# 如何实现XA式、非XA式Spring分布式事务

2015/04/22 | 分类： 基础技术 | 0 条评论 | 标签： SPRING, 分布式
分享到： 31

本文由 ImportNew - 乔永琪 翻译自 javaworld。欢迎加入翻译小组。转载请见文末要求。

## Spring应用的几种事务处理机制

Java Transaction API和XA协议是Spring常用的分布式事务机制，不过你可以选择选择其他的实现方式。理想的实现取决于你的应用程序使用何种资源，你愿意在性能、安全、系统稳健性、数据完整方面做出何种权衡。在这次JavaWorld大会上，来自SpringSource的David Syer跟大家分享了Spring应用的几种事务处理机制、三种XA式、四种非XA式事务协议。

Spring框架支持Java Transaction API（JTA），这样应用就可以脱离Java EE容器，转而利用分布式事务以及XA协议。然而即使有这样的支持，XA开销是昂贵的，不稳定而且笨重不利于管理，不过一些其他的应用可以避免使用XA协议。

为了让大家对所涉及的几种分布式事务有所了解，我会分析七种事务处理模式，并 给出具体代码实现。并且从安全或者稳定性入手倒序展示，可以看看从安全、稳定性出发，如何在一般场景下，保障数据高完整性和原子性。当然随着话题的深入， 更多的说明以及限制就会出现。模式也可以从运行时开销倒序展示。考虑到所有模式都是结构化或者学术性的，这一点有别于业务模型，因此我不打算展开业务用例 分析，仅仅关注每种模式其少部分代码如何工作的。

尽管只有起初的三种模式涉及到 XA协议，不过从性能角度出发，这些模式或许无法满足需求。考虑到这些模式无处不在，我不想做过多地扩展，只是对第一种模式做一个简单的展示。读完此文，你可以了解可以用分布式事务做些什么、不能做什么以及如何、何时避免使用XA，何时必须使用。

## 分布式事务以及原子性

分布式事务涉及不止一个事务资源。比如，在关系数据库和消息中间件之间通信的连接器，通常这些资源拥有类似begin()、rollback()、commit()的API。在此，一个事务资源通常是一个工厂产品，这个工厂通常由底层平台提供：以数据库为例，DataSource提供Connection，或者Java Persistence API(JPA)的EntityManager接口；又如Java Message Service(JMS)提供的Session。

一个典型的例子，一个JMS消息触发一次数据库更新。此过程可以分解成一时间线，一个成功的交互顺序是下面这样：

1. 开启消息事务

**2. 接受消息**

3. 开启数据库事务

**4. 更新数据库**

5. 提交数据库事务

6. 提交消息事务

如果数据库出错，比如更新时出现诸如违反约束的问题，一个理想的顺序应该是下面这个样子：

1. 开启消息事务

**2. 接受消息**

3. 开启数据库事务

**4. 更新数据库失败**

5. 回滚数据库事务

6. 回滚消息事务

在这个案例中，最后的回滚发生后消息返回给中间件，并且在某种程度返回的消息会被其他事务所接收。通常这是件好事，可能你并没有对失败做记录。自动重试处理异常机制超出了本文的范畴。

以上两种时间线中最重要的特性是它们的原子性，形成一个单一的逻辑事务单元，要么都成功要么都失败。

那么用什么确保时间线会的顺序呢？事务资源之间必须保持某种同步，一旦对某个数据源做提交，要么都提交了，要么都回滚。否者整个事务就不缺乏原子性。之所以是分布式事务，是因为有多个数据源，没有同步就没有原子性。分布式事务技术和概念的核心问题都是围绕资源的同步或者无法同步展开的。

前三种模式的以下讨论都是基于XA协议，考虑到这三种模式分布广泛，本文不会涉及太多的细节，倘若你熟悉XA模式或许愿意直接跳到共享事务资源模式。

## 二阶段提交完整XA协议

如果你需要近乎完美的防护 （close-to-bulletproof）确保你的应用事务在断电后恢复以及服务器崩溃，完整XA是不二之选。共享资源通常需要做事务同步，在此情况下，它是一个采用XA协议协调处理过程的信息特殊的事务管理器。在Java领域，从开发者的角度看，这个协议是通过JPA UserTransaction暴露给大家。

基于系统接口，XA作为一种促成科技（enabling technology）对多数开发人员不可见，因此他们需要知道XA在哪、促成什么、耗损如何以及如何利用事务资源。事务管理器采用二阶段提交（2PC）协议，在确保事务结束前所有资源采用同一个事务结果的同时，也会带来性能耗损。

如 果是Spring促成的（Spring-enabled），应用会采用Spring的JtaTransactionManager以及Spring声明式 事务管理，这样会隐藏到了底层事务同步的具体细节。对于开发人员用没用XA的差别就在于对工厂资源的配置:DataSource实例，以及应用的事务管理 器。本文会通过一个应用案例（atomikos-db项目）来揭示这个配置，数据库实例和事务管理器仅是XA或者JTA特定的应用元素。

为了揭示此案例如何工作，在com.springsource.open.db.下运行这个单元测试。一个简单的 MulipleDataSourceTests类仅是将数据插入两个数据源中，并且采用Spring整合支持的特性对事务进行回滚，代码见清单1:

**清单1、事务回滚**

```
@Transactional
  @Test
  public void testInsertIntoTwoDataSources() throws E

    int count = getJdbcTemplate().update(
        "INSERT into T_FOOS (id,name,foo_date) valu
        "foo");
    assertEquals(1, count);

    count = getOtherJdbcTemplate()
        .update(
            "INSERT into T_AUDITS (id,operation,nam
            0, "INSERT", "foo", new Date());
    assertEquals(1, count);

    // Changes will roll back after this method exi

  }
```

接着验证这两个操作是否同时回滚，代码清单如清单2:

**清单2、回滚验证**

```
@AfterTransaction
  public void checkPostConditions() {

    int count = getJdbcTemplate().queryForInt("selec
    // This change was rolled back by the test fram
    assertEquals(0, count);

    count = getOtherJdbcTemplate().queryForInt("sel
    // This rolled back as well because of the XA
    assertEquals(0, count);

  }
```

更进一步理解Spring事务管理如何工作以及如何配置，请参看Spring参考指南。

## 一阶段提交优化XA协议

许多事务管理器采用这种优化模式，可以避免单一事务资源下的2PC过度开销，你的应用服务器最好能够判别此种情况。

## 协议和最终资源策略

多数XA事务管理器另一个特性是，不论是单一XA兼 容资源还是所有资源都XA兼容，事务管理器均能提供相同的恢复保障。它们是通过给资源排序，并且给非XA资源投票实现，倘若事务提交失败，所有其他的资源 都能回滚。事务有近乎百分百的保障，但缺点是，倘若事务失败，此时不会留下太多信息。换言之，如果要获取这些信息，需要做一些额外的步骤，比如在一些高级实现。

## 共享事务资源模式

这个模式不错，系统所有的事务资源由一个相同的资源提供支持进而移除XA，降低系统的复杂度，提高吞吐量。当然不能拿来处理所有的用例，但却是如XA般坚固，而且处理速度更加的快。共享事务资源模式作为一种保障存在与特定的平台和处理场景中。

一个简单熟悉的例子就是共享一个数据库的Connection，它存在于一个对象关系模型（ORM）控件和一个JDBC控件之间。Spring事务管理器就是如此，它支持ORM工具，比如Hibernate、EclipseLink以及Java Persistence API（JPA）。相同的事务能安全的跨越ORM和JDBC控件之间，通常此事务是由service层受事务控制的执行方法所驱动的。

此模式的另外一个特点是，消息驱动的单个数据库更新，如本文初始阶段的简单例子。消 息中间件系统需要存储这些数据，通常是关系型数据库。实现这种模式，需要将消息系统指定到相同的用于存储业务数据的数据库中。这种模式依赖消息中间件供应 商所提供的存储策略细节，以便能够将消息中间件配置在相同的数据库中，并嵌入相同的事务处理。

不是所有的供应商都提供了此种模式，不过一种可替代，几乎可以用于任何数据库的方式，即利用Apache ActiveMQ的传递消息，并且插入一个存储策略进入消息代理中。一旦你知道了其中的诀窍，配置起来很容易的，我会在本文的 shared-jms-db 项目案例中演示。此模式的所用的代码无需关注，它们会在Spring配置中得到声明。

名为 SynchronousMessageTriggerAndRollbackTests 的案例中，单元测试校验所有与同步消息接收者相关的讯息。 testReceiveMessageUpdateDatabase方法接受两个消息，接着利用消息插入两条记录到数据库中。如果此方法退出，测试

框架回 滚事务，这样就能校验消息和数据库更新是否发生回滚，如清单3所示：

**清单3、验证消息和数据库更新是否回滚**

```
@AfterTransaction
public void checkPostConditions() {

  assertEquals(0, SimpleJdbcTestUtils.countRowsInTa
  List<String> list = getMessages();
  assertEquals(2, list.size());

}
```

配置文件最重要的部分就是ActiveMQ的持久化策略，连接消息系统和相同数据源作为业务数据，Spring JmsTemplate的flag标签用来接收消息。清单4 展示了如何配置ActiveMQ持久化策略：

**清单4、ActiveMQ持久化配置**

```
<bean id="connectionFactory" depends-on="brokerServic
      <property name="brokerURL" value="vm://localhos
</bean>

<bean id="brokerService" init-method="start" destroy-
       <property name="persistenceAdapter">
          <bean>
            <property name="dataSource">
                 <bean>
                      <property name="targetDataSour
                      <property name="jmsTemplate" re
                 </bean>
            </property>
            <property name="createTablesOnStartup" va
          </bean>
       </property>
</bean>
```

清单5展示了Spring JmsTemplate中用来接收消息的flag标签：

**清单5、设置JmsTemplate事务应用**

```
<bean id="jmsTemplate">
        <!-- This is important... -->
        <property name="sessionTransacted" value="tru
</bean>
```

若 sessionTransacted不为true，JMS session transaction API就无法被调用，消息接受者将无法回滚。最为重要的是，嵌入的代理包含一个特殊的async=false参数以及DataSource外包类，这样就可以确保ActiveMQ拥有同Spring一样的事务JDBC Connection。

一个共享数据库源可以由独立的单个数据源组成，特别是这些数据源拥有同样的RDBMS平台。企业级数 据库供应商均提供同名概念（the notion of synonyms）支持，表可以作为一个同名（synonyms）声明于多个schema中。借助这个手段，分布在不同物理平台上的数据，可以均可 JDBC client同意Connection事务管理。比如在一个真实的系统中，采用ActiveMQ共享资源模式实现，通常需要为消息和业务数据创建同名。

## 性能和JDBCPersistenceAdapter

ActiveMQ 社区的一些开发人员表示JDBCPersistenceAdapter会引发性能问题。然而，许多项目和上线系统采用ActiveMQ与关系型数据库搭配使 用。在这些案例中，公认的是日志版本的适配器可以用来提供性能，当然这对共享资源模式来说是不利的，因为日志本身就是一个新的事务资源。然而，对于JDBCPersistenceAdapter大家众说纷纭，看法不一。确实，有理由相信采用共享资源或许能提高日志案例的性能。这个一结论来自 Sping以及AtiveMQ工程师团队的研究。

非消息场景（多个数据库）的另一种共享资源技术就是，在RDBMS平台一级利用Oracle数据库链接一个特征到两个数据库schema中。或许这需要改变应用代码，或者创建同名，因为表的别名会指向一个已链接数据库，此数据库包含链接的名称。

## 最大努力一次提交模式

开发人员必须知道，最大努力一次提交模式应用相当的普遍，但是在一些场景并不适用。这是一种非XA模式，它包含一 个同步大量资源单一相提交（single-phase commit）。参与者应该意识到这种折中，如果不用两阶段提交，那最大努力一次提交模式的安全性不如XA事务但也是相当不错。许多海量数据、大吞吐量事 务处理系统用最大努力一次提交模式提高性能。

最基本的理念就是在单一事务中尽可能的延迟所有资源的提交，这样唯一可能发生错误的就是基础组件，而非业务处理错误。 采用最大努力一次提交模式的系统假定基础组件出错的可能性非常小，因此能够承受风险获得较高的吞吐量收益。如果业务处理服务也被设计为一个幂等式 （idempotent），发生错误的可能性也很小。（译者注：幂等式是数学和计算机科学特定运算的一个特性，应用初始化以后多次操作其结果都不会再发生改变）

为了帮助大家更好的理解这个模式分析失败结果，我会用消息驱动数据库更新作为例子来加以说明。

本事务中两个资源将被统计、并且计算。消息事务在一个数据库之前开启，然后

逆序结束。成功的顺序或许和本文开始的时候一模一样：

1. 开启消息事务
2. **接受消息**
3. 开启数据库事务
4. **更新数据库**
5. 提交数据库事务
6. 提交消息事务

准确的说，此顺序前四个步骤都不重要，重要的是消息必须在数据库更新之前被接受，并且每个事务必须在对应的资源被调用之前开启，因此合理的顺序应该如下：

1. 开启消息事务
2. 开启数据库事务
3. **接受消息**
4. **更新数据库**
5. 提交数据库事务
6. 提交消息事务

关键点是最后两步很重要，它们必须放在最后按顺序执行。按序之所以重要，其本身就是一个技术问题，不过这个顺序是有业务需 要决定的。这个顺序告诉开发者，其中的一个事务资源是特殊的，这个资源包含了如何在其他资源上运作的指令。这是一个业务顺序:系统不能自动告知走事务到哪一步了。即使消息和数据库是两个资源，事务也常常遵循这一流程。按序之所以重要，是因为事务必须处理失败案例。迄今最常见的失败案例是，诸如坏数据、编程错误等失败的业务处理。本例中，这两个事务很容易用来相应一个异常并且回滚。这样，业务数据的完成性得到保障，时间线与本文开始列出的理想失败案例神似。

引发回滚机制的精确性不是很重要的，这样的机制有好一些。最重要的是，提交或者回滚必须按照资源中业务顺序的逆序发生。在一个应用案例中，消息事务必须在最后提交，因为处理业务的指令包含在这个资源中，这是因为，很少有失败案例 其第一次提交成功，第二次失败。在这个点上，所有设计业务处理的部分都已完成，那唯一能引起部分失败的因素，可能消息中间件的基础问题。

注意如果 数据库资源提交失败，那么事务最终会发生回滚。所以是说非原子性失败模型（failure mode）其第一个事务会提交，第二个事务发生回滚。通常，事务中有n个资源，那么就存在n-1个这样的失败模型，这会导致一个子事务回滚后，其它一些资源处在提交后的不一致状态。在消息数据库用例中，失败模型的结局是，消息会回滚，然后是其他的事务，即使其他这些事务都经成功处理；可

以断定最糟糕的事情 就是重复消息（duplicate message）被传递过来。什么是重复消息呢？通常情况下，事务中的早期资源被认为是包含有后续资源处理流程的讯息，因此失败模型的结果可以被认为就是重复消息。

一 些富有冒险精神的人认为重复消息发生的可能性微乎其微，因此懒得去预测这些消息。然而，为了更加确信业务数据是准确性和一致性，还是需要在业务逻辑层面对 此有清晰地认识。 如果怀疑重复消息可能发生，那么必须核实，业务处理过程是否处理过数据，在处理数据之前是否什么都没做。这个特定的说明有时指幕等业务服务模型 （Idempotent Business Service pattern）。

相关案例包括两个采用此模型的同步事务资源例子，我会在后面做一一分析，以及一些其它选项。

## Spring和消息驱动POJO

案例best-jms-db项目中的代码，开发人员采用主流配置，这样就可以使用最大努力一次提交模式。具体的做法是这样的，通过一个异步的监听器将消息传给一个队列，并将此数据插入 数据库表中。

TransactionAwareConnectionFactoryProxy 是Spring的一个存储控件，应用于这个模式中，也是最关键的组成部分。放弃采用供应商提供的粗颗粒度的ConnectionFactory，configuration采用装饰模式包装了一个ConnectionFactory，用它来处理事务同步问题。 具体配置见jms-context.xml，如下清单6所示：

**清单6、配置一个TransactionAwareConnectionFactoryProxy去包装一个供应商提供的JMS ConnectionFactory**

```
<bean id="connectionFactory">
        <property name="targetConnectionFactory">
            <bean depends-on="brokerService">
                <property name="brokerURL" value="vm
            </bean>
        </property>
        <property name="synchedLocalTransactionAllow
</bean>
```

ConnectionFactory 无需知道哪个事务管理器与其同步，每一时刻仅有一个事务处在活动（active）状态。这些是由Spring内部在处理。事务驱动是由配置在data- source-context.xml中的DataSourceTransactionManager完成的，事务管理器必须由轮询和接受消息的JMS监听器容器监控。

```
<jms:listener-container transaction-manager="transac
    <jms:listener destination="async" ref="fooHandle
</jms:listener-container>
```

fooHandler和方法会告知监听器容器某个具体的控件的具体方法得到调用，当一个消息达到”异步”队列。handler是如此实现的，接受一个String作为参数消息，并将其作为数据插入记录中。

```
1
2
3
4
5
6
```

```
public void handle(String msg) {

  jdbcTemplate.update(
      "INSERT INTO T_FOOS (ID, name, foo_date) value

}
```

为了模拟失败，代码用一个FailureSimulator切面，它会检查消息内容是否真的失败；如清单7所示，在FooHandler在事务结束之前，处理消息之后，maybeFail()方法得到调用，所以它能影响事务的结果。

**清单7、maybeFail()方法**

```
1
2
3
4
5
6
7
8
9
10
```

```
@AfterReturning("execution( ..*Handler+.handle(Stri
public void maybeFail(String msg) {
        if (msg.contains("fail")) {
              if (msg.contains("partial")) {
                    simulateMessageSystemFailure(
              } else {
                simulateBusinessProcessingFailure(
              }
        }
}
```

simulateBusinessProcessingFailure() 方法会抛出DataAccessException，就像数据库访问真的失败。一旦这个方法被调用，最理想的结局是所有的数据库以及消息事务都能回滚。 这个场景在案例项目AsynchronousMessageTriggerAndRollbackTests单元测试得到测试。

simulateMessageSystemFailure() 方法通过破坏底层的JMS Session来模拟消息系统失败。预期的结果是事务部分提交:数据库提交了，但消息回滚。这个在synchronousMessageTriggerAndPartialRollbackTests单元测试验证过。

同样，在AsynchronousMessageTriggerSunnyDayTests类中，包含一个所有事务成功提交的单元测试。

相 同的JMS配置，相同的业务逻辑同样可以用在同步的环境中，消息由存储在业务逻辑中的阻塞请求所接收，而非监听器容器。此方法在best-jms-db案 例项目中得到展示。 sunny-day case以及事务全部回滚分别在SynchronousMessageTriggerSunnyDayTests和SynchronousMessageTriggerAndRollbackTests得到测试。

## 链式事务管理器

在其他的最大努力一阶段提交模式案例中，一个粗糙的事务管理器实现仅仅是将一系列其他的事务管理器链接在一起，去实现事务同步。倘若业务处理成功，所有的事务将会提交， 否则它们都能回滚。

ChainedTransactionManager 接受一系列其他的事务管理器作为注入属性，如清单8所示：

**清单8、配置**

```
<bean id="transactionManager">
        <property name="transactionManagers">
            <list>
                <bean>
                    <property name="dataSource" ref="da
                </bean>
                <bean>
                    <property name="dataSource" ref="
                </bean>
            </list>
        </property>
</bean>
```

此配置简单的测试，仅是同时插入数据到两个数据库，回滚，同时确保两个运行回滚到最初状态。此实现作为存在MulipleDataSourceTests中的一个单元测试，如同XA案例中的 atomikos-db项目。倘若回滚没有同步，有事务提交了，那测试就算失败。

记 住，资源顺序很重要，它们是嵌套的，提交或者回滚以它们参与的相反顺序进行。其中一个事务最为特别：如果存在问题，最重要的事务会回滚，即便是这问题是一个资源失败。 同样，testInsertWithCheckForDuplicates()测试方法展示了幂等式业务处理如何从部分失败中保护系统，此实现作为一个里层资源（otherDataSource）中业务运算防御检测。

```
int count = otherJdbcTemplate.update("UPDATE T_AUDIT
if (count == 0) { count = otherJdbcTemplate.update("
```

update首先尝试和一个where子句执行，不出意外，update中的数据会插入数据库中。本例中的幂等式处理一个额外的花销是sunny-day case中额外的查询，这个额外的花销在复杂的每个事务执行多个查询的业务处理中微乎其微。

## 其他选择

案例中的ChainedTransactionManager拥有简洁优势，而且扩展优化已做的很好。另一个方式是， 当第二个资源加入时，利用Spring的

TransactionSychronization API给当前事务 注册一个回调函数，此方式在best-jms-db案例中，最大的特点是TransactionAwareConnectionFactory和一个DataSourceTransactionManager的结合。利用TransactionSynchronizationManager，这个特殊的案例可以扩展并且泛化到包含non-JMS的资源中。这样理论上有个优势，就是只有加入事务的资源得到支持，而非链上的 所有资源。然而配置依旧需要监听某个潜在的事务与之对应的资源。

同样，Spring工程师团队考虑将最大努力一阶段提交事务管理器特性作为Spring核心。你可以在JJRA issue中投票，如果你喜欢这种模式，希望Spring中显示以及更加透明地支持此种模式。

## 非事务访问模式

非事务访问模式需要一个特殊的业务处理，这样才有意义。理想的状态是有时，其中一些你需要访问的资源边缘化，一点都不需要事务。比如，或许你需要将一行数据插入一个 审核表中，此操作是独立的，和业务事务是否成功无关。仅仅记录试图做了某事。更普遍的场景，人们高估了他们需要对其中一个资源做读写的频次，事实上，只有 访问就很好了。 否则，写操作需要得到很好地控制，因此如果发生任何错误，写操作可以被记录下来或者忽略。

在以上的案例中资源恰当地原理全局事务，但仍然有其自己的本地事务，本地事务无需与其他发生的事情保持同步。如果你使用的是Spring，主要的事务由一个PlatformTransactionManager驱动， 边缘资源或许是一个数据库Connection，它来自一个不受事务管理器控制的DataSource。每一次访问边缘资源需要将缺省环境设为 autoCommit=true。updates对读操作不可见，前者可以与其他非提交事务并发进行，但写操作带来的影响通常对其他操作来说是立即可见的。

这个模式需要更多精细地分析，以及更多自信去涉及业务处理，但它同最大努力一阶段提交没 什么区别。当任何事情出错，一个通用的补偿事务服务对多数项目来说太过庞大。不过简单的用例所涉及的服务，它是幕等式的仅仅执行一个写的操作，这种现象再 普通不过了。这些是非事务策略的理想场景。

## Wing-and-a-Prayer：一种反模式

最后一种模式是一种反模式，它出现这样一个场景中，开发者不理解或者没有意识到他们已经存在一个分布式事务时。无需显示的调用底层的资源事务API，你不确 定所有的资源是否在一个事务中。倘若用的是一个Spring事务管理器而非JtaTransactionManager，此管理器会将一个事务资源加入其中。这个事务管理器将会拦截Spring声明事务管理特性的执行方法，比如@Transactional；其他的

资源不会注册到相同的事务中。通常的结局 是任何事情都运转正常，不过很快用户会发现存在一个异常，其中一个资源没有回滚。一个典型的错误导致的问题是利用一个 DataSourceTransactionManager以及一个利用Hibernate实现的仓库。

## 该用哪个模式呢?

我会通过分析已介绍过的模式其利弊，帮助大家做出取舍。第一步是分析你的系统是否需要分布式事务。一个必须但不充分条件是，存在不止一个事务资源的单一处理。充分条件是这些资源都在一个单独的用例中，通常由系统架构的service层调用来驱动。

如果你不认为这是分布式事务，那最好采用Wing-and-a-Prayer模式，接着你会看见数据应该回滚但没有。或许你会看到这种影响从失败发生直至其下游一直存在，而且很难追溯回去。**Wing-and-a-Prayer**的 使用也可能会开发人员所疏忽，他们认为受到了XA保障，其实并没与配置底层资源加入到事务中。我曾经做过一个这样的项目，数据库是其他人员搭建的，他们在 安装数据库的过程中关闭了XA支持。运行了个把月没有任何问题，接着各种奇怪的失败开始侵入业务处理中，需要花很长的时间去找出问题。

如果是一个包含异质资源的简单用例，你可以分析甚至做一些重构，那么**非事务访问模式**或 许是个不错的选择，特别是其中一个几乎是只读资源，双检测确保写操作。即便是失败了，非事务资源中的数据在业务术语中必须有意义。审核、版本控制、甚至日 志信息能很好的切入到此目录中，失败变得相对很平常—-任何时间真实事务中的任何事情都可回滚，但你需确信这样做不存在负面影响就好。

对系统而言，**最大努力一阶段提交**需要通用的失败保护机制，但有不存在2PC那么大的开销，而且性能得到极大的提升。相对非事务资源，它的建立需要更多的技巧，但无需太多的分析， 通常应用于更加通用的数据类型中。完成数据一致性的某些特性，需要保障业务处理对外层资源（”outer” resources:第一个提交的资源）而言是幕等式。消息驱动的数据库更新就是一个完美的例子，并且在Spring中得到很好的支持。不常见的场景需要一些额外的框架代码，这些代码终究会成为Spring的一部分。

**共享资源模式**是一种特定的例子，通常涉及一个特定的类型和平台两个资源，比如，ActiveMQ和任何一个RDBMS或者OracleAQ与一个Oracle数据库共存。这样做最大的收益是相当的灵活以及出色的性能。

**Full XA with 2PC**是一种通用模式，在应对多个异质资源事务 失败时提供很好的无忧保证。不利的是它的开销很大，需要遵循特定的I/O协议和特定的平台。

有开源的JTA实现，提供了一种摆脱应用服务器的方式，但多数 开发人员依旧认为它们并非最好。可以确信的是，人们花更多的时间去思考系统的事务界限，会更倾向于使用他们并不那么需要的JTA和XA。至少使用 Spring的开发人员，他们的业务逻辑无需知道事务如何被处理的，暂时无需考虑平台选择的问题。