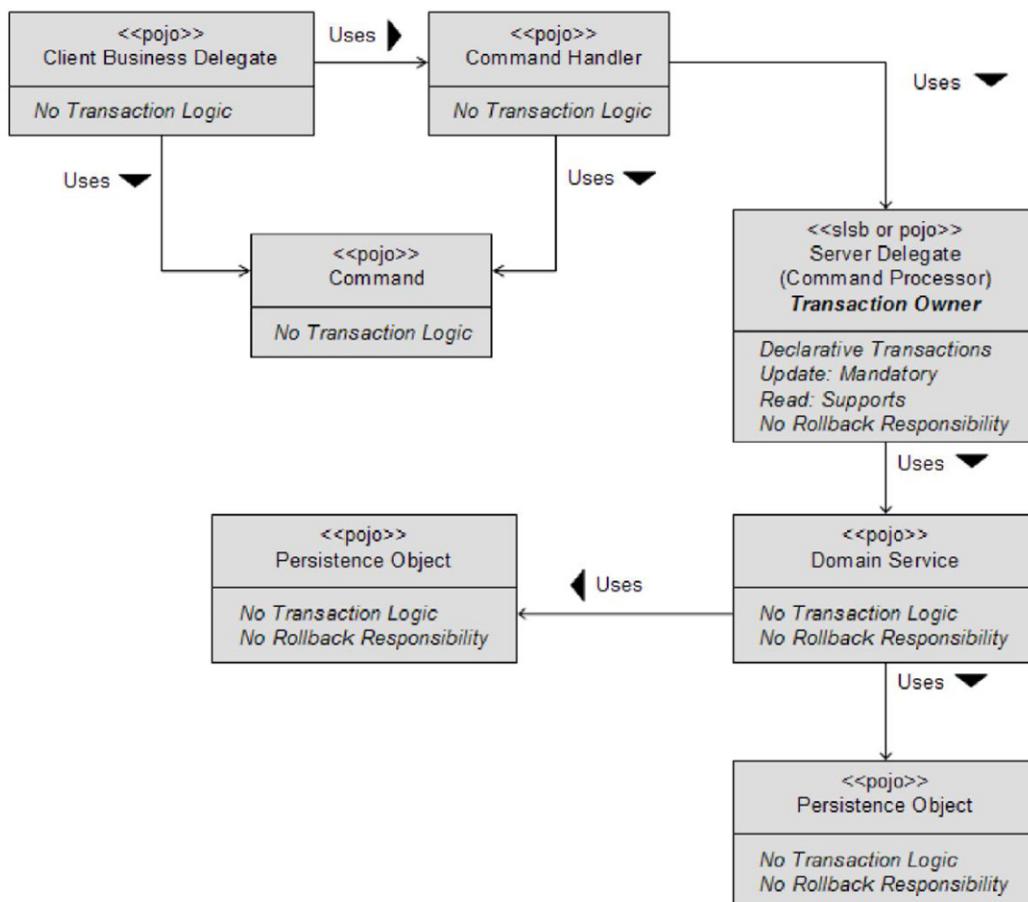


# EJB和Spring的事务模型示例

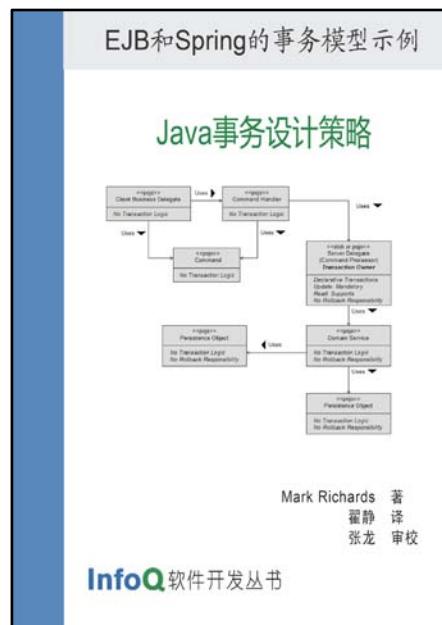
## Java事务设计策略



Mark Richards 著  
翟静 译  
张龙 审校

# 免费在线版本

( 非印刷免费在线版 )



了解本书更多信息请登录[本书的官方网站](#)

InfoQ 中文站出品



本书由 InfoQ 中文站免费发放，如果您从其他渠道获取本书，请注册 InfoQ 中文站以支持作者和出版商，并免费下载更多 InfoQ 软件开发系列图书。

本迷你书主页为

<http://www.infoq.com/cn/minibooks/JTDS>

## 致谢

我应当感谢 Alan Beaulieu (《Learning SQL》的作者和《Mastering Oracle SQL》的联名作者) 和 Stuart Dabbs Halloway (《Component Development for the Java Platform》的作者), 你们自始至终审校了本书并支持我完成了这一“工程”。在审查最终手稿, 提出出色的注解和建议时, 你们对技术方面和排版方面的见解不仅有用, 而且深刻(有时也很幽默!)。我要感谢 Mark Little (《Java Transaction Processing》的作者) 对本书最终稿仔细的审阅, 向我提出了很多出色的技术意见和建议。您关于本书的各种技术细节的拨冗研讨非常有帮助, 对拙著的最终成籍有很大裨益。我也要特别感谢 Floyd Marinescu (TheServerSide.com 的创始人和《EJB Design Pattern》的作者) 为出版本书作出的努力, 您在我写作过程中给出了很多绝妙的指引与反馈。您在技术和排版方面指出的方向无疑是十分正确的, 没有您的帮助, 本书不可能像现在这样好。

我尤其要感谢贤内 Rebecca 对此书自始至终的支援。你帮助审阅了初稿的头几章, 并在写作的过程中给予我持续不断的 support。有了你的鼓励和信心加强, 我才得以最终完成此书。现在终于写完出版了, 我保证我在家的时候会尽量远离电脑(对!这样的日子一如既往地——不远了!)

# 目录

致谢 .....	1
目录 .....	2
第一章 引言 .....	3
第二章 本地事务模型 .....	11
第三章 编程式事务模型 .....	19
第四章 声明式事务模型 .....	29
第五章 XA 事务处理 .....	50
第六章 事务设计模式 .....	61
第七章 客户端拥有事务的设计模式 .....	63
第八章 领域服务拥有事务的设计模式 .....	73
第九章 服务器端代理拥有事务的设计模式 .....	80
第十章 总结 .....	89
关于作者 .....	89



促进软件开发领域知识与创新的传播



中文 | 英文 | 日文 | 葡文 | .....

## 第一章 引言

在当今世界上有各种各样的企业级 Java 应用。虽然有些不过是简单的 Web 应用，只使用了一些本地 JavaBeans 或 POJOs ( Plain Old Java Objects )，但还有很多是复杂的 N 层应用，他们会使用商业或开源的应用服务器，使用很多框架，如远程的 Enterprise Java Beans( EJBs )，JMS ( Java Messaging Service )，甚至是 Spring。虽然有些复杂的企业应用运转良好，但大多数应用都会时不时地遭受到无法解释的数据集成问题，比如不正确的账户结余、一个客户的多个订单、系统“丢失”了订单、以及在数据库的表之间出现的常规数据同步问题。这些问题通常都是由错误或没有使用事务管理策略造成的。

作为一名程序员“老兵”，我发现我见过的大多数 Java 应用都不具备合理的事务设计策略，或仅仅是简单地依赖数据库去管理复杂事务。我经常询问那些应用架构师和开发人员，他们对应用的事务管理整体设计策略是怎样的。不论是使用 EJB 或是 Spring 框架，我得到的答案常常是：“我们使用声明式事务”。然而通过本书，我们将可以了解到，术语“声明式事务”指的是一种事务模型，可以用作管理事务的基础，但就其本身而言它却必定不是一种事务设计策略。

开发人员、设计师和架构师应该关注事务处理过程。这大概因为有三件事我们这些人在一生中是逃不掉的，那就是“死亡”，“缴税”，以及“系统故障”。我臆想，除非像电影“王牌大贱谍”那样被冷冻起来，我们基本上对“死亡”没有任何办法；同样，无论您活的长或短，“缴税”都是不可避免的<sup>\*</sup>；幸运的是，我们可以针对“系统故障”做一些事情。而懂得了 Java 中事务管理是怎么工作的，随即开发出正确的事务设计策略，能够帮助您避免应用与数据库中的很多数据完整性问题，从而减少“系统故障”带来的杯具。

正如书名所暗示的那样，本书的主旨是讲述怎样利用 EJB 和 Spring 等基于 Java 的编程框架提供的事务模型来设计高效的事务管理策略。虽然，我会深入到它们各自事务模型的细节中，这些特定的章节本质上却是为了描述技术特点、最佳实践，以及这些模型之中的陷阱。本书的事务设计模式章节会把这些概念和技术汇总，描述如何利用这些模型在您的 EJB 或 Spring 应用中高效地管理事务。

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。”大多数读者不具备时间和耐心去阅读一部冗长的关于事务管理的著作。基于这个原因，本书的目标定位为：以一种简明的方式，为架构师和开发人员展示为各种规模的 Java

---

<sup>\*</sup> 在美国至少是如此。——译者

应用构建有效的事务设计策略所需要的信息，而不论他们采用哪一种编程框架。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。

我将分别使用基于 EJB2.1、EJB 3.0 和 Spring 框架的实例来描绘本书阐述的事务管理概念。虽然这些框架及其对应持久化框架所涉及的有关事务处理的所有细节已经大大超出了本书范围，我仍然希望能提供给读者足够的细节去理解建立一个高效的事务设计策略所需的概念和实现技术。

## 不同的事务模型

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）。虽然，在随后的章节将分别深入讲解，我想在这里还是有必要将它们分别概述一下。

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（data source）提供者。举例来说，对于数据库，资源管理器是通过数据库驱动和数据库管理系统（Database Management System，DBMS）来实现的。对于 JMS，所谓资源管理器就是通过特定的 JMS 提供者（provider）实现的队列（queue）或主题（topic）的连接工厂（connection factory）。经由本地事务模型，开发人员管理的是“连接（connection）”，而非“事务”。DBMS 和 JMS 的提供者真正管理了本地事务。关于“本地事务模型”的细节将在第二章介绍。

“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”。通过使用 javax.transaction.UserTransaction 接口，开发人员调用 begin()方法开始一个事务，调用 commit()或 rollback()方法去终止这个事务。虽然通常情况下不鼓励大量使用编程式事务，然而，在客户端发起的，对远程无状态会话 Bean（Stateless Session Beans）的 EJB 事务访问的场景下，它还是有可能用到的。我们将在第三章仔细讨论这种事务模型。

“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。有关“声明式事务模型”的细节将在本书第四章讨论。

## 伙计，ACID 特性在哪里？

如果我们生活在二十世纪六零年代，这个话题或许将转化成本书中最流行的章节。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。无论在本次事务中产生了多少数量的更改，所有的更改需被当作一个整体处理。原子特性有时被称作逻辑工作单元（LUW，Logical Unit of Work）特性，或称单一工作单元（SUW，Single Unit of Work）特性。

“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。这意味着在事务期间，每次对数据库实施的插入、更新或删除操作时，数据库的完整性约束（integrity constraints）都要得到保证——即使在事务还未被提交时也必须如此。这个特性对开发人员带来了影响。例如，一致性原则隐式地表明，在事务处理的过程中，您不能在未添加主表记录（summary record）的情况下，先去添加从表记录（detail record）。虽然某些数据库允许将约束检查延迟到提交时进行，一般而言，您不能在事务处理期间破坏外关键字限制，即使您打算随后修正它也不行。

“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。独立性是由一致性和并发度共同决定的。在独立性级别提高时，一致性将会更好，但并发程度将降低。我们将在第四章对“独立性”稍作更深入的讨论。

“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——无论是我们对服务器控制台泼啤酒，还是在服务器上玩网游造成的宕机。而今很多主流数据库厂商采用的一些精巧的缓存策略有可能对修改的持久性造成问题，但基本上来说，当我们收到提交成功的信息时，我们就能被确保我们的修改是恒久的而且不被丢失了。

## JTA 和 JTS

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。然而，无论您采用 EJB 还是 Spring 框架，了解 Java 对分布式事务处理的限制是非常重要的。

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。 JTA 是开发人员用于事务管理的接口。与之对应， Java 事务服务（ JTS ），是被大多数开源或商业应用服务器所使用的，实现了 JTA 的底层事务服务（注意市场上也有 JTS 之外的其他事务服务软件，能够用于某些应用服务器）。我们可以认为 JTA 和 JTS 的关系与 JDBC 和对应的底层数据库驱动类似。 JTA 可以通过商业中间件本身，或通过开源的事务管理器，例如 JBoss 事务服务（ <http://www.jboss.com/products/transactions> ）、 JOTM （ <http://jotm.objectweb.org> ）等实现。

Java 事务服务（ JTS ）是 CORBA 中 OTS1.1 规范的 Java 版（ OTS 是 Object Transaction Service 的缩写）。对于一般开发人员，这并不特别重要——除非您具有打破砂锅问到底的嗜好，或者要对付一次变态的面试。虽然 JTS 不是 J2EE 规范所要求必须实现的，但对于异构系统实现间的分布式事务交互来说， JTS 是必须的。因为按照规范， JTA 必须同时支持 JTS 和非 JTS 的实现，通常很难简单通过查看 JTA 接口的实现程度区分其支持的具体功能。例如，虽然 JTS 规范规定了对嵌套事务（ nest transaction ）的可选支持， J2EE 并不支持这一特性。您并不能通过审查 JTA 体会到这个区别。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。当在 EJB 环境中使用声明式事务时，我们主要关心 EJBContext 接口中的 setRollbackOnly() 方法。我们诚然可以利用 javax.transaction 包中提供的接口做很多事情，但开发人员实际上也不会怎么用到它们。例如我们将会在本书的后续章节看到的一些直接访问事务管理器（ transaction manager ）的情况。这些情形发生在我们使用声明式事务 然而需要手动暂停或继续事务的时候。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()**

在编程式事务模型中，begin()方法用于开启一个新的事务，并且将此事务与当前线程相关联。如果某个事务已经与当前线程建立过关联，并且底层事务服务不支持嵌套事务，该方法会抛出一个 NotSupportedException 异常。

#### **javax.transaction.UserTransaction.commit()**

在编程式事务模型中，commit()方法用于提交和当前线程关联的事务，并且终止该事务。这个方法同时将此事务与当前线程解关联。在 Java 中，仅仅有一个事务能够与当前线程建立关联。在 XA 环境下，这个方法可能抛出 HeuristicMixedException 或 HeuristicRollbackException，表示资源管理器做出了独立于事务管理器的决定，在两阶段提交过程中回滚或者部分提交了该事务。我们将在第五章对两阶段提交和“经验异常”处理做更深入的讨论。

#### **javax.transaction.UserTransaction.rollback()**

在编程式事务模型中，rollback()方法用于回滚和当前线程关联的事务，并终止该事务。这个方法同时将此事务与当前线程解关联。

#### **javax.transaction.UserTransaction.getStatus()**

在编程式事务模型中，getStatus()方法返回一个整型数值，用以表示当前事务的状态。这个整型返回值初看是没有什么意义的，不过我们可以使用 javax.transaction.Status 接口来确定 getStatus()方法返回的值是什么含义。关于 javax.transaction.Status 接口的细节将在本章稍后的部分讨论。

### **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 TransactionManager 接口，除非您需要暂停（suspend）或继续（resume）一个事务。

#### **javax.transaction.TransactionManager.suspend()**

在声明式或编程式事务模型中，suspend()方法用于暂停关联于当前线程的事务。该方法返回当前事务的引用；如果没有事务和当前线程关联，则返回 null。在我们需要暂时停止当前事

务，执行一些不兼容 XA 的代码或存储过程时，这个方法是相当有用的。我们将会在本书的第五章中看到有关这个方法的例子。

#### **javax.transaction.TransactionManager.resume()**

在声明式或编程式事务模型中，resume()方法用于继续之前暂停的事务。该方法的传入参数为之前暂停的事务 \*，将此事务和当前线程关联，随即继续此事务。

#### **EJBContext 接口**

EJBContext 接口使用在 EJB 环境下的声明式事务模型中，对于事务管理，仅仅一个方法有用，这就是 setRollbackOnly()。

#### **javax.ejb.EJBContext.setRollbackOnly()**

在声明式事务模型下，setRollbackOnly()方法用于通知容器：当前事务仅可能产生的后果便是回滚。有意思的是这个方法本身在被调用到的时候并不真正立即回滚事务；它只是标记事务是“需要回滚”的，如果调用 getStatus()方法，此时会返回 STATUS\_MARKED\_ROLLBACK 状态。使用 TransactionManager.setRollbackOnly()也会产生同样的结果。不过我们既已习惯使用了 SessionContext 或 MessageDrivenContext 等上下文对象，我们自然会使用(EJBContext 提供的)这个方法。

#### **Status 接口**

如前所述，我们能够通过 javax.transaction.Status 接口枚举值获取到事务的状态，这个值是经由调用 UserTransaction.getStatus()返回得到的。我花专门的一节来阐述这个问题，因为它非常酷，它给我们提供了关于当前事务状态非常丰富的有用信息。Status 接口包括下面一些枚举值：

- STATUS\_ACTIVE
- STATUS\_COMMITTED
- STATUS\_COMMITTING
- STATUS\_MARKED\_ROLLBACK
- STATUS\_NO\_TRANSACTION
- STATUS\_PREPARED
- STATUS\_PREPARING

---

\* API 的方法签名实际上是 void resume(Transaction tobj) ——译者

- STATUS\_ROLLEDBACK
- STATUS\_ROLLING\_BACK
- STATUS\_UNKNOWN

在以上列出的所有状态值中，对主流 Java 业务应用开发人员真正有用的是这几个：

STATUS\_ACTIVE , STATUS\_MARKED\_ROLLBACK 和 STATUS\_NO\_TRANSACTION。下面将会更详细地究其细节和用途加以叙述。

### **STATUS\_ACTIVE**

有时候，看看当前事务是否已经和线程关联的状态是重要的。例如，出于调试或优化的目的，我们也许需要在查询的过程中加入一个拦截器或切面，查看事务是否存在。使用这个切面和状态值，我们能够侦测到当前事务设计策略可能的待优化点。另一种情况下，我们如果需要暂停当前事务，或执行可能造成应用失败的代码（例如在 XA 环境下执行带有 DDL 的存储过程），我们也会去检查这个状态。下面的代码片段展示了 STATUS\_ACTIVE 状态的使用：

```
...
if (txn.getStatus() == Status.STATUS_ACTIVE)
    logger.info("在查询操作中事务是活动的");
...
```

此例的逻辑是，“如果我们在查询方法中发现有事务存在，则在日志文件中记录”。这可能意味着我们虽然已具有事务，但实际上并不需要它。据此，可识别出一个可能的优化机会，或暴露出我们事务设计整体策略中的问题。

### **STATUS\_MARKED\_ROLLBACK**

在声明式事务模型下，这个状态常常是有用的。为了性能优化的需要，我们常常想要对之前方法调用中被标记为回滚的事务进行跳过处理。因此，如果我们想要查看此事务是否被标记回滚了，我们可以这样安排代码：

```
...
if (txn.getStatus() == Status.STATUS_MARKED_ROLLBACK)
    throw new Exception(
        "后续处理因事务回滚而终止");
...
```

### **STATUS\_NO\_TRANSACTION**

这个状态非常重要，因为它是确定是否真的没有事务上下文的唯一途径。和 STATUS\_ACTIVE

类似，它也可用来检查事务的存在与否，以便调试或优化。使用有关的切面或拦截器（或内联式代码），我们能够监测到事务设计策略可能的漏洞，以便修改。下面例子代码说明了 STATUS\_NO\_TRANSACTION 的用法：

```
...
if (txn.getStatus() == Status.STATUS_NO_TRANSACTION)
throw new Exception(
"Transaction needed but none exists");
...
```

注意我们不能简单地检查状态是否等于 STATUS\_ACTIVE 来确定事务上下文的存在，因为状态不等于 STATUS\_ACTIVE 并不意味着事务上下文不存在——事务上下文也可能处于以上列表中的任何其他状态。

## 总结

后续三章将聚焦于三种不同的事务模型。本书主要使用 EJB 作为例子来阐述这些概念，我也提供了一些 Spring 的例子。虽然不同的持久性框架设计到的所有完整的例子不再本书涵盖范围之内，我强烈建议读者通过阅读本书去理解有关这些事务模型的各种概念、技巧、陷阱和最佳实践，而后查阅您所专注的持久化框架的文档，去进一步了解实现细节。



# 全球软件开发大会（北京站）2012

QCon Beijing 2011 International Software Development Conference

数据密集型实时应用 (DIRT)

缓存、NoSQL和网格计算——银行能教给我们什么？

云计算技术在美国宇航局的应用  
——银行能教给我们什么？

敏捷软件开发怪诞行为学

将Node.js用于数据密集型实时应用 (DIRT)

软件研发，不仅仅是持续集成

利用云技术实现Netflix快速规模化增长

从创意到盈利：产品成功背后的奥秘

3月31日前

享 折优惠 = 3780 元

团购优惠更多

4月18-20日

北京京仪大酒店

## 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：

```

public void updateTradeOrder(TradeOrderData order)
        throws Exception {
    DataSource ds = (DataSource)
        (new InitialContext()).lookup("jdbc/MasterDS");
    Connection conn = ds.getConnection();
    conn.setAutoCommit(false);
    Statement stmt = conn.createStatement();
    String sql = "update trade_order ... ";
    try {
        stmt.executeUpdate(sql);
        conn.commit();
    } catch (Exception e) {
        conn.rollback();
        throw e;
    } finally {
        stmt.close();
        conn.close();
    }
}
    
```

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：

```

public void updateTradeOrder(TradeOrderData order)
throws Exception {
    Connection conn = DataSourceUtils
        .getConnection(dataSource);
    conn.setAutoCommit(false);
    Statement stmt = conn.createStatement();
    String sql = "update trade_order ... ";
    try {
        stmt.executeUpdate(sql);
        conn.commit();
    } catch (Exception e) {
        conn.rollback();
        throw e;
    } finally {
        stmt.close();
        conn.close();
    }
}
    
```

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：

```

<bean id="datasource"
      class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MasterDS"/>
</bean>
<bean id="TradingService"
      
```

```

class="com.trading.server.TradingService">

    <property name="dataSource">
        <ref local="datasource"/>
    </property>
</bean>

```

对于本章随后的代码示例，如果读者习惯于使用 Spring，可以将数据源查找部分和 getConnection() 方法部分替代为上面的代码。

## 自动提交和连接 ( Connection ) 管理

无论您使用 EJB 还是 Spring，自动提交标志 ( auto commit flag ) 在本地事务模型中都非常重要。缺省地这个标志常常设置为 true，表示每条更新的 SQL 语句执行之后 DBMS 会提交 ( 或者回滚 ) 有关连接。可参考下面的例子代码。代码中有一条单独的 SQL 更新语句，但没有任何编程式的连接管理。因为自动提交标志缺省是设置为 true 的，底层 DBMS 会管理连接，提交或者回滚有关更改。

```

public void updateTradeOrder(TradeOrderData order)
    throws Exception {
    DataSource ds = (DataSource)
        (new InitialContext()).lookup("jdbc/MasterDS");
    Connection conn = ds.getConnection();
    Statement stmt = conn.createStatement();
    String sql = "update trade_order ... ";
    try {
        stmt.executeUpdate(sql);
    } catch (Exception e) {
        throw e;
    } finally {
        stmt.close();
        conn.close();
    }
}

```

```
}
```

上面的代码是可以正常工作的，因为方法中仅有一条产生更新的 SQL 语句。然而，当我们假设同一方法中有多条更新语句执行的情形，像下面这样，情况就有所变化了：

```
public void updateTradeOrder(TradeOrderData order)
    throws Exception {
    double fee = calculateFee(order);
    DataSource ds = (DataSource)
        (new InitialContext()).lookup("jdbc/MasterDS");
    Connection conn = ds.getConnection();
    Statement stmt = conn.createStatement();
    String sqlOrder = "update trade_order ... ";
    String sqlTrade = "update trade_fee ... ";
    try {
        stmt.executeUpdate(sqlOrder);
        stmt.executeUpdate(sqlTrade);
    } catch (Exception e) {
        throw e;
    } finally {
        stmt.close();
        conn.close();
    }
}
```

在此例中，我们加入了更多的代码，重新计算有关交易的费用（fee），而后更新有关table以记录下与此交易单据（trade order）相关联的新的费用。虽然这样编写代码可以成功编译和执行，但却不符合ACID特性。首先，因为自动提交标志是缺省设为true的，在第一个executeUpdate()执行后，连接将被提交。如果第二个executeUpdate()语句失败了，整个方法将会抛出异常，但第一个SQL语句被提交的事实不会被改变，因此违反了ACID中的原子性原则。其次，这两个语句造成的更新操作，作为一个逻辑工作单元（LUW，Logic Unit of Work），

并未与操作同一个table或同一些行（row）的其他处理过程相隔绝<sup>\*</sup>，因此违反了ACID中的独立性原则。

出于事务和逻辑工作单元的观点，要让上面的代码正常的工作，我们必须将自动提交标志设置为 false，然后为代码加入提交和回滚的逻辑。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：

```

public void updateTradeOrder(TradeOrderData order)
    throws Exception {
    double fee = calculateFee(order);
    DataSource ds = (DataSource)
        (new InitialContext()).lookup("jdbc/MasterDS");
    Connection conn = ds.getConnection();
conn.setAutoCommit(false);
    Statement stmt = conn.createStatement();
    String sqlOrder = "update trade_order ... ";
    String sqlTrade = "update trade_fee ... ";
    try {
        stmt.executeUpdate(sqlOrder);
        stmt.executeUpdate(sqlTrade);
conn.commit();
    } catch (Exception e) {
        conn.rollback();
        throw e;
    } finally {
        stmt.close();
        conn.close();
    }
}

```

---

<sup>\*</sup> 在两次提交的间隙，很可能关于同一个表的更新或查询语句在兄弟连接中被执行——译者

```

    }
}
}
```

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：

```

public void updateTradeOrder(TradeOrderData order)
    throws Exception {
    OrderDAO orderDao = new OrderDAO();
    TradeDAO tradeDao = new TradeDAO();
    try {
        //在 DAO 类中的 SQL 和 Connection 逻辑
        orderDao.update(order);
        tradeDao.update(order);
    } catch (Exception e) {
        logger.fatal(e);
        throw e;
    }
}
```

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。

有的开发人员可能会通过类似连接传递（connection passing）的技术尝试解决这个问题。在连接传递方式下，我们在更上一层的方法中建立数据库连接，而后将连接对象当作参数传入 DAO 的 update()方法。使用连接传递技术我们能够更改上面的代码示例以传递连接，是这些代码符合 ACID 的准则。如下面代码段所示：

```

public void updateTradeOrder(TradeOrderData order)
    throws Exception {
```

```

DataSource ds = (DataSource)
        (new InitialContext()).lookup("jdbc/MasterDS");

Connection conn = ds.getConnection();
conn.setAutoCommit(false);

OrderDAO orderDao = new OrderDAO();
TradeDAO tradeDao = new TradeDAO();

try {

    //在 DAO 类中的 SQL 和连接逻辑

    orderDao.update(order, conn);
    tradeDao.update(order, conn);
    conn.commit();

} catch (Exception e) {
    logger.fatal(e);
    conn.rollback();
    throw e;
}

} finally {
    conn.close();
}
}
    
```

注意在上面的代码中，我们之前放置在 DAO 对象中的连接逻辑被移动到产生调用的方法那里了。该连接被传递到 DAO，而后用于获得 statement 对象，执行更新，而后返回调用者。

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

时。

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。并且，如果目标方法涉及到管理连接，开发人员在调用前必须非常仔细地审查这些方法。除非您的应用仅仅是简单地涉及到单表更新（single-table updates），没有一种有效的手段可以为您保证对单个请求总是产生单一的事务工作单元。

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）：

```

public void updateTradeOrder(TradeOrderData order)
    throws Exception {
    UserTransaction txn = sessionCtx.getUserTransaction();
    txn.begin();
    try {
        TradeOrderDAO dao = new TradeOrderDAO();
        dao.updateTradeOrder(order);
        txn.commit();
    } catch (Exception e) {
        log.fatal(e);
        txn.rollback();
        throw e;
    }
}
    
```

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。

虽然通常不鼓励使用编程式事务模型，在某些情况下它还是非常有用的。这些情形将在 3.4

节中介绍。如果我们打算要在 EJB 环境中使用编程式事务，我们需要在 ejb-jar.xml 部署描述文件中将<transaction-type>节点的值设置为 Bean，以告知应用服务器（准确的说是 EJB 容器）我们想自行管理事务。通常<transaction-type>节点的默认值是 Container，表示会使用声明式事务（声明式事务将在下一章讨论）。选择使用编程式模型还是声明式模型是在每个 bean 的粒度设置的。这意味着您可以混合使用两种模型。虽然，这是一种应该避免的不良方式。

在 EJB 3.0 中，您可以使用元数据标注（metadata annotations）的方式指明何处使用编程式事务管理，即像下面的代码那样使用@TransactionManagement 标签：

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class TradingServiceBean implements TradingService
{
    ...
}
```

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：

```
public void updateTradeOrder(TradeOrderData order)
    throws Exception {
    transactionTemplate.execute(new TransactionCallback() {
        public Object doInTransaction(
            TransactionStatus status) {
            try {
                TradeOrderDAO dao = new TradeOrderDAO();
                dao.updateTradeOrder(order);
            } catch (Exception e) {
                status.setRollbackOnly();
                throw e;
            }
        }
    });
}
```

```

        }
    });
}

```

```

<bean id="transactionTemplate"
      class="org.springframework.transaction.support.
      TransactionTemplate">
    <property name="transactionManager">
      <ref local="transactionManager"/>
    </property>
</bean>
<bean id="tradingService"
      class="com.trading.server.TradingService">
    <property name="transactionTemplate">
      <ref local="transactionTemplate"/>
    </property>
</bean>

```

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。

## 获取到 JTA UserTransaction 的引用

当在客户端（无论 web 客户端或应用客户端）使用基于 EJB 的编程式事务模型时，您在使用 EJB 时必须获得 InitialContext，并像下面的代码一样进行一次 JNDI 查找：

```

...
InitialContext ctx = new InitialContext()
UserTransaction txn = (UserTransaction)
ctx.lookup("javax.transaction.UserTransaction")

```

...

在这里事情变得稍微有点复杂，以上代码片段中的查找名称

( "javax.transaction.UserTransaction" ) 是与应用服务器类型相关的。这意味着基本上客户端里的这部分代码在不同的应用服务器上是不通用的。不同的应用服务器将 UserTransaction 绑定给不同的 JNDI 名。下表列出了最流行的几种应用服务器上的 JNDI 绑定：

JBoss	"UserTransaction"
WebLogic	"javax.transaction.UserTransaction"
WebSphere v5+	"java:comp/UserTransaction"
WebSphere v4	"jta/usertransaction"
SunONE	"java:comp/UserTransaction"
JRun4	"java:comp/UserTransaction"
Resin	"java:comp/UserTransaction"
Orion	"java:comp/UserTransaction"
JOnAS	"java:comp/UserTransaction"

看看更复杂的情况，让我们假设我们将要访问一个使用声明式事务的 EJB，客户端在容器外（非基于 web 的客户端），或客户端是一个 JUnit 测试用例。我们不能使用下面的代码，因为大多数情况下 JNDI 绑定对容器外环境都是不可用的：

```
UserTransaction txn = (UserTransaction)
ctx.lookup("javax.transaction.UserTransaction")
```

这为我们打算从 JUnit 测试 EJB，或从基于 Swing 的应用访问事务带来了困难。解决这个问题的方法是使用 TransactionManager 接口启动事务。首先，您需要使用 Class.forName()方法装载应用服务器相关的 TransactionManagerFactory 类。然后，使用反射调用 getTransactionManager()方法。这样，便产生了一个 TransactionManager 实例，以用于开始和结束事务。下面的代码片段阐述了技术细节，使用 IBM WebSphere 中的 TransactionManagerFactory：

```
...
//装载 transaction manager 工厂类
```

```

Class txnClass = Class.forName(
    "com.ibm.ejs.jts.jta.TransactionManagerFactory");

//使用反射调用 getTransactionManager()

//方法拿到对 transaction manager 的引用
TransactionManager txnMgr = (TransactionManager)
    txnClass.getMethod("getTransactionManager",null)
        .invoke(null, null);

//通过 transaction manager 开始事务
txnMgr.begin();

...

```

虽然上面的例子是为 IBM WebSphere 的 TransactionManager 而做的，读者可以将类似的代码用于任何应用服务器，将 Class.forName() 中的类名参数替换为特定的应用服务器的有关实现即可。这段代码可以加在 JUnit 的测试用例中，应用测试客户端里，甚至放到需要从容器外环境控制事务的 Swing 应用也可以。

对 EJB 来说，您可以在会话 Bean 中使用 SessionContext.getUserTransaction()，或在消息驱动 Bean 中使用 MessageDrivenContext.getUserTransaction() 来获得 UserTransaction 实例的引用。这种方式并不需要任何类型转换。下面的代码片段演示了如何从 SessionContext 中获取一个 UserTransaction 接口：

```

...
UserTransaction txn = sessionCtx.getUserTransaction();
...
```

## 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）：

```
public void updateTradeOrder(TradeOrderData order)
```

```

throws Exception {

UserTransaction txn = sessionCtx.getUserTransaction();
txn.begin();
try {
    TradeOrderDAO dao = new TradeOrderDAO();
    dao.updateTradeOrder(order);
    txn.commit();
} catch (ApplicationException e) {
    log.fatal(e);
    txn.rollback();
    throw e;
}
}

```

如果我们碰到了一个运行时异常（例如 `NullPointerException` 或 `ClassCastException`），容器会为我们自动处理它，并回滚事务。然而，如果我们忘记处理任何应用异常或运行时异常，像下面代码那样，将会产生什么结果呢？

```

public void updateTradeOrder(TradeOrderData order)
throws Exception {
    UserTransaction txn = sessionCtx.getUserTransaction();
    txn.begin();
    TradeOrderDAO dao = new TradeOrderDAO();
    dao.updateTradeOrder(order);
    txn.commit();
}

```

此时我们通过在方法签名中使用 `throws Exception`，将所有异常的处理权交给本方法的调用者。如果运行中上面的代码发生了应用异常，下面的异常信息将会被抛出：

```

java.lang.Exception: [EJB:011063] Stateless session
beans with bean-managed transactions must commit or
rollback their transactions before completing a business method.

```

也就是说，一旦使用编程式事务，我们开发人员便有责任去管理事务。一个方法，启动了事务，它就必须负责终止该事务。因为上面的异常本身是一个运行时异常，这种错误写法常常不会在测试中被暴露，特别对于复杂的系统更是如此。

现在假设我们在下面的代码中，忘记在编写事务逻辑时调用 commit()方法：

```
public void updateTradeOrder(TradeOrderData order)
    throws Exception {
    UserTransaction txn = sessionCtx.getUserTransaction();
    txn.begin();
    try {
        TradeOrderDAO dao = new TradeOrderDAO();
        dao.updateTradeOrder(order);
    } catch (ApplicationException e) {
        log.fatal(e);
        txn.rollback();
        throw e;
    }
}
```

当我们执行这段代码，我们会再次得到下面的错误信息：

```
java.lang.Exception: [EJB:011063] Stateless session beans
with bean-managed transactions must commit or rollback
their transactions before completing a business method.
```

如读者从上面的例子中可以看到，当使用编程式事务时，开发人员必须对关于异常处理的部分多加注意。开发人员必须确保在启动事务的同一个方法内去终止事务。这点常常是说来简单做来难，特别是当应用庞大、复杂，并且夹杂着复杂的异常处理时，很难 100% 保证做到。

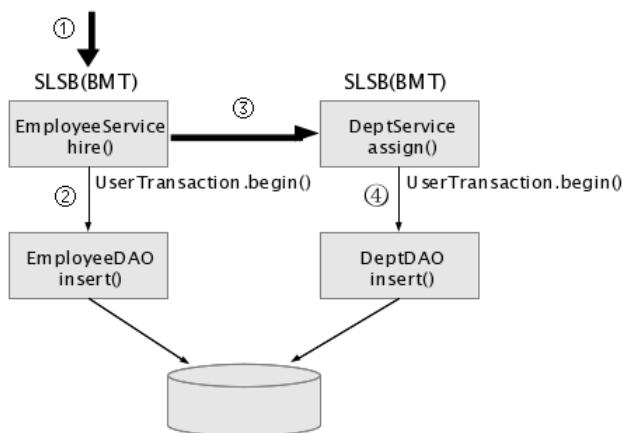
### 事务上下文问题 ( Transaction Context Problem )

在 EJB 中（特别是无状态会话 Bean），有一个关于编程式事务的非常严重的架构缺陷，我称之为事务上下文问题。事务上下文问题是指这样的限制：您不能从正在使用编程式事务的一个 bean 中将事务上下文传递到使用编程式事务的另外一个 bean 中。然而，您可以将源自

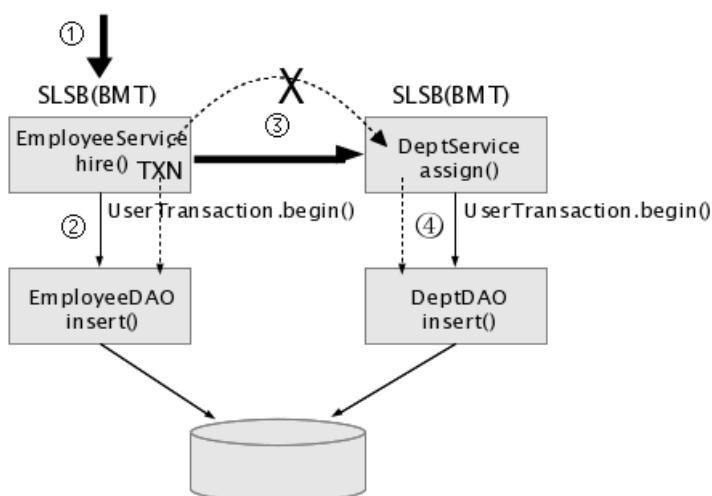
EJB 的，或客户端对象的，使用编程式事务的事务上下文传递到使用声明式事务的 EJB 中。

假设这样的场景，我们有两个无状态会话 bean，EmployeeService 和 DeptService。

EmployeeService 处理所有有关雇员的功能需求（例如雇佣一名雇员，或给一名雇员升职）。而 DeptService 处理所有有关部门的功能需求（添加、移除雇员，指派职务）。这两个 EJB 都使用编程式事务。我们进一步假设这两个 EJB 都具有相关的数据访问对象（DAO）。下图说明了这一场景：



当在无状态会话 Bean 中使用编程式事务，进行服务间通信时，“事务上下文问题”便出现了。考虑上例描述的问题，当雇佣了一名雇员，我们需要同时为该雇员指派为一个特定部门下的职务。因此，`EmployeeService.hire()`方法在同一个业务事务中调用了 `DeptService.assign()`方法。我们期望发生的是，事务在 `hire()`方法中开始，其上下文能够传播到 `assign()`方法中，使得两个操作能够在同一个事务中完成。然而，这是不可能实现的，因为在这样的事务模型下，我们不能向 `assign()`方法传播事务。还有，`assign()`方法在另一个线程中运行，因为它能够被当做独立的服务单独调用，它使用 `UserTransaction.begin()`开启自己的事务。下图阐明了这个问题：



非常致命的是，除非开发人员特别地去构造测试用例，事务上下文问题不会主动浮现。在上面的场景中真正发生的是：

- hire()方法启动一个事务，通过 EmployeeDAO 对雇员表执行必要的 insert 操作。
- 而后，hire()方法查找获取 DeptService 服务，并执行它的 assign()方法。
- 属于不同 SLSB 的 assign()方法，在另一个线程中开始，并开启它自己的事务。此时 hire()方法启动的事务会处于等待状态，直到 assign()方法结束。
- 一旦 assign()方法完成，该方法开动的事务被提交（从而事务终止）。
- 控制被交回 hire()方法，原有线程和事务继续。

在这样的场景中，我们不能维护 ACID 中的原子性和独立性原则，因此谈不上正确的事务管理。如果 hire()方法启动的事务在 assign()方法调用完成后回滚了，因 assign()方法包含的事务已经提交，数据库将被置于一个不一致的状态。

## 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（client-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（ Declarative Transaction 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。有时，EJB 中的声明式事务也被称为容器管理事务（ Container-Managed Transactions , CMT ），但大多数人倾向于使用“声明式事务”这一术语，以体现事务模型的框架无关性。

从 Java 编码的观点看，我们在使用 EJB 声明式事务时，需编码的仅仅只有 setRollbackOnly() 方法。该方法告诉容器，当前事务产生的唯一可能结果便是回滚所有更改，不管后续有什么样的处理逻辑。下面的例子展示了 EJB 中 setRollbackOnly() 方法的使用。注意，字体加粗的部分是参与事务管理仅有的 Java 代码：

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void updateTradeOrder(TradeOrderData order)
    throws Exception {
    try {
        TradeOrderDAO dao = new TradeOrderDAO();
        dao.updateTradeOrder(order);
    } catch (Exception e) {
        sessionCtx.setRollbackOnly();
        throw e;
    }
}
```

在 Spring 中您不用显示地去写 setRollbackOnly() 方法。替代之，您可以在 TransactionAttributeSource 拦截器中指明何时回滚事务的规则。我们将在本节稍后的部分看到有关示例。

使用 EJB 时，我们通过设置 ejb-jar.xml 部署描述文件中的<transaction-type> 节点值为 Container 去告知应用服务器我们将使用声明式事务。这通常不用显示配置，因为它是大多数应用服务

器的默认值。

对于 EJB3.0，可以使用元数据标注的方式指明要使用声明式事务，这个标注是 @TransactionManagement，像下面这样：

```

@Stateless
@TransactionManagement(
    TransactionManagementType.CONTAINER)
public class TradingServiceBean implements TradingService
{
    ...
}

```

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：

```

<bean id="tradingServiceTarget"
      class="com.trading.server.TradingServiceBean">
    ...
</bean>

<bean id="tradingService"
      class="org.springframework.transaction.interceptor.
          TransactionProxyFactoryBean">
    <property name="transactionManager" ref="txnMgr"/>
    <property name="target" ref="tradingServiceTarget"/>
    <property name="transactionAttributes">
        <props>
            <prop key="*">PROPAGATION_SUPPORTS</prop>
            <prop key="update*">
                PROPAGATION_REQUIRED,-Exception
            

```

```
</bean>
```

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？EJB 里 ejb-jar.xml 部署描述文件中的事务属性（对于 Spring，则位于 TransactionAttributeSource 中），告知容器如何管理 JTA 事务。有六种事务属性的设置：

- Required（需要）
- Mandatory（强制必须）
- RequiresNew（需要新的）
- Supports（支持）
- NotSupported（不支持）
- Never（不用）

除此之外 Spring 还有一个附加的属性 PROPAGATION\_NESTED，该属性告知 Spring 进行事务嵌套，并采用 Required 属性。当然，如果要使用这个设置，底层事务服务实现必须要能支持嵌套事务。虽然以上列出的事务属性能被应用于 bean 对象级别，通常我们会将它们应用在 bean 的方法级别。当事务属性被应用在 bean 级别，bean 中所有的方法都被赋予相同的事务属性。如果有方法特别的应用了其他事务属性，方法级别的属性会被使用，而对象级别的属性指派在此方法就不起作用了。（本章稍后可看到细节）

### REQUIRED

Required 属性（对应 Spring 中的 PROPAGATION\_REQUIRED 属性）告诉容器，目标对象方法需要一个事务。如果事务上下文已经存在，容器将会使用它；否则，容器将为此方法开启一

个新事务。这是最常用到的事务属性，大多数开发人员都会使用它进行事务处理。然而，正如我们将会在后续章节看到的，有时候我们有更好的理由使用 Mandatory 属性。

### MANDATORY

Mandatory 属性（对应 Spring 中的 PROPAGATION\_MANDATORY 属性）告知容器，特定的对象方法需要一个事务。然而，和 Required 属性不同的是，此属性标注后，容器并不为目标对象开启新的事务。当使用这个事务属性时，当方法被调用时，一个事先存在的事务上下文必须存在。如果此事务上下文不存在，容器将会抛出一个 TransactionRequiredException，宣告它需要一个已存在的事务，但此事务没有找到。

### REQUIRESNEW

RequiresNew 属性（对应 Spring 中的 PROPAGATION\_REQUIRES\_NEW 属性）告知容器，当对象方法被调用时，“总是”需要开启一个新的事务。如果先于方法调用前一个事务已经开启了，此事务将被暂时挂起，容器启动一个新的事务。当这个新事务随着方法调用完成终止后，老的事务将会继续。在调用前已有事务上下文建立的情况下，使用 RequiresNew 属性违背了事务的 ACID 特性。这是因为直到新的事务结束时，原有事务一直处于被挂起的状态。

一个事务如果需要与其外围包裹事务（surrounding transaction）相独立，不受其执行结果的影响，自行完成（也就是提交）时，这个属性相当有用。一个典型的例子便是记录审计日志。例如，在大多数交易系统中，无论任务成功与否，每个操作都必须要求要记录日志。看看商家进行股票交易的例子，假设一个事务从 placement() 方法调用开始。该方法调用了另一个 EJB 提供的，通用的 audit() 方法去记录每一次尝试动作到数据库中。如果 audit() 方法和外面的 placement() 方法是在同一个事务中，当 placement() 因其中业务操作发生问题回滚时，audit() 也回滚了。这就与我们“无论任务成功与否，每个操作都必须要求要记录日志”的初衷相悖了。因此，通过设定 audit() 方法的事务属性为 RequiresNew，便能够保证 audit() 方法总能够提交，无论其外围 placement() 方法操作后果如何。

### SUPPORTS

Supports 属性（对应 Spring 中的 PROPAGATION\_SUPPORTS 属性）告知容器，对象方法并不需要一个事务上下文，但当调用到这个方法而事务上下文碰巧存在时，该方法会使用它。Supports 属性在事务管理中很强大很有用。考虑这样的例子，一个获取某商家的所有交易的简单数据库查询，执行这个操作并不一定需要事务存在。于是，使用 Supports 事务属性，告知容器在调用此方法时不必开启一个事务。然而，如果此查询是在一个正在进行的事务中完成的，对目标方法应用 Supports 属性会让容器使用当前事务上下文，参考数据库操作记录，从而将事务中作出的各种修改也包括到查询结果中。

为了演示 `Supports` 属性的使用，我们假设这样的场景，一个商家在每天最多只能进行 100 万股的股票交易。使用 `Support` 属性，会造成下面的事情发生：

- 第一步：当天，迄今已经交易了 90 万股
- 第二步：事务开始
- 第三步：商家操作一笔数额为 20 万股的股票交易
- 第四步：系统通过使用 `Supports` 事务属性的查询，得知今天的交易总额已达 110 万股
- 第五步：违背交易上限的异常抛出，事务回滚

如果我们使用（下文将要介绍到的）`NotSupported` 属性，我们便不能得到这个异常。因为查询不在此事务的范围之内，并不能够看到试图进行的交易：

- 第一步：当天，迄今已经交易了 90 万股
- 第二步：事务开始
- 第三步：商家操作一笔数额为 20 万股的股票交易
- 第四步：系统通过使用 `NotSupported` 事务属性的查询，得知今天的交易总额还是 90 万股
- 第五步：允许交易，交易提交完成

#### NOTSUPPORTED

`NotSupported` 属性（对应 Spring 中的 `PROPAGATION_NOT_SUPPORTED` 属性）告知容器，被调用的对象方法不使用事务。如果一个事务已启动，容器会将此事务暂停直至方法调用结束。如果调用方法时没有事务存在，容器也不会为此方法开启任何事务。在方法逻辑中有排斥事务上下文的代码存在时，这个属性就非常有用了。例如，在 XA 事务的上下文中调用包含 DDL 的存储过程便会导致异常。如果没有办法去修改这个存储过程，使用 `NotSupported` 事务属性便是一个绕开问题的方法了。`NotSupported` 属性会让在执行包含该存储过程的代码的方法前暂停事务，待其完成后再继续。

#### NEVER

`Never` 属性（对应 Spring 中的 `PROPAGATION_NEVER` 属性）告知容器，在调用对象方法时，

不允许有事务上下文存在。要注意它和 NOTSUPPORTED 属性的区别。当使用 NotSupported 属性时，如果方法调用之前事务存在，容器会暂停此事务以执行方法，方法在没有事务上下文的环境下执行；而使用 NEVER 属性时，如果方法调用前事务存在，容器直接会抛出异常，告知您这个方法是决不能和事务有任何关系的。使用 Never 属性会造成一些不可预期和不希望发生的运行时异常，因此，在没有确定绝对必要时，最好不使用它。事实上，也没有太多的需求需要使用这个属性去构造事务解决方案。当我们为此事纠结时，选择 NOTSUPPORTED 属性也许就够了。

以上这些属性在使用上是有一些限制的。根据 EJB 规范，对于实体 Bean ( Entity Bean )，只有 Required ,Mandatory 和 RequiresNew 等三个属性能够使用。消息驱动 Bean( Message Driven Bean )只能使用 Required 和 NotSupported 属性。如果 EJB 实现了 Synchronization 接口，那它也只能使用 Required , Mandatory 和 RequiresNew 等三个属性。Synchronization 为 JTA 事务提供了 afterBegin() , beforeCompletion() 和 afterCompletion() 三个回调方法。这在开发人员需要在这些事件的发生点时做一些操作，但没有事务管理的控制权时非常有用。例如，开源对象关系映射 ( object-relational mapping , ORM ) 和持久化框架 hibernate ( <http://www.hibernate.org> ) 在它的事务处理过程中使用 Synchronization 接口，将 JTA 事务管理器集成，在 afterCompletion() 回调中调用 Session.afterTransactionCompletion() 方法，用于清除一级和二级缓存的对象锁。

## 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。对于 EJB，事务属性的值在 ejb-jar.xml 部署描述文件中配置，如下面 xml 的例子：

```
...
<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>TradingService</ejb-name>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Mandatory</trans-attribute>
  </container-transaction>
```

...

注意，TradingService EJB 的所有方法都被配置为 Mandatory 的事务属性（由 <method-name>\*</method-name> 指定）。我们通常不推崇这样的写法，因为它没有对那些查询方法做优化。例如，一个 getAllTraderByID() 方法并不需要事务，但由于上面的 Mandatory 属性设定，它总是被附加一个事务。

在 EJB 3.0 中，您可以使用元数据标注，像下面这样指定事务属性：

```
@Stateless
@TransactionAttribute(TransactionAttributeType.MANDATORY)
public class TradingServiceBean implements TradingService
{
    ...
}
```

为简单起见，我将在随后的 EJB 代码例子中普遍使用 EJB 3.0 的标注风格。如果您是 EJB 2.x 的用户，将对应的之前所述的 XML 简单替代元数据标注的部分即可。

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*

```
<bean id="transactionAttributeSource"
    class="org.springframework.transaction.interceptor.
        NameMatchTransactionAttributeSource">
    <property name="properties">
        <props>
            <prop key="*">>PROPAGATION_MANDATORY</prop>
        </props>
    </property>
</bean>

<bean id="tradingService"
```

```

class="org.springframework.transaction.interceptor.

    TransactionProxyFactoryBean">

<property name="transactionAttributeSource">
    <ref local="transactionAttributeSource">
</property>

...
</bean>

```

## 技巧2. 使用 *TransactionProxyFactoryBean*

```

<bean id="tradingService"
    class="org.springframework.transaction.interceptor.

        TransactionProxyFactoryBean">

<property name="transactionManager" ref="txnMgr"/>
<property name="target" ref="tradingServiceTarget"/>

<property name="transactionAttributes">
    <props>
        <prop key="*">>PROPAGATION_MANDATORY</prop>
    </props>
</property>

</bean>

```

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。最佳实践描述如下：

### 最佳实践

当给方法指派事务属性时，最好的做法是为类级别指派所需要的最严格的事务属性，而后在方法的级别上按需微调。

我们能将这条实践应用于之前的 XML 部署描述文件实例，在 EJB 中将所有方法的事务默认值配置为 Mandatory，而将查询方法配置为 Supports：

```

...
<assembly-descriptor>
    <container-transaction>
        <method>
            <ejb-name>TradingService</ejb-name>
            <method-name>*</method-name>
        </method>
        <trans-attribute>Mandatory</trans-attribute>
    </container-transaction>
    <container-transaction>
        <method>
            <ejb-name>TradingService</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>getAllTradersByID</method-name>
        </method>
        <trans-attribute>Supports</trans-attribute>
    </container-transaction>
...

```

在 Spring 中，我们这样去应用这个实践：

```

<bean id="tradingService" ...>
    ...
    <property name="transactionAttributes">
        <props>
            <prop key="*">PROPAGATION_MANDATORY</prop>
            <prop key="getAllTradersByID">
                PROPAGATION_SUPPORTS
            </prop>
        </props>
    </property>

```

```

        </props>
</property>
</bean>
```

正如我们在本章开头看到的，EJB 3.0 使用了元数据标注，简化了事务属性的配置。由于事务属性标注可以应用在类和方法的级别，我们能如此应用最佳实践：

```

@Stateless
@TransactionAttribute(TransactionAttributeType.MANDATORY)
public class TradingServiceBean implements TradingService {
    ...
}

@TransactionAttribute(TransactionAttributeType.SUPPORTS)
public Collection getAllTradersByID() {
    ...
}
```

在类级别应用最严格的事务属性通常是较为安全的做法，因为这样可以保证，当开发人员忘记为一个新的方法指定事务属性时，该方法被默认指派一个事务。举例来说，我们将上面的例子做相反的处理，将（类级别）缺省属性调整为 Supports，如果加入一个新的更新方法而没有给它指派任何事务属性，在调用到这个更新方法时，可能没有任何事务上下文存在。这样的情形常常是难以侦测的。

## 异常处理和 setRollback()方法

在使用声明式事务模型时，虽然应用容器或框架负责管理了事务，在 EJB 中，我们还是需要使用少量的 Java 代码来确保声明式事务正确的工作。看看下面的固定收益交易的例子，买卖一笔债券或国库券：

```

@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void placeFixedIncomeTrade(TradeData trade)
    throws Exception {
    try {
```

```

...
Placement placement =
    placementService.placeTrade(trade);
    executionService.executeTrade(placement);
} catch (TradeExecutionException e) {
    log.fatal(e);
    throw e;
}
}

```

如是，我们在单一业务工作单元中进行了多次更改操作。如果在运行这个方法的过程中产生了一个 TradeExecutionException，应用容器会在将异常之前做完的所有工作提交，并将异常传播给调用者。这个例子要说明的是，“容器不会在应用在发生异常时自行将事务标记为回滚”。使用声明式事务而忘记处理应用异常是很多应用数据完整性问题的根源。

为了解决这个问题，我们必须在应用发生异常（被检查异常）时告知容器回滚事务。在 EJB 中，需要使用 EJBContext 接口的 setRollback()方法；在 Spring 中，则需要在 TransactionAttributeSource 的 bean 定义属性中申明回滚规则。使用这样的方法并非会直接回滚事务，它仅仅是通知容器，当前事务唯一产生的后果便是回滚。这个方法一旦被调用，致使事务被回滚的动作便不可反悔了。以下列出了正确的代码：

```

@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void placeFixedIncomeTrade(TradeData trade)
    throws Exception {
    try {
        ...
        Placement placement =
            placementService.placeTrade(trade);
            executionService.executeTrade(placement);
    } catch (TradeExecutionException e) {
        log.fatal(e);
    }
}

```

```
sessionCtx.setRollbackOnly();

throw e;

}

}
```

当我们加入了 `setRollbackOnly()` 方法的调用，在应用异常发生时，应用容器会正确的回滚之前所有对数据库的更新。

尽管我们在应用中引入了附加的编码工作，我们也必须要记住容器在应用异常时不会自动回滚事务，这是非常重要的。举例来说，在处理业务的过程中我们如果要发送一份 email 确认。如果 SMTP 服务器无法访问，SMTP 服务可能会抛出一个应用异常（为被检查异常）。在大多数情况下，我们并不希望容器在无法发送邮件的情况下就回滚整个事务。捕获这个异常，控制事务逻辑是更好的做法，在此我们决定是否继续事务还是将其回滚。

在您将自己的企业级 Java 应用做总体事务设计时，需要做出的一个决定是在什么地方和什么时机去调用 `setRollbackOnly()` 方法。以下列出了针对这一问题的最佳实践：

### 最佳实践：

事务管理应在开启事务的方法中进行。因此，开始事务的业务方法具备调用 `setRollbackOnly()` 的责任。

当事务上下文能够被开启事务的方法传递给被调用到的其他 bean 或方法时，该实践能得到最好的应用。这种情况通常发生在事务性 bean 或包含多层事务性 bean 的应用在进行服务间通信时（即会话门面层和持久化管理层之间的通信）。这个最佳实践的道理来自两方面：第一，基于一个简单组件责任模型，事务管理的任务应该被包括在启动事务的方法内，从管理的角度看，如果将事务管理责任分散到应用范围，则增加了复杂度同时也降低了可维护性；第二，一旦 `setRollbackOnly()` 方法被调用，它便不可回复，这意味着启动事务的方法（也就是管理事务的方法）不能够就该方法的“已调用既成事实”采取任何补救措施来继续事务的处理。

该最佳实践的必然推论之一是，不负责将事务标记为回滚的方法应该在抛出异常时提供足够的信息，以便开启事务的方法能够就调用 `setRollbackOnly()` 方法与否做出正确的决定。一个很好的例子是，在 SMTP 服务器不可用时，抛出一个被检查异常，以标识确认邮件无法送达。在这种情况下，启动事务的方法可以延迟通知邮件的发送，并提交事务。显然，仅仅在邮件

服务器宕机的情况下就去回滚每笔订单是非常愚蠢的。

## EJB 3.0 的考虑

EJB 3.0 规范中包含一个@ApplicationException 的元数据标注，该标注告知应用容器，当有异常抛出时是否该自动将事务标记为回滚。该标注有一个 boolean 值的参数。下面的代码展示了如何使用此标注：

```
@ApplicationException(rollback=true)
public class MySeriousApplicationException
    extends Exception {
    ...
}
```

当 MySeriousApplicationException 抛出时，容器会自动将事务标记为回滚，并不需要程序员显式的调用 setRollbackOnly()方法。这种机制好的一面是我们不再必须在应用中编写 setRollbackOnly()这样的代码了。同时，和 setRollbackOnly()方法不一样的是，容器不会在没有事务上下文的时候抛出一个 IllegalStateException 异常——它仅仅会忽略回滚事务的请求。

尽管这个标注仅仅是试图去模仿 Spring 的回滚规则处理机制，使用这一标注还是会带给您的整体事务设计策略造成重大的影响的。因为这个标注对一个“异常类”的范围起作用，该异常类又是任何方法都有可能抛出的，任何对象，无论它是否开启事务，如果它抛出这个异常，都会导致事务被标记为回滚。——这就与我们之前讨论的“仅事务开启者负责标记事务回滚”的最佳实践相悖了。如果使用这个标注，负责处理事务的方法不具备任何对回滚的补救措施。例如，如果一个关于 SMTP 邮件服务的方法抛出了含有这个标注的异常，整个事务将会由于无法发送确认邮件而回滚。更甚之，使用带有 rollback 属性的这个标注将事务管理和异常处理搅在一起，难免会带来更多问题。

### setRollbackOnly()方法的替代品

在 EJB 的世界，setRollbackOnly()方法的另一个选择是抛出一个 javax.ejb.EJBException 系统异常，强制事务回滚。这是一种优缺点都十分突出的技术。下面的代码演示了其用法：

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void placeFixedIncomeTrade(TradeData trade)
```

```

throws Exception {
try {
...
Placement placement =
    placementService.placeTrade(trade);
executionService.executeTrade(placement);
} catch (TradeExecutionException e) {
    log.fatal(e);
throw new EJBException(e.getMessage());
}
}

```

当运行时异常 ( runtime exception ) 发生时 , 容器将事务标记为回滚。这样的代码和之前使用 setRollbackOnly() 的效果相同。

这种用法的好处在于 , 避免了使用 setRollbackOnly() 方法有可能带来的运行时异常。如果我们使用 setRollbackOnly() 方法 , 并在部署描述文件将这段代码的事务属性改为 Supports , 容器会抛出 IllegalStateException 异常。 EJB 2.1 的规范这样写道 :

*“在业务方法以 Supports , NotSupported 或 Never 事务属性执行的时候 , 它如果调用到 ejbContext.setRollbackOnly() 方法 , 容器必须抛出 java.lang.IllegalStateException 异常。”*

使用 EJBException 便能够避免这一运行时异常。但这绝不是一个很好的理由 , 因为这会在我们的事务设计策略中带来潜在的问题。

使用 EJBException 的第一个问题是 EJBException 产生的日志输出没有 setRollbackOnly() 方法丰富和准确。以下是使用 setRollbackOnly() 方法时 , 重复交易订单异常产生的输出 :

```

WSCL0014I: Invoking the Application Client class
com.trading.client.ClientApp
Starting Trading Client
com.trading.common.TradingException: Duplicate Order
Duplicate Order

```

使用几乎同样的代码 , 以下是使用 EJBException 时 , 交易客户端能看到的日志输出 :

WSCL0014I: Invoking the Application Client class

com.trading.client.ClientApp

Starting Trading Client

javax.transaction.TransactionRolledbackException:

CORBA TRANSACTION\_ROLLEDBACK 0x0 Yes; nested exception

is: org.omg.CORBA.TRANSACTION\_ROLLEDBACK:

两种做法，虽然结果是一样的（事务都被回滚了），“重复订单”这一信息却在对调用者抛出 EJBException 异常的时候丢失了。

使用 EJBException 的另一个缺陷是，没有适当地将异常处理和事务管理两个话题分开。开发人员对代码一眼看去，可能不会立即意识到 EJBException 代码的目的是要将事务标记回滚。将这个逻辑移除或更改，可能使事务管理策略失效，致使原打算回滚的事务没有回滚。基于这些原因，我严重推荐使用 setRollbackOnly()方法，尽量避免利用 EJBException 系统异常来做事务管理。

### 使用 Required 和 Mandatory 事务属性的对比

什么时候使用 Required 属性，什么时候使用 Mandatory 属性，很多时候是令人混淆的。他们都提供了事务上下文，不同的是，Required 属性会在没有事务的时候启动一个新事务，而 Mandatory 属性不这样做。不论采用什么样的事务设计策略，下面的最佳实践能够帮助澄清在何时使用哪种属性：

#### 最佳实践

如果一个方法需要事务上下文，但不负责将此事务标记为回滚 (rollback only) 的状态，该方法应使用 Mandatory 事务属性。

这个最佳实践的理由基于事务的所有者。除开有状态会话 bean 的情况，开始事务的方法必须有责任去终止事务。事务的管理，包括什么时候将事务标记为回滚，必须位于事务的宿主方法之内。使用 Required 属性的一个潜在问题是，一个方法开启了事务，在发生应用异常时却没有能够在此方法内回滚事务。在客户端发起事务的情形，这种场景经常出现。如果客户端开始了事务，然后调用无状态会话 bean，远端的无状态会话 bean 不应为回滚事务负责，因为事务的发起者是客户端。因此，由于 SLSB 组件没有将事务标记为 rollback only 的责任，所以应该将此会话 bean 标记为 Mandatory 属性。假设这里的事务属性设置为了 Required，

就有这样的可能，会话 bean 的方法开始了事务，但是没有办法回滚它。

## 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。



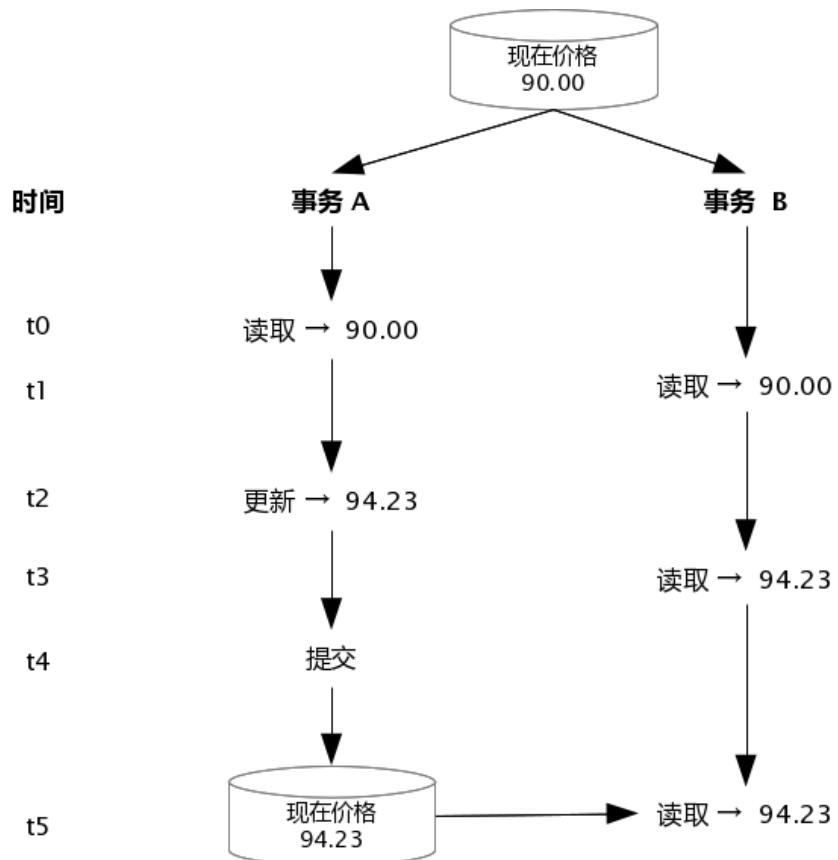
该设置能直接影响应用的性能和数据完整性。例如，对于高性能的应用，诸如信用卡处理程序，您可以通过降低隔离级别以提高并发度（但这样会损害数据完整性）。对低并发、需要高度数据完整性的金融类应用，您需要提高隔离级别，以增强整体数据一致性（但损害了性能）。大多数应用服务器和数据库都具有平衡并发度和一致性的一个默认设置值。然而，在 EJB 和 Spring 中，您能够通过更改相关设置在需要的时候优化应用。

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

### **TransactionReadUncommitted**

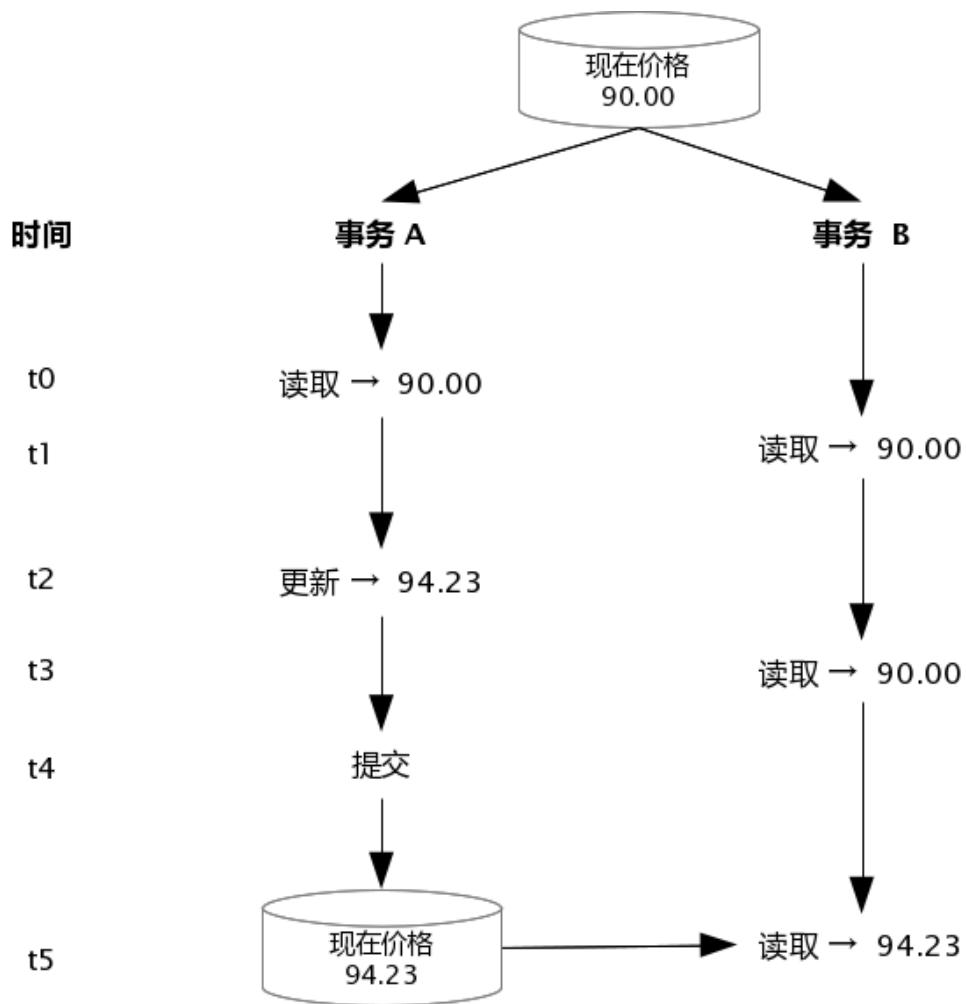
这是 EJB 和 Spring 皆支持的最低隔离度设置。这一隔离级别允许事务读取其他事务在提交到数据库之前产生的未提交更改。为了展示这一隔离级别是如何工作的，我们假设当前一笔特定的股票交易，在每股 90 美金时进行。两个事务（事务 A 和事务 B）试图同时访问同一数据，事务 A 进行更新操作，而事务 B 进行读取操作。下图展示了在设置 TransactionReadUncommitted 时两个事务是如何交互的：



注意，在图示中，事务 A 于时间点 t2 对数据库进行了一次更新，而事务 B 能够在时间点 t3 读取这一数据，读取的时候该数据的修改是并未被提交进数据库的。如图所示，事务 A 做出的修改无法与事务 B 相隔离。如果事务 A 回滚了这一修改，事务 B 读取到的数据极有可能是错误的。这一隔离级别设置违反了基本的 ACID 准则，很多数据库厂商并不支持这一隔离级别（包括 Oracle）。

#### TransactionReadCommitted

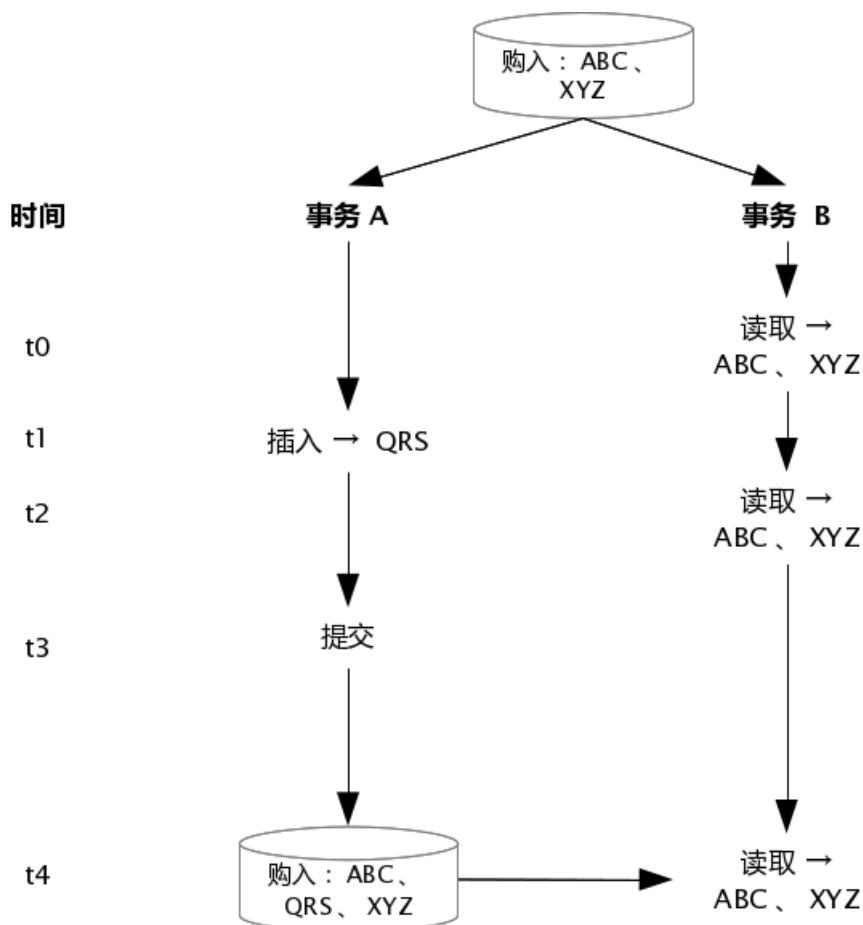
该隔离级别设置允许多个事务访问同一份数据，但将未提交的数据对其他事务隐藏，直至数据提交。同样使用上面的例子，假设当前以 90 美金的价格交易股票，两笔事务（事务 A 和事务 B）试图在同一时间访问同一数据，事务 A 做更新而事务 B 做读取。下图展示了在设置 TransactionReadCommitted 时两个事务是如何交互的：



注意，当事务 A 在时间点 t2 更改了数据，事务 B 在时间点 t3 的读取操作中并未能看到这一修改。这是一个非常好的隔离设置；它允许事务 B 访问数据（支持并发），并同时隐藏了其他事务对同一份数据的未提交更改，直至更改提交。这是大多数数据库缺省的隔离设定，很多数据库厂商都支持它。

#### TransactionRepeatableRead

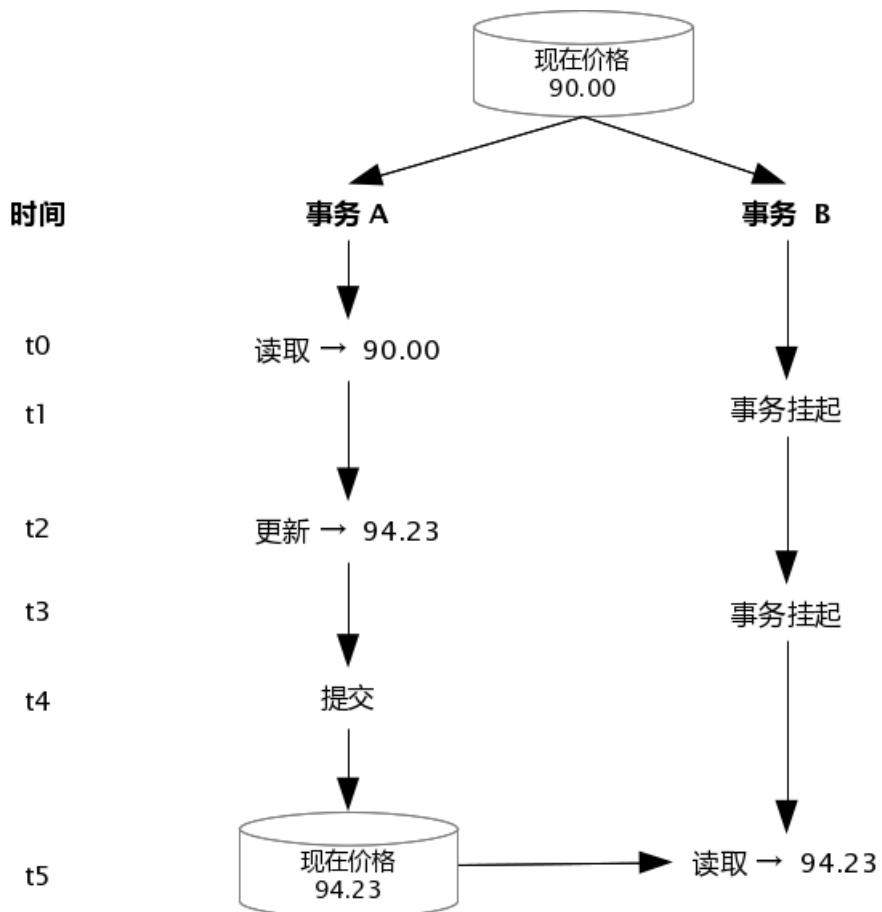
和以上让事务交错发生的策略不同，这个隔离级别保持了事务彼此隔绝。这一隔离级别保证，一旦在某一事务中读取了数据库的一个值集，在后续的每次查询操作中都读到同样的值（除非此事务拿到这些数据的读写锁，并自行更改了数据）。为了演示这个隔离级别，我们假设有一个 select 语句查询数据库，以获取今天迄今为止买入的所有股票。事务 A 在主管查询的事务 B 的进行过程中插入了一些数据。下图展示了在设置为 TransactionRepeatableRead 隔离级别时事务间是如何交互的：



注意在这个例子中，虽然事务 B 的执行过程中购入了一支新的股票，事务 B 却自始至终返回同样的查询结果，不论其他事务的最新提交结果如何。仅当事务 B 提交了之后，它才能够看到最近购入的那一笔股票。请注意在这样的隔离设置下，目标数据在被查询和更改时被同时加上读锁和写锁，使用这一隔离级别应该小心，因为在 Repeatable Read 隔离级别下，一个事务如果要更改数据，而这一数据被其他事务读取时，此事务需要等待占用数据事务提交的操作（或直接返回失败）。

#### TransactionSerializable

这是 Java 所能支持的最低的事务隔离级别。在这一隔离级别设置下，交错发生的事务被“堆迭”起来，以致同一时间点仅仅有一个事务具备访问目标数据的权力（我们在本节随后的部分可以看到，这在 Oracle 数据库不完全是成立的）。如果使用这一设置，并发度将会收到非常大的影响，但数据一致性将会得到极大提高。使用上面同样的例子，假设当前以 90 美金的价格交易股票，两笔事务（事务 A 和事务 B）试图在同一时间访问同一数据，事务 A 做更新而事务 B 做读取。下图展示了在设置 TransactionSerializable 时两个事务是如何交互的：



注意事务 B 将一直挂起，直到事务 A 完成。虽然所有的数据库厂商都支持这一设置选项，Oracle 处理该设置稍有不同。由于 Oracle 使用数据版本 ( data versioning ) 技术，它实际上并不真正挂起事务 B。反之，如果在事务 B 的进行过程中，事务 B 试图访问目标数据，Oracle 将抛出 ORA-08177 错误，指出对该事务无法序列化访问数据。

### 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。以下是 WebLogic 中设置隔离级别的例子 ( weblogic-ejb-jar.xml )：

```

...
<transaction-isolation>
    <isolation-level>
        TransactionSerializable
    </isolation-level>

```

```

<method>
    <ejb-name>TradingService</ejb-name>
    <method-inf>Remote</method-inf>
    <method-name>placeTrade</method-name>
</method>
</transaction-isolation>
...

```

以下展示了 Spring 中同样的配置：

```

<bean id="tradingService" ...>
    ...
    <property name="transactionAttributes">
        <props>
            <prop key="placeTrade">
                PROPAGATION_MANDATORY,ISOLATION_SERIALIZABLE
            </prop>
        </props>
    </property>
</bean>

```

EJB 和 Spring 的这些事务隔离设置依赖于底层数据库。也就是说，虽然编程框架支持上述种种隔离级别设定，但需要底层数据库也要支持这些级别设定，设定才会生效。如果数据库不支持某一个设定级别，而应用框架这样设置了，数据库将会采用缺省的隔离级别。例如，数据库用的是 Oracle，而 EJB 或 Spring 中将隔离级别设置为 TransactionRepeatableRead，Oracle 会拒绝这一设置转而采用 TransactionReadCommitted。不幸的是，这种情况下不会有任何异常抛出，因此您必须注意设定值和数据库实际生效值不一致的情况。并且，在更改隔离级别设置时，一定要记住这些设置的数据库相关性，不同的数据库对隔离的支持不同。

在没有充分了解到数据库的隔离能力，并且没有足够理由改变隔离设置的情况下，最好避免使用 TransactionReadCommitted 之外的其他设置。隔离级别设置在应用的整体事务设计策略中扮演非常重要的角色，但使用时需慎之又慎。

# QClub

我们影响有影响力的人

北京 上海 广州 大连 西安 太原 成都 杭州 武汉 南京 深圳...

QClub

邀请  
业内知名专家

自由开放的  
讨论氛围

定期举办的线下活动

结识  
圈内技术好友

InfoQ



中文 | 英文 | 日文 | 葡文 | .....

## 第五章 XA 事务处理

为了说明 X/Open XA 接口在 JTA 事务管理中的重要性，以及它使用的时机，我们以前一章提到的一段固定收入交易的 EJB 代码为例：

```

@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void placeFixedIncomeTrade(TradeData trade)
    throws Exception {
    try {
        ...
        Placement placement =
            placementService.placeTrade(trade);
        executionService.executeTrade(placement);
    } catch (TradeExecutionException e) {
        log.fatal(e);
        sessionCtx.setRollbackOnly();
        throw e;
    }
}
    
```

这段代码中，首先预置了一笔交易，而后执行交易，这两个操作更改了数据库中不同的表。正如我们在前面章节看到的，如果在标准的非 XA 环境，这段代码保证遵循 ACID 准则。假设我们收到了一个新的需求，该公司要求在每一笔固定收入交易时都向其他系统发送一条 JMS 消息，以关注交易活动。为简单起见，我们同样假设在 sendPlacementMessage()方法中实现所有的 JMS 消息逻辑。上面的例子被修改如下以实现新功能：

```

@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void placeFixedIncomeTrade(TradeData trade)
    throws Exception {
    try {
        ...
        Placement placement =
    }
}
    
```

```

placementService.placeTrade(trade);

placementService.sendPlacementMessage(placement);

executionService.executeTrade(placement);

} catch (TradeExecutionException e) {

    log.fatal(e);

    sessionCtx.setRollbackOnly();

    throw e;

}

}

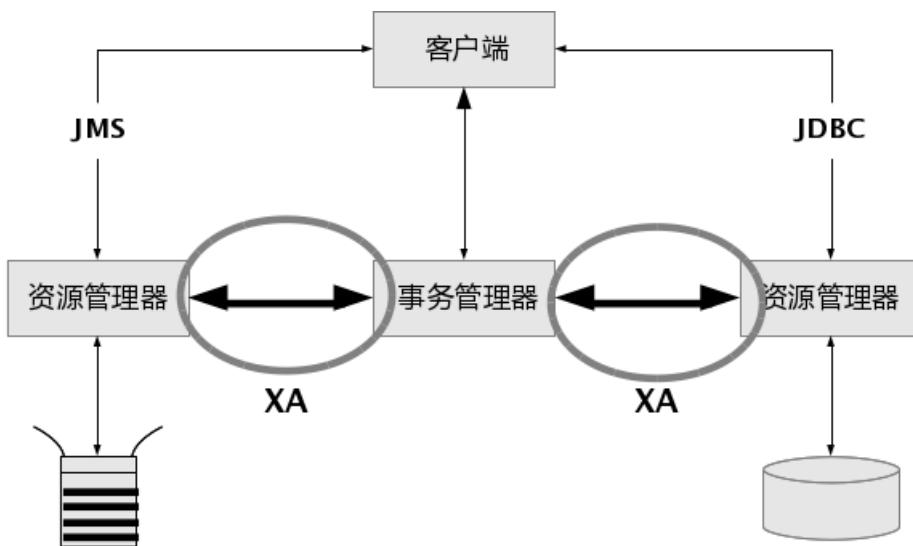
```

虽然以上的修改足够简单，这段代码却不会保证 ACID 准则。如果 executeTrade()方法抛出了 TradeExecutionException，数据库更改将会回滚，但交易预置的消息将被发送到 JMS 的 queue 或 topic。事实上，交易预置消息很可能在 sendPlacementMessage()方法执行完成后就被 queue 或 topic 释放（消费）掉了。

因为在非 XA 环境中，消息队列的插入过程独立于数据库更新操作，ACID 准则中的原子性和独立性不能得到保证，从而整体上数据完整性受到损害。我们需要的是，有一种方式能够让消息队列和数据库处于单一事务的控制之下，以至于两个资源能被协调形成单一工作单元。使用 X/Open 的 XA 接口，我们便能够做到协调多个资源，保证维持 ACID 准则。

## 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。

本章所叙述的内容很难覆盖 XA 接口的所有细节。如果读者关心 XA 的细节，请参考 X/Open XA 接口规范（可在 <http://www.opengroup.org/onlinepubs/009680699/toc.pdf> 通过 pdf 的格式拿到）。

### 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。下面的最佳实践描述了什么时候应当使用 XA：

#### 最佳实践

仅在同一个事务上下文中需要协调多种资源（即数据库，以及消息主题或队列）时，才有必要使用 X/Open XA 接口。

这里体现了一个重要的观点，即虽然您的应用可能使用到多个资源，但仅当这些资源必须在同一个事务范畴内被协调时，才有必要用到 XA。多个资源的情形包括访问两个或更多的数

据库（并不止是多个表，而是彼此分开的多个数据库），或者一个数据库加上一个消息队列，又或者是多个消息队列。您可能有一个应用同时使用到一个数据库和一个消息队列。然而，如果这些资源并不在同一个事务中使用，就没有必要去用 XA。本章开始的代码，预置一个固定收入交易，而后向队列发送一条消息，就是需要使用 XA 以便维护 ACID 特性的例子。

需要并使用 XA 最常见的场景是在同一个事务中协调数据库更改和消息队列（或主题）。注意这两种操作有可能在应用完全不同的地方出现（特别是在使用像 hibernate 这样的 ORM 框架的时候）。XA 事务必须在回滚事件发生时协调两种类型的资源，或让更改与其他事务保持隔离。如果没有 XA，送往队列或主题的消息甚至会在事务终止前到达并被读取。而在 XA 环境下，队列中的消息在事务提交之前不会被释放。此外，如果是协调一个操作型数据库和一个只读数据库（即参考数据库），您就不需要 XA。然而，由于 XA 支持“只读优化”，当把一个只读数据源引入 XA 事务时，您可能并不会看到任何的性能损失。

当意图在您的企业 Java 应用中使用 XA 时，有几个隐含的问题是需要考虑的。这些问题包括两阶段提交（2PC，two-phase commit process），经验异常，以及 XA 驱动的使用。以下章节分别详述了这些问题。

## 两阶段提交

两阶段提交协议（The two-phase commit protocol，2PC）是 XA 用于在全局事务中协调多个资源的机制。两阶段协议遵循 OSI（Open System Interconnection，开放系统互联）/DTP 标准，虽然它比标准本身早若干年出现。两阶段提交协议包含了两个阶段：第一阶段（也称准备阶段）和第二阶段（也称提交阶段）。一个描述两阶段提交很好的类比是典型的结婚仪式，每个参与者（结婚典礼中的新郎和新娘）都必须服从安排，在正式步入婚姻生活之前说“我愿意”。考虑有的杯具情形，“参与者”之一在做出承诺前的最后一刻反悔。两阶段提交之于此的结果也成立，虽然不具备那么大的破坏性。

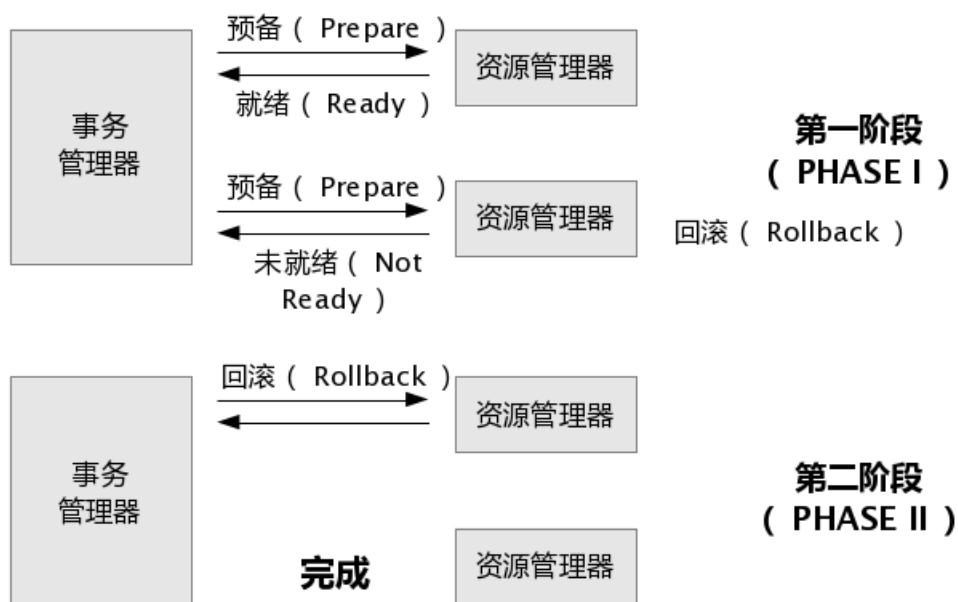
当 commit() 请求从客户端向事务管理器发出，事务管理器开始两阶段提交过程。在第一阶段，所有的资源被轮询到，问它们是否准备好了提交作业。每个参与者可能回答“准备好（READY）”，“只读（READ\_ONLY）”，或“未准备好（NOT\_READY）”。如果有任意一个参与者在第一阶段响应“未准备好（NOT\_READY）”，则整个事务回滚。如果所有参与者都回答“准备好（READY）”，那这些资源就在第二阶段提交。回答“只读（READ\_ONLY）”的资源，则在协议的第二阶段处理中被排除掉。

由于 XA 环境中双向通信的能力，两阶段提交变得可能。在非 XA 事务环境中，通信仅仅是

单向的，两阶段提交没法做到，这是因为事务管理器没法接收到来自资源管理器的响应。大多数事务管理器为了优化性能，尽快释放资源的目的，用多线程处理第一阶段轮询以及第二阶段提交流程。下图展示了两阶段提交的基本流程：



下图展示了当资源管理器之一（DBMS）在第一阶段轮询时发生错误的情况下两阶段提交的过程，



在这个示例中，一个提交请求被运行全局事务（global transaction，一个运行于 XA 之下的 JTA 事务）的客户端发送到事务管理器。在第一阶段，第二个资源管理器回给事务管理器一个“未准备好（NOT\_READY）”响应。在本例中事务管理器对所有参与者发出回滚请求，因此协调了在全局事务中的所有资源。

一些商业的应用容器提供一种称之为“最后参与者支持（Last Participant Support）”的特性，该特性有个另外的名字叫“最后资源提交优化（Last Resource Commit Optimization）”。“最后参与者支持”允许非 XA 资源参与进全局事务。在“最后参与者支持”下，当一个 XA 环境的提交请求到达事务管理器，事务管理器会首先对 XA 资源发起第一阶段流程。一旦 XA 参与者产生的结果一致返回，事务管理器随即对非 XA 参与者发起提交（或回滚）的请求。这个请求的结果决定了两阶段提交流程剩下的工作如何进行。如果对非 XA 资源的请求成功，事务管理器会发起第二阶段，并对 XA 参与者发起提交请求。如果对非 XA 参与者的请求不成功，事务管理器会发起第二阶段，并要求所有 XA 参与者回滚事务。

“最后参与者支持”机制存在两个问题。第一，它不是在应用容器间可移植的。第二，因为在第一阶段轮询过程和非 XA 最后参与资源提交之间有较长的时间等待，您会发现在使用这一特性时发生经验异常（Heuristic Exception）的几率增加了（将在下一节详述）。基于这些原因，“最后参与者支持”特性应该在一般情况下避免使用，除非迫不得已。

大多数商业应用服务器还支持另一个优化，称之为“一阶段提交优化（One-Phase Commit Optimization）”。如果事务只包括一个参与者，第一阶段处理会被忽略，单一的参与者被通知提交。在这种情况下，整个 XA 事务的后果取决于单一参与者的工作。

## 经验异常<sup>\*</sup>（Heuristic Exception）处理

在两阶段提交的过程，资源管理器可能会使用“经验化决策”的策略，或者提交，或者回滚它自己的工作，而不受事务管理器的控制。“经验化决策”是指根据多种内部和外部因素做出智能决定的过程。当资源管理器这么做了，它会向客户端报上一个经验异常（Heuristic Exception）。

---

<sup>\*</sup> 有关 Heuristic Exception 的翻译，Heuristic 在大多数计算机书籍中被翻译为“启发式”，指根据掌握的有限事实，容易“诱导”出来的大概规律和判断，这样的规律和判断不一定是经过严格证明或验证的，或缺乏严格验证的可能性。在哲学领域，其根本含义是“经验（主义）的”，如我们常常看到的“经验算法”，“经验公式”等。显然，在 Java 事务处理，两阶段提交的这个语境下，Heuristic 并不是所谓“启发式”的含义。在这里，一定程度上的具体解释是，在两阶段提交的第一阶段，事务管理器接受到了所协调资源的状态，获得了对全局事务处理状态的一定“认知”，但在随后的处理中，由于种种原因，被协调的资源无法兑现之前的（提交）承诺，于是这种既有的“认知”就发生问题了，造成了 Heuristic Exception——因此，个人认为，将它翻译为“经验异常”比较准确。

所幸的是，经验异常并不是特别常见。它仅仅发生在 XA 环境下，做两阶段提交的过程中，特别是事务参与者在第一阶段产生了响应之后。经验异常最常见的原因是第一阶段和第二阶段之间的超时情况。当通讯延迟或丢失，资源管理器或许要做出提交或回滚其工作的决定，以释放资源。不出意料，经验异常发生最频繁的时候正是高资源利用时间段。当您在应用中发现经验异常时，您应该查找是否有事务超时问题，资源锁定问题，以及资源使用过量问题，这些问题常常是经验异常的根本原因。偶尔网络延迟或网络故障也会导致经验异常。同样的，如上面的章节所述，使用“最后参与者支持”特性会导致经验异常更为频繁的发生。

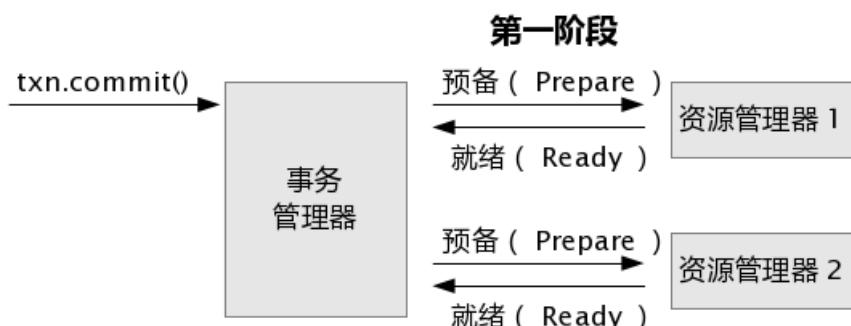
JTA 暴露出的三种 JTA 经验异常为 HeuristicRollbackException，HeuristicCommitException，以及 HeuristicMixedException。我们分别用下面的场景说明之：

### 场景 1：在 commit 操作阶段的 HeuristicRollbackException 异常

在此场景中，客户端在 XA 环境下执行更新操作，向事务管理器发起提交当前事务的请求。事务管理器开启两阶段提交流程的第一阶段，随即轮询资源管理器。所有资源管理器向事务管理器报告说它们已经做好了提交事务的准备。然而，在（两阶段提交流程的）第一阶段和第二阶段之间每个资源管理器独立的做出了回滚它们已完成工作的经验性决定。当进入第二阶段，提交请求被发送到资源管理器时，因为所做的工作已经在此之前回滚了，事务管理器将会向调用者报告 HeuristicRollbackException 异常。

当接受到此类异常时，常用的正确处理方式是将此异常传回客户端，让客户端重新提交请求。我们不能简单的再次调用 commit 请求，因为对数据库产生的更新已经随回滚操作从数据库事务日志中删除了。下面的序列图说明了这一场景：

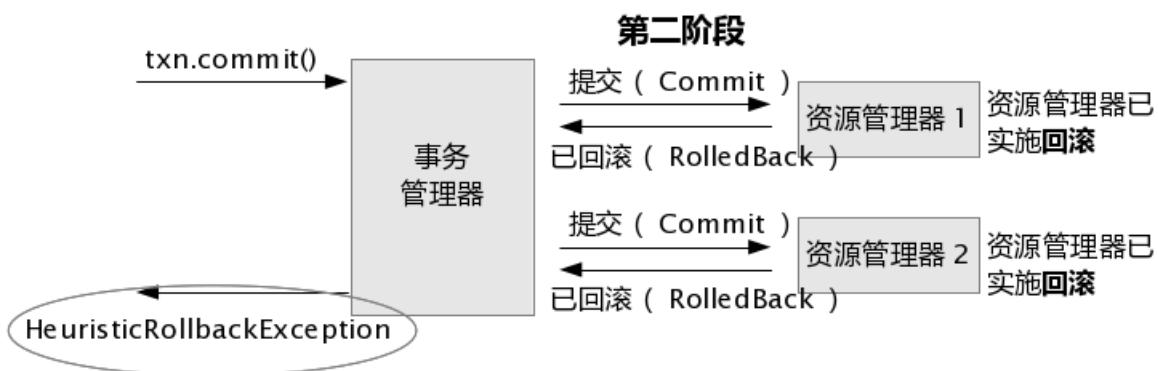
#### 第一步：第一阶段处理（准备阶段）



#### 第二步：在第一阶段和第二阶段之间



### 第三步：第二阶段处理（提交阶段）



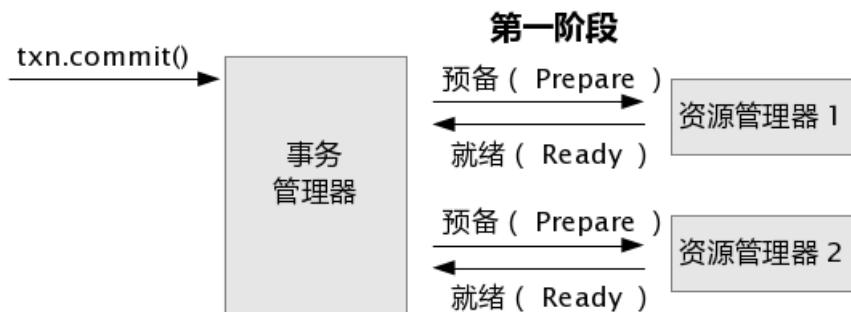
正如您从序列图中看到的，两个资源管理器回滚了他们自己的工作，虽然他们在第一阶段都向事务管理器报告了 READY 的响应。别担心这些异常因何发生，我们在后续章节会做深入探讨。

### 场景 2：在 commit 操作阶段的 HeuristicMixedException 异常

在此场景中，客户端在 XA 环境下执行更新操作，向事务管理器发起提交当前事务的请求。事务管理器开启两阶段提交流程的第一阶段，随即轮询资源管理器。所有资源管理器向事务管理器报告说它们已经做好了提交事务的准备。和第一种场景不同的是，在第一阶段和第二阶段发生的间隙，有资源管理器（例如消息队列）做出了经验性的决定提交其工作，而其他资源管理器（例如数据库）做出了回滚的经验性决定。在这种情况下，事务管理器向调用者报告 `HeuristicMixedException` 异常。

这种情况下，非常难于选择正确的后续应对方式，因为我们不知道哪些资源提交了工作，哪些资源回滚了工作。所有目标资源因此处于一种不一致的状态。因为资源管理器彼此互不干预的独立操作，就经验性决定而言，他们之间没有任何协调和通信。解决这一异常通常需要人力介入。下面的序列图说明了这一场景：

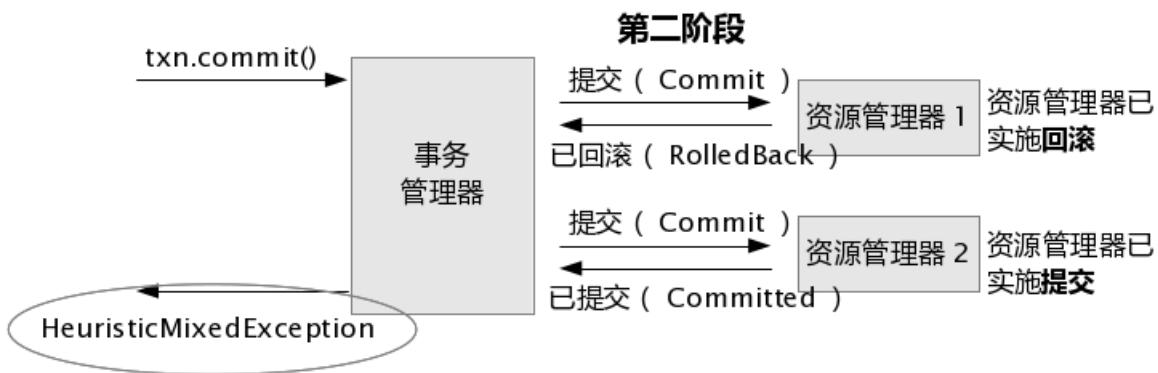
### 第一步：第一阶段处理（准备阶段）



### 第二步：在第一阶段和第二阶段之间



### 第三步：第二阶段处理（提交阶段）



注意在上面的图示中，一个资源管理器提交了它的工件，而其他资源管理器选择回滚其工件。  
 在这种情况下事务管理器将会报告 HeuristicMixedException。

### 对消息队列或主题使用 XA

在 XA 接口下使用的资源必须实现 javax.transaction.xa.XAResource 接口，以便自己能够加入 XA 全局事务。对于 JMS 目标（队列或主题），这可以通过在特定的应用服务器控制台或管理程序中配置完成。真正激活 XA 的部分是 JMS 连接工厂（Connection Factory）。一旦 JMS 连接工厂支持 XA，发送给 JMS 队列或主题的消息在两阶段提交过程结束之前不会被释放。在没有 XA 的情况下，不论所处的事务上下文结果如何，发送给 JMS 目标的消息会被立即释放，

并可被接收者拾取。

在 WebLogic 应用服务器中，可在管理控制台的 Services|JMS|Connection Factories 配置中激活 XA 的 JMS 连接工厂。在 Transactions 这个页面，有一个名为 XA Connection Factory Enabled 的选项，经由此可以将 JMS 目标包含到 JTA 全局事务中。对于 IBM WebSphere，可在管理控制台的 Resources|WebSphere JMS Providers|Connection Factories 路径下选择使用 XA 的 JMS 连接工厂功能。勾上名为 EnableXA 的选择框就激活了 XA 的 JMS 连接工厂。

### 为数据库使用 XA

可通过使用 XA 版的数据库驱动来使数据库支持 XA。由于 XA 版的数据库驱动通常比非 XA 的难用许多，一个忠告是不到不得已的时候别使用 XA 驱动。

使用 XA 版的数据库驱动常会导致不可预期且难于解决的错误。例如，将非 XA 驱动替换为 XA 版驱动常常会产生难于跟踪的错误。因此，应该在项目开发和测试阶段尽早的引入 XA 驱动（，及早暴露问题并解决之）。

在使用 XA 版的数据库驱动时，可能碰到的错误种类包括本地事务错误和嵌套事务错误。当您在一个 XA 全局事务正在进行的过程中试图开启新的事务，这些错误就会产生。这种情形会在多个环境下发生，但最常见的情况是混合本地事务模型与声明式事务模型导致，以及在 XA 环境下使用存储过程的情况。

当在 XA 环境下使用存储过程，在存储过程里调用 DDL（数据定义语句，如 CREATE TABLE，BEGIN TRAN，END TRAN）常导致错误。这是最频繁导致 XA 错误的罪魁祸首，并很难修正。例如在 Oracle 中，使用 XA 时，您可能会看到下面的错误信息：

#### ORA-02089: COMMIT is not allowed in a subordinate session

如果使用非 XA 的数据库驱动，大概您不会看到这个错误，因为 DDL 语句执行的时候 JTA 事务会暂停。当看到这个错误信息，表明了您的存储过程中包含 DDL 代码，并且（由资源管理器管理的）本地事务尝试提交它的工作。

通常要从存储过程中删除既有的 DDL 语句是困难的，因为它们在那里一定有存在的理由，或者也许这些存储过程被其他应用所共享（，要删除它们牵扯面太大）。一个有效的绕开问题的做法是，调用这些存储过程之前，手工的暂停事务；而在这些存储过程返回后，继续事务。使用这个技巧会避免 XA 相关的本地和嵌套错误。然而，如果这样做，存储过程做出的修改会独立于 JTA 全局事务提交，因此违背了事务的 ACID 特性。所以说，这种做法仅仅是

绕开问题，而不是解决问题。下面的代码片段展示了此技巧的细节：

```
...
InitialContext ctx = new InitialContext();
TransactionManager tm =
(javax.transaction.TransactionManager)
ctx.lookup("javax.transaction.TransactionManager");
Transaction currentTx = null;
try {
    currentTx = tm.suspend();
    invokeSPWithDDL();
} finally {
    if (currentTx != null)
        tm.resume();
}
```

即便在声明式事务的环境下，我们仍然可以使用 TransactionManager 去用代码方式暂停和继续事务。这个技巧能够避免 XA 环境下的 SQL 异常信息，但它没有真正的解决问题。——真正解决问题的唯一方法是在有关存储过程中删除那些犯规的 DDL 语句，或者使用支持嵌套事务的 JTA 事务服务。

## 总结

本章要表达的最重要的思想是，理解什么时候您真正需要使用 XA 版的数据库驱动。许多开发人员和架构师总是坚持要使用 XA 版的数据库驱动，虽然事实上不存在使用它们的合理理由。如果您需要在同一个事务中协调多个更改的资源（数据库、消息队列、主题，或者 JCA），那么毫无疑问您需要引入 XA 接口。否则，千万避开 XA。

另一条关于使用 XA 的建议是，碰到问题时别总去猜测您使用的是一个可能错误百出的 XA 数据库驱动。问题很有可能是您的应用代码或事务处理逻辑造成的，而非 XA 驱动。

## 第六章 事务设计模式

以上章节介绍的事务模型提供了一种在 Java 中管理事务的框架，但它们没有告诉使用者如何创造一个有效的事务管理策略。制定一个健壮有效的事务管理策略并非难事。使用本书介绍的事务设计模式能够简化事务处理，使您能够构建更强大的应用，能帮助您为您的应用程序制定有效的事务设计策略奠定坚实的基础。

在这短短的一章内，我们将简要介绍各种事务设计模式（Transaction Design Pattern），并告知您它们所应用的典型架构。这可以帮助您确定在特定条件下应该应用哪一种模式，不需要每次都难于权衡于累牍。

### 关于模式的简要介绍

事务设计模式描述了一种特定应用架构下的事务整体设计策略。每一种事务设计模式涵盖的内容包括应该应用的事务模型种类，事务设定，负责管理事务的应用组件种类，以及所描述的特定事务设计模式所适用的应用架构。

下面三章介绍到的事务设计模式分别是“客户端拥有事务的设计模式（Client Owner Transaction Design Pattern）”，“领域服务拥有事务的设计模式（Domain Service Owner Transaction Design Pattern）”，和“服务器端代理拥有事务的设计模式（Server Delegate Owner Transaction Design Pattern）”。正如其名，这些事务设计模式的划分基于组件责任模型，基于有关模型某个应用组件类型拥有开启和管理事务的责任。这些设计模式体现了前面几章所描述到的事务管理最佳实践。抽取出这些模式的目的在于，能够有一种方法对您的应用实施一个“事务模板”，以减少诸如“应该安排哪个组件负责管理事务”和“应该应用什么声明式事务设定”方面的疑问。

对于大多数企业级 Java 应用，您应该会使用“领域服务拥有事务的设计模式”去构建事务管理策略。该事务设计模式是应用于 EJB 和 Spring 进行事务管理的典型模式。正如您将在第八章看到的，该设计模式依赖于声明式事务模型，并能简单实现。该模式可应用于需要远程访问后端服务对象的企业级 Java 应用架构，或者是并没有利用到远程服务的简单应用架构。

当您需要在企业级 Java 应用的表示层管理事务，您应该采用“客户端拥有事务的设计模式”去设计和实现您的事务设计策略。虽然此模式绝大多数情况下都应用于需要远程访问后端服务对象的企业级 Java 应用架构，它也可应用于包含本地细粒度服务对象的非远程应用。此模式将在第七章详述。

“服务器端代理拥有事务的设计模式”为利用“命令模式（Command Pattern）”或“服务器端代理设计模式（Server Delegate Design Pattern）”的场景专用。该模式是“领域服务拥有事务的设计模式”的特例，通常能解决大多数关于基于客户端的事务管理和 EJB 的问题。这是最容易实现的事务模式，并能够应用于 EJB 和 Spring 等两种编程框架。第九章涵盖了此事务设计模式的内容，同时简要介绍了与之相关的“命令模式”和“服务器端代理设计模式”的基本内容。

## 第七章 客户端拥有事务的设计模式

正如其名称所揭示的，在这种事务设计模式下“客户端代理（Client Delegate）”组件拥有 JTA 事务。虽然这种模式主要用在客户端是基于 WEB 的、或基于 Swing 的，而服务器部分使用 EJB 和远程无状态 SessionBean 的情况下。它也可以用在非远程的但包含多个细粒度的本地服务对象的应用中，如果一次客户端请求包含多次对这些服务对象的交互，就可以利用到此模式。

### 背景（Context）

虽然情非得已，在某些情形下将一个企业级 Java 应用的事务管理强制推到客户端去处理也是必要的。考虑这样一种再常见不过的情形，单一请求从客户端发起，但客户端需要多个（远程的）对服务器的调用以完成这个请求。这样的情形发生在领域服务过于细分，并且没有聚集服务（aggregate services）存在时。当此情形发生时，我们需要将事务管理上推到表现层去维持 ACID 特性。

看看下面的例子，客户端接收到一个产生固定收入交易的下单请求。所谓固定收入交易是指买入或卖出债券或国库券之类的交易，而产权交易（equity trade）是指买入或卖出特定股票（如 Google 的股票）之类的交易。当固定收入交易下单时，客户端调用 placeTrace()方法，而后在同一事务中执行 executeTrade()方法。与之相反，在产权交易下单过程中，这两个方法可能是独立调用的。下面的代码片段中，客户端业务代理对象的使用说明了有关场景：

```
public void placeFixedIncomeTrade(TradeData trade)
    throws Exception {
    try {
        ...
        Placement placement =
            placementService.placeTrade(trade);
        executionService.executeTrade(placement);
    } catch (Exception e) {
        log.fatal(e);
        throw e;
    }
}
```

```
}
```

```
}
```

因为这些动作必须作为单一的工作单元看待，他们必须被封装在同一事务中。如果缺乏客户端的事务处理，以上代码可能导致 ACID 特性的破坏。因为 placeTrade()方法所造成的修改操作可能会先于 executeTrade()方法的调用前被提交到数据库。

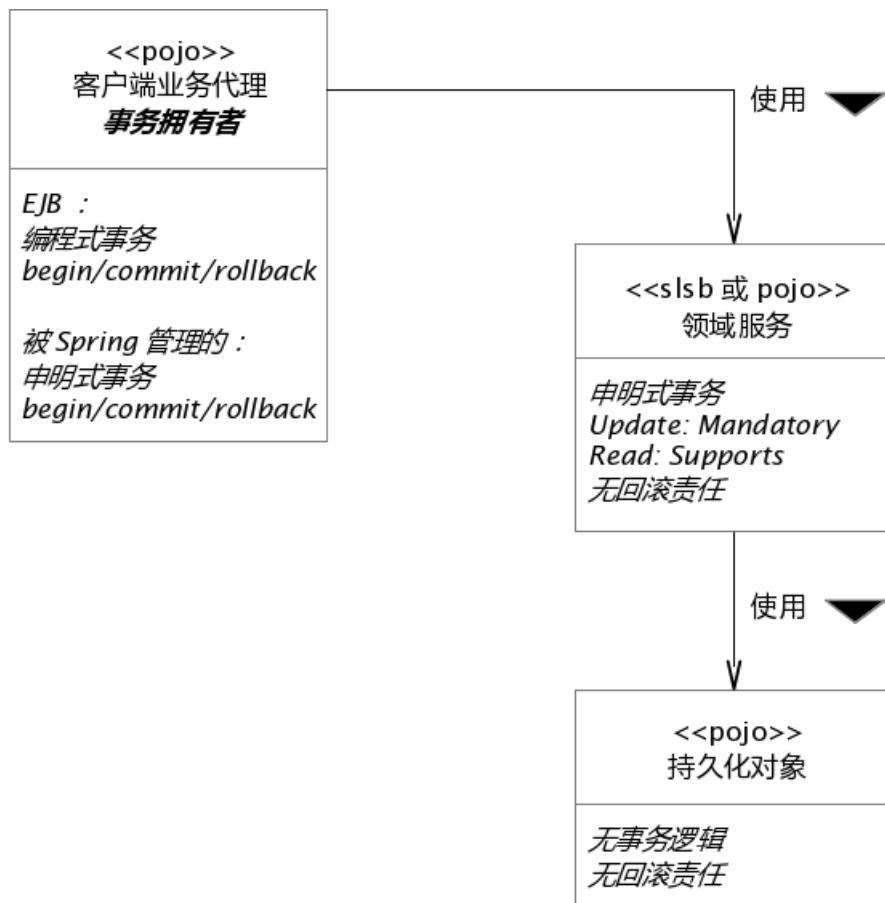
虽然有很多种方法可以调整应用架构，以避免这样的陷阱，但应用使用基于客户端的事务管理的方式仍然在某些情况下不得不存在。伴随基于客户端的事务管理的一个问题是，过多的平台相关的责任被放在客户端。特别是在用到基于 web 的框架，框架直接和远端 EJB 领域服务对象（Domain Service objects）或基于 Spring 的对象通信，而没有使用业务代理对象（business delegate object，更多信息参见后续讲到的业务代理设计模式）时，此情形尤之为甚。避免在客户端使用事务管理的一种方法是使用命令模式或服务器端代理模式。这两种模式都使用到了服务器端代理拥有者事务模式，在本书第 9 章我们会详细描述。

## 制约条件 ( Forces )

- 客户端需要对服务对象发起多个远程的或本地的调用以完成一个业务请求。
- ACID 特性必须被保证，意味着事务处理是必须的，以维护数据一致性。
- 领域服务对象划分粒度较细，并且没有聚集服务存在以合并多个业务请求。
- 不具备重构应用或重新设计架构，以提供单一领域业务方法调用满足一次请求完成多个交互的条件。

## 解决方案 ( Solution )

对基于客户端事务管理场景的解决方案是使用客户端拥有者事务设计模式作为整体的事务设计策略。该模式使用了将事务管理责任放置在客户端的组件责任模型。以下图表展示了 EJB 和 Spring 框架下此模式的细节：



如上图所示，应用中负责事务管理的唯一组件是“客户端业务代理( Client Business Delegate )”（即事务所有者）。如果客户端业务代理被 Spring 框架管理，该组件则能使用声明式事务管理。否则，必须使用编程式事务。对 EJB 而言，该组件必须使用第三章中描述的 JTA 用户事务 ( UserTransaction ) 接口去实现编程式事务。

在这种模式下，领域服务 ( Domain Service ) 组件必须使用声明式事务。正如我们在第三章中看到的，如果领域服务中使用编程式事务，客户端发起的事务上下文就不能被传递到远端或本地的领域服务中。然而，如果领域服务使用声明式事务，客户端组件发起的编程式事务上下文就可以传播到它。

由于 Spring 框架提供了对事务性 POJO 的支持，客户端业务代理发起的事务上下文能够被传播到 Spring 管理的领域服务中。如果使用了 Spring 的 remote bean，您则必须使用支持事务传播的远程访问协议（如 RMI）去完成这一功能。

如本模式所表明的，所有进行插入、更改、删除操作的领域方法必须设置事务属性为 Mandatory。更重要的是，领域服务对象一定不能将事务标记为在应用级异常 ( checked，被检查异常 ) 时回滚。如第四章描述到的，“将事务标记为回滚”是事务管理的一种形式，这个任务需要交给负责事务的组件完成。在本模式中，此组件即为“客户端业务代理 ( Client

Business Delegate ) ” , 而非 “ 领域服务 ( Domain Service ) ” 组件。在领域服务组件中 , 如果处理应用级 ( 被检查 ) 异常而将事务标记回滚 , 会导致客户端业务代理无法进行修正性的编程操作去解决有关问题 , 以继续事务。

注意本事务设计模式需要在一个应用里同时使用编程式和声明式两种事务。虽然一般来说在同一应用中将编程式和声明式事务混起来用不是什么好办法 , 但当使用基于客户端的事务去交互远端服务时 , 这是一种可接受的和必要的实践 ( 特别是在使用 EJB 时 ) 。

## 后果 ( Consequences )

- 如果使用 EJB , 管理事务的客户端对象必须使用编程式事务 ; 如果客户端对象被 Spring 管理 , 则也可以使用声明式事务。
- 如果基于服务器的领域服务对象在远端 , 则有关事务必须能够被传播到这些远程服务对象。要注意 , Spring 中使用的远程调用协议大多数不支持事务传播。
- 因为编程式事务不能被传播到其他编程式事务所管理的对象中 , 基于服务器的领域对象必须使用声明式事务。
- 为了保证 ACID 特性 , 服务器端的对象不要去开启或提交一个事务。并且 , 服务器端的对象也别把一个事务标记为 “ 需要回滚 ” 。
- 客户端开启的事务会被传播到所有有关的本地 POJO , 如果这些 POJO 在事务可视范围之内的话。
- 服务器端的领域对象必须将所有有关的事务性方法标记为 Mandatory 事务属性。
- 无论在客户端或服务器端 , 对查询相关的方法 , 没有必要去考虑事务逻辑。然而 , 对服务器端领域对象而言 , 必须对查询方法设置 Supports 的事务属性。
- 领域服务组件用到的持久化对象 ( Persistence objects ) , 无论采用哪种持久化框架 , 不能包含任何事务或回滚的逻辑。

- 如果使用 EJB2.1 的实体 Bean , 对于更新操作的事务属性必须设置为 Mandatory , 并且没有任何回滚的逻辑。对于读取操作 , 容器管理持久化 ( Container-managed Persistence , CMP ) 的实体 Bean 方法必须设置事务属性为 Required , 而 Bean 管理持久化 ( Bean-managed Persistence , BMP ) 的实体 Bean 方法必须设置事务属性为 Supports。

## 实现 ( Implementation )

下面的代码展示了在 EJB 和 Spring 中本模式的实现。为了说明问题的需要 , 我假设 EJB 上构建的领域服务组件是远程的 , 而对于 Spring 的例子 , 假设客户端代理和领域服务组件都在 Spring 框架的管理之下。

### EJB

为更新操作编制的客户端业务代理组件和 EJB 领域服务组件源代码如下( 事务逻辑部分用黑体显示 ):

```
public class ClientModel {
    public void updateTradeOrder(TradeData trade)
        throws Exception {
        InitialContext ctx = new InitialContext();
        UserTransaction txn = (UserTransaction)
            ctx.lookup("java:comp/UserTransaction");
        try {
            txn.begin();
            TradingService tradingService =
                ServiceLocator.getTradingService();
            tradingService.updateTradeOrder(trade);
            txn.commit();
        } catch (TradeUpdateException e) {
            txn.rollback();
        }
    }
}
```

```

        log.fatal(e);

        throw e;

    }

}

}

```

```

@Stateless
@TransactionAttribute(TransactionAttributeType.MANDATORY)

public class TradingServiceImpl implements TradingService
{
    public void updateTradeOrder(TradeData trade)
        throws TradeUpdateException {
        validateUpdate();
        TradeOrderDAO dao = new TradeOrderDAO();
        dao.update(trade);
    }
}

```

注意，基于 EJB 的领域服务组件处理应用异常时并没有去调用 setRollbackOnly()方法。因为我们并没打算对这个被检查异常采取什么修正性的措施，我们没必要在上面的例子中引入 try/catch 代码块。如第四章的最佳实践表明，如果组件没有开启一个事务，并且它不负责管理这个事务，那（对组件上方法）的事务属性需要被设置为 Mandatory，并且需要保证没有 setRollbackOnly()方法被调用。调用 setRollbackOnly()方法可能是在领域服务代码中 catch 被检查异常的唯一真正原因。

我们同时可以从上面的代码中看出，客户端业务代理组件使用 UserTransaction 接口开启事务并实施 commit()或 rollback()操作。经由此模式，所有事务逻辑皆包含在客户端中。因为客户端完全负责管理事务，就绝不允许领域服务组件自行开启一个事务，这正是将其事务属性设置为 Mandatory 的原因。因为领域服务组件中使用声明式事务模型，一个非检查异常（non-checked exception）会将事务标记为回滚。然而，因为事务上下文是客户端开启的，容器将会仅仅把事务标记为回滚，而并不真正回滚这个事务。在这种情况下，如果客户端尝试去提交事务，一个异常会被抛出，表明该事务已经被标记为回滚，不能被提交了。

使用本模式时，对只读的请求，客户端并不需要事务上下文。因此，客户端业务代理组件没有包含任何针对读取请求的事务代码。在更新操作发生的过程中，如领域服务组件的一个查询方法在事务上下文范围内被使用到，则该领域服务组件被赋予 Supports 的事务属性（参见第四章）。读取操作的客户端业务代理组件和 EJB 领域服务组件源代码如下所示（事务逻辑用粗体显示）：

```
public class ClientModel
{
    public TradeData getTradeOrder(TradeKey key)
        throws Exception {
        TradingService tradingService =
            ServiceLocator.getTradingService();
        return tradingService.getTrade(key);
    }
}
```

```
@Stateless
@TransactionAttribute(TransactionAttributeType.MANDATORY)
public class TradingServiceImpl implements TradingService
{
    @TransactionAttribute(
        TransactionAttributeType.SUPPORTS)
    public TradeData getTrade(TradeKey key)
        throws Exception {
        TradeOrderDAO dao = new TradeOrderDAO();
        return dao.get(key);
    }
}
```

如上面展示的，依照事务的观点，没有在任一个组件部分去写有关的事务代码。我们只需将 EJB 领域服务方法标记为 Supports 事务属性即可。我们也不需要任何 try/catch 异常的逻辑，因为我们在这里没去刻意管理事务。

## Spring 框架

在 Spring 中 , 如果要使用这种模式 , 对于更新和读取操作 , 客户端业务代理组件和远程领域服务组件的 XML 配置文件书写如下 ( 事务的逻辑用粗体表示 ) :

```

<!-- 定义客户端代理 Bean -->

<bean id="clientModelTarget"
      class="com.trading.client.ClientModel">
    <property name="tradingService" ref="tradingService"/>
</bean>

<bean id="clientModel"
      class="org.springframework.transaction.interceptor.
          TransactionProxyFactoryBean">
    <property name="transactionManager" ref="txnmgr"/>
    <property name="target" ref="clientModelTarget"/>
    <property name="transactionAttributes">
      <props>
        <prop key="*>
          PROPAGATION_REQUIRED,-Exception
        </prop>
        <prop key="get*>>PROPAGATION_SUPPORTS</prop>
      </props>
    </property>
</bean>

```

```

<!-- 定义领域服务 Bean -->

<bean id="tradingServiceTarget"
      class="com.trading.server.TradingServiceImpl">
</bean>

```

```

<bean id="tradingService"
      class="org.springframework.transaction.interceptor.
      TransactionProxyFactoryBean">

    <property name="transactionManager" ref="txnMgr"/>
    <property name="target" ref="tradingServiceTarget"/>
    <property name="transactionAttributes">
        <props>
            <prop key="*">PROPAGATION_MANDATORY</prop>
            <prop key="get*>>PROPAGATION_SUPPORTS</prop>
        </props>
    </property>
</bean>

```

因为在 Spring 中，可以在客户端代理和领域服务组件中皆使用声明式事务，负责更新和查询的 Java 代码都不会包含任何事务逻辑。Spring 通过在形如上面例子的 XML 配置文件中通过 `-Exception` 自动处理 `setRollbackOnly()`。注意，在 `tradingService` 对象的 bean 定义中，没有关于回滚规则的特别配置。这是因为我们希望使用缺省的行为，即碰到被检查异常时不要回滚事务。

```

public class ClientModel {
    public void updateTradeOrder(TradeData trade)
        throws Exception {
        tradingService.updateTradeOrder(trade);
    }

    public TradeData getTradeOrder(TradeKey key)
        throws Exception {
        return tradingService.getTrade(key);
    }
}

```

```
}
```

```
public class TradingServiceImpl implements TradingService
{
    public void updateTradeOrder(TradeData trade)
        throws TradeUpdateException {
        validateUpdate();
        TradeOrderDAO dao = new TradeOrderDAO();
        dao.update(trade);
    }

    public TradeData getTrade(TradeKey key)
        throws Exception {
        TradeOrderDAO dao = new TradeOrderDAO();
        return dao.get(key);
    }
}
```

注意上面在客户端和服务器端的代码中都没有任何事务逻辑。如果客户端模型不被 Spring 所管理，就需要在其中应用和之前 EJB 例子中同样的处理逻辑。



Microsoft®  
Visual Studio 11 Beta

# 没有理由再等了！

即刻下载Visual Studio 11 Beta，  
为下一阶段的开发做准备。  
你不能预知未来，但可以第一时间体验未来！

立即下载



## 第八章 领域服务拥有事务的设计模式

“领域服务拥有事务的设计模式”是事务设计模式中最常用到的方式，在大多数基于 Java 的应用架构中基本上都会用到。在这种模式中，领域服务组件拥有事务，负载所有有关事务的管理。在 EJB 环境下，领域服务组件通常被实现为无状态会话 bean ( Stateless SessionBean )，于是使用声明式事务模型去管理事务。在 Spring 环境下，领域服务组件被实现为 POJO，同样也使用声明式事务管理。

### 背景 ( Context )

大多数架构师和开发人员都同意这样的观点：在企业级的 Java 应用中，应该由服务器端来负责管理事务。其原因如下：首先，我们也许并不知道什么类型的客户端会访问后端的服务。例如，从功能的角度看，一个 Web Services 的客户端并不能开启一个事务，并将它传播到我们的领域服务中。其次，如果将事务管理的责任分配给客户端，势必给客户端加重了负担。客户端本应该是主要负责展现逻辑 ( presentation logic ) 的，而不是事务处理这样的服务端逻辑或中间件逻辑。最后，基于客户端的事务管理架构通常都有性能问题，因为这样客户端对服务器产生了很多的远程调用，导致服务器忙于交互以至于降低了整体性能。总而言之，基于客户端的事务管理为应用的事务整体设计策略带来了很多不必要的复杂性。

基于这些原因，企业级的 Java 应用通常采用中等乃至粗略的领域服务粒度规划，将一个业务请求作为单一工作单元处理。这样做即简化了整体架构，并提升了应用整体性能。在这种情况下，服务器端的对象掌控了事务管理。于是乎，这变成了哪个组件负责开启事务、提交事务、将事务标注为回滚，以及事务怎样向持久化框架传播的问题。

对于之前举过的那个例子，客户端向服务器发起进行固定收入交易的下单请求。正如我们在前面章节看到的，此请求包含对 placeTrade() 方法和 executeTrade() 方法的调用，且在统一事务工作单元中完成。如果使用粗粒度的服务，处理这一行为的客户端代码可以这样写：

```
public void placeFixedIncomeTrade(TradeData trade)
    throws Exception {
    try {
        ...
        tradingService.placeFixedIncomeTrade(trade);
    } catch (Exception e) {
```

```
    log.fatal(e);
    throw e;
}
}
```

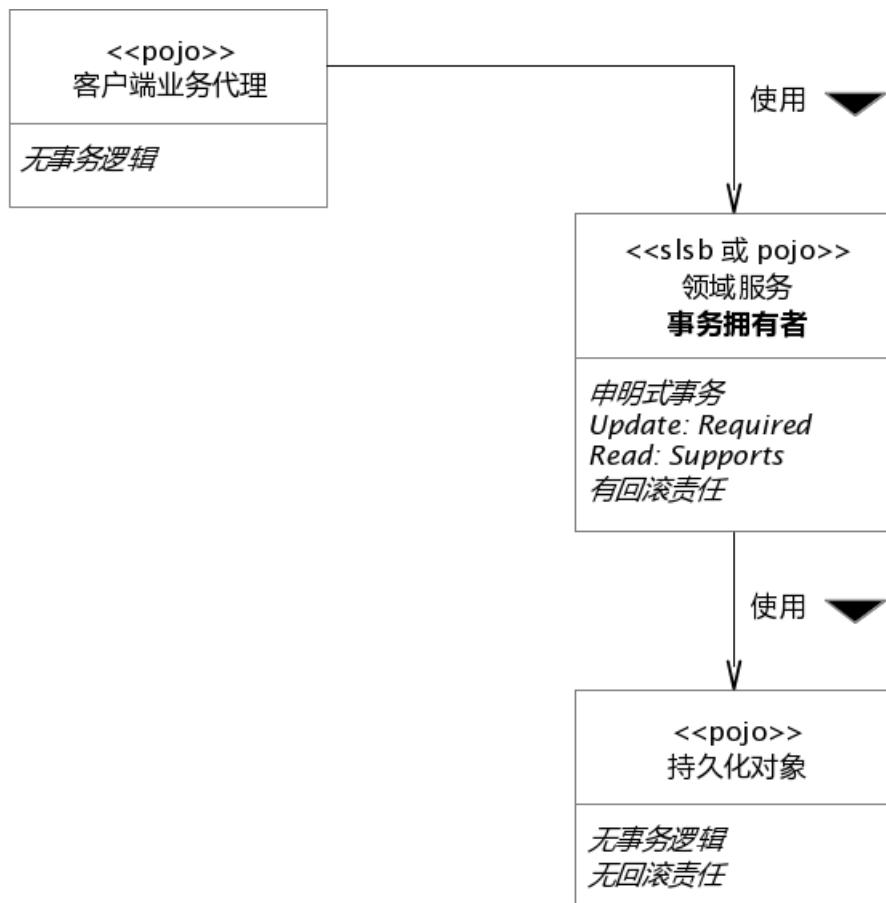
因为客户端只对服务器发起一次调用，客户端没有必要去管理事务，即便我们知道该调用将包含多个服务器端方法的交互。因此，事务管理的逻辑必须位于服务器端对象的某个地方。

## 制约条件 ( Forces )

- 客户端总是对领域服务对象发起单一请求，以完成单一的客户端业务逻辑。
- ACID 属性必须得到保证，意味着事务处理过程需要维护数据完整性。
- 领域服务对象是粗粒度的，或者使用服务间通信 ( inter-service communication )，以完成服务请求的聚集 ( aggregate )。

## 解决方案 ( Solution )

将领域服务拥有事务的设计模式作为整体事务设计策略是典型的服务器端事务管理场景解决方案。该模式使用了将事务管理的全部责任附加在应用的领域服务组件上的组件责任模型，不管领域服务位于本地或远端皆如此。下图展示了 EJB 和 Spring 框架两种编程模型下该模式的细节：



如上图所示，应用中唯一负责事务管理的组件是领域服务组件。架构中没有其他的组件类型关注事务，它们甚至感觉不到事务处理的存在。不论使用哪种编程框架（EJB 或 Spring），领域服务对象皆使用声明式事务模型。对所有的插入、更新及删除操作，需要将事务属性设置为 Required；而对所有读取操作，则应该设置为 Supports。该模式的核心原则是“服务器端的入口点（即调用领域服务方法的地方）拥有事务”，因此，此领域服务引用到的所有对象将位于事务控制范围之内。

在此模式中，各领域服务组件皆应用 Required 的事务属性。而不像在领域组件间的服务相互通信事件中，设置事务属性为 RequiresNew 的做法——在企业级 Java 应用中，这也是相当常见的。这种事务设计模式比较简单，因为仅仅由领域服务组件拥有事务控制权，编码和配置的工作量是很小的，该模式相对来说容易实现和维护。

领域服务组件间的跨服务通信给此模式带来了一个两难困局。所谓跨服务通信发生在一个领域服务方法调用其他领域服务中另一个方法的时候。因为所有修改相关方法的事务属性都被标记为 Required，如果一个事务上下文已经建立，该方法会使用既有的事务上下文，而不是去新开启一个事务。然而，如果被调用到的跨服务方法抛出一个异常，它可能会在离开方法之前将事务标记为回滚，因此，没办法让“调用者”去采取修正性的措施。这种情况固然不能

忽视，但它实际上不会对事务设计模式造成太大的影响。

这是企业级 Java 应用中最普遍的一种事务模型。当领域服务组件提供的服务是粗粒度的，并且客户端只需与服务器交互一次就可以完成单一业务请求时，此模式就可以施展了。相比第七章描述的多次远程调用架构，此类应用架构不但提供了更好的性能，而且从道理上讲，事务处理也被安置在了服务器端——一个它应该在的地方。在使用 web services 客户端或不能轻易修改的第三方客户端软件包时，这就显得特别重要了。

## 后果 (Consequences)

- 客户端（无论哪种类型）不包含任何事务逻辑，也不管理事务处理的任何方面。
- 由于领域服务组件开启和管理事务，更新（update）方法必须在应用异常（被检查异常）时负责调用 `setRollbackOnly()` 方法。
- 领域服务用到的持久化对象，无论用到哪种持久化框架，不包含任何事务或回滚逻辑。
- 服务器端的领域对象使用声明式事务，并且对更新相关的方法使用 `Required` 事务属性，而对读取操作使用 `Supports` 属性。
- 为了保证 ACID 特性，客户端的对象决不开启事务，提交事务，或者将事务标记为回滚。
- 领域服务组件使用到的持久化对象，无论用到哪种持久化框架，不包含任何事务或回滚逻辑。<sup>\*</sup>
- 如果使用 EJB2.1 的实体 Bean，更新操作的事务属性必须设置为 `Mandatory`，并且不能使用任何回滚逻辑。对于读取操作，必须对容器管理持久化（Container-managed Persistence, CMP）的实体 Bean 设置 `Required` 属性 或对 Bean 管理持久化（Bean-managed Persistence）的实体 Bean 设置 `Supports` 属性。

---

\* 原文和第三条重复——译者

## 实现 ( Implementation )

下面的源代码展示了在 EJB 和 Spring 中这种模式的实现。为了简化编码示例的需要，我会假设 EJB 中的领域服务组件是位于远端的，而在 Spring 的例子中，则假设客户端代理和领域服务组件都是由 Spring 框架管理的。

### EJB

和客户端拥有事务的设计模式不同，本模式中只有一种组件类型（领域服务组件）包含事务代码。因此，由于客户端不包含任何组件代码，我们为了说明模式的实现，只是有必要展示领域服务部分的代码了。在 EJB 环境下，更新和读取操作部分的有关源代码如下所示（事务逻辑用粗体显示）：

```
@Stateless
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class TradingServiceImpl implements TradingService {
    public void updateTradeOrder(TradeData trade)
        throws TradeUpdateException {
        try {
            validateUpdate();
            TradeOrderDAO dao = new TradeOrderDAO();
            dao.update(trade);
        } catch (TradeUpdateException e) {
            sessionContext.setRollbackOnly();
            log.fatal(e);
            throw e;
        }
    }
}

@TransactionAttribute(
TransactionAttributeType.SUPPORTS)
```

```

public TradeData getTrade(TradeKey key)
    throws Exception {
    TradeOrderDAO dao = new TradeOrderDAO();
    return dao.get(key);
}
}
    
```

请注意 updateTradeOrder 方法在处理应用异常（被检查异常）时调用了 setRollbackOnly() 方法。并且请注意，为了和第 4 章谈到的最佳实践保持一致，bean 类缺省的被赋予了 Required 事务属性，而读取操作 getTrade() 重载了缺省行为，被赋予 Supports 事务属性。因为事先并没有事务上下文存在，在方法被调用时，因 Required 事务属性的缘故，新的事务将被自动创建。

## Spring 框架

在 Spring 环境中，本模式完全通过 Spring 的 XML 配置实现。EJB 实现中的 setRollbackOnly() 方法调用在 Spring 中实现为 bean 配置文件中的回滚规则指令。针对领域服务组件更新和读取操作的配置代码如下所示（事务逻辑部分用粗体显示）：

```

<!-- 定义领域服务 Bean -->
<bean id="tradingServiceTarget"
    class="com.trading.server.TradingServiceImpl">
</bean>

<bean id="tradingService"
    class="org.springframework.transaction.interceptor.
        TransactionProxyFactoryBean">
    <property name="transactionManager" ref="txnMgr"/>
    <property name="target" ref="tradingServiceTarget"/>
    <property name="transactionAttributes">
        <props>
            <prop key="*>
                PROPAGATION_REQUIRED,-Exception

```

```

        </prop>
        <prop key="get*>">PROPAGATION_SUPPORTS</prop>
    </props>
</property>
</bean>

```

由于本模式对领域服务模型用到了声明式事务的处理，在领域服务组件的更新和查询的 Java 代码中皆不包含任何事务逻辑。正如先前提到的，`setRollbackOnly()`的逻辑被 Spring 自动处理了，这是通过上面 XML 配置文件中的`-Exception`回滚规则实现的。下面的代码展示了在本模式中 Java 代码本身并不包含任何事务逻辑：

```

public class TradingServiceImpl implements TradingService {
    public void updateTradeOrder(TradeData trade)
        throws TradeUpdateException {
        validateUpdate();
        TradeOrderDAO dao = new TradeOrderDAO();
        dao.update(trade);
    }

    public TradeData getTrade(TradeKey key)
        throws Exception {
        TradeOrderDAO dao = new TradeOrderDAO();
        return dao.get(key);
    }
}

```

## 第九章 服务器端代理拥有事务的设计模式

当您在应用架构中用到命令模式（Command Pattern）或服务器端代理设计模式（Server Delegate Design pattern）时，本章描述的事务设计模式就比较适合了。在本模式中，服务器端代理组件，作为对服务器的远程接入点，拥有事务并负责对事务实施全面的管理。其他任何组件，包括客户端组件、领域服务组件、或是持久化组件都不负责管理事务，它们甚至不会察觉到它们正在使用到了事务。

命令模式是一种非常有用的设计模式，它解决了关于客户端事务管理以及 EJB 中的很多常见问题。这种设计模式背后最基本的原则是，客户端功能被包装在所谓“命令（command）”中，提交给服务器端以便执行。该命令可能包含了一个或多个对领域服务方法的调用。然而，这些领域服务方法的调用都是通过服务器端的被称为“命令实现（Command Implementation）”的对象去执行的，而不是在客户端执行。使用命令模式，使得用户可以从客户端对领域服务组件发起单一请求，并同时允许服务器端而不是客户端来管理事务处理过程。

服务器端代理模式的机制类似，唯一不同的是，它没有使用类似于命令模式的框架，而是简单地将客户侧的业务代理逻辑放置在服务器端的代理对象中去了。最终结果是一样的，事务处理过程被移到了服务器端，并且客户端对服务器的请求从多个减少到一个。

“服务器端代理拥有事务的设计模式”是“领域服务拥有事务模式”的特例。两者的主要区别在于，如果使用命令模式，会有“一个独立的对象”拥有事务，而不是由“一类组件”拥有事务。例如，如果您的应用包含 40 个不同的领域服务，在“领域服务拥有事务模式”下，可能由 40 个 bean 去管理事务。然而，在“服务器端代理拥有事务的设计模式”下，您只需要一个 bean（命令执行者，Command Processor）去管理事务。

### 背景（Context）

下面的代码示例展示了在 EJB 环境下，为了完成一个业务请求，客户端对象必须形成多次对服务器调用的场景。

```
public class ClientModel
{
    public void placeFixedIncomeTrade(TradeData trade)
        throws Exception {
```

```

InitialContext ctx = new InitialContext();
UserTransaction txn = (UserTransaction)
    ctx.lookup("java:comp/UserTransaction");
try {
    txn.begin();
    ...
    Placement placement =
        placementService.placeTrade(trade);
    executionService.executeTrade(placement);
    txn.commit();
} catch (TradeUpdateException e) {
    txn.rollback();
    log.fatal(e);
    throw e;
}
}
}
}
    
```

在这个实例中，客户端必须使用编程式事务来确保一个独立的事务性工作单元中满足 ACID 特性。在此情形下，不仅仅是客户端要负责事务管理，而且因为多次对（远端）领域服务的调用，性能也受到影响。还有，在这个例子中，领域服务由无状态会话 Bean 的 EJB 实现，更大程度上使得架构复杂化。

为了简化此类情景，我们需要将无状态会话 Bean 实现的领域服务重构为 POJO，从客户端移除事务逻辑，而后为所涉及的所有客户端请求构造一个单一的对服务器端的调用。我们可以使用命令模式或是服务器端代理设计模式去达到部分或全部的目标。当应用这两个模式中任何一个时，本来位于客户层的事务处理将被搬到服务器层了。下面的代码示例展示了利用命令模式后，改动过的客户端代码：

```

public class ClientModel
{
    public void placeFixedIncomeTrade(TradeData trade)
    }
}
    
```

```

throws Exception {
    PlaceFITradeCommand command =
        new PlaceFITradeCommand();
    command.setTrade(trade);
    CommandHandler.execute(command);
}
}

```

可以看出，这是对之前代码做了非常大简化的版本。然而，有一个问题留下来了，“事务处理的逻辑在哪里呢”？在这个场景中，可以使用“服务器端代理者拥有事务”设计模式，甄别由哪个组件负责事务管理，以及对这些组件声明式事务该如何设定。

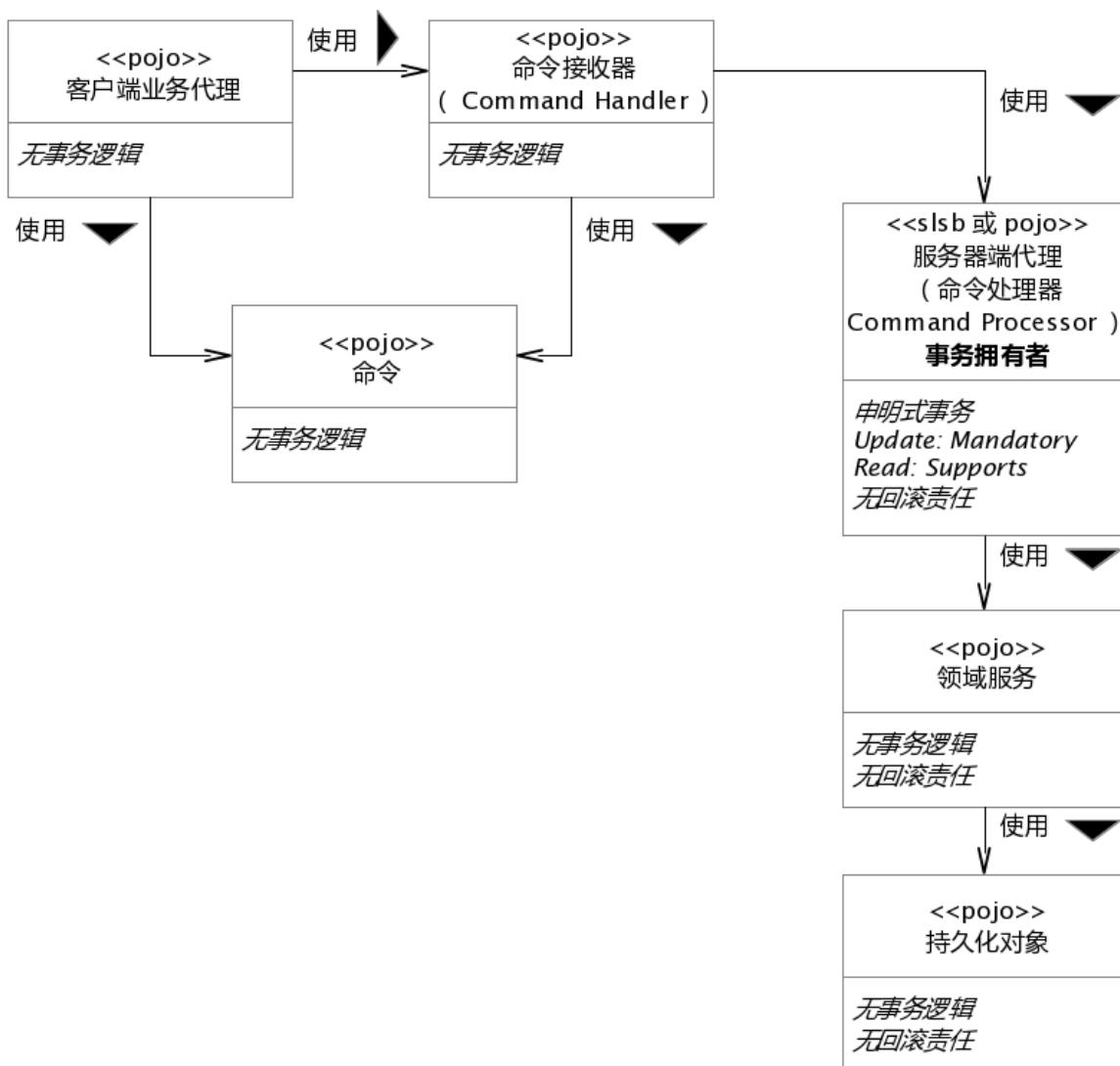
## 制约条件 ( Forces )

- 使用命令模式和服务器端代理设计模式设计应用架构和客户端与服务器的通信。
- 客户端总是与服务器端代理进行单一交互以完成业务请求。
- 必须保证 ACID 特性，意味着需要进行事务处理以维护数据完整性。
- 领域服务对象使用 POJO 实现，可以位于远程或本地。
- 命令处理器或服务器端代理是从客户端到领域服务的单一访问点。

## 解决方案 ( Solution )

当使用命令模式或服务器端代理设计模式时，服务器端代理者拥有事务设计模式能够作为此类应用架构的整体事务设计策略。该模式使用将整个事务管理的责任放在服务器端代理组件上的责任模型。当使用命令模式时，服务器端代理组件实现为单一的命令处理器 ( Command Processor ) 组件，该组件定位有关的命令实现对象并执行命令。在 EJB 中，该组件通常被实现为无状态会话 Bean。在 Spring 中，它可以被实现为 Spring 管理的 bean ( POJO )。当使用服务器端代理设计模式时，每个客户端请求的功能集合可以实现为彼此独立的服务器端代理 ( Spring 中的 POJO 或是 EJB 中的 SLSB )。

下图说明了在 EJB 和 Spring 框架两种情况下本模式的实现细节：



服务器端代理组件使用声明式事务，有关的更新方法被赋予 Required 的事务属性，而所有的读取方法被赋予 Supports 属性。服务器端代理组件在处理应用异常时，也会负责调用 setRollbackOnly() 方法。

这种模式适用的应用架构非常单一。在命令模式的用例下，服务器端代理被实现为命令处理器，简单地接收客户端发送来的命令对象，执行这些命令。在命令模式框架下自始至终使用接口 (interface) 的编程风格则保证了命令接收 (Command Handler) 组件、命令处理 (Command Processor) 组件、命令实现接口 (Command Implementation Interface) 以及命令接口 (Command Interface) 本身都保持通用和应用无关性。服务器端代理建立的事务上下文被传播到它调用到的所有对象。

在服务器端代理设计模式中，服务器端代理组件是作为对应用架构中领域服务组件的客户端门面存在的。在这种设计模式中，客户端的逻辑实际上被搬到了服务器，并放在客户端代理组件中。

这些设计模式的主要缺点是，服务器端代理组件包含了客户端的逻辑，而不是纯的服务器逻辑。并且，本模式在很多情况下比较难于实现，因为客户端业务代理常常是与使用的 web 框架紧密绑定的。这种情况直接的例子是 struts，在 struts 框架中 Action 类具备扮演客户端业务代理角色的能力（事实上很多时候它也是这么用的）。此外，也许将客户端逻辑搬走很难，因为其代码包含了对完全基于客户端的对象，例如 HttpSession、HTTPRequest，以及 HTTPResponse 的引用，这些对象要搬到服务器侧去是比较困难的。

然而，这两种设计模式最为明显的一个优势在于，包含处理请求业务逻辑主体的领域服务组件实现为 POJO（简单 Java 对象），而不是 EJB。因此领域服务组件从 EJB 框架中解耦合，使得它们易于测试。服务器端代理拥有事务的设计模式另一个独特之处在于，由于服务器端代理常常实现为一个无状态会话 Bean 的单例（singleton）组件，整个应用的事务逻辑位于单一的对象内。因此，从实现和维护的观点讲，它是最简单的事务设计模式。并且，该事务设计模式将事务管理的责任重担放在了领域服务组件的上面一层，将应用中服务器端的核心功能从事务管理之类的基础架构方面中解放了。通过使用本模式，领域服务组件可用 POJO 编写，由于它们不包含事务逻辑，在容器外的环境中测试就十分方便。

## 后果 ( Consequences )

- 客户端（无论哪种类型）不包含任何事务逻辑，不管理事务处理的方方面面。
- 由于服务器端代理组件开启和管理事务，更新方法在碰到应用异常时必须调用 setRollbackOnly() 方法。
- 服务器端代理建立的事务被传播到基于 POJO 的领域服务对象，同时也传播到领域服务用到的持久化对象（无论使用哪一种持久化框架）。而且，这些组件不包含任何事务或回滚的逻辑。
- 服务器代理对象使用声明式事务，对更新相关的方法应用 Required 的事务属性，对读取操作应用 Supports 属性。
- 为了维护 ACID 特性，客户端对象绝不开启事务、提交事务，或将事务标记为回滚。
- 持久化对象和领域服务不包含任何事务或回滚逻辑。

- 如果使用 EJB2.1 的实体 Bean , 更新操作的事务属性必须设置为 Mandatory , 并不要使用任何回滚逻辑。对读取操作而言 , 如果使用容器管理持久化 ( Container-managed Persistence , CMP ) , 需要为实体 Bean 设置 Required 的事务属性 ; 如果使用 Bean 管理的持久化 ( Bean-managed Persistence , BMP ) , 则需要设置事务属性为 Supports。

## 实现 ( Implementation )

下面的代码分别展示了 EJB 和 Spring 下这种模式的实现。为举例方便的需要 , 我们假设使用命令模式。对 EJB , 我假设命令处理器用无状态会话 Bean 实现 ; 对 Spring , 我假设命令处理器为 Spring 框架所管理。

### EJB

在命令模式下 , 该事务设计模式只有一个组件包含事务代码 ( 即事务处理器组件 ) 。因此 , 由于客户端不包含事务代码 , 我们仅仅有必要展示服务器端代理 ( 事务处理器 ) 关于这个模式的代码实现。 EJB 中的领域服务组件 , 针对更新和读取操作的代码如下所示 ( 事务逻辑用粗体表示 ) :

```

@Stateless
public class CommandProcessorImpl
    implements CommandProcessor
{
    @TransactionAttribute(
        TransactionAttributeType.SUPPORTS)
    public BaseCommand executeRead(BaseCommand command)
        throws Exception {
        CommandImpl implementationClass =
            getCommandImpl(command);
        return implementationClass.execute(command);
    }
}

```

```

@TransactionAttribute(
    TransactionAttributeType.REQUIRED)

public BaseCommand executeUpdate(BaseCommand command)
    throws Exception {
    try {
        CommandImpl implementationClass =
            getCommandImpl(command);
        return implementationClass.execute(command);
    } catch (Exception e) {
        sessionCtx.setRollbackOnly();
        throw e;
    }
}
}

```

上面例子中的 `getCommandImpl()` 方法使用反射 ( reflection ) 来装载和实例化命令实现对象。而后它被执行，然后通过命令对象向客户端返回结果。注意，该实现与“领域服务拥有事务的设计模式”非常相似，因为我们对读取操作附加了 `Supports` 的事务属性，并且对更新相关操作附加了 `Required` 的事务属性，并辅以 `setRollbackOnly()` 方法。

## Spring 框架

在 Spring 框架下，这个模式完全通过 Spring 的 XML 配置文件实现。在 EJB 实现中需要调用 `setRollbackOnly()` 方法，而在 Spring 中，通过配置文件中的回滚规则指令就可以处理了。以下的配置代码展示了本模式如何设置服务器端代理组件( 命令处理器 )去处理更新和读取操作 ( 事务逻辑加用粗体 )：

```

<!-- 定义服务器端代理命令处理器 -->

<bean id="commandProcessorTarget"
    class="com.commandframework.server.
        commandProcessorImpl">

</bean>

```

```

<bean id="commandProcessor"
      class="org.springframework.transaction.interceptor.
          TransactionProxyFactoryBean">

    <property name="transactionManager" ref="txnMgr"/>
    <property name="target" ref="tradingServiceTarget"/>
    <property name="transactionAttributes">
        <props>
            <prop key="executeUpdate">
                PROPAGATION_REQUIRED,-Exception
            </prop>
            <prop key="executeRead">
                PROPAGATION_SUPPORTS
            </prop>
        </props>
    </property>
</bean>

```

因为这个模式针对服务器端代理组件确定了声明式事务的使用，在领域服务组件中无论是更新还是读取的代码都不包含任何事务逻辑。如之前讲到的，`setRollbackOnly()`的逻辑被 Spring 自动处理了，我们可以从上面 XML 代码中看出端倪。下面则演示了在 Spring 中实现该模式不需要任何事务逻辑的 Java 代码：

```

public class CommandProcessorImpl
    implements CommandProcessor
{
    public BaseCommand executeRead(BaseCommand command)
        throws Exception {
        CommandImpl implementationClass =
            getCommandImpl(command);
        return implementationClass.execute(command);
    }
}

```

```
public BaseCommand executeUpdate(BaseCommand command)
    throws Exception {
    CommandImpl implementationClass =
        getCommandImpl(command);
    return implementationClass.execute(command);
}
```

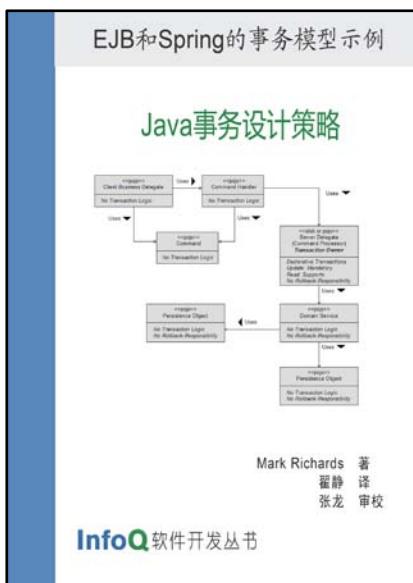
注意，在本例中，Spring 在两个方法中的实现实质上做了同样的事情。不像 EJB 实现，异常处理并不必要，因为 setRollbackOnly 逻辑已经包含在 XML 的回滚策略设置中了。

## 第十章 总结

通过理解三种不同的事务模型，以及使用本书讲解的各种事务设计模式，您能够简化您的事务处理，并为目标应用创建更健壮与高效的事务设计策略。事务处理非常重要，它也是保证应用与数据库数据完整性必不可少的环节。在 Java 中，事务处理并不一定是复杂的；使用本书讲解的事务设计模式让事务处理变得简单易懂，并且易于实现和维护。

## 关于作者

Mark Richards 是 IBM 认证的高级 IT 架构师，他在 IBM 公司从事大型系统面向服务架构的设计和架构工作，使用 J2EE 与其他技术，主要为金融行业服务。作者早在 1984 年起就加入软件行业，从开发人员做起，直至设计师、架构师。他在 J2EE 架构和开发、面向对象设计开发、系统集成方面具有丰富的专业知识和经验。Mark 在 1997 年和 1998 年任波士顿 Java 用户组社区的主席，从 1999 到 2003 年任整个新英格兰地区 Java 用户组的主席。除本书之外，他也是“NFJS 文选 2006 版”的作者之一（于 2006 年 6 月出版），同时他也是新英格兰 Java 用户组编码规范 SIG 出版的 Java 编码规范书籍的联名作者之一。Mark 是 IBM 认证的应用架构师、Sun 认证的 J2EE 业务组件开发工程师、Sun 认证的 J2EE 企业架构师、Sun 认证的 Java 程序员，以及 BEA WebLogic 认证的工程师/Java 讲师。他从波士顿大学获取了计算机科学硕士学位。Mark 是“No Fluff Just Stuff”论坛的定期发言人，经常在全美的各个用户群组与技术会议上作演讲。



# Java 事务设计策略

作者 : Mark Richards

翻译 : 翟静

审校 : 张龙

迷你书责任编辑 : 侯伯薇

迷你书美术编辑 : 胡伟红

本迷你书主页为

<http://www.infoq.com/cn/minibooks/JTDS>

本书属于 InfoQ 软件开发丛书。

如果您打算订购 InfoQ 的图书 , 请联系 [books@c4media.com](mailto:books@c4media.com)

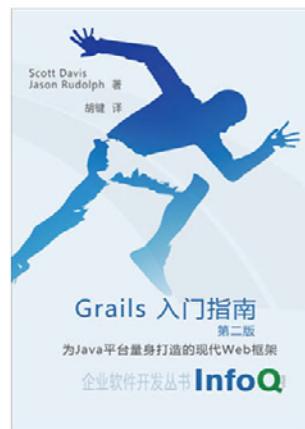
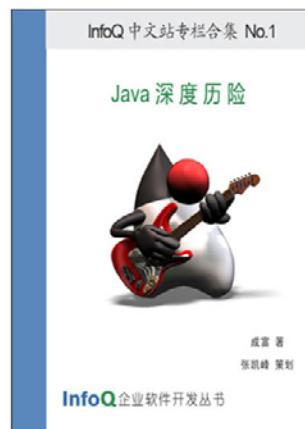
未经出版者预先的书面许可 , 不得以任何方式复制或者抄袭本书的任何部分 , 本书任何部分不得用于再印刷 , 存储于可重复使用的系统 , 或者以任何方式进行电子、机械、复印和录制等形式传播。

本书提到的公司产品或者使用到的商标为产品公司所有。

如果读者要了解具体的商标和注册信息 , 应该联系相应的公司。

# InfoQ 软件开发丛书

欢迎免费下载



商务合作: [sales@cn.infoq.com](mailto:sales@cn.infoq.com)

读者反馈/内容提供: [editors@cn.infoq.com](mailto:editors@cn.infoq.com)