Spring源码分析-事务源码分析

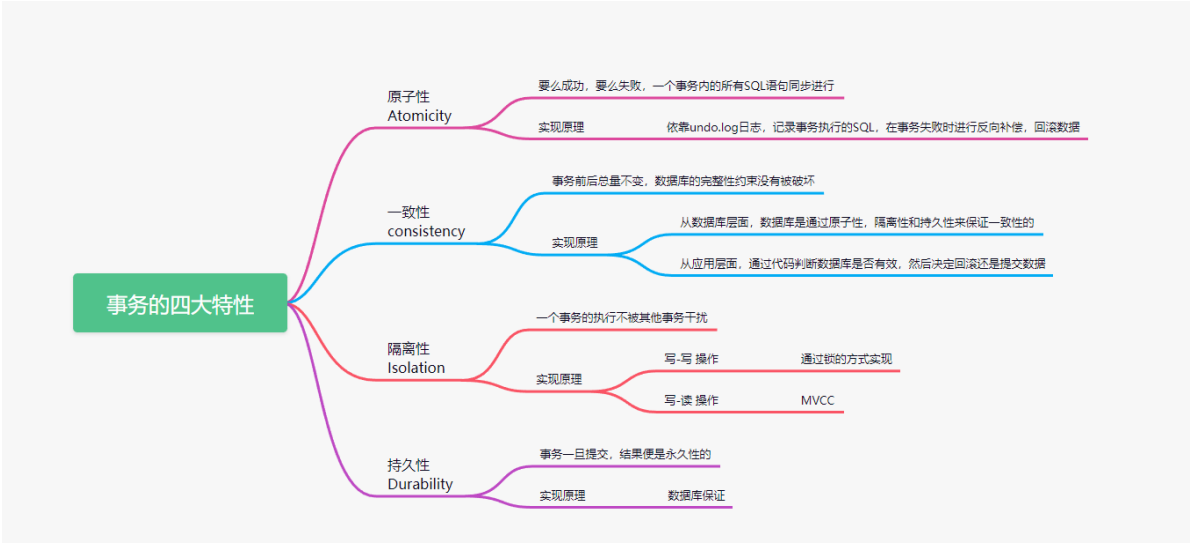
一、事务的本质

1. 何为事务管理

数据库事务(Database Transaction) ，是指作为单个逻辑工作单元执行的一系列操作，要么完全地执行，要么完全地不执行。

事务处理可以确保除非事务性单元内的所有操作都成功完成，否则不会永久更新面向数据的资源。通过将一组相关操作组合为一个要么全部成功要么全部失败的单元，可以简化错误恢复并使应用程序更加可靠。

一个逻辑工作单元要成为事务，必须满足所谓的 `ACID`（原子性、一致性、隔离性和持久性）属性。事务是数据库运行中的逻辑工作单位，由DBMS中的事务管理子系统负责事务的处理。



2. JDBC中的事务管理

事务的本质我们还是要先来看下JDBC中对事务的处理。首先准备如下两张表[案例讲解以MYSQL为主]

```
-- MYSQL
CREATE TABLE t_user (
  id varchar(30) NOT NULL,
  user_name varchar(60) NOT NULL,
  PRIMARY KEY (id)
);

CREATE TABLE t_log (
  id varchar(32) DEFAULT NULL,
  log varchar(20) DEFAULT NULL
);
```

然后创建对应的实体对象

```
/**
 * 用户
```

```
 */
public class User implements Serializable {

    private static final long serialVersionUID = -5575893900970589345L;

    private String id;

    private String userName;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }
}

/**
 * 日志
 */
public class Log implements Serializable {

    private static final long serialVersionUID = -5575893900970589345L;

    private String id;

    private String log;

    public Log() {
    }

    public Log(String id, String log) {
        super();
        this.id = id;
        this.log = log;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getLog() {
        return log;
    }
```

```
    public void setLog(String log) {
        this.log = log;
    }

}
```

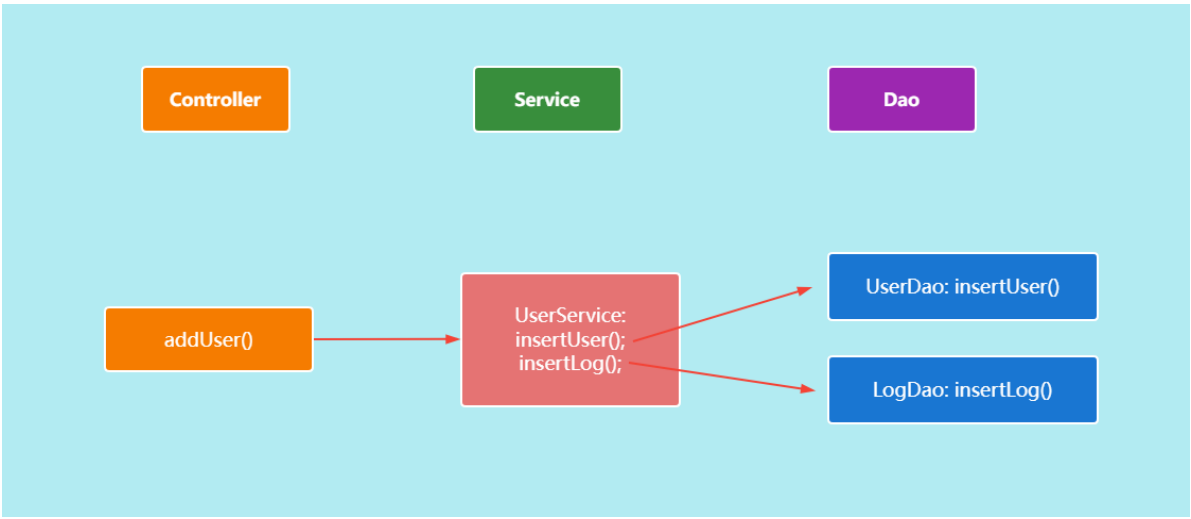
然后我们通过JDBC操作来同时完成添加用户和添加日志的操作。

```
    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;
        try {
            // 注册 JDBC 驱动
            // Class.forName("com.mysql.cj.jdbc.Driver");
            // 打开连接
            conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/test?
characterEncoding=utf-8&serverTimezone=UTC", "root", "123456");
            // 执行查询
            stmt = conn.createStatement();
            conn.setAutoCommit(false); // 关闭自动提交
            // 添加用户信息
            String sql = "INSERT INTO T_USER(id,user_name)values(1,'管理员')";
            stmt.executeUpdate(sql);
            // 添加日志问题
            sql = "INSET INTO t_log(id,log)values(1,'添加了用户:管理员')";
            stmt.executeUpdate(sql);
            conn.commit(); // 上面两个操作都没有问题就提交
        } catch (Exception e) {
            e.printStackTrace();
            // 出现问题就回滚
            try {
                conn.rollback();
            } catch (SQLException throwables) {
                throwables.printStackTrace();
            }
        } finally {
            try {
                if (stmt != null) stmt.close();
            } catch (SQLException se2) {
            }
            try {
                if (conn != null) conn.close();
            } catch (SQLException se) {
                se.printStackTrace();
            }
        }
    }
```

通过上面的代码我们发下关键的操作有这三个：

3. Spring中的事务管理

实际工作中我们更多的是结合Spring来做项目的这时我们要满足的情况是这种。



从上图可以看出我们在Service中是可能调用多个Dao的方法来操作数据库中的数据的，我们要做的就是要保证UserService中的 `adduser()` 方法中的相关操作满足事务的要求。在Spring中支持两种事务的使用方式

第一种基于配置文件的方式：

```
<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:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd
        http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.3.xsd
```

```
        http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-4.1.xsd">
    <!-- 开启扫描 -->
    <context:component-scan base-package="com.dpb.*"></context:component-scan>

    <!-- 配置数据源 -->
    <bean class="org.springframework.jdbc.datasource.DriverManagerDataSource"
id="dataSource">
        <property name="url" value="jdbc:oracle:thin:@localhost:1521:orcl"/>
        <property name="driverClassName"
value="oracle.jdbc.driver.OracleDriver"/>
        <property name="username" value="pms"/>
        <property name="password" value="pms"/>
    </bean>

    <!-- 配置JdbcTemplate -->
    <bean class="org.springframework.jdbc.core.JdbcTemplate" >
        <constructor-arg name="dataSource" ref="dataSource"/>
    </bean>

    <!--
    Spring中，使用XML配置事务三大步骤:
        1. 创建事务管理器
        2. 配置事务方法
        3. 配置AOP
     -->
     <bean
class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
id="transactionManager">
        <property name="dataSource" ref="dataSource"/>
     </bean>
     <tx:advice id="advice" transaction-manager="transactionManager">
        <tx:attributes>
            <tx:method name="fun*" propagation="REQUIRED"/>
        </tx:attributes>
     </tx:advice>
     <!-- aop配置 -->
     <aop:config>
         <aop:pointcut expression="execution(* *..service.*.*(..))" id="tx"/>
         <aop:advisor advice-ref="advice" pointcut-ref="tx"/>
     </aop:config>
</beans>
```

第二种基于注解的使用方式：

```
@Configuration
@ComponentScan("com.study.spring.sample.tx")
@ImportResource("classpath:com/study/spring/sample/tx/application.xml")
@EnableTransactionManagement
public class TxMain {

    public static void main(String[] args) {
        try (AnnotationConfigApplicationContext context = new AnnotationConfigApplicationConte
            User user = new User();
            user.setId("2234561");
            user.setUserName("mike-666666666");

            UserService userService = context.getBean(UserService.class);
            userService.insertUser(user);
            // userService.addUser(user);
        }
    }
}
```

但是我们需要先开启事务注解的方式。然后在对应的方法头部可以添加 `@Transactional`

```
    @Transactional
    public void insertUser(User u) {
        this.userDao.insert(u);
        Log log = new Log(System.currentTimeMillis() + "",
System.currentTimeMillis() + "-" + u.getUserName());
        this.logDao.insert(log);
    }
```

当然上面的操作中涉及到了两个概念 `事务的传播属性` 和 `事务的隔离级别`。参考这两篇文章

传播属性：https://blog.csdn.net/qq_38526573/article/details/87898161

隔离级别：https://blog.csdn.net/qq_38526573/article/details/87898730

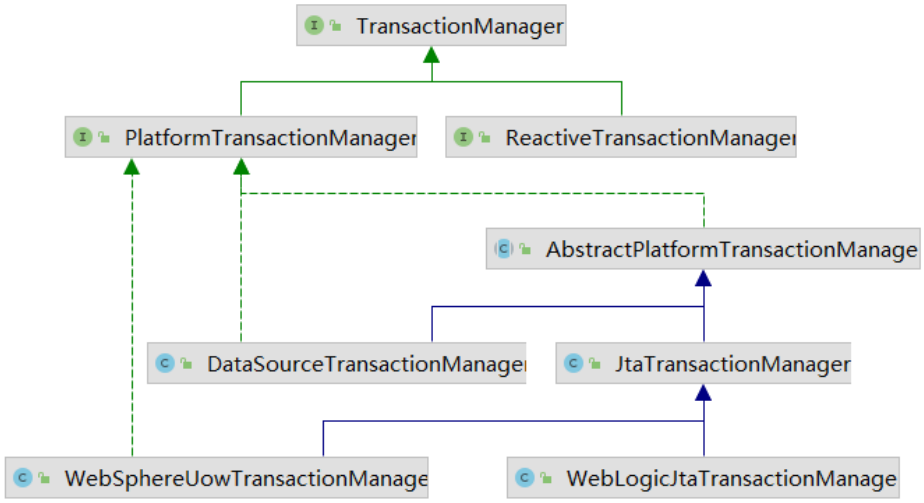
二、Spring事务原理

然后我们来分析下Spring中事务这块的源码实现。

1.Spring事务的源码设计

1.1 事务管理器

我们来看看事务管理器(PlatformTransactionManager).



TransactionManager:是顶级接口，里面是空的。

```
public interface TransactionManager {

}
```

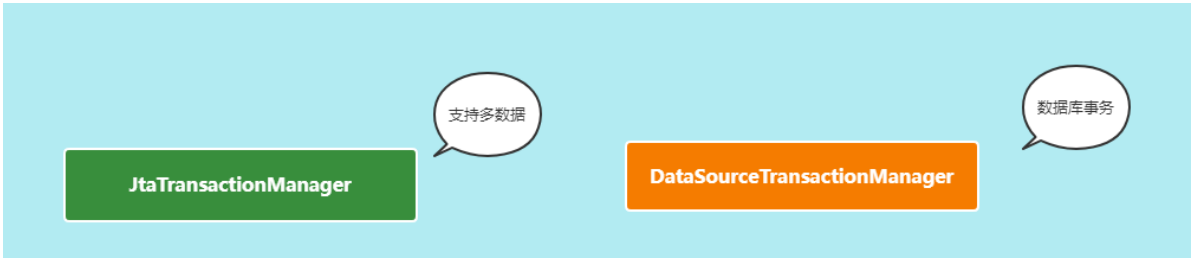
PlatformTransactionManager:平台事务管理器

ReactiveTransactionManager：响应式编程的事务管理器

我们关注的重点是PlatformTransactionManager：

```
public interface PlatformTransactionManager extends TransactionManager {

    /**
            获取事务
    */
    TransactionStatus getTransaction(@Nullable TransactionDefinition definition)
            throws TransactionException;

    /**
      提交数据
     */
    void commit(TransactionStatus status) throws TransactionException;

    /**
        回滚数据
     */
    void rollback(TransactionStatus status) throws TransactionException;

}
```

PlatformTransactionManager也是个接口，在他下面的实现有两个比较重要实现

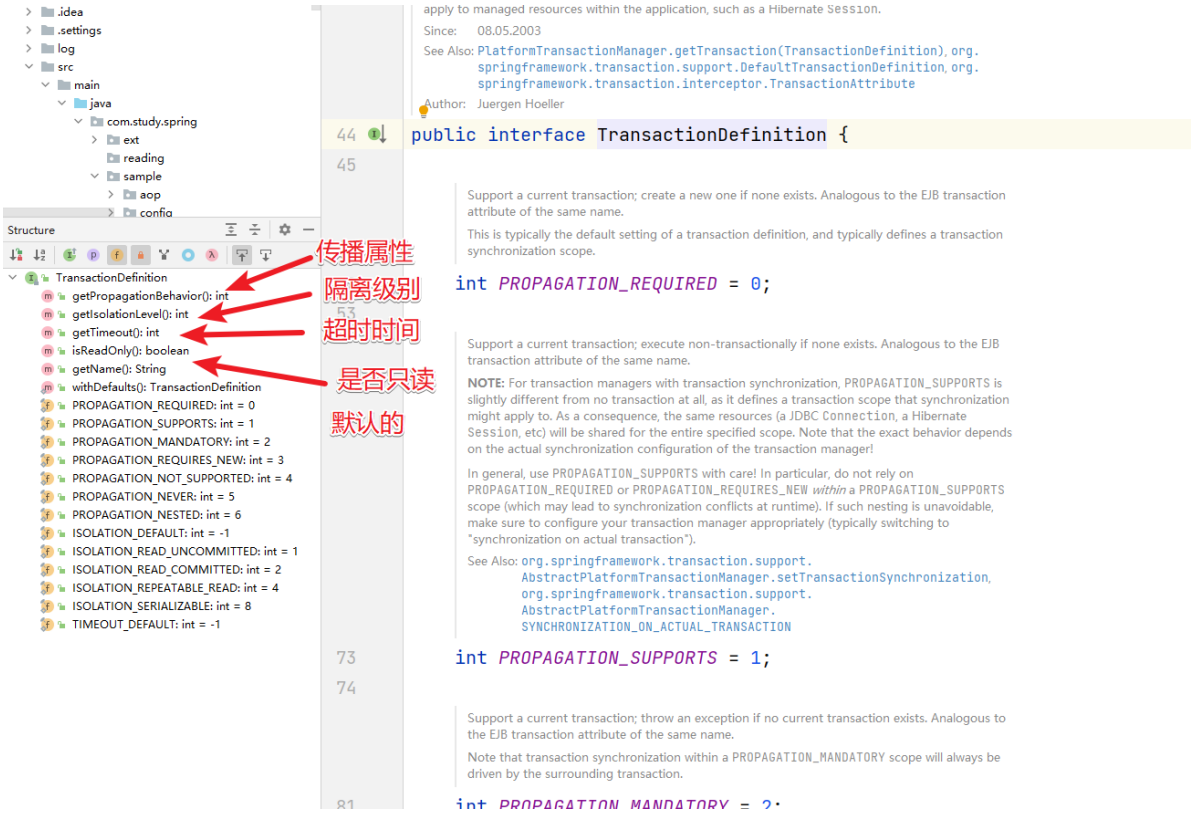


JtaTransactionManager：支持分布式事务【本身服务中的多数据源】

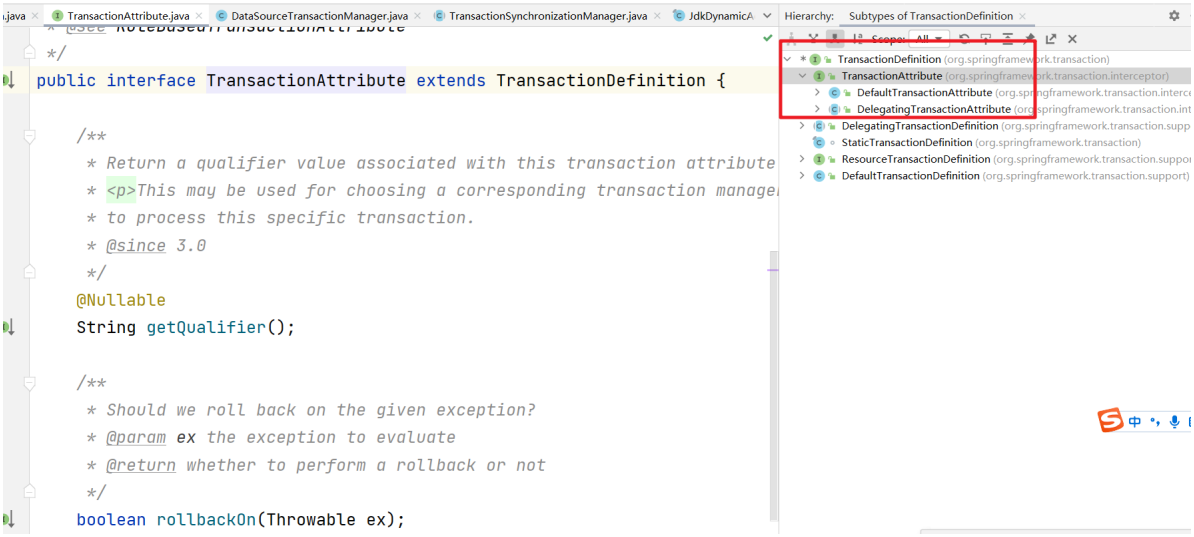
DataSourceTransactionManager：数据源事务管理器。在但数据源中的事务管理，这个是我们分析的重点。

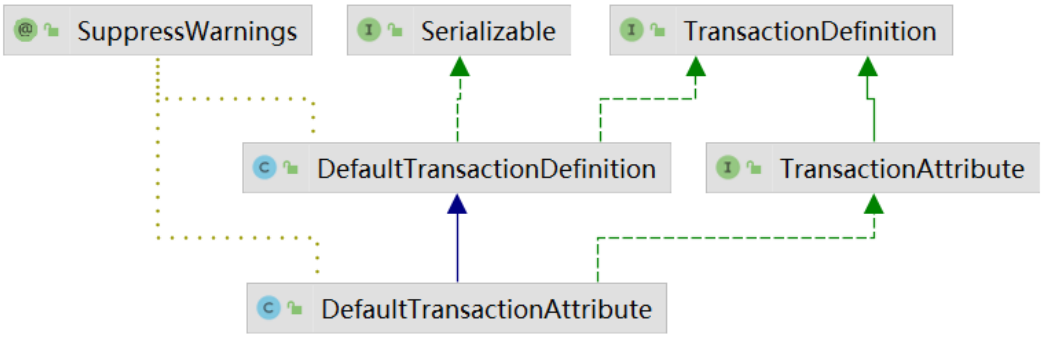
1.2 事务定义

然后我们在上面的 `PlatformTransactoinManager` 中看到了 `TransactionDefinition` 这个对象，通过字面含义是 `事务定义`。我们来看看结构。



也就是 `TransactionDefinition` 中定义了事务的 `传播属性` 和 `隔离级别`，然后来看看具体的体系结构





DefaultTransactionDefinition：是事务定义的默认实现

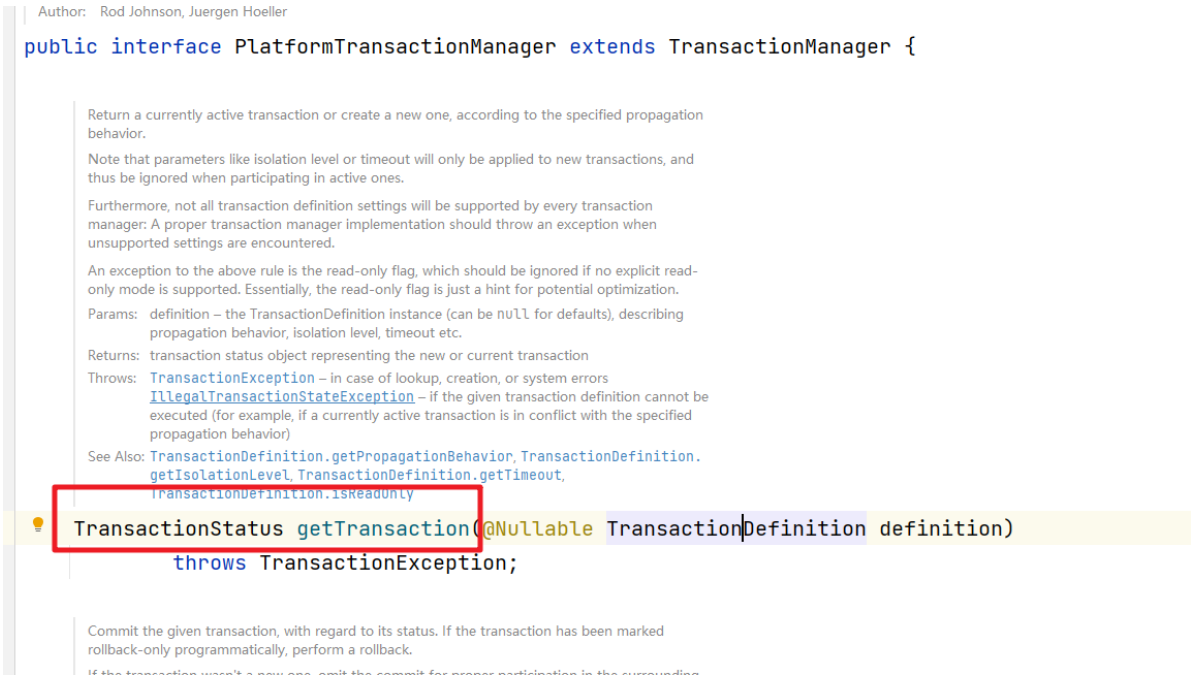
DefaultTransactionAttribute：扩展了TransactionAttribute中的属性的实现

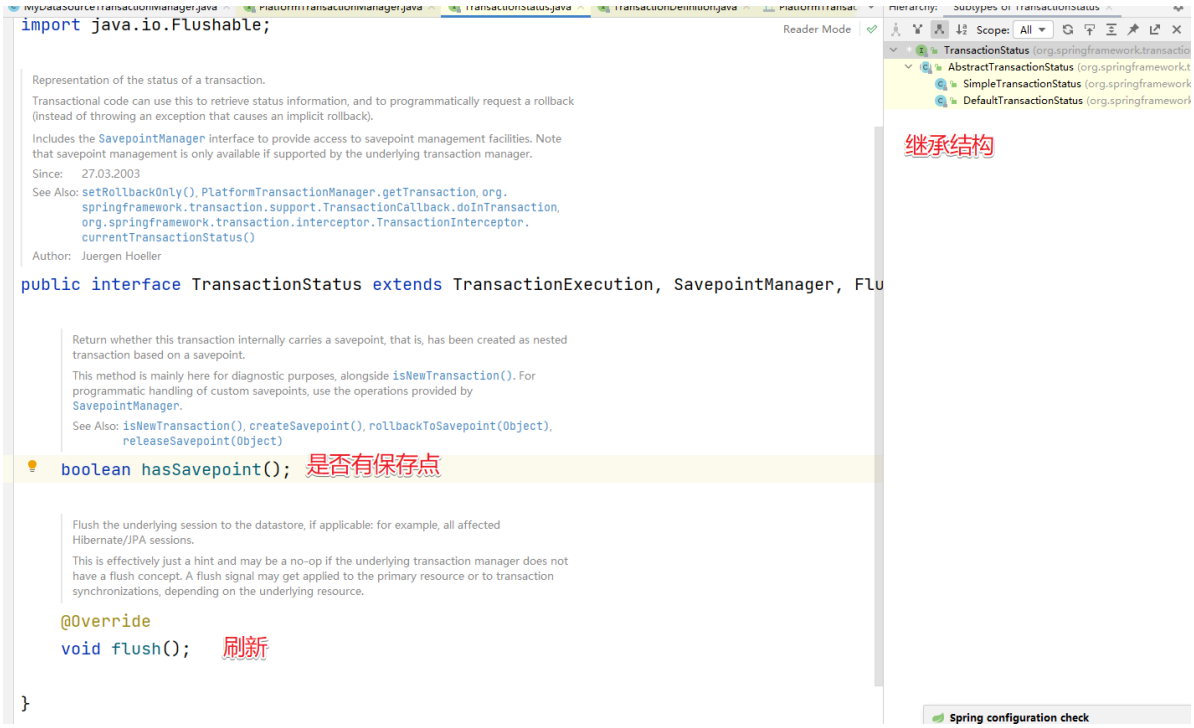
@Transactional:该组件就会被解析加载为对应的 `TransactionDefinition` 对象。

```
@Transactional(propagation = Propagation.REQUIRES_NEW) // 根本没产生新事务
public void insertUser(User u) {

    this.userDao.insert(u);
    Log log = new Log( id: System.currentTimeMillis() + "", log: System.currentTimel
    this.logService.insertLog(log);
}

@Transactional
```

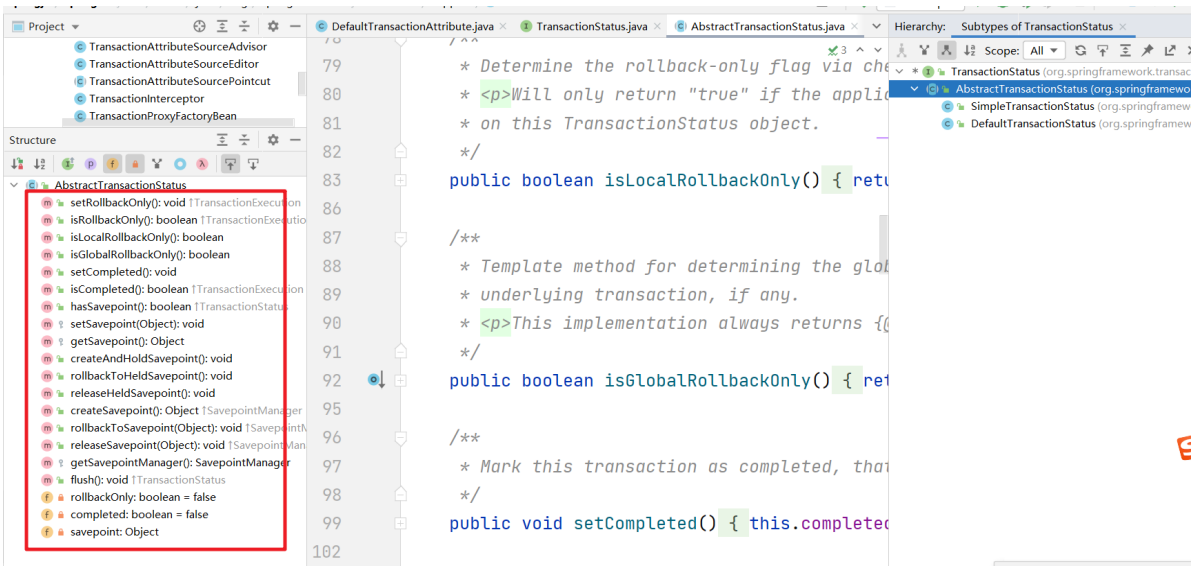
1.3 事务的开启

然后在 `PlatformTransactionManager` 中获取事务的时候返回的是 `TransactionStatus` 对象。我们来看看这个对象。


Author: Rod Johnson, Juergen Hoeller
public interface PlatformTransactionManager extends TransactionManager {
Return a currently active transaction or create a new one, according to the specified propagation behavior.
Note that parameters like isolation level or timeout will only be applied to new transactions, and thus be ignored when participating in active ones.
Furthermore, not all transaction definition settings will be supported by every transaction manager: A proper transaction manager implementation should throw an exception when unsupported settings are encountered.
An exception to the above rule is the read-only flag, which should be ignored if no explicit read-only mode is supported. Essentially, the read-only flag is just a hint for potential optimization.
Params: definition – the TransactionDefinition instance (can be null for defaults), describing propagation behavior, isolation level, timeout etc.
Returns: transaction status object representing the new or current transaction
Throws: TransactionException – in case of lookup, creation, or system errors
IllegalTransactionStateException – if the given transaction definition cannot be executed (for example, if a currently active transaction is in conflict with the specified propagation behavior)
See Also: TransactionDefinition.getPropagationBehavior, TransactionDefinition.getIsolationLevel, TransactionDefinition.getTimeout, TransactionDefinition.isReadOnly
TransactionStatus getTransaction(@Nullable TransactionDefinition definition)
throws TransactionException;
Commit the given transaction, with regard to its status. If the transaction has been marked rollback-only programmatically, perform a rollback.


import java.io.Flushable;
Reader Mode
Representation of the status of a transaction.
Transactional code can use this to retrieve status information, and to programmatically request a rollback (instead of throwing an exception that causes an implicit rollback).
Includes the SavepointManager interface to provide access to savepoint management facilities. Note that savepoint management is only available if supported by the underlying transaction manager.
Since: 27.03.2003
See Also: setRollbackOnly(), PlatformTransactionManager.getTransaction, org.springframework.transaction.support.TransactionCallback.doInTransaction, org.springframework.transaction.interceptor.TransactionInterceptor.currentTransactionStatus()
Author: Juergen Hoeller
public interface TransactionStatus extends TransactionExecution, SavepointManager, Flu
Return whether this transaction internally carries a savepoint, that is, has been created as nested transaction based on a savepoint.
This method is mainly here for diagnostic purposes, alongside isNewTransaction(). For programmatic handling of custom savepoints, use the operations provided by SavepointManager.
See Also: isNewTransaction(), createSavepoint(), rollbackToSavepoint(Object), releaseSavepoint(Object)
boolean hasSavepoint();
是否有保存点
Flush the underlying session to the datastore, if applicable: for example, all affected Hibernate/JPA sessions.
This is effectively just a hint and may be a no-op if the underlying transaction manager does not have a flush concept. A flush signal may get applied to the primary resource or to transaction synchronizations, depending on the underlying resource.
@Override
void flush();
刷新
}
Hierarchy:
Scope: All
TransactionStatus (org.springframework.transactio
AbstractTransactionStatus (org.springframework.t
SimpleTransactionStatus (org.springframework
DefaultTransactionStatus (org.springframework
继承结构
Spring configuration check

子类中扩展了


Project
TransactionAttributeSourceAdvisor
TransactionAttributeSourceEditor
TransactionAttributeSourcePointcut
TransactionInterceptor
TransactionProxyFactoryBean
Structure
AbstractTransactionStatus
setRollbackOnly(): void
isRollbackOnly(): boolean
isLocalRollbackOnly(): boolean
isGlobalRollbackOnly(): boolean
setCompleted(): void
isCompleted(): boolean
hasSavepoint(): boolean
setSavepoint(Object): void
getSavepoint(): Object
createAndHoldSavepoint(): void
rollbackToHeldSavepoint(): void
releaseHeldSavepoint(): void
createSavepoint(): Object
rollbackToSavepoint(Object): void
releaseSavepoint(Object): void
getSavepointManager(): SavepointManager
flush(): void
rollbackOnly: boolean = false
completed: boolean = false
savepoint: Object
DefaultTransactionAttribute.java
TransactionStatus.java
AbstractTransactionStatus.java
79 * Determine the rollback-only flag via che
80 * <p>Will only return "true" if the applic
81 * on this TransactionStatus object.
82 */
83 public boolean isLocalRollbackOnly() { retu
86
87 /**
88 * Template method for determining the glob
89 * underlying transaction, if any.
90 * <p>This implementation always returns {@
91 */
92 public boolean isGlobalRollbackOnly() { ret
95
96 /**
97 * Mark this transaction as completed, that
98 */
99 public void setCompleted() { this.completed
102
Hierarchy:
Subtypes of TransactionStatus
Scope: All
TransactionStatus (org.springframework.transac
AbstractTransactionStatus (org.springframewo
SimpleTransactionStatus (org.springframew
DefaultTransactionStatus (org.springframew

1.4 核心方法讲解

然后再看看核心的 `getTransaction()` 方法

```
    /**
     * This implementation handles propagation behavior. Delegates to
     * {@code doGetTransaction}, {@code isExistingTransaction}
     * and {@code doBegin}.
     * @see #doGetTransaction
     * @see #isExistingTransaction
     * @see #doBegin
     */
    @Override
    public final TransactionStatus getTransaction(@Nullable TransactionDefinition
definition)
            throws TransactionException {

        // Use defaults if no transaction definition given.
        // 如果没有事务定义信息则使用默认的事务管理器定义信息
        TransactionDefinition def = (definition != null ? definition :
TransactionDefinition.withDefaults());

        // 获取事务
        Object transaction = doGetTransaction();
        boolean debugEnabled = logger.isDebugEnabled();

        // 判断当前线程是否存在事务，判断依据为当前线程记录的连接不为空且连接中的
transactionActive属性不为空
        if (isExistingTransaction(transaction)) {
            // Existing transaction found -> check propagation behavior to find
out how to behave.
            // 当前线程已经存在事务
            return handleExistingTransaction(def, transaction, debugEnabled);
        }

        // Check definition settings for new transaction.
        // 事务超时设置验证
        if (def.getTimeout() < TransactionDefinition.TIMEOUT_DEFAULT) {
            throw new InvalidTimeoutException("Invalid transaction timeout",
def.getTimeout());
        }

        // No existing transaction found -> check propagation behavior to find
out how to proceed.
        // 如果当前线程不存在事务，但是PropagationBehavior却被声明为
PROPAGATION_MANDATORY抛出异常
        if (def.getPropagationBehavior() ==
TransactionDefinition.PROPAGATION_MANDATORY) {
            throw new IllegalTransactionStateException(
                    "No existing transaction found for transaction marked with
propagation 'mandatory'");
        }
        // PROPAGATION_REQUIRED，PROPAGATION_REQUIRES_NEW，PROPAGATION_NESTED都需
要新建事务
        else if (def.getPropagationBehavior() ==
TransactionDefinition.PROPAGATION_REQUIRED ||
```

```
                def.getPropagationBehavior() ==
TransactionDefinition.PROPAGATION_REQUIRES_NEW ||
                def.getPropagationBehavior() ==
TransactionDefinition.PROPAGATION_NESTED) {
            //没有当前事务的话，REQUIRED，REQUIRES_NEW，NESTED挂起的是空事务，然后创建一
个新事务
            SuspendedResourcesHolder suspendedResources = suspend(null);
            if (debugEnabled) {
                logger.debug("Creating new transaction with name [" +
def.getName() + "]: " + def);
            }
            try {
                return startTransaction(def, transaction, debugEnabled,
suspendedResources);
            }
            catch (RuntimeException | Error ex) {
                // 恢复挂起的事务
                resume(null, suspendedResources);
                throw ex;
            }
        }
        else {
            // Create "empty" transaction: no actual transaction, but
potentially synchronization.
            // 创建一个空的事务
            if (def.getIsolationLevel() !=
TransactionDefinition.ISOLATION_DEFAULT && logger.isWarnEnabled()) {
                logger.warn("Custom isolation level specified but no actual
transaction initiated; " +
                        "isolation level will effectively be ignored: " + def);
            }
            boolean newSynchronization = (getTransactionSynchronization() ==
SYNCHRONIZATION_ALWAYS);
            return prepareTransactionStatus(def, null, true, newSynchronization,
debugEnabled, null);
        }
    }
```

关键的方法：doGetTransaction()方法

```
    /**
     * 创建一个DataSourceTransactionObject当作事务，设置是否允许保存点，然后获取连接持有器
ConnectionHolder
     * 里面会存放JDBC的连接，设置给DataSourceTransactionObject,当然第一次是空的
     *
     * @return
     */
    @Override
    protected Object doGetTransaction() {
        // 创建一个数据源事务对象
        DataSourceTransactionObject txObject = new
DataSourceTransactionObject();
        // 是否允许当前事务设置保持点
        txObject.setSavepointAllowed(isNestedTransactionAllowed());
        /**
         * TransactionSynchronizationManager 事务同步管理器对象(该类中都是局部线程变量)
```

```
         * 用来保存当前事务的信息,我们第一次从这里去线程变量中获取 事务连接持有器对象 通过数据
源为key去获取
         * 由于第一次进来开始事务 我们的事务同步管理器中没有被存放.所以此时获取出来的
conHolder为null
         */
        ConnectionHolder conHolder =
                (ConnectionHolder)
TransactionSynchronizationManager.getResource(obtainDataSource());
        // 非新创建连接则写false
        txObject.setConnectionHolder(conHolder, false);
        // 返回事务对象
        return txObject;
    }
```

然后事务管理的代码

```
    /**
     * Create a TransactionStatus for an existing transaction.
     */
    private TransactionStatus handleExistingTransaction(
            TransactionDefinition definition, Object transaction, boolean
debugEnabled)
            throws TransactionException {

        /**
         * 判断当前的事务行为是不是PROPAGATION_NEVER的
         * 表示为不支持事务,但是当前又存在一个事务,所以抛出异常
         */
        if (definition.getPropagationBehavior() ==
TransactionDefinition.PROPAGATION_NEVER) {
            throw new IllegalTransactionStateException(
                    "Existing transaction found for transaction marked with
propagation 'never'");
        }

        /**
         * 判断当前的事务属性不支持事务,PROPAGATION_NOT_SUPPORTED,所以需要先挂起已经存在
的事务
         */
        if (definition.getPropagationBehavior() ==
TransactionDefinition.PROPAGATION_NOT_SUPPORTED) {
            if (debugEnabled) {
                logger.debug("Suspending current transaction");
            }
            // 挂起当前事务
            Object suspendedResources = suspend(transaction);
            boolean newSynchronization = (getTransactionSynchronization() ==
SYNCHRONIZATION_ALWAYS);
            // 创建一个新的非事务状态(保存了上一个存在事务状态的属性)
            return prepareTransactionStatus(
                    definition, null, false, newSynchronization, debugEnabled,
suspendedResources);
        }

        /**
         * 当前的事务属性状态是PROPAGATION_REQUIRES_NEW表示需要新开启一个事务状态
         */
```

```
        if (definition.getPropagationBehavior() ==
TransactionDefinition.PROPAGATION_REQUIRES_NEW) {
            if (debugEnabled) {
                logger.debug("Suspending current transaction, creating new
transaction with name [" +
                        definition.getName() + "]");
            }
            // 挂起当前事务并返回挂起的资源持有器
            SuspendedResourcesHolder suspendedResources = suspend(transaction);
            try {
                // 创建一个新的非事务状态(保存了上一个存在事务状态的属性)
                return startTransaction(definition, transaction, debugEnabled,
suspendedResources);
            }
            catch (RuntimeException | Error beginEx) {
                resumeAfterBeginException(transaction, suspendedResources,
beginEx);
                throw beginEx;
            }
        }

        // 嵌套事务
        if (definition.getPropagationBehavior() ==
TransactionDefinition.PROPAGATION_NESTED) {
            // 不允许就报异常
            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() + "]");
            }
            // 嵌套事务的处理
            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.
                // 如果没有可以使用保存点的方式控制事务回滚，那么在嵌入式事务的建立初始简历保
存点
                DefaultTransactionStatus status =
                        prepareTransactionStatus(definition, transaction, false,
false, debugEnabled, null);
                // 为事务设置一个回退点
                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.
```

```
                // 有些情况是不能使用保存点操作
                return startTransaction(definition, transaction, debugEnabled,
null);
            }
        }

        // 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);
    }
```

最后来看看 startTransaction() 方法

```
    /**
     * Start a new transaction.
     */
    private TransactionStatus startTransaction(TransactionDefinition definition,
Object transaction,
            boolean debugEnabled, @Nullable SuspendedResourcesHolder
suspendedResources) {

        // 是否需要新同步
```

```
        boolean newSynchronization = (getTransactionSynchronization() !=
SYNCHRONIZATION_NEVER);
        // 创建新的事务
        DefaultTransactionStatus status = newTransactionStatus(
                definition, transaction, true, newSynchronization, debugEnabled,
suspendedResources);
        // 开启事务和连接
        doBegin(transaction, definition);
        // 新同步事务的设置，针对于当前线程的设置
        prepareSynchronization(status, definition);
        return status;
    }
```

doBegin方法开启和连接事务

```
    @Override
    protected void doBegin(Object transaction, TransactionDefinition definition)
{
        // 强制转化事务对象
        DataSourceTransactionObject txObject = (DataSourceTransactionObject)
transaction;
        Connection con = null;

        try {
            // 判断事务对象没有数据库连接持有器
            if (!txObject.hasConnectionHolder() ||

txObject.getConnectionHolder().isSynchronizedWithTransaction()) {
                // 通过数据源获取一个数据库连接对象
                Connection newCon = obtainDataSource().getConnection();
                if (logger.isDebugEnabled()) {
                    logger.debug("Acquired Connection [" + newCon + "] for JDBC
transaction");
                }
                // 把我们的数据库连接包装成一个ConnectionHolder对象 然后设置到我们的
txObject对象中去
                txObject.setConnectionHolder(new ConnectionHolder(newCon),
true);
            }

            // 标记当前的连接是一个同步事务
            txObject.getConnectionHolder().setSynchronizedWithTransaction(true);
            con = txObject.getConnectionHolder().getConnection();

            // 为当前的事务设置隔离级别
            Integer previousIsolationLevel =
DataSourceUtils.prepareConnectionForTransaction(con, definition);
            // 设置先前隔离级别
            txObject.setPreviousIsolationLevel(previousIsolationLevel);
            // 设置是否只读
            txObject.setReadOnly(definition.isReadOnly());

            // 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).
```

```
            // 关闭自动提交
            if (con.getAutoCommit()) {
                //设置需要恢复自动提交
                txObject.setMustRestoreAutoCommit(true);
                if (logger.isDebugEnabled()) {
                    logger.debug("Switching JDBC Connection [" + con + "] to
manual commit");
                }
                // 关闭自动提交
                con.setAutoCommit(false);
            }

            // 判断事务是否需要设置为只读事务
            prepareTransactionalConnection(con, definition);
            // 标记激活事务
            txObject.getConnectionHolder().setTransactionActive(true);

            // 设置事务超时时间
            int timeout = determineTimeout(definition);
            if (timeout != TransactionDefinition.TIMEOUT_DEFAULT) {
                txObject.getConnectionHolder().setTimeoutInSeconds(timeout);
            }

            // Bind the connection holder to the thread.
            // 绑定我们的数据源和连接到我们的同步管理器上，把数据源作为key,数据库连接作为
value 设置到线程变量中
            if (txObject.isNewConnectionHolder()) {
                // 将当前获取到的连接绑定到当前线程

TransactionSynchronizationManager.bindResource(obtainDataSource(),
txObject.getConnectionHolder());
            }
        }

        catch (Throwable ex) {
            if (txObject.isNewConnectionHolder()) {
                // 释放数据库连接
                DataSourceUtils.releaseConnection(con, obtainDataSource());
                txObject.setConnectionHolder(null, false);
            }
            throw new CannotCreateTransactionException("Could not open JDBC
Connection for transaction", ex);
        }
    }
```

在doBegin方法中核心的关闭了自动提交

```
        // 设置是否只读
        txObject.setReadOnly(definition.isReadOnly());

        // Switch to manual commit if necessary. This is very expensive in some JDBC dr
        // so we don't want to do it unnecessarily (for example if we've explicitly
        // configured the connection pool to set it already).
        // 关闭自动提交
        if (con.getAutoCommit()) {
            //设置需要恢复自动提交
            txObject.setMustRestoreAutoCommit(true);
            if (logger.isDebugEnabled()) {
                logger.debug("Switching JDBC Connection [" + con + "] to manual commit"
            }
            // 关闭自动提交
            con.setAutoCommit(false);
        }

        // 判断事务是否需要设置为只读事务
```

同时把连接绑定到本地线程中bindResource方法

```
     */
    public static void bindResource(Object key, Object value) throws IllegalStateException {
        Object actualKey = TransactionSynchronizationUtils.unwrapResourceIfNecessary(key);
        Assert.notNull(value, message: "Value must not be null");//每次在进行获取的时候都要根据obtainDataSo
        Map<Object, Object> map = resources.get();
        // set ThreadLocal Map if none found
        if (map == null) {
            map = new HashMap<>();
            resources.set(map);
        }
        Object oldValue = map.put(actualKey, value);
        // Transparently suppress a ResourceHolder that was marked as void...
        if (oldValue instanceof ResourceHolder && ((ResourceHolder) oldValue).isVoid()) {
            oldValue = null;
        }
        if (oldValue != null) {
            throw new IllegalStateException("Already value [" + oldValue + "] for key [" +
                    actualKey + "] bound to thread [" + Thread.currentThread().getName()
        }
        if (logger.isTraceEnabled()) {
```


2.Spring事务源码串联

2.1 编程式事务

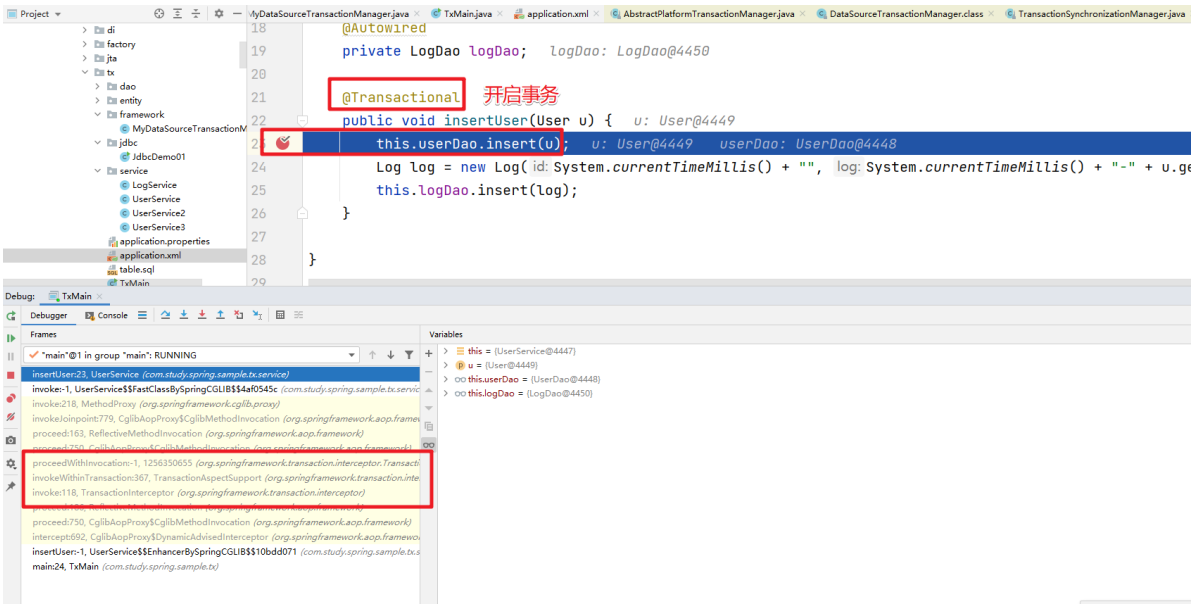
结合上面的设计我们就可以来实现事务的处理了

```
    @Autowired
    private UserDao userDao;

    @Autowired
    private PlatformTransactionManager txManager;

    @Autowired
    private LogService logService;

    @Transactional
    public void insertUser(User u) {

        // 1、创建事务定义
```

```
        DefaultTransactionDefinition definition = new
DefaultTransactionDefinition();

        // 2、根据定义开启事务
        TransactionStatus status = txManager.getTransaction(definition);

        try {
            this.userDao.insert(u);
            Log log = new Log(System.currentTimeMillis() + "",
System.currentTimeMillis() + "-" + u.getUserName());
            // this.doAddUser(u);
            this.logService.insertLog(log);
            // 3、提交事务
            txManager.commit(status);
        } catch (Exception e) {
            // 4、异常了，回滚事务
            txManager.rollback(status);
            throw e;
        }
    }
```

2.2 AOP事务

上面的案例代码我们可以看到在Service中我们通过事务处理的代码实现了事务管理，同时结合我们前面学习的AOP的内容，我们发现我们完全可以把事务的代码抽取出来，然后我们来看看Spring中这块是如何处理的。



我们可以通过Debug的方式看到处理的关键流程 `TransactionInterceptor` 就是事务处理的 advice

```
@Override
    @Nullable
    public Object invoke(MethodInvocation invocation) throws Throwable {
        // Work out the target class: may be {@code null}.
        // The TransactionAttributeSource should be passed the target class
        // as well as the method, which may be from an interface.
        Class<?> targetClass = (invocation.getThis() != null ?
AopUtils.getTargetClass(invocation.getThis()) : null);

        // Adapt to TransactionAspectSupport's invokeWithinTransaction...
        return invokeWithinTransaction(invocation.getMethod(), targetClass,
invocation::proceed);
    }
```

进入到invokeWithinTransaction方法中

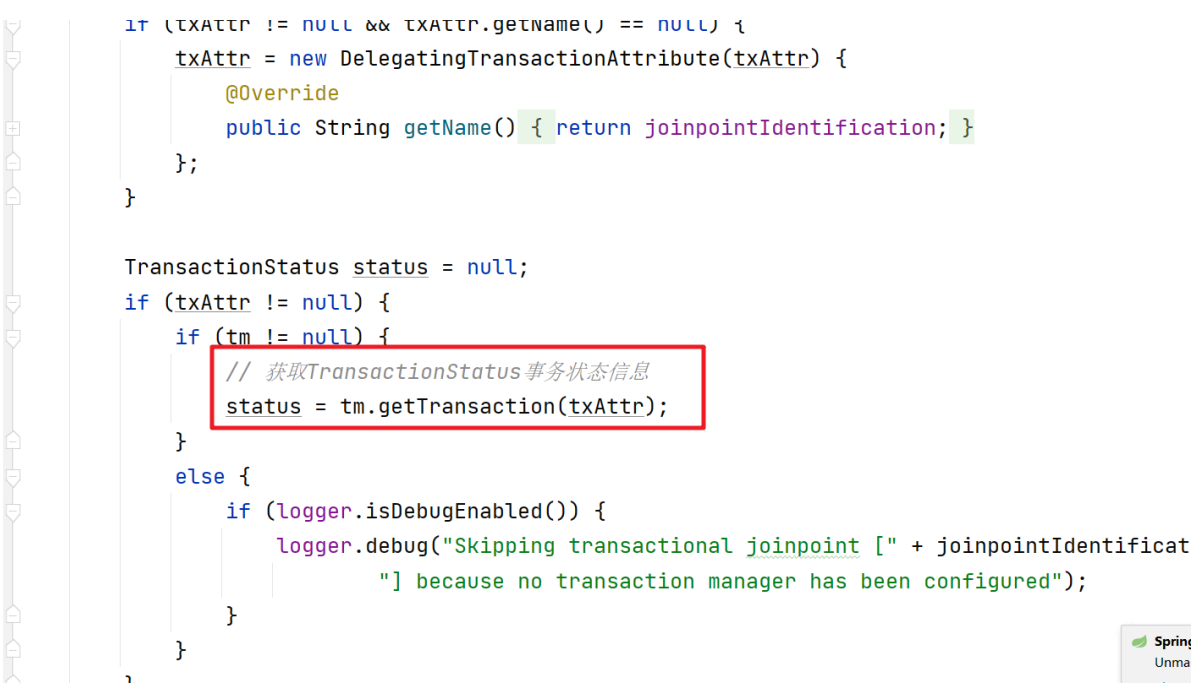
```
    @Nullable
    protected Object invokeWithinTransaction(Method method, @Nullable Class<?>
targetClass,
            final InvocationCallback invocation) throws Throwable {

        // If the transaction attribute is null, the method is non-
transactional.
        // 获取我们的事务属性源对象
        TransactionAttributeSource tas = getTransactionAttributeSource();
        // 通过事务属性源对象获取到当前方法的事务属性信息
        final TransactionAttribute txAttr = (tas != null ?
tas.getTransactionAttribute(method, targetClass) : null);
        // 获取我们配置的事务管理器对象
        final TransactionManager tm = determineTransactionManager(txAttr);

        if (this.reactiveAdapterRegistry != null && tm instanceof
ReactiveTransactionManager) {
            ReactiveTransactionSupport txSupport =
this.transactionSupportCache.computeIfAbsent(method, key -> {
                if (KotlinDetector.isKotlinType(method.getDeclaringClass()) &&
KotlinDelegate.isSuspend(method)) {
                    throw new TransactionUsageException(
                            "Unsupported annotated transaction on suspending
function detected: " + method +
                            ". Use TransactionalOperator.transactional
extensions instead.");
                }
                ReactiveAdapter adapter =
this.reactiveAdapterRegistry.getAdapter(method.getReturnType());
                if (adapter == null) {
                    throw new IllegalStateException("Cannot apply reactive
transaction to non-reactive return type: " +
                            method.getReturnType());
                }
                return new ReactiveTransactionSupport(adapter);
            });
            return txSupport.invokeWithinTransaction(
                    method, targetClass, invocation, txAttr,
(ReactiveTransactionManager) tm);
```

```
        }

        PlatformTransactionManager ptm = asPlatformTransactionManager(tm);
        // 获取连接点的唯一标识  类名+方法名
        final String joinpointIdentification = methodIdentification(method,
targetClass, txAttr);

        // 声明式事务处理
        if (txAttr == null || !(ptm instanceof
CallbackPreferringPlatformTransactionManager)) {
            // Standard transaction demarcation with getTransaction and
commit/rollback calls.
            // 创建TransactionInfo
            TransactionInfo txInfo = createTransactionIfNecessary(ptm, txAttr,
joinpointIdentification);

            Object retVal;
            try {
                // This is an around advice: Invoke the next interceptor in the
chain.
                // This will normally result in a target object being invoked.
                // 执行被增强方法,调用具体的处理逻辑
                retVal = invocation.proceedWithInvocation();
            }
            catch (Throwable ex) {
                // target invocation exception
                // 异常回滚
                completeTransactionAfterThrowing(txInfo, ex);
                throw ex;
            }
            finally {
                //清除事务信息，恢复线程私有的老的事务信息
                cleanupTransactionInfo(txInfo);
            }

            if (retVal != null && vavrPresent && VavrDelegate.isVavrTry(retVal))
{
                // Set rollback-only in case of Vavr failure matching our
rollback rules...
                TransactionStatus status = txInfo.getTransactionStatus();

                if (status != null && txAttr != null) {
                    retVal = VavrDelegate.evaluateTryFailure(retVal, txAttr,
status);
                }
            }

            //成功后提交，会进行资源储量，连接释放，恢复挂起事务等操作
            commitTransactionAfterReturning(txInfo);
            return retVal;
        }

        else {
            // 编程式事务处理
            Object result;
            final ThrowableHolder throwableHolder = new ThrowableHolder();
```

```
            // It's a CallbackPreferringPlatformTransactionManager: pass a
TransactionCallback in.
            try {
                result = ((CallbackPreferringPlatformTransactionManager)
ptm).execute(txAttr, status -> {
                    TransactionInfo txInfo = prepareTransactionInfo(ptm, txAttr,
joinpointIdentification, status);
                    try {
                        Object retVal = invocation.proceedWithInvocation();
                        if (retVal != null && vavrPresent &&
VavrDelegate.isVavrTry(retVal)) {
                            // Set rollback-only in case of Vavr failure
matching our rollback rules...
                            retVal = VavrDelegate.evaluateTryFailure(retVal,
txAttr, status);
                        }
                        return retVal;
                    }
                    catch (Throwable ex) {
                        if (txAttr.rollbackOn(ex)) {
                            // A RuntimeException: will lead to a rollback.
                            if (ex instanceof RuntimeException) {
                                throw (RuntimeException) ex;
                            }
                            else {
                                throw new ThrowableHolderException(ex);
                            }
                        }
                        else {
                            // A normal return value: will lead to a commit.
                            throwableHolder.throwable = ex;
                            return null;
                        }
                    }
                    finally {
                        cleanupTransactionInfo(txInfo);
                    }
                });
            }
            catch (ThrowableHolderException ex) {
                throw ex.getCause();
            }
            catch (TransactionSystemException ex2) {
                if (throwableHolder.throwable != null) {
                    logger.error("Application exception overridden by commit
exception", throwableHolder.throwable);
                    ex2.initApplicationException(throwableHolder.throwable);
                }
                throw ex2;
            }
            catch (Throwable ex2) {
                if (throwableHolder.throwable != null) {
                    logger.error("Application exception overridden by commit
exception", throwableHolder.throwable);
                }
                throw ex2;
            }
```

```
            // Check result state: It might indicate a Throwable to rethrow.
            if (throwableHolder.throwable != null) {
                throw throwableHolder.throwable;
            }
            return result;
        }
    }
```

然后进入到createTransactionIfNecessary方法中



然后进入 getTransaction 这个方法我们前面看过

```
            //没有当前事务的话，REQUIRED，REQUIRES_NEW，NESTED挂起的是空事务，然后创建一个新事务
            SuspendedResourcesHolder suspendedResources = suspend( transaction: null);
            if (debugEnabled) {
                logger.debug("Creating new transaction with name [" + def.getName() + "]: " + def);
            }
            try {
                return startTransaction(def, transaction, debugEnabled, suspendedResources);
            }
            catch (RuntimeException | Error ex) {
                // 恢复挂起的事务
                resume( transaction: null, suspendedResources);
                throw ex;
            }
        }
        else {
            // Create "empty" transaction: no actual transaction, but potentially synchronization.
            // 创建一个空的事务
            if (def.getIsolationLevel() != TransactionDefinition.ISOLATION_DEFAULT && logger.isWarnEnab
                logger.warn("Custom isolation level specified but no actual transactio
                        "isolation level will effectively be ignored: " + def);
```

```
    private TransactionStatus startTransaction(TransactionDefinition definition, Object transaction,
            boolean debugEnabled, @Nullable SuspendedResourcesHolder suspendedResources) {

        // 是否需要新同步
        boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION_NEVER);
        // 创建新的事务
        DefaultTransactionStatus status = newTransactionStatus(
                definition, transaction, newTransaction: true, newSynchronization, debugEnabled, suspendedR
        // 开启事务和连接
        doBegin(transaction, definition);
        // 新同步事务的设置，针对于当前线程的设置
        prepareSynchronization(status, definition);
        return status;
    }
```

核心的是doBegin方法。完成 自动提交的关闭和 本地线程 对象的存储

```
            }
            // 关闭自动提交
            con.setAutoCommit(false);
        }

        // 判断事务是否需要设置为只读事务
        prepareTransactionalConnection(con, definition);
        // 标记激活事务
        txObject.getConnectionHolder().setTransactionActive(true);

        // 设置事务超时时间
        int timeout = determineTimeout(definition);
        if (timeout != TransactionDefinition.TIMEOUT_DEFAULT) {...}

        // Bind the connection holder to the thread.
        // 绑定我们的数据源和连接到我们的同步管理器上，把数据源作为key,数据库连接作为value 设置到线程变量中
        if (txObject.isNewConnectionHolder()) {
            // 将当前获取到的连接绑定到当前线程
            TransactionSynchronizationManager.bindResource(obtainDataSource(), txOb
        }
    }
```

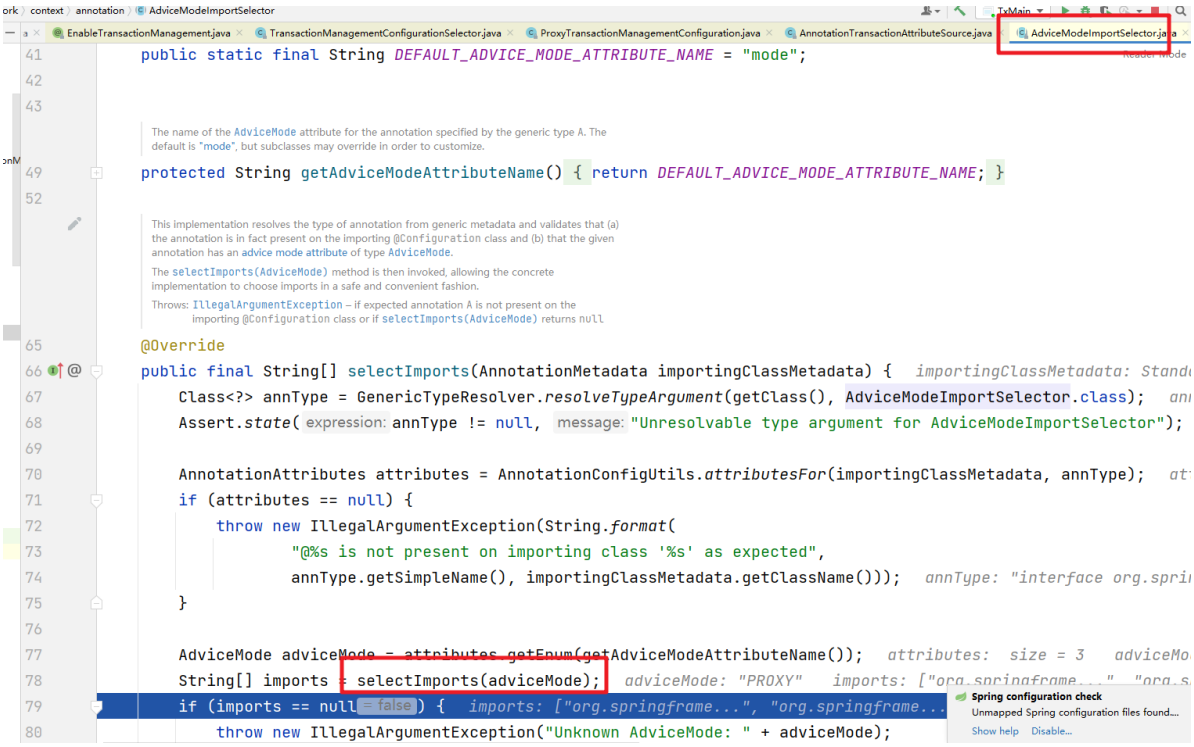
2.3 TransactionInterceptor

接下来看看TransactionInterceptor是如何注入到容器中的，首先来看看事务的开启@EnableTransactionManagement

```java
@Configuration
@ComponentScan("com.study.spring.sample.tx")
@ImportResource("classpath:com/study/spring/sample/tx/application.xml")
@EnableTransactionManagement
public class TxMain {

    public static void main(String[] args) {
        try (AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(TxMain.
            User user = new User();
            user.setId("2234561");
            user.setUserName("mike-666666666");

            UserService userService = context.getBean(UserService.class);
            userService.insertUser(user);
            // userService.addUser(user);
        }
    }
}
```

```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(TransactionManagementConfigurationSelector.class)
public @interface EnableTransactionManagement {

    /**
     * Indicate whether subclass-based (CGLIB) proxies are to be created (true) as opposed to standard
     * Java interface-based proxies (false). The default is false. Applicable only if mode() is set to
     * AdviceMode.PROXY.
     * <p>Note that setting this attribute to true will affect all Spring-managed beans requiring proxying,
     * not just those marked with @Transactional. For example, other beans marked with Spring's
     * @Async annotation will be upgraded to subclass proxying at the same time. This approach has no
     * negative impact in practice unless one is explicitly expecting one type of proxy vs another, e.g. in
     * tests.
     */
    boolean proxyTargetClass() default false;

    /**
     * Indicate how transactional advice should be applied.
     * <p><b>The default is AdviceMode.PROXY.</b> Please note that proxy mode allows for interception of calls
     * through the proxy only. Local calls within the same class cannot get intercepted that way; an
     * Transactional annotation on such a method within a local call will be ignored since Spring's
     * interceptor does not even kick in for such a runtime scenario. For a more advanced mode of
     * interception, consider switching this to AdviceMode.ASPECTJ.
     */
    AdviceMode mode() default AdviceMode.PROXY;

    /**
     * Indicate the ordering of the execution of the transaction advisor when multiple advices are
     * applied at a specific joinpoint.
     * <p>The default is Ordered.LOWEST_PRECEDENCE.
     */
    int order() default Ordered.LOWEST_PRECEDENCE;

}
```

一步步进入

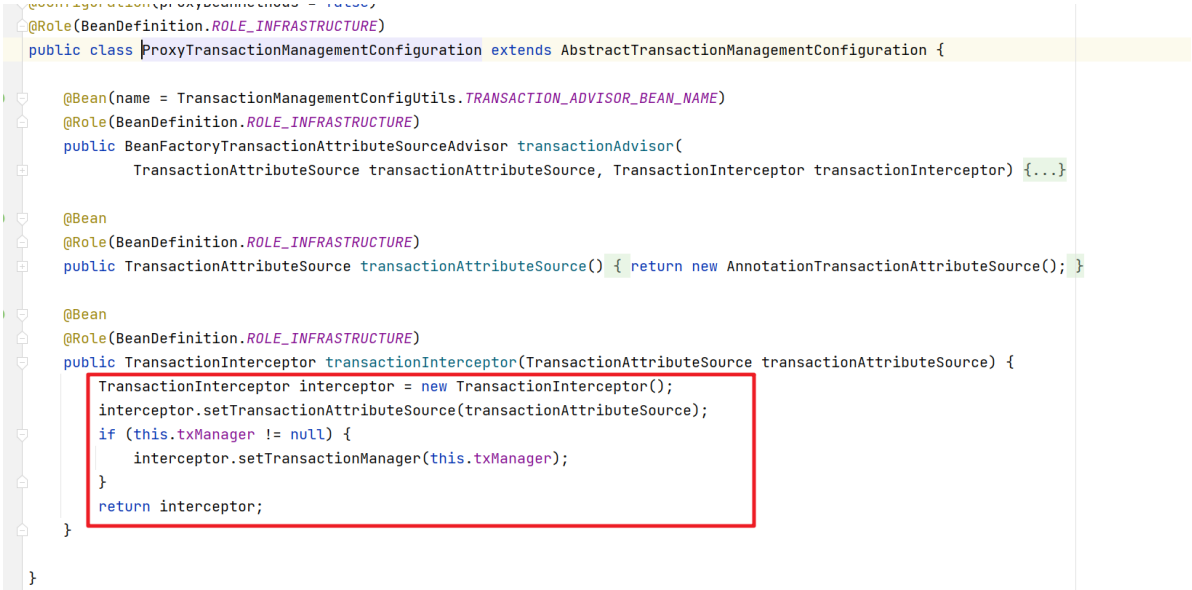


```
public static final String DEFAULT_ADVICE_MODE_ATTRIBUTE_NAME = "mode";

protected String getAdviceModeAttributeName() { return DEFAULT_ADVICE_MODE_ATTRIBUTE_NAME; }

@Override
public final String[] selectImports(AnnotationMetadata importingClassMetadata) {
    Class<?> annType = GenericTypeResolver.resolveTypeArgument(getClass(), AdviceModeImportSelector.class);
    Assert.state( expression: annType != null, message: "Unresolvable type argument for AdviceModeImportSelector");

    AnnotationAttributes attributes = AnnotationConfigUtils.attributesFor(importingClassMetadata, annType);
    if (attributes == null) {
        throw new IllegalArgumentException(String.format(
                "@%s is not present on importing class '%s' as expected",
                annType.getSimpleName(), importingClassMetadata.getClassName()));
    }

    AdviceMode adviceMode = attributes.getEnum(getAdviceModeAttributeName());
    String[] imports = selectImports(adviceMode);
    if (imports == null) {
        throw new IllegalArgumentException("Unknown AdviceMode: " + adviceMode);
```

```
public class TransactionManagementConfigurationSelector extends AdviceModeImportSelector<EnableTransactio

@Override
protected String[] selectImports(AdviceMode adviceMode) {
    switch (adviceMode) {
        case PROXY:
            return new String[] {AutoProxyRegistrar.class.getName(),
                    ProxyTransactionManagementConfiguration.class.getName()};
        case ASPECTJ:
            return new String[] {determineTransactionAspectClass()};
        default:
            return null;
    }
}
```

这个是关键

可以看到对应的拦截器的注入



```
@Role(BeanDefinition.ROLE_INFRASTRUCTURE)
public class ProxyTransactionManagementConfiguration extends AbstractTransactionManagementConfiguration {

    @Bean(name = TransactionManagementConfigUtils.TRANSACTION_ADVISOR_BEAN_NAME)
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    public BeanFactoryTransactionAttributeSourceAdvisor transactionAdvisor(
            TransactionAttributeSource transactionAttributeSource, TransactionInterceptor transactionInterceptor) {...}

    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    public TransactionAttributeSource transactionAttributeSource() { return new AnnotationTransactionAttributeSource(); }

    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    public TransactionInterceptor transactionInterceptor(TransactionAttributeSource transactionAttributeSource) {
        TransactionInterceptor interceptor = new TransactionInterceptor();
        interceptor.setTransactionAttributeSource(transactionAttributeSource);
        if (this.txManager != null) {
            interceptor.setTransactionManager(this.txManager);
        }
        return interceptor;
    }

}
```

然后可以看到拦截器关联到了Advisor中了

```
Author: Chris Beams, Sebastien Deleuze
@Configuration(proxyBeanMethods = false)
@Role(BeanDefinition.ROLE_INFRASTRUCTURE)
public class ProxyTransactionManagementConfiguration extends AbstractTransactionManagementConfiguration {

    @Bean(name = TransactionManagementConfigUtils.TRANSACTION_ADVISOR_BEAN_NAME)
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    public BeanFactoryTransactionAttributeSourceAdvisor transactionAdvisor(
            TransactionAttributeSource transactionAttributeSource, TransactionInterceptor transactionInterceptor) {

        BeanFactoryTransactionAttributeSourceAdvisor advisor = new BeanFactoryTransactionAttributeSourceAdvisor();
        advisor.setTransactionAttributeSource(transactionAttributeSource);
        advisor.setAdvice(transactionInterceptor);
        if (this.enableTx != null) {
            advisor.setOrder(this.enableTx.<~>getNumber( attributeName: "order"));
        }
        return advisor;
    }
```

到这儿就分析完了