## Spring事务使用

### 事务基本特性

#### 原子性（Atomicity）

　　原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

#### 一致性（Consistency）

　一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

　　拿转账来说，假设用户A和用户B两者的钱加起来一共是5000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是5000，这就是事务的一致性。

#### 隔离性（Isolation）

　　隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

　　即要达到这么一种效果：对于任意两个并发的事务T1和T2，在事务T1看来，T2要么在T1开始之前就已经结束，要么在T1结束之后才开始，这样每个事务都感觉不到有其他事务在并发地执行。

　　关于事务的隔离性数据库提供了多种隔离级别，稍后会介绍到。

#### 持久性（Durability）

持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

例如我们在使用JDBC操作数据库时，在提交事务方法后，提示用户事务操作完成，当我们程序执行完成直到看到提示后，就可以认定事务以及正确提交，即使这时候数据库出现了问题，也必须要将我们的事务完全执行完成，否则就会造成我们看到提示事务处理完毕，但是数据库因为故障而没有执行事务的重大错误。

### 事务控制分类

#### 编程式事务控制

自己手动控制事务，就叫做编程式事务控制。

Jdbc代码：

Conn.setAutoCommite(false); // 设置手动控制事务

Hibernate代码：

Session.beginTransaction(); // 开启一个事务

【细粒度的事务控制： 可以对指定的方法、指定的方法的某几行添加事务控制】

(比较灵活，但开发起来比较繁琐： 每次都要开启、提交、回滚.)

#### 声明式事务控制

Spring提供了对事务的管理, 这个就叫声明式事务管理。

Spring提供了对事务控制的实现。用户如果想用Spring的声明式事务管理，只需要在配置文件中配置即可； 不想使用时直接移除配置。这个实现了对事务控制的最大程度的解耦。

Spring声明式事务管理，核心实现就是基于Aop。

【粗粒度的事务控制： 只能给整个方法应用事务，不可以对方法的某几行应用事务。】

(因为aop拦截的是方法。)

Spring声明式事务管理器类：

Jdbc技术：DataSourceTransactionManager

Hibernate技术：HibernateTransactionManager

# 手写Spring事务框架

## 编程事务实现

### 概述

所谓编程式事务指的是通过编码方式实现事务，即类似于JDBC编程实现事务管理。管理使用TransactionTemplate或者直接使用底层的PlatformTransactionManager。对于编程式事务管理，spring推荐使用TransactionTemplate。

### 案例

#### 使用编程事务实现手动事务

使用编程事务实现，手动事务 begin、commit、rollback

|  |
| --- |
| @Component  **public** **class** TransactionUtils {  @Autowired  **private** DataSourceTransactionManager dataSourceTransactionManager;  // 开启事务  **public** TransactionStatus begin() {  TransactionStatus transaction = dataSourceTransactionManager.getTransaction(**new** DefaultTransactionAttribute());  **return** transaction;  }  // 提交事务  **public** **void** commit(TransactionStatus transactionStatus) {  dataSourceTransactionManager.commit(transactionStatus);  }  // 回滚事务  **public** **void** rollback(TransactionStatus transactionStatus) {  dataSourceTransactionManager.rollback(transactionStatus);  }  }  @Service  **public** **class** UserService {  @Autowired  **private** UserDao userDao;  @Autowired  **private** TransactionUtils transactionUtils;  **public** **void** add() {  TransactionStatus transactionStatus = **null**;  **try** {  transactionStatus = transactionUtils.begin();  userDao.add("wangmazi", 27);  **int** i = 1 / 0;  System.***out***.println("我是add方法");  userDao.add("zhangsan", 16);  transactionUtils.commit(transactionStatus);  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **if** (transactionStatus != **null**) {  transactionStatus.rollbackToSavepoint(transactionStatus);  }  }  }  } |

#### AOP技术封装手动事务

|  |
| --- |
| @Component  @Aspect  **public** **class** AopTransaction {  @Autowired  **private** TransactionUtils transactionUtils;  // // 异常通知  @AfterThrowing("execution(\* com.itmayiedu.service.UserService.add(..))")  **public** **void** afterThrowing() {  System.***out***.println("程序已经回滚");  // 获取程序当前事务 进行回滚  TransactionAspectSupport.*currentTransactionStatus*().setRollbackOnly();  }  // 环绕通知  @Around("execution(\* com.itmayiedu.service.UserService.add(..))")  **public** **void** around(ProceedingJoinPoint proceedingJoinPoint) **throws** Throwable {  System.***out***.println("开启事务");  TransactionStatus begin = transactionUtils.begin();  proceedingJoinPoint.proceed();  transactionUtils.commit(begin);  System.***out***.println("提交事务");  }  } |

#### 使用事务注意事项

事务是程序运行如果没有错误,会自动提交事物,如果程序运行发生异常,则会自动回滚。

如果使用了try捕获异常时.一定要在catch里面手动回滚。

事务手动回滚代码

TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();

## 声明事务实现

### 概述

管理建立在AOP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，只需在配置文件中做相关的事务规则声明(或通过基于@Transactional注解的方式)，便可以将事务规则应用到业务逻辑中。

显然声明式事务管理要优于编程式事务管理，这正是spring倡导的非侵入式的开发方式。

声明式事务管理使业务代码不受污染，一个普通的POJO对象，只要加上注解就可以获得完全的事务支持。和编程式事务相比，声明式事务唯一不足地方是，后者的最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别。但是即便有这样的需求，也存在很多变通的方法，比如，可以将需要进行事务管理的代码块独立为方法等等。

### XML实现声明

### 注解版本声明

|  |
| --- |
| <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop.xsd*  *http://www.springframework.org/schema/tx*  *http://www.springframework.org/schema/tx/spring-tx.xsd"*>  <!-- 开启注解 -->  <context:component-scan base-package=*"com.itmayiedu"*></context:component-scan>  <!-- 1. 数据源对象: C3P0连接池 -->  <bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>  <property name=*"driverClass"* value=*"com.mysql.jdbc.Driver"*></property>  <property name=*"jdbcUrl"* value=*"jdbc:mysql://localhost:3306/test"*></property>  <property name=*"user"* value=*"root"*></property>  <property name=*"password"* value=*"root"*></property>  </bean>  <!-- 2. JdbcTemplate工具类实例 -->  <bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>  <property name=*"dataSource"* ref=*"dataSource"*></property>  </bean>  <!-- 配置事物 -->  <bean id=*"dataSourceTransactionManager"*  class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>  <property name=*"dataSource"* ref=*"dataSource"*></property>  </bean>  <!-- 开启注解事物 -->  <tx:annotation-driven transaction-manager=*"dataSourceTransactionManager"* />  </beans> |

用法

|  |
| --- |
| @Transactional  **public** **void** add() {  userDao.add("wangmazi", 27);  **int** i = 1 / 0;  System.***out***.println("我是add方法");  userDao.add("zhangsan", 16);  } |

## 手写Spring注解版本事务

### 注解

Jdk1.5新增新技术，注解。很多框架为了简化代码，都会提供有些注解。可以理解为插件，是代码级别的插件，在类的方法上写：@XXX，就是在代码上插入了一个插件。

注解不会也不能影响代码的实际逻辑，仅仅起到辅助性的作用。

**注解分类：内置注解(也成为元注解 jdk 自带注解)、自定义注解（Spring框架）**

##### 什么是内置注解

（1） @SuppressWarnings   再程序前面加上可以在javac编译中去除警告--阶段是SOURCE  
（2） @Deprecated   带有标记的包，方法，字段说明其过时----阶段是SOURCE  
（3）@Overricle   打上这个标记说明该方法是将父类的方法重写--阶段是SOURCE

###### @Overricle 案例演示

|  |
| --- |
| **@Override**  **public String toString() {**  **return null;**  **}** |

###### @Deprecated案例演示

|  |
| --- |
| **new Date().parse("");** |

###### @SuppressWarnings  案例演示

|  |
| --- |
| @SuppressWarnings({ "all" })  **public** **void** save() {  java.util.List list = **new** ArrayList();  } |

##### 实现自定义注解

元注解的作用就是负责注解其他注解。Java5.0定义了4个标准的meta-annotation类型，它们被用来提供对其它 annotation类型作说明。Java5.0定义的元注解：  
[@Target](mailto:1.@Target)

@Target说明了Annotation所修饰的对象范围：Annotation可被用于 packages、types（类、接口、枚举、Annotation类型）、类型成员（方法、构造方法、成员变量、枚举值）、方法参数和本地变量（如循环变量、catch参数）。在Annotation类型的声明中使用了target可更加明晰其修饰的目标。

1. CONSTRUCTOR:用于描述构造器
2. FIELD:用于描述域
3. LOCAL\_VARIABLE:用于描述局部变量
4. METHOD:用于描述方法
5. PACKAGE:用于描述包
6. PARAMETER:用于描述参数
7. TYPE:用于描述类、接口(包括注解类型) 或enum声明

[**2.@Retention**](mailto:2.@Retention)

表示需要在什么级别保存该注释信息，用于描述注解的生命周期（即：被描述的注解在什么范围内有效）  
**3.@Documented  
4.@Inherited**

使用@interface 定义注解。

|  |
| --- |
| @Target(value = { ElementType.***METHOD*** })  @Retention(RetentionPolicy.***RUNTIME***)  **public** **@interface** AddAnnotation {  **int** userId() **default**  **t** 0;  String userName() **default** "默认名称";  String[]arrays();  }  反射读取注解信息  **public** **static** **void** main(String[] args) **throws** ClassNotFoundException {  Class classInfo = Class.*forName*("com.itmayiedu.entity.User");  // 获取到所有方法  Method[] methods = classInfo.getDeclaredMethods();  **for** (Method method : methods) {  System.***out***.println(method);  AddAnnotation declaredAnnotation = method.getDeclaredAnnotation(AddAnnotation.**class**);  **if** (declaredAnnotation == **null**) {  // 结束本次循环  **continue**;  }  // 获取userId  **int** userId = declaredAnnotation.userId();  System.***out***.println("userId:" + userId);  // 获取userName  String userName = declaredAnnotation.userName();  System.***out***.println("userName:" + userName);  // 获取arrays  String[] arrays = declaredAnnotation.arrays();  **for** (String str : arrays) {  System.***out***.println("str:" + str);  }  }  } |

### 自定义事务注解

|  |
| --- |
| //编程事务（需要手动begin 手动回滚 手都提交）  @Component()  @Scope("prototype") // 设置成原型解决线程安全  **public** **class** TransactionUtils {  **private** TransactionStatus transactionStatus;  // 获取事务源  @Autowired  **private** DataSourceTransactionManager dataSourceTransactionManager;  // 开启事务  **public** TransactionStatus begin() {  transactionStatus = dataSourceTransactionManager.getTransaction(**new** DefaultTransactionAttribute());  **return** transactionStatus;  }  // 提交事务  **public** **void** commit(TransactionStatus transaction) {  dataSourceTransactionManager.commit(transaction);  }  // 回滚事务  **public** **void** rollback() {  System.***out***.println("rollback");  dataSourceTransactionManager.rollback(transactionStatus);  }  }  **注解类**  @Autowired  **private** TransactionUtils transactionUtils;  @AfterThrowing("execution(\* com.itmayiedu.service.\*.\*.\*(..))")  **public** **void** afterThrowing() **throws** NoSuchMethodException, SecurityException {  // isRollback(proceedingJoinPoint);  System.***out***.println("程序发生异常");  // TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();  // TransactionStatus currentTransactionStatus =  // TransactionAspectSupport.currentTransactionStatus();  // System.out.println("currentTransactionStatus:" +  // currentTransactionStatus);  transactionUtils.rollback();  }  // // 环绕通知 在方法之前和之后处理事情  @Around("execution(\* com.itmayiedu.service.\*.\*.\*(..))")  **public** **void** around(ProceedingJoinPoint proceedingJoinPoint) **throws** Throwable {  // 调用方法之前执行  TransactionStatus transactionStatus = begin(proceedingJoinPoint);  proceedingJoinPoint.proceed();// 代理调用方法 注意点： 如果调用方法抛出异常不会执行后面代码  // 调用方法之后执行  commit(transactionStatus);  }  **public** TransactionStatus begin(ProceedingJoinPoint pjp) **throws** NoSuchMethodException, SecurityException {  // // 判断是否有自定义事务注解  ExtTransaction declaredAnnotation = getExtTransaction(pjp);  **if** (declaredAnnotation == **null**) {  **return** **null**;  }  // 如果有自定义事务注解，开启事务  System.***out***.println("开启事务");  TransactionStatus transactionStatu = transactionUtils.begin();  **return** transactionStatu;  }  **public** **void** commit(TransactionStatus transactionStatu) {  **if** (transactionStatu != **null**) {  // 提交事务  System.***out***.println("提交事务");  transactionUtils.commit(transactionStatu);  }  }  **public** ExtTransaction getExtTransaction(ProceedingJoinPoint pjp) **throws** NoSuchMethodException, SecurityException {  // 获取方法名称  String methodName = pjp.getSignature().getName();  // 获取目标对象  Class<?> classTarget = pjp.getTarget().getClass();  // 获取目标对象类型  Class<?>[] par = ((MethodSignature) pjp.getSignature()).getParameterTypes();  // 获取目标对象方法  Method objMethod = classTarget.getMethod(methodName, par);  // // 判断是否有自定义事务注解  ExtTransaction declaredAnnotation = objMethod.getDeclaredAnnotation(ExtTransaction.**class**);  **if** (declaredAnnotation == **null**) {  System.***out***.println("您的方法上,没有加入注解!");  **return** **null**;  }  **return** declaredAnnotation;  }  // 回滚事务  **public** **void** isRollback(ProceedingJoinPoint pjp) **throws** NoSuchMethodException, SecurityException {  // // 判断是否有自定义事务注解  ExtTransaction declaredAnnotation = getExtTransaction(pjp);  **if** (declaredAnnotation != **null**) {  System.***out***.println("已经开始回滚事务");  // 获取当前事务 直接回滚  TransactionAspectSupport.*currentTransactionStatus*().setRollbackOnly();  **return**;  }  }  **使用自定义注解**  @ExtTransaction  **public** **void** add() {  userDao.add("test001", 20);  **int** i = 1 / 0;  System.***out***.println("################");  userDao.add("test002", 21);  } |

## Spring事物传播行为

Spring中事务的定义：

Propagation（key属性确定代理应该给哪个方法增加事务行为。这样的属性最重要的部份是传播行为。）有以下选项可供使用：

PROPAGATION\_REQUIRED—如果当前有事务，就用当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。

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。//如果外层方法没有事务，就会以非事务进行执行。

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。

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。

--- 如果当前有事务，就是以非事务进行执行

PROPAGATION\_NEVER--以非事务方式执行，如果当前存在事务，则抛出异常。

默认传播行为为REQUIRED