正如我们即将从这本书中学到的一样，“声明式事务”这个词是指一种可选的基础事务管理方式，但它自身绝不是一种事务设计策略。

正如著名数学家 B.帕斯卡的名言，“我之所以将这封书信弄得有些长，是因为我没有时间将其缩短了。

我写此书的目的是要帮助读者了解每一种事务模型，理解这些模型蕴含的最佳实践，知晓事务设计模式，同时懂得如何将这些事务设计模式应用在各种各样的软件架构中。

无论您是使用 Spring 还是 EJB，理解可用的各种事务模型都是很重要的。

Java 中有三种可以的事务模型，分别称作

本地事务模型（Local Transaction Model），

编程式事务模型（Programmatic Transaction Model），

和声明式事务模型（Declarative Transaction Model)。

所谓“本地事务模型”，得名于事实上不是编程框架本身来管理事务，事务是交给本地资源管理器（local resource manager）来管理的。资源管理器是用于通信的、事实上的数据源（datasource）提供者。

“编程式事务模型”利用了 Java 事务 API（Java Transaction API, JTA）及其底层事务服务实现的能量以提供事务支持，突破了“本地事务模型”的种种限制。通过编程式事务模型，开发人员的编码对象是“事务”，而非“连接”。

“声明式事务模型”，在 EJB 的世界中也成为容器托管的事务模型（Container-Managed

Transactions），是本书通篇所主要聚焦的事务模型。在声明式事务模型的环境下，软件框架或“容器”管理了事务的开始和结束（或者提交，或者回滚）。开发人员仅仅需要告诉软件框架，碰到应用异常时“去回滚事务”即可，对事务的配置都是通过 EJB 中的 XML 部署描述文件（例如 ejb-jar.xml）或 Spring 中的 bean 定义文件（例如 ApplicationContext.xml）来完成的。

然而，时至今日，之于 ACID 本身的含义，相较于 1960 年，已经有些许改变了。ACID 是描述有关事务的期望特性的字首缩写，这些特性分别是原子性（Atomicity），一致性（Consistency），独立性（Isolation），和持久性（Durability）。

“原子性”的含义是，一个事务必须将它产生的所有更改作为一个单独的工作单元提交，或

者回滚。

“一致性”的含义是，在活动事务的处理过程中，数据库必须时刻要避免被置于不一致

（inconsistent）的状态。

“独立性”指的是各个独立事务之间的交互程度方面。ACID 特性的遵循度，决定了本事务在怎样的程度下保证自身的未提交更改不受访问同一块信息的其他事务的影响。

“持久化特性”的含义是，当我们收到了一个事务成功提交的信息，我们便能得到这个事务完成的保证了，同时系统对数据库或 JMS 对象产生了永久的更改，这样的更改不会因为系统失败而丢失。

Java 应用开发人员并不需要知道 Java 事务服务（Java Transaction Service, JTS）的幕后细节，便能够有效地管理事务。

无论使用何种框架，大多数企业 Java 应用使用 Java 事务接口（Java Transaction API，JTA）进行事务管理。

幸运的是，在处理事务时，并没有很多接口是开发人员需要真正关心的。例如，当使用编程式事务时，我们仅仅需要使用的唯一接口是javax.transaction.UserTransaction。这个接口让我们能够显式的开始一个事务，提交一个事务，回滚一个事务，并获取事务状态。我们当然也能够使用 TransactionManager接口发起、提交或回滚一个特定的事务。

**UserTransaction 接口**

UserTransaction 接口仅仅用于编程式事务模型，而且主要在 EJB 中使用。编程人员仅仅需要

关心其中的如下方法：

 begin()

 commit()

 rollback()

 getStatus()

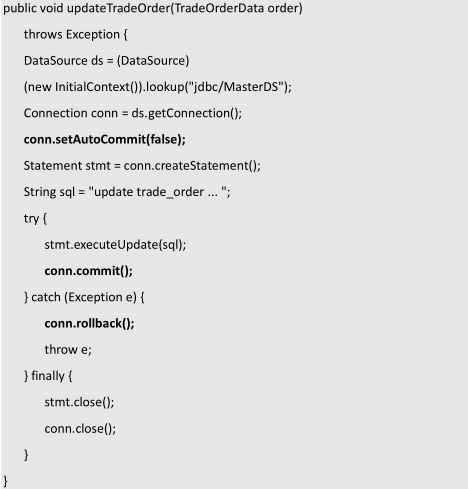
javax.transaction.UserTransaction.begin()

**TransactionManager 接口**

javax.transaction.TransactionManager 接口主要用于声明式事务模型。在编程式事务下，使用TransactionManager 接口，您能够做到使用 UserTransaction 接口基本上同样多的事情。然而，对大多数方法而言，最好使用 UserTransaction，别去碰 TransactonManager 接口，除非您需要暂停（suspend）或继续（resume）一个事务。

# 第二章 本地事务模型

“本地事务”这个术语指的是这样一个事实：事务被底层数据库（DBMS）或在 JMS 中被底层消息服务提供者所管理。从开发人员的角度来看，在本地事务模型中，我们所管理的并非“事务”，而是“连接”。下面的代码展示了直接使用 JDBC 编码来进行本地事务模型管理的实例：



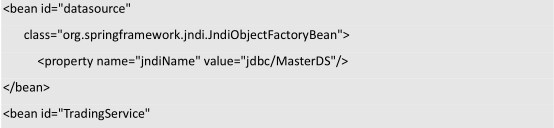
注意在上面的例子中，我们对 Connection.setAutoCommit(false)和 Connection.commit()、Connection.rollback()等方法的结合使用。setAutoCommit()方法是开发者控制的连接管理场景中非常重要的部分。自动提交标志（auto commit flag）告诉底层数据库，在每个 SQL 语句执行完毕后，是否应该立即提交连接。true 值让数据库在执行了每个 SQL 语句后立即提交或回滚本连接，而 false 值将导致连接仍然处于活动状态，不被提交，直至一个显式的 commit()方法调用。一般来说，这个标志默认设置为 true。因此，如果我们有多条 update 的 SQL 语句，每条语句将被单独执行，单独提交，彼此独立，在这样的情况下对 Connection.commit()和 Connection.rollback()的调用将被忽略。

在 Spring 框架中，可以简单地以如下方式使用

org.springframework.jdbc.datasource.DataSourceUtils 接口进行针对 JDBC 的底层进行编码：



在 Spring 环境下，数据源以及对应的业务逻辑对象可以在 Spring 配置文件中定义，举例如下：





通过设置自动提交标志为 false，我们告诉底层 DBMS 我们将自行调用 commit()和 rollback()方法，自行管理连接。通过这样的方式，我们能将更新的 SQL 聚集在一起，在单独的原子事务中形成单个逻辑工作单元。下面列出了管理多条更新语句的例子代码：



通过在上面的代码中添加 conn.setAutoCommit(false)、commit()和 rollback()方法，两条更新SQL 将被当作单个独立工作单元处理。

**为了说明本地事务模型的缺陷**，可以看看下面的代码示例，我们将两个更新 SQL 分隔在数据访问对象（Data Access Object，DAO）方法中：



OrderDAO 和 TradeDAO 对象包含了之前代码实例具备的同样的 SQL 语句和连接逻辑。在这个例子中，不论在各自的 DAO 方法中怎样配置自动提交和连接管理，每个更新都是分别处理的。这意味着在每个 DAO 的 update()方法最后数据库更新都将被提交。

虽然此技术在大多数情况下是可行的，连接传递却不被认为是一种高效的事务设计策略。使

用连接传递极易造成错误，并且需要很大的编程工作量编码维护。如果您发现自己不得不将

代码改成上面那样，那我可以告诉您，是到了放弃本地事务模型，采用编程式模型或声明式

模型的时候了。

# 本地事务的费神之处与限制

对于小型的应用环境下简单的更新操作，本地事务模型工作得很好。然而，一旦应用的复杂

度增加，这个模型就捉襟见肘了。本地事务模型本身的限制会对您的应用架构形成相当的掣肘。

第一个问题是，一旦使用本地事务模型，开发人员就连接逻辑造成错误代码的几率是很大的。

开发人员必须非常关注自动提交标志的设置，特别在同一方法中进行多个更新操作时更是如

此。

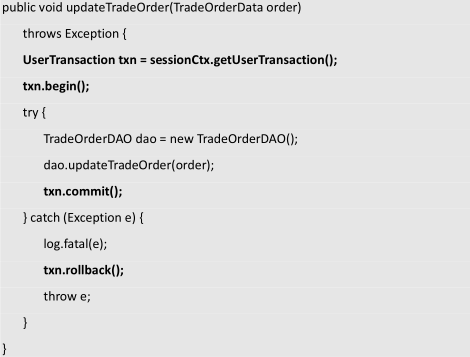
本地事务模型带来的另一个问题是，本地事务不能在使用 XA 全局事务协调多个资源时并发地存在（我们将会在第五章中看到有关细节）。当需要协调多个资源（如数据库和 JMS 目标，如队列或主题）时，您不可能在保证 ACID 特性的前提下使用本地事务模型。鉴于这些限制和不足，本地事务模型最好只能在单表更新的，最简单的基于 WEB 的 Java 应用中使用。

# 第三章 编程式事务模型

编程式事务模型和本地事务模型两者最大区别之一是，开发人员使用编程式模型，管理的是

事务（transaction），而不是连接（connection）。

当在 EJB 中时，编程式事务模型常常被称为Bean 管理事务（Bean-Managed Transactions），或 BMT。“BMT”这个术语不太常使用，因为在 EJB 容器之外，编程式事务模型也可用在 servlet 容器之中，能应用于 POJO（而不仅仅是EJB）。下面的代码展示了 EJB 框架结合 JTA 使用编程式事务模型的实例（事务逻辑用粗体体现）：



在上面的例子里，事务上下文（transaction context）被传递到 TradeOrderDAO 对象中，因此不像本地事务模型那样，TradeOrderDAO 并不需要管理连接和事务。它所需要做的仅仅是从连接池中获得一个连接，用完后还回去。

在编程式事务模型中，开发人员负责开启和终止事务。在 EJB 环境下，这是通过

UserTransaction 接口完成的。

begin()方法用于开启事务，commit()或 rollback()方法用于终止事务。

在 Spring 框架里，这些操作是通过使用 org.springframework.transaction 包下的TransactionTemplate 或 PlatformTransactionManager 完成的。

在 Spring 框架中，您可以选择使用 TransactionTemplate 或 PlatformTransactionManager 。下面的代码示例展示了如何使用 TransactionTemplate 技术的例子，这是更为常用的方式：





正如您从上面例子看到的，Spring 使用事务回调（transaction callback）将包含在业务方法中的逻辑在事务上下文中包裹起来。注意，当使用此技术的时候，并非要像在 EJB 中那样一定需要显示的调用 begin()和 commit()。并且，这里通过 TransactionStatus.setRollbackOnly()处理回滚，而不像 EJB 那样调用 Transaction.rollback()处理之。

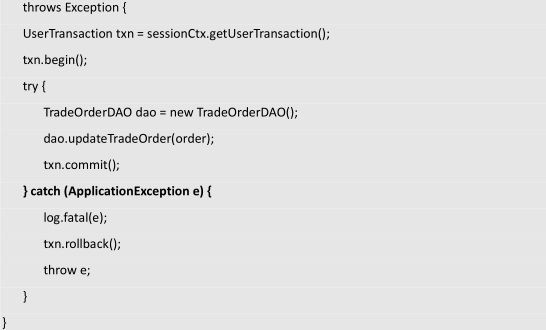
# 编程式事务中的编码陷阱

当使用编程式事务模型时，开发人员必须十二分的警惕异常的处理（exception handling）。

看看下面的代码，我们仅仅捕获和处理了应用异常（一个被检查异常，checked exception），

而忽略了运行时异常（runtime exception）：





# 编程式事务的使用场景

虽然通常不推荐使用编程式事务，在有的场景下编程式事务还是非常有用的。**编程式事务通**

**常的用武之地是客户端发起事务（cient-initiated transactions）的情形**。如果客户端为一个业

务请求做多次远程方法调用，从道理上讲事务必须由客户端开启。使用 JTA 时，就需要使用UserTransaction 接口和编程式事务。对这样的需求，您必须在客户端 bean 使用编程式事务，而在远程的 EJB 使用声明式事务——因为事务上下文不能被传递给使用编程式事务的 EJB。

**另一个可能的场景是使用本地的 JTA 事务（localized JTA transactions）**。JTA 事务处理是非常消耗资源的。有时候您需要在每一个细节上都考虑足够的性能优化（如信用卡处理业务）。

当有价值的资源（如数据库和消息队列）被消耗殆尽了，整个应用，而不是单个线程的吞吐

量和整体性能将会受到严重影响。因此，为了性能调优的目的，您也许会选择在 JTA 事务之外执行相当一部分代码，而在万不得已时再使用 JTA。以信用卡处理为例，您也许不会在数据装载，数据校验，数据验证，以及过帐时使用 JTA 事务。然而，当您需要将 money 从一个账户转向开户银行时，您就需要开启事务了。这个事务将在账户处理完毕后立即终止，而后剩下的流程都在没有事务上下文的环境下进行。这就是本地 JTA 事务的例子。这种情况下，使用声明式事务非常不便，因为它缺乏对事务何时开始，何时终止灵活的控制。

还有一个场景是使用长时间运行（long-running）的 JTA 事务。在 EJB 中，长时间运行的事务是通过有状态会话 Bean（Stateful SessionBeans）实现的。有时候您想要一个事务跨越对服务器的多次请求。此时，您可能会在一个有状态会话 Bean 的方法中开启事务，而在另一个有状态会话 Bean 中终止事务。虽然在技术上这是可行的，但从设计的观点看这是一个很不好的做法——这样数据库或 JMS 资源在这个较长的处理时间段内会被很快耗尽。尽管如此，像本地 JTA 事务一样，如果使用声明式事务模型，这点是决不可能做到的。

如果开发人员不具有充分的理由，最好别使用编程式事务模型。若遇到客户端开启事务，本

地 JTA 事务，或长时间运行事务的情况，方可考虑用之。对于其他的场景，您应该选择声明

式事务模型。

# 第四章 声明式事务模型

如我们在编程式事务模型中看到的，开发人员必须显示的调用 begin()方法去开启事务，调

用 commit()或和 rollback()去提交或回滚事务。

如果使用声明式事务模型（DeclarativeTransaction Model），容器将管理事务，这意味着开发人员不用编写任何代码，便可开始或提交事务了。然而，开发人员必须要告诉容器“如何”去管理事务。

在 EJB 中，这通常是通过ejb-jar.xml 部署描述文件中的设置，或特定应用服务器的附加部署描述文件来配置；在 Spring中，则是通过 ApplicationContext.xml 来配置。

对于 Spring，我们通过使用 TransactionProxyFactoryBean 代理指明我们要使用声明式事务。

下面的代码演示了在 Spring 中怎么做：



如上所示，如果要在 Spring 中使用声明式事务，需要将参与事务的 bean 用一个 proxy 包裹住（见上面的粗体代码）。而且，上面配置文件中使用“-Exception”告知 Spring 在碰到任何Exception 的情况下都回滚事务，于是我们不需要亲自去调用 SetRollback()方法了。通常开发人员需要在这里指定一个被检查异常（checked exception），而不会像我这样简单的配置

Exception 类。

# 事务属性（Transaction Attributes ）

在使用声明式事务时，我们必须告诉容器如何管理事务。例如，什么时候开启事务？哪些方

法需要事务？当事务不存在时，容器是否要开启事务？

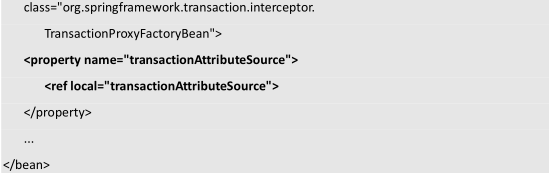
# 配置事务属性

事务属性通常和方法相关联，即使一个缺省属性应用在整个 bean 上也如此。

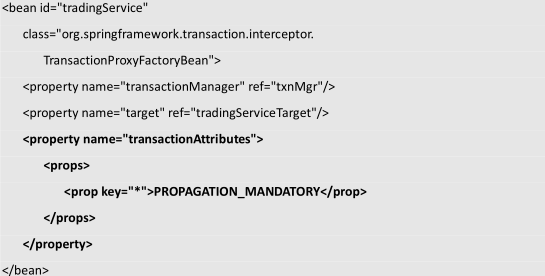
在 Spring 中，可以通过使用 **TransactionAttributeSource** 或 **TransactionProxyFactoryBean** 的属性来指明事务属性。下面的例子同时说明了这两种情况：

技巧 1. 使用 TransactionAttributeSource





技巧 2. 使用 TransactionProxyFactoryBean



如上例所示，第二种技巧正是指派事务属性的一种简洁途径，这样做就避免了将

TransactionAttributeSource 定义为一个单独的 bean。当使用第二种方法时，Spring 实际在幕

后创建了一个 NameMatchTransactionAttributeSource 对象。

使用事务属性的一种方式是，在方法级别而非类级别去指定属性。然而，**比为每个方法显式**

**指派各自的事务属性更好的做法是，在类的级别指派一个缺省事务属性，然后在方法级别去**

**做必要的优化**。

# 事务隔离级别的现实

另一个对开发人员可用的事务设置是事务的隔离级别。事务隔离（Transaction Isolation）是

指交织在一起发生的事务之间相互影响的程度。它决定了在其他事务访问和更新同一份数据

时，一个事务对更新所允许的可见程度。关系数据库系统（DBMS），EJB，Spring 都允许设

定事务隔离级别。然而，这样的设置，是数据库和应用服务器实现相依赖的。应用服务器可

能支持多种隔离级别，但对应的数据库必须支持这些同样的隔离级别，这些设置才可能真正

生效。

事务隔离同时是数据库并发度（concurrency）和一致性（consistency）的函数。当我们提高

了事务隔离度，我们实际上降低了数据库并发程度，但提高了一致性。下面的图示说明了三

者之间的关系。



该设置能直接影响应用的性能和数据完整性。例如，对于高性能的应用，诸如信用卡处理程

序，您可以通过降低隔离级别以提高并发度（但这样会损害数据完整性）。对低并发、需要

高度数据完整性的金融类应用，您需要提高隔离以别，以增强整体数据一致性（但损害了性

能）。大多数应用服务器和数据库都具有平衡并发度和一致性的一个默认设置值。

EJB 和 Spring 均支持四种主要的隔离级别，这些设置（隔离度从低到高）分别是：

 TransactionReadUncommitted 读取未提交

 TransactionReadCommitted 读取已提交

 TransactionRepeatableRead 可重复读

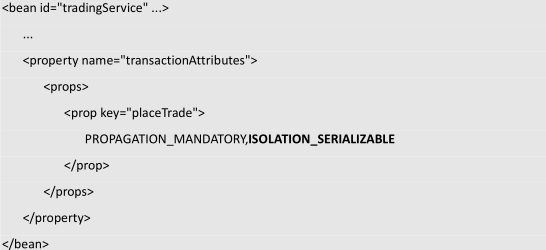
 TransactionSerializable 可序列化

# 事务隔离级别设置的现实

在 EJB 规范中，隔离级别设置是应用服务器的扩展功能，因此只能在特定应用服务器的部署描述文件的扩展部分设置。在 Spring 环境里，隔离级别是随事务属性（transaction attribute）

来设置的。

以下展示了 Spring 中同样的配置：



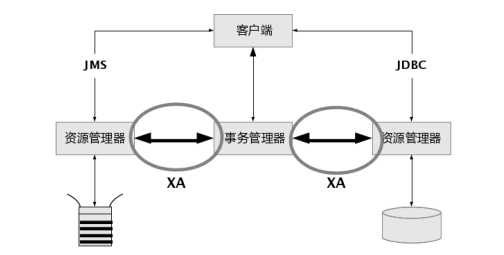
# 第五章 XA 事务处理

XA 接口详解

X/Open XA 接口是双向的系统接口，在事务管理器（Transaction Manager）以及一个或多个

资源管理器（Resource Manager）之间形成通信桥梁。事务管理器控制着 JTA 事务，管理事

务生命周期，并协调资源。在 JTA 中，事务管理器抽象为 javax.transaction.TransactionManager接口，并通过底层事务服务（即 JTS）实现。资源管理器负责控制和管理实际资源（如数据库或 JMS 队列）。下图说明了事务管理器、资源管理器，以及典型 JTA 环境中客户端应用之间的关系：



注意，上图中 XA 接口形成了事务管理器和资源管理器之间的通信桥梁。因为 XA 接口的双

向特质，XA 支持两阶段提交协议，我们将在本章的后续部分讨论。

什么时候应该使用 XA ？

在 Java 事务管理中，常常令人困惑的一个问题是什么时候应该使用 XA，什么时候不应使用XA。由于大多数商业应用服务器执行单阶段提交（one-phase commit）操作，性能下降并非一个值得考虑的问题。然而，非必要性的在您的应用中引入 XA 数据库驱动，会导致不可预料的后果与错误，特别是在使用本地事务模型（Local Transaction Model）时。因此，一般来说在您不需要 XA 的时候，应该尽量避免使用它。