# Jdbc事务

我们看一下使用jdbc如何进行事务操作。

public void save(User user) throws SQLException {  
 Connection conn = JdbcDao.*getConnection*();  
 try {  
 conn.setAutoCommit(false);  
 PreparedStatement ps = conn.prepareStatement("INSERT INTO users(name, age) VALUES(?,?)");  
 ps.setString(1, user.getName());  
 ps.setInt(2, user.getAge());  
 ps.execute();  
 conn.commit();  
 } catch (SQLException e) {  
 e.printStackTrace();  
 conn.rollback();  
 } finally {  
 conn.close();  
 }  
}

分析以下几点：

1、怎么使用事务？将自动提交设置为false，即conn.setAutoCommit(false)，然后手动commit或者rollback。

2、谁在执行事务？在代码中，我们通过操作Connection来执行事务，但这并不代表Connection具有事务功能，而是Connection把一些命令如commit、rollback传递给数据库，使用了数据库自身的事务功能。

3、单纯的使用jdbc执行事务存在哪些问题？ commit、rollback等这些属于事务代码，其他的执行sql语句的属于业务代码。可以看到，业务代码都要嵌套在try-catch事务模板代码中，导致代码混杂。因此，最好是将事务代码和业务代码解耦，但要注意的是，只有事务代码和业务代码使用的是同一个Connection时，事务的回滚和提交才能正常进行，所以如果要实现事务代码和业务代码的分离，就必须要保证它们使用的是同一个Connection。

# Hibernate事务

public void save(User user) {  
 Session session = HibernateDao.*getCurrentSession*();  
 Transaction tx = null;  
 try {  
 tx = session.beginTransaction();  
 session.save(user);  
 tx.commit();  
 } catch (Exception e) {  
 tx.rollback();  
 } finally {  
 session.close();  
 }  
}

分析以下几点：

1、事务功能和业务功能的分离：jdbc事务中，Connection承载了事务管理和执行sql语句这两方面的功能，Hibernate把这两方面的功能单独开来，将事务功能交给了Transaction，将执行sql语句交给了Session，从而明确分工。

2、Hibernate中的事务原理：在Session和Transaction内部有一个相同的Connection，这样就保证了事务代码和业务代码使用的是同一个Connection。Transaction事务的提交、回滚等都是依靠内部的Connection来完成的，这一点从Hibernate源码可以看出。

（1）开始事务，通过Connection.setAutoCommit(false)设置自动提交为false。

@Override  
public void begin() {  
 try {  
 if ( !doConnectionsFromProviderHaveAutoCommitDisabled() ) {  
 *log*.trace( "Preparing to begin transaction via JDBC Connection.setAutoCommit(false)" );  
 getConnectionForTransactionManagement().setAutoCommit( false );  
 *log*.trace( "Transaction begun via JDBC Connection.setAutoCommit(false)" );  
 }  
 status = TransactionStatus.*ACTIVE*;  
 }  
 catch( SQLException e ) {  
 throw new TransactionException( "JDBC begin transaction failed: ", e );  
 }  
}

（2）提交事务，通过Connection.commit()提交事务。

@Override  
public void commit() {  
 try {  
 *log*.trace( "Preparing to commit transaction via JDBC Connection.commit()" );  
 getConnectionForTransactionManagement().commit();  
 status = TransactionStatus.*COMMITTED*;  
 *log*.trace( "Transaction committed via JDBC Connection.commit()" );  
 }  
 catch( SQLException e ) {  
 status = TransactionStatus.*FAILED\_COMMIT*;  
 throw new TransactionException( "Unable to commit against JDBC Connection", e );  
 }  
  
 afterCompletion();  
}

（3）回滚事务，通过Connection.rollback()回滚事务。

@Override  
public void rollback() {  
 try {  
 *log*.trace( "Preparing to rollback transaction via JDBC Connection.rollback()" );  
 getConnectionForTransactionManagement().rollback();  
 status = TransactionStatus.*ROLLED\_BACK*;  
 *log*.trace( "Transaction rolled-back via JDBC Connection.rollback()" );  
 }  
 catch( SQLException e ) {  
 throw new TransactionException( "Unable to rollback against JDBC Connection", e );  
 }  
  
 afterCompletion();  
}

# Spring事务功能的顶层接口设计与实现

由于各个厂商实现事务功能的方式不同，于是Spring进行了统一的抽象，形成了PlatformTransactionManager事务管理器接口，事务的提交、回滚等操作都交给它来实现。

## Spring事务功能的顶层接口设计

Spring主要用以下三个接口抽象出事务功能。

PlatformTransactionManager：事务管理器

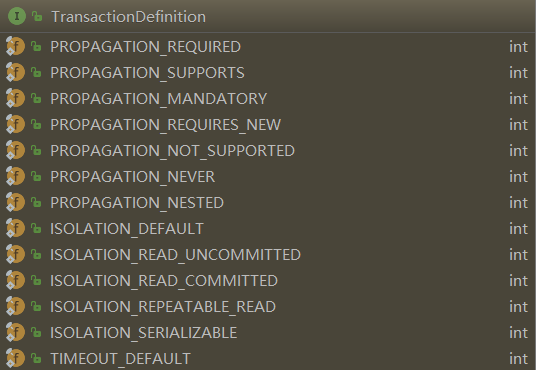
TransactionDefinitioin：事务属性，如事务的传播行为、隔离级别、超时时间等

TransactionStatus：事务状态，如是否是一个新的事务、是否已被标记为回滚等

接下来分别解析以上三个接口，以及各厂商对接口的实现。

### TransactionDefinition

该接口定义了Spring支持的事务属性，包括事务的传播行为、隔离级别、超时时间、只读。

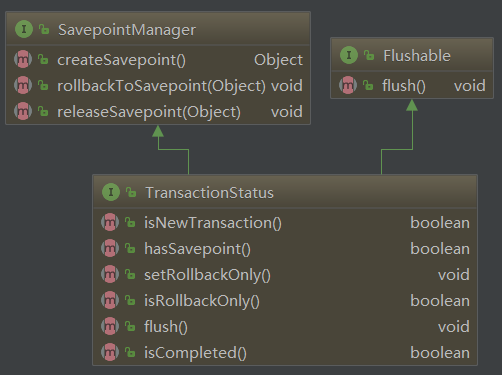


TransactoinDefinition接口的默认实现是DefaultTransactionDefinition，该类设置了事务的一些默认属性。为

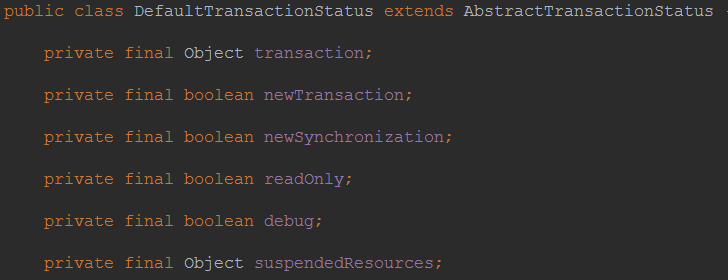
// 事务传播属性为PROPAGATION\_REQUIRED  
private int propagationBehavior = *PROPAGATION\_REQUIRED*;  
  
// 事务隔离级别采用底层数据库默认的隔离级别  
private int isolationLevel = *ISOLATION\_DEFAULT*;  
  
// 事务超时时间采用底层数据库默认的超时时间  
private int timeout = *TIMEOUT\_DEFAULT*;  
  
// 事务只读属性为false  
private boolean readOnly = false;

### TransactionStatus

TransactionStatus接口中主要存储的是事务的一些状态信息，例如是否是一个新的事务、是否有保存点、是否已被标记为回滚。同时，TransactionStatus接口继承了SavepointManager接口，对事务中的保存点功能进行了封装，例如创建保存点、回滚到指定保存点、释放指定保存点。



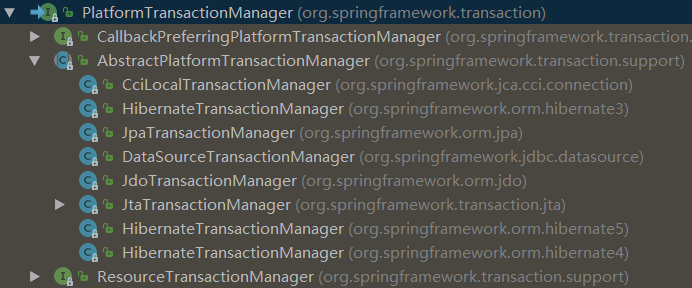
TransactionStatus接口的默认实现类是DefaultTransactionStatus。



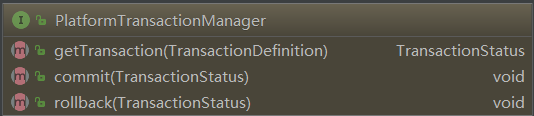
可以看到，DefaultTransactionStatus类中用一个Object类型的对象表示事务，这是为了兼容不同厂商对事务的不同实现方式，例如jdbc事务是通过Connection来实现的，Hibernate事务是通过其自身定义的Transaction接口来实现的，因此Spring只能以Object transaction的形式来表示各个厂商的事务。事务的回滚和提交等操作最终都是委托给上述Object transaction来完成的。

### PlatformTransactionMnanager

PlatformTransactionManager接口的类图如下：



PlatformTransactionManager接口提供的方法如下：



接下来重点分析AbstractPlatformTransactionManager抽象类及其两种实现类DatasourceTransactionMnaager、HibernateTransactionManager是如何实现PlatformTransactionManager接口的。

1、TransactionStatus getTransaction(TransactionDefinition definition)：根据事务定义获取事务状态。该方法的具体实现过程是：先获取transaction对象，然后判断当前事务是否存在，如果存在，则进行事务传播属性处理（见“事务传播属性解析”一节）；如果不存在，则新建一个TransactionStatus实例，此时如果事务定义中的传播属性是REQUIRED、REQUIRED\_NEW或者NESTED，则开启事务然后返回新建的TransactionStatus实例，否则，直接返回新建的TransactionStatus实例。

下面具体分析以上过程。

1.1、首先获取transaction对象。不同的事务管理器获取的是不同的transaction对象，如下：

（1）DataSourceTransactionManager获取的是DataSourceTransactionObject实例。

@Override  
protected Object doGetTransaction() {

// 新建一个DataSourceTransactionObject实例txObject  
 DataSourceTransactionObject txObject = new DataSourceTransactionObject();  
 txObject.setSavepointAllowed(isNestedTransactionAllowed());

// 获取与当前线程绑定的connectionHolder  
 ConnectionHolder conHolder =  
 (ConnectionHolder) TransactionSynchronizationManager.*getResource*(this.dataSource);

// 为txObject设置connectionHolder属性  
 txObject.setConnectionHolder(conHolder, false);  
 return txObject;  
}

（2）HibernateTransactionManager获取的是HibernateTransactionObject实例。

@Override  
protected Object doGetTransaction() {

// 新建一个HibernateTransactoinObject实例txObject  
 HibernateTransactionObject txObject = new HibernateTransactionObject();  
 txObject.setSavepointAllowed(isNestedTransactionAllowed());

// 获取与当前线程绑定的sessionHolder  
 SessionHolder sessionHolder =  
 (SessionHolder) TransactionSynchronizationManager.*getResource*(getSessionFactory());  
 if (sessionHolder != null) { // 如果sessionHolder不为空，则为txObject设置sessionHolder属性

txObject.setSessionHolder(sessionHolder);  
 }  
 else if (this.hibernateManagedSession) { // 如果sessionHolder为空，则从sessionFactory中获取一个session并

//将其封装为sessionHolder，然后为txObject设置sessionHolder属性  
 try {  
 Session session = this.sessionFactory.getCurrentSession();  
 txObject.setExistingSession(session);  
 }  
 catch (HibernateException ex) {  
 throw new DataAccessResourceFailureException(  
 "Could not obtain Hibernate-managed Session for Spring-managed transaction", ex);  
 }  
 }  
  
 if (getDataSource() != null) { // 如果datasource不为空，则获取与当前线程绑定的connectionHolder，并为txObject

// 设置connectionHolder属性  
 ConnectionHolder conHolder = (ConnectionHolder)  
 TransactionSynchronizationManager.*getResource*(getDataSource());  
 txObject.setConnectionHolder(conHolder);  
 }

// 返回txObject  
 return txObject;  
}

1.2、然后判断事务是否存在。

（1）DataSourceTransactionManager判断事务是否存在。

@Override  
protected boolean isExistingTransaction(Object transaction) {

// 将Object transaction强转为DataSourceTransactionObject txObject  
 DataSourceTransactionObject txObject = (DataSourceTransactionObject) transaction;

// 从txObject中获取connectoinHolder属性，并判断事务是否存在  
 return (txObject.getConnectionHolder() != null && txObject.getConnectionHolder().isTransactionActive());  
}

（2）HibernateTransactoinManager判断事务是否存在。

@Override  
protected boolean isExistingTransaction(Object transaction) {

// 将Object transaction强转为HibernateTransactionObject txObject

HibernateTransactionObject txObject = (HibernateTransactionObject) transaction;

// 判断事务是否存在  
 return (txObject.hasSpringManagedTransaction() ||  
 (this.hibernateManagedSession && txObject.hasHibernateManagedTransaction()));  
}

public boolean hasHibernateManagedTransaction() {

// 从txObject中获取sessionHolder属性，并判断事务是否存在  
 return (this.sessionHolder != null &&  
 this.sessionHolder.getSession().getTransaction().getStatus() == TransactionStatus.*ACTIVE*);  
}

1.3、如果事务已经存在，进行事务传播属性处理（见“事务传播属性解析”一节）。

1.4、如果事务不存在，则新建一个TransactionStatus，此时如果事务定义中的传播属性是REQUIRED、REQUIRED\_NEW或者NESTED，则开启事务然后返回新建的TransactionStatus实例，否则，直接返回新建的TransactionStatus实例。

（1）DataSourceTransactionManager开启事务。

首先将Object transaction强转为DataSourceTransactionObject txObject，然后从txObject中取出connectionHolder，如果为空，则从datasource中创建一个connection并将其封装为connectionHolder给txObject，接着将事务设置为手动提交，最后将txObject绑定到当前线程。

@Override  
protected void doBegin(Object transaction, TransactionDefinition definition) {

// 将Object transaction强转为DataSourceTransactionObject txObject  
 DataSourceTransactionObject txObject = (DataSourceTransactionObject) transaction;  
 Connection con = null;  
  
 try {

// 从txObject中获取connectionHolder，如果为空，则从datasource中新建一个connection并将其封装为

// connectionHolder给txObject  
 if (txObject.getConnectionHolder() == null ||  
 txObject.getConnectionHolder().isSynchronizedWithTransaction()) {  
 Connection newCon = this.dataSource.getConnection();  
 if (logger.isDebugEnabled()) {  
 logger.debug("Acquired Connection [" + newCon + "] for JDBC transaction");  
 }  
 txObject.setConnectionHolder(new ConnectionHolder(newCon), true);  
 }  
  
 txObject.getConnectionHolder().setSynchronizedWithTransaction(true);  
 con = txObject.getConnectionHolder().getConnection();  
  
 Integer previousIsolationLevel = DataSourceUtils.*prepareConnectionForTransaction*(con, definition);  
 txObject.setPreviousIsolationLevel(previousIsolationLevel);

// 设置事务提交形式为手动提交  
 // Switch to manual commit if necessary. This is very expensive in some JDBC drivers,  
 // so we don't want to do it unnecessarily (for example if we've explicitly  
 // configured the connection pool to set it already).  
 if (con.getAutoCommit()) {  
 txObject.setMustRestoreAutoCommit(true);  
 if (logger.isDebugEnabled()) {  
 logger.debug("Switching JDBC Connection [" + con + "] to manual commit");  
 }  
 con.setAutoCommit(false);  
 }  
  
 prepareTransactionalConnection(con, definition);

// 设置该事务为活跃状态，即将该事务标记为已存在  
 txObject.getConnectionHolder().setTransactionActive(true);  
  
 int timeout = determineTimeout(definition);  
 if (timeout != TransactionDefinition.*TIMEOUT\_DEFAULT*) {  
 txObject.getConnectionHolder().setTimeoutInSeconds(timeout);  
 }

// 如果刚才中txObject中获取的connectionHolder为空，则此处将新创建的connectionHolder绑定到当前线程  
 // Bind the connection holder to the thread.  
 if (txObject.isNewConnectionHolder()) {  
 TransactionSynchronizationManager.*bindResource*(getDataSource(), txObject.getConnectionHolder());  
 }  
 }  
  
 catch (Throwable ex) {  
 if (txObject.isNewConnectionHolder()) {  
 DataSourceUtils.*releaseConnection*(con, this.dataSource);  
 txObject.setConnectionHolder(null, false);  
 }  
 throw new CannotCreateTransactionException("Could not open JDBC Connection for transaction", ex);  
 }  
}

（2）HibernateTransactionManager开启事务。

也是同样的逻辑，首先将Object transaction强转为HibernateTransactionObject txObject；然后从txObject中取出sessionHolder，如果为空，则从sessionFactory中创建一个session并将其封装为sessionHolder给txObject；接着从session中获取transaction并开启事务；然后判断datasource是否为空，如果不为空，则新建一个connectionHolder给txObject；最后将txObject绑定到当前线程。

2、void rollback(TransactionStatus status)：回滚事务。

DataSourceTransactionManager使用DataSourceTransactionObject对象中的Connection来回滚事务，如下：

@Override  
protected void doRollback(DefaultTransactionStatus status) {  
 DataSourceTransactionObject txObject = (DataSourceTransactionObject) status.getTransaction();  
 Connection con = txObject.getConnectionHolder().getConnection();  
 if (status.isDebug()) {  
 logger.debug("Rolling back JDBC transaction on Connection [" + con + "]");  
 }  
 try {  
 con.rollback();  
 }  
 catch (SQLException ex) {  
 throw new TransactionSystemException("Could not roll back JDBC transaction", ex);  
 }  
}

HibernateTransactionManager使用HibernateTransactionObject对象中的SessionHolder里的Session创建的Transaction对象来回滚事务，如下：

@Override  
protected void doRollback(DefaultTransactionStatus status) {  
 HibernateTransactionObject txObject = (HibernateTransactionObject) status.getTransaction();  
 if (status.isDebug()) {  
 logger.debug("Rolling back Hibernate transaction on Session [" +  
 txObject.getSessionHolder().getSession() + "]");  
 }  
 try {  
 txObject.getSessionHolder().getTransaction().rollback();  
 } finally {  
 if (!txObject.isNewSession() && !this.hibernateManagedSession) {  
 // Clear all pending inserts/updates/deletes in the Session.  
 // Necessary for pre-bound Sessions, to avoid inconsistent state.  
 txObject.getSessionHolder().getSession().clear();  
 }  
}

3、void commit(TransactionStatus status)：提交事务。

同理，DataSourceTransactionManager使用DataSourceTransactionObject对象中的Connection来提交事务，HibernateTransactionManager使用HibernateTransactionObject对象中的SessionHolder里的Session创建的Transaction对象来提交事务。

### 事务传播属性

分析PlatformTransactionManager源码时讲到，在执行TransactionStatus getTransaction(TransactionDefinition definition)方法时，如果事务已经存在，则进行事务传播处理，即执行TransactionStatus handleExistingTransaction(TransactionDefinition definition, Object transaction, boolean debugEnabled)方法，该方法中针对不同的事务传播属性进行了不同的处理。

1、PROPAGATION\_NEVER：当前方法不应该运行在事务中，如果有运行的事务，则抛出异常。

if (definition.getPropagationBehavior() == TransactionDefinition.*PROPAGATION\_NEVER*) {  
 throw new IllegalTransactionStateException(  
 "Existing transaction found for transaction marked with propagation 'never'");  
}

2、PROPAGATION\_NOT\_SUPPORTED：当前方法不应该运行在事务中，如果有运行的事务，则需要将事务挂起，等到方法执行完毕后，再将挂起的事务恢复。

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);  
}

对于DataSourceTransactionManager来说，事务的挂起就是把ConnectionHolder与当前线程解除绑定；对于HibernateTransactionManager俩说，事务的挂起就是把SessionHolder与当前线程解除绑定。无论是哪种TransactionManager，在事务挂起的最后都会创建一个新的TransactionStatus，同时把一个transaction为空的事务以及挂起的资源传递给它。

在恢复挂起事务时，对于DataSourceTransactionManager来说，就是把ConnectionHolder重新绑定到当前线程；对于HibernateTransactionManager来说，就是把SessionHolder重新绑定到当前线程。

3、PROPAGATION\_REQUIRES\_NEW：当前方法必须启动新事务，并在它自己的事务中运行，如果有事务正在运行，应该将其挂起。

SuspendedResourcesHolder suspendedResources = suspend(transaction);  
try {  
 boolean newSynchronization = (getTransactionSynchronization() != *SYNCHRONIZATION\_NEVER*);  
 DefaultTransactionStatus status = newTransactionStatus(  
 definition, transaction, true, newSynchronization, debugEnabled, suspendedResources);  
 // 启动新事务

doBegin(transaction, definition);  
 prepareSynchronization(status, definition);  
 return status;  
}

4、PROPAGATION\_NESTED：如果有事务在运行，当前方法就应该在这个事务的嵌套事务中运行，否则，就启动一个新的事务，并在它自己的事务中运行。

最容易令人混淆的是PROPAGATION\_REQUIRES\_NEW和PROPAGATION\_NESTED。PROPAGATION\_REQUIRES\_NEW启动一个新的，不依赖于环境的"内部"事务，这个事务将被完全commited或rolledback而不依赖于外部事务，它拥有自己的隔离范围、自己的锁等等，当内部事务开始执行时，外部事务将被挂起，内务事务结束时，外部事务将继续执行PROPAGATION\_NESTED开始一个"嵌套的"事务，它是已经存在事务的一个真正的子事务，嵌套事务开始执行时，它将取得一个savepoint，如果这个嵌套事务失败，我们将回滚到此savepoint，潜套事务是外部事务的一部分，只有外部事务结束后它才会被提交。可以看到，PROPAGATION\_REQUIRES\_NEW和PROPAGATION\_NESTED的最大区别在于：PROPAGATION\_REQUIRES\_NEW完全是一个新的事务，而PROPAGATION\_NESTED则是外部事务的子事务，如果外部事务commit，嵌套事务也会被commit，这个规则同样适用于rollback。举个例子：

有一个业务场景有两点需求：

（1）需要事务BC与事务AD一起commit，即：作为事务AD的子事务，事务BC只有在事务AD成功commit时才commit。这个需求简单称之为“联合成功”，这一点PROPAGATION\_REQUIRED可以做到。

（2）需要事务BC的rollback不影响事务AD的commit。这个需求简单称之为“隔离失败”，这一点PROPAGATION\_REQUIRES\_NEW可以做到。

使用PROPAGATION\_REQUIRED满足需求1，但子事务BC的rollback会无条件地使父事务AD也rollback，不能满足需求2；使用PROPAGATION\_REQUIRES\_NEW满足需求2，但内部事务BC是完全新的事务上下文，外部事务AD的成功与否完全不影响BC的提交，不能满足需求1。同时满足上述两条需求就要用到PROPAGATION\_NESTED了。

PROPAGATION\_NESTED在事务AD执行到B点时，设置了savePoint。当BC事务成功commit时，PROPAGATION\_NESTED的行为与PROPAGATION\_REQUIRED一样，只有当事务AD在D点成功commit时，事务BC才真正commit，从而满足了“联合成功”；当BC事务异常导致rollback时，因为设置了savePoint，AD事务可以选择与BC一起rollback或继续执行其他分支，而仅仅让BC回滚，从而满足了“隔离失败”。

5、PROPAGATION\_SUPPORTS：如果有事务在运行，当前方法就在这个事务内运行，否则，它不运行在事务中。

6、PROPAGATION\_REQUIRED：如果有事务在运行，当前方法就在这个事务内运行，否则，就启动一个新的事务，并在自己的事务中运行。

## Spring事务模型

除了本地事务模型之外，Spring还提供了两种事务模型：编程式事务模型和声明式事务模型。

### 编程式事务

编程式事务：通过手动编程的方式实现事务操作，通常是由开发人员来管理事务的提交和回滚，但也可能是Spring来管理事务，如Spring的TransactionTemplate。

配置文件如下：

<context:component-scan base-package="com.kevin" use-default-filters="true">  
 <context:exclude-filter type="annotation" expression="org.springframework.stereotype.Controller"/>  
 <context:exclude-filter type="annotation"  
 expression="org.springframework.web.bind.annotation.ControllerAdvice"/>  
</context:component-scan>  
  
<!-- 引入外部文件 -->  
<context:property-placeholder location="c3p0.properties"/>  
  
<!-- 配置数据源 -->  
<bean id="comboPooledDataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">  
 <property name="driverClass" value="${jdbc.driverClass}"/>  
 <property name="user" value="${jdbc.username}"/>  
 <property name="password" value="${jdbc.password}"/>  
 <property name="jdbcUrl" value="${jdbc.url}"/>  
 <property name="initialPoolSize" value="${jdbc.initialPoolSize}"/>  
 <property name="maxPoolSize" value="${jdbc.maxPoolSize}"/>  
 <property name="acquireIncrement" value="${jdbc.acquireIncrement}"/>  
 <property name="acquireRetryAttempts" value="${jdbc.acquireRetryAttempts}"/>  
</bean>  
  
<!--\*\*\*\*\*\*\*\*\*配置使用DatasourceTransactionManager进行事务管理\*\*\*\*\*\*\*\*\*\*\*-->  
<!-- 配置JdbcTemplate -->  
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">  
 <property name="dataSource" ref="comboPooledDataSource"/>  
</bean>  
  
<!-- 配置DataSourceTransactionManager事务管理器 -->  
<bean id="dataSourceTransactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
 <property name="dataSource" ref="comboPooledDataSource"/>  
</bean>  
  
<!-- 配置TransactoionTemplate编程式事务 -->  
<bean id="dataSourceTransactionTemplate" class="org.springframework.transaction.support.TransactionTemplate">  
 <property name="transactionManager" ref="dataSourceTransactionManager"/>  
</bean>  
  
<!--\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*HibernateTransactionManager进行事务管理\*\*\*\*\*\*\*\*\*\*\*\*-->  
<!-- 配置SessionFactory -->  
<bean id="sessionFactory" class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">  
 <property name="dataSource" ref="comboPooledDataSource"/>  
 <property name="hibernateProperties">  
 <props>  
 <prop key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>  
 <prop key="hibernate.show\_sql">true</prop>  
 <prop key="hibernate.hbm2ddl.auto">update</prop>  
 </props>  
 </property>  
 <property name="packagesToScan">  
 <list>  
 <value>com.kevin.domain</value>  
 </list>  
 </property>  
</bean>  
  
<!-- 配置HibernateTemplate -->  
<bean id="hibernateTemplate" class="org.springframework.orm.hibernate5.HibernateTemplate">  
 <property name="sessionFactory" ref="sessionFactory"/>  
</bean>  
  
<!-- 配置HibernateTransactionManager事务管理器 -->  
<bean id="hibernateTransactionManager" class="org.springframework.orm.hibernate5.HibernateTransactionManager">  
 <property name="sessionFactory" ref="sessionFactory"/>  
</bean>  
  
<!-- 配置TransactoionTemplate编程式事务 -->  
<bean id="hibernateTransactionTemplate" class="org.springframework.transaction.support.TransactionTemplate">  
 <property name="transactionManager" ref="hibernateTransactionManager"/>  
</bean>

测试用例如下：

@Test  
public void testTransactionTemplateForDataSourceTransactionManager() {  
 transactionTemplate.execute(new TransactionCallback<User>() {  
 @Override  
 public User doInTransaction(TransactionStatus status) {  
 String sql = "INSERT INTO users(name, age) VALUES ('Tank', 23)";  
 jdbcTemplate.update(sql);  
 return null;  
 }  
 });  
}

@Test  
public void testTransactionTemplateForHibernateTransactoinManager() {  
 transactionTemplate2.execute(new TransactionCallback<User>() {  
 @Override  
 public User doInTransaction(TransactionStatus status) {  
 User user = new User("Kite", 25);  
 hibernateTemplate.save(user);  
 return null;  
 }  
 });  
}

接下来详解如何使用Spring提供的TransactionTemplate来进行编程式事务。

TransactionTemplate继承了DefaultTransactionDefinition，因此拥有默认的事务定义，当然，我们也可以自定义事务传播属性、隔离级别等。另外，TransactionTemplate注入一个PlatformTransactionManage事务管理器，用来执行事务。最后，TransactionTemplate的execute()方法中提供了一个回调函数，我们可以在该回调函数中执行业务代码，而事务代码则被封装起来，包裹在业务代码周围，如下：

@Override  
public <T> T execute(TransactionCallback<T> action) throws TransactionException {  
 if (this.transactionManager instanceof CallbackPreferringPlatformTransactionManager) {  
 return ((CallbackPreferringPlatformTransactionManager) this.transactionManager).execute(this, action);  
 }  
 else {  
 TransactionStatus status = this.transactionManager.getTransaction(this);  
 T result;  
 try {  
 result = action.doInTransaction(status);  
 }  
 catch (RuntimeException ex) {  
 // Transactional code threw application exception -> rollback  
 rollbackOnException(status, ex);  
 throw ex;  
 }  
 catch (Error err) {  
 // Transactional code threw error -> rollback  
 rollbackOnException(status, err);  
 throw err;  
 }  
 catch (Throwable ex) {  
 // Transactional code threw unexpected exception -> rollback  
 rollbackOnException(status, ex);  
 throw new UndeclaredThrowableException(ex, "TransactionCallback threw undeclared checked exception");  
 }  
 this.transactionManager.commit(status);  
 return result;  
 }  
}

首先，PlatformTransactionManager事务管理器通过TransactionStatus getTransaction(TransactionDefinition definition)方法根据事务定义获取事务状态，不同的厂商对该方法的实现不尽相同，“PlatformTransactionManager”一节中已经讲过。

然后，在TransactionCallback的doInTransaction()方法中执行业务代码。

（1）如果我们使用的是DataSourceTransactionManager，则可以使用jdbcTemplate来执行业务逻辑，或者通过DataSourceUtil获取Connection，再使用Connection来执行业务逻辑。

因为事务代码是通过与当前线程绑定的ConnectionHolder中的Connection来执行的，所以要保证业务代码和事务代码中使用的是同一个Connection，才能正确地提交和回滚事务。JdbcTemplate在执行sql语句时，会使用DataSourceUtils中的Connection getConnection(DataSource dataSource)方法从DataSource中获取一个Connetion，过程如下：

public static Connection doGetConnection(DataSource dataSource) throws SQLException {  
 Assert.*notNull*(dataSource, "No DataSource specified");  
  
 ConnectionHolder conHolder =

(ConnectionHolder) TransactionSynchronizationManager.*getResource*(dataSource);  
 if (conHolder != null && (conHolder.hasConnection() || conHolder.isSynchronizedWithTransaction())) {  
 conHolder.requested();  
 if (!conHolder.hasConnection()) {  
 *logger*.debug("Fetching resumed JDBC Connection from DataSource");  
 conHolder.setConnection(dataSource.getConnection());  
 }  
 return conHolder.getConnection();  
 }

可以看到，DataSourceUtils获取Connection时，也是先获取与当前线程绑定的ConnectionHolder，然后再获取Connection。因为事务在执行业务逻辑前已经开启，因此已经有了和当前线程绑定的ConnectionHolder，这样就保证了事务也业务使用的是同一个Connection。

（2）如果我们使用的是HibernateTransactionManager，则可以使用HibernateTemplate来执行业务逻辑，或者通过SessionFactory的getCurrentSession()方法来获取与当前线程绑定的Session执行业务逻辑。

因为事务代码是通过与当前线程绑定的SessionHolder中的Session中的Transaction来执行的，所以要保证业务代码和事务代码使用的是同一个Session，才能正确地提交和回滚事务。HibernateTemplate在执行save()等方法时，最终调用的都是doExecute()方法，在该方法中会使用如下方式获取一个与当前线程绑定的Session：

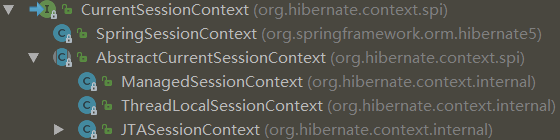
session = getSessionFactory().getCurrentSession();

即采用SessionFactory.getCurrentSession()方法来获取与当前线程绑定的Session。

我们还可以进一步分析SessionFactory.getCurrentSession()方法时如何获取与当前线程绑定的Session的。分析源码：

public Session getCurrentSession() throws HibernateException {  
 if ( currentSessionContext == null ) {  
 throw new HibernateException( "No CurrentSessionContext configured!" );  
 }  
 return currentSessionContext.currentSession();  
}

可以看到，当前Session是通过调用CurrentSessionContext接口中的currentSession()得到的，该接口的类图如下：



在整合了Hibernate的Spring环境中，默认使用的SpringSessionContext，它获取当前Session的方式如下：

public Session currentSession() throws HibernateException {

// 获取与当前线程绑定的SessionHolder  
 Object value = TransactionSynchronizationManager.*getResource*(this.sessionFactory);  
 if (value instanceof Session) {  
 return (Session) value;  
 }  
 else if (value instanceof SessionHolder) {  
 SessionHolder sessionHolder = (SessionHolder) value;

// 通过SessionHolder获取Session  
 Session session = sessionHolder.getSession();  
 if (!sessionHolder.isSynchronizedWithTransaction() &&  
 TransactionSynchronizationManager.*isSynchronizationActive*()) {  
 TransactionSynchronizationManager.*registerSynchronization*(  
 new SpringSessionSynchronization(sessionHolder, this.sessionFactory, false));  
 sessionHolder.setSynchronizedWithTransaction(true);  
 // Switch to FlushMode.AUTO, as we have to assume a thread-bound Session  
 // with FlushMode.MANUAL, which needs to allow flushing within the transaction.  
 FlushMode flushMode = SessionFactoryUtils.*getFlushMode*(session);  
 if (flushMode.equals(FlushMode.*MANUAL*) &&  
 !TransactionSynchronizationManager.*isCurrentTransactionReadOnly*()) {  
 session.setFlushMode(FlushMode.*AUTO*);  
 sessionHolder.setPreviousFlushMode(flushMode);  
 }  
 }  
 return session;  
 }

可以看到，SpringSessionContext也是先获取与当前线程绑定 的SessionHolder，然后再获取Session。因为事务在执行业务逻辑前已经开启，因此已经有了和当前线程绑定的SessionHolder，这样就保证了事务也业务使用的是同一个Sessoin。

最后，通过this.transactionManager.commit(status)提交事务。

### 声明式事务

Spring的声明式事务有三种配置方式，这三种配置方式的拦截原理都是一样的，我们先通过一个Demo分析声明式事务的拦截原理。

Demo如下：

@Repository  
public class AopUserDao implements InitializingBean {  
  
 @Autowired  
 private UserDao userDao;  
  
 @Autowired  
 @Qualifier("hibernateTransactionManager")  
 private PlatformTransactionManager hiberTransactionManager;  
  
 private UserDao proxyUserDao;  
  
 @Override  
 public void afterPropertiesSet() throws Exception {  
 ProxyFactory proxyFactory = new ProxyFactory();  
 proxyFactory.setTarget(userDao);  
  
 TransactionInterceptor transactionInterceptor = new TransactionInterceptor();  
 transactionInterceptor.setTransactionManager(hiberTransactionManager);  
 Properties properties = new Properties();  
 // 对UserDao中的所有方法设置事务传播属性为PROPAGATION\_REQUIRED  
 properties.setProperty("\*", "PROPAGATION\_REQUIRED");  
 transactionInterceptor.setTransactionAttributes(properties);  
  
 proxyFactory.addAdvice(transactionInterceptor);  
 proxyUserDao = (UserDao) proxyFactory.getProxy();  
 }  
  
 public void save(User user) {  
 proxyUserDao.save(user);  
 }  
}

总的来说，还是针对目标对象创建一个Aop动态代理对象，然后通过代理对象代理对目标对象的访问。

对目标对象UserDao进行代理的具体过程如下；

（1）创建一个ProxyFactory类实例；

（2）设置要代理的对象，即设置目标对象；

（3）创建一个事务拦截器，并设置事务管理器transactionManager和事务属性transactionDefinition，由该事务管理器负责完成事务的创建、提交和回滚。

（4）对代理对象加入事务拦截器。这里有两种拦截方式，一种是调用proxyFactory.addAdvice()方法，表示拦截UserDao中的所有方法；一种是调用proxyFactory.addAdvisor()方法，表示拦截与Pointcut相匹配的方法。

（5）创建代理对象。

当调用代理对象的save()方法时，首先会被事务拦截器拦截，并执行其invoke()方法，invoke()方法中会继续执行invokeWithinTransaction()方法，如下：

protected Object invokeWithinTransaction(Method method, Class<?> targetClass, final InvocationCallback invocation)  
 throws Throwable {  
  
 // If the transaction attribute is null, the method is non-transactional.

// 获取事务属性  
 final TransactionAttribute txAttr = getTransactionAttributeSource().getTransactionAttribute(method, targetClass);

// 获取事务管理器  
 final PlatformTransactionManager tm = determineTransactionManager(txAttr);

// 获取连接点  
 final String joinpointIdentification = methodIdentification(method, targetClass, txAttr);  
  
 if (txAttr == null || !(tm instanceof CallbackPreferringPlatformTransactionManager)) {  
 // Standard transaction demarcation with getTransaction and commit/rollback calls.

// 根据事务属性，使用事务管理器创建事务(即调用事务管理器的getTransaction(TransactionDefition)方法创建事务)  
 TransactionInfo txInfo = createTransactionIfNecessary(tm, txAttr, joinpointIdentification);  
 Object retVal = null;  
 try {  
 // This is an around advice: Invoke the next interceptor in the chain.  
 // This will normally result in a target object being invoked.

// 继续执行下一个拦截器，并最终执行目标对象的目标方法

retVal = invocation.proceedWithInvocation();  
 }  
 catch (Throwable ex) {  
 // target invocation exception

// 如果发生异常，则使用事务管理器进行回滚  
 completeTransactionAfterThrowing(txInfo, ex);  
 throw ex;  
 }  
 finally {  
 cleanupTransactionInfo(txInfo);  
 }

// 如果没有发生异常，则使用事务管理器提交事务  
 commitTransactionAfterReturning(txInfo);  
 return retVal;  
 }

// 略  
}

接下来看一下声明式事务的三种配置方式。

第一种：使用<aop:config>和<tx:advice>标签配置声明式事务。

配置文件如下（接编程式事务配置文件）：

<!--\*\*\*\*\*\*\*\*\*使用aop:config和tx:advice标签来进行声明式事务\*\*\*\*\*\*\*\*\*\*\*\*\*-->  
<tx:advice id="txAdvice" transaction-manager="hibernateTransactionManager">  
 <tx:attributes>  
 <tx:method name="save" propagation="REQUIRED"/>  
 </tx:attributes>  
</tx:advice>  
  
<aop:config>  
 <aop:pointcut id="txPointcut" expression="execution(\* com.kevin.dao.TxAdviceUserDao.\*(..))"/>  
 <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut"/>  
</aop:config>

第二种：使用@Transactional注解配置声明式事务。

配置文件如下（接编程式事务配置文件）：

<!--\*\*\*\*\*\*\*\*\*使用@Transactional注解来进行声明式事务\*\*\*\*\*\*\*\*\*\*\*-->  
<tx:annotation-driven transaction-manager="hibernateTransactionManager"/>

第三种：使用TransactionProxyFactoryBean配置声明式事务。

<!--\*\*\*\*\*\*\*使用TransactionProxyFactoryBean来进行声明式事务-->  
<bean id="transactionProxyFactoryBeanUserDao" class="com.kevin.dao.TransactionProxyFactoryBeanUserDao">  
 <property name="hibernateTemplate" ref="hibernateTemplate"/>  
</bean>  
  
<bean id="transactionProxyFactoryBean" class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">  
 <!-- 配置事务管理器 -->  
 <property name="transactionManager" ref="hibernateTransactionManager"/>  
 <!-- 配置目标对象 -->  
 <property name="target" ref="transactionProxyFactoryBeanUserDao"/>  
 <!-- 配置事务属性 -->  
 <property name="transactionAttributes">  
 <props>  
 <prop key="\*">PROPAGATION\_REQUIRED</prop>  
 </props>  
 </property>  
</bean>

使用TransactionProxyFactoryBean配置声明式事务的缺点很明显：只能针对一个target进行代理，如果要代理多个target，则需要配置多个TransactionProxyFactoryBean，比较麻烦。

# 分布式事务概述

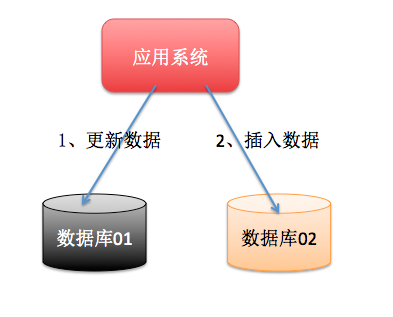
转载自：

http://www.cnblogs.com/zengkefu/p/5742617.html

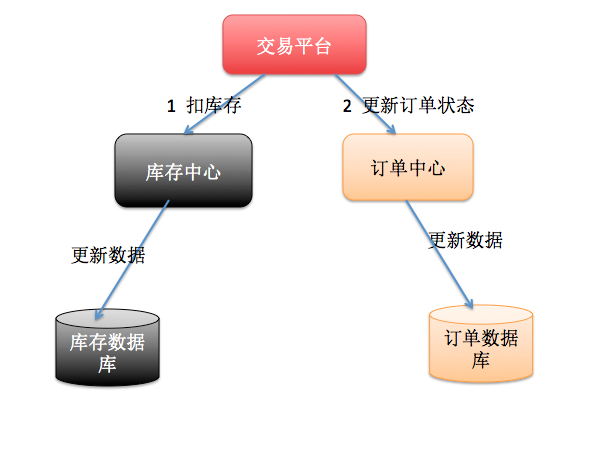
## 分布式事务产生的原因

有多种场景会产生分布式事务，这里举两个例子。

数据库分库分表：当数据库单表一年产生的数据超过1000W，那么就要考虑分库分表。这时候，如果一个操作既访问01库，又访问02库，而且要保证数据的一致性，那么就要用到分布式事务。



应用SOA化：所谓的SOA化，就是业务的服务化，比如原来单机支撑了整个电商网站，现在对整个网站进行拆分，分离出了订单中心、用户中心、库存中心。对于订单中心，有专门的数据库存储订单信息，用户中心也有专门的数据库存储用户信息，库存中心也会有专门的数据库存储库存信息。这时候如果要同时对订单和库存进行操作，那么就会涉及到订单数据库和库存数据库，为了保证数据一致性，就需要用到分布式事务。



以上两种情况虽然表象不同，但是本质相同，都是因为要操作的数据库变多了。

## 事务的ACID特性

原子性(Atomicity)：在整个事务中的所有操作，要么全部完成，要么全部不做，没有中间状态。如果事务在执行中发生错误，则所有的操作都会被回滚，整个事务就像从没被执行过一样。

一致性(Consistency)：事务的执行必须保证系统的一致性，以转账为例，A有500元，B有300元，如果在一个事务中，A成功转50元给B，那么不管并发多少，不管发生什么，只要事务执行成功了，那么最后A账户一定是450元，B账户一定是350元。

隔离性(Isolation)：事务与事务之间不会互相影响，一个事务的中间状态不会被其他事务感知。

持久性(Durability)：一旦事务完成了，那么事务对数据所做的变更就完全保存在了数据库中，即使发生停电或者系统宕机，也是如此。

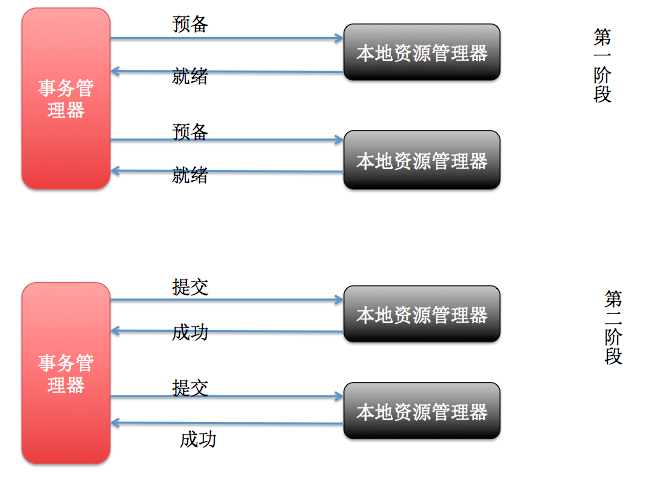
## 分布式事务的应用场景

支付：最经典的场景就是支付了。一笔支付，是对买家账户进行扣款，同时对卖家账户进行加钱，这些操作必须在一个事务里执行，要么全部成功，要么全部失败。买家账户属于买家中心，对应的是买家数据库；卖家账户属于卖家中心，对应的是卖家数据库，对不同数据库的操作必然需要引入分布式事务。

在线下单：买家在电商平台下单，往往会涉及到两个动作，一个是扣库存，一个是更新订单状态。库存和订单一般属于不同的数据库，需要使用分布式事务保证数据一致性。

## 常用的分布式事务解决方案

基于XA协议的两阶段提交：XA是一个分布式事务协议，由Tuxedo提出。XA协议指的是事务管理器和资源管理器之间进行通信的标准接口，其中资源管理器往往由数据库实现，比如Oracle、DB2等商业数据库都实现了XA接口；事务管理器作为全局的调度者，负责各个本地资源的提交和回滚。XA实现分布式事务的原理如下：



总的来说，XA协议比较简单，而且一旦商业数据库实现了XA协议，使用分布式事务的成本也会比较低。但是，XA也有致命的缺点，那就是性能不理想，特别是在交易下单链路，往往并发量很高，XA无法满足高并发场景。因此，总的来说，XA比较适合传统的单体应用，并且在同一个方法中存在跨库操作的情况，而不适合高并发和高性能要求的场景。

XA目前在商业数据库支持的比较理想，在MySQL数据库库中支持的不太理想，MySQL的XA实现没有记录Prepare阶段日志，主备切换会导致主库与备库数据不一致。许多NoSQL也没有支持XA，这让XA的应用场景变得非常狭隘。

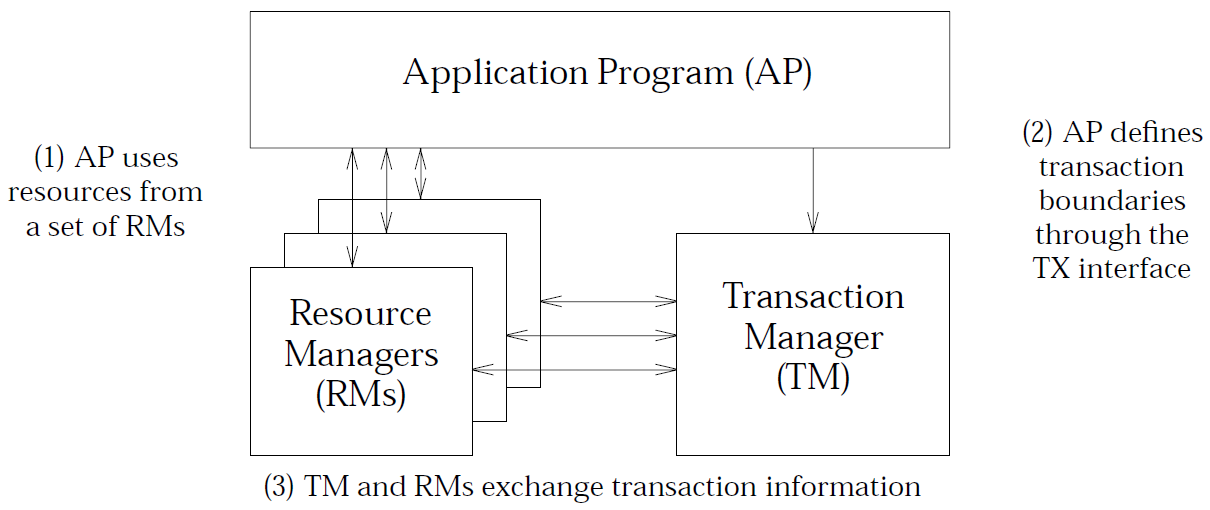
消息事务+最终一致性：所谓的消息事务就是基于消息中间件的两阶段提交，本质上是对消息中间件的一种特殊利用，它是将本地事务和发消息放到了一个分布式事务中，保证要么本地事务操作成功，并且对外发消息成功，要么两者都失败，开源的RocketMQ就支持这一特性。

# 分布式事务处理模型

## X/Open DTP模型

X/Open是1984年由多个欧洲的Unix系统制造商联合创建的一个用于确定和推广信息技术领域开放标准的联盟。

X/Open针对分布式事务提出了一个模型，即DTP模型，全称是Distributed Transaction Process，分布式事务处理模型。之前我们接触的事务都是针对单个数据库的操作，如果涉及到多个数据库的操作，同时又想保证原子性，就要用到分布式事务。X/Open DTP就是一种分布式事务处理模型，其模型图如下：



主要涉及到三个对象：

Application Programe(AP)：应用程序。

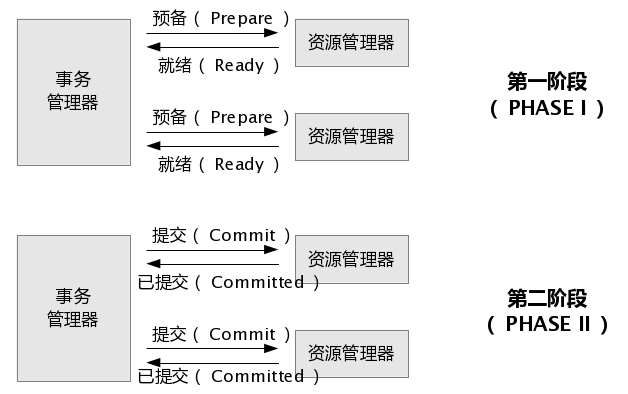
TrasanctionManager(TM)：事务管理器，负责协调和管理事务，不等同于Spring的事务管理器PlatformTransactionManager。

ResourceManager(RM)：资源管理器，可以理解为数据库或者JMS。

以上三个对象的关系是：AP可以通过TM来操作多个RM，也可以通过RM的本地事务来操作单个RM；TM和RM之间可以双向通信，其依赖的通信协议就是XA协议。

## XA协议

XA协议采用两阶段提交(two-phase commit)方式来管理分布式事务，。两阶段提交的基本流程如下图所示：



下图展示了当其中一个资源管理器在第一阶段轮询时发生错误的情况下，两阶段提交的过程。



第一个阶段：事务管理器通过XA协议，向资源管理器发送Prepare命令，询问它们预提交是否成功，每个资源管理器给出自己的响应，表示预提交成功或者失败。

第二个阶段：根据第一个阶段资源管理器的回复情况，如果全部预提交成功，则事务管理器向所有的资源管理器发送最终的提交命令，全部进行提交；如果有一个资源管理器回复预提交失败，则事务管理器向所有的资源管理器发送回滚命令，全部进行回滚。

通过这样的两阶段提交协议，即可完成分布式事务的功能。这种协议正是利用了事务管理器和资源管理器的双向通信能力，在之前的本地事务中，例如Jdbc事务，我们只能单向地通过Connection向数据库发送命令，而不能从数据库获取事务的一些执行信息。

利用两阶段提交协议实现分布式事务也是存在一些问题的：

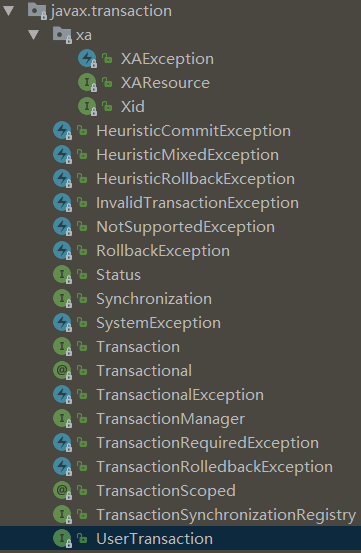
（1）两阶段提交协议是阻塞式协议，所以事务管理器必须等待每一个资源管理器发送回复之后，才能进行下一步操作。一旦资源管理器挂掉，则事务管理器接收不到响应，从而造成事务管理器一直处于等待状态，此时就必须引入超时机制，一旦超过某个时间没有收到回复，则认为失败，回滚所有操作。以上这些都是非常耗性能的。

（2）一旦事务管理器挂掉，则资源管理器一直等待来自事务管理器的命令，此时就需要使用备份的事务管理器来接替原来的事务管理器的工作。

## JTA

Java通过JTA(Java Transaction API)实现了X/Open提出的DTP规范。

JTA的包结构如下：



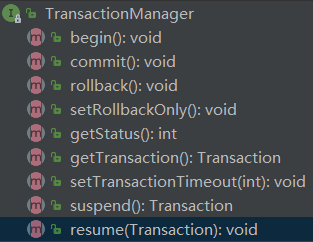
接下来我们分析一下JTA中的几个关键接口。

首先分析AP、TM、RM这三个对象。

AP是我们编写的应用程序，略过；TM是事务管理器，即javax.transaction.TransactoinManager；RM是资源管理器，例如数据库或者JMS，JTA提供了javax.transaction.XAResource，我们称之为与资源管理器的一个通信代表，我们通过XAResource接口方法与资源管理器进行通信。

接下来详细说明以上几个接口。

javax.transaction.TransactionManager表示事务管理器，接口定义如下：



部分接口方法说明如下：

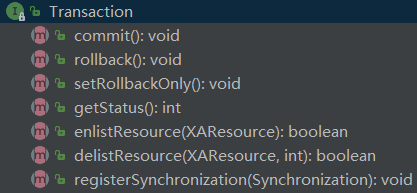
begin()：创建一个新事务，并将其关联到当前线程。

getTransaction()：获取与当前线程关联的事务。

commit()：提交与当前线程相关联的事务。

rollback()：回滚与当前线程相关联的事务。

javax.transaction.Transaction表示事务，接口定义如下：



commit()：提交事务。

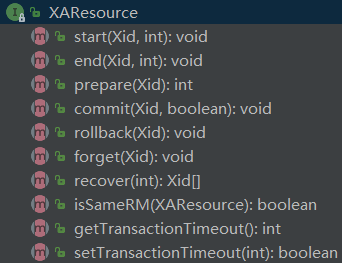
rollback()：回滚事务。

enlistResource(XAResource xaRes)：把给定的XAResource（资源管理器的一个通信代表）加入到当前事务中。一个分布式事务会与多个资源管理器通信，该操作就是把某个资源管理器的通信代表加入到当前事务中。

delistResource(XAResource xaRes)：把给定的XAResource（资源管理器的一个通信代表）从当前事务中移除。

javax.transaction.xa.Xid用来标识事务，我们可以把Xid想象成一个事务的标识符，每次创建新事务时，都会为事务分配一个Xid。Xid包含三个元素：formatId、globalTransactionId和branchQualifier，其中formatId用来指定事务族(transaction family)并被用来解释Xid中的另外两个字段，globalTransactionId用来表示全局事务的id号，branchQualifier用来指定全局事务具体分支。

javax.transaction.xa.XAResource是资源管理器RM的一个通信代表，我们通过XAResource与资源管理器通信，所有XAResource的实现类实现了两阶段提交协议。该接口的接口定义如下：



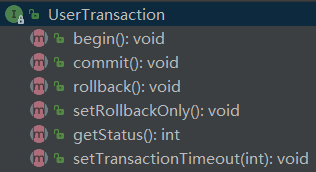
部分接口方法如下：

start(Xid xid, int flags)、end(Xid xid, int flags)：用来告知资源管理器的事务边界，即在这两个方法之间的sql等操作才会被纳入xid所对应的分布式事务中。

prepare(Xid xid)：两阶段提交协议中的第一个阶段，即针对xid对应的分布式事务，向资源管理器发送与预提交命令。

commit(Xid xid, boolean onePhase)、rollback(Xid xid)：如果所有资源管理器都回复OK，则向所有资源管理器发送commit提交事务，否则发送rollback回滚事务。

javax.transaction.UserTransaction接口是给开发人员使用的事务接口，屏蔽了底层的实现细节，通过该接口就可以操作一个分布式事务。UserTransaction接口的实现通常是委托给事务管理器来完成，接口定义如下：



可以看到，UserTransaction接口中的接口方法在事务管理器中都有对应的方法。

## JTA中涉及的角色

JTA中涉及到以下几种角色。

开发人员：只需通过使用UserTransaction接口来操作分布式事务。

TransactionManager接口的提供者：实现UserTransaction、TransactionManager、Transaction接口，通过与XAResource接口进行交互来实现分布式事务。TransactionManager接口的提供者有Jotm、Atomikos，相应的实现类如下：

（1）Jotm中UserTransaction接口的实现类是org.objectweb.jotm.Current；

（2）Jotm中TransactionManager接口的实现类是org.objectweb.jotm.Current；

（3）Jotm中Transaction接口的实现类是org.objectweb.jotm.TransactionImpl；

（4）Atomikos中UserTransaction接口的实现类是com.atomikos.icatch.jta.UserTransactionImp；

（5）Atomikos中TransactionManager接口的实现类是com.atomikos.icatch.jta.TransactionManagerImp；

（6）Atomikos中Transaction接口的实现类是com.atomikos.icatch.jta.TransactionImp。

XAResource接口的提供者：需要由资源管理器来实现，如果资源管理器是数据库，则数据库需提供XAResource接口的实现，例如MySQL数据库对应的实现类为com.mysql.jdbc.jdbc2.optional.MysqlXAConnection，这种形式下的数据库驱动就是支持分布式事务的数据库驱动。

更多的支持分布式事务的数据库驱动见链接：

https://www.atomikos.com/Documentation/ConfiguringTransactionsEssentials#Third\_Party\_JDBC\_XA\_Configuratio

## JTS

JTA定义了一套接口，其中约定了几种主要的角色：TransactionManager、Transaction、XAResource、UserTransaction，并定义了这些角色之间需要遵守的规范。

JTS(Java Transaction Service)也是一组规范，上面提到JTA需要角色之间的交互，那应该如何交互？JTS就是约定了交互细节的规范。

总体来说，JTA更多的是从框架的角度来约定程序角色的接口，而JTS则是从具体实现的角度来约定程序角色之间的接口，两者各司其职。

## Q&A

1、是不是数据库驱动必须支持XA协议，即支持分布式事务？

不是，可以使用第三方框架，如Jotm、Atomikos，来模拟XA协议。

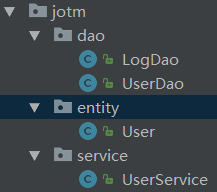
2、使用分布式事务，是不是一定要应用服务器的支持？

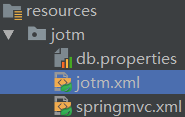
不是。JBoss是支持的，所以我们可以直接使用JBoss自己实现的UserTransaction来实现分布式事务（使用JNDI方式，此时就要求数据库驱动也要支持XA协议）；Tomcat是不支持的，但是我们可以使用第三方框架来实现分布式事务，比如Jotm、Atomikos。

# Jotm分布式案例

## 与Spring集成使用

项目结构如下：



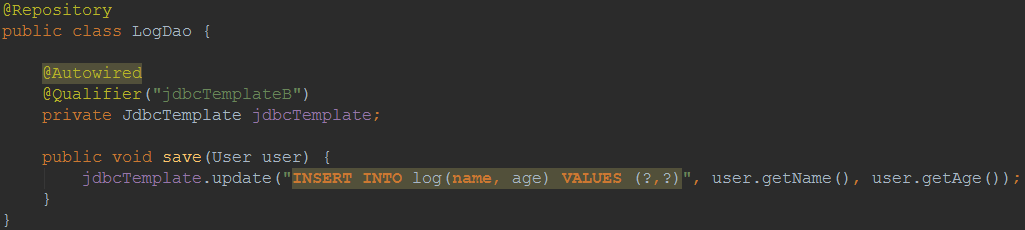


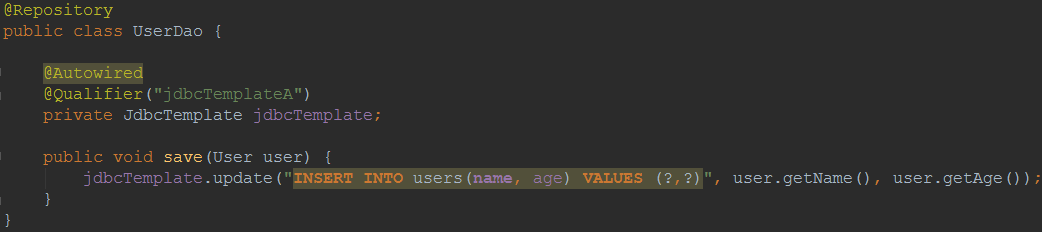
整个工程是使用jotm和XAPool来实现分布式事务的，jotm提供事务管理器javax.transaction.TransactionManager和面向开发人员的编程接口javax.transaction.UserTransaction，而XAPool则是对非XA数据库驱动进行包装来模拟XA数据库驱动，因此依赖的pom如下：

<dependency>  
 <groupId>org.ow2.jotm</groupId>  
 <artifactId>jotm-core</artifactId>  
 <version>${jotm.version}</version>  
</dependency>  
<dependency>  
 <groupId>org.ow2.jotm</groupId>  
 <artifactId>jotm-datasource</artifactId>  
 <version>${jotm.version}</version>  
</dependency>  
<!-- Jotm End -->  
  
<!-- XAPool Begin -->  
<dependency>  
 <groupId>com.experlog</groupId>  
 <artifactId>XAPool</artifactId>  
 <version>1.5.0</version>  
</dependency>  
<!-- XAPool End -->

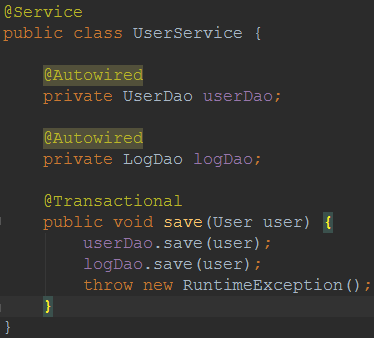
pom文件中的jotm-datasource主要是为了将StandardXAPoolDataSource数据源放置到web容器中，例如tomcat，而不是应用程序中。这样，应用程序可以通过JNDI的方式从web容器中取到上述数据源。对于本工程来讲，jotm-datasource可以不要，但如果想要通过JNDI的方式从web容器中获取数据源，则必须加上jotm-datasource。

LogDao和UserDao如下：





UserService如下：



配置文件jotm.xml中，关于数据源和JdbcTemplate的配置如下：

<!-- 配置第一个数据源 -->  
<bean id="dataSourceA" class="org.enhydra.jdbc.pool.StandardXAPoolDataSource" destroy-method="shutdown">  
 <property name="dataSource">  
 <bean class="org.enhydra.jdbc.standard.StandardXADataSource" destroy-method="shutdown">  
 <property name="transactionManager" ref="jotm"/>  
 <property name="driverName" value="${jdbc.driverName}"/>  
 <property name="url" value="${jdbc.url}"/>  
 </bean>  
 </property>  
 <property name="user" value="${jdbc.user}"/>  
 <property name="password" value="${jdbc.password}"/>  
</bean>  
  
<!-- 配置第二个数据源 -->  
<bean id="dataSourceB" class="org.enhydra.jdbc.pool.StandardXAPoolDataSource" destroy-method="shutdown">  
 <property name="dataSource">  
 <bean class="org.enhydra.jdbc.standard.StandardXADataSource" destroy-method="shutdown">  
 <property name="transactionManager" ref="jotm"/>  
 <property name="driverName" value="${jdbc.driverName}"/>  
 <property name="url" value="${jdbc.url2}"/>  
 </bean>  
 </property>  
 <property name="user" value="${jdbc.user}"/>  
 <property name="password" value="${jdbc.password}"/>  
</bean>  
  
<!-- 配置第一个JdbcTemplate -->  
<bean id="jdbcTemplateA" class="org.springframework.jdbc.core.JdbcTemplate">  
 <property name="dataSource" ref="dataSourceA"/>  
</bean>  
  
<!-- 配置第二个JdbcTemplate -->  
<bean id="jdbcTemplateB" class="org.springframework.jdbc.core.JdbcTemplate">  
 <property name="dataSource" ref="dataSourceB"/>  
</bean>

这里我们配置了两个数据源，对应两个不同的数据库。平常我们使用的数据源大都是c3p0的ComboPooledDataSource，此时需要换成可以模拟XA协议的StandardXAPoolDataSource，这个连接池是由com.experlog.XAPool提供的。

这里简单介绍一下XAPool。

（1）它设计了一个GenericPool，其中可以存放任何实现了PoolHelper接口的对象；

（2）它提供了DataSource的实现类，即StandardPoolDataSource；同时还提供了针对分布式的DataSource的实现类，即StandardXAPoolDataSource，该类可以通过普通的数据库驱动来模拟两阶段提交协议中XAResource的作用。另外，以上两个实现类均实现了DataSource接口，同时也实现了PoolHelper接口。

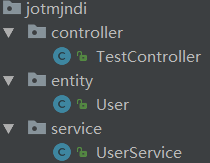
配置文件jotm.xml中，关于事务的配置如下：

<!-- 使用Jotm进行分布式事务处理 -->  
<bean id="jotm" class="org.objectweb.jotm.Current"/>  
  
<!-- 配置分布式事务管理器 -->  
<bean id="jtaTransactionManager" class="org.springframework.transaction.jta.JtaTransactionManager">  
 <property name="userTransaction" ref="jotm"/>  
</bean>  
  
<!-- 启用事务注解 -->  
<tx:annotation-driven transaction-manager="jtaTransactionManager"/>

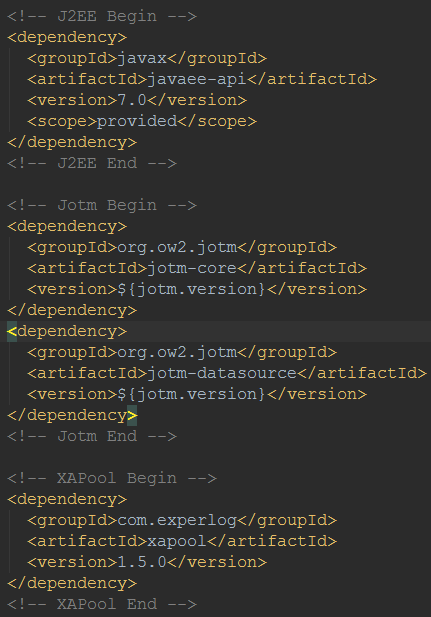
我们想使用@Transactional注解，因此需要配置tx:annotation-driven标签，该标签需要一个TransactionManager实例。由于这里是分布式事务，因此我们需要配置一个分布式事务管理器，即JtaTransactionManager，而JtaTransactionManager需要一个UserTransaction实例。另外，在使用StandardXAPoolDataSource时，需要为其配置一个TransacntionManager类实例。对于Jotm来说，TransactionManager和UserTransaction的实现类均为Current，因此在配置文件中配置该bean即可。

## 通过JNDI使用

项目结构图如下：



pom依赖如下：



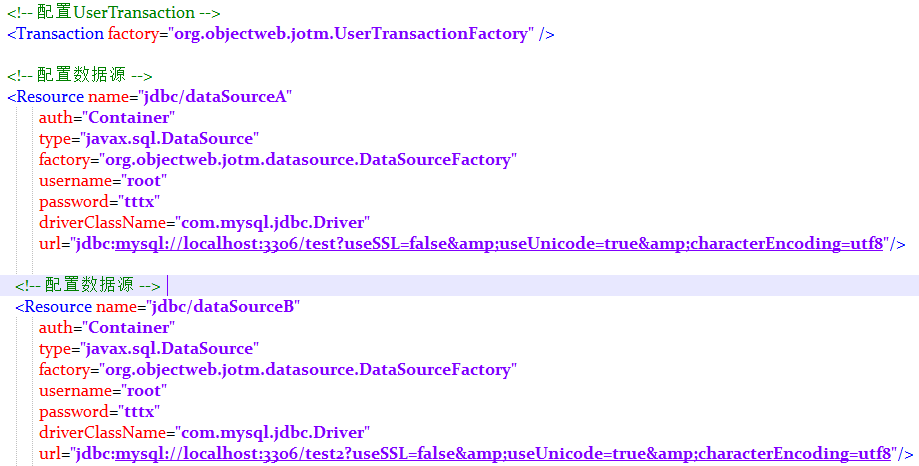
UserService.java文件如下：

@Service  
public class UserService {  
  
 public void save(User user) {  
 try {  
 Context ctx = new InitialContext();  
 DataSource dataSourceA = (DataSource) ctx.lookup("java:comp/env/jdbc/dataSourceA");  
 DataSource dataSourceB = (DataSource) ctx.lookup("java:comp/env/jdbc/dataSourceB");  
 UserTransaction userTransaction = (UserTransaction) ctx.lookup("java:comp/UserTransaction");  
 userTransaction.begin();  
  
 Connection connA = dataSourceA.getConnection();  
 Connection connB = dataSourceB.getConnection();  
 String sqlA = "insert into users(name, age) values(?, ?)";  
 String sqlB = "insert into log(name, age) values(?, ?)";  
 PreparedStatement preparedStatementA = connA.prepareStatement(sqlA);  
 PreparedStatement preparedStatementB = connB.prepareStatement(sqlB);  
 preparedStatementA.setString(1, user.getName());  
 preparedStatementA.setInt(2, user.getAge());  
 preparedStatementB.setString(1, user.getName());  
 preparedStatementB.setInt(2, user.getAge());  
 preparedStatementA.execute();  
 preparedStatementB.execute();  
  
 userTransaction.commit();  
 }

// 略

}  
}

UserTransaction和DataSource都是在tomcat的conf/context.xml文件中配置的，如下：



其中<Transaction>元素的作用是配置一个用于实例化UserTransaction对象的资源工厂，实例化的UserTransaction对象可以通过java:comp/UserTransaction获取到。也就是说，这个配置将"java:comp/UserTransaction"和上述的UserTransactionFactory生成的UserTransaction实例关联了起来，我们可以通过如下方式获取该UserTransaction实例：

UserTransaction userTransaction = (UserTransaction) ctx.lookup("java:comp/UserTransaction");

我们来看下UserTransactionFactory生成的UserTransaction实例的具体形式，如下；

public Object getObjectInstance(Object objref, Name name, Context ctx, Hashtable env) throws Exception {  
 if (TraceTm.*jta*.isDebugEnabled()) {  
 TraceTm.*jta*.debug("UserTransactionFactory.getObjectInstance");  
 }  
   
 Reference ref = (Reference) objref;  
 Current ut = null;  
  
 if (ref.getClassName().equals("javax.transaction.UserTransaction")  
 || ref.getClassName().equals("org.objectweb.jotm.Current")) {  
 // create the UserTransaction object  
 // No need to init a TMFactory in the client!  
 // ut = Current.getInstance();  
 ut = Current.*getCurrent*();  
  
 if (ut == null) {  
 ut = new Current();  
  
 // Get the timeout default value that was configured in the server  
 String timeoutStr = (String) ref.get("jotm.timeout").getContent();  
 Integer i = new Integer(timeoutStr);  
 int timeout = i.intValue();  
 ut.setDefaultTimeout(timeout);  
 }  
 }  
   
 if (TraceTm.*jta*.isDebugEnabled()) {  
 TraceTm.*jta*.debug("UserTransactionFactory.getObjectInstance ut= " + ut);  
 }  
 return ut;  
}

可以看到，UserTransactionFactory提供的UserTransaction实例是org.objectweb.jotm.Current。

<Resource>元素的作用是配置一个资源以供应用使用，相当于在web.xml文件中配置的<resource-ref>元素。也就是说，这个配置将"java:comp/env/xxx"和上述的DataSourceFactory生成的DataSource实例关联了起来，我们可以通过如下方式获取该DataSource实例：

DataSource dataSourceA = (DataSource) ctx.lookup("java:comp/env/jdbc/dataSourceA");

我们来看下DataSourceFactory生成的DataSource实例的具体形式，如下；

public Object getObjectInstance(Object obj, Name n, Context nameCtx, Hashtable environment)  
 throws Exception {  
 StandardXAPoolDataSource xads = null;  
 StandardXADataSource ds = null;  
 try {  
 Reference ref = (Reference) obj;  
 ds = new StandardXADataSource();  
 xads = new StandardXAPoolDataSource(ds);  
 Enumeration addrs = ref.getAll();  
 while (addrs.hasMoreElements()) {  
 RefAddr addr = (RefAddr) addrs.nextElement();  
 String name = addr.getType();  
 String value = (String) addr.getContent();  
 if (name.equals("driverClassName")) {  
 ds.setDriverName(value);  
 } else if (name.equals("url")) {  
 ds.setUrl(value);  
 } else if (name.equals("username")) {  
 xads.user = value;  
 ds.setUser(value);  
 } else if (name.equals("password")) {  
 ds.setPassword(value);  
 xads.password = value;  
 }

// 略  
 }  
 xads.setTransactionManager(*jotm*.getTransactionManager());  
 xads.setDataSource(ds);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 if (*table*.containsKey(ds)) {  
 return *table*.get(ds);  
 } else {  
 *table*.put(ds, xads);  
 return xads;  
 }  
}

可以看到，这里也是创建一个StandardXADataSource实例和一个StandardXAPoolDataSource实例，然后为其设置参数，比如数据库驱动、用户名、密码、连接url以及TransactionManager属性。

需要注意的是，在为StandardXAPoolDataSource实例设置TransactionManager属性时，即：

xads.setTransactionManager(*jotm*.getTransactionManager());

如果使用的是jotm-core-2.3.1-M1.jar，则会出现一个bug。这个bug是由于新建jotm实例时导致的，我们看下DataSourceFactory中相应的源码，如下：

public class DataSourceFactory implements ObjectFactory {  
  
 private static Hashtable *table* = new Hashtable();  
  
 static {  
 try {  
 *jotm* = new Jotm(true, false);  
 } catch (NamingException e) {  
 e.printStackTrace();  
 }  
 }  
  
 public static Jotm *jotm*;

可以看到，在加载DataSourceFactory时，就会调用new Jotm(true, false)创建该jotm实例。在jotm-core-2.3.1-M1.jar中，Jotm类的构造函数如下：

public Jotm(boolean local, boolean bound) throws NamingException {  
 this(local, bound, null);  
}

可以看到，第三个参数为null，继续追源码：

public Jotm(boolean local, boolean bound, RmiConfiguration rmiConfiguration) throws NamingException {  
 this.local = local;  
 this.bound = bound;  
 this.rmiConfiguration = rmiConfiguration;  
  
 // RMI initialization  
 *log*.info("RMI initialization");  
 try {  
 rmiConfiguration.init();  
 } catch (RmiConfigurationException e) {  
 *log*.error("RMI initialization failed", e);  
 }

可以看到，第三个参数为null时，会抛出一个空指针异常，而这个异常并没有被catch语句捕获到，因此会继续向上层传递，而DataSourceFactory的catch语句中也没有捕获该异常，最终导致DataSourceFactory类加载失败。

因此，解决的方案是换成低版本的jotm-core，比如2.2.3就可以。但需要注意的是，如果是与Spring集成，则应当使用2.3.1-M1版本的Jotm，否则会报错：org.omg.CORBA.BAD\_INV\_ORDER: The Servant has not been associated with an ORB instance，这应该是旧版的一个bug。

# Jotm分布式事务源码解析

## XAPool中相关的类和接口

在分析XAPool中相关的类和接口之前，我们先分析以下javax.sql.DataSource接口。

### DataSource接口

JDBC1.0使用DriverManager类来获取一个连接，JDBC2.0则使用DataSource替代DriverManager来获取连接。

一个DataSource对象代表了一个真正的数据源，根据DataSource的实现方式不同，数据源既可以是一个关系型数据库，也可以是一个电子表格，还可以是一个表格形式的文件。DataSource对象的成员属性中包含了定位数据源所需的信息，比如数据库的名称、端口号等等，这样，应用开发者就不需要把驱动的名称硬编码到程序中，如果数据源需要移植到另一个数据库驱动中，则只需要修改DataSource对象的相应属性即可，而使用DataSource对象的代码不需要做任何修改。

通常，由系统管理员或者有相应权限的人来设置DataSource对象的成员属性，并把它注册到JNDI名字服务中。一旦配置好了DataSource，应用开发者就可以用它来产生一个到数据源的连接，下面的代码片段展示了如何通过JNDI上下文获得一个数据源对象，然后通过该数据源对象获取一个与数据源的连接。

Context ctx = new InitialContext();

DataSource ds = (DataSource)ctx.lookup("jdbc/dataSourceA");

Connection conn = ds.getConnection();

DataSource对象是由驱动供应商实现的，有如下三种实现方式：

（1）基本实现，即提供一个标准的Connection对象。

（2）连接池实现，即产生的连接会自动参与到连接池中。连接池是这么一种机制，当应用程序关闭一个Connection时，该连接会被回收，而不是被销毁。因为建立一个连接是一个很费资源的操作，如果能把回收的连接重新利用，就可以减少新连接创建的次数，从而显著的提高程序性能。通过DataSource对象的geteConnection()方法返回的连接是否是一个连接池中的连接，完全取决于DataSource对象的实现方式，如果DataSource对象实现与一个中间层的连接池管理器一起工作，则该DataSource对象就会自动返回连接池中的连接。

（3）分布式事务实现，即产生的连接可能会被用在分布式事务中，并且几乎总是会参与到连接池中。同样地，通过DataSource对象的getConnection()方法返回的连接是否支持分布式事务，完全取决于DataSource对象的实现方式，如果DataSource对象实现与一个中间层的事务管理器一起工作，则该DataSource对象返回的连接就支持分布式事务。

在配置支持连接池的DataSource时，会涉及到配置ConnectionPoolDataSource对象，该对象是中间层用来管理连接池的对象。ConnectionPoolDataSource提供的连接为PooledConnection，代表了一个与数据源的物理连接，但该连接可以被重复使用。

public interface ConnectionPoolDataSource extends CommonDataSource {  
PooledConnection getPooledConnection() throws SQLException;  
PooledConnection getPooledConnection(String user, String password)  
 throws SQLException;  
 }

public interface PooledConnection {  
Connection getConnection() throws SQLException;

// 略  
 }

同样地，在配置支持分布式事务的DataSource时，会设置到配置XADataSource，该对象是中间层用来管理分布式事务的对象。XADataSource提供的连接为XAConnection，用来为分布式事务提供支持。

public interface XADataSource extends CommonDataSource {  
XAConnection getXAConnection() throws SQLException;  
XAConnection getXAConnection(String user, String password)  
 throws SQLException;  
 }

public interface XAConnection extends PooledConnection {  
javax.transaction.xa.XAResource getXAResource() throws SQLException;  
  
 }

可以看到，通过XAConnection不仅可以获取到与数据源的连接(PooledConnection的功能)，还可以获得数据源的一个通信代表XAResource。

### XA数据库驱动

XA数据库驱动就是实现了XADataSource接口的数据库驱动，这样我们就可以获得XAConnection，从而既可以获取Connection，也可以获取XAResource，实现与数据库的双向交互，进而实现两阶段提交协议。

目前，大部分数据库都是支持XA驱动的，例如MySQL提供的XADataSource实现类是com.mysql.jdbc.jdbc2.optional.MysqlXADataSource，Oracle提供的XADataSource实现类是oracle.jdbc.xa.client.OracleXADataSource。

按道理上讲，XA数据库驱动应该由数据库来提供，而XAPool就是在数据库不支持XA驱动的前提下模拟XA驱动。

接下来我们分析以下XAPool中提供的几个主要类和接口。

（1）org.enhydra.jdbc.pool.GenericPool：该类实现了一个pool的功能，可以存储和管理任何实现了PoolHelper接口的对象。看下GenericPool类的构造函数：

public GenericPool(PoolHelper helper) {  
 this(  
 helper,  
 *DEFAULT\_MINSIZE*,  
 *DEFAULT\_MAXSIZE*,  
 *DEFAULT\_EXPIRATION*,  
 *DEFAULT\_SLEEPTIME*);  
}

可以看到，实例化GenericPool类时，需要提供一个PoolHelper实例和一些参数设置，例如pool中object个数的最大值、最小值等。

看下PoolHelper接口定义，如下：

public interface PoolHelper {  
  
 public void expire(Object o); // object specific work to kill the object  
 public boolean checkThisObject(Object o);  
 // check if the object is still valid  
 public boolean testThisObject(Object o); // check if the object is closed  
 public GenerationObject create() throws SQLException;  
 public GenerationObject create(String \_user, String \_password)  
 throws SQLException;  
 public String toString();  
}

可以看到，GenericPool通过PoolHelper对外暴露出一些方法，例如通过PoolHelper的create()方法创建一个对象并将其存放到pool中，通过PoolHelper的expire()方法将一个object从pool中删除。

（2）org.enhydra.jdbc.standard.StandardDataSource：StandardDataSource类实现了javax.sql.DataSource接口，因此能够获取Connection，这个实现比较简单，就是利用DrvierManager来获取，如下：

ret = DriverManager.*getConnection*(url, prop);

（3）org.enhydra.jdbc.standard.StandardXADataSource：StandardXADataSource类实现了javax.sql.XADataSource接口，因此能够获取XAConnection，进而通过XAConnection获取到XAResource。

（4）org.enhydra.jdbc.pool.StandardPoolDataSource：StandardPoolDataSource类实现了PoolHelper接口，并且其内部有一个GenericPool pool成员属性用于管理某种类型的对象。

我们来看下StandardPoolDataSource类将什么对象交给pool来管理了，即看下StandardPoolDataSource是如何实现PoolHelper接口的create()方法，如下：

public GenerationObject create(String \_user, String \_password)  
 throws SQLException {  
 log.debug(  
 "StandardPoolDataSource:create create a connection for the pool");  
 GenerationObject genObject;  
 PooledConnection pooledCon = cpds.getPooledConnection(\_user, \_password);  
 // get the pooled connection  
  
 pooledCon.addConnectionEventListener(this);  
 // add it to the event listener  
 log.debug("StandardPoolDataSource:create create a object for the pool");  
 genObject =  
 new GenerationObject(  
 pooledCon,  
 pool.getGeneration(),  
 \_user,  
 \_password);  
 return genObject; // return a connection  
}

可以看到，StandardPoolDataSource首先通过ConnectionPoolDataSource cpds成员属性获取一个PooledConnection实例，然后将该PooledConection放到pool中进行管理。

（5）org.enhydra.jdbc.pool.StandardXAPoolDataSource：StandardXAPoolDataSource类继承自StandaardPoolDataSource类，因此其内部也拥有一个GenericPool pool成员属性用于管理某种类型的对象。我们来看下StandardXAPoolDataSource类将什么对象交给pool来管理了，即看下StandardXAPoolDataSource是如何实现PoolHelper接口的create()方法，如下：

public GenerationObject create(String \_user, String \_password)  
 throws SQLException {  
 GenerationObject genObject;  
 XAConnection xaCon = xads.getXAConnection(\_user, \_password);  
 // get the xa connection  
 xaCon.addConnectionEventListener(this); // add it to the event listener  
 log.debug(  
 "StandardXAPoolDataSource:create create a object for the pool");  
 genObject =  
 new GenerationObject(xaCon, pool.getGeneration(), \_user, \_password);  
  
 return genObject;  
}

可以看到，StandardXAPoolDataSource首先通过XADataSource xads成员属性获取一个XAConnection实例，具体一点就是获取一个StandardXAConnection实例，然后将该StandardXAConnection放到pool中进行管理。

StandardXAConnection类的定义如下：

public class StandardXAConnection  
 extends StandardPooledConnection  
 implements XAConnection, XAResource, Referenceable, Runnable {

可以看到，StandardXAConnection既是一个PooledConnection，又是一个XAConnection。

## Jotm中相关的类和接口

Jotm需要实现的JTA接口有以下几个。

javax.transaction.UserTransaction：面向开发者的接口，Jotm的实现类是org.objectweb.jotm.Current。

javax.transaction.TransactionManager：事务管理器接口，Jotm的实现类是org.objectweb.jotm.Current。

javax.transaction.Transaction：事务接口，Jotm的实现类是org.objectweb.jotm.TransactionImpl。

javax.transaction.xa.Xid：全局事务的唯一标识，Jotm的实现类是org.objectweb.jotm.XidImpl。

接下来分析Jotm是如何通过UserTransaction来开启事务，并与资源管理器进行交互的。

在分析UserTransaction之前，我们先分析一下org.objectweb.jotm.TransactionImpl类，因为分析源码可知，通过UserTransaction来开启事务并与资源管理器进行交互，其内部调用的都是TransactionImpl类中的相应方法来完成的。

### org.objectweb.jotm.TransactionImpl

TransactionImpl类几个重要的成员属性，如下：

private List<XAResource> enlistedXARes = Collections.*synchronizedList*(new ArrayList<XAResource>());

private List<XAResource> delistedXARes = null;

private List<javax.transaction.xa.Xid> enlistedJavaxXid =  
 Collections.*synchronizedList*(new ArrayList<javax.transaction.xa.Xid>());

private SubCoordinator subcoord = null;

其中enlistedXARes是enlisted XAResource列表，delistedXARes是suspended XAResource列表，enlistedJavaXid是enlistedXARes中各个XAResource对应的xid，这意味着xid和XAResource是通过位置关联起来的。TransactionImpl类内部有一个SubCoordinator类型的成员属性subcoord，这是两阶段提交过程中协调者，其接口定义如下：

public class SubCoordinator extends PortableRemoteObject implements Resource {

private TransactionImpl tx = null;  
  
private Vector resourceList = new Vector();  
 private Vector javaxxidList = new Vector();

其中，resourceList是enlisted XAResource列表，javaxxidList是resourceList中各个XAResource对应的xid，即xid和XAResource是通过位置关联起来的。也就是说，TransactionImpl类中保存了一份XAResource列表以及对应的Xid列表，SubCoordinator类中也保存了一份。

TransactionImpl类中提供了prepare()、commit()、rollback()等方法来执行两段提交协议，这些方法内部最终都会调用TransactionImpl类中SubCoordiante类型的subcoord属性中对应的方法来执行真正的操作，所以我们分析TransactionImpl是如何进行prepare、commit或者rollback，就是分析subcoord属性中对应的方法是如何实现的。

我们看下SubCoordinator实现的org.objectweb.jotm.Resource接口定义，如下：

public interface Resource extends Remote {  
  
 */\*\*   
 \* phase 1 of the 2PC.  
 \*  
 \** ***@return*** *int vote commit, rollback, or readonly.  
 \*/* public int prepare() throws RemoteException;  
 public final static int *VOTE\_COMMIT* = 0;  
 public final static int *VOTE\_ROLLBACK* = 1;  
 public final static int *VOTE\_READONLY* = 2;  
  
 */\*\*   
 \* rollback transaction  
 \*/* public void rollback() throws RemoteException;  
  
 */\*\*   
 \* phase 2 of the 2PC.  
 \*/* public void commit() throws RemoteException;  
  
 */\*\*   
 \* commit 1 phase.  
 \*/* public void commit\_one\_phase() throws RemoteException;  
  
 */\*\*   
 \* forget heuristics about this transaction.  
 \*/* public void forget() throws RemoteException;  
}

可以看到，Resource接口就是针对两阶段提交协议来定义的，先用prepare()方法进行投票，根据返回值结果来选择下一步是执行commit还是rollback亦或者是forget。

我们继续分析SubCoordinate类是如何实现以上接口方法的。

int prepare()方法的执行流程如下：

（1）将所有的XAResource结束事务边界，即分别调用各个XAResource的end()方法结束事务边界。具体过程如下：

/\*\*\* doDetach()方法 \*\*\*/

// 将enlistedXARes复制到delistedXARes中，然后依次delist各个XAResource

delistedXARes = new ArrayList<XAResource>(enlistedXARes);  
for (XAResource xar : delistedXARes) {  
 delistResource(xar, flag);  
}

/\*\*\* delistResource()方法 \*\*\*/

// 结束事务边界

xares.end (myjavaxxid, flag);

// 从enlistedXARes中移除该XAResource，从enlistedJavaxXid中移除相应的Xid  
enlistedXARes.remove(xares);  
enlistedJavaxXid.remove(javaxxid);  
return true;

（2）遍历SubCoordinate中的所有XAResource，然后对每个XAResource进行预提交。具体过程如下：

/\*\*\* doPrepare()方法 \*\*\*/

int ret = *VOTE\_READONLY*;  
int errors = 0;

for (int i = 0; i < resourceList.size(); i++) {  
 XAResource res = (XAResource) resourceList.elementAt(i);  
 javax.transaction.xa.Xid myjavaxxid = (javax.transaction.xa.Xid) javaxxidList.elementAt(i);

if (errors > 0) {

res.rollback(myjavaxxid);

} else {  
 try {  
 switch (res.prepare(myjavaxxid)) {  
 case XAResource.*XA\_OK* :   
 ret = *VOTE\_COMMIT*;   
 break;  
 case XAResource.*XA\_RDONLY* :   
 break;  
 }  
 } catch (XAException e) {  
 ret = *VOTE\_ROLLBACK*;  
 errors++;  
 }  
 }  
}

由于doPrepare()方法是synchronized方法，因此效率较低。

void rollback()方法的执行流程如下：

（1）同prepare()方法一样，也是将所有的XAResource结束事务边界，即分别调用各个XAResource的end()方法结束事务边界。

（2）遍历SubCoordinate中的所有XAResource，然后对每个XAResource进行回滚。

void commit()方法也是类似，遍历SubCoordinate中的所有XAResource，然后对每个XAResource进行提交。

分析完SubCoordinate类中的几个主要方法之后，我们回到TransactionImpl类。

TransactionImpl类还提供了一个enlistResource()方法，将资源管理器的通信代表XAResource加入到当前事务中并开启XAResource的事务边界。该方法的执行流程如下；

（1）将XAResource加入到SubCoordinate的resourceList中，即：

found = subcoord.addResource(xares);

（2）创建该XAResource对应的xid，并把该xid加入到SubCoordinate的javaxxidList中，即：

Xid resXid = new XidImpl( getXid(),subcoord.getXaresIndex(xares) );  
javax.transaction.xa.Xid javaxxid = new JavaXidImpl(resXid);

subcoord.addJavaxXid(javaxxid);

（3）开启XAResource的事务边界，即：

xares.start (javaxxid, flag);

（4）把该XAResource和xid也存放到TransactionImpl中，即：

enlistedXARes.add(xares);  
enlistedJavaxXid.add(javaxxid);

### org.objectweb.jotm.Current

分析完TransactionImpl后，我们再来分析Current类中的相应方法。

void begin()方法用于开启一个事务，其执行流程如下：

（1）创建一个全局事务的唯一标识xid。

XidImpl otid = new XidImpl();

（2）利用xid创建一个事务tx。

tx = new TransactionImpl(otid, transactionTimeout);

（3）将事务tx绑定到当前线程。

*threadTx*.set(tx);

这里的threadTx是Current类的一个ThreadLocal类型的成员属性，其定义如下：

private static transient ThreadLocal<TransactionImpl> *threadTx* = new ThreadLocal<TransactionImpl>();

（4）将事务tx和事务标识xid通过一个映射关联起来。

putTxXid(otid, tx);

这里会将映射关系放到Current类的一个Map类型的成员属性中，其定义如下：

private transient static Map<Xid, TransactionImpl> *txXids* = new HashMap<Xid, TransactionImpl>();

void commit()方法用于提交一个事务，其执行流程如下：

（1）从当前线程中获取绑定的事务，即TransactionImpl，然后依靠该TransactionImpl进行事务提交，而TransactionImpl实质上是依靠其内部的SubCoordinate类型的subcoord属性来进行事务提交。

（2）清除与当前线程绑定的事务。

void rollback()方法用于回滚一个事务，其执行流程与commit()方法类似，也是通过TransactionImpl来回滚事务，然后清除与当前线程绑定的事务。

## Jotm分布式事务流程解析

@Transactional  
 public void save(User user) {  
 userDao.save(user);  
 logDao.save(user);  
// throw new RuntimeException();  
 }

（1）因为我们使用了@Transactional注解，因此Spring会针对UserService创建一个代理对象，该代理对象中加入了事务拦截器。

（2）在执行save()方法之前，会创建一个事务管理器，这里就是JtaTransactionManager，然后通过事务管理器获取一个事务对象，这里就是JtaTransactionObject，该类的类定义如下：

public class JtaTransactionObject implements SmartTransactionObject {  
  
 private final UserTransaction userTransaction;

可以看到，JtaTransactionObject中有一个UserTransaction类型的成员属性，这个是我们在配置文件中配置的，即Jotm的Current类实例。

接下来会调用UserTransaction的begin()方法开启一个事务，并创建一个Xid对该事务进行标识，最后将该事务绑定到当前线程，具体的执行流程在讲解org.objectweb.jotm.Current类时已经详细解析过。

（3）使用JdbcTemplate执行业务逻辑时，即执行以下方法时：

public void save(User user) {  
 jdbcTemplate.update("INSERT INTO users(name, age) VALUES (?,?)", user.getName(), user.getAge());  
}

JdbcTemplate会从dataSource中获取一个Connection来执行sql语句，即调用StandardXAPoolDataSource的父类StandardPoolDataSource类中的getConnection()方法来获取一个连接，具体流程如下；

I. 调用pool.start()方法初始化连接池。在初始化连接池时，会调用StandardXAPoolDataSource.create()方法创建一定数量的StandardXAConnection对象，并把这些StandardXAConnection放到连接池中进行管理。

II. 调用pool.checkOut()方法从连接池中获取一个StandardXAConnection。

III. 将获取到的StandardXAConnection封装为StandardXAConnectionHandle对象并返回。

IV. 获取到连接之后，在执行sql语句之前，JdbcTemplate会创建一个PrepareStatement对象。这里是通过调用StandardXAConnectionHandle类的prepareStatement()方法完成的，如下：

try {  
 Transaction ntx = this.getTransaction();  
 if (ntx != null) {  
 log.debug(  
 "StandardXAConnectionHandle:prepareStatement (found a transaction)");  
 tx = ntx;  
 xacon.thisAutoCommit = this.getAutoCommit();  
 if (this.getAutoCommit()) {  
 this.setAutoCommit(false);  
 }  
 try {  
 tx.enlistResource(xacon.getXAResource());  
 // enlist the xaResource in the transaction  
 } catch (RollbackException n) {  
 log.debug(  
 "StandardXAConnectionHandle:prepareStatemnet enlistResource exception : "  
 + n.toString());  
 }  
 } else {  
 log.debug(  
 "StandardXAConnectionHandle:prepareStatement (no transaction found)");  
 }  
}

可以看到，在执行sql语句之前，会先获取与当前线程绑定的事务，然后设置StandardXAConnection底层的物理连接为非自动提交，最后获取StandardXAConnection对应的XAResource，并调用TransactionImpl类的enlistResource()方法将其加入到当前事务中并开启XAResoure的事务边界。enlistResource()方法的具体执行流程在前面讲解org.objectweb.jotm.TransactionImpl时已经分析过。

（4）调用Current类的commit()方法进行两阶段提交事务，该方法会获取与当前线程绑定的TransactionImpl进行事务提交（即调用TransactionImpl的commit()方法）。TransactionImpl会委托给内部的SubCoordinate（即调用SubCoordinate的commit\_one\_phase()方法）， SubCoordinate会针对resourceList中的每个XAResource进行预提交（即调用doPrepare()方法），如果预提交结果都是VOTE\_COMMIT，则开始真正的提交事务（即调用doCommit()方法）。

（5）一旦执行过程中抛出异常，则事务管理器JtaTransactionManager会调用其内部的事务对象JtaTransactionObject中的UserTransaction（这里是Current）的rollback()方法进行回滚，该方法会获取与当前线程绑定的TransactionImpl进行事务回滚（即调用TransactionImpl的rollback()方法）。TransactionImpl会委托给内部的SubCoordinate（即调用SubCoordinate的rollback()方法）， SubCoordinate会针对resourceList中的每个XAResource进行回滚（即调用doRollback()方法）。

## 模拟Jotm分布式事务

我们可以简单模拟一下Jotm分布式事务，如下：

Connection connA = null;  
Connection connB = null;  
PreparedStatement psA = null;  
PreparedStatement psB = null;  
try {  
 connA = dataSourceA.getConnection();  
 connB = dataSourceB.getConnection();  
 String sqlA = "insert into users(name, age) values(?,?)";  
 String sqlB = "insert into log(name, age) values(?,?)";  
 psA = connA.prepareStatement(sqlA);  
 psB = connA.prepareStatement(sqlB);  
 connA.setAutoCommit(false);  
 connB.setAutoCommit(false);  
 psA.execute();  
 psB.execute();  
 connA.commit();  
 connB.commit();  
} catch (SQLException e) {  
 e.printStackTrace();  
 try {  
 connA.rollback();  
 connB.rollback();  
 } catch (SQLException e1) {  
 e1.printStackTrace();  
 }  
} finally {  
 try {  
 connA.close();  
 connB.close();  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
}

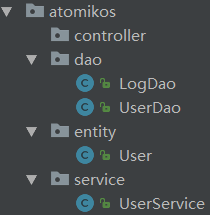
我们把所有的Connection的自动提交都设置为false，一旦执行过程中发生异常，就调用每个Connection的回滚方法；如果没异常，则全部提交，从而实现分布式事务。

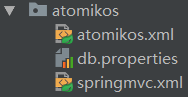
Jotm也是同样的思路，在使用JdbcTemplate执行业务逻辑操作时，就会把使用的Connection的自动提交设置为false，同时把这个Connection交给事务管理，一旦抛出异常，事务就会把它拥有的所有Connection全部回滚。

# Atomikos分布式案例

## 使用非XA数据库驱动

项目结构如下：

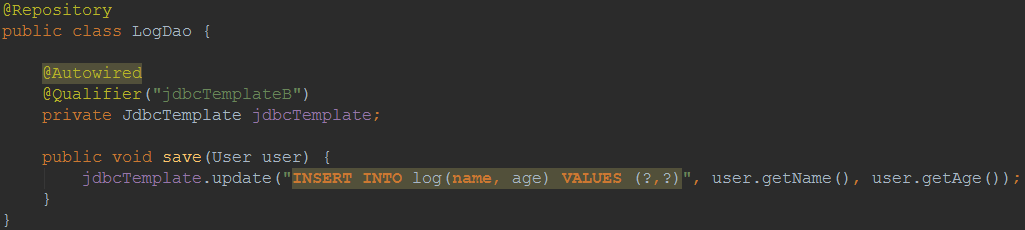


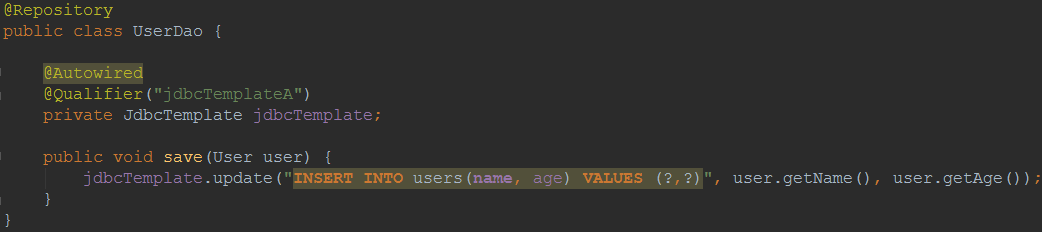


该工程是使用Atomikos来实现分布式事务的，因此依赖的pom如下：

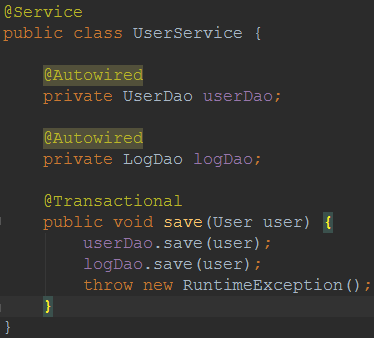
<!-- Atomikos Begin -->  
<dependency>  
 <groupId>com.atomikos</groupId>  
 <artifactId>transactions-jdbc</artifactId>  
 <version>4.0.4</version>  
</dependency>  
<!-- Atomikos End -->

LogDao和UserDao如下：





UserService如下：



配置文件atomikos.xml中，关于数据源和JdbcTemplate的配置如下：

<!-- 配置第一个数据源 -->  
<bean id="dataSourceA" class="com.atomikos.jdbc.nonxa.AtomikosNonXADataSourceBean">  
 <property name="uniqueResourceName" value="XA1DBMS"/>  
 <property name="driverClassName" value="${jdbc.driverName}"/>  
 <property name="user" value="${jdbc.user}"/>  
 <property name="password" value="${jdbc.password}"/>  
 <property name="url" value="${jdbc.url}"/>  
 <property name="poolSize" value="${jdbc.poolSize}"/>  
 <property name="minPoolSize" value="${jdbc.minPoolSize}"/>  
 <property name="maxPoolSize" value="${jdbc.maxPoolSize}"/>  
</bean>  
  
<!-- 配置第二个数据源 -->  
<bean id="dataSourceB" class="com.atomikos.jdbc.nonxa.AtomikosNonXADataSourceBean">  
 <property name="uniqueResourceName" value="XA2DBMS"/>  
 <property name="driverClassName" value="${jdbc.driverName}"/>  
 <property name="user" value="${jdbc.user}"/>  
 <property name="password" value="${jdbc.password}"/>  
 <property name="url" value="${jdbc.url2}"/>  
 <property name="poolSize" value="${jdbc.poolSize}"/>  
 <property name="minPoolSize" value="${jdbc.minPoolSize}"/>  
 <property name="maxPoolSize" value="${jdbc.maxPoolSize}"/>  
</bean>  
  
<!-- 配置第一个JdbcTemplate -->  
<bean id="jdbcTemplateA" class="org.springframework.jdbc.core.JdbcTemplate">  
 <property name="dataSource" ref="dataSourceA"/>  
</bean>  
  
<!-- 配置第二个JdbcTemplate -->  
<bean id="jdbcTemplateB" class="org.springframework.jdbc.core.JdbcTemplate">  
 <property name="dataSource" ref="dataSourceB"/>  
</bean>

这里我们配置了两个数据源，对应两个不同的数据库。平常我们使用的数据源大都是c3p0的ComboPooledDataSource，此时需要换成AtomikosNonXADataSourceBean。

配置文件atomikos.xml中，关于事务的配置如下：

<!-- 使用Atomikos进行分布式事务处理 -->  
<bean id="atomikos" class="com.atomikos.icatch.jta.UserTransactionImp"/>  
  
<!-- 配置分布式事务管理器 -->  
<bean id="jtaTransactionManager" class="org.springframework.transaction.jta.JtaTransactionManager">  
 <property name="userTransaction" ref="atomikos"/>  
</bean>  
  
<!-- 启用事务注解 -->  
<tx:annotation-driven transaction-manager="jtaTransactionManager"/>

我们想使用@Transactional注解，因此需要配置tx:annotation-driven标签，该标签需要一个TransactionManager实例。由于这里是分布式事务，因此我们需要配置一个分布式事务管理器，即JtaTransactionManager，而JtaTransactionManager需要一个UserTransaction实例。对于Atomikos来说，UserTransaction的实现类均为UserTransactionImp，因此在配置文件中配置该bean即可。

## 使用XA数据库驱动

这里只用修改atomikos.xml文件中的数据源配置即可，如下：

<!-- 配置第一个数据源 -->  
<bean id="dataSourceA" class="com.atomikos.jdbc.AtomikosDataSourceBean">  
 <property name="uniqueResourceName" value="XA1DBMS"/>  
 <property name="xaDataSourceClassName" value="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource"/>  
 <property name="xaProperties">  
 <props>  
 <prop key="user">${jdbc.user}</prop>  
 <prop key="password">${jdbc.password}</prop>  
 <prop key="url">${jdbc.url}</prop>  
 </props>  
 </property>  
 <property name="poolSize" value="${jdbc.poolSize}"/>  
 <property name="minPoolSize" value="${jdbc.minPoolSize}"/>  
 <property name="maxPoolSize" value="${jdbc.maxPoolSize}"/>  
</bean>  
  
<!-- 配置第二个数据源 -->  
<bean id="dataSourceB" class="com.atomikos.jdbc.AtomikosDataSourceBean">  
 <property name="uniqueResourceName" value="XA2DBMS"/>  
 <property name="xaDataSourceClassName" value="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource"/>  
 <property name="xaProperties">  
 <props>  
 <prop key="user">${jdbc.user}</prop>  
 <prop key="password">${jdbc.password}</prop>  
 <prop key="url">${jdbc.url}</prop>  
 </props>  
 </property>  
 <property name="poolSize" value="${jdbc.poolSize}"/>  
 <property name="minPoolSize" value="${jdbc.minPoolSize}"/>  
 <property name="maxPoolSize" value="${jdbc.maxPoolSize}"/>  
</bean>