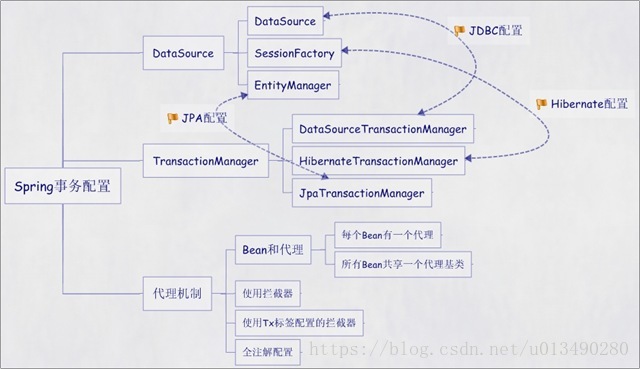
<https://blog.csdn.net/u013490280/article/details/82598097?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.nonecase&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.nonecase>



Spring事务源码阅读梳理（建议珍藏）

当你面对一个完全不熟悉的事物时，一定要想办法找到一个突破口，然后逐步深入。

那Spring事物的突破口在哪里呢？很明显在@EnableTransactionManagement注解里，因为是它启用了事物功能。

@EnableTransactionManagement

发现注解还引入了一个类TransactionManagementConfigurationSelector。

class TransactionManagementConfigurationSelector

发现如果采用代理的方式时，又引入了一个类ProxyTransactionManagementConfiguration。

class ProxyTransactionManagementConfiguration

发现这个类往容器中注册了3个bean

/\*\*/

BeanFactoryTransactionAttributeSourceAdvisor。它以Advisor结尾说明它是Spring AOP范畴里的东西。

在AOP里，Advisor = Pointcut + Advice，Pointcut是切入点，表示要拦截的方法，Advice是增强，表示要加进去的事物功能。

TransactionAttributeSourcePointcut类以Pointcut结尾，说明它是一个切入点，就是标识要被拦截的方法。类名的前缀部分表明了这个切入点的实现原理。

这个源可以向外提供事务属性，其实就是判断一个类的方法上是否标有@Transactional注解，如果有的话还可以获取这个注解的属性（即事务属性）。

整体来说就是，Pointcut拦截住了方法，然后使用这个“源”去方法和类上获取事务属性，如果能获取到，说明此方法需要参与事务，则进行事务增强，反之则不增强。

/\*切面实现\*/

TransactionInterceptor就是一个Advice，因为它实现了Advice接口，包含了把事物加进去的逻辑。

Advice就是AOP中的增强，TransactionInterceptor实现了Advice接口，所以它就是事务增强。

Pointcut是切入点，表示要拦截的方法。它是一个静态的概念，即程序不运行时它也是存在的。

那么在真正运行时，已经拦截住了，此时该怎么表示这个情况呢？是用Joinpoint来表示的，所以Joinpoint是一个运行时的概念，只有在运行时才存在

Joinpoint

第一个方法proceed()是“继续”的意思，调用它表示去执行被拦截住的方法本身，返回方法本身的返回值。

第二个方法getThis()是获取this对象，即方法运行时所在的目标对象。如果是静态方法，则为null，因为静态方法是属于类本身的，运行时不需要对象。

第三个方法getStaticPart()，其实就表示了被拦截住的方法，即就是一个Method。Method其实算是“元数据”，是属于类型本身的，也有“静态”的意思。

Invocation extend Jointpoint

方法getArguments()就表示运行时传递给被拦截住方法的参数。

再看一个接口，MethodInvocation，它继承了Invocation

MethodInvocation extends Invocation

方法getMethod()返回一个Method，它就是当前正在执行的方法，是对本拦截方法的一个友好实现，返回相同的结果。

可见MethodInvocation接口已经包含了一个方法调用的全量信息，方法，参数，目标对象。这其实就是运行时被拦截住的东西。

再看下面这个接口，MethodInterceptor，方法拦截器

interface MethodInterceptor extend Intercepor

它只有一个方法invoke，方法参数就是上面介绍的MethodInvocation。所以拦截器可以使用这个参数来对目标方法进行调用，当然在调用前/后可以加入自己的逻辑。

TransactionInterceptor类就实现了这个接口，因此可以在对目标方法的调用前后插入事务逻辑代码来进行事务增强。

\\\\下面是事务拦截器对该方法的实现

invoke(MethodInterceptor invocation)

invokeWithinTransaction(invocation.getMethod(),,invocation::proceed)

它调用的invokeWithinTransaction方法是在父类里的

前两行获取事务属性“源”，再用这个“源”来获取事务属性。咦，有点奇怪，上面不是已经获取过了吗？是的，上面是在Pointcut里获取的，那只是用于判断那个方法是否要被拦截而已。这里获取的属性才是真正用于事务的。

第三行是根据事务属性，来确定出一个事务管理器来。

接下来是使用事务管理器打开事务。

接下来是对被拦截住的目标方法的调用执行，当然要try/catch住这个执行。

如果抛出了异常，则进行和异常相关的事务处理，然后将这个异常继续向上抛出。

如果没有抛出异常，则进行事务提交。

最后的else分支是对编程式事务的调用，事务的打开/提交/回滚是开发人员自己写代码控制，所以就不需要事务管理器操心了。

/\*\*/

TransactionAttributeSource虽然不是一个Pointcut，但是它被Pointcut所用，用于检测一个类的方法上是否有@Transactional注解，来确定该方法是否需要事物增强。

现在这个“源”还是个黑盒子，下面来揭开它的面纱。它的实现类是AnnotationTransactionAttributeSource，以Annotation开头，说明是基于注解实现的。

AnnotationTransactionAttributeSource

第一个方法从类上找事务属性，第二个方法从方法上找事务属性，它俩都调用了第三个方法来实现。

第三个方法使用多个事务注解解析器（TransactionAnnotationParser）去解析注解，为啥是多个解析器呢？因为事务注解不仅Spring提供了，Java后来也提供了，就是javax.transaction.Transactional。

Spring对自己注解的解析器实现类是SpringTransactionAnnotationParser，

SpringTransactionAnnotationParser

使用工具类来读取注解@Transactional的属性，然后逐个解析出属性值并进行类型转换，接着把这些属性封装到一个类里，这个类其实就是事务属性，即TransactionAttribute。

/\*事务处理实现\*/

interface PlatformTransactionManager

PS：我们都知道，方法上的注解优先级高于类上的，是因为找注解时先找方法上的，找不到时再去类上找。所以方法上的优先级高。此部分代码逻辑在父类里写着呢，这里不再展示了。

@Service

public class MyTsA {

@Resource

private UserServiceImpl userService;

@Resource

private MyTsB myTsB;

@Transactional(rollbackFor = Exception.class)

public void testNewRollback(){

User user = new User();

user.setId(1);

user.setName("张三");

userService.save(user);

myTsB.save();

}

}

@Service

public class MyTsB {

@Resource

private UserServiceImpl userService;

@Transactional(rollbackFor = RuntimeException.class, propagation = ???)

public void save(){

User user = new User();

user.setId(2);

user.setName(“李四”)

userService.save(user);

}

如上面所示有两个事务A和事务B，事务A方法中调用了事务B方法，区分两种回滚情况。

A提交，B回滚。

A回滚，B提交。

对于不同的传播级别：

1.当传播级别为为REQUIRED时，第一种情况A和B都会回滚。

2.。当传播级别为RQUIRED\_NEW时，第一种情况A可以成功提交，B回滚，第二种情况A回滚，B可以正确提交，二者互不影响。（在A事务捕获B事务异常的情况下，保证A事务提交）

3.当传播级别为NESTED时，第一种情况A可以正常提交，B回滚，第二种情况二者都会回滚。

传播级别

介绍

Spring事务的传播级别描述的是多个使用了@Transactional注解的方法互相调用时，Spring对事务的处理。包涵的传播级别有：

REQUIRED, 如果当前线程已经在一个事务中，则加入该事务，否则新建一个事务。

SUPPORT, 如果当前线程已经在一个事务中，则加入该事务，否则不使用事务。

MANDATORY(强制的)，如果当前线程已经在一个事务中，则加入该事务，否则抛出异常。

REQUIRES\_NEW，无论如何都会创建一个新的事务，如果当前线程已经在一个事务中，则挂起当前事务，创建一个新的事务。

NOT\_SUPPORTED，如果当前线程在一个事务中，则挂起事务。

NEVER，如果当前线程在一个事务中则抛出异常。

NESTED, 执行一个嵌套事务，有点像REQUIRED，但是有些区别，在Mysql中是采用SAVEPOINT来实现的。

隔离级别对开始事务的影响(获取TransactionStatus)

public final TransactionStatus getTransaction(@Nullable TransactionDefinition definition) throws TransactionException {

private TransactionStatus handleExistingTransaction(

可以看到，如果当前不存在事务时，创建一个新的TransactionStatus对象，否则进入到handleExistingTransaction。下面来看这个方法

1.如果是NEVER传播级别则抛出异常

2. 如果是不支持，则挂起事务，将当前事务对象设置为null,newTransaction设置为false，把线程的相关Threadlocal变量改的就像当前不存在事务一样

3. 如果是required\_NEW的话，则挂起当前事务，同时创建一个新的事务，执行doBegin操作，设置newTransaction为true

4.如果是嵌入事务，则创建一个SAVEPOINT

5.对于REQUIRED传播级别会把newTransaction标志位设置为false

隔离级别对回滚事务的影响

private void processRollback(DefaultTransactionStatus status, boolean unexpected) {

如果newTransaction设置为ture，则真正执行回滚。

如果有保存点，则回滚到保存点。

否则不会真正回滚，而是设置回滚标志位。

隔离级别对提交事务的影响

@Override

public final void commit(TransactionStatus status) throws TransactionException {

就算事务方法没有抛出异常，走到了commit方法中，但是依然有可能回滚事务。

对于REQUIRED传播级别，即使父事务中没有抛出异常，但是子事务中已经设置了回滚标志，那么父事务依然会回滚

只有newTransaction标志位为true的事务才会真正执行commit操作。

为什么我的事务不生效

1.如果不是Innodb存储引擎，MyISAM不支持事务。

2.没有指定rollbackFor参数。

3. 没有指定transactionManager参数，默认的transactionManager并不是我期望的，以及一个事务中涉及到了多个数据库。

4. 如果AOP使用了JDK动态代理，对象内部方法互相调用不会被Spring的AOP拦截，@Transactional注解无效。

5. 如果AOP使用了CGLIB代理，事务方法或者类不是public，无法被外部包访问到，或者是final无法继承，@transactional注解无效。