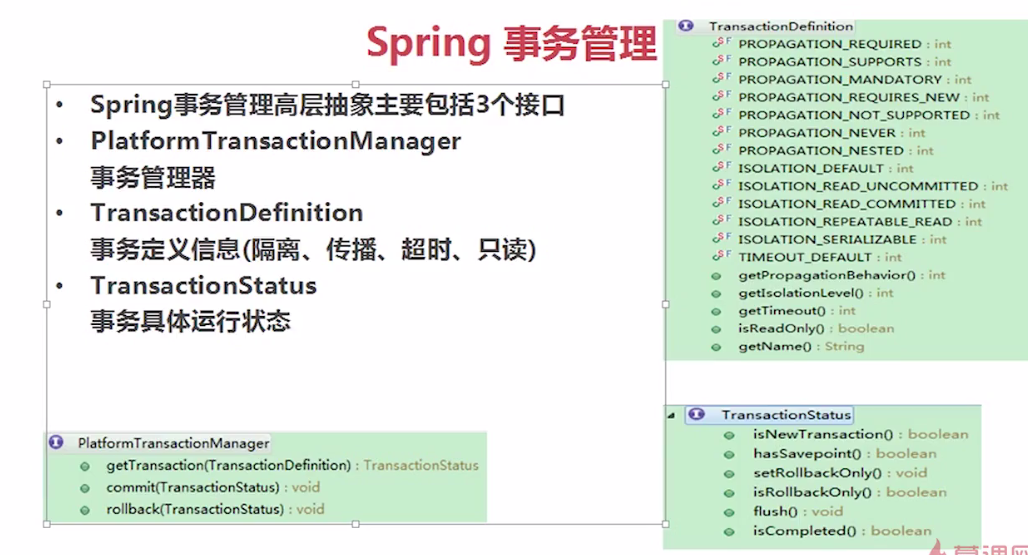
# Spring事务简介

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。

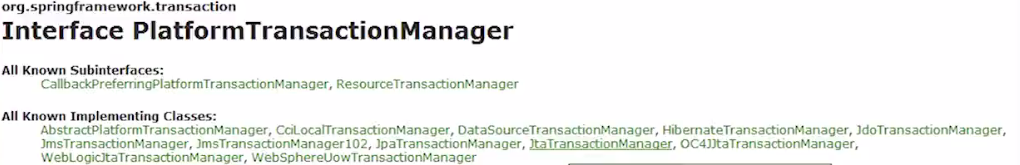
四大特性：原子性 一致性 隔离性 持久性

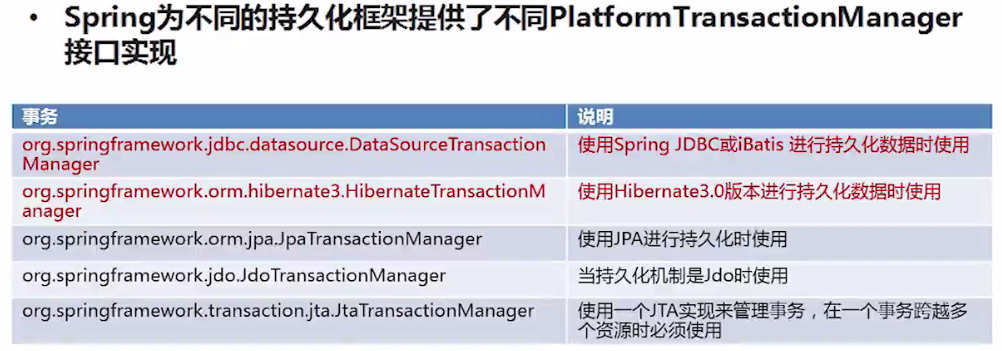


事务运行状态：如事务是否提交 是否有保存点 是不是新的事务

事务管理器：对事务进行回滚，如提交 回滚

## PlatformTransactionManager





## TransactionDefinition

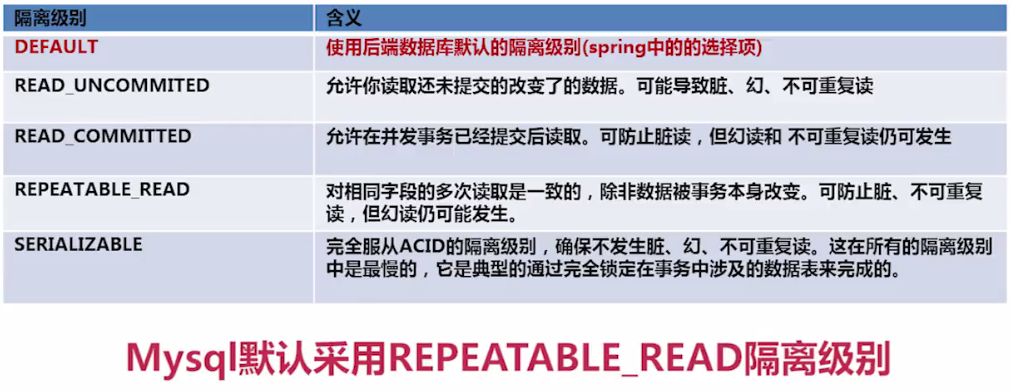
### 隔离级别

如果不考虑隔离性，会引发安全问题：脏读 不可重复读 幻读

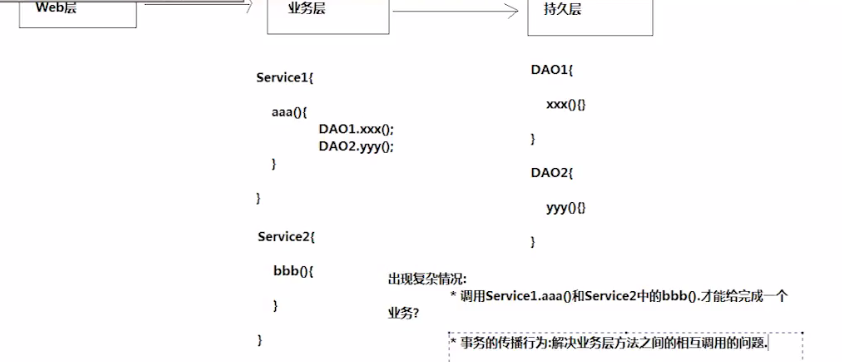
脏读：一个事务读取到了另外一个事务没有提交的数据

不可重复读：在同一个事务中，多次读取同一数据返回的结果有所不同

幻读：一个事务读取了另外一个事务插入的一些记录



### 传播行为





1 如果aaa中有事务，则在调用bbb时直接加入到aaa事务中，如果aaa中没有事务，则创建新的事务，包含aaa和bbb操作

2 如果aaa中有事务，则使用，如果aaa中没有事务，则bbb中不使用事务

3如果aaa中有事务，则使用，如果aaa中没有事务，则抛出异常

4 如果aaa中有事务，则执行完aaa之后，将aaa事务挂起，再创建一个新的事务用来执行bbb，也就是说aaa和bbb的方法不在一个事务中

5如果aaa中有事务，则挂起，然后执行bbb

6如果aaa中有事务，则抛出异常

7 执行完aaa时，使用事务创建一个保存点，然后执行bbb，如果bbb没有异常，则一起提交，如果bbb中异常，可以回滚到保存点或者最开始的地方。

（如果没有spring，直接使用hibernate时，当一个service层调用其他或者本业务层的方法时，为了保证事务，必须要把当前的hibernate session传递到下一个方法中，或者采用ThreadLocal的方法，将session传递给下一个方法）

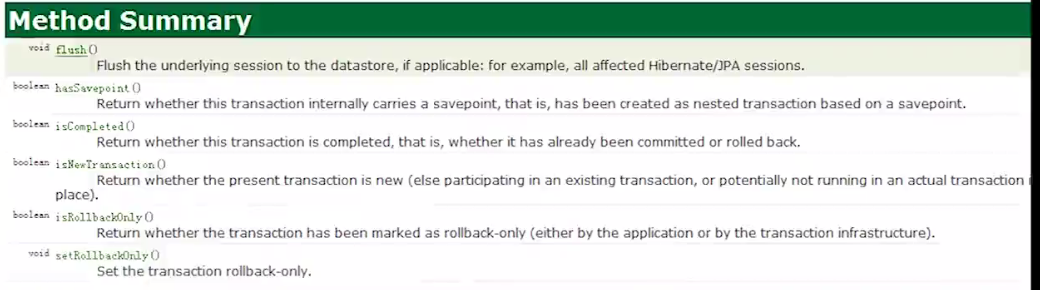
### 事务超时时间和只读

设置事务的超时时间，单位为秒，默认为-1表示使用底层事务的超时时间

使用setTimeout(100)来设置超时时间，如果超时，则抛出异常org.springframework.transaction.TransactionTimedOutException，并将当前事务标记为应该回滚，即超时后的事务会被自动回滚。

事务只读：只能读取，不能做任何修改

## TransactionStatus



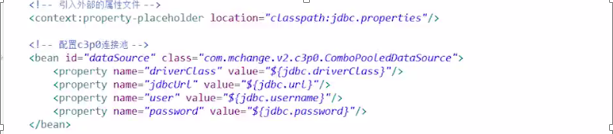
## spring事务管理

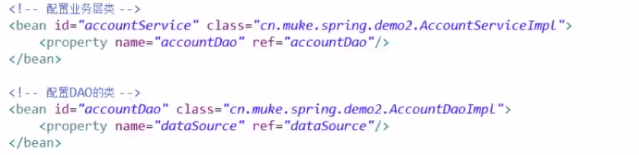


声明式事务是通过aop实现，其实就是对一段代码的增强操作，就是在代码开始执行开启事务，在执行结束提交或者回滚事务。

## 转账案例

### applicationContext.xml





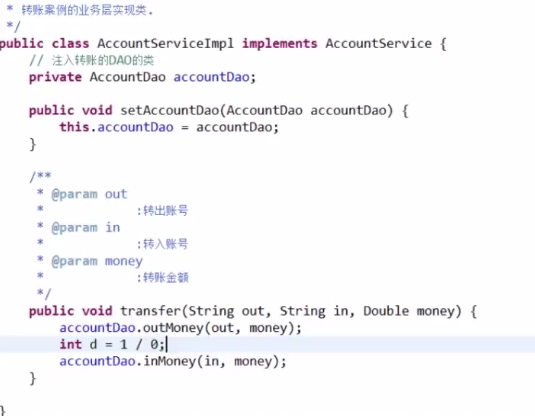


### DAO

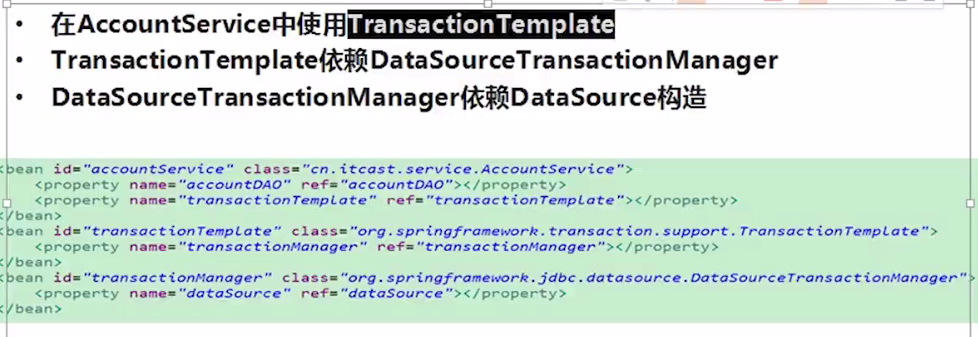


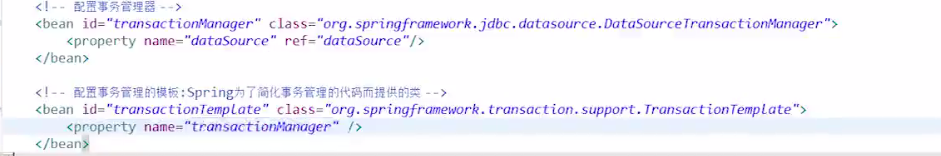
### Service

#### 没有事务管理

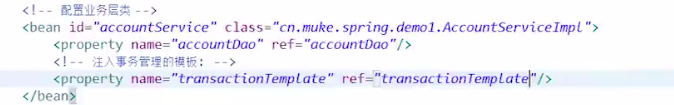


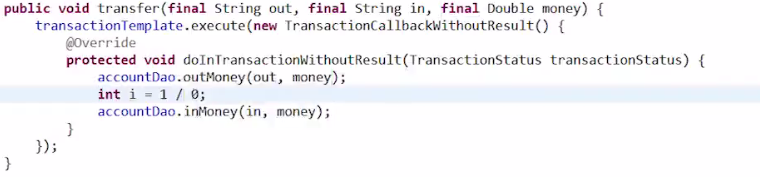
#### 编程式事务管理





事务模板只是为了简化事务管理的代码，真正的管理还是由transactionmanager进行管理。



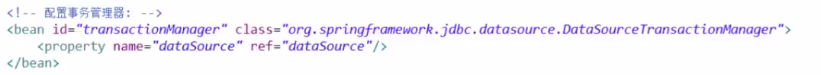


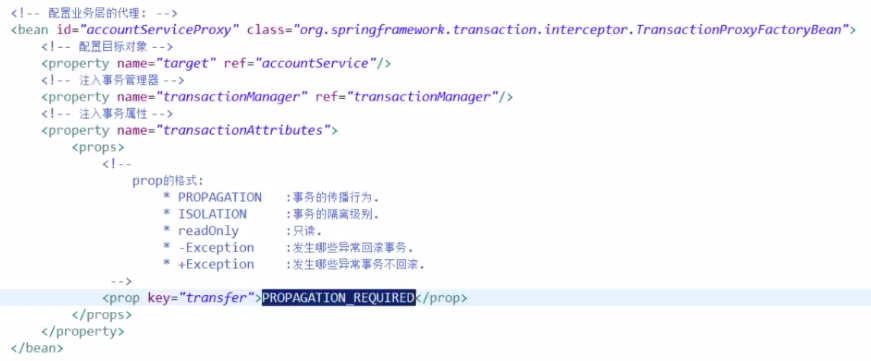
需要手动改写service层的代码

#### 声明式事务管理1

**基于TransactionProxyFactoryBean**

需要引入aop相关的jar包





缺点：需要为每一个类创建一个TransactionProxyBean。

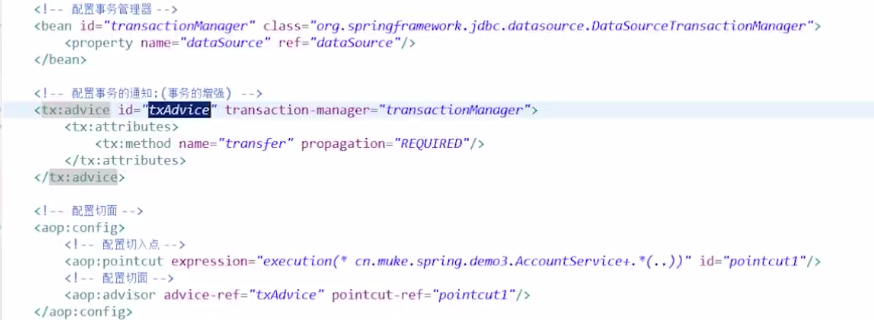


**这里注入的时候需要注入代理对象**

#### 声明式事务管理2

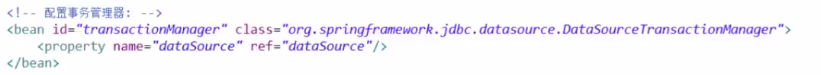
**基于Aspectj的配置**

引入aspectj的相关jar包。

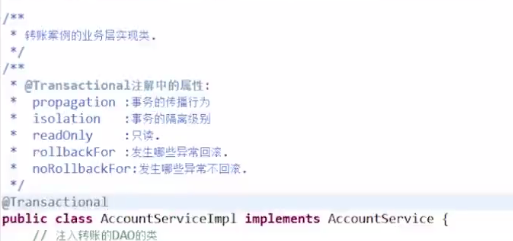


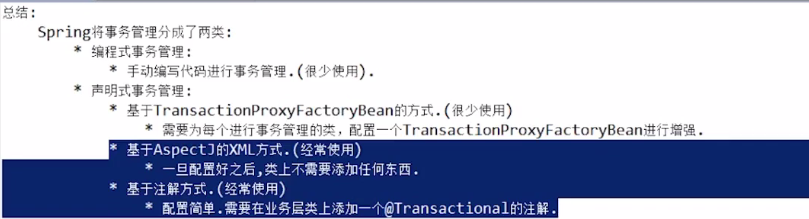
在测试类中直接注入这个类即可，不需要注入代理类。因为这种方式的配置，是基于BeanPostProcessor这个类，在这个类的生成过程中，这个类就是一个代理对象。

#### 使用注解配置声明式事务









# Spring事务中出现的问题

http://fyting.iteye.com/blog/109236

**Spring事务在同一个类中自身方法互相调用时，无法拦截，aop不生效**

1. **public** **class** SomeServiceImpl **implements** SomeService
2. {
4. **public** **void** someMethod()
5. {
6. someInnerMethod();
7. //foo...
8. }
10. **public** **void** someInnerMethod()
11. {
12. //bar...
13. }
14. }

两个方法都经过aop代理，执行时都实现系统日志记录。单独使用someInnerMethod时，没有问题，但是someMethod就有问题了。someMethod里调用的someInnerMethod方法是原始的，未经过AOP增强的。我们期望调用一次someMethod会记录下两条系统日志，分别是someInnerMethod和someMethod的，但实际上只能记录下someMethod的日志，也就是只有一条。在配置事务时也可能会出现问题，比如someMethod方法是REQUIRED，someInnerMethod方法是REQUIRES\_NEW，someInnerMethod的配置将不起作用，与someMethod方法会使用同一个事务，不会按照所配置的打开新事务。

**（因为这里通过someMethod调用someInnerMethod时是普通的java函数调用，并不会到proxy上，所以aop不起作用）**解决：可以从spring容器中取得该对象，或者把该对象作为参数传到方法里，或者在SomeServiceImpl类里加一个属性来引用自己的代理对象，如名为self，在使用前先设置一下，someService.setSelf(someService);把代理对象的引用保存到对象里面，在嵌套调用时调用self的方法而不是this。  
由于java这个静态类型语言限制，最后想到个曲线救国的办法，出现这种特殊情况时，不要直接调用自身方法，而通过AOP代理后的对象。**在实现里保留一个AOP代理对象的引用，调用时通过这个代理即可**。比如：

1. //从beanFactory取得AOP代理后的对象
2. SomeService someServiceProxy = (SomeService)beanFactory.getBean("someService");
4. //把AOP代理后的对象设置进去
5. someServiceProxy.setSelf(someServiceProxy);
7. //在someMethod里面调用self的someInnerMethod，这样就正确了
8. someServiceProxy.someMethod();

但这个代理对象还要我们手动set进来，幸好SpringBeanFactory有BeanPostProcessor扩展，在bean初始化前后会统一传递给BeanPostProcess处理，繁琐的事情就可以交给程序了，代码如下，首先定义一个BeanSelfAware接口，实现了此接口的程序表明需要注入代理后的对象到自身。

1. **public** **class** SomeServiceImpl **implements** SomeService,BeanSelfAware
3. {
5. **private** SomeService self;//AOP增强后的代理对象


9. //实现BeanSelfAware接口
11. **public** **void** setSelf(Object proxyBean)
13. {
15. **this**.self = (SomeService)proxyBean
17. }


21. **public** **void** someMethod()
23. {
25. someInnerMethod();//注意这句，通过self这个对象，而不是直接调用的
27. //foo...
29. }
31. **public** **void** someInnerMethod()
33. {
35. //bar...
37. }
39. }

再定义一个BeanPostProcessor，beanFactory中的每个Bean初始化完毕后，调用所有BeanSelfAware的setSelf方法，把自身的代理对象注入自身

1. **public** **class** InjectBeanSelfProcessor **implements** BeanPostProcessor
2. {
4. **public** Object postProcessAfterInitialization(Object bean, String beanName) **throws** BeansException
5. {
6. **if**(bean **instanceof** BeanSelfAware)
7. {
8. System.out.println("inject proxy：" + bean.getClass());
9. BeanSelfAware myBean = (BeanSelfAware)bean;
10. myBean.setSelf(bean);
11. **return** myBean;
12. }
13. **return** bean;
14. }
16. **public** Object postProcessBeforeInitialization(Object bean, String beanName) **throws** BeansException
17. {
18. **return** bean;
19. }
20. }

最后，在BeanFactory配置中组合起来，只需要把BeanPostProcesser加进去就可以了，比平常多一行配置而已。

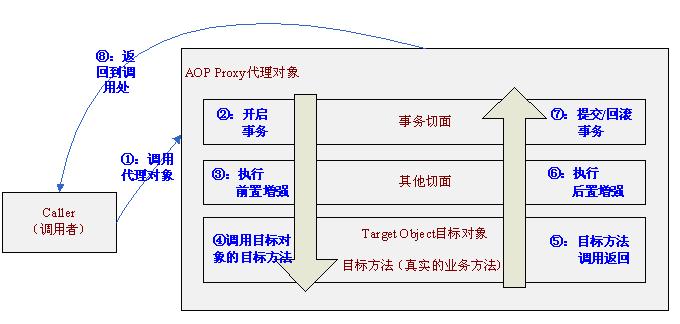
1. <!-- 注入代理后的bean到bean自身的BeanPostProcessor... -->
2. <bean **class**=" org.mypackage.InjectBeanSelfProcessor"></bean>
4. <bean id="someServiceTarget" **class**="org.mypackage.SomeServiceImpl" />
6. <bean id="someService" **class**="org.springframework.aop.framework.ProxyFactoryBean">
7. <property name="target">
8. <ref local="someServiceTarget" />
9. </property>
10. <property name="interceptorNames">
11. <list>
12. <value>someAdvisor</value>
13. </list>
14. </property>
15. </bean>
17. <!-- 调用spring的DebugInterceptor记录日志,以确定方法是否被AOP增强 -->
18. <bean id="debugInterceptor" **class**="org.springframework.aop.interceptor.DebugInterceptor" />
20. <bean id="someAdvisor" **class**="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
21. <property name="advice">
22. <ref local="debugInterceptor" />
23. </property>
24. <property name="patterns">
25. <list>
26. <value>.\*someMethod</value>
27. <value>.\*someInnerMethod</value>
28. </list>
29. </property>
30. </bean>

这里的someService#someInnerMethod就表现出预期的行为了，无论怎样，它都是经过AOP代理的，执行时都会输出日志信息。   
用XmlBeanFactory进行测试需要注意，所有的BeanPostProcessor并不会自动生效，需要执行以下代码：

1. XmlBeanFactory factory = **new** XmlBeanFactory(...);
2. InjectBeanSelfProcessor postProcessor = **new** InjectBeanSelfProcessor();
3. factory.addBeanPostProcessor(postProcessor);

## 预备知识

使用aop代理后，方法调用执行过程：



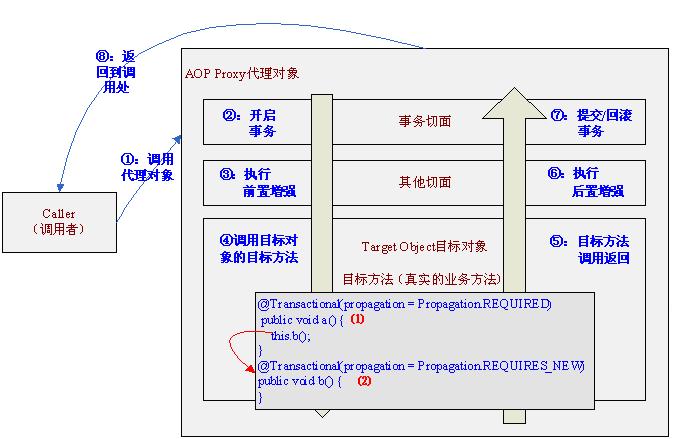
我们首先调用的是AOP代理对象而不是目标对象，首先执行事务切面，事务切面内部通过TransactionInterceptor环绕增强进行事务的增强，即进入目标方法之前开启事务，退出目标方法时提交/回滚事务。

## 测试代码

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. }

## 问题

目标对象内部的自我调用将无法实施切面中的增强



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。

## 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.

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；

### 方案1

#### 通过暴露ThreadLocal暴露AOP代理对象

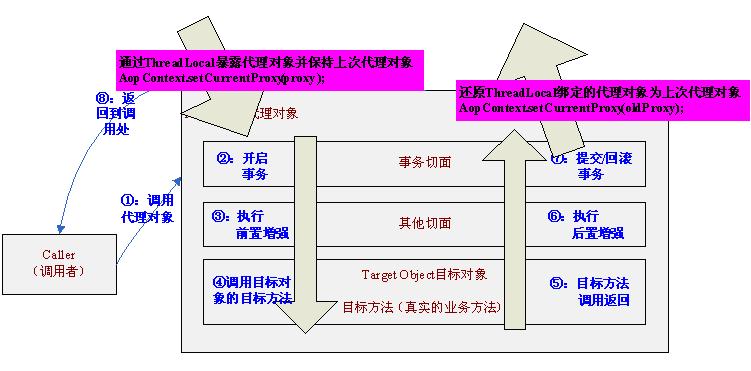
开启暴露AOP代理到ThreadLocal支持

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

修改业务类

this.b();-----------修改为--------->((AService) AopContext.currentProxy()).b();

#### 实现原理分析



1在进入代理对象之后通过AopContext.setCurrentProxy(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；

2接下来我们可以通过 AopContext.currentProxy() 获取当前代理对象；

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.setCurrentProxy(oldProxy)。

### 方案2

#### 通过初始化方法在目标对象中注入代理对象

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. }

### 方案3

#### 通过BeanPostProcessor

http://www.iteye.com/topic/1122859

##### 定义BeanPostprocessor需要使用的接口

<http://fyting.iteye.com/blog/109236>

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。

##### Bean实现

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。

##### InjectBeanSelfProcessor实现

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。

### 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**

和上述一样

**（2.2、Bean实现**

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. }
18. @Service
19. **public** **class** BServiceImpl **implements** BService, BeanSelfAware {//此处省略Aservice接口定义
20. @Autowired
21. **private** AService aService;  //①  通过@Autowired方式注入AService
22. **private** BService self;      //②  注入自己的AOP代理对象
23. **public** **void** setSelf(Object proxyBean) {  //③ 通过InjectBeanSelfProcessor注入自己（目标对象）的AOP代理对象
24. **this**.self = (BService) proxyBean;
25. System.out.println("BService=" + AopUtils.isAopProxy(**this**.self)); //如果输出true标识AOP代理对象注入成功
26. }
27. @Transactional(propagation = Propagation.REQUIRED)
28. **public** **void** a() {
29. self.b();
30. }
31. @Transactional(propagation = Propagation.REQUIRES\_NEW)
32. **public** **void** b() {
33. }
34. }

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**

和上述一样

**（2.4、测试用例**

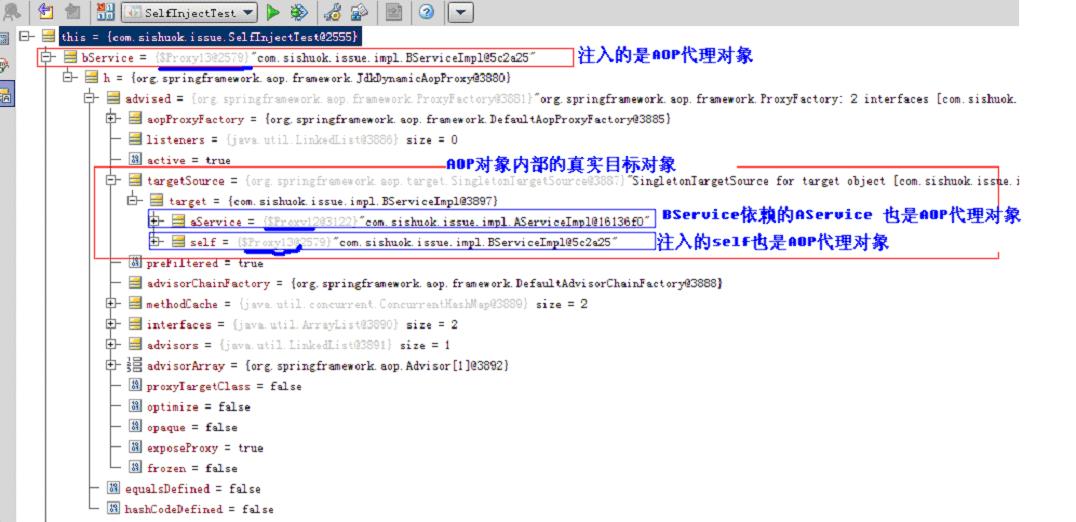
1. @RunWith(value = SpringJUnit4ClassRunner.**class**)
2. @ContextConfiguration(value = {"classpath:spring-config.xml"})
3. **public** **class** SelfInjectTest {
4. @Autowired
5. AService aService;
6. @Autowired
7. BService bService;
8. @Test
9. **public** **void** test() {
10. }
11. }

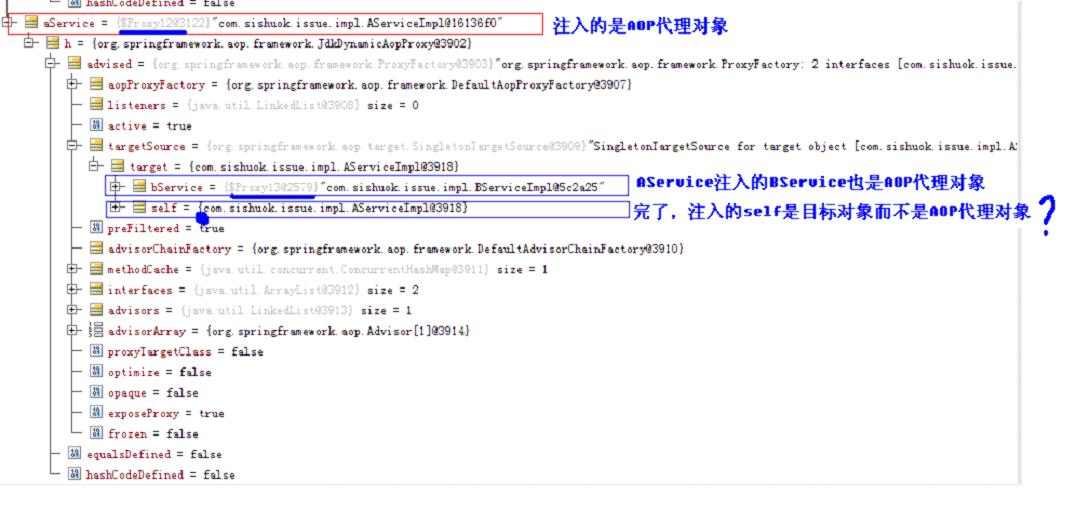
执行如上测试用例会输出：

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





### 改进版的InjectBeanSelfProcessor的解决方案

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. }

## 总结

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。