咕泡学院 JavaVIP 高级课程教案

Spring5 源码分析(第 2 版)

第三章

Spring 数据访问篇

关于本文档

主题	咕泡学院 Java VIP 高级课程教案--Spring5 源码分析（第二版）
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	Spring 5.0.2.RELEASE
IDE 版本	IntelliJ IDEA 2017.1.4

五、Spring 事务原理详解

5.1、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（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。

5.2、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：
获取连接 Connection con = DriverManager.getConnection()
开启事务 con.setAutoCommit(true/false);
执行 CRUD
提交事务/回滚事务 con.commit() / con.rollback();
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。

5.3、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。

5.4、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

5.5、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。

5.6、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：

捕获异常，执行异常分支逻辑

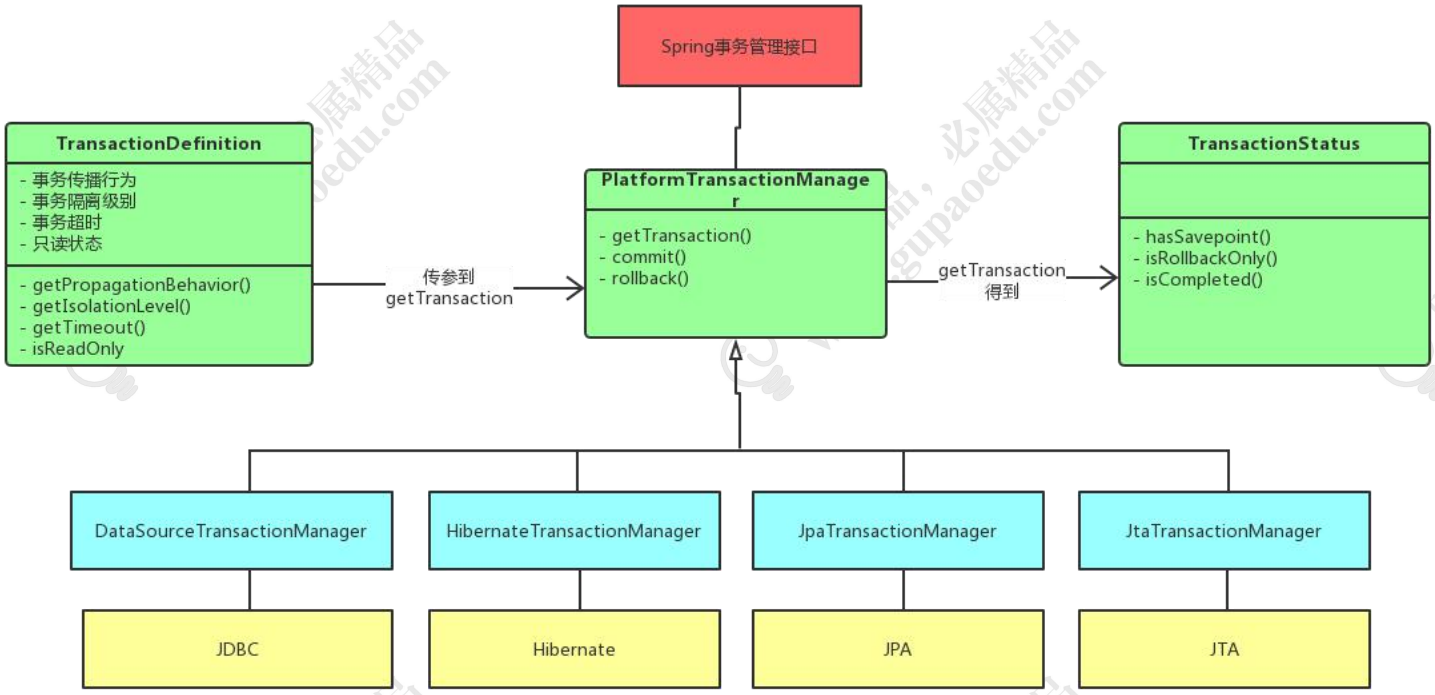
```
void MethodA() {
   try {
      ServiceB.MethodB();
   } catch (SomeException) {
      // 执行其他业务, 如 ServiceC.MethodC();
   }
}
```

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。

5.7、Spring 事务 API 架构图



5.8、Spring AOP 设计原理及应用场景

4.7.1、SpringAOP 应用示例

AOP 是 OOP 的延续，是 Aspect Oriented Programming 的缩写，意思是面向切面编程。可以通过预编译方式和运行期动态代理实现在不修改源代码的情况下给程序动态统一添加功能的一种技术。AOP 设计模式孜孜不倦追求的是调用者和被调用者之间的解耦，AOP 可以说也是这种目标的一种实现。

我们现在做的一些非业务，如：日志、事务、安全等都会写在业务代码中(也即是说，这些非业务类横切于业务类)，但这些代码往往是重复，复制——粘贴式的代码会给程序的维护带来不便，AOP 就实现了把这些业务需求与系统需求分开来做。这种解决的方式也称代理机制。

先来了解一下 AOP 的相关概念

切面（Aspect）：官方的抽象定义为“一个关注点的模块化，这个关注点可能会横切多个对象”。“切面”在 ApplicationContext 中<aop:aspect>来配置。

连接点（Joinpoint）：程序执行过程中的某一行为，例如，MemberService .get 的调用或者 MemberService .delete 抛出异常等行为。

通知（Advice）：“切面”对于某个“连接点”所产生的动作。其中，一个“切面”可以包含多个“Advice”。

切入点（Pointcut）：匹配连接点的断言，在 AOP 中通知和一个切入点表达式关联。切面中的所有通知所关注的连接点，都由切入点表达式来决定。

目标对象（Target Object）：被一个或者多个切面所通知的对象。例如，AServcieImpl 和 BServiceImpl，当然在实际运行时，Spring AOP 采用代理实现，实际 AOP 操作的是 TargetObject 的代理对象。

AOP 代理（AOP Proxy）：在 Spring AOP 中有两种代理方式，JDK 动态代理和 CGLIB 代理。默认情况下，TargetObject 实现了接口时，则采用 JDK 动态代理，例如，AServiceImpl；反之，采用 CGLIB 代理，例如，BServiceImpl。强制使用 CGLIB 代理需要将 <aop:config>的 proxy-target-class 属性设为 true。

通知（Advice）类型：

前置通知（Before advice）：在某连接点（JoinPoint）之前执行的通知，但这个通知不能阻止连接点前的执行。ApplicationContext 中在<aop:aspect>里面使用<aop:before>元素进行声明。例如，TestAspect 中的 doBefore 方法。

后置通知（After advice）：当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。ApplicationContext 中在<aop:aspect>里面使用<aop:after>元素进行声明。例如，ServiceAspect 中的 returnAfter 方法，所以 Teser 中调用 UserService.delete 抛出异常时，returnAfter 方法仍然执行。

返回后通知（After return advice）：在某连接点正常完成后执行的通知，不包括抛出异常的情况。ApplicationContext 中在<aop:aspect>里面使用<after-returning>元素进行声明。

环绕通知（Around advice）：包围一个连接点的通知，类似 Web 中 Servlet 规范中的 Filter 的 doFilter 方法。可以在方法的调用前后完成自定义的行为，也可以选择不执行。ApplicationContext 中在<aop:aspect>里面使用<aop:around>元素进行声明。例如，ServiceAspect 中的 around 方法。

抛出异常后通知（After throwing advice）：在方法抛出异常退出时执行的通知。ApplicationContext 中在<aop:aspect>里面使用<aop:after-throwing>元素进行声明。例如，ServiceAspect 中的 returnThrow 方法。

注：可以将多个通知应用到一个目标对象上，即可以将多个切面织入到同一目标对象。

使用 Spring AOP 可以基于两种方式，一种是比较方便和强大的注解方式，另一种则是中规中矩的 xml 配置方式。

先说注解，使用注解配置 Spring AOP 总体分为两步，第一步是在 xml 文件中声明激活自动扫描组件功能，同时激活自动代理功能（来测试 AOP 的注解功能）：

```
<?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.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-2.0.xsd
      http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

   <context:component-scan base-package="com.gupaoedu"/>
   <context:annotation-config />
</beans>
```

第二步是为 Aspect 切面类添加注解：

```
//声明这是一个组件
@Component
//声明这是一个切面 Bean
@Aspect
public class AnnotaionAspect {

   private final static Logger log = Logger.getLogger(String.valueOf(AnnotaionAspect.class));

   //配置切入点,该方法无方法体,主要为方便同类中其他方法使用此处配置的切入点
   @Pointcut("execution(* com.gupaoedu.aop.service..*(..))")
   public void aspect(){  }

   /*
    * 配置前置通知,使用在方法 aspect()上注册的切入点
    * 同时接受 JoinPoint 切入点对象,可以没有该参数
    */
   @Before("aspect()")
   public void before(JoinPoint joinPoint){
      log.info("before " + joinPoint);
   }

   //配置后置通知,使用在方法 aspect()上注册的切入点
   @After("aspect()")
   public void after(JoinPoint joinPoint){
      log.info("after " + joinPoint);
```

```
    }

    //配置环绕通知,使用在方法 aspect()上注册的切入点
    @Around("aspect()")
    public void around(JoinPoint joinPoint){
        long start = System.currentTimeMillis();
        try {
            ((ProceedingJoinPoint) joinPoint).proceed();
            long end = System.currentTimeMillis();
            log.info("around " + joinPoint + "\tUse time : " + (end - start) + " ms!");
        } catch (Throwable e) {
            long end = System.currentTimeMillis();
            log.info("around " + joinPoint + "\tUse time : " + (end - start) + " ms with exception : " + e.getMessage());
        }
    }

    //配置后置返回通知,使用在方法 aspect()上注册的切入点
    @AfterReturning("aspect()")
    public void afterReturn(JoinPoint joinPoint){
        log.info("afterReturn " + joinPoint);
    }

    //配置抛出异常后通知,使用在方法 aspect()上注册的切入点
    @AfterThrowing(pointcut="aspect()", throwing="ex")
    public void afterThrow(JoinPoint joinPoint, Exception ex){
        log.info("afterThrow " + joinPoint + "\t" + ex.getMessage());
    }

}
```

测试代码

```
@ContextConfiguration(locations = {"classpath*:application-context.xml"})
@RunWith(SpringJUnit4ClassRunner.class)
public class AnnotationTester {
    @Autowired
    MemberService annotationService;
    @Autowired
    ApplicationContext app;


    @Test
// @Ignore
    public void test(){
        System.out.println("=====这是一条华丽的分割线======");
```

```
        AnnotaionAspect aspect = app.getBean(AnnotaionAspect.class);
        System.out.println(aspect);
        annotationService.save(new Member());

        System.out.println("=====这是一条华丽的分割线======");
        try {
            annotationService.delete(1L);
        } catch (Exception e) {
            //e.printStackTrace();
        }
    }

}
```

控制台输出如下：

```
=====这是一条华丽的分割线======
com.gupaoedu.aop.aspect.AnnotaionAspect@6ef714a0
[INFO ] [13:04:46] com.gupaoedu.aop.aspect.AnnotaionAspect - before execution(void
com.gupaoedu.aop.service.MemberService.save(Member))
[INFO ] [13:04:46] com.gupaoedu.aop.aspect.ArgsAspect - beforeArgUser execution(void
com.gupaoedu.aop.service.MemberService.save(Member))
[INFO ] [13:04:46] com.gupaoedu.aop.aspect.AnnotaionAspect - save member Method . . .
[INFO ] [13:04:46] com.gupaoedu.aop.aspect.AnnotaionAspect - around execution(void
com.gupaoedu.aop.service.MemberService.save(Member))   Use time : 38 ms!
[INFO ] [13:04:46] com.gupaoedu.aop.aspect.AnnotaionAspect - after execution(void
com.gupaoedu.aop.service.MemberService.save(Member))
[INFO ] [13:04:46] com.gupaoedu.aop.aspect.AnnotaionAspect - afterReturn execution(void
com.gupaoedu.aop.service.MemberService.save(Member))
=====这是一条华丽的分割线======
[INFO ] [13:04:46] com.gupaoedu.aop.aspect.AnnotaionAspect - before execution(boolean
com.gupaoedu.aop.service.MemberService.delete(long))
[INFO ] [13:04:46] com.gupaoedu.aop.aspect.ArgsAspect - beforeArgId execution(boolean
com.gupaoedu.aop.service.MemberService.delete(long))   ID:1
[INFO ] [13:04:46] com.gupaoedu.aop.aspect.AnnotaionAspect - delete Method . . .
[INFO ] [13:04:46] com.gupaoedu.aop.aspect.AnnotaionAspect - around execution(boolean
com.gupaoedu.aop.service.MemberService.delete(long))   Use time : 3 ms with exception : spring aop ThrowAdvice
演示
[INFO ] [13:04:46] com.gupaoedu.aop.aspect.AnnotaionAspect - after execution(boolean
com.gupaoedu.aop.service.MemberService.delete(long))
[INFO ] [13:04:46] com.gupaoedu.aop.aspect.AnnotaionAspect - afterReturn execution(boolean
com.gupaoedu.aop.service.MemberService.delete(long))
```

可以看到，正如我们预期的那样，虽然我们并没有对 MemberService 类包括其调用方式做任何改变，但是 Spring 仍然拦截到了其中方法的调用，或许这正是 AOP 的魔力所在。

再简单说一下 xml 配置方式，其实也一样简单：

```
<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-3.0.xsd
                  http://www.springframework.org/schema/tx
                  http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
                  http://www.springframework.org/schema/aop
                  http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
   <aop:aspectj-autoproxy proxy-target-class="true"/>

   <bean id="xmlAspect" class="com.gupaoedu.aop.aspect.XmlAspect"></bean>
   <!-- AOP 配置 -->
   <aop:config>
      <!-- 声明一个切面,并注入切面 Bean,相当于@Aspect -->
      <aop:aspect ref="xmlAspect">
         <!-- 配置一个切入点,相当于@Pointcut -->
         <aop:pointcut expression="execution(* com.gupaoedu.aop.service..*(..))" id="simplePointcut"/>
         <!-- 配置通知,相当于@Before、@After、@AfterReturn、@Around、@AfterThrowing -->
         <aop:before pointcut-ref="simplePointcut" Method="before"/>
         <aop:after pointcut-ref="simplePointcut" Method="after"/>
         <aop:after-returning pointcut-ref="simplePointcut" Method="afterReturn"/>
         <aop:after-throwing pointcut-ref="simplePointcut" Method="afterThrow" throwing="ex"/>
      </aop:aspect>
   </aop:config>

</beans>
```

个人觉得不如注解灵活和强大，你可以不同意这个观点，但是不知道如下的代码会不会让你的想法有所改善：

```
//配置切入点,该方法无方法体,主要为方便同类中其他方法使用此处配置的切入点
@Pointcut("execution(* com.gupaoedu.aop.service..*(..))")
public void aspect(){  }

//配置前置通知,拦截返回值为 cn.ysh.studio.spring.mvc.bean.User 的方法
@Before("execution(com.gupaoedu.model.Member com.gupaoedu.aop.service..*(..))")
public void beforeReturnUser(JoinPoint joinPoint){
   log.info("beforeReturnUser " + joinPoint);
}

//配置前置通知,拦截参数为 cn.ysh.studio.spring.mvc.bean.User 的方法
@Before("execution(* com.gupaoedu.aop.service..*(com.gupaoedu.model.Member))")
public void beforeArgUser(JoinPoint joinPoint){
   log.info("beforeArgUser " + joinPoint);
```

```
}

//配置前置通知,拦截含有 long 类型参数的方法,并将参数值注入到当前方法的形参 id 中
@Before("aspect()&&args(id)")
public void beforeArgId(JoinPoint joinPoint, long id){
   log.info("beforeArgId " + joinPoint + "\tID:" + id);
}
```

以下是 MemberService 的代码：

```
@Service
public class MemberService {

   private final static Logger log = Logger.getLogger(AnnotaionAspect.class);

   public Member get(long id){
      log.info("getMemberById Method . . .");
      return new Member();
   }

   public Member get(){
      log.info("getMember Method . . .");
      return new Member();
   }

   public void save(Member member){
      log.info("save member Method . . .");
   }

   public boolean delete(long id) throws Exception{
      log.info("delete Method . . .");
      throw new Exception("spring aop ThrowAdvice 演示");
   }

}
```

应该说学习 Spring AOP 有两个难点，第一点在于理解 AOP 的理念和相关概念，第二点在于灵活掌握和使用切入点表达式。概念的理解通常不在一朝一夕，慢慢浸泡的时间长了，自然就明白了，下面我们简单地介绍一下切入点表达式的配置规则吧。

通常情况下，表达式中使用”execution“就可以满足大部分的要求。表达式格式如下：

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern)
throws-pattern?
```

modifiers-pattern：方法的操作权限

ret-type-pattern：返回值

declaring-type-pattern：方法所在的包

name-pattern：方法名

parm-pattern：参数名

throws-pattern：异常

其中，除 ret-type-pattern 和 name-pattern 之外，其他都是可选的。上例中，execution(* com.spring.service.*.*(..))表示 com.spring.service 包下，返回值为任意类型；方法名任意；参数不作限制的所有方法。

最后说一下通知参数

可以通过 args 来绑定参数，这样就可以在通知（Advice）中访问具体参数了。例如，<aop:aspect>配置如下：

```
<aop:config>
   <aop:aspect ref="xmlAspect">
      <aop:pointcut id="simplePointcut"
                expression="execution(* com.gupaoedu.aop.service..*(..)) and args(msg,..)" />
      <aop:after pointcut-ref="simplePointcut" Method="after"/>
   </aop:aspect>
</aop:config>
```

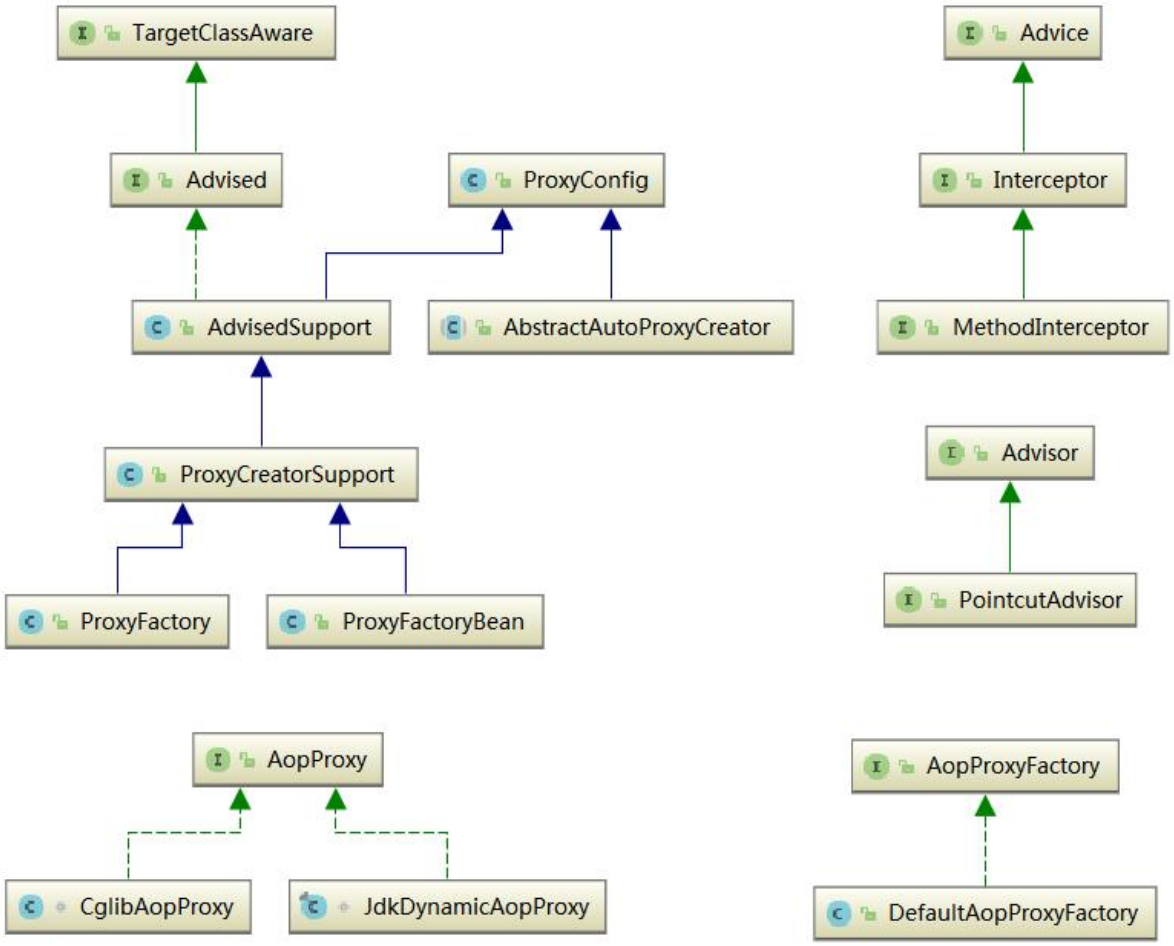
上面的代码 args(msg,..)是指将切入点方法上的第一个 String 类型参数添加到参数名为 msg 的通知的入参上，这样就可以直接使用该参数啦。

访问当前的连接点

在上面的 Aspect 切面 Bean 中已经看到了，每个通知方法第一个参数都是 JoinPoint。其实，在 Spring 中，任何通知（Advice）方法都可以将第一个参数定义为 org.aspectj.lang.JoinPoint 类型用以接受当前连接点对象。JoinPoint 接口提供了一系列有用的方法， 比如 getArgs()（返回方法参数）、getThis()（返回代理对象）、getTarget()（返回目标）、getSignature()（返回正在被通知的方法相关信息）和 toString()（打印出正在被通知的方法的有用信息）。

4.7.2、SpringAOP 设计原理及源码分析

开始之前先上图，看看 Spring 中主要的 AOP 组件



Spring 提供了两种方式来生成代理对象：JDKProxy 和 Cglib，具体使用哪种方式生成由 AopProxyFactory 根据 AdvisedSupport 对象的配置来决定。默认的策略是如果目标类是接口，则使用 JDK 动态代理技术，否则使用 Cglib 来生成代理。下面我们来研究一下 Spring 如何使用 JDK 来生成代理对象，具体的生成代码放在 JdkDynamicAopProxy 这个类中，直接上相关代码：

```
/**
 * 获取代理类要实现的接口,除了 Advised 对象中配置的,还会加上 SpringProxy, Advised(opaque=false)
 * 检查上面得到的接口中有没有定义 equals 或者 hashcode 的接口
 * 调用 Proxy.newProxyInstance 创建代理对象
 */
@Override
public Object getProxy(@Nullable ClassLoader classLoader) {
   if (logger.isDebugEnabled()) {
      logger.debug("Creating JDK dynamic proxy: target source is " + this.advised.getTargetSource());
   }
   Class<?>[] proxiedInterfaces = AopProxyUtils.completeProxiedInterfaces(this.advised, true);
   findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
   return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);
```

```
}
```

那这个其实很明了，注释上我也已经写清楚了，不再赘述。

下面的问题是，代理对象生成了，那切面是如何织入的？

我们知道 InvocationHandler 是 JDK 动态代理的核心，生成的代理对象的方法调用都会委托到 InvocationHandler.invoke()方法。而通过 JdkDynamicAopProxy 的签名我们可以看到这个类其实也实现了 InvocationHandler，下面我们就通过分析这个类中实现的 invoke()方法来具体看下 Spring AOP 是如何织入切面的。

```
public Object invoke(Object proxy, Method Method, Object[] args) throws Throwable {
   MethodInvocation invocation;
   Object oldProxy = null;
   boolean setProxyContext = false;

   TargetSource targetSource = this.advised.targetSource;
   Object target = null;

   try {
      //eqauls()方法，具目标对象未实现此方法
      if (!this.equalsDefined && AopUtils.isEqualsMethod(Method)) {
         return equals(args[0]);
      }
      //hashCode()方法，具目标对象未实现此方法
      else if (!this.hashCodeDefined && AopUtils.isHashCodeMethod(Method)) {
         return hashCode();
      }
      else if (Method.getDeclaringClass() == DecoratingProxy.class) {
         return AopProxyUtils.ultimateTargetClass(this.advised);
      }
      //Advised 接口或者其父接口中定义的方法,直接反射调用,不应用通知
      else if (!this.advised.opaque && Method.getDeclaringClass().isInterface() &&
            Method.getDeclaringClass().isAssignableFrom(Advised.class)) {
         return AopUtils.invokeJoinpointUsingReflection(this.advised, Method, args);
      }

      Object retVal;

      if (this.advised.exposeProxy) {
         oldProxy = AopContext.setCurrentProxy(proxy);
         setProxyContext = true;
      }

      //获得目标对象的类
      target = targetSource.getTarget();
```

```
        Class<?> targetClass = (target != null ? target.getClass() : null);

        //获取可以应用到此方法上的 Interceptor 列表
        List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(Method, targetClass);

        //如果没有可以应用到此方法的通知(Interceptor)，此直接反射调用 Method.invoke(target, args)
        if (chain.isEmpty()) {
           Object[] argsToUse = AopProxyUtils.adaptArgumentsIfNecessary(Method, args);
           retVal = AopUtils.invokeJoinpointUsingReflection(target, Method, argsToUse);
        }
        else {
           //创建 MethodInvocation
           invocation = new ReflectiveMethodInvocation(proxy, target, Method, args, targetClass, chain);
           retVal = invocation.proceed();
        }

        Class<?> returnType = Method.getReturnType();
        if (retVal != null && retVal == target &&
              returnType != Object.class && returnType.isInstance(proxy) &&
              !RawTargetAccess.class.isAssignableFrom(Method.getDeclaringClass())) {
           retVal = proxy;
        }
        else if (retVal == null && returnType != Void.TYPE && returnType.isPrimitive()) {
           throw new AopInvocationException(
                 "Null return value from advice does not match primitive return type for: " + Method);
        }
        return retVal;
     }
     finally {
        if (target != null && !targetSource.isStatic()) {
           targetSource.releaseTarget(target);
        }
        if (setProxyContext) {
           AopContext.setCurrentProxy(oldProxy);
        }
     }
  }
```

主流程可以简述为：获取可以应用到此方法上的通知链（Interceptor Chain），如果有，则应用通知，并执行 joinpoint；如果没有，则直接反射执行 joinpoint。而这里的关键是通知链是如何获取的以及它又是如何执行的，下面逐一分析下。

首先，从上面的代码可以看到，通知链是通过Advised.getInterceptorsAndDynamicInterceptionAdvice()这个方法来获取的，我们来看下这个方法的实现：

```
public List<Object> getInterceptorsAndDynamicInterceptionAdvice(Method Method, @Nullable Class<?> targetClass)
{
   MethodCacheKey cacheKey = new MethodCacheKey(Method);
   List<Object> cached = this.MethodCache.get(cacheKey);
   if (cached == null) {
      cached = this.advisorChainFactory.getInterceptorsAndDynamicInterceptionAdvice(
            this, Method, targetClass);
      this.MethodCache.put(cacheKey, cached);
   }
   return cached;
}
```

可以看到实际的获取工作其实是由AdvisorChainFactory.getInterceptorsAndDynamicInterceptionAdvice()这个方法来完成的，获取到的结果会被缓存。下面来分析下这个方法的实现：

```
/**
 * 从提供的配置实例 config 中获取 advisor 列表,遍历处理这些 advisor.如果是 IntroductionAdvisor,
 * 则判断此 Advisor 能否应用到目标类 targetClass 上.如果是 PointcutAdvisor,则判断
 * 此 Advisor 能否应用到目标方法 Method 上.将满足条件的 Advisor 通过 AdvisorAdaptor 转化成 Interceptor 列表返回.
 */
@Override
public List<Object> getInterceptorsAndDynamicInterceptionAdvice(
      Advised config, Method Method, @Nullable Class<?> targetClass) {

   List<Object> interceptorList = new ArrayList<>(config.getAdvisors().length);
   Class<?> actualClass = (targetClass != null ? targetClass : Method.getDeclaringClass());
   //查看是否包含 IntroductionAdvisor
   boolean hasIntroductions = hasMatchingIntroductions(config, actualClass);
   //这里实际上注册一系列 AdvisorAdapter,用于将 Advisor 转化成 MethodInterceptor
   AdvisorAdapterRegistry registry = GlobalAdvisorAdapterRegistry.getInstance();

   for (Advisor advisor : config.getAdvisors()) {
      if (advisor instanceof PointcutAdvisor) {
         PointcutAdvisor pointcutAdvisor = (PointcutAdvisor) advisor;
         if (config.isPreFiltered() || pointcutAdvisor.getPointcut().getClassFilter().matches(actualClass)) {
            //这个地方这两个方法的位置可以互换下
            //将 Advisor 转化成 Interceptor
            MethodInterceptor[] interceptors = registry.getInterceptors(advisor);
            //检查当前 advisor 的 pointcut 是否可以匹配当前方法
            MethodMatcher mm = pointcutAdvisor.getPointcut().getMethodMatcher();
            if (MethodMatchers.matches(mm, Method, actualClass, hasIntroductions)) {
               if (mm.isRuntime()) {
                  for (MethodInterceptor interceptor : interceptors) {
                     interceptorList.add(new InterceptorAndDynamicMethodMatcher(interceptor, mm));
```

```
                }
            }
            else {
                interceptorList.addAll(Arrays.asList(interceptors));
            }
        }
    }
    else if (advisor instanceof IntroductionAdvisor) {
        IntroductionAdvisor ia = (IntroductionAdvisor) advisor;
        if (config.isPreFiltered() || ia.getClassFilter().matches(actualClass)) {
            Interceptor[] interceptors = registry.getInterceptors(advisor);
            interceptorList.addAll(Arrays.asList(interceptors));
        }
    }
    else {
        Interceptor[] interceptors = registry.getInterceptors(advisor);
        interceptorList.addAll(Arrays.asList(interceptors));
    }
}

return interceptorList;
}
```

这个方法执行完成后，Advised 中配置能够应用到连接点或者目标类的 Advisor 全部被转化成了 MethodInterceptor.

接下来我们再看下得到的拦截器链是怎么起作用的。

```
if (chain.isEmpty()) {
    Object[] argsToUse = AopProxyUtils.adaptArgumentsIfNecessary(Method, args);
    retVal = AopUtils.invokeJoinpointUsingReflection(target, Method, argsToUse);
}
else {
    //创建 MethodInvocation
    invocation = new ReflectiveMethodInvocation(proxy, target, Method, args, targetClass, chain);
    retVal = invocation.proceed();
}
```

从这段代码可以看出，如果得到的拦截器链为空，则直接反射调用目标方法，否则创建 MethodInvocation，调用其 proceed 方法，触发拦截器链的执行，来看下具体代码

```
public Object proceed() throws Throwable {
    //如果 Interceptor 执行完了，则执行 joinPoint
    if (this.currentInterceptorIndex == this.interceptorsAndDynamicMethodMatchers.size() - 1) {
        return invokeJoinpoint();
    }
    Object interceptorOrInterceptionAdvice =
```

```
        this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIndex);
    //如果要动态匹配 joinPoint
        InterceptorAndDynamicMethodMatcher dm =
              (InterceptorAndDynamicMethodMatcher) interceptorOrInterceptionAdvice;
        //动态匹配：运行时参数是否满足匹配条件
        if (dm.MethodMatcher.matches(this.Method, this.targetClass, this.arguments)) {
            return dm.interceptor.invoke(this);
        }
        else {
            //动态匹配失败时,略过当前 Intercetpor,调用下一个 Interceptor
            return proceed();
        }
    }
    else {
        //执行当前 Intercetpor
        return ((MethodInterceptor) interceptorOrInterceptionAdvice).invoke(this);
    }
}
```

5.9、基于 Spring JDBC 开发 ORM 框架

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 JDBC 2.0 以上的数据库驱动的支持，其中一些高级的功能可能需要 JDBC 3.0 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 JDK 1.4 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 JDK 1.4 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 Java 5 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 Java 5 以上的环境中。

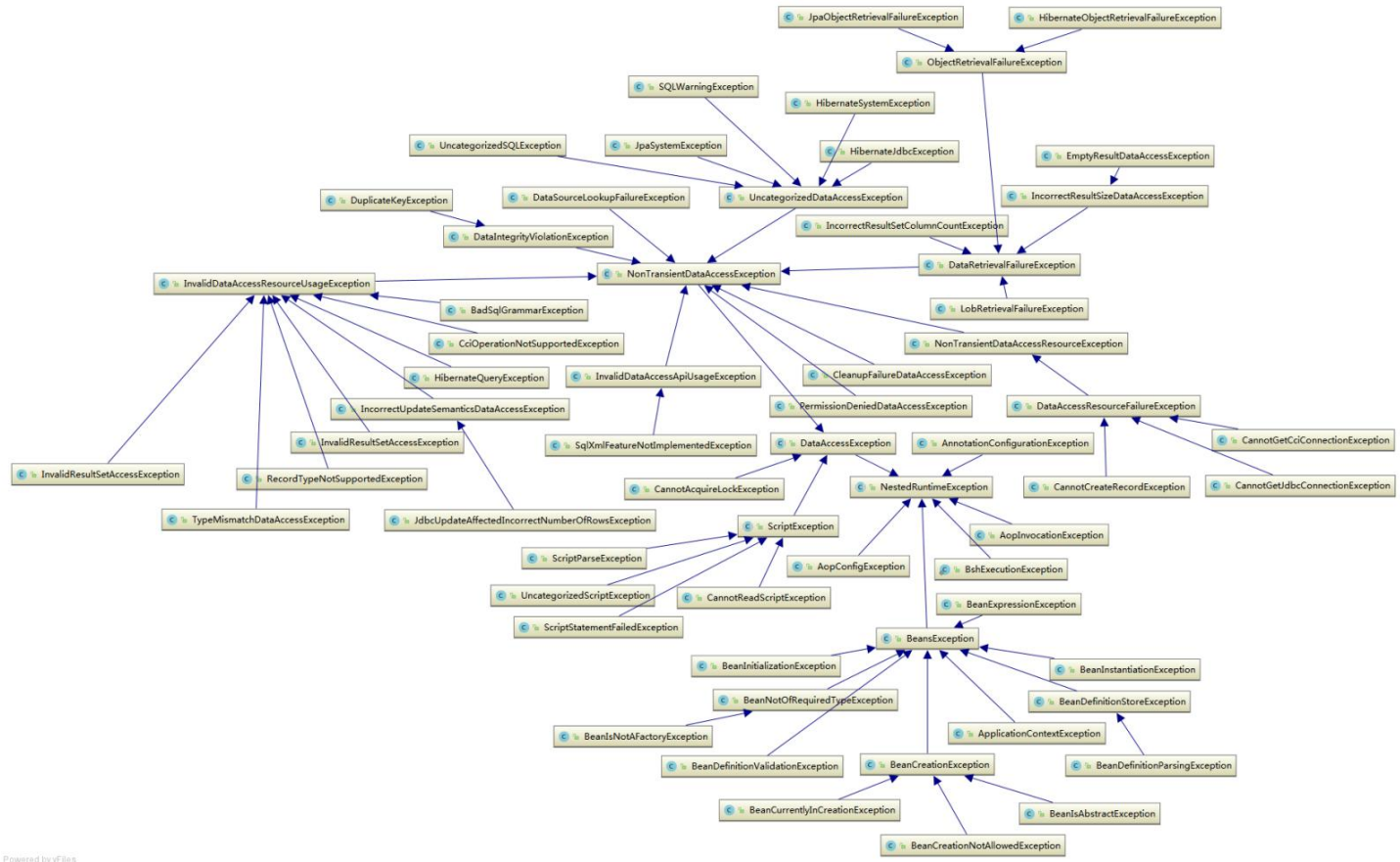
SimpleJdbcInsert 和 SimpleJdbcCall - 这两个类可以充分利用数据库元数据的特性来简化配置。通过使用这两个类进行编程，你可以仅仅提供数据库表名或者存储过程的名称以及一个 Map 作为参数。

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 JDK 5 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 JDK 1.4 以上。

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）。

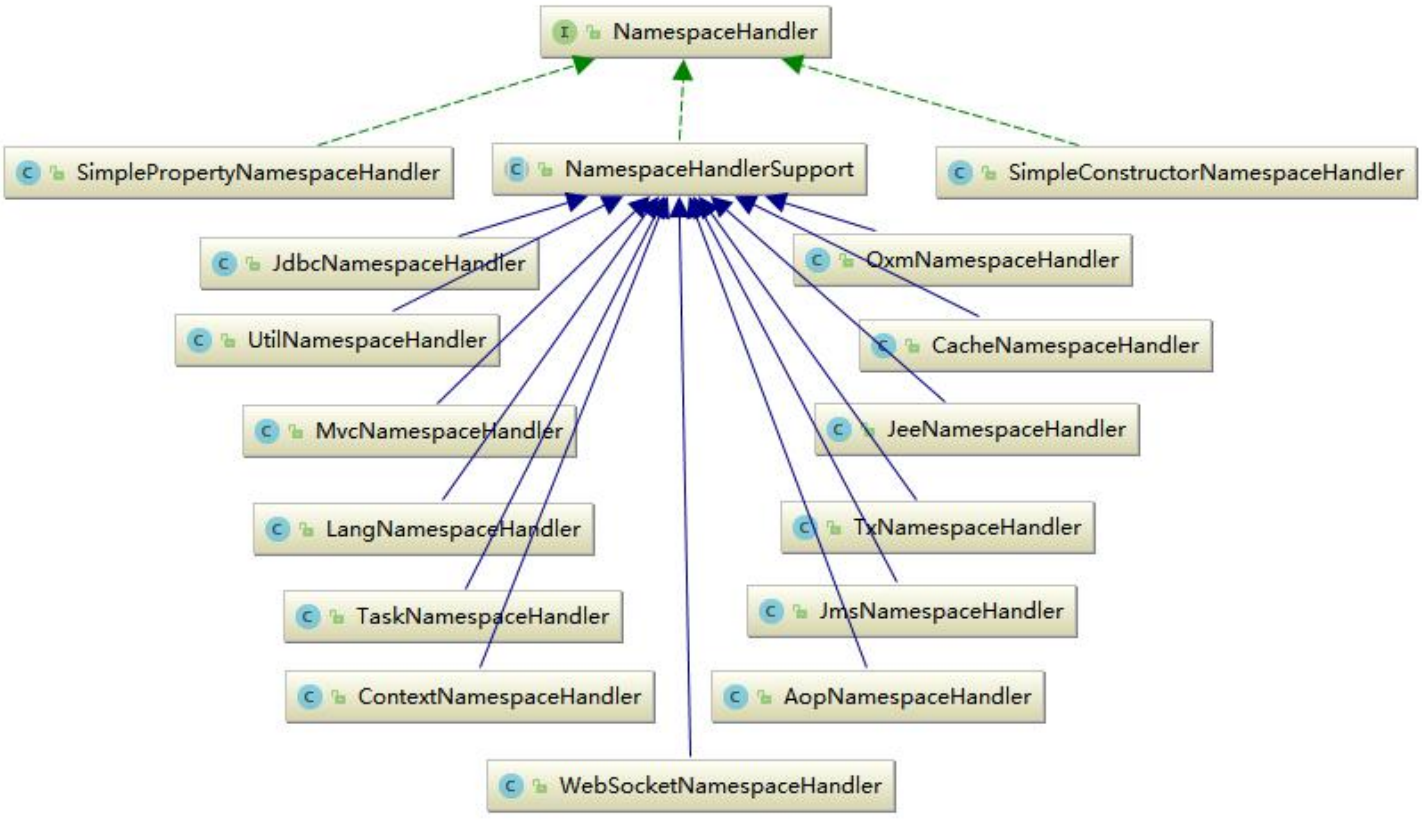
SQLErrorCodeSQLExceptionTranslator 是 SQLExceptionTranslator 的默认实现。 该实现使用指定数据库厂商的 error code，比采用 SQLState 更精确。转换过程基于一个 JavaBean（类型为 SQLErrorCodes）中的 error code。 这个 JavaBean 由 SQLErrorCodesFactory 工厂类创建，其中

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 fallback 翻译器。
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 {
```

```
    @Override
    public void init() {
        registerBeanDefinitionParser("embedded-database", new EmbeddedDatabaseBeanDefinitionParser());
        registerBeanDefinitionParser("initialize-database", new InitializeDatabaseBeanDefinitionParser());
    }
}
```

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。

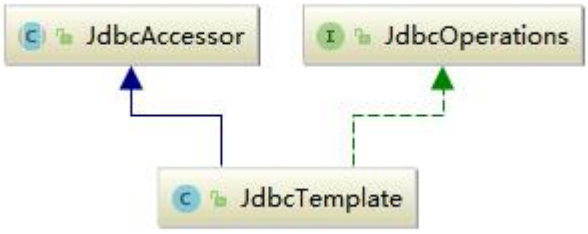
Text： 该 Text 接口继承自 CharacterData，并且表示 Element 或 Attr 的文本内容（在 XML 中称为 字符数据）。

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。

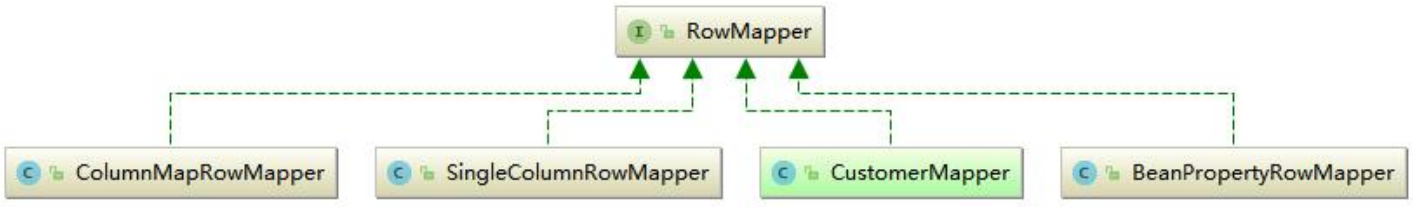
UserDataHandler： 当使用 Node.setUserData() 将一个对象与节点上的键相关联时，当克隆、导入或重命名该对象关联的节点时应用程序可以提供调用的处理程序。

3. core 模块

3.1 JdbcTeamplate 对象，其结构如下：

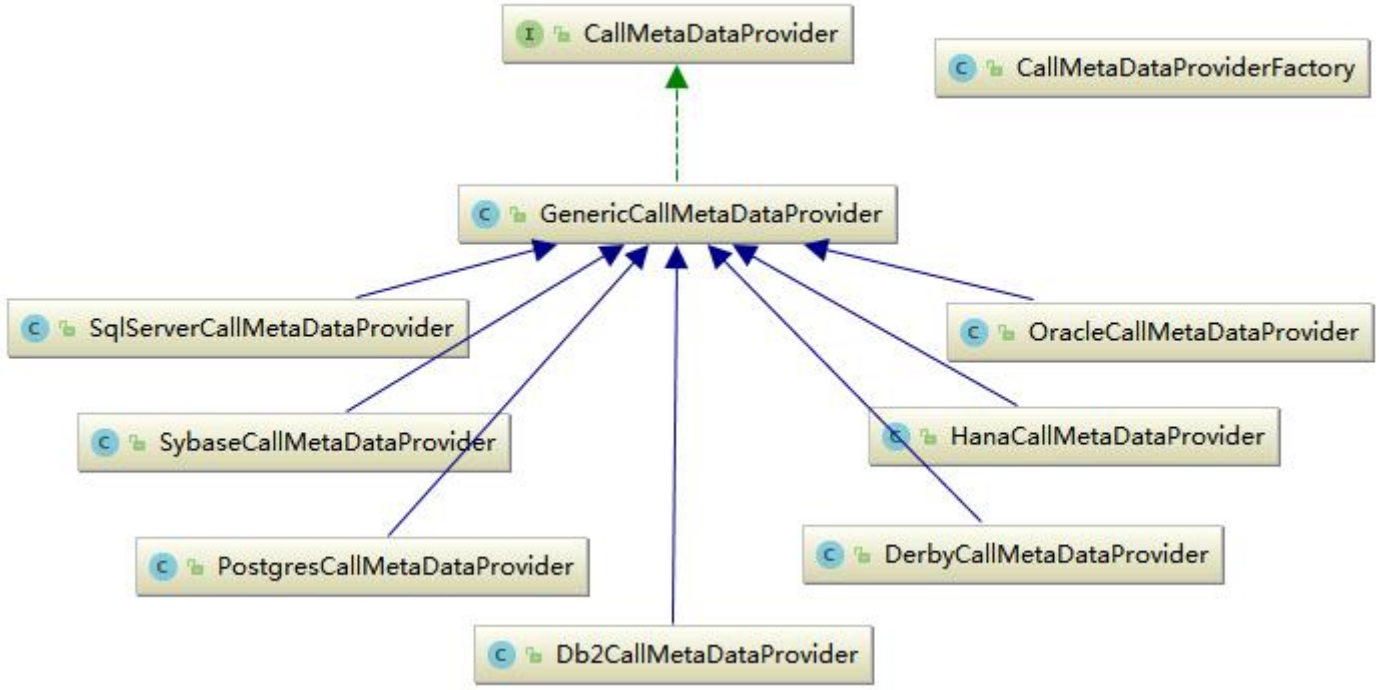


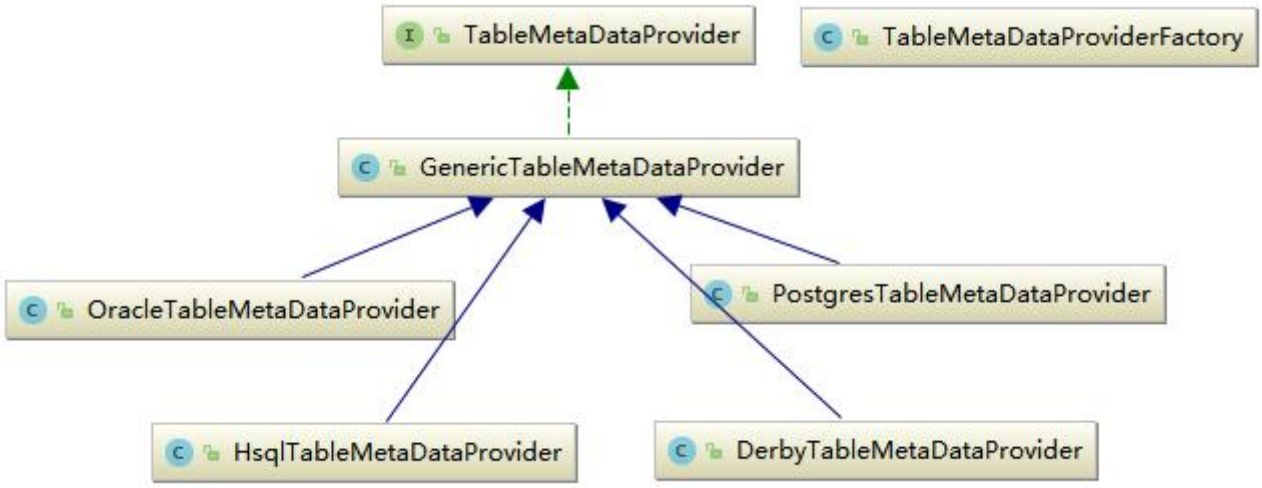
3.2 RowMapper



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 static final List<String> supportedDatabaseProductsForProcedures = Arrays.asList(
     "Apache Derby",
     "DB2",
     "MySQL",
     "Microsoft SQL Server",
     "Oracle",
     "PostgreSQL",
     "Sybase"
   );
```

```
/** List of supported database products for function calls */
public static final List<String> supportedDatabaseProductsForFunctions = Arrays.asList(
      "MySQL",
      "Microsoft SQL Server",
      "Oracle",
      "PostgreSQL"
   );

static public CallMetaDataProvider createMetaDataProvider(DataSource dataSource, final CallMetaDataContext
context) {
   try {
      CallMetaDataProvider result = (CallMetaDataProvider) JdbcUtils.extractDatabaseMetaData(dataSource,
databaseMetaData -> {
         String databaseProductName = JdbcUtils.commonDatabaseName(databaseMetaData.getDatabaseProductName());
         boolean accessProcedureColumnMetaData = context.isAccessCallParameterMetaData();
         if (context.isFunction()) {
            if (!supportedDatabaseProductsForFunctions.contains(databaseProductName)) {
               if (logger.isWarnEnabled()) {
                  logger.warn(databaseProductName + " is not one of the databases fully supported for function calls
" +
                        "-- supported are: " + supportedDatabaseProductsForFunctions);
               }
               if (accessProcedureColumnMetaData) {
                  logger.warn("Metadata processing disabled - you must specify all parameters explicitly");
                  accessProcedureColumnMetaData = false;
               }
            }
         }
         else {
            if (!supportedDatabaseProductsForProcedures.contains(databaseProductName)) {
               if (logger.isWarnEnabled()) {
                  logger.warn(databaseProductName + " is not one of the databases fully supported for procedure
calls " +
                        "-- supported are: " + supportedDatabaseProductsForProcedures);
               }
               if (accessProcedureColumnMetaData) {
                  logger.warn("Metadata processing disabled - you must specify all parameters explicitly");
                  accessProcedureColumnMetaData = false;
               }
            }
         }
         CallMetaDataProvider provider;
         if ("Oracle".equals(databaseProductName)) {
            provider = new OracleCallMetaDataProvider(databaseMetaData);
```

```
        }
        else if ("DB2".equals(databaseProductName)) {
           provider = new Db2CallMetaDataProvider((databaseMetaData));
        }
        else if ("Apache Derby".equals(databaseProductName)) {
           provider = new DerbyCallMetaDataProvider((databaseMetaData));
        }
        else if ("PostgreSQL".equals(databaseProductName)) {
           provider = new PostgresCallMetaDataProvider((databaseMetaData));
        }
        else if ("Sybase".equals(databaseProductName)) {
           provider = new SybaseCallMetaDataProvider((databaseMetaData));
        }
        else if ("Microsoft SQL Server".equals(databaseProductName)) {
           provider = new SqlServerCallMetaDataProvider((databaseMetaData));
        }
        else if ("HDB".equals(databaseProductName)) {
           provider = new HanaCallMetaDataProvider((databaseMetaData));
        }
        else {
           provider = new GenericCallMetaDataProvider(databaseMetaData);
        }
        if (logger.isDebugEnabled()) {
           logger.debug("Using " + provider.getClass().getName());
        }
        provider.initializeWithMetaData(databaseMetaData);
        if (accessProcedureColumnMetaData) {
           provider.initializeWithProcedureColumnMetaData(databaseMetaData,
                 context.getCatalogName(), context.getSchemaName(), context.getProcedureName());
        }
        return provider;
     });
     return result;
  }
  catch (MetaDataAccessException ex) {
     throw new DataAccessResourceFailureException("Error retrieving database metadata", ex);
  }
}
```

TableMetaDataProviderFactory 创建 TableMetaDataProvider 工厂类，其创建过程如下：

```
static public CallMetaDataProvider createMetaDataProvider(DataSource dataSource, final CallMetaDataContext
context) {
  try {
     CallMetaDataProvider result = (CallMetaDataProvider) JdbcUtils.extractDatabaseMetaData(dataSource,
databaseMetaData -> {
```

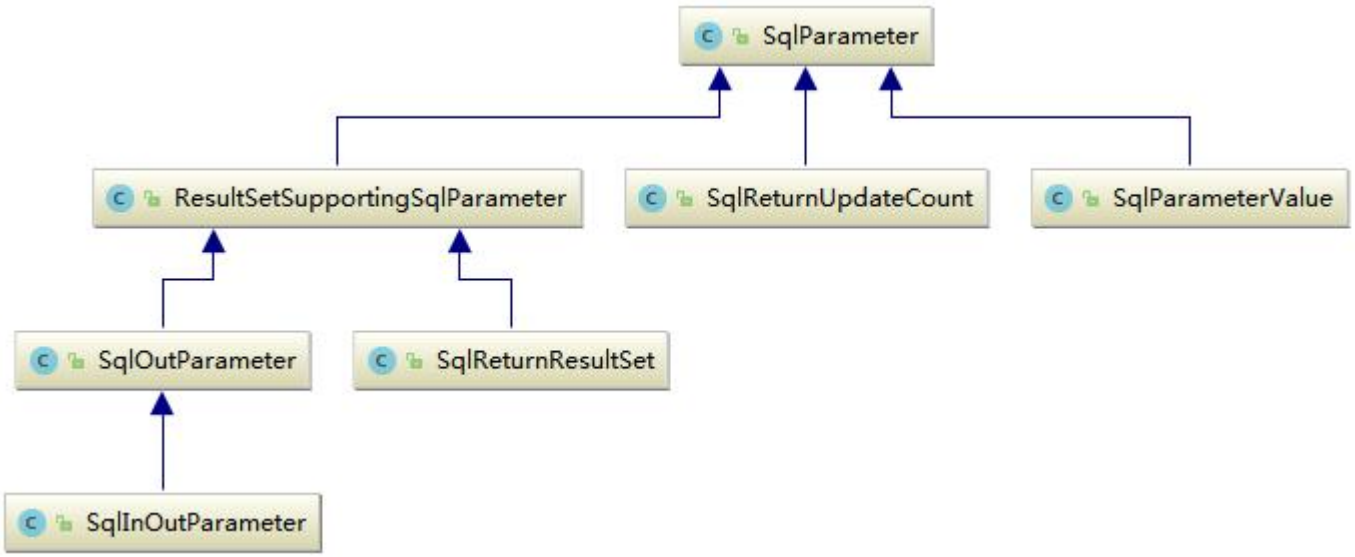
```
        String databaseProductName = JdbcUtils.commonDatabaseName(databaseMetaData.getDatabaseProductName());
        boolean accessProcedureColumnMetaData = context.isAccessCallParameterMetaData();
        if (context.isFunction()) {
           if (!supportedDatabaseProductsForFunctions.contains(databaseProductName)) {
              if (logger.isWarnEnabled()) {
                 logger.warn(databaseProductName + " is not one of the databases fully supported for function calls
" +
                       "-- supported are: " + supportedDatabaseProductsForFunctions);
              }
              if (accessProcedureColumnMetaData) {
                 logger.warn("Metadata processing disabled - you must specify all parameters explicitly");
                 accessProcedureColumnMetaData = false;
              }
           }
        }
        else {
           if (!supportedDatabaseProductsForProcedures.contains(databaseProductName)) {
              if (logger.isWarnEnabled()) {
                 logger.warn(databaseProductName + " is not one of the databases fully supported for procedure
calls " +
                       "-- supported are: " + supportedDatabaseProductsForProcedures);
              }
              if (accessProcedureColumnMetaData) {
                 logger.warn("Metadata processing disabled - you must specify all parameters explicitly");
                 accessProcedureColumnMetaData = false;
              }
           }
        }
        CallMetaDataProvider provider;
        if ("Oracle".equals(databaseProductName)) {
           provider = new OracleCallMetaDataProvider(databaseMetaData);
        }
        else if ("DB2".equals(databaseProductName)) {
           provider = new Db2CallMetaDataProvider((databaseMetaData));
        }
        else if ("Apache Derby".equals(databaseProductName)) {
           provider = new DerbyCallMetaDataProvider((databaseMetaData));
        }
        else if ("PostgreSQL".equals(databaseProductName)) {
           provider = new PostgresCallMetaDataProvider((databaseMetaData));
        }
        else if ("Sybase".equals(databaseProductName)) {
           provider = new SybaseCallMetaDataProvider((databaseMetaData));
        }
```

```
        else if ("Microsoft SQL Server".equals(databaseProductName)) {
           provider = new SqlServerCallMetaDataProvider((databaseMetaData));
        }
        else if ("HDB".equals(databaseProductName)) {
           provider = new HanaCallMetaDataProvider((databaseMetaData));
        }
        else {
           provider = new GenericCallMetaDataProvider(databaseMetaData);
        }
        if (logger.isDebugEnabled()) {
           logger.debug("Using " + provider.getClass().getName());
        }
        provider.initializeWithMetaData(databaseMetaData);
        if (accessProcedureColumnMetaData) {
           provider.initializeWithProcedureColumnMetaData(databaseMetaData,
                 context.getCatalogName(), context.getSchemaName(), context.getProcedureName());
        }
        return provider;
     });
     return result;
  }
  catch (MetaDataAccessException ex) {
     throw new DataAccessResourceFailureException("Error retrieving database metadata", ex);
  }
}
```

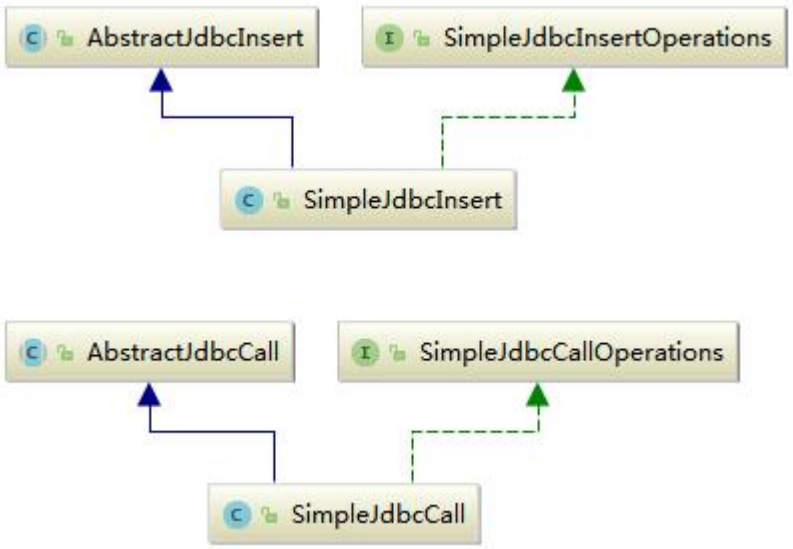
3.4 使用 SqlParameterSource 提供参数值

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。



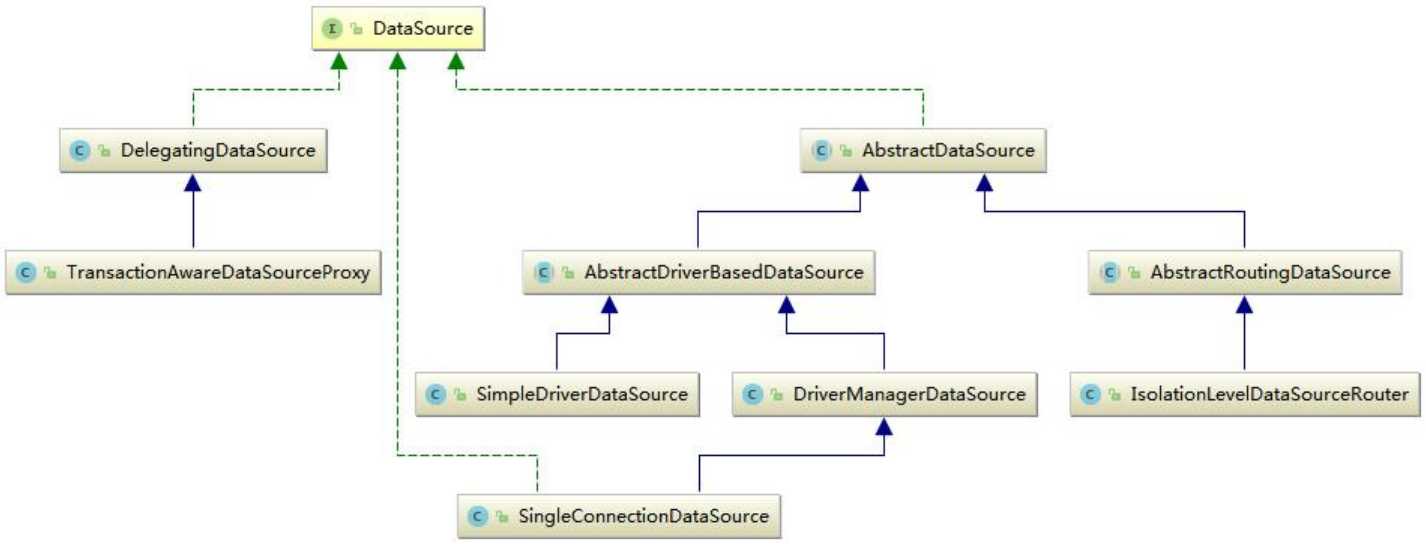
3.5 simple 实现



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。流行的第三方实现由 apache Jakarta Commons dbcp 和 c3p0.



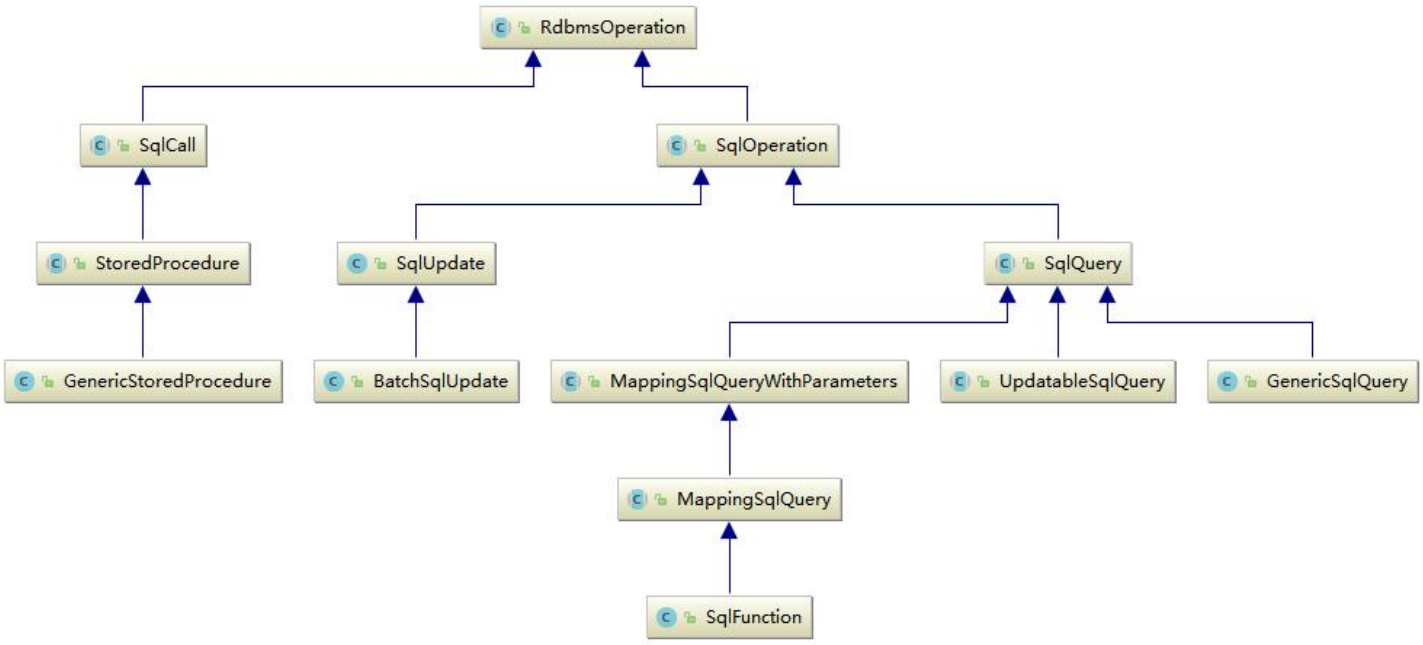
TransactionAwareDataSourceProxy 作为目标 DataSource 的一个代理， 在对目标 DataSource 包装的同时，还增加了 Spring 的事务管理能力， 在这一点上，这个类的功能非常像 J2EE 服务器所提供的事务化的 JNDI DataSource。

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。

5. object 模块



6. 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。 CallableStatementCreateor 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 容器中总是配制成一个 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。
7. 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。

5.10、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。

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，所以分区容忍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。

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。