太难了~面试官让我结合案例讲讲自己对Spring事务传播行为的理解

三太子敖丙 敖丙 2020-04-30 08:48

作者|handaqiang

地址|https://segmentfault.com/a/1190000013341344

前言

大家好，我是敖丙！最近在重新整理 Spring 事务相关的内容，在看 Spring 事务传播行为这块内容的时候，发现了这篇优秀的文章，分享一下。

Spring 在 TransactionDefinition 接口中规定了 7 种类型的事务传播行为。事务传播行为是 Spring 框架独有的事务增强特性，他不属于的事务实际提供方数据库行为。

这是 Spring 为我们提供的强大的工具箱，使用事务传播行可以为我们的开发工作提供许多便利。

但是人们对他的误解也颇多，你一定也听过“service 方法事务最好不要嵌套”的传言。

要想正确的使用工具首先需要了解工具。本文对七种事务传播行为做详细介绍，内容主要代码示例的方式呈现。

基础概念

1. 什么是事务传播行为？

事务传播行为用来描述由某一个事务传播行为修饰的方法被嵌套进另一个方法的时事务如何传播。

用伪代码说明：

```
public void methodA(){
   methodB();
   //doSomething
}

@Transaction(Propagation=XXX)
public void methodB(){
```

```
    //doSomething
}
```

代码中methodA()方法嵌套调用了methodB()方法，methodB()的事务传播行为由@Transaction(Propagation=XXX)设置决定。这里需要注意的是methodA()并没有开启事务，某一个事务传播行为修饰的方法并不是必须要在开启事务的外围方法中调用。

2. Spring 中七种事务传播行为

事务传播行为类型	说明
PROPAGATION_REQUIRED	如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。这是最常见的选择。
PROPAGATION_SUPPORTS	支持当前事务，如果当前没有事务，就以非事务方式执行。
PROPAGATION_MANDATORY	使用当前的事务，如果当前没有事务，就抛出异常。
PROPAGATION_REQUIRES_NEW	新建事务，如果当前存在事务，把当前事务挂起。
PROPAGATION_NOT_SUPPORTED	以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
PROPAGATION_NEVER	以非事务方式执行，如果当前存在事务，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。

定义非常简单，也很好理解，下面我们就进入代码测试部分，验证我们的理解是否正确。

代码验证

文中代码以传统三层结构中两层呈现，即 Service 和 Dao 层，由 Spring 负责依赖注入和注解式事务管理，DAO 层由 Mybatis 实现，你也可以使用任何喜欢的方式，例如，Hibernate,JPA,JDBCTemplate 等。数据库使用的是 MySQL 数据库，你也可以使用任何支持事务的数据库，并不会影响验证结果。

首先我们在数据库中创建两张表：

user1

```
CREATE TABLE `user1` (
  `id` INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(45) NOT NULL DEFAULT '',
  PRIMARY KEY(`id`)
)
ENGINE = InnoDB;
```

user2

```
CREATE TABLE `user2` (
  `id` INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(45) NOT NULL DEFAULT '',
  PRIMARY KEY(`id`)
)
ENGINE = InnoDB;
```

然后编写相应的 Bean 和 DAO 层代码：

User1

```
public class User1 {
    private Integer id;
    private String name;
   //get和set方法省略...
}
```

User2

```
public class User2 {
    private Integer id;
    private String name;
   //get和set方法省略...
}
```

User1Mapper

```
public interface User1Mapper {
    int insert(User1 record);
    User1 selectByPrimaryKey(Integer id);
    //其他方法省略...
}
```

User2Mapper

```
public interface User2Mapper {
    int insert(User2 record);
    User2 selectByPrimaryKey(Integer id);
    //其他方法省略...
}
```

最后也是具体验证的代码由 service 层实现，下面我们分情况列举。

1.PROPAGATION_REQUIRED

我们为 User1Service 和 User2Service 相应方法加上Propagation.REQUIRED属性。

User1Service 方法：

```
@Service
public class User1ServiceImpl implements User1Service {
    //省略其他...
    @Override
    @Transactional(propagation = Propagation.REQUIRED)
    public void addRequired(User1 user){
        user1Mapper.insert(user);
    }
}
```

User2Service 方法：

```
@Service
public class User2ServiceImpl implements User2Service {
    //省略其他...
    @Override
    @Transactional(propagation = Propagation.REQUIRED)
    public void addRequired(User2 user){
        user2Mapper.insert(user);
    }
    @Override
    @Transactional(propagation = Propagation.REQUIRED)
    public void addRequiredException(User2 user){
        user2Mapper.insert(user);
        throw new RuntimeException();
    }

}
```

1.1 场景一

此场景外围方法没有开启事务。

验证方法 **1**：

```
    @Override
    public void notransaction_exception_required_required(){
        User1 user1=new User1();
        user1.setName("张三");
        user1Service.addRequired(user1);

        User2 user2=new User2();
        user2.setName("李四");
        user2Service.addRequired(user2);

        throw new RuntimeException();
    }
```

验证方法 **2**：

```
    @Override
    public void notransaction_required_required_exception(){
        User1 user1=new User1();
        user1.setName("张三");
        user1Service.addRequired(user1);

        User2 user2=new User2();
        user2.setName("李四");
        user2Service.addRequiredException(user2);
    }
```

分别执行验证方法，结果：

验证方法序号	数据库结果	结果分析
1	“张三”、“李四”均插入。	外围方法未开启事务，插入“张三”、“李四”方法在自己的事务中独立运行，外围方法异常不影响内部插入“张三”、“李四”方法独立的事务。
2	“张三”插入，“李四”未插入。	外围方法没有事务，插入“张三”、“李四”方法都在自己的事务中独立运行,所以插入“李四”方法抛出异常只会回滚插入“李四”方法，插入“张三”方法不受影响。

结论：通过这两个方法我们证明了在外围方法未开启事务的情况下Propagation.REQUIRED修饰的内部方法会新开启自己的事务，且开启的事务相互独立，互不干扰。

1.2 场景二

外围方法开启事务，这个是使用率比较高的场景。

验证方法 **1**：

```
@Override
 @Transactional(propagation = Propagation.REQUIRED)
 public void transaction_exception_required_required(){
     User1 user1=new User1();
     user1.setName("张三");
     user1Service.addRequired(user1);

     User2 user2=new User2();
     user2.setName("李四");
     user2Service.addRequired(user2);

     throw new RuntimeException();
 }
```

验证方法 **2**：

```
@Override
@Transactional(propagation = Propagation.REQUIRED)
public void transaction_required_required_exception(){
    User1 user1=new User1();
    user1.setName("张三");
    user1Service.addRequired(user1);

    User2 user2=new User2();
    user2.setName("李四");
    user2Service.addRequiredException(user2);
}
```

验证方法 **3**：

```
@Transactional
@Override
public void transaction_required_required_exception_try(){
    User1 user1=new User1();
    user1.setName("张三");
    user1Service.addRequired(user1);
```

```
        User2 user2=new User2();
        user2.setName("李四");
        try {
            user2Service.addRequiredException(user2);
        } catch (Exception e) {
            System.out.println("方法回滚");
        }
    }
```

分别执行验证方法，结果：

验证方法序号	数据库结果	结果分析
1	“张三”、“李四”均未插入。	外围方法开启事务，内部方法加入外围方法事务，外围方法回滚，内部方法也要回滚。
2	“张三”、“李四”均未插入。	外围方法开启事务，内部方法加入外围方法事务，内部方法抛出异常回滚，外围方法感知异常致使整体事务回滚。
3	“张三”、“李四”均未插入。	外围方法开启事务，内部方法加入外围方法事务，内部方法抛出异常回滚，即使方法被catch不被外围方法感知，整个事务依然回滚。

结论：以上试验结果我们证明在外围方法开启事务的情况下Propagation.REQUIRED修饰的内部方法会加入到外围方法的事务中，所有Propagation.REQUIRED修饰的内部方法和外围方法均属于同一事务，只要一个方法回滚，整个事务均回滚。

2.PROPAGATION_REQUIRES_NEW

我们为 User1Service 和 User2Service 相应方法加上Propagation.REQUIRES_NEW属性。

User1Service 方法：

```
@Service
public class User1ServiceImpl implements User1Service {
    //省略其他...
    @Override
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void addRequiresNew(User1 user){
        user1Mapper.insert(user);
    }
    @Override
    @Transactional(propagation = Propagation.REQUIRED)
    public void addRequired(User1 user){
        user1Mapper.insert(user);
    }
}
```

User2Service 方法：

```
@Service
public class User2ServiceImpl implements User2Service {
    //省略其他...
    @Override
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void addRequiresNew(User2 user){
        user2Mapper.insert(user);
    }

    @Override
```

```
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void addRequiresNewException(User2 user){
        user2Mapper.insert(user);
        throw new RuntimeException();
    }
}
```

2.1 场景一

外围方法没有开启事务。

验证方法 **1**：

```
@Override
public void notransaction_exception_requiresNew_requiresNew(){
    User1 user1=new User1();
    user1.setName("张三");
    user1Service.addRequiresNew(user1);

    User2 user2=new User2();
    user2.setName("李四");
    user2Service.addRequiresNew(user2);
    throw new RuntimeException();

}
```

验证方法 **2**：

```
@Override
public void notransaction_requiresNew_requiresNew_exception(){
    User1 user1=new User1();
    user1.setName("张三");
    user1Service.addRequiresNew(user1);

    User2 user2=new User2();
    user2.setName("李四");
    user2Service.addRequiresNewException(user2);
}
```

分别执行验证方法，结果：

验证方法序号	数据库结果	结果分析
1	“张三”插入，“李四”插入。	外围方法没有事务，插入“张三”、“李四”方法都在自己的事务中独立运行,外围方法抛出异常回滚不会影响内部方法。
2	“张三”插入，“李四”未插入	外围方法没有开启事务，插入“张三”方法和插入“李四”方法分别开启自己的事务，插入“李四”方法抛出异常回滚，其他事务不受影响。

结论：通过这两个方法我们证明了在外围方法未开启事务的情况下Propagation.REQUIRES_NEW修饰的内部方法会新开启自己的事务，且开启的事务相互独立，互不干扰。

2.2 场景二

外围方法开启事务。

验证方法 1：

```
@Override
@Transactional(propagation = Propagation.REQUIRED)
public void transaction_exception_required_requiresNew_requiresNew(){
    User1 user1=new User1();
    user1.setName("张三");
    user1Service.addRequired(user1);

    User2 user2=new User2();
    user2.setName("李四");
    user2Service.addRequiresNew(user2);

    User2 user3=new User2();
    user3.setName("王五");
    user2Service.addRequiresNew(user3);
    throw new RuntimeException();
}
```

验证方法 2：

```
@Override
@Transactional(propagation = Propagation.REQUIRED)
public void transaction_required_requiresNew_requiresNew_exception(){
    User1 user1=new User1();
    user1.setName("张三");
    user1Service.addRequired(user1);

    User2 user2=new User2();
    user2.setName("李四");
    user2Service.addRequiresNew(user2);

    User2 user3=new User2();
    user3.setName("王五");
    user2Service.addRequiresNewException(user3);
}
```

验证方法 3：

```
@Override
@Transactional(propagation = Propagation.REQUIRED)
public void transaction_required_requiresNew_requiresNew_exception_try(){
    User1 user1=new User1();
    user1.setName("张三");
    user1Service.addRequired(user1);

    User2 user2=new User2();
    user2.setName("李四");
    user2Service.addRequiresNew(user2);
    User2 user3=new User2();
    user3.setName("王五");
    try {
        user2Service.addRequiresNewException(user3);
    } catch (Exception e) {
        System.out.println("回滚");
```

```
        }
    }
```

分别执行验证方法，结果：

验证方法序号	数据库结果	结果分析
1	“张三”未插入，“李四”插入，“王五”插入。	外围方法开启事务，插入“张三”方法和外围方法一个事务，插入“李四”方法、插入“王五”方法分别在独立的新建事务中，外围方法抛出异常只回滚和外围方法同一事务的方法，故插入“张三”的方法回滚。
2	“张三”未插入，“李四”插入，“王五”未插入。	外围方法开启事务，插入“张三”方法和外围方法一个事务，插入“李四”方法、插入“王五”方法分别在独立的新建事务中。插入“王五”方法抛出异常，首先插入 “王五”方法的事务被回滚，异常继续抛出被外围方法感知，外围方法事务亦被回滚，故插入“张三”方法也被回滚。
3	“张三”插入，“李四”插入，“王五”未插入。	外围方法开启事务，插入“张三”方法和外围方法一个事务，插入“李四”方法、插入“王五”方法分别在独立的新建事务中。插入“王五”方法抛出异常，首先插入“王五”方法的事务被回滚，异常被catch不会被外围方法感知，外围方法事务不回滚，故插入“张三”方法插入成功。

结论：在外围方法开启事务的情况下Propagation.REQUIRES_NEW修饰的内部方法依然会单独开启独立事务，且与外部方法事务也独立，内部方法之间、内部方法和外部方法事务均相互独立，互不干扰。

3.PROPAGATION_NESTED

我们为 User1Service 和 User2Service 相应方法加上Propagation.NESTED属性。

User1Service 方法：

```
@Service
public class User1ServiceImpl implements User1Service {
    //省略其他...
    @Override
    @Transactional(propagation = Propagation.NESTED)
    public void addNested(User1 user){
        user1Mapper.insert(user);
    }
}
```

User2Service 方法：

```
@Service
public class User2ServiceImpl implements User2Service {
    //省略其他...
    @Override
    @Transactional(propagation = Propagation.NESTED)
    public void addNested(User2 user){
        user2Mapper.insert(user);
    }

    @Override
    @Transactional(propagation = Propagation.NESTED)
    public void addNestedException(User2 user){
```

```
        user2Mapper.insert(user);
        throw new RuntimeException();
    }
}
```

3.1 场景一

此场景外围方法没有开启事务。

验证方法 **1**：

```
@Override
public void notransaction_exception_nested_nested(){
    User1 user1=new User1();
    user1.setName("张三");
    user1Service.addNested(user1);

    User2 user2=new User2();
    user2.setName("李四");
    user2Service.addNested(user2);
    throw new RuntimeException();
}
```

验证方法 **2**：

```
@Override
public void notransaction_nested_nested_exception(){
    User1 user1=new User1();
    user1.setName("张三");
    user1Service.addNested(user1);

    User2 user2=new User2();
    user2.setName("李四");
    user2Service.addNestedException(user2);
}
```

分别执行验证方法，结果：

验证方法序号	数据库结果	结果分析
1	“张三”、“李四”均插入。	外围方法未开启事务，插入“张三”、“李四”方法在自己的事务中独立运行，外围方法异常不影响内部插入“张三”、“李四”方法独立的事务。
2	“张三”插入，“李四”未插入。	外围方法没有事务，插入“张三”、“李四”方法都在自己的事务中独立运行,所以插入“李四”方法抛出异常只会回滚插入“李四”方法，插入“张三”方法不受影响。

结论：通过这两个方法我们证明了在外围方法未开启事务的情况下Propagation.NESTED和Propagation.REQUIRED作用相同，修饰的内部方法都会新开启自己的事务，且开启的事务相互独立，互不干扰。

3.2 场景二

外围方法开启事务。

验证方法 **1**：

```
@Transactional
@Override
public void transaction_exception_nested_nested(){
    User1 user1=new User1();
    user1.setName("张三");
    user1Service.addNested(user1);

    User2 user2=new User2();
    user2.setName("李四");
    user2Service.addNested(user2);
    throw new RuntimeException();
}
```

验证方法 **2**：

```
@Transactional
@Override
public void transaction_nested_nested_exception(){
    User1 user1=new User1();
    user1.setName("张三");
    user1Service.addNested(user1);

    User2 user2=new User2();
    user2.setName("李四");
    user2Service.addNestedException(user2);
}
```

验证方法 **3**：

```
@Transactional
@Override
public void transaction_nested_nested_exception_try(){
    User1 user1=new User1();
    user1.setName("张三");
    user1Service.addNested(user1);

    User2 user2=new User2();
    user2.setName("李四");
    try {
        user2Service.addNestedException(user2);
    } catch (Exception e) {
        System.out.println("方法回滚");
    }
}
```

分别执行验证方法，结果：

验证方法序号	数据库结果	结果分析
1	“张三”、“李四”均未插入。	外围方法开启事务，内部事务为外围事务的子事务，外围方法回滚，内部方法也要回滚。
2	“张三”、“李四”均未插入。	外围方法开启事务，内部事务为外围事务的子事务，内部方法抛出异常回滚，且外围方法感知异常致使整体事务回滚。
3	“张三”插入、“李四”未插入。	外围方法开启事务，内部事务为外围事务的子事务，插入“李四”内部方法抛出异常，可以单独对子事务回滚。

结论：以上试验结果我们证明在外围方法开启事务的情况下Propagation.NESTED修饰的内部方法属于外部事务的子事务，外围主事务回滚，子事务一定回滚，而内部子事务可以单独回滚而不影响外围主事务和其他子事务

4. REQUIRED,REQUIRES_NEW,NESTED 异同

由“1.2 场景二”和“3.2 场景二”对比，我们可知：**NESTED** 和 **REQUIRED** 修饰的内部方法都属于外围方法事务，如果外围方法抛出异常，这两种方法的事务都会被回滚。但是 **REQUIRED** 是加入外围方法事务，所以和外围事务同属于一个事务，一旦 **REQUIRED** 事务抛出异常被回滚，外围方法事务也将被回滚。而 **NESTED** 是外围方法的子事务，有单独的保存点，所以 **NESTED** 方法抛出异常被回滚，不会影响到外围方法的事务。

由“2.2 场景二”和“3.2 场景二”对比，我们可知：**NESTED** 和 **REQUIRES_NEW** 都可以做到内部方法事务回滚而不影响外围方法事务。但是因为 **NESTED** 是嵌套事务，所以外围方法回滚之后，作为外围方法事务的子事务也会被回滚。而 **REQUIRES_NEW** 是通过开启新的事务实现的，内部事务和外围事务是两个事务，外围事务回滚不会影响内部事务。

5. 其他事务传播行为

鉴于文章篇幅问题，其他事务传播行为的测试就不在此一一描述了，感兴趣的读者可以去源码中自己寻找相应测试代码和结果解释。传送门：
https://github.com/TmTse/transaction-test

模拟用例

介绍了这么多事务传播行为，我们在实际工作中如何应用呢？下面我来举一个示例：

假设我们有一个注册的方法，方法中调用添加积分的方法，如果我们希望添加积分不会影响注册流程（即添加积分执行失败回滚不能使注册方法也回滚），我们会这样写：

```
@Service
public class UserServiceImpl implements UserService {

    @Transactional
    public void register(User user){

        try {
            membershipPointService.addPoint(Point point);
        } catch (Exception e) {
           //省略...
        }
        //省略...
    }
    //省略...
}
```

我们还规定注册失败要影响addPoint()方法（注册方法回滚添加积分方法也需要回滚），那么addPoint()方法就需要这样实现：

```
@Service
public class MembershipPointServiceImpl implements MembershipPointService{

    @Transactional(propagation = Propagation.NESTED)
    public void addPoint(Point point){
```

```
        try {
            recordService.addRecord(Record record);
        } catch (Exception e) {
           //省略...
        }
        //省略...
    }
    //省略...
}
```

我们注意到了在addPoint()中还调用了addRecord()方法，这个方法用来记录日志。他的实现如下：

```
@Service
public class RecordServiceImpl implements RecordService{

    @Transactional(propagation = Propagation.NOT_SUPPORTED)
    public void addRecord(Record record){

        //省略...
    }
    //省略...
}
```

我们注意到addRecord()方法中propagation = Propagation.NOT_SUPPORTED，因为对于日志无所谓精确，可以多一条也可以少一条，所以addRecord()方法本身和外围addPoint()方法抛出异常都不会使addRecord()方法回滚，并且addRecord()方法抛出异常也不会影响外围addPoint()方法的执行。

通过这个例子相信大家对事务传播行为的使用有了更加直观的认识，通过各种属性的组合确实能让我们的业务实现更加灵活多样。

结论

通过上面的介绍，相信大家对 Spring 事务传播行为有了更加深入的理解，希望大家日常开发工作有所帮助。

喜欢此内容的人还喜欢