

# 咕泡学院 JavaVIP 高级课程教案

## 深入分析 Spring 源码（第四阶段）

### 关于本文档

主题	咕泡学院 Java VIP 高级课程教案--深入分析 Spring 源码(第四阶段)
主讲	Tom 老师
适用对象	咕泡学院 Java 高级 VIP 学员及 VIP 授课老师
源码版本	v3.2.6.RELEASE

## 八、Spring 事务原理详解

### 8.1、什么是事务(Transaction)

事务(Transaction)是访问并可能更新数据库中各种数据项的一个程序执行单元(unit)。

#### 特点:

事务是恢复和并发控制的基本单位。

事务应该具有 4 个属性：原子性、一致性、隔离性、持久性。这四个属性通常称为 **ACID 特性**。

原子性 (atomicity)。一个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做。

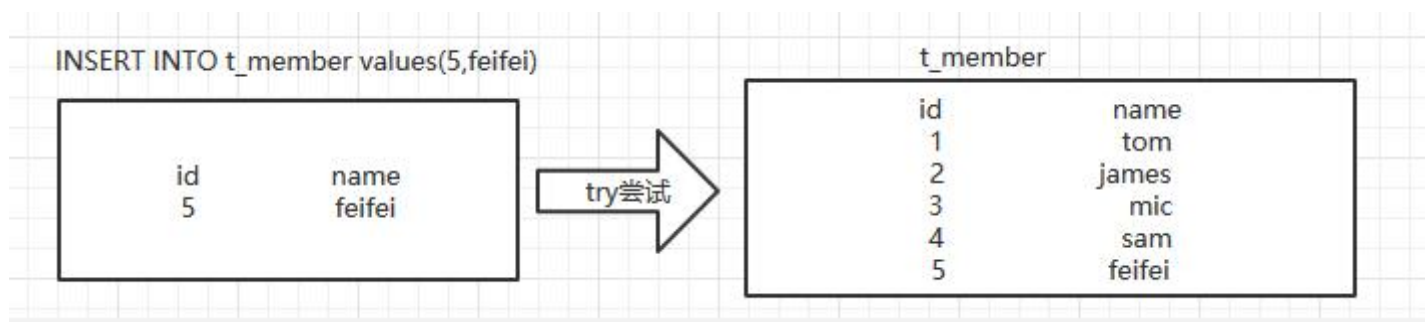
一致性 (consistency)。事务必须是使数据库从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的。

隔离性 (isolation)。一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。

持久性 (durability)。持久性也称永久性 (permanence)，指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响。

#### 插入

- 1、先把要插入的数据放入临时表
- 2、将临时表中数据插入实际表中去
- 3、如果没问题，就复制一份到实际表中，并将临时表中的数据删除
- 4、如果有问题，返回错误信息，临时表清空



#### 删除

- 1、先根据条件从原始表中查询来满足条件的数据行
- 2、将这些数据行复制一份到临时表
- 3、执行删除，如果出现错误，原来的数据原封不动，清空临时表中满足本次条件的记录，返回错误码
- 4、如果执行成功，真正的干掉原始表中的记录。返回影响行数

提供了一个后悔的机会（常常说，世界上没有后悔药，但是，在我们的虚拟世界中存在后悔药）

#### 事务操作的基本流程

- 1、事务开启(open)
- 2、执行事务(execute)

- 3、提交事务（自动提交 AutoCommit/ 手动提交(CustomCommit)）  
事务回滚（rollback）（如果出现错误）
- 4、关闭事务(close)

8.2、事务的基本原理

Spring 事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，spring 是无法提供事务功能的。对于纯 JDBC 操作数据库，想要用到事务，可以按照以下步骤进行：

- 1. 获取连接 Connection con = DriverManager.getConnection()
- 2. 开启事务 con.setAutoCommit(true/false);
- 3. 执行 CRUD
- 4. 提交事务/回滚事务 con.commit() / con.rollback();
- 5. 关闭连接 conn.close();

使用 Spring 的事务管理功能后，我们可以不再写步骤 2 和 4 的代码，而是由 Spring 自动完成。那么 Spring 是如何在我们书写的 CRUD 之前和之后开启事务和关闭事务的呢？解决这个问题，也就可以从整体上理解 Spring 的事务管理实现原理了。下面简单地介绍下，注解方式为例

- 1. 配置文件开启注解驱动，在相关的类和方法上通过注解@Transactional 标识。
- 2. spring 在启动的时候会去解析生成相关的 bean，这时候会查看拥有相关注解的类和方法，并且为这些类和方法生成代理，并根据@Transaction 的相关参数进行相关配置注入，这样就在代理中为我们把相关的事务处理掉了（开启正常提交事务，异常回滚事务）。
- 3. 真正的数据库层的事务提交和回滚是通过 binlog 或者 redo log 实现的。

8.3、Spring 事务的传播属性

所谓 spring 事务的传播属性，就是定义在存在多个事务同时存在的时候，spring 应该如何处理这些事务的行为。这些属性在 TransactionDefinition 中定义，具体常量的解释见下表：

常量名称	常量解释
PROPAGATION_REQUIRED	支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择，也是 Spring 默认的事务的传播。
PROPAGATION_REQUIRES_NEW	新建事务，如果当前存在事务，把当前事务挂起。新建的事务将和被挂起的事务没有任何关系，是两个独立的事务，外层事务失败回滚之后，不能回滚内层事务执行的结果，内层事务失败抛出异常，外层事务捕获，也可以不处理回滚操作
PROPAGATION_SUPPORTS	支持当前事务，如果当前没有事务，就以非事务方式执行。
PROPAGATION_MANDATORY	支持当前事务，如果当前没有事务，就抛出异常。
PROPAGATION_NOT_SUPPORTED	以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
PROPAGATION_NEVER	以非事务方式执行，如果当前存在事务，则抛出异常。

	常。
PROPAGATION_NESTED	如果一个活动的事务存在，则运行在一个嵌套的事务中。如果没有活动事务，则按 REQUIRED 属性执行。它使用了一个单独的事务，这个事务拥有多个可以回滚的保存点。内部事务的回滚不会对外部事务造成影响。它只对 DataSourceTransactionManager 事务管理器起效。

8.4、数据库隔离级别

隔离级别	隔离级别的值	导致的问题
Read-Uncommitted	0	导致脏读
Read-Committed	1	避免脏读，允许不可重复读和幻读（默认的）
Repeatable-Read	2	避免脏读，不可重复读，允许幻读
Serializable	3	串行化读，事务只能一个一个执行，避免了脏读、不可重复读、幻读。执行效率慢，使用时慎重

脏读：一事务对数据进行了增删改，但未提交，另一事务可以读取到未提交的数据。如果第一个事务这时候回滚了，那么第二个事务就读到了脏数据。

不可重复读：一个事务中发生了两次读操作，第一次读操作和第二次操作之间，另外一个事务对数据进行了修改，这时候两次读取的数据是不一致的。

幻读：第一个事务对一定范围的数据进行批量修改，第二个事务在这个范围增加一条数据，这时候第一个事务就会丢失对新增数据的修改。

总结：

隔离级别越高，越能保证数据的完整性和一致性，但是对并发性能的影响也越大。

大多数的数据库默认隔离级别为 Read Committed，比如 SqlServer、Oracle

少数数据库默认隔离级别为：Repeatable Read 比如： MySQL InnoDB

8.5、Spring 中的隔离级别

常量	解释
ISOLATION_DEFAULT	这是个 PlatfromTransactionManager 默认的隔离级别，使用数据库默认的事务隔离级别。另外四个与 JDBC 的隔离级别相对应。
ISOLATION_READ_UNCOMMITTED	这是事务最低的隔离级别，它充许另外一个事务可以看到这个事务未提交的数据。这种隔离级别会产生脏读，不可重复读和幻像读。
ISOLATION_READ_COMMITTED	保证一个事务修改的数据提交后才能被另外一个事务读取。另外一个事务不能读取该事务未提交的

	数据。
ISOLATION_REPEATABLE_READ	这种事务隔离级别可以防止脏读，不可重复读。但是可能出现幻像读。
ISOLATION_SERIALIZABLE	这是花费最高代价但是最可靠的事务隔离级别。事务被处理为顺序执行。

## 8.6、事务的嵌套

通过上面的理论知识的铺垫，我们大致知道了数据库事务和 spring 事务的一些属性和特点，接下来我们通过分析一些嵌套事务的场景，来深入理解 spring 事务传播的机制。

假设外层事务 Service A 的 Method A() 调用 内层 Service B 的 Method B()

### PROPAGATION\_REQUIRED(spring 默认)

如果 ServiceB.methodB() 的事务级别定义为 PROPAGATION\_REQUIRED，那么执行 ServiceA.methodA() 的时候 spring 已经起了事务，这时调用 ServiceB.methodB()，ServiceB.methodB() 看到自己已经运行在 ServiceA.methodA() 的事务内部，就不再起新的事务。

假如 ServiceB.methodB() 运行的时候发现自己没有在事务中，他就会为自己分配一个事务。

这样，在 ServiceA.methodA() 或者在 ServiceB.methodB() 内的任何地方出现异常，事务都会被回滚。

### PROPAGATION\_REQUIRES\_NEW

比如我们设计 ServiceA.methodA() 的事务级别为 PROPAGATION\_REQUIRED，ServiceB.methodB() 的事务级别为 PROPAGATION\_REQUIRES\_NEW。


那么当执行到 ServiceB.methodB() 的时候，ServiceA.methodA() 所在的事务就会挂起，ServiceB.methodB() 会起一个新的事务，等待 ServiceB.methodB() 的事务完成以后，它才继续执行。

他与 PROPAGATION\_REQUIRED 的事务区别在于事务的回滚程度了。因为 ServiceB.methodB() 是新起一个事务，那么就是存在两个不同的事务。如果 ServiceB.methodB() 已经提交，那么 ServiceA.methodA() 失败回滚，ServiceB.methodB() 是不会回滚的。如果 ServiceB.methodB() 失败回滚，如果他抛出的异常被 ServiceA.methodA() 捕获，ServiceA.methodA() 事务仍然可能提交(主要看 B 抛出的异常是不是 A 会回滚的异常)。

### PROPAGATION\_SUPPORTS

假设 ServiceB.methodB() 的事务级别为 PROPAGATION\_SUPPORTS，那么当执行到 ServiceB.methodB() 时，如果发现 ServiceA.methodA() 已经开启了一个事务，则加入当前的事务，如果发现 ServiceA.methodA() 没有开启事务，则自己也不开启事务。这种时候，内部方法的事务性完全依赖于最外层的事务。

### PROPAGATION\_NESTED

现在的情况就变得比较复杂了，ServiceB.methodB() 的事务属性被配置为 PROPAGATION\_NESTED，此时两者之间又将如何协作呢？  
 ServiceB#methodB 如果 rollback，那么内部事务(即 ServiceB#methodB) 将回滚到它执行前的 SavePoint 而外部事务(即 ServiceA#methodA) 可以有以下两种处理方式：

a、捕获异常，执行异常分支逻辑

```
void methodA() {  
  
    try {  
  
        ServiceB.methodB();  
  
    } catch (SomeException) {  
  
        // 执行其他业务，如 ServiceC.methodC();  
  
    }  
  
}
```

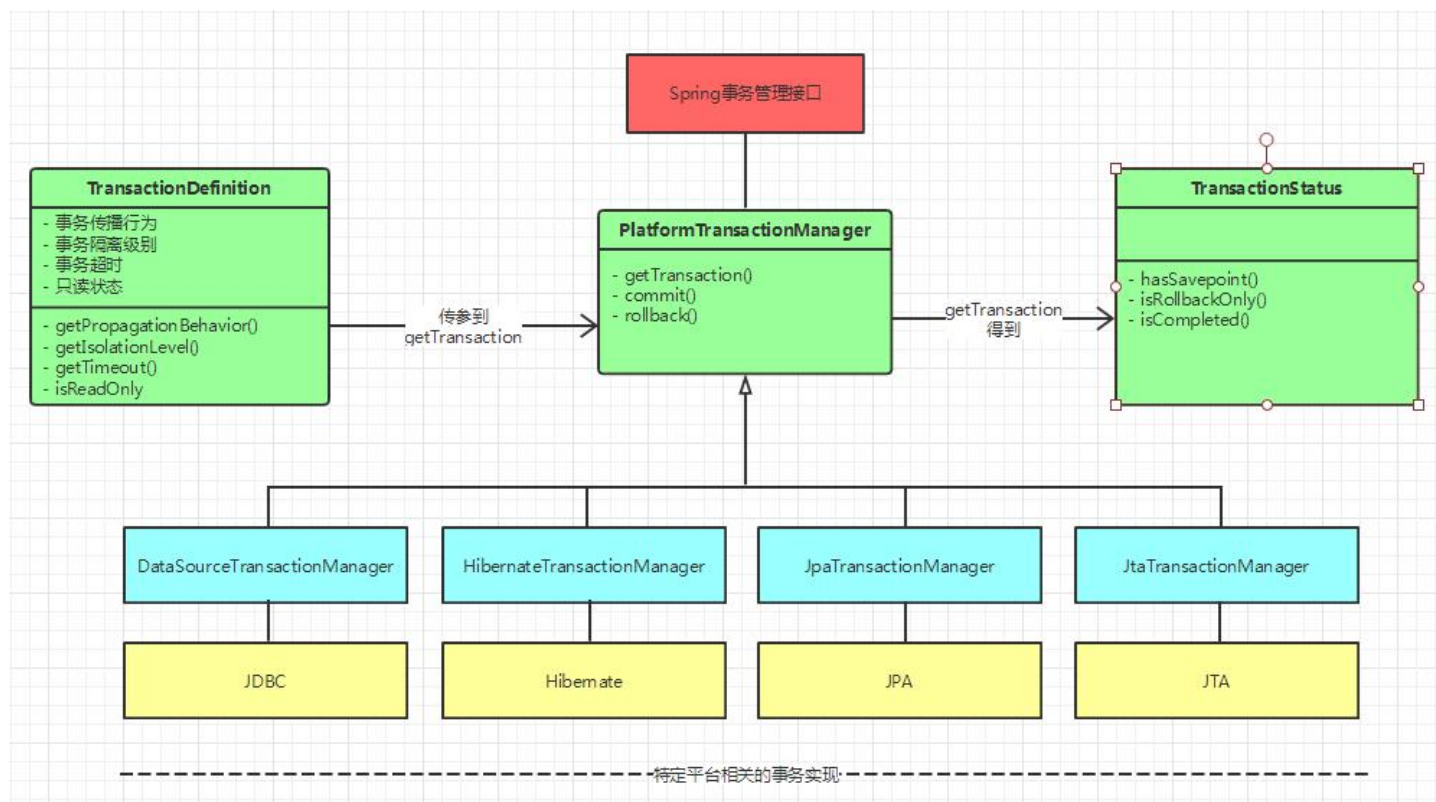
这种方式也是嵌套事务最有价值的地方，它起到了分支执行的效果，如果 ServiceB.methodB 失败，那么执行 ServiceC.methodC()，而 ServiceB.methodB 已经回滚到它执行之前的 SavePoint，所以不会产生脏数据(相当于此方法从未执行过)，这种特性可以用在某些特殊的业务中，而 PROPAGATION\_REQUIRED 和 PROPAGATION\_REQUIRES\_NEW 都没有办法做到这一点。

b、 外部事务回滚/提交 代码不做任何修改，那么如果内部事务(ServiceB#methodB) rollback，那么首先 ServiceB.methodB 回滚到它执行之前的 SavePoint(在任何情况下都会如此)，外部事务(即 ServiceA#methodA) 将根据具体的配置决定自己是 commit 还是 rollback

另外三种事务传播属性基本用不到，在此不做分析。

## 九、Spring 事务源码分析

分析源码之前先来看一张图



紧接着，我们来看一看 Spring 事务是如何配置的，找找程序的入口到底在哪里。通常来说，我们都是这样子来配置的 Spring 声明式事务的。

```

<aop:aspectj-autoproxy proxy-target-class="true"/>

<!--

1、数据源：不管是哪个厂商都要是实现 DataSource 接口，拿到实际上就是包含了 Connection 对象
2、使用 Spring 给我们提供的工具类 TransactionMagager 事务管理器，来管理所有的 事务操作(肯定要拿到连接对象)

-->

<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>

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

<!-- 3、利用切面编程来实现对某一类方法进行事务统一管理(声明式事务) -->
<aop:config>
<aop:pointcut expression="execution(public * com.gupaoedu.vip..*.service..*Service.*(..))"
  
```

```

id="transactionPointcut"/>
    <aop:advisor pointcut-ref="transactionPointcut" advice-ref="transactionAdvice"/>
</aop:config>

<!-- 4、配置通知规则 -->
<tx:advice id="transactionAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="add*" propagation="REQUIRED" rollback-for="Exception, RuntimeException"/>
        <tx:method name="remove*" propagation="REQUIRED"
rollback-for="Exception, RuntimeException"/>
        <tx:method name="modify*" propagation="REQUIRED"
rollback-for="Exception, RuntimeException"/>
        <tx:method name="login" propagation="REQUIRED"/>
        <tx:method name="query*" read-only="true"/>
    </tx:attributes>
</tx:advice>

```

以上的配置相信很多人已经很熟悉了，在此不赘述。而是具体分析一下原理。

先来分析<tx:advice>...</tx:advice>。

tx 是 TransactionNameSpace。对应的是 handler 是 TxNamespaceHandler。

这个类一个 init 方法：

```

public void init() {
    registerBeanDefinitionParser("advice", new TxAdviceBeanDefinitionParser());
    registerBeanDefinitionParser("annotation-driven", new
AnnotationDrivenBeanDefinitionParser());
    registerBeanDefinitionParser("jta-transaction-manager", new
JtaTransactionManagerBeanDefinitionParser());
}

```

这个方法是在 DefaultNamespaceHandlerResolver 的 resolve 中调用的。在为对应的标签寻找 namespacehandler 的时候，调用这个 resolve 方法。resolve 方法先寻找 namespaceUri 对应的 namespacehandler,如果找到了就先调用 Init 方法。

OK. 我们的<tx:advice>对应的解析器也注册了，那就是上面代码里面的。现在这个 parser 的 parse 方法在 NamespaceHandlerSupport 的 parse 方法中被调用了，下面我们来看看这个 TxAdviceBeanDefinitionParser 的 parse 方法吧，这个方法在 TxAdviceBeanDefinitionParser 的祖父类 AbstractBeanDefinitionParser 中：

```

public final BeanDefinition parse(Element element, ParserContext parserContext) {
    //注意这一行
    AbstractBeanDefinition definition = parseInternal(element, parserContext);
    if (definition != null && !parserContext.isNested()) {
        try {
            String id = resolveId(element, definition, parserContext);
            if (!StringUtils.hasText(id)) {
                parserContext.getReaderContext().error(
                    "Id is required for element '" +
parserContext.getDelegate().getLocalName(element)

```



```

        + "' when used as a top-level tag", element);
    }
    String[] aliases = new String[0];
    String name = element.getAttribute(NAME_ATTRIBUTE);
    if (StringUtils.hasLength(name)) {
        aliases =
StringUtils.trimArrayElements(StringUtils.commaDelimitedListToStringArray(name));
    }
    BeanDefinitionHolder holder = new BeanDefinitionHolder(definition, id, aliases);
    registerBeanDefinition(holder, parserContext.getRegistry());
    if (shouldFireEvents()) {
        BeanComponentDefinition componentDefinition = new
BeanComponentDefinition(holder);
        postProcessComponentDefinition(componentDefinition);
        parserContext.registerComponent(componentDefinition);
    }
}
catch (BeanDefinitionStoreException ex) {
    parserContext.getReaderContext().error(ex.getMessage(), element);
    return null;
}
}
return definition;
}
}

```

注意 `parseInternal()` 方法是在 `TxAdviceBeanDefinitionParser` 的父类 `AbstractSingleBeanDefinitionParser` 中实现的，代码如下：

```

protected final AbstractBeanDefinition parseInternal(Element element, ParserContext parserContext)
{
    BeanDefinitionBuilder builder = BeanDefinitionBuilder.genericBeanDefinition();
    String parentName = getParentName(element);
    if (parentName != null) {
        builder.getRawBeanDefinition().setParentName(parentName);
    }
    //获取被代理的对象
    Class<?> beanClass = getBeanClass(element);
    if (beanClass != null) {
        builder.getRawBeanDefinition().setBeanClass(beanClass);
    }
    else {
        String beanClassName = getBeanClassName(element);
        if (beanClassName != null) {
            builder.getRawBeanDefinition().setBeanClassName(beanClassName);
        }
    }
}
}

```

```

builder.getRawBeanDefinition().setSource(parserContext.extractSource(element));
if (parserContext.isNested()) {
    // Inner bean definition must receive same scope as containing bean.
    builder.setScope(parserContext.getContainingBeanDefinition().getScope());
}
if (parserContext.isDefaultLazyInit()) {
    // Default-lazy-init applies to custom bean definitions as well.
    builder.setLazyInit(true);
}
doParse(element, parserContext, builder);
return builder.getBeanDefinition();
}

```

getBeanClass()方法是在 TxAdviceBeanDefinitionParser 中实现的，很简单：

```

protected Class<?> getBeanClass(Element element) {
    return TransactionInterceptor.class;
}

```

至此，这个标签解析的流程已经基本清晰了。那就是：解析除了一个以 TransactionInerceptor 为 classname 的 beandefinition 并且注册这个 bean。剩下来要看的，就是这个 TranscationInterceptor 到底是什么？

看看这个类的接口定义，就明白了：

```

public class TransactionInterceptor extends TransactionAspectSupport implements MethodInterceptor,
Serializable

```

这根本就是一个 spring AOP 的 advice 嘛！现在明白为什么事务的配置能通过 aop 产生作用了吧？

```

public Object invoke(final 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, new InvocationCallback()
{
    public Object proceedWithInvocation() throws Throwable {
        return invocation.proceed();
    }
});
}

```

接下来，我们再来看一下 DataSourceTransactionManager 是如何工作的

```

if (con.getAutoCommit()) {
    txObject.setMustRestoreAutoCommit(true);
    if (logger.isDebugEnabled()) {
        logger.debug("Switching JDBC Connection [" + con + "] to manual commit");
    }
}

```

```
    }  
    con.setAutoCommit(false);  
}
```

doGetTransaction() //从 ThreadLocal 中获取一个 connection（相互对立的）

doBegin() //开启事务

//执行业务逻辑

//根据业务逻辑的执行结果来判断是否要提交还是回滚

//doCommit() 提交

//doRollback() 回滚

---

总结：

### 1、什么是事务？

一个整体的执行逻辑单元，只有两个结果，要么全失败，要么全成功。

### 2、事务的特性

原子性、隔离性、持久性、一致性

### 3、事务的基本原理

从数据库角度来说：就是提供了一种后悔机制（代码写错了，可以 SVN、Git）

用临时表才能实现后悔

执行增、删、改之前，先将满足条件的数据查询出来放入到临时表中

将数据操作现在临时表中完成，完成过程中如果没出现任何问题，就将数据同步(剪切)到实际的数据表中，并返回影响行数

将数据操作现在临时表中完成，完成过程中一旦出现错误，那就将临时表中满足条件的数据清掉，并返回错误码

两个都是增、删、改操作，就会出现死锁(死锁超时，就需要人工解决)

杀进程（写 SQL 语句杀死进程）

如果要想对一个数据表的数据进行清空（千万千万别用 DELETE FROM，这种情况，一定就是锁表）

加了 WHERE 条件，就是行锁

### 4、Spring 的事务配置

AOP 配置，配置哪些方法需要加事务

声明式事务配置，事务的传播属性、隔离级别、回滚的条件

传播属性： DEFAULT REQUIRED REQUIRES\_NEW SUPPORTS NESTED

隔离级别： DEFAULT READ\_UNCOMMITTED READ\_COMMITTED REPEATABLE\_READ SERIALIZABLE

### 5、源码

通过解析配置文件，得到 TransactionDefinition 实际上就是 AOP 中的 MethodInterceptor（方法代理）

就可以在满足条件的方法调用之前和调用之后加一些东西

PlatformTransactionManger 中的方法

getTransaction 调用了 TransactionSynchronizationManager 类的 getResource()

从 ThreadLocal 里面取值，Map<Key:DataSource,Value:ConnectionHolder(相当于获取一个连接对象 Connection);

conn.setAutoCommit(false);

commit conn.commit()

rollback conn.rollback();