Spring 事务——事务的传播机制

作者：reese
原文链接：https://ld246.com/article/1599034326955
来源网站：链滴

一、什么是事务的传播?

简单的理解就是多个事务方法相互调用时,事务如何在这些方法间传播。

举个栗子，方法A是一个事务的方法，方法A执行过程中调用了方法B，那么方法B有无事务以及方法B事务的要求不同都会对方法A的事务具体执行造成影响，同时方法A的事务对方法B的事务执行也有影，这种影响具体是什么就由两个方法所定义的事务传播类型所决定。

二、Spring事务传播类型枚举Propagation介绍

在Spring中对于事务的传播行为定义了七种类型分别是：**REQUIRED、SUPPORTS、MANDATOR、REQUIRES_NEW、NOT_SUPPORTED、NEVER、NESTED。**

在Spring源码中这七种类型被定义为了枚举。源码在org.springframework.transaction.annotation下的Propagation，源码中注释很多，对传播行为的七种类型的不同含义都有解释，后文中锤子我也给大家分析，我在这里就不贴所有的源码，只把这个类上的注解贴一下，翻译一下就是：表示与TransactionDefinition接口相对应的用于@Transactional注解的事务传播行为的枚举。

也就是说枚举类Propagation是为了结合@Transactional注解使用而设计的，这个枚举里面定义的事传播行为类型与TransactionDefinition中定义的事务传播行为类型是对应的，所以在使用@Transactinal注解时我们就要使用Propagation枚举类来指定传播行为类型，而不直接使用TransactionDefinitin接口里定义的属性。

在TransactionDefinition接口中定义了Spring事务的一些属性，不仅包括事务传播特性类型，还包括事务的隔离级别类型（事务的隔离级别后面文章会详细讲解），更多详细信息，大家可以打开源码自翻译一下里面的注释

```
package org.springframework.transaction.annotation;
import org.springframework.transaction.TransactionDefinition;
/**
 * Enumeration that represents transaction propagation behaviors for use
 * with the {@link Transactional} annotation, corresponding to the
 * {@link TransactionDefinition} interface.
 *
 * @author Colin Sampaleanu
 * @author Juergen Hoeller
 * @since 1.2
 */
public enum Propagation {
    ...
}
```

三、七种事务传播行为详解与示例

在介绍七种事务传播行为前，我们先设计一个场景，帮助大家理解，场景描述如下

现有两个方法A和B，方法A执行会在数据库ATable插入一条数据，方法B执行会在数据库BTable插入条数据，伪代码如下:

```
//将传入参数a存入ATable
pubilc void A(a){
    insertIntoATable(a);
```

```
}
//将传入参数b存入BTable
public void B(b){
    insertIntoBTable(b);
}
```

接下来，我们看看在如下场景下，没有事务，情况会怎样

```
public void testMain(){
    A(a1);  //调用A入参a1
    testB();    //调用testB
}
public void testB(){
    B(b1);  //调用B入参b1
    throw Exception;     //发生异常抛出
    B(b2);  //调用B入参b2
}
```

在这里要做一个重要提示：**Spring中事务的默认实现使用的是AOP，也就是代理的方式，如果大家使用代码测试时，同一个Service类中的方法相互调用需要使用注入的对象来调用，不要直接使用this方法名来调用，this.方法名调用是对象内部方法调用，不会通过Spring代理，也就是事务不会起作用**

以上伪代码描述的一个场景，方法testMain和testB都没有事务，执行testMain方法，那么结果会怎样呢?

相信大家都知道了，就是a1数据成功存入ATable表，b1数据成功存入BTable表，而在抛出异常后b2据存储就不会执行，也就是b2数据不会存入数据库，这就是没有事务的场景。

可想而知，在上一篇文章（认识事务）中举例的转账操作，如果在某一步发生异常，且没有事务，那钱是不是就凭空消失了，所以事务在数据库操作中的重要性可想而知。接下我们就开始理解七种不同务传播类型的含义

REQUIRED(Spring默认的事务传播类型)

如果当前没有事务，则自己新建一个事务，如果当前存在事务，则加入这个事务

源码说明如下：

```
/**
     * Support a current transaction, create a new one if none exists.
     * Analogous to EJB transaction attribute of the same name.
     * <p>This is the default setting of a transaction annotation.
     */
    REQUIRED(TransactionDefinition.PROPAGATION_REQUIRED),
```

*(示例1)*根据场景举栗子,我们在testMain和testB上声明事务，设置传播行为REQUIRED，伪代码如：

```
@Transactional(propagation = Propagation.REQUIRED)
public void testMain(){
    A(a1);  //调用A入参a1
    testB();    //调用testB
}
@Transactional(propagation = Propagation.REQUIRED)
```

```
public void testB(){
    B(b1);  //调用B入参b1
    throw Exception;     //发生异常抛出
    B(b2);  //调用B入参b2
}
```

该场景下执行testMain方法结果如何呢?

数据库没有插入新的数据，数据库还是保持着执行testMain方法之前的状态，没有发生改变。testMain上声明了事务，在执行testB方法时就加入了testMain的事务（**当前存在事务，则加入这个事务**），执行testB方法抛出异常后事务会发生回滚，又testMain和testB使用的同一个事务，所以事务回滚后testMain和testB中的操作都会回滚，也就使得数据库仍然保持初始状态

*(示例2)*根据场景再举一个栗子,我们只在testB上声明事务，设置传播行为REQUIRED，伪代码如下：

```
public void testMain(){
    A(a1);  //调用A入参a1
    testB();    //调用testB
}
@Transactional(propagation = Propagation.REQUIRED)
public void testB(){
    B(b1);  //调用B入参b1
    throw Exception;     //发生异常抛出
    B(b2);  //调用B入参b2
}
```

这时的执行结果又如何呢?

数据a1存储成功，数据b1和b2没有存储。由于testMain没有声明事务，testB有声明事务且传播行为REQUIRED，所以在执行testB时会自己新建一个事务（**如果当前没有事务，则自己新建一个事务**），testB抛出异常则只有testB中的操作发生了回滚，也就是b1的存储会发生回滚，但a1数据不会回滚，所最终a1数据存储成功，b1和b2数据没有存储

SUPPORTS

当前存在事务，则加入当前事务，如果当前没有事务，就以非事务方法执行

源码注释如下(太长省略了一部分)，其中里面有一个提醒翻译一下就是：“对于具有事务同步的事务理器，SUPPORTS与完全没有事务稍有不同，因为它定义了可能应用同步的事务范围”。这个是与事同步管理器相关的一个注意项，这里不过多讨论。

```
/**
     * Support a current transaction, execute non-transactionally if none exists.
     * Analogous to EJB transaction attribute of the same name.
     * <p>Note: For transaction managers with transaction synchronization,
     * {@code SUPPORTS} is slightly different from no transaction at all,
     * as it defines a transaction scope that synchronization will apply for.
     ...
     */
    SUPPORTS(TransactionDefinition.PROPAGATION_SUPPORTS),
```

*(示例3)*根据场景举栗子，我们只在testB上声明事务，设置传播行为SUPPORTS，伪代码如下：

```
public void testMain(){
```

```
    A(a1);  //调用A入参a1
    testB();    //调用testB
}
@Transactional(propagation = Propagation.SUPPORTS)
public void testB(){
    B(b1);  //调用B入参b1
    throw Exception;     //发生异常抛出
    B(b2);  //调用B入参b2
}
```

这种情况下，执行testMain的最终结果就是，a1，b1存入数据库，b2没有存入数据库。由于testMai没有声明事务，且testB的事务传播行为是SUPPORTS，所以执行testB时就是没有事务的（**如果当前有事务，就以非事务方法执行**），则在testB抛出异常时也不会发生回滚，所以最终结果就是a1和b1储成功，b2没有存储。

那么当我们在testMain上声明事务且使用REQUIRED传播方式的时候，这个时候执行testB就满足**当存在事务，则加入当前事务**，在testB抛出异常时事务就会回滚，最终结果就是a1，b1和b2都不会存到数据库

MANDATORY

当前存在事务，则加入当前事务，如果当前事务不存在，则抛出异常。

源码注释如下：

```
/**
     * Support a current transaction, throw an exception if none exists.
     * Analogous to EJB transaction attribute of the same name.
     */
    MANDATORY(TransactionDefinition.PROPAGATION_MANDATORY),
```

*(示例4)*场景举栗子，我们只在testB上声明事务，设置传播行为为MANDATORY，伪代码如下：

```
public void testMain(){
    A(a1);  //调用A入参a1
    testB();    //调用testB
}
@Transactional(propagation = Propagation.MANDATORY)
public void testB(){
    B(b1);  //调用B入参b1
    throw Exception;     //发生异常抛出
    B(b2);  //调用B入参b2
}
```

这种情形的执行结果就是a1存储成功，而b1和b2没有存储。b1和b2没有存储，并不是事务回滚的原，而是因为testMain方法没有声明事务，在去执行testB方法时就直接抛出事务要求的异常（**如果当事务不存在，则抛出异常**），所以testB方法里的内容就没有执行。

那么如果在testMain方法进行事务声明，并且设置为REQUIRED，则执行testB时就会使用testMain经开启的事务，遇到异常就正常的回滚了。

REQUIRES_NEW

创建一个新事务，如果存在当前事务，则挂起该事务。

可以理解为设置事务传播类型为REQUIRES_NEW的方法，在执行时，不论当前是否存在事务，总是新建一个事务。

源码注释如下

```
/**
     * Create a new transaction, and suspend the current transaction if one exists.
     ...
     */
    REQUIRES_NEW(TransactionDefinition.PROPAGATION_REQUIRES_NEW),
```

*(示例5)*场景举栗子，为了说明设置REQUIRES_NEW的方法会开启新事务，我们把异常发生的位置到了testMain，然后给testMain声明事务，传播类型设置为REQUIRED，testB也声明事务，设置传类型为REQUIRES_NEW，伪代码如下

```
@Transactional(propagation = Propagation.REQUIRED)
public void testMain(){
    A(a1);  //调用A入参a1
    testB();    //调用testB
    throw Exception;     //发生异常抛出
}
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void testB(){
    B(b1);  //调用B入参b1
    B(b2);  //调用B入参b2
}
```

这种情形的执行结果就是a1没有存储，而b1和b2存储成功，因为testB的事务传播设置为REQUIRES_EW,所以在执行testB时会开启一个新的事务，testMain中发生的异常时在testMain所开启的事务中所以这个异常不会影响testB的事务提交，testMain中的事务会发生回滚，所以最终a1就没有存储，而1和b2就存储成功了。

与这个场景对比的一个场景就是testMain和testB都设置为REQUIRED，那么上面的代码执行结果就所有数据都不会存储，因为testMain和testMain是在同一个事务下的，所以事务发生回滚时，所有数据都会回滚

NOT_SUPPORTED

始终以非事务方式执行,如果当前存在事务，则挂起当前事务

可以理解为设置事务传播类型为NOT_SUPPORTED的方法，在执行时，不论当前是否存在事务，都以非事务的方式运行。

源码说明如下

```
/**
     * Execute non-transactionally, suspend the current transaction if one exists.
     ...
     */
    NOT_SUPPORTED(TransactionDefinition.PROPAGATION_NOT_SUPPORTED),
```

*(示例6)*场景举栗子，testMain传播类型设置为REQUIRED，testB传播类型设置为NOT_SUPPORTE

，且异常抛出位置在testB中，伪代码如下

```
@Transactional(propagation = Propagation.REQUIRED)
public void testMain(){
    A(a1);  //调用A入参a1
    testB();    //调用testB
}
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public void testB(){
    B(b1);  //调用B入参b1
    throw Exception;     //发生异常抛出
    B(b2);  //调用B入参b2
}
```

该场景的执行结果就是a1和b2没有存储，而b1存储成功。testMain有事务，而testB不使用事务，所执行中testB的存储b1成功，然后抛出异常，此时testMain检测到异常事务发生回滚，但是由于testB在事务中，所以只有testMain的存储a1发生了回滚，最终只有b1存储成功，而a1和b1都没有存储

NEVER

不使用事务，如果当前事务存在，则抛出异常

很容易理解，就是我这个方法不使用事务，并且调用我的方法也不允许有事务，如果调用我的方法有务则我直接抛出异常。

源码注释如下：

```
/**
     * Execute non-transactionally, throw an exception if a transaction exists.
     * Analogous to EJB transaction attribute of the same name.
     */
    NEVER(TransactionDefinition.PROPAGATION_NEVER),
```

*(示例7)*场景举栗子，testMain设置传播类型为REQUIRED，testB传播类型设置为NEVER，并且把tstB中的抛出异常代码去掉，则伪代码如下

```
@Transactional(propagation = Propagation.REQUIRED)
public void testMain(){
    A(a1);  //调用A入参a1
    testB();    //调用testB
}
@Transactional(propagation = Propagation.NEVER)
public void testB(){
    B(b1);  //调用B入参b1
    B(b2);  //调用B入参b2
}
```

该场景执行，直接抛出事务异常，且不会有数据存储到数据库。由于testMain事务传播类型为REQUIED，所以testMain是运行在事务中，而testB事务传播类型为NEVER，所以testB不会执行而是直接出事务异常，此时testMain检测到异常就发生了回滚，所以最终数据库不会有数据存入。

NESTED

如果当前事务存在，则在嵌套事务中执行，否则REQUIRED的操作一样（开启一个事务）

这里需要注意两点：

● 和REQUIRES_NEW的区别

REQUIRES_NEW是新建一个事务并且新开启的这个事务与原有事务无关，而NESTED则是当前存在务时（我们把当前事务称之为父事务）会开启一个嵌套事务（称之为一个子事务）。在NESTED情况父事务回滚时，子事务也会回滚，而在REQUIRES_NEW情况下，原有事务回滚，不会影响新开启的务。

● 和REQUIRED的区别

REQUIRED情况下，调用方存在事务时，则被调用方和调用方使用同一事务，那么被调用方出现异常，由于共用一个事务，所以无论调用方是否catch其异常，事务都会回滚而在NESTED情况下，被调用发生异常时，调用方可以catch其异常，这样只有子事务回滚，父事务不受影响

*(示例8)*场景举栗子，testMain设置为REQUIRED，testB设置为NESTED，且异常发生在testMain，伪代码如下

```
@Transactional(propagation = Propagation.REQUIRED)
public void testMain(){
    A(a1);  //调用A入参a1
    testB();    //调用testB
    throw Exception;     //发生异常抛出
}
@Transactional(propagation = Propagation.NESTED)
public void testB(){
    B(b1);  //调用B入参b1
    B(b2);  //调用B入参b2
}
```

该场景下，所有数据都不会存入数据库，因为在testMain发生异常时，父事务回滚则子事务也跟着回了，可以与*(示例5)*比较看一下，就找出了与REQUIRES_NEW的不同

*(示例9)*场景举栗子，testMain设置为REQUIRED，testB设置为NESTED，且异常发生在testB中，代码如下

```
@Transactional(propagation = Propagation.REQUIRED)
public void testMain(){
    A(a1);  //调用A入参a1
    try{
        testB();    //调用testB
    }catch（Exception e){
    }
    A(a2);
}
@Transactional(propagation = Propagation.NESTED)
public void testB(){
    B(b1);  //调用B入参b1
    throw Exception;     //发生异常抛出
    B(b2);  //调用B入参b2
}
```

这种场景下，结果是a1,a2存储成功，b1和b2存储失败，因为调用方catch了被调方的异常，所以只

子事务回滚了。

同样的代码，如果我们把testB的传播类型改为REQUIRED，结果也就变成了：没有数据存储成功。算在调用方catch了异常，整个事务还是会回滚，因为，调用方和被调方共用的同一个事务