Spring 事务控制注意点介绍

# 编写背景

近期在使用atlas读写分离代理的过程中，经常出现读取的数据不完整的情况。在定位的过程中，发现不使用atlas，直接连接主库并没有此问题。因此，怀疑是atlas存在问题，但是负责运维的大神缺怀疑是研发编写的代码存在问题，于是两方面入手排查问题，一方面从atlas日志排查，一方面从代码排查。

从atlas日志查看发现，出现大量的set aotucommit=1和set aotucommit=0操作，即使是读操作，前后也有此操作，运维建议将读操作不进行事务控制。于是从代码层面设置事务传播属性，将包含读操作的服务代码事务传播属性设置为NOT\_SUPPORTED，于是问题“暂时解决”。

之所以称为暂时解决，是由于这没有从根本上解决问题，为什么atlas在读操作有事务控制时读取的数据不完整，于是换了一个官方的代理（maxscale），再也没有出现问题。到此，问题解决。

从排查问题的过程中，有个问题非常值得反思：为什么我们代码的事务控制没有按照预想的事务设置进行控制，出现大量读操作也加事务控制？

# Spring 事务

## 事务的基本原理

在回答第一节提出的问题之前，我们先详细解释spring 事务的概念。关于事务，觉得很神秘，根本原因是事务控制被spring框架完美控制，用户已经很少需要关注事务的细节方面。其实，Spring框架设计的事务控制是非常完美的，出现问题，是错误使用事务导致。

Spring事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，spring是无法提供事务功能的。对于纯JDBC操作数据库，想要用到事务，可以按照以下步骤进行：  
    1）获取连接 Connection con = DriverManager.getConnection()  
    2）开启事务con.setAutoCommit(true/false);  
    3）执行CRUD  
    4）提交事务/回滚事务 con.commit() / con.rollback();  
    5）关闭连接 conn.close();  
使用Spring的事务管理功能后，我们可以不再写步骤 2 和 4 的代码，而是由Spirng 自动完成。 那么Spring是如何在我们书写的 CRUD 之前和之后开启事务和关闭事务的呢？解决这个问题，也就可以从整体上理解Spring的事务管理实现原理了。下面简单地介绍下，注解方式为例子  
    配置文件开启注解驱动，在相关的类和方法上通过注解@Transactional标识。  
    spring 在启动的时候会去解析生成相关的bean，这时候会查看拥有相关注解的类和方法，并且为这些类和方法生成代理，并根据@Transaction的相关参数进行相关配置注入，这样就在代理中为我们把相关的事务处理掉了（开启正常提交事务，异常回滚事务）。

## Spring事务的传播属性

所谓spring事务的传播属性，就是在多个事务同时存在的时候，spring应该如何处理这些事务的行为。这些属性在TransactionDefinition中定义，具体常量的解释如下：

* ROPAGATION\_REQUIRED: 支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择，也是 Spring 默认的事务的传播。
* PROPAGATION\_REQUIRES\_NEW:  新建事务，如果当前存在事务，把当前事务挂起。新建的事务将和被挂起的事务没有任何关系，是两个独立的事务，外层事务失败回滚之后，不能回滚内层事务执行的结果，内层事务失败抛出异常，外层事务捕获，也可以不处理回滚操作。
* 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\_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。
* PROPAGATION\_NESTED: 如果一个活动的事务存在，则运行在一个嵌套的事务中。如果没有活动事务，则按REQUIRED属性执行。它使用了一个单独的事务，这个事务拥有多个可以回滚的保存点。内部事务的回滚不会对外部事务造成影响。它只对DataSourceTransactionManager事务管理器起效。

## 隔离级别

### 数据库隔离级别

|  |  |  |
| --- | --- | --- |
| 隔离级别 | 隔离级别的值 | 导致的问题 |
| Read-Uncommitted | 0 | 导致脏读 |
| Read-Committed | 1 | 避免脏读，允许不可重复读和幻读 |
| Repeatable-Read | 2 | 避免脏读，不可重复读，允许幻读 |
| Serializable | 3 | 串行化读，事务只能一个一个执行，避免了脏读、不可重复读、幻读。执行效率慢，使用时慎重 |

**脏读**：一事务对数据进行了增删改，但未提交，另一事务可以读取到未提交的数据。如果第一个事务这时候回滚了，那么第二个事务就读到了脏数据。  
**不可重复读**：一个事务中发生了两次读操作，第一次读操作和第二次操作之间，另外一个事务对数据进行了修改，这时候两次读取的数据是不一致的。  
**幻读**：第一个事务对一定范围的数据进行批量修改，第二个事务在这个范围增加一条数据，这时候第一个事务就会丢失对新增数据的修改。  
总结：  
隔离级别越高，越能保证数据的完整性和一致性，但是对并发性能的影响也越大。  
大多数的数据库默认隔离级别为 Read Commited，比如 SqlServer、Oracle  
少数数据库默认隔离级别为：Repeatable Read 比如： MySQL InnoDB

### Spring中的隔离级别

|  |  |
| --- | --- |
| 常量 | 解释 |
| ISOLATION\_DEFAULT | 这是个 PlatfromTransactionManager 默认的隔离级别，使用数据库默认的事务隔离级别。另外四个与 JDBC 的隔离级别相对应。 |
| ISOLATION\_READ\_UNCOMMITTED | 这是事务最低的隔离级别，它充许另外一个事务可以看到这个事务未提交的数据。这种隔离级别会产生脏读，不可重复读和幻读。 |
| ISOLATION\_READ\_COMMITTED | 保证一个事务修改的数据提交后才能被另外一个事务读取。另外一个事务不能读取该事务未提交的数据。 |
| ISOLATION\_REPEATABLE\_READ | 这种事务隔离级别可以防止脏读，不可重复读。但是可能出现幻读。 |
| ISOLATION\_SERIALIZABLE | 这是花费最高代价但是最可靠的事务隔离级别。事务被处理为顺序执行。 |

## 事务嵌套

通过上面的理论知识的铺垫，我们大致知道了数据库事务和spring事务的一些属性和特点，接下来我们通过分析一些嵌套事务的场景，来深入理解spring事务传播的机制。

假设外层事务 Service A 的 Method A() 调用 内层Service B 的 Method B()。

**PROPAGATION\_REQUIRED**(spring 默认)

如果ServiceB.methodB() 的事务级别定义为 PROPAGATION\_REQUIRED，那么执行 ServiceA.methodA() 的时候spring已经起了事务，这时调用 ServiceB.methodB()，ServiceB.methodB() 看到自己已经运行在 ServiceA.methodA() 的事务内部，就不再起新的事务。

假如 ServiceB.methodB() 运行的时候发现自己没有在事务中，他就会为自己分配一个事务。

这样，在 ServiceA.methodA() 或者在 ServiceB.methodB() 内的任何地方出现异常，事务都会被回滚。

**PROPAGATION\_REQUIRES\_NEW**

比如我们设计 ServiceA.methodA() 的事务级别为 PROPAGATION\_REQUIRED，ServiceB.methodB() 的事务级别为 PROPAGATION\_REQUIRES\_NEW。

那么当执行到 ServiceB.methodB() 的时候，ServiceA.methodA() 所在的事务就会挂起，ServiceB.methodB() 会起一个新的事务，等待 ServiceB.methodB() 的事务完成以后，它才继续执行。

他与 PROPAGATION\_REQUIRED 的事务区别在于事务的回滚程度了。因为 ServiceB.methodB() 是新起一个事务，那么就是存在两个不同的事务。如果 ServiceB.methodB() 已经提交，那么 ServiceA.methodA() 失败回滚，ServiceB.methodB() 是不会回滚的。如果 ServiceB.methodB() 失败回滚，如果他抛出的异常被 ServiceA.methodA() 捕获，ServiceA.methodA() 事务仍然可能提交(主要看B抛出的异常是不是A会回滚的异常)。

**PROPAGATION\_SUPPORTS**

假设ServiceB.methodB() 的事务级别为 PROPAGATION\_SUPPORTS，那么当执行到ServiceB.methodB()时，如果发现ServiceA.methodA()已经开启了一个事务，则加入当前的事务，如果发现ServiceA.methodA()没有开启事务，则自己也不开启事务。这种时候，内部方法的事务性完全依赖于最外层的事务。

**PROPAGATION\_NESTED**

现在的情况就变得比较复杂了, ServiceB.methodB() 的事务属性被配置为 PROPAGATION\_NESTED, 此时两者之间又将如何协作呢?  ServiceB#methodB 如果 rollback, 那么内部事务(即 ServiceB#methodB) 将回滚到它执行前的 SavePoint 而外部事务(即 ServiceA#methodA) 可以有以下两种处理方式:  
a、捕获异常，执行异常分支逻辑  
void methodA() {   
        try {   
            ServiceB.methodB();   
        } catch (SomeException) {   
            // 执行其他业务, 如 ServiceC.methodC();   
        }   
    }  
这种方式也是嵌套事务最有价值的地方, 它起到了分支执行的效果, 如果 ServiceB.methodB 失败, 那么执行 ServiceC.methodC(), 而 ServiceB.methodB 已经回滚到它执行之前的 SavePoint, 所以不会产生脏数据(相当于此方法从未执行过), 这种特性可以用在某些特殊的业务中, 而 PROPAGATION\_REQUIRED 和 PROPAGATION\_REQUIRES\_NEW 都没有办法做到这一点。  
b、 外部事务回滚/提交 代码不做任何修改, 那么如果内部事务(ServiceB#methodB) rollback, 那么首先 ServiceB.methodB 回滚到它执行之前的 SavePoint(在任何情况下都会如此), 外部事务(即 ServiceA#methodA) 将根据具体的配置决定自己是 commit 还是 rollback  
另外三种事务传播属性基本用不到，在此不做分析。

# 出现的问题

我们的项目是通过配置文件的方式配置spring事务控制，这里面就存在切点切入的问题，我们统一从服务层切入，切入方法通过匹配服务的方法名。默认的配置中，服务的方法配置了一些常用的动词开头的方法，比如：save、select。但是项目成员每个人的编码风格都不同，服务方法取名也比较随意，导致事务切点无法切入，从而没有事务控制。

此问题只能通过规范项目成员编写的服务方法名称，统一命名。

另外一个问题就是服务方法之间相互调用问题。目前项目采用的分层设计，从上至下分为controller/service/dao，有的项目成员service层直接调用了其他service的dao层，这就导致调用其他服务的dao层的方法不受预先定义的事务控制。所以需要严格控制服务层之间的调用只能通过service层提供的服务方法调用。

最后一个问题就是同一个服务中两个方法之间的调用。这个问题，网上也有一堆回答，我直接粘贴一篇好文章。

地址：<http://blog.csdn.net/dapinxiaohuo/article/details/52092447>

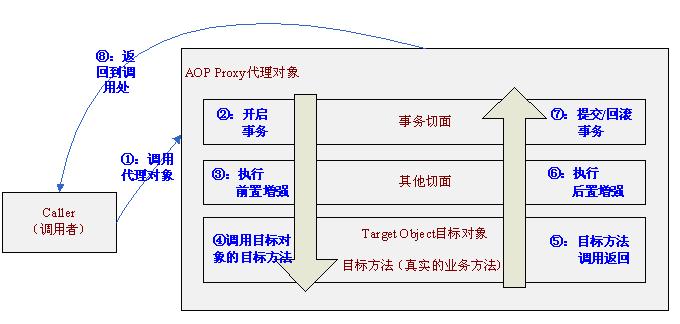
前些日子一朋友在需要在目标对象中进行自我调用，且需要实施相应的事务定义，且网上的一种通过BeanPostProcessor的解决方案是存在问题的。因此专门写此篇帖子分析why。

**1、预备知识**

aop概念请参考【<http://www.iteye.com/topic/1122401>】和【<http://jinnianshilongnian.iteye.com/blog/1418596>】

spring的事务管理，请参考【<http://jinnianshilongnian.iteye.com/blog/1441271>】

使用AOP 代理后的方法调用执行流程，如图所示



也就是说我们首先调用的是AOP代理对象而不是目标对象，首先执行事务切面，事务切面内部通过TransactionInterceptor环绕增强进行事务的增强，即进入目标方法之前开启事务，退出目标方法时提交/回滚事务。

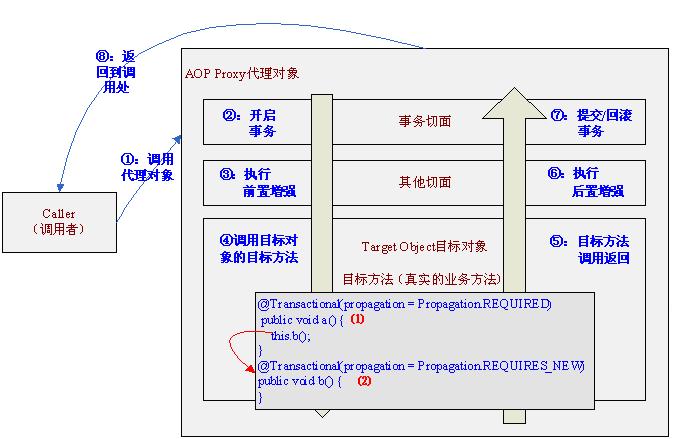
**2、测试代码准备**

Java代码  收藏代码

1. public interface AService {
2. public void a();
3. public void b();
4. }
6. @Service()
7. public class AServiceImpl1 implements AService{
8. @Transactional(propagation = Propagation.REQUIRED)
9. public void a() {
10. this.b();
11. }
12. @Transactional(propagation = Propagation.REQUIRES\_NEW)
13. public void b() {
14. }
15. }

**3、问题**

目标对象内部的自我调用将无法实施切面中的增强，如图所示

  
 此处的this指向目标对象，因此调用this.b()将不会执行b事务切面，即不会执行事务增强，因此b方法的事务定义“@Transactional(propagation = Propagation.REQUIRES\_NEW)”将不会实施，即结果是b和a方法的事务定义是一样的，可以从以下日志看出：

 org.springframework.transaction.annotation.AnnotationTransactionAttributeSource Adding transactional method 'a' with attribute: PROPAGATION\_REQUIRED,ISOLATION\_DEFAULT; ''

org.springframework.beans.factory.support.DefaultListableBeanFactory Returning cached instance of singleton bean 'txManager'

org.springframework.orm.hibernate4.HibernateTransactionManager Creating new transaction with name [com.sishuok.service.impl.AServiceImpl1.a]: PROPAGATION\_REQUIRED,ISOLATION\_DEFAULT; ''  -----创建a方法事务

org.springframework.orm.hibernate4.HibernateTransactionManager Opened new Session …… for Hibernate transaction  ---打开Session

……

org.springframework.transaction.support.TransactionSynchronizationManager Initializing transaction synchronization

org.springframework.transaction.interceptor.TransactionInterceptor Getting transaction for [com.sishuok.service.impl.AServiceImpl1.a]

org.springframework.transaction.interceptor.TransactionInterceptor Completing transaction for [com.sishuok.service.impl.AServiceImpl1.a] ----完成a方法事务

org.springframework.orm.hibernate4.HibernateTransactionManager Triggering beforeCommit synchronization

org.springframework.orm.hibernate4.HibernateTransactionManager Triggering beforeCompletion synchronization

org.springframework.orm.hibernate4.HibernateTransactionManager Initiating transaction commit

org.springframework.orm.hibernate4.HibernateTransactionManager Committing Hibernate transaction on Session ……---提交a方法事务

或

org.springframework.orm.hibernate4.HibernateTransactionManager Rolling back Hibernate transaction on Session ……---如果有异常将回滚a方法事务

 org.springframework.orm.hibernate4.HibernateTransactionManager Triggering afterCommit synchronization

org.springframework.orm.hibernate4.HibernateTransactionManager Triggering afterCompletion synchronization

org.springframework.transaction.support.TransactionSynchronizationManager Clearing transaction synchronization

……

org.springframework.orm.hibernate4.HibernateTransactionManager Closing Hibernate Session …… after transaction     --关闭Session

我们可以看到事务切面只对a方法进行了事务增强，没有对b方法进行增强。

**3、解决方案**

此处a方法中调用b方法时，只要通过AOP代理调用b方法即可走事务切面，即可以进行事务增强，如下所示：

Java代码  收藏代码

1. public void a() {
2. aopProxy.b();//即调用AOP代理对象的b方法即可执行事务切面进行事务增强
3. }

判断一个Bean是否是AOP代理对象可以使用如下三种方法：

AopUtils.isAopProxy(bean)        ： 是否是代理对象；

AopUtils.isCglibProxy(bean)       ： 是否是CGLIB方式的代理对象；

AopUtils.isJdkDynamicProxy(bean) ： 是否是JDK动态代理方式的代理对象；

**3.1、通过ThreadLocal暴露Aop代理对象**

**1、开启暴露Aop代理到ThreadLocal支持（如下配置方式从spring3开始支持）**

Java代码  收藏代码

1. <aop:aspectj-autoproxy expose-proxy="true"/><!—注解风格支持-->

Java代码  收藏代码

1. <aop:config expose-proxy="true"><!—xml风格支持-->

**2、修改我们的业务实现类**

**this.b();-----------修改为--------->**

**((AService) AopContext.currentProxy()).b();**

**3、执行测试用例，日志如下**

 org.springframework.beans.factory.support.DefaultListableBeanFactory Returning cached instance of singleton bean 'txManager'

org.springframework.orm.hibernate4.HibernateTransactionManager Creating new transaction with name [com.sishuok.service.impl.AServiceImpl2.a]: PROPAGATION\_REQUIRED,ISOLATION\_DEFAULT; ''   -----创建a方法事务

org.springframework.orm.hibernate4.HibernateTransactionManager Opened new Session ……for Hibernate transaction  --打开a Session

org.springframework.orm.hibernate4.HibernateTransactionManager Preparing JDBC Connection of Hibernate Session ……

org.springframework.orm.hibernate4.HibernateTransactionManager Exposing Hibernate transaction as JDBC transaction ……

……

org.springframework.transaction.support.TransactionSynchronizationManager Initializing transaction synchronization

org.springframework.transaction.interceptor.TransactionInterceptor Getting transaction for [com.sishuok.service.impl.AServiceImpl2.a]

org.springframework.transaction.annotation.AnnotationTransactionAttributeSource Adding transactional method 'b' with attribute: PROPAGATION\_REQUIRES\_NEW,ISOLATION\_DEFAULT; ''

……

org.springframework.orm.hibernate4.HibernateTransactionManager Suspending current transaction, creating new transaction with name [com.sishuok.service.impl.AServiceImpl2.b]  -----创建b方法事务（并暂停a方法事务）

……

org.springframework.orm.hibernate4.HibernateTransactionManager Opened new Session  for Hibernate transaction  ---打开b Session

……

org.springframework.transaction.support.TransactionSynchronizationManager Initializing transaction synchronization

org.springframework.transaction.interceptor.TransactionInterceptor Getting transaction for [com.sishuok.service.impl.AServiceImpl2.b]

org.springframework.transaction.interceptor.TransactionInterceptor Completing transaction for [com.sishuok.service.impl.AServiceImpl2.b] ----完成b方法事务

 org.springframework.orm.hibernate4.HibernateTransactionManager Triggering beforeCommit synchronization

org.springframework.orm.hibernate4.HibernateTransactionManager Triggering beforeCompletion synchronization

org.springframework.orm.hibernate4.HibernateTransactionManager Initiating transaction commit

org.springframework.orm.hibernate4.HibernateTransactionManager Committing Hibernate transaction on Session …… ---提交b方法事务

org.springframework.orm.hibernate4.HibernateTransactionManager Triggering afterCommit synchronization

org.springframework.orm.hibernate4.HibernateTransactionManager Triggering afterCompletion synchronization

org.springframework.transaction.support.TransactionSynchronizationManager Clearing transaction synchronization

……

org.springframework.orm.hibernate4.HibernateTransactionManager Closing Hibernate Session …… after transaction  --关闭 b Session

-----到此b方法事务完毕

org.springframework.orm.hibernate4.HibernateTransactionManager Resuming suspended transaction after completion of inner transaction ---恢复a方法事务

……

org.springframework.transaction.support.TransactionSynchronizationManager Initializing transaction synchronization

org.springframework.transaction.interceptor.TransactionInterceptor Completing transaction for [com.sishuok.service.impl.AServiceImpl2.a] ----完成a方法事务

org.springframework.orm.hibernate4.HibernateTransactionManager Triggering beforeCommit synchronization

org.springframework.orm.hibernate4.HibernateTransactionManager Triggering beforeCompletion synchronization

org.springframework.orm.hibernate4.HibernateTransactionManager Initiating transaction commit

org.springframework.orm.hibernate4.HibernateTransactionManager Committing Hibernate transaction on Session ……---提交a方法事务

org.springframework.orm.hibernate4.HibernateTransactionManager Triggering afterCommit synchronization

org.springframework.orm.hibernate4.HibernateTransactionManager Triggering afterCompletion synchronization

org.springframework.transaction.support.TransactionSynchronizationManager Clearing transaction synchronization

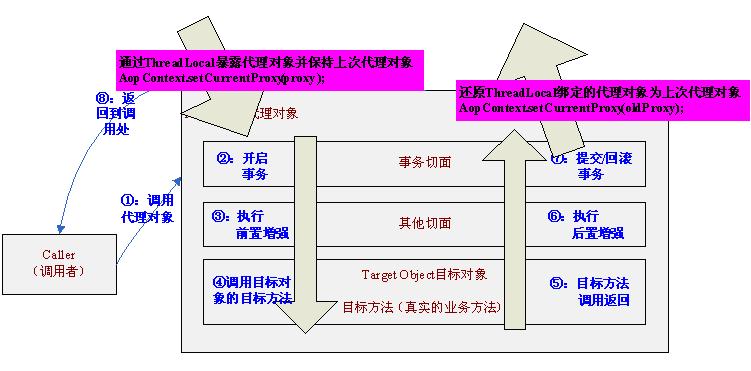
……

org.springframework.orm.hibernate4.HibernateTransactionManager Closing Hibernate Session …… after transaction  --关闭 a Session

**此处我们可以看到b方法的事务起作用了。**

以上方式是解决目标对象内部方法自我调用并实施事务的最简单的解决方案。

**4、实现原理分析**



4.1、在进入代理对象之后通过AopContext.serCurrentProxy(proxy)暴露当前代理对象到ThreadLocal，并保存上次ThreadLocal绑定的代理对象为oldProxy；

4.2、接下来我们可以通过 AopContext.currentProxy() 获取当前代理对象；

4.3、在退出代理对象之前要重新将ThreadLocal绑定的代理对象设置为上一次的代理对象，即AopContext.serCurrentProxy(oldProxy)。

有些人不喜欢这种方式，说通过ThreadLocal暴露有性能问题，其实这个不需要考虑，因为事务相关的（Session和Connection）内部也是通过SessionHolder和ConnectionHolder暴露到ThreadLocal实现的。

不过自我调用这种场景确实只有很少情况遇到，因此不用这种方式我们也可以通过如下方式实现。

3.2、通过初始化方法在目标对象中注入代理对象

Java代码  收藏代码

1. @Service
2. public class AServiceImpl3 implements AService{
3. @Autowired  //①  注入上下文
4. private ApplicationContext context;
6. private AService proxySelf; //②  表示代理对象，不是目标对象
7. @PostConstruct  //③ 初始化方法
8. private void setSelf() {
9. //从上下文获取代理对象（如果通过proxtSelf=this是不对的，this是目标对象）
10. //此种方法不适合于prototype Bean，因为每次getBean返回一个新的Bean
11. proxySelf = context.getBean(AService.class);
12. }
13. @Transactional(propagation = Propagation.REQUIRED)
14. public void a() {
15. proxySelf.b(); //④ 调用代理对象的方法 这样可以执行事务切面
16. }
17. @Transactional(propagation = Propagation.REQUIRES\_NEW)
18. public void b() {
19. }
20. }

此处日志就不分析，和3.1类似。此种方式不是很灵活，所有需要自我调用的实现类必须重复实现代码。

3.3、通过**BeanPostProcessor**在目标对象中注入代理对象

此种解决方案可以参考<http://fyting.iteye.com/blog/109236>。

BeanPostProcessor 的介绍和使用敬请等待我的下一篇分析帖。

**一、定义BeanPostProcessor 需要使用的标识接口**

Java代码  收藏代码

1. public interface BeanSelfAware {
2. void setSelf(Object proxyBean);
3. }

 即我们自定义的BeanPostProcessor （InjectBeanSelfProcessor）如果发现我们的Bean是实现了该标识接口就调用setSelf注入代理对象。

**二、Bean实现**

Java代码  收藏代码

1. @Service
2. public class AServiceImpl4 implements AService, BeanSelfAware {//此处省略接口定义
3. private AService proxySelf;
4. public void setSelf(Object proxyBean) { //通过InjectBeanSelfProcessor注入自己（目标对象）的AOP代理对象
5. this.proxySelf = (AService) proxyBean;
6. }
7. @Transactional(propagation = Propagation.REQUIRED)
8. public void a() {
9. proxySelf.b();//调用代理对象的方法 这样可以执行事务切面
10. }
11. @Transactional(propagation = Propagation.REQUIRES\_NEW)
12. public void b() {
13. }
14. }

实现BeanSelfAware标识接口的setSelf将代理对象注入，并且通过“proxySelf.b()”这样可以实施b方法的事务定义。

**三、InjectBeanSelfProcessor实现**

Java代码  收藏代码

1. @Component
2. public class InjectBeanSelfProcessor implements BeanPostProcessor {
3. public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
4. return bean;
5. }
6. public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
7. if(bean instanceof BeanSelfAware) {//如果Bean实现了BeanSelfAware标识接口，就将代理对象注入
8. ((BeanSelfAware) bean).setSelf(bean); //即使是prototype Bean也可以使用此种方式
9. }
10. return bean;
11. }
12. }

postProcessAfterInitialization根据目标对象是否实现BeanSelfAware标识接口，通过setSelf(bean)将代理对象（bean）注入到目标对象中，从而可以完成目标对象内部的自我调用。

关于BeanPostProcessor的执行流程等请一定参考我的这篇帖子，否则无法继续往下执行。

**四、InjectBeanSelfProcessor的问题**

**（1、场景：**通过InjectBeanSelfProcessor进行注入代理对象且循环依赖场景下会产生前者无法通过setSelf设置代理对象的问题。 循环依赖是应该避免的，但是实际工作中不可避免会有人使用这种注入，毕竟没有强制性。

**（2、用例**

**（2.1、定义BeanPostProcessor 需要使用的标识接口**

和3.1中一样此处不再重复。

**（2.2、Bean实现**

Java代码  收藏代码

1. @Service
2. public class AServiceImpl implements AService, BeanSelfAware {//此处省略Aservice接口定义
3. @Autowired
4. private BService bService;   //①  通过@Autowired方式注入BService
5. private AService self;       //②  注入自己的AOP代理对象
6. public void setSelf(Object proxyBean) {
7. this.self = (AService) proxyBean;  //③ 通过InjectBeanSelfProcessor注入自己（目标对象）的AOP代理对象
8. System.out.println("AService=="+ AopUtils.isAopProxy(this.self)); //如果输出true标识AOP代理对象注入成功
9. }
10. @Transactional(propagation = Propagation.REQUIRED)
11. public void a() {
12. self.b();
13. }
14. @Transactional(propagation = Propagation.REQUIRES\_NEW)
15. public void b() {
16. }
17. }

Java代码  收藏代码

1. @Service
2. public class BServiceImpl implements BService, BeanSelfAware {//此处省略Aservice接口定义
3. @Autowired
4. private AService aService;  //①  通过@Autowired方式注入AService
5. private BService self;      //②  注入自己的AOP代理对象
6. public void setSelf(Object proxyBean) {  //③ 通过InjectBeanSelfProcessor注入自己（目标对象）的AOP代理对象
7. this.self = (BService) proxyBean;
8. System.out.println("BService=" + AopUtils.isAopProxy(this.self)); //如果输出true标识AOP代理对象注入成功
9. }
10. @Transactional(propagation = Propagation.REQUIRED)
11. public void a() {
12. self.b();
13. }
14. @Transactional(propagation = Propagation.REQUIRES\_NEW)
15. public void b() {
16. }
17. }

此处A依赖B，B依赖A，即构成循环依赖，此处不探讨循环依赖的设计问题（实际工作应该避免循环依赖），只探讨为什么循环依赖会出现注入代理对象失败的问题。

循环依赖请参考我的博文【<http://jinnianshilongnian.iteye.com/blog/1415278>】。

依赖的初始化和销毁顺序请参考我的博文【<http://jinnianshilongnian.iteye.com/blog/1415461>】。

**（2.3、InjectBeanSelfProcessor实现**

和之前3.3中一样 此处不再重复。

**（2.4、测试用例**

Java代码  收藏代码

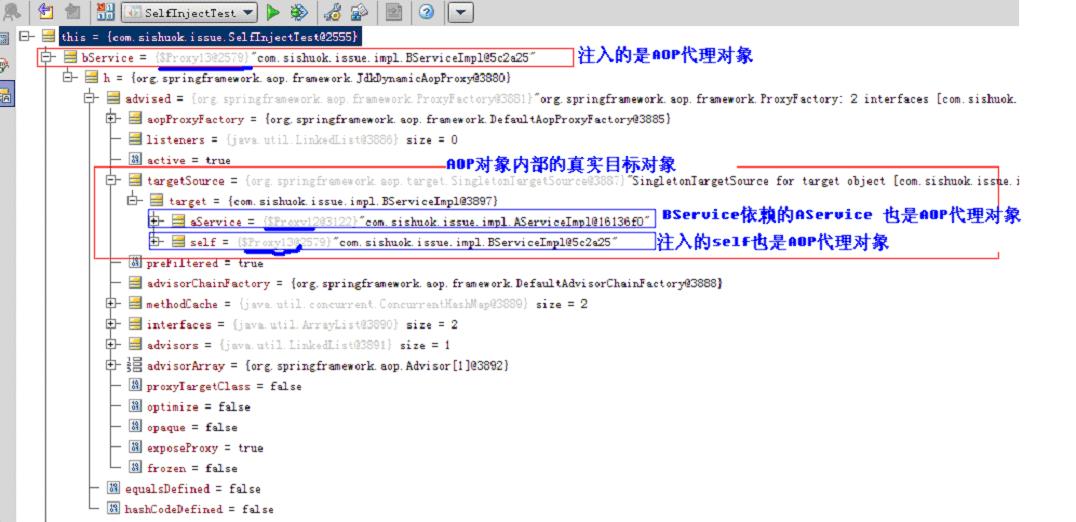
1. @RunWith(value = SpringJUnit4ClassRunner.class)
2. @ContextConfiguration(value = {"classpath:spring-config.xml"})
3. public class SelfInjectTest {
4. @Autowired
5. AService aService;
6. @Autowired
7. BService bService;
8. @Test
9. public void test() {
10. }
11. }

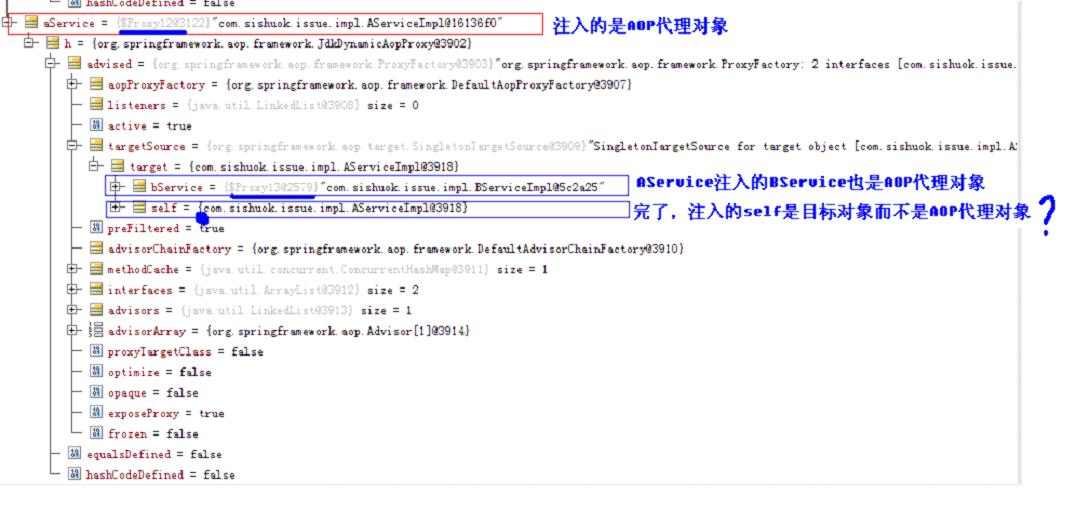
执行如上测试用例会输出：

BService=true

AService==false

即BService通过InjectBeanSelfProcessor注入代理对象成功，而AService却失败了（实际是注入了目标对象），如下是debug得到的信息：





**（2. 5、这是为什么呢，怎么在循环依赖会出现这种情况？**

敬请期待我的下一篇分析帖。

3.4、改进版的InjectBeanSelfProcessor的解决方案

Java代码  收藏代码

1. @Component
2. public class InjectBeanSelfProcessor2 implements BeanPostProcessor, ApplicationContextAware {
3. private ApplicationContext context;
4. //① 注入ApplicationContext
5. public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
6. this.context = applicationContext;
7. }
8. public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
9. if(!(bean instanceof BeanSelfAware)) { //② 如果Bean没有实现BeanSelfAware标识接口 跳过
10. return bean;
11. }
12. if(AopUtils.isAopProxy(bean)) { //③ 如果当前对象是AOP代理对象，直接注入
13. ((BeanSelfAware) bean).setSelf(bean);
14. } else {
15. //④ 如果当前对象不是AOP代理，则通过context.getBean(beanName)获取代理对象并注入
16. //此种方式不适合解决prototype Bean的代理对象注入
17. ((BeanSelfAware)bean).setSelf(context.getBean(beanName));
18. }
19. return bean;
20. }
21. public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
22. return bean;
23. }
24. }

5、总结

**纵观其上：**

【3.1 通过ThreadLocal暴露Aop代理对象】适合解决所有场景（不管是singleton Bean还是prototype Bean）的AOP代理获取问题（即能解决目标对象的自我调用问题）；

【3.2 通过初始化方法在目标对象中注入代理对象】 和【3.4 改进版的InjectBeanSelfProcessor的解决方案】能解决普通（无循环依赖）的AOP代理对象注入问题，而且也能解决【3.3】中提到的循环依赖（应该是singleton之间的循环依赖）造成的目标对象无法注入AOP代理对象问题，但该解决方案不适合解决循环依赖中包含prototype Bean的自我调用问题；

【3.3 通过BeanPostProcessor 在目标对象中注入代理对象】：只能解决 普通（无循环依赖）的 的Bean注入AOP代理，无法解决循环依赖的AOP代理对象注入问题，即无法解决目标对象的自我调用问题。

jingnianshilongnian 写道

spring允许的循环依赖（只考虑单例和原型Bean）：  
A----B  
B----A  
只有在A和B都不为原型是允许的，即如果A和B都是prototype则会报错（无法进行原型Bean的循环依赖）。  
A（单例）---B（单例） 或 A（原型）---B（单例） 这是可以的，但 A（原型）---B（原型）或 A（原型）---B（单例Lazy）【且context.getBean("A")】时 这是不允许的。  
  
一、A（原型）---B（原型） A（原型）---B（单例Lazy）【且context.getBean("A")】 会报：  
Error creating bean with name 'BServiceImpl': Injection of autowired dependencies failed; nested exception is org.springframework.beans.factory.BeanCreationException: Could not autowire field: private com.sishuok.issue.AService com.sishuok.issue.impl.BServiceImpl.aService; nested exception is org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'AServiceImpl': Injection of autowired dependencies failed; nested exception is org.springframework.beans.factory.BeanCreationException: Could not autowire field: private com.sishuok.issue.BService com.sishuok.issue.impl.AServiceImpl.bService; nested exception is org.springframework.beans.factory.BeanCurrentlyInCreationException: Error creating bean with name 'BServiceImpl': Requested bean is currently in creation: [color=red]Is there an unresolvable circular reference[/color]?  
  
二、A（原型）---B（单例） 和 A（单例）---B（单例）  
这种方式 使用我的 【3.3 通过BeanPostProcessor 在目标对象中注入代理对象】 是没有问题的。  
  
因此【 3.4 改进版的InjectBeanSelfProcessor的解决方案 】 可以作为最后的解决方案。