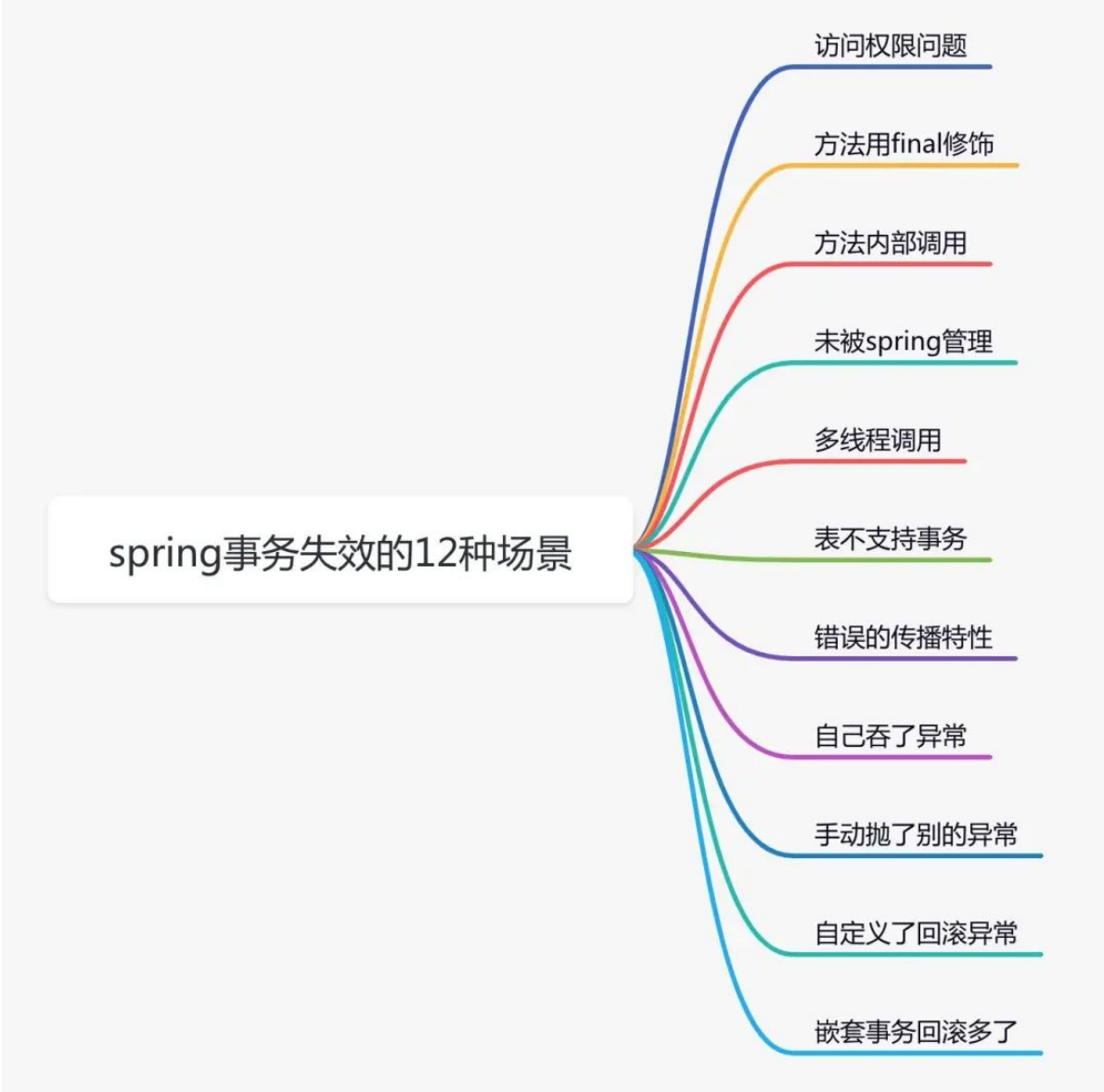
@来源 [啪！啪！@Transactional 注解的12种失效场景，这坑我踩个遍]



一 事务不生效

1.访问权限问题

众所周知，java的访问权限主要有四种：private、default、protected、public，它们的权限从左到右，依次变大。

但如果我们在开发过程中，把有某些事务方法，定义了错误的访问权限，就会导致事务功能出问题，例如：

```java
@Service
public class UserService {

    @Transactional
    private void add(UserModel userModel) {
         saveData(userModel);
         updateData(userModel);
    }
}
```

我们可以看到add方法的访问权限被定义成了private，这样会导致事务失效，Spring要求被代理方法必须是 **public** 的。

说白了，在 **AbstractFallbackTransactionAttributeSource** 类的 **computeTransactionAttribute** 方法中有个判断，如果目标方法不是public，则**TransactionAttribute返回null**，即不支持事务。

```java
    protected TransactionAttribute computeTransactionAttribute(Method method, @Nullable Class<?> targetClass) {
        // Don't allow no-public methods as required.
        if (allowPublicMethodsOnly() && !Modifier.isPublic(method.getModifiers())) {
            return null;
        }

        // The method may be on an interface, but we need attributes from the target class.
        // If the target class is null, the method will be unchanged.
        Method specificMethod = AopUtils.getMostSpecificMethod(method, targetClass);

        // First try is the method in the target class.
        TransactionAttribute txAttr = findTransactionAttribute(specificMethod);
        if (txAttr != null) {
            return txAttr;
        }

        // Second try is the transaction attribute on the target class.
        txAttr = findTransactionAttribute(specificMethod.getDeclaringClass());
        if (txAttr != null && ClassUtils.isUserLevelMethod(method)) {
            return txAttr;
        }

        if (specificMethod != method) {
            // Fallback is to look at the original method.
            txAttr = findTransactionAttribute(method);
            if (txAttr != null) {
                return txAttr;
            }
            // Last fallback is the class of the original method.
            txAttr = findTransactionAttribute(method.getDeclaringClass());
            if (txAttr != null && ClassUtils.isUserLevelMethod(method)) {
                return txAttr;
            }
        }
        return null;
```

```
    }
```

也就是说，如果我们自定义的事务方法（即目标方法），它的访问权限不是public，而是private、default或protected的话，spring则不会提供事务功能。

2.方法用final修饰

有时候，某个方法不想被子类重新，这时可以将该方法定义成final的。普通方法这样定义是没问题的，但如果将事务方法定义成final，例如：

```
@Service
public class UserService {
    @Transactional
    public final void add(UserModel userModel){
        saveData(userModel);
        updateData(userModel);
    }
}
```

我们可以看到add方法被定义成了 **final** 的，这样会导致事务失效。

为什么?

如果你看过spring事务的源码，可能会知道**spring事务底层使用了aop**，也就是通过jdk动态代理或者cglib，帮我们生成了代理类，在代理类中实现的事务功能。

但如果某个方法用final修饰了，那么在它的代理类中，就无法重写该方法，而添加事务功能。

> 注意：**如果某个方法是static的，同样无法通过动态代理，变成事务方法。**

3.方法内部调用

有时候我们需要在某个Service类的某个方法中，调用另外一个事务方法，比如：

```
@Service
public class UserService {

    @Autowired
    private UserMapper userMapper;

    @Transactional
    public void add(UserModel userModel) {
        userMapper.insertUser(userModel);
        updateStatus(userModel);
    }

    @Transactional
    public void updateStatus(UserModel userModel) {
        doSameThing();
    }
}
```

我们看到在事务方法add中，直接调用事务方法updateStatus。从前面介绍的内容可以知道，updateStatus方法拥有事务的能力是因为Spring aop生成代理了对象，但是这种方法**直接调用了this对象的方法**，所以updateStatus方法不会生成事务。

由此可见，**在同一个类中的方法直接内部调用，会导致事务失效。**

那么问题来了，如果有些场景，确实想在同一个类的某个方法中，调用它自己的另外一个方法，该怎么办呢？

3.1 新加一个Service方法

这个方法非常简单，只需要新加一个Service方法，把@Transactional注解加到新Service方法上，把需要事务执行的代码移到新方法中。具体代码如下：

```
@Servcie
public class ServiceA {
  @Autowired
  prvate ServiceB serviceB;

  public void save(User user) {
        queryData1();
        queryData2();
        serviceB.doSave(user);
  }
}

@Servcie
public class ServiceB {

   @Transactional(rollbackFor=Exception.class)
   public void doSave(User user) {
      addData1();
      updateData2();
   }
}
```

3.2 在该Service类中注入自己

如果不想再新加一个Service类，在该Service类中注入自己也是一种选择。具体代码如下：

```
@Servcie
public class ServiceA {
   @Autowired
   prvate ServiceA serviceA;

   public void save(User user) {
         queryData1();
         queryData2();
         serviceA.doSave(user);
   }

   @Transactional(rollbackFor=Exception.class)
   public void doSave(User user) {
       addData1();
       updateData2();
    }
}
```

可能有些人可能会有这样的疑问：这种做法会不会出现循环依赖问题？

答案：不会。

其实spring ioc内部的三级缓存保证了它，不会出现循环依赖问题。但有些坑，如果你想进一步了解循环依赖问题，可以看看我之前文章《spring：我是如何解决循环依赖的？》。

3.3 通过AopContent类

在该Service类中使用AopContext.currentProxy()获取代理对象

上面的方法2确实可以解决问题，但是代码看起来并不直观，还可以通过在该Service类中使用AOPProxy获取代理对象，实现相同的功能。具体代码如下：

```
@Servcie
public class ServiceA {
   public void save(User user) {
         queryData1();
         queryData2();
         ((ServiceA)AopContext.currentProxy()).doSave(user);
   }

   @Transactional(rollbackFor=Exception.class)
   public void doSave(User user) {
       addData1();
       updateData2();
   }
}
```

4.未被spring管理

在我们平时开发过程中，有个细节很容易被忽略。即使用spring事务的前提是：对象要被Spring管理，需要创建bean实例。

通常情况下，我们通过@Controller、@Service、@Component、@Repository等注解，可以自动实现bean实例化和依赖注入的功能。

当然创建bean实例的方法还有很多，有兴趣的小伙伴可以看看我之前写的另一篇文章《@Autowired的这些骚操作，你都知道吗？》

如果有一天，你匆匆忙忙的开发了一个Service类，但忘了加@Service注解，比如：

```
//@Service
public class UserService {
    @Transactional
    public void add(UserModel userModel) {
         saveData(userModel);
         updateData(userModel);
    }
}
```

从上面的例子，我们可以看到UserService类没有加 **@Service** 注解，那么该类不会交给spring管理，所以它的add方法也不会生成事务。

5.多线程调用

在实际项目开发中，多线程的使用场景还是挺多的。如果Spring事务用在多线程场景中，会有问题吗？

```
@Slf4j
@Service
```

```
public class UserService {

    @Autowired
    private UserMapper userMapper;
    @Autowired
    private RoleService roleService;

    @Transactional
    public void add(UserModel userModel) throws Exception {
        userMapper.insertUser(userModel);
        new Thread(() -> {
            roleService.doOtherThing();
        }).start();
    }
}

@Service
public class RoleService {

    @Transactional
    public void doOtherThing() {
        System.out.println("保存role表数据");
    }
}
```

从上面的例子中，我们可以看到事务方法add中，调用了事务方法doOtherThing，但是事务方法doOtherThing是在另外一个线程中调用的。

这样会导致两个方法不在同一个线程中，获取到的数据库连接不一样，从而是两个不同的事务。如果想doOtherThing方法中抛了异常，add方法也回滚是不可能的。

如果看过spring事务源码的朋友，可能会知道spring的事务是通过数据库连接来实现的。当前线程中保存了一个map，key是数据源，value是数据库连接。

```
private static final ThreadLocal<Map<Object, Object>> resources =
  new NamedThreadLocal<>("Transactional resources");
```

我们说的**同一个事务，其实是指同一个数据库连接，只有拥有同一个数据库连接才能同时提交和回滚。**如果在不同的线程，拿到的数据库连接肯定是不一样的，所以是不同的事务。

6.表不支持事务

周所周知，在mysql5之前，默认的数据库引擎是myisam。

它的好处就不用多说了：索引文件和数据文件是分开存储的，对于查多写少的单表操作，性能比innodb更好。

有些老项目中，可能还在用它。

在创建表的时候，只需要把ENGINE参数设置成MyISAM即可：

```sql
CREATE TABLE `category` (
  `id` bigint NOT NULL AUTO_INCREMENT,
  `one_category` varchar(20) COLLATE utf8mb4_bin DEFAULT NULL,
  `two_category` varchar(20) COLLATE utf8mb4_bin DEFAULT NULL,
  `three_category` varchar(20) COLLATE utf8mb4_bin DEFAULT NULL,
  `four_category` varchar(20) COLLATE utf8mb4_bin DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM AUTO_INCREMENT=4 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
```

myisam好用，但有个很致命的问题是：不支持事务。

如果只是单表操作还好，不会出现太大的问题。但如果需要跨多张表操作，由于其不支持事务，数据极有可能会出现不完整的情况。

此外，myisam还不支持行锁和外键。

所以在实际业务场景中，myisam使用的并不多。在mysql5以后，myisam已经逐渐退出了历史的舞台，取而代之的是innodb。

> 有时候我们在开发的过程中，发现某张表的事务一直都没有生效，那不一定是spring事务的锅，最好确认一下你使用的那张表，是否支持事务。

7.未开启事务

有时候，事务没有生效的根本原因是没有开启事务。

你看到这句话可能会觉得好笑。

开启事务不是一个项目中，最最最基本的功能吗？

为什么还会没有开启事务？

没错，如果项目已经搭建好了，事务功能肯定是有的。

但如果你是在搭建项目demo的时候，只有一张表，而这张表的事务没有生效。那么会是什么原因造成的呢？

当然原因有很多，但没有开启事务，这个原因极其容易被忽略。

如果你使用的是Springboot项目，那么你很幸运。因为springboot通过**DataSourceTransactionManagerAutoConfiguration**类，已经默默的帮你开启了事务。

你所要做的事情很简单，只需要配置**Spring.datasource**相关参数即可。

但如果你使用的还是**传统的Spring项目**，则需要在applicationContext.xml文件中，**手动配置事务相关参数**。如果忘了配置，事务肯定是不会生效的。

具体配置如下信息：

```xml
<!-- 配置事务管理器 -->
<bean
class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
id="transactionManager">
    <property name="dataSource" ref="dataSource"></property>
</bean>
<tx:advice id="advice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="*" propagation="REQUIRED"/>
    </tx:attributes>
</tx:advice>
```

```
<!-- 用切点把事务切进去 -->
<aop:config>
    <aop:pointcut expression="execution(* com.susan.*.*(..))"
id="pointcut"/>
    <aop:advisor advice-ref="advice" pointcut-ref="pointcut"/>
</aop:config>
```

默默的说一句，如果在pointcut标签中的切入点匹配规则，配错了的话，有些类的事务也不会生效。

二 事务不回滚

1.错误的传播特性

其实，我们在使用@Transactional注解时，是可以指定 **propagation** 参数的。

该参数的作用是指定事务的传播特性，spring目前支持7种传播特性：

- `REQUIRED` 如果当前上下文中存在事务，那么加入该事务，如果不存在事务，创建一个事务，这是默认的传播属性值。
- `SUPPORTS` 如果当前上下文存在事务，则支持事务加入事务，如果不存在事务，则使用非事务的方式执行。
- `MANDATORY` 如果当前上下文中存在事务，否则抛出异常。
- `REQUIRES_NEW` 每次都会新建一个事务，并且同时将上下文中的事务挂起，执行当前新建事务完成以后，上下文事务恢复再执行。
- `NOT_SUPPORTED` 如果当前上下文中存在事务，则挂起当前事务，然后新的方法在没有事务的环境中执行。
- `NEVER` 如果当前上下文中存在事务，则抛出异常，否则在无事务环境上执行代码。
- `NESTED` 如果当前上下文中存在事务，则嵌套事务执行，如果不存在事务，则新建事务。

如果我们在手动设置propagation参数的时候，把传播特性设置错了，比如：

```
@Service
public class UserService {

    @Transactional(propagation = Propagation.NEVER)
    public void add(UserModel userModel) {
        saveData(userModel);
        updateData(userModel);
    }
}
```

我们可以看到add方法的事务传播特性定义成了Propagation.NEVER，这种类型的传播特性不支持事务，如果有事务则会抛异常。

目前只有这三种传播特性才会创建新事务：REQUIRED，REQUIRES_NEW，NESTED。

2.自己吞了异常

事务不会回滚，最常见的问题是：开发者在代码中手动try...catch了异常。比如：

```java
@Slf4j
@Service
public class UserService {
    @Transactional
    public void add(UserModel userModel) {
        try {
            saveData(userModel);
            updateData(userModel);
        } catch (Exception e) {
            log.error(e.getMessage(), e);
        }
    }
}
```

这种情况下spring事务当然不会回滚，因为开发者自己捕获了异常，又没有手动抛出，换句话说就是把异常吞掉了。

如果想要spring事务能够正常回滚，必须抛出它能够处理的异常。如果没有抛异常，则spring认为程序是正常的。

3.手动抛了别的异常

即使开发者没有手动捕获异常，但如果抛的异常不正确，spring事务也不会回滚。

```java
@Slf4j
@Service
public class UserService {
    @Transactional
    public void add(UserModel userModel) throws Exception {
        try {
             saveData(userModel);
             updateData(userModel);
        } catch (Exception e) {
            log.error(e.getMessage(), e);
            throw new Exception(e);
        }
    }
}
```

上面的这种情况，开发人员自己捕获了异常，又手动抛出了异常：Exception，事务同样不会回滚。

因为spring事务，**默认情况下只会回滚RuntimeException（运行时异常）和Error（错误）**，对于**普通的Exception（非运行时异常）**，它**不会回滚。**

4.自定义了回滚异常

在使用@Transactional注解声明事务时，有时我们想自定义回滚的异常，spring也是支持的。可以通过设置**rollbackFor**参数，来完成这个功能。

但如果这个参数的值设置错了，就会引出一些莫名其妙的问题，例如：

```java
@Slf4j
@Service
public class UserService {
    @Transactional(rollbackFor = BusinessException.class)
    public void add(UserModel userModel) throws Exception {
       saveData(userModel);
       updateData(userModel);
    }
}
```

如果在执行上面这段代码，保存和更新数据时，程序报错了，抛了SqlException、DuplicateKeyException等异常。而BusinessException是我们自定义的异常，报错的异常不属于BusinessException，所以事务也不会回滚。

即使rollbackFor有默认值，但阿里巴巴开发者规范中，还是要求开发者重新指定该参数。

这是为什么呢?

因为如果使用默认值，一旦程序抛出了Exception，事务不会回滚，这会出现很大的bug。所以，建议一般情况下，将该**参数设置成**：**Exception** 或 **Throwable**。

5.嵌套事务回滚多了

```java
public class UserService {

    @Autowired
    private UserMapper userMapper;

    @Autowired
    private RoleService roleService;

    @Transactional
    public void add(UserModel userModel) throws Exception {
        userMapper.insertUser(userModel);
        roleService.doOtherThing();
    }
}

@Service
public class RoleService {
    @Transactional(propagation = Propagation.NESTED)
    public void doOtherThing() {
        System.out.println("保存role表数据");
    }
}
```

这种情况使用了嵌套的内部事务，原本是希望调用roleService.doOtherThing方法时，如果出现了异常，只回滚doOtherThing方法里的内容，不回滚 userMapper.insertUser里的内容，即回滚保存点。。但事实是，insertUser也回滚了。

why?

因为doOtherThing方法出现了异常，没有手动捕获，会继续往上抛，到外层add方法的代理方法中捕获了异常。所以，这种情况是直接回滚了整个事务，不只回滚单个保存点。

怎么样才能只回滚保存点呢?

```java
@Slf4j
@Service
public class UserService {

    @Autowired
    private UserMapper userMapper;

    @Autowired
    private RoleService roleService;

    @Transactional
    public void add(UserModel userModel) throws Exception {

        userMapper.insertUser(userModel);
        try {
            roleService.doOtherThing();
        } catch (Exception e) {
            log.error(e.getMessage(), e);
        }
    }
}
```

可以将内部嵌套事务放在try/catch中，并且不继续往上抛异常。这样就能保证，如果内部嵌套事务中出现异常，只回滚内部事务，而不影响外部事务。

三 其他

1.大事务问题

在使用Spring事务时，有个让人非常头疼的问题，就是大事务问题。

通常情况下，我们会在方法上**@Transactional**注解，填加事务功能，比如：

```java
@Service
public class UserService {

    @Autowired
    private RoleService roleService;

    @Transactional
    public void add(UserModel userModel) throws Exception {
       query1();
       query2();
       query3();
       roleService.save(userModel);
       update(userModel);
    }
}

@Service
public class RoleService {

    @Autowired
    private RoleService roleService;

    @Transactional
    public void save(UserModel userModel) throws Exception {
```

```
        query4();
        query5();
        query6();
        saveData(userModel);
    }
}
```

但@Transactional注解，如果被加到方法上，有个缺点就是整个方法都包含在事务当中了。

上面的这个例子中，在UserService类中，其实只有这两行才需要事务：

```
roleService.save(userModel);
update(userModel);
```

在RoleService类中，只有这一行需要事务：

```
saveData(userModel);
```

现在的这种写法，会导致所有的query方法也被包含在同一个事务当中。

如果query方法非常多，调用层级很深，而且有部分查询方法比较耗时的话，会造成整个事务非常耗时，而从造成大事务问题。

关于大事务问题的危害，可以阅读一下我的另一篇文章《让人头痛的大事务问题到底要如何解决？》，上面有详细的讲解。

2.编程式事务

上面聊的这些内容都是基于@Transactional注解的，主要说的是它的事务问题，我们把这种事务叫做：**声明式事务。**

其实，Spring还提供了另外一种创建事务的方式，即通过手动编写代码实现的事务，我们把这种事务叫做：**编程式事务。**例如：

```
    @Autowired
    private TransactionTemplate transactionTemplate;

    ...

    public void save(final User user) {
          queryData1();
          queryData2();
          transactionTemplate.execute((status) => {
             addData1();
             updateData2();
             return Boolean.TRUE;
          })
```

```
14      }
```

在Spring中为了支持编程式事务，专门提供了一个类：TransactionTemplate，在它的execute方法中，就实现了事务的功能。

相较于@Transactional注解声明式事务，我更建议大家使用，基于**TransactionTemplate**的编程式事务。主要原因如下：

1. 避免由于Spring aop问题，导致事务失效的问题。
2. 能够更小粒度的控制事务的范围，更直观。

> 建议在项目中少使用@Transactional注解开启事务。但并不是说一定不能用它，如果项目中有些业务逻辑比较简单，而且不经常变动，使用@Transactional注解开启事务开启事务也无妨，因为它更简单，开发效率更高，但是千万要小心事务失效的问题。