



## 19 | Spring 事务常见错误（上）

2021-06-04 傅健

《Spring编程常见错误50例》

[课程介绍 >](#)



讲述：傅健

时长 11:11 大小 10.25M



你好，我是傅健。

通过上节课的学习，我们了解了 Spring Data 操作数据库的一些常见问题。这节课我们聊一聊数据库操作中的一个非常重要的话题——事务管理。

Spring 事务管理包含两种配置方式，第一种是使用 XML 进行模糊匹配，绑定事务管理；第二种是使用注解，这种方式可以对每个需要进行事务处理的方法进行单独配置，你只需要添加上 `@Transactional`，然后在注解内添加属性配置即可。在我们的错误案例示范中，我们统一使用更为方便的注解式方式。



另外，补充一点，Spring 在初始化时，会通过扫描拦截对事务的方法进行增强。如果目标方法存在事务，Spring 就会创建一个 Bean 对应的代理（Proxy）对象，并进行相关的事

务处理操作。

在正式开始讲解事务之前，我们需要搭建一个简单的 Spring 数据库的环境。这里我选择了当下最为流行的 MySQL + Mybatis 作为数据库操作的基本环境。为了正常使用，我们还需要引入一些配置文件和类，简单列举一下。

## 1. 数据库配置文件 jdbc.properties，配置了数据连接信息。

[复制代码](#)

```
1 jdbc.driver=com.mysql.cj.jdbc.Driver
2
3 jdbc.url=jdbc:mysql://localhost:3306/spring?useUnicode=true&characterEncoding=
4
5 jdbc.username=root
6 jdbc.password=pass
```

## 2. JDBC 的配置类，从上述 jdbc.properties 加载相关配置项，并创建 JdbcTemplate、DataSource、TransactionManager 相关的 Bean 等。

[复制代码](#)

```
1 public class JdbcConfig {
2     @Value("${jdbc.driver}")
3     private String driver;
4
5     @Value("${jdbc.url}")
6     private String url;
7
8     @Value("${jdbc.username}")
9     private String username;
10
11     @Value("${jdbc.password}")
12     private String password;
13
14     @Bean(name = "jdbcTemplate")
15     public JdbcTemplate createJdbcTemplate(DataSource dataSource) {
16         return new JdbcTemplate(dataSource);
17     }
18
19     @Bean(name = "dataSource")
20     public DataSource createDataSource() {
21         DriverManagerDataSource ds = new DriverManagerDataSource();
22         ds.setDriverClassName(driver);
23         ds.setUrl(url);
```

```
24         ds.setUsername(username);
25         ds.setPassword(password);
26         return ds;
27     }
28
29     @Bean(name = "transactionManager")
30     public PlatformTransactionManager createTransactionManager(DataSource
31         return new DataSourceTransactionManager(dataSource);
32     }
33 }
```

3. 应用配置类，通过注解的方式，配置了数据源、MyBatis Mapper 的扫描路径以及事务等。

[复制代码](#)

```
1 @Configuration
2 @ComponentScan
3 @Import({JdbcConfig.class})
4 @PropertySource("classpath:jdbc.properties")
5 @MapperScan("com.spring.puzzle.others.transaction.example1")
6 @EnableTransactionManagement
7 @EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
8 @EnableAspectJAutoProxy(proxyTargetClass = true, exposeProxy = true)
9 public class AppConfig {
10     public static void main(String[] args) throws Exception {
11         ApplicationContext context = new AnnotationConfigApplicationContext(App
12     }
13 }
```

完成了上述基础配置和代码后，我们开始进行案例的讲解。

## 案例 1：unchecked 异常与事务回滚

在系统中，我们需要增加一个学生管理的功能，每一位新生入学后，都会往数据库里存入学生的信息。我们引入了一个学生类 Student 和与之相关的 Mapper。

其中，Student 定义如下：

[复制代码](#)

```
1 public class Student implements Serializable {
2     private Integer id;
3     private String realname;
4     public Integer getId() {
```

```
5         return id;
6     }
7     public void setId(Integer id) {
8         this.id = id;
9     }
10    public String getRealname() {
11        return realname;
12    }
13    public void setRealname(String realname) {
14        this.realname = realname;
15    }
16 }
17
```

Student 对应的 Mapper 类定义如下：

[复制代码](#)

```
1 @Mapper
2 public interface StudentMapper {
3     @Insert("INSERT INTO `student`(`realname`) VALUES (#{realname})")
4     void saveStudent(Student student);
5 }
6
```

对应数据库表的 Schema 如下：

[复制代码](#)

```
1 CREATE TABLE `student` (
2   `id` int(11) NOT NULL AUTO_INCREMENT,
3   `realname` varchar(255) DEFAULT NULL,
4   PRIMARY KEY (`id`)
5 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

业务类 StudentService，其中包括一个保存的方法 saveStudent。执行一下保存，一切正常。

接下来，我们想要测试一下这个事务会不会回滚，于是就写了这样一段逻辑：如果发现用户名是小明，就直接抛出异常，触发事务的回滚操作。

[复制代码](#)

```
1 @Service
2 public class StudentService {
3     @Autowired
4     private StudentMapper studentMapper;
5
6     @Transactional
7     public void saveStudent(String realname) throws Exception {
8         Student student = new Student();
9         student.setRealname(realname);
10        studentMapper.saveStudent(student);
11        if (student.getRealname().equals("小明")) {
12            throw new Exception("该学生已存在");
13        }
14    }
15 }
16 }
```

然后使用下面的代码来测试一下，保存一个叫小明的学生，看会不会触发事务的回滚。

[复制代码](#)

```
1 public class AppConfig {
2     public static void main(String[] args) throws Exception {
3         ApplicationContext context = new AnnotationConfigApplicationContext(Ap
4         StudentService studentService = (StudentService) context.getBean("stud
5         studentService.saveStudent("小明");
6     }
7 }
```

执行结果打印出了这样的信息：

[复制代码](#)

```
1 Exception in thread "main" java.lang.Exception: 该学生已存在
2     at com.spring.puzzle.others.transaction.example1.StudentService.saveStudent(
```

可以看到，异常确实被抛出来，但是检查数据库，你会发现数据库里插入了一条新的记录。

但是我们的常规思维可能是：在 Spring 里，抛出异常，就会导致事务回滚，而回滚以后，是不应该有数据存入数据库才对啊。而在这个案例中，异常也抛了，回滚却没有如期而至，这是什么原因呢？我们需要研究一下 Spring 的源码，来找找答案。

## 案例解析

我们通过 debug 沿着 saveStudent 继续往下跟，得到了一个这样的调用栈：

```
✓ "main"@1 in group "main": RUNNING
rollbackOn:187, DefaultTransactionAttribute (org.springframework.transaction.interceptor)
rollbackOn:156, RuleBasedTransactionAttribute (org.springframework.transaction.interceptor)
rollbackOn:65, DelegatingTransactionAttribute (org.springframework.transaction.interceptor)
completeTransactionAfterThrowing:670, TransactionAspectSupport (org.springframework.transaction.interceptor)
invokeWithinTransaction:392, TransactionAspectSupport (org.springframework.transaction.interceptor)
invoke:119, TransactionInterceptor (org.springframework.transaction.interceptor)
proceed:186, ReflectiveMethodInvocation (org.springframework.aop.framework)
proceed:750, CglibAopProxy$CglibMethodInvocation (org.springframework.aop.framework)
intercept:692, CglibAopProxy$DynamicAdvisedInterceptor (org.springframework.aop.framework)
saveStudent:-1, StudentService$$EnhancerBySpringCGLIB$$f71996a8 (learning.spring.ioc.service)
main:10, App (learning.spring.ioc.service)
```

从这个调用栈中我们看到了熟悉的 CglibAopProxy，另外事务本质上也是一种特殊的切面，在创建的过程中，被 CglibAopProxy 代理。事务处理的拦截器是 TransactionInterceptor，它支撑着整个事务功能的架构，我们来分析下这个拦截器是如何实现事务特性的。

首先，TransactionInterceptor 继承类 TransactionAspectSupport，实现了接口 MethodInterceptor。当执行代理类的目标方法时，会触发 invoke()。由于我们的关注重点是在异常处理上，所以直奔主题，跳到异常处理相关的部分。当它 catch 到异常时，会调用 completeTransactionAfterThrowing 方法做进一步处理。

复制代码

```
1 protected Object invokeWithinTransaction(Method method, @Nullable Class<?> targetClass,
2     final InvocationCallback invocation) throws Throwable {
3     //省略非关键代码
4     Object retVal;
5     try {
6         retVal = invocation.proceedWithInvocation();
7     }
8     catch (Throwable ex) {
9         completeTransactionAfterThrowing(txInfo, ex);
10        throw ex;
11    }
12    finally {
13        cleanupTransactionInfo(txInfo);
14    }
15    //省略非关键代码
16 }
```



在 `completeTransactionAfterThrowing` 的代码中，有这样一个方法 `rollbackOn()`，这是事务的回滚的关键判断条件。当这个条件满足时，会触发 `rollback` 操作，事务回滚。

[复制代码](#)

```
1 protected void completeTransactionAfterThrowing(@Nullable TransactionInfo txIn
2     //省略非关键代码
3     //判断是否需要回滚
4     if (txInfo.transactionAttribute != null && txInfo.transactionAttribute.rol
5         try {
6             //执行回滚
7 txInfo.getTransactionManager().rollback(txInfo.getTransactionStatus());
8         }
9         catch (TransactionSystemException ex2) {
10             ex2.initApplicationException(ex);
11             throw ex2;
12         }
13         catch (RuntimeException | Error ex2) {
14             throw ex2;
15         }
16     }
17     //省略非关键代码
18 }
```

`rollbackOn()` 其实包括了两个层级，具体可参考如下代码：

[复制代码](#)

```
1 public boolean rollbackOn(Throwable ex) {
2     // 层级 1：根据"rollbackRules"及当前捕获异常来判断是否需要回滚
3     RollbackRuleAttribute winner = null;
4     int deepest = Integer.MAX_VALUE;
5     if (this.rollbackRules != null) {
6         for (RollbackRuleAttribute rule : this.rollbackRules) {
7             // 当前捕获的异常可能是回滚“异常”的继承体系中的“一员”
8             int depth = rule.getDepth(ex);
9             if (depth >= 0 && depth < deepest) {
10                 deepest = depth;
11                 winner = rule;
12             }
13         }
14     }
15     // 层级 2：调用父类的 rollbackOn 方法来决策是否需要 rollback
16     if (winner == null) {
17         return super.rollbackOn(ex);
18     }
19     return !(winner instanceof NoRollbackRuleAttribute);
20 }
```

## 1. RuleBasedTransactionAttribute 自身的 rollbackOn()

当我们在 @Transactional 中配置了 rollbackFor，这个方法就会用捕获到的异常和 rollbackFor 中配置的异常做比较。如果捕获到的异常是 rollbackFor 配置的异常或其子类，就会直接 rollback。在我们的案例中，由于在事务的注解中没有加任何规则，所以这段逻辑处理其实找不到规则（即 winner == null），进而走到下一步。

## 2. RuleBasedTransactionAttribute 父类 DefaultTransactionAttribute 的 rollbackOn()

如果没有在 @Transactional 中配置 rollback 属性，或是捕获到的异常和所配置异常的类型不一致，就会继续调用父类的 rollbackOn() 进行处理。

而在父类的 rollbackOn() 中，我们发现了一个重要的线索，只有在异常类型为 RuntimeException 或者 Error 的时候才会返回 true，此时，会触发 completeTransactionAfterThrowing 方法中的 rollback 操作，事务被回滚。

[复制代码](#)

```
1 public boolean rollbackOn(Throwable ex) {  
2     return (ex instanceof RuntimeException || ex instanceof Error);  
3 }
```


查到这里，真相大白，Spring 处理事务的时候，如果没有在 @Transactional 中配置 rollback 属性，那么只有捕获到 RuntimeException 或者 Error 的时候才会触发回滚操作。而我们案例抛出的异常是 Exception，又没有指定与之匹配的回滚规则，所以我们不能触发回滚。

## 问题修正

从上述案例解析中，我们了解到，Spring 在处理事务过程中，并不会对 Exception 进行回滚，而会对 RuntimeException 或者 Error 进行回滚。

这么看来，修改方法也可以很简单，只需要把抛出的异常类型改成 RuntimeException 就可以了。于是这部分代码就可以修改如下：



 复制代码

```
1 @Service
2 public class StudentService {
3     @Autowired
4     private StudentMapper studentMapper;
5
6     @Transactional
7     public void saveStudent(String realname) throws Exception {
8         Student student = new Student();
9         student.setRealname(realname);
10        studentMapper.saveStudent(student);
11        if (student.getRealname().equals("小明")) {
12            throw new RuntimeException("该用户已存在");
13        }
14    }
15 }
```

再执行一下，这时候异常会正常抛出，数据库里不会有新数据产生，表示这时候 Spring 已经对这个异常进行了处理，并将事务回滚。

但是很明显，这种修改方法看起来不够优美，毕竟我们的异常有时候是固定死不能随意修改的。所以结合前面的案例分析，我们还有一个更好的修改方式。

具体而言，我们在解析 RuleBasedTransactionAttribute.rollbackOn 的代码时提到过 rollbackFor 属性的处理规则。也就是我们在 @Transactional 的 rollbackFor 加入需要支持的异常类型（在这里是 Exception）就可以匹配上我们抛出的异常，进而在异常抛出时进行回滚。

于是我们可以完善下案例中的注解，修改后代码如下：

 复制代码


```
1 @Transactional(rollbackFor = Exception.class)
```

再次测试运行，你会发现一切符合预期了。

## 案例 2：试图给 private 方法添加事务

接着上一个案例，我们已经实现了保存学生信息的功能。接下来，我们来优化一下逻辑，让学生的创建和保存逻辑分离，于是我就对代码做了一些重构，把 Student 的实例创建和

保存逻辑拆到两个方法中分别进行。然后，把事务的注解 `@Transactional` 加在了保存数据库的方法上。

 复制代码

```
1 @Service
2 public class StudentService {
3     @Autowired
4     private StudentMapper studentMapper;
5
6     @Autowired
7     private StudentService studentService;
8
9     public void saveStudent(String realname) throws Exception {
10         Student student = new Student();
11         student.setRealname(realname);
12         studentService.doSaveStudent(student);
13     }
14
15     @Transactional
16     private void doSaveStudent(Student student) throws Exception {
17         studentMapper.saveStudent(student);
18         if (student.getRealname().equals("小明")) {
19             throw new RuntimeException("该用户已存在");
20         }
21     }
22 }
```

执行的时候，继续传入参数“小明”，看看执行结果是什么样子？

异常正常抛出，事务却没有回滚。明明是在方法上加上了事务的注解啊，为什么没有生效呢？我们还是从 Spring 源码中找答案。

## 案例解析

通过 debug，我们一步步寻找到了问题的根源，得到了以下调用栈。我们通过 Spring 的源码来解析一下完整的过程。

```

✓ "main"@1 in group "main": RUNNING
allowPublicMethodsOnly:188, AnnotationTransactionAttributeSource (org.springframework.transaction.annotation)
computeTransactionAttribute:167, AbstractFallbackTransactionAttributeSource (org.springframework.transaction.interceptor)
getTransactionAttribute:124, AbstractFallbackTransactionAttributeSource (org.springframework.transaction.interceptor)
matches:47, TransactionAttributeSourcePointcut (org.springframework.transaction.interceptor)
canApply:252, AopUtils (org.springframework.aop.support)
canApply:289, AopUtils (org.springframework.aop.support)
findAdvisorsThatCanApply:321, AopUtils (org.springframework.aop.support)
findAdvisorsThatCanApply:128, AbstractAdvisorAutoProxyCreator (org.springframework.aop.framework.autoproxy)
findEligibleAdvisors:97, AbstractAdvisorAutoProxyCreator (org.springframework.aop.framework.autoproxy)
getAdvicesAndAdvisorsForBean:78, AbstractAdvisorAutoProxyCreator (org.springframework.aop.framework.autoproxy)
wrapIfNecessary:337, AbstractAutoProxyCreator (org.springframework.aop.framework.autoproxy)
getEarlyBeanReference:240, AbstractAutoProxyCreator (org.springframework.aop.framework.autoproxy)
getEarlyBeanReference:967, AbstractAutowireCapableBeanFactory (org.springframework.beans.factory.support)
lambda$doCreateBean$1:595, AbstractAutowireCapableBeanFactory (org.springframework.beans.factory.support)
getObject:-1, 1128948651 (org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory$$Lambda$44)
getSingleton:194, DefaultSingletonBeanRegistry (org.springframework.beans.factory.support)
getSingleton:168, DefaultSingletonBeanRegistry (org.springframework.beans.factory.support)
doGetBean:256, AbstractBeanFactory (org.springframework.beans.factory.support)
getBean:208, AbstractBeanFactory (org.springframework.beans.factory.support)
resolveCandidate:276, DependencyDescriptor (org.springframework.beans.factory.config)
doResolveDependency:1380, DefaultListableBeanFactory (org.springframework.beans.factory.support)
resolveFieldValue:657, AutowiredAnnotationBeanPostProcessor$AutowiredFieldElement (org.springframework.beans.factory.annotation)
inject:640, AutowiredAnnotationBeanPostProcessor$AutowiredFieldElement (org.springframework.beans.factory.annotation)
inject:119, InjectionMetadata (org.springframework.beans.factory.annotation)
postProcessProperties:399, AutowiredAnnotationBeanPostProcessor (org.springframework.beans.factory.annotation)
populateBean:1413, AbstractAutowireCapableBeanFactory (org.springframework.beans.factory.support)
doCreateBean:601, AbstractAutowireCapableBeanFactory (org.springframework.beans.factory.support)
createBean:524, AbstractAutowireCapableBeanFactory (org.springframework.beans.factory.support)
lambda$doGetBean$0:335, AbstractBeanFactory (org.springframework.beans.factory.support)
getObject:-1, 1478150312 (org.springframework.beans.factory.support.AbstractBeanFactory$$Lambda$43)

```

前一段是 Spring 创建 Bean 的过程。当 Bean 初始化之后，开始尝试代理操作，这个过程是从 AbstractAutoProxyCreator 里的 postProcessAfterInitialization 方法开始处理的：

[复制代码](#)

```

1 public Object postProcessAfterInitialization(@Nullable Object bean, String beanName) {
2     if (bean != null) {
3         Object cacheKey = getCacheKey(bean.getClass(), beanName);
4         if (this.earlyProxyReferences.remove(cacheKey) != bean) {
5             return wrapIfNecessary(bean, beanName, cacheKey);
6         }
7     }
8     return bean;
9 }

```

我们一路往下找，暂且略过那些非关键要素的代码，直到到了 AopUtils 的 canApply 方法。这个方法就是针对切面定义里的条件，确定这个方法是否可以被应用创建成代理。其中有一段 methodMatcher.matches(method, targetClass) 是用来判断这个方法是否符合这样的条件：

[复制代码](#)

```

1 public static boolean canApply(Pointcut pc, Class<?> targetClass, boolean hasInterface) {

```

```
2 //省略非关键代码
3 for (Class<?> clazz : classes) {
4     Method[] methods = ReflectionUtils.getAllDeclaredMethods(clazz);
5     for (Method method : methods) {
6         if (introductionAwareMethodMatcher != null ?
7             introductionAwareMethodMatcher.matches(method, targetClass, has
8             methodMatcher.matches(method, targetClass)) {
9             return true;
10        }
11    }
12 }
13 return false;
14 }
```

从 matches() 调用到了 AbstractFallbackTransactionAttributeSource 的 getTransactionAttribute :

[复制代码](#)

```
1 public boolean matches(Method method, Class<?> targetClass) {
2     //省略非关键代码
3     TransactionAttribute tas = getTransactionAttributeSource();
4     return (tas == null || tas.getTransactionAttribute(method, targetClass) !=
5 }
```

其中，getTransactionAttribute 这个方法是用来获取注解中的事务属性，根据属性确定事务采用什么样的策略。

[复制代码](#)

```
1 public TransactionAttribute getTransactionAttribute(Method method, @Nullable C
2     //省略非关键代码
3     TransactionAttribute txAttr = computeTransactionAttribute(method, target
4     //省略非关键代码
5 }
6 }
```

接着调用到 computeTransactionAttribute 这个方法，其主要功能是根据方法和类的类型确定是否返回事务属性，执行代码如下：

[复制代码](#)

```
1 protected TransactionAttribute computeTransactionAttribute(Method method, @Nul
2     //省略非关键代码
```

```
3     if (allowPublicMethodsOnly() && !Modifier.isPublic(method.getModifiers()))
4         return null;
5     }
6     //省略非关键代码
7 }
```

这里有这样一个判断 `allowPublicMethodsOnly() && !Modifier.isPublic(method.getModifiers())`，当这个判断结果为 `true` 的时候返回 `null`，也就意味着这个方法不会被代理，从而导致事务的注解不会生效。那此处的判断值到底是不是 `true` 呢？我们可以分别看一下。

### 条件 1 : `allowPublicMethodsOnly()`

`allowPublicMethodsOnly` 返回了 `AnnotationTransactionAttributeSource` 的 `publicMethodsOnly` 属性。

```
1 protected boolean allowPublicMethodsOnly() {
2     return this.publicMethodsOnly;
3 }
```

[复制代码](#)


而这个 `publicMethodsOnly` 属性是通过 `AnnotationTransactionAttributeSource` 的构造方法初始化的，默认为 `true`。

```
1 public AnnotationTransactionAttributeSource() {
2     this(true);
3 }
```

[复制代码](#)

### 条件 2 : `Modifier.isPublic()`

这个方法根据传入的 `method.getModifiers()` 获取方法的修饰符。该修饰符是 `java.lang.reflect.Modifier` 的静态属性，对应的几类修饰符分别是：`PUBLIC: 1`，`PRIVATE: 2`，`PROTECTED: 4`。这里面做了一个位运算，只有当传入的方法修饰符是 `public` 类型的时候，才返回 `true`。

 复制代码

```
1 public static boolean isPublic(int mod) {
2     return (mod & PUBLIC) != 0;
3 }
```

综合上述两个条件，你会发现，只有当注解为事务的方法被声明为 public 的时候，才会被 Spring 处理。

## 问题修正

了解了问题的根源以后，解决它就变得很简单了，我们只需要把它的修饰符从 private 改成 public 就可以了。

不过需要额外补充的是，我们调用这个加了事务注解的方法，必须是调用被 Spring AOP 代理过的方法，也就是不能通过类的内部调用或者通过 this 的方式调用。所以我们的案例的 StudentService，它含有一个自动装配（Autowired）了自身（StudentService）的实例来完成代理方法的调用。这个问题我们在之前 Spring AOP 的代码解析中重点强调过，此处就不再详述了。

 复制代码

```
1 @Service
2 public class StudentService {
3     @Autowired
4     private StudentMapper studentMapper;
5
6     @Autowired
7     private StudentService studentService;
8
9     public void saveStudent(String realname) throws Exception {
10         Student student = new Student();
11         student.setRealname(realname);
12         studentService.doSaveStudent(student);
13     }
14
15     @Transactional
16     public void doSaveStudent(Student student) throws Exception {
17         studentMapper.saveStudent(student);
18         if (student.getRealname().equals("小明")) {
19             throw new RuntimeException("该学生已存在");
20         }
21     }
22 }
```



重新运行一下，异常正常抛出，数据库也没有新数据产生，事务生效了，问题解决。

[复制代码](#)

```
1 Exception in thread "main" java.lang.RuntimeException: 该学生已存在
2   at com.spring.puzzle.others.transaction.example2.StudentService.doSaveStuden
3
```

## 重点回顾

通过以上两个案例，相信你对 Spring 的声明式事务机制已经有了进一步的了解，最后总结下重点：

Spring 支持声明式事务机制，它通过在方法上加上 `@Transactional`，表明该方法需要事务支持。于是，在加载的时候，根据 `@Transactional` 中的属性，决定对该事务采取什么样的策略；

`@Transactional` 对 `private` 方法不生效，所以我们应该把需要支持事务的方法声明为 `public` 类型；

Spring 处理事务的时候，默认只对 `RuntimeException` 和 `Error` 回滚，不会对 `Exception` 回滚，如果有特殊需要，需要额外声明，例如指明 `Transactional` 的属性 `rollbackFor` 为 `Exception.class`。

## 思考题

`RuntimeException` 是 `Exception` 的子类，如果用 `rollbackFor=Exception.class`，那对 `RuntimeException` 也会生效。如果我们需要对 `Exception` 执行回滚操作，但对于 `RuntimeException` 不执行回滚操作，应该怎么做呢？

期待你的思考，我们留言区见！

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

 赞 2     提建议

© 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 18 | Spring Data 常见错误

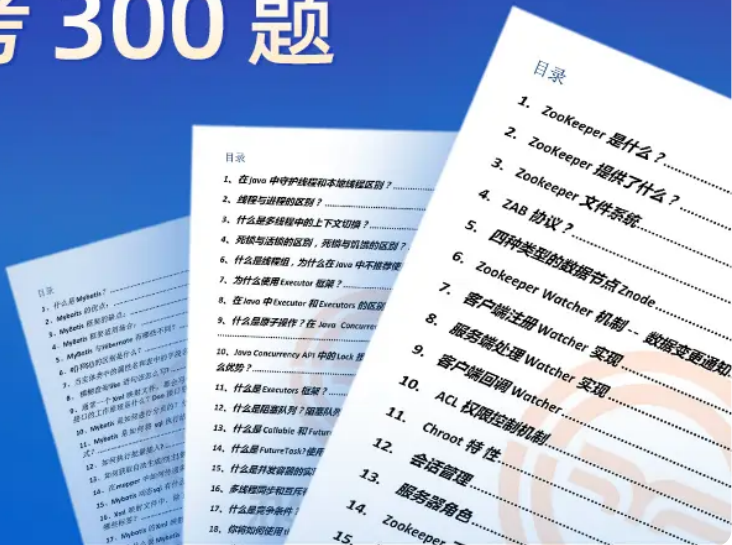
下一篇 20 | Spring 事务常见错误 (下)

更多学习推荐

# Java 面试必考 300 题

最新汇总

限时免费领取



## 精选留言 (6)

写留言



龙猫

2021-06-04

@Transactional(rollbackFor = Exception.class, noRollbackFor = RuntimeException.class)



Monday

2021-07-19

案例2中，进入doSaveStudent()方法后，抛NPE ( studentMapper==null 为true )



陈汤姆

2021-06-15

学到了，一直以为注解只能对public生效是因为动态代理的原因！



**qchang**

2021-06-04

思考题：一种是try-catch判断异常类型后，非RuntimeException抛出；另一种可以采用注解@Transactional(noRollbackFor = RuntimeException.class)

展开 ▾

**LkS**

2021-06-04

可以使用noRollbackFor

@Transactional(rollbackFor = Exception.class,noRollbackFor = RuntimeException.class)

展开 ▾

**手撕嘴啃Spring**

2021-06-04

@Transactional(rollbackFor = Exception.class, noRollbackFor = RuntimeException.class)

