一口气说出 6种，@Transactional注解的失效场景

2020年03月19日 11:48 · 阅读 34263


目录

> 整理了一些Java方面的架构、面试资料（微服务、集群、
> 以关注公众号【程序员内点事】，无套路自行领取

引言

昨天公众号粉丝咨询了一个问题，说自己之前面试被问 `@Transactional` 注解哪些场景下会失效，一时语塞致使面试失败。所以今天简单的和大家分享一下 `@Transactional` 相关的知识。

`@Transactional` 注解相信大家并不陌生，平时开发中很常用的一个注解，它能保证方法内多个数据库操作要么同时成功、要么同时失败。使用 `@Transactional` 注解时需要注意许多的细节，不然你会发现 `@Transactional` 总是莫名其妙的就失效了。

一、事务

事务 { 编程式事务
声明式事务 { xml配置文件
注解

事务管理在系统开发中是不可缺少的一部分， `Spring` 提供了很好事务管理机制，主要分为 `编程式事务` 和 `声明式事务` 两种。

编程式事务：是指在代码中手动的管理事务的提交、回滚等操作，代码侵入性比较强，如下示例：

```php
try {
    //TODO something
     transactionManager.commit(status);
} catch (Exception e) {
    transactionManager.rollback(status);
    throw new InvoiceApplyException("异常失败");
}
```

声明式事务：基于 `AOP` 面向切面的，它将具体业务与事务处理部分解耦，代码侵入性很低，所以在实际开发中声明式事务用的比较多。声明式事务也有两种实现方式，一是基于 `TX` 和 `AOP` 的xml配置文件方式，二种就是基于@Transactional注解了。

```less
    @Transactional
    @GetMapping("/test")
    public String test() {

        int insert = cityInfoDictMapper.insert(cityInfoDict);
    }
```

二、@Transactional介绍

1、@Transactional注解可以作用于哪些地方?

@Transactional 可以作用在 `接口` 、 `类` 、 `类方法` 。

- **作用于类**：当把@Transactional 注解放在类上时，表示所有该类的 `public` 方法都配置相同的事务属性信息。
- **作用于方法**：当类配置了@Transactional，方法也配置了@Transactional，方法的事务会覆盖类的事务配置信息。
- **作用于接口**：不推荐这种使用方法，因为一旦标注在Interface上并且配置了Spring AOP 使用CGLib动态代理，将会导致@Transactional注解失效

```less
@Transactional
@RestController
@RequestMapping
public class MybatisPlusController {
    @Autowired
    private CityInfoDictMapper cityInfoDictMapper;

    @Transactional(rollbackFor = Exception.class)
    @GetMapping("/test")
    public String test() throws Exception {
        CityInfoDict cityInfoDict = new CityInfoDict();
        cityInfoDict.setParentCityId(2);
        cityInfoDict.setCityName("2");
        cityInfoDict.setCityLevel("2");
        cityInfoDict.setCityCode("2");
        int insert = cityInfoDictMapper.insert(cityInfoDict);
        return insert + "";
    }
}
```

2、@Transactional注有哪些属性?

REQUIRED：有则加入，无则创建
SUPPORTS：有则加入
MANDATORY：有则加入，无则抛异常
REQUIRED：创建新事务

propagation属性

`propagation` 代表事务的传播行为，默认值为 `Propagation.REQUIRED` ，其他的属性信息如下：

有则加入，无则创建

- `Propagation.REQUIRED` ：如果当前存在事务，则加入该事务，如果当前不存在事务，则创建一个新的事务。(也就是说如果A方法和B方法都添加了注解，在默认传播模式下，A方法内部调用B方法，会把两个方法的事务合并为一个事务)

有则加入

- `Propagation.SUPPORTS` ：如果当前存在事务，则加入该事务；如果当前不存在事务，则以非事务的方式继续运行。

有则加入 无则抛异常

强制的

- `Propagation.MANDATORY` ：如果当前存在事务，则加入该事务；如果当前不存在事务，则抛出异常。

创建新事务

- `Propagation.REQUIRES_NEW` ：重新创建一个新的事务，如果当前存在事务，暂停当前的事务。(当类A中的 a 方法用默认 `Propagation.REQUIRED` 模式，类B中的 b方法加上采用 `Propagation.REQUIRES_NEW` 模式，然后在 a 方法中调用 b方法操作数据库，然而 a方法抛出异常后，b方法并没有进行回滚，因为 `Propagation.REQUIRES_NEW` 会暂停 a方法的事务)

从非事务运行

- `Propagation.NOT_SUPPORTED` ：以非事务的方式运行，如果当前存在事务，暂停当前的事务。

从非事务执行 有则抛异常

- `Propagation.NEVER` ：以非事务的方式运行，如果当前存在事务，则抛出异常。
- `Propagation.NESTED` ：和 Propagation.REQUIRED 效果一样。

isolation 属性

`isolation` ：事务的隔离级别，默认值为 `Isolation.DEFAULT` 。

低
隔离级别
高

- Isolation.DEFAULT：使用底层数据库默认的隔离级别。
- Isolation.READ_UNCOMMITTED：一个事务能读取其它事务修改，但还没有提交的数据
- Isolation.READ_COMMITTED：一个事务能读取其它事务提交的数据
- Isolation.REPEATABLE_READ：一个事务一旦开始，事务过程中所读取的所有数据不允许被其它事务修改
- Isolation.SERIALIZABLE：系统中所有事务以串行方式逐个执行

timeout 属性

`timeout` ：事务的超时时间，默认值为 -1。如果超过该时间限制但事务还没有完成，则自动回滚事务。

readOnly 属性

`readOnly` ：指定事务是否为只读事务，默认值为 false；为了忽略那些不需要事务的方法，比如读取数据，可以设置 read-only 为 true。

rollbackFor 属性

`rollbackFor` ：用于指定能够触发事务回滚的异常类型，可以指定多个异常类型。

noRollbackFor属性**

`noRollbackFor`：抛出指定的异常类型，不回滚事务，也可以指定多个异常类型。

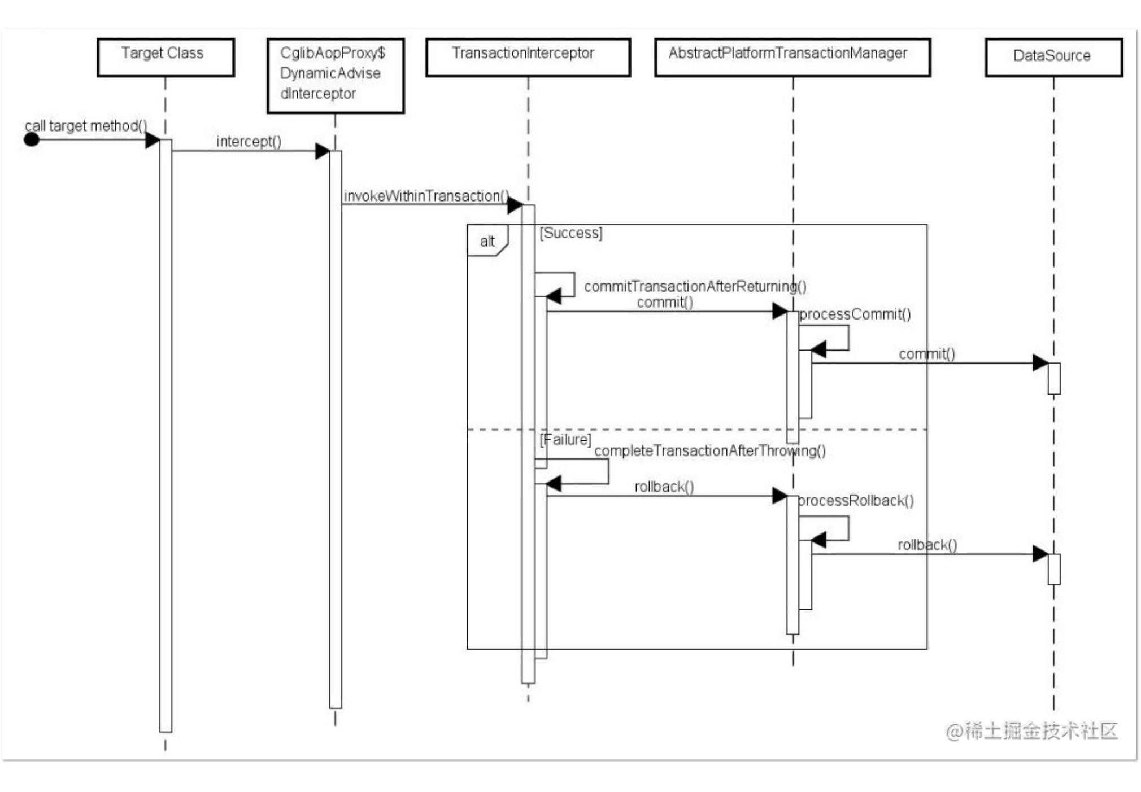
二、@Transactional失效场景

1.@Transactional应用在非public方法上
2.@Transactional中的Propagation属性设置错误
3.@Transactional中的rollbackFor属性设置错误
4.A方法调用B方法，A中没有声明事务，B中声明事务 但Propagation属性值为默认值
5.异常被catch吃掉了
6.数据库引擎不支持事务

接下来我们结合具体的代码分析一下哪些场景下，@Transactional 注解会失效。

1、@Transactional 应用在非 public 修饰的方法上

如果 `Transactional` 注解应用在非 `public` 修饰的方法上，Transactional将会失效。



之所以会失效是因为在Spring AOP 代理时，如上图所示 `TransactionInterceptor` （事务拦截器）在目标方法执行前后进行拦截， `DynamicAdvisedInterceptor` （CglibAopProxy 的内部类）的 intercept 方法或 `JdkDynamicAopProxy` 的 invoke 方法会间接调用 `AbstractFallbackTransactionAttributeSource` 的 `computeTransactionAttribute` 方法，获取Transactional 注解的事务配置信息。

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

此方法会检查目标方法的修饰符是否为 public，不是 public则不会获取@Transactional 的属性配置信息。

注意： `protected` 、 `private` 修饰的方法上使用 `@Transactional` 注解，虽然事务无效，但不会有任何报错，这是我们很容犯错的一点。

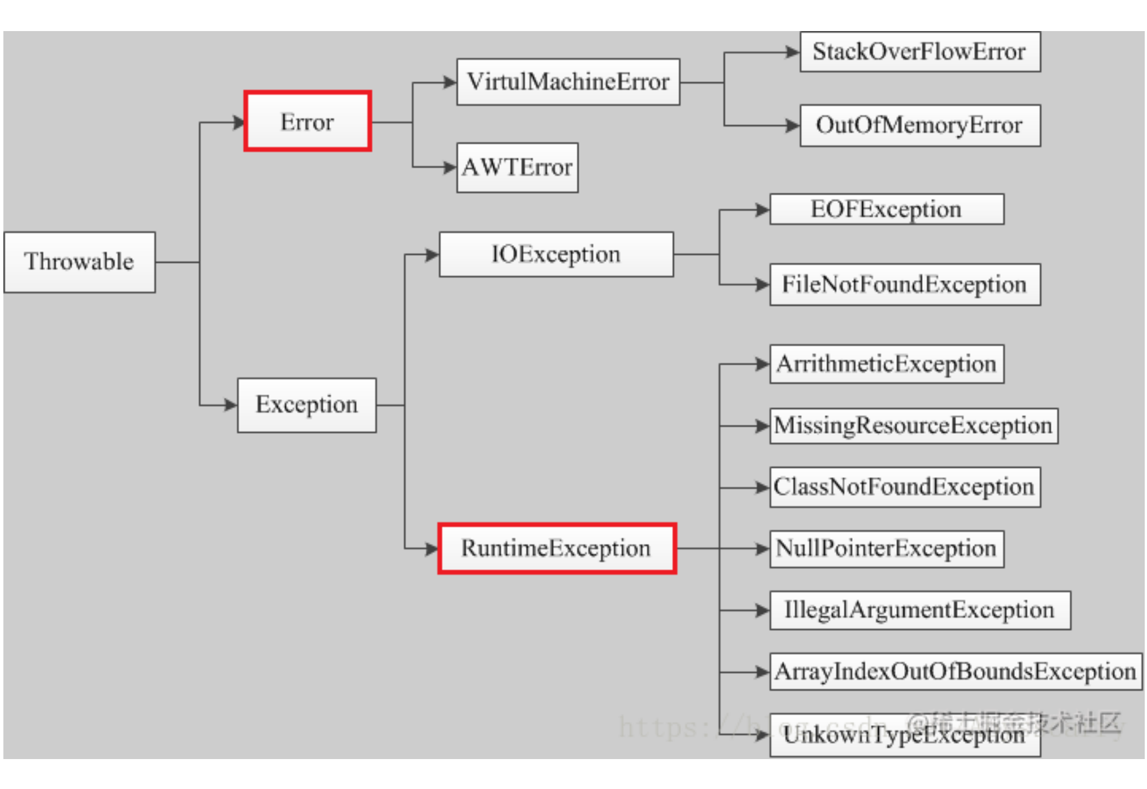
2、@Transactional 注解属性 propagation 设置错误

这种失效是由于配置错误，若是错误的配置以下三种 propagation，事务将不会发生回滚。

`TransactionDefinition.PROPAGATION_SUPPORTS` ：如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。 `TransactionDefinition.PROPAGATION_NOT_SUPPORTED` ：以非事务方式运行，如果当前存在事务，则把当前事务挂起。 `TransactionDefinition.PROPAGATION_NEVER` ：以非事务方式运行，如果当前存在事务，则抛出异常。

3、@Transactional 注解属性 rollbackFor 设置错误

`rollbackFor` 可以指定能够触发事务回滚的异常类型。Spring默认抛出了未检查 `unchecked` 异常（继承自 `RuntimeException` 的异常）或者 `Error` 才回滚事务；其他异常不会触发回滚事务。如果在事务中抛出其他类型的异常，但却期望 Spring 能够回滚事务，就需要指定 **rollbackFor**属性。



```dart
// 希望自定义的异常可以进行回滚
@Transactional(propagation= Propagation.REQUIRED,rollbackFor= MyException.class
```

若在目标方法中抛出的异常是 `rollbackFor` 指定的异常的子类，事务同样会回滚。Spring源码如下：

```kotlin
private int getDepth(Class<?> exceptionClass, int depth) {
        if (exceptionClass.getName().contains(this.exceptionName)) {
            // Found it!
            return depth;
}
        // If we've gone as far as we can go and haven't found it...
        if (exceptionClass == Throwable.class) {
            return -1;
}
return getDepth(exceptionClass.getSuperclass(), depth + 1);
}
```

4、同一个类中方法调用，导致@Transactional失效

开发中避免不了会对同一个类里面的方法调用，比如有一个类Test，它的一个方法A，A再调用本类的方法B（不论方法B是用public还是private修饰），但方法A没有声明注解事务，而B方法有。则外部调用方法A之后，方法B的事务是不会起作用的。这也是经常犯错误的一个地方。

那为啥会出现这种情况？其实这还是由于使用 `Spring AOP` 代理造成的，因为只有当事务方法被当前类以外的代码调用时，才会由 `Spring` 生成的代理对象来管理。

```java
//@Transactional
    @GetMapping("/test")
    private Integer A() throws Exception {
        CityInfoDict cityInfoDict = new CityInfoDict();
        cityInfoDict.setCityName("2");
        /**
         * B 插入字段为 3的数据
         */
        this.insertB();
        /**
         * A 插入字段为 2的数据
         */
        int insert = cityInfoDictMapper.insert(cityInfoDict);

        return insert;
    }

    @Transactional()
    public Integer insertB() throws Exception {
        CityInfoDict cityInfoDict = new CityInfoDict();
        cityInfoDict.setCityName("3");
        cityInfoDict.setParentCityId(3);

        return cityInfoDictMapper.insert(cityInfoDict);
    }
```

5、异常被你的 catch“吃了”导致@Transactional失效

这种情况是最常见的一种@Transactional注解失效场景，

```ini
    @Transactional
    private Integer A() throws Exception {
        int insert = 0;
        try {
            CityInfoDict cityInfoDict = new CityInfoDict();
            cityInfoDict.setCityName("2");
            cityInfoDict.setParentCityId(2);
            /**
             * A 插入字段为 2的数据
             */
            insert = cityInfoDictMapper.insert(cityInfoDict);
            /**
             * B 插入字段为 3的数据
             */
            b.insertB();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

如果B方法内部抛了异常，而A方法此时try catch了B方法的异常，那这个事务还能正常回滚吗？

答案：不能！

会抛出异常：

```sql
org.springframework.transaction.UnexpectedRollbackException: Transaction rolled back because it has bee
```

因为当 `ServiceB` 中抛出了一个异常以后， `ServiceB` 标识当前事务需要 `rollback` 。但是 `ServiceA` 中由于你手动的捕获这个异常并进行处理， `ServiceA` 认为当前事务应该正常 `commit` 。此时就出现了前后不一致，也就是因为这样，抛出了前面的 `UnexpectedRollbackException` 异常。

`spring` 的事务是在调用业务方法之前开始的，业务方法执行完毕之后才执行 `commit` or `rollback` ，事务是否执行取决于是否抛出 `runtime异常` 。如果抛出 `runtime exception` 并在你的业务方法中没有catch到的话，事务会回滚。

在业务方法中一般不需要catch异常，如果非要catch一定要抛出 `throw new RuntimeException()` ，或者注解中指定抛异常类型 `@Transactional(rollbackFor=Exception.class)` ，否则会导致事务失效，数据commit造成数据不一致，所以有些时候try catch反倒会画蛇添足。

6、数据库引擎不支持事务

这种情况出现的概率并不高，事务能否生效数据库引擎是否支持事务是关键。常用的MySQL数据库默认使用支持事务的 `innodb` 引擎。一旦数据库引擎切换成不支持事务的 `myisam` ，那事务就从根本上失效了。

总结

@Transactional 注解的看似简单易用，但如果对它的用法一知半解，还是会踩到很多坑的。

今天就说这么多，如果本文对您有一点帮助，希望能得到您一个点赞👍哦

您的认可才是我写作的动力！

小福利：

几百本各类技术电子书相送，嘘~，**免费** 送给小伙伴们。关注公众号回复【666】自行领取