第 19 章 Spring事务王国的架构

19

本章内容

- 统一中原的过程
- 和平年代

Spring的事务框架将开发过程中事务管理相关的关注点进行适当的分离，并对这些关注点进行合理的抽象，最终打造了一套使用方便，却功能强大的事务管理“利器”。通过Spring的事务框架，我们可以按照统一的编程模型来进行事务编程，却不用关心所使用的数据访问技术以及具体要访问什么类型的事务资源。并且，Spring的事务框架与Spring提供的数据访问支持可以紧密结合，更是让我们在事务管理与数据访问之间游刃有余。而最主要的是，结合Spring的AOP框架，Spring的事务框架为我们带来了原来只有CMT才有的声明式事务管理的特殊待遇，却无需绑定到任何的应用服务器上。

其他溢美之词咱就先放一边，还是赶快进入正题吧！

Spring的事务框架设计理念的基本原则是：让事务管理的关注点与数据访问关注点相分离。

- 当在业务层使用事务的抽象API进行事务界定的时候，不需要关心事务将要加诸于上的事务资源是什么，对不同的事务资源的管理将由相应的框架实现类来操心。
- 当在数据访问层对可能参与事务的数据资源进行访问的时候，只需要使用相应的数据访问API进行数据访问，不需要关心当前的事务资源如何参与事务或者是否需要参与事务。这同样将由事务框架类来打理。

在以上两个关注点被清晰地分离出来之后，对于我们开发人员来说，唯一需要关心的，就是通过抽象后的事务管理API对当前事务进行界定而已，如代码清单19-1所示。

代码清单19-1　使用事务管理抽象API进行事务界定的代码示例

```
public class FooService
{
    private PlatformTransactionManager transactionManager;

    public void serviceMethod()
    {
        TransactionDefinition definition = ...;
        TransactionStatus txStatus = getTransactionManager().getTransaction(definition);
        try
        {
            // dao1.doDataAccess();
            // dao2.doDataAccess();
            // ...
        }
        catch(DataAccessException e)
        {
```

```
            getTransactionManager().rollback(txStatus);
            throw e;
        }
        catch(OtherNecessaryException e)
        {
            getTransactionManager().rollback(txStatus);
            throw e;
        }
        getTransactionManager().commit(txStatus);
    }

    public PlatformTransactionManager getTransactionManager() {
        return transactionManager;
    }

    public void setTransactionManager(PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }
}
```

不管数据访问方式如何变换，事务管理实现也可以岿然不动。只要有了这样一个统一的事务管理编程模型，剩下的声明式事务管理也就很自然地成为锦上添花之作了！从此之后，事务管理就是事务管理，数据访问只关心数据访问，再也不用因为它们之间的纠缠而烦恼。

19.1 统一中原的过程

org.springframework.transaction.PlatformTransactionManager是Spring事务抽象架构的核心接口，它的主要作用是为应用程序提供事务界定的统一方式。既然事务界定的需要很简单，那么PlatformTransactionManager的定义看起来也不会过于复杂，如下所示：

```
public interface PlatformTransactionManager {
    TransactionStatus getTransaction(TransactionDefinition definition) throws ➥
    Transaction Exception;
    void commit(TransactionStatus status) throws TransactionException;
    void rollback(TransactionStatus status) throws TransactionException;
}
```

PlatformTransactionManager是整个事务抽象策略的顶层接口，它就好像我们的战略蓝图，而战略的具体实施则将由相应的PlatformTransactionManager实现类来执行。

Spring的事务框架针对不同的数据访问方式以及全局事务场景，提供了相应的PlatformTransactionManager实现类。在每个实现类的职责完成之后，Spring事务框架的“统一大业”就完成了。在深入了解各个PlatformTransactionManager实现类的奥秘之前，不妨先考虑一下，如果让我们来实现一个PlatformTransactionManager，要如何去做？

不妨先以针对JDBC数据访问方式的局部事务管理为例。对于层次划分清晰的应用来说，我们通常都是将事务管理放在Service层，而将数据访问逻辑放在DAO层。这样做的目的是，可以不用因为将事务管理代码放在DAO层，而降低数据访问逻辑的重用性，也可以在Service层根据相应逻辑，来决定提交或者回滚事务。一般的Service对象可能需要在同一个业务方法中调用多个数据访问对象的方法，类似于图19-1这样的情况。

因为JDBC的局部事务控制是由同一个java.sql.Connection来完成的，所以要保证两个DAO的数据访问方法处于一个事务中，我们就得保证它们使用的是同一个java.sql.Connection。要做到这一点，通常会采用称为connection-passing的方式，即为同一个事务中的各个dao的数据访问方法传递当

前事务对应的同一个java.sql.Connection。这样，我们的业务方法以及数据访问方法都得做一定的修改，如图19-2所示。

```
public void serviceMethod()
{
        Object txObject=transactionManager.beginTransaction();
        ...
        dao1.doDataAccess();
        dao2.doDataAccess();
        ...
        transactionManager.commitTransaction(txObject);
}
```

图19-1　普通情况下的事务管理代码

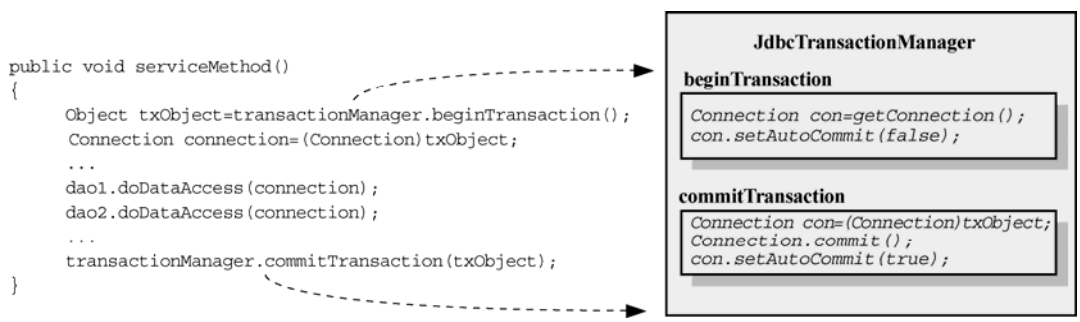

图19-2　connection-passing方式的事务管理代码

我们只要把java.sql.Connection的获取并设置Autocommit状态的代码，以及使用java.sql.Connection提交事务的代码，重构到原来的开启事务以及提交事务的方法中，针对JDBC的局部事务管理的整合看起来离成功也就是咫尺之遥了。不过，这看起来的咫尺之遥，实际上却依然遥远。

使用这种方式，最致命的一个问题就是，不但事务管理代码无法摆脱java.sql.Connection的纠缠，而且数据访问对象的定义要绑定到具体的数据访问技术上来。现在是使用JDBC进行数据访问，我们要在数据访问方法中声明对java.sql.Connection的依赖，那要是使用Hibernate的话，是不是又要声明对Session的依赖呢？显然，这样的做法是不可行的。不过好消息是，传递Connection的理念是对的，只不过，在具体实施过程中，我们所采用的方法有些不对头。

要传递java.sql.Connection，我们可以将整个事务对应的java.sql.Connection实例放到统一的一个地方去，无论是谁，要使用该资源，都从这一个地方来获取，这样就解除了事务管理代码和数据访问代码之间通过java.sql.Connection的“直接”耦合。具体点儿说就是，我们在事务开始之前取得一个java.sql.Connection，然后将这个Connection绑定到当前的调用线程。之后，数据访问对象在使用Connection进行数据访问的时候，就可以从当前线程上获得这个事务开始的时候绑定的Connection实例。当所有的数据访问对象全部使用这个绑定到当前线程的Connection完成了数据访问工作时，我们就使用这个Connection实例提交或者回滚事务，然后解除它到当前线程的绑定。整个过程如图19-3所示。

这时的java.sql.Connection就像那大河上的一条船，从启航（事务开始）到航程结束（事务完

成）这整个期间，大河沿岸都可以与该船打交道。而至于说是发“木船”还是发“铁轮”，那要根据情况来决定了：发JDBC的船，那就是Connection；发Hibernate的船，那就是Session……

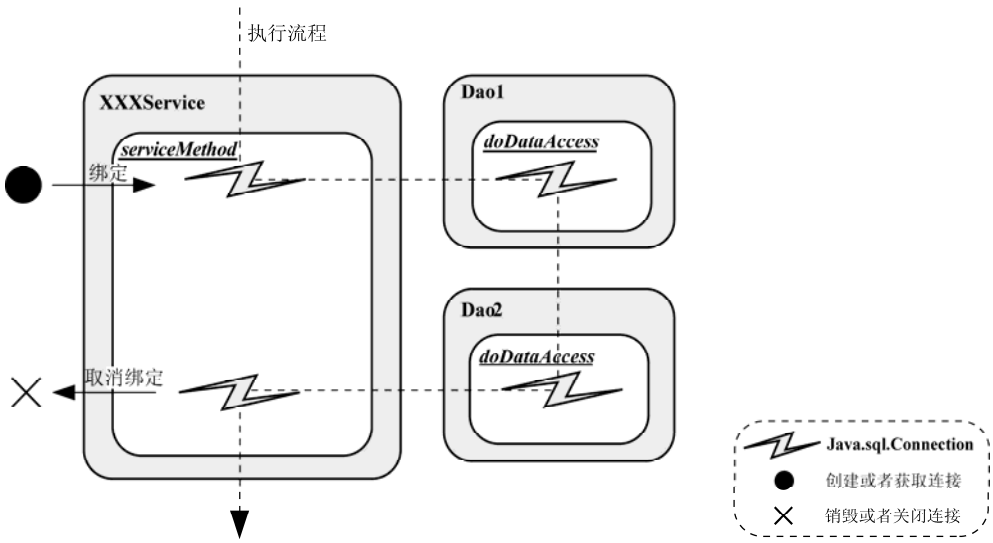

图19-3 java.sql.Connection绑定到线程示意图

假设TransactionResourceManager就是存放java.sql.Connection（或者其他事务资源）的地方，那么，它看起来将如代码清单19-2所示（过多的逻辑检验代码略去）。

代码清单19-2 TransactionResourceManager原型定义代码示例

```
public class TransactionResourceManager
{
    private static ThreadLocal resources = new ThreadLocal();

    public static Object getResource()
    {
        return resources.get();
    }

    public static void bindResource(Object resource)
    {
        resources.set(resource);
    }

    public static Object unbindResource()
    {
        Object res = getResource();
        resources.set(null);
        return res;
    }
}
```

对于我们要实现的针对JDBC的PlatformTransactionManager，只需要在事务开始的时候，通过我们的TransactionResourceManager将java.sql.Connection绑定到线程，然后在事务结束的时候解除绑定即可（原型代码参照代码清单19-3）。

代码清单19-3 面向JDBC的PlatformTransactionManager原型实现代码示例

```
public class JdbcTransactionManager implements PlatformTransactionManager
    {
    private DataSource dataSource;

    public JdbcTransactionManager(DataSource dataSource)
    {
        this.dataSource = dataSource;
    }

    public TransactionStatus getTransaction(TransactionDefinition definition) ➥
    throws TransactionException {
            Connection connection;
        try
        {
            connection = dataSource.getConnection();
            TransactionResourceManager.bindResource(connection);
            return new ➥
            DefaultTransactionStatus(connection,true,true,false,true,null);
        }
        catch (SQLException e)
        {
            throw new ➥
            CannotCreateTransactionException("can't get connection for tx",e);
        }
    }

    public void rollback(TransactionStatus txStatus) throws TransactionException {
        Connection connection = ➥
        (Connection)TransactionResourceManager.unbindResource();
        try
        {
            connection.rollback();
        }
        catch (SQLException e)
        {
            throw new ➥
            UnexpectedRollbackException("rollback failed with SQLException",e);
        }
        finally
        {
            try {
                connection.close();
            } catch (SQLException e) {
                // 记录异常信息, 但通常不会有进一步有效的处理
            }
        }
    }
    public void commit(TransactionStatus txStatus) throws TransactionException {
        Connection connection = ➥
        (Connection)TransactionResourceManager.unbindResource();
        try
        {
            connection.commit();
        }
        catch (SQLException e)
        {
            throw new ➥
            TransactionSystemException("commit failed with SQLException",e);
        }
```

```
            finally
            {
                try {
                    connection.close();
                } catch (SQLException e) {
                    // 记录异常信息，但通常不会有进一步有效的处理
                }
            }
        }
    }
```

因为Connection在事务开始和结束期间都可以通过我们的TransactionResourceManager获得，所以所有的DAO层数据访问对象在使用JDBC进行数据访问的时候，就可以直接从TransactionResourceManager获得数据库连接并进行数据访问。这样就可以保证在整个事务期间，所有的数据访问对象对应的是同一个Connection，如下所示：

```
public class FooJdbcDao implements IDao{
    public void doDataAccess(){
        Connection con = (Connection)TransactionResourceManager.getResource();
        // ...
    }
}
```

至此，我们完成了PlatformTransactionManager的具体实现类，并解除了它与相应数据访问对象之间通过java.sql.Connection的直接耦合。进行事务控制的时候，我们只需要为Service对象提供相应的PlatformTransactionManager实现类，Service对象中的事务管理功能就算大功告成了，而不需要关心到底对应的是什么样的事务资源，甚至什么样的数据访问方式。

当然，为了便于你理解Spring抽象层的实现原理，以上的代码实例都是简化后的模型，所以，不要试图将它们应用于生产环境。原型代码永远都是原型代码，要做的事情还有许多，如下所述。

(1) 如何保证PlatformTransactionManager的相应方法以正确的顺序被调用？如果哪个方法没有被正确调用，也会造成资源泄漏以及事务管理代码混乱的问题。在稍后介绍使用Spring进行编程事务管理的时候，你将看到Spring是如何解决这个问题的。

(2) 数据访问对象的接口定义不会因为最初的connection-passing方式而改变契约了，但是，现在却要强制每个数据访问对象使用TransactionResourceManager来获取数据资源接口。另外，如果当前数据访问对象对应的数据方法不想参与跨越多个数据操作的事务的话，甚至于不想（或不能）使用类似的事务管理支持，是否就意味着无法获得connection进行数据访问了呢？

不知道你是否还记得我们在介绍Spring的数据访问相关内容的时候，曾经提到的org.springframework.jdbc.datasource.DataSourceUtils工具类。当时我们只是强调了DataSourceUtils提供的异常转译能力。实际上，DataSourceUtils最主要的工作是对connection的管理，DataSourceUtils会从类似TransactionResourceManager的类（Spring中对应org.springframework.transaction.support.TransactionSynchronizationManager）那里获取Connection资源。如果当前线程之前没有绑定任何connection，那么它就通过数据访问对象的DataSource引用获取新的connection，否则就使用绑定的那个connection。这就是为什么要强调，当我们要使用Spring提供的事务支持的时候，必须通过DataSourceUtils来获取连接。因为它提供了Spring事务管理框架在数据访问层需要提供的基础设施中不可或缺的一部分，而JdbcTemplate等类内部已经使用DataSourceUtils来管理连接了，所以，我们不用操心这些细节。从这里，我们也应可以看出，Spring的事务管理与它的数据访问框架是紧密结合的。

注意 对应Hibernate的SessionFactoryUtils，对应JDO的PersistenceManagerFactoryUtils以及对应其他数据访问技术的Utils类，它们的作用与DataSourceUtils是相似的。除了提供异常转译功能，它们更多地被用于数据访问资源的管理工作，以配合对应的PlatformTransactionManager实现类进行事务管理。

实际上，Spring在实现针对各种数据访问技术的PlatformTransactionManager的时候，要考虑很多东西，不像原型以及提出的几个问题所展示的那么简单。不过，各个实现类的基本思路与原型所展示的是基本吻合的。当我们了解了针对JDBC的PlatformTransactionManager是如何实现的时候，其他的实现类基本上就是平推了。

19.2 和平年代

Spring的事务抽象包括3个主要接口，即PlatformTransactionManager、TransactionDefinition以及TransactionStatus，它们之间的关系如图19-4所示。

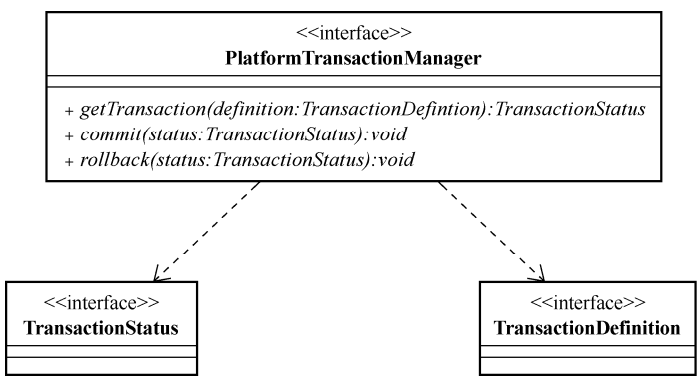

图19-4 Spring事务抽象接口关系图

这3个接口以org.springframework.transaction.PlatformTransactionManager为中心，互为犄角，多少有点儿“晋西北铁三角”的味道。org.springframework.transaction.PlatformTransactionManager负责界定事务边界。org.springframework.transaction.TransactionDefinition负责定义事务相关属性，包括隔离级别、传播行为等。org.springframework.transaction.PlatformTransactionManager将参照org.springframework.transaction.TransactionDefinition的属性定义来开启相关事务。事务开启之后到事务结束期间的事务状态由org.springframework.transaction.TransactionStatus负责，我们也可以通过org.springframework.transaction.TransactionStatus对事务进行有限的控制。

19.2.1 TransactionDefinition

1. TransactionDefinition简介

org.springframework.transaction.TransactionDefinition主要定义了有哪些事务属性可以指定，这包括：

- 事务的隔离（Isolation）级别
- 事务的传播行为（Propagation Behavior）
- 事务的超时时间（Timeout）
- 是否为只读（ReadOnly）事务

TransactionDefinition内定义了如下5个常量用于标志可供选择的隔离级别。

- ISOLATION_DEFAULT。如果指定隔离级别为ISOLATION_DEFAULT，则表示使用数据库默认的隔离级别，通常情况下是Read Committed。

- ISOLATION_READ_UNCOMMITTED。对应Read Uncommitted隔离级别，无法避免脏读，不可重复读和幻读。
- ISOLATION_READ_COMMITTED。对应Read Committed隔离级别，可以避免脏读，但无法避免不可重复读和幻读。
- ISOLATION_REPEATABLE_READ。对应Repeatable read隔离级别，可以避免脏读和不可重复读，但不能避免幻读。
- ISOLATION_SERIALIZABLE。对应Serializable隔离级别，可以避免所有的脏读，不可重复读以及幻读，但并发性效率最低。

事务的传播行为（Propagation Behavior）我们之前没有提到，如果你之前接触过EJB的CMT的话，对它应该也不会陌生。事务的传播行为表示整个事务处理过程所跨越的业务对象，将以什么样的行为参与事务（我们将在声明式事务中更多地依赖于该属性）。比如，当有FoobarService调用FooService和BarService两个方法的时候，FooService的业务方法和BarService的业务方法可以指定它们各自的事务传播行为（如图19-5所示）。

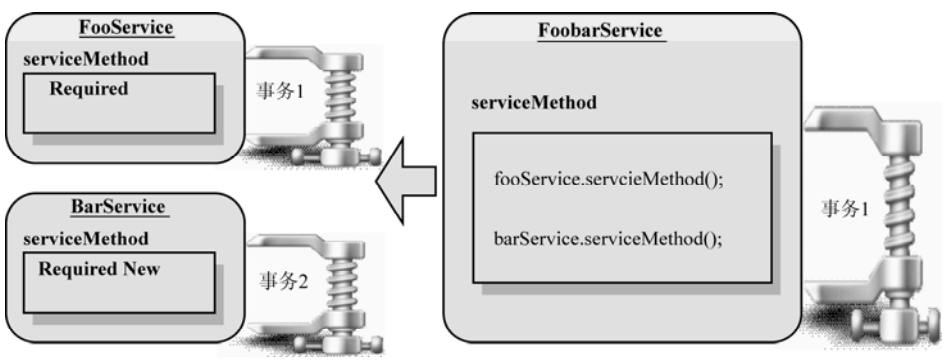

图19-5 业务方法的传播行为

FooService的业务方法的传播行为被我们指定为Required，表示如果当前存在事务的话，则加入当前事务。因为FoobarService在调用FooService的业务方法的时候已经启动了一个事务，所以，FooSerivce的业务方法会直接加入FoobarService启动的事务1中。BarService的业务方法的传播行为被指定为Required New，表示无论当前是否存在事务，都需要为其重新启动一个事务，所以，它使用的是自己启动的事务2。

针对事务的传播行为，TransactionDefinition提供了以下几种选择，除了PROPAGATION_NESTED是Spring特有的外，其他传播行为的语义与CMT基本相同。

- PROPAGATION_REQUIRED。如果当前存在一个事务，则加入当前事务。如果不存在任何事务，则创建一个新的事务。总之，要至少保证在一个事务中运行。PROPAGATION_REQUIRED通常作为默认的事务传播行为。
- PROPAGATION_SUPPORTS。如果当前存在一个事务，则加入当前事务。如果当前不存在事务，则直接执行。对于一些查询方法来说，PROPAGATION_SUPPORTS通常是比较合适的传播行为选择。如果当前方法直接执行，那么不需要事务的支持。如果当前方法被其他方法调用，而其他方法启动了一个事务，使用PROPAGATION_SUPPORTS可以保证当前方法能够加入当前事务，并洞察当前事务对数据资源所做的更新。比如，A.service()会首先更新数据库，然后调用

B.service()进行查询，那么，如果B.service()是PROPAGATION_SUPPORTS的传播行为，就可以读取A.service()之前所做的最新更新结果（如图19-6所示）。而如果使用稍后所提到的PROPAGATION_NOT_SUPPORTED，则B.service()将无法读取最新的更新结果，因为A.service()的事务在这时还没有提交（除非隔离级别是Read Uncommitted）。

- PROPAGATION_MANDATORY。PROPAGATION_MANDATORY强制要求当前存在一个事务，如果不存在，则抛出异常。如果某个方法需要事务支持，但自身又不管理事务提交或者回滚，那么比较适合使用PROPAGATION_MANDATORY。可以参照*Java Transaction Design Strategies*一书中对REQUIRED和MANDATORY两种传播行为的比较，来更深入地了解PROPAGATION_ MANDATORY可能的应用场景。

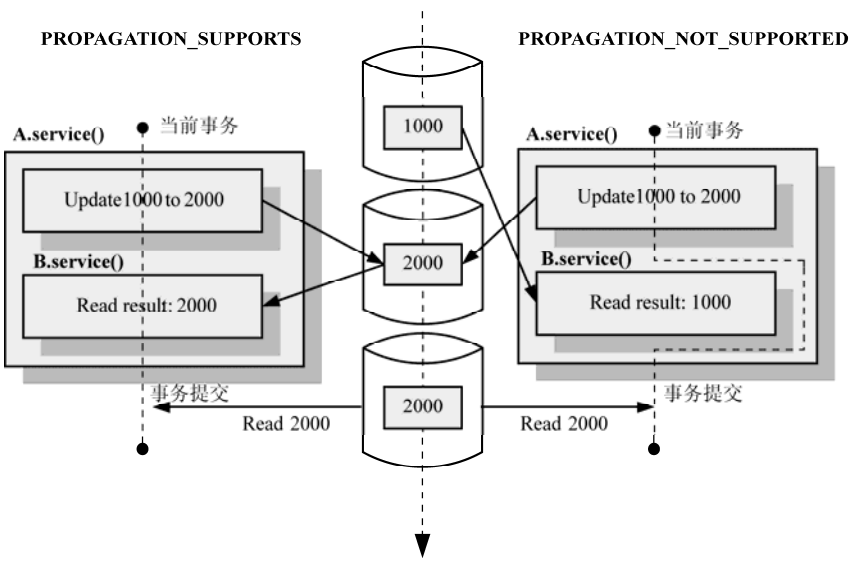

图19-6　PROPAGATION_SUPPORTS可能场景

- PROPAGATION_REQUIRES_NEW。不管当前是否存在事务，都会创建新的事务。如果当前存在事务，会将当前的事务挂起（Suspend）。如果某个业务对象所做的事情不想影响到外层事务，PROPAGATION_REQUIRES_NEW应该是合适的选择。比如，假设当前的业务方法需要向数据库中更新某些日志信息，但即使这些日志信息更新失败，我们也不想因为该业务方法的事务回滚，而影响到外层事务的成功提交。因为这种情况下，当前业务方法的事务成功与否对外层事务来说是无关紧要的。
- PROPAGATION_NOT_SUPPORTED。不支持当前事务，而是在没有事务的情况下执行。如果当前存在事务的话，当前事务原则上将被挂起（Suspend），但这要看对应的PlatformTransaction-Manager实现类是否支持事务的挂起。更多情况请参照TransactionDefinition的Javadoc文档。PROPAGATION_NOT_SUPPORTED与PROPAGATION_SUPPORTS之间的区别，可以参照PROPAGATION_SUPPORTS部分的实例内容。
- PROPAGATION_NEVER。永远不需要当前存在事务，如果存在当前事务，则抛出异常。
- PROPAGATION_NESTED。如果存在当前事务，则在当前事务的一个嵌套事务中执行，否则与PRO-

PAGATION_REQUIRED的行为类似，即创建新的事务，在新创建的事务中执行。PROPAGATION_NESTED粗看起来好像与PROPAGATION_REQUIRES_NEW的行为类似，实际上二者是有差别的。PROPAGATION_REQUIRES_NEW创建的新事务与外层事务属于同一个“档次”，即二者的地位是相同的。当新创建的事务运行的时候，外层事务将被暂时挂起。而PROPAGATION_NESTED创建的嵌套事务则不然，它是寄生于当前外层事务的，它的地位比当前外层事务的地位要小一号。当内部嵌套事务运行的时候，外层事务也是处于active状态，图19-7演示了PROPAGATION_REQUIRES_NEW和PROPAGATION_NESTED中涉及的多个事务相互之间的地位。

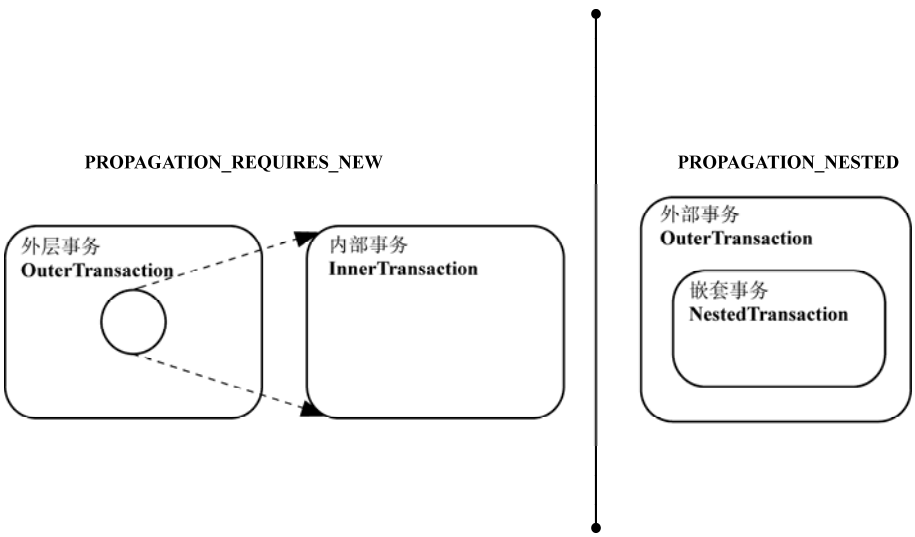

图19-7 PROPAGATION_REQUIRES_NEW与PROPAGATION_NESTED创建的事务的区别

也就是说，虽然PROPAGATION_REQUIRES_NEW新创建的事务是在当前外层事务内执行，但新创建的事务是独立于当前外层事务而存在的，二者拥有各自独立的状态而互不干扰。而PROPAGATION_NESTED创建的事务属于当前外层事务的内部子事务（Sub-transaction），内部子事务的处理内容属于当前外层事务的一部分，不能独立于外层事务而存在，并且与外层事务共有事务状态。我想这也就是为什么称其为内部嵌套事务的原因。

PROPAGATION_NESTED可能的应用场景在于，你可以将一个大的事务划分为多个小的事务来处理，并且外层事务可以根据各个内部嵌套事务的执行结果，来选择不同的执行流程。比如，某个业务对象的业务方法A.service()，可能调用其他业务方法B.service()向数据库中插入一批业务数据，但当插入数据的业务方法出现错误的时候（比如主键冲突），我们可以在当前事务中捕捉前一个方法抛出的异常，然后选择另一个更新数据的业务方法C.service()来执行。这时，我们就可以把B.service()和C.serivce()方法的传播行为指定为PROPAGATION_NESTED[①]。如果用伪代码演示的话，看起来如代码清单19-4所示。

① 当然，如果只是检测单条数据插入的主键冲突，然后改为更新数据的话，更多时候，我们会直接在一个数据访问方法中解决。

代码清单19-4 嵌套事务应用代码示例

```
/**
 * PROPAGATION_REQUIRED
 */
A.service()
{
    try
    {
        // PROPAGATION_NESTED
        B.service();
    }
    catch(Exception e)
    {
        // PROPAGATION_NESTED
        C.service();
    }
}
```

不过，并非所有的PlatformTransactionManager实现都支持PROPAGATION_NESTED类型的传播行为。现在只有org.springframework.jdbc.datasource.DataSourceTransactionManager在使用JDBC 3.0数据库驱动的情况下才支持（当然，数据库和相应的驱动程序也需要提供支持）。另外，某些JtaTransactionManager也可能提供支持，但是JTA规范并没有要求提供对嵌套事务的支持。

对TransactionDefinition所提供的这几个传播行为选项的使用，最好是建立在充分理解的基础上。当然，除非特殊的场景，通常情况下，PROPAGATION_REQUIRED将是我们最常用的选择。

TransactionDefinition提供了TIMEOUT_DEFAULT常量定义，用来指定事务的超时时间。TIMEOUT_DEFAULT默认值为–1，这会采用当前事务系统默认的超时时间。我们可以通过TransactionDefinition的具体实现类提供自定义的事务超时时间。

TransactionDefinition提供的最后一个重要信息就是将要创建的是否是一个只读（ReadOnly）的事务。如果需要创建一个只读的事务的话，可以通过TransactionDefinition的相关实现类进行设置。只读的事务仅仅是给相应的ResourceManager提供一种优化的提示，但最终是否提供优化，则由具体的ResourceManager来决定。对于一些查询来说，我们通常会希望它们采用只读事务。

2. TransactionDefinition相关实现

TransactionDefinition只是一个接口定义，要为PlatformTransactionManager创建事务提供信息，需要有相应的实现类提供支持。虽然TransactionDefinition的相关实现类不多，但为了便于理解，我们依然将它们划分为“两派”，如图19-8所示。

我们将TransactionDefinition的相关实现类按照编程式事务场景和声明式事务场景划分为两个分支。这只是出于每个类在相应场景中出现的频率这一因素考虑的，而不是说声明式事务场景的实现类不能在编程式事务场景中使用。

org.springframework.transaction.support.DefaultTransactionDefinition 是 TransactionDefinition接口的默认实现类，它提供了各事务属性的默认值，并且通过它的setter方法，我们可以更改这些默认设置。这些默认值包括：

- propagationBehavior = PROPAGATION_REQUIRED
- isolationLevel = ISOLATION_DEFAULT
- timeout = TIMEOUT_DEFAULT
- readOnly = false

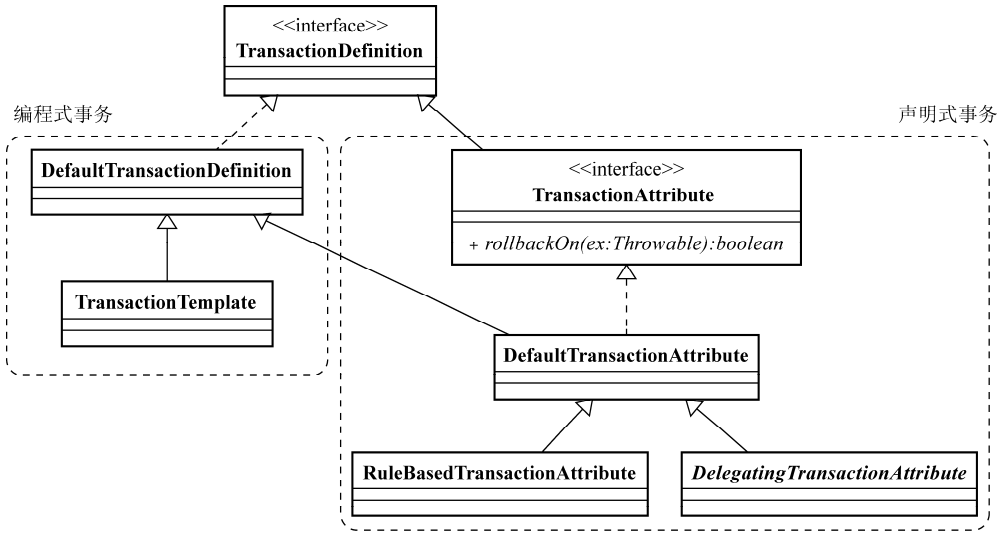

图19-8 TransactionDefinition继承层次图

org.springframework.transaction.support.TransactionTemplate是Spring提供的进行编程式事务管理的模板方法类（我们将稍后提到该类的使用），它直接继承了DefaultTransactionDefinition。所以，我们在使用TransactionTemplate的时候就可以直接通过TransactionTemplate本身提供事务控制属性。

org.springframework.transaction.interceptor.TransactionAttribute是继承自TransactionDefinition的接口定义，主要面向使用Spring AOP进行声明式事务管理的场合。它在TransactionDefinition定义的基础上添加了一个rollbackOn方法，如下所示：

```
boolean rollbackOn(Throwable ex);
```

这样，我们可以通过声明的方式指定业务方法在抛出哪些的异常的情况下可以回滚事务。

TransactionAttribute的默认实现类是DefaultTransactionAttribute，它同时继承了DefaultTransactionDefinition。在DefaultTransactionDefinition的基础上增加了rollbackOn的实现，DefaultTransactionAttribute的实现指定了，当异常类型为unchecked exception的情况下将回滚事务。

DefaultTransactionAttribute下有两个实现类，即RuleBasedTransactionAttribute和DelegatingTransactionAttribute。RuleBasedTransactionAttribute允许我们同时指定多个回滚规则，这些规则以包含org.springframework.transaction.interceptor.RollbackRuleAttribute或者org.springframework.transaction.interceptor.NoRollbackRuleAttribute的List形式提供。RuleBasedTransactionAttribute的rollbackOn将使用传入的异常类型与这些回滚规则进行匹配，然后再决定是否要回滚事务。

DelegatingTransactionAttribute是抽象类，它存在的目的就是被子类化，DelegatingTransactionAttribute会将所有方法调用委派给另一个具体的TransactionAttribute实现类，比如DefaultTransactionAttribute或者RuleBasedTransactionAttribute。不过，除非不是简单的直接委派（什么附加逻辑都不添加），否则，实现一个DelegatingTransactionAttribute是没有任

何意义的。

19.2.2 TransactionStatus

org.springframework.transaction.TransactionStatus接口定义表示整个事务处理过程中的事务状态，更多时候，我们将在编程式事务中使用该接口。

在事务处理过程中，我们可以使用TransactionStatus进行如下工作。

- 使用TransactionStatus提供的相应方法查询事务状态。
- 通过setRollbackOnly()方法标记当前事务以使其回滚。
- 如果相应的PlatformTransactionManager支持Savepoint，可以通过TransactionStatus在当前事务中创建内部嵌套事务。

在稍后将为你介绍如何使用Spring进行编程式事务管理的部分，可以更直观地了解这些。TransactionStatus的实现层次比较简单，见图19-9。

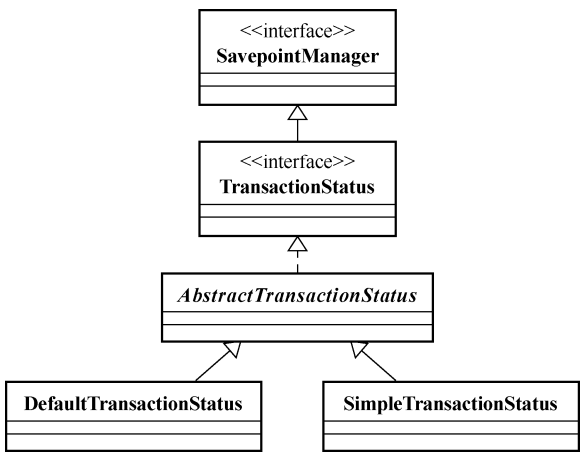

图19-9 TransactionStatus继承层次

org.springframework.transaction.SavepointManager是在JDBC 3.0的基础上，对Savepoint的支持提供的抽象。通过继承SavepointManager，TransactionStatus获得可以管理Savepoint的能力，从而支持创建内部嵌套事务。

org.springframework.transaction.support.AbstractTransactionStatus 是 TransactionStatus的抽象类实现，主要为其他实现子类提供一些“公共设施”。它下面主要有两个子类，DefaultTransactionStatus和SimpleTransactionStatus。其中，DefaultTransactionStatus是Spring事务框架内部使用的主要TransactionStatus实现类。Spring事务框架内的各个TransactionManager的实现，大都借助于DefaultTransactionStatus来记载事务状态信息。SimpleTransactionStatus在Spring框架内部的实现中没有使用到，目前来看，主要用于测试目的。

19.2.3 PlatformTransactionManager

PlatformTransactionManager是Spring事务抽象框架的核心组件，我们之前已经提过了它的定义以及作用，所以，这里我们将更多地关注整个PlatformTransactionManager的层次体系，以及针对不同数据访问技术的实现类。

PlatformTransactionManager的整个抽象体系基于Strategy模式，由PlatformTransactionManager对事务界定进行统一抽象，而具体的界定策略的实现则交由具体的实现类。下面我们先来看一下有哪些可供我们使用的实现类。

1. PlatformTransactionManager实现类概览

PlatformTransactionManager的实现类可以划分为面向局部事务和面向全局事务两个分支。

面向局部事务的PlatformTransactionManager实现类。Spring为各种数据访问技术提供了现成的PlatformTransactionManager实现支持。表19-1给出了各种数据访问技术与它们对应的实现类的关

系。

表19-1 数据访问技术与PlatformTransactionManager实现类对应关系

数据访问技术	PlatformTransactionManager实现类
JDBC/iBATIS	DataSourceTransactionManager
Hibernate	HibernateTransactionManager
JDO	JdoTransactionManager
JPA（Java Persistence API）	JpaTransactionManager
TopLink	TopLinkTransactionManager
JMS	JmsTransactionManager
JCA Local Transaction	CciLocalTransactionManager

在这些实现类中，CciLocalTransactionManager可能是比较少见的实现，CCI是Common Client Interface的缩写。CciLocalTransactionManager主要是面向JCA的局部事务，本书不打算对JCA的集成做过多的阐述。如果你在实际项目中需要使用到JCA进行EIS（Enterprise Information System）系统集成，那么可以从Spring的参考文档获得使用Spring提供的JCA集成支持的相关信息。

有了这些实现类，我们在使用Spring的事务抽象框架进行事务管理的时候，只需要根据当前使用的数据访问技术，选择对应的PlatformTransactionManager实现类即可。

提示 如果我们的应用程序需要同时使用Hibernate以及JDBC（或者iBATIS）进行数据访问，那么可以使用HibernateTransactionManager对基于Hibernate和JDBC（或者iBATIS）的事务进行统一管理，只要Hibernate的SessionFactory和JDBC（或者iBATIS）引用的是同一个DataSource就行。能猜到为什么吗？

面向全局事务的PlatformTransactionManager实现类。org.springframework.transaction.jta.JtaTransactionManager是Spring提供的支持分布式事务的PlatformTransac- tionManager实现。直接使用JTA规范接口进行分布式事务管理有以下几个问题。

- UserTransaction接口使用复杂不说（一长串的异常处理我们之前也见过了），它所公开的事务管理能力有限，对于事务的挂起（Suspend）以及恢复（Resume）操作，只有JTA的TransactionManager才支持。
- JTA规范并没有明确要求对TransactionManager的支持，这就造成虽然当下各个JTA提供商提供了TransactionManager的实现，但在应用服务器中公开的位置各有差异。为了进一步支持REQUIRES_NEW和NOT_SUPPORTED之类需要事务挂起以及恢复操作的事务传播行为，我们需要通过不同的方式来获取不同JTA提供商公开的TransactionManager实现。

鉴于此，JtaTransactionManager对各种JTA实现提供的分布式事务支持进行了统一封装，只不过它的所有的事务管理操作，最终都会委派给具体的JTA实现来完成。

对于典型的基于JTA的分布式事务管理，我们直接使用JtaTransactionManager就可以了。但某些时候，如果需要使用到各JTA产品的TransactionManager的特性，我们就可以为JtaTransactionManager注入这些JTA产品的javax.transaction.TransactionManager的实现。而至于说我们是通过应用服务器获取该TransactionManager，还是直接使用本地定义的TransactionManager（比如JOTM或者Atomikos等独立JTA实现产品的TransactionManager），则需要完全根

据当时的场景来决定了。能够为JtaTransactionManager提供具体的TransactionManager实现，为我们扩展JtaTransactionManager提供了很好的一个切入点。

JtaTransactionManager 有两个子类 OC4JJtaTransactionManager 和 WebLogicJtaTransactionManager，分别面向基于Oracle OC4J和Weglogic的JTA分布式事务管理。在这些情况下，需要使用具体的子类来代替通常的JtaTransactionManager。不过，大多数情况下，使用Spring提供的FactoryBean机制来获取不同JTA提供商提供的TransactionManager实现，然后注入JtaTransactionManager使用，是比较好的做法。org.springframework.transaction.jta包下，Spring提供了面向JOTM、Weblogic和Websphere的TransactionManager查找FactoryBean实现。如果需要，我们也可以根据情况，实现并给出其他的TransactionManager实现对应的用于查找的FactoryBean。

有了Spring的事务抽象框架，事务管理策略的转换也变得很简单，通常也只是简单的配置文件变更而已。如果我们最初只需要处理单一资源的事务管理，那么局部场景中的面向不同数据访问技术的PlatformTransactionManager实现，将是我们的最佳选择。即使后来需要引入分布式资源的事务管理，对于我们来说，也只是从局部事务场景中的某个PlatformTransactionManager实现转向JtaTransactionManager的变动而已。无论是编程式注入还是通过Spring的IoC容器注入，对于应用程序来说都不会造成很大的冲击。

2. 窥一斑而知全豹

PlatformTransactionManager的各个子类在实现时，基本上遵循统一的结构和理念。所以，我们不妨选择以DataSourceTransactionManager这一实现类作为切入点，以管中窥豹之势，一探Spring的抽象事务框架中各个PlatformTransactionManager实现类的奥秘所在。

不过，在开始之前，我们有必要先了解如下几个概念。

- transaction object。transaction object承载了当前事务的必要信息，PlatformTransactionManager实现类可以根据transaction object所提供的信息来决定如何处理当前事务。transaction object的概念类似于JTA规范中的javax.transaction.Transaction定义。
- TransactionSynchronization。TransactionSynchronization是可以注册到事务处理过程中的回调接口。它就像是事务处理的事件监听器，当事务处理的某些规定时点发生时，会调用TransactionSynchronization上的一些方法来执行相应的回调逻辑，如在事务完成后清理相应的系统资源等操作。Spring事务抽象框架所定义的TransactionSynchronization类似于JTA规范的javax.transaction.Synchronization，但比JTA的Synchronization提供了更多的回调方法，允许我们对事务的处理添加更多的回调逻辑。
- TransactionSynchronizationManager。类似于JTA规范中的javax.transaction.TransactionSynchronizationRegistry，我们通过TransactionSynchronizationManager来管理TransactionSynchronization、当前事务状态以及具体的事务资源。在介绍Spring事务框架实现原理的原型中，我们提到会将具体的事务资源，比如java.sql.Connection或者Hibernate Session绑定到线程，TransactionSynchronizationManager就是这些资源绑定的目的地。当然，从该类的名字也可以看出，它更多关注与事务相关的Synchronization的管理。

将它们与JTA规范中定义的接口相提并论，是因为这些概念只限于局部场景中对应的PlatformTransactionManager实现类使用，而JtaTransactionManager直接就使用对应的JTA产品提供的对应设施了，JtaTransactionManager的最终工作都是委派给具体的JTA实现产品，记得吗？

好的，有了这些铺垫，我们开始进入正题……

Spring的事务抽象框架以PlatformTransactionManager作为顶层抽象接口，具体的实现交给不

同的实现类，使用对象可以根据当前场景，选择使用或者替换哪一个具体的实现类。从这个层次看（见图19-10），整个框架的设计是以Strategy模式为基础的。不过，从各个实现类的继承层次上来看，Spring事务框架的实现则更多地依赖于模板方法模式。

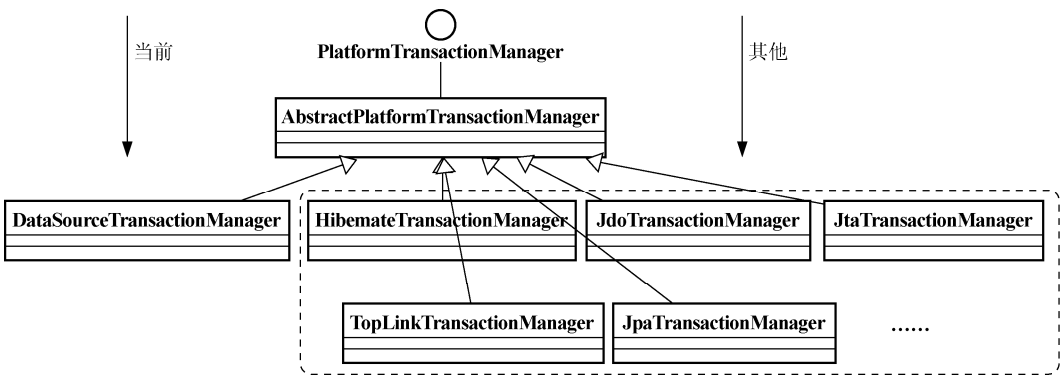

图19-10 DataSourceTransactionManager的实现层次

org.springframework.transaction.support.AbstractPlatformTransactionManager 作为DataSourceTransactionManager的父类，以模板方法的形式封装了固定的事务处理逻辑，而只将与事务资源相关的操作以protected或者abstract方法的形式留给DataSourceTransactionManager来实现。作为模板方法父类，AbstractPlatformTransactionManager替各子类实现了以下固定的事务内部处理逻辑：

- 判定是否存在当前事务，然后根据判断结果执行不同的处理逻辑；
- 结合是否存在当前事务的情况，根据TransactionDefinition中指定的传播行为的不同语义执行后继逻辑；
- 根据情况挂起或者恢复事务；
- 提交事务之前检查readOnly字段是否被设置，如果是的话，以事务的回滚代替事务的提交；
- 在事务回滚的情况下，清理并恢复事务状态；
- 如果事务的Synchonization处于active状态，在事务处理的规定时点触发注册的Synchonization回调接口。

这些固定的事务内部处理逻辑大都体现在以下几个主要的模板方法中：

- public final TransactionStatus getTransaction(TransactionDefinitiondefinition) throws TransactionException
- public final void rollback(TransactionStatus status) throws TransactionException
- public final void commit(TransactionStatus status) throws TransactionException
- protected final SuspendedResourcesHolder suspend(Object transaction) throws TransactionException
- protected final void resume(Object transaction, SuspendedResourcesHolder resourcesHolder) throws TransactionException

我们可不打算把这几个模板方法都讲一遍。毕竟，只要了解了前面两三个模板方法的流程，整个事务处理的图景基本上就可以展现在我们眼前了。

我们先从第一个模板方法getTransaction(TransactionDefinition)开始。getTransaction(TransactionDefinition)的主要目的是开启一个事务，但需要在此之前判断一下之前是否存在一个事务。如果存在，则根据TransactionDefinition中的传播行为决定是挂起当前事务还是抛出异常。同样的，不存在事务的情况下，也需要根据传播行为的具体语义来决定如何处理。getTransaction(TransactionDefinition)方法的处理逻辑，基本上按照下面的流程执行[①]。

(1) 获取transaction object，以判断是否存在当前事务。

```
Object transaction = doGetTransaction();
```

以上这行代码有如下两点需要申明。

- 获取的transaction object类型会因具体实现类的不同而各异，DataSourceTransactionManager会返回DataSourceTransactionManager.DataSourceTransactionObject类型实例，HibernateTransactionManager会返回HibernateTransactionManager.HibernateTransactionObject类型的实例，等等。AbstractPlatformTransactionManager不需要知道具体实现类返回的transaction object具体类型是什么，因为最终对transaction object的依赖都将通过方法参数进行传递，只要具体的实现类在取得transaction object参数后知道如何转型就行，所以，这一步返回的transaction为Object类型。
- doGetTransaction()是getTransaction(TransactionDefinition)模板方法公开给子类来实现的abstract类型方法。DataSourceTransactionManager在实现doGetTransaction()方法逻辑的时候，会从TransactionSynchronizationManager获取绑定的资源，然后添加到DataSourceTransactionObject之后返回。以此类推，其他AbstractPlatformTransactionManager子类都采用类似的逻辑实现了doGetTransaction()方法。

(2) 获取Log类的debug信息，避免之后的代码重复。如以下代码所示：

```
boolean debugEnabled = logger.isDebugEnabled();
if (debugEnabled) {
    logger.debug("Using transaction object [" + transaction + "]");
}
```

debugEnabled将以方法参数的形式在各方法调用间传递，以避免每次都调用logger.isDebugEnabled()获取debug日志状态。这一步与具体的事务处理流程关系不大。

(3) 检查TransactionDefinition参数合法性。如以下代码所示：

```
if (definition == null) {
    // Use defaults if no transaction definition given.
    definition = new DefaultTransactionDefinition();
}
```

如果definition参数为空，则创建一个DefaultTransactionDefinition实例以提供默认的事务定义数据。

(4) 根据先前获得的transaction object判断是否存在当前事务，根据判定结果采取不同的处理方式。如以下代码所示：

---

① 因为UML的Sequence图对于条件判断等逻辑无法以恰当的方式表示，而Activity图整个表示下来过大，不便显示，所以，最后决定以文字描述的方式进行。

```
if (isExistingTransaction(transaction)) {
    // Existing transaction found -> check propagation behavior to find out how to behave.
    return handleExistingTransaction(definition, transaction, debugEnabled);
}
```

默认情况下，isExistingTransaction(transaction)返回false，该方法需要具体子类根据情况进行覆写。对于DataSourceTransactionManager来说，它会根据传入的transaction所记载的信息进行判断，如下所示：

```
DataSourceTransactionObject txObject = (DataSourceTransactionObject) transaction;
return (txObject.getConnectionHolder() != null && txObject.getConnectionHolder().➥
isTransactionActive());
```

对于HibernateTransactionManager来说，则会将transaction强制转型为HibernateTransactionObject，然后根据HibernateTransactionObject所记载的信息来判断之前是否存在一个事务。其他具体实现类对isExistingTransaction(transaction)的处理亦是如此。

不管isExistingTransaction(transaction)返回结果如何，下面的处理主体上都是以TransactionDefinition中的传播行为为中心进行的。比如同样是PROPAGATION_REQUIRED，在存在当前事务与不存在当前事务两种情况下的处理是不同的，前者会使用之前的事务，后者则会创建新的事务，其他的传播行为的处理也是按照不同的场景分别处理。

(a) 如果isExistingTransaction(transaction)方法返回true，即存在当前事务的情况下，由handleExistingTransaction()方法统一处理存在当前事务，应该如何创建事务对应的TransactionStatus实例并返回。

如果definition定义的传播行为是PROPAGATION_NEVER，抛出异常并退出。如下：

```
if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_NEVER) {
    throw new IllegalTransactionStateException(➥
    "Existing transaction found for transaction marked with propagation 'never'");
}
```

这是由TransactionDefinition.PROPAGATION_NEVER的语义决定的。

如果definition定义的传播行为是PROPAGATION_NOT_SUPPORTED，则挂起当前事务，然后返回。如下：

```
if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_NOT_ ➥
SUPPORTED) {
    if (debugEnabled) {
        logger.debug("Suspending current transaction");
    }
    Object suspendedResources = suspend(transaction);
    boolean newSynchronization = (getTransactionSynchronization() == SYNCHRONIZATION_ ➥
    ALWAYS);
    return newTransactionStatus(
            definition, null, false, newSynchronization, debugEnabled, suspended- ➥
            Resources);
}
```

newTransactionStatus()方法返回一个DefaultTransactionStatus实例，因为我们挂起了当前事务。而PROPAGATION_NOT_SUPPORTED不需要事务，所以，返回的DefaultTransactionStatus不包含transaction object的信息（构造方法第二个参数）。

如果definition定义的传播行为是PROPAGATION_REQUIRES_NEW，则同样挂起当前事务，并开始一个新的事务并返回。如下：

```
if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_REQUIRES_NEW) {
    if (debugEnabled) {
        logger.debug("Suspending current transaction, creating new transaction with name [" +
        definition.getName() + "]");
    }
    SuspendedResourcesHolder suspendedResources = suspend(transaction);
    try {
        doBegin(transaction, definition);
    }
    catch (TransactionException beginEx) {
        try {
            resume(transaction, suspendedResources);
        }
        catch (TransactionException resumeEx) {
            logger.error(
            "Inner transaction begin exception overridden by outer transaction
            resume exception", beginEx);
            throw resumeEx;
        }
        throw beginEx;
    }
    boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION_
    NEVER);
    return newTransactionStatus(
    definition, transaction, true, newSynchronization, debugEnabled,
    suspendedResources);
}
```

AbstractPlatformTransactionManager首先将当前事务挂起，然后调用doBegin()方法开始新的事务。如果开始事务过程中出现异常，那么恢复之前挂起的事务。doBegin(transaction, definition)方法为abstract方法，需要具体子类来实现。在DataSourceTransactionManager中，doBegin()方法会首先检查传入的transaction，以提取必要信息判断之前是否存在绑定的connection信息。如果没有，则从DataSource中获取新的connection，然后将其AutoCommit状态改为false，并绑定到TransactionSynchronizationManager。当然，这期间也会涉及事务定义的应用以及条件检查等逻辑。在所有一切搞定之后，newTransactionStatus会创建一个包含definition、transaction object以及挂起的事务信息和其他状态信息的DefaultTransactionStatus实例并返回。

如果definition定义的传播行为是PROPAGATION_NESTED，根据情况创建嵌套事务，如通过Savepoint或者JTA的TransactionManager。如下：

```
if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_NESTED) {
    if (!isNestedTransactionAllowed()) {
        throw new NestedTransactionNotSupportedException(
        "Transaction manager does not allow nested transactions by default - " +
        "specify 'nestedTransactionAllowed' property with value 'true'");
    }
    if (debugEnabled) {
        logger.debug("Creating nested transaction with name [" + definition.getName() + "]");
    }
    if (useSavepointForNestedTransaction()) {
        // Create savepoint within existing Spring-managed transaction,
        // through the SavepointManager API implemented by TransactionStatus.
        // Usually uses JDBC 3.0 savepoints. Never activates Spring synchronization.
        DefaultTransactionStatus status =
                newTransactionStatus(definition, transaction, false, false, debugEnabled, null);
        status.createAndHoldSavepoint();
```

```
            return status;
        }
        else {
            // Nested transaction through nested begin and commit/rollback calls.
            // Usually only for JTA: Spring synchronization might get activated here
            // in case of a pre-existing JTA transaction.
            doBegin(transaction, definition);
            boolean newSynchronization = (getTransactionSynchronization() != ➥
            SYNCHRONIZATION_NEVER);
            return newTransactionStatus(definition, transaction, true, newSynchronization, ➥
            debugEnabled, null);
        }
    }
```

在这种情况下，会首先通过isNestedTransactionAllowed()方法检查AbstractPlatform-TransactionManager的nestedTransactionAllowed属性状态。如果允许嵌套事务，那么还得分两种情况处理。对于DataSourceTransactionManager来说，因为它支持使用Savepoint创建嵌套事务，所以，会使用TransactionStatus创建相应的Savepoint并返回。而JtaTransactionManager则要依赖于具体JTA产品的TransactionManager提供嵌套事务支持。

useSavepointForNestedTransaction()方法默认返回true，即默认使用Savepoint创建嵌套事务。如果具体子类不支持使用Savepoint创建嵌套事务，则需要覆写该方法，如JtaTransactionManager。

如果需要检查事务状态匹配情况，则对当前存在事务与传入的defintion中定义的隔离级别与ReadOnly属性进行检查，如果数据不吻合，则抛出异常。如下：

```
if (isValidateExistingTransaction()) {
    // validate isolation
    // validate read only
    ...
}
```

AbstractPlatformTransactionManager的validateExistingTransaction属性默认值为false。如果你想进一步加强事务属性之间的一致性，可以将validateExistingTransaction属性设置为true，那么这时，以上代码即会被触发执行。

剩下的就是在其他情况下，直接构建TransactionStatus返回。比如对应PROPAGATION_ SUPPORTS和PROPAGATION_REQUIRED的情况。

(b) 如果isExistingTransaction(transaction)方法返回false，即不存在当前事务的情况下。

当definition中定义的传播行为是PROPAGATION_MANDATORY的时候，抛出异常。因为不存在当前事务，所以根据PROPAGATION_MANDATORY的语义，理当如此。

当definition中定义的传播行为是PROPAGATION_REQUIRED、PROPAGATION_REQUIRES_NEW或者PROPAGATION_NESTED的时候，开启新的事务。如下：

```
if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_REQUIRED ||➥
definition.getPropagationBehavior() == TransactionDefinition. ➥
PROPAGATION_REQUIRES_NEW ||➥
definition.getPropagationBehavior() == TransactionDefinition. ➥
PROPAGATION_NESTED) {
    SuspendedResourcesHolder suspendedResources = suspend(null);
    if (debugEnabled) {
        logger.debug("Creating new transaction with name [" + definition.getName() + "]: "➥
        + definition);
    }
    try {
```

```
            doBegin(transaction, definition);
        }
        catch (TransactionException ex) {
            resume(null, suspendedResources);
            throw ex;
        }
        boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION_➥
        NEVER);
        return newTransactionStatus( ➥
        definition, transaction, true, newSynchronization, debugEnabled, ➥
        suspendedResources);
    }
```

之所以在doBegin之前先调用传入null的suspend()方法，是因为考虑到如果有注册的Synchronization，需要暂时将这些与将要开启的新事务无关的Synchronization先放一边。

剩下的其他情况，则返回不包含任何transaction object的TransactionStatus。这种情况下虽然是空的事务，但有可能需要处理在事务过程中相关的Synchronization。

从getTransaction(TransactionDefinition)的逻辑可以看出，AbstractPlatformTransactionManager更多关注的是事务处理过程中的总体逻辑，而跟具体事务资源相关的处理则交给了具体的子类来实现。

事务处理的完成有两种情况，即回滚事务或者提交事务，AbstractPlatformTransactionManager提供的rollback(TransactionStatus)和commit(TransactionStatus)两个模板方法，分别对应这两种情况下的处理。因为事务提交过程中可能需要处理回滚逻辑，我们不妨以commit(TransactionStatus)的实现流程看一下AbstractPlatformTransactionManager是如何处理事务完成的。

(1) 因为在事务处理过程中，我们可以通过TransactionStatus的setRollbackOnly()方法标记事务回滚，所以，commit(TransactionStatus)在具体提交事务之前会检查rollBackOnly状态。如果该状态被设置，那么转而执行事务的回滚操作。

rollback(TransactionStatus)的逻辑主要包含如下3点。

- **回滚事务**。这里的回滚事务又可分为如下3种情况。
  - 如果是嵌套事务，则通过TransactionStatus释放Savepoint。
  - 如果TransactionStatus表示当前事务是一个新的事务，则调用子类的doRollback(TransactionStatus)方法真正的回滚事务。doRollback(TransactionStatus)是抽象方法，具体子类必须实现它。DataSourceTransactionManager在实现该方法的时候，无疑是调用connection.rollback()。HibernateTransactionManager会通过它Session上的Transaction的rollback()方法回滚事务。其他子类对doRollback(TransactionStatus)的实现逻辑依此类推。
  - 如果当前存在事务，并且rollbackOnly状态被设置，则调用由子类实现的doSetRollbackOnly(TransactionStatus)方法，各子类实现通常会将当前的transaction object的状态设置为rollBackOnly。
- **触发Synchronization事件**。
  回滚时触发的事件比提交时触发的事件要少，只有triggerBeforeCompletion (status)和triggerAfterCompletion()。
- **清理事务资源**。如下所述。

- 设置TransactionStatus中的completed为完成状态。
- 清理与当前事务相关的Synchronization。
- 调用doCleanupAfterCompletion()释放事务资源，并解除到Transaction Synchronizationmanager的资源绑定。对于DataSourceTransactionManager来说，是关闭数据库连接，然后解除对DataSource对应资源的绑定。
- 如果之前有挂起的事务，恢复挂起的事务。

(2) 如果rollBackOnly状态没被设置，则执行正常的事务提交操作。

commit(TransactionStatus)方法的其他逻辑与rollback(TransactionStatus)方法基本相似，只是几个具体操作有所差别，如下所述。

- 回滚事务现在变成是提交事务。提交事务的时候，也会涉及如下几种情况。
  - 决定是否提前检测全局的rollBackOnly标志。如果最外层事务已经被标记为rollBackOnly，并且failEarlyOnGlobalRollbackOnly为true，则抛出异常(如代码清单19-5所示)。
  - 如果提交事务之前发现TransactionStatus持有Savepoint，则释放它。这实际上是在处理嵌套事务的提交。
  - 如果TransactionStatus表示要提交的事务是一个新的事务，则调用子类的doCommit(TransactionStatus)方法实现提交事务。doCommit(TransactionStatus)也是AbstractPlatformTransactionManager公开给子类实现的抽象方法，子类必须实现该方法。对于DataSourceTransactionManager来说，因为事务的提交由Connection决定，所以会直接调用connection.commit()提交事务。其他的子类也会使用自身的局部事务API在该方法中实现事务的提交。
- 需要触发Synchronization相关事件。不过，触发的事件比rollback(TransactionStatus)中的要多，包括triggerBeforeCommit()、triggerBeforeCompletion()、triggerAfterCommit()和triggerAfterCompletion()。
- 如果AbstractPlatformTransactionManager的rollbackOnCommitFailure状态被设置为true，则表示如果在事务提交过程中出现异常，需要回滚事务。所以，当commit(TransactionStatus)方法捕获相应异常，并且检测到该字段被设置的时候，需要回滚事务。rollbackOnCommitFailure的默认值是false，表示即使提交过程中出现异常，也不回滚事务。
- 既然commit(TransactionStatus)与rollback(TransactionStatus)一样，都是意味着事务的完成，那么也需要在最后进行事务资源清理的工作，具体内容可以参照rollback(TransactionStatus)部分。

代码清单19-5 DataSourceTransactionManager事务提交部分代码摘录

```
boolean globalRollbackOnly = false;
if (status.isNewTransaction() || isFailEarlyOnGlobalRollbackOnly()) {
    globalRollbackOnly = status.isGlobalRollbackOnly();
}
...
if (globalRollbackOnly) {
    throw new UnexpectedRollbackException( ➥
    "Transaction silently rolled back because it has been marked as rollback-only");
}
```

suspend和resume两个方法的逻辑更好理解了。前者会把TransactionSynchronization- Manager上当前事务对应的Synchroniazation信息以及资源获取到SuspendedResourcesHolder中，然后解

除这些绑定。后者则会将`SuspendedResourcesHolder`中保持的信息重新绑定到`Transaction-SynchronizationManager`。

实际上，如果将`AbstractPlatformTransactionManager`中处理`Synchroniaztion`回调以及事务传播行为的逻辑剥离一下的话，就会发现，整个的逻辑流程就是本章开始部分展示的实现原型所表达的那样。

图19-11展示了`AbstractPlatformTransactionManager`需要子类实现或者覆写的方法。对于各个子类来说，无非就是根据自身需要管理的资源和事务管理API提供这些方法的实现而已。

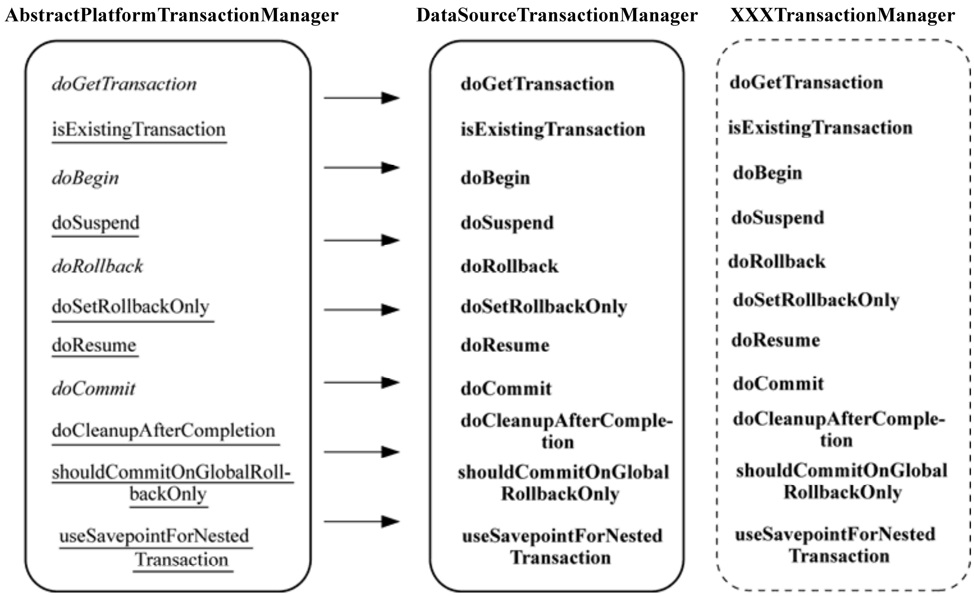

图19-11　模板类与实现类之间的纽带

19.3　小结

为了帮助你更容易地理解整个Spring事务管理抽象框架的设计和实现，我们在统一中原的过程中推导并实现了相应的`PlatformTransactionManager`以及相关原型实现。随后，在原型的基础上，对Spring事务管理抽象框架中的主要类进行了介绍和分析，尤其是对`DataSourceTransactionManager`的实现做了剖析，以期能够“以点带面”地帮助大家更好地理解Spring事务管理抽象框架中各`PlatformTransactionManager`通用的实现原理。

在了解了以上内容之后，我们将一起了解一下，在实际开发过程中，如何使用Spring的事务管理抽象框架让我们的日常开发生活变得更加美好。