**Spring Framework**

**事务管理**

### Spring事务管理优点

#### 全局事务和局部事务

    对于事务管理，J2EE开发者有两种选择：全局事务和局部事务，两种选择都有其局限性。

##### 全局事务

全局事务确保你可以在多个事务源之间工作，典型的如关系型数据库和消息队列。全局事务由应用服务器通过JTA进行管理，JTA是一个笨重的API(部分原因是因为其异常模式)，并且，一个JTA的UserTransaction通常需要通过JNDI获取，这就意味着使用JTA就必须使用JNDI。很明显使用全局事务将潜在的局限应用代码的复用，并且JTA是通常只能在应用程序服务器环境。

以前，使用全局事务比较流行的方法是采用EJB CMT，CMT是声明式事务管理的一种形式(区别于编程式事务管理)。尽管使用EJB本身就需要使用JNDI，EJB CMT不需要事务相关的JNDI lookups。EJB CMT不需要编写大量的Java代码来控制事务。使用CMT最大的不足之处是它被束缚在JTA和应用服务器上。CMT只有当在EJB中实现业务逻辑时才可以发挥作用，或者至少处于事务性EJB Façade中。由于EJB存在诸多的不足并且目前存在其他可选的声明式事务管理的解决方案，所以不太建议使用EJB方式。

    1.1、全局事务的优点：

    （1）全局事务支持多个事务性资源间的相互工作（如：关系型数据库和消息队列）。

    1.2、全局事务的缺点：

    （1）采用全局事务需要使用JTA，而JTA是一个笨重的API。

    （2）通常情况下，JTA UserTransaction需要从JNDI获取。这意味着，如果我们使用JTA，就需要同时使用JTA和JNDI。

    （3）通常JTA只能在应用服务器环境下使用，因此使用JTA会限制代码的复用性。

##### 局部事务

    局部事务是资源特有的（resource-specific），最常见的例子是与JDBC连接相关联的事务。使用局部事务，应用服务器不参与事务管理并且不能保证访问多个资源的正确性。值得注意的是：大多数应用程序使用单个的事务性资源。

    2.1、局部事务的优点：

    （1）易用

    2.2、局部事务的缺点：

    （1）局部事务不支持多个事务性资源间的相互工作，比如：使用JDBC连接来管理事务的代码不能在全局JTA事务上运行。

    （2）局部事务趋向于编程模型，编程模型具有侵入性。

    Spring可以使应用程序开发者在任何环境下使用统一的编程模型。应用程序开发者只需要编写一次代码，就可以使用不同环境下的不同事务管理策略。Spring既支持声明式事务管理，也支持编程式事务管理。

    我们推荐使用声明式事务管理。采用编程式事务管理，开发者需要使用Spring提供的事务抽象（它可以在任何基础的事务架构上运行）。采用声明式事务管理，开发者仅需要编写少量甚至不需要编写与事务管理相关的代码，因此它不依赖于Spring的事务API。

#### Spring Framework一致的编程模型

Spring解决了局部事务和全局事务的确定，确保应用开发者在任何环境下使用一致的编程模型，一旦编写好代码，可以让你在不同的环境、不同的事务策略中受益。Spring Framework提供了声明式和编程式事务管理。需要用户喜欢使用声明式事务管理，这也是我们所推荐的。

使用编程式事务管理，开发者需要使用可以运行在任何事务基础下的Spring Framework Transaction abstraction；使用比较受欢迎的声明式事务管理，开发者只需要写少量代码或者不写代码来管理事务管理，并且因此不需要依赖Spring Framework事务API，或者其他事务API。

##### 你需要一个应用服务器事务管理吗？

当企业级Java应用需要一个应用服务时，Spring Framework事务管理改变了传统传统的规则。

详细地说，你不需要通过ejb应用服务器只是为了声明性事务。事实上，即使您的应用程序服务器强大的JTA功能，你可能会决定,Spring框架的声明性事务提供更多的力量和更有效率比EJB CMT的编程模型。

通常只需要一个应用程序服务器的JTA能力如果您的应用程序需要处理跨多个资源的事务，这个不是许多应用所必须的，许多高端的应用使用了单一的、高扩展性的数据库(如Oracle RAC)替代。单独的事务管理器如Atomikos Transactions和JOTM也是其他的选择。当然，你也可能需要例如Java Message Service(JMS)和J2EE Connector Architecture(JCA)等应用服务。

The Spring Framework gives you the choice of when to scale your application to a fully loaded application server. Gone are the days when the only alternative to using EJB CMT or JTA was to write code with local transactions such as those on JDBC connections, and face a hefty rework if you need that code to run within global, container-managed transactions. With the Spring Framework, only some of the bean definitions in your configuration file, rather than your code, need to change.

### 理解Spring Framework事务抽象

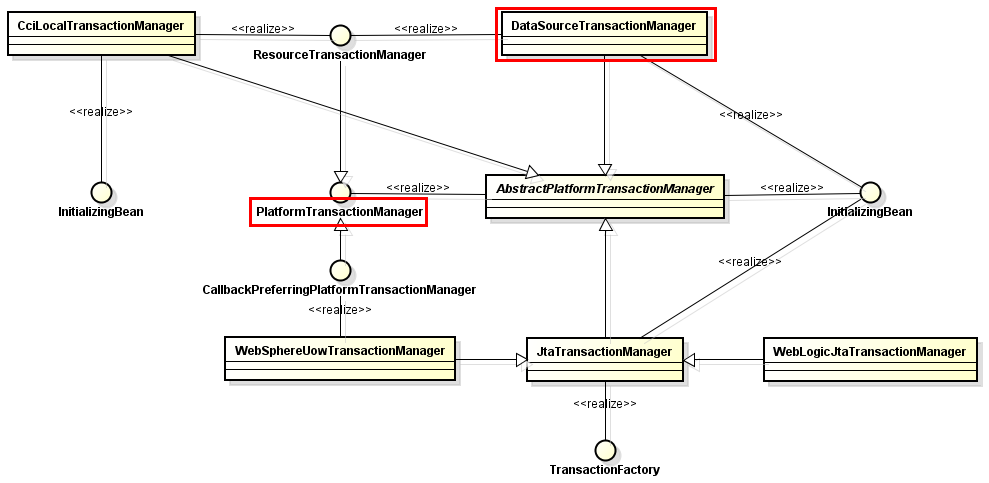
#### PlatformTransactionManager

Spring事务的关键抽象的概念是一种事务策略,一个事务策略通过org.springframework.transaction.PlatformTransactionManager接口定义。定义了如下接口：

TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;

void commit(TransactionStatus status) throws TransactionException;

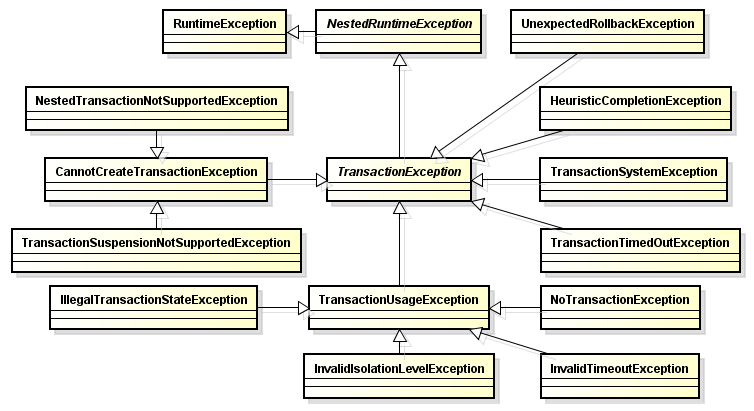
void rollback(TransactionStatus status) throws TransactionException;



PlatformTransactionManager类关系图

PlatformTransactionManager主要是一个服务提供借口(SPI)，虽然它可以通过你的应用代码进行编程。因为PlatformTransactionManager是一个接口，所以有必要它非常容易被模仿和剔除，它不依赖于JNDI查询策略。PlatformTransactionManager的是现在像其它bean一项在Spring IoC容器中被定义。即使你直接使用JTA，事务代码也是非常容易被测试的。

PlatformTransactionManager的任何方法可以抛出TransactionException类型的异常，此异常是非检查异常(unchecked Exception，继承了java.lang.RuntimeException)。事务底层基础故障是非常致命的，极少数情况下应用程序代码可以从事务底层基础故障中恢复过来。应用程序开发人员仍然可以选择捕获和处理TransactionException，突出的一点是,开发人员不被迫这样做



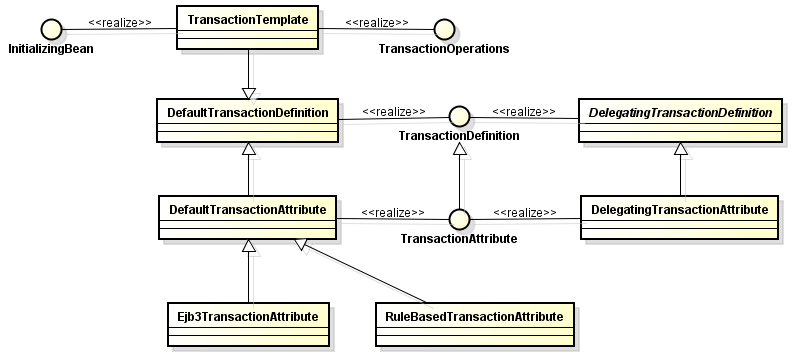
TransactionException类关系图

PlatformTransactionManager中的getTransaction(..)方法返回一个TransactionStatus对象，依赖于TransactionDefinition参数。TransactionStatus可能代表了一个新的Trasaction或一个已经存在与当前线程调用栈的Transaction。

#### TransactionDefinition

TransactionDefinition接口声明：

1. Isolation：决定了当前事务与其他事务的隔离级别。
2. Propagation：所有的代码在事务中被限定在一个事务范围内(transaction scope)，然而，您可以选择指定行为的执行事务方法当事务上下文已经存在.Spring提供了类似于EJB CMT的事务传播方式(tracsaction paopagation)。
3. Timeout：设置事务超时时间，超时之后由Spring底层事务框架自动回滚事务。
4. Read-only status：一个read-only的事务可以读取数据，但是不能修改数据。在某些情况下read-only的事务是一个非常有用的性能优化。例如在Hibernate中。

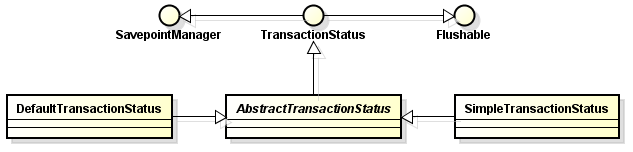


TransactionDefinition类关系图

#### TransactionStatus

TransactionStauts接口提供了一个非常简便的方式去控制事务的执行和查询事务的状态,这跟其他事务APIs是共通的。

public interface TransactionStatus extends SavepointManager {  
boolean isNewTransaction();  
boolean hasSavepoint();  
void setRollbackOnly();  
boolean isRollbackOnly();  
void flush();  
boolean isCompleted();

} 

TransactionStauts类关系图

#### PlatformTransactionManager相关配置

在Spring中，无论你选择的是声明式事务还是编程式事务，定义前挡的PlatformTransactionManager实现绝对是必要的，典型的定义PlatformTransactionManager实现是通过依赖注入方式(DI)。

定义JDBC数据源：

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroymethod="close">

<property name="driverClassName" value="${jdbc.driverClassName}" />

<property name="url" value="${jdbc.url}" />

<property name="username" value="${jdbc.username}" />

<property name="password" value="${jdbc.password}" />

</bean>

关联的PlatformTransactionManager的bean定义需要引用DataSource的定义，如下：

<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"/>

</bean>

如果在Java EE容器中使用JTA，然后你需要一个容器DataSource，通过JNDI获取它，结合Spring的JtaTransactionManager.如下：

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:jee="http://www.springframework.org/schema/jee"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/jee

http://www.springframework.org/schema/jee/spring-jee.xsd">

<jee:jndi-lookup id="dataSource" jndi-name="jdbc/jpetstore"/>

<bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManager"/>

<!-- other <bean/> definitions here -->

</beans>

JtaTransactionManager无需知道DataSource，或者指定资源，因为它采用了容器的全局事务管理框架。

如果DataSource使用了非JTA(non-JTA)事务管理，通过JNDI搜索和Java EE容器管理，那么它将是非事务性(non-Trasaction)，因为Spring将管理它而不是Java EE容器。

##### 提醒

如果你是用JTA，那么不管你使用什么数据访问技术，你的事务管理定义看上去都是相同的。可以你的数据访问技术是JDBC、Hibernate JPA或者其他被支持的技术。这都是因为JTA事务是全局事务，它能支持任何的事务资源。

### 源码分析 声明式事务管理 for 局部事务

#### Spring上下文配置

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:p=*"http://www.springframework.org/schema/p"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*

*http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd*

*http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd*

*http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.0.xsd"*>

<!-- %%%%Spring JDBC%%%% start -->

<!-- %数据库连接配置% start -->

<context:property-placeholder location=*"classpath:db.properties"* />

<!-- %数据库连接配置% end -->

<!-- %数据源/数据库连接池c3p0配置% start -->

<bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"* destroy-method=*"close"*>

<property name=*"driverClass"* value=*"${db.driverClassName}"*></property>

<property name=*"jdbcUrl"* value=*"${db.url}"*></property>

<property name=*"user"* value=*"${db.username}"*></property>

<property name=*"password"* value=*"${db.password}"*></property>

</bean>

<!-- %数据源/数据库连接池c3p0配置% end -->

<!-- %配置事务管理器% id如果省略，则默认为"transactionManager", start -->

<bean id=*"transactionManager"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

<property name=*"dataSource"* ref=*"dataSource"* />

</bean>

<!-- %配置事务管理器% end -->

<!-- %基于AOP的声明式事务配置% start -->

<!-- %配置事务通知属性% start -->

<tx:advice id=*"txAdvice"* transaction-manager=*"transactionManager"*>

<tx:attributes>

<!-- %CRUD操作事务属性% start -->

<tx:method name=*"create\*"* propagation=*"REQUIRED"*/>

<tx:method name=*"read\*"* propagation=*"REQUIRED"* read-only=*"true"* />

<tx:method name=*"update\*"* propagation=*"REQUIRED"* />

<tx:method name=*"delete\*"* propagation=*"REQUIRED"* isolation=*"DEFAULT"* read-only=*"false"* />

<!-- %CRUD操作事务属性% end -->

</tx:attributes>

</tx:advice>

<!-- %配置事务通知属性% end -->

<aop:config>

<aop:pointcut id=*"serviceOperation"* expression=*"execution(\* com..\*.service..\*ServiceImpl.\*(..))"* />

<aop:advisor advice-ref=*"txAdvice"* pointcut-ref=*"serviceOperation"* />

</aop:config>

<!-- %基于AOP的声明式事务配置% end -->

<!-- %%%%Spring JDBC%%%% end -->

<!-- %%%%自动扫描注解组件%%%%Start -->

<context:component-scan base-package=*"com"* />

<!-- %%%%自动扫描注解组件%%%%End -->

</beans>

#### <tx:advice/>解析

在Spring IoC容器初始化的时候，对Schema有对应的解析类对其进行解析，对<tx:\*\*/>的解析类在”spring-tx/src/main/resources/META-INF/spring.handlers”文件中进行了声明，解析类为TxNamespaceHandler。

从TxNamespaceHandler的init()方法可以知道，<tx:advice/>的解析类为TxAdviceBeanDefinitionParser。

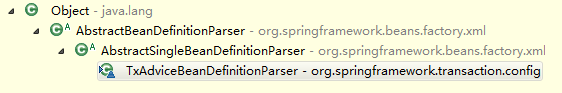
**public** **void** init() {

registerBeanDefinitionParser("advice", **new** TxAdviceBeanDefinitionParser());

registerBeanDefinitionParser("annotation-driven", **new** AnnotationDrivenBeanDefinitionParser());

registerBeanDefinitionParser("jta-transaction-manager",**new** JtaTransactionManagerBeanDefinitionParser());

}



TxAdviceBeanDefinitionParser类继承关系

从上图我们可以知道TxAdviceBeanDefinitionParser实现了AbstractSingleBeanDefinitionParser类，AbstractSingleBeanDefinitionParser又实现了AbstractBeanDefinitionParser类，解析的方法从AbstractBeanDefinitionParser的parse(Element,ParserContext)方法开始，parse(Element,ParserContext)中又会转调子类[AbstractSingleBeanDefinitionParser]的parseInternal(Element,ParserContext)完成解析，如下：

**protected** **final** AbstractBeanDefinition parseInternal(Element element, ParserContext parserContext) {

BeanDefinitionBuilder builder = BeanDefinitionBuilder.*genericBeanDefinition*();

String parentName = getParentName(element);

**if** (parentName != **null**) {

builder.getRawBeanDefinition().setParentName(parentName);

}

Class<?> beanClass = getBeanClass(element);

**if** (beanClass != **null**) {

builder.getRawBeanDefinition().setBeanClass(beanClass);

} **else** {

String beanClassName = getBeanClassName(element);

**if** (beanClassName != **null**) {

builder.getRawBeanDefinition().setBeanClassName(beanClassName);

}

}

builder.getRawBeanDefinition().setSource(parserContext.extractSource(element));

**if** (parserContext.isNested()) {

// Inner bean definition must receive same scope as containing bean.

builder.setScope(parserContext.getContainingBeanDefinition().getScope());

}

**if** (parserContext.isDefaultLazyInit()) {

// Default-lazy-init applies to custom bean definitions as well.

builder.setLazyInit(**true**);

}

doParse(element, parserContext, builder);

**return** builder.getBeanDefinition();

}

从parseInternal中的方法实现来看，parseInternal方法只做一些公共的实现，最后<tx:advice/>元素的解析还是由TxAdviceBeanDefinitionParser的doParse(Element,ParserContext,BeanDefinitionBuilder)方法实现。

**protected** **void** doParse(Element element, ParserContext parserContext, BeanDefinitionBuilder builder) {

builder.addPropertyReference("transactionManager",TxNamespaceHandler.*getTransactionManagerName*(element));

List<Element> txAttributes = DomUtils.*getChildElementsByTagName*(element, ***ATTRIBUTES\_ELEMENT***);

**if** (txAttributes.size() > 1) {

parserContext.getReaderContext().error("Element <attributes> is allowed at most once inside element <advice>", element);

} **else** **if** (txAttributes.size() == 1) {

// Using attributes source.

Element attributeSourceElement = txAttributes.get(0);

RootBeanDefinition attributeSourceDefinition = parseAttributeSource(attributeSourceElement, parserContext);

builder.addPropertyValue("transactionAttributeSource", attributeSourceDefinition);

} **else** {

// Assume annotations source.

builder.addPropertyValue("transactionAttributeSource",**new** RootBeanDefinition("org.springframework.transaction.annotation.AnnotationTransactionAttributeSource"));

}

}

1、Spring为<tx:advice/>元素在Spring IoC容器中注册TransactionInterceptor对应的bean定义。

2、获取<tx:advice/>的”transaction-manager”属性对应的值，此值对应的是事务管理器[TransactionManager]的bean定义id，如上面的”transactionManager”。如果”transaction-manager”没有值，则默认去id为” transactionManager”的bean定义。

3、获取<tx:advice/>下的子元素<tx:attributes/>

a、如果<tx:advice/>下的<tx:attributes/>数量超过1，则抛错。

b、如果<tx:advice/>下的<tx:attributes/>数量为0，则为TransactionInterceptor的transactionAttributeSource字段添加属性引用AnnotationTransactionAttributeSource。

builder.addPropertyValue("transactionAttributeSource",**new** RootBeanDefinition("org.springframework.transaction.annotation.AnnotationTransactionAttributeSource"));

c、如果<tx:advice/>下的<tx:attributes/>属性为1，则进行以下逻辑：

Element attributeSourceElement = txAttributes.get(0);

RootBeanDefinition attributeSourceDefinition = parseAttributeSource(attributeSourceElement, parserContext);// NameMatchTransactionAttributeSource

builder.addPropertyValue("transactionAttributeSource", attributeSourceDefinition);

// 详细见[parseAttributeSource(Element attrEle, ParserContext parserContext)](#_parseAttributeSource(Element_attrEl)

##### parseAttributeSource(Element attrEle, ParserContext parserContext)

**private** RootBeanDefinition parseAttributeSource(Element attrEle, ParserContext parserContext) {

List<Element> methods = DomUtils.*getChildElementsByTagName*(attrEle, ***METHOD\_ELEMENT***);

ManagedMap<TypedStringValue, RuleBasedTransactionAttribute> transactionAttributeMap = **new** ManagedMap<TypedStringValue, RuleBasedTransactionAttribute>(methods.size());

transactionAttributeMap.setSource(parserContext.extractSource(attrEle));

**for** (Element methodEle : methods) {

String name = methodEle.getAttribute(***METHOD\_NAME\_ATTRIBUTE***);

TypedStringValue nameHolder = **new** TypedStringValue(name);

nameHolder.setSource(parserContext.extractSource(methodEle));

RuleBasedTransactionAttribute attribute = **new** RuleBasedTransactionAttribute();

String propagation = methodEle.getAttribute(***PROPAGATION\_ATTRIBUTE***);

String isolation = methodEle.getAttribute(***ISOLATION\_ATTRIBUTE***);

String timeout = methodEle.getAttribute(***TIMEOUT\_ATTRIBUTE***);

String readOnly = methodEle.getAttribute(***READ\_ONLY\_ATTRIBUTE***);

**if** (StringUtils.*hasText*(propagation)) {

attribute.setPropagationBehaviorName(RuleBasedTransactionAttribute.***PREFIX\_PROPAGATION*** + propagation);

}

**if** (StringUtils.*hasText*(isolation)) {

attribute.setIsolationLevelName(RuleBasedTransactionAttribute.***PREFIX\_ISOLATION*** + isolation);

}

**if** (StringUtils.*hasText*(timeout)) {

**try** {

attribute.setTimeout(Integer.*parseInt*(timeout));

} **catch** (NumberFormatException ex) {

parserContext.getReaderContext().error("Timeout must be an integer value: [" + timeout + "]", methodEle);

}

}

**if** (StringUtils.*hasText*(readOnly)) {

attribute.setReadOnly(Boolean.*valueOf*(methodEle.getAttribute(***READ\_ONLY\_ATTRIBUTE***)));

}

List<RollbackRuleAttribute> rollbackRules = **new** LinkedList<RollbackRuleAttribute>();

**if** (methodEle.hasAttribute(***ROLLBACK\_FOR\_ATTRIBUTE***)) {

String rollbackForValue = methodEle.getAttribute(***ROLLBACK\_FOR\_ATTRIBUTE***);

addRollbackRuleAttributesTo(rollbackRules,rollbackForValue);

}

**if** (methodEle.hasAttribute(***NO\_ROLLBACK\_FOR\_ATTRIBUTE***)) {

String noRollbackForValue = methodEle.getAttribute(***NO\_ROLLBACK\_FOR\_ATTRIBUTE***);

addNoRollbackRuleAttributesTo(rollbackRules,noRollbackForValue);

}

attribute.setRollbackRules(rollbackRules);

transactionAttributeMap.put(nameHolder, attribute);

}

RootBeanDefinition attributeSourceDefinition = **new** RootBeanDefinition(NameMatchTransactionAttributeSource.**class**);

attributeSourceDefinition.setSource(parserContext.extractSource(attrEle));

attributeSourceDefinition.getPropertyValues().add("nameMap", transactionAttributeMap);

**return** attributeSourceDefinition;

}

1、迭代<tx:attributes/>下<tx:method/>子元素。

2、根据每一个<tx:method/>元素的name属性值构建TypedStringValue，其他属性则将其构建到RuleBasedTransactionAttribute中的不同字段中。

a、将propagation属性的值设置到RuleBasedTransactionAttribute的propagationBehavior字段中。

int PROPAGATION\_REQUIRED = 0;

int PROPAGATION\_SUPPORTS = 1;

int PROPAGATION\_MANDATORY = 2;

int PROPAGATION\_REQUIRES\_NEW = 3;

int PROPAGATION\_NOT\_SUPPORTED = 4;

int PROPAGATION\_NEVER = 5;

int PROPAGATION\_NESTED = 6;

b、将isolation属性的值设置到RuleBasedTransactionAttribute的isolationLevel字段中。

int ISOLATION\_DEFAULT = -1;

int ISOLATION\_READ\_UNCOMMITTED = Connection.TRANSACTION\_READ\_UNCOMMITTED; // 1

int ISOLATION\_READ\_COMMITTED = Connection.TRANSACTION\_READ\_COMMITTED; // 2

int ISOLATION\_REPEATABLE\_READ = Connection.TRANSACTION\_REPEATABLE\_READ; // 4

int ISOLATION\_SERIALIZABLE = Connection.TRANSACTION\_SERIALIZABLE; // 8

c、将timeout属性的值设置到RuleBasedTransactionAttribute的timeout字段中.

d、将read-only属性的值设置到RuleBasedTransactionAttribute的readOnly字段中。

e、获取rollback-for属性的值，将值按”,”[逗号]切分，然后根据切分后的值构建RollbackRuleAttribute对象，并将其加入RuleBasedTransactionAttribute的rollbackRules列表。

f、获取no-rollback-for属性的值，将值按”,”[逗号]切分，然后根据切分后的值构建NoRollbackRuleAttribute对象，并将其加入RuleBasedTransactionAttribute的rollbackRules列表。

3、根据步骤2中的结果，将其设值到transactionAttributeMap中，transactionAttributeMap的KEY为步骤2中的TypedStringValue值，value为RuleBasedTransactionAttribute。

4、创建NameMatchTransactionAttributeSource对应的bean定义，并将transactionAttributeMap设置到NameMatchTransactionAttributeSource的nameMap属性中。

#### 基于AOP的事务控制

我们在AOP 源码分析的时候已经知道，Spring AOP可以基于JDK动态代理，也可以基于CGLIB代理。这里我们以基于JDK动态代理的AOP实现为例。

在JdkDynamicAopProxy中会为每个符合要求的目标对象创建代理对象，JdkDynamicAopProxy实现了InvocationHandler接口，在调用动态代理对象接口时，其实会回调InvocationHandler.invoke(…)方法，在invoke(…)方法中，会根据AOP的相关配置找到对应的事务拦截器，也就是[<tx:advice/>解析](#_<tx:advice/>解析)中注册的TransactionInterceptor对象，调用其invoke(MethodInvocation)方法完成事务控制，接下来我们就从这个方法开始我们Spring 基于AOP的事务控制源码分析。

**public** Object invoke(**final** MethodInvocation invocation) **throws** Throwable {

Class<?> targetClass = (invocation.getThis() != **null** ? AopUtils.*getTargetClass*(invocation.getThis()) : **null**);

**return** invokeWithinTransaction(invocation.getMethod(), targetClass, **new** InvocationCallback() {

@Override

**public** Object proceedWithInvocation() **throws** Throwable {

**return** invocation.proceed();

}

});

}

获取目标对象的class，调用TransactionAspectSupport的invokeWithinTransaction(…)方法。

**protected** Object invokeWithinTransaction(Method method, Class<?> targetClass, **final** InvocationCallback invocation) **throws** Throwable {

// If the transaction attribute is null, the method is non-transactional.

**final** TransactionAttribute txAttr = getTransactionAttributeSource().getTransactionAttribute(method, targetClass);

**final** PlatformTransactionManager tm = determineTransactionManager(txAttr);

**final** String joinpointIdentification = methodIdentification(method, targetClass);

**if** (txAttr == **null** || !(tm **instanceof** CallbackPreferringPlatformTransactionManager)) {

// Standard transaction demarcation with getTransaction and commit/rollback calls.

TransactionInfo txInfo = createTransactionIfNecessary(tm, txAttr, joinpointIdentification);

Object retVal = **null**;

**try** {

retVal = invocation.proceedWithInvocation();

} **catch** (Throwable ex) {

completeTransactionAfterThrowing(txInfo, ex);

**throw** ex;

} **finally** {

cleanupTransactionInfo(txInfo);

}

commitTransactionAfterReturning(txInfo);

**return** retVal;

} **else** {

**try** {

Object result = ((CallbackPreferringPlatformTransactionManager) tm).execute(txAttr, **new** TransactionCallback<Object>() {

@Override

**public** Object doInTransaction(TransactionStatus status) {

TransactionInfo txInfo = prepareTransactionInfo(tm, txAttr, joinpointIdentification, status);

**try** {

**return** invocation.proceedWithInvocation();

} **catch** (Throwable ex) {

**if** (txAttr.rollbackOn(ex)) {

**if** (ex **instanceof** RuntimeException) {

**throw** (RuntimeException) ex;

} **else** {

**throw** **new** ThrowableHolderException(ex);

}

} **else** {

**return** **new** ThrowableHolder(ex);

}

} **finally** {

cleanupTransactionInfo(txInfo);

}

} });

**if** (result **instanceof** ThrowableHolder) {

**throw** ((ThrowableHolder) result).getThrowable();

} **else** {

**return** result;

}

} **catch** (ThrowableHolderException ex) {

**throw** ex.getCause();

}

}

}

1、调用TransactionAttributeSource的getTransactionAttribute(Method,Class)获取事务属性TransactionAttribute。从前面的[<tx:advice/>解析](#_<tx:advice/>解析)章节我们可以知道，Spring会根据<tx:advice/>中的<tx:attributes/>子元素注册不同的TransactionAttributeSource，可能为AnnotationTransactionAttributeSource和NameMatchTransactionAttributeSource。这里我们NameMatchTransactionAttributeSource来解析[getTransactionAttribute(Method,Class)](#_getTransactionAttribute(Method,Clas)方法。

2、获取事务管理器PlatformTransactionManager。

3、调用以下逻辑获取连接点标识：

**protected** String methodIdentification(Method method, Class<?> targetClass) {

**return** (targetClass != **null** ? targetClass : method.getDeclaringClass()).getName() + "." + method.getName();

}

4、根据给定的事务属性创建事务(如果有必要)，详细见[createTransactionIfNecessary(PlatformTransactionManager, TransactionAttribute,String)](#_createTransactionIfNecessary(Platfo)

5、Spring的事务拦截器是一个AOP的环绕通知，所以需要调用InvocationCallback.proceedWithInvocation()执行下一个拦截器。

6、如果在执行其他拦截器和目标方法的期间发生了异常，需要调用[completeTransactionAfterThrowing(TransactionInfo, Throwable)](#_completeTransactionAfterThrowing(Tr)方法完成相关的事务处理。

7、调用cleanupTransactionInfo(TransactionInfo)方法重置ThreadLocal中的事务信息[TransactionInfo].

8、成功完成调用后执行以下逻辑：

**protected** **void** commitTransactionAfterReturning(TransactionInfo txInfo) {

**if** (txInfo != **null** && txInfo.hasTransaction()) {

txInfo.getTransactionManager().commit(txInfo.getTransactionStatus());

}

}

可以看出，上述方法其实本质还是调用事务管理器的commit方法。

文件位置：org/springframework/transaction/support/AbstractPlatformTransactionManager.java

**public** **final** **void** commit(TransactionStatus status) **throws** TransactionException {

**if** (status.isCompleted()) { //

**throw** **new** IllegalTransactionStateException("Transaction is already completed - do not call commit or rollback more than once per transaction");

}

DefaultTransactionStatus defStatus = (DefaultTransactionStatus) status;

**if** (defStatus.isLocalRollbackOnly()) {

processRollback(defStatus);

**return**;

}

**if** (!shouldCommitOnGlobalRollbackOnly() && defStatus.isGlobalRollbackOnly()) {

processRollback(defStatus);

**if** (status.isNewTransaction() || isFailEarlyOnGlobalRollbackOnly()) {

**throw** **new** UnexpectedRollbackException("Transaction rolled back because it has been marked as rollback-only");

}

**return**;

}

processCommit(defStatus);

}

##### getTransactionAttribute(Method,Class)

文件位置：org/springframework/transaction/interceptor/NameMatchTransactionAttributeSource.java

**public** TransactionAttribute getTransactionAttribute(Method method, Class<?> targetClass) {

**if** (!ClassUtils.*isUserLevelMethod*(method)) {

**return** **null**;

}

// Look for direct name match.

String methodName = method.getName();

TransactionAttribute attr = **this**.nameMap.get(methodName);

**if** (attr == **null**) {

// Look for most specific name match.

String bestNameMatch = **null**;

**for** (String mappedName : **this**.nameMap.keySet()) {

**if** (isMatch(methodName, mappedName) && (bestNameMatch == **null** || bestNameMatch.length() <= mappedName.length())) {

attr = **this**.nameMap.get(mappedName);

bestNameMatch = mappedName;

}

}

}

**return** attr;

}

1、首先根据方法名完全匹配<tx:method/>的name属性值，如果相等则返回匹配到的TransactionAttribute。如果没匹配到，则进行一下匹配。

2、迭代NameMatchTransactionAttributeSource中的nameMap，然后再进一步根据name匹配出最优的属性，最优的就是name值最长的[从方法上来看是酱紫的]。

##### createTransactionIfNecessary(PlatformTransactionManager, TransactionAttribute,String)

文件位置：org/springframework/transaction/interceptor/TransactionAspectSupport.java

根据给定的事务属性创建事务(如果有必要)

##### completeTransactionAfterThrowing(TransactionInfo, Throwable)

#### DataSource

Spring通过一个DataSource获取一个连接，DataSource是JDBC说明书的一部分，并且它通常还是一个连接工厂.它允许一个容器或一个框架通过应用代码去隐藏连接池和事务管理。

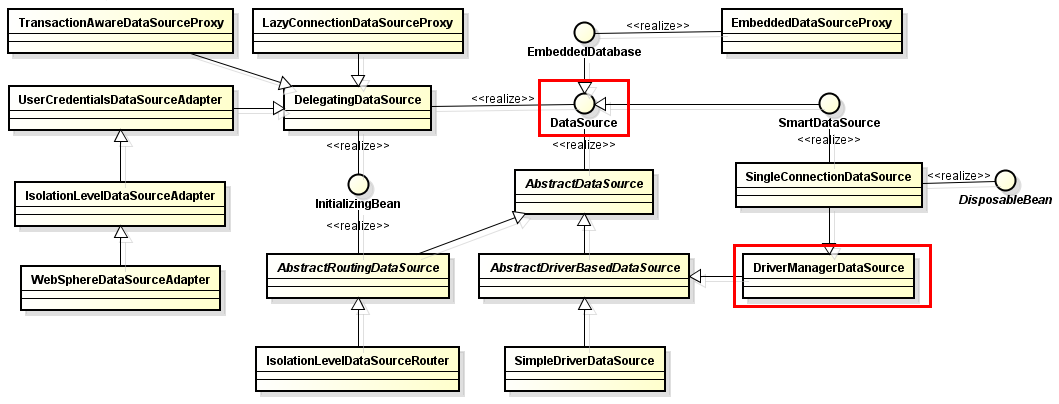
当你使用Spring JDBC层，你可以通过JNDI或配置的第三方提供商连接池实现来获取数据源。比较受欢迎的实现有Apache Jakarta Commons DBCP and C3P0.

##### DriverManagerDataSource

通常DriverManagerDataSource仅仅是被用来测试，因为它不提供连接池并且多个请求连接时表现不佳。

通过Java方式配置DriverManagerDataSource.

DriverManagerDataSource dataSource = **new** DriverManagerDataSource();  
dataSource.setDriverClassName(***"org.hsqldb.jdbcDriver"***);  
dataSource.setUrl(***"jdbc:hsqldb:hsql://localhost:"***);  
dataSource.setUsername(***"sa"***);  
dataSource.setPassword(***""***);



Spring DataSource类关系图

通过XML方式配置

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">  
<property name="driverClassName" value="${jdbc.driverClassName}"/>  
<property name="url" value="${jdbc.url}"/>  
<property name="username" value="${jdbc.username}"/>  
<property name="password" value="${jdbc.password}"/>

</bean>

<context:property-placeholder location="jdbc.properties"/>

DriverManagerDataSource继承了AbstractDriverBasedDataSource抽象类，通过getConnection(..)获取连接，逻辑如下：

代码位置：AbstractDriverBasedDataSource

@Override

public Connection getConnection() throws SQLException {

return getConnectionFromDriver(getUsername(), getPassword());

}

@Override

public Connection getConnection(String username, String password) throws SQLException {

return getConnectionFromDriver(username, password);

}

protected Connection getConnectionFromDriver(String username, String password) throws SQLException {

Properties mergedProps = new Properties();

Properties connProps = getConnectionProperties();

if (connProps != null) {

mergedProps.putAll(connProps);

}

if (username != null) {

mergedProps.setProperty("user", username);

}

if (password != null) {

mergedProps.setProperty("password", password);

}

return getConnectionFromDriver(mergedProps);

}

最后调用子类的getConnectionFromDriver(..)方法获取连接，在DriverManagerDataSource中只是最原始的调用DriverManager.getConnection()获取连接，并没有做连接池的控制。

##### BasicDataSource

通过XML配置：

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroymethod="close">

<property name="driverClassName" value="${jdbc.driverClassName}"/>

<property name="url" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

<context:property-placeholder location="jdbc.properties"/>

BasicDataSource是DBCP提供的一个DataSource实现，DBCP(DataBase connection pool),[数据库连接池](http://baike.baidu.com/view/84055.htm)。是 apache上的一个 java 连接池项目，也是 tomcat 使用的连接池组件。单独使用dbcp需要3个包：commons-dbcp.jar,commons-pool.jar,commons-collections.jar由于建立数据库连接是一个非常耗时[耗资源](http://baike.baidu.com/view/630544.htm)的行为，所以通过连接池预先同数据库建立一些连接，放在内存中，应用程序需要建立数据库连接时直接到连接池中申请一个就行，用完后再放回去。

DBCP官网地址：<http://commons.apache.org/proper/commons-dbcp/configuration.html>

##### ComboPooledDataSource

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource" destroymethod="close">

<property name="driverClass" value="${jdbc.driverClassName}"/>

<property name="jdbcUrl" value="${jdbc.url}"/>

<property name="user" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

<context:property-placeholder location="jdbc.properties"/>

C3P0是一个开源的JDBC连接池，它实现了数据源和JNDI绑定，支持JDBC3规范和JDBC2的标准扩展。

c3p0与dbcp区别

dbcp没有自动回收空闲连接的功能.

c3p0有自动回收空闲连接功能.

使用C3P0需要依赖mchange-commons-java-0.2.3.4.jar包。

C3P0 V0.9.5完全支持JDBC4.0

In particular, c3p0 provides several useful services:

A class whichs adapt traditional DriverManager-based JDBC drivers to the newer javax.sql.DataSource scheme for acquiring database Connections.

Transparent(透明的；显然的；坦率的；易懂的) pooling of Connection and PreparedStatements behind DataSources which can "wrap" around traditional drivers or arbitrary(任意的；武断的；专制的) unpooled DataSources.

The library tries hard to(努力) get the details right:

c3p0 DataSources are both Referenceable and Serializable, and are thus suitable(适当的) for binding to a wide-variety of JNDI-based naming services.

Statement and ResultSets are carefully cleaned up when pooled Connections and Statements are checked in, to prevent(预防，防止；阻止) resource-exhaustion(枯竭；耗尽；精疲力竭) when clients use the lazy but common resource-management strategy of only cleaning up their Connections. (Don't be naughty.[ 顽皮的，淘气的；不听话的；没规矩的；不适当的；下流的])

The library adopts the approach defined by the JDBC 2 and 3 specification (even where these conflict with the library author's preferences). DataSources are written in the JavaBean style, offering all the required and most of the optional properties (as well as some non-standard ones), and no-arg constructors. All JDBC-defined internal interfaces are implemented (ConnectionPoolDataSource, PooledConnection, ConnectionEvent-generating Connections, etc.) You can mix c3p0 classes with compliant(顺从的；服从的；应允的) third-party implementations (although not all c3p0 features will work with external implementations of ConnectionPoolDataSource).

c3p0-0.9.5-pre10 requires a level 1.6.x or above Java Runtime Environment.

c3p0 supportsnamed configurations so that you can configure multiple DataSources.

ComboPooledDataSource cpds = new ComboPooledDataSource("intergalactoApp");

C3P0官网地址：<http://www.mchange.com/projects/c3p0/>

#### PlatformTransactionManager

<!-- %配置事务管理器% id如果省略，则默认为"transactionManager", start -->

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource" />

</bean>

<!-- %配置事务管理器% end -->

PlatformTransactionManager相关介绍见[PlatformTransactionManager相关配置](#_PlatformTransactionManager相关配置)。PlatformTransactionManager是一个接口，其定义了getTransaction、commit和rollback三个方法，如下：

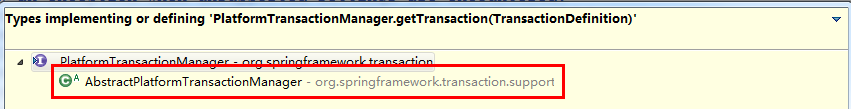
TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;

void commit(TransactionStatus status) throws TransactionException;

void rollback(TransactionStatus status) throws TransactionException;

现在，我们针对这三个方法的具体实现进行源码分析：

##### getTransaction



通过上图可以发现，getTransaction()方法在AbstractPlatformTransactionManager中得到了实现。源代码如下：

public final TransactionStatus getTransaction(TransactionDefinition definition)throws TransactionException {

Object transaction = doGetTransaction();

boolean debugEnabled = logger.isDebugEnabled();

if (definition == null) { // 如果没有定义事务定义，则采用默认的

// Use defaults if no transaction definition given.

definition = new DefaultTransactionDefinition();

}

if (isExistingTransaction(transaction)) {

// Existing transaction found -> check propagation behavior to find out how to behave.

return handleExistingTransaction(definition, transaction, debugEnabled);

}

// Check definition settings for new transaction.

if (definition.getTimeout() < TransactionDefinition.TIMEOUT\_DEFAULT) {

throw new InvalidTimeoutException("Invalid transaction timeout", definition.getTimeout());

}

// No existing transaction found -> check propagation behavior to find out how to proceed.

if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_MANDATORY) {

throw new IllegalTransactionStateException("No existing transaction found for transaction marked with propagation 'mandatory'");

} else if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_REQUIRED ||

definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_REQUIRES\_NEW ||

definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_NESTED) {

SuspendedResourcesHolder suspendedResources = suspend(null);

if (debugEnabled) {

logger.debug("Creating new transaction with name [" + definition.getName() + "]: " + definition);

}

try {

boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION\_NEVER);

DefaultTransactionStatus status = newTransactionStatus(definition, transaction, true, newSynchronization, debugEnabled, suspendedResources);

doBegin(transaction, definition);

prepareSynchronization(status, definition);

return status;

} catch (RuntimeException ex) {

resume(null, suspendedResources);

throw ex;

} catch (Error err) {

resume(null, suspendedResources);

throw err;

}

} else {

// Create "empty" transaction: no actual transaction, but potentially synchronization.

boolean newSynchronization = (getTransactionSynchronization() == SYNCHRONIZATION\_ALWAYS);

return prepareTransactionStatus(definition, null, true, newSynchronization, debugEnabled, null);

}

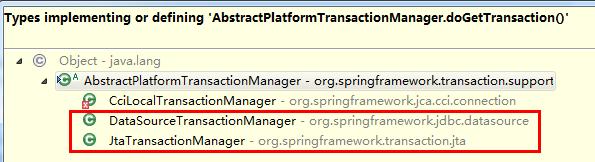
}

代码详解：

1、

Object transaction = doGetTransaction();

调用AbstractPlatformTransactionManager的实现类方法，为当前事务状态返回一个事务对象。详细见[doGetTransaction](#_doGetTransaction).



2、

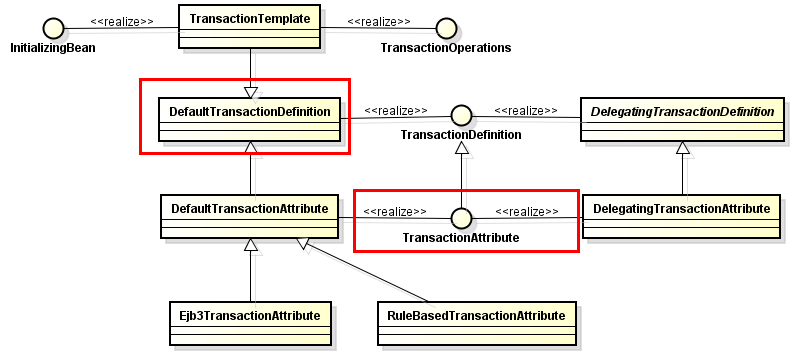
if (definition == null) {

// Use defaults if no transaction definition given.

definition = new DefaultTransactionDefinition();

}

// 如果没有事务定义，则采用默认的事务定义配置。



3、

if (isExistingTransaction(transaction)) {

// Existing transaction found -> check propagation behavior to find out how to behave.

return handleExistingTransaction(definition, transaction, debugEnabled);

}

判断当前事务是否已经存在并且已经开启。不同的实现类有不同的判断逻辑，详细见[isExistingTransaction](#_isExistingTransaction)。

4、

if (definition.getTimeout() < TransactionDefinition.TIMEOUT\_DEFAULT) {

throw new InvalidTimeoutException("Invalid transaction timeout", definition.getTimeout());

}

判断是否定义中的timeout属性是否有效，TransactionDefinition.TIMEOUT\_DEFAULT值为-1;

5、根据不同的事务级别处理不同的逻辑，如下：

A、

if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_MANDATORY) {

throw new IllegalTransactionStateException("No existing transaction found for transaction marked with propagation 'mandatory'");

}

如果事务传播行为为TransactionDefinition.PROPAGATION\_MANDATORY[2],则终止受理。

B、

else if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_REQUIRED ||

definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_REQUIRES\_NEW ||

definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_NESTED) {

SuspendedResourcesHolder suspendedResources = suspend(null);

try {

boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION\_NEVER);

DefaultTransactionStatus status = newTransactionStatus(definition, transaction, true, newSynchronization, debugEnabled, suspendedResources);

doBegin(transaction, definition);

prepareSynchronization(status, definition);

return status;

} catch (RuntimeException ex) {

resume(null, suspendedResources);

throw ex;

} catch (Error err) {

resume(null, suspendedResources);

throw err;

}

}

事务传播行为为TransactionDefinition.PROPAGATION\_REQUIRED[0]、TransactionDefinition.PROPAGATION\_REQUIRES\_NEW [3]、TransactionDefinition.PROPAGATION\_NESTED[6]处理逻辑。

SuspendedResourcesHolder suspendedResources = suspend(null);/

悬挂当前事务，详细见[suspend](#_suspend)。

boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION\_NEVER);

DefaultTransactionStatus status = newTransactionStatus(definition, transaction, true, newSynchronization, debugEnabled, suspendedResources);

判断当前事务的同步属性是否为SYNCHRONIZATION\_NEVER[2],创建一个新的TransactionStatus实例。

doBegin(transaction, definition);

开启一个事务，有实现类处理，详细见[doBegin](#_doBegin).

prepareSynchronization(status, definition);

如果一个新的事务同步打开了该事务,则将相关信息交给TransactionSynchronizationManager管理。

c、

else {

// Create "empty" transaction: no actual transaction, but potentially synchronization.

boolean newSynchronization = (getTransactionSynchronization() == SYNCHRONIZATION\_ALWAYS);

return prepareTransactionStatus(definition, null, true, newSynchronization, debugEnabled, null);

}

如果事务传播行为在A、B得不到支持，则创建一个空事务。

##### doGetTransaction

不同的事务管理器有不同的实现，这里我们可以看看局部事务管理器(DataSourceTransactionManager)和全局事务管理器(JtaTransactionManager)对此方法的实现。

**DataSourceTransactionManager**

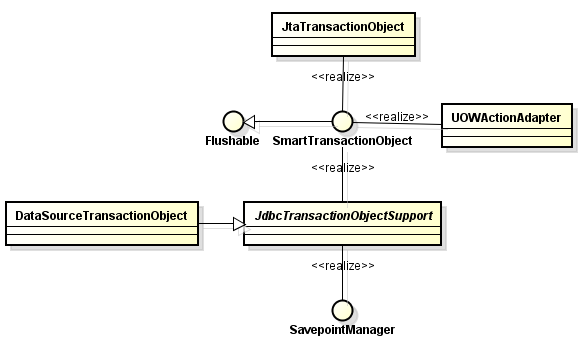
DataSourceTransactionObject txObject = new DataSourceTransactionObject();

txObject.setSavepointAllowed(isNestedTransactionAllowed());

ConnectionHolder conHolder = (ConnectionHolder) TransactionSynchronizationManager.getResource(this.dataSource);

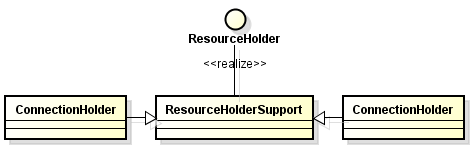
txObject.setConnectionHolder(conHolder, false);

创建一个事务对象DataSourceTransactionObject，此事务对象继承了JdbcTransactionObjectSupport类，类关系图如下：



判断当前是否是否为嵌套式事务，并设置其保存点。

根据DataSource信息，封装ConnectHolder对象，并将其设置到DataSourceTransactionObject中。



**JtaTransactionManager**

**前提**

可以在Spring上下文配置文件中进行以下配置开启JTA全局事务管理：

<tx:jta-transaction-manager/>

那么在IoC容器初始化的时候，会调用JtaTransactionManagerBeanDefinitionParser去解析此xml 元素并获取对应的事务管理器JtaTransactionManager。调用JtaTransactionManager的无参构造器实例化：

public JtaTransactionManager() {

setNestedTransactionAllowed(true); // 设置事务管理器允许嵌入式事务管理

}

实例化之后，BeanFactory会调用JtaTransactionManager(实现了InitializingBean)的afterPropertiesSet()完成实例化最后的配置。

public void afterPropertiesSet() throws TransactionSystemException {

initUserTransactionAndTransactionManager(); // 实例化UserTransaction和TransactionManager，通过JNDI获取

checkUserTransactionAndTransactionManager();// 检验UserTransaction和TransactionManager是否被实例化

initTransactionSynchronizationRegistry();// 实例化TransactionSynchronizationRegistry

}

**处理**

UserTransaction ut = getUserTransaction();

if (ut == null) { // JtaTransactionManager实例化的时候会初始化UserTransaction

throw new CannotCreateTransactionException("No JTA UserTransaction available - " + "programmatic PlatformTransactionManager.getTransaction usage not supported");

}

if (!this.cacheUserTransaction) { // 判断是否缓存了UserTransaction，如果没有则通过JNDI去查询。

ut = lookupUserTransaction(this.userTransactionName != null ? this.userTransactionName : DEFAULT\_USER\_TRANSACTION\_NAME);

}

return doGetJtaTransaction(ut);// 将UserTransaction包装成JtaTransactionObject。

##### isExistingTransaction

**DataSourceTransactionManager**

protected boolean isExistingTransaction(Object transaction) {

DataSourceTransactionObject txObject = (DataSourceTransactionObject) transaction;

return (txObject.getConnectionHolder() != null && txObject.getConnectionHolder().isTransactionActive());

}

JDBC事务是否为活动态，如果为活动态则返回true。

**JtaTransactionManager**

判断UserTransaction中的status，如果不为Stauts.STATUS\_NO\_TRANSACTION[6]则返回true。

Stauts.STATUS\_ACTIVE = 0;

Stauts.STATUS\_MARKED\_ROLLBACK = 1;

Stauts.STATUS\_PREPARED = 2;

Stauts.STATUS\_COMMITTED = 3;

Stauts.STATUS\_ROLLEDBACK = 4;

Stauts.STATUS\_UNKNOWN = 5;

Stauts.STATUS\_NO\_TRANSACTION = 6;

Stauts.STATUS\_PREPARING = 7;

Stauts.STATUS\_COMMITTING = 8;

Stauts.STATUS\_ROLLING\_BACK = 9;

如果isExistingTransaction返回true，则AbstractPlatformTransactionManager中的getTransaction会继续调用handleExistingTransaction处理已存在异常。详细见[handleExistingTransaction](#_handleExistingTransaction)。

##### handleExistingTransaction

private TransactionStatus handleExistingTransaction(TransactionDefinition definition, Object transaction, boolean debugEnabled) throws TransactionException {

if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_NEVER) {

throw new IllegalTransactionStateException("Existing transaction found for transaction marked with propagation 'never'");

}

if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_NOT\_SUPPORTED) {

if (debugEnabled) {

logger.debug("Suspending current transaction");

}

Object suspendedResources = suspend(transaction);

boolean newSynchronization = (getTransactionSynchronization() == SYNCHRONIZATION\_ALWAYS);

return prepareTransactionStatus(definition, null, false, newSynchronization, debugEnabled, suspendedResources);

}

if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_REQUIRES\_NEW) {

if (debugEnabled) {

logger.debug("Suspending current transaction, creating new transaction with name [" + definition.getName() + "]");

}

SuspendedResourcesHolder suspendedResources = suspend(transaction);

try {

boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION\_NEVER);

DefaultTransactionStatus status = newTransactionStatus(definition, transaction, true, newSynchronization, debugEnabled, suspendedResources);

doBegin(transaction, definition);

prepareSynchronization(status, definition);

return status;

} catch (RuntimeException beginEx) {

resumeAfterBeginException(transaction, suspendedResources, beginEx);

throw beginEx;

} catch (Error beginErr) {

resumeAfterBeginException(transaction, suspendedResources, beginErr);

throw beginErr;

}

}

if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION\_NESTED) {

if (!isNestedTransactionAllowed()) {

throw new NestedTransactionNotSupportedException("Transaction manager does not allow nested transactions by default - " + "specify 'nestedTransactionAllowed' property with value 'true'");

}

if (debugEnabled) {

logger.debug("Creating nested transaction with name [" + definition.getName() + "]");

}

if (useSavepointForNestedTransaction()) {

// Create savepoint within existing Spring-managed transaction,

// through the SavepointManager API implemented by TransactionStatus.

// Usually uses JDBC 3.0 savepoints. Never activates Spring synchronization.

DefaultTransactionStatus status = prepareTransactionStatus(definition, transaction, false, false, debugEnabled, null);

status.createAndHoldSavepoint();

return status;

} else {

// Nested transaction through nested begin and commit/rollback calls.

// Usually only for JTA: Spring synchronization might get activated here

// in case of a pre-existing JTA transaction.

boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION\_NEVER);

DefaultTransactionStatus status = newTransactionStatus(definition, transaction, true, newSynchronization, debugEnabled, null);

doBegin(transaction, definition);

prepareSynchronization(status, definition);

return status;

}

}

// Assumably PROPAGATION\_SUPPORTS or PROPAGATION\_REQUIRED.

if (debugEnabled) {

logger.debug("Participating in existing transaction");

}

if (isValidateExistingTransaction()) {

if (definition.getIsolationLevel() != TransactionDefinition.ISOLATION\_DEFAULT) {

Integer currentIsolationLevel = TransactionSynchronizationManager.getCurrentTransactionIsolationLevel();

if (currentIsolationLevel == null || currentIsolationLevel != definition.getIsolationLevel()) {

Constants isoConstants = DefaultTransactionDefinition.constants;

throw new IllegalTransactionStateException("Participating transaction with definition [" + definition + "] specifies isolation level which is incompatible with existing transaction: " + (currentIsolationLevel != null ? isoConstants.toCode(currentIsolationLevel, DefaultTransactionDefinition.PREFIX\_ISOLATION) : "(unknown)"));

}

}

if (!definition.isReadOnly()) {

if (TransactionSynchronizationManager.isCurrentTransactionReadOnly()) {

throw new IllegalTransactionStateException("Participating transaction with definition [" + definition + "] is not marked as read-only but existing transaction is");

}

}

}

boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION\_NEVER);

return prepareTransactionStatus(definition, transaction, false, newSynchronization, debugEnabled, null);

}

在一个已存在的事务上，这个方法的主要逻辑是针对不同的事务传播行为进行不同的处理：

TransactionDefinition.PROPAGATION\_NEVER：此事务传播行为不支持。

TransactionDefinition.PROPAGATION\_NOT\_SUPPORTED：暂停当前事务，创建一个新DefaultTransactionStatus（不包含新事务）实例，并指明前一个暂停的事务。

TransactionDefinition.PROPAGATION\_REQUIRES\_NEW：暂停当前事务，创建一个新DefaultTransactionStatus（包含新事务）实例，并指明前一个暂停的事务，开启事务。

TransactionDefinition.PROPAGATION\_NESTED：判断PlatformTransactionManager是否支持嵌入式事务传播行为，如果不支持则抛出NestedTransactionNotSupportedException异常。判断PlatformTransactionManager是否允许设置保存点，默认是为true的，但是JtaTransactionManager是不允许的。

##### suspend

##### doBegin

事务开启

**JtaTransactionManager**

applyIsolationLevel(txObject, definition.getIsolationLevel());// JTA不允许自定义事务隔离级别

int timeout = determineTimeout(definition);// 获取超时时间

applyTimeout(txObject, timeout);// 设置UserTransaction超时时间

txObject.getUserTransaction().begin();// UserTransaction开启事务、最后UserTransaction将委派TransactionManager开启事务管理。而TransactionManager将创建物理Transaction，真正开启事务管理。具体的实现有JTA服务提供商实现。

##### DataSourceTransactionManager

从DataSourceTransactionObject中获取ConnectionHolder，如果没有则从DataSource中获取到Connection，并封装新的ConnectionHolder到DataSourceTransactionObject。设置ConnectionHolder相关参数，设置Connection的AutoCommit。

### 源码分析 声明式事务管理 for 全局事务

目前jotm和atomikos是JTA比较好的实现

### 源码分析 编程式事务管理

### 源码分析 Spring JDBC