

## 06 | 20%的业务代码的Spring声明式事务，可能都没处理正确

2020-03-21 朱晔

Java 业务开发常见错误 100 例

[进入课程 >](#)



讲述：王少泽

时长 20:43 大小 18.98M



你好，我是朱晔。今天，我来和你聊聊业务代码中与数据库事务相关的坑。

Spring 针对 Java Transaction API (JTA)、JDBC、Hibernate 和 Java Persistence API (JPA) 等事务 API，实现了一致的编程模型，而 Spring 的声明式事务功能更是提供了极其方便的事务配置方式，配合 Spring Boot 的自动配置，大多数 Spring Boot 项目只需要在方法上标记 `@Transactional` 注解，即可一键开启方法的事务性配置。



据我观察，大多数业务开发同学都有事务的概念，也知道如果整体考虑多个数据库操作要么成功要么失败时，需要通过数据库事务来实现多个操作的一致性和原子性。但，在使用上大

多仅限于为方法标记 @Transactional，不会去关注事务是否有效、出错后事务是否正确回滚，也不会考虑复杂的业务代码中涉及多个子业务逻辑时，怎么正确处理事务。

事务没有被正确处理，一般来说不会过于影响正常流程，也不容易在测试阶段被发现。但当系统越来越复杂、压力越来越大之后，就会带来大量的数据不一致问题，随后就是大量的人工介入查看和修复数据。

所以说，一个成熟的业务系统和一个基本可用能完成功能的业务系统，在事务处理细节上的差异非常大。要确保事务的配置符合业务功能的需求，往往不仅仅是技术问题，还涉及产品流程和架构设计的问题。今天这一讲的标题“20% 的业务代码的 Spring 声明式事务，可能都没处理正确”中，20% 这个数字在我看来还是比较保守的。

我今天要分享的内容，就是帮助你在技术问题上理清思路，避免因为事务处理不当让业务逻辑的实现产生大量偶发 Bug。

## 小心 Spring 的事务可能没有生效

在使用 @Transactional 注解开启声明式事务时，第一个最容易忽略的问题是，很可能事务并没有生效。

实现下面的 Demo 需要一些基础类，首先定义一个具有 ID 和姓名属性的 UserEntity，也就是一个包含两个字段的用户表：

 复制代码

```
1 @Entity
2 @Data
3 public class UserEntity {
4     @Id
5     @GeneratedValue(strategy = AUTO)
6     private Long id;
7     private String name;
8
9     public UserEntity() { }
10
11     public UserEntity(String name) {
12         this.name = name;
13     }
14 }
```


为了方便理解，我使用 Spring JPA 做数据库访问，实现这样一个 Repository，新增一个根据用户名查询所有数据的方法：

 复制代码

```
1 @Repository
2 public interface UserRepository extends JpaRepository<UserEntity, Long> {
3     List<UserEntity> findByName(String name);
4 }
```

定义一个 UserService 类，负责业务逻辑处理。如果不清楚 @Transactional 的实现方式，只考虑代码逻辑的话，这段代码看起来没有问题。


定义一个入口方法 createUserWrong1 来调用另一个私有方法 createUserPrivate，私有方法上标记了 @Transactional 注解。当传入的用户名包含 test 关键字时判断为用户名不合法，抛出异常，让用户创建操作失败，期望事务可以回滚：

 复制代码

```
1 @Service
2 @Slf4j
3 public class UserService {
4     @Autowired
5     private UserRepository userRepository;
6
7     //一个公共方法供Controller调用，内部调用事务性的私有方法
8     public int createUserWrong1(String name) {
9         try {
10             this.createUserPrivate(new UserEntity(name));
11         } catch (Exception ex) {
12             log.error("create user failed because {}", ex.getMessage());
13         }
14         return userRepository.findByName(name).size();
15     }
16
17     //标记了@Transactional的private方法
18     @Transactional
19     private void createUserPrivate(UserEntity entity) {
20         userRepository.save(entity);
21         if (entity.getName().contains("test"))
22             throw new RuntimeException("invalid username!");
23     }
24
25     //根据用户名查询用户数
26     public int getUserCount(String name) {
27         return userRepository.findByName(name).size();
28     }
29 }
```

```
28     }
29 }
```

下面是 Controller 的实现，只是调用一下刚才定义的 UserService 中的入口方法 createUserWrong1。

 复制代码

```
1 @Autowired
2 private UserService userService;
3
4
5 @GetMapping("wrong1")
6 public int wrong1(@RequestParam("name") String name) {
7     return userService.createUserWrong1(name);
8 }
```

调用接口后发现，即使用户名不合法，用户也能创建成功。刷新浏览器，多次发现有十几个的非法用户注册。

这里给出 @Transactional 生效原则 1，**除非特殊配置（比如使用 AspectJ 静态织入实现 AOP），否则只有定义在 public 方法上的 @Transactional 才能生效**。原因是，Spring 默认通过动态代理的方式实现 AOP，对目标方法进行增强，private 方法无法代理到，Spring 自然也无法动态增强事务处理逻辑。

你可能会说，修复方式很简单，把标记了事务注解的 createUserPrivate 方法改为 public 即可。在 UserService 中再建一个入口方法 createUserWrong2，来调用这个 public 方法再次尝试：

 复制代码

```
1 public int createUserWrong2(String name) {
2     try {
3         this.createUserPublic(new UserEntity(name));
4     } catch (Exception ex) {
5         log.error("create user failed because {}", ex.getMessage());
6     }
7     return userRepository.findByName(name).size();
8 }
9
10 //标记了@Transactional的public方法
11 @Transactional
```

```

12 public void createUserPublic(UserEntity entity) {
13     userRepository.save(entity);
14     if (entity.getName().contains("test"))
15         throw new RuntimeException("invalid username!");
16 }

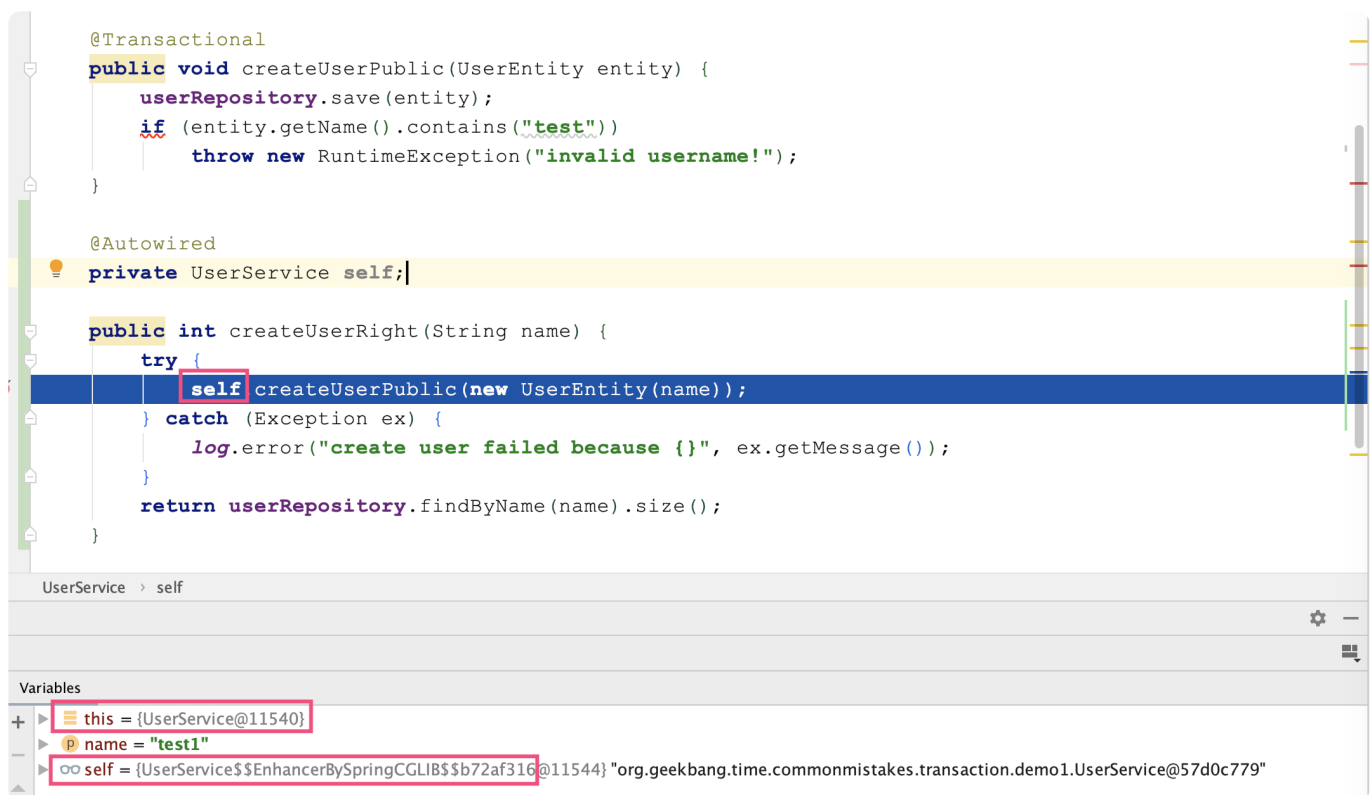
```

测试发现，调用新的 createUserWrong2 方法事务同样不生效。这里，我给出 @Transactional 生效原则 2，**必须通过代理过的类从外部调用目标方法才能生效。**

Spring 通过 AOP 技术对方法进行增强，要调用增强过的方法必然是调用代理后的对象。我们尝试修改下 UserService 的代码，注入一个 self，然后再通过 self 实例调用标记有 @Transactional 注解的 createUserPublic 方法。设置断点可以看到，self 是由 Spring 通过 CGLIB 方式增强过的类：

CGLIB 通过继承方式实现代理类，private 方法在子类不可见，自然也就无法进行事务增强；

this 指针代表对象自己，Spring 不可能注入 this，所以通过 this 访问方法必然不是代理。



The screenshot shows the UserService class in an IDE. The class has a @Transactional method createUserPublic and a @Autowired private UserService self; field. The createUserRight method is shown, which calls self.createUserPublic(new UserEntity(name)). The debugger window shows the self variable as an enhanced class: org.geekbang.time.commonmistakes.transaction.demo1.UserService@57d0c779.

```

@Transactional
public void createUserPublic(UserEntity entity) {
    userRepository.save(entity);
    if (entity.getName().contains("test"))
        throw new RuntimeException("invalid username!");
}

@Autowired
private UserService self;

public int createUserRight(String name) {
    try {
        self.createUserPublic(new UserEntity(name));
    } catch (Exception ex) {
        log.error("create user failed because {}", ex.getMessage());
    }
    return userRepository.findByName(name).size();
}

```

Debugger Variables:

- this = {UserService@11540}
- name = "test1"
- self = {UserService\$\$EnhancerBySpringCGLIB\$\$b72af316@11544} "org.geekbang.time.commonmistakes.transaction.demo1.UserService@57d0c779"

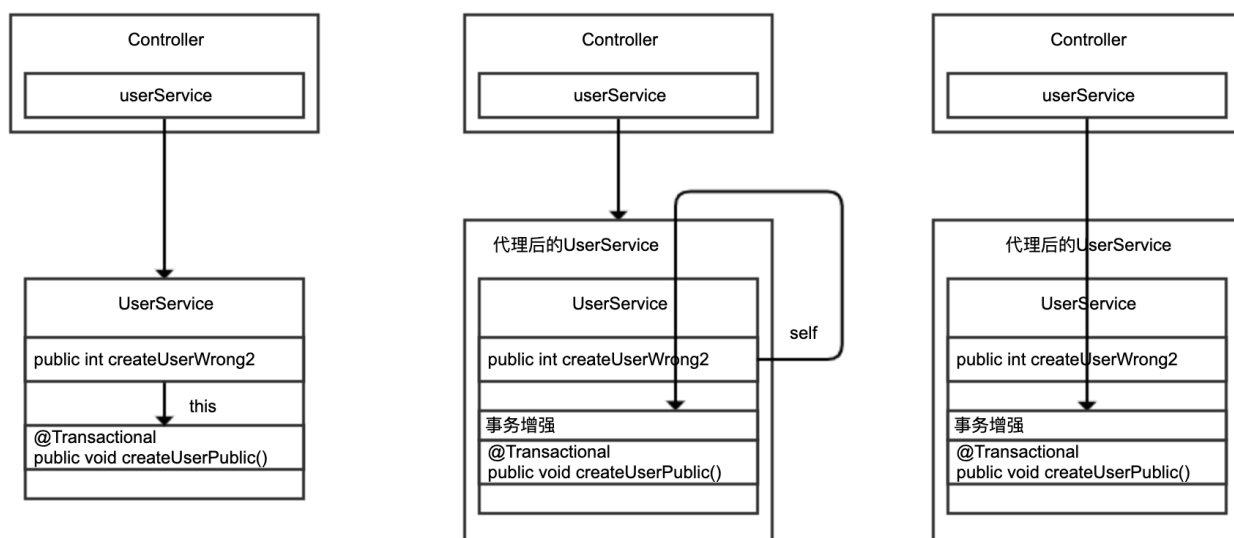
把 this 改为 self 后测试发现，在 Controller 中调用 createUserRight 方法可以验证事务是生效的，非法的用户注册操作可以回滚。

虽然在 UserService 内部注入自己调用自己的 createUserPublic 可以正确实现事务，但更合理的实现方式是，让 Controller 直接调用之前定义的 UserService 的 createUserPublic 方法，因为注入自己调用自己很奇怪，也不符合分层实现的规范：

复制代码

```
1 @GetMapping("right2")
2 public int right2(@RequestParam("name") String name) {
3     try {
4         userService.createUserPublic(new UserEntity(name));
5     } catch (Exception ex) {
6         log.error("create user failed because {}", ex.getMessage());
7     }
8     return userService.getUserCount(name);
9 }
```

我们再通过一张图来回顾下 this 自调用、通过 self 调用，以及在 Controller 中调用 UserService 三种实现的区别：



通过 this 自调用，没有机会走到 Spring 的代理类；后两种改进方案调用的是 Spring 注入的 UserService，通过代理调用才有机会对 `createUserPublic` 方法进行动态增强。

这里，我还有一个小技巧，**强烈建议你在开发时打开相关的 Debug 日志，以方便了解 Spring 事务实现的细节，并及时判断事务的执行情况。**

我们的 Demo 代码使用 JPA 进行数据库访问，可以这么开启 Debug 日志：



```
1 logging.level.org.springframework.orm.jpa=DEBUG
```

开启日志后，我们再比较下在 UserService 中通过 this 调用和在 Controller 中通过注入的 UserService Bean 调用 createUserPublic 区别。很明显，this 调用因为没有走代理，事务没有在 createUserPublic 方法上生效，只在 Repository 的 save 方法层面生效：

```
1 //在UserService中通过this调用public的createUserPublic
2 [10:10:19.913] [http-nio-45678-exec-1] [DEBUG] [o.s.orm.jpa.JpaTransactionMana
3 //在Controller中通过注入的UserService Bean调用createUserPublic
4 [10:10:47.750] [http-nio-45678-exec-6] [DEBUG] [o.s.orm.jpa.JpaTransactionMana
```

你可能还会考虑一个问题，这种实现在 Controller 里处理了异常显得有点繁琐，还不如直接把 createUserWrong2 方法加上 @Transactional 注解，然后在 Controller 中直接调用这个方法。这样一来，既能从外部（Controller 中）调用 UserService 中的方法，方法又是 public 的能够被动态代理 AOP 增强。

你可以试一下这种方法，但很容易就会踩第二个坑，即因为没有正确处理异常，导致事务即便生效也不一定能回滚。

## 事务即便生效也不一定能回滚

通过 AOP 实现事务处理可以理解为，使用 try...catch...来包裹标记了 @Transactional 注解的方法，**当方法出现了异常并且满足一定条件的时候**，在 catch 里面我们可以设置事务回滚，没有异常则直接提交事务。

这里的“一定条件”，主要包括两点。

第一，**只有异常传播出了标记了 @Transactional 注解的方法，事务才能回滚**。在 Spring 的 TransactionAspectSupport 里有个 invokeWithinTransaction 方法，里面就是处理事务的逻辑。可以看到，只有捕获到异常才能进行后续事务处理：

```
1 try {
```

```

2    // This is an around advice: Invoke the next interceptor in the chain.
3    // This will normally result in a target object being invoked.
4    retVal = invocation.proceedWithInvocation();
5 }
6 catch (Throwable ex) {
7     // target invocation exception
8     completeTransactionAfterThrowing(txInfo, ex);
9     throw ex;
10 }
11 finally {
12     cleanupTransactionInfo(txInfo);
13 }

```

**第二，默认情况下，出现 RuntimeException（非受检异常）或 Error 的时候，Spring 才会回滚事务。**

打开 Spring 的 DefaultTransactionAttribute 类能看到如下代码块，可以发现相关证据，通过注释也能看到 Spring 这么做的原因，大概的意思是受检异常一般是业务异常，或者说是类似另一种方法的返回值，出现这样的异常可能业务还能完成，所以不会主动回滚；而 Error 或 RuntimeException 代表了非预期的结果，应该回滚：

 复制代码

```

1  /**
2   * The default behavior is as with EJB: rollback on unchecked exception
3   * ({@link RuntimeException}), assuming an unexpected outcome outside of any
4   * business rules. Additionally, we also attempt to rollback on {@link Error} \
5   * is clearly an unexpected outcome as well. By contrast, a checked exception \
6   * considered a business exception and therefore a regular expected outcome of
7   * transactional business method, i.e. a kind of alternative return value whicl
8   * still allows for regular completion of resource operations.
9   * <p>This is largely consistent with TransactionTemplate's default behavior,
10  * except that TransactionTemplate also rolls back on undeclared checked excep
11  * (a corner case). For declarative transactions, we expect checked exceptions
12  * intentionally declared as business exceptions, leading to a commit by defau
13  * @see org.springframework.transaction.support.TransactionTemplate#execute
14  */
15  @Override
16  public boolean rollbackOn(Throwable ex) {
17      return (ex instanceof RuntimeException || ex instanceof Error);
18  }

```


接下来，我和你分享 2 个反例。



重新实现一下 UserService 中的注册用户操作：

在 createUserWrong1 方法中会抛出一个 RuntimeException，但由于方法内 catch 了所有异常，异常无法从方法传播出去，事务自然无法回滚。

在 createUserWrong2 方法中，注册用户的同时会有一次 otherTask 文件读取操作，如果文件读取失败，我们希望用户注册的数据库操作回滚。虽然这里没有捕获异常，但因为 otherTask 方法抛出的是受检异常，createUserWrong2 传播出去的也是受检异常，事务同样不会回滚。

 复制代码

```
1 @Service
2 @Slf4j
3 public class UserService {
4     @Autowired
5     private UserRepository userRepository;
6
7     //异常无法传播出方法，导致事务无法回滚
8     @Transactional
9     public void createUserWrong1(String name) {
10         try {
11             userRepository.save(new UserEntity(name));
12             throw new RuntimeException("error");
13         } catch (Exception ex) {
14             log.error("create user failed", ex);
15         }
16     }
17
18     //即使出了受检异常也无法让事务回滚
19     @Transactional
20     public void createUserWrong2(String name) throws IOException {
21         userRepository.save(new UserEntity(name));
22         otherTask();
23     }
24
25     //因为文件不存在，一定会抛出一个IOException
26     private void otherTask() throws IOException {
27         Files.readAllLines(Paths.get("file-that-not-exist"));
28     }
29 }
```

Controller 中的实现，仅仅是调用 UserService 的 createUserWrong1 和 createUserWrong2 方法，这里就贴出实现了。这 2 个方法的实现和调用，虽然完全避开

了事务不生效的坑，但因为异常处理不当，导致程序没有如我们期望的文件操作出现异常时回滚事务。


现在，我们来看下修复方式，以及如何通过日志来验证是否修复成功。针对这 2 种情况，对应的修复方法如下。

第一，如果你希望自己捕获异常进行处理的话，也没关系，可以手动设置让当前事务处于回滚状态：

 复制代码


```
1 @Transactional
2 public void createUserRight1(String name) {
3     try {
4         userRepository.save(new UserEntity(name));
5         throw new RuntimeException("error");
6     } catch (Exception ex) {
7         log.error("create user failed", ex);
8         TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
9     }
10 }
```

运行后可以在日志中看到 Rolling back 字样，确认事务回滚了。同时，我们还注意到 “Transactional code has requested rollback” 的提示，表明手动请求回滚：

 复制代码


```
1 [22:14:49.352] [http-nio-45678-exec-4] [DEBUG] [o.s.orm.jpa.JpaTransactionMana
2 [22:14:49.353] [http-nio-45678-exec-4] [DEBUG] [o.s.orm.jpa.JpaTransactionMana
3 [22:14:49.353] [http-nio-45678-exec-4] [DEBUG] [o.s.orm.jpa.JpaTransactionMana
```

第二，在注解中声明，期望遇到所有的 Exception 都回滚事务（来突破默认不回滚受检异常的限制）：

 复制代码

```
1 @Transactional(rollbackFor = Exception.class)
2 public void createUserRight2(String name) throws IOException {
3     userRepository.save(new UserEntity(name));
4     otherTask();
5 }
```

运行后，同样可以在日志中看到回滚的提示：

 复制代码


```
1 [22:10:47.980] [http-nio-45678-exec-4] [DEBUG] [o.s.orm.jpa.JpaTransactionMana;
2 [22:10:47.981] [http-nio-45678-exec-4] [DEBUG] [o.s.orm.jpa.JpaTransactionMana;
```

在这个例子中，我们展现的是一个复杂的业务逻辑，其中有数据库操作、IO 操作，在 IO 操作出现问题时希望让数据库事务也回滚，以确保逻辑的一致性。在有些业务逻辑中，可能会包含多次数据库操作，我们不一定希望将两次操作作为一个事务来处理，这时候就需要仔细考虑事务传播的配置了，否则也可能踩坑。

## 请确认事务传播配置是否符合自己的业务逻辑

有这么一个场景：一个用户注册的操作，会插入一个主用户到用户表，还会注册一个关联的子用户。我们希望将子用户注册的数据库操作作为一个独立事务来处理，即使失败也不会影响主流程，即不影响主用户的注册。

接下来，我们模拟一个实现类似业务逻辑的 UserService：

 复制代码

```
1 @Autowired
2 private UserRepository userRepository;
3
4 @Autowired
5 private SubUserService subUserService;
6
7 @Transactional
8 public void createUserWrong(UserEntity entity) {
9     createMainUser(entity);
10    subUserService.createSubUserWithExceptionWrong(entity);
11 }
12
13 private void createMainUser(UserEntity entity) {
14     userRepository.save(entity);
15     log.info("createMainUser finish");
16 }
```


SubUserService 的 createSubUserWithExceptionWrong 实现正如其名，因为最后我们抛出了一个运行时异常，错误原因是用户状态无效，所以子用户的注册肯定是失败的。我们

期望子用户的注册作为一个事务单独回滚，不影响主用户的注册，这样的逻辑可以实现吗？

 复制代码


```
1 @Service
2 @Slf4j
3 public class SubUserService {
4
5     @Autowired
6     private UserRepository userRepository;
7
8     @Transactional
9     public void createSubUserWithExceptionWrong(UserEntity entity) {
10         log.info("createSubUserWithExceptionWrong start");
11         userRepository.save(entity);
12         throw new RuntimeException("invalid status");
13     }
14 }
```

我们在 Controller 里实现一段测试代码，调用 UserService：

 复制代码

```
1 @GetMapping("wrong")
2 public int wrong(@RequestParam("name") String name) {
3     try {
4         userService.createUserWrong(new UserEntity(name));
5     } catch (Exception ex) {
6         log.error("createUserWrong failed, reason:{}", ex.getMessage());
7     }
8     return userService.getUserCount(name);
9 }
```


调用后可以在日志中发现如下信息，很明显事务回滚了，最后 Controller 打出了创建子用户抛出的运行时异常：

 复制代码

```
1 [22:50:42.866] [http-nio-45678-exec-8] [DEBUG] [o.s.orm.jpa.JpaTransactionMana
2 [22:50:42.869] [http-nio-45678-exec-8] [DEBUG] [o.s.orm.jpa.JpaTransactionMana
3 [22:50:42.869] [http-nio-45678-exec-8] [ERROR] [t.d.TransactionPropagationCont
```


你马上就会意识到，不对呀，因为运行时异常逃出了 @Transactional 注解标记的 createUserWrong 方法，Spring 当然会回滚事务了。如果我们希望主方法不回滚，应该把子方法抛出的异常捕获了。

也就是这么改，把 subUserService.createSubUserWithExceptionWrong 包裹上 catch，这样外层主方法就不会出现异常了：

 复制代码

```
1 @Transactional
2 public void createUserWrong2(UserEntity entity) {
3     createMainUser(entity);
4     try{
5         subUserService.createSubUserWithExceptionWrong(entity);
6     } catch (Exception ex) {
7         // 虽然捕获了异常，但是因为没有开启新事务，而当前事务因为异常已经被标记为rollback
8         log.error("create sub user error:{}", ex.getMessage());
9     }
10 }
```

运行程序后可以看到如下日志：

 复制代码

```
1 [22:57:21.722] [http-nio-45678-exec-3] [DEBUG] [o.s.orm.jpa.JpaTransactionMana
2 [22:57:21.739] [http-nio-45678-exec-3] [INFO ] [t.c.transaction.demo3.SubUserSe
3 [22:57:21.739] [http-nio-45678-exec-3] [DEBUG] [o.s.orm.jpa.JpaTransactionMana
4 [22:57:21.739] [http-nio-45678-exec-3] [DEBUG] [o.s.orm.jpa.JpaTransactionMana
5 [22:57:21.740] [http-nio-45678-exec-3] [DEBUG] [o.s.orm.jpa.JpaTransactionMana
6 [22:57:21.740] [http-nio-45678-exec-3] [DEBUG] [o.s.orm.jpa.JpaTransactionMana
7 [22:57:21.740] [http-nio-45678-exec-3] [ERROR] [.g.t.c.transaction.demo3.UserSe
8 [22:57:21.740] [http-nio-45678-exec-3] [DEBUG] [o.s.orm.jpa.JpaTransactionMana
9 [22:57:21.740] [http-nio-45678-exec-3] [DEBUG] [o.s.orm.jpa.JpaTransactionMana
10 [22:57:21.743] [http-nio-45678-exec-3] [DEBUG] [o.s.orm.jpa.JpaTransactionMana
11 [22:57:21.743] [http-nio-45678-exec-3] [ERROR] [t.d.TransactionPropagationCont
12 org.springframework.transaction.UnexpectedRollbackException: Transaction silen
13 ...
```

需要注意以下几点：

如第 1 行所示，对 createUserWrong2 方法开启了异常处理；

如第 5 行所示，子方法因为出现了运行时异常，标记当前事务为回滚；

如第 7 行所示，主方法的确捕获了异常打印出了 create sub user error 字样；

如第 9 行所示，主方法提交了事务；

奇怪的是，如第 11 行和 12 行所示，**Controller 里出现了一个 UnexpectedRollbackException，异常描述提示最终这个事务回滚了，而且是静默回滚的**。之所以说是静默，是因为 createUserWrong2 方法本身并没有出异常，只不过提交后发现子方法已经把当前事务设置为了回滚，无法完成提交。

这挺反直觉的。**我们之前说，出了异常事务不一定回滚，这里说的却是不出异常，事务也不一定可以提交**。原因是，主方法注册主用户的逻辑和子方法注册子用户的逻辑是同一个事务，子逻辑标记了事务需要回滚，主逻辑自然也不能提交了。

看到这里，修复方式就很明确了，想办法让子逻辑在独立事务中运行，也就是改一下 SubUserService 注册子用户的方法，为注解加上 propagation = Propagation.REQUIRES\_NEW 来设置 REQUIRES\_NEW 方式的事务传播策略，也就是执行到这个方法时需要开启新的事务，并挂起当前事务：

 复制代码

```
1 @Transactional(propagation = Propagation.REQUIRES_NEW)
2 public void createSubUserWithExceptionRight(UserEntity entity) {
3     log.info("createSubUserWithExceptionRight start");
4     userRepository.save(entity);
5     throw new RuntimeException("invalid status");
6 }
```

主方法没什么变化，同样需要捕获异常，防止异常漏出去导致主事务回滚，重新命名为 createUserRight：

 复制代码

```
1 @Transactional
2 public void createUserRight(UserEntity entity) {
3     createMainUser(entity);
4     try{
5         subUserService.createSubUserWithExceptionRight(entity);
6     } catch (Exception ex) {
7         // 捕获异常，防止主方法回滚
8         log.error("create sub user error:{}", ex.getMessage());
9     }
10 }
```



改造后，重新运行程序可以看到如下的关键日志：

第 1 行日志提示我们针对 createUserRight 方法开启了主方法的事务；

第 2 行日志提示创建主用户完成；

第 3 行日志可以看到主事务挂起了，开启了一个新的事务，针对 createSubUserWithExceptionRight 方案，也就是我们的创建子用户的逻辑；

第 4 行日志提示子方法事务回滚；

第 5 行日志提示子方法事务完成，继续主方法之前挂起的事务；

第 6 行日志提示主方法捕获到了子方法的异常；

第 8 行日志提示主方法的事务提交了，随后我们在 Controller 里没看到静默回滚的异常。

 复制代码

```
1 [23:17:20.935] [http-nio-45678-exec-1] [DEBUG] [o.s.orm.jpa.JpaTransactionMana
2 [23:17:21.079] [http-nio-45678-exec-1] [INFO ] [.g.t.c.transaction.demo3.UserSi
3 [23:17:21.082] [http-nio-45678-exec-1] [DEBUG] [o.s.orm.jpa.JpaTransactionMana
4 [23:17:21.153] [http-nio-45678-exec-1] [DEBUG] [o.s.orm.jpa.JpaTransactionMana
5 [23:17:21.160] [http-nio-45678-exec-1] [DEBUG] [o.s.orm.jpa.JpaTransactionMana
6 [23:17:21.161] [http-nio-45678-exec-1] [ERROR] [.g.t.c.transaction.demo3.UserSi
7 [23:17:21.161] [http-nio-45678-exec-1] [DEBUG] [o.s.orm.jpa.JpaTransactionMana
8 [23:17:21.161] [http-nio-45678-exec-1] [DEBUG] [o.s.orm.jpa.JpaTransactionMana
```

运行测试程序看到如下结果，getUserCount 得到的用户数量为 1，代表只有一个用户也就是主用户注册完成了，符合预期：

 localhost:45678/transactionpropagation/right?name=zhuye

1

## 重点回顾

今天，我针对业务代码中最常见的使用数据库事务的方式，即 Spring 声明式事务，与你总结了使用上可能遇到的三类坑，包括：

第一，因为配置不正确，导致方法上的事务没生效。我们务必确认调用 `@Transactional` 注解标记的方法是 `public` 的，并且是通过 Spring 注入的 Bean 进行调用的。

第二，因为异常处理不正确，导致事务虽然生效但出现异常时没回滚。Spring 默认只会对标记 `@Transactional` 注解的方法出现了 `RuntimeException` 和 `Error` 的时候回滚，如果我们的方法捕获了异常，那么需要通过手动编码处理事务回滚。如果希望 Spring 针对其他异常也可以回滚，那么可以相应配置 `@Transactional` 注解的 `rollbackFor` 和 `noRollbackFor` 属性来覆盖其默认设置。

第三，如果方法涉及多次数据库操作，并希望将它们作为独立的事务进行提交或回滚，那么我们需要考虑进一步细化配置事务传播方式，也就是 `@Transactional` 注解的 `Propagation` 属性。

可见，正确配置事务可以提高业务项目的健壮性。但，又因为健壮性问题往往体现在异常情况或一些细节处理上，很难在主流程的运行和测试中发现，导致业务代码的事务处理逻辑往往容易被忽略，因此**我在代码审查环节一直很关注事务是否正确处理**。

如果你无法确认事务是否真正生效，是否按照预期的逻辑进行，可以尝试打开 Spring 的部分 Debug 日志，通过事务的运作细节来验证。也建议你在单元测试时尽量覆盖多的异常场景，这样在重构时，也能及时发现因为方法的调用方式、异常处理逻辑的调整，导致的事务失效问题。

今天用到的代码，我都放在了 GitHub 上，你可以点击 [🔗 这个链接](#) 查看。

## 思考与讨论

1. 考虑到 Demo 的简洁，文中所有数据访问使用的都是 Spring Data JPA。国内大多数互联网业务项目是使用 MyBatis 进行数据访问的，使用 MyBatis 配合 Spring 的声明式事务也同样需要注意文中提到的这些点。你可以尝试把今天的 Demo 改为 MyBatis 做数据访问实现，看看日志中是否可以体现出这些坑。

2. 在第一节中提到，如果要针对 private 方法启用事务，动态代理方式的 AOP 不可行，需要使用静态织入方式的 AOP，也就是在编译期间织入事务增强代码，可以配置 Spring 框架使用 AspectJ 来实现 AOP。你能否参阅 Spring 的文档 “[Using @Transactional with AspectJ](#)” 试试呢？注意：AspectJ 配合 lombok 使用，还可能踩一些坑。

有关数据库事务，你还遇到过去其他坑吗？我是朱晔，欢迎在评论区与我留言分享，也欢迎你把这篇文章分享给你的朋友或同事，一起交流。

点击参与 

进入朱晔老师「读者群」带你  
攻克 Java 业务开发常见错误



添加Java班长，报名入群



新版升级：点击「请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 05 | HTTP调用：你考虑到超时、重试、并发了吗？

下一篇 07 | 数据库索引：索引并不是万能药

## 精选留言 (16)

 写留言



梦倚栏杆

2020-03-21

很多注解貌似都是需要public才能生效比如：@cacheable @async

 2



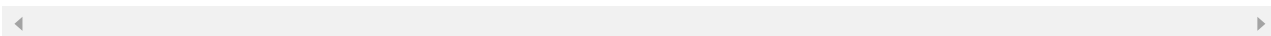
**Monday**

2020-03-21

本节的三个坑，老师总结得很到位，我也理解原理了。但是让我自己看源码，就理不清思绪了。不知道从哪条线哪个类入手。

展开 ∨

作者回复: spring的代码是非常复杂的，理不清思路也很正常，遇到不明白的原理可以先网上搜索一下类似源码剖析的文章，再自己去源码里面看



2

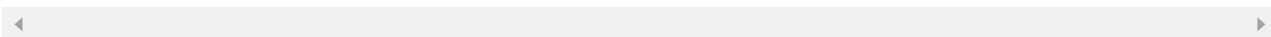


**hanazawakana**

2020-03-21

否则只有定义在 public 方法上的 @Transactional 才能生效。这里一定要用public吗，用protected不行吗，protected在子类中应该也可见啊，是因为包不同吗

作者回复: 这个问题很好，首先JDK动态代理肯定是不行的只能是public，理论上CGLIB方式的代理是可以代理protected方法的，不过如果支持，那么意味着事务可能会因为切换代理实现方式表现不同，大大增加出现Bug的可能性，我觉得为了一致性所以Spring考虑只支持public，这是最好的。



2



**汝林外史**

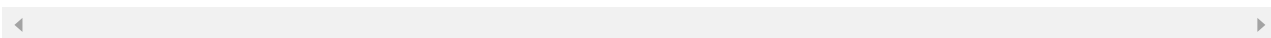
2020-03-23

很明显，this 调用因为没有走代理，事务没有在 createUserPublic 方法上生效，只在 Repository 的 save 方法层面生效。

createUserPublic这个方法不是本来就一个save操作吗，既然save层面生效了，那这个方法的事务难道不也就生效了吗？

展开 ∨

作者回复: 生效，但是出异常的并不是save本身，所以Spring无法回滚



1



**Darren**

2020-03-23

AspectJ与lombok，都是字节码层面进行增强，在一起使用时会有问题，根据AspectJ维

护者Andy Clement的当前答案是由于ECJ (Eclipse Compiler for Java) 软件包存在问题在AspectJ编译器基础结构中包含和重命名。

解决问题可以参考下面连接:

<http://aspectj.2085585.n4.nabble.com/AspectJ-with-Lombok-td4651540.html...>

展开 ▾

作者回复: 📬



1



**Seven.Lin** 澤耿

2020-03-22

我还遇到一个坑, 就是子方法使用了REQUIRES\_NEW, 但是业务逻辑需要的数据是来源于父方法的, 也就是父方法还没提交, 子方法获取不到。当时的解决方法是把事务隔离级别改成RC, 现在回想起来, 不知道这种解决方法是否正确?

展开 ▾

作者回复: 你说的隔离级别应该是指READ\_UNCOMMITTED。我不认为这是很好的解决方案, 子方法内需要依赖的数据来自父方法, 可以方法传值, 而不是用这种隔离级别。



1



**Seven.Lin** 澤耿

2020-03-22

老师, 可以问一下为啥国内大多数公司使用MyBatis呢? 是为了更加接近SQL吗? 难道国外业务不会遇到复杂的场景吗?

展开 ▾

作者回复: 1、容易上手简单

2、国内BAT大厂对于Mybatis的使用量大, 影响力大

3、国内大部分项目还是面向表结构的编程, 从下到上的思考方式而非OOP的思考方式



1



**九时四**

2020-03-21

老师您好, 有个数据库事务和spring事务的问题想请教下 (我是一个入职半年的菜鸟)。业务场景: 为了实现同一个时间的多个请求, 只有一个请求生效, 在数据库字段上加了一个字段 (signature\_lock) 标识锁状态。 (没有使用redis锁之类的中间件, 只讨论数据库

事务和Spring的事务，以下的请求理解为同时请求)

...

展开 ▾

作者回复: 如果要通过数据库来实现锁，那么加锁解锁，需要是单独的事务，不能跟业务的sql事务混合在一起，加锁和业务在一个事务里了，锁就没用了，因为每个事务里，都认为自己拿到了锁。



1



水户洋平

2020-03-24

看完了老师讲的这篇文章，发现自己用@Transactional的时候，也只是为了用而用，并不看它的真实效果，就会导致以后的数据不正确，有脏数据。今天把案例中的代码敲一遍，然后再看自己项目中的用法是否正确。

作者回复: 👍



2020-03-23

Spring的坑，看来还需要多读些源码啊。

展开 ▾



刘楠

2020-03-22

学习到了，这几个坑，

展开 ▾



每天晒白牙

2020-03-22

工作中的业务场景没用到事务，但老师说的这些坑，我也不清楚，先学习了



mz

2020-03-21



看完了，我滚去检查代码了。 😊

展开 ▾



**pedro**

2020-03-21

AspectJ 配合 lombok，需要使用 aspectj 插件的就地编织功能，大致会修改 pom.xml 配置如下：

```
<showWeaveInfo/>  
<forceAjcCompile>true</forceAjcCompile> ...
```

展开 ▾



**傲宇**

2020-03-21

Spring事务的用法感觉已经是spring用户的常识性问题了，但奈何还是有人会犯。我给出一个神奇的例子：我公司有一个同事，在controller调用多个service的事务方法，当后面的业务失败前面的业务方法自然就产生出不少的脏数据。

展开 ▾



**Jialin**

2020-03-21

这节spring事务操作非常实用，减少开发踩坑，好好研究

