https://www.ibm.com/developerworks/cn/java/j-master-spring-transactional-use/index.html

事务管理是应用系统开发中必不可少的一部分。Spring 为事务管理提供了丰富的功能支持。Spring 事务管理分为编码式和声明式的两种方式。编程式事务指的是通过编码方式实现事务；声明式事务基于 AOP,将具体业务逻辑与事务处理解耦。声明式事务管理使业务代码逻辑不受污染, 因此在实际使用中声明式事务用的比较多。声明式事务有两种方式，一种是在配置文件（xml）中做相关的事务规则声明，另一种是基于@Transactional 注解的方式。注释配置是目前流行的使用方式，因此本文将着重介绍基于@Transactional 注解的事务管理。

@Transactional 注解管理事务的实现步骤

使用@Transactional 注解管理事务的实现步骤分为两步。第一步，在 xml 配置文件中添加如清单 1 的事务配置信息。除了用配置文件的方式，@EnableTransactionManagement 注解也可以启用事务管理功能。这里以简单的 DataSourceTransactionManager 为例。

清单 1. 在 xml 配置中的事务配置信息

|  |  |
| --- | --- |
| 1  2  3  4  5 | <tx:annotation-driven />  <bean id="transactionManager"  class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <property name="dataSource" ref="dataSource" />  </bean> |

第二步，将@Transactional 注解添加到合适的方法上，并设置合适的属性信息。@Transactional 注解的属性信息如表 1 展示。

表 1. @Transactional 注解的属性信息

| **属性名** | **说明** |
| --- | --- |
| name | 当在配置文件中有多个 TransactionManager , 可以用该属性指定选择哪个事务管理器。 |
| propagation | 事务的传播行为，默认值为 REQUIRED。 |
| isolation | 事务的隔离度，默认值采用 DEFAULT。 |
| timeout | 事务的超时时间，默认值为-1。如果超过该时间限制但事务还没有完成，则自动回滚事务。 |
| read-only | 指定事务是否为只读事务，默认值为 false；为了忽略那些不需要事务的方法，比如读取数据，可以设置 read-only 为 true。 |
| rollback-for | 用于指定能够触发事务回滚的异常类型，如果有多个异常类型需要指定，各类型之间可以通过逗号分隔。 |
| no-rollback- for | 抛出 no-rollback-for 指定的异常类型，不回滚事务。 |

除此以外，@Transactional 注解也可以添加到类级别上。当把@Transactional 注解放在类级别时，表示所有该类的公共方法都配置相同的事务属性信息。见清单 2，EmployeeService 的所有方法都支持事务并且是只读。当类级别配置了@Transactional，方法级别也配置了@Transactional，应用程序会以方法级别的事务属性信息来管理事务，换言之，方法级别的事务属性信息会覆盖类级别的相关配置信息。

清单 2. @Transactional 注解的类级别支持

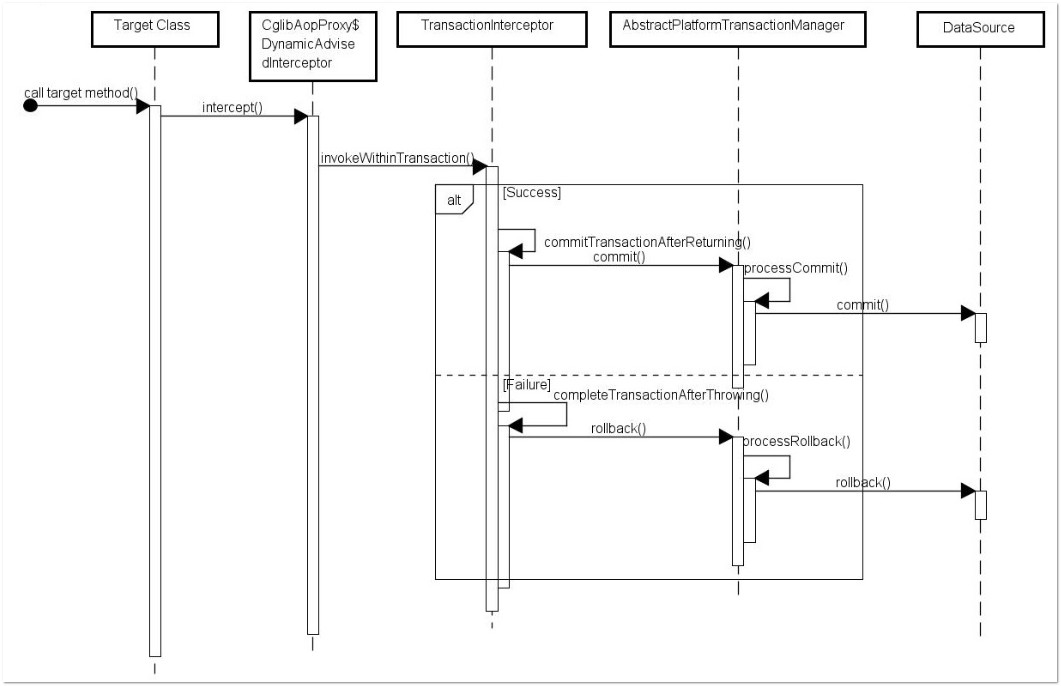
|  |  |
| --- | --- |
| 1  2  3 | @Transactional(propagation= Propagation.SUPPORTS,readOnly=true)  @Service(value ="employeeService")  public class EmployeeService |

到此，您会发觉使用@Transactional 注解管理事务的实现步骤很简单。但是如果对 Spring 中的 @transaction 注解的事务管理理解的不够透彻，就很容易出现错误，比如事务应该回滚（rollback）而没有回滚事务的问题。接下来，将首先分析 Spring 的注解方式的事务实现机制，然后列出相关的注意事项，以最终达到帮助开发人员准确而熟练的使用 Spring 的事务的目的。

Spring 的注解方式的事务实现机制

在应用系统调用声明@Transactional 的目标方法时，Spring Framework 默认使用 AOP 代理，在代码运行时生成一个代理对象，根据@Transactional 的属性配置信息，这个代理对象决定该声明@Transactional 的目标方法是否由拦截器 TransactionInterceptor 来使用拦截，在 TransactionInterceptor 拦截时，会在在目标方法开始执行之前创建并加入事务，并执行目标方法的逻辑, 最后根据执行情况是否出现异常，利用抽象事务管理器(图 2 有相关介绍)AbstractPlatformTransactionManager 操作数据源 DataSource 提交或回滚事务, 如图 1 所示。

图 1. Spring 事务实现机制



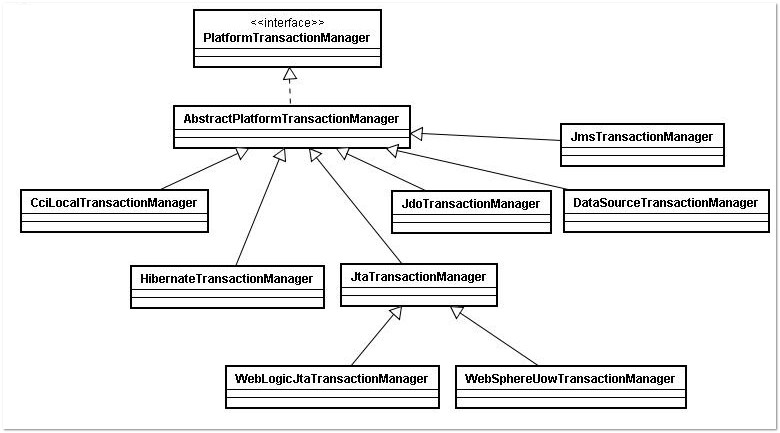
[点击查看大图](https://www.ibm.com/developerworks/cn/java/j-master-spring-transactional-use/index.html#N100A6)

Spring AOP 代理有 CglibAopProxy 和 JdkDynamicAopProxy 两种，图 1 是以 CglibAopProxy 为例，对于 CglibAopProxy，需要调用其内部类的 DynamicAdvisedInterceptor 的 intercept 方法。对于 JdkDynamicAopProxy，需要调用其 invoke 方法。

正如上文提到的，事务管理的框架是由抽象事务管理器 AbstractPlatformTransactionManager 来提供的，而具体的底层事务处理实现，由 PlatformTransactionManager 的具体实现类来实现，如事务管理器 DataSourceTransactionManager。不同的事务管理器管理不同的数据资源 DataSource，比如 DataSourceTransactionManager 管理 JDBC 的 Connection。

PlatformTransactionManager，AbstractPlatformTransactionManager 及具体实现类关系如图 2 所示。

图 2. TransactionManager 类结构



[点击查看大图](https://www.ibm.com/developerworks/cn/java/j-master-spring-transactional-use/index.html#N100B6)

注解方式的事务使用注意事项

当您对 Spring 的基于注解方式的实现步骤和事务内在实现机制有较好的理解之后，就会更好的使用注解方式的事务管理，避免当系统抛出异常，数据不能回滚的问题。

正确的设置@Transactional 的 propagation 属性

需要注意下面三种 propagation 可以不启动事务。本来期望目标方法进行事务管理，但若是错误的配置这三种 propagation，事务将不会发生回滚。

1. TransactionDefinition.PROPAGATION\_SUPPORTS：如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。
2. TransactionDefinition.PROPAGATION\_NOT\_SUPPORTED：以非事务方式运行，如果当前存在事务，则把当前事务挂起。
3. TransactionDefinition.PROPAGATION\_NEVER：以非事务方式运行，如果当前存在事务，则抛出异常。

* TransactionDefinition.PROPAGATION\_REQUIRED：如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。这是默认值。
* TransactionDefinition.PROPAGATION\_REQUIRES\_NEW：创建一个新的事务，如果当前存在事务，则把当前事务挂起。
* TransactionDefinition.PROPAGATION\_SUPPORTS：如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。
* TransactionDefinition.PROPAGATION\_NOT\_SUPPORTED：以非事务方式运行，如果当前存在事务，则把当前事务挂起。
* TransactionDefinition.PROPAGATION\_NEVER：以非事务方式运行，如果当前存在事务，则抛出异常。
* TransactionDefinition.PROPAGATION\_MANDATORY：如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。
* TransactionDefinition.PROPAGATION\_NESTED：如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于TransactionDefinition.PROPAGATION\_REQUIRED。
* **事务隔离级别:**
* @Transactional(isolation = Isolation.READ\_UNCOMMITTED)：读取未提交数据(会出现脏读, 不可重复读) 基本不使用  
  　　@Transactional(isolation = Isolation.READ\_COMMITTED)：读取已提交数据(会出现不可重复读和幻读)  
  　　@Transactional(isolation = Isolation.REPEATABLE\_READ)：可重复读(会出现幻读)  
  　　@Transactional(isolation = Isolation.SERIALIZABLE)：串行化
* MYSQL: 默认为REPEATABLE\_READ级别  
  　　SQLSERVER: 默认为READ\_COMMITTED
* **脏读** : 一个事务读取到另一事务未提交的更新数据  
  **不可重复读** : 在同一事务中, 多次读取同一数据返回的结果有所不同, 换句话说,   
  后续读取可以读到另一事务已提交的更新数据. 相反, "可重复读"在同一事务中多次  
  读取数据时, 能够保证所读数据一样, 也就是后续读取不能读到另一事务已提交的更新数据  
  **幻读** : 一个事务读到另一个事务已提交的insert数据

正确的设置@Transactional 的 rollbackFor 属性

默认情况下，如果在事务中抛出了未检查异常（继承自 RuntimeException 的异常）或者 Error，则 Spring 将回滚事务；除此之外，Spring 不会回滚事务。

如果在事务中抛出其他类型的异常，并期望 Spring 能够回滚事务，可以指定 rollbackFor。例：

@Transactional(propagation= Propagation.REQUIRED,rollbackFor= MyException.class)

通过分析 Spring 源码可以知道，若在目标方法中抛出的异常是 rollbackFor 指定的异常的子类，事务同样会回滚。

清单 3. RollbackRuleAttribute 的 getDepth 方法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | private int getDepth(Class<?> exceptionClass, int depth) {          if (exceptionClass.getName().contains(this.exceptionName)) {              // Found it!              return depth;  }          // If we've gone as far as we can go and haven't found it...          if (exceptionClass == Throwable.class) {              return -1;  }  return getDepth(exceptionClass.getSuperclass(), depth + 1);  } |

@Transactional 只能应用到 public 方法才有效

只有@Transactional 注解应用到 public 方法，才能进行事务管理。这是因为在使用 Spring AOP 代理时，Spring 在调用在图 1 中的 TransactionInterceptor 在目标方法执行前后进行拦截之前，DynamicAdvisedInterceptor（CglibAopProxy 的内部类）的的 intercept 方法或 JdkDynamicAopProxy 的 invoke 方法会间接调用 AbstractFallbackTransactionAttributeSource（Spring 通过这个类获取表 1. @Transactional 注解的事务属性配置属性信息）的 computeTransactionAttribute 方法。

清单 4. AbstractFallbackTransactionAttributeSource

|  |  |
| --- | --- |
| 1  2  3  4  5 | protected TransactionAttribute computeTransactionAttribute(Method method,      Class<?> targetClass) {          // Don't allow no-public methods as required.          if (allowPublicMethodsOnly() && !Modifier.isPublic(method.getModifiers())) {  return null;} |

这个方法会检查目标方法的修饰符是不是 public，若不是 public，就不会获取@Transactional 的属性配置信息，最终会造成不会用 TransactionInterceptor 来拦截该目标方法进行事务管理。

避免 Spring 的 AOP 的自调用问题

在 Spring 的 AOP 代理下，只有目标方法由外部调用，目标方法才由 Spring 生成的代理对象来管理，这会造成自调用问题。若同一类中的其他没有@Transactional 注解的方法内部调用有@Transactional 注解的方法，有@Transactional 注解的方法的事务被忽略，不会发生回滚。见清单 5 举例代码展示。

清单 5.自调用问题举例

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @Service  -->public class OrderService {      private void insert() {  insertOrder();  }  @Transactional      public void insertOrder() {          //insert log info          //insertOrder          //updateAccount         }  } |

insertOrder 尽管有@Transactional 注解，但它被内部方法 insert 调用，事务被忽略，出现异常事务不会发生回滚。

上面的两个问题@Transactional 注解只应用到 public 方法和自调用问题，是由于使用 Spring AOP 代理造成的。为解决这两个问题，使用 AspectJ 取代 Spring AOP 代理。

需要将下面的 AspectJ 信息添加到 xml 配置信息中。

清单 6. AspectJ 的 xml 配置信息

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <tx:annotation-driven mode="aspectj" />  <bean id="transactionManager"  class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <property name="dataSource" ref="dataSource" />  </bean>  </bean  class="org.springframework.transaction.aspectj.AnnotationTransactionAspect"  factory-method="aspectOf">  <property name="transactionManager" ref="transactionManager" />  </bean> |

同时在 Maven 的 pom 文件中加入 spring-aspects 和 aspectjrt 的 dependency 以及 aspectj-maven-plugin。

清单 7. AspectJ 的 pom 配置信息

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aspects</artifactId>  <version>4.3.2.RELEASE</version>  </dependency>  <dependency>  <groupId>org.aspectj</groupId>  <artifactId>aspectjrt</artifactId>  <version>1.8.9</version>  </dependency>  <plugin>  <groupId>org.codehaus.mojo</groupId>  <artifactId>aspectj-maven-plugin</artifactId>  <version>1.9</version>  <configuration>  <showWeaveInfo>true</showWeaveInfo>  <aspectLibraries>  <aspectLibrary>  <groupId>org.springframework</groupId>  <artifactId>spring-aspects</artifactId>  </aspectLibrary>  </aspectLibraries>  </configuration>  <executions>  <execution>  <goals>  <goal>compile</goal>  <goal>test-compile</goal>  </goals>  </execution>  </executions>  </plugin> |

总结

通过本文的介绍，相信读者能够清楚的了解基于@Transactional 注解的实现步骤，能够透彻的理解的 Spring 的内部实现机制，并有效的掌握相关使用注意事项，从而能够正确而熟练的使用基于@Transactional 注解的事务管理方式。

参考资源

* [Spring in Action 英文第四版](https://www.manning.com/books/spring-in-action-fourth-edition)：深入的介绍了 Spring 背后的原理,引领读者全面的理解 Spring 的框架。
* 您可以通过 developerWorks 社区与[刘万振](https://www.ibm.com/developerworks/mydeveloperworks/profiles/user/fisher_liu)进行交流。