事务处理

- **ACID概念**
  - **一致性Consistency：**保证系统中所有的数据都是符合期望的，且相互关联的数据之间不会产生矛盾。
  - **原子性Atomic：**在同一项业务处理过程中，事务保证了对多个数据的修改，要么同时成功，要么同时失败。
  - **隔离性Isolation：**在不同的业务处理过程中，事务保证了各业务在读写的数据相互独立，不会影响彼此。
  - **持久性Durability：**事务保证所有成功被提交的数据修改都能正确地被持久化，不丢失数据。
- **本地事务**
  - **定义：**指仅操作单一事务资源的、不需要全局事务管理器进行协调的事务。仅适用于单服务使用单个数据源的场景，直接依赖于数据源本身提供的事务能力来工作。
  - **实现原子性和持久性**
    - **难处：**实现原子性和持久性的最大困难在于“写入磁盘”这个操作不是原子的
    - **可能出现情形**
      - **未提交事务，写入后崩溃：**程序还没完整修改完数据，但数据库将其中几个数据的变动写入了磁盘，若此时出现了崩溃，一旦重启后，数据库必须有办法知道崩溃前做了一次不完整的磁盘写入，需要将已经修改过的数据从磁盘恢复成未修改状态。
      - **已提交事务，写入前崩溃：**程序已完成修改完数据，但数据库并未全部将数据变动写入磁盘，若此时出现崩溃，一旦重启后，数据库必须有办法知道崩溃前发生过一次完整的操作，需要将没写入磁盘的部分数据重新写入。
    - **解决方法**
      - **提交日志Commit Logging：**事务不直接修改磁盘，而是将数据修改操作记录在日志中，日志记录全部写入后，添加Commit Record标志，数据库根据此信息才开始真正修改磁盘数据，修改完成后在日志中添加End Record标志。
      - **Shadow Paging：**对数据的变动写入数据的副本，当事务成功提交时，数据的修改都成功持久化后，最后一步是修改数据的引用指针。但是涉及隔离性和并发锁时，事务并发能力较弱，因此不适用于高性能数据库。
    - **Commit Logging改进**
      - **缺陷：**所有对数据的修改必须发生在事务提交之后，此时即使磁盘I/O足够空闲也不被允许在事务提交前写入磁盘数据。因此，引入提前写入日志Write-Ahead Logging。
      - **Write-Ahead Logging相关概念**
        - **FORCE：**是否要求事务提交后同步写入数据
        - **STEAL：**是否允许事务提交前写入数据
      - **Undo Log：**提前写入增加了Undo Log日志类型，在变动数据写入磁盘前，必须记录在Undo Log中哪些数据发生了什么变动，以便在事务回滚或者崩溃恢复时根据记录进行数据变动的擦除。此时，Commit Logging的日志称为Redo Log。
    - **Write-Ahead Logging**
      崩溃恢复
      - **分析阶段：**从最后一次检查点开始扫描日志（检查点前的记录都已正常持久化），找出没有End Record的记录，组成待恢复的事务集合，包括事务表和脏页表。
      - **重做阶段Redo：**从待恢复的事务集合中，找出包含Commit Record的日志，将这些日志的修改写入磁盘，并在日志后添加End Record标志，移出待恢复事务集合。
      - **回滚阶段Undo：**经过重做阶段后的事务集合中只剩下需要回滚的事务，根据Undo Log将提交到磁盘的数据变动进行恢复
    - **FORCE和STEAL组合**
      - **FORCE+NO-STEAL：**最慢，但是不需要日志
      - **FORCE+STEAL：**需要回滚日志
      - **NO-FORCE+NO-STEAL：**需要重做日志，即Commit Logging
      - **NO-FORCE+STEAL：**最快，需要重做日志和回滚日志，即提前写入
  - **实现隔离性**
    - **三种锁**
      - **写锁/排他锁X-Lock：**数据有写锁时，只有持有写锁的事务才能对数据进行写入操作。数据持有写锁时，其他事务不能写入，也不能添加写锁。
      - **读锁S-Lock：**多个事务可以对同一个数据添加读锁，数据被添加上读锁之后无法被加上写锁。若一个数据只有一个事务添加了读锁，则允许将其升级为写锁。但是读取数据不一定要添加读锁。
      - **范围锁：**对某个范围直接加排他锁，在这个范围内的数据不能被写入。
    - **隔离级别**
      - **可串行化Serializable：**对事务所有读、写的数据都加上锁，并且使用范围锁。
      - **可重复读Repeatable Read：**对事务所有读、写的数据都加上锁，但是不使用范围锁。因此相比较可串行化，会出现**幻读问题**，指的是在事务执行过程中，两个完全相同的范围查询会得到不同的结果。基于ARIES理论，实际情况有出入，如MYSQL还有MVCC等支持，只读事务不会出现幻读，读写事务才会出现幻读。
      - **读已提交Read Commit：**数据加的写锁会持续到事务结束，但读锁在查询操作之后就直接释放。因此相比较可重复读，会出现**不可重复读问题**，指的是在事务执行过程中对同一行数据的两次查询得到不同的结果。
      - **读未提交Read UnCommitted：**只对事务涉及的数据加写锁并持续到事务结束，但不添加读锁（即不通过读锁来读取数据，直接读

取）。因此相比较读已提交，会出现**脏读问题**，指的是事务读取到其他事务还未提交的数据。

- **MVCC**
  - **定义：**多版本并发控制，是一种无锁读取优化方案，指的是读取时不需要加锁。针对读+写场景，写+写场景还是要上锁，只是区分乐观锁和悲观锁。
  - **基本思路：**对数据库的修改不直接覆盖之前的数据，而是产生新版本和老版本。
  - **实现原理**
    - **隐藏字段：**DB_TRX_ID（最后一次修改该记录的事务ID）、DB_ROLL_PTR（配合undo log指向上一个旧版本）、DB_ROW_ID（隐藏主键）
    - **Read View：**事务进行快照读操作的时候生产的读视图，在该事务执行快照读的那一刻，会生成一个数据系统当前的快照，记录并维护系统当前活跃事务的id。包含三个全局属性，**trx_list**（维护正活跃事务ID）、**up_limit_id**（trx_list中最小ID）、**low_limit_id**（Read View生成时系统尚未分配的下一个事务ID）。
    - **比较规则：判断某一行是否被判断为可读取的数据**。
      当DB_TRX_ID小于up_limit_id时，则为可读取数据；
      当DB_TRX_ID大于low_limie_id时，则为不可读取数据；
      当DB_TRX_ID处于它们之间时，若DB_TRX_ID在trx_list中，
      则不可读取，在trx_list之外则可读取
  - **RR和RC**
    - **RR可重复读：同一个事务中的第一个快照读才会创建Read View**，之后的快照读获取的都是同一个Read View，保证可重复读的特性，避免不可重复读。
    - **RC读已提交：每个快照读都会生成并获取最新的Read View**，不可避免会出现不可重复度现象。

全局事务

- **定义：**与本地事务相对应，指适用于单个服务多个数据源的事务解决方案。是一种在分布式环境中仍然追求强一致性的事务处理方案。
- **X/Open XA**
  事务处理架构
  - **组件**
    - 全局的事务管理器Transaction Manager，用于协调全局事务
    - 局部的资源管理器Resource Manager，用于驱动本地事务
  - **两段式提交2PC**
    - **准备阶段：**协调者询问参与者是否准备好，参与者进行回复Prepared或Non-Prepared。数据库进行准备操作时，**在重做日志中进行到事务提交前的最后一步Commit Record不写入**，此时事务未完整提交，仍然持有锁，维持隔离性。
    - **提交阶段：**协调者若收到所有的Prepared消息，则将自己的本地事务状态标为Commit，之后向所有参与者发送Commit消息，让参与者进行事务提交操作；若收到Non-Prepared消息或有参与者超时未回复，则协调者将自身事务状态标为Abort，并且向参与者发送Abort消息，让参与者进行回滚。
  - **前提条件**
    - **必须假设网络在提交阶段的短时间内是可靠的，不会丢失消息**。在投票阶段失败了可以回滚补救，但是若提交阶段失败了则无法补救，只能等崩溃的节点重新恢复。因此该阶段尽可能耗时较短。
    - **必须假设因为网络分区、机器崩溃或其他原因而导致失联的节点最终都能恢复**
  - **缺点**
    - **单点问题：**协调者重要性太大，参与者等待协调者指令无法做超时处理，若协调者宕机则所有参与者都会受影响。
    - **性能问题：**所有参与者都被绑定为一个统一调度的整体，在2PC需要经过两次远程服务调用、三次数据持久化（准备阶段写重做日志、协调者做状态持久化、提交阶段写入事务提交），整个过程必须等待参与者中最慢的那一个。
    - **一致性风险：**若协调者在持久化本地事务后，网络突然断开，无法向所有参与者发出Commit消息，则会出现协调者数据已提交，但参与者的数据未提交，且无法回滚，导致数据不一致。
  - **三段式提交3PC**
    - **背景：**解决2PC中的单点问题和性能问题
    - **过程：**3PC将准备阶段拆分为CanCommit和PreCommit阶段，将提交阶段称为DoCommit阶段。其中，CanCommit是询问阶段，让参与者评估事务是否可以顺利完成，避免2PC时有一个参与者准备失败，其他参与者所作工作都白费的情况发生。若PreCommit之后协调者宕机，则默认操作是参与者提交事务，避免单点问题。

共享事务

- **定义：**与全局事务刚好相反，指多个服务共用一个数据源
- **实现方式**
  - 直接让各个服务共享数据库链接，但要求数据源的使用者在同一个进程内。增加一个中间数据源服务器的角色，其他服务都通过它来与数据库交互
  - 使用消息队列服务器来代替中间数据源服务器，即通过消息的消费者来统一完成本地事务
- **缺点：**数据库一般是压力最大且最不易伸缩扩展的资源，现实中几乎没有反过来代理一个数据库为多个服务提供事务协调。

分布式事务

- **定义：**指多个服务同时访问多个数据源的事务处理机制，即在分布式服务环境下的事务处理机制。
- **CAP**
  - **一致性Consistency：**代表数据在任何时刻、任何分布式节点所看到的都是符合预期的。
  - **可用性Availability**：代表系统不间断地提供服务的能力。包括可靠性和可维护性两个指标，**可靠性使用平均无故障时间MTBF度量，可维护性用平均可修复时间MTTR来度量**。可用性A=MTBF/(MTBF+MTTR)
  - **分区容忍性Partition Tolerance：**代表分布式环境中部分节点因为网络原因而彼此失联，与其他节点形成网络分区时，系统仍能正确地提

供服务的能力。

- **放弃CAP的影响**
  - **放弃分区容忍性：**即假设节点之间的通信永远可靠，但是这是不可能存在的
  - **放弃可用性：**意味着一旦发生网络分区，则节点之间信息的同步时间可以无限延长
  - **放弃一致性：**意味着一旦发生网络分区，节点之间提供的数据可能不一致。
    AP是当前分布式系统的主流选择，毕竟选择分布式就是为了追求高可用。
    选择CP是在一些宁愿中断服务也不允许数据不一致的情况。
    选用AP不意味着完全放弃一致性，而是追求最终一致性。此时ACID成为刚性事务，其余分布式事务称为柔性事务。
- **可靠事件队列**
  - **定义：**也叫做最大努力一次提交，将最有可能出错的业务以本地事务的方式完成后，采用不断重试的方式来促使同一个分布式事务中的其他关联业务全部完成。
  - **过程**
    - 1、最终用户发出一个请求，涉及多个服务，生成分布式事务
    - 2、根据服务的出错概率进行动态排序，选择最容易出错的最先进行
    - 3、将最容易出错的服务的业务数据操作和消息写入作为本地事务写入数据库
    - 4、在系统中建立消息服务，定时轮询消息表，将进行中的消息发给其他服务
    - 4-1、若其他关联服务完成了各自的事务，则向最初的服务返回执行结果，初始服务将消息状态改为已完成，至此分布式事务结束。
    - 4-2、若有关联服务未收到消息，则消息服务器将在轮询时向未响应的服务重发发生消息，因此消息处理必须是幂等性的。
    - 4-3、若关联服务无法完成工作，此时消息服务器依旧会持续发生消息，直至操作成功或人工介入。因此可靠事件队列一旦第一个服务事务成功了，就没有失败回滚的概念，只允许后续操作成功。
    - 4-4、若关联服务完成了事务，但响应消息丢失，则消息服务器将持续发送消息，直到收到响应消息。
  - **缺点：**整个过程没有任何隔离性，一些业务若缺乏隔离性将导致出现异常情况。例如，存在购买业务，两个用户各自购买的数量未超过商品数量，但是合起来的数量却超过了，而未做隔离性设置，将导致这两个业务都成功，但实际上关联业务出现异常。
- **TCC事务**
  - **定义：**Try-Confirm-Cancel事务，相比较于可靠事件队列，增加了隔离性。是一种业务侵入性较强的事务方案。
  - **阶段**
    - **Try：**尝试执行阶段，完成所有业务可执行性的检查，并且预留好全部需要用到的业务资源，保障隔离性，必要时冻结资源。
    - **Confirm：**确认执行阶段，直接使用Try预留的资源来完成业务处理，由于此阶段需要循环直至业务完成，因此要求操作需要幂等性。
    - **Cancel：**取消执行阶段，释放Try预留的资源，同样要求操作具有幂等性。
  - **优势和不足：**TCC类似于2PC的准备和提交阶段，TCC在业务执行时只操作预留资源，几乎不会涉及锁和资源的竞争，有一定的性能潜力。但是TCC也带来更高的开发成本和业务入侵性，一般由分布式事务中间件来完成。
  - **限制：**它的业务入侵性很强，但是存在有些第三方业务是不被允许锁定资源的情况，因此第一阶段Try就无法进行。
- **Saga事务**
  - **定义：**提升长时间事务运作效率的方式，将大事务拆分成可以交错执行的一系列子事务。相比较TCC，Saga不需要冻结资源，补偿操作一般比冻结操作简单。
  - **操作**
    - 1、将大事务拆分成多个小事务，即将事务T拆分为T1~TN
    - 2、为每一个子事务设计补偿动作，要求子事务和补偿操作都具备幂等性，同时补偿操作必须能提交成功
  - **恢复策略**
    - **正向恢复：**一直重试直到所有子事务完成
    - **反向恢复：**从失败的子事务开始，逆序执行补偿操作
  - **前提：**Saga必须保证所有子事务都能提交或补偿，由于Saga本身也有可能崩溃，因此被设计为与数据库类似的日志机制，保证系统恢复后可以追踪到子事务的执行情况。
- **AT事务模式**
  - **做法：**在业务数据提交时拦截所有SQL，将SQL对数据修改前、修改后的结果保存快照，生成行锁，通过本地事务一起提交到操作的数据源中，相当于自动记录了重做和回滚日志。若分布式事务成功提交，则自动清理每个数据源中对应的日志数据，若分布式事务需要回滚，则根据日志数据自动生成补偿的逆向SQL
  - **优势：**基本逆向SQL的补偿方式，分布式事务的每一个数据源都可以独立提交，然后立即释放锁和资源。相比较2PC，提升了系统的吞吐量。
  - **缺点：**大幅度牺牲了隔离性，甚至影响了原子性。在缺乏隔离性的前提下，以补偿代替回滚并不是总是成功的，可能本地事务提交后，分布式事务完成之前，该数据被补偿前被其他操作修改了，出现了脏写，此时分布式事务若需要回滚，则只能人工介入。因此GTS增加了全局锁来实现写隔离，避免脏写。