事务

1. 事务的定义
    - 需要保证**原子性、隔离性、一致性**和**持久性**的一个或多个数据库操作（sql语句）的集合称之为一个事务（Transaction），目前只有InnoDB和NDB存储引擎支持事务。事务的4个特征简记为ACID
2. 事务的相关操作
    1. 开启一个事务 开启一个事务，可以通过begin或者start transaction语句来进行开启一个事务。这二者的区别如下：
        - start transaction语句后边可以跟随修饰符
            - read only：标识当前事务是一个只读事务，也就是属于该事务的数据库操作只能读取数据，而不能修改数据。注意：对于临时表来说（我们使用create temporary table创建的表），由于它们只能在当前会话中可见，所以只读事务其实也是可以对临时表进行增、删、改操作的。创建一个只读事务：start transaction read only。
            - read write：标识当前事务是一个读写事务，也就是属于该事务的数据库操作既可以读取数据，也可以修改数据。
            - 如果我们不显式指定事务的访问模式，那么该事务的访问模式就是读写模式
        - begin语句默认开启的事务是一个读写事务，并且不能加修饰符修饰
    2. 提交和终止一个事务
        1. 提交事务的语句是commit，如果我们写了几条语句之后发现上边的某条语句写错了，我们可以手动的使用rollback语句来将数据库恢复到事务执行之前的样子，称之为手动终止事务。但是rollback语句是我们程序员手动的去回滚事务时才去使用的，如果事务在执行过程中遇到了某些错误（可能是sql语句的语法错误）而无法继续执行的话，事务自身会自动的回滚，从而保证事务的原子性。
        2. 自动提交：mysql中有一个系统变量autocommit，默认值为ON，也就是说默认情况下，如果我们不显式的使用start transaction或者 begin语句开启一个事务，那么每一条sql语句都算是一个独立的事务，当执行完这条语句便会自动提交，这种特性称之为**事务的自动提交**。
    3. 事务的状态
        1. 活动的：事务的sql语句没有执行完，仍然在执行sql语句的过程中
        2. 部分提交的：事务已经提交，即已经执行完了commit;语句，但是数据还没有从内存中刷新到磁盘
        3. 失败的：因为各种原因，比如操作系统错误，比如物理出错（断电），比如事务本身的语法错误等，事务没有执行完毕便结束
        4. 终止的：事务在失败的状态通过undo log撤销前面sql语句所做的修改后便进入到了终止的状态

5. 提交的：事务已经将内存中修改的数据成功刷新到了磁盘

3. 事务的特征 ACID

1. 原子性Atomicity

- 对大多数程序员来说，我们的任务就是把现实世界的业务场景映射到数据库的世界中。现实生活中的转账操作映射到mysql中可能对应着数据库的多条操作语句，而在现实生活中，转账操作要么成功，要么失败，没有中间结果。所以映射到mysql中，对应转账的sql语句集合，要么全部执行，要么全部不执行。不能只执行一部分，这是很明显的，因为只有当sql语句全部执行，才意味着一个转账操作的完成。即属于一个事务的sql语句要么全部执行成功，要么全部不执行，这被称之为事务的原子性。但是原子性在某些情况下可能被破坏。

1. 破坏原子性的可能操作

- 数据库操作语句本身的错误（语法错误），导致事务无法继续执行下去，或者是操作系统错误，甚至是直接断电之类的，最终的结果都是导致只执行了某些语句，事务没有能确保原子性，造成数据错误。

- 程序员可以在事务执行过程中手动输入rollback语句结束当前的事务的执行，从而rollback语句之后的语句不会继续执行，类似于java中的break语句。但是之前已经执行了的sql语句对数据库中的数据进行了修改。

2. mysql如何保证原子性：通过undo log来保证原子性

- 在我们对数据库的某条记录进行改动之前，比如对这条记录进行删除（delete），更新（update），以及新增一条记录（insert）时，都会先将回滚（rollback，回到原来的未改动时的初始状态）时需要的信息记录下来到相应的undo log里。（注意：select查询是不会产生undo log）。

- 比如：你插入一条记录时，至少要把这条记录的**主键值记下来**，之后回滚的时候只需要把这个主键值对应的记录删掉就好了。你删除了一条记录，至少要把这条**记录中的内容**都记下来，这样之后回滚时再把由这些内容组成的记录插入到表中就好了。你修改了一条记录，至少要把修改这条记录前的**旧值都记录下来**，这样之后回滚时再把这条记录更新为旧值就好了。

- 数据库中的任何记录都会有一个格外的隐藏列叫roll-pointer，这是一个指针，指向了对这条记录进行改动而生成的undo log，事务的不断改动从而形成了一个链表，本质上可以称之为版本链。这样的话，回滚时就可以通过这个指针找到undo log，其实也就是找到了这条记录的各个版本。由于undo log包含了恢复这条记录原始状态的全部数据，从而可以使该条记录能成功恢复到原始状态。所以，当一个事务因某些错误未能正确提交，或者程序员手动执行rolllback语句，让事务提前终止，从而mysql会自动调用这些undo log，从而该事务之前已经执行的sql语句进行的修改会回到原来未经修改的初始状态。

2. 一致性Consistency

- 主要需要程序的编写者来保证。如果数据库中的数据全部符合现实世界中的约束（all defined rules），我们说这些数据就是一致的。即比如设计一个成绩数据库，总分100分。那么存入该数据库的成绩是不可能多于100分的。也不j可能是负数。因为数据库本身是对现实生活的映射，所以一致性本身即是要保证数据符合客观实际规律。

1. 如何保证一致性

    - mysql**自身的语法支持**，比如not null。比如可以设置一些触发器来检查不合理的输入。
    - 更多的**一致性需求需要靠写业务代码的程序员自己保证**

3. 隔离性Isolation

- 若有一个账户转账操作，即A，B均向C转账100元。若该两个人同时（注意：强调同时性）对这个账户转账，那么最终转账的结果应该是这两次转账的和，即这两次转账的操作应该是互不影响的，这符合我们的预期。类比于java中的原子性操作。当某个线程对某个变量进行操作时，其他线程应该处于等待状态，只有当该线程释放了该变量的锁，其他线程才能去争抢这个锁，只有争抢到了这个锁的线程才能有机会对该变量进行操作。现实生活中，账户的最终余额肯定是两次转账的和。而映射到mysql中，我们也要保证结果的正确性。即不同的操作应该是相互隔离的，虽然他们可能在同时操作同一条数据。mysql是一个客户端/服务器软件，对于运行着mysql服务的某个服务器来说，可能有若干个客户端与之连接，每个客户端与服务器连接上之后，就可以称之为一个会话。每个客户端都可以在自己的会话中向服务器发出请求语句，一个请求语句可能是某个事务sql语句集的一部分，也有可能一个语句就对应着一个事务。也就是说对于服务器来说，可以同时处理多个事务。那么就会出现对某个记录，可能同时有多个事务对其操作。从而会出现某些数据不安全的情况：
    - 脏写: 比如一个事务，在执行过程中修改了某一个数据，但是该事务并没有提交，此时，由于服务端可以同时处理多个会话请求，此时另外一个事务紧接着也修改了这个数据，从而使之前的事务出现很懵逼的情况，他以为自己已经成功修改了，但是实际上没有。称这种情况为脏写。脏写是一种很可怕的情况，所以mysql在任何隔离级别下都不会让脏写发生。
    - 脏读: 比如一个事务，在执行过程中修改了某一个数据，但是该事务并没有提交，由于服务端可以同时处理多个会话请求，此时另外一个事务读了这个数据，但是之前的事务进行了rollback操作，导致后面的事务相当于读取到了不存在的数据。称这种情况为脏读。
    - 不可重复读: 对于某个事务，该事务会不断的重复读取(也可能只读取两次，或者多次，反正大于等于2)某个数据的值，但是在这事务进行的过程中，有其他事务(可能多个事务)修改了这个数据的值，并且提交了。那么这个事务就很懵逼，为啥我每次读取这个数据都不一样呢？把这种情况叫做不可重复读。
    - 幻读: 如果一个事务先根据某些条件查询出一些记录，之后另一个事务又向表中插入了符合这些条件的记录，并且该事务也进行了提交。原先的事务再次按照该条件查询时，会把另一个事务插入的记录也读出来，那就意味着发生了幻读。
- 由于以上问题都是由于并发造成的。但是根据不同的需求，我们可能不在意某些情况发生。所以设置了各种隔离级别，对每一种隔离级别，解决了不同的问题。SQL规定：
    - read uncommitted：可能发生脏读，不可重复读，幻读
    - read committed：不会发生脏读，但是可能会发生重复读和幻读
    - repeatable read：可能会发生幻读的现象，但是不会发生不可重复读和脏读，实质上mysql在REPEATABLE READ的隔离级别下，通过MVCC解决了幻读的问题，即

mysql在REPEATABLE READ隔离级别下，是不会发生幻读现象的，这也是mysql默认条件下的隔离级别。

- serializable：各种问题都不会发生。
- mysql的默认隔离级别为REPEATABLE READ，我们可以手动修改一下事务的隔离级别
- SET [GLOBAL|SESSION] TRANSACTION ISOLATION LEVEL level。其中：其中的level可选值有4个，即对应于上面的四个隔离级别:

```
level:
{
    REPEATABLE READ
    READ COMMITTED
    READ UNCOMMITTED
    SERIALIZABLE
}
```

4. 持久性Durability

- 即当某个事务正确完成并且commit之后，那么这个事务对数据的状态改变应该是永久的，应该在磁盘上保存下来，不应该因为其他的原因而被撤销。比如断电啥的可能发生的情况。

1. 为了保证持久性，那么需要将修改后的数据从内存中刷新到磁盘中。但是如果这样做，此时会出现以下几个问题:

    - 刷新一个完整的数据页太浪费
        - 假设有时候我们仅仅修改了某个页面中的一个字节，但是我们知道InnoDB存储引擎是以页为单位来进行磁盘IO的，也就是说一次会将一页的数据也就是16k的数据读取到内存中，刷新的时候，会将一页的数据从内存刷新到磁盘。也就是说如果我们这样做，我们在该事务提交时不得不将一个完整的页面从内存中刷新到磁盘，我们又知道一个页面默认是16KB大小，只修改一个字节就要刷新16KB的数据到磁盘上显然是太浪费了。
    - 随机IO刷新起来比较慢
        - 一个事务可能包含很多sql语句，不同的sql语句操作的数据可能分布在不同的页面。比如同一个事务中的sql语句A修改的记录可能在页面1上，语句B修改的记录可能在页面2上。即使是一条语句也可能修改许多页面，倒霉的是该事务修改的这些页面可能并不相邻，这就意味着在将某个事务修改的页面刷新到磁盘时，需要进行很多的随机IO（即这些页存储在磁盘的各个地方，不是在一起存储，因为一个页面16k，磁盘很大可能给不出连续的16k的存储空间，所以会在磁盘上的由存储空间的地方会给出这16k的存储空间，造成数据存储不是连续的），随机IO比顺序IO（所有的数据在磁盘上连续的存放着，这样寻找这些数据直接就把所有的数据给找到了，效率高）要慢的多，尤其对于传统的机械硬盘来说。所以为了保持一致性，在事务提交时就将数据刷新到磁盘是不理智的。

- 本质需求
    - 我们只是想让已经提交了的事务对数据库中数据所做的修改永久生效，即使后来系统崩溃，在重启后也能把这种修改恢复出来。所以我们其实没有必要在每次事务提交时就把该事务在内存中修改过的全部页面刷新到磁盘，只需要把修改了哪些东西记录一下就好。

2. 解决办法：通过`redo log`来解决

- `redo log`本质上只是记录了一下事务对数据库做了哪些修改。比如有个记录，有个`name`属性，名字叫`Mary`。然后我们有一个事务将`Mary`改成了`Jerry`。并且假设这条记录在页面`A`（页面也就是一段大小为`16k`的存储空间）上，并且该事务进行了提交。从而我们可以写一个`redo log`，可以这么写（大白话）：页面`A`在某个地方的值（即`Mary`在这个`16k`内存空间所处的内存位置）由`Mary`变成了`Jerry`。这样就以很小的内存空间记录下了哪个页面的哪个地方由什么改成了什么。这样我们可以换个思路，不把数据页刷新到磁盘，毕竟`16k`太大了，我们只把这些`redo log`刷新到磁盘。这些`redo log`记录了这个事务做了哪些改变，当发生了一些意外情况，可以通过`redo log`进行数据的回复。这样做有以下几个优点：

1. `redo log`占用的空间非常小
    - 因为只记录了哪个页面哪个位置哪个地方改动了什么，所以相比较于很大的`16k`数据页，就特别小的内存。
2. `redo log`是顺序写入磁盘的
    - 在执行事务的过程中，每执行一条语句，就可能产生若干条`redo log`，这些日志是按照产生的顺序写入磁盘的，即这些很小的`redo log`在磁盘上是顺序存储的。也就是使用顺序`IO`。而数据页可能不一样，由于数据页一次需要磁盘`16k`的内存，所以数据页都是分散在磁盘各处的。所以当访问这些数据页，就是随机`IO`速度很慢。