种情况, 但是有些应用场景是不能接受这种情况的.

由于网络是异步的, 系统之间的物理时钟是不能精确同步的, 所以这里的P0作为全局的scheduler永

远无法知道基于物理时间的全局顺序是怎样的. 用一个scheduler就可以解决事件的全序集问题了么? 显然, 不行.

扩展一下这个例子到在实际应用, 你把P0想象成为一个NoSQL数据库的后端存储节点, P1和P2作为客户端, 如果P1写入一个数据A到P0, 然后P1又发

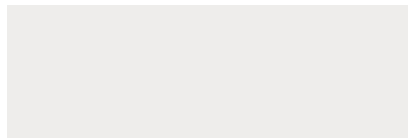
消息B通知P2做一个耗时的计算, 让P2去把A消息经过一系列计算来验证A是否被篡改或者损坏过. 如果P2的验证计算结果表明A是正确的, 那么P2什么也不用做, 否则要去把A修复的结果作为C消息存储到P0, 那么由于网络的延迟, 如果P0先收到了C消息, 后来才收到A消

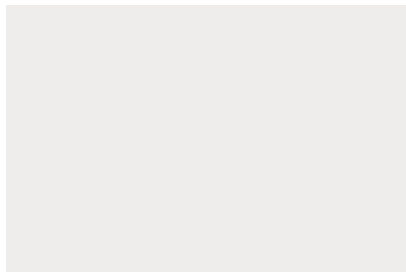
息, P0按照时间顺序操作  
那么就会把错误的A替换  
掉正确的C. 聪明的读者  
可能会想到Riak的Vector  
Clock, 但是在那之前, 我  
们先来看看Lamport  
Clock.

Lamport在他的论文中举  
了一个例子, 下面的图中  
P/Q/R是三个进程(这里的  
进程可以是多路处理器系

统其中的一个处理器, 也可以是分布式系统中的一个节点), 从下往上代表物理时间的流逝,  $p_1, p_2, q_1, q_2, r_1, r_2 \dots$  表示事件, 严格讲这些事件因为有起始和结束过程, 应该是沿着时间线的一段加粗线, 这里为了简化, 缩小到了一个点. 波浪线表示事件的发送, 比如  $p_1 \rightarrow q_2$  表

示 P把p1事件发送给了Q,  
Q接受此消息作为q2事件. 假设C是Lamport  
Clock, 并且是一个单调递增的计数器函数, 那么C  
应该满足任何两个事件a,  
b, 如果存在偏序关系(a先于或同时于b发生)  $a \rightarrow b$ ,  
必然有  $C(a) < C(b)$ .





图片来源: *Time, Clocks,  
and the Ordering of  
Events in a Distributed  
System*

为了满足这个Clock我们



需要两个条件:

C1. 对于单个进程 $P_i$ , 如果 $a$ 发生早于 $b$ , 那么 $C_i(a)$

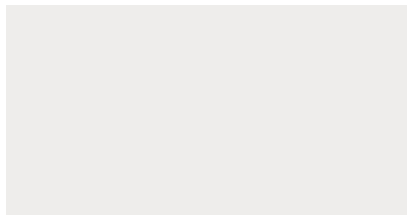
$< C_i(b)$

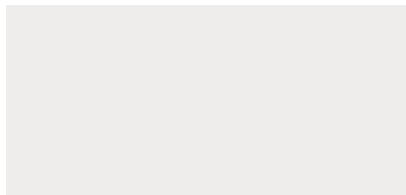
C2. 对于跨进程的情况, 如果 $P_i$ 发送 $a$ 到 $P_j$ 作为 $b$ , 那么 $C_i(a) < C_i(b)$

要满足第一个条件C1, 很简单, 只要 $P_i$ 每次两个连续事件发生中都要递增 $C_i$

即可. 要满足第二个条件, 稍微麻烦点, 就是要 $P_i$ 把 $a$ 附带自己的 $C_i$ 传递给 $P_j$ , 然后 $P_j$ 要递增到 $\max(C_i, C_j) + 1$ .

这样刚才的图就可以变成下面的图:





图片来源: *Time, Clocks,  
and the Ordering of  
Events in a Distributed  
System*

横向虚线就变成了全局的  
clock tick, 而每个进程的  
clock都是一致的.

Lamport Clock的作用就是找到causally related关系, 之后1988年出现的Vector Clock是一个更加实用的算法. 关于Vector Clock, Riak的工程师有两篇非常好的文章大家可以去看看(Why Vector Clocks Are Easy 和 Why Vector Clocks Are Hard). 这里受制于篇幅此处不再

详细描述, 在后继介绍  
Casual Consistency的时  
候会给大家详细介绍. 对  
于1978年这篇论文有兴  
趣的同学可以去检索  
Time, Clock, and  
Ordering of Events in a  
Distributed System. 这篇  
论文被认为是分布式系统  
的一致性问题的开山之  
作, 其中  $\max(C_i, C_j) + 1$

这个单调递增的  
timestamp后来影响了很  
多的设计, 比如Vector  
Clock, 比如一些Conflict-  
free Replicated Data  
Types, Paxos和Raft等.  
这篇论文也是Leslie  
Lamport被引用最多的文  
章, 按照Lamport自己的  
说法, 这篇论文影响了后  
人对于分布式系统的一致

性问题的思考方式, 而后基于人们对一致性, 性能, 实现难度之间的平衡, 产生了很多不同的一致性模型.

### 3理想的一致性模型

对于一致性, 我们需要不同的模型, 不同的级别. 理想的情况下任何事件都准确无误的”瞬时”对其他进

程或者节点可见. 比如, 一个服务节点对某个状态的写入, 瞬时可以被另外一个服务器同步复制, 因为“瞬时”的要求, 即便是 Lamport Clock 本质上是个 Logical Clock, 而非 Physical Clock, 所以也达不到这个要求. 如果存在这样的模型, 我们不需要通过 Logical Clock 去找事



件之间的因果关系 (causality), 事件的结束时间就可以解决事件的全序关系了. 有人称作这种模型叫做strict consistency. 实际上, 这在现实世界上根本不存在这样理想的模型, 两个服务器无论是通过金属电路还是光学载体连接, 都会有传输延迟, 这是物理决定的, 物理时间

在分布式系统中永远不可能同步, 因此我们对于一致性的时间要求必须降低, 比如

Linearizability(strong consistency)或者

Sequential consistency.

实际中我们经常比较关心两个有因果关系的事件的顺序而对于可以并行无因果关系的事件我们不太关

心他们的顺序, 再考虑到  
并发能力和实现难度. 我  
们需要多种不同的一致性  
模型.

## **4Sequential Consistency, 1979**

Leslie Lamport在  
Lamport Clock之后第二

年1979年提出了

Sequential Consistency,

Leslie Lamport的定义如

下:

A multiprocessor is said to be sequentially consistent if the result of any execution is the same as if the operations of all the processors were

executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.[How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs by Leslie

| Lamport ,1979]

这看起来很晦涩, 我们用通俗但不严谨的语言来表述, 他蕴含了两个部分, 第一是事件历史在各个进程上看全局一致, 第二是单个进程的事件历史在全局历史上符合程序顺序 (program order).

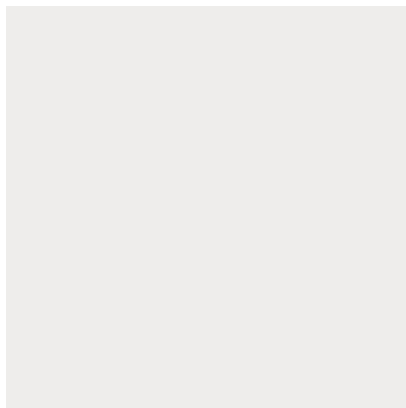
第一个要求比较容易理

解, 给个例子: 假设有四个进程P0, P1, P2, P3分别读取和写入一个变量x, 横轴从左往右表示物理时间的流逝. 对于下图中A的情况这就是大家比较容易理解的情况, 这就是 Sequential Consistency. B图中你可能觉得, 好像P2和P3读出来的x变量顺序和物理时间不一致了

么,但是对于P2和P3来说,他们虽然对x的历史的顺序和真实物理时间不一致,但是P2和P3至少”错的一致”啊,所以只要P0, P1, P2, P3全体都认为x是  
先被P2写入的2, 后被P0写入的1, 那么我们认为这种情况仍然是一致的. 这样的一致性模型就是Sequential

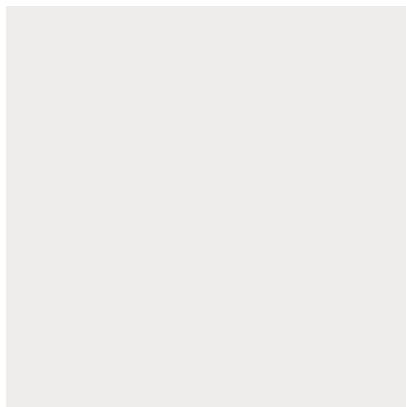


## Consistency.



如果像下面这样, 那么P3  
和P2互相就不一致了, 这

就不能满足sequential  
consistency了.



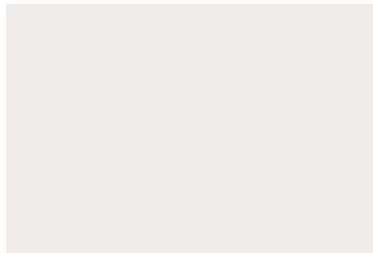
如果对于x的两次写入都

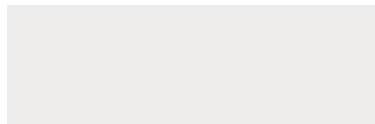
是发生在同一个进程, 比如P0, 那么前面B的情况也不符合Sequential Consistency了. 为什么呢? 因为这样此P0就不符合program order了. 我们来看Lamport的定义中第二个方面关于program order的解释.

第二个方面对于没有多线程开发经验的工程师稍微

难理解一些. 比如两个进程P1, P2上发生的事件分别是1,2, 3, 4和5, 6, 7, 8, 如下图所示. 那么从全局来看我们必须要让事件交错成一个有序的集合. 从下图右边global-1的观点来看这样的交错和P1/P2是符合Sequential Consistency的, 但是global-2就不是, 其中

1,4,3,2的顺序并不是P1的"program order". 第一种情况中P1和P2的原始顺序在交错中仍然得到了保留, 这个过程叫做arbitrary order-preserving interleaving.

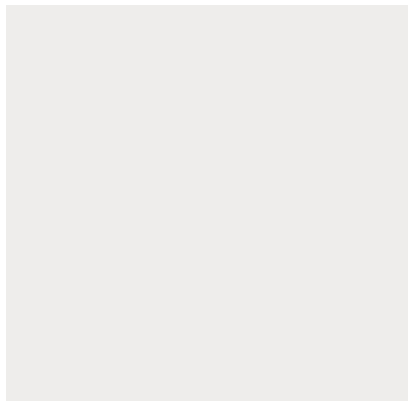




不熟悉多线程编程的工程师可能会问, 为什么P1和global-2中对于3和2的顺序有不同的观点? 为什么program order还会变? 我这里稍微解释一下CPU读写内存的工作原理, 熟悉C++/Java内存模型的程序员可以跳过这部分.

下面是典型的志强两路处理器的样子. 每个处理器的每个core有自己的L1/L2 cache, 所有的core共享L3 cache, 然后两颗处理器之间通过环形QPI通道实现cache coherence. 这样, 整个cache系统就成为了所有处理器核心看内存的窗口, 或者说是唯一事实.

---



处理器的一个cycle很快,  
1ns都不到, 而内存访问  
很慢, 需要几百个cycle了,  
就算是最快的L1的访问也



需要三个cycle, 只有寄存器能跟得上CPU cycle, 所以为了充分利用处理器, 在core和L1之间插入了接近寄存器速度的Load Buffer和Store Buffer.

LB和SB有多重方式可以提高处理器整体性能. 比如你对一个线程间的共享变量做密集的循环, 这个

变量的i++可能是发生在寄存器内或者store buffer内, 当循环结束的时候才写入L1 cache, 然后通过QPI让另外的处理器看到变化, 在这个密集循环过程中另外一个处理是看不到这个变量的变化历史的. 还有就是如果你对三个变量写入, 那么三次内存访问需要大概900个

cycle, 如果这三个变量地址连续, 那么很有可能他们碰巧在同一个cache line里, 那么处理器可能会把三个变量先写入store buffer, 稍后合并成一次写操作回到L1 cache, 这就只需要300个cycle, 这种优化叫做write combine. Store Buffer提升了处理器利用率但是会导致一个

CPU的内存写入对另外一个CPU的读取显现出延迟或者不可见. Java里面volatile就是放了一个full memory fence等待write buffer flush到缓存系统处理结束. (很多Java程序员认为volatile是让CPU直接访问主存来避免visibility问题, 这是错误的观点).

从另外一个角度看, 读取也有stale的问题. 因为解析地址和读取内存也是非常慢, 总共要几百个cycle, 所以处理器会使用 speculative read, 说白了就是打乱指令顺序提前发fetch到内存, 然后干别的事情去了, 这样读出来的内容放在load buffer里, 处理器稍后真用这个变量

的时候再去load buffer读取. 但是这样你会读到过期的数据. 要读到最新的数据, 也需要一个memory fence让之前load buffer内的数据被处理完.

所以, 处理器之间的可见性问题和乱序问题, 会让每个处理器对历史事件产生不同的观点. write

combine和speculative read再加上其他的优化会让处理器执行产生reordering. (memory fence会打破处理器的pipeline产生stall, 目前的主流服务器处理器有48个pipeline stages, 产生一次stall的代价非常高. 这就是为什么我们需要共享控制变量或者变量之间存

在因果关系的时候我们才需要memory fence, 这也是为什么我们需要将来会介绍的Causal Consistency和Weak Consistency模型.)

好了, 你现在明白了为什么program order会被reorder, 现在回过来看看Lamport在他的论文中为了阐述Sequential



Consistency而提出的这个有趣的小例子, 从一个更高层的角度, 程序员的角度来看第二个条件.

```
process 1
a = 1
if b = 0 then //
critical section
    a = 0
```

```
process 2
b = 1
if a = 0 then //
critical section
    b = 0
```

这看起来像是Dekker的  
Mutual Exclusion算法去  
掉turn变量的简化版, a和  
b初始为0, 分别是两个进  
程的控制变量来阻止对方

进入critical section, 如果系统满足sequential consistency, 那么process 1和2是不能同时进入critical section的. 这两个进程可能都无法进入, 但是最多有一个进程可以进入. 但是如果系统不满足sequential consistency的第二个条件, 那么有可能两个进程

同时进入这段critical section. 你可能会问, 怎么可能? 如果你看明白了上面关于处理器访问内存的原理, 你会知道有可能进程1写入a=1被write combine延迟了, 或者判断b=0的那个读取被speculative read了, 那么两个进程就会同时进入critical section.

Leslie Lamport为此提出了如何实现sequential consistency的内存访问:

- 1) Each processor issues memory requests in the order specified by the program.
- 2) Memory requests to any individual memory

module are serviced  
from a single FIFO  
queue.

第一点就是禁止  
reordering, 第二点就是  
FIFO的内存控制器. 这样  
的系统可想而知, 会比我  
们今天使用的系统慢几百  
倍, 没有任何一款主流处  
理器能够达到sequential

consistency. 所以为了换取效率, 维护数据一致性变成了开发人员的责任, 系统只提供给你memory fence, CAS, LL/SC之类的基础工具. 无论你是C++还是Java开发人员都要非常小心的使用线程, 其实即便是非常有经验的开发人员有时候也会在线程上犯错, 近年来出现的

lock-free和wait-free的数据结构比基于锁的实现运行期更加动态, 更加难以正确实现, 出现了bug之后非常难解决. 所以我们必须深入理解一致性, 在设计的时候想清楚各种边界条件来避免多线程的问题.

早期一致性问题都是在多路处理器级别研究的, 但



是往更宏观的分布式系统来看, 多个服务器节点的写操作也经常会有本地buffer, 也会产生类似write combine的副作用, 这些问题在不同规模上看都是类似的.

大家或许注意到了, Sequential Consistency 对于时间不敏感, 只要存在一致的全序关系即可,

所以又出现了对时间敏感, 一致性强于 Sequential Consistency 的Linearizability.

## **5Linearizability, 1987**

Linearizability模型的一致性高于 sequential consistency, 有时候也叫做strong consistency或

者atomic consistency, 可以说是我们能够实现的最高的一致性模型. 它是Herlihy and Wing在1987年提出的. 通过前面的讨论我们现在知道sequential consistency只关心所有进程或者节点的历史事件存在唯一的偏序关系, 它不关心时间顺序. 而linearizability相当

于在sequential consistency的基础上再去关心时间顺序, 在不依赖物理时钟的前提下, 区分非并发操作的先后. Linearizability是一个很重要的概念, 如果你不了解它你就无法真正理解Raft算法等.

我们可以用一个高等数据结构比如队列来描述

Linearizability的定义(来源于Herlihy & Wing). 假设Enq  $x$   $A$ 表示 $A$ 进程开始尝试把 $x$ 加入队列尾部, OK  $A$ 表示操作结束, Deq  $A$ 表示 $A$ 进程开始从队列头部取出一个元素, OK  $y$   $A$ 表示取出了 $y$ 并操作结束. 下面左边是物理时间上的操作事件顺序, 那么我们如果再最后面加一个

OK A那么, 你会发现所有的事件的起始/结束都是成对的.

Enq x A, OK A, Deq B,  
OK x B, Enq y A

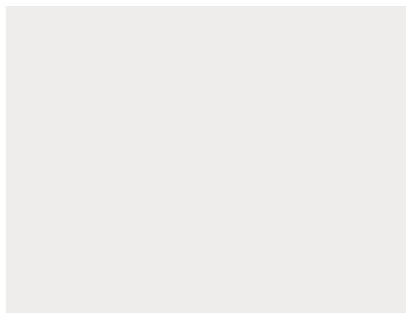
如果H能够通过追加0到1个事件成为H', 并且H'中的事件是起始/结束成对的(H'是H的完整历史), 那

么 $H'$ 可以线性化为全序关系 $S$ , 使得

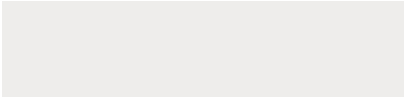
- 1).  $S$ 中所有的事件起始/结束是“紧挨着”的, 事件具有原子性不可拆开.
- 2). 并且一个事件 $e_0$ 的结束早于 $e_1$ 的开始, 那么这个全序关系在 $S$ 中仍然存在.

那么在不违背队列的正确

行为的前提下, S就是H的一个linearization. 下面的例子中, 最左边是物理时间上发生的H, 中间是补齐成对之后的H', 右边是H'的一个linearization.

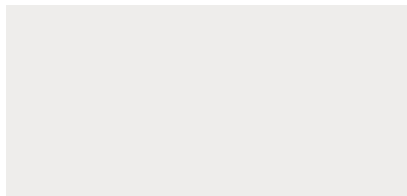


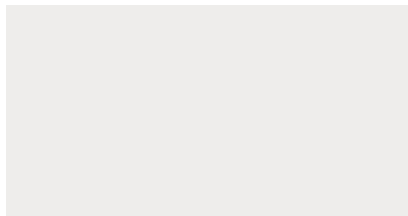




用通俗但是不严谨的话来说, Linearization蕴含了两个特性, 第一, S中的事件起始/结束是”紧挨着”的表示粒度上必须是整个事件, 事件之间不能交错, (就是假设一个调用的结束事件返回之前我们就认为事件已经完成了, 把并

发的调用时间缩短来产生顺序关系). 第二,  $H$ 中全序关系延续到 $S$ 中, 说明 $S$ 仍然存在原始事件的物理时间的顺序. 如果你没看明白这么拗口的定义, 没关系, 看看下面这个图你就很容易明白了.





并不是所有的情况都是  
Linerizable的, 比如我们  
的队列行为是这样的:

1. Enq x A

2. OK A

3. Enq y B

4. Deq A

5. OK B

6. OK y A

因为OK A早于Enq y B,  
那么接下来的事件历史中  
x就应该比y早出来, 而这  
个例子中y先出来了, 这违  
背了队列这种数据结构的  
定义, 除非程序写错了否  
则不应该有这样的历史.  
如果我们尝试满足那两个  
条件, 那么这个历史有两  
种排法:

Enq x A -> OK A -> Deq

A -> OK y A -> Enq y B  
-> OK B

Enq x A -> OK A -> Enq  
y B -> OK B -> Deq A ->  
OK y A

但是这两个排法都违背了  
队列的FIFO行为, 这样的  
事件顺序不符合  
Linearizable.

再给一个例子, 这个是可以的

1. Enq x A

2. Deq B

3. OK x B

首先补齐结尾的OK A变

成H', 然后可以把S排成:

Enq x A -> OK A -> Deq  
B -> OK x B

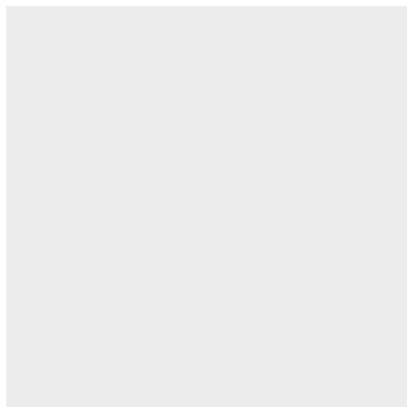
你要是觉得这种定义很晦涩, 没关系, 用通俗的语言来讲, 就是A写入了x结束后, 接下来B一定能读出来, 这虽然不严谨, 但是确实是Linearizability的本质. 下图中从左往右是物



理时间, 长条是指我们观察到的一个事件的开始和结束, 其中P0把x的值从0改写为1. 长条是一个事件的时间窗口. 这个过程中如果P1去读取x, 如果P1的读和P0的写在时间上有重叠, 二者是并发的, 那么P1读出来的无论是0还是1, 都算符合 linearizability, 你认为P1

的读事件发生在P0的写事件之前或者之后,都可以. P2和P0之间也是这样. 但是如果P2的个读也发生在P1上, 那么两个读的结果必须要么0-0, 要么0-1, 要么1-1, 但是不能是1-0. 最后的P3因为起始晚于P0的结束, 所以要符合linerizability就只能读出 $x=1$ 才行, 我们必须认

为P3的读事件发生在P0  
的写事件之后.



当事件之间没有重叠, 中间有个”小间隔”的时候, Linearizability必须给出明确的先后顺序. 而前面介绍的sequential consistency则不需要这样的要求, 如果上面的例子中如果P3读出来是0, 那么也是符合Sequential Consistency的.

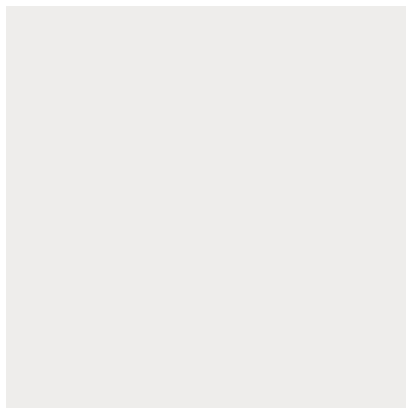
当多个事件之间有重叠,

他们实际是并行的，  
Linearizability要求不管是  
不是并行的事件，所有进  
程都必须给出一致的事件  
顺序，这样所有进程都会  
对事件历史有完全同样的  
线性观点，这就像是一个  
消除并行的过程，所叫我  
们称之为做Linearization。  
一个历史可以有多种  
Linearization，比如前面

的例子中P0和P1的事件是并行的, 可以互换顺序, 所以这个历史至少可以有两种linearization. 要所有进程都全局一致还需要其他的条件, 本系列文章后面要介绍的Paxos算法就是这样例子.

下面是Herlihy和Wing的论文中给出的例子, 大家可以看看, 哪些是符合

## Linearizability的?



图片来源: *Linearizability*  
: *A Correctness*

## *Condition for Concurrent Objects*

Linearizability的两个很好的特性Locality和Non-blocking, 有兴趣的同学可以自己去看看, 限于篇幅本文不再介绍了. 如果你的理解还有点不清晰, 下一篇文章中我们介绍Paxos算法的时候你应该能更好的理解它.

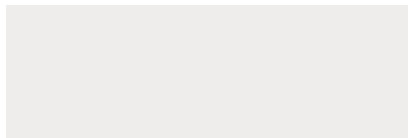


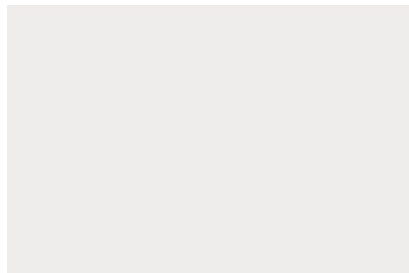
Linearizability和

Sequential Consistency

你可以把它们想象成一个烧烤架, 上面有很多烤串, 你不可以把肉和洋葱在一个烤叉上互换位置(单个进程需要符合program order), 但是你可以拨动所有烤叉上的肉和洋葱, 让他们从左往右排列出来一个先后顺序. 不管是

Linearizability还是  
Sequential Consistency,  
那么下面A和C谁在前面  
都可以, 因为A和C是并行  
的, 但是C和B在  
Linearizability中必须C在  
前B在后, 而Sequential  
Consistency中B和C的顺  
序可以互换.





这么看Linearizability和  
Sequential Consistency  
是不是很像？过去绝大多  
数人会认为Linearizability  
的定义只是比Sequential  
consistency多了一个物

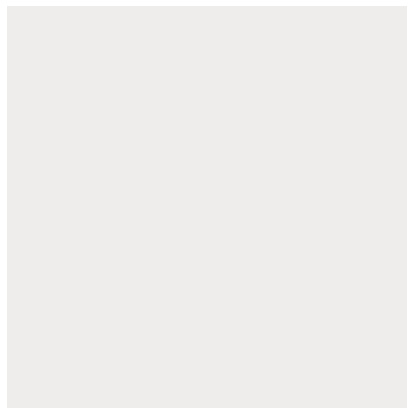
理时间先后的约束, 所以认为他们很像, 甚至认为它们基本是一个东西, 性能也应该差不多. 但是后来以色列科学家Hagit Attiya和Jennifer Welch在1994年发表的论文让大家意识到他们其实是完全不同的概念, 性能差别很大. (Sequential Consistency versus

Linearizability, ACM  
Transactions on  
Computer Systems, Vol  
12, No 2, May 1994,  
Pages 91-122) 过去人们  
在探讨这两种一致性模型  
的时候没有真正分析过他  
们的理论上的性能差别，  
Attiya第一次给出了具体  
的分析.

假设我们的网络延迟有一

个上限 $d$  (尽管现实中不可能, 这里纯粹做理论分析用), 那么最慢和最快的请求波动在 $u$ 以内. 论文证明了下图中上半部分绿色四边形是Linearizability的性能根据读写比例从 $u/4$ 到 $u/2$ , 下面黄色三角形是Sequential Consistency的性能, 最糟糕不会超过 $d * 2$ . 从性能角度来看, 二

者的差别还是巨大的.



至于Linearizability的实现, 可以说是所有一致性

模型里最丰富的了, 我们会在本系列下一篇文章中介绍.

如果你耐心看到这里了, 至此, 我们已经开了一个好头, 你一定是对分布式系统的一致性问题很感兴趣, 在本系列下一篇文章中我们将会提到 Linearizability 和 Sequential Consistency



的一些应用. 将来我们还会继续介绍Casual Consistency, PRAM, Eventual Consistency这些性能更好但是一致性更弱的模型, 所以, stay tuned!

---

本文作者：吴强（点融黑帮），现任点融网首席社交平台架构师，前盛大

架构师, 专注分布式系统和移动应用, 有十三年的开发经验, 目前在点融做最爱的两件事情: 写代码和重构代码.

---

**随着点融网新一轮融资，  
点融网即将开始大规模的  
扩张，需要各种优秀人才的  
加入，如果你觉得自己  
够优秀，欢迎加入我们！**

**获取更多职位信息，请关注点融黑帮。**