**Spring Boot 中的事务处理**

**数据库事务简介**

数据库事务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，事务是数据库运行中的逻辑工作单位，由数据库中的事务管理子系统负责事务的处理。

**Spring Boot 事务机制**

**首先需要明确的一点是 Spring Boot 事务机制实质上就是 Spring 的事务机制**，是采用统一的机制处理来自不同数据访问技术的事务处理，只不过 Spring Boot 基于自动配置的特性作了部分处理来节省开发者的配置工作。

Spring 事务管理分两种方式：

* 编程式事务，指的是通过编码方式实现事务；
* 声明式事务，基于 AOP，将具体业务逻辑与事务处理解耦。

**声明式事务**

声明式事务是建立在 AOP 机制之上的，其本质是对方法前后进行拦截，然后在目标方法开始之前创建或者加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。

声明式事务最大的优点，就是通过 AOP 机制将具体业务逻辑与事务处理解耦，不需要通过编程的方式管理事务，这样就不需要在业务逻辑代码中掺杂事务管理的代码，因此在实际使用中声明式事务用的比较多。

声明式事务有两种方式：一种是在 XML 配置文件中做相关的事务规则声明；另一种是基于 @Transactional 注解的方式（@Transactional 注解是来自 org.springframework.transaction.annotation 包），便可以将事务规则应用到业务逻辑中。

**未使用 Spring Boot 时的事务配置**

 SSM 框架整合时的事务配置：

<!-- 事务管理 -->

<bean id="transactionManager"

class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"/>

</bean>

<!-- 配置事务通知属性 -->

<tx:advice id="txAdvice" transaction-manager="transactionManager">

<!-- 定义事务传播属性 -->

<tx:attributes>

<tx:method name="insert\*" propagation="REQUIRED"/>

<tx:method name="import\*" propagation="REQUIRED"/>

<tx:method name="update\*" propagation="REQUIRED"/>

<tx:method name="upd\*" propagation="REQUIRED"/>

<tx:method name="add\*" propagation="REQUIRED"/>

<tx:method name="set\*" propagation="REQUIRED"/>

<tx:method name="remove\*" propagation="REQUIRED"/>

<tx:method name="delete\*" propagation="REQUIRED"/>

<tx:method name="get\*" propagation="REQUIRED" read-only="true"/>

<tx:method name="\*" propagation="REQUIRED" read-only="true"/>

</tx:attributes>

</tx:advice>

<!-- 配置事务切面 -->

<aop:config>

<aop:pointcut id="serviceOperation"

expression="(execution(\* com.ssm.demo.service.\*.\*(..)))"/>

<aop:advisor advice-ref="txAdvice" pointcut-ref="serviceOperation"/>

</aop:config>

通过这段代码我们也能够看出声明式事务的配置过程：

1. 配置事务管理器
2. 配置事务通知属性
3. 配置事务切面

这样配置后，相关方法在执行时都被纳入事务管理下了，一旦发生异常，事务会正确回滚。

**Spring Boot 项目中的事务控制**

在 SpringBoot 中，建议采用注解 @Transactional 进行事务的控制，只需要在需要进行事务管理的方法或者类上添加 @Transactional 注解即可。

项目结构目录如下：

lou-springboot

├── pom.xml

├── README.md

└── src

├── main

│ ├── java

│ │ └── com

│ │ └── lou

│ │ └── springboot

│ │ ├── Application.java

│ │ ├── controller

│ │ │ ├── HelloController.java

│ │ │ ├── JdbcController.java

│ │ │ ├── MyBatisController.java

│ │ │ └── TransactionTestController.java

│ │ ├── dao

│ │ │ └── UserDao.java

│ │ ├── entity

│ │ │ └── User.java

│ │ └── service

│ │ └── TransactionTestService.java

│ └── resources

│ ├── application.properties

│ ├── mapper

│ │ └── UserDao.xml

│ └── templates

└── test

└── java

└── com

└── lou

└── springboot

└── ApplicationTests.java

**新建 TransactionTestService.java**

首先新建 service 包作为业务代码包，事务处理一般在 service 层做，当然在 controller 层中处理也可以，但是建议还是在业务层进行处理，之后在包中新建 TransactionTestService 类，代码如下：

@Service  
public class TransactionTestService *{* @Resource  
 UsersMapper userdao;  
  
 public boolean test1*()  
 {* Users users = new Users*()*;  
 users.setPassword*(*"1002"*)*;  
 users.setName*(*"test1"*)*;  
 userdao.insertSelective*(*users*)*;  
 //发生异常  
 System.*out*.println*(*1/0*)*;  
 return true;  
 *}* @Transactional  
 public boolean test2*()  
 {* Users users = new Users*()*;  
 users.setPassword*(*"1002"*)*;  
 users.setName*(*"test2"*)*;  
 userdao.insertSelective*(*users*)*;  
 //发生异常  
 System.*out*.println*(*1/0*)*;  
 return true;  
 *}  
}*

首先在类上添加 @Service 将其注册到 IOC 容器中管理，之后注入 **UsersMapper**对象以实现后续的数据层操作，最后实现两个业务方法 test1() 和 test2()，二者实现类似，只是两个方法添加的用户对象名称和密码字符串不同，且 test2() 方法上添加了 @Transactional 注解，而 test1() 方法上并没有添加，在方法中我们都添加了一句代码，让数字 1 去除以 数字 0，这段代码一定会出现异常，我们用这个来模拟在发生异常时事务处理能否成功。

按照正常理解，在执行 SQL 语句后，一旦发生异常，这次数据库更改一定会被事务进行回滚，正常情况下数据库中会有 test1 的数据而没有 test2 的数据，**因为 test1() 方法并没有纳入事务管理中，而 test2() 方法由于加上了 @Transactional 注解是会被事务管理器处理的。**

**新建 TransactionTestController.java**

@Controller  
public class TransactionTestController *{* @Resource  
 private TransactionTestService transactionTestService;  
  
 @GetMapping*(*"/transaction"*)* @ResponseBody  
 //ResponseBody的作用其实是将java对象转为json格式的数据。  
 public String Transaction*()  
 {* //test1方法，未添加@Transactional  
 transactionTestService.test1*()*;  
 //test2方法，添加了注解Transactional  
 transactionTestService.test2*()*;  
 return "完成";  
 *}*

实现一个控制器方法处理 /transactionTest 请求，方法中会进行业务层 test1() 方法和 test2() 方法的调用，也就是说在项目启动后一旦访问 /transactionTest 请求就能够看到这次事务处理测试的结果了。

test1() 方法由于没有添加 @Transactional 注解，所以在发生异常后事务没有回滚，insert 语句依然起到了作用，而 test2() 方法添加了 @Transactional 注解，所以在发生异常后事务正常回滚，数据库中并没有 test2 的记录！

**Spring Boot 事务管理器自动配置**

声明式事务的配置过程，首先需要配置事务管理器，但是我们在开发时并没有进行该管理器的配置但是事务管理却起到了作用，这是为什么呢？

答案是 Spring Boot 在启动过程中已经将该对象自动配置完成了，所以我们在 Spring Boot 项目中可以直接使用 @Transactional 注解来处理事务，感兴趣的同学可以查看源码进行学习，事务管理器的自动配置类如下：

* org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration
* org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration

可以在你本地开发工具 (IDEA 或者 Eclipse) 中搜索这两个类的源码来看。

**@Transactional 事务实现机制**

**@Transactional 不仅可以注解在方法上，也可以注解在类上。当注解在类上时，意味着此类的所有 public 方法都是开启事务的。如果类级别和方法级别同时使用了 @Transactional 注解，则使用在类级别的注解会重载方法级别的注解。**

在应用系统调用声明了 @Transactional 的目标方法时，Spring Framework 默认使用 AOP 代理，在代码运行时生成一个代理对象，根据 @Transactional 的属性配置信息，这个代理对象决定该声明 @Transactional 的目标方法是否由拦截器 TransactionInterceptor 来使用拦截，在 TransactionInterceptor 拦截时，会在目标方法开始执行之前创建并加入事务，并执行目标方法的逻辑, 最后根据执行情况是否出现异常，利用抽象事务管理器 AbstractPlatformTransactionManager 操作数据源 DataSource 提交或回滚事务。