事务的概念

事务指逻辑上的一组操作,组成这组操作的各个单元,要么全部成功,要么全部不成功.

管理事务

1.数据库默认的事务

数据库默认支持事务的,但是数据库默认的事务是一条sql语句独占一个事务,这种模式,意义不大. 2.手动控制事务　如果希望自己控制事务也是可以的:

```
start transaction;
---开启事务,在这条语句之后的所有的sql将处在同一个事务中,要么同时完成要么同时不完成.
.......
--事务中的sql在执行时,并没有真正修改数据库中的数据
commit;
--提交事务,将整个事务对数据库的影响一起发生
rollback;
--回滚事务,将这个事务对数据库的影响取消掉
```

3.JDBC中控制事务

当jdbc程序数据库获得一个Connection对象时,默认情况下这个Connection对象会自动向数据库提交在它上面发送的SQL语句,若想关闭这种默认提交方式,让多条SQL在一个事务中执行,可使用下列语句:

```
conn.setAutoCommit(false);
--关闭自动连接后,conn将不会帮我们提交事务,在这个连接上执行所有的sql语句将处在同一个事务中,需要我们是手动的进行提交或回滚.
conn.commit();
--提交事务
conn.rollback();
--回滚事务
也可以设置回滚点回滚部分事务
SavePoint sp = conn.setSavePoint();
conn.rolback(sp);
--注意,回到回滚点后,回滚点之前的代码虽然没被回滚但是也没被提交,如果想起作用还要做commit操作.
```

事务的四大特性

事务的四大特性是事务本身具有的特点,简称ACID.

```
*原子性(Atomicity).
原子性是指事务是一个不可分割的工作单位,事务中的操作要么都发生,要么都不发生.
*一致性(Consistency)
事务前后数据的完整性必须保持一致
*隔离性(Isolation)
```

事务的隔离性是指多个用户并发访问数据库时,一个用户的事务不能被其它用户的事务锁干扰,多个并发事务之间数据要相互隔离.
*持久性
持久性是指一个事务一旦被提交,它对数据库中数据的改变就是永久性的,接下来即使数据库发生故障也不应该对其有任何影响.

隔离性

1.脏读:

```
一个事务读取到另一个事务未提交的数据
--------------------------------
a   1000
b   1000
--------------------------------
a:
    start transaction;
    update account set money=money-100 where name='a';
    update account set money=money+100 where name='b';
-------------------------------------------------
b:
    start transaction;
    select * from account;
    a   900
    b   1100
    commit;
------------------------------
a:
    rollback;
--------------------------------
b:
    select transaction;
    select * from account;
    a   1000
    b   1000
    commit;
```

2.不可重复读:
一个事务多次读取数据库中同一条记录,多次查询的结果不同(一个事务读取到另一个事务已经提交的数据)

```
a   1000    1000    1000
-----------------------------
b:  start transaction;
    select 活期 from account where name = 'a';---活期存款:1000元
    select 定期 from account where name = 'a';---定期存款:1000元
    select 固定 from account where name = 'a';---固定资产:1000元
    ----------------------
    a:
        start transaction;
        update account set 活期=活期-1000 where name='a';
        commit;
```

```
        ------------------
    select 活期+定期+固定 from account where name='a';---总资产:2000元
```

3.虚读(幻读)

有可能出现,有可能不出现:一个事务多次查询整表数据,多次查询,由于有其他事务增删数据,造成的查询结果不同(一个事务读取到另一个事务已经提交的数据)

```
a   1000
b   1000
--------------------------
d:
    start transaction;
    select sum(money) from account;---总存款3000元
    select count(*) from account;---总账户2个
    -----------------
    c:
        start transaction;
        insert into account values('c',3000);
        commit;
        ----------------
    select avg(mone) from account;---平均每个账户:2000元
```

数据库的隔离级别

```
数据库的四大隔离级别:
Read uncommitted;(读未提交)
---不做任何隔离,可能造成脏读 不能重复读 虚读(幻读)问题
Read Committed;(读已提交)
---可以防止脏读,但是不能防止不可重复读 虚读(幻读)问题
Repeatable Read;(可重复读)
---可以防止脏读 不可重复读,但是不能防止虚读(幻读)问题
Serializable;(序列化)
---可以防止所有隔离性的问题,但是数据库就被设计为了串行化的数据库,性能很低
```

从安全性上考虑:

```
Serializable > Repeatable Read > Read Committed > Read uncommitted
```

从性能上考虑

```
Read uncommitted > Read committed > Repeatable Read >Serializable
```

其中Serializable性能太低用的不多,Read uncommitted安全性太低用的也不多,我们通常从Repeatable Read和Read committed中选择一个.

如果需要防止不可重复读选择Repeatable Read,如果不需要防止选择Read committed

数据库默认的隔离级就是Repeatable Read
Oracle数据库默认的隔离级别是Read committed

操作数据库的隔离级别

1.查询数据库的隔离级别:

```
select @@tx_isolation;
```

2.修改数据库的隔离级别:

```
set[session/global] transaction isolation level xxxx;
```

不写默认是session,修改的是当前客户端和服务器交互时使用的隔离级别,并不会影响其他客户端的隔离级别
如果写成global,修改的是数据库默认的隔离级别(即新开客户端时,默认的隔离级别),并不会修改当前客户端和已经开启的客户端的隔离级别. ###数据库的锁 ### 1.共享锁
共享锁和共享锁可以共存,共享锁和排他锁不能共存,在非Serializable隔离级别下做查询不加任何锁,在Serializable隔离级别下做查询加共享锁.
2.排他锁
排他锁和共享锁不能共存,排他锁和排他锁也不能共存,在任何隔离级别下做增删改都加排他锁.
3.死锁
当两边都是Serializable隔离级别时,两边都先进行查询,再尝试进行修改,则互相等待对方释放共享锁,都无法接着进行执行.
死锁的解决方法有两种办法:避免死锁,解决死锁.
mysql可以自动检测到死锁,错误退出一方执行另一方

更新丢失问题

1.更新丢失问题的产生
两个并发的事务基于同一个查询结果进行修改,后提交的事务忽略了先提交的事务对数据库的影响,造成了先提交的事务对数据库的影响丢失,这个过程就叫做更新丢失.
2.更新丢失解决方案　将数据库设为Serializable隔离级别,但是我们一般不会将数据库设置为Serializable
在非Serializable下可以用乐观锁和悲观锁解决更新丢失.

```
悲观锁:在查询时,手动的加排他锁,从而在查询时就排除可能的更新丢失.
乐观锁:在表中设计版本字段,在进行修改时,要求根据具体版本进行修改,并将版本字段+1,如果更新失败,说明更新丢失,需要重新进行更新.
总结:两种解决方案各有优缺点,如果查询多于修改用乐观锁,如果修改多于查询用悲观锁.
```

事务的传播级别

1.propagation_required: 默认的Spring事务传播级别，使用该级别的特点是：如果上下文中已经存在事务，那么就加入到事务中执行；如果当前上下文中不存在事务，则新建事务执行。所以这个级别通常能满足处理大多数的业务场景。

2.propagation_supports: 从字面意思就知道，supports，支持，该传播级别的特点是：如果上下文存在事务，则支持事务加入事务，如果没有事务，则使用非事务的方式执行。

所以说，并非所有的包含在TransactionTemplate.execute方法中的代码都会有事务支持。这个通常是用来处理那些并非原子性的非核心业务逻辑操作。应用场景较少。

3.propagation_mandatory: 该级别的事务要求上下文中必须要存在事务，否则就会抛出异常！配置该方式的传播级别是有效的控制上下文调用代码遗漏添加事务控制的保证手段。

比如一段代码不能单独被调用执行，但是一旦被调用，就必须有事务包含的情况，就可以使用这个传播级别。

4.propagation_requires_new: 从字面即可知道，new，每次都要一个新事务，该传播级别的特点是：每次都会新建一个事务，并且同时将上下文中的事务挂起，执行当前新建事务完成以后，上下文事务恢复再执行。

这是一个很有用的传播级别，举一个应用场景：现在有一个发送100个红包的操作，在发送之前，要做一些系统的初始化、验证、数据记录操作，然后发送100封红包，然后再记录发送日志，发送日志要求100%的准确，如果日志不准确，那么整个父事务逻辑需要回滚。

怎么处理整个业务需求呢？就是通过这个PROPAGATION_REQUIRES_NEW级别的事务传播控制就可以完成。发送红包的子事务不会直接影响到父事务的提交和回滚。

5.propagation_not_supported: 这个也可以从字面上看出，农田supported:不支持，当前级别的特点是：若上下文中农工存在事务，则挂起事务，执行当前逻辑，结束后恢复上下文的事务。

这个级别有什么好处？可以帮助你将事务尽可能的缩小，我们知道一个事务越大，它存在的风险就越多，所以在处理事务的过程中，要保证尽可能的缩小范围，比如一段代码，是每次逻辑操作都必须调用的，比如循环1000次的某个非核心业务逻辑操作，这样的代码如果包在事务中，势必造成事务太大，导致出现一些难以考虑的异常情况，所以事务的这个传播级别就派上用场了。用当前级别的事务模块包含起来就可以了。

6.propagation_never: 该事务更严格，上面一个事务传播级别只是不支持而已，有事务就挂起，而propagation_never传播级别要求上下文中不能存在事务，一旦有事务，就抛出runtime异常，强制停止执行！这个级别上辈子跟事务有仇。

7.propagation_nested: 从字面上看，nested:嵌套级别事务，该传播级别的特征是：如果上下文中存在事务，则嵌套事务执行，如果不存在事务，则新建事务。

嵌套是子事务嵌套在父事务中执行，子事务是父事务的一部分，在进入子事务之前，父事务建立一个回滚点，叫save point,然后执行子事务，这个子事务的执行也算是父事务的一部分，然后子事务执行结束，父事务继续执行。重点就在于那个save point.看几个问题就明了了：

(1).如果子事务回滚，会发生什么？
父事务会回滚到进入到子事务钱建立的save point,然后尝试其他的事务或者其他的业务逻辑，父事务之前的操作不会受影响，更不会自动回滚。

(2).如果父事务回滚，会发生什么？
父事务回滚，子事务也会跟着回滚，为什么呢，因为父事务结束之前，子事务不会提交，我们说子事务是父事务的一部分，由父事务统一提交。

(3).事务的提交，是什么情况？
是父事务先提交，子事务后提交？还是子事务先提交，父事务后提交？答案是第二种情况，还是那句话，子事务是父事务的一部分，由父事务统一提交。

使用即在代码上加上`@Transactional(propagation = Propagation.REQUIRED)`