Spring 事务机制详解

原文出处： 陶邦仁

Spring事务机制主要包括声明式事务和编程式事务，此处侧重讲解声明式事务，编程式事务在实际开发中得不到广泛使用，仅供学习参考。

Spring声明式事务让我们从复杂的事务处理中得到解脱。使得我们再也无需要去处理获得连接、关闭连接、事务提交和回滚等这些操作。再也无需要我们在与事务相关的方法中处理大量的try…catch…finally代码。我们在使用Spring声明式事务时，有一个非常重要的概念就是事务属性。事务属性通常由事务的传播行为，事务的隔离级别，事务的超时值和事务只读标志组成。我们在进行事务划分时，需要进行事务定义，也就是配置事务的属性。

下面分别详细讲解，事务的四种属性，仅供诸位学习参考：

Spring在TransactionDefinition接口中定义这些属性,以供PlatfromTransactionManager使用,
PlatfromTransactionManager是spring事务管理的核心接口。

```
public interface TransactionDefinition {
int getPropagationBehavior();//返回事务的传播行为。
int getIsolationLevel();//返回事务的隔离级别，事务管理器根据它来控制另外一个事务可以看到本事务内的哪些数据。
int getTimeout();//返回事务必须在多少秒内完成。
boolean isReadOnly();//事务是否只读，事务管理器能够根据这个返回值进行优化，确保事务是只读的。
}
```

1. TransactionDefinition接口中定义五个隔离级别：

ISOLATION_DEFAULT 这是一个PlatfromTransactionManager默认的隔离级别，使用数据库默认的事务隔离级别.另外四个与JDBC的隔离级别相对应；

ISOLATION_READ_UNCOMMITTED 这是事务最低的隔离级别，它充许别外一个事务可以看到这个事务未提交的数据。这种隔离级别会产生脏读，不可重复读和幻像读。

ISOLATION_READ_COMMITTED 保证一个事务修改的数据提交后才能被另外一个事务读取。另外一个事务不能读取该事务未提交的数据。这种事务隔离级别可以避免脏读出现，但是可能会出现不可重复读和幻像读。

ISOLATION_REPEATABLE_READ 这种事务隔离级别可以防止脏读，不可重复读。但是可能出现幻像读。它除了保证一个事务不能读取另一个事务未提交的数据外，还保证了避免下面的情况产生(不可重复读)。

ISOLATION_SERIALIZABLE 这是花费最高代价但是最可靠的事务隔离级别。事务被处理为顺序执行。除了防止脏读，不可重复读外，还避免了幻像读。

1： Dirty reads（脏读）。也就是说，比如事务A的未提交（还依然缓存）的数据被事务B读走，如果事务A失败回滚，会导致事务B所读取的的数据是错误的。
2： non-repeatable reads（数据不可重复读）。比如事务A中两处读取数据-total-的值。在第一读的时候，total是100，然后事务B就把total的数据改成200，事务A再读一次，结果就发现，total竟然就变成200了，造成事务A数据混乱。
3： phantom reads（幻象读数据），这个和non-repeatable reads相似，也是同一个事务中多次读不一致的问题。但是non-repeatable reads的不一致是因为他所要取的数据集被改变了（比如total的数据），但是phantom reads所要读的数据的不一致却不是他所要读的数据集改变，而是他的条件数据集改变。比如Select account.id where account.name=”ppgogo*”,第一次读去了6个符合条件的id，第二次读取的时候，由于事务b把一个帐号的名字由”dd”改成”ppgogo1″，结果取出来了7个数据。

2. 在TransactionDefinition接口中定义了七个事务传播行为：

（1）PROPAGATION_REQUIRED 如果存在一个事务，则支持当前事务。如果没有事务则开启一个新的事务。

Java代码：

```
1  //事务属性 PROPAGATION_REQUIRED
2  methodA{
3  ……
4  methodB();
5  ……
6  }
7
8  //事务属性 PROPAGATION_REQUIRED
9  methodB{
10    ……
11 }
```

使用spring声明式事务，spring使用AOP来支持声明式事务，会根据事务属性，自动在方法调用之前决定是否开启一个事务，并在方法执行之后决定事务提交或回滚事务。

单独调用methodB方法：

Java代码

```
1  main{
2
3  metodB();
4
5  }
```

相当于

Java代码

```
1  Main{
2
3  Connection con=null;
4
5  try{
6
7  con = getConnection();
8
```

```
con.setAutoCommit(false);

//方法调用

methodB();

//提交事务

con.commit();

}

Catch(RuntimeException ex){

            //回滚事务

con.rollback();

}

finally{

//释放资源

closeCon();

}

}
```

Spring保证在methodB方法中所有的调用都获得到一个相同的连接。在调用methodB时，没有一个存在的事务，所以获得一个新的连接，开启了一个新的事务。

单独调用MethodA时，在MethodA内又会调用MethodB.

执行效果相当于：

Java代码

```
         main{

Connection con = null;

try{

con = getConnection();

    methodA();

con.commit();
```

```
}

catch(RuntimeException ex){

con.rollback();

}

finally{

   closeCon();

}

}
```

调用MethodA时，环境中没有事务，所以开启一个新的事务.当在MethodA中调用MethodB时，环境中已经有了一个事务，所以methodB就加入当前事务。

（2）PROPAGATION_SUPPORTS 如果存在一个事务，支持当前事务。如果没有事务，则非事务的执行。但是对于事务同步的事务管理器，PROPAGATION_SUPPORTS与不使用事务有少许不同。

Java代码：

```
//事务属性 PROPAGATION_REQUIRED
methodA(){
  methodB();
}

//事务属性 PROPAGATION_SUPPORTS
methodB(){
  ……
}
```

单纯的调用methodB时，methodB方法是非事务的执行的。当调用methdA时,methodB则加入了methodA的事务中,事务地执行。

（3）PROPAGATION_MANDATORY 如果已经存在一个事务，支持当前事务。如果没有一个活动的事务，则抛出异常。

Java代码：

```
//事务属性 PROPAGATION_REQUIRED
    methodA(){
  methodB();
                }

    //事务属性 PROPAGATION_MANDATORY
    methodB(){
    ……
    }
```

当单独调用methodB时，因为当前没有一个活动的事务，则会抛出异常throw new IllegalTransactionStateException(“Transaction propagation ‘mandatory’ but no existing transaction found”);当调用methodA时，methodB则加入到methodA的事务中，事务地执行。

（4）PROPAGATION_REQUIRES_NEW 总是开启一个新的事务。如果一个事务已经存在，则将这个存在的事务挂起。

Java代码：

```
//事务属性 PROPAGATION_REQUIRED
methodA(){
   doSomeThingA();
methodB();
doSomeThingB();
}

//事务属性 PROPAGATION_REQUIRES_NEW
methodB(){
   ……
}
```

Java代码：

```
main(){
  methodA();
}
```

相当于

Java代码：

```
main(){
  TransactionManager tm = null;
try{
  //获得一个JTA事务管理器
    tm = getTransactionManager();
    tm.begin();//开启一个新的事务
    Transaction ts1 = tm.getTransaction();
    doSomeThing();
    tm.suspend();//挂起当前事务
    try{
      tm.begin();//重新开启第二个事务
      Transaction ts2 = tm.getTransaction();
      methodB();
      ts2.commit();//提交第二个事务
   }
  Catch(RunTimeException ex){
      ts2.rollback();//回滚第二个事务
  }
  finally{
     //释放资源
   }
```

```
    //methodB执行完后，复恢第一个事务
    tm.resume(ts1);
doSomeThingB();
    ts1.commit();//提交第一个事务
}
catch(RunTimeException ex){
   ts1.rollback();//回滚第一个事务
}
finally{
   //释放资源
}
}
```

在这里，我把ts1称为外层事务，ts2称为内层事务。从上面的代码可以看出，ts2与ts1是两个独立的事务，互不相干。Ts2是否成功并不依赖于ts1。如果methodA方法在调用methodB方法后的doSomeThingB方法失败了，而methodB方法所做的结果依然被提交。而除了methodB之外的其它代码导致的结果却被回滚了。使用PROPAGATION_REQUIRES_NEW,需要使用JtaTransactionManager作为事务管理器。

（5）PROPAGATION_NOT_SUPPORTED 总是非事务地执行，并挂起任何存在的事务。使用PROPAGATION_NOT_SUPPORTED,也需要使用JtaTransactionManager作为事务管理器。（代码示例同上，可同理推出）

（6）PROPAGATION_NEVER 总是非事务地执行，如果存在一个活动事务，则抛出异常；

（7）PROPAGATION_NESTED如果一个活动的事务存在，则运行在一个嵌套的事务中. 如果没有活动事务, 则按TransactionDefinition.PROPAGATION_REQUIRED 属性执行。这是一个嵌套事务,使用JDBC 3.0驱动时,仅仅支持DataSourceTransactionManager作为事务管理器。需要JDBC 驱动的java.sql.Savepoint类。有一些JTA的事务管理器实现可能也提供了同样的功能。使用PROPAGATION_NESTED，还需要把PlatformTransactionManager的nestedTransactionAllowed属性设为true;而nestedTransactionAllowed属性值默认为false;

Java代码：

```
//事务属性 PROPAGATION_REQUIRED
methodA(){
   doSomeThingA();
   methodB();
   doSomeThingB();
}

//事务属性 PROPAGATION_NESTED
methodB(){
  ……
}
```

如果单独调用methodB方法，则按REQUIRED属性执行。如果调用methodA方法，相当于下面的效果：

Java代码：

```
main(){
Connection con = null;
Savepoint savepoint = null;
try{
   con = getConnection();
```

```
    con.setAutoCommit(false);
    doSomeThingA();
    savepoint = con2.setSavepoint();
    try{
        methodB();
    }catch(RuntimeException ex){
       con.rollback(savepoint);
    }
    finally{
      //释放资源
  }

    doSomeThingB();
    con.commit();
}
catch(RuntimeException ex){
  con.rollback();
}
finally{
   //释放资源
}
}
```

当methodB方法调用之前，调用setSavepoint方法，保存当前的状态到savepoint。如果methodB方法调用失败，则恢复到之前保存的状态。但是需要注意的是，这时的事务并没有进行提交，如果后续的代码(doSomeThingB()方法)调用失败，则回滚包括methodB方法的所有操作。

嵌套事务一个非常重要的概念就是内层事务依赖于外层事务。外层事务失败时，会回滚内层事务所做的动作。而内层事务操作失败并不会引起外层事务的回滚。

PROPAGATION_NESTED 与PROPAGATION_REQUIRES_NEW的区别:它们非常类似,都像一个嵌套事务，如果不存在一个活动的事务，都会开启一个新的事务。使用PROPAGATION_REQUIRES_NEW时，内层事务与外层事务就像两个独立的事务一样，一旦内层事务进行了提交后，外层事务不能对其进行回滚。两个事务互不影响。两个事务不是一个真正的嵌套事务。同时它需要JTA事务管理器的支持。

使用PROPAGATION_NESTED时，外层事务的回滚可以引起内层事务的回滚。而内层事务的异常并不会导致外层事务的回滚，它是一个真正的嵌套事务。DataSourceTransactionManager使用savepoint支持PROPAGATION_NESTED时，需要JDBC 3.0以上驱动及1.4以上的JDK版本支持。其它的JTA TrasactionManager实现可能有不同的支持方式。

PROPAGATION_REQUIRES_NEW 启动一个新的, 不依赖于环境的 “内部” 事务. 这个事务将被完全 commited 或 rolled back 而不依赖于外部事务, 它拥有自己的隔离范围, 自己的锁, 等等. 当内部事务开始执行时, 外部事务将被挂起, 内务事务结束时, 外部事务将继续执行。

另一方面, PROPAGATION_NESTED 开始一个 “嵌套的” 事务，它是已经存在事务的一个真正的子事务. 潜套事务开始执行时，它将取得一个 savepoint. 如果这个嵌套事务失败, 我们将回滚到此 savepoint. 潜套事务是外部事务的一部分, 只有外部事务结束后它才会被提交。

由此可见, PROPAGATION_REQUIRES_NEW 和 PROPAGATION_NESTED 的最大区别在于, PROPAGATION_REQUIRES_NEW 完全是一个新的事务, 而 PROPAGATION_NESTED 则是外部事务的子事务, 如果外部事务 commit, 潜套事务也会被 commit, 这个规则同样适用于 roll back.
PROPAGATION_REQUIRED应该是我们首先的事务传播行为。它能够满足我们大多数的事务需求。

标签: Spring, 事务

好文要顶 关注我 收藏该文

CSniper
关注 - 8
粉丝 - 14
+加关注

0 0

« 上一篇：Java开发常用的在线工具
» 下一篇：MySQL 加锁处理分析

posted @ 2016-05-28 00:49 CSniper 阅读(7539) 评论(0) 编辑 收藏
刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。