SpringBoot事务管理

什么是事务

- 事务指的是逻辑上的一组操作，这组操作要么全部成功，要么全部失败。

事务特性（ACDI）

- 原子性（**Atomicity**）
  - 是指事务是一个不可分割的工作单位，事务中的操作要么全都发生，要么都不发生。
- 一致性（**Consistency**）
  - 指事务前后数据的完整性必须保持一致
- 隔离性（**Isolation**）
  - 指多个用户并发访问数据库时，一个用户的事务不能被其他用户的事务所干扰，多个并发事务之间数据要相互隔离。
- 持久性（**Durability**）
  - 指的是一个事务一旦被提交，它对数据库中数据的改变就是持久的，即使数据库发生故障也不应该对其有任何影响。

spring接口

spring事务管理高层抽象主要包括3个接口

- Platformtransaction

  平台事务管理

  - 事务提交
  - 事务回滚

    。。。
- TransactionDefinition

  事务的定义信息

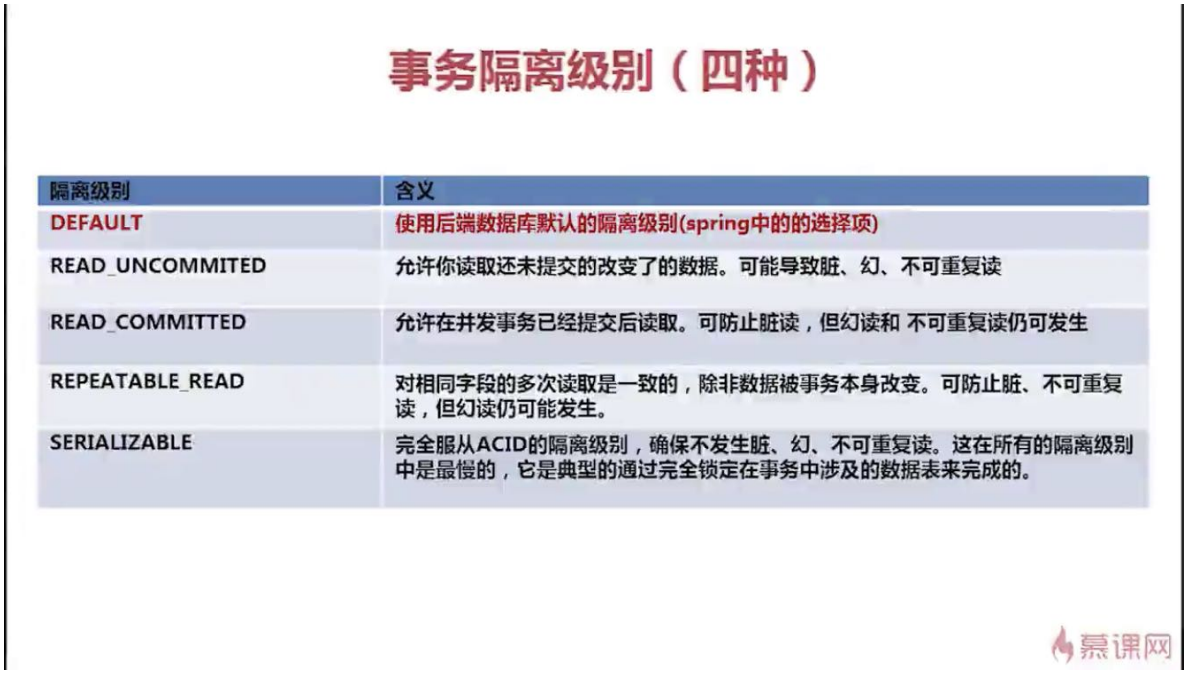
  - 隔离级别
  - 传播行为
  - 是否超时
  - 是否只读

- TransactionStatus

  事务的具体运行状态

  - 事务是否提交
  - 事务是否有保存点
  - 事务是否是一个新的事务

事务的隔离级别



- 补充

  Serializable： 最严格，串行处理，消耗资源大

  Repeatable Read：保证了一个事务不会修改已经由另一个事务读取但未提交（回滚）的数据

  Read Committed：大多数主流数据库的默认事务等级

  Read Uncommitted：保证了读取过程中不会读取到非法数据。

- SpringBoot新增一种隔离级别，**DEFAULT**

  默认隔离级别，每种数据库支持的事务隔离级别不一样，如果Spring配置事务时将isolation设置为这个值的话，那么将使用底层数据库的默认事务隔离级别。顺便说一句，如果使用的MySQL，可以使用"**select @@tx_isolation**"来查看默认的事务隔离级别

- MySQL默认的事务隔离级别为repeatable-read
- Oracle 默认:读已提交Read Committed

事务传播行为

传播行为	含义
PROPAGATION_REQUIRED（XML 文件中为 REQUIRED)	表示当前方法必须在一个具有事务的上下文中运行，如有客户端有事务在进行，那么被调用端将在该事务中运行，否则的话重新开启一个事务。（如果被调用端发生异常，那么调用端和被调用端事务都将回滚）
PROPAGATION_SUPPORTS(XML 文件中为 SUPPORTS）	表示当前方法不必需要具有一个事务上下文，但是如果有一个事务的话，它也可以在这个事务中运行
PROPAGATION_MANDATORY(XML 文件中为 MANDATORY）	表示当前方法必须在一个事务中运行，如果没有事务，将抛出异常
PROPAGATION_NESTED(XML 文件中为 NESTED)	表示如果当前方法正有一个事务在运行中，则该方法应该运行在一个嵌套事务中，被嵌套的事务可以独立于被封装的事务中进行提交或者回滚。如果封装事务存在，并且外层事务抛出异常回滚，那么内层事务必须回滚，反之，内层事务并不影响外层事务。如果封装事务不存在，则同 PROPAGATION_REQUIRED 的一样
PROPAGATION_NEVER（XML 文件中为 NEVER)	表示当方法务不应该在一个事务中运行，如果存在一个事务，则抛出异常
PROPAGATION_REQUIRES_NEW(XML 文件中为 REQUIRES_NEW）	表示当前方法必须运行在它自己的事务中。一个新的事务将启动，而且如果有一个现有的事务在运行的话，则这个方法将在运行期被挂起，直到新的事务提交或者回滚才恢复执行。
PROPAGATION_NOT_SUPPORTE（XML 文件中 NOT_SUPPORTED）	表示该方法不应该在一个事务中运行。如果有一个事务正在运行，他将在运行期被挂起，直到这个事务提交或者回滚才恢复执行
PROPAGATION_REQUIRED（XML 文件中为 REQUIRED)	表示当前方法必须在一个具有事务的上下文中运行，如有客户端有事务在进行，那么被调用端将在该事务中运行，否则的话重新开启一个事务。（如果被调用端发生异常，那么调用端和被调用端事务都将回滚）
PROPAGATION_SUPPORTS(XML 文件中为 SUPPORTS）	表示当前方法不必需要具有一个事务上下文，但是如果有一个事务的话，它也可以在这个事务中运行
PROPAGATION_MANDATORY(XML 文件中为 MANDATORY）	表示当前方法必须在一个事务中运行，如果没有事务，将抛出异常
PROPAGATION_NESTED(XML 文件中为 NESTED)	表示如果当前方法正有一个事务在运行中，则该方法应该运行在一个嵌套事务中，被嵌套的事务可以独立于被封装的事务中进行提交或者回滚。如果封装事务存在，并且外层事务抛出异常回滚，那么内层事务必须回滚，反之，内层事务并不影响外层事务。如果封装事务不存在，则同 PROPAGATION_REQUIRED 的一样

- PROPAGATION_REQUIRED--支持当前事务，如果当前没有事务，就新建一个事务,最常见的选择。
- PROPAGATION_SUPPORTS--支持当前事务，如果当前没有事务，就以非事务方式执行。

- PROPAGATION_MANDATORY--支持当前事务，如果当前没有事务，就抛出异常。
- PROPAGATION_REQUIRES_NEW--新建事务，如果当前存在事务，把当前事务挂起, 两个事务之间没有关系，一个异常，一个提交，不会同时回滚
- PROPAGATION_NOT_SUPPORTED--以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
- PROPAGATION_NEVER--以非事务方式执行，如果当前存在事务，则抛出异常

脏读、幻读、重复读

- 脏读

  所谓脏读，就是指事务A读到了事务B还没有提交的数据，比如银行取钱，事务A开启事务，此时切换到事务B，事务B开启事务-->取走100元，此时切换回事务A，事务A读取的肯定是数据库里面的原始数据，因为事务B取走了100块钱，并没有提交，数据库里面的账务余额肯定还是原始余额，这就是脏读。

- 幻读

  所谓幻读，就是指在一个事务里面的操作中发现了未被操作的数据。比如学生信息，事务A开启事务-->修改所有学生当天签到状况为false，此时切换到事务B，事务B开启事务-->事务B插入了一条学生数据，此时切换回事务A，事务A提交的时候发现了一条自己没有修改过的数据，这就是幻读，就好像发生了幻觉一样。幻读出现的前提是并发的事务中有事务发生了插入、删除操作。

- 不可重复读

  所谓不可重复读，就是指在一个事务里面读取了两次某个数据，读出来的数据不一致。还是以银行取钱为例，事务A开启事务-->查出银行卡余额为1000元，此时切换到事务B事务B开启事务-->事务B取走100元-->提交，数据库里面余额变为900元，此时切换回事务A，事务A再查一次查出账户余额为900元，这样对事务A而言，在同一个事务内两次读取账户余额数据不一致，这就是不可重复读。

SpringBoot 事务的使用

启动事务管理

> 首先使用注解 @EnableTransactionManagement 开启事务支持
>
> ```
> 1 @EnableTransactionManagement // 启注解事务管理，等同于xml配置方式的 <tx:annotation-driven />
> 2 @SpringBootApplication
> 3 public class ProfiledemoApplication {
> 4
> ```
>
> 在启动类似添加该注解

- # Service层引入事务

service逻辑引入事务
@Transantional(propagation=Propagation.REQUIRED)

Transantional相关属性

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

注意事项：

- service实现类(一般不建议在接口上)上添加@Transactional，可以将整个类纳入spring事务管理，在每个业务方法执行时都会开启一个事务，不过这些事务采用相同的管理方式。
- Transactional 注解只能应用到 public 可见度的方法上。 如果应用在protected、private或者 package可见度的方法上，也不会报错，不过事务设置不会起作用。
- **默认情况下**，Transactional 注解的事物所管理的方法中，如果方法抛出**运行时异常**或**error**，那么会进行事务回滚；如果方法抛出的是非运行时异常，那么不会回滚。

  注：

  SQL异常属于检查异常（有的框架将SQL异常重写为了运行时异常），但是有时我们写SQL时，检查异常并不会提示；而默认情况下，事物对检查异常不会作出回滚处理。

  注：

  在很多时候，我们除了catch一般的异常或自定义异常外，我们还习惯于catch住Exception异常；然后再抛出 Exception异常。

  但是Exception异常属于非运行时异常(即：检查异常)，因为默认是运行时异常时事物才进 行回滚，那么这种情况下，是不会回滚的。我们可以在@Transacional注解中，通过 rollbackFor = {Exception.class} 来解决这个问题。

  即：设置当Exception异常或Exception的所有任意子 类异常时事物会进行回滚。

  注：

  被catch处理了的异常，不会被事物作为判断依据；如果异常被catch了，但是又在catch中抛出了新的异 常，那么事物会以这个新的异常作为是否进行回滚的判断依据。

事务补充：

同一个事务里面，对某一条数据的增删改、 都会影响到这个事务里面接下来的对这个条数的增删改查，如(举例部分情况):

一个事务里面，debug未完成时， 数据会入库吗？	**不会**
一个事务里面，执行一半时，程序莫名停了，数据会回滚吗？	会
同一个事务里面，插入(数据a) -> 查询(数据a) -> 修改(数据a) -> 插入(数据a)，可以吗？	可以
同一个事务里面，插入(数据a) -> 修改(数据a) -> 再次修改(数据a) -> 查询(数据a)，可以吗？	可以
同一个事务里面，插入(数据a) -> 修改(数据a) -> 删除(数据a)，可以吗？	可以

阿里piapia规范推荐：

事务场景中，抛出异常被catch后，如果需要回滚，一定要手动回滚事务。

如：使用

```
@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private DataSourceTransactionManager transactionManager;

    @Override
    @Transactional
    public void save(User user) {
        DefaultTransactionDefinition def = new DefaultTransactionDefinition();
        // explicitly setting the transaction name is something that can only be done programmatically
        def.setName("SomeTxName");
        def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);

        TransactionStatus status = transactionManager.getTransaction(def);
        try {
            // execute your business logic here
            //db operation
        } catch (Exception ex) {
            transactionManager.rollback(status);
            throw ex;
        }
    }
}
```


替换

```java
@Service
public class UserServiceImpl implements UserService {

    @Override
    @Transactional(rollbackFor = {Exception.class})
    public void save(User user) {
        try {
            // execute your business logic here
            //db operation
        } catch (Exception ex) {
            throw ex;
        }
    }
}
```


阿里推荐的方式属于自动提交/手动回滚，那如果我们想要手动提交、手动回滚的话，可参考:

```java
@Service
public class SynchronizeDataServiceImpl implements SynchronizeDataService {

    private static final Logger LOGGER = LoggerFactory.getLogger(SynchronizeDataServiceImpl.class);

    /**
     * 自动装配 数据源事务管理器
     * 注:前提是容器中有可装配的事务管理器，如果没有那么需要注入
     */
    private final DataSourceTransactionManager transactionManager;

    public SynchronizeDataServiceImpl(DataSourceTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }

    @Override
    public String synchronizeData() {
        // 配置事务策略
        DefaultTransactionDefinition def = new DefaultTransactionDefinition();
        def.setName("planOne-transaction");
        def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);
        // 设置状态点
        TransactionStatus transactionStatus = transactionManager.getTransaction(def);
        try {
            // do something

            // 手动提交
            transactionManager.commit(transactionStatus);
        } catch (Exception e) {
            // 手动回滚
            transactionManager.rollback(transactionStatus);
        }
        return "";
    }
}
```


解决Transactional注解不回滚

1、检查你方法是不是public的

2、你的异常类型是不是unchecked异常 如果我想check异常也想回滚怎么办，注解上面写明异常类型即可

```
@Transactional(rollbackFor=Exception.class)
```

类似的还有norollbackFor，自定义不回滚的异常

3、数据库引擎要支持事务，如果是MySQL，注意表要使用支持事务的引擎，比如innodb，如果是myisam，事务是不起作用的

4、是否开启了对注解的解析

```
<tx:annotation-driven transaction-manager="transactionManager"
proxy-target-class="true"/>
```

5、spring是否扫描到你这个包，如下是扫描到org.test下面的包

```
<context:component-scan base-package="org.test" >
</context:component-scan>
```

参考链接

1： https://blog.csdn.net/justry_deng/article/details/80828180

2：http://www.voidcn.com/article/p-bozwatyd-st.html

3：https://www.ibm.com/developerworks/cn/java/j-master-spring-transactional-use/