小结:

　编程式事务管理:

　１　基于底层ＡＰＩ(platFormTransactionManager, TransactionStauts,,TransactionDefinition) 的编程式事务管理

　　　２.　基于TransactionTemplate的编程式事务管理(对比基于底层ＡＰＩ的编程式对于事务的管理启动,提交,会滚放到template中)　声明式事务管理:　　实现原理基于Spring AOP.

1. 基于TranacitonInterceptor的声明式事务管理.利用transacitonInterceptor 来声明transacton属性,　然后结合

proxyFactoryBean来对单个的bean管理,或是自动代理BeanNameAutoProxyCreator来进行代理.

1. 基于TransactionProxyFactoryBean 的声明式事务管理..在TransactionProxyFactoryBean 内声明transaction 属性,　然后

直接把targetBean和TansactionManager 绑定,或者是把声明的TransactionProxyFactoryBean　bean 做base类.

　　　３. .基于 <tx> 命名空间的声明式事务管理　以tx:advice 的方式来声明事务,然后结合aop:config 命名空间来绑定pointcut(需

要被此事务管理的bean) 和声明为advice 的事务.

　　　４. 基于 @Transactional 的声明式事务管理　@Transactional 可以作用于接口、接口方法、类以及类方法上　但是必须声明:

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

**Spring 事务属性分析**

事务管理对于企业应用而言至关重要。它保证了用户的每一次操作都是可靠的，即便出现了异常的访问情况，也不至于破坏后台数据的完整性。就像银行的自助取款机，通常都能正常为客户服务，但是也难免遇到操作过程中机器突然出故障的情况，此时，事务就必须确保出故障前对账户的操作不生效，就像用户刚才完全没有使用过取款机一样，以保证用户和银行的利益都不受损失。

在 Spring 中，事务是通过 TransactionDefinition 接口来定义的。该接口包含与事务属性有关的方法。具体如清单1所示：

清单1. TransactionDefinition 接口中定义的主要方法

|  |
| --- |
| public interface TransactionDefinition{  int getIsolationLevel();  int getPropagationBehavior();  int getTimeout();  boolean isReadOnly();  } |

也许你会奇怪，为什么接口只提供了获取属性的方法，而没有提供相关设置属性的方法。其实道理很简单，事务属性的设置完全是程序员控制的，因此程序员可以自定义任何设置属性的方法，而且保存属性的字段也没有任何要求。唯一的要求的是，Spring 进行事务操作的时候，通过调用以上接口提供的方法必须能够返回事务相关的属性取值。

事务隔离级别

隔离级别是指若干个并发的事务之间的隔离程度。TransactionDefinition 接口中定义了五个表示隔离级别的常量：

·         TransactionDefinition.ISOLATION\_DEFAULT：这是默认值，表示使用底层数据库的默认隔离级别。对大部分数据库而言，通常这值就是TransactionDefinition.ISOLATION\_READ\_COMMITTED。

·         TransactionDefinition.ISOLATION\_READ\_UNCOMMITTED：该隔离级别表示一个事务可以读取另一个事务修改但还没有提交的数据。该级别不能防止脏读和不可重复读，因此很少使用该隔离级别。

·         TransactionDefinition.ISOLATION\_READ\_COMMITTED：该隔离级别表示一个事务只能读取另一个事务已经提交的数据。该级别可以防止脏读，这也是大多数情况下的推荐值。

·         TransactionDefinition.ISOLATION\_REPEATABLE\_READ：该隔离级别表示一个事务在整个过程中可以多次重复执行某个查询，并且每次返回的记录都相同。即使在多次查询之间有新增的数据满足该查询，这些新增的记录也会被忽略。该级别可以防止脏读和不可重复读。

·         TransactionDefinition.ISOLATION\_SERIALIZABLE：所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能。通常情况下也不会用到该级别。

事务传播行为

所谓事务的传播行为是指，如果在开始当前事务之前，一个事务上下文已经存在，此时有若干选项可以指定一个事务性方法的执行行为。在TransactionDefinition定义中包括了如下几个表示传播行为的常量：

·         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。

·         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。**嵌套事务一个非常重要的概念就是内层事务依赖于外层事务。外层事务失败时，会回滚内层事务所做的动作。而内层事务操作失败并不会引起外层事务的回滚**。

这里需要指出的是，前面的六种事务传播行为是 Spring 从 EJB 中引入的，他们共享相同的概念。而 PROPAGATION\_NESTED是 Spring 所特有的。以 PROPAGATION\_NESTED 启动的事务内嵌于外部事务中（如果存在外部事务的话），此时，内嵌事务并不是一个独立的事务，它依赖于外部事务的存在，只有通过外部的事务提交，才能引起内部事务的提交，嵌套的子事务不能单独提交。如果熟悉 JDBC 中的保存点（SavePoint）的概念，那嵌套事务就很容易理解了，其实嵌套的子事务就是保存点的一个应用，一个事务中可以包括多个保存点，每一个嵌套子事务。另外，外部事务的回滚也会导致嵌套子事务的回滚。

事务超时

所谓事务超时，就是指一个事务所允许执行的最长时间，如果超过该时间限制但事务还没有完成，则自动回滚事务。在 TransactionDefinition 中以 int 的值来表示超时时间，其单位是秒。

事务的只读属性

事务的只读属性是指，对事务性资源进行只读操作或者是读写操作。所谓事务性资源就是指那些被事务管理的资源，比如数据源、 JMS 资源，以及自定义的事务性资源等等。如果确定只对事务性资源进行只读操作，那么我们可以将事务标志为只读的，以提高事务处理的性能。在 TransactionDefinition 中以 boolean 类型来表示该事务是否只读。

事务的回滚规则

通常情况下，如果在事务中抛出了未检查异常（继承自 RuntimeException 的异常），则默认将回滚事务。如果没有抛出任何异常，或者抛出了已检查异常，则仍然提交事务。这通常也是大多数开发者希望的处理方式，也是 EJB 中的默认处理方式。但是，我们可以根据需要人为控制事务在抛出某些未检查异常时任然提交事务，或者在抛出某些已检查异常时回滚事务。

**Spring 事务管理 API 分析**

Spring 框架中，涉及到事务管理的 API 大约有100个左右，其中最重要的有三个：TransactionDefinition、PlatformTransactionManager、TransactionStatus。所谓事务管理，其实就是“按照给定的事务规则来执行提交或者回滚操作”。“给定的事务规则”就是用 TransactionDefinition 表示的，“按照……来执行提交或者回滚操作”便是用 PlatformTransactionManager 来表示，而 TransactionStatus 用于表示一个运行着的事务的状态。打一个不恰当的比喻，TransactionDefinition 与 TransactionStatus 的关系就像程序和进程的关系。

TransactionDefinition

该接口在前面已经介绍过，它用于定义一个事务。它包含了事务的静态属性，比如：事务传播行为、超时时间等等。Spring 为我们提供了一个默认的实现类：DefaultTransactionDefinition，该类适用于大多数情况。如果该类不能满足需求，可以通过实现 TransactionDefinition 接口来实现自己的事务定义。

**PlatformTransactionManager**

PlatformTransactionManager 用于执行具体的事务操作。接口定义如清单2所示：

清单2. PlatformTransactionManager 接口中定义的主要方法

|  |
| --- |
| Public interface PlatformTransactionManager{   TransactionStatus getTransaction(TransactionDefinition definition)     throws TransactionException;     void commit(TransactionStatus status)throws TransactionException;     void rollback(TransactionStatus status)throws TransactionException;  } |

根据底层所使用的不同的持久化 API 或框架，PlatformTransactionManager 的主要实现类大致如下：

·         DataSourceTransactionManager：适用于使用JDBC和iBatis进行数据持久化操作的情况。

·         HibernateTransactionManager：适用于使用Hibernate进行数据持久化操作的情况。

·         JpaTransactionManager：适用于使用JPA进行数据持久化操作的情况。

·         另外还有JtaTransactionManager 、JdoTransactionManager、JmsTransactionManager等等。

如果我们使用JTA进行事务管理，我们可以通过 JNDI 和 Spring 的 JtaTransactionManager 来获取一个容器管理的 DataSource。JtaTransactionManager 不需要知道 DataSource 和其他特定的资源，因为它将使用容器提供的全局事务管理。而对于其他事务管理器，比如DataSourceTransactionManager，在定义时需要提供底层的数据源作为其属性，也就是 DataSource。与 HibernateTransactionManager 对应的是 SessionFactory，与 JpaTransactionManager 对应的是 EntityManagerFactory 等等。

TransactionStatus

PlatformTransactionManager.getTransaction(…) 方法返回一个 TransactionStatus 对象。返回的TransactionStatus 对象可能代表一个新的或已经存在的事务（如果在当前调用堆栈有一个符合条件的事务）。TransactionStatus 接口提供了一个简单的控制事务执行和查询事务状态的方法。该接口定义如清单3所示：

清单3. TransactionStatus 接口中定义的主要方法

|  |
| --- |
| public interface TransactionStatus{     boolean isNewTransaction();     void setRollbackOnly();     boolean isRollbackOnly();  } |

**编程式事务管理**

Spring **的编程式事务管理概述**

在 Spring 出现以前，编程式事务管理对基于 POJO 的应用来说是唯一选择。用过 Hibernate 的人都知道，我们需要在代码中显式调用beginTransaction()、commit()、rollback()等事务管理相关的方法，这就是编程式事务管理。通过 Spring 提供的事务管理 API，我们可以在代码中灵活控制事务的执行。在底层，Spring 仍然将事务操作委托给底层的持久化框架来执行。

基于底层 **API 的编程式事务管理**

根据PlatformTransactionManager、TransactionDefinition 和 TransactionStatus 三个核心接口，我们完全可以通过编程的方式来进行事务管理。示例代码如清单4所示：

清单4. 基于底层 API 的事务管理示例代码

|  |
| --- |
| public class BankServiceImpl implements BankService {  private BankDao bankDao;  private TransactionDefinition txDefinition;  private PlatformTransactionManager txManager;  ......  public boolean transfer(Long fromId， Long toId， double amount) {  TransactionStatus txStatus = txManager.getTransaction(txDefinition);  boolean result = false;  try {  result = bankDao.transfer(fromId， toId， amount);  txManager.commit(txStatus);  } catch (Exception e) {  result = false;  txManager.rollback(txStatus);  System.out.println("Transfer Error!");  }  return result;  }  } |

相应的配置文件如清单5所示：

清单5. 基于底层API的事务管理示例配置文件

|  |
| --- |
| <bean id="bankService" class="footmark.spring.core.tx.programmatic.origin.BankServiceImpl">  <property name="bankDao" ref="bankDao"/>  <property name="txManager" ref="transactionManager"/>  <property name="txDefinition">  <bean class="org.springframework.transaction.support.DefaultTransactionDefinition">  <property name="propagationBehaviorName" value="PROPAGATION\_REQUIRED"/>  </bean>  </property>  </bean> |

如上所示，我们在类中增加了两个属性：一个是 TransactionDefinition 类型的属性，它用于定义一个事务；另一个是 PlatformTransactionManager 类型的属性，用于执行事务管理操作。

如果方法需要实施事务管理，我们首先需要在方法开始执行前启动一个事务，调用PlatformTransactionManager.getTransaction(...) 方法便可启动一个事务。创建并启动了事务之后，便可以开始编写业务逻辑代码，然后在适当的地方执行事务的提交或者回滚。

基于 **TransactionTemplate 的编程式事务管理**

通过前面的示例可以发现，这种事务管理方式很容易理解，但令人头疼的是，事务管理的代码散落在业务逻辑代码中，破坏了原有代码的条理性，并且每一个业务方法都包含了类似的启动事务、提交/回滚事务的样板代码。幸好，Spring 也意识到了这些，并提供了简化的方法，这就是 Spring 在数据访问层非常常见的模板回调模式。如清单6所示：

清单6. 基于 TransactionTemplate 的事务管理示例代码

1.    public class BankServiceImpl implements BankService {

2.      private BankDao bankDao;

3.      private TransactionTemplate transactionTemplate;

4.    ......

5.

6.      public boolean transfer(final Long fromId， final Long toId， final double amount) {

7.

8.    return (Boolean) transactionTemplate.execute(new TransactionCallback(){

9.

10. public Object doInTransaction(TransactionStatus status) {

11. Object result;

12. try {

13.   result = bankDao.transfer(fromId， toId， amount);

14. } catch (Exception e) {

15.   status.setRollbackOnly();

16.   result = false;

17. System.out.println("Transfer Error!");

18. }

19. return result;

20. }

21. });

22. }

23. }



相应的XML配置如下：

清单 7. 基于 TransactionTemplate 的事务管理示例配置文件

|  |
| --- |
| <bean id="bankService"  class="footmark.spring.core.tx.programmatic.template.BankServiceImpl">  <property name="bankDao" ref="bankDao"/>  <property name="transactionTemplate" ref="transactionTemplate"/>  </bean> |

TransactionTemplate 的 execute() 方法有一个 TransactionCallback 类型的参数，该接口中定义了一个 doInTransaction() 方法，通常我们以匿名内部类的方式实现 TransactionCallback 接口，并在其 doInTransaction() 方法中书写业务逻辑代码。这里可以使用默认的事务提交和回滚规则，这样在业务代码中就不需要显式调用任何事务管理的 API。doInTransaction() 方法有一个TransactionStatus 类型的参数，我们可以在方法的任何位置调用该参数的 setRollbackOnly() 方法将事务标识为回滚的，以执行事务回滚。

根据默认规则，如果在执行回调方法的过程中抛出了未检查异常，或者显式调用了TransacationStatus.setRollbackOnly() 方法，则回滚事务；如果事务执行完成或者抛出了 checked 类型的异常，则提交事务。

TransactionCallback 接口有一个子接口 TransactionCallbackWithoutResult，该接口中定义了一个 doInTransactionWithoutResult() 方法，TransactionCallbackWithoutResult 接口主要用于事务过程中不需要返回值的情况。当然，对于不需要返回值的情况，我们仍然可以使用 TransactionCallback 接口，并在方法中返回任意值即可。

**声明式事务管理**

Spring **的声明式事务管理概述**

Spring 的声明式事务管理在底层是建立在 AOP 的基础之上的。其本质是对方法前后进行拦截，然后在目标方法开始之前创建或者加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。

声明式事务最大的优点就是不需要通过编程的方式管理事务，这样就不需要在业务逻辑代码中掺杂事务管理的代码，只需在配置文件中做相关的事务规则声明（或通过等价的基于标注的方式），便可以将事务规则应用到业务逻辑中。因为事务管理本身就是一个典型的横切逻辑，正是 AOP 的用武之地。Spring 开发团队也意识到了这一点，为声明式事务提供了简单而强大的支持。

声明式事务管理曾经是 EJB 引以为傲的一个亮点，如今 Spring 让 POJO 在事务管理方面也拥有了和 EJB 一样的待遇，让开发人员在 EJB 容器之外也用上了强大的声明式事务管理功能，这主要得益于 Spring 依赖注入容器和 Spring AOP 的支持。依赖注入容器为声明式事务管理提供了基础设施，使得 Bean 对于 Spring 框架而言是可管理的；而 Spring AOP 则是声明式事务管理的直接实现者，这一点通过清单8可以看出来。

通常情况下，笔者强烈建议在开发中使用声明式事务，不仅因为其简单，更主要是因为这样使得纯业务代码不被污染，极大方便后期的代码维护。

和编程式事务相比，声明式事务唯一不足地方是，后者的最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别。但是即便有这样的需求，也存在很多变通的方法，比如，可以将需要进行事务管理的代码块独立为方法等等。

下面就来看看 Spring 为我们提供的声明式事务管理功能。

基于**TransactionInterceptor 的声明式事务管理**

最初，Spring 提供了 TransactionInterceptor 类来实施声明式事务管理功能。先看清单8的配置文件：

清单 8. 基于 TransactionInterceptor 的事务管理示例配置文件

|  |
| --- |
| <beans...>  ......  <bean id="transactionInterceptor"  class="org.springframework.transaction.interceptor.TransactionInterceptor">  <property name="transactionManager" ref="transactionManager"/>  <property name="transactionAttributes">  <props>  <prop key="transfer">PROPAGATION\_REQUIRED</prop>  </props>  </property>  </bean>  <bean id="bankServiceTarget"  class="footmark.spring.core.tx.declare.origin.BankServiceImpl">  <property name="bankDao" ref="bankDao"/>  </bean>  **//方式一:　使用proxyFactoryBean 来组装　target　和　interceptor**  <bean id="bankService"  　class="org.springframework.aop.framework.ProxyFactoryBean">  <property name="target" ref="bankServiceTarget"/>  <property name="interceptorNames">  <list>  <idref bean="transactionInterceptor"/>  </list>  </property>  </bean>    **//方式二:　使用beanNameAutoProxyCreator 来植入interceptor.**  <bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">          <property name="beanNames">               <list>                   <value>\*Dao</value>             </list>           </property>           <property name="interceptorNames">               <list>                   <value>transactionInterceptor</value>               </list>           </property>       </bean>  ......  </beans> |

首先，我们配置了一个 TransactionInterceptor 来定义相关的事务规则，他有两个主要的属性：一个是 transactionManager，用来指定一个事务管理器，并将具体事务相关的操作委托给它；另一个是 Properties 类型的 transactionAttributes 属性，它主要用来定义事务规则，该属性的每一个键值对中，键指定的是方法名，方法名可以使用通配符，而值就表示相应方法的所应用的事务属性。

指定事务属性的取值有较复杂的规则，这在 Spring 中算得上是一件让人头疼的事。具体的书写规则如下：

|  |
| --- |
| 传播行为 [，隔离级别] [，只读属性] [，超时属性] [不影响提交的异常] [，导致回滚的异常] |

·         传播行为是唯一必须设置的属性，其他都可以忽略，Spring为我们提供了合理的默认值。

·         传播行为的取值必须以“PROPAGATION\_”开头，具体包括：PROPAGATION\_MANDATORY、PROPAGATION\_NESTED、PROPAGATION\_NEVER、PROPAGATION\_NOT\_SUPPORTED、PROPAGATION\_REQUIRED、PROPAGATION\_REQUIRES\_NEW、PROPAGATION\_SUPPORTS，共七种取值。

·         隔离级别的取值必须以“ISOLATION\_”开头，具体包括：ISOLATION\_DEFAULT、ISOLATION\_READ\_COMMITTED、ISOLATION\_READ\_UNCOMMITTED、ISOLATION\_REPEATABLE\_READ、ISOLATION\_SERIALIZABLE，共五种取值。

·         如果事务是只读的，那么我们可以指定只读属性，使用“readOnly”指定。否则我们不需要设置该属性。

·         超时属性的取值必须以“TIMEOUT\_”开头，后面跟一个int类型的值，表示超时时间，单位是秒。

·         不影响提交的异常是指，即使事务中抛出了这些类型的异常，事务任然正常提交。必须在每一个异常的名字前面加上“+”。异常的名字可以是类名的一部分。比如“+RuntimeException”、“+tion”等等。

·         导致回滚的异常是指，当事务中抛出这些类型的异常时，事务将回滚。必须在每一个异常的名字前面加上“-”。异常的名字可以是类名的全部或者部分，比如“-RuntimeException”、“-tion”等等。

以下是两个示例：

|  |
| --- |
| <property name="\*Service">  PROPAGATION\_REQUIRED，ISOLATION\_READ\_COMMITTED，TIMEOUT\_20，  +AbcException，+DefException，-HijException  </property> |

以上表达式表示，针对所有方法名以 Service 结尾的方法，使用 PROPAGATION\_REQUIRED 事务传播行为，事务的隔离级别是 ISOLATION\_READ\_COMMITTED，超时时间为20秒，当事务抛出 AbcException 或者 DefException 类型的异常，则仍然提交，当抛出 HijException 类型的异常时必须回滚事务。这里没有指定"readOnly"，表示事务不是只读的。

|  |
| --- |
| <property name="test">PROPAGATION\_REQUIRED，readOnly</property> |

以上表达式表示，针对所有方法名为 test 的方法，使用 PROPAGATION\_REQUIRED 事务传播行为，并且该事务是只读的。除此之外，其他的属性均使用默认值。比如，隔离级别和超时时间使用底层事务性资源的默认值，并且当发生未检查异常，则回滚事务，发生已检查异常则仍提交事务。

配置好了 TransactionInterceptor，我们还需要配置一个 ProxyFactoryBean 来组装 target 和advice。这也是典型的 Spring AOP 的做法。或者是声明一个BeanNameAutoProxyCreator通过 生成的代理类就是织入了事务管理逻辑后的目标类。至此，声明式事务管理就算是实现了。我们没有对业务代码进行任何操作，所有设置均在配置文件中完成，这就是声明式事务的最大优点。

基于 **TransactionProxy... 的声明式事务管理**

前面的声明式事务虽然好，但是却存在一个非常恼人的问题：配置文件太多。我们必须针对每一个目标对象配置一个 ProxyFactoryBean；另外，虽然可以通过父子 Bean 的方式来复用 TransactionInterceptor 的配置，但是实际的复用几率也不高；这样，加上目标对象本身，每一个业务类可能需要对应三个 <bean/> 配置，随着业务类的增多，配置文件将会变得越来越庞大，管理配置文件又成了问题。

为了缓解这个问题，Spring 为我们提供了 TransactionProxyFactoryBean，用于将TransactionInterceptor 和 ProxyFactoryBean 的配置合二为一。如清单9所示：

清单9. 基于 TransactionProxyFactoryBean 的事务管理示例配置文件

|  |
| --- |
| <beans......>  ......  <bean id="bankServiceTarget"  class="footmark.spring.core.tx.declare.classic.BankServiceImpl">  <property name="bankDao" ref="bankDao"/>  </bean>  <bean id="bankService"  class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">  <property name="target" ref="bankServiceTarget"/>  <property name="transactionManager" ref="transactionManager"/>  <property name="transactionAttributes">  <props>  <prop key="transfer">PROPAGATION\_REQUIRED</prop>  </props>  </property>  </bean>  ...//或者是用transacitonProxyFactoyBean来做基类      <bean id="transactionBase"               class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"              lazy-init="true" abstract="true">           <!-- 配置事务管理器 -->           <property name="transactionManager" ref="transactionManager" />           <!-- 配置事务属性 -->           <property name="transactionAttributes">               <props>                   <prop key="\*">PROPAGATION\_REQUIRED</prop>               </props>           </property>       </bean>             <!-- 配置DAO -->     <bean id="userDaoTarget" class="com.bluesky.spring.dao.UserDaoImpl">         <property name="sessionFactory" ref="sessionFactory" />     </bean>          <bean id="userDao" parent="transactionBase" >           <property name="target" ref="userDaoTarget" />        </bean> </beans>...  </beans> |

如此一来，配置文件与先前相比简化了很多。我们把这种配置方式称为 Spring 经典的声明式事务管理。相信在早期使用 Spring 的开发人员对这种配置声明式事务的方式一定非常熟悉。

但是，显式为每一个业务类配置一个 TransactionProxyFactoryBean 的做法将使得代码显得过于刻板，为此我们可以使用自动创建代理的方式来将其简化，使用自动创建代理是纯 AOP 知识，请读者参考相关文档，不在此赘述。

基于 **<tx> 命名空间的声明式事务管理**

前面两种声明式事务配置方式奠定了 Spring 声明式事务管理的基石。在此基础上，Spring 2.x 引入了 <tx> 命名空间，结合使用 <aop> 命名空间，带给开发人员配置声明式事务的全新体验，配置变得更加简单和灵活。另外，得益于 <aop> 命名空间的切点表达式支持，声明式事务也变得更加强大。

如清单10所示：

清单10. 基于 <tx> 的事务管理示例配置文件

|  |
| --- |
| <beans......>  ......  <bean id="bankService"  class="footmark.spring.core.tx.declare.namespace.BankServiceImpl">  <property name="bankDao" ref="bankDao"/>  </bean>  <tx:advice id="bankAdvice" transaction-manager="transactionManager">  <tx:attributes>  <tx:method name="transfer" propagation="REQUIRED"/>  </tx:attributes>  </tx:advice>    <aop:config>  <aop:pointcut id="bankPointcut" expression="execution(\* \*.transfer(..))"/>  <aop:advisor advice-ref="bankAdvice" pointcut-ref="bankPointcut"/>  </aop:config>  ......  </beans> |

如果默认的事务属性就能满足要求，那么代码简化为如清单 11 所示：

清单 11. 简化后的基于 <tx> 的事务管理示例配置文件

|  |
| --- |
| <beans......>  ......  <bean id="bankService"  class="footmark.spring.core.tx.declare.namespace.BankServiceImpl">  <property name="bankDao" ref="bankDao"/>  </bean>  <tx:advice id="bankAdvice" transaction-manager="transactionManager">  <aop:config>  <aop:pointcut id="bankPointcut" expression="execution(\*\*.transfer(..))"/>  <aop:advisor advice-ref="bankAdvice" pointcut-ref="bankPointcut"/>  </aop:config>  ......  </beans> |

由于使用了切点表达式，我们就不需要针对每一个业务类创建一个代理对象了。另外，如果配置的事务管理器 Bean 的名字取值为“transactionManager”，则我们可以省略 <tx:advice> 的 transaction-manager 属性，因为该属性的默认值即为“transactionManager”。

基于 **@Transactional 的声明式事务管理**

除了基于命名空间的事务配置方式，Spring 2.x 还引入了基于 Annotation 的方式，具体主要涉及@Transactional 标注。@Transactional 可以作用于接口、接口方法、类以及类方法上。当作用于类上时，该类的所有 public 方法将都具有该类型的事务属性，同时，我们也可以在方法级别使用该标注来覆盖类级别的定义。如清单12所示：

清单12. 基于 @Transactional 的事务管理示例配置文件

|  |
| --- |
| @Transactional(propagation = Propagation.REQUIRED)  public boolean transfer(Long fromId， Long toId， double amount) {  return bankDao.transfer(fromId， toId， amount);  } |

Spring 使用 BeanPostProcessor 来处理 Bean 中的标注，因此我们需要在配置文件中作如下声明来激活该后处理 Bean，如清单13所示：

清单13. 启用后处理Bean的配置

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

与前面相似，transaction-manager 属性的默认值是 transactionManager，如果事务管理器 Bean 的名字即为该值，则可以省略该属性。

虽然 @Transactional 注解可以作用于接口、接口方法、类以及类方法上，但是 Spring 小组建议不要在接口或者接口方法上使用该注解，因为这只有在使用基于接口的代理时它才会生效。另外， @Transactional 注解应该只被应用到 public 方法上，这是由 Spring AOP 的本质决定的。如果你在 protected、private 或者默认可见性的方法上使用 @Transactional 注解，这将被忽略，也不会抛出任何异常。

基于 <tx> 命名空间和基于 @Transactional 的事务声明方式各有优缺点。基于 <tx> 的方式，其优点是与切点表达式结合，功能强大。利用切点表达式，一个配置可以匹配多个方法，而基于 @Transactional 的方式必须在每一个需要使用事务的方法或者类上用 @Transactional 标注，尽管可能大多数事务的规则是一致的，但是对 @Transactional 而言，也无法重用，必须逐个指定。另一方面，基于 @Transactional 的方式使用起来非常简单明了，没有学习成本。开发人员可以根据需要，任选其中一种使用，甚至也可以根据需要混合使用这两种方式。

如果不是对遗留代码进行维护，则不建议再使用基于 TransactionInterceptor 以及基于TransactionProxyFactoryBean 的声明式事务管理方式，但是，学习这两种方式非常有利于对底层实现的理解。

虽然上面共列举了四种声明式事务管理方式，但是这样的划分只是为了便于理解，其实后台的实现方式是一样的，只是用户使用的方式不同而已。

**结束语**

本教程的知识点大致总结如下：

·         基于 TransactionDefinition、PlatformTransactionManager、TransactionStatus 编程式事务管理是 Spring 提供的最原始的方式，通常我们不会这么写，但是了解这种方式对理解 Spring 事务管理的本质有很大作用。

·         基于 TransactionTemplate 的编程式事务管理是对上一种方式的封装，使得编码更简单、清晰。

·         基于 TransactionInterceptor 的声明式事务是 Spring 声明式事务的基础，通常也不建议使用这种方式，但是与前面一样，了解这种方式对理解 Spring 声明式事务有很大作用。

·         基于 TransactionProxyFactoryBean 的声明式事务是以上方式的改进版本，简化了配置文件的书写，这是 Spring 早期推荐的声明式事务管理方式，但是在 Spring 2.0 中已经不推荐了。

·         基于 <tx> 和 <aop> 命名空间的声明式事务管理是目前推荐的方式，其最大特点是与 Spring AOP 结合紧密，可以充分利用切点表达式的强大支持，使得管理事务更加灵活。

·         基于 @Transactional 的方式将声明式事务管理简化到了极致。开发人员只需在配置文件中加上一行启用相关后处理 Bean 的配置，然后在需要实施事务管理的方法或者类上使用 @Transactional 指定事务规则即可实现事务管理，而且功能也不比其他方式逊色