理解 JTS —— 事务简介

**简介：** Java Transaction Service 是 J2EE 架构的关键元素。它与 Java Transaction API 结合在一起，使我们能够构建对于各种系统和网络故障都非常健壮的分布式应用程序。事务是可靠应用程序的基本构建块 —— 如果没有事务的支持，编写可靠的分布式应用程序将是非常困难的。幸运的是，JTS 执行的大部分工作对于程序员都是透明的；J2EE 容器使事务划分和资源征用对程序员来说几乎是不可见的。这个由三个部分组成的系列文章的第一期讲述了一些基础知识，包括什么是事务，以及事务对于构建可靠的分布式应用程序来说至关重要的原因。

如果您阅读过任何有关 J2EE 的介绍性文章或者书籍，那么就会发现，只有一小部分资料是专门针对 Java Transaction Service（JTS）或 Java Transaction API（JTA）的。这并不是因为 JTS 是 J2EE 中不重要的部分或者可选部分 —— 恰恰相反。JTS 受到的关注之所以会比 EJB 技术少，是因为它为应用程序提供的服务非常透明 —— 很多开发人员甚至没有注意到在他们的应用程序中事务在哪里开始和结束。在某种意义上，JTS 的默默无闻恰恰是它的成功：因为它非常有效地隐藏了事务管理的很多细节，因此，我们没有听说过或者谈论过很多关于它的内容。但是，您可能想了解它在幕后都为您执行什么功能。

毫不夸张地说，没有事务就不能编写可靠的分布式应用程序。事务允许采用某种控制方式修改应用程序的持久性状态，以便使应用程序对于各种各样的系统故障（包括系统崩溃、网络故障、电源故障甚至自然灾害）更加健壮。事务是构建容错、高可靠性以及高可用性应用程序所需的基本构建块之一。

**事务的动机**

假设您正在从一个账户向另一个账户进行转账个账户差额由数据库表中的某一行来表示。如果您想从账户 A 转账到账户 B，则可能执行如下这些 SQL 代码：

|  |
| --- |
| SELECT accountBalance INTO aBalance  FROM Accounts WHERE accountId=aId;  IF (aBalance >= transferAmount) THEN  UPDATE Accounts  SET accountBalance = accountBalance - transferAmount  WHERE accountId = aId;  UPDATE Accounts  SET accountBalance = accountBalance + transferAmount  WHERE accountId = bId;  INSERT INTO AccountJournal (accountId, amount)  VALUES (aId, -transferAmount);  INSERT INTO AccountJournal (accountId, amount)  VALUES (bId, transferAmount);  else  FAIL "Insufficient funds in account";  END if |

到目前为止，这段代码看起来非常简单易懂。如果手头的资金充足，则从一个账户中减去资金，并添加到另一个账户中。但是，如果出现系统电源故障或者崩溃，又会发生什么情况呢？表示账户 A 和账户 B 的行可能不会存储在同一个磁盘块中，这意味着要完成转账需要进行多个磁盘 IO。如果在已写入第一个磁盘块之后，在写入第二个磁盘块之前，系统发生故障，又会发生什么情况呢？A 账户中的资金已经划走，但是没有出现在账户 B 中（A 和 B 客户都不会愿意），或者资金将出现在 账户 B 中，但是没有记入账户 A 的借出账中（银行不会愿意）。如果账户已正确更新，而账户日记账没有更新，又会发生什么情况呢？那么账户 A 和账户 B 的每月银行结账单将与它们账户的余额不一致。

不仅不可能同时将多个数据块写入磁盘，而且每当进行修改时马上将每个数据块写入磁盘，也对系统性能有不利影响。将磁盘写入延迟到比较适宜的时间可能会大大改善应用程序的吞吐量，但是，需要采用不损害数据完整性的方式执行。

甚至在系统没有发生故障时，上面讨论的代码还有另一种风险 —— 并发性。如果账户 A 中有 100 美元，但是却同时开始向它的两个不同的账户分别转账 100 美元，那么会发生什么情况呢？如果时间上凑巧，并且没有适当的锁定机制，两次转账都可能成功，从而使账户 A 的余额为负值。

这些情况似乎都是非常可能发生的，因此希望企业数据系统能够解决这些问题是理所应当的。我们希望在发生火灾、洪水、电源故障、磁盘以及系统出现故障时，银行都能够保持正确的账户记录。可以通过冗余（冗余的磁盘、计算机以及数据中心）来提供容错，但是*事务* 使得构建容错的软件应用程序成为可能。事务提供了一个框架，用于在系统或组件发生故障时保持数据一致性和完整性。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp0305.html#ibm-pcon)

**什么是事务？**

那么到底什么是事务呢？在定义这个术语之前，我们首先定义*应用程序状态* 的概念。应用程序的状态包含影响应用程序操作的所有内存和磁盘中的数据项目 —— 应用程序 “知道” 的所有内容。应用程序状态可以存储在内存、文件或者数据库中。如果系统发生故障，例如应用程序、网络或者计算机系统崩溃，则我们想确保当重新启动系统时，可以恢复应用程序的状态。

现在，我们将*事务* 定义为对应用程序状态的相关操作的集合。事务具有*原子性*、*一致性*、*隔离性* 以及*持久性* 这几个属性。这些属性统称为 *ACID* 属性。

*原子性* 意味着要么所有事务操作都应用于应用程序状态，要么都不应用；事务是不可拆分的工作单元。

*一致性* 意味着事务代表应用程序状态的正确转换 —— 即事务不能违反应用程序中固有的任何完整性限制。实际上，一致性的概念是特定于应用程序的。例如，在记账应用程序中，一致性可能包括所有资产账户的总和始终等于所有负债账户的总和这个不变式。在本系列的第 3 部分中讨论事务划分时，我们将详细讨论这个需求。

*隔离性* 意味着一个事务的效果不影响正在同时执行的其他事务。从事务的角度讲，它意味着事务按顺序执行而不是并行执行。在数据库系统中，通常通过使用锁机制来实现隔离性。为了使应用程序获得最佳性能，有时也会对某些事务放松隔离性的要求。

*持久性* 意味着一旦成功完成某个事务，对应用程序状态所做的更改将 “经得起失败”。

什么是 “经得起失败”呢？它由什么组成？这取决于系统，一个设计良好的系统将明确地标识可以从哪些故障中恢复过来。在我的桌面工作站上运行的事务数据库，对于系统崩溃和电源故障非常稳定健壮，但是对于我的办公大楼发生大火灾却没有任何作用。银行可能不仅仅在数据中心具有冗余的磁盘、网络以及系统，而且还可能在别的城市有冗余的数据中心，该冗余数据中心通过冗余的通信链路连接，目的是允许从严重的故障（如自然灾害）中进行恢复。军用的数据系统甚至可能有更严格的容错要求。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp0305.html#ibm-pcon)

**事务的剖析**

典型事务有几个参与者 —— 应用程序、事务监视器（TPM）以及一个或多个资源管理器（RM）。资源管理器存储应用程序状态，常常是数据库，但也可能是消息队列服务器（在 J2EE 应用程序中，它们将是 JMS 提供者）或其他事务性资源。TPM 协调 RM 的活动，以确保事务 “要么全有要么全无” 属性。

当应用程序请求容器或事务监视器启动新的事务时，事务开始。由于应用程序访问各种各样的 RM，因此，在事务中对它们进行*征用*。RM 必须使对应用程序状态所做的任何更改与请求更改的事务相关联。

当发生以下事件之一或者两个事件都发生时，事务结束：事务应用程序*提交* 该事务；通过应用程序或者由于其中一个 RM 失败，*回滚* 该事务。如果事务成功提交，则将写入与该事务相关联的更改，以使更改持久化并使其对于新的事务可见。如果事务被回滚，则该事务所做的所有更改都将被丢弃；就好像该事务从来没有发生过一样。

**事务日志 —— 持久性的关键**

事务 RM 通过在一个事务日志中记录多个事务的结果，获得持久性以及可接受的性能。事务日志存储为连续的磁盘文件（有时存储在原始分区中），并且一般只是用于写入而不用于读取，回滚或恢复的情况例外。在我们的银行账户示例中，与账户 A 和账户 B 相关联的余额将在内存中进行更新，新的余额和旧的余额将被写入到事务日志中。编写事务日志的更新记录不需要将全部数据都写入磁盘（只需要写入已更改的数据，而不需要写入全部磁盘块），而且所需的磁盘寻道时间也会更少（原因是所有更改都包含在日志中连续的磁盘块中）。此外，与多个并发事务关联的更改可以合并到一起，一次写入事务日志，这意味着每次磁盘写入时我们可以处理多个事务，而不需要 每个事务进行几次磁盘写入。之后，RM 将根据所更改的数据更新实际的磁盘块。

**重新启动时进行恢复**

如果系统出现故障，重新启动时要做的第一件事就是重新应用所有已提交事务的作用，所有这些已提交的事务都 位于日志中，但是它们的数据块尚未更新。采用这种方式，日志保证了故障之间的持久性，而且还能够减少所执行的磁盘 IO 操作的数量，或者至少使它们延迟到对系统性能影响更小的时间。

**两阶段提交**

很多事务只涉及一个 RM —— 通常是数据库。在这种情况下，RM 通常执行提交或回滚事务所需的大部分工作。（几乎所有事务 RM 都有它们自己的内置的事务管理器，这个管理器可以处理*本地事务* —— 只涉及该 RM 的事务）。但是，如果事务涉及两个 RM 或多个 RM —— 可能是两个单独的数据库，或者是一个数据库和一个 JMS 队列，或者是两个单独的 JMS 提供者 —— 我们想确保 “要么全有要么全无” 的语义不仅仅应用于这个 RM 中，而且还应用于事务中的所有 RM。在这种情况下，TPM 将组织一个*两阶段提交*。在两阶段提交中，TPM 首先向每个 RM 发送一个 “准备” 消息，询问它是否准备就绪以及是否能够提交事务；如果它收到来自所有 RM 的确认应答，则将事务在其自己的事务日志中标记为已提交，然后指示所有 RM 提交事务。如果某个 RM 失败，则重新启动时它将向 TPM 询问有关失败时未处理的所有事务的状态，并提交它们或者对它们执行回滚操作。

两个阶段提交类似于社会上的结婚典礼 —— 牧师或神父询问双方 “您愿意让这个男人/女人作为您的丈夫/妻子吗？” 如果双方都回答是，则将宣布他们成为夫妻；否则，双方不能结婚。不管双方中的哪一方首先说 “我愿意”，在一方没有回答时，另一方决不能完成结婚。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp0305.html#ibm-pcon)

**事务作为处理异常的机制**

您可能观察到，事务向对块进行同步的应用程序数据提供很多与内存中数据相同的功能 —— 保证原子性、更改的可见性以及显而易见的排序。但是，当同步主要是并发控制的机制时，则事务主要是处理异常的机制。如果在一个磁盘不会发生故障、系统和软件不会崩溃以及电源是百分百可靠的世界中，我们将不需要事务。事务在企业应用程序中所起的作用与合同法在社会上所起的作用一样 —— 它们规定，如果一方不能履行他那一部分合同，则交易将失效。当我们编写合同时，我们通常希望它是多余的，令人感到欣慰的是大部分时候都是如此。

与比较简单的 Java 程序进行类比，事务在应用程序级别所提供的一些优势与 catch 和 finally 块在方法级别所提供的优势相同；它们使我们不用编写很多错误复原代码，即可执行可靠的错误复原。考虑下面这个方法，该方法将一个文件复制到另一个文件：

|  |
| --- |
| public boolean copyFile(String inFile, String outFile) {  InputStream is = null;  OutputStream os = null;  byte[] buffer;  boolean success = true;  try {  is = new FileInputStream(inFile);  os = new FileOutputStream(outFile);  buffer = new byte[is.available()];  is.read(buffer);  os.write(buffer);  }  catch {IOException e) {  success = false;  }  catch (OutOfMemoryError e) {  success = false;  }  finally {  if (is != null)  is.close();  if (os != null)  os.close();  }  return success;  } |

忽略为整个文件分配一个缓冲区是一个不好的想法，但是在这个方法中哪里错了呢？有很多东西。输入文件可能不存在，或者该用户可能没有这个文件的读权限。用户可能没有输出文件的写权限，或者该文件被另一个用户锁定。可能没有足够的磁盘空间来完成该文件的写操作，或者由于没有足够的内存可用，分配缓冲区可能失败。幸运的是，所有这些都由 finally 语句来处理，该语句释放了 copyFile() 所使用的所有资源。

如果您使用原来的 C 语言编写这个方法，则对于每个操作（打开输入、打开输出、malloc、读、写），必须测试返回状态，如果操作失败，则取消以前成功的所有操作，并返回适当的状态代码。由于需要这么多错误处理代码，该代码可能更大，因此更难阅读。同时在错误处理代码（它也是最难测试的部分）中很容易出错，比如不能释放资源、对一个资源释放两次或者释放尚未分配的资源。更复杂的方法可能涉及更多资源，而不仅仅是两个文件和一个缓冲区，这使得问题变得更加复杂。在大量错误复原代码中，很难发现实际的程序逻辑。

现在，假设您正在执行一个复杂的操作，该操作涉及在多个数据库中插入或更新多个行，其中一个操作违反了完整性约束并失败了。如果您管理自己的错误复原，则必须跟踪已经执行的操作，并知道在随后的操作失败的情况下如何取消每个操作。如果工作单元分布在多个方法或组件上，则会更加困难。借助事务来构造应用程序，就可以将所有这些清理工作委托给数据库（即进行 ROLLBACK），并取消自从事务开始所执行的所有操作。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp0305.html#ibm-pcon)

**结束语**

通过借助事务构造应用程序，我们定义一组正确的应用程序状态转换，并确保应用程序始终处于正确的状态，甚至在系统或组件发生故障之后也是如此。事务使我们能够将很多异常处理和恢复工作委托给 TPM 和 RM，从而简化了我们的代码，并使我们能够空出更多时间来考虑应用程序逻辑。

在此系列的第 2 部分中，我们将探讨这对于 J2EE 应用程序意味着什么 —— J2EE 如何使我们能够将事务语义告知 J2EE 组件（EJB 组件、servlet 以及 JSP 页面）；它如何使资源征用对应用程序（甚至对于 bean 管理的事务）完全透明；单个事务如何 透明地遵循从一个 EJB 组件到另一个 EJB 组件，或者从一个 servlet 到一个 EJB 组件，甚至跨越多个系统的控制流程。

尽管 J2EE 提供了相当透明的对象事务服务，但是应用程序设计者仍然必须仔细考虑在哪里划分事务，以及如何在应用程序中使用事务资源 —— 不正确的事务划分可能会使应用程序处于不一致的状态，而不正确地使用事务资源可能会造成非常严重的性能问题。在此系列的第 3 部分中，我们将讨论这些问题并提供一些关于如何构造应用程序的建议。

在这个系列的 [第 1 部分](http://www.ibm.com/developerworks/library/j-jtp0305.html?S_TACT=105AGX52&S_CMP=cn-a-j)，我们讨论了事务并研究了它们的基本属性 ― 原子性（atomicity）、一致性（consistency）、孤立性（isolation）和持久性（durability）。事务是企业应用程序的基本构件；没有它们，几乎不可能构建有容错能力的企业应用程序。幸运的是，Java 事务服务（Java Transaction Service，JTS）和 J2EE 容器自动为您做了大量的事务管理工作，这样您就不必将事务意识直接集成到组件代码中。结果简直是一种魔术 ― 通过遵守几条简单的规则，J2EE 应用程序就可以自动获得事务性语义，只需极少或根本不需要额外的组件代码。本文旨在通过展示事务管理如何发生，以及发生在何处来揭开这个魔术的神秘面纱。

**什么是 JTS？**

JTS 是一个 *组件事务监视器（component transaction monitor）*。这是什么意思？在第 1 部分，我们介绍了 *事务处理监视器*（TPM）这个概念，TPM 是一个程序，它代表应用程序协调分布式事务的执行。TPM 与数据库出现的时间长短差不多；在 60 年代后期，IBM 首先开发了 CICS，至今人们仍在使用。经典的（或者说 *程序化*）TPM 管理被程序化定义为针对事务性资源（比如数据库）的操作序列的事务。随着分布式对象协议，如 CORBA、DCOM 和 RMI 的出现，人们希望看到事务更面向对象的前景。将事务性语义告知面向对象的组件要求对 TPM 模型进行扩展 ― 在这个模型中事务是按照事务性对象的调用方法定义的。JTS 只是一个组件事务监视器（有时也称为 *对象事务监视器（object transaction monitor）*），或称为 CTM。

JTS 和 J2EE 的事务支持设计受 CORBA 对象事务服务（CORBA Object Transaction Service，OTS）的影响很大。实际上，JTS 实现 OTS 并充当 Java 事务 API（Java Transaction API）― 一种用来定义事务边界的低级 API ― 和 OTS 之间的接口。使用 OTS 代替创建一个新对象事务协议遵循了现有标准，并使 J2EE 和 CORBA 能够互相兼容。

乍一看，从程序化事务监视器到 CTM 的转变好像只是术语名称改变了一下。然而，差别不止这一点。当 CTM 中的事务提交或回滚时，与事务相关的对象所做的全部更改都一起被提交或取消。但 CTM 怎么知道对象在事务期间做了什么事？象 EJB 组件之类的事务性组件并没有 commit() 或 rollback() 方法，它们也没向事务监视器注册自己做了什么事。那么 J2EE 组件执行的操作如何变成事务的一部分呢？

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp/part2/index.html#ibm-pcon)

**透明的资源征用**

当应用程序状态被组件操纵时，它仍然存储在事务性资源管理器（例如，数据库和消息队列服务器）中，这些事务性资源管理器可以注册为分布式事务中的资源管理器。在第 1 部分中，我们讨论了如何在单个事务中征用多个资源管理器，事务管理器如何协调这些资源管理器。资源管理器知道如何把应用程序状态中的变化与特定的事务关联起来。

但这只是把问题的焦点从组件转移到了资源管理器 ― 容器如何断定什么资源与该事务有关，可以供它征用？请考虑下面的代码，在典型的 EJB 会话 bean 中您可能会发现这样的代码：

**清单 1. bean 管理的事务的透明资源征用**

|  |
| --- |
| InitialContext ic = new InitialContext();  UserTransaction ut = ejbContext.getUserTransaction();  ut.begin();  DataSource db1 = (DataSource) ic.lookup("java:comp/env/OrdersDB");  DataSource db2 = (DataSource) ic.lookup("java:comp/env/InventoryDB");  Connection con1 = db1.getConnection();  Connection con2 = db2.getConnection();  // perform updates to OrdersDB using connection con1  // perform updates to InventoryDB using connection con2  ut.commit(); |

注意，这个示例中没有征用当前事务中 JDBC 连接的代码 ― 容器会为我们完成这个任务。我们来看一下它是如何发生的。

**资源管理器的三种类型**

当一个 EJB 组件想访问数据库、消息队列服务器或者其它一些事务性资源时，它需要到资源管理器的连接（通常是使用 JNDI）。而且，J2EE 规范只认可三种类型的事务性资源 ― JDBC 数据库、JMS 消息队列服务器和“其它通过 JCA 访问的事务性服务”。后面一种服务（比如 ERP 系统）必须通过 JCA（J2EE Connector Architecture，J2EE 连接器体系结构）访问。对于这些类型资源中的每一种，容器或提供者都会帮我们把资源征调到事务中。

在清单 1 中， con1 和 con2 好象是普通的 JDBC 连接，比如那些从 DriverManager.getConnection() 返回的连接。我们从一个 JDBC DataSource 得到这些连接，JDBC DataSource 可以通过查找 JNDI 中的数据源名称得到。EJB 组件中被用来查找数据源（ java:comp/env/OrdersDB ）的名称是特定于组件的；组件的部署描述符的 resource-ref 部分将其映射为容器管理的一些应用程序级 DataSource 的 JNDI 名称。

**隐藏的 JDBC 驱动器**

每个 J2EE 容器都可以创建有事务意识的池态 DataSource 对象，但 J2EE 规范并不向您展示如何创建，因为这不在 J2EE 规范内。浏览 J2EE 文档时，您找不到任何关于如何创建 JDBC 数据源的内容。相反，您不得不为您的容器查阅该文档。创建一个数据源可能需要向属性或配置文件添加一个数据源定义，或者也可以通过 GUI 管理工具完成，这取决于您的容器。

每个容器（或连接池管理器，如 PoolMan）都提供它自己的创建 DataSource 机制，JTA 魔术就隐藏在这个机制中。连接池管理器从指定的 JDBC 驱动器得到一个 Connection ，但在将它返回到应用程序之前，将它与一个也实现 Connection 的虚包包在一起，将自己置入应用程序和底层连接之间。当创建连接或者执行 JDBC 操作时，包装器询问事务管理器当前线程是不是正在事务的上下文中执行，如果事务中有 Connection 的话，就自动征用它。

其它类型的事务性资源，JMS 消息队列和 JCA 连接器，依靠相似的机制将资源征用隐藏起来，使用户看不到。如果要使 JMS 队列在部署时对 J2EE 应用程序可用，您就要再次使用特定于提供者的机制来创建受管 JMS 对象（队列连接工厂和目标），然后在 JNDI 名称空间内发布这些对象。提供者创建的受管对象包含与 JDBC 包装器（由容器提供的连接池管理器添加）相似的自动征用代码。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp/part2/index.html#ibm-pcon)

**透明的事务控制**

两种类型的 J2EE 事务 ― 容器管理的和 bean 管理的 ― 在如何启动和结束事务上是不同的。事务启动和结束的地方被称为 *事务划分（transaction demarcation）*。清单 1 中的示例代码演示了 bean 管理的事务（有时也称为 *编程（programmatic）*事务）。Bean 管理的事务是由组件使用 UserTransaction 类显式启动和结束的。通过 ejbContext 使UserTransaction 对 EJB 组件可用，通过 JNDI 使其对其它 J2EE 组件可用。

容器根据组件的部署描述符中的事务属性代表应用程序透明地启动和结束容器管理的事务（或称为 *宣告式事务（declarative transaction）*）。通过将 transaction-type 属性设置为 Container 或 Bean 您可以指出 EJB 组件是使用 bean 管理的事务性支持还是容器管理的事务性支持。

使用容器管理的事务，您可以在 EJB 类或方法级别上指定事务性属性；您可以为 EJB 类指定缺省的事务性属性，如果不同的方法会有不同的事务性语义，您还可以为每个方法指定属性。这些事务性属性在装配描述符（assembly descriptor）的 container-transaction 部分被指定。 清单 2 显示了一个装配描述符示例。 trans-attribute 的受支持的值有：

* Supports
* Required
* RequiresNew
* Mandatory
* NotSupported
* Never

trans-attribute 决定方法是否支持在事务内部执行、当在事务内部调用方法时容器会执行什么操作以及在事务外部调用方法时容器会执行什么操作。最常用的 容器管理的事务属性是 Required 。如果设置了 Required ，过程中的事务将在该事务中征用您的 bean，但如果没有正在运行的事务，容器将为您启动一个。在这个系列的第 3 部分，当您可能想使用每个事务属性时，我们将研究各个事务属性之间的区别。

**清单 2. EJB 装配描述符样本**

|  |
| --- |
| <assembly-descriptor>  ...  <container-transaction>  <method>  <ejb-name>MyBean</ejb-name>  <method-name>\*</method-name>  </method>  <trans-attribute>Required</trans-attribute>  </container-transaction>  <container-transaction>  <method>  <ejb-name>MyBean</ejb-name>  <method-name>updateName</method-name>  </method>  <trans-attribute>RequiresNew</trans-attribute>  </container-transaction>  ...  </assembly-descriptor> |

**功能强大，但很危险**

与清单 1 中的示例不同，由于有宣告式事务划分，这段组件代码中没有事务管理代码。这不仅使结果组件代码更加易读（因为它不与事务管理代码混在一起），而且它还有另一个更重要的优点 ― 不必修改，甚至不必访问组件的源代码，就可以在应用程序装配时改变组件的事务性语义。

尽管能够指定与代码分开的事务划分是一种非常强大的功能，但在装配时做出不好的决定会使应用程序变得不稳定，或者严重影响它的性能。对容器管理的事务进行正确分界的责任由组件开发者和应用程序装配人员共同担当。组件开发者需要提供足够的文档说明组件是做什么的，这样应用程序部署者就能够明智地决定如何构建应用程序的事务。应用程序装配人员需要理解应用程序中的组件是怎样相互作用的，这样就可以用一种既强制应用程序保持一致又不削弱性能的方法对事务进行分界。在这个系列的第 3 部分中我们将讨论这些问题。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp/part2/index.html#ibm-pcon)

**透明的事务传播**

在任何类型的事务中，资源征用都是透明的；容器自动将事务过程中使用的任意事务性资源征调到当前事务中。这个过程不仅扩展到事务性方法使用的资源（比如在清单 1 中获得的数据库连接），还扩展到它调用的方法（甚至远程方法）使用的资源。我们来看一下这是如何发生的。

**容器用线程与事务相关联**

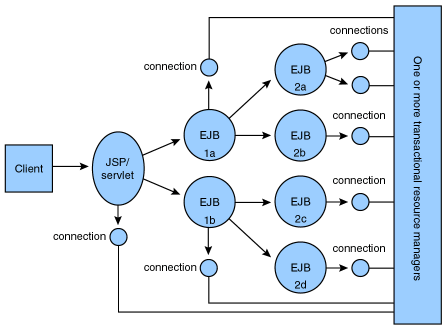
我们假设对象 A 的 methodA() 启动一个事务，然后调用对象 B 的 methodB() （对象 B 将得到一个 JDBC 连接并更新数据库）。 B 获得的连接将被自动征调到 A 创建的事务中。容器怎么知道要做这件事？

当事务启动时，事务上下文与执行线程关联在一起。当 A 创建事务时， A 在其中执行的线程与该事务关联在一起。由于本地方法调用与主调程序（caller）在同一个线程内执行，所以 A 调用的每个方法也都在该事务的上下文中。

**橱中骸骨**

如果对象 B 其实是在另一个线程，甚至另一个 JVM 中执行的 EJB 组件的存根，情况会怎样？令人吃惊的是，远程对象 B访问的资源仍将在当前事务中被征用。EJB 对象存根（在主调程序的上下文中执行的那部分）、EJB 协议（IIOP 上的 RMI）和远端的骨架对象协力要使其透明地发生。存根确定调用者是不是正在执行一个事务。如果是，事务标识，或者说 *Xid*，被作为 IIOP 调用的一部分与方法参数一起传播到远程对象。（IIOP 是 CORBA 远程-调用协议，它为传播执行上下文（比如事务上下文和安全性上下文）的各种元素而备；关于 RMI over IIOP 的更多信息，请参阅 [参考资料](http://www.ibm.com/developerworks/cn/java/j-jtp/part2/index.html#resources)。）如果调用是事务的一部分，那么远程系统上的骨架对象自动设置远程线程的事务上下文，这样，当调用实际的远程方法时，它已经是事务的一部分了。（存根和骨架对象还负责开始和提交容器管理的事务。）

事务可以由任何 J2EE 组件来启动 ― 一个 EJB 组件、一个 servlet 或者一个 JSP 页面（如果容器支持的话，还可以是一个应用程序客户机）。这意味着，应用程序可以在请求到达时在 servlet 或者 JSP 页面中启动事务、在 servlet 或者 JSP 页面中执行一些处理、作为页面逻辑的一部分访问多个服务器上的实体 bean 和会话 bean 并使所有这些工作透明地成为一个事务的一部分。图 1 演示了事务上下文怎样遵守从 servlet 到 EJB，再到 EJB 的执行路径。

**图 1.单个事务中的多个组件**  


[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp/part2/index.html#ibm-pcon)

**最优化**

让容器来管理事务允许容器为我们做出某些最优化决定。在图 1 中，我们看到一个 servlet 和多个 EJB 组件在单个事务的上下文中访问一个数据库。每个组件都获得到数据库的 Connection ；很可能每个组件都在访问同一个数据库。即使多个连接是从不同的组件到同一个资源，JTS 也可以检测出多个资源是否和事务有关，并最优化该事务的执行。您可以从第 1 部分回忆起来，单个事务要包含多个资源管理器需要使用两阶段提交协议，这比单个资源管理器使用的单阶段提交代价要高。JTS 能够确定事务中是不是只征用了一个资源管理器。如果它检测出所有与事务相关的资源都一样，它可以跳过两阶段提交并让资源管理器自己来处理事务。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp/part2/index.html#ibm-pcon)

**结束语**

这个虑及透明事务控制、资源征用和透明传播的魔术不是 JTS 的一部分，而是 J2EE 容器如何在幕后代表 J2EE 应用程序使用 JTA 和 JTS 服务的一部分。在幕后有许多实体合力使这个魔术透明地发生；EJB 存根和骨架、容器厂商提供的 JDBC 驱动器包装器、数据库厂商提供的 JDBC 驱动器、JMS 提供器和 JCA 连接器。所有这些实体都与事务管理器进行交互，于是应用程序代码就不必与之交互了。

在第 3 部分，我们将看一下关于管理 J2EE 上下文中事务的一些实际问题 ― 事务划分和孤立 ― 以及它们对应用程序一致性、稳定性和性能的影响。

在本系列的第 1 部分（“ [An introduction to transactions](http://www.ibm.com/developerworks/java/library/j-jtp0305.html?S_TACT=105AGX52&S_CMP=cn-a-j)”）和第 2 部分（“ [The magic beind the scenes](http://www.ibm.com/developerworks/java/library/j-jtp0410/?S_TACT=105AGX52&S_CMP=cn-a-j)”）中，我们定义了什么是事务，列举了事务的基本特性（property），并探讨了 Java 事务服务（Java Transaction Service）和 J2EE 容器如何合作为事务提供对 J2EE 组件的透明支持。在本文中，我们将讨论事务的划分和隔离这个主题。

为 EJB 组件定义事务划分和隔离属性（attribute）的职责由应用程序装配人员来承担。如果这些属性设置不当，会对应用程序的性能、可伸缩性或容错能力造成严重的后果。不幸的是，并没有一种必须遵守的规则用于正确设置这些属性，但有一些指导可以帮助我们在并发危险和性能危险之间找到一种平衡。

我们在第 1 部分中讨论过，事务主要是一种异常处理机制。事务在程序中的用途与合法合同在日常业务中的用途相似：如果出了什么问题它们可以帮助恢复。但由于大多数时间内都没实际 *发生*什么错误，我们就希望能够尽量减少它们的开销以及对其余时间的占用。我们在应用程序中如何使用事务会对应用程序的性能和可伸缩性产生很大的影响。

**事务划分**

J2EE 容器提供了两种机制用来定义事务的起点和终点：bean 管理的事务和容器管理的事务。在 bean 管理的事务中，用UserTransaction.begin() 和 UserTransaction.commit() 在 bean 方法中显式开始和结束一个事务。另一方面，容器管理的事务提供了更多的灵活性。通过在装配描述符中为每个 EJB 方法定义事务性属性，您可以指定每个方法的事务性需求并让容器确定何时开始和结束一个事务。无论在哪种情况下，构建事务的基本指导方针都是一样的。

**进来，出去**

事务划分的第一条规则是“尽量短小”。事务提供并发控制；这通常意味着资源管理器将代表您获得您在事务期间访问的数据项的锁，并且它必须一直持有这些锁，直到事务结束。（请回忆一下本系列第 1 部分所讨论的 *ACID*特性，其中“ACID”的“I”代表“隔离”（Isolation）。也就是说，一个事务的结果影响不到与该事务并发执行的其它事务。）当您拥有锁时，任何需要访问您锁定的数据项的其它事务将不得不一直等待，直到您释放锁。如果您的事务很长，那些其它的所有事务都将被锁定，您的应用程序吞吐量将大幅度下降。

**规则 1：使事务尽可能短小。**

通过使事务尽量短小，您可以把阻碍其它事务的时间缩到最短，从而提高应用程序的可伸缩性。保持事务尽可能短小的最好方法当然是不在事务中间做任何不必要耗费时间的事，特别是不要在事务中间等待用户输入。

开始一个事务，从数据库检索一些数据，显示数据，然后在仍处于事务中时请用户做出一个选择可能比较诱人。千万别这么做！即使用户注意力集中，也要花费数秒来响应 ― 而在数据库中拥有锁数秒的时间已经是很长的了。如果用户决定离开计算机，或许是去吃午餐或者甚至回家一天，会发生什么情况？应用程序将只好无奈停机。在事务期间执行 I/O 是导致灾难的秘诀。

**规则 2：在事务期间不要等待用户输入。**

**将相关的操作归在一起**

由于每个事务都有不小的开销，您可能认为最好是在单个事务中执行尽可能多的操作以使每个操作的开销达到最小。但规则 1 告诉我们长事务对可伸缩性不利。那么如何实现最小化每个操作的开销和可伸缩性之间的平衡呢？

我们把规则 1 设置为逻辑上的极端 ― 每个事务一个操作 ― 这样不仅会导致额外开销，还会危及应用程序状态的一致性。假定事务性资源管理器维护应用程序状态的一致性（请回忆一下第 1 部分，其中“ACID”的“C”代表“一致性”（Consistency）），但它们依赖应用程序来定义一致性的意思。实际上，我们在描述事务时使用的一致性的定义有点圆滑：应用程序说一致性是什么意思它就是什么意思。应用程序把几组应用程序状态的变化组织到几个事务中，结果应用程序的状态就成了 *定义上的（by definition）*一致。然后资源管理器确保如果它必须从故障恢复的话，就把应用程序状态恢复到最近的一致状态。

在第 1 部分中，我们给出了一个在银行应用程序中将资金从一个帐户转移到另一个帐户的示例。清单 1 展示了这个示例可能的 SQL 实现，它包含 5 个 SQL 操作（一个选择，两个更新和两个插入操作）：

**清单 1. 资金转移的样本 SQL 代码**

|  |
| --- |
| SELECT accountBalance INTO aBalance  FROM Accounts WHERE accountId=aId;  IF (aBalance >= transferAmount) THEN  UPDATE Accounts  SET accountBalance = accountBalance - transferAmount  WHERE accountId = aId;  UPDATE Accounts  SET accountBalance = accountBalance + transferAmount  WHERE accountId = bId;  INSERT INTO AccountJournal (accountId, amount)  VALUES (aId, -transferAmount);  INSERT INTO AccountJournal (accountId, amount)  VALUES (bId, transferAmount);  ELSE  FAIL "Insufficient funds in account";  END IF |

如果我们把这个操作作为五个单独的事务来执行会发生什么情况？这样不仅会使执行速度变慢（由于事务开销），还会失去一致性。例如，如果一个人从帐户 A 取了钱，作为执行第一次 SELECT（检查余额）和随后的记入借方 UPDATE 之间的一个单独事务的一部分，会发生什么情况？这样会违反我们认为这段代码会强制遵守的业务规则 ― 帐户余额应该是非负的。如果在第一次 UPDATE 和第二次 UPDATE 之间系统失败会发生什么情况？现在，当系统恢复时，钱已经离开了帐户 A 但还没有记入帐户 B 的贷方，并且也无记录说明原因。这样，哪个帐户的所有者都不会开心。

清单 1 中的五个 SQL 操作是单个相关操作 ― 将资金从一个帐户转移到另一个帐户 ― 的一部分。因此，我们希望要么全部执行它们，要么一个也不执行，建议在单个事务中全部执行它们。

**规则 3：将相关操作归到单个事务中。**

**理想化的平衡**

规则 1 说事务应尽可能短小。清单 1 中的示例表明有时候我们必须把一些操作归到一个事务中来维护一致性。当然，它要依赖应用程序来确定“相关操作”是由什么组成的。我们可以把规则 1 和 3 结合在一起，提供一个描述事务范围的一般指导，我们规定它为规则 4：

**规则 4：把相关操作归到单个事务中，但把不相关的操作放到单独的事务中。**

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp0514/#ibm-pcon)

**容器管理的事务**

在使用容器管理的事务时，不是显式声明事务的起点和终点，而是为每个 EJB 方法定义事务性需求。bean 的 assembly-descriptor 的 container-transaction 部分的 trans-attribute 元素中定义了事务模式。（清单 2 中显示了一个 assembly-descriptor 示例。）方法的事务模式以及状态 ― 调用方法是否早已在事务中被征用 ― 决定了当 EJB 方法被调用时容器应该进行下面几个操作中的哪一个：

* 征用现有事务中的方法。
* 创建一个新事务，并征用该事务中的方法。
* 不征用任何事务中的方法。
* 抛出一个异常。

**清单 2. 样本 EJB 装配描述符**

|  |
| --- |
| <assembly-descriptor>  ...  <container-transaction>  <method>  <ejb-name>MyBean</ejb-name>  <method-name>\*</method-name>  </method>  <trans-attribute>Required</trans-attribute>  </container-transaction>  <container-transaction>  <method>  <ejb-name>MyBean</ejb-name>  <method-name>logError</method-name>  </method>  <trans-attribute>RequiresNew</trans-attribute>  </container-transaction>  ...  </assembly-descriptor> |

J2EE 规范定义了六种事务模式： Required 、 RequiresNew 、 Mandatory 、 Supports 、 NotSupported 和 Never 。表 1 概述了每种模式的行为 ― 在现有事务中被调用和不在事务内调用时的行为 ― 并描述了每种模式受哪些类型的 EJB 组件支持。（一些容器可能允许您在选择事务模式时有更多的灵活性，但这种使用要依赖特定于容器的功能，因此不适合跨容器的情况）。

**表 1. 事务模式**

|  |  |  |  |
| --- | --- | --- | --- |
| **事务模式** | **Bean 类型** | **在事务 T 内被调用时的行为** | **在事务外被调用时的行为** |
| Required | 会话、实体、消息驱动 | 在 T 中征用 | 新建事务 |
| RequiresNew | 会话、实体 | 新建事务 | 新建事务 |
| Supports | 会话、消息驱动 | 在 T 中征用 | 不带事务运行 |
| Mandatory | 会话、实体 | 在 T 中征用 | 出错 |
| NotSupported | 会话、消息驱动 | 不带事务运行 | 不带事务运行 |
| Never | 会话、消息驱动 | 出错 | 不带事务运行 |

在只使用容器管理的事务的应用程序中，只有组件调用事务模式为 Required 或 RequiresNew 的 EJB 方法时才启动事务。如果容器创建一个事务作为调用事务性方法的结果，当该方法完成时将关闭该事务。如果方法正常返回，容器将提交事务（除非应用程序已经要求回滚事务）。如果方法通过抛出一个异常退出，容器将回滚事务并传播该异常。如果在现有事务 T 中调用了一个方法，并且事务模式指定应该不带事务运行该方法或者在新事务中运行该方法，那么事务 T 将被暂挂，一直到方法完成，然后先前的事务 T 被恢复。

**选择一种事务模式**

那么我们应该为自己的 bean 方法选择哪种模式呢？对于会话 bean 和消息驱动 bean，您通常想使用 Required 来确保每个调用都被作为事务的一部分执行，但仍将允许方法作为一个更大的事务的组件。请小心使用 RequiresNew ；只有在确定自己的方法的行为应该与调用您的方法的行为分开提交时，才应该使用这种模式。 RequiresNew 一般情况下只和与系统中其它对象关系很少或没什么关系的对象（比如日志对象）一起使用。（把 RequiresNew 与日志对象一起使用比较有意义，因为您可能希望在不管外围事务是否提交的情况下提交日志消息。）

RequiresNew 使用不当会导致与上面的描述相似的情况，其中，清单 1 中的代码在五个分开的事务而不是一个事务中执行，这样会使应用程序处于不一致状态。

对于 CMP（容器管理的持久性，container-managed persistence）实体 bean，通常是希望使用 Required 。 Mandatory 也是一个合理的选项，特别是在最初开发时；这将会警告您实体 bean 方法在事务外被调用这种情况，这时可能会指出一个部署错误。您几乎从不希望把 RequiresNew 和 CMP 实体 bean 一起使用。 NotSupported 和 Never 旨在用于非事务性资源，比如 Java 事务 API（Java Transaction API，JTA）事务中无法征用的外部非事务性系统或事务性系统的适配器。

如果 EJB 应用程序设计得当，应用上面的事务模式指导往往会自然地产生规则 4 建议的事务划分。原因是 J2EE 体系架构鼓励把应用程序分解为最小的方便处理的块，并且每个块都作为一个单独的请求被处理（ 不管是以 HTTP 请求的形式还是作为在 JMS 队列中排队的消息的结果）。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp0514/#ibm-pcon)

**重温隔离**

在第 1 部分中，我们定义了 *隔离（isolation）*的意思是：一个事务的影响对与该事务并发执行的其它事务是不可见的；从事务的角度来看，好象事务是连续执行而非并行执行。尽管事务性资源管理器经常可以同时处理许多事务并提供隔离的假象，但有时隔离限制实际上要求把新事务延迟到现有事务完成后才开始。由于完成一个事务至少包括一个同步磁盘 I/O（写到事务日志），这就会把每秒的事务数限制到接近每秒的写磁盘次数，这对可伸缩性不利。

实际上，通常是充分放松隔离需求以允许更多的事务并发执行并使系统响应能够得到改善，使可伸缩性变得更强。几乎所有的数据库都支持标准隔离级别：读未提交的（Read Uncommitted）、读已提交的（Read Committed）、可重复的读（Repeatable Read） 和可串行化的（Serializable）。

不幸的是，为容器管理的事务管理隔离目前是在 J2EE 规范的范围之外。但是，许多 J2EE 容器，比如 IBM WebSphere 和 BEA WebLogic，将提供特定于容器的扩展，这些扩展允许您以每方法（per-method）为基础设置事务隔离级别，设置方法与在装配描述符中设置事务模式的方法相同。对于 bean 管理的事务，您可以通过 JDBC 或者其它资源管理器连接设置隔离级别。

为阐明隔离级别之间的差异，我们首先把几个并发危险分类 ― 这几种危险是当没有适当地隔离时一个事务可能会干涉另一个事务的情况。下列的所有这些危险都与这种情况（ 第二个事务已经启动后第一个事务变得对第二个事务 *可见*）的结果有关：

* **脏读（Dirty Read）**：当一个事务的中间（未提交的）结果对另一个事务可见时就会发生这种情况。
* **不可重复的读（Unrepeatable Read）**：当一个事务读取一个数据项，然后重新读取这个数据项并看到不同的值时就是发生了这种情况。
* **虚读（Phantom Read）**：当一个事务执行返回多个行的查询，稍后再次执行同一个查询并看到第一次执行该查询没出现的额外行时就是发生了这种情况。

四个标准隔离级别与这三个隔离危险相关，如表 2 所示。最低的隔离级别“读未提交的”并不能保护事务不被其它事务更改，但它的速度最快，因为它不需要争夺读锁。最高的隔离级别“可串行化的”与上面给出的隔离的定义相当；每个事务好象都与其它事务的影响完全隔离。

**表 2. 事务隔离级别**

|  |  |  |  |
| --- | --- | --- | --- |
| **隔离级别** | **脏读** | **不可重复的读** | **虚读** |
| 读未提交的 | 是 | 是 | 是 |
| 读已提交的 | 否 | 是 | 是 |
| 可重复的读 | 否 | 否 | 是 |
| 可串行化的 | 否 | 否 | 否 |

对于大多数数据库，缺省的隔离级别为“读已提交的”，这是个很好的缺省选择，因为它阻止事务在事务中的任何给定的点看到应用程序数据的不一致视图。“读已提交的”是一个很不错的隔离级别，用于大多数典型的短事务，比如获取报表数据或获取要显示给用户的数据的时候（多半是作为 Web 请求的结果），也用于将新数据插入到数据库的情况。

当您需要所有事务间有较高级别的一致性时，使用较高的隔离级别“可重复的读”和“可串行化的”比较合适，比如在清单 1 示例中，您希望从检查余额以确保有足够的资金到您实际取钱期间账户余额一直保持不变；这就要求至少要用“可重复的读”隔离级别。在数据一致性绝对重要的情况下，比如审核记帐数据库以确保一个帐户的所有借方金额和贷方金额的总数等于它目前的余额时，可能还需要防止创建新行。这种情况下就需要使用“可串行化的”隔离级别。

最低的隔离级别“读未提交的”很少使用。它适用于您只需要获得近似值，否则查询将导致您不希望的性能开销这种情况。当您想要估计一个变化很快的数量，如定单数或者今天所下定单的总金额（以美元为单位）时一般使用““读未提交的”。

因为隔离和可伸缩性之间实际是一种此消彼长的关系，所以您在为事务选择隔离级别时应该小心行事。选择太低的级别对数据比较危险。选择太高的级别可能对性能不利，尽管负载比较轻时可能不会这样。一般来说，数据一致性问题比性能问题更严重。如果拿不准，应该以小心为主，选择一个较高的隔离级别。这就引出了规则 5：

**规则 5：使用保证数据安全的最低隔离级别，但如果拿不准，请使用“可串行化的”。**

即使您打算刚开始时以小心为主并希望结果性能可以接受 ―（被称为“拒绝和祈祷（denial and prayer）”的性能管理技术 ― 很可能是最常用的性能策略，尽管大多数开发者都不承认这一点），在开发组件时考虑隔离需求也是有利的。您应该努力编写能够容忍级别较低但实用的隔离级别的事务，这样，当稍后性能成为问题时，自己就不会陷入困境。因为您需要知道方法正在做什么以及这个方法中隐藏了什么一致性假设来正确设置隔离级别，那么在开发期间仔细说明并发需求和假设，以便在装配应用程序时帮助作出正确的决定也不失为一个好主意。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp0514/#ibm-pcon)

**结束语**

本文中提供的许多指导可能看起来有点互相矛盾，因为象事务划分和隔离这种问题本来就是此消彼长的。我们正在努力平衡安全性（如果我们不关心安全性，那就压根不必用事务了）和我们用来提供安全限度的工具的性能开销。正确的平衡要依赖许多因素，包括与系统故障或当机时间相关的代价或损害以及组织的风险承受能力。