MySQL事务

MySQL如何保证一致性?

- 1. 事务（Transactions）
  - 事务是数据库管理系统 (DBMS) 用来确保数据一致性的重要机制。事务具有四个重要特性，统称为 ACID 特性：
  - 原子性（Atomicity）：事务中的所有操作要么全部完成，要么全部不执行。MySQL 通过事务日志（如 InnoDB 的 redo log 和 undo log）来实现原子性。在事务提交时，将所有操作写入 redo log；如果事务失败，使用 undo log 回滚所有已执行的操作。
  - 一致性（Consistency）：事务使数据库从一个一致状态转换到另一个一致状态。MySQL 通过确保事务执行的所有操作符合数据库的约束和规则来维持一致性。
  - 隔离性（Isolation）：不同事务之间的操作是相互隔离的。MySQL 提供多种隔离级别（详细见后），通过锁和 MVCC（多版本并发控制）来实现隔离性。
  - 持久性（Durability）：一旦事务提交，其所做的更改是永久性的，即使系统崩溃也不会丢失。InnoDB 使用 redo log 确保数据持久性。
- 2. 锁机制（Locks） —— MySQL 通过锁机制来控制并发访问：
  - 表锁（Table Locks）：MySQL 允许对整个表加锁，以避免多个事务同时修改同一表的数据。这种锁的粒度较大，但管理相对简单。
  - 行锁（Row Locks）：InnoDB 支持行级锁，即只锁住特定的行。这种锁的粒度较小，有助于提高并发性能。行锁使用两阶段锁协议：在需要时获取锁，并在事务结束时释放锁。
- 3. 外键约束（Foreign Key Constraints） —— 外键约束用于维护表之间的引用完整性：
  - 引用完整性：外键约束确保子表中的外键值必须在父表中存在。例如，删除或更新父表的主键值时，如果子表中存在引用该值的外键，操作将被拒绝或同步更新。
  - 触发机制：当进行插入、更新或删除操作时，MySQL 会检查相关外键约束，确保数据的一致性。
- 4. 触发器（Triggers） —— 触发器是数据库表上定义的自动执行的存储过程：
  - 事件驱动：触发器在特定事件（如 INSERT、UPDATE 或 DELETE）发生时自动执行。MySQL 支持行级触发器，即每行操作都会触发一次触发器。
  - 维护一致性：触发器可以用来强制复杂的一致性规则，例如在更新某个表时自动更新另一个表的数据。
- 5. 自动提交（Auto-Commit） —— MySQL 的自动提交模式控制每个 SQL 语句是否在执行后自动提交：
  - 默认行为：MySQL 默认处于自动提交模式，每个 SQL 语句在执行后都会自动提交。这样可以简化事务管理，但在复杂操作中可能需要显式管理事务。
  - 显式事务管理：通过 SET AUTOCOMMIT = 0 可以关闭自动提交模式，并使用 START TRANSACTION、COMMIT 和 ROLLBACK 手动管理事务。
- 6. 隔离级别（Isolation Levels）
  - MySQL 支持多种隔离级别，防止不同类型的并发问题：
  - 读未提交（READ UNCOMMITTED）：最低级别，允许读取未提交的数据，可能会导致脏读。
  - 读提交（READ COMMITTED）：只允许读取已提交的数据，避免脏读，但可能会导致不可重复读。
  - 可重复读（REPEATABLE READ）：同一事务中的多个读操作看到的数据一致，避免不可重复读，但可能会导致幻读。
  - 串行化（SERIALIZABLE）：最高级别，通过锁定所有读取的数据行，完全避免并发问题，但性能较差。
  - InnoDB 通过 MVCC 实现可重复读级别的隔离，使用 undo log 保存旧版本的数据行，读取时根据事务的开始时间戳确定读取哪个版本的数据。
- 7. 双写缓冲区（Doublewrite Buffer） —— InnoDB 的双写缓冲区机制确保数据的一致性：
  - 写操作流程：在写数据页到数据文件前，InnoDB 会先将数据页写入双写缓冲区。数据页成功写入后，再写入数据文件。
  - 崩溃恢复：如果系统崩溃，InnoDB 可以从双写缓冲区中恢复未完整写入的数据页，确保数据文件的一致性。
- 8. 校验和（Checksums） —— InnoDB 使用校验和来检测和防止数据损坏：
  - 校验和计算：在写入数据页时，InnoDB 会计算数据页的校验和并存储在数据页的页头。
  - 数据验证：在读取数据页时，InnoDB 会重新计算校验和并与存储的校验和进行比较。如果校验和不匹配，说明数据页可能被损坏，InnoDB 可以采取相应措施来处理。

什么是事务

数据库事务(transaction)是访问并可能操作各种数据项的一个数据库操作序列，这些操作要么全部执行，要么全部不执行，是一个不可分割的工作单位。

四大特性

- 原子性 —— 事务中的操作要么全部完成，要么全部不执行 —— redo log称为重做日志，用来保证事务的原子性和持久性
- 一致性 —— 一致性确保事务从一个一致的状态转换到另一个一致的状态。 —— undo log用来保证事务的一致性
- 隔离性 —— 隔离性确保并发事务的操作是相互隔离的，防止多个事务同时执行时互相干扰。不同的事务之间不会看到对方的中间状态。 —— 锁用来保证事物的隔离性
- 持久性 —— 持久性确保一旦事务被提交，它对数据库中数据的改变就是永久性的。即使发生系统故障，事务执行的结果也不会丢失。持久存储基于磁盘 —— redo log称为重做日志，用来保证事务的原子性和持久性

MySQL事务隔离级别

- READ-UNCOMMITTED(读取未提交) —— 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读。
- READ-COMMITTED(读取已提交) —— 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生。
- REPEATABLE-READ(可重复读)
  - 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。
  - **是MySQL InnoDB 存储引擎的默认支持的隔离级别**
- SERIALIZABLE(可串行化) —— 最高的隔离级别，完全服从 ACID 的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。

隔离级别	脏读	不可重复读	幻读
READ-UNCOMMITTED	√	√	√
READ-COMMITTED	×	√	√
REPEATABLE-READ	×	×	√
SERIALIZABLE	×	×	×

- 脏读
  - 指一个事务读取了另一个事务未提交的数据。如果那个事务回滚，读取的数据就是“脏”的，因为它从未被真正确认过。
  - Innodb如何解决脏读 —— 在“读已提交”（Read Committed）隔离级别下，事务的行为确保了它只能看到在它执行读操作时已经提交的事务所做的更改。
- 幻读
  - 发生在一个事务读取了一些行，然后另一个事务插入了一些新的行。当原事务再次读取同一范围的行时，会发现一些之前不存在的“幻影”行。同一个事物执行两次相同的查询，第一次查询的结果数量与第二次查询的结果数量不一致。 —— 解决幻读的方法
    - 将事务隔离级别调整为SERIALIZABLE可串行化
    - 在可重复读RR的事务级别下，给事务操作的这张表添加表锁。
    - 在可重复读RR的事务级别下，给事务操作的这张表添加Next-key Lock（Record Lock+Gap Lock）
  - RR可重复读下，幻读问题真的解决了吗？并没有
    - InnoDB中在RR这种隔离级别通过**间隙锁+MVCC**解决了大部分的幻读问题，但是并不是所有的幻读都能解读，想要彻底解决幻读，需要使用Serializable的隔离级别
      - RR中，通过间隙锁解决了部分当前读的幻读问题，通过增加间隙锁将记录之间的间隙锁住，避免新的数据插入。
      - RR中，通过MVCC机制的，解决了快照读的幻读问题，RR中的快照读只有第一次会进行数据查询，后面都是直接读取快照，所以不会发生幻读。
      - 但是，如果两个事务，事务1先进行快照读，然后事务2插入了一条记录并提交，再在事务1中进行update新插入的这条记录是可以更新出成功的，这就是发生了幻读。
      - 还有一种场景，如果两个事务，事务1先进行快照读，然后事务2插入了一条记录并提交，在事务1中进行了当前读之后，再进行快照读也会发生幻读
      - MVCC解决幻读 —— 快照读 —— RR的快照读解决
    - 场景一：事务1先查没有，然后事务2插入一条数据，现在事务1再去修改这条数据，因为此时这条记录的trx_id已经变成了当前事务1，和生成ReadView的事务id一样，所有在接下来就能查看到了。归根结底就是因为之间使用update 加了锁 是当前读 所以一定能读到最新的数据
    - 场景二：事务1先执行快照读 select * from t_test where id > 100 得到三条记录，然后事务2 往表中插入一个id = 200的记录并提交。那么此时事务1在执行 当前读 select * from t_test where id > 100 for update 就能得到4条数据。针对这种就需要 开启事务之后 就马上执行 select ... for update当前读语句，这样就会对记录加上 next-key lock 从而避免其它事务再插入一条记录
- 为什么默认RR，大厂要改成RC
  - 提升并发
    - RC 在加锁的过程中，是不需要添加Gap Lock和 Next-Key Lock 的，只对要修改的记录添加行级锁就行了。
    - 这就使得并发度要比 RR 高很多
    - 因为 RC 还支持"半一致读"，可以大大的减少了更新语句时行锁的冲突；对于不满足更新条件的记录，可以提前释放锁，提升并发度
  - 减少死锁 —— 因为RR这种事务隔离级别会增加Gap Lock和 Next-Key Lock，这就使得锁的粒度变大，那么就会使得死锁的概率增大
- 不可重复读
  - 指在同一事务中，多次读取同一数据集合时，后续读取的结果与前面的不一致，通常是因为其他事务在这两次读取之间更新了数据。
    - 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
    - 使用Next-Key Lock算法解决不可重复读
  - 如何解决 —— InnoDB 通过使用 MVCC 来解决**不可重复读**和**脏读**的问题。在RR这种隔离级别下，当我们使用快照读进行数据读取的时候，只会在第一次读取的时候生成一个Read View，后续的所有快照读都是用的同一个快照，所以就不会发生不可重复读的问题了

ReadView

- ReadView是一种数据结构，它在数据库管理系统中，特别是在MySQL的InnoDB存储引擎中，对于实现事务的隔离级别起着关键作用。ReadView主要包含以下几个关键部分：
  - m_ids：记录了当前正在执行但还未提交的事务ID。
  - min_trx_id：m_ids中的最小值。
  - max_trx_id：下一个要生成的事务ID，它总是比当前所有事务的ID都要大。
  - creator_trx_id：创建该ReadView的事务ID。
- 当事务尝试读取一条记录时，它会使用ReadView来判断该记录是否对当前事务可见。具体来说，它会比较记录的trx_id与ReadView中的各个值，以确定该记录是否可以被当前事务读取。

MVCC

- 如何理解MVCC
  - MVCC，是Multiversion Concurrency Control的缩写，翻译过来是多版本并发控制，和数据库锁一样，他也是一种并发控制的解决方案 —— 工作原理
    - 每当一个数据项被修改时，MVCC不会直接覆写原数据，而是创建一个数据项的新版本（或称为“快照”）。每个版本都会被标记一个时间戳或事务ID
    - 读操作：当执行读操作时，MVCC会返回给事务开始时刻之前已经提交的数据版本（快照）
    - 写操作：当事务修改数据时，会创建新的数据版本，并在提交事务时使其可用。
  - 大体就是通过使用快照产生的Read View根据Read View中的信息按照可见性算法，寻找版本链中合适的一条记录
    - undo log 版本链
    - Read View
      - m_ids：当前有哪些事务还在执行，暂未提交的事务id集合
      - min_trx_id：m_ids中最小的事务id
      - max_trx_id：下一个要生成的事务id，这个事务id比当前所有的事务id都要大
      - creator_trx_id：生成ReadView的那个事务id
    - 隐藏字段：trx_id + roll_point
    - 可见性算法 —— 这样在访问某条记录时，只需要按照下边的步骤判断该记录在版本链中的某个版本（trx_id）是否可见
      - 1、trx_id<m_ids列表中最小的事务id —— 表明生成该版本的事务在生成ReadView前已经提交，所以该版本可以被当前事务访问。
      - 2、trx_id>m_ids列表中最大的事务id —— 表明生成该版本的事务在生成ReadView后才生成，所以该版本不可以被当前事务访问。
      - 3、m_ids列表中最小的事务id<trx_id<m_ids列表中最大的事务id —— 此处比如m_ids为[5,6,7,9,10]
      - ①、若trx_id在m_ids中，比如是6，说明创建ReadView时生成该版本的事务还是活跃的，该版本不可以被访问。
      - ②、若trx_id不在m_ids中，比如是8：说明创建ReadView时生成该版本的事务已经被提交，该版本可以被访问。
  - 解决RR下的幻读问题
    - MVCC：因为可重复读每次用的都是刚开始读的那个产生的那个ReadView，所以哪怕中间插入了数据，也查不出来，都是复用的之前的结果，前提是快照读的情况下，如果中间使用了当前读，那么还是会有幻读现象
    - 临键锁(next-key lock)：当执行select ... for updae等语句的时候，会加上next-key lock，如果有其它事务在next-key lock锁范围内插入一条记录，那么这个插入语句就会阻塞，无法插入成功，很好的解决了幻读现象

InnoDB的一次更新事务是怎么实现的

- 1、在Buffer Pool中读取数据：当InnoDB需要更新一条记录时，首先会在Buffer Pool中查找该记录是否在内存中。如果没有在内存中，则从磁盘读取该页到Buffer Pool中。
- 2、记录UndoLog：在修改操作前，InnoDB会在Undo Log中记录修改前的数据。Undo Log是用来保证事务原子性和一致性的一种机制，用于在发生事务回滚等情况时，将修改操作回滚到修改前的状态，以达到事务的原子性和一致性。UndoLog的写入最开始写到内存中的，然后由1个后台线程定时刷新到磁盘中的。
- 3、在Buffer Pool中更新：当执行update语句时，InnoDB会先更新已经读取到Buffer Pool中的数据，而不是直接写入磁盘。同时，InnoDB会将修改后的数据页状态设置为“脏页”（Dirty Page）状态，表示该页已经被修改但尚未写入磁盘。
- 4、记录RedoLog Buffer：InnoDB在Buffer Pool中记录修改操作的同时，InnoDB 会先将修改操作写入到 redo log buffer 中。
- 5、提交事务：在执行完所有修改操作后，事务被提交。在提交事务时，InnoDB会将Redo Log写入磁盘，以保证事务持久性。
- 6、写入磁盘：在提交过程后，InnoDB会将Buffer Pool中的脏页写入磁盘，以保证数据的持久性。但是这个写入过程并不是立即执行的，是有一个后台线程异步执行的，所以可能会延迟写入，总之就是MYSQL会选择合适的时机把数据写入磁盘做持久化。
- 7、记录Binlog：在提交过程中，InnoDB会将事务提交的信息记录到Binlog中。Binlog是MySQL用来实现主从复制的一种机制，用于将主库上的事务同步到从库上。在Binlog中记录的信息包括：事务开始的时间、数据库名、表名、事务ID、SQL语句等。

MySQL 的 select * 会用到事务吗

这种普通的读取操作其实也会在事务的上下文中执行，即使没有明确的开启事务语句，InnoDB存储引擎也会为查询自动开启一个隐式事务。

数据库读写冲突怎么做

- 锁定（Locking）
  - 排他锁（Exclusive Locks）：当一个事务在一个数据项上加上排他锁时，只允许该事务对该数据项进行读写操作。其他事务必须等待直到锁被释放。
  - 共享锁（Shared Locks）：允许多个事务同时读取同一个数据项，但在共享锁定的数据项上，任何事务都不能进行写操作。如果一个事务需要写入，它必须等待直到所有的共享锁被释放，并且获得排他锁。
- 乐观并发控制 —— 在乐观并发控制中，事务在执行过程中不会立即锁定数据，而是在事务提交时检查数据是否被其他事务修改。如果在事务读取数据后和提交之前，数据被其他事务修改，当前事务会被回滚，并可能重新尝试。
- MVCC多版本并发控制 —— MVCC允许每个读操作访问数据的一个快照，而不是最新的版本。这意味着读操作可以不受写操作的影响继续进行，因为它们操作的是不同版本的数据。这种方式特别适用于读操作远多于写操作的数据库。
- 两阶段锁定协议 —— 这是保证事务序列化的一种方法，分为两个阶段：加锁阶段和解锁阶段。在加锁阶段，事务可以获得任意数量的锁，但一旦释放了任何锁，它就不能再获得新的锁。这种方法可以防止某些类型的并发相关的问题，但可能导致死锁。
- 时间戳 —— 每个事务都有一个唯一的时间戳。使用时间戳来决定事务的优先级，早的事务优先于晚的事务。如果一个事务尝试访问一个已经被具有较新时间戳的事务修改的数据项，那么较早的事务可能会被回滚。

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。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。
- 通过XA事务来支持分布式事务的实现
  - 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
- 2PC
  - 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
- TCC
  - TCC 是 Try、Confirm、Cancel 三个词语的缩写
  - 预处理 Try —— Try 操作做业务检查及资源预留
  - 确认 Confirm —— Confirm 做业务确认操作
  - 撤销 Cancel —— Cancel 实现一个与 Try 相反的操作即回滚操作
- 最为常见的内部XA事务存在于binlog与InnoDB存储引擎之间

MySQL日志

主从复制?

- MySQL的主从复制是一种常用的数据备份、读写分离和故障恢复的技术，它允许一台MySQL服务器（主服务器）的数据被复制到一个或多个MySQL服务器（从服务器）上。这种机制在提高数据库的可用性和扩展性方面非常有效。
- 主从复制的工作流程：
  - 二进制日志（Binary Log）：主服务器上的所有数据修改（如INSERT, UPDATE, DELETE等）都会被记录到二进制日志中。这些日志详细记录了数据的变更，以便可以在其他服务器上重放这些变更。
  - 日志传输：从服务器上的I/O线程会连接到主服务器，并请求从上次同步之后的二进制日志。主服务器接收到请求后，将二进制日志的内容发送到从服务器。
  - 重放日志：从服务器接收到二进制日志后，另一个线程（SQL线程）会异步地读取并执行日志中记录的SQL语句，以此来在从服务器上应用这些变更。
- 主从复制的组件：
  - 主服务器：变更数据并记录这些变更到二进制日志中的服务器。
  - 从服务器：从主服务器接收二进制日志并应用这些日志以保持数据同步的服务器。
  - I/O线程：运行在从服务器上，负责从主服务器读取二进制日志。
  - SQL线程：同样运行在从服务器上，负责执行I/O线程接收到的二进制日志中的SQL命令。
- 主从复制的优点和缺点：
  - 优点：
    - 数据冗余：提供数据的备份，增加了数据的安全性。
    - 读写分离：通过主从复制，可以将查询负载分散到一个或多个从服务器上，主服务器专注处理写操作，从服务器处理读操作，从而提高系统的整体性能。
    - 故障恢复：在主服务器出现故障时，可以快速地将其中一个从服务器提升为新的主服务器，减少系统的停机时间。
  - 缺点：
    - 复制延迟：在高负载情况下，从服务器可能会稍微落后于主服务器，尤其是在异步复制的情况下。
    - 数据一致性问题：如果复制过程中出现问题（如网络问题），可能会导致主从服务器间的数据不一致。
    - 资源消耗：维护复制需要额外的服务器资源和网络带宽。
- MySQL在主从模式下是基于什么实现数据同步的
  - 主服务器上的所有数据更改（如INSERT、UPDATE、DELETE等操作）都会被记录到binlog中。然后，这些日志文件会被从服务器读取并重放，从而实现数据的同步。
  - 主服务器将数据更改写入binlog后，从服务器上的IO线程会连接到主服务器，并请求从上次同步以后的binlog内容。这些日志被传输到从服务器后，另一个SQL线程会读取并执行这些日志文件中的SQL语句，以此来更新从服务器上的数据，保持与主服务器的数据一致。
  - 如果主库在数据写入 binlog 但尚未完成刷盘的过程中宕机，可能会发生以下情况
    - 1. **未刷盘数据丢失**：如果主库宕机前尚未完成 binlog 的刷盘操作，那么这部分未刷盘的数据可能会丢失。这意味着部分写入操作可能无法被从 binlog 中恢复，导致数据不一致或者丢失。
    - 2. **数据恢复**：为了尽量减少数据丢失的风险，数据库系统通常会采取一些措施来确保 binlog 数据的可靠性。例如，可以将 binlog 写入到持久化的存储介质（如磁盘）中，并定期进行备份。在主库宕机后，可以通过 replay binlog 的方式来恢复数据。在进行数据恢复时，可能会丢失宕机前未刷盘的部分数据，但可以尽量保证数据的一致性和完整性。
    - 3. **备库数据同步**：如果存在备库（从库），备库通常会定期从主库同步 binlog 数据，并将其应用到自身的数据副本中。在主库宕机后，可以将备库提升为新的主库，以确保系统的可用性和持续性。

有哪些

- 错误日志 —— 记录MySql的启动、运行和关闭过程中的错误信息，它可以帮助我们诊断和解决MySql的问题，如数据库启动失败、连接问题、语法错误等
- 查询日志
  - 一般查询日志(General Log)：记录所有对MySql数据库的请求，无论请求是否成功执行。一般查询日志对于调试和排查问题非常有用，但由于记录了大量的信息，对数据库性能有一定的影响，因此在生产环境一般不启用
  - 慢查询日志(Slow Query Log)：用于记录执行时间超过预定阈值（由配置参数long_query_time定义）的查询语句。慢查询日志可以帮助开发人员和DBA分析和优化执行效率较低的查询语句，以提升数据库性能
- 事务日志
  - undo log
    - 为什么需要undo log
      - 概述：一个事务在执行过程中，还没提交事务之前，如果MySql发生了崩溃，使用undo log日志能进行回滚，它保证了事务中的原子性
      - 实现机理：在事务没提交之前，MySql会先记录更新前的数据到undo log日志文件里
      - 怎么存的?
        - undo存放在数据库内部的一个特殊段（segment）中，这个段称为undo段（undo segment）。undo段位于共享表空间内
        - 在插入一条记录时，要把这条记录的主键记录下来，之后只需要把这个主键值对应的记录删掉就好了
        - 在删除一条记录时，要把这条记录中的内容记录下来，之后回滚只需要把这条记录再插入到表中就好了
        - 在更新一条记录时，要把被更新的列的旧值记下来，之后回滚只需要把这些列更新为旧值就行了。这里涉及到一个更新版本链，也就是会把每个版本记录下来，组成一个链条，这个undo log版本链 + ReadView就可以实现MVCC(多版本并发控制)，对使用MVCC机制实现读已提交、可重复读提供帮助
    - 作用
      - 实现事务回滚，保障事务的原子性
      - 实现MVCC(多版本并发控制)关键元素之一 —— MVCC的实现是通过undo来完成。当用户读取一行记录时，若该记录已经被其他事务占用，当前事务可以通过undo读取之前的行版本信息，以此实现非锁定读取。
    - undo log什么时候刷盘的 —— undo log和数据页的刷盘时机是一样的 因为在Buffer pool中有一个undo页，同时如果对undo页的修改也会记录到redo log中，以保证在断电故障的时候，undo信息不会丢失，从而避免回滚信息丢失，利用redo来保证数据页和undo页的持久化
  - Redo Log
    - 为了保证事务的持久性，主要用于崩溃恢复 —— Redo Log 记录了事务的所有数据更改（这些日志不仅仅记录了数据更改的最终结果，而且还记录了实现这些更改的具体操作）
    - 构成
      - 内存中的重做日志缓冲（redo log buffer），其是易失的
      - 重做日志文件（redo log file），其是持久的 —— 当事务提交（COMMIT）时，必须先将该事务的所有日志写入到重做日志文件进行持久化，待事务的COMMIT操作完成才算完成
    - redo存放在重做日志文件中
    - 在InnoDB存储引擎层产生
    - 在事务进行中不断地被写入
- 归档日志 —— binlog
  - MySql再完成一条更新操作后，Server层还会生成一条binlog，等之后事务提交的时候，会将该事务执行过程中产生的所有binlog统一写入到binlog文件中，binlog文件是记录了所有数据库表结构变更和表数据修改的日志，不会记录查询类的操作
  - 为什么有了binlog还要有redo log? —— 最开始MySql没有InnoDB引擎，MySql自带的引擎是MyISAM，但是MyISAM没有crash-safe的能力，binlog只能用于归档，后来InnoDB以插件的方式引入到MySql的，既然只依靠binlog是没有crash-fase能力的，所以InnoDB使用redo log来实现crash-safe能力
  - binLog是怎么完成数据的一致性
    - 事务记录： —— binlog以“事务”的形式保存更改。这意味着每次数据变更都会被记录为一个完整的事务，包括变更前的状态和变更后的状态。这种记录方式有助于在数据恢复或主从复制时保持数据的一致性。
    - 顺序性和原子性： —— binlog中的事件是按照它们发生的顺序记录的，这保证了操作的顺序性。同时，由于事务的原子性，每个事务内的操作要么全部成功提交，要么全部回滚，这也有助于维护数据的一致性。
    - 恢复和复制的准确性： —— 在主从复制场景中，从服务器会读取主服务器的binlog，并按照相同的顺序执行其中的事务，从而确保主从服务器之间的数据保持一致。在数据恢复过程中，通过重放binlog中的事务，可以准确地将数据库恢复到某个特定时间点的状态。
    - 数据完整性校验： —— binlog还提供了数据完整性校验的功能。例如，在主从复制过程中，从服务器可以通过校验binlog中的事件来确保接收到的数据是完整且未被篡改的，这进一步保证了数据的一致性。
    - 故障恢复： —— 在发生故障时，可以通过binlog来进行数据恢复。由于binlog记录了所有的数据变更操作，因此可以通过重放这些操作来恢复到故障发生前的状态，从而确保数据的完整性和一致性。
  - redo log和binlog有什么区别?
    - 适用对象不同
      - binlog是MySql的Server层实现的，所有存储引擎都可以使用
      - redo log是InnoDB存储引擎实现的日志
    - 文件格式不同
      - binlog有3种格式类型，分别是STATEMENT(默认格式)、ROW、MIXED
        - STATEMENT：每一条修改数据的SQL都会被记录到binlog中(相当于逻辑日志，所以针对这种格式，binlog可以称为逻辑日志)，主从复制中slave端再根据SQL语句重现。但STATEMENT有动态函数的问题，比如用了uuid或者now这些函数，在主库执行的结果和在从库执行的结果不一样
        - ROW：记录行数据最终被修改成什么样了(这种格式的日志，就不能称为逻辑日志了)，不会出现STATEMENT下动态函数的问题。但ROW的缺点是每行数据的变化结果都会被记录，比如执行批量update语句，更新多少行数据就会产生多少条记录，使binlog文件过大，而在STATEMENT格式下只会记录一个update语句而已
        - MIXED：包含了STATEMENT和ROW模式，他会根据不同的情况自动使用ROW模式和STATEMENT模式
      - redo log是物理日志，记录的是在某个数据页做了什么修改，比如我在XXX表空间中的YYY数据页ZZZ偏移量的地方做了AAA修改
    - 写入方式不同
      - binlog是追加写，写满一个文件，就创建一个新的文件继续写，不会覆盖以前的日志，保存的是全量的日志
      - redo log是循环写，日志空间大小是固定的，全部写满就从头开始，保存未被刷入到磁盘中的脏页数据
    - 用途不同
      - binlog用于备份恢复、主从复制
      - redo log用于掉电等故障恢复
  - 如果不小心整个数据库的数据被删除了，能使用redo log文件恢复吗？不可以使用redo log文件恢复，只能使用binlog文件恢复
    - 因为redo log文件是循环写，是会变写边擦除日志的，只记录未被刷入到磁盘中的数据的物理日志，已经刷入到磁盘中的数据都会从redo log文件中擦除
    - 而binlog文件保留的是全量的日志，也就是保存了所有数据变更的情况，理论上只要记录在binlog上的数据都可以恢复，所以如果不小心整个数据库的数据被删除了，得用binlog文件恢复数据
  - binlog什么时候刷盘
    - 事务提交过程中，先把日志写到binlog cache，事务提交的时候，再把binlog cache写到binlog文件中
      - 注意：**一个事务的binlog是不能被拆开的**，无论这个事务有多大，也要保证一次性写入。这是因为有一个线程只能同时有一个事务再执行的设定，所以每当执行一个begin/start transaction的时候，就会默认提交上一个事务，这样如果一个事务的binlog被拆开的时候，再备库执行就会被当作多个事务分段执行，这样就破坏了原子性
      - MySql给每个线程分配了一篇内存用于缓冲binlog，该内存叫binlog cache，参数binlog_cache_size用于控制当个线程内 binlog cache所占内存的大小，如果超过了这个参数规定的大小，就要暂存到磁盘中
    - 什么时候binlog cache会写到binlog文件?
      - 在事务提交的时候，执行器把binlog cache里的完整事务写入到binlog文件中，并清空binlog cache
      - 虽然每个线程有自己binlog cache，但是最终都写到同一个binlog文件
        - MySql提供一个sync_binlog参数来控制数据库的binlog刷到磁盘上的频率
          - sync_binlog = 0 —— 表示每次提交事务都只write，不fsync，后续交由操作系统决定何时将数据持久化到磁盘
          - sync_binlog = 1 —— 表示每次提交事务都会write然后马上fsync
          - sync_binlog = n —— 表示每次提交事务的write，但累计N个事务后才sync
        - MySql默认设置是sync_binlog = 0 也就是不做任何强制性的磁盘刷盘指令，这种性能最好，但是风险也是最高的。一旦主机操作系统发生故障，还没持久化到磁盘中的数据就会丢失
        - 当sync_binlog设置成1的时候，是最安全但是性能损耗最大的。当设置为1的时候，即使主机发生了异常重启，最多丢失一个事务的binlog，已经持久化到磁盘中的数据不会有影响，就是对写入性能影响太大
        - 如果能允许少量事务的binlog日志丢失的风险，为了提高写入的性能，一般会将sync_binlog设置100-1000中的某个数

为什么需要两阶段提交

- 事务提交之后，redo log和binlog都要持久化到磁盘，但是这两个是独立的逻辑，可能出现半成功状态，这样就造成两份日志之间逻辑不一致
  - 如果在redo log刷入到磁盘之后，MySql突然宕机了，而binlog还没有来得及写。就会导致主机数据更新了但从机数据还没更新
  - 如果binlog刷入磁盘之后，MySql突然宕机了，而redo log还没来得及写。就会导致主机数据没有更新，但是从机数据却更新了
  - 为了避免这种出现两份日志之间的逻辑不一致的问题，使用了两阶段提交来解决（两阶段提交其实是分布式事务一致性协议），两阶段提交把单个事务的提交拆分成2个阶段，分别是Prepare（准备阶段）和Commit（提交）阶段。
  - MySql使用了内部XA事务，由binlog作为协调者，存储引擎作为参与者，当客户端执行commit语句的时候或者自动提交的时候。MySql内部会开启一个XA事务，分两阶段完成XA事务的提交
  - 将redo log的写入拆分成了两个步骤：prepare和commit，中间再穿插写入binlog：具体
    - prepare阶段：将XID(内部事务ID)写入到redo log，同时将redo log对应的事务状态设置成prepare，然后将redo log持久化到磁盘中(innodb_flush_log_trx_commit = 1的作用)
    - commit阶段：将XID写入到binlog，然后将binlog持久化到磁盘(sync_binlog = 1的作用)，接着调用存储引擎的提交事务接口，将redo log状态设置成commit，此时该状态并不需要持久化到磁盘，只要write到文件系统的page cache中就够了，因为只要binlog写磁盘成功，就算redo log的状态还是prepare也没有关系，一样会被认为事务已经执行成功
- 异常重启会怎么样?
  - 比如在时刻A和时刻B都可能发生故障，不管是时刻A（redo log已经写入到磁盘，binlog还没写入到磁盘），还是时刻B(redo log和binlog都已经写入到磁盘，但是还没写入commit标识)崩溃，此时的redo log都处于prepare状态
  - MySql重启之后会按顺序扫描redo log文件，碰到处于prepare状态的redo log，就拿着redo log中的XID去binlog查看是否存在此XID
  - 如果binlog中没有当前内部XA事务的XID，说明redolog完成刷盘，但是binlog还没刷盘，则回滚事务。对应时刻A奔溃恢复的情况
  - 如果binlog中有当前内部XA事务的XID，说明redolog和binlog都已经完成了刷盘，则提交事务。对应时刻B奔溃恢复的情况
  - 可以看出，对于prepare阶段的redolog，既可以提交事务，也可以回滚事务，取决于是否能在binlog中查找到与redolog中相同的XID,如果存在就提交事务，如果没有就回滚事务。这样就保证了redo log和binlog这两份日志的一致性了，所以说两阶段提交是以binlog写成功为事务提交成功的标识
- 处于prepare阶段的redo log加上完整binlog，重启就提交事务，MySql为什么要这么设计? —— binlog已经写入了，之后就会被从库使用，所以在主库上面也要提交这个事务，采用这个策略，主库和从库的数据就保持一致了
- 事务未提交的时候，redo log会被持久化到磁盘吗?
  - 会的，因为从redo log buffer持久化到磁盘，会被后台线程每隔1秒一起持久化到磁盘，也就是说事务还没提交的时候，redo log也是可能被持久化到磁盘中的
  - 如果mysql奔溃了，还没提交事务的redo log已经被持久化到磁盘了，MySql重启后，数据不就不一致了？就看binlog有没有持久化到磁盘中，如果没有，那么MySql重启之后就会进行回滚操作。所以就是redo log可以在事务还没提交之前持久化到磁盘，但是binlog必须在事务提交之后，才可以提交到磁盘

两阶段提交会有什么问题

- 磁盘IO次数变高：因为"双一配置"，每个事务提交都会进行两次fsync（刷盘）
  - 因为sync_binglog=1 表示每次提交事务都会将binlog cache里的binlog直接持久化到磁盘
  - 因为innodb_flush_log_at_trx_commit = 1表示每次提交事务时，都将缓存在redo log buffer里的redo log直接持久化到磁盘
  - 所以在一次事务提交过程中，至少会调用2次刷盘操作
- 锁竞争激烈：两阶段提交能够保证单事务的日志保持一致，但是在多事务的情况下，就不能保证两者提交的顺序一致了，所以在两阶段提交的流程基础上，还需要加上一个锁来保证提交的原子性，从而保证多事务的情况下，两个日志的提交顺序一致
- 解决办法：MySql使用binlog组提交(group commit)机制，当有多个事务提交的时候，会将多个binlog刷盘操作合并成一个，从而减少I/O次数
- 引入组提交之后，prepare阶段不变，只针对commit阶段，将commit阶段拆分成三个过程，每个阶段都有一个队列，每个阶段有锁进行保护，保证了事务写入的顺序，第一个进入队列的事务成为leader，leader领导所在队列的所有事务，全权负责整队的操作，完成后通知队列其他事务操作结束。锁粒度减小了，这样就使得多个阶段可以并发执行，从而提升效率
  - flush阶段：多个事务按进入的顺序将binlog 从cache写入文件(不刷盘) —— 第一个事务成为flush阶段的leader，此时后面到来的事务都是Follower，接着获取队列中的事务组，由绿色事务组的leader对redo log做一次write + fsync。也就是将同组事务的redolog刷盘。完成prepare阶段后，将绿色这一组的事务执行过程中产生的binlog写入binlog文件(调用write，不会调用fsync，不会真正刷盘，只是将binlog缓存在操作系统的文件系统中)。可以知道flush阶段队列的作用是用于支撑redo log的组提交。如果在这一步完成之后数据库奔溃，由于binlog中没有该组事务的记录，所以MySql会在重启后回滚该组事务
  - sync阶段：对binlog文件做fsync操作(多个事务的binlog合并一次磁盘) —— 一组事务的binlog写入到binlog文件中(操作系统cache中)，并不会马上执行刷盘操作，而是会等待一段时间，由Binlog_group_commit_sync_delay参数控制，目的是为了组合更多的binlog，然后再一起刷盘。如果在等待的过程中，如果事务的数量提前达到了Binlog_group_commit_sync_no_delay_count参数设定的值，就不用继续等待了，就马上将binlog刷到磁盘中。所以sync阶段队列的作用是用于支持binlog的组提交。如果在这一步完成后数据库奔溃，由于binlog中已经有了事务记录，MySql会在重启后通过redo log刷盘的数据继续进行事务的提交
  - commit阶段：各个事务 按顺序做InnoDB commit操作 —— 最后进入commit阶段，调用引擎的提交事务接口，将redo log状态设置为commit。commit阶段队列的作用是承接sync阶段的事务，完成最后的引擎提交，使得sync可以尽早的处理下一组事务，最大话组提交的效率
- 除了有binlog组提交，在mysql5.7版本也有redo log组提交，在prepare阶段不再让事务各自执行redo log刷盘操作，而是推迟到组提交的flush阶段。prepare阶段融合在flush阶段