

聊聊 spring 事务失效的 12 种场景，太坑了

菜鸟教程 今天

以下文章来源于苏三说技术，作者因为热爱所以坚持ing



苏三说技术

作者就职于知名互联网公司，掘金月度优秀作者，从事开发、架构和部分管理工作。实...

来源 | 苏三说技术

作者 | 因为热爱所以坚持ing

前言

对于从事 java 开发工作的同学来说，spring 的事务肯定再熟悉不过了。

在某些业务场景下，如果一个请求中，需要同时写入多张表的数据。为了保证操作的原子性（要么同时成功，要么同时失败），避免数据不一致的情况，我们一般都会用到 spring 事务。

确实，spring 事务用起来贼爽，就用一个简单的注解：`@Transactional`，就能轻松搞定事务。我猜大部分小伙伴也是这样用的，而且一直用一直爽。

但如果你使用不当，它也会坑你于无形。

今天我们就一起聊聊，事务失效的一些场景，说不定你已经中招了。不信，让我们一起来看看。

访问权限问题

方法用final修饰

方法内部调用

未被spring管理

spring事务失效的12种场景

多线程调用

表不支持事务

错误的传播特性

自己吞了异常

手动抛了别的异常

自定义了回滚异常

嵌套事务回滚多了

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。

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

```
@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。

```
protected TransactionAttribute computeTransactionAttribute(Method method, @Nullable Class<?>
    // 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写，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;

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

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

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：

```
@Service
public class ServiceA {

    @Autowired
    private ServiceB serviceB;
```

```

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

@Service
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：

```

@Service
public class ServiceA {

    @Autowired
    private 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。

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：

```
@Service
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。

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

```
//@Service
```

```
//UserService.java
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

众所周知，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：

```
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，已经默默地帮你开启了事务。

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，事务肯定是不生效的。

具体配置信息如下：

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

默默地说一句，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，则加入该事务，如果不存在事务，则创建一个事务，这是默认的传播属性值。

- **SUPPORTS** 如果当前上下文中存在事务，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** 当前上下文中必须存在事务，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：

```
@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。

```

@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：

```
@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

```
public class UserService {

    @Autowired
```

```

@Service
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。

怎样才能只回滚保存点呢？

```

@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，我们会在方法上加 `@Transactional` 注解，添加事务功能，比如：

```

@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，而造成大事务问题。

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;
    })
}
```

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. 能够更小粒度地控制事务的范围，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。

喜欢此内容的人还喜欢

Spring中的18个注解，那些是你没用过的？

互联网后端架构