



下载APP



15 | 分布式事务串讲：重难点回顾+思考题答疑+知识全景图

2020-09-11 王磊

分布式数据库30讲

[进入课程 >](#)**讲述：王磊**


时长 19:49 大小 18.16M



你好，我是王磊，你也可以叫我 Ivan。

今天这一讲是我们这门课的第二个答疑篇，我会带你回顾第 9 讲到第 14 讲的主要内容，集中解答留给你的思考题，同时回复一些留言区的热点问题。这几讲涉及的也是数据库的核心话题“事务”，咱们一定得学扎实了。

第 9 讲：原子性提交协议

在 [第 9 讲](#)，我们首先讨论了两种不同的原子性协议，分别是面向应用层的协议 TCC  向资源层的协议 2PC。

使用 TCC 协议时，由应用系统负责协议的实现，数据库没有额外工作。所以，TCC 更加灵活，但对业务的侵入性更高。反之，使用 2PC 协议时，主要靠数据库来实现，应用层工作很少，所以业务侵入少，但是存在同步阻塞、单点故障和数据不一致这三个问题。

针对 2PC 协议的这些问题，我又介绍了两种改进型协议。一种是 Percolator 模型，在 TiDB 和 CockroachDB 中有具体落地。另一种是 GoldenDB 中实现的“一阶段提交”协议。它们都较好地解决了单点故障和数据不一致的问题。而有关同步阻塞带来的高延迟问题，我们没有展开，而是留到了 [第 10 讲](#)。

这一讲的是思考题是：2PC 第一阶段“准备阶段”也被称为“投票阶段”，和 Paxos 协议处理阶段的命名相近，你觉得 2PC 和 Paxos 协议有没有关系，如果有又是什么关系呢？

“Chenchukun” “tt” “李鑫磊” “piboye” 和 “Bryant.C” 等几位同学的留言都很好地回答了这个问题。总的来说，就是 Paxos 是对单值或者一种状态达成共识的过程，而 2PC 是对多个不同数据项的变更或者多个状态，达成一致的过程。它们是有区别的，Paxos 不能替换 2PC，但它们也是有某种联系的。那到底是什么联系呢？

别着急，我们先看看 “Lost Horizon” 和 “平风造雨” 两位同学的提问。

“Lost Horizon” 在第 9 讲提出的问题是：如果向单元 A 发出 Confirm 操作成功且收到成功应答，向单元 B 发出 Confirm 操作成功，但没有收到成功应答，是否应该先确认 B 的状态，然后再决定是否需要回滚（或者补偿）A 的变更呢？

“平风造雨” 在 [第 10 讲](#) 提出的问题是：如果客户端没有在一一定的时间内得到所有意向写的反馈（不知道反馈是成功还是失败），要如何处理？

这两位同学的问题虽然不一样但有一个共同点，就是**事务协调者收不到事务参与者的反馈怎么办**。

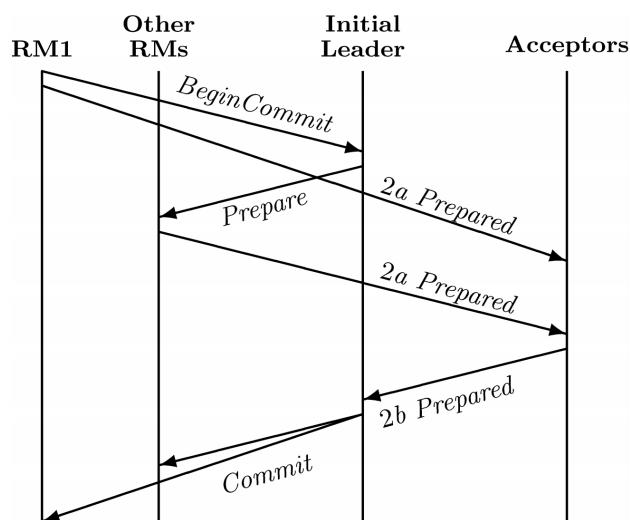
为什么会收不到呢？多数情况是和网络故障有关。那么，Paxos 协议这个针对网络分区而设计的共识算法，能不能帮助解决这个问题呢？

Paxos 确实有帮助，但真正能解决问题的是在其之上衍生的 Paxos Commit 协议，这是一个融合了 Paxos 算法的原子提交协议，也是我们前面所说的 2PC 和 Paxos 的联系所在。

Paxos Commit 协议

Paxos Commit 协议是 2006 年在论文 “[Consensus on Transaction Commit](#)” 中首次提出的。值得一提的是，这篇论文的两位作者，正是我们课程中多次提到的 Jim Gray 和 Leslie Lamport。他们分别是数据库和分布式领域的图灵奖获得者，也分别是 2PC 和 Paxos 的提出者。

我们结合论文中的配图，简单学习一下 Paxos Commit 的思路。



Paxos Commit 协议中有四个角色，有两个与 2PC 对应，分别是 TM (Transaction Manager, 事务管理者) 也就是事务协调者，RM (Resource Manager, 资源管理者) 也就是事务参与者；另外两个角色与 Paxos 对应，一个是 Leader，一个是 Acceptor。其中，TM 和 Leader 在逻辑上是不可能分的，所以在图中隐去了。因为 Leader 是选举出来的，所以第一个 Leader 标识为 Initial Leader。

下面，我们来描述下这个处理过程。

1. 首先由 RM1，就是某个 RM，向 Leader 发送 Begin Commit 指令。这个操作和第 9 讲介绍的 2PC 稍有不同，但和客户端向 Leader 发送指令在效果上是大致相同的。同时，RM1 要向所有 Acceptor 发送 Prepared 指令。因为把事务触发也算进去了，所以整个协议有三个阶段构成，Prepare 是其中的第二阶段，而 RM 对 Prepare 指令的响应过程又拆分成了 a 和 b 两个子阶段。所以，这里的 Prepared 指令用 2a Prepared 表示，要注意这是一个完成时，表示已经准备完毕。

2. Leader 向除 RM1 外的所有 RM，发送 Prepare 指令。RM 执行指令后，向所有 Acceptor 发送消息 2a Prepared。这里的关键点是，一个 2a Prepared 消息里只包含一个 RM 的执行情况。而每个 Acceptor 都会收到所有 RM 发送的消息，从而得到全局 RM 的执行情况。
3. 每个 Acceptor 向 Leader 汇报自己掌握的全局状态，载体是消息 2b Prepared。2b Prepared 是对 2a Prepared 的合并，每个消息都记录了所有 RM 的执行情况。最后，Leader 基于多数派得出了最终的全局状态。这一点和 2PC 完全不同，事务的状态完全由投票决定，Leader 也就是事务协调者，是没有独立判断逻辑的。
4. Leader 基于已知的全局状态，向所有 RM 发送 Commit 指令。

这个过程中，如果 Acceptor 总数是 $2F+1$ ，那么每个 RM 就有 $2F+1$ 条路径与 Leader 通讯。只要保证其中 $F+1$ 条路径是畅通的，整个协议就可以正常运行。因此，Paxos Commit 协议的优势之一，就是在很大程度上避免了网络故障对系统的影响。但是，相比于 2PC 来说，它的消息数量大幅增加，而且多了一次消息延迟。目前，实际产品中还很少有使用 Paxos Commit 协议的。

第 10 讲：2PC 的延迟优化

🔗第 10 讲的核心内容是 2PC 的低延迟技术，我们先是分析了延迟的主要构成，发现延迟时间与事务中的写入操作数量线性相关，然后将延迟时间的计量单位统一为共识算法延迟 L_c ，最后得到了下面的延迟计算公式：

$$L_{txn} = (W + 1) * L_c$$

随后，我为你讲解了三种优化技术，都是基于 Percolator 模型的，分别是缓存提交写、管道和并行提交。TiDB 采用“缓存提交写”达到了 2 倍共识算法延迟，但这个方案的缺点是缓存 SQL 的节点会出现瓶颈，而且不再是交互事务。CockroachDB 采用了管道和并行提交技术，整体延迟缩短到了 1 倍共识算法延迟，可能是目前最极致的优化方法了。

这一讲的是思考题是：虽然 CockroachDB 的优化已经比较极致了，但还有些优化方法也很有趣，请你介绍下自己了解的 2PC 优化方法。

关于这个问题，我们刚刚讲的 Paxos Commit 其实已经是一种 2PC 的优化方法了。另外，在 Spanner 论文 “[Spanner: Google' s Globally-Distributed Database](#)” 中也介绍了它的 2PC 优化方式。Spanner 的 2PC 优化特点在于由客户端负责第一段协调，发送 prepare 指令，减少了节点间的通讯。具体的内容，你可以参考下这篇论文。

在留言区，我看到很多同学对并行提交有不同的理解。我要再提示一下，并行提交中的**异步写事务日志只是根据每个数据项的写入情况，追溯出事务的状态，然后落盘保存，整个过程并没有任何重试或者回滚的操作**。这是因为，在之前的同步操作过程中，负责管道写入的同步线程，已经明确知道了每个数据项的写入情况，也就是确定了事务的状态，不同步落盘只是为了避免由此带来的共识算法延迟。

第 11 讲：读写冲突、MVCC 与快照

在 [第 11 讲](#) 中，我们介绍如何避免读写冲突的解决方案，其中很重要的概念就是 MVCC 和快照。MVCC 是单体数据库普遍使用的一种技术，通过记录数据项历史版本的方式，提升系统应对多事务访问的并发处理能力。

在 MVCC 出现前读写操作是相互阻塞的，并行能力受到很大影响。而使用 MVCC 可以实现读写无阻塞，并能够达到 RC（读已提交）隔离级别。基于 MVCC 还可以构建快照，使用快照则能够更容易地实现 RR（可重复读）和 SI（快照隔离）两个隔离级别。

首先，我们学习了 PGXC 风格分布式数据库的读写冲突处理方案。PGXC 因为使用单体数据库作为数据节点，所以沿用了 MVCC 来实现 RC。但如果要实现 RR 级别，则需要全局事务管理器（GTM）承担产生事务 ID 和记录事务状态的职责。

然后，我们介绍了 TiDB 和 CockroachDB 这两种 NewSQL 风格分布式数据库的读写冲突处理方案。TiDB 没有设置全局事务列表，所以读写是相互阻塞的。CockroachDB 虽然有全局事务列表，但由于它的目标隔离级别是可串行化，所以也没有采用快照方式，读写也是相互阻塞的。

这一讲的思考题是：在介绍的几种读写冲突的处理方案中，时间都是非常重要的因素，但时间是有误差的，那么你觉得时间误差会影响读写冲突的处理吗？

其实，这个问题就是引导你思考，以便更好地理解第 12 讲的内容。MVCC 机制是用时间戳作为重要依据来判别哪个数据版本是可读取的。但是，如果这个时间戳本身有误差，就

需要特定的机制来管理这个误差，从而读取到正确的数据版本。更详细的内容，你可以去学习下 [🔗 第 12 讲](#)。

“真名不叫黄金”同学的答案非常准确，抓住了时钟置信区间这个关键点，分析思路也很清晰，点赞。

第 12 讲：读写操作与时间误差

在 [🔗 第 12 讲](#) 中，我们给出了时间误差的具体控制手段，也就是写等待和读等待。

Spanner 采用了写等待方案，也就是 Commit Wait，理论上每个写事务都要等待一个时间置信区间。对 Spanner 来说这个区间最大是 7 毫秒，均值是 4 毫秒。但是，由于 Spanner 的 2PC 设计，需要再增加一个时间置信区间，来确保提交时间戳晚于预备时间戳。所以，实际上 Spanner 的写等待时间就是两倍时间置信区间，均值达到了 8 毫秒。传说中，Spanner 的 TPS 是 125 就是用这个均值计算的（1 秒 / 8 毫秒），但如果事务之间操作的数据不重叠，其实是不受这个限制的。

CockroachDB 采用了读等待方式，就是在所有的读操作执行前处理时间置信区间。读等待的优点是偶发，只有读操作落入写操作的置信区间才需要重启，进行等待。但是，重启后的读操作可能继续落入其他写操作的置信区间，引发多次重启。所以，读等待的缺点是等待时间可能比较长。

这一讲的思考题是：读等待和写等待都是通过等待的方式，度过不确定的时间误差，从而给出确定性的读写顺序，但性能会明显下降。那么在什么情况下，不用“等待”也能达到线性一致性或因果一致性呢？”

我为这个问题准备了两个答案。

第一个答案是要复习 [🔗 第 5 讲](#) 的相关知识。如果分布式数据库使用了 TSO，保证全局时钟的单向递增，那么就不再需要等待了，因为在事件发生时已经按照全序排列并进行了记录。

第二个答案是就时间的话题做下延展。“等待”是为了让事件先后关系明确，消除模糊的边界，但这个思路还是站在上帝视角。

我们试想一种场景，事件 1 是小明正在北京的饭馆里与人谈论小刚的大学趣事，事件 2 是小刚在温哥华的公寓里打了一个喷嚏。如果事件 1 发生后 4 毫秒，事件 2 才发生，那么从绝对时间，也就是全序上看，两者是有先后关系的。但是从因果关系上，也就是偏序上看，事件 1 与事件 2 是没有联系的。因为科学承认的最快速度是光速，从北京到温哥华，即使是光速也无法在 4 毫秒内到达。那么，从偏序关系上看，事件 1 和事件 2 是并发的，因为事件 1 没有机会影响到事件 2。

当然，这个理论不是我看多了科幻电影想出来的，它来自 Lamport 的论文 “[Time, Clocks, and the Ordering of Events in a Distributed System](#)”。

所以，假设两个事件发生地的距离除以光速得到一个时间 X ，两个事件的时间戳间隔是 Y ，时钟误差是 Z 。如果 $X > Y + Z$ ，那么可以确定两个事件是并行发生的，事件 2 就不用读等待了。这是因为既然事件是并行的，事件 2 看不到事件 1 的结果也就是正常的了。

第 13 讲：广义乐观和狭义乐观

在[第 13 讲](#)中，我们开始探讨“写写冲突”的控制技术，这也是并发控制最核心的内容。大型系统之所以能够承载海量并发，就在于底层数据库有强大的并发处理能力。

并发控制分为乐观协议和悲观协议两大类，单体数据库大多使用悲观协议。TiDB 和 CockroachDB 都在早期版本中提供了乐观协议，但在后来的产品演进又改回了悲观协议，其主要原因是事务竞争激烈和对遗留应用系统的兼容。

我们还从经典理论教材中提取了并发控制的四阶段，忽略掉计算（C）阶段后，悲观协议与乐观协议的区别在于有效性验证（V）、读（R）、写（W）这三阶段的排序不同。在分布式架构下，有效性验证又分为局部有效性验证和全局有效性验证。因此，乐观又分为狭义乐观和广义乐观，而狭义乐观就是学术领域常说的 OCC。TiDB 的乐观锁，因为没有全局有效性验证，不严格符合 VRW 悲观协议排序，所以是广义乐观。而 TiDB 后来增加的悲观锁，增加了全局有效性验证，是严格的 VRW，所以是悲观协议。

这一讲的思考题是：在了解乐观协议及 TiDB 乐观转悲观的设计后，请你来推测下 CockroachDB 向悲观协转换大概会采用什么方式？

这个问题是为了引出第 14 讲的主题。CockroachDB 早期的乐观协议也是广义乐观，在局部看是悲观协议，使用了串行化图检测（SGT）的方式。SGT 是区别于锁的另一种控制技

术，具有更好的性能。CockroachDB 的改良方式是增加了全局的锁表（Lock Table），局部保留了原有的 SGT。

第 14 讲：悲观协议

在第 14 讲中，我们首先讨论了完整的并发控制技术体系，选择了 “[Transactional Information Systems](#)” 定义狭义乐观和其他悲观协议这种的组织形式，而后对 2PL 的定义和各种变体进行了说明。我们根据 S2PL 的定义，可以推导出 Percolator 模型属于 S2PL 的结论。S2PL 虽然使用广泛，但不能在生产级支持可串行化隔离。

PostgreSQL 的 SSI 设计给出了另一种实现，它的理论基础是 SGT。CockroachDB 在此基础上设计了读时间戳缓存（RTC），降低了原有 SIREAD 的开销，达到了生产级性能要求。最后，我还和你一起学习了 CockroachDB 采用全局锁表实现悲观协议的原理。

这一讲的思考题是：我们之前已经介绍过 MVCC，它是一项重要的并发控制技术，你觉得该如何理解它和乐观协议、悲观协议的关系？

就像我们在 [第 11 讲](#)中所说的，MVCC 已经是数据库的底层技术，与乐观协议、悲观协议下的各项技术是两个不同的维度，最后形成了 MVTO、MV2PL、MVSGT 等技术。这些技术考虑了多版本情况下的处理，但遵循的基本原理还是一样的。

小结

正如我在这一讲开头提到的，第 9 到第 14 这 6 讲的内容，都是围绕着分布式数据库的事务展开的，重点就是原子性和隔离性的协议、算法和工程实现。

对于原子性，我们主要关注非功能性指标背后的架构优化和理论创新，尤其是 NewSQL 风格分布式数据库在 2PC 的三个传统难题，也就是同步阻塞、单点故障、数据一致性上都取得的突破。

隔离性则不同，早在单体数据库时代，架构设计就在正确性，也就是隔离级别上作了妥协，换取性能。而分布式数据库在重新挑战了隔离性这个难题，CockroachDB 在这方面的探索更是意义重大，它实践了一种兼顾正确性和性能的技术方案。

分布式数据全景图 2/4



James C. Corbett et al.: [☞ *Spanner: Google's Globally-Distributed Database*](#)

Jim Gray and Leslie Lamport: [☞ *Consensus on Transaction Commit*](#)

Leslie Lamport: [☞ *Time, Clocks, and the Ordering of Events in a Distributed System*](#)

Gerhard Weikum and Gottfried Vossen: [☞ *Transactional Information Systems*](#)

提建议

更多课程推荐

程序员的数学基础课

在实战中重新理解数学

黄申

LinkedIn 资深数据科学家



涨价倒计时 🕒

今日秒杀 **¥79**, 9月11日涨价至 **¥129**

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 14 | 隔离性：实现悲观协议，除了锁还有别的办法吗？

精选留言 (1)

写留言



趁早
2020-09-11

如果学完一个章节有一个测试掌握程度的就更好了
展开

