**数据库事务**

主讲人：team

分享时间：2020/9/16

分享成员：卞娜云、赵菊文、张华、赵飞

1. **什么是事务**

事务是指由一系列数据库操作组成的一个完整的逻辑过程，这个过程中的所有操作要么都成功，要么都不成功。比如：常见的例子就是银行转账的例子，一次转账操作会包含多个数据库操作，而这些数据库操作需要放到一个事务当中，保证其要么都成功，要么都不成功。

**1.1 ACID**

ACID 四个特性。也即：原子性，一致性，隔离性，持久性。

A 原子性。指整个数据库事务是不可分割的工作单位。只有使事务中所有的数据库操作都执行成功，才算整个事务成功。事务中任何一个SQL语句执行失败，已经执行成功的SQL语句也必须撤销，数据库状态应该退回到执行事务前的状态。通过 undo log 实现。

D 持久性。事务一旦提交，其结果就是永久性的。即使发生宕机等故障，数据库也能将数据恢复。通过 redo log 实现，

I 隔离性。事务的隔离性要求每个读写事务的对象对其他事务的操作对象能相互分离，即该事务提交前对其他事务都不可见，通常这使用锁来实现。通过读写锁和 MVCC 实现。

C 一致性。指事务将数据库从一种状态转变为下一种一致的状态。在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。通过 AID 共同实现。

**1.2 事务分类**

**扁平事务（Flat Transactions）**

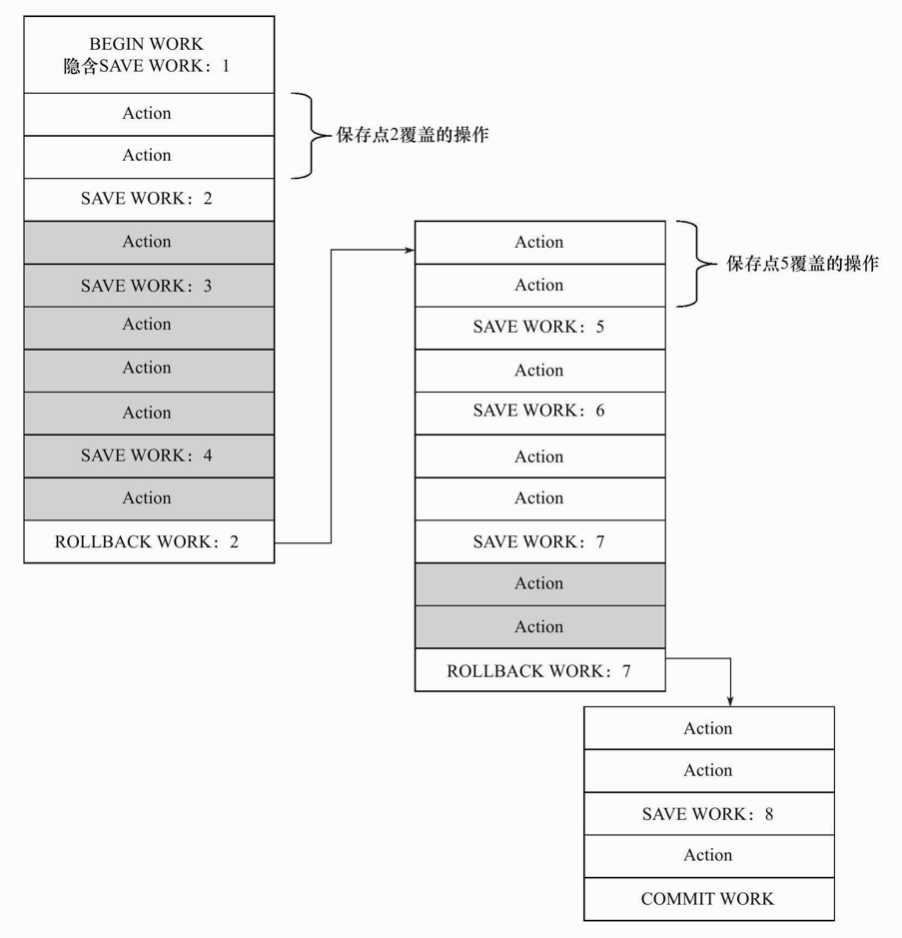
扁平事务是事务类型中最简单的一种，但在实际生产环境中，这可能是使用最为频繁的事务。在扁平事务中，所有操作都处于同一层次，其由BEGIN WORK开始，由COMMIT WORK或ROLLBACK WORK结束，其间的操作是原子的，要么都执行，要么都回滚。因此扁平事务是应用程序成为原子操作的基本组成模块。

主要限制：不能提交或者回滚事务的某一部分，或分几个步骤提交。

**带有保存点的扁平事务（Flat Transactions with Savepoints）**

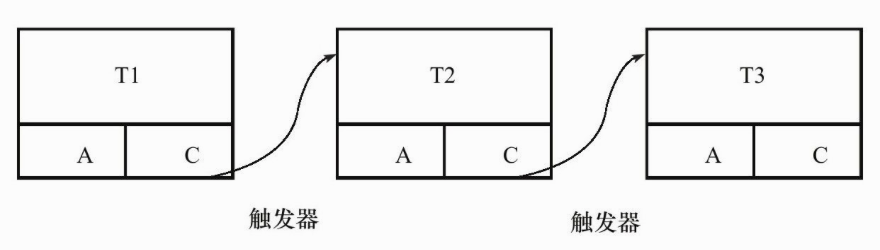
*GORM V2 refers to: https://gorm.io/docs/transactions.html#SavePoint-RollbackTo*

除了支持扁平事务支持的操作外，允许在事务执行过程中回滚到同一事务中较早的一个状态。这是因为某些事务可能在执行过程中出现的错误并不会导致所有的操作都无效，放弃整个事务不合乎要求，开销也太大。保存点（Savepoint）用来通知系统应该记住事务当前的状态，以便当之后发生错误时，事务能回到保存点当时的状态。对于扁平的事务来说，其隐式地设置了一个保存点，就是事务开始时的状态。



**链事务（Chained Transactions）**

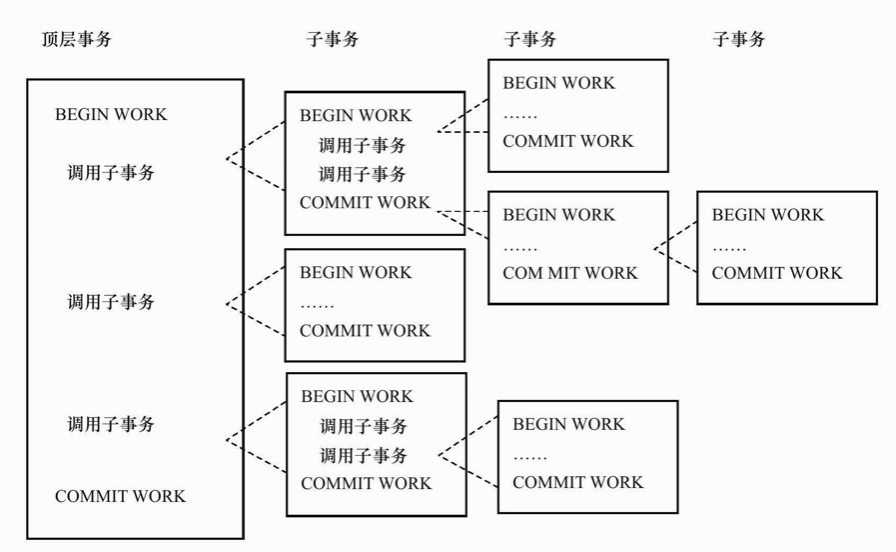
链事务在提交一个事务时，释放不需要的数据对象，将必要的处理上下文隐式地传给下一个要开始的事务。注意，提交事务操作和开始下一个事务操作将合并为一个原子操作。这意味着下一个事务将看到上一个事务的结果，就好像在一个事务中进行的一样。



**嵌套事务（Nested Transactions）**

嵌套事务是一个层次结构框架。由一个顶层事务（top-level transaction）控制着各个层次的事务。顶层事务之下嵌套的事务被称为子事务（subtransaction），其控制每一个局部的变换。

* 嵌套事务是由若干事务组成的一棵树，子树既可以是嵌套事务，也可以是扁平事务。
* 处在叶节点的事务是扁平事务。但是每个子事务从根到叶节点的距离可以是不同的。
* 位于根节点的事务称为顶层事务，其他事务称为子事务。事务的前驱称（predecessor）为父事务（parent），事务的下一层称为儿子事务（child）。
* 子事务既可以提交也可以回滚。但是它的提交操作并不马上生效，除非其父事务已经提交。因此可以推论出，任何子事物都在顶层事务提交后才真正的提交。
* 树中的任意一个事务的回滚会引起它的所有子事务一同回滚，故子事务仅保留A、C、I特性，不具有D的特性。



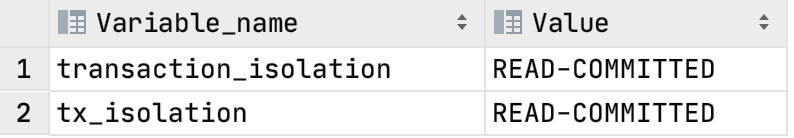
**分布式事务（Distributed Transactions）**

分布式事务通常是一个在分布式环境下运行的扁平事务，因此需要根据数据所在位置访问网络中的不同节点。因为节点不能通过调用一台数据库就完成任务。其需要访问网络中多个节点的数据库，而在每个节点的数据库执行的事务操作又都是扁平的。对于分布式事务，其同样需要满足ACID特性，要么都发生，要么都失效。

**1.3 隔离级别**

**读未提交（read uncommitted）：**一个事务还没提交时，它做的变更就能被别的事务看到-脏读；

**读提交（read committed）：**一个事务提交之后，它做的变更才会被其他事务看到；但是一个事务两次读到的数据可能是不一样的。一般来说，不可重复读的问题是可以接受的，因为其读到的是已经提交的数据，本身并不会带来很大的问题。在InnoDB存储引擎中，通过使用Next-Key Lock算法来避免不可重复读的问题。在MySQL官方文档中将不可重复读的问题定义为Phantom Problem，即幻像问题。在Next-Key Lock算法下，不仅会锁住索引，而且还锁住这些索引对应的行。



**可重复读（repeatable read）：**

* 一个事务执行过程中看到的数据，总是跟这个事务在启动时看到的数据是一致的。当然在可重复读隔离级别下，未提交变更对其他事务也是不可见的；
* InnoDB存储引擎默认支持的隔离级别是REPEATABLE READ，但是与标准SQL不同的是，InnoDB存储引擎在REPEATABLE READ事务隔离级别下，使用Next-Key Lock锁的算法，因此避免幻读的产生。这与其他数据库系统（如Microsoft SQL Server数据库）是不同的。InnoDB存储引擎在默认的REPEATABLE READ的事务隔离级别下已经能完全保证事务的隔离性要求，即达到SQL标准的SERIALIZABLE隔离级别。

**串行化（serializable ）：**对于同一行记录，“写”会加“写锁”，“读”会加“读锁”。当出现读写锁冲突的时候，后访问的事务必须等前一个事务执行完成，才能继续执行。

**举例说明：**

|  |
| --- |
| SQL CREATE TABLE example\_table(  v int ) engine=InnoDB;  INSERT INTO example\_table(v) VALUES(1); |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **事务 A** | **事务 B** | **读未提交** | **读提交** | **可重复读** | **串行化** |
| 启动事务，查询得到值1 | 启动事务 |  |  |  |  |
|  | 查询得到值1 |  |  |  |  |
|  | 将1改为2 |  |  |  |  |
| 查询得到值V1 |  | 2 | 1 | 1 | 1 |
|  | 提交事务 |  |  |  |  |
| 查询得到值V2 |  | 2 | 2 | 1 | 1 |
| 提交事务 |  |  |  |  |  |
| 查询得到值V3 |  | 2 | 2 | 2 | 2 |

**1.4 事务控制语句**

在MySQL命令行的默认设置下，事务都是自动提交（auto commit）的，即执行SQL语句后就会马上执行COMMIT操作。因此要显式地开启一个事务需使用命令BEGIN、START TRANSACTION，或者执行命令SET AUTOCOMMIT=0，禁用当前会话的自动提交。

* START TRANSACTION|BEGIN：显式地开启一个事务。
* COMMIT：要想使用这个语句的最简形式，只需发出COMMIT。也可以更详细一些，写为COMMIT WORK，不过这二者几乎是等价的。COMMIT会提交事务，并使得已对数据库做的所有修改成为永久性的。
* ROLLBACK：要想使用这个语句的最简形式，只需发出ROLLBACK。同样地，也可以写为ROLLBACK WORK，但是二者几乎是等价的。回滚会结束用户的事务，并撤销正在进行的所有未提交的修改。
* SAVEPOINT identifier∶SAVEPOINT允许在事务中创建一个保存点，一个事务中可以有多个SAVEPOINT。
* RELEASE SAVEPOINT identifier：删除一个事务的保存点，当没有一个保存点执行这句语句时，会抛出一个异常。
* ROLLBACK TO[SAVEPOINT]identifier：这个语句与SAVEPOINT命令一起使用。可以把事务回滚到标记点，而不回滚在此标记点之前的任何工作。
* SET TRANSACTION：这个语句用来设置事务的隔离级别。InnoDB存储引擎提供的事务隔离级别有：READ UNCOMMITTED、READ COMMITTED、REPEATABLE READ、SERIALIZABLE。

**1.5 隐式提交的SQL语句**

以下这些SQL语句会产生一个隐式的提交操作，即执行完这些语句后，会有一个隐式的COMMIT操作。

* DDL语句：ALTER DATABASE...UPGRADE DATA DIRECTORY NAME，ALTER EVENT，ALTER PROCEDURE，ALTER TABLE，ALTER VIEW，CREATE DATABASE，CREATE EVENT，CREATE INDEX，CREATE PROCEDURE，CREATE TABLE，CREATE TRIGGER，CREATE VIEW，DROP DATABASE，DROP EVENT，DROP INDEX，DROP PROCEDURE，DROP TABLE，DROP TRIGGER，DROP VIEW，RENAME TABLE，TRUNCATE TABLE。
* 用来隐式地修改MySQL架构的操作：CREATE USER、DROP USER、GRANT、RENAME USER、REVOKE、SET PASSWORD。
* 管理语句：ANALYZE TABLE、CACHE INDEX、CHECK TABLE、LOAD INDEX INTO CACHE、OPTIMIZE TABLE、REPAIR TABLE。

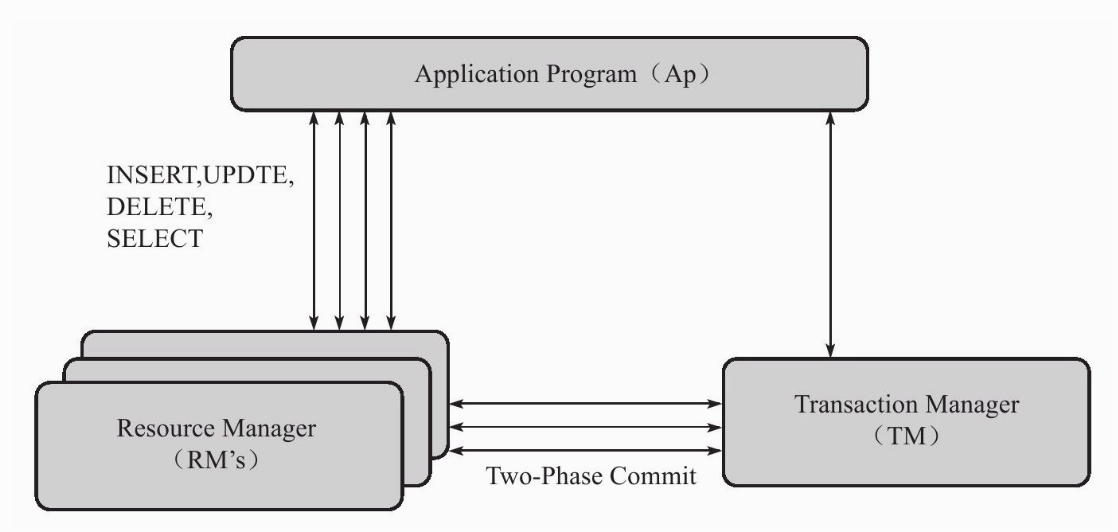
**1.6 事务统计指标**

* QPS：每秒请求数（Question Per Second）
* TPS：每秒事务处理的能力（Transaction Per Second）。
* 计算TPS的方法是（com\_commit+com\_rollback）/Uptime。但是利用这种方法进行计算的前提是：所有的事务必须都是显式提交的，如果存在隐式地提交和回滚（默认autocommit=1），不会计算到com\_commit和com\_rollback变量中，其中Uptime为数据库启动至今的时间。
* MySQL数据库中另外还有两个参数handler\_commit和handler\_rollback用于事务的统计操作。

**1.7 分布式事务**

分布式事务指的是允许多个独立的事务资源（transactional resources）参与到一个全局的事务中。事务资源通常是关系型数据库系统，但也可以是其他类型的资源。全局事务要求在其中的所有参与的事务要么都提交，要么都回滚，这对于事务原有的ACID要求又有了提高。

分布式事务使用两段式提交（two-phase commit）的方式。在第一阶段，所有参与全局事务的节点都开始准备（PREPARE），告诉事务管理器它们准备好提交了。在第二阶段，事务管理器告诉资源管理器执行ROLLBACK还是COMMIT。如果任何一个节点显示不能提交，则所有的节点都被告知需要回滚。可见与本地事务不同的是，分布式事务需要多一次的PREPARE操作，待收到所有节点的同意信息后，再进行COMMIT或是ROLLBACK操作。



**外部XA事务：**

XA事务允许不同数据库之间的分布式事务，如一台服务器是MySQL数据库的，另一台是Oracle数据库的，又可能还有一台服务器是SQL Server数据库的，只要参与在全局事务中的每个节点都支持XA事务。

XA事务由一个或多个资源管理器（Resource Managers）、一个事务管理器（Transaction Manager）以及一个应用程序（Application Program）组成。

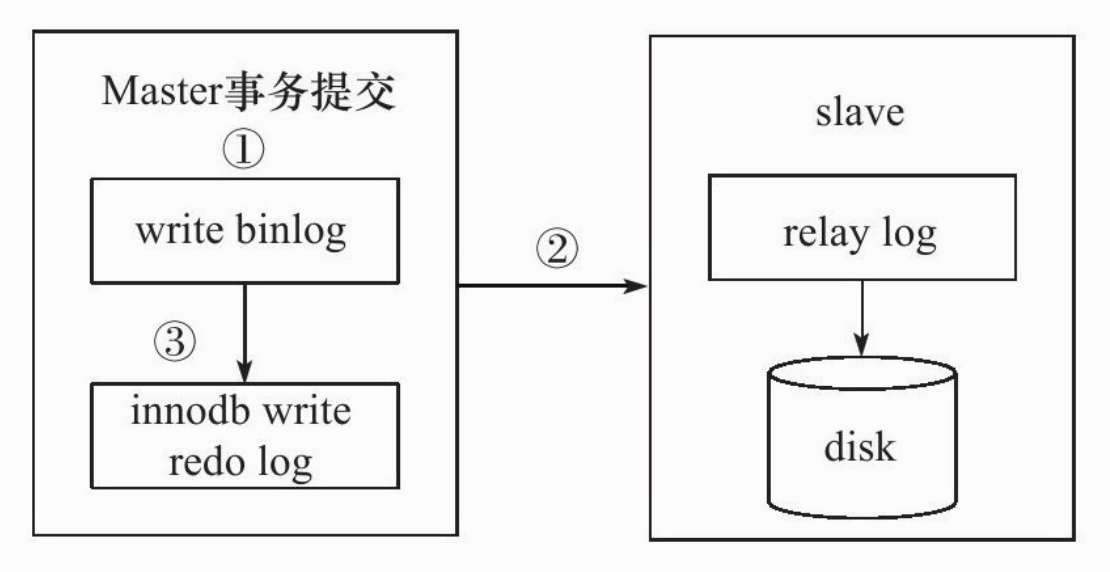
* 资源管理器：提供访问事务资源的方法。通常一个数据库就是一个资源管理器。
* 事务管理器：协调参与全局事务中的各个事务。需要和参与全局事务的所有资源管理器进行通信。
* 应用程序：定义事务的边界，指定全局事务中的操作。

在MySQL数据库的分布式事务中，资源管理器就是MySQL数据库，事务管理器为连接MySQL服务器的客户端。

**内部XA事务：**

上面讨论的分布式事务是外部事务，即资源管理器是MySQL数据库本身。在MySQL数据库中还存在另外一种分布式事务，其在存储引擎与插件之间，又或者在存储引擎与存储引擎之间，称之为内部XA事务。

由于复制的需要，因此目前绝大多数的数据库都开启了binlog功能。在事务提交时，先写二进制日志，再写InnoDB存储引擎的重做日志。对上述两个操作的要求也是原子的，即二进制日志和重做日志必须同时写入。若二进制日志先写了，而在写入InnoDB存储引擎时发生了宕机，那么slave可能会接收到master传过去的二进制日志并执行，最终导致了主从不一致的情况。



如果执行完①、②后在步骤③之前MySQL数据库发生了宕机，则会发生主从不一致的情况。为了解决这个问题，MySQL数据库在binlog与InnoDB存储引擎之间采用XA事务。当事务提交时，InnoDB存储引擎会先做一个PREPARE操作，将事务的xid写入，接着进行二进制日志的写入。如果在InnoDB存储引擎提交前，MySQL数据库宕机了，那么MySQL数据库在重启后会先检查准备的UXID事务是否已经提交，若没有，则在存储引擎层再进行一次提交操作。

**事务原理和实现**

*MySQL 的事务在引擎层实现，并不是所有的引擎都支持事务。*

ACD：**Undo log （回滚日志）和Redo log（重做日志）**

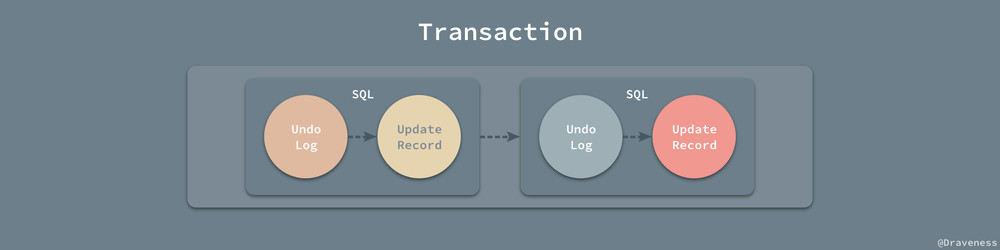
隔离性：锁和时间戳

**2.1 回滚日志**

作用：

　　保存了事务发生之前的数据的一个版本，可以用于回滚，同时可以提供多版本并发控制下的读（MVCC），也即非锁定读

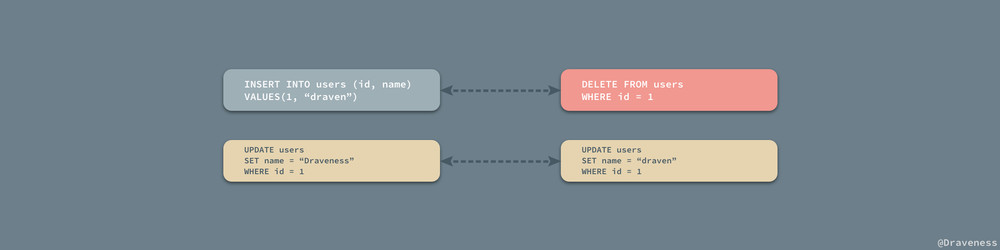
想要保证事务的原子性，就需要在异常发生时，对已经执行的操作进行**回滚**，而在 MySQL 中，恢复机制是通过*回滚日志*（undo log）实现的，所有事务进行的修改都会先记录到这个回滚日志中，然后在对数据库中的对应行进行写入。



这个过程其实非常好理解，为了能够在发生错误时撤销之前的全部操作，肯定是需要将之前的操作都记录下来的，这样在发生错误时才可以回滚。

回滚日志除了能够在发生错误或者用户执行 ROLLBACK 时提供回滚相关的信息，它还能够在整个系统发生崩溃、数据库进程直接被杀死后，当用户再次启动数据库进程时，还能够立刻通过查询回滚日志将之前未完成的事务进行回滚，这也就需要回滚日志必须先于数据持久化到磁盘上，是我们需要先写日志后写数据库的主要原因。

回滚日志并不能将数据库物理地恢复到执行语句或者事务之前的样子；它是逻辑日志，当回滚日志被使用时，它只会按照日志**逻辑地**将数据库中的修改撤销掉看，可以**理解**为，我们在事务中使用的每一条 INSERT 都对应了一条 DELETE，每一条 UPDATE 也都对应一条相反的 UPDATE 语句。

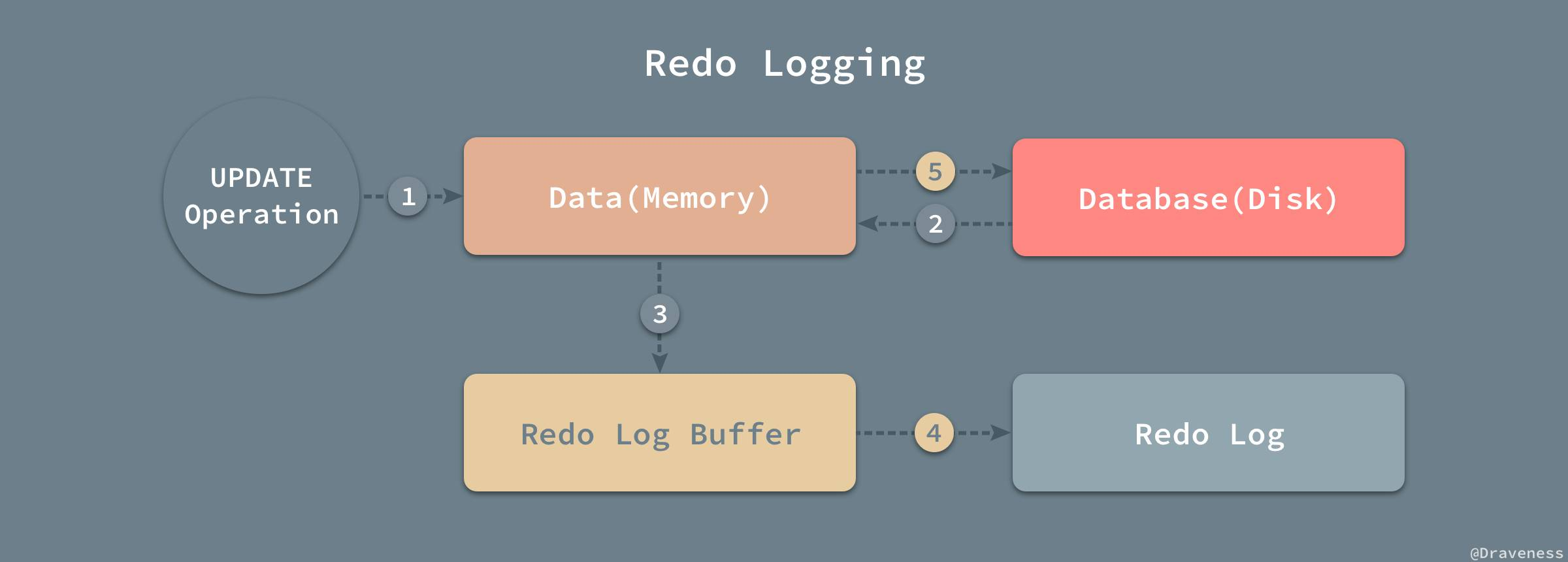


**2.2 重做日志**

作用：

　　确保事务的持久性。防止在发生故障的时间点，尚有脏页未写入磁盘，在重启mysql服务的时候，根据redo log进行重做，从而达到事务的持久性这一特性。

重做日志由两部分组成，一是内存中的重做日志缓冲区，因为重做日志缓冲区在内存中，所以它是易失的，另一个就是在磁盘上的重做日志文件，它是持久的。



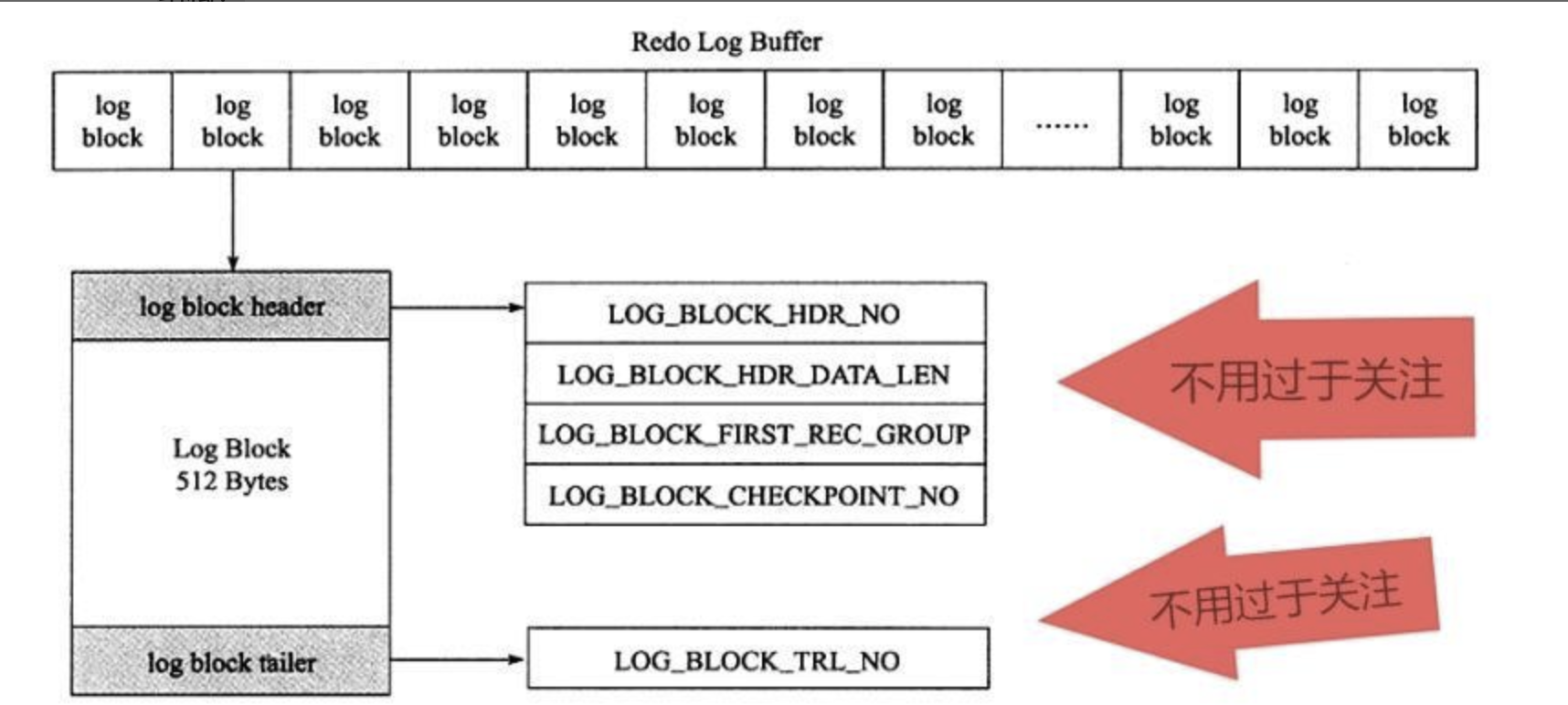
当我们在一个事务中尝试对数据进行修改时，它会先将数据从磁盘读入内存，并更新内存中缓存的数据，然后生成一条重做日志并写入重做日志缓存，当事务真正提交时，MySQL 会将重做日志缓存中的内容刷新到重做日志文件，再将内存中的数据更新到磁盘上，图中的第 4、5 步就是在事务提交时执行的。

除了所有对数据库的修改会产生重做日志，回滚日志也是需要持久存储的

存储：

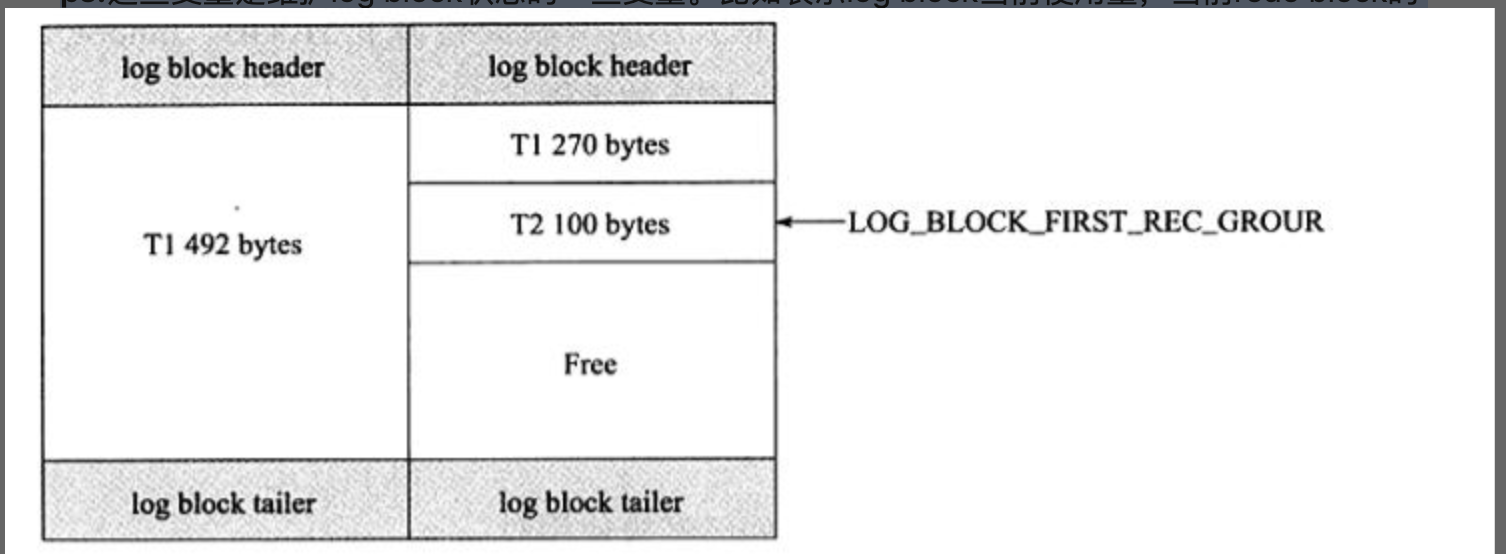
在InnoDB存储引擎中，重做日志都是以512字节为单位进行存储的，这意味着重做日志缓存、重做日志文件块都是以块（block）的方式进行保存的，称为重做日志块(redo log block)。每块的大小512字节。由于重做日志块的大小和磁盘扇区大小一样，都是512字节，因此重做日志的写入可以保证原子性，不需要double write技术。

每个重做日志块的内容快除了日志记录本身之外，还由日志块头(log block header)及日志块尾(log block tailer)两部分组成。重做日志头一共占用12字节，重做日志尾占用8字节。这两部分是固定的。故每个重做日志块实际可以存储的大小为492字节(512-12-8)，如下图显示重做日志块缓存的结构：



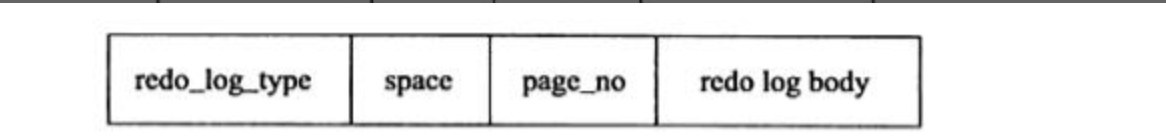
**ps:**这些变量是维护log block状态的一些变量。比如表示log block当前使用量，当前redo block的第一个redo log开始位置等等。举个例子吧：

事务T1的重做日志1占用762字节，事务T2的重做日志占用100字节，。由于每个log block实际只能保存492字节，因此其在log buffer的情况应该如下图所示：



实现这个功能就是靠log block的头部的字段来实现的。

重做日志块中出去header和tailer的内容就是具体的redo log了。不同的数据库操作会有对应的重做日志格式。此外，由于InnoDB存储引擎的存储管理是基于页的，故其重做日志格式也是基于页的。虽然有着不同的重做日志格式，但他们有着通用的头部格式，如图：



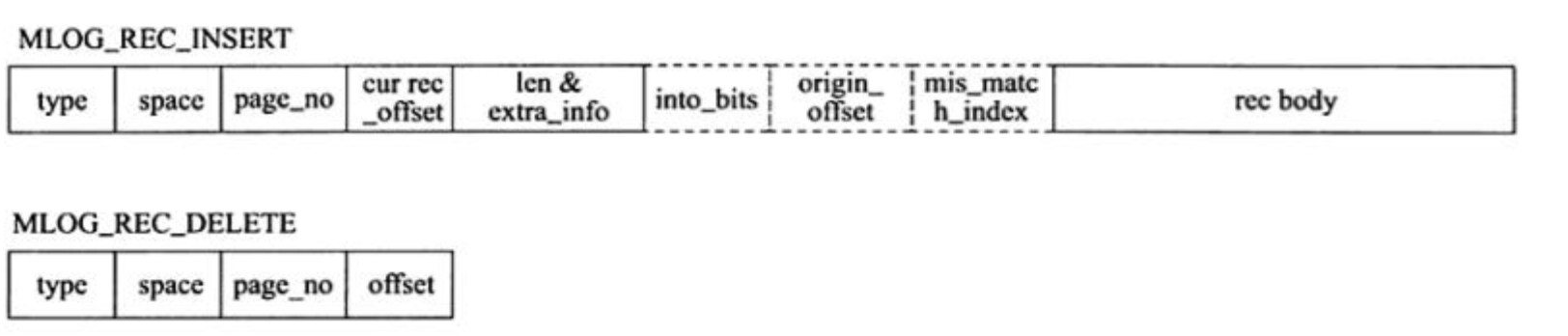
通用的头部格式由一下3部分组成

**redo\_log\_type：** 重做日志类型

**space:**表空间ID

**page\_no** 页的偏移量即页的位置

之后是redo log body ，根据重做日志类型的不同，会有不同的存储内容，例如，对于页上记录的插入和删除操作，分别对应的如图的格式（同样，不要细扣每一个字段的含义，这不是我们要抓的重点）：



大体上的redo log结构介绍完了。在说从redo log file恢复之前，还要说一个LSN的概念，LSN是Log Sequence Number的缩写，其代表的是**日志序列号**，在InnoDB存储引擎中，LSN占用8个字节，并且单调递增。

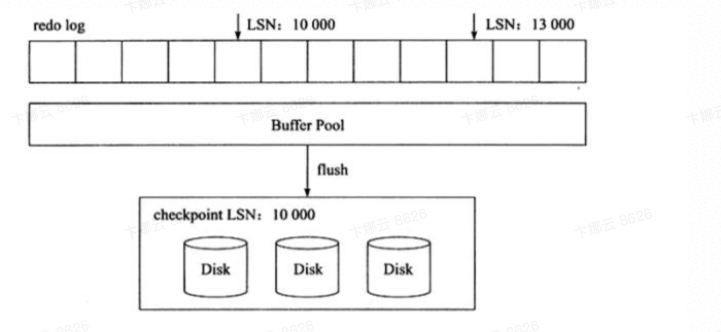
**LSN表示事务写入重做日志字节的总量**。例如当前重做日志的LSN为1000，有一个事务T1写入了100字节的重做日志，那么LSN就变成1100，若又有事务T2写入200字节的重做日志，那么LSN就变为1300。

LSN不仅记录在重做日志中，还存在每个页中，在每个页的头部，有一个值FIL\_PAGE\_LSN，记录了该页的LSN，在页中，**LSN表示该页最后刷新时LSN的大小**。因为重做日志记录的是每个页的日志，因此页中的LSN可以判断页是否需要进行恢复操作。例如，页P1的LSN为10000，而数据库启动时，InnoDB检测到写入重做日志中的LSN为13000，并且事务已经提交，那么数据库需要进行恢复操作。将重做日志应用到P1页中，同样的，对于重做日志中LSN小于P1页的LSN，不需要进行重做，因为**P1页中的LSN表示已经被刷新到该位置**，在此位置之前的内容已经被成功的处理了。

接下来就是恢复操作了：

InnoDB存储引擎在启动时不管上次数据运行是否正常关闭，都会尝试进行恢复操作，因为重做日志记录的是物理日志（不要纠结这个），因此恢复的速度比逻辑日志，如二进制日志要快的多，于此同时，InnoDB存储引擎自身也对恢复进行了一定程度的优化，如顺序读取及并行应用重做日志，这样可以进一步提高数据库恢复的速度

由于**checkpoint表示已经刷新到磁盘页上的LSN**，因此在恢复过程中仅需恢复checkpoint开始的日志部分。对于图中的例子，当数据库在checkpoint的LSN为10 000时发生宕机，恢复操作仅恢复LSN 10000~13000范围内的日志。



物理日志

举个例子，对于Insert操作，物理日志记录的是每个页的变化：

若执行SQL语句：

INSERT INTO t SELECT 1,2;

其记录的重做日志大致类似这个样子：

page(2,3),offset 32,value 1,2

回滚日志（undo log）和重做日志（redo log）；在数据库系统中，事务的原子性和持久性是由事务日志（transaction log）保证的，在实现时也就是上面提到的两种日志，前者用于对事务的影响进行撤销，后者在错误处理时对已经提交的事务进行重做，它们能保证两点：

1. 发生错误或者需要回滚的事务能够成功回滚（原子性）；

2. 在事务提交后，数据没来得及写会磁盘就宕机时，在下次重新启动后能够成功恢复数据（持久性）；

**2.3 隔离性**

**数据库为何需要锁？**

在多用户的应用场景中，为了确保多事务并发执行时数据的完整性，数据库要求每个事务均满足ACID。其中，锁是数据库提供事务功能的关键所在。

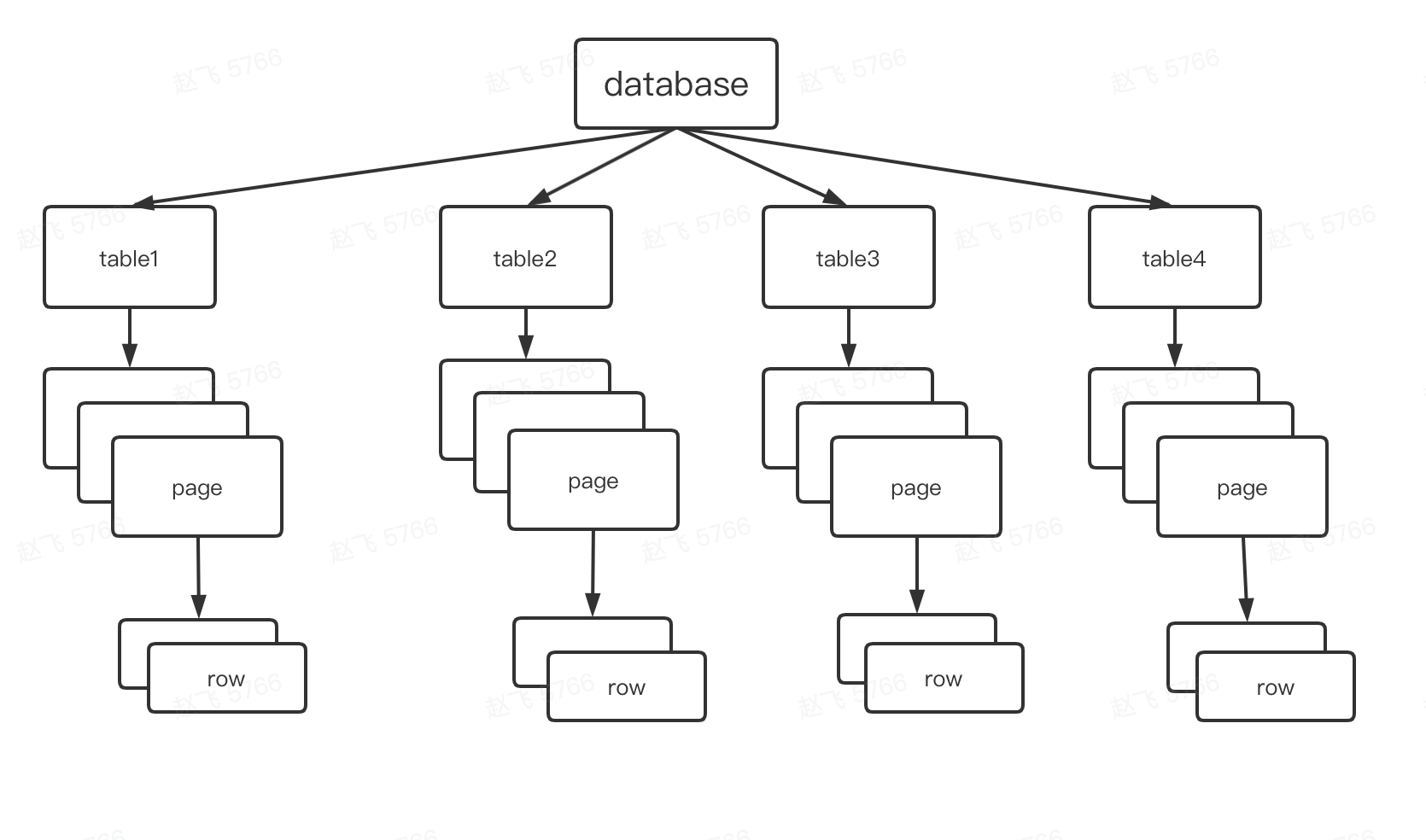
当事务对数据施加锁时，所有其他要访问数据的事务均需要等待，直到锁被释放，而在等待期间不同的隔离级别对于施加了锁的数据有不同可见性。

**如何设计数据库锁？**

前面是几句概念的废话，接下来让我们思考下该如何实现锁？

**锁的粒度：**

我们都知道锁是用来保证数据并发操作安全的，那么首先我们看一下数据库中数据的存储,以inndodb为例：



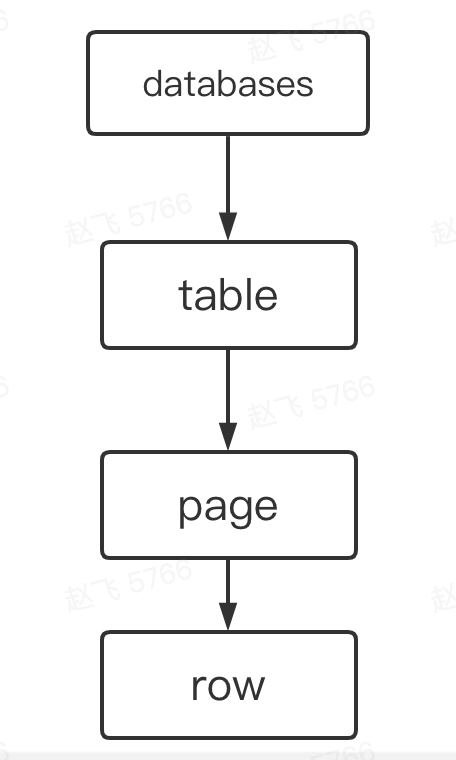
最简单粗暴的形式可能就是database层级上进行加锁，即全局锁。

让整个库处于只读状态的时候，使用这个命令，之后其他线程的以下语句会被阻塞：数据更新语句（数据的增删改）、数据定义语句（包括建表、修改表结构等）和更新类事务的提交语句。最典型的应用场景：**全局锁的典型使用场景是，做全库逻辑备份。**但是，如果只有全局锁确定也是很明显的，即锁的粒度太大，即并发性会很低。

**全局锁之所以并发很低就是由于锁的粒度过大的原因**，所以我们需将锁的粒度进行更细维度的划分。可以在table级别加锁，如表锁；在page页面上加锁page锁；在每row上加锁row锁。

**总结：**不同的存储引擎支持的锁的粒度是不一样的，如**MyISAM** 仅支持表锁不支持行锁和page锁，和支持page锁以及row行锁的innodb相比，并发度不高是其被代替主要原因。

数据库中锁存在存储结构，锁的层次从最高层数据库开始，通过表和页向下到达最底层的行。



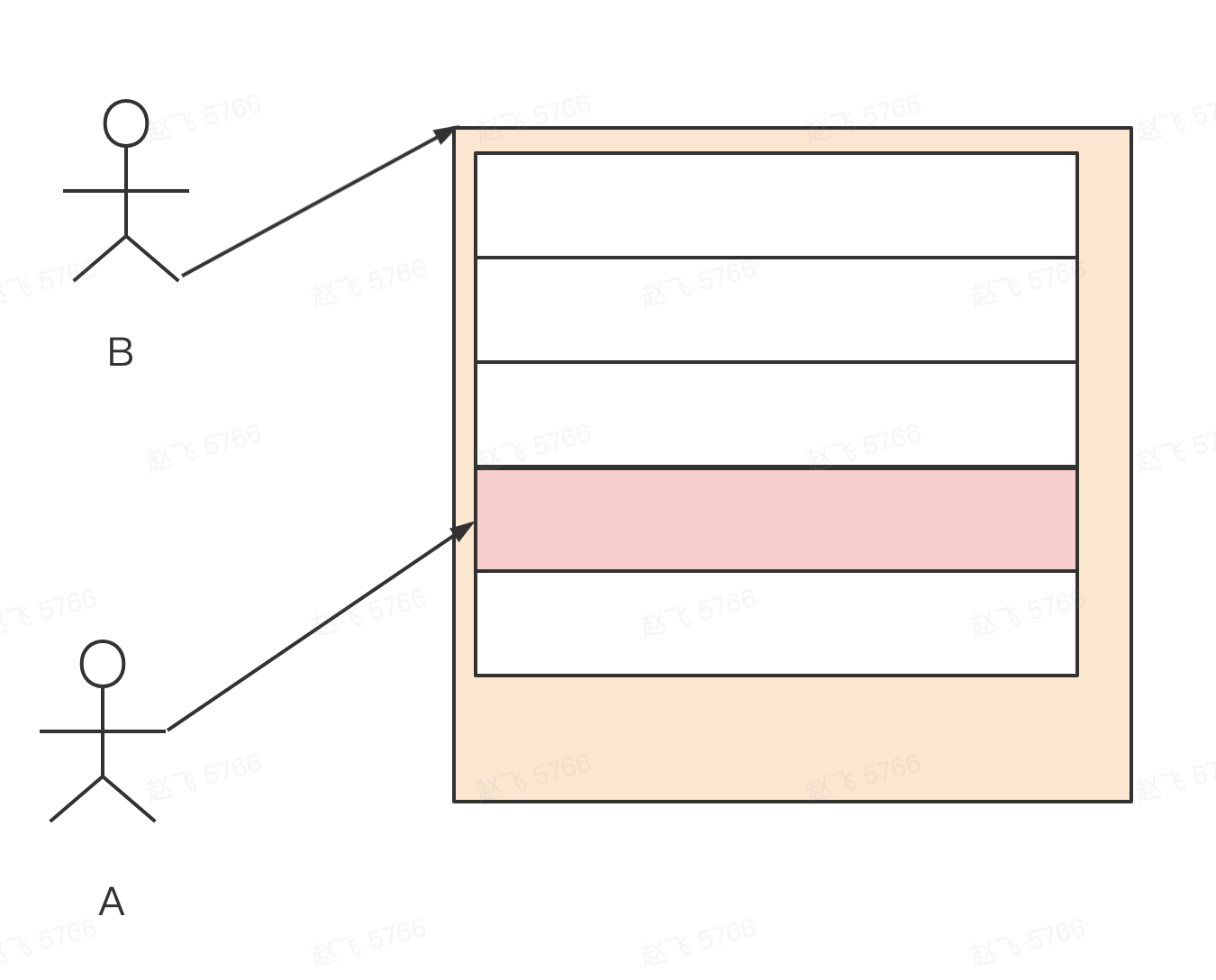
**引入多粒度的锁是否存在问题呢？**

**首先，使用相同粒度的锁是不会存在问题的，即两个事务均使用表锁或者行锁。**

那么不同粒度的锁混合使用时是否有问题呢？

**试想这样一个场景：**事务A正在修改表A中某一行的数据，事务B想给表A加个字段。

分析：此时由于事务A由于需要修改表A中的某一行，则会使用row锁将改行锁定。那么对于事务B,由于需要修改表A的结构则需要获取表A的写锁。如果事务AB获取锁均会成功，则按照表锁的定义获取表锁则可以修改表中的任意行，由于事务AB均可以修改事务A要修改的行，则数据的正确性无法进行保证。



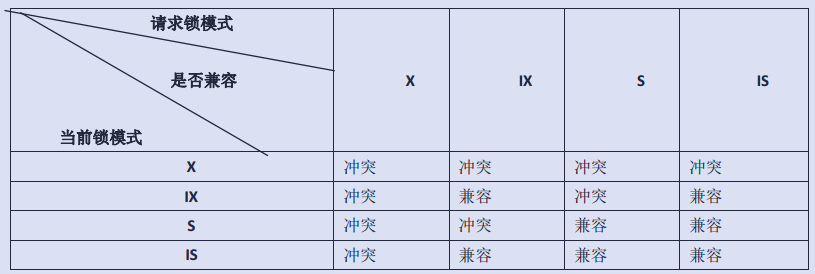
**那么如何解决不同事务使用不同粒度锁这个问题呢？**

很显然锁是需要定义优先级的，即事务获取行写锁的同时必须获取表写锁，即禁止其他事务获取表写锁。 如何解决这个问题：此时引入一个新的概念意向锁，接下来分析意向锁是如何解决这个问题的。

* 共享锁（S）：允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁。
* 排他锁（X）：允许获得排他锁的事务更新数据，阻止其他事务取得相同数据集的共享读锁和排他写锁。
* 意向共享锁（IS）：事务在给一个数据行加共享锁前必须先取得该表的 IS 锁。
* 意向排他锁（IX）：事务在给一个数据行加排他锁前必须先取得该表的 IX 锁。

**加锁规则：**低粒度的资源加锁高粒度自动加上意向锁，同时是否加锁成功遵循以下规定。

不同事务间请求锁兼容方式：



**分析引入意向锁后场景分析：**事务A想修改某行，需要获取该行的排他锁（X）。则会先获取所在page的意向排他锁（IX）所在表的意向排他锁（IX）。此时假定没有其他事务占有锁，则事务A获取成功。接下来事务B在该表添加字段，需要获取该表的排他锁（X），此时由于事务A拥有该表的意向排他锁(IX)，即处于请求锁模式（X）和当前锁模式（IX）不能兼容，则事务B不能获取表的排他锁（X）,此时无法添加。

**行锁的实现**

**只锁住一行是否可以？这里可能不需要举栗子了，会引起幻读，为了解决幻读需要引入间隙锁，即gap锁。**

行锁:记录锁，gap 锁以及next-key lock（gap锁+记录锁）

**总结**：***多事务并发修改时为了保证数据安全-》解决方案加锁-》针对table加锁-》锁的粒度太大并发低-》划分细粒度的锁后-》混合使用不同粒度的锁有问题-》引入意向锁-》补充只针对某行加锁有幻读问题-》引入gap锁***

貌似已经结束了，但是实际还没有。。。。

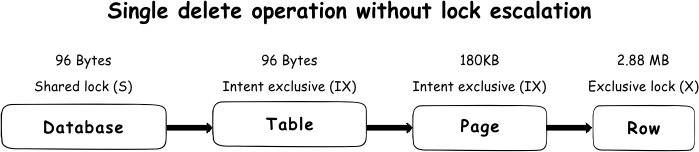
作为一个有追求的程序员需要不断的进行优化。

**优化：**

问题1，行锁占有内存问题。首先，明确锁是占用内存滴，每个锁96字节。

**考虑下这样的应用场景：** 在30000行的表上执行删除操作，其中每数据行大小为500字节。

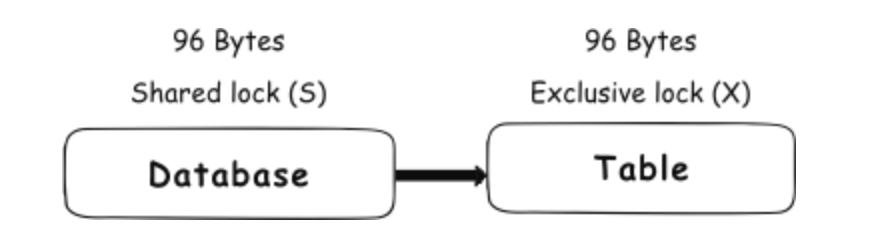
分析：按照我们前面的知识可知，将数据库上加一个共享锁（S）,表上加一个意向排他锁（IX）,页面上1875个意向排他锁（IX）（16\*500\*1875=30000每个page8kb）和行本身30000个排他锁（X）总计31877个锁。每个锁96字节，则每执行一次删除需要3MB内存的操作。



从上图可以看出其中行锁占大部分内存，在数据量越多时则行锁占的越多。但实际中，好像我们的执行删除时貌似并没有这么高的内存，那又是因为什么呢？

**一种解决方案：锁升级**

此时，是通过提高锁的粒度，降低并发来换取内存。即此时数据库会升级到表级排他锁（X）



通过对比可知仅需要192bytes，和3MB可以忽略不计。这又是balance思想在计算机中的一次使用。

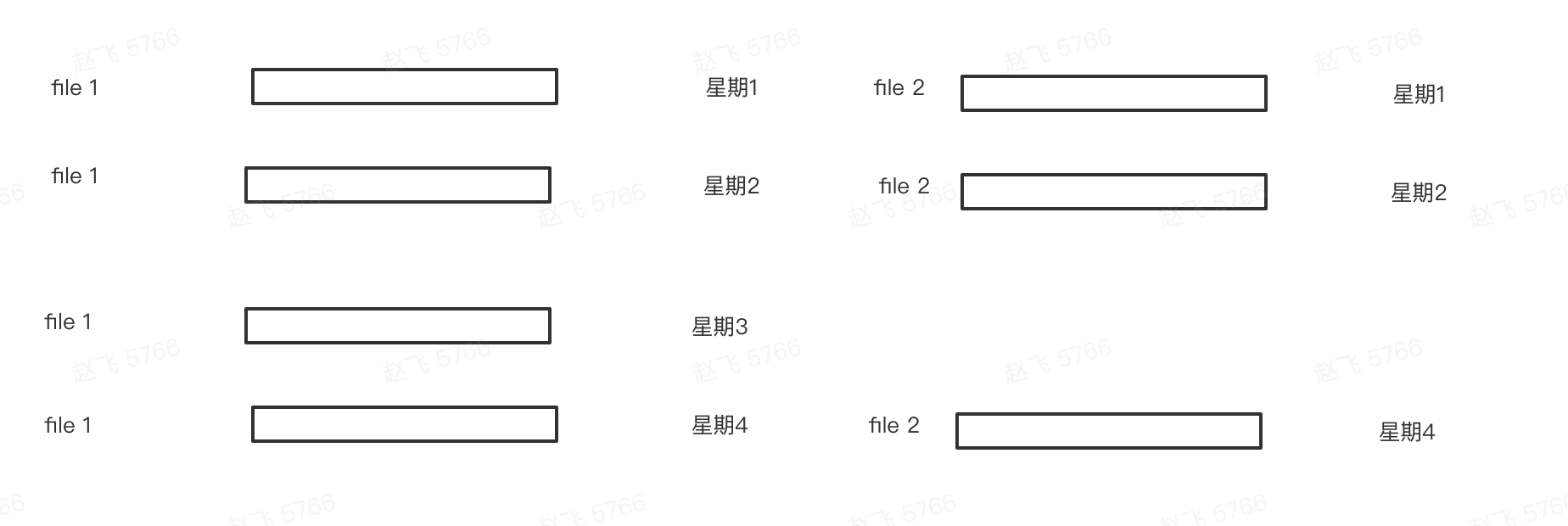
**另一种解决方案：MVCC多版本时间戳。**

把时间戳和多版本控制结合就形成了多版本时间戳机制，多版本时间戳机制的一大好处在于，**一个读取数据的事务永远不会失败也不需要等待**。在一个读多写少的场景下，相比于先前介绍的两种机制，会有很好的性能提升。

先来理解版本数据。我们可能想到版本数据库（HDFS），其实还有一种就是我们每天使用的git的底层存储。

**git存储浅析多版本数据存储**

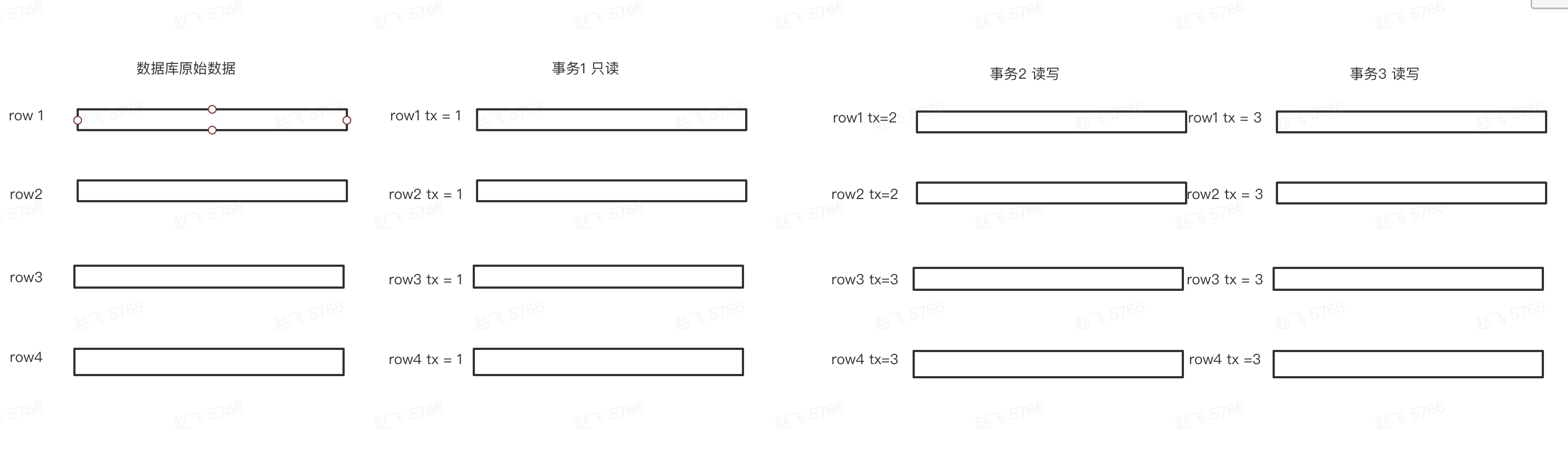
由于commit好是时间压缩后不直观这里直接使用星期天代替。假定我们有四个文件，file1, file2



其中file1,星期1，2，3，4均进行了修改，file2星期1，2，4进行了修改。

获取每个commit的规则。每个文件有当前commit的直接获取，没有获取最近的。周四file1.file2均有提交，所以全部获取，获取周三的，fiel2没有，则fiel2获取星期2的。

接下来看下多版本数据是如何在mysql中进行使用的。



如果事务1进行只读不进行修改，是否我们在事务开启时把当前数据打一个快照，后面这个事务里面的所以读操作都是基于这个时间戳。其他事务读写均不影响该事务，是不是就可以提供该事务的并发了，因为木有锁。

此时会引入另外一个问题，如果**需要在事务里读取最新的怎么办？修改数据怎么办？即事务2和3** 只引入多版本时间戳就无法解决了，需要前面我们将的锁结合起来。

这里需要引入两个概念**，快照读和当前读。**

**快照读，**即每个事务开启时，设置的时间戳如果是快照读，只会读取基于这个时间戳。

**当前读：**读取目前已经其他事务已经提交了的数据。注意！！！当前读需要加锁，否则两次读数据不一致。

哪些会引起当前读：insert，update，delete， select ... for update， select ... lock in share mode

总结：引入mvcc进行读写分离-》但是mvcc直接写无法保证数据安全（具体参考多个分支提交冲突）-》当需要当前读时进行加锁-》即**mvcc+ 锁这也是目前innodb隔离性是实现。**

接下来总结下innodb四种事务隔离级别：

Read uncommited: 无mvcc和锁

Read commited: mvcc + 行锁

Read repeated: mvcc + 行锁+ gap锁

Serializable: 所有行均加锁。

**总结：**

把对数据库数据的操作划分称为“事务”的基本原子单位，一个事务要么全做，要么全不做（All-or-Nothing 原则）；用户在对数据库发出操作请求时，需要对有关数据进行“加锁”，防止不同用户之间相互干扰；在事务运行过程中，采用“日志”记录事务的运行状态，以便发生故障时进行恢复；对数据库的任何更新都采用“两阶段提交”策略。以上方法称为数据库的事务处理技术。

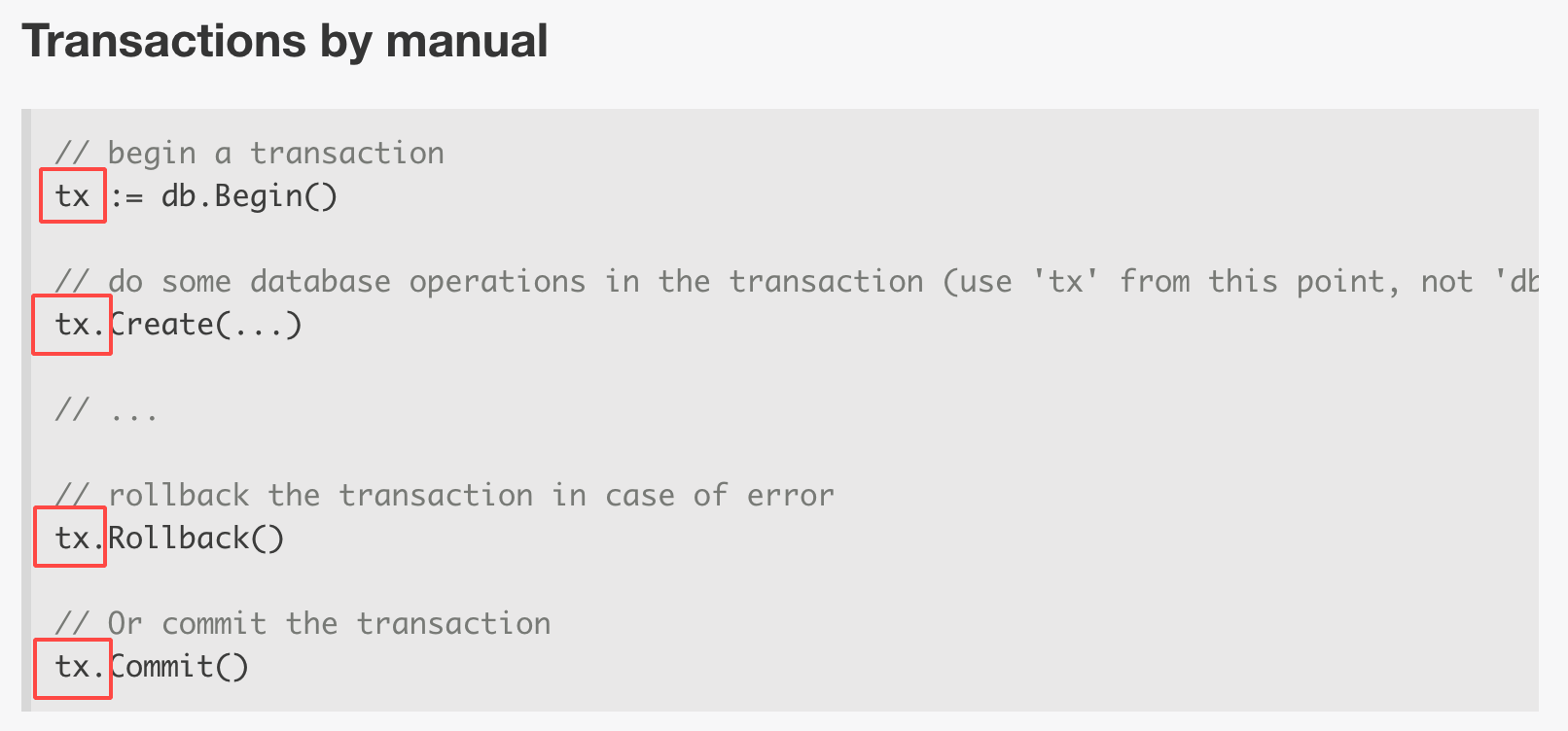
**实践与注意事项**

**3.1 实践**

*以下 MySQL 相关内容仅针对 GORM V1*

MySQL：

*参考 https://gorm.io/docs/transactions.html*



错误示例：

|  |
| --- |
| Go func (mysql \*Mysql) ExecuteTransaction(action func() error) error {  tx := mysql.DBW.Begin()   if tx.Error != nil {  return tx.Error  }   if err := action(); err != nil {  if e := tx.Rollback().Error; e != nil {  logs.Error("transaction rollback err:%v", e)  }  logs.Info("rollback transaction")  return err  }  commit := tx.Commit()  logs.Info("commit transaction")  return commit.Error } |

正确示例：

|  |
| --- |
| Go func ExecuteTransaction(action func(db \*gorm.DB) error) error {  tx := GetMySQL().DBW.Begin()  defer func() {  if r := recover(); r != nil {  tx.Rollback()  }  }()  if tx.Error != nil {  return tx.Error  }  if err := action(tx); err != nil {  if e := tx.Rollback().Error; e != nil {  logs.Error("transaction rollback err:%v", e)  }  logs.Info("rollback transaction")  return err  }  commit := tx.Commit()  logs.Info("commit transaction")  return commit.Error } |

Mongo：

*参考：https://docs.mongodb.com/manual/core/transactions/*

|  |
| --- |
| Go session, err := conn.MongoClient.StartSession() if err != nil {  return err }  if err = session.StartTransaction(); err != nil {  return err }  if err = mongo.WithSession(ctx, session, func(sessionContext mongo.SessionContext) error {  defer recover2.Recover(logs.Errorf, func() {  session.AbortTransaction(ctx)  })   action(sessionContext) // Your action here   if err = session.CommitTransaction(ctx); err != nil {  return err  }  return nil }); err != nil {  return err }  session.EndSession(ctx) |

**3.2 注意事项**

**3.2.1 死锁**

造成死锁的原因：

|  |
| --- |
| Ruby # transaction 1 start transaction; update bits\_work\_item\_value set field\_value='test field\_2' where field\_key='field\_1' and work\_item\_id=979; select *sleep*(10); # 模拟业务线程处理过程 update bits\_work\_item\_value set field\_value='test name' where field\_key='name' and work\_item\_id=979; commit ;  # transaction 2 start transaction; update bits\_work\_item\_value set field\_value='test name' where field\_key='name' and work\_item\_id=979; select *sleep*(10); update bits\_work\_item\_value set field\_value='test field\_2' where field\_key='field\_1' and work\_item\_id=979; commit ; |

同时执行以上两个事务会造成死锁，原因在于两个事务在执行更新字段过程中互相给对方所需要更新的字段上锁。

**预防的办法：**

**代码层面**

* 操作数据时按照约定顺序执行
* 事务执行的时候如果遇到死锁，其中某事务rollback后等待一个随机时间再执行

**数据库引擎InnoDB应对死锁的办法**

InnoDB存储引擎并不会回滚大部分的错误异常，但是死锁除外。发现死锁后，InnoDB存储引擎会马上回滚一个事务，这点是需要注意的。因此如果在应用程序中捕获了1213这个错误，其实并不需要对其进行回滚。

|  |
| --- |
| Go func ExecuteTransactionWithRetry(action func(db \*gorm.DB) error) error {  tx := GetMySQL().DBW.Begin()  defer func() {  if r := recover(); r != nil {  tx.Rollback()  }  }()  if tx.Error != nil {  return tx.Error  }  if err := action(tx); err != nil {  if e, ok := err.(\*md.MySQLError); ok {  if e.Number == 1213 {  if ei := action(tx); ei != nil {  logs.Error("retry failed with err:%v", ei)  }  }  } else {  if e := tx.Rollback().Error; e != nil {  logs.Error("transaction rollback err:%v", e)  }  logs.Info("rollback transaction")  return err  }  }  commit := tx.Commit()  logs.Info("commit transaction")  return commit.Error } |

**3.2.2 循环中提交事务**

问题一：当发生错误时，数据库会停留在一个未知的位置

例如，用户需要插入10000条记录，但是在插入5000条时，发生了错误，这时前5000条记录已经存放在数据库中，那应该怎么处理呢？ 因为事务是一个个提交的。

问题二：每一次提交都要做一个redo日志，性能很差。

例如，用户需要插入10000条记录，那就要写10000次重做日志文件

解决思路：可以将循环过程放在事务里面，这样所有的操作最后只会被提交一次。

**3.2.3 使用自动提交**

MySQL数据库默认设置使用自动提交（autocommit），可以使用START TRANSACTION，BEGIN来显式地开启一个事务。在编写应用程序开发时，最好把事务的控制权限交给开发人员，即在程序端进行事务的开始和结束。

*参考*

1. 《事务处理：概念与技术》
2. 《MySQL 技术内幕：InnoDB 存储引擎》
3. 《高性能 MySQL》
4. 《DDIA》
5. https://gorm.io
6. https://www.infoq.cn/article/gaOh3me9PmJBiQFD2j15

后续补充：

1、大家想要补充的

2、Spring transaction refers to: https://segmentfault.com/a/1190000020386113

3、 undolog实现mvc

*Q&A*

1. 嵌套事务需要父事务提交后子事务才能生效，等同于save point？Q:李君奇 A:赵飞，张华
2. 控制策略补充
3. undo log什么时候写入？Q:杨晓军 A:赵飞，卞娜云
4. 参数控制刷盘频率
5. 锁粒度可以更小么？Q:杨晓军 A:赵飞
6. 为什么是加的是意向锁IX锁，而不是X锁? Q:邵黎明 A:赵飞

*反馈*

**[This type of content is unavailable for download at the moment.]**