事务的教案

什么是事务，为什么要有事务

eg. 客户下了订单，这个时候需要从账户余额付款。流程是从账户余额 -> 电商 那么操作流程如下

1. 用户账户余额扣钱
2. 电商账户余额加钱
3. 标记用户付款成功

如果我们执行了 步骤1 成功之后，不凑巧，停电了。会出现什么情况。（其它情况如何？）

1. 用户账户的钱被扣了，但是电商账户的钱没有增加，付款也没有成功

能不能准确的处理这类问题，比如要么都操作，要目就不操作？ ！
答案是可以的，就是用事务。

数据库事务(Database Transaction) ，是指作为单个逻辑工作单元执行的一系列操作，要么完全地执行，要么完全地不执行。 事务处理可以确保除非事务性单元内的所有操作都成功完成，否则不会永久更新面向数据的资源。通过将一组相关操作组合为一个要么全部成功要么全部失败的单元，可以简化错误恢复并使应用程序更加可靠。

一个逻辑工作单元要成为事务，必须满足所谓的ACID（原子性、一致性、隔离性和持久性）属性。

名称	内容
原子性 范围需正确 （Atomic） （Atomicity)	事务必须是原子工作单元；对于其数据修改，要么全都执行，要么全都不执行。通常，与某个事务关联的操作具有共同的目标，并且是相互依赖的。如果系统只执行这些操作的一个子集，则可能会破坏事务的总体目标。原子性消除了系统处理操作子集的可能性。
一致性 （Consistent） (Consistency)	事务在完成时，必须使所有的数据都保持一致状态。在相关数据库中，所有规则都必须应用于事务的修改，以保持所有数据的完整性。事务结束时，所有的内部数据结构（如 B 树索引或双向链表）都必须是正确的。某些维护一致性的责任由应用程序开发人员承担，他们必须确保应用程序已强制所有已知的完整性约束。例如，当开发用于转帐的应用程序时，应避免在转帐过程中任意移动小数点。
隔离性 （Insulation） (Isolation)	由并发事务所作的修改必须与任何其它并发事务所作的修改隔离。事务查看数据时数据所处的状态，要么是另一并发事务修改它之前的状态，要么是另一事务修改它之后的状态，事务不会查看中间状态的数据。这称为隔离性，因为它能够重新装载起始数据，并且重播一系列事务，以使数据结束时的状态与原始事务执行的状态相同。当事务可序列化时将获得最高的隔离级别。在此级别上，从一组可并行执行的事务获得的结果与通过连续运行每个事务所获得的结果相同。由于高度隔离会限制可并行执行的事务数，所以一些应用程序降低隔离级别以换取更大的吞吐量。
持久性 （Duration） (Durability）	事务完成之后，它对于系统的影响是永久性的。该修改即使出现致命的系统故障也将一直保持。

事务和性能是相互制约的。
事务性越强，性能就越低

白话说ACID

名称	白话内容
原子性	要么数据全部修改，要么数据都不修改，
一致性	如果表里面定义一个字段是唯一的，那么一致性就解决所有操作中导致这个字段不唯一的情况。如果带有一个外键的一行记录被删除，那么其外键相关记录也应该被删除。
隔离性	MySQL有行级锁，表级锁，比如 数据库表T 有10行数据。行级别锁，数据id为1的数据和数据id为2的数据，可以被不同的事务隔离做写操作。但是在表级锁的情况，一个事务锁定了表T，那么别的事务就不能更改数据了。（共享锁（s锁/读锁）/排他锁（x锁/写锁）只是简单介绍下，不深究，后面的事务隔离状态要用到）
持久性	可以理解为一旦完成操作，事务就被持久化了。

ACID主要面向是的传统的关系型数据库
而且这些属性是完备的。

扩展内容CAP

互联网的迅猛发展出现了很多分布式NoSQL数据库，这个时候还有一个概念叫CAP原则与之对应。但是CAP原则不是完备的，正常情况下，只能同时满足两个需求。

CAP是分布式系统中进行平衡的理论

名称	内容
Consistent一致性	同样数据在分布式系统中所有地方都是被复制成相同。
Available可用性	所有在分布式系统活跃的节点都能够处理操作且能响应查询。
Partition Tolerant分区容错性	在两个复制系统之间，如果发生了计划之外的网络连接问题，对于这种情况，有一套容错性设计来保证。

一次封锁or两段锁?

因为有大量的并发访问，为了预防死锁，一般应用中推荐使用一次封锁法，就是在方法的开始阶段，已经预先知道会用到哪些数据，然后全部锁住，在方法运行之后，再全部解锁。这种方式可以有效的避免循环死锁，但在数据库中却不适用，因为在事务开始阶段，数据库并不知道会用到哪些数据。
数据库遵循的是两段锁协议，将事务分成两个阶段，加锁阶段和解锁阶段（所以叫两段锁）

- 加锁阶段：在该阶段可以进行加锁操作。在对任何数据进行读操作之前要申请并获得S锁（共享锁，其它事务可以继续加共享锁，但不能加排它锁），在进行写操作之前要申请并获得X锁（排它锁，其它事务不能再

获得任何锁）。加锁不成功，则事务进入等待状态，直到加锁成功才继续执行。
- 解锁阶段：当事务释放了一个封锁以后，事务进入解锁阶段，在该阶段只能进行解锁操作不能再进行加锁操作。

事务	加锁/解锁处理
begin;	
insert into test	加insert对应的锁
update test set...	加update对应的锁
delete from test	加delete对应的锁
commit;	事务提交时，同时释放insert、update、delete对应的锁

这种方式虽然无法避免死锁，但是两段锁协议可以保证事务的并发调度是串行化（串行化很重要，尤其是在数据恢复和备份的时候）的。

事务的四种隔离级别（重中之重）

在数据库操作中，为了有效保证并发读取数据的正确性，提出的事务隔离级别。我们的数据库锁，也是为了构建这些隔离级别存在的。

隔离级别

1. 未提交读（Read uncommitted）完全没有事务可言，任何操作都不会加锁，不讨论。
2. 已提交读（Read committed）
3. 可重复读（Repeatable read）
4. 可串行化（Serializable）完全串行化的读，每次读都需要获得表级共享锁，读写相互都会阻塞

隔离级别	脏读（**Dirty Read**）	不可重复读（**NonRepeatable Read**）	幻读（**Phantom Read**）
未提交读（Read uncommitted）	可能	可能	可能
已提交读（Read committed）	不可能	了能	可能
可重复读（Repeatable read）	不可能	不可能	可能
可串行化（Serializable）	不可能	不可能	不可能

隔离级别	说明
未提交读（Read uncommitted）	允许脏读，也就是可能读取到其他会话中未提交事务修改的数据
已提交读（Read committed）	只能读取到已经提交的数据。Oracle等多数数据库默认都是该级别 (不重复读)
可重复读	

（Repeatable read）	可重复读。在同一个事务内的查询都是事务开始时刻一致的，InnoDB默认级别。在SQL标准中，该隔离级别消除了不可重复读，但是还存在幻象读
可串行化（Serializable）	完全串行化的读，每次读都需要获得表级共享锁，读写相互都会阻塞

- 脏读(Dirty Read): 脏读就是指当一个事务正在访问数据，并且对数据进行了修改，而这种修改还没有提交到数据库中，这时，另外一个事务也访问这个数据，然后使用了这个数据。
  读的数据和实际的不一样

你返回的是会被roll back的数据，实际值和你读到的值，不一致。

- 不可重复读(Unrepeatable Read): 不可重复读意味着，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。
  一个事务里两次读的不一样
  例如：事务B中对某个查询执行两次，当第一次执行完时，事务A对其数据进行了修改。事务B中再次查询时，数据发生了改变

就是读了2次 第一次读了老值，第二次读了新值

- 幻读(phantom read): 幻读,是指当事务不是独立执行时发生的一种现象，例如第一个事务对一个表中的数据进行了修改，这种修改涉及到表中的全部数据行。同时，第二个事务也修改这个表中的数据，这种修改是向表中插入一行新数据。那么，以后就会发生操作第一个事务的用户发现表中还有没有修改的数据行，就好象发生了幻觉一样. 新加了数据导致，collection里有增加删除，导致操作异常

第一次读，事务A发现没有数据，准备插入数据，事务B直接插入了数据，数据A插入的时候发现插入失败（如unique唯一限定）。

- 由于MySQL的InnoDB默认是使用的RR级别
- SQLSERVER: 默认为READ_COMMITTED

原来的事务是如何处理的

我们先来看一段代码

```
public int example() {
    Connection conn = null;
    PreparedStatement pstmt = null;
    ResultSet rs = null;
    try {
        String sql = "INSERT INTO table1 (value01,value02) values(?,?)";
        conn = ConnectionUtil.getConnection();
        conn.setAutoCommit(false);
        pstmt = conn.prepareStatement(sql.toString(), Statement.RETURN_GENERATED_KEYS);
        pstmt.setString(1, "1");
        pstmt.setString(2, "2");
        result = pstmt.executeUpdate();
        conn.commit();//手动提交
        return result;
    } catch(Exception e) {
        try {
            conn.rollback(); //回滚
        } catch (SQLException e1) {
            e1.printStackTrace();
        }
```

```
        return 0;
    } finally {
        try {
            if(pstmt != null)pstmt.close();
            if(conn != null)conn.close();
            if(rs != null)rs.close();
        } catch (Exception e2) {
            e2.printStackTrace();
        }
    }
}
```

现在我们是怎么处理的

看Spring org/springframework/jdbc/datasource/DataSourceTransactionManager.java 的代码，这变spring在doBegin()方法下已经关闭了autoCommit;

```
if (con.getAutoCommit()) {
    txObject.setMustRestoreAutoCommit(true);
    if (this.logger.isDebugEnabled()) {
        this.logger.debug("Switching JDBC Connection [" + con + "] to manual commit");
    }

    con.setAutoCommit(false);
}
```

spring使用的小例子，这个Transcation是加在方法上的，本质上也是spring通过aop来实现。

```
@Transcation(rollbackFor=Exception.class)
public void doSomething(Foo foo){
  fooRepo.save(foo);
  foo.modify();
  fooRepo.save(foo);
}
```

点开Transcation注解

```
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface Transactional {
  @AliasFor("transactionManager")
  String value() default "";
  @AliasFor("value")
  String transactionManager() default "";
  Propagation propagation() default Propagation.REQUIRED;
  Isolation isolation() default Isolation.DEFAULT;
  int timeout() default TransactionDefinition.TIMEOUT_DEFAULT;
  boolean readOnly() default false;
  Class<? extends Throwable>[] rollbackFor() default {};
  String[] rollbackForClassName() default {};
  Class<? extends Throwable>[] noRollbackFor() default {};
  String[] noRollbackForClassName() default {};
}
```

属性	类型	描述
value	String	可选的限定描述符，指定使用的事务管理器
propagation	enum: Propagation	可选的事务传播行为设置
isolation	enum: Isolation	可选的事务隔离级别设置
readOnly	boolean	读写或只读事务，默认读写
timeout	int (in seconds granularity)	事务超时时间设置
rollbackFor	Class对象数组，必须继承自Throwable	导致事务回滚的异常类数组
rollbackForClassName	类名数组，必须继承自Throwable	导致事务回滚的异常类名字数组
noRollbackFor	Class对象数组，必须继承自Throwable	不会导致事务回滚的异常类数组
noRollbackForClassName	类名数组，必须继承自Throwable	不会导致事务回滚的异常类名字数组

事务隔离级别

- TransactionDefinition.ISOLATION_DEFAULT：这是默认值，表示使用底层数据库的默认隔离级别。对大部分数据库而言，通常这值就是TransactionDefinition.ISOLATION_READ_COMMITTED。
- TransactionDefinition.ISOLATION_READ_UNCOMMITTED：该隔离级别表示一个事务可以读取另一个事务修改但还没有提交的数据。该级别不能防止脏读，不可重复读和幻读，因此很少使用该隔离级别。比如PostgreSQL实际上并没有此级别。
- TransactionDefinition.ISOLATION_READ_COMMITTED：该隔离级别表示一个事务只能读取另一个事务已经提交的数据。该级别可以防止脏读，这也是大多数情况下的推荐值。
- TransactionDefinition.ISOLATION_REPEATABLE_READ：该隔离级别表示一个事务在整个过程中可以多次重复执行某个查询，并且每次返回的记录都相同。该级别可以防止脏读和不可重复读。
- TransactionDefinition.ISOLATION_SERIALIZABLE：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。但是这将严重影响程序的性能。通常情况下也不会用到该级别。

事务传播行为

所谓事务的传播行为是指，如果在开始当前事务之前，一个事务上下文已经存在，此时有若干选项可以指定一个事务性方法的执行行为。在TransactionDefinition定义中包括了如下几个表示传播行为的常量：

- TransactionDefinition.PROPAGATION_REQUIRED：如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。这是默认值。
- TransactionDefinition.PROPAGATION_REQUIRES_NEW：创建一个新的事务，如果当前存在事务，则把当前事务挂起。
- TransactionDefinition.PROPAGATION_SUPPORTS：如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。
- TransactionDefinition.PROPAGATION_NOT_SUPPORTED：以非事务方式运行，如果当前存在事务，则把当前事务挂起。
- TransactionDefinition.PROPAGATION_NEVER：以非事务方式运行，如果当前存在事务，则抛出异常。
- TransactionDefinition.PROPAGATION_MANDATORY：如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。

- TransactionDefinition.PROPAGATION_NESTED：如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于TransactionDefinition.PROPAGATION_REQUIRED。

事务超时

所谓事务超时，就是指一个事务所允许执行的最长时间，如果超过该时间限制但事务还没有完成，则自动回滚事务。在 TransactionDefinition 中以 int 的值来表示超时时间，其单位是秒。

默认设置为底层事务系统的超时值，如果底层数据库事务系统没有设置超时值，那么就是none，没有超时限制。

事务只读属性

只读事务用于客户代码只读但不修改数据的情形，只读事务用于特定情景下的优化，比如使用Hibernate的时候。默认为读写事务。