TX

2021年11月22日　　8:16

AT
TCC
SAGA
XA

分布式从最后开始。

==========

==========

https://www.jianshu.com/p/ad43961f20c6

MVCC　Multi-Version Concurrency Control
多版本并发控制。

MVCC使得大部分支持行锁的事务引擎，不再单纯地使用 行锁来进行数据库的并发控制，而是将数据库的行锁 和 行的多个版本结合起来，只需要很小的开销，就以实现非锁定读，从而大大提高数据库的并发性能

如果有人读取数据库的 同时 其他人在写入数据，则 读取数据库的人 可能看到 ″半写″ 或不一致的数据。
最简单的方法，通过加锁，让 所有的 读者 等待写者完成，但是效率很差。
MVCC使用了 不同的手段： 每个连接到 数据库的 读者，在某个瞬间看到的是 数据库的一个快照，写者写操作 造成的变化 在 写操作完成之前（或数据库事务提交之前）对于其他读者来说 是不可见的。

MVCC就像是接口，各大厂商实现的 机制 不尽相同。可以认为 MVCC 是 行级锁的 一个变种，但是它在很多情况下 避免了 加锁操作，因此开销更低。虽然机制有所不同，但是大都实现了非阻塞的读操作，写操作也只是锁定必要的行。
MVCC会保存某个时间点上的数据快照。这意味着事务可以看到一个一致的数据视图，不管它们需要运行多久。这同时也意味着不同的事务 在同一个时间点 看到的 同一个表的数据 可能是不同的。
前面提到不同的存储引擎的MVCC实现是不同的，典型的有乐观并发控制 和 悲观并发控制。

MVCC实现的读写不阻塞正如其名：多版本并发控制 -- 通过一定机制 生成一个数据请求时

间点的 一致性数据快照，并用这个快照来提供一定级别(语句级或事务级) 的一致性读取。从用户角度来看，就像是数据库提供 同一个数据的多个版本。

MySQL 的 InnoDB 存储引擎实现 MVCC的策略

InnoDB的 MVCC，是通过在每行记录后面保存2个 隐藏的列来实现的。这2个列，一个保存了行的创建时间，一个保存行的过期时间(或删除时间)。当然存储的并不是 实际的时间值，而是 系统版本号。每开始一个新事务，系统版本号会自动递增。事务开始时刻的 系统版本号会作为 事务版本号，用来和 查询到的 每行记录的版本号进行对比。

Select：

InnoDB 会根据以下2个条件 检查每行记录：

1. InnoDB 只查询 版本号 <= 当前事务版本号的 数据行。这样可以保证，事务读取的行，要么是 事务开始前就已经存在的，要么是 事务自身插入 或者 修改过的。
2. 行的删除版本号 要么未定义，要么大于当前事务版本号。这可以确保事务读取到的行，在事务开始之前 没有被删除。

Insert：

InnoDB 为新插入的行 保存 当前系统版本号作为 行版本号。

Delete：

InnoDB 为删除的每一行保存当前系统版本号作为 行删除标识。

Update：

InnoDB 为插入一行新记录，保存当前系统版本号作为行版本号，同时保存当前系统版本号到原来的行 作为删除标识 (这只是理论，实际上innoDB 是通过 undo log 来备份旧记录的)

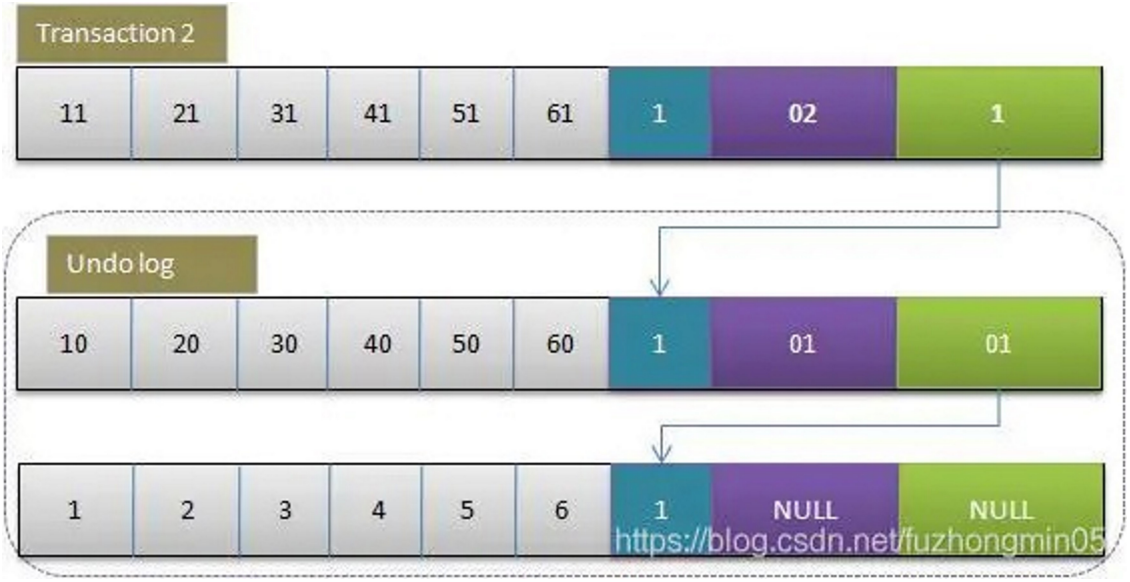
innoDB存储的最基本row中包含一些额外的存储信息
DATA_TRX_ID、DATA_ROLL_PTR、DB_ROW_ID、DELETE BIT。

DATA_TRX_ID：标记了最新更新这条行数据的 transaction id。
DATA_ROLL_PTR：指向了当前记录的 rollback segment 的 undo log 记录，找之前的版本的数据就是通过这个指针。
DB_ROW_ID：当由 innoDB 自动产生 聚集索引时，聚集索引包括这个 DB_ROW_ID 的值，否则聚集索引中不包括这个值，这个用于索引当中
DELETE BIT：标识该记录是否被删除，这里不是真正的删除，而是标识，真正的删除是在 commit 时。

当事务1 修改某行的值时：

1. 用排它锁锁定该行
2. 记录redo log
3. 把该行修改前的值 复制到 undo log。
4. 修改当前行的值，填写事务编号，使得 回滚指针 指向undo log 中修改前的行。

事务2 修改该行的值时：



。。这图应该是修改完后的。

和事务1 相同，此时 undo log 中存在 2 行记录，并且通过回滚指针连在一起。
因此，如果undo log 一直不删除，则会通过当前记录的回滚指针 回溯到 该行创建时的初始内容，所幸的是，在innoDB中 存在 purge 线程，它会查询那些 比 现在最老的 活动事务 还早 的 undo log，并删除它们，从而保证 undo log 文件不会无限 增长。
。。undo的时候 还会判断的吧。 不然 purge 2次中间 就。。
。。还有 最老的活动，是指 这行上的，还是整个db？。。 整个db的话，长事务 就可能影响其他事务的性能了
。。还有。。本来想说： 一个事务执行的时候， 有一个 短事务 比它慢开始，但是更快结束， 那么 事务 完全感知不到 那个短事务的。。 不过 这里上面的 第一点说到了 排它锁。。 所以 不可能有 2个 写事务？ 那这个并发性能真的。。当然 乐观锁 应该好一点。不，看情况，写频繁 就 悲观锁(排它锁)。

当事务正常提交时，只需要更改事务状态 为 commit 即可，不需要做其他额外的工作，而 Rollback 则稍微复杂点，需要根据当前 回滚指针 从 undo log中找出 事务 修改前的 版本并恢复。如果事务影响的 行非常多，回滚则可能 变得低效，根据经验值，事务行数 在 1000-10000之间，innoDB的效率还是 非常高的。 很显然，innoDB 是一个 commit效率 比 rollback 高的 存储引擎。

更新(update，insert，delete)是一个事务过程，在innoDB中，查询也是一个事务， 只读事务。
当 读 写 事务并发访问同一个行数据时，能读到什么内容，依赖于事务级别：
read_uncommitted: 读未提交，读事务直接读取 主记录，无论更新事务 是否完成
read_committed: 读已提交，读事务 每次都读取 距离 undo log 最近的那个版本，因此，2次对同一字段的读可能读到 不同的数据(幻读)，但能保证每次都读到 最新的数据。
repeatable_read: 每次都读取指定的版本，不会幻读，但是读到的可能不是最新的。
serializable: 锁表，读写相互阻塞，使用很少。

MVCC只在 repeatable_read，read_committed 2个隔离级别下工作。其他2个隔离级别 和 MVCC 不兼容，read_uncommitted 总是读取 最新的 数据行，而不是 符合 当前事务版本的数据行， 而serializable 会对 所有的读取的行 都加锁。

读事务 一般由 select 语句触发，在innodb中保证其非阻塞，但带 for update 的 select 除外，它会对 行 加 排它锁，等待更新事务完成后 读取其 最新的内容。

InnoDB实现的 MVCC 有何特殊性
上述更新前 建立 undo log，根据各种策略 读取时 非阻塞的就是 MVCC，undo log 中的行就是 MVCC中的 多版本，这可能和我们所理解的 MVCC有较大出入，一般 我们认为 MVCC有下面的几个特点：

1. 每行数据都存在一个 版本，每次数据更新时 都更新该版本
2. 修改时，copy出当前版本 随意修改，各个事务间 无干扰
3. 保存时 比较 版本号，如果成功则 commit 并覆盖原记录，失败则放弃copy(rollback)

。。这个不是乐观锁？那之前说 加排它锁 是什么？。。。下面。。。

就是每行都有版本号，保存时，根据版本号决定是否成功，听起来有 乐观锁的 味道，而InnoDB的 实现方式是：

1. 事务以排它锁的形式修改原始数据
2. 修改时copy出当前版本随意修改，各个事务之间 无干扰
3. 保存时，比较版本号，如果成功则 commit 并覆盖原记录，失败则放弃 copy(rollback)

。。排他了，还怎么干扰。。。。ok， 历史版本， 事务1开始， 事务2开始 并且排他锁 修改了数据 commit， 事务1 读取 数据， 有历史版本，所以 事务1 读取不到 事务2的 东西。
。。只不过，这种只能用来读啊， 如果事务1 想写的话， 它和 事务2冲突的， 这种 是什么？ 还是乐观锁啊。

两者最本质的区别是，当修改数据时 是否要排他锁， 如果锁定了 还算不算是 MVCC。

MVCC可以保证 不阻塞地 读到一致的数据，但是 mvcc理论 没有对 实现细节作出 约束，因此，不同数据库的 语义有所不同：
postgres 对写操作 也是 乐观并发控制；在表中保存同一行数据记录的 多个不同版本，每次写操作都是 创建，而回避更新；在事务提交时，按版本号检查当前事务提交的数据是否存在写冲突，有就抛出异常，回滚事务。
InnoDB只对读 无锁， 写操作 仍是 上锁的 悲观并发控制，这意味着，InnoDB中只会看到因为 死锁 和 不变性约束 而回滚， 而看不到 因为写冲突而回滚；不像 postgres 那样 对数据修改 在表中创建新记录， 而是每行记录 在表中只有一份，在更新数据时 上行锁，同时将 旧版数据 写入到 undo log； 表和undo log 中 行数据 都记录着 事务id，在检索时根据事务 隔离级别 去 读取行数据。 可见MVCC 中的 写操作 仍然按 悲观并发控制 实现；

快照读和当前读
快照读，就是读取数据的时候会根据一定的规则 读取 事务可见版本的 数据，不用加锁
当前读，读取最新版本，并且对读取的记录加锁，保证其他事务 不会并发修改这条数据，避免出现安全问题。

使用当前读的场景：
select…lock in share mode (共享读锁)
select…for update
update
delete
insert

使用快照读：
单纯的select操作，不包括上述 select … lock in share mode、select … for update

总结
所谓的MVCC（Multi-Version Concurrency Control 多版本并发控制）指的就是在使用读已提交（READ COMMITTD）、可重复读（REPEATABLE READ）这两种隔离级别的事务在执行普通的SELECT操作时访问记录的版本链的过程，这样子可以使不同事务的读-写、写-读操作并发执行，从而提升系统性能。

这两个隔离级别的一个很大不同就是：生成ReadView的时机不同，READ COMMITTD在每一次进行普通SELECT操作前都会生成一个ReadView，而REPEATABLE READ只在第一次进行普通SELECT操作前生成一个ReadView，数据的可重复读其实就是ReadView的重复使用。

InnoDB通过为每一行记录添加两个额外的隐藏的值来实现MVCC，这两个值一个记录这行 数据何时被创建，另外一个记录这行数据何时过期（或者被删除）。但是InnoDB并不存储这些事件发生时的实际时间，相反它只存储这些事件发生时的系统版本号。这是一个随着事务的创建而不断增长的数字。每个事务在事务开始时会记录它自己的系统版本号。每个查询必须去检查每行数据的版本号与事务的版本号是否相同。

这种额外的记录所带来的结果就是对于大多数查询来说根本就不需要获得一个锁。
他们只是简单地以最快的速度来读取数据，确保只选择符合条件的行。这个方案的缺点在于存储引擎必须为每一行存储更多的数据，做更多的检查工作，处理更多的善后操作。
使用MVCC多版本并发控制比锁定模型的主要优点是在MVCC里， 对检索（读）数据的锁要求与写数据的锁要求不冲突， 所以读不会阻塞写，而写也从不阻塞读。
在数据库里也有表和行级别的锁定机制， 用于给那些无法轻松接受 MVCC 行为的应用。 不过，恰当地使用 MVCC 总会提供比锁更好地性能。

==========

MySQL 间隙锁

间隙锁 是一个 在索引记录 之间的 间隙上的锁。

在InnoDB引擎中，如果操作的是一个 区间的数据，会锁住这个区间所有的记录，即使这个记录不存在，此时，另一个事务 去插入 这个区间的数据，就必须等待 上一个结束。

```
begin;
update mayikt_account set name='mayikt6' where id>18 and id<22;
commit;
```
从id>18 and id <22 上了间隙锁，在没有释放锁的时候 其他的session无法对该段位做操作。

主键索引 或 唯一索引 会使用间隙锁吗？

1. 如果where 条件都命中的情况下，不会发生间隙锁，只会增加记录锁
2. 如果where 条件 部分命中 或 全部没有命中的情况下，则使用 间隙锁。

如果避免 行锁升级为 表锁
InnoDB 的 行锁 是针对 索引 加的锁，不是 针对记录加的锁，并且 该索引不能失效，否则都会从 行锁升级为 表锁

修改的时候，查询条件不是 索引字段，会走全表扫描，全表扫描的时候 会对 每行数据 都加上 行锁，最终形成表锁

```
unlock tables;
```
删除表锁

优化事项
1. 尽可能让所有数据检索 都通过索引来完成，避免无索引行锁 升级为 表锁
2. 尽可能减少索引条件范围，避免间隙锁
3. 尽可能控制事务大小，减少 锁定资源量 和时间长度，涉及事务加锁的sql 尽量 放在事务最后执行。

==========

脏写
事务A覆盖了 其他事务尚未提交的 写入

脏读
事务A 读取了 其他事务尚未提交的写入

读倾斜
事务A在执行中，对某个值的 2个读取，读到不同的值， 也称为 不可重复读

更新丢失
2个事务同时 执行 读-修改-写入，出现了其中一个 覆盖了 另一个事务的写入，但是没有包含 对方 最新值， 导致被覆盖的数据 发生了 丢失更新。

幻读
事务 查询了 某些符合条件的数据，同时 另一个事务 执行写入，改变了先前的查询结果

写倾斜
事务 查询了 数据库，然后根据 返回的结果做出了某些决定，然后修改数据库。 在事务提交时，支持决策的 条件不再成立。
写倾斜是 幻读的一种情况，是由于 读-写 事务冲突导致的 幻读。
写倾斜 也可以看做 一种更广义 的更新丢失问题。 即 如果2个事务读取 同一个对象，然后更新其中一部分：不同的事务更新不同对象，可能发生写倾斜；不同的事务更新同一个对象，可能发生 脏写 或更新丢失。

==========

==========

==========

==========

==========

==========

==========

==========

2PL
2 phase locking

两阶段锁 强调的是 加锁(增长阶段) 和 解锁(缩减阶段) 这2项操作，每个操作 为一个阶段，这就是说 不管同一个事务内 需要在多少个 数据项上 加锁，那么 所有的 加锁操作 都只能在同一个 阶段内完成，在这个阶段内，不允许对 已经加锁的 数据项 进行解锁操作，

即 加锁 和 解锁 操作 不能交叉执行(同一个事务内)。

为了提高并发度，才对锁进行分类，分出共享锁(读锁) 和 排它锁(写锁)，有4种情况：

对数据加 共享锁，则 此读锁不阻塞 其他事务的读取，所以 读读并发。 但是阻塞 其他事务写入，所以是 读写 不可以并发。 读读 和 读写 ，第一个都是 读。

对数据加 排它锁，则其他事务 不可以 读取，写入 这个数据。

共享锁 是允许 向 排它锁 升级的， 排它锁 允许 向共享锁 降级。 升级 和 降级操作，称为 锁转换。 升级 只能发生在 增长阶段，降级 只能发生在 缩减阶段。

所以，两阶段的 含义 是指 在同一个事务内，对所 涉及的所有数据 进行加锁，然后才对所有的数据项解锁。 但 两阶段封锁第一阶段 加共享锁后 影响了其他事务的 写操作，加排它锁后，影响了 其他事务的 读写操作，所以较大地影响了其他事务的运行(如果不操作相同的数据则互不影响)。只有在第二阶段 释放了所有的 数据项上的锁之后，才能运行其他要操作这些数据的 事务。

---

https://cloud.tencent.com/developer/article/2056138?from=article.detail.2056150
精通Java事务编程(1)-深入理解事务

严格的数据存储系统中，可能遇到很多出错的情况：

数据库软件，硬件可能随时失效(包括正在执行写过程当中)
应用程序可能随时崩溃(包括一系列操作的中间某步)
网络中断可能意外切断 db 和应用的连接，或db之间的连接
多个客户端可能同时写入db，导致数据覆盖
客户端可能读到无意义的，部分更新的数据
客户端之间犹豫 边界条件竞争 引入各种奇怪问题

事务一直是简化这些问题的首选机制，事务将 应用的 多个读，写操作 组成一个逻辑单元，即事务中的 读写是个执行的整体。整个事务 要么成功(提交)，要么失败(中止或回滚)。

并非所有应用需要事务，有时可以弱化事务处理 或完全放弃事务 (为了更高的性能和 可用性)。一些安全相关属性 也可能会 避免引入事务

目前几乎所有关系型DB和一些非关系DB都支持事务。大多遵循IBM System R（第一个SQL数据库）在1975年的设计。50年来，尽管一些细节实现变化，但总体思路大同小异。MySQL、PostgreSQL、Oracle 和 SQL Server 等DB中的事务支持与 System R 极为相似。

2000年后，NoSQL普及，目标在关系DB现状上，通过提供新数据模型和内置的复制和分区改进传统的关系模型。然而，事务成了这变革的受害者：新一代DB完全放弃事务或重新定义，即替换为比以前弱得多的保证。

随新型分布式DB炒作，人们普遍认为事务是可扩展性的对立面，大型系统都必须放弃事务以获得更高性能和高可用性。但另一方面，还有一些DB厂商坚称事务是 “关键应用” 和 “高价值数据” 所必备的重要功能。这两种观点都有些夸张。

事务所提供的安全保证即ACID
atomicity,consistency,isolation,durability

但实际上不同DB的 ACID 实现不尽相同。仅隔离性含义就有很多争议。当一个系统声称自己“兼容ACID” 时，实际上能提供什么保证并不清楚。ACID现在几乎已经变成一个营销术语。

不符合ACID的系统有时被称为BASE：
基本可用性（Basically Available）
软状态（Soft State）
最终一致性（Eventual consistency）
听起来比 ACID 还含糊不清，BASE唯一能确定的是 “它不是 ACID”，此外没有承诺任何东西。

BEGIN TRANSACTION 和 COMMIT 之间的所有内容都属于同一事务

原子性和隔离性也适用单个对象更新。如若向DB写入20KB的JSON文档：
若发送第一个10KB后网络连接中断，DB是否只存储了无法完整解析的10KB JSON片段呢？
若DB正在覆盖磁盘上的前一个值的过程中电源发生故障，最终是否导致新旧值混杂
若另一个客户端在写入过程中读取该文档，是否会看到部分更新的内容

https://cloud.tencent.com/developer/article/2056142?from=article.detail.2056143
精通Java事务编程(4)-弱隔离级别之防止更新丢失

RC 和 快照隔离级别 主要都是为 解决 只读事务遇到并发写时 可以看到什么（虽然中间也涉及脏写)， 还没有触及另一种情况： 2个写事务并发，而脏写只是 写并发的特例。

写事务并发 的最著名的例子 就是 丢失更新，如2个并发计数器增量。

原子写
许多DB支持 原子更新，避免在应用中执行 读取-修改-写入。 这些操作 通常是最好的方案。如下指令 在大多数 关系型DB中 并发安全：
```
UPDATE counters SET value = value + 1 WHERE key = 'foo';
```

类似像：
MongoDB文档DB提供了对 JSON 文档的一部分进行本地修改的原子操作
Redis支持修改数据结构（如优先级队列）的原子操作

实现方案：

1. 一般采用对读取对象加 排它锁 来实现，以便在更新完成之前 没有其他事务可以读它。这种技术有时被称为 游标稳定性(cursor stability)
2. 另一种方案是 强制 所有原子操作 在单线程 执行。

但是ORM框架很容易导致 执行 不安全的 读取-修改-写入，而不是使用 数据库提供的 原子操作。如果你知道 自己在做什么，或许这不会引发问题，但是往往会埋下潜在bug

显示加锁

如果DB不支持 内置原子操作，防止丢失更新的另一个选择是 让应用程序 显示 锁定待更新对象。然后 应用程序 执行 读取-修改-写入， 此时如果其他事务尝试 同时读取对象，则必须等待，直到第一个 读取-修改-写入 完成。

如 多人游戏，其中几个玩家能同时异动一个数字，只靠原子操作可能不够，因为应用还需要确保 玩家的移动符合规则。这可能涉及一些 应用层逻辑，不可能将其剥离转移给DB层在查询时执行。此时可以用锁 来防止 2名玩家 同时 移动相同棋子。

```
BEGIN TRANSACTION;

SELECT * FROM figures
  WHERE name = 'robot' AND game_id = 222
  # 指示DB对返回的所有结果行要加锁。
  FOR UPDATE;

-- 检查玩家的操作是否有效，然后更新先前 SELECT 返回棋子的位置
UPDATE figures SET position = 'c4' WHERE id = 1234;

COMMIT;
```

这有效，但要正确执行，还需要仔细考虑 应用层逻辑。 在代码中 某处 忘记加锁 很容易引入竞争条件。

自动检测更新丢失

原子操作 和 锁 是通过强制 读取-修改-写入 串行执行 来避免丢失更新。

另一种方法是允许它们并发，但若 事务管理器检测到 丢失更新，则中止当前事务，并强制它们回退到 安全你的 读取-修改-写入。

这个方案的一个优点是，DB能结合 快照隔离高效执行 检查。PostgreSQL 的可重复读，Oracle的可串行化 和 SQL Server 的快照隔离级别， 都能自动检测 到丢失更新，并中止违规的事务。但MySQL(InnoDB) 的可重复读 并不会检测 丢失更新。
一些作者认为，DB必须防止丢失更新，才称得上 提供了 快照隔离。所以在这种定义下，MySQL属于没有安全支持快照级别隔离。

丢失更新检测是个好功能，应用代码因此不依赖某些特殊的 DB功能， 你可能忘记 使用锁或 原子操作，但 丢失更新的检测是自动生效的，不太容易出错。

CAS
不提供事务的 DB 有时支持CAS，可以避免丢失更新。
。。乐观锁。

冲突解决和复制
支持多副本的数据库中，防止丢失更新还需要考虑：由于 多节点上存在数据副本，不同节点可能并发修改数据，需要采取额外措施 防止丢失更新。

加锁，cas的前提都是 只有一个 数据副本。但是 多主 或 无主复制的 多副本DB，通常允许多个并发写，并异步复制到副本，所以会出现多个 最新的数据副本。此时 加锁 或 CAS 将不再适用。

多 副本DB 通常允许 并发写入 创建多个冲突版本的值，并使用应用层代码 或特殊 数据结构 来解决，合并这些多版本。

如果操作可以交换(顺序无关)，则原子操作 在多副本下 也能工作。如 递增计数器 或 向集合添加 元素都是 典型的 可交换操作，这个 Riak 2.0 新数据类型思想， 当一个值被不同客户端同时更新时，Riak 自动将更新合并在一起，避免发生 更新丢失。

而最后写入胜利(LWW) 的冲突解决方案 容易丢失更新，不幸的是， LWW 是许多 多副本DB的默认配置。

https://cloud.tencent.com/developer/article/2056143?from=article.detail.2056139
精通Java事务编程(5)-弱隔离级别之写倾斜与幻读

多个事务并发写相同对象时，会出现脏写 和 更新丢失 2种竞争条件。为避免数据不一致，可以：

1. 借助DB内置机制
2. 通过显式加锁，执行原子写操作。

但这还不是 并发写 可能导致的 全部问题
2个人一起请假，查询在职人员，由于DB使用快照隔离，两次都返回2。所以2个事务都进入下一阶段。两个事务都成功提交。最后 全部都 请假了。

定义写倾斜
这种异常称为写倾斜， 不是脏写，也不是丢失更新，这2事务更新的是2个不同对象(2个人 各

自的 状态)。这里发生的冲突不是 那么明显，但是 显然确实是 竞争状态:如果2个事务串行，则第二个人 就不能请假。 异常行为只有 在事务并发时 才可能。
。。这种是外部的限制，数据库没有这种限制的。感觉增加一张表保存 在职人数 就可以了。。请假的时候 乐观锁下。

可以将写倾斜 视为 广义的丢失更新。即若 2事务读取 相同一组对象，然后更新其中一部分：

1. 不同事务可能更新不同对象，即可能发生写倾斜。
2. 而若更新同一对象，则可能脏写 或 丢失更新。

我们有很多方法防止丢失更新，但是对于 写倾斜，方案更受限制：

1. 涉及多个对象，单对象的原子操作无效
2. 基于快照隔离 来实现 自动检测 丢失更新 也有问题：PostgreSQL 的可重复度，MySQL(InnbDB)的可重复度，Oracle可串行化，SQL Server快照隔离级别中， 都不支持自动检测写倾斜。自动防止 写倾斜 要求 真正的可串行化隔离。
3. 某些DB 支持 自定义约束，然后由 DB 强制执行 (如唯一性，外键约束或 特定值限制)。但 指定至少有一人在职，涉及多个对象的约束， 大多DB都没有内置这种约束，但你可以使用 触发器 或物化视图 来实现类似约束。
4. 如果无法使用可串行化，次优方案 可能是 显式 锁定事务依赖的行：

```
BEGIN TRANSACTION;

SELECT * FROM doctors
  WHERE on_call = TRUE
  # 告诉DB锁定返回的所有结果行，以用于更新
  AND shift_id = 1234 FOR UPDATE;

UPDATE doctors
  SET on_call = FALSE
  WHERE name = 'Alice'
  AND shift_id = 1234;

COMMIT;
```

导致写倾斜的幻读

所有的写倾斜都遵循类似的模式：

1. select 搜索，如 至少有1人在岗，不存在对会议室的某时间段的预定，棋盘某位置上没有棋子，用户名没有被注册，账户里还有余额 等
2. 根据查询结果，决定代码是否继续
3. 如果应用决定继续执行，就发起 DB写入(插入，更新，删除)，并提交事务。而该写操作 改变步骤2 做出决定的 前提条件。 即如果提交写入后，在执行步骤1的 查询，会得到不同的结果。

上述步骤可能有不同的执行顺序，如，可以先写，然后select查询，最后根据 查询结果 决定是 提交 还是 中止。

一个事务中的写入改变另一个事务的搜索查询结果，即幻读。快照隔离避免了只读查询中的幻读，但是在像我们讨论的例子那样的读写事务中，幻读会导致特别棘手的写倾斜

物化冲突
如果幻读的问题是 没有对象可以加锁，也许可以考虑人为在DB引入一个 锁对象？
如会议室预定，想象 创建一个 关于 时间槽 和房间的 表。此表中的每行 对应于 特定时间段 的特定房间。可以提前插入房间和时间的所有可能组合行。
现在，要创建预定的事务，可以锁定(select for update) 表中 与 所需房间和时间段 对应的行。锁定后，它可以检查重叠预定并 像以前那样插入新预定。 该表不是用来存储 预定相关信息的，它完全就是 一组锁，以防止 同时修改 同一房间 和范围 内的 预定。

这被称为物化冲突(materializing conflicts) 方案，因为它将 幻读 变成 DB 中一组 具体行上的锁冲突。
但弄清楚如何物化冲突很难，也很容易出错，而且 让并发控制机制 泄漏到 应用 数据模型 是很丑陋的做法。出于这些原因，物化冲突 是最后的手段。 大多数情况下，可串行化(serializable) 隔离级别更可取。

https://cloud.tencent.com/developer/article/2056146
精通Java事务编程(7)-可串行化隔离级别之两阶段锁定（2PL，two-phase locking）

近30年，DB只有一种广泛使用的串行化算法： 两阶段加锁

我们知道，加锁可以防止脏写：即如果2个事务同时尝试写入同一个对象，则锁可以确保 第二个写 必须等 第一个写完成事务(回滚或提交)才能继续。

两阶段锁 类似，但是锁的 强制性更高。只要没有写入，就允许多个事务 同时读取 同一个对象。 但是对象只要有些，就得加锁 独占访问。

- 如果事务A 已读某行数据，此时B想写入这行，就必须等待A完成， 这确保了 B不在 A执行过程中 意外修改对象
- 如果事务A 已写某对象，此时B想读取，则B必须等A完成， 读旧版本的数据 在2PL 下是不可接受的

2PL 并发写互斥， 读写直接也互斥。 快照级别隔离 是 读写不互斥，这是 2PL 和 快照隔离的 关键区别。
2PL提供串行化，所以可以防止 前文(这个是一系列的文章。。)讨论的所有 竞争条件，包括丢失更新和 写倾斜。

实现原理
2PL已在：

- MySQL(InnoDB) 和 SQL Server 实现 可串行化
- DB2 中可重复读

读与写的阻塞是通过为 数据库中每个对象 添加锁来实现的， 锁可以处于 共享模式 或独占模式，使用如下：

1. 若事务要读对象，则须先共享模式 获得锁。允许 多个事务 同时 拥有一个对象的 共享锁。但是 如果某事务已经 持有 对象的 独占锁，则其他事务必须等待
2. 如果事务要写对象，则必须 以 独占模式 获得锁。禁止 其他事务 同时 持有锁(无论共享模式 还是 独占模式)，即 如果对象上存在锁(。。无论读锁写锁)，则写事务就必须等待。
3. 如果事务 先读再写对象，则需将 共享锁 升级为 独占锁。升级锁的流程 和 直接获得独占锁 相同
4. 事务获得锁后，必须一直持有锁，直到事务结束。这就是 两阶段 名字来源：第一阶段(当事务正在执行时)获得锁，第二阶段(事务结束时)释放所有锁。

由于使用了这么多锁，很容易死锁：如 事务A等待 B释放锁，B等待A释放锁。DB会自动检测事务之间的死锁，并强行终止一个，被终止的事务需要 由 应用层重试。

没有被广泛使用的原因是 性能：事务吞吐量 和 查询响应时间 比 弱隔离级别 差太多。部分因为 获取，释放锁的 开销，更重要的是 并发性能降低。按设计，如果2个并发事务 视图做 任何可能导致 竞争条件的 事情，则其一必须等待 另一完成。

传统关系DB 不限制事务的 执行时间，因为它们 是为 等待人类输入 的交互式应用设计的。结合2PL，最终结果是，当一个事务还需要等待另一个事务时，最终等待时间 几乎无上限。即使能保证所有事务都很短，如果有多个事务同时访问 同一个对象，会形成一个队列，事务需要等待前面的事务完成后才能继续。

因此，2PL DB的访问延迟 具有 极大不确定性。

基于锁实现的 RX 也可能死锁，但是2PL 下取决于事务的访问模式，死锁更频繁。这可能是一个额外的性能问题： 当事务 由于死锁而被中止，应用需要 从头重试。

谓词锁

对加锁，忽略了一个 微妙但重要的 细节。在 写倾斜幻读中 的幻读问题， 是一个事务改变了 另一个事务的查询结果。可串行化隔离也必须防止幻读。

会议室预定案例，如果事务在查询 某段时间内的 一个房间的 预定情况，则另一个事务 不能同时 插入 或更新 同一个时间段内 该房间的 预定。

要实现就需要 谓词锁(predicate lock)，类似 共享/独占 锁， 但不属于特定对象(如表的某行)，而是作用于 所有符合某些搜索条件的对象，如：

```
SELECT * FROM bookings
WHERE room_id = 123
AND end_time > '2018-01-01 12:00'
AND start_time < '2018-01-01 13:00';
```

谓词锁会限制如下访问：

1. 如果事务A 想要读取 某些满足 匹配条件的对象，如select 查询，必须获取查询条件上的 共享谓词锁(shared mode predicate lock)。如果事务B 持有任何 满足 这一查询

条件对象的 独占锁，则A必须等B释放锁后 才能继续执行查询
2. 如果事务A想插入，更新或删除 任何对象，必须先检查所有旧值 或 新值 是否和现有谓词匹配。如果 B持有 匹配的谓词锁，则A 需要等待 B 完成或中止 后才能继续。

关键在于，谓词锁甚至适用于数据库中尚不存在，但将来可能会添加的对象（幻象）。如果两阶段锁定包含谓词锁，则数据库将阻止所有形式的写入偏差和其他竞争条件，因此其隔离实现了可串行化。
。。原文就是这样的。

索引范围锁
谓词锁 性能不佳，如果活跃事务有很多锁，则检查匹配的锁 很耗时。因此，大多 2PL DB 实际上 实现的是 索引范围锁 (index-range locking 也称为 next-key locking)。 本质是对 谓词锁的 简化 或近似。

简化谓词锁的方式 是扩大其保护的 对象，这肯定是安全的。如果 你有 12点-13点的 123 房间的 谓词锁，则 锁定 123 房间的所有时间 或 12点-13点的所有房间， 就是 安全的近似。 这样，任何与 原始谓词锁 冲突的操作 肯定和 近似后的区间锁 冲突。

房间预订 数据库，一般在 room_id 列建立索引，并/或 在 start_time,end_time 上有索引。
无论哪种，查询条件的近似值都附加到某个索引上，如果另一事务想插入，更新或删除 同一房间 和/或 重叠时间段的 预订，则需更新这些索引的 相同部分，就会和 共享锁冲突。
这有效防止了 幻读 和 写倾斜。索引范围锁 并不像 谓词锁 精确 (会锁定 更大范围的对象，超出维持可串行化所必需的的范围)， 但是由于开销低很多，是很好的 折中方案。
如果没有可挂载范围锁的 索引，则DB 可退化到 使用整表的 共享锁。这对性能不利。

https://cloud.tencent.com/developer/article/2056148?from=article.detail.2056146
精通Java事务编程(8)-可串行化隔离级别之可串行化的快照隔离

本系列文章描述了DB 并发控制的黯淡：
1. 2PL 虽然保证了 串行化，但是 性能和扩展不好
2. 性能良好的 弱隔离级别，但易出现 各种竞争条件(丢失更新，写倾斜，幻读)

串行化的隔离级别 和 高性能 是相互矛盾的吗？ 也许不是，一种被称为 可串行化快照隔离 (SSI，serializable snapshot isolation) 算法很有前途。 提供完整的可串行化保证，而性能与 快照隔离 相比 只有很小的性能损失。
SSI 在2008年 首次被剔除，如今既用于 单节点DB (PostgreSQL 9.1 后的可串行化)，也用于分布式DB(FoundationDB)。
SSI还能年轻，还在实践中证明自己

悲观锁，乐观锁
两阶段锁是一种 悲观锁机制。设计原则：如果操作可能出错(如与其他事务发生锁冲突)，则直接放弃，等待直到绝对安全。 和多线程编程中的 互斥锁一致。

某种意义上，串行执行是很悲观的：事务期间，每个事务对整个DB（或DB的一个分区）持有互斥锁，我们只能假定每笔事务执行够快、短时持锁，来稍微弥补悲观色彩

相比之下，串行化快照隔离 是一种 乐观锁。如若存在潜在冲突，也不阻止事务，而是继续执行事务，寄希望于一切平安。而当事务想提交时（只有可串行化的事务才被允许提交。），DB会检查是否冲突（即违反隔离性原则）：若是，则中止事务并重试。

乐观锁是古老的想法，其优缺点争论已久。若存在很多冲突，则性能不佳，大量事务需中止。若系统已接近最大吞吐量，重试的额外负载会使系统性能更差。

但若系统有足够性能提升空间，且事务之间争用不大，乐观锁比悲观锁更高效。可交换的原子操作能减少争用：如若多个事务同时增加某计数器，则应用增量的顺序（只要计数器不在同一个事务中读取）就无关紧要，所以并发增量可全部应用且无需冲突。

。。有些东西 只是 + 和 -， 并不需要 之前是什么，然后是什么。 弱一致性，保证最终是一致的，但是中间可能发生不一致的情况。。 或者说 之前只要在某个范围内，就是 有效的， 比如 付款，只要钱包里 多于 10元，那么他就能买 10元的东西。
。。update account amount-=10 where id=xx when amount>=10

SSI基于快照隔离，即事务中的所有读取都基于DB的一致性快照（参阅本文的快照隔离、可重复读），这和早期乐观锁的主要区别。在快照隔离基础上，SSI新增一种算法检测写入之间的串行化冲突，并确定要中止哪些事务。

基于过期条件来决策
讨论写倾斜时，有一种场景，事务先从DB读一些数据，根据查询后结果 决定 后续操作，如修改数据。但快照隔离下，数据可能在查询期间 就已经被其他事务修改，导致 原事务提交时 决策的依据信息已变。

在应用执行查询时，DB不知道 应用会如何使用 查询结果。为了安全，DB假定 对该结果集的变更都可能会使该事务中的写无效。 即 事务中查询 和 写 可能存在 因果依赖关系。 为了提供可串行化隔离，DB必须检测事务是否会修改 其他事务的查询结果，并在此情况下 中止写事务。
DB如何知道查询结果是否已变，可以分为如下情况：

1. 读取是否作用于 一个（即将）过期的 MVCC对象（读取之前已经有 未提交的写入）
2. 检查写 是否影响 即将完成的 读（读取后，又有新写入）

检测旧MVCC读取
快照隔离通常采用MVCC实现。当事务从MVCC DB的一致性快照读时，会忽略创建快照时 还没有提交的事务。

。图，就是 事务 A 读取，乐观锁update， 事务B读取，乐观锁update， A commit， B commit 。 B的commit 会被拒绝，B重试。

为了防止这种异常，DB需要跟踪一个事务 由于 MVCC可见性规则 而被忽略的其他事务写。当事务提交时，DB会检查是否存在 被忽略的 写 现在已经被提交， 如果有，当前事务必须中止。
为什么要等待提交？当检测到 读旧值时，为什么不是立刻中止 事务B？：

1. 如果事务B是 只读的， 那么不需要中止，因为没有 写倾斜风险
2. 当B读取数据时，DB还不知道 后续要执行 写操作。

3. 事务A可能 在事务B提交时，被中止或仍处于未提交状态。

通过避免不必要的中止，SSI可以高效支持哪些 需要 在一致性快照中运行很长时间的读事务

检测写是否影响了之前的读

。图：事务A读取数据库，B读取，A乐观锁update，B乐观锁Update，Acommit，B提交被拒绝。B需要重试。

2PL下讨论了 索引范围锁，允许DB 锁定 与某查询匹配的所有行，如 where shift_id=1234。可在此使用类似技术，只有一点差异：SSI锁不阻塞其他事务。

当另一事务写时，先检查索引，从而确定是否在最近存在一些读目标数据的其它事务。这过程类似在受影响字段范围上获取写锁，但锁不会阻塞其它事务读取，而是直到读事务提交时才进一步通知它们：所读到的数据已变化。

性能
许多工程细节会影响算法实际效果。如一个需要权衡考虑的是 跟踪事务的 读 写 粒度：

1. 如果DB 详细跟踪每个 事务的操作(细粒度)，确实能准确确定哪些事务需要中止，但记录元数据的开销可能很大。
2. 而跟踪速度更快时(粗粒度)，可能导致更多不必要的事务中止

有的场景，读过期数据不会造成太大的影响。

相比于 2PL，可串行化快照隔离 最大优点：事务无需阻塞等待 其他事务持有的锁。这和快照隔离一样，读写不互相阻塞。这使得查询延迟更稳定，可预测。尤其是只读查询 可运行在一致快照，无需任何锁，对 读密集系统友好。

相比于 串行执行，可串行化 快照隔离 可突破单 CPU 核 吞吐量限制：FoundationDB将检测到的 串行化冲突分布在多台机器，从而提高吞吐量。即使数据可能跨多台机器分区，事务也能在保证可串行化隔离等级同时，读写多个分区中的数据。

事务中止率会 显著影响 SSI 性能。如 长时间读，写数据 的事务 很可能会发生 冲突并中止，因此SSI 要求 读写型事务 尽量短。 总体上，对慢事务，SSI 比 2PL 或 串行 更能容忍。

https://cloud.tencent.com/developer/article/2056150?from=article.detail.2056148
精通Java事务编程(9)-总结

事务作为抽象层，允许应用忽略DB内部 一些复杂并发问题和某些硬件，软件故障，简化应用层的处理逻辑：事务中止(transaction abort)， 而应用仅需要重试。 对于复杂访问模式，事务可以大大减少 需要考虑的潜在错误情景 数量。

如果没有事务，各种错误情况(进程崩溃，网络中断，停电，磁盘满，意外并发) 意味着数据可能各种不一致。

如非规范化的数据 可能很容易与源数据不同步。没有事务处理，就很难 推断 复杂的 交互访问可能 对 数据库造成 影响。

隔离等级要点：

脏读：

客户单读到另一个客户端尚未提交的写。 读已提交 或更强 隔离级别可以防止

脏写：

客户端覆盖了另一个客户端尚未提交的写，几乎所有事务都可以防止

读倾斜(不可重复读)

同一个事务中，客户端在不同时间点 看到数据库 不同值。快照隔离 用于解决这个问题，允许事务 从某特定时间点的 一致性快照中读数据，MVCC实现。

。。？ok 不可重复读，是指在数据库访问中，一个事务范围内两个相同的查询却返回了不同数据。这是由于查询时系统中其他事务修改的提交而引起的。

更新丢失

2个客户单同时执行 读取-修改-写入，其中一个写操作，在没有合并 另一个写入变更的 情况下，直接覆盖了 另一个写结果，导致数据丢失。快照隔离的 一些实现 可以自动防止这种异常，而另一实现则需要手动锁定(select for update)

写倾斜

一个事务读取一些东西，根据它所看到的的值决定，并将该决定写入到数据库，但写时，该决定的前提不再成立。 只有可串行化隔离 才能防止。

幻读

事务读取某些符合查询条件的对象，同时另一个客户端写，改变了 先前查询结果。 快照隔离 可以防止 简单的幻读，但 写倾斜的幻读 需要特殊处理，如采用 索引范围锁定。

弱隔离级别可防止上述的一些异常，但还得应用程序开发人员手动处理其它复杂 case，如显式加锁。

只有可串行化隔离级别能防所有这些问题，有三种不同实现方案：

严格串行执行事务

若每个事务的执行很快，且单CPU核即可满足事务吞吐要求，这是简单有效的选择

2PL

数十年来，一直是可串行化的标准实现，但许多应用考虑性能而放弃使用之

可串行化快照隔离（SSI）

最新算法，避免先前方案的大部分缺点。使用乐观锁机制，允许事务并发执行而不互相阻塞。仅当事务提交时，才检查可能的冲突，若发现违背串行化，则中止事务

==========

2PC

==========

严格两阶段锁定（SS2PL, strong strict two-phase locking）

==========

事务的ACID通过 InnoDB 日志 和 锁 来保证。

事务的隔离性是通过 数据库锁 的机制 实现的，持久性通过redo log 来实现。原子性和一致性通过 undo log 实现。

undo log 的原理很简单，为了满足事务的原子性，在操作任何数据之前，先将数据备份到某个地方（这个存储数据备份的地方被称为 undo log）。 然后进行数据的修改。如果出现了错误 或 用户执行了rollback ，系统可以使用 undo log 的备份将数据恢复到 事务开始之前的状态。

和undo log 相反，redo log 记录的是 新数据的备份。在事务提交前，只需要将 redo log 持久化即可，不需要将 数据持久化。当系统崩溃时，虽然数据 没有持久化，但是 redo log 已经持久化。 系统可以根据redo log 的内容，将所有数据恢复到 最新的状态。

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

https://juejin.cn/post/6844903647197806605

分布式事务

分布式事务就是指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上。简单的说，就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。本质上来说，分布式事务就是为了保证不同数据库的数据一致性。
。。。不同 微服务 同一个数据库。。能 跨 微服务共享 一个数据库的一个事务吗？ 感觉是可以做到的啊。。

我们可以分为2块，一个是 service 多个节点， 一个是 resource 多个节点。

service 多个节点
随着发展，微服务，SOA等 服务架构模式 被大规模使用。
用户的资产可能被分为多个部分，如，余额，积分，优惠券 等。所以 可能 积分功能由一个微服务团队 维护(有自己的db)， 优惠券 是另一个 微服务团队维护(有自己的db)。
这样就很难保证 积分扣减之后，优惠券是否扣减成功。

resource 多个节点
MySQL 一般 千万级别的数据 就需要 分库分表。
需要确保 这些数据库之间的 数据一致。

分布式事务的基础
数据库的ACID 已经无法满足 分布式事务。

CAP
又称布鲁尔定理。

C 一致	对于客户端，读操作能返回最新的写操作。对于数据分布在不同节点上的数据来

性	说，如果某个节点更新了数据，那么其他节点如果 都能读取到这个最新的数据，那么就称为强一致，如果某个节点没有读取到，那么就是分布式不一致。
A 可用性	非故障节点在 合理时间内返回合理的响应(不是错误和超时的响应)。可用性的2个关键，一个是合理的时间，一个是合理的响应。合理的时间是指 请求不能无限被阻塞，应该在合理的时间给出返回。合理的响应是指 系统应该明确返回结果 并且结果是正确的，这里的正确是指 应该返回50 而不是返回51.
P 分区容错性	当出现网络分区后，系统能继续工作。 比如，集群中有多态机器，某台机器网络出现问题后，当这个集群仍然可以正常工作。

CAP 3者不能共存，在分布式系统中，网络不是100%可靠，分区是必然会发生的，如果我们选择CA，那么当分区发生时，为了保证一致性，这个时候必须拒绝请求，但是A又不允许，所以分布式系统 理论上不可能选择CA架构。

。。那么C 应该是 分区里 可以立刻读到 其他节点的 写入。

CP，放弃可用性，追求 一致性和分区容错性，zookeeper
AP，放弃一致性(这里指 强一致性)。追求分区容错性和可用性。这是很多分布式系统设计时的选择，后面的BASE 也是根据AP发展来的。

CAP理论中 是忽略网络延迟，也就是当事务提交时，从节点A瞬间复制到 节点B，但是现实中明显不可能，所以总会有一定的时间 是不一致的。

如果你选择了CP，并不是让你放弃A。因为P出现的概率很小，大部分时间你仍然需要保证CA。就算分区出现了， 你也需要为 后续的 A 做准备，比如通过 一些日志的手段，使其他机器恢复 可用。

BASE
Basic Available, Soft state, Eventually consistent
是AP的扩展

基本可用	分布式系统在出现故障时，允许损失部分可用功能，保证核心功能可用
软状态	允许系统中存在中间状态，这个状态不影响系统可用性，这里指的是CAP中的不一致
最终一致性	经过一段时间后，所有节点数据 都会一致

BASE 解决了 CAP中 理论没有网络延迟，在BASE中使用 软状态 和 最终一致，保证了延迟后的一致性。BASE 和 ACID 是相反的，它完全不同于ACID的 强一致性模型，而是通过 牺牲强一致性 来获得可用性，允许数据在一段时间内是不一致的，但最终达到一致状态。

分布式事务

是否真的需要分布式事务

出现分布式事务的2个原因(多个service，多个resource)，其中有个原因就是 微服务过多。微服务过多就会引出 分布式事务，这个时候 不建议 采取下面的方案， 而是 把 需要事务的微服务 聚合成一个 单机服务，使用数据库的 本地事务。

2PC
说到2PC 就不得不说到 数据库分布式事务中的 XA Transactions

在XA协议中分为2阶段：
1. 事务管理器 要求每个涉及到事务的数据库预提交(pre commit) 此操作，并反馈 是否可以提交
2. 事务协调器 要求每个数据库提交数据，或回滚数据

优点：尽量保证了数据的 强一致，实现成本较低，在各大主流数据库都有自己实现，对于MySQL是从5.5开始支持。
缺点：
1. 单点问题：事务管理器很重要，如果它宕机，比如在 第一阶段已经完成，在第二阶段正准备提交的时候事务管理器宕机，资源管理器会一直阻塞，导致数据库不可用。
2. 同步阻塞：准备就绪之后，资源管理器中的资源一直处于阻塞，直到提交完成，释放资源。
3. 数据不一致：2阶段提交 协议 虽然 为 分布式数据 强一致性 所设计，但仍存在 数据不一致的可能。比如，第二阶段中，假设协调者发出了事务commit 的通知，但是因为网络问题 该通知仅被 一部分参与者 收到并执行commit，其余的参与者因为没有收到通知 而一直处于阻塞状态，这就产生了 数据的不一致。

总的来说，XA协议 比较简单，成本较低，但是 其单点问题，以及 不能支持 高并发(由于同步阻塞) 依然是 其最大的弱点。

TCC
try confirm cancel。
TCC 对于 XA，解决了 XA的几个缺点：
1. 解决了协调者单点，由主业务方发起并完成这个业务活动。业务活动管理器也变成多点，引入集群。
2. 同步阻塞，引入超时，超时后进行补偿，并且不会锁定整个资源，并将资源转换为 业务逻辑形式，粒度变小。
3. 数据一致性，有了补偿机制后，由 业务活动管理器 控制一致性。

try阶段：尝试执行，完成所有业务检查(一致性)，预留必须业务资源(准隔离性)
confirm阶段：确认执行真正执行业务，不做任何业务检查，只使用try阶段预留的业务资源，confirm满足幂等性。要求具备幂等性设计，confirm失败后需要重试
cancel阶段：取消执行，释放try阶段预留的业务资源，cancel 操作满足幂等性，cancel阶段的异常和confirm阶段异常处理方案基本上一致。

举例，100元买水：
try：检查是否有100元，并锁住这100元，水也一样
如果有失败，则cancel(释放100元和水)，如果cancel失败，不论什么失败都进行重试 cancel，所以需要保持幂等性
如果都成功，则进行confirm，确认这100元 和 1瓶水 被卖， 如果 comfirm 失败 无论什么失败 都重试 (会依靠活动日志 进行重试)

对于TCC来说，更适合一些：
强隔离性：严格一致性要求的活动业务
执行时间较短的业务

本地消息表
本地消息表，最初由 ebay (应该是人名) 提出。
https://queue.acm.org/detail.cfm?id=1394128

核心是 将需要分布式处理的 任务 通过 消息日志的方式 来异步执行。消息日志可以存储到本地文本，数据库 或 MQ，再通过 业务规则 自动或 人工发起重试。人工重试更多的是应用于 支付场景，通过对账系统对 事后问题的处理。

对于本地消息队列来说 核心是把 大事务转变为 小事务。
还是以 100元买一瓶水 为例：

1. 扣钱的时候，你需要在 你扣钱的服务器上 新增一个本地消息表，你需要把你 扣钱和写入减去水的库存 到本地消息表 放入同一个事务 (依靠数据库 本地事务 保证一致性)
2. 定时任务轮询 这个本地事务表，把没有发送的消息，扔给 商品库存服务器，让它减去水的库存，到达商品库存服务器后，先写入这个服务器的 事务表，然后进行扣减，扣减成功后，更新事务表中的状态。
3. 商品库存服务器 通过 定时任务 扫描消息表 或直接通知 扣钱服务器，扣钱服务器 对本地消息表进行状态更新。
4. 针对一些异常情况，定时扫描未成功处理的消息，进行重新发送，在商品服务器接收消息之后，首先判断是否重复，如果已经接受，再判断是否执行，如果执行，马上通知事务，如果没有，需要重新执行，由于有幂等性，所以不会多扣。

本地消息队列 是BASE 理论，是最终一致模型，适用于 对一致性要求不高的。实现这个模型需要注意重试的幂等。

MQ事务
RocketMQ中实现了分布式事务，实际上 其实 是对 本地消息表的 一个封装，将本地消息表异动到了MQ 内部。
基本流程如下：
第一阶段 prepared 消息，会拿到消息的地址
第二阶段 执行本地事务
第三阶段 通过第一阶段拿到的地址去访问消息，并修改状态。消息接受者就能使用这个消息。

如果确认消息失败，在RocketMQ Broker 中提供了 定时扫描 没有更新状态的消息，如果有消息没有得到确认，会向 消息发送者 发送消息，来判断是否提交，在RocketMQ 中是以 listener 的形式 给发送者，用来处理。

如果消费超时，则需要一直重试，消息接收端需要确保 幂等性。 如果消息消费失败，需要人工处理，因为这个改了较低，所以不值得设计复杂的 流程。

Saga事务
核心思想是将 长事务拆分成 多个本地事务，由 Saga 事务协调器 协调，如果正常结束那就正常完成，如果某个步骤失败，则根据相反顺序 调用 补偿操作。

每个Saga 由一系列 sub-transaction Ti 组成，每个Ti 有对应的补偿动作 Ci 用于撤销Ti造成的影响。这里每个Ti 都是一个本地事务。
可以看到和 TCC相比，Saga没有 "预留Try" 动作，它的Ti 直接提交到数据库

Sage 有2种恢复策略，
T1, T2, ..., Tj, Cj,..., C2, C1，其中0 < j < n
T1, T2, ..., Tj(失败), Tj(重试),..., Tn

Sage 不保证 隔离性，因为没有锁住资源，其他事务依然可以覆盖或者影响当前事务。

Sage 没有隔离性，影响还是比较大的，可以参考 华为的解决方案： 从业务层 入手，加入 Session 及 锁机制 来保证 串行化操作资源。也可以在业务层 通过预先冻结资金的方式 隔离这部分资源，最后 在业务操作的过程中 可以通过 及时读取当前状态的方式 获得 最新的更新。

最后
能不用分布式事务 就不用。如果不得不使用，结合自己的业务分析，看业务适合哪种？在乎强一致性还是最终一致性。

==========

==========