

开涛的博客

博客

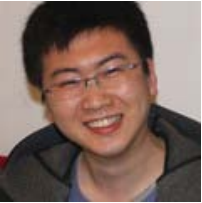
微博

相册

收藏

留言

关于我



jinnianshilongnian

浏览: 2548974 次

性别:

我现在离线

最近访客 [更多访客>>](#)



zhouyvi



xiaochong4



fangwi



beanlai

博客专栏



[跟我学spring3](#)
浏览量: 639384



[Spring杂谈](#)
浏览量: 473367



[跟开涛学SpringMVC...](#)
浏览量: 819966



[Servlet3.1规范翻...](#)
浏览量: 76193



[springmvc杂谈](#)
浏览量: 160266



[hibernate 杂谈](#)
浏览量: 70433



[跟我学Shiro](#)
浏览量: 198344

文章分类

基于JDK动态代理和CGLIB动态代理的实现Spring注解管理事务 (@Trasactional) 到底有什么区别。

博客分类: [spring杂谈](#)

基于JDK动态代理和CGLIB动态代理的实现Spring注解管理事务 (@Trasactional) 到底有什么区别。

我还是喜欢基于Schema风格的Spring事务管理，但也有很多人在用基于@Trasactional注解的事务管理，但在通过基于JDK动态代理和CGLIB动态代理的实现Spring注解管理事务是有区别的，我们接下来看看到底有哪些区别。

一、基础工作

首先修改我们上一次做的 [SpringMVC + spring3.1.1 + hibernate4.1.0 集成及常见问题总结](#)，如下所示：

将xml声明式事务删除

java代码：

Java 代码

```
1. <aop:config expose-proxy="true">
2.     <!-- 只对业务逻辑层实施事务 -->
3.     <aop:pointcut id="txPointcut" expression="execution(* cn.javass..service..*(..))"
4.         />
5.     <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut"/>
</aop:config>
```

并添加注解式事务支持：

java代码：

Java 代码

```
1. <tx:annotation-driven transaction-manager="txManager"/>
```

在我们的BaseService接口上添加 @Transactional 使该方法开启事务

- 全部博客 (285)
 - 跟我学spring (54)
 - 跟开涛学SpringMVC (34)
 - spring4 (11)
 - spring杂谈 (50)
 - springmvc杂谈 (22)
 - 跟我学Shiro (26)
 - shiro 杂谈 (3)
 - hibernate 杂谈 (10)
 - java 开发常见问题分析 (36)
 - 加速Java应用开发 (5)
 - Servlet 3.1规范[翻译] (21)
 - servlet3.x (2)
 - websocket协议[翻译] (14)
 - websocket规范[翻译] (1)
 - java web (5)
 - db (1)
 - js & jquery & bootstrap (4)
 - 非技术 (4)
 - reminder[转载] (23)
 - 跟叶子学把妹 (5)
- 社区版块
- 我的资讯 (9)
 - 我的论坛 (1112)
 - 我的问答 (2428)
- 存档分类
- 2014-08 (5)
 - 2014-07 (5)
 - 2014-04 (8)
 - 更多存档...
- 评论排行榜
- 跟我学Shiro目录贴
 - 跟叶子学把妹——教程序猿把妹第一集
 - Spring4新特性——泛型限定式依赖注入
 - 第二章 身份验证——《跟我学Shiro》
 - 第五章 编码/加密——《跟我学Shiro》
- 最新评论

java代码:

Java 代码



```
1. package cn.javass.common.service;
2. public interface IBaseService<M extends java.io.Serializable, PK extends java.io.Serializable> {
3.     @Transactional //开启默认事务
4.     public int countAll();
5. }
```

在我们的log4j.properties中添加如下配置，表示输出spring的所有debug信息

java代码:

Java 代码



```
1. log4j.logger.org.springframework=INFO,CONSOLE
```

在我们的resources.properties里将hibernate.show_sql=true 改为true，为了看到hibernate的sql。

单元测试类:

java代码:

Java 代码



```
1. package cn.javass.ssonline.spider.service.impl;
2.
3. import org.junit.Test;
4. import org.junit.runner.RunWith;
5. import org.springframework.beans.factory.annotation.Autowired;
6. import org.springframework.test.context.ContextConfiguration;
7. import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
8. import org.springframework.test.context.transaction.TransactionConfiguration;
9.
10. import cn.javass.demo.service.UserService;
11. @RunWith(SpringJUnit4ClassRunner.class)
12. @ContextConfiguration(locations = {"classpath:spring-config.xml"})
13. public class UserServiceTest2 {
14.
15.     @Autowired
16.     private UserService userService;
17.     @Test
18.     public void testCreate() {
19.         userService.countAll();
20.     }
21. }
```

基础工作做好，接下来我们详细看看 Spring基于 JDK动态代理 和 CGLIB类级别代理到底有什么区别。

二、基于JDK动态代理:

java代码:



ynyee: tao哥, 问个问题。就是我service层在创建用户的时候想使 ...

第五章 编码/加密——《跟我学Shiro》

yaoweinan: 你好, 请问是不是自定义了sessionmanager 之后 r ...

第十二章 与Spring集成——《跟我学Shiro》

jinnianshilongnian: LinApex 写道请问 开涛, 有了解过组件化这一块吗? 看到了 ...

Spring4.1新特性——Spring缓存框架增强

Night舞夜: I白I 写道楼主有个疑问, 不指定id的bean, 在获取接口的实 ...

【第二章】IoC之2.3 IoC的配置使用——跟我学Spring3

LinApex: 请问 开涛, 有了解过组件化这一块吗? 看到了一个网上打着开源项目 ...

Spring4.1新特性——Spring缓存框架增强

Java代码

1.

<tx:annotation-driven transaction-manager="txManager"/>

该配置方式默认就是JDK动态代理方式

运行单元测试，核心日志如下：

java代码:

Java代码

☆

1.

2012-03-07 09:58:44 [main] DEBUG org.springframework.orm.hibernate4.HibernateTransactionManager - Creating new transaction with name [cn.javass.common.service.impl.BaseService.countAll]: PROPAGATION_REQUIRED,ISOLATION_DEFAULT; ' ' //开启事务

2.

2012-03-07 09:58:44 [main] DEBUG org.springframework.orm.hibernate4.HibernateTransactionManager - Opened new Session

3.

4.

2012-03-07 09:58:44 [main] DEBUG org.springframework.transaction.support.TransactionSynchronizationManager - Bound value [org.springframework.orm.hibernate4.SessionHolder@1184a4f] for key [org.hibernate.internal.SessionFactoryImpl@107b56e] to thread [main] //绑定session到ThreadLocal

5.

2012-03-07 09:58:44 [main] DEBUG org.springframework.transaction.support.TransactionSynchronizationManager - Initializing transaction synchronization

6.

2012-03-07 09:58:44 [main] DEBUG org.springframework.transaction.interceptor.TransactionInterceptor - Getting transaction for [cn.javass.common.service.impl.BaseService.countAll]

7.

2012-03-07 09:58:44 [main] DEBUG org.springframework.transaction.support.TransactionSynchronizationManager - Retrieved value [org.springframework.orm.hibernate4.SessionHolder@1184a4f] for key [org.hibernate.internal.SessionFactoryImpl@107b56e] bound to thread [main]

8.

2012-03-07 09:58:44 [main] DEBUG org.springframework.orm.hibernate4.HibernateTransactionManager - Found thread-bound Session

9.

10.

2012-03-07 09:58:44 [main] DEBUG org.springframework.transaction.support.TransactionSynchronizationManager - Retrieved value [org.springframework.orm.hibernate4.SessionHolder@1184a4f] for key [org.hibernate.internal.SessionFactoryImpl@107b56e] bound to thread [main]

11.

Hibernate:

12.

select

13.

count(*) as col_0_0_

14.

from

15.

tbl_user userModel0_

16.

17.

2012-03-07 09:58:44 [main] DEBUG org.springframework.orm.hibernate4.HibernateTransactionManager - Committing Hibernate transaction on Session //提交事务

18.

19.

2012-03-07 09:58:44 [main] DEBUG org.springframework.transaction.support.TransactionSynchronizationManager - Removed value [org.springframework.orm.hibernate4.SessionHolder@1184a4f] for key [org.hibernate.internal.SessionFactoryImpl@107b56e] from thread [main] //解除绑定session到ThreadLocal

到此我们可以看到事务起作用了，也就是说即使把@Transactional放到接口上 基于JDK动态代理也是可以工作的。

三、基于CGLIB类代理：

java代码:

Java代码

☆


1.

<tx:annotation-driven transaction-manager="txManager" proxy-target-class="true"/>

该配置方式是基于CGLIB类代理

启动测试会报错，No Session found for current thread，说明事务没有起作用


java代码:

Java代码 

```
1. org.hibernate.HibernateException: No Session found for current thread
2.   at org.springframework.orm.hibernate4.SpringSessionContext.currentSession(SpringSessionContext.java:97)
3.   at org.hibernate.internal.SessionFactoryImpl.getCurrentSession(SessionFactoryImpl.java:1024)
4.   at cn.javass.common.dao.hibernate4.BaseHibernateDao.getSession(BaseHibernateDao.java:63)
5.   )
6.   at cn.javass.common.dao.hibernate4.BaseHibernateDao.aggregate(BaseHibernateDao.java:238)
7.   )
8.   at cn.javass.common.dao.hibernate4.BaseHibernateDao.countAll(BaseHibernateDao.java:114)
9.   at cn.javass.common.service.impl.BaseService.countAll(BaseService.java:60)
10.  at cn.javass.common.service.impl.BaseService$$FastClassByCGLIB$$5b04dd69.invoke(<generated>)
11.  at net.sf.cglib.proxy.MethodProxy.invoke(MethodProxy.java:149)
12.  at org.springframework.aop.framework.Cglib2AopProxy$DynamicAdvisedInterceptor.intercept(Cglib2AopProxy.java:618)
    at cn.javass.demo.service.impl.UserServiceImpl$$EnhancerByCGLIB$$7d46c567.countAll(<generated>)
    at cn.javass.ssonline.spider.service.impl.UserServiceTest2.testCreate(UserServiceTest2.java:20)
```

如果将注解放在具体类上或具体类的实现方法上才会起作用。


java代码:

Java代码 

```
1. package cn.javass.common.service.impl;
2. public abstract class BaseService<M extends java.io.Serializable, PK extends java.io.Serializable> implements IBaseService<M, PK> {
3.
4.     @Transactional() //放在抽象类上
5.     @Override
6.     public int countAll() {
7.         return baseDao.countAll();
8.     }
9. }
```

运行测试类，将发现成功了，因为我们的UserService继承该方法，但如果UserService覆盖该方法，如下所示，也将无法织入事务(报错)：

java代码:

Java代码 

```
1. package cn.javass.demo.service.impl;
2. public class UserServiceImpl extends BaseService<UserModel, Integer> implements UserService {
3.
4.     //没有@Transactional
5.     @Override
6.     public int countAll() {
7.         return baseDao.countAll();
8.     }
9. }
```

```
7.     }
8. }
}
```

四、基于aspectj的

java代码:

Java代码

```
1. <tx:annotation-driven transaction-manager="txManager" mode="aspectj" proxy-target-class="true"/>
```

在此就不演示了，我们主要分析基于JDK动态代理和CGLIB类代理两种的区别。

五、结论:

基于JDK动态代理，可以将@Transactional放置在接口和具体类上。

基于CGLIB类代理，只能将@Transactional放置在具体类上。

因此 在实际开发时全部将@Transactional放到具体类上，而不是接口上。

六、分析

1、JDK动态代理

1.1、Spring使用JdkDynamicAopProxy实现代理:

java代码:

Java代码

```
1. package org.springframework.aop.framework;
2. final class JdkDynamicAopProxy implements AopProxy, InvocationHandler, Serializable {
3.     //注意此处的method 一定是接口上的method (因此放置在接口上的@Transactional是可以发现的)
4.     public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
5.     {
6.     }
```

注意此处的method 一定是接口上的method（因此放置在接口上的@Transactional是可以发现的）

1.2、如果<tx:annotation-driven 中 proxy-target-class="true"，Spring将使用CGLIB动态代理，而内部通过Cglib2AopProxy实现代理，而内部通过DynamicAdvisedInterceptor进行拦截:


java代码:

Java代码

```
1. package org.springframework.aop.framework;
2. final class Cglib2AopProxy implements AopProxy, Serializable {
3.     private static class DynamicAdvisedInterceptor implements MethodInterceptor, Serializable {
4.         //注意此处的method 一定是具体类上的method (因此只用放置在具体类上的@Transactional是可以发现的)
5.         public Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {
6.         }
7.     }
8. }
```

1.3、Spring使用AnnotationTransactionAttributeSource通过查找一个类或方法是否有@Transactional注解事务来返回TransactionAttribute（表示开启事务）：


java代码:

Java代码 

```
1. package org.springframework.transaction.annotation;
2. public class AnnotationTransactionAttributeSource extends AbstractFallbackTransactionAttributeSource implements Serializable {
3.     protected TransactionAttribute determineTransactionAttribute(AnnotatedElement ae)
4.     {
5.         for (TransactionAnnotationParser annotationParser : this.annotationParsers
6.         ) {
7.             TransactionAttribute attr = annotationParser.parseTransactionAnnotation(ae);
8.             if (attr != null) {
9.                 return attr;
10.            }
11.        }
12.        return null;
13.    }
14. }
```

而AnnotationTransactionAttributeSource又使用SpringTransactionAnnotationParser来解析是否有@Transactional注解:

java代码:

Java代码 

```
1. package org.springframework.transaction.annotation;
2.
3. public class SpringTransactionAnnotationParser implements TransactionAnnotationParser, Serializable {
4.
5.     public TransactionAttribute parseTransactionAnnotation(AnnotatedElement ae) {
6.         Transactional ann = AnnotationUtils.getAnnotation(ae, Transactional.class);
7.
8.         if (ann != null) {
9.             return parseTransactionAnnotation(ann);
10.        }
11.        else {
12.            return null;
13.        }
14.    }
15.
16.     public TransactionAttribute parseTransactionAnnotation(Transactional ann) {
17.
18.     }
19. }
```


此处使用AnnotationUtils.getAnnotation(ae, Transactional.class); 这个方法只能发现当前方法/类上的注解，不能发现父类的注解。 Spring还提供了 一个 AnnotationUtils.findAnnotation()方法 可以发现父类/父接口中的注解（但spring没有使用该接口）。

如果Spring此处换成AnnotationUtils.findAnnotation()，将可以发现父类/父接口中的注解。

这里还一个问题，描述如下：

在接口中删除@Transactional //开启默认事务


java代码：

Java代码 

```
1. package cn.javass.common.service;
2. public interface IBaseService<M extends java.io.Serializable, PK extends java.io.Serializable> {
3.     public int countAll();
4. }
```

在具体类中添加@Transactional

java代码：

Java代码 

```
1. package cn.javass.common.service.impl;
2. public abstract class BaseService<M extends java.io.Serializable, PK extends java.io.Serializable> implements IBaseService<M, PK> {
3.
4.     @Transactional() //开启默认事务
5.     @Override
6.     public int countAll() {
7.         return baseDao.countAll();
8.     }
9. }
```


问题：

我们之前说过，基于JDK动态代理时，method 一定是接口上的method（因此放置在接口上的@Transactional是可以发现的），但现在我们放在具体类上，那么Spring是如何发现的呢？

还记得发现TransactionAttribute是通过AnnotationTransactionAttributeSource吗？具体看步骤1.3：

而AnnotationTransactionAttributeSource 继承AbstractFallbackTransactionAttributeSource

java代码：

Java代码 

```
1. package org.springframework.transaction.interceptor;
2. public abstract class AbstractFallbackTransactionAttributeSource implements TransactionAttributeSource {
3.
4.     public TransactionAttribute getTransactionAttribute(Method method, Class<?> targetC
```

```
class) {
5.         //第一次 会委托给computeTransactionAttribute
6.     }
7.
8.     //计算TransactionAttribute的
9.     private TransactionAttribute computeTransactionAttribute(Method method, Class<?> ta
    rgetClass) {
10.
11.         //省略
12.
13.         // Ignore CGLIB subclasses - introspect the actual user class.
14.         Class<?> userClass = ClassUtils.getUserClass(targetClass);
15.         // The method may be on an interface, but we need attributes from the tar
    get class.
16.         // If the target class is null, the method will be unchanged.
17.         //①此处将查找当前类覆盖的方法
18.         Method specificMethod = ClassUtils.getMostSpecificMethod(method, userClass)
    ;
19.         // If we are dealing with method with generic parameters, find the origin
    al method.
20.         specificMethod = BridgeMethodResolver.findBridgedMethod(specificMethod);
21.
22.         // First try is the method in the target class.
23.         TransactionAttribute txAtt = findTransactionAttribute(specificMethod);
24.         if (txAtt != null) {
25.             return txAtt;
26.         }
27.
28.         //找类上边的注解
29.         // Second try is the transaction attribute on the target class.
30.         txAtt = findTransactionAttribute(specificMethod.getDeclaringClass());
31.         if (txAtt != null) {
32.             return txAtt;
33.         }
34.         //②如果子类覆盖的方法没有 再直接找当前传过来的
35.         if (specificMethod != method) {
36.             // Fallback is to look at the original method.
37.             txAtt = findTransactionAttribute(method);
38.             if (txAtt != null) {
39.                 return txAtt;
40.             }
41.             // Last fallback is the class of the original method.
42.             return findTransactionAttribute(method.getDeclaringClass());
43.         }
44.         return null;
45.     }
46. }
```

```
//①此处将查找子类覆盖的方法
Method specificMethod = ClassUtils.getMostSpecificMethod(method, userClass);

// ClassUtils.getMostSpecificMethod
public static Method getMostSpecificMethod(Method method, Class<?> targetClass) {
    Method specificMethod = null;
    if (method != null && isOverridable(method, targetClass) &&
        targetClass != null && !targetClass.equals(method.getDeclaringClass())) {
        try {
            specificMethod = ReflectionUtils.findMethod(targetClass, method.getName(),
```



```
method.getParameterTypes());
    } catch (AccessControlException ex) {
        // security settings are disallowing reflective access; leave
        // 'specificMethod' null and fall back to 'method' below
    }
}
return (specificMethod != null ? specificMethod : method);
}
```

可以看出将找到当前类的那个方法。因此我们放置在BaseService countAll方法上的@Transactional起作用了。

```
//②如果子类覆盖的方法没有 再直接找当前传过来的
if (specificMethod != method) {
    // Fallback is to look at the original method.
    txAtt = findTransactionAttribute(method);
    if (txAtt != null) {
        return txAtt;
    }
    // Last fallback is the class of the original method.
    return findTransactionAttribute(method.getDeclaringClass());
}
```

查找子类失败时直接使用传过来的方法。

因此，建议大家使用基于Schema风格的事务（不用考虑这么多问题，也不用考虑是类还是方法）。而@Transactional建议放置到具体类上，不要放置到接口。

作者原创【<http://sishuok.com/forum/blogPost/list/0/3845.html#9317>】



6

顶

1

踩

分享到：

◀ [Spring3 Web MVC下的数据类型转换（第一篇 ...](#) | [Spring Web MVC中的页面缓存支持 ——跟我 ...](#) ▶

2012-05-02 22:41 | 浏览 6457 | 评论(7) | 分类:企业架构 | 相关推荐 [▶ MORE](#)

评论

7 楼 [calmfire](#) 2014-05-06

calmfire 写道

开涛兄，使用JDK动态代理，注解@Transactional在具体子类上，当子类有自定义方法时，context.getBean便会出现以下异常，何故？
Exception in thread "main" org.springframework.beans.factory.BeanNotOfRequiredTypeException: Bean named 'testDao' must be of type [com.test.dao.TestDao], but was actually of type [com.sun.proxy.\$Proxy13]
at org.springframework.beans.factory.support.AbstractBeanFactory.doGetBean(AbstractBeanFactory.java:376)
at org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:200)
at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:979)
at test.Test.main(Test.java:12)

了解了，JDK代理是动态创建实现接口的子类，CGLIB是针对子类的扩展。所以上面会出现cast问题。

6 楼 [calmfire](#) 2014-05-06

开涛兄，使用JDK动态代理，注解@Transactional在具体子类上，当子类有自定义方法时，context.getBean便会出现以下异常，何故？
Exception in thread "main" org.springframework.beans.factory.BeanNotOfRequiredTypeException: Bean named 'testDao' must be of type [com.test.dao.TestDao], but was actually of type [com.sun.proxy.\$Proxy13]
at org.springframework.beans.factory.support.AbstractBeanFactory.doGetBean(AbstractBeanFactory.java:376)
at org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:200)
at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:979)
at test.Test.main(Test.java:12)

5 楼 [hekuilove](#) 2012-10-29

jdk动态代理只支持接口代理，cglib可以支持类代理

4 楼 [jinnianshilongnian](#) 2012-09-07

rentianchou 写道

如雷贯耳👍

谢谢

3 楼 [rentianchou](#) 2012-09-07

如雷贯耳👍

2 楼 [jinnianshilongnian](#) 2012-05-18

飞天奔月 写道

佩服 这两个的区别都研究得这么透彻

核心就是 aop时 JDK动态代理 CGLIB类代理 的区别不？

对 是的

1 楼 [飞天奔月](#) 2012-05-18

佩服 这两个的区别都研究得这么透彻

核心就是 aop时 JDK动态代理 CGLIB类代理 的区别不？

发表评论

[您还没有登录,请您登录后再发表评论](#)



声明：ITeye 文章版权属于作者，受法律保护。没有作者书面许可不得转载。若作者同意转载，必须以超链接形式标明文章原始出处和作者。
© 2003-2014 ITeye.com. All rights reserved. [京ICP证110151号 京公网安备110105010620]