一口气怼完12种@Transactional的失效场景

剽悍一小兔 java小白翻身 2023-05-12 06:35 Posted on 江苏

收录于合集
#Java面试
26个 >

数据库事务是后端开发中不可缺少的一块知识点。Spring为了更好的支撑我们进行数据库操作，在框架中支持了两种事务管理的方式：

编程式事务

声明式事务

日常我们进行业务开发时，基本上使用的都是声明式事务，即为使用@Transactional注解的方式。

常规使用时，Spring能帮我们很好的实现数据库的ACID **（这里需要注意哦，Spring只是进行了编程上的事务，最终数据上的事务还是有数据库实现的）** 。

但是，只要是人写的代码，就一定会有Bug。

如果我们不了解@Transactional的失效场景或者说踩坑点，那么在业务开发的过程中总是会出现一些匪夷所思的Bug。

同样它也是面试时高频的考点哦！

本文将罗列@Transactional的失效场景，并分析其失效原因。

一、失效场景集一：代理不生效

Spring中对注解解析的尿性都是基于代理的，如果目标方法无法被Spring代理到，那么它将无法被Spring进行事务管理。

Spring生成代理的方式有两种：

基于接口的JDK动态代理，要求目标代理类需要实现一个接口才能被代理
基于实现目标类子类的CGLIB代理

Spring在2.0之前，目标类如果实现了接口，则使用JDK动态代理方式，否则通过CGLIB子类的方式生成代理。

而在2.0版本之后，如果不在配置文件中显示的指定spring.aop.proxy-tartget-class的值，默认情况下生成代理的方式为CGLIB，如下图

顺着代理的思路，我们来看看哪些情况会因为代理不生效导致事务管控失败。

（1）将注解标注在接口方法上

@Transactional是支持标注在方法与类上的。一旦标注在接口上，对应接口实现类的代理方式如果是CGLIB，将通过生成子类的方式生成目标类的代理，将无法解析到@Transactional，从而事务失效。

这种错误我们还是犯得比较少的，基本上我们都会将注解标注在接口的实现类方法上，官方也不推荐这种。

（2）被final、static关键字修饰的类或方法

CGLIB是通过生成目标类子类的方式生成代理类的，被final、static修饰后，无法继承父类与父类的方法。

（3）类方法内部调用

事务的管理是通过代理执行的方式生效的，如果是方法内部调用，将不会走代理逻辑，也就调用不到了。

例如

```java
    @Override
    @Transactional(rollbackFor = Exception.class)
    @SneakyThrows
    public void createUser(){
        this.createUser1();
        throw new RuntimeException();
    }

    @Override
    @Transactional(rollbackFor = Exception.class,propagation = Propagation.REQUIRES_NEW)
    public void createUser1(){
        UserPO userPO = new UserPO();
        userPO.setUserName("baiyanEventTest");
        userPO.setRealName("柏炎事务测试");
        userPO.setPassword("123");
        userPO.setIp("1.1.1.1");
        userPO.setState(1);
        userPO.setId(1111111L);
        save(userPO);
    }
}
```


在**createUser**中调用了内部方法**createUser1**，并且**createUser1**方法上设置了事务传播策略为：**REQUIRES_NEW**，但是因为是内部直接调用，createUser1不能不代理处理，无法进行事务管理。在createUser1方法抛出异常后就插入数据失败了。

但是这种操作在我们业务开发的过程中貌似还挺常见的，怎么样才能保证其成功呢?

方式1：新建一个Service，将方法迁移过去，有点麻瓜。

方式2：在当前类注入自己，调用createUser1时通过注入的userService调用

方式3：通过AopContext.currentProxy()获取代理对象

```java
@Override
@Transactional(rollbackFor = Exception.class)
@SneakyThrows
public void createUser(){
    ((UserService) AopContext.currentProxy()).createUser1();
    throw new RuntimeException();
}

@Override
@Transactional(rollbackFor = Exception.class,propagation = Propagation.REQUIRES_NEW)
public void createUser1(){
    UserPO userPO = new UserPO();
    userPO.setUserName("baiyanEventTest");
    userPO.setRealName("柏炎事务测试");
    userPO.setPassword("123");
    userPO.setIp("1.1.1.1");
    userPO.setState(1);
    userPO.setId(1111111L);
    save(userPO);
}
```


道理类似于方式2，就是为了通过代理来访问内部方法

（4）当前类没有被Spring管理

这个没什么好说的，都没有被Spring管理成为IOC容器中的一个bean，更别说被事务切面代理到了。

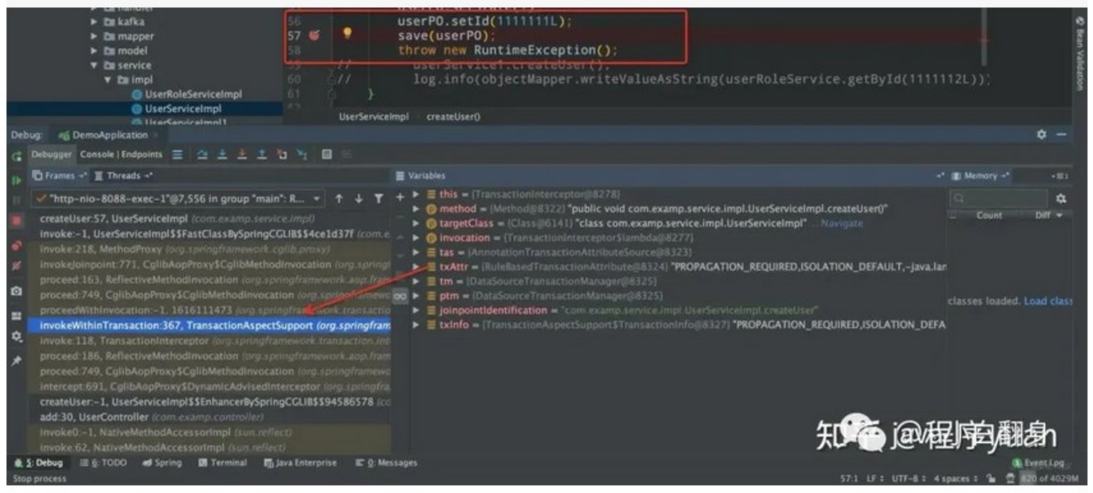
这种Bug看上去比较蠢，但没准真的有人犯错。

二、失效场景集二：框架或底层不支持的功能

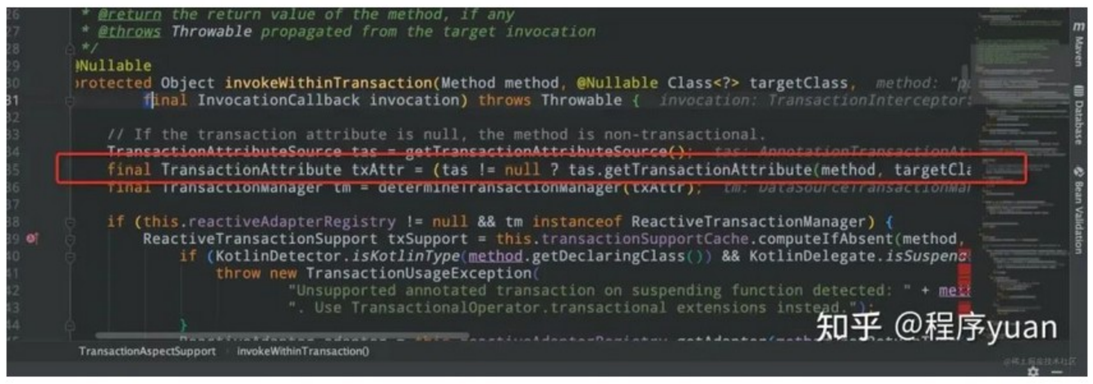
这类失效场景主要聚焦在框架本身在解析@Transactional时的内部支持。如果使用的场景本身就是框架不支持的，那事务也是无法生效的。

（1）非public修饰的方法

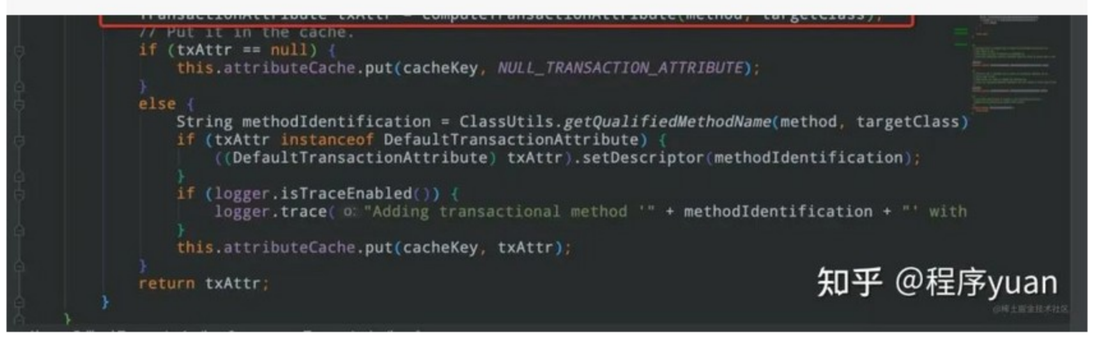
我们在标有@Transactional的任意方法上打个断点，在idea内能看到事务切面点如下图所示



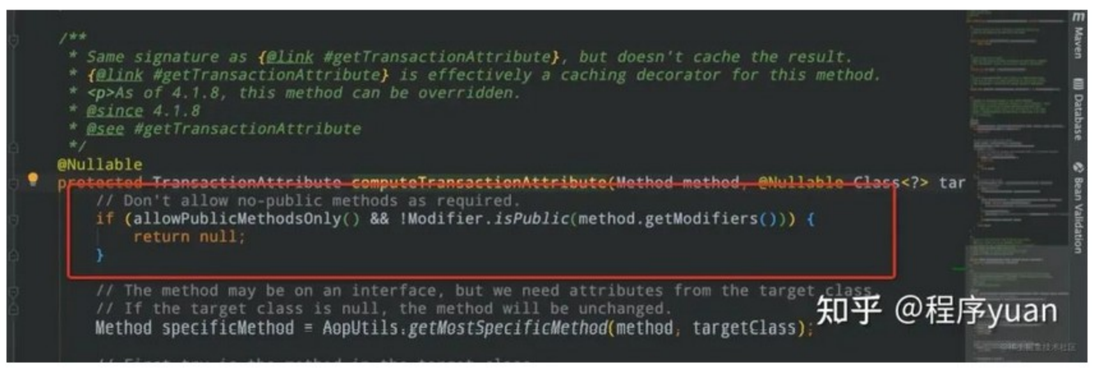
点击去这个方法，在开头有这么一个调用



继续进去



就能看到这么一句话了

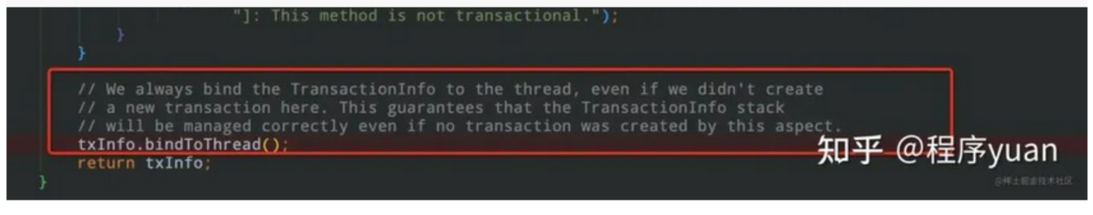


不支持非public修饰的方法进行事务管理。

（2）多线程调用

跟上面一样的的操作，我们能够逐层进入到TransactionAspectSupport.prepareTransactionInfo方法。

注意看以下这段话



从这里我们得知，事务信息是跟线程绑定的。

因此在多线程环境下，事务的信息都是独立的，将会导致Spring在接管事务上出现差异。

这个场景我们要尤其注意！

给大家举个例子

主线程A调用线程B保存Id为1的数据，然后主线程A等待线程B执行完成再通过线程A查询id为1的数据。

这时你会发现在主线程A中无法查询到id为1的数据。因为这两个线程在不同的Spring事务中，本质上会导致它们在Mysql中存在不同的事务中。

Mysql中通过MVCC保证了线程在快照读时只读取小于当前事务号的数据，在线程B显然事务号是大于线程A的，因此查询不到数据。

（3）数据库本身不支持事务

比如Mysql的Myisam存储引擎是不支持事务的，只有innodb存储引擎才支持。

这个问题出现的概率极其小，因为Mysql5之后默认情况下是使用innodb存储引擎了。

但如果配置错误或者是历史项目，发现事务怎么配都不生效的时候，记得看看存储引擎本身是否支持事务。

（4）未开启事务

这个也是一个比较麻瓜的问题，在Springboot项目中已经不存在了，已经有**DataSourceTransactionManagerAutoConfiguration**默认开启了事务管理。

但是在MVC项目中还需要在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。

三、失效场景集三：错误使用@Transactional

注意啦注意啦，下面这几种都是高频会出现的Bug！

（1）错误的传播机制

Spring支持了7种传播机制，分别为：

事务行为	说明
PROPAGATION_REQUIRED	支持当前事务，假设当前没有事务。就新建一个事务
PROPAGATION_SUPPORTS	支持当前事务，假设当前没有事务，就以非事务方式运行
PROPAGATION_MANDATORY	支持当前事务，假设当前没有事务，就抛出异常
PROPAGATION_REQUIRES_NEW	新建事务，假设当前存在事务。把当前事务挂起
PROPAGATION_NOT_SUPPORTED	以非事务方式运行操作。假设当前存在事务，就把当前事务挂起
PROPAGATION_NEVER	以非事务方式运行，假设当前存在事务，则抛出异常
PROPAGATION_NESTED	如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION_REQUIRED类似的操作。

上面不支持事务的传播机制
为：PROPAGATION_SUPPORTS，PROPAGATION_NOT_SUPPORTED，PROPAGATION_NEVER。

如果配置了这三种传播方式的话，在发生异常的时候，事务是不会回滚的。

（2）rollbackFor属性设置错误

默认情况下事务仅回滚运行时异常和Error，不回滚受检异常（例如IOException）。

因此如果方法中抛出了IO异常，默认情况下事务也会回滚失败。

我们可以通过指定@Transactional(rollbackFor = Exception.class)的方式进行全异常捕获。

（3）异常被内部catch

UserService

```
@Autowired
UserService1 userService1;

@Autowired
ObjectMapper objectMapper;

@Override
@Transactional(rollbackFor = Exception.class)
public void createUser(){
    try{
        this.createUser1();
        userService1.createUser();
    }catch (Exception e){

    }
}

@Override
public void createUser1(){
    UserPO userPO = new UserPO();
    userPO.setUserName("baiyanEventTest");
    userPO.setRealName("柏炎事务测试");
    userPO.setPassword("123");
    userPO.setIp("1.1.1.1");
    userPO.setState(1);
    userPO.setId(1111111L);
    save(userPO);
}
```


UserService1

```
    userPO.setPassword("123");
    userPO.setIp("1.1.1.1");
    userPO.setState(1);
    userPO.setId(1111112L);
    save(userPO);
    throw new RuntimeException();
}
```


如上代码UserService调用了UserService1中的方法，并且捕获了UserService1中抛出的异常。

你将能看到控制台出现这样一个报错：

```
org.springframework.transaction.UnexpectedRollbackException: Transactio
复制代码
```

默认情况下标注了@Transactional注解的方法的事务传播机制是**REQUIRED**，它的特性是支持当前事务，也就说加入当前事务。我们在UserService中开始事务，然后再UserService1中抛出异常回滚UserService中的事务，将其标记为只读。

但是在UserSevice中我们捕获了异常，此时UserService上的事务认为正常提交事务。最后在提交时发现事务只读，已经被回滚，则抛出了上述异常。

因此这里如果需要对特定的异常进行捕获处理，记得再次将异常抛出，让最外层的事务感知到。

（4）嵌套事务

上面是我想同时回滚UserService与UserService1。但是也会有这种场景只想回滚UserService1中报错的数据库操作，不影响主逻辑UserService中的数据落库。

有两种方式可以实现上述逻辑：

1.直接在UserService1内的整个方法用try/catch包住

2.在UserService1使用Propagation.REQUIRES_NEW传播机制

```java
@Override
@Transactional(rollbackFor = Exception.class,propagation = Propagation.REQUIRES_NEW)
public void createUser(){
    UserPO userPO = new UserPO();
    userPO.setUserName("baiyanEventTest");
    userPO.setRealName("柏炎事务测试");
    userPO.setPassword("123");
    userPO.setIp("1.1.1.1");
    userPO.setState(1);
    userPO.setId(1111112L);
    save(userPO);
    throw new RuntimeException();
}
```


四、总结

本文为大家分析@Transactional注解使用过程中失效的12种场景

最后，@Transactional注解虽香，但是复杂业务逻辑下，为了更好的管理事务与把控业务处理时事务的细粒度，我还是推荐大家使用编程式事务。

收录于合集 #Java面试 26

上一篇 · 面试官：如何实现一个读写分离的中间件？

People who liked this content also liked

MySQL经典29问！
java小白翻身

Python 高手必会 20 个单行代码
全栈 派

10 种超好用的 MyBatis 写法
java小白翻身