10 | 事务处理与恢复（上）：数据库崩溃后如何保证数据不丢失?

上一讲我们探讨了一个典型的面向分布式数据库所使用的存储引擎——LSM 树。那么这一讲，我将为你介绍存储引擎的精华部分，也就是事务管理。首先我将从使用者角度，介绍事务的特性，也就是 ACID；而后简要说明存储引擎是通过什么组件来支持这些特性的。

为了保持这些特性，事务管理器需要考虑各种可能的问题与故障，如数据库进程崩溃、磁盘故障等。在面临各种故障时，如何保证 ACID 特性，我会在“数据库恢复”部分为你介绍。

由于这部分内容较多，我分成了上下两讲来向你讲述。下一讲我会接着介绍数据库的隔离级别与并发控制，它们是数据库提供给应用开发人员的礼物，可以让其非常轻易地实现并发数据的一致性。

以上就是这部分内容的学习脉络，现在让我们从事务的概述说起。

事务概述

事务管理是数据库中存储引擎的一个相当独立并且重要的组件，它可以保证对数据库的一系列操作看起来就像只有一步操作一样。这大大简化了面向数据库的应用的开发，特别是在高并发场景下，其意义更为重要。

一个典型的案例就是转账操作：从甲处转 100 元给乙。现实生活中，这个操作是原子的，因为纸币是不可复制的。但是在计算机系统内，这个操作实际上是由两个操作组成：甲账户减 100、乙账户加 100。两个操作就会面临风险，比如在转账的同时，丁又从甲处转走 100（此时甲给乙的 100 未扣除），而如果此时账户内钱不够，这两笔操作中的一笔可能会失败；又比如，两个操作过程中数据库崩溃，重启后发现甲的账户已经没了 100，而乙账户还没有增加，或者相反。

为了解决上面类似的问题，人们在数据库特别是存储引擎层面提出了事务的概念。下面我来说说事务的经典特性 ACID。

ACID

A：原子性

原子性保证了事务内的所有操作是不可分割的，也就是它们要么全部成功，要么全部失败，不存在部分成功的情况。成功的标志是在事务的最后会有提交（Commit）操作，它成功后会被认为整个事务成功。而失败会分成两种情况，一种是执行回滚（Rollback）操作，另一种就是数据库进程崩溃退出。

原子性是数据库提供给使用者的保证，是为了模拟现实原子操作，如上文提到的转账。在现实生活中，一些看似不可分割的操作转换为计算机操作却并不是单一操作。而原子性就是对现实生活中原子操作的保证。

C：一致性

一致性其实是受用户与数据库共同控制的，而不只是数据库提供的一个服务。它首先是一个业务层面的约束，比如开篇中的例子，甲向乙转 100 元。业务应用首先要保证在甲账户扣款 100 元，而且在乙账户增加 100 元，这个操作所带来的一致性与数据库是无关的。而数据库是通过 AID 来保证这两个正确的动作可以得到最终正确的结果。

这里的一致性与模块一中的分布式一致性有本质区别，想了解详细对比的同学，请移步到“05 | 一致性与 CAP 模型：为什么需要分布式一致性”，这里就不进行赘述了。

I：隔离性

事务的一个伟大之处是能处理并发操作，也就是不同的事务在运行的时候可以互相不干扰，就像没有别的事务发生一样。做并发编程的同学会对此深有体会，处理并发操作需要的精力与经验与处理串行操作完全不在一个等级上。而隔离性恰好能将实际上并发的操

作，转化为从使用者角度看却是串行的，从而大大降低使用难度。

当然在实际案例中，以上描述的强并发控制性能会偏低。一般数据库会定义多种的隔离级别来提供不同等级的并发处理能力，也就是一个事务在较低隔离级别下很可能被其他事务看见。详细内容我会在“隔离级别”部分进行说明。

D：持久性

持久性比较简单，就是事务一旦被提交，那么它对数据库的修改就可以保留下来。这里要注意这个“保存下来”不仅仅意味着别的事务能查询到，更重要的是在数据库面临系统故障、进程崩溃等问题时，提交的数据在数据库恢复后，依然可以完整地读取出来。

以上就是事务的四种重要的特性，那么事务在存储引擎内部有哪些组件来满足上面的特性呢？我接下来要为你介绍的就是一个典型的事务管理组件。

事务管理器

事务主要由事务管理器来控制，它负责协调、调度和跟踪事务状态和每个执行步骤。当然这与分布式事务两阶段提交（2PC）中的事务管理器是不同的，关于分布式事务的内容我将在下一个模块详细介绍。

页缓存

关于事务管理器，首先要提到的就是页缓存（Page Cache）或者缓冲池（Buffer Pool），它是磁盘和存储引擎其他组件的一个中间层。数据首先被写入到缓存里，而后同步到数据磁盘上。它一般对于其他组件，特别是对于写入来说是透明的，写入组件以为是将数据写入磁盘，实际上是写入了缓存中。这个时候如果系统出现故障，数据就会有丢失的风险，故需要本讲后面“如何恢复事务”要介绍的手段来解决这个问题。

缓存首先解决了内存与磁盘之间的速度差，同时可以在不改变算法的情况下优化数据库的性能。但是，内存毕竟有限，不可能将磁盘中的所有数据进行缓存。这时候就需要进行刷盘来释放缓存，刷盘操作一般是异步周期性执行的，这样做的好处是不会阻塞正常的写入和读取。

刷盘时需要注意，脏页（被修改的页缓存）如果被其他对象引用，那么刷盘后不能马上释放空间，需要等到它没有引用的时候再从缓存中释放。**刷盘操作同时需要与提交日志检查点进行配合，从而保证 D**，也就是持久性。

当缓存到达一定阈值后，就不得不将有些旧的值从缓存中移除。这个时候就需要缓存淘汰算法来帮忙释放空间。这里有 FIFO、LRU、表盘（Clock）和 LFU 等算法，感兴趣的话你可以根据这几个关键词自行学习。

最后存在部分数据我们希望它们一直在缓存中，且不受淘汰算法的影响，这时候我们可以把它们“锁”（Pinned）在缓存里。比如 B 树的高节点，它们一般数据量不大，且每次查询都需要访问。还有一些经常访问的元数据也会长期保存在缓存中。

日志管理器

其次是日志管理器，它保存了一组数据的历史操作记录。缓存内的数据没有刷入磁盘前，系统就崩溃了，通过回放日志，缓存中的数据可以恢复出来。另外，在回滚场景，这些日志可以将修改前的数据恢复出来。

锁管理器

最后要介绍的就是非常重要的锁管理器，它保证了事务访问共享资源时不会打破这些资源的完整性约束。同时，它也可以保证事务可以串行执行。关于锁的内容我会在后面详细说明。

以上就是事务管理的主要组件，下面我将从数据库恢复事务的角度介绍日志管理相关内容。

数据库如何恢复事务

数据库系统是由一系列硬件和软件组成的复杂生态系统，其中每个组件都有产生各种稳定性问题的可能，且将它们组合为数据库系统后，这种可能被进一步放大了。而数据库的设计者必须为这种潜在的稳定性问题给出自己的解决方案，并使数据库作出某种“承诺”。

提交日志，即 CommitLog 或 WAL（Write-Ahead Log）就是应对此种问题的有效手段。这种日志记录了数据库的所有操作，并使用追加（Append）模式记录在磁盘的日志文件中。

上文中我们知道数据库的写操作首先是写入了缓存，而后再刷入磁盘中。但是在刷盘之前，其实这些数据已经以日志的形式保存在了磁盘的提交日志里面。当数据没有刷入磁盘而仅仅驻留在缓存时，这些日志可以保证数据的持久性。也就是，一旦数据库遭遇故障，可以从日志中恢复出来数据。

那么提交日志具有哪些特性来保障这些功能呢？下面来看一下日志的特性。

提交日志的特性

首先，提交日志非常类似于上一讲介绍的 LSM 树的磁盘文件特性，都是顺序写入且不可变。其益处也是相同的，顺序写保障了写入的高性能，不可变保证了读取可以安全地从前到后读取里面的数据。

提交日志一般都会被分配一个序列号作为唯一键，这个序号不是一个自增数字，就是一个时间戳。此外，每条日志都非常小，有些数据库会将它们进行缓存而后批量写入磁盘。这就导致，默写情况下日志不能完全恢复数据库，这是对于性能的考虑，大部分数据库会给出不同的参数来描述日志缓存刷盘的行为，用户可在性能与恢复数据完整性上作出平衡。

而事务在提交的时候，一定要保证其日志已经写入提交日志中。也就是事务内容完全写入日志是事务完成的一个非常重要的标志。

日志在理论上可以无限增长，但实际上没有意义。因为一旦数据从缓存中被刷入磁盘，该操作之前的日志就没有意义了，此时日志就可以被截断（Trim），从而释放空间。而这个被截断的点，我们一般称为检查点。**检查点之前的页缓存中的脏页需要被完全刷入磁盘中**。

日志在实现的时候，一般是由一组文件组成。日志在文件中顺序循环写入，如果一个文件中的数据都是检查点之前的旧数据，那么新日志就可以覆盖它们，从而避免新建文件的问题。同时，将不同文件放入不同磁盘，以提高日志系统的可用性。

物理日志 Redo Log 与逻辑日志 Undo Log

事务对数据的修改其实是一种状态的改变，比如将 3 改为 5。这里我们将 3 称为前镜像（before-image），而 5 称为后镜像（after-image）。我们可以得到如下公式：

1. 前镜像+redo log=后镜像
2. 后镜像+undo log=前镜像

redo log 存储了页面和数据变化的所有历史记录，我们称它为物理日志。而 undo log 需要一个原始状态，同时包含相对这个状态的操作，所以又被称为逻辑日志。我们使用 redo 和 undo 就可以将数据向前或向后进行转换，这其实就是事务操作算法的基础。

Steal 与 Force 策略

redo 和 undo 有两种写入策略：steal 和 force。

steal 策略是说允许将事务中未提交的缓存数据写入数据库，而 no-steal 则是不能。可以看到如果是 steal 模式，说明数据从后镜像转变为前镜像了，这就需要 undo log 配合，将被覆盖的数据写入 undo log，以备事务回滚的时候恢复数据，从而可以恢复到前镜像状态。

force 策略是说事务提交的时候，需要将所有操作进行刷盘，而 no-force 则不需要。可以看到如果是 no-force，数据在磁盘上还是前镜像状态。这就需要 redo log 来配合，以备在系统出现故障后，从 redo log 里面恢复缓存中的数据，从而能转变为后镜像状态。

从上可知，**当代数据库存储引擎大部分都有 undo log 和 redo log，那么它们就是 steal/no-force 策略的数据库**。

下面再来说一个算法。

ARIES 数据恢复算法

这个算法全称为 Algorithm for Recovery and Isolation Exploiting Semantics。

该算法同时使用 undo log 和 redo log 来完成数据库故障崩溃后的恢复工作，其处理流程分为如下三个步骤。

1. 首先数据库重新启动后，进入分析模式。检查崩溃时数据库的脏页情况，用来识别需要从 redo 的什么位置开始恢复数据。同时搜集 undo 的信息去回滚未完成的事务。
2. 进入执行 redo 的阶段。该过程通过 redo log 的回放，将在页缓存中但是没有持久化到磁盘的数据恢复出来。这里注意，**除了恢复了已提交的数据，一部分未提交的数据也恢复出来了**。
3. 进入执行 undo 的阶段。这个阶段会回滚所有在上一阶段被恢复的未提交事务。为了防止该阶段执行时数据库再次崩溃，存储引擎会记录下已执行的 undo 操作，防止它们重复被执行。

ARIES 算法虽然被提出多年，但其概念和执行过程依然在现代存储引擎中扮演着重要作用。

以上我们讲解了数据库如何恢复数据，保持一致性状态。它对应着 AID（C 如前文所示，是一种约束，一般不认为是数据库提供的功能）中的 AD。同时我们也要知道以提交日志为代表的数据库恢复技术，在没有事务概念的数据库中也扮演着重要的作用，因为页缓存是无处不在的，解决主存掉电丢失数据的问题，是提交日志的主要功能。

总结

那么这一讲就介绍到这了。我们从 ACID 原理出发，介绍了管理事务的众多组件，而后重点介绍了如何保证数据的持久性，这主要通过提交日志来实现。其余有关隔离性的概念，我将会在下一讲接着介绍。

教学相长

依然留给你一道思考题：有没有不使用日志方式来恢复数据库的方案呢?

欢迎你在评论区留言，我们一起探讨，一起进步，下一讲再见。