课程目标
内容定位
1. 从Spring事务配置说起
    1.1. 先看看Spring事务的基础配置
2. 数据库事务原理详解
    2.1. 事务基本概念
    2.2. 事务的基本原理
    2.3. Spring事务的传播属性
    2.4. 数据库隔离级别
    2.5. Spring中的隔离级别
    2.6. 事务的嵌套
        2.6.1. PROPAGATION_REQUIRED（Spring 默认）
        2.6.2. PROPAGATION_REQUIRES_NEW
        2.6.3. PROPAGATION_SUPPORTS
        2.6.4. PROPAGATION_NESTED
    2.7. Spring事务API架构图
        2.7.1. 异常处理
        2.7.2. config模块
        2.7.3. core模块
            2.7.3.1. JdbcTemplate
            2.7.3.2. RowMapper
            2.7.3.3. 元数据metaData模块
            2.7.3.4. 使用SqlParameterSource提供参数值
            2.7.3.5. simple实现
        2.7.4. DataSource
        2.7.5. object模块
        2.7.6. JdbcTemplate
        2.7.7. NamedParameterJdbcTemplate
3. 浅谈分布式事务
    3.1. 分布式系统的特性
    3.2. 数据一致性理解
4. 补充: Spring的事务操作示例
    4.1. 建立工程spring-tx-lesson
    4.2. 引入依赖
    4.3. 创建entity,dao, service
    4.4. 创建配置文件

4.5. 创建测试类
4.6. SQL

课程目标

1、掌握Spring 事务传播原理。
2、掌握基于Spring的数据库事务操作原理。
3、初步了解分布式事务。

内容定位

1、适合有Spring 开发经验并且希望深度掌握事务控制原理的人群。
2、通过本章内容的学习，能够掌握基于Spring AOP的事务设计原理。

1. 从Spring事务配置说起

1.1. 先看看Spring事务的基础配置

先看看Spring事务的基础配置

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-4.3.xs
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx-4.3.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">
    <aop:aspectj-autoproxy proxy-target-class="true"/>

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

    <!-- 配置事务通知属性 -->
    <tx:advice id="transactionAdvice" transaction-manager="transactionManager">
        <tx:attributes>
            <tx:method name="add*" propagation="REQUIRED" rollback-for="Exception,Runti
            <tx:method name="remove*" propagation="REQUIRED" rollback-for="Exception,Ru
```

```
            <tx:method name="edit*" propagation="REQUIRED" rollback-for="Exception,Runt
            <tx:method name="login" propagation="NOT_SUPPORTED"/>
            <tx:method name="query*" read-only="true"/>
        </tx:attributes>
    </tx:advice>

    <aop:config>
        <aop:advisor advice-ref="transactionAdvice" pointcut-ref="transactionPointcut"/
        <aop:aspect ref="dataSource">
            <aop:pointcut id="transactionPointcut"
                          expression="execution(public * cn.sitedev..*.service..*Servic
        </aop:aspect>
    </aop:config>

</beans>
```

Spring 事务管理基于AOP来实现，主要是统一封装非功能性需求。

2. 数据库事务原理详解

2.1. 事务基本概念

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（Automicity）。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。

2.2. 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 conn = DriverManager.getConnection();`

2、开启事务 `conn.setAutoCommit(true/false);`

3、执行CRUD

4、提交事务/回滚事务 `conn.commit()/conn.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Spirng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，注解方式为例子

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。

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）。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。

2.3. 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。
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。

2.4. 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
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 Commited，比如SqlServer、Oracle
少数数据库默认隔离级别为：Repeatable Read比如：MySQL InnoDB

2.5. 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，它允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。

2.6. 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 ServiceA的MethodA()调用内层ServiceB的MethodB()

2.6.1. 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。

2.6.2. 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）。

2.6.3. 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。

2.6.4. 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：

捕获异常，执行异常分支逻辑

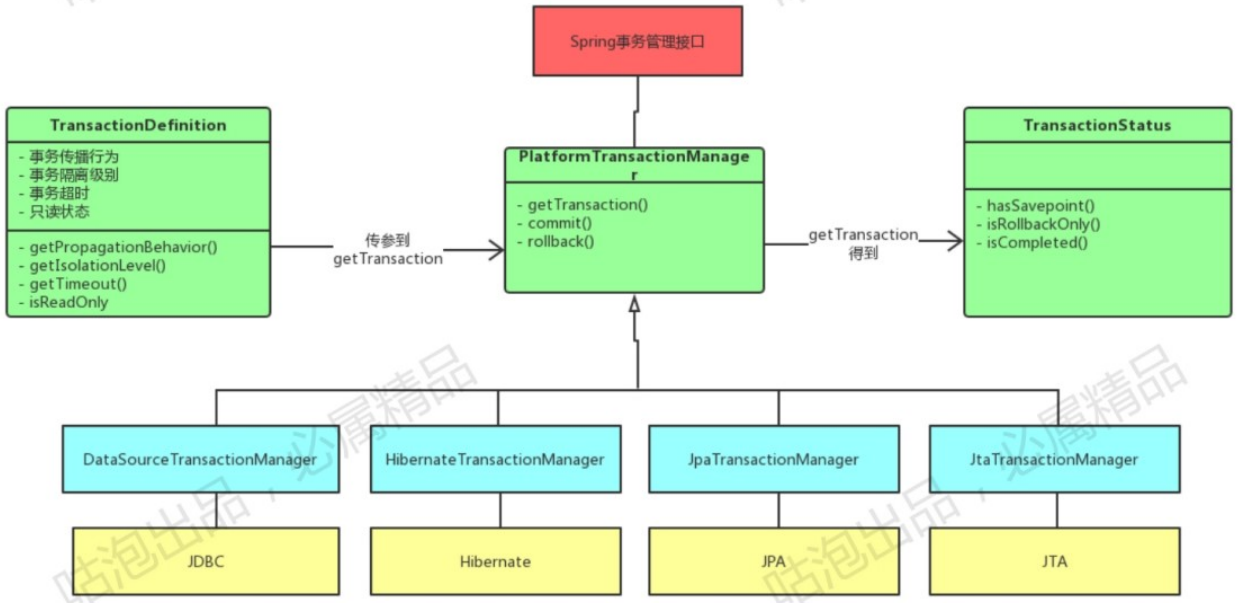
```
void MethodA() {
    try {
        ServiceB.MethodB();
    } catch (SomeException) {
        //执行其他业务，如ServiceC.MethodC();
    }
}
```

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。

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。

2.7. Spring事务API架构图



使用Spring 进行基本的JDBC访问数据库有多种选择。Spring至少提供了三种不同的工作模式：JdbcTemplate，一个在Spring2.5中新提供的SimpleJdbc类能够更好的处理数据库元数据；还有一种称之为RDBMS Object的风格的面向对象封装方式，有点类似于JDO的查询设计。我们在这里简要列举你采取某一种工作方式的主要理由.不过请注意，即使你选择了其中的一种工作模式，你依然可以在你的代码中混用其他任何一种模式以获取其带来的好处和优势。所有的工作模式都必须要求JDBC2.0以上的数据库驱动的支持，其中一些高级的功能可能需要JDBC3.0以上的数据库驱动支持。

JdbcTemplate-这是经典的也是最常用的Spring对于JDBC访问的方案。这也是最低级别的封装，其他的工作模式事实上在底层使用了JdbcTemplate作为其底层的实现基础。JdbcTemplate在JDK1.4以上的环境上工作得很好。

NamedParameterJdbcTemplate-对JdbcTemplate做了封装，提供了更加便捷的基于命名参数的使用方式而不是传统的JDBC所使用的"?"作为参数的占位符。这种方式在你需要为某个SQL指定许多个参数时，显得更加直观而易用。该特性必须工作在JDK1.4以上。

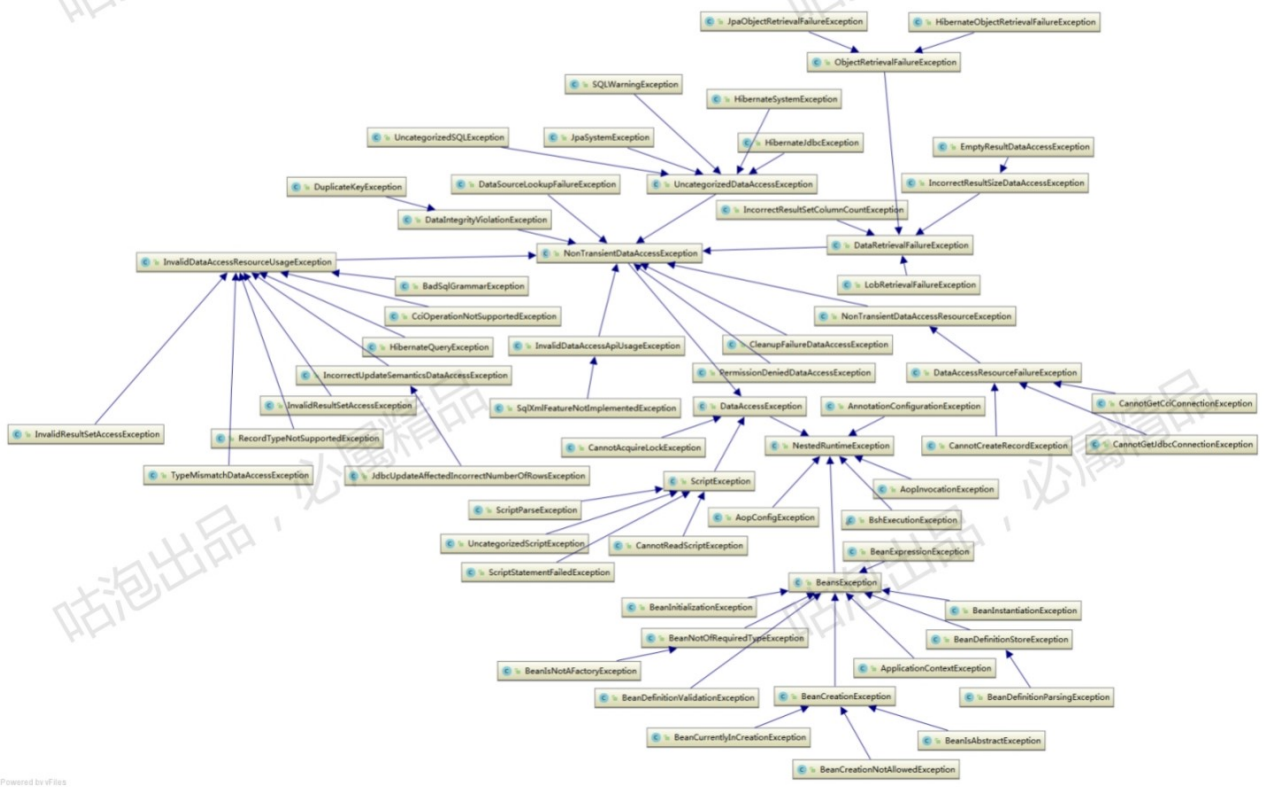
SimpleJdbcTemplate-这个类结合了JdbcTemplate和NamedParameterJdbcTemplate的最常用的功能，同时它也利用了一些Java5的特性所带来的优势，例如泛型、varargs和autoboxing等，从而提供了更加简便的API访问方式。需要工作在Java5以上的环境中。

SimpleJdbcInsert 和SimpleJdbcCall-这两个类可以充分利用数据库元数据的特性来简化配置。通过使用这两个类进行编程，你可以仅仅提供数据库表名或者存储过程的名称以及一个Map作为参数。其中Map的key 需要与数据库表中的字段保持一致。这两个类通常和SimpleJdbcTemplate配合使用。这两个类需要工作在JDK5以上，同时数据库需要提供足够的元数据信息。

RDBMS 对象包括MappingSqlQuery，SqlUpdate and StoredProcedure-这种方式允许你在初始化你的数据访问层时创建可重用并且线程安全的对象。该对象在你定义了你的查询语句，声明查询参数并编译相应的Query之后被模型化。一旦模型化完成，任何执行函数就可以传入不同的参数对之进行多次调用。这种方式需要工作在JDK1.4以上

2.7.1. 异常处理

异常结构如下：



SQLExceptionTranslator 是一个接口，如果你需要在SQLException和 org.springframework.dao.DataAccessException 之间作转换，那么必须实现该接口。

转换器类的实现可以采用一般通用的做法（比如使用JDBC的SQLState code），如果为了使转换更准确，也可以进行定制（比如使用Oracle的error code）。

SQLErrorCodeSQLExceptionTranslator 是SQLExceptionTranslator的默认实现。该实现使用指定数据库厂商的error code，比采用SQLState更精确。转换过程基于一个JavaBean（类型为SQLErrorCodes）中的error code。这个JavaBean由SQLErrorCodesFactory工厂类创建，其中的内容来自于"sql-error-codes.xml"配置文件。该文件中的数据库厂商代码基于Database MetaData信息中的DatabaseProductName，从而配合当前数据库的使用。

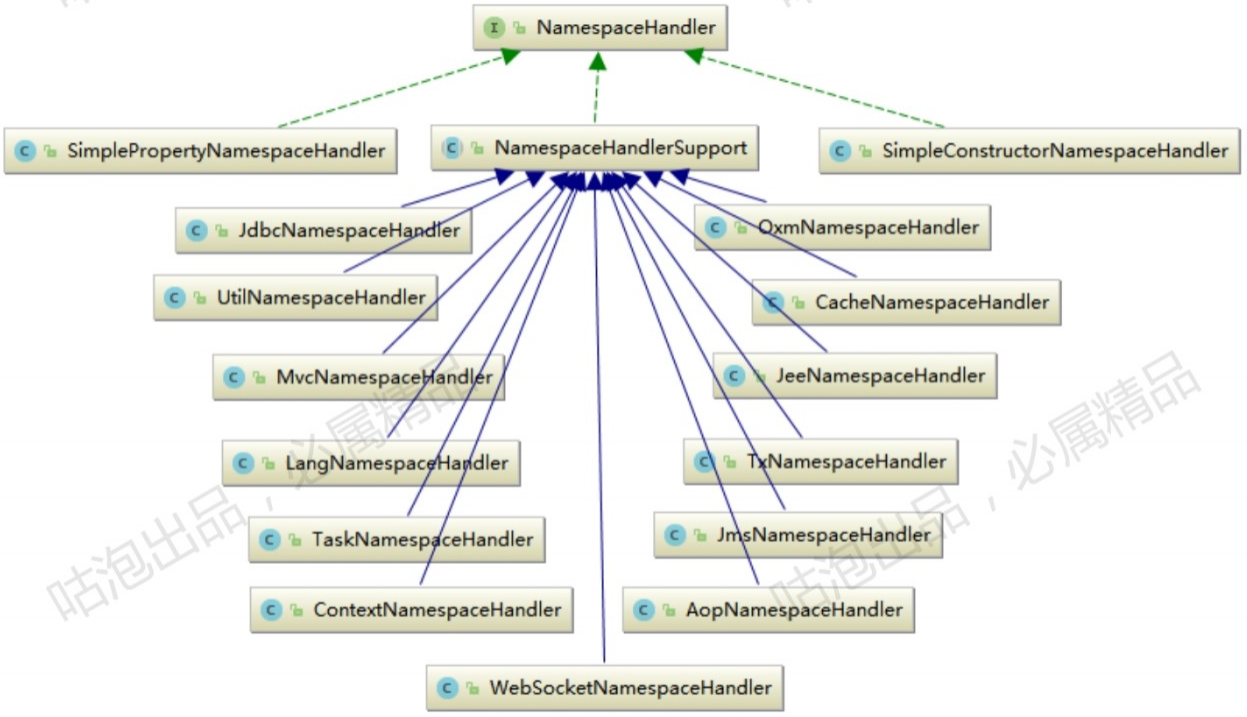
SQLErrorCodeSQLExceptionTranslator 使用以下的匹配规则：

> 首先检查是否存在完成定制转换的子类实现。通常SQLErrorCodeSQLExceptionTranslator 这个类可以作为一个具体类使用，不需要进行定制，那么这个规则将不适用。
>
> 接着将SQLException的 error code与错误代码集中的error code进行匹配。默认情况下错误代码集将从SQLErrorCodesFactory取得。错误代码集来自classpath下的sql-error-codes.xml文件，它们将与数据库metadata信息中的database name进行映射。

> 使用fallback 翻译器。SQLStateSQLExceptionTranslator 类是缺省的falback 翻译器。

2.7.2. config模块

NamespaceHandler 接口，DefaultBeanDefinitionDocumentReader 使用该接口来处理在spring xml配置文件中自定义的命名空间。



在jdbc模块，我们使用JdbcNamespaceHandler来处理jdbc配置的命名空间，其代码如下：

```
public class JdbcNamespaceHandler extends NamespaceHandlerSupport {
    public JdbcNamespaceHandler() {
    }

    public void init() {
        this.registerBeanDefinitionParser("embedded-database", new EmbeddedDatabaseBean
        this.registerBeanDefinitionParser("initialize-database", new InitializeDatabase
    }
}
```

其中，EmbeddedDatabaseBeanDefinitionParser 继承了AbstractBeanDefinitionParser，解析 `<embedded-database>` 元素，并使用EmbeddedDatabaseFactoryBean 创建一个BeanDefinition。顺便介绍一下用到的软件包org.w3c.dom。

软件包org.w3c.dom：为文档对象模型（DOM）提供接口，该模型是Java API for XML Processing的组件API。该Document Object Model Level 2 Core API 允许程序动态访问和更新文档的内容和结构。

Attr:Attr 接口表示Element 对象中的属性。

CDATASection:CDATA节用于转义文本块，该文本块包含的字符如果不转义则会被视为标记。

CharacterData:CharacterData 接口使用属性集合和用于访问DOM中字符数据的方法扩展节点。

Comment：此接口继承自CharacterData表示注释的内容，即起始 `<!--` 和结束 `-->` 之间的所有字符。

Document:Document 接口表示整个HTML或XML文档。

DocumentFragment:DocumentFragment是“轻量级”或“最小”Document对象。

DocumentType：每个 Document都有 doctype属性，该属性的值可以为null，也可以为 DocumentType 对象。

DOMConfiguration：该DOMConfiguration 接口表示文档的配置，并维护一个可识别的参数表。

DOMError:DOMError 是一个描述错误的接口。

DOMErrorHandler:DOMErrorHandler 是在报告处理XML数据时发生的错误或在进行某些其他处理（如验证文档）时DOM实现可以调用的回调接口。

DOMImplementation:DOMImplementation 接口为执行独立于文档对象模型的任何特定实例的操作提供了许多方法。

DOMImplementationList:DOMImplementationList 接口提供对DOM实现的有序集合的抽象，没有定义或约束如何实现此集合。

DOMImplementationSource：此接口允许DOM实现程序根据请求的功能和版本提供一个或多个实现，如下所述。

DOMLocator:DOMLocator 是一个描述位置（如发生错误的位置）的接口。

DOMStringList:DOMStringList 接口提供对DOMString值的有序集合的抽象，没有定义或约束此集合是如何实现的。

Element:Element 接口表示HTML或XML文档中的一个元素。

Entity：此接口表示在XML文档中解析和未解析的已知实体。

EntityReference:EntityReference 节点可以用来在树中表示实体引用。

NamedNodeMap：实现NamedNodeMap接口的对象用于表示可以通过名称访问的节点的集合。

NameList: NameList 接口提供对并行的名称和名称空间值对（可以为null值）的有序集合的抽象，无需定义或约束如何实现此集合。

Node：该Node 接口是整个文档对象模型的主要数据类型。

NodeList:NodeList 接口提供对节点的有序集合的抽象，没有定义或约束如何实现此集合。

Notation：此接口表示在DTD中声明的表示法。

ProcessingInstruction:ProcessingInstruction 接口表示“处理指令”，该指令作为一种在文档的文本中保持特定于处理器的信息的方法在XML中使用。

Text：该Text 接口继承自CharacterData，并且表示Element或Attr的文本内容（在XML中称为字符数据）。

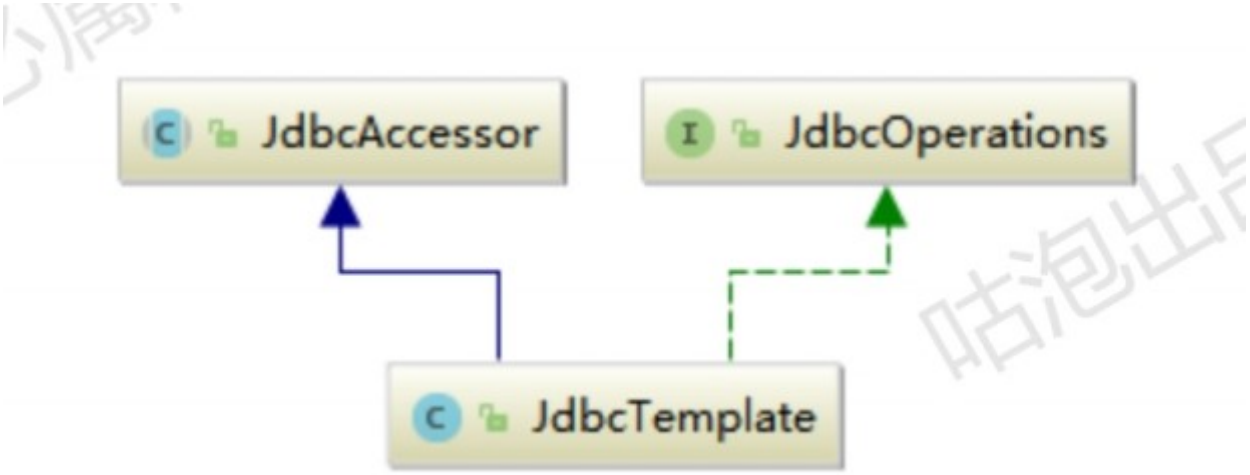
TypeInfo:TypeInfo 接口表示从Element或Attr节点引用的类型，用与文档相关的模式指定。

UserDataHandler：当使用Node.setUserData()将一个对象与节点上的键相关联时，当克隆、导入或重命名该对象关联的节点时应用程序可以提供调用的处理程序。

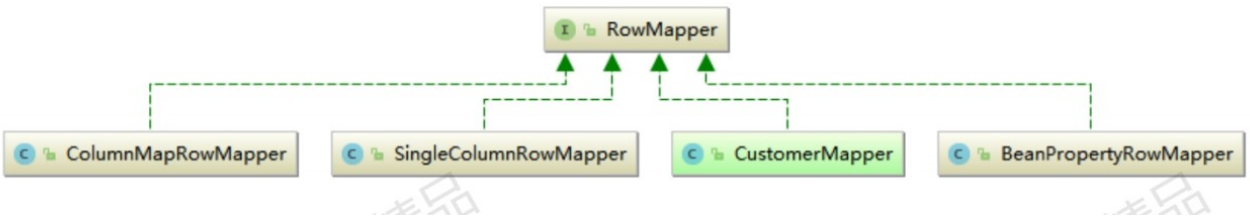
2.7.3. core模块

2.7.3.1. JdbcTemplate

JdbcTemplate对象，其结构如下：

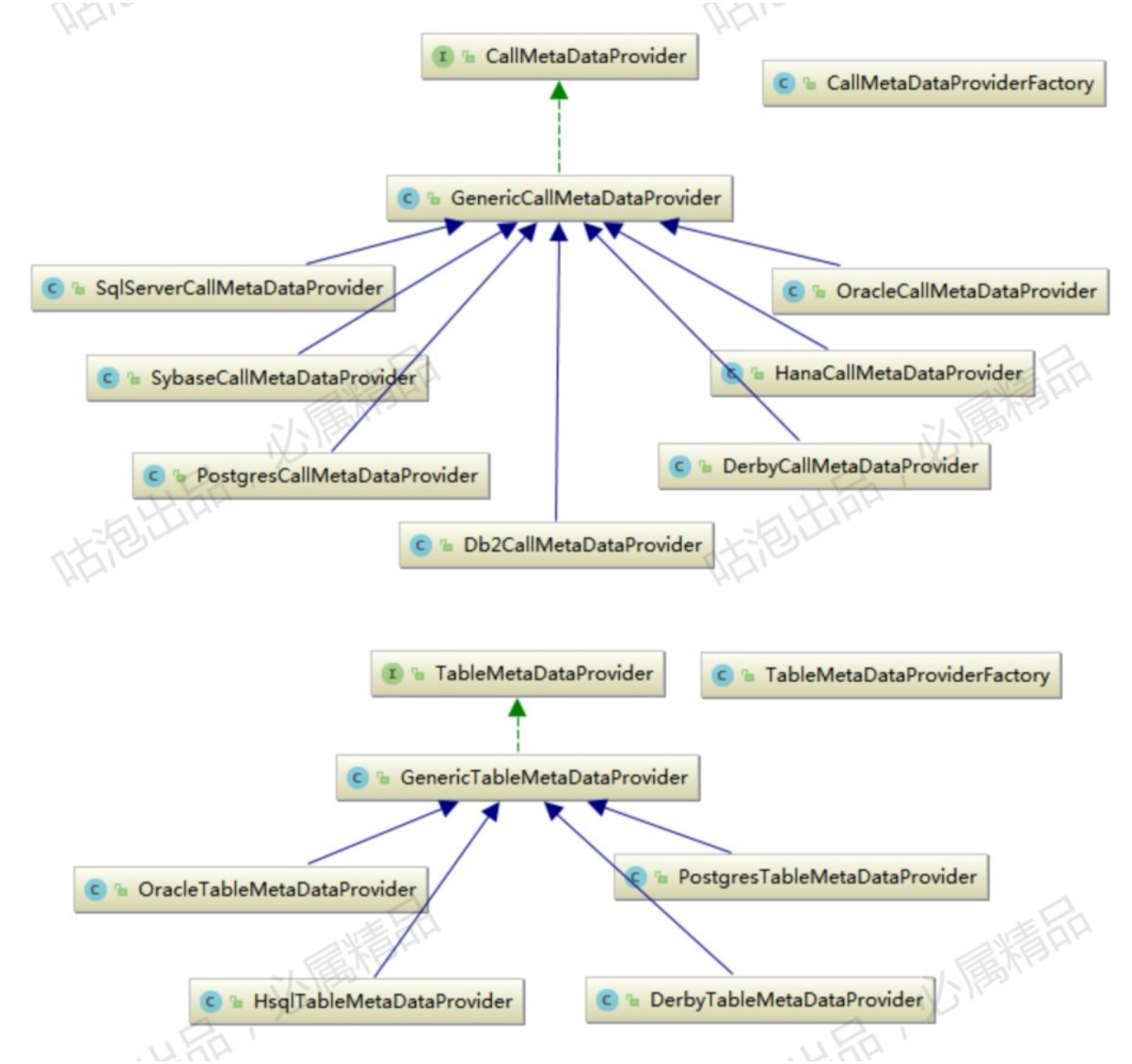


2.7.3.2. RowMapper



2.7.3.3. 元数据metaData模块

本节中spring应用到工厂模式，结合代码可以更具体了解。



CallMetaDataProviderFactory 创建CallMetaDataProvider的工厂类，其代码如下：

```
public class CallMetaDataProviderFactory {
    public static final List<String> supportedDatabaseProductsForProcedures = Arrays.as
    public static final List<String> supportedDatabaseProductsForFunctions = Arrays.asL
    private static final Log logger = LogFactory.getLog(CallMetaDataProviderFactory.cla

    public CallMetaDataProviderFactory() {
    }

    public static CallMetaDataProvider createMetaDataProvider(DataSource dataSource, Ca
        try {
            CallMetaDataProvider result = (CallMetaDataProvider)JdbcUtils.extractDataba
                String databaseProductName = JdbcUtils.commonDatabaseName(databaseMetaD
                boolean accessProcedureColumnMetaData = context.isAccessCallParameterMe
                if (context.isFunction()) {
```

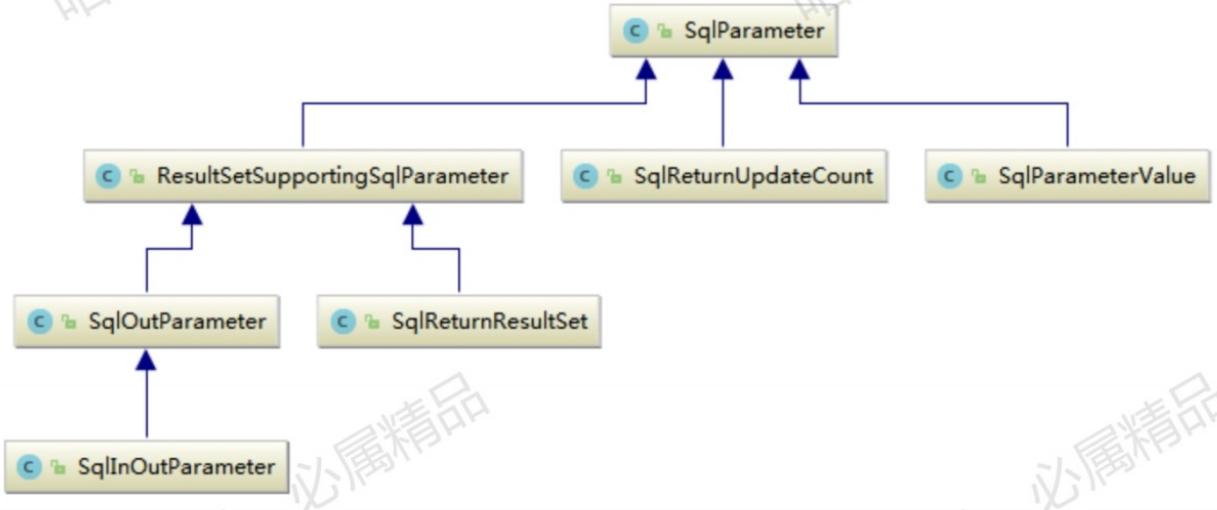
```
15                      if (!supportedDatabaseProductsForFunctions.contains(databaseProduct
16                          if (logger.isWarnEnabled()) {
17                              logger.warn(databaseProductName + " is not one of the datab
18                          }
19
20                          if (accessProcedureColumnMetaData) {
21                              logger.warn("Metadata processing disabled - you must specif
22                              accessProcedureColumnMetaData = false;
23                          }
24                      }
25                  } else if (!supportedDatabaseProductsForProcedures.contains(databasePro
26                      if (logger.isWarnEnabled()) {
27                          logger.warn(databaseProductName + " is not one of the databases
28                      }
29
30                      if (accessProcedureColumnMetaData) {
31                          logger.warn("Metadata processing disabled - you must specify al
32                          accessProcedureColumnMetaData = false;
33                      }
34                  }
35
36                  Object provider;
37                  if ("Oracle".equals(databaseProductName)) {
38                      provider = new OracleCallMetaDataProvider(databaseMetaData);
39                  } else if ("DB2".equals(databaseProductName)) {
40                      provider = new Db2CallMetaDataProvider(databaseMetaData);
41                  } else if ("Apache Derby".equals(databaseProductName)) {
42                      provider = new DerbyCallMetaDataProvider(databaseMetaData);
43                  } else if ("PostgreSQL".equals(databaseProductName)) {
44                      provider = new PostgresCallMetaDataProvider(databaseMetaData);
45                  } else if ("Sybase".equals(databaseProductName)) {
46                      provider = new SybaseCallMetaDataProvider(databaseMetaData);
47                  } else if ("Microsoft SQL Server".equals(databaseProductName)) {
48                      provider = new SqlServerCallMetaDataProvider(databaseMetaData);
49                  } else if ("HDB".equals(databaseProductName)) {
50                      provider = new HanaCallMetaDataProvider(databaseMetaData);
51                  } else {
52                      provider = new GenericCallMetaDataProvider(databaseMetaData);
53                  }
54
55                  if (logger.isDebugEnabled()) {
56                      logger.debug("Using " + provider.getClass().getName());
57                  }
```

```
58
59                  ((CallMetaDataProvider)provider).initializeWithMetaData(databaseMetaDat
60                  if (accessProcedureColumnMetaData) {
61                      ((CallMetaDataProvider)provider).initializeWithProcedureColumnMetaD
62                  }
63
64                  return provider;
65              });
66              return result;
67          } catch (MetaDataAccessException var3) {
68              throw new DataAccessResourceFailureException("Error retrieving database met
69          }
70      }
71  }
```

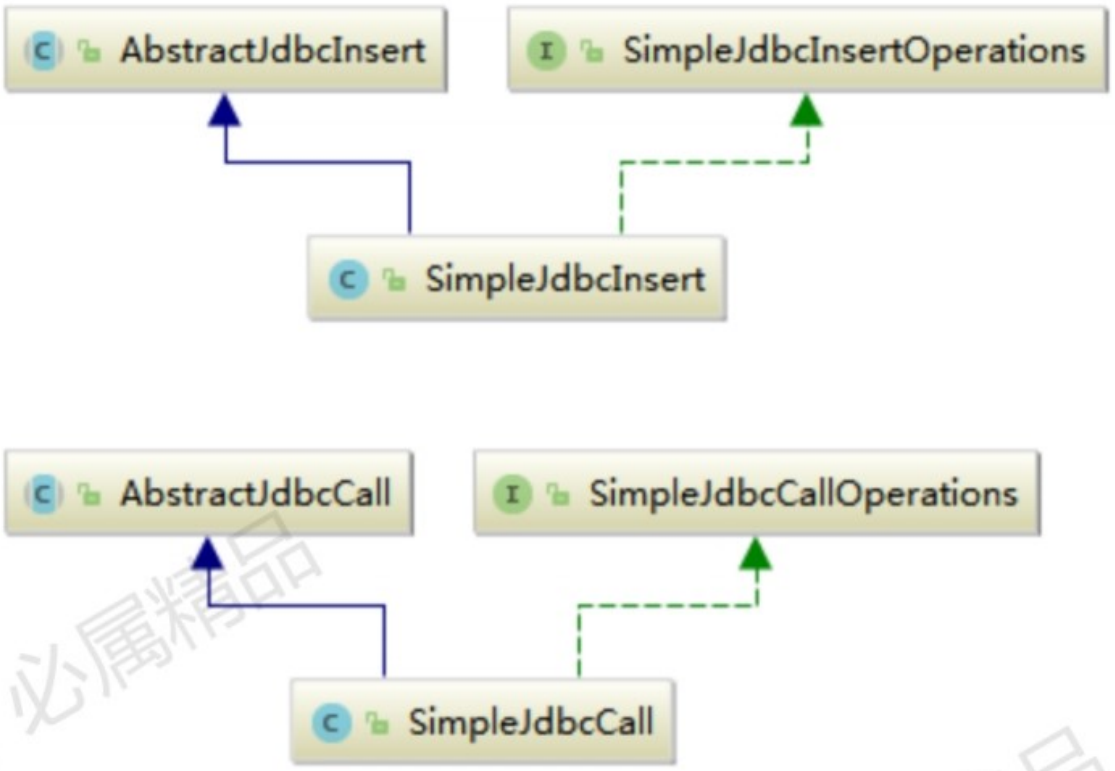
2.7.3.4. 使用SqlParameterSource提供参数值

使用Map来指定参数值有时候工作得非常好，但是这并不是最简单的使用方式。Spring提供了一些其他的SqlParameterSource实现类来指定参数值。我们首先可以看看BeanPropertySqlParameterSource类，这是一个非常简便的指定参数的实现类，只要你有一个符合JavaBean规范的类就行了。它将使用其中的 getter方法来获取参数值。

SqlParameter 封装了定义sql参数的对象。CallableStateMentCallback，PrePareStateMentCallback，StateMentCallback，ConnectionCallback 回调类分别对应JdbcTemplate中的不同处理方法。



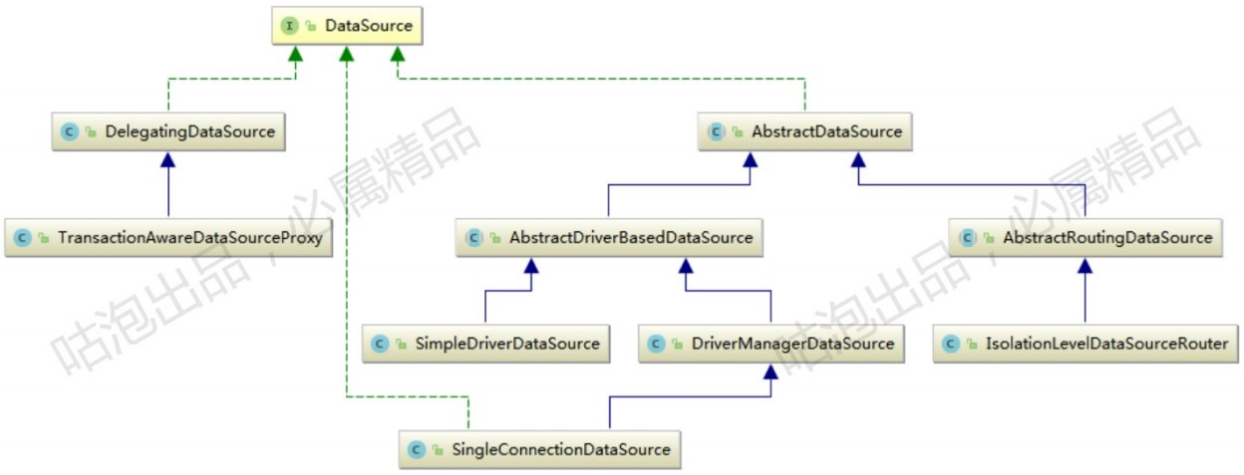
2.7.3.5. simple实现



2.7.4. DataSource

spring 通过DataSource获取数据库的连接。Datasource是jdbc规范的一部分，它通过 ConnectionFactory 获取。一个容器和框架可以在应用代码层中隐藏连接池和事务管理。

当使用spring的jdbc层，你可以通过JNDI来获取 DataSource，也可以通过你自己配置的第三方连接池实现来获取。流行的第三方实现有apache Jakarta Commons dbcp和c3p0。

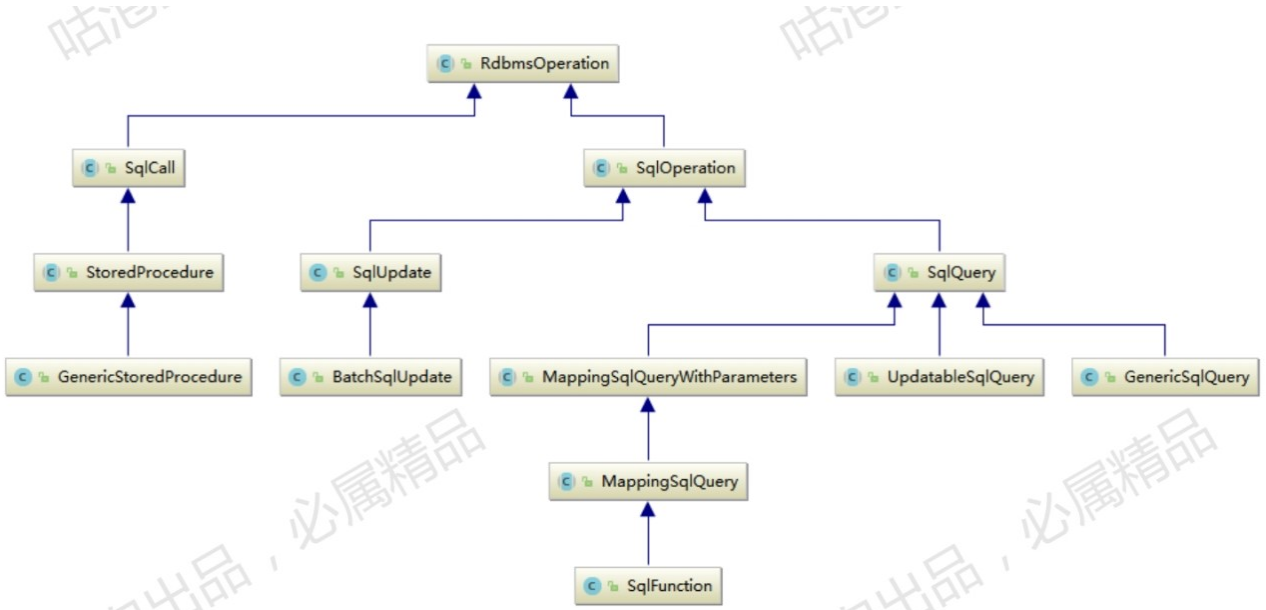


TransactionAwareDataSourceProxy作为目标 DataSource的一个代理，在对目标DataSource包装的同时，还增加了Spring的事务管理能力，在这一点上，这个类的功能非常像J2EE服务器所提供的事务化的JNDI DataSource。

NOTE
该类几乎很少被用到，除非现有代码在被调用的时候需要一个标准的JDBC DataSource接口实现作为参数。这种情况下，这个类可以使现有代码参与Spring的事务管理。通常最好的做法是使用更高层的抽象来对数据源进行管理，比如JdbcTemplate和DataSourceUtils等等。

注意：DriverManagerDataSource仅限于测试使用，因为它没有提供池的功能，这会导致在多个请求获取连接时性能很差。

2.7.5. object模块



2.7.6. JdbcTemplate

JdbcTemplate是core包的核心类。它替我们完成了资源的创建以及释放工作，从而简化了我们对JDBC的使用。它还可以帮助我们避免一些常见的错误，比如忘记关闭数据库连接。JdbcTemplate将完成JDBC核心处理流程，比如SQL 语句的创建、执行，而把SQL语句的生成以及查询结果的提取工作留给我们的应用代码。它可以完成SQL查询、更新以及调用存储过程可以对ResultSet进行遍历并加以提取。它还可以捕获JDBC异常并将其转换成org.springframework.dao包中定义的通用的，信息更丰富的异常。

使用JdbcTemplate进行编码只需要根据明确定义的一组契约来实现回调接口。

PreparedStatementCreator 回调接口通过给定的Connection创建一个PreparedStatement，包含SQL和任何相关的参数。

CallableStatementCreator实现同样的处理，只不过它创建的是CallableStatement。
RowCallbackHandler 接口则从数据集的每一行中提取值。

我们可以在DAO实现类中通过传递一个DataSource引用来完成JdbcTemplate的实例化，也可以在Spring的IOC容器中配置一个JdbcTemplate的bean并赋予DAO实现类作为一个实例。需要注意的是DataSource在Spring的IOC容器中总是配置成一个bean，第一种情况下，DataSource bean 将传递给 service，第二种情况下DataSource bean 传递给JdbcTemplate bean。

2.7.7. NamedParameterJdbcTemplate

NamedParameterJdbcTemplate 类为JDBC操作增加了命名参数的特性支持，而不是传统的使用（"`?`"）作为参数的占位符。

NamedParameterJdbcTemplate类对JdbcTemplate 类进行了封装，在底层，JdbcTemplate完成了多数的工作。

3. 浅谈分布式事务

现今互联网界，分布式系统和微服务架构盛行。一个简单操作，在服务端非常可能是由多个服务和数据库实例协同完成的。在一致性要求较高的场景下，多个独立操作之间的一致性问题显得格外棘手。

基于水平扩容能力和成本考虑，传统的强一致的解决方案（e.g.单机事务）纷纷被抛弃。

其理论依据就是响当当的CAP原理。往往为了可用性和分区容错性，忍痛放弃强一致支持，转而追求最终一致性。

3.1. 分布式系统的特性

在分布式系统中，同时满足CAP定律中的一致性Consistency、可用性Availability和分区容错性Partition Tolerance三者是不可能的。

在绝大多数的场景，都需要牺牲强一致性来换取系统的高可用性，系统往往只需要保证最终一致性。

分布式事务服务（Distributed Transaction Service，DTS）是一个分布式事务框架，用来保障在大规模分布式环境下事务的最终一致性。
CAP理论告诉我们在分布式存储系统中，最多只能实现上面的两点。而由于当前的网络硬件肯定会出现延迟丢包等问题，所以分区容错性是我们必须需要实现的，所以我们只能在一致性和可用性之间进行权衡。

为了保障系统的可用性，互联网系统大多将强一致性需求转换成最终一致性的需求，并通过系统执行幂等性的保证，保证数据的最终一致性。

3.2. 数据一致性理解

强一致性：当更新操作完成之后，任何多个后续进程或者线程的访问都会返回最新的更新过的值。这种是对用户最友好的，就是用户上一次写什么，下一次就保证能读到什么。

根据CAP理论，这种实现需要牺牲可用性。

弱一致性：系统并不保证后续进程或者线程的访问都会返回最新的更新过的值。系统在数据写入成功之后，不承诺立即可以读到最新写入的值，也不会具体的承诺多久之后可以读到。

最终一致性：弱一致性的特定形式。系统保证在没有后续更新的前提下，系统最终返回上一次更新操作的值。在没有故障发生的前提下，不一致窗口的时间主要受通信延迟，系统负载和复制副本的个数影响。DNS是一个典型的最终一致性系统。

4. 补充: Spring的事务操作示例

4.1. 建立工程spring-tx-lesson

4.2. 引入依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
    <parent>
        <artifactId>chap02_01_11_Spring_Tx</artifactId>
        <groupId>cn.sitedev</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <packaging>war</packaging>
    <artifactId>spring-tx-lesson</artifactId>
    <;properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <!-- dependency versions -->
        <spring.version>5.0.2.RELEASE</spring.version>
    </properties>

    <dependencies>
        <!-- spring framework start -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-beans</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aop</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context-support</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
```

```
            <groupId>org.springframework</groupId>
            <artifactId>spring-expression</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-tx</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>${spring.version}</version>
            <scope>test</scope>
        </dependency>
        <!-- spring framework end -->

        <!-- requied start -->

        <dependency>
            <groupId>aopalliance</groupId>
            <artifactId>aopalliance</artifactId>
            <version>1.0</version>
        </dependency>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
            <version>1.6.2</version>
        </dependency>
        <dependency>
            <groupId>cglib</groupId>
            <artifactId>cglib-nodep</artifactId>
            <version>3.1</version>
        </dependency>

        <!-- requied end -->


```

```
        <!-- apache commons start -->
        <dependency>
            <groupId>commons-beanutils</groupId>
            <artifactId>commons-beanutils</artifactId>
            <version>1.8.3</version>
        </dependency>
        <dependency>
            <groupId>commons-collections</groupId>
            <artifactId>commons-collections</artifactId>
            <version>3.2.1</version>
        </dependency>
        <dependency>
            <groupId>commons-fileupload</groupId>
            <artifactId>commons-fileupload</artifactId>
            <version>1.3</version>
        </dependency>
        <dependency>
            <groupId>commons-io</groupId>
            <artifactId>commons-io</artifactId>
            <version>2.4</version>
        </dependency>
        <dependency>
            <groupId>commons-lang</groupId>
            <artifactId>commons-lang</artifactId>
            <version>2.6</version>
        </dependency>
        <dependency>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
            <version>1.2</version>
        </dependency>
        <!-- apache commons end -->


        <!-- jdbc driver start -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.14</version>
        </dependency>
        <!-- jdbc driver end -->

        <!-- others start -->
```

```
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.17</version>
        </dependency>
        <dependency>
            <groupId>javax.persistence</groupId>
            <artifactId>persistence-api</artifactId>
            <version>1.0</version>
        </dependency>
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>fastjson</artifactId>
            <version>1.2.4</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.0.9</version>
        </dependency>

    </dependencies>

</project>
```

4.3. 创建entity,dao, service

```
package cn.sitedev.spring.transaction.entity;

import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
```

```java
import java.io.Serializable;

@Entity
@Table(name = "t_member")
@Data
@NoArgsConstructor
public class Member implements Serializable {
    @Id
    private Long id;
    private String name;
    private String addr;
    private Integer age;

    public Member(String name, String addr, Integer age) {
        this.name = name;
        this.addr = addr;
        this.age = age;
    }
}
//////////////////////////////////////////////
package cn.sitedev.spring.transaction.dao;

import cn.sitedev.spring.transaction.entity.Member;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;

import javax.annotation.Resource;
import javax.sql.DataSource;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

@Repository
public class MemberDao {
    private JdbcTemplate jdbcTemplate;

    @Resource(name = "dataSource")
    protected void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public List<Member> selectAll() throws Exception {
```

```
        String sql = "SELECT * FROM t_member";
        return this.jdbcTemplate.query(sql, new RowMapper<Member>() {
            @Override
            public Member mapRow(ResultSet resultSet, int i) throws SQLException {
                Member member = new Member();
                member.setName(resultSet.getString("name"));
                member.setId(resultSet.getLong("id"));
                member.setAddr(resultSet.getString("addr"));
                member.setAge(resultSet.getInt("age"));
                return member;
            }
        });
    }

    public boolean insert(Member member) throws Exception {
        String sql = "INSERT INTO t_member(id, name, addr, age) VALUES (?, ?, ?, ?)";
        int count = this.jdbcTemplate.update(sql, member.getId(), member.getName(), mem
        return count > 0;
    }

    public boolean delete(long id) throws Exception {
        return this.jdbcTemplate.update("DELETE FROM t_member WHERE id = ?", id) > 0;
    }

    public boolean update(long id, String name) throws Exception {
        return this.jdbcTemplate.update("UPDATE t_member SET name = ? WHERE id = ?", na
    }
}
////////////////////////////////////////////////////////////////////
package cn.sitedev.spring.transaction.service;

import cn.sitedev.spring.transaction.dao.MemberDao;
import cn.sitedev.spring.transaction.entity.Member;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class MemberService {
    @Autowired
    private MemberDao memberDao;

```

```
95      public List<Member> queryAll() throws Exception {
96          return this.memberDao.selectAll();
97      }
98
99      public boolean add(Member member) throws Exception {
100         boolean flag = this.memberDao.insert(member);
101         throwEx(true);
102         return flag;
103     }
104
105     public boolean remove(long id) throws Exception {
106         boolean flag = this.memberDao.delete(id);
107         throwEx(true);
108         return flag;
109     }
110
111     public boolean modify(long id, String name) throws Exception {
112         return this.memberDao.update(id, name);
113     }
114
115     public boolean login(long id, String name) throws Exception {
116         boolean flag = this.modify(id, name);
117         throwEx(true);
118         return flag;
119     }
120
121     private void throwEx(boolean throwEx) throws Exception {
122         if (throwEx) {
123             throw new Exception("手动抛出异常");
124         }
125     }
126 }
```

4.4. 创建配置文件

log4j.properties

```
1 ### set log levels ###
2 log4j.rootLogger=INFO , console , debug , error
3 ### console ###
4 log4j.appender.console=org.apache.log4j.ConsoleAppender
```

```
log4j.appender.console.Target=System.out
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%-d{yyyy-MM-dd HH\:mm\:ss} [%p]-[%c] %m
### log file ###
log4j.appender.debug=org.apache.log4j.DailyRollingFileAppender
log4j.appender.debug.File=../logs/debug.log
log4j.appender.debug.Append=true
log4j.appender.debug.Threshold=INFO
log4j.appender.debug.layout=org.apache.log4j.PatternLayout
log4j.appender.debug.layout.ConversionPattern=%-d{yyyy-MM-dd HH\:mm\:ss} [%p]-[%c] %m%n
### exception ###
log4j.appender.error=org.apache.log4j.DailyRollingFileAppender
log4j.appender.error.File=../logs/error.log
log4j.appender.error.Append=true
log4j.appender.error.Threshold=ERROR
log4j.appender.error.layout=org.apache.log4j.PatternLayout
log4j.appender.error.layout.ConversionPattern=%-d{yyyy-MM-dd HH\:mm\:ss} [%p]-[%c] %m%n
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ISO8601} %l %c%n%p: %m%n
```

db.properties

```
#sysbase database mysql config

mysql.jdbc.driverClassName=com.mysql.jdbc.Driver
mysql.jdbc.url=jdbc:mysql://127.0.0.1:3306/spring-tx?characterEncoding=UTF-8&rewriteBat
mysql.jdbc.username=root
mysql.jdbc.password=root

#alibaba druid config
dbPool.initialSize=1
dbPool.minIdle=1
dbPool.maxActive=200
dbPool.maxWait=60000
dbPool.timeBetweenEvictionRunsMillis=60000
dbPool.minEvictableIdleTimeMillis=300000
dbPool.validationQuery=SELECT 'x'
dbPool.testWhileIdle=true
dbPool.testOnBorrow=false
```

```
dbPool.testOnReturn=false
dbPool.poolPreparedStatements=false
dbPool.maxPoolPreparedStatementPerConnectionSize=20
dbPool.filters=stat,log4j,wall
```

application-db.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.sprin
        http://www.springframework.org/schema/tx http://www.springframework.org/schema/
        http://www.springframework.org/schema/context http://www.springframework.org/sc
           http://www.springframework.org/schema/aop http://www.springframework.org/sch

    <bean id="datasourcePool" abstract="true" class="com.alibaba.druid.pool.DruidDataSo
          destroy-method="close">
        <property name="initialSize" value="${dbPool.initialSize}"/>
        <property name="minIdle" value="${dbPool.minIdle}"/>
        <property name="maxActive" value="${dbPool.maxActive}"/>
        <property name="maxWait" value="${dbPool.maxWait}"/>
        <property name="timeBetweenEvictionRunsMillis" value="${dbPool.timeBetweenEvict
        <property name="minEvictableIdleTimeMillis" value="${dbPool.minEvictableIdleTim
        <property name="validationQuery" value="${dbPool.validationQuery}"/>
        <property name="testWhileIdle" value="${dbPool.testWhileIdle}"/>
        <property name="testOnBorrow" value="${dbPool.testOnBorrow}"/>
        <property name="testOnReturn" value="${dbPool.testOnReturn}"/>
        <property name="poolPreparedStatements" value="${dbPool.poolPreparedStatements}
        <property name="maxPoolPreparedStatementPerConnectionSize"
                  value="${dbPool.maxPoolPreparedStatementPerConnectionSize}"/>
        <property name="filters" value="${dbPool.filters}"/>

    </bean>

    <bean id="dataSource" parent="datasourcePool">
        <property name="driverClassName" value="${mysql.jdbc.driverClassName}"/>
        <property name="url" value="${mysql.jdbc.url}"/>
        <property name="username" value="${mysql.jdbc.username}"/>
```

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

    </bean>
</beans>
```

application-common.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.sprin
       http://www.springframework.org/schema/util http://www.springframework.org/schema
       http://www.springframework.org/schema/context http://www.springframework.org/sch

    <context:component-scan base-package="cn.sitedev"/>
    <context:annotation-config/>

    <bean id="propertyConfigurer" class="org.springframework.beans.factory.config.Prope
        <property name="locations">
            <list>
                <value>classpath:db.properties</value>
            </list>
        </property>
    </bean>

</beans>
```

application-beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.sprin

</beans>
```

application-aop.xml

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-4.3.xs
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx-4.3.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">
    <aop:aspectj-autoproxy proxy-target-class="true"/>

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

    <!-- 配置事务通知属性 -->
    <tx:advice id="transactionAdvice" transaction-manager="transactionManager">
        <tx:attributes>
            <tx:method name="add*" propagation="REQUIRED" rollback-for="Exception,Runti
            <tx:method name="remove*" propagation="REQUIRED" rollback-for="Exception,Ru
            <tx:method name="edit*" propagation="REQUIRED" rollback-for="Exception,Runt
            <tx:method name="login" propagation="NOT_SUPPORTED"/>
            <tx:method name="query*" read-only="true"/>
        </tx:attributes>
    </tx:advice>

    <aop:config>
        <aop:advisor advice-ref="transactionAdvice" pointcut-ref="transactionPointcut"/
        <aop:aspect ref="dataSource">
            <aop:pointcut id="transactionPointcut"
                          expression="execution(public * cn.sitedev..*.service..*Servic
        </aop:aspect>
    </aop:config>

</beans>
```

application-context.xml

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

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.sprin

    <import resource="classpath:application-beans.xml"/>
    <import resource="classpath:application-common.xml"/>
    <import resource="classpath:application-db.xml"/>
    <import resource="classpath:application-aop.xml"/>
</beans>
```

4.5. 创建测试类

```java

package cn.sitedev.spring.transaction.service;

import cn.sitedev.spring.transaction.entity.Member;
import com.alibaba.fastjson.JSON;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import java.util.List;

@ContextConfiguration(locations = {"classpath*:application-context.xml"})
@RunWith(SpringJUnit4ClassRunner.class)
public class MemberServiceTest {
    @Autowired
    private MemberService memberService;

    @Test
    public void queryAll() {
        List<Member> list = null;
        try {
            list = memberService.queryAll();
            System.out.println(JSON.toJSONString(list, true));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```java

    @Test
    public void testRemove() {
        try {
            boolean flag = memberService.remove(1L);
            System.out.println(flag);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    @Test
    public void testLogin() {
        try {
            memberService.login(1L, "sitedevvvvvv");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    @Test
    public void testAdd() {
        Member member = new Member("sitedev", "China", 18);
        try {
            memberService.add(member);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

4.6. SQL

```sql
/*
 Navicat MySQL Data Transfer

 Source Server         : LOCALHOST
 Source Server Type    : MySQL
 Source Server Version : 50717
 Source Host           : localhost:3306
```

```
 Source Schema         : gp-vip-spring-tx

 Target Server Type    : MySQL
 Target Server Version : 50717
 File Encoding         : 65001

 Date: 17/04/2019 17:30:02
*/

SET NAMES utf8mb4;
SET FOREIGN_KEY_CHECKS = 0;

-- ----------------------------
-- Table structure for t_member
-- ----------------------------
DROP TABLE IF EXISTS `t_member`;
CREATE TABLE `t_member`  (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) CHARACTER SET utf8 COLLATE utf8_unicode_ci NULL DEFAULT NULL,
  `age` int(11) NULL DEFAULT NULL,
  `addr` varchar(255) CHARACTER SET utf8 COLLATE utf8_unicode_ci NULL DEFAULT NULL,
  PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB AUTO_INCREMENT = 1 CHARACTER SET = utf8 COLLATE = utf8_unicode_ci ROW

SET FOREIGN_KEY_CHECKS = 1;
```