25丨事务（四）：持久性，吃一碗粉就付一碗粉的钱

2022-03-30 陈现麟

《深入浅出分布式技术原理》 课程介绍

讲述：张浩
时长 11:31　大小 10.56M

你好，我是陈现麟。

通过上节课的学习，我们掌握了通过 MVCC 和 2PL 实现分布式事务隔离性的技术原理，并且也明白了隔离级别是事务在正确性和性能之间的一个权衡，以后在极客时间的业务研发中，我们就可以根据业务特点，对事情的隔离级别做出正确的选择了。

虽然事务在有了一致性、原子性和隔离性的保障后，已经可以很好地保障业务在各种使用场景下的正确性了，但是如果机器突然断电或者崩溃，导致已经提交成功的事务数据丢失了，最终也就功亏一篑了。

所以在这节课中，我们将一起来讨论如何确保机器在突然断电、崩溃等异常情况下，不会将已经成功的事务数据丢失掉的问题，即事务的持久性。

我们首先会通过持久性的定义分析出它面临的挑战，然后再一起讨论一下如何通过非易失性存储来保障事务的持久性，最后一起讨论在分布式系统中，如何通过数据复制来进一步提高事务的持久性。

持久性的挑战

持久性在事务中的定义是，如果一个事务已经提交，不论什么原因，它产生的结果都是永久存在的，这保证了事务的结果不会丢失。通过持久性的定义，我们会发现要保证事务的持久性，一个显而易见的思路就是，将事务的结果立即写入到非易失性存储设备中，比如 SSD 硬盘和 SATA 硬盘等，并且写入的副本数越多，持久性就越高。

但是理想很完美，现实却很骨感，**将数据写入到硬盘中其实是非常消耗性能的**。如果将每一个事务所有的操作结果，都实时写入到持久化的存储设备中，这样的数据库几乎就是不可用的，更不用说多副本的写入了，那么我们如何来解决存储设备的写入性能和事务持久性之间的矛盾呢？

如何通过非易失性存储保障持久性

关于这个问题，我们需要从磁盘设备的特性开始说起。对于 SATA 硬盘来说，可以将它简单理解为一个有很多同心圆的圆盘，在写入数据的时候，会经历以下几个步骤：

- 寻道，找到数据所在的同心圆，这个时间是毫秒级别的；
- 寻址，找到数据所在的同心圆的位置，这个时间也是毫秒级别的；
- 开始读写数据，每秒可以读写的数据量为 100M 级别的数据，这个是非常快的。

我们可以从上面看出，如果没有寻道和寻址这两个步骤， SATA 硬盘的性能其实是非常不错的。那么如何避免寻道和寻址呢？如果第一次寻道和寻址后，持续对数据进行大量的读写，即顺序读写，是可以忽略寻道和寻址的时间消耗的。而对应顺序读写的是**随机读写，它每一次读写的数据量很小，并且数据位置不相邻，都需要先寻道、寻址，然后才能进行数据读写，所以随机读写的性能是非常差的**。

对于 SSD 硬盘，寻址的情况则大大改观，不需要像 SATA 硬盘一样机械地寻道、寻址，它可以通过电路直接获得读写的地址。但是，SSD 硬盘与传统的 SATA 硬盘有一点不同，即它不能够覆盖写，所以对于已经存在数据的 SSD 磁盘来说，一次数据的写入需要分为 2 个步骤：

- 擦除 SSD 上已有的数据；
- 写入新的数据。

但是对于 SSD 来说，一般每次写入的最小单位为 Page ，一个 Page 的大小为 4KB，而每次擦除的大小单位为 Block ， Block 通常由 64 或 128 个 Page 组成。

由此看出， SSD 的写入与擦除的单位大小不匹配，那么如果仅仅是要修改一个 Page 的数据，在单个 Block 之中没有了空余的 Page 时，需要先读取 Block 的内容，然后擦除一个 Block 的数据，再将 Block 的内容和修改的内容进行合并，写入一个 Block 的数据。而这就会导致原本只需要写入 4KB 的数据，最终却写入了 64 倍甚至是 128 倍的数据，出现写放大的问题。

从上面的讨论中，**我们发现对于 SSD 磁盘来说，写放大是无法避免的，相比于顺序写入，随机写入会大大加剧写放大的问题**。

总而言之，不论是 SATA 硬盘还是 SSD 硬盘，从硬盘自身的特点来说，顺序读写的性能都要远远高于随机读写。另外从系统的角度来看，顺序读写在预读和缓存命中率等方面也要大大优于随机读写。

现在，我们就可以回答存储设备的写入性能和事务持久性之间矛盾的问题了：由于事务的持久性是必须的，如果一个事务已经提交，不论什么原因，它产生的结果都是永久存在的，所以对于单节点来说，我们可以先在内存中将事务的操作完成，然后将处理的结果顺序写入日志文件中，这就避免了事务操作结果随机写入存储的性能问题了。

然后我们再提交事务，这样一来，哪怕事务提交后，机器立即崩溃了，在机器故障恢复后，系统依然能通过日志文件，恢复已经提交的事务。

所以，通过顺序写入日志的形式，避免了非易失性存储设备随机写入性能差的问题，达到了事务提交时，所有事务操作结果都写入存储设备的目的。在这个时候，即使系统崩溃，事务的持久性也是有保障的。**我们把这种通过顺序写入日志的形式，称之为重做日志（RedoLog）或预写日志（Write Ahead Log）**。

如何通过数据冗余保障持久性

通过 Redo Log 或 WAL，是否可以完美地解决事务持久性的问题呢？其实还是不够的。虽然 Redo Log 能保证系统在崩溃、重启等问题出现时的持久性，但是当存储设备出现了故障，比如数据都不可读的时候，还是会出现即使事务已经提交成功，但是事务结果却丢失的情况。那么这个问题应该如何处理呢？

有一种思路是，**通过磁盘阵列，从磁盘内部通过冗余数据来解决**。比如 RAID 1 ，我们将多块硬盘组成一个磁盘阵列，磁盘阵列中每块磁盘都有一个或多个是副本磁盘。事务的每一次写入都同时写入所有的副本硬盘，这样只要不是所有的副本磁盘同时出现故障的情况，我们就都可以正常从磁盘上读到数据，不会影响事务的持久性。还有一种磁盘阵列的方式是 RAID 5 ，它是通过冗余校验数据的方法来保障持久性。

磁盘阵列的方法确实可以解决事务的持久性问题，但是由于磁盘阵列上多块硬盘的地理位置通常都是在一起的，这样如果出现地震、火灾和洪水等自然灾害时，可能会导致整个磁盘阵列上的硬盘都不可用，那么事务的持久性就不能被保证了。

而另外一个思路是**通过增加副本，通过网络复制数据来解决**。其实这个问题在“复制”系列课程中已经详细讨论过了，但是对于事务的场景来说，由于数据的复制必须是线性一致性的，所以我们只能采用同步的主从复制，但是这个方式在性能和可用性方面都存在问题。

- 性能问题：一次写入必须等所有的节点都写入成功，整体的写入性能取决于最慢的节点的写入性能，并且网络的不确定性会加剧性能问题。
- 可用性问题：对于同步复制来说，如果一个节点出现故障，就会导致写入失败，非常影响系统整体的可用性。

对于事务场景，如果我们不采用同步的主从复制，是否有其他的办法来解决呢？

其实我们可以通过 Raft 或者 Paxos 之类的共识算法来解决。对于数据复制到多个副本来说，其实就是多个副本对写入的结果达成共识，利用 Raft 或者 Paxos 之类的共识算法进行数据的复制，可以实现线性一致性，同时共识算法可以避免同步主从复制在性能、可用性问题和磁盘阵列多副本地理位置相近的问题。

- 性能问题：一次写入只需要等大多数的节点都写入成功即可，整体的写入性能取决于最快的大多数节点的写入性能。

- 可用性问题：只要出现故障的节点数不超过大多数，系统就会写入成功，它能容忍少数节点的故障。
- 地理位置相近的问题：数据通过网络复制，可以将副本分布到不同的数据中心、城市或者大洲，进一步提高事务的持久性。

对于共识算法，我们会在后面的课程“一致性与共识”中详细讨论，在这节课里，你只要知道对于存储系统内部的多节点数据副本，一般都通过共识算法来解决即可。

到这里，我们就学习完了“事务”序列的课程，这里从事务的四个特性的角度，总结一下：

- **一致性（C）**：是指事务的正确性，需要底层数据的线性一致性，事务层的原子性、隔离性、持久性，以及数据库层面的约束检测和应用层的约束检测来保证。
- **原子性（A）**：是指事务操作的不可分割性，一般通过原子提交协议 2PC 或 3PC 来保障。
- **隔离性（I）**：是指事务操作在并发控制，一般数据库都提供弱隔离性，是数据库在性能和正确性之间的衡权，一般通过 MVCC 或者 2PL 来实现。
- **持久性（D）**：是指已提交的事务结果不可丢失，一般在单机上通过非易失性存储来保障，在分布式场景下，通过数据冗余来保障。

总结

本节课，我们知道了事务在实现持久性的过程中，会面临性能和可用性这两个方面的挑战。

首先，要保障事务在系统宕机情况下的持久性，必须保证事务的操作结果能够立即保存到硬盘之类的非易失性存储中，但是不论是 SATA 硬盘还是 SSD 硬盘，对于这一类随机读写操作都会面临严重的性能问题，**目前我们主要是通过重做日志（RedoLog）或预写日志（Write Ahead Log），将随机读写转化为顺序读写来提高事务的性能**。

其次，要保障事务在磁盘故障情况下的持久性，必须将数据复制到多块磁盘上，这节课我们介绍了两种思路：**一是通过磁盘阵列，从磁盘内部复制数据来解决；另一种是通过外部的数据复制来解决**。

其中，磁盘阵列的多块硬盘的地理位置通常都是在一起的，地震、火灾和洪水等自然灾害，可能会导致整个磁盘阵列同时毁坏；而外部的数据复制方法需要保证数据的强一致性，它会面临性能和可用性的问题，这里我们主要通过 Raft 或者 Paxos 之类的共识算法来解决。

最后，我们可以看到，事务的持久性会让事务的结果被持久保存下来，不会出现因为数据丢失、毁坏而导致不认账的情况，这也就是本节课标题提到的“吃一碗粉就付一碗粉的钱”的真正含义。

思考题

通过课程的学习，我们知道了存储设备的特性会影响存储引擎的设计，同样，业务存储的数据特点也会影响存储引擎的设计，请你来思考一下，如果我们的业务需要存储很多非常小的文件（比如平均几十 K），应该怎么来设计存储引擎呢？

欢迎你在留言区发表你的看法。如果这节课对你有帮助，也推荐你分享给更多的同事、朋友。

上一篇 24丨事务（三）：隔离性，正确与性能之间权衡的艺术

下一篇 26丨一致性与共识（一）：数据一致性都有哪些级别？

精选留言 (1)

写留言

wenxuan

2022-03-31

思考题的一些想法，可以把多个小文件合并成大文件（比如 8 M）存储，记录下每个原始文件的起始位置和偏移量，通过 `seek` 快速定位到起始位置进行读取。删除导致的文件空洞可以通过定期合并来解决。

作者回复：非常赞的想法，Facebook 的图片存储 haystack 就是这样做的。



3