JBoss 企业级应用程序平台 4.3
JBoss Transactions 管理员指南

适用于 JBoss 企业级应用程序平台 4.3
版 4.3.0

Red Hat Documentation Group

JBoss 企业级应用程序平台 4.3 JBoss Transactions 管理员指南

适用于 JBoss 企业级应用程序平台 4.3
版 4.3.0

Red Hat Documentation Group

摘要

本书是面向 JBoss 企业级应用程序平台的《JBoss Transactions 管理员指南》

目录

序言

1. 文档约定

本手册使用几个约定来突出某些用词和短语以及信息的某些片段。

在 PDF 版本以及纸版中，本手册使用在 Liberation 字体套件中选出的字体。如果您在您的系统中安装了 Liberation 字体套件，它还可用于 HTML 版本。如果没有安装，则会显示可替换的类似字体。请注意：红帽企业 Linux 5 以及其后的版本默认包含 Liberation 字体套件。

1.1. 排版约定

我们使用四种排版约定突出特定用词和短语。这些约定及其使用环境如下。

单行粗体

用来突出系统输入，其中包括 shell 命令、文件名以及路径。还可用来突出按键以及组合键。例如：

> 要看到文件您当前工作目录中文件 **my_next_bestselling_novel** 的内容，请在 shell 提示符后输入 **cat my_next_bestselling_novel** 命令并按 **Enter** 键执行该命令。

以上内容包括一个文件名，一个 shell 命令以及一个按键，它们都以固定粗体形式出现，且全部与上下文有所区别。

按键组合与单独按键之间的区别是按键组合是使用加号将各个按键连在一起。例如：

> 按 **Enter** 执行该命令。
>
> 按 **Ctrl+Alt+F2** 切换到虚拟终端。

第一个示例突出的是要按的特定按键。第二个示例突出了按键组合：一组要同时按下的三个按键。

如果讨论的是源码、等级名称、方法、功能、变量名称以及在段落中提到的返回的数值，那么都会以上述形式出现，即**固定粗体**。例如：

> 与文件相关的等级包括用于文件系统的 **filesystem**、用于文件的 **file** 以及用于目录的 **dir**。每个等级都有其自身相关的权限。

比例粗体

这是指在系统中遇到的文字或者短语，其中包括应用程序名称、对话框文本、标记的按钮、复选框以及单选按钮标签、菜单标题以及子菜单标题。例如：

> 在主菜单条中选择**「系统」** → **「首选项」** → **「鼠标」**启动 **鼠标首选项**。在**「按钮」**标签中点击**「惯用左手鼠标」** 复选框并点击 **关闭**切换到主鼠标按钮从左向右（让鼠标适合左手使用）。
>
> 要在 **gedit** 文件中插入特殊字符，请在主菜单栏中选择**「应用程序」** → **「附件」** → **「字符映射表」**。接下来选择从 **Character Map** 菜单中选择**Search** → **「查找.....」**，在**「搜索」**字段输入字符名称并点击**「下一个」**按钮。此时会在**「字符映射表」**中突出您搜索的字符。双击突出的字符将其放在**「要复制的文本」**字段中，然后点击**「复制」**按钮。现在返回您的文档，并选择 **gedit** 菜单中的**「编辑」** → **「粘贴」**。

以上文本包括应用程序名称、系统范围菜单名称及项目、应用程序特定菜单名称以及按钮和 GUI 界面中的文本，所有都以比例粗体出现并与上下文区别。

固定粗斜体 或者 ***比例粗斜体***

无论固定粗体或者比例粗体，附加的斜体表示是可替换或者变量文本。斜体表示那些不直接输入的文本或者那些根据环境改变的文本。例如：

> 要使用 ssh 连接到远程机器，请在 shell 提示符后输入 **ssh *username@domain.name***。如果远程机器是 **example.com** 且您在该其机器中的用户名为 john，请输入 **ssh john@example.com**。
>
> **mount -o remount *file-system*** 命令会重新挂载命名的文件系统。例如：要重新挂载 **/home** 文件系统，则命令为 **mount -o remount /home**。
>
> 要查看目前安装的软件包版本，请使用 **rpm -q *package*** 命令。它会返回以下结果：***package-version-release***。

请注意上述使用黑斜体的文字 -- username、domain.name、file-system、package、version 和 release。每个字都是一个站位符，可用作您执行命令时输入的文本，也可作为该系统显示的文本。

不考虑工作中显示标题的标准用法，斜体表示第一次使用某个新且重要的用语。例如：

> Publican 是一个 *DocBook* 发布系统。

1.2. 抬升式引用约定

终端输出和源代码列表要与周围文本明显分开。

将发送到终端的输出设定为 **Mono-spaced Roman** 并显示为：

```
books        Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

源码列表也设为 **Mono-spaced Roman**，但添加下面突出的语法：

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
   public static void main(String args[])
       throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object         ref    = iniCtx.lookup("EchoBean");
      EchoHome       home   = (EchoHome) ref;
      Echo           echo   = home.create();

      System.out.println("Created Echo");

      System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
   }
}
```

1.3. 备注及警告

最后，我们使用三种视觉形式来突出那些可能被忽视的信息。

注意

备注是对手头任务的提示、捷径或者备选的解决方法。忽略提示不会造成负面后果，但您可能会错过一个更省事的诀窍。

重要

重要框中的内容是那些容易错过的事情：配置更改只可用于当前会话，或者在应用更新前要重启的服务。忽略'重要'框中的内容不会造成数据丢失但可能会让您抓狂。

警告

警告是不应被忽略的。忽略警告信息很可能导致数据丢失。

2. 反馈

如果你在本指南里发现了印刷错误，或者你有改进该手册的建议，我们希望听到你的声音！请提交报告到 JIRA 并指明：产品：JBoss 企业级应用程序平台, 版本：***<version>***，组件：*Doc*。如果是改进本文档的建议，请尽量具体化；如果是其他错误，请指出章节号以及具体的内容，这样我们就可以尽快改正。

3. 本书包含的内容

《管理员指南》包含如何管理 JBoss Transactions API 4.2.3 的信息。

4. 读者

本书面向负责 JBoss Transactions API 4.2.3 安装管理的工程师。

5. 预备条件

为了管理 JBoss Transactions，你首先必须理解它的很多事务方面的功能依赖于 ArjunaCore。因此，在管理 JBoss Transactions 之前，先阅读《ArjunaCore 管理员指南》是很重要的。

6. 本书的结构

本指南包含下面的章节：

1. 第一章，JBoss Transactions 管理：描述如何使用不同的 JTA 实现来管理 JBoss Transactions：纯 local 或 remote 方式（允许分布式事务）。

7. 其他文档

除了本指南外， JBoss Transactions API 4.2.3 的文档集里还有如下的文档：

1. JBoss Transactions API 4.2.3 发行注记：提供 JBoss Transactions API 4.2.3 的最新信息。
2. JBoss Transactions API 4.2.3 安装指南：提供 JBoss Transactions API 4.2.3 的安装说明。

3. JBoss Transactions API 4.2.3 程序员指南：指导如何编写相关应用程序。

第 1 章 简介

为了确保系统的正常运行，你需要对 JBoss Transactions 进行一点持续的管理。下面是一些需要注意的地方：

1. 目前 JBoss TA 系统的实现没有提供安全性或对数据的保护。存储在 JBoss Transactions 的 object store 里的对象通常为运行创建这些对象的应用程序的用户所有。Object Store 和 Object Manager 机制甚至没有强制 Unix/Windows 所提供的有限的保护。对这些对象的读和写并不需要检查用户或组的 ID。
2. 除非调用 StateManager.destroy 方法或某个应用程序显性地进行删除，在 Object Store 里创建的持久性对象会永远存在。这意味着 Object Store 会逐渐地积累垃圾（特别是在程序开发和测试阶段）。目前我们还没有自动的垃圾回收机制。而且，我们也还没解决 dangling 引用的问题：某个持久性对象 A，可能在其磁盘上的 passive representation 里存储了另外一个持久性对象 B 的 UID。应用程序有可能删除了 B，即使 A 仍然具有对其的引用。之后 A 被激活并试图访问 B 时，就会出现一个运行错误。
3. 目前没有对类结构改变时对象或数据库重配置的版本控制的支持。这是我们还没解决的一个复杂的研究领域。目前，如果你修改了持久性对象的类的定义，你需要完全负责确保 Object Store 里的现存实例转换成新的结构。JBoss Transactions 既不能检测也不能改正通过新的操作版本对旧的对象状态的引用，反之亦然。
4. 对于事务服务来说，Object store 的管理至关重要。

1.1. 管理 ObjectStore

在事务访问安装里，只要有事务被创建或是用于 Java 的 Transactional Object 被使用，object store 就会被更新。在无故障的环境里，唯一应该驻留在 object store 里的对象状态时那些代表用 Java API 然而，如果发生了故障，事务日志可能保留在 object store 里，直到故障恢复机制已经恢复了它们所代表的事务。因此，删除 object store 的内容需要很小心和谨慎，这很重要，因为这会导致无法解决可疑的事务。此外，如果多个用户共享相同的 object store，他们也需明白这一点。他们不能假定 object store 里的内容是独占资源而简单地进行删除。

1.2. JBoss Transactions 的运行信息

组成 JBoss Transactions 的每个模块都拥有一个名为 info 的类。这些类都提供单一的 toString 方法来返回代表该模块的配置信息的 XML 文档。例如：

```
<module-info name="arjuna">
 <source-identifier>unknown</source-identifier>
 <build-information>
    Arjuna Technologies [mlittle] (Windows 2000 5.0)
 </build-information>
 <version>unknown</version>
 <date>2002/06/15 04:06 PM</date>
 <notes></notes>
 <configuration>
  <properties-file dir="null">arjuna.properties</properties-file>
  <object-store-root>null</object-store-root>
 </configuration>
</module-info>
```

1.3. 失效切换管理

JBoss Transactions 的失效切换子系统将确保在即使应用程序进程或主机崩溃或断开了网络连接的情况下，事务的结果都可以持续地应用到事务所影响的资源里。在主机（系统）崩溃或网络故障时，失效切换将直

到系统或网络恢复时才进行，但原来的应用程序不需要重启 — 切换的责任委托给了 Recovery Manager 进程（请参考下面内容）。失效切换要求和事务和资源相关的信息：这些信息保存在作为 ObjectStore 一部分的 ActionStore 里。

1. 如果 ObjectStore 被销毁或修改，失效切换有可能无法进行。

直到失效切换过程完成之前，故障发生时正进行的事务所影响的资源都可能无法访问。对于数据库资源而言，可能出现表或行被“in-doubt transactions”锁定。对于用于 Java 资源的 TransactionalObjects，试图激活 Transactional Object（尝试获取锁）将失败。

1.3.1. Recovery Manager

JBoss Transactions 的失效切换（failure recovery）子系统要求为每个 ObjectStore（通常是网络上的每个运行 JBoss Transactions 应用程序的节点）运行独立的 Recovery Manager 进程。RecoveryManager 文件位于 **`com.arjuna.ats.arjuna.recovery.RecoveryManager`** 包里的 arjunacore jar 中。要启动 Recovery Manager，你可以使用下面的命令：

```
java com.arjuna.ats.arjuna.recovery.RecoveryManager
```

如果使用了 **`-test`** 标签，它将在初始化时显示“Ready”信息，也就是：

```
java com.arjuna.ats.arjuna.recovery.RecoveryManager -test
```

1.3.2. 配置 Recovery Manager

RecoveryManager 读取 **`arjuna.properties`** 文件里定义的属性，然后读取相同目录下的属性文件 **`RecoveryManager.properties`**。RecoveryManager 属性文件里的条目将覆盖 TransactionService 属性文件里相同的条目。大多数条目都是 Recovery Manager 所专有的。

本发行版本提供一个缺省的 **`RecoveryManager.properties`** 版本 — 它无需修改就可以使用，除了一些调试跟踪的字段（请参考下面的 Output）。本节剩下的内容将讨论将属性设置为其他值所相关的问题（按照在缺省版本里出现的顺序）。

1.3.2.1. Output

很可能安装将希望从 RecoveryManager 里获得某种形式的输出来提供发生恢复的记录。RecoveryManager 使用 Arjuna Common Logging Framework (CLF) 提供的日志跟踪机制，它提供了一个隐藏了现有的 logging API 如 Jakarta log4j 或 JDK 1.4 logging API 之间的差异的高级别的接口。CLF 通过 Apache Commons Logging 框架间接处理日志，其配置假定存在于该框架内。

用 CLF 应用程序来调用 logger 对象的日志调用。Logger 可以使用日志级别来决定具体的日志信息。每个日志信息都有一个相关的日志级别来表示重要和紧急程度。日志级别可以是 DEBUG、INFO、WARN、ERROR 和 FATAL。按整形值的顺序排列为：DEBUG < INFO < WARN < ERROR < FATAL。

CLF 提供按照定义的颗粒度过滤日志消息的扩展。这就是说，当 DEBUG 级别的日志消息被提供给 logger时，你可以指定其他的条件来决定是否启用这个日志消息。

1. 只有启用了 DEBUG 级别且日志请求由应用程序专有的调试颗粒度执行时，才能应用这些条件。

被启用后，日志消息根据 3 个变量进行有条件地过滤：

1. 调试级别（Debugging level）：DEBUG 级别的日志请求产生的地方，如构造函数或基本方法。
2. 可见级别（Visibility level）：产生调试信息的构造函数、方法等的可见性。
3. Facility code：例如产生调试信息的包或子模块，如 ObjectStore。

根据这些变量，CLF 定义了 3 个接口。特定的产品可以按照需要实现自己的类。JBoss Transactions 使用 CLF 提供的缺省 Debugging level 和 Visibility level，但它定义自己的 Facility Code。JBoss Transactions

使用分配给 logger 对象的缺省的级别（DEBUG）。然而，它使用 finer debugging 特征来禁用或启用调试消息。JBoss Transactions 使用的 finer debugging 值有：

Debugging level – JBoss Transactions 使用 **`com.arjuna.common.util.logging.DebugLevel`** 类里定义的缺省值

1. NO_DEBUGGING: No diagnostics. 这个值分配的 Logger 对象丢弃所有的调试请求。
2. FULL_DEBUGGING: Full diagnostics. 如果 facility code 和 the visibility level 符合 logger 所允许的值，这个值分配的 Logger 对象允许所有的调试请求。

其他的调试值有：

1. CONSTRUCTORS: Diagnostics from constructors.
2. DESTRUCTORS: Diagnostics from finalizers.
3. CONSTRUCT_AND_DESTRUCT: Diagnostics from constructors and finalizers.
4. FUNCTIONS: Diagnostics from functions.
5. OPERATORS: Diagnostics from operators, such as equals.
6. FUNCS_AND_OPS: Diagnostics from functions and operations.
7. ALL_NON_TRIVIAL: Diagnostics from all non-trivial operations.
8. TRIVIAL_FUNCS: Diagnostics from trivial functions.
9. TRIVIAL_OPERATORS: Diagnostics from trivial operations, and operators.
10. ALL_TRIVIAL: Diagnostics from all trivial operations.

Visibility level – JBoss Transactions 使用 **`com.arjuna.common.util.logging.VisibilityLevel`** 类里定义的缺省值

1. VIS_NONE: No Diagnostic
2. VIS_PRIVATE : only from private methods.
3. VIS_PROTECTED only from protected methods.
4. VIS_PUBLIC only from public methods.
5. VIS_PACKAGE only from package methods.
6. VIS_ALL: Full Diagnostic

Facility Code – JBoss Transactions 使用 **`com.arjuna.common.util.logging.VisibilityLevel`** 类定义的下列值

1. FAC_ATOMIC_ACTION = 0x0000001 (atomic action core module).
2. FAC_BUFFER_MAN = 0x00000004 (state management (buffer) classes).
3. FAC_ABSTRACT_REC = 0x00000008 (abstract records).
4. FAC_OBJECT_STORE = 0x00000010 (object store implementations).
5. FAC_STATE_MAN = 0x00000020 (state management and StateManager).
6. FAC_SHMEM = 0x00000040 (shared memory implementation classes).
7. FAC_GENERAL = 0x00000080 (general classes).
8. FAC_CRASH_RECOVERY = 0x00000800 (detailed trace of crash recovery module and classes).
9. FAC_THREADING = 0x00002000 (threading classes).
10. AC_JDBC = 0x00008000 (JDBC 1.0 and 2.0 support).
11. FAC_RECOVERY_NORMAL = 0x00040000 (normal output for crash recovery manager).

为了保证合适的输出，我们有必要在 **`CommonLogging.xml`** 文件里对一些调试属性进行更详细的设置，启用 JBoss Transactions 模块登记的日志信息。

描述启动和 RecoveryManager 的定期行为的消息使用 INFO 级别的输出。如果需要其他调试跟踪信息，你应该设置更详细的调试级别。例如，下面 CommonLogging.xml 里的配置，启用了和 Crash Recovery 协议相关以及 JBoss Transactions 产生的所有调试信息。

```
<!-- Common logging related properties. -->
<property
  name="com.arjuna.common.util.logging.DebugLevel"
  value="0x00000000"/>
<propertyname="com.arjuna.common.util.logging.FacilityLevel"
  value="0xffffffff"/>
<propertyname="com.arjuna.common.util.logging.VisibilityLevel"
  value="0xffffffff"/>
```

1. 这里提供了两个 logger 对象，一个管理 I18N 消息而另外一个则处理其他消息。

把常规恢复消息设置为 INFO 级别允许 RecoveryManager 产生适度的报告。如果没有发生任何事情，它仅仅报告模块的定期登录。要禁用 Recovery Manager 提供的 INFO 消息，调试级别可以设为更高的 ERROR 级别。这表示 RecoveryManager 将只产生错误、警告或致命错误信息。

1.3.3. 定期恢复

RecoveryManager 扫描 ObjectStore 以及信息的其他位置，寻找需要恢复的事务和资源。这个扫描和恢复过程由恢复模块来执行（实现 com.arjuna.ats.arjuna.recovery.RecoveryModule 接口的类的实例），每个都负责特定类别的事务和资源。恢复模块使用 RecoveryManager 属性文件里找到的属性动态地进行加载。

这个接口有两个方法：**`periodicWorkFirstPass`** 和 **`periodicWorkSecondPass`**。根据 **`com.arjuna.ats.arjuna.recovery.periodicRecoveryPeriod`** 属性定义的间隔，RecoveryManager 件将在每个属性上调用第一个 pass 方法，然后等待一段短暂的时间（由 **`com.arjuna.ats.arjuna.recovery.recoveryBackoffPeriod`** 属性定义）。通常，在第一个 pass 方法里，模块通过扫描（例如 ObjectStore 的相关部分）来发现可疑的事务和资源（也就是提交过程进行一部分）。在第二个 pass 方法里，如果任何相同的事务或资源仍然可疑，那么原来的应用程序就有可能已经崩溃，这些事务和资源就需要进行恢复。

RecoveryManager 试图恢复仍然在原来进程里进行的事务有可能会破坏一致性。因此，恢复模块使用一个机制（在 **`com.arjuna.ats.arjuna.recovery.TransactionStatusManager`** 包里实现）来检查原来的进程是否仍然活动，且事务是否仍在进行。RecoveryManager 只在原始进程已经结束时才进行恢复，如果进程仍然活动，则在事务完成后进行。（如果服务器进程或主机崩溃了，但初始事务的基础仍然活动，事务也会完成并通常产生一个警告。RecoveryManager 也负责恢复这样的事务。）

设置合适的时间间隔显然很重要。重复时间将是 **`periodicRecoveryPeriod`**、**`recoveryBackoffPeriod`** 和所有恢复模块扫描 ObjectStore 并试图恢复可疑事务所需时间的总和。恢复时间可能包括试图和已经崩溃或无法访问的进程或主机通信所需的连接超时时间（这是为什么在恢复系统里有机制避免试图恢复相同的事务）。总共的重复时间将影响在发生故障后资源保持无法访问状态的时间 — **`periodicRecoveryPeriod`** 应该相应地进行设置（缺省是 120 秒）。**`recoveryBackoffPeriod`** 可以相对较短（缺省是 10 秒） — 它的目的主要是减少事务作为恢复备选对象的次数（这需要访问原始进程，检查事务是否仍在进行）。

1. JBoss Transactions 之前的版本里没有 contact 机制，你得把 backoff period 设置为足够久来避免缓存事务冲突。从 3.0 版本后，这不再是一个问题。

JBoss Transactions 提供了两个恢复模块（**`com.arjuna.ats.arjuna.recovery.RecoveryModule`** 接口的实现），它们支持不同的事务恢复（包括 JDBC 恢复）。高级用户有可能创建自己的恢复模块并向 Recovery Manager 注册。恢复模块使用以“**`com.arjuna.ats.arjuna.recovery.RecoveryExtension`**”开始的属性向 Recovery Manager 注册。它们将按照属性名称的顺序调用其定期恢复的 pass 方法（但请注意在定期恢复的 pass 方法运行时，

可能会发生应用程序进程失效）。缺省的 Recovery Extension 设置是：

```
com.arjuna.ats.arjuna.recovery.recoveryExtension1 =
  com.arjuna.ats.internal.ts.arjuna.recovery.AtomicActionRecoveryModule

com.arjuna.ats.arjuna.recovery.recoveryExtension2 =
  com.arjuna.ats.txoj.recovery.TORecoveryModule
```

1.3.4. 删除过期的条目

恢复子系统的操作将导致 ObjectStore 创建的某些条目在常规情况下不会被删除。RecoveryManager 里有一个工具可以扫描和删除那些很旧的的条目。**`com.arjuna.ats.arjuna.recovery.ExpiryScanner`** 接口的实现执行这种扫描和删除。这个实现通过把类名作为以 **`com.arjuna.ats.arjuna.recovery.expiryScanner`** 开始的属性的值来加载。RecoveryManager 按照属性 **`com.arjuna.ats.arjuna.recovery.expiryScanInterval`** 指定的时间间隔调用每个加载的 Expiry Scanner 实现的 **`scan()`** 方法。这个值以小时为单位 – 缺省为 12。零值的 **`expiryScanInterval`** 将忽略任何过期扫描。如果这个值为正值，RecoveryManager 启动时将执行第一次扫描；如果为负值，第一次扫描将直到第一个间隔（使用绝对值）后才发生。

扫描过期的条目类型包括：

TransactionStatusManager 条目：使用 JBoss Transactions 的每个应用程序进程创建的条目 – 它们包含允许 RecoveryManager 决定初始化事务的进程是否仍然活动、以及事务状态的信息。过期时间由属性 **`com.arjuna.ats.arjuna.recovery.transactionStatusManagerExpiryTime`**（以小时为单位 – 缺省为 12，0 表示永不过期）设置。过期时间应该大于任何单个 JBoss Transactions-using 进程的生存时间。

Expiry Scanner 属性是：

```
com.arjuna.ats.arjuna.recovery.expiryScannerTransactionStatusManager =
  com.arjuna.ats.internal.ts.arjuna.recovery.ExpiredTransactionStatusManagerScanner
```

1.4. 错误和异常

本节将涵盖在事务行应用程序运行时可能抛出或报告的错误和异常类型，并给出可能的原因。

1. NO_MEMORY：应用程序已耗尽内存（抛出 **`OutOfMemoryError`**）且 JBoss Transactions 在重新抛出这个异常时已经试图进行清理（通过运行 garbage collector）。这可能是一个暂时的问题，重试应该能够成功。
2. com.arjuna.ats.arjuna.exceptions.FatalError：表示事务系统必须关闭的错误已经发生。在抛出这个错误之前，服务将确保所有的事务都进行了回滚。如果捕获了这个异常，应用程序应该清理并退出。如果进行进一步尝试，可能会破坏应用程序的一致性。
3. com.arjuna.ats.arjuna.exceptions.LicenceError：尝试以和当前许可证不一致的方式使用事务服务。事务服务将不会允许对现有的或新的事务进行进一步的行动。
4. com.arjuna.ats.arjuna.exceptions.ObjectStoreError：事务服务试图使用 object store 时产生了错误。进一步的行动不被允许。
5. 在失效切换的正常执行过程中，可能出现关于状态访问的 Object store 的警告。这是因为在相同的事务上执行多个并行的失效切换。它可以被忽略而无安全隐患。

1.5. 选择 JTA 实现

目前可通过相同的接口访问两个 JTA 实现的变种。它们是：

1. 纯本地的 JTA，它只允许执行非分布式的 JTA 事务。JBoss Transactions 产品里只有这个版本可用。
2. 远程的，基于 CORBA 的 JTA，它允许执行分布式的 JTA 事务。它只在 ArjunaJTS 里可用且需要对 CORBA ORB 的支持。
3. 这两种实现都和 JBoss Transactions 提供的事务性的 JDBC 驱动完全兼容。

要选择本地的 JTA 实现，你必须执行下面的步骤：

1. 确保属性 **`com.arjuna.ats.jta.jtaTMImplementation`** 被设置为 **`com.arjuna.ats.internal.jta.transaction.arjunacore.TransactionManagerImple`**。
2. 确保属性 **`com.arjuna.ats.jta.jtaUTImplementation`** 被设置为 **`com.arjuna.ats.internal.jta.transaction.arjunacore.UserTransactionImple`**。
3. 这些设置都是这些属性的缺省值，如果使用本地实现的话，你不需要进行指定。

修订历史记录

修订 4.3.0-2.400	**2013-10-31**	**Rüdiger Landmann**
Rebuild with publican 4.0.0		
修订 4.3.0-2	**2012-07-18**	**Anthony Towns**
Rebuild for Publican 3.0		
修订 4.3-0	**Wed Sep 15 2010**	