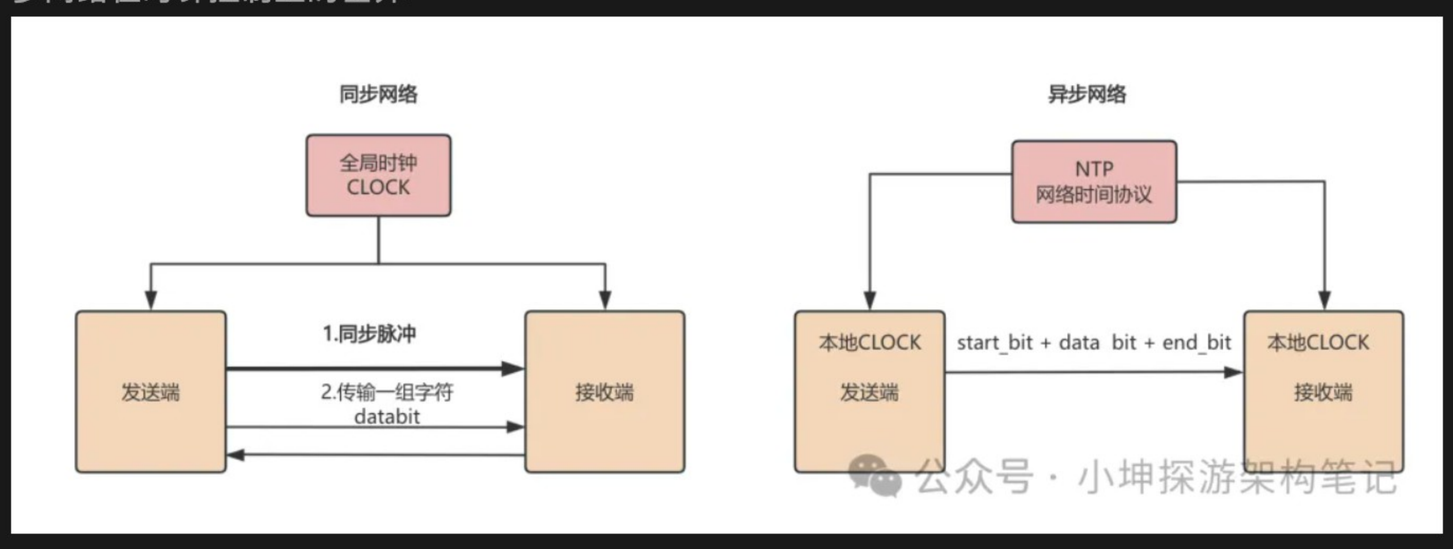
分布式系统不可靠时钟问题

原创 疾风先生 小坤探游架构笔记 2025年05月04日 20:03 广东

今天我同样结合《设计数据密集系统》来聊分布式系统另外一个话题,即分布式系统的不可靠时钟问题.

同步网络与异步网络时钟对比

在前面我们讲述了分布式系统网络不可靠的原因是由于我们数据中心服务通信采用的是异步网络实现, 同样地我们来看为什么同步与异步网络在时钟上也存在差异呢?对此我们可以先看下面同步与异步网络在时钟控制上的差异:



通过上述我们可以清晰地知道:

- 同步网络**存在全局时钟控制, 异步网络中每个节点是拥有自己的时钟独立运行**,并通过NTP网络协议来协调纠正时钟不同步问题,但存在网络延迟问题.

- 同步网络发送消息需要**先同步脉冲后再传递数据信息**, 而异步网络是通过**开始以及结束位标志**进行数据同步保证数据的完整性.

对此我们对同步网络以及异步网络的时钟精度做一个对比总结如下:

指标	同步网络（PTP/SyncE）	异步网络（NTP/UART）
时钟同步精度	纳秒级（1–100 ns） 150 165 PDF	毫秒级（1–50 ms） 51 128
漂移抑制能力	动态补偿，长期误差趋近零	累积误差不可控
传输延迟敏感性	通过路径补偿消除影响	依赖对称性假设，误差较大
硬件成本	高（专用时钟芯片）	低（通用振荡器）

单调时钟以及日时钟的不可靠性

通过时钟可以来反映我们系统想要表达的时间问题, 一般在我们应用程序会拆分两类层次的含义:

> 持续时间: 描述的是一个事件发生的开始到结束整个过程经历的时间

比如一个请求的p99耗时是多少, 这个就是我们需要依赖时钟去测量应用程序请求开始到结束的持续时间.

> 时间点: 指事件发生的时候对应的那一时刻

比如我们的应用程序发生NullPointerException异常的时候日志系统对应的时间戳,这个就是时间点.

在现代计算机系统中时钟又拆分为**单调时钟以及日时钟**.即:

> 日时钟: 它根据某种日历返回当前的日期和时间, 也称为我们墙上的时钟时间, 比如Java的System.currentTimeMillis(), 用于表示时间点. 日时钟通常与网络时间协议（NTP）同步，这意味着一台机器上的时间戳（理想情况下）与另一台机器上的时间戳含义相同.然而由于存在硬件时钟与NTP协议的不稳定,不适合用于测量持续时间.

通过以上描述, 我们可以依赖日时钟来描述我们事件发生的时间点.但是不适合测量持续时间,为什么? 我们先来看下单调时钟:

> 单调时钟: 主要用于测量持续时间, 比如超时时间或者服务响应时间, 比如Java的System.nanoTime()就是单调时钟.需要注意一点就是单调时钟仅在单机单进程下计算对应的差异才有意义, 否则没有实际比较性, 因为该时钟可能是计算机启动以来的纳秒数，或者是类似的任意数值开始统计. 由于单调时钟不需要与NTP同步, 但如果NTP发现本地石英晶体振荡器比NTP走得快或者慢, 就会调整单调时钟向前走的频率, 即时钟微调.

我们可以看到**单调时钟始终是被保证向前移动**,因此在分布式系统中单进程内计算持续时间可以采用单调时钟而非日时钟.这是因为日时钟存在同步以及准确性问题如下:

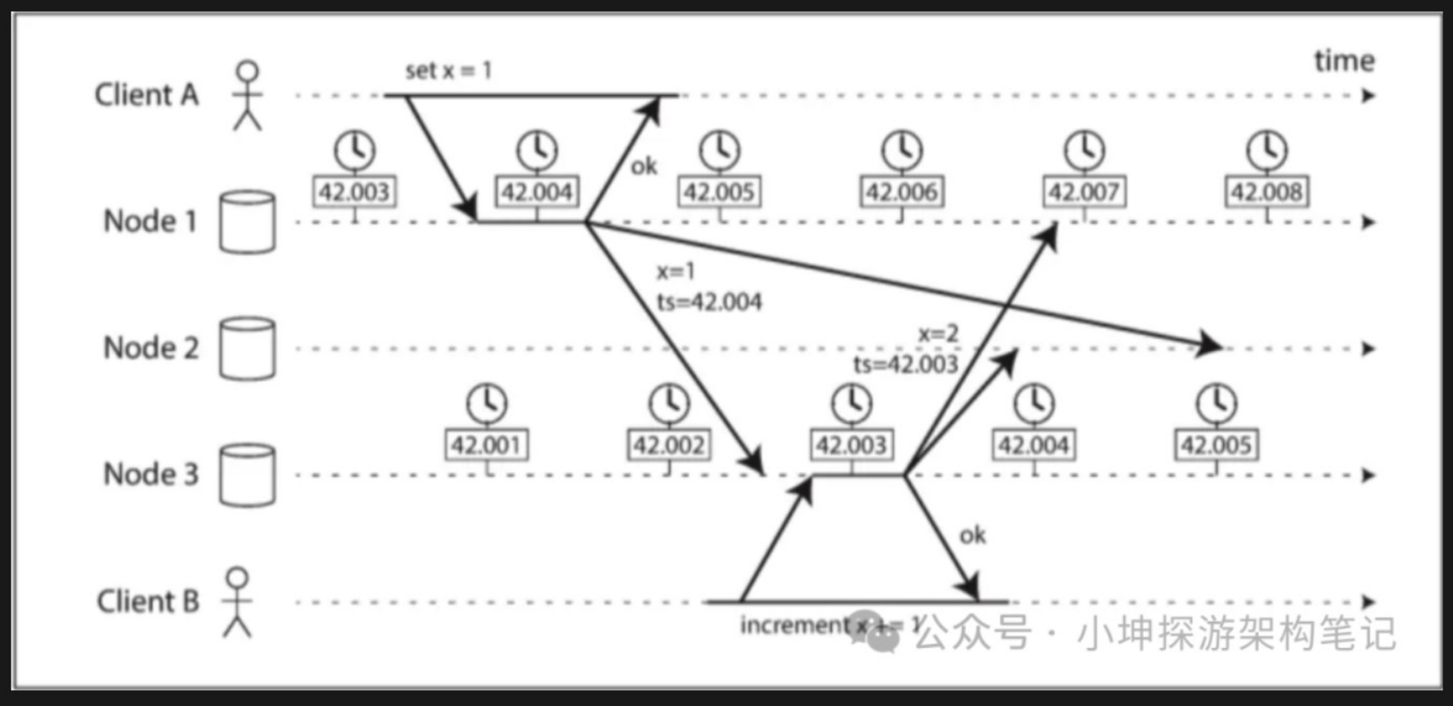
- 计算机的石英钟存在漂移导致时间不精准,可能存在向后跳跃;
- 本地时钟与NTP服务的时间差较大拒绝同步会被重置;
- NTP服务配置错误导致;
- NTP同步到机器节点存在网络无界延迟问题.

因此在分布式系统中我们设计应用程序也需要充分考虑到日时钟存在不可靠的问题.

分布式系统依赖时钟带来的问题

> 依赖时间戳进行事件排序导致乱序

如果两个客户端向一个分布式数据库写入数据,那么谁先到达? 如果以**最后写入为准机制(LWW)**的话会发生什么问题呢? 如下所示(来自《设计数据密集系统》):



可以看到上述的时钟在不同节点的差异, 我们可以看下上述的逻辑处理过程:

- 首先ClientA在42.004秒时刻发起一个 `set x = 1` 的写请求到Node1节点, 这个时候Node1分别向副本节点Node2、Node3也分别发起命令复制的写操作请求;
- 其次ClientB在42.003秒时刻也发起一个 `set x = 2`的写请求到Node3节点, 同样地Node3节点也分别向Node1、Node2节点发起命令复制的写请求;
- 我们可以看到Node2节点接收到命令`set x = 1`对应的时间戳是42.004秒, 而命令`set x = 2`对应的时间戳是42.003秒,如果我们采用LWW机制, 即直接采用最新的时间戳的方式进行作为解决并发写冲突的机制, 那么Node2节点就会丢弃掉`set x = 2`的命令,这个时候在数据库节点Node2上就会神秘发现数据丢失了.

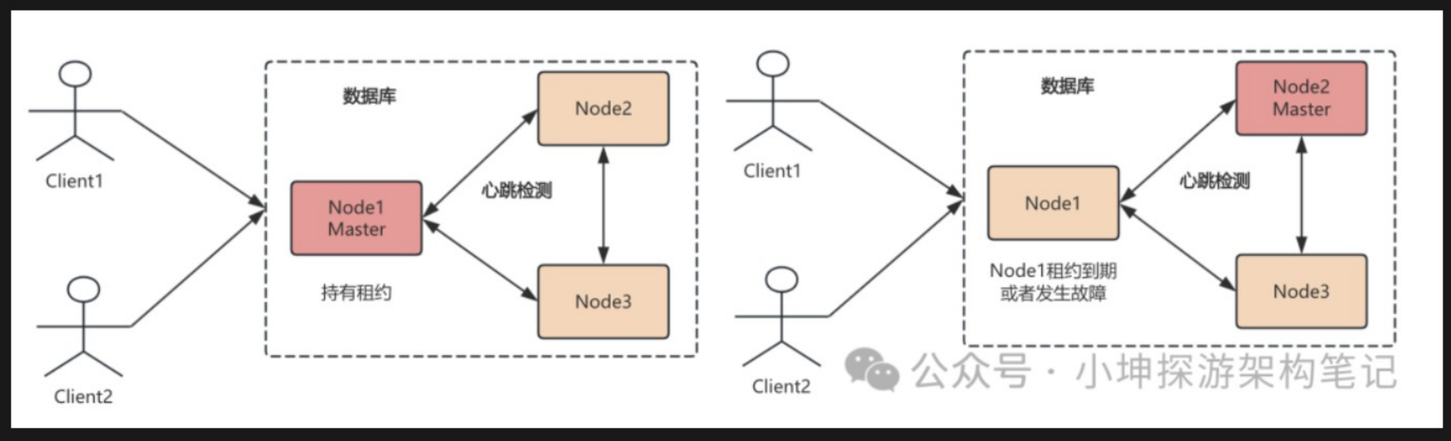
在上述的例子中由于Node3节点与Node1节点存在时钟相差的间隔,导致Node1先写入的时间戳要大于Node3节点写入的时间戳, 从而导致Node2节点接收到其他节点的复制命令时候, 由于并发写冲突的存在且采用LWW机制最终导致`set x = 2`的命令被丢弃导致数据丢失.

那么有什么解决方案吗? 这里我们会用到一个概念称为**逻辑时钟, 即基于全局递增计数器而非振荡的石英晶体, 同时在分布式数据库环境中, 要保证是全局性递增,逻辑时钟不测量具体日期时间或者秒数, 仅测量事件的相对顺序, 即事件发生在另一事件之前还是之后, 这样对于LWW机制解决并发写冲突是一种更为安全的选择**.

而上述的单调时钟或者是日时钟我们称为物理时钟.但是逻辑时钟依赖因果关系实现并需要具备持久化等机制防止丢失,在实现上更为复杂.

> STW引发数据写入安全问题

假如现在有一个数据库,并且每个数据库分区仅有一个主节点能够接受写入操作,同时数据库的主节点与其他副本节点通过租约实现共识,也就是Master节点持有一份从其他节点获取带有过期时间的租约, 并在在租约的有效期内是能够处理外部的写入请求并将命令复制到其他节点实现同步直到租约过期, 如下:



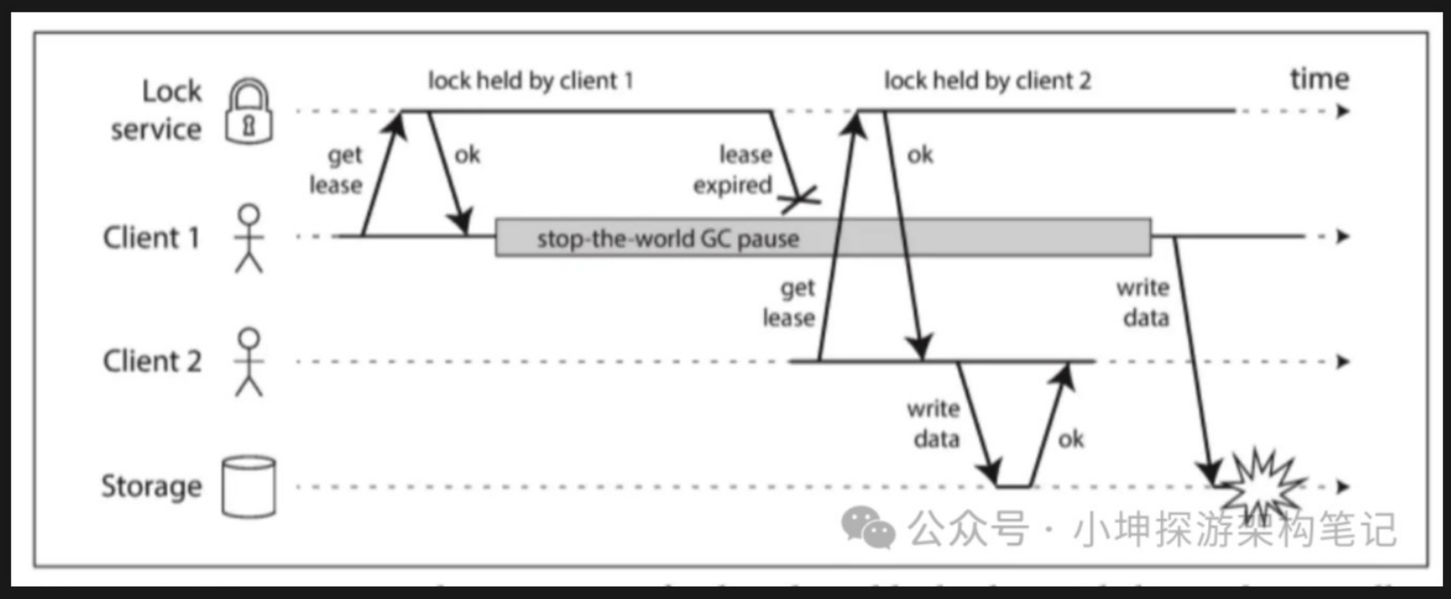
每个Node节点对应的伪代码如下:

```
while (true) {
    request = getIncomingRequest();

    // Ensure that the lease always has at least 10 seconds remaining
    if (lease.expiryTimeMillis - System.currentTimeMillis() < 10000) {
        lease = lease.renew();
    }

    if (lease.isValid()) {
        process(request);
    }
}
```


上述代码存在什么问题? 主要有两个方面:

- 判断过期时间依赖外部节点的时间戳, 前文我们讲到日时钟存在不可靠,如果时钟不同步超过几秒,即本没有到期但却判断到期导致重新发起续约请求重新选举master节点, 在选举过程中对外服务不可用直接丢弃写请求; 如果时钟后退几秒钟, 那么也将导致其他节点重新发起master选举, 如果是在跨region网络分区的数据分布下, 那么将会出现两份同时具备租约的数据节点, 即脑裂问题.

- 如果我们将上述更改为本地协议为单调时钟呢? 其实也存在问题, 比如进程内部发生GC/代码执行到某个SafePoint/进行刷盘fsync等操作导致进程暂停, 这个时候由于其他副本节点就会存在误判master节点不可用, 导致重新选举然后接受client的写操作, 这个时候就会发生数据不一致甚至被损坏的问题.



因此在分布式系统中我们必须要假定其中一个节点在执行过程中任何时刻都可能存在被暂停一段时间, 甚至是在函数执行的中间也不例外, 只有这样的假定基础上去设计我们构建的分布式系统才能确保我们数据的可靠性以及完整性.

全局快照的同步时钟

在上述我们提到使用全局单调递增的计算器来保证我们执行事务前后的顺序性,也就是说我们的数据库是分布在多台机器上甚至是多数据中心分布, 那么我们要实现一个全局的、单调递增的计数器就会变得很困难, 因为多数据中心存在跨区问题, 需要进行协调. 一般地我们要么会增加同步互斥锁机制, 类似数据库的悲观锁仅允许一个操作后再进行下一个操作, 但是这样会严重影响性能.

为了提升性能并兼顾对外部提供读取操作, 在我们数据库层面中会引入一个快照隔离机制, 对于那些支持小型、快速的读写事务又能支持大型、长时间运行的只读事务的数据库而言是一项非常有用的功能, 但是也存在同样的问题, 数据分布在多台机器怎么办呢? 快照隔离和我们处理事件的顺序性有共同之处: **事务B如果能够读取事务A的数据, 那么事务B的ID必须要高于事务A, 否则快照就是不一致的**.

目前在业界中, 谷歌云Spanner就是基于日时钟作为事务ID实现多数据中心的快照隔离, 为什么它能够利用日时钟实现呢? 它主要采用TrueTime API明确报告本地时钟的置信区间(**所谓的时钟置信区间, 我们可以理解为它是一个日时钟的时间段而不是一个时间点,即[最早时间,最晚时间]**), 并基于以下观察结果:

- 假设现在有两个提交事务ID对应的时间戳a以及b, 同样也会对应A以及B两个时钟的置信区间A和B;
- 如果A与B区间不存在重叠且A的最晚时间 < B的最早时间, 那么A一定是在B之前; 如果A的最早时间 > B的最晚时间, 那么B一定是在A之前;
- 当A与B发生重叠的时候, 为反映A与B发生前后的因果关系, **Spanner通过在提交读写事务的时候特意等待一段置信区间长度的时间**, 为了尽可能缩短等待时间, Spanner需要尽可能减小时钟的不确定性; 为此谷歌在每个数据中心都部署一个全球定位系统（GPS）接收器或原子钟, 使得时钟能够同步到误差在大约7ms以内. 通过这样操作, 它确保任何可能读取该数据的事务都处于足够晚的时间.

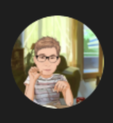
我们对比下逻辑时钟实现以及TrueTime的不同维度:

维度	TrueTime方案	逻辑时钟（如HLC）
全局有序性	物理时间严格单调，无因果冲突	依赖因果关系，需处理逻辑时钟回退
跨系统一致性	支持多独立系统间原子快照 14	仅限单一系统内部协调
故障恢复复杂度	无需维护逻辑时钟状态	需持久化时钟状态防丢失
网络分区容忍	快照读可访问历史版本 10	分区期间无法生成新时间戳

分布式系统问题小结

最后我对前面阐述的分布式系统问题做一个总结如下, 即我们在搭建一个分布式系统时需要建立对应的系统模型, 思考在面临组件/服务故障、网络延迟、时钟依赖以及节点暂停等情况下要如何进行设计与取舍来保证分布式系统的数据可靠性.

你好,我是疾风先生, 主要从事互联网搜广推行业, 技术栈为java/go/python, 记录并分享个人对技术的理解与思考, 欢迎关注我的公众号, 致力于做一个有深度,有广度,有故事的工程师,欢迎成长的路上有你陪伴,关注后回复greek可添加私人微信,欢迎技术互动和交流,谢谢!



小坤探游架构笔记

10年后端技术架构设计 | AI工程化基础建设 & 存储架构设计 & 性能优化 | 前网易、斗鱼、i...

52篇原创内容

公众号

分布式架构 · 目录

上一篇
Raft算法

下一篇
分布式系统不可靠的网络问题

个人观点，仅供参考

修改于2025年05月04日