



下载APP



## 20 | Spring 事务常见错误（下）

2021-06-07 傅健

《Spring编程常见错误50例》

[课程介绍 >](#)**讲述：傅健**

时长 15:21 大小 14.07M



你好，我是傅健。

通过上一节课的学习，我们了解了 Spring 事务的原理，并解决了几个常见的问题。这节课我们将继续讨论事务中的另外两个问题，一个是关于事务的传播机制，另一个是关于多数数据源的切换问题，通过这两个问题，你可以更加深入地了解 Spring 事务的核心机制。

### 案例 1：嵌套事务回滚错误

上一节课我们完成了学生注册功能，假设我们需要对这个功能继续进行扩展，当学生注册完成后，需要给这个学生登记一门英语必修课，并更新这门课的登记学生数。为此，我增加了两个表。



## 1. 课程表 course , 记录课程名称和注册的学生数。

[复制代码](#)

```
1 CREATE TABLE `course` (  
2   `id` int(11) NOT NULL AUTO_INCREMENT,  
3   `course_name` varchar(64) DEFAULT NULL,  
4   `number` int(11) DEFAULT NULL,  
5   PRIMARY KEY (`id`)  
6 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

## 2. 学生选课表 student\_course , 记录学生表 student 和课程表 course 之间的多对多关联。

[复制代码](#)

```
1 CREATE TABLE `student_course` (  
2   `student_id` int(11) NOT NULL,  
3   `course_id` int(11) NOT NULL  
4 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

同时我为课程表初始化了一条课程信息 , id = 1 , course\_name = "英语" , number = 0。

接下来我们完成用户的相关操作 , 主要包括两部分。

### 1. 新增学生选课记录

[复制代码](#)

```
1 @Mapper  
2 public interface StudentCourseMapper {  
3     @Insert("INSERT INTO `student_course`(`student_id`, `course_id`) VALUES (#  
4     void saveStudentCourse(@Param("studentId") Integer studentId, @Param("cour  
5 }
```


### 2. 课程登记学生数 + 1

[复制代码](#)

```
1 @Mapper
```


```
2 public interface CourseMapper {
3     @Update("update `course` set number = number + 1 where id = #{id}")
4     void addCourseNumber(int courseId);
5 }
```

我们增加了一个新的业务类 `CourseService`，用于实现相关业务逻辑。分别调用了上述两个方法来保存学生与课程的关联关系，并给课程注册人数 +1。最后，别忘了给这个方法加上事务注解。

 复制代码

```
1 @Service
2 public class CourseService {
3     @Autowired
4     private CourseMapper courseMapper;
5
6     @Autowired
7     private StudentCourseMapper studentCourseMapper;
8
9     //注意这个方法标记了“Transactional”
10    @Transactional(rollbackFor = Exception.class)
11    public void regCourse(int studentId) throws Exception {
12        studentCourseMapper.saveStudentCourse(studentId, 1);
13        courseMapper.addCourseNumber(1);
14    }
15 }
16
```

我们在之前的 `StudentService.saveStudent()` 中调用了 `regCourse()`，实现了完整的业务逻辑。为了避免注册课程的业务异常导致学生信息无法保存，在这里 `catch` 了注册课程方法中抛出的异常。我们希望的结果是，当注册课程发生错误时，只回滚注册课程部分，保证学生信息依然正常。


 复制代码

```
1 @Service
2 public class StudentService {
3     //省略非关键代码
4     @Transactional(rollbackFor = Exception.class)
5     public void saveStudent(String realname) throws Exception {
6         Student student = new Student();
7         student.setRealname(realname);
8         studentService.doSaveStudent(student);
9         try {
10             courseService.regCourse(student.getId());

```


```
11         } catch (Exception e) {
12             e.printStackTrace();
13         }
14     }
15     //省略非关键代码
16 }
```

为了验证异常是否符合预期，我们在 `regCourse()` 里抛出了一个注册失败的异常：

 复制代码

```
1 @Transactional(rollbackFor = Exception.class)
2 public void regCourse(int studentId) throws Exception {
3     studentCourseMapper.saveStudentCourse(studentId, 1);
4     courseMapper.addCourseNumber(1);
5     throw new Exception("注册失败");
6 }
```

运行一下这段代码，在控制台里我们看到了以下提示信息：

 复制代码

```
1 java.lang.Exception: 注册失败
2   at com.spring.puzzle.others.transaction.example3.CourseService.regCourse(Cou
3 //.....省略非关键代码.....
4 Exception in thread "main" org.springframework.transaction.UnexpectedRollbackE
5   at org.springframework.transaction.support.AbstractPlatformTransactionManage
6   at org.springframework.transaction.support.AbstractPlatformTransactionManage
7   at org.springframework.transaction.interceptor.TransactionAspectSupport.comm
8   at org.springframework.transaction.interceptor.TransactionAspectSupport.invo
9   at org.springframework.transaction.interceptor.TransactionInterceptor.invoke
10  at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(Refl
11  at org.springframework.aop.framework.CglibAopProxy$DynamicAdvisedInterceptor
12  at com.spring.puzzle.others.transaction.example3.StudentService$$EnhancerByS
13  at com.spring.puzzle.others.transaction.example3.AppConfig.main(AppConfig.ja
```

其中，注册失败部分的异常符合预期，但是后面又多了一个这样的错误提示：Transaction rolled back because it has been marked as rollback-only。

最后的结果是，学生和选课的信息都被回滚了，显然这并不符合我们的预期。我们期待的结果是即便内部事务 `regCourse()` 发生异常，外部事务 `saveStudent()` 俘获该异常后，内部事务应自行回滚，不影响外部事务。那么这是是什么原因造成的呢？我们需要研究一下 Spring 的源码，来找找答案。

## 案例解析

在做进一步的解析之前，我们可以先通过伪代码把整个事务的结构梳理一下：

[复制代码](#)

```
1  // 外层事务
2  @Transactional(rollbackFor = Exception.class)
3  public void saveStudent(String realname) throws Exception {
4      //.....省略逻辑代码.....
5      studentService.doSaveStudent(student);
6      try {
7          // 嵌套的内层事务
8          @Transactional(rollbackFor = Exception.class)
9          public void regCourse(int studentId) throws Exception {
10             //.....省略逻辑代码.....
11         }
12     } catch (Exception e) {
13         e.printStackTrace();
14     }
15 }
```

可以看出来，整个业务是包含了 2 层事务，外层的 `saveStudent()` 的事务和内层的 `regCourse()` 事务。

在 Spring 声明式的事务处理中，有一个属性 `propagation`，表示打算对这些方法怎么使用事务，即一个带事务的方法调用了另一个带事务的方法，被调用的方法它怎么处理自己事务和调用方法事务之间的关系。

其中 `propagation` 有 7 种配置：`REQUIRED`、`SUPPORTS`、`MANDATORY`、`REQUIRES_NEW`、`NOT_SUPPORTED`、`NEVER`、`NESTED`。默认是 `REQUIRED`，它的含义是：如果本来有事务，则加入该事务，如果没有事务，则创建新的事务。

结合我们的伪代码示例，因为在 `saveStudent()` 上声明了一个外部的事务，就已经存在一个事务了，在 `propagation` 值为默认的 `REQUIRED` 的情况下，`regCourse()` 就会加入到已有的事务中，两个方法共用一个事务。

我们再来看下 Spring 事务处理的核心，其关键实现参考 `TransactionAspectSupport.invokeWithinTransaction()`：

```
1  protected Object invokeWithinTransaction(Method method, @Nullable Class<?> targetClass,
2      final InvocationCallback invocation) throws Throwable {
3
4      TransactionAttributeSource tas = getTransactionAttributeSource();
5      final TransactionAttribute txAttr = (tas != null ? tas.getTransactionAttribute(
6      final PlatformTransactionManager tm = determineTransactionManager(txAttr);
7      final String joinpointIdentification = methodIdentification(method, targetClass);
8      if (txAttr == null || !(tm instanceof CallbackPreferringPlatformTransactionManager))
9          // 是否需要创建一个事务
10         TransactionInfo txInfo = createTransactionIfNecessary(tm, txAttr, joinpointIdentification);
11         Object retVal = null;
12         try {
13             // 调用具体的业务方法
14             retVal = invocation.proceedWithInvocation();
15         }
16         catch (Throwable ex) {
17             // 当发生异常时进行处理
18             completeTransactionAfterThrowing(txInfo, ex);
19             throw ex;
20         }
21         finally {
22             cleanupTransactionInfo(txInfo);
23         }
24         // 正常返回时提交事务
25         commitTransactionAfterReturning(txInfo);
26         return retVal;
27     }
28     //.....省略非关键代码.....
29 }
```

整个方法完成了事务的一整套处理逻辑，如下：


1. 检查是否需要创建事务；
2. 调用具体的业务方法进行处理；
3. 提交事务；
4. 处理异常。

这里要格外注意的是，当前案例是两个事务嵌套的场景，外层事务 `doSaveStudent()` 和内层事务 `regCourse()`，每个事务都会调用到这个方法。所以，这个方法会被调用两次。下面我们来具体来看下内层事务对异常的处理。




当捕获了异常，会调用

TransactionAspectSupport.completeTransactionAfterThrowing() 进行异常处理：

 复制代码


```
1  protected void completeTransactionAfterThrowing(@Nullable TransactionInfo txIn
2      if (txInfo != null && txInfo.getTransactionStatus() != null) {
3          if (txInfo.transactionAttribute != null && txInfo.transactionAttribute.r
4              try {
5                  txInfo.getTransactionManager().rollback(txInfo.getTransactionStatu
6              }
7              catch (TransactionSystemException ex2) {
8                  logger.error("Application exception overridden by rollback excepti
9                  ex2.initApplicationException(ex);
10                 throw ex2;
11             }
12             catch (RuntimeException | Error ex2) {
13                 logger.error("Application exception overridden by rollback excepti
14                 throw ex2;
15             }
16         }
17         //.....省略非关键代码.....
18     }
19 }
```

在这个方法里，我们对异常类型做了一些检查，当符合声明中的定义后，执行了具体的 rollback 操作，这个操作是通过 TransactionManager.rollback() 完成的：

 复制代码

```
1  public final void rollback(TransactionStatus status) throws TransactionExcepti
2      if (status.isCompleted()) {
3          throw new IllegalTransactionStateException(
4              "Transaction is already completed - do not call commit or rollback
5      }
6
7      DefaultTransactionStatus defStatus = (DefaultTransactionStatus) status;
8      processRollback(defStatus, false);
9  }
```

而 rollback() 是在 AbstractPlatformTransactionManager 中实现的，继续调用了 processRollback()：

 复制代码

```
1 private void processRollback(DefaultTransactionStatus status, boolean unexpected
2     try {
3         boolean unexpectedRollback = unexpected;
4
5         if (status.hasSavepoint()) {
6             // 有保存点
7             status.rollbackToHeldSavepoint();
8         }
9         else if (status.isNewTransaction()) {
10            // 是否为一个新的事务
11            doRollback(status);
12        }
13        else {
14            // 处于一个更大的事务中
15            if (status.hasTransaction()) {
16                // 分支1
17                if (status.isLocalRollbackOnly() || isGlobalRollbackOnParticipation
18                    doSetRollbackOnly(status);
19            }
20        }
21        if (!isFailEarlyOnGlobalRollbackOnly()) {
22            unexpectedRollback = false;
23        }
24    }
25
26    // 省略非关键代码
27    if (unexpectedRollback) {
28        throw new UnexpectedRollbackException(
29            "Transaction rolled back because it has been marked as rollback
30    }
31 }
32 finally {
33     cleanupAfterCompletion(status);
34 }
35 }
```

这个方法里区分了三种不同类型的情况：

1. 是否有保存点；
2. 是否为一个新的事务；
3. 是否处于一个更大的事务中。

在这里，因为我们用的是默认的传播类型 REQUIRED，嵌套的事务并没有开启一个新的事务，所以在这种情况下，当前事务是处于一个更大的事务中，所以会走到情况 3 分支 1 的代码块下。



这里有两个判断条件来确定是否设置为仅回滚：

```
if (status.isLocalRollbackOnly() || isGlobalRollbackOnParticipationFailure())
```

满足任何一个，都会执行 `doSetRollbackOnly()` 操作。`isLocalRollbackOnly` 在当前的情况下是 `false`，所以是否分设置为仅回滚就由 `isGlobalRollbackOnParticipationFailure()` 这个方法来决定，其默认值为 `true`，即是否回滚交由外层事务统一决定。

显然这里的条件得到了满足，从而执行 `doSetRollbackOnly`：

[复制代码](#)

```
1 protected void doSetRollbackOnly(DefaultTransactionStatus status) {
2     DataSourceTransactionObject txObject = (DataSourceTransactionObject) status
3     txObject.setRollbackOnly();
4 }
```

以及最终调用到的 `DataSourceTransactionObject` 中的 `setRollbackOnly()`：

[复制代码](#)

```
1 public void setRollbackOnly() {
2     getConnectionHolder().setRollbackOnly();
3 }
```

到这一步，内层事务的操作基本执行完毕，它处理了异常，并最终调用到了 `DataSourceTransactionObject` 中的 `setRollbackOnly()`。

接下来，我们来看外层事务。因为在外层事务中，我们自己的代码捕获了内层抛出来的异常，所以这个异常不会继续往上抛，最后的事务会在 `TransactionAspectSupport.invokeWithinTransaction()` 中的 `commitTransactionAfterReturning()` 中进行处理：

[复制代码](#)

```
1 protected void commitTransactionAfterReturning(@Nullable TransactionInfo txInf
2     if (txInfo != null && txInfo.getTransactionStatus() != null) {      txInfo.g
3     }
4 }
```

在这个方法里我们执行了 commit 操作，代码如下：

[复制代码](#)

```
1 public final void commit(TransactionStatus status) throws TransactionException
2     //.....省略非关键代码.....
3     if (!shouldCommitOnGlobalRollbackOnly() && defStatus.isGlobalRollbackOnly())
4         processRollback(defStatus, true);
5         return;
6     }
7
8     processCommit(defStatus);
9 }
```

在 `AbstractPlatformTransactionManager.commit()` 中，当满足了 `shouldCommitOnGlobalRollbackOnly()` 和 `defStatus.isGlobalRollbackOnly()`，就会回滚，否则会继续提交事务。其中 `shouldCommitOnGlobalRollbackOnly()` 的作用为，如果发现了事务被标记了全局回滚，并且在发生了全局回滚的情况下，判断是否应该提交事务，这个方法的默认实现是返回了 `false`，这里我们不需要关注它，继续查看 `isGlobalRollbackOnly()` 的实现：

[复制代码](#)


```
1 public boolean isGlobalRollbackOnly() {
2     return ((this.transaction instanceof SmartTransactionObject) &&
3         ((SmartTransactionObject) this.transaction).isRollbackOnly());
4 }
```

这个方法最终进入了 `DataSourceTransactionObject` 类中的 `isRollbackOnly()`：

[复制代码](#)

```
1 public boolean isRollbackOnly() {
2     return getConnectionHolder().isRollbackOnly();
3 }
```

现在让我们再次回顾一下之前的内部事务处理结果，其最终调用到的是 `DataSourceTransactionObject` 中的 `setRollbackOnly()`：

 复制代码

```
1 public void setRollbackOnly() {  
2     getConnectionHolder().setRollbackOnly();  
3 }
```

isRollbackOnly() 和 setRollbackOnly() 这两个方法的执行本质都是对 ConnectionHolder 中 rollbackOnly 属性标志位的存取，而 ConnectionHolder 则存在于 DefaultTransactionStatus 类实例的 transaction 属性之中。

至此，答案基本浮出水面了，我们把整个逻辑串在一起就是：外层事务是否回滚的关键，最终取决于 **DataSourceTransactionObject 类中的 isRollbackOnly()**，而该方法的返回值，正是我们在内层异常的时候设置的。


所以最终外层事务也被回滚了，从而在控制台中打印出异常信息："Transaction rolled back because it has been marked as rollback-only"。

所以到这里，问题也就清楚了，Spring 默认的事务传播属性为 REQUIRED，如我们之前介绍的，它的含义是：如果本来有事务，则加入该事务，如果没有事务，则创建新的事务，因而内外两层事务都处于同一个事务中。所以，当我们在 regCourse() 中抛出异常，并触发了回滚操作时，这个回滚会进一步传播，从而把 saveStudent() 也回滚了。最终导致整个事务都被回滚了。

## 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，有个默认的传播属性 REQUIRED，在整个事务的调用链上，任何一个环节抛出的异常都会导致全局回滚。

知道了这个结论，修改方法也就很简单了，我们只需要对传播属性进行修改，把类型改成 REQUIRES\_NEW 就可以了。于是这部分代码就修改成这样：

 复制代码

```
1 @Transactional(rollbackFor = Exception.class, propagation = Propagation.REQUIR  
2 public void regCourse(int studentId) throws Exception {  
3     studentCourseMapper.saveStudentCourse(studentId, 1);  
4     courseMapper.addCourseNumber(1);  
5     throw new Exception("注册失败");  
6 }
```

运行一下看看：

[复制代码](#)

```
1 java.lang.Exception: 注册失败
2     at com.spring.puzzle.others.transaction.example3.CourseService.regCourse(Cou
```

异常正常抛出，注册课程部分的数据没有保存，但是学生还是正常注册成功。这意味着此时 Spring 只对注册课程这部分的数据进行了回滚，并没有传播到上一级。

这里我简单解释下这个过程：

当子事务声明为 `Propagation.REQUIRES_NEW` 时，在 `TransactionAspectSupport.invokeWithinTransaction()` 中调用 `createTransactionIfNecessary()` 就会创建一个新的事务，独立于外层事务。

而在 `AbstractPlatformTransactionManager.processRollback()` 进行 `rollback` 处理时，因为 `status.isNewTransaction()` 会因为它处于一个新的事务中而返回 `true`，所以它走入到了另一个分支，执行了 `doRollback()` 操作，让这个子事务单独回滚，不会影响到主事务。

至此，这个问题得到了很好的解决。

## 案例 2：多数据源间切换之谜

在前面的案例中，我们完成了学生注册功能和课程注册功能。假设新需求又来了，每个学生注册的时候，需要给他们发一张校园卡，并给校园卡里充入 50 元钱。但是这个校园卡管理系统是一个第三方系统，使用的是另一套数据库，这样我们就需要在一个事务中同时操作两个数据库。

第三方的 Card 表如下：

[复制代码](#)

```
1 CREATE TABLE `card` (
2   `id` int(11) NOT NULL AUTO_INCREMENT,
3   `student_id` int(11) DEFAULT NULL,
```

```
4  `balance` int(11) DEFAULT NULL,  
5  PRIMARY KEY (`id`)  
6  \ ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

对应的 Card 对象如下：

[复制代码](#)

```
1 public class Card {  
2     private Integer id;  
3     private Integer studentId;  
4     private Integer balance;  
5     //省略 Get/Set 方法  
6 }
```

对应的 Mapper 接口如下，里面包含了一个 saveCard 的 insert 语句，用于创建一条校园卡记录：

[复制代码](#)

```
1 @Mapper  
2 public interface CardMapper {  
3     @Insert("INSERT INTO `card`(`student_id`, `balance`) VALUES (#{studentId},  
4     @Options(useGeneratedKeys = true, keyProperty = \"id\")  
5     int saveCard(Card card);  
6 }
```

Card 的业务类如下，里面实现了卡与学生 ID 关联，以及充入 50 元的操作：

[复制代码](#)

```
1 @Service  
2 public class CardService {  
3     @Autowired  
4     private CardMapper cardMapper;  
5  
6     @Transactional  
7     public void createCard(int studentId) throws Exception {  
8         Card card = new Card();  
9         card.setStudentId(studentId);  
10        card.setBalance(50);  
11        cardMapper.saveCard(card);  
12    }  
13 }
```

## 案例解析

这是一个相对常见的需求，学生注册和发卡都要在一个事务里完成，但是我们都默认只会连一个数据源，之前我们一直连的都是学生信息这个数据源，在这里，我们还需要对校园卡的数据源进行操作。于是，我们需要在一个事务里完成对两个数据源的操作，该如何实现这样的功能呢？


我们继续从 Spring 的源码中寻找答案。在 Spring 里有这样一个抽象类 `AbstractRoutingDataSource`，这个类相当于 `DataSource` 的路由中介，在运行时根据某种 `key` 值来动态切换到所需的 `DataSource` 上。通过实现这个类就可以实现我们期望的动态数据源切换。

这里强调一下，这个类里有这么几个关键属性：

`targetDataSources` 保存了 `key` 和数据库连接的映射关系；

`defaultTargetDataSource` 标识默认的连接；

`resolvedDataSources` 存储数据库标识和数据源的映射关系。

 复制代码

```
1 public abstract class AbstractRoutingDataSource extends AbstractDataSource implements
2
3     @Nullable
4     private Map<Object, Object> targetDataSources;
5
6     @Nullable
7     private Object defaultTargetDataSource;
8
9     private boolean lenientFallback = true;
10
11     private DataSourceLookup dataSourceLookup = new JndiDataSourceLookup();
12
13     @Nullable
14     private Map<Object, DataSource> resolvedDataSources;
15
16     @Nullable
17     private DataSource resolvedDefaultDataSource;
18
19     //省略非关键代码
20 }
```



AbstractRoutingDataSource 实现了 InitializingBean 接口，并覆写了 afterPropertiesSet()。该方法会在初始化 Bean 的时候执行，将多个 DataSource 初始化到 resolvedDataSources。这里的 targetDataSources 属性存储了将要切换的多数据源 Bean 信息。

[复制代码](#)

```
1 @Override
2 public void afterPropertiesSet() {
3     if (this.targetDataSources == null) {
4         throw new IllegalArgumentException("Property 'targetDataSources' is requ
5     }
6     this.resolvedDataSources = new HashMap<>(this.targetDataSources.size());
7     this.targetDataSources.forEach((key, value) -> {
8         Object lookupKey = resolveSpecifiedLookupKey(key);
9         DataSource dataSource = resolveSpecifiedDataSource(value);
10        this.resolvedDataSources.put(lookupKey, dataSource);
11    });
12    if (this.defaultTargetDataSource != null) {
13        this.resolvedDefaultDataSource = resolveSpecifiedDataSource(this.default
14    }
15 }
```


获取数据库连接的是 getConnection()，它调用了 determineTargetDataSource() 来创建连接：

[复制代码](#)

```
1 @Override
2 public Connection getConnection() throws SQLException {
3     return determineTargetDataSource().getConnection();
4 }
5
6 @Override
7 public Connection getConnection(String username, String password) throws SQLEx
8     return determineTargetDataSource().getConnection(username, password);
9 }
```

determineTargetDataSource() 是整个部分的核心，它的作用就是动态切换数据源。有多少个数据源，就存多少个数据源在 targetDataSources 中。

targetDataSources 是一个 Map 类型的属性，key 表示每个数据源的名字，value 对应的是每个数据源 DataSource。

 复制代码

```
1 protected DataSource determineTargetDataSource() {
2     Assert.notNull(this.resolvedDataSources, "DataSource router not initialized");
3     Object lookupKey = determineCurrentLookupKey();
4     DataSource dataSource = this.resolvedDataSources.get(lookupKey);
5     if (dataSource == null && (this.lenientFallback || lookupKey == null)) {
6         dataSource = this.resolvedDefaultDataSource;
7     }
8     if (dataSource == null) {
9         throw new IllegalStateException("Cannot determine target DataSource for");
10    }
11    return dataSource;
12 }
```

而选择哪个数据源又是由 `determineCurrentLookupKey()` 来决定的，此方法是抽象方法，需要我们继承 `AbstractRoutingDataSource` 抽象类来重写此方法。该方法返回一个 key，该 key 是 Bean 中的 `beanName`，并赋值给 `lookupKey`，由此 key 可以通过 `resolvedDataSources` 属性的键来获取对应的 `DataSource` 值，从而达到数据源切换的效果。


 复制代码

```
1 protected abstract Object determineCurrentLookupKey();
```

这样看来，这个方法的实现就得由我们完成了。接下来我们将会完成一系列相关的代码，解决这个问题。

## 问题修正


首先，我们创建一个 `MyDataSource` 类，继承了 `AbstractRoutingDataSource`，并覆写了 `determineCurrentLookupKey()`：

 复制代码

```
1 public class MyDataSource extends AbstractRoutingDataSource {
2     private static final ThreadLocal<String> key = new ThreadLocal<String>();
3
4     @Override
5     protected Object determineCurrentLookupKey() {
6         return key.get();
7     }
8 }
```

```
9     public static void setDataSource(String dataSource) {
10         key.set(dataSource);
11     }
12
13     public static String getDataSource() {
14         return key.get();
15     }
16
17     public static void clearDataSource() {
18         key.remove();
19     }
20 }
21
```

其次，我们需要修改 JdbcConfig。这里我新写了一个 dataSource，将原来的 dataSource 改成 dataSourceCore，再将新定义的 dataSourceCore 和 dataSourceCard 放进一个 Map，对应的 key 分别是 core 和 card，并把 Map 赋值给 setTargetDataSources

 复制代码


```
1 public class JdbcConfig {
2     //省略非关键代码
3     @Value("${card.driver}")
4     private String cardDriver;
5
6     @Value("${card.url}")
7     private String cardUrl;
8
9     @Value("${card.username}")
10    private String cardUsername;
11
12    @Value("${card.password}")
13    private String cardPassword;
14
15    @Autowired
16    @Qualifier("dataSourceCard")
17    private DataSource dataSourceCard;
18
19    @Autowired
20    @Qualifier("dataSourceCore")
21    private DataSource dataSourceCore;
22
23    //省略非关键代码
24
25    @Bean(name = "dataSourceCore")
26    public DataSource createCoreDataSource() {
27        DriverManagerDataSource ds = new DriverManagerDataSource();
28        ds.setDriverClassName(driver);
29    }
30
31    @Bean(name = "dataSourceCard")
32    public DataSource createCardDataSource() {
33        DriverManagerDataSource ds = new DriverManagerDataSource();
34        ds.setDriverClassName(cardDriver);
35        ds.setUrl(cardUrl);
36        ds.setUsername(cardUsername);
37        ds.setPassword(cardPassword);
38    }
39
40    @PostConstruct
41    public void init() {
42        setTargetDataSources();
43    }
44
45    private void setTargetDataSources() {
46        Map<String, DataSource> targetDataSources = new HashMap<>();
47        targetDataSources.put("core", dataSourceCore);
48        targetDataSources.put("card", dataSourceCard);
49        this.setTargetDataSources(targetDataSources);
50    }
51}
```

```
29         ds.setUrl(url);
30         ds.setUsername(username);
31         ds.setPassword(password);
32         return ds;
33     }
34
35     @Bean(name = "dataSourceCard")
36     public DataSource createCardDataSource() {
37         DriverManagerDataSource ds = new DriverManagerDataSource();
38         ds.setDriverClassName(cardDriver);
39         ds.setUrl(cardUrl);
40         ds.setUsername(cardUsername);
41         ds.setPassword(cardPassword);
42         return ds;
43     }
44
45     @Bean(name = "dataSource")
46     public MyDataSource createDataSource() {
47         MyDataSource myDataSource = new MyDataSource();
48         Map<Object, Object> map = new HashMap<>();
49         map.put("core", dataSourceCore);
50         map.put("card", dataSourceCard);
51         myDataSource.setTargetDataSources(map);
52         myDataSource.setDefaultTargetDataSource(dataSourceCore);
53         return myDataSource;
54     }
55
56     //省略非关键代码
57 }
```

最后还剩下一个问题，setDataSource 这个方法什么时候执行呢？

我们可以用 Spring AOP 来设置，把配置的数据源类型都设置成注解标签，Service 层中在切换数据源的方法上加上注解标签，就会调用相应的方法切换数据源。

我们定义了一个新的注解 @DataSource，可以直接加在 Service() 上，实现数据库切换：

 复制代码

```
1  @Documented
2  @Target({ElementType.TYPE, ElementType.METHOD})
3  @Retention(RetentionPolicy.RUNTIME)
4  public @interface DataSource {
5      String value();
6
7      String core = "core";
8  }
```

```
9     String card = "card";
10 }
```

声明方法如下：

```
1 @DataSource(DataSource.card)
```

[复制代码](#)

另外，我们还需要写一个 Spring AOP 来对相应的服务方法进行拦截，完成数据源的切换操作。特别要注意的是，这里要加上一个 `@Order(1)` 标记它的初始化顺序。这个 `Order` 值一定要比事务的 AOP 切面的值小，这样可以获得更高的优先级，否则自动切换数据源将会失效。

```
1 @Aspect
2 @Service
3 @Order(1)
4 public class DataSourceSwitch {
5     @Around("execution(* com.spring.puzzle.others.transaction.example3.CardSer
6     public void around(ProceedingJoinPoint point) throws Throwable {
7         Signature signature = point.getSignature();
8         MethodSignature methodSignature = (MethodSignature) signature;
9         Method method = methodSignature.getMethod();
10        if (method.isAnnotationPresent(DataSource.class)) {
11            DataSource dataSource = method.getAnnotation(DataSource.class);
12            MyDataSource.setDataSource(dataSource.value());
13            System.out.println("数据源切换至：" + MyDataSource.getDatasource());
14        }
15        point.proceed();
16        MyDataSource.clearDataSource();
17        System.out.println("数据源已移除！");
18    }
19 }
```

[复制代码](#)

最后，我们实现了 Card 的发卡逻辑，在方法前声明了切换数据库：

```
1 @Service
2 public class CardService {
3     @Autowired
4     private CardMapper cardMapper;
```

[复制代码](#)

```
5
6     @Transactional(propagation = Propagation.REQUIRES_NEW)
7     @DataSource(DataSource.card)
8     public void createCard(int studentId) throws Exception {
9         Card card = new Card();
10        card.setStudentId(studentId);
11        card.setBalance(50);
12        cardMapper.saveCard(card);
13    }
14 }
```

并在 saveStudent() 里调用了发卡逻辑：

 复制代码

```
1 @Transactional(rollbackFor = Exception.class)
2 public void saveStudent(String realname) throws Exception {
3     Student student = new Student();
4     student.setRealname(realname);
5     studentService.doSaveStudent(student);
6     try {
7         courseService.regCourse(student.getId());
8         cardService.createCard(student.getId());
9     } catch (Exception e) {
10        e.printStackTrace();
11    }
12 }
```


执行一下，一切正常，两个库的数据都可以正常保存了。

最后我们来看一下整个过程的调用栈，重新过一遍流程（这里我略去了不重要的部分）。

```
determineTargetDataSource:202, AbstractRoutingDataSource (org.springframework.jdbc.datasource.lookup)
getConnection:169, AbstractRoutingDataSource (org.springframework.jdbc.datasource.lookup)
doBegin:262, DataSourceTransactionManager (org.springframework.jdbc.datasource)
getTransaction:378, AbstractPlatformTransactionManager (org.springframework.transaction.support)
createTransactionIfNecessary:474, TransactionAspectSupport (org.springframework.transaction.interceptor)
invokeWithinTransaction:289, TransactionAspectSupport (org.springframework.transaction.interceptor)
invoke:98, TransactionInterceptor (org.springframework.transaction.interceptor)
proceed:186, ReflectiveMethodInvocation (org.springframework.aop.framework)
proceed:88, MethodInvocationProceedingJoinPoint (org.springframework.aop.aspectj)
around:28, DataSourceSwitch (learning.spring.ioc.service.data)
```

在创建了事务以后，会通过 DataSourceTransactionManager.doBegin() 获取相应的数据库连接：



 复制代码

```
1 protected void doBegin(Object transaction, TransactionDefinition definition) {
2     DataSourceTransactionObject txObject = (DataSourceTransactionObject) transa
3     Connection con = null;
4
5     try {
6         if (!txObject.hasConnectionHolder() ||
7 txObject.getConnectionHolder().isSynchronizedWithTransaction()) {
8             Connection newCon = obtainDataSource().getConnection();
9             txObject.setConnectionHolder(new ConnectionHolder(newCon), true);
10        }
11
12        //省略非关键代码
13    }
```

这里的 `obtainDataSource().getConnection()` 调用到了 `AbstractRoutingDataSource.getConnection()`，这就与我们实现的功能顺利会师了。

 复制代码

```
1 public Connection getConnection() throws SQLException {
2     return determineTargetDataSource().getConnection();
3 }
```

## 重点回顾

通过以上两个案例，相信你对 Spring 的事务机制已经有了深刻的认识，最后总结下重点：

Spring 在事务处理中有一个很重要的属性 `Propagation`，主要用来配置当前需要执行的方法如何使用事务，以及与其它事务之间的关系。

Spring 默认的传播属性是 `REQUIRED`，在有事务状态下执行，如果当前没有事务，则创建新的事务；

Spring 事务是可以对多个数据源生效，它提供了一个抽象类 `AbstractRoutingDataSource`，通过实现这个抽象类，我们可以实现自定义的数据库切换。

## 思考题

结合案例 2，请你思考这样一个问题：在这个案例中，我们在 `CardService` 类方法上声明了这样的事务传播属性，`@Transactional(propagation =`

Propagation.REQUIRES\_NEW), 如果使用 Spring 的默认声明行不行, 为什么?

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 19 | Spring 事务常见错误 (上)

下一篇 21 | Spring Rest Template 常见错误

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 (3)

写留言



Wallace Pang

2021-06-23

spring boot多数据源更简单

展开

**梦尘**

2021-06-22

如果用默认的传播属性，切换应该会失败，会一直使用前一个数据源。

AbstractPlatformTransactionManager.getTransaction()下的isExistingTransaction应该是true，所以DataSourceTransactionManager.doBegin()不会再次进入了。



2021-06-18

**干货满满**

展开 ▾

