Java 事务设计策略

Java Transaction Design Strategies

Mark Richards 著

翻译自 InfoQ 在线电子书, 原文地址:http://www.infoq.com/minibooks/JTDS .

译者: 低调求生存 (mail: wenbois2000@gmail.com)

目录

一.介绍

在如今，存在着许多需求不同的 Java 企业应用程序. 这些应用程序中一部分，可能只是使用 JavaBean 或 POJO 来实现的简单 web 应用. 但对于许多复杂的 N 端应用而言，大部分都采用了商业或开源的应用服务器所提供的技术，例如 EJB 的远程调用，使用 JMS 发送消息或使用 Spring 框架. 在这些复杂应用中，大多数都遭受着随机的以及有时无法解释的数据一致性问题.例如,不正确的账户余额,向用户发送多张订单,系统丢失订单信息以及常规的数据库中表之间的数据同步出错. 这类问题通常是由无效的事务管理或缺失事务管理策略所导致的.

在多年的编程工作中,我发现我所参与的大部分 Java 应用都没有一个有效的事务设计策略，大多数只是一种简单地依赖数据库去管理复杂事务的方式.我经常要求架构师与开发者描述他们的关于事务管理设计的观点. 而无论他们使用的是 EJB 或 Spring Framework，我经常得到的回复是: ”使用声明式事务”. 正如我们既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.

无论是开发者，设计者还是架构师，都应该关注事务处理问题. 对我们而言，有三件事是能够保证的: 死亡,纳税和系统失败. 除了像<<王牌大贱谍>>中那样被低温冷冻,我们对于死亡是束手无策的. 而无论我们活多久，纳税都是无法避免的. 幸运的是，三件事中我们还能做一件--使系统出现故障. 理解事务管理在 Java 中是怎样工作,并开发出一个有效的事务管理策略能够帮助我们在应用程序和数据库中避免数据不一致问题. 并减轻不可避免的系统故障所造成的损失.

正如本书标题所示，这本书是讨论基于 Java 的框架(EJB ,Spring 等)所提供的事务模型中怎样设计一个有效的事务管理策略的问题. 虽然我会在后面的章节中详述每个事务处理模型的实现技术，最佳原则和编程陷阱. 但在事务设计模式这一章，将会汇总这些事务模型的概念和实现技术，并描述怎样使用这些模型去有效地管理基于 EJB

或 Spring 的 Java 应用程序.

著名的数学家 Blaise Pascal 曾经写过:”这封信我写得比以往长了些，只是因为我没时间把它写短些”(I have made this letter longer than usual, only because I have not had the time to make it shorter.), 多数读者没有时间和耐性去看完一本沉长的事务管理书籍.所以这本书的写作目的也很明了: 用一种简明的方式, 将如何构建一个有效的事务管理策略传达给架构师和开发者. 这些策略在无论你使用什么框架的情况下都可用, 并且同时适用于大/小规模的 Java 企业程序.我写这本书的目标是帮助读者理解各种事务模型,理解它们的最佳实践,学习关于事务的设计模式, 以及理解怎样应用事务设计模式到各种不同的应用架构中.

我将会使用书中描述的 EJB2.1,EJB3.0 和 Spring Framework 中的例子来阐明事务管理的概念. 虽然完全地讨论任何一个上述以及其他持久化框架中的事务处理已经远超出了本书的范围, 但我希望这本书中的例子将会提供给读者足够的信息去理解这些概念和实现技术, 并以此来构建一个更有效的事务设计策略.

1) 事务模型

无论你是使用 EJB 或 Spring, 理解各种事务处理模型都是很重要的. 在 Java 事务管理中, 三种可用的模型分别为 本地事务模型, 编程式事务模型 以及 声明式事务模型. 虽然我将分别使用一章来详细介绍上述模型, 但这里先对各个模型做一个简短的摘要也是很有用的.

本地事务模型 正如其名所示, 它不是一个真正的事务管理框架. 而是一个本地资源管理者. 资源管理者是你所使用的数据源的实际提供者. 例如, 对于数据库(DB)而言, 资源管理者是 数据库管理系统(DBMS) 以及相应的 JDBC 驱动实现. 而对于 JMS, 资源管理者 是不同厂商所提供的队列(主题,Topic) 连接工厂的实现.在使用本地事务模型时,开发者管理的只是资源连接(Connections). 而不是事务. 此时, 真正管理着本地事务的是 数据库系统 或 JMS 的提供者. 我

将在第二章在详细谈论本地事务模型.

编程式事务模型基于 Java 事务处理 API (JTA) 和 其底层事务实现来提供事务支持. 这种事务模型去除了本地事务模型中无法手动控制事务的限制. 在使用这种模型时, 开发者编写代码来管理事务, 而不是简单地管理连接. 使用 Javax.transaction.UserTransaction 接口, 开发者可以用 begin() 方法来开始事务, 也可使用 commit() 或 rollback() 方法来终止事务. 虽然并不鼓励使用编程式事务, 这种模型仍常用于在 EJB 中为客户端访问远程无状态组件(Remote Stateless SessionBean) 提供事务支持.我们将在第三章对该模型进行更详细的学习.

声明式事务模型(在 EJB 中也称为容器管理的事务-CMT) 是我将在本书中主要谈论的事务模型. 在声明式模型中,事务框架或应用程序容器管理着事务的开始和结束(即 commit 或 rollback).此时开发者只需要向框架说明何时以及遇到何种类型的应用程序异常后回滚一个事务. 这通常需要在配置文件中进行配置. 例如, 对于 EJB 可以通过 XML 部署描述文件来配置(ejb-jar.xml). 而对于 Spring ,可以使用组件描述文件来进行配置(ApplicationContext.xml). 关于详细的声明式事务模型, 我将在第四章进行讨论.

2) ACID 在何处?

如果在 60 年代, 这段的内容可能是最吸引人的. 可惜, 对于今天的技术世界而言, ACID 的意义与过去相比是大相径庭的(ACID 亦指 60 年代的当红摇滚乐队). 在技术术语中, ACID 是事务所持有的属性名缩写, 这些属性为: 原子性(Atomicity), 一致性(Consistency), 隔离性(Isolation) 和 持久性(Durability).

原子性指一个事务必须同时提交或回滚. 它所作的更新操作必须处于同一个工作单元. 无论做了多少次更新, 所有的操作必须被视为一个集合的整体. 这个属性有时也指 逻辑工作单元(Logical Unit Of Work, LUW) 或 单一工作单元(Single Unit of Work, SUW).

一致性指在一个活动事务的执行中，数据库将处于一个一致的状态. 在事务环境中,这意味着每当一个插入,更新或删除操作在数据库执行的时候，如果不提交事务，数据库的完整性约束仍是有效的. 对于开发者，这也会产生影响. 例如，一致性意味着一个事务执行的时候，你不能先添加一条详情记录，然后再添加一条摘要记录(此时详情表外键引用到摘要表). 虽然有些数据库允许你将约束检查延迟到提交时刻，但你通常无法避免事务执行时的外键约束问题,即使在事务执行的后面你想矫正这个错误.

隔离性指多个独立事务之间的交互深度.这个属性决定在多个事务访问相同信息时，怎样保护未提交的数据. 隔离性是一致性(Consistency)与并发性能(Concurrency)之间的杠杆. 随着隔离级别的增加，一致性也随着增加，但并发性能要随之下降. 我将会在第四章对事务隔离进行更多的讨论.

持久性指当一个事务成功地提交后,数据库或JMS提供者将会保证我们所作的更新生效，并可以抵御一个系统性的失败(例如将一瓶啤酒倒在服务器上,或在服务器上玩我们喜欢的多人游戏-导致服务器卡死). 几个主要的数据库提供厂商中，虽然一部分所使用的复杂缓存策略被质疑不能再在一些更新中保持一致性，但一般而言当我们收到一个成功提交的消息后，我们可以保证所作的更新将是永久的并且不会丢失.

3) JTA 与 JTS

为了有效地在企业 Java 应用中管理事务，开发者并不需要知道后端 JTS(Java Transaction Service)规范的详细情况. 但是，无论你使用何种具体技术，它对于理解 Java 中分布式事务处理的限制而言,都是很重要的.

无论你使用什么框架，对于事务管理而言，大多数 Java 企业应用都是基于 JTA. JTA 是一组可供开发者们管理事务的接口. 但另一方面, JTS 是 JTA 的底层事务实现者. JTS 也在很多商业以及开源应用程

序服务器中得到了使用(值得注意的是,除了 JTS 外, 在市场上也存在其他的事务服务). JTA 与 JTS 的关系可以想象为类似 JDBC 与其相应的底层数据库驱动一样. JTA 等效于 JDBC API, 而 JTS 则等效于相应的实现驱动. JTA 可以通过商业的应用程序服务器(JTS)或者开源的事务管理者(例如 Jboss Transaction Service 或 JOTM) 来实现.

Java 事务服务(JTS) 是 CORBA OTS 1.1 规范(Object Transaction Service) 的 Java 版本. 对于开发者而言, 这一点并不是十分的重要. 除非你需要实现一些特殊的需求或者进行高难度的面试. JTS 并不受管于 J2EE 体系, 而是受管于各种实现者. 它们之间以此来协调分布式事务的互操作性. 但因为 JTA 必须支持 JTS 实现与非 JTS 实现, 只是简单地通过查看 JTA 接口是很难精确地找出它们所支持的功能的. 例如, 虽然 JTS 规范对嵌入事务有可选的支持, 但是 J2EE 并不支持这个特性.所以你并不能从 JTA 上区分出这些不同.

幸运的是,在处理事务的时候,开发者只需要关心为数不多的几个接口.例如,当使用编程式事务的时候, 唯一一个我们需要使用的接口是 javax.transaction.UserTransaction. 这个接口允许我们编程来开始一个事务, 提交一个事务, 回滚一个事务 并且获取事务的状态.

当在 EJB 中使用声明式的时候, 我们主要关心的是 EJBContext 接口中的 SetRollbackOnly()方法. 使用 javax.transaction.*包里的类, 我们能够做更多的事, 但开发者并不常使用这些功能. 例如, 就像我们将在将在后续章节中看到的一样, 有时候我们或许需要直接访问事务管理器.

当使用声明式事务的时候, 我们需要手动地挂起(suspend) 或 恢复(resume)一个事务. 我们也能开始,提交和回滚一个事务.这些都可以使用 TransactionManager 接口来完成.

4) UserTransaction 接口

UserTransaction 是编程式事务模型中唯一一个需要使用到的接口, 主要在 EJB 中使用. 在这个接口中,开发者所需关心的几个方法如下:

- begin()
- commit()
- rollback()
- getStatus()

javax.transaction.UserTrasnaction.begin()

这个方法用于在编程式事务模型中开始一个新的事务并且关联事务到当前线程.如果有一个事务已经关联到当前的线程并且底层的事务实现不支持嵌套事务的话,这个方法将会抛出一个NotSupportedException.

javax.transaction.UserTrasnaction.commit()

这个方法用于在编程式事务模型中提交一个关联到当前线程的事务并且终止事务. 这个方法也会去除事务与当前线程之间的关联. 在 Java 中, 只有单个事务能够被关联到当前的线程. 在 XA 环境下, 这个方法可能会抛出一个 HeuristicMixedException 或者 HeuristicRollbackException 来表明资源管理者做出了一个不依赖于事务管理者的操作. 这可能是两阶段提交过程中第一阶段到第二阶段之间的回滚或者部分提交所导致的. 我将会在本书的第五章详细讨论两阶段提交(Two-Phase Commit) 和启发式异常(heuristic Exception)处理.

javax.transaction.UserTrasnaction.rollback()

这个方法用于在编程式事务模型中回滚一个关联到当前线程的事务并且终止事务.这个方法也会去除事务与当前线程之间的关联.

javax.transaction.UserTrasnaction.getStatus()

这个方法用于在编程式事务模型中返回一个整数值来指示当

前事务的状态. 初看之下这个返回的整数值是毫无意义的, 但是我们能够使用 javax.transaction.Status 接口中常量来判断所获取的是哪一个状态. 我将会在本章后面讨论 java.transaction.Status 接口.

5) TransactionManager 接口

TransactionManager 接口主要用在声明式事务模型中, 它可以完成编程式事务模型中 UserTransaction 的所有功能(拥有 UserTransaction 中所有的方法). 但是,对于大部分方法最好是使用 UserTransaction 接口, 而非 TransactionManager. 除非你需要挂起(suspend)或者恢复(resume)一个事务.

javax.transaction.TransactionManager.suspend()

这个方法用于在声明式事务模型或编程式事务模型中挂起关联到当前线程的事务. 该方法返回一个当前事务的引用. 如果没有事务关联到当前线程, 则返回 null. 当我们需要挂起当前的事务来执行一些代码或调用一些不兼容 XA 的存储过程时,这个方法将会变得很有用. 我们将会在本书的第五章看见一个使用该方法的例子.

javax.transaction.TransactionManager.resume(Transaction)

这个方法用于在声明式事务模型或者编程式事务模型中恢复一个前面挂起的事务.它使用一个之前挂起事务的引用作为参数, 将事务关联到当前线程, 并接着恢复事务环境.

6) EJBContext 接口

EJBContext 接口主要用在 EJB 中的声明式事务模型. 这个接口只包含一个对于事务管理有用的方法. 这个方法为 setRollbackOnly().

javax.ejb.EJBContext.setRollbackOnly ()

这个方法用于在声明式事务模型中通知容器当前事务只能够回滚.对于这个方法, 有趣的在于它并不是在调用的时候立即回滚事务,而只是标记当前事务为回滚. 此时调用 getStatus()方法将返回 STATUS_MARKED_ROLLBACK 状态. 相同的结果也可以通过调用 TransactionManager.setRollbackOnly()来达到, 但是因为我们已经获取了 SessionContext 或者 MessageDriverContext. 所以最好使用该方法替代.

7) Status 接口

正如我们前面段落中看到的,我们使用 javax.transaction.Status 接口来获取事务的状态. UserTransaction.getStatus()方法返回的结果在这个接口中进行了声明. 我用一节来写事务状态是因为这些状态排列很整洁并且可以提供给我们许多关于当前事务状态的有用信息. Status 接口包含下列常量值:

- STATUS_ACTIVE
- STATUS_COMMITTED
- STATUS_COMMITTING
- STATUS_MARKED_ROLLBACK
- STATUS_NO_TRANSACTION
- STATUS_PREPARED
- STATUS_PREPARING
- STATUS_ROLLEDBACK
- STATUS_ROLLING_BACK
- STATUS_UNKNOW

在上面所有列出的状态中, 对于大部分的 Java 应用而言真正用到的有: STATUS_ACTIVE, STATUS_MARKED_ROLLBACK 和

STATUS_NO_TRANSACTION. 下面的小节将描述这些状态值.

STATUS_ACTIVE

有时可能查看是否有事务关联到当前线程是很重要的.例如,出于调试或者优化目的,我们可能想添加一个切面(Aspect)或者拦截器(Interceptor)来在查询操作时检查当前事务是否存在. 使用这个切面和状态值, 我们能够侦测到对于我们的事务设计策略中可能存在的优化点. 其他情况下如果我们想要挂起当前事务或者执行一些可能导致应用失败的代码(例如在 XA 下执行一段包含 DDL 语句的存储过程)时判断该状态是很有用的.下面的代码片段显示了 STATUS_ACTIVE 状态的使用:

```
// ...
if(txn.getStatus() == Status.STATUS_ACTIVE){
    logger.info("Transaction active in query operation");
}
// ...
```

在这个例子中, 我们有一个记录当前查询方法中是否存在事务的逻辑. 这可能是一种迹象表明我们在不需要事务的地方出现了事务, 因此也在我们的整体事务设计策略中找出了一个可能的优化机会或设计问题.

STATUS_MARKED_ROLLBACK

当使用声明式事务模型的时候, 这可能是个很有用的状态.出于性能优化目的, 如果当前事务已经在前面的方法调用中被标记为 rollback,我们可能想跳过一些耗时的处理. 因此, 如果我们想查看当前事务是否已经被标记为 rollback. 我们可以检查它的状态, 如下面代码所示:

```
// ...
if(txn.getStatus() == Status.STATUS_MARKED_ROLLBACK){
    throw new Exception(
          "Further Processing Halted Due to Txh Rollback");
}
```

```
// ...
```

STATUS_NO_TRANSACTION

这个状态通常是很有用的，因为它是唯一能确定当前是否处于无事务环境的状态．像 STATUS_ACTIVE 一样，这个状态也能够被用来在调试或优化时来检测在应该存在事务的地方是否有事务存在．使用切面或者拦截器(或者内嵌代码)．我们能够检测到在事务设计策略中对那些需要进行更新操作的方法可能存在的缺陷.下面的代码显示了 STATUS_NO_TRANSACTION 状态的使用:

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

需要注意的是我们不能简单地检查一个状态不等于 STATUS_ACTIVE 来判断没有事务．缺乏 STATUS_ACTIVE 状态并不表明当前没有事务环境．因为有可能是处于一个上述状态列表中其他状态的事务环境中.

8) 总结

接下来的三个章节中将集中到不同的事务模型上．虽然这本书的概念主要通过 EJB 来举例．我也提供了 Spring 中的例子．所有持久化框架的完整例子必然是超出了本书的范围．但我还是鼓励你去阅读本书来理解这些事务处理模型的概念,技巧,编程陷阱以及最佳实现．并且将这些信息转换成你所用的持久化框架中的具体实现.

二.本地事务模型

本地事务模型这个词是指事务管理是被底层的数据库系统或 JMS 消息提供者来处理的．从开发者的角度来说，我们并不需要在本地事务模型中管理事务．相反,我们真正管理的是数据源连接

(Connections). 下面的代码说明了在 JDBC 代码中本地事务模型的使用.

```
public void updateTradeOrder(TradeOrderData order)
        throws Exception
{
    DataSource ds = (DataSource) InitialContext
                .doLookup("jdbc/OracleDS");
    Connection con = ds.getConnection();

    con.setAutoCommit(false);

    Statement stmt = con.createStatement();
    String sql = "update trade_order ....";
    try{
        stmt.executeUpdate(sql);
        con.commit();
    }
    catch (Exception e){
        con.rollback();
        throw e;
    }
    finally{
        stmt.close();
        con.close();
    }
}
```

注意上面的代码中 Connection.setAutoCommit(false) 与 Connection.commit() 和 Connection.rollback() 方法的关联使用. setAutoCommit()方法是基于开发者的连接管理中很重要的一部分. autoCommit 这个标志是用于告诉底层的数据库系统是否应该在每条 SQL 执行后立即提交连接. 这个值为 true 时数据库系统将会在每条 SQL 语句执行后立即提交或回滚连接. 与之相反, 当这个值为 false 时数据库系统将会保持连接为活动状态并且不会提交所作的变更,直到显式地调用了一个 commit()方法. 默认情况下, 这个标志通

常设置为 true. 因此, 默认情况下如果我们执行多条 SQL 更新语句, 这些语句中每条的执行和提交都是独立于其他的.并且此时的 Connection.commit()和 Connection.rollback()调用将会被忽略.

对于 Spring 框架中的低级别 JDBC 代码, 你可以简单地使用 org.springframework.jdbc.datasource.DataSourceUtils 来执行前面的代码:

```
package com.trading.server;
public class TradingService
{
  private DataSource ds;

  public void updateTradeOrder(TradeOrderData order)
   throws Exception
  {
     Connection con = DataSourceUtils.getConnection(ds);

     con.setAutoCommit(false);

     Statement stmt = con.createStatement();
     String sql = "update trade_order ....";
     try{
        stmt.executeUpdate(sql);
        con.commit();
     }
     catch (Exception e){
        con.rollback();
       throw e;
     }
     finally {
        stmt.close();
        con.close();
     }
  }
}
```

在Spring中，数据源和相应的业务对象可以在Spring配置文件中定为，如下所示:

```
<bean id="dataSource"
    class="org.springframework.jndi.JndiObjectFactoryBean">
   <property name="jndiName" value="jdbc/OracleDS"/>
</bean>
<bean id="TradingService"
       class="com.trading.server.TradingService">
   <property name="ds">
      <ref local="dataSource"/>
   </property>
</bean>
```

如果你使用的是 Spring,对于本章中剩余的代码你可以按照上面的示例来替换 DataSource 查找和 getConnection()方法的调用.

1) 自动提交与连接管理

无论你使用的是 EJB 或 Spring,自动提交标志是本地事务模型中非常重要的元素. 默认情况下，这个标志通常设置为 true. 这表明数据库系统应该在执行每条更新语句后提交(或回滚)连接. 考虑下面的 JDBC 代码例子. 这个代码没有使用连接管理且只有单条更新语句. 因为自动提交标志默认是设置为 true. 底层的数据库系统将会管理连接并提交(回滚)更新语句.

```
public void updateTradeOrder(TradeOrderData order)
      throws Exception
{
   DataSource ds = (DataSource) InitialContext
            .doLookup("jdbc/OracleDS");
   Connection con = ds.getConnection();
   Statement stmt = con.createStatement();
   String sql = "update trade_order ....";
   try{
      stmt.executeUpdate(sql);
```

```
    }
    catch (Exception e){
       throw e;
    }
    finally{
        stmt.close();
        con.close();
    }
}
```

上面的代码能正常工作只是因为该方法中只存在单条更新语句. 但是, 让我们假设在同一个方法中需要执行多条更新语句. 如下:

```
public void updateTradeOrder(TradeOrderData order)
        throws Exception
{
    DataSource ds = (DataSource) InitialContext
                .doLookup("jdbc/OracleDS");
    Connection con = ds.getConnection();
    Statement stmt = con.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();
        con.close();
    }
}
```

上面的代码示例中我们多添加了一条语句来重新计算当前交易的费用,并更新该交易订单对应表中的记录为新的费用. 上面的代码可

以编译并执行，但是它并不支持 ACID 属性. 首先，因为默认情况下自动提交标志是设置为 true，这样当第一个 executeUpdate()方法执行后当前连接将会被提交. 如果第二个 executeUpdate()语句执行失败，这个方法将会抛出一个异常. 但是第一条 SQL 执行的结果已经提交完成了. 二者没有处于同一原子范围内(同时成功或同时失败)，因此违犯了 ACID 中的原子性. 其次，这些更新操作作为逻辑工作单元(LUW)中的一部分,是没有与其他可能访问相同表或记录的操作相隔离的,因此违犯了 ACID 中的隔离性.

为了让上面的代码从事务和 LUW 的角度正常的工作，我们必须设置自动提交标志为 false，并且在代码中添加连接提交和回滚的逻辑. 通过将自动提交标志设置为 false，我们告诉了底层的数据库系统我们将会手动管理连接并且进行 commit()和 rollback()操作. 在这种方式下,我们能将多条 SQL 更新操作聚集在一起,从而在具有原子性的事务中组成为一个单一的逻辑工作单元(LUW).下面的代码列表显示了如何管理多个更新操作.

```
public void updateTradeOrder(TradeOrderData order)
      throws Exception
{
   DataSource ds = (DataSource) InitialContext
            .doLookup("jdbc/OracleDS");
   Connection con = ds.getConnection();
   con.setAutoCommit(false);
   Statement stmt = con.createStatement();
   String sqlOrder = "update trade_order ....";
   String sqlTrade = "update trade_fee ....";

   try{
      stmt.executeUpdate(sqlOrder);
      stmt.executeUpdate(sqlTrade);
       con.commit();
   }
   catch (Exception e){
     con.rollback();
```

```
            throw e;
        }
        finally{
            stmt.close();
            con.close();
        }
}
```

通过在上面的代码中添加 con.setAutoCommit(false) 与 commit()和 rollback()操作，两条 SQL 更新操作将会被视为一个单一的工作单元.

为了证明本地事务模型的缺点,请考虑下面的代码，其中我们将 SQL 更新操作移动到不同的 DAO(Data Access Object)中:

```
public void updateTradeOrder(TradeOrderData order)
        throws Exception
{
    OrderDAO orderDao = new OrderDAO();
    TradeDAO tradeDao = new TradeDAO();
    try{
        // SQL与连接逻辑移动到DAO中
        orderDao.update(order);
        tradeDao.update(order);
    }
    catch (Exception e){
        logger.fatal(e);
        throw e;
    }
}
```

OrderDAO 与 TradeDAO 对象均包含前面示例代码中的 SQL 更新与连接逻辑.在这个例子里，无论每个 DAO 中的自动提交标志是否设置,每个更新操作都是独立的处理的. 这意味着每个 update()方法调用后数据库都会提交执行结果.

我们可能会使用连接对象传递的技术来尝试解决这个问题.使用连接对象传递时，我们在高级别的方法中建立数据库连接，并传递

这个连接到 DAO 对象的 update()方法中. 我们能修改代码上面的示例代码为使用连接对象传递的方式, 这样可以使其兼容 ACID. 下面的代码显示了一个使用连接对象传递的例子.

```
public void updateTradeOrder(TradeOrderData order)
        throws Exception
{
    DataSource ds = (DataSource) InitialContext
              .doLookup("jdbc/OracleDS");
    Connection con = ds.getConnection();
    con.setAutoCommit(false);

    OrderDAO orderDao = new OrderDAO();
    TradeDAO tradeDao = new TradeDAO();
    try{
        // SQL 逻辑移动到DAO中
        orderDao.update(order,con);
        tradeDao.update(order,con);
        con.commit();
    }
    catch (Exception e){
       logger.fatal(e);
       con.rollback();
       throw e;
    }
}
```

注意上面代码中曾用在 DAO 对象中的数据库连接逻辑现在已经移动到调用方法中. 连接对象是被传递到 DAO 对象中, DAO 对象通过传入的连接来获取 Statement 并执行更新操作, 之后返回连接对象给调用者.

虽然这种技术在大多数情况下能够工作,但它并不是一种有效(率)的事务设计策略.使用连接对象传递很容易出错并且需要花费大量的编程劳动来维护.如果你最终的编码是与上面的相似的话, 你是时候弃用本地事务模型并选用编程式事务或者声明式事务模型了.

2) 本地事务的思考与限制

本地事务模型在简单的更新操作和小型应用中工作的很好.但是,只要我们添加一些复杂逻辑到我们的应用中, 这种模型将会出现故障.下面是这种模型的几种限制,这些缺点能够对你的应用架构造成重大的限制.

本地事务模型的第一个问题是它很容易使开发者在编写连接逻辑时出错. 开发者必须将注意力高度的集中到自动提交标志的设置上,特别是当在同一个方法内进行多个更新操作的时候. 此外, 开发者也必须总是注意被调用的方法没有自己管理连接. 除非你的小型应用里大部分都是单表更新操作, 否则是没有有效的方法来确定在你的应用中, 一次请求只保证出现在单个事务化的工作单元内.

本地事务模型的另一个问题就是本地事务模型在使用 XA 全局事务(我们将在第五章介绍)协调多个资源的时候,是不能同时并发存在的.当协调诸如数据库,JMS 目的地(队列或主题)的多个资源时,你不能在保持 ACID 属性的情况下使用本地事务模型.

根据所述的这些约束与限制,本地事务模型应该只被用于只有简单的表更新操作的,基于 web 的小型 Java 应用.

三.编程式事务模型

编程式事务模型与本地事务模型最大的区别在于前者管理的是事务,而不是连接. 当使用 EJB 的时候, 编程式事务模型有时也指 Bean 管理的事务(Bean-Managed Transactions, BMT).”BTM” 这个词并不是使用的非常多.因为编程式事务模型可以应用于 Servlet 容器以及 EJB 容器中, 并且也可以用于 POJO 中(并不只是 EJB). 下面的代码显示了在 EJB 中用 JTA 来进行编程式事务处理.

```
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){
        logger.fatal(e);
        txn.rollback();
        throw e;
    }
}
```

上面的代码示例中事务环境(context)传播到了 TradeOrderDAO 对象中,所以这里不像本地事务模型一样需要管理连接或事务. TradeOrderDAO 所需要做的就是从连接池获取连接并返回.

在编程式事务模型中,开发者负责的是事务的开始与终止.对于 EJB, 这是通过 UserTransaction 接口来完成的. 该接口中的 begin()方法用于开始一个事务. Commit()与 rollback()方法用于终止该事务. 而在 Spring 中,这是通过使用 org.springframework.transaction 包中的 TransactionTemplate 或 PlatformTransactionManager 来完成的.

虽然通常是不鼓励使用编程式事务模型的, 但也有一些该模型非常适用的场景.这些场景将会在 3.4 小节中进行讨论.如果我们在 EJB 中使用编程式事务, 我们可以通过在 ejb-jar.xml 部署描述文件中设置 <transaction-type>的值为 Bean, 来告诉应用程序服务器(特别是 EJB 容器)我们想要自己管理事务,让其使用编程式事务模型.正常境况下,<transaction-type>的默认值是 Container.它表明我们想要使用声明式事务(下一章讨论). 编程式或声明式模型的使用是可以在每个 Bean 的配置处进行选择的. 这意味这你可以混合使用编程式与声明式模型. 但是,这并不是一个好的实践且应该避免出现这种情况.

对于 EJB 3.0, 你可以使用元信息注解来说明使用编程式事务.此时可以用 @TransactionManager 注解来完成, 如下所示:

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

而在 Spring 框架中,你可以选择使用 TransactionTemplate 或 PlatformTransactionManager. 下面的代码显示了最常使用的 TransactionTemplate 技术:

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

Bean 配置文件:

```
<bean id="transactionTemplate"
    calass="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>
```

正如你在上面例子中看到的,Spring 用一个事务回调来将业务逻辑包裹起来，使其处于事务环境中．请注意，在使用这种技术的时候你不必像在 EJB 中那样显式的调用 begin()与 commit()方法．此外,事务的回滚是通过 TransactionStatus.setRollbackOnly()方法来处理的，而不是 EJB 中的 Transaction.rollback()方法.

1) 获取一个 JTA 用户事务(UserTransaction)

当在 EJB 客户端(基于 Web 或应用程序)中使用编程式事务模型时，你必须获取 InitialContext 并且进行 JNDI 查找，如下所示:

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

在这里事情变得稍微复杂了，上面代码片段中的查找名(“javax.transaction.UserTransaction”)是依赖于具体的应用程序服务器的.这意味着这段客户端代码将不能交叉多个应用服务器运行．每个应用服务器绑定 UserTransaction 到其特殊的 JNDI 名上.下面的列表显示了大部分流行应用程序服务器所绑定的 JNDI 名.

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”

更复杂的情况中,假设我们想在容器外部(不基于 web 的应用)访问一个使用声明式事务的 EJB 或是使用 Junit 进行单元测试时.我们是不能使用上述代码的. 因为JNDI 绑定通常在容器环境外部是无效的.

这在使用 JUnit 对 EJB 进行测试或在基于 Swing 的程序中访问事务时也产生了一些问题.这个问题的解决方法是可以使用 TransactionManager 接口来开始一个事务.你可以首先使用 Class.forName() 方法来加载具体的应用程序服务器的 TransactionManagerFactory 类. 然后使用反射来执行 getTransactionManager()方法. 这个方法返回一个可用来开始和停止事务的事务管理器.下面的代码片段显示了在 IBM WebSphere 中使用 TransactionManagerFactory:

```
//装载事务管理器工厂类
Class<?> txnClass = Class
 .forName("com.ibm.ejs.jts.jta.TransactionManagerFactory");

//使用反射来调用getTransactionManager()方法.
//以便得到一个事务管理器的应用
TransactionManager txnMgr = (TransactionManager) txnClass
   .getMethod("getTransactionManager").invoke(null);

//通过事务管理器来开始一个事务
txnMgr.begin();
```

虽然上面的代码使用的是 IBM WebSphere 的事务管理器工厂, 但你也可以通过替换相应的类名来在其他应用程序服务器中使用. 这段代码可以添加到 Junit 测试用例, 应用程序测试客户端, 甚至是基于 Swing 的应用中来获取一个容器环境外事务控制.

对于容器内的 EJB, 如果是会话(Session) Bean, 你可以使用 SessionContext.getUserTransaction()方法来获取. 如果是消息驱动 Bean(MDB), 你可以使用 MessageDriverContext.getUserTransaction()方

法来获取. 此时不需要进行转型操作.下面的代码显示了在 SessionContext 中获取 UserTransaction 接口:

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

2) 编程式事务编码陷阱

在使用编程式事务的时候, 开发者必须特别关注异常处理.考虑一下在下面的代码中, 我们忽略了运行时异常并且只捕捉了应用(checked)异常.

```
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){
        logger.fatal(e);
        txn.rollback();
        throw e;
    }
}
```

此时如果我们的程序中出现了一个运行时异常(例如 NullPointerException 或 ClassCastException), 容器将会自动的为我们回滚当前的事务.但是, 如果我们没有进行任何的异常处理会发生什么呢?

```
public void updateTradeOrder(TradeOrderData order)
        throws Exception
{
    UserTransaction txn = sessionCtx.getUserTransaction();
```

```
    txn.begin();
    TradeOrderDAO dao = new TradeOrderDAO();
    dao.updateTradeOrder(order);
    txn.commit();
}
```

在上面的代码里，我们通过方法签名中的 throws Exception 来委托处理了所有的异常. 此时如果我们的代码中产生了一个应用程序异常(checked)，将会返回下面的异常:

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

使用编程式事务意味着开发者是负责管理整个事务. 一个方法开始了事务后也必须终止这个事务.因为这是一个运行时异常,这个错误可能不总是在测试中被捕捉到. 对于复杂的系统来说是更是如此.

现在假设我们忘记在我们的编程式事务中使用 commit()方法，如下面代码所示:

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

如果我们执行这段代码我们将会再次得到上述的异常.

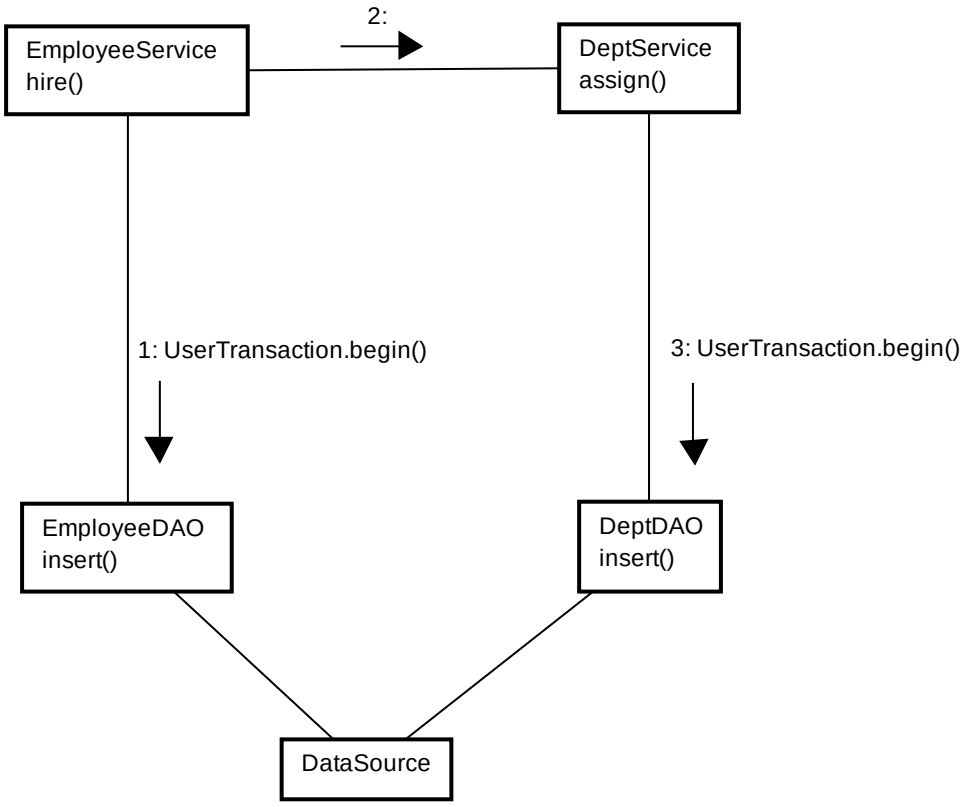
就像你在你前例子中看到一样,当使用编程式事务时,因为关联到

异常处理的原因,开发者必须非常关注事务的管理. 开发者必须保证在方法中开始一个事务后总是终止当前的事务. 这通常是弊大于利的.特别是对于一些大型应用中复杂的异常处理.

3) 事务环境(Context)问题

在 EJB(特别是无状态会话 Bean)中,编程式事务模型存在一个严重的架构限制. 我称这个问题为事务环境问题(Transaction Context Problem).该问题的最大限制在于你不能在从一个 Bean 中传递事务环境到另外一个 Bean. 但是,在 EJB 中使用声明式事务时,你却可以传递一个起始于 EJB 或者客户端对象中的编程式事务环境.

请考虑一个场景:我们有两个别分叫做 EmployeeService 和 DeptService 的无状态会话 Bean. 其中 EmployeeService 处理所与雇员有关的功能(例如雇佣一个雇员,给一个雇员加薪等), 而 DeptService 处理所有与部门有关的功能(例如添加,删除和分配雇员).两者都是使用编程式事务. 此外,可以假设两个 EJB 都有相应的 DAO.下面的图描述了该场景:



在上述服务组件作为无状态会话 Bean 实现时,内部服务之间的通信将会产生事务环境问题.考虑上图中, 当我们雇佣(hire())一个雇员的时候我们也应该同时分配该雇员到相应的部门. 因此 ,EmployeeService.hire() 方法在相同的业务事务中调用了 DeptService.assign()方法. 我们期待的情况是在 hire()方法中开始的事务能够传播到 assign()方法中去. 因而两个操作将会处于单个事务当中. 但是因为事务在这种情况下是不可能传播到 assign()方法中的,所以上述做法也是不可行的. 此外,因为 DeptService 是作为一个可以被单独调用的服务组件, 其 assign()方法也是处于一个独立的线程中,并使用 UserTransaction.begin()方法来开始它自己的事务.

事务环境问题的一个重要特点在于除非你对该场景进行特殊的测试,否则它在正常情况下并不会浮现.上面的场景实际上发生了如下事情:

- Hire()方法开始一个事务,执行了必要的插入操作并通过 EmployeeDAO 来更新 Employee 表.

- 接下来,hire()方法调用了 DeptService 中的 assign()方法.
- 此时 assign()方法属于一个不同的无状态会话 Bean.所以其产生了一个独立的线程来执行并开始了一个自己的事务.而 hire()方法所产生的事务只是处于等待状态直到 assign()方法执行完成.
- 执行控制权传递回 hire()方法,此时恢复了原始的线程和相应的事务的执行.

这种情况下我们没有符合 ACID 中的隔离性和原子性, 因此也没有事务管理. 如果原来开始于 hire()方法中的事务在 assign()方法执行后进行了回滚操作, 这将会在数据库中留下不一致的状态. 因为开始于 assign()中的事务在此时已经提交了.

4) 编程式事务适用的情况

虽然在一般情况下使用编程式事务是不推荐的, 但还是有许多编程式事务模型适用的场景.

使用编程式事务的最常见场景为使用客户端初始的事务. 比如如果客户端为了单项业务而进行了多次远程调用,此时事务必须由客户端来开始. 当使用 JTA 时这需要使用 UserTransaction 接口并进行编程式事务处理. 在这种情况下客户端必须使用编程式模型而远程 EJB 使用声明式事务模型. 这是因为事务环境是不可能传递给一个使用编程式事务模型的托管 EJB 的.

另一个可能的适用场景为局部化 JTA 事务. 由于 JTA 事务处理对于性能和资源使用而言是非常昂贵的, 因而你有时可能会想尽可能的提高应用的性能. 当宝贵的资源(例如数据库或消息队列)已经很匮乏, 而事务的使用可能会对整个系统的性能造成影响. 此时,为了优化性能你可能不会使用 JTA 事务来执行大部分代码, 而是只对一些必要的功能使用. 例如进行信用卡处理时, 你可能不会对数据装载, 效验输入值, 验证用户的操作使用事务. 而对于转账业务时,你将使用事务来处理. 完成之后终止该事务. 并且接下来的处理过程中也不使用事务. 这是一个使用局部化JTA事务的例子. 对声明式事务而

言,由于不知晓事务何时开始和终止的原因,想要达到这个功能是非常困难的.

还有一个适用场景为使用长时间运行的 JTA 事务. 在 EJB 中这是在使用有状态会话 Bean 时出现的. 有时你可能想要一个事务跨多个服务器请求, 在这种情况下你可以在一个有状态会话 Bean 的方法中开始事务并可以在其他方法中终止该事务. 虽然这是可以完成的, 但这并不是一种好的设计. 因为数据库或 JMS 资源可能因为长时间的等待而变得非常的匮乏. 与局部化 JTA 事务中的原因一样,声明式事务是不可能完成这个功能的.

编程式事务模型只应在必要的情况下才使用. 上述的三个场景(客户端初始事务,局部化 JTA 事务,长时间运行的事务)属于这种情况.而其他情况下你应该使用声明式事务模型.

四.声明式事务模型

在使用编程式事务模型时,开发者必须显式的使用 begin(),commit()和 rollback()方法来开始,提交和回滚一个事务.而对于声明式事务模型而言,容器对事务进行了管理. 这意味开发者不需要再编写代码来开始或提交事务. 但是, 开发者必须告诉容器怎样去管理事务. 在 EJB 中, 这是通过设置 ejb-jar.xml 部署描述文件来完成的. 而对于 spring, 可以通过 ApplicationContext.xml 这个 Bean 配置文件来做到.有时 EJB 中的声明式事务模型也指容器管理的事务(CMT),但是大多数开发者更喜欢使用声明式事务这个术语.

从 Java 代码的角度来说, 我们在 EJB 中必须担心的只有处理 setRollbackOnly()方法. 这个方法告诉容器, 无论后续的处理结果如何, 当前事务只能进行回滚操作.下面的列表显示了一个在 EJB 中使用 setRollbackOnly()方法的例子.注意只有粗体标注的一行参与了事务管理.

```
@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;
    }
}
```

使用 Spring 时你不需要显式地调用 setRollbackOnly()方法，作为替代的,你可以在 TransactionAttributeSource 拦截器中说明规则来控制何时进行事务回滚.我们将在本节的后面部分显示调用示例.

当使用 EJB 时,我们通过配置 ejb-jar.xml 中的<transaction-type>节点来告诉容器我们想使用声明式事务.大多数应用程序服务器都默认设置为声明式事务管理.

在 EJB 3.0 中，你也可以使用@TransactionManagement 这个元信息注解来向容器说明使用声明式事务管理.如下所示:

```
@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
public class TradingServiceBean implements TradeService
{
    //...
}
```

在 Spring 中我们可以通过使用 TransactionProxyFactoryBean 代理类来说明我们想使用声明式事务.如下所示:

```
<bean id="tradingServiceTarget"
    calass="com.trading.server.TradingServiceBean">
</bean>
<bean id="tradingService"
     class="org.springframework.transaction.interceptor.Transa
ctionProxyFactoryBean">
```

```
    <property name="transactionManager" ref=”txnMgr” />
    <property name="target" ref=”tradingServiceTarget” />
    <property name="transactionAttributes">
       <prop key=”*”>PROPAGATION_SUPPORTS</prop>
       <prop key=”update*”>
          PROPAGATION_REQUIRED,-Exception
       </prop>
    </property >
</bean>
```

正如你在上面例子中看到的一样，在 Spring 中使用声明式事务时你需要使用代理来包装事务化 Bean. 上面例子中使用了-Exception 来告诉 Spring 在遇到任何异常后都进行回滚，这使得我们自己不需要在代码中编写 setRollbackOnly()方法. 为了简单起见我们在这儿只是说明了 Exception，但正常情况下你应该说明一个具体的应用异常(checked)类型.

1) 事务属性

当使用声明式事务的时候，我们必须告诉容器怎样来管理事务.例如，容器应该何时开始一个事务？那些方法需要一个事务？容器在已存在事务的情况下是否应该开始一个事务?. 此时可以通过事务属性来告诉容器怎样管理 JTA 事务. 对于 EJB 可以在 ejb-jar.xml 中说明，而 Spring 可以在 TransactionAttributeSource 中配置. 下面是 6 种事务属性的设置值:

- Required
- Mandatory
- RequresNew
- Supports
- NotSupported
- Never

Spring 自己增加了一个额外的 PROPAGATION_NESTED 属性. 该属性告诉 Spring 对事务进行嵌套处理并对嵌套的事务使用 Required 属性. 当然的, 在使用该设置时底层的事务服务实现者必须支持嵌套事务才行.虽然上面列出的事务属性可以在类级别进行说明,但事务属性本身总是关联到类中的方法上的. 当一个事务属性说明在类级别时, 该类中所有的方法都会采用该属性来作为默认值. 但具体方法所设置的属性值可以覆盖类级别的默认值(我们将在本章的后面看到).

REQUIRED

Required 属性(在 Spring 中为 PROPAGATION_REQUIRED)告诉容器指定的方法需要一个事务来执行.如果当前已经存在一个事务环境, 将使用当前的事务. 否则将开始一个新的事务.这个属性是最常见并被大多数开发者用来管理事务的属性 但是,正如下面将会讲到的, 有时更需要使用 Mandatory 来替代该属性.

MANDATORY

Mandatory 属性(在 Spring 中为 PROPAGATION_MANDATORY)告诉容器指定的方法需要一个事务来执行.但不像 Required 属性那样,该属性将不会开始一个新的事务. 当使用这个事务属性时,在方法调用之前必须存在一个先前的事务环境.如果方法调用时不存在事务环境,容器将会抛出一个 TransactionRequiredException. 该异常指示此时需要事务但并没有事务存在.

REQUIRESNEW

RequiresNew 属性(在 Spring 中 PROPAGATION_REQUIRES_NEW)告诉容器指定的方法应该总是开始一个新事务来执行.如果之前已经存在一个事务, 该事务将会被挂起, 并生产一个新的事务来执行.当新事务终止执行后, 原来挂起的事务将会恢复.在使用

RequiresNew 属性时,如果之前已经建立了事务环境, 该操作将会违犯 ACID 性质.这是因为先前的事务是被挂起的, 一直到新事务执行完成时才恢复.

该属性对于必须将所处事务独立进行提交的操作而言是非常有用的.一个使用该属性的例子为审计日志的处理. 例如,在大多数交易系统中无论操作成功或失败,每个活动环节都必须被记录. 假设一个场景中交易者正试图购买特殊的股票. 一个新的 JTA 事务将会在 placement()方法调用前产生. 然后 placement()方法又调用了另一个 EJB 中公共的 aduit()方法, 该方法向数据库中插入操作详情. 此时因为 aduit()方法与 placement()方法是处于相同事务中的,如果 placement()方法中产生了异常, 将会导致 audit()方法中的插入操作被回滚. 这将违反上面的需求中无论成功或失败必须进行日志记录这一规则.因此, 使用 RequresNew 属性为 audit()方法分配一个新的事务将能够保证该方法中的插入操作能够被提交. 此时无论外面的 placement()方法是否产生异常, 都不会影响日志的记录.

SUPPORTS

Supports 属性(在 Spring 中为 PROPAGATION_SUPPORTS)告诉容器指定的方法不需要使用事务来执行,但如果已经存在一个事务,则会使用该事务. Supports 属性在事务管理中是非常有用的. 假设有一个对数据库的简单查询来获取一个交易者的所有交易额. 使用事务来执行该操作是不必要的, 所以使用 Supports 来告诉容器不需要开始一个新事务来执行方法调用. 但是, 如果查询是包含在带有更新语句的事务中.使用 Supports 将会让容器使用事务环境来查找数据库记录. 因而查询的结果包含了此次更新操作所改动的数据(因为它们处于同一事务中.如果是不同事务,因为事务隔离的关系如果该事务不提交,单独的查询是不能获取所更新的数据的).

为了例证 Supports 属性的使用, 我们假设一个特殊的交易者每

天的交易额不多于 1 百万股. 当使用 Supports 属性时将会产生下面的执行步骤:

- 第一步: 目前总股份额 = 900,000
- 第二步: 事务开始
- 第三步: 交易者进行 200,000 股份的交易
- 使用 Supports 属性来调用查询方法,得到总交易额(**1,111,000**)
- 异常抛出,总交易额超出限制.事务回滚(交易无效).

而当我们使用 NotSupported(见下一节)属性时,上述操作将不会产生异常. 因为此时查询是在事务范围之外的.因此将不会看到更新的结果:

- 第一步: 目前总股份额 = 900,000
- 第二步: 事务开始
- 第三步: 交易者进行 200,000 股的交易
- 使用 NotSupports 属性来调用查询方法,得到总交易额(**900 ,000**)
- 交易被允许,事务提交.

NOTSUPPORTED

NotSupported 属性(在 Spring 中为 PROPAGATION_NOT_SUPPORTED)告诉容器指定的方法不需要使用事务来执行. 如果已经存在一个事务, 该事务将会被挂起直到指定的方法执行完成. 而没有事务存在时, 容器只是简单地调用指定的方法.该属性在可能会在事务环境中产生异常的操作而言是很有用的. 例如,在 XA 事务环境中执行一段包含 DDL 代码的存储过程时,如果存储过程不被底层实现所支持的话, 将会产生异常. 在你不确定是否存储过程被支持时, 你可以使用 NotSupported 属

性来在方法调用前挂起之前的事务.

NEVER

Never 属性(在 Spring 中为 PROPAGATION_NEVER)告诉容器指定的方法不能够在事务环境中执行. 请注意该属性与 NotSupported 属性之间的不同,对于 NotSupported 属性, 如果有一个之前存在的事务,在指定方法调用之前该事务将会被挂起.但对于 Never 属性, 如果有一个之前存在的事务,容器将会抛出一个异常来指示事务是不被允许的. 使用 Never 属性将会导致不期望的运行时异常. 所以该属性只应该在绝对必要的时候使用. 而这种情况并不多见. 当你不确定是否该使用时, 请使用 NotSupported 属性来替代.

2) 指定事务属性

事务属性总是关联到一个具体的方法上.当为整个 bean 定义事务属性时, 该 Bean 中的所有方法都使用此属性作为默认属性. 对于 EJB 而言, 事务属性可以在 ejb-jar.xml 部署描述文件中进行说明. 如下所示:

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

请注意上面示例中TradingService的所有方法都被分配了Mandatory属性(通过使用通配符<method-name>*</method-name>).这种方式通常是被认为一种不好的实践. 因为这样做时查询方法将不会得到优化.例如, 有一个getAllTradersByID()的查询方法, 这个方法实际上是不需要事务环境的.但上面的配置中仍会为该方法使用事

务.

如果使用EJB 3.0，你可以使用元信息注解来说明事务属性. 如下面的代码所示:

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

为了简化起见，我在后面的代码示例中都将使用注解风格来说明事务属性. 如果你使用的是EJB 2.x，你可以很简单地将元信息注解替换为前面示例中的XML配置.

对于Spring而言,事务属性可以通过TransactionAttributeSource来说明或者在TransactionAttributeSource中通过属性对来配置. 下面的例子中显示了这2种方式的用法:

1. 使用TransactionAttributeSource

```
<bean id="transactionAttributeSource"
   calass="org.springframework.transaction.interceptor.NameM
atchTransactionAttributeSource">
  <property name="properties">
    <props>
      <prop key="*">PROPAGATION_MANDATORY</prop>
    </props>
  </property>
</bean>
<bean id="tradingService"
    class="org.springframework.transaction.interceptor.Trans
actionProxyFactoryBean">
   <property name="transactionAttributeSource" >
     <ref local=”transactionAttributeSource” />
   </property>
    ...
```

```
</bean>
```

2. 使用TransactionProxyFactoryBean

```
<bean id="tradingServiceTarget"
    calass="com.trading.server.TradingServiceBean">
</bean>
<bean id="tradingService"
     class="org.springframework.transaction.interceptor.Trans
actionProxyFactoryBean">
    <property name="transactionManager" ref=”txnMgr” />
    <property name="target" ref=”tradingServiceTarget” />
    <property name="transactionAttributes">
       <prop key=”*”>PROPAGATION_MANDATORY</prop>
    </property>
</bean>
```

在上面的例子中，第二种方法实际上是对第一种配置的简化.它避免了必须将TransactionAttributeSource定义为一个独立的Bean.当使用第二种配置时, Spring在背后创建了一个NameMatchTransactionAttributeSource对象来进行配置.

还有一种与配置事务属性有关的技术是将属性配置在方法级别而非类级别. 但是，这并不是说需要显式的对每个方法都说明属性.一种更好的方法是在类级别分配一个默认的事务属性,并在需要的时候对具体方法进行优化. 下面是这种方式的最佳实践:

```
最佳实践:
  当为方法分配事务属性的时候,把类中对大部分方法最具限制性的属性作为类级别的默认属性，然后再对有特殊需要的方法进行微调.
```

我们可以在前面的XML部署描述例子中应用该最佳实践,下面将Mandatory作为默认属性，而对查询方法使用Supports属性:

```
<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>
</assembly-descriptor>
```

在Spring中，我们可以如下方法来应用最佳实践:

```
<bean id="tradingServiceTarget"
    calass="com.trading.server.TradingServiceBean">
</bean>
<bean id="tradingService"
     class="org.springframework.transaction.interceptor.Trans
actionProxyFactoryBean">
    <property name="transactionManager" ref=”txnMgr” />
    <property name="target" ref=”tradingServiceTarget” />
    <property name="transactionAttributes">
      <prop key=”*”>PROPAGATION_MANDATORY</prop>
      <prop key=”getAllTradersByID”>
        PROPAGATION_SUPPORTS
      </prop>
    </property>
</bean>
```

最后对于EJB 3.0中使用元信息注解来简化事务配置这种方式而言，因为事务属性可以在类级别以及方法级别说明，我们也可以通过下面的方式来在EJB 3.0中使用最佳实践:

```
@Stateless
@TransactionAttribute(TransactionAttributeType.MANDATORY)
public class TradingServiceBean implements TradeService
```

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

在类级别使用对大部分方法最具限制性的属性来作为默认值将总是安全的.因为我们能够保证开发者忘记为新方法分配属性时仍有一个有效的事务环境. 例如, 如果我们将上面例子中的属性值反转, 此时类级别的默认值为Supports. 如果一个添加了一个新的更新方法而没有对其说明事务属性的话, 该方法将不会处于事务环境之内. 而且这种情况通常是很难被发现的.

3) 异常处理与 setRollback() 方法

虽然使用声明式事务后容器或者框架管理了事务,但在 EJB 中我们仍需要进行小量 Java 编码来使声明式事务正常工作.请考虑下面例子中进行一笔固定收益的交易.这是一种像证券或国债的安全投资.

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

在此处代码中我们有多个更新操作存在于单个业务工作单元中. 如果这个方法执行到一半就抛出一个 TradeExecutionException 的话, 容器将会提交异常发生前的操作并将异常传播给调用者. 这个例子想要证明的是, **容器并不会对一个应用(checked)异常进行事务回滚.**

当使用声明式事务时忘记处理应用异常是大多数程序中导致数据一致性问题的常见原因.

为了修正该问题, 我们必须通知容器在出现某种应用(checked)异常时进行事务回滚.在 EJB 中, 可以通过使用 EJBContext 接口的 setRollback()方法来达到该目的. 而在 Spring 中, 这是通过在 TransactionAttributeSource 这个 Bean 的定义中声明事务回滚规则来完成的. 需要注意的是使用该方法时并不会直接回滚事务, 它只是通知容器当前的事务只可能进行回滚操作. 当该方法被使用来标记当前事务为 STATUS_MARKED_ROLLBACK 状态后,当前事务的状态将不能在改变. 下面的示例中显示了修正后的代码:

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

此时添加 setRollbackOnly()方法后, 当应用异常抛出时容器将会正确地回滚所有先前的数据库更新操作.

虽然我们必须在我们的程序中添加一些额外的代码, 但这是很必要的. 重要的是容器不会自动地对一个应用异常进行事务回滚. 例如,假设在我们的处理中有一部分是需要发送确认邮件, 如果 SMTP 服务器不可用的话, SMTP 服务将会抛出一个应用(checked)异常.在大多数情况下我们并不想容器因此而回滚整个事务.此时应该捕获这个应用异常并进行相应的事务控制.

当为你的企业 Java 应用设计总体的事务策略的时候,你必须做出的一个决定是何时及何地去调用 setRollbackOnly()方法. 对此的最佳实践如下:

```
最佳实践:
  事务管理应该被包含在开始该事务的方法中. 因此,应该只让那些开启了新事务的方法去调用 setRollbackOnly().
```

这个实践特别适用于开始新事务的方法使用了其他 Bean 或方法的情况.此时调用方法的事务环境对于其他被调用者是有效的.在事务化的 Bean 或者包含多层事务化 Bean 的应用(例如会话门面(façade)层与持久化管理层)之间的服务进行内部通讯时将会产生上述情况. 这条最佳实践的合理性可以体现在两方面: 第一, 基于一种简单的组件责任模型,事务管理应该被包含在开始该事务的方法中.如果将事务管理分散到整个应用程序中, 则会增加管理的复杂度,并且从事务管理的观点来说这也降低了应用程序的可维护性. 第二,只要 setRollbackOnly()方法被调用,是没有其他方法来取消这次回滚操作的.这意味着开始事务的方法(因此该方法也管理着事务)在其他方法回滚事务后是没有本办法来矫正错误并使处理继续进行的.

该最佳实践的一个推论为那些不负责进行事务回滚的方法应该在它们所抛出的异常中提供足够的信息. 这样开始该事务的方法可以根据这些信息来决定是否应该调用 setRollbackOnly()方法.一个好的例子为当因为 SMTP 服务器故障而无法发送确认邮件时, 将抛出一个检查异常来通知调用者.这种情况下开始事务的方法可以发送延时邮件并将事务进行提交. 对于因电子邮件服务器故障而回滚一笔订单是非常不可取的方法.

4) EJB 3.0 所需考虑的事项

当前的 EJB 3.0 规范中包含了一个@ApplicationException 的元信息注解.这个注解可以通知容器是否在其所标识的应用异常抛出后自动地回滚事务. 该注解接受一个 boolean 值作为参数, 如下所示:

```
@ApplicationException(rollback=true)
```

```
public class MySeriousApplicationException extends Exception
{
}
```

当 MySeriousApplicationException 异常抛出时,容器将自动地标记当前事务为回滚状态，这避免了开发者手动调用 setRollbackOnly()方法来进行回滚．使用该注解的好处在于避免了手动地在应用中编写 setRollbackOnly()方法．此外，该方式在处于没有的事务环境时,并不会抛出一个 IllegalStateException，而只是简单地忽略事务回滚请求．对于 setRollbackOnly()方法，在没有事务环境时对其进行调用将抛出 IllegalStateException 异常．

虽然这个注解试图模仿 Spring 的规则式回滚处理，但使用该注解也可能对你的整体事务设计策略产生严重的影响．因为该注解是应用于异常类上,而任何方法都有可能抛出该异常．对于其他对象，无论是否开始了事务，抛出该异常也会导致当前的事务被标记为回滚状态.这无疑与我们的最佳实践中只有开始事务的方法来负责管理事务这一原则背道而驰.如果使用该注解的话，负责管理事务的方法是没有合适的方式来进行管理的．例如，如果一个 SMTP 邮件服务方法抛出了一个使用了该注解的异常，此时仅仅因为不能发送验证邮件，我们的整个事务都会被回滚.此外，当该注解与事务回滚属性混合使用来进行事务管理时将会可能会出现其他问题.

5) 可替代 setRollbackOnly() 的方法

在 EJB 中，一种可以替代调用 serRollbackOnly()来回滚事务的方法是抛出一个 javax.ejb.EJBException 异常，这个系统异常将会强制对事务进行回滚．这种常见的技术也有它自己的利弊．下面的代码显示了这种技术的使用:

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
Public void placeFixedIncomeTrade(TradeData trade){
   try{
      // ...
      Placement placement =
```

```
            placementService.placeTrade(trade);
        executionService.executeTrade(placement);
    }catch(TradeExecutionException e){
      log.fatal(e);
      throw new EJBException(e.getMessage());
    }
}
```

因为容器会自动地为运行时异常进行事务回滚且 EJBException 本身就是一个运行时异常. 上面的代码将会产生与使用 setRollbackOnly()方法相同的效果.

该技术的优势在于它避免了在使用 setRollbackOnly()方法时可能抛出的运行时异常.如果我们将上面代码中部署描述文件的事务属性改为 Supports, 容器将会抛出一个 IllegalStateException. EJB 2.1 规范中对此的声明如下:

> ”如果 ejbContext.setRollbackOnly() 被业务逻辑方法调用时使用的是 Supports, NotSupported 或 Never 事务属性,容器必须抛出一个 java.lang.IllegalStateException.”

此时使用 EJBException 将会避免这种运行时异常.但这并不是一种很合适的使用方法. 因为这将在我们的事务设计策略中隐藏潜在的问题.

这种技术的一个缺点在于 EJBException 异常产生的输出信息将是模糊不清的. 下面的输出是一个应用中在产生重复的交易订单时使用 setRollbackOnly()方法产生的信息:

```
WSCL0014I: invoking the Application Client class com.trading.client.ClientApp
starting Trading Client
com.trading .common.TradingException:    Duplicate Order
```

相同的代码, 在使用 EJBException 时的输出如下:

```
WSCL0014I: invoking the Application Client calss com.trading.client.ClientApp
starting Trading Client
javax.transaction.TransactionRollbackException:
CORBA TRANSACTION_ROLLBACK 0X0 Yes; Nested exception is:
   org.omg.CORBA.TRANSACTION_ROLLBACK:
```

虽然两种技术都产生相同的结果(他们都使事务进行了回滚),但EJBException 方式将会丢失调用者构造的异常描述信息.

EJBException 技术的另一个缺点是缺乏对处理应用异常与进行事务管理之间概念上的划分.开发者查看代码时可能不会意识到抛出EJBException 的目的是对当前事务进行回滚. 移除或修改该逻辑的代码可能会使其不像原始意愿那样回滚当前事务.且可能使当前事务管理策略无效.

出于这些理由, 我更推荐使用 setRollbackOnly()方法并避免使用EJBException 异常来作为一种事务管理的手段.

6) 比较 使用@Required 与 命令式(Mandatory) 事务属性

对于在决定是使用 Required 还是 Mandatory 事务属性时,一些人常常会变得迷惑. 虽然二者都提供了事务环境, 但是 Required 属性在不存在事务时将开始一个新的事务, 而 Mandatory 属性则不会.无论你使用的是那种事务设计策略, 下面的最佳原则都有助于你理解何时对二者进行选择.

最佳实践:
如果一个方法需要一个事务环境但不负责对事务进行回滚的话,该方法应该使用 Mandatory 事务属性.

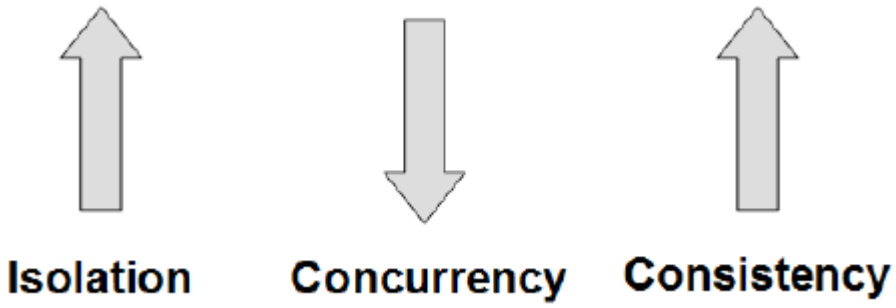
这条最佳实践的合理性是基于事务所有关系的. 除了有状态会话 Bean 外,事务在哪个方法开启的也必须在哪个方法结束.事务的管理(包括何时标记事务为回滚状态)应该总是在所有者方法中进行. 使用 Required 属性将出现一个潜在的问题,因为事务可能被某个方法开启,但并没有在抛出应用异常时将整个事务回滚(回滚的只是它自己开启的事务).

这种境况最常发生在客户端初始化的事务中,如果客户端开始了一个事务并调用了一个远程无状态会话 Bean(Stateless SessionBean,SLSB), 由于该事务是被客户端开启的,远程 SLSB 不应该负责对事务进行回滚. 因此该 SLSB 组件应该使用 Mandatory 事务属

性.而如果事务属性被设置为 Required. 将可能出现事务被客户端方法开启后永远不会得到回滚的情况.

7) 现实中的事务隔离级别

对开发者而言, 还有一个可用的事务设置是控制事务的隔离级别. 事务隔离级别是指事务之间的交互隔离度.在多个事务访问和更新相同数据时, 隔离级别决定了事务的更新是否可见.数据库系统,EJB 以及 Spring 都允许用户设置事务的隔离级别.但是,这个设置是依赖于具体的应用程序服务器和数据库的. 应用程序服务器可以支持多种隔离级别, 但数据库也必须同时支持这些隔离级别才有效.事务隔离是数据库并发性与一致性之间的杠杆.在提高事务隔离级别的同时, 我们也提升了一致性, 但降低了数据库的并发性能. 下面的图解释了这一关系:



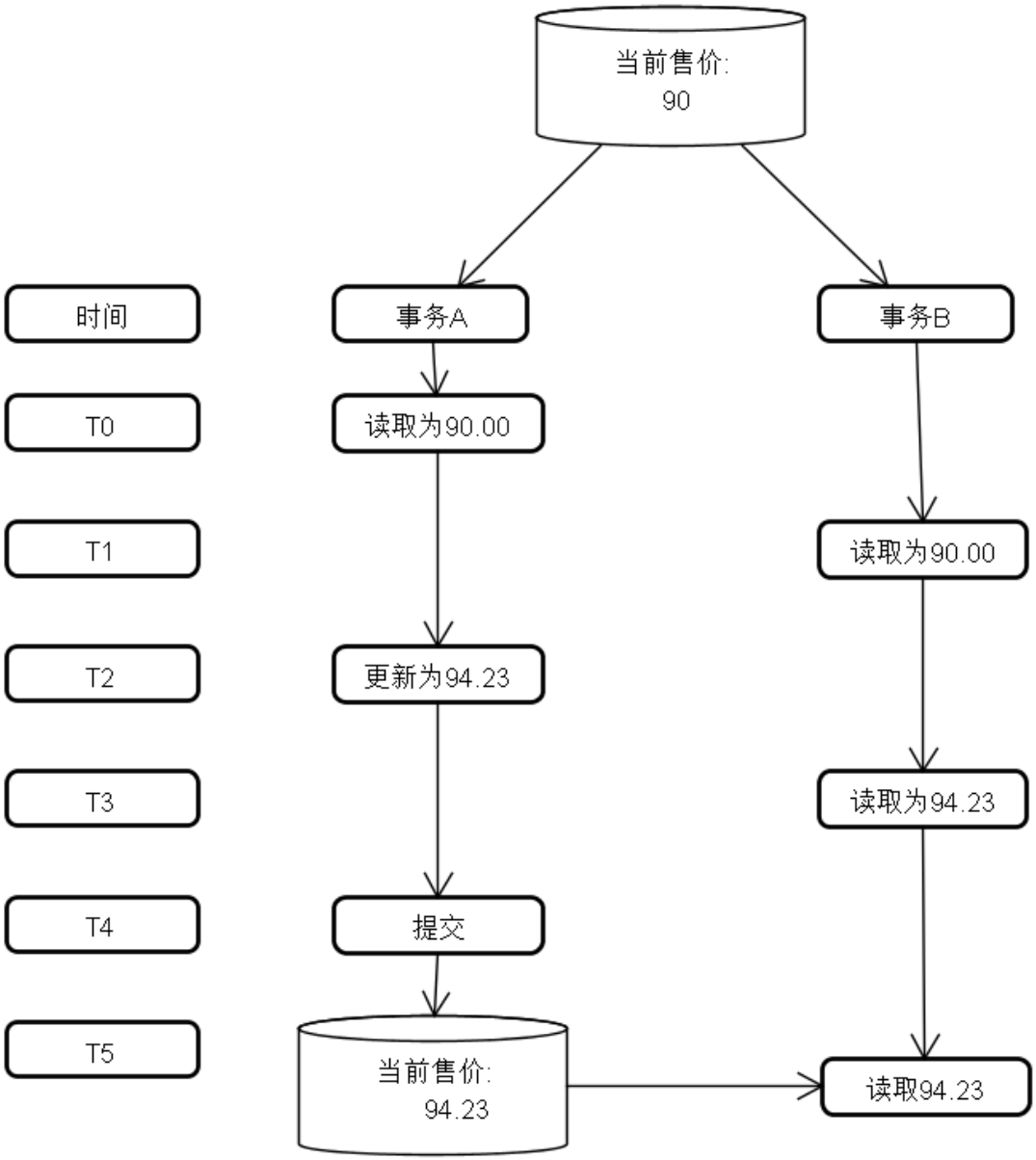
该设置能够影响一个应用程序的性能与数据一致性. 例如,对于像信用卡处理这种高性能的应用,你可以通过降低隔离性(但影响数据一致性)来增加并发性能. 对于像金融处理这种对并发要求较低, 而对数据一致性有着高要求的应用则可以增加隔离级别. 大多数应用程序服务器和数据库系统都有将并发与一致性之间的均衡值作为默认设置. 但在 EJB 或 Spring 中, 你可以在需要的情况下修改这个默认值来优化你的应用程序.EJB 与 Spring 均支持主要的事务隔离级别. 下面按照隔离度由低到高来列出这些设置:

- TransactionReadUncommitted

- TransactionReadCommitted
- TransactionRepeatableRead
- TransactionSerializable

Transaction-ReadUnCommitted

这个设置是隔离级别中的最低值. EJB 和 Spring 均支持该设置. 该隔离级别允许事务读取到其他事务未提交到数据库中的数据. 为了证明这个隔离级别是怎样工作的, 可以假设我们有一个特殊的股票目前交易价格为每股 90.00 .现有两个事务(事务 A 与事务 B)在相同时间访问了相同的数据. 此时事务 A 执行更新操作, 而事务 B 执行读取操作. 下面的示例图解释了在 TransactionReadUnCommitted 设置下事务的交互过程:

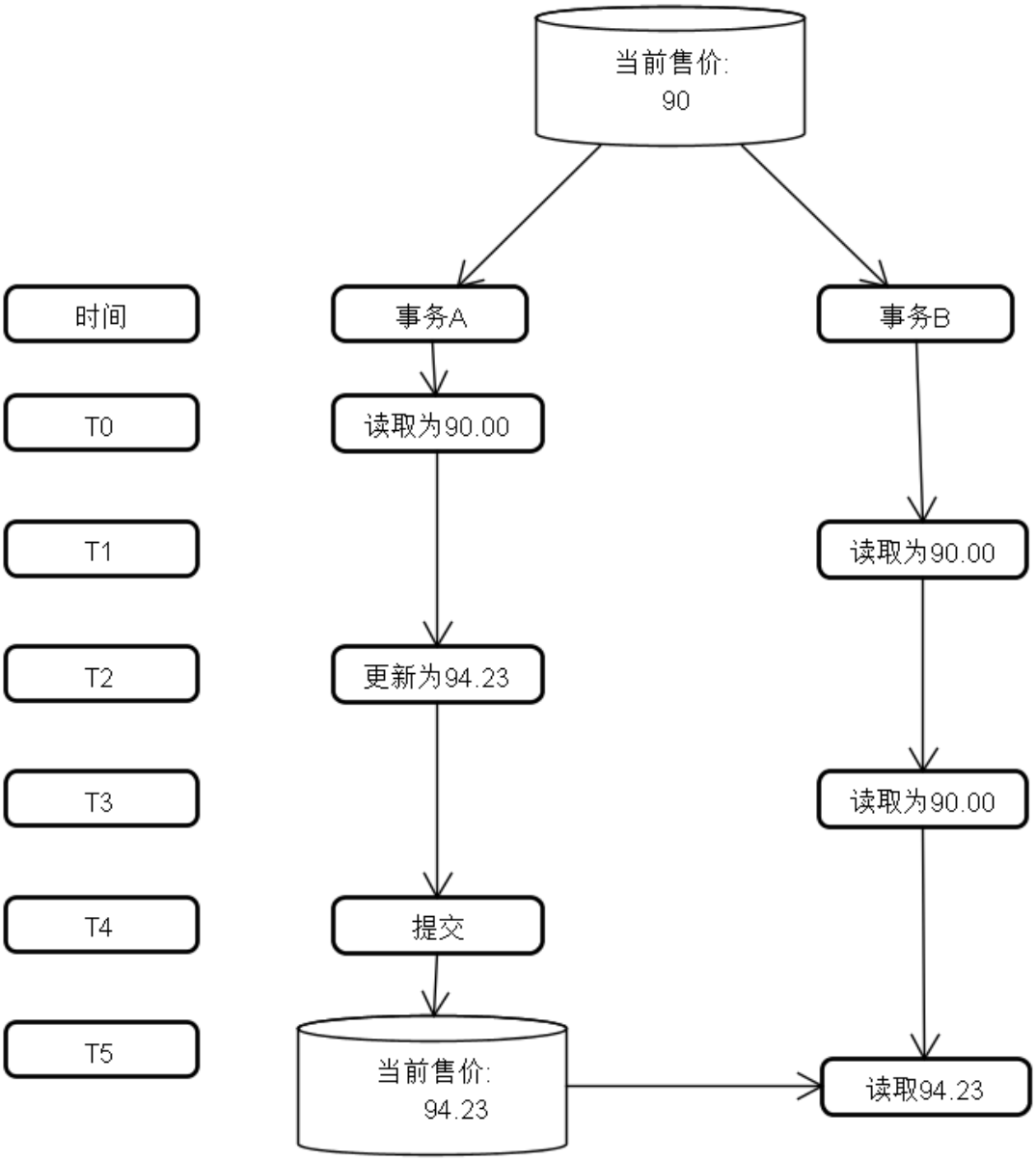


注意上面图中在 t2 时刻，事务 A 对数据库进行了跟新，但事务 B 在却在 T3 时刻读取到了事务 A 未提交数据.正如图中所示，事务 A 的更新并没有与事务 B 相隔离．如果事务 A 回滚了此次更新操作，事务 B 所读取到的数据将会是无效的(脏读).这个事务隔离级别违犯了基本的 ACID 属性，并且大多数的数据库厂商都不支持该级别(包括 Oracle).

Transaction-ReadCommitted

该隔离级别允许在多个事务访问相同的数据时,隐藏其他事务

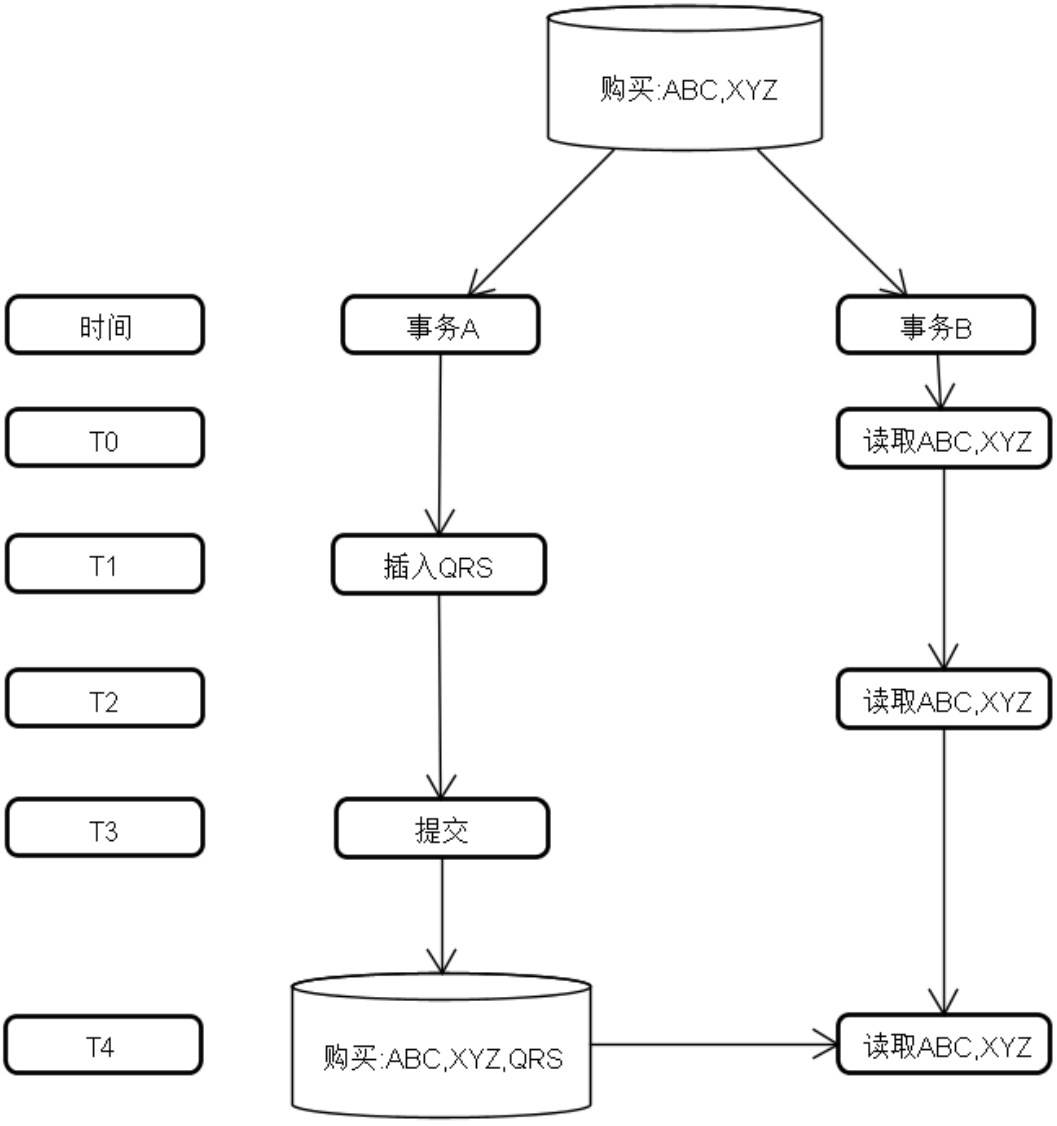
未提交的数据. 使用与前面相同的例子,下面的图示解释了在 TransactionReadCommitted 设置下事务的交互过程



注意当 T2 时刻事务 A 更新了数据后, 事务 B 在 t3 时刻并没有读取到未提交的数据. 该级别是一个很好的隔离级别, 它允许事务 B 并发地访问数据且隐藏掉了其他事务中未提交的更新. 因此这是大部分数据库的默认隔离级别, 并且所有的数据库厂商都支持该级别.

Transaction-RepeatableRead

与事务的交互不同，这个隔离级别将一个事务与另一个完全地隔离开来.该级别保证在事务中所读取到的数据集都是相同的(可重复读).在当前事务未提交之前，即使其他事务所作的更新已经提交，当前查询出来的结果集也与之前所查询的结果集是相同的(除非事务在保持读写锁的情况下修改数据).为了证明该隔离级别，假设我们有一个 Select 语句来查询到**目前为止**所有被购买的股票. 事务 A 在事务 B 查询的时候插入了一个新的购买者. 下面的图示解释了在 TransactionRepeatableRead 设置下事务的交互过程

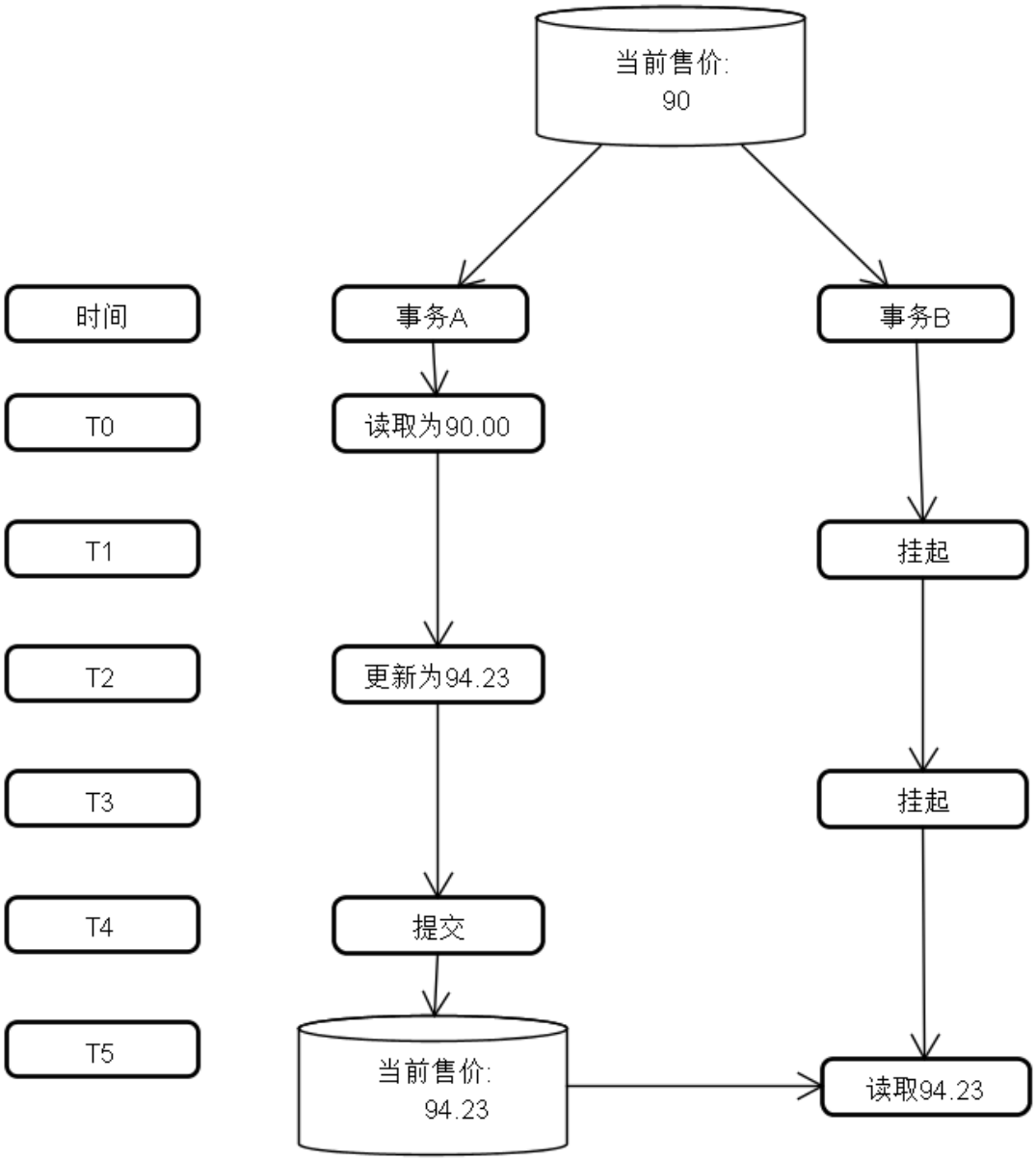


注意上面的例子中虽然事务 A 在事务 B 查询的时候购买了新的股票(QRS),但事务 B 所查询的结果仍然是相同的.只有在事务 B 提交结束后它才可以看到被更新的数据.值得注意的是在使用该隔

离级别时数据的查询与修改都会被加上读写锁.所以应该谨慎地使用该隔离级别, 因为在一个使用可重复读的事务查询数据时,其他事务是不能修改数据的. 修改操作会一直处于等待状态(或者失败)直到该可重复读的事务进行提交.

Transaction-Serializable

这是 Java 所支持的隔离级别中最高的设置, 当使用这个设置时, 事务之间的交互将被 ”层叠” 起来, 这样某一时刻只有唯一一个事务被允许执行(但对于 Oracle 而言, 这并不是绝对的).在这种情况下将会对并发性能造成严重的影响, 不过这也使数据一致性得到了最大化的上升. 使用与上面股票售价相同的例子, 下面的图示解释了在 TransactionSerializable 设置下事务的交互过程



注意在事务 A 完成前事务 B 将会一直处于挂起状态. 虽然所有的数据库产商都支持该设置, 但对于 Oracle 而言, 这还是有一点不同的. 因为 Oracle 使用数据版本机制,它将不会真正的挂起事务 B.但是,如果事务 B 在事务 A 执行时尝试访问数据, Oracle 将会返回一个 ORA-08177 错误消息, 这个消息指示不能对该事务进行序列化访问.

现实中的隔离设置

在 EJB 中事务隔离级别是应用服务器所作的扩展,因此我们必

须在具体的应用服务器的部署描述文件中来对事务隔离级别进行说明.对于 Spring 而言，事务隔离级别可以和事务属性一起设置.下面的例子显示了在 WebLogic 中事务隔离级别的设置(weblogic-ejb-jar.xml).

```
<transaction-isolation>
   <isolation-level>
     TransactionSerializable
   </isolation-level>
   <method>
        <ejb-name>TradingService</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>getAllTradersByID</method-name>
  </method>
</transaction-isolation>
```

下面的示例显示了在 Spring 中相等效设置:

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

EJB 与 Spring 中对事务隔离级别的支持是依赖于底层的数据库系统的.因此，虽然二者都支持上面所述的 4 种隔离设置，但数据库也必须同时支持才行.如果数据库不支持框架中说明的设置值，数据库将会使用默认值来替代. 例如，在使用 Oracle 数据库时用 TransactionRepeatableRead 来设置，Oracle 将拒绝该设置并使用 TransactionReadCommitted 来替代. 不幸的是,这种情况并不会显式的抛出异常. 所以你必须考虑你所使用的设置是否被实际的数据库所支持，避免你指定了所选设置但数据库却使用了默认设置来替代.此外，还需要注意在你更改事务隔离级别时,你的代码可

能是不会在多种数据库平台上交叉运行的.因为不同的数据库厂商对隔离级别的支持是不一样的.

大多数情况下,最好使用 TransactionReadCommitted 来进行设置. 如果有必要进行更改的话,请确定你所用的数据库支持你在框架中所作的设置.隔离级别的设置将会在你的事务设计策略中扮演很重要的部分, 对它的使用应该持谨慎态度.

五.XA(X/Open 提出的分布式事务接口)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何时去使用它,请考虑下面的 EJB 代码示例. 该示例与前面章节中出现的相同:

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

该代码首先预定一笔交易,接着在执行该交易. 二者在数据库中更新不同的表. 正如我们在前面章节中看到的, 这段代码可以在非 XA 环境中保持 ACID 属性.假设我们出现一个新的需求, 此时要发送一个 JMS 消息给公司中的另一个系统去处理所有的固定收入交易.为了简单起见我们假设所有的 JMS 处理逻辑都在 sendPlacementMessage()这个方法中, 该代码修改后增加了对新方法的调用, 如下:

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
Public void placeFixedIncomeTrade(TradeData trade){
   try{
```

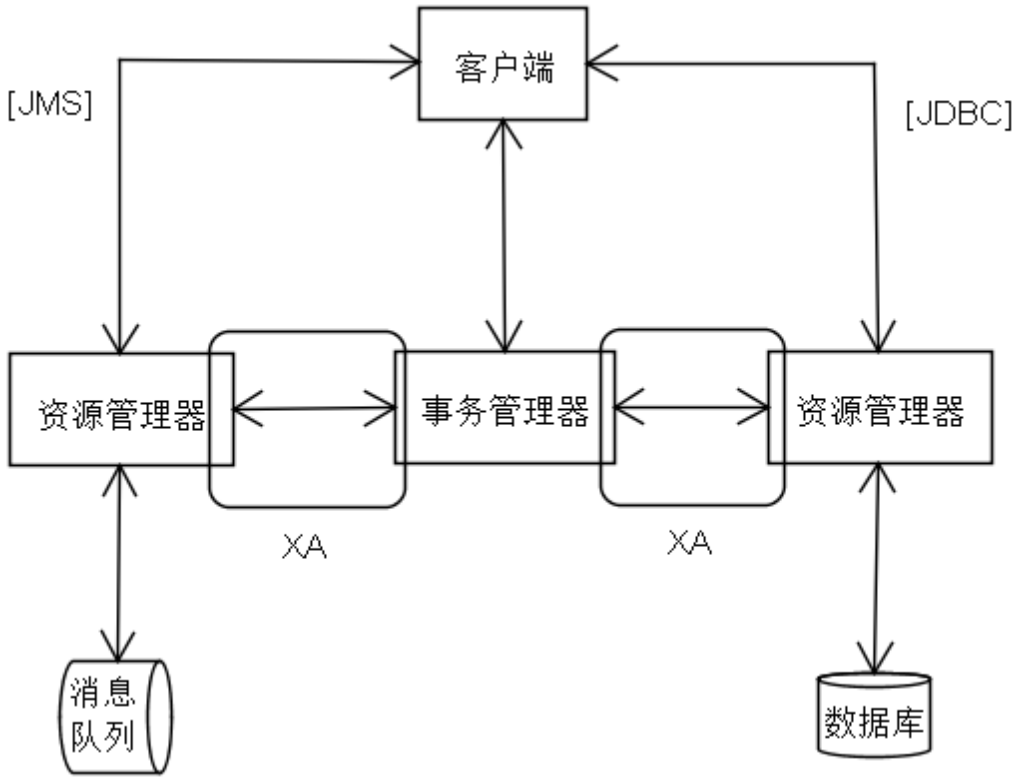
```
        // ...
        Placement placement =
                placementService.placeTrade(trade);
        placementService.sendPlacementMessage(placement);
        executionService.executeTrade(placement);
    }catch(TradeExecutionException e){
      log.fatal(e);
      throw e;
    }
}
```

虽然这个变更似乎显得很不实际，但这个代码将不再保持 ACID 属性.如果 executeTrade()方法抛出一个 TradeExecutionException,数据库中的更新将会被回滚，但交易预定消息已经发送给 JMS 队列(或主题)了.实际上,交易预定的消息很可能在 sendPlacementMessage()方法执行结束后就已经被立即发布到 JMS 队列(或主题)上了.

因为消息队列的插入在非 XA 环境中是不依赖于数据库更新的, ACID 中的原子性和隔离性将不复存在.这样总体的数据一致性将会大打折扣. 我们所需要的是一种保持消息队列与数据库处于单个全局事务中的方法，这样两种资源将被整合为一个单一的工作单元.使用 X/Open XA 接口,我们可以协调多个资源并使其兼容 ACID 属性.

1) XA 接口解释

X/Open XA 接口是系统级别的双向接口,该接口是单个事务管理器与一个或多个资源管理器之间的沟通桥梁. 此时的事务管理器控制着 JTA 事务,管理着事务的生命周期并协调着多个资源.在 JTA 中事务管理器的抽象接口为 javax.transaction.TransactionManager.该接口被底层的事务服务(JTS)所实现. 而资源管理器负责控制和管理实际的资源(数据库或 JMS 队列). 下面的图说明了在典型的 JTA 环境中事务管理器,资源管理器与应用客户端之间的关系.



注意上面图中 XA 接口是作为事务管理器与各种资源管理器的通讯桥梁.因为 XA 接口拥有双向性，XA 接口也支持两阶段提交协议.我将在后面的章节中讨论两阶段提交.

对于XA接口,除了本章所介绍的外，还有许多其他的内容.如果读者对XA感兴趣并想深入了解的话，可以阅读X/Open 上的XA接口规范: http://www.opengroup.org/onlinepubs/009680699/toc.pdf.

2) 应该何时使用 XA?

在 Java 事务管理中,一个常见的困惑是何事应该使用 XA. 因为大多数商业应用程序服务器都会执行一阶段提交操作，此时不会很关心性能的下降. 但是，如果在你的应用中不必要的引入了 XA 数据库驱动的话，将可能出错并产生不期望的结果.在使用本地事务模型时更是如此. 因此，一般情况下如非必要就应该避免使用 XA. 下面的最佳实践描述了何时应该使用 XA:

最佳实践：

X/Open XA 接口应该只被用于你想要在相同事务环境内协调多个资源(例如数据库或 JMS 消息)的情况.

此处的重点在于虽然你的应用中可能会出现多个资源,但你应该只在需要对多个资源进行协调来使其处于相同事务范围内的时候才使用 XA. 这里的多个资源可能是指多个数据库, 一个数据库和一个消息队列或是多个消息队列的情况. 在你的应用中, 可能分别使用到了数据库和 JMS 消息, 但如果这些资源并不需要处在相同的事务之内, 你是不需要使用XA的. 在本章开头的代码中,预定一笔固定收入的交易并发送一个消息到队列中是一个需要使用 XA 来维护 ACID 属性的例子.

使用 XA 的最常见场景为在相同的事务环境内协调数据库的更新与JMS消息队列.注意这两个操作可能发生应用程序中完全不同的区域(特别是使用像 Hibernate 这样的 ORM 技术的时候). XA 事务必须协调两种不同类型的资源以便将回滚或更新操作与其他事务隔离开来. 如果没有 XA, 发送到消息队列的消息很可能在事务提交之前就已经被接受者读取了.在使用 XA 的情况下, 发送到消息队列的消息在所处的事务未被提交的情况下将不会被发布给接受者.另一个值得注意的情况是,如果你所协调的资源是一个可操作的数据库和一个只读数据库(即参考数据库)时, 你并不需要使用 XA. 但是由于 XA 会对只读资源进行优化, 在你将只读数据源放在XA事务中时应该不会出现额外的开销.

在你的企业 Java 应用中使用 XA 时仍有许多事项需要考虑.这些事项包括两阶段提交处理(2PC), 启发式异常 以及 XA 驱动的使用. 下面几个小节将分别进行详述.

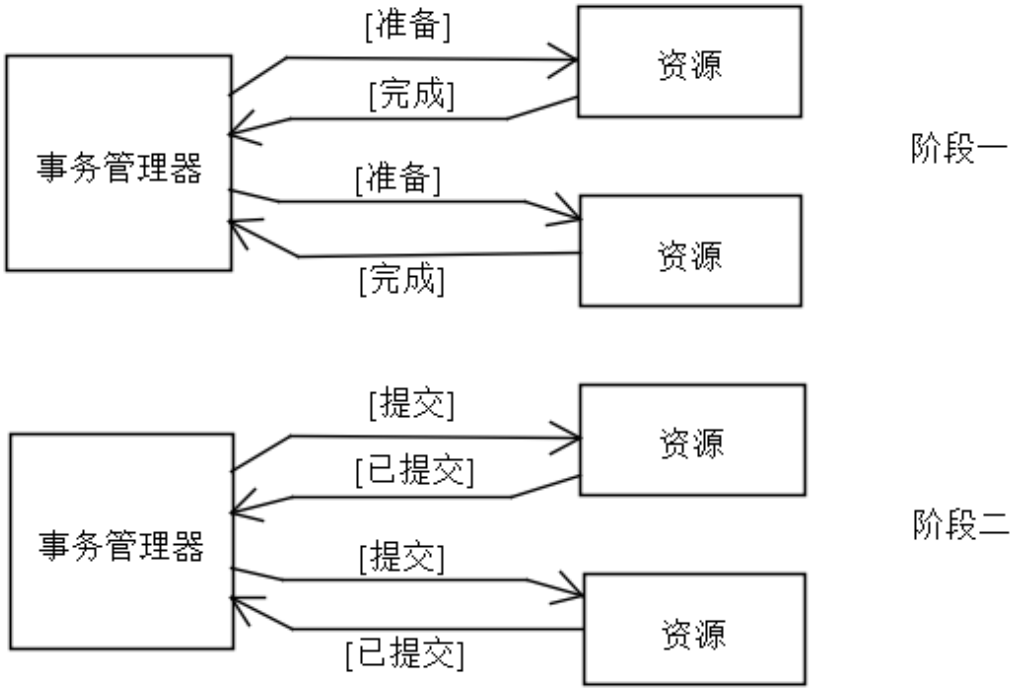
3) 两阶段(Two-Phase)提交

两阶段提交协议(2PC)是 XA 在全局事务中协调多个资源时所采用的架构.虽然两阶段提交协议比 OSI/DTP(开放式系统互联)标准预先存在了好几年, 但是它是符合OSI/DTP标准的. 两阶段提交协议由两部分组成: 准备阶段(阶段一) 与 提交阶段(阶段二). 一个对两阶段

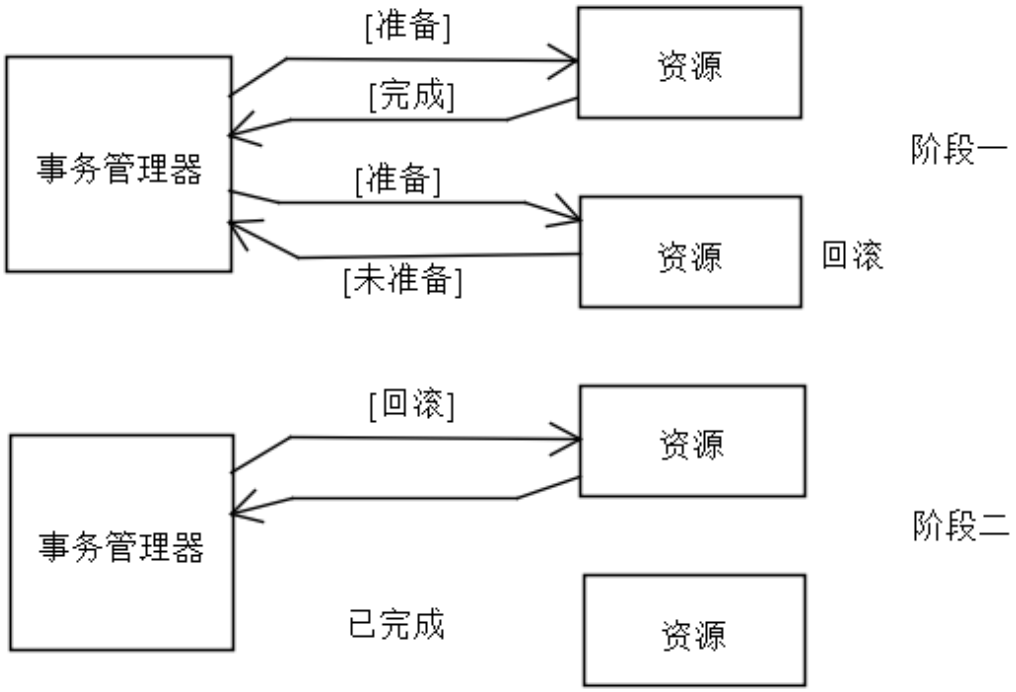
提交的类比为结婚典礼.当婚礼进行时,参与人(此处为新娘与新郎)都必须先在安排下许诺"我愿意", 之后才能正式地完成仪式.请考虑一下如果最后一分钟时其中一位参与者没有进行许诺所出现的可怕结果. 对于两阶段提交处理而言,虽然后果没有类比中的严重, 但出现"没有许诺" 时产生的结果也是一样的.

当一个 commit()请求从客户端发送到事务管理器时, 事务管理器开始两阶段提交处理. 在第一阶段, 所有的资源都会被询问是否已经准备好提交它们的事务. 此时每个参与资源都可以回复 Ready, READ_ONLY 或 NOT_READY 这三种响应.如果在阶段一里任何一个资源回复了 NOT_READY 响应,当前整个事务都会被回滚. 只有当所有的资源都回复 READY 响应时, 当前事务才会进入阶段二的提交处理. 如果参与者回复了 READ_ONLY 作为响应, 该参与者将不会进入阶段二的处理过程中(被忽略).

两阶段提交之所以可行是因为 XA 的双向通讯特性. 在非 XA 环境中,事务的通讯是单向的. 这样事务管理器就不能接收到一个资源管理器发送的响应信息. 两阶段提交在这种情况下也就不可能存在了. 大多数事务管理器会出于优化目的而在阶段一(通知资源)和阶段二(提交处理)时使用多线程来通知各个资源管理器, 并且会尽快地释放所占用的资源. 下面的图说明了基本的两阶段提交处理过程:



下面的示例图说明了在阶段一中通知资源时其中一个资源管理器发生错误(例如 DBMS)后的处理过程:



在这个例子中,一个运行在全局事务上的客户端发送了一个提交请求到事务管理器.但在阶段一中第二个资源管理器响应了

NOT_READY 状态.在这种情况下事务管理器将对所有的参与者发送回滚请求.以此来协调所有的资源,使其处于全局事务内.

一些商业的容器也提供了一种叫做最后的参与者支持的特性(也称为最后的资源提交优化). 最后的参与者支持允许一个不处于 XA 环境的资源参与全局事务.在这种情况下,当一个提交请求发送到事务管理器后,处于 XA 环境的事务管理器将首先进入阶段一去通知所有 XA 资源进行准备, 当所有处于 XA 环境的参与者都响应状态后,事务管理器将发送一个提交(或回滚)请求给非 XA 环境的参与者. 该非 XA 参与者返回的结果将直接影响后面的阶段二提交过程.如果非 XA 参与者返回结果是成功的,事务管理器将会进入阶段二来提交处于 XA 环境下的资源.反之,如果返回结果是失败的,事务管理器将会进入阶段二来回滚所有处于 XA 环境下的资源.

最后的参与者支持有两个显著的问题: 第一,该方式并不能交叉容器运行. 第二, 因为阶段一中通知过程结束与非 XA 参与者提交处理之间存在时间延迟,你可能会出现额外的启发式异常(下一小节讨论). 因上所述, 在不是绝对必要的情况下应该避免使用最后的参与者支持.

大多数商业应用程序服务器中, 还有一个优化处理为阶段一的提交优化.如果事务只包含一个参与者,阶段一将被忽略而这个唯一的参与者将被直接提交. 此时整个 XA 事务的结果就是基于该唯一参与者所返回的结果.

4) 启发式(heuristic) 异常处理

在两阶段提交的处理过程中,资源管理器可能会使用启发式决策,并在独立于事务管理器的情况下进行对它的事务进行提交或回滚操作. 启发式决策是一种做出智能选择的处理过程, 它是基于各种内外部因素的.当一个资源管理器使用了启发式决策, 并独立地提交或回滚事务后, 它将通过一个 HeuristicException 来通告客户端.

幸运地是,除了在 XA 下的两阶段提交处理外,启发式异常并不经常

出现. 启发式异常只在两阶段提交处理中发生, 特别是在阶段一中参与者回应之后. 该异常最常见的原因是在协调阶段一和阶段二的过程中出现了超时. 当通讯丢失或延迟时, 资源管理器为了释放资源可能会提交或回滚它所处的事务. 因而,在资源使用处于高峰期的时候启发式异常的出现是最频繁的.如果你的应用中出现了启发式异常, 你应该从事务超时, 资源锁定以及资源超载等问题上下手.偶尔的网络延迟或网络故障也可能导致启发式异常.还有就是正如前面章节中所述的, 在使用最后的参与者支持特性时也会使启发式异常出现的更频繁.

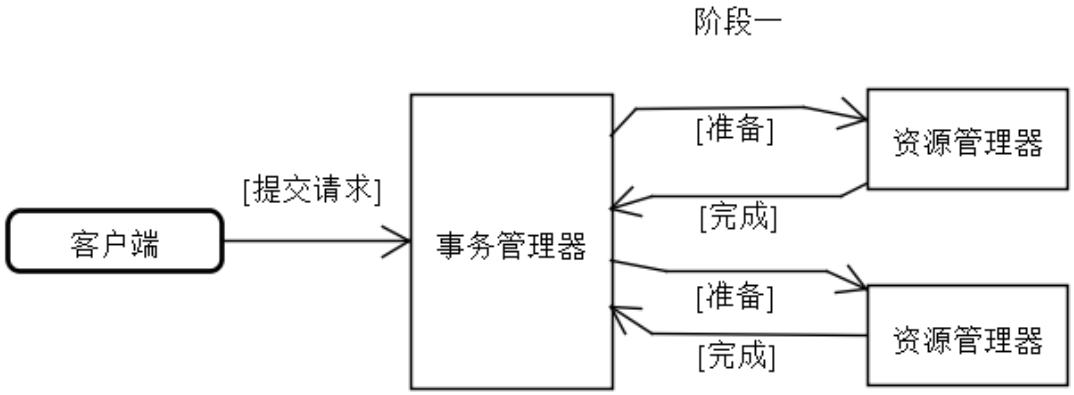
JTA 通过 3 个异常类来描述启发式异常.这 3 个异常类为: HeuristicRollbackException, HeuristicCommitException 以及 HeuristicMixedException. 下面将通过两个场景来解释这些异常:

场景一: 提交操作中的 HeuristicRollbackException

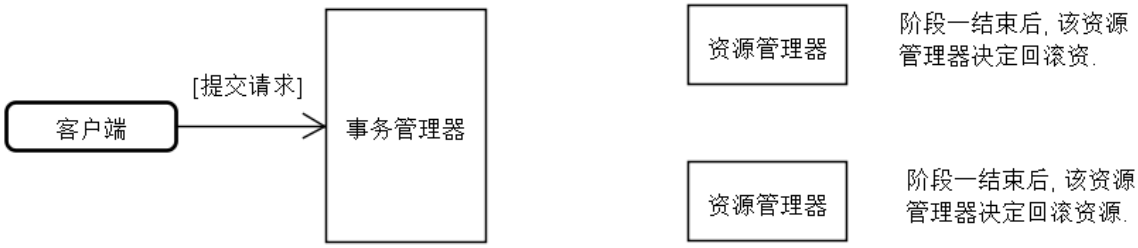
在该场景中客户端在 XA 环境下执行了更新操作,并向事务管理器发送了一个提交请求. 事务管理器进入了阶段一的处理, 通知了所有的资源管理器. 资源管理器分别响应事务管理器它们已做好事务提交准备.但在阶段一和阶段二之间资源管理器们都独立地做出了启发式决策来回滚它们的事务.紧接着事务处理进入阶段二, 提交请求被发送给了各个资源管理器. 此时因为有些资源在两个阶段之间已经进行了回滚, 事务管理器将会抛出一个 HeuristicRollbackException 给调用者.

当出现这种类型的异常时,典型的更正措施为将异常传递给外部客户端,让客户端再次发送提交请求. 我们不能在异常发生后又简单地再次调用 commit()方法, 因为此时在全局事务中其他的数据库所作的更新已经被回滚掉. 下面的序列图说明了这个场景.

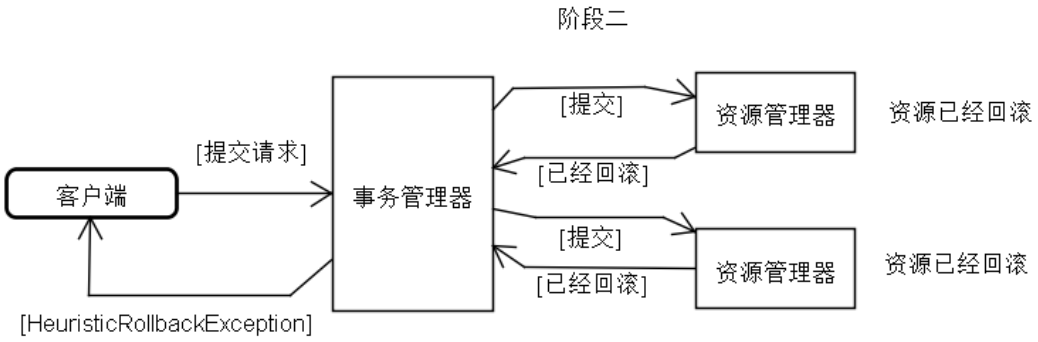
- **第一步: 阶段一处理(准备阶段)**



- **第二步: 阶段一与阶段二之间**



- **第三步: 阶段二处理(提交阶段)**



正如你从上图中看到的一样,虽然两个资源管理器都在阶段一响应了”准备回滚” 信息给事务管理器, 但二者都回滚了它们的资源. 不过不用在这里担心为什么会发生异常. 我将会在后面详述它.

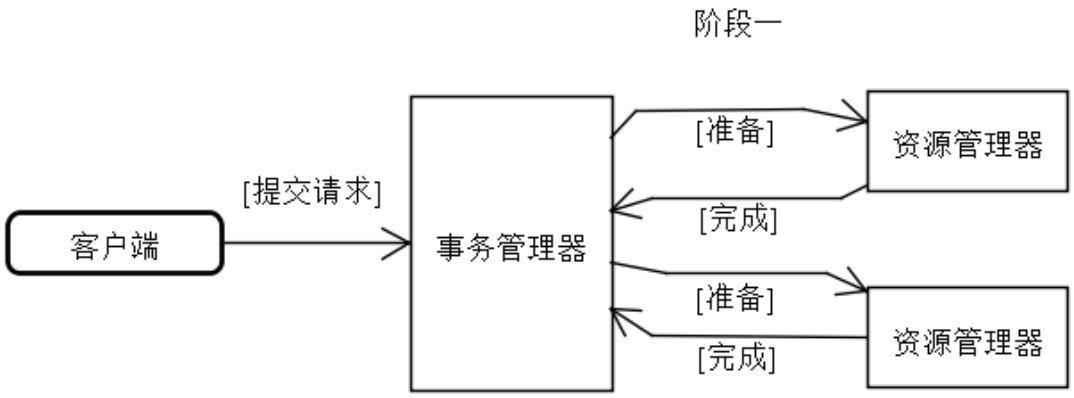
场景二: 提交操作中的 HeuristicMixedException

在该场景中客户端在 XA 环境下执行了更新操作,并向事务管理器发送了一个提交请求. 事务管理器进入了阶段一的处理, 通知了所

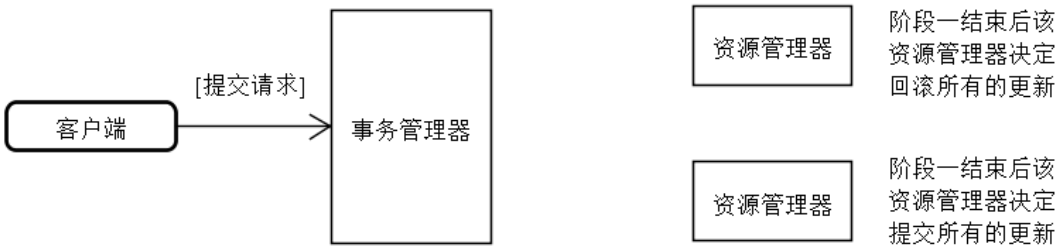
有的资源管理器. 资源管理器分别响应事务管理器它们已做好事务提交准备. 而在阶段一与阶段二之间,这个场景与场景一的不同之处在于一些资源管理器(例如 JMS 消息队列)做出启发式决策来提交它们的资源. 然而其他资源管理器(例如数据库)做出的启发式决策则是回滚它们的资源. 这种情况下事务管理器将会抛出一个 HeuristicMixedException 给调用者.

这种情况下就更难做出更正措施了, 因为我们并不知道那些资源已经被提交, 而那些资源已经被回滚了. 这些资源都处于一种不一致的状态中.因为一个资源管理器在独立地执行决策操作时并没有与其他资源管理器协调. 处理这种异常时通常是需要手动地进行干预的. 下面的序列图说明了这个场景.

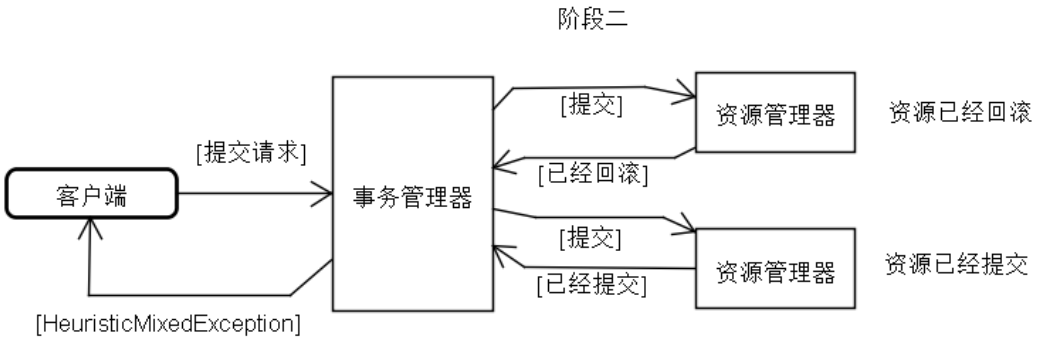
- **第一步: 阶段一处理(准备阶段)**



- **第二步: 阶段一与阶段二之间**



- **第三步: 阶段二处理(提交阶段)**



注意上面示例中一个资源管理器提交了它的更新，而另外一个则选择了回滚它的更新．在这种情况下事务管理器将抛出一个HeuristicMixedException.

5) 在 JMS 中使用 XA

在 XA 环境下一个资源必须实现 javax.transaction.xa.XAResource 接口才能被包含进 XA 全局事务.对于 JMS 的目的地(队列或主题)而言，这是通过在具体的应用程序服务器中使用管理控制台或者配置文件的方式来说明的．而你开启 XA 功能的配置所真正启用的是 JMS 连接工厂.当 JMS 连接工厂开启 XA 特性后，发送到 JMS 队列(或主题)的消息将不会被发布，而这将一直持续到两阶段提交处理结束后．在没有 XA 环境时，无论外部事务的结果如何，发送到 JMS 目的地的消息将会被立即发布，并且接受者可以立即提取消息.

在 Weblogic 中，对 JMS 连接工厂启用 XA 特性是在管理控制台下的 Services|JMS|ConnectionFactoys 中．在事务选项卡中有一个可选的”XA Connection Factory Enabled”选项，勾选该选项将对你的 JMS 连接工厂启用 XA.

在 WebSphere 中，对 JMS 连接工厂启用 XA 特性是在管理控制台下的 Resources|WebSphere JMS Providers|ConnectionFactoys 中．勾选”Enable XA” 选项将对你的 JMS 连接工厂启用 XA.

6) 在数据库中使用 XA

在数据库中启用 XA 是通过使用相应的数据库驱动来实现的.因为支持 XA 的数据库驱动在使用上比非 XA 的更困难.所以你应该只在你的总体事务管理中有用到 XA 的情况下才使用该驱动. 使用 XA 数据库驱动可能导致不期望的错误, 而且这些错误都是很难解决的. 例如, 将非 XA 数据库驱动替换为 XA 驱动后可能经常会产生一些非常难于跟踪的错误. 出于这个理由, XA 的数据库驱动应该尽早地在开发或测试周期中使用.

在使用 XA 数据库驱动时你可能遇到的错误类型为本地事务错误和嵌套事务错误. 这些错误是在你试图开始一个新事务,然而当前已有一个 XA 全局事务正在进行处理的情况下发生的. 这种情景将会在很多情况下发生. 但最常见的两个为混合使用本地事务与声明式事务和在 XA 环境下调用存储过程.

在 XA 环境下使用存储过程时,最常出错的情况是该存储过程中包含了 DDL 语句(即 CREATE TABLE, BEGIN TRANSACTION, END TRANSACTION 等). 在大多数情况下这都会导致 XA 错误, 并且也是最难被更正的. 例如, 在 Oracle 中, 你能在使用 XA 的时候遇到过下面的错误:

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

如果使用的是非 XA 的数据库驱动, 你就不会看到上面的错误.因为在非 XA 驱动下,JTA 事务在 DDL 语句执行时将会被挂起. 出现这个错误也指示你的存储过程中包含了 DDL 代码, 并且有一个本地事务(被资源管理器所管理着)正试图提交它的事务.

出于业务或者被多个应用所共享的原因, 从存储过程中移除 DDL 语句通常是很困难的.一个有效的变通方法是在调用存储过程之前手动地挂起当前的事务,并在调用后恢复事务的执行. 使用这个技术将避免与 XA 相关的本地和嵌套事务错误. 但是,使用该方法时存储过程所作的更新将不处于全局 JTA 事务之内,而是被独立地提交.这违反了事务的 ACID 属性. 因此这也是为什么该技术只是一种变通方法,而不是一种解决方法. 下面的代码片段说明了该技术的使用:

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

即使在使用声明式事务时我们可以获取 TransactionManager 来编程的挂起和恢复事务.并且该技术在 XA 环境下可以避免 SQL 异常的出现. 但它并没有真正地解决核心问题. 真正的解决方案是在存储过程中移除掉冲突的 DDL 语句或者使用一个支持嵌套事务的 JTA 事务服务.

7) 总结

本章中最重要的概念是理解你何时真正的需要使用 XA 数据库驱动.许多架构师和开发者总是坚持使用 XA 数据库驱动, 而实际上这是不必要的.如果你需要在相同事务内协调多于一个的可更新资源(数据库,JMS 或 JCA),你才有必要使用 XA 接口.否则,是不需要使用 XA 的.

另一个忠告是,当使用 XA 的时候,不要总是假设你的问题是由于 XA 数据库驱动导致的.很有可能不是驱动的问题,而是你的应用程序代码和事务处理逻辑的问题.

六.事务设计模式

前面章节中所描述的事务模型可以在 Java 中提供一个事务管理的

框架，但是它们并没有解决怎样创建一个有效的事务管理策略的问题.其实开发一个强健和有效的事务管理策略并不是很复杂的事情.使用本书中描述的事务设计模式将简化你的事务处理，允许你构建一个更强健的应用，并指引你开发一个有效的事务管理策略.

本章的目的是提供一个对事务设计模式的简单介绍,并描述各种应用架构中所使用的典型技术.这些将可以让你清楚下面三章所介绍的模式中那一种最适合你.

1) 关于模式的简短介绍

事务设计模式描述了针对某种特定类型的应用中,其所采用架构的总体事务设计策略.每个事务设计模式都包含了应该使用的事务模型,事务参数的设定和各种组件类型所应负责的事务管理.最终,某个所描述的事务设计模式将可以在你的应用程序架构中使用.

下面三个章节中将要描述的事务设计模式分别为: 客户端所有者的事务设计模式,领域服务所有者的事务设计模式 以及 服务委托所有者的事务设计模式.正如其名所示,这些事务设计模式都是基于组件责任模型的,每种组件类型分别负责管理自己的事务.这些事务设计模式包含了大部分前面章节中所叙述的最佳实践. 它们的目的是为你的应用中的事务管理提供一个模板，以减免你对应该使用什么样的设置所存在的疑惑.

典型情况下,对于大多数 Java 企业应用你都会使用领域服务所有者的事务设计模式来构建事务管理策略.该模式在 EJB 和 Spring 中都被广泛地使用. 正如你将在第八章所看到的，该模式依赖于声明式事务模型并且是非常容易实现的.它适用于 Java 企业应用架构中需要对后端服务对象进行远程访问以及不基于远程服务的简单应用中.

当你需要在应用中的持久层进行事务管理时,你应该使用客户端所有者的事务设计模式来实现你的事务设计策略.虽然该模式大多都使用于应用架构需要远程访问后端服务的时候，但它也是可以使

用在非远程访问的应用中,这类应用通常都在包含了本地细粒度服务对象.该事务设计模式将会在第七章描述.

服务委托所有者的事务设计模式是针对基于命令模式或服务委托模式的企业 Java 应用.该模式是领域服务所有者模式的特例, 它解决了一般情况下基于客户端/EJB 的事务管理中所产生的大部分问题.该模式是最容易实现的一个,并且可以在 EJB 和 Spring 中使用.第九章将会简短地介绍该模式以及和该模式关联的命令模式和服务委托模式.

七.客户端所有者的事务设计模式

正如其名所示, 在这种事务设计模式中客户端委托组件拥有着 JTA 事务.虽然该模式主要是在基于 Web 或 Swing 的客户端中调用 EJB 组件时使用, 但它也可以被用在包含本地细粒度服务对象的非远程应用中. 此时针对服务对象的多个请求需要处于单一的事务环境中.

1) 适用场景

在一个企业 Java 应用中,虽然强制地将事务管理移动到客户端是一种不良的设计, 但在某些特定情况我们还是不得不这么做. 请考虑一个最常见的情况, 某个客户端发起了单个请求, 但为了满足该请求却需要进行多个针对服务器的远程调用. 这种境况最常发生在领域服务太过于细粒度并且没有聚合服务存在时.在此类情况发生时,我们不得不将事务管理下移到呈现层来维护 ACID 属性.

下面例子中, 客户端接收到了一个预定固定收入交易的请求. 一个固定收入交易的例子是购买或出售债券及国债. 而普通的股票交易则是对某个特定股票(例如 Google)的购买或出售. 当一个固定收入交易预定时, 客户端在相同的事务内调用了 placeTrade()和 executeTrade()方法. 作为对比,在普通股票交易中这些方法将被单独的调用. 下面客户端业务委托对象中的代码说明了该情景

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
```

```
Public void placeFixedIncomeTrade(TradeData trade){
   try{
      // ...
      Placement placement =
             placementService.placeTrade(trade);
      placementService.sendPlacementMessage(placement);
      executionService.executeTrade(placement);
   }catch(TradeExecutionException e){
     log.fatal(e);
     throw e;
   }
}
```

因为上面的调用过程必须处于单个工作单元中，所以它们必须被包裹在事务环境内. 如果上面客户端层的代码中没有事务处理的话，因为 placeTrade()可能在 executeTrade()调用前就已经将结果提交到数据库中了，它们将不能维持事务的 ACID 属性.

虽然有许多其他的架构重构方法来避免使应用处于这种窘境，但仍可能存在一些特殊的必须使用基于客户端事务管理的情况. 基于客户端的事务管理所存在的一个问题是通常客户端包含了太多底层知识. 特别是对于在基于 web 的框架中，不使用委托对象而直接与远程 EJB 服务对象(或 Spring Bean)进行通讯的情况.避免使用客户端事务管理的一种方法是使用命令模式或服务委托模式. 这两种设计模式都属于服务委托所有者的事务设计，这将在第九章详述.

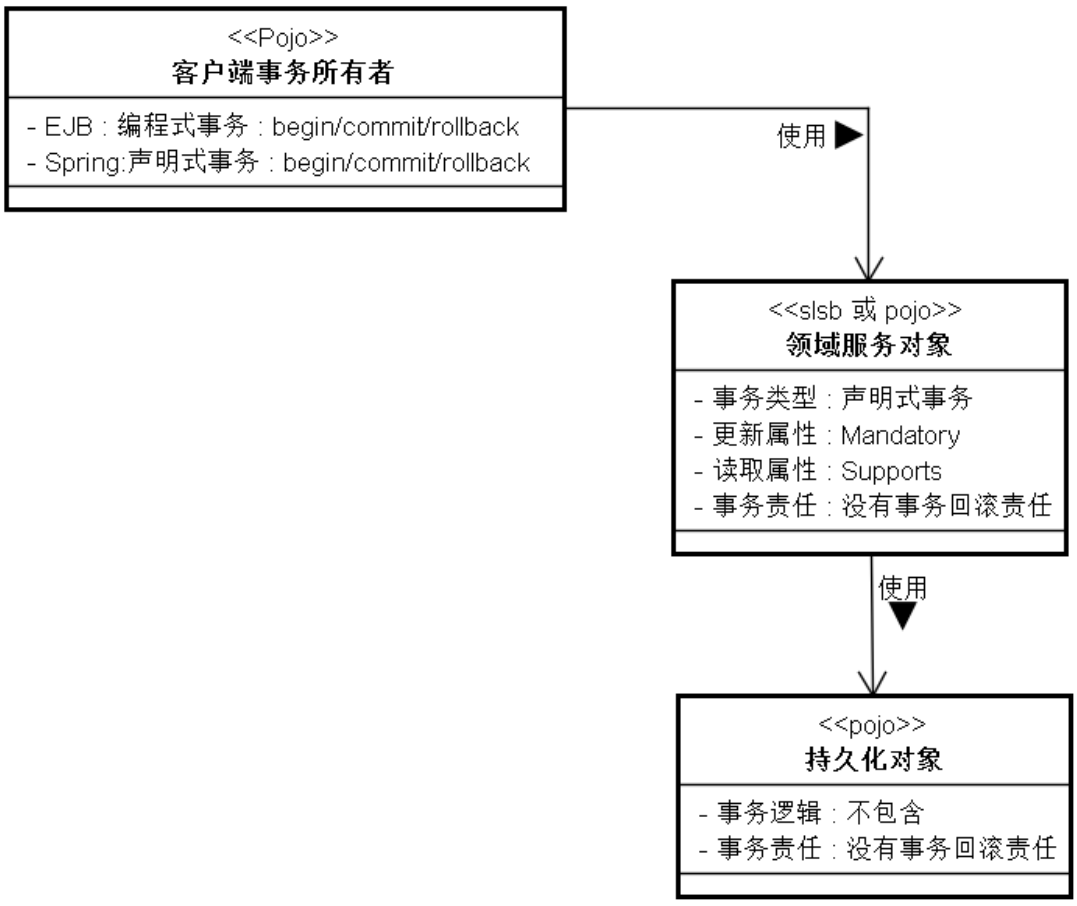
2) 所需满足的条件

- 为了满足单个业务请求,客户端需要发起多次针对远程或本地领域服务对象的调用.
- 事务的 ACID 属性必须被维持,这意味着必须进行事务处理来保证数据的完整性.
- 领域服务对象属于细粒度设计，并且当前没有将多个业务请求进行合并的聚合服务存在.

- 不能对应用程序的架构进行重构, 使其提供一个聚合方法来简化多次业务请求.

3) 解决方案

针对基于客户端的事务管理中存在的缺陷, 解决方案是使用客户所有者事务设计模式来作为总体的事务设计策略.该模式使用组件责任模型来替换客户端的事务管理责任. 下面的实例图说明了该模式在 EJB 和 Spring 中的使用 :



如上图所示, 在这种模式下管理事务的唯一组件是客户端业务委托者(即事务所有者). 如果客户端业务委托者是 Spring Framework 中的托管 Bean, 那么该组件可以使用声明式事务管理.而在其他情况下,则必须使用编程式事务管理. 例如 EJB 中该模型就必须使用第三章所述的 UserTransaction 接口.

在该模式下,领域服务组件必须使用声明式事务模型.我们曾在第三章中叙述过, 不能使用编程式事务来将一个在客户端建立的事务环境传递到远程(本地)服务对象中.但是, 当领域对象使用声明式事务时则可以进行传递.

因为 Spring 框架提供了对事务化 POJO 的支持, 一个在客户端业务委托者中建立的事务化环境也是可以传递到被 Spring 所托管的领域服务中. 但如果你在 Spring 中使用的是远程 Bean,你必须使用一个支持事务传播的远程协议(例如 RMI).

如上图中所说明的,领域对象中所有执行插入,更新和删除的方法必须将事务属性设置为 Mandatory. 最重要的是,领域服务对象在发生应用(checked)异常后决不能回滚事务. 正如第四章中所说的,事务回滚作为一种管理方式, 应该是被委托给开始该事务的组件. 在当前情况下, 开始事务的组件是客户端业务委托者, 而不是领域服务组件. 在领域对象中回滚事务将可能会使客户端无法采取正确的修复措施来使事务继续执行.

注意该事务设计模式在一个应用中同时使用了编程式和声明式事务.虽然混合使用编程式和声明式事务通常是一个不好的实践, 但当在基于客户端的事务中调用远程服务时,这是可以接受也是必须的一步(特别是对于使用 EJB 的情况).

4) 结论

- 管理事务的客户端对象在 EJB 中必须使用编程式事务.但如果客户端对象是 Spring 中的托管 Bean, 则可以使用声明式事务.
- 如果服务器端的领域服务对象是通过远程访问的, 当前的事务必须被传播到远程服务对象中. 例如,Spring 中所使用的大部分远程协议都不支持事务传播.
- 因为编程式事务无法传播到其他使用编程式事务的服务对象中, 服务器端的领域对象必须使用声明式事务.

- 为了维护事务的 ACID 属性，服务器端对象即不能开启或提交事务，也不能对事务进行回滚.
- 客户端所开启的事务是可以在当前事务范围内传播到任何被调的本地 POJO 中.
- 服务器端领域对象中所有的交互方法必须使用 Mandatory 事务属性.
- 虽然有时客户端和服务器端中的查询方法都不需要事务逻辑，但服务端领域对象必须对查询方法使用 Support 事务属性.
- 无论使用何种持久化框架,领域服务组件所使用的持久化对象都不应该包含任何事务处理逻辑.
- 如果在 EJB 2.1 中使用到了实体(Entity)Bean，所有的更新方法都必须将事务属性设置为 Mandatory，并且不需要处理事务回滚. 对于实体 Bean 中的读取操作，如果是使用容器管理的持久化(CMP)，需要将事务属性设置为 Required，如果是 Bean 自身管理的持久化，则需要将事务属性设置为 Supports.

5) 实现

下面的代码说明了该模式在 EJB 和 Spring 中的实现方式. 为了起到示例作用，我将假设 EJB 中的领域服务组件都是远程组件，而对于 Spring，客户端和领域服务组件都是托管 Bean.

Enterprise JavaBeans(EJB)

下面是客户端业务委托组件源代码:

```
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(TradeExecutionException e){
      txn.rollback();
      log.fatal(e);
      throw e;
    }
  }
}
```

EJB 服务组件中更新方法的源代码:

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

请注意 EJB 领域服务组件并没有捕获应用(checked)异常并调用 setRollbackOnly() 来回滚事务.因为此时 EJB 服务组件并不能在应用异常发生后对事务采取正确的补救措施. 我们并不需要在前面例子中那样使用 try/catch 块语句. 在第四章所述的最佳实践中, 如果组件并没有开始当前事务, 那么他就不会负责管理事务. 此时的事务属性应该被设置为 Mandatory, 并且不应该调用 setRollbackOnly()方

法.

Spring Framework

下面是客户端业务委托对象在 Spring 中的配置.

```
<bean id="clientModelTarget"
        class="com.trading.client.ClientMoel">
    <property name="tradingService" ref="tradingService">
    </property>
</bean>
<bean id="clientModel"
        class="org.springframework.transaction.interceptor.Tr
nsactionProxyFactoryBean">
    <property name="transactionManager" ref="txnMgr">
    <property name="target" ref="clientModelTarget">
    <property name="transactionAttributes">
      <prop key="*">PROPAGATION_Required,-Exception</prop>
      <prop key="get*">
        PROPAGATION_SUPPORTS
      </prop>
    </property>
</bean>
<bean id="tradingServiceTarget"
        class="com.trading.server.TradingServiceImpl">
</bean>
<bean id="tradingService"
           class="org.springframework.transaction.intercepto
.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="txnMgr">
    <property name="target" ref="clientModelTarget">
    <property name="transactionAttributes">
      <prop key="*">PROPAGATION_MANDATORY</prop>
      <prop key="get*">
        PROPAGATION_SUPPORTS
      </prop>
```

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

因为 Spring 能同时在客户端委托者和领域服务组件中使用声明式事务,所以下面代码中无论是更新还是查询的方法都没有包含任何事务处理逻辑. 通过如上面 XML 配置中设置 –Exception, Spring 将会自动地处理 setRollbackOnly(). 但请注意在 TradingService 中病没有设置相同的回滚规则. 这时因为我们不想在发生应用异常(checked)时回滚事务(这是 Spring 的默认设置).

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

  public void 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 void getTradeOrder(TradeKey key)
              throws Exception
```

```
  {
     TradeOrderDAO dao = new TradeOrderDAO();
     return dao.get(key);
  }
}
```

注意上面代码中无论客户端或服务组件中均不包含事务逻辑代码,但如果客户端模型不是被 Spring 托管的, 其事务处理代码将会同前面 EJB 例子中的相同.

八.领域服务所有者的事务设计模式

领域服务所有者事务设计模式是最常使用的事务设计模式, 并在你可能遇到的大部分基于 Java 的系统架构中都得到了应用.在该模式中,领域服务组件拥有着事务并进行所有事务相关的管理. 对于 EJB, 实现领域服务组件典型情况下是作为一个无状态会话 Bean 出现, 并因此而使用了声明式事务模型来管理事务. 而对于 Spring, 领域服务组件是通过 POJO 来实现的, 该方式下也是使用声明式事务模型来管理事务.

1) 适用场景

在企业 Java 应用中,服务层来负责管理事务是大多数架构师和开发者都认可的一种方式. 使用该方式的理由有以下几点: 第一, 我们可能并不知道访问后端服务的客户端是何种类型的. 例如,对于一个 Web Services 客户端而言, 它是很难开启新的事务并将事务传播到我们的领域服务中. 第二,如果将事务管理的责任分配给客户端, 典型情况下将会使客户端的实现难度加大. 而客户端主要负责的应该是呈现逻辑, 并不是像事务处理这样的服务端或中间件的处理.第三,使用客户端事务管理的架构在一般情况下将会使系统性能下降. 这是因为客户端通常需要进行多次对服务端的远程调用,而这将加大服务器的会话数, 因此降低了应用的总体性能. 最后,基于客户端的事务管理将会对总体的事务设计策略增加不必要的复杂性.

出于这些理由,企业 Java 应用中的领域服务对象通常被设计成中粒度至粗粒度. 这种粒度的领域对象把单次业务请求作为单个工作来处理. 随之也简化了总体的架构设计并提高了应用的总体性能.在这种情况下,服务端对象控制着事务的管理. 但由此而出现的问题是哪个组件应该负责开启,提交和回滚事务, 以及事务应该怎么被传播到持久化框架中.

请考虑下面的例子中,一个客户端向服务端发起调用请求来预订一笔固定收入的交易.就如我们在前面章节中看到的一样,这是通过在相同的事务工作单元内调用 placeTrade()和 executeTrade()方法来完成的.在使用粗粒度服务时,客户端执行该功能的代码将会变成下面这个样子:

```
Public void placeFixedIncomeTrade(TradeData trade){
   try{
      // ...
      placementService.placeFixedIncomeTrade(trade);
   }catch(TradeExecutionException e){
     log.fatal(e);
     throw e;
   }
}
```

因为客户端只向服务器发起了单次请求, 该客户端并不需要管理当前的事务. 即使该次请求在服务端可能包含多次对其他方法的调用. 因此,此时的事务管理逻辑必须包含在服务器端对象中.

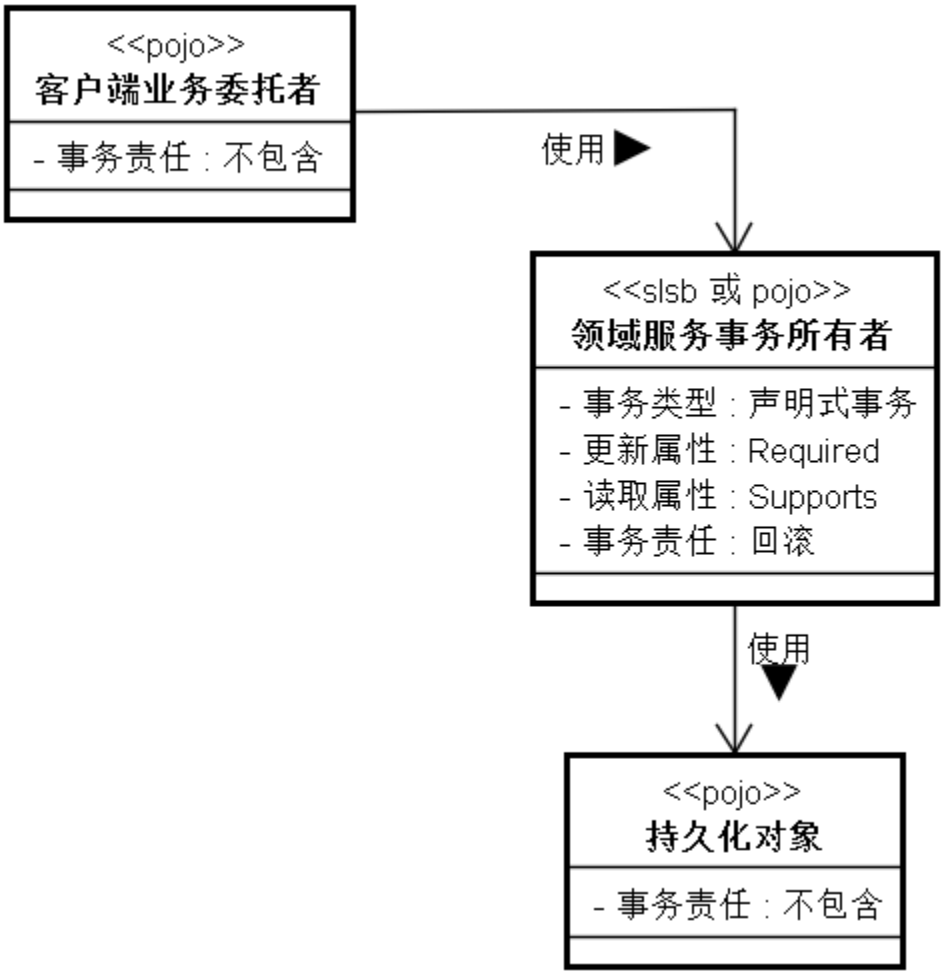
2) 所需满足的条件

- 客户端总是在只向领域服务对象发起单次调用的情况下满足业务需求.
- 此时的 ACID 属性必须被维护, 这意味着为了维护数据的一致性,必须对事务进行处理.
- 领域服务对象是使用粗粒度设计, 并(或)在服务对象之间进行

内部调用来执行聚合性的请求.

3) 解决方案

对于服务端事务管理的场景,典型的解决方案是使用领域服务所有者的事务设计模式来作为整体的事务设计策略.该模式下,无论领域对象是远程还是本地类型,均使用组件责任模型来管理应用中领域组件的事务处理逻辑.下面的图示说明了该模式在 EJB 和 Spring 中的使用.



如上图所示,在这种模式下领域服务组件是应用中唯一一个负责事务管理的组件.该架构中的其他组件类型均不知道事务的处理是否存在. 无论使用了何种框架,该模式中的领域服务组件均需使用声明式事务模型. 对于所有的插入,更新和删除方法, 需要将事务属性

设置为 Requred. 对于读取操作,则需要将事务属性设置为 Supports. 这个模式的核心原则是服务端的入口点(即领域组件中的方法被调用)拥有着当前事务,而其他被引用到的对象都将处于当前所开启的事务环境内.

该模式中对于领域服务组件所需要使用的事务属性是 Required, 而不像领域服务组件之间的通讯时所使用的 RequresNew 属性(有时这在企业 Java 应用中是很常见的). 这种事务设计模式是相当的简单并且在实现和维护上也是相对容易的. 因为此时只有领域服务组件包含事务处理逻辑, 这使得编码和配置的花费是很小的.

值得注意的是在该模式下领域服务组件之间的内部服务通讯将会处于两难的境地.内部服务通讯发生在一个领域服务方法调用了另一个领域对象中的方法时. 因为此时所有与更新相关的方法都是使用 Required 来作为事务属性, 如果调用时已经有一个事务环境了, 将会事务该事务环境而非创建一个新的事务. 但如果被调用的内部服务方法抛出了异常, 这将可能导致被调用方法在调用结束前就已经回滚了当前事务. 因此将使调用方法无法采取正确的修复措施. 虽然该情况是必须被考虑到的, 但它并不会对当前的事务设计模式产生重大影响.

对于大多数企业 Java 应用而言, 该模式是最常用的事务处理模型. 当领域服务组件所提供的方法是粗粒度设计并且当前的客户端只需单次调用就可满足业务需求时, 应该使用该模式. 相对于第七章中所述的使用多次远程请求的应用架构,该模式下的应用架构不仅提供了更好的性能,而且事务处理被放到了服务器端这个最合适的地方. 这对于使用 WebService 的客户端或第三方不容易修改的服务包而言是特别重要的.

4) 结论

- 无论何种类型的客户端, 均不需要包含任何事务处理逻辑.
- 因为领域服务组件开始和管理着事务,当更新方法中出现应用

(checked)异常后必须调用 setRollbackOnly()方法.

- 服务器端领域对象使用声明式事务模型，并且对于更新方法需要将事务属性设置为 Required，而读取方法需要设置为 Supports.
- 为了维护 ACID 事务属性，客户端对象决不能开始,提交或回滚事务.
- 无论使用那种持久化框架，领域服务组件所使用的持久化对象不能包含任何事务处理逻辑.
- 如果在 EJB 2.1 中使用到了实体(Entity)Bean，所有的更新方法都必须将事务属性设置为 Mandatory，并且不需要处理事务回滚. 对于实体 Bean 中的读取操作，如果是使用容器管理的持久化(CMP)，需要将事务属性设置为 Required，如果是 Bean 自身管理的持久化，则需要将事务属性设置为 Supports.

5) 实现

下面的代码说明了该模式在 EJB 和 Spring 中的实现方式. 为了起到示例作用，我将假设 EJB 中的领域服务组件都是远程组件，而对于 Spring，客户端和领域服务组件都是托管 Bean.

Enterprise JavaBeans(EJB)

不像客户端所有者的事务设计模式那样,当前模式下只有组件类型包含事务代码. 所以这里只显示该模式在领域服务对象中的实现.

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

请注意 updateTradeOrder()方法在发生应用(checked)异常后调用了 setRollbackOnly()来回滚事务. 并且也请注意该例子对第四章中所述的最佳原则的体现，该组件在类级别设置了约束较大的 Required 属性，然而对于 getTrade()这个读取方法则使用了 Supports 来覆盖类级别的默认属性. 因为在前面并没有事务环境存在，此处的 Required 属性将会在方法被调用时开启一个新的事务.

Spring Framework

该模式在 Spring 中是通过其专有的 XML 配置文件实现的.通过在 Bean 配置文件中说明事务回滚规则可以得到与 EJB 中调用 setRollbackOnly() 相同的效果.下面是 Spring 中领域服务组件的配置代码.

```
<bean id="tradingServiceTarget"
```

```
        class="com.trading.server.TradingServiceImpl">
</bean>
<bean id="tradingService"
            class="org.springframework.transaction.intercepto
.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="txnMgr">
    <property name="target" ref="clientModelTarget">
    <property name="transactionAttributes">
       <prop key="*">PROPAGATION_RRQUIRED,-Exception</prop>
       <prop key="get*">
          PROPAGATION_SUPPORTS
       </prop>
    </property>
</bean>
```

因为在该模式下领域服务组件使用的是声明式事务，在该领域服务组件的 Java 代码中无论是更新还是查询方法，均没有包含任何事务处理逻辑.正如前面所述的，Spring 将会通过配置中-Exception 这个回滚规则来自动地处理 setRollbackOnly()方法的调用. 下面的代码中说明了 Java 代码在该模式下并不包含任务事务逻辑.

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

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

```
}
```

九.服务委托所有者的事务设计模式

当你的应用架构中使用到了命令模式或服务委托模式的时候,服务委托所有者的事务设计模式是最适用的.在该模式下,服务委托组件将作为服务端的远程进入点, 其拥有着当前事务并且负责对所有的事务进行管理. 此时,包括客户端组件,领域服务组件以及持久化对象等其他组件均不处理事务逻辑.它们甚至也不需要知道它们正在被使用.

命令模式是一种很有用的设计模式.它可以在总体上解决客户端事务管理以及 EJB 中的许多问题.该模式背后的基本原则是客户端功能的实现是作为一个命令(Command)来传送到服务端执行的. 该命令可能包含一个或多个对服务方法的调用. 但是, 那些针对服务方法的调用是通过实现相应的命令对象后在服务器端被执行的. 使用该模式允许你只需从客户端向服务端发起单次请求就可满足业务需求.并且也使得事务的处理是在服务端进行而非客户端.

服务委托设计模式与命令模式在概念上是相似的, 唯一的不同之处在于该模式不是使用像命令模式这样的过程执行框架, 而是简单地将客户端委托者的业务逻辑代码移动到服务端的委托对象中. 二者的最终结果都是相似的: 事务处理被移动到服务端,由此将针对服务器的多次调用请求减少为单次调用.

服务委托所有者的事务设计模式是领域服务所有者设计模式的一个特例. 二者的主要不同在于当前者使用命令模式来实现时,拥有事务环境的是单个命令对象, 而非某个组件类型.例如,如果你的应用中包含了 40 个领域服务组件,当使用领域服务所有者模式时这 40 个组件都管理着各自的事务, 而使用服务委托所有者模式时却只有唯一一个组件(命令处理者)管理着事务.

1) 适用场景

请考虑下面在 EJB 中的示例代码,此处客户端对象必须发起多次向服务端的远程调用来满足业务需求.

```
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;
     }
  }
}
```

在这种情况下客户端必须使用编程式事务来将多个操作包含在事务化的单一工作单元内，以此确保事务的 ACID 属性.在这种方式下,不仅使客户端负责繁重的事务管理，而且由于发起了多次远程调用的缘故，也会对应用的性能造成影响.此外，在该例子中领域服务对象是在 EJB 中作为无状态会话 Bean 存在的,这进一步使架构变得复杂化.

为了更一步的简化这种情况，我们可以将无状态会话 Bean 重构为 POJO，从客户端移除事务逻辑,并只向服务端发起一次远程请求.为了达到这些目的，我们可以使用命令模式或服务委托模式.当使用二者之一后，原来客户端中的事务处理代码将会移动到服务端.下面的

代码显示了使用命令模式后的客户端:

```
public class ClientModel
{
  public void placeFixedIncomeTrade(TradeData trade)
              throws Exception
  {
    PlaceFITradeCommand command = new PlaceFITradeCommand();
    command.setTrade(trade);
    commandHandler.execute(command);
  }
}
```

这时的代码比前面的版本简化了许多. 但是,仍然存在的问题是事务处理逻辑需要放在那里. 在这种场景下,服务委托所有者的事务设计模式能够识别出那些组件需要进行事务管理, 以及这些组件需要什么样的事务设定.

2) 所需满足的条件

- 应用架构中使用命令模式或服务委托模式来进行客户-服务器端的通讯.
- 客户端总是在只向领域服务对象发起单次调用的情况下满足业务需求.
- 此时的 ACID 属性必须被维护, 这意味着为了维护数据的一致性,必须对事务进行处理.
- 领域服务对象是被实现为一个 POJO, 它可以是远程或本地类型.
- 命令处理者或服务委托者是领域服务中客户端所能见的唯一进入点.

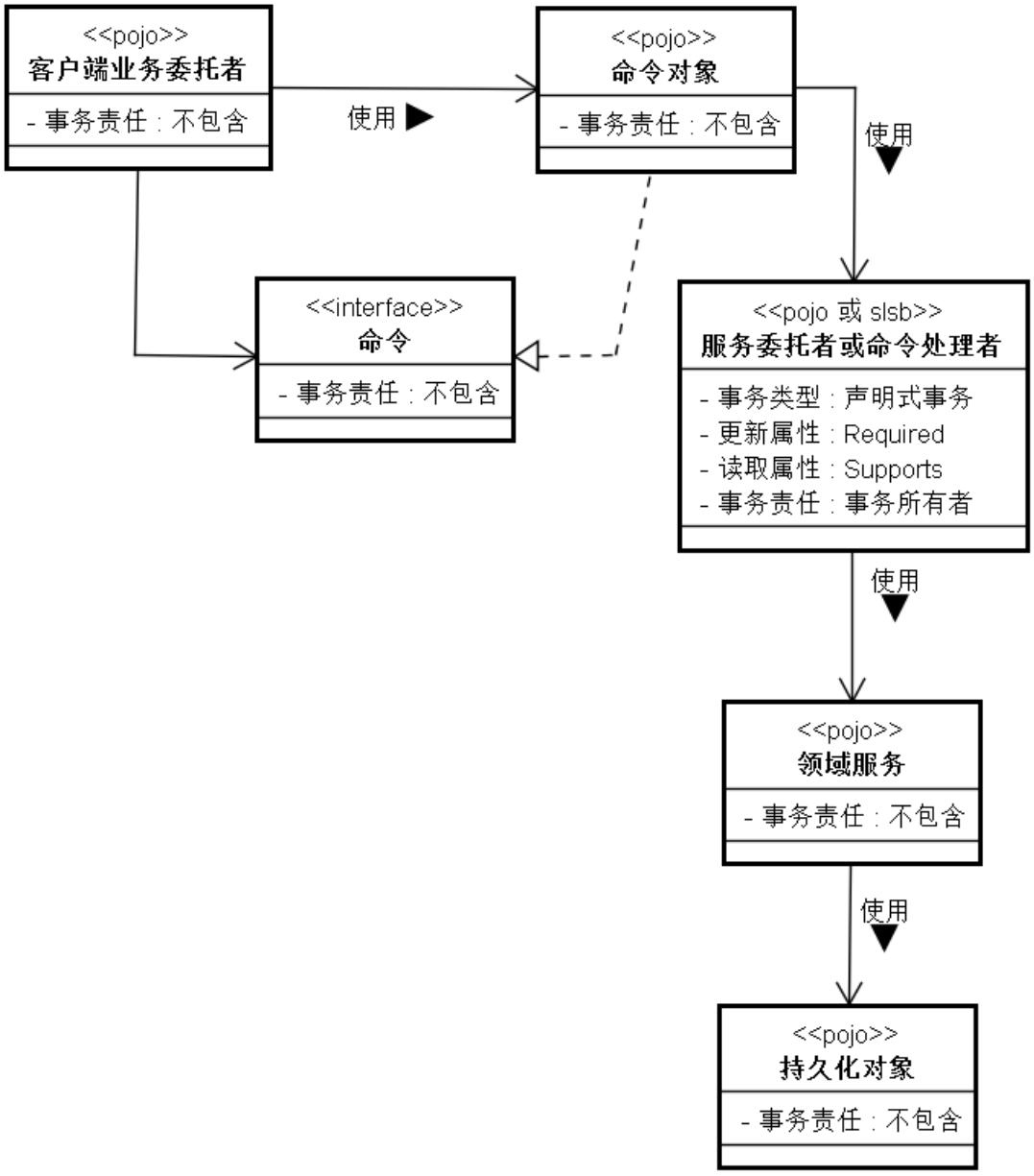
3) 解决方案

当应用架构中使用了命令模式或服务委托模式来封装事务处理时,可以采用服务委托所有者的事务设计模式来作为总体的事务设计策略.该模式在服务委托组件上使用组件责任模型来负责管理所有的事务.

当使用命令模式时,服务委托组件将实现为一个命令处理者.该命令处理者接受相应的命令实现对象, 并执行这些命令对象. 对于 EJB 而言, 典型的情况下这些组件都是作为一个无状态会话 Bean 来实现. 而对于 Spring, 这些组件将被实现为 Spring 托管的 Bean(POJO).

当使用服务委托模式时,对应客户端请求的每个功能组都将被实现为独立的服务委托者(Spring 中的 POJO 或 EJB 中的无状态会话 Bean).

下面的图说明了该模式在 EJB 和 Spring 中的实现:



如上图所示,服务委托组件使用声明式事务，对于所有更新方法,将事务属性设置为 Required. 而对于读取方法，设为 Supports. 此时的服务委托组件在出现应用异常后需要调用 setRollbackOnly()来回滚事务.

使用该模式后的应用架构是非常独特的. 在使用命令模式的情况下,服务委托组件通常是作为一个命令处理者实现的,该处理者简单地从客户端接受命令对象，然后执行该对象.在命令模式中接口的使用确保了命令对象和命令处理者都保持通用性，并且二者对于其他组件都是不知晓的. 此时服务委托组件所建立的事务环境将会传播

到所有被它调用的对象中.

而对于使用服务委托设计模式的情况,服务委托组件将作为应用中领域服务组件的一个门面(Façade). 该门面向客户端暴露了简化后的调用接口. 在这种模式下,客户端的处理逻辑在本质上是移动到了服务端的委托组件中.

这些设计模式的一个主要缺点是服务委托组件包含了客户端的处理逻辑,而非纯服务端处理逻辑. 此外,因为客户端业务处理是经常与具体的 Web 框架相耦合的,这使得该模式在某些情况下的实现变得很困难. 例如,当使用 Struts 来作为前端 Web 框架时,其中的 Action 类是经常作为客户端业务委托者的. 在这种情况下,由于包含了对诸如 HTTPSession,HTTPRequest 和 HTTPResponse 这类客户端对象的引用, 将处理逻辑移动到服务端将变得很困难.

但是,这些模式的一个明显优点是领域服务组件由 EJB 变成了 POJO. 领域服务组件也因此与 EJB 解除了耦合关系. 这使得领域服务组件变得更容易测试. 服务委托所有者的事务设计模式还有另一个独特面,由于服务委托者通常是实现为单例的无状态会话 Bean, 这使得整个应用中的事务处理逻辑都包含在了单个对象中. 由此产生的结果是该模式在实现和维护上变成了最简单的事务设计模式. 此外,这个事务设计模式也使得领域服务组件卸下了事务管理的重担. 将领域服务组件从诸如事务管理这类的底层知识中释放出来, 让其集中在处理业务逻辑上. 这种情况下,由于领域服务组件处于 POJO 状态, 将更容易在容器环境外对其进行测试.

4) 结论

- 无论何种类型的客户端,均不包含任何事务处理逻辑.
- 因为服务委托组件开始和管理着事务,当更新方法中出现应用(checked)异常后必须调用 setRollbackOnly()方法.
- 服务委托者建立的事务环境将传播到 POJO 状态的领域服务对象以及其所使用的持久化对象中(无论使用什么持久化框架).

此外,这些被传播的对象均不包含任何事务逻辑.

- 服务器端领域对象使用声明式事务模型，并且对于更新方法需要将事务属性设置为 Required，而读取方法需要设置为 Supports.
- 为了维护 ACID 事务属性，客户端对象决不能开始,提交或回滚事务.
- 无论使用那种持久化框架，领域服务组件所使用的持久化对象不能包含任何事务处理逻辑.
- 如果在 EJB 2.1 中使用到了实体(Entity)Bean，所有的更新方法都必须将事务属性设置为 Mandatory，并且不需要处理事务回滚. 对于实体 Bean 中的读取操作，如果是使用容器管理的持久化(CMP)，需要将事务属性设置为 Required，如果是 Bean 自身管理的持久化，则需要将事务属性设置为 Supports.

5) 实现

下面的代码说明了该模式在 EJB 和 Spring 中的实现.为了起到示例作用，我将假设 EJB 中的领域服务组件都是远程组件，而对于 Spring，客户端和领域服务组件都是托管 Bean.

Enterprise JavaBeans(EJB)

在命令模式下该事务设计模式只有唯一一个包含事务处理代码的组件(命令处理者组件).因此，客户端并不需要包含任何事务处理代码.所以这里只需要显示该模式在服务委托者(命令处理者) 中的代码.

```
@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(TradeUpdateException e){
        sessionCtx.setRollbackOnly();
        throw e;
      }
  }
}
```

上面示例代码中的 getCommandImpl()方法使用反射来装载命令实现类,并实例化命令实现对象.在执行结束后,执行结果将通过命令对象传递到客户端. 注意该实现与领域服务所有者的事务设计模式是很相像的. 它们均对读取操作使用了 Supports 属性, 对更新操作使用了 Required 属性并在发生异常后调用 setRollbackOnly()方法.

Spring Framework

该模式在 Spring 中是通过其专有的 XML 配置文件实现的.通过在 Bean 配置文件中说明事务回滚规则可以得到与 EJB 中调用 setRollbackOnly() 相同的效果.下面是 Spring 中领域服务组件的配置代码.

```
<bean id="commandProcessorTarget"
    class="com.commandframework.server.commandProcessorImpl">
</bean>
<bean id="commandProcessor"
            class="org.springframework.transaction.intercepto
.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="txnMgr">
    <property name="target" ref="clientModelTarget">
    <property name="transactionAttributes">
      <prop key="executeUpdate">
        PROPAGATION_REQUIRED,-Exception
      </prop>
      <prop key="executeRead">
        PROPAGATION_SUPPORTS
      </prop>
    </property>
</bean>
```

因为在该模式下领域服务组件使用的是声明式事务，在该领域服务组件的 Java 代码中无论是更新还是查询方法，均没有包含任何事务处理逻辑.正如前面所述的，Spring 将会通过配置中-Exception 这个回滚规则来自动地处理 setRollbackOnly()方法的调用.下面的代码中说明了 Java 代码在该模式下并不包含任务事务逻辑.

```
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);
    }
}
```

请注意在上面代码中,由于 Spring 将会根据配置的回滚规则来自动处理 setRollbackOnly(). 所以并不需要像 EJB 实现中那样手动处理异常. 这使得更新和读取这两个方法的代码都是一样的.

十.全文总结

通过理解三种事务模型并使用本书中描述的设计模式,你可以简化你的事务处理并为你的应用创建一个更有效,更强健的事务设计策略. 为了在你的应用程序和数据库中维护数据一致性,事务处理是非常重要和必须的. Java 中的事务管理并没有那么复杂. 使用前面章节中描述的事务设计模式将使事务处理变得容易理解,实现和维护.

关于作者

马克·理查兹(Mark Richards)是IBM的高级认证IT架构师，他参与并设计了在J2EE和其他技术中的大型面向服务应用，其中主要是金融服务行业。他从 1948 年以来就进入了软件行业，并曾担任过开发人员，软件设计师和架构师。他在J2EE架构和开发，面向对象的设计和开发以及系统集成上有着丰富的经验和专业的知识。马克曾在 1997 年和 1998 年担任波士顿Java用户组主席，以及从 1999 年至 2003 年担任新英格兰Java用户组主席。 除了这本书外，他也是一本即将到来的书籍<<NFJS文选 2006 年版>>（2006 年 6 月）的合著者，新英格兰Java用户组下编码标准规范小组制作的<<Java编码标准>>一书的合著者。马克是一名IBM认证的应用程序架构师，Sun认证的J2EE业务组件开发人员，Sun认证的J2EE企业架构师，Sun认证的 Java程序员，BEA认证的WebLogic开发人员以及获得认证的Java讲师。他拥有波士顿大学的计算机科学硕士学位。马克是No Fluff Just Stuff Symposium 系列的固定演讲者，并经常在用户群体和全国各地的其他会议演讲。